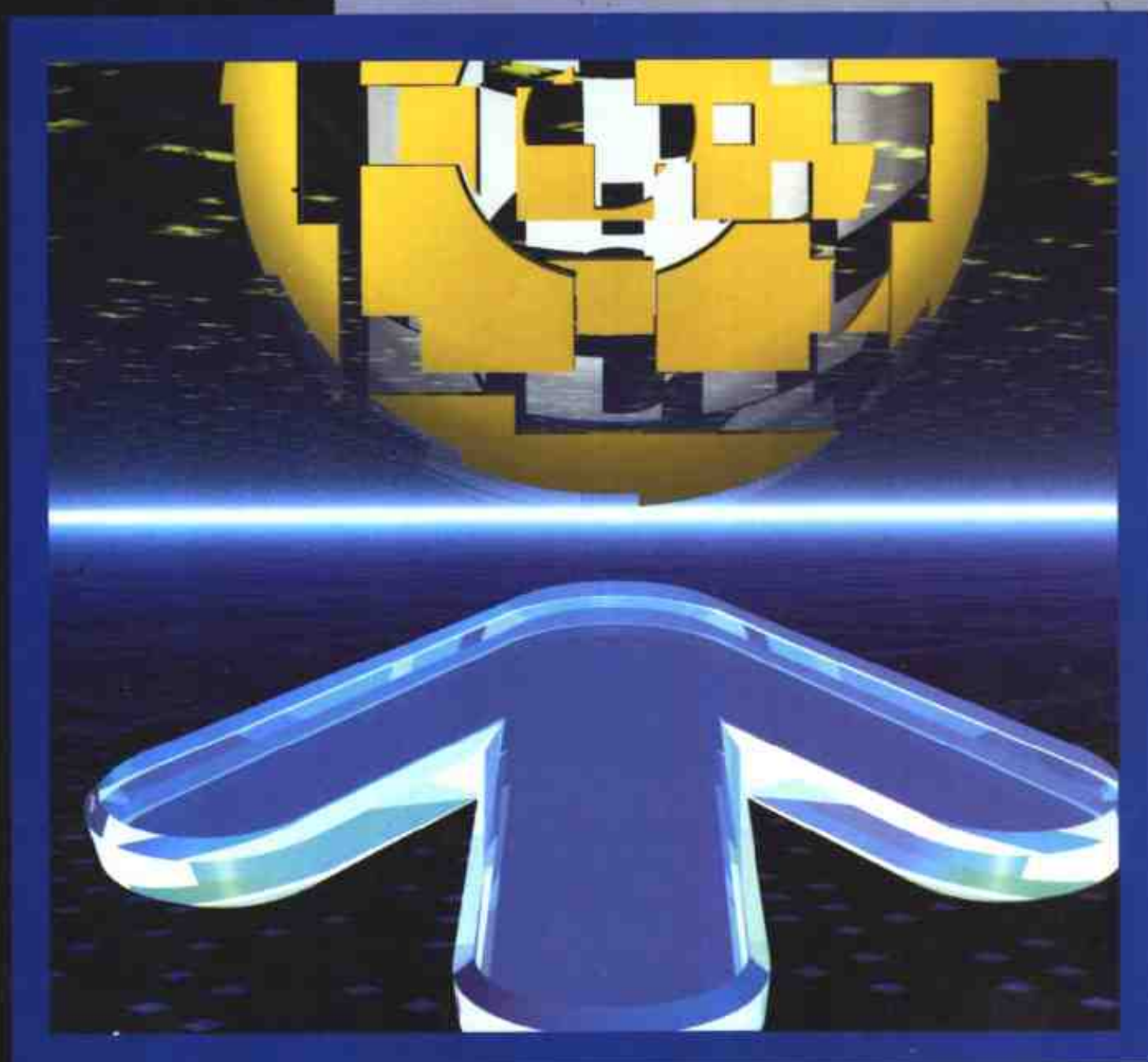


C# Network Programming



C#网络应用编程

C#简化了网络编程技术。如果把它与本书中的指导相结合，你会发现编写网络应用程序比过去任何时候都更加容易和快捷。

[美] Richard Blum 著

高春蓉 谷 宇 阎 隽 等译
李双庆 审校



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

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

C#网络应用编程



当C#新手刚刚接触网络编程的时候，本书帮助他们从基本的网络编程方法开始，包括这种语言的网路类、Winsock接口和DNS解决方案。在这里花费时间的多少可根据读者自己的需要而定，然后就要深入探索网络层的核心课题。读者将学习通过TCP进行套接字的连接，通过UDP进行“无连接”的连接。读者会发现，对于编程中困难麻烦的事务，C#会给你带来多么大的帮助，例如异步套接字编程技术、多线程和组播技术。

网络层技术只是一些终端的工具，所以在本书中提供了许多详细的应用层编程的范例，这些范例展示了在真实的网络环境中怎样操作真实的协议，建立和实现各种应用程序，怎样用SNMP管理网络设备、用SMTP与远程邮件服务器通信和用HTTP使用户的应用程序具备Web功能，使用C#本身固有的类查询和修改活动目录条目。

所有这些技术是一种极大的优势，把用户的C#网络编程技术推动到更高的境界。例如，读者将学会两种在网络上共享应用程序方法的方法：网络服务方法和远程技术。读者还将掌握C#.NET固有的安全特性——保护用户编程项目的特性。

作者简介：

Richard Blum在美国国防部担任了14年网络和系统管理员的工作，负责一个由许多工作站和服务器组成的大型网络的技术支持工作。在空闲时间，还帮助维护他女儿学校的一个小型网络。他在教区的礼拜乐队中演奏电子低音。他持有Purdue大学颁发的电气工程师学士学位和管理信息系统硕士学位。

本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书。

ISBN 7-5053-8617-4



9 787505 386174 >

责任编辑：陈宇



ISBN 7-5053-8617-4/TP·4997 定价：56.00元

C# Network Programming

C#网络应用编程

〔美〕 Richard Blum 著

高春蓉 谷 宇 阎 隽 等译

李双庆 审校

電子工業出版社

Publishing House of Electronics Industry

北京 • BEIJING

内 容 提 要

本书是一本全面介绍C#编程语言的书。书中详细介绍了各种C#网络类和方法，用大量的范例程序演示这些类和方法在程序中的运用，帮助读者在自己的实际工作中编写出更具专业水准的网络程序。作者采用对比的方法，分析了多种编程技术的优缺点，使读者更能体会到C#语言给编程者带来的方便和快捷。

本书特别适合对C#语言网络编程感兴趣的程序员们阅读，已经熟悉其他语言（例如C、C++或者Java）的读者，阅读本书后会感到用C#编写网络程序多么容易。没有编程经验的读者也会从本书受到启发，因为作者对网络编程基本方法和思路的叙述，会使初学者更快地成为一名C#语言的行家。



Copyright©2003 SYBEX Inc., 1151 Marina Village Parkway, Alameda, CA 94501. World rights reserved. No part of this publication may be stored in a retrieval system, transmitted, or reproduced in any way, including but not limited to photocopy, photograph, magnetic or other record, without the prior agreement and written permission of the publisher.

本书英文版由美国SYBEX公司出版，SYBEX公司已将中文版独家版权授予中国电子工业出版社及北京美迪亚电子信息有限公司。未经许可，不得以任何形式和手段复制或抄袭本书内容。

版权贸易合同登记号：01-2002-6041

图书在版编目 (CIP) 数据

C#网络应用编程/ (美) 布劳姆 (Blum, R.) 著; 高春蓉等译. —北京: 电子工业出版社, 2003.5

书名原文: C# Network Programming

ISBN 7-5053-8617-4

I. C… II. ①布… ②高… III. C语言—程序设计 IV. TP312

中国版本图书馆CIP数据核字 (2003) 第023159号

责任编辑: 陈 宇

印 刷: 北京大学颖华印刷厂

出版发行: 电子工业出版社 <http://www.phei.com.cn>

北京市海淀区万寿路173信箱 邮编: 100036

北京市海淀区翠微东里甲2号 邮编: 100036

经 销: 各地新华书店

开 本: 787×1092 1/16 印张: 33.75 字数: 860 千字

版 次: 2003年5月第1版 2003年5月第1次印刷

定 价: 56.00元

凡购买电子工业出版社的图书，如有缺损问题，请向购买书店调换，若书店售缺，请与本社发行部联系。联系电话: (010) 68279077

谨以此书献给修女Marie Imelda, C.S.C, 她曾在20世纪70年代后期竭尽全力、孜孜不倦地教会一群愚笨的高中生编写计算机程序。谢谢, Marie修女。“不要依赖自己的理解, 而要用你整个的心灵相信我主, 用自己的方式感谢主的恩赐, 他将使你前面的道路变得笔直而平坦。”

圣经.旧约 3: 5-6 (NIV)

致 谢

首先, 所有的荣誉、辉煌和赞颂都属于上帝, 他使一切事情都成为可能, 他给了我们永恒的生命。

我要感谢Sybex公司所有优秀员工的大力帮助, 感谢他们细心的指导和精深的专业技巧。感谢采集编辑Denise Santoro Lincoln给我提供了编写本书的机会。还要感谢开发编辑Carol Henry对本书的指导工作, 她的帮助使书中的许多段落更有意义。

非常感谢拷贝编辑Sally Engelfried, 他把我糟糕的叙述变成了完善的句子。也要感谢技术编辑Dan Frumin, 他认真负责地纠正了书中的程序错误并指出我技术上的问题。还要对Waterside Productions的Carole McClendon表示谢意, 她为我安排了本书的出版。

最后, 要感谢我的父母, Mike Blum和Joyce Blum, 对我写书的全力支持。感谢我的妻子Barbara, 女儿Katie Jane和Jessica, 感谢她们对我的信任、爱和理解, 特别是在我写这本书的时候。

译 者 的 话

随着计算机技术的发展，特别是因特网的推广应用，网络技术已深入到我们日常生活和工作的各个方面。网络编程，这个过去只有少数高级程序员才能做到的事情，现在已经由越来越多的普通计算机用户来完成了。早期是在UNIX环境下用C语言编写网络程序的，后来是用Java语言编写的。用这些语言编写网络程序难度大、费时多，程序的运行速度慢。C#语言大大地简化了网络编程技术，通过TCP进行套接字的连接，通过UDP进行“无连接”的连接，通过使用异步套接字技术、多线程和多点传送技术，C#语言使过去困难的网络编程工作现在可以轻松容易地完成。

本书的作者Richard Blum在美国国防部做了14年网络和系统管理员的工作，他丰富的网络管理经验和编程实践给编写本书打下了坚实的基础。书中大量的程序实例使读者可以学到更多网络编程的实用知识。本书语言简炼、层次分明，全面系统地介绍了C#语言编程技术。

本书的第1章~第4章由李双庆翻译，第5章~第7章由高春蓉翻译，第8章~第12章由谷宇翻译，第13章~第17章由阎隽翻译，全书由李双庆审校。在本书的翻译过程中，刘文林、张亚宁、刘贡生、王炎、李苏云、许柏庆、马文清、彭苏鲁、卢刚、王向荣、刘杰等也参加了部分翻译、校对工作，在此对他们的支持和帮助表示诚挚的感谢。书中翻译的错误和不当之处，欢迎读者批评指正。

简介

在过去的20年中，网络（和网络编程）已经走过了一段很长的路程。在早期（20世纪80年代）的网络计算机技术中，网络编程是高级程序员的事情，他一般（大多数情况下）是在UNIX环境中用C程序设计语言编写应用程序。可是现在，从大型公司到家庭用户，网络无处不在。这么多的计算机通过网络连接在一起，网络组件应用程序是必不可少的。现有的应用程序必须结合网络功能才能市场中拥有竞争力，必须要给应用程序加上网络通信能力。从孩子们的游戏程序到先进的公司数据库系统，处处都要用到网络程序。

网络编程技术一直是微软Windows操作系统的重要特点。不过很遗憾，要想利用Windows中网络编程的特点必须要了解高级的C或C++编程的概念。可是现在，.NET框架语言简化了把网络特性加到应用程序中的工作。.NET库提供了许多可以集成网络编程技术的网络类。

我作为一名网络管理员，曾经用C和Java语言为Windows和UNIX平台编写过许多网络程序。当前，网络管理和安全性方面的要求使得与网络设备通信和跟踪网络上的工作站变得非常重要。在C套接字API结构中（特别是在WinSock中）工作时，要想快速地写出网络代码是很困难的。由于Java应用程序处理的速度太慢且不受Windows支持，运行Java应用程序通常是很痛苦的。

C#语言解决了许多网络编程的难题，因为我们可以用C#类迅速地生成和开发出网络应用程序。将编写图形代码的C# Forms库与编写网络代码的C# Socket库相结合，使得创建专业水平的网络应用程序变得十分简单。有了C#网络类，过去需要一天时间编写的程序现在最多只需要一个小时就完成了。

本书适合哪些读者

很显然，本书对于那些对网络编程感兴趣的C#程序员是特别有用的，书中详细介绍了各种网络C#类，并加上了许多范例程序，帮助读者在自己的工作中实现这些类。如果读者过去从来没有写过程序，我也在书中叙述了网络编程最基本的概念和思路，还说明了在网络设备之间传送数据的最常用的方法。

读者也许已经熟悉用其他语言（例如C、C++或者Java）编写网络程序，在这种情况下，你会高兴地看到用C#语言编写网络程序是多么容易。

如果读者对C#语言感到陌生，本书在第1章中讲述了创建和编译C#程序的基本方法。你也许想跳过本书第一部分的其他各章，这些章节介绍了网络编程的基本方法，深入探讨了各种特殊的C#网络编程类。

本书内容是怎样安排的

本书分为三个部分，涉及到网络编程的各个方面。

第一部分：网络编程基础

前四章是为那些刚刚开始学习网络编程的程序员们写的，他们可能希望了解网络编程技术发展的历史背景，想知道网络编程需要了解哪些信息。

第1章“C#语言”给初学C#语言的读者提供了一些C#的基础知识，例如，开发工作使用哪种C#包和怎样编译C#程序等。

第2章“IP编程基础”介绍了网络编程技术是怎样通过WinSock接口从UNIX领域发展到Windows领域的，并说明.NET是怎样跨网络资源利用WinSock接口的。

第3章“C#网络编程的类”简要介绍了整个C#网络库，并展示了类的几种基本格式。

第4章“DNS和C#”是介绍性的一章，告诉网络新手们DNS是怎样解决主机地址问题的，并介绍怎样使用C# DNS类。

第二部分：网络层编程

第二部分的几章是本书的核心，讲述了网络编程技术的重要课题。每一章都讨论一个创建C#网络程序的主要课题。

第5章“面向连接的套接字”一开始讨论了利用TCP的流编程技术，除了讨论流编程技术中使用标准C# Socket类之外，还讨论了存在的共同缺陷，以帮助读者建立可以在实际网络中工作的流程序。

第6章“无连接套接字”讨论了怎样用Socket类建立UDP应用程序。这一章除了告诉读者怎样建立UDP应用程序之外，还讨论了UDP程序设计方法的缺点和不足，并用创建的应用程序范例说明在真实网络环境中可能遇到的问题。

第7章“C#套接字的助手类”讨论了C# TcpClient、TcpListener和UdpClient类，这些都是.NET中特殊的类，可以帮助程序员用最小的努力创建网络程序。这一章还讨论了通过网络发送不同类型数据的基本方法。

第8章“异步套接字编程”讨论了在网络编程领域中使用的异步编程技术（在Windows程序中流行的）。

第9章“使用线程”提供了在网络程序中使用多线程应用程序技术所需要的信息。该技术一般用于必须同时为多个客户机提供服务的服务器应用程序。

第10章“IP组播”描述了怎样用广播和组播的方法把信息包发送到多个客户机，以便削减网络带宽。

第三部分：应用层编程实例

本书的最后一部分描述了一些特殊的网络应用程序，并说明怎样用C#网络类实现这些应用程序。

第11章“ICMP”说明怎样用C# raw套接字实现特定协议的应用程序。在这一章中，向读者展示了在C#网络程序上下文中普通ping和traceroute程序的作用。

第12章“SNMP”介绍了怎样用C#写网络管理应用程序。SNMP（简单网络管理协议）允许用户与网络上的许多设备通信，以便搜寻和获取网络统计数据。在这一章列举了一些读取销售商的MIB（管理数据库）表格的范例，可以创建一个C#应用程序，从网络设备获得

MIB数据。

第13章“SMTP”介绍了C#电子邮件类，列举了用这些类把使用SMTP（简单邮件传输协议）的邮件传送到远程邮件服务器的范例。本章还举了一个用其他邮件协议（例如POP3）的例子。

第14章“HTTP”介绍了C#的web类，讲述了怎样用这些类创建具有Web功能的C#应用程序。还介绍了.NET的网络服务和讨论了怎样用网络服务把用户的应用程序宿主在一个IIS服务器上。

第15章“活动目录”介绍了与微软活动目录服务器有关的各种C#类。举了一些例子说明怎样在活动目录中查询、修改、添加和删除条目。

第16章“远程技术”讨论了.NET的远程概念，它允许应用程序与网络上的客户机共享方法。本章举例演示了创建远程服务器和客户机的方法。

第17章“网络安全”描述了.NET框架是怎样处理网络安全问题的，介绍怎样在应用程序中使用加密技术保护网络数据的安全。

不断更新软件版本

本书所有的范例都是用.NET框架1.0版软件包创建和编译的。每个例子用任何一个微软Visual Studio软件包（包括Visual C#）都很容易进行编译。

编写本书的时候，.NET框架的版本是1.0版和Service Pack 1版。微软公司.NET框架Web站点的网址是<http://www.microsoft.com/netframework>，读者可以在该网站找到各种关于.NET的消息和通告。

目 录

第一部分 网络编程基础	1
第1章 C#语言	1
.NET基础	1
安装C#开发环境	3
C#运行时环境	6
C#编程基础	7
C#的特点	17
小结	30
第2章 IP编程基础	32
监视网络通信量	32
分析网络包	40
用TCP和UDP编程	55
寻找IP地址信息	57
使用DNS	67
小结	68
第3章 C#网络编程的类	69
套接字编程技术的引导者	69
C#套接字编程技术	81
C#套接字助手类	97
小结	101
第4章 DNS和C#	102
域名系统 (DNS)	103
Windows DNS的客户机信息	111
C#中的DNS类	125
小结	134
第二部分 网络层编程	137
第5章 面向连接的套接字	137
简单的TCP服务器	137

简单TCP客户机	142
TCP通信故障	145
在TCP通信中使用C#流	165
小结	173
第6章 无连接套接字	175
简单的UDP应用	175
UDP消息的区分	184
UDP通信故障	187
完整的UDP应用	204
小结	210
第7章 C#套接字的助手类	211
TcpClient类	211
TcpListener类	216
UdpClient类	219
在网络上传输数据	224
小结	245
第8章 异步套接字编程	247
Windows事件编程	247
使用异步套接字	252
使用异步套接字的例子程序	257
使用非阻塞套接字方法	270
Poll()程序的例子	271
小结	281
第9章 使用线程	283
如何在Windows中运行应用程序	283
在程序中创建线程	298
在服务器中使用线程	301
用线程发送和接收数据	305
线程池	311
在服务器中使用线程池	315
小结	319
第10章 IP组播	320
什么是广播	320

用广播包发布服务器广告	325
什么是组播	332
C# IP组播支持	335
组播应用程序例子	343
小结	347
第三部分 应用层编程实例	349
第11章 ICMP	349
ICMP协议	349
使用Raw套接字	352
创建一个ICMP类	353
一个简单的Ping程序	359
高级的Ping程序	362
TraceRoute.cs程序	367
FindMask程序	370
小结	374
第12章 SNMP	375
理解SNMP	375
使用SNMP包工作	379
创建简单的SNMP类	383
SimpleSNMP程序	388
使用供货商MIB	393
使用GetNextRequest查询	399
小结	403
第13章 SMTP	404
电子邮件基础	404
SMTP与Windows	407
SmtpMail类	409
使用扩展的邮件报文格式	411
邮件附件	417
MailAttachment类	420
POP3客户机	422
小结	430
第14章 HTTP	432
WebClient类	432

高级Web类	441
网络服务	447
小结	452
第15章 活动目录	454
网络目录基础	454
用活动目录工作	457
使用C#访问网络目录	460
修改目录数据	463
搜索网络目录	471
小结	476
第16章 远程技术	477
移动数据并再次访问	477
远程技术总览	487
使用远程技术	489
用soapsuds创建代理类	496
小结	501
第17章 网络安全	502
应用程序安全涉及哪些问题	502
套接字权限	510
保护网络数据	516
小结	526

第一部分 网络编程基础

第1章 C#语言

- .NET基础
- 安装C#开发环境
- C#运行时环境
- C#编程基础
- C#的特点

在很短的时间中，微软的.NET技术就已经迅速成为一种开发微软Windows工作站和服务器的流行编程平台。虽然大多数媒体的注意力都集中在.NET网络应用能力方面，但它还是有许多对于Windows编程者非常有用的特点。

专门用于开发.NET的新的C#编程语言就是这些特点中的一个。C#正在成为一种编程平台，成为需要网件和Windows系统独立应用程序的程序设计师们广泛使用的编程语言。这种语言提供了许多资源，可以帮助建立健壮的基于Windows的应用程序。为了获得这些资源的优势，现在许多程序设计师正在转移到C#语言上来。

在学习C#网络编程的基础知识之前，读者首先应当了解C#的编程环境、.NET的基础以及怎样创建和分配C#应用程序。本章将向读者介绍怎样在自己的系统上建立C#的开发环境，并介绍怎样保证用户建立的C#程序能够在其他的Windows工作站和服务器的上运行。本章的最后将简要介绍C#语言，并提出一些与网络编程有关的C#程序设计课题。本章介绍的所有概念将帮助读者做好网络编程的一切准备。

.NET基础

.NET编程语言组与以前版本的Windows编程语言的区别是，在Windows系统中建立程序和运行程序的方法不同。如果读者还不熟悉C#程序操作方法的话，本节将简要叙述为了在.NET技术的基础上展开应用程序所应当了解的基础知识。

公用语言运行时

微软.NET技术的核心是公用语言运行时（Common Language Runtime, CLR）环境。

该环境使编程者能够用多种编程语言创建程序，并在任何一个支持CLR的平台上运行这些程序。CLR的中心思想就是提供一个多种应用程序接口（Application Program Interface, API）的中间层，该中间层作用于低级的Windows Win32 API功能和应用程序代码之间。用提供一个公用的中间层的方法，微软使大量的应用程序语言有权使用核心的Windows技术（例如网络支持）。

应用程序在CLR环境中的运行方案如图1.1所示。用各种.NET语言（例如Visual Basic .NET、Visual C++.NET、Visual J#.NET，当然还有Visual C#.NET）编写的高级应用程序被编译成一种特殊的叫做微软中间语言（Microsoft Intermediate Language, MSIL）的特殊语言，当程序在主机操作系统上运行的时候，CLR再把MSIL代码翻译成普通的可执行程序。当然，不使用CLR的传统程序仍然可以像以前一样直接访问低级的Windows Win32 API。

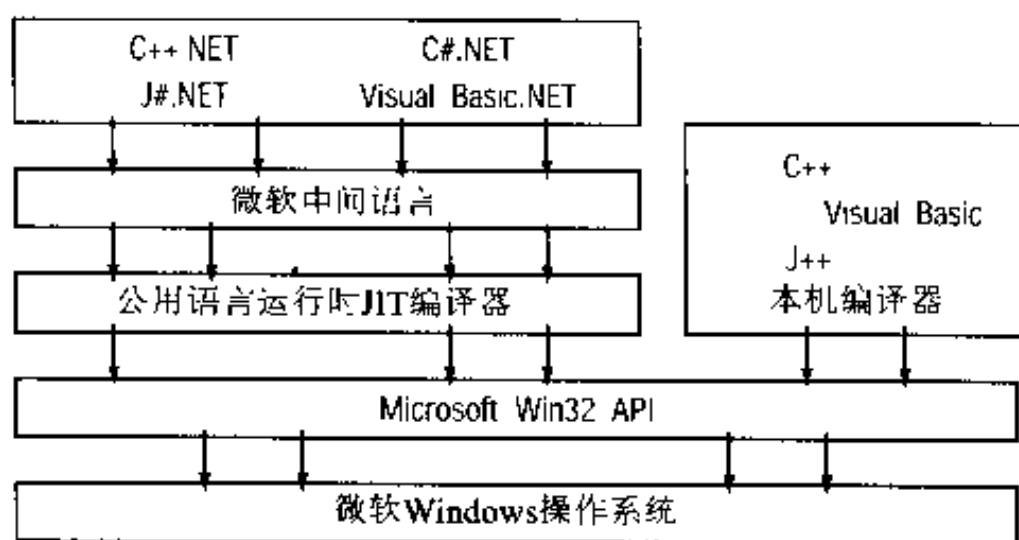


图1.1 公用语言运行时（CLR）环境

这种CLR模式也适合于其他操作系统。由于CLR被移植到其他操作系统，所以.NET程序不需要重新编译就可以在新的主机系统中运行。当前，微软支持共享源公用语言接口（CLI）方案（别名为Rotor），该方案把CLR环境转换到FreeBSD操作系统。Rotor（转子）方案也将扩展到其他操作系统，在编写本书的时候，读者可以在下列Web站点获得更多有关Rotor的信息：

<http://msdn.microsoft.com/downloads/default.asp?URL=/downloads/sample.asp?url=/msdn-files/027/001/901/msdncompositedoc.xml>

为了使程序在CLR上运行，程序必须要编译成MSIL格式。.NET C#编译器用来把C#语言的程序转换成在CLR环境中运行的MSIL格式。下一节将介绍MSIL代码。

MSIL代码

当编译C#程序的时候，会产生一个可执行文件。不过，这个可执行文件与读者过去见过的其他Windows编译器所产生的可执行文件不同。它不是一个可直接在Windows中运行的低级的汇编程序，这种可执行文件包括两个组成部分：

- 启动CLR编译器的存根汇编语言程序
- 编译后应用程序的MSIL代码

存根程序启动CLR即时（Just-In-Time, JIT）编译器，JIT编译器则把MSIL程序代码编译成可以在系统中运行的本地Win32代码。与原来的Windows应用程序不同的是，原Windows应用程序直接影响低级的Win32 API系统，而.NET应用程序要依靠.NET框架CLR来运行。如果在没有安装.NET框架的系统中运行.NET应用程序，就会产生如图1.2所示的错误信息。对任何.NET应用程序来说（不论是在Windows工作站上还是在服务器上运行），.NET框架都是至关重要的，没有它，MSIL代码就无法运行。任何打算运行.NET程序的Windows工作站或服务器都必须安装.NET框架。

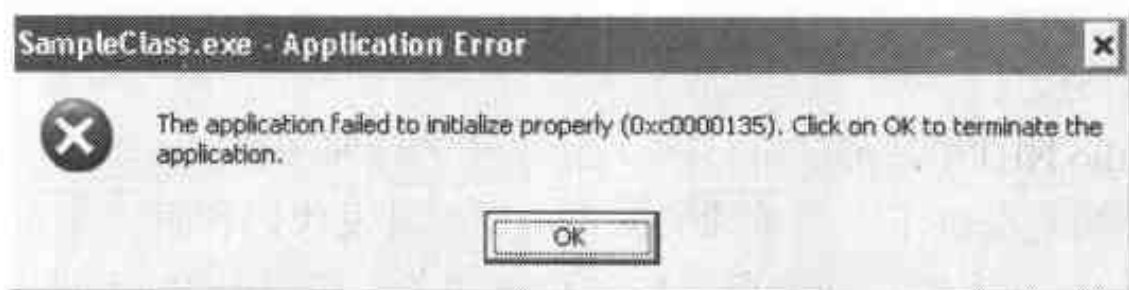


图1.2 没有.NET框架的情况下试图运行.NET应用程序

微软公司保证在它今后所有版本的Windows操作系统中都会装有.NET框架，不过，旧版本的Windows必须手工安装.NET框架。下一节将介绍在一个开发者的环境中安装.NET框架的方法和步骤，以便使用户可以创建、编译和运行.NET应用程序。

安装C#开发环境

在用C#语言编程之前，必须要有一个C#开发环境，即一个用来创建、编译和调试C#程序的系统。很遗憾，对于.NET应用程序开发最苛刻的需求就是指定用于开发的OS平台，当前，.NET要求用以下系统中的一个进行完全的C#程序开发：

- Windows NT 4 Workstation或Server（带有Service Pack 6a）
- Windows 2000 Professional或Server（带有Service Pack 2）
- Windows XP Home Edition或Professional Edition

无法使用以上系统的程序员就不能开发C#程序，但对此也有一个解决的办法，将在本章的“C#运行时环境”一节中进行介绍。

C#开发选项

微软为C#开发者提供了三种开发环境，每种开发环境都有它自己的优缺点，本节将向读者介绍这些C#开发环境和它们之间的区别。

- Visual Studio.NET
- Visual C#.NET
- .NET框架软件开发套件（SDK）

说明：本书所有的范例程序都可以在任何.NET开发环境中进行编译。为了使叙述简单化，书中显示的是使用.NET框架命令行编译器的范例，这样就保证了不管使用哪种开发环境，任何人都可以使用这些范例程序。

Visual Studio.NET

Visual Studio.NET软件包是Microsoft.NET最重要的开发产品。这个综合开发环境（Integrated Development Environment, IDE）提供了许多功能部件，可帮助用户进行Windows应用程序的程序设计。微软把Visual Studio软件包形容为“一个快速的应用程序开发（RAD）工具，使编程者能快速地编码和调试.NET应用程序”。它有一个完整的图形环境，可以创建Windows的窗体、键入代码和调试程序。除了一个优良的开发环境之外，Visual Studio还支持所有的.NET编程语言——Visual Basic.NET、Visual C++.NET、Visual J#.NET和Visual C#.NET。如果用户想用所有的.NET语言进行程序设计的话，Visual Studio软件包正是值得考虑的对象。

在Visual Studio.NET中，可以用4种编程语言中的任何一种在图形环境中创建应用程序。IDE（综合开发环境）提供了一些单独的窗口，用于开发代码和直观地安排Windows的各种应用程序控件，包括文本框、列表框、按钮和滚动条。Visual Studio.NET提供了一种容易的方法来创建、检测和调试.NET应用程序，不管这些应用程序是独立的Windows应用程序还是ASP.NET Web页面。

根据用户的开发需要（和财务状况），Visual Studio.NET软件包分好几种等级，越高的等级包括的功能越多，当然售价也越高。

Visual C#.NET

如果用户只对用C#语言编程感兴趣，他就不需要购买全部的Visual Studio.NET软件包。微软提供的Visual C#.NET软件包具有与Visual Studio.NET相同的功能，但是只支持C#语言。这对于C#编程者是一种节省得多的方法。Visual C#软件包与Visual Studio一样，也有多种软件包等级，从最基本的学生版本到全功能的专业开发者版本都有。但是在购买这个版本之前一定要搞清楚，这种版本的软件包不具备Visual Studio.NET的某些高级功能，例如自动数据库支持。

不管是Visual Studio还是C#的开发选项，都要求用户购买微软的商业软件开发软件包。这两种软件都是极好的软件开发工具，当用户创建Windows应用程序和调试程序的时候，可以节约用户大量的时间。但是，许多初学编程的和具有某些特殊爱好的编程者可能会感到这些IDE软件的价格太高。

.NET框架SDK

如果想找一种省钱的办法学习C#编程方法的话，可以选择使用.NET框架SDK。SDK是微软公司免费提供的，它包含编译和调试.NET程序的命令行工具，包括C#。这个软件包使得用户可以不用花钱购买昂贵的IDE开发环境就可以获得开发C#应用程序的感受。用户可以通过从微软的.NET框架站点下载完整的软件包来获得.NET框架SDK，或者花一点钱直接从微软购买一张光碟来获得这个软件包（有关Web站点的信息，参见下一节）。

如果用户对于C#还完全是个新手的话，最好先从网上下载.NET框架SDK并试着用它编程。人们经常说，学习编程语言的最好方法就是动手编写程序代码，经历代码的逻辑推理过程——我认为，网络编程尤其如此。随着用户C#编程技巧的不断提高，就可以转移到用Visual

C#.NET软件包创建Windows窗体和调试复杂的应用程序了。为了帮助用户起步，下一节将叙述下载和安装.NET框架SDK的方法。

下载.NET框架SDK

在编写本书的时候，.NET框架SDK的当前版本是第1版，此时可以从微软站点免费下载或者购买一张光碟。如果读者选择从微软下载，有两种方法可供选择。因为SDK很大（131MB），所以既可以把它下载成一块，也可以分为10个小的（13.1MB）软件包来下载，下载之后再组合在一起。两种方法需要下载的数据量是一样的，但是连接速度慢的用户可能想一次下载一小块数据。

当前.NET框架站点的URL是www.microsoft.com/netframework/。因为它是一个Web上的公用站点，所以在读者见到这本书的时候，它的网址已经变了。在这种情况下，读者可以直接进入Microsoft的主页（www.microsoft.com），查找有关.NET的内容即可。

.NET框架站点含有大量关于SDK的信息，其中包括一个到单独的软件下载页面的链接。下载页面上有各种下载该软件的选项。单文件下载是一个名为setup.exe的文件，它可以下载到用户的工作站或服务器上安装。如果用户选择多部分下载的选项，那就必须把所有分开的SDK文件和一个setup.bat文件下载到一个临时的目录中。在下载完所有的文件之后，必须运行setup.bat文件，这个文件会从所有的SDK文件建立一个主控setup.exe文件。

不管用两种方法中的哪一种，最终都形成一个setup.exe文件，必须运行这个文件来安装.NET框架SDK软件包。下一节将介绍安装的过程。

安装.NET框架SDK

在获得了setup.exe文件之后，不管是用一次下载的方式还是用分成多部分下载的方式，或者是从光碟上获得的，我们就可以安装.NET框架SDK了。在DOS命令提示下运行setup.exe文件或者在Windows资源管理器中双击它都可以。

在安装开始之后，对话框问用户是否想安装.NET框架SDK。单击Yes按钮之后就启动安装过程。

.NET安装程序首先把操作安装过程的文件放到TEMP环境变量指向的临时目录下，如果用户的磁盘空间比较紧张的话，可以把临时操作文件提取出来放在与系统驱动器（通常指C:\）不同的某个驱动器上。在取出操作文件之后，安装程序会更新工作站的Windows安装器软件包，然后用.NET框架安装程序启动Windows安装器。

在开始屏幕和一个版权协议屏幕之后，程序会问用户想安装哪些SDK软件包的组件，如图1.3所示。

如果用户的磁盘空间紧张，可以取消对SDK samples复选框的选择，不把SDK样本加载到工作站。在Install Options（安装选项）屏幕之后，程序会问是否要安装SDK组件。这只适用于某些组件，例如DLL（动态链接库）和必须安装在系统驱动器（通常是C:\）上的可执行文件。在用户选择了安装目的地之后，安装便开始进行。在安装完成之后就可以编译和运行C#程序了。

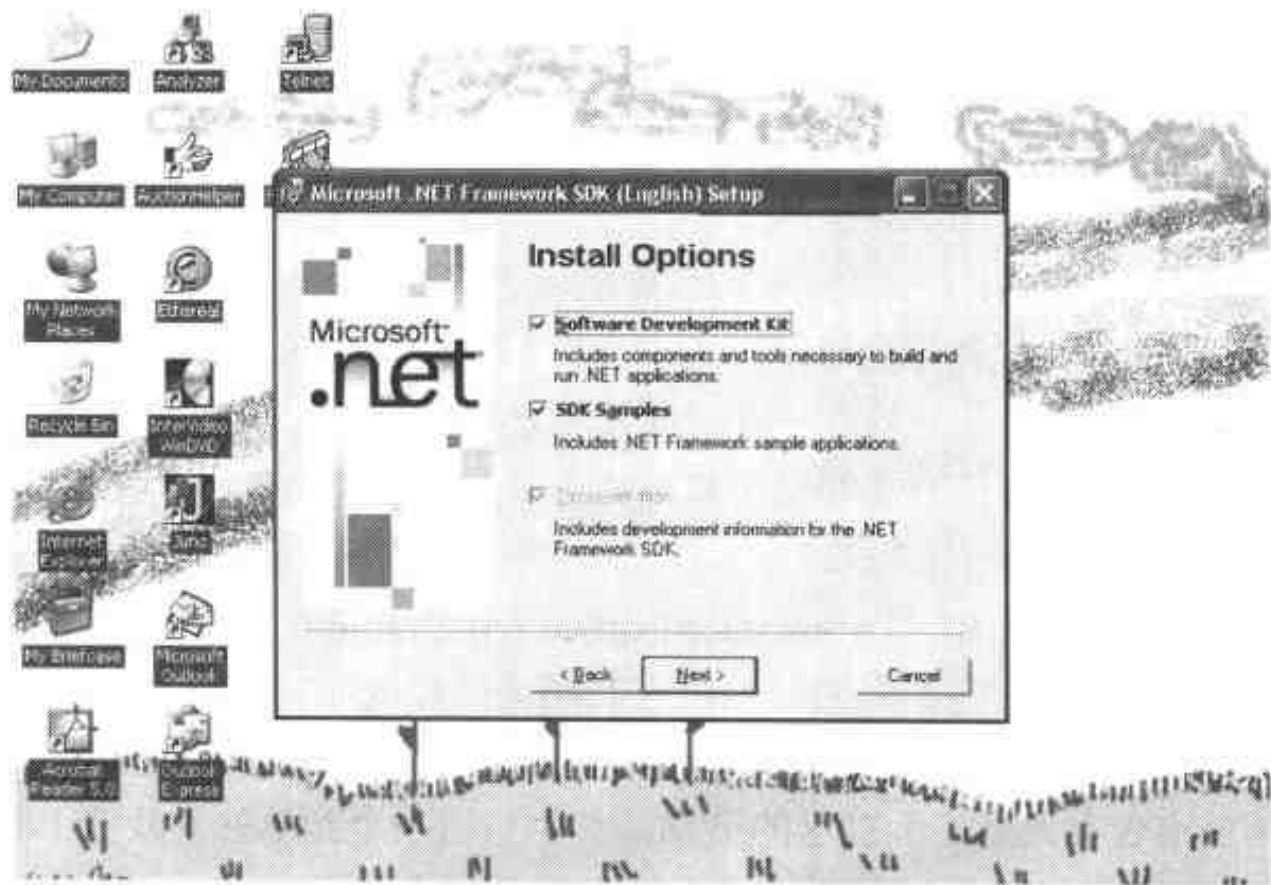


图1.3 .NET框架的安装选项屏幕

提示：用户最好时常地访问一下微软.NET框架主页，看看是否有新的服务软件包发行。然后按指示的步骤把这些软件包下载下来。

C#运行时环境

.NET框架不仅提供了一个开发环境，而且还为只运行.NET程序的Windows工作站和服务器的提供了一个CLR环境。.NET框架包含许多DLL文件，用于提供支持.NET应用程序编程的库。这些库必须被装载到用户想运行.NET程序的工作站和服务器的上。不过，不需要把.NET框架SDK装载到运行用户程序的所有计算机上，正如本书前面间接提到过的，微软有一个处理这个问题的解决方案。

微软已经明确表示，从Windows.NET Server版开始，今后所有的Windows操作系统都将包含.NET框架运行时（CLR）环境。这将保证不需要用户手工安装任何.NET框架库，就可以在新的操作系统上运行.NET应用程序。很可惜不能在老版本的Windows（98、Me、NT、2000和XP）上运行，解决这个问题的办法是手工安装.NET框架库的运行时版本。

.NET框架的运行时版本是比SDK软件包发行量小得多的软件。它只包含在Windows工作站和服务器的上运行.NET应用程序所必需的文件。下一节将介绍在老版本的Windows平台上下载和安装.NET框架运行时文件的方法。

下载和安装C#运行时软件包

C#运行时软件包是.NET框架可重分布软件包的一部分。该可重分布软件包带有所有.NET语言的运行时环境。类似于SDK软件包，可重分布软件包可以从.NET框架的主页 www.microsoft.com/netframework/ 下载得到。由于它是一个比较小的软件，因此没有多个部

分下载的选项，整个软件包（大约20MB）必须作为一段程序下载。

下载文件dotnetredist.exe实际上是一个压缩的文件，必须把它解压缩到一个工作目录中。当运行这个程序的时候，它会问用户使用的是哪个工作目录，用户可以选择任何一个有足够大空间的地方（大约22MB）来存放它。解压缩文件名为dotnetfx.exe。

.NET框架可重分布软件包没有更多需要安装的。在启动dotnetfx.exe文件之后，程序会要求用户确认自己想要安装，然后才开始安装。不需要用户选择安装的位置，可重分布软件包会自动安装在系统盘上默认的地方。

用C#运行时开发

当前，.NET框架可重分布软件包包含C#编译器程序csc.exe。它允许没有运行Windows NT 4、2000或XP的工作站或服务器的开发者不安装.NET框架SDK就编译C#程序。任何一台支持.NET框架运行时的计算机都可以用csc.exe来编译C#应用程序，包括Windows 98和Me工作站。虽然包含编译器，但不包括任何文档和调试程序，所以它是一个极精炼的开发环境。

如果用户想从可重分布软件包编译C#程序，必须把csc.exe程序的位置添加到系统的PATH环境变量。不同的用户系统添加的方法也不同。

- 对于Windows 98和Me系统，可以把PATH语句加到C:\目录中的autoexec.bat文件中。可能此目录下已经有autoexec.bat文件了，文件中也已经有了一个PATH语句。如果是这样，用户只需要把csc.exe文件的路径添加到已有的PATH语句中即可。
- 对于.NET框架可重分布软件包第1版，csc.exe文件的位置是在C:\Windows\Microsoft.Net\Framework\v1.0.3705目录中（记住，在Windows 98和Me中，Windows系统目录是Windows；而Windows NT和2000的系统目录是Winnt）。

在重新启动系统之后，用户应当能够从命令行提示访问csc.exe编译器，如下例所示：

```
C:\>csc
Microsoft (R) Visual C# .NET Compiler version 7.00.9466
for Microsoft (R) .NET Framework version 1.0.3705
Copyright (C) Microsoft Corporation 2001. All rights reserved.

fatal error CS2008: No inputs specified

C:\>
```

现在已经准备就绪，可以用C#语言开始编程了。

C#编程基础

虽然.NET框架SDK是一个精炼的开发环境，但它还是含有相当多的工具，允许用户用来创建、编译和调试C#程序。本节将介绍用户可以使用的一部分工具。

创建C#程序

如果读者现在正在使用某个Microsoft Visual产品开发程序的话（Visual Studio.NET或

Visual C#.NET)，那就拥有了一个完整的程序编辑环境，包括帮助文件、图形向导和命令完成向导。如果使用.NET框架SDK软件包的话，用户就必须自己来生成和编译自己的C#代码。虽然SDK的这一特点与流行的Visual软件包相比显得逊色，但这恰恰是生成C#应用程序的正确方法。

在.NET框架开发环境中操作C#程序的第一步就是把C#源代码文件扩展名与文本编辑器相关联。这会使程序编辑起来容易得多，用户只需要从Windows资源管理器程序中双击一个程序就可以对它开始编辑。选择编辑器的类型十分重要，应当选择一种允许把源代码文件保存为文本方式的，而不是保存成Microsoft Word.doc文件或者其他的字处理文档，因为C#编译器必须能够翻译每一行代码。如果读者选择一种字处理软件包来编辑C#程序的话，一定要把整个文件保存为纯文本格式。

在选择了编辑器之后，使Windows资源管理器中编辑器的应用程序与.CS文件类型相关联：右键单击一个C#程序，然后选择Open With选项，并从列表中选择适当的应用程序。

如果读者是一位初学C#的新手，有可能想实际体会一下编辑和调试C#程序的过程。为此，首先必须有一个可进行操作的样本程序。程序清单1.1就是这样的一个样本，可以演示C#的基本编程原则。

程序清单1.1 SampleClass.cs程序

```
class DataClass
{
    private int a;
    private int b;

    public DataClass(int x, int y)
    {
        a = x;
        b = y;
    }

    public int addem()
    {
        return a + b;
    }
}

class SampleClass
{
    static int sampleX;
    static int sampleY;

    public SampleClass()
    {
        DataClass sample = new DataClass(sampleX, sampleY);
        System.Console.WriteLine("The result is: {0}", sample.addem());
    }
}
```

```
    }

    public static void Main(string[] argv)
    {
        if (argv.Length != 2)
        {
            System.Console.WriteLine(" Usage: SampleClass x y");
            return;
        }
        sampleX = System.Convert.ToInt16(argv[0]);
        sampleY = System.Convert.ToInt16(argv[1]);
        SampleClass starthere = new SampleClass();
    }
}
```

这个样本程序包含两个单独的C#类，即DataClass和SampleClass。DataClass类声明了两个专有的整数（也就是说，它们只能从DataClass类访问）、一个类的构造器和一种操作该类数据的方法。当用具体例证说明一个程序的时候，DataClass构造器详细说明发生了什么事情：

```
public DataClass(int x, int y)
{
    a = x;
    b = y;
}
```

默认的构造器要求两个整数，这两个整数被赋予了在类中定义的a和b两个变量。定义的一种方法是把它们相加，返回一个整数值是两个专有变量的和：

```
public int addem()
{
    return a + b;
}
```

说明：在程序中定义了DataClass类之后，程序中的其他类就可以使用它了。与C++不同的是，在C#中没有事先声明之前就可以先使用类，C#编辑器会意识到它需要的类一定是在程序的后面。用户甚至可以在不同的文件中声明类，如果用户声明的类从来没有在命令行所列的程序文件中出现的话，编辑器也只是抱怨一下而已。

SampleClass类包含两个静态整数变量、一个构造器和一个Main()方法，该方法指示C#编辑器从哪里开始执行程序。Main()方法首先检查两个命令行参数，确保它们已经输入，把它们转换成整数值，并把它们赋予两个已定义的整型变量。然后，它产生一个SampleClass实例：

```
SampleClass starthere = new SampleClass();
```

这将迫使CLR执行SampleClass构造器，以产生该类的一个新实例。

SampleClass构造器代码产生了一个DataClass类的实例，把两个整数传送给DataClass类构造器。从示例的SampleClass变量中调用addem()方法，并返回加法的结果。用下面这行代码把addem()方法的结果显示在控制台的屏幕上：

```
System.Console.WriteLine("The result is: {0}", sample.addem());
```

符号{0}作为一个位置值，表示列在文本串后面的一个变量。在此例的情况下，它替代sample.addem()方法的返回值。也可以用一些连续的位置编号（{1}，{2}，等）把其他的变量添加到变量列表中，添加的变量之间用逗号分开。

在键入程序代码之后，必须使用.CS扩展名保存文件，这就标识着这个文件是C#代码文件。在文件保存之后就可以对它进行编译了。

编译和运行C#程序

.NET框架SDK和可重分布软件包中都含有C#编译器csc.exe。任何C#程序，不论多么复杂都可以用这个编译器来编译。在命令行可以用许多不同的开关来控制编译器的功能状态，表1.1中列出了部分最常用的开关。

表1.1 csc命令行开关

开关	功能
/out:filename	定义程序的可执行文件名
/main:classname	定义包含Main()方法的类
/target:target	定义程序的类型。其结果是：exe用于表示基于控制台的应用程序；winexe用于表示Windows的图形应用程序；library用于表示Windows DLL（动态链接库）文件；module用于表示汇编模块
/debug:type	用于创建可执行文件的调试信息。其类型可以是full（默认情况下），它能够把调试器联系到一个正在运行的程序上；类型也可以是pdbonly，它只为在调试工具中进行调试而创建一个.pdb数据库文件
/resource:<res>	嵌入可执行文件中指定的资源

当用户确定了自己需要的命令行选项（如果有的话）之后，用csc命令行编译器编译C#程序是很简单的：

```
C:\>csc SampleClass.cs
Microsoft (R) Visual C# .NET Compiler version 7.00.9466
for Microsoft (R) .NET Framework version 1.0.3705
Copyright (C) Microsoft Corporation 2001. All rights reserved.

C:\>
```


如果命令提示行没有返回任何文本信息，就说明编译是成功的。如果编译器显示出错误或警告信息，用户就必须编辑源代码文件来纠正错误。编译器产生的每一个错误或警告都表示程序行中存在错误。下面是当源代码中有键入错误时产生错误信息的例子：

```
C:\>csc SampleClass.cs
Microsoft (R) Visual C# .NET Compiler version 7.00.9466
for Microsoft (R) .NET Framework version 1.0.3705
Copyright (C) Microsoft Corporation 2001. All rights reserved.

SampleClass.cs(36,12): error CS0117: 'System.Convert' does not contain a
        definition for 'ToInt16'

C:\>
```

注意，行号和程序行中的位置都放在错误信息中的圆括号中。另外还会看到对错误的清楚描述，它帮助用户找出产生错误的原因。如果用户使用一个支持行号的文本编辑器，那就可以很容易地到源代码中找到并纠正错误。否则的话，只好一行行地数了！

在成功地编译了程序之后，就可以从命令行运行该程序了：

```
C:\>SampleClass 5 10
The result is: 15

C:\>
```

可以看到该程序成功地运行，并显示出命令行参数相加的结果。当然这个简单的例子并没有对命令行参数做更多的检查，所以要注意，只能输入数字，否则程序会放大，并产生错误信息——更多有关这方面的信息，将在后面“C#异常编程”一节中加以介绍。

使用多重源文件

SampleClass.cs范例程序在一个源代码文件中定义了两个单独的类，对于一个小的实例是很容易的，但如果在较大的程序中，当把多个类组合在一个源代码文件中的时候，有可能会搞糊涂。最好为应用程序中使用的每一个类创建一个单独的源代码文件，这样就可以进行更好的代码管理，尤其当好几个人同时操作一个包含几百个类的应用程序的时候：

- DataClass.cs用于DataClass类代码
- SampleClass2.cs只用于SampleClass类代码

当把一些类分离出来放到离散的源代码文件中的时候，有些事情必须注意。首先是要保证C#编译器在编译的时候能够找到它们。做到这一点最容易的办法是在命令行中包含所有有关的源代码文件，如下例所示：

```
C:\>csc SampleClass2.cs DataClass.cs
```

不过，在这样做的时候一定要注意，因为源代码文件首先列出的是默认的.exe文件名。如果用户想修改.exe文件名，可以使用/out:命令行开关：

```
C:\>csc /out:SampleClass2.exe DataClass.cs SampleClass2.cs
```

另一件重要的事情是要告诉编译器从程序的什么地方开始执行。如果只有一个类定义了Main()部分，执行时不会有什么问题。不过，有时候不同的类可以使用其他类中的方法，但是两个类可能都包含Main()方法，这会使得编译器混乱，不知道从哪个Main()方法开始运行程序。

csc.exe程序的命令行开关解决了这个问题。/main: switch指定了用户要使用的那个包含Main()方法的类：

```
C:\>csc /main:SampleClass SampleClass2.cs DataClass.cs
```

注意，必须指定包含Main()方法的类，而不是源代码的文件名。

调试C#程序

.NET框架SDK提供了两种极好的调试C#程序的方法：

- dbgclr是一个GUI（图形用户界面）的调试程序
- cordbg是一个命令行文本调试程序

图形dbgclr程序和文本方法cordbg程序有类似的功能，但呈现不同的界面。这两个程序都允许用户一步步地通过C#程序，在执行的过程中可以看到变量和输出。不过，在执行这些程序之前，必须用csc编译器的/debug选项编译可执行程序：

```
C:\>csc /debug SampleClass.cs
```

以上命令完成两个动作：第一，把属性设置成可执行文件，该文件通知CLR JIT编译器必须进行代码跟踪；第二，生成一个程序员数据库（PDB，programmer database）文件，该文件包含调试器的代码跟踪信息。相加的属性调用JITTracking标记，它通知CLR JIT编译器，必须把代码从产生的本地代码中分解出来，返回MSIL指令，最终映射回到初始的源代码。所有这些信息都包含在可执行文件的PDB文件中。

使用dbgclr程序

dbgclr程序提供了一个Windows环境，可用来观看和跟踪正在运行的C#程序，以便发现编码的错误。dbgclr程序存放在用户安装SDK时所指定的Microsoft.Net目录下。默认的位置如下：

```
C:\Program Files\Microsoft.Net\FrameworkSDK\GuiDebug\dbgclr.exe
```

当dbgclr运行时，用户必须指定应用程序的源代码和可执行文件的位置。可按以下步骤进行：

1. 在Menu（菜单）栏中单击Debug►Program to Debug选项。
2. 在Program（程序）文本框的旁边，单击省略符号（...）按钮，然后选择我们要调试的SampleClass.exe程序（记住，可执行程序必须已经用/debug开关编译过）。Working Directory（工作目录）文本框会自动显示可执行文件的目录位置。另外，这时还要在Arguments（参数）文本框中键入程序所要求的参数。对于SampleClass程序来说，就是键入任意两个数字。完成之后单击OK。

3. 单击File►Open►File, 选择应用程序的SampleClass.cs源代码文件, 并单击Open (打开)。

此时, dbgclr程序会显示出以下4个单独的窗口:

- 源代码文件窗口
- Solution Explorer (解决方案管理器) 窗口
- 应用程序输出窗口
- 命令窗口

这时可以在源代码窗口看到SampleClass.cs文件, Solution Explorer (解决方案管理器) 窗口会在Miscellaneous Files部位列出这个文件。为了启动调试过程, 在菜单栏单击Debug►Step Into项。这样便启动了程序, 并允许单步通过代码 (如图1.4)。

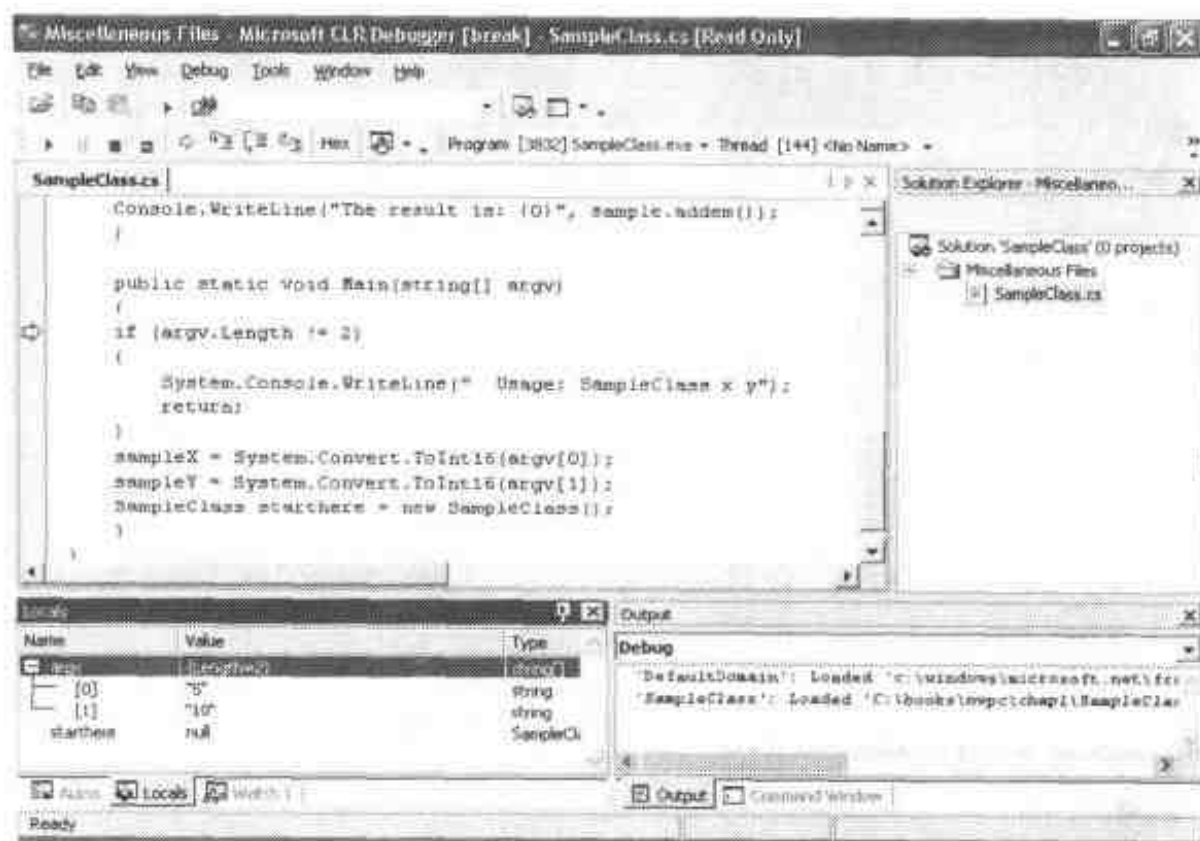


图1.4 用dbgclr单步通过应用程序

当调试过程开始之后, 出现了一个新的窗口, 显示出应用程序代码中使用的各种变量。注意在图1.4中, 用户输入的命令行参数值显示在Locals选项卡下, 还显示有长度的值 (这个值是2)。这使得用户在程序执行的过程中可以方便地看到变量的值。当用户在程序中发现有损毁的变量值, 想调查问题出在何处的时候, 这是很方便的。

Step Into函数从Main()部分开始一行一行地执行程序。当前的代码行用黄色突出显示, 按下F11键, 用户可以单步执行整个程序。也可以单击工具条上的一些按钮来跳过代码, 也可以通过 (或者返回) 某个代码段。这样就使用户对于观察程序的执行过程有极大的控制能力。

使用cordbg程序

cordbg命令行工具没有图形窗口, 它与dbgclr程序有类似的功能。它也允许用户单步地执行程序并在执行的过程中监视变量值, 但允许文本输入和提供文本输出。程序清单1.2展

示了用cordbg程序完成调试过程的例子。

程序清单1.2 从cordbg输出的范例

```
C:\>cordbg
Microsoft (R) Common Language Runtime Test Debugger Shell Version 1.0.3705.0
Copyright (C) Microsoft Corporation 1998-2001. All rights reserved.

(cordbg) run SampleClass.exe 5 10
Process 356/0x164 created.
Warning couldn't load symbols for c:\winnt\microsoft.net\framework\v1.0.3705\ms
corlib.dll
[thread 0xff] Thread created.

031:    if (argv.Length != 2)
(cordbg) sh
025:    System.Console.WriteLine("The result is: {0}", sample.addem());
027:    }
028:
029:    public static void Main(string[] argv)
030:    {
031:*   if (argv.Length != 2)
032:    {
033:        System.Console.WriteLine(" Usage: SampleClass x y");
034:        return;
035:    }
036:    sampleX = System.Convert.ToInt16(argv[0]);
(cordbg) pro
PID=0x164 (356) Name=C:\SampleClass.exe
      ID=1 AppDomainName=SampleClass.exe
(cordbg) p argv
argv=(0x00e718b8) array with dims=[2]
  argv[0] = (0x00e718d0) "5"
  argv[1] = (0x00e718e4) "10"
(cordbg) s
036:    sampleX = System.Convert.ToInt16(argv[0]);
(cordbg) so
037:    sampleY = System.Convert.ToInt16(argv[1]);
(cordbg) so
038:    SampleClass starthere = new SampleClass();
(cordbg) s
023:    public SampleClass()
(cordbg) s
```

```
{0007] nop
(cordbg) s

[001c] mov          ecx,0B65210h
(cordbg) s

006:      public DataClass(int x, int y)
(cordbg) s

[0007] nop
(cordbg) s

[0014] mov          dword ptr [esi+4],edi
(cordbg) s

009:      b = y;
(cordbg) s

010:      }
(cordbg) s

025:      DataClass sample = new DataClass(sampleX, sampleY);
(cordbg)
```

注意，当cordbg程序被启动之后，可以使用run命令和可执行程序的文件名，以及该可执行程序的所有有关的命令行参数。或者也可以用可执行程序 and 命令行变量来运行cordbg命令。

cordbg程序用文本命令通过程序并显示有关的信息。表1.2描述了部分可以使用的文本命令。

表1.2 cordbg文本命令

命令	功能
s	进入源代码的一行
si	进入源代码的一行
so	越过代码的下一行
ss	进入下一条本机的或IL（中间语言）的指令
p arg	打印变量arg的当前值
pro	显示运行程序的系统进程信息
reg	显示当前线程的CPU注册表
run prog	在调试器中运行程序prog
break	设置或显示代码中的一个断点
sh	显示代码的当前行和它的前后五行

正如论证的那样，在dbgclr中可以做的事在cordbg中也都可以做。事实上，许多高级程序开发者都更喜欢用cordbg，因为做同样的事情它要比dbgclr更快，因为图形dbgclr程序需要等待。

观看C#程序的执行是一种调试用户应用程序的方法。下一节将介绍一种工具，它可以用来观察csc编译器所生成的实际的MSIL代码，并由CLR运行。

调试MSIL代码

如果想了解用户程序内部的情况，必须要看MSIL代码——即为了创建主机本机的系统代码，由CLR JIT编译器编译过的实际代码。.NET框架SDK给用户提供了一个完成这项工作的工具：微软中间语言反汇编程序（Intermediate Language Disassembler, IL DASM）。用户必须从命令行运行ildasm.exe程序，并给看代码的监视器提供CLR可执行程序名。

```
C:\>ildasm SampleClass.exe
```

图1.5显示出IL DASM窗口，还有程序中包含的类和变量。IL DASM显示出分等级的代码视图，把类和变量与类中的方法分开。如果想看实际的MSIL代码，可以双击某个单独的部分。图1.6显示出单击addem()方法后的结果。即使对MSIL了解得不多，用户也可以看出，这部分代码从内存中检索到两个值，并把它们相加。

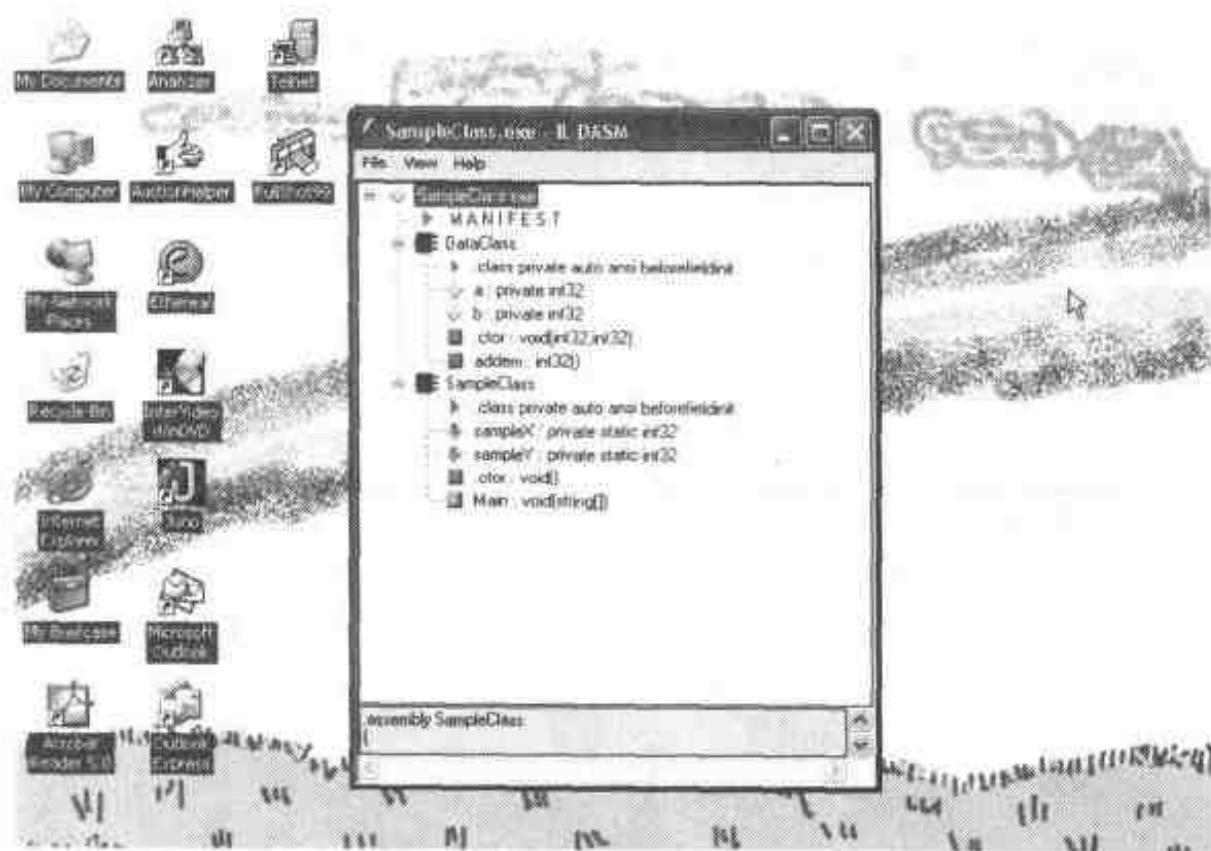


图1.5 IL DASM窗口

说明：如果想在CLR级调试应用程序的话，必须要了解MSIL汇编代码才行。这已经超越了本书的范围。

现在读者已经熟悉了C#的开发环境，可以开始编写C#代码了。不过首先还要了解C#的特点，那些与其他编程语言不同而又经常在网络编程中使用的特点。如果读者已经熟悉C#，完全可以跳到下一章去学习。

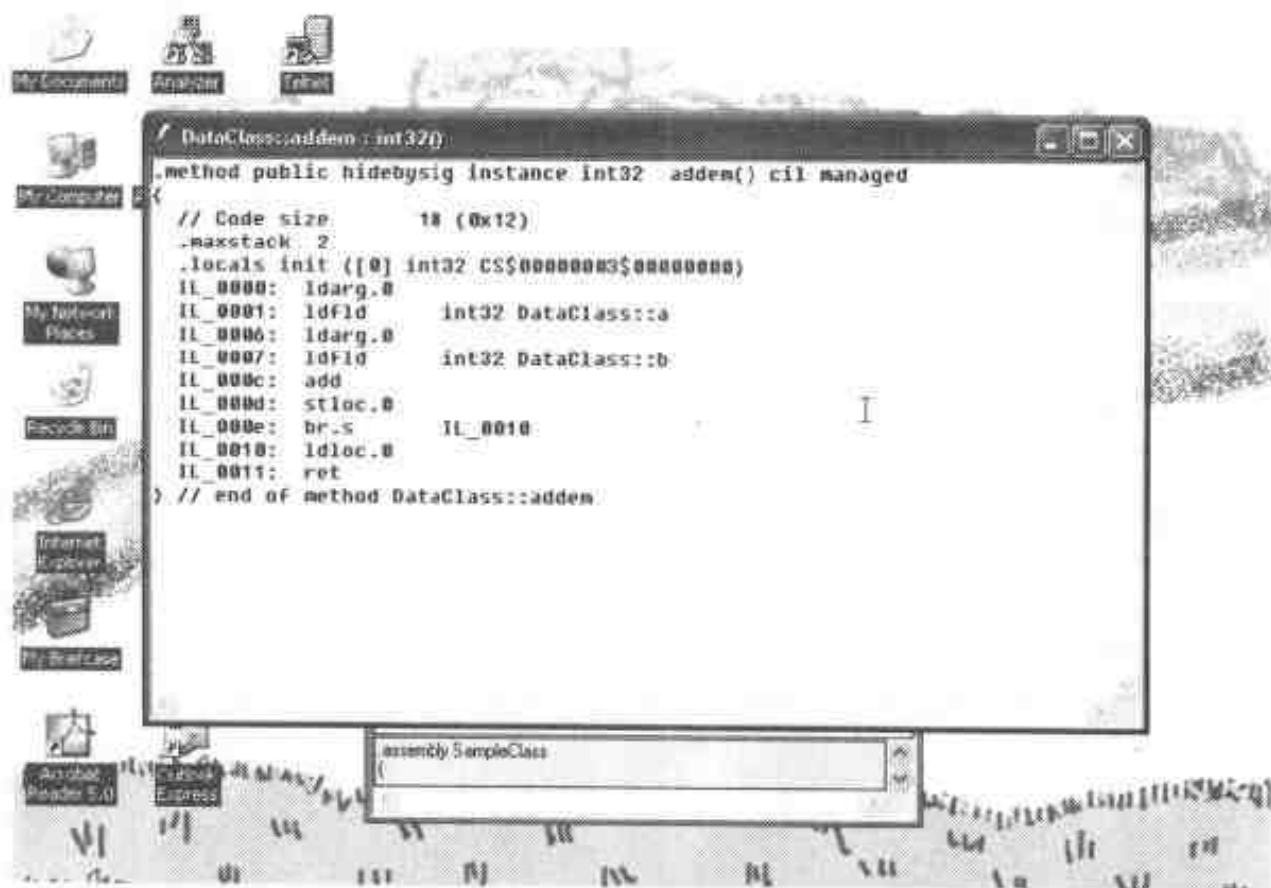


图1.6 由addem()方法产生的MSIL代码

C#的特点

C#是微软推出的一种面向对象的程序设计语言，它结合了许多新的功能和特点，这些特点是富有经验的C、C++和Visual Basic程序员过去不曾使用过的。如果读者对于C#编程根本不熟悉，建议先买一本C#程序设计的书看一看，例如Jason Price和Michael Gunderloy编写的“Mastering Visual C#.NET”或Harold Davis编写的“Visual C#.NET Programming”，这两本书都是Sybex出版的，书中的内容全面探索了这种令人激动的语言的特点（这两本书的中文版已由电子工业出版社出版，书名分别是《Visual C#.NET从入门到精通》和《Visual C#.NET应用编程》）。下面一节只是概要性地介绍C#的一些独特功能，在开始网络编程前先了解这些特性是至关重要的。

C#命名空间

对于在.NET框架中提供的所有的类来说，最容易使人感到困惑的是到底哪些类执行哪些功能和应当由特殊的类使用的一些方法。为了使问题简化，微软在对.NET框架的类分类的时候使用了命名空间。

什么是命名空间

正如我们在SampleClass程序中所看到的，每个C#应用程序都含有一个或多个类。每个类定义一个可以包含数据和操作数据的方法的对象。在每个应用程序中至少有一个类必须包含一个叫做Main()的程序接口方法。Main()方法使C#编译器知道从哪里开始执行这个程序。其他的类可以在程序中定义（例如DataClass），或者甚至可以与其他程序共享。

在程序之间共享类是面向对象程序设计要达到的目标。在类共享中，涉及到的一件事是：惟一的和有意义的类名十分重要。如果用户是单独一个人在一个简单的项目上工作，两

个（或更多的）类使用相同的名字不见得会引起多大的麻烦。可是，在一个需要创建几百个类的大型开发队伍中，具有一个适当的类命名结构对于项目成功是十分关键的。

C#命名空间用于识别高层次的类名，允许用户在一个命名空间里把相似的类分成组。命名空间在源代码文件中是在定义类之前定义的，使用命名空间的命令如下：

```
namespace Test1;

class testProgram
{
}

namespace Test2;
class testProgram
{
}
```

对于没有声明命名空间的程序（例如SampleClass程序），定义的类就成为全局命名空间的一部分。这些类对于CLR中所有的应用程序是全局可用的。

每一个命名空间都可以惟一地识别它里面的程序。注意，两个样本命名空间只包含一个叫做testProgram的类；很可能它们执行不同的功能。如果用户的程序需要使用某一个或两个testProgram类，就必须用命名空间指出你想用的是那个类。

.NET框架包含有许多离散在命名空间中的类，命名空间有助于把各种类按共同的功能分成组。用户必须了解怎样适当地声明使用的类，才能在程序被编译的时候不会使编译器混淆。下面来研究一下.NET框架中定义的命名空间。

.NET框架的命名空间

.NET框架用命名空间帮助把CLR中使用的库类进行分类，这有助于程序员确定各种类的位置和怎样在程序中定义它们。

许多.NET框架命名空间形成了核心CLR类。表1.3列出了部分在C#网络应用程序中会经常碰到的命名空间。

表1.3 .NET框架类的命名空间

命名空间	类的描述
Microsoft.Win32	处理由OS和注册表处理类而引起的事件
System	定义通常使用的数据类型和数据转换的基本.NET类
System.Collections	定义列表、队列、位数组和字符串集
System.Data	定义ADO.NET数据库结构
System.Data.OleDb	封装OLE DB.NET数据库结构
System.Drawing	提供对基本图形功能的访问

(续表)

命名空间	类的描述
System.IO	允许读和写数据流和文件
System.Management	提供对Windows系统管理方法（WMI）底层结构的访问
System.Net	提供对Windows网络功能的访问
System.Net.Sockets	提供对Windows套接字（Winsock）接口的访问
System.Runtime.Remoting	提供对Windows分布式计算平台的访问
System.Security	提供对CLR安全许可系统的访问
System.Text	代表ASCII、Unicode、UTF-7和UTF-8字符编码
System.Threading	允许多线程编程
System.Timers	允许在指定的时间间隔引发一个事件
System.Web	允许浏览器和Web服务器功能
System.Web.Mail	允许发送邮件信息
System.Windows.Forms	创建使用标准Windows图形接口的基于Windows的应用程序
System.XML	提供对处理XML文档的支持

在程序中使用命名空间

正如已经说明的那样，每个命名空间都提供了对某个特定类组的支持。用户在找到了含有自己程序中需要的类的命名空间以后，必须在自己的程序中定义这些命名空间，然后才能访问这些类。有两种在程序中标识.NET框架类的方法。

第一种是在SampleClass程序中演示的方法：

```
System.Console.WriteLine("The result is {0}", sample.addem());
```

这个命令使用了在System命名空间中Console类的WriteLine()方法。注意在引用方法时使用的层次。首先宣布命名空间，后面跟着的是类名，最后是方法名。用户可以在自己的程序中用这种方法引用所有的.NET框架类，不过也许很快就感觉到键盘录入的疲劳。

一种较容易的方法是，在程序的开头处用C#的using指令来声明命名空间。用using声明的命名空间中包含的所有类没有必要用它们的命名空间名来引用：

```
using System;
Console.WriteLine("The result is {0}", sample.addem());
```

C#编译器搜索所有Console类声明过的命名空间，自动地引用正确的命名空间。

警告：使用这种声明命名空间的方法时一定要注意，因为可能会遇到这样的情况，两个声明的命名空间具有相同名字的类（例如Timer类）。在这种情况下，当引用类的时候必须提供完整的命名空间名。否则，C#编译器就无法确定哪一个类是用户想引用的。

在声明了命名空间和用户程序中使用了命名空间的类之后，必须要保证，当编译用户程序的时候C#编译器能够找到正确的类库。下一节将介绍应当怎样做。

用命名空间编译程序

.NET框架把CLR类库当做一个DLL（动态链接库）文件集来实现。每个DLL文件包含一个特殊的来自一个或多个命名空间的类的子集。不仅当运行用户的.NET程序时必须使用这些DLL，而且当编译程序的时候还必须在命令行引用它们。

用户必须引用程序中声明的支持所有命名空间所需要的每一个DLL，当引用DLL时，应当使用csc编译器的/resource命令行开关：

```
C:\> csc /r:System.dll SampleClass.cs
```

（/resource开关可以省略为/r）。这里，System命名空间的类包含在System.dll文件中，所以为了正确地编译SampleClass.cs文件，必须在命令行引用System.dll文件。读者可能会感到不解，为什么过去编译程序的时候不需要这样做呢？这里有一个诀窍。

csc.exe编译器程序使用一个配置文件，该配置文件设置了一些标准的命令行参数，其中包括要引用的默认的DLL文件。这个配置文件叫做csc.rsp，它与csc.exe位于同一个目录下。

读者可以用任何一个文本编辑器（例如微软的记事本）检查csc.rsp文件。程序清单1.3显示了在我的.NET框架版本中安装的默认csc.rsp文件。

程序清单1.3 默认的csc.rsp文件

```
# This file contains command-line options that the C#
# command line compiler (CSC) will process as part
# of every compilation, unless the "/noconfig" option
# is specified.

# Reference the common Framework libraries
/r:Accessibility.dll
/r:Microsoft.Vsa.dll
/r:System.Configuration.Install.dll
/r:System.Data.dll
/r:System.Design.dll
/r:System.DirectoryServices.dll
/r:System.dll
/r:System.Drawing.Design.dll
/r:System.Drawing.dll
/r:System.EnterpriseServices.dll
/r:System.Management.dll
/r:System.Messaging.dll
/r:System.Runtime.Remoting.dll
/r:System.Runtime.Serialization.Formatters.Soap.dll
/r:System.Security.dll
```

```
/r:System.ServiceProcess.dll
/r:System.Web.dll
/r:System.Web.RegularExpressions.dll
/r:System.Web.Services.dll
/r:System.Windows.Forms.dll
/r:System.XML.dll
```

大多数流行的命名空间DLL都在csc.rsp文件中被引用了。这个功能给用户带来了方便，当用户使用几个命名空间的类时就不需要在命令行引用许多文件。按照csc.rsp文件的注释中所说明的，用户也可以使用/noconfig命令行开关，完全不考虑csc.rsp的值：

```
C:\>csc /noconfig /r:System.dll SampleClass.cs
```

这个命令将只月System.dll文件的引用进行编译。

说明：把引用增加到其他的DLL文件并不会最后增加可执行文件的大小。引用的目的只不过是告诉编译器到哪里去找命名空间的定义，而不是为了在DLL代码中进行编译。库类代码仍然从DLL运行。这就是.NET框架必须安装在目标工作站和服务器的原因。

在C#程序中使用字符串

C编程最困难的一个部分就是对字符串的处理。许多程序的安全漏洞都是由于字符串缓冲器溢出而造成的。在这些程序中程序员对字符串使用了字符数组，而黑客就把比内存字节数还要多的字符分配给这个字符串。为了减少在C#程序中处理字符串所带来的问题，微软在C#语言中设置了两个字符串处理类。由于许多网络协议都涉及到发送和接收文本字符串，所以读者最好能很好地掌握在C#网络程序中字符串的使用。这一节将讨论如何使用C#语言中的.NET字符串类，帮助读者解决使用字符串的问题。

String（字符串）类

在C#中，字符支持最基本的部分就是String（字符串）类。String类允许用户把一连串字符赋值给一个变量，并在程序中作为一个单独的单元来处理这个变量。String类也包含有几个方法，可以对字符串对象执行操作，例如确定字符串的长度和组合两个字符串。

String构造函数过载（overloaded），提供了几种生成字符串变量的方法。表1.4描述了字符串构造函数。

表1.4 String构造函数

构造函数	说明
string(char[])	根据指定的字符数组生成一个字符串
string(char, int)	把指定的字符重复整数倍生成一个字符串
string(char[], int1, int2)	根据指定的字符数组生成一个字符串，从整数1位置开始，长度为整数2个字节

根据C#的一个特点，用户在定义字符串的时候，既可以在String声明中使用大写字母S也可以使用小写字母s。下面是几个声明字符串变量的例子：

```
string test = "This is a test string";
string test2 = test;
string anotherTest = new string('a', 10);
```

上面列出的第一种方法是生成新字符串最普通的方法。在生成了字符串之后，就可以用好几种方法对字符串进行操作和处理。表1.5向读者展示了一些比较常用的方法。

表1.5 String方法

方法	说明
Clone	返回对字符串的引用
Compare	比较两个指定的字符串
CompareTo	把一个字符串与另一个对象比较
Concat	连接两个字符串
Copy	用一个现有字符串的值生成一个新的字符串
CopyTo	从一个字符串的指定位置开始把指定数量的字符复制到另一个字符串中
EndsWith	判断一个字符串是否用一个指定的字符串结束
Equals	判断两个字符串是否具有相同的值
IndexOf	返回一个字符串中的第一个指定字符串的具体位置
Insert	在一个字符串指定的位置上插入指定的字符串
Intern	检索系统对字符串的引用
Join	在字符串数组的每个元素之间连接一个指定的字符串
LastIndexOf	返回字符串中最后一个指定字符串具体位置的索引值
PadLeft	右对齐字符串的字符并把最左边的字符设置为空格
PadRight	左对齐字符串的字符并把最右边的字符设置为空格
Remove	从字符串中删除指定数量的字符
Replace	用另一个指定的字符或字符串替代所有指定字符或字符串的具体值
Split	根据一个指定的分离字符在字符串中识别子字符串
StartsWith	判断一个字符串是否用一个指定的字符串开始
ToCharArray	把字符串中的字符复制到一个字符数组中
ToLower	返回一个字符串的副本，把所有的字符都设置为小写字母

(续表)

方法	说明
ToString	把对象的值转换为一个字符串
ToUpper	返回一个字符串的副本, 把所有的字符都设置为大写字母
Trim	从一个字符串的开头和结尾删除一组指定字符所有的具体值
TrimEnd	从一个字符串的结尾删除一组指定字符所有的具体值
TrimStart	从一个字符串的开头删除一组指定字符所有的具体值

有了这些可供用户使用的字符串方法, C#程序中的字符串就容易操作了。更多复杂的字符串操作和比较工作也已经为用户完成。程序清单1.4是一个字符串程序的例子, 演示了以上部分功能。

程序清单1.4 字符串程序范例StringTest.cs

```
using System;

class StringTest
{
    public static void Main ()
    {
        string test1 = "This is a test string";
        string test2, test3;

        test2 = test1.Insert(15, "application ");
        test3 = test1.ToUpper();

        Console.WriteLine("test1: '{0}'", test1);
        Console.WriteLine("test2: '{0}'", test2);
        Console.WriteLine("test3: '{0}'", test3);

        if (test1 == test3)
            Console.WriteLine("test1 is equal to test3");
        else
            Console.WriteLine("test1 is not equal to test3");

        test2 = test1.Replace("test", "sample");
        Console.WriteLine("the new test2: '{0}'", test2);
    }
}
```

程序的输出应当如下:

```
C:\>StringTest
test1: 'This is a test string'
```

```
test2: 'This is a test application string'
test3: 'THIS IS A TEST STRING'
test1 is not equal to test3
the new test2: 'This is a sample string'

C:\>
```

C#为每一个新的字符串建立一定数量的内存，因此字符串是不可改变的，即字符串是不能修改的。比如说，读者也许会看到下面的C#代码：

```
string newString = new string("test");
string newString += "ing";
```

变量newString的结果是testing。

有人可能会问：“如果字符串是不可改变的话，怎样修改现有的字符串呢？”对这个问题的回答是，不能修改；C#只不过采取了一些手段。C#并不修改现有的字符串，它只生成了一个带有新值的崭新字符串。为旧字符串保留的内存区现在不再使用了，该内存区会在下一次垃圾回收周期内被清理（CLR自动恢复许多内存）。如果用户在程序中进行大量字符串操作的话，这些操作就会带来许多额外的内存开销。为了补偿此开销，修改字符串对象，微软创建了另外一种类型的字符串类。

StringBuilder类

正如名称所暗示的那样，StringBuilder类在每次重新生成新字符串时没有开销，允许用户生成和修改字符串。它产生一个不确定的字符序列，当字符串被修改时可以动态地改变序列的大小，在需要时分配更多的内存。

字符串使用的内存量称为容量。一个StringBuilder字符串默认的容量通常设置为16字节（StringBuilder文件指出，这个值将来也许会修改）。如果用户建立一个大于16字节的字符串，StringBuilder会自动分配更多的内存。如果用户想精确地控制StringBuilder能够使用的内存量的话，当最初创建实例时，可以用StringBuilder方法和各种StringBuilder构造函数手工地增加或减少字符串的容量。

6种可用于创建StringBuilder实例的构造函数如表1.6所示。

表1.6 StringBuilder类构造函数

构造函数	说明
StringBuilder()	以16字节的大小初始化一个新的默认实例
StringBuilder(int)	用整数（int）容量初始化一个新的实例
StringBuilder(string)	用一个默认字符串的值初始化一个新的实例
StringBuilder(int1, int2)	用默认容量为整数1、最大容量为整数2初始化一个新的实例
StringBuilder(string, int)	用一个字符串的默认值和一个整数容量初始化一个新的实例
StringBuilder(string, int1, int2, int3)	用从字符串整数1位置开始的默认值、整数2的字符长度和整数3的容量初始化一个新的实例

在创建了**StringBuilder**实例之后，用户可以使用一些属性、方法和操作来修改和检查字符串的属性。其中最有用的一个属性就是**Length**（长度），它允许用户动态地修改字符串的容量。程序清单1.5展示了一个用**Length**（长度）属性修改**StringBuilder**字符串容量的例子。

程序清单1.5 **SampleBuilder.cs**程序

```
using System;
using System.Text;

class SampleBuilder
{
    public static void Main ()
    {
        StringBuilder sb = new StringBuilder("test string");
        int length = 0;

        length = sb.Length;
        Console.WriteLine("The result is: '{0}'", sb);
        Console.WriteLine("The length is: {0}", length);

        sb.Length = 4;
        length = sb.Length;
        Console.WriteLine("The result is: '{0}'", sb);
        Console.WriteLine("The length is: {0}", length);

        sb.Length = 20;
        length = sb.Length;
        Console.WriteLine("The result is: '{0}'", sb);
        Console.WriteLine("The length is: {0}", length);
    }
}
```

StringSample程序的输出应当如下：

```
C:\>SampleBuilder
The result is: 'test string'
The length is: 11
The result is: 'test'
The length is: 4
The result is: 'test
The length is: 20

C:\>
```

原字符中的长度是11个字节，但是在把字符串长度设置成4之后，结果字符串就只剩下原字符串的前4个字节了，在把字符串长度设置成20之后，字符串就变成20个字节长了，不过第4个字节后面的原来的数据都丢失了，这些字节的位置用空格补上了。

在用StringBuilder对象生成最终的字符串之后，用户也许需要把它转换成一个字符串对象，发送到网络流中。使用ToString()方法，这是一个很简单过程：

```
string outbound = sb.ToString();
```

现在，这个字符串输出就可以作为一个普通的字符串对象使用了。只不过要记住，它现在是一个不可改变的字符串，应当不被修改（或者要明白，如果修改的话或许要承担额外的开销）。

流（正如前面一段所提到过的）是为了网络编程的需要，用户必须密切关注的C#语言的另外一个特性。下一节将介绍C#的流及其使用。

C#流

数据处理是编程工作中最重要的一项任务。在C#世界，有许多存储和检索数据的方法——文件、内存、输入/输出设备、中间进程通信管道和网络。也有许多从对象读和写数据的方法。大多数对象都允许按照逐个字节的原则读和写数据，这种方法一次只把一个字节的数据传入或传出数据对象。这种方法当然可以工作，但它不是一种最有效的处理数据的方式。

C#提供了一个接口，可以帮助程序员把大块的数据移入或移出数据对象。数据流允许把多字节的数据同时传输到数据对象，这样程序就可以对数据块进行操作，而不需要一次一个字节地构造数据元素。

流支持三种基本操作：

- 把数据从流传输到内存缓冲器（读）
- 把数据从内存缓冲器传输到流（写）
- 从指定的字节模式中搜索流（查找）

并非所有的流都支持以上这些功能。很显然，CD-ROM驱动器就不支持写数据的流，网络连接就不支持查找数据的流。

.NET的System.IO命名空间包含各种各样的流类，可以把数据源的字节结合成容易操作便于管理的块。

FileStream类就是一个用流来简化读写数据操作的极好的例子。这个类提供了一个容易把数据读写到磁盘的流接口。不妨看下面的例子。

假设读者正在写一个把信息记入日志文件的程序，很可能你会一次记录（写出）信息的一个文本行。你会将代码写成在文件中逐个字节地放置每个字符串，并且保证在每个文本行写完时正确地加上回车。然后，当你想用程序读这个日志文件的时候，你就会创建逐个字节读文件的代码，在读文件时你会知道，每个日志文件条目都是用回车结束的，而一个新的条目就会开始。必须对读到的每一个字节进行检查，判断它是否是回车。

为代替这种单调乏味的过程，可以采用先进的FileStream类和StreamWriter类，在日志文件中方便地读写文本行。程序清单1.6显示的范例程序采用流简化了文件的访问。

程序清单1.6 日志写程序范例TestLog.cs

```
using System;  
using System.IO;
```

```
class TestLog
{
    public static void Main ()
    {
        string logFile = "LOGFILE.TXT";

        FileStream fs = new FileStream(logFile,
            FileMode.OpenOrCreate, FileAccess.Write);

        StreamWriter sw = new StreamWriter(fs);
        StreamReader sr = new StreamReader(fs);

        sw.WriteLine("First log entry");
        sw.WriteLine("Second log entry");

        while(sr.Peek() > -1)
        {
            Console.WriteLine(sr.ReadLine());
        }

        sw.Close();
        sr.Close();
        fs.Close();
    }
}
```

注意这个例子中的下列问题：

- **FileStream**对象既用于从流读出数据也用于把数据写入流。
- **StreamReader**和**StreamWriter**对象都引用同一个**FileStream**，但它们执行不同的功能。
- 每个流对象都有它自己的流指针。在**StreamWriter**对象在它的流中插入两个新行之后，**StreamReader**对象读它流中的第一个对象，就是文件的第一行。
- 每个打开的流必须明确地关闭，包括基本的**FileStream**对象。许多编程新手会忘记关闭基本流，因疏忽使它悬挂起来。

最常见的流技术是为读和写创建两个单独的流：

```
StreamWriter sw = new StreamWriter(fs);
StreamReader sr = new StreamReader(fs);
```

这使用户可以用单独的流完全控制对于流的数据访问。

这个程序没有完成的一件事情是，对文件打开进行错误检查。如果程序无法打开日志文件，它就会向用户显示一条难看的错误信息。下一节将介绍怎样在C#程序中得体地处理错误情况。

C#异常编程

对程序中不正常情况的处理是程序员面临的一大难题。缺乏经验的程序员往往忘记弥补错误环境，例如被零除的情况。为了保证编写的代码是用户友好的，应当努力弥补大部分

类型的错误环境。这样的错误环境或者在程序执行过程出现的意想不到的行为就叫做异常(exception)。程序清单1.7显示了一个例子。

程序清单1.7 BadError.cs程序

```
using System;

class BadError
{
    public static void Main ()
    {
        int var1 = 1000, var2 = 0, var3;

        var3 = var1 / var2;
        Console.WriteLine("The result is: {0}", var3);
    }
}
```

可以看出, 这个程序从一开始就注定是有问题的, 由于用零作除数的错误, 第6行肯定是要造成灾难的。编译和运行这个例子, 看看会出现什么问题:

```
C:\>csc BadError.cs
Microsoft (R) Visual C# .NET Compiler version 7.00.9466
for Microsoft (R) .NET Framework version 1.0.3705
Copyright (C) Microsoft Corporation 2001. All rights reserved.

C:\>BadError

Unhandled Exception: System.DivideByZeroException: Attempted to divide by zero.
   at BadError.Main()

C:\>
```

csc编译器编译这个代码没什么问题, 它对眼前的错误并不在意。当程序运行的时候, 一个窗口弹了出来, 表示出现了一个错误, 问用户是否想调试该应用程序。这正是用户不想看到的事情。在单击了OK按钮之后, 在控制台上产生了文本的错误信息, 指出碰到的是什么错误, 程序停在产生问题的代码行上, 不再执行其他行。

C#提供了一种当异常出现时发现和捕获异常的方法, 帮助程序员编制处理异常的代码。用捕获异常的方法, 程序员可以有效地提供可读的错误信息, 程序继续执行, 或者也可以在必要时得体地终止程序。在C#中, try-catch块便可以做到这一点。

try-catch程序块执行一条或更多的语句。如果任何语句产生了异常, 就执行代码的catch块, 而不是停止执行程序。根据catch块中包含的内容, 程序可能会得体地终止, 如果是无关紧要的错误就允许程序继续执行:

```
try
{
    // one or more lines of code
```



```
    }  
    catch ()  
    {  
        // one or more lines of code to execute if an error  
    }  
    finally  
    {  
        // one or more lines of code to execute at all times  
    }
```

在程序过程正常的时候，执行try块中的语句。如果在try块中出现了错误情况，程序就转移到catch块中。在执行了catch块中的代码之后，控制权又交回给try-catch块后面的主程序语句，就好像没有发生过错误一样（除非catch块中的代码过程停止或者程序的执行改变了方向）。

用户可以选择在try或者catch块结束之后增加一个finally块。有时，这个finally块用于清除那些不管成功还是失败都必须运行的函数。不过，在大多数情况下都不用finally。

说明：注意在catch语句中的圆括号，这是一个过滤器，允许用户指定想要捕获的异常类型。用户可以根据try块中完成的动作类型定义指定的异常，或者也可以定义普通的Exception类，它会捕获所有出现的异常。用户甚至可以定义多个catch块。catch块是按顺序评估的，因此，应当把特殊异常列在常规异常的前面。

程序清单1.8显示了一个简单的try-catch块的例子。

程序清单1.8 CatchError.cs程序

```
using System;  
  
class CatchError  
{  
    public static void Main ()  
    {  
        int var1 = 1000, var2 = 0, var3;  
  
        try  
        {  
            var3 = var1 / var2;  
        }  
        catch (ArithmeticException e)  
        {  
            Console.WriteLine("Exception: {0}",  
                               e.ToString());  
            var3 = -1;  
        }  
        catch (Exception e)  
        {
```

```
        Console.WriteLine("Exception: {0}",  
            e.ToString());  
        var3 = 2;  
    }  
    Console.WriteLine("The result is: {0}", var3);  
}  
}
```

在CatchError程序中，原程序由于增加了try-catch块而被修改。程序中有两个catch语句：一个专门注意ArithmeticExceptions，另一个注意所有普通的Exception。注意，首先列出特殊的ArithmeticException，否则的话，普通Exception块就会在更详细的异常出现之前处理所有异常。还要注意，catch块为结果变量设置了一个值，以后可以在程序中检查结果变量的值。这个程序运行的结果如下：

```
C:\>csc CatchError.cs  
Microsoft (R) Visual C# .NET Compiler version 7.00.9466  
for Microsoft (R) .NET Framework version 1.0.3705  
Copyright (C) Microsoft Corporation 2001. All rights reserved.  
  
C:\>CatchError  
Exception: System.DivideByZeroException: Attempted to divide by zero.  
    at CatchError.Main()  
The result is: -1  
  
C:\>
```

try-catch块从容地捕获算法异常，按照catch块中的指令显示错误，把变量设置成以后在程序逻辑中可以检查得到的指定值。

说明：在try-catch块中经常使用网络功能。在网络编程中，通常不可能判断一个行为是否会成功。例如，由于路由器或目的主机的停机故障，网上发出去的包也许到达不了预定的目的地。可以把问题报告用户，而不要由于异常的网络事件而中断程序。

小结

微软的.NET技术给网络程序员提供了极好的编程环境。C#编程语言使程序员们能够在短时间里容易地编制出标准的应用程序，并使他们设计出具有Windows网络功能优势的健壮网络应用程序。

用两个全面的微软Visual.NET软件产品或者新的.NET框架SDK软件包，创建一个C#开发环境是很容易的。这三个环境都包括创建C#应用程序的编译器和调试器。

C#语言还提供了许多功能，为程序员编写网络应用程序排除了困难。例如，用StringBuilder类来代替创建和修改字符串对象，可以节省一些应用程序的执行时间。另外，在网络编程中使用流操作可以简化网络传输过程中的数据处理。在程序中使用异常可以使用

户的应用程序看起来更具专业水准，并避免了许多客户的担忧。

本章讨论了C#语言的基础。读者会在第3章读到更多的基础知识，本书将在那里介绍网络编程使用的特殊C#类。由于增加了网络编程helper（助手）类，C#使得网络编程可以轻松完成。可是，首先要在第2章对IP编程的基本原则进行有益的回顾。

第2章 IP编程基础

- 监视网络通信量
- 分析网络信息包
- 用TCP和UDP编程
- 查找IP地址信息

因特网协议 (Internet Protocol, IP) 是网络编程的核心。不论在局域网 (LAN) 环境还是在广域网环境, IP都是系统之间传输数据的媒介。虽然还有其他Windows网络程序员可用的网络协议, 但对于在网络设备之间传送数据 (尤其当设备位于因特网上时) IP提供了最健壮的技术。

用IP编程通常是一个复杂的过程。对于怎样在网上发送数据有许多要考虑的因素: 客户和服务器的数量、网络的类型、网络的拥塞程度和网络上的错误情况。因为所有这些因素都会影响用户数据从一个设备传输到另一个设备, 理解它们的影响对于成功地进行网络编程是至关重要的。经常, 用户在程序中想到会发生的问题并没有在网上发生。本章将介绍在网上监视IP通信量的方法, 帮助用户调试自己的网络程序。

本章还要探讨为了理解网络通信必须了解的有关IP通信的内容, 并介绍使用IP的两个最流行协议的内部工作方式: 即传输控制协议 (Transmission Control Protocol, TCP) 和用户数据报协议 (User Datagram Protocol, UDP)。最后介绍的是最容易使人糊涂的IP网络编址技术。本章以微软Windows系统IP地址处理结束, 告诉读者怎样编程确定程序运行系统的IP地址结构。

监视网络通信量

网络程序员面临的最大困难是不可能确切地知道网上发生了什么事情。情况经常是这样, 用户已经创建了理想的客户机/服务器应用程序, 花费很多时间在机器之间发送和接收应用程序的数据, 只是为了找出网络上某个用户数据不能通过的位置。网络分析器是网络程序员最好的朋友, 能帮助发现“线上”正在发生的事情。通过监视应用程序数据从一个设备到另一个设备的传送情况, 通常能探测出编程错误。

本节将向读者介绍, 怎样在Windows工作站或服务服务器上安装一个简单的公用域网络分析器, 帮助排除网络编程中出现的问题。

说明: 如果用户正在Windows NT 4或2000服务器上开发C#程序, 那在自己的系统中就已经建立了一个网络分析器程序netmon了。该程序提供了一种方法, 可以捕获和观看进入和离开服务器的信息包。很可惜, 在其他的Windows平台上不能使用这个程序。所以, 为了对所有的C#开发平台保持一致, 本章讨论在所有运行.NET框架的Windows平台上都可以操作的公共域网络分析器的实现。这个分析器就是WinPcap驱动程序和它的附加工具, WinDump和Analyzer。

安装WinPcap程序

意大利都灵市的NetGroup开发了WinPcap驱动程序，允许Windows系统捕获网络包。NetGroup已经向使用WinPcap驱动程序和所有在Windows系统工作的公共域推出了一整套网络监视工具。这些工具允许网络程序员探索网络的“罩子下面”，看发生了什么事情。可用的网络工具有WinDump和Analyzer，WinDump是一个命令行的包捕获工具，Analyzer是一个基于Windows的包捕获工具。

WinPcap驱动程序是该软件包的核心。它允许Windows工作站或服务器上的大多数类型的网卡用混杂的方式放置。在混杂方式中，网卡可以接收到网上所有的通信量，不管包是发到什么地方去的（通常，网卡驱动程序只接受发送到某一个设备地址去的包）。WinDump和Analyzer程序可以把网络数据显示成可读的Ethernet（以太网）包和IP包，当包在网络设备间传递的时候使用户可以看到用户应用程序的数据。

WinPcap驱动程序和网络工具可以从<http://netgroup-serv.polito.it/netgroup/tools.html>的NetGroup主页下载得到。每个软件包工具都有它自己的主页，可以从主工具的Web页面引用。每个软件包都必须单独下载和安装。

在写本书的时候，WinPcap驱动程序的当前版本是2.3版。在下载了安装文件之后（当前为winpcap_2_3.exe），从Windows资源管理器中双击该文件。这时会出现一个安装窗口，引导用户通过简单的安装过程。WinPcap驱动程序文件全都存放在预先设定的位置，所以在安装时不需要用户做任何选择。如果用户想卸载WinPcap驱动程序（建议在升级到新版本之前进行卸载），可以从控制面板中的Add/Remove（添加/删除）程序中选择WinPcap软件，再选择删除选项。

警告：如果用户碰巧发现一个较早版本的WinPcap驱动程序，一定要注意，2.3版是第一个保证能在Windows XP系统中正确操作的版本。

在WinPcap驱动程序加载之后，就需要从各自的主页安装WinDump和Analyzer程序，在写本书的时候，最新的版本是WinDump 3.6.2和Analyzer 2.2。WinDump程序作为一个可以运行的单独可执行文件下载，可以把它存储在用户系统的任何地方。Analyzer程序作为一个自提取（self-extracting）安装程序下载，必须把Analyzer程序和支持文件都放到一个单独的目录中运行。

在用户下载并安装了WinPcap驱动程序和网络监视工具之后，就可以开始监视网络上的IP通信量了。下一节将告诉读者，怎样使用WinDump命令行网络监视器监视用户的数据。

WinDump程序

如果读者在UNIX系统上工作过，一定对tcpdump程序非常熟悉，它是UNIX环境中一个流行的基于文本的网络监视工具。它能够显示在网上看到的每一个包内容的摘要，或者显示每个包详细内容的十六进制的转储。tcpdump用简单的命令行选项来判断，需要捕获哪种类型的包。WinDump程序的目标是在Windows的命令行环境中重新建立tcpdump程序的外观和感觉。

命令行选项

使用WinDump工具的第一步是要确定，到底要监视哪一个网络接口。许多计算机都有几个网络接口，这些接口包含多个网卡和调制解调器，拨号进入使用点对点协议（Point-to-Point Protocol, PPP）的因特网服务提供商（Internet Service Providers, ISP）。为了看到用户系统中当前网络接口的列表，可以在WinDump命令行使用-D选项：

```
C:\monitor>windump -D
1.\Device\Packet_{E0D13BFC-D26F-45D6-BC12-534854E3AD71} (Novell 2000 Adapter.)
2.\Device\Packet_NdisWanIp (NdisWan Adapter)
3.\Device\Packet_NdisWanBh (NdisWan Adapter)

C:\monitor>
```

这里举例的工作站的配置包括一个网卡（模仿一个Novell 2000网络适配器）、一个带有两个单独PPP连接配置的调制解调器。在默认情况下，WinDump会监视1号接口的通信量。如果用户想监视其他接口的通信量，必须在命令行用-i选项指定：

```
C:\monitor>windump -i 2
windump: listening on \Device\Packet_NdisWanIp
```

WinDump程序是一个多用途的程序。正如读者可以猜想到的，它有许多可用来更改它行为的命令行选项。表2.1显示了用户可能会经常需要用的部分选项。

表2.1 WinDump命令行选项

选项	说明
-a	试图把网络和广播地址转换成名字
-B size	把接收缓冲器的尺寸设置为size字节
-c count	只捕获count数量的包
-D	显示系统中所有可用的网络接口
-e	在输出的每一行打印链接级的信息
-F file	从文件名file中读取过滤器的表达式
-i interface	监视网络的interface接口，interface可能是一个接口名，也可能是-D命令显示的一个编号
-n	指定不把地址转换为名字
-N	指定不打印完全合格的域名
-q	打印快速的（较少的）包信息
-r file	从转储文件file中读取包
-S	打印绝对的TCP顺序号
-s snaplen	从包中捕获snaplen个字节，默认值是68

(续表)

选项	说明
-t	指定不在每一行打印时间标记
-w file	把输出内容写到file中
-X	用十六进制和ASCII码打印每个包
-x	用十六进制打印每个包

可以在命令行组合多个选项来创建用户需要的网络监视环境。例如，下列命令可以捕获每个包的前200个字节，用十六进制打印它们，并把输出写到一个文件中：

```
C:\monitor>windump -s 200 -x -w testcap
```

过滤器表达式

在默认情况下，WinDump会试图捕获在网络接口上它看到的所有的包。根据用户的网络情况（和分析工作站在网络上的位置），“所有的包”也许能表现网络真实的通信量。通常在一组不相关的网络包中很难跟踪单独的IP会话。用户可以用WinDump来指定一个过滤器，把捕获的通信量减少到只监视用户感兴趣的信息。

WinDump用缩写法定义过滤器。过滤器表达式（filter expression）详细说明了想让WinDump捕获的网络通信量。根据需要，用户可以用不同的过滤器表达式，命令WinDump监视各种包特性。

过滤器表达式由一个或多个基本要素（primitive）组成，一个基本要素描述了过滤器的一个特殊项目，由网络名或数字加上一个或多个限定词（qualifier）组成。有三种类型的限定词：

- 用ID名或编号引用的对象类型
- 与过滤对象有关的包流动方向
- 一个过滤器寻址的特殊协议

在实践过程中，使用这些限定词非常简单。如果用户只对看到网络上的IP包感兴趣，可使用下列命令：

```
windump ip
```

这个命令捕获所有的IP包，不管它的源地址或目的地址是什么。如果想看到指定IP地址的通信量，可使用以下命令：

```
windump ip host 192.168.1.6
```

这使得WinDump只捕获与主机192.168.1.6相关的IP包。默认情况下，WinDump既捕获来自指定设备的包也捕获进入该指定设备的包。如果用户只想看到来自那个地址的包，可以加上一个方向限定词：

```
windump ip src 192.168.1.6
```


以上命令使WinDump只捕获来自192.168.1.6设备（或源）的IP包。不捕获到该地址去的答复包。

用户也可以指定网络地址，捕获来自一个指定子网的所有主机的包：

```
windump ip host 192.168.1
```

这个命令捕获来自192.168.1.0子网上所有主机的IP包。

警告：用户必须清楚自己开发网络应用程序的网络类型。如果开发工作站被插在一个网络交换机里，用户将看不到来自网上其他设备的任何通信量，因为交换机会阻挡通信量。通常，当打算调试网络应用程序的时候，最好使用网络集线器而不要使用交换机，这样才可以看到所有的网络通信量。

运行WinDump

windump命令的输出显示出每个包捕获到的必要的信息。在默认情况下，WinDump将试图判断设备的网络名，并显示所有使用该设备的网络名而不是IP地址的网络包。用户可以根据两台通信设备的设备名来跟踪网络会话。程序清单2.1显示了一个Telnet会话的范例。

程序清单2.1 WinDump会话范例

```
C:\monitor>windump
windump listening on \Device\NPF{E0D13BFC-D26F-45D6-BC12-534854E3AD71}
18:46:49.583176 arp who-has SHADRACH tell ABEDNEGO
18:46:49.583677 arp reply SHADRACH is-at 0:e0:7d:74:df:c7
18:46:49.583717 ABEDNEGO.1037 > SHADRACH.23: S 334792806:334792806(0) win 16384
<mss 1460,nop,nop,sackOK> (DF)
18:46:49.584169 SHADRACH.23 > ABEDNEGO.1037: S 1564900369:1564900369(0) ack 334
92807 win 32120 <mss 1460,nop,nop,sackOK> (DF)
18:46:49.584271 ABEDNEGO.1037 > SHADRACH.23: . ack 1 win 17520 (DF)
18:46:49.996842 SHADRACH.23 > ABEDNEGO.1037: P 1:13(12) ack 1 win 32120 (DF)
18:46:49.997496 ABEDNEGO.1037 > SHADRACH.23: P 1:7(6) ack 13 win 17508 (DF)
18:46:49.997955 SHADRACH.23 > ABEDNEGO.1037: . ack 7 win 32120 (DF)
18:46:49.998081 SHADRACH.23 > ABEDNEGO.1037: P 13:16(3) ack 7 win 32120 (DF)
18:46:49.998174 ABEDNEGO.1037 > SHADRACH.23: P 7:16(9) ack 16 win 17505 (DF)
18:46:49.998657 SHADRACH.23 > ABEDNEGO.1037: P 16:28(12) ack 16 win 32120 (DF)
18:46:49.998986 ABEDNEGO.1037 > SHADRACH.23: P 16:25(9) ack 28 win 17493 (DF)
18:46:50.002249 SHADRACH.23 > ABEDNEGO.1037: . ack 25 win 32120 (DF)
18:46:50.002334 ABEDNEGO.1037 > SHADRACH.23: P 25:41(16) ack 28 win 17493 (DF)
18:46:50.012285 SHADRACH.23 > ABEDNEGO.1037: . ack 41 win 32120 (DF)
18:46:50.177333 SHADRACH.23 > ABEDNEGO.1037: P 28:40(12) ack 41 win 32120 (DF)
18:46:50.177966 ABEDNEGO.1037 > SHADRACH.23: P 41:44(3) ack 40 win 17481 (DF)
18:46:50.192238 SHADRACH.23 > ABEDNEGO.1037: . ack 44 win 32120 (DF)
18:46:50.192334 ABEDNEGO.1037 > SHADRACH.23: P 44:53(9) ack 40 win 17481 (DF)
18:46:50.193672 SHADRACH.23 > ABEDNEGO.1037: P 40:114(74) ack 53 win 32120 (DF)
18:46:50.194002 ABEDNEGO.1037 > SHADRACH.23: P 53:56(3) ack 114 win 17407 (DF)
```

```
18:46:50.212238 SHADRACH.23 > ABEDNEGO.1037: . ack 56 win 32120 (DF)
18:46:50.212437 ABEDNEGO.1037 > SHADRACH.23: P 56:59(3) ack 114 win 17407 (DF)
18:46:50.232199 SHADRACH.23 > ABEDNEGO.1037: . ack 59 win 32120 (DF)
18:46:50.753865 SHADRACH.23 > ABEDNEGO.1037: P 114:121(7) ack 59 win 32120 (DF)
18:46:50.859647 ABEDNEGO.1037 > SHADRACH.23: . ack 121 win 17400 (DF)
```

WinDump输出的每一行表示一次单独的网络包捕获。前两个包显示该客户工作站（abednego）用ARP协议寻找服务器（shadrach）的网址。在设备确定了正确的地址之后，开始了IP会话。

WinDump输出的每个IP行都包含以下信息：

- 时间标记
- 源IP地址（或主机名）和TCP或UDP端口
- 目的IP地址（或主机名）和TCP或UDP端口
- TCP或UDP包信息

WinDump的捕获结果用文本方式列表显示每个网络包捕获的相关信息。如果希望看到更详细的信息，可以用-s命令增加包捕获的长度，也可以用-X命令把数据用十六进制和ASCII码打印出来。下一节将介绍WinDump的伙伴程序Analyzer怎样提供同样的包信息，但却采用了更容易读的图形格式。

Analyzer程序

Analyzer程序提供了捕获和分析网络包的图形环境。它与WinDump程序具有同样的功能，只是具有更加方便的用户界面。

为了启动Analyzer程序，可以双击analyzer.exe文件，或者如果用户选择在安装时创建它，也可以单击Analyzer（分析器）桌面图标。一个空白的Analyzer窗口应当出现，如图2.1所示。

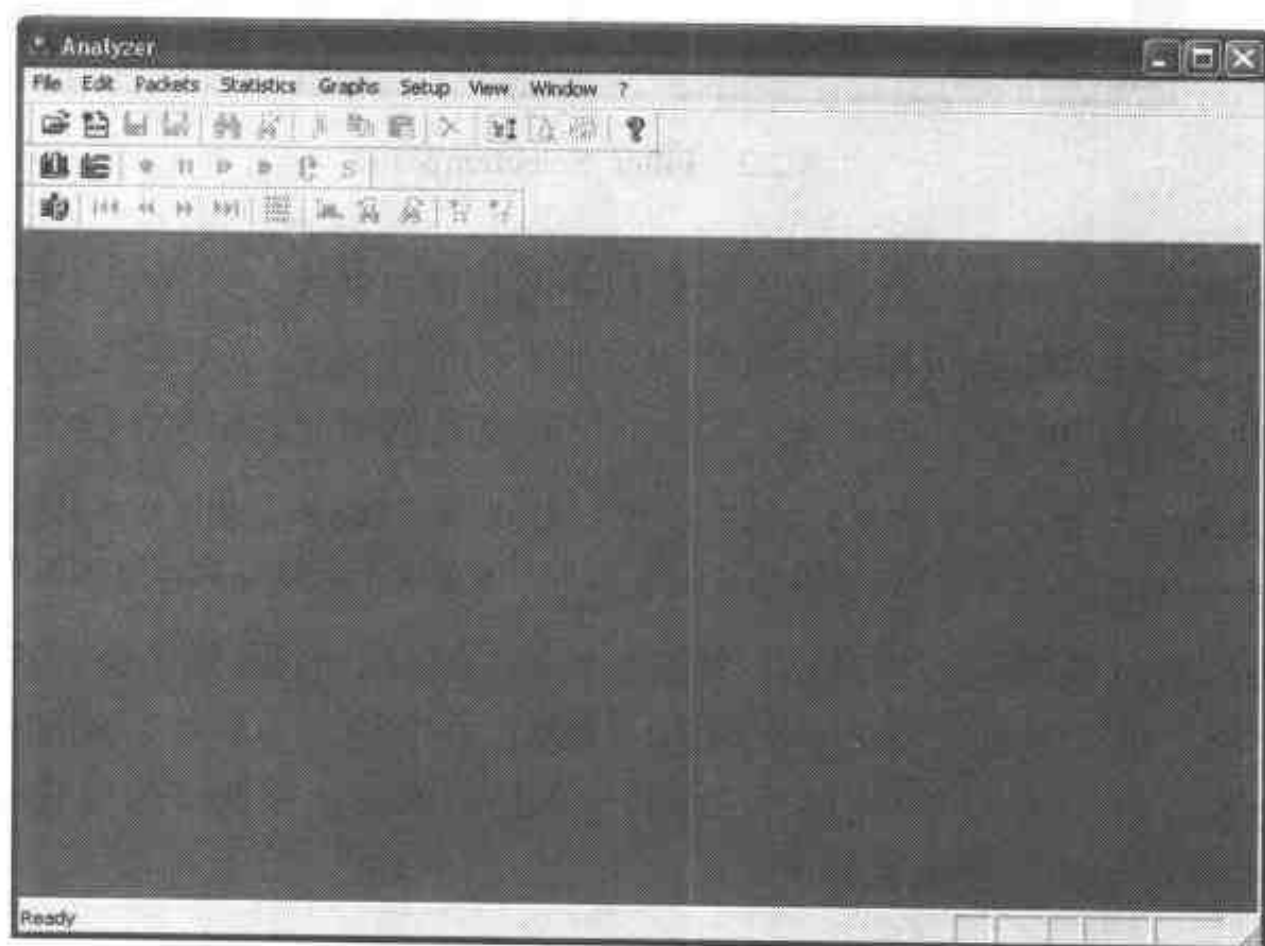


图2.1 Analyzer窗口

Analyzer程序可以执行以下4种基本功能：

- 捕获并显示网络包
- 显示存储在一个文件中的包
- 捕获网络统计资料
- 执行实时网络监视

由于本节所讨论的要点是捕获网络包的问题，因此在这里就不再描述Analyzer程序的网络统计和实时监视功能。不过，这些功能对于网络故障诊断是很有用的，读者应当自己研究掌握它们。

要捕获网络包，必须单击包捕获图标，它是工具条第三行的第一个图标。当单击这个图标之后，Filter Selection（过滤器选择）窗口出现了，如图2.2所示。

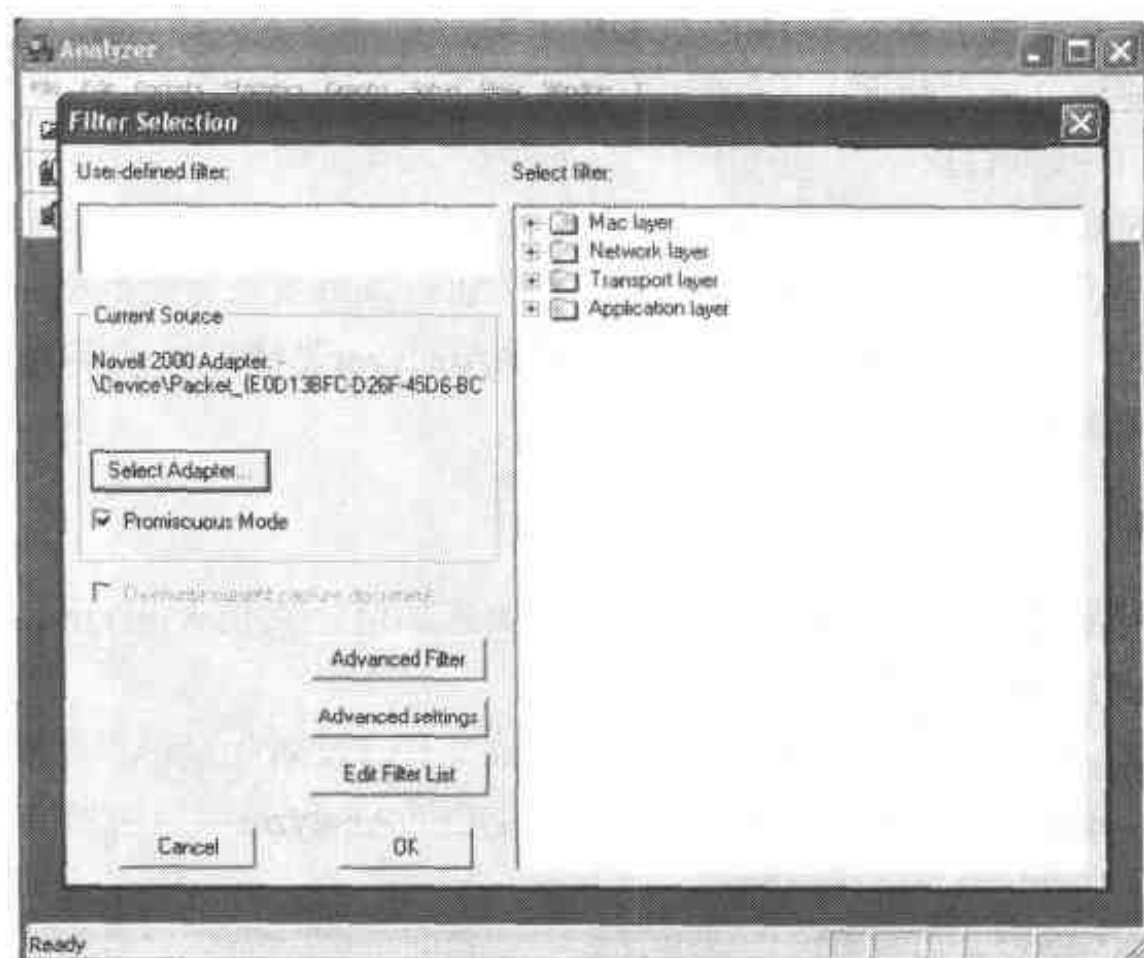


图2.2 Filter Selection窗口

Filter Selection窗口允许用户选择捕获包的接口和定义一个用于包捕获的过滤器。用户可以单击Select Adapter按钮来选择使用哪一个网络适配器，应当出现一个与windump -D命令行选项所得到的相同的列表。另外，用户定义的所有PPP连接也会在列表中显示。

警告：监视PPP连接的时候一定要注意，在写本书的时候，WinPcap驱动程序的2.3版在Windows 2000和XP的PC机上监视PPP连接还有一些困难。希望这个问题在后来的版本中得到解决。

如果想捕获全部的网络包，就必须选择Promiscuous Mode复选框。在选择了使用的网络适配器之后，也许要定义一个要使用的过滤器。在右边的窗口中，Analyzer程序显示出几个普通过滤器的树。用户可以用展开一个特定网络层的方法来选择要捕获的特定包类型。图2.3显示了一些可用的过滤器选项。

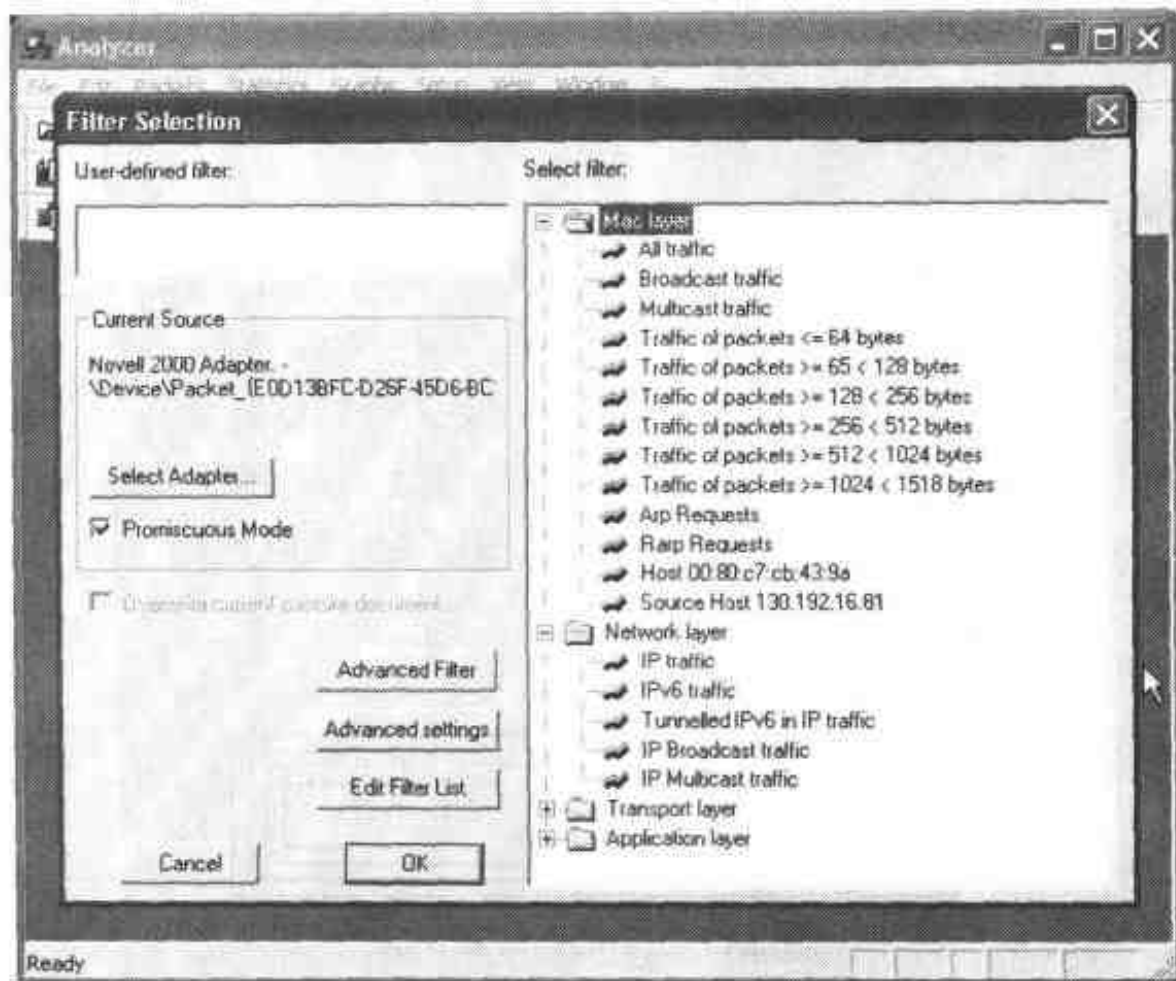


图2.3 Analyzer Filter Selection选项

在选择了想要的过滤器选项和单击了OK按钮之后，Analyzer程序就开始捕获包。这时，Capture in Progress（捕获正在进行）窗口出现，显示出捕获进行了多长时间、已经有多少个包被过滤器捕获、多少个包已经被Analyzer分析以及有多少个包已经被丢失（抛弃）。如果要结束捕获过程，可以按Stop（停止）按钮。

当停止捕获时，会出现一个Capture（捕获）文档窗口（读者可以从图2.5、图2.6和图2.7中看到该窗口的例子）。这个窗口分三个部分，说明如下：

Packet index 这是窗口顶部的一个框架，按顺序显示所有捕获到的包。

Packet details 这是包类型信息的一个树形视图，显示在窗口的左下角。该框架给出有关包的详细信息，根据包数据中出现的各种协议分成好几部分。例如，对于典型的TELNET会话包，会出现下列协议：

- Ethernet（以太网）层传输信息
- IP网络层信息
- TCP传输层信息
- TELNET应用程序数据

当前包的十六进制和ASCII打印输出 这个信息在窗口右下角的框架中。它显示出整个包的未经加工的原信息（用户也许需要上下或左右滚动窗口才能看到全部信息）。信息包括两部分：包中每个字节的原十六进制表示，和每个字节的ASCII代码。这可以帮助用户方便地解码包的数据部分。

为了成功地跟踪和调试网络应用程序，应当知道怎样解码和理解网络包中的每个信息层。下一节将探讨层并告诉读者怎样解码层的信息。

分析网络包

监视IP会话和译解会话的含义是一名网络程序员极其重要的技能。为了充分理解网络编程后面的概念，首先必须理解IP协议和它是怎样在网络设备之间传送数据的。熟悉这些知识可以节省程序员大量查找网络程序故障的时间，这些程序没有按程序员设想的行为方式工作。

用户捕获的网络包包含好几层信息，帮助数据在两个网络设备之间传输。每一层信息包含着按预定顺序安排的字节，指定了该协议特有的参数。在IP网络编程中用户使用的大多数包都将包含三个单独的信息协议层和网络设备之间传输的实际数据。图2.4表示了IP包使用的协议层，正如下面几节所讨论的。

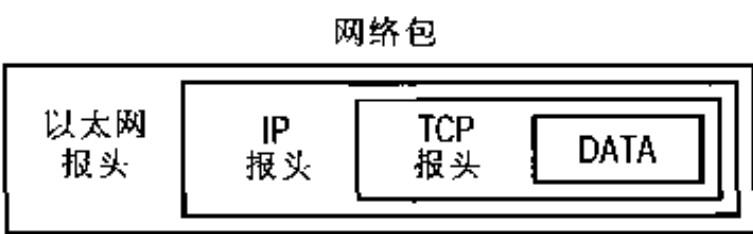


图2.4 包中的网络协议层

Ethernet层

在Analyzer捕获结果中显示的第一个网络协议层称为Ethernet header（以太网报头）。在用户的网络中可能会看到三种类型的Ethernet（以太网）协议：Ethernet 802.2、Ethernet 802.3和Ethernet第2版。

Ethernet 802.2和802.3是为以太网层通信量而定义的IEEE标准协议。Ethernet第2版是一个传统的协议，本身不是一个标准协议，但它是以太网上用得最普遍的协议。默认情况下几乎所有的设备（包括Windows系统）都使用Ethernet第2版协议来传输IP包。读者可以在图2.5中看到Packet Details（包详细资料）框架。

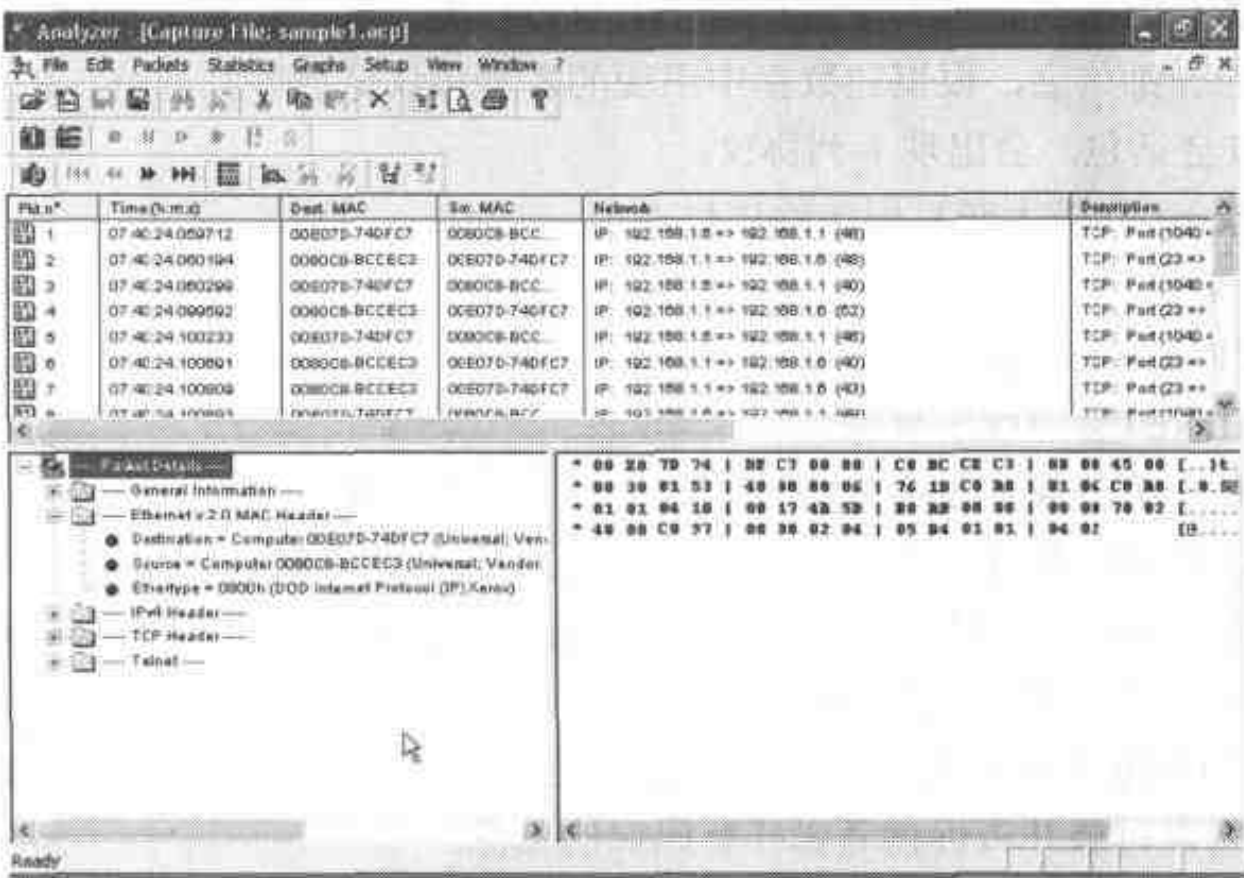


图2.5 在分析器Packet Details部分中的以太网报头数据

Ethernet第2版报头显示出在以太网上用于识别单个设备的介质访问卡（Media Access Card, MAC）地址和一个用来标识以太网包中包含的下一层协议的以太网协议号码。每一个以太网包都遵守预先确定的方案：

- 6字节的目的地以太网（或MAC）地址
- 6字节的源以太网（或MAC）地址
- 2字节的下一层协议标识符
- 46字节到1500字节的数据有效载荷
- 4字节的检验和（checksum）

数据有效载荷必须至少包含46个字节，以保证以太网包至少为64个字节长（这有助于防止冲突，因为发送较短的包会在包发送之后在网络上引起冲突，而发送设备不会知道）。如果数据不够46个字节长，就要加一些填充字符。检验和（checksum）值提供一个网上数据初步的错误检查机制。如果包在传输过程中被破坏，算出来检验和的值就不正确，就给这个包做上坏包的标记。

下面我们仔细地讨论一下在以太网报头中需要检查的两个最重要的值：以太网地址和协议标识符。

以太网地址

设备的以太网地址是由网卡销售商分配的，不能被修改（即使某些场合软件允许不考虑这个地址）。以太网上的每个设备必须有一个惟一的MAC地址。MAC地址由两部分组成：

- 3字节的销售商标识符，或机构惟一标识符（Organizationally Unique Identifier, OUI）
- 对销售商惟一的3字节的设备序号

因此，不管网络有多大，这6个字节的网址就惟一地标识了网上的网卡。为了保证标识符不重复，因特网编号分配管理机构（Internet Assigned Numbers Authority, IANA）给一些销售商们指派了MAC地址。分配的编号发表在因特网请求注释（Request For Comments, RFC）文档中（最新分配的RFC编号是RFC 1700）。IANA还维持着一个Web页面（www.iana.org），页面上包含有大部分当前以太网销售商的信息。表2.2列出了一些销售商MAC地址的标识符。

表2.2 销售商的MAC标识符

标识符	销售商
00000C	Cisco
00001D	Cabletron
00002A	TRW
00005E	IANA
000065	Network General
0000A2	Wellfleet

(续表)

标识符	销售商
0000A6	Network General (内部分配, 不用于产品)
0000A9	Network Systems
0000AA	Xerox machines
0000C0	Western Digital
0000E2	Acer Counterpoint
0020AF	3COM
008064	Wyse Technology / Link Technologies
0080C2	IEEE 802.1 Committee
0080D3	Shiva
00AA00	Intel
00DD00	Ungermann-Bass
00DD01	Ungermann-Bass
020701	Racal InterLan
02608C	3Com IBM PC; Imagen; Valid; Cisco
080002	3Com (以前的Bridge)
080009	Hewlett-Packard
08000B	Unisys
080011	Tektronix, Inc.
08001A	Data General
08001B	Data General
08001E	Apollo
080020	Sun machines
08002B	DEC
08002F	Prime Computer Prime 50-Series LHC300
080046	Sony
080047	Sequent
08004C	Encore
08005A	IBM
080069	Silicon Graphics
800010	AT&T

在3个字节的销售商部分之后加上3个字节的惟一序号就产生了惟一的6字节的地址。用户可能看到几种引用以太网地址的方法；下面是几种传统的格式：

- 作为单一的值：0020AFCCEC3
- 用冒号：00:20:AF:BC:CE:C3
- 用连字符：00-20-AF-BC-CE-C3
- 用一个连字符：0020AF-BCCEC3（如Analyzer程序中使用的那样）

除了单独的设备地址外，以太网编址方案还允许广播和组播地址。在广播地址（broadcast address）中，所有的目的地址位都设置为1（用十六进制的FFFFFFFFFFFF）。以太网上的每一个设备都应当接受地址为以太网广播地址的包。这个地址对于必须把查询包送到网上每个设备的协议是非常有用的；例如，ARP（地址解析协议），用于当工作站寻找使用某个特定IP地址的设备的时候。

组播地址（multicast address）是一种专用形式的以太网地址；只有网络设备的子集会接受这个包。网络设备上的以太网软件必须配置成可以接受这种特殊组播地址才行。表2.3是一些以太网组播地址的列表。

表2.3 以太网组播地址

地址	说明
01-80-C2-00-00-00	生成树（用于网桥）
09-00-09-00-00-01	HP探针
09-00-09-00-00-01	HP探针
09-00-09-00-00-04	HP DTC
09-00-2B-00-00-00	DEC MUMPS
09-00-2B-00-00-01	DEC DSM/DTP
09-00-2B-00-00-02	DEC VAXELN
09-00-2B-00-00-03	DEC局域网网桥通信量监测器（LTM）
09-00-2B-00-00-04	DEC MAP终端系统Hello
09-00-2B-00-00-05	DEC MAP中间系统Hello
09-00-2B-00-00-06	DEC CSMA/CD加密
09-00-2B-00-00-07	DEC NetBios仿真器
09-00-2B-00-00-0F	DEC本地区域传输（LAT）
09-00-2B-00-00-1x	DEC实验的
09-00-2B-01-00-00	DEC局域网网桥复制包（所有网桥）
09-00-2B-02-00-00	DEC DNA Lev.2路由层路由器
09-00-2B-02-01-00	DEC DNA命名服务广告

(续表)	
地址	说明
09-00-2B-02-01-01	DEC DNA命名服务请求
09-00-2B-02-01-02	DEC DNA时间服务
09-00-2B-03-xx-xx	DEC默认用网桥过滤
09-00-2B-04-00-00	DEC本地区域系统传输 (LAST)
09-00-2B-23-00-00	DEC Argonaut控制台
09-00-4E-00-00-02	Novell IPX
09-00-77-00-00-01	Retix生成树网桥
09-00-7C-02-00-05	Vitalink诊断
09-00-7C-05-00-01	Vitalink网关
0D-1E-15-BA-DD-06	HP
CF-00-00-00-00-00	以太网配置测试协议 (回送)

以太网协议类型

以太网报头另外一个重要的部分是Protocol Type字段。Ethernet第2版的包与Ethernet 802.2、Ethernet 802.3的包之间的区别表现在Protocol Type字段。Ethernet 802.2、Ethernet 802.3的包都在包中相同的位置含有一个2字节的值，它们都用这个值表示以太网包的总尺寸。Ethernet第2版使用相同的2个字节指出以太网包中包含的下一级包使用的协议类型（见图2.5）。这使得网络应用程序能迅速地识别出包数据的下一层协议。

类似于以太网销售商地址的是IANA给各种协议指定的值，根据这些值可以在以太网协议类型字段中惟一地识别出各种协议。表2.4列出了部分指定的协议值。

表2.4 以太网协议值

值	协议	值	协议
0800	IP	80D5	IBM SNA
0806	ARP	8137	Novell
0BAD	Banyan VINES	8138	Novell
8005	HP Probe	814C	Raw SNMP
8035	Reverse ARP	86DD	IPv6
809B	AppleTalk	876B	TCP/IP压缩

说明：由于本书要研究的课题是用IP协议在网上传输数据，因此读者会在本书范例的所有捕获包的协议类型字段中看到0800这个值。这个值表示下一层的协议是IP。

IP层

Analyzer捕获包的下一个协议层是IP包。在图2.6中，包的详细框架显示出某个包的Analyzer捕获的IP层部分。从图中可以看出，IP协议对于以太网协议信息定义了几个额外的字段。表2.5描述了Analyzer捕获的IP字段。

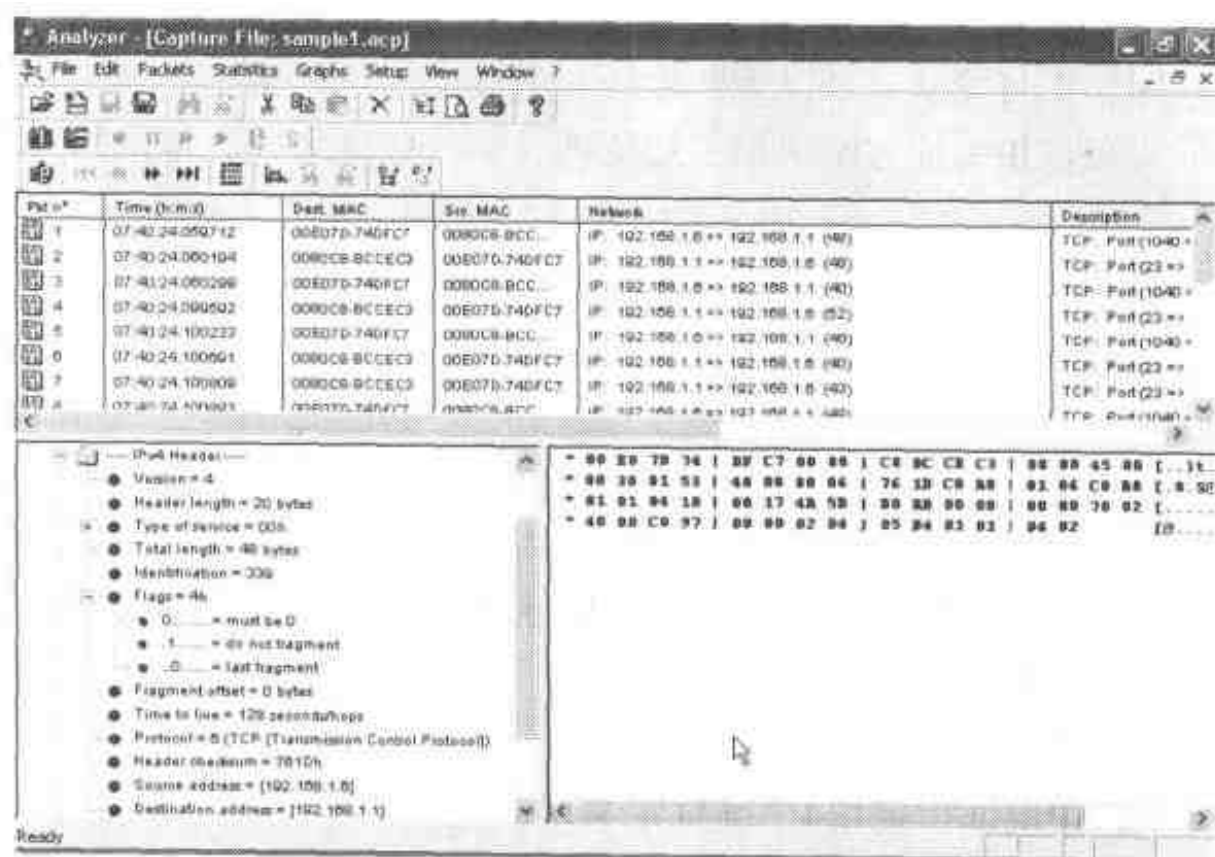


图2.6 在Analyzer中显示IP层网络信息

表2.5 IP字段

字段	位	说明
Version	4	IP报头的版本格式（当前版本是4）
Header Length	4	IP包报头部分的长度，用八位字节
Type of Service	8	想得到的包服务质量（QoS）类型
Total Length	16	整个IP包的长度，用八位字节
Identification	16	唯一识别单个IP包的ID值
Flags	3	指出是否该IP包可以分段，或指出它是否有多个片段
Fragment offset	13	IP包中片段的位置
Time to Live (TTL)	8	允许包在网络中保留的最长时间（按秒计）
Protocol	8	下一层数据的协议类型
Header Checksum	16	IP报头数据的校验和
Source Address	32	发送设备的IP地址
Destination Address	32	接收设备的IP地址
Options	可变的	可以进一步定义IP包特性的可选字段

在Analyzer包详细框架中，每个字段的值都能看到。包含多种信息的字段，例如Type of Service（服务类型）和Flags（标志）字段，可以扩展以显示字段中所有的值。

下一节将介绍用户在跟踪时需要监视的IP包中一些比较重要的字段。

地址字段

虽然以太网地址对于惟一地识别局域网（LAN）上的设备非常有效，但它却根本不能识别远程设备，因为不可能从一个以太网地址看出该设备是在哪一个网络上。为了惟一地识别不同网络上的设备，IP编址方案采用了32位地址的特殊位来标识地址的特点。该IP地址分为三个部分：

- 一个预置的高位数字，用于标识使用的包地址方案
- 一个预先确定的网络地址
- 网络上的一个设备地址

从1到4的高位（high bits）标识了怎样在整个的32位IP地址中确定网络和设备地址。网络地址标识了设备所在的惟一网络，每一个网络都应当有一个分配给它的惟一的IP地址。IP路由器可以用这个网络地址信息决定单个IP包到正确远程网络去的路线，即使这些包要跨因特网传送。设备地址在网络地址中惟一地标识了该设备。在同样的网络地址中没有两个设备能具有相同的设备地址。

默认情况下，IP地址指定用4个高位定义网络地址的四个类，如表2.6所示。

表2.6 IP网络地址的类

高位	网络地址	主机地址	类型
0	7位	24位	A类
10	14位	16位	B类
110	21位	8位	C类
1110	0位	28位	C类

这看起来好像不容易理解，可实际上它相当简单明了。32位的IP地址总是分成4个8位（1字节）的段，然后这些段用相互之间带圆点的十进制数表示，这种方法就叫做点分十进制表示法。这很可能是用户最常见的IP地址格式。十进制数的安排如下：左边是网络地址部分，右边是主机部分。

使用IP地址的点分十进制表示法，网络类的范围划分如下：

- A类 0.x.x.x—127.x.x.x
- B类 128.x.x.x—191.x.x.x
- C类 192.x.x.x—223.x.x.x
- D类 224.x.x.x—254.x.x.x，它又被进一步分为两部分：
 - IP组播地址：224.x.x.x—239.x.x.x
 - 实验网络：240.x.x.x—255.x.x.x

使事情变得更加复杂的是，单独的类网络可以进一步地分成不同的网络，或者子网。每个子网必须用一个公共的网址把它与该类网络中的其他子网区分开。为了识别IP地址的网络部分，为特定的子网定义了子网掩码（subnet mask）。子网掩码标识出用来定义网络地址部分的IP地址的一些位，这些位用于指定主机地址。

让我们举个例子来说明。在默认情况下，一个B类网址有一个255.255.0.0的子网掩码，说明高16位（或者前两个十进制数）是专门用于表示网络地址的。而低16位（或者后面两个十进制数）是用来表示网上单个主机的。这样，在默认的B类网络中，主机130.100.1.6与主机130.100.10.5是在同一个网络上的。但是，使用子网就可以在网址上增加额外的位，来建立该B类网络的子网。例如，如果该B类网络的地址是130.100。使用255.255.255.0子网掩码，第三个八位字节也变成了地址中的网络部分。这样，单个的子网就变成了130.100.x.0，这里的x可以是0到255的任意数字。这使得地址130.100.1.6与地址130.100.10.5是在不同的网络上，因为它们的第三个八位字节是不一样的。

在子网网络上主机的地址范围可以从130.100.x.1到130.100.x.254（全部为1的主机地址，192.168.x.255，被保留作为广播地址）。这使得单独的一个B类地址最多可以包含256个子网，全都支持不同的网络。这种技术在大楼局域网（LAN）和校园局域网中用得十分普遍，它可以帮助把网络上的设备总数分成容易管理的数字。

分段标志

IP包的尺寸问题是许多复杂问题中的一个。一个IP包的最大尺寸可以达到65 536字节。这对于一个包来说是个极大的数据量。事实上，许多较低级的传输都不支持大的IP包用一个块传送（记住，以太网数据部分只能是1500字节长）。为了解决这个问题，IP包采用了分段的办法把IP包分成较小的块传送到目的地。当小块到达目的地的时候，接收软件必须有办法识别出分段包，并把片段重新组装成一个IP包。

分段是用IP包的三个字段完成的，即分段标志字段、片段偏移字段和识别字段。分段标志（fragmentation flags）由三个1字节的标志组成：

- 一个必须为零的保留标志
- 一个不分段标志，这个标志表示IP包也许不被分段
- 一个多片段标志，该标志表示这个包是一个IP片段，还有更多的片段在传送途中

IP识别字段唯一地标识出每一个IP包。任何包的所有片段都有相同的识别号。这告诉接收软件，需要把这些包连接在一起形成原来的IP包。片段偏移字段表示片段在原来包中的位置。

服务类型字段

服务类型（Type of Service）字段标识了IP包的服务质量（QoS）的类型，给一个IP包（或者IP包的流）做上标记，表示它具有特殊的优先权。过去这个字段用得不多。不过最近，为了要区分实时数据（例如视频和音频流）的优先次序，推动了这个字段的使用。这会帮助路由器和其他网络设备给予实时数据较高的服务优先权，以减少在它们传输过程中的延迟。在大多数网络通信中，服务类型字段会被全部设置为零，表示是普通的常规数据。不过，随

着IP用于传输实时数据的增加，用户也许会碰到用服务类型（Type of Service）字段区分包优先权的应用程序。

服务类型（ToS）字段包含8位信息，分为以下几个子字段：

- 3位作为优先次序字段
- 1位表示普通或低延迟
- 1位表示普通或高吞吐量
- 1位表示普通或高可靠性
- 2位为将来使用保留

3位的优先次序字段允许有8个级别的优先权，表2.7是8个级别的详细说明。

表2.7 在ToS字段中定义的IP包优先权

位	优先权
000	常规
001	优先
010	立即
011	急报
100	急报最优先
101	CRITIC/ECP
110	因特网网络控制
111	网络控制

协议字段

协议字段用于识别IP包中所包含的下一层协议。IANA已经定义了135个可以在IP包中用于该字段的值，不过在典型的网络实践中用户只能看到其中的一些。表2.8详细说明了一些比较流行的值。

表2.8 IP协议字段的值

值（十进制）	说明
1	因特网控制消息（Internet Control Message, ICMP）
2	因特网组报文（Internet Group Message, IGMP）
6	传输控制（Transmission Control, TCP）
8	外部网关（Exterior Gateway, EGP）
9	内部网关（Interior Gateway, Cisco IGP）
17	用户数据报（User Datagram, UDP）
88	Cisco EIGRP

本书讨论表2.8中的两个协议，分别是TCP（协议6）和UDP（协议17）。用户用C#编码的大部分应用程序都将使用这两个较高级协议中的一个。下面两节将分别讨论这两个协议。

TCP层

传输控制协议（Transmission Control Protocol, TCP）把连接信息加到数据包上。该协议允许程序在两个网络设备之间建立端到端的连接，给数据的传播提供了一条可靠的路径。TCP保证数据会可靠地传递到目的设备，如果没有到达，则发送器就会收到网络出现错误的指示。由于这一特点，TCP又被称做一个面向连接的（connection-oriented）协议。每一个TCP连接（或会话）都包括一定数量的包开销，用于建立两个设备之间的连接。一旦建立起这种连接，就可以在两个设备之间传输数据，不需要应用程序检查丢失或错位的数据。

作为一个网络程序员，对于TCP是怎样工作的，尤其它是怎样在网络设备之间移动数据的，最好有一个基本的理解。图2.7显示的是Analyzer（分析器）捕获窗口，在包详细资料树中的TCP Header（TCP报头）字段被扩展。为了实现先进的连接和数据可靠特点，这些字段包含对于TCP必需的信息。

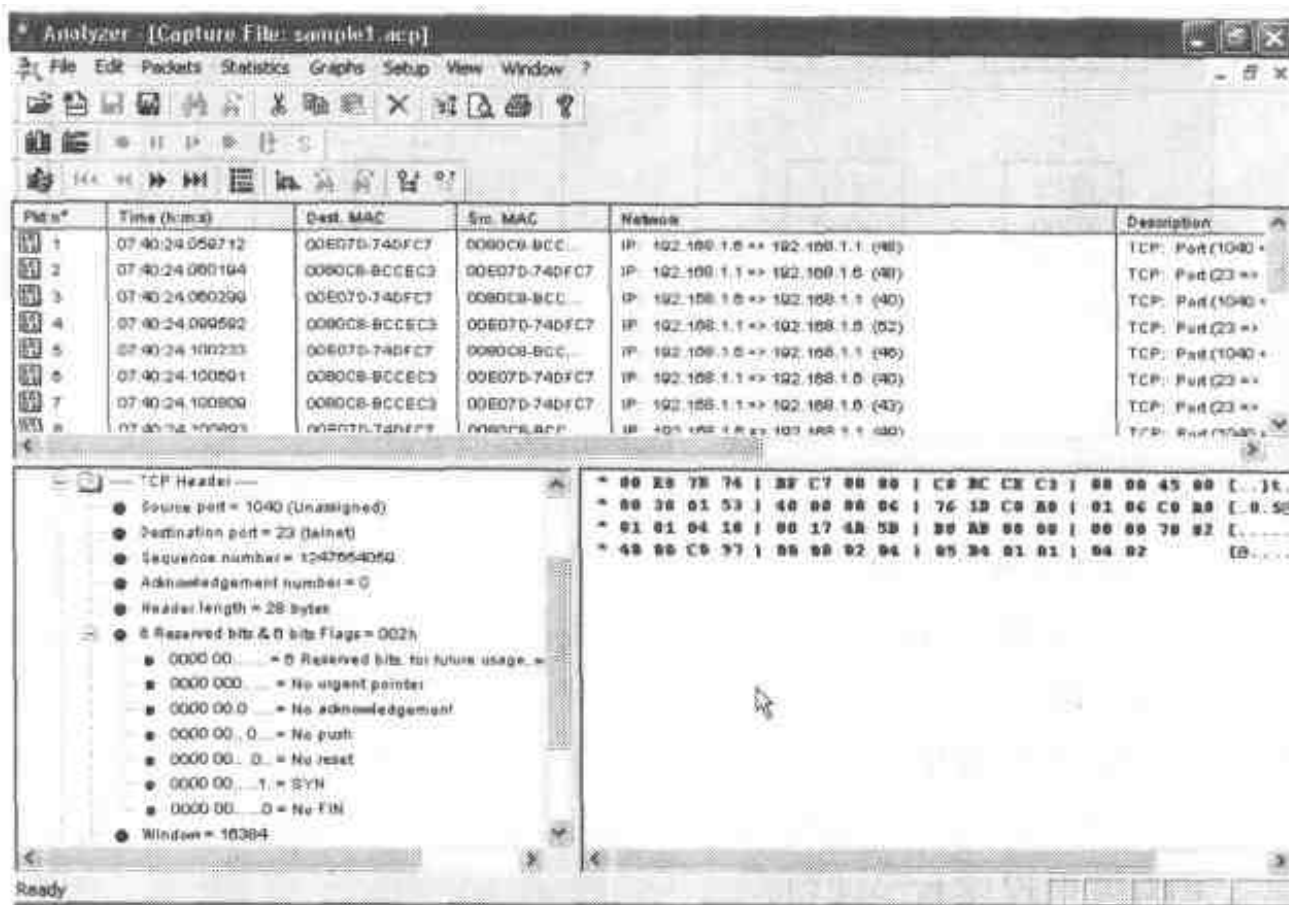


图2.7 Analyzer中显示的TCP Header（TCP报头）字段

TCP报头中有许多字段，每个字段都与TCP会话中的一个特殊功能有关。值得关注的是以下这些功能：

- 跟踪设备（源和目的端口）之间的多个连接
- 跟踪包的顺序和请求重新传输丢失的包（序列和应答号）
- 为数据传输打开和关闭设备之间的连接流（TCP标志）

下一节将叙述这些功能以及它们在网络编程中所起的作用。

TCP应用程序端口

在Analyzer捕获窗口中扩展的TCP报头中的前两个条目是源和目的端口（port）。TCP用端口来识别网络设备上的每一个TCP连接。端口值标识了用于特定应用程序的设备上的TCP终端。为了在远程设备上与应用程序通信，用户必须知道两条信息：

- 远程设备的IP地址
- 分配给远程应用程序的TCP端口

为了建立一个TCP连接，远程设备必须接收分配的端口收到的每一个包。因为在一台设备上会运行着许多使用TCP的应用程序，所以设备必须给每个应用程序分配一个专用的端口号。这样就可以告诉客户机，某个特定的应用程序应当使用哪个端口，并告诉主机应当把收到的包转送给哪一个应用程序。图2.8用图例说明了客户机和服务器是怎样用TCP端口在应用程序之间形成数据信道的。

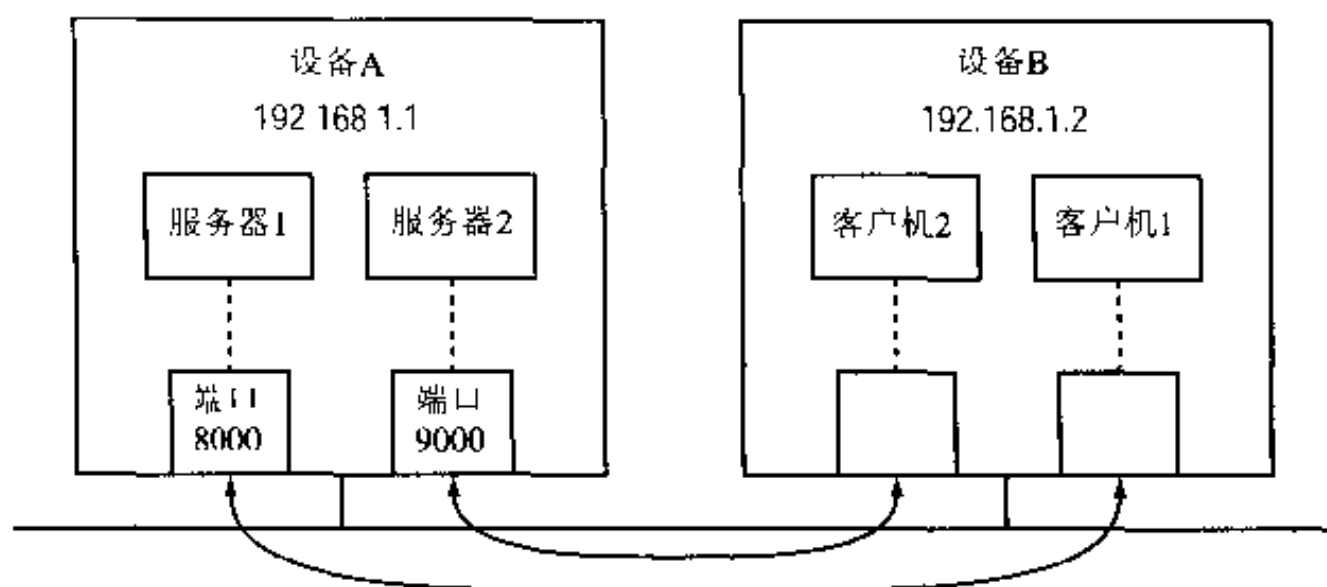


图2.8 TCP连接的例子

在图2.8中，网络设备A正在运行两个服务器应用程序，等待着远程设备发来的包。一个应用程序被分配使用设备的8000端口，另一个应用程序分配使用9000端口。网络设备B是一个想与服务器上的应用程序连接的客户机。任何一个往远程设备发送包的设备，它也必须从操作系统得到一个免费的TCP端口，该端口在会话期间保持开放状态。客户机的TCP端口号通常不重要，可以分配到设备上的任何一个可用端口。客户机把来自设备B上可用端口的包转发到设备A上的应用程序TCP端口。

一个IP地址与一个端口号的组合就定义了一个IP终端（IP endpoint）。一个TCP会话被定义为一个本地IP终端与一个远程IP终端的组合。只有一个会话可以有这些同样的属性。单独的网络应用程序可以使用相同的本地IP终端，但每一个远程连接必须既有单独的IP地址又有远程端口号。

IANA已经规定了一个分配给特定应用程序的标准TCP端口的列表。这保证了任何运行该特定应用程序的主机都会在为那个应用程序分配的TCP端口接受连接。表2.9在指定的许多不同应用程序端口中列出了其中的一部分。

编号从0到1023的端口称为众所周知（well-known）端口，因为这些端口分配给了具体的普通应用程序。如果用户正在创建一个新的应用程序，应当避免把它指定到这些端口，以

免发生冲突。事实上，如果一个应用程序已经使用了某个端口，系统就不允许其他应用程序再使用这个端口。从1024到65 535的端口号是开放的，可供任何应用程序使用。即使在这样大的范围内，也仍然有可能发生两个应用程序选择使用相同端口的情况。因此，许多定制的应用程序给了使用者选择使用自定义端口号的自由。

表2.9 TCP应用程序端口

端口	说明
7	Echo
13	Daytime
17	Quote of the day
20	FTP（数据通道）
21	FTP（控制通道）
22	SSH
23	Telnet
25	SMTP
37	Time
80	HTTP
110	POP3
119	NNTP
123	Network Time Protocol（NTP）
137	NETBIOS命名服务
138	NETBIOS数据报服务
143	因特网消息访问协议（Internet Message Access Protocol, IMAP）
389	轻型目录访问协议（Lightweight Directory Access Protocol, LDAP）
443	Secure HTTP（HTTPS）
993	Secure IMAP
995	Secure POP3

对于网络程序员，TCP端口是极为重要的元素。所有的服务器应用程序都必须在网络设备上使用合法的（和可用的）端口。它涉及到用户怎样为自己的应用程序指定TCP端口。正如前面提到过的，有个始终有效的办法就是，给你的客户一个选择应用程序TCP端口的机会。不过要记住，如果服务器的端口号改变了，客户机应用程序必须知道这个变化，否则它会无法找到服务器应用程序。

保证包的可靠性

在端口字段之后，TCP报头中的下一个字段是序列和应答号。这些值使得TCP能够跟踪包并保证从网络上按正确的次序接收这些包。如果有任何包丢失了，TCP系统会请求重新传输丢失的包，并把数据流重新组合起来再传递给应用程序。

发送出去的每个包都有一个对该TCP会话惟一的序列号。给会话中发出的第一个包选择一个随机的号，设备后来发出的每一个包都用前面一个包的TCP数据字节数增加序列号。这样就保证了在TCP数据流中每个包都是惟一确定的。

接收设备用应答（acknowledgement）字段来表明已经从远程设备接收到最后一个序列号。接收设备在发出应答之前能够接收很多个包。返回的应答号应该是收到数据中最大的连续序列号。这个技术就叫做滑动窗口（sliding window）。收到次序不对的包可以存放在一个缓冲器中，在其他包都顺利收到之后再插到正确的位置上，而不是重新传送这个包。如果有的包丢失了，接收者会看到缺失的序列号，并发出一个较低的应答号请求发送丢失的包。如果没有滑动窗口，每个包都必须个别地应答，必然就增加了网络的通信量和延迟时间。

在查找TCP故障问题的时候，序列和应答号通常是最难调试的部分。往往最好为来自每个单独设备的每个包建一个包序列号的列表。用比较每个包的TCP序列号的方法，用户可以判断出包是否经常被重新传送（在一个TCP会话中重复的或重叠的序列号）。这可能意味着线上某个地方存在着网络故障。

建立TCP会话

TCP报头（TCP Header）在报头长度字段的后面显示的是Flags（标志）字段，可以把这个字段扩展，以看到表2.10中定义的各个字段。这些标志是用来控制两个设备之间的TCP连接的。

表2.10 TCP标志

标志	说明
保留的6位	保留供将来使用——始终为零
1位的URG标志	把包标记为含有紧急数据的
1位的ACK标志	应答收到了一个包
1位的PUSH标志	指出数据马上就要推到应用程序
1位的RESET标志	把TCP连接重新复位成最初的状态
1位的SYN标志	指示一个TCP同步包（会话的开始）
1位的FIN标志	指示一个TCP会话结束

TCP用连接状态（connection states）来确定设备之间的连接情况。用一个特定的握手协议来建立这些连接，并监视在会话期间连接的状况。TCP会话分三个阶段：

- 打开握手

- 会话通信
- 关闭握手

每个阶段都要求标志位按照一定的次序设置。打开握手通常叫做三次握手（three-way handshake），需要三步完成连接：

1. 发起会话的主机发出SYN标志表示要开始一个会话。
2. 接收主机在同一个包中发出SYN和ACK两种标志，表示它同意开始这个会话。
3. 发起会话的主机发出一个ACK标志表示会话被打开，准备接收包。

在会话建立之后，用户会看到包上设置的ACK标志，表示该设备同意接收带有特定序列号的包，为了关闭会话要进行另一次使用FIN包的握手：

1. 启动关闭的主机发出一个FIN标志。
2. 远程主机在同一个包中发出FIN和ACK两种标志，表示它同意关闭这个会话。
3. 启动主机发出一个ACK标志正式结束这个会话。

有时候把这些步骤形象化有助于理解，图2.9显示了TCP握手协议的动作过程。

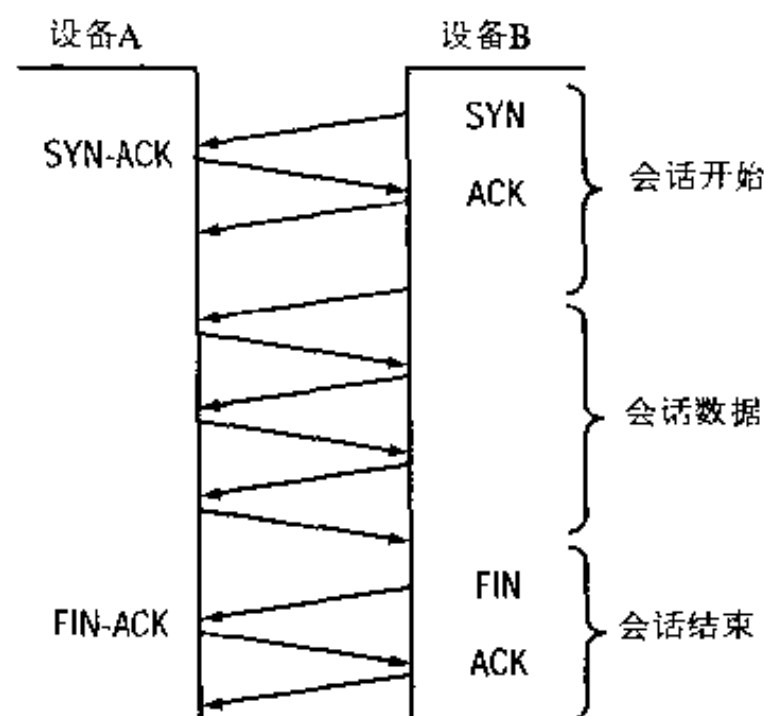


图2.9 TCP握手协议的步骤

TCP会话的几个阶段可以与连接状态名相关联。每一种连接状态都表示在握手过程中会话的当前状况。提供给客户机和服务器的连接状态是完全相等的，在TCP会话中的两个设备遵循同样的TCP状态。表2.11列出了TCP连接状态以及相关标志的说明。

表2.11 TCP连接状态

状态	说明
CLOSED	会话不起作用
LISTEN	设备正在指定的端口等待数据的到来
SYN-SENT	为了启动一个会话，设备发出一个SYN标志，正在等待远程设备的应答和返回一个SYN标志
SYN-RECEIVED	设备收到一个SYN标志，返回SYN和ACK两个标志来完成会话的开始，并在等待接收一个ACK标志

(续表)

状态	说明
ESTABLISHED	两个设备完成了打开握手过程，可以互相之间传送数据包
FIN-WAIT-1	为结束会话，设备发出FIN标志
FIN-WAIT-2	设备收到一个FIN标志，返回FIN和ACK标志
LAST-ACK	设备已经发出一个FIN标志以响应接收到一个FIN标志，正在等待远程设备发出一个ACK标志
TIME-WAIT	在LAST-ACK状态之后，设备要等待一段预先设定的时间才关闭会话端口，以确保远程设备没有发送其他的数据

说明：在Windows系统中，用户用netstat命令提示程序可以看到这些正在起作用的TCP连接状态。netstat程序允许用户列出Windows工作站或服务器上所有TCP和UDP端口的当前状态。

UDP层

用户数据报协议（User Datagram Protocol，UDP）是IP通信中另一个很流行的高级协议。在学习了TCP之后，UDP看起来就像是一个快照。与TCP不同的是，UDP在传输数据的网络设备之间提供了一个无连接的路径，这样就不需要建立标志和连接状态的所有会话开销。每个UDP“会话”只不过是单向传输的一个包。图2.10显示的是在Analyzer（分析器）捕获窗口的包细节框架中一个UDP包的范例。

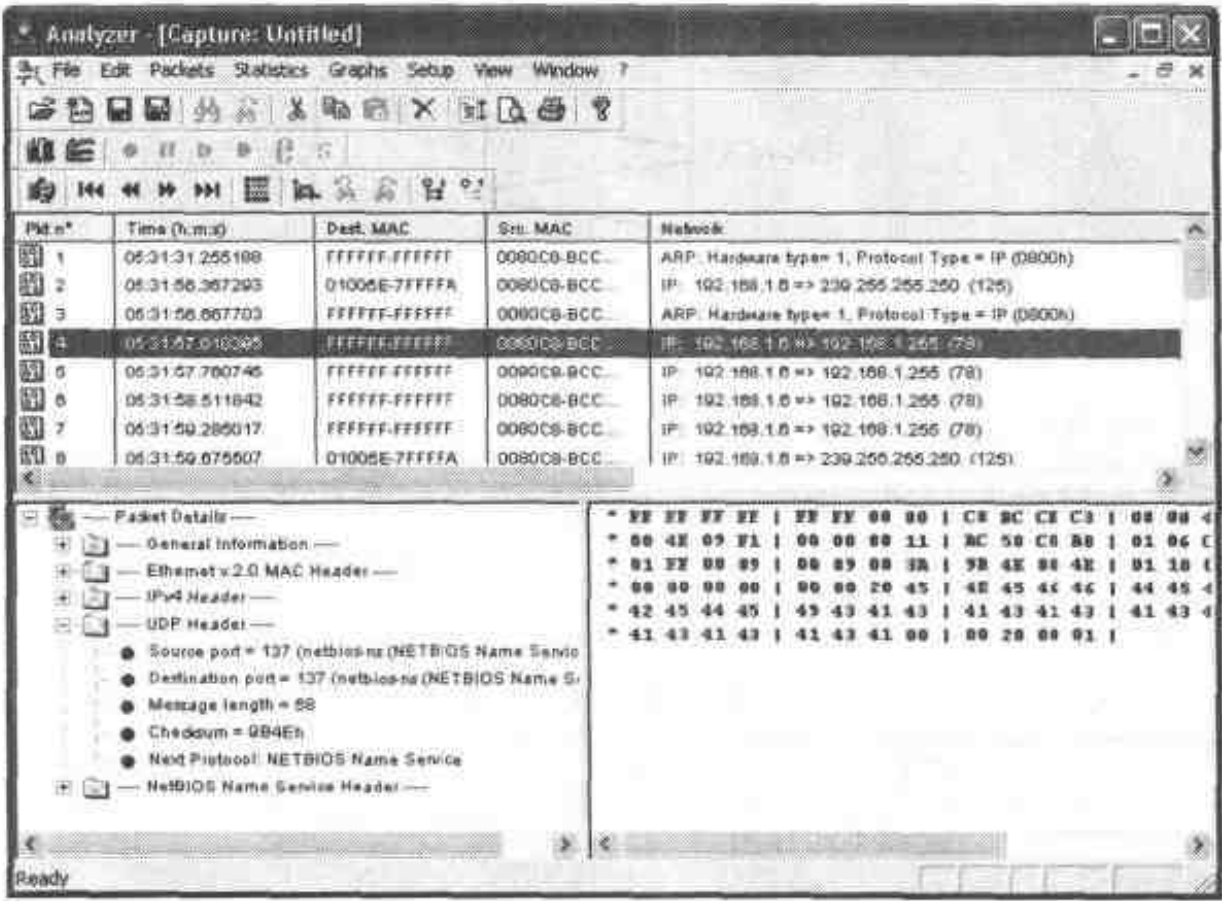


图2.10 在Analyzer（分析器）中显示的UDP报头

UDP报头字段非常简单明了：

- 源端口

- 目的端口
- 消息长度
- 检验和
- 下一级协议

像TCP一样，UDP也用端口号跟踪单个的连接，指定从0到1023端口号为保留的应用程序端口。从1024到65 536端口供用户的应用程序使用。表2.12列出了部分众所周知的UDP端口号。

表2.12 众所周知的UDP端口号

端口	说明
53	Domain Name System (域名系统)
69	Trivial File Transfer Protocol (简易文件传输协议)
111	Remote Procedure Call (远程过程调用)
137	NetBIOS name service (NetBIOS命名服务)
138	NetBIOS datagram (NetBIOS数据报)
161	Simple Network Management Protocol (简单网络管理协议)

许多应用程序都保留着供它们使用的TCP和UDP端口，虽然实际上应用程序只使用一个或另外一个。不管怎样，使用保留端口号范围以上的端口号，以免将来协议改变时发生可能的冲突总是最安全的。

本章的下一节将介绍用户应当了解的内容，关于怎样在用户程序中使用TCP和UDP协议的问题。

用TCP和UDP编程

现在，读者已经看到了TCP和UDP的概况，下面进一步探讨对于网络程序员来说每个协议有什么意义。这两个协议用完全不同的方法在网络设备之间移动数据。下面几节将详细说明每一个协议对于网络程序员表现出来的特点和缺陷，以帮助读者决定你的应用程序最好使用哪种协议。

TCP编程的特点

对于TCP，要记住的最重要的事情是它是一个面向连接的协议。两个设备之间一旦已经存在连接，就会形成一个可靠的数据流，保证数据从一个设备准确地移送到另一个设备。虽然使用TCP的用户不需要担心数据的丢失和错位，但是当用TCP编程的时候有个需要考虑的大问题：缓冲器。

由于TCP必须保证数据的完整性，所以，在从远程设备收到“已收到数据”的肯定的应

答信息之前，它一直把所有发送的数据都存放在一个本地缓冲器中。同样地，当从网络接收数据的时候，TCP也必须有一个接受数据的本地缓冲器，以保证在把数据递送给应用程序之前所有的数据块是按顺序接收的。由于这个单独的TCP缓冲器，在用户程序与远程主机上目的程序之间移动数据操作起来显得比预期的困难一些。图2.11显示了操作的过程。

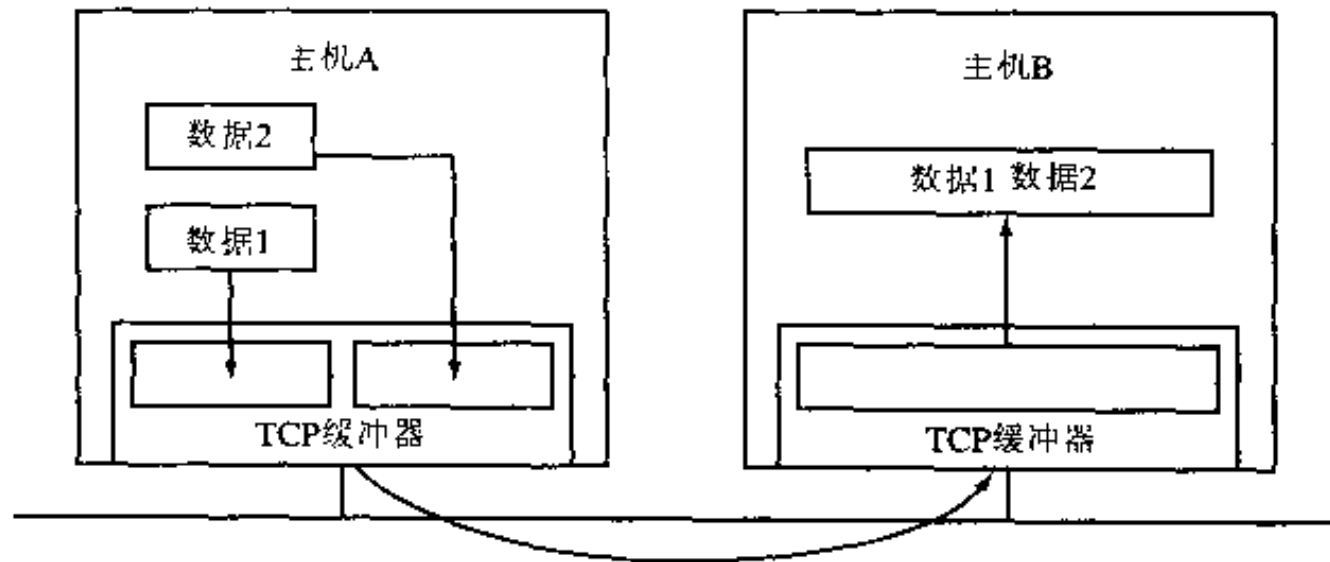


图2.11 TCP数据传递过程

Windows操作系统上的TCP子系统负责接收来自用户应用程序和远程设备的数据。但不是立即在网上传输数据，而是把数据放在缓冲器中一定的时间。此时，用户应用程序也许给远程主机发送更多的数据，这些数据也依次加到缓冲器（如图中的数据1和数据2数据包）。当TCP子系统做好向远程设备发送数据的准备时，它就打算把缓冲器中的全部内容一起发送出去，而不是只发送一块数据，所以，数据1和数据2缓冲器的内容被作为一个包发送。接收主机收到了一个由数据1和数据2缓冲器内容组成的包。现在就需要应用程序来判断，包中到底是两块数据还是一大块数据。

这对于网络程序员意味着什么呢？当把消息中的数据发送给远程设备的时候，远程设备不需要接收相同消息单位编号的数据，TCP子系统会把一条消息所有的数据都放在TCP缓冲器中，根据用户发送数据的速率和接收设备接收数据的速率，这些消息（或至少是其中的一些消息）也许被一起推进数据流中。TCP通信的这个“特点”令许多初学网络编程的人感到惊讶。

由于TCP不保留数据消息的界线，所以编程者必须在自己的网络程序中加以弥补。有两种处理这个问题的方法：

- 创建一个协议，要求从主机发出的每一条数据消息都一对一地响应
- 设计一个数据消息标记系统，辨认数据流中数据消息的边界

两种技术都要求在编码之前做好充分的预先计划。大多数使用TCP的因特网协议都执行第一种分离数据的方法。FTP和类似的协议实现命令/响应系统，系统中的客户机一次发出一条消息命令，并等待从远程主机收到每条命令的响应。

UDP编程的特点

创建UDP就是为了解决TCP的消息边界问题。UDP保留了从应用程序发送到网络上的所有消息的数据边界。由于UDP是特殊设计的，不担心可靠数据的传输问题，它不需要使用本

地缓冲器保留发送的或接收的数据。代替的做法是，当从应用程序接收到消息时，每条消息都作为一个单独的包转发。同样，从网络上收到的每条消息也作为一个单独的包转发给应用程序。UDP在网络包中保留了消息的边界，如图2.12所示。

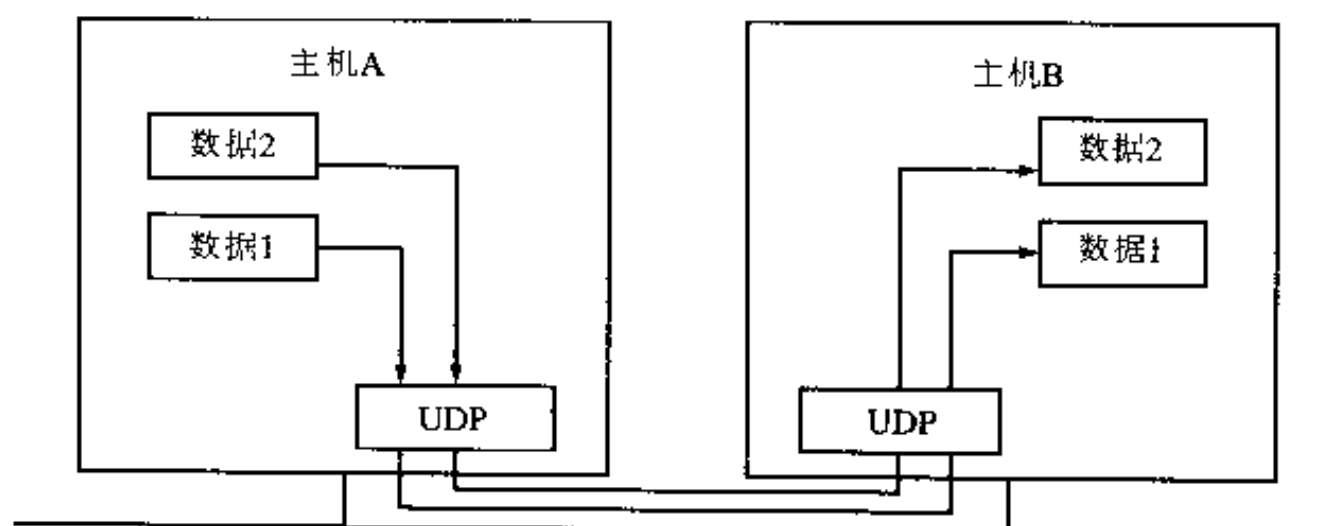


图2.12 UDP数据传递

UDP除了具有保留消息边界的功能之外，它还存在另外的问题。由于UDP不保证数据的传递，所以如果涉及到数据到达目的地的问题，用户应用程序必须执行那种功能。因为一台设备发出了UDP数据包并不意味着接收设备收到数据包，所以用户必须保证用户的程序能够处理丢失包的问题。

解决这个问题的一种方法是，在发出数据包之后总是期待从远程设备返回某种类型的答复（命令/响应的方法）。如果没有收到答复，就应当认为发出的数据没能通过网络到达远程设备。这使得通过UDP发送数据的过程需要有4个步骤：

1. 发送数据到远程设备。
2. 启动一个计时器，设置一个预定的时间周期。
3. 等待远程设备的响应。收到响应时停止计时器计时，并继续用户的程序。
4. 如果还没收到响应计时器就到时了，返回重复步骤1。如果重复了步骤1多次（重试的次数）都没有答复，就认为用户不能与该远程主机通信。

虽然用UDP发送数据在消息边界方面稍微容易一些，但它比TCP还是复杂得多，因为用户需要自己检查丢失的数据包。

寻找IP地址信息

网络编程的最大挑战之一就是处理每个工作站和服务器的网络配置。当通过网络发送数据的时候，需要确定运行程序的系统的IP网络信息。Windows操作系统系列提供了许多种确定IP配置信息的方法，有手工操作的也有从程序中获得的。这一节将向读者演示其中的一些可以使用的方法，用户既可以手工地也可以从C#程序中得到IP配置信息。

使用ipconfig程序

ipconfig程序可以显示系统中每个有效网络接口的IP网络信息。ipconfig的默认形式是显示每个网络设备的基本信息：

```
C:\>ipconfig
```

```
Windows NT IP Configuration
```

```
Ethernet adapter El90x1:
```

```
IP Address. . . . . : 192.168.1.6
Subnet Mask . . . . . : 255.255.255.0
Default Gateway . . . . . : 192.168.1.1
```

```
PPP adapter NdisWan5:
```

```
IP Address. . . . . : 0.0.0.0
Subnet Mask . . . . . : 0.0.0.0
Default Gateway . . . . . :
```

```
C:\>
```

可以看出，**ipconfig**程序发现在工作站上配置有两个网络接口。第一个接口是为局域网配置的以太网卡，用一个静态IP地址指定该工作站。第二个设备是为工作站使用调制解调器而配置的一个PPP连接。由于当PPP会话开始时，PPP连接是动态地获得其IP信息的，所以工作站的这些值全都默认地设置为零。当PPP连接建立时，分配的IP信息就会显示出来。

为了得到有关设备更详细的信息，可使用**ipconfig**命令的/all开关，如下所示：

```
C:\>ipconfig /all
```

```
Windows NT IP Configuration
```

```
Host Name . . . . . : shadrach.blum.lan
DNS Servers . . . . . : 192.168.1.1
                        192.168.1.2
                        192.168.1.3
Node Type . . . . . : Broadcast
NetBIOS Scope ID. . . . . :
IP Routing Enabled. . . . . : No
WINS Proxy Enabled. . . . . : No
NetBIOS Resolution Uses DNS : No
```

```
Ethernet adapter El90x1:
```

```
Description . . . . . : 3Com EtherLink PCI
Physical Address. . . . . : 00-50-DA-10-78-67
DHCP Enabled. . . . . : No
IP Address. . . . . : 192.168.1.6
Subnet Mask . . . . . : 255.255.255.0
Default Gateway . . . . . : 192.168.1.1
```

```
PPP adapter NdisWan5:
```

```
Description . . . . . : NdisWan Adapter
Physical Address. . . . . : 00-00-00-00-00-00
```

```
DHCP Enabled. . . . . : No
IP Address. . . . . : 0.0.0.0
Subnet Mask . . . . . : 0.0.0.0
Default Gateway . . . . . :
```

```
C:\>
```

使用注册表

注册表是装有Windows工作站和服务器的“珍宝箱”。它包含着许多关于安装软件和硬件的信息，包括特殊的配置信息。用户可以用C#的各个Registry（注册表）类浏览系统的注册表，以确定系统的许多特性，包括当前的IP网络配置。这一节将介绍，在注册表的什么地方可以找到IP信息，以及怎样用C#程序提取这些信息。

注册表中的IP信息

注册表是由许多存储着一个或多个数据项的关键字组成的。每个关键字都存储着系统上某个实体（例如一个网卡）的信息。每个数据项都有一个定义数据信息的值。注册表关键字的组织方法类似于一个目录/子目录的结构。关键字可以包含数据也可以包含几个子关键字。每个子关键字本身也许包含数据和另外的子关键字；结构可能有好几层深。

Windows注册表由6个基本的关键字组成，所有其他关键字都建立在这6个关键字之上：

```
CLASSES_ROOT
CURRENT_CONFIG
CURRENT_USER
DYN_DATA
LOCAL_MACHINE
USERS
```

注册表基本关键字用术语HKEY作为前缀，所以LOCAL_MACHINE基本关键字就用关键字HKEY_LOCAL_MACHINE来访问，为了缩写，通常就叫做HKLM。

子关键字用目录惯例同样的方法来引用，用反斜线（\）来分隔不同级的子关键字：

```
HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion
```

警告：注意，注册表包含的关键字名可以全部大写、大小混写和两个词之间用空格分开。这种矛盾状况在为注册表编程的时候经常引起问题。

注册表包含有系统中安装的每个网络设备的网络信息。很可惜，不同的Windows平台在各自不同的位置上存储这些信息，有的平台把网络信息分别放在各个网络接口卡（NIC）上，而另外的平台则把所有的网络信息都存放在一个公共的地方。下面几节向读者介绍怎样找到每个平台的网络信息。

Windows 98和Me

不管系统中配置了多少网卡，Windows 98和Me设备都把网络信息保存在一个单独的地方。用户总是可以在相同的子关键字中找到系统的IP配置：

```
HKLM\SYSTEM\CurrentControlSet\Services\Class\NetTrans
```

在这个子关键字中，每个网络接口的信息都按一个单独的子关键字保存，这些子关键字从0000开始，按顺序命名。每个子关键字包含着为一个网络设备的IP信息定义的数据值。表2.13列出了网络接口子关键字中包含的IP数据值。

表2.13 网络信息的注册表数据名

数据名	说明
IPAddress	分配给该接口的IP地址
DefaultGateway	分配到该接口的IP路由器
IPMask	定义该接口使用的网络地址的子网掩码

Windows NT、2000和XP

如果读者正在寻找一个Windows NT、2000或XP系统上的网络信息，那首先必须知道系统上配置有哪些网卡。这些平台的网络信息都存储在各个网卡的注册表关键字中，而不是像Windows 98和Me那样都存储在一个地方。为获得IP地址信息必须搜索每一个网卡，并从各个地方提取网络信息。

第一步就是找到系统中配置的所有网络设备的注册表关键字。Windows NT、2000和XP系统有一个单独的注册表，列出系统中配置的所有网络接口。这个关键字列在HKEY_LOCAL_MACHINE (HKLM) 根关键字下，实际的子关键字表示如下：

```
HKLM\Software\Microsoft\Windows NT\CurrentVersion\NetworkCards
```

这是系统中每个网络接口子关键字的引用方法。每个子关键字都按顺序编号，并包含系统中安装的单个接口的信息，这些信息作为数据值存储到子关键字中。用户需要提取的有关数据值是ServiceName数据值，这个值指出了存有IP信息的Services注册表中网络设备的入口。

在获得了每个网络设备的ServiceName值之后，下一步就是查看每个网络接口ServiceName的Services信息区。对应的ServiceName关键字将作为基本系统服务关键字下的子关键字出现：

```
HKLM\SYSTEM\CurrentControlSet\Services
```

每个系统服务都在这个关键字下有一个子关键字，包含该特殊服务的信息。在第一步检索到的网络接口ServiceName作为子关键字名使用。所以只要有了所有ServiceName的列表，找到正确的信息就容易了。

IP信息作为数据值存储在Service关键字的parameters\tcpip子关键字下。该子关键字中的IP信息使用的数据名类似Windows 98和Me系统中的数据名：

IPAddress

SubnetMask

DefaultGateway

获得存储在数据值中的IP信息是一件很简单的事情，在Windows NT、2000和XP中惟一需要慎重对待的问题是，每个网络设备可能会有多组IP信息。这意味着每个设备的注册表IP数据值可能包含多个值。用户必须确保检索到了该IP数据值所有可能的值。

说明：这种方法只对用静态IP地址的网卡有效。如果用户的网络是用DHCP分配IP地址的，可能需要检查你Windows平台的DhcpIPAddress注册表关键字的值。

用C#搜索注册表

一旦读者确定了到哪里去找目标系统注册表中的网络信息，就可以编写C#代码来搜索信息并把信息显示给用户（或在自己的程序中使用它）。在这一节，我们通过考察一个C#程序来查询系统注册表中的IP信息。

.NET框架提供了一组类，可以帮助用户查询和操作注册表的入口。.NET Registry（注册表）类包含在两个命名空间中：

Microsoft.Win32.Registry Microsoft.Win32.Registry命名空间包含所有访问基本注册表关键字名所需要的基类：Current_User、Local_Machine、Classes_Root、Current_Config、Dyn_Data和Users。用户必须用一个基本注册表关键字名建立特定的注册表子关键字。

Microsoft.Win32.RegistryKey Microsoft.Win32.RegistryKey命名空间包含查询和修改注册表关键字和数据需要的类和方法。OpenSubKey()方法允许用户打开一已知关键字的子关键字，并访问该子关键字中包含的任何数据值。当用户得到需要的子关键字之后，可以用GetValue()方法来获取指派给该子关键字的数据值。

程序清单2.2展示了CardGrab.cs程序，该程序可以用系统注册表找出安装在Windows NT、2000或XP系统中的网络设备，并显示每个设备的IP信息。

程序清单2.2 CardGrab.cs程序

```
using System;
using Microsoft.Win32;

class CardGrab
{
    public static void Main ()
    {
        RegistryKey start = Registry.LocalMachine;
        RegistryKey cardServiceName, networkKey;
        string networkcardKey = "SOFTWARE\\Microsoft\\
            Windows NT\\CurrentVersion\\NetworkCards";
        string serviceKey =
```

```

        "SYSTEM\\CurrentControlSet\\Services\\";
string networkcardKeyName, deviceName;
string deviceServiceName, serviceName;

RegistryKey serviceNames =
    start.OpenSubKey(networkcardKey);
if (serviceNames == null)
{
    Console.WriteLine("Bad registry key");
    return;
}

string[] networkCards = serviceNames.GetSubKeyNames();
serviceNames.Close();

foreach(string keyName in networkCards)
{
    networkcardKeyName = networkcardKey + "\\" + keyName;
    cardServiceName = start.OpenSubKey(networkcardKeyName);
    if (cardServiceName == null)
    {
        Console.WriteLine("Bad registry key: {0}",
            networkcardKeyName);
        return;
    }
    deviceServiceName =
        (string)cardServiceName.GetValue("ServiceName");
    deviceName =
        (string)cardServiceName.GetValue("Description");
    Console.WriteLine("\nNetwork card: {0}", deviceName);

    serviceName = serviceKey + deviceServiceName +
        "\\Parameters\\Tcpip";
    networkKey = start.OpenSubKey(serviceName);
    if (networkKey == null)
    {
        Console.WriteLine("    No IP configuration set");
    } else
    {
        string[] ipaddresses =
            (string[])networkKey.GetValue("IPAddress");
        string[] defaultGateways =
            (string[])networkKey.GetValue("DefaultGateway");
        string[] subnetmasks =
            (string[])networkKey.GetValue("SubnetMask");

        foreach(string ipAddress in ipaddresses)

```



```

        {
            Console.WriteLine("    IP Address: {0}", ipaddress);
        }
        foreach(string subnetmask in subnetmasks)
        {
            Console.WriteLine("    Subnet Mask: {0}",
                               subnetmask);
        }
        foreach(string defaultGateway in defaultGateways)
        {
            Console.WriteLine("    Gateway: {0}",
                               defaultGateway);
        }
        networkKey.Close();
    }
}
start.Close();
}
}

```

必须指定一个开始进行关键字搜索的根关键字，这是用注册表类来完成的：

```
RegistryKey start = Registry.LocalMachine;
```

这建立了一个注册表关键字对象`start`，并把它设置到根类`HKLM_LOCAL_MACHINE`。

下一步继续往下挖掘，设置一个关键字对象到相应的包含网络设备信息的子关键字：

```

string networkcardKey = "SOFTWARE\\Microsoft\\ ➡
                        Windows NT\\CurrentVersion\\NetworkCards";
RegistryKey serviceNames =
                        start.OpenSubKey(networkcardKey);

```

现在，关键字对象`serviceNames`指出了安装网络设备的子关键字的位置。

下一步，对于每个网络设备的子关键字名重复使用`GetSubKeyNames()`方法。由于服务关键字使用相同的关键字名，所以可以把那个值加到已知的服务关键字的位置，用`OpenSubKey()`方法打开该子关键字，同样要从根关键字开始引用：

```

serviceName = serviceKey + deviceServiceName + "\\Parameters\\Tcpip";
networkKey = start.OpenSubKey(serviceName);

```

在打开了每个子关键字之后，就可以用该关键字对象的`GetValue()`方法从服务关键字中检索IP信息了。由于每条IP信息的数据入口可以有多个值，所以用户必须把结果分配到一个字符串数组，并用`foreach`函数显示完整的数组：

```

string[] ipaddresses =
        (string[])networkKey.GetValue("IPAddress");
foreach(string ipaddress in ipaddresses)

```



```
{  
    Console.WriteLine("    IP Address: {0}", ipaddress);  
}
```

CardGrab程序的样例输出应当如下所示:

```
C:\>CardGrab
```

```
Network card: D-Link DE660 PCMCIA LAN adapter  
IP Address: 192.168.1.6  
Subnet Mask: 255.255.255.0  
Gateway: 192.168.1.1
```

```
C:\>
```

可以用类似的方法找到Windows 98或Me设备的IP信息。事实上,这个方法要简单得多,因为只需要探索IP信息的一个位置就行了。

使用WMI

除了注册表外,还有另外一个数据源可用来获得Windows系统的IP网络信息。Windows Management Instrumentation (WMI, 视窗系统管理方法) 是微软Web-Based Enterprise Management (WBEM, 基于Web企业管理) 的实现,它是网络环境中访问系统信息的一个标准,是由Distributed Management Task Force股份有限公司(DMTF)开发的。从Windows 2000开始,微软就把含有系统软、硬件信息和当前硬件状态的数据库包含在操作系统(OS)中了。C#语言可以直接查询该数据库,以判断管理软、硬件组件(包括网络设备)的状态。

提示: 旧的Windows系统可以安装从WMI软件开发工具箱单独下载的WMI系统。可以从微软的软件下载Web站点 (<http://www.microsoft.com/downloads/search.asp?>) 免费下载获得。该WMI由多个存储系统信息的数据库表组成,当每个监视设备的状态改变时这些信息会被更新。C# WMI类允许用户查询数据库,并提取关于运行系统更新后的信息。

WMI中的IP信息

WMI中的WMI Win32_NetworkAdapterConfiguration表包含关于系统中安装的网络设备的信息。该表查询的结果是每个网络接口的信息集。每个网络接口都表现为返回信息集中的一条记录,网络接口的IP信息作为记录中单独的字段出现。

WMI数据库中包含着种类繁多的网络数据值。表2.14描述了Win32_NetworkAdapterConfiguration表中包含的一些有用的IP数据字段。

应用程序可以用标准SQL语法的查询访问WMI数据库。查询必须指定要查询的表和返回的字段(或者用*通配符返回所有的字段)。一个样例SQL查询表示如下:

```
SELECT IPAddress, IPSubnet from Win32_NetworkAdapterConfiguration
```

这个SQL语句返回系统中配置的每个网络接口的IP地址和子网掩码。再次提醒读者,任何一个接口都可能有多IP地址,所以返回的信息采用多值数组的形式。

现在读者已经知道WMI网络信息存放在什么地方了。下面来看一下用C#提取该信息的范例。

表2.14 Win32_NetworkAdapterConfiguration数据字段

字段	说明
DefaultIPGateway	分配给设备的IP路由器地址数组
Description	对网络设备的说明
DHCPEnabled	是否该设备动态地分配它的IP地址
DHCPServer	服务器使用的DHCP，用来分配一个IP地址
DNSHostName	为分解主机名而使用的DNS主机
IPAddress	分配到设备的IP地址数组
IPEnabled	是否该设备在网络上使用IP
IPSubnet	设备上使用的IP子网地址数组
MACAddress	分配到网络设备的以太网MAC地址

用C#查询WMI

.NET System.Management命名空间包含着用于查询WMI数据库的类。用户可以创建一个C#程序查询WMI，找到系统中安装的每一个网络接口，并显示返回的IP信息。用C#程序查询WMI数据库的步骤如下：

- 1. 为查询数据库表，创建一个包含SQL选择语句的ManagementObjectSearcher对象。
- 2. 创建一个ManagementObjectCollection对象，执行SQL查询来获得结果集。
- 3. 在foreach循环中，给ManagementObjectCollection中的每个对象分配一个新的ManagementObject。
- 4. 给合格的类型变量分配想要的字段。

程序清单2.3显示了WMICardGrab.cs程序，该程序用C#的WMI类获得网络接口的IP信息。

程序清单2.3 WMICardGrab.cs程序

```
using System;
using System.Management;

class WMICardGrab
{
    public static void Main ()
    {
        ManagementObjectSearcher query = new
        ManagementObjectSearcher("SELECT * FROM ↵
        Win32_NetworkAdapterConfiguration WHERE IPEnabled = 'TRUE'");
        ManagementObjectCollection queryCollection = query.Get();
        foreach( ManagementObject mo in queryCollection )
        {
```

```

        string[] addresses = (string[])mo["IPAddress"];
        string[] subnets = (string[])mo["IPSubnet"];
        string[] defaultgateways =
            (string[])mo["DefaultIPGateway"];

        Console.WriteLine("Network Card: {0}",
            mo["Description"]);
        Console.WriteLine("    MAC Address: {0}",
            mo["MACAddress"]);

        foreach(string ipaddress in addresses)
        {
            Console.WriteLine("    IP Address: {0}',
                ipaddress);
        }
        foreach(string subnet in subnets)
        {
            Console.WriteLine("    Subnet Mask: {0}", subnet);
        }
        foreach(string defaultgateway in defaultgateways)
        {
            Console.WriteLine("    Gateway: {0}",
                defaultgateway);
        }
    }
}

```

C# WMI类ManagementObjectSearcher指定了将对WMI数据库执行的SQL查询。为了有利于自己，用户可以用IPEnabled字段，过滤出所有没有IP配置的网络设备：

```

ManagementObjectSearcher query = new
    ManagementObjectSearcher("SELECT * FROM ↵
        Win32_NetworkAdapterConfiguration WHERE IPEnabled = 'TRUE'");
ManagementObjectCollection queryCollection = query.Get();

```

查询的结果存储在**ManagementObjectCollection**中。每一个集合项表示结果集中的一条记录，每条记录表示一个网络接口。方便的foreach C#函数可循环用于对检索出来的记录集集合逐条检查每一个记录，把它分配给一个**ManagementObject**对象。记录中的字段用它们正确的字段名引用：

```

        string[] addresses = (string[])mo["IPAddress"];
        string[] subnets = (string[])mo["IPSubnet"];
        string[] defaultgateways = (string[])mo["DefaultIPGateway"];

```

与注册表版本类似，IPAddress、IPSubnet和DefaultIPGateway值都是可以包含多个值的

数组。为了适应这种情况，用户必须把结果分配到一个字符串数组，并再一次用foreach语句通过该字符串数组。

从WMICardGrab程序中检索得到的IP信息应当与从注册表版本中得到的相同：

```
C:\>WMICardGrab
Network Card: D-Link DE560 PCMCIA LAN adapter
MAC Address: 00:80:C8:BC:CE:C3
IP Address: 192.168.1.6
Subnet Mask: 255.255.255.0
Gateway: 192.168.1.1

C:\>
```

使用DNS

最后一种确定系统IP信息的方法是利用C# DNS（域名系统）类。第4章“DNS和C#”中对于怎样用DNS获得因特网主机名和IP地址有详细全面的描述。可以利用这个系统，获得用户程序运行系统的本地IP地址。

C# System.Net命名空间包含Dns类，该类由几个方法组成，允许用户获得关于主机的DNS信息。GetHostName()方法检索本地系统的主机名。GetHostByName()方法试图找到主机名的IP地址。用户可以把这两个函数结合起来，找到本地系统的IP地址。程序清单2.4显示了DNSName.cs程序，该程序演示了怎样用这些函数找到系统的本地IP地址。

程序清单2.4 DNSName.cs程序

```
using System;
using System.Net;

class DNSName
{
    public static void Main ()
    {
        string hostName = Dns.GetHostName();
        Console.WriteLine("Local hostname: {0}", hostName);
        IPHostEntry myself = Dns.GetHostByName(hostName);
        foreach (IPAddress address in myself.AddressList)
        {
            Console.WriteLine("IP Address: {0}", address.ToString());
        }
    }
}
```

同样，与注册表和WMI程序类似的是，用户必须考虑到本地系统指定有多个IP地址的

可能性。IPHostEntry类的AddressList属性是一个类型IPAddress的数组。用户可以用foreach函数提取数组中的每个IP地址。这个程序的输出非常简单：

```
C:\>DNSName
Local hostname: abednego
IP Address: 192.168.1.6

C:\>
```

这是一个获得本地系统IP地址的非常通用的方法，可以用它确定本地IP地址以便用于建立服务器的网络连接。读者在后续章节中还会多次见到这个代码。

小结

本章告诉读者怎样用IP在网络上传输数据，用户怎样才能看到数据。WinPcap驱动程序可以使任何一个Windows系统能够捕获网络包，以便进行监视。WinDump和Analyzer程序可以显示WinPcap驱动程序捕获的网络数据。

用户在捕获了网络包之后，必须知道解码的方法，这样才能理解数据是怎样通过网络传送的。读者需要很好地理解以太网和IP层协议，以及更高层的TCP和UDP协议。每个协议都建立在前一级协议之上，再加上关于数据怎样在网络上传输的更多的信息。作为一名C#网络程序员，应当清楚地理解这些协议对于编写网络程序会有什么影响。在网络上传输数据的过程中，TCP不保留数据消息的边界，而UDP不保证消息传递到目的地。对于网络编程来说，两个协议都有各自的优缺点。

最后，本章介绍了几种可用于确定Windows系统IP地址的方法，可以用C#注册表、WBEM的微软WMI的实现和DNS类。Windows注册表包含有可以用.NET Registry和RegistryKey类访问的网络信息。通过查询注册表的网卡信息，可以获得用户程序运行系统的静态和动态IP地址。

System.Management命名空间包含有可以访问Windows 2000和2000以后系统中的WMI数据库的类（较早期的Windows平台可以手工安装WMI数据库）。使用标准的SQL查询和ManagementObjectSearcher、ManagementObjectCollection类，可以确定所有需要的IP地址信息。

.NET库也包含标准的DNS类，允许用户查询本地主机名和IP地址。这个方法将在第4章中更详细地叙述，第4章探讨了.NET库中所有可用的DNS函数并向读者展示怎样在C#网络程序中使用这些函数。

第3章 C#网络编程的类

- 套接字编程技术的引导者
- Windows中的套接字编程技术
- C#套接字编程技术
- C#套接字助手类

本章从介绍C#网络编程的类开始，把读者引向C#网络编程的旅程。在第2章中读者已经学习了TCP和UDP以及在通过网络把数据传送到远程设备的过程中它们所起的作用。现在，读者将会看到这一切在C#网络程序中是怎样完成的。

Windows中的许多功能和特性都是从UNIX领域借鉴来的，这已经不是什么秘密了。网络编程就是借用的功能之一，而套接字编程是C#网络编程方法的核心部分。

在深入探讨C#网络套接字类之前，最好对它是从哪里来的以及为什么它们用这种方式工作的历史有一点了解。如果读者已经熟悉UNIX的套接字编程技术或者Windows的Winsock接口，也许可以跳过“套接字编程技术的引导者”一节的内容，如果不是这样的话，那还是坐下来，阅读网络编程的这段简要历史，相信值得一读。

套接字编程技术的引导者

UNIX操作系统彻底改革了编程领域的许多特征，文件描述符（file descriptor）就是其中之一。文件描述符给文件对象提供了一个编程的接口。因为几乎在UNIX系统中所有的对象都被定义成一个文件，文件描述符可以用来发送和接收数据，这些数据跨UNIX系统有许多对象。这使UNIX程序员的工作大大简单化。实际上，不论用户想访问哪种类型的设备（或文件），同样类型的编程模式都有效。

从4.2BSD版的UNIX开始，网络访问也是用文件描述符定义的。网络文件描述符叫做套接字（socket）。UNIX（和Windows）的网络程序都在所有的网络通信中利用套接字。这一节将概括地介绍套接字网络编程的特点，使读者作好充分的准备，更好地理解C#网络编程技术背后的概念。紧接着是“Windows中的套接字编程技术”一节，这一节描述了套接字的Windows Winsock实现。Winsock是建立C# Socket类实现的基础。

套接字（Socket）

在基于套接字的网络编程技术中，用户不直接访问发送和接收包的接口设备，而是建立一个中间文件描述符来处理编程接口到网络的操作。UNIX操作系统处理操作的细节，决定用哪一个网络设备发送数据和怎样发送。

用来引用网络连接的特殊的文件描述符就叫做套接字。套接字定义如下：

- 一个特殊的通信域，例如 一个网络连接或者 一个UNIX进程间通信（Interprocess Communication, IPC）管道
- 一个特殊的通信类型，例如流或数据报
- 一个特殊的协议，例如TCP或UDP

在建立了套接字之后，必须把它绑定到系统中一个指定的网络地址和端口，或绑定到一个远程网络地址和端口，一旦套接字被绑定，就可以用它在网络上发送和接收数据。图3.1表示了这个过程。

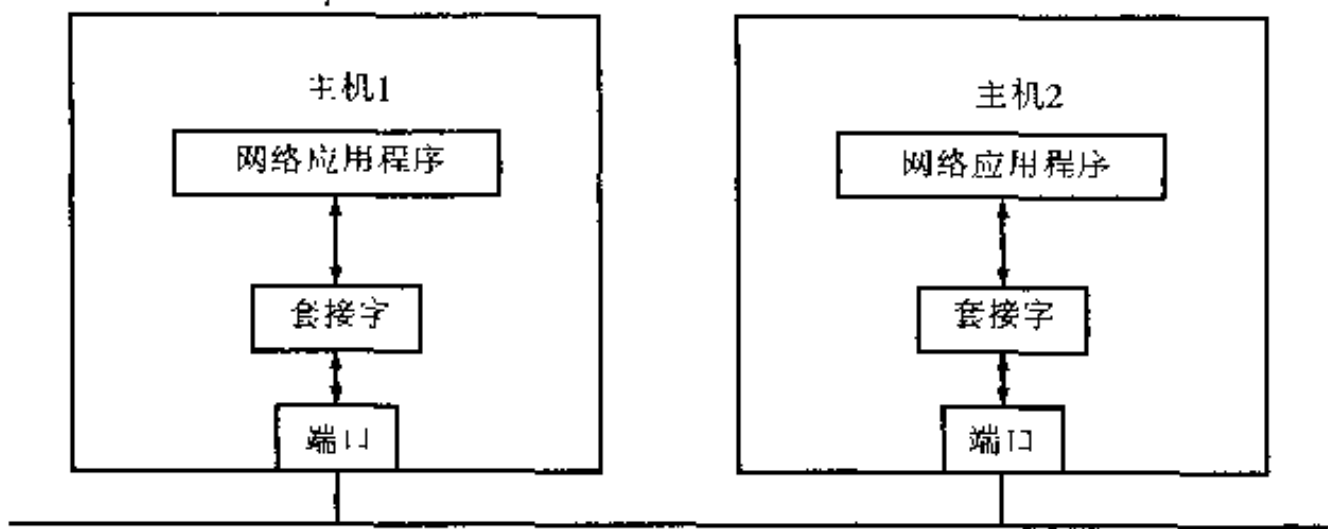


图3.1 套接字接口

UNIX提供了socket() C函数建立新的套接字：

```
int socket(int domain, int type, int protocol)
```

socket()函数返回一个套接字描述符，然后用这个套接字描述符在网上发送和接收数据（更详细内容以后叙述）。用于建立套接字的三个参数定义了通信使用的域（domain）、类型（type）和协议（protocol）。首先看一下可以使用的域值，见表3.1。

表3.1 套接字的域参数值

域值	说明
PF_UNIX	UNIX IPC通信
PF_INET	IPv4因特网协议，它是本书使用的协议类型
PF_INET6	IPv6因特网协议
PF_IPX	Novell协议
PF_NETLINK	Kernel用户接口驱动程序
PF_X25	ITU-T X.25 /ISO-8208协议
PF_AX25	业余爱好者无线AX.25协议
PF_ATMPVC	对原ATM PVC的访问
PF_APPLETALK	AppleTalk协议
PF_PACKET	低级包接口

类型（type）值定义了域中传输数据包时使用的网络通信类型。表3.2表示了可以使用的类型值。

表3.2 套接字的类型值

类型值	说明
SOCK_STREAM	使用面向连接的通信包
SOCK_DGRAM	使用无连接通信包
SOCK_SEQPACKET	使用有固定最大长度的面向连接的通信包
SOCK_RAW	使用原IP包
SOCK_RDM	使用不保证包次序的可靠数据报层

两个在IP通信中最通用的类型值是用于面向连接通信的SOCK_STREAM和用于无连接通信的SOCK_DGRAM。

用于建立套接字的特殊协议（protocol）值取决于用户选择使用哪种类型值。大多数套接字的类型（例如SOCK_STREAM和SOCK_DGRAM）只有用它们默认的协议（TCP用于SOCK_STREAM，UDP用于SOCK_DGRAM）时才能安全使用。为了指定默认的协议，用户可以在协议参数中指定零值，而不指定一般的协议值。

用以上指导方针为网络通信建立UNIX中的套接字是相当简单的事情。例如：

```
int newsocket;
newsocket = socket(PF_INET, SOCK_STREAM, 0);
```

以上例子创建了一个把数据传送到远程主机的标准TCP套接字。创建套接字本身并没有指定套接字连接到哪里，这个问题以后再介绍。

在套接字建立以后，用户就可以用返回值引用它；在上面的例子中，就是newsocket变量。UNIX还允许程序员修改套接字的一些字符，以便控制通信参数。下一节将介绍怎样在UNIX中设置套接字选项。

套接字选项

UNIX的套接字接口提供了一种方法，这种方法可以改变与套接字通信的协议参数；setsockopt()函数，该函数可以改变建立的套接字默认的行为。下面是setsockopt()函数的使用格式：

```
int setsockopt(int s, int level, int optname,
               const void *optval, socklen_t optlen);
```

- 参数s引用socket()函数创建的套接字。
- level参数引用变化的级。对于IP套接字，有两个可以使用的选项级：
 - SOL_SOCKET

• IPPROTO_IP

如果用户正在操作TCP套接字，也可以使用IPPROTO_TCP级。

每个变化的级都包含optname参数，该参数说明了要改变的套接字选项。optval和optlen参数定义了选项变化的值和长度。

用户最常见的一些套接字选项是SO_BROADCAST（它允许套接字发送广播消息）和IP_ADD_MEMBERSHIP（它允许套接字接收组播的数据包）。

网络地址

在建立了套接字之后，必须把它绑定到一个网络地址和端口对上。UNIX套接字系统使用IP地址和TCP或UDP端口的方式是套接字网络编程技术中比较容易混淆的一个部分。一个特殊的C结构sockaddr用于指定地址信息。sockaddr结构包含以下两个元素：

sa_family 一个地址系列，定义为短型

sa_data 一个设备地址，定义为14个字节

指定的地址系列（sa_family）允许sockaddr结构引用许多类型的地址。因此，14字节的地址元素（sa_data）很难直接使用。代替的办法是，UNIX提供了一个IP特有的地址结构sockaddr_in，它使用下列元素。使用sockaddr_in结构需要把正确的IP地址和端口值放在适当的数据元素中。

sin_family 一个地址系列，定义为短型

sin_port 一个端口号，定义为短型

sin_addr 一个地址，定义为长型（4字节）的IP地址

sin_data 8字节的填充符

为了总结这些函数的使用方法，下面的代码用于获得一个主机的IP地址和端口对：

```
struct sockaddr_in myconnection;  
myconnection.sin_family = AF_INET;  
myconnection.sin_addr.s_addr = inet_addr("192.168.1.1");  
myconnection.sin_port = htons(8000);
```

注意，sin_addr元素也是一个用元素指定网络地址的结构。s_addr元素用于表示IP地址。

现在读者已经知道怎样定义地址/端口对了，可以把套接字与IP地址相匹配，开始移动数据。用户必须根据套接字是面向连接的还是无连接的，在两种函数调用之间进行选择。下面几节将讲述通信类型和它们使用的方法之间的区别。

使用面向连接的套接字

IP连接领域不外乎两种通信类型：面向连接的（connection-oriented）和无连接的（connectionless）。在面向连接的套接字（使用SOCK_STREAM类型）中，使用TCP协议来建立两个IP地址端点之间的会话（连接）。建立连接涉及到相当大数量的开销，不过一旦建立了连接数据就可以在设备之间可靠地传输。为了建立面向连接的套接字，对于服务器程序和客户机程序必须使用单独的函数序列（见图3.2）。

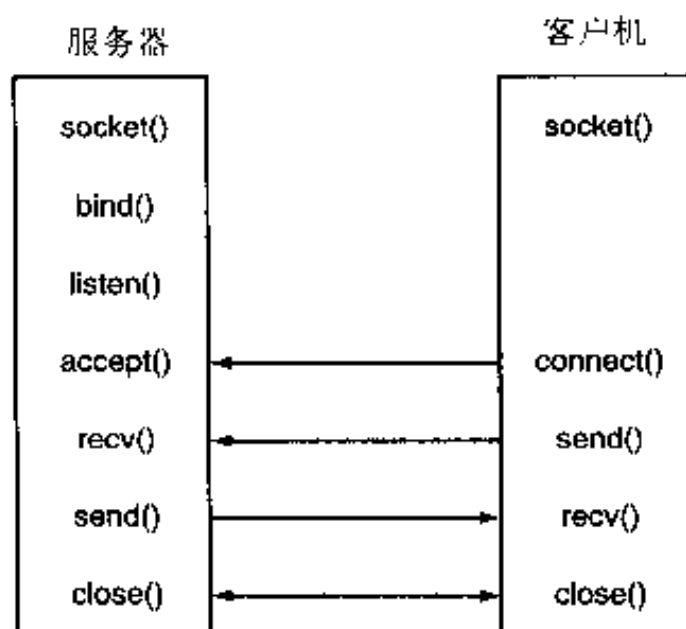


图3.2 面向连接套接字编程的函数

服务器函数

对于服务器程序，建立的套接字必须绑定到用于TCP通信的本地IP地址和端口号上。可以用UNIX的`bind()`函数来完成这项工作：

```
int bind(int socket, sockaddr *addr, int length);
```

在`bind()`函数中，`socket`参数引用从`socket()`函数返回的值。`addr`参数引用一个`sockaddr`地址/端口对，以定义本地网络连接。由于服务器通常接受在它自己IP地址上的连接，所以这是本地设备的IP地址和为应用程序分配的TCP端口。如果不知道本地系统的IP地址，可以使用`INADDR_ANY`值，允许套接字绑定在系统的任何一个本地地址上。

在套接字绑定到一个地址和端口之后，服务器程序必须准备接受从远程客户机的连接。这个过程分两步完成：首先，程序寻找一个进入的连接；下一步，发送和接收数据。

程序必须首先用`listen()`函数“收听”网络，寻找一个进入的连接。下一步它必须用`accept()`函数接受客户机的连接尝试。`listen()`函数的格式如下：

```
int listen(int socket, int backlog);
```

正如读者可以想得到的，`socket`参数引用`socket()`函数建立的套接字描述符。`backlog`参数引用系统可接受的正在等待进行的连接数。例如，假设这个值设定为2。如果有两个单独的客户机试图连接到该端口，系统就接受其中的一个连接用于处理，在第一个连接完成之前一直保留着另一个连接。如果有第三个连接的要求到来了，系统就会拒绝它，因为`backlog`的值已经满足了。

在`listen()`函数之后，必须调用`accept()`函数等待进入的连接。`accept()`函数的格式如下：

```
int accept(int socket, sockaddr *from, int *fromlen);
```

读者现在已经熟悉`socket`参数了。`from`和`fromlen`参数指向一个`sockaddr`地址结构和地址长度。当需要时，客户机的地址信息用这种结构存储。

在收到连接之后，服务器就可以调用`send()`和`recv()`函数，发送和接收客户机的数据了：

```
int send(int socket, const void *message, int length,  
        int flags)  
int recv(int socket, void *message, int length, int flags)
```

这里，**socket**还是引用连接的开放套接字。**message**参数或者引用待发送数据的缓冲器，或者引用接受数据的空缓冲器。**length**参数表示缓冲器的大小，**flags**参数指出是否需要特殊的标志（例如，为了在**TCP**包中把数据标记为紧急的）。

客户机函数

在面向连接的套接字中，客户机必须绑定到特定的主机地址和应用程序端口。对于客户机程序，使用**connect()**函数，而不用**listen()**函数：

```
int connect(int socket, sockaddr *addr, int addrlen);
```

像服务器函数中一样，**socket**参数引用建立的**socket()**函数值。**addr**参数指向一个包含远程IP地址和**TCP**端口号的创建的**sockaddr**结构。

一旦**connect()**函数成功了，客户机就连接到服务器，可以用标准的**send()**和**recv()**函数把数据从服务器传回或传到服务器去。

关闭连接

当客户机和服务器结束数据传送之后，应当用两个命令正确地结束连接：

- **shutdown(int socket, int how)**
- **close(int socket)**

close()函数有可能单独使用（读者会经常看到只用这个函数关闭连接的程序）。不过，更温和一点的方法是先用**shutdown()**函数，然后再用**close()**函数。**shutdown()**函数用**how**参数使程序员可以决定怎样才能得体地关闭连接，可用的选项如下：

- 0 不再可以接收包。
- 1 不再可以发送包。
- 2 不再可以发送和接收包。

用选择0或者1的方法使套接字不能再接收或发送数据，但还允许套接字完成“挂起”数据的发送或接收。在连接有机会赶出所有“挂起”的数据之后，**close()**函数就被调用来结束连接，没有任何数据的丢失。

使用无连接套接字

由于**SOCK_DGRAM**类型的套接字使用**UDP**协议，所以不需要在网络设备之间发送连接信息。因此，经常很难确定哪个设备是作为“服务器”的，哪个设备是作为“客户机”的。如果一个设备最初是在等待远程设备的信息，那套接字就必须用**bind()**函数绑定到一个本地地址/端口对上。完成绑定之后，该设备就可以从套接字发送数据或者从套接字接收接入的数据了。由于客户机设备没有建立到特定服务器地址的连接，所以**connect()**函数不需要用于**UDP**客户机程序。图3.3表示了编一个无连接套接字程序的操作次序。

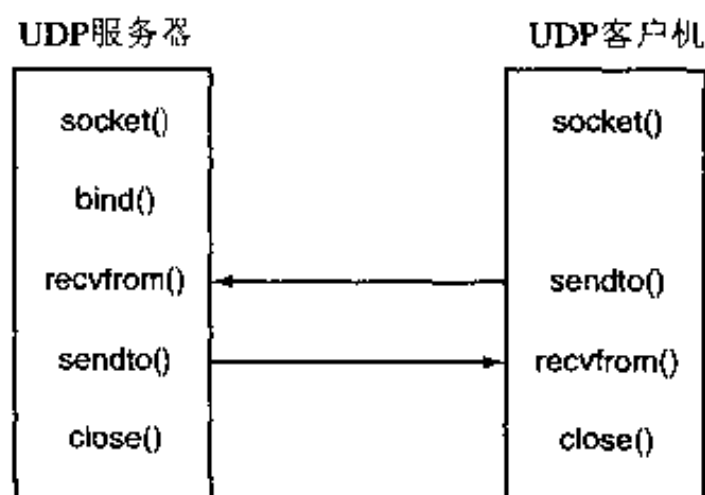


图3.3 无连接套接字编程操作

由于不存在确定的连接，所以普通的send()和recv()函数不能使用，因为不允许指定数据的目的地址。代替的作法是，套接字提供sendto()和recvfrom()函数：

```
int sendto(int socket, char *message, int length,
           int flags, sockaddr *dest, int destlength);
int recvfrom(int socket, char *message, int length,
             int flags, sockaddr *from, int *fromlength);
```

这两个函数使用UDP地址/端口对，给dest参数指定目的地址，用from参数给接收的包指定发送主机。在两个设备之间的通信结束之后，可以像TCP中使用的方法一样，对套接字使用shutdown()和close()函数。

无阻塞I/O方法

标准的UNIX网络编程模式有一个缺点，就是如果I/O函数不能立即处理，I/O函数就会阻塞（block）。阻塞指的是程序停止执行，等待着某个特殊语句的完成。例如，当一个程序执行到recv()函数的时候，程序会停止执行，一直等到在套接字上读到数据为止。实际上，在套接字上出现数据之前，recv()函数阻止了程序的进一步执行。如果远程设备不发送任何数据，程序就不能继续下去。

虽然这个原则在用户可以控制发送和接收数据方案的情况下，对于单一连接的客户机/服务器程序也许可以很好地工作，但是对于其他类型的不管发送或接收数据中的错误，必须连续处理其他事件的程序就会引起问题。有两种技术可以解决这个问题：使用无阻塞套接字或使用多路套接字技术。

无阻塞套接字

防止不希望的阻塞有一个简单初步的解决方案是，当调用I/O函数的时候把套接字设置成不阻塞的。可以用fcntl()函数把无阻塞特性设置为套接字上的一个特殊选项。fcntl()函数用于执行文件描述符上的各种低级操作，设置一个套接字的阻塞就是这些操作中的一个。

下面是fcntl()函数的格式：

```
int fcntl(int fd, int cmd, int arg)
```

fd参数应当是一个开放文件描述符（或者在这种情况下是套接字）。cmd参数指出对文件描

述符将进行什么操作。例如，用命令F_SETFL读或设置文件描述符的标志选项。arg参数用于指定要设置的标志（或查询）。

所以，为了把一个套接字设置成无阻塞模式，可用以下方法：

```
int newsocket;  
newsocket = sock(PF_INET, SOCK_STREAM, 0);  
fcntl(newsocket, F_SETFL, O_NONBLOCK);
```

O_NONBLOCK标志表示该套接字应当设置为无阻塞模式。不论何时，只要对newsocket套接字执行recv()函数，程序就不等待数据。如果数据没有立即出现，recv()函数就返回一个-1值，UNIX的errno值就会送给EWOULDBLOCK。

使用无阻塞套接字，用户可以选择任何开放套接字寻找接入的数据，或者决定是否它准备发出数据。

多路套接字

套接字阻塞问题的另外一个解决方案是使用select()函数多路传输所有有效的套接字。select()函数使用户能够看到多个事件的套接字（例如要从套接字读出的数据或写入的数据），只处理需要处理的套接字。没有任何未决问题的套接字被跳过，所以它们不会阻塞程序的执行。

select()函数的格式如下：

```
int select(int numfd, fd_set *readfds, fd_set *writefds,  
           fd_set *exceptfds, struct timeval *timeout)
```

numfd参数指定文件描述符（套接字）的最大值，这个值是被选择函数监测到的值再加1。这样，在测试套接字设置的时候select()函数就知道要循环多少次。

readfds、writefds和exceptfds三个参数为特定的数据函数指出select()监视的套接字的以下列表项（或设置）：

readfds 检查套接字是否数据可以读

writefds 检查套接字是否准备写数据

exceptfds 检查套接字是否异常

timeout参数指出一个timeval结构，以设置任何套接字发生一个事件时select()函数应当等待多长时间。

套接字多路传输中最难处理的问题就是把套接字分配到readfds、writefds和exceptfds参数。的确，还有另外一个完成这项工作的整个函数集，如表3.3中所示。

表3.3 select()助手函数

函数	说明
FD_ZERO(set)	把多路传输集的set置为零
FD_SET(socket, set)	把套接字加到多路传输集的set
FD_CLR(socket, set)	从多路传输集的set中删除套接字
FD_ISSET(socket, set)	检查多路传输集的set中是否包含套接字

在每次调用select()时必须用助手函数设置每一个套接字集。一次select()调用删除以前所有的select()调用。这样,在下一次调用select()函数之前,必须从现有的集中添加或删除所有新的套接字。

下面是一个使用select()方法的样例:

```
sock1 = socket(PF_INET, SOCK_STREAM, 0);
sock2 = socket(PF_INET, SOCK_STREAM, 0);
connect(sock1, addr, addrlen);
connect(sock2, addr2, addr2len);
FD_ZERO(&sockset);
FD_SET(sock1, &sockset);
FD_SET(sock2, &sockset);
if (sock1 > sock2)
    maxfd = sock1 + 1;
else
    maxfd = sock2 + 1;
timeout.tv_sec = 30;
timeout.tv_usec = 0;
select(maxfd, &sockset, NULL, NULL, &timeout);
if (FD_ISSET(sock1, &sockset))
    recv(sock1, buffer1, sizeof(buffer1), 0)
if (FD_ISSET(sock2, &sockset))
    recv(sock2, buffer2, sizeof(buffer2), 0)
```

从这个例子可以看出, select()函数是怎样用来监视两个单独的套接字连接的。一旦用适当的套接字集调用了select()函数,任何时候用户都可以在程序中用FD_ISSET助手函数来检查数据是否可以用于套接字。在select()函数结束之后(或者接收到了一个事件,或者超过时间), sockset值只包含那些已经有事件触发器的套接字。由于使用FD_ISSET,用户可以判断是否两个套接字都收到了数据,如果两个套接字都没有数据,它就不是集合的一部分,就不会阻塞其余的程序运行。

Windows中的套接字编程技术

当读者熟悉了在UNIX环境中的网络编程之后, Windows网络编程就容易理解了。这一节将介绍Windows网络编程接口与UNIX网络编程模式之间有什么关系,以及Windows套接字编程是怎样形成.NET框架网络类的基础的。

Windows套接字函数

可以说Windows网络编程技术的模式是从对照UNIX模式得到的。Windows操作系统的许多特性的根源都在UNIX系统中。许多Windows网络编程技术都模仿UNIX Berkeley(伯克利)套接字方法。它称为(不要惊讶)Windows套接字,或简称为Winsock。Winsock接口的目的是,使来自UNIX环境的网络程序员们不需要经过大量学习的曲折过程,就能够容易地把现有的网络程序用于Windows环境,或在Windows环境中创建新的网络程序。

Winsock API作为开发者的报头和库文件集的实现，应用程序使用DLL（动态链接库）文件。有两种基本Winsock库的版本：1.1版最初是与Windows 95工作站一起发表的，它提供了基本的套接字功能。后来，版本2作为Windows 95机器的添加版发表。它明显增加了更多的套接字函数和可以被网络程序员采纳的协议。在Windows 98发表的时候，Winsock库已经是成熟的2.2版了，该版本仍然是当前各Windows操作系统版本的组成部分。

说明：这种配置方案只有Windows CE平台是例外。在编写本书的时候，Windows CE仍然只支持Winsock 1.1库。

当然，Winsock环境的核心是套接字。正像在UNIX中一样，为了建立与Windows系统中下层网络接口的链接，所有Winsock网络程序都建立了套接字。所有在UNIX领域中使用的标准套接字函数调用都可以用于Windows系统。不过，UNIX套接字与Winsock之间还有一些区别。下面几节将讲述这些区别。

WSAStartup()函数

为了启动Winsock程序，首先要调用WSAStartup()函数。该函数会通知操作系统，程序需要使用哪个版本的Winsock。操作系统会装载套接字函数操作时需要用的Winsock库。

WSAStartup()函数的格式如下：

```
int WSAStartup(WORD wVersion, LPWSADATA lpWSADATA)
```

第一个参数指出了程序要求的版本。如果程序要求2.2版的Winsock，可现在只有1.1版的可用，WSAStartup()函数会返回一条错误信息。不过，如果应用程序要求1.1版，而加载了2.2版的Winsock，函数可以成功操作。

当函数操作成功之后，lpWSADATA参数指向一个结构，该结构包含Winsock库加载之后有关该Winsock库的信息，例如系统中使用的实际Winsock版本。然后，这些信息可以用来确定程序运行的系统所具有的网络性能。

WSACleanup()函数

当Winsock程序运行结束时，它必须释放Winsock库。WSACleanup()用于每个Winsock程序的结尾，表示程序不再使用其他Winsock函数，可以释放Winsock库了。WSACleanup()函数不使用任何参数，它只是程序中Winsock函数结束的信号。在WSACleanup()函数之后如果使用任何一个Winsock函数，都会引起错误的结果。

Winsock函数

在WSAStartup()和WSACleanup()两个函数之间，Winsock程序的行为与UNIX套接字程序完全一样，使用socket()、bind()、connect()、listen()和accept()调用。事实上，Winsock接口使用与UNIX同样的地址结构(sockaddr_in)和用同样的值定义协议族和类型（例如，SOCK_STREAM协议族）。这样做的目的就是尽可能容易地把UNIX网络程序移植到Windows环境中。

除了标准的UNIX网络函数之外，Winsock第2版接口还包括一组它自己的网络函数，这些函数的前面都冠以WSA。这些函数扩展了标准UNIX网络函数的功能。例如，WSARecv()函数可以代替标准的UNIX recv()函数调用。不过，WSARecv()在原函数调用的基础上增加了两个额外的参数，允许Windows特有的建立重叠I/O和部分数据报通告的功能。图3.4显示了怎样用Winsock WSA函数代替标准的UNIX函数。

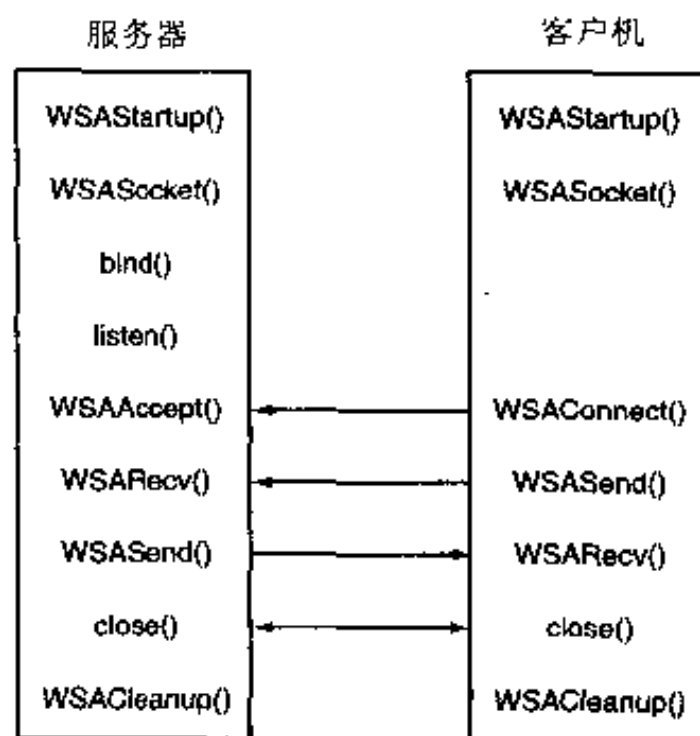


图3.4 Winsock WSA用于服务器和客户机的编程函数

Winsock无阻塞套接字函数

另一个与UNIX网络环境类似的地方是，Winsock也提供了一些方法，防止网络I/O函数阻塞程序的执行。Winsock支持标准的UNIX方法，这种方法用ioctlsocket()函数把一个套接字设置成无阻塞的（类似UNIX的fcntl()函数），select()函数多路传输多个套接字。

ioctlsocket()函数的格式如下：

```
ioctlsocket(SOCKET s, long cmd, u_long FAR* argp)
```

被修改的套接字是s，cmd参数指出要对套接字进行哪种操作，argp参数用于指定命令参数。

除了这些标准的套接字函数之外，Winsock接口额外还提供了一些允许无阻塞网络I/O的方法。

WSAAsyncSelect()函数

Windows区别于标准UNIX程序的一个特点是事件（events）的概念。与有一组执行方法的普通结构化程序不同的是，Windows程序通常是事件驱动的。程序中方法被执行是对于程序运行时事件出现的响应——例如按钮被单击、菜单项被选择等。等待数据在网络套接字上出现的标准技术不适合Windows中的事件模式。对网络套接字的事件驱动访问是解决这个问题的办法。

WSAAsyncSelect()函数扩展了标准UNIX select()函数的功能，它允许Windows进行查询套接字的操作。创建的WSAAsyncSelect()方法包括要监视的套接字和一个Windows消息值，

当某个套接字事件出现的时候（例如要读的数据可以访问的时候，或套接字准备好接受写数据的时候），该消息值就会被送到窗口。WSAAsyncSelect()函数的格式如下：

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

要监视的套接字用s参数指定，接收事件消息的父窗口用hWnd参数指定。要发送的实际事件用wMsg参数指定。最后一个参数lEvent指定为套接字监视的事件。用户也可以用对表3.4中的事件执行位“或（OR）”操作的方法，为一个套接字监视多个事件。

表3.4 WSAAsyncSelect()事件类型

事件	说明
FD_ACCEPT	用套接字建立一个新的连接
FD_ADDRESS_LIST_CHANGE	为套接字的协议族而改变的本地地址列表
FD_CLOSE	一个现有的连接被关闭
FD_CONNECT	已经完成与远程主机连接的套接字
FD_GROUP_QOS	套接字组的服务质量值已经改变
FD_OOB	套接字收到了带外的数据
FD_QOS	套接字的服务质量值已经改变
FD_READ	套接字有准备读的数据
FD_ROUTING_INTERFACE_CHANGE	套接字的路由接口已经改变到一个特殊的目的地
FD_WRITE	套接字准备写数据

一个WSAAsyncSelect()函数的样例如下所示：

```
WSAAsyncSelect(sock, hwnd, WM_SOCKET, FD_READ | FD_CLOSE);
```

在这个例子中，如果套接字有可以读的数据，或者如果远程主机关闭了连接，WM_SOCKET消息就会送到Window消息的wParam中的hwnd窗口。然后由hwnd窗口负责检测和处理WM_SOCKET消息，并根据哪个事件被触发来执行相应的函数。这几乎总是在一个Windows过程（WindowProc）方法中为使用case语句的窗口进行处理。

WSAEventSelect()函数

WSAEventSelect()不是用Windows消息处理套接字通告，它是用事件对象句柄（event object handle）来处理。事件对象句柄是当惟一事件被触发时在被调用程序中定义的一个独立的方法。这个技术允许用户建立单独的方法来处理各种套接字事件。

为了使用这种技术，必须首先用WSACreateEvent()函数定义一个惟一的事件。在事件创建之后，还必须用WSAEventSelect()函数与一个套接字匹配：

```
WSASelect(SOCKET s, WSAEVENT hEvent, long lNetworkEvents)
```

通常情况下，s参数指定要监视的套接字，hEvent指定当套接字事件发生的时候将被调用的创建事件。与WSAAsyncSelect()函数类似，lNetworkEvent参数也是所有监视的套接字事件按位（bitwise）组合的。像对WSAAsyncSelect()函数一样，对WSAEventSelect()函数使用相同的事件定义。当一个套接字事件出现的时候，由WSACreateEvent()注册的事件方法就被执行。

重叠I/O

可能Winsock接口最好的特性之一就是重叠I/O的概念了。这种技术用一种特殊的数据结构允许程序一次公布一个或多个异步I/O请求。这种数据结构（WSAOVERLAPPED）定义多个匹配在一起的套接字和事件对象。这些事件被看做是重叠的，在多重事件中可以作为套接字接收事件被同时调用。

为了使用这种重叠技术，必须用重叠允许标志（socket()函数不包括这个标志）以WSASocket()函数调用建立套接字。同样，所有的数据通信必须用WSARecv()和WSASend()函数完成。这些Winsock函数用重叠I/O标志表示数据使用WSAOVERLAPPED数据结构。

虽然使用重叠I/O可以大大改善网络程序的性能，但它解决不了所有可能出现的难题。重叠I/O技术的一个缺点就是它最多只能定义64个事件。对于一个需要几百个连接的大型网络应用程序来说，这个技术无法起作用。

完成端口

重叠I/O技术另外一个不利的特点是，所有的事件都在程序中的一个线程中处理。为了允许在线程中把事件分开，Windows采用了完成端口（completion port）方法。完成端口允许程序员在一个程序中定义多个线程，并把多个事件分配给单个的线程。把重叠I/O技术与完成端口方法相结合，程序员就可以用分散的程序线程处理重叠的套接字事件。在含有多个处理器的系统中，这一技术产生了真实的和引人注目的结果。用为每个处理器建立单独线程的办法，可以使多个套接字在每个处理器上同时被监视。

C#套接字编程技术

创建的.NET框架网络类给用.NET编程语言的程序员提供了到本地Winsock网络API的方便接口。现在，读者已经知道Winsock API是怎样处理网络编程的，已经作好了准备可以探索C#处理套接字的方法了。下面几节将介绍C#网络编程的类和怎样在网络程序中使用这些类。

C#中的IP地址

在.NET网络库中用户会注意到最大的一个优点就是IP地址和端口对的处理方法。这是一个相当直截了当的处理过程，它对于旧的、令人混淆的UNIX方法进行了受到欢迎的改进。.NET在System.Net命名空间定义了两个类，以处理各种类型的IP地址信息：

- IPAddress
- IPEndPoint

IPAddress类

IPAddress对象用于表示一个IP地址。这个值然后可以在各种不同的套接字方法中用来代表IP地址。IPAddress默认的构造函数如下：

```
public IPAddress(long address)
```

默认的构造函数取一个长值，并把它转换成一个IPAddress值。实际上，几乎从来不用这个默认值（读者有多少次碰巧看到长值的IP地址呢？）。在IPAddress类中，可以用几个方法来创建和操作IP地址。表3.5定义了这些方法。

表3.5 IPAddress方法

方法	说明
Equals	比较两个IP地址
GetHashCode	对一个IPAddress对象返回一个散列值
GetType	返回IP地址实例的类型
HostToNetworkOrder	把IP地址从主机字节次序转换成网络字节次序
IsLoopBack	指出是否认为IP地址是回送地址
NetworkToHostOrder	把IP地址从网络字节次序转换成主机字节次序
Parse	把一个字符串转换成一个IPAddress实例
ToString	把一个IPAddress转换成该IP地址用点分十进制格式表示的字符串

Parse()方法最经常用于创建IPAddress实例：

```
IPAddress newaddress = IPAddress.Parse("192.168.1.1");
```

这种格式允许用户在字符串格式中使用标准的点分四节IP地址，并把它转换成一个IPAddress对象。

IPAddress类还提供了4个只读字段，它们代表程序中使用的特殊IP地址：

- Any** 用于代表本地系统可用的任何IP地址
- Broadcast** 用于代表本地网络的IP广播地址
- Loopback** 用于代表系统的回送地址
- None** 用于代表系统上没有网络接口

程序清单3.1是一个演示使用IPAddress类的方法和字段的样例程序。

程序清单3.1 AddressSample.cs程序

```
using System;
using System.Net;

class AddressSample
{
    public static void Main ()
    {
        IPAddress test1 = IPAddress.Parse("192.168.1.1");
        IPAddress test2 = IPAddress.Loopback;
        IPAddress test3 = IPAddress.Broadcast;
        IPAddress test4 = IPAddress.Any;
        IPAddress test5 = IPAddress.None;
        IPHostEntry ihe =
            Dns.GetHostByName(Dns.GetHostName());
        IPAddress myself = ihe.AddressList[0];
        if (IPAddress.IsLoopback(test2))
            Console.WriteLine("The Loopback address is: {0}",
                               test2.ToString());
        else
            Console.WriteLine("Error obtaining the loopback address");
        Console.WriteLine("The Local IP address is: {0}\n",
                           myself.ToString());
        if (myself == test2)
            Console.WriteLine("The loopback address is the same as local address.\n");
        else
            Console.WriteLine("The loopback address is not the local address.\n");
        Console.WriteLine("The test address is: {0}",
                           test1.ToString());
        Console.WriteLine("Broadcast address: {0}",
                           test3.ToString());
        Console.WriteLine("The ANY address is: {0}",
                           test4.ToString());
        Console.WriteLine("The NONE address is: {0}",
                           test5.ToString());
    }
}
```

AddressSample.cs程序显示出用IPAddress对象可以做到的一些事情。比较有趣的一件事是用这个方法可以获得本地IP地址:

```
IPHostEntry ihe =
    Dns.GetHostByName(Dns.GetHostName());
IPAddress myself = ihe.AddressList[0];
```

这个代码在第2章“IP编程基础”中介绍过。它是用System.Net.Dns类的GetHostByName()和GetHostName()方法来确定本地IP地址和建立了一个IPHostEntry对象。第4章“DNS和C#”将要更详细地介绍IPHostEntry对象，不过现在只需要知道它包含AddressList属性，该属性是IPAddress对象的数组就足够了。AddressSample.cs程序取出列表中的第一个地址，并把它分配给myselfIP地址对象。

这个程序的输出应当如下所示：

```
C:\>AddressSample
The Loopback address is: 127.0.0.1
The Local IP address is: 192.168.1.6

The loopback address is not the local address.

The test address is: 192.168.1.1
Broadcast address: 255.255.255.255
The ANY address is: 0.0.0.0
The NONE address is: 255.255.255.255

C:\>
```

这个程序输出最有趣的地方是它输出的结果为Any（任何）和None（无）地址。这些值可能看起来与读者想像的不一样；Any IPAddress对象指向0.0.0.0地址，可能读者会以为它什么都不代表。不过，当一个系统有多个网络接口，而用户不想把套接字绑定到任何特定接口的时候，这个地址是最经常使用的。None IPAddress对象指向255.255.255.255地址，它经常用于当系统想创建一个伪套接字，而且不想把它绑定到任何接口的时候。

IPEndPoint类

类似于UNIX的sockaddr_in结构，.NET框架使用IPEndPoint对象来表示指定的IP地址/端口组合。当把套接字绑定到本地地址的时候，或者在把套接字连接到远程地址的时候，使用IPEndPoint对象。我们首先来研究一下IPEndPoint，然后再看一下它在程序中所起的作用。

有两个创建IPEndPoint实例的构造函数：

- IPEndPoint(long address, int port)
- IPEndPoint(IPAddress address, int port)

两个构造函数都使用两个参数：一个IP地址值，或者用长值表示或者用一个IPAddress对象表示；一个整数端口号。读者可能已经猜到，最经常使用的是IPAddress格式的构造函数。

表3.6说明了可以用于IPEndPoint对象的各种方法。

SocketAddress类是System.Net命名空间中的一个特殊的类。它代表IPEndPoint对象的一系列版本。可以用这个类存储IPEndPoint实例，然后再用IPEndPoint.Create()方法重新创建该实例。SocketAddress类的格式如下：

- 1字节表示对象的AddressFamily。
- 1字节表示对象的大小。

- 2字节表示对象的端口号。
- 其余字节表示对象的IP地址。

表3.6 IPEndPoint方法

方法	说明
Create	从一个SocketAddress对象创建一个EndPoint对象
Equals	比较两个IPEndPoint对象
GetHashCode	从一个IPEndPoint对象返回一个散列值
GetType	返回IPEndPoint实例的类型
Serialize	创建IPEndPoint实例的SocketAddress实例
ToString	创建一个代表SocketAddress实例的字符串

除了方法之外，IPEndPoint类还包含三个可以设置的或者从实例中获得的属性：

Address 得到或设置IP地址属性

AddressFamily 得到IP地址系列

Port 得到或设置TCP或UDP端口号

以上每一个属性都可以与IPEndPoint实例一起使用，来获得关于IPEndPoint对象单独端口的信息。Address和Port属性也可以在现有的IPEndPoint对象中设置单个的值。

还有两个字段可以与IPEndPoint对象一起使用，以便从系统中获得可用的端口范围：

MaxPort 可以指定到端口号的最大值

MinPort 可以指定到端口号的最小值

IPEndPoint的操作范例

程序清单3.2显示了一个样例程序，演示了IPEndPoint类以及它的方法、属性和字段。

程序清单3.2 IPEndPointSample.cs程序

```
using System;
using System.Net;

class IPEndPointSample
{
    public static void Main ()
    {
        IPAddress test1 = IPAddress.Parse("192.168.1.1");
        IPEndPoint ie = new IPEndPoint(test1, 8000);

        Console.WriteLine("The IPEndPoint is: {0}",
                           ie.ToString());
        Console.WriteLine("The AddressFamily is: {0}",
```

```

        ie.AddressFamily);
    Console.WriteLine("The address is: {0}, and the ➡
        port is: {1}\n", ie.Address, ie.Port);
    Console.WriteLine("The min port number is: {0}",
        IPEndPoint.MinPort);
    Console.WriteLine("The max port number is: {0}\n",
        IPEndPoint.MaxPort);

    ie.Port = 80;
    Console.WriteLine("The changed IPEndPoint value ➡
        is: {0}", ie.ToString());
    SocketAddress sa = ie.Serialize();
    Console.WriteLine("The SocketAddress is: {0}",
        sa.ToString());
}
}

```

IPEndPointSample.cs程序演示了几个重要的**IPEndPoint**特性。注意，可以把全部的**IPEndPoint**对象显示成一个字符串，也可以提取对象的某一部分：

```

    Console.WriteLine("The IPEndPoint is: {0}",
        ie.ToString());
    Console.WriteLine("The AddressFamily is: {0}",
        ie.AddressFamily);
    Console.WriteLine("The address is: {0}, and the ➡
        port is: {1}\n", ie.Address, ie.Port);

```

该程序也演示了怎样用**Port**属性单独地改变**IPEndPoint**对象的端口值：

```
ie.Port = 80;
```

这使得用户可以在对象中修改单个的地址和端口值，不需要创建新的对象。

这个程序的输出看起来应当如下所示：

```

C:\>IPEndPointSample
The IPEndPoint is: 192.168.1.1:8000
The AddressFamily is: InterNetwork
The address is: 192.168.1.1, and the port is: 8000

The min port number is: 0
The max port number is: 65535

The changed IPEndPoint value is: 192.168.1.1:80
The SocketAddress is: ➡
InterNetwork:16:{0,80,192,168,1,1,0,0,0,0,0,0,0,0,0,0}

C:\>

```


使用C#套接字

`System.Net.Sockets`命名空间包含一些可以给低级的Winsock API提供实际.NET接口的类。这一节将简要地概述C#的Socket（套接字）类。后续的章节将在概述的基础上对几种类型的套接字加以详细介绍和举例说明。

套接字的结构

`System.Net.Sockets`命名空间的核心是Socket（套接字）类。它提供了Winsock API的C#管理代码的实现。Socket类的构造函数如下：

```
Socket(AddressFamily af, SocketType st,
        ProtocolType pt)
```

可以看出，Socket构造函数的基本格式就是模仿了原来的UNIX `socket()`函数。它用三个参数定义创建的套接字类型：

- 用AddressFamily定义网络类型
- 用SocketType定义数据连接的类型
- 用ProtocolType指定具体的网络协议

这些参数的每一个都可以用`System.Net.Sockets`命名空间中一个单独的枚举来表现。每个枚举都包含可以使用的值。对网络上的IP通信，AddressFamily应当总是使用AddressFamily.InterNetwork值。使用InterNetwork AddressFamily，SocketType参数必须与特定的ProtocolType参数相匹配。不允许用户把SocketTypes和ProtocolTypes混淆匹配。表3.7表示了可用于IP通信的组合。

表3.7 IP套接字定义组合

套接字类型	协议类型	说明
Dgram	Udp	无连接通信
Stream	Tcp	面向连接的通信
Raw	Icmp	Internet控制消息协议
Raw	Raw	简单IP包通信

使用枚举值让用户容易记住所有的选项（虽然确实使Socket()语句变得相当长！）。例如：

```
Socket newsock = Socket(AddressFamily.InterNetwork,
                          SocketType.Stream, ProtocolType.Tcp);
```

如表3.8中所说明的那样，可以用Socket类的几个属性从创建的Socket对象中检索信息。

表3.8 套接字属性

属性	说明
AddressFamily	得到套接字的地址系列
Available	得到准备读的数据数量
Blocking	得到或设置是否套接字处于阻塞模式
Connected	得到一个值，该值显示是否套接字连接到远程设备
Handle	得到操作系统对套接字的控制柄
LocalEndPoint	得到套接字的本地EndPoint对象
ProtocolType	得到套接字的协议类型
RemoteEndPoint	得到套接字的远程EndPoint信息
SocketType	得到套接字的类型

说明：除了LocalEndPoint和RemoteEndPoint两个属性之外，所有Socket类的属性都可以在套接字创建之后立即使用。LocalEndPoint和RemoteEndPoint属性只用于绑定的套接字。

程序清单3.3是一个演示Socket属性的简单程序。由于LocalEndPoint属性需要一个绑定的Socket对象，我用Bind()方法把该套接字绑定到系统的回送地址（127.0.0.1）了。

程序清单3.3 SockProp.cs套接字属性样例程序

```
using System;
using System.Net;
using System.Net.Sockets;

class SockProp
{
    public static void Main ()
    {
        IPAddress ia = IPAddress.Parse("127.0.0.1");
        IPEndPoint ie = new IPEndPoint(ia, 8000);

        Socket test = new Socket(AddressFamily.InterNetwork,
                                  SocketType.Stream, ProtocolType.Tcp);

        Console.WriteLine("AddressFamily: {0}",
                           test.AddressFamily);
        Console.WriteLine("SocketType: {0}",
                           test.SocketType);
        Console.WriteLine("ProtocolType: {0}",
                           test.ProtocolType);
        Console.WriteLine("Blocking: {0}", test.Blocking);
    }
}
```

```
test.Blocking = false;
Console.WriteLine("new Blocking: {0}", test.Blocking);
Console.WriteLine("Connected: {0}", test.Connected);
test.Bind(ie);
IPEndPoint iep = (IPEndPoint)test.LocalEndPoint;
Console.WriteLine("Local EndPoint: {0}",
    iep.ToString());

test.Close();
}
}
```

程序的输出看起来应当如下所示:

```
C:\>SockProp
AddressFamily: InterNetwork
SocketType: Stream
ProtocolType: Tcp
Blocking: True
New Blocking: False
Connected: False
Local EndPoint: 127.0.0.1:8000

C:\>
```

用把Blocking属性设置为假(false)的方法, 用户可以使用无阻塞套接字, 类似于UNIX的fcntl()函数。不过, 详细内容后面还要介绍。

套接字选项

像UNIX套接字一样, .NET套接字允许用户为创建Socket对象设置协议选项。不过, 由于C#是一种面向对象的语言, 所以有一点曲折。

SetSocketOption()不是一个函数, 它叫做一个方法。像它的UNIX对应函数一样, SetSocketOption()可以配置用户想调整的套接字参数, 使通信参数用户化。SetSocketOption()方法是超负荷的, 使用三种不同的格式:

```
SetSocketOption(SocketOptionLevel sl,
    SocketOptionName sn, byte[] value)
SetSocketOption(SocketOptionLevel sl,
    SocketOptionName sn, int value)
SetSocketOption(SocketOptionLevel sl,
    SocketOptionName sn, object value)
```

使用的参数类似于UNIX setsockopt()函数。sl定义要设置的套接字选项的类型。表3.9列出了可用的SocketOptionLevels值。

表3.9 SocketOptionLevel值

值	说明
IP	对IP套接字的选项
Socket	对套接字的选项
Tcp	对TCP套接字的选项
Udp	对UDP套接字的选项

sn指定了将在SocketOptionLevel中设置的具体套接字选项。表3.10列出了可用的SocketOptionNames值。

表3.10 SocketOptionName值

值	SocketOptionLevel	说明
AcceptConnection	Socket	如果为真，套接字为收听模式
AddMembership	IP	加上IP组成员的数目
AddSourceMembership	IP	增加一个源组
BlockSource	IP	阻碍来自源的数据
Broadcast	Socket	如果为真，允许发送广播消息
BsdUrgent	IP	使用紧急数据（只能发送一次，不能关闭）
ChecksumCoverage	Udp	设置或得到UDP校验和覆盖
Debug	Socket	如果为真就记录调试信息
DontFragment	IP	IP包不分成小片段
DontLinger	Socket	不等待数据，温和地关闭套接字
DontRoute	Socket	直接把包发送到接口地址
DropMembership	IP	减少IP组成员的数目
DropSourceMembership	IP	减少一个源组
Error	Socket	获得和清楚错误状态
ExclusiveAddressUse	Socket	使套接字被绑定以便惟一的访问
Expedited	IP	使用加快数据（只能发送一次，不能关闭）
HeaderIncluded	IP	指出发送到套接字的数据将包括IP头
IPOptions	IP	指定IP选项用于输出包中
IpTimeToLive	IP	设置IP包的寿命值

(续表)

值	SocketOptionLevel	说明
KeepAlive	Socket	发送TCP保留包
Linger	Socket	在关闭套接字之后等待额外数据
MaxConnections	Socket	设置使用的最大队列长度
MulticastInterface	IP	设置组播包使用的接口
MulticastLoopback	IP	IP组播回送
MulticastTimeToLive	IP	设置IP组播的寿命
NoChecksum	Udp	发送校验和设置为零的UDP包
NoDelay	Tcp	禁止对TCP包的Nagle算法
OutOfBandInline	Socket	允许接收带外 (out-of-band) 数据
PacketInformation	IP	返回关于接收包的信息
ReceiveBuffer	Socket	设置为接收包而保留的每个套接字缓冲器的总数
ReceiveLowWater	Socket	收到低水印 (water mark)
ReceiveTimeout	Socket	收到超时值
ReuseAddress	Socket	允许套接字被绑定在一个已经使用的端口地址
SendBuffer	Socket	设置为发送包而保留的每个套接字缓冲器的总数
SendLowWater	Socket	发送低水印 (water mark)
SendTimeout	Socket	发送超时值
Type	Socket	得到套接字类型
TypeOfService	IP	设置服务字段的IP类型
UnblockSource	IP	设置套接字为无阻塞模式
UseLoopback	Socket	可能时绕开该网络接口

value参数指定所使用的套接字选项名的值。根据使用的SocketOptionName不同, 值的格式也不同。

在创建和修改了套接字之后, 就已经准备就绪, 或者等待接入的连接, 或者连接到远程设备。下面几节将介绍怎样在C#中用面向连接的和无连接的通信套接字用C# Socket类进行通信。

使用面向连接的套接字

同样, .NET框架的概念与UNIX网络编程技术类似。在.NET框架中, 用户可以跨过网络

与远程主机建立面向连接的通信。由于C#是一个面向对象的语言，所以所有UNIX套接字函数都作为Socket类的方法实现。借助于从Socket实例对方法的引用，用户可以用显示的套接字执行网络操作。

说明：这一节介绍面向连接的通信所使用的Sockets类中的C#方法。第5章“面向连接的套接字”将展开讨论这个问题，向读者展示更多详细信息和多个范例。

服务器函数

与UNIX服务器类似，在服务器套接字建立之后，必须把它绑定到系统中的一个本地网络地址。**Bind()**方法就用于完成这一功能：

```
Bind(EndPoint address)
```

address参数必须指向一个有效的**IPEndPoint**实例，该实例包括一个本地IP地址和一个端口号。在套接字绑定到本地地址之后，就用**Listen()**方法等待客户机发出的连接尝试：

```
Listen(int backlog)
```

backlog参数指出系统等待用户程序服务排队的连接数。超过等待连接数的任何客户的尝试都会被拒绝。用户应当记住，在这里指定一个人的编号可能会造成用户的服务器程序被执行的结果。每个待解决连接尝试都暂存在TCP缓冲器区中的缓冲器空间，这意味着只有较少的缓冲器空间用于发送和接收包。

在**Listen()**方法执行过之后，服务器已经做好了接受任何引进连接的准备。这是用**Accept()**方法来完成的，该方法返回一个新的套接字描述符，这个连接所有的通信调用都可以使用该描述符。

下面是一个C#服务器代码的例子，设置了必要的套接字片段：

```
IPHostEntry local = Dns.GetHostByName(Dns.GetHostName());
IPEndPoint iep = new TPEndPoint(local.AddressList[0],
                                8000);
Socket newserver = new Socket(AddressFamily.InterNetwork,
                               SocketType.Stream, ProtocolType.Tcp);
newserver.Bind(iep);
newserver.Listen(5);
Socket newclient = newserver.Accept();
```

这个程序会在**Accept()**语句处暂停，等待一个客户机的连接。一旦一个客户机连接到服务器，**newclient** Socket对象将含有新的连接信息，这些信息应当在与该远程客户机的一切通信中被利用。**newserver** Socket对象将仍然绑定到原来的**IPEndPoint**对象，并能用于接受更多与其他**Accept()**方法的连接。如果没有其他的**Accept()**方法被调用，客户机将不响应任何其他客户机连接尝试。

在客户机连接被接受之后，客户机和服务器就可以开始传递数据了。**Receive()**和**Send()**方法用于执行传送功能。这两种方法用方法的四种格式重复加载，表3.11显示了每种方法可用的格式。

表3.11 Receive()和Send()方法

方法	说明
Receive(byte[] data)	接收数据，并把数据放在指定的字节数组
Receive(byte[] data, SocketFlags sf)	设置套接字属性，接收数据，并把数据放在指定的字节数组
Receive(byte[] data, int size, SocketFlags sf)	设置套接字属性，接收指定尺寸的数据，并把数据存放在指定的字节数组
Receive(byte[] data, int offset, int size, SocketFlags sf)	设置套接字属性，接收size字节的数据，并以偏移量offset把数据存放在data字节数组中
Send(byte[] data)	发送在字节数组中指定的数据
Send(byte[] data, SocketFlags sf)	设置套接字属性，并发送在字节数组中指定的数据
Send(byte[] data, int size, SocketFlags sf)	设置套接字属性，并发送在指定的字节数组中指定尺寸的数据
Send(byte[] data, int offset, int size, SocketFlags sf)	设置套接字属性，并在data字节数组中从偏移量offset开始发送size字节的数据

Send()和Receive()方法的简单格式发送一个字节的数组，或者接收数据并把数据放进指定的字节数组中。如果用户想指定任何特殊的SocketFlags，也可以把它加到该方法调用中。表3.12显示了可供使用的SocketFlag值。

表3.12 SocketFlag值

值	说明
DontRoute	不用内部路由表发送数据
MaxIOVectorLength	给用于发送和接收数据的WSABUF结构数提供一个标准值
None	对这次调用不使用标志
OutOfBand	处理带外的（out-of-band）数据
Partial	部分地发出或接收消息
Peek	只对进入的消息取数

Receive()和Send()方法中可用的其他参数使用户可以指定发送或接收多少字节的数据，以及想在缓冲器中把数据放在什么地方。这些将在第5章中更详细地论证。

客户机函数

客户机设备也必须把一个地址绑定到创建的Socket对象，不过它不使用Connect()方法，而是使用Bind()方法。像Bind()函数一样，Connect()也要求一个该客户机需要连接的远程设备的IPEndPoint对象：

```
IPAddress host = IPAddress.Parse("192.168.1.1");
IPEndPoint hostep = new IPEndPoint(host, 8000);
Socket sock = new Socket(AddressFamily.InterNetwork,
    SocketType.Stream, ProtocolType.Tcp);
sock.Connect(hostep);
```

在连接建立起来之前，Connect()方法会阻止程序运行。如果连接无法建立，它就会产生一个异常（参阅本章后面“C#套接字异常”一节）。

一旦连接已经建立，客户机就可以像服务器使用的方法一样，使用Socket类的Send()和Receive()方法。当通信完成之后，必须把Socket实例关闭。也像UNIX套接字一样，Socket类使用shutdown()方法先得体地停止会话，然后用close()方法把会话真正关闭。shutdown()方法用一个参数来确定套接字将怎样关闭。Socket.Shutdown()方法可以使用的值在表3.13加以说明。

表3.13 Socket.Shutdown()值

值	说明
SocketShutdown.Both	在套接字上既防止发送也防止接收数据
SocketShutdown.Receive	防止在套接字上接收数据，如果收到额外的数据，将发出一个RST信号
SocketShutdown.Send	防止在套接字上发送数据，在所有存留在缓冲器中的数据发送出去之后，发出一个FIN信号

下面是得体地关闭一个连接的典型方法：

```
sock.Shutdown(SocketShutdown.Both);
sock.Close();
```

这个方法允许Socket对象适当地等待，直到数据从它的内部缓冲器发送完为止。

使用无连接的套接字

.NET使用与UNIX模式相同的连接套接字功能。当用户用SocketType.Dgram套接字类型创建套接字的时候，用UDP协议在网络上传输包。类似于UNIX模式，用户必须为服务器设置Bind()方法，在特定的端口建立套接字。服务器与客户机不需要使用Listen()或Connect()方法，这也与UNIX模式一样。

由于没有为通信而建立连接，所以标准的Receive()和Send()方法不起作用。用户必须用特殊的ReceiveFrom()和SendTo()方法。这些方法的格式由基本参数组成，这些参数与Re-

ceive()和Send()方法的基本参数相同（如表3.11所示）。此外，有一个额外的引用到EndPoint对象的参数，这个参数指定数据到哪里去（对于SendTo方法）或者数据从哪里来（ReceiveFrom方法）。例如，该方法最简单的格式如下：

```
ReceiveFrom(byte[], ref EndPoint)
SendTo(byte[], ref EndPoint)
```

对于UDP通信，EndPoint对象将指向一个IPEndPoint对象。如果读者是一位初学C#的新手，可能以前还没有见过ref关键字。ref关键字表示这个方法将用内存引用的办法访问EndPoint对象，而不是用它的值。这在C和C++编程技术中是一种普遍使用的方法，但是在C#程序中并不多见。

无阻塞编程技术

像UNIX网络编程模式一样，.NET Socket类的I/O方法在默认情况下使用阻塞的方法。当一个程序到达被阻碍的网络函数（比如Receive()函数）时，程序需要在那里等待，直到函数完成（比如当从套接字收到数据）后才能继续。有三个可以避免使用阻塞网络调用的C#技术：无阻塞套接字、多路传输套接字和异步套接字。

无阻塞套接字

正像在“使用C#套接字”一节中提到过的那样，C# Socket对象包含有多个可以查询它们值的属性。不过，其中有一个Blocking属性也可以被设置。用户可以把一个套接字的Blocking属性设置成假（false），使得套接字变成无阻塞模式。

当套接字是无阻塞模式的时候，它就会不等待一个I/O方法的完成。它将会检查这个方法，如果它不能完成，这个方法会失败，而程序会继续进行。例如，如果Blocking设置为假（false），Receive()方法将不等待数据出现在套接字上。而是该方法会返回一个0值，表示在套接字上没有可用的数据。

多路传输套接字

正像UNIX中一样，Socket类提供了Select()方法。这个方法用于多路传输多个套接字实例，以便找到一个已经准备读或写的套接字。不过在C#中，Select()的使用有些不同。下面是Select()方法的格式：

```
Select(IList read, IList write, IList error,
      int microseconds)
```

read、write和error参数都是IList对象，这些对象都是包含要监视的套接字的数组。microseconds参数指定Select()方法等待事件发生时要等待多长时间（用微秒）。

下面是个小的代码段，演示了Select()方法的用法：

```
ArrayList socketList = new ArrayList(5);
SocketList.Add(sock1);
SocketList.Add(sock2);
```

```
Socket.Select(socketList, null, null, 1000);

byte[] buffer = new byte[1024];
for (i = 0; i < socketList.Length - 1; i++)
{
    socketList[i].Receive(buffer);
    Console.WriteLine(Encoding.ASCII.GetString(buffer));
}
```

注意，**Select()**方法将监视sock1和sock2两个套接字上接入的数据。如果在两个套接字上都没有数据出现，**Receive()**方法不会阻碍程序进行。

异步套接字编程技术

.NET框架使用由Windows Winsock API推出的异步套接字模式，这一点也不使人感到奇怪。这种方法在套接字准备接收或发送数据的时候，允许用户使用单独的方法。用户不是用**Receive()**方法等待来自客户机的数据，而是使用**BeginReceive()**方法，这种方法会注册一个当套接字上的数据可用时可以被调用的代表（delegate）。在代表的方法中，用户必须用**EndReceive()**方法阻止从套接字异步读和检索数据。这个概念在第8章“异步套接字编程”中再详细介绍。

C#套接字异常

在.NET框架中有一个不论是UNIX还是Winsock API都没有用过的套接字编程功能，就是套接字异常（socket exceptions）。到目前为止，本章展示的所有范例和代码段都有个重要的假设：所有的套接字调用都会成功。在当今的网络世界中这是个危险的假设。

在第1章“C#语言”中可以看出，C#用尝试—捕获（try-catch）的机制在程序运行时捕获错误和异常。用户应当周密地考虑到这些异常，并小心地决定，当异常出现时应当采取什么行动。

所有Socket类的方法都使用SocketException异常。任何用户编制的套接字程序都应当总是检查SocketException异常，然后再试图从错误中恢复，或至少警告用户出现了什么问题。

程序清单3.4是一个套接字客户机程序的范例，它用异常编程的方法确定在正常的网络会话中是否出现了异常。在单个网络调用的周围使用try-catch块，用户可以找出自己程序中哪里出现了问题，并给自己的客户更好的信息，使他们能采取相应的措施来纠正问题。

程序清单3.4 SocketExcept.cs程序

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class SocketExcept
{
```

```
public static void Main ()
{
    IPAddress host = IPAddress.Parse("192.168.1.1");
    IPEndPoint hostep = new IPEndPoint(host, 8000);
    Socket sock = new Socket(AddressFamily.InterNetwork,
        SocketType.Stream, ProtocolType.Tcp);

    try
    {
        sock.Connect(hostep);
    } catch (SocketException e)
    {
        Console.WriteLine("Problem connecting to host");
        Console.WriteLine(e.ToString());
        sock.Close();
        return;
    }
    try
    {
        sock.Send(Encoding.ASCII.GetBytes("testing"));
    } catch (SocketException e)
    {
        Console.WriteLine("Problem sending data");
        Console.WriteLine(e.ToString());
        sock.Close();
        return;
    }
    sock.Close();
}
```

C#套接字助手类

.NET框架支持用于高级网络编程的标准套接字接口，可是，为了简易的网络编程，它也提供了一种简单的接口。简化套接字助手类帮助网络程序员用较简单的语句和较少的编码（两个对任何程序员都重要的优点）创建套接字程序。这一节介绍C#套接字助手类。读者在第7章“C#套接字的助手类”中将会看到更详细的范例，告诉读者在实际程序中怎样使用这些类。

下面是套接字编程使用的三个助手类：

- TcpClient
- TcpListener
- UdpClient

每一个助手类都专门支持一个特殊的套接字编程函数，并简化这个函数的程序需要的接口。显然，**TcpClient**和**TcpListener**类用于创建TCP客户机和服务器程序；**UdpClient**类用于创建UDP程序。

TcpClient类

TcpClient类的各种方法，用于按照面向连接的网络模式创建客户机网络程序。**TcpClient**方法反映了常规的套接字编程技术，但为了简化编程工作，许多步骤被压缩了。

首先，创建**TcpClient**对象并把它连接到远程主机，可以有三种方法，用每种方法创建套接字都比手工**Socket**类的方法更加容易。

默认的构造函数 默认的构造函数格式可以在任何一个可用的本地端口创建套接字。然后再用**Connect()**方法连接到远程主机：

```
TcpClient newclient = new TcpClient();  
newclient.Connect("www.isp.net", 8000);
```

第一个语句建立了一个新的**TcpClient**对象，并把该对象绑定到一个本地地址和端口。第二个语句把套接字连接到远程主机地址和端口号。注意，远程主机的地址可以用主机名指定。**TcpClient**类会自动地把主机名正确地解析成IP地址。其实它已经替用户完成了许多工作！

指定的EndPoint对象 第二种构造函数的格式更进了一步，它允许用户指定一个在建立套接字时使用的具体本地**EndPoint**对象：

```
IPAddress ia = Dns.GetHostByName(  
    Dns.GetHostName()).AddressList[0];  
IPEndPoint iep = new IPEndPoint(ia, 10232);  
TcpClient newclient2 = new TcpClient(iep);  
newclient2.Connect("www.isp.net", 8000);
```

指定的远程主机 第三种构造函数格式是最常用的。它允许用户指定在构造函数中要连接的远程主机的地址和端口，不需要使用**Connect()**方法：

```
TcpClient newclient3 = new TcpClient("www.isp.net", 8000);
```

只用一个简单的步骤，用户就创建了一个新的**TcpClient**对象到一个随机本地端口号，并把它连接到指定的远程主机。还有，如果用户用主机名代表主机地址，**TcpClient**对象会自动地解析它。建立一个套接字并连接到远程主机只用一个语句——多么有效的系统！

一旦创建了**TcpClient**实例，用户也许就想用它来发送和接收数据。**GetStream()**方法用于建立一个**NetworkStream**对象，允许用户在套接字上发送和接收字节。一旦用户有了一个套接字的**NetworkStream**实例，它就是一种快速方法（snap），可以用标准流的**Read()**和**Write()**方法，把数据移进或移出该套接字。下面的代码段演示了把**NetworkStream**对象分配给**TcpClient**实例，并把数据写到套接字或从套接字读出的过程：

```
TcpClient newclient = new TcpClient("www.isp.net", 8000);  
NetworkStream ns = newclient.GetStream();
```

```
byte[] outbytes = Encoding.ASCII.GetBytes("Testing");
ns.Write(outbytes, 0, outbytes.Length);
byte[] inbytes = new byte[1024];
ns.Read(inbytes, 0, inbytes.Length);
string instring = Encoding.ASCII.GetString(inbytes);
Console.WriteLine(instring);
ns.Close();
newclient.Close();'
```

说明：像在UNIX套接字领域中一样，用户必须记住，当工作完成之后，一定要用Close()方法把TcpClient对象正确地关闭。要注意，关闭NetworkStream对象但仍然保持TcpClient对象打开状态，这是许多初学网络编程的新手容易忽略的一个小小细节。

TcpListener类

正像TcpClient类简化了客户机套接字程序一样，TcpListener类简化了服务器程序；它们的类构造函数也是类似的。下面是三个构造函数的格式：

- TcpListener(int port)绑定到一个指定的本地端口号
- TcpListener(IPEndPoint ie)绑定到一个指定的本地EndPoint
- TcpListener(IPAddress addr, int port)绑定到一个指定的本地IPAddress和端口号。

这三个格式允许用户控制怎样为TcpListener对象建立下层的套接字。一旦这个对象建立起来，用户就可以用Start()方法开始收听新的连接尝试。

在Start()方法之后，用户必须用AcceptSocket()或者AcceptTcpClient()方法接受接入的连接尝试。正如从它们的名字上可以猜得到的那样，这两个方法在接受了接入的连接之后，或者返回一个Socket对象或者返回一个TcpClient对象。用户已经取得了TcpListener类的优点，所以可能想用AcceptTcpClient()方法为新的连接创建一个新的TcpClient对象。

一旦为新的连接创建了TcpClient对象，就可以用标准的TcpClient方法与客户机开始通信。一个样例代码段如下所示：

```
TcpListener newserver = new TcpListener(9050);
newserver.Start();
TcpClient newclient = newserver.AcceptTcpClient();
NetworkStream ns = newclient.GetStream();
byte[] outbytes = Encoding.ASCII.GetBytes("Testing");
ns.Write(outbytes, 0, outbytes.Length);
byte[] inbytes = new byte[1024];
ns.Read(inbytes, 0, inbytes.Length);
string instring = Encoding.ASCII.GetString(inbytes);
Console.WriteLine(instring);
ns.Close();
newclient.Close();
newserver.Stop();
```

这个程序会停在AcceptTcpClient()方法的地方，等待新的客户机连接。一旦连接成功，就用GetStream()方法创建一个NetworkStream对象，用于把数据读出和写数据到套接字。

记住，当工作结束时一定要关闭NetworkStream和TcpClient对象。必须用Stop()方法关闭TcpListener对象。

UdpClient类

对于需要连接套接字的应用程序，UdpClient类提供了一个简单的到UDP的接口。用户也许感到奇怪，为什么UDP的助手类没有一个收听者的版本呢。答案很简单：用户不需要。要记住，UDP是一个无连接的协议，所以没有什么客户机或服务器的区别；只有等待数据或发送数据的UDP套接字。用户不需要把UDP套接字绑定到特定的地址和等待接入的数据。

UdpClient构造函数采用与TcpClient一样的格式，允许用户指定创建UDP套接字或引用一个远程地址和UDP端口所需要的信息。

Receive()方法允许用户在UDP端口接收数据。由于没有建立连接的会话，所以当在一个UDP端口接收数据的时候，用户不需要知道数据是从哪里来的（除非在UdpClient构造函数中指定）。为了弥补这一点，Receive()方法包括一个IPEndPoint参数，该参数提供了来自远程主机的信息。做法如下：

```
UdpClient newconn = new UdpClient(8000);
IPEndPoint remoteclient = new IPEndPoint(IPAddress.Any, 0);
byte[] recv = newconn.Receive(ref remoteclient);
string data = Encoding.ASCII.GetString(recv);
Console.WriteLine("From: {0}", remoteclient.ToString());
Console.WriteLine("  Data: \"{0}\"", data);
newconn.Close();
```

UdpClient对象是使用端口号的实例，用户想让应用程序在该端口接收UDP包。注意，remoteclient对象是一个使用IPAddress.Any值的实例，这个值作为一个伪值使用，它将在Receive()方法中被来自远程客户机的信息取代。

Send()方法有三种格式。如果UdpClient对象引用了一个远程主机地址和端口，Send()方法就不需要指定数据的地址，如果UdpClient对象没有引用远程主机，Send()方法就必须引用，可以用IPEndPoint对象或者用一个字符串主机名或者地址值和一个整数端口号来完成。发送数据的代码如下所示：

```
UdpClient newclient = new UdpClient(8001);
IPEndPoint remotehost = new
    IPEndPoint(IPAddress.Parse("192.168.1.6"), 8001);
byte[] bytes = Encoding.ASCII.GetBytes("test string");
newclient.Send(bytes, bytes.Length, remotehost);
newclient.Close();
```

UdpClient构造函数上使用的端口号不一定要与远程主机的端口号相同，但是保持它们一致有助于跟踪被特殊应用程序使用的端口。远程主机的地址信息存储在一个IPEndPoint对象中，作为在Send()方法中的参数使用。

小结

本章用大量篇幅讨论了网络编程的历史和其与C#网络编程的关系。在UNIX系统发展的早期，伯克利套接字库对网络编程技术起到了真正的推动作用。BSD UNIX第4.2版操作系统包括一个公共的套接字接口，帮助程序员通过网络与其他系统通信。

Windows套接字接口Winsock是在现有的UNIX网络套接字编程模式的基础之上，加上许多Winsock特有的功能，增强了标准套接字编程模式的功能。许多增强的功能都是围绕着增加异步套接字I/O功能为中心的。WSAAsyncSelect()和WSAEventSelect()函数使Winsock程序能够用标准的Windows消息和事件把信号传给套接字事件。

.NET框架提供了一个访问Winsock API的套接字接口。System.Net.Sockets命名空间提供了Socket类，用控制的代码给C#程序提供Winsock功能。除了普通的Socket类之外，C#还提供TcpClient、TcpListener和UdpClient类。这些类完成了许多创建套接字所需要的编程工作，它们给初学网络编程的新手提供了一种容易操作的简易接口。

正如读者所看到的，网络地址对于套接字编程是至关重要的。很可惜，当前有很多在因特网（甚至在一个本地网络）上引用地址的方法。下一章将介绍怎样在C#中用域名服务（Domain Name Service, DNS）类把主机名解析为用户网络程序中的IP地址。

第4章 DNS和C#

- 域名系统 (DNS)
- Windows DNS客户机信息
- DNS配置
- 用C#检查DNS配置
- 用nslookup解析主机名
- C#中的DNS类
- 同步方法
- 异步方法

IP地址和网络编程相互联系、密不可分。当写套接字程序的时候，用户必须或者指定服务器应用程序本地主机的IP地址或者指定客户机应用程序远程主机的IP地址。有时候，这并不是一件容易的事情，相同的IP地址和主机名会使事情变得复杂。如果任何时候都只使用IP地址的话，网络编程会变得简单得多。

不过，在真实世界里到处都可以看到主机名，在书上、杂志上、Web站点上、电视广告上等等地方。因特网已经使主机名成为一个家喻户晓的普通术语。因此，几乎所有的网络程序都必须允许用户输入主机名和IP地址，这使得网络程序员可以保证程序能够找到与主机名正确对应的IP地址。实现这个的主要控制机制就是域名系统 (DNS)。域名系统用于控制主机名，并给想找另一个主机名的IP地址的主机提供信息。

本章讨论DNS基础并告诉读者怎样用DNS解决用户C#网络程序中的主机名问题。首先，概括地叙述一下DNS的处理过程，然后介绍在标准Windows工作站和服务器的使用。最后，本章将深入探讨在用户的C#网络程序中如何使用DNS，并伴有范例程序向读者展示DNS类所起的作用。

计算机名的历史

回想过去 (20世纪80年代) 当因特网还很小的时候，想在网上找到另外一台计算机并不很困难。因特网上每个系统都有一个数据库，把主机名与分配的IP地址相互匹配。系统管理员可以用任何想用的名称作为主机名，例如Fred、Barney、acctg1等。信息交换中心对所有主机名和因特网上的IP地址保持跟踪，并在网上发表包含所有地址信息的主控数据库。通常，大约一个星期左右，系统管理员就要下载含有他们本地系统的主机名和地址的主控数据库。

一开始这个系统还可以正常工作，可是没有多久它就不能适应需要了。随着越来越多的主机增加到因特网中，这个数据库也就越来越大。当然随着数据库的变大，下载需要的时间也越来越长，而且在每个系统上占用越来越多的空间。很显然，这个系统急需改进，这个改进就是域名系统 (Domain Name System, DNS)。

域名系统 (DNS)

为了简化计算机命名方法难以处理的状态，产生了域名系统。它允许把主控主机数据库分开并分布到因特网上的多个系统中。DNS使用分层数据库的方法建立多个信息层，这些信息可以分开并存储在遍布因特网的不同系统中。

除了分开数据库外，DNS还给客户机提供了一种实时查询数据库的方法。所有客户机需要知道的就是在数据库层中的DNS服务器位置（也许一两个备份服务器）。如果客户机查询的主机名没有存储在DNS服务器的本地数据库中，服务器可以查询另外一个包括此信息的DNS服务器，并把信息转送给客户机。

为了实现DNS的基本原理，建立了新的数据库和协议在客户机和服务器中传送DNS信息，并使得DNS服务器相互更新。这一节将叙述DNS系统的工作方式，并告诉读者客户机是怎样通过查询DNS服务器获得主机名信息的。

DNS的结构

分层数据库的结构类似于一个按树的方式用节点连接起来的结构图（这就是分层的部分）。最上面的节点叫做根（root）。在因特网主机地址中，并没有明确地表示出根节点来，所以它通常叫做“无名”节点。在根下面建立了多个种类，把数据库分成多个叫做域（domain）的块。每个域都包含多个域名服务器，这些域名服务器负责维护数据库所在区域的计算机名数据库（这就是分布的部分）。图4.1表示了DNS域是怎样分布的。

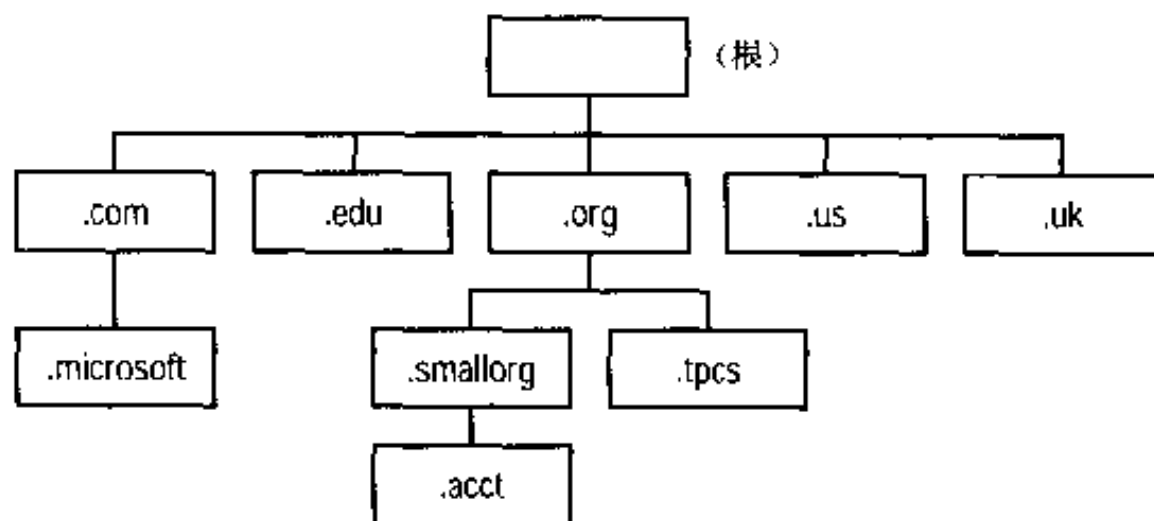


图4.1 因特网的DNS系统

分布结构的第一层（或顶层）根据国家代码分成多个域。其他的顶级域用于特殊机构，建立这些域是为了防止国家域太拥挤。表4.1描述了最初顶级DNS域的方案。

最近，又在DNS系统上增加了一些新的顶级域。从这些顶级域可以识别出特殊的工业领域，不仅仅只是在某个国家内。这些工业领域的公司可以在他们特殊的工业中注册主机名。表4.2显示了7个新的顶级域。

随着因特网的发展，每个顶级域又分成一些子域，或者区（zone）。每个区本身都是一个独立的域，但又依靠它的父域与数据库连接。父区必须允许子区的存在，并对子区的行为负责（正像现实生活一样）。每个区必须至少有两个维持该区DNS数据库的DNS服务器。

表4.1 DNS最初的顶级域名

域名	说明
.com	商业机构
.edu	教育公共机构
.mil	美国的军事站点
.gov	美国政府机构
.net	因特网服务提供商（ISP）
.org	非盈利机构
.us	其他美国机构（例如地方政府机构）
.ca	加拿大机构
.de	德国机构
(other countries)	其他国家的机构

表4.2 2001年增加的DNS顶级域名

域名	说明
.aero	航空运输业中的公司
.biz	普通商业
.coop	合作团体
.info	无限制使用
.museum	博物馆
.name	个人
.pro	专门职业（医生、律师等）

最初的DNS规范规定，为了容错的目的，一个区的DNS服务器必须有单独到因特网的连接，并且必须放在单独的地方。由于这个规定，所以许多机构依靠其他机构来宿主他们的第二和第三DNS服务器。

一个区中的主机把域名加到他们的主机名上就形成了一个惟一的Internet名。这样，在smallorg.org域中的一台名为fred的计算机就叫做fred.smallorg.org。这个约定可能引起一点混乱，因为一个域可能包含主机和区。

例如，smallorg.org域可能包含主机fred.smallorg.org，也可能批准一个区acctg.smallorg.org为一个子域，这个区又可以有另外一个主机barney.acctg.smallorg.org。虽然这种方法简化了数据库系统，但它使得在因特网上找到主机更复杂了。图4.2向读者展示了一个域和一个与它相关的子域。

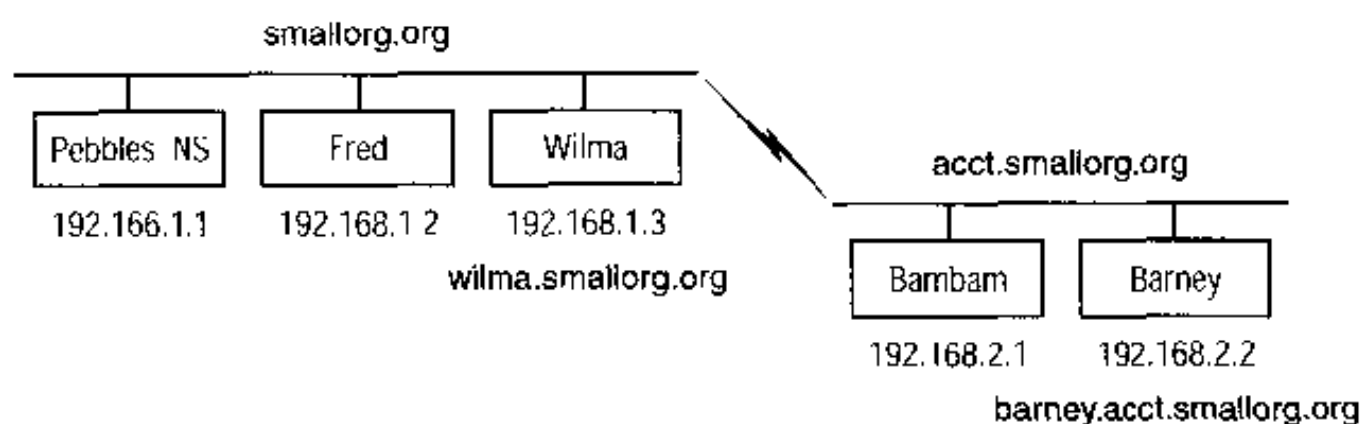


图4.2 因特网上域和子域的例子

域名的控制

因特网域名已经成为一个热门的话题。过去，一个实体（InterNIC公司）控制着美国所有在.com、.net和.org域中的域名。现在一个非赢利机构，因特网名称和地址号码分配公司（Internet Corporation for Assigned Names and Numbers (ICANN)）控制着美国的域名分配。ICANN负责管理所有美国的域名。人们可以从多个销售商那里购买域名，不止一个公司。不过所有在美国范围内使用的域名都必须由ICANN清空。

查找DNS中的主机名

前面已经提到过，DNS使得客户机通过查询本地DNS服务器来获得主机名的信息。寻找主机名的过程可能会有三种情况：

- 寻找一个在本地域内的主机
- 寻找一个远程主机，该主机名不在本地DNS服务器上
- 寻找一个远程主机，该主机名在本地DNS服务器的高速缓存中

本地域主机名

找一个主机名最简单的模式就是一个客户机试图寻找在它本地域中另外一个系统的主机名。在这种情况下，客户机用本地DNS服务器作为它默认的服务器，发出查找本地主机名的DNS查询。DNS服务器在它的数据库中找到了该主机名，并把结果返回客户机。这个过程如图4.3所示。

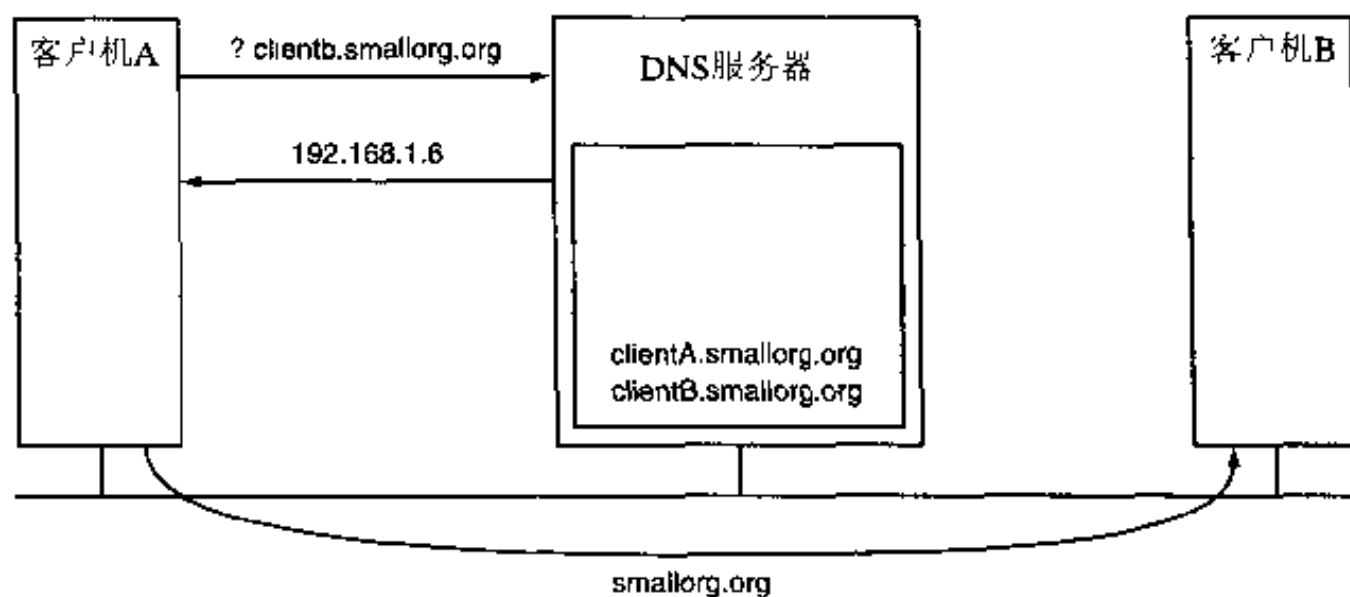


图4.3 客户机A解析DNS服务器上的一个本地主机名

远程域主机名

当客户机想找-一个在远程域中系统的主机名时，可能会有两种可能的情况：

- 在本地DNS服务器上没有发现远程主机名
- 在本地DNS服务器的高速缓存上发现了远程主机名

在本地DNS服务器上没有发现远程主机

当客户机向本地DNS服务器发出-一个查询，要寻找一个本地DNS服务器数据库中没有包含的系统的主机名时，会发生以下系列动作：

1. 本地DNS服务器询问根DNS服务器是否有该主机名。
2. 根DNS服务器决定应当到哪个域去找该主机名，并把请求送给负责该主机域的DNS服务器。
3. 负责该主机域的本地DNS服务器把主机名解析成IP地址，并把结果返回到根DNS服务器。
4. 根DNS服务器把结果送回到客户机的本地DNS服务器。
5. 客户机的本地DNS服务器把得到的IP地址返回给客户机。

在这个过程的后面进行了许多的工作，可最终的结果是客户机收到了远程系统正确的IP地址，所有客户机需要做的事就是向本地DNS服务器发出-一个查询。

在本地DNS服务器的高速缓存上发现了远程主机

DNS有一个优点就是，当本地DNS服务器向根DNS服务器发出查询远程主机名的请求时，它可以把结果存储在本地高速缓存里。当本地客户机向本地DNS服务器查询远程主机名的时候，该服务器已经有了关于该主机名的信息，本地DNS服务器可以直接返回高速缓存的信息，不需要再向根DNS服务器发出查询请求。这极大地改善了客户机查询的响应时间。

对于这个过程有一句警告。当根DNS服务器返回主机的IP地址信息时，还包括一个存留时间（time to live, TTL）值（主机名信息在本地DNS服务器的高速缓存中应当保存多长的时间才被放掉）。负责该远程主机的本地DNS服务器设置这个值。根据本地网络主机名的变更率，TTL值可以设置为从几分钟到几个星期的任意数值。一旦TTL值已经期满，当本地DNS服务器收到新的客户机查询时，就必须向根DNS服务器查询。

DNS数据库

每个DNS服务器都负责跟踪它那个区里的主机名。为了完成跟踪任务，DNS服务器必须用一种方法把主机信息存储在供远程计算机查询的数据库中。DNS数据库是一个文本文件，它由区域中主机和网络函数的资源记录（resource records, RR）组成。DNS服务器必须运行DNS服务器软件包才能把DNS信息从本地域数据库传送到远程DNS服务器。

DNS服务器的数据库必须定义各种类型的网络资源，例如：

- 本地域的定义
- 域中每一台注册的主机
- 域中主机的通用别名

- 特殊服务，例如DNS服务器和邮件服务器

创建的RR格式用于跟踪DNS服务器需要的所有信息。表4.3描述了一个DNS数据库中可能包含的一些基本资源记录。

说明：由于研究人员总是想把更多的信息加到DNS数据库中，并且要控制库中信息的安全，因此DNS数据库的设计方案成了一个很关键的问题。新的记录类型在不断地加到DNS数据库中，表4.3列出的是在DNS数据库中建立一个区需要的核心记录。

表4.3 DNS数据库核心记录

记录类型	说明
SOA	开始授权
A	因特网地址
NS	名字服务器
CNAME	规范名字（别名）
HINFO	主机信息
MX	邮件服务器
PTR	指针（IP地址）

在数据库的顶部列有一条关于这个域的SOA记录。这个域所有其他的资源记录可以按任何顺序加在SOA记录的后面。每个域的DNS服务器都包含该域中所有注册主机的资源记录。下面几节将较详细地叙述各个DNS记录的类型。

授权开始记录（SOA）

每个数据库都用一个SOA记录开始，这个记录定义了数据库所驻留的区域。SOA记录的格式如下：

```
domain name      [TTL] [class] SOA origin person (
                    serial number
                    refresh
                    retry
                    expire
                    minimum)
```

- domain name** 指定的区名（通常用@符号作为占位符来表示计算机的默认域）。
- TTL** 时间长度（秒）。要求计算机在它的本地名字高速缓存中把所有这个区的DNS信息保留这个时间长度。指定这个值是可选的。
- class** 使用协议的种类（对于因特网上的主机，它总是IN类）。
- origin** 主控区数据库所在的计算机名。在主机名的后面应当使用一个句号；不然本地域名就会附在主机名的后面（当然，如果用户想利用这个特点的话，也可以省去句

号)。

person 这个区负责人的E-mail地址。这个值的格式与读者经常看到的有一点区别。@符号已经用于表示默认的域名，所以就不能再用在电子邮件地址中了，我们是用一个句点来代替。例如，可以用sysadm.smallorg.org来代替sysadm@smallorg.org。如果在名字部分有任何句点的话，必须用一个反斜线把它们忽略。这种情况的一个例子是，地址john.jones@smallorg.org，可转换成john\jones.smallorg.org。

serial number 用于标识区数据库文件版本的惟一号码。通常，这里使用的是日期加版本数（例如200210151）。

refresh 时间长度（秒）。指的是第二DNS服务器应当向主DNS服务器查询，以核对SOA序列号的时间长度（秒），如果第二服务器的SOA序列号与主服务器的不同，第二服务器就会请求更新它的数据库。这个值通常规定为一个小时（3600秒）。

retry 时间（秒）。指的是当第二DNS服务器的数据库刷新尝试失败之后应当过多长时间再重试的时间。

expire 一段时间周期（秒），在这段时间周期里，第二DNS服务器可以使用从没有重新刷新过的主DNS服务器中检索到的数据。这个值通常取得比较充裕，比如3600000（大约42天）。

minimum 时间长度（秒）。指的是作为这个区中所有资源记录（RR）留存时间的时间长度。通常86400（1天）就是比较合适的值。

因特网地址记录（A）

区中数据库定义的每一台主机都应当有一个合法的A记录，把它的主机名定义到因特网。A记录的格式如下：

```
host      [TTL]      [class]      A      address
```

host 计算机完全合格的主机名（包括域名）。

address 计算机的IP地址。

TTL和class 这些参数是可选的，其意义与SOA记录一样。

规范名字记录（CNAME）

除了普通的主机名之外，许多计算机还有自己的别名，这不需要给域中的计算机重新命名，对于识别特殊的服务是很有用的。例如，用户可能给主机fred.acctgl.smallorg.org分配了一个别名叫做www.smallorg.org，CNAME记录会把别名与真正的主机名链接在一起。CNAME记录的格式如下：

```
nickname  [TTL]      [class]      CNAME      hostname
```

nickname（别名）和**hostname**（主机名）两个参数的作用非常明显。它们分别代表计算机分配的别名和原来的主机名。

同样，这里的TTL和class参数是可选项，其意义与SOA记录中一样。

名字服务器记录 (NS)

每个区至少应该有两个DNS服务器。NS记录用于把这些服务器与试图解析区内主机名的其他DNS服务器识别出来。NS记录的格式如下：

```
domain [TTL] [class] NS server
```

domain DNS服务器负责的那个区的域名。如果它空白，NS记录指的就是在SOA记录中定义的区。

server DNS服务器的主机名。在数据库中也应当有一条相关的记录，以便识别该主机的IP地址。

TTL和class 同样，这两个参数是可选项，其意义与SOA记录中一样。

主机信息记录 (HINFO)

DNS服务器还可以用HINFO记录来获得其他关于计算机的有用信息。HINFO记录的格式如下：

```
host [TTL] [class] HINFO hardware software
```

host 应用信息的计算机的主机名。

hardware 计算机使用的硬件类型。

software 计算机操作系统的类型和版本。

TTL和class 同样，这两个参数是可选项，其意义与SOA记录中一样。

指针记录 (PTR)

区中的每台计算机除了有一个A记录之外，还应当有一个PTR记录。PTR记录允许DNS服务器从计算机的IP地址执行相反的查询。没有PTR信息，远程服务器就不能确定IP地址所在地的域名。PTR记录的格式如下：

```
IN-ADDR name [TTL] [class] PTR name
```

IN-ADDR name IP地址反转的DNS名。如果听起来觉得搞糊涂了的话，确实如此。这个名字允许DNS服务器根据计算机的IP地址反方向地进行操作。IN-ADDR.ARPA地址是一个特殊的域，支持网关的位置和因特网地址到主机的映射。它不需要反向查询，因为IP地址被映射到一个假想的主机名。这样，IP地址为192.168.0.1的计算机的IN-ADDR name就应当是1.0.168.192.IN-ADDR.ARPA。

name 在A记录中出现的计算机的主机名。

TTL和class 同样，这两个参数是可选项，其意义与SOA记录中一样。

邮件交换记录 (MX)

MX记录用于表示域中特殊类型的主机。它指示远程邮件服务器在哪里转发该域的邮件。MX记录的格式如下：

```
name [TTL] [class] MX preference host
```

name 域名（或者当name空白时的SOA域）。如果用户想为网络中特殊的主机重定向邮件的话，也可以是主机名。

preference 整数，表示当指定多个邮件服务器的时候，远程服务器试图连接时应当使用的顺序，最高的优先权是0，数字的增加表示优先权的降低。

preference 这个特点用于建立一个域的主要和第二邮件服务器。当远程邮件服务器查询DNS服务器，想找到负责那个域的邮件服务器时，DNS服务器会发送全部的服务器和优先权的列表。远程邮件服务器应当试图与列表中最高优先权的邮件服务器连接，如果失败，再按列表中的优先权顺序与下一个邮件服务器连接。

host 邮件服务器的IP地址或主机名。该记录也应当有一个相关的A记录，用来识别该邮件服务器的IP地址。

TTL和class 同样，这两个参数是可选项，其意义与SOA记录中一样。

警告：使用CNAME记录作为MX主机的时候一定要谨慎。有些E-mail服务器包用CNAME（别名）主机不能正常地操作。

一个DNS数据库的例子

当一个ISP为一家公司提供域名服务的时候，ISP会在DNS数据库中放有一些记录，在因特网上标识出这个域。SOA记录可以识别出域名，但它把该ISP的主机指定为授权主机。域名的NS记录将指向该ISP的DNS服务器。如果该公司有它自己的邮件服务器，MX记录将指向这个邮件服务器。如果ISP为公司操作邮件服务器，则MX记录指向ISP的邮件服务器。

在因特网的其他部分看来，这些计算机都是公司域的一个部分——即使这些计算机并没有真正在公司的网络上，而是在ISP那里。程序清单4.1表示在ISP的DNS数据库中定义域的ISP记录的一个实例。

程序清单4.1 DNS数据库条目

```
smallorg.org IN SOA master.isp.net. postmaster.master.isp.net
    postmaster.master.isp.net (
                                1999080501 ;unique serial number
                                8H          ; refresh rate
                                2H          ;retry period
                                1W          ; expiration period
                                1D)         ; minimum

    NS ns1.isp.net. ;defines primary nameserver
    NS ns2.isp.net. ;defines secondary nameserver

    MX 10 mail1.isp.net. ; defines primary mail server
    MX 20 mail2.isp.net. ; defines secondary mail server

    www CNAME host1.isp.net ;defines a www server at the ISP
    ftp CNAME host1.isp.net ; defines an FTP server at the ISP

    host1.isp.net A 10.0.0.1

    1.0.0.10.IN-ADDR.ARPA PTR host1.isp.net ; pointer for reverse DNS
```


程序清单4.1的第一段是这个新域的SOA记录。ISP将域名smallorg.org指向它的服务器master.isp.net。接着，用NS记录类型指定主DNS服务器和第二DNS服务器。在NS记录的下面，用MX记录指定主邮件服务器（mail1.isp.net）和第二邮件服务器（mail2.isp.net）。由于mail1.isp.net服务器的优先权号比较低，所以认为它是主邮件服务器。如果该服务器情况正常的话，到smallorg.org域的任何邮件都应当首先递送到这个服务器。

在MX记录的后面，CNAME记录定义主机名www.smallorg.org为一个指向ISP服务器的别名，该服务器宿主有公司的Web页面。地址ftp.smallorg.org也定义为指向同一个ISP服务器的别名，它也宿主FTP站点。当客户花不起钱拥有到因特网的专门连接，可是又想为他们的用户提供Web和FTP服务的情况下，为Web和FTP服务器使用别名地址是大多数ISP为客户们提供的一项服务。

A和PTR记录的设置为ISP主机提供了因特网主机名和IP地址信息，使得远程客户机可以连接到这个服务器上。

说明：PTR记录通常放在服务器上的一个单独的数据库文件中，以便简化该数据库。对于程序清单4.1的例子这不成什么问题，因为它只有一条PTR记录。可是当有几十条或数百条记录的时候就有这个问题了。

Windows DNS的客户机信息

所有的Microsoft Windows平台都以一种或另一种方式支持DNS协议。每个工作站平台都允许用户配置用于解析应用程序中主机名的远程DNS服务器。此外，在Windows 2000服务器平台上，用户还可以创建一个DNS服务器数据库，以支持一个域的域名服务器（DNS）。

为了使C# DNS程序有效工作，DNS必须在Windows平台上正确地设置。这一节将介绍怎样配置Windows系统上的DNS客户机组件，以及怎样从C#程序获得系统的DNS客户机信息。

DNS配置

所有的Windows系统都有解析主机名的能力。有两种把主机名解析为IP地址的方法：

- 访问主机名的本地文件
- 查询远程DNS服务器

两种方法解析主机名的途径不同，下面几节将介绍它们的区别。

使用主机文件

解析有限数量的主机名最简单的方法就是在系统中维持一个本地主机名文件。对于用户经常访问的远程主机名，这种方法对于Windows是最有效的，因为它不需要到外面去访问远程DNS服务器，允许更直接的访问。不过，当用于解析大量主机名的时候，例如在因特网上冲浪的情况，显然，这种方法不可取。

主机名文件的文件名为hosts，它的位置根据用户使用的Windows版本不同而不同；

- 对于Windows 95、98、Me和XP，hosts文件在C:\WINDOWS\SYSTEM32\DRIVERS\ETC目录中。
- 对于Windows NT和2000，hosts文件是在C:\WINNT\SYSTEM32\DRIVERS\ETC目录中。

hosts文件采用文本格式，它的每一行代表每个主机的一条记录。列在最前面的是主机的IP地址，后面跟着一个或更多的空格，然后是主机名；例如：

```
127.0.0.1      localhost
192.168.1.1    shadrach.blum.lan
192.168.1.2    meshach.blum.lan
192.168.1.6    abednego.blum.lan
```

注意hosts文件中的第一条。127.0.0.1地址代表特殊的回送地址，总是指向localhost主机名。localhost主机名本身就是一个特殊的主机名，因为它代表系统的内部网络地址。应用程序通常用这个地址把IP包引导到同一个系统——这是在同一个系统上运行的程序之间进行通信的方法。

在localhost定义的后面，用户经常访问的每一个远程主机都应当列在自己的行。比方说读者运行一个本地内部用的网络，使用一个假想的域名（像前面一个例子）；如果你把它们输入到每个系统的hosts文件中，那你仍然可以在网络程序中使用该主机名。

很容易看出，这种方法对于有限数量的主机将是有效的，但如果用户想要访问许多主机的话，hosts文件就会变得相当庞大难以处理。显而易见，应当用DNS系统来解决主机名的问题。下一节将说明怎样在Windows平台上解决主机名的问题。

使用远程DNS服务器

当用户需要解析标准的因特网主机名时，最简单的办法就是找一个可以查询的DNS服务器。大多数（如果不是全部的话）ISP都提供一个或多个DNS服务器地址供客户DNS查询用。为了把DNS服务器配置到用户自己的系统，通常根本不需要用户做任何事情。因为如果ISP为用户的系统动态地分配IP地址的话，很可能也动态地分配了DNS服务器。

另一方面，如果没有动态地分配DNS服务器的话，用户可能就需要自己配置。DNS服务器的条目存储在系统网络属性的Internet Protocol Properties（因特网协议属性）部分中。根据用户工作的不同Windows平台，获得这些属性的方法也稍有区别。在Windows 2000或者XP工作站，单击Control Panel（控制面板），再双击Network Connections（网络连接）项目。会出现一个所有可用网络接口的列表。右键单击你想要配置DNS服务器的网络接口，再选择Properties（属性）选项。

图4.4显示了一个网络接口的Properties（属性）窗口。窗口显示的是为这个接口加载了哪些协议和服务。DNS配置设置存储在Internet Protocol（TCP/IP）项目中。

当用户双击，显示该网络接口的TCP/IP设置的时候，会看到如图4.5所示的Properties（属性）窗口（对于Windows XP Professional工作站）。用户可以在此窗口设置正确的IP地址、子网掩码和默认的网关设置，还有两个默认的DNS服务器。列在第一个的服务器将是主服务器。如果Windows没有从该主服务器接收到响应，它会试用备用的第二服务器。



图4.4 网络接口属性，通过Control Panel▶ Network Connections得到该窗口

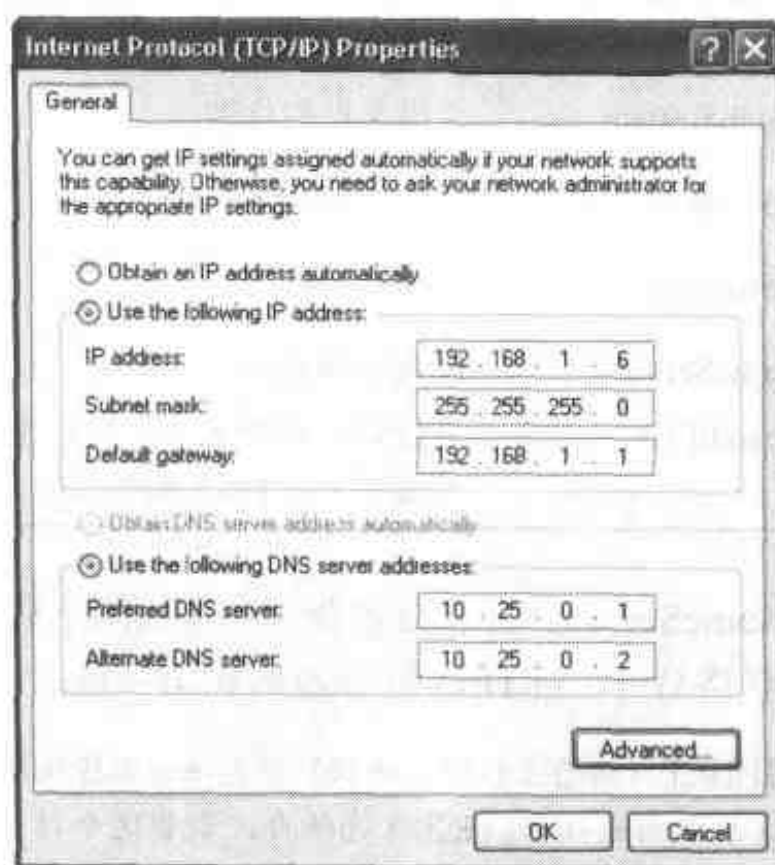


图4.5 配置DNS服务器地址的Internet Protocol Properties（因特网协议属性）窗口

DNS服务器配置和Windows

不同的Windows平台有它们自己的表示Internet Protocol Properties（因特网协议属性）的方法。虽然到达IP Properties（IP属性）窗口用户通过的途径不同，但在所有情况下，DNS服务器地址都放在IP Properties（IP属性）窗口。

同样，如果用户与ISP使用DHCP（动态主机配置协议）的话，用户也许根本没有任何DNS服务器配置，因为当用户连接到ISP的时候，这些是动态配置的。想知道是否已经正确地配置了DNS，最容易的方法是做一个简单的ping测试。可以在命令提示窗口，键入ping www.microsoft.com，看会出现什么。如果主机名解析为一个IP地址，说明你的DNS配置工作正常。

用C#调查DNS配置情况

当你的C#应用程序在自己客户的系统中运行的时候，不能保证客户系统中是否有DNS配置。你可以用.NET Registry类的方法检查DNS Registry值，来确定配置了什么DNS服务器（如果有的话）。

幸好，所有的Windows平台都在相同的地方存储DNS服务器信息：

```
HKLM\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters
```

这个关键字（key）在Internet Protocol Properties（因特网协议属性）窗口中按单独的数据值存储了所有的DNS参数。表4.4说明了在这个关键字下存储的一些值。

表4.4 DNS服务器的注册表数据值

数据值	说明
DatabasePath	主机文件的位置
Domain	系统的域名
Hostname	系统的DNS主机名
NameServer	DNS服务器列表
SearchList	在DNS名的搜索中，附在主机名后面的一个DNS域的列表

这里最有趣的值是NameServer。它应当包含一个单独的字符串值，表示所有配置的DNS服务器，服务器之间用空格分开。主DNS服务器列在第一位，后面跟着的是备用服务器。

说明：如果工作站使用DHCP（动态主机配置协议）动态地分配IP地址的话，它可能也动态地分配DNS服务器地址。这些值用连接的DHCP信息保存，代替这个讨论中的常规IP参数部分。

用户可以创建一个C#程序，用.NET Registry类来查询这个注册表值。程序清单4.2的FindDNSServers.cs程序演示了怎样用C#程序查询。

程序清单4.2 FindDNSServer.cs程序

```
using System;
using Microsoft.Win32;

class FindDNSServers
{
```

```

public static void Main()
{
    RegistryKey start = Registry.LocalMachine;
    string DNSservers = @"SYSTEM\CurrentControlSet\Services\Tcpip\Parameters";

    RegistryKey DNSserverKey = start.OpenSubKey(DNSservers);
    if (DNSserverKey == null)
    {
        Console.WriteLine("Unable to open DNS servers key");
        return;
    }

    string serverlist = (string)DNSserverKey.GetValue("NameServer");

    Console.WriteLine("DNS Servers: {0}", serverlist);
    DNSserverKey.Close();
    start.Close();

    char[] token = new char[1];
    token[0] = ' ';
    string[] servers = serverlist.Split(token);

    foreach(string server in servers)
    {
        Console.WriteLine("DNS server: {0}", server);
    }
}
}

```

在FindDNSServer.cs程序中，第一步是用HKLM值创建一个基本的RegistryKey对象：

```
RegistryKey start = Registry.LocalMachine;
```

在设置了基本关键字之后，可以用OpenSubKey()方法创建一个RegistryKey对象，指定DNS信息放在什么地方：

```

string DNSservers =
    @"SYSTEM\CurrentControlSet\Services\Tcpip\Parameters";
RegistryKey DNSserverKey = start.OpenSubKey(DNSservers);

```

DNSservers字符串指出DNS信息的正确注册表键位置的文本版本。

提示：注意，这个定义用@符号告诉C#编译器，反斜线符号不用于表示忽略次序，而是作为一个反斜线符号文字使用。这样防止用户每次键入两个反斜线符号。

OpenSubKey()方法用于把新的关键字分配给DNSServerKey RegistryKey对象。当正确地引用了关键字之后就可以用GetValue()方法从关键字检索数据值了：

```

string serverlist =
    (string)DNSserverKey.GetValue("NameServer");

```

`serverlist`变量现在应当包含一个字符串，该字符串包括了所有配置的DNS服务器（如果有的话）。为了把它们分成单个的IP地址，应当使用字符串类的`Split()`方法。

`Split()`方法要求用一个字符数组指定分开字符串的字符。由于用户知道每个DNS服务器值都是用两个空格分隔开的，所以可以指定一个只包含一个条目的字符数组，并用`Split()`方法把字符串分成字符串数组：

```
char[] token = new char[1];
token[0] = ' ';
string[] servers = serverlist.Split(token);
```

在有了一个DNS服务器的数组之后，就可以用过去用过的C# `foreach`把它们分成单个的字符串值了：

```
foreach(string server in servers)
{
    Console.WriteLine("DNS server: {0}", server);
}
```

如果用户需要使用程序中的这些地址的话，只要简单地用服务器的数组引用它们就行了（`servers[0]`是主DNS服务器，`servers[1]`是第一备用服务器，依此类推）。

该程序的输出是很简单的。首先显示整个的服务器字符串，然后分成一个服务器一行：

```
C:\>FindDNSServers
DNS Servers: 10.25.0.1 10.25.0.2
DNS server: 10.25.0.1
DNS server: 10.25.0.2
C:\>
```

这个输出应当与用户在Internet Protocol Properties（因特网协议属性）窗口中手工检查DNS服务器的条目相匹配。两种DNS服务器的条目都以它们从注册表检索出来最初的字符串出现。然后再用`Split()`字符串方法把单个的服务器分开。

用nslookup解析主机名

在设置了DNS服务器的配置之后，可以用命令行`nslookup`程序查询服务器的DNS信息，手工地检测DNS功能。`nslookup`程序是所有Windows平台带有的标准程序。这一节将向读者展示怎样用`nslookup`获取域和主机的DNS信息。

默认的nslookup查询

有许多使用`nslookup`程序的方法。最简单的方法是执行一个对主机名的查询：

```
C:\>nslookup www.microsoft.com
Server: dns.ispnet.net
Address: 10.25.0.1

Non-authoritative answer:
```



```
Name:      www.microsoft.akadns.net
Addresses: 207.46.197.100, 207.46.230.218, 207.46.197.113, 207.46.197.102
           207.46.230.220, 207.46.230.219
Aliases:   www.microsoft.com

C:\>
```

nslookup命令的输出显示出好几条信息:

- 查询的DNS服务器的名字和地址:

```
Server:  dns.ispnet.net
Address: 10.25.0.1
```

- 查询结果的状态:

```
Non-authoritative answer:
```

- 查询的结果:

```
Name:      www.microsoft.akadns.net
Addresses: 207.46.197.100, 207.46.230.218, 207.46.197.113, 207.46.197.102
           207.46.230.220, 207.46.230.219
Aliases:   www.microsoft.com
```

在这种情况下要注意, **www.microsoft.com**这个DNS主机名实际上是主机**www.microsoft.akadns.net**的一个别名。这个特殊的主机还有6个分配给它的IP地址。

结果的状态告诉用户信息是从哪里得来的。这个例子中的结果状态是**nonauthoritative answer** (非权威性的解答), 这意味着该信息是从本地DNS服务器的一个高速缓存入口得到的, 不是从分配给这个域的DNS服务器得到的。

为了证实是否本地DNS服务器是高速缓存查询, 可以试着查询一个新的过去没有查询过的主机名, 然后试着进行该查询的第二次查询:

```
C:\>nslookup msdn.microsoft.com
Server:  dns.ispnet.net
Address: 10.25.0.1

Name:    msdn.microsoft.com
Addresses: 207.46.239.122, 207.46.196.115

C:\>nslookup msdn.microsoft.com
Server:  dns.ispnet.net
Address: 10.25.0.1

Non-authoritative answer:
Name:    msdn.microsoft.com
Addresses: 207.46.196.115, 207.46.239.122

C:\>
```

第一个查询没有显示结果的状态行, 这说明这个结果直接来自远程主机域的DNS服务

器。不过，第二个查询显示结果是非权威性的（nonauthoritative），这说明结果来自本地DNS服务器的高速缓存。

用户也可以做一次颠倒IP地址的查询，以确定IP地址是分配给哪个主机的（如果有的话）：

```
C:\>nslookup 207.46.196.115
Server:  dns.ispnet.net
Address: 10.25.0.1

Name:     msdn.microsoft.com
Address:  207.46.196.115

C:\>
```

与主机查询不同的是，IP地址查询没有存在本地DNS服务器的高速缓存中。所以，如果用户执行第二次查询的话，仍然是到域的DNS服务器中去查询，返回的是一个权威性的结果。

监视查询

用户可以用WinDump或者Analyzer程序从本地系统监视DNS查询（这些工具程序在第2章“IP编程基础”中有介绍）。程序清单4.3显示的是，在使用nslookup命令时windump输出的例子。

程序清单4.3 nslookup windump输出

```
D:\winpcap>windump -s 200 udp port 53
windump listening on\Device\Packet_E190x1
07:16:45.208103 192.168.1.6.1219 > dns.ispnet.net.53: 2+ A? www.microsoft.com
. (35)
07:16:45.208888 dns.ispnet.net.53 > 192.168.1.6.1219: 2 7/7/7 CNAME www.micro
soft.akadns.net., A microsoft.com, A microsoft.com, A microsoft.com, A www.inter
national.microsoft.com, A microsoft.com, (400)
07:16:50.260472 192.168.1.6.1222 > dns.ispnet.net.53: 3+ PTR? 100.197.46.207.
in-addr.arpa. (45)
07:16:50.261416 dns.ispnet.net.53 > 192.168.1.6.1222: 3 4/13/9 PTR microsoft.
com., PTR microsoft.net., PTR www.domestic.microsoft.com., PTR www.us.microsoft.
com. (499)
07:16:50.263891 192.168.1.6.1223 > dns.ispnet.net.53: 4+ PTR? 219.230.46.207.
in-addr.arpa. (45)
07:16:50.265999 dns.ispnet.net.53 > 192.168.1.6.1223: 4 4/13/8 PTR microsoft.
com., PTR microsoft.net., PTR www.international.microsoft.com., PTR www.us.micro
soft.com. (488)
07:16:50.268427 192.168.1.6.1224 > dns.ispnet.net.53: 5+ PTR? 218.230.46.207.
in-addr.arpa. (45)
07:16:50.270266 dns.ispnet.net.53 > 192.168.1.6.1224: 5 4/13/9 PTR microsoft.
com., PTR microsoft.net., PTR www.domestic.microsoft.com., PTR www.us.microsoft.
```



```

com. (499)
07:16:50.272684 192.168.1.6.1225 > dns.ispnet.net.53: 6+ PTR? 113.197.46.207.
in-addr.arpa. (45)
07:16:50.273576 dns.ispnet.net.53 > 192.168.1.6.1225: 6 2/13/11 PTR www.inter
national.microsoft.com., PTR www.us.microsoft.com. (498)
07:16:50.278458 192.168.1.6.1226 > dns.ispnet.net.53: 7+ PTR? 220.230.46.207.
in-addr.arpa. (45)
07:16:50.399074 dns.ispnet.net.53 > 192.168.1.6.1226: 7* 4/0/0 PTR microsoft.
com., PTR microsoft.net., PTR www.domestic.microsoft.com., PTR www.us.microsoft.
com. (147)

239 packets received by filter
0 packets dropped by kernel

D:\winpcap>

```

可以看出，在一个简单的nslookup查询后面发生了许多动作。下面是一个用于进行这种捕获的WinDump命令：

```
windump -s 200 udp port 53
```

这个命令用-s选项把显示数据的数量设置成200个字节。由于DNS包经常包含很多返回的数据，所以这个设置有助于WinDump显示比默认的64个字节更多的包信息。当然有可能在返回包中出现更多的数据，不过现在，这已经是足够的信息供用户开始调查用了。还要注意，参数被设置成只捕获DNS包。DNS协议使用UDP，被指定到一个众所周知的端口号53。幸好，WinDump解码DNS查询后呈现给用户的是文本信息，所以这里没有协议调试的必要。

第一个包是预料之中的，是对于www.microsoft.com主机名A记录的DNS查询：

```
07:16:45.208103 192.168.1.6.1219 > dns.ispnet.net.53: 2+ A? www.microsoft.com
. (35)
```

返回的包含有DNS查询的信息：

```
07:16:45.208888 dns.ispnet.net.53 > 192.168.1.6.1219: 2 7/7/7 CNAME www.micro
soft.akadns.net., A microsoft.com, A microsoft.com, A microsoft.com, A www.inter
national.microsoft.com, A microsoft.com, (400)
```

这表示www.microsoft.com主机名的确是另一个主机的别名。A记录信息告诉我们，该系统真实的主机名是www.microsoft.akadns.net。

然后nslookup程序继续查询真实主机名的IP地址信息。PTR记录信息显示分配给该主机的IP地址，以及给每个IP地址分配的主机名。

```

007:16:50.260472 192.168.1.6.1222 > dns.ispnet.net.53: 3+ PTR? 100.197.46.207.
in-addr.arpa. (45)
07:16:50.261416 dns.ispnet.net.53 > 192.168.1.6.1222: 3 4/13/9 PTR microsoft.
com., PTR microsoft.net., PTR www.domestic.microsoft.com., PTR www.us.microsoft.
com. (499)

```

第一个IP地址207.46.197.100用于查询PTR记录的信息。返回的信息显示，这个地址分配给了microsoft.com域。

高级查询

前面几节已经介绍过，默认的nslookup格式提供了关于被查询主机名适当的简单信息。如果用户想了解其他DNS记录的类型就必须使用nslookup的选项。nslookup的选项可以在命令行中指定，但用交互式的模式使用nslookup通常更加容易。

为了进入交互式模式，可以在命令提示下不加任何参数简单地键入nslookup。使用的默认DNS服务器会显示出来，还显示出一个nslookup提示：

```
C:\>nslookup
Default Server:  dns.ispnet.net
Address: 10.25.0.1

>
```

在nslookup提示下，用户可以键入各种专用的命令，来修改nslookup查询的行为。各种nslookup命令及其说明列在表4.5中。

表4.5 nslookup命令

命令	说明
NAME	解析主机名NAME
NAME1 NAME2	用DNS服务器NAME2解析主机名NAME1
Help	列出所有可用的nslookup命令和选项
Set	设置一个nslookup选项
Server NAME	设置默认的DNS服务器为NAME，使用当前默认的服务器
Lserver NAME	设置默认的DNS服务器为NAME，使用最初的服务器
Finger [USER]	用Finger实用程序在当前默认的主机上找出USER
Root	把当前默认的DNS服务器设置为根服务器
ls DOMAIN	列出所有已经在DOMAIN中注册的地址
View	观看一个用ls命令建立的文件
Exit	退出nslookup命令模式

正如读者在表4.5中所看到的，用户可以用set命令建立各种DNS查询的选项。表4.6列出了这些选项的定义，在接下来的几节里讨论了其中几个最常用的命令。

表4.6 nslookup Set (设置) 选项

选项	说明
All	打印选项, 当前服务器和主机
[no]debug	打印 (或者不打印) 调试信息
[no]d2	打印 (或者不打印) 详尽的调试信息
[no]defname	在每个查询的后面附加 (或不附加) 域名
[no]recurse	要求对每个查询递归的结果
[no]search	使用域搜索目录
[no]vc	总是使用虚拟回路
Domain=NAME	设置默认的域名为NAME
Srchlist=N1[N2/.../N6]	设置域为N1, 搜索目录为N1、N2等
root=NAME	设置DNS根服务器为NAME
Retry=X	设置重试次数为X
Timeout=X	设置最初的超时间隔时间为X秒
type=X	设置查询类型
Querytype=X	设置查询类型
Class=X	设置查询类
[no]msxfr	使用Microsoft的快速区域传输
Ixfrver=X	设置在IXFR传输请求中使用的当前版本

debug (调试) 选项

debug选项使用户可以看到与DNS服务器的DNS通信。程序清单4.4显示的是一个debug (调试) 选项打开的DNS查询的范例。这个信息正确地显示出在WinDump结果中看到的内容: 最初对于www.microsoft.com主机名的查询, 其结果表示它是www.microsoft.akadns.net主机的别名, 并显示了所有与它有关的IP地址。

程序清单4.4 debug (调试) 选项打开的nslookup查询

```
C:\>nslookup
Default Server  dns.ispnet.net
Address: 10.25.0.1

> set debug
> www.microsoft.com
Server:  dns.ispnet.net
Address: 10.25.0.1
```

Got answer:

HEADER:

opcode = QUERY, id = 2, rcode = NOERROR

header flags: response, want recursion, recursion avail.

questions = 1, answers = 7, authority records = 7, additional = 7

QUESTIONS:

www.microsoft.com, type = A, class = IN

ANSWERS:

-> www.microsoft.com
canonical name = www.microsoft.akadns.net
ttl = 7116 (1 hour 58 mins 36 secs)

-> www.microsoft.akadns.net
internet address = 207.46.230.219
ttl = 216 (3 mins 36 secs)

-> www.microsoft.akadns.net
internet address = 207.46.230.220
ttl = 216 (3 mins 36 secs)

-> www.microsoft.akadns.net
internet address = 207.46.197.100
ttl = 216 (3 mins 36 secs)

-> www.microsoft.akadns.net
internet address = 207.46.230.218
ttl = 216 (3 mins 36 secs)

-> www.microsoft.akadns.net
internet address = 207.46.197.102
ttl = 216 (3 mins 36 secs)

-> www.microsoft.akadns.net
internet address = 207.46.197.113
ttl = 216 (3 mins 36 secs)

AUTHORITY RECORDS:

-> akadns.net
nameserver = ZA.akadns.net
ttl = 127117 (1 day 11 hours 18 mins 37 secs)

-> akadns.net
nameserver = ZC.akadns.net
ttl = 127117 (1 day 11 hours 18 mins 37 secs)

-> akadns.net
nameserver = ZD.akadns.net
ttl = 127117 (1 day 11 hours 18 mins 37 secs)

-> akadns.net
nameserver = ZE.akadns.net
ttl = 127117 (1 day 11 hours 18 mins 37 secs)

-> akadns.net

```
nameserver = ZF.akadns.net
ttl = 127117 (1 day 11 hours 18 mins 37 secs)
-> akadns.net
nameserver = ZG.akadns.net
ttl = 127117 (1 day 11 hours 18 mins 37 secs)
-> akadns.net
nameserver = ZH.akadns.net
ttl = 127117 (1 day 11 hours 18 mins 37 secs)
ADDITIONAL RECORDS:
-> ZA.akadns.net
internet address = 216.32.65.105
ttl = 127117 (1 day 11 hours 18 mins 37 secs)
-> ZC.akadns.net
internet address = 63.241.199.50
ttl = 127117 (1 day 11 hours 18 mins 37 secs)
-> ZD.akadns.net
internet address = 206.132.160.36
ttl = 127117 (1 day 11 hours 18 mins 37 secs)
-> ZE.akadns.net
internet address = 12.47.217.11
ttl = 127117 (1 day 11 hours 18 mins 37 secs)
-> ZF.akadns.net
internet address = 63.215.198.79
ttl = 127117 (1 day 11 hours 18 mins 37 secs)
-> ZG.akadns.net
internet address = 204.248.36.131
ttl = 127117 (1 day 11 hours 18 mins 37 secs)
-> ZH.akadns.net
internet address = 63.208.48.42
ttl = 127117 (1 day 11 hours 18 mins 37 secs)

-----
Non-authoritative answer:
Name:      www.microsoft.akadns.net
Addresses: 207.46.230.219, 207.46.230.220, 207.46.197.100, 207.46.230.218
           207.46.197.102, 207.46.197.113
Aliases:   www.microsoft.com

>exit

C:\>
hostname
```

querytype (查询类型) 选项

另外一个有用的功能是设置querytype选项。用户用这个选项可以缩小主机特定信息的

范围，或者考虑全部域。例如，用户可以设置查询类型为SOA，以检索该域的SOA记录：

```
C:\>nslookup
Default Server:  dns.ispnet.net
Address: 10.25.0.1

> set querytype=soa
> microsoft.com
Server:  dns.ispnet.net
Address: 10.25.0.1

microsoft.com
    primary name server = dns.cp.msft.net
    responsible mail addr = msnhst.microsoft.com
    serial    = 2002061201
    refresh  = 900 (15 mins)
    retry    = 600 (10 mins)
    expire   = 7200000 (83 days 8 hours)
    default TTL = 7200 (2 hours)
dns.cp.msft.net internet address = 207.46.138.10
>
```

现在可以看到，当请求的域在本地DNS服务器中定义时域中全部的SOA记录。当用户想要确定某个域的邮件服务器的时候，这种方法同样有效。请看下面的例子：

```
C:\>nslookup
Default Server:  dns.ispnet.net
Address: 10.25.0.1

> set querytype=mx
> microsoft.com
Server:  dns.ispnet.net
Address: 10.25.0.1

microsoft.com  MX preference = 10, mail exchanger = maila.microsoft.com
microsoft.com  MX preference = 10, mail exchanger = mailb.microsoft.com
microsoft.com  MX preference = 10, mail exchanger = mailc.microsoft.com
maila.microsoft.com    internet address = 131.107.3.125
maila.microsoft.com    internet address = 131.107.3.124
mailb.microsoft.com    internet address = 131.107.3.123
mailb.microsoft.com    internet address = 131.107.3.122
mailc.microsoft.com    internet address = 131.107.3.126
mailc.microsoft.com    internet address = 131.107.3.121
>
```

这个结果可以看出，microsoft.com域有三个独立的邮件服务器。信息显示邮件服务器的主机名，以及它们的IP地址。

域转储选项 (ls)

nslookup命令的**ls**选项允许系统管理员获得一个域指定的DNS数据库全部的转储资料。可以用这个选项研究整个域中所有的记录。不过要预先警告读者，我在这里说“可以”是因为我不能保证它起作用。因为为了防止未经授权的用户看到整个DNS数据库的内容，许多DNS服务器都禁止使用这个命令。

如果读者有权使用**ls**命令的话，可以用以下重定向命令(>)把全部的DNS数据库转储到一个文件中：

```
C:\>nslookup
Default Server: dns.ispnet.net
Address: 10.25.0.1

> ls testdomain.com > td.txt
> view td.txt
```

然后再用**view**命令观看下载的含有域DNS数据库的文本文件。

C#中的DNS类

System.Net命名空间包含**Dns**类，这个类提供了C#程序中所有需要的DNS功能。这一节将介绍**Dns**类的方法，并告诉读者为了利用系统的DNS功能怎样在C#程序中使用这些方法。

同步方法

在**Dns**类中定义了以下4种同步方法：

- **GetHostName()**
- **GetHostByName()**
- **GetHostByAddress()**
- **Resolve()**

GetHostName()方法

读者在第2章中已经看到过这种方法。**GetHostName()**用于确定程序运行的本地系统的主机名。网络程序经常需要这种信息，所以读者会经常看到使用这个方法。它的格式很简单，不需要输入任何参数，结果是一个字符串对象：

```
string hostname = Dns.GetHostName();
```

用**GetHostName()**主机名检索到的信息应当与注册表的**Hostname**数据值中出现的是相同的名字，还有为建立系统完全合格的域名(FQDN)所要求的**Domain**数据值。FQDN包括本地主机名和全部域名信息。

GetHostByName()方法

GetHostByName()方法用系统中配置的默认DNS服务器为指定的主机名，执行一个DNS查询。这个方法的格式如下：

```
IPHostEntry GetHostByName(string hostname)
```

由GetHostByName()方法返回的IPHostEntry本身是一个有趣的对象。这个对象把一个DNS主机名与一个别名和IP地址的数组相关联。它包含三个属性：

- AddressList: 一个IPAddress对象的数组，一个对象用于一个分配的IP地址
- Aliases: 一个字符串对象数组，一个对象用于一个别名
- HostName: 一个用于主机名的字符串对象

AddressList和Aliases对象必须分成单独的数组元素才能检索存储的信息。通常用C#中的foreach功能来实现。

程序清单4.5是一个演示GetHostByName()方法的范例程序。

程序清单4.5 GetDNSHostInfo.cs程序

```
using System;
using System.Net;

class GetDNSHostInfo
{
    public static void Main(string[] argv)
    {
        if (argv.Length != 1)
        {
            Console.WriteLine("Usage: GetDNSHostInfo hostname");
            return;
        }

        IPHostEntry results = Dns.GetHostByName(argv[0]);

        Console.WriteLine("Host name: {0}",
                           results.HostName);

        foreach(string alias in results.Aliases)
        {
            Console.WriteLine("Alias: {0}', alias);
        }

        foreach(IPAddress address in results.AddressList)
        {
            Console.WriteLine("Address: {0}",
                               address.ToString());
        }
    }
}
```

GetDNSHostInfo.cs程序是相当易懂的。它从命令行输入参数，试图用它执行GetHostByName()方法，并转储从这个方法返回的IPHostEntry。如果有多个IP地址分配到这个主机名，

每个地址都将出现在AddressList属性中。foreach函数用于提取并显示每一个IP地址。以下是一个样本会话的输出：

```
C:\>GetDNSHostInfo www.microsoft.com
Host name: www.microsoft.akadns.net
Alias: www.microsoft.com
Alias: www.microsoft.com
Address: 207.46.197.102
Address: 207.46.230.220
Address: 207.46.230.218
Address: 207.46.230.219
Address: 207.46.197.100
Address: 207.46.197.113

C:\>
```

GetHostByName()方法返回的信息与默认的nslookup命令接收到的信息相同。事实上，用户可以运行windump或者analyzer程序，并观看由GetHostByName()方法产生的DNS查询。它看起来与程序清单4.4中默认的nslookup样本所产生的结果类似。

正如第2章的程序清单2.4中所显示的那样，GetHostByName()方法经常与GetHostName()方法一起使用，用于确定本地系统的IP地址。用法如下：

```
IPHostEntry localaddrs = Dns.GetHostByName(Dns.GetHostName());
```

最后的IPHostEntry对象将包含AddressList属性中为本地主机配置的所有IP地址。

GetHostByAddress()方法

当使用程序清单4.5中的GetDNSHostInfo.cs程序并输入IP地址的时候，很可惜不会产生主机名：

```
C:\>GetDNSHostInfo 207.46.197.100
Host name: 207.46.197.100
Address: 207.46.197.100

C:\>
```

显然，这不能使人感到振奋，GetHostByName()方法只能解析主机名。

当用户需要找到一个已知IP地址的主机名的时候，可以使用GetHostByAddress()方法。这个方法有两种格式：

```
IPHostEntry GetHostByAddress(IPAddress address)
IPHostEntry GetHostByAddress(string address)
```

当用户有一个像IPAddress对象那样的IP地址的时候，使用第一种格式。当用户想使用点分格式IP地址（例如207.46.197.100）的字符串表示法的时候，使用第二种格式。

程序清单4.6中的GetDNSAddressInfo.cs程序演示了GetHostByAddress()方法的使用。

程序清单4.6 GetDNSAddressInfo.cs程序

```
using System;
using System.Net;

class GetDNSAddressInfo
{
    public static void Main(string[] argv)
    {
        if (argv.Length != 1)
        {
            Console.WriteLine("Usage: GetDNSAddressInfo address");
            return;
        }

        IPAddress test = IPAddress.Parse(argv[0]);

        IPHostEntry iphe = Dns.GetHostByAddress(test);

        Console.WriteLine("Information for {0}",
                           test.ToString());

        Console.WriteLine("Host name: {0}", iphe.HostName);
        foreach(string alias in iphe.Aliases)
        {
            Console.WriteLine("Alias: {0}", alias);
        }
        foreach(IPAddress address in iphe.AddressList)
        {
            Console.WriteLine("Address: {0}", address.ToString());
        }
    }
}
```

只是为了娱乐（和体验），这个程序把字符串变量转换成一个`IPAddress`对象，该对象被用做`GetHostByAddress()`方法的参数。输出如下：

```
C:\>GetDNSAddressInfo 207.46.197.113
Information for 207.46.197.113
Host name: www.international.microsoft.com
Address: 207.46.197.113

C:\>
```

再一次，用户可以用`windump`或`analyzer`程序观看这个查询产生的DNS网络通信量。

警告：`GetHostByAddress()`方法并不总是有效的，用户可能经常会遇到IP地址没有解析成主机名的情况。发生这种情况有两个原因。一个是没有给该地址分配DNS主机名。另一个原因是，虽然存在一个DNS A记录，但DNS管理员没有使用指向该地址的PTR记录返回主机名。

Resolve()方法

正如读者在前一节中所看到的，用**GetHostByName()**和**GetHostByAddress()**方法的主要优点是它们都是专门使用一种或另外一种地址信息的。如果把IP地址提供给**GetHostByName()**方法，它只会返回该地址。更糟的是，如果用户试图把一个字符串主机名提供给**GetHostByAddress()**方法的话，程序会产生异常（因为它在提供的字符串IP地址上使用**IPAddress.Parse()**方法）。

假设用户正在编写一个程序，他的客户需要输入一个远程主机的地址——用户可能不知道客户会用哪种格式。其实不需要费事地去判断客户输入的是数字还是文本，然后再使用正确的GetHostBy方法，Dns类提供了简单的解决方案：**Resolve()**方法。**Resolve()**方法可以接受或者是主机名格式或者是IP地址格式的任何一种地址，并返回在IPHostEntry对象中的DNS信息。程序清单4.7演示了这种方法。

程序清单4.7 GetResolveInfo.cs程序

```
using System;
using System.Net;

class GetResolveInfo
{
    public static void Main(string[] argv)
    {
        if (argv.Length != 1)
        {
            Console.WriteLine("Usage: GetResolveInfo address");
            return;
        }

        IPEndPoint iphe = Dns.Resolve(argv[0]);

        Console.WriteLine("Information for {0}", argv[0]);
        Console.WriteLine("Host name: {0}", iphe.HostName);
        foreach (string alias in iphe.Aliases)
        {
            Console.WriteLine("Alias: {0}", alias);
        }
        foreach (IPAddress address in iphe.AddressList)
        {
            Console.WriteLine("Address: {0}",
                              address.ToString());
        }
    }
}
```

Resolve()方法试图用输入的地址信息作为参数提供给**IPHostEntry**对象。用户用主机名或者用IP地址都可以，而**Resolve()**方法将返回正确的信息。这是一种处理客户在程序中可能输入任何主机信息的比较友好的方法。

```
C:\>GetResolveInfo www.microsoft.com
Information for www.microsoft.com
Host name: www.microsoft.akadns.net
Alias: www.microsoft.com
Alias: www.microsoft.com
Address: 207.46.230.219
Address: 207.46.230.218
Address: 207.46.197.100
Address: 207.46.197.102
Address: 207.46.230.220
Address: 207.46.197.113

C:\>GetResolveInfo 207.46.197.100
Information for 207.46.197.100
Host name: microsoft.com
Address: 207.46.197.100

C:\>
```

异步方法

正规的Dns方法可能会给用户的C#程序带来麻烦，因为它们用阻塞方式与远程DNS服务器通信。如果读者运行本节前面提供的一个DNS类程序的话，可能会注意到，程序经常需要几秒钟时间等待从远程DNS服务器返回的DNS信息。这也许看起来不算什么问题，可是在允许用户同时做其他事情的Windows程序中这就有问题了。因为在程序等待DNS服务器响应的时候，用户不能够单击任何按钮也不能往文本字段输入任何信息。对某些应用程序来说，这是无法接受的性能。

第3章“C#网络编程的类”中，讲述了可用于防止网络阻塞调用的技术。其中的一种技术就是用异步套接字方法（在第8章“异步套接字编程”中有详细的叙述）。这些方法允许用户启动一个网络函数并指定一个当网络函数完成时调用的代表。在此期间，其他程序按自己的方式完成其他的工作。当网络函数完成时，它会用信号通知程序运行代表方法（delegate）。

Dns类提供了下列异步方法，允许用户程序执行异步和DNS功能调用：

- **BeginGetHostByName()**
- **BeginResolve()**
- **EndGetHostByName()**
- **EndResolve()**

每个异步方法都有一个相应的同步方法。这一次，没有异步的**GetHostByAddress()**调用。

这些方法都是成对调用的。用户从程序中调用**Beginxxx**方法并指定一个包含有**Endxxx**方法的代表方法。例如，**BeginResolve()**使用以下格式：

```
public static IAsyncResult BeginResolve(string hostname,
                                       AsyncCallback requestCallback, object stateObject)
```

这个方法使用三个参数：

- 一个主机名或IP地址的字符串代表hostname
- 一个AsyncCallback对象，requestCallback，它指定代表
- 一个普通对象，stateObject，该对象指定方法的状态

建立的AsyncCallback对象指定当DNS结果返回的时候使用的代表方法。为了做到这一点，首先要创建AsyncCallback对象，然后再给它指派一个代表方法：

```
private AsyncCallback OnResolved;
OnResolved = new AsyncCallback(Resolved);
Object state = new Object();
.
.
Dns.BeginResolve(addr, OnResolved, state);
.
}
private void Resolved(IAsyncResult ar)
{
    IPEndPoint iphe = Dns.EndResolve(ar);
}
```

OnResolved AsyncCallback对象指定了代表方法Resolved，这个方法将包含完成DNS调用的EndResolve()方法。state对象允许用户把信息传递给代表方法。EndResolve()方法的格式为：

```
public static IPEndPoint EndResolve(IAsyncResult ar)
```

IAsyncResult对象ar是指最初BeginResolve()方法的EndResolve()方法。EndResolve()方法的结果是标准的IPEndPoint对象，它可以像本书前面讨论过的标准的Resolve()方法那样被分开，以提取返回的信息。

程序清单4.8是在Windows窗体环境中使用BeginResolve()和EndResolve()方法的一个范例程序。

程序清单4.8 AsyncResolve.cs程序

```
using System;
using System.Drawing;
using System.Net;
using System.Text;
using System.Windows.Forms;

class AsyncResolve Form:
{
    TextBox address;
```

```
Listbox results;
private AsyncCallback OnResolved;

public AsyncResolve()
{
    Text = "DNS Address Resolver";
    Size = new Size(400,380);
    OnResolved = new AsyncCallback(Resolved);

    Label label1 = new Label();
    label1.Parent = this;
    label1.Text = "Enter address to resolve:";
    label1.AutoSize = true;
    label1.Location = new Point(10, 10);

    address = new TextBox();
    address.Parent = this;
    address.Size = new Size(200, 2 * Font.Height);
    address.Location = new Point(10, 35);

    results = new Listbox();
    results.Parent = this;
    results.Location = new Point(10, 65);
    results.Size = new Size(350, 20 * Font.Height);

    Button checkit = new Button();
    checkit.Parent = this;
    checkit.Text = "Resolve";
    checkit.Location = new Point(235,32);
    checkit.Size = new Size(7 * Font.Height, 2 * Font.Height);
    checkit.Click += new EventHandler(ButtonResolveOnClick);
}

void ButtonResolveOnClick(object obj, EventArgs ea)
{
    results.Items.Clear();
    string addr = address.Text;
    Object state = new Object();

    Dns.BeginResolve(addr, OnResolved, state);
}

private void Resolved(IAsyncResult ar)
{
    string buffer;

    IPHostEntry iphe = Dns.EndResolve(ar);

    buffer = "Host name: " + iphe.HostName;
```



```

        results.Items.Add(buffer);

        foreach(string alias in iphe.Aliases)
        {
            buffer = "Alias: " + alias;
            results.Items.Add(buffer);
        }
        foreach(IPAddress addrs in iphe.AddressList)
        {
            buffer = "Address: " + addrs.ToString();
            results.Items.Add(buffer);
        }
    }

    public static void Main()
    {
        Application.Run(new AsyncResolve());
    }
}

```

由于Windows窗体程序是在图形环境中操作的，在此环境下使用异步Dns方法比同步方法操作起来更困难一些。**AsyncResolve.cs**程序的第一部分定义了这个类的构造函数（constructor）。构造函数定义了一个标准的Windows窗体并在窗体中加上了4个控件：

- 一个显示一些指示性文本的Label（标签）对象
- 一个允许用户输入要解析的地址的TextBox（文本框）对象
- 一个把DNS查询的输出方便地返回给用户的ListBox（列表框）对象
- 一个允许用户启动DNS查询的Button（按钮）对象

在定义了构造函数之后，定义了两个方法。**ButtonResolveOnClick()**方法被注册为使用Click（单击）属性的按钮的代表方法：

```
checkit.Click += new EventHandler(ButtonResolveOnClick);
```

注意，**EventHandler**对象被加到了一个对于控件的现有的事件操作器列表中。**EventHandler**对象指的是，当察觉到按钮的Click（单击）事件时要调用的代表。

当Button（按钮）对象被单击的时候，就执行**ButtonResolveOnClick()**代表。它读取输入到TextBox（文本框）对象中的值，并用该对象的值、前面建立的**AsyncCallback**对象和空的状态对象调用**BeginResolve()**方法。

```
Dns.BeginResolve(addr, OnResolved, state);
```

OnResolved参数被定义为**AsyncCallback**对象，并在程序的类中定义**Resolved()**方法：

```
private AsyncCallback OnResolved = null;
OnResolved = new AsyncCallback(Resolved);
```

Resolved()方法调用**EndResolve()** Dns类方法，该方法完成**BeginResolve/EndResolve**对。

它必须用IAsyncResult对象的参数作为参数值:

```
private void Resolved(IAsyncResult ar)
{
    string buffer;

    IPHostEntry iphe = Dns.EndResolve(ar);
```

然后,用本章中其他DNS程序所使用的标准方法把IPHostEntry对象分开,找出从DNS查询中返回的主机名、所有的别名和IP地址。为了使操作更简单一些,结果被写在了一个ListBox(列表框)对象中。图4.6显示了该程序输出的一个例子。



图4.6 写到一个ListBox(列表框)对象中的AsyncResult程序的结果

AsyncResult程序可以显示主机名或者IP地址查询中得到的结果。当每次查询其他地址的时候,ListBox被清除,以便容易读取新的信息。如果用户想保留所有查询过程的记录的话,可以从ButtonResolveOnClick()方法中删除下列行:

```
results.Items.Clear();
```

如果用户这样做的话,可能还想在DNS查询信息写到ListBox(列表框)之后加上某种分隔符线,在查询之间起到分隔的作用。下面是一个例子:

```
results.Items.Add("-----");
```

当查询记录过长,在列表框中放不下的时候,垂直滚动条就会出现,允许用户滚动观看整个查询记录。这样短小的程序就产生了如此漂亮、整洁的结果!

小结

域名系统(DNS)可以被C#程序用于确定主机的IP地址。DNS是一分布式的分层数据库系统,它按层分配主机名,层与层之间用句点分开。创建DNS服务器是用于存储各个域的

DNS数据库。客户机可以查询DNS服务器，得到与主机名或其他域信息有关的IP地址。如果一个本地DNS服务器不能对查询做出答复，程序就查询更高一级的DNS服务器，直到得出查询结果，并将结果返回客户机。

Windows系统用两种方法解析主机名。第一种方法是使用主机文件，手工地输入主机名并提供快速的主机名解析方法。第二种方法是在系统中配置DNS服务器，与远程DNS服务器联系，并解析DNS信息。C#语言提供了RegistryKey类，可以查询本地注册表来获得系统中配置的DNS服务器地址。

Windows系统还包括nslookup程序，该程序使客户可以进行人工DNS查询。nslookup程序有多个选项，可以改变查找和显示DNS信息的方式。debug选项使用户可以看到实际的DNS查询送到远程DNS服务器的过程。querytype选项允许用户查询某个域的特殊DNS记录信息。

C#在System.Net命名空间包含的Dns类提供了基本的DNS查询功能。GetHostByName()和GetHostByAddress()方法使得C#程序能够确定给出特殊主机数据的DNS信息。Resolve()方法既可以找出主机名的DNS信息也可以找出IP地址的DNS信息。

本章还涉及到异步Dns方法，在那些不允许在等待DNS服务器返回响应的时候占用着系统资源的应用程序中，可以用Dns方法进行DNS查询。异步方法使用标准的C#代表(delegate)系统，在DNS查询返回的时候进行调用。这极大地改善了基于Windows的网络程序的性能。

第二部分 网络层编程

第5章 面向连接的套接字

- 在一个简单的TCP服务器应用程序中使用面向连接的套接字
- 使用面向连接套接字的TCP客户机应用程序
- 灵活应用TCP缓冲区大小
- 针对TCP的局限性保护数据的安全
- 在TCP中使用C#流：三个重要的类

本书的第二部分以第5章中所讲述的三个重要的类开始，这标志着我们的重点将从单纯的定义套接字转入实际的网络编程。在第二部分的每一章中，都列举了实际网络编程中如何使用C#套接字类的范例。本章中所举的范例主要针对所有使用TCP协议的通信中如何使用面向连接的套接字模型。

首先，我们将浏览使用基本Windows套接字接口的最简单的TCP服务器和客户机，并了解它们的局限性。接着，看一下复杂一些的TCP服务器和客户机，在这些应用中采取了一定的措施来解决TCP通信的一些固有问题。最后，我们将介绍第1章“C#语言”中讲述过的C# I/O流的使用方法以简化数据的读写操作。对于网络编程的新手来说，该技术能够简化TCP通信。

简单的TCP服务器

正如在第3章中我们所讲述过的，在服务器能够向客户机连接传输数据之前，必须做以下四件事情：

- 创建一个套接字
- 将所创建的套接字与本地IPEndPoint绑定
- 设置套接字为收听模式
- 在套接字上接收接入的连接

本小节内容主要介绍在创建简单的TCP服务器时上述过程的实现。在这里没有任何修饰的内容，仅仅是启动服务器并监听客户机连接所必需的内容，也是读者实现其他功能时应该遵循的一般原则。

创建服务器

构建TCP服务器的第一步是创建Socket对象的一个实例，其他的三个功能都是在该Socket对象的基础上实现。请看下面这一小段C#程序：

```
IPEndPoint ipep = new IPEndPoint(IPAddress.Any, 9050);
Socket newsock = Socket(AddressFamily.InterNetwork,
    SocketType.Stream, ProtocolType.Tcp);
newsock.Bind(ipep);
newsock.Listen(10);
Socket client = newsock.Accept();
```

由Accept()所创建的Socket对象能够用于在服务器和远程客户机之间的任意方向上传送数据。

上述所有的基本步骤都在SimpleTcpSrvr.cs中，代码如程序清单5.1所示。

程序清单5.1 SimpleTcpSrvr.cs程序

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class SimpleTcpSrvr
{
    public static void Main()
    {
        int recv;
        byte[] data = new byte[1024];
        IPEndPoint ipep = new IPEndPoint(IPAddress.Any,
            9050);

        Socket newsock = new
            Socket(AddressFamily.InterNetwork,
                SocketType.Stream, ProtocolType.Tcp);

        newsock.Bind(ipep);
        newsock.Listen(10);
        Console.WriteLine("Waiting for a client...");
        Socket client = newsock.Accept();
        IPEndPoint clientep =
            (IPEndPoint)client.RemoteEndPoint;

        Console.WriteLine("Connected with {0} at port {1}",
            clientep.Address, clientep.Port);

        string welcome = "Welcome to my test server";
```

```

        data = Encoding.ASCII.GetBytes(welcome);
        client.Send(data, data.Length,
                    SocketFlags.None);
    while(true)
    {
        data = new byte[1024];
        recv = client.Receive(data);
        if (recv == 0)
            break;

        Console.WriteLine(
            Encoding.ASCII.GetString(data, 0, recv));
        client.Send(data, recv, SocketFlags.None);
    }
    Console.WriteLine("Disconnected from {0}",
                    clientep.Address);
    client.Close();
    newsock.Close();
}
}

```

如前所述，在这个程序中没有任何修饰或技巧。首先，定义一个空字节数组作为数据缓冲器，用于缓存流入和流出的消息。切记，不论传输什么类型的数据，**Socket Receive()**和**Send()**只能使用字节数组，所有通过这个**Socket**传输的数据必须转换为字节数组。既然本例只能使用文本字符串，则**System.Text**命名空间中的**Encoding.ASCII**用于在字符串和字节数组之间进行相互转换。

接下来，为本地服务器定义**IPEndPoint**对象：

```
IPEndPoint ipep = new IPEndPoint(IPAddress.Any, 9050);
```

使用**IPAddress.Any**字段，该服务器能够接收可配置在系统中任何网络接口上的接入连接请求。如果希望仅在某特定的接口接收包信息，可以采用下列方式指定接口的IP地址：

```
IPEndPoint ipep = new
    IPEndPoint(IPAddress.Parse("192.168.1.6"), 9050);
```

在决定要使用的恰当的**IPEndPoint**对象后，接下来可以调用**Socket()**构造函数创建**TCP**套接字，然后使用**Bind()**和**Listen()**将该套接字与新**IPEndPoint**对象进行绑定，并收听接入连接。

最后，使用**Accept()**方法接收来自客户机的接入连接尝试，**Accept()**返回一个新**Socket**对象，在与客户机的所有通信中必须使用这个新**Socket**对象。

Accept()接收连接尝试后，从使用**RemoteEndPoint**属性所返回的套接字中可以得到发出请求的客户机的IP地址信息：

```
IPEndPoint clientep =
    (IPEndPoint)client.RemoteEndPoint;
```

因为`RemoteEndPoint`属性指定的是任意类型的`EndPoint`对象，因此必须将它与一个`IPEndPoint`对象相匹配，一旦使用远程客户机信息创建了这个`IPEndPoint`对象，我们就可以使用`Address`和`Port`属性来访问它（如程序中所示）。该技术在识别连接到服务器上的单独客户机时非常方便。

在套接字创建后，这个客户机/服务器必须对传输数据的系统达成协议。如果客户机和服务器都使用块套接字（默认情况），而且同时试图接收数据，则它们将进入死锁状态，无休止地等待下去。同样，如果二者同时发送数据，也会发生类似的死锁现象。因此，必须达到一种“收/发”效果，在一方发送数据时，另一方等待接收数据。

在这个范例中，服务器直接发送一个欢迎消息到客户机，然后等待客户机的消息：

```
string welcome = "Welcome to my test server";
data = Encoding.ASCII.GetBytes(welcome);
client.Send(data, data.Length,
            SocketFlags.None);

while(true)
{
    data = new byte[1024];
    recv = client.Receive(data);
    if (recv == 0)
        break;

    Console.WriteLine(
        Encoding.ASCII.GetString(data, 0, recv));
    client.Send(data, recv, SocketFlags.None);
}
```

希望与这个服务器进行通信的客户机必须准备好，一旦连接建立，就接收来自服务器的欢迎消息，接收欢迎消息后，客户机则必须决定是发送还是接收数据。

因为套接字以字节格式发送和接收数据，因此必须将字符串消息转换为一个字节长度的数组类型，将流入的字节数组转换为字符串对象，最简便的方法是使用`System.Text`命名空间中的`Encoding`类。如果在服务器中使用的是ASCII码，则在`GetBytes()`和`GetString()`中使用这些ASCII码能够方便地进行字符串和字节数组之间的转换，如下所示：

```
Byte[] data = Encoding.ASCII.GetBytes("test string");
string word = Encoding.ASCII.GetString(data);
```

服务器收到来自客户机的消息后，立即将接收到的消息发送回客户机，这个功能叫做`echo server`，经常用于测试通信信道。上述代码中非常有趣的是服务器如何得知不再等待其他数据的方法，在`SimpleTcpSrvr`程序中的这段代码采用检查`recv`变量是否为零的方式。

Receive()方法

读者可能已经注意到在程序清单5.1中`while`循环的第一行，这一行代码将`data`变量复位为一个新字节数组。这是非常关键的一个细节，没经验的网络程序员经常忘记这一点。由于

`Receive()`将数据放置在数据缓冲区中，因此就设置了数据缓冲区的大小。如果数据缓冲区没有复位成它原来的尺寸，下一个使用该缓冲区的`Receive()`调用将只能使用前一个调用所使用过的数据缓冲区的大小。

正常情况下，`Receive()`返回从远程客户机中所接收到的数据的数量。如果没有接收到数据，该`Receive()`阻塞，等待数据。读者可能猜想，`Receive()`怎么会返回零值？很简单，当TCP检测到远程客户机初始化了一个关闭会话（通过发送TCP FIN包）时，就允许`Receive()`返回零值。

因此可以检查`Receive()`的返回值是否为零，如果是零，则该程序必须努力关闭新近创建的客户机套接字（可以使用`Close()`和`Shutdown()`）。在客户机套接字关闭后，就不再有其他通信了。然而必须记住，原始的主Socket对象仍然是活动的，而且仍然可以用于接收接入连接，直到它也被关闭。如果根本不需要其他的连接，也可以关闭原始Socket对象。

测试服务器

使用Microsoft Telnet程序能够非常方便地测试服务器代码（Telnet在所有的Windows平台上都有）。Windows 98、Windows Me和Windows NT上的Telnet版本基于Windows的图形系统，Windows 2000和Windows XP中的版本则基于文本，采用命令行方式进行操作。使用Telnet程序，能够通过TCP与任意地址和任意端口进行连接。

要启动上述例子中的TCP服务器，首先打开一个命令行窗口，键入SimpleTcpSrvr，服务器将显示问候语并等待接入客户机：

```
C:\>SimpleTcpSrvr
Waiting for a client...
```

服务器运行起来后，打开另一个命令行窗口（要么是同一个系统中的另一个命令行窗口，要么是网络中另外系统中的命令行窗口），启动Telnet与服务器地址进行连接，使用9050端口号：

```
C:\>telnet 127.0.0.1 9050
```

连接将启动，服务器显示欢迎屏幕。

从现在开始，服务器开始等待客户机的消息。由于使用Telnet，所以发生了一件非常奇怪的事情：我们试图键入一条消息，由于每一个字符都是单独发送到服务器的，所以服务器逐个进行回话。这种状况是由于Telnet应用向远程服务器传送数据的方式而引起的：Telnet自动将每一个字符发送到远程服务器中，不等待回车或换行才发送数据。

监视服务器

如果服务器和客户机位于不同的机器上，可以使用Windump或Analyzer来监视由程序所产生的网络流量。不幸的是，如果服务器和Telnet使用同一系统进行测试，则无法使用上述两个工具查看网络流量。甚至如果套接字与外部IP地址进行了绑定，Windows系统在内部处理流量，而不使用网卡，因此，WinPcap驱动程序不能获取流量信息。图5.1所示是Analyzer输出的例子，使用了程序清单5.1的简单服务器和位于不同的网络系统中的Telnet。

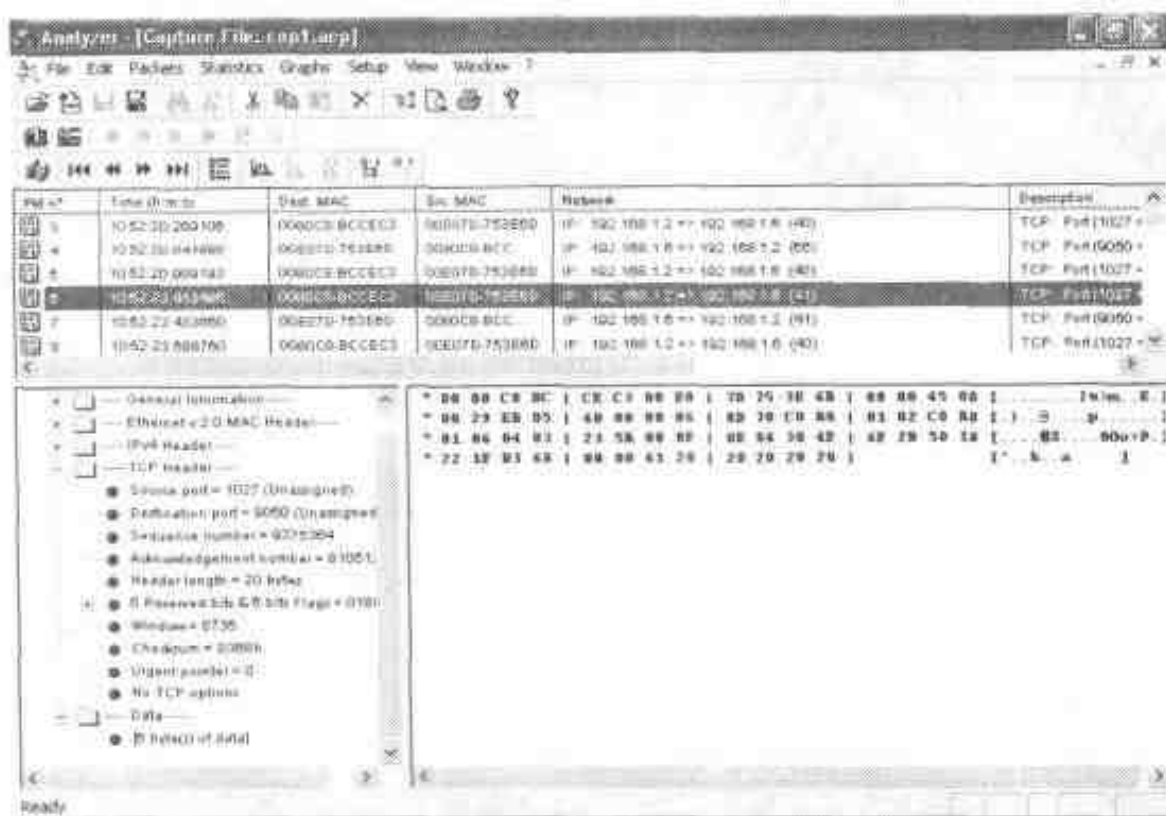


图5.1 来自服务器和远程客户机测试中的Analyzer输出

这一个例子就足够说明问题，在Analyzer输出中，可以看到数据的踪迹情况，Telnet将每一个键入的字符都作为单独的包发送到服务器程序（也就是说，在每一个包里只有一个数据字符）。因为所有的字符都是分离的，因此服务器的Receive()把每一个字符都作为一个单独的消息来处理，并发送反馈消息。

下一节我们将讲述如何创建一个简单的TCP客户机程序，以解决上述问题。

简单TCP客户机

既然我们已经有了一个TCP服务器，因此可以创建一个简单的TCP客户机与它进行通信。在第3章中我们已经讲过，创建与TCP服务器相连的客户机只有两步，很简单：

- 创建一个套接字
- 将套接字与远程服务器地址相连

本节将详细讲述使用C#进行基本TCP客户机编程的步骤。

创建客户机

如同服务器编程一样，进行客户机程序设计的第一步是创建一个Socket对象。Socket.Connect()使用这个Socket对象将所创建的套接字与远程主机相连：

```
IPEndPoint ipep = new IPEndPoint(IPaddress.Parse("192.168.1.6"), 9050);

Socket server = new Socket(AddressFamily.InterNetwork,
    SocketType.Stream, ProtocolType.Tcp);

server.Connect(ipep);
```

这个例子试图将套接字与位于IP地址为192.168.1.6的机器上的服务器相连。当然，也可以与Dns.Resolve()一样使用主机名。读者可能注意到在服务器的编程中，没有使用任何异常

处理方法，在客户机的程序设计中，就不能这样了。对于客户机的Connect()来说，最大的挑战就是如果远程服务器不能使用，将会产生异常情况，对于顾客来说将导致错误消息。解决这个问题较好的方法是在使用Connect()方法时用try-catch块来获取SocketExceptions，从而可以为顾客提供非常友好的提示。

一旦远程的服务器TCP程序接收到连接请求，客户机程序就准备使用标准的Send()和Receive()方法同服务器传送数据。程序清单5.2是SimpleTcpClient.cs程序，在这个程序中对我们刚刚提到的问题进行了示范。

程序清单5.2 SimpleTcpClient.cs程序

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class SimpleTcpClient
{
    public static void Main()
    {
        byte[] data = new byte[1024];
        string input, stringData;
        IPEndPoint ipep = new IPEndPoint(
            IPAddress.Parse("127.0.0.1"), 9050);

        Socket server = new Socket(AddressFamily.InterNetwork,
            SocketType.Stream, ProtocolType.Tcp);

        try
        {
            server.Connect(ipep);
        } catch (SocketException e)
        {
            Console.WriteLine("Unable to connect to server.");
            Console.WriteLine(e.ToString());
            return;
        }

        int recv = server.Receive(data);
        stringData = Encoding.ASCII.GetString(data, 0, recv);
        Console.WriteLine(stringData);

        while(true)
        {
            input = Console.ReadLine();
            if (input == "exit")
                break;
        }
    }
}
```

```

        server.Send(Encoding.ASCII.GetBytes(input));
        data = new byte[1024];
        recv = server.Receive(data);
        stringData = Encoding.ASCII.GetString(data, 0, recv);
        Console.WriteLine(stringData);
    }
    Console.WriteLine("Disconnecting from server...");
    server.Shutdown(SocketShutdown.Both);
    server.Close();
}
}

```

如同服务器端的程序一样，这个客户机程序也仅仅为服务器创建一个**IPEndPoint**（如果要与远程服务器连接，则必须将服务器的IP地址写入**ipep IPEndPoint**值），并试图连接服务器：

```

IPEndPoint ipep = new IPEndPoint(
    IPAddress.Parse("127.0.0.1"), 9050);

Socket server = new Socket(AddressFamily.InterNetwork,
    SocketType.Stream, ProtocolType.Tcp);

try
{
    server.Connect(ipep);
} catch (SocketException e)
{
    Console.WriteLine("Unable to connect to server.");
    Console.WriteLine(e.ToString());
    return;
}

```

try-catch块中的**Connect()**方法用于在远程服务器不能用的时候减缓顾客的不满。一旦连接建立，客户机程序将等待服务器发送来的欢迎消息，并将所收到的消息显示在控制台上。

随后，客户机程序进入一个循环中，从控制台接收文本输入，并将所接收的文本作为独立的消息发给服务器。服务器接收到消息后发送应答给控制台。当输入文本为**exit**时，**while**循环退出，客户机关闭连接。当服务器得知连接关闭（也就是当**Receive()**返回0）时，服务器也退出**while**循环，并关闭连接。

测试客户机

进行客户机测试首先要做的事情是，测试对找不到服务器时的异常进行处理的代码。方法非常简单：不启动服务器程序，而是运行**SimpleTcpClient**程序，产生在异常代码中事先定好的警告消息：

```
C:\>SimpleTcpClient
Unable to connect to server.
System.Net.Sockets.SocketException: Unknown error (0x274d)
   at System.Net.Sockets.Socket.Connect(EndPoint remoteEP)
   at SimpleTcpClient.Main()
```

```
C:\>
```

很显然，在实际的网络编程中，我们会希望能够更细致地处理上述问题，可能会允许顾客再试一次。但是即便是如此简单的几行代码，就比出现`SocketException`时由.NET所生成的标准异常消息友好得多。

现在我们已经解决了对异常问题的测试，下面来试着与服务器进行连接。首先，在指定的机器上启动`SimpleTcpSrvr`程序，一旦该程序指出它正等待客户机的连接，则在同一台机器或者在网络中的另外一台机器上打开一个新命令行窗口，启动`SimpleTcpClient`。当客户机建立了TCP连接时，它应该显示来自服务器的问候语。从这时起，客户机就处于就序状态，等待接收来自控制台的数据，因此我们可以开始输入数据。

在这里我们将注意到从控制台输入的完整句子是作为一条消息发送到服务器并显示，而且是作为一个单条消息返回客户机并显示。因为客户机的`Send()`方法以字节块来发送数据，在服务器端`Receive()`方法以字节块来接收消息。另外，如果从两台不同的机器上运行这个测试，就能够通过WinDump或Analyzer监视这个过程，以验证数据是以字节块的形式而不是以字符来发送的。

虽然这个简单的TCP程序现在运行良好，但是该程序中存在一个严重的缺陷。我们所使用的测试程序，与这些源程序一样都是在一个严格控制的测试环境下使用小数据块，因此绝大多数情况会运行良好。这里之所以说是“绝大多数情况”是因为我们不能绝对保证，在TCP通信过程中会出现很多意外的情况。下一节内容，我们将讲述一些在面向连接的程序中可能发生的故障情况，并指出在设计TCP客户机/服务器程序时如何避免这些故障。

TCP通信故障

到目前为止，本章中我们所学习过的简单的范例程序都假定了通过TCP通道发送和接收数据时的很多情况。然而，还有很多可能的情况我们没有考虑。在面向连接的客户机和服务器程序设计中，有两个问题经常困扰网络程序员的新手：

- 数据缓冲区处理不当
- 网络上的消息处理不当

幸好，上述问题能够通过适当的编程技术来解决。

数据缓冲区故障

在前面提到的简单的客户机和服务器范例中，使用了一个字节数组作为数据缓冲区，在套接字中进行数据的发送和接收。因为程序是在一个受控的环境中运行，而且所有的消息都采用事先设定的格式（文本）和尺寸（小），因此这种类型的缓冲区不会出故障。

然而，在现实世界中，与未知远程服务器或客户机进行通信时，我们可能不知道流入数据的类型或尺寸。如果缓冲区中收到大小不一的消息会发生什么情况？当多于事先定义的数据量到达数据缓冲区时该如何处理？下面我们将要讲述类似这些问题的解决方案。

适当使用数据缓冲区

由于数据是通过TCP来接收的，因此存放在系统的内部缓冲区中。对Receive()方法的每一个调用都试图从TCP缓冲区中取走数据。Receive()方法读取的数据量受下面两个因素的控制：

- 为Receive()方法指定的数据缓冲区的大小
- 为Receive()方法指定的缓冲区大小参数

在SimpleTcpSrvr程序中（参见程序清单5.1），data变量定义为大小1024的字节数组：

```
byte[] data = new byte[1024];
```

在这里使用Receive()方法的一个简单形式来指定存放流入数据的缓冲区。因为没有在Receive()调用时指定数据大小，该缓冲区自动设置为数据缓冲区定义时的默认尺寸（1024个字节）。

Receive()方法试图一次读取1024个字节的数据，并将读取的数据放入data变量中：

```
recv = client.Receive(data);
```

在Receive()方法被调用时，如果TCP缓冲区中的数据小于1024个字节，则该方法返回所有的数据，并将recv变量设置为实际读取的数据量，这是一个非常重要的数值，因为它表示在数据缓冲区中的有效数据量，在用到数据缓冲区中的数据时经常要使用这个变量，否则将发生一些令人吃惊的事情。

例如，下面的一行代码使用data数组中的第一个recv字节创建了一个string对象：

```
stringData = Encoding.ASCII.GetString(data, 0, recv);
```

上述代码是正确调用缓冲区数据的方法。GetString()方法只使用在缓冲区中最新设置的字节的数量。如果在这个Receive()方法调用之前，接收到一个大型数据缓冲区，而且没有使用recv值来创建这个字符串，那么在这个字符串后将追加额外的字节，从而产生错误的数值。

如果在SimpleTcpClient.cs程序中将GetString()方法的最后两个参数删除，向服务器发送大小不同的字符串，则将会出现上述问题。SimpleTcpClient程序的输出类似于下面的代码（为了缩短输出信息，删除了一些空行）：

```
C:\>SimpleTcpClient
Welcome to my test server

test
testome to my test server

a bad test
a bad test my test server
```

```
exit
Disconnecting from server...

C:\>
```

因为在**GetString()**方法中没有定义数据缓冲区的大小，整个数据缓冲区（1024个字节）都用于显示消息，即便多数消息根本不是所接收数据，显然，这不是最佳显示方式。没有指定**recv**变量表示有效数据量，数据缓冲区将产生错误的结果。

小缓冲区的处理

缓冲区太大的对立面是缓冲区太小。如果**Receive()**调用中存放流入数据而使用的数据缓冲区过小会出现一系列的问题。

如图5.2所示，Windows系统中的TCP子系统带有自己的数据缓冲区，用于缓存流入和流出的数据。因为TCP必须能够在任意时候转发数据，因此，这个缓冲区是必需的。一旦数据被确认，则从这个缓冲区中安全地删除。

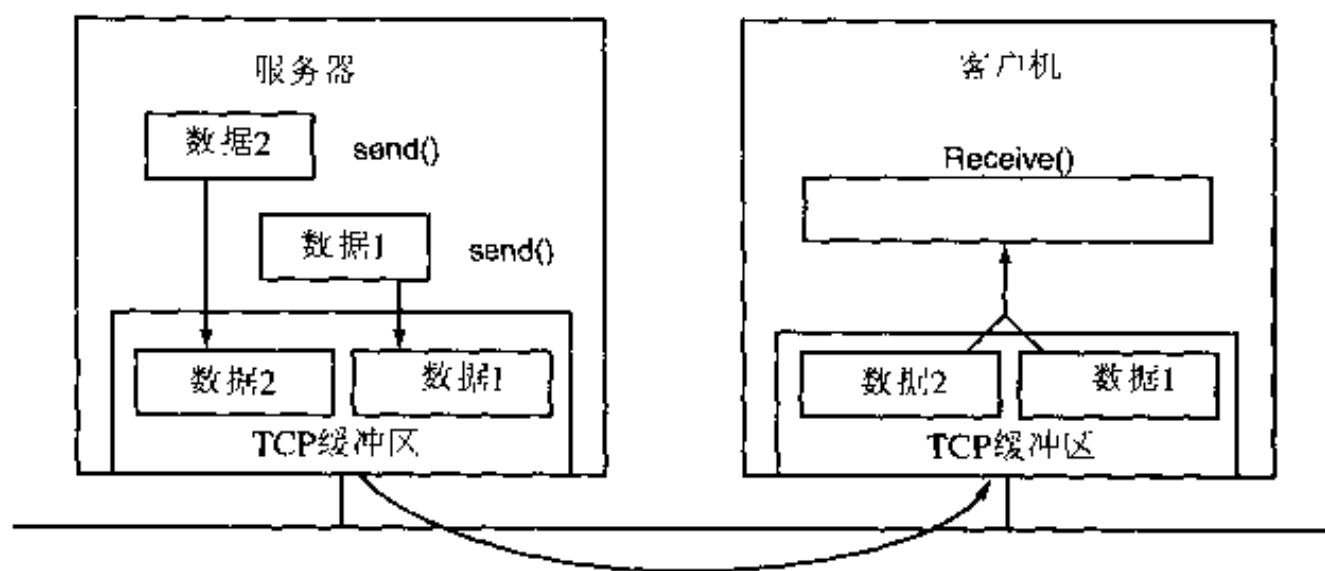


图5.2 TCP缓冲区

流入数据的工作原理也是这样。在**Receive()**方法读取之前，流入数据将一直缓存在缓冲区中。如果缓冲区中所有的数据都没有被这个**Receive()**调用读取，则将等待下一个**Receive()**调用。只要不断有**Receive()**调用从缓冲区中进行数据读取，就不会发生数据丢失的情况，但是这种方法没法读取大块的数据。

下面我们来设置一个环境，以便看看所发生的事情。在SimpleTcpClient.cs程序中将数据缓冲区的尺寸设置为很小，比如20字节：

```
byte data = new byte[20];
```

在重新编译后，运行测试程序，运行结果如下：

```
C:\>SimpleTcpClient
Welcome to my test s
test
erver
another test
test
```

```
oops
another test
exit
Disconnecting from server...

C:\>
```

因为数据缓冲区不够大，在**Receive()**方法中无法装入所有TCP缓冲区数据，仅仅取走与缓冲区大小相同的数据量，而其他的数据则等待下一个**Receive()**调用。因此，获取服务器的欢迎词需要两次**Receive()**调用。在文本传输中，缓冲区中的下一个数据块是从TCP缓冲区中读取的，而且从此以后客户机在数据传输中就位于后台。

鉴于上述原因，在编程时，必须特别留心以确保所有的数据都能够从TCP缓冲区中正确读取。缓冲区太小会导致消息的不匹配，另一方面，缓冲区过大，则导致消息的混合，所带来的麻烦是区分不同的消息，因为数据是从套接字中读取的。下一节我们将要讲述如何处理有关消息的问题，并保证数据正确地处理。

TCP消息

对于网络编程新手来说，最大的障碍就是使用TCP连接中面向消息的通信。关于使用TCP进行网络通信时必须记住的一点是TCP不考虑消息的边界问题。这一节讲述一个非常差劲的TCP程序，以及对其进行修改的方法。

典型的无保护TCP消息边界

对于网络编程新手来说，肯定已经了解了许多有关TCP编程的长处，并使用TCP套接字着手编制在网络上两个设备之间来来回回传递消息的客户机/服务器应用。然而，他们没有意识到像TCP这样的流协议固有的消息边界问题，程序员在客户机上编写了一系列**Send()**方法，在服务器上编写了一系列**Receive()**方法，如图5.3所示。

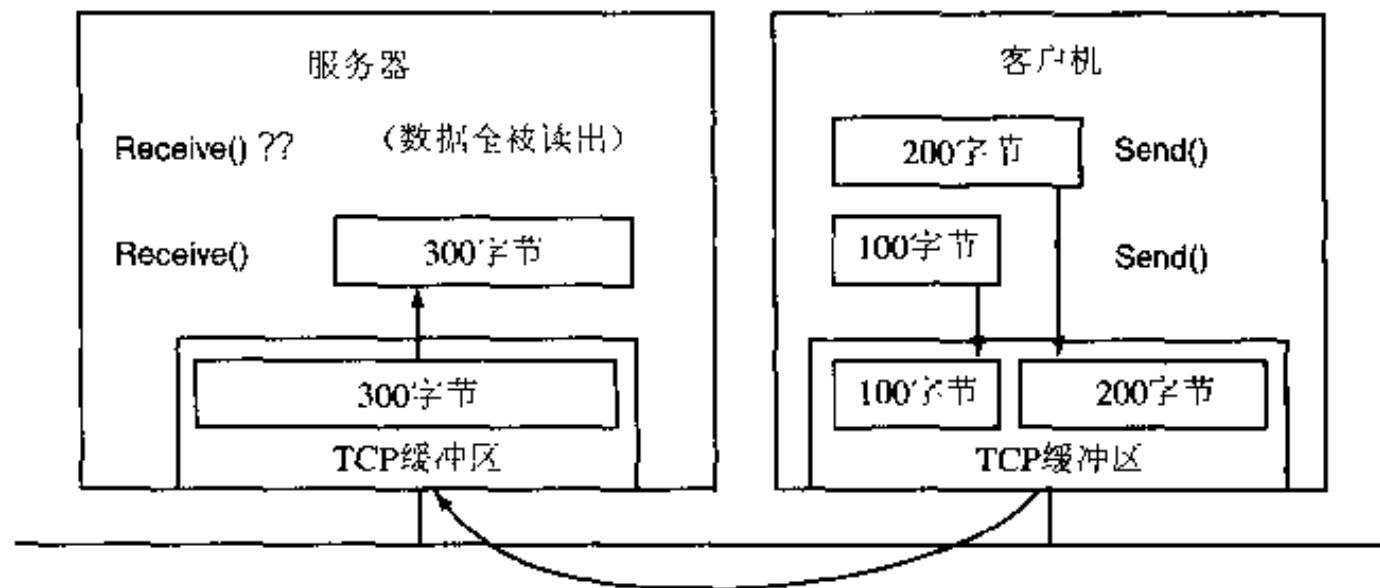


图5.3 不恰当的TCP通信技术

使用这种方式进行通信的最严重的缺陷是不能保证来自每一个单个**Send()**方法的数据能够被单个**Receive()**方法读取。在前面讲述关于缓冲器问题时提到过，**Receive()**方法读取的所有数据实际上不是直接从网络上读取的。而且，**Receive()**方法从系统内部的TCP缓冲区中读

取数据，因为新TCP包从网络中接收，它们被顺序地放入TCP缓冲区中。当调用Receive()方法时，Receive()方法读取该TCP缓冲区中的所有数据，而不仅仅是第一包的数据。

我们可以通过使用一些有缺陷的程序来论述这个问题。首先，看一下程序清单5.3，这是一个非常糟糕的服务器程序。

程序清单5.3 BadTcpSrvr.cs程序

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class BadTcpSrvr
{
    public static void Main()
    {
        int recv;
        byte[] data = new byte[1024];
        IPEndPoint ipep = new IPEndPoint(IPAddress.Any, 9050);
        Socket newsock = new Socket(AddressFamily.InterNetwork,
                                     SocketType.Stream, ProtocolType.Tcp);

        newsock.Bind(ipep);
        newsock.Listen(10);
        Console.WriteLine("Waiting for a client...");

        Socket client = newsock.Accept();
        string welcome = "Welcome to my test server";
        data = Encoding.ASCII.GetBytes(welcome);
        client.Send(data, data.Length,
                   SocketFlags.None);
        IPEndPoint newclient = (IPEndPoint)client.RemoteEndPoint;
        Console.WriteLine("Connected with {0} at port {1}",
                          newclient.Address, newclient.Port);

        for (int i = 0; i < 5; i++)
        {
            recv = client.Receive(data);
            Console.WriteLine(Encoding.ASCII.GetString(data, 0, recv));
        }
        Console.WriteLine("Disconnecting from {0}", newclient.Address);
        client.Close();
        newsock.Close();
    }
}
```

这个服务器程序建立了一个正常的TCP套接字，收听接入连接的情况。当接收到一个连接时，服务器发送一个标准问候语消息，随后，尝试接收来自客户机的五条独立的消息：

```
for (int i = 0; i < 5; i++)
{
    recv = client.Receive(data);
    Console.WriteLine(Encoding.ASCII.GetString(data, 0, recv));
}
```

每一次调用Receive()方法都试图读取TCP缓冲区中所有的数据。在接收到第五条消息后，关闭连接和原始服务器套接字。

下面我们来看看同这个糟糕的服务器程序一起合作的糟糕的客户机程序。程序清单5.4是BadTcpClient.cs，该程序生成一个套接字，试图连接远程服务器（读者要使用自己的服务器设备来替代范例中的IP地址或主机名）。在与远程服务器建立连接后，客户机使用五个独立的Send()调用向服务器程序传送一个文本消息。

程序清单5.4 BadTcpClient.cs程序

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class BadTcpClient
{
    public static void Main()
    {
        byte[] data = new byte[1024];
        string stringData;
        IPEndPoint ipep = new IPEndPoint(
            IPAddress.Parse("127.0.0.1"), 9050);

        Socket server = new Socket(AddressFamily.InterNetwork,
            SocketType.Stream, ProtocolType.Tcp);

        try
        {
            server.Connect(ipep);
        } catch (SocketException e)
        {
            Console.WriteLine("Unable to connect to server.");
            Console.WriteLine(e.ToString());
            return;
        }

        int recv = server.Receive(data);
        stringData = Encoding.ASCII.GetString(data, 0, recv);
```



```
        Console.WriteLine(stringData);

        server.Send(Encoding.ASCII.GetBytes("message 1"));
        server.Send(Encoding.ASCII.GetBytes("message 2"));
        server.Send(Encoding.ASCII.GetBytes("message 3"));
        server.Send(Encoding.ASCII.GetBytes("message 4"));
        server.Send(Encoding.ASCII.GetBytes("message 5"));
        Console.WriteLine("Disconnecting from server...");
        server.Shutdown(SocketShutdown.Both);
        server.Close();
    }
}
```

从不同的客户机和服务器上多次运行这些程序，读者将发现运行结果是不可预测的。很可能服务器上的所有**Receive()**调用读取的是客户机**Send()**方法发送来的某一条消息。笔者曾经尝试过几次，运行结果各不相同。似乎当客户机和服务器位于同一台机器上时，通常能够出现预期的结果。然而，客户机和服务器位于不同的机器上时（读者可以想像在网络中这种情况是正常情况），会发生一些非常有趣的运行结果。

下面来比较两个有代表性的服务器的输出结果：

```
C:\>BadTcpSrvr
Waiting for a client...
Connected with 192.168.1.6 at port 2345
message 1message 2
message 3message 4
message 5

Disconnecting from 192.168.1.6

C:\>BadTcpSrvr
Waiting for a client...
Connected with 192.168.1.6 at port 1065
message 1
message 2message 3message
4message 5

Disconnecting from 192.168.1.6

C:\>
```

在第一次尝试中，第一个服务器的**Receive()**调用读取了来自头两个客户机**Send()**调用中的消息；然后，第二个**Receive()**调用读取接下来两个客户机**Send()**调用中的消息；最后，第三个服务器**Receive()**调用仅仅读取最后一次客户机**Send()**调用中的消息。一旦客户机关闭连接，剩下的服务器**Receive()**调用则没有读取数据，而且服务器正常关闭连接。

第二次尝试就更有意思了。第一个服务器**Receive()**调用读取来自第一个客户机**Send()**调用的完整的一条消息（到目前为止，一切运行良好）；随后，第二个服务器**Receiver()**调用读

取接下来的两个客户机Send()调用的消息，以及第四个Send()调用的一部分消息。这是一个消息在Receive()调用中被砍断的典型例子。最后，第三个服务器Receive()调用读取来自第四个客户机Send()调用余下的部分消息，以及最后一次Send()调用的消息。哈哈，实在是乱成一团糟。

从上述例子中我们总结出两条经验教训：

- 不要仅仅在本地客户机和服务器上测试网络程序并希望它们能够在网络上运转良好。
- 不要假想TCP能够如自己在程序中所设计的那样发送消息。

下一节讲述通过一些编程技巧，使用TCP协议从远程系统中获取数据，并控制这些数据。

TCP消息故障的解决方案

为了解决糟糕的服务器和客户机程序所带来的无防卫的消息边界问题，必须使用一定的技术区分远程系统中的不同消息，常用技巧如下：

- 永远发送固定长度的消息
- 将消息尺寸与消息一起发送
- 使用标记系统分隔消息

上述技术都为远程系统区分从网络上接收到的消息提供了方案，这些技术也有各自的优缺点，本节将介绍这些技术以及它们在C#网络编程中的应用情况。

固定尺寸的消息

最简单但是有可能是最昂贵的解决TCP消息问题的方案是：设计一种协议，永远以固定长度传递消息。通过将所有的消息都设置为相同尺寸，在从远程设备中接收到完整的消息时，TCP接收程序就能毫无疑问地了解情况。

使用固定尺寸消息的另一个优点是当接收到多个消息时，能够根据消息中字节数将它们区分开。采用这种方式，一起接收的消息能够精确分隔。下面将描述如何在Send()和Receive()方法中实现固定尺寸的消息。

发送固定尺寸的消息

发送固定尺寸的消息时，必须保证整个消息是从Send()方法中发送的（不要假设TCP系统发送完整的消息）。在一个简单的Send()方法中，会有如下代码：

```
byte[] data = new byte[1024];  
.  
.  
int sent = socket.Send(data);
```

以上述代码为基础，读者可能会认为整个1024字节的数据缓冲区能够采用这种方式发送到远程设备中，并在自己的程序中使用这种方法。非常遗憾，这种思路是错误的。

根据内部TCP缓冲区的大小和所传送的数据量，很可能不是所有提供给Send()方法的数据都被发送。Send()返回值sent表明实际发送到TCP缓冲区的数据的字节数。然而，是否所有

的数据都发送，这是程序员的事情。如果没有全部发送，程序员需要再尝试一次，重发未发完的消息。通常可以使用一个简单的while()循环语句，不断检查Send()方法的返回值与数据缓冲区的大小。当然，如果尺寸不匹配，就需要采用其他的方法发送余下的数据。

使用上述方法的程序代码如下：

```
int SendData(Socket s, byte[] data)
{
    int total = 0;
    int size = data.Length;
    int dataleft = size;
    int sent;

    while (total < size)
    {
        sent = s.Send(data, total, dataleft, SocketFlags.None);
        total += sent;
        dataleft -= sent;
    }
    return total;
}
```

因为在发送之前就事先知道数据缓冲区的大小，因此可以使用循环语句，直到所有的字节都发送完为止。上述代码段论述了如何通过Send()方法进行循环操作，发送大数据块直到发送成功。每一个新Send()方法调用很显然都必须发送上次调用剩下的信息。当然，很有可能在第一次尝试时所有的数据都发送成功，无疑这是最好的情况。

更复杂一点的Send()方法代码如下：

```
sent = s.Send(data, total, dataleft);
```

第一个参数表示整个数据缓冲区都要发送；第二个参数表明从缓冲区的什么位置开始发送数据；最后一个参数表明要发送的数据的大小。如同一般的Send()方法，这个功能更强大的版本返回发送到远程设备中的全部字节数。

接收固定尺寸的消息

如同发送数据一样，必须保证在一个Receive()调用中接收所有的数据消息。通过在Receive()中使用循环也能够做到这一点。当然，只有在确知来自服务器的数据量时才能这样做。请看下面这段程序：

```
byte[] ReceiveData(Socket s, int size)
{
    int total = 0;
    int dataleft = size;
    byte[] data = new byte[size];
    int recv;

    while (total < size)
```

```
{
    recv = s.Receive(data, total, dataleft, 0);
    if (recv == 0)
    {
        data = Encoding.ASCII.GetBytes("exit ");
        break;
    }
    total += recv;
    dataleft -= recv;
}
return data;
}
```

ReceiveData()方法使用套接字接收数据并设置数据消息的尺寸，使用**Receive()**方法中的循环，读取数据直到预先设置的字节数收到为止。一旦这个过程结束，在消息返回发出调用的应用之前，**ReceiveData()**能够保证接收到完整的数据消息。

测试奇特的Send()和Receive()方法

SendData()和**ReceiveData()**的使用非常简单。在程序清单5.5所示的FixedTcpSrvr.cs程序中，使用这两个新方法，对前面讲述过的那个糟糕的TCP服务器程序重新进行了编写。

程序清单5.5 FixedTcpSrvr.cs程序

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class FixedTcpSrvr
{
    private static int SendData(Socket s, byte[] data)
    {
        int total = 0;
        int size = data.Length;
        int dataleft = size;
        int sent;

        while (total < size)
        {
            sent = s.Send(data, total, dataleft, SocketFlags.None);
            total += sent;
            dataleft -= sent;
        }
        return total;
    }
}
```

```
private static byte[] ReceiveData(Socket s, int size)
{
    int total = 0;
    int dataleft = size;
    byte[] data = new byte[size];
    int recv;

    while(total < size)
    {
        recv = s.Receive(data, total, dataleft, 0);
        if (recv == 0)
        {
            data = Encoding.ASCII.GetBytes("exit");
            break;
        }
        total += recv;
        dataleft -= recv;
    }
    return data;
}

public static void Main()
{
    byte[] data = new byte[1024];
    IPEndPoint ipep = new IPEndPoint(IPAddress.Any, 9050);
    Socket newsock = new Socket(AddressFamily.InterNetwork,
                                SocketType.Stream, ProtocolType.Tcp);

    newsock.Bind(ipep);
    newsock.Listen(10);
    Console.WriteLine("Waiting for a client...");

    Socket client = newsock.Accept();
    IPEndPoint newclient = (IPEndPoint)client.RemoteEndPoint;
    Console.WriteLine("Connected with {0} at port {1}",
                      newclient.Address, newclient.Port);

    string welcome = "Welcome to my test server";
    data = Encoding.ASCII.GetBytes(welcome);
    int sent = SendData(client, data);

    for (int i = 0; i < 5; i++)
    {
        data = ReceiveData(client, 9);
        Console.WriteLine(Encoding.ASCII.GetString(data));
    }
    Console.WriteLine("Disconnected from {0}", newclient.Address);
}
```

```
        client.Close();  
        newsock.Close();  
    }  
}
```

服务器（采用固定尺寸）现在使用**SendData()**和**ReceiveData()**方法以保证发送和接收的数据的字节数。从客户机中接收5个数据包的程序段现在能够保证每个接收到的消息都只有9个字节。

```
for (int i = 0; i < 5; i++)  
{  
    data = ReceiveData(client, 9);  
    Console.WriteLine(Encoding.ASCII.GetString(data));  
}
```

因为服务器希望每一个TCP消息都恰好是9个字节，因此需要设计一个只能发送9个字节大小的消息的客户机。程序清单5.6中的**FixedTcpClient.cs**程序能够完成这项功能。

程序清单5.6 FixedTcpClient.cs程序

```
using System;  
using System.Net;  
using System.Net.Sockets;  
using System.Text;  
  
class FixedTcpClient  
{  
    private static int SendData(Socket s, byte[] data)  
    {  
        int total = 0;  
        int size = data.Length;  
        int dataleft = size;  
        int sent;  
  
        while (total < size)  
        {  
            sent = s.Send(data, total, dataleft, SocketFlags.None);  
            total += sent;  
            dataleft -= sent;  
        }  
        return total;  
    }  
  
    private static byte[] ReceiveData(Socket s, int size)  
    {  
        int total = 0;  
        int dataleft = size;  
        byte[] data = new byte[size];  

```



```
int recv;
while(total < size)
{
    recv = s.Receive(data, total, dataleft, 0);
    if (recv == 0)
    {
        data = Encoding.ASCII.GetBytes("exit ");
        break;
    }
    total += recv;
    dataleft -= recv;
}
return data;
}

public static void Main()
{
    byte[] data = new byte[1024];
    int sent;
    IPEndPoint ipep = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 9050);
    Socket server = new Socket(AddressFamily.InterNetwork,
                               SocketType.Stream, ProtocolType.Tcp);

    try
    {
        server.Connect(ipep);
    } catch (SocketException e)
    {
        Console.WriteLine("Unable to connect to server.");
        Console.WriteLine(e.ToString());
        return;
    }

    int recv = server.Receive(data);
    string stringData = Encoding.ASCII.GetString(data, 0, recv);
    Console.WriteLine(stringData);

    sent = SendData(server, Encoding.ASCII.GetBytes('message 1'));
    sent = SendData(server, Encoding.ASCII.GetBytes('message 2'));
    sent = SendData(server, Encoding.ASCII.GetBytes('message 3'));
    sent = SendData(server, Encoding.ASCII.GetBytes('message 4'));
    sent = SendData(server, Encoding.ASCII.GetBytes('message 5'));
    Console.WriteLine("Disconnecting from server...");
    server.Shutdown(SocketShutdown.Both);
    server.Close();
}
}
```

使用这个新版本的客户机，每一个被发往服务器的消息永远是9个字节：

```
sent = SendData(server, Encoding.ASCII.GetBytes("message 1"), 9);  
sent = SendData(server, Encoding.ASCII.GetBytes("message 2"), 9);  
sent = SendData(server, Encoding.ASCII.GetBytes("message 3"), 9);  
sent = SendData(server, Encoding.ASCII.GetBytes("message 4"), 9);  
sent = SendData(server, Encoding.ASCII.GetBytes("message 5"), 9);
```

说明：可以使用位于网络上的不同的客户机并使用不同大小的固定尺寸消息对FixedTcpSrvr和FixedTcpClient进行测试（虽然每一个程序只能使用一个设好的消息尺寸）。使用这些新版本的程序，笔者多次试验没有再出现以前的问题，甚至在网络上不同的机器上运行也一样：客户机上每一个发自SendData()的消息，都能被服务器上的ReceiveData()方法正确接收。

使用消息尺寸信息

通过前面的讨论，我们了解到TCP通信存在消息边界的问题，使用固定长度的消息能够解决这个问题，但是这种方法缺乏灵活性，必须保证所有的消息都恰好是同一长度，这就意味着必须将短消息加长，这将造成网络带宽资源的浪费。

对这个问题最简单的解决方案是允许使用可变长度的消息，惟一的不足就是接收端的远程设备必须了解每一个变长消息的确切尺寸。具体方法就是在发送消息的时候，一起发送一个消息尺寸指示器，表明该消息的尺寸。那么如何在消息里加入一个消息尺寸指示器就成了眼下要解决的问题。

在消息包里包含消息大小的方法非常多，最简单的方法就是创建一个文本来表示消息的大小，并把它追加在消息前面。例如，如程序清单5.6所示，message 1是9个字节，因此在这个消息之前追加文本“9”，这样一来所传递的消息就是下面这个样子：

```
9message 1
```

位于消息前面的9表明实际发送的消息的确切大小。接收端的设备读取每一个消息的第一个字节后马上就能知道整个消息的具体尺寸。

当然这种解决方案也有不足之处，但这不是什么坏事。采用这种方案的时候，客户机和服务器程序都必须了解用于表示消息的尺寸信息的字节数。显然，对于大型数据包，仅用一个字符肯定不够，而是用3个到4个字符来表示尺寸信息，对于较小的数据包来说这又是一种资源浪费。

如何解决这个问题呢？除了使用文本来表示消息尺寸外，我们还可以直接使用消息尺寸实际的整型数值。这就需要将整型数转换为字节数组。接下来的内容会对如何使用这种技术进行描述。

发送变长消息

SendVarData()方法是由SendData()方法改进而来，在这个方法中，消息的尺寸信息用一个4个字节的整型数来表示，附加在每一个消息的开始位置。使用4个字节的整型数值，只要是小于65KB的消息都能使用相同的编码方式来表示大小，请看下面的程序代码：


```

private static int SendVarData(Socket s, byte[] data)
{
    int total = 0;
    int size = data.Length;
    int dataleft = size;
    int sent;

    byte[] datasize = new byte[4];
    datasize = BitConverter.GetBytes(size);
    sent = s.Send(datasize);

    while (total < size)
    {
        sent = s.Send(data, total, dataleft, SocketFlags.None);
        total += sent;
        dataleft -= sent;
    }
    return total;
}

```

SendVarData()方法首先使用System命名空间中的BitConverter类将表示数据大小的整型数值转换为字节数组中的4个字节，**GetBytes()**把这个32位整型数值转换为4个字节的数组，然后数组将发往远程系统。消息尺寸信息发送完毕后，**SendVarData()**方法采用标准的循环方式发送消息体，直到发送完毕为止。

接收变长消息

下一步就是创建一个方法来解析接收到的消息格式（包含消息大小信息和整个消息体）。**ReceiveData()**方法必须做相应的改进，以便从套接字中接受4个字节的数，并将它们转换为表示消息尺寸的整型数值。判断消息的尺寸之后，使用前面**ReceiveData()**方法中标准的方法将整个消息体读入数据缓冲区。改进后的**ReceiveVarData()**如下：

```

private static byte[] ReceiveVarData(Socket s)
{
    int total = 0;
    int recv;
    byte[] datasize = new byte[4];

    recv = s.Receive(datasize, 0, 4, 0);
    int size = BitConverter.ToInt32(datasize);
    int dataleft = size;
    byte[] data = new byte[size];

    while (total < size)
    {
        recv = s.Receive(data, total, dataleft, 0);
        if (recv == 0)

```

```
        {
            data = Encoding.ASCII.GetBytes("exit ");
            break;
        }
        total += recv;
        dataleft -= recv;
    }
    return data;
}
```

ReceiveVarData()读取消息的前4个字节信息，使用**BitConverter**类的**GetInt32()**方法将它们转换为表示消息大小信息的整型数值：

```
recv = s.Receive(datasize, 0, 4, 0);
int size = BitConverter.ToInt32(datasize);
```

在这里必须使用**Receive()**中的标准技术，以保证从缓冲区中只能读取消息的前4个字节。接下来，执行正常的**Receive()**方法中的循环，从套接字中读取数据直到整个消息体读出为止。

测试新方法

现在我们已经讲述了改进后的**SendVarData()**和**ReceiveVarData()**，因此，就可以使用这两个新方法对前面讲述过的那个糟糕的**TCP**程序进行改进。使用这种新方法发送和接收变长消息，可以设计发送大型文本消息而不必担心消息重叠等问题的**TCP**消息应用。程序清单5.7就是这样一个**TCP**服务器程序的例子，在这个例子中使用了**SendVarData()**和**ReceiveVarData()**方法。

程序清单5.7 VarTcpSrvr.cs程序

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class VarTcpSrvr
{
    private static int SendData(Socket s, byte[] data)
    {
        int total = 0;
        int size = data.Length;
        int dataleft = size;
        int sent;

        byte[] datasize = new byte[4];
        datasize = BitConverter.GetBytes(size);
        sent = s.Send(datasize);
```

```
while (total < size)
{
    sent = s.Send(data, total, dataleft, SocketFlags.None);
    total += sent;
    dataleft -= sent;
}
return total;
}

private static byte[] ReceiveVarData(Socket s)
{
    int total = 0;
    int recv;
    byte[] datasize = new byte[4];

    recv = s.Receive(datasize, 0, 4, 0);
    int size = BitConverter.ToInt32(datasize, 0);
    int dataleft = size;
    byte[] data = new byte[size];

    while(total < size)
    {
        recv = s.Receive(data, total, dataleft, 0);
        if (recv == 0)
        {
            data = Encoding.ASCII.GetBytes("exit ");
            break;
        }
        total += recv;
        dataleft -= recv;
    }
    return data;
}

public static void Main()
{
    byte[] data = new byte[1024];
    IPEndPoint ipep = new IPEndPoint(IPAddress.Any, 9050);

    Socket newsock = new Socket(AddressFamily.InterNetwork,
                                SocketType.Stream, ProtocolType.Tcp);

    newsock.Bind(ipep);
    newsock.Listen(10);
    Console.WriteLine("Waiting for a client...");

    Socket client = newsock.Accept();
    IPEndPoint newclient = (IPEndPoint)client.RemoteEndPoint;
```

```

        Console.WriteLine("Connected with {0} at port {1}",
            newclient.Address, newclient.Port);

        string welcome = "Welcome to my test server";
        data = Encoding.ASCII.GetBytes(welcome);
        int sent = SendVarData(client, data);

        for (int i = 0; i < 5; i++)
        {
            data = ReceiveVarData(client);
            Console.WriteLine(Encoding.ASCII.GetString(data));
        }
        Console.WriteLine("Disconnected from {0}", newclient.Address);
        client.Close();
        newsock.Close();
    }
}

```

这个程序的功能与前面在程序清单5.3中讲述过的原始的BadTcpSrvr.cs一样：当客户机连接上时发送一个标语消息，然后在关闭连接之前，试图接收来自客户机的五个独立的消息。当然，二者之间的不同是新应用接收到的消息采用新格式，在每一个消息的前面加上了4个字节的消息尺寸信息。

因为采用了新消息格式，因此不能使用任意TCP客户机与这个服务器进行通信，必须使用特殊的客户机程序，能够发送和服务器所接收的消息一样格式的消息。程序清单5.8列举了这样一个类似的TCP客户机的例子，这个客户机程序使用新SendVarData()方法来产生新格式的消息。另外，如果要在网络中测试这个程序，切记使用服务器机器的合适的IP地址来替换回退IP地址。

程序清单5.8 VarTcpClient.cs程序

```

using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class VarTcpClient
{
    private static int SendVarData(Socket s, byte[] data)
    {
        int total = 0;
        int size = data.Length;
        int dataleft = size;
        int sent;

        byte[] datasize = new byte[4];
    }
}

```

```
datasize = BitConverter.GetBytes(size);
sent = s.Send(datasize);

while (total < size)
{
    sent = s.Send(data, total, dataleft, SocketFlags.None);
    total += sent;
    dataleft -= sent;
}
return total;
}

private static byte[] ReceiveVarData(Socket s)
{
    int total = 0;
    int recv;
    byte[] datasize = new byte[4];

    recv = s.Receive(datasize, 0, 4, 0);
    int size = BitConverter.ToInt32(datasize, 0);
    int dataleft = size;
    byte[] data = new byte[size];

    while (total < size)
    {
        recv = s.Receive(data, total, dataleft, 0);
        if (recv == 0)
        {
            data = Encoding.ASCII.GetBytes("exit ");
            break;
        }
        total += recv;
        dataleft -= recv;
    }
    return data;
}

public static void Main()
{
    byte[] data = new byte[1024];
    int sent;
    IPEndPoint ipep = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 9050);

    Socket server = new Socket(AddressFamily.InterNetwork,
                               SocketType.Stream, ProtocolType.Tcp);

    try
    {

```

```

        server.Connect(ip);
    } catch (SocketException e)
    {
        Console.WriteLine("Unable to connect to server.");
        Console.WriteLine(e.ToString());
        return;
    }

    data = ReceiveVarData(server);
    string stringData = Encoding.ASCII.GetString(data);
    Console.WriteLine(stringData);

    string message1 = "This is the first test";
    string message2 = "A short test";
    string message3 = "This string is an even longer test. The quick brown
        fox jumps over the lazy dog.";
    string message4 = "a";
    string message5 = "The last test";
    sent = SendVarData(server, Encoding.ASCII.GetBytes(message1));
    sent = SendVarData(server, Encoding.ASCII.GetBytes(message2));
    sent = SendVarData(server, Encoding.ASCII.GetBytes(message3));
    sent = SendVarData(server, Encoding.ASCII.GetBytes(message4));
    sent = SendVarData(server, Encoding.ASCII.GetBytes(message5));
    Console.WriteLine("Disconnecting from server...");
    server.Shutdown(SocketShutdown.Both);
    server.Close();
}
}

```

我们可以在同一台机器上不同的命令行窗口下启动VarTcpSrvr，对新客户机和服务器程序进行测试，也可以在网络中不同的机器上进行测试。服务器程序的输出如下所示：

```

C:\>VarTcpSrvr
Waiting for a client...
Connected with 127.0.0.1 at port 1044
This is the first test
A short test
This string is an even longer test. The quick brown fox jumps over the lazy dog.
a
The last test
Disconnected from 127.0.0.1

C:\>

```

到目前为止，我们已经设计了一个完整的TCP客户机和服务器程序，能够在客户机和服务器之间来回发送和接收文本消息。

警告：在使用BitConverter()时，整型数到字节数组的转换是根据本地机器的字节顺序进行的，只要是使用Intel芯片、Windows操作系统的机器，都没有什么问题。如果使用的是其他系统，采用不同的字节顺序，麻烦就会来临。在本书第7章“C#套接字的助手类”将要探讨这个问题，并提出解决方案。

使用消息标记

在发送TCP消息时，还有另外一种方法能够对消息进行控制，这种方法就是消息标记系统。这个系统使用预先确定的一个字符（或多个字符）来指定消息的结束，通过这种方式分隔不同的消息。因为消息是从套接字中接收的，挨个字符对数据进行检查，当检查到一个标记时，就认为是一个完整的消息，并将这个消息发送到应用中。标记后面的数据是一个新消息的开始。

使用消息标记的缺点之一就是：所接收到的数据流中的每一个字符必须都要检查以确定是否为标记。对于大型消息来说，这可能导致系统性能的下降。而且，某一些字符必须设计为标记，这些字符要与普通字符相互区分（否则事情就麻烦了）。

除了设计自己专用的消息标记系统外，C#语言提供了一些类，能够用于简化这个过程。下一节内容将描述如何使用System.IO命名空间流类，在TCP连接中方便地读写文本信息。

在TCP通信中使用C#流

对于程序员来说，在TCP连接中消息的处理是非常重要而且也是富有挑战性的问题，因此.NET框架提供了一些特殊的类来简化这项工作。这一节内容我们将要讲述NetworkStream类，在这个类中提供了一个套接字的流接口，以及两个其他的流类：StreamReader和StreamWriter，这两个类用于使用TCP协议发送和接收文本消息。

NetworkStream类

在本书第1章里，我们曾经讲述过，C#使用流技术来传递大数据块。System.Net.Sockets命名空间里包含NetworkStream类，这个类就为套接字提供了一个流的接口。

有很多构造函数能够创建NetworkStream对象，其中最简单（而且是最流行的）方式是使用Socket对象，请看下面的代码：

```
Socket newsock = new Socket(AddressFamily.InterNetwork,
                             SocketType.Stream, ProtocolType.Tcp);
NetworkStream ns = new NetworkStream(newsock);
```

这就创建了一个新的NetworkStream流对象，我们在以后的编程中可以使用这个流对象而不再使用Socket对象。在NetworkStream对象创建后，Socket对象可以使用许多属性和方法，如表5.1所示。

属性中最常用的是DataAvailable，通过这个属性，我们可以迅速查看在套接字的缓冲区里是否有数据等待读出。

表5.1 NetworkStream类的属性

属性	说明
CanRead	NetworkStream支持读操作时该属性值为真
CanSeek	对于NetworkStream对象来说，该属性值永远为假
CanWrite	NetworkStream支持写操作时该属性值为真
DataAvailable	有数据可读时该属性值为真

NetworkStream类中含有许多在流中存取数据的方法，这些方法都在表5.2中列出。

表5.2 NetworkStream类的方法

方法	说明
BeginRead()	启动一个异步NetworkStream读操作
BeginWrite()	启动一个异步NetworkStream写操作
Close()	关闭NetworkStream对象
CreateObjRef()	为NetworkStream创建一个代理
EndRead()	结束一个异步NetworkStream读操作
EndWrite()	结束一个异步NetworkStream写操作
Equals()	判断两个NetworkStream是否相同
Flush()	从NetworkStream中取走所有的数据
GetHashCode()	获取NetworkStream的一个哈希码
GetLifetimeService()	获取NetworkStream生存期对象
GetType()	获取NetworkStream类型
InitializeLifetimeService()	为NetworkStream初始化一个生命周期内的服务对象，来控制生存期
Read()	从NetworkStream中读取数据
ReadByte()	从NetworkStream中读取一个字节的的数据
ToString()	返回字符串
Write()	向NetworkStream写数据
WriteByte()	向NetworkStream写一个字节的的数据

使用Read()方法从NetworkStream中读数据块，采用的格式如下面代码所示，其中buffer是存放所读取数据的字节数组缓冲区，offset是缓冲区位置（从这个位置开始存放数据），size是要读取数据的字节数：


```
int Read(byte[] buffer, int offset, int size)
```

Read()方法返回一个整型数值，表示实际上从**NetworkStream**中读取的字节数，该整型数被放入缓冲区中。

和类似，**Write()**方法的格式如下面代码所示，其中从**buffer**中读取数据发送，**offset**是缓冲区中的一个地址（从这个位置开始读取数据），**size**是要发送的字节数：

```
void Write(byte[] buffer, int offset, int size)
```

程序清单5.9所示为另一个版本的TCP客户端程序，该程序使用**NetworkStream**对象进行通信。

程序清单5.9 NetworkStreamTcpClient.cs程序

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class NetworkStreamTcpClient
{
    public static void Main()
    {
        byte[] data = new byte[1024];
        string input, stringData;
        int recv;
        IPEndPoint ipep = new IPEndPoint(
            IPAddress.Parse("127.0.0.1"), 9050);

        Socket server = new Socket(AddressFamily.InterNetwork,
            SocketType.Stream, ProtocolType.Tcp);

        try
        {
            server.Connect(ipep);
        } catch (SocketException e)
        {
            Console.WriteLine("Unable to connect to server.");
            Console.WriteLine(e.ToString());
            return;
        }

        NetworkStream ns = new NetworkStream(server);

        if (ns.CanRead)
        {
            recv = ns.Read(data, 0, data.Length);
            stringData = Encoding.ASCII.GetString(data, 0, recv);
            Console.WriteLine(stringData);
        }
    }
}
```

```
    }
    else
    {
        Console.WriteLine("Error: Can't read from this socket");
        ns.Close();
        server.Close();
        return;
    }

    while(true)
    {
        input = Console.ReadLine();
        if (input == "exit")
            break;
        if (ns.CanWrite)
        {
            ns.Write(Encoding.ASCII.GetBytes(input), 0, input.Length);
            ns.Flush();
        }

        recv = ns.Read(data, 0, data.Length);
        stringData = Encoding.ASCII.GetString(data, 0, recv);
        Console.WriteLine(stringData);
    }
    Console.WriteLine("Disconnecting from server...");
    ns.Close();
    server.Shutdown(SocketShutdown.Both);
    server.Close();
}
}
```

这个程序从Socket对象中生成了一个NetworkStream对象:

```
NetworkStream ns = new NetworkStream(server);
```

一旦NetworkStream创建, Socket对象就不再使用, 直到在程序结束时被关闭。所有与远程服务器的通信都通过这个NetworkStream来完成:

```
recv = ns.Read(data, 0, data.Length);

ns.Write(Encoding.ASCII.GetBytes(input), 0, input.Length);
ns.Flush();
```

在每一个Write()方法后使用Flush()方法, 以保证NetworkStream中的数据能够被立即发送到远程系统, 而不是在发送前在TCP缓冲区中等待其他数据的到来。

因为NetworkStreamTcpClient发送和接收同一个数据包, 因此可以使用本章一开始讲述过的SimpleTcpSrvr程序对其进行测试(见程序清单5.1)。NetworkStreamTcpClient工作方式

与SimpleTcpClient一样，发送消息到服务器，接收来自服务器的应答消息。

比之Socket对象，虽然NetworkStream对象有一些另外的功能，但从套接字中发送和接收数据时仍然受到限制。与使用一般的Socket对象发送和接收数据一样，同样存在消息边界的问题。接下来的内容将要讲述两个类，通过这两个类可以对套接字中的数据进行更好的控制。

StreamReader和StreamWriter

System.IO命名空间中含有两个类：StreamReader和StreamWriter，这两个类控制一个数据流上对文本消息的读写操作。这两个类都可以使用NetworkStream对象进行部署以便定义TCP消息的标记。

StreamReader类为许多应用提供多个构造函数，使用NetworkStream是最简单格式，请看下面的代码：

```
public StreamReader(Stream stream)
```

变量stream可以使用任意类型的Stream对象，包括NetworkStream对象。在StreamReader创建后，我们有很多属性和方法可以应用，如表5.3所示。

表5.3 StreamReader类方法

方法	说明
Close()	关闭StreamReader对象
CreateObjRef()	创建StreamReader的代理
DiscardBufferedData()	丢弃StreamReader中当前的数据
Equals()	判断两个StreamReader是否相同
GetHashCode()	返回StreamReader对象的哈希码
GetLifetimeService()	获取StreamReader的生存期服务对象
GetType()	获取StreamReader对象类型
InitializeLifetimeService()	为StreamReader创建一个生命期服务对象
Peek()	返回数据中下一个可用字节，该字节依然留在流中
Read()	从StreamReader中读取一个或者多个字节数据
ReadBlock()	从StreamReader流中读取一组字节，并将所读取的数据放入指定的缓冲区位置
ReadLine()	从StreamReader对象中读取数据，直到第一个换行符（包括该换行符）为止
ReadToEnd()	读取数据直到数据流结束
ToString()	创建StreamReader对象的字符串表示


```
newsock.Bind(ipep);
newsock.Listen(10);
Console.WriteLine("Waiting for a client...");

Socket client = newsock.Accept();
IPEndPoint newclient = (IPEndPoint)client.RemoteEndPoint;
Console.WriteLine("Connected with {0} at port {1}",
    newclient.Address, newclient.Port);

NetworkStream ns = new NetworkStream(client);
StreamReader sr = new StreamReader(ns);
StreamWriter sw = new StreamWriter(ns);

string welcome = "Welcome to my test server";
sw.WriteLine(welcome);
sw.Flush();

while(true)
{
    try
    {
        data = sr.ReadLine();
    } catch (IOException)
    {
        break;
    }

    Console.WriteLine(data);
    sw.WriteLine(data);
    sw.Flush();
}
Console.WriteLine("Disconnected from {0}", newclient.Address);
sw.Close();
sr.Close();
ns.Close();
}
}
```

StreamTcpSrvr程序使用**StreamWriter** **WriteLine()**方法发送以换行符结束的文本消息。对**NetworkStream**对象来说，在每一次**WriteLine()**调用后最好使用**Flush()**方法确保TCP缓冲区中所有的数据都被发送。

读者可能注意到，在这个方法和原始的**SimpleTcpSrvr**程序之间有一个主要的不同：即**StreamTcpSrvr**如何了解远程连接是否断开的方式。因为**ReadLine()**方法是对数据流进行操作，而不是对套接字进行操作。在远程连接关闭的时候该方法不返回0，相反，如果没有了底层的套接字，该方法将生成一个异常故障。因此，在进行网络编程的时候，必须考虑这个问题，即在套接字关闭的时候要进行异常处理。请看下面的代码：

```
try
{
    data = sr.ReadLine();
} catch (IOException)
{
    break;
}
```

流客户机

现在我们已经设计了一个服务器能够接收以换行符结尾的消息。下面我们将设计一个客户机，以服务器所能接收的消息格式发送消息。程序清单5.11中的StreamTcpClient程序，就是这样一个在TCP客户机应用中使用StreamReader和StreamWriter对象的例子。

程序清单5.11 StreamTcpClient.cs程序

```
using System;
using System.IO;
using System.Net;
using System.Net.Sockets;
using System.Text;

class StreamTcpClient
{
    public static void Main()
    {
        string data;
        string input;
        IPEndPoint ipep = new IPEndPoint(
            IPAddress.Parse("127.0.0.1"), 9050);

        Socket server = new Socket(AddressFamily.InterNetwork,
            SocketType.Stream, ProtocolType.Tcp);

        try
        {
            server.Connect(ipep);
        } catch (SocketException e)
        {
            Console.WriteLine("Unable to connect to server.");
            Console.WriteLine(e.ToString());
            return;
        }

        NetworkStream ns = new NetworkStream(server);
        StreamReader sr = new StreamReader(ns);
        StreamWriter sw = new StreamWriter(ns);
```



```
        data = sr.ReadLine();
        Console.WriteLine(data);

        while(true)
        {
            input = Console.ReadLine();
            if (input == "exit")
                break;
            sw.WriteLine(input);
            sw.Flush();

            data = sr.ReadLine();
            Console.WriteLine(data);
        }
        Console.WriteLine("Disconnecting from server...");
        sr.Close();
        sw.Close();
        ns.Close();
        server.Shutdown(SocketShutdown.Both);
        server.Close();
    }
}
```

这个客户机程序与前面我们讲述过的SimpleTcpClient功能相同，但是它发送的消息以换行符号结束。这是一个非常重要的特征，读者一定要深刻理解。如果在原来的SimpleTcpSrvr程序中使用StreamTcpClient，则程序无法执行。这是因为由SimpleTcpSrvr所生成的问候语不是以换行符号结尾，由于没有换行符，客户机中的ReadLine()方法无法结束，将堵塞程序的执行，一直等待换行符。无论什么时候，读者都要切记这个问题。

警告： 在使用ReadLine()方法时要记住的另一件事情是保证数据本身不带有换行符，如果数据本身有换行符，则这些换行符将被ReadLine()方法错误地作为标记，影响数据的读写。

小结

本章主要描述了如何使用TCP套接字发送消息。因为TCP套接字是面向连接的，在数据传输之前，发送和接收的双方都必须建立好连接。数据以数据流的形式发送到远程机器上，因此在TCP会话中消息是没有边界的。

在这种使用TCP协议的环境中，不能保护消息的边界，因此，程序员在编写网络层的TCP应用时，必须考虑如何识别单个消息。有多种方法可以实现消息的识别：一种方法是等停，即发送一条消息后等待接收方的应答后再发送下一条消息，另一种方法是使用固定长度的消息，第三种方法是在消息内加入消息的尺寸信息，也可以采用对所发送的消息以特殊预定符号结尾的方式。上述所有的方法都有各自的优缺点，以及所带来的一些问题，本章对这些问题均提出了解决方案。

为了简化使用TCP协议传输文本消息的过程，C#库带有能够使用换行符号作为文本消息分隔符的流方法。在StreamReader和StreamWriter类中使用了ReadLine()和WriteLine()方法，简化在套接字连接之间文本消息的发送。

下一章将要对使用C# Socket类进行无连接的网络编程，对于读者来说将是一个全新的话题。

第6章 无连接套接字

- 无连接通信UDP应用基础
- UDP客户机程序中Connect()方法的应用
- 优化UDP消息的“包装”
- 使用较小的缓冲区以预防消息的丢失
- 使用套接字超时机制避免丢包
- 消息包重传机制
- 一个完整、具体而实用的UDP应用范例

在前面一章中，我们讲述了如何使用TCP协议和C# Socket类设计面向连接的套接字，本章我们将换个角度，看看无连接通信的情况。本章的主要内容是讲述使用UDP协议和C# Socket类设计无连接应用的基本原理和过程。

使用无连接的套接字，我们能够在自我包含的数据包里发送消息，采用独立的读函数读取整个消息，所读取的消息是使用独立的发送函数发送的。这就避免了在数据包内进行消息边界的匹配问题。遗憾的是，UDP数据包不能保证可靠传输，存在许多因素，比如网络繁忙等，都有可能阻止数据包如期到达目的地。

本章首先来看一个简单的UDP服务器和客户机应用，在这个应用中示范了UDP消息的工作原理。接着将讨论对丢失的UDP数据包和不同的UDP缓冲区尺寸的处理。最后，将所有的UDP技术进行融合，并举例说明。

简单的UDP应用

正如第3章中“C#网络编程的类”中所描述过的，UDP是一种无连接的协议。因此，为了使服务器应用能够发送和接收UDP数据包，程序员必须处理两件事情：

- 创建一个Socket对象
- 将创建的套接字对象与本地IPEndPoint进行绑定

完成上述两个步骤后，所创建的套接字就能够在IPEndPoint上接收流入的UDP数据包，或者将流出的UDP数据包发送到网络中任意其他设备上。使用UDP进行通信时，不需要TCP连接。

说明：对于不需要在指定的UDP端口上接收UDP数据包的客户机UDP应用，不需要执行上述的第二个步骤，即不需要将所创建的套接字与指定的IPEndPoint进行绑定，只需要创建套接字并发送数据即可。

因为在异地主机之间没有连接，UDP应用不能使用标准的Send()和Receive() Socket方法，与之相对应，在UDP通信中使用两个新方法：SendTo()和ReceiveFrom()。

SendTo() 该方法指定要发送的数据，以及目标机器的IPEndPoint。该方法有多种使用方式，可以根据具体的应用需要进行选择：

```
SendTo(byte[] data, EndPoint Remote)
```

上述程序代码中SendTo()方法，发送一个字节数据到由不同的Remote所指定的EndPoint上。稍微复杂一些的使用方式如下：

```
SendTo(byte[] data, SocketFlags Flags, EndPoint Remote)
```

这种使用方式带有一个SocketFlags类型的对象Flags，用于指定可以使用的任意特定的UDP套接字选项。使用下面这种方式，可以指定来自字节数组中的字节数：

```
SendTo(byte[] data, int Size, SocketFlags Flags, EndPoint Remote)
```

SendTo()方法还有一种格式，使用这种方法可以在字节数组内指定发送数据的起始位置，请看下面的代码：

```
SendTo(byte[] data, int Offset, int Size, SocketFlags Flags, EndPoint Remote)
```

ReceiveFrom() 该方法与SendTo()方法的使用非常类似，二者之间最重要的不同就是EndPoint对象声明的方式不一样。ReceiveFrom()方法的调用方式如下：

```
ReceiveFrom(byte[] data, ref EndPoint Remote)
```

一般来说，需要定义一个字节数组来接收数据。

在这个方法的使用中，真正有趣的是第二个参数的使用：ref EndPoint。在这里，所传递的不是一个EndPoint对象，而是将参数传递给一个EndPoint对象。虽然在C和C++程序中为变量传递参数非常普遍，然而，C#程序不是这样。在C#程序中，参数是变量在内存中的位置，而不是变量值。ReceiveFrom()方法将从远程设备中将EndPoint信息放置到参数所表示的EndPoint对象的内存中。

在后面我们将讲述的简单UDP服务器和客户机的例子中，这两个方法都要用到。这些例子示范了使用C#套接字进行无连接通信的最基本的方法。

UDP服务器

虽然UDP应用不是严格意义上真正的服务器和客户机，为了简便起见，我们仍然把下面这个应用叫做UDP服务器。在这个应用中，创建了一个Socket对象，并将这个对象和一套IPEndPoint对象进行了绑定，因此，能够等待流入的数据包：

```
IPEndPoint ipep = new IPEndPoint(IPAddress.Any,
                                   9050);
Socket newsock = Socket(AddressFamily.InterNetwork,
                          SocketType.Dgram, ProtocolType.Udp);
newsock.Bind(ipep);
```

在无连接通信中，必须指定Dgram SocketType和Udp ProtocolType。请记住一点，如

果在应用中不需要从特定的UDP端口中接收UDP数据，就不需要将套接字与指定的IPEndPoint进行绑定。然而，如果需要收听特定的端口，比如服务器端口等，则必须使用Bind()方法。

程序清单6.1中的SimpleUdpSrvr.cs程序示范了一个简单的UDP服务器的基本部件，在这个服务器中，将一个专门的套接字进行了绑定，以便与其他的UDP客户机进行连接。

程序清单6.1 SimpleUdpSrvr.cs程序

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class SimpleUdpSrvr
{
    public static void Main()
    {
        int recv;
        byte[] data = new byte[1024];
        IPEndPoint ipep = new IPEndPoint(IPAddress.Any, 9050);

        Socket newsock = new Socket(AddressFamily.InterNetwork,
                                     SocketType.Dgram, ProtocolType.Udp);

        newsock.Bind(ipep);
        Console.WriteLine("Waiting for a client...");

        IPEndPoint sender = new IPEndPoint(IPAddress.Any, 0);
        EndPoint Remote = (EndPoint)(sender);

        recv = newsock.ReceiveFrom(data, ref Remote);

        Console.WriteLine("Message received from {0}:", Remote.ToString());
        Console.WriteLine(Encoding.ASCII.GetString(data, 0, recv));

        string welcome = "Welcome to my test server";
        data = Encoding.ASCII.GetBytes(welcome);
        newsock.SendTo(data, data.Length, SocketFlags.None, Remote);

        while(true)
        {
            data = new byte[1024];
            recv = newsock.ReceiveFrom(data, ref Remote);

            Console.WriteLine(Encoding.ASCII.GetString(data, 0, recv));
            newsock.SendTo(data, recv, SocketFlags.None, Remote);
        }
    }
}
```

如上所述,对于接收流入的UDP消息的UDP服务器程序来说,必须将程序与本地系统中指定的UDP端口进行绑定。这可以通过使用合适的本地IP地址(或者就像这里所示范的,IPAddress.Any使用系统中任意网络接口)创建一个IPEndPoint对象,以及合适的UDP端口:

```
IPEndPoint ipep = new IPEndPoint(IPAddress.Any, 9050);

Socket newsock = new Socket(AddressFamily.InterNetwork,
                             SocketType.Dgram, ProtocolType.Udp);

newsock.Bind(ipep);
```

在这个范例程序中的UDP服务器能够在端口9050从网络上接收任意流入的UDP数据包。

说明: 由于在两个主机之间没有已建立的连接,因此,UDP对于数据包的来源不是很在意(这一点与TCP不同)。然而,由于UDP的这个特点,当与多个UDP客户机进行通信时,必须十分小心,需要认真检查所接收到的数据包的传输点。

在创建样本UDP服务器时,必须十分留心,不能选择已经被机器上的其他应用使用的UDP端口。可以在命令行提示符下使用netstat命令监视特殊的UDP端口,以便合理使用UDP端口号。请看下面的例子:

```
C:\>netstat -a
```

```
Active Connections
```

Proto	Local Address	Foreign Address	State
TCP	abednego:epmap	0.0.0.0:0	LISTENING
TCP	abednego:microsoft-ds	0.0.0.0:0	LISTENING
TCP	abednego:1025	0.0.0.0:0	LISTENING
TCP	abednego:1034	0.0.0.0:0	LISTENING
TCP	abednego:5000	0.0.0.0:0	LISTENING
TCP	abednego:12174	0.0.0.0:0	LISTENING
TCP	abednego:38292	0.0.0.0:0	LISTENING
TCP	abednego:1026	0.0.0.0:0	LISTENING
TCP	abednego:netbios-ssn	0.0.0.0:0	LISTENING
TCP	abednego:11133	0.0.0.0:0	LISTENING
UDP	abednego:epmap	*:*	
UDP	abednego:microsoft-ds	*:*	
UDP	abednego:isakmp	*:*	
UDP	abednego:1027	*:*	
UDP	abednego:1035	*:*	
UDP	abednego:9050	*:*	
UDP	abednego:38037	*:*	
UDP	abednego:38293	*:*	
UDP	abednego:ntp	*:*	
UDP	abednego:1900	*:*	
UDP	abednego:ntp	*:*	
UDP	abednego:netbios-ns	*:*	
UDP	abednego:netbios-dgm	*:*	

```
UDP      abednego:1036      *:*
UDP      abednego:1900      *:*
UDP      abednego:12564     *:*
```

C:\>

`netstat -a`命令显示系统中所有活动的TCP和UDP会话。根据系统中所配置的网络应用的数量，我们可以看到很多输出。第一列显示所采用的协议类型，第二列显示主机名或者IP地址以及端口号，第三列显示IP主机名或地址以及远程设备的端口号（在TCP通信中，如果该端口与远程设备相连的话），最后一列显示TCP连接中的TCP状态。

在设计应用时，只能选择使用在`netstat`列表中没有作为活动状态的UDP端口而列出的端口。一旦`SimpleUdpSrvr`激活，则在前面`netstat`的输出列表中将有选择的UDP端口的一条信息：

```
UDP      abednego:9050      *.*
```

与TCP协议的客户机/服务器模式相类似，与UDP进行通信的远程设备通信必须对发送数据和接收数据达成一致。如果客户机和服务器同时等待接收数据，两个设备都将处于阻塞状态。为了达成这个协议，`SimpleUdpSrvr`程序必须遵循下列规则：

- 等待从客户机中接收消息
- 向客户机发送欢迎标语
- 等待其他的客户机消息并发送回去

这些功能都可以使用`Socket`类中的`ReceiveFrom()`和`SendTo()`方法来实现。

因为无连接套接字没有建立连接，因此，每一个`SendTo()`方法中必须包含有远程设备的`EndPoint`信息。这一点对于网络编程人员来说是强制性的，可以从所接收到的消息中获取这个信息。为了获取`EndPoint`信息，首先创建一个空`EndPoint`对象，并在`ReceiveFrom()`方法中引用这个对象：

```
IPEndPoint sender = new IPEndPoint(IPAddress.Any, 0);
EndPoint Remote = (EndPoint)(sender);
recv = newsock.ReceiveFrom(data, ref Remote);
```

`Remote`对象中含有将消息发送到服务器中的远程设备的IP信息，这个信息指明了流入的数据，如果服务器希望向客户机返回消息，也可以使用这个信息：

```
string welcome = "Welcome to my test server";
data = Encoding.ASCII.GetBytes(welcome);
newsock.SendTo(data, data.Length, SocketFlags.None, Remote);
```

警告：在本书第5章“面向连接的套接字”中的TCP服务器中，在`ReceiveFrom()`方法调用之前，必须将接收数据缓冲器复位成最大尺寸，或者`Length`属性将表明前面接收到的消息的长度。但是在UDP通信中，这样做会导致一系列问题。

与第5章中所讨论过的TCP服务器不同，如果没有合适的UDP客户机，则不能测试UDP服务器。接下来我们将描述客户机的创建过程。

UDP客户机

UDP客户机程序与服务器程序非常类似。

因为客户机不需要在指定的UDP端口等待流入的数据，因此，不使用Bind()方法，而是使用在数据发送时系统随机指定的一个UDP端口，而且使用同一个端口接收返回的消息。在开发产品时，要为客户机指定一套UDP端口，以便服务器和客户机程序使用相同的端口号。

警告：在同一台机器上使用UDP服务器和客户机时必须特别小心。不能使用同一个端口号绑定两个程序，否则将会出现错误。一次只能有一个应用绑定到一个特定的端口号。要么选择不同的端口号，要么可以让系统为客户机随机分配一个端口号。

程序清单6.2所示的SimpleUdpClient.cs程序示范了UDP通信的基本原理。

程序清单6.2 SimpleUdpClient.cs程序

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class SimpleUdpClient
{
    public static void Main()
    {
        byte[] data = new byte[1024];
        string input, stringData;
        IPEndPoint ipep = new IPEndPoint(
            IPAddress.Parse("127.0.0.1"), 9050);

        Socket server = new Socket(AddressFamily.InterNetwork,
            SocketType.Dgram, ProtocolType.Udp);

        string welcome = "Hello, are you there?";
        data = Encoding.ASCII.GetBytes(welcome);
        server.SendTo(data, data.Length, SocketFlags.None, ipep);

        IPEndPoint sender = new IPEndPoint(IPAddress.Any, 0);
        EndPoint Remote = (EndPoint)sender;

        data = new byte[1024];
        int recv = server.ReceiveFrom(data, ref Remote);

        Console.WriteLine("Message received from {0}:", Remote.ToString());
        Console.WriteLine(Encoding.ASCII.GetString(data, 0, recv));

        while(true)
        {
            input = Console.ReadLine();
```



```
        if (input == "exit")
            break;
        server.SendTo(Encoding.ASCII.GetBytes(input), Remote);
        data = new byte[1024];
        recv = server.ReceiveFrom(data, ref Remote);
        stringData = Encoding.ASCII.GetString(data, 0, recv);
        Console.WriteLine(stringData);
    }
    Console.WriteLine("Stopping client");
    server.Close();
}
}
```

UDP客户机程序首先定义一个`EndPoint`，UDP服务器将发送数据包到这个`EndPoint`。如果在远程设备上运行`SimpleUdpSrvr`程序，在`EndPoint`定义中必须输入适当的IP地址和UDP端口号信息，请看下面的程序代码：

```
EndPoint ipep = new EndPoint(
    IPAddress.Parse("127.0.0.1"), 9050);
```

客户机程序迅速向服务器机器发送一个消息进行自我介绍，然后等待欢迎标语的返回。因为客户机程序不需要在特定的端口上接收UDP消息，因此客户机不用绑定`Socket`对象。它将在原来发送消息的端口上接收返回的UDP消息。

`SimpleUdpClient`程序读取控制台输入信息，并在退出`while`循环之前等待`exit`语句，一旦退出循环，套接字就关闭。

测试客户机和服务器程序

在指定的服务器机器上启动`SimpleUdpSrvr`程序，当这个服务器程序启动后，显示一条短消息，如下所示：

```
C:\>SimpleUdpSrvr
Waiting for a client...
```

这里没有什么奇怪的。在服务器启动后，就可以在同一台机器的不同命令行窗口下或者在网络中其他的机器上运行`SimpleUdpClient`程序（如果使用网络中其他的远程机器，则要将127.0.0.1 IP地址改为服务器的IP地址）。

`SimpleUdpClient`立即向服务器发送一条消息，服务器返回欢迎消息，在客户机端，显示信息如下：

```
C:\>SimpleUdpClient
Message received from 127.0.0.1:9050:
Welcome to my test server
```

远程服务器的`EndPoint`对象表明确实是从自己所绑定的UDP端口发送的数据，端口是9050。

在服务器端，来自服务器的连接消息如下所示：

```
C:\>SimpleUdpSrvr
Waiting for a client...
Message received from 127.0.0.1:1340:
Hello, are you there?
```

由于客户机没有与特定的UDP端口进行绑定，因此选择了一个空闲的UDP端口进行通信，一旦通信建立起来，就可以在客户机的控制台中输入消息，查看消息向服务器的发送和显示，以及返回客户机的消息和显示。

如果从远程客户机中与服务器进行连接，可以使用WinDump和Analyzer程序监视应用程序所带来的网络流量情况。图6.1所示是一个Analyzer输出窗口的例子，在这个窗口中有来自一个简单客户机程序的输出窗口。

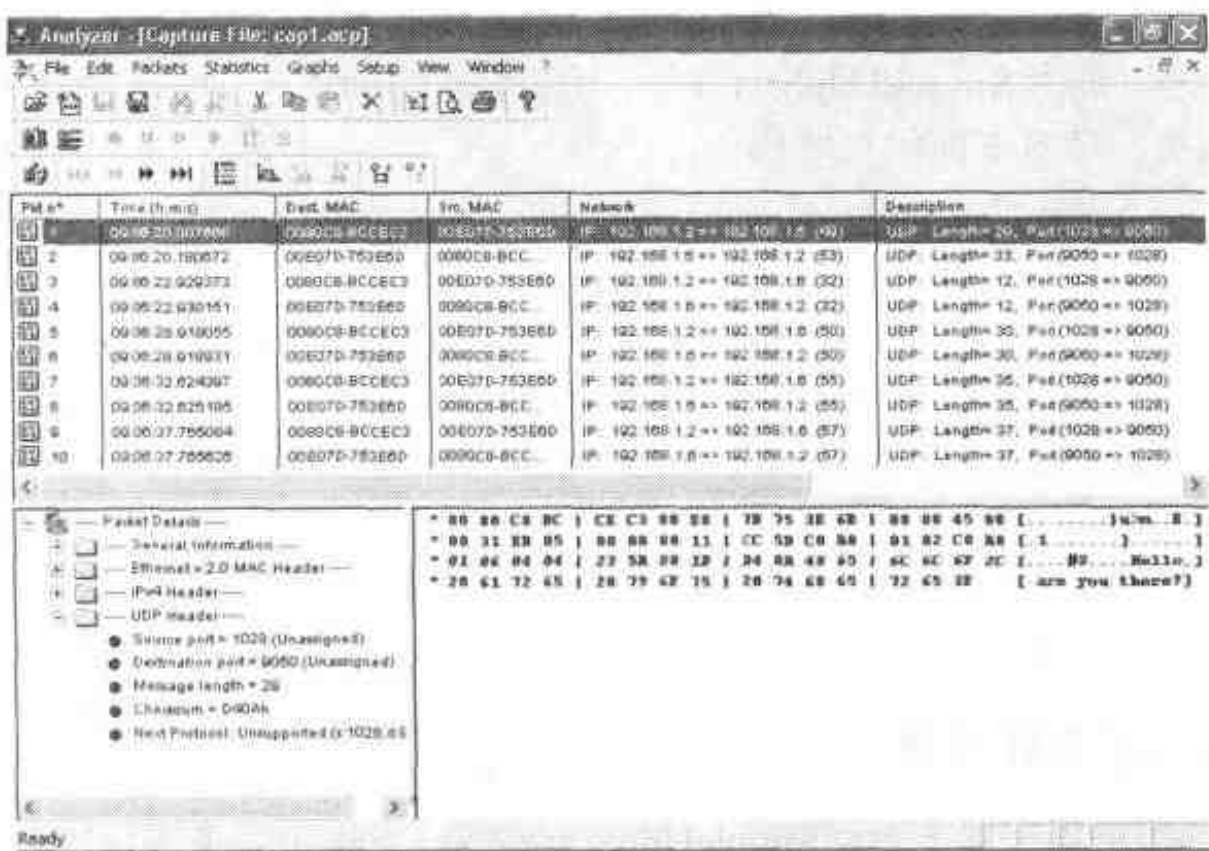


图6.1 SimpleUdpClient程序的Analyzer输出

从网络中数据包的情况我们可以看出，每一个UDP消息都是在单个数据包中发送到服务器的，而且每一个UDP消息的返回信息也是作为独立的数据包返回客户机。这就是UDP通信中能够区分消息边界的原因。

终止UDP服务器程序

接下来我们分析上述UDP服务器程序的一个严重缺陷。当在客户机控制台中输入exit命令时，客户机程序关闭套接字并终止。然而，由于客户机和服务器之间没有连接，因此UDP服务器（不像TCP服务器）无法知道什么时候远程的客户机已经关闭了套接字。因为不知道客户机已经终止，因此，服务器将仍然在指定的UDP端口处等待数据包。

在SimpleUdpSrvr程序仍然处于运行状态时，在同一台机器上或者从网络中其他的机器上重新启动SimpleUdpClient程序就可以观察到上述现象的发生。服务器程序将继续接收来

自客户机的问候消息，以及其他消息（为了终止SimpleUdpSrvr，按Ctrl+C）。

在实际的应用中，我们需要编写相应的程序，从而使服务器能够监视外部事件的发生，比如来自控制台的特殊的命令等，然后决定是否关闭套接字。

UDP客户机程序中Connect()方法的应用

读者可能感到非常纳闷，我们的应用非常简单，仅仅需要向一个UDP服务器接收和发送数据，为什么UDP客户机应用的ReceiveFrom()和SendTo()方法却如此复杂？当然，我们可以简化这个问题。前面我们已经提到过，UDP方法能够在任意时候向网络中的任意主机发送UDP数据包。因为在UDP中不需要建立连接，因此，必须在每一个SendTo()和ReceiveFrom()中指定目标地址。如果不打算从同一个主机上发送和接收数据，则可以走捷径。

UDP套接字创建后，可以使用标准的Connect()方法指定远程UDP服务器（同TCP程序中一样），然后使用Receive()和Send()方法向远程主机传输数据。这个通信将仍然使用UDP数据包，而且，节省了大量的工作。在程序清单6.3所示的程序中示范了这种技术。

程序清单6.3 OddUdpClient.cs程序

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class OddUdpClient
{
    public static void Main()
    {
        byte[] data = new byte[1024];
        string input, stringData;
        IPEndPoint ipep = new IPEndPoint(
            IPAddress.Parse("127.0.0.1"), 9050);

        Socket server = new Socket(AddressFamily.InterNetwork,
            SocketType.Dgram, ProtocolType.Udp);

        server.Connect(ipep);
        string welcome = "Hello, are you there?";
        data = Encoding.ASCII.GetBytes(welcome);
        server.Send(data);

        data = new byte[1024];
        int recv = server.Receive(data);

        Console.WriteLine("Message received from {0}:", ipep.ToString());
        Console.WriteLine(Encoding.ASCII.GetString(data, 0, recv));

        while(true)
```

```
(
    input = Console.ReadLine();
    if (input == "exit")
        break;
    server.Send(Encoding.ASCII.GetBytes(input));
    data = new byte[1024];
    recv = server.Receive(data);
    stringData = Encoding.ASCII.GetString(data, 0, recv);
    Console.WriteLine(stringData);
}
Console.WriteLine("Stopping client");
server.Close();
}
}
```

OddUdpClient与**SimpleUdpClient**非常类似，二者之间惟一的不同是：一旦UDP套接字创建，它就与一个特殊的**IPEndPoint**进行连接，请看下面的代码：

```
IPEndPoint ipep = new IPEndPoint(
    IPAddress.Parse("127.0.0.1"), 9050);

Socket server = new Socket(AddressFamily.InterNetwork,
    SocketType.Dgram, ProtocolType.Udp);

server.Connect(ipep);
```

在这种情况下，**Connect()**方法没有实际操作，因为这是一个UDP数据包套接字，在UDP通信中没有实际的链路，但是这个套接字的信息设置到**IPEndPoint**对象中。所有**Send()**方法和**Receive()**方法的调用都自动使用**IPEndPoint**对象的信息，从而避免了使用**Send()**和**Receive()**方法对数据的处理。

我们可以采用常规的方式启动**SimpleUdpSrvr**来测试**OddUdpClient**程序，使用**OddUdpClient**程序来发送数据，甚至可以使用**WinDump**或**Analyzer**来监视网络上的流量，而且每一个消息都是以单个数据包发送到服务器，保留了所有的消息边界，就像在**SimpleUdpClient.cs**程序中一样。

UDP消息的区分

UDP突出的优点是解决了在TCP通信中存在的消息边界的问题。UDP保存了所有所发送消息的边界，**ReceiveFrom()**方法的每一次调用，都是对独立的**SendTo()**调用的结果数据进行读取操作。

现在我们来回顾一下UDP套接字的有关情况，通过前面的介绍，我们了解到一旦UDP套接字创建，它能够从任何一个客户机中接收数据。UDP套接字区分所接收到的数据是来自哪一个客户机的方式是强制性的，即要求每一个消息中都必须带有源设备的IP地址，从而接收设备能够区分消息本身和发送方。

上述过程可以用两个程序段来实现。**TestUdpClient.cs**程序向服务器发送多个消息，**TestUdpSrvr.cs**程序试图分别读取每一个消息（有点类似于上一章我们讲述过的那个失败的TCP消息测试程序）。首先，请看程序清单6.4所示的**TestUdpSrvr.cs**程序。

程序清单6.4 TestUdpSrvr.cs程序

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class TestUdpSrvr
{
    public static void Main()
    {
        int recv;
        byte[] data = new byte[1024];
        IPEndPoint ipep = new IPEndPoint(IPAddress.Any, 9050);

        Socket newsock = new Socket(AddressFamily.InterNetwork,
                                     SocketType.Dgram, ProtocolType.Udp);

        newsock.Bind(ipep);
        Console.WriteLine("Waiting for a client...");

        IPEndPoint sender = new IPEndPoint(IPAddress.Any, 0);
        EndPoint tmpRemote = (EndPoint)(sender);

        recv = newsock.ReceiveFrom(data, ref tmpRemote);

        Console.WriteLine("Message received from {0}:", tmpRemote.ToString());
        Console.WriteLine(Encoding.ASCII.GetString(data, 0, recv));

        string welcome = "Welcome to my test server";
        data = Encoding.ASCII.GetBytes(welcome);
        newsock.SendTo(data, data.Length, SocketFlags.None, tmpRemote);

        for(int i = 0; i < 5; i++)
        {
            data = new byte[1024];
            recv = newsock.ReceiveFrom(data, ref tmpRemote);
            Console.WriteLine(Encoding.ASCII.GetString(data, 0, recv));
        }
        newsock.Close();
    }
}
```

TestUdpSrvr在9050端口¹与一个UDP套接字进行了绑定，等待客户机发送来的问候消息，

当接收到问候语时，服务器发送一个欢迎标语，并试图从客户机接收位于一行内的5条消息。

说明：这个服务器程序有一个非常好的特点，即应用程序协议本身指定在发送完五条消息后，客户机将关闭套接字，因此，服务器也可以关闭套接字。

接下来看看程序清单6.5所示的TestUdpClient.cs程序，在这个程序中示范了向服务器发送多个UDP消息。

程序清单6.5 TestUdpClient.cs程序

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class TestUdpClient
{
    public static void Main()
    {
        byte[] data = new byte[1024];
        IPEndPoint ipep = new IPEndPoint(
            IPAddress.Parse("127.0.0.1"), 9050);

        Socket server = new Socket(AddressFamily.InterNetwork,
            SocketType.Dgram, ProtocolType.Udp);

        string welcome = "Hello, are you there?";
        data = Encoding.ASCII.GetBytes(welcome);
        server.SendTo(data, data.Length, SocketFlags.None, ipep);

        IPEndPoint sender = new IPEndPoint(IPAddress.Any, 0);
        EndPoint tmpRemote = (EndPoint)sender;

        data = new byte[1024];
        int recv = server.ReceiveFrom(data, ref tmpRemote);

        Console.WriteLine("Message received from {0}:", tmpRemote.ToString());
        Console.WriteLine(Encoding.ASCII.GetString(data, 0, recv));

        server.SendTo(Encoding.ASCII.GetBytes("message 1"), tmpRemote);
        server.SendTo(Encoding.ASCII.GetBytes("message 2"), tmpRemote);
        server.SendTo(Encoding.ASCII.GetBytes("message 3"), tmpRemote);
        server.SendTo(Encoding.ASCII.GetBytes("message 4"), tmpRemote);
        server.SendTo(Encoding.ASCII.GetBytes("message 5"), tmpRemote);

        Console.WriteLine("Stopping client");
        server.Close();
    }
}
```

这个TestUdpClient程序向指定的服务器发送问候消息，然后接收来自服务器的欢迎标语。在接收到欢迎标语后，它在一行内连续发送五条独立的消息，然后关闭套接字。

如果可能，可以尝试一下在网络中对这些程序进行测试，当进行测试时，我们将看到每一个消息接收的方式同发送的方式相同，UDP系统能够维持消息的边界。然而，读者可能已经注意到，由于所采用的网络不同，出现了消息丢失的现象。这是UDP通信的缺憾之一。虽然消息保持了完整，但是却没有保证消息的可靠传递。下面的章节我们将讨论UDP通信的一些自身问题。

UDP通信故障

在TCP通信中我们要解决消息的边界问题，在UDP通信中我们要解决一些其他的问题：

- ReceiveFrom()方法导致的数据丢失
- 丢失数据包的监测和认可

这两个问题经常将习惯了TCP通信的网络编程人员致于意想不到的进退两难的境地，而且是每一个UDP网络编程人员必须慎重考虑的问题，接下来的内容将围绕这两个问题，介绍在使用C#进行网络编程时需要掌握的有关技术和工具。

预防数据丢失

TCP通信的优点之一就是其固有的TCP缓冲区机制。在本书第5章里我们已经讲述过，TCP套接字发送的所有数据在被发送之前都放置在一个内部的缓冲区域中，同样，所有接收到的数据在被Receive()方法读取之前也被放置在这个缓冲区中，当Receive()方法试图从缓冲区中读取数据时，如果只读取了部分数据，那么剩下的数据将继续留在缓冲区内等待下一个Receive()的读取。但是在UDP通信中，情况却不是这样的。

UDP不考虑数据包的重传问题，因此没有缓冲区机制。由套接字发到网络上的所有的数据都是立即发送，同样，接收到的每一组数据也是立即转发到下一个ReceiveFrom()调用。这种机制带来了一个非常重要的存在潜在危险的问题，网络程序员在进行系统设计时经常忽视，即没有从UDP套接字中立即读取的数据将永久丢失。

在程序中使用ReceiveFrom()方法时，程序员必须保证所指定的数据缓冲区足够大，能够容下所有来自UDP套接字的数据。如果缓冲区太小，将会发生数据丢失的现象。我们将程序清单6.2中的SimpleUdpClient程序做一些改动，就能够观察这种现象的发生。执行结果见程序清单6.6 BadUdpClient.cs。

程序清单6.6 BadUdpClient.cs程序

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class BadUdpClient
```

```

{
    public static void Main()
    {
        byte[] data = new byte[30];
        string input, stringData;
        IPEndPoint ipep = new IPEndPoint(
            IPAddress.Parse("127.0.0.1"), 9050);

        Socket server = new Socket(AddressFamily.InterNetwork,
            SocketType.Dgram, ProtocolType.Udp);

        string welcome = "Hello, are you there?";
        data = Encoding.ASCII.GetBytes(welcome);
        server.SendTo(data, data.Length, SocketFlags.None, ipep);

        IPEndPoint sender = new IPEndPoint(IPAddress.Any, 0);
        EndPoint tmpRemote = (EndPoint)sender;

        data = new byte[30];
        int recv = server.ReceiveFrom(data, ref tmpRemote);

        Console.WriteLine("Message received from {0}:", tmpRemote.ToString());
        Console.WriteLine(Encoding.ASCII.GetString(data, 0, recv));

        while(true)
        {
            input = Console.ReadLine();
            if (input == "exit")
                break;
            server.SendTo(Encoding.ASCII.GetBytes(input), tmpRemote);
            data = new byte[30];
            recv = server.ReceiveFrom(data, ref tmpRemote);
            stringData = Encoding.ASCII.GetString(data, 0, recv);
            Console.WriteLine(stringData);
        }
        Console.WriteLine("Stopping client");
        server.Close();
    }
}

```

在这个BadUdpClient程序中定义的数据缓冲区仅有30个字节大小:

```
byte[] data = new byte[30];
```

对上述程序进行编译之后,我们就可以使用SimpleUdpSrvr程序对其进行测试。在最一开始,运转状况似乎正常,BadUdpClient向服务器发送一个标准的问候消息,接收来自服务器的欢迎标语。也可以测试一些短消息,程序的运行似乎也很正常,但是当发送超过30个字符的长消息时,就会出现问題。请看下面的信息:


```
C:\>BadUdpClient
```

```
Message received from 127.0.0.1:9050:
```

```
Welcome to my test server
```

```
test message
```

```
test message
```

```
longer test message
```

```
longer test message
```

```
This is an even longer test message
```

```
Unhandled Exception: System.Net.Sockets.SocketException: A message sent on a
datagram socket was larger than the internal message buffer or some other
network limit, or the buffer used to receive a datagram into was smaller
than the datagram itself
```

```
at System.Net.Sockets.Socket.ReceiveFrom(Byte[] buffer, Int32 offset, Int32
size, SocketFlags socketFlags, EndPoint& remoteEP)
```

```
at System.Net.Sockets.Socket.ReceiveFrom(Byte[] buffer, EndPoint& remoteEP)
```

```
at BadUdpClient.Main()
```

```
C:\>
```

输入最长的消息时，应用产生一个异常，当然BadUdpClient不能处理这个异常，从而导致许多问题的发生。最根本的问题是ReceiveFrom()方法检测到数据太多，在所提供的缓冲区中无法放置所有的数据。因此ReceiveFrom()方法试图通过发送一个SocketException通报这种情况。在遇到数据过大时，这是一种非常方便的通报方式，同时也意味着将会有数据的丢失现象发生。

虽然我们可以找回原始数据，但是一般来说，在这种情况下，我们可以采取一定的补偿措施。在ReceiveFrom()方法的周围设置一个try-catch块，当由于缓冲区太小而造成数据丢失，并试图修改其大小而满足下一个接收的数据尺寸时，就能检测到。在程序清单6.7所示的BetterUdpClient.cs中示范了这种技术，请看下面的程序清单。

程序清单6.7 BetterUdpClient.cs程序

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class BetterUdpClient
{
    public static void Main()
    {
        byte[] data = new byte[30];
        string input, stringData;
        IPEndPoint ipep = new IPEndPoint(
            IPAddress.Parse("127.0.0.1"), 9050);

        Socket server = new Socket(AddressFamily.InterNetwork,
```

```

        SocketType.Dgram, ProtocolType.Udp);

    string welcome = "Hello, are you there?";
    data = Encoding.ASCII.GetBytes(welcome);
    server.SendTo(data, data.Length, SocketFlags.None, ipep);

    IPEndPoint sender = new IPEndPoint(IPAddress.Any, 0);
    EndPoint tmpRemote = (EndPoint)sender;

    data = new byte[30];
    int recv = server.ReceiveFrom(data, ref tmpRemote);

    Console.WriteLine("Message received from {0}:", tmpRemote.ToString());
    Console.WriteLine(Encoding.ASCII.GetString(data, 0, recv));

    int i = 30;
    while(true)
    {
        input = Console.ReadLine();
        if (input == "exit")
            break;
        server.SendTo(Encoding.ASCII.GetBytes(input), tmpRemote);
        data = new byte[i];
        try
        {
            recv = server.ReceiveFrom(data, ref tmpRemote);
            stringData = Encoding.ASCII.GetString(data, 0, recv);
            Console.WriteLine(stringData);
        } catch (SocketException)
        {
            Console.WriteLine("WARNING: data lost, retry message.");
            i += 10;
        }
    }
    Console.WriteLine("Stopping client");
    server.Close();
}
}

```

在这个**BetterUdpClient**程序中，没有使用固定大小的数据缓冲区数组，而是使用了一个变量，能够根据每一次**ReceiveFrom()**方法的调用而设置为不同的数值，请看下面的代码段：

```

    data = new byte[i];
    try
    {
        recv = server.ReceiveFrom(data, ref tmpRemote);
        stringData = Encoding.ASCII.GetString(data, 0, recv);
    }
}

```



```
        Console.WriteLine(stringData);  
    } catch (SocketException)  
    {  
        Console.WriteLine("WARNING: data lost, retry message.");  
        i += 10;  
    }  
}
```

如果发生了`SocketException`，我们知道这是由于在`ReceiveFrom()`方法中的数据缓冲区越界，程序警告客户加大缓冲区的大小，在此以后的所有`ReceiveFrom()`调用都使用增大后的数据缓冲区。数据结果如下：

```
C:\>BetterUdpClient .  
Message received from 127.0.0.1:9050:  
Welcome to my test server  
test  
test  
This is an even longer test message  
WARNING: data lost, retry message.  
This is an even longer test message  
This is an even longer test message  
This is a really, really, really long test message  
WARNING: data lost, retry message.  
This is a really, really, really long test message  
This is a really, really, really long test message  
exit  
Stopping client  
  
C:\>
```

在笔者测试的过程中，曾经输入过一个超过30个字节的句子，当然，出现了`SocketException`，而且显示了警告信息。为了确保数据缓冲区已经增大，我们又输入了一个大于30个字符的句子，看看结果如何。正如所期望的，数据被完整地接收了。这个例子示范了在UDP套接字中避免数据丢失的一个简单可行的方法。

说明：在实际的应用中，大多数情况下采用几何方法增加缓冲区尺寸，从而使数据的重传降低到最小，其中包括将数据缓冲区的尺寸翻倍的方法。

在这一节中，我们讲述了一个使用数据缓冲区的重要问题，即应用程序如何使用大小恰当的缓冲区。当然，我们也可以采用另外一种方法，就是将数据缓冲区设置得特别大，满足所有可能的情况，但是，这样做的不足就是占用大量的内存资源，造成系统资源的浪费。

预防数据包丢失

在UDP通信中另一个非常棘手的问题是数据包的丢失。因为UDP是一个无连接的协议，发送数据方无法得知所发送的数据是否准确地达到了远端设备。对于有些UDP应用来说，这也许没什么问题。例如，许多游戏使用UDP传输位置信息，每隔几秒钟，操作人员的位置信

息和游戏的状态就被传送一遍，如果在传输的过程中有数据包的丢失发生，由于很快就会发送下一次数据包，因此，基本上没有影响。

但是对于有些其他的应用来说，数据包的丢失可能导致巨大的问题。比如在使用简单网络管理协议（SNMP）接收网络管理数据时，管理员就希望能够接收到设备的响应消息，否则非常麻烦。

解决数据包丢失问题的最简单的方法就是，设计一种类似于TCP重传机制的协议。成功发送到远程主机上的数据包能够从远程设备上返回一个确认信息。如果在一定的时间内没有接收到来自远程设备的确认信息，将重发原始数据包。

在实现UDP重传时，有两种可以使用的技术：

异步套接字和定时器对象 这种技术需要使用能够监听流入数据包的一个套接字，在套接字设置为异步读取模式后，设置一个Timer对象。如果在异步读取数据完成之前，Timer关闭，则进行数据重传。在本书第8章“异步套接字编程”中将详细讲述有关这方面的内容。

同步套接字和套接字超时机制 在这种情况下，我们使用接下来将要讨论的SetSocketOption()方法。

套接字超时机制

在第3章中我们讲述过，ReceiveFrom()是一个能够导致阻塞的方法。该方法将阻塞程序的进行直到接收到一个数据包为止。在UDP程序设计中这非常糟糕，我们无法保证成功接收数据包，如果没有数据包到达，程序将永远等待，或者客户无法容忍而手工结束这次通信。

默认情况下，ReceiveFrom()方法无限期地等待数据，然而，使用套接字的选项，我们可以对其进行控制。我们能够控制该方法等待的时间，而且能够允许终止等待。SetSocketOption()方法为Socket对象提供一些选项，其中之一就是ReceiveTimeout，该选项设置了套接字在发出超时信号之前的等待时间。

SetSocketOption()方法的格式如下面代码所示：

```
SetSocketOption(SocketOptionLevel so, SocketOptionName sn, int value)
```

在上面代码中，SocketOptionLevel指出套接字选项的类型，SocketOptionName定义要设置的特殊的选项，最后一个参数（int value）指明选项的值。

由于ReceiveTimeout是一个套接字级别的选项，为指定ReceiveTimeout选项，必须使用下面代码中所示的格式：

```
server.SetSocketOption(SocketOptionLevel.Socket,  
    SocketOptionName.ReceiveTimeout, 3000);
```

因为SetSocketOption是Socket类中的一个方法，我们只能在活动的Socket对象中使用（在前面的范例中，称为Server）。整型参数用于定义在触发超时之前套接字等待数据的时间（毫秒）。

Timeout设置

在没有运行SimpleUdpSrvr程序时运行SimpleUdpClient程序将观察到超时现象。SimpleUdpClient程序等待服务器返回数据（这是不可能的，在这种情况下只能通过按“Ctrl+C”来手工结束客户机程序）。

在SimpleUdpClient程序中加入几行代码，设置一个超时机制。除了设置套接字的超时选项外，在程序清单6.8中还使用了GetSocketOption()方法显示超时数值真的发生了改变。

程序清单6.8 TimeoutUdpClient.cs程序

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class TimeoutUdpClient
{
    public static void Main()
    {
        byte[] data = new byte[1024];
        string input, stringData;
        int recv;
        IPEndPoint ipep = new IPEndPoint(
            IPAddress.Parse("127.0.0.1"), 9050);

        Socket server = new Socket(AddressFamily.InterNetwork,
            SocketType.Dgram, ProtocolType.Udp);

        int sockopt = (int)server.GetSocketOption(SocketOptionLevel.Socket,
            SocketOptionName.ReceiveTimeout);
        Console.WriteLine("Default timeout: {0}", sockopt);
        server.SetSocketOption(SocketOptionLevel.Socket,
            SocketOptionName.ReceiveTimeout, 3000);
        sockopt = (int)server.GetSocketOption(SocketOptionLevel.Socket,
            SocketOptionName.ReceiveTimeout);
        Console.WriteLine("New timeout: {0}", sockopt);

        string welcome = "Hello, are you there?";
        data = Encoding.ASCII.GetBytes(welcome);
        server.SendTo(data, data.Length, SocketFlags.None, ipep);

        IPEndPoint sender = new IPEndPoint(IPAddress.Any, 0);
        EndPoint tmpRemote = (EndPoint)sender;

        data = new byte[1024];
        recv = server.ReceiveFrom(data, ref tmpRemote);
        Console.WriteLine("Message received from {0}:", tmpRemote.ToString());
        Console.WriteLine(Encoding.ASCII.GetString(data, 0, recv));
    }
}
```

```

        while(true)
        {
            input = Console.ReadLine();
            if (input == "exit")
                break;
            server.SendTo(Encoding.ASCII.GetBytes(input), tmpRemote);
            data = new byte[1024];
            rcv = server.ReceiveFrom(data, ref tmpRemote);
            stringData = Encoding.ASCII.GetString(data, 0, rcv);
            Console.WriteLine(stringData);
        }
        Console.WriteLine("Stopping client");
        server.Close();
    }
}

```

TimeoutUdpClient首先从套接字中读取最初的**ReceiveTimeout**，并显示，然后将其设置为3秒；

```

int sockopt = (int)server.GetSocketOption(SocketOptionLevel.Socket,
    SocketOptionName.ReceiveTimeout);
Console.WriteLine("Default timeout: {0}", sockopt);
server.SetSocketOption(SocketOptionLevel.Socket,
    SocketOptionName.ReceiveTimeout, 3000);

```

GetSocketOption()方法返回一个**Object**类型的对象，因此必须将它匹配成一个整型数值。在对上述程序进行编译后，我们可以在不启动**SimpleUdpSrvr**程序的前提下运行这个程序以对其进行测试。输出结果将如下所示：

C:\>**TimeoutUdpClient**

Default timeout: 0

New timeout: 3000

Unhandled Exception: System.Net.Sockets.SocketException: An existing connection was forcibly closed by the remote host

at System.Net.Sockets.Socket.ReceiveFrom(Byte[] buffer, Int32 offset, Int32 size, SocketFlags socketFlags, EndPoint& remoteEP)

at System.Net.Sockets.Socket.ReceiveFrom(Byte[] buffer, EndPoint& remoteEP)

at TimeoutUdpClient.Main()

C:\>

正如我们所看到的，原始**ReceiveTimeout**为零，表明该程序将无限时等待数据。在加入**SetSocketOption()**方法后，**ReceiveTimeout**设置为3000毫秒，**GetSocketOption()**方法返回3000。

显示**ReceiveTimeout**数之后，上述程序中的**ReceiveFrom()**方法将等待流入的数据，等待

时间大约是3秒，然后，程序将生成一个`SocketException`并退出。

在套接字存在的全部时间内，`ReceiveTimeout`选项的设置都有效。启动`SimpleUdpSrvr`程序并运行`TimeoutUdpClient`，就可以进行测试。在正常交换过问候消息后，发送一些测试消息，使用`Ctrl+C`人为终止`SimpleUdpSrvr`。然后试着从客户机发送一条消息，三秒钟后，将会出现异常现象。

当`ReceiveTimeout`值到达时，套接字将停止等待数据并且产生一个`SocketException`。要在程序中使用这种超时机制，必须编写一定的程序代码来捕获这个`SocketException`。接下来我们将要讲述有关这方面的内容。

异常消息的捕获

既然我们已经了解到在超时的时候套接字将产生一个`SocketException`，因此，只要我们能够捕获这个`Exception`，就能够以一种文雅的方式通知顾客，并允许客户做其他的选择。而这个`ExceptionUdpClient.cs`程序将实现这个功能（见程序清单6.9）。

程序清单6.9 `ExceptionUdpClient.cs`程序

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class ExceptionUdpClient
{
    public static void Main()
    {
        byte[] data = new byte[1024];
        string input, stringData;
        int recv;
        IPEndPoint ipep = new IPEndPoint(
            IPAddress.Parse("127.0.0.1"), 9050);

        Socket server = new Socket(AddressFamily.InterNetwork,
            SocketType.Dgram, ProtocolType.Udp);

        int sockopt = (int)server.GetSocketOption(SocketOptionLevel.Socket,
            SocketOptionName.ReceiveTimeout);
        Console.WriteLine("Default timeout: {0}", sockopt);
        server.SetSocketOption(SocketOptionLevel.Socket,
            SocketOptionName.ReceiveTimeout, 3000);
        sockopt = (int)server.GetSocketOption(SocketOptionLevel.Socket,
            SocketOptionName.ReceiveTimeout);
        Console.WriteLine("New timeout: {0}", sockopt);

        string welcome = "Hello, are you there?";
        data = Encoding.ASCII.GetBytes(welcome);
```

```

server.SendTo(data, data.Length, SocketFlags.None, ipep);

IPEndPoint sender = new IPEndPoint(IPAddress.Any, 0);
EndPoint Remote = (EndPoint)sender;

data = new byte[1024];
try
{
    recv = server.ReceiveFrom(data, ref Remote);
    Console.WriteLine("Message received from {0}:", Remote.ToString());
    Console.WriteLine(Encoding.ASCII.GetString(data, 0, recv));
} catch (SocketException)
{
    Console.WriteLine("Problem communicating with remote server");
    return;
}

while(true)
{
    input = Console.ReadLine();
    if (input == "exit")
        break;
    server.SendTo(Encoding.ASCII.GetBytes(input), ipep);
    data = new byte[1024];
    try
    {
        recv = server.ReceiveFrom(data, ref Remote);
        stringData = Encoding.ASCII.GetString(data, 0, recv);
        Console.WriteLine(stringData);
    } catch (SocketException)
    {
        Console.WriteLine("Error receiving message");
    }
}
Console.WriteLine("Stopping client");
server.Close();
}
}

```

在这个ExceptionUdpClient.cs程序中，套接字的ReceiveTimeout设置为3000毫秒。所有的ReceiveFrom()方法调用都位于一个try-catch块内，以捕获在设置好的ReceiveTimeout时间内没有数据到达时产生的SocketException。

```

try
{
    recv = server.ReceiveFrom(data, ref tmpRemote);

```



```
        Console.WriteLine("Message received from {0}:", tmpRemote.ToString());
        Console.WriteLine(Encoding.ASCII.GetString(data, 0, recv));
    } catch (SocketException)
    {
        Console.WriteLine("Problem communicating with remote server");
        return;
    }
}
```

如果在设置好的超时时间段内，没有接收到数据，则产生一个`SocketException`，而且`catch`块内的语句会通知顾客没有来自远程设备的任何响应信息。`catch`段的具体执行方式取决于故障出现的位置。比如说，如果在通信的开始时没有收到来自服务器的数据包，就假定没有服务器，程序终止；在消息交换的过程中，单个数据包的丢失不能造成程序的终止，而是在控制台上显示一条警告消息，接收数据包的循环继续进行。

在没有启动响应的服务器程序时运行`ExceptionUdpClient`程序，运行结果如下：

```
C:\>ExceptionUdpClient
Default timeout: 0
New timeout: 3000
Problem communicating with remote server
C:\>
```

下面，我们将讲述一种更加温和的方式来处理数据包的丢失。

警告：在Windows XP Professional工作环境下测试超时程序，作者曾经注意到一种现象，当连接本地IP地址或者回退地址时，所设置的套接字超时似乎不太正常。然而，当与远程的IP地址进行连接时，一切正常（这也是必须在实际的网络环境中测试自己应用程序的理由之一）。

如果`SimpleUdpSrvr`一开始运行正常，但是在发送了一些消息后就停止了，输出结果如下所示：

```
C:\>ExceptionUdpClient
Default timeout: 0
New timeout: 3000
Message received from 127.0.0.1:9050:
Welcome to my test server
test message
test message
Second test message
Second test message
Third test message
Error receiving message
Last test message
Error receiving message
exit
Stopping client
C:\>
```

在这种情况下，消息循环中丢失的消息能够识别出来，但是客户仍然可以控制并且能够继续发送消息或者使用`exit`命令终止程序。

这种意外情况捕获技术监视所有的`ReceiveFrom()`方法，如果有数据丢失则通知顾客。这个特点非常好但是不能解决实际问题，我们的问题是将发送的原始消息成功发送到目的地。要解决这个问题，首先要重发原始消息，下面我们将讲述有关内容。

数据重传

有很多原因能够造成UDP数据包无法到达目的地，有些原因只是暂时的，再试一次或几次就有可能成功，这叫做“重试”。许多产品级UDP应用都允许在最终放弃通信之前多试几次。

如果我们要设计的UDP应用环境必须发送消息并且必须接收返回消息，如果没有接收到返回消息，我们就可以使用这种重试机制将数据重发几次。在最终接收到返回消息后，可以继续下面的应用，如果没有消息返回，则必须向客户声明并提供解决方案。

创建重传机制

实现消息重传的最简单的方式是在类中创建一些独立的方法来处理所有发送和接收的消息。具体步骤如下：

1. 向远程主机发送一条消息；
2. 等待远程主机的应答；
3. 如果收到应答，接收应答消息，并退出方法（带着所接收到的数据和数据的大小信息）；
4. 如果在规定超时时间内没有接收到应答消息，重试的次数加1；
5. 检查重试次数。如果比预定的数值小，返回步骤1。如果等于预先设定的次数，终止重传，并向顾客报告结果。

一旦创建了发送和接收UDP数据包的类方法，这些方法就能够用在程序中任意发送消息到远程主机并等待响应消息的位置。在C#代码中，这个方法类似于程序清单6.10所示。

程序清单6.10 SndRcvData()方法

```
int SndRcvData(Socket s, byte[] message, EndPoint rmtdevice)
{
    int recv;
    int retry = 0;

    while (true)
    {
        Console.WriteLine("Attempt #{0}", retry);
        try
        {
            s.SendTo(message, message.Length, SocketFlags.None, rmtdevice);
            data = new byte[1024];
```



```

        recv = s.ReceiveFrom(data, ref rmtdevice);
    } catch (SocketException)
    {
        recv = 0;
    }

    if (recv > 0)
    {
        return recv;
    } else
    {
        retry++;
        if (retry > 4)
        {
            return 0;
        }
    }
}
}

```

这个方法需要三个参数：

- 已经由调用程序创建的一个**Socket**对象
- 发送到远程主机上的数据消息
- 指向IP地址和远程主机的一个**EndPoint**对象

传递给新方法的**Socket**对象要作为UDP套接字建立，并且将**ReceiveTimeout SocketOption**的值设为在特定的网络上等待响应消息的时间。

SndRcvData()方法首先试图使用标准的**SendTo()**方法发送消息到远程主机。在发送消息之后，**SndRcvData()**阻塞**ReceiveFrom()**方法，等待返回消息。如果在**ReceiveTimeout**所设置的时间内接收到了返回消息，**SndRcvData()**把接收到的返回消息放入一个全局数据缓冲区内（该缓冲区在类里定义为一个字节数组）并返回接收到的字节数。调用程序使用**SndRcvData()**方法返回的消息大小值能够从全局数据缓冲区中读取所接收到的消息。

如果在**ReceiveTimeout**设置的时间内没有消息返回，则**SocketException**产生并且**catch**程序块将起作用。在**catch**块内，**recv**设置为零。**try-catch**块执行之后，对**recv**的值进行测试，正值意味着成功接收了消息，零值表明没有接收到消息，因此将重试次数加1，然后监测重试次数是否已经达到最大值。如果还没有到最大值，则从消息重发开始，重复执行整个步骤。

如果**retry**已经为预先设置的最大值，则**SndRcvData()**方法返回一个零值。调用方法能够检测到这个零值并做出相应的处理。

在程序中使用方法

程序清单6.11中所示的**RetryUdpClient.cs**程序，实现了一种新的消息重传技术，能够与远程UDP回声服务器进行通信。

程序清单6.11 RetryUdpClient.cs程序

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class RetryUdpClient
{
    private byte[] data = new byte[1024];
    private static IPEndPoint sender = new IPEndPoint(IPAddress.Any, 0);
    private static EndPoint Remote = (EndPoint)sender;

    private int SndRcvData(Socket s, byte[] message, EndPoint rmtdevice)
    {
        int recv;
        int retry = 0;

        while (true)
        {
            Console.WriteLine("Attempt #{0}", retry);
            try
            {
                s.SendTo(message, message.Length, SocketFlags.None, rmtdevice);
                data = new byte[1024];
                recv = s.ReceiveFrom(data, ref Remote);
            } catch (SocketException)
            {
                recv = 0;
            }
            if (recv > 0)
            {
                return recv;
            } else
            {
                retry++;
                if (retry > 4)
                {
                    return 0;
                }
            }
        }
    }

    public RetryUdpClient()
    {
        string input, stringData;
```

```
int recv;
IPEndPoint ipep = new IPEndPoint(
    IPAddress.Parse("127.0.0.1"), 9050);

Socket server = new Socket(AddressFamily.InterNetwork,
    SocketType.Dgram, ProtocolType.Udp);

int sockopt = (int)server.GetSocketOption(SocketOptionLevel.Socket,
    SocketOptionName.ReceiveTimeout);
Console.WriteLine("Default timeout: {0}", sockopt);
server.SetSocketOption(SocketOptionLevel.Socket,
    SocketOptionName.ReceiveTimeout, 3000);
sockopt = (int)server.GetSocketOption(SocketOptionLevel.Socket,
    SocketOptionName.ReceiveTimeout);
Console.WriteLine("New timeout: {0}", sockopt);

string welcome = "Hello, are you there?";
data = Encoding.ASCII.GetBytes(welcome);

recv = SndRcvData(server, data, ipep);
if (recv > 0)
{
    stringData = Encoding.ASCII.GetString(data, 0, recv);
    Console.WriteLine(stringData);
} else
{
    Console.WriteLine("Unable to communicate with remote host");
    return;
}

while(true)
{
    input = Console.ReadLine();
    if (input == "exit")
        break;
    recv = SndRcvData(server, Encoding.ASCII.GetBytes(input), ipep);
    if (recv > 0)
    {
        stringData = Encoding.ASCII.GetString(data, 0, recv);
        Console.WriteLine(stringData);
    } else
        Console.WriteLine("Did not receive an answer");
}
Console.WriteLine("Stopping client");
server.Close();
}
```

```
public static void Main()
{
    RetryUdpClient ruc = new RetryUdpClient();
}
}
```

在与远程服务器进行的所有通信中，**RetryUdpClient**程序使用了**SndRcvData()**方法。一般来说，首先向远程主机发送一个问候消息，如果**SndRcvData()**接收到消息，**Main()**方法通过测试返回值，就能够知道。请看下面的代码：

```
recv = SndRcvData(server, data, ipep);
if (recv > 0)
{
    stringData = Encoding.ASCII.GetString(data, 0, recv);
    Console.WriteLine(stringData);
} else
{
    Console.WriteLine("Unable to communicate with remote host");
    return;
}
```

如果返回值大于零，则表明已经接收到消息，在全局数据缓冲区中含有所接收到的消息，返回值表示所接收到的消息的长度。如果返回值为零，则说明没有从远程主机中接收到消息，程序会做出相应处理。

这个消息循环用**SndRcvData()**方法发送由控制台输入的消息，等待响应。当然，在消息循环中必须含有检测**SndRcvData()**方法返回值的代码行，并做出是否消息已被正确接收的判断。如果远程主机没有正确接收消息，则在控制台上显示信息提示客户。

程序测试

接下来我们将要进行在网络程序设计中很有趣的一件事，即对我们刚刚创建的重传机制进行测试。首先测试一下在没有服务器的情况下是否运转正常。不启动相应的UDP服务器，我们运行**RetryUdpClient**程序。请看下面的代码：

```
C:\>RetryUdpClient
Default timeout: 0
New timeout: 3000
Attempt #0
Attempt #1
Attempt #2
Attempt #3
Attempt #4
Unable to communicate with remote host
C:\>
```

正如我们所期望的，程序中启动了重试机制并且向服务器发送了5次问候消息。第五次尝试后，则向Main()返回一个零值，然后退出。如果是在网络环境下进行测试，可以使用WinDump或者Analyzer监视重试的数据包的情况。每一次重试企图都生成一个问候数据包，产生数据包的时间间隔就是程序中设置的超时时间。

接下来，在网络上模拟一下丢失数据包的情况，看看重试方法的运转。启动RetryUdpClient程序，进入一组重试循环中，然后在目标机上启动SimpleUdpSrvr。Retry循环将能够检测到服务器，就像从一开始就一切正常一样：

```
C:\>RetryUdpClient
Default timeout: 0
New timeout: 3000
Attempt #0
Attempt #1
Attempt #2
Attempt #3
Attempt #4
Welcome to my test server
This is a test message.
Attempt #0
This is a test message.
Here's another message.
Attempt #0
Here's another message.
exit
Stopping client

C:\>
```

这个测试也如同期望的一样运转良好。在四次从服务器试图接收响应消息失败后，最后一次努力成功地接收到来自服务器的欢迎标语，下面进入正常的通信过程。

最后一个测试可以在消息传输的过程中关闭服务器，看看发生的情况。具体做法如下：在指定的服务器机器上启动SimpleUdpSrvr，在指定的客户机机器上启动RetryUdpClient，这时候一切运转良好（除非真的出现了丢包现象）。在发送了一对测试消息后，按Ctrl+C终止SimpleUdpSrvr程序。接下来试着从RetryUdpClient发送一条消息，运行结果将如下所示：

```
C:\>RetryUdpClient
Default timeout: 0
New timeout: 3000
Attempt #0
Welcome to my test server
First test message
Attempt #0
First test message
Second test message
```

```
Attempt #0
Second test message
Third test message
Attempt #0
Attempt #1
Attempt #2
Attempt #3
Attempt #4
Did not receive an answer
Last test message
Attempt #0
Attempt #1
Attempt #2
Attempt #3
Attempt #4
Did not receive an answer
exit
Stopping client

C:\>
```

如同我们所期望的一样，客户机程序与远程服务器进行通信，发送问候消息，并接收到来自服务器的欢迎标语。前两个消息的发送很正常。当服务器程序终止后，发送第三个测试消息，经过五次努力后，客户机程序放弃并显示指定的错误信息。这时候客户机程序仍然处于活动状态，并能够继续发送消息，但是继续努力的结果也是显示出错信息。

使用上面我们讲述过的重传技术，能够设计出健壮的UDP应用，抵制住一般的网络拥塞故障，并能从远程主机中获取数据。

完整的UDP应用

前面的几节内容我们讨论了UDP协议的一些特点，以及如何针对这些特点进行编程，并列出了范例程序示范专门的UDP编程特性。这些范例都非常好，非常适合教科书使用，但是在现实生活中，如果设计UDP应用，仅有这些范例是远远不够的。

本节内容将把前面我们讲述过的UDP的一些特点进行综合，并讲述一个完整的UDP应用范例，在这个范例中将用到本章涉及到的所有技术。

通过监视错误代码捕获多个异常

到目前为止，我们已经了解了，如果数据缓冲区对于流入的数据来说太小的时候以及当一组ReceiveTimeout时间都超时如何捕获SocketExceptions，但是，如何在一个程序中同时监测这两种状况呢？

由于两个事件产生同一个Exception，无法根据Exception编写不同的catch代码段分别进行处理，而且必须区分是哪一个事件导致的Exception。因此必须根据有关套接字错误代码信

息进行处理。这一小节的内容主要讲述如何判断导致SocketException的原因以及处理的方式。

当Socket对象生成一个SocketException时，是由于底层的WinSock套接字产生了某种错误状态。确切的错误状况可以使用ErrorCode属性从SocketException对象中获取。ErrorCode属性含有产生SocketException的WinSock错误消息的数值表示，有许多WinSock错误代码，表6.1中列出的是在网络编程中经常遇到的一些代码。

表6.1 WinSock错误代码

错误代码	说明
10004	方法调用中断
10013	未被许可
10014	地址错误
10022	无效参数
10024	打开的套接字太多
10035	资源暂时难以获得
10036	马上操作
10037	已经开始操作
10038	无效套接字操作
10039	未输入目标地址
10040	消息过长
10041	错误的协议类型
10042	协议选项错误
10043	所用协议不支持
10044	套接字类型不支持
10045	操作不支持
10046	协议族不支持
10047	协议族不支持地址族
10048	地址已用
10049	地址无法分配
10050	网络关闭
10051	网络故障
10052	网络故障导致连接复位
10053	软件导致连接终止

(续表)

错误代码	说明
10054	连接复位
10055	缓冲区空间不够
10056	套接字已经连接
10057	套接字没有连接
10058	套接字关闭后不能发送数据
10060	连接超时
10061	拒绝连接
10064	主机关机
10065	没有到主机的路由
10067	进程太多
10091	网络子系统故障
10101	安全关闭进程
10109	类型没有找到
11001	未找到主机

在这里，我们只需要关心两个错误代码：

- 10040，接收数据缓冲区过小时，需要关心这个代码
- 10054，ReceiveFrom()方法超时，需要关心这个代码

要设计一个方法能够同时监视这两种错误情形，必须在catch代码块中编写一些代码以检测SocketException对象的ErrorCode属性，并做出相应的处理。请看下面的代码：

```
    } catch (SocketException e)
    {
        if (e.ErrorCode == 10054)
        {
            // statements to handle ReceiveFrom() timeout
        }
        else if (e.ErrorCode == 10040)
        {
            //statements to handle data buffer overflow
        }
    }
```

在catch过滤器中创建一个SocketException对象的实例（e），就可以使用这个实例的ErrorCode属性对产生Exception的确切的错误代码进行检查。我们可以在需要的时候随时随

地编写代码检查任何一种错误情形。这个例子对本章我们讨论过的两种特殊的错误情形进行检查：**ReceiveFrom()**超时和数据缓冲区越界。在实际的网络编程中，可能会需要对其他的错误故障进行检查。

对于频繁产生**SocketException**错误的程序，这里我们给出一个解决的小窍门，即可以使用**ErrorCode**属性确定导致**SocketException**的原因，并对其进行相应编码，甚至可以使用**Message**属性查看错误代码的有关信息：

```
    } catch (SocketException e)
    {
        Console.WriteLine("Socket error {0}: {1}", e.ErrorCode, e.Message)
    }
```

完整的客户机程序

程序清单6.12中所示的**BestUdpClient.cs**程序，使用了**AdvSndRecvData()**方法，这个方法综合了本章讲述过的所有的解决方案。请看下面的程序代码。

程序清单6.12 **BestUdpClient.cs**程序

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class BestUdpClient
{
    private byte[] data = new byte[1024];
    private static IPEndPoint sender = new IPEndPoint(IPAddress.Any, 0);
    private static EndPoint Remote = (EndPoint)sender;
    private static int size = 30;

    private static int AdvSndRecvData(Socket s, byte[] message,
                                      EndPoint rmtdevice)
    {
        int recv = 0;
        int retry = 0;

        while (true)
        {
            Console.WriteLine("Attempt #{0}", retry);
            try
            {
                s.SendTo(message, message.Length, SocketFlags.None, rmtdevice);
                data = new byte[size];
                recv = s.ReceiveFrom(data, ref Remote);
            } catch (SocketException e)
```

```
{
    if (e.ErrorCode == 10054)
        recv = 0;
    else if (e.ErrorCode == 10040)
    {
        Console.WriteLine("Error receiving packet");
        size += 10;
        recv = 0;
    }
}

if (recv > 0)
{
    return recv;
} else
{
    retry++;
    if (retry > 4)
    {
        return 0;
    }
}
}

}

public static void Main()
{
    string input, stringData;
    int recv;
    IPEndPoint ipep = new IPEndPoint(
        IPAddress.Parse("127.0.0.1"), 9050);

    Socket server = new Socket(AddressFamily.InterNetwork,
        SocketType.Dgram, ProtocolType.Udp);

    int sockopt = (int)server.GetSocketOption(SocketOptionLevel.Socket,
        SocketOptionName.ReceiveTimeout);
    Console.WriteLine("Default timeout: {0}", sockopt);
    server.SetSocketOption(SocketOptionLevel.Socket,
        SocketOptionName.ReceiveTimeout, 3000);
    sockopt = (int)server.GetSocketOption(SocketOptionLevel.Socket,
        SocketOptionName.ReceiveTimeout);
    Console.WriteLine("New timeout: {0}", sockopt);

    string welcome = "Hello, are you there?";
    data = Encoding.ASCII.GetBytes(welcome);

    recv = AdvSndRcvData(server, data, ipep);
```

```
        if (recv > 0)
        {
            stringData = Encoding.ASCII.GetString(data, 0, recv);
            Console.WriteLine(stringData);
        } else
        {
            Console.WriteLine("Unable to communicate with remote host");
            return;
        }
    }

    while(true)
    {
        input = Console.ReadLine();
        if (input == "exit")
            break;
        recv = AdvSndRcvData(server, Encoding.ASCII.GetBytes(input), ipep);
        if (recv > 0)
        {
            stringData = Encoding.ASCII.GetString(data, 0, recv);
            Console.WriteLine(stringData);
        } else
            Console.WriteLine("Did not receive an answer");
    }
    Console.WriteLine("Stopping client");
    server.Close();
}
}
```

AdvSndRcvData()方法在一个调用中实现了多种功能:

- 从UDP套接字发送一条消息到远程目标机器。
- 在规定的时间内等待来自服务器的响应。
- 如果没有收到响应消息（在规定的时间内），则重新试几次。
- 如果返回消息比预先定义好的数据缓冲区大，则将定义缓冲区大小的字节数组加大，重新发送原始消息，并等待响应。

读者可能已经注意到我们在类中将size变量设置为一个静态变量，而不是在AdvSndRcvData()方法中设置。这种使用方式能够在两次AdvSndRcvData()调用之间维持一个较大的缓冲区。当然，如果读者认为自己的应用中仅仅可能有一两个大数据包，也可以将size变量放在方法中，每一次方法调用都将该值复位为30。

可以使用SimpleUdpSrvr对BestUdpClient进行测试，如同在RetryUdpClient.cs程序中对读者的建议一样，最好尝试启、停服务器的不同组合，看看它们对客户机程序的影响。

提示：从上述例子中我们可以看出，在程序中带有一些小型函数方法来实现某项网络功能非常方便，是值得提倡的良好的编程习惯。这样一来，我们就能够为每一次传输的错误状况进行详细检测，从而避免每次都发送或者接收消息。

小结

本章主要讲述了通过网络发送UDP消息到远程主机时所需要的一些技术。为了使用UDP协议，Socket对象必须使用SocketType.Dgram和ProtocolType.Udp参数。一旦UDP Socket对象被创建，就可以使用SendTo()和ReceiveFrom()方法发送和接收UDP数据包。

因为UDP是一种无连接的协议，因此，必须为每一个数据包指定目标地址；也可以使用标准的Connect()方法一次性指定目标地址，然后使用Receive()和Send()方法进行数据的接收和发送。

UDP通信的一个重要的优点是保持了消息的边界。所有在一个SendTo()方法中发送的数据都被目标主机上的一个ReceiveFrom()方法接收。另一方面，UDP的一个不足就是所有的数据包不能保证成功到达目标主机。

对丢失的数据包进行处理的方法之一是采用消息重传机制。每一个发送到目的地的数据包都将得到一个响应消息。如果在规定的时间内没有接收到响应消息，原始数据包将被重发。如果连续几次重发之后仍然没有响应消息返回，应用程序应该提醒顾客有可能是网络故障。ReceiveTimeout套接字选项能够用来设置套接字的超时时间。

UDP的另一个不足是数据缓冲区小。当ReceiveFrom()方法从流入数据包读取数据时，必须使用足够大的数据缓冲区来接收所有的数据，任何没有接收的数据都要丢失。在每一次出现Exception时增加缓冲区尺寸，可以采用几何方法来防止数据的丢失。

使用try-catch程序段能够监视ReceiveFrom()方法，而且能够测试出现故障的套接字触发的Exception。SocketException的ErrorCode属性用于判断底层的套接字错误代码。如果错误代码是10040，说明所接收到的消息比接收缓冲区大，必须重发。如果错误代码为10054，则说明远程客户机已经与服务器断开，而且连接会话可能被终止了。

下一章我们将讲述在网络编程中如何使用异步套接字方法，这样一来事件驱动的Windows程序就能够利用有关网络编程技术，而不用担心网络方法调用所带来的阻塞问题。

第7章 C#套接字的助手类

- TcpClient类组件和范例
- TcpListener类组件和范例
- UdpClient类组件和范例
- 在网络中传输二进制数据
- 与其他类型主机之间的通信
- 使用数据类在网络上传输复杂对象

前面两章中我们讲述了如何使用底层的Socket类设计TCP和UDP网络应用程序。许多有经验的网络程序员非常熟悉这种类型的程序设计，但是也有一些编程新手感觉Socket类非常难，而且经常混淆。为了简化网络编程，Microsoft公司提供了一套简化版本的助手类，帮助网络程序员设计标准网络应用。

本章主要讲述在网络编程中如何使用这些助手类，并且举例说明在实际的网络环境中的具体应用方式。另外，本章还将讲述如何使用这些类在网络中的主机之间传输数据。

一般来说，应用程序的数据有多种类型，有些类型的数据非常容易传输，有些类型的数据却难以传输。在这一章里，我们将讲述使用助手类在网络上传输复杂二进制数据的技术。

TcpClient类

TcpClient类位于System.Net.Sockets命名空间内，用于在不使用Socket类来实现TCP通信时方便TCP客户机应用的设计。这一小节将讲述TcpClient类，并举例说明该类的具体应用。

TcpClient类的构造函数

我们能够使用三种构造函数格式创建一个TcpClient对象。

TcpClient() 这是第一种格式的构造函数，创建一个新的TcpClient对象，将套接字与本地系统地址和一个随机TCP端口号进行绑定。在默认的TcpClient对象创建后，它必须使用Connect()方法与远程设备连接，在后面的“TcpClient类方法”一节中我们将要讲述这个问题，这里我们仅仅说明构造函数的格式并指定一个TCP端口号，请看下面的代码：

```
TcpClient newcon = new TcpClient();  
newcon.Connect("www.ispnet.net", 8000);
```

TcpClient(IPEndPoint localEP) 这是第二种格式的构造函数，在这种格式中能够指定一个特殊的本地IP地址以及一个特殊的TCP端口号。当设备中装有两块以上的网卡，并且希望从指定的某个网卡中发送数据包时经常使用这种构造函数。另外，TcpClient对象创建后必须使用Connect()方法与远程设备进行连接。请看下面的代码：

```
IPEndPoint iep = new TPEndPoint(TPAddress.Parse("192.168.1.6"), 8000);
TcpClient newcon = new TcpClient(iep);
newcon.Connect("www.isp.net", 8000);
```

TcpClient(String host, int port) 这是第三种格式的构造函数，是最常用的一种构造函数，可以在这个构造函数内指定远程设备，而且不需要使用Connect()方法。远程设备可以通过地址和TCP端口号来指定，地址可以是字符串形式的主机名或者是一个字符串形式的IP地址，TcpClient的构造函数能够将字符串形式的主机名字自动解析成恰当的IP地址。请看下面的代码：

```
TcpClient newcon = new TcpClient("www.isp.net", 8000);
```

一旦TcpClient对象被创建，就可以使用该对象的属性和方法在本地和远程设备之间来回回传输数据。

TcpClient类方法

TcpClient类中有许多有用的属性和方法，在设计TCP客户机程序时，经常要使用这些属性和方法。表7.1中列出的就是TcpClient中的一些方法。

表7.1 TcpClient方法

方法	说明
Close()	关闭TCP连接
Connect()	与远程设备建立TCP连接
Equals()	判断两个TcpClient对象是否相等
GetHashCode()	获取在哈希函数中能够使用的一个哈希码
GetStream()	获取一个能够发送和接收数据的Stream对象
GetType()	获取当前实例的类型
ToString()	将当前实例转换为一个Stream对象

在前面我们已经讲述过，Connect()方法将TcpClient对象与一个远程设备进行连接。一旦远程设备连接上，GetStream()对象就对NetworkStream对象进行赋值来发送和接收数据，请看下面的代码：

```
TcpClient newcon = new TcpClient()
newcon.Connect("www.ispnet.net", 8000);
NetworkStream ns = new NetworkStream(newcon);
```

在这个NetworkStream对象被创建后，我们就可以使用Read()和Write()方法进行数据的接收和发送（在第5章“面向连接的套接字”中已经讲述过这方面的有关内容）。

在TcpClient类中除了这些方法外，还有一些有用的属性，在表7.2中进行了描述。利用这些属性能够对TcpClient对象底层Socket对象的选项进行设置。

表7.2 TcpClient对象属性

属性	说明
LingerState	得到或者设置套接字保持的时间
NoDelay	得到或者设置向未满的TCP缓冲区发送或接收数据的延迟时间
ReceiveBufferSize	得到或者设置TCP接收缓冲区的尺寸
ReceiveTimeout	得到或者设置套接字接收数据的超时时间
SendBufferSize	得到或者设置TCP发送缓冲区的大小
SendTimeout	得到或者设置套接字发送数据的超时时间

简单的客户机程序

之所以设计出**TcpClient**类就是为了使网络编程尽可能简单，读者或许已经想像到，一个基本的**TCP**客户机程序能够仅用几行代码就完成。在程序清单7.1中所示的代码就是一个简单的**TCP**客户机程序，这个客户机程序能够和第5章中的**SimpleTcpSrvr**程序进行交互。

程序清单7.1 TcpClientSample.cs程序

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class TcpClientSample
{
    public static void Main()
    {
        byte[] data = new byte[1024];
        string input, stringData;
        TcpClient server;

        try
        {
            server = new TcpClient("127.0.0.1", 9050);
        } catch (SocketException)
        {
            Console.WriteLine("Unable to connect to server");
            return;
        }

        NetworkStream ns = server.GetStream();

        int recv = ns.Read(data, 0, data.Length);
        stringData = Encoding.ASCII.GetString(data, 0, recv);
        Console.WriteLine(stringData);
    }
}
```

```
        while(true)
        {
            input = Console.ReadLine();
            if (input == "exit")
                break;
            ns.Write(Encoding.ASCII.GetBytes(input), 0, input.Length);
            ns.Flush();

            data = new byte[1024];
            recv = ns.Read(data, 0, data.Length);
            stringData = Encoding.ASCII.GetString(data, 0, recv);
            Console.WriteLine(stringData);
        }
        Console.WriteLine("Disconnecting from server...");
        ns.Close();
        server.Close();
    }
}
```

在这个例子中我们所使用的**TcpClient**构造函数能够自动与指定的远程服务器进行连接，因此远程服务器没有准备好的时候在**try-catch**代码段中适合使用这种构造函数。对于**Socket**方法**Connect()**来说，如果远程服务器没有准备好，将生成一个**SocketException**。

警告：在**try-catch**代码段中使用这种格式的**TcpClient**构造函数时，必须考虑一个问题就是在**try-catch**块中的变量只能在本代码块中使用。因此为了在**try-catch**代码块外也能够使用**TcpClient**对象变量，必须在**try-catch**代码块外面进行变量的声明，如程序清单7.1所示。

NetworkStream对象被创建后，就可以用普通的**Read()**和**Write()**方法对数据进行操作：

```
while(true)
{
    input = Console.ReadLine();
    if (input == "exit")
        break;
    ns.Write(Encoding.ASCII.GetBytes(input), 0, input.Length);
    ns.Flush();

    data = new byte[1024];
    recv = ns.Read(data, 0, data.Length);
    stringData = Encoding.ASCII.GetString(data, 0, recv);
    Console.WriteLine(stringData);
}
```

Read()方法有三个参数：

- 存放所接收到的数据的字节数组（数组中的数据类型为接收数据的类型）
- 在缓冲区放置数据的起始位置

- 数据缓冲区的长度

如同Socket方法Receive()一样, 这个对象的Read()方法也是按照缓冲区的大小来读取数据, 如果缓冲区过小, 留下的数据将继续位于数据流中等待下一次Read()调用。

Write()方法也有三个参数:

- 发送数据的数组
- 从缓冲区中发送数据的起始位置
- 发送数据的长度

说明: 在进行网络编程时, 有一点必须记牢, 那就是TCP不保留消息边界, TcpClient类也有这样一个特点。我们讲述到这里, 想必读者已经意识到这个方法与TCP Socket类中的Receive()方法的处理方式一样, 使用循环来保证从数据流中读取所需的所有数据。

程序测试

我们可以使用在第5章中介绍过的SimpleTcpSrvr(见程序清单5.1)程序对这个新TcpClientSample进行测试。方法非常简单, 只要在指定的设备上启动SimpleTcpSrvr, 然后从同一台机器上的命令行窗口中运行TcpClientSample程序, 或者从网络上其他的机器上运行TcpClientSample程序。切记, 将程序中的IP地址改为实际使用的服务器设备的IP地址。

TcpClientSample的运行过程将与第5章(程序清单5.9)中讲述的NetworkStream-TcpClient程序完全一样, 我们可以在控制台窗口中输入语句, 观察服务器上的显示情况以及服务器对客户机的反馈信息。

如果要观察测试过程中产生的网络流量, 可以使用windump或者analyzer命令进行监视。图7.1所示为一个测试过程的Analyzer的输出。

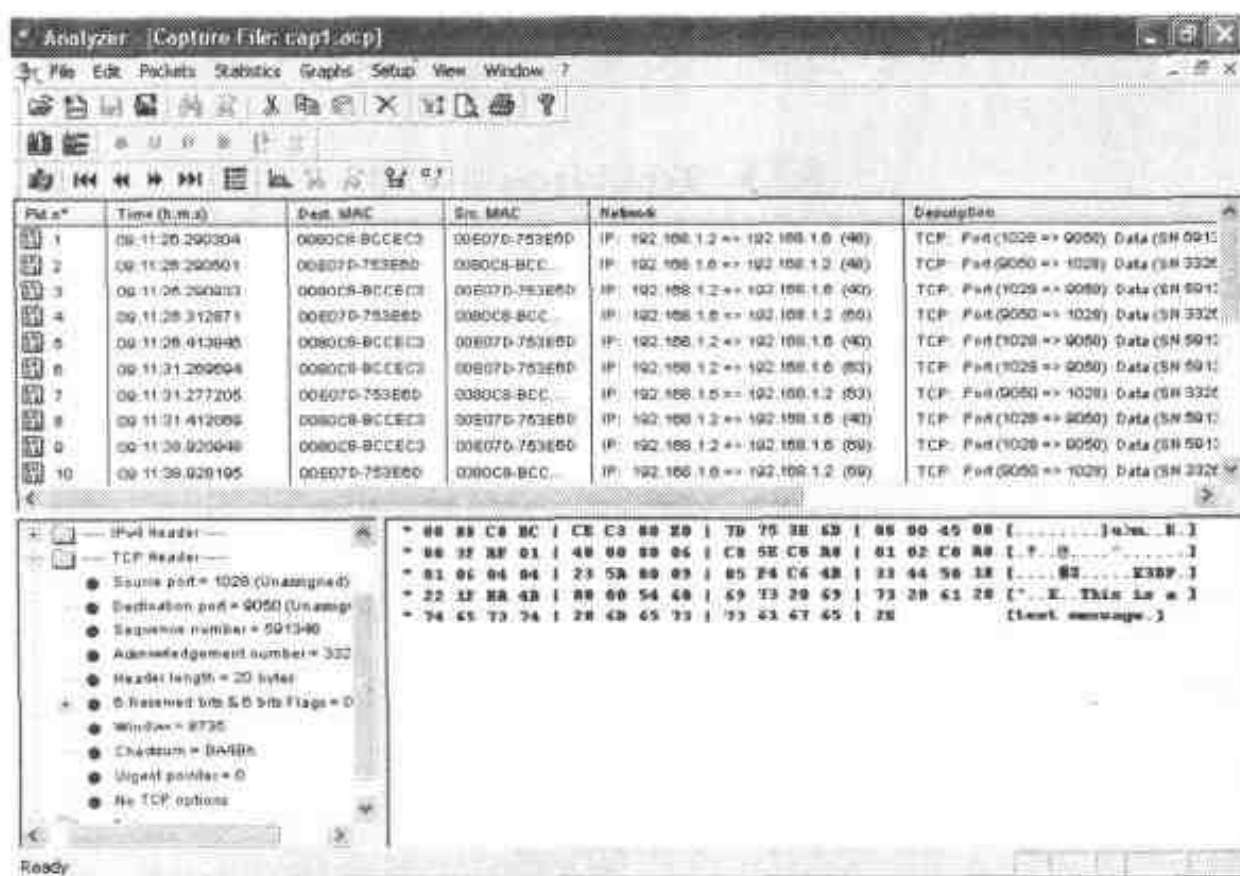


图7.1 对TcpClientSample进行测试时的Analyzer输出

从上面的输出中可以看出, TcpClientSample程序的运转过程与第5章讲述过的SimpleTcp-

Client程序完全一样，SimpleTcpClient程序使用Socket对象。这就说明我们可以使用TcpClient类来代替Socket对象，提高编程效率。

TcpListener类

与TcpClient一样，TcpListener类（也是位于System.Net.Sockets命名空间内）提供了设计TCP服务器程序的一个简单途径。在这一小节内，我们将讲述TcpListener类以及如何在简单的服务器应用中使用这个类。

TcpListener类的构造函数

TcpListener类有三种格式的构造函数：

TcpListener (int port) 这个构造函数与指定的本地端口号进行绑定。

TcpListener (IPEndPoint ie) 这个构造函数与本地EndPoint进行绑定。

TcpListener (IPAddress addr, int port) 这个构造函数与指定的本地IPAddress对象和端口号进行绑定。

与TcpClient类的构造函数不同，TcpListener的构造函数至少需要一个参数：端口号，客户机在这个端口号上收听连接。如果服务器所在的机器有多个网卡，并且希望收听某一个指定的网卡，可以使用IPEndPoint对象指定要收听的网络IP地址和要收听的TCP端口号。在上面讲述的最后一个构造函数中使用IPAddress对象指定所需的IP地址，使用另外一个参数指定端口号。

TcpListener类方法

TcpListener类方法（参见表7.3），用于在所创建的TcpListener对象上执行一定的功能。

表7.3 TcpListener类方法

方法	说明
AcceptSocket()	从端口处接收一个流入的连接并将它赋给一个Socket对象
AcceptTcpClient()	从端口处接收一个流入的连接并将它赋给一个TcpClient对象
Equals()	判断两个TcpListener对象是否相等
GetHashCode()	获取一个能够用于哈希函数中的哈希码
GetType()	得到当前的实例类型
Start()	启动对连接尝试的收听
Stop()	结束对连接尝试的收听（关闭套接字）
ToString()	创建TcpListener对象的字符串表示

生成TcpListener对象并收听流入连接的过程如下：

```
TcpListener server = new TcpListener(IPAddress.Parse("127.0.0.1"), 9050);
server.Start();
TcpClient newclient = server.AcceptTcpClient();
```

Start()方法与在**Socket**类中使用**Bind()**和**Listen()**两个方法功能相似，将套接字与在**TcpListen**构造函数中定义的端点进行绑定，并将**TCP**端口设置为收听模式，准备新连接。**AcceptTcpClient()**方法与**Accept()**类似，接收流入的连接尝试并将它们赋给**TcpClient**对象。

TcpClient对象创建后，与远程设备的所有通信都可以使用这个**TcpClient**对象来实现而不再使用原来的**TcpListener**对象。因此**TcpListener**对象能够用于接收其他的连接并将连接传递给其他的**TcpClient**对象，要关闭**TcpListener**对象，必须使用**Stop()**方法，请看下面代码：

```
server.Stop();
```

说明：如果有打开的客户机连接，在最初的**TcpListener**对象关闭之前不必关闭这些连接，然而，必须使用**Close()**方法逐个关闭**TcpClient**对象。

简单服务器程序范例

现在我们来查看一个使用**TcpListener**类的简单**TCP**服务器程序。在这个范例（程序清单7.2）中，模仿了第5章（程序清单5.1）中所讲述的**SimpleTcpSrvr**程序中的功能，请看下面代码。

程序清单7.2 TcpListenerSample.cs程序

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class TcpListenerSample
{
    public static void Main()
    {
        int recv;
        byte[] data = new byte[1024];

        TcpListener newsock = new TcpListener(9050);
        newsock.Start();
        Console.WriteLine("Waiting for a client...");

        TcpClient client = newsock.AcceptTcpClient();
        NetworkStream ns = client.GetStream();

        string welcome = "Welcome to my test server";
        data = Encoding.ASCII.GetBytes(welcome);
        ns.Write(data, 0, data.Length);
    }
}
```

```
        while (true)
        {
            data = new byte[1024];
            recv = ns.Read(data, 0, data.Length);
            if (recv == 0)
                break;

            Console.WriteLine(
                Encoding.ASCII.GetString(data, 0, recv));
            ns.Write(data, 0, recv);
        }
        ns.Close();
        client.Close();
        newsock.Stop();
    }
}
```

这个TcpListenerSample程序首先创建一个TcpListener对象，使用UDP的端口号9050，Start()方法将这个新对象设置为收听模式。然后AcceptTcpClient()方法等待流入的TCP连接和请求消息并将它赋给一个TcpClient对象，请看下面代码：

```
TcpListener newsock = new TcpListener(9050);
newsock.Start();

TcpClient client = newsock.AcceptTcpClient();
NetworkStream ns = client.GetStream();
```

建立TcpClient对象后，将一个NetworkStream对象赋给这个TcpClient对象，并与远程客户机进行连接。所有的通信都是通过这个NetworkStream对象的Read()和Write()方法实现。在这里要记住一点：必须将所有数据都放入一个字节数组中以便Write()方法使用。同样，所有Read()方法接收到的数据也都要被赋到一个字节数组中。

可以启动TcpListenerSample程序，并将它与前面讲述过的TcpClientSample程序进行连接，可以看出这两个程序的执行过程与在第5章中讲述过的TcpClientSample（程序清单5.1）和SampleTcpClient（程序清单5.2）一样。一般来说，如果在网络上测试这些程序，可以使用WinDump和Analyzer程序监视所生成的网络流量的情况。

StreamReader和StreamWriter类

因为NetworkStream对象使用数据流在网络主机之间传输数据，因此，必须处理数据流内一般的消息识别问题。使用在第5章中我们讲述过的一些标准技术，我们就可以在数据流内对消息进行限定。具体技术如下：

- 发送固定尺寸的消息
- 在消息体前面附加消息的尺寸信息
- 使用消息定界字符

从TcpClient对象中派生出NetworkStream对象后，就可以使用这个NetworkStream创建StreamReader和StreamWriter对象（这部分内容在第5章的“在TCP通信中使用C#数据流”一节中讲述过），这些对象将创建消息定界符号以简化数据在网络中的传输过程。请看下面代码：

```
TcpClient client = new TcpClient("127.0.0.1", 9050);
NetworkStream ns = client.GetStream();
StreamReader sr = new StreamReader(ns);
StreamWriter sw = new StreamWriter(ns);

sw.WriteLine("This is a test");
sw.Flush();
string data = sr.ReadLine();
```

默认状态下，StreamReader和StreamWriter类使用换行符作为消息分界符号来区分消息。ReadLine()和WriteLine()方法二者都使用String对象，文本消息能够使用String类创建并发送而不必使用大容量的数据字节数组。

UdpClient类

UdpClient类能够简化网络程序员的UDP网络程序设计工作。这一节我们将要描述UdpClient类以及它的方法，并使用这个类创建一个简单的UDP服务器和客户机程序。

UdpClient类的构造函数

UdpClient类有四种格式的构造函数：

UdpClient() 这种格式的构造函数创建一个没有与任何指定的地址或端口绑定的新UdpClient实例。

UdpClient(int port) 这种格式将所构造的新UdpClient对象与指定的UDP端口号进行绑定。

UdpClient(IPEndPoint iep) 这种格式的构造函数将新创建的UdpClient对象与指定的本地IP地址和端口号进行绑定。

UdpClient(string host, int port) 这种格式的构造函数将新创建的UdpClient对象与任意IP地址和端口号进行绑定，并将它与指定的远程IP地址和端口进行联合。

UdpClient构造函数与相对应的TcpClient函数工作原理一样，可以使用系统进行UDP端口的选择，也可以在构造函数中选择一个特定的端口。如果UDP应用必须从特定的端口接收数据，则要在UdpClient构造函数中定义所使用的端口。

UdpClient对象创建后，可以使用不同的函数设置底层的套接字和传输数据。

UdpClient类方法

UdpClient类提供的类方法用于控制数据流入和流出UDP套接字，在表7.4中对可用方法进行了描述。

表7.4 UdpClient类方法

方法	说明
Close()	关闭底层套接字
Connect()	指定用于发送和接收数据的远程IP端点
DropMulticastGroup()	从UDP组播组中删除套接字
Equals()	判断两个Udp对象是否相等
GetHashCode()	为UdpClient对象获取一个哈希代码
GetType()	获取当前对象的类型
JoinMulticastGroup()	将套接字添加到一个UDP组播组中
Receive()	从套接字接收数据
Send()	从套接字向远程主机发送数据
ToString()	创建UdpClient对象的字符串表示形式

说明：通过JoinMulticastGroup()和DropMulticastGroup()方法，能够使用IP组播技术进行UDP编程。这个特点在本书第10章“IP组播”一章中介绍。

在程序中使用UdpClient类

UdpClient类中的Receive()和Send()两个方法略有不同，从而使得这两个方法与Socket方法ReceiveFrom()和SendTo()不同。

Receive()方法

UdpClient类使用Receive()方法在指定的接口和端口上接收UDP数据包。只有一种Receive()格式：

```
byte[] Receive(ref IPEndPoint iep)
```

Receive()方法从UdpClient构造函数中所指定的IP地址和UDP端口处接收UDP数据包（系统自己指定或者构造函数指定均可），下面我们来看一下，Receive()方法格式与标准UDP Socket对象的ReceiveFrom()方法的不同。

鉴于触发的原因，从套接字中接收到的数据在这个方法调用中没有放入字节数组中，而是由方法返回，因此，必须指定一个空字节数组用于接收数据。

UdpClient方法Receive()和Socket方法ReceiveFrom()之间的第二个不同是，远程主机信息的返回方式不同。ReceiveFrom()将远程主机信息放置在一个EndPoint对象中，而Receive()却使用IPEndPoint对象。这样一来，对程序员来说，获取远程主机的IP地址和UDP端口号就容易多了。

下面这一小段代码示范了在一个UDP应用中如何使用Receive()方法：


```
IPEndPoint ipep = new IPEndPoint(IPAddress.Any, 9050);
UdpClient newsock = new UdpClient(ipep);
byte[] data = new byte[1024];
IPEndPoint ipep2 = new IPEndPoint(IPAddress.Any, 0);
data = host.Receive(ref ipep2);
Console.WriteLine("The remote host is: {0}, port {1}',
    ipep2.Address, ipep2.Port);
Console.WriteLine(Encoding.ASCII.GetString(data));
```

在这个代码小段中，从网络上接收UDP数据包，端口号为9050，并且在控制台上显示出来。

说明：Receive()方法的优点是当所接收到的数据量超过缓冲区的大小时能够容下，Socket对象在遇到这种情况时，产生一个SocketException，而UdpClient则返回一个足够大的缓冲区处理所接收到的数据。结果就是：这个特点轻而易举地解决了大量程序设计上的麻烦，提高了编程效率。

Send()方法

使用Send()方法向远程主机发送数据时有二种格式：

Send (byte[] data, int sz) 这种格式向默认远程主机发送大小为sz的字节数组数据。要使用这种格式，必须用UdpClient构造函数或者Connect()方法（见下面代码）指定一个默认的远程UDP主机：

```
UdpClient host = new UdpClient("127.0.0.1", 9050);
```

Send (byte[] data, int sz, IPEndPoint iep) 这种格式向远程主机发送大小为sz的字节数组数据，远程主机由iep指定。

Send (byte[] data, int sz, string host, int port) 这种格式向主机host在端口port发送大小为sz的字节数组数据。

提示：如果要设计的UDP应用不需要在特定的UDP端口收听流入数据包，则可以使用UdpClient构造函数（指定远程主机信息），然后使用Receive()和Send()方法在远程主机之间来回传输数据。

简单UdpClient服务器程序范例

在程序清单7.3中的UdpSrvrSample.cs程序，示范了如何在服务器环境下使用UdpClient类方法。请看下面代码。

程序清单7.3 UdpSrvrSample.cs程序

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class UdpSrvrSample
{
```

```
public static void Main()
{
    byte[] data = new byte[1024];
    IPEndPoint ipep = new IPEndPoint(IPAddress.Any, 9050);
    UdpClient newsock = new UdpClient(ipep);

    Console.WriteLine("Waiting for a client...");

    IPEndPoint sender = new IPEndPoint(IPAddress.Any, 0);

    data = newsock.Receive(ref sender);

    Console.WriteLine("Message received from {0}:", sender.ToString());
    Console.WriteLine(Encoding.ASCII.GetString(data, 0, data.Length));

    string welcome = "Welcome to my test server";
    data = Encoding.ASCII.GetBytes(welcome);
    newsock.Send(data, data.Length, sender);

    while(true)
    {
        data = newsock.Receive(ref sender);

        Console.WriteLine(Encoding.ASCII.GetString(data, 0, data.Length));
        newsock.Send(data, data.Length, sender);
    }
}
```

UdpSrvrSample程序从**IPEndPoint**对象中派生出**UdpClient**对象，在服务器上指定任意IP地址并使用UDP端口9050。这个程序使用**Receive()**方法，直接等待从任意远程客户机流入的UDP数据包，请看下面代码：

```
IPEndPoint sender = new IPEndPoint(IPAddress.Any, 0);
data = newsock.Receive(ref sender);
```

发送方的变量存放客户机的IP信息，这个信息在将消息发送回客户机时要用到，请看下面代码：

```
newsock.Send(data, data.Length, sender);
```

因为**UdpClient**对象不知道目标主机的IP信息，因此**Send()**方法的每一次调用都要指定目标主机的IP信息。

说明：如同我们在这个**UdpSrvrSample**程序中看到的一样，数据字节数组在每一次**Receive()**方法调用之后不需要复位为原来的尺寸，这是**UdpClient**类的一大特点。

简单UdpClient客户机程序范例

接下来我们讲述与上面的服务器程序相对应的客户机程序**UdpClientSample.cs**（见程序清单7.4），在这个程序中示范了如何在一个UDP客户机应用中使用**UdpClient**类方法。

程序清单7.4 UdpClientSample.cs程序

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class UdpClientSample
{
    public static void Main()
    {
        byte[] data = new byte[1024];
        string input, stringData;
        UdpClient server = new UdpClient("127.0.0.1", 9050);

        IPEndPoint sender = new IPEndPoint(IPAddress.Any, 0);

        string welcome = "Hello, are you there?";
        data = Encoding.ASCII.GetBytes(welcome);
        server.Send(data, data.Length);
        data = server.Receive(ref sender);

        Console.WriteLine("Message received from {0}:", sender.ToString());
        stringData = Encoding.ASCII.GetString(data, 0, data.Length);
        Console.WriteLine(stringData);

        while(true)
        {
            input = Console.ReadLine();
            if (input == "exit")
                break;

            server.Send(Encoding.ASCII.GetBytes(input), input.Length);
            data = server.Receive(ref sender);
            stringData = Encoding.ASCII.GetString(data, 0, data.Length);
            Console.WriteLine(stringData);
        }
        Console.WriteLine("Stopping client");
        server.Close();
    }
}
```

因为这个**UdpClientSample**程序不需要在指定的UDP端口收听流入的消息，因此我们可以使用全内置的构造函数格式指定远程主机的IP地址和UDP端口号，请看下面代码：

```
UdpClient server = new UdpClient("127.0.0.1", 9050);
```

当然，如果客户机正与一个远程UDP服务器连接，要记住在构造函数中设置IP地址或远程服务器的主机名而不是回退地址。

创建UdpClient对象后，发送方法Send()向服务器发送问候信息：

```
server.Send(data, data.Length);
```

在这里，我们注意到由于目标主机的信息已经在UdpClient构造函数中指定过，因此，不需要在这个Send()方法中再做指定。

范例程序的测试

编译过UdpSrvrSample和UdpClientSample程序后，可以在同一台机器上或者在网络上对这两个程序进行测试。

测试输出很像第6章“无连接套接字”中讲述过的SimpleUdpClient程序（见程序清单6.2）。每一个由Send()方法发送的消息都作为一个独立的UDP数据包，并且使用单独的Receive()方法调用读取数据。

因为UdpClient类使用UDP数据包进行数据的传送，也就存在与UDP Socket对象同样的问题（参见本书第6章），因此在进行程序设计时必须考虑到这点，通常我们可以使用在第6章中讲述的一些技术。

警告：使用UDP Socket对象时存在一个使用UdpClient对象中遇不到的问题，即数据的丢失问题。如果提供给UdpClient的Receive()方法的数据缓冲区太小以至于无法容纳流入的数据，则该方法返回大缓冲区以匹配数据的大小，没有数据丢失的现象发生，也没有SocketException产生。

在网络上传输数据

到目前为止，我们已经介绍了使用Socket的助手类如何有效地在网络上从一个设备到另一个设备之间传输文本数据，一般来说，对于大多数应用这些技术已经足够了，但是对于一些其他的应用来说，可能需要另外的一些功能处理非文本数据，比如二进制数据和多种类型的数据组合等。

在这一节中，我们将讲述一些在网络中传输不同类型的二进制数据的技术。在异构系统之间设计通信程序时，了解二进制数据在设备中的存储方式以及在网络中的传输原理非常重要。同时本节将讲述如何在网络上的设备之间传输复杂数据类型（比如类中的数据元素等等）。

二进制数据的传输

不论是进行TCP还是UDP通信，在网络中传输二进制数据都是非常复杂的一件事，存在许多因素将导致错误的发生，因此，在设计的过程中必须考虑周全。在这一节里我们将主要描述在设备之间传输二进制数据时的建议。

二进制数据的表达式

或许在网络中传输二进制数时，关键就是数据的表示形式，不同类型的机器表示二进制数据的形式都不一样，因此，在进行设计时，必须保证发送方的二进制数据能够正确地被

接收方机器识别出来。

使用Microsoft Windows操作系统的机器（Intel芯片或者兼容芯片），通常各种数据类型都使用同一套模式存储信息，因此，在向非Windows平台的远程主机发送二进制数据时，必须了解数据表示的含义，表7.5中列出了在C#语言中使用的二进制数据类型。

表7.5 C#二进制数据类型

数据类型	字节	说明
sbyte	1	- 128到127的有符号整数
byte	1	0到255之间的无符号整数
short	2	- 32 768到32 768之间的有符号短整数
ushort	2	0~65 535之间的无符号短整数
int	4	- 2 147 483 648到2 147 483 647的标准整数
uint	4	0到4 294 967 295之间的无符号整数
long	8	- 9 223 372 036 854 775 808到9 223 372 036 854 775 807之间的有符号较大整数
ulong	8	0到18 446 744 073 709 551 615之间的无符号较大整数
float	4	1.5×10^{-45} 到 $3.4 \times 1,038$ 之间的浮点小数（精度为7位）
double	8	5.0×10^{-324} 到 1.7×10^{308} 之间的浮点小数（精度为15~16位）
decimal	16	1.0×10^{-28} 到 7.9×10^{28} 之间的高精度浮点小数（精度为28~29位）

每一个二进制数据的数据类型在使用Send()和SendTo()方法发送之前都必须转换为一个以列为主的字节数组，幸运的是，.NET中提供了专门实现这个功能的一个类：BitConverter类。

二进制数据类型的转换

在第5章的VarTcpClient程序（见程序清单5.8）和VarTcpSrvr程序（见程序清单5.7）中我们已经看到，.NET提供了类BitConverter，该类能够将二进制数据类型转换为字节数组（反之也行），这个类对于在网络上远程主机之间准确传输二进制数据至关重要。

二进制数据类型的发送

BitConverter类的方法GetBytes()将标准二进制数据类型转换为一个字节数组。请看下面代码：

```
int test = 1990;
byte[] data = BitConverter.GetBytes(test);
newsock.Send(data, data.Length);
```

这几行简单的代码说明了如何将一个标准的4个字节的整型数转换为一个4个字节的字

节数组，在Write()方法和Send()方法调用中向远程设备转发数据时要用到这一小段程序。

警告：如果是使用TCP协议发送二进制数据，就不能使用StreamWriter或者StreamReader类，因为这两个类中使用字符串数据，而不是二进制数据，二进制数据中的“0”将会影响字符串的转换。

说明：为二进制数据类型创建字节数组时，在字节数组中必须分配足够的空间来容纳所有二进制数据类型的字节数。如果转换后的二进制数据没有发送完，接收端就无法将接收到的数据重新组合成原来的数据类型。

二进制数据类型的接收

在上一小节的说明中，我们提到过接收程序必须能够将所接收的二进制数据类型转换为原始二进制数据类型，这个功能也由BitConverter来完成。在表7.6中列出了BitConverter类可用的转换方法。

表7.6 BitConverter类中进行数据转换的方法

方法	说明
ToBoolean()	将单字节的字节数组转换为Boolean值
ToChar()	将双字节的字节数组转换为采用双字节对字符进行编码的字符值
ToDouble()	将8字节的字节数组转换为双精度浮点数
ToInt16()	将双字节的字节数组转换为16位有符号整型数值
ToInt32()	将4字节的字节数组转换为32位有符号整型数值
ToSingle()	将4字节的字节数组转换为单精度浮点数值
ToString()	将字节数组中所有的字节转换为表示该二进制数据的十六进制数值的字符串
ToUInt16()	将双字节字节数组转换为16位无符号整型数值
ToUInt32()	将4字节字节数组转换为32位无符号整型数值
ToUInt64()	将8字节的字节数组转换为64位无符号整型数值

上述所有的转换方法调用格式都一样，请看下面代码：

```
BitConverter.ToInt16(byte[] data, int offset)
```

在上面方法中，第一个参数是要转换的字节数组，第二个参数是在这个字节数组中进行转换的位置。在这里要记住，方法本身知道在字节数组中需要多少字节进行转换。

只要所接收到的字节数组被转换成二进制数据类型，在程序中就能够像其他数据类型一样使用，请看下面代码：

```
double total = 0.0;
byte[] data = newsock.Receive(ref sender);
double test = BitConverter.ToDouble(data, 0);
total += test;
```

范例程序

使用UDP数据包发送二进制信息相当容易，可以设想为每一条消息发送一个数值（在后面的章节中我们将会接触到多值的情况）。因为UDP保留了消息的边界，因此只要收到数据包，我们就可以保证消息的惟一性。设想，服务器了解数据包中二进制数据的类型，那么将所接收到的消息解码为原来的二进制数值就非常简单。

在程序清单7.5中列出了一个从网络中读取二进制数据的UDP服务器程序的范例，请看代码。

程序清单7.5 BinaryUdpSrvr.cs程序

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class BinaryUdpSrvr
{
    public static void Main()
    {
        byte[] data = new byte[1024];
        IPEndPoint ipep = new IPEndPoint(IPAddress.Any, 9050);
        UdpClient newsock = new UdpClient(ipep);

        Console.WriteLine("Waiting for a client...");

        IPEndPoint sender = new IPEndPoint(IPAddress.Any, 0);

        data = newsock.Receive(ref sender);

        Console.WriteLine("Message received from {0}:", sender.ToString());
        Console.WriteLine(Encoding.ASCII.GetString(data, 0, data.Length));

        string welcome = "Welcome to my test server";
        data = Encoding.ASCII.GetBytes(welcome);
        newsock.Send(data, data.Length, sender);

        byte[] data1 = newsock.Receive(ref sender);
        int test1 = BitConverter.ToInt32(data1, 0);
        Console.WriteLine("test1 = {0}", test1);

        byte[] data2 = newsock.Receive(ref sender);
        double test2 = BitConverter.ToDouble(data2, 0);
        Console.WriteLine("test2 = {0}", test2);

        byte[] data3 = newsock.Receive(ref sender);
        int test3 = BitConverter.ToInt32(data3, 0);
        Console.WriteLine("test3 = {0}", test3);

        byte[] data4 = newsock.Receive(ref sender);
```

```

        bool test4 = BitConverter.ToBoolean(data4, 0);
        Console.WriteLine("test4 = {0}", test4.ToString());

        byte[] data5 = newsock.Receive(ref sender);
        string test5 = Encoding.ASCII.GetString(data5);
        Console.WriteLine("test5 = {0}", test5);

        newsock.Close();
    }
}

```

在这个BinaryUdpSrvr程序中使用了Receive()方法等待来自远程客户机的问候消息，在收到问候消息后即向客户机发送欢迎标语。

接下来，BinaryUdpSrvr程序希望从远程客户机中接收一行信息，这一行信息中含有五条独立的消息，每一条消息都含有一种与其他消息不同类型的数据，消息的顺序非常关键，因为要根据消息的位置将这五条消息还原为原来的消息形式。当然，使用UDP协议，不能保证数据包能够到达目的地，因此，从这一点上来说，这个范例在真正的网络程序设计中并不实用，借鉴意义不大。使用这种技术时，必须特别当心，以防数据包的丢失。

在这个程序中我们必须注意，BitConverter类中含有将二进制代码转换为二进制数据类型的方法，但并不是转换为文本数据类型。BitConverter有ToString()方法，但是这个方法的功能与我们想像的不一样，该方法不是将二进制数据转换为能够显示并打印的字符串形式，而是将二进制代码转换成十六进制形式，例如，请看下面代码：

```

string data = "this is a test";
string test = BitConverter.ToString(Encoding.ASCII.GetBytes(data));
Console.WriteLine("data = '{0}'", data);
Console.WriteLine("test = '{0}'", test);

```

输出结果如下：

```

C:\>test
data = 'this is a test'
test = '74-68-69-73-20-69-73-20-61-20-74-65-73-74'

C:\>

```

说明：BitConverter类的方法ToString()能够将字节数据方便地转换为其十六进制的数值，但是，如果希望显示转换后的文本字符串，则必须使用Encoding.ASCII.GetString()方法，这个方法在前面我们已经讲述过（参见程序清单7.5）。

在程序清单7.6中的BinaryUdpClient程序，是与BinaryUdpSrvr程序功能非常相近的一个程序，该程序向服务器程序发送5种类型的数据，请看下面代码。

BinaryUdpClient程序使用Send()方法向服务器发送问候消息（在UdpClient构造函数中指定了服务器地址）并等待来自服务器的欢迎消息。一旦接收到欢迎消息，则立即发送5个消息系列，每一个消息中均含有一种不同数据类型的数据。

程序清单7.6 BinaryUdpClient.cs程序

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class BinaryUdpClient
{
    public static void Main()
    {
        byte[] data = new byte[1024];
        string stringData;
        UdpClient server = new UdpClient("127.0.0.1", 9050);

        IPEndPoint sender = new IPEndPoint(IPAddress.Any, 0);

        string welcome = "Hello, are you there?";
        data = Encoding.ASCII.GetBytes(welcome);
        server.Send(data, data.Length);

        data = new byte[1024];
        data = server.Receive(ref sender);

        Console.WriteLine("Message received from {0}:", sender.ToString());
        stringData = Encoding.ASCII.GetString(data, 0, data.Length);
        Console.WriteLine(stringData);

        int test1 = 45;
        double test2 = 3.14159;
        int test3 = -1234567890;
        bool test4 = false;
        string test5 = "This is a test.";

        byte[] data1 = BitConverter.GetBytes(test1);
        server.Send(data1, data1.Length);

        byte[] data2 = BitConverter.GetBytes(test2);
        server.Send(data2, data2.Length);

        byte[] data3 = BitConverter.GetBytes(test3);
        server.Send(data3, data3.Length);

        byte[] data4 = BitConverter.GetBytes(test4);
        server.Send(data4, data4.Length);

        byte[] data5 = Encoding.ASCII.GetBytes(test5);
        server.Send(data5, data5.Length);

        Console.WriteLine("Stopping client");
        server.Close();
    }
}
```

BinaryUdpSrvr程序的运行结果如下所示:

```
C:\>BinaryUdpSrvr
Waiting for a client...
Message received from 127.0.0.1:1252:
Hello, are you there?
test1 = 45
test2 = 3.14159
test3 = -1234567890
test4 = False
test5 = This is a test.

C:\>
```

从上面的结果中我们可以看出, 5种不同数据类型的数据都被服务器成功接收, 并且在需要的时候, 服务器能够使用所接收到的数据进行有关计算。

按照惯例, 在网络环境下进行测试的时候, WinDump和Analyzer能够监视数据包的情况。

特别要注意每一种数据类型的数据在单个数据包中发送的情况, 含有整型数据的数据包中是数值为45的整型数的字节数组, 以4个字节的二进制形式0x2D 0x00 0x00 0x00进行发送。

另外还要注意数据包中二进制数据的顺序, 在下面的内容中我们将要讲述有关方面的问题。

与其他类型主机之间的通信

在使用Intel微处理器的平台上, 运行Microsoft Windows OS的两台机器之间传输二进制数据时, 不必考虑二进制数据的表示方式, 通信双方都能够识别二进制数据, BitConverter.ToInt32()方法将BitConverter.GetBytes()方法所生成的字节数组转换为合适的二进制数据类型。

当然, 问题不仅仅这么简单, 在网络编程中, C#语言能够进行异构机器之间的通信程序的设计。因此, 很可能服务器和客户机程序所表示的二进制数据类型不一致, 在这一小节中, 我们将要讲述如何处理这类问题以及如何使C#网络程序适应异构机器之间的通信。

二进制数据类型的表示

CPU平台所带来的二进制数据类型的问题可能导致二进制数据类型的存储非常困难。因为使用了多种字节类型, 它们可能以两种方式进行存储:

- 最次要的字节优先 (称为little endian)
- 最重要的字节优先 (称为big endian)

在机器之间进行数据的发送和接收时, 必须对二进制数据类型进行解析。如果使用错误的数据类型表示形式转换原始二进制字节数组, 则程序运行就会出错。

在程序清单7.7中列出的BinaryDataTest.cs程序, 使用了BitConverter.ToString()方法来示

范系统中不同二进制数据类型的存储方式，请看下面代码。

程序清单7.7 BinaryDataTest.cs程序

```
using System;
using System.Net;
using System.Text;

class BinaryDataTest
{
    public static void Main()
    {
        int test1 = 45;
        double test2 = 3.14159;
        int test3 = -1234567890;
        bool test4 = false;
        byte[] data = new byte[1024];
        string output;

        data = BitConverter.GetBytes(test1);
        output = BitConverter.ToString(data);
        Console.WriteLine("test1 = {0}, string = {1}", test1, output);

        data = BitConverter.GetBytes(test2);
        output = BitConverter.ToString(data);
        Console.WriteLine("test2 = {0}, string = {1}", test2, output);

        data = BitConverter.GetBytes(test3);
        output = BitConverter.ToString(data);
        Console.WriteLine("test3 = {0}, string = {1}", test3, output);

        data = BitConverter.GetBytes(test4);
        output = BitConverter.ToString(data);
        Console.WriteLine("test4 = {0}, string = {1}", test4, output);
    }
}
```

这个程序实际上就是BinaryDataTest对不同的数据类型做了一些简单的BitConverter操作，并且使用BitConverter.ToString()方法显示相应结果。BinaryDataTest程序的输出如下所示：

```
C:\>BinaryDataTest
test1 = 45, string = 2D-00-00-00
test2 = 3.14159, string = 6E-86-1B-F0-F9-21-09-40
test3 = -1234567890, string = 2E-FD-69-B6
test4 = False, string = 00

C:\>
```

我们分析上面的结果（test1），可以看出，在这个机器上使用的标准字节表示是小缓冲区格式（在0前面是2D），如果是大缓冲区格式，则这个整数值将会存储为00-00-00-2D，因此，如果向使用大缓冲区的机器发送数据时，除非在程序中进行了相应调整，否则就会出现错误。

转换二进制数据的表示方法

使用不同的二进制表示方法的问题是UNIX环境下一个非常重要的问题，因为太多的平台能够运行UNIX操作系统，不能设想远程的系统使用与本地系统相同的表示方式。因此，在UNIX系统领域内提出了一种解决方案：以普通（generic）方法方式发送二进制数据。

二进制数据类型的网络字节顺序（network byte order）是网络上传输的二进制数据的一种中间存储，也就是说，每一个网络程序都可以将本地的二进制数据转换为网络字节顺序，然后再进行发送。在接收端，则要将流入的数据从网络字节顺序转换为本地的字节顺序。这样一来，就保证了二进制数据能够转换为目标主机可以识别的表示方式。图7.2举例说明了这样一个网络字节顺序的转换过程。

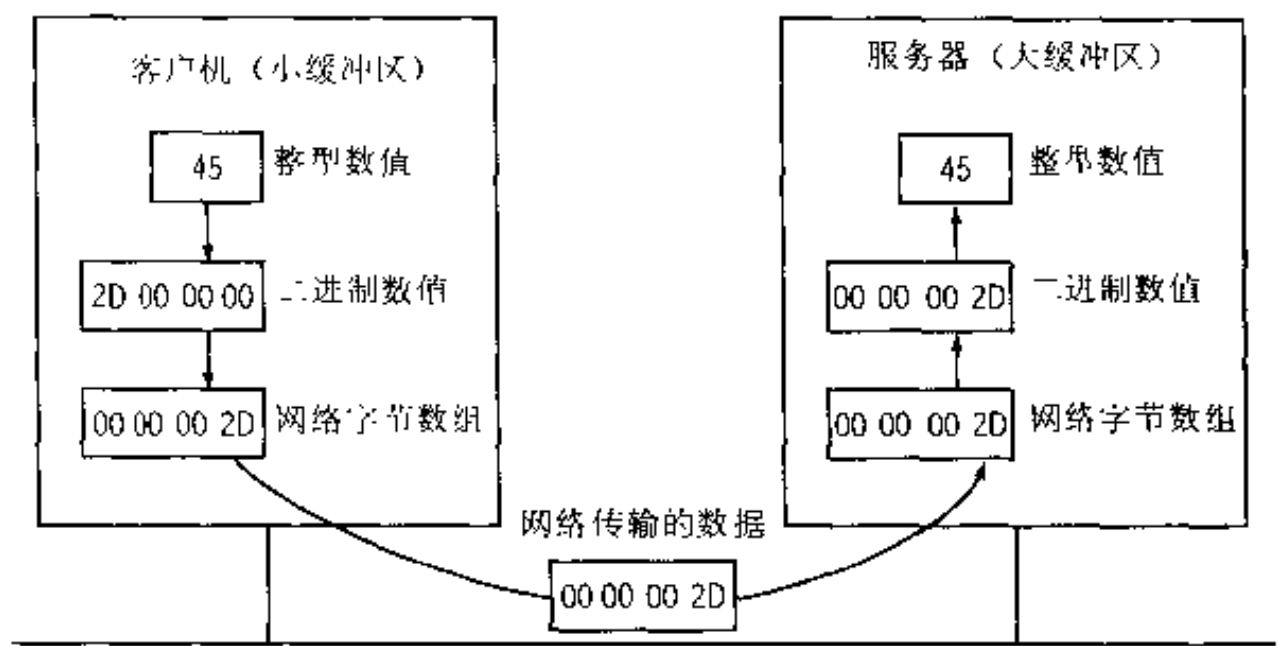


图7.2 主机之间用网络字节顺序进行信息传输

.NET库中含有将整型数据转换为网络字节顺序以及进行反向转换的方法，这些函数都位于IPAddress类中，在System.Net命名空间内。其中一个方法是HostToNetworkOrder()，这个方法将整型数据类型转换为网络字节顺序表示，在程序清单7.8中，BinaryNetworkByteOrder.cs程序示范了这个方法的使用情况，请看下面代码。

程序清单7.8 BinaryNetworkByteOrder.cs程序

```
using System;
using System.Net;
using System.Text;
class BinaryNetworkByteOrder
{
    public static void Main()
```

```

    {
        short test1 = 45;
        int test2 = 314159;
        long test3 = -123456789033452;
        byte[] data = new byte[1024];
        string output;

        data = BitConverter.GetBytes(test1);
        output = BitConverter.ToString(data);
        Console.WriteLine("test1 = {0}, string = {1}", test1, output);

        data = BitConverter.GetBytes(test2);
        output = BitConverter.ToString(data);
        Console.WriteLine("test2 = {0}, string = {1}", test2, output);

        data = BitConverter.GetBytes(test3);
        output = BitConverter.ToString(data);
        Console.WriteLine("test3 = {0}, string = {1}", test3, output);

        short test1b = IPAddress.HostToNetworkOrder(test1);
        data = BitConverter.GetBytes(test1b);
        output = BitConverter.ToString(data);
        Console.WriteLine("test1 = {0}, nbo = {1}", test1b, output);

        int test2b = IPAddress.HostToNetworkOrder(test2);
        data = BitConverter.GetBytes(test2b);
        output = BitConverter.ToString(data);
        Console.WriteLine("test2 = {0}, nbo = {1}", test2b, output);

        long test3b = IPAddress.HostToNetworkOrder(test3);
        data = BitConverter.GetBytes(test3b);
        output = BitConverter.ToString(data);
        Console.WriteLine("test3 = {0}, nbo = {1}", test3b, output);
    }
}

```

这个BinaryNetworkByteOrder程序创建了三类型整型数据，并且使用HostToNetworkOrder()方法将它们转换为网络字节顺序。在笔者进行试验的机器上，BinaryNetworkByteOrder程序的输出结果如下所示：

```

C:\>BinaryNetworkByteOrder
test1 = 45, string = 2D-00
test2 = 314159, string = 2F-CB-04-00
test3 = -123456789033452, string = 14-CE-F1-79-B7-8F-FF-FF
test1 = 11520, nbo = 00-2D
test2 = 801833984, nbo = 00-04-CB-2F
test3 = 1499401231033958399, nbo = FF-FF-8F-B7-79-F1-CE-14

C:\>

```

读者可能注意到，在上述结果中有一些奇怪的现象，`HostToNetworkOrder()`方法的返回值与原来的数值是同一数据类型。字节值已经被置为网络字节顺序，准备在网络中进行发送，不幸的是，如果网络字节顺序与本地主机采用不同的二进制表示方式，数据值将不一样。例如，变量`test1`赋值为45，当`test1`转换为网络字节顺序时，被赋为另一个变量`test1b`，现在，变量`test1b`仍然是一个有效短整型变量，但是数值却是11520，很显然这个值与原来的45不一样，当`test1b`在网络上进行传送时，必须转换为本地主机顺序以便获取原来的数值45。

警告：这里，读者必须记住，当数据被转换成网络字节顺序时，不能与原始数据相同，网络字节顺序只能用于在网络上数据的传输。

在目标主机能够使用所接收到的数据之前，必须将数据转换为本地二进制数据的表示方式。

按照网络中字节顺序读取数据

在整型数据转换为网络字节顺序并发送到远程系统后，必须被转换回主机字节顺序表示方式，以便原始数据能够在程序中使用。`IPAddress`类的`NetworkToHostOrder()`方法将所接收到的网络字节顺序的数据转换为运行该程序的主机所采用的合适字节顺序。与`HostToNetworkOrder()`类似，`NetworkToHostOrder()`方法将以网络字节顺序表示的整型数值转换为本地主机所采用的字节顺序，在实际应用中，很有可能相互通信的主机采用相同的顺序，不需要进行转换，但是为了保险起见，最好使用这两个方法进行转换。

范例程序

程序清单7.9中列出的`NetworkOrderClient.cs`程序，示范了使用`HostToNetworkOrder()`方法和`NetworkToHostOrder()`方法在网络上如何进行数据的传输，请看下面代码。

程序清单7.9 NetworkOrderClient.cs程序

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class NetworkOrderClient
{
    public static void Main()
    {
        byte[] data = new byte[1024];
        string stringData;
        TcpClient server;

        try
        {
            server = new TcpClient("127.0.0.1", 9050);
```

```
    } catch (SocketException)
    {
        Console.WriteLine("Unable to connect to server");
        return;
    }
    NetworkStream ns = server.GetStream();

    int recv = ns.Read(data, 0, data.Length);
    stringData = Encoding.ASCII.GetString(data, 0, recv);
    Console.WriteLine(stringData);

    short test1 = 45;
    int test2 = 314159;
    long test3 = -123456789033452;

    short test1b = IPAddress.HostToNetworkOrder(test1);
    data = BitConverter.GetBytes(test1b);
    Console.WriteLine("sending test1 = {0}", test1);
    ns.Write(data, 0, data.Length);
    ns.Flush();

    int test2b = IPAddress.HostToNetworkOrder(test2);
    data = BitConverter.GetBytes(test2b);
    Console.WriteLine("sending test2 = {0}", test2);
    ns.Write(data, 0, data.Length);
    ns.Flush();

    long test3b = IPAddress.HostToNetworkOrder(test3);
    data = BitConverter.GetBytes(test3b);
    Console.WriteLine("sending test3 = {0}", test3);
    ns.Write(data, 0, data.Length);
    ns.Flush();

    ns.Close();
    server.Close();
}
}
```

NetworkOrderClient程序使用**TcpClient**类创建到服务器的TCP连接，然后创建**NetworkStream**对象与远程服务器进行数据的发送和接收。一旦连接建立起来，该程序就为三种数据类型的变量分别赋值，然后按照网络字节顺序将数据发送给服务器。

在下面的程序清单7.10中所示的**NetworkOrderSrvr.cs**程序，接收网络上的数据，并将所接收到的数据转换为主机字节顺序。

请看下面代码。

程序清单7.10 NetworkOrderSrvr.cs程序

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class NetworkOrderSrvr
{
    public static void Main()
    {
        int recv;
        byte[] data = new byte[1024];

        TcpListener server = new TcpListener(9050);
        server.Start();
        Console.WriteLine("waiting for a client...");

        TcpClient client = server.AcceptTcpClient();
        NetworkStream ns = client.GetStream();

        string welcome = "Welcome to my test server";
        data = Encoding.ASCII.GetBytes(welcome);
        ns.Write(data, 0, data.Length);
        ns.Flush();

        data = new byte[2];
        recv = ns.Read(data, 0, data.Length);
        short test1t = BitConverter.ToInt16(data, 0);
        short test1 = IPAddress.NetworkToHostOrder(test1t);
        Console.WriteLine("received test1 = {0}", test1);

        data = new byte[4];
        recv = ns.Read(data, 0, data.Length);
        int test2t = BitConverter.ToInt32(data, 0);
        int test2 = IPAddress.NetworkToHostOrder(test2t);
        Console.WriteLine("received test2 = {0}", test2);

        data = new byte[8];
        recv = ns.Read(data, 0, data.Length);
        long test3t = BitConverter.ToInt64(data, 0);
        long test3 = IPAddress.NetworkToHostOrder(test3t);
        Console.WriteLine("received test3 = {0}", test3);

        ns.Close();
        client.Close();
        server.Stop();
    }
}
```


NetworkOrderSrvr程序使用**TcpListener**类在TCP的端口19050处收听流入的连接尝试。当接收到一个连接尝试时，这个程序就创建一个**TcpClient**对象，然后使用**GetStream()**方法创建一个**NetworkStream**对象从远程主机发送和接收数据。

网络连接建立并发送欢迎消息后，**NetworkOrderSrvr**程序希望从远程主机中接收三个整型数据类型的数值。记住TCP是面向数据流的通信，不能保证这三种类型的数据以三种不同的消息形式发送，为了解决这个问题，**NetworkOrderSrvr**为每一个数据从**NetworkStream**中读取一个字节数，这就保证了无论数据以什么方式被接收，都能够从TCP缓冲区中读取正确的消息。

一旦从TCP缓冲区中读取了数据，就用**BitConverter**方法将数据转换为合适的二进制数据类型，请看下面代码：

```
data = new byte[2];
recv = ns.Read(data, 0, data.Length);
short test1t = BitConverter.ToInt16(data, 0);
```

在这里，读者应该注意，由于不能确定网络字节顺序对于运行程序的主机来说是否正确，因此转换后的数值不能直接使用，仅仅是一个临时数值，为了生成正确的数值，必须将这个临时数值转换为主机所采用的字节顺序表示的数值，请看下面代码：

```
short test1 = IPAddress.NetworkToHostOrder(test1t);
```

这就保证了数据是以运行程序的主机所采用的二进制数据表示方式来表示的。

NetworkOrderClient和**NetworkOrderSrvr**的输出结果非常类似，客户机程序的输出结果如下所示：

```
C:\>NetworkOrderClient
Welcome to my test server
sending test1 = 45
sending test2 = 314159
sending test3 = -123456789033452

C:\>
```

NetworkOrderSrvr程序的输出结果如下所示：

```
C:\>NetworkOrderSrvr
waiting for a client...
received test1 = 45
received test2 = 314159
received test3 = -123456789033452

C:\>
```

从服务器程序的输出中看出，在服务器端所接收到的数据与客户机端所发送的一样，表明这三种类型的整型数据都被正确地转换、发送、接收并反向转换。

一般来说，如果在网络上运行这个程序，可以使用**WinDump**或者**Analyzer**程序监视单个数据包的传输情况，通过将**BinaryNetworkOrder**程序中以网络字节顺序表示的数据包的数据

字节与Analyzer跟踪的数据包相比较，可以看出数据包中的数据值。

在笔者进行的跟踪过程中，TCP数据字节的内容显示了test2 (00 04 CB 2F) 和test3 (FF 8F B7 79 F1 CE 14) 以网络字节顺序在同一个TCP数据包中发送（将这些值与程序清单7.8中的程序BinaryNetworkByteOrder的输出结果相比较，可以看出不同），当目标主机收到这些数值的时候，就将它们转换为主机字节顺序，以获取原始数据值。

复杂对象的传输

现在我们已经能够向远程主机发送单个二进制数据，读者可能会对下一步操作感到疑惑：如何通过网络向远程主机发送成组数据？接下来就介绍这方面的内容，即如何将一组数据作为单个数据进行发送，并将接收到的数据进行相应转换获取正确数值。

创建数据类集

在网络上不同的系统之间传输数据的最常用方式是创建一个类，在这类中将所有的数据包包含进去，并创建一个特殊的函数将数据转换为字节数组形式，在类中含有通信时数据元素能够用到的基本变量，请看下面代码：

```
class Employee
{
    public int EmployeeID;
    public string LastName;
    public string FirstName;
    public int YearsService;
    public double Salary;

    public int LastNameSize;
    public int FirstNameSize;
    public int size;
}
```

在这里，类Employee可以看做与记录相似，变量与记录中的字段相似，类的每一个实例表示数据库中的一个记录。因为这两个字符串元素长度可变，因此，需要额外的元素定义大小，与第5章中讲述过的可变文本字段方法相对比。

最后，创建一个数据元素，存放这个类实例全部字节表示的尺寸信息，这一点也是必需的，因为这个类实例本身长度可变。

GetBytes()方法

在数据集中，我们创建一个GetBytes()方法用于将所有的元素转换为一个单字节数组，以便发送到网络上，请看下面代码：

```
public byte[] GetBytes()
{
    byte[] data = new byte[1024];
    int place = 0;
```



```

Buffer.BlockCopy(BitConverter.GetBytes(EmployeeID), 0, data, place, 4);
place += 4;
Buffer.BlockCopy(BitConverter.GetBytes(LastName.Length), 0, data, place, 4);
place += 4;
Buffer.BlockCopy(Encoding.ASCII.GetBytes(LastName), 0,
    data, place, LastName.Length);
place += LastName.Length;
Buffer.BlockCopy(BitConverter.GetBytes(FirstName.Length),
    0, data, place, 4);
place += 4;
Buffer.BlockCopy(Encoding.ASCII.GetBytes(FirstName), 0,
    data, place, FirstName.Length);
place += FirstName.Length;
Buffer.BlockCopy(BitConverter.GetBytes(YearsService), 0, data, place, 4);
place += 4;
Buffer.BlockCopy(BitConverter.GetBytes(Salary), 0, data, place, 8);
place += 8;
size = place;
return data;
}

```

GetBytes()方法有三种功能:

- 将类中的每一个元素转换为字节数组
- 将所有独立的字节数组设置为单字节数组
- 计算字节数组的总尺寸

到目前为止,读者已经非常熟悉使用**BitConverter**方法转换不同类型的二进制数据为字节数组形式,读者不熟悉的就是**Buffer**类的**BlockCopy()**方法。这个**BlockCopy()**方法用于将整个字节数组复制到另一个字节数组的某一个位置,请看下面代码:

```
BlockCopy(byte[] array1, int start, byte[] array2, int offset, int size)
```

参数array1是要复制到array2的数组, array1中进行复制的起始位置从第一个字节开始,每一次添加数组项之后,第二个数组内的偏移位置就会发生变化,短整数据偏移2个字节,双精度浮点数偏移8个字节。

在这里我们并不知道这两个可变长度的字符串数值,因此,就需要从类中获取字符串尺寸信息,因为我们知道字符串实例的长度,在将字符串实例放置到字节数组中时可以使用这个数值,如图7.3所示。

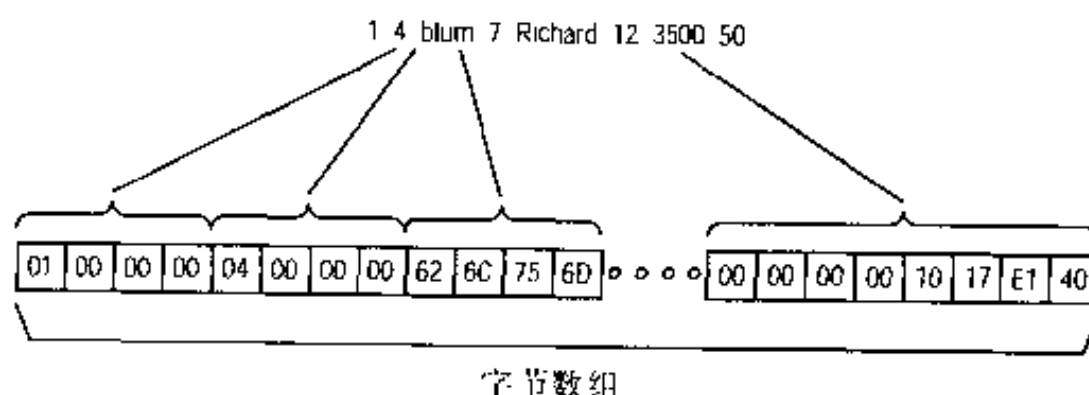


图7.3 在字节数组内设置数据值

在这里基本的思路就是将每一个字节数组放置到数据数组中合适的位置。因此必须准确计算每一个位置，因为一旦完成这个操作，字节数组就准备在网络上发送。

构造函数

这个数据类有两种构造函数，一种是默认的构造函数，用于手动输入数据元素；另一种是读取**GetBytes()**方法所生成的字节数组，并将其转换为一个类实例，请看下面代码：

```
public Employee()
{
}

public Employee(byte[] data)
{
    int place = 0;
    EmployeeID = BitConverter.ToInt32(data, place);
    place += 4;
    LastNameSize = BitConverter.ToInt32(data, place);
    place += 4;
    LastName = Encoding.ASCII.GetString(data, place, LastNameSize);
    place = place + LastNameSize;
    FirstNameSize = BitConverter.ToInt32(data, place);
    place += 4;
    FirstName = Encoding.ASCII.GetString(data, place, FirstNameSize);
    place += FirstNameSize;
    YearsService = BitConverter.ToInt32(data, place);
    place += 4;
    Salary = BitConverter.ToDouble(data, place);
}
```

使用默认的构造函数能够为类实例指定数据元素，在实例中存放数值。第二种构造函数分析整个字节数组，并获取每一数据元素数值。变量长度字符串字段需要尺寸信息以确定每一个字符串分配的字节数，因为，这个尺寸由**GetBytes()**方法嵌入在字节数组中，因此，为字符串读取每一个信息，并取出适当的字节数非常重要。

完整的类程序

将上述所有的元素和方法合并在一起就构成了**Employee**类文件，请看下面的程序清单7.11。

程序清单7.11 Employee.cs程序

```
using System;
using System.Text;

class Employee
{
```

```
public int EmployeeID;
private int LastNameSize;
public string LastName;
private int FirstNameSize;
public string FirstName;
public int YearsService;
public double Salary;
public int size;

public Employee()
{
}

public Employee(byte[] data)
{
    int place = 0;
    EmployeeID = BitConverter.ToInt32(data, place);
    place += 4;
    LastNameSize = BitConverter.ToInt32(data, place);
    place += 4;
    LastName = Encoding.ASCII.GetString(data, place, LastNameSize);
    place = place + LastNameSize;
    FirstNameSize = BitConverter.ToInt32(data, place);
    place += 4;
    FirstName = Encoding.ASCII.GetString(data, place, FirstNameSize);
    place += FirstNameSize;
    YearsService = BitConverter.ToInt32(data, place);
    place += 4;
    Salary = BitConverter.ToDouble(data, place);
}

public byte[] GetBytes()
{
    byte[] data = new byte[1024];
    int place = 0;
    Buffer.BlockCopy(BitConverter.GetBytes(EmployeeID), 0, data, place, 4);
    place += 4;
    Buffer.BlockCopy(BitConverter.GetBytes(
        LastName.Length), 0, data, place, 4);
    place += 4;
    Buffer.BlockCopy(Encoding.ASCII.GetBytes(
        LastName), 0, data, place, LastName.Length);
    place += LastName.Length;
    Buffer.BlockCopy(BitConverter.GetBytes(
        FirstName.Length), 0, data, place, 4);
```

```

        place += 4;
        Buffer.BlockCopy(Encoding.ASCII.GetBytes(
            FirstName), 0, data, place, FirstName.Length);
        place += FirstName.Length;
        Buffer.BlockCopy(BitConverter.GetBytes(YearsService), 0, data, place, 4);
        place += 4;
        Buffer.BlockCopy(BitConverter.GetBytes(Salary), 0, data, place, 8);
        place += 8;
        size = place;
        return data;
    }
}

```

由于Employee.cs本身只是一个数据的容器，不能单独运行，因此，在这个程序中没有Main()方法，不能使用csc命令对这个Employee.cs程序进行编译，这个程序必须与使用Employee类的程序一起编译。

Data类的使用

Employee.cs程序设计好后，在客户机和服务器程序中使用这个程序非常快捷。在程序清单7.12中列出的是一个TCP客户机程序的样本，在这个TCP程序中使用了Employee类，将雇员信息发送给服务器。

程序清单7.12 EmployeeClient.cs程序

```

using System;
using System.Net;
using System.Net.Sockets;

class EmployeeClient
{
    public static void Main()
    {
        Employee emp1 = new Employee();
        Employee emp2 = new Employee();
        TcpClient client;

        emp1.EmployeeID = 1;
        emp1.LastName = "Blum";
        emp1.FirstName = "Katie Jane";
        emp1.YearsService = 12;
        emp1.Salary = 35000.50;

        emp2.EmployeeID = 2;
        emp2.LastName = "Blum";
        emp2.FirstName = "Jessica";
    }
}

```

```
emp2.YearsService = 9;
emp2.Salary = 23700.30;

try
{
    client = new TcpClient("127.0.0.1", 9050);
} catch (SocketException)
{
    Console.WriteLine("Unable to connect to server");
    return;
}

NetworkStream ns = client.GetStream();

byte[] data = empl.GetBytes();
int size = empl.size;
byte[] packsize = new byte[2];
Console.WriteLine("packet size = {0}", size);
packsize = BitConverter.GetBytes(size);
ns.Write(packsize, 0, 2);
ns.Write(data, 0, size);
ns.Flush();

data = emp2.GetBytes();
size = emp2.size;
packsize = new byte[2];
Console.WriteLine("packet size = {0}", size);
packsize = BitConverter.GetBytes(size);
ns.Write(packsize, 0, 2);
ns.Write(data, 0, size);
ns.Flush();

ns.Close();
client.Close();
}
}
```

创建了Employee类的两个实例后，将数据输入到了数据元素中。GetByte()方法将数据转换为字节数据并发送到服务器中。在字节数组发送之前，先发送数组的大小，以便服务器读取一个完整的数据包的字节数。

相类似，程序清单7.13中给出的EmployeeSrvr.cs程序，使用Employee类实现服务器功能。请看下面代码。

程序清单7.13 EmployeeSrvr.cs程序

```
using System;
using System.Net;
using System.Net.Sockets;
```

```
class EmployeeSrvr
{
    public static void Main()
    {
        byte[] data = new byte[1024];
        TcpListener server = new TcpListener(9050);

        server.Start();
        TcpClient client = server.AcceptTcpClient();
        NetworkStream ns = client.GetStream();

        byte[] size = new byte[2];
        int recv = ns.Read(size, 0, 2);
        int packsize = BitConverter.ToInt16(size, 0);
        Console.WriteLine("packet size = {0}", packsize);
        recv = ns.Read(data, 0, packsize);

        Employee empl = new Employee(data);
        Console.WriteLine("empl.EmployeeID = {0}", empl.EmployeeID);
        Console.WriteLine("empl.LastName = {0}", empl.LastName);
        Console.WriteLine("empl.FirstName = {0}", empl.FirstName);
        Console.WriteLine("empl.YearsService = {0}", empl.YearsService);
        Console.WriteLine("empl.Salary = {0}\n", empl.Salary);

        size = new byte[2];
        recv = ns.Read(size, 0, 2);
        packsize = BitConverter.ToInt16(size, 0);
        data = new byte[packsize];
        Console.WriteLine("packet size = {0}", packsize);
        recv = ns.Read(data, 0, packsize);

        Employee emp2 = new Employee(data);
        Console.WriteLine("emp2.EmployeeID = {0}", emp2.EmployeeID);
        Console.WriteLine("emp2.LastName = {0}", emp2.LastName);
        Console.WriteLine("emp2.FirstName = {0}", emp2.FirstName);
        Console.WriteLine("emp2.YearsService = {0}", emp2.YearsService);
        Console.WriteLine("emp2.Salary = {0}", emp2.Salary);

        ns.Close();
        client.Close();
        server.Stop();
    }
}
```

这个EmployeeSrvr程序从网络上读取了2个字节的数据，然后将它们转换为一个整型数值，数值的大小就表示从数据包中读取的字节。一旦数据包被读取，就可以使用Employee构造方法将它转换为一个Employee实例。

要对EmployeeClient.cs和EmployeeSrvr.cs进行编译，必须打开Employee.cs文件：

```
csc EmployeeClient.cs Employee.cs
csc EmployeeSrvr.cs Employee.cs
```

在对上述两个程序进行编译后，启动EmployeeSrvr程序（在命令行窗口中启动）和EmployeeClient程序（在命令行窗口中启动或者在网络上启动）进行测试。EmployeeSrvr程序的运行结果如下所示：

```
C:\>EmployeeSrvr
packet size = 30
emp1.EmployeeID = 1
emp1.LastName = Blum
emp1.FirstName = Katie Jane
emp1.YearsService = 12
emp1.Salary = 35000.5

packet size = 27
emp2.EmployeeID = 2
emp2.LastName = Blum
emp2.FirstName = Jessica
emp2.YearsService = 9
emp2.Salary = 23700.3

C:\>
```

从上述结果我们能够看出，客户机程序为每一个实例成功地向服务器传输了雇员数据。

说明：读者可能已经注意到，在Employee程序中没有使用网络字节顺序进行数据的传输，因此这些程序只能在网络中同种类型的机器之间使用。要在网络上任意平台之间使用，读者可以尝试将字节数值转换为网络字节顺序。

说明：注意，我们这里讨论的是一个串行器，它的功能是将一个类串行化为一个二进制数据流。NET有自己的串行器，有关内容我们将在本书第16章“远程技术”中讲述。

小结

在这一章里，我们重点对网络编程中如何使用.NET助手类进行了讨论。使用Socket类我们可以采用传统UNIX环境下的网络编程方法，在本章中我们讲述了三个新类：TcpClient、TcpListener和UdpClient，使用这三个类能够简化网络程序设计，减少程序代码量，其中在TCP网络程序设计中使用TcpClient和TcpListener类，而UDPCClient类用于UDP程序。

在网络通信中，必须保证所发送的数据，接收端能够识别。文本数据通常不成问题，但是在发送二进制数据时，情况则非常复杂。C#语言中提供了二进制的数据类型，在发送之前，这些类型的数据都必须转换为字节数组，转换工作可以由BitConverter来完成。只要二进制数据转换为字节数组的形式，就可以使用相应的技术在网络上进行传输。

当然，不是所有的系统都采用相同的二进制表示方式，因此多字节数据中字节的存放顺序就非常重要。缓冲区大的系统一般不会立即翻译小缓冲区系统发送来的数据，因此，要进行二进制数据的传输就要在传输之前将数据转换为一定的网络字节顺序。在远程系统中，所接收到的数据必须能够按照相应的网络字节顺序转换为本地字节表示顺序。

在网络上进行复杂数据的传输也非常困难，最简单的途径就是将每个数据元素转换为一个字节数组，并将转换后的字节数组合并为一个大数组进行传输；在接收端，则将接收到的大数组重新拆分为原来的数据形式。

第8章 异步套接字编程

- Windows事件编程
- 使用Events和Delegates
- 使用异步套接字
- 使用非阻塞的套接字
- Poll()方法
- Select()方法

我们已经了解了如何在阻塞方式下使用套接字创建网络程序。在模块化方式下，套接字将保持等待状态，直到完成它们的功能，并且维持在应用程序中的其他功能直至结束。在这种方式下，很多程序可以很快地运行，但是对于在Windows程序环境下的应用程序来说，这可能是个问题。

在第3章“C#网络编程的类”中，我们介绍了异步编程的概念。非阻塞的程序使用异步的Socket方法执行，允许网络程序继续执行，而不是等待网络操作。另外，用户可以使用在第4章“DNS和C#”中的异步DNS类的方法进行异步编程的测试。

在本章中，我们将详细的讨论异步套接字。首先，我们要对Windows编程环境进行一个简短的描述。用户将了解Windows事件改变了网络程序运行的方法。我们将讨论两种方法，来避免在网络应用程序中使用阻塞的socket:

- 使用.NET异步套接字
- 使用传统的非阻塞的socket方法

在Windows网络应用程序中，这些模型都有其正面和反面的考虑。在下面的章节中，我们将讨论在这些模型中使用的方法和处理，另外还有一些可以在C#网络程序中使用的技巧。

Windows事件编程

在本书中，几乎所有的程序例子都使用了.NET中的控制台模式。Windows控制台模式编程使用传统的结构化编程模型。在结构化编程中，程序流的控制由程序本身完成。在程序执行中，类中的每种方法由函数依次调用。处理程序允许的部分，用户无法对程序的执行进行操作。与之相比，Windows编程使用事件编程模型。

Windows事件编程基于事件的程序流。当程序中的事件发生时，将调用特殊的方法并基于事件执行，如图8.1所示。然而，在阻塞网络功能下，这种方法将无法正常工作。当应用程序为用户呈现图形化界面时，应用程序将处于等待用户状态，确定什么功能将被执行。对于事件编程来说，当其他功能在执行时（例如网络访问），客户同样可以控制图形界面。这样就允许在等待网络响应时，用户执行其他的功能，甚至放弃网络连接。但是，如果使用了

阻塞的网络功能，程序将处于等待状态，直到功能被执行完毕，并且用户无法对界面或者程序进行控制。

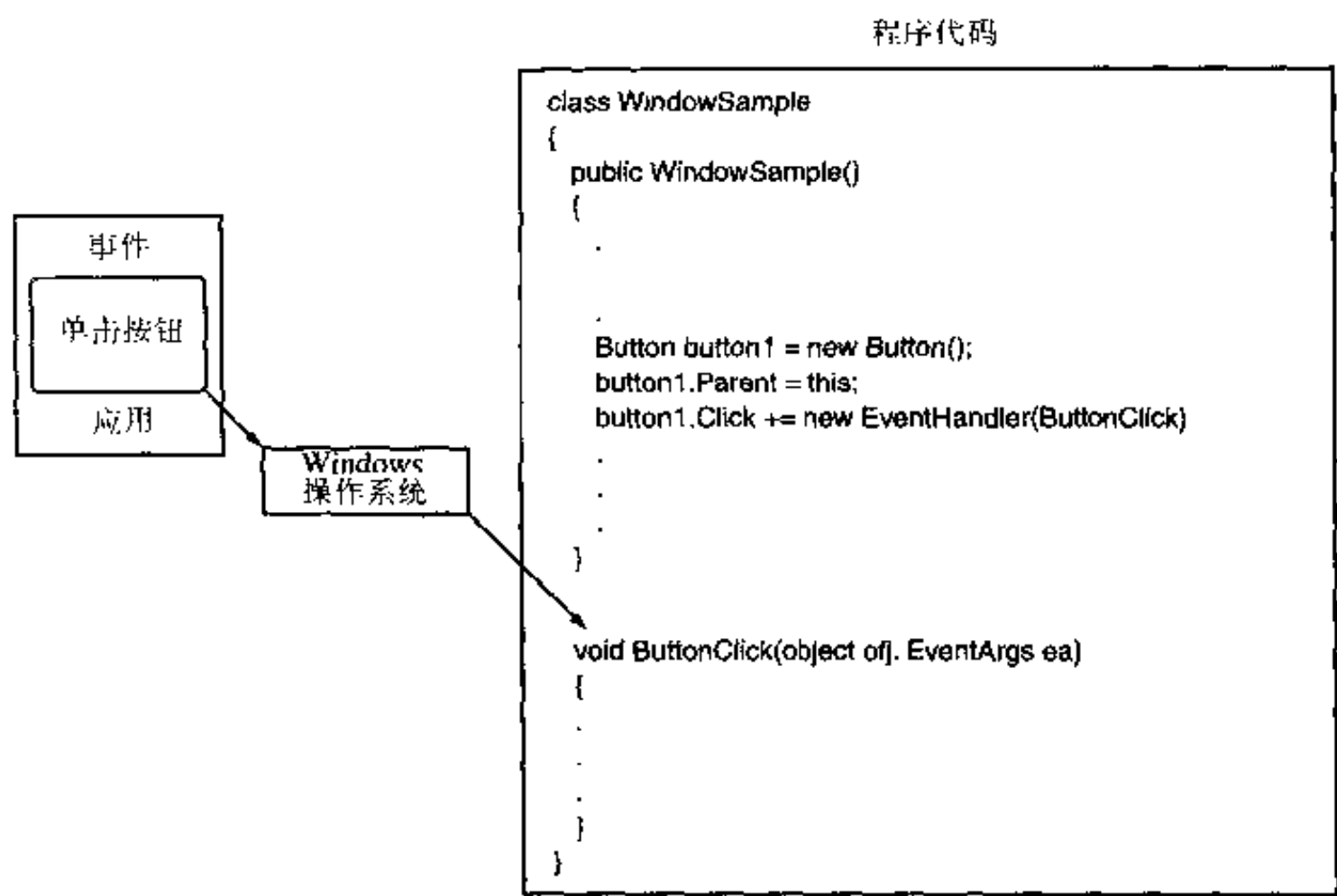


图8.1 Windows事件编程模式

本节内容将描述Windows事件编程如何使用C#，并且如何在Windows事件编程模式下使用.NET异步网络方法。

使用Events和Delegates

.NET程序开发环境非常类似于Windows环境，所以.NET能够完全支持事件编程方式。在.NET事件编程中，两个非常重要的结构分别是events和delegates。

event是由一个表现已经发生行为的对象发送的一条消息。该消息用于确认该行为并且提供该行为所需的任何有用数据。事件可以为任何行为，当用户单击按钮（在这里信息表示按钮名称），套接字将收到一个数据包（在这里信息表示套接字收到了数据）。事件的发送者无需知道事件消息传送到Windows系统后如何处理。事件接收器收到消息后在Windows系统上进行注册，并且提示哪种类型的事件接收器想要接收。图8.2显示这种功能。

在Windows系统中，事件接收器通过名为delegate的指针类进行确认。该代表为一个类，包含对接收到的事件处理的相关方法。当Windows接收到一个事件，系统要检测是否由代表注册来处理该事件。如果已经是代表注册来处理该事件，事件消息将传送到该代表定义的方法。在方法完成处理后，Windows系统处理下一个发生的事件，直到收到程序结束事件信号。

事件程序示例

成功的事件编程是Windows程序。在Windows图形程序中产生的每一个对象都可以产生

一个或者更多的，基于客户的操作的对象。.NET框架System.Windows.Forms命名空间包含很多类，在Windows环境中，Windows对象要创建全部功能的图形程序必需这些类。这些类如下：

- Buttons（按钮）
- Text boxes（文本框）
- List boxes（列表框）
- Combo boxes（组合框）
- Check boxes（复选框）
- Text labels（文本标记）
- Scroll bars（滚动条）
- Window menus（Windows菜单）

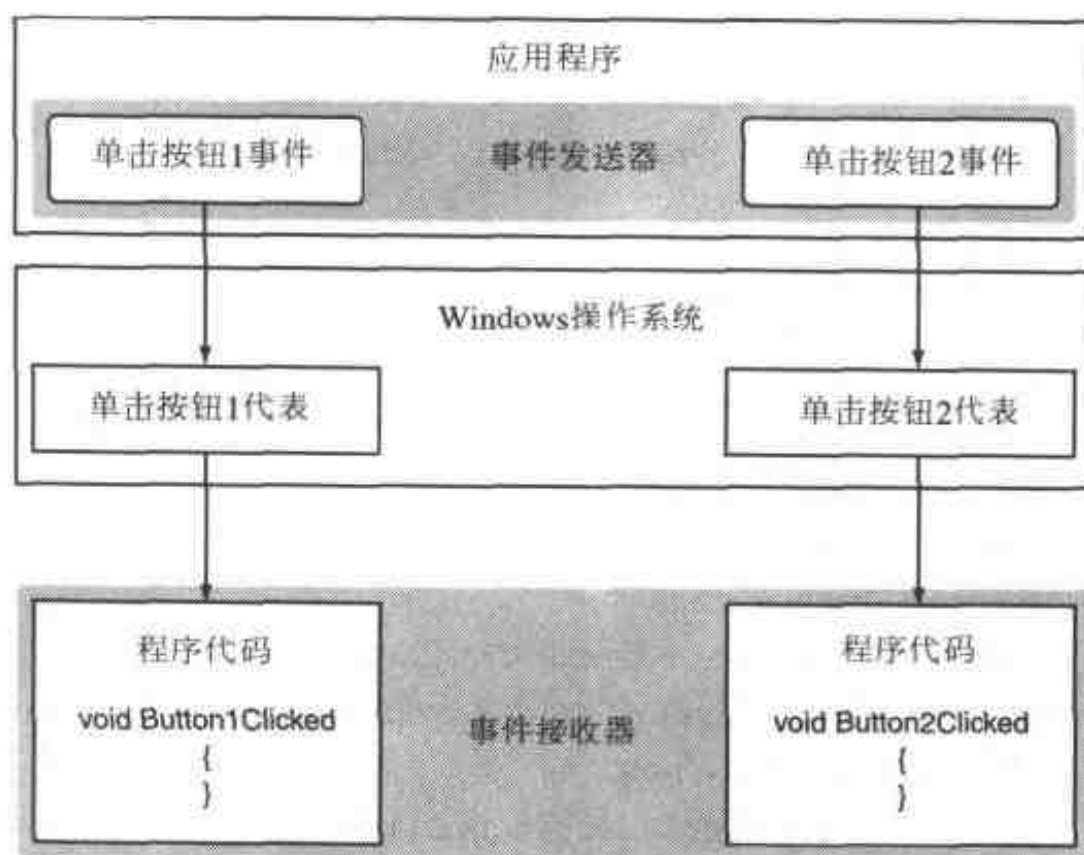


图8.2 Windows事件发送器和接收器

使用Windows对象可以非常容易地创建一个专业的网络程序。当每个对象被加载到Windows表单时，用户必须将用于事件处理的方法进行注册。当事件产生时，Windows将程序的控制传递到事件处理方法。程序清单8.1示范了一个简单的Windows表单程序，该程序使用了一些简单的Windows对象。

程序清单8.1 WindowSample.cs程序

```
using System;
using System.Drawing;
using System.Windows.Forms;

class WindowSample Form:
{
```

```
private TextBox data;
private ListBox results;

public WindowSample()
{
    Text = "Sample Window Program";
    Size = new Size(400, 380);

    Label label1 = new Label();
    label1.Parent = this;
    label1.Text = "Enter text string:";
    label1.AutoSize = true;
    label1.Location = new Point(10, 10);

    data = new TextBox();
    data.Parent = this;
    data.Size = new Size(200, 2 * Font.Height);
    data.Location = new Point(10, 35);

    results = new ListBox();
    results.Parent = this;
    results.Location = new Point(10, 65);
    results.Size = new Size(350, 20 * Font.Height);

    Button checkit = new Button();
    checkit.Parent = this;
    checkit.Text = "test";
    checkit.Location = new Point(235, 32);
    checkit.Size = new Size(7 * Font.Height, 2 * Font.Height);
    checkit.Click += new
        EventHandler(checkit_OnClick);
}

void checkit_OnClick(object obj, EventArgs ea)
{
    results.Items.Add(data.Text);
    data.Clear();
}

public static void Main()
{
    Application.Run(new WindowSample());
}
```

从表单的角度来看，这个程序非常简单，可以帮助用户理解事件编程。首先要记住，所有的Windows表单程序必须使用System.Windows.Forms命名空间和System.Drawing命名空间，用于在窗口中对对象进行定位。

```
using System.Drawing;  
using System.Windows.Forms;
```

因为应用程序创建了一个窗口，该窗口必须继承标准的Form类：

```
class WindowSample : Form
```

用于类的构造器必须定义在表单中使用的所有图形对象。首先，要定义Windows标题使用的标准值和默认尺寸：

```
Text = "Sample Window Program";  
Size = new Size(400, 380);
```

其次，要定义在窗口中出现的所有对象和窗口自身。这个例子创建了如下对象：

- Label对象用于显示说明性文本字符串
- TextBox对象允许用户输入数据
- ListBox可以方便地为用户显示输出
- 当行为发生时，Button对象允许用户进行控制

在这个程序中，行为的关键在于EventHandler注册到Button对象，在Button对象checkit中，每个注册的方法ButtonOnClick()都带有一个单击事件。

```
checkit.Click += new EventHandler(ButtonOnClick);
```

当用户单击按钮，程序控制移动到ButtonOnClick()方法：

```
void ButtonOnClick(object obj, EventArgs ea)  
{  
    results.Items.Add(data.Text);  
    data.Clear();  
}
```

这个简单的方法只执行两个功能。首先，该方法提取在TextBox对象中输入的文本字符串并且将内容写入ListBox对象。接下来，该方法清除TextBox中的文本。当用户每次单击Button对象时，将在ListBox中填入一行新的文本字符串。在本章后面章节“使用异步套接字的例子程序”中的网络编程例子中，将利用这个简单的Windows表单编程对象。

说明：当编译一个Windows表单程序，用户需要使用csc命令编译器的/t:winexe选项，如下：csc /t:winexe WindowSample.cs。这个选项将创建一个Windows可执行程序，在此运行中，无需打开一个命令提示符窗口。

AsyncCallback类

正如事件可以引发代表，.NET同样提供一种方法来引发代表。.NET AsyncCallback类允许方法启动异步的功能并且在异步功能结束后提供一个代表方法调用。

在事件不是由Windows对象产生的程序中，这种处理不同于标准的事件编程，但是这种方法胜于其他方法。当方法结束自身功能时，方法本身注册一个AsyncCallback代表进行调

用。这种情况下，方法将向Windows操作系统提供完成信息，从而触发一个事件，并将程序的控制传递到AsyncCallback代表注册定义的方法。

Socket类通过利用在AsyncCallback中定义的方法允许网络功能在后台处理中进行异步的操作。该操作将提示操作系统网络功能已经结束并且传递程序的控制到AsyncCallback方法来结束网络功能。在Windows编程环境中，这些方法通常用于防止在等待网络功能结束过程中应用程序产生锁死状态。

使用异步套接字

Socket对象包含一些方法，当网络功能结束时，这些方法利用AsyncCallback类调用结束方法。这样就可以在等待网络操作完成自身工作时，允许应用程序继续处理其他的事件。

Socket异步方法将公共网络编程功能分为两个部分：

- Begin方法，启动网络功能并且注册AsyncCallback方法。
- End方法，当AsyncCallback方法被调用时，用于结束功能。

表8.1显示了一个异步方法，可以和Socket对象一起使用。每个Begin方法有一个匹配的End方法来结束功能。

表8.1 网络异步Socket方法

请求开始的方法	请求说明	请求结束的方法
BeginAccept()	接收一个接入的连接	EndAccept()
BeginConnect()	连接到一个远程主机	EndConnect()
BeginReceive()	从套接字检索数据	EndReceive()
BeginReceiveFrom()	从指定的远程主机检索数据	EndReceiveFrom()
BeginSend()	从套接字发送数据	EndSend()
BeginSendTo()	把数据发送到远程主机	EndSendTo()

说明：注意，这些方法只能应用于Socket对象。在.NET Framework版本1中，TcpClient、TcpListener和UdpClient类不包含异步方法。

在本节中，我们将讨论使用每种异步Socket方法来处理网络功能。然后我们将观察一些示范程序。

建立连接

用于和远程主机建立连接的方法主要依靠程序是否扮演服务器（等待客户机连接）或者客户机（试图与远程服务器连接）。对于服务器来说，需要使用BeginAccept()方法；对于客户机来说，需要使用BeginConnect()方法。

BeginAccept()和EndAccept()方法

要接受一个从远程客户机的外部连接，用户必须使用BeginAccept()方法。该方法格式如下：

```
IAsyncResult BeginAccept(AsyncCallback callback, object state)
```

BeginAccept()方法使用两个参数：AsyncCallback方法的名称用于结束功能，普通的状态对象可以在异步方法间传递信息。

以下是BeginAccept()方法典型的用法：

```
Socket sock = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
    ProtocolType.Tcp);
IPEndPoint iep = new IPEndPoint(IPAddress.Any, 9050);
sock.Bind(iep);
sock.Listen(5);
sock.BeginAccept(new AsyncCallback(CallAccept), sock);
```

本段代码创建了一个Socket对象，并且分配一个本地IP地址和TCP端口号用于侦听外部的连接请求。当一个连接试图检测套接字时，将使用BeginAccept()方法定义的代表方法。最后的传递到BeginAccept()方法的参数是Socket对象产生的。

当BeginAccept()方法结束后，当连接试图发生时，将调用定义的AsyncCallback方法。AsyncCallback方法必须包括EndAccept()方法来结束套接字接收。这里是EndAccept()方法的格式：

```
Socket EndAccept(IAsyncResult iar);
```

IAsyncResult对象参数将IAsyncResult值从关联的BeginAccept()方法传递到EndAccept()方法——这就是BeginAccept()和EndAccept()匹配的方法。

类似于同步的Accept()方法，EndAccept()方法返回的Socket对象用于与客户机建立新的连接。所有未来和远程客户机的通信都要使用Socket对象。EndAccept() AsyncCallback方法的一段示范代码如下：

```
private static void CallAccept(IAsyncResult iar)
{
    Socket server = (Socket)iar.AsyncState;
    Socket client = server.EndAccept(iar);
    .
    .
    .
}
```

AsyncCallback方法的名称必须和BeginAccept()方法参数中的名称相匹配。方法的第一步就是得到原始的服务器套接字。这必须使用IAsyncResult类的AsyncState属性。这个属性传递放置在BeginAccept() object参数中的原始对象。因为该对象是作为普通对象定义的，因此必须融合到Socket对象中。

在得到原始的Socket对象后，EndAccept()方法可以从客户机连接中获得一个新的Socket对象。IAsyncResult对象参数必须和传递到AsyncCallback方法的相同。

一旦创建客户机Socket对象，可以像其他Socket对象一样使用，使用同步或者异步的方法对套接字进行读取和写入数据。

最后，BeginAccept()/EndAccept()异步方法可以产生和Accept()同步方法同样的结果。用户会发现，所有异步的Begin()和End()方法都可以适用。这就允许在网络程序中，采用这些异步的方法直接替代同步的方法。

BeginConnect()和EndConnect()方法

对于客户机应用程序采用异步方法连接到远程服务器，用户必须使用BeginConnect()方法。格式如下：

```
IAsyncResult BeginConnect(EndPoint ep, AsyncCallback callback, Object state)
```

第一个参数是远程主机连接的EndPoint值，传递到BeginConnect()方法。和BeginAccept()方法一样，BeginConnect()方法指定AsyncCallback方法的代表方法名称，当连接准备结束时进行调用。最后一个参数是状态对象，传递到EndConnect()方法，用于传输必要的信息。

这里是BeginConnect()方法的示例代码段：

```
Socket newsock = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
    ProtocolType.Tcp);
IPEndPoint iep = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 9050);
newsock.BeginConnect(iep, new AsyncCallback(Connected), newsock);
```

这个代码段创建了一个Socket对象newsock和一个IPEndPoint对象iep，用于远程的主机。BeginConnect()方法参考AsyncCallback方法（Connected），并且传递原始的Socket对象newsock到AsyncCallback方法。

当连接结束，AsyncCallback方法被调用。AsyncCallback使用EndConnect()方法来结束连接。

EndConnect()方法的格式如下：

```
EndConnect(IAsyncResult iar)
```

再一次，IAsyncResult对象指定从BeginConnect()方法传递对象值。示例表明，BeginConnect()方法将返回IAsyncResult对象。一个使用该方法的例子如下：

```
public static void Connected(IAsyncResult iar)
{
    Socket sock = (Socket)iar.AsyncState;
    try
    {
        sock.EndConnect(iar);
    } catch (SocketException)
    {
    }
}
```



```
        Console.WriteLine("Unable to connect to host");
    }
}
```

在AsyncCallback方法中的第一条语句得到用于BeginConnect()方法调用的原始套接字，使用IAsyncResult对象的AsyncState特性传递到AsyncCallback方法。

当重新创造原始套接字后，调用EndConnect()方法，使用IAsyncResult对象指向原始的BeginConnect()方法。因为远程主机可能无法使用，所以最好将EndConnect()方法放入try-catch块中。如果EndConnect()方法不成功，将产生一个SocketException错误。

发送和接收数据

当建立连接后，用户可能非常希望与远程主机进行发送和接收数据。异步方法同样可以完成此项工作。

BeginSend()和EndSend()方法

BeginSend()方法发送数据到连接的套接字。格式如下：

```
IAsyncResult BeginSend(byte[] buffer, int offset, int size,
    SocketFlags sockflag, AsyncCallback callback, object state)
```

如用户所见，BeginSend()方法使用了大量的参数。用户会注意到这些参数非常类似同步的Send()方法。在这些参数中，第一个参数是字节数组，包括用户将发送的数据。offset参数在buffer参数中指示出发送数据的起始位置，size参数指明buffer参数中多少字节将被发送。接下来，sockflag参数指明用户将用于通信的特殊的套接字标记。最后，当BeginSend()方法成功，并且状态对象发送信息到EndSend()方法时，调用AsyncCallback callback和state对象。

BeginSend()方法的示例如下：

```
sock.BeginSend(data, 0, data.Length, SocketFlags.None,
    new AsyncCallback(SendData), sock);
```

这个例子发送全部的数据缓存，并且当套接字准备发送数据时，调用SendData()方法。sock Socket对象将被发送到AsyncCallback()方法。

EndSend()方法用于结束发送数据。这个方法的格式如下，IAsyncResult参数用于定义一个空的对象，用于调用BeginSend()方法的结果：

```
int EndSend(IAsyncResult iar)
```

EndSend()方法返回从套接字成功发送的字节数量。

这里有一个使用EndSend() AsyncCallback方法的例子：

```
private static void SendData(IAsyncResult iar)
{
    Socket server = (Socket)iar.AsyncState;
    int sent = server.EndSend(iar);
}
```

原始的sock套接字是通过IAsyncResult对象的AsyncState特性重新创建的，传送到AsyncCallback方法。再一次，又涉及到BeginSend()方法中指定的状态对象。

BeginSendTo()和EndSendTo()方法

BeginSendTo()方法用于连接套接字，从而开始与远程主机进行异步数据的传送：

```
IAsyncResult BeginSendTo(byte[] buffer, int offset, int size,
    SocketFlags sockflag, EndPoint ep, AsyncCallback callback, object state)
```

如用户所见，对于BeginSendTo()方法的绝大部分来说，格式与SendTo()方法格式相类似，同时增加了BeginSend()方法的异步功能。

sBeginSendTo()方法的例子代码看起来如下所示：

```
IPEndPoint iep = new IPEndPoint(IPAddress.Parse("192.168.1.6"), 9050);
sock.BeginSendTo(data, 0, data.Length, SocketFlags.None, iep,
    new AsyncCallback(SendDataTo), sock);
```

这个调用和BeginSend()调用的惟一不同在于，BeginSend()调用是远程主机在参数中增加了IPEndPoint。

EndSendTo()方法使用End方法的标准格式：

```
int EndSendTo(IAsyncResult iar)
```

和其他一样，IAsyncResult参数接受了BeginSendTo()方法返回的IAsyncResult值。EndSendTo()方法返回从套接字发送出去的字节数量。

BeginReceive()和EndReceive()方法

BeginReceive()方法接受从远程主机套接字上发送的数据。第一种方法的格式如下：

```
IAsyncResult BeginReceive(byte[] buffer, int offset, int size,
    SocketFlags sockflag, AsyncCallback callback, object state)
```

第一个参数是字节数组，用于接受外部的数据。offset和size参数用于指定缓存中放置数据的起始位置和长度。sockflags参数用于设置通信的套接字标记。callback和state参数允许用户传送信息到EndReceive()方法。

这里有一个BeginReceive()方法的调用：

```
sock.BeginReceive(data, 0, data.Length, SocketFlags.None,
    new AsyncCallback(ReceivedData), sock);
```

BeginReceive()方法传送原始套接字到EndReceive()方法，这样就可以为AsyncCallback方法重新创建套接字。AsyncCallback方法用于EndReceive()方法，如下：

```
void ReceivedData(IAsyncResult iar)
{
    Socket remote = (Socket)iar.AsyncState;
    int recv = remote.EndReceive(iar);
```

```
string receivedData = Encoding.ASCII.GetString(data, 0, recv);
Console.WriteLine(stringData);
}
```

用户将再一次看到AsyncCallback的行为：第一个语句从IAsyncResult对象重新创建原始套接字。第二个语句使用套接字和EndReceive()方法调用来结束套接字接收。外部发送的数据放置在通过相应的BeginReceive()方法调用指定的数据缓存中。要从AsyncCallback方法访问数据缓存，用户必须确认数据缓存可以被两种方法使用（例如使用全局变量或者数据类）。

BeginReceiveFrom()和EndReceiveFrom()方法

BeginReceiveFrom()方法接受从连接套接字上任何一个远程主机发送的数据，这里是该方法的格式：

```
IAsyncResult BeginReceiveFrom(byte[] buffer, int offset, int size,
    SocketFlags sockflag, ref EndPoint ep, AsyncCallback callback, object state)
```

注意，BeginReceiveFrom()方法与BeginReceive()方法类似，只不过该方法是指定一个关联到EndPoint对象。EndPoint对象用于定义远程主机IP地址和发送数据的端口号码。

警告：记住，BeginReceiveFrom()使用一个关联到EndPoint对象，而不是该对象本身。这里非常容易忽略ref关键词，这样在处理的时候会产生问题。

BeginReceiveFrom()方法的例子如下：

```
sock.BeginReceive(data, 0, data.Length, SocketFlags.None, ref iep, new
    AsyncCallback(ReceiveData), sock);
```

相应的EndReceiveFrom()方法放置在对应的AsyncCallback方法中：

```
void ReceiveData(IAsyncResult iar)
{
    Socket remote = (Socket)iar.AsyncState;
    int recv = remote.EndReceiveFrom(iar);
    string stringData = Encoding.ASCII.GetString(data, 0, recv);
    Console.WriteLine(stringData);
}
```

EndReceiveFrom()方法返回从套接字读取的字节和在BeginReceiveFrom()方法中定义的、放置在数据缓存中的接收数据的数量。再一次，如果用户需要访问在AsyncCallback方法中的数据，用户需要确认数据缓存可以被两种方法使用。

使用异步套接字的例子程序

用户已经学习了所有的片段，现在是将这些片段组合起来创建一个真正的Windows异步网络程序的时候了。下面的两个程序重新创建了在第5章“面向连接的套接字”中介绍的SimpleTcpSrvr和SimpleTcpClient程序（程序清单5.1和程序清单5.2），但是这一次程序使用的是异步网络方法。

客户机程序

AsyncTcpClient.cs程序（程序清单8.2）使用.NET Windows窗体库为简单的TCP客户机创建Windows GUI环境。在Windows环境中，程序对用户Windows事件的响应是非常重要的，在网络中同样重要。

程序清单8.2 AsyncTcpClient.cs程序

```
using System;
using System.Drawing;
using System.Net;
using System.Net.Sockets;
using System.Text;
using System.Windows.Forms;

class AsyncTcpClient Form:
{
    private TextBox newText;
    private TextBox conStatus;
    private ListBox results;
    private Socket client;
    private byte[] data = new byte[1024];
    private int size = 1024;

    public AsyncTcpClient()
    {
        Text = "Asynchronous TCP Client";
        Size = new Size(400, 380);

        Label label1 = new Label();
        label1.Parent = this;
        label1.Text = "Enter text string:";
        label1.AutoSize = true;
        label1.Location = new Point(10, 30);

        newText = new TextBox();
        newText.Parent = this;
        newText.Size = new Size(200, 2 * Font.Height);
        newText.Location = new Point(10, 55);

        results = new ListBox();
        results.Parent = this;
        results.Location = new Point(10, 85);
        results.Size = new Size(360, 18 * Font.Height);

        Label label2 = new Label();
        label2.Parent = this;
```

```
label2.Text = "Connection Status:";
label2.AutoSize = true;
label2.Location = new Point(10, 330);

conStatus = new TextBox();
conStatus.Parent = this;
conStatus.Text = "Disconnected";
conStatus.Size = new Size(200, 2 * Font.Height);
conStatus.Location = new Point(110, 325);

Button sendit = new Button();
sendit.Parent = this;
sendit.Text = "Send";
sendit.Location = new Point(220, 52);
sendit.Size = new Size(5 * Font.Height, 2 * Font.Height);
sendit.Click += new EventHandler(ButtonSendOnClick);

Button connect = new Button();
connect.Parent = this;
connect.Text = "Connect";
connect.Location = new Point(295, 20);
connect.Size = new Size(6 * Font.Height, 2 * Font.Height);
connect.Click += new EventHandler(ButtonConnectOnClick);

Button discon = new Button();
discon.Parent = this;
discon.Text = "Disconnect";
discon.Location = new Point(295, 52);
discon.Size = new Size(6 * Font.Height, 2 * Font.Height);
discon.Click += new EventHandler(ButtonDisconOnClick);
}

void ButtonConnectOnClick(object obj, EventArgs ea)
{
    conStatus.Text = "Connecting...";
    Socket newsock = new Socket(AddressFamily.InterNetwork,
                                SocketType.Stream, ProtocolType.Tcp);
    IPEndPoint iep = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 9050);
    newsock.BeginConnect(iep, new AsyncCallback(Connected), newsock);
}

void ButtonSendOnClick(object obj, EventArgs ea)
{
    byte[] message = Encoding.ASCII.GetBytes(newText.Text);
    newText.Clear();
    client.BeginSend(message, 0, message.Length, SocketFlags.None,
                     new AsyncCallback(SendData), client);
}
```

```

void ButtonDisconOnClick(object obj, EventArgs ea)
{
    client.Close();
    conStatus.Text = "Disconnected";
}
void Connected(IAsyncResult iar)
{
    client = (Socket)iar.AsyncState;
    try
    {
        client.EndConnect(iar);
        conStatus.Text = "Connected to: " + client.RemoteEndPoint.ToString();
        client.BeginReceive(data, 0, size, SocketFlags.None,
            new AsyncCallback(ReceiveData), client);
    } catch (SocketException)
    {
        conStatus.Text = "Error connecting";
    }
}
void ReceiveData(IAsyncResult iar)
{
    Socket remote = (Socket)iar.AsyncState;
    int recv = remote.EndReceive(iar);
    string stringData = Encoding.ASCII.GetString(data, 0, recv);
    results.Items.Add(stringData);
}
void SendData(IAsyncResult iar)
{
    Socket remote = (Socket)iar.AsyncState;
    int sent = remote.EndSend(iar);
    remote.BeginReceive(data, 0, size, SocketFlags.None,
        new AsyncCallback(ReceiveData), remote);
}
public static void Main()
{
    Application.Run(new AsyncTcpClient());
}
}

```

类构造器为程序创建了窗口对象。对于简单的客户机程序，需要创建下列对象：

- 一个Label对象和一个TextBox对象，这些对象允许用户输入文本消息来发送到远程主机
- 一个ListBox对象用于显示从远程主机返回的消息

- 一个Label对象和一个TextBox对象，用于显示客户机程序的连接状态
- 三个Button对象，一个用于建立与远程主机的TCP连接，一个用来发送文本消息和接收回复，一个用于断开TCP对话

对于每个对象的放置需要使用Point类，尺寸需要使用Size类；这些类都是来自System.Drawing命名空间。用户可以手动试验，在Windows窗体中为对象定位。

说明：如果用户拥有一个Microsoft Visual产品，例如Visual Studio或者Visual C#，用户可以使用图形编辑环境来放置所有的Windows对象，并且可以自动创建窗体代码。

客户机程序流

用户可能已经注意到，在类构造器中，根本没有网络编程代码。所有的网络功能都在EventHandler()方法中发生。要让AsyncTcpClient程序执行和原始SimpleTcpClient程序相同的方式，用户必须首先确定如何将程序协议和程序中将要触发的异步事件连接。然后用户必须将其编入Windows事件代码方法中。在图8.3中的图形说明了每种需求的方法和这些方法如何与Windows事件代码进行交互。随后的段落描述了通过简单客户机程序达到目的的步骤：连接、接收、发送和断开。

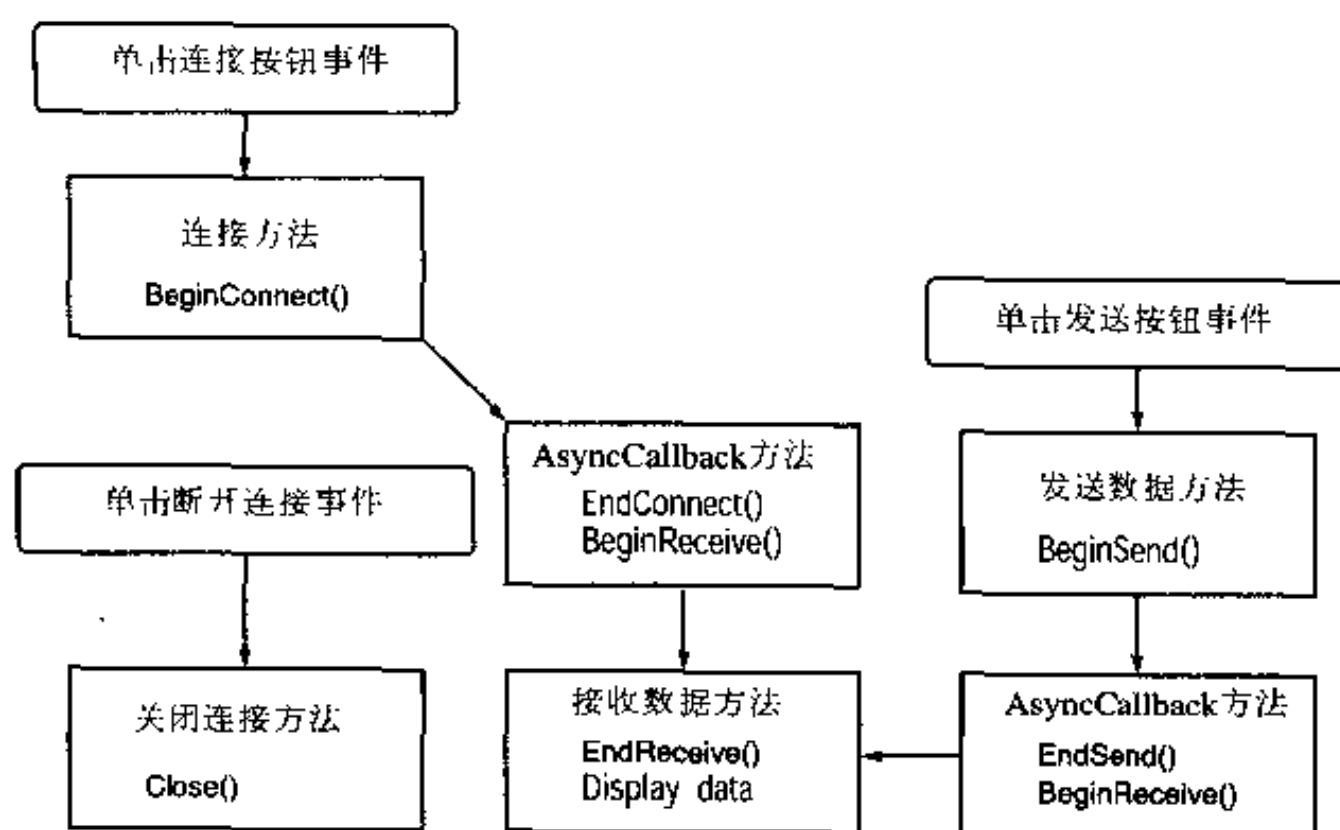


图8.3 异步事件简图

连接

如图中所示，当客户单击连接（Connect）按钮，用于启动连接的方法被执行。在程序代码中，这种方法是ButtonConnectOnClick()方法：

```

void ButtonConnectOnClick(object obj, EventArgs ea)
{
    conStatus.Text = "Connecting...";
    Socket newsock = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
        ProtocolType.Tcp);

```

```

IPEndPoint iep = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 9050);
newsock.BeginConnect(iep, new AsyncCallback(Connected), newsock);
}

```

ButtonConnectOnClick()方法创建了一个新的Socket对象用于通信。然后该方法使用远程主机地址信息、关联的AsyncCallback方法名称和最新创建的Socket对象启动BeginConnect()方法。当方法结束执行，Windows等待BeginConnect()方法触发事件，该事件表明，已经与远程的主机成功连接。当事件发生，程序控制将传送到Connected()方法：

```

void Connected(IAsyncResult iar)
{
    client = (Socket)iar.AsyncState;
    try
    {
        client.EndConnect(iar);
        conStatus.Text = "Connected to: " + client.RemoteEndPoint.ToString();
        client.BeginReceive(data, 0, size, SocketFlags.None,
            new AsyncCallback(ReceiveData), client);
    } catch (SocketException)
    {
        conStatus.Text = "Error connecting";
    }
}

```

在Connected()方法中首先要做的事情就是得到用于连接的原始Socket对象。IAsyncResult类的AsyncState特性返回传递到AsyncCallback方法的对象，在这种情况下BeginConnect()将创建新的Socket对象。

当Socket对象被重新建立，EndConnect()方法将被执行。因为EndConnect()方法存在失败的可能性（例如，如果远程主机无法利用），将其放置在try-catch模块中比较合适。如果EndConnect()失败，用户可以通过conStatus TextBox对象来通知客户。

在这个程序协议中，当连接建立后，服务器首要的事情就是发送欢迎标题。要完成这项工作，AsyncTcpClient程序必须准备在成功建立连接后立即接收一个外部的消息。这项工作是在Connected()方法结束后使用BeginReceive()方法调用来实现的。

接收数据

在新连接建立后，每个消息发送后都要执行ReceiveData()方法。BeginReceive()方法说明ReceiveData()方法，当接收到数据后，执行AsyncCallback方法：

```

void ReceiveData(IAsyncResult iar)
{
    Socket remote = (Socket)iar.AsyncState;
    int recv = remote.EndReceive(iar);
    string stringData = Encoding.ASCII.GetString(data, 0, recv);
    results.Items.Add(stringData);
}

```


再一次，第一条语句重新创建通信套接字。在原始套接字重新创建后，使用**EndReceive()**方法，参考原始**IAAsyncResult**对象，该过程和原始的**BeginReceive()**方法调用相对应。当从远程主机接收到消息，将被放置在与**BeginReceive()**方法调用关联的数据缓存中。因为数据变量定义为全局变量，方法使用该变量将消息放置在**results ListBox**中。

发送数据

在**TextBox**对象中键入一条消息后，客户单击**Send**发送按钮，执行**SendData()**方法。**EventHandler**用于将按钮指向**ButtonSendOnClick()**方法：

```
void ButtonSendOnClick(object obj, EventArgs ea)
{
    byte[] message = Encoding.ASCII.GetBytes(newText.Text);
    newText.Clear();
    client.BeginSend(message, 0, message.Length, SocketFlags.None,
        new AsyncCallback(SendData), client);
}
```

消息从**TextBox**对象中摘取，转换到字节数组，并且发出套接字。因为**ButtonSendOnClick()**方法没有原始的**Socket**对象进行传递，因此需要使用一个类成员来引用连接的套接字。**BeginSend()**方法用于连接套接字。该方法指定当**BeginSend()**函数准备结束时调用的消息和**AsyncCallback**方法，以及将要传递到**EndSend()**方法的对象。在这里，该对象为连接的套接字。

当套接字指示准备发送消息时，触发**AsyncCallback**方法**SendData()**：

```
void SendData(IAAsyncResult iar)
{
    Socket remote = (Socket)iar.AsyncState;
    int sent = remote.EndSend(iar);
    remote.BeginReceive(data, 0, size, SocketFlags.None,
        new AsyncCallback(ReceiveData), remote);
}
```

再一次，用户可以看到类似的行为：原始连接的套接字是使用从**BeginSend()**方法传递的状态对象的**AsyncState**重新创建的。当原始套接字被重新创建，**EndSend()**方法结束数据传输。**IAAsyncResult**对象将**EndSend()**与原始的**BeginSend()**和**EndSend()**返回的从套接字实际发送出的字节数量结合。

在这个应用程序中，当消息发送后，服务器等待回应。了解这些，执行**BeginReceive()**方法调用来启动接收过程。**BeginReceive()**再一次调用同样的**ReadData()**方法，就像接收原始连接的欢迎标题一样。

断开连接

现在，用户已经初步掌握了事件编程，最后，用户可以创建一个按钮来控制从远程主机断开（而不是定义一个控制字来停止客户机）。这是工作的原理：

```
void ButtonDisconOnClick(object obj, EventArgs ea)
{
    client.Close();
    conStatus.Text = "Disconnected";
}
```

通过为客户机套接字分发Close()方法, 连接将被断开。要建立一个新的连接, 客户可以单击Connect连接按钮。

警告: 在这个简单的客户机例子中, 很多地方可能会产生错误, 包括客户机单击按钮的错误。一个真正的产品应用程序将在每次单击按钮时进行检测, 确保功能可以执行。

测试异步客户机

因为AsyncTcpClient程序使用第5章中SimpleTcpClient程序相同的协议模型, 用户可以使用SimpleTcpSrvr程序来测试新的异步版本。

1. 在服务器的命令行提示窗口启动SimpleTcpSrvr。
2. 启动AsyncTcpClient程序, 可以在单独的命令行窗口或者其他的网络设备上。
3. 单击Connect按钮。状态窗口将显示客户机已经连接到服务器, 并且服务器的欢迎标题将返回显示在列表框中, 如图8.4所示。

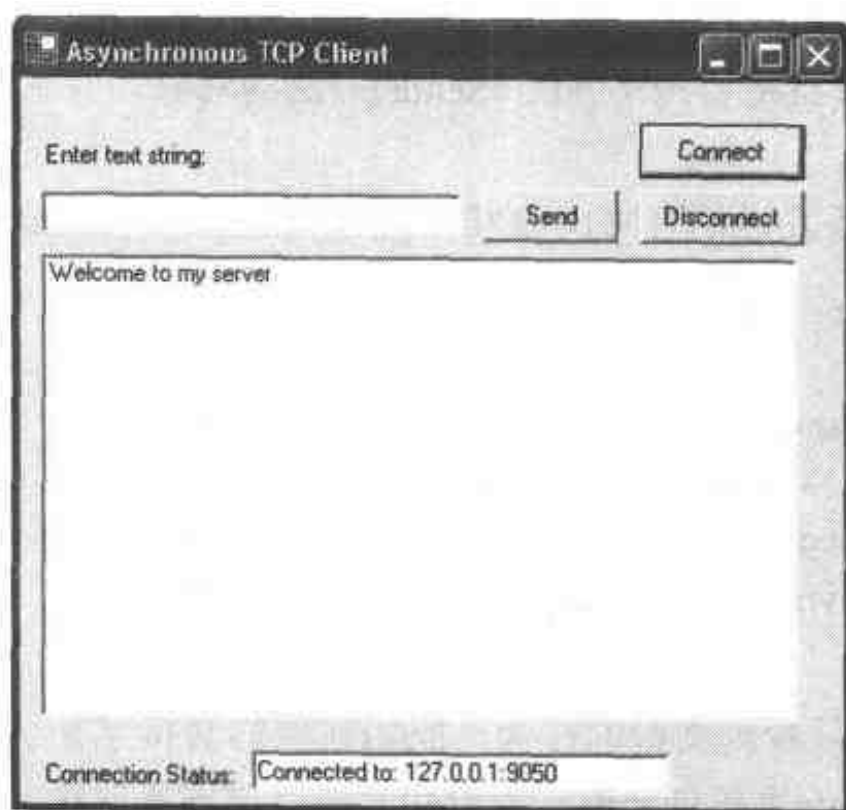


图8.4 AsyncTcpClient程序, 在列表框中显示服务器的欢迎标题

4. 在原始连接后, 通过单击Send按钮, 试图将文本框中的消息发送到服务器程序。每个消息将被显示在服务器的SimpleTcpSrvr控制台上, 并且显示在AsyncTcpClient程序的列表框中。
5. 要停止连接, 单击Disconnect按钮。SimpleTcpSrvr程序将被中断, 并且在AsyncTcpClient程序中的状态文本框将显示客户机已经从服务器断开。

服务器程序

用户已经将客户机程序移入Windows世界，现在是对付服务器程序的时候了。程序清单8.3显示的AsyncTcpSrvr.cs程序，该程序模拟第5章中的SimpleTcpSrvr程序的功能（程序清单5.1），但是使用了一个Windows窗体环境和异步套接字。

程序清单8.3 AsyncTcpSrvr.cs程序

```
using System;
using System.Drawing;
using System.Net;
using System.Net.Sockets;
using System.Text;
using System.Windows.Forms;

class AsyncTcpSrvr Form:
{
    private TextBox conStatus;
    private ListBox results;
    private byte[] data = new byte[1024];
    private int size = 1024;
    private Socket server;

    public AsyncTcpSrvr()
    {
        Text = "Asynchronous TCP Server";
        Size = new Size(400, 380);

        results = new ListBox();
        results.Parent = this;
        results.Location = new Point(10, 65);
        results.Size = new Size(350, 20 * Font.Height);

        Label label1 = new Label();
        label1.Parent = this;
        label1.Text = "Text received from client:";
        label1.AutoSize = true;
        label1.Location = new Point(10, 45);

        Label label2 = new Label();
        label2.Parent = this;
        label2.Text = "Connection Status:";
        label2.AutoSize = true;
        label2.Location = new Point(10, 330);

        conStatus = new TextBox();
        conStatus.Parent = this;
```

```
conStatus.Text = "Waiting for client...";
conStatus.Size = new Size(200, 2 * Font.Height);
conStatus.Location = new Point(110, 325);

Button stopServer = new Button();
stopServer.Parent = this;
stopServer.Text = "Stop Server";
stopServer.Location = new Point(260, 32);
stopServer.Size = new Size(7 * Font.Height, 2 * Font.Height);
stopServer.Click += new EventHandler(ButtonStopOnClick);

server = new Socket(AddressFamily.InterNetwork,
    SocketType.Stream, ProtocolType.Tcp);
IPEndPoint iep = new IPEndPoint(IPAddress.Any, 9050);
server.Bind(iep);
server.Listen(5);
server.BeginAccept(new AsyncCallback(AcceptConn), server);
}

void ButtonStopOnClick(object obj, EventArgs ea)
{
    Close();
}

void AcceptConn(IAsyncResult iar)
{
    Socket oldserver = (Socket)iar.AsyncState;
    Socket client = oldserver.EndAccept(iar);
    conStatus.Text = "Connected to: " + client.RemoteEndPoint.ToString();
    string stringData = "Welcome to my server";
    byte[] message1 = Encoding.ASCII.GetBytes(stringData);
    client.BeginSend(message1, 0, message1.Length, SocketFlags.None,
        new AsyncCallback(SendData), client);
}

void SendData(IAsyncResult iar)
{
    Socket client = (Socket)iar.AsyncState;
    int sent = client.EndSend(iar);
    client.BeginReceive(data, 0, size, SocketFlags.None,
        new AsyncCallback(ReceiveData), client);
}

void ReceiveData(IAsyncResult iar)
{
    Socket client = (Socket)iar.AsyncState;
    int recv = client.EndReceive(iar);
}
```

```
if (recv == 0)
{
    client.Close();
    conStatus.Text = "Waiting for client...";
    server.BeginAccept(new AsyncCallback(AcceptConn), server);
    return;
}
string receivedData = Encoding.ASCII.GetString(data, 0, recv);
results.Items.Add(receivedData);
byte[] message2 = Encoding.ASCII.GetBytes(receivedData);
client.BeginSend(message2, 0, message2.Length, SocketFlags.None,
    new AsyncCallback(SendData), client);
}

public static void Main()
{
    Application.Run(new AsyncTcpSrvr());
}
```

与客户机程序一样，AsyncTcpSrvr程序使用基本的Windows窗体对象来创建客户图形环境：

- 一个Label对象和一个ListBox对象，用于识别从连接的客户机接收到的数据。
- 一个Label对象和一个TextBox对象，用于显示套接字和连接的客户机的状态。
- 一个Button对象，允许客户停止服务器并推出程序。

在创建了客户接口后，用户必须编写网络编程部分的代码。与客户机不同，AsyncTcpSrvr程序包含网络编程代码，必须在类构造器中完成。

服务器程序流程

与第5章的SimpleTcpSrvr不同，AsyncTcpSrvr程序必须遵守一套用于与远程客户机通信的程序。图8.5说明了服务器需要的步骤。

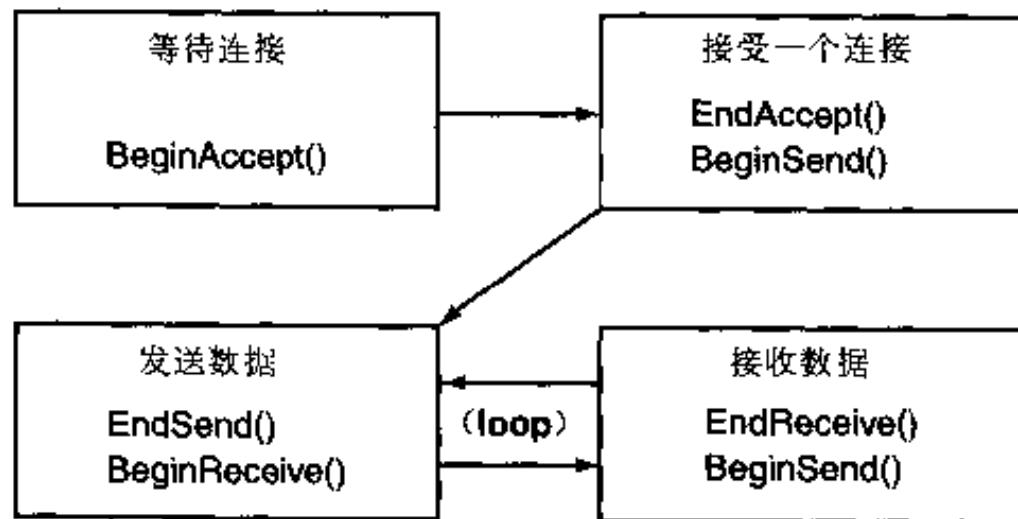


图8.5 AsyncTcpSrvr过程

等待新客户连接

如用户所知，服务器必须立即检测外部的连接，当窗体对象创建后，需要的网络编程代码附加在类构造器中：

```
server = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
    ProtocolType.Tcp);
IPEndPoint iep = new IPEndPoint(IPAddress.Any, 9050);
server.Bind(iep);
server.Listen(5);
server.BeginAccept(new AsyncCallback(AcceptConn), server);
```

现在，用户认为它是标准的服务器套接字代码：创建一个新的Socket对象，并将该对象绑定到本地的IPEndPoint对象，检测新的连接。当连接接收到后，BeginAccept()方法指明使用AsyncCallback方法，并且对象传送到AsyncCallback对象。

当连接尝试被检测到后，执行AsyncCallback方法注册的BeginAccept()方法：

```
void AcceptConn(IAsyncResult iar)
{
    Socket oldserver = (Socket)iar.AsyncState;
    Socket client = oldserver.EndAccept(iar);
    conStatus.Text = "Connected to: " + client.RemoteEndPoint.ToString();
    string stringData = "Welcome to my server";
    byte[] message1 = Encoding.ASCII.GetBytes(stringData);
    client.BeginSend(message1, 0, message1.Length, SocketFlags.None,
        new AsyncCallback(SendData), client);
}
```

AsyncCallback方法使用IAsyncResult对象的AsyncState特性重新创建原始的Socket对象。当调用EndAccept()时，该方法使用IAsyncResult对象和BeginAccept()对象组合。EndAccept()方法返回新的Socket对象，用于与远程客户机的所有通信。

如程序流程所反映的，服务器下一步工作是发送欢迎标题到客户机。这就需要使用BeginSend()方法，当套接字准备发送数据时，该方法注册一个AsyncCallback方法来结束发送操作。

警告：注意，传递到AsyncCallback方法的Socket对象是最新创建的客户机套接字，而不是原始的服务器套接字。该套接字是连接到远程客户机的套接字，并且用于与客户机的所有通信。

发送数据

EndSend()方法是通过注册在BeginSend()方法中的AsyncCallback方法建立的：

```
void SendData(IAsyncResult iar)
{
    Socket client = (Socket)iar.AsyncState;
    int sent = client.EndSend(iar);
    client.BeginReceive(data, 0, size, SocketFlags.None,
        new AsyncCallback(ReceiveData), client);
}
```


这个例子程序使用了标准的**EndSend()**格式。一个优秀的产品程序同样需要将这个值与发送消息的原始字节数量进行比较，并且如果某些信息没有正确发送的话，将要再次执行**BeginSend()**方法。

服务器程序流模型指明，在发送环境标题后，服务器将等待客户机发送的外部消息。这与**BeginReceive()**方法相一致，该方法指明一个数据缓存，在这个缓存中放置接收到的数据，在接收到数据后，调用**AsyncCallback**方法。

接收数据

EndReceive()方法放置在**AsyncCallback**方法中，用于**BeginReceive()**方法：

```
void ReceiveData(IAsyncResult iar)
{
    Socket client = (Socket)iar.AsyncState;
    int recv = client.EndReceive(iar);
    if (recv == 0)
    {
        client.Close();
        conStatus.Text = "Waiting for client...";
        server.BeginAccept(new AsyncCallback(AcceptConn), server);
        return;
    }
    string receivedData = Encoding.ASCII.GetString(data, 0, recv);
    results.Items.Add(receivedData);
    byte[] message2 = Encoding.ASCII.GetBytes(receivedData);
    client.BeginSend(message2, 0, message2.Length, SocketFlags.None, new
        AsyncCallback(SendData), client);
}
```

与其他方法相比，**ReceiveData()**方法有一点冗长，这是因为该方法需要在程序流中适应两种方案。首先，使用标准的**AsyncState**特性创建客户机套接字。接下来，调用**EndReceive()**方法，使用**IAsyncResult**对象将该方法与适当的**BeginReceive()**方法结合。**EndReceive()**返回从套接字接收到的字节数量。如前面说过的，真正的产品程序必须将接收到的字节数量和预期的消息尺寸进行对比检测。

如果接收到的消息为0个字节，可以假设远程的客户机断开了**TCP**对话。在这种情况下，客户机套接字关闭，服务器通过使用**BeginAccept()**方法开始侦听新的客户机连接，并且在此重新启动整个程序流程。

如果从远程客户机接收到一个消息，将显示在**ListBox**对象，并且给客户机进行反应。这个过程是通过再次使用**BeginSend()**方法实现的。因为程序流程现在从发送欢迎标题复制了原始流程，原始的**SendData()**方法将用于**AsyncCallback**方法。

进行程序联合测试

AsyncTcpSrvr程序可以和**AsyncTcpClient**程序或者第5章中原始的**SimpleTcpClient**程序一

起进行测试。

1. 在命令提示行窗口或者Windows资源管理器中启动AsyncTcpClient和SimpleTcpClient程序。
2. 当启动AsyncTcpClient程序，单击Connect按钮。服务器和客户机程序都将进行识别，确认连接已经建立，并且显示远程主机的地址。图8.6显示的是客户机连接后AsyncTcpSrvr的输出窗口。

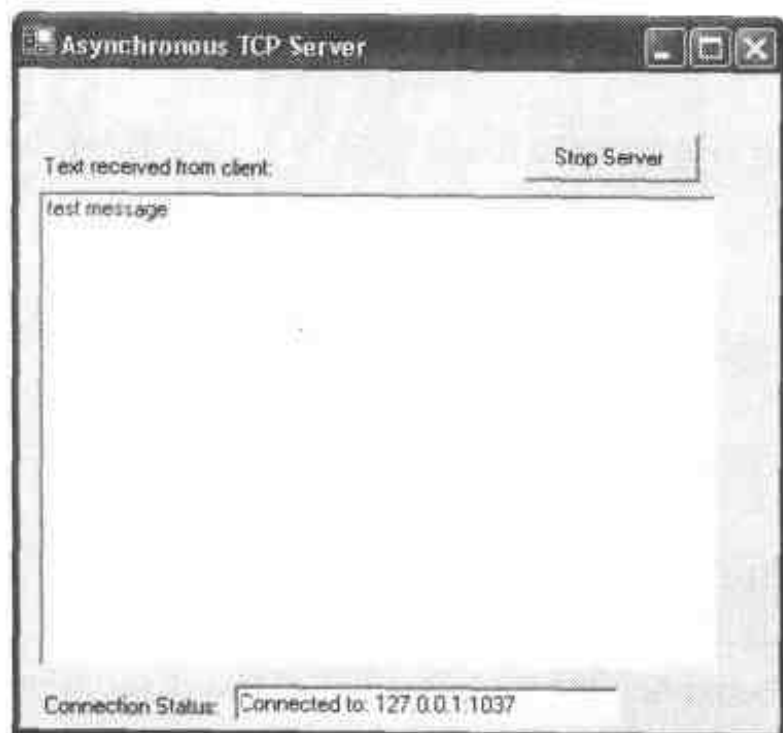


图8.6 AsyncTcpSrvr程序窗口

3. 从AsyncTcpClient发送一个消息。AsyncTcpClient程序每次发送消息，消息都将显示在AsyncTcpClient的列表框中，并且回应AsyncClient列表框。
4. 要关闭连接，在客户机程序上单击Disconnect按钮。在关闭当前客户机后，AsyncTcpSrvr程序将等待新的客户机连接请求。

使用非阻塞套接字方法

异步套接字专门为Windows环境而创建。如第3章讲述的，传统的UNIX网络编程不需要使用异步套接字。相反，程序包含非阻塞的套接字methods，在没有实际确认的阻塞功能下对套接字进行检测。这样就允许程序继续执行，即使网络操作会阻塞程序。如果套接字准备用于网络功能，阻塞功能将被执行。

.NET套接字库包括非阻塞的套接字方法，这些方法建立在UNIX套接字库的基础上。这样就允许网络编程者很容易地将UNIX网络程序移植到Windows环境。这一节将描述Poll()和Select()方法，用于检测网络功能是否被阻塞，并且帮助网络功能避免受到阻塞。

Poll()方法

通常情况下，当用户试图执行一个阻塞网络功能，例如Receive()命令，在开始执行命令前，用户需要具有检测套接字的能力。

Socket方法**Poll()**提供给你这种能力。该方法检测**Socket**对象，从而确认网络方法调用是否阻塞或者可以成功地结束。如果指示说明方法将在没有阻塞的情况下执行，用户就自由了。否则，用户可以执行一些其他的功能，并在最后再次进行检测。

Poll()方法的格式如下：

```
bool Poll(int microseconds, SelectMode mode);
```

该方法返回一个简单的**boolean**值：如果行为可以结束，返回**true**，否则，如果行为被阻塞，返回**false**。

int参数允许用户设置大量的时间（毫秒级），用于**Poll()**方法进行等待并观察套接字指示的事件。使用**SelectMode**参数用于说明要观察何种类型的行为。**SelectMode**类列举了三种可能的事件，用于**Poll()**方法进行监视：

SelectRead **SelectRead**值用于**SelectMode**，将导致**Poll()**在下面的情况下返回**true**值：

- 如果**Accept()**方法调用将要成功
- 如果套接字上的数据可以被使用
- 如果连接被关闭

SelectWrite **SelectWrite**值用于**SelectMode**，在下面的情况下将导致**Poll()**方法返回**true**值：

- 如果**Connect()**方法调用已经成功
- 如果套接字上的数据可以被发送

SelectError **SelectError**值用于**SelectMode**，在下面的情况下将导致**Poll()**方法返回**true**值：

- 如果**Connect()**方法调用失败
- 如果带外的数据可用，并且**Socket OutOfBandInline**特性没有被设置

基于**SelectMode**值的设置，用户可以通过检测**Poll()**方法返回的值，来确定套接字是否准备好用于阻塞功能。最后的**Poll()**方法调用如下：

```
result = sock.Poll(1000000, SelectMode.SelectRead);
```

Socket对象**sock**，将被检测1 000 000微秒（1秒），来确定套接字上的数据是否准备好。如果返回值为**true**，**Receive()**方法将在没有阻塞的情况下成功结束。

Poll()程序的例子

程序清单8.4中的**TcpPollSrvr.cs**程序，示范了在服务器程序中**Poll()**方法工作的方式。

程序清单8.4 TcpPollSrvr.cs程序

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class TcpPollSrvr
```

```
{
    public static void Main()
    {
        int recv;
        byte[] data = new byte[1024];
        IPEndPoint ipep = new IPEndPoint(IPAddress.Any,
                                          9050);

        Socket newsock = new
            Socket(AddressFamily.InterNetwork,
                  SocketType.Stream, ProtocolType.Tcp);

        newsock.Bind(ipep);
        newsock.Listen(10);
        Console.WriteLine("Waiting for a client...");
        bool result;
        int i = 0;
        while(true)
        {
            i++;
            Console.WriteLine("polling for accept#{0}...", i);
            result = newsock.Poll(1000000, SelectMode.SelectRead);
            if (result)
            {
                break;
            }
        }
        Socket client = newsock.Accept();

        IPEndPoint newclient =
            (IPEndPoint)client.RemoteEndPoint;
        Console.WriteLine("Connected with {0} at port {1}",
                          newclient.Address, newclient.Port);

        string welcome = "Welcome to my test server";
        data = Encoding.ASCII.GetBytes(welcome);
        client.Send(data, data.Length,
                    SocketFlags.None);

        i = 0;
        while(true)
        {
            Console.WriteLine("polling for receive #{0}...", i);
            i++;
            result = client.Poll(3000000, SelectMode.SelectRead);
            if(result)
            {

```

```

        data = new byte[1024];
        i = 0;
        recv = client.Receive(data);
        if (recv == 0)
            break;

        Console.WriteLine(
            Encoding.ASCII.GetString(data, 0, recv));
        client.Send(data, recv, 0);
    }
}

Console.WriteLine("Disconnected from {0}",
    newclient.Address);
client.Close();
newsock.Close();
}
}

```

在创建服务器套接字后，使用标准的Bind()和Listen() Socket方法绑定到本地IPEndPoint对象和放置在收听方式。在标准的阻塞程序中，下一步是使用Accept()方法等待来自外部客户端机的连接请求。这将引起程序进行等待，并且禁止进行其他的网络活动，直到接收到外部连接。

相反，在这个例子中，非阻塞Poll()方法轮询套接字，以检测外部的连接是否可用：

```

while(true)
{
    Console.WriteLine("polling for receive #{0}...", i);
    i++;
    result = client.Poll(3000000, SelectMode.SelectRead);
    if(result)
    {
        data = new byte[1024];
        i = 0;
        recv = client.Receive(data);
        if (recv == 0)
            break;

        Console.WriteLine(
            Encoding.ASCII.GetString(data, 0, recv));
        client.Send(data, recv, 0);
    }
}

```

在这里，`while()`循环用于创建持续的循环，检测套接字以便得到`SelectRead`属性。如果`SelectRead`属性已经设置，`Poll()`将返回`true`，表示外部连接可用，并且可以在没有阻塞的情况下安全使用`Accept()`方法。

在调用`Poll()`方法前，程序可以执行任何功能，包括调用其他方法。这个程序仅仅是增加了一个计数器并在控制台上进行显示，提示用户程序在轮询的过程中仍然可以运行。

在外部连接建立后，其他的轮询循环将等待外部数据，并且使用`Receive()`方法。再一次，在`Poll()`调用之间，程序可以执行任何需要的程序。

测试Poll()方法

用户可以通过在命令提示符下运行该程序来进行测试，观察程序的轮询，并且等待外部的连接。无论是第5章中的`SimpleTcpClient`程序还是本章前面创建的`AsyncTcpClient`程序，都可以连接到`TcpPollSrvr`。在运行`TcpPollSrvr`几秒钟后，启动客户程序来建立连接。一旦检测到连接，`Poll()`返回一个`true`值并且程序继续执行，为客户机发送欢迎标题，然后进入下一个轮询来接收数据。

下面的例子来自这个测试：

```
C:\>TcpPollSrvr
Waiting for a client...
polling for accept#1...
polling for accept#2...
polling for accept#3...
polling for accept#4...
polling for accept#5...
polling for accept#6...
polling for accept#7...
Connected with 192.168.1.2 at port 1985
polling for receive #0...
polling for receive #1...
polling for receive #2...
polling for receive #3...
This is a test message.
polling for receive #0...
polling for receive #1...
polling for receive #2...
This is another test message.
polling for receive #0...
Disconnected from 192.168.1.2

C:\>
```

如用户所见，程序轮询并且等待外部连接。一个连接建立，程序将再次轮询，等待数据。当每次收到消息，将建立一个新的轮询，再次等待数据。当远程客户机关闭连接，`Poll()`确定关闭并且返回一个`true`值。因为返回的值为`true`，因此将调用`Receive()`方法并且返回一个

0值。

说明：与Accept()或者Receive()方法不同，Connect()方法必须在Poll()方法前启动并使用，要确认套接字不会被阻塞，用户必须手动使用非阻塞的套接字操作。

Poll()方法可以正常运行，等待单独的套接字，但是如果要是监视多个套接字，将是极其复杂的。在这种情况下，用户需要使用Select()方法。

Select()方法

Socket类的Select()方法可以在阻塞功能中轮询一个或者多个套接字。当套接字可以被用于读取或者写入，Select()方法可以检测到哪个套接字可以被使用，哪个套接字不能使用。

Select()格式

因为Select()方法被声明为静态方法，因此该方法无法在Socket对象的例子中使用。相反，用户必须在一般的Socket声明后使用该方法，这里是该方法的格式：

```
Socket.Select(IList checkRead, IList checkWrite, IList checkError, int
microseconds)
```

三个IList对象代表对套接字行为的三种监视：

- checkRead监视指定的套接字，用于监视从套接字上读取数据的能力。
- checkWrite监视指定的套接字，用于监视向套接字写入数据的能力。
- checkError监视指定的套接字，用于监视错误情况下的套接字。

IList对象代表一个对象集合，该集合可以被一个索引值单独访问。当Select()方法退出，每个IList对象将被修改，只允许包含符合状态的Socket对象（读取、写入或者错误）。

IList对象必须允许Select()方法对列表中的条目进行操作，因为套接字作为数据被增加和删除。创建可以被操作的IList对象的最简单方法是使用ArrayList对象。ArrayList类可以在System.Collections命名空间中找到。在Select()方法中，使用了Socket对象的ArrayList。microseconds参数用于检测Select()方法检测套接字的时间长度（类似Poll()方法）。

例如下面的代码片段创建三个单独的套接字，并且用于监视最近的连接：

```
IPEndPoint iep1 = new IPEndPoint(IPAddress.Any, 9050);

Socket sock = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
ProtocolType.Tcp);

sock.Bind(iep1);
sock.Listen(5);
Socket newsock1 = sock.Accept();

Socket newsock2 = sock.Accept();
Socket newsock3 = sock.Accept();
ArrayList socketList = new ArrayList(3);
socketList.Add(newsock1);
socketList.Add(newsock2);
```

```
socketList.Add(newsock3);

Socket.Select(sockList, null, null, 10000000);
for(int I = 0; I < sockList.Count; i++)
{
    client = (Socket)sockList[i];
    data = new byte[1024];
    recv = client.Receive(data);
}
```

这个代码片段表明在ArrayList中放置三个套接字的位置，并且传送到Select()方法用于监视。三个套接字是客户机连接到放置在收听方式的单独套接字的结果。Select()方法等待10秒，然后退出。结果是sockList ArrayList将包含等待读取数据的套接字。

警告：一定要记住，传递到Select()方法的ArrayList将被修改。在返回时，ArrayList中只包含通过Select过滤的套接字。如果用户需要使用原始的套接字ArrayList，在传递到Select()方法前，一定要进行备份。

在Select()方法返回后，ArrayList中的单独套接字可以被提取，并进行数据读取。

Select()程序示例

本节将讲述如何使用Select()方法在程序中同时监视多个套接字。对于服务器程序来说，该技术特别有用，因为服务器程序需要同时为多个客户机提供服务。

Select()服务器程序

SelectTcpSrvr.cs程序（程序清单8.5）使用Select()方法同时监视两个独立的客户机。该程序创建了一个单独的套接字，放置在收听方式，并且等待两个客户机连接到套接字。每个连接都会产生单独的套接字，套接字放置在ArrayList中。套接字的ArrayList使用Select()方法监视外部发送的数据。当套接字上检测到数据，将对客户机进行回应。

程序清单8.5 SelectTcpSrvr.cs程序

```
using System;
using System.Collections;
using System.Net;
using System.Net.Sockets;
using System.Text;

class SelectTcpSrvr
{
    public static void Main()
    {
        ArrayList sockList = new ArrayList(2);
        ArrayList copyList = new ArrayList(2);
        Socket main = new Socket(AddressFamily.InterNetwork,
```



```
        SocketType.Stream, ProtocolType.Tcp);

    IPEndPoint iep = new IPEndPoint(IPAddress.Any, 9050);
    byte[] data = new byte[1024];
    string stringData;
    int recv;

    main.Bind(iep);
    main.Listen(2);

    Console.WriteLine("Waiting for 2 clients...");
    Socket client1 = main.Accept();
    IPEndPoint iep1 = (IPEndPoint)client1.RemoteEndPoint;
    client1.Send(Encoding.ASCII.GetBytes("Welcome to my server"));
    Console.WriteLine("Connected to {0}", iep1.ToString());
    sockList.Add(client1);

    Console.WriteLine("Waiting for 1 more client...");
    Socket client2 = main.Accept();
    IPEndPoint iep2 = (IPEndPoint)client2.RemoteEndPoint;
    client2.Send(Encoding.ASCII.GetBytes("Welcome to my server"));
    Console.WriteLine("Connected to {0}", iep2.ToString());
    sockList.Add(client2);
    main.Close();

    while(true)
    {
        copyList = new ArrayList(sockList);
        Console.WriteLine("Monitoring {0} sockets...", copyList.Count);
        Socket.Select(copyList, null, null, 10000000);

        foreach(Socket client in copyList)
        {
            data = new byte[1024];
            recv = client.Receive(data);
            stringData = Encoding.ASCII.GetString(data, 0, recv);
            Console.WriteLine("Received: {0}", stringData);
            if (recv == 0)
            {
                iep = (IPEndPoint)client.RemoteEndPoint;
                Console.WriteLine("Client {0} disconnected.", iep.ToString());
                client.Close();
                sockList.Remove(client);
                if (sockList.Count == 0)
                {
                    Console.WriteLine("Last client disconnected, bye");
                    return;
                }
            }
        }
    }
}
```

```

        }
    }
    else
        client.Send(data, recv, SocketFlags.None);
    }
}
}
}
}

```

SelectTcpSrvr程序通过创建一个TCP套接字来监听外部的连接，因此非常简单。该程序使用两个单独的**Accept()**方法调用来等待两个客户机连接。每个客户机连接都保存在单独的**Socket**对象中，每个**Socket**对象都被加入**ArrayList sockList**，用于**Select()**方法。

当创建了**ArrayList**对象后，程序进入一个循环，使用**Select()**方法对**ArrayList**对象进行检测，用于外部发送的数据。为了避免**Select()**方法中使用的**ArrayList**被消除，在每次使用中都会创建一个副本：

```

copyList = new ArrayList(sockList);
Console.WriteLine("Monitoring {0} sockets...", copyList.Count);
Socket.Select(copyList, null, null, 10000000);

```

当任何一个套接字接收到数据，或者出现10秒钟的空白（无论哪个先到），**Select()**方法都会退出。**copyList ArrayList**对象将包含接收到数据的套接字。**for**循环将在**copyList**数组返回的所有套接字中进行重复。

在**for**循环中，第一步就是得到接收到数据的套接字。包含在**copyList ArrayList**中的**Socket**对象将复制到**client Socket**对象中。然后该对象可以被任何普通的套接字使用，使用**Receive()**方法等待接收数据：

```

client = (Socket)copyList[i];
data = new byte[1024];
recv = client.Receive(data);

```

Receive()方法的返回值被检测，已确定远程客户机是否断开了对话。该操作可以通过检测一个零返回值来完成。如果远程客户机断开，用户必须关闭套接字并从**ArrayList**中删除。通过使用**Remove()**方法，可以轻松地完成这项工作：

```

if (recv == 0)
{
    iep = (IPEndPoint)client.RemoteEndPoint;
    Console.WriteLine("Client {0} disconnected.", iep.ToString());
    client.Close();
    sockList.Remove(client);
    if (sockList.Count == 0)
    {
        Console.WriteLine("Last client disconnected, bye");
    }
}

```



```

        return;
    }
}

```

警告：当ArrayList中不再留有任何套接字，要确保结束Select()循环，否则用户将得到一个异常错误。

最后，如果套接字接收到正确的数据，套接字将使用标准的Send()方法向原始的客户机进行回应。当for循环结束后，整个处理将在此开始（在原始的套接字ArrayList被复制回到工作副本后）。

Select()客户机程序

SelectTcpClient.cs程序（程序清单8.6）是一个客户机程序例子，用于连接指定的服务器，并且等待客户在控制台中输入文本信息。消息发送到SelectTcpSrvr程序，该程序将回应客户机。消息显示在控制台中，并且程序等待其他的消息被键入。SelectTcpClient程序本身不需要使用Select()方法，因为该程序只与一个服务器进行连接。

程序清单8.6 SelectTcpClient.cs程序

```

using System;
using System.Collections;
using System.Net;
using System.Net.Sockets;
using System.Text;

class SelectTcpClient
{
    public static void Main()
    {
        Socket sock = new Socket(AddressFamily.InterNetwork,
                                   SocketType.Stream, ProtocolType.Tcp);
        IPEndPoint iep = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 9050);
        byte[] data = new byte[1024];
        string stringData;
        int recv;

        sock.Connect(iep);
        Console.WriteLine("Connected to server");
        recv = sock.Receive(data);
        stringData = Encoding.ASCII.GetString(data, 0, recv);
        Console.WriteLine("Received: {0}", stringData);

        while(true)
        {
            stringData = Console.ReadLine();
            if (stringData == "exit")

```

```
        break;
        data = Encoding.ASCII.GetBytes(stringData);
        sock.Send(data, data.Length, SocketFlags.None);
        data = new byte[1024];
        recv = sock.Receive(data);
        stringData = Encoding.ASCII.GetString(data, 0, recv);
        Console.WriteLine("Received: {0}", stringData);
    }
    sock.Close();
}
}
```

测试SelectTcpClient.cs程序

要测试这个设置，用户需要打开三个命令提示行窗口——一个用于SelectTcpSrvr程序，另外两个用于SelectTcpClient程序。如果需要，用户可以从网络上的其他的工作站运行SelectTcpClient程序；记住要为服务器地址设置IPEndPoint值。

当用户启动SelectTcpSrvr程序，该程序将等待来自客户机的两个连接。当两个连接都连接到服务器后，在任意时间，两个客户机都可以向服务器发送消息。

服务器程序监视用于外部数据的所有套接字。Console.WriteLine()方法显示状态信息，所以用户可以了解在服务器上发生的任何情况。这里是服务器数据的示例：

```
C:\>SelectTcpSrvr
Waiting for 2 clients...
Connected to 127.0.0.1:1893
Waiting for 1 more client...
Connected to 127.0.0.1:1894
Monitoring 2 sockets...
Monitoring 2 sockets...
Received: This is a test message
Monitoring 2 sockets...
Received: This is another test message
Monitoring 2 sockets...
Received:
Client 127.0.0.1:1894 disconnected.
Monitoring 1 sockets...
Received: This is a final test message
Monitoring 1 sockets...
Received:
Client 127.0.0.1:1893 disconnected.
Last client disconnected, bye

C:\>
```

服务器接受两个客户机连接并且进入while循环来监视外部数据的套接字。当每个客户

机发送消息到服务器，服务器需要处理并对客户机进行回应。图8.7显示的是监视两个客户机和服务器的分析器输出。

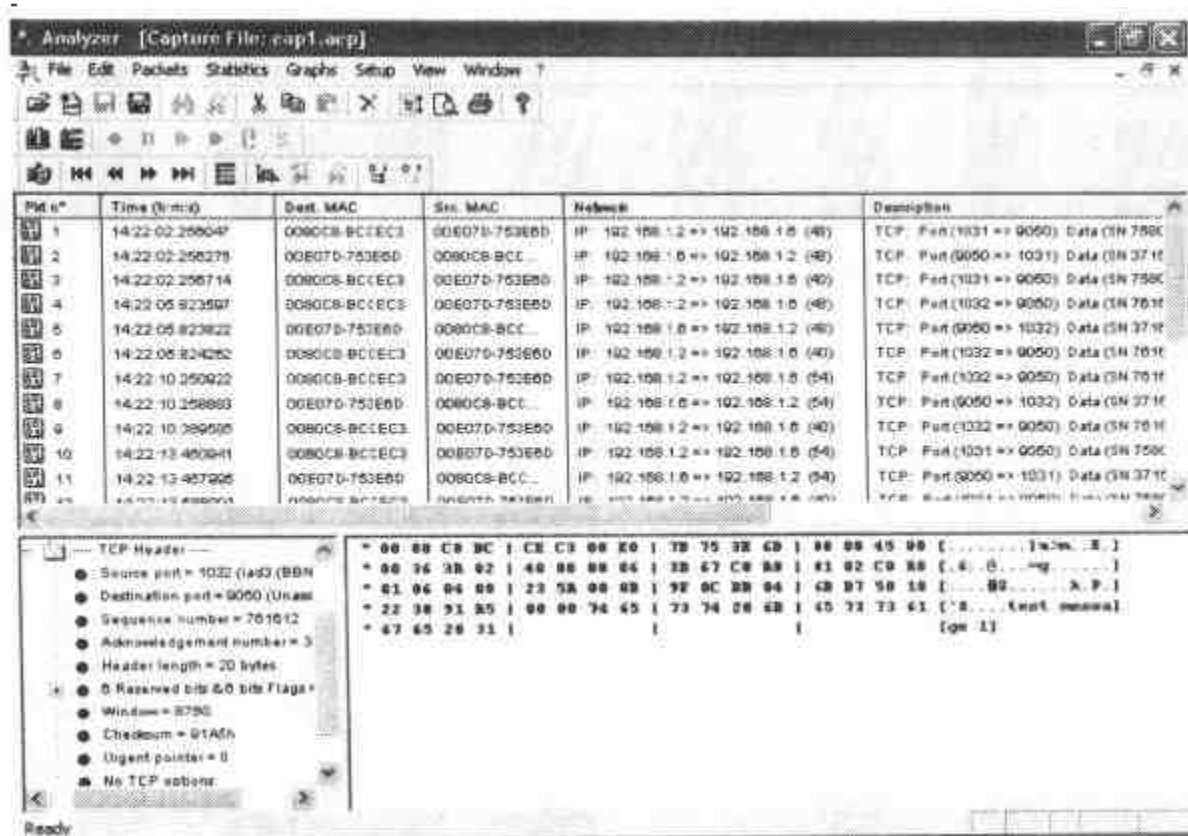


图8.7 使用分析器Analyzer程序监视SelectTcpSrvr程序

跟踪显示了两个客户机连接，其中一个使用1031号端口，另一个使用1032号端口。服务器独立地处理每个客户机TCP对话。当其中一个客户机断开与服务器的连接，其他的客户机可以继续发送消息到服务器进行处理。当所有的客户机断开连接，服务器停止处理套接字并且退出。

提示： 尽管本节中的例子使用了手动的方法创建两个客户机连接，Select方法同样可以用于单独的服务。服务器接受大量的客户机。主要的套接字轮询新连接尝试，并且传递到Accept()方法，通过这种方法，可以为每个连接分配单独的套接字，并且放置在ArrayList中进行监视。

小结

本章讨论了如何在Windows图形导向程序中使用网络功能——该程序环境不同于传统的UNIX程序开发环境，因为该环境支持事件编程模型。事件编程支持用户操作程序的控件并进行执行。程序不同于传统的程序流程。相反，当用户与程序发生交互时，程序将对发生的事件进行响应（单击按钮、双击列表条目、选择菜单条目等）。程序控件在方法间进行跳转，对用户和网络产生的事件进行响应。

.NET网络类中包括异步Socket方法，可以在Windows事件程序中进行交互。这些异步方法无法阻止程序执行来等待网络事件。相反，当异步方法准备执行时，异步方法从AsyncCallback类中指示一条方法来结束网络功能。例如，要替代Receive()方法的功能，当套接字准备使用数据时，BeginReceive()方法将注册一个AsyncCallback方法。当数据准备好后，注册的方法将被调用，并且EndReceive()方法将读取该数据。

.NET网络库同样支持传统的UNIX方法，可以使用非阻塞的套接字。**Socket**方法的**Poll()**允许编程者检测套接字，以满足网络的可用性——无论是数据的可用性，还是数据发生了错误。如果套接字没有准备好，**Poll()**方法将指示，并且程序可以避免网络功能。大多数编程者在其他功能执行的间隙使用循环的方式来检测**Poll()**方法。

其他传统的非阻塞的套接字方法属于**Socket**方法**Select()**，该功能将监视多个**Socket**对象来适应网络可用性。类似**Poll()**方法，当套接字准备进行写入时，**Select()**将检测数据的可用性和数据发生的任何错误。**Socket**对象的**ArrayList**发送**Select()**方法，监视预定的时间。如果事件在多个套接字上产生，**Select()**将返回**ArrayList**进行修改，只允许一个套接字进行网络操作。如果在任何套接字准备好之前时间已满，**ArrayList**将返回为空。

Select()方法通常用于服务器应用程序，用于监视大量的客户机套接字行为。每个行为的套接字都被放置在**ArrayList**对象中，并且使用**Select()**方法监视进入的数据。当进入的数据在一组套接字上被监测到，数据将被传送到读取方法。当客户机从服务器中断连接时，**Socket**将从**ArrayList**中删除并且不再进行监视。

异步**Socket**方法使用主应用程序的后台线程进行操作。在下一章“使用线程”中，将讲述用户如何在网络程序中使用手工线程来执行独立于主应用程序的网络功能。

第9章 使用线程

- 从C#程序中访问系统进程信息
- 线程和ProcessThread类
- 在程序中创建线程
- 在服务器中使用线程
- 使用线程发送和接收数据
- 线程池

在本章中，用户将继续学习在Windows系统中用C#进行网络编程。前面的章节讨论了在Windows程序中，使用异步套接字的方法在后台执行所有的网络套接字；程序的主要部分包括所有需要完成的套接字功能。在后台，.NET框架利用threads来执行这一任务。除了异步套接字方法，用户可以在网络程序中使用手工线程的方法。通过运行单独的线程，用户可以让服务器同时和多个客户机进行通信。

本章将定义线程，并且讲述如何在C#程序中创建和使用线程。我们将研究如何在网络程序中使用线程，并且监视一些网络程序的例子。通过检查线程池和其在网络程序中的角色，我们将对线程有更深入的了解。

如何在Windows中运行应用程序

当用户在Windows中运行程序时，操作系统软件必须知道如何维持程序，从而确保程序可以得到足够的CPU时间、内存访问和I/O设备。要做到这点，操作系统将每个程序作为自控的过程。每一个进程都代表一个单独的程序实体。如果程序的另一个副本同时运行，程序的每个实体将创建一个新的进程。用户必须使用Task Manager（任务管理器）在Windows系统中监视单独的进程。图9.1显示了运行在Windows XP Professional workstation上的不同进程（在Windows 95或者98系统上，按下Ctrl+Alt+Del键来监视运行进程的列表）。

如图9.1所示，在系统上运行的每个进程都是通过惟一的进程ID值进行识别的。任务管理器同样可以显示进程使用CPU时间、存储执行代码、数据使用内存的当前情况。如在Windows 95和98进程列表中，用户可以选中一个单独的进程并且通过单击End Process（结束进程）按钮中断该进程。

尽管Windows操作系统看起来可以同时运行多个进程，实际上，CPU同时只能运行一个进程。每个进程必须轮流使用CPU。对于每个进程来说，信息被交换到CPU区域进行执行，当进程结束后，信息再被交换出来。对于具有多个CPU的系统来说，存在两种情况：每个CPU同时运行一个进程，或者一个进程被在两个CPU间分担，如本章后面讨论线程的部分所述。



图9.1 在Windows XP Professional workstation 上的任务管理器进程窗口

为每个单独的进程分配CPU时间是由进程优先权决定的。Windows系统使用两种进程优先权：

- 由优先级类定义一个基本的进程优先权
- 在优先级类中经过优选得到特殊的进程优先权

表9.1说明优先级类可以赋予一个进程。

表9.1 进程优先级类

类	说明
Idle	只有在系统空闲时进程才运行
BelowNormal	进程比普通的优先级稍低，但是比idle优先级高
Normal	对于所有进程的默认优先级
AboveNormal	进程优先级比普通优先级高，但是低于高级优先级
High	进程具有时间有限任务，必须马上运行
RealTime	霸占所有的进程，包括系统进程

优先级类枚举在ProcessPriorityClass类中。用户在为进程定义一个新的优先级时可以进行枚举：

```
proc.PriorityClass = ProcessPriorityClass.High;
```


使用C#发现进程信息

.NET库中包含允许用户的C#程序访问系统进程信息的类。当程序运行过程中，用户可以使用这些信息检测程序的状态，或者用户可以知道同时运行在系统上的其他程序的信息。

.NET进程类

System.Diagnostics命名空间包括Process类，C#程序使用该类得到并且为系统中运行的个别进程设置信息。Process类可以启动、停止、控制和监视用户程序中的所有进程。

说明：当用户试图启动或者停止进程，要首先了解系统赋予用户程序的权利（用户所使用的ID）。普通级别的账户不允许停止其他用户启动的进程。具有管理员级别的账户可以对系统上大部分进程进行操作，有些系统进程不允许任何人停止。

Process类中最有用的元素就是类属性，存在于表9.2中。这些类属性用于得到或者设置正在运行的进程的信息。

表9.2 Process类属性

属性	说明
BasePriority	获取进程的基本优先级
Container	获取进程的Icontainer对象
EnableRaisingEvents	获取或者设置当进程中断时，是否要上升到Exited事件
ExitCode	当进程中断时，获取进程指定的值
ExitTime	获取进程中断的时间
Handle	获取进程的本机处理
HandleCount	获取进程打开的处理号码
HasExited	获取进程是否中断的验证值
Id	获取进程的分配标识符
MachineName	获取运行该进程的计算机名称
MainModule	获取进程的主模块
MainWindowHandle	获取进程主窗口的窗口操作
MainWindowTitle	获取进程主窗口的窗口标题
MaxWorkingSet	获取或者设置进程可使用的工作最大空间
MinWorkingSet	获取或者设置进程可使用的工作最小空间
Modules	获取进程加载的模块
NonpagedSytemMemorySize	得到为进程分配的非分页式系统内存

(续表)

属性	说明
PagedMemorySize	得到为进程分配的分页式内存尺寸
PagedSystemMemorySize	得到为进程分配的分页式系统内存
PeakPagedMemorySize	得到为进程分配的分页式内存尺寸的最大值
PeakVirtualMemorySize	得到为进程分配的虚拟内存的最大值
PeakWorkingSet	得到进程最大工作区尺寸
PriorityBoostEnabled	得到或者设置值，说明当主窗口聚焦时，进程的优先级是否被提高
PriorityClass	得到或者设置进程的优先级种类
PrivateMemorySize	得到进程使用的独立内存数量
PrivilegedProcessorTime	得到进程使用的特许进程器时间
ProcessName	得到进程的名称
ProcessorAffinity	得到或者设置线程运行的进程器
Responding	得到一个值，说明进程的用户界面是否响应
Site	得到或者设置ISite部件
StandardError	得到一个StreamReader对象，通过该对象读取进程的错误输出
StandardInput	得到一个StreamWriter对象，通过该对象将输入写入进程
StandardOutput	得到一个StreamReader对象，通过该对象读取进程的输出
StartInfo	得到或者设置属性，传递到进程Start()方法
StartTime	得到进程的启动时间
SynchronizingObject	得到或者设置对象，用于在进程退出时排列时间进程调用的执行
Threads	得到进程中运行线程的列表
TotalProcessorTime	得到进程使用的处理器时间总计
UserProcessorTime	得到用于该进程的用户处理器时间总计
VirtualMemorySize	得到进程使用的虚拟内存尺寸
WorkingSet	得到进程习惯用法的物理内存

如用户从表9.2中看到的，用户可以收集运行的进程的大量信息。通常情况，用户更关心进程名称、ID、处理器时间和内存的利用情况。

说明：不要对MachineName属性感到奇怪。如果进程运行在本地机器上，MachineName将设置为一个句号(.)，而不是实际的机器名称。如果进程运行在远程的机器上，机器名称将是远程主机的主机名称。

Process类的方法包含几种静态方法，用户可以获取系统上普通进程的信息，无需关联到特殊的Process实体。这里同样有很多方法，可以用于特殊的Process实体。表9.3显示了Process实体使用的Process方法。

表9.3 Process类方法

方法	说明
Close()	释放与进程相关的所有部件
CloseMainWindow()	通过向主窗口发送关闭消息关闭进程
CreateObjRef()	创建一个代理对象，用于与远程进程对象进行通信
Dispose()	释放进程使用的所有资源
Equals()	如果两个Process对象相同时进行确定
GetHashCode()	得到进程的惟一散列码
GetLifetimeService()	得到当前寿命服务器对象，控制进程的寿命原则
GetType()	得到进程的对象类型
InitializeLifetimeService()	得到一个寿命服务对象，控制进程的寿命原则
Kill()	立即停止所有的进程
Refresh()	从进程对象缓存中删除所有的信息
Start()	启动一个进程，并且关联到Process()对象
ToString()	得到进程的文字表示
WaitForExit()	设置一个时间区域，等待进程退出并且阻塞当前的线程，直到超时或者进程退出
WaitForInputIdle()	导致Process对象等待进程进入idle状态

每个Process类方法都可以用于Process实体，只需修改方法的行为或者获取方法的信息。Process静态方法用于获取系统上进程的普通信息。表9.4显示了这些方法。

表9.4 Process静态方法

方法	说明
EnterDebugMode()	允许一个Process对象在特殊模式下与操作系统进程进行交互
GetCurrentProcess()	创建一个新的Process对象，并将该对象与当前活动的进程进行关联
GetProcessById()	创建一个新的Process对象，并且通过进程ID与进程关联
GetProcesses()	创建一个Process对象数组，并且与系统上运行的进程进行关联
GetProcessByName()	创建新的Process对象，并且通过进程名称将该对象与进程标识联系起来
LeaveDebugMode()	通过在特殊模式下与操作系统进程交互操作来停止Process对象

如果用户希望找到特殊的进程, 用户可以使用`GetProcessById()`或者`GetProcessByName()`方法基于进程ID或者进程名称得到信息。如果用户不知道系统上运行的进程, 可以使用`GetProcesses()`方法得到所有的进程, 并且放入数组进行提取, 获得每个进程的信息。

当前进程的信息

要确定当前程序的进程信息, 可以使用`GetCurrentProcess()`方法创建一个`Process`对象, 并且使用一些属性。程序清单9.1显示了`GetProc.cs`程序, 可以完成这项工作。

程序清单9.1 `GetProc.cs`程序

```
using System;
using System.Diagnostics;

class GetProc
{
    public static void Main()
    {
        Process thisProc = Process.GetCurrentProcess();

        string procName = thisProc.ProcessName;
        DateTime started = thisProc.StartTime;
        int procID = thisProc.Id;
        int memory = thisProc.VirtualMemorySize;
        int priMemory = thisProc.PrivateMemorySize;
        int physMemory = thisProc.WorkingSet;
        int priority = thisProc.BasePriority;
        ProcessPriorityClass priClass = thisProc.PriorityClass;
        TimeSpan cpuTime = thisProc.TotalProcessorTime;

        Console.WriteLine("Process: {0}, ID: {1}", procName, procID);
        Console.WriteLine("    started: {0}", started.ToString());
        Console.WriteLine("    CPU time: {0}", cpuTime.ToString());
        Console.WriteLine(
            "    priority class: {0} priority: {1}", priClass, priority);
        Console.WriteLine("    virtual memory: {0}", memory);
        Console.WriteLine("    private memory: {0}", priMemory);
        Console.WriteLine("    physical memory: {0}", physMemory);

        Console.WriteLine("\n    trying to change priority...");
        thisProc.PriorityClass = ProcessPriorityClass.High;
        priClass = thisProc.PriorityClass;
        Console.WriteLine("    new priority class: {0}", priClass);
    }
}
```

GetProc程序使用GetCurrentProcess()方法获得用于程序进程的几个Process属性。用户可以通过尝试其他的属性进行试验。作者系统上的输出结果如下:

```
C:\>GetProc
Process: GetProc, ID: 313
    started: 7/17/02 8:09:54 PM
    CPU time: 00:00:00.7911376
    priority class: Normal  priority: 8
    virtual memory: 70676480
    private memory: 3981312
    physical memory: 5636096

    trying to change priority...
    new priority class: High

C:\>
```

如样例输出所示, 进程成功地改变了在系统上的优先级。程序任何的其他操作在优先级改变后都被特许访问系统上的CPU时间。

所有进程的信息

如前面提到的, 用户可以使用任务管理器来手动显示系统上运行的所有进程的进程信息。在用户的C#程序中, GetProcesses()方法可以完成同样的任务。程序清单9.2中的ListProcs.cs程序, 列出了系统上所有运行的进程, 包括每个进程的有用信息。

程序清单9.2 ListProcs.cs程序

```
using System;
using System.Diagnostics;

class ListProcs
{
    public static void Main()
    {
        int totMemory = 0;
        Console.WriteLine("Info for all processes:");

        Process[] allProcs = Process.GetProcesses();
        foreach(Process thisProc in allProcs)
        {
            string procName = thisProc.ProcessName;
            DATETIME STARTED = THISPROC.STARTTIME;
            int procID = thisProc.Id;
            int memory = thisProc.VirtualMemorySize;
            int priMemory = thisProc.PrivateMemorySize;
            int physMemory = thisProc.WorkingSet;
```

```

        totMemory += physMemory;
        int priority = thisProc.BasePriority;
        TimeSpan cpuTime = thisProc.TotalProcessorTime;

        Console.WriteLine("Process: {0}, ID: {1}", procName, procID);
        Console.WriteLine("    started: {0}", started.ToString());
        Console.WriteLine("    CPU time: {0}", cpuTime.ToString());
        Console.WriteLine("    virtual memory: {0}", memory);
        Console.WriteLine("    private memory: {0}", priMemory);
        Console.WriteLine("    physical memory: {0}", physMemory);
    }

    Console.WriteLine("\nTotal physical memory used: {0}", totMemory);
}
}

```

当GetProcesses()方法将进程信息放置在Process对象的数组中后, 用户可以使用foreach函数在数组中进行迭代, 分配每个Process数组条目到单独的Process对象, 并且提取相关信息。根据系统的不同, 输出的结果可能有点长。这里是作者系统输出结果中的片段:

```

C:\>ListProcs
Info for all processes:
Process: winword, ID: 318
    started: 7/18/02 7:38:58 PM
    CPU time: 00:00:11.3363008
    virtual memory: 59056128
    private memory: 4009984
    physical memory: 9977856
Process: drvmgr, ID: 94
    started: 7/18/02 6:48:49 PM
    CPU time: 00:00:00.0701008
    virtual memory: 25489408
    private memory: 1691648
    physical memory: 204800
Process: CMD, ID: 307
    started: 7/18/02 7:46:13 PM
    CPU time: 00:00:00.2303312
    virtual memory: 18448384
    private memory: 471040
    physical memory: 1810432
Process: spoolss, ID: 82
    started: 7/18/02 6:48:46 PM
    CPU time: 00:00:00.5307632
    virtual memory: 62898176
    private memory: 4313088

```

```
physical memory: 1732608
.
.
.
Process: SysTray, ID: 62
    started: 7/18/02 7:36:50 PM
    CPU time: 00:00:00.0701008
    virtual memory: 25239552
    private memory: 1642496
    physical memory: 1413120
Process: MsgSys, ID: 167
    started: 7/18/02 6:49:05 PM
    CPU time: 00:00:00.0901296
    virtual memory: 30765056
    private memory: 921600
    physical memory: 77824
Process: ListProcs, ID: 365
    started: 7/18/02 7:47:19 PM
    CPU time: 00:00:00.7510800
    virtual memory: 70676480
    private memory: 4001792
    physical memory: 5619712
Process: OUTLOOK, ID: 356
    started: 7/18/02 7:38:31 PM
    CPU time: 00:00:01.1516560
    virtual memory: 57663488
    private memory: 2883584
    physical memory: 5099520
Process: System, ID: 2
    started: 7/18/02 6:48:16 PM
    CPU time: 00:00:31.4151728
    virtual memory: 544768
    private memory: 36864
    physical memory: 204800
Process: Idle, ID: 0
    started: 7/18/02 6:48:16 PM
    CPU time: 00:51:37.8344672
    virtual memory: 0
    private memory: 0
    physical memory: 16384
Total physical memory used: 75292672

C:\>
```

检测例子的输出，用户将看到一些类似的进程名称——winword、outlook和ListProc。同样，注意不同的系统进程，如果在系统桌面上不可见，那么就是运行在系统后台。用户可以将CPU时间和每个进程进行比较来监视哪个应用程序使用了系统上大部分时间。

线程

进程由一个或者多个线程组成。线程是作为程序中的单独操作流程而定义。当程序在CPU上执行，将程序的语句贯穿整个线程，直到线程结束。多线程应用程序将在多个程序流中分配函数，允许两个或者更多的执行路线。每个执行路线都是单独的线程。

多线程程序创建程序自己的线程，可以独立于程序的主线程执行。所有的线程都是通过主线程创建的，和主线程共享同样的内存地址空间。通常情况下，第二线程用于执行计算机的增强功能，允许主线程继续响应Windows事件。在没有第二线程的时候，用户无法从菜单中选择任何事物，或者在计算机执行数学计算时无法单击任何按钮。

线程是在Windows操作系统上执行的最基本单元。每个进程都是通过线程的规划在CPU上运行。在多进程器系统中，同时可以运行多个线程，允许操作系统在同一个进程或者同时运行的单独进程中规划多个线程。

线程有多个操作状态，该状态使用.NET ThreadState细目枚举，存在于System.Threading命名空间中。表9.5列举了Windows操作系统中可以使用的线程状态。

表9.5 ThreadState细目

线程状态	说明
Initialized	线程已经被初始化，但是没有启动
Ready	线程等待处理器
Running	线程正在使用处理器
Standby	线程和使用处理器有关
Terminated	线程结束并且准备退出
Transition	线程等待处理器之外的资源
Unknown	系统无法确定线程状态
Wait	线程没有准备好使用处理器

线程在寿命周期中可以改变状态，在Running、Standby、Transition或者Wait状态间转换。操作系统自身可以将运行的线程放置到其他状态，从而取代线程的高级别进程。

.NET框架的ProcessThread类

.NET ProcessThread存在于System.Diagnostics命名空间中，允许用户访问C#程序进程的线程。类似Process类，ProcessThread类包含很多属性和方法，可以用于获取进程的线程信息。表9.6定义了可以使用的ProcessThread属性，以获取线程操作的信息。

表9.6 ProcessThread属性

属性	说明
BasePriority	得到线程的基本优先权
Container	得到包含线程对象的IContainer
CurrentPriority	得到线程的当前优先级
Id	得到线程的惟一线程标识
IdealProcessor	设置线程运行首选的处理器
PriorityBoostEnabled	得到或者设置一个值，指示当主窗口聚焦时，操作系统是否会临时提高线程的优先级
PriorityLevel	得到或者设置线程的优先级
PrivilegedProcessorTime	得到线程在操作系统核心运行代码所花费的时间
ProcessorAffinity	设置线程可以运行的处理器
Site	得到或者设置线程对象的ISite
StartAddress	得到启动线程的函数的内存地址
ThreadState	得到线程的当前状态
TotalProcessorTime	得到线程使用处理器的总时间
UserProcessorTime	得到应用程序中线程运行代码的总时间
WaitReason	得到线程处于Wait状态的原因

说明：Id属性创建一个惟一的号码，用于识别系统中的线程。在系统上同时运行的线程不可能拥有相同的线程Id。IdealProcessor和ProcessorAffinity属性允许用户对多进程器上的线程进行操作。

确定一个进程中的线程

编写一个快速的程序，显示当前程序中的线程。程序清单9.3显示了一个示例程序，该程序可以得到线程信息。

程序清单9.3 GetThreads.cs程序

```
using System;
using System.Diagnostics;

class GetThreads
{
    public static void Main()
    {
        Process thisProc = Process.GetCurrentProcess();
```

```
        ProcessThreadCollection myThreads = thisProc.Threads;

        foreach(ProcessThread pt in myThreads)
        {
            DateTime startTime = pt.StartTime;
            TimeSpan cpuTime = pt.TotalProcessorTime;
            int priority = pt.BasePriority;
            ThreadState ts = pt.ThreadState;

            Console.WriteLine("thread: {0}", pt.Id);
            Console.WriteLine("    started: {0}", startTime.ToString());
            Console.WriteLine("    CPU time: {0}", cpuTime);
            Console.WriteLine("    priority: {0}", priority);
            Console.WriteLine("    thread state: {0}", ts.ToString());
        }
    }
}
```

ProcessThreadCollection类包含从**Process**类的**Threads()**方法中返回的所有线程的信息。该例子在程序清单9.3中, 使用**foreach**函数在**ProcessThreadCollection**的所有线程中进行叠代, 获取每个线程的属性信息。

运行**GetThreads**程序可能会得到一个意外的结果:

```
C:\>GetThreads
thread: 263
    started: 7/18/02 11:25:45 PM
    CPU time: 00:00:00.8211808
    priority: 8
    thread state: Running
thread: 382
    started: 7/18/02 11:25:45 PM
    CPU time: 00:00:00
    priority: 8
    thread state: Wait
thread: 378
    started: 7/18/02 11:25:45 PM
    CPU time: 00:00:00
    priority: 10
    thread state: Ready

C:\>
```

如用户从该输出中看到的, **GetThreads**程序显示了该进程中有三个单独的线程在运行。在程序中, 只有一个执行路程, 所有用户可能会奇怪另外两个线程从何而来。用户可以在.NET框架中找到答案。如第1章“C#语言”中所阐述的, 程序并不是一个完全执行的程序, 需要编译到微软中间语言(MSIL)。当每个MSIL程序运行时, JIT编译器必须同时运行, 进

程MSIL代码中的每条语句——这就是每个程序有多个线程的原因。

说明：前面的例子程序是在安装了.NET框架运行时的机器上运行。在安装了SDK的机器上，单进程会使用更多的线程。

另外需要注意的是，这个输出是三个线程的状态。因为这个程序可以在单处理器系统上运行，所以同时只会有一个线程处于运行状态。最后一个线程是**Ready**准备状态，表示该线程等待处理器进行处理。在程序中的中间线程还没有准备处理。

监视所有进程中的线程

用户现在可以将**ProcList**和**GetThreads**程序放在一起，得到一个所有进程的线程列表。程序清单9.4中的**ListThreads.cs**程序，可以显示系统上所有运行线程的信息。

程序清单9.4 ListThreads.cs程序

```
using System;
using System.Diagnostics;
class ListThreads
{
    public static void Main()
    {
        Process[] allProcs = Process.GetProcesses();
        foreach(Process proc in allProcs)
        {
            ProcessThreadCollection myThreads = proc.Threads;
            Console.WriteLine("process: {0}, id: {1}", proc.ProcessName, proc.Id);
            foreach(ProcessThread pt in myThreads)
            {
                DateTime startTime = pt.StartTime;
                TimeSpan cpuTime = pt.TotalProcessorTime;
                int priority = pt.BasePriority;
                ThreadState ts = pt.ThreadState;
                Console.WriteLine("  thread: {0}", pt.Id);
                Console.WriteLine("    started: {0}", startTime.ToString());
                Console.WriteLine("    CPU time: {0}", cpuTime);
                Console.WriteLine("    priority: {0}", priority);
                Console.WriteLine("    thread state: {0}", ts.ToString());
            }
        }
    }
}
```

ListThreads程序使用**GetProcesses()**方法获取系统上运行的一组进程。然后将每个进程独立开，并使用**Threads()**方法创建一个**ProcessThreadCollection**对象。从此，在进程中的每个单

独的线程将被提取到ProcessThread对象中。在ProcessThread对象创建后，用户可以根据需要提取任何线程属性。

根据用户系统的繁忙程度，ListThreads程序的输出可能非常庞大。这里有一个短小的输出结果例子：

```
C:\>ListThreads
process: ListThreads, id: 318
  thread: 327
    started: 7/18/02 11:42:22 AM
    CPU time: 00:00:00.7711088
    priority: 8
    thread state: Running
  thread: 371
    started: 7/18/02 11:42:22 AM
    CPU time: 00:00:00
    priority: 8
    thread state: Wait
  thread: 377
    started: 7/18/02 11:42:22 AM
    CPU time: 00:00:00
    priority: 10
    thread state: Ready
process: SysTray, id: 62
  thread: 330
    started: 7/18/02 7:36:50 AM
    CPU time: 00:00:00.1001440
    priority: 8
    thread state: Wait
  thread: 351
    started: 7/18/02 7:36:51 AM
    CPU time: 00:00:00.0200288
    priority: 10
    thread state: Wait
process: winword, id: 374
  thread: 264
    started: 7/18/02 10:51:48 AM
    CPU time: 00:01:40.3943600
    priority: 8
    thread state: Wait
  thread: 381
    started: 7/18/02 10:51:49 AM
    CPU time: 00:00:00
    priority: 8
    thread state: Wait
```

```
thread: 155
  started: 7/18/02 10:53:24 AM
  CPU time: 00:00:00.0100144
  priority: 10
  thread state: Wait
thread: 376
  started: 7/18/02 10:57:50 AM
  CPU time: 00:00:00
  priority: 8
  thread state: Wait
process: Explorer, id: 350
  thread: 277
    started: 7/18/02 7:36:46 AM
    CPU time: 00:00:05.1173584
    priority: 9
    thread state: Wait
  thread: 363
    started: 7/18/02 7:36:47 AM
    CPU time: 00:00:00.1301872
    priority: 8
    thread state: Wait
  thread: 317
    started: 7/18/02 7:36:54 AM
    CPU time: 00:00:00.0100144
    priority: 8
    thread state: Wait
  thread: 294
    started: 7/18/02 7:42:55 AM
    CPU time: 00:00:00
    priority: 8
    thread state: Wait
  thread: 296
    started: 7/18/02 9:27:43 AM
    CPU time: 00:00:03.4048960
    priority: 8
process: notepad, id: 337
  thread: 378
    started: 7/18/02 11:38:40 AM
    CPU time: 00:00:00.0901296
    priority: 8
    thread state: Wait
process: Idle, id: 0
C:\>
```

通过ListThreads程序的输出结果，用户可以学到一些内容。如果用户使用单处理器系统，列表中只有一个线程处于运行状态——ListThreads程序的主线程。如果用户拥有一个多处理器系统，用户会发现线程的数量和处理器的数量相同。

注意每个进程使用线程的数量。简单程序，如笔记本只使用一个线程，而更复杂的程序，例如Winword和资源管理器使用多个线程。明显的复杂程序，例如Winword需要进行大量的后台工作，创建新的线程进行工作，从而保证主用户界面不受影响。

在程序中创建线程

现在，用户已经了解程序可以包含多个线程，现在我们讲述如何利用多个线程创建C#程序。C#语言提供System.Threading命名空间，该命名空间包含用于创建和控制线程的类。这节内容将说明如何在C#程序中使用这些类。

Thread（线程）类

使用Thread类创建一个新的Thread对象，该类在当前进程中产生一个新的线程。Thread构造器的格式如下，在构造器中启动的是ThreadStart代表：

```
Thread(ThreadStart start)
```

ThreadStart代表指向在新线程中执行的方法。

这里是一个创建新线程的例子的缩写：

```
.  
.   
Thread newThread = new Thread(new ThreadStart(newMethod));  
.   
.   
}  
void newMethod()  
{  
.   
.   
}
```

newMethod参数定义了一个方法，当处理器到达接收时间时，该方法在线程中执行。在这个例子中，方法作为Thread对象包含在同一个类中。用户同样可以运行定义在其他类中的方法。

在创建Thread对象后，可以使用不同的Thread类方法对该对象进行控制，该方法说明在表9.7中。

在创建Thread对象后，必须使用Start()方法继续。

警告：要记住，除非使用Start()方法，否则，Thread对象将无法开始处理。

表9.7 线程类方法

方法	说明
Abort()	中断线程
Equals()	确定两个线程是否相同
GetHashCode()	得到线程的惟一表示
GetType()	得到当前线程的类型
Interrupt()	在等待线程状态中中断线程
Join()	在线程中断前阻塞调用线程
Resume()	恢复挂起的线程
Start()	导致操作系统改变线程的状态为运行
Suspend()	将执行的线程挂起
ToString()	得到Thread对象的字符串表示

在线程启动后，可以使用Abort()、Interrupt()、Resume()或者Suspend()方法。

使用线程类

学习线程的最佳方法就是监视线程的行为。程序清单9.5提供一个示例程序，说明线程的作用。

程序清单9.5 ThreadSample.cs程序

```
using System;
using System.Threading;

class ThreadSample
{
    public static void Main()
    {
        ThreadSample ts = new ThreadSample();
    }

    public ThreadSample()
    {
        int i;
        Thread newCounter = new Thread(new ThreadStart(Counter));
        Thread newCounter2 = new Thread(new ThreadStart(Counter2));

        newCounter.Start();
        newCounter2.Start();
    }
}
```

```
        for(i = 0; i < 10; i++)
        {
            Console.WriteLine("main: {0}", i);
            Thread.Sleep(1000);
        }
    }

    void Counter()
    {
        int i;
        for (i = 0; i < 10; i++)
        {
            Console.WriteLine("  thread: {0}", i);
            Thread.Sleep(2000);
        }
    }

    void Counter2()
    {
        int i;
        for (i = 0; i < 10; i++)
        {
            Console.WriteLine("    thread2: {0}", i);
            THREAD.SLEEP(3000);
        }
    }
}
```

ThreadSample.cs程序使用**Thread**构造器（**ThreadStart**构造器），创建独立于主程序线程的两个单独的线程。主程序执行一个**for**循环，每秒显示一个计数器值，并且使用**Thread**类的**Sleep()**静态方法。第二个线程完成同样的工作，但是使用2秒的延时；第三个线程使用3秒的延时。再一次记住，只有在使用**Start()**方法时线程才会开始运行，而不是在构造器结束时开始。

用户可以通过运行**ThreadSample**程序来监视输出：

```
C:\>ThreadSample
main: 0
  thread: 0
    thread2: 0
main: 1
  thread: 1
main: 2
    thread2: 1
main: 3
```



```
thread: 2
main: 4
main: 5
thread2: 2
thread: 3
main: 6
main: 7
thread: 4
main: 8
thread2: 3
main: 9
thread: 5
thread2: 4
thread: 6
thread: 7
thread2: 5
thread: 8
thread2: 6
thread: 9
thread2: 7
thread2: 8
thread2: 9
```

C:\>

在程序输出中，有一些非常有趣的事情。首先注意，直到线程结束处理，程序都没有退出，即使主线程已经结束了任务。这是线程非常重要的功能。同样，用户可以看到，在程序开始时，线程输出的间隔可能和用户期望的不一样。这是因为设置和启动每个新的线程有额外的时间需求。线程一旦开始运行，输出的时间间隔将保持一致。

所有这一切都可以说明线程的一个重要属性：创建新线程是一项非常占用CPU的操作。

在本章的后面（在“线程池”部分），用户可以研究一些减少用户程序创建线程时影响的小技巧，例如创建线程池。

警告：如果编写程序时使用多个线程，要非常小心线程的内部操作。多个线程可能产生问题，例如死锁（当两个线程停止响应，等待彼此结束）和静态情况（在两个或者多个线程之间，在非预期的间隙产生错误的结果）。优秀的程序必须要考虑这些因素。

在服务器中使用线程

现在用户已经掌握了如何使用线程，让我们看看如何在网络程序中发挥线程的功能。网络服务器设计的最大问题就是容纳多个客户机。回到第8章“异步套接字编程”，介绍了Select()方法，本书中的所有例子都只允许同时有一个客户机访问服务器。

使用Select()方法处理多个客户机的问题在于，代码会变得令人费解。试图容纳多个客

户机在不同的时间间隔中发送和接收数据是非常复杂的编程。同样，当使用Select()时，同时只能服务于一个客户机，有效的阻塞访问可以用于其他的客户机。最好的解决方法在于，将每个客户机分配到单独的世界，单独地进行数据处理。线程就可以帮助用户完成该项工作。

创建线程服务器

创建线程服务器的关键在于，要规划出对每个客户机的单独处理，并且推算到成百（或者成千）的客户机。图9.2显示了一个线程服务器程序的操作示例。

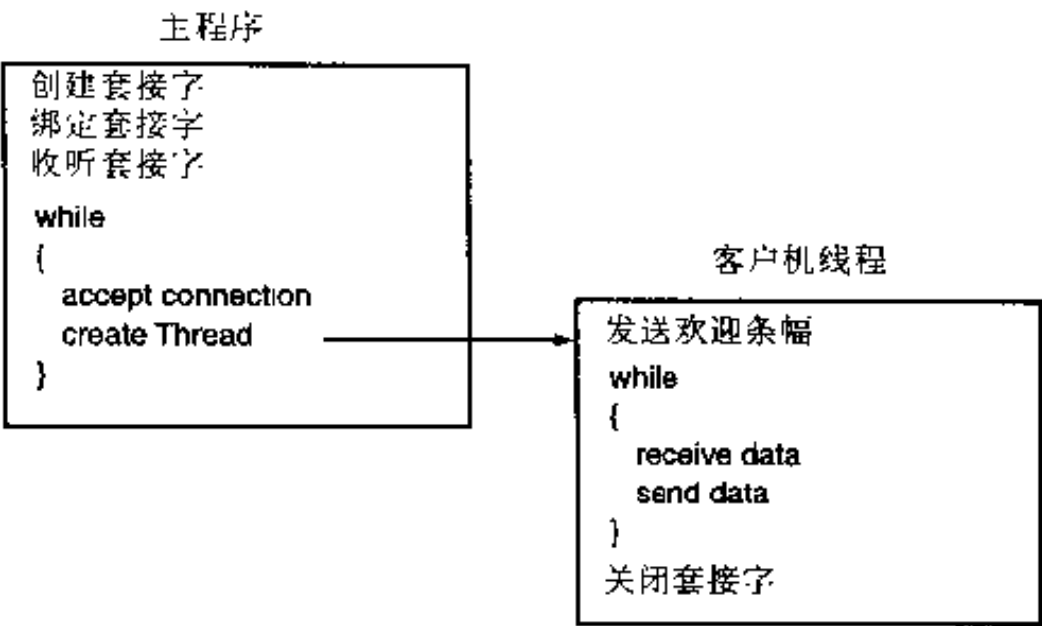


图9.2 线程服务器的操作

对于线程服务器来说，关键在于在主程序中创建一个主服务器Socket对象。当每个客户机连接到服务器，主程序创建一个单独的Thread对象来处理连接。因为Socket Accept()方法创建了一个新的Socket对象进行连接，新对象将传递到代表方法，并且用于新线程的所有客户机通信。原始的Socket对象将被释放，接受更多的客户机连接。

当用户确定新服务器程序将如何创建线程处理新连接后，可以非常容易地使用C#规划模型。程序清单9.6是一个线程TCP服务器的示例，可以用于与多个客户机的通信。

程序清单9.6 ThreadedTcpSrvr.cs程序

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;
using System.Threading;

class ThreadedTcpSrvr
{
    private TcpListener client;

    public ThreadedTcpSrvr()
    {
        client = new TcpListener(9050);
```



```
        client.Start();

        Console.WriteLine("Waiting for clients...");
        while(true)
        {
            while (!client.Pending())
            {
                Thread.Sleep(1000);
            }

            ConnectionThread newconnection = new ConnectionThread();
            newconnection.threadListener = this.client;
            Thread newthread = new Thread(new
                ThreadStart(newconnection.HandleConnection));
            newthread.Start();
        }
    }

    public static void Main()
    {
        ThreadedTcpSrvr server = new ThreadedTcpSrvr();
    }
}

class ConnectionThread
{
    public TcpListener threadListener;
    private static int connections = 0;

    public void HandleConnection()
    {
        int recv;
        byte[] data = new byte[1024];

        TcpClient client = threadListener.AcceptTcpClient();
        NetworkStream ns = client.GetStream();
        connections++;
        Console.WriteLine("New client accepted: {0} active connections",
            connections);

        string welcome = "Welcome to my test server";
        data = Encoding.ASCII.GetBytes(welcome);
        ns.Write(data, 0, data.Length);

        while(true)
        {
            data = new byte[1024];
            recv = ns.Read(data, 0, data.Length);
            if (recv == 0)
```

```
        break;

        ns.Write(data, 0, recv);
    }
    ns.Close();
    client.Close();
    connections--;
    Console.WriteLine("Client disconnected: {0} active connections",
        connections);
}
}
```

ThreadedTcpSrvr.cs启动后非常类似本书中其他的TCP服务器程序。该程序使用**TcpListener**类在标准端口（9050）来收听外部的TCP连接，并且使用**Start()**方法等待新的客户机。这个例子的不同之处在于如何接受新的客户机：

```
while(true)
{
    while (!client.Pending())
    {
        Thread.Sleep(1000);
    }

    ConnectionThread newconnection = new ConnectionThread();
    newconnection.threadListener = this.client;
    Thread newthread = new Thread(new
        ThreadStart(newconnection.HandleConnection));
    newthread.Start();
}
```

服务器主程序进入了无尽的循环，每秒进行一次检测，监视是否有客户机的连接。如果有客户机连接，主程序将创建一个新的线程来接受连接。一个单独的类用于处理与客户机的通信。**TcpListener**对象将设置为客户机连接的原始**TcpListener**对象。

这个技术将创建很多线程，用于客户机连接到服务器（在极端的情况下，当然这并不是一个好方法。根据服务器CPU的能力，用户可能需要限制客户机的数量）。

测试服务器

在启动**ThreadedTcpSrvr**程序后，用户可以根据需要启动多个客户机程序连接到服务器。这个最简单的方法可以通过第8章中的**AsyncTcpClient**程序进行说明，因为用户可以在单独的命令提示符窗口中运行多个客户机。

当服务器接受每个客户机的连接，服务器将创建一个单独的套接字，并且将套接字传递到新的线程。每个线程都是自控的，可以执行标准的服务器功能，无需将连接传递到主程序。

每个客户机程序可以进行完全独立的操作。用户可以从客户机发送消息，接收正确的消息回应。同样，每个客户机也可以在不影响其他客户机的情况下中断连接。

监视线程

当服务器运行时，用户可以通过使用**ListThreads**程序来监视为每个客户机创建的线程。如果用户将**ListThreads**程序的输出定向到一个文件，用户可以使用记事本程序来搜索**ThreadedTcpSrvr**进程，并且监视线程的行为：

```
process: ThreadedTcpSrvr, id: 944
  thread: 1480
    started: 7/21/2002 9:22:54 PM
    CPU time: 00:00:00.3404896
    priority: 8
    thread state: Wait
  thread: 3648
    started: 7/21/2002 9:22:55 PM
    CPU time: 00:00:00
    priority: 8
    thread state: Wait
  thread: 1012
    started: 7/21/2002 9:22:55 PM
    CPU time: 00:00:00
    priority: 10
    thread state: Wait
  thread: 1612
    started: 7/21/2002 9:25:29 PM
    CPU time: 00:00:00
    priority: 8
    thread state: Wait
  thread: 2340
    started: 7/21/2002 9:25:32 PM
    CPU time: 00:00:00
    priority: 8
    thread state: Wait
```

当没有任何客户机连接到**ThreadedTcpSrvr**程序，该程序将使用三个线程。当有两个客户机连接，程序使用五个线程。如预期的那样，附加的线程在**ThreadedTcpSrvr**进程中创建，作为每个客户机的连接。当每个客户机断开，线程将被中断。

用线程发送和接收数据

为了支持多个客户机，在没有明确协议模型的情况下，可以在网络程序中使用线程。在本书的示例程序中，采用了一致的模型定义，这样客户机和服务器可以正确地理解

接收和发送数据。对于网络程序来说，并不都是这样。一个例子就是通用的聊天程序，该程序允许两个人通过消息白板系统进行通信，在系统中输入的文本将被显示到远程系统上。问题在于，在连接过程中，每个人都可以决定发送消息——这将对网络协议模型造成巨大的破坏。

幸运的是，可以使用线程解决问题。第二线程可以创建与主程序独立的，用于阻塞在 `Receive()` 的方法，所以任何时间的任何外部发送数据都可以显示在消息白板上。这一概念显示在图9.3中。聊天的参与者可以选择作为对话服务器，放置在收听方式，等待接收新的对话客户机。当客户机连接，服务器创建一个单独的线程用于等待外部的数据。

同样的，当聊天客户机连接到服务器，同样会创建一个单独的线程来等待从服务器发送的外部数据。因为第二个线程永远处于收听方式，第一线程只需在用户发送消息到远程主机时进行操作。

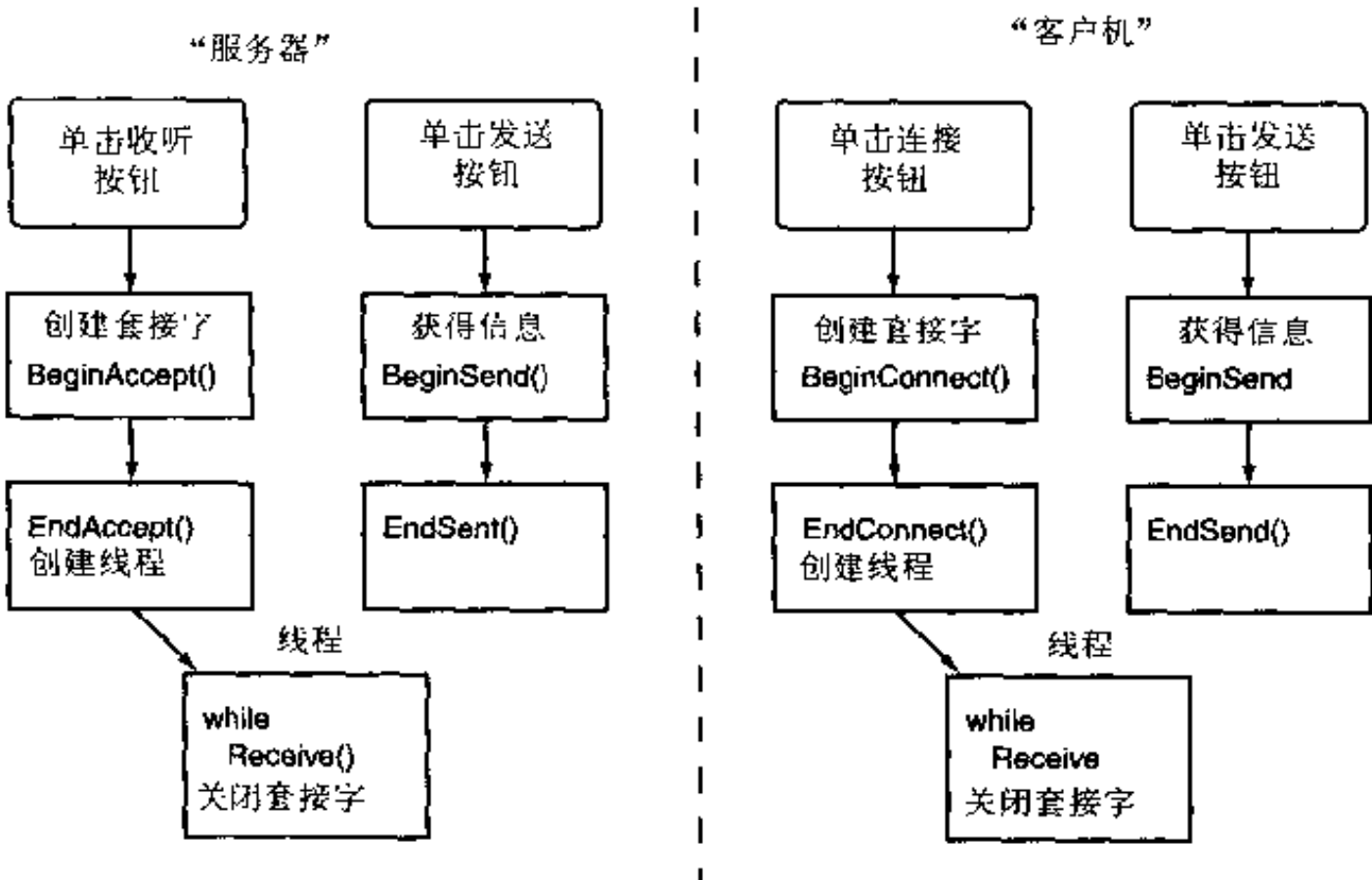


图9.3 在网络聊天程序中使用线程

TcpChat程序

现在将应用程序进行图形化，用户将发现非常容易理解和实践程序清单9.7中的C#程序，`TcpChat.cs`程序示例了一个简单的网络聊天程序。

程序清单9.7 `TcpChat.cs`程序

```
using System;
using System.Drawing;
using System.Net;
using System.Net.Sockets;
using System.Text;
```

```
using System.Threading;
using System.Windows.Forms;

class TcpChat Form
{
    private static TextBox newText;
    private static ListBox results;
    private static Socket client;
    private static byte[] data = new byte[1024];

    public TcpChat()
    {
        Text = "TCP Chat Program";
        Size = new Size(400, 380);

        Label label1 = new Label();
        label1.Parent = this;
        label1.Text = "Enter text string:";
        label1.AutoSize = true;
        label1.Location = new Point(10, 30);

        newText = new TextBox();
        newText.Parent = this;
        newText.Size = new Size(200, 2 * Font.Height);
        newText.Location = new Point(10, 55);

        results = new ListBox();
        results.Parent = this;
        results.Location = new Point(10, 85);
        results.Size = new Size(360, 18 * Font.Height);

        Button sendit = new Button();
        sendit.Parent = this;
        sendit.Text = "Send";
        sendit.Location = new Point(220, 52);
        sendit.Size = new Size(5 * Font.Height, 2 * Font.Height);
        sendit.Click += new EventHandler(ButtonSendOnClick);

        Button connect = new Button();
        connect.Parent = this;
        connect.Text = "Connect";
        connect.Location = new Point(295, 20);
        connect.Size = new Size(6 * Font.Height, 2 * Font.Height);
        connect.Click += new EventHandler(ButtonConnectOnClick);

        Button listen = new Button();
        listen.Parent = this;
        listen.Text = "Listen";
        listen.Location = new Point(295, 52);
```

```
listen.Size = new Size(6 * Font.Height, 2 * Font.Height);
listen.Click += new EventHandler(ButtonListenOnClick);
}

void ButtonListenOnClick(object obj, EventArgs ea)
{
    results.Items.Add("Listening for a client...");
    Socket newsock = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
        ProtocolType.Tcp);
    IPEndPoint iep = new IPEndPoint(IPAddress.Any, 9050);
    newsock.Bind(iep);
    newsock.Listen(5);
    newsock.BeginAccept(new AsyncCallback(AcceptConn), newsock);
}

void ButtonConnectOnClick(object obj, EventArgs ea)
{
    results.Items.Add("Connecting...");
    client = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
        ProtocolType.Tcp);
    IPEndPoint iep = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 9050);
    client.BeginConnect(iep, new AsyncCallback(Connected), client);
}

void ButtonSendOnClick(object obj, EventArgs ea)
{
    byte[] message = Encoding.ASCII.GetBytes(newText.Text);
    newText.Clear();
    client.BeginSend(message, 0, message.Length, 0,
        new AsyncCallback(SendData), client);
}

void AcceptConn(IAsyncResult iar)
{
    Socket oldserver = (Socket)iar.AsyncState;
    client = oldserver.EndAccept(iar);
    results.Items.Add("Connection from: " + client.RemoteEndPoint.ToString());
    Thread receiver = new Thread(new ThreadStart(ReceiveData));
    receiver.Start();
}

void Connected(IAsyncResult iar)
{
    try
    {
        client.EndConnect(iar);
        results.Items.Add("Connected to: " + client.RemoteEndPoint.ToString());
    }
}
```



```

        Thread receiver = new Thread(new ThreadStart(ReceiveData));
        receiver.Start();
    } catch (SocketException)
    {
        results.Items.Add("Error connecting");
    }
}

void SendData(IAsyncResult iar)
{
    Socket remote = (Socket)iar.AsyncState;
    INT SENT = REMOTE.ENDSEND(IAR);
}

void ReceiveData()
{
    int recv;
    string stringData;
    while (true)
    {
        recv = client.Receive(data);
        stringData = Encoding.ASCII.GetString(data, 0, recv);
        if (stringData == "bye")
            break;
        results.Items.Add(stringData);
    }
    stringData = "bye";
    byte[] message = Encoding.ASCII.GetBytes(stringData);
    client.Send(message);
    client.Close();
    results.Items.Add("Connection stopped");
    return;
}

public static void Main()
{
    Application.Run(new TcpChat());
}
}

```

因为TcpChat程序必须作为客户机和服务器使用，因此该程序具有多功能性。主程序创建Windows窗体，该窗体包括下列对象：

- 一个收听按钮，用于设置程序到收听方式
- 一个连接按钮，用于连接到远程的聊天伙伴
- 一个发送按钮，用于将输入TextBox的内容发送到远程聊天伙伴

- 一个**ListBox**对象，用于显示信息状态和从远程聊天伙伴获得的消息

如果客户单击**Listen**按钮，将使用**BeginAccept()**方法创建一个套接字，并且收听外部的连接：

```
void ButtonListenOnClick(object obj, EventArgs ea)
{
    results.Items.Add("Listening for a client...");
    Socket newsock = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
        ProtocolType.Tcp);
    IPEndPoint iep = new IPEndPoint(IPAddress.Any, 9050);
    newsock.Bind(iep);
    newsock.Listen(5);
    newsock.BeginAccept(new AsyncCallback(AcceptConn), newsock);
}
```

当其他的聊天程序连接，**AcceptConn()**方法为客户机创建一个单独的套接字，传递到第二个线程。

如果客户单击**Connect**按钮，将使用**BeginConnect()**方法创建一个套接字并且试图连接到远程主机。当远程主机接收连接，套接字将传递到第二个线程。第二个线程的惟一目的就是等待外部的数据、并且将数据传送到列表框：

```
void ReceiveData()
{
    int recv;
    string stringData;
    while (true)
    {
        recv = client.Receive(data);
        stringData = Encoding.ASCII.GetString(data, 0, recv);
        if (stringData == "bye")
            break;
        results.Items.Add(stringData);
    }
    stringData = "bye";
    byte[] message = Encoding.ASCII.GetBytes(stringData);
    client.Send(message);
    client.Close();
    results.Items.Add("Connection stopped");
    return;
}
```

如果接收的数据中包含“bye”字符串，程序会使用同样的字符串进行回应，从而关闭套接字。两个聊天程序都会认识到聊天已经结束，正确地关闭TCP连接。

因为第二个线程是和主程序线程相互独立的，因此可以使用标准的阻塞网络方法，例

如Receive(); 第二个线程将不会妨碍Windows窗体的操作。

同时, 对于主程序来说, 当客户单击Send发送按钮, 在文本框中输入的文本将发送到远程的聊天主机。

测试聊天程序并监视线程

要测试程序, 用户必须将两个程序的副本同时运行。如果用户选择通过网络进行测试, 需要在每个程序的IPEndPoint对象中输入正确的远程主机地址。

通过单击Listen按钮, 将其中一个TcpChat副本设置在收听方式。另外一个副本通过单击Connect按钮连接到远程主机。在建立连接后, 消息可以在两个主机间进行传递。要中断聊天, 必须向远程的聊天伙伴说“再见”。图9.4示例了一个TcpChat程序在与远程副本进行通信的情况。



图9.4 TcpChat程序在运行

如用户对ThreadedTcpSrvr程序进行操作, 可以使用ListThreads程序监视第二个线程如何在连接过程中创建。每个TcpChat程序将创建一个单独的线程, 用于接收数据。在远程TcpChat主机断开连接后, 第二个线程将被中断, 并且不再作为激活的线程进行列表。主程序将保持运行, 当然, 该程序必须手动中断。

线程池

如用户在“使用Thread类”一节中观察到的, 当程序创建新的线程时, 会需要大量的内部开销。 .NET库提供一种方法, 可以避免一些开销。 Windows操作系统允许用户维持一池“预先建立的”线程, 这个线程池为应用程序中指定的方法提供工作线程。一个线程控制线程池的操作, 并且应用程序可以分配附加的线程进行处理。在默认情况下, 在线程池中有25个预处理的线程, 所以这种方法无法满足大规模应用程序的需要。

说明: 用户可以通过.NET ThreadPool类访问线程池。

ThreadPool类

System.Threading命名空间包含ThreadPool类，该类将代表分配给线程池中的线程。在ThreadPool类中的所有方法都是静态的，所以ThreadPool对象的实体都没有使用构造器。相反，一旦调用ThreadPool方法，操作系统会自动创建一个线程池，并且使用静态的ThreadPool方法对线程池中的线程进行操作。

表9.8是ThreadPool类中定义的静态方法。

表9.8 ThreadPool静态方法

方法	说明
BindHandle()	绑定操作系统的操作到线程池
GetAvailableThreads()	得到在线程池中可使用的线程数量
GetMaxThreads()	得到在线程池中可以使用的线程的最大数量
QueueUserWorkItem()	在线程池中查询用户代表
RegisterWaitForSingleObject()	注册一个代表，等待WaitHandle对象
UnsafeQueueUserWorkItem()	在线程池中查询一个不安全的用户代表，但是不会传播调用堆栈到工作线程
UnsafeRegisterWaitForSingleObject()	注册一个不安全的代表，等待WaitHandle对象

要为线程池中的线程注册一个代表，使用下面的格式：

```
ThreadPool.QueueUserWorkItem(new WaitCallback(Counter));
```

Counter参数代表运行在线程中的方法。

与Thread对象不同，代表一旦放置在线程池查询中，将被处理；不需要其他的方法启动该项工作。

警告：当主程序线程退出，所有的线程池线程都将终止。主线程不会等待线程池线程结束。

ThreadPool程序例子

学习线程池的最好方法就是监视线程池的行为。在程序清单9.8中的ThreadPoolSample.cs程序显示了一个例子，该例子在线程池中创建了两个线程，用于处理主程序线程之外的方法。

程序清单9.8 ThreadPoolSample.cs程序

```
using System;
using System.Threading;

class ThreadPoolSample
```

```
{

    public static void Main()
    {
        ThreadPoolSample tps = new ThreadPoolSample();
    }

    public ThreadPoolSample()
    {
        int i;

        ThreadPool.QueueUserWorkItem(new WaitCallback(Counter));
        ThreadPool.QueueUserWorkItem(new WaitCallback(Counter2));

        for(i = 0; i < 10; i++)
        {
            Console.WriteLine("main: {0}", i);
            Thread.Sleep(1000);
        }
    }

    void Counter(object state)
    {
        int i;
        for (i = 0; i < 10; i++)
        {
            Console.WriteLine("  thread: {0}", i);
            Thread.Sleep(2000);
        }
    }

    void Counter2(object state)
    {
        int i;
        for (i = 0; i < 10; i++)
        {
            Console.WriteLine("    thread2: {0}", i);
            Thread.Sleep(3000);
        }
    }
}
```

这个例子并不是非常明显——主程序在线程池查询中放置了两个方法调用，并且继续执行自身的功能，计数从0到9，在每个计数之间有1秒的延时。

类似ThreadSample程序，每个单独的方法都执行同样的任务，但是使用不同延时间隔。

测试程序并且监视线程

当用户运行ThreadPoolSample程序，要监视不同方法工作的输出。注意在程序的主线程停止循环后，程序如何停止。如我们提到的，应用程序将不会等待线程池线程的结束。当主应用程序线程结束后，主程序将放弃所有运行的线程池线程，如这个例子所显示的：

```
C:\>ThreadPoolSample
main: 0
    thread: 0
        thread2: 0
main: 1
    thread: 1
main: 2
main: 3
    thread2: 1
    thread: 2
main: 4
main: 5
    thread: 3
main: 6
    thread2: 2
main: 7
    thread: 4
main: 8
main: 9
    thread2: 3
    thread: 5

C:\>
```

使用前面创建的ListThreads程序来监视ThreadPoolSample产生的线程。在命令提示符窗口启动ThreadPoolSample，然后在另外一个命令提示符窗口启动ListThreads。为了便于监视ThreadPoolSample处理的线程，用户可以将ListThreads程序的输出结果定向到一个文件，并在Notepad（记事本）中检测该文件：

```
C:\>ListThreads > threads.txt
C:\>notepad threads.txt
```

用户将监视到ThreadPoolSample程序使用的几个线程：

```
process: ThreadPoolSample, id: 2676
thread: 2632
    started: 7/23/2002 11:57:31 PM
    CPU time: 00:00:00.1502160
    priority: 8
    thread state: Wait
```

```
lthread: 2620
  started: 7/23/2002 11:57:31 PM
  CPU time: 00:00:00
  priority: 8
  thread state: Wait
thread: 2508
  started: 7/23/2002 11:57:31 PM
  CPU time: 00:00:00
  priority: 10
  thread state: Wait
thread: 3032
  started: 7/23/2002 11:57:31 PM
  CPU time: 00:00:00.0200288
  priority: 8
  thread state: Wait
thread: 3036
  started: 7/23/2002 11:57:31 PM
  CPU time: 00:00:00
  priority: 8
  thread state: Wait
thread: 2888
  started: 7/23/2002 11:57:32 PM
  CPU time: 00:00:00
  priority: 8
  thread state: Wait
thread: 3040
  started: 7/23/2002 11:57:32 PM
  CPU time: 00:00:00
  priority: 8
  thread state: Wait
```

说明：用户可能会感到奇怪，为什么只创建了两个额外的线程，却有很多线程出现在列表中。答案是，当两个或者更多的线程在使用时，线程池必须创建自己的线程。这些“内部操作”线程控制线程池运行的线程。如果只创建了一个线程池线程的话，用户将不会看到这些控制线程。

在服务器中使用线程池

对于服务器来说，如果只是服务几个有限的客户机（少于线程池的最大线程数25），ThreadPool类可以在无需创建Thread对象的情况下使用单独的线程提供快速的服务。每个客户机将被分配到用户工作条目，并且通过线程池中的线程进行处理。

说明：要牢记，当每个ThreadPool对象处于激活状态时，主程序必须保持激活，否则连接将被中断。一个主程序中断，线程池中的每个线程同样会中断，可能会导致激活的连接中断，并且导致客户机失控——当然，用户可以对客户机进行编程，以适应这样的情况，如紧急状态处理。

ThreadPool服务器

在本书中，ThreadPoolTcpSrvr.cs程序使用ThreadPool对象来说明标准的TCP服务器模型。当每个客户机连接到服务器，单独的ThreadPool用户工作条目将对套接字进行处理。程序清单9.9显示的是ThreadPoolTcpSrvr.cs程序。

程序清单9.9 ThreadPoolTcpSrvr.cs程序

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;
using System.Threading;

class ThreadPoolTcpSrvr
{
    private TcpListener client;

    public ThreadPoolTcpSrvr()
    {
        client = new TcpListener(9050);
        client.Start();

        Console.WriteLine("Waiting for clients...");
        while(true)
        {
            while (!client.Pending())
            {
                Thread.Sleep(1000);
            }
            ConnectionThread newconnection = new ConnectionThread();
            newconnection.threadListener = this.client;
            ThreadPool.QueueUserWorkItem(new
                WaitCallback(newconnection.HandleConnection));
        }
    }

    public static void Main()
    {
        ThreadPoolTcpSrvr tpts = new ThreadPoolTcpSrvr();
    }
}

class ConnectionThread
{
    public TcpListener threadListener;
    private static int connections = 0;
```



```
public void HandleConnection(object state)
{
    int recv;
    byte[] data = new byte[1024];

    TcpClient client = threadListener.AcceptTcpClient();
    NetworkStream ns = client.GetStream();
    connections++;
    Console.WriteLine("New client accepted: {0} active connections",
                      connections);

    string welcome = "Welcome to my test server";
    data = Encoding.ASCII.GetBytes(welcome);
    ns.Write(data, 0, data.Length);

    while(true)
    {
        data = new byte[1024];
        recv = ns.Read(data, 0, data.Length);
        if (recv == 0)
            break;

        ns.Write(data, 0, recv);
    }
    ns.Close();
    client.Close();
    connections--;
    Console.WriteLine("Client disconnected: {0} active connections",
                      connections);
}
```

类似本书中其他的TCP服务器程序，ThreadPoolTcpSrvr程序创建一个套接字来收听新的连接。在这种情况下，ThreadPoolTcpSrvr类创建套接字并且接收外部的连接。当检测到连接后，新的ConnectionThread类实体将被创建，并且TcpListener对象将被复制，用于继续通信。HandleConnection()方法将传递到QueueUserWorkItem()方法，用于为新连接提供新的ThreadPool线程。

当连接中断，套接字将被关闭，并且线程池线程将被中断。因为主程序处于不停的循环状态，等待新的客户，所以除非程序被手动放弃，否则不会造成激活连接中断的危险。

测试程序并且监视线程

ThreadPoolTcpSrvr程序可以使用本书中的任何TCP客户机进行测试。首先，在命令提示符下启动ThreadPoolTcpSrvr程序，然后启动多个TCP客户机（例如第8章中的AsyncTcpClient程序）。

每个客户机创建一个新线程，该线程将传递到线程池中新用户的工作条目。用户工作条目然后被分配到新的线程并且进行正常的处理。每个客户机都可以在不影响其他客户机对话的情况下断开。因为服务器处于不停的循环中，等待新的客户机，用户必须通过按Ctrl+C来手动停止该循环（如果在每个连接都处于激活状态时进行该项操作，连接将被放弃并且客户机将收到一个Exception）。

用户可以使用ListThreads程序来监视ThreadPoolTcpSrvr程序使用线程的情况。首先，可以得到已被接受连接使用的线程列表：

```
process: ThreadPoolTcpSrvr, id: 1876
  thread: 980
    started: 7/23/2002 11:23:10 PM
    CPU time: 00:00:00.4306192
    priority: 8
    thread state: Wait
  thread: 1544
    started: 7/23/2002 11:23:11 PM
    CPU time: 00:00:00
    priority: 8
    thread state: Wait
  thread: 536
    started: 7/23/2002 11:23:11 PM
    CPU time: 00:00:00
    priority: 10
    thread state: Wait
```

这说明程序使用了三个标准的线程。接下来，使用TCP客户机程序连接到服务器，然后将得到线程使用情况的其他说明：

```
process: ThreadPoolTcpSrvr, id: 1876
  thread: 980
    started: 7/23/2002 11:23:09 PM
    CPU time: 00:00:00.4806912
    priority: 8
    thread state: Wait
  thread: 1544
    started: 7/23/2002 11:23:10 PM
    CPU time: 00:00:00
    priority: 8
    thread state: Wait
  thread: 536
    started: 7/23/2002 11:23:10 PM
    CPU time: 00:00:00
    priority: 10
    thread state: Wait
```



```
thread: 1184
  started: 7/23/2002 11:25:53 PM
  CPU time: 00:00:00.0400576
  priority: 8
  thread state: Wait
```

如用户在**ThreadPoolTcpSrvr**进程的**ListThreads**输出中可以看到，线程池的线程将被作为新线程进行计算，用于主程序处理客户机的连接。当第二个客户机连接到服务器时，情况变得非常有趣。当第二个客户机连接到服务器，获取程序使用线程的情况。并非用户所期望的5个线程，用户将会看到7个线程。这是因为，当第二个线程池线程被创建，线程池将创建另外一个线程来管理现有的线程。

小结

本章说明了C#网络程序中的进程和线程。进程是通过基于进程优先权的OS进行控制的。每个进程都可以得到使用系统CPU和内存的机会。

.NET库的**Process**类允许用户控制C#程序中的进程。当前进程的最佳获取方法是**GetCurrentProcess()**方法。其他进程可以通过使用进程名称或者ID进行。一旦创建**Process**对象，**Process**类的方法可以用于控制。进程的属性可以使用**Process**类的属性进行显示。

每个进程都包含一个或者多个线程，在程序中，每个线程都使用单独的流。很多程序只有一个操作流程，贯穿程序的开始到结束。这些程序只包含一个线程。其他程序可以创建第二个线程，当主应用程序在前台运行时，这些程序使用第二线程处理后台信息。

.NET库的**Thread**类允许用户创建并编辑**Thread**对象。每个**Thread**类创建一个新的线程，用于系统进程。一旦从**Process**类**Threads()**方法获得线程，这些线程将被加入**ProcessThreadCollection**对象。每个线程从**ProcessThreadCollection**对象中提取出来，并且得到线程状态和系统使用的信息。

C#网络程序可以创建第二个线程，用于处理多个客户机的连接。当接收到新客户机的连接后，该连接将被传递到一个新的线程。每个线程都独立于主程序，该主程序继续等待新的连接请求。新的线程接下来会处理客户机发送的外部数据。

尽管创建新线程非常有用，但是同样会需要额外的CPU资源。创建大量的新线程将导致系统性能的下降。一个可供选择的方法就是使用**ThreadPool**对象，该对象允许用户使用线程池中的线程来处理新的客户机连接。这些线程已经被创建，并且等待分配任务。这项技术可以避免为新客户机连接而创建新线程带来的过载。

在下一章中，用户将学习**IP组播**，该技术允许单独的设备通过发送单独的数据包与多个远程的客户机进行通信。这样可以降低程序向多个主机发送同样信息时对网络带宽的需求。

第10章 IP组播

- 广播、本地与全球
- 使用C#语言进行广播
- 使用广播包
- 如何进行组播
- C#套接字组播
- C# UdpClient组播
- 组播应用程序的例子

前面章节中使用的所有例子程序都有一个共性（不是指使用C#语言）：这些例子是用来一次向某一个远程设备发送消息。如果涉及到多个客户机，那么要将每条消息单独向每个客户机发送。

本章将改变介绍的风格，向用户介绍IP广播的概念，该方法可以同时向子网上的所有设备发送信息，而IP组播可以同时向网络上的多个设备发送信息。这些IP通信技术可以减少应用程序对网络带宽的需求，应用程序用户可以使用一个单独的数据包向一组设备同时发送信息。

我们将对IP广播做个概述，讲述一些使用该技术的C#网络程序例子。在此过程中，我们将提高第9章中的聊天程序（程序清单9.7，TcpChat.cs）。然后，我们将讲述IP组播的概念。我们将介绍两种控制组播对话的方法：对等网络和中心服务器与组客户机进行通信两种方法。“C# IP组播支持”一节介绍了如何使用Socket和UdpClient对象创建组播包和如何将这两个对象组合到网络程序中。

什么是广播

网络设备使用IP广播发送单独的信息包，可以到达网络上的所有设备。因为TCP通信需要两个设备使用专用的连接，因此在严格的TCP环境中无法发送广播消息。相反，必须使用UDP包，因为该协议具有在没有定义指定连接的情况下发送消息的能力。

本地与全球广播

广播消息包含一个特殊的目的地IP地址。IP地址格式允许两种类型的广播消息地址：本地广播和全球广播。

网络程序使用本地广播地址向子网上所有的设备发送广播消息。这样做的目的在于广播消息是局部的，其他网络不会受到广播的影响。

正如第2章“IP编程基础”中阐述的，IP地址分为两个部分，网络地址和主机地址。标

准网络地址部分组成了本地广播地址的第一部分，全部为1的地址用于主机地址部分（在八字节地址中是十进制值255）。图10.1为示例。因此，对于B类网络192.168.0.0，使用子网掩码255.255.0.0，本地广播地址将为192.168.255.255。

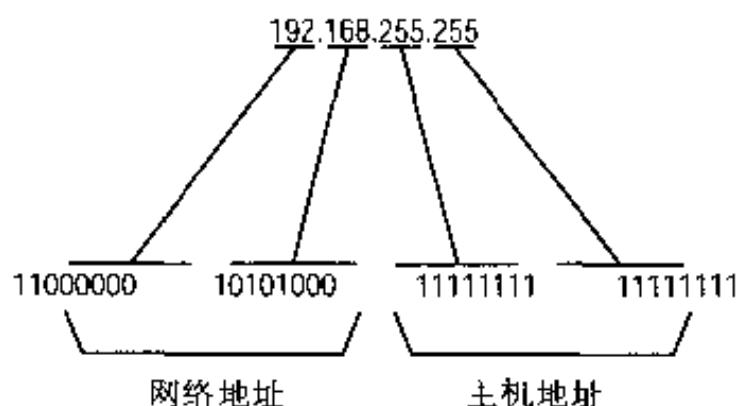


图10.1 本地广播地址的IP网络和主机地址部分

类似的，如果子网使用子网掩码255.255.255.0进行进一步细化。子网192.168.1.0将具有广播地址192.168.1.255，直到子网192.168.254.0，它具有广播地址192.168.254.255。

全球广播（global broadcast）最初的目的是允许设备向互联网络上的所有设备发送数据包。在IP地址中使用全部为1的地址，创建地址255.255.255.255。这个特定的广播地址显示，数据包的目的地址是网络上能见到该消息的所有设备。

因特网的巨大范围和循序的增长，意味着这种方法将发生改变。否则，某些无聊的程序将创建一个全球广播数据流，向因特网的所有网络发送，导致全世界的系统阻塞，影响正常的传输。要排除这种可能性，除非特定的配置（实际不可能），路由器不能发送全球IP广播到其他网络。相反，路由器将对全球广播不予理睬，有效地将这些全球广播转变为局部广播，只对所需的本地网络进行广播。

使用C#进行广播

.NET网络库包含用于发送和接收广播数据包的元素。本节内容描述在用户网络应用程序中处理广播数据包的必要技术。

发送广播数据包

在默认情况下，套接字不允许发送广播消息。用户可以通过运行程序清单10.1中的示例程序进行测试。

程序清单10.1 BadBroadcast.cs程序

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class BadBroadcast
{
```

```
public static void Main()
{
    Socket sock = new Socket(AddressFamily.InterNetwork, SocketType.Dgram,
                             ProtocolType.Udp);
    IPEndPoint iep = new IPEndPoint(IPAddress.Broadcast, 9050);
    byte[] data = Encoding.ASCII.GetBytes("This is a test message");
    sock.SendTo(data, iep);
    sock.Close();
}
}
```

当用户试图运行这个程序，将得到一个异常消息：

C:\>**BadBroadcast**

```
Unhandled Exception: System.Net.Sockets.SocketException: An attempt was made to
access a socket in a way forbidden by its access permissions
   at System.Net.Sockets.Socket.SendTo(Byte[] buffer, Int32 offset, Int32 size,
SocketFlags socketFlags, EndPoint remoteEP)
   at System.Net.Sockets.Socket.SendTo(Byte[] buffer, EndPoint remoteEP)
   at BadBroadcast.Main()
```

C:\>

BadBroadcast.cs程序看起来非常简单，但是在运行时，.NET环境不允许**SendTo()**方法发送消息。如果用户试图从默认的UDP套接字发送广播消息，用户将得到一个**SocketException**错误。

对于C#应用程序发送广播消息，广播套接字选项必须设置在使用**Socket**类的**SetSocketOption()**方法的创建套接字（在第3章“C#网络编程的类”中讨论过**SetSocketOption**）。这里的代码段就是在这种方式下使用的**SetSocketOption()**方法：

```
Socket sock = new Socket(AddressFamily.InterNetwork,
                          SocketType.Dgram, ProtocolType.Udp);
sock.SetSocketOption(SocketOptionLevel.Socket, SocketOptionName.Broadcast, 1);
```

注意，**SocketType**必须设置为**Dgram**进行广播消息。**SocketOptionName.Broadcast**选项是一个**Socket**级别的选项，所以**SocketOptionLevel**参数必须设置为**Socket**。

因为**SetSocketOption()**方法不允许数据参数为**boolean**值，用户必须设置参数值为任何非“0”值，表示**true boolean**值。“0”表示**false boolean**值。

当套接字选项设置后，用户指定一个广播地址和端口，用于广播和使用**SendTo()**方法发送消息：

```
IPEndPoint iep = new IPEndPoint(IPAddress.Broadcast, 9050);
byte[] data = Encoding.Ascii.GetBytes("test message");
sock.SendTo(data, iep);
```

使用IPAddress类的Broadcast特性作为目的地地址，程序将发送广播消息到设备上配置的所有网络接口。要限定广播到指定的端口，用户必须手动提供一个本地广播地址，作为目的地地址。程序清单10.2表明在C#程序Broadcast.cs中使用IPAddress.Broadcast地址和本地广播地址。

程序清单10.2 Broadcast.cs程序

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class Broadcast
{
    public static void Main()
    {
        Socket sock = new Socket(AddressFamily.InterNetwork, SocketType.Dgram,
                                   ProtocolType.Udp);

        IPEndPoint iep1 = new IPEndPoint(IPAddress.Broadcast, 9050);
        IPEndPoint iep2 = new IPEndPoint(IPAddress.Parse("192.168.1.255"), 9050);

        string hostname = Dns.GetHostName();
        byte[] data = Encoding.ASCII.GetBytes(hostname);
        sock.SetSocketOption(SocketOptionLevel.Socket,
                              SocketOptionName.Broadcast, 1);

        sock.SendTo(data, iep1);
        sock.SendTo(data, iep2);
        sock.Close();
    }
}
```

Broadcast程序创建两个单独的IPEndPoint对象，一个使用IPAddress.Broadcast地址，另一个使用本地广播地址192.168.1.255。接下来，设备的本地主机名称使用标准的Dns GetHostName()方法，并且在两个单独的广播消息中进行发送。

如用户在本书所见，用户可以使用WinDump或Analyzer监视网络的广播消息。图10.2显示了在Analyzer中Broadcast程序的输出。

Analyzer程序捕获两个单独的广播消息包发送到网络。第一个数据包发送到255.255.255.255全球广播地址，将从设置目的地地址到IPAddress.Broadcast获得结果。第二个广播数据包是发送到本地地址192.168.1.255，在程序中手动指定的本地广播地址。

接收广播数据包

接收广播数据包要比发送略微复杂。在默认情况下，没有特殊的选项可以用于套接字接收广播消息。如果套接字收听指定的UDP端口，套接字将接受任何发送到广播地址或者本

地IP地址的消息。程序清单10.3中的RecvBroadcast.cs程序通过创建一个在UDP端口9050上收听数据包的套接字，示例了这种情况，在同样的端口号上，Broadcast程序用于发送广播消息。

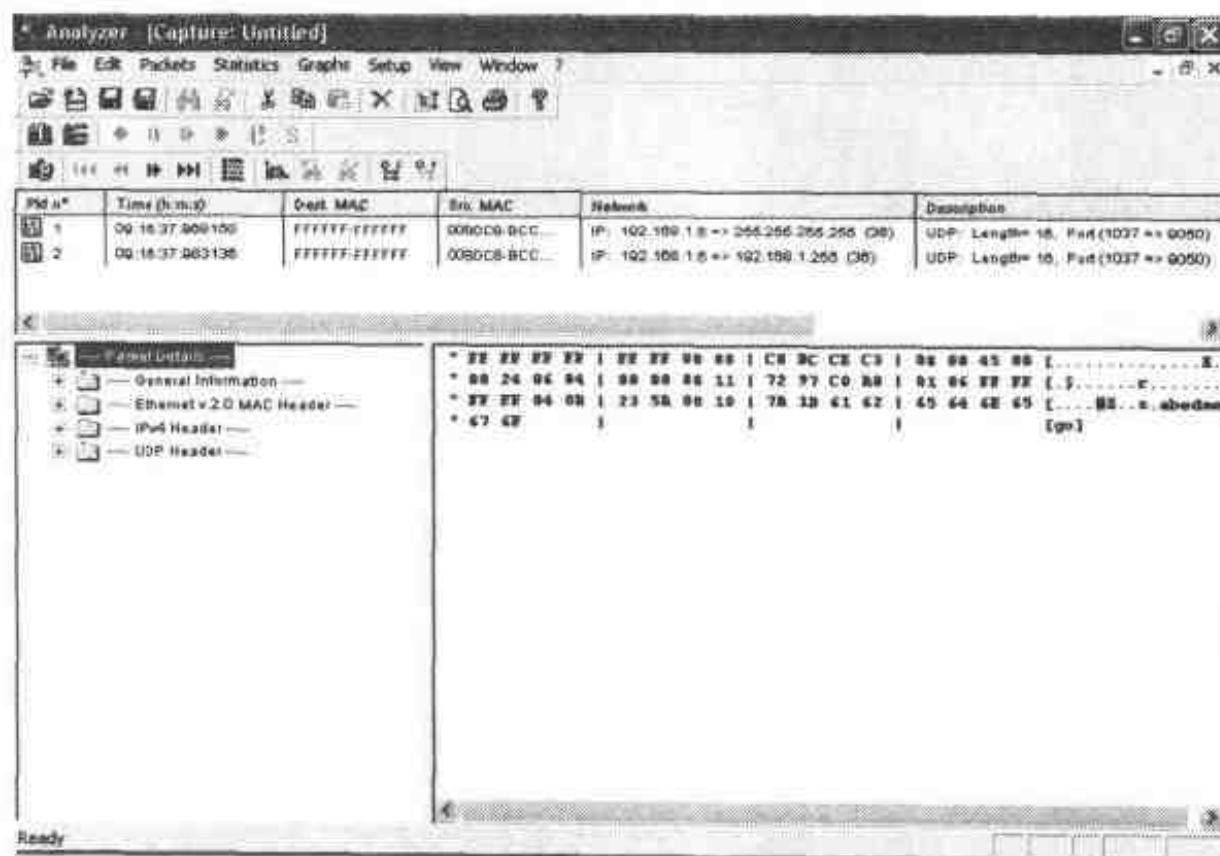


图10.2 在Analyzer中Broadcast程序的输出

程序清单10.3 RecvBroadcast.cs程序

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class RecvBroadcast
{
    public static void Main()
    {
        Socket sock = new Socket(AddressFamily.InterNetwork,
                                   SocketType.Dgram, ProtocolType.Udp);
        IPEndPoint iep = new IPEndPoint(IPAddress.Any, 9050);
        sock.Bind(iep);
        EndPoint ep = (EndPoint)iep;
        Console.WriteLine("Ready to receive...");

        byte[] data = new byte[1024];
        int recv = sock.ReceiveFrom(data, ref ep);
        string stringData = Encoding.ASCII.GetString(data, 0, recv);
        Console.WriteLine("received: {0} from: {1}",
                          stringData, ep.ToString());

        data = new byte[1024];
        recv = sock.ReceiveFrom(data, ref ep);
    }
}
```



```
stringData = Encoding.ASCII.GetString(data, 0, recv);
Console.WriteLine("received: {0} from: {1}",
                  stringData, ep.ToString());
sock.Close();
}
}
```

RecvBroadcast程序是一个简单的UDP程序，该程序创建一个套接字，并绑定到设备上的所有的网络接口，使用9050号端口，在同样的端口上，**Broadcast**程序用于发送广播消息。在绑定套接字到UDP端口后，程序试图接收发送到端口的两条消息，并且显示消息和发送主机地址信息。

用户可以在命令提示符窗口下启动**RecvBroadcast**程序，然后从同一设备的其他命令行提示符窗口或者同一子网上的其他设备上运行**Broadcast**程序。**RecvBroadcast**程序的示范输出结果如下：

```
C:\>RecvBroadcast
received: abednego from: 192.168.1.6:1042
received: abednego from: 192.168.1.6:1042
C:\>
```

如所期望的那样，**RecvBroadcast**程序接收到从**Broadcast**程序发送的广播消息。

警告：当测试广播程序时，记住要使用在同一子网的设备。

用广播包发布服务器广告

广播包的一个通用用途就是在网络上广告某个服务器的存在。通过在设置的时间间隔内重复发送包含有服务器信息的广播包，用户可以让客户机轻松地确定本地子网上服务器的情况。这个技术可以用于多种类型的应用程序，从Windows应用程序服务器到网络打印机。

本节将讲述一个简单的方法，可以在第9章“使用线程”的**TcpChat**程序中（程序清单9.7）加入广播广告。这个版本允许客户机知道在同一子网内的其他聊天程序何时启动。

广告循环

在网络应用程序中加入广告的最简单方法是创建一个后台线程，可以在设置的时间间隔内重复发送包含服务器信息的广播。该技巧通过重复发送广告，使得客户机可以快速发现广播服务器，但是本地网络可能无法忍受大量的广告数据流。通常情况下采用1分钟作为时间间隔。

广告线程是自包含的，创建线程本身的套接字并且在预定的时间间隔内发送广播包。数据包将作为后台线程被创建，并且分配一个比主程序线程低的优先级。这样可以确保广告线程不会抢夺主程序的处理时间，从而影响程序的响应。

在程序清单10.4中的**Advertise.cs**程序创建一个后台线程，每分钟发送包含运行应用程序

的主机信息的广播包。

程序清单10.4 Advertise.cs程序

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;
using System.Threading;

class Advertise
{
    public static void Main()
    {
        Advertise server = new Advertise();
    }

    public Advertise()
    {
        Thread advert = new Thread(new ThreadStart(sendPackets));
        advert.IsBackground = true;
        advert.Start();
        Console.Write("Press Enter to stop");
        string data = Console.ReadLine();
    }

    void sendPackets()
    {
        Socket sock = new Socket(AddressFamily.InterNetwork,
                                   SocketType.Dgram, ProtocolType.Udp);
        sock.SetSocketOption(SocketOptionLevel.Socket,
                              SocketOptionName.Broadcast, 1);
        IPEndPoint iep = new IPEndPoint(IPAddress.Broadcast, 9050);

        string hostname = Dns.GetHostName();
        byte[] data = Encoding.ASCII.GetBytes(hostname);
        while (true)
        {
            sock.SendTo(data, iep);
            Thread.Sleep(60000);
        }
    }
}
```

Advertise.cs程序使用sendPackets()方法创建一个新的Thread对象，其为广播地址创建一个Socket对象，每60秒钟广播一次主机名称。Thread.Sleep()方法将线程设置为广播间的idle

模式。广播线程利用Thread类的IsBackground特性放置在后台模式。

因为应用程序的主体只创建一个后台线程，因此程序必须等待，否则后台线程将被中断。在中断主程序前，使用Console.ReadLine()方法等待控制台的输入。

用户可以通过运行RecvBroadcast程序来监视Advertise.cs程序，该程序将显示头两个广播消息。用户可以使用WinDump或者Analyzer程序来监视广播消息如何在网络上发送。任何在9051号端口上收听的客户机都将接收到消息，并且知道服务器目前可用。

在第9章中介绍的TcpChat程序（程序清单9.7）有一个严重的缺陷：对于该程序的通信，用户必须使用手工方式将远程主机的地址汇编到程序中——明显地，对于商用应用程序来说，这不是一种好方法。根据对广播消息的掌握，用户可以对TcpChat程序进行修改，使该程序对用户更加友好。

要达到这个目的，需要在原始的TcpChat程序中加入两个部分：

- 一个后台线程，在子网上广播一个识别消息
- 一种监听广播消息并且检测子网上运行的其他聊天程序的方法

如在Advertise.cs程序中看到的，第一部分非常简单：用户可以通过使用Thread类创建后台线程，使用和对话通信分开的UDP端口来广播一个消息。第二部分——接收器——是需要执行的一些技巧。

首先，程序必须能够收听子网上聊天程序的广播消息。这项工作可以通过其他的后台线程来收听网络上的广播。当检测到其他聊天程序的广播消息，这些信息将被列表显示，以便用户做出选择。创建该列表的问题在于，所有客户机都会每分钟发送一次信息。用户无法简单地将任何消息加入列表中，或者某个单独的聊天客户机将出现多次。一个高质量的产品级应用程序必须保持每个客户机的轨迹，并且加入列表中。对于简单的例子，用户可以通过让程序只将第一个广告聊天程序的信息放入ListBox来解决这个问题。

这个进程的第二部分是允许用户在列表框中做出选择（在这个例子中列表框为空）。然后用户需要单击Connect按钮后与选择的客户机进行聊天。远程客户机的主机名称和地址都是通过广播包中的信息收集的，然后聊天连接将被创建。

NewTcpChat程序

所有这些功能都在NewTcpChat.cs程序中进行了描述，如程序清单10.5所示。

程序清单10.5 NewTcpChat.cs程序

```
using System;
using System.Drawing;
using System.Net;
using System.Net.Sockets;
using System.Text;
using System.Threading;
using System.Windows.Forms;

class NewTcpChat : Form
```

```
{
    private static TextBox newText;
    private static ListBox results;
    private static ListBox hosts;
    private static Socket client;
    private static byte[] data = new byte[1024];

    public NewTcpChat()
    {
        Text = "New TCP Chat Program";
        Size = new Size(400, 380);

        Label label1 = new Label();
        label1.Parent = this;
        label1.Text = "Enter text string:";
        label1.AutoSize = true;
        label1.Location = new Point(10, 30);

        newText = new TextBox();
        newText.Parent = this;
        newText.Size = new Size(200, 2 * Font.Height);
        newText.Location = new Point(10, 55);

        results = new ListBox();
        results.Parent = this;
        results.Location = new Point(10, 85);
        results.Size = new Size(360, 10 * Font.Height);

        Label label2 = new Label();
        label2.Parent = this;
        label2.Text = "Active hosts";
        label2.AutoSize = true;
        label2.Location = new Point(10, 240);

        hosts = new ListBox();
        hosts.Parent = this;
        hosts.Location = new Point(10, 255);
        hosts.Size = new Size(360, 5 * Font.Height);

        Button sendit = new Button();
        sendit.Parent = this;
        sendit.Text = "Send";
        sendit.Location = new Point(220, 52);
        sendit.Size = new Size(5 * Font.Height, 2 * Font.Height);
        sendit.Click += new EventHandler(ButtonSendOnClick);

        Button connect = new Button();
        connect.Parent = this;
        connect.Text = "Connect";
    }
}
```

```

connect.Location = new Point(295, 20);
connect.Size = new Size(6 * Font.Height, 2 * Font.Height);
connect.Click += new EventHandler(ButtonConnectOnClick);

Button listen = new Button();
listen.Parent = this;
listen.Text = "Listen";
listen.Location = new Point(295, 52);
listen.Size = new Size(6 * Font.Height, 2 * Font.Height);
listen.Click += new EventHandler(ButtonListenOnClick);

Thread fh = new Thread(new ThreadStart(findHosts));
fh.IsBackground = true;
fh.Start();
}

void ButtonListenOnClick(object obj, EventArgs ea)
{
    results.Items.Add("Listening for a client...");
    Socket newsock = new Socket
        (AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);
    IPEndPoint iep = new IPEndPoint(IPAddress.Any, 9050);
    newsock.Bind(iep);
    newsock.Listen(5);
    newsock.BeginAccept(new AsyncCallback(AcceptConn), newsock);
    Thread advertise = new Thread(new ThreadStart(srvrAdvertise));
    advertise.IsBackground = true;
    advertise.Start();
}

void ButtonConnectOnClick(object obj, EventArgs ea)
{
    results.Items.Add("Connecting...");
    client = new Socket
        (AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);
    string selectedhost = (string)hosts.SelectedItem;
    string[] hostarray = selectedhost.Split(':');
    IPEndPoint iep = new IPEndPoint(IPAddress.Parse(hostarray[1]), 9050);
    client.BeginConnect(iep, new AsyncCallback(Connected), client);
}

void ButtonSendOnClick(object obj, EventArgs ea)
{
    byte[] message = Encoding.ASCII.GetBytes(newText.Text);
    newText.Clear();
    client.BeginSend(message, 0, message.Length, 0,
        new AsyncCallback(SendData), client);
}

```

```
}

void AcceptConn(IAsyncResult iar)
{
    Socket oldserver = (Socket)iar.AsyncState;
    client = oldserver.EndAccept(iar);
    results.Items.Add("Connection from: " + client.RemoteEndPoint.ToString());
    Thread receiver = new Thread(new ThreadStart(ReceiveData));
    receiver.IsBackground = true;
    receiver.Start();
}

void Connected(IAsyncResult iar)
{
    try
    {
        client.EndConnect(iar);
        results.Items.Add("Connected to: " + client.RemoteEndPoint.ToString());
        Thread receiver = new Thread(new ThreadStart(ReceiveData));
        receiver.IsBackground = true;
        receiver.Start();
    } catch (SocketException)
    {
        results.Items.Add("Error connecting");
    }
}

void SendData(IAsyncResult iar)
{
    Socket remote = (Socket)iar.AsyncState;
    int sent = remote.EndSend(iar);
}

void ReceiveData()
{
    int recv;
    string stringData;
    while (true)
    {
        recv = client.Receive(data);
        stringData = Encoding.ASCII.GetString(data, 0, recv);
        if (stringData == "bye")
            break;
        results.Items.Add(stringData);
    }
    stringData = "bye";
}
```

```
byte[] message = Encoding.ASCII.GetBytes(stringData);
client.Send(message);
client.Close();
results.Items.Add("Connection stopped");
return;
}

void srvrAdvertise()
{
    Socket server = new Socket(AddressFamily.InterNetwork,
                               SocketType.Dgram, ProtocolType.Udp);
    server.SetSocketOption(SocketOptionLevel.Socket,
                           SocketOptionName.Broadcast, 1);
    IPEndPoint iep = new IPEndPoint(IPAddress.Broadcast, 9051);
    byte[] hostname = Encoding.ASCII.GetBytes(Dns.GetHostName());
    while (true)
    {
        server.SendTo(hostname, iep);
        Thread.Sleep(60000);
    }
}

void findHosts()
{
    while (true)
    {
        Socket remoteHosts = new Socket(AddressFamily.InterNetwork,
                                         SocketType.Dgram, ProtocolType.Udp);
        IPEndPoint iep = new IPEndPoint(IPAddress.Any, 9051);
        EndPoint ep = (EndPoint)iep;
        remoteHosts.Bind(iep);
        byte[] data = new byte[1024];
        int recv = remoteHosts.ReceiveFrom(data, ref ep);
        string stringData = Encoding.ASCII.GetString(data, 0, recv);
        string entry = stringData + ":" + ep.ToString();
        if (!hosts.Items.Contains(entry))
            hosts.Items.Add(entry);
    }
}

public static void Main()
{
    Application.Run(new NewTcpChat());
}
}
```

NewTcpChat程序使用TcpChat程序的基本形式，但是为广播增加了新的部分。当单击Listen按钮，将程序设置在收听方式，后台线程被启动。srvrAdvertise()方法将执行广播广告程序，在网络上宣布聊天程序所在系统的名称。这样就可以确保只有运行聊天程序的客户机才会收到网络上的广告。

NewTcpChat的第二个提高就是使用新的方法findHosts()，该方法在UDP 9051号端口上进行收听。因为这个方法在应用程序启动时必须运行在后台线程中，所以该方法被放置在主程序部分。

最后，注意Connect按钮方法已经被修改。从列表框中的被选择的条目分析出目的地IP地址，取代预先的地址，并被用于连接尝试。

测试新应用程序

不幸的是，因为NewTcpChat程序自动在UDP 9051号端口上收听广播，用户在机器上只能启动一个程序实体（记住，每次只能有一个程序绑定到端口）。要测试这个程序，用户要使用同一子网上的两个机器。

在两个机器的命令行提示符窗口中启动NewTcpChat程序。在第一个机器上，单击Listen按钮，将其设置在收听方式。其他的程序将看到收听程序的广播，并且将机器的主机名称、IP地址和端口号码显示在列表框中，如图10.3所示。

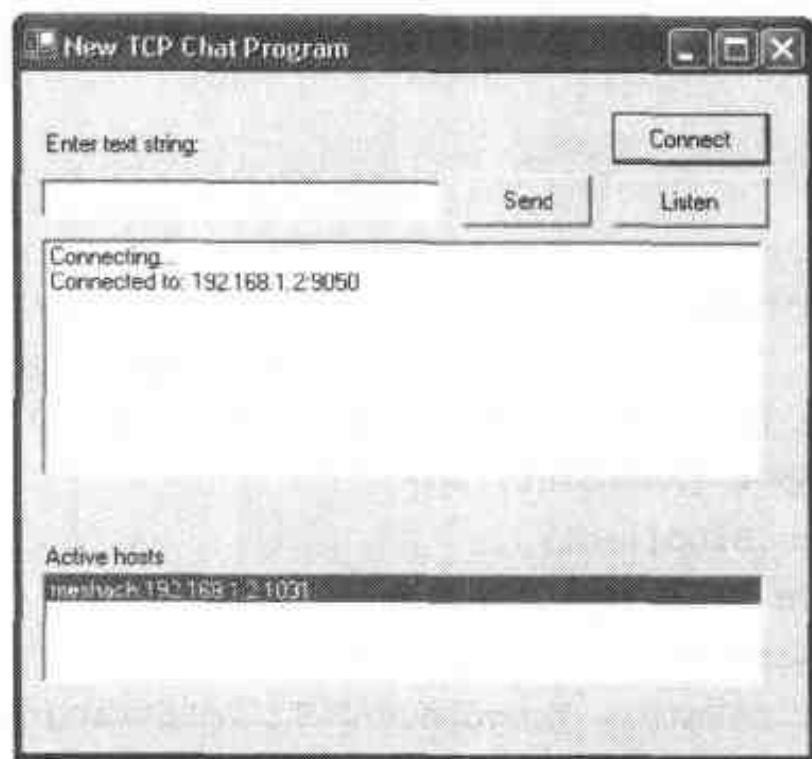


图10.3 察觉到远程聊天程序后的NewTcpChat程序

当用户在列表框中看到主机信息，单击选择该信息，然后单击Connect按钮。这样就可以和远程聊天程序建立连接，并且用户可以开始发送文本消息。

什么是组播

广播是一种非常出色的方法，可以向子网上的所有设备发送信息，但是该方法有一个缺点：广播包只能限制于子网内。IP组播的设计目标就是允许应用程序可以向本网或者全网

内选择的设备子集发送单独的数据包。这个功能允许应用程序加入一个组播组，进行广域的会议。

与广播类似，IP组播使用特殊的IP地址。IP组播使用特殊的IP地址范围，来表示不同的组播组。每个组播组由一系列收听同一IP地址的设备组成。当包发送到组播组地址，每个处于收听状态的设备都会收到该包。

IP地址224.0.0.1到239.255.255.255都表示组播组。根据因特网请求说明（RFC）3171，组的分配如表10.1所示。

表10.1 IP组播地址分配

范围	分配
224.0.0.0~224.0.0.255	本地网络控制块
224.0.1.0~224.0.1.255	网间控制块
224.0.2.0~224.0.255.0	AD-HOC块
224.1.0.0~224.1.255.255	ST组播组
224.2.0.0~224.2.255.255	SDP/SAP块
224.252.0.0~224.255.255.255	DIS临时块
225.0.0.0~231.255.255.255	保留的
232.0.0.0~232.255.255.255	特定源组播块
233.0.0.0~233.255.255.255	GLOP块
234.0.0.0~238.255.255.255	保留的
239.0.0.0~239.255.255.255	管理范围块

在这些地址块中，单独的IP地址被分配到特殊的项目。例如，地址224.0.0.1和224.0.0.2保留，用于路由器之间进行组播组信息的通信。用户要避免使用这些块中的组播组地址。

组播技术

- 控制组播会话，需要使用两种技术：
- 对等网络技术，通过这种技术，组内的所有客户机之间都可以互相发送消息
 - 中央服务器发送消息到组内的客户机

对等网络技术

在对等网络组播组（图10.4）中，组播组中的所有客户机都享有同样的权利。在组中的任何客户机都可以与组内的其他客户机交换消息。

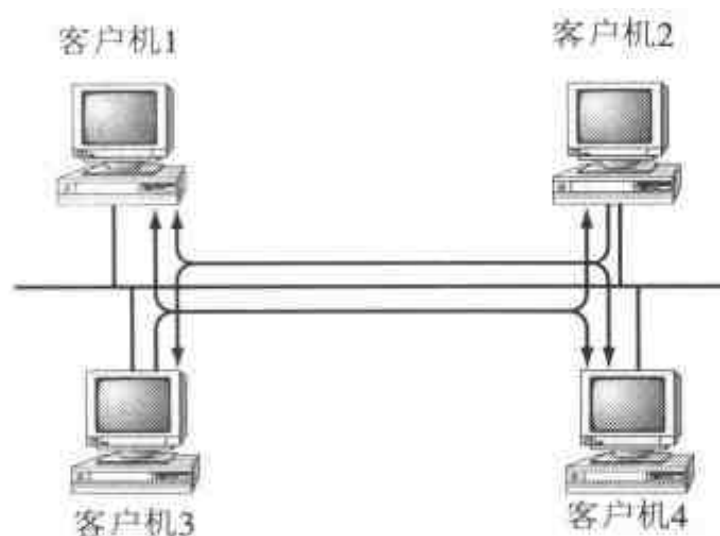


图10.4 在对等网络组播组中，所有的客户机可以与组内的其他客户机交换消息

IP系统通过允许网络上的任何设备接收或者发送目的地为组播组IP地址的包，来支持对等网络组播组。在默认情况下，客户机加入组播组是没有限制的。某些工具可以执行加密技术，防止未经授权的客户机对接收的消息进行解密，但是没有方法可以阻止客户机接收数据。

中央服务器

其他的组播系统使用中央服务器，在网络上，这个服务器将控制所有的组播组行动。单独的客户机如果要介入组播组，必须从中央服务器得到许可。如果中央服务器拒绝客户机加入组播组，将不会有组播包发送到请求的客户机。该技术如图10.5所示。

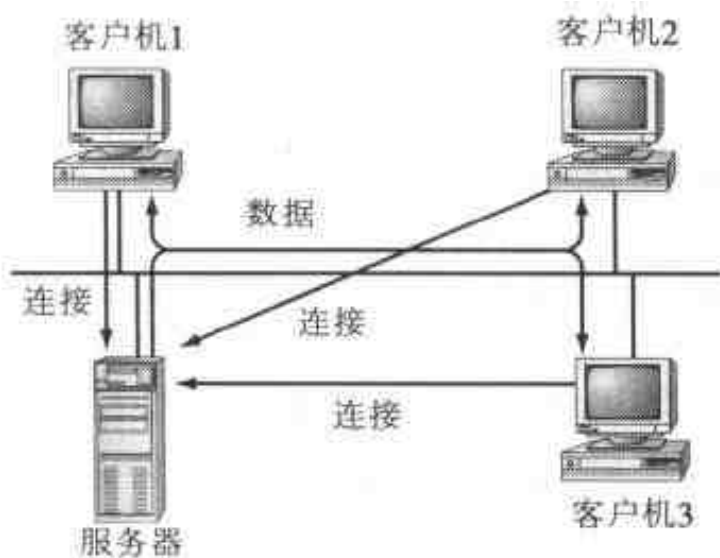


图10.5 中央服务器控制组播组

说明：中央服务器组播组系统无法使用IP。当然，异步传输模式（ATM）网络是惟一支持中央服务器组播组的网络。

通过路由器发送组播包

尽管组播包可以通过网络边界进行传递，要完成此项工作，部分网络路由器需要进行改进。在默认情况下，大多数路由器无法传递组播包到其他子网。如果路由器将所有接收到的组播包传递到所有的接口，路由器将导致网络流量因为组播包而剧增。相反，已经开发了

一个系统，允许被选择的组播包被转发。

因特网组管理协议（IGMP）可以帮助路由器完成将组播包传递到不同的子网。当某个网络设备要加入组播组，设备需要发送IGMP包到本地子网的路由器。IGMP包将注册网络设备和组播组地址，设备必须从该地址收到消息。这样就可以让路由器知道，必须将接收到的组播消息转发到网络设备组。图10.6描述了组播组注册的原理。

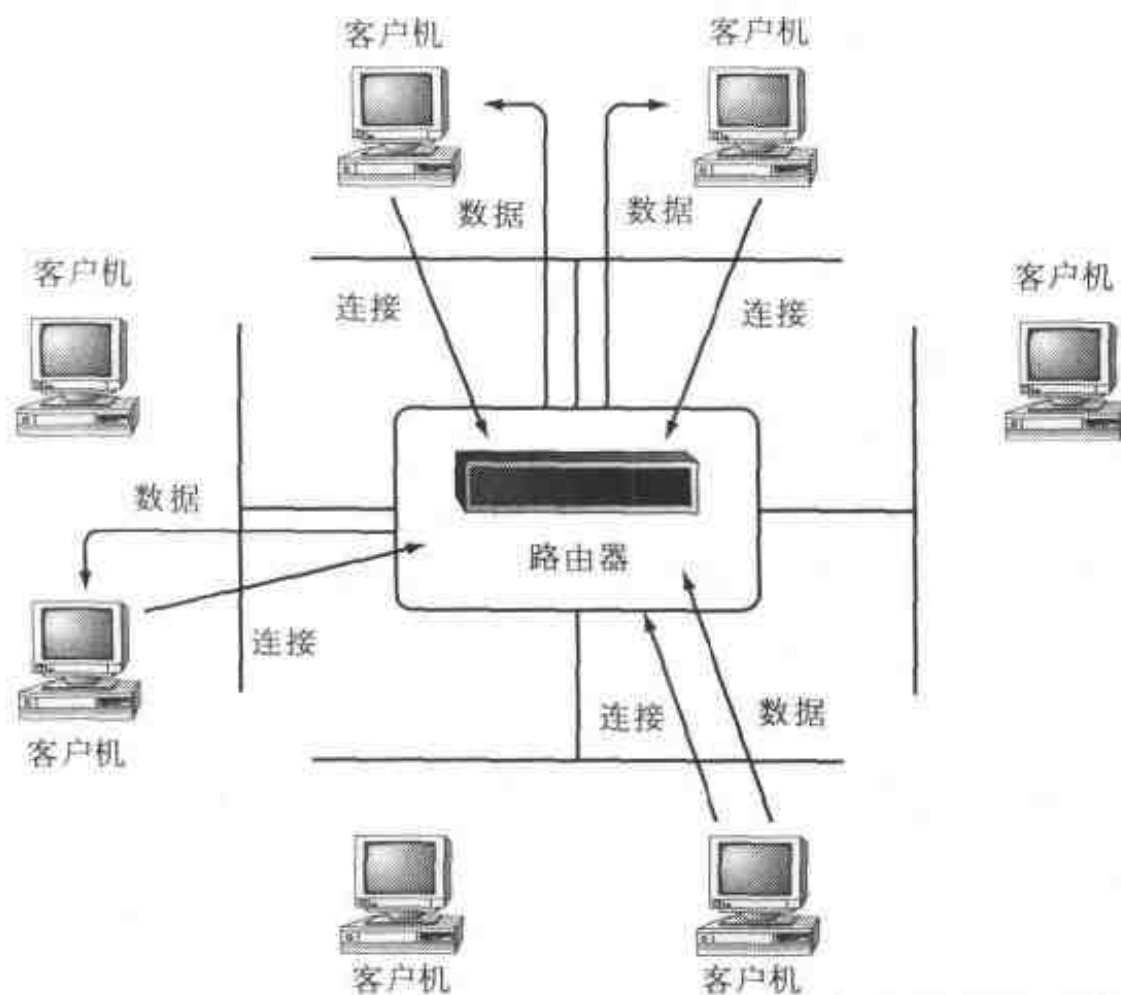


图10.6 网络设备在路由器上注册，成为组播组成员

如图10.6所示，当每个网络设备注册到组播组，路由器必须将接收到的组播包转发到包含注册主机的接口。不包含注册主机的接口将不会接收组播包。

类似的，当网络主机离开组播组，另外一个IGMP包将发送到路由器，告知主机将不再需要组播组转发的包。当路由器某个接口上的最后一个网络主机离开，路由器将停止向该接口转发组播包。

C# IP组播支持

.NET网络库通过使用Socket选项支持IP组播。一旦创建Socket对象，该对象将设置加入组播组或者离开组播组。该技术可以用于UdpClient对象。

C#套接字组播

Socket类通过使用套接字的SetSocketOption()方法支持IP组播，但是与广播采用不同的方法。

这里有两个套接字选项，可以用于组播：

- 增加套接字到组播组
- 从组播组中删除套接字

这些操作都是通过IP级Socket选项名称AddMembership和DropMembership定义的。

SetSocketOption()方法的值参数不是普通的数据值。相反，该方法使用MulticastOption类。

MulticastOption类定义组播组，套接字将被增加或者删除。这两个构造器用于MulticastOption类：

```
MulticastOption(IPAddress)
MulticastOption(IPAddress, IPAddress)
```

第一个构造器格式定义了IP组播地址，用于SetSocketOption。该构造器的默认行为是允许系统上的所有接口都可以被SetSocketOption()方法操作。如果用户要限制系统上某个接口的行为，第二个构造器可以允许具体说明接口的IPAddress。

例如，如果用户希望在组播组224.100.0.1增加一个套接字，用户将使用下面的语句：

```
sock.SetSocketOption(SocketOptionLevel.IP, SocketOptionName.AddMembership,
    new MulticastOption(IPAddress.Parse("224.100.0.1"));
```

这个设置将套接字加入组播组，用于系统上所有的接口。

接收组播包

在程序清单10.6中的MultiRecv.cs程序设置套接字可以接收发送到224.100.0.1组播组的组播包。

程序清单10.6 MultiRecv.cs程序

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class MultiRecv
{
    public static void Main()
    {
        Socket sock = new Socket(AddressFamily.InterNetwork,
                                   SocketType.Dgram, ProtocolType.Udp);
        Console.WriteLine("Ready to receive...");

        IPEndPoint iep = new IPEndPoint(IPAddress.Any, 9050);
        EndPoint ep = (EndPoint)iep;
        sock.Bind(iep);
    }
}
```

```

        sock.SetSocketOption(SocketOptionLevel.IP, ➡
        SocketOptionName.AddMembership, ➡
        new MulticastOption(IPAddress.Parse("224.100.0.1")));

        byte[] data = new byte[1024];
        int recv = sock.ReceiveFrom(data, ref ep);
        string stringData = Encoding.ASCII.GetString(data, 0, recv);
        Console.WriteLine("received: {0} from: {1}", stringData, ep.ToString());
        sock.Close();
    }
}

```

MultiRecv程序创建一个UDP套接字，使用标准的**Socket**类方法。在将套接字加入组播组后，套接字将阻塞**ReceiveFrom()**方法的调用，等待数据的到达。

关于组播的一些注意事项

在使用组播套接字时，有两个非常重要的要点需要记住：

首先，**SetSocketOption()**方法调用必须在**Bind()**方法调用后执行。这样就允许组播组设置特殊的**IPEndPoint**地址到套接字。

第二，套接字一旦被加入组播组，**ReceiveFrom()**方法将接受下列包：

- 在**Bind()**方法中说明的**IPEndPoint**地址和端口
- 在**MulticastOption**构造器中指定的组播组IP地址和在**IPEndPoint**对象中指定的端口
- 用于指定**IPEndPoint**端口的广播包

很多新的网络编程人员会忽视第二条。当执行**ReceiveFrom()**方法时，用户无法保证只接收到组播组的包。所以，要小心其他源发送的包。用户将很难区分这些包。

发送组播包

发送组播包非常简单。要向组播组中的成员发送包，无需进行什么特殊的工作，用户只需指明组播组IP地址，作为包的目的地地址。程序清单10.7使用**MultiSend.cs**程序进行了说明。

程序清单10.7 MultiSend.cs程序

```

using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class MultiSend
{
    public static void Main()
    {

```

```

        Socket server = new Socket(AddressFamily.InterNetwork,
                                    SocketType.Dgram, ProtocolType.Udp);
        IPEndPoint iep = new IPEndPoint(IPAddress.Parse("224.100.0.1"), 9050);
        byte[] data = Encoding.ASCII.GetBytes("This is a test message");
        server.SendTo(data, iep);
        server.Close();
    }
}

```

注意，在套接字上没有使用**SetSocketOption()**。如果套接字只是发送包到本地子网的组播组，是没有必要使用该方法的。用户以后将会看到，如果希望转发组播包到其他网络，必须使用**SetSocketOption()**。同样的，如果套接字需要接收组播包，也必须使用**SetSocketOption()**和适当的**MulticastOption**构造器。

监视程序

用户可以在自己的系统上测试这些程序，并且使用**WinDump**或**Analyzer**监视网络上的传输。在这个例子中，因为**MulticastOption**构造器没有指定接口，即使发送器和接收器都运行在同一台机器上，用户也可以监视网络包。

首先，在命令行提示符窗口启动**MultiRecv**程序。然后在第二个命令行提示符窗口运行**MultiSend**程序。用户将看到下列文本和发送机器的地址，显示在**MultiRecv**控制台中：

```

C:\>MultiRecv
received: This is a test message from: 192.168.1.6:1294
C:\>

```

更有趣的是运行**MultiSend**和**MultiRecv**的跟踪文件。程序清单10.8显示了从**WinDump**程序的输出。

程序清单10.8 组播测试的WinDump输出例子

```

C:\>windump -X ip host 158.18.125.32
windump listening on\Device\Packet_E190x1
09:05:20.776334 158.18.125.32 > 224.100.0.1: 158.18.125.32 > 224.100.0.1: igmp v
2 report 224.100.0.1 [ttl 1]
0x0000 4600 0020 f613 0000 0102 332c 9e12 7d20 F.....3,...}.
0x0010 e064 0001 9404 0000 1600 099a e064 0001 .d.....d..
09:05:24.357183 158.18.125.32 > 224.100.0.1: 158.18.125.32 > 224.100.0.1: igmp v
2 report 224.100.0.1 [ttl 1]
0x0000 4600 0020 fc13 0000 0102 2d2c 9e12 7d20 F.....-,...}.
0x0010 e064 0001 9404 0000 1600 099a e064 0001 .d.....d..
09:05:28.433814 158.18.125.32.1502 > 224.100.0.1.9050: udp 22 [ttl 1]
0x0000 4500 0032 0114 0000 0111 bd0f 9e12 7d20 E..2.....}.

```

```

0x0010    e064 0001 05de 235a 001e dfda 5468 6973    .d....#z....This
0x0020    2069 7320 6120 7465 7374 206d 6573 7361    .is.a.test.messa
0x0030    6765                                         ge
09:05:28.435433 158.18.125.32 > ALL-ROUTERS.MCAST.NET: 158.18.125.32 > ALL-ROUTE
RS.MCAST.NET: igmp leave 224.100.0.1 [ttl 1]
0x0000    4600 0020 0214 0000 0102 278f 9e12 7d20    F.....'...}.
0x0010    e000 0002 9404 0000 1700 089a e064 0001    .....d..

289 packets received by filter
0 packets dropped by kernel

C:\>

```

当启动MultiRecv程序，该程序将发送两个IGMP包到组播组地址。这些包不属于程序方法的部分，而是被.NET网络库功能自动发送的。

IGMP包用于通知网络上的所有路由器，设备将加入组播组，并且组播组地址的所有包将被转发到子网。当IGMP包发送后，MultiSend程序发送UDP包进行测试，如用户期望的，使用组播组地址作为目的地地址。

当MultiRecv程序关闭Socket对象，另一个IGMP对象将发送到网络，表示设备已经离开组播组。用户可能会注意到，这个包没有发送到组播组地址，而是发送到ALL-ROUTERS.MCAST.NET指定的地址。组播地址224.0.0.2，该地址是一个特殊的路由器控制组播组。这个包将通知网络上的所有路由器，这些路由器无需再将组播组包转发到该设备。

组播TTL值

如果用户仔细监视MultiSend程序发送的组播包，用户会注意到这些包具有TTL值1。对于该值，TCP包将不允许路由器进行转发。这表示在默认情况下，Socket类允许套接字发送消息到组播组，但是限于本地子网。要通过多个路由器发送组播包，用户必须增加组播包的TTL值。该过程可以通过SetSocketOption()方法实现，但是非常复杂。

对于接收的组播套接字，要使用SetSocketOption()方法，用户必须绑定套接字到本地IPEndPoint对象，说明套接字将被用于组播，然后设置一个新的TTL值。如下：

```

Socket sock = new Socket(AddressFamily.InterNetwork,
                          SocketType.Dgram, ProtocolType.Udp);
IPEndPoint iep = new IPEndPoint(IPAddress.Any, 9050);
sock.Bind(iep);
sock.SetSocketOption(SocketOptionLevel.IP, SocketOptionName.AddMembership,
                     new MulticastOption(IPAddress.Parse("224.100.0.1")));
sock.SetSocketOption(SocketOptionLevel.IP,
                     SocketOptionName.MulticastTimeToLive, 50);

```

现在，套接字已经被加入组播组，并且TTL值已经被改变为50。

使用这个概念，程序清单10.9中的NewMultiSend.cs程序可以跨网发送组播包。

程序清单10.9 NewMultiSend.cs程序

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class NewMultiSend
{
    public static void Main()
    {
        Socket server = new Socket(AddressFamily.InterNetwork,
                                    SocketType.Dgram, ProtocolType.Udp);
        IPEndPoint iep = new IPEndPoint(IPAddress.Any, 9051);
        IPEndPoint iep2 = new IPEndPoint(IPAddress.Parse("224.100.0.1"), 9050);
        server.Bind(iep);

        byte[] data = Encoding.ASCII.GetBytes("This is a test message");
        server.SetSocketOption(SocketOptionLevel.IP, SocketOptionName.AddMembership,
                               new MulticastOption(IPAddress.Parse("224.100.0.1")));
        server.SetSocketOption(SocketOptionLevel.IP,
                               SocketOptionName.MulticastTimeToLive, 50);
        server.SendTo(data, iep2);
        server.Close();
    }
}
```

现在，当发送组播包，该包具有一个值为50的TTL值，允许传送50次跳越。用户可以使用MultiRecv程序监视组播包在网络上传输。用户同样可以使用WinDump或者Analyzer程序监视包。

C# UdpClient组播

.NET UdpClient类同样支持IP组播组地址。UdpClient类使用特殊的方法允许套接字加入组播组，或者从组播组中被删除。下面的两种方法可以使用：

- JoinMulticastGroup()
- DropMulticastGroup()

如用户所期望的，JoinMulticastGroup()方法允许套接字接收发送到特殊组播组地址的消息。这个方法可以通过使用两种格式之一进行创建：

```
JoinMulticastGroup(IPAddress)
JoinMulticastGroup(IPAddress, int)
```


第一个构造器允许套接字使用`IPAddress`对象加入组播组。第二个构造器格式允许用户在套接字中增加TTL值，允许套接字接收发送到远程主机的组播包。

这里是一个例子代码段，使用`UdpClient`组播方法：

```
UdpClient uc = new UdpClient(9050);
uc.JoinMulticastGroup(IPAddress.Parse("224.100.0.1"));
```

警告：记住，在`UdpClient`构造器中要使用默认的UDP端口号码。如果用户没有使用端口号码，`JoinMulticastGroup()`方法将因为无法得知接收组播消息的端口而导致失败。

UdpClient组播接收器

在程序清单10.10中的`UdpClientMultiRecv`程序描述了在组播组中加入一个`UdpClient`对象的编程方法，以及接收组中的包。

程序清单10.10 `UdpClientMultiRecv.cs`程序

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class UdpClientMultiRecv
{
    public static void Main()
    {
        UdpClient sock = new UdpClient(9050);
        Console.WriteLine("Ready to receive...");
        sock.JoinMulticastGroup(IPAddress.Parse("224.100.0.1"), 50);
        IPEndPoint iep = new IPEndPoint(IPAddress.Any, 0);
        byte[] data = sock.Receive(ref iep);
        string stringData = Encoding.ASCII.GetString(data, 0, data.Length);
        Console.WriteLine("received: {0} from: {1}", stringData, iep.ToString());
        sock.Close();
    }
}
```

这里没有太多的幻想。用户创建了一个简单的`UdpClient`对象，并且使用`JoinMulticastGroup()`方法允许该对象接收发送到224.100.0.1组播组地址的消息。当接收到第一个消息后，套接字将被关闭，并且程序将中断。

UdpClient组播发送器

和`Socket`对象类似，`UdpClient`类不需要任何专门的方法发送数据包到组播组地址。只是跟随标准的`UdpClient`过程发送消息到指定的IP地址。程序清单10.11显示了`UdpClient`

MultiSend.cs程序，该程序示例了使用**UdpClient**方法发送消息到组播组地址。

程序清单10.11 UdpClientMultiSend.cs程序

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class UdpClientMultiSend
{
    public static void Main()
    {
        UdpClient sock = new UdpClient();
        IPEndPoint iep = new IPEndPoint(IPAddress.Parse("224.100.0.1"), 9050);
        byte[] data = Encoding.ASCII.GetBytes("This is a test message");
        sock.Send(data, data.Length, iep);
        sock.Close();
    }
}
```

程序测试

和**Socket**版本的组播程序一样，用户可以通过在命令提示符窗口首先启动**UdpClientMultiRecv**，然后再运行**UdpClientMultiSend**程序来对**UdpClient**版本的组播程序进行测试。用户将看到**UdpClientMultiRecv**程序控制台的测试消息。

因为传输是通过所有的接口，用户可以使用**WinDump**或**Analyzer**进行监视。程序清单10.12显示了**WinDump**跟踪的输出结果。和**Socket**类一样，**UdpClient**类发送两个**IGMP**数据包到希望加入组播组的设备的本地路由器。程序的剩余工作和**Socket**版本完全相同。

程序清单10.12 UdpClient组播测试的WinDump输出结果

```
C:\>windump -X ip host 158.18.125.32
windump listening on\Device\Packet_E190x1
10:52:19.653155 158.18.125.32 > 224.100.0.1: 158.18.125.32 > 224.100.0.1: igmp v
2 report 224.100.0.1 [ttl 1]
0x0000 4600 0020 f922 0000 0102 301d 9e12 7d20 F...."....0...}.
0x0010 e064 0001 9404 0000 1600 099a e064 0001 .d.....d..
10:52:21.615597 158.18.125.32 > 224.100.0.1: 158.18.125.32 > 224.100.0.1: igmp v
2 report 224.100.0.1 [ttl 1]
0x0000 4600 0020 fc22 0000 0102 2d1d 9e12 7d20 F...."....-...}.
0x0010 e064 0001 9404 0000 1600 099a e064 0001 .d.....d..
10:52:23.101905 158.18.125.32.1718 > 224.100.0.1.9050: udp 22 [ttl 1]
0x0000 4500 0032 fe22 0000 0111 c000 9e12 7d20 E..2.".....}.
0x0010 e064 0001 06b6 235a 001e df02 5468 6973 .d....#Z....This
```



```

0x0020  2069 7320 6120 7465 7374 206d 6573 7361      .is.a.test.messa
0x0030  6765                                           ge
10:52:23.104539 158.18.125.32 > ALL-ROUTERS.MCAST.NET: 158.18.125.32 > ALL-ROUTE
RS.MCAST.NET: igmp leave 224.100.0.1 [ttl 1]
0x0000  4600 0020 ff22 0000 0102 2a80 9e12 7d20      F....*....*...}.
0x0010  e000 0002 9404 0000 1700 089a e064 0001      .....d..

329 packets received by filter
0 packets dropped by kernel

C:\>

```

组播应用程序例子

在没有维持连接到每个设备或者发送多个消息包（每个消息包对应一个设备）的情况下，某个设备需要发送信息到多个远程主机设备时，可以使用组播应用程序。本节内容讲述了一个简单的组播白板/聊天系统，在该应用程序中，任何介入组播组的客户机都可以发送消息到组播组中的所有设备，并且从组里的其他设备接收消息。这个公共的应用程序可以允许用户组播通信。通过允许网络上的任何人加入组播组聊天，用户可以创建一个环境，在这个环境中，每个人都可以与整个组进行闲聊，并且看到从组中任何人发送的闲聊信息。

这个程序的基本格式就是第9章中介绍的Broadcast程序（程序清单9.7）。与发送信息包到每个单独的设备不同，聊天程序发送消息到224.100.0.1组播IP地址。任何收听这个组播组的客户机都将接收到消息。类似的，TcpChat程序将被修改，收听同样的组播组地址，并且显示从网络上其他聊天客户机上接收的消息。

修改现有TcpChat程序的最简单方法就是创建一个新的Thread对象，该对象监视组播组地址。当地址上接收到消息，将被显示在列表框中。消息发送功能将继续保持，不同的是组播组IP地址必须使用。程序清单10.13中的组播聊天程序使用了这一概念。

程序清单10.13 MulticastChat.cs程序

```

using System;
using System.Drawing;
using System.Net;
using System.Net.Sockets;
using System.Text;
using System.Threading;
using System.Windows.Forms;

class MulticastChat Form
{
    TextBox newText;
    ListBox results;
    Socket sock;

```

```
Thread receiver;
IPEndPoint multiep = new IPEndPoint(IPAddress.Parse("224.100.0.1"), 9050);

public MulticastChat()
{
    Text = "Multicast Chat Program";
    Size = new Size(400, 380);

    Label label1 = new Label();
    label1.Parent = this;
    label1.Text = "Enter text string:";
    label1.AutoSize = true;
    label1.Location = new Point(10, 30);

    newText = new TextBox();
    newText.Parent = this;
    newText.Size = new Size(200, 2 * Font.Height);
    newText.Location = new Point(10, 55);

    results = new ListBox();
    results.Parent = this;
    results.Location = new Point(10, 85);
    results.Size = new Size(360, 18 * Font.Height);

    Button sendit = new Button();
    sendit.Parent = this;
    sendit.Text = "Send";
    sendit.Location = new Point(220, 52);
    sendit.Size = new Size(5 * Font.Height, 2 * Font.Height);
    sendit.Click += new EventHandler(ButtonSendOnClick);

    Button closeit = new Button();
    closeit.Parent = this;
    closeit.Text = "Close";
    closeit.Location = new Point(290, 52);
    closeit.Size = new Size(5 * Font.Height, 2 * Font.Height);
    closeit.Click += new EventHandler(ButtonCloseOnClick);

    sock = new Socket(AddressFamily.InterNetwork, SocketType.Dgram,
        ProtocolType.Udp);
    IPEndPoint iep = new IPEndPoint(IPAddress.Any, 9050);
    sock.Bind(iep);
    sock.SetSocketOption(SocketOptionLevel.IP, SocketOptionName.AddMembership,
        new MulticastOption(IPAddress.Parse("224.100.0.1")));
    receiver = new Thread(new ThreadStart(packetReceive));
    receiver.IsBackground = true;
    receiver.Start();
}
```

```

void ButtonSendOnClick(object obj, EventArgs ea)
{
    byte[] message = Encoding.ASCII.GetBytes(newText.Text);
    newText.Clear();
    sock.SendTo(message, SocketFlags.None, multiep);
}

void ButtonCloseOnClick(object obj, EventArgs ea)
{
    receiver.Abort();
    sock.Close();
    Close();
}

void packetReceive()
{
    EndPoint ep = (EndPoint)multiep;
    byte[] data = new byte[1024];
    string stringData;
    int recv;
    while (true)
    {
        recv = sock.ReceiveFrom(data, ref ep);
        stringData = Encoding.ASCII.GetString(data, 0, recv);
        results.Items.Add("from " + ep.ToString() + ": " + stringData);
    }
}

public static void Main()
{
    Application.Run(new MulticastChat());
}
}

```

MulticastChat程序与本书中的其他聊天程序不同。最大的不同之处在于，该方法创建一个**Socket**对象在224.100.0.1组播组地址上接收和发送数据：

```

sock = new Socket(AddressFamily.InterNetwork,
    SocketType.Dgram, ProtocolType.Udp);
IPEndPoint iep = new IPEndPoint(IPAddress.Any, 9050);
sock.Bind(iep);
sock.SetSocketOption(SocketOptionLevel.IP, SocketOptionName.AddMembership,
    new MulticastOption(IPAddress.Parse("224.100.0.1")));

```

因为套接字使用**SetSocketOption()**方法加入组播组，因此必须使用**Bind()**方法绑定到指定的端口。为了简便，最好绑定到组播组使用的同一个端口。

Thread对象通常使用**packetReceive()**方法在组播组地址上收听消息。**packetReceive()**方法进入循环，等待组播组的消息，并将消息显示在**ListBox**对象中：

```
void packetReceive()
{
    EndPoint ep = (EndPoint)multip;
    byte[] data = new byte[1024];
    string stringData;
    int recv;
    while (true)
    {
        recv = sock.ReceiveFrom(data, ref ep);
        stringData = Encoding.ASCII.GetString(data, 0, recv);
        results.Items.Add("from " + ep.ToString() + ": " + stringData);
    }
}
```

因为组播聊天程序是一个非连接应用程序，用户必须告诉程序何时停止和关闭套接字。当单击关闭按钮，接收线程将被关闭，创建的套接字将被关闭，窗口窗体将被关闭：

```
void ButtonCloseOnClick(object obj, EventArgs ea)
{
    receiver.Abort();
    sock.Close();
    Close();
}
```

测试组播聊天程序

由于该程序是**Windows**窗体程序，当用户编译**MulticastChat.cs**时，可能会希望使用**/t**编译器操作，使得该程序可以在**Windows**中执行：

```
csc /t:winexe MulticastChat.cs
```

在编译结束后，用户可以通过双击**Windows**资源管理器窗口中的程序名称或者从命令提示符窗口中运行该程序。

警告：因为**MulticastChat**程序绑定到指定的**UDP**端口，因此一个程序实体只能在一台机器上执行。

当程序启动，聊天窗口将显示，并且准备接收从组播组发送的消息。要检查结果，用户可以首先监视在**netstat**输出中是否有**UDP**端口：

```
C:\>netstat -a
```

```
Active Connections
```

Proto	Local Address	Foreign Address	State
.			

```
.
UDP    abednego:9050          *: *
.
.
C:\>
```

毫无疑问，系统在UDP 9050号端口创建了套接字。用户可以使用WinDump或者Analyzer来监视由MulticastChat程序产生的网络传输。在程序启动时，用户可以看到程序发送的标准IGMP数据包：

```
C:\>windump igmp
windump: listening on \Device\Packet_E190x1
07:24:09.950210 192.168.1.6 > 224.100.0.1: 192.168.1.6 > 224.100.0.1: igmp v2
report 224.100.0.1 [ttl 1]
07:24:14.799582 192.168.1.6 > 224.100.0.1: 192.168.1.6 > 224.100.0.1: igmp v2
report 224.100.0.1 [ttl 1]

1237 packets received by filter
0 packets dropped by kernel

C:\>
```

在程序启动后，用户可以在文本框中输入文本消息，然后单击发送按钮。因为应用程序本身收听组播地址，消息将在列表框中显示。如果用户使用的网络具有多个Windows设备，用户可以通过同时多个设备上运行应用程序来进行试验。应用程序的每个实体将显示子网上其他设备发送的消息。

说明：记住，如果用户网络使用了路由器，必须对路由器进行配置，允许组播进行前向传送，以便程序可以通过子网工作。

小结

本章讨论了可以同时与多个网络主机进行通信的技术，该技术使用广播和组播数据包。因为IP广播和组播是无连接的通信，因此这两种方法只能使用UDP套接字。

C#编程语言通过提供SetSocketOption()方法的Broadcast选项支持广播包。在默认情况下，任何套接字都被允许接收从默认的UDP端口接收的广播消息。然而，要发送广播消息，SetSocketOption()方法必须允许套接字使用Broadcast选项。

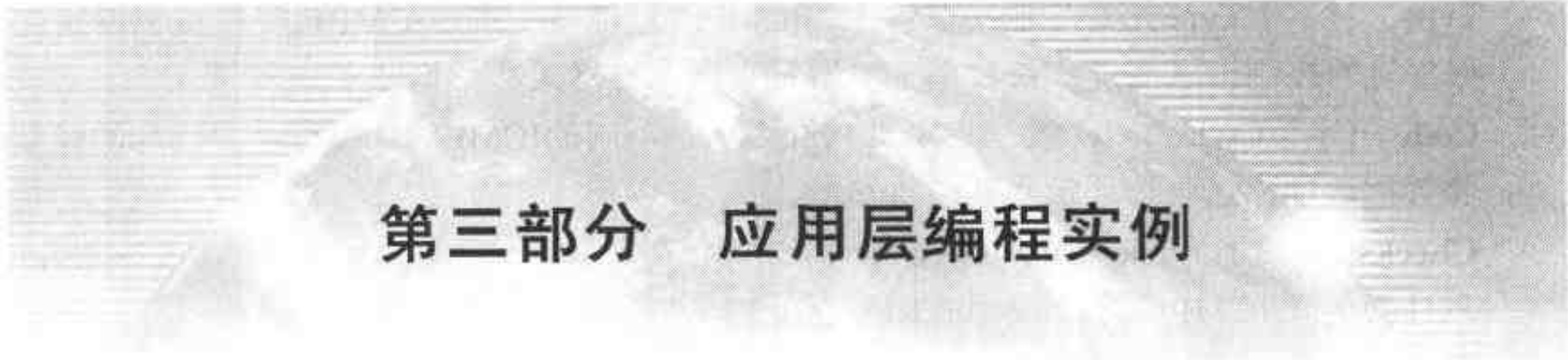
很多服务器应用程序采用固定的时间间隔广播服务器服务。客户机可以在网络上检测服务器的情况，并且知道如何与服务器进行通信。这种广播通常使用后台线程，在设置的时间间隔持续循环重复广播消息。

C#编程语言同样通过提供SetSocketOption()方法的MulticastOption类支持IP组播组。AddMembership和DropMembership套接字选项允许套接字加入或者离开组播组。MulticastOption类定义组播组的IP地址。一旦套接字加入组播组，套接字将接收到发送到组

播组IP地址的任何消息。类似地，UdpClient类提供JoinMulticastGroup()和DropMulticastGroup()方法。这些方法为UdpClient对象提供加入或者离开组播组的功能。

C#网络应用程序可以利用组播组与选择的网络设备子集进行通信。本章中的简单网络聊天程序将示范在不发送数据包副本的情况下同时与网络上的多个设备进行通信。

本章是第二部分的一章，在本章中，用户已经学习了很多C#环境中网络层编程的技术。在第三部分的其他章节中，将介绍使用C#的网络应用程序例子。其中，前面的两章，第11章“ICMP”和第12章“SNMP”，将介绍如何使用原始的套接字与远程的网络设备进行通信。其余的章节将讨论应用层编程，例如SMTP和HTTP，以及在网络程序中使用Active Directory（活动目录）、remoting（远程技术）和安全功能。



第三部分 应用层编程实例

第11章 ICMP

- ICMP协议
- 使用raw套接字
- 创建C# ICMP类
- 简单的ping程序
- 高级的ping程序
- traceroute程序
- 发现子网掩码程序

本书前面部分阐述了使用C#编程语言进行网络编程的基础。在第三部分中，用户将利用C#网络类创建一些真正的应用。本章将带领用户使用因特网控制消息协议（ICMP）创建一些网络工具，并且与网络上的远程主机进行通信。很多流行的网络工具，例如ping和traceroute，都是基于ICMP的。

在讨论ICMP后，用户将创建一个单独的ICMP类，示范如何在类中包含一个协议。最后，将利用三个单独的ICMP工具程序来示范如何在C#网络程序中使用ICMP类。

ICMP协议

ICMP定义在RFC 792中，允许网络设备在数据包处理中报告错误。自从该协议创建以来，ICMP已经增加了一些功能，可以更好地适应通信手段的错误和主机间的网络信息。

ICMP使用IP在网络间进行通信。尽管使用了IP，ICMP还是一个与TCP或者UDP独立的协议。如在第2章“IP编程基础”中阐述的，IP包可以使用协议Type字段来识别包含在数据段内的下一层协议。图11.1显示了放置在IP包中的ICMP包区域。

ICMP包格式

与TCP和UDP类似，ICMP使用了特殊的包格式来识别在包中的信息。如图11.1所示，ICMP包包含以下区域：

Type 1字节Type元素，定义在包中的ICMP消息种类。很多ICMP包类型用于发送远程主机的控制请求消息。每种消息类型具有自己的格式和数据要求。

Code 1字节Code元素，进一步定义Type区域。不同的ICMP消息类型需要特殊的控制和数据选项。这些选项定义在Code区域。

Checksum 2字节Checksum元素，确保ICMP包在没有被破坏的情况下到达。Checksum只计算包的ICMP部分，使用在RFC 792中定义的特殊算法。当计算校验和值时，Checksum区域将被设置为0。

Message 多字节Message元素包含不同的数据元素，并且每种ICMP消息类型都是唯一的。Message数据字段通常用于包含发送或者发自远程主机的信息。很多ICMP消息类型定义在Message元素中的头两个区域作为标识符和序列号。对于主机来说，这些区域都是ICMP的惟一标识。

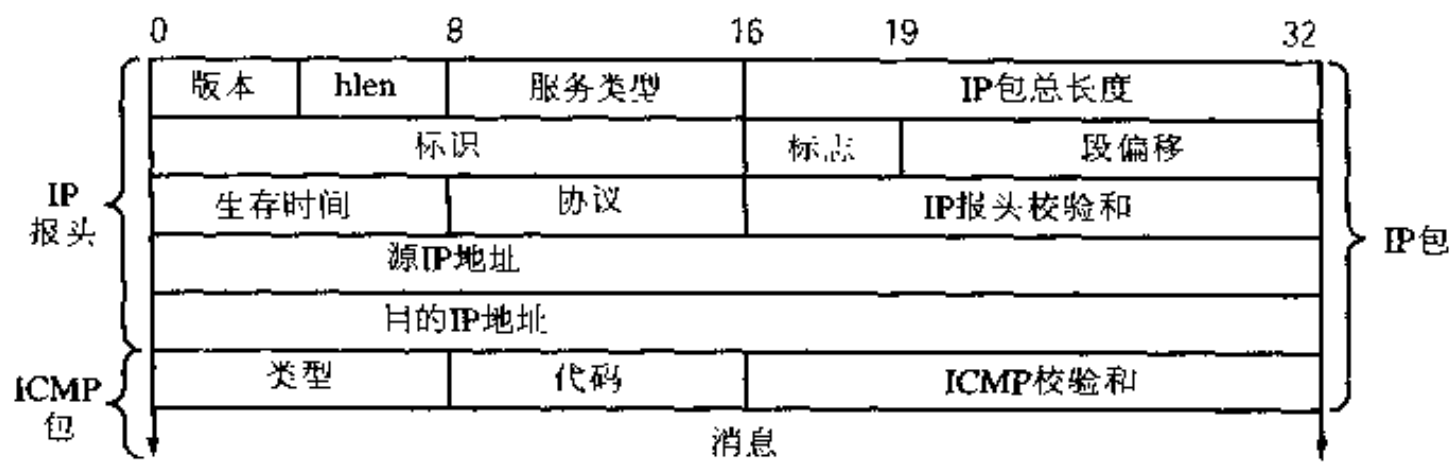


图11.1 IP和ICMP包格式

ICMP包类型

ICMP包具有很多类型。每种类型的ICMP包都是在Type元素的1字节值中定义的。表11.1列出了ICMP类型，是RFC 792中定义的原始类型。

表11.1 ICMP包类型

类型代码	说明
0	回送回复
3	目的地无法到达
4	源消除
5	重定向
8	回送请求
11	超时
12	参数问题
13	时间印章请求
14	时间印章回复
15	信息请求
16	信息回复

自从1981年9月发布RFC 792以来，创建了很多新的ICMP类型。ICMP包与简单回送测试对复杂网络错误报告的内容有关。下面的描述是ICMP包的一些通用用途。

回送请求和回送回复包

ICMP包的两个最基本用途就是回送请求和回送回复。包允许设备从网络上的远程设备请求一个ICMP响应——ping工具的核心已经变为网络管理的通用工具。

回送请求包使用ICMP Type 8，代码值为0。Message数据区包含三个元素：

- 1字节的标识符，Echo Request包的惟一标识符。
- 1字节的序列号，提供ping包中ICMP包的附加标识。
- 多字节数据元素，包含接收主机返回的数据。

当设备接收到Echo Request包，设备必须使用Echo Reply包进行响应，ICMP Type0。Echo Reply包必须包含同样的标识符和序列号码值，作为Echo Request包进行响应。同样的，数据元素值一定与Echo Request包接收的相同。

目的地不可达包

目的地不可达ICMP包（Type 3）通常由路由设备在接收到无法达到目的地的IP包后返回。目的地不可达包的数据部分包含IP标题和数据包的前64位。

在这个包中，代码区表示包无法被路由器转发的原因。表11.2是一个可能会遇到的代码值列表。

表11.2 目的地不可达代码值

代码	说明
0	网络不可到达
1	主机不可到达
2	协议不可到达
3	端口不可到达
4	需要分段存储和DF标志设置
5	源路由失败
6	目的网络不可知
7	目的地主机不可知
8	源主机隔离
9	禁止与目的网络通信
10	禁止与目的主机通信
11	服务类型网络不可到达
12	服务类型主机不可到达

说明：RFC 792，描述ICMP的原始的RFC，只定义了目的地不可达代码0~5。后来RFC得到了扩展，定义了表11.2中的13个代码。

超时包

超时（Time Exceeded）包已经成为了一个重要的工具，可以用于网络故障排除。该包报告一个IP包已经超过了在IP头中定义的寿命时间值（TTL）。

当IP包在网络路由器间传递一次，TTL值将被减1。如果IP包在到达真正的目的地前TTL值为0，最后接收到该IP包的路由器将发送超时ICMP包到发送的主机。如用户所见，tracert程序将使用该方法。

使用Raw套接字

因为ICMP包既不使用TCP，也不使用UDP，因此用户无法使用套接字助手类TcpClient或者UdpClient。相反，用户必须使用Raw套接字，它执行Socket类的功能。Raw套接字允许用户在IP层上定义自己的网络包。当然，这意味着用户必须进行手工工作，创建在ICMP包中的所有单独区域，而不是使用.NET库创建包，如TCP和UDP一样。

Raw套接字格式

要创建一个Raw套接字，用户必须使用SocketType.Raw类型。这里有几个ProtocolType值，用户可以用于匹配Raw套接字类型；这些值列在表11.3中。

表11.3 Raw套接字ProtocolType（协议类型）值

值	说明
Ggp	网关到网关协议
Icmp	因特网控制消息协议
Idp	IDP协议
Igmp	因特网组管理协议
IP	原始的IP包
Ipx	Novell IPX协议
ND	网络磁盘协议
Pup	Xerox PARC通用协议
Raw	原始的IP包
Spx	Novell SPX协议
SpxII	Novell SPX第二版协议
Unknown	无法确认的协议
Unspecified	未指明的协议

列出的特殊协议允许.NET库创建IP包。通过使用ProtocolType.Icmp值，由套接字定义的IP包将下一层协议定义为ICMP（IP值Type 1）。这样就允许远程主机立即将包认证为ICMP包，从而进行处理。

很明显，对于本章中的应用程序，用户可以使用ProtocolType.Icmp值。这里是创建ICMP包套接字的命令：

```
Socket sock = new Socket(AddressFamily.InterNetwork, SocketType.Raw,
    ProtocolType.Icmp);
```

发送Raw包

因为ICMP是一个无连接协议，用户无需绑定套接字到指定的本地端口发送包，或者使用Connect()方法连接到指定的远程主机。用户必须使用SendTo()方法来指定目的地地址的IPEndPoint对象。下面的例子创建了一个没有端口的IPEndPoint目的地对象，并对该对象发送包：

```
IPEndPoint iep = new IPEndPoint(IPAddress.Parse("192.168.1.2"), 0);
sock.SendTo(packet, iep);
```

警告：记住，因为原始套接字没有格式化数据，在包字节数组中输入的任何值都将被转发“as-is”到远程主机。这意味着用户在字节数组中必须手动创建ICMP包，并且发送到远程主机。在创建包的过程中，任何错误都将导致接收主机产生错误，并且用户将无法得到返回的ICMP包。

接收Raw包

从Raw套接字中接收数据要比在Raw套接字中发送数据更为灵活。要接收Raw套接字中的数据，用户必须使用ReceiveFrom()方法。因为Raw套接字无法识别高级别的协议，从ReceiveFrom()方法调用中返回的数据包含全部的IP包内容，用户必须从Raw IP包中提取数据信息，来创建ICMP包元素。如用户在图11.1中所见，IP包数据起始于字节20。因此，要提取ICMP包数据和标题，用户必须在接收到的数据包的第20个字节位置开始读取字节数组。

警告：因为ReceiveFrom()方法返回整个IP包，用户必须记住在声明接收缓存尺寸时，至少要比预期接收的数据大20个字节。

创建一个ICMP类

如刚才提到的，Raw套接字无法自动格式化用户的ICMP包，所以用户必须自己完成这项工作。C#是一种面向对象的语言。要创建C# ICMP类，用户可以格式化一个ICMP包，并且对包的内容进行操作。这样就允许用户在使用了ICMP包的网络应用程序中使用ICMP类。

ICMP类构造函数

ICMP类将对在ICMP包中的每个元素定义一个数据变量。表11.4显示了用于定义普通ICMP包的数据变量。

表11.4 ICMP类普通数据元素

数据变量	尺寸	类型
Type（类型）	1字节	字节
Code（代码）	1字节	字节
Checksum（校验和）	2字节	无符号16位整数
Message（消息）	多字节	字节数组

默认构造函数

默认的ICMP构造函数创建了一个ICMP实体，但是并不为数据变量分配任何值。在ICMP应用程序中，当用户准备创建一个ICMP包时，可以完成赋值工作。

这里是ICMP类默认构造函数的格式：

```
class ICMP
{
    public byte Type;
    public byte Code;
    public UInt16 Checksum;
    public int MessageSize;
    public byte[] Message = new byte[1024];
    public ICMP()
    {
    }
}
```

尽管这不是ICMP包的部分，但还是要将MessageSize变量添加进去，描绘可变长度变量Message的实际尺寸。当用户看到这些时，将会明白这些内容的重要性。

要构造一个新的ICMP包，应用程序需要创建一个新的ICMP对象，并且为数据元素分配适当的值，如下：

```
ICMP packet = new ICMP();
packet.Type = 0x08;
packet.Code = 0x00;
packet.Checksum = 0;
```

这个代码段显示了创建Echo Request ICMP包的第--部分的例子。Echo Request包示例了一个通用的场景，当创建ICMP包时，用户将会深入。因为Echo Request包在ICMP Message元素中定义区域（标识符和序号区域），用户必须确定如何进一步创建Message元素。要完成此项工作，用户具有两种选择：

- 创建其他的类，用于惟一的Echo Request区域，并且获取类的字节数组。
 - 将Message区域分别转换到字节数组，并且放置在数据元素中。
- 如果用户需要创建大量的Echo Request包，可以通过创建一个Ping类来定义Echo Re-

quest区域。否则，对于少量的Echo Request包，将值转换到单独字节数组并加载到Message元素将不是什么大的工作：

```
Buffer.BlockCopy(BitConverter.GetBytes((short)1), 0, packet.Message, 0, 2);
Buffer.BlockCopy(BitConverter.GetBytes((short)1), 0, packet.Message, 2, 2);
byte[] data = Encoding.ASCII.GetBytes("test packet");
Buffer.BlockCopy(data, 0, packet.Message, 4, data.Length);
packet.MessageSize = data.Length + 4;
```

这里是Echo Reply包的标识区域和序号区域，被转换到字节数组，并被放置在Message元素的适当位置。

重新创建ICMP对象

在发送ICMP包后，用户会从远程设备返回的数据中得到ICMP包。要轻松地对包中的内容进行译码，用户需要创建其他的ICMP类构造函数，可以对Raw ICMP字节数组进行操作，并且将值放入类中的适当数据元素：

```
public ICMP(byte[] data, int size)
{
    Type = data[20];
    Code = data[21];
    Checksum = BitConverter.ToUInt16(data, 22);
    MessageSize = size - 24;
    Buffer.BlockCopy(data, 24, Message, 0, MessageSize);
}
```

记住，Raw套接字返回整个IP包。这意味着在提取ICMP包信息前，用户必须跳过IP头信息。要启动ICMP信息和Type元素，要从字节数组的第20位开始操作。在ICMP包中的单独数据元素将被逐个字节地提取到适当的ICMP元素。

在使用接收的包数据创建新的ICMP对象后，用户可以分别引用数据元素：

```
int recv = ReceiveFrom(data, ref ep);
ICMP response = new ICMP(data, recv);
Console.WriteLine("Received ICMP packet:");
Console.WriteLine("  Type {0}", response.Type);
Console.WriteLine("  Code: {0}", response.Code);
Int16 Identifier = BitConverter.ToInt16(response.Message, 0);
Int16 Sequence = BitConverter.ToInt16(response.Message, 2);
Console.WriteLine("  Identifier: {0}", Identifier);
Console.WriteLine("  Sequence: {0}", Sequence);
stringData = Encoding.ASCII.GetString(response.Message, 4,
    response.MessageSize - 4);
Console.WriteLine("  data: {0}", stringData);
```

获取接收到的ICMP包的前两个数据元素是非常简单的，用户需要做的就是从ICMP类元

素中读取它们的值。从Message元素中提取数据区域有一点复杂。因为前2个字节组成无符号的整形标识符区域，并且下2个字节是无符号的整形序号区域，用户可以使用BitConverter从适当的字节中分配这些值。Message元素的剩余部分将分配到Echo Reply包的Message区域。

通过这个代码段，可以轻松地理解为什么MessageSize数据元素被加载到ICMP类。如果没有该类，将很难从接收的包中构造Message元素。

ICMP包创建器

当创建了新的ICMP对象后，包数据元素将被定义，用户将发送包到远程网络设备。很可惜，用户无法直接在SendTo()方法中发送ICMP对象；必须转变到字节数组中操作。

如在第7章“使用C#套接字助手类”中描述的，通过网络发送复杂类的最简单方法就是创建一个方法，将每个数据元素转变为字节数组，并且将字节数组构造到单独的大字节数组。这项工作是使用Buffer.BlockCopy()方法完成的：

```
public byte[] getBytes()
{
    byte[] data = new byte[MessageSize + 9];
    Buffer.BlockCopy(BitConverter.GetBytes(Type), 0, data, 0, 1);
    Buffer.BlockCopy(BitConverter.GetBytes(Code), 0, data, 1, 1);
    Buffer.BlockCopy(BitConverter.GetBytes(Checksum), 0, data, 2, 2);
    Buffer.BlockCopy(Message, 0, data, 4, MessageSize);
    return data;
}
```

让我们看看发生了什么事情。数据字节数组是通过最新转换的ICMP对象数据元素创建的。通过使用BitConverter类方法，每个数据元素被转为字节数组，并且放置在数据数组中，用于getBytes()方法。当所有的数据元素被转变为字节数组，数组将包含适当的格式化ICMP包，被发送到远程网络设备：

```
IPEndPoint iep = new IPEndPoint(IPAddress.Parse("192.168.1.2"), 0);
sock.SendTo(packet.getBytes(), iep);
```

记住，因为ICMP不使用端口，当为目的地地址创建IPEndPoint时，用户可以将端口参数设置为0。

说明：用户可能已经注意到，在被放置到字节数组前，标识符和序号区域没有被转换为网络字节顺序。这是因为远程设备将返回和用户译码相同的包，当用户得到返回的包时，这些包将是同样的格式。

ICMP校验和方法

在创建ICMP包时，最可怕的部分就是计算包的校验和值（除非用户喜欢二进制计算）。完成这项工作最简便的方法就是创建一个独立的方法，用于计算校验和并放置在ICMP类中，以便ICMP应用程序使用。

ICMP RFC定义Checksum元素作为“计算ICMP消息和的一个16位部件，使用ICMP类型开始。为了计算校验和，Checksum元素将被设置为“0”。

幸运的是，有一些可以使用的校验和工具。程序清单11.1显示了我选择的一种校验和方法，在ICMP类中执行。

程序清单11.1 getChecksum() ICMP方法

```
public UInt16 getChecksum()
{
    UInt32 chcksm = 0;

    byte[] data = getBytes();
    int packetsize = MessageSize + 8;

    int index = 0;

    while ( index < packetsize)
    {
        chcksm += Convert.ToUInt32(BitConverter.ToUInt16(data, index));
        index += 2;
    }

    chcksm = (chcksm >> 16) + (chcksm & 0xffff);
    chcksm += (chcksm >> 16);
    return (UInt16)(~chcksm);
}
```

因为ICMP校验和值使用16位的运算，该算法可以同时读取2字节的ICMP包片段（使用BitConverter类的ToUInt16()方法），并且执行必要的运算。返回值是一个16位的无符号整形值。

要在ICMP应用程序中使用校验和值，首先要填充所有的数据元素，设置Checksum元素为0。接下来，调用getChecksum()方法来计算ICMP包的校验和值，然后将结果放置在包的Checksum元素中：

```
packet.Checksum = 0;
packet.Checksum = packet.getChecksum();
```

当Checksum元素计算后，包准备使用SendTo()方法发送到目的地主机。

说明：在产品应用程序中，当接收到远程主机的ICMP包，用户需要提取Checksum元素值，并且与对包的计算值进行比较。如果两个值不匹配，说明发生了错误，并且包将被重新传输。

各种元素一起工作

现在，用户已经看到了ICMP类的所有单独元素，我们将观察在C#代码中定义的全部类。程序清单11.2是ICMP.cs程序，该程序执行ICMP类，帮助用户创建并且读取ICMP包。

程序清单11.2 ICMP.cs程序

```
using System;
using System.Net;
using System.Text;

class ICMP
{
    public byte Type;
    public byte Code;
    public UInt16 Checksum;
    public int MessageSize;
    public byte[] Message = new byte[1024];

    public ICMP()
    {
    }

    public ICMP(byte[] data, int size)
    {
        Type = data[20];
        Code = data[21];
        Checksum = BitConverter.ToUInt16(data, 22);
        MessageSize = size - 24;
        Buffer.BlockCopy(data, 24, Message, 0, MessageSize);
    }

    public byte[] getBytes()
    {
        byte[] data = new byte[MessageSize + 9];
        Buffer.BlockCopy(BitConverter.GetBytes(Type), 0, data, 0, 1);
        Buffer.BlockCopy(BitConverter.GetBytes(Code), 0, data, 1, 1);
        Buffer.BlockCopy(BitConverter.GetBytes(Checksum), 0, data, 2, 2);
        Buffer.BlockCopy(Message, 0, data, 4, MessageSize);
        return data;
    }

    public UInt16 getChecksum()
    {
        UInt32 chcksm = 0;
        byte[] data = getBytes();
        int packetSize = MessageSize + 8;

        int index = 0;

        while (index < packetSize)
        {
            chcksm += Convert.ToUInt32(BitConverter.ToUInt16(data, index));
            index += 2;
        }
    }
}
```



```

    }
    checksum = (checksum >> 16) + (checksum & 0xffff);
    checksum += (checksum >> 16);
    return (UInt16)(~checksum);
}
}

```

要记住：不要使用ICMP.cs类本身进行编译。该类不包含Main()方法，因此无法创建可执行程序。相反，该类支持ICMP网络应用程序，允许用户轻松地创建、发送和解释ICMP包。程序的剩余部分使用这个类来创建ICMP包。

一个简单的Ping程序

第一个讨论的ICMP应用程序是一个简单的ping程序，几乎可以在所有网络设备上发现。Ping是一个重要和基本的诊断工具，告诉用户网络设备是否连接到网络，并且连接到网络上的远程设备。Ping程序使用ICMP Echo Request包（Type 8）向远程主机发送简单的消息。当远程主机接收到消息，该主机将使用ICMP Echo Reply包进行应答（Type 0），该包包含原始消息。图11.2显示了这个过程。

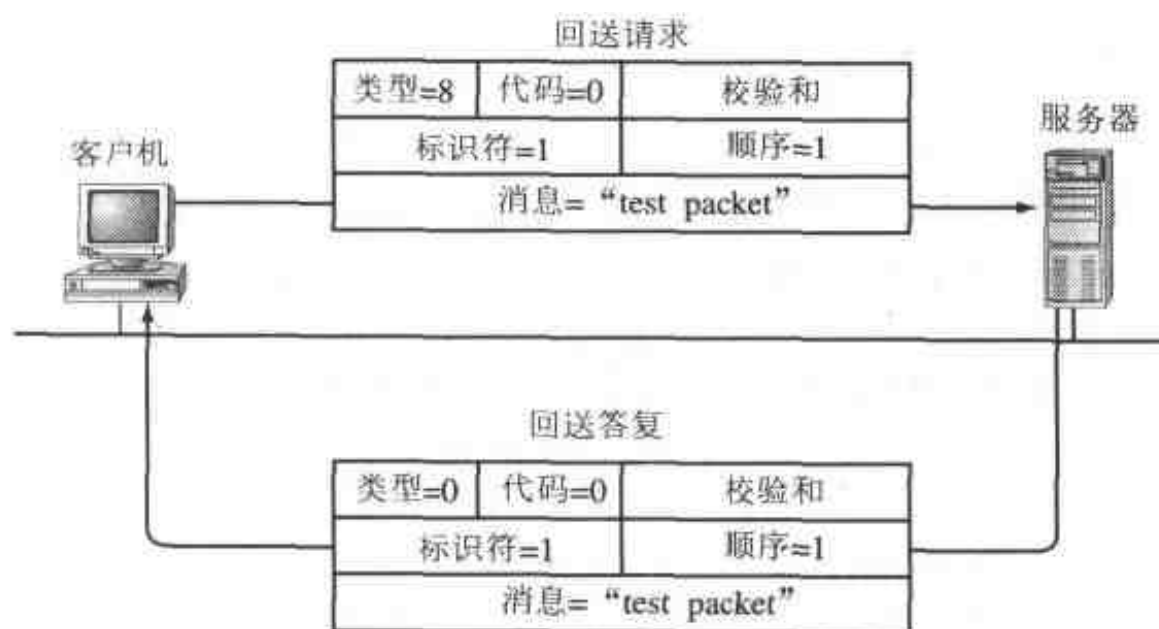


图11.2 在ping程序后台的ICMP控制消息

因为用户现在具有ICMP类，执行简单的ping程序非常简单。程序清单11.3是SimplePing.cs程序，该程序使用ICMP与网络上的远程主机进行ping操作。

程序清单11.3 SimplePing.cs程序

```

using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class SimplePing

```

```

{
    public static void Main (string[] argv)
    {
        byte[] data = new byte[1024];
        int recv;
        Socket host = new Socket(AddressFamily.InterNetwork, SocketType.Raw,
                                   ProtocolType.Icmp);
        IPEndPoint iep = new IPEndPoint(IPAddress.Parse(argv[0]), 0);
        EndPoint ep = (EndPoint)iep;
        ICMP packet = new ICMP();

        packet.Type = 0x08;
        packet.Code = 0x00;
        packet.Checksum = 0;
        Buffer.BlockCopy(
            BitConverter.GetBytes((short)1), 0, packet.Message, 0, 2);
        Buffer.BlockCopy(
            BitConverter.GetBytes((short)1), 0, packet.Message, 2, 2);
        data = Encoding.ASCII.GetBytes("test packet");
        Buffer.BlockCopy(data, 0, packet.Message, 4, data.Length);
        packet.MessageSize = data.Length + 4;
        int packetsize = packet.MessageSize + 4;

        UInt16 chcksum = packet.getChecksum();
        packet.Checksum = chcksum;

        host.SetSocketOption(SocketOptionLevel.Socket,
                               SocketOptionName.ReceiveTimeout, 3000);
        host.SendTo(packet.getBytes(), packetsize, SocketFlags.None, iep);
        try
        {
            data = new byte[1024];
            recv = host.ReceiveFrom(data, ref ep);
        } catch (SocketException)
        {
            Console.WriteLine("No response from remote host");
            return;
        }
        ICMP response = new ICMP(data, recv);
        Console.WriteLine("response from: {0}", ep.ToString());
        Console.WriteLine("  Type {0}", response.Type);
        Console.WriteLine("  Code: {0}", response.Code);
        int Identifier = BitConverter.ToInt16(response.Message, 0);
        int Sequence = BitConverter.ToInt16(response.Message, 2);
        Console.WriteLine("  Identifier: {0}", Identifier);
        Console.WriteLine("  Sequence: {0}", Sequence);
    }
}

```

```
        string stringData = Encoding.ASCII.GetString(response.Message,
            4, response.MessageSize - 4);
        Console.WriteLine("    data: {0}", stringData);

        host.Close();
    }
}
```

这个简单的程序不包含任何与真正ping程序不同的报警符号。对于发起者，SimplePing要求在程序命令行上使用IP地址（如果愿意，用户可以使用Dns.Resolve()方法来代替IPAddress.Parse()方法，允许主机名称替代IP地址）。

在程序的第一部分，创建一个ICMP包，定义ICMP类型元素为8和代码元素0。该操作创建一个Echo Request包，使用标识符和序号元素来跟踪单独的ping包，并且允许用户在数据元素中输入任何文本。

类似无连接的UDP程序，超时值用于使用SetSocketOption()方法的套接字。如果在3秒钟内没有从远程主机接收到ICMP包，将导致异常并且程序退出。

返回的ICMP包（如果其中一个是返回的）创建一个新的ICMP对象，该对象用于确定接收到的包是否与发送的ICMP包匹配。接收到的包的标识符、序号和数据元素应该与发送ICMP包相同。如果不同，用户需要拦截同一设备上的其他应用程序的ICMP包，并且需要收听返回的ICMP包。

要编译SimplePing程序，用户必须在csc命令行中包含ICMP.cs程序：

```
csc SimplePing.cs ICMP.cs
```

因为Main()方法定义在SimplePing.cs程序中，编译器创建一个SimplePing.exe可执行文件来执行该程序。

测试SimplePing

用户可以通过启动命令行窗口，并且ping本地设备地址来检测SimplePing程序。输出显示，用户成功地完成了ping操作：

```
C:\>SimplePing 127.0.0.1
response from: 127.0.0.1:0
    Type 0
    Code: 0
    Identifier: 1
    Sequence: 1
    data: test packet

C:\>
```

接下来的测试是与一个不存在的IP地址进行ping操作。这样将强迫套接字超时功能启动，并且产生一个异常：

```
C:\>SimplePing 192.168.1.111
No response from remote host

C:\>
```

到此为止，非常好。最后的测试将确保ping包可以在网络上成功传输。用户可以通过与因特网或者本地网络的远程设备进行ping操作进行测试：

```
C:\>SimplePing 192.168.1.2
response from: 192.168.1.2:0
  Type 0
  Code: 0
  Identifier: 1
  Sequence: 1
  data: test packet

C:\>
```

警告：在高度安全网络的时代，如果与远程设备ping操作失败，不要感到惊奇。很多站点使用防火墙防止收到ping攻击。

用户可以通过使用WinDump或者Analyzer来观察ping包的发送和返回。每个ping包显示网络上ICMP包的轨迹。

高级的Ping程序

SimplePing的设计目的是说明使用ICMP Echo Request/Reply序列与远程主机进行ping操作的基础。如果用户使用商用的ping产品，用户将知道一些在这里无法执行的功能。SimplePing具有以下限制：

- 目的地地址必须为IP地址。
- 只能发送和接收一个ping消息。
- Ping数据的尺寸无法进行配置。
- Ping的处理时间无法被记录。

本节显示了如何创建一个更先进的ping程序，消除这些功能限制。

AdvPing.cs是一个基于Windows的ping程序，可以执行一些更为复杂的ping功能。

要允许用户在查询文本框中输入IP地址或者主机名称，需要使用Dns.Resolve()方法。在第4章“DNS和C#”中，我们讲述了Dns.Resolve()方法，该方法可以将主机名称或者文本的IP地址分解到IPHostEntry数组。IPHostEntry数组包含所有与主机名称或IP地址关联的IP地址。因为ping程序只需要一个地址，因此会选用列表中的第一个地址：

```
IPHostEntry iphe = Dns.Resolve(hostbox.Text);
IPEndPoint iep = new IPEndPoint(iphe.AddressList[0], 0);
```

用户可能会希望多个ping包，所以AdvPing使用单独的线程来处理实际的ping功能。当

客户单击Start按钮，将创建一个新的线程。新线程包含一个循环，可以持续与远程主机进行ping操作，直到线程被中断。要完成这项工作，客户接口包含一个按钮，允许客户停止ping操作。

程序发送的每个单独的ping ICMP包都被分配一个连续的序列号码，以便区别单独的ping。因为每个ping包都使用不同的序号，ICMP校验和值将会为每个包而重新计算。

程序清单11.4是AdvPing.cs程序。

程序清单11.4 AdvPing.cs程序

```
using System;
using System.Drawing;
using System.Net;
using System.Net.Sockets;
using System.Text;
using System.Threading;
using System.Windows.Forms;

class AdvPing Form
{
    private static int pingstart, pingstop, elapsedtime;
    private static TextBox hostbox, databox;
    private static ListBox results;
    private static Thread pinger;
    private static Socket sock;

    public AdvPing()
    {
        Text = "Advanced Ping Program";
        Size = new Size(400, 380);

        Label label1 = new Label();
        label1.Parent = this;
        label1.Text = "Enter host to ping:";
        label1.AutoSize = true;
        label1.Location = new Point(10, 30);

        hostbox = new TextBox();
        hostbox.Parent = this;
        hostbox.Size = new Size(200, 2 * Font.Height);
        hostbox.Location = new Point(10, 55);

        results = new ListBox();
        results.Parent = this;
        results.Location = new Point(10, 85);
        results.Size = new Size(360, 18 * Font.Height);

        Label label2 = new Label();
```

```
label2.Parent = this;
label2.Text = "Packet data:";
label2.AutoSize = true;
label2.Location = new Point(10, 330);

databox = new TextBox();
databox.Parent = this;
databox.Text = "test packet";
databox.Size = new Size(200, 2 * Font.Height);
databox.Location = new Point(80, 325);

Button sendit = new Button();
sendit.Parent = this;
sendit.Text = "Start";
sendit.Location = new Point(220, 52);
sendit.Size = new Size(5 * Font.Height, 2 * Font.Height);
sendit.Click += new EventHandler(ButtonSendOnClick);

Button stopit = new Button();
stopit.Parent = this;
stopit.Text = "Stop";
stopit.Location = new Point(295, 52);
stopit.Size = new Size(5 * Font.Height, 2 * Font.Height);
stopit.Click += new EventHandler(ButtonStopOnClick);

Button closeit = new Button();
closeit.Parent = this;
closeit.Text = "Close";
closeit.Location = new Point(300, 320);
closeit.Size = new Size(5 * Font.Height, 2 * Font.Height);
closeit.Click += new EventHandler(ButtonCloseOnClick);

sock = new Socket(AddressFamily.InterNetwork,
    SocketType.Raw, ProtocolType.Icmp);
sock.SetSocketOption(SocketOptionLevel.Socket,
    SocketOptionName.ReceiveTimeout, 3000);
}

void ButtonSendOnClick(object obj, EventArgs ea)
{
    pinger = new Thread(new ThreadStart(sendPing));
    pinger.IsBackground = true;
    pinger.Start();
}

void ButtonStopOnClick(object obj, EventArgs ea)
{
    pinger.Abort();
}
```



```

        results.Items.Add("Ping stopped");
    }

    void ButtonCloseOnClick(object obj, EventArgs ea)
    {
        sock.Close();
        Close();
    }

    void sendPing()
    {
        IPHostEntry iphe = Dns.Resolve(hostbox.Text);
        IPEndPoint iep = new IPEndPoint(iphe.AddressList[0], 0);
        EndPoint ep = (EndPoint)iep;
        ICMP packet = new ICMP();
        int recv, i = 1;

        packet.Type = 0x08;
        packet.Code = 0x00;
        Buffer.BlockCopy(BitConverter.GetBytes(1), 0, packet.Message, 0, 2);
        byte[] data = Encoding.ASCII.GetBytes(databox.Text);
        Buffer.BlockCopy(data, 0, packet.Message, 4, data.Length);
        packet.MessageSize = data.Length + 4;
        int packetsize = packet.MessageSize + 4;

        results.Items.Add("Pinging " + hostbox.Text);
        while(true)
        {
            packet.Checksum = 0;
            Buffer.BlockCopy(BitConverter.GetBytes(i), 0, packet.Message, 2, 2);
            UInt16 chcksum = packet.getChecksum();
            packet.Checksum = chcksum;

            pingstart = Environment.TickCount;
            sock.SendTo(packet.getBytes(), packetsize, SocketFlags.None, iep);
            try
            {
                data = new byte[1024];
                recv = sock.ReceiveFrom(data, ref ep);
                pingstop = Environment.TickCount;
                elapsedtime = pingstop - pingstart;
                results.Items.Add("reply from: " + ep.ToString() + ", seq: " + i +
                    ", time = " + elapsedtime + "ms");
            } catch (SocketException)
            {
                results.Items.Add("no reply from host");
            }
        }
    }

```

```
        i++;  
        Thread.Sleep(3000);  
    }  
}  
  
public static void Main()  
{  
    Application.Run(new AdvPing());  
}  
}
```

在AdvPing类构造函数中，将创建Windows窗体。该窗体呈现给客户一个界面，可以输入主机名称或者IP地址进行ping操作，在ListBox对象中显示ping的结果，两个按钮可以启动或者结束ping线程。用于ICMP连接的Socket对象同样被建立，并且ReceiveTimeout值设置为3秒：

```
sock = new Socket(AddressFamily.InterNetwork,  
    SocketType.Raw, ProtocolType.Icmp);  
sock.SetSocketOption(SocketOptionLevel.Socket,  
    SocketOptionName.ReceiveTimeout, 3000);
```

当客户输入主机名称或者IP地址后，单击Start按钮，将创建一个新的线程来处理ping进程。sendPing()方法类似于SimplePing程序，不同的是，sendPing()将不停地执行循环，与远程目的地进行ping操作。

Environment.TickCount特性花费大量的时间跟踪发送和接收ping包。每个TickCount代表系统进行操作的时间长度（毫秒级）。通过扣除发送第一个包前的时间，程序可以确定ping进程所用的时间。

当客户单击Stop按钮，ping线程将被中断，但是套接字仍然开放。这样客户机就可以选择其他的主机名称或者IP地址，通过单击Start按钮进行ping操作。当单击Close按钮，Socket对象将被关闭，程序将结束。

类似SimplePing程序，AdvPing程序必须使用ICMP.cs类文件进行编译：

```
csc /t:winexe AdvPing.cs ICMP.cs
```

/t:winexe参数同样用于编译器命令行，创建Windows执行程序。

测试AdvPing

用户可以通过在Windows资源管理器中启动来测试AdvPing程序，或者通过打开命令提示符窗口并且运行程序。当程序启动，用户将看到客户界面（图11.3）。当输入主机名称或者IP地址和用户希望发送的文本，单击Start按钮启动ping操作。每个ping的尝试都会在中心列表框中被记录。



图11.3 AdvPing客户界面窗口

TraceRoute.cs程序

Ping的近亲是traceroute。traceroute程序发送ICMP Echo Request到远程主机，但是采用曲折的路由。要检测ICMP包经过了哪些路由器到达目的地，traceroute将利用另一种ICMP消息包。

通过增加IP包TTL值，traceroute程序可以强迫ICMP包在到达目的地的路径中死亡。当TTL值到期，最后一个接收到该包的网路路由器将发送ICMP Time Exceeded (Type 11) 包到发送的主机。通过将TTL设置为1时开始启动，并且每次尝试都增加1，traceroute程序强迫每个路径中的路由器返回一个ICMP Time Exceeded包。通过显示每个包的发送地址，用户可以观察ping包路径中的每个路由器。

traceroute操作可以通过使用SetSocketOption()方法调用来执行，并且IPTimeToLive套接字选项操作在IP包中的TTL值。再一次，最新创建的ICMP类将被使用，创建一个Echo Request包发送到远程主机。因为只有IP TTL值被改变，ICMP包可以被创建一次，并且用于所有的尝试。

程序清单11.5中的TraceRoute.cs程序执行这些功能。

程序清单11.5 TraceRoute.cs程序

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class TraceRoute
{
    public static void Main (string[] argv)
    {
```

```

byte[] data = new byte[1024];
int recv, timestart, timestop;
Socket host = new Socket(AddressFamily.InterNetwork,
    SocketType.Raw, ProtocolType.Icmp);
IPHostEntry iphe = Dns.Resolve(argv[0]);
IPEndPoint iep = new IPEndPoint(iphe.AddressList[0], 0);
EndPoint ep = (EndPoint)iep;
ICMP packet = new ICMP();

packet.Type = 0x08;
packet.Code = 0x00;
packet.Checksum = 0;
Buffer.BlockCopy(BitConverter.GetBytes(1), 0, packet.Message, 0, 2);
Buffer.BlockCopy(BitConverter.GetBytes(1), 0, packet.Message, 2, 2);
data = Encoding.ASCII.GetBytes("test packet");
Buffer.BlockCopy(data, 0, packet.Message, 4, data.Length);
packet.MessageSize = data.Length + 4;
int packetsize = packet.MessageSize + 4;

UInt16 checksum = packet.GetChecksum();
packet.Checksum = checksum;

host.SetSocketOption(SocketOptionLevel.Socket,
    SocketOptionName.ReceiveTimeout, 3000);

int badcount = 0;
for (int i = 1; i < 50; i++)
{
    host.SetSocketOption(SocketOptionLevel.IP,
        SocketOptionName.IpTimeToLive, i);
    timestart = Environment.TickCount;
    host.SendTo(packet.GetBytes(), packetsize, SocketFlags.None, iep);
    try
    {
        data = new byte[1024];
        recv = host.ReceiveFrom(data, ref ep);
        timestop = Environment.TickCount;
        ICMP response = new ICMP(data, recv);
        if (response.Type == 11)
            Console.WriteLine("hop {0}: response from {1}, {2}ms",
                i, ep.ToString(), timestop-timestart);
        if (response.Type == 0)
        {
            Console.WriteLine("{0} reached in {1} hops, {2}ms.",
                ep.ToString(), i, timestop-timestart);
            break;
        }
    }
}

```

```

        badcount = 0;
    } catch (SocketException)
    {
        Console.WriteLine("hop {0}: No response from remote host", i);
        badcount++;
        if (badcount == 5)
        {
            Console.WriteLine("Unable to contact remote host");
            break;
        }
    }
}

host.Close();
}
}

```

TraceRoute.cs程序创建一个普通的ICMP Echo Request包，并且进入一个循环，设置IP TTL值，并且不断增加该值，直到到达目的地主机：

```

for (int i = 1; i < 50; i++)
{
    host.SetSocketOption(SocketOptionLevel.IP,
        SocketOptionName.IpTimeToLive, i);
    timestart = Environment.TickCount;
    host.SendTo(packet.getBytes(), packet.size, SocketFlags.None, iep);
}

```

当每次发送Echo Request包，程序将等待响应。返回包的Type元素将进行检查，以确定包是否到达了目的地。两种类型的ICMP包将被返回：

- 当TTL值到期，路由器将发送Time Exceeded (Type 11) 包。
- 当目的地主机接收到Echo Request包，将发送Echo Reply (Type 0) 包。

通过检测Time Exceeded包的源，用户可以看到路径中使用了哪些路由器。在网络路径中，某些路由器可能配置为忽略ICMP包。这些路由器将产生“主机没有响应”的错误消息，但是TTL值被增长，并且下一个路由器将被查询。通常，用户将在网络路径中遇到几个忽略ICMP包的路由器。但是，目的地主机也可能会忽略ICMP包，或者无法被ICMP包激活。这个情况下，要防止包进入无休止的循环，当遇到5次没有响应，程序将假设远程主机无法到达。

同样，类似于AdvPing程序，Environment.TickCount特性值将测量每个路由器对Echo Request包的响应时间。

在这里，用户必须在编译器命令行中使用ICMP.cs文件来编译TraceRoute.cs程序：

```
csc TraceRoute.cs ICMP.cs
```

测试TraceRoute.cs

用户可以通过在命令行窗口中运行来测试TraceRoute.cs程序，使用已知的主机对ping进行响应：

```
C:\>TraceRoute www.cisco.com
hop 1: No response from remote host
hop 2: response from 206.148.207.106:0, 220ms
hop 3: response from 65.123.106.113:0, 140ms
hop 4: response from 205.171.20.125:0, 141ms
hop 5: response from 205.171.20.142:0, 140ms
hop 6: response from 205.171.1.162:0, 130ms
hop 7: response from 144.232.26.54:0, 130ms
hop 8: response from 144.232.8.117:0, 191ms
hop 9: response from 144.232.3.138:0, 190ms
hop 10: response from 144.228.44.14:0, 190ms
hop 11: response from 128.107.239.89:0, 190ms
hop 12: response from 128.107.239.102:0, 191ms
198.133.219.25:0 reached in 13 hops, 180ms.

C:\>
```

如用户所见，该程序从我的工作站到ISP连接中使用了13次转发，到达www.cisco.com主机。从这些转发中，用户可以看到单独路由器的IP地址。在路径中，当TTL值到期时，路径中的第一个路由器没有发送回ICMP包。

注意路径中每个路由器的不同响应时间。响应时间无需相同，因为一些网络和设备要比其他的快。

观察几个远程网络地址是非常有趣的。通常网络采用不同的路径到达不同的站点。用户可以看到ISP通过跟踪远程站点的路由连接到因特网。

FindMask程序

FindMask程序使用其他的ICMP消息类型自动得到连接到子网的设备的子网掩码。本节内容将描述ICMP Subnet Request包类型和FindMask.cs程序，该程序将检测本地网络的子网掩码。

Subnet Request（子网请求）包

ICMP包的一种延伸就是Subnet Request包（ICMP Type 17）。该包可以查询网络上的设备，确定网络的子网掩码。图11.4显示了Subnet Request包的格式。该包的标识符和序号区域与Echo Request包类似。每个包必须具有惟一的标识符和序号值，以便和设备发送的其他Subnet Request包进行区分。

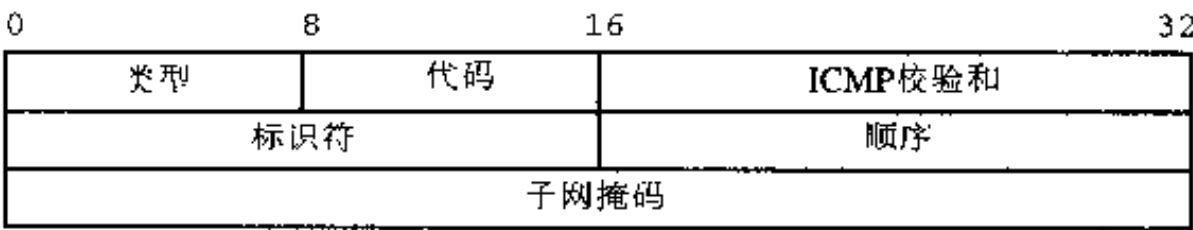


图11.4 ICMP Subnet Request包格式

标识符和序号区域后面具有4个字节的整形区域，来识别子网。Subnet Request包在这些区域全部设置为0。网络上的响应设备将使用子网的适当值来替换这些0。因为该值是长整形，用户将转换这些值为IPAddress对象，以便读取。IPAddress构造函数如下：

```
ICMP response = new ICMP(data, recv);
long answer = BitConverter.ToUInt32(response.Data, 4);
IPAddress netmask = new IPAddress(answer);
```

BitConverter类从接收到的ICMP包中提取4个字节的值，并且转换为长整形值。长整形值可以创建新的IPAddress对象。

程序清单11.6，FindMask.cs程序在本地网络上广播一个Subnet Request ICMP包。如果配置的任何设备对ICMP包进行响应，设备将返回一个应答。

程序清单11.6 FindMask.cs程序

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class FindMask
{
    public static void Main ()
    {
        byte[] data = new byte[1024];
        int recv;
        Socket host = new Socket(AddressFamily.InterNetwork,
            SocketType.Raw, ProtocolType.Icmp);
        IPEndPoint iep = new IPEndPoint(IPAddress.Broadcast, 0);
        EndPoint ep = (EndPoint)iep;
        ICMP packet = new ICMP();

        packet.Type = 0x11;
        packet.Code = 0x00;
        packet.Checksum = 0;
        Buffer.BlockCopy(BitConverter.GetBytes(1), 0, packet.Message, 0, 2);
        Buffer.BlockCopy(BitConverter.GetBytes(1), 0, packet.Message, 2, 2);
        Buffer.BlockCopy(BitConverter.GetBytes(0), 0, packet.Message, 4, 4);
        packet.MessageSize = 8;
```

```

int packetSize = packet.MessageSize + 4;

UInt16 chksm = packet.GetChecksum();
packet.Checksum = chksm;

host.SetSocketOption(SocketOptionLevel.Socket,
    SocketOptionName.ReceiveTimeout, 3000);
host.SetSocketOption(SocketOptionLevel.Socket,
    SocketOptionName.Broadcast, 1);
host.SendTo(packet.GetBytes(), packetSize, SocketFlags.None, iep);
try
{
    data = new byte[1024];
    recv = host.ReceiveFrom(data, ref ep);
} catch (SocketException)
{
    Console.WriteLine("Unable to determine subnet mask for this subnet");
    return;
}
ICMP response = new ICMP(data, recv);
Console.WriteLine("Received an ICMP type {0} packet", response.Type);
long answer = BitConverter.ToUInt32(response.Message, 4);
IPAddress netmask = new IPAddress(answer);
Console.WriteLine("The subnet mask for this subnet is: {0}",
    netmask.ToString());
}
}

```

类似本章介绍的其他ICMP程序，FindMask.cs程序必须使用ICMP.cs类文件进行编译：

```
csc FindMask.cs ICMP.cs
```

FindMask程序首先通过创建一个ICMP对象，并且在类中加入适当的元素来建立一个ICMP Subnet Request包。子网区域设置为0，以便进行请求并且由响应的网络设备进行填充。

如果没有收到响应，用户需要等待程序结束，ReceiveTimeout套接字选项将设置为一个适当的值。同样，这个程序使用IP广播来发送请求到子网上的所有设备，所以Broadcast套接字选项也必须被设置：

```

host.SetSocketOption(SocketOptionLevel.Socket,
    SocketOptionName.ReceiveTimeout, 3000);
host.SetSocketOption(SocketOptionLevel.Socket, SocketOptionName.Broadcast, 1);

```

当发送广播包后，ReceiveFrom()方法将被调用，等待网络上设备的响应。当接收到响应，子网掩码地址将被提取，并被转换到IPAddress对象。

测试FindMask.cs

要测试FindMask程序，需要在命令行提示符窗口运行程序，如果子网上的一个或者多

个设备对请求进行响应，用户将看到结果显示：

```
C:\>FindMask
Received an ICMP type 18 packet
The subnet mask for this subnet is: 255.255.252.0

C:\>
```

有一种可能，用户网络上的设备将不会对Type 17 ICMP包进行响应。同样，网络上的设备可能会对ICMP包响应一个ICMP类型错误包。如果用户收到类型18以外的包，表示网络上的设备发送了一个不同的ICMP包。

比输出更有趣的是，观察网络幕后所发生的事情。用户可以使用WinDump或者Analyzer来观察有多少设备对用户的请求进行了响应。程序清单11.7显示了当执行FindMask程序时，WinDump程序的输出例子。

程序清单11.7 FindMask程序的WinDump输出

```
C:\>windump -X icmp
windump listening on \Device\Packet_E190x1
13:21:49.576074 192.168.1.32 > 255.255.255.255: icmp: address mask request
0x0000 4500 0020 b355 0000 8001 6c5b c0a8 0120 E....U....lU..}.
0x0010 ffff ffff 1100 ecff 0100 0100 0000 0000 .....
13:21:49.576203 192.168.1.149 > 192.168.1.32: icmp: address mask is 0xfffffc00

0x0000 4500 0020 6103 0000 ff01 23ff c0a8 0195 E...a.....#...}.
0x0010 c0a8 0120 1200 effe 0100 0100 ffff fc00 ..}.....
0x0020 0000 0000 0000 0000 0000 0000 0000 .....
13:21:49.576236 192.168.1.143 > 192.168.1.32: icmp: address mask is 0xfffffc00

0x0000 4500 0020 8db8 0000 ff01 f74f c0a8 018f E.....O..}.
0x0010 c0a8 0120 1200 effe 0100 0100 ffff fc00 ..}.....
0x0020 0000 0000 0000 0000 0000 0000 0000 .....
13:21:49.576320 192.168.1.5 > 192.168.1.32: icmp: address mask is 0xfffffc00

0x0000 4500 0020 de96 0000 3c01 6afc c0a8 0105 E.....<.j...l.
0x0010 c0a8 0120 1200 effe 0100 0100 ffff fc00 ..}.....
0x0020 0000 0000 0000 0000 0000 0000 0000 .....
0x0030 0000 0000 0000 .....
408 packets received by filter
0 packets dropped by kernel

C:\>
```

在这个特殊的子网中，三个单独的设备对Subnet Request广播响应。如期望的那样，这些设备都返回同样的子网掩码值。

小结

本章显示了如何利用因特网控制消息协议（ICMP）创建程序。ICMP允许网络设备快速发送查询和错误消息到其他的网络设备。很多ICMP包类型可以在网络上使用，这些类型的公共之处是Echo Request包。当Echo Request包发送到一个设备，设备将使用一个Echo Reply包进行响应。该过程就是通用ping程序的基础，该程序测试网络设备是否可以到达，并且发送包到设备要多长时间。

程序开发者可以使用Socket类发送非TCP和非UDP包，该类包含SocketType.Raw特性。该特性允许使用ICMP数据创建原始的IP包。当使用这些原始套接字进行工作，用户必须小心处理接收到的消息。接收到的消息将包含完整的IP包信息，包括IP头区域。用户必须确保用户的代码通过缓存在返回的数据中增加IP包信息，就好像在缓存中对包的ICMP部分进行译码时一样。

对于ICMP程序来说，ICMP类可以确保处理ICMP包的详细信息。需要创建或者读取ICMP包的应用程序可以创建一个ICMP对象实体，并且放置需要的数据包到类的数据变量。该类同样包括一种方法，允许用户获取包的原始字节，以便使用SendTo()方法进行发送。当接收到ICMP包，一个单独的ICMP类构造函数将ICMP包中的原始数据放置到ICMP类，以便可以轻松访问。

本章介绍了一些ICMP工具，包含简单和高级的ping程序。本章同时创建了traceroute程序，该程序采用路由器处理包的方法，将完整的IP时间作为TTL值。当路由器接收到一个包，该包的TTL值将被下一个路由中止，并且发送一个ICMP Time Exceeded包，返回包的发送者。通过使用小的TTL值启动并对该值进行增加，用户可以跟踪包在路程中被处理的路由器。最后，ICMP Subnet Request将对网络设备进行查询，以便确认配置的网络子网掩码。

第12章 SNMP

- SNMP命令
- 团体名称
- 公共管理信息基准 (CMIB)
- 创建SNMP包
- 创建简单的SNMP类
- 一个示范SNMP程序: SimpleSNMP
- 厂家管理信息数据库
- 使用GetNextRequest查询

在网络管理中通常使用的协议是简单网络管理协议 (SNMP)。SNMP用于从中央管理位置查询并控制网络设备。

SNMP的行为和其他网络协议略有区别。与其他协议的设计不同, SNMP设计改变为依靠查询的类型和被查询数据的类型。在学习如何建立网络程序时, 这将是一个有趣的练习。

本章将讲述SNMP原理的基本概念和与网络上具有SNMP能力的设备通信所需要的所有内容。我们将示范几种典型的SNMP程序。当用户使用这些程序时, 用户将看到如何与多种网络设备进行通信, 并且从网络设备上获得不同类型的SNMP信息。如果用户负责网络的管理工作, 用户可以创建自己的SNMP程序来监视和控制网络上的设备。

理解SNMP

当因特网还处于发展初期时, 设计者认为必须有一种简便的方法来监视网络设备。经过多年的努力, 开发出很多类型的监视协议, 这就需要选择其中的一种作为网络监视的标准。最后, SNMP从众多协议中脱颖而出, 得到了广泛的接受, 用于监视网络设备。

SNMP的原理非常简单。每个网络设备都维持一个用于网络统计的数据库, 可以对远程设备进行查询。其中核心的数据库被称为管理信息数据库 (MIB), 定义在RFC 1155中。每种执行SNMP的网络设备都包含同样的核心MIB结构。MIB包含简单网络统计记录, 例如:

- 设备的网络名称
- 设备的网络接口号码
- 每个接口输入和输出包计数
- 每个网络接口的错误率
- 特殊协议计数, 例如TCP和UDP包

SNMP包含用于查询和设置数据库值的技术。因此, 很多网络环境的提供商使用SNMP数据库来存储网络设备的配置和设置。这样网络管理站就可以从网络上的某个位置获取和改

变网络上所有设备的配置和设置。

在用户开始编写SNMP程序前，用户必须理解SNMP如何工作，并且如何使用网络包来搜寻和获取网络数据。这节将描述这些内容。

SNMP和MIB版本

SNMP已经从1990年5月RFC 1157中定义的原始协议得到了改进，成为了一个功能齐全的工具，目前的版本是第3版。新版本的SNMP没有完全抛弃早期的版本，只是增加了一些新功能（例如认证和加密）。因此，所有的SNMP网络设备必须对原始SNMP版本1的请求进行响应。对于原始MIB数据库，同样存在这个问题。很多网络管理软件包仍然使用SNMP版本1协议来访问MIB信息。

本章描述了SNMP工作的基本概念，但是没有涉及认证和加密，提供的程序使用SNMP版本1请求来访问MIB信息。当工作网络环境中使用SNMP版本1时，需要格外小心，因为该版本无法使用纯文本发送设备信息。

SNMP命令

替代使用大量的命令来管理网络设备，SNMP使用MIB数据库来完成这些工作。每个控制功能都在数据库中分配一个记录。控制功能非常简单，好像在数据库中设置数值。这意味着SNMP仅需考虑获取和设置数据库中的值，而不是大量单独的命令，在网络设备上执行功能。

例如，替代使用特殊的SNMP命令关闭设备上的网络接口，设备包含一个数据库记录，关联到接口的状态。如果记录值表示“off”，设备关闭接口。这样就允许任何网络管理站发送SNMP命令来修改数据库值并且关闭接口。

使用MIB数据库降低SNMP包的复杂性。每个SNMP命令被称为协议数据单元（PDU）。在SNMP版本1中，只有5种类型的PDU，用于获取和设置数据库的值：

GetRequest GetRequest PDU查询远程SNMP设备，得到MIB数据库中的一个单值。

GetNextRequest GetNextRequest PDU查询远程SNMP设备，得到一系列数据值，这些数据值的起点都在MIB中指明了。当每个数据值返回，下一个MIB对象也被返回到查询设备下一个要查询的对象。这样就允许SNMP客户机使用远程SNMP设备上整个MIB数据库表。

GetResponse GetResponse PDU返回在GetRequest或者GetNextRequest查询中请求的信息。

SetRequest SetRequest PDU试图设置远程SNMP设备上的MIB数据库内的值。发送设备必须具有存取权限，在远程设备MIB数据库中写入数据。

Trap Trap PDU发送未经请求的数据值到远程SNMP管理站。Trap PDU允许设备报告错误情况到网络上的管理站，无需进行查询。

团体名称

因为SNMP控制可以超越设备，未经认证的用户必须被禁止修改网络设备的MIB数据库。SNMP提供一种通过团体名称系统的认证方法。

在网络设备上运行的SNMP应用程序包含一个单独的MIB数据库。团体名称是一个密码，可以对MIB数据库中的特殊级别进行访问（称为view）。团体名称可以对不同的MIB提供访问，SNMP允许对MIB数据库元素的两种访问方式：只读和读写。

具有SNMP MIB视点的这两种SNMP访问模式被称为团体概述文档。两种团体概述文档（community profile）的团体名称被称为SNMP access policy（访问规则）。很多网络设备允许用户创建大量的访问规则，但是有些设备只允许两种规则（一种读取整个MIB数据库，另外一种可以对整个MIB数据库进行读写操作）。

团体名称作为字节字符串定义。这些字符串没有长度的限制。两种最为流行的团体名称（通常作为网络设备的默认值）是public，用于只读访问，和private，用于读写访问。明显的，如果用户配置SNMP访问规则，用户不会希望使用这两种众所周知的值。

公共管理信息库

公共MIB是默认的标准数据库，用于SNMP在网络设备上执行。该标准定义在RFC 1155中，为网络设备提供一套数据库格式。自从该标准诞生以来，已经延伸到第二版（MIB-II），并且成为了当前的标准。

MIB结构

MIB结构是SNMP中最令人费解的部分之一。MIB使用树状层次数据库结构，将数据记录定义为对象。每个对象都是通过对象标识符进行引用，并且具有一个对象值。

对象标识符采用抽象化语法表示，版本1（称为ASN.1）命名约定。这个标准的语法为每个对象分配一个数字值，在基于树状的结构中引用对象，采用等级相同的命名规则。命名规则类似于DNS的命名规则，使用点“.”将名称等级分开，SNMP根节点是最高级别。

对于SNMP，根对象命名为iso，对于国际标准化组织，分配了对象标识符为1。所有的MIB条目都通过这个对象进行定位。iso对象包含一个子节点，命名为org，可以使用名称（iso.org）或者通过对象标识符（1.3）进行引用。第三级对象，dod，具有一个对象标识符1.3.6。

所有的MIB-II数据库条目都通过internet对象进行定位，该对象的子节点是dod对象，具有对象标识符1.3.6.1。

用户可以参考官方的ASN.1语法命名。ASN.1语法使用高级别对象为对象定义，因此，对于internet对象，ASN.1语法定义如下：

```
internet ::= { iso(1) org(3) dod(6) 1 }
```

该语法在MIB数据库结构中为internet定义精确的位置。该对象同样可以通过名称iso.org.dod.internet进行引用，但是更多情况是使用对象标识符进行引用。

说明：因特网体系结构（IAB）可以用于internet对象下的所有MIB对象标识符。

在对象下的每个子节点都被分配一个唯一的对象标识符。要引用单独的对象，用户必须列出每个对象级别的对象标识符，从顶级开始，在internet对象下有四个子节点，如图12.1所示。

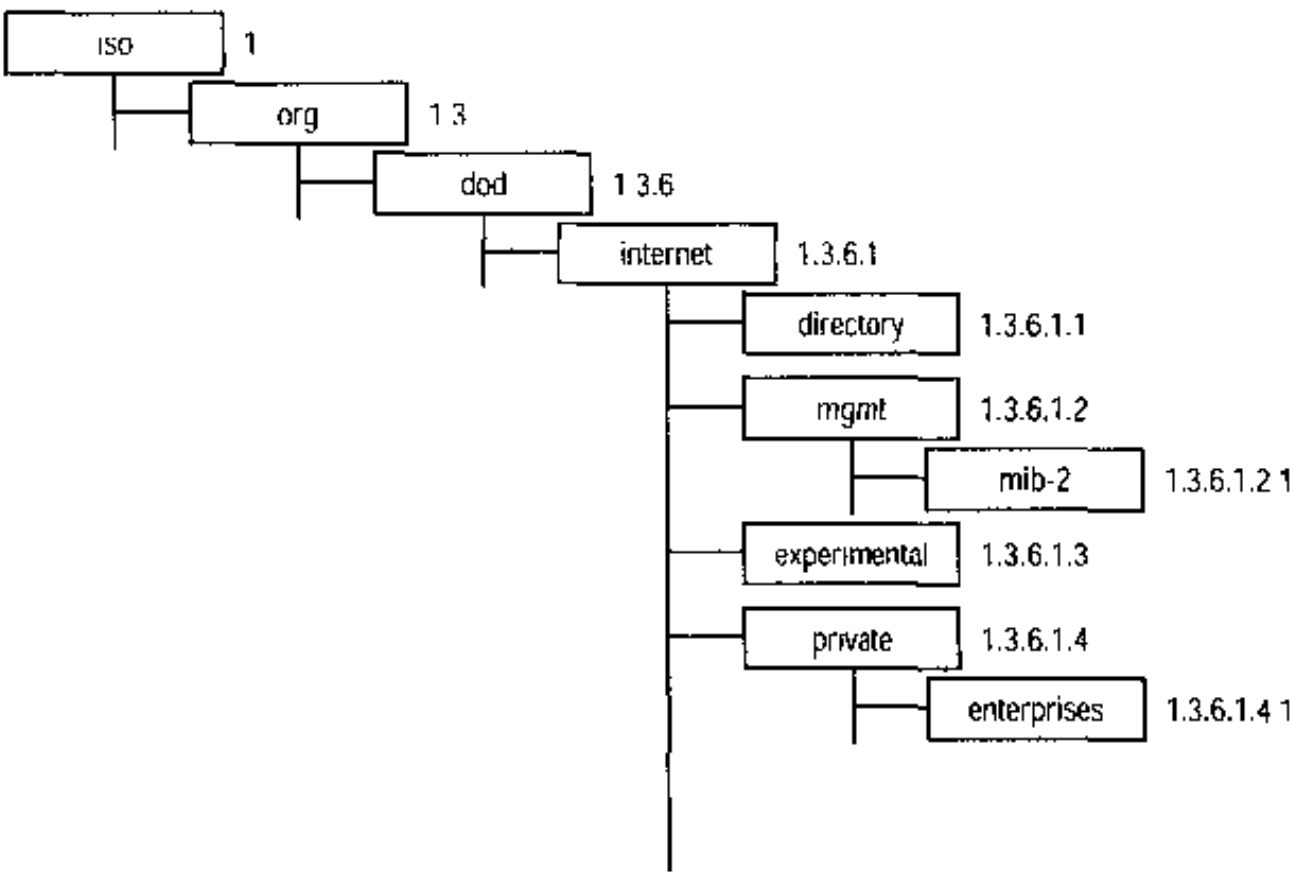


图12.1 iso.org.dod.internet对象

如图12.1所示，mgmt对象包含mib-2对象（1.3.6.1.2.1），该对象包含MIB-II数据库对象。这些mib-2对象包含很多网络统计对象，允许用户查询网络设备，用于性能监视。mib-2对象列在表12.1中。

表12.1 mib-2对象用于网络统计

对象	节点标识符	说明
System	1	特殊系统值
Interfaces	2	网络接口统计
At	3	地址翻译表
Ip	4	IP统计
Icmp	5	ICMP统计
Tcp	6	TCP统计
Udp	7	UDP统计
Egp	8	EGP统计
Transmission	10	传输统计
Snmp	11	SNMP统计

每个mib-2对象都包含多个子节点，代表网络设备的数据库值。例如，system对象包含下列对象：

sysDescr (1) 网络设备描述

sysObjectID (2) 对象的惟一供货商标识符

sysUpTime (3) 从网络管理部分进行系统初始化以来的时间（百秒级）

sysContact (4) 网络设备的接触点

sysName (5) 为设备分配的管理名称

sysLocation (6) 网络设备的描述位置

sysServices (7) 描述设备提供服务的值

每个系统子节点都被分配一个惟一的节点标识符（显示在圆括号中），并通过ASN.1语法的对象标识符进行引用。例如，要观察网络设备的系统描述，用户可以查询1.3.6.1.2.1.1.1对象标识符。

说明： 当在GetRequest SNMP命令实体中指定某个单独的对象实体，需要在对象标识符的后面加一个0。这样就可以引用对象单独的实体值。在GetRequest包中，SysDescr对象的一个实体可以引用为1.3.6.1.2.1.1.1.0。

MIB对象值

一旦用户找到所需对象的标识符，用户就可以获取数据。要完成这项工作，用户需要知道这些信息如何存储在数据库中。**MIB**对象有很多数据类型；其中一些类型是通用的数据类型，如下：

Integer	类型2
Byte string	类型4
Null	类型5
Object Identifier	类型6
Sequence	类型48（用于本章的例子中）
TimeTick	类型67

例如，sysDescr对象（1.3.6.1.2.1.1.1.0）内保存着byte string（类型4）。byte string可以作为C#语言字符串变量被读取和存储，并且系统描述进行显示。

说明： Integer（整形）数据类型很难使用，因为该类型无法设置长度。该类型可以为1个字节或者4个字节。长度通常在接收到的SNMP包中说明。这样就使得SNMP包在处理小或者大值时非常灵活，对于网络编程者来说，将接收到的SNMP包进行译码将是个挑战。

使用SNMP包工作

现在，用户已经看到了SNMP系统的构成，让我们讨论一些如何在包中放置这些部分，来发送到远程设备。这节内容将讨论如何创建SNMP包，并且如何使用C#语言进行创建和发送。

SNMP包格式

每个SNMP包都被定义为一个序列。该序列由3个基本部分组成：

- SNMP版本号
- 在查询中使用的团体名称
- SNMP PDU（协议数据单元）

版本和团体名称

SNMP包的第一部分是标准的PDU类型。对于SNMP版本1包来说，版本号为0（很奇怪，但是真实的）。第二部分，团体名称，同样采用标准的PDU类型。在团体名称中，每个字符都放置在包的一个字节中。如果SNMP包中传递的团体名称无效，网络设备将不会进行响应。

PDU

包的最后一部分是连续的PDU。该部分包含特定信息，说明包的PDU类型。每个SNMP包都只包含一个PDU区域，该区域只包含一种PDU类型。

SNMP版本1具有5种PDU类型，但是其中3种（GetRequest、GetNextRequest和GetResponse）都使用相同的PDU格式，这样可以简便操作。让我们看看这些PDU类型的具体内容。它们的共同格式包含下面的区域，定义PDU信息发送到SNMP设备：

- PDU类型
- Request-ID
- 错误状态
- 错误索引
- 一个或者多个变量绑定

PDU类型

PDU类型为一个1字节的值，说明SNMP包使用的PDU格式。对于PDU格式，该值如下：

GetRequest	0xA0
GetNextRequest	0xA1
GetResponse	0xA2
SetRequest	0xA3
Trap	0xA4

Request-ID

request-ID为一个4字节的整形变量，作为每个查询的惟一标识符发送到设备。设备的响应必须包含机器的request-ID。

错误状态和错误索引

错误状态和错误索引区域都是1字节的整形值。这两个区域包含1个0，用于GetRequest。

GetResponse包包含的值，用于说明在SNMP传输中是否发生了错误，并且错误的类型如何，如表12.2所示。错误索引功能定义在SNMP传输中错误状态的特殊错误情况。

表12.2 SNMP错误状态值

错误状态	名称	描述
0	No error	在传输过程中没有错误发生
1	tooBig	SNMP包过大
2	noSuchName	MIB对象不存在
3	badValue	MIB对象值不是正确的数据类型，或者没有在正确的范围
4	readOnly	在只读条件下，SetRequest消息试图修改MIB条目
5	genErr	一般错误情况

变量绑定

在SNMP包中，变量绑定代表一个单独的对象标识符/对象值。例如，下面是sysName MIB对象的一个有效变量绑定：

```
1.3.6.1.2.1.1.5.0Jessica's workstation
```

在这个例子中，对象标识符1.3.6.1.2.1.1.5.0是sysName对象，并且对象值为Jessica's workstation。

对于GetRequest和GetNextRequest PDU，无法确定对象值（因为该对象用于查询远程设备值），在这种情况下，该对象将设置为空。远程设备返回的GetResponse包将对象值设置为对象查询的数据库值。

一个单独的SNMP包可以包含多个变量绑定。变量绑定使用连续的标识符进行分隔（在“SNMP包设计”部分讨论）。在变量绑定的数量上没有任何限制。GetResponse包将包含由查询设备发送的每个变量绑定的信息。

SNMP包设计

运行SNMP时，感到麻烦的就是将所有的SNMP放置在包中。对于这些包，有很多的规则，如果任何事情发生了错误，远程主机将忽略该SNMP包。因为SNMP使用UDP，用户将无法确认SNMP查询是否在网络上丢失，或者发生了什么错误，使得远程设备忽略该包。这就需要故障分析。

学习SNMP包设计的最佳方法就是显示一个SNMP包的例子，并且一个字节一个字节地观察。让我们看看图12.2。

SNMP包使用连续的字节（0×30）开始，表示SNMP传输的开始。

接下来是整个SNMP包的长度（0×28），以便远程设备可以确定接收的包是否完整。

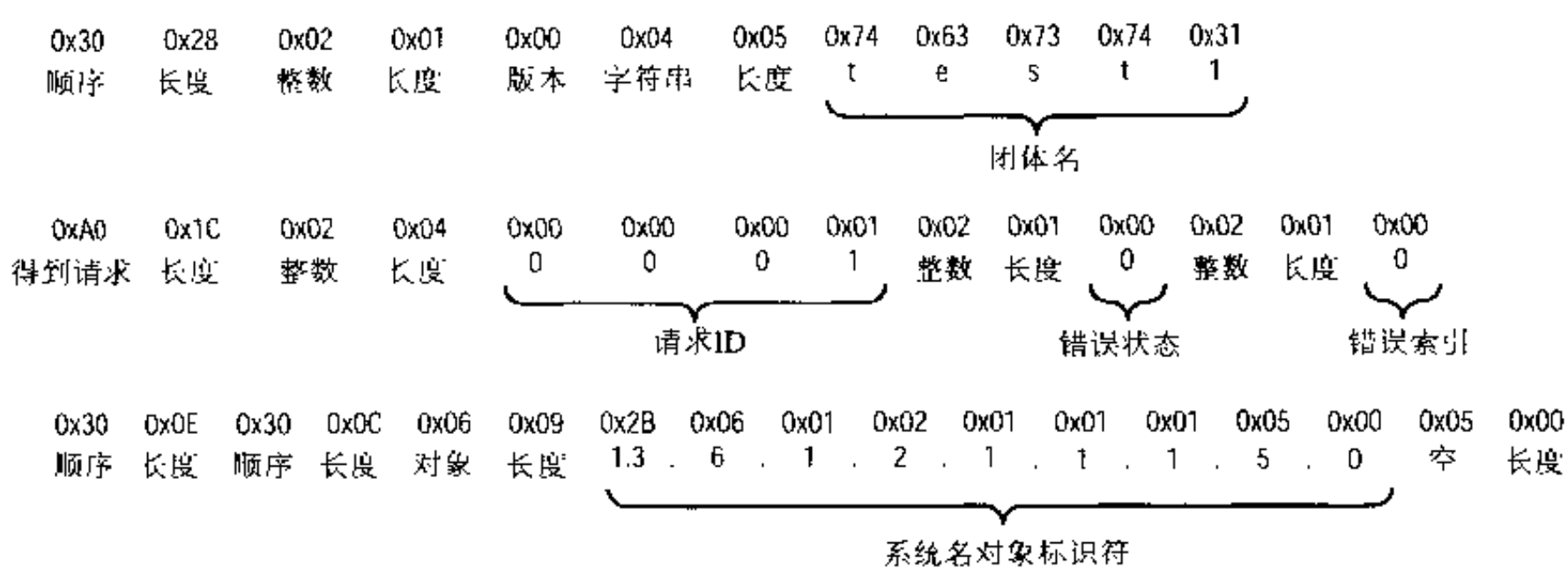


图12.2 简单SNMP GetRequest包设计

在长度字节后，每个包含的数据值前都放置2字节的序列。在这些序列中，第一个字节表示对象值的SNMP数据类型。第二个字节表示值的长度。这里有一个2字节值的例子，如图12.2所示。

SNMP版本号 一个整形的数据类型（0×02），长度为1（0×01）。该版本号值本身为0（0×00），表示使用的是SNMP版本1包（见图12.2）。

团体名称 开头使用2字节，然后是字符串对象数据类型（0×04）和团体名称字符串的长度（0×05），接下来是团体名称字节。

SNMP PDU类型 一个单独的字节。因为这里的值为0×A0，该包为GetRequest PDU包。

包中PDU区域的长度 一个单独的字节（0×1c）。

Request ID 整形数据类型（0×02），长度为4字节（0×04），是真实的请求ID。request ID可以为任何与GetResponse Request ID匹配的值。在这个例子中，该值为1（0×00 0×00 0×00 0×01）。

错误信息 信息中包含错误状态和错误索引值，使用各自的数据类型和长度。这里是一个GetRequest包，所以错误状态和错误索引值为0。

变量绑定部分 因为变量绑定可能不止1个，因此该部分为一个Sequence字节（0x30），和全部变量绑定部分的长度（0×0e）。在这以后，每个单独的变量绑定将跟随各自的Sequence和长度字节。

在变量绑定中的对象标识符/对象值对 对象标识符具有数据类型，显示为Object Identifier（0×06）。跟随数据类型的为对象标识符的长度（0×09），随后是对象标识符本身。

注意，这里有一点奇怪，对象标识符如何放置在SNMP包中的呢？每个单独的对象标识符节点值都被转换为一个单字节值，除了两种例外：

- 在对象标识符中（1.3），前两个对象被转换为单字节值0×2B
- 节点值超过127，被转换到2字节的带符号整数

在例子中，对象标识符为1.3.6.1.2.1.1.5.0。这表示sysName对象的单个实体。

对象值 这里是一个GetRequest包，所以对象值为空，由null数据类型（0×05）表示，并且长度为0（0×00）。

· SNMP通信

SNMP包使用UDP与远程设备进行通信。因为UDP是无连接协议，用户的SNMP程序可能会遇到典型的UDP问题，如第6章“无连接套接字”中描述的。在发送和接收SNMP包时，最大的问题是，包在网络中丢失，永远无法被接收。

SNMP使用两个单独的UDP端口进行通信：端口161用于GetRequest、GetNextRequest和SetRequest PDU，端口162用于Trap PDU。

个别的SNMP包可以通过在SNMP字节数组中放置适当的数据而创建。当整个包被创建，字节数组将发送到远程网络设备，并且使用适当的UDP端口。接收GetRequest、GetNextRequest和SetRequest查询的设备必须使用端口161进行收听，而SNMP管理站必须使用端口162收听SNMP设备发送的Trap包。

创建简单的SNMP类

创建SNMP类不同于创建ICMP类（该内容在第11章中讨论）。因为SNMP包没有固定的字节格式，用户必须按照自己的需求创建SNMP包，加入所需要的值。

这样就使得类的构造函数非常复杂，因为构造函数必须非常灵活地改变区域的尺寸和PDU类型。我发现可以采用一个类的方法来创建SNMP包，该方法允许用户创建包，并且直接发送和接收SNMP包。因此，用户可以快速发送和接收多个SNMP包，无需为每个包创建新的类。

在创建SNMP包时，需要3个部分的信息：

- 包的PDU类型
- 查询的MIB对象标识符
- 获得对MIB数据库有效访问的团体名称

这些值都将被传递到类的方法并且创建到SNMP包中。当包返回，方法可以返回一个原始的SNMP包到应用程序，或者进行译码，将值放入类中。

SNMP类程序

程序清单12.1中的SNMP.cs程序采用默认的构造函数和方法定义了一个SNMP类。get()方法用于将所有的方法用于创建GetRequest或者GetNextRequest PDU包，并且发送到指定的远程主机。从远程主机接收到的响应包将直接传送到调用程序。该方法定义了创建SNMP包所需的每个字节，如前面“SNMP包设计”中阐述的。

程序清单12.1 SNMP.cs程序

```
using System;
using System.Net;
using System.Net.Sockets;
```

```

using System.Text;

class SNMP
{
    public SNMP()
    {
    }

    public byte[] get(string request, string host, string community, string
mibstring)
    {
        byte[] packet = new byte[1024];
        byte[] mib = new byte[1024];
        int snmplen;
        int comlen = community.Length;
        string[] mibvals = mibstring.Split('.');
        int miblen = mibvals.Length;
        int cnt = 0, temp, i;
        int orgmiblen = miblen;
        int pos = 0;

        // Convert the string MIB into a byte array of integer values
        // Unfortunately, values over 128 require multiple bytes
        // which also increases the MIB length
        for (i = 0; i < orgmiblen; i++)
        {
            temp = Convert.ToInt16(mibvals[i]);
            if (temp > 127)
            {
                mib[cnt] = Convert.ToByte(128 + (temp / 128));
                mib[cnt + 1] = Convert.ToByte(temp - ((temp / 128) * 128));
                cnt += 2;
                miblen++;
            } else
            {
                mib[cnt] = Convert.ToByte(temp);
                cnt++;
            }
        }

        snmplen = 29 + comlen + miblen - 1; //Length of entire SNMP packet

        //The SNMP sequence start
        packet[pos++] = 0x30; //Sequence start
        packet[pos++] = Convert.ToByte(snmplen - 2); //sequence size

        //SNMP version
        packet[pos++] = 0x02; //Integer type
    }
}

```

```

packet[pos++] = 0x01; //length
packet[pos++] = 0x00; //SNMP version 1

//Community name
packet[pos++] = 0x04; // String type
packet[pos++] = Convert.ToByte(comlen); //length
//Convert community name to byte array
byte[] data = Encoding.ASCII.GetBytes(community);
for (i = 0; i < data.Length; i++)
{
    packet[pos++] = data[i];
}

//Add GetRequest or GetNextRequest value
if (request == "get")
    packet[pos++] = 0xA0;
else
    packet[pos++] = 0xA1;

packet[pos++] = Convert.ToByte(20 + miblen - 1); //Size of total MIB

//Request ID
packet[pos++] = 0x02; //Integer type
packet[pos++] = 0x04; //length
packet[pos++] = 0x00; //SNMP request ID
packet[pos++] = 0x00;
packet[pos++] = 0x00;
packet[pos++] = 0x01;

//Error status
packet[pos++] = 0x02; //Integer type
packet[pos++] = 0x01; //length
packet[pos++] = 0x00; //SNMP error status

//Error index
packet[pos++] = 0x02; //Integer type
packet[pos++] = 0x01; //length
packet[pos++] = 0x00; //SNMP error index

//Start of variable bindings
packet[pos++] = 0x30; //Start of variable bindings sequence

packet[pos++] = Convert.ToByte(6 + miblen - 1); // Size of variable
binding
packet[pos++] = 0x30; //Start of first variable bindings sequence
packet[pos++] = Convert.ToByte(6 + miblen - 1 - 2); // size
packet[pos++] = 0x06; //Object type
packet[pos++] = Convert.ToByte(miblen - 1); //length

//Start of MIB

```

```

    packet[pos++] = 0x2b;
    //Place MIB array in packet
    for(i = 2; i < miblen; i++)
        packet[pos++] = Convert.ToByte(mib[i]);
    packet[pos++] = 0x05; //Null object value
    packet[pos++] = 0x00; //Null

    //Send packet to destination
    Socket sock = new Socket(AddressFamily.InterNetwork, SocketType.Dgram,
        ProtocolType.Udp);
    sock.SetSocketOption(SocketOptionLevel.Socket,
        SocketOptionName.ReceiveTimeout, 5000);
    IPHostEntry ihe = Dns.Resolve(host);
    IPEndPoint iep = new IPEndPoint(ihe.AddressList[0], 161);
    EndPoint ep = (EndPoint)iep;
    sock.SendTo(packet, snmplen, SocketFlags.None, iep);

    //Receive response from packet
    try
    {
        int recv = sock.ReceiveFrom(packet, ref ep);
    } catch (SocketException)
    {
        packet[0] = 0xff;
    }
    return packet;
}
}

```

使用类

当用户观察SNMP类时可以理解为什么SNMP编程非常复杂。手工创建SNMP包，不是一个心脏虚弱的人干得了的事情。

转换MIB

首先，MIB对象标识符转换到字节数组：

```

byte[] mib = new byte[1024];
string[] mibvals = mibstring.Split('.');
int miblen = mibvals.Length;
int cnt = 0, temp, i;
int orgmiblen = miblen
for (i = 0; i < orgmiblen; i++)
{
    temp = Convert.ToInt16(mibvals[i]);
    if (temp > 127)

```

```

    {
        mib[cnt] = Convert.ToByte(128 + (temp / 128));
        mib[cnt + 1] = Convert.ToByte(temp - ((temp / 128) * 128));
        cnt += 2;
        miblen++;
    } else
    {
        mib[cnt] = Convert.ToByte(temp);
        cnt++;
    }
}

```

从MIB中提取的对象标识符将放置在字符串数组中，使用字符串Split()方法，该值只包含数字的值（作为字符串存储）。接下来，每个标识符将转换为一个整形值。SNMP使得该任务具有一定的挑战性。

整形MIB值被转换为16位带符号整形，尽管没有负的标识符值。如果值放入一个单独的字节，就好像放入了字节数组。如果值需要两个字节（长度大于127），高端字节将首先放入包中，然后才是低端字节。当值需要两个字节，那么MIB的总长度值miblength将随之增长。在创建真正的SNMP包时，这个值非常重要。产生的字节数组mib[]将包含MIB对象标识符的字节。

创建包

当MIB被转变为字节数组，整个SNMP包将创建。SNMP包的每个部分都被单独地增加到包的字节数组中。团体名称从字符串转变为字节数组，并且使用for循环放入数组中。MIB字节数组同样放置在包中，使用同样的技术。

SNMP GetRequest和GetNextRequest包的惟一不同之处在于PDU类型值。GetRequest包值为0×A0，而GetNextRequest的值为0×A1。因为类get()方法可以用于这两种类型的包，通过调用程序，该方法必须提供调用应用程序的方法，并且指定使用哪种PDU类型。这种方法使用简单的字符串值。当值等于“get”，将使用GetRequest PDU类型，否则，将使用GetNextRequest PDU类型。

发送包

当包被创建，将创建一个Socket对象与远程设备的SNMP端口连接（UDP端口161）：

```

Socket sock = new Socket(AddressFamily.InterNetwork,
    SocketType.Dgram,
    ProtocolType.Udp);
sock.SetSocketOption(SocketOptionLevel.Socket,
    SocketOptionName.ReceiveTimeout, 5000);
IPHostEntry ihe = Dns.Resolve(host);
IPEndPoint iep = new IPEndPoint(ihe.AddressList[0], 161);
EndPoint ep = (EndPoint)iep;
sock.SendTo(packet, iep);

```

SetSocketOption()方法设置ReceiveTimeout套接字选项为适当的值，等待远程设备的响应。

接收响应

当包被发送后，Receive()方法将等待响应：

```
try
{
    int recv = sock.ReceiveFrom(packet, ref ep);
} catch (SocketException)
{
    packet[0] = 0xff;
}
return packet;
```

如果在5秒钟内没有从远程设备接收到响应，返回包的第一个字节将被设置为0xff。调用的程序将检测该值，确认是否接收到有效的SNMP包。

SimpleSNMP程序

系统对象标识符通常用于跟踪和标识网络上的设备。它们同样可以用于帮助网络管理人员跟踪从中央网络管理站发送的信息。对于用户的第一个简单SNMP程序，用户可以通过使用查询网络设备，得到MIB数据库内的系统信息。

表12.3 在SimpleSNMP.cs中使用的系统对象值

对象	对象标识符	数据类型	说明
sysUptime	1.3.6.1.2.1.1.3.0	整形	设备已经启动的时间（百秒）
sysContact	1.3.6.1.2.1.1.4.0	字符串	设备的负责人
sysName	1.3.6.1.2.1.1.5.0	字符串	设备的网络名称
sysLocation	1.3.6.1.2.1.1.6.0	字符串	设备的位置

说明：该程序需要用户拥有具有SNMP能力的网络设备和有效的团体名称，以便在设备上查询数据，包括Windows NT或者2000服务器版。

程序清单12.2中的SimpleSNMP.cs程序是一个简单的网络程序，用于从远程使用SNMP类的网络设备获取系统对象。要编译SimpleSNMP.cs程序，用户必须在编译器命令行中使用SNMP.cs类文件：

```
csc SimpleSNMP.cs SNMP.cs
```


程序清单12.2 SimpleSNMP.cs程序

```
using System;
using System.Text;

class SimpleSNMP
{
    public static void Main(string[] argv)
    {
        int commlength, miblength, datatype, datalength, datastart;
        int uptime = 0;
        string output;
        SNMP conn = new SNMP();
        byte[] response = new byte[1024];

        Console.WriteLine("Device SNMP information:");

        // Send sysName SNMP request and get the response
        response = conn.get("get", argv[0], argv[1], "1.3.6.1.2.1.1.5.0");
        if (response[0] == 0xff)
        {
            Console.WriteLine("No response from {0}", argv[0]);
            return;
        }

        // Get the community name and MIB lengths from the packet
        commlength = Convert.ToInt16(response[6]);
        miblength = Convert.ToInt16(response[23 + commlength]);

        // Extract the MIB data from the SNMP response
        datatype = Convert.ToInt16(response[24 + commlength + miblength]);
        datalength = Convert.ToInt16(response[25 + commlength + miblength]);
        datastart = 26 + commlength + miblength;
        output = Encoding.ASCII.GetString(response, datastart, datalength);
        Console.WriteLine(" sysName - Datatype: {0}, Value: {1}",
            datatype, output);

        // Send a sysLocation SNMP request
        response = conn.get("get", argv[0], argv[1], "1.3.6.1.2.1.1.6.0");
        if (response[0] == 0xff)
        {
            Console.WriteLine("No response from {0}", argv[0]);
            return;
        }

        // Get the community name and MIB lengths from the response
        commlength = Convert.ToInt16(response[6]);
        miblength = Convert.ToInt16(response[23 + commlength]);
    }
}
```

```

// Extract the MIB data from the SNMP response
datatype = Convert.ToInt16(response[24 + commlength + miblength]);
datalength = Convert.ToInt16(response[25 + commlength + miblength]);
datastart = 26 + commlength + miblength;
output = Encoding.ASCII.GetString(response, datastart, datalength);
Console.WriteLine(" sysLocation - Datatype: {0}, Value: {1}",
    datatype, output);

// Send a sysContact SNMP request
response = conn.get("get", argv[0], argv[1], "1.3.6.1.2.1.1.4.0");
if (response[0] == 0xff)
{
    Console.WriteLine("No response from {0}", argv[0]);
    return;
}

// Get the community and MIB lengths
commlength = Convert.ToInt16(response[6]);
miblength = Convert.ToInt16(response[23 + commlength]);

// Extract the MIB data from the SNMP response
datatype = Convert.ToInt16(response[24 + commlength + miblength]);
datalength = Convert.ToInt16(response[25 + commlength + miblength]);
datastart = 26 + commlength + miblength;
output = Encoding.ASCII.GetString(response, datastart, datalength);
Console.WriteLine(" sysContact - Datatype: {0}, Value: {1}",
    datatype, output);

// Send a SysUptime SNMP request
response = conn.get("get", argv[0], argv[1], "1.3.6.1.2.1.1.3.0");
if (response[0] == 0xff)
{
    Console.WriteLine("No response from {0}", argv[0]);
    return;
}

// Get the community and MIB lengths of the response
commlength = Convert.ToInt16(response[6]);
miblength = Convert.ToInt16(response[23 + commlength]);

// Extract the MIB data from the SNMP response
datatype = Convert.ToInt16(response[24 + commlength + miblength]);
datalength = Convert.ToInt16(response[25 + commlength + miblength]);
datastart = 26 + commlength + miblength;

// The sysUptime value may be a multi-byte integer
// Each byte read must be shifted to the higher byte order
while(datalength > 0)

```



```

    {
        uptime = (uptime << 8) + response[datastart++];
        datalength--;
    }
    Console.WriteLine(" sysUptime - Datatype: {0}, Value: {1}",
        datatype, uptime);
}
}

```

SimpleSNMP程序使用SNMP类的get()方法将每个GetRequest包发送到远程网络设备。当每个GetResponse包被接收到，将通过SNMP包进行译码。

说明：这里提供了一个非常好的机会，可以使用WinDump或者Analyzer程序。当发送测试的SNMP请求包后，用户将看到SNMP设备返回的响应包。用户必须对SNMP响应包进行译码，来确定包中是否有用户希望的数据。在这些例子中，用于引用变量的号码来自位于测试SNMP响应包中的计数字节。

首先，团体名称长度和MIB长度要通过返回包进行计算：

```

int commlength = Convert.ToInt16(response[6]);
int miblength = Convert.ToInt16(response[23 + commlength]);

```

团体长度非常简单，它总是位于包中的同一位置。miblength字节的长度依赖团体名称的长度。

在团体名称和MIB长度后，用户要确定返回对象值的数据类型和起始位置、长度。当用户知道字节字符串的位置和长度，用户可以从包中将数据提取出来：

```

int datatype = Convert.ToInt16(response[24 + commlength + miblength]);
int datalength = Convert.ToInt16(response[25 + commlength + miblength]);
int datastart = 26 + commlength + miblength;
string output = Encoding.ASCII.GetString(response, datastart, datalength);
Console.WriteLine("response:");
Console.WriteLine(" Datatype: {0}, length: {1}", datatype, datalength);
Console.WriteLine(" data: {0}", output);

```

SysNameMIB对象表示一个字符串值，所以很容易找到字符串的位置（知道包的格式和团体名称、MIB长度）。

SysUptime译码可能有一点麻烦。这里无法预先确定字节的数量。用户必须设计好多种可能，从1个字节到5个字节。datalength值表示值使用了多少字节。当每个字节被读取，该值将增加1。当另一个字节被读取，前面的字节将上升8位，进入高端字节。

测试程序

当用户使用只读团体名称进行访问时，可以使用网络上具有SNMP能力的设备测试SimpleSNMP程序。很多网络集线器和交换机，包括Windows NT和2000服务器，可以替代SNMP设备进行测试：

```
C:\>SimpleSNMP 192.168.1.100 test1
Device SNMP information:
  sysName - Datatype: 4, Value: Rich's workstation
  sysLocation - Datatype: 4, Value: The office
  sysContact - Datatype: 4, Value: Rich Blum
  sysUptime - Datatype: 67, Value: 4906397

C:\>
```

对于这个例子，远程设备位于IP地址192.168.1.100，并且使用只读团体名称进行test1。如果用户创建了一个SNMP包，并且使用有效的团体名称，用户可以从远程设备得到一个SNMP响应。如果用户没有得到响应，用户可以使用WinDump或者Analyzer来观察发送的SNMP包。

说明：Windows 2000服务器在默认情况下没有SNMP服务。用户必须手动安装SNMP服务，并且配置系统信息。默认的团体名称为“public”。

观察包

比观察这个程序输出结果更为有趣的是观察程序产生的包。要理解SNMP包的最佳方法就是观察网络上SNMP的传输。程序清单12.3显示了SNMP GetRequest和GetResponse包，这些包是由SimpleSNMP程序在进行sysName SNMP查询时产生的。

程序清单12.3 SimpleSNMP程序网络轨迹 (WinDump)

```
C:\>windump -X udp port 161
windump listening on \Device\Facket_E190x1
10:19:22.728534 192.168.1.6.1450 > 192.168.1.100.161: [12 extra after iSEQ]C=test1 GetRequest(28) system.sysName.0
0x0000 4500 041c fb1f 0000 8011 0667 c0a8 0106 E.....g..}.
0x0010 c0a8 0164 05aa 00a1 0408 5f8b 3028 0201 ..1....._.0(..
0x0020 0004 0574 6373 7431 a01c 0204 0000 0001 ...test1.....
0x0030 0201 0002 0100 300e 300c 0608 2b06 0102 .....0.0...+...
0x0040 0101 0500 0500 0000 0000 0000 0000 0000 .....
0x0050 0000 ..
10:19:22.731609 192.168.1.100 > 192.168.1.6.1450: C=test1 GetResponse(40)
system.sysName.0="Rich's workstat"
0x0000 4500 005b 755e 0000 1e11 f1e9 c0a8 0164 E..[u^.....}.
0x0010 c0a8 0106 00a1 05aa 0047 f69e 303d 0201 ..}.....G..0=..
0x0020 0004 0574 6373 7431 a231 0201 0102 0100 ...test1.1.....
0x0030 0201 0030 2630 2406 082b 0601 0201 0105 ...0&0$.+.....
0x0040 0004 1252 6964 6827 7320 776f 726b 7374 ...Rich's workst
0x0050 6274 at

551 packets received by filter
0 packets dropped by kernel

C:\>
```

WinDump程序发现SNMP包，并进行同样的译码工作。包的标题显示每个SNMP包的相关信息：

```
10:19:22.728534 192.168.1.6.1450 > 192.168.1.100.161: [12 extra after iSEQ]C=test1 GetRequest(28) system.sysName.0
```

该包为一个GetRequest包，用于查询sysName对象，位于mib-2 system对象。注意，团体名称同样显示在WinDump头（C=test1）。

原始的包包含IP和UDP包头，和SNMP包信息相同。SNMP包起始于0×001c，由0×30 sequence字节组成。用户将发现包中的字节。团体名称和MIB对象标识符都包含在包中。

WinDump同样检测出SNMP GetResponse包中由远程设备返回的重要信息：

```
10:19:22.731609 192.168.1.100 > 192.168.1.6.1450: C=test1 GetResponse(40) system.sysName.0-"Rich's workstat"
```

注意对象值是被删节的，这是由于WinDump默认的尺寸造成的。如果用户希望看到全部的GetResponse包，用户可以使用-s选项来增加缓存的尺寸。

使用供货商MIB

大多数网络设备供货商将企业MIB对象（1.3.6.1.4.1）增加到自己设备的配置和统计中，以扩大标准的MIB-II统计。IAB分配子节点值给供货商。当编写真正的SNMP网络应用程序时，用户经常要使用企业数据库对象，来控制网络设备，提取有用信息。

本节内容描述如何解释供货商MIB数据和如何创建一个SNMP程序，查询网络设备的供货商MIB。使用SNMP应用程序可以完成这项工作，CiscoRouter可以作为网络管理使用。对于网络上的几种Cisco路由器，会发现每个路由器都处于整天工作状态，没有进行过手动Telnet来观察状态。在Cisco路由器上，观察CPU的利用率是非常有用的。通过观察CPU利用率，用户可以知道路由器何时过载，或者网络上发生了故障。

CiscoRouter程序发送一个SNMP包到Cisco路由器，应用程序Cisco供货商MIB查询CPU的5分钟平均利用率。

Cisco CPU MIB

第一步工作是编写一个SNMP应用程序获取供货商的MIB数据，以便用于监视。

幸运的是，Cisco为其网络设备提供了大量的MIB文件，可以从<ftp://ftp.cisco.com/pub/mibs/v1/>免费下载。不同的MIB种类和设备定义了不同MIB文件。通常情况下，在Cisco主页进行快速搜索可以得到用户希望的数据。包含CPU利用率信息的MIB信息包含在OLD-CISCO-CPU-MIB.my文件中，该文件定义用于监视CPU统计的MIB。

说明：有一个新的MIB文件可以用于监视CPU的统计，但是MIB无法保证可以适用于所有的Cisco路由器。OLD-CISCO-CPU-MIB.my文件可以适用于所有的Cisco路由器。

MIB文件使用ASN.1语法定义每个MIB对象和对象标识符。MIB文件的顶部列出了MIB

中的高级别对象。根据用户监视CPU利用率的频率，用户可以使用三个MIB对象中的一个，如表12.4所示。

说明：一些供货商MIB（例如Cisco）使用UNIX文本格式，而不是Dos文本格式。所以每行的结尾使用换行符。如果用户使用记事本进行观察，会觉得看上去很奇怪。用户可以通过使用微软的Word程序来观察MIB。

表12.4 Cisco CPU利用率MIB

MIB对象	说明
busyPer	CPU 5秒钟平均利用率
avgbusy1	CPU 1分钟平均利用率
avgbusy5	CPU 5分钟平均利用率

每个MIB对象都具有惟一的对象标识符，用户获取存储在设备数据库中的值。ASN.1对象定义显示每个对象的节点值，但不是整个对象标识符。程序清单12.4显示了使用的三个MIB值。

程序清单12.4 Cisco CPU利用率MIB

```
busyPer OBJECT-TYPE
    SYNTAX  INTEGER
    ACCESS  read-only
    STATUS  mandatory
    DESCRIPTION
        "CPU busy percentage in the last 5 second
        period. Not the last 5 realtime seconds but
        the last 5 second period in the scheduler."
    ::= { lcpu 56 }

avgBusy1 OBJECT-TYPE
    SYNTAX  INTEGER
    ACCESS  read-only
    STATUS  mandatory
    DESCRIPTION
        "1 minute exponentially-decayed moving
        average of the CPU busy percentage."
    ::= { lcpu 57 }

avgBusy5 OBJECT-TYPE
    SYNTAX  INTEGER
    ACCESS  read-only
    STATUS  mandatory
    DESCRIPTION
```

```
"5 minute exponentially-decayed moving
average of the CPU busy percentage."
 ::= { lcpu 58 }
```

ASN.1注释定义了5个单独值:

- 数据类型 (称为SYNTAX)
- 访问方式
- 在每个设备中, 对象是任意的还是强制的
- 对象的文本描述
- 对象标识符的ASN.1定义

MIB指出, avgBusy5对象的节点值为58, 并且是lcpu对象的子节点。该对象本身不提供足够的信息来定义对象的标识符。用户必须进行一些收集工作来确定完整的对象标识符。

要得到对象的完整标识符, 用户可以通过跟踪MIB来发现每个对象的标识符。lcpu在同一个MIB文件中定义, 定义如下:

```
lcpu OBJECT IDENTIFIER ::= { local 1 }
```

lcpu对象定义为节点1, 在local对象下。所以用户至少知道了avgBusy5对象的标识符结尾为1.58。

不幸的是, 本地对象没有在MIB文件中定义。然而, 在MIB文件的注释中, 用户可以看到本地对象在CISCO-SMI MIB文件中的参考。要得到剩余的MIB对象标识符, 用户必须下载CISCO-SMI-VISMI.my文件。在这个MIB文件中, local对象定义如下:

```
local OBJECT IDENTIFIER ::= { cisco 2 }
```

local对象是cisco对象下的子节点2。

cisco对象同样定义在同一个MIB文件中, 如下:

```
cisco OBJECT IDENTIFIER ::= { enterprises 9 }
```

好的, 现在用户已经得到了。cisco对象是enterprise对象的子节点9。enterprise对象是一个标准的MIB对象, 对象标识符为1.3.6.1.4.1。

将所有这些集中到一起, avgBusy5对象的整个标识符为1.3.6.1.4.1.9.2.1.58.0。现在, 用户知道了要查询的MIB对象, 用户可以编写SNMP程序, 在5分钟的间隔内查询交换机的avgBusy5 MIB对象。

CiscoRouter程序

程序清单12.5中的CiscoRouter.cs程序使用了avgBusy5 MIB对象得到5分钟的CPU利用率平均值。该值将每5分钟返回一次, 用户将看到一个时期CPU的利用率。程序在窗口中显示信息, 并且在日志文件中进行记录。

程序清单12.5 CiscoRouter.cs程序

```
using System;
using System.Drawing;
using System.IO;
using System.Threading;
using System.Windows.Forms;

class CiscoRouter Form
{
    private TextBox host;
    private TextBox community;
    private ListBox results;
    private Thread monitor;
    private FileStream fs;
    private StreamWriter sw;

    public CiscoRouter()
    {
        Text = "Cisco Router Utilization";
        Size = new Size(400, 380);

        Label label1 = new Label();
        label1.Parent = this;
        label1.Text = "Host:";
        label1.AutoSize = true;
        label1.Location = new Point(10, 30);

        host = new TextBox();
        host.Parent = this;
        host.Size = new Size(170, 2 * Font.Height);
        host.Location = new Point(40, 27);

        Label label2 = new Label();
        label2.Parent = this;
        label2.Text = "Community:";
        label2.AutoSize = true;
        label2.Location = new Point(10, 60);

        community = new TextBox();
        community.Parent = this;
        community.Size = new Size(170, 2 * Font.Height);
        community.Location = new Point(75, 57);

        results = new ListBox();
        results.Parent = this;
        results.Location = new Point(10, 85);
        results.Size = new Size(360, 18 * Font.Height);
```



```

    Button start = new Button();
    start.Parent = this;
    start.Text = "Start";
    start.Location = new Point(250, 52);
    start.Size = new Size(5 * Font.Height, 2 * Font.Height);
    start.Click += new EventHandler(ButtonStartOnClick);

    Button stop = new Button();
    stop.Parent = this;
    stop.Text = "Stop";
    stop.Location = new Point(320, 52);
    stop.Size = new Size(5 * Font.Height, 2 * Font.Height);
    stop.Click += new EventHandler(ButtonStopOnClick);
}

void ButtonStartOnClick(Object obj, EventArgs ea)
{
    monitor = new Thread(new ThreadStart(checkRouter));
    monitor.IsBackground = true;
    monitor.Start();
}

void ButtonStopOnClick(Object obj, EventArgs ea)
{
    monitor.Abort();
    sw.Close();
    fs.Close();
}

void checkRouter()
{
    int commlength, miblength, datastart, cpuUtil;
    SNMP conn = new SNMP();
    byte[] response = new byte[1024];
    DateTime time;
    string logFile = "routerlog.txt";
    fs = new FileStream(logFile, FileMode.OpenOrCreate,
        FileAccess.ReadWrite);
    sw = new StreamWriter(fs);

    while (true)
    {
        response = conn.get("get", host.Text,
            community.Text, "1.3.6.1.4.1.9.2.1.58.0");
        if (response[0] == 0xff)
        {
            results.Items.Add("No reponse from host");
        }
    }
}

```

```
        sw.WriteLine("No response from host");
        sw.Flush();
        break;
    }
    commlength = Convert.ToInt16(response[6]);
    miblength = Convert.ToInt16(response[23 + commlength]);
    datastart = 26 + commlength + miblength;

    cpuUtil = Convert.ToInt16(response[datastart]);
    time = DateTime.Now;
    results.Items.Add(time + " CPU Utilization: " + cpuUtil + "%");
    sw.WriteLine("{0} CPU Utilization: {1}%", time, cpuUtil);
    sw.Flush();
    Thread.Sleep(5 * 60000);
}
}

public static void Main()
{
    Application.Run(new CiscoRouter());
}
}
```

CiscoRouter程序使用两个**TextBox**对象来允许客户输入远程设备的主机名称（或者地址）和正确的团体名称，进行只读访问。当信息输入后，单击**Start**按钮。接下来，将创建一个新的**Thread**对象，该对象创建一个**SNMP**对象，并且使用**avgBusy5**对象标识符创建一个**SNMP**包。返回的包包含**CUP**利用率值。因为这个值不会大于**100**，用户知道该值将会放置在一个字节中，并可以转换为整形值。除了**CPU**利用率，还有当前的日期和时间，都显示在**Listbox**对象中，并记录到日志文件。图12.3显示了**CiscoRouter**应用程序运行的例子。

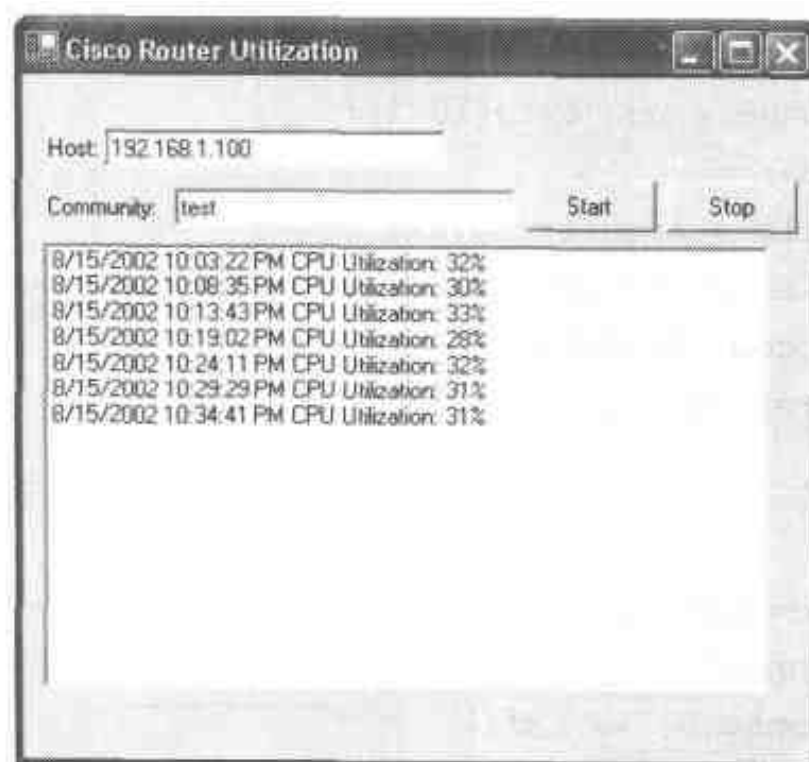


图12.3 CiscoRouter应用程序

使用GetNextRequest查询

如在SimpleSNMP和CiscoRouter程序中所见，使用SNMP类编写GetRequest程序是非常简单的。下一步就是使用SNMP GetNextRequest进行查询试验。这些查询和以前我们看到的GetRequest查询略有不同。

提取下一个MIB

GetNextResponse PDU用于用户从网络设备获取一系列数据。这些数据通常采用一些方法使用，例如安装在设备上的一系列网络结构，或者某一段时间的统计。

数据串使用单独的对象存储在MIB中，每个数据都定位于原始对象下。例如，interface对象可能有几个子节点，每个子节点都对应于设备上的一个接口。图12.4显示了这种情况。

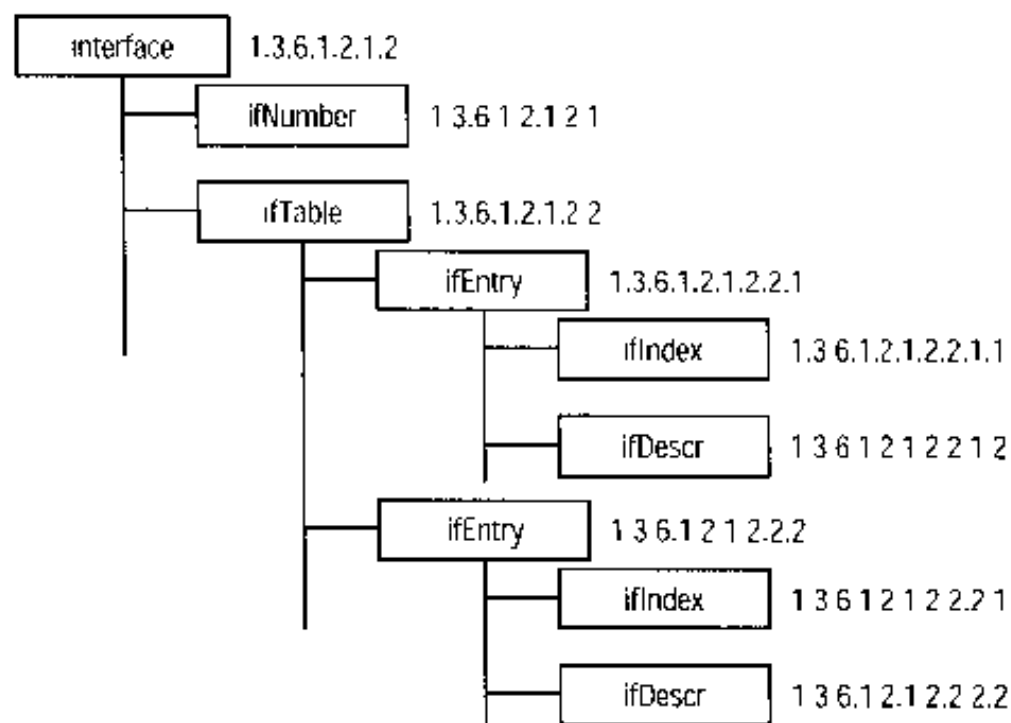


图12.4 接口对象在MIB数据库中定义单独的接口

MIB数据库包含用于接口表（ifTable）的对象，位于对象标识符1.3.6.1.2.1.2.2。按照每个设备的顺序，在序列中查询每个接口对象，必须从ifTable对象开始，并且“步行”过数据库中相关的子节点。这样做对SNMP设备可以起到帮助作用。

当GetNextRequest包发送到网络设备，设备使用标准的GetResponse包进行响应。不同之处在于包中的MIB条目值发生了改变。新的MIB条目是下一个MIB，将被一系列信息继续查询。

要跟随这个流程，用户必须从GetResponse包中提取新的MIB值，并且作为下一个GetNextRequest查询的MIB值。这一过程将持续，直到远程设备返回错误状态，或者用户决定终止收集数据。

getnextMIB()方法

因为从GetResponse包中提取下一个MIB值是非常通用的功能，因此最好在SNMP类中创

建一个新的方法。任何SNMP应用程序都可以调用该方法。在get()方法后，程序清单12.6显示的getNextMIB()方法将加入SNMP.cs类文件中。

程序清单12.6 SNMP类的getNextMIB()方法

```
public string getNextMIB(byte[] mibin)
{
    string output = "1.3";
    int commlength = mibin[6];
    int mibstart = 6 + commlength + 17; //find the start of the mib section
    //The MIB length is the length defined in the SNMP packet
    // minus 1 to remove the ending .0, which is not used
    int miblength = mibin[mibstart] - 1;
    mibstart += 2; //skip over the length and 0x2b values
    int mibvalue;

    for(int i = mibstart; i < mibstart + miblength; i++)
    {
        mibvalue = Convert.ToInt16(mibin[i]);
        if (mibvalue > 128)
        {
            mibvalue = (mibvalue/128)*128 + Convert.ToInt16(mibin[i+1]);
            i++;
        }
        output += "." + mibvalue;
    }
    return output;
}
```

getNextMIB()进行和get()方法相反的操作。这个方法使用字节数组表示MIB并且将其转换为字符串值。令人感到棘手的是，要记住观察对象节点值，如果节点值大于128，将占用2个字节，而不是通常情况下的1个字节。

当MIB转换为字符串，将返回调用的应用程序。应用程序将新的MIB进行显示，并在另一个get()方法中使用，来检测对象的MIB值。

MAC地址程序

这里有一些应用程序，需要在网络商标上进行单步调试。我发现的一个应用程序，用于检测Cisco交换机的MAC地址。Cisco交换机为每个配置的VLAN保存MAC地址轨迹，这些交换机可以在interface对象的公共MIB中看到。使用SNMP类，可以非常简单地编写应用程序，对整个表进行操作，获取VLAN的所有MAC地址。

程序清单12.7显示了MacAddress.cs程序，该程序查询Cisco交换机，获取所有的MAC地址。

程序清单12.7 MacAddress.cs程序

```

using System;

class MacAddress
{
    public static void Main(string[] argv)
    {
        int commlength, miblength, datastart, datalength;
        string nextmib, value;
        SNMP conn = new SNMP();
        string mib = "1.3.6.1.2.1.17.4.3.1.1";
        int orgmiblength = mib.Length;
        byte[] response = new byte[1024];

        nextmib = mib;

        while (true)
        {
            response = conn.get("getnext", argv[0], argv[1], nextmib);
            commlength = Convert.ToInt16(response[6]);
            miblength = Convert.ToInt16(response[23 + commlength]);
            datalength = Convert.ToInt16(response[25 + commlength + miblength]);
            datastart = 26 + commlength + miblength;
            value = BitConverter.ToString(response, datastart, datalength);
            nextmib = conn.getNextMIB(response);

            if (!(nextmib.Substring(0, orgmiblength) == mib))
                break;

            Console.WriteLine("{0} = {1}", nextmib, value);
        }
    }
}

```

令人惊讶的是，应用程序非常小。对于应用程序来说，令人棘手的问题就是了解什么时候停止对**MIB**对象进行操作。

Cisco交换机使用dot1netTable (1.3.6.1.2.1.17.4.3.1.1) 对象，位于mib-2数据库 (1.3.6.1.2.1)，可以访问MAC地址表。一旦用户引用表的基本对象，Cisco交换机将用户指向返回的**MIB**单独值。

交换机将持续在用户指定的**MIB**中进行步行，但是在运行时，交换机将试图到同一级别的下一个**MIB**对象。要防止这种情况，用户必须保存原始**MIB**对象标识符的副本，并且和**GetResponse**包中返回的新**MIB**进行比较。在对象标识符中，每个子节点将拥有更多的节点值，所以用户需要做的就是比较原始**MIB**和新**MIB**的长度。当新**MIB**和原始**MIB**长度发生了改变，用户可以知道返回了同一级别的新对象，用户可以停止步行。

举个例子，原始MIB值如下：

```
1.3.6.1.2.1.17.4.3.1.1
```

在这个表中的第一个数据对象具有下面的MIB值：

```
1.3.6.1.2.1.17.4.3.1.1.0.0.12.7.1/2.124
```

随后的值使用深度优先方法：低级的MIB位置得到扩展，直到这些值被穷举，然后高级别的MIB位置也得到扩展，直到基本MIB值改变如下：

```
1.3.6.1.2.1.17.4.3.1.2
```

当基本的MIB值改变，程序将检测到，并且停止搜索。

因为MAC地址通常使用十六进制值，用户可以使用BitConverter类的ToString()方法产生一个字符串版本十六位MAC地址。

测试程序

在编译MacAddress程序后（使用新版本的SNMP类文件），用户可以使用一个Cisco交换机运行该程序。对于Cisco设备来说，一个非常奇怪的问题就是该设备处理团体名称的方法。

Cisco使用一个被称为索引团体名称的系统。普通的团体名称，例如test，只能访问普通的交换机信息。要访问某个特殊VLAN的信息，用户必须在团体名称中增加VLAN号码。例如，要访问MAC地址表为VLAN100，用户需要使用团体名称test@100。

程序清单12.8显示了一个繁忙的Cisco交换机上运行MacAddress程序的输出片段。

程序清单12.8 MacAddress程序输出例子

```
C:\>MacAddress 192.168.1.5 horse@100
1.3.6.1.2.1.17.4.3.1.1.0.0.12.7.172.124 = 00-00-0C-07-AC-7C
1.3.6.1.2.1.17.4.3.1.1.0.0.170.91.20.67 = 00-00-AA-5B-14-43
1.3.6.1.2.1.17.4.3.1.1.0.0.170.91.23.14 = 00-00-AA-5B-17-0E
1.3.6.1.2.1.17.4.3.1.1.0.0.170.91.24.202 = 00-00-AA-5B-18-CA
1.3.6.1.2.1.17.4.3.1.1.0.1.3.190.158.77 = 00-01-03-BE-9E-4D
1.3.6.1.2.1.17.4.3.1.1.0.1.230.77.209.187 = 00-01-E6-4D-D1-BB
1.3.6.1.2.1.17.4.3.1.1.0.1.230.84.119.202 = 00-01-E6-54-77-CA
1.3.6.1.2.1.17.4.3.1.1.0.4.0.248.44.12 = 00-04-00-F8-2C-0C
1.3.6.1.2.1.17.4.3.1.1.0.6.91.52.17.87 = 00-06-5B-34-11-57
1.3.6.1.2.1.17.4.3.1.1.0.6.91.52.108.108 = 00-06-5B-34-6C-6C
1.3.6.1.2.1.17.4.3.1.1.0.6.91.129.249.64 = 00-06-5B-81-F9-40
1.3.6.1.2.1.17.4.3.1.1.0.6.91.130.3.112 = 00-06-5B-82-03-70
1.3.6.1.2.1.17.4.3.1.1.0.6.91.166.250.231 = 00-06-5B-A6-FA-E7
1.3.6.1.2.1.17.4.3.1.1.0.16.131.13.143.121 = 00-10-83-0D-8F-79
1.3.6.1.2.1.17.4.3.1.1.0.80.218.16.120.6 = 00-50-DA-10-78-06
```

```
.  
. 1.3.6.1.2.1.17.4.3.1.1.0.224.41.146.58.140 = 00-E0-29-92-3A-8C  
1.3.6.1.2.1.17.4.3.1.1.8.0.9.216.30.170 = 08-00-09-D8-1E-AA  
1.3.6.1.2.1.17.4.3.1.1.8.0.32.117.204.112 = 08-00-20-75-CC-70  
C:\>
```

从Cisco交换机返回的MIB显示了一些有趣的值。注意，在返回的MIB值中，没有特殊的顺序。只有Cisco知道每个节点值的意义——我们所知道的只是这些值表示一个MAC地址值。

值得注意的其他功能是，当返回的MIB对象标识符不同于原始MIB对象标识符，程序将被中断。

小结

对于程序开发者来说，使用简单网络管理协议（SNMP）编写应用程序是一个挑战。因为该协议非常复杂，很多新的程序开发者要避免使用该协议。本章为用户提供了一个了解SNMP的机会，并且理解了如何使用SNMP协议创建C#网络程序，对网络设备进行查询。

SNMP允许网络设备维持一个网络状态的数据库，并进行配置和设置。数据库可以在网络远程的设备上，通过SNMP网络协议进行查询和操作。每个SNMP设备必须执行公共管理信息库（称为MIB-II），跟踪设备接口的网络状态。除了公共MIB，很多销售商都提供他们自己的数据库对象，可以从远程工作站对网络上的设备进行控制。

SNMP使用5种类型的命令包。GetRequest和GetNextRequest包用于从设备上获取单值或者一系列值。GetResponse包返回查询中请求的信息。SetRequest包改变数据库对象的值。Trap包用于网络设备发送未经请求的网络信息包到中心网络监视站。

每个SNMP使用访问方式和用户简档来确保安全。团体名称在MIB数据库中享有特权。远程设备使用团体名称作为密码，查询和更新数据库中的值。

SNMP包很难创建，因为包中包含不同长度的信息。包含在SNMP包中的团体名称和MIB对象的长度会发生变化。SNMP命令控制在协议数据单元（PDU）中。每个PDU代表一个单独的查询，但是可以包含多个MIB对象和对象值。

在本章中，创建了一个SNMP类，允许程序开发者提供一种创建SNMP包并发送到远程设备的方法。返回的响应包被转发到应用程序进行译码。

用户学习了三个网络应用程序例子：一个简单的SNMP应用程序，从网络设备获取系统信息，一个应用程序用于监视Cisco路由器CPU的利用率，一个应用程序获取Cisco交换机的MAC地址表。

接下来，我们将转移到因特网E-mail。.NET网络库提供一个类，来帮助网络编程人员在网络应用程序中集成因特网E-mail。第13章“SMTP”将显示如何使用.NET Mail类通过用户程序和因特网发送电子邮件。

第13章 SMTP

- 电子邮件基础
- SMTP与Windows
- Smtplib类
- 使用扩展邮件消息格式
- MailMessage类
- 邮件附件
- MailAttachment类
- POP3客户机

前面两章讲述了如何使用标准C#网络类库进行网络编程，本章则介绍一个独立的C#网络命名空间，它包含了编写电子邮件网络程序的类。

`System.Web.Mail`命名空间中的类可以用于创建新邮件并发送到远程主机，它既可以使用Windows 2000和XP中提供的Windows SMTP（Simple Mail Transfer Protocol，简单邮件传输协议）服务，又可以使用其他邮件服务器。

本章讲述因特网电子邮件系统的原理，以及如何对邮件格式化，使其向远程主机发送文本和二进制数据。本章还要介绍如何使用.NET电子邮件类为C#网络程序增加电子邮件功能。通过介绍POP（Post Office Protocol，邮局协议）这个电子邮件客户机协议，并辅以一个实例，读者可以掌握在C#网络程序中设计电子邮件客户机功能的方法。

电子邮件基础

几乎所有的因特网电子邮件软件都使用UNIX电子邮件模式实现与远程主机之间的通信，在向本地用户和远程用户发送消息方面，这种模式已经非常流行并且被广泛应用。

UNIX电子邮件模式将电子邮件的功能分为以下3个部分：

- MTA（Message Transfer Agent，报文发送代理）
- MDA（Message Delivery Agent，报文传输代理）
- MUA（Message User Agent，报文用户代理）

电子邮件程序中的发送、传递和显示功能由上面3个部分分别完成。图13.1展示了它们之间的关联。

MTA进程

MTA进程用来接收和发送邮件，其功能也分成两部分：

- 决定向哪里以及如何发送邮件

- 决定向哪里传递接收到的邮件

上述两个功能的处理过程各不相同。

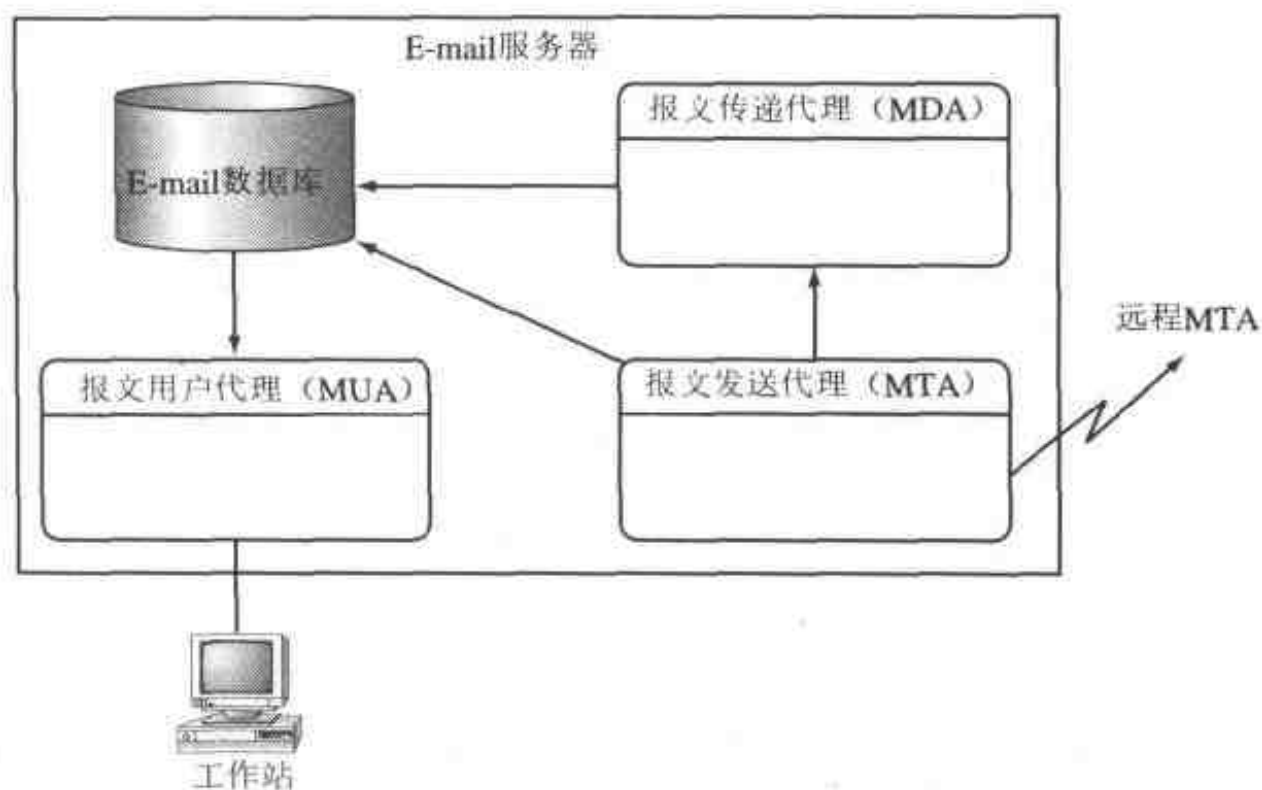


图13.1 UNIX电子邮件模式

发送邮件

对于每一封外发邮件，**MTA**必须根据其接收地址确定发送目的地。如果是发往本地的邮件，**MTA**可以直接将其传送到本地邮件系统，或者将此邮件移交给**MDA**来传递。

如果目的地是远程域，那么**MTA**必须完成两项工作：

- 检查DNS中的MX记录，确定该域的邮件服务器
- 与远程邮件服务器建立连接并传送邮件

与远程邮件服务器建立连接时通常使用**SMTP**，因特网上不同的主机系统可以通过这个标准协议所实现的通用通信技术传递邮件。

接收邮件

MTA的功能是接收来自本地系统用户和远程主机用户的邮件。此功能看似简单，但很可能产生一些意外情况。程序必须仔细检查邮件的接收方地址，并决定是否由本地系统发送该邮件。

邮件可以使用三种目标地址：本地系统账户、本地别名账户和远程用户账户。

本地系统账户 每种邮件系统，无论Windows、UNIX或者Macintosh都有其自己定义的可以访问系统的本地账户。**MTA**必须能识别出发往本地账户的邮件，将其直接转发到用户邮箱或者交给**MDA**处理。

本地别名账户 很多**MTA**程序允许使用用户别名。别名本身不能收发邮件，它的作用相当于一个指针，指向一个或多个可以收发邮件的系统用户的账户。一旦**MTA**确定别名有效，它会将目标地址转换为真实系统用户的账户名，然后发送该邮件或者交给**MDA**处理。

远程用户账户 很多MTA程序可以接收那些并非发往本地系统用户的账户的邮件。从传递邮件的角度看，这实在令人匪夷所思，而且一直以来，人们对于MTA处理发往远程用户账户的邮件的方式存在很多批评和争论。这项技术被称为邮件转发（replaying），邮件服务器收到发往远程用户的邮件后，会自动转发给远程用户。对于很多ISP来说，这一功能很有用，因为拨号用户无法直接向远程主机发送因特网邮件，只能把邮件先发到ISP的邮件服务器上，再由服务器转发到远程目标。不幸的是，邮件转发容易被滥用，个别不道德的人会利用转发邮件服务器隐藏他们的原始地址，继而向远程邮件用户发送大量广告邮件。这些信被称为Unsolicited Commercial E-mail（UCE）、Unsolicited Bulk E-mail（UBE）或者spam（垃圾邮件），它们非常令人讨厌，邮件管理员必须阻止这种行为。

说明：spam一词有可能源自一出Monty Python滑稽剧中散播SPAM肉罐头的剧情。SPAM肉罐头由Hormel公司生产，Hormel公司承诺，除非完全用大写字母书写，否则使用spam一词并不造成商标侵权（详见http://www.spam.com/ci/ci_in.htm）。

为了阻止垃圾邮件，如今很多ISP使用选择性转发，只允许某段IP地址或者授权的拨号用户使用他们的邮件服务器转发邮件，阻止其他上机通过服务器转发邮件。

MDA进程

MDA的主要作用是传递发往本地系统用户账户的邮件。为了实现这一功能，MDA必须知道每个用户的邮箱类型和位置。很多电子邮件系统使用数据库来存储本地用户的邮件。MDA必须有足够的权限将接收到的邮件存储在用户邮箱中。

除了发送邮件，很多MDA具备一些更高级的功能：

自动过滤邮件 对接收邮件加以过滤，这可能是MDA最重要的和最受欢迎的特性。对于邮件量很大的用户而言，这一特性能起到秘书的作用，根据邮件标题甚至标题中包含的一个字，就可以将邮件自动存入不同的文件夹。

很多MDA允许邮件管理员定义全局过滤器来阻止垃圾邮件和病毒邮件。该功能通过使用标准文本通配符表达式来实现，每个表达式与接收的邮件标题报头做匹配，符合表达式的邮件会被剔除。

自动回复邮件 很多MDA允许用户为电子邮件定义一个自动回复邮件，一些自动回复邮件可以设置为自动回复某一个用户收到的全部邮件，其功能好比是电话答录机。有些自动回复邮件则像邮件过滤器那样，只自动回复标题报头中含有特定文字的邮件，当收到的邮件与文字表达式匹配的时候，就自动发送特定的回复邮件。

自动运行程序 MDA的另一功能是根据收到的邮件自动启动程序。当然，必须严格控制这一功能，以防滥用。不过只要合理使用，这的确是个便捷的工具。管理员可以用它来远程启动服务器程序，或者在服务器以外的其他机器上修改服务器的设置。

MUA进程

远离邮件服务器的用户可以通过MUA访问邮箱。对MUA的误解之一是它们可以发送邮件，MUA其实只能读取邮件，它并不发送和接收邮件。一些MUA程序中会包含一个小型

MTA，提供向邮件服务器转发新邮件的功能，这可能是误解的来源。

在MUA之下，有两个体系负责读取和保存接收到的邮件。

在客户机存储邮件

一些ISP希望用户将邮件直接下载到客户机，因为邮件被下载后就会从邮件服务器上删除，这有助于邮件管理员维护服务器的磁盘空间并且防止邮件阻塞磁盘系统。

实现这种邮件访问方式的协议是POP（通常是POP第3版，记为POP3）。POP3的MUA允许用户连接到服务器并且下载所有邮件到客户机上的邮箱。如图13.2所示。

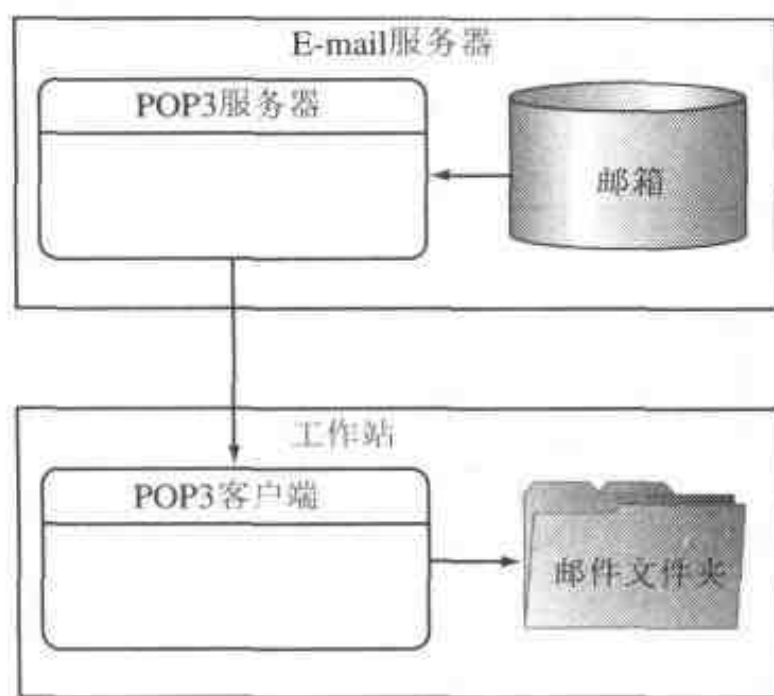


图13.2 用POP3软件下载邮件

用POP3的缺点是用户从哪里连接邮件服务器，邮件就保存在哪个客户机上。如果用户使用多个客户机连接，邮件就分开保存在不同客户机上。这给用户读取下载邮件造成不便。

说明：一些ISP可能允许用户用POP3下载邮件以后不从服务器上删除。这并不是规范的做法。请不要与后文描述的IMAP特性混淆。

在服务器上保存邮件

除了POP3访问方式以外，还可以选择IMAP（Interactive Mail Access Protocol，交互式邮件访问协议，通常用的版本是IMAPrev4）。IMAP允许用户在邮件服务器上建立文件夹保存邮件而不下载邮件到客户机上。如图13.3所示。因为邮件保存在服务器上，用户可以从任何客户机上连接服务器并且在文件夹中看到相同的邮件。

此系统的缺点是邮件管理员要更多关注服务器磁盘空间的耗费。现如今的邮件和附件占满系统磁盘并不需要很长时间。

SMTP与Windows

.NET的mail库使用了Microsoft CDO SYS消息组件，可以通过SMTP向远程主机发送邮

件。但是，Windows操作系统将从何时开始以及将在多大程度上支持CDOSYS，CDOSYS会对.NET产生多大的影响，都还没有定论。本节将对这两个问题进行一些讨论，并解答关于CDOSYS的一些问题。

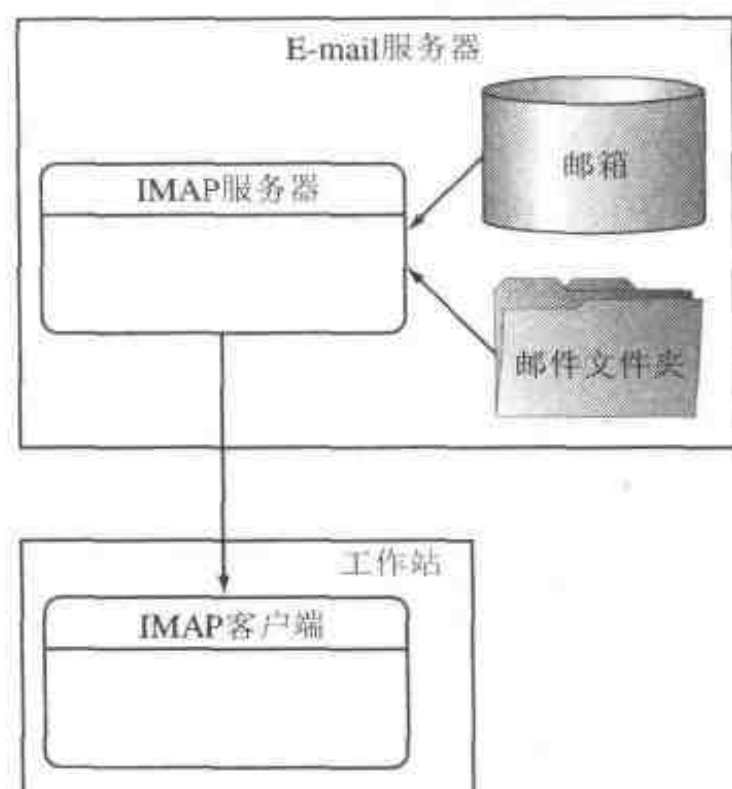


图13.3 用IMAP在服务器文件夹中保存邮件

协作数据对象（CDO）

Microsoft在不同平台上支持因特网电子邮件已有相当长的历史。要了解.NET邮件功能的情况，必须先了解Microsoft支持邮件功能的历史。它是从Microsoft Exchange 5邮件系统和Active Messaging库（OLEMSG32.DLL）开始的，该库允许程序员编写代码，利用Exchange邮件系统向每个用户邮箱，以及Exchange网络中的其他远程Exchange主机发送邮件。

Exchange 5.5发布的时候，出现了一种新的邮件系统，这就是CDO（Collaboration Data Objects，协作数据对象）。不同于Active Messaging，CDO使用MAPI（Message Application Program Interface，邮件应用程序接口）标准为程序员提供更简单的发送邮件的办法。CDO应用了好几年，其间推出了1.1、1.2和1.2.1三个升级版本。与Microsoft NT Server平台一同问世的CDO NT Server库（CDONTS），放弃了MAPI标准，改用诸如SMTP的因特网标准在远程主机之间传递邮件。

当前的CDO版本（CDO 2，又名CDO For Windows 2000）是随Windows 2000平台发布的，也可以在Windows XP平台应用。它使用因特网标准向程序员提供邮件和新闻服务库，扩展了CDONTS库。CDO 2的另一个新功能是可以处理多附件邮件和多种邮件协议。这一特性让程序员可以很容易地设计全能邮件服务器。CDO 2极大地扩展了因特网邮件功能，.NET也用它来实现因特网邮件功能。

在Windows 2000和XP平台，CDO 2库是文件CDOSYS.DLL，.NET的mail库函数必须使用该文件。由于CDO 2与CDO 1.x版本完全不同，所以两个版本的库可以并存在同一台主机上。CDO 2库不能向下兼容以往的Windows操作系统。

警告：System.Web.Mail类不能工作在没有CDOSYS.DLL文件的Windows平台上。这类平台包括Windows 95、98、Me和NT客户机。

SMTP邮件服务

Windows 2000和XP平台都提供了一个基本SMTP服务器，可以使用SMTP协议收发邮件。该功能是IIS（Internet Information Services，因特网信息服务）包的一部分。NTE mail类可以利用IIS SMTP服务器直接向远程主机发送邮件。

Windows XP系统包括了IIS 5.1版，该版本包含SMTP服务在内的很多东西。在使用SMTP服务以前，必须根据因特网邮件环境正确设置。一般通过Computer Management窗口进行设置。

以下是访问Default SMTP Virtual Server Properties窗口的步骤：

1. 在开始菜单中右键单击My Computer，选择Manage菜单项。
2. 当出现Computer Management窗口时，展开Services and Applications，然后是Internet Information Services。
3. 右键单击Default SMTP Virtual Server并且选择Properties。

Properties窗口就是设置SMTP服务的地方。

在General页选择允许接收SMTP连接的网卡，规定允许SMTP同时连接的数量。在这一页也可以启动或者关闭SMTP服务的日志。

在Delivery页（图13.4），设置发送重试次数。单击Advanced按钮可以为邮件服务器设置一个智能主机。这很重要，因为有些ISP不允许通过SMTP直接与远程主机通信。在这种情况下通常使用ISP的邮件服务器做远程邮件转发。这就必须把ISP的邮件服务器地址作为智能主机的地址，SMTP服务会通过该邮件服务器转发所有的邮件。



图13.4 设置SMTP的Default SMTP Virtual Server Properties对话框

SmtpMail类

System.Web.Mail命名空间中生成的SmtpMail类可用于在C#网络程序中发送SMTP邮件。本节讨论SmtpMail类的方法和属性，并且给出一些C#典型应用的实例。

类的方法和属性

SmtpMail类在Windows 2000和XP系统上提供了.NET到CDOSYS mail库的接口。它没有用构造器来生成类的实例，读者必须使用静态的类的方法和属性向CDOSYS库传递信息。

SmtpMail类包含标准的Equals()、GetHashCode()和ToString()方法，不过很少使用。另一方面，Send()方法用得比较多，有以下两种格式：

```
Send(MailMessage message)
Send(string from, string to, string subject, string body)
```

第一种格式允许发送MailMessage对象。MailMessage类是一种自含的电子邮件，是组合了邮件和目标地址这些类的属性生成的。本书将在“使用MailMessage类”这一节介绍这一过程。

第二种格式允许发送原始信息，手工定制电子邮件报头字段：

- 编写电子邮件发件人地址。
- 编写收件人地址，多个收件人地址之间用逗号隔开。
- 编写电子邮件主题。

Send()方法的最后一个参数是邮件信体本身，信体可以采用普通文本或者HTML格式。

SmtpMail类的唯一属性是**SmtpServer**。这是个静态属性，指定邮件转发服务器地址，用以转发邮件。在默认情况下，如果安装了IIS SMTP服务，**SmtpServer**属性就按此设置。如果没安装IIS，**SmtpServer**属性就为空，试图发送邮件的时候会报错。

如果使用转发邮件服务器的话，必须要在发送邮件以前设置**SmtpServer**属性：

```
SmtpMail.SmtpServer = "mailsrvr.myisp.net";
```

设置好该属性，所有的外发邮件将通过此邮件服务器转发。当然必须确认转发邮件服务器允许通过它来转发邮件，否则信会被拒绝发送。

使用SmtpMail类

程序清单13.1列出的MailTest.cs程序演示如何生成一封简单的邮件，并且通过远程邮件转发服务器发送给接收者。

程序清单13.1 MailTest.cs程序

```
using System;
using System.Net;
using System.Web.Mail;

class MailTest
{
    .
    public static void Main()
    {
        string from = "jessica@myisp.net";
        string to = "katie@anotherisp.net";
```

```
        string subject = "This is a test mail message";  
        string body = "Hi Katie, I hope things are going well today.";  
  
        Smtplib.SmtpServer = "192.168.1.150";  
        Smtplib.Send(from, to, subject, body);  
    }  
}
```

这个MailTest程序非常简单。它的作用就是使用Smtplib属性定义转发邮件服务器、定义每个必需的字段，并且用Send()方法发送邮件。如果这些全部实现，指定的收件人就会在邮箱中收到这封信。

如果在没有使用CDO 2库的Windows平台上运行以上程序，就会收到错误提示，平台不同，消息也不尽相同。以Windows NT 4为例，会出现如下信息：

C:\>MailTest

```
Unhandled Exception: System.Web.HttpException: Could not create 'CDONTS.NewMail'  
object.  
    at System.Web.Mail.LateBoundAccessHelper.get_LateBoundType()  
    at System.Web.Mail.CdoNtsHelper.Send(String from, String to, String subject,  
String messageText)  
    at System.Web.Mail.SmtpMail.Send(String from, String to, String subject, Stri  
ng messageText)  
    at MailTest.Main()
```

C:\>

正如所看到的那样，SmtpMail试图在NT系统上使用CDONTS库但是失败了。所以编写需要在多个Windows平台上运行的代码时，最好生成自己的消息提示：

```
try  
{  
    Smtplib.SmtpServer = "192.168.1.150";  
    Smtplib.Send(from, to, subject, body);  
} catch (System.Web.HttpException)  
{  
    Console.WriteLine("Sorry, this application only works  
on Windows 2000 and XP platforms.");  
}
```

上面这个简单的程序使用Send()方法仅仅设置了邮件的From:、To:和Subject:报头。还有更多报头可以定义并且在电子邮件中使用。下一节描述了这些报头以及如何在邮件中应用。

使用扩展的邮件报文格式

电子邮件的流行带来兼容性问题。很多电子邮件系统按自己的方式格式化邮件，不久

以前有了一个标准电子邮件格式，允许不同电子邮件系统之间通信并且编译彼此的邮件。**RFC 2822**定义了标准电子邮件格式，**.NET MailMessage**类可以提供标准方式生成**RFC 2822**邮件发送给远程用户。

RFC 2822邮件格式

RFC 2822规定每封邮件要由下面两个部分组成：

- 包含邮件信息的报头
- 包含邮件内容的信体

RFC 2822标准邮件报头包含独立区域，提供邮件的附加信息（如图13.5所示）。报头位于邮件中的分割线以上，包含文本格式的数据。报头的格式如下（注意**header**与后面的文本数据以冒号分开）：

```
header: text data
```

独立的报头字段不需要在邮件中规定特别的顺序。报头必须位于信体前面而且必须以一行空格分开。

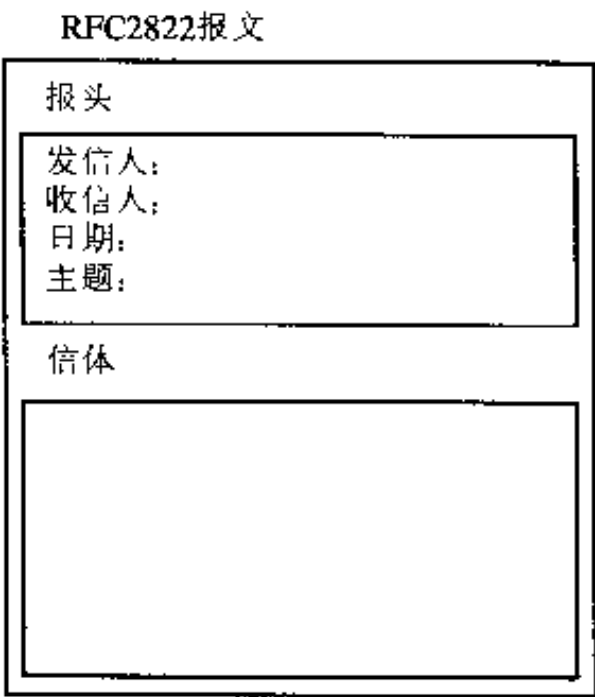


图13.5 RFC2822邮件格式

始发日期字段

始发日期字段记录邮件发出的日期和时间。它包含关键字**Date:**，后面是日期与时间：

```
Date: Wed, 21 Aug 2002 18:30:00 -0500
```

这非常重要，根据始发日期可以识别发送邮件的主机的时区，收信人可以确定邮件发送的真正时间。

作者字段

可以根据三个作者字段了解邮件是从哪里发来的：

```
From: Mailbo-list
Sender: mailbox
Reply-To: mailbox-list
```


在一封邮件中至少要有以上三项中的一项，**mailbox-list**可以包含单一的邮件地址，或者用逗号分开的多个邮件地址。

From项标识信的作者，而**Sender**项标识发送该信的人（这两项不必相同）。**Reply-To**项表示发信人希望收信人回复该邮件时使用的目的地址。

目标地址字段

以下三项目标地址项标识出邮件应该往哪里发送：

```
To: address-list  
Cc: address-list  
Bcc: address-list
```

每个目标地址项都可以在**address-list**中设定一个或者多个地址，多个地址之间用逗号分开。

To项标识邮件的主要接收人，**Cc**项指定抄送地址。

Bcc项是密送地址，其中的收件人地址不会在邮件中显示出来。**Bcc**项不同于其他目标地址项，尽管发送的邮件可能包含有**Bcc**项，它仍会在发送给目标地址之前被隐藏起来。

身份标识字段

身份标识字段可以唯一地标识邮件的邮件系统和收件人。身份标识字段包含很多项目，以下是几个较为常用的：

```
Message-ID: msg-id  
In Reply To: msg-id  
References: msg-id
```

Message-ID项赋予邮件唯一的跟踪码。每封由MTA发出的邮件都包含一个由MTA赋予的**Message-ID**项，是该邮件的唯一标识。**msg-id**由字母和数字组成，唯一标识邮件。

In-Reply-To和**References**项是关于先前的邮件的。通常回复邮件的时候原始邮件的**Message-ID**项会包含在**In-Reply-To**项中。

信息字段

信息项是可选的；它有助于收件人识别和过滤邮件。信息项如下：

```
Subject: subject-text  
Comments: comment-text  
Keywords: phrase-text
```

Subject项是最常见的一项。它用很短的描述邮件主题的字符串标识邮件。**Comments**项以长一些的字符串更进一步描述信体。**Keywords**项用于MDA过滤软件，为接收的邮件分类。

MailMessage类属性

.NET **MailMessage**类很容易生成RFC 2822格式的邮件并且使用**SmtpMail**类发送。在

SmtpMail.Send()方法允许的常规信息以外可以添加更多的报头信息。

MailMessage类默认的构造器很简单：

```
MailMessage newmessage = new MailMessage();
```

如上所示，无需记住特殊参数。所有的工作由类的属性完成。表13.1列出可以利用的属性。

表13.1 MailMessage类的属性

属性	说明
Attachments	指定加到邮件中的附件文件列表
Bcc	在Bcc:报头字段设置以分号分开的地址列表
Body	设置电子邮件信体
BodyEncoding	设置信体的编码方式
BodyFormat	设置信体内容格式
Cc	在Cc:报头字段设置以分号分开的地址列表
From	在From:报头字段设置地址
Headers	设置用户报头信息
Priority	设置邮件优先级
Subject	在Subject:报头字段设置文本字符串
To	在To:报头字段设置以分号分开的地址列表
UrlContentBase	读写HTTP报头中的Content-Base字段
UrlContentLocation	读写HTTP报头中的Content-Location字段

最常用的属性是To、From、Cc和Bcc。可以使用Header属性增添报头信息，不过这其实没有太大差别。

Header（报头）属性

Header属性使用IDictionary类存储报头信息。最简单的实现方式是使用Add()方法：

```
newmessage.Header.Add("Reply-To", "testing@myisp.net");
```

可以使用Header属性指定任何有效的RFC 2822报头项并将其包含在邮件报头中。用以下格式向邮件中添加日期项是非常容易的：

```
DateTime mydate = DateTime.Now;  
newmessage.Header.Add("Date", mydate.ToString());
```

Body（信体）属性

Body、BodyEncoding和BodyFormat属性用于指定邮件信体的内容和格式。邮件的文本

由Body属性指定。在远程邮件服务器上看到的邮件外观是由BodyEncoding和BodyFormat属性控制的。

BodyEncoding属性使用System.Text.Encoding类指定邮件使用的文本类型。这些类型包括ASCII、Unicode、UTF-7和UTF-8。默认情况下使用系统默认的编码方式（在美国使用ASCII）。

相对于文本编码，BodyFormat属性规定的是这些文本在邮件中如何表现。在MailFormat中列举了如何定义该属性：

MailFormat.Html 使用超文本格式

MailFormat.Text 使用纯文本格式

在默认状态下，通常使用Text格式。这可以确保每个电子邮件客户机都可以阅读该邮件。也可以选择Html格式在邮件中组合特殊的文本格式。

Priority（优先权）属性

Priority属性允许设置Priority报头字段。尽管这不是标准报头，很多邮件客户机仍然会识别出优先权报头，以此作为邮件重要性的标志。System.Web.Mail命名空间用枚举型变量MailPriority设置该属性。设置如下：

MailPriority.High 需要特别注意的重要邮件

MailPriority.Normal 普通邮件

MailPriority.Low 无关紧要的邮件，例如广告

设置邮件的优先级，只需要设置正确的Priority属性：

```
newmessage.Priority = MailPriority.High;
```

Attachment（附件）属性

Attachment属性确定需要作为附件包含在邮件中的文件。使用以下格式指定一个或者多个MailAttachment类对象作为附件：

```
MailAttachment ma = new MailAttachment("c:\\tempfile.bmp");  
newmessage.Attachment.Add(ma);
```

MailAttachment对象指定的文件经过编码加到邮件中，这一过程将在后面一节“邮件附件”中阐述。

使用MailMessage类

清单13.2中的FancyMailTest.cs程序使用MailMessage类生成一封邮件发送给接收者。

程序清单13.2 FancyMailTest.cs程序

```
using System;  
using System.Web.Mail;  
  
class FancyMailTest
```

```
{
    public static void Main()
    {
        MailMessage mm = new MailMessage();
        mm.From = "haley@myisp.net";
        mm.To = "riley@yourisp.net;rich@shadrach.ispnet1.net";
        mm.Cc = "matthew@anotherisp.net;chris@hisisp.net";
        mm.Bcc = "katie@herisp.net;jessica@herisp.net";
        mm.Subject = "This is a fancy test message";
        mm.Headers.Add("Reply-To", "haley@myisp.net");
        mm.Headers.Add("Comments", "This is a test HTML message");
        mm.Priority = MailPriority.High;
        mm.BodyFormat = MailFormat.Html;
        mm.Body = "<html><body><h1>This is a test message</h1><h2>This message  

        should have HTML-type formatting</h2>Please use an HTML-capable  

viewer.";

        try
        {
            Smtplib.Send(mm);
        } catch (System.Web.HttpException)
        {
            Console.WriteLine("This device is unable to send Internet messages");
        }
    }
}
```

FancyMailTest程序生成一个MailMessage对象并且给各种属性赋值。To、Cc和Bcc属性中设置了多个收件人地址，每个用分号分开。Priority属性设为High，并且在邮件中多加了2行报头。

由于BodyFormat类型设为Html，所以可以在信体文本中使用任何有效的HTML语句。图13.6展示了使用Microsoft Outlook Express邮件客户机看到的邮件。

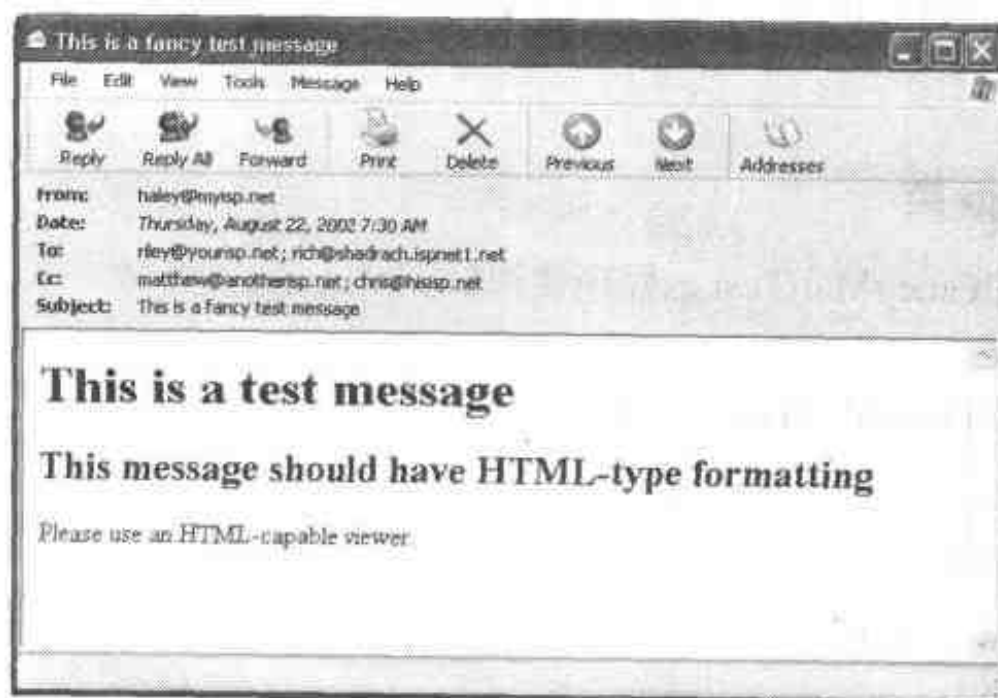


图13.6 使用Outlook Express阅读FancyMailTest邮件

邮件附件

信体属性只允许键入文本，另一个属性用来加入附件。这是由SMTP的工作原理和邮件服务器只能处理二进制数据决定的。

SMTP只能在因特网上传递文本信息。为了适应这个要求，邮件客户机必须在发送邮件到邮件服务器以前，将二进制数据（如附件文件）转换为ASCII文本信息。当然接收者的邮件客户机必须能够将文本信息转回原来的二进制数据。

当前常用两种技术将二进制数据转换为文本：uuencode格式和MIME（Multipurpose Internet Mail Extensions，多用途因特网邮件扩展）格式。很多邮件系统允许使用这两种技术，虽然仍然存在一些需要特殊格式的地方。

uuencode

数年前因特网开始普及，UNIX管理员用一种程序将二进制数据转换为ASCII文本并且把其藏在邮件中通过调制解调器发送出去。他们用来将二进制数据转换为ASCII码的程序被称为uuencode。uu表示支持UNIX-to-UNIX，作为UUCP（UNIX-to-UNIX Copy Protocol，UNIX系统间数据复制协议）的一部分用于在UNIX主机之间通过调制解调器发送邮件。

uuencode使用3-to-4编码方式，将二进制数据的3个字节转换为ASCII码的4个字节。这种方式使转换后的文件变得很大，但是可以确保该编码文件安全地转回原始的二进制数据。

由于uuencode非常普及，很多邮件系统仍然支持这种方式对二进制数据编码。而一种新的因特网二进制编码标准已经开发出来了，这就是MIME（多用途因特网邮件扩展协议）。

MIME

MIME格式由RFC 2045和2046定义。MIME比uuencode更为通用，因为它在转换文件中包含了二进制文件的附加信息。解码器可以根据它们自动监测和解码不同类型的二进制文件。

MIME标准也可以把MIME编码方法直接结合到一封标准RFC 2822邮件中。5种新的报头可以用来识别嵌入在邮件中的二进制数据类型。这些项加在RFC 2822报头之后和信体之前。如图13.7所示，所有MIME附件都附加在报头之后。

表13.2 MIME消息报头字段

字段	说明
MIME-Version	指定编码使用的MIME版本
Content-Transfer-Encoding	指定用来将二进制数据转换为ASCII的编码方式
Content-ID	为邮件指定惟一的标识符
Content-Description	鉴别邮件的很短的说明
Content-Type	指定编码文件包含的内容类型

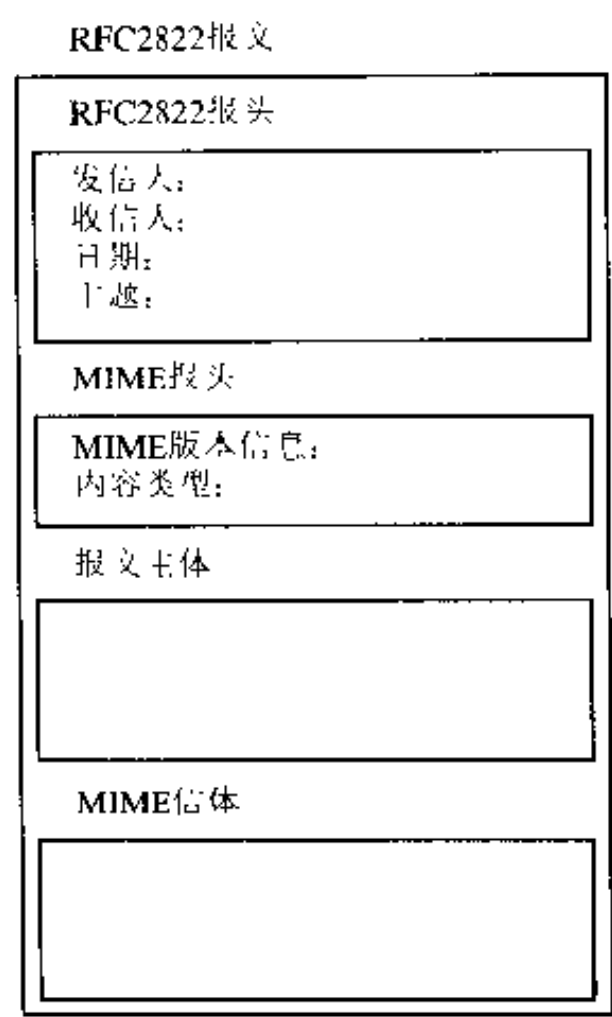


图13.7 MIME邮件格式

MIME版本信息

MIME-Version项标识发送者用来给邮件编码的MIME编码版本:

```
MIME-Version: 1.0
```

一些软件包在版本号后面添加了发行版本信息:

```
MIME-Version: 1.0 (software test 2.3a)
```

接收邮件服务器的MIME解码软件会将这些附加信息忽略掉。

Content-Transfer-Encoding字段

Content-Transfer-Encoding字段标识邮件中的二进制数据是如何编码的。在表13.3中列出当前MIME定义的7种方法。注意其中前二种方法并非对数据进行编码。7位编码方式假定编码后的数据是7位ASCII码而不是二进制数据。如果没有Content-Transfer-Encoding字段出现的话就以此作为默认使用。

Base64是二进制编码最常用的方法。这种方法通过将6位二进制数据映射为8位ASCII文本字节来对二进制数据进行编码。这样在编码文件中“浪费的空间”就比uuencode方法少多了，可以生成一个很小的编码文件。

Content-ID字段

Content-ID字段以唯一的验证码来标识MIME项。一个MIME内容项可以通过使用这个唯一的字段值涉及到另一个MIME信息。

表13.3 MIME编码方法

方法	说明
7-bit	标准7位ASCII文本
8-bit	标准8位ASCII文本
binary	原始二进制数据
quoted-printable	将二进制数据编码为U.S.-ASCII的打印体
base64	将二进制数据编码为8位打印体
ietf-token	RFC 2045定义的扩展标记编码
x-token	令牌跟随的两种字符X或者x（中间没有空格）

Content-Description字段

Content-Description字段是描述性的ASCII文本，用来在电子邮件中将其识别出来。此文本可以是任意长度的ASCII文本。

Content-Type字段

Content-Type字段表示所有的操作。这个字段标识了附在MIME邮件中的数据。用类型（type）和子类型（subtype）两个不同的值标识数据。以下是这个字段的格式：

```
Content-Type: type/subtype
```

以下是7种基本类型的描述：

text 数据采用ASCII格式，子类型可以是以下三者之一：

plain 无格式的ASCII文本

html 格式化的文本

enriched 富文本格式（RTF）的文本

也可以用charset参数以指定字符集对数据编码：

```
Content-Type: text/plain; charset=us-ascii
```

这个命令行标识MIME段为plain文本，通常用于U.S.ASCII编码系统中。

message 在一个简单邮件中包含多个RFC 2822格式的邮件。它有三种子类型：

rfc822 指定普通的RFC 2822格式嵌入邮件

partial 将一个长邮件拆分成独立段落

external-body 指定一个不包含在邮件中的对象的指针

image 定义内嵌的二进制图形图像数据。当前有两种子类型：JPEG格式和GIF格式。

video 定义内嵌的二进制视频数据。只有一种子类型，定义为MPEG格式。

audio 定义内嵌的二进制音频数据。对应的子类型就是基本格式，即单通道综合业务数字网络（ISDN）mu-law编码，采样速率8kHz。

application 定义内嵌的二进制应用程序数据，例如电子表格、字处理文件以及其他应用程序。有两种子类型对应：用于Postscript格式打印文档的postscript格式和octet-stream格式，后者定义了包含任意二进制数据的邮件。octet-stream子类型表示的大多数都是特定应用程序数据，比如Microsoft Word文档和Microsoft Excel电子表格。

multipart 这是一种特殊类型，它标识的邮件将多种类型的数据结合在一个邮件中。这种格式在电子邮件包中很常用，可以用不同方式表示一封邮件，例如明文ASCII文本和HTML，还可以在邮件中包含多种附件。有4种子类型对应：

Mixed 各部分独立，必须按接收时的先后次序送交收信人。

Parallel 各部分独立，可以按任意次序送交收信人。

Alternative 各部分含义相同但表现形式不同，只有一个部分可以送交收信人。

Digest 与mixed子类型的处理方法相同，但是规定邮件必须始终保持RFC 822格式。

说明：除了上述7种基本格式以外还有很多Content-Types。某些电子邮件包会定义它们自己的类型。只要邮件服务器和邮件客户端能识别的ContentType就可以使用。不过要小心，当使用非标准的ContentType时，其他邮件客户端可能无法识别它们。

MailAttachment类

为简化电子邮件附件，.NET库包含MailAttachment类。这种类指定了包含在邮件中的任意附件的内容和格式。

有两种构造器可以生成一个新的邮件附件：

```
MailAttachment(string filename);  
MailAttachment(string filename, MailEncoding encodetype);
```

第一种构造器格式使用指定文件路径下的文件并且用UUEncode格式编码生成新的邮件附件。

第二种构造器格式允许使用选择性的编码方式，例如用于MIME邮件的Base64类型。

System.Web.Mail命名空间包含以下MailEncoding，是两种不同的编码类型：

MailEncoding.Base64 用于base64 MIME编码

MailEncoding.UUEncode 用于uuencode编码

可以用Encoding和Filename属性查询MailAttachment对象的编码和文件名属性：

```
MailAttachment newfile = new  
    MailAttachment("c:\\testfile.bmp", MailEncoding.Base64);  
string filename = newfile.Filename;  
if (newfile.Encoding == MailEncoding.Base64)  
    Console.WriteLine("{0} is a MIME encoded file", filename);
```

创建MailAttachment对象后，可以通过Attachments属性将其纳入MailMessage对象之中：

```
MailMessage newmessage = new MailMessage();  
MailAttachment newfile = new MailAttachment("c:\\testfile.bmp");  
newmessage.Attachments.Add(newfile);
```



```
newmessage.From = "frank@myisp.net";
newmessage.To = "melanie@myisp.net;nicholas@myisp.net";
newmessage.Body = "Here's my picture!!";
SmtpMail.Send(newmessage);
```

程序清单13.3中的MailAttachTest.cs程序演示了如何向邮件中添加文件附件。

程序清单13.3 MailAttachTest.cs程序

```
using System;
using System.Web.Mail;

class MailAttachTest
{
    public static void Main()
    {
        MailAttachment myattach =
            new MailAttachment("c:\\temp\\MailAttachTest.exe",
                MailEncoding.Base64);
        MailMessage newmessage = new MailMessage();
        newmessage.From = "barbara@shadrach.ispnet1.net";
        newmessage.To = "rich@shadrach.ispnet1.net";
        newmessage.Subject = "A test mail attachment message";
        newmessage.Priority = MailPriority.High;
        newmessage.Headers.Add("Comments",
            "This message attempts to send a binary attachment");
        newmessage.Attachments.Add(myattach);
        newmessage.Body = "Here's a test file for you to try";

        try
        {
            SmtpMail.SmtpServer = "192.168.1.100";
            SmtpMail.Send(newmessage);
        } catch (System.Web.HttpException)
        {
            Console.WriteLine("This device cannot send Internet messages");
        }
    }
}
```

MailAttachTest程序用Base64编码方式生成一个MailAttachment对象。下面生成一个普通的MailMessage对象，并且把MailAttachment对象作为附件添加进去：

```
newmessage.Attachments.Add(myattach);
```

这个例子也要使用远程SMTP邮件转发服务器来向收信人转发邮件。

POP3客户机

前面介绍了如何使用C#发送SMTP邮件，接下来讲述如何从邮箱中接收这些邮件。

在“电子邮件基础”一节提到，MUA可以允许远程用户阅读存储在其邮箱中的邮件。最常用于从远程邮件服务器收取信件的网络协议是POP3。

本节描述了POP3如何工作，并且给出一个客户读取存储在远程邮件服务器上的邮件的程序样例。

POP3协议

POP3是一种常用的基本协议。一旦客户机与服务器建立一个连接，就发出简单文本命令，并且服务器以恰当的信息来响应。必须知道使用的命令是什么、以什么样的顺序发出，才能正确地服务器上收取信件。

POP3服务监听TCP的110端口是否有人尝试连接。当一个新的连接建立起来，POP3服务器会发送一个欢迎消息并且等待客户机发送命令。

权限检查命令

第一步（建立连接以后），使用以下方法之一登录POP3服务器：

- 发送明文的用户ID和密码
- 发送加密的权限检查令牌

用加密的权限检查令牌连接服务器是一种比较安全的方式，不过这里并不打算介绍加密方法，读者可以自行了解有关的MD-5加密算法。本例中使用明文传输用户ID和密码，用USER和PASS命令完成：

USER用户名

PASS密码

分别输入每个命令并且从服务器收到响应。所有的POP3服务器响应前面都有+OK或者-ERR，前者表示肯定确认，后者表示否定确认。

远程登录到110端口并且输入正确的命令可以手工测试POP3服务器的登录序列：

```
C:\>telnet 192.168.1.100 110
+OK QPOP (version 2.53) at shadrach.ispnet1.net starting.
<6352.1030035698@shadrach.ispnet1.net>
USER rich
+OK Password required for rich.
PASS myaccount1
+OK rich has 2 messages (889 octets).
QUIT
+OK Pop server at shadrach.ispnet1.net signing off.
C:\>
```


在这个例子中，每个命令都从POP3服务器收到了肯定的答复。当PASS命令被确认时，响应消息会包含邮箱中邮件的数量。可惜这个消息是由POP3服务器软件指定的，不能通过它有效检查邮件的数量。

获取信息命令

STAT和LIST命令可以得到关于邮箱的信息。STAT命令响应如下所示：

```
+OK n mmm
```

这个响应包含邮箱中邮件的数量（n）和全部邮件大小（字节）。

LIST命令收到一份个人邮件列表，列出每封邮件的大小，如下所示：

```
STAT
+OK 2 889
LIST
+OK 2 messages (889 octets)
1 464
2 425
.
```

注意LIST响应的末尾是一个独占一行的句号，表示该列表已经到达末尾。

检索邮件

用于检索邮箱中消息文本的比较有用的命令有两个：TOP和RETR。

虽然TOP命令是可选命令，大多数电子邮件系统都支持它。它能取得邮件的报头并且从信体中挑选出文本行的数目。TOP命令格式如下，这里n表示信件数量（如同LIST命令的结果一样），mm表示信体的文本行数量：

```
TOP n mm
```

如果只需要获取邮件报头，可以指定0作为第2个参数：

```
TOP 1 0
+OK 464 octets
Return-Path: rich
Received: (from root@localhost)
    by shadrach.ispnet1.net (8.9.3/8.9.3) id KAA06215
    for rich; Thu, 22 Aug 2002 10:32:34 -0500 (EST)
    (envelope-from rich)
Date: Thu, 22 Aug 2002 10:32:34 -0500 (EST)
From: Rich Blum <rich>
Message-Id: <200208221532.KAA06215@shadrach.ispnet1.net>
To: rich
Subject: Another test message
X-UIDL: 5291f9aeb6f2e88b4f7acdc5511c4ae9
Status: RC
.
```

首先, POP3服务器返回一个肯定确认消息(+OK)。很多服务器也会返回邮件的字节数。接下来返回邮件的报头。然后在响应信息的末尾是一个句点。

要想获取整个邮件, 需要使用RETR命令。RETR命令用到一个参数, 获取邮件的数目:

```
RETR 1
+OK 464 octets
Return-Path: rich
Received: (from root@localhost)
    by shadrach.ispnet1.net (8.9.3/8.9.3) id KAA06215
    for rich; Thu, 22 Aug 2002 10:32:34 -0500 (EST)
    (envelope-from rich)
Date: Thu, 22 Aug 2002 10:32:34 -0500 (EST)
From: Rich Blum <rich>
Message-Id: <200208221532.KAA06215@shadrach.ispnet1.net>
To: rich
Subject: Another test message
X-UIDL: 5291f9ae56f2e88b4f7acdc5511c4ae9
Status: RC

This is a test message to test the popcheck program with.
.
```

与TOP命令相似, RETR命令返回的是邮件的全部字节数目和整个邮件的文本。

当POP3会话结束, 要向服务器发送QUIT命令来正确终止会话, 并且关闭邮箱。

警告: 收取带有二进制附件的邮件时要小心。这种邮件会复原为原始的文本格式, 包括二进制附件的文本编码。

编写POP3客户机程序

现在读者应该已经了解了有关POP3的信息, 可以编写一个简单的网络客户机程序来检查远程POP3邮箱并且获取其中邮件的一些信息。程序清单13.4是PopCheck.cs程序, 示范一些基本原理。

程序清单13.4 PopCheck.cs程序

```
using System;
using System.Drawing;
using System.IO;
using System.Net;
using System.Net.Sockets;
using System.Text;
using System.Threading;
using System.Windows.Forms;

class PopCheck Form
```

```
{
    private TextBox hostname;
    private TextBox username;
    private TextBox password;
    private TextBox status;
    private ListBox messages;

    private TcpClient mailclient;
    private NetworkStream ns;
    private StreamReader sr;
    private StreamWriter sw;

    public PopCheck()
    {
        Text = "popcheck - A POP3 e-mail checker";
        Size = new Size(400, 380);

        Label label1 = new Label();
        label1.Parent = this;
        label1.Text = "Hostname:";
        label1.AutoSize = true;
        label1.Location = new Point(10, 33);

        hostname = new TextBox();
        hostname.Parent = this;
        hostname.Size = new Size(200, 2 * Font.Height);
        hostname.Location = new Point(75, 30);

        Label label2 = new Label();
        label2.Parent = this;
        label2.Text = "User name:";
        label2.AutoSize = true;
        label2.Location = new Point(10, 53);

        username = new TextBox();
        username.Parent = this;
        username.Size = new Size(200, 2 * Font.Height);
        username.Location = new Point(75, 50);

        Label label3 = new Label();
        label3.Parent = this;
        label3.Text = "Password:";
        label3.AutoSize = true;
        label3.Location = new Point(10, 73);

        password = new TextBox();
        password.Parent = this;
        password.PasswordChar = '*';
    }
}
```

```

password.Size = new Size(200, 2 * Font.Height);
password.Location = new Point(75, 70);

Label label4 = new Label();
label4.Parent = this;
label4.Text = "Status:";
label4.AutoSize = true;
label4.Location = new Point(10, 325);

status = new TextBox();
status.Parent = this;
status.Text = "Not connected";
status.Size = new Size(200, 2 * Font.Height);
status.Location = new Point(50, 322);

messages = new ListBox();
messages.Parent = this;
messages.Location = new Point(10, 108);
messages.Size = new Size(360, 16 * Font.Height);
messages.DoubleClick += new EventHandler(getmessagesDoubleClick);

Button login = new Button();
login.Parent = this;
login.Text = "Login";
login.Location = new Point(295, 32);
login.Size = new Size(5 * Font.Height, 2 * Font.Height);
login.Click += new EventHandler(ButtonloginOnClick);

Button close = new Button();
close.Parent = this;
close.Text = "Close";
close.Location = new Point(295, 62);
close.Size = new Size(5 * Font.Height, 2 * Font.Height);
close.Click += new EventHandler(ButtoncloseOnClick);
}

void ButtonloginOnClick(object obj, EventArgs ea)
{
    status.Text = "Checking for messages...";
    Thread startlogin = new Thread(new ThreadStart(loginandretr));
    startlogin.IsBackground = true;
    startlogin.Start();
}

void ButtoncloseOnClick(object obj, EventArgs ea)
{
    if (ns != null)
    {

```

```
        sw.Close();
        sr.Close();
        ns.Close();
        mailclient.Close();
    }
    Close();
}

void loginandretr()
{
    string response;
    string from = null;
    string subject = null;
    int totmessages;

    try
    {
        mailclient = new TcpClient(hostname.Text, 110);
    } catch (SocketException)
    {
        status.Text = "Unable to connect to server";
        return;
    }

    ns = mailclient.GetStream();
    sr = new StreamReader(ns);
    sw = new StreamWriter(ns);

    response = sr.ReadLine(); //Get opening POP3 banner

    sw.WriteLine("User " + username.Text); //Send username
    sw.Flush();

    response = sr.ReadLine();
    if (response.Substring(0,3) == "-ER")
    {
        status.Text = "Unable to log into server";
        return;
    }

    sw.WriteLine("Pass " + password.Text); //Send password
    sw.Flush();

    try
    {
        response = sr.ReadLine();
    } catch (IOException)
    {
```

```

        status.Text = "Unable to log into server";
        return;
    }
    if (response.Substring(0,4) == "-ERR")
    {
        status.Text = "Unable to log into server";
        return;
    }

    sw.WriteLine("stat"); //Send stat command to get number of messages
    sw.Flush();

    response = sr.ReadLine();
    string[] nummess = response.Split(' ');
    totmessages = Convert.ToInt16(nummess[1]);
    if (totmessages > 0)
    {
        status.Text = "you have " + totmessages + " messages";
    } else
    {
        status.Text = "You have no messages" ;
    }

    for (int i = 1; i <= totmessages; i++)
    {
        sw.WriteLine("top " + i + " 0"); //read header of each message
        sw.Flush();
        response = sr.ReadLine();

        while (true)
        {
            response = sr.ReadLine();
            if (response == ".")
                break;
            if (response.Length > 4)
            {
                if (response.Substring(0, 5) == "From:")
                    from = response;
                if (response.Substring(0, 8) == "Subject:")
                    subject = response;
            }
        }
        messages.Items.Add(i + " " + from + " " + subject);
    }
}

```



```

void getmessagesDoubleClick(object obj, EventArgs ea)
{
    string text = (string)messages.SelectedItem;
    string[] textarray = text.Split(' ');
    ShowMessage sm = new ShowMessage(ns, textarray[0]);
    sm.ShowDialog();
}

public static void Main()
{
    Application.Run(new PopCheck());
}
}

class ShowMessage : Form
{
    public ShowMessage(NetworkStream ns, string messnumber)
    {
        StreamReader sr = new StreamReader(ns);
        StreamWriter sw = new StreamWriter(ns);
        string response;

        Text = "Message " + messnumber;
        Size = new Size(400, 380);
        ShowInTaskbar = false;

        TextBox display = new TextBox();
        display.Parent = this;
        display.Multiline = true;
        display.Dock = DockStyle.Fill;
        display.ScrollBars = ScrollBars.Both;

        sw.WriteLine("retr " + messnumber); //Retrieve entire message
        sw.Flush();
        response = sr.ReadLine();

        while (true)
        {
            response = sr.ReadLine();
            if (response == ".")
                break;
            display.Text += response + "\r\n";
        }
    }
}
}

```

PopCheck程序采用了很多前面章节中介绍过的网络编程技术。图13.8展示了PopCheck程序运行的窗口。

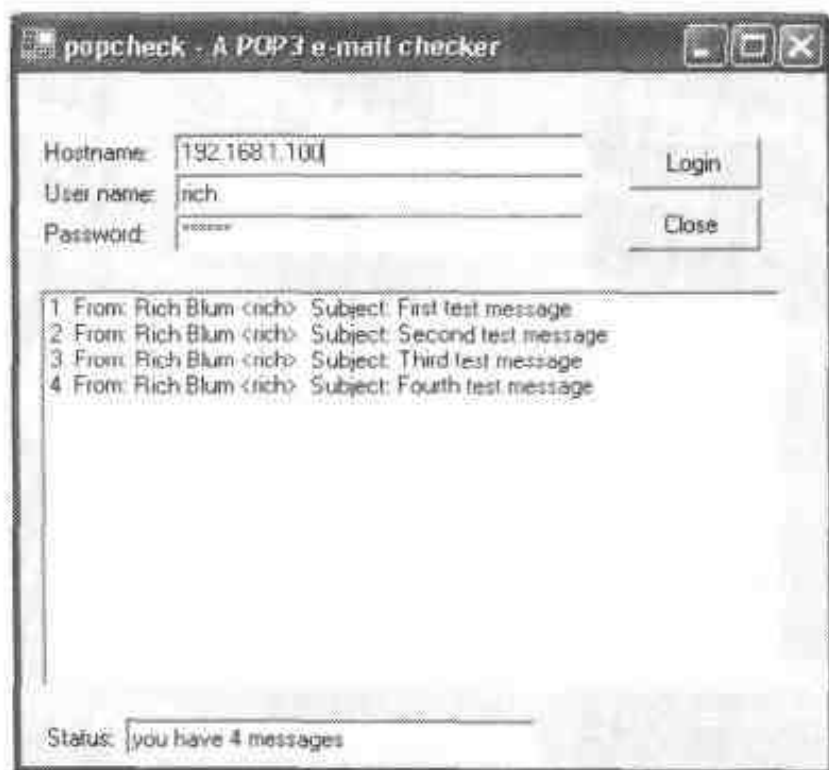


图13.8 PopCheck程序界面

当用户分别在文本框中输入主机名、用户名和密码以后，单击登录按钮。系统就会开始用loginandretr()方法，以指定用户名和密码登录某个远程POP3服务器，这一线程开始了。在用户登录以后，STAT命令会检查邮箱中邮件的数量，TOP命令会获取每个邮件的报头信息。

邮件数量与From和Subject报头字段信息一同列在清单框中。用户可以单击列在清单框中的某个邮件来显示整个邮件。

PopCheck.cs与其他程序不同，它在同一个源文件中包括一个第二类的定义，这个第二类定义了一个对话框窗口用来显示获得的邮件文本。主程序将连接的NetworkStream对象与所需要的邮件数目一同传递给对话框类。该信息用于向POP3服务器发送RETR命令并且获取邮件文本。由于已经连接的NetworkStream对象被传递过去了，所以对话框不需要与服务器生成一个新的连接。

小结

本章讲述如何编写网络程序来操作网络邮件。多数网络邮件系统将邮件功能分成三个部分：报文发送代理（MTA）使用SMTP向本地和远程系统发送邮件；报文传输代理（MDA）向本地用户发送邮件，可以实现包括邮件过滤在内的很多功能；报文用户代理（MUA）允许远程用户读取他们邮箱中的邮件。

Windows 2000和XP平台包含有协作数据对象（CDO）库，使用这个库，可以实现用SMTP发送邮件以及其他网络邮件和新闻功能。.NET mail库利用CDO支持System.Web.Mail邮件功能。Windows 2000和XP平台允许用户安装因特网信息服务包（IIS），其中包含一个因特网SMTP邮件服务器，可以从CDO库转发和接收邮件。

SmtpMail类用于发送电子邮件，用的是内部SMTP服务器或者外部的邮件转发服务器。邮件可以用MailMessage类生成，包含独特的文本格式和特定的邮件报头。MailAttachment类

用于向**SmtpMail**类发送的邮件中添加二进制附件。这样网络程序就可以很轻易地生成全面的邮件程序。

本章还涉及到了邮局协议（**POP3**）以及如何构造一个简单的**POP3**邮件客户机程序来从远程邮件服务器获取邮件。

除了电子邮件，因特网上最具影响力的应用恐怕非万维网（**WWW**）莫属了，**.NET**类库自然也包含很多用于**Web**编程的类。下一章就将讨论这些类的用法。

第14章 HTTP

- 用于数据上传和下载的WebClient类
- 如何使用证书
- 如何使用高级类HttpWebRequest和HttpWebResponse
- 如何实现网络服务

Web编程技术已经成为当前的热门话题，因此，微软公司在.NET网络类库中集成了很多Web特性，使用它们，程序员将编制出基于网件的网络应用程序。

本章将介绍.NET中的WebClient类，这个类是实现C#应用程序HTTP通信功能的最佳选择。本章还将介绍WebRequest和WebResponse这两个类的功能，使用它们可以创建功能完善的网络应用程序。接下来，还会通过一个例子，演示创建网络应用程序的步骤。最后，将会谈谈有关网络服务更高层的功能，即.NET如何实现C#应用程序与Web服务器之间的直接通信。

WebClient类

要实现C#应用程序同Web服务器之间的通信，最简便的办法就是使用WebClient类，该类存放在System.Net命名空间内。这个类中实现了程序与服务器交换信息的各种方法，可以非常方便地为应用程序增加网络功能。

下载Web数据

在下载数据方面，WebClient类提供了三个方法：

DownloadData() 从指定URL下载数据，保存为字节数组

DownloadFile() 从指定URL下载数据，保存为本地文件

OpenRead() 启动一个只读类型的流，通过流从指定URL下载数据

以上三种方法都必须指定URL，明确网络资源的位置。URL的书写格式必须符合标准HTTP URI规范的要求。

DownloadData()方法

DownloadData()方法可以处理针对HTTP请求的HTML形式的响应，然后将其转换为字节数组的形式保存下来。有了字节数组，程序就可以运行HTML页面，然后在网页上搜索感兴趣的内容。程序清单14.1是一个名为DownloadDataTest.cs的程序，该程序的功能很简单，就是到指定网址下载一个页面，然后显示网页内容。

程序清单14.1 DownloadDataTest.cs程序

```
using System;
using System.Net;
using System.Text;

class DownloadDataTest
{
    public static void Main (string[] argv)
    {
        WebClient wc = new WebClient();
        byte[] response = wc.DownloadData(argv[0]);

        Console.WriteLine(Encoding.ASCII.GetString(response));
    }
}
```

在命令提示符后输入网址，程序就可以显示出页面的HTML源代码：

```
C:\>DownloadDataTest http://localhost/test.htm
<html>
<title>This is a test web page</title>
<body>
<h1>This is a test web page</h1>
</body>
</html>

C:\>
```

可见，网页的HTML源代码以字符形式完整地显示在控制台上。

说明：如果计算机上安装了Internet Explorer，那么WebClient类会自动地使用其代理服务器设置；如果没有安装Internet Explorer，WebClient将无法使用代理服务器。关于如何在网络应用程序内使用代理服务器的内容，请阅读本章“高级Web类”中对WebRequest类的介绍。

DownloadFile()方法

如果使用DownloadFile()方法，就可以将下载得到的网络数据保存为一个本地文件，而不是存储成为字节数组。程序清单14.2是名为DownloadFileTest.cs的程序，可以实现下载网络数据并保存为本地文件的功能。

程序清单14.2 DownloadFileTest.cs程序

```
using System;
using System.Net;

class DownloadFileTest
{
    public static void Main (string[] argv)
```

```
{
    WebClient wc = new WebClient();
    string filename = "webpage.htm";
    wc.DownloadFile(argv[0], filename);
    Console.WriteLine("file downloaded");
}
}
```

DownloadFileTest这个程序将下载的网页保存为本地文件，名为webpage.htm:

```
C:\>DownloadFileTest http://localhost/test.htm
file downloaded

C:\>type webpage.htm
<html>
<title>This is a test web page</title>
<body>
<h1>This is a test web page</h1>
</body>
</html>

C:\>
```

OpenRead()方法

OpenRead()方法可以把网页变成一个Stream对象，然后调用StreamReader对象进行处理，从而获取网页数据。程序清单14.3中的OpenReadTest.cs程序完成了这个过程。

程序清单14.3 OpenReadTest.cs程序

```
using System;
using System.IO;
using System.Net;

class OpenReadTest
{
    public static void Main (string[] argv)
    {
        WebClient wc = new WebClient();
        string response;

        Stream strm = wc.OpenRead(argv[0]);

        StreamReader sr = new StreamReader(strm);

        while(sr.Peek() > -1)
        {
            response = sr.ReadLine();
            Console.WriteLine(response);
        }
    }
}
```

```

    }
    sr.Close();
}
}

```

OpenReadTest程序的输出和**DownloadDataTest**程序的输出完全一致，都是网页的HTML源代码。不过，在**OpenRead()**这种方式下，程序员可以使用其他的一些方法，比如**StreamReader.Peek()**，因此，程序员对整个下载过程和得到的数据具有更加灵活的操作能力。

查看HTTP报头

WebClient类有一个名为**ResponseHeaders**的属性，通过这个属性，可以查看HTTP通信的报头内容。报头数据虽然并不出现在网页上，但是对于分析HTTP通信状况是非常有帮助的。程序清单14.4所示的**RcsponseHeadersTest.cs**程序可以说明报头数据该如何使用。

程序清单14.4 ResponseHeadersTest.cs程序

```

using System;
using System.Collections.Specialized;
using System.Net;

class ResponseHeadersTest
{
    public static void Main (string[] argv)
    {
        WebClient wc = new WebClient();

        byte[] response = wc.DownloadData(argv[0]);
        WebHeaderCollection whc = wc.ResponseHeaders;
        Console.WriteLine("header count = {0}", whc.Count);
        for (int i = 0; i < whc.Count; i++)
        {
            Console.WriteLine(whc.GetKey(i) + " = " + whc.Get(i));
        }
    }
}

```

ResponseHeaders属性将会返回一个**WebHeaderCollection**对象。和.NET中类似情形相同，可以调用**GetKey()**方法访问该对象的项目名称（报头内容），或者调用**Get()**方法访问该对象的值。下面给出一个**ResponseHcadertest**程序的运行实例，其中显示了HTTP响应所有的报头数据和值。

```

C:\>ResponseHeadersTest http://www.microsoft.com
header count = 10
Date = Tue, 27 Aug 2002 04:06:34 GMT

```

```
Server = Microsoft-IIS/6.0
P3P = CP= 'ALL IND DSP COR ADM CONo CUR CUSo IVAo IVDo PSA PSD TAI TELo OUR SAMo
CNT COM INT NAV ONL PHY PRE PUR UNI'
Content-length = 31257
Content-Type = text/html
Expires = Tue, 27 Aug 2002 18:06:34 GMT
Cache-Control = private
Age = 0
X-Cache = MISS from proxy.ispnet.net
Proxy-Connection = keep-alive

C:\ >
```

更新Web数据

在更新Web服务器内容方面，WebClient类提供了4种方法：

OpenWrite() 以流的形式向Web服务器发送数据

UploadData() 以字节数组形式向Web服务器发送数据

UploadFile() 向Web服务器发送一个本地文件

UploadValues() 将数据名称和值的NameValueCollection发送给Web服务器

与下载数据时类似，在调用上传数据的各种方法时都必须指明有效的URI，而且必须以标准格式指定。

警告：向Web服务器上传数据必须具有足够的操作权限。

OpenWrite()方法

用OpenWrite()方法向Web服务器上传数据时会生成一个Stream对象，该方法有如下两种调用方式：

```
OpenWrite(string URI);
OpenWrite(string URI, string method);
```

第一种方式在以流的方式向Web服务器传送数据时，固定使用HTTP POST方法。第二种方式则可以由程序员选择各种HTTP方法实现数据传送。

创建Stream对象之后，可以创建一个StreamWriter对象，简化以HTML格式向Web服务器传送文本的工作。程序清单14.5是一个名为OpenWriteTest.cs的程序，其中就应用了这种技术。OpenWriteTest程序在传送数据时使用了HTTP POST方法，URI则在命令行中指定。

程序清单14.5 OpenWriteTest.cs程序

```
using System;
using System.IO;
using System.Net;

class OpenWriteTest
```



```
{
    public static void Main (string[] argv)
    {
        WebClient wc = new WebClient();
        string data = "Data up upload to server";

        Stream strm = wc.OpenWrite(argv[0]);
        StreamWriter sw = new StreamWriter(strm);

        sw.WriteLine(data);
        sw.Close();
        strm.Close();
    }
}
```

UploadData()方法

UploadData()方法能够以字节数组形式向Web服务器发送数据。与OpenWrite()方法类似，UploadData()方法也有两种调用方式：

```
UploadData(string URI, byte[] array);
UploadData(string URI, string method, byte[] array);
```

默认情况下，UploadData()方法使用HTTP POST方法向Web服务器传送数据。第二种调用方式则允许使用其他HTTP方法传送数据。

程序清单14.6是一个名为UploadDataTest.cs的程序，该程序调用UploadData()方法上传数据。

程序清单14.6 UploadDataTest.cs程序

```
using System;
using System.Net;
using System.Text;

class UploadDataTest
{
    public static void Main (string[] argv)
    {
        WebClient wc = new WebClient();

        string data = "This is the data to post";
        byte[] dataarray = Encoding.ASCII.GetBytes(data);

        wc.UploadData(argv[0], dataarray);
    }
}
```

UploadFile()方法

UploadFile()方法能够以发送本地文件的形式上传数据，用起来十分简单。UploadFile()

方法有如下两种调用方式:

```
UploadFile(string URI, string filename);  
UploadFile(string URI, string method, string filename);
```

在使用UploadFile()方法时,必须指明URI和文件名。使用第二种方式时,还可以指定用什么HTTP方法完成上传(默认方法为POST)。

UploadValues()方法

与其他上传方法不同,UploadValues()方法并不向Web服务器上传数据,而是上传数据的名称/取值,这与查询请求中用URI表示数据很相似。每个名称/取值之间用符号“&”分隔。

```
?lastname=Blum&firstname=Rich
```

UploadValues()方法定义了一个名为NameValueCollection的对象,生成数据名称和取值。

```
NameValueCollection nvc = new NameValueCollection();  
nvc.Add("lastname", "Blum");  
nvc.Add("firstname", "Rich");  
byte[] response = wc.UploadValues(uri, "POST", nvc);
```

调用UploadValues()方法的第二个参数用于指定上传数据使用什么HTTP方法。绝大多数网页都是用GET或POST方法来上传数据的。

程序清单14.7列出一个名为UploadValuesTest.cs的程序,可以向一个示例性网页发送数据的名称/取值。这个程序在传送数据时会创建一个NameValueCollection对象,URI则在调用时指定。

说明: 本例中的目标网页是一个需要输入数据的表格,因此,要运行本例,必须要指定一个能够接收HTML表格数据的网页。

程序清单14.7 UploadValuesTest.cs程序

```
using System;  
using System.Collections.Specialized;  
using System.Net;  
using System.Text;  
  
class UploadValuesTest  
{  
    public static void Main (string[] argv)  
    {  
        WebClient wc = new WebClient();  
        string uri = "http://localhost/testform.aspx"; //  
        NameValueCollection nvc = new NameValueCollection();
```



```
nvc.Add("lastname", 'Blum ');
nvc.Add("firstname", "Rich");

byte[] response = wc.UploadValues(uri, nvc);

Console.WriteLine(Encoding.ASCII.GetString(response));
}
}
```

使用Credential属性

WebClient类的另一个特点是具有Credential属性，当网页需要检查权限时，利用这个属性可以向Web服务器传送用户名和密码。在编程时，必须为Credential属性设定认证级别，即设定为NetworkCredential或CredentialCache，或者二者都是。

NetworkCredential类

NetworkCredential类（见System.Net命名空间）用来为客户机在Web服务器上设定一组用户名和密码（如果是基于Windows的Web服务器，还可以设定一个指定的域名）。创建一个NetworkCredential对象共有三种方式：

```
NetworkCredential()
NetworkCredential(string username, string password)
NetworkCredential(string username, string password, string domain)
```

用第一种方式创建的NetworkCredential对象具有UserName、Password和Domain属性；其他两种方式在创建对象时就同时设定了这三种属性。

程序清单14.8给出了一个例程，名为CredTest.cs，这个程序中使用了一个NetworkCredential对象，为一个网站设定了一组用户名和密码。该程序首先创建一个NetworkCredential对象来设定用户名和密码，在Web服务器上为客户机建立访问权限。当客户机访问服务器时，如果是合法用户，那么就可以下载服务器上的内容；如果访问不合法，那么程序将生成一个WebException异常。

程序清单14.8 CredTest.cs程序

```
using System;
using System.Net;
using System.Text;

class CredTest
{
    public static void Main()
    {
        WebClient wc = new WebClient();

        NetworkCredential nc = new NetworkCredential("alex", "mypassword");
```

```
        wc.Credentials = nc;

        byte[] response = wc.DownloadData("http://localhost/testlogin.");

        Console.WriteLine(Encoding.ASCII.GetString(response));
    }
}
```

CredentialCache类

在网络编程中，有时程序必须同时与多个需要进行权限验证的网站进行连接，如果要想所有的连接都同时保持稳定可用，对编程而言无疑是件非常困难的事情。针对这种情况，.NET网络类库给出了一种解决方案——System.Net命名空间中的CredentialCache类。这个类可以建立一个缓冲区，存放那些经常调用的NetworkCredential对象，并为每个对象配套创建URI对象和一个授权验证方法。当需要用NetworkCredential对象访问网站时，程序员可以直接操作整个CredentialCache对象，这时，NetworkCredential对象、URI和授权验证方法都会自动启用。

使用CredentialCache类的Add()方法可以创建CredentialCache对象，但并不会创建新的NetworkCredential对象。

```
Add(Uri website, string authType, NetworkCredential cred)
```

URI对象由Uri类创建，用来声明网站的地址。授权验证的类型用一个字符串来表示，代表了所用的HTTP授权验证方法。目前，共有两种类型的授权验证方法：

Basic 用纯文本格式发送用户名和密码

Digest 用MD-5加密方法发送用户名和密码

程序清单14.9中的例程名为CredCacheTest.cs，这个程序会对三个各自独立的URI网站生成一个统一的CredentialCache对象。CredentialCache对象cc囊括了与三个网站有关的全部NetworkCredential对象，这样，程序员的操作目标就不再是一个个NetworkCredential对象，而是cc这个单一的CredentialCache对象。之后，由WebClient对象自动定位NetworkCredential对象。

程序清单14.9 CredCacheTest.cs程序

```
using System;
using System.Net;
using System.Text;

class CredCacheTest
{
    public static void Main()
    {
        WebClient wc = new WebClient();
        string website1 = "http://remotel.ispnet.net";
        string website2 = "http://remote2.ispnet.net";
```

```

        string website3 = 'http://remote3.ispnet.net/login';

        NetworkCredential nc1 = new NetworkCredential('mike', "guitars");
        NetworkCredential nc2 = new NetworkCredential
            ("evonne", "singing", "home");
        NetworkCredential nc3 = new NetworkCredential('alex', "drums");

        CredentialCache cc = new CredentialCache();
        cc.Add(new Uri(website1), "Basic", nc1);
        cc.Add(new Uri(website2), "Basic", nc2);
        cc.Add(new Uri(website3), "Digest", nc3);

        wc.Credentials = cc;

        wc.DownloadFile(website1, "website1.htm");
        wc.DownloadFile(website2, "website2.htm");
        wc.DownloadFile(website3, "website3.htm");
    }
}

```

高级Web类

除了WebClient类，.NET类库还提供了WebRequest类和WebResponse类，这两个类都包含在System.Net命名空间中，程序员们可以用它们实现很多高级的网络功能。WebRequest类和WebResponse类都是抽象基类，也就是说，它们在程序中不能直接作为对象使用，必须被继承。实际使用中，要根据URI参数中的URI前缀选用它们合适的子类。

对于http和https这类URI，HttpWebRequest和HttpWebResponse类可以用于处理程序同Web服务器之间的HTTP通信。与WebClient类相比，这两个类可实现更多与HTTP相关的高级功能。如果是文件型URI（如访问本地保存的HTML网页），应该使用FileWebRequest类和FileWebResponse类，这两个类的功能与上面两个针对HTTP的类相当。

HttpWebRequest类

HttpWebRequest类实现了很多有关通过HTTP访问Web服务器上网页的高级功能。要创建HttpWebRequest对象，必须通过WebRequest.Create()这个全局方法来定义。如下例：

```

HttpWebRequest hwr =
    (HttpWebRequest)WebRequest.Create("http://remotehost/webpage.htm");

```

上面代码中，WebRequest类方法返回一个HttpWebRequest对象。创建了这个对象后，就可以通过HttpWebRequest类的属性，设置很多HTTP报头字段的内容。

使用Web代理服务器

面对来自因特网黑客的威胁，越来越多的商业组织会在其内部网络和因特网之间架设防火墙，防火墙可以阻挡所有的访问行为，包括正常的网络访问。

一般说来，要让合法的网络访问请求通过防火墙，需要运行Web代理服务器。内部网中所有对Web的访问请求首先会发送给代理服务器，再由代理服务器转发给因特网上相应的网址。安装在内部网客户机上的Web程序必须先将HTTP消息发送到Web代理服务器，而不是直接发往目的地址。

可以用HttpRequest类定义一个Web代理服务器，供所有HTTP通信程序使用。属性Proxy定义了Web代理服务器的URI，要设置属性Proxy，必须使用WebProxy类。

WebProxy类有很多关于设置Web代理服务器参数的构造器，比如可以给用户登录增加验证功能，还可以选择是否允许外部网络通过代理服务器访问内部网站。定义一个WebProxy对象的最简单的办法就是将Web代理服务器的URI作为字符串：

```
HttpRequest hwr = (HttpRequest)WebRequest.Create(  
    "http://remotel.ispnet2.net");  
WebProxy proxysrv = new WebProxy("http://proxy1.ispnet.net:2000");  
hwr.Proxy = proxysrv;
```

定义了Web代理服务器之后，在发送和接收HTTP报文时，并不会感觉到客户机和Web服务器之间存在Web代理服务器。

发送数据

必须使用GetRequestStream()方法来向Web服务器发送数据，示例如下：

```
HttpRequest hwr = (HttpRequest)WebRequest.Create("http://localhost");  
Stream strm = hwr.GetRequestStream();  
StreamWriter sw = new StreamWriter(strm);  
sw.WriteLine(data);
```

上述方法返回一个Stream对象，这个对象可以直接发送数据给Web服务器。

警告：GetRequestStream()方法返回的Stream对象仅能用于向服务器发送数据，而不能用来接收数据。

检索数据

要检索来自HttpRequest对象的数据有两种方式：

GetResponse() 将Web服务器的返回结果存放在一个HttpWebResponse对象中。

BeginGetResponse() 启动一个异步函数调用，将Web服务器返回的数据存放在一个HttpWebResponse对象中，这种方法用到了异步方法。有关异步方法的介绍详见第8章“异步套接字编程”。

这两种方法都能以HttpWebResponse对象的形式返回服务器的数据。

HttpWebResponse类

HttpWebResponse类继承自WebResponse类，专门用于处理从Web服务器返回的HTTP响应。这个类实现了很多方法，具有很多属性，可以全面处理接收到的网页信息。

检索HTTP报头字段

在HttpWebResponse类中，对于大多数通用的HTTP报头字段，都有独立的属性与其对应。因此，程序员可以通过这些属性方便地访问位于HTTP接收报文报头字段中的信息。HttpWebResponse对象可用的属性如表14.1所示。

表14.1 HttpWebResponse类的属性

属性	Gets
CharacterSet	接收内容所用的字符集
ContentEncoding	接收内容所用的编码方法
ContentLength	接收内容的长度
ContentType	接收内容的类型
Cookies	与接收内容有关的cookie
Headers	接收内容的HTTP报头
LastModified	对接收内容最近一次修改的发生时间和日期
Method	接收内容中所用的HTTP方法
ProtocolVersion	接收内容所用的HTTP协议版本
ResponseUri	对请求做出响应的服务器的URI
Server	发送该接收内容的服务器的名称
StatusCode	接收内容的HTTP状态代码
StatusDescription	接收内容中所用的HTTP文本状态定义

程序员既可以用单独的报头字段属性去处理某个报头字段，提取该字段的信息；也可以使用Header属性创建一个WebHeaderCollection对象，与WebClient的ResponseHeader属性相似，这个对象包含了HTTP返回报文的全部报头信息。

提取特殊的报头字段

虽然HttpWebResponse类具有很多属性，可以提取常见的HTTP报头信息，但是在客户机收到的报文中仍然会有很多其他种类的报头字段。为了处理这些特殊的字段，需要用到GetResponseHeader()方法，该方法将报头字段的内容视为字符，并以字符串的格式返回字段的内容。

```
string header = hwr.GetResponseHeader("X-Cache");
```

上例用来提取一个名为X-Cache的特殊HTTP报头字段，这个字段用来说明HTTP代理服务在寻找该URI时，究竟是从本地缓冲区中找到的，还是从网上找到的。

提取数据

HttpWebRequest对象使用GetResponse()方法时，会返回一个HttpWebResponse对象。为了提取出HTTP返回报文中的信息，需要使用GetResponseStream()方法，该方法会返回一个Stream对象，可以读取HTTP返回报文。创建这个Stream对象后，可以创建一个StreamReader对象，这样，就可以对HTTP返回报文中的HTML源代码进行逐行的处理。

处理Cookie

借助于Cookie属性，可以查看Web服务器发送到客户机的Cookie的全部内容。Cookie是一种用来记录信息的文件，由Web服务器创建并发送到客户机，Web服务器可以读取Cookie，并通过其信息判断和跟踪客户机的状态。举例而言，网上商店常常用Cookie来记录顾客放入虚拟购物车内的商品情况。

提取一次HTTP会话中Cookie的内容需要两个步骤：

1. 在发出HTTP请求之前，用HttpWebRequest类的CookieContainer属性定义一个CookieContainer对象，内容为空：

```
HttpWebRequest hwr =  
(HttpWebRequest)WebRequest.Create("http://www.amazon.com");  
hwr.CookieContainer = new CookieContainer();
```

2. 按照以前介绍的方法，用HttpWebResponse类提取来自Web服务器的网页，不过必须要把HttpWebResponse对象的Cookie属性赋值给第一步中创建的HttpWebRequest对象中的CookieContainer对象。

```
HttpWebResponse hwrsp = (HttpWebResponse)hwr.GetResponse();  
hwrsp.Cookies = hwr.CookieContainer.GetCookies(hwr.RequestUri);
```

经过以上两个步骤，HttpWebResponse对象的Cookies属性中就包含了与本HTTP会话相关的全部cookie的内容，可以用foreach语句逐行访问Cookie的内容。

```
foreach(Cookie cky in hwrsp.Cookies)  
{  
    Console.WriteLine(cky.Name + " = " + cky.Value);  
}
```

高级Web Client实例

到目前为止，.NET类库所提供的高级Web功能都已介绍完毕，下面将通过一个示例程序，说明如何向Web服务器发送请求，以及如何处理服务器返回的信息。程序清单14.10所示即为示例程序WebGet.cs，这个程序使用HttpWebRequest类和HttpWebResponse类与Web服务器通信，然后从返回的报文中提取报头字段、cookie和HTML代码。

程序清单14.10 WebGet.cs程序

```
using System;  
using System.Drawing;
```



```
using System.IO;
using System.Net;
using System.Windows.Forms;

class WebGet Form
{
    private TextBox uribox;
    private ListBox headers;
    private ListBox cookies;
    private ListBox response;

    public WebGet()
    {
        Text = "WebGet - a web page retriever";
        Size = new Size(500, 450);

        Label label1 = new Label();
        label1.Parent = this;
        label1.Text = "URI:";
        label1.AutoSize = true;
        label1.Location = new Point(10, 23);

        uribox = new TextBox();
        uribox.Parent = this;
        uribox.Size = new Size(200, 2 * Font.Height);
        uribox.Location = new Point(35, 20);

        Label label2 = new Label();
        label2.Parent = this;
        label2.Text = "Headers:";
        label2.AutoSize = true;
        label2.Location = new Point(10, 46);

        headers = new ListBox();
        headers.Parent = this;
        headers.HorizontalScrollbar = true;
        headers.Location = new Point(10, 65);
        headers.Size = new Size(450, 6 * Font.Height);

        Label label3 = new Label();
        label3.Parent = this;
        label3.Text = "Cookies:";
        label3.AutoSize = true;
        label3.Location = new Point(10, 70 + 6 * Font.Height);

        cookies = new ListBox();
        cookies.Parent = this;
        cookies.HorizontalScrollbar = true;
        cookies.Location = new Point(10, 70 + 7 * Font.Height);
    }
}
```

```

cookies.Size = new Size(450, 6 * Font.Height);

Label label4 = new Label();
label4.Parent = this;
label4.Text = "HTML:";
label4.AutoSize = true;
label4.Location = new Point(10, 70 + 13 * Font.Height);

response = new ListBox();
response.Parent = this;
response.HorizontalScrollbar = true;
response.Location = new Point(10, 70 + 14 * Font.Height);
response.Size = new Size(450, 12 * Font.Height);

Button sendit = new Button();
sendit.Parent = this;
sendit.Text = "GetIt";
sendit.Location = new Point(275, 18);
sendit.Size = new Size(7 * Font.Height, 2 * Font.Height);
sendit.Click += new EventHandler(ButtonGetitOnClick);
}

void ButtonGetitOnClick(object obj, EventArgs ea)
{
    headers.Items.Clear();
    cookies.Items.Clear();
    response.Items.Clear();

    HttpWebRequest hwr = (HttpWebRequest)WebRequest.Create(uribox.Text);
    hwr.CookieContainer = new CookieContainer();
    HttpWebResponse hwrsp = (HttpWebResponse)hwr.GetResponse();
    WebHeaderCollection whc = hwrsp.Headers;
    for (int i = 0; i < whc.Count; i++)
    {
        headers.Items.Add(whc.GetKey(i) + " = " + whc.Get(i));
    }

    hwrsp.Cookies = hwr.CookieContainer.GetCookies(hwr.RequestUri);
    foreach(Cookie cky in hwrsp.Cookies)
    {
        cookies.Items.Add(cky.Name + " = " + cky.Value);
    }

    Stream strm = hwrsp.GetResponseStream();
    StreamReader sr = new StreamReader(strm);

    while (sr.Peek() > -1)
    {
        response.Items.Add(sr.ReadLine());
    }
}

```



```
    }  
    sr.Close();  
    strm.Close();  
}  
  
public static void Main()  
{  
    Application.Run(new WebGet());  
}
```

WebGet程序生成一个简单的窗口，其中有三个ListBox对象：一个用于显示返回报文中的HTTP报头信息，一个用来显示返回报文中的cookie，另一个显示返回报文中的HTML源代码。

ButtongetitOnClick()方法中囊括了全部与Web相关的代码，该方法创建一个WebHeader-Container对象来处理HTTP报头字段，创建一个CookieContainer对象处理cookie。用WebGet程序访问www.amazon.com时的情形如图14.1所示。

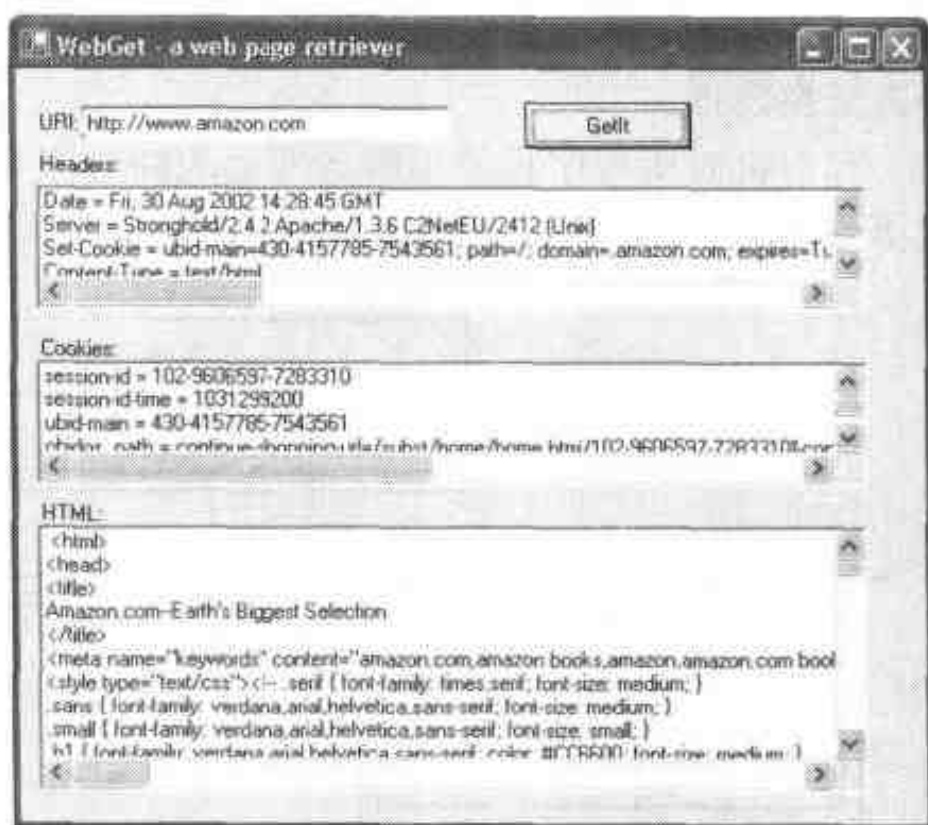


图14.1 WebGet的运行结果

警告：WebGet程序不支持代理服务器的环境，读者如果需要使用代理服务器，可以自行添加WebProxy项目。

网络服务

网络服务是.NET Framework众多新增功能之一，它可以让Web服务器在网络上公告指定的方法，供远程应用程序应用。程序员们可以用它实现分布式计算功能，商业企业可以用它向使用网络服务客户机程序的用户发送信息，而客户们也可以自己编写应用程序，并通过

网络服务从商业企业那里获取信息。

说明： 本节主要讲述如何在Microsoft IIS服务器上使用ASP.NET的网络服务功能。如果读者想运行本节中的示例程序，必须先安装IIS环境，并安装.NET Framework包，或者是访问那些已支持网络服务服务器。

.NET的网络服务特点是，它使用ASP.NET的技术来发布网页上的C#方法，这样，远程客户机就可以知道C#方法的存在，进而使用这些方法。在向网络服务发送数据时，需要用到简单对象访问协议（Simple Object Access Protocol——SOAP），SOAP与XML（eXtensible Markup Language，可扩展标志语言）一起，利用一个标准的HTTP连接，格式化Web服务器吞吐的数据。由于所有的数据都是通过HTTP来传输，应用程序之间可以容易地穿越防火墙和其他网络设备进行通信。

网络服务应用程序由三部分组成，如图14.2所示。

网络服务服务器 网络服务服务器使用一台运行ASP.NET的标准IIS Web服务器提供一个或多个应用程序需要的方法。这些方法可以通过SOAP协议接收来自远程客户机程序的数据，进行数据处理，然后将结果通过SOAP协议返送给远程客户机程序，返回的数据全部采用XML格式。

代理对象 用网络服务方法的数据定义功能可以创建代理对象，远程客户机程序用它与网络服务服务器交换数据。

客户机应用程序 客户机应用程序将使用服务器端的网络服务方法，这种应用程序可以写成需要从网络服务服务器获取信息的本地Windows程序。

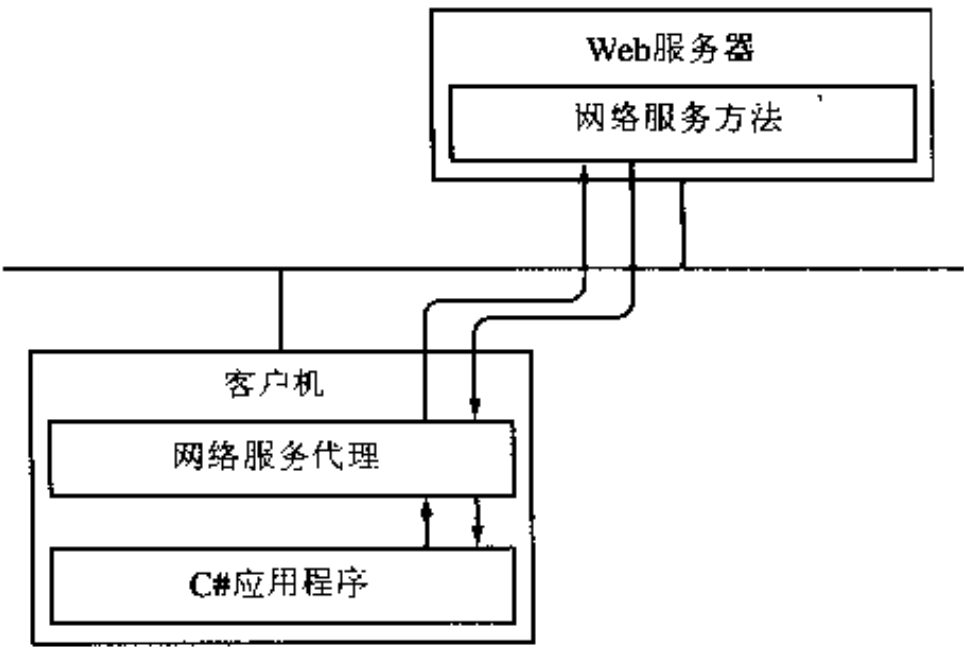


图14.2 网络服务对话框

创建网络服务服务器

要创建一个网络服务，首先要编写可以在IIS服务器上运行的C#方法，然后将C#方法融入ASP.NET网页中。网络服务网页文件的后缀名必须是.aspx。程序清单14.11中所示为MathService.aspx程序，可以在网络服务中实现一些简单的数学计算功能。

程序清单14.11 MathService.asmx程序

```
<%@ WebService Language="c#" Class="MathService"%>

using System;
using System.Web.Services;

[WebService(Namespace="http://localhost/test")]
public class MathService : WebService
{
    [WebMethod]
    public int Add(int a, int b)
    {
        int answer;
        answer = a + b;
        return answer;
    }

    [WebMethod]
    public int Subtract(int a, int b)
    {
        int answer;
        answer = a - b;
        return answer;
    }

    [WebMethod]
    public int Multiply(int a, int b)
    {
        int answer;
        answer = a * b;
        return answer;
    }

    [WebMethod]
    public int Divide(int a, int b)
    {
        int answer;
        if (b != 0)
        {
            answer = a / b;
            return answer;
        } else
            return 0;
    }
}
```

除了几处小区别以外，**MathService**程序和一般的C#程序看起来差不多。程序的第一行声明了程序的类型、编程语言（比如C#、Visual Basic等）和该服务程序的类名称，目的是表明该程序是一个网络服务程序。

```
<%@ WebService Language="c#" Class="MathService"%>
```

声明网络服务之后，必须为程序中的方法分配唯一的名称，即建立命名空间。一般情况下，多用网络服务所在服务器的URI来标识，如果没有定义命名空间，系统将自动用http://tempuri.org作为全局性的命名空间：

```
[WebService(Namespace="http://localhost/test")]
```

在这个简单的例子中，用**localhost**作了网络服务方法的名称。实际编程时，需要用实际Web服务器的URI来标识。

定义命名空间之后，接下来需要逐个声明网络服务的全部方法。每个方法都必须加上一个**WebMethod**标记，以表明该方法是一个网络服务方法。普通的C#方法也要进行声明，返回值、返回数据类型，以及用来向方法传递消息的所有参数都要进行声明。

测试网络服务

创建**MathService.asmx**程序之后，需要将其存放到IIS服务器的wwwroot目录下。为了与服务器上的其他网页相区别，可以为其建立一个单独的子目录，比如下面的test目录。

```
C:\Inetpub\wwwroot\test\MathService.asmx
```

保存了网络服务文件，接下来就可以在IE中通过响应的URI来访问它（本例是http://localhost/test/MathService.asmx），这时会出现一个窗口，显示可用的网络服务方法，窗口如图14.3所示。

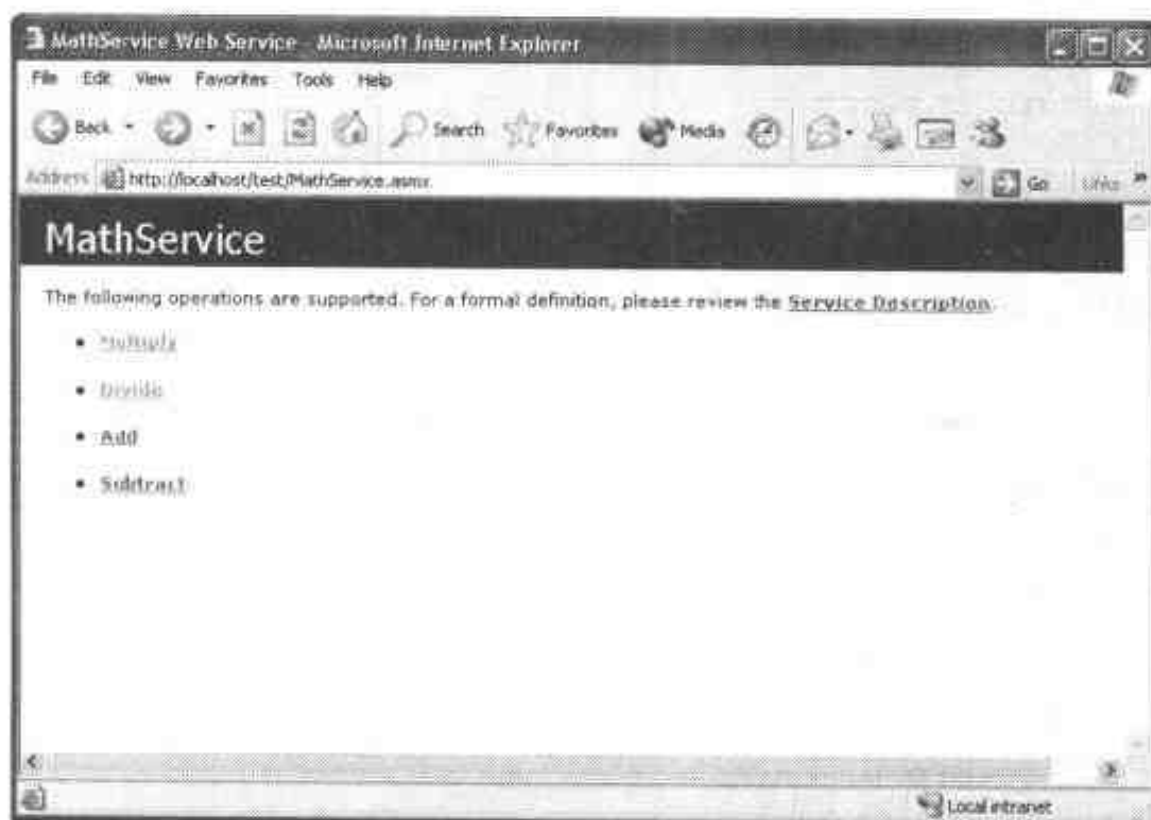


图14.3 MathService网络服务主页面

单击页面中任何一个方法，都将看到该网络服务方法的帮助页面，在这个帮助页面中可以对方法进行测试。图14.4所示为方法Multiply()的帮助页面，为两个参数选定一组值然后单击Invoke按钮，如果运行正常，就会启动一个新的IE窗口显示出XML格式的运行结果。请注意，在显示的XML结果中还包含有数据类型方面的信息，这就是XML的独到之处。

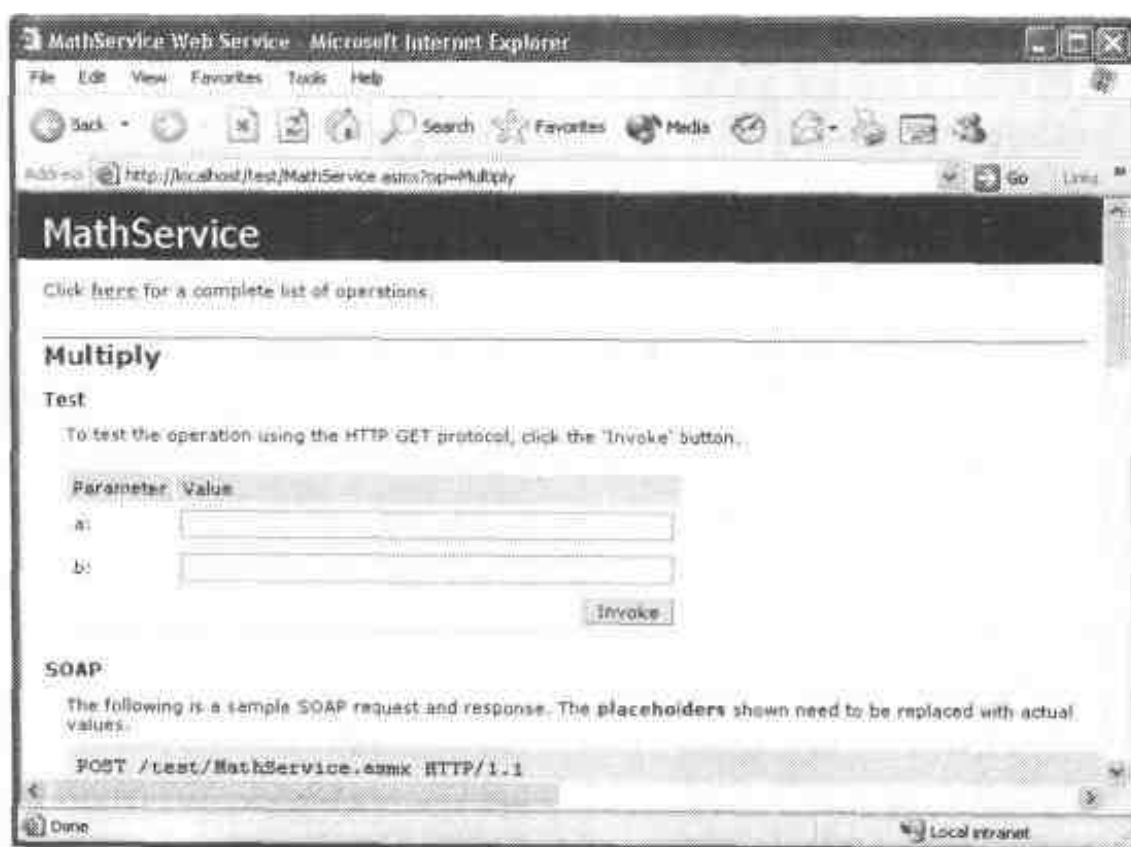


图14.4 Multiply()方法的帮助页面

创建网络服务代理

完成了对网络服务的测试之后，就要创建一个代理类库来供未来的C#客户机程序使用。用.NET Framework SDK中的wsdl程序可以完成这个工作，该程序首先连接到网络服务，然后生成一个.cs文件定义该网络服务的所有可用方法。默认情况下，wsdl程序会以网络服务的类名作为.cs文件的名称。下面的命令将会创建名为MathService.cs的文件：

```
C:\>wsdl http://localhost/test/MathService.asmx
```

读者可以仔细研究一下这个文件的内容，以便了解wsdl程序如何创建方法，完成与网络服务服务器的连接，以及如何将有关的参数传递给该方法。

MathService.cs文件创建之后，可以在创建客户机应用程序时合并编译，或者以它为基础生成一个DLL库文件：

```
csc /t:library MathService.cs
```

这条命令将创建MathService.dll文件，用来生成C#客户机程序。新的文件中包含了用同步和异步方法访问网络服务时需要用到的方法。

创建C#网络服务客户机程序

到此为止，网络服务服务器和代理类库都已经创建完成，接下来应该创建一个客户机程序来访问网络服务。程序清单14.12即是这样一个程序，名为ServiceTest.cs。

程序清单14.12 ServiceTest.cs程序

```
using System;

class ServiceTest
{
    public static void Main (string[] argv)
    {
        MathService ms = new MathService();

        int x = Convert.ToInt16(argv[0]);
        int y = Convert.ToInt16(argv[1]);

        int sum = ms.Add(x, y);
        int sub = ms.Subtract(x, y);
        int mult = ms.Multiply(x, y);
        int div = ms.Divide(x, y);
        Console.WriteLine("The answers are:");
        Console.WriteLine("  {0} + {1} = {2}", x, y, sum);
        Console.WriteLine("  {0} - {1} = {2}", x, y, sub);
        Console.WriteLine("  {0} * {1} = {2}", x, y, mult);
        Console.WriteLine("  {0} / {1} = {2}", x, y, div);
    }
}
```

如上例所示，网络服务一旦创建，使用它实在是非常简单。只要创建一个网络服务的实例，然后就可以像对付其他类方法那样使用它。

在编译上例程序时，必须要引用**MathService.dll**类：

```
csc /r:MathService.dll ServiceTest.cs
```

编译好的程序可以在网络上任何地方运行：

```
C:\>ServiceTest 100 50
```

```
The answers are:
```

```
100 + 50 = 150
```

```
100 - 50 = 50
```

```
100 * 50 = 5000
```

```
100 / 50 = 2
```

```
C:\>
```

小结

本章主要介绍如何在C#网络编程中使用**Web**资源。**HTTP**则是人们共享数据时的一种简单、易用的手段。

HTTP有两个消息模式——请求消息和响应消息。客户机向**Web**服务器发出请求消息，

指定要访问的网络资源；Web服务器向客户机发送响应消息，返回客户机想要的**数据**。在响应消息中还包含有一个状态代码，表示请求是否被接受，或者发生了错误。如果请求被接受，那么响应消息的主体部分通常会包含很多的**HTML**代码。

借助.NET网络类库，可以编写出功能完善的网络程序。**WebClient**类面向简单的Web访问，绝大多数基本的Web功能都包含在其中，因此它适合用于编写简单的网络程序。**WebRequest**和**WebResponse**及其子类面向更高级的Web应用，这些类中实现了基于多种协议的很多额外属性和方法。譬如对于HTTP通信，可以使用**HttpWebRequest**和**HttpWebResponse**类，这些类功能完善，可以用来处理报文的报头字段和**cookie**。

在本章的最后，介绍了.NET的新概念——网络服务。使用网络服务技术，可以在HTTP环境下访问类方法。远程客户机程序可以借助**SOAP**协议访问远程主机上的类方法。在网络服务传输过程中，所有的数据都采用**XML**格式，这使得不同种类的计算机可以自由地共享数据类型很复杂的信息。

下一章将讲解网络目录。如果读者正在编写Windows网络应用程序，那么极有可能是在Windows网络环境下工作。**Microsoft**活动目录系统可以控制很多网络资源以及对这些资源的访问，甚至包括用户名和密码。使用**C#**，可以很容易掌握活动目录，并且能熟练地进行用户信息处理。

第15章 活动目录

- 网络目录基础
- 用活动目录工作
- 使用C#访问网络目录
- 修改目录数据
- 搜索网络目录

在本书第三部分，读者已经学习了如何使用.NET网络库通过标准网络协议与远程网络设备通信。本章与前文不同，将讨论另一种类型的网络通信协议。Microsoft公司的活动目录（AD）用于在Windows 2000服务器环境中管理网络资源，包括文件共享、网络打印机、用户名和密码。如果要编写用在Windows 2000服务器环境中的网络程序，可以利用AD数据库来查找、添加和修改网络中的用户和系统。.NET库包括特殊的类来帮助使用AD数据库访问网络数据，比如用户信息。

本章首先大致回顾一下网络目录服务，描述网络目录服务中的轻型目录访问协议（Lightweight Directory Access Protocol, LDAP）。读者将看到如何在Windows网络环境中使用AD来控制网络信息。接下来深入讨论.NET的AD库，演示在网络目录中如何使用每一个类。

网络目录基础

网络目录服务技术已经成为计算机网络中最大的热点之一。它允许网络中的多个主机从网络上的公共数据库中共享各种信息，如应用程序、资源和用户。Novell公司的Netware目录服务和Microsoft公司的AD就是两个网络目录服务的例子。其他很多公司正在开发控制网络资源的技术。

因特网协会开发了一种标准的网络目录服务协议，任何TCP/IP网络上的主机或者用户都可以用它来访问目录服务数据库中的信息。由RFC定义的轻型目录访问协议（LDAP）已经广泛应用。任何使用LDAP的用户可以访问任何使用LDAP的网络服务数据库。Microsoft公司的AD就是基于LDAP模式的，在了解到LDAP是如何工作之后理解AD就很容易了。

LDAP系统

LDAP数据库基于分级数据库而设计，类似于DNS数据库存储域名的方式。分级数据库通常用于匹配组织机构，因为它们可以很容易地体现出公司的层次。它们因其飞快的访问速度而知名，不过写入速度很慢。分级的方法假设一旦一个对象写入到数据库中，不同的网络用户会多次访问它。这非常符合很多网络目录的情况，比如在数据库中生成一个用户名和密

码以后，用户每次试图使用服务器上的网络资源的时候，网络服务器就会访问这些记录。

LDAP数据库中的对象在树形结构中与其他对象相连，如图15.1所示。每个LDAP树必须有一个根对象，其他对象依次与之相连。对象根据引用与其他对象相连，类似于DNS命名系统。

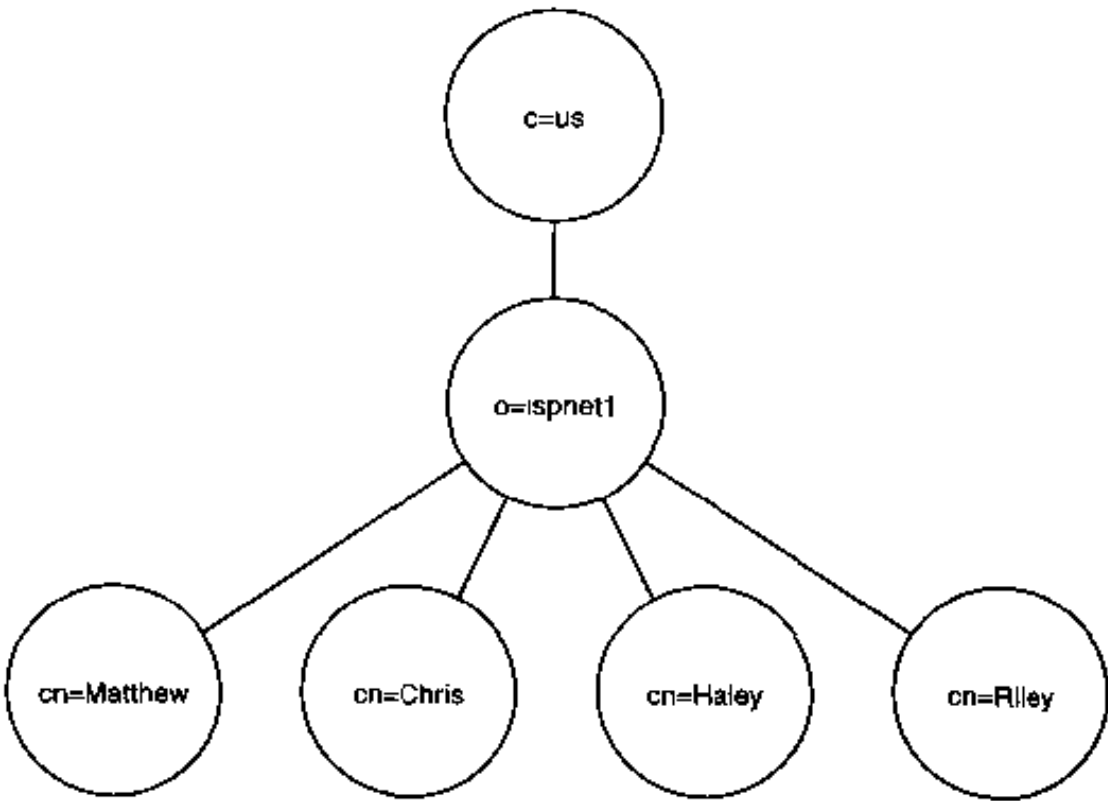


图15.1 LDAP数据库的一个例子

由于LDAP数据库与DNS的结构很相似，很多LDAP产品（包括AD）完全按照DNS系统的方式设计LDAP数据库，只要像DNS系统中顶级域名和子域名那样给数据库中的对象命名就可以了。

对象和属性

LDAP树使用LDAP类型和值引用数据库中的每个对象。LDAP类型指的是数据库中的对象类型。对应LDAP数据库对象定义了不同对象类型。大多数源自标准X.500网络命名惯例，如表15.1所示。

表15.1 X.500对象类型

对象类型	描述
C	一个国家对象
O	一个组织对象
OU	一个组织单元对象
CN	一个普通名称对象
DC	一个目录上下文对象

一个对象的类型和值在数据库中标识这个对象。例如，条件o=ispnet1表示Organization对象ispnet1。除此之外，LDAP数据库必须能够在数据库中惟一标识每个对象。每个对象由它们的标识名（distinguished name, dn）惟一参照。dn合并了树的所有对象，从对象到树的根。从图15.1的例子可以看出ispnet1组织使用dn为o=ispnet1, c=us。这保证了c=us对象下的其他对象的值不会为ispnet1。

尽管如此，很多公司使用目录上下文对象在数据库中引用他们的DNS域对象，而不用组织和country类型识别对象。例如下面的例子中，dn引用对象包含信息为ispnet1域中的engineering组织单元里的用户名rmullen：

```
cn=rmullen, ou=engineering, dc=ispnet1, dc=net
```

目录上下文和组织单元对象被称做容器对象，因为它们可以包含其他对象，比如其他DC和OU对象。普通名称对象通常用来定义用户对象并且可以包含在任意类型的容器对象之中。

LDAP数据库中的每个对象在网络上都表现为一个实体。LDAP数据库中的对象通过两个要素实现对象定义——对象类名称和相关的属性集。对象类名称显示该对象所标识的信息种类。

在LDAP中使用了很多objectClass。表15.2列出最常用的一些。

表15.2 常用LDAP objectClass

objectClass	说明
country	表示国家对象
locality	表示城市、县或者州对象
organization	表示组织或者商业机构
organizationPerson	表示组织中的某个人
organizationRole	表示组织中的某个具有特殊角色的人（比如管理员）
organizationUnit	表示组织中的一个实体
person	表示普通的个人实体
residentialPerson	表示某个人的个人信息，如地址和电话号码
applicationEntity	表示通过网络访问的应用

每个objectClass包含存储了对象信息的属性。例如，person objectClass包含cn（普通名称）属性，代表个人姓名（如Rich Blum），sn（姓）代表此人的姓（如Blum），telephoneNumber存储此人的联系电话。在LDAP数据库中查询cn=rblum, dc=engineering, dc=ispnet1, dc=net对象会看到相关对象的属性，查到被查询者的信息。

说明：注意cn属性值与CN普通名称对象类型是不同的。

用活动目录工作

Microsoft公司的活动目录（AD）系统取代了原先的Windows NT域系统。在发布Windows 2000服务器版本的同时，Microsoft公司将域系统转换为LDAP数据库模式。与在单层数据库中控制域信息不同，分层网络目录服务系统允许更复杂的网络构造。AD数据库包含代表原先域数据库中所有网络资源的对象。对象属性用来跟踪每个对象的特殊信息，如用户名、密码、登录时间和个人信息。

网络管理员使用AD存储网络目录数据库中域的每个项目的服务器、工作站和网络打印机信息。

活动目录的各个部分

AD系统由几个部分组成，用来从域中存储和获取网络对象。

域

域中AD的概念与原来的Windows NT的概念类似。AD域包含一组共享同一个数据库结构和安全策略的网络资源。在AD系统中，网络管理员使用分层数据库在域中的目录上下文内组织网络资源。这样的域管理方式比Windows NT域的方法简单得多。

在大多数AD网络中，机构的DNS名用做域名。一个具有简单的DNS域名的机构可以定义为一个简单的AD域。从图15.2中可以看出如何实现。

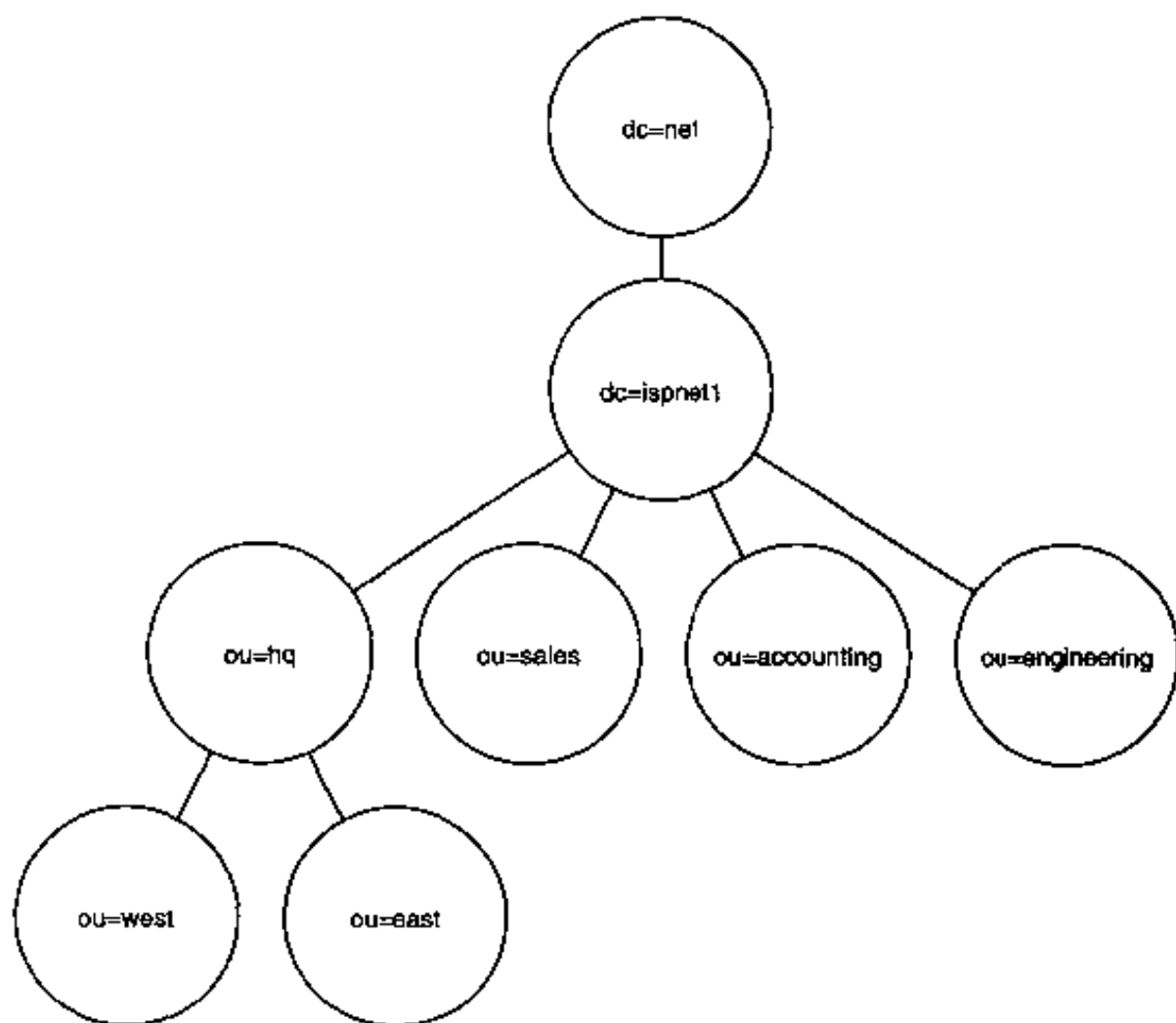


图15.2 简单的活动目录域

组织单元

域本身可以细分为更小的实体。组织单元（OU）是域中聚合相关对象的容器（类似于LDAP中的目录上下文）。通常OU基于机构的商业结构，每个商业单元（或者分公司）的网络资源对象都在他们自己的OU容器中。

OU的一个好的特性是AD允许网络管理员在这一层次委托本地管理员。可以从本地OU中指派一个用户管理组织单元中所有的资源，而不需要担心它们同时会得到AD数据库其他部分的安全权限。

树

在Windows NT域系统中，访问位于其他域中的网络资源意味着网络管理员要设置复杂的域安全规则。如果想从一个域访问另一个域的资源，被访问的域必须与远程域建立信任关系。

信任关系是定义在域中的一组规则，允许来自某个域的用户（或其他网络对象）访问另一个域的网络资源。一个具有多个域的机构要想允许每个域可以访问其他所有域，必须在每组相关的两个域之间都建立信任关系。这会很快成为管理员巨大的烦恼。

为了解决这个问题，AD系统产生了树的概念。树将分散的域互相联系起来，全部由一个单一的AD数据库控制。指定树中的一个域为主域（称为根域），其他域设置为根域下面的对象（类似于基本LDAP结构）。树中的所有域共享共同的根域命名空间。

对于使用DNS域和子域的机构来说，可以很容易地为网络中的每个子域定义AD树。图15.3给出一个例子，关于如何在AD树中定义子域系统。正如图15.2显示的那样，每个三角关系表示一个自包含AD域。根域ispnet1.net包含的对象用于控制整个树（域管理员用户和主AD服务器）。每个子域是一个自包含域，包括本地域用户和服务器对象。

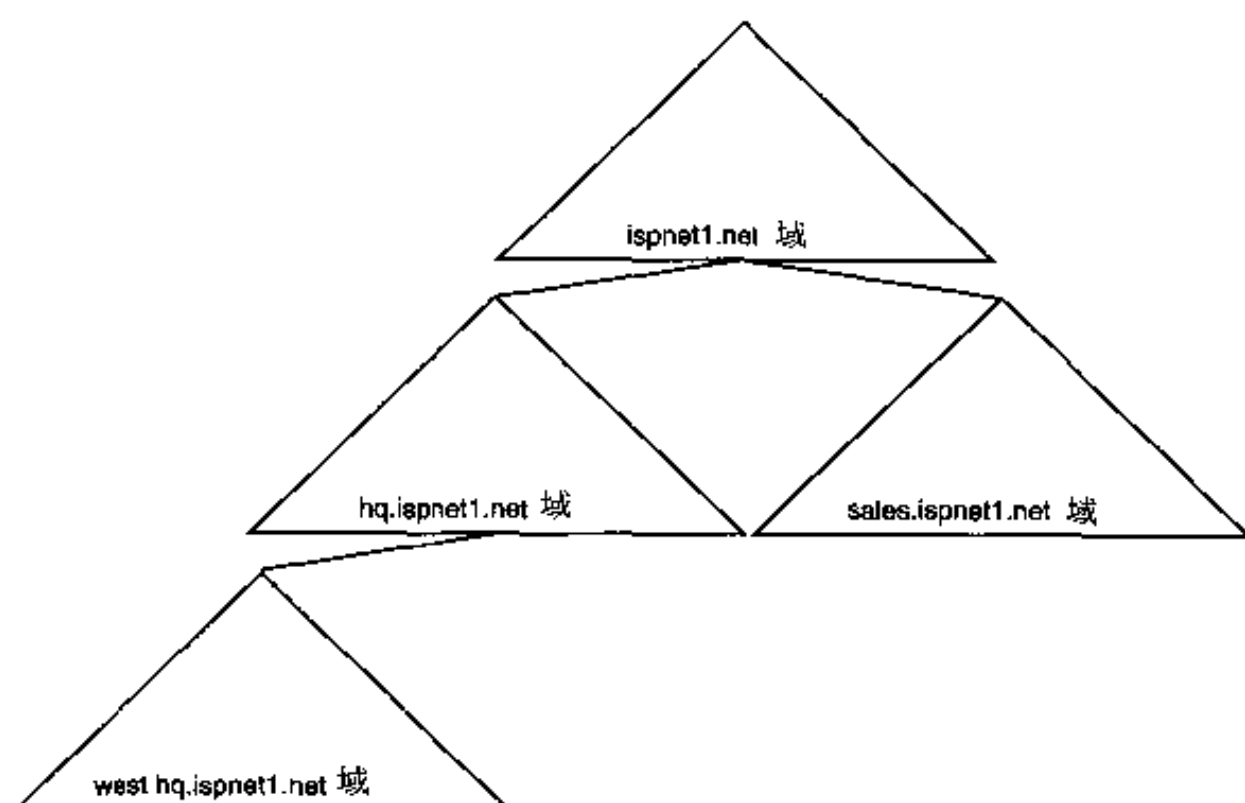


图15.3 AD树结构

森林

由于树的概念允许一个机构将机构中邻近的域连接到一个单一的AD数据库中，往往会发生这样的情况，机构包含两个或者更多必须严格控制的树。为了解决这种问题，AD要合并为森林的概念。森林集中两个或者更多合并到相同AD数据库中的树。尽管树使用同一数据库，它们没有使用同一命名空间，这就是说，每个树仍然保持它自己的根对象，并且基于树的根对象引用。图15.4给出这种排列的一个例子。

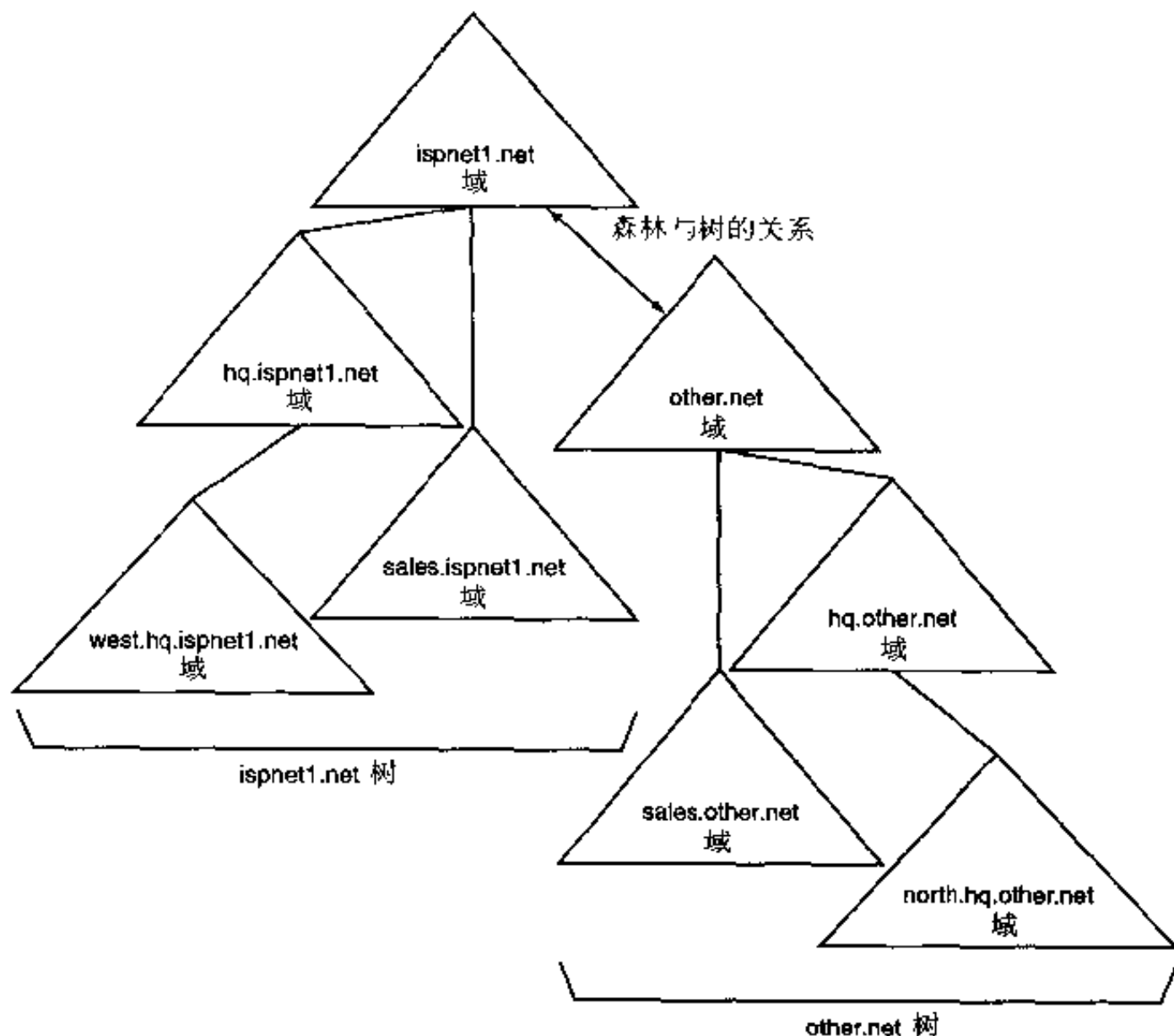


图15.4 具有两个树的AD森林结构

对于网络管理员来说，很困难的是决定将域合并为树还是森林的时候。非常重要的一点是，在森林中，每个树要被其他树访问网络资源的话必须保持与其他树之间的信任关系（类似于以前的NT域的信任关系方式）。对于合并而且共享实质资源的机构来说，将分散的树合并为单一的树比生成森林处理信任关系简单多了。

活动目录对象

AD系统类似于LDAP，使用的某些术语和概念不同。目录结构中的基本项目仍然称为对象。对象表示网络中的简单实体，比如服务器、工作站、打印机或者用户。每个对象由类来定义，也称为对象类。这与LDAP的对象类的概念很相似。每个对象类由用于掌握对象信息的特殊属性组成。

连接到活动目录服务器

要想与Windows 2000服务器连接，工作站必须运行AD服务接口程序（ADSI）。所有与AD服务器的通信，无论来自操作系统还是系统上的用户程序，都要经过（ADSI）。.NET的AD库对其所有的AD功能使用ADSI。

要使一个网络设备在AD中工作，必须装载ADSI软件库。所有的Windows 2000专业版工作站和服务设备，以及Windows XP专业版工作站，都包括对应AD服务器的ADSI。Windows 98、Me和NT系统不包括ADSI。可以从Microsoft公司为这三种系统下载ADSI客户机程序包（还有任何运行在这些系统上的AD程序），使这些系统也可以工作在AD环境中。

使用C#访问网络目录

在网络目录服务对象中主要使用的.NET类是DirectoryEntry类，是在System.DirectoryServices命名空间中建立的。DirectoryEntry类允许绑定到一个目录对象上并且执行对此对象的获取、添加、删除和修改信息等操作。

DirectoryEntry类有很多构造函数格式，取决于数据库的需要。

说明：要执行本章给出的程序例子，读者需要访问一些LDAP数据库系统，Windows 2000活动目录服务器或者LDAP服务器。这些例子使用的是OpenLDAP服务器，而这里讨论的设计原理是普遍适用的，无论使用何种LDAP服务器。

匿名目录登录

很多网络目录服务允许网络上的任何人未经允许读取所有或者某些数据库对象。这会允许网络用户不需要目录服务器的高级权限就可以查看网络上的资源，比如用户电话号码和地址。

要生成DirectoryEntry，不使用权限参考目录对象，可以使用下面的构造函数格式：

```
DirectoryEntry(string ldappath)
```

ldappath参数值是一个字符串，表示数据库对象的位置。对象必须使用URI格式的语法参照，这种语法包括访问方式、网络目录服务地址和对象的显著名称。

ADSI库针对连接不同种类的网络目录服务提供很多访问方式。每种访问方式由URI格式指定。表15.3列出常用的访问方式。

警告：访问方式是区分大小写的，所以使用的时候要小心。

对于AD访问来说，应该使用LDAP访问方式，因为它提供了很多很好的访问方式，包括搜索能力（详见“搜索网络目录”一节）。

在访问方式之后要指定目标目录服务器地址，还要指定对象参考的完整标识名。以下是一些正确的LDAP路径的例子：

```
LDAP://server1.ispnet1.net/dc=ispnet1, dc=net
LDAP://server1.ispnet1.net/cn=kblum, ou=sales, dc=ispnet1, dc=net
LDAP://192.168.1.100/ou=accounting, dc=ispnet1, dc=net
```


表15.3 ADSI访问方法

访问方法	访问
WinNT	Windows NT域
IIS	微软的因特网信息服务服务器
LDAP	任意适用于LDAP的网络目录（包括活动目录）
NDS	Novell Netware目录服务服务器
NWCOMPAT	Novell Netware 3.x bindary服务

程序清单15.1给出BindObject.cs程序，它给目录对象绑定了一个变量并且显示对象关联的LDAP路径。

程序清单15.1 BindObject.cs程序

```
using System;
using System.DirectoryServices;

class BindObject
{
    public static void Main()
    {
        DirectoryEntry de = new DirectoryEntry(
            "LDAP://192.168.1.100/dc=ispnet1, dc=net");

        string ldappath = de.Path;
        Console.WriteLine("The LDAP path is: {0}", ldappath);
        de.Close();
    }
}
```

尽管BindObject.cs程序是很小的一个程序（视读者需要，来了解绑定到对象上的LDAP路径），它说明的是给目录服务对象绑定变量需要的基本步骤。Path属性获取对象的LDAP路径，是一个字符串值，可以用于DirectoryEntry绑定在对象上。这将在后面的“搜索网络目录”一节中通过目录服务中的对象再次讨论。

这个例子的结果非常简单。如果读者能成功地绑定到目录服务对象上，LDAP路径就会显示在控制台上。这同生成DirectoryEntry对象是一样的。如果程序不能绑定到目录服务对象，就会出现一个错误信息，说明错误原因。最常见的错误之一是目录服务服务器需要授权才能绑定到对象上。

说明：读者可能注意到很多AD代码产生的错误是关于COM错误的信息。ADSI库使用Windows的COM接口访问AD环境。

在目录中登录

需要用户权限的数据库操作可以使用两种形式。下面的构造函数允许指定用户名和密码登录到目录服务上:

```
DirectoryEntry(string ldappath, string username, string password)
```

一旦授权连接, 就可以进行指定的该用户名所允许的操作, 包括添加、删除或者修改对象。

第二种构造函数允许指定用于登录的特殊授权类型:

```
DirectoryEntry(string ldappath, string username, string password,
    AuthenticationTypes authtype)
```

AuthenticationTypes项指定用于登录到目录服务服务器上的权限类型。

表15.4列出了常用的权限类型。

表15.4 权限类型

权限类型	说明
Anonymous	未授权 (Windows NT下不支持)
Delegation	授权ADSI委托用户安全
Encryption	给所有与服务器之间的数据交换加密
FastBind	不试图查询objectClass属性, 只显露ADSI支持的基本接口
None	作为空的引用使用
ReadonlyServer	表示对服务器的只读访问权限
Sealing	使用Kerberos方法加密数据
Secure	请求安全授权
SecureSocketsLayer	用加密套接字协议层 (SSL) 和一个广为人知的证书
ServerBind	当服务器由LDAP路径指定时, 用于将会话记录到指定服务器上
Signing	标记所有的包来验证数据完整性

下面是使用授权访问目录对象的一个例子:

```
DirectoryEntry de = DirectoryEntry("LDAP://192.168.1.100/dc=ispnet1, dc=net",
    "rich", "password", AuthenticationTypes.ServerBind);
```

这个例子使用服务器用户名“rich”和正确的密码登录到目录服务服务器上。

说明: 几乎所有的网络目录服务在实现管理员功能 (如添加、删除和修改对象) 之前都需要用户登录。

修改目录数据

当给网络目录服务对象绑定一个变量之后，可以使用DirectoryEntry属性和方法获取对象，并且利用对象本身或者其属性。

警告：在.NET ADSI库中，对象的属性（properties）由attributes一词替代，二者含义相同，不要因为术语上的变化感到迷惑。

操作对象属性

DirectoryEntry对象的Properties属性访问所有关于对象的属性。Properties属性返回PropertyCollection类对象，可以用来共同作用于不同对象属性。

列出对象属性

PropertyCollection类包含PropertyNames和Values属性，获取现有的对象属性和它们的值：

```
DirectoryEntry de = new DirectoryEntry(
    "LDAP://192.168.1.100/dc=ispnet1, dc=net");
PropertyCollection pc = de.Properties
foreach(string propName in pc.PropertyNames)
    Console.WriteLine("property: {0}", propName);
```

要想获得objectClass属性值，可以使用Properties属性。使用属性名作为索引获取每个值。下面的声明获取对象的objectClass属性：

```
string value = de.Properties["objectClass"];
```

程序清单15.2列出的GetProperties.cs是一个简单的例子，介绍如何通过一个对象的所有属性重申并且显示所有属性值。

程序清单15.2 GetProperties.cs程序

```
using System;
using System.DirectoryServices;

class GetProperties
{
    public static void Main()
    {
        DirectoryEntry de = new DirectoryEntry(
            "LDAP://192.168.1.100/cn=kblum, ou=sales, dc=ispnet1, dc=net");
        Console.WriteLine("object: {0}", de.Path);
        PropertyCollection pc = de.Properties;
        foreach(string propName in pc.PropertyNames)
```

```

    {
        foreach(object value in de.Properties[propName])
            Console.WriteLine("  property = {0}    value = {1}",
                               propName, value);
    }
}

```

第一个foreach通过PropertyCollection重申，找到关于对象的所有属性名。第二个foreach取得关联于每个属性名的属性值。下面是这个例子的输出结果：

```

C:\>GetProperties
object: LDAP://192.168.1.100/cn=kblum, ou=sales, dc=ispnet1, dc=net
  property - objectClass  value = person
  property = cn    value = Katie Blum
  property = sn    value = Blum

C:\>

```

结果显示，object cn=kblum, ou=sales, dc=ispnet1, dc=net是一个个人objectClass并且有两个附加属性，一个普通名字Katie Blum，另一个是姓Blum。

修改属性

有了属性名之后，可以用它来修改赋给属性的值。只有一个缓存用在这里，系统可能会捕获数据库信息。

ADSI系统有一种机制可以捕获从目录服务服务器上获取的信息。对这个数据实施的任何操作都由系统缓存完成。有两种途径改变数据，影响目录服务数据库：

- 使用CommitChanges()方法迫使缓存中数据的变化传给目录服务服务器
- 设置UsePropertyCache属性为false，确保任何数据改变都会立刻发送给服务器

以上保存数据变化的任何一种方式都适合AD环境。如果改变数据库超过一定时间，可能需要使用UsePropertyCache属性，来确保所有改变在系统故障的情况下也会保存。另一方面，如果在短时间内修改大量数据，那么先在缓存中修改，然后使用CommitChanges()方法一次提交全部修改会比较快捷。

程序清单15.3给出ModifyProperty.cs程序，演示如何在目录服务数据库中修改对象属性值。

程序清单15.3 ModifyProperty.cs程序

```

using System;
using System.DirectoryServices;

class ModifyProperty
{
    public static void Main()

```

```

{
    DirectoryEntry de = new DirectoryEntry(
        "LDAP://192.168.1.100/cn=kblum, ou=sales, dc=ispnet1, dc=net",
        "cn=Administrator, dc=ispnet1, dc=net", "password",
        AuthenticationTypes.ServerBind);

    de.Properties["sn"][0] = "Mullen";
    de.CommitChanges();
    Console.WriteLine("New property value: {0}", de.Properties["sn"][0]);
    de.Close();
}
}

```

由于要修改数据库中的对象属性，可能必须包括目录服务服务器的用户名和密码（可能还有AuthenticationTypes值）。

注意属性值本身必须使用两维队列参照。队列的第一个元素是属性名，第二个元素是属性值索引。这说明属性可能具有不止一个值与其相关，不过这种情况并不多见。

添加属性

要向对象添加一个新的属性，必须使用PropertyCollection类的Add()方法。要添加的属性必须是对象的objectClass的有效属性。目录服务计划包括所有objectClass类型的正式定义，以及什么样的属性可以用于每个objectClass。

添加新属性最简单的方法是参照使用属性名为Properties队列索引的新属性，然后使用Add()方法：

```
de.Properties["telephoneNumber"].Add("(111)222-3333");
```

这个声明将telephoneNumber属性加入到DirectoryEntry对象de中并且赋值，详见程序清单15.4列出的AddProperty.cs。

程序清单15.4 AddProperty.cs程序

```

using System;
using System.DirectoryServices;

class AddProperty
{
    public static void Main()
    {
        DirectoryEntry de = new DirectoryEntry(
            "LDAP://192.168.1.100/cn=kblum, ou=sales, dc=ispnet1, dc=net",
            "cn=Administrator, dc=ispnet1, dc=net", "password",
            AuthenticationTypes.ServerBind);
        de.Properties["telephoneNumber"].Add("(111)222-3333");
    }
}

```

```

        de.Properties["telephoneNumber"].Add("(444)555-6666");
        de.CommitChanges();
        de.Close();
    }
}

```

编译并且运行AddProperty程序之后，可以使用GetProperties程序查看附加属性是否加入进去了：

```

C:\>GetProperties
object: LDAP://192.168.1.100/cn=kblum, ou=sales, dc=ispnet1, dc=net
property = objectClass    value = person
property = cn             value = Katie Blum
property = sn             value = Blum
property = telephoneNumber value = (111)222-3333
property = telephoneNumber value = (444)555-6666

C:\>

```

说明：这个例子也说明，通常一个对象的某个属性可以有多种形式。如果编写AD程序不重视这一点，这一特性会导致不完整的信息产生。一定要获取属性的全部，而不仅仅是第一个值。

作用于对象

在网络目录服务树中，任何目录容器对象都可以将其他对象作为子对象（这样树才能形成它的结构）。可以使用Children属性添加和删除一个对象的子对象。

列出子对象

Children属性返回一个DirectoryEntries对象，包含对象所有的子对象。然而，DirectoryEntries类是IEnumerator类型的类，所以可以用foreach声明从DirectoryEntries对象中取得每个DirectoryEntry对象。

程序清单15.5给出ListObjects.cs程序，列出一个基本对象的全部子对象。

程序清单15.5 ListObjects.cs程序

```

using System;
using System.Collections;
using System.DirectoryServices;

class ListObjects
{
    public static void Main()
    {
        DirectoryEntry de = new DirectoryEntry(

```

```
"LDAP://192.168.1.100/dc=ispnet1, dc=net");  
  
Console.WriteLine(de.Path);  
DirectoryEntries des = de.Children;  
foreach(DirectoryEntry entry in des)  
{  
    Console.WriteLine("  child: " + entry.Name);  
}  
}  
}
```

在目录服务数据库中，ListObjects程序给对象绑定一个DirectoryEntry对象。Children属性得到DirectoryEntries对象，然后列举查找每个单独的子对象。程序输出结果如下：

```
C:\>ListObjects  
LDAP://192.168.1.100/dc=ispnet1, dc=net  
child: cn=Administrator  
child: ou=accounting  
child: ou=engineering  
child: ou=sales  
child: ou=hq  
  
C:\>
```

注意所有的对象，无论它们是什么对象类型，都体现为基本对象的子对象。

添加一个子对象

可以使用Children属性的Add()方法来添加新的子对象并且使用CommitChanges()方法确保新的对象保存在目录服务数据库中。Add()方法格式如下：

```
Add(string objectname, string schemaclassname)
```

要添加一个新的对象必须了解新对象的目标计划objectClass。通常可以从目录服务数据库中一个类似的对象复制这个值。程序清单15.6是AddObject.cs程序，演示如何在现有目录结构中添加一个新的organizationalUnit对象。

说明：在这个例子中，从目录中一个现有类似对象获取计划objectClass。可以使用同样的方法，或者可以使用其他计划生成新的objectClass对象。

程序清单15.6 AddObject.cs程序

```
using System;  
using System.DirectoryServices;  
  
class AddObject  
{  
    public static void Main()  
    {
```

```

DirectoryEntry de = new DirectoryEntry(
    "LDAP://192.168.1.100/ou=accounting, dc=ispnet1, dc=net",
    "cn=Administrator, dc=ispnet1, dc=net", "password",
    AuthenticationTypes.ServerBind);

DirectoryEntries children = de.Children;
DirectoryEntry newchild = children.Add("ou=auditing", de.SchemaClassName);
newchild.Properties["ou"].Add("Auditing Department");
newchild.CommitChanges();
newchild.Close();
de.Close();

DirectoryEntry de2 = new DirectoryEntry(
    "LDAP://192.168.1.100/ou=auditing, dc=accounting, dc=ispnet1, dc=net");
string newpath = de2.Path;
Console.WriteLine("new path: {0}", newpath);
de2.Close();
}
}

```

AddObject程序首先绑定新对象的父对象并且从父对象的**Children**属性生成**DirectoryEntries**对象。**Add()**方法用在**DirectoryEntries**对象上，指定使用新对象的名字和**objectClass**（在这种情况下，**objectClass**是从父对象复制过来的，新对象也是一个**organizationalUnit**对象）。

新对象的区分名将仍然包括父对象名（在这个例子中**ou=auditing**，**ou=accounting**，**dc=ispnet1**，**dc=net**）。

警告：记住要以有权限的用户身份登录到服务器上，该权限要允许添加新对象。对于Windows 2000活动目录服务器来说，可能必须使用授权类型的安全技术来绑定到**DirectoryEntry**对象。

删除子对象

只能从AD中删除空的子对象。有两种方式从目录服务数据库中删除对象：

- 使用**DirectoryEntries Find()**和**Remove()**方法
- 使用**DirectoryEntry DeleteTree()**方法

使用DirectoryEntries方法

第一种技术使用**DirectoryEntries**类的**Find()**和**Remove()**方法首先定位然后删除目标对象。在下面的代码段中，可以看到如何根据对象名查找对象，生成**DirectoryEntry**对象，并且使用**Remove()**方法从目录中删除它：

```

DirectoryEntries children = de.Children;
DirectoryEntry badObject = children.Find("auditing");
if (badObject != null)
    children.Remove(badObject);
children.CommitChanges();

```


最后一步是确保改动传递到目录服务服务器中。

程序清单15.7列出RemoveObject.cs程序，使用这种技术删除子对象。这个程序给希望删除对象的父对象绑定一个变量，然后使用Find()方法给希望删除的对象绑定一个DirectoryEntry对象。接下来用希望删除对象的DirectoryEntry变量使用Remove()方法删除对象。

程序清单15.7 RemoveObject.cs程序

```
using System;
using System.DirectoryServices;

class RemoveObject
{
    public static void Main()
    {
        DirectoryEntry de = new DirectoryEntry(
            "LDAP://192.168.1.100/ou=accounting, dc=ispnet1, dc=net",
            "cn=Administrator, dc=ispnet1, dc=net", "password",
            AuthenticationTypes.ServerBind);

        DirectoryEntries children = de.Children;
        try
        {
            DirectoryEntry badObject = children.Find("ou=auditing");
            children.Remove(badObject);
            de.CommitChanges();
            Console.WriteLine("the object was removed");
        } catch (Exception)
        {
            Console.WriteLine("the object was not found");
        }
    }
}
```

使用DeleteTree()方法

DirectoryEntry类包括DeleteTree()方法，可以用于从目录服务数据库中删除绑定的对象。

警告： DeleteTree()方法只可以用在未绑定的DirectoryEntry对象，否则将出现错误提示。

程序清单15.8列出的DeleteObject.cs程序使用DeleteTree()方法。

程序清单15.8 DeleteObject.cs程序

```
using System;
using System.DirectoryServices;

class DeleteObject
{

```

```

public static void Main()
{
    DirectoryEntry de = new DirectoryEntry(
        "LDAP://192.168.1.100/ou=accounting, dc=ispnet1, dc=net",
        "cn=Administrator, dc=ispnet1, dc=net", "password",
        AuthenticationTypes.ServerBind);

    DirectoryEntries children = de.Children;
    try
    {
        DirectoryEntry badObject = children.Find("ou=auditing");
        badObject.DeleteTree();
        de.CommitChanges();
        Console.WriteLine("the object has been deleted");
    } catch (Exception e)
    {
        Console.WriteLine("the object was not found or deleted:");
        Console.WriteLine(e.ToString());
    }
}
}

```

使用DeleteTree()方法时，不能删除当前正在绑定的对象。应该像上面的例子那样，必须绑定到父对象，然后使用Find()方法寻找需要删除的子对象。

警告： DeleteTree()方法不允许删除不是其他对象的容器的对象。

重命名子对象

Rename()方法重命名目录对象。Rename()方法的惟一参数是赋给对象的新名称：

```
de.Rename("cn=testing");
```

警告： 使用Rename()方法要谨慎。它不仅影响重命名的对象，还影响目录树中该对象下面对象的区别名。

程序清单15.9列出的RenameObject.cs程序演示如何使用Rename()方法重命名Person对象。

程序清单15.9 RenameObject.cs程序

```

using System;
using System.DirectoryServices;

class RenameObject
{
    public static void Main()

```



```
{
    DirectoryEntry de = new DirectoryEntry(
        "LDAP://192.168.1.100/ou=auditing, ou=accounting, cc=ispnet1, dc=net",
        "cn=Administrator, dc=ispnet1, dc=net", "password",
        AuthenticationTypes.ServerBind);

    DirectoryEntries des = de.Children;
    DirectoryEntry badObject = des.Find("cn=test");
    badObject.Rename("cn=testing");
    de.CommitChanges();
    de.Close();
}
}
```

搜索网络目录

在网络目录服务中最频繁使用的功能之一是搜索信息。为使这个功能更简单地实现，.NET库包含DirectorySearcher类，提供充足数量的属性来帮助定义搜索和获取结果。

第1步：定义搜索属性

DirectorySearcher类的四种属性控制如何执行搜索。

SearchRoot

SearchRoot属性定义从目录结构中开始搜索的位置。SearchRoot值可以使用标准URI技术指定DirectoryEntry对象：

```
SearchRoot = "LDAP://192.168.1.100/dc=ispnet1, dc=net";
```

对于一些网络目录服务（包括AD）来说，会赋给根对象一个特殊名。RootDSE对象可以在目录中引用根对象：

```
SearchRoot = "LDAP://192.168.1.100/RootDSE";
```

如果SearchRoot属性没有设值，程序会试图寻找根对象并且从那里开始搜索。

Filter

默认情况下，DirectorySearcher对象会返回目录服务数据库中的全部对象。Filter（过滤器）属性可以定义什么样的对象会返回到搜索结果中。

Filter属性是一个字符串值，必须括在括号中。默认过滤值如下：

```
(objectClass=*)
```

这个值返回数据库中全部objectClass对象（过滤值可以使用通配符）。还可以选择过滤任何objectClass，或者任何在数据库中定义的对象属性。例如：

(objectClass=person) 返回具有person的objectClass的全部对象

(sn=B1*) 返回以B1开始的姓的全部对象

(sn>=Blum) 返回字母顺序大或等于Blum的姓的全部对象

最后的例子说明任何形式的数学表达式都可以用在过滤值中。也可以指定多重过滤值，同boolean操作一同使用。例如，下面返回的对象具有person的objectClass，以及sn值为Blum：

```
(&(objectClass=person)(sn=Blum))
```

同样的，可以使用多重boolean操作：

```
(&(objectClass=person)(!(sn=Blum)(sn=Pierce)))
```

混合返回person的objectClass的全部对象和姓为Blum或者Pierce的对象。

PropertiesToLoad

默认情况下，DirectorySearcher对象返回关联于对象的全部属性。对于某些对象来说，返回的数据太多。如果只对一部分属性感兴趣，可以指示DirectorySearcher返回何种数据，节约时间和网络带宽。PropertiesToLoad属性是StringCollection对象，使用Add()方法给获取的搜索结果添加属性名。

```
ds.PropertiesToLoad.Add("cn");  
ds.PropertiesToLoad.Add("sn");
```

返回的结果集（见“提取搜索结果”一节）只包含用户在PropertiesToLoad属性中指定的对象属性。

SearchScope

默认情况下，DirectorySearcher对象会搜索SearchRoot对象下面全部的树。可以使用SearchScope属性定义搜索深度。它可以是以下三种值的其中之一：

SearchScope.Base 只搜索SearchRoot属性中定义的基本对象

SearchScope.OneLevel 只搜索SearchRoot属性中定义的基本对象的下一级子对象

SearchScope.Subtree 搜索SearchRoot属性中定义的基本对象下面的全部树

第2步：检索搜索结果

一旦决定了DirectorySearcher参数，可以开始搜索并且获取结果。使用FindOne()或者FindAll()从搜索中获取结果。

FindOne()方法

FindOne()方法只返回SearchResult类对象中定义的搜索的一个结果。如果搜索结果中发现更多项目，只返回第一个项目。

SearchResult类类似于DirectoryEntry对象，但是仅仅引用数据库对象而不能访问对象。SearchResult对象包括Path和Properties属性，可以取得关于数据库对象的信息，比如它在数据

库中的位置，它的属性名和值。

Path属性可以生成**DirectoryEntry**对象、直接访问对象、修改或者删除对象或者对象的任何属性。

FindAll()方法

FindAll()方法返回在**SearchResultCollection**类对象中搜索的全部结果。**SearchResultCollection**对象是**SearchResult**对象的集合，每个对象对应搜索的每个结果。

可以使用**foreach**声明通过**SearchResultCollection**对象不断重复，获取每个**SearchResult**对象。

第3步：提取搜索结果

一旦从搜索中获得**SearchResult**或者**SearchResultCollection**对象，可以使用**Properties**属性提取每个数据库对象信息：

```
SearchResult sr = ds.FindOne();
string user = sr.Properties["cn"][0];
```

这个代码段从搜索中查找结果并且提取普通名（**cn**）属性值。

还可以使用**Path**属性生成**DirectoryEntry**对象，用于进一步的操作：

```
SearchResult sr = ds.FindOne();
string newpath = sr.Path;
DirectoryEntry de = new DirectoryEntry(newpath);
```

这里，生成**DirectoryEntry**对象之后，它可以提取、修改或者删除对象的任何属性，或者对象本身，详见前面的“修改目录数据”一节。

执行搜索

程序清单15.10的**SimpleSearch.cs**程序演示如何对数据库中所有项目进行简单网络目录服务搜索。

程序清单15.10 SimpleSearch.cs程序

```
using System;
using System.DirectoryServices;

class SimpleSearch
{
    public static void Main()
    {
        DirectoryEntry root = new DirectoryEntry(
            "LDAP://192.168.1.100/DC=ispnet1,DC=net",
            "cn=Administrator, dc=ispnet1, dc=net", "password",
            AuthenticationTypes.ServerBind);
```

```

DirectorySearcher searcher = new DirectorySearcher(root);
searcher.Filter = "(&(objectClass=person)(sn=Blum))";
searcher.PropertiesToLoad.Add("cn");
searcher.PropertiesToLoad.Add("telephoneNumber");
SearchResultCollection results = searcher.FindAll();

foreach(SearchResult result in results)
{
    string searchpath = result.Path;
    Console.WriteLine("path: {0}", searchpath);
    ResultPropertyCollection rpc = result.Properties;
    foreach(string property in rpc.PropertyNames)
    {
        foreach(object value in rpc[property])
            Console.WriteLine("  property = {0}  value = {1}",
                               property, value);
    }
}
}
}

```

SimpleSearch程序使用一个过滤器，指定只获得使用person objectClass和以Blum作为姓的对象。结果保存在SearchResultCollection并且使用foreach声明来提取。

对于每一个结果来说，使用ResultPropertyCollection对象和PropertyNames属性提取属性。输出结果应该包含目标对象。例如，在下面的输出结果中，可以看到搜索返回的那些属性是PropertiesToLoad Add()方法指定的属性：

C:\>SimpleSearch

```

path: LDAP://192.168.1.100/cn=kblum, ou=sales, dc=ispnet1, dc=net
  property= cn  value = Katie Blum
  property= telephonenumber  value = (111)222-3333
  property= telephonenumber  value = (444)555-6666
  property= adspath  value = LDAP://192.168.1.100/cn=kblum, ou=sales, ➡
dc=ispnet1, dc=net
path: LDAP://192.168.1.100/cn=jblum, ou=accounting, dc=ispnet1, dc=net
  property= cn  value = Jessica Blum
  property= adspath  value = LDAP://192.168.1.100/cn=jblum, ou=accounting, ➡
dc=ispnet1, dc=net

```

C:\>

说明：必须注意对象包含的一个特殊属性adspath。这是一个内部属性，用于网络目录服务，不能被用户修改或者删除。

高级搜索特性

DirectorySearcher类也提供一些高级功能，可以帮助进行搜索和返回结果。

使用服务器推举

如果在一个AD树环境中运行AD程序，会访问到多个域。如果在DirectorySearcher对象中指定服务器地址，实际的数据对象可能包含在不同服务器。在这种情况下，试图绑定的服务器会发送一个服务器选择提示，通知适当的服务器，用来绑定数据库对象。

在默认情况下，DirectorySearcher对象收到服务器推举时会产生一个错误提示。可以使用ReferralChasing属性修改，定义DirectorySearcher对象从服务器收到服务器推举时的反应。可以指定四种选项，每一项由ReferralChasingOption列举：

None 指定程序不理睬服务器推举，跳出错误提示。

Subordinate 指定程序依从服务器推举而选择在目录树中包含下级对象的服务器。

External 指定程序依从服务器推举而选择在本地网络以外的服务器。

All 指定程序依从网络目录服务服务器的全部服务器推举。

限制搜索

在大型网络目录服务数据库中搜索，可能必须限制返回数据的数量，防止应用程序在搜索过程中出现“锁定”。在DirectorySearcher类中使用下面的属性指定搜索限制：

ClientTimeout 允许在超时和报错之前设置一段时间（以秒为单位），等待服务器应答。

PageSize 定义一次搜索返回的最大对象数量。默认值为0，定义对象数量为无限。如果PageSize属性设为大于0的值，在返回结果中显示很多对象，下一次的搜索将从前一次搜索结束的地方开始搜索。这就允许应用程序获得更多对象，并且继续搜索，直到找到正确对象为止。

PropertyNamesOnly 一个boolean值。如果这个值设为true，DirectorySearcher对象只返回对象属性的名称而没有任何数据值。这个值默认为false。

ServerPageTimeLimit 允许程序指定服务器用多长时间（以秒为单位）为一页（详见PageSize属性定义）寻找搜索结果。如果搜索超过时间限制，服务器将返回在限制的时间内获得的结果，以及从何处恢复搜索的信息。默认值为-1，定义为没有页时间限制。

ServerTimeLimit 定义服务器处理搜索要求使用的全部时间（以秒为单位）。如果超过时间限制，服务器将返回在限制的时间内获得的结果，以及从何处恢复搜索的信息。默认值为-1，表示没有服务器时间限制。

SizeLimit 设置服务器对于一次搜索返回对象的总数。如果结果过多，服务器应忽略它们。不能从SizeLimit点继续搜索。每个网络目录服务服务器都有它自己的SizeLimit值。如果SizeLimit属性设置超过服务器默认值，将使用服务器值。

小结

本章讨论如何在System.DirectoryServices命名空间中使用类，从C#应用程序访问AD和其他网络目录服务服务器。

轻型目录访问协议（LDAP）定义在分层网络数据库中的网络资源。数据库中的一个对象代表一个网络资源，并且每个对象包含属性对应资源定义值。网络上的设备可以访问数据库并且获取关于网络资源的信息。

Microsoft公司的AD，用在Windows 2000服务器网络中，使用LDAP模式存储域中的网络资源信息。来自多个域中的信息可以合并形成树，而多个树可以合并形成森林。网络上的网络设备访问AD数据库获取信息，包括用户、密码、打印机、服务器和其他网络资源。AD使用AD服务接口（ADSI）允许客户机工作站访问AD服务器。

.NET库包括DirectoryEntry和DirectorySearcher类，从C#程序实现ADSI功能。DirectoryEntry类给网络目录对象绑定一个变量。一旦变量绑定到对象，可以使用该变量获取对象和属性信息（在.NET中调用属性）。

DirectorySearcher类用于构成复杂搜索，在网络目录中帮助取得网络资源的信息。Filter属性减少搜索返回的对象数量，PropertiesToLoad属性限制返回的对象属性数量。

第16章 远程技术

- 移动数据并再次访问
- 使用串行化类
- 串行化产生的问题
- 什么是远程技术
- 使用远程技术
- 创建远程技术代理类
- 创建远程技术服务器程序
- 创建远程技术客户机程序

让分布于不同网络的多个计算机上的应用程序互相通信，是.NET Framework众多令人兴奋的特性之一。NET Framework支持远程技术，该功能类似于第14章“HTTP”中介绍的网络服务，可以让网络中不同计算机上的应用程序共享类的数据和方法。借助于远程技术，计算机不必运行Microsoft IIS服务器，多个C#程序就可以共享类方法。

远程技术依赖于数据串行化，后者在第7章“使用C#套接字助手类”中已经做了讲解。串行化就是将任意数据类型的类数据元素以流的形式传输并重组。对于远程技术，传输数据的流就是NetworkStream对象。

在本章的开始，介绍了串行化以及如何利用串行化而不使用远程技术，在网络设备上的网络应用程序间发送数据。接下来将介绍.NET远程技术的要素，以及在为C#网络程序提供分布式计算环境时这些要素之间的互相影响。最后给出一个在网络程序中实现远程技术的例子。

移动数据并再次访问

在第7章中介绍了如何将复杂的C#类转换为二进制序列，以便通过网络发送到远程系统。NET库包含一些类可以完成这样的工作。

本节将介绍两个类，用它们对数据进行串行化后通过网络传送，它们是BinaryFormatter类和SoapFormatter类。用这些类可以很容易地将类转换为串行数据流，通过网络发送到远程系统，最后再变换成原始数据。

使用串行化类

将类串行化并通过网络发送需要以下三个步骤：

1. 创建串行化类的库对象。
2. 编写发送程序，创建串行化类的实例，并且以流的形式发送。

3. 编写接收程序从流中读取数据，重新得到原来的串行化类的数据。

创建串行化类

每个通过网络传送数据的数据类必须在源代码文件中标上[Serializable]属性。这表明在默认情况下，类中的所有的数据元素将被串行化传送。程序清单16.1说明如何创建第7章中介绍的Employee类的串行化形式。

程序清单16.1 SerialEmployee.cs程序

```
using System;

[Serializable]
public class SerialEmployee
{
    public int EmployeeID
    public string LastName;
    public string FirstName;
    public int YearsService;
    public double Salary;

    public SerialEmployee()
    {
        EmployeeID = 0;
        LastName = null;
        FirstName = null;
        YearsService = 0;
        Salary = 0.0;
    }
}
```

SerialEmployee.cs文件包含SerialEmployee类的类定义。类中包含的数据元素用于跟踪基本雇员信息，同时还有一个简单的默认类构造器。[Serializable]属性放在类定义之前，表示该类可以用格式类之一转换为串行字节流。

要想使用这个类传送雇员数据，首先要创建一个可以编译成应用程序的库文件：

```
csc /t:library SerialEmployee.cs
```

该命令行的结果是SerialEmployee.dll文件。此文件必须作为引用文件包含在所有使用SerialEmployee类的程序中。

警告：在串行化中，非常重要的是，要记住所有使用串行数据类的应用必须使用同一数据类库。当.NET创建串行数据流时，数据流中含有定义数据所用的类名。如果读取数据的时候类名不匹配，程序就不能将流转换回原来的样子。

创建发送程序

在创建数据类之后，可以使用新的类建立应用并且将新的数据串行化转化为流。如前

所述, **BinaryFormatter**类和**SoapFormatter**类用来串行化数据。

BinaryFormatter将数据串行化为二进制流, 非常类似于程序清单7.11中**Employee.cs**程序中的**GetBytes()**方法。除当前数据之外, 一些附加信息比如类名和版本号也会加入到串行化数据中。

另一种方法, 是使用**SoapFormatter**类以XML格式传递数据, 类似于第14章网络服务程序采用的技术。使用XML的好处是任何系统或者应用程序都可以识别XML格式。

首先, 要发送数据, 必须创建**Stream**类的实例, 它可以是任意类型的流, 包括**FileStream**、**MemoryStream**或**NetworkStream**。下一步, 创建相应的串行化类的实例, 然后用**Serialize()**方法将数据发送给**Stream**对象:

```
Stream str = new FileStream(
    "testfile.bin", FileMode.Create, FileAccess.ReadWrite);
IFormatter formatter = new BinaryFormatter();
formatter.Serialize(str, data);
```

IFormatter类创建所需串行化类 (**BinaryFormatter**或者**SoapFormatter**) 的实例, 并且数据是使用格式程序的**Serialize()**方法串行化的。

要想看到在串行化数据类中什么样的信息通过网络传输, 可以将其输出保存到**FileStream**对象中, 以方便查看。程序清单16.2列出**SoapTest.cs**程序, 使用**SoapFormatter**在一个文件中串行化两个**SerialEmployee**类的实例。

程序清单16.2 SoapTest.cs程序

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Soap;

class SoapTest
{
    public static void Main()
    {
        SerialEmployee emp1 = new SerialEmployee();
        SerialEmployee emp2 = new SerialEmployee();

        emp1.EmployeeID = 1;
        emp1.LastName = "Blum";
        emp1.FirstName = "Katie Jane";
        emp1.YearsService = 12;
        emp1.Salary = 35000.50;

        emp2.EmployeeID = 2;
        emp2.LastName = "Blum";
        emp2.FirstName = "Jessica";
        emp2.YearsService = 9;
```

```

emp2.Salary = 23700.30;

Stream str = new FileStream("soaptest.xml", FileMode.Create,
    FileAccess.ReadWrite);
IFormatter formatter = new SoapFormatter();

formatter.Serialize(str, emp1);
formatter.Serialize(str, emp2);
str.Close();
    }
}

```

SoapFormatter类出现在**System.Runtime.Serialization.Formatters.Soap**命名空间中，所以必须对它进行声明。如果要使用**BinaryFormatter**类来代替，它就要出现在**System.Runtime.Serialization.Formatters.Binary**命名空间中。**IFormatter**接口是在**System.Runtime.Serialization**命名空间中的，所以也要包括。在创建**SerialEmployee**类的两个实例之后，程序创建一个**FileStream**对象用来保存输出结果，然后使用**Serialize()**方法保存这两个流。

要想编译这个程序，不要忘记将**SerialEmployee.dll**文件作为引用文件包括在其中：

```
csc /r:SerialEmployee.dll SoapTest.cs
```

运行完**SoapTest.exe**程序之后，就可以查看创建的**soaptest.xml**文件了（见程序清单16.3）。

程序清单16.3 soaptest.xml文件

```

<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ➡
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:SOAP-ENC= ➡
'http://schemas.xmlsoap.org/soap/encoding/' xmlns:SOAP-ENV= ➡
'http://schemas.xmlsoap.org/soap/envelope/' xmlns:clr= ➡
'http://schemas.microsoft.com/soap/encoding clr/1.0' SOAP-ENV:encodingStyle= ➡
'http://schemas.xmlsoap.org/soap/encoding/'>
<SOAP-ENV:Body>
<a1:SerialEmployee id="ref-1" xmlns:a1= ➡
'http://schemas.microsoft.com/clr/assem/SerialEmployee%2C%20Version%3D0. ➡
0.0.0%2C%20Culture%3Dneutral%2C%20PublicKeyToken%3Dnull">
<EmployeeID>1</EmployeeID>
<LastName id="ref-3">Blum</LastName>
<FirstName id="ref-4">Katie Jane</FirstName>
<YearsService>12</YearsService>
<Salary>35000.5</Salary>
</a1:SerialEmployee>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ➡
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:SOAP-ENC= ➡
'http://schemas.xmlsoap.org/soap/encoding/' xmlns:SOAP-ENV= ➡

```

```

"http://schemas.xmlsoap.org/soap/envelope/" xmlns:clr= ➡
"http://schemas.microsoft.com/soap/encoding/clr/1.0" SOAP-ENV:encodingStyle= ➡
"http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
<a1:SerialEmployee id="ref-1" xmlns:a1= ➡
"http://schemas.microsoft.com/clr/assem/SerialEmployee%2C%20Version%3D0. ➡
0.0.0%2C%20Culture%3Dneutral%2C%20PublicKeyToken%3Dnull">
<EmployeeID>2</EmployeeID>
<LastName id="ref-3">Blum</LastName>
<FirstName id="ref-4">Jessica</FirstName>
<YearsService>9</YearsService>
<Salary>23700.3</Salary>
</a1:SerialEmployee>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>

```

查看soaptest.xml文件，可以看出SOAP如何在串行化类中定义每个数据元素。下面的命令行给出XML数据很重要的一个特性：

```

<a1:SerialEmployee id="ref-1" xmlns:a1= ➡
"http://schemas.microsoft.com/clr/assem/SerialEmployee%2C%20Version%3D0. ➡
0.0.0.%2C%20Culture%3Dneutral%2C%20PublicKeyToken%3Dnull">

```

这里，当前串行化数据类的类名用在XML定义数据中。这非常重要，如果接受程序使用不同的类名定义同一个数据类，这与从流中读取XML数据将会不匹配。类必须匹配，否则读取将失败。

说明：如程序清单16.3所示，尽管SoapFormatter加入了与其他系统类信息的沟通能力，仍然会在发送中产生大量数据。

现在读者已经了解如何串行化数据，可以编写网络应用程序让数据串行化并且发送给运行在远程设备上的程序。程序清单16.4列出的BinaryDataSender.cs程序，使用BinaryFormatter类发送SerialEmployee数据。

程序清单16.4 BinaryDataSender.cs程序

```

using System;
using System.Net;
using System.Net.Sockets;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;

class BinaryDataSender
{
    public static void Main()
    {

```

```
        SerialEmployee emp1 = new SerialEmployee();
        SerialEmployee emp2 = new SerialEmployee();

        emp1.EmployeeID = 1;
        emp1.LastName = "Blum";
        emp1.FirstName = "Katie Jane";
        emp1.YearsService = 12;
        emp1.Salary = 35000.50;

        emp2.EmployeeID = 2;
        emp2.LastName = "Blum";
        emp2.FirstName = "Jessica";
        emp2.YearsService = 9;
        emp2.Salary = 23700.30;

        TcpClient client = new TcpClient("127.0.0.1", 9050);
        IFormatter formatter = new BinaryFormatter();
        NetworkStream strm = client.GetStream();

        formatter.Serialize(strm, emp1);
        formatter.Serialize(strm, emp2);
        strm.Close();
        client.Close();
    }
}
```

BinaryDataSender程序使用**BinaryFormatter**串行化**SerialEmployee**类的两个实例，并且向**TcpClient**对象指定的远程设备发送。发送完两个实例后，**Stream**和**TcpClient**两个对象都关闭了。

说明：由于**BinaryFormatter**和**SoapFormatter**类都需要**Stream**对象发送串行化数据，所以必须使用**TCP Socket**对象或者**TcpClient**对象来发送数据，**UDP**不能直接与串行化类一起使用。

创建接收程序

通过网络移动数据的第三步也是最后一步，是编写程序，从流中读取数据并且还原为原来的类数据元素。这一步要再次用到**BinaryFormatter**或者**SoapFormatter**类。显而易见，这一步使用的类必须与在发送程序中串行化数据使用的类相同。

当格式类还原数据的时候，数据元素被提取到一个普通的**Object**对象中。必须选用适当的类来提取数据元素：

```
IFormatter formatter = new BinaryFormatter();
SerialEmployee emp1 = (SerialEmployee)formatter.Deserialize(str);
```

当从流中接收数据的时候，数据被重新组合为原来的数据类。应该注意确保重新构造数据类的的数据量是正确的。

程序清单16.5列出BinaryDataRcvr.cs程序，从BinaryDataSender程序中接收串行化数据并且重新创建原始数据类。

程序清单16.5 BinaryDataRcvr.cs程序

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;

class BinaryDataRcvr
{
    public static void Main()
    {
        TcpListener server = new TcpListener(9050);
        server.Start();
        TcpClient client = server.AcceptTcpClient();
        NetworkStream strm = client.GetStream();
        IFormatter formatter = new BinaryFormatter();

        SerialEmployee emp1 = (SerialEmployee)formatter.Deserialize(strm);
        Console.WriteLine("emp1.EmployeeID = {0}", emp1.EmployeeID);
        Console.WriteLine("emp1.LastName = {0}", emp1.LastName);
        Console.WriteLine("emp1.FirstName = {0}", emp1.FirstName);
        Console.WriteLine("emp1.YearsService = {0}", emp1.YearsService);
        Console.WriteLine("emp1.Salary = {0}\n", emp1.Salary);

        SerialEmployee emp2 = (SerialEmployee)formatter.Deserialize(strm);
        Console.WriteLine("emp2.EmployeeID = {0}", emp2.EmployeeID);
        Console.WriteLine("emp2.LastName = {0}", emp2.LastName);
        Console.WriteLine("emp2.FirstName = {0}", emp2.FirstName);
        Console.WriteLine("emp2.YearsService = {0}", emp2.YearsService);
        Console.WriteLine("emp2.Salary = {0}", emp2.Salary);
        strm.Close();
        server.Stop();
    }
}
```

BinaryDataRcvr程序创建TcpListener对象，绑定在指定端口并且等待来自BinaryDataSender程序的连接尝试。当连接建立起来，Deserialize()方法将接收到的数据恢复成原先的数据类。

说明：这个例子使用的是简单的数据类型，稍做改动，读者可以很容易地加入复杂的数据类型，例如雇员开始上班的日期。向SerialEmployee.cs文件中加入新的数据元素，重新创建DLL库文件，并且重新建立发送和接收程序。

串行化产生的问题

尽管举出了很多例子，表明串行化传输复杂的数据类是很简单的，然而在真正的网络环境中并不那么容易。读者可能已经注意到使用BinaryDataSender和BinaryDataRcvr程序是基于这样一种重要的假设：这两种程序都期望所有的数据由BinaryFormatter的接收程序接收，然后将流恢复成原来的类数据。在真实网络环境中，这个条件不一定能实现。

在执行Deserialize()方法之前如果没有从流中接收到全部数据，就会产生问题。当Deserialize()方法没有足够的数据来完成重新组合的时候，会产生一个错误提示，并且不能正确创建数据类。解决这个问题方法是使用混合技术，本书的第7章曾经描述过使用数据拼接的方法，合并当前的串行化类。如果发送者在发送实际对象之前先发送每个串行化数据对象的大小，接收方就可以在试图重组数据前先确定接收的数据有多少字节。

最简单的办法是将串行化数据从流中放入MemoryStream对象中。MemoryStream对象将全部数据存放在缓存中，它允许用户很容易地确定串行化数据的总量。当数据流的总量确定以后，数据大小和缓存中的串行化数据就会先后以NetworkStream的形式发送到远程设备。

程序清单16.6列出BetterDataSender.cs程序，就是使用上述方式向远程设备发送SerialEmployee数据类的两个实例。

程序清单16.6 BetterDataSender.cs程序

```
using System;
using System.IO;
using System.Net;
using System.Net.Sockets;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Soap;

class BetterDataSender
{
    public void SendData (NetworkStream strm, SerialEmployee emp)
    {
        IFormatter formatter = new SoapFormatter();
        MemoryStream memstrm = new MemoryStream();

        formatter.Serialize(memstrm, emp);
        byte[] data = memstrm.GetBuffer();
        int memsize = (int)memstrm.Length;
        byte[] size = BitConverter.GetBytes(memsize);
        strm.Write(size, 0, 4);
        strm.Write(data, 0, memsize);
        strm.Flush();
        memstrm.Close();
    }

    public BetterDataSender()
```



```

{
    SerialEmployee emp1 = new SerialEmployee();
    SerialEmployee emp2 = new SerialEmployee();

    emp1.EmployeeID = 1;
    emp1.LastName = "Blum";
    emp1.FirstName = "Katie Jane";
    emp1.YearsService = 12;
    emp1.Salary = 35000.50;

    emp2.EmployeeID = 2;
    emp2.LastName = "Blum";
    emp2.FirstName = "Jessica";
    emp2.YearsService = 9;
    emp2.Salary = 23700.30;

    TcpClient client = new TcpClient("127.0.0.1", 9050);
    NetworkStream strm = client.GetStream();

    SendData(strm, emp1);
    SendData(strm, emp2);
    strm.Close();
    client.Close();
}

public static void Main()
{
    BetterDataSender bds = new BetterDataSender();
}
}

```

BetterDataSender程序使用**SendData()**方法创建**MemoryStream**对象，向远程设备发送串行化对象。从**MemoryStream**对象中可以确定数据的大小并且发送给远程接收者。然后缓存中的串行化数据会由**MemoryStream**对象发送给远程接收者。

现在看一下程序清单16.7列出的**BetterDataRcvr.cs**程序，展示如何根据数据大小重组接收到的串行化数据。

程序清单16.7 BetterDataRcvr.cs程序

```

using System;
using System.IO;
using System.Net;
using System.Net.Sockets;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Soap;

class BetterDataRcvr

```

```

{
    private SerialEmployee RecvData (NetworkStream strm)
    {
        MemoryStream memstrm = new MemoryStream();
        byte[] data = new byte[4];
        int recv = strm.Read(data, 0, 4);
        int size = BitConverter.ToInt32(data, 0);
        int offset = 0;
        while(size > 0)
        {
            data = new byte[1024];
            recv = strm.Read(data, 0, size);
            memstrm.Write(data, offset, recv);
            offset += recv;
            size -= recv;
        }
        IFormatter formatter = new SoapFormatter();
        memstrm.Position = 0;
        SerialEmployee emp = (SerialEmployee)formatter.Deserialize(memstrm);
        memstrm.Close();
        return emp;
    }

    public BetterDataRcvr()
    {
        TcpListener server = new TcpListener(9050);
        server.Start();
        TcpClient client = server.AcceptTcpClient();
        NetworkStream strm = client.GetStream();

        SerialEmployee emp1 = RecvData(strm);
        Console.WriteLine("emp1.EmployeeID = {0}", emp1.EmployeeID);
        Console.WriteLine("emp1.LastName = {0}", emp1.LastName);
        Console.WriteLine("emp1.FirstName = {0}", emp1.FirstName);
        Console.WriteLine("emp1.YearsService = {0}", emp1.YearsService);
        Console.WriteLine("emp1.Salary = {0}\n", emp1.Salary);

        SerialEmployee emp2 = RecvData(strm);
        Console.WriteLine("emp2.EmployeeID = {0}", emp2.EmployeeID);
        Console.WriteLine("emp2.LastName = {0}", emp2.LastName);
        Console.WriteLine("emp2.FirstName = {0}", emp2.FirstName);
        Console.WriteLine("emp2.YearsService = {0}", emp2.YearsService);
        Console.WriteLine("emp2.Salary = {0}", emp2.Salary);
        strm.Close();
        server.Stop();
    }
}

```



```
public static void Main()
{
    BetterDataRcvr bdr = new BetterDataRcvr();
}
}
```

BetterDataRcvr程序使用**RecvData()**方法从发送者那里获取输入数据。首先收到一个值，这个值有4字节长。这个值表示传输的串行化数据部分有多大。然后**RecvData()**方法开始循环，直到所有的字节都从**NetworkStream**中收到以后才停止。数据到达以后，就加入到**MemoryStream**对象中。当全部数据收到以后，**MemoryStream**对象创建数据类。

警告：有一点非常重要，**MemoryStream**对象在用于**Deserialize()**方法之前必须复位，指向流的开端。

要测试**BetterDataSender**和**BetterDataRcvr**程序，必须使用相同的**SerialEmployee.dll**库文件编译这两个程序。

说明：当向远程主机传输数据的时候，**BinaryFormatter**和**SoapFormatter**可以适应各种网络字节传输次序。

远程技术总览

.NET可以向网络中的设备发送类数据，除此之外，还提供一种技术通过网络发送全部的类方法信息。这跟第14章中介绍的网络服务类似，远程技术允许在网络上的主机之间共享C#类方法。跟网络服务不同的是，远程技术不需要任何其他服务器软件来处理收到的请求。使用.NET 远程技术可以很容易地将分布式计算功能融入网络应用中。

远程技术的目的是利用网络上其他计算机的资源。远程技术使程序员可以编写应用程序，向网络上任何人提供应用服务。建立.NET 远程技术应用程序有5个要素：

- 远程类提供C#方法，供网络中的客户机应用程序使用。
- 远程技术服务器控制远程类，允许网络用户连接和处理方法调用。
- 信道从网络客户机向远程技术服务器传递方法调用，并且将结果返回到网络客户机。
- 代理类运行在客户机，接受来自客户机的方法调用并且通过信道向远程类传递。
- 客户机应用程序向代理类传递方法调用，并且解释通过代理类传来的远程类的响应。

图16.1展示各个.NET 远程技术要素的关联。

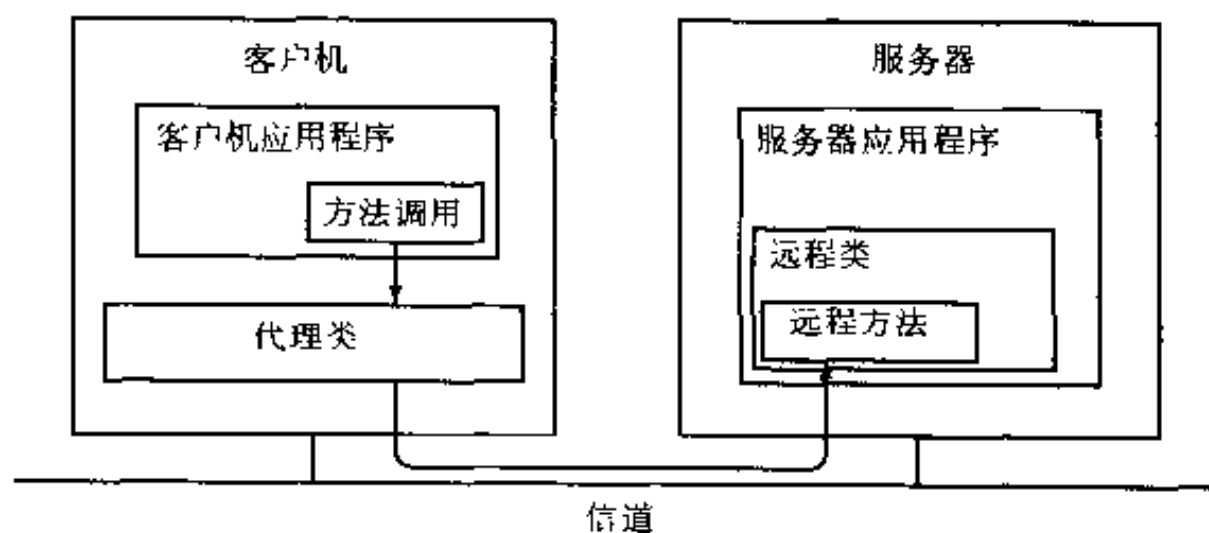


图16.1 .NET 远程技术的结构

远程类

.NET远程技术中使用远程类提供数据元素和方法，这些数据和方法则由远程技术服务器向客户机发布。迄今为止，所有的程序都包含在单一的应用域（application domain）中。应用域定义为一个区域，在这个区域中所有的应用程序共享相同的存储空间。任何脱离应用域类即被认为是远程类，如图16.2所示。

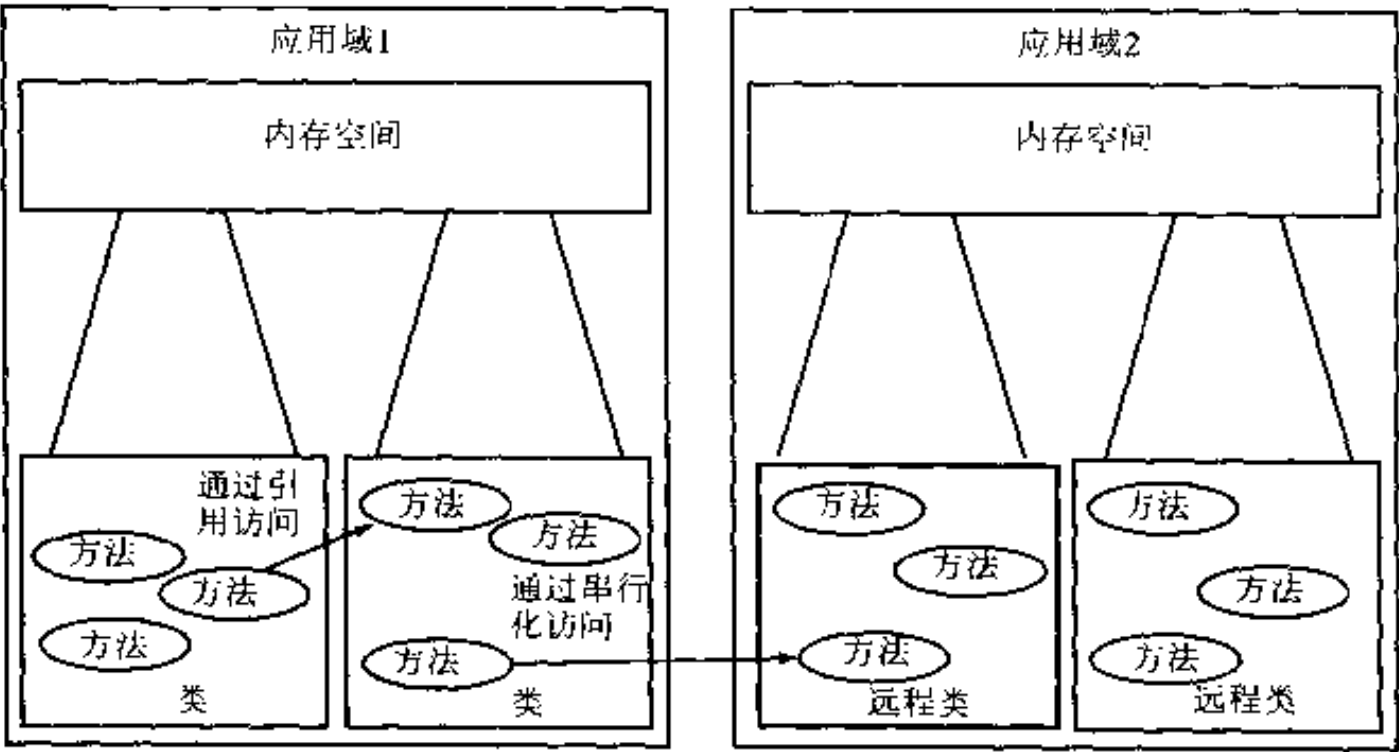


图16.2 客户机应用域

注意，远程类不一定在远程设备上。一个工作站可以包含多个应用域，域之间只有安全隔离而没有物理隔离，操作系统的安全规则为应用域提供了安全隔离。处于应用域以外的类不能直接访问（因为没有共享的存储区域），而必须经过串行化并通过信道传输给其他客户机应用程序。

远程技术服务器

远程技术服务器是一个C#应用程序，作为.NET远程技术系统和远程类之间的接口。它在计算机上的应用域中注册远程类并且监听指定信道发来的请求。远程技术服务器有两种技术处理远程类发来的请求：

SingleCall对象模式 SingleCall模式为每个客户机请求创建一个远程类的实例。客户机连接终止，该实例也被删除。在两个客户机请求之间并不共享状态信息。当需要严格区分类的实例的信息时，应该使用SingleCall对象模式。

Singleton对象模式 Singleton模式用一个远程类的实例控制所有发来的请求。Singleton模式允许在服务器端的众多客户机连接之间建立服务器程序，保存一些数据。Singleton模式下，远程类的实例有一定的寿命，当寿命耗尽时，远程类的实例就被删除。这时如果收到一个新的客户机请求，就创建一个新的远程类实例。当出于性能方面的考虑需要共享类实例时，应该使用Singleton对象模式。

信道

.NET远程技术系统非常依赖信道向客户机传递远程类信息。当客户机访问远程类的时候，关于类方法的信息、方法参数和方法返回的值必须在设备之间传递。这可以经由TCP通信传递，在指定端口直接使用TCP传递串行化类信息。或者使用HTTP信道，用它提供的标准通信技术在客户机和远程类之间传递数据。

警告： 当在一个设备上使用多个远程类的时候，必须记住每一端口每一信道只能进行一个通信业务。

代理类

客户机机器上的代理类代表远程技术服务器上可以访问的远程类方法。代理类提供给客户机程序一个接口，用来访问运行在远程技术服务器上的远程类。

代理类的作用是串行化数据并且将其发送给远程类服务器。另一方面它还要重组收到的数据，并且根据客户机程序的要求将数据放入合适的类数据元素中。

没有代理类，客户机应用程序就不知道类数据元素如何格式化，也不清楚数据元素使用何种数据类型。

客户机程序

客户机程序要使用远程类中的类方法，必须知道什么方法是有效的以及如何访问。客户机程序必须与代理类一起编译以确定远程类方法的格式。所有对远程类方法的引用都传递给代理类来处理。

客户机程序连接到远程类方法时有两个选项，一个是注册一个信道用于远程类本身，操控所有与远程类的通信，用代理类处理远程类方法的信息。另一个是允许代理类注册信道并且操控所有的通信。

两种技术都有优缺点。如果允许代理类操控所有的信道请求，客户机程序的复杂度可以降低，但是它必须绑定在指定信道上。另一方面，如果让客户机程序操控信道，当远程类转移到别的服务器上时，可以很方便地进行修改。

使用远程技术

建立使用远程技术的应用程序必须编写以下3个独立程序：

- 远程类，允许客户机访问并且使用其中的方法。
- 服务器程序，通报并且管理计算机上的服务器类。
- 客户机程序，使用服务器类方法。

本节描述如何实现各个部分，从而用远程技术创建C#网络程序。

创建远程类代理

每个远程应用程序都需要一套方法，可以由网络上的客户机访问。像串行化数据一样，

这些方法必须包含在特殊类中并且编译为代理库对象，用于服务器和客户机程序中。

判断一个类是不是远程类，要看它是否源自`MarshalByRefObject`类。客户机可以通过代理访问远程类方法。程序清单16.8列出的`MathClass.cs`程序，就定义了一个类，用做远程类。

程序清单16.8 MathClass.cs程序

```
using System;

public class MathClass MarshalByRefObject
{
    public int Add(int a, int b)
    {
        int c = a + b;
        return c;
    }

    public int Subtract(int a, int b)
    {
        int c = a - b;
        return c;
    }

    public int Multiply(int a, int b)
    {
        int c = a * b;
        return c;
    }

    public int Divide(int a, int b)
    {
        int c;
        if (b != 0)
            c = a / b;
        else
            c = 0;
        return c;
    }
}
```

`MathClass`类源自`MarshalByRefObject`类，这意味着它可以在服务器程序中作为一个远程类。它定义了4个常用的数学函数，可以被远程网络客户机调用。每个函数从客户机收到两个参数，并且将计算结果返回给客户机。

与串行化技术相似，必须从远程类程序创建一个DLL文件。可以用`csc`编译器以及`/t:library`选项完成：

```
csc /t:library MathClass.cs
```

MathClass.dll文件包含全部必需的信息，以便服务器和客户机程序访问远程类中的数据元素和方法。

警告：与串行化技术相似，必须保证服务器和客户机程序定义远程类使用的代理类库文件是相同的。如果这些程序用的代理类库不相同，将不能串行化，并且远程函数会失败。另外，代理类库是根据类名分类的，而不是文件名，要确保类库引用类名都正确。

创建服务器程序

服务器程序控制网络上的远程类，接收来自客户机的请求并且向远程类发送。服务器程序必须注册一个信道，支持与远程类的通信，然后在.NET远程技术服务器中将远程类注册为服务。

为远程类建立服务器程序有以下3步：

1. 创建信道，访问远程类。
2. 注册信道及远程技术信道服务。
3. 将作为服务的远程类与远程技术服务共同注册。

创建信道

首先必须定义信道。为实现网络访问，.NET远程技术可以用两个类定义两种类型的信道，这两个类分别是**TcpChannel**和**HttpChannel**。

TcpChannel **TcpChannel**类，属于**System.Runtime.Remoting.Channels.Tcp**命名空间，用二进制格式的TCP分组接收和发送远程类。**TcpChannel**类的实例可以使用TCP端口监听来自远程类的请求：

```
TcpChannel chan = new TcpChannel(9050);
```

默认情况下，**TcpChannel**类使用**BinaryFormatter**类串行化所有在信道上发送和接收的数据。

HttpChannel **HttpChannel**类，属于**System.Runtime.Remoting.Channels.Http**命名空间，用SOAP格式在HTTP会话中传输远程类。**HttpChannel**类的实例也使用TCP端口号接收来自远程类的HTTP请求：

```
HttpChannel chan = new HttpChannel(9050);
```

默认情况下，**HttpChannel**类使用**SoapFormatter**类串行化所有在信道上接收和发送的数据。

注册信道

创建**TcpChannel**或者**HttpChannel**对象以后，必须在接收网络信息之前在系统中注册。**ChannelServices**类的**RegisterChannel()**方法用于在系统中注册信道。**ChannelServices**类是在**System.Runtime.Remoting.Channels**命名空间中建立的：

```
ChannelServices.RegisterChannel(chan);
```

惟一的参数（**chan**）表示注册的信道，**TcpChannel**或者**HttpChannel**均可。

注册远程类服务

创建并且注册信道之后，服务器程序必须将远程类注册为系统服务。`System.Runtime.Remoting`命名空间包含远程技术`Configuration`类，为远程技术系统设置了一个新的服务。在注册远程技术服务的时候这个类使用指定的参数，表16.1列出必须指定的参数。

表16.1 远程技术Configuration参数

参数	说明
type	远程类对象的完整类型名称，以及定义类的文件名
objectUri	远程类对象的URI
mode	服务器如何操作远程类对象

`type`参数指定类名和定义远程类的DLL文件名。因为`MathClass`例子中使用的DLL文件名和类名相同，所以应该定义如下：

```
type="MathClass, MathClass"
```

`objectUri`参数定义标识服务器上远程类的惟一URI值。这个值不必与远程类名有任何关联，但是要避免具有多个远程类的服务器产生混淆。它通常与使用的类名稍微有点关联。

必须指定使用`SingleCall`还是`Singleton`对象模式来操控从远程类发来的请求。这由`mode`选项完成：

```
mode = "SingleCall"
```

在远程服务器上定义远程类需要确定远程技术`Configuration`参数，在此之后，必须在服务器程序中声明这些参数。这有两种方法，使用单独的配置文件（使用XML格式），或者通过服务器程序声明。

使用单独的配置文件

当使用配置文件的时候，该文件包含XML标记以识别远程技术`Configuration`对象需要的参数。程序清单16.9给出一个例子，是服务器上定义`MathClass`远程类的配置文件。

程序清单16.9 server.xml配置文件

```
<configuration>
  <system.runtime.Remoting>
    <application>
      <service>
        <wellknown
          type="MathClass, MathClass"
          objectUri="MyMathServer"
          mode="SingleCall"
        />
      </service>
    </application>
  </system.runtime.Remoting>
</configuration>
```



```
</service>
</application>
</system.runtime.Remoting>
</configuration>
```

创建配置文件之后，必须使用**Configure()**方法从文件中读取参数。下面的例子中假设**server.xml**文件与服务器程序位于同一目录下：

```
RemotingConfiguration.Configure("server.xml");
```

如果情况并非如此，则必须参照配置文件的完整路径名。

使用程序声明

这种方法不使用单独的配置文件，**RegisterWellKnownServiceType()**方法可以在应用程序中定义远程类的属性：

```
RemotingConfiguration.RegisterWellknownServiceType(
    Type.GetType("MathClass, MathClass"), "MyMathServer",
    WellKnownObjectMode.SingleCall);
```

在**RegisterWellKnownServiceType()**方法中定义了与配置文件相同的参数。

在程序声明中定义参数的缺点是在动态环境中不容易改动。如果应用程序是在一个不变的环境中，使用**RegisterWellKnownServiceType()**方法定义参数会比较理想。

完整的服务器程序

MathServer.cs程序（程序清单16.10）将所有部分合并在一起，演示如何创建远程服务器程序控制远程类对象。

程序清单16.10 MathServer.cs程序

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;

public class MathServer
{
    public static int Main()
    {
        HttpChannel chan = new HttpChannel(9050);
        ChannelServices.RegisterChannel(chan);
        RemotingConfiguration.RegisterWellKnownServiceType(
            Type.GetType("MathClass, MathClass"), "MyMathServer",
            WellKnownObjectMode.SingleCall);
        Console.WriteLine("Hit <enter> to exit...");
    }
}
```

```
        Console.ReadLine();  
        return 0;  
    }  
}
```

MathServer程序使用9050端口创建并且注册**HttpChannel**，接收来自**MathClass**远程类对象的请求。注册信息之后，服务器程序必须处于活动状态以便接收发来的请求。一旦程序退出，服务就会从远程技术系统中删除。为了保证服务器程序持续工作，可以使用简单的**Console.ReadLine()**方法等待键盘上的回车键事件。

服务器程序必须与**MathClass.dll**文件一起编译：

```
csc /r:MathClass.dll MathServer.cs
```

创建客户机程序

客户机程序连接到服务器程序并且使用远程类对象的方法。与服务器程序类似，客户机程序必须创建信道（与服务器使用的类型相同），并且必须在客户机计算机上的本地远程技术系统中注册信道：

```
HttpChannel chan = new HttpChannel();  
ChannelServices.RegisterChannel(chan);
```

因为客户机程序不需要监听指定端口，所以它不需要在**HttpChannel**构造器中指定端口号。

创建并且注册信道之后，可以用代理类创建远程类对象。代理类会模拟远程类的功能进行本地调用。当代理收到对远程类方法的调用，它通过建立通往服务器的信道传递这个调用。服务器接收这个调用并且传给远程类，处理调用并且返回信息，通过服务器程序返回给代理类。

有两种方法在客户机程序中创建代理对象：

- 在远程技术系统中注册远程类为客户机类型，并且为代理类创建新的远程类
- 使用**Activator**类为代理类注册远程类信息

注册远程类代理对象

类似于服务器注册远程类，客户机程序也能使用远程技术**Configuration**类注册需要的信息，使代理对象连接到远程类服务器。**System**命名空间中的**Activator**类用于实现这一功能。这个类创建代理对象，从应用程序到正确位置传递方法调用，也许是到本地程序，也许是远程类的正确的远程URI位置。**GetObject()**方法为代理对象定义远程类的位置。

GetObject()方法需要两个参数，远程类的类型和传递请求的URI：

```
Activator.GetObject(Type objecttype, string uri)
```

objecttype参数定义使用的远程类名，**uri**参数定义去哪里访问远程类。

完整的客户机程序

程序清单16.11列出的MathClient.cs程序演示如何创建能够访问服务器控制的远程类方法的客户机程序。

程序清单16.11 MathClient.cs程序

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;

public class MathClient
{
    public static int Main (string[] argv)
    {
        HttpChannel chan = new HttpChannel();
        ChannelServices.RegisterChannel(chan);
        MathClass obj = (MathClass)Activator.GetObject(
            typeof(MathClass), "http://127.0.0.1:9050/MyMathServer");
        if (obj == null)
            System.Console.WriteLine("Could not locate server");
        else
        {
            int a = Convert.ToInt32(argv[0]);
            int b = Convert.ToInt32(argv[1]);
            int c = obj.Add(a, b);
            Console.WriteLine("a + b = {0}", c);
            c = obj.Subtract(a, b);
            Console.WriteLine("a - b = {0}", c);
            c = obj.Multiply(a, b);
            Console.WriteLine("a * b = {0}", c);
            c = obj.Divide(a, b);
            Console.WriteLine("a / b = {0}", c);
        }
        return 0;
    }
}
```

代理对象创建以后，MathClient程序使用MathClass类中定义的各种类方法处理数据。代理类每次调用一个类方法，通过网络转发数据到服务器程序，处理数据并且将结果返回给代理类。

执行以下程序可以看到MathClient函数运算：

```
C:\>MathClient 1 5
a + b = 6
a - b = -4
a * b = 5
a / b = 0

C:\>MathClient 100 50
a + b = 150
a - b = 50
a * b = 5000
a / b = 2

C:\>
```

如果使用不同的计算机分别运行MathServer和MathClient程序（假设改变客户机程序中的URL），可以用WinDump或者Analyzer程序监视网络上的数据传输情况。因为HttpChannel信道用于通信，所有的传输都用XML格式，读取数据很方便。

用soapsuds创建代理类

本章给出的远程技术的示例都是基于这样一个重要假设——程序员同时掌握远程服务器和客户机应用程序，而在实际中往往并非如此。读者可能碰到这样的情况，要求使用远程技术服务器控制的方法编写客户机应用程序，却不清楚远程类是如何定义的。本节阐述如何在没有创建客户机的原始远程类文件的情况下，建立远程客户机应用程序。

查看远程类接口

当HttpChannel用于与远程服务器的通信时，可以使用普通的Web浏览器查看远程类定义。为远程技术服务器注册的URI说明了如何通过Web访问远程类。要查看类接口，必须在URI结尾添加?wsdl标记：

```
http://192.168.1.100:9050/MyMathServer?wsdl
```

URI产生标准SOAP网页，定义了程序清单16.8中的MathClass类。类中的每个方法在WSDL（Web Services Definition Language，Web服务描述语言）页面中定义，其中还有每个参数的定义和返回的值。图16.3说明WSDL的部分定义。

要为远程代理类创建库DLL文件，可以使用WSDL定义和soapsuds程序，而不必手工创建。

soapsuds程序

.NET Framework SDK包含有soapsuds程序，该程序从远程类中获取类信息（读者可能还记得wsdl程序中的这个功能，在本书第14章中的“网络服务”一节中曾经讨论过），获取的信息形成DLL代理文件，用于客户机应用程序与远程类之间的通信。

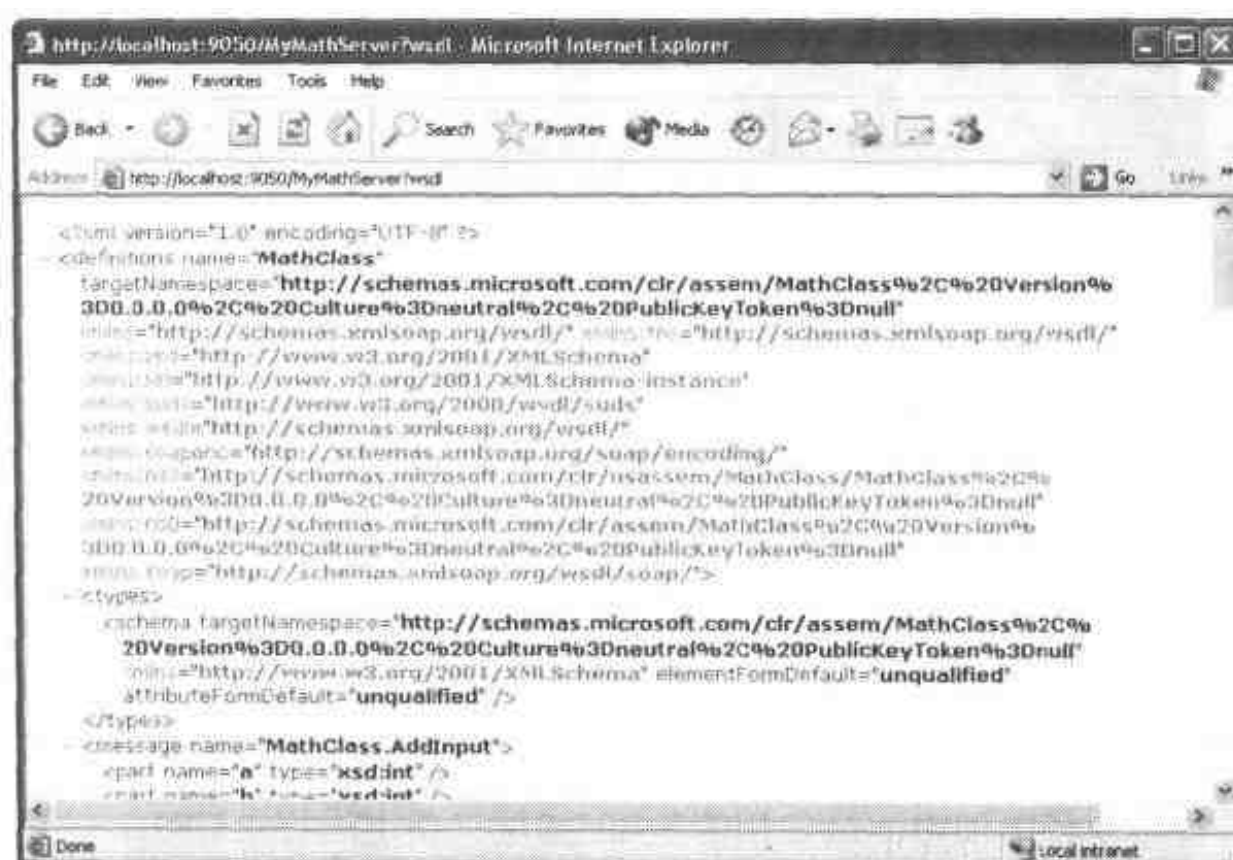


图16.3 MathClass WSDL定义

soapsuds程序提供了命令行选项，具有很多特性，如表16.2所示。

表16.2 soapsuds命令行选项

选项	详细说明
-d:domain	当连接到服务器时使用的域名
-gc	产生连接到服务器的C#代码
-hpn:name	用于连接到服务器的HTTP代理服务器
-hpp:port	用于连接到服务器的HTTP代理端口
-ia:file	输入的汇编文件
-id:dir	输入的DLL文件的位置
-is:file	输入的XML计划文件
-nowp	不创建预先包装的代理
-oa:file	保存输出信息的文件
-od:dir	输出文件的目录
-os:file	保存XML计划信息的文件
-p:password	用于连接到服务器的密码
-pn:namespace	在创建的代理文件中代码的代理命名空间
-sdl	创建服务描述语言（SDL）格式文件

(续表)

选项	详细说明
-se:url	服务终点放置输出文件的URL
-sn:file	包括用于标记汇编的关键字的文件
-type:info	输入类型列表
-url:url	服务的URL，从中可以获取XML代码
-u:username	连接到服务器的用户名
-wp	创建预先包装的代理
-wsdl	创建WSDL格式的文件

在以上所有命令行选项中，最常用的是-url选项（指定远程类URL），以及-oa选项（指定DLL文件存储信息）。

建立代理类

下面的命令行创建基本的DLL代理文件，代替原来的远程类文件：

```
soapsuds -url:http://192.168.1.100:9050/MyMathServer?wsdl
-oa:MathClass.dll -nowp
```

运行后产生的MathClass.dll文件随后可以创建客户机应用程序，这如同手工创建原来的类文件（见程序清单16.8）。也可以加入-gc命令行选项，创建DLL文件的C#代码（见程序清单16.13）。

程序清单16.13 用soapsuds程序生成的MathClass.cs文件

```
using System;
using System.Runtime.Remoting.Messaging;
using System.Runtime.Remoting.Metadata;
using System.Runtime.Remoting.Metadata.W3cXsd2001;
[Serializable, SoapType(XmlNamespace=@"http://schemas.microsoft.com/clr/assem/
➡MathClass%2C%20Version%3D0.0.0.0%2C%20Culture%3Dneutral%2C%20PublicKeyToken%3
➡Dnull", %3D0.0.0.0%2C%20Culture%3Dneutral%2C%20PublicKeyToken%3Dnull" )]]
public class MathClass : System.MarshalByRefObject
{
    [SoapMethod(SoapAction=@"http://schemas.microsoft.com/clr/nsassem/
➡ MathClass/MathClass#Add" )]]
    public Int32 Add(Int32 a, Int32 b)
    {
        return((Int32) (Object) null);
    }
}
```

```

[SoapMethod(SoapAction=@"http://schemas.microsoft.com/clr/nsassem/
➡ MathClass/MathClass#Subtract")]
public Int32 Subtract(Int32 a, Int32 b)
{
    return((Int32) (Object) null);
}
[SoapMethod(SoapAction=@"http://schemas.microsoft.com/clr/nsassem/
➡ MathClass/MathClass#Multiply")]
public Int32 Multiply(Int32 a, Int32 b)
{
    return((Int32) (Object) null);
}
[SoapMethod(SoapAction=@"http://schemas.microsoft.com/clr/nsassem/
➡ MathClass/MathClass#Divide")]
public Int32 Divide(Int32 a, Int32 b)
{
    return((Int32) (Object) null);
}
}

```

soapsuds生成的代理类文件复制了远程类的代码，显示出每个类方法是如何定义的，以及类中使用了什么参数。**soapsuds**创建DLL文件之后，就可以着手编写客户机程序，就像手工创建DLL文件一样：

```
csc /r:MathClass.dll MathClient.cs
```

上面这行编译命令将生成**MathClient.exe**文件，其运行结果与程序清单16.11中原来的代理类文件生成的可执行文件一样。

使用预先包装的代理

在先前程序清单16.13的**soapsuds**例子中，**-nowp**选项产生访问远程类的普通代码。客户机可以从任何一个存在远程类的远程技术服务器访问远程类。远程技术**Configuration**信息也被编译到客户机应用程序之中，指明远程类的位置。

如果在**soapsuds**命令行中不包括**-nowp**选项，**soapsuds**将创建预先包装的代理（**wrapped proxy**），它能控制所有的远程技术通信参数配置，包括建立远程类的构造器，连接到**soapsuds**命令中指定的URI。如果使用**-gc**选项，可以看到如下内容：

```

// Constructor
public MathClass()
{
    base.ConfigureProxy(this.GetType(), @"http://192.168.1.100:9050/ ➡
MyMathServer");
}

```

当使用预先包装的代理选项的时候，创建类构造器指定访问远程类的确切位置，所以客户机应用程序不需要远程技术Configuration段。客户机只需要创建远程类的实例，远程技术系统可以自动访问类方法。

程序清单16.14给出NewMathClient.cs程序，其中就使用这种技术访问远程类。

程序清单16.14 NewMathClient.cs程序

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;

public class NewMathClient
{
    public static int Main(string[] argv)
    {
        HttpChannel chan = new HttpChannel();
        ChannelServices.RegisterChannel(chan);
        MathClass obj = new MathClass();
        if (obj == null)
            System.Console.WriteLine("Could not locate server");
        else
        {
            int a = Convert.ToInt32(argv[0]);
            int b = Convert.ToInt32(argv[1]);
            int c = obj.Add(a, b);
            Console.WriteLine("a + b = {0}", c);
            c = obj.Subtract(a, b);
            Console.WriteLine("a - b = {0}", c);
            c = obj.Multiply(a, b);
            Console.WriteLine("a * b = {0}", c);
            c = obj.Divide(a, b);
            Console.WriteLine("a / b = {0}", c);
        }
        return 0;
    }
}
```

使用预先包装的代理DLL会使实际客户机代码更简练。只需要使用默认构造器创建新的MathClass对象，代理的包装会完成剩余的工作。然而，如前所述，远程类服务器的网络地址信息很难编码到代理类中。如果远程类服务器发生变化，代理类必须重新创建。

性能考虑

可以采用多种远程技术组合方式,比如两种信道(TCP和HTTP)、两种串行化格式(BinaryFormatter和SoapFormatter)之间的组合。在开发一个新的应用程序时,可能不清楚哪种组合更适合。很多人认为,在任何情况下最好使用标准协议,比如HTTP信道和SoapFormatter。然而Microsoft公布的测试结果表明,使用TCP信道结合BinaryFormatter具有最高的效率,明显优于HTTP/SOAP的解决方案。如果跨平台兼容性对于应用程序来说并不重要,那么最好使用TCP信道结合BinaryFormatter。

小结

.NET Framework提出了远程技术概念,使应用程序可以在应用域中共享方法,不论是在同一台计算机上,还是在网络中不同的计算机上。

远程技术的一个重要特性是客户机有能力串行化数据并且发送到不同应用域中。.NET库有两种串行化格式类, BinaryFormatter和SoapFormatter,这两个类中的每一个都可以脱离远程技术单独使用,在网络连接中传输串行化数据。

虽然BinaryFormatter和SoapFormatter类可以串行化数据元素,但是仍然要确保所有数据通过网络传输的正确性。可以结合第7章介绍的技术,与串行化数据类一起使用,确保数据正确发送到远程计算机。为串行化数据创建单独的缓存,并且在发送数据的同时发送数据对象的大小,可以确保服务器在收到全部串行化数据之后才开始组装数据。

用.NET远程类创建的类可以被网络上的任意客户机访问。远程类必须源自MarshalByRefObject类,允许远程应用域中的类访问它的方法和数据元素。

为使客户机应用程序访问远程类,必须创建代理类从客户机接收方法调用,并且传递给远程类。创建代理类的或者是远程类的原始数据库,或者使用soapsuds程序访问远程类。soapsuds程序允许确定必要的远程类接口正确访问类方法而不需要原始数据类文件。

远程类必须由远程服务器程序在应用域中注册。远程技术服务器在.NET远程技术系统中注册远程类,随后可以访问远程类的指定信道。远程系统所有部分就位以后,客户机应用程序可以访问远程类中的方法,如同它们处于同一应用域中。

下一章将讨论.NET Framework的安全性。在应用程序安全性日益受到重视的今天,Microsoft公司为.NET Framework设计了很多安全特性,可以用在C#网络程序中。

第17章 网络安全

- 应用程序安全基础知识（策略、组和权限）
- 设置安全策略的工具：caspol和mscorcfg程序
- 说明性和强制性套接字权限
- 实现说明性套接字权限
- 用加密技术保护网络数据

发生在因特网和本地局域网上的黑客攻击越来越多，安全性已经成为程序员们的热门话题。网络编程也不例外，在网络中时刻确保程序和数据的安全是至关重要的。

本章讨论一些.NET安全特性，可以用来维护网络程序的安全，抵抗黑客攻击。第一节介绍用在那些运行于CLR中的应用程序上的.NET安全特性，以及如何使用这些特性控制程序的使用权。第二节说明如何使用套接字权限来阻止服务器程序访问未授权的套接字，或者阻止客户机程序访问未授权的服务器。另外还要介绍说明性和强制性套接字权限这两个.NET概念，并且给出一个使用说明性权限的例子。最后将带领读者认识.NET密码系统库，通过一个例子，掌握如何在网络程序中实现密码保护。

应用程序安全涉及哪些问题

.NET Framework提供很好的安全模式处理应用程序安全问题。每个在系统上运行（或试图运行）的应用程序必须通过安全系统的检查。读者必须在将C#网络程序交付用户之前了解安全模式如何工作，以免网络程序不能在用户的安全环境中运行。

本节讲述.NET应用程序安全模式以及如何在网络应用程序中使用。

安全策略

.NET Framework在CLR（Common Language Runtime，公共语言运行时）中以严格的安全策略控制应用程序。这些策略检查是否允许应用程序在CLR中运行并且访问资源。

CLR安全策略是这样一组规则，它由系统管理员定义，为系统资源设置存取权限。CLR中所有管理代码由安全策略管理。在CLR中有四级安全策略：

- 企业级策略
- 单机级策略
- 用户级策略
- 应用域策略

安全策略是分等级的，这就是说，安全权限从顶端的企业级策略开始逐级向下。较低级别的策略不能将其权限提高到较高级别策略的水平，但是可以降低权限。默认情况下，用

户级和应用域策略的限制少于高级别策略。

企业级策略 企业级策略是CLR中安全策略的最高级别。它们只能由系统管理员（或者管理员组中的用户）设置，并且它们影响运行在企业域中所有的应用程序。

单机级策略 单机级安全策略针对运行在独立计算机系统上的全部应用程序定义安全规则。这个级别定义系统的应用程序策略，是最常用的设置，运行在系统上的全部应用程序都由单机级策略控制。

用户级策略 用户级策略针对计算机系统中单个用户定义安全规则。基于用户级策略，用户会被限制访问系统中的资源。

应用域策略 应用域策略针对运行在不同应用域中的全部应用程序定义安全规则。一个系统可能会同时存在多个应用域，所以一个应用程序可能受到多个应用域策略影响。

安全组

安全组是指控制一组应用程序的策略。可以根据应用程序的特性定义组，例如应用程序生成者或者应用程序的原始位置。

组成员可以交迭，一个应用程序可以是多个组的成员，各组的安全策略都将作用在该应用程序上。表17.1是CLR对应用程序分类的安全组，每组可以有多个实例。例如，可以有多个站点成员组，每组包含来自某个HTTP站点的应用程序。

表17.1 CLR应用程序组

组	作用范围
All code	系统中的全部应用程序
Application directory	应用程序的安装目录
Cryptographic hash	应用程序加密后形成的乱码
Software publisher	应用程序发行人，以数字签名为依据
Site membership	代码所在的HTTP、HTTPS或者FTP站点
Strong name	应用程序经加密的strong name
URL	代码所在的URL路径
Zone	代码所在的区

Zone组定义应用程序的一般位置并且包含以下值：

Internet 从因特网站点下载的应用程序

Intranet 从内部网络服务器下载的应用程序

MyComputer 本地系统中的应用程序

NoZone 未指定区的应用程序

Trusted 来自管理员信任站点的应用程序

Untrusted 来自非管理员信任站点的应用程序

每个区域都分配了不同的访问权限来访问系统资源。来自Internet区域运行的应用程序访问某些系统资源的时候受到限制。MyComputer区域中的应用程序对全部系统资源有完全访问权限。

安全权限

可以让应用程序在访问受限资源之前请求相应的权限，CLR安全策略设置决定赋予权限或者拒绝访问请求。

权限类型

应用程序所在系统的系统管理员负责制定CLR的安全设置，这些设置决定了是否赋予访问者权限。 .NET Framework提供三种类型的权限：

代码访问权限 代码访问权限根据各个应用程序的资源使用情况分配应用程序权限。当应用程序在CLR中运行时，这个权限将检查程序的功能，看它访问哪些资源。如果访问到受限资源，CLR就会抛出一个安全异常，终止程序。

验证权限 验证权限的分配取决于应用程序向CLR提交的一系列证书。作为一个Identity对象提交给CLR的信任书，是用户登录到系统的时候建立的。

角色权限 角色权限与执行应用程序的用户以及希望程序完成的功能有关。可以将权限授予某个用户来实现某个功能，或者授予一组用户实现多个功能。

权限集

一个权限集，标识在一个安全组上实施的一组策略。表17.2列出CLR的7个权限集。

表17.2 CLR权限集

权限集	说明
Nothing	没有授予权限
Execution	应用程序可以运行，但是不能访问受保护的资源
Internet	运行时可以有限制地访问资源，这是来自未知站点应用程序的默认权限设置
LocalIntranet	可以较大范围地访问资源，这是企业内部应用程序的默认权限设置
SkipVerification	应用程序不必经过任何组的验证，授予全部权限
Everything	除SkipVerification以外，授予全部标准权限
FullTrust	提供全部系统资源的全部访问权限

权限集对应安全组，代表系统上的安全策略。例如，单机级策略中的MyComputer组区域可以设置为FullTrust权限，这使得MyComputer区域中的全部应用程序完全访问全部系统资源。

安全工具

.NET Framework在设置系统的CLR安全策略方面提供了两种方法——caspol和mscorcfg。安全策略一旦设定，那么它就会一直生效，直到系统管理员对其进行修改。

caspol程序

caspol程序是一个用于查看和设置系统CLR安全策略的控制台程序，这个程序的选项和参数很多，见表17.3。

表17.3 caspol程序选项

选项	说明
-af	执行汇编，生成一个自定义安全策略
-ag	在现有组结构中添加一个新编码组
-ap	在现有权限集中添加一个新的权限集
-all	用于指明本命令中所有的选项对全部的策略都有效
-cg	改变一个编码组的组成员条件
-cp	改变权限集中的权限
-ca	用于指明本命令中所有的选项适用于指定的用户策略
-cu	指定要修改的自定义用户策略
-e	启用/取消对执行某代码的权限的检查
-en	将所有的权限设置为企业级策略
-f	强行改变策略，即使改变后无法再次运行caspol程序
-h	显示帮助信息
-l	显示当前的编码组合权限集
-listdescription	显示与指定策略级别相应的全部编码组的说明
-lf	显示指定策略级别的完全信任汇编列表
-lg	显示指定的编码组及其权限
-lp	显示指定的权限集
-m	将所有的权限设置为单机级策略
-pp	启用/取消策略等级改变时显示的提示
-r	从备份文件中恢复策略
-rf	从完全信任权限集中删除一个汇编项

(续表)

选项	说明
-rg	删除一个编码组
-rp	删除一个权限集
-rs	将策略等级恢复为默认级别
-rsg	显示包含某个汇编项的所有组
-rsp	显示可用于某个汇编项的所有权限集
-s	启用/关闭CLR安全选项
-u	将所有的权限设置为用户级策略

列出组的权限

使用-lg选项，可以查看系统当前采用的所有组权限：

```
C:\>caspol -lg
Microsoft (R) .NET Framework CasPol 1.0.3705.288
Copyright (C) Microsoft Corporation 1998-2001. All rights reserved.

Security is ON
Execution checking is ON
Policy change prompt is ON

Level = Machine

Code Groups:

1. All code: Nothing
  1.1. Zone - MyComputer: FullTrust
    1.1.1. StrongName - 436518206DC093344D5AD293: FullTrust
    1.1.2. StrongName - 00000000000000000040000000000000: FullTrust
  1.2. Zone - Intranet: LocalIntranet
    1.2.1. All code: Same site Web.
    1.2.2. All code: Same directory FileIO - Read, PathDiscovery
  1.3. Zone - Internet: Nothing
  1.4. Zone - Untrusted: Nothing
  1.5. Zone - Trusted: Internet
    1.5.1. All code: Same site Web.

Success

C:\>
```

清单所示的前三项是系统当前的全局安全设置，可以看到，CLR安全选项已经启用，并启用了所有的安全策略。要取消系统的安全策略，可以使用-s选项：

```
C:\>caspol -s off
Microsoft (R) .NET Framework CasPol 1.0.3705.288
Copyright (C) Microsoft Corporation 1998-2001. All rights reserved.

Success

C:\>
```

这条命令的威力很大，它可以停止系统全部的安全策略，这时网络上的任何.NET应用程序都可以获取系统的最高控制权。再次使用-s选项可以重新启用安全策略。

警告：如果停用系统内全部的CLR安全策略，系统就无法抵御非法攻击，这个问题在连接因特网的情况下更为突出。如果在安全策略方面遇到麻烦，千万不要用停用安全策略这样的办法来回避问题，应该搞清楚系统需要使用哪些安全策略，然后尽可能多地启用它们。

接下来的策略级别选项也处于开启状态，并显示当前的策略级别是单机级。如果没有选定策略级别，那么在这里不会显示任何级别。

接下来显示的是每个组及其策略情况。**All-code**是最高级别的组，其权限集是**Nothing**，这是系统的默认设置，使任何应用程序都不能访问任何主机资源，无疑这是最安全的。在**All-code**组之后显示的是该组内的各个区：

- 第一个区名为**MyComputer**，这个小组内的应用程序都具有**FullTrust**权限集，这表示它们可以任意访问主机资源。**C#**程序如果拥有这个级别的权限，就可以访问计算机的硬盘和网络之类的资源。
- **MyComputer**区又分成了两个大同小异的子区，两个子区都包含一个名为**Strong name**的程序，其名称均以数字密码显示。第一个这样的程序是**Microsoft**创建的，这个安全策略允许经**Microsoft**签署的任何应用程序获得**FullTrust**权限集，允许它们使用系统资源。另一个程序则由**ECMA**创建。
- 第二个区名为**LocalIntranet**，用来给位于同一个网站或同一个应用程序目录中的全部应用程序赋予一定的权限。
- 另外两个区，一个是**Internet**，另一个是**Untrusted**，被赋予的权限集是**Nothing**，这表示这两个区的成员不能访问系统资源。
- 最后一个区是**Trusted**，为标志成可信的应用程序分配**Internet**权限集。

列出应用程序的权限

如果提交用户的程序由于用户安全策略设置问题不能正常运行，那么就可能要为应用程序组设置别的权限集。通过**Microsoft**网络的网络共享功能运行的应用程序就是个常见的例子。

位于网络共享驱动器上的应用程序，它属于**Internet**区还是**Intranet**区，取决于远程文件服务器是否属于企业内部互联网。如果不清楚.NET会将该文件服务器分为哪一类，可以使用**caspol -resolvegroup**选项检查.NET将应用程序划在哪一区：

```
Z:\>caspol -resolvegroup NewTcpChat.exe
Microsoft (R) .NET Framework CasPol 1.0.3705.288
```

```
Copyright (C) Microsoft Corporation 1998-2001. All rights reserved.

Level - Enterprise

Code Groups:

1. All code: FullTrust

Level - Machine

Code Groups:

1. All code: Nothing
  1.1. Zone - Intranet: LocalIntranet
    1.1.1. All code: Same site Web.
    1.1.2. All code: Same directory FileIO - Read, PathDiscovery

Level - User

Code Groups:

1. All code: FullTrust

Success

Z:\>
```

这个应用程序所在的远程文件服务器被归入**Intranet**（单机级策略）并且分配了**LocalIntranet**权限集。该程序可以运行，但是访问本地资源受到一定限制。

对于网络程序员来说，套接字是受到限制的本地资源之一。为使网络程序工作正常，必须给**Intranet**区提高权限，以允许访问套接字。

修改权限

可以使用**caspol**程序修改安全组的权限集，使应用程序正常运行。观察前文所用的**caspol -lg**命令的结果可见，**Intranet**组具有**LocalIntranet**权限集。为使网络程序能在共享网络驱动器中运行，必须使它们具备**FullTrust**或者**Everything**权限集。可以将其分配给**Everything**权限集，授予其足够的而不是全部的权限。

要修改组的权限集，可以使用**caspol -cg**选项。由于是为单机级策略而设置，所以必须要包括**-m**选项。可以用分给安全组的编号来引用安全组，正如上文中使用**-lg**时的输出结果所示。对于**Intranet**区，安全组编号为1.2。

下面显示**caspol**命令修改单机级策略上**Intranet**区的权限集时的输出结果：

```
C:\>caspol -m -cg 1.2 Everything
Microsoft (R) .NET Framework CasPol 1.0.3705.288
Copyright (C) Microsoft Corporation 1998-2001. All rights reserved.

The operation you are performing will alter security policy.
Are you sure you want to perform this operation? (yes/no)
Y
Changed code group permission set to "Everything" in the Machine level.
```


Success

C:\>

通过-lg命令的输出结果可以证明权限集已经修改了。下面的命令行说明Intranet组的权限集现在已经修改为Everything:

1.2. Zone - Intranet: Everything

现在网络程序就可以从网络服务器的共享驱动器运行了。

mscorcfg程序

在Windows 2000和XP系统中,可以在Microsoft管理控制台(Microsoft Management Console, mmc.exe)中使用.NET插件,查看及修改安全策略。mscorcfg.msc程序以图形界面方式查看和修改CLR系统中的安全策略。程序位于.NET应用程序目录,通常在系统的Windows或者Winnt目录下的Microsoft.NET子目录中。

双击该应用程序,出现mscorcfg主界面。在界面左侧,可以看到设置.NET Framework的选项。这些选项之一是Runtime Security Policy(运行时安全策略)。展开这一项将显示三个主要的策略级:Enterprise、Machine和User。展开Machine策略级可以看到分配给这个策略级的代码组和权限集,如图17.1所示。

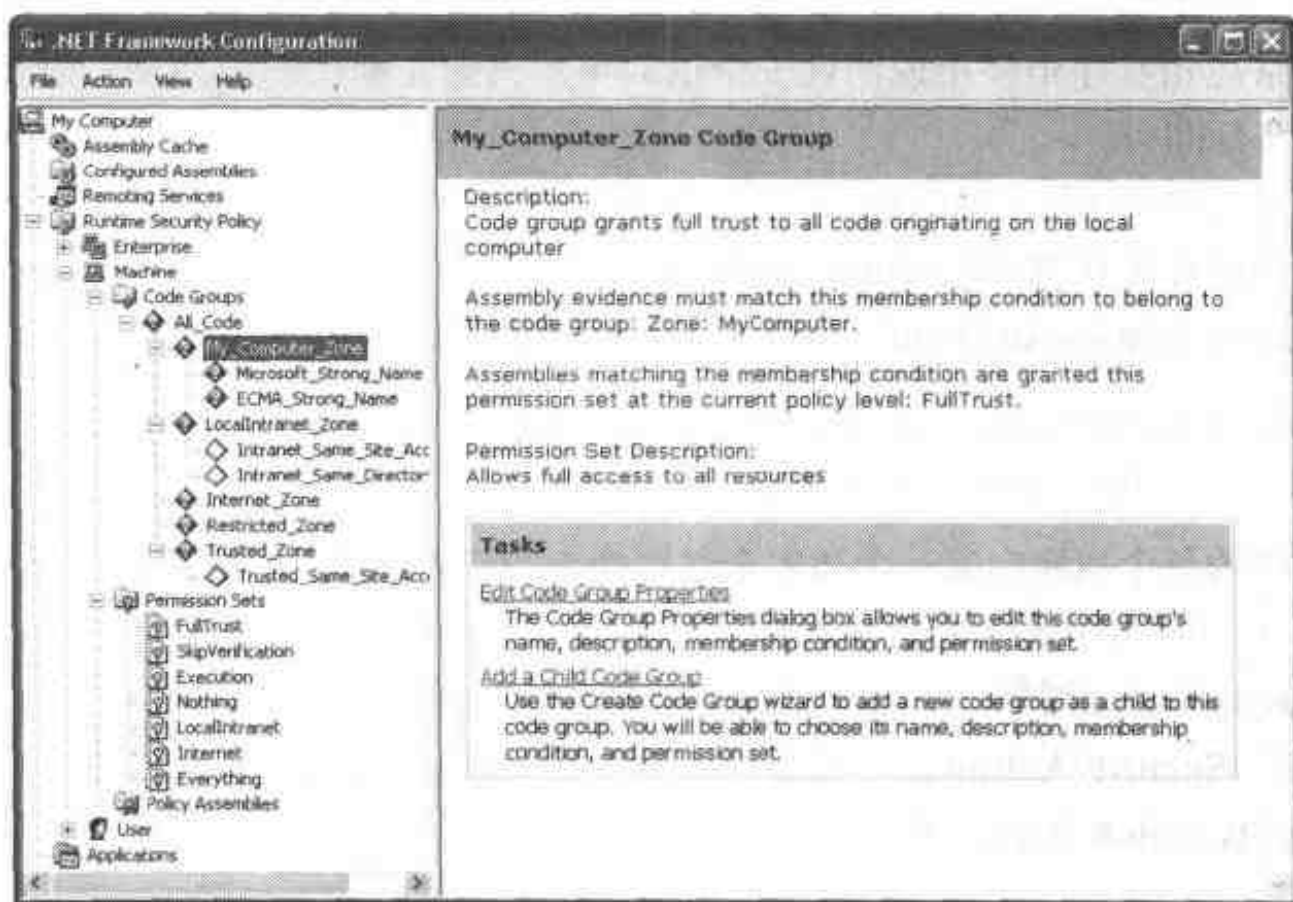


图17.1 mscorcfg运行时安全策略界面

展开每一项,可以看到All Code组下面的全部代码组以及为每个组定义的权限集。这些信息显示在应用程序窗口的右侧,还包括执行一般策略管理功能的链接。运行mscorcfg程序得到的信息与运行caspol程序得到的非常相似,只不过mscorcfg用图形方式显示。

类似于caspol程序,可以使用应用程序窗口右侧的链接修改每个策略级的安全设置。

套接字权限

网络应用程序的一个常见特性是，可以允许或者拒绝网络中特定客户机对应用程序的访问。这一特性允许应用程序管理员规定一个基本安全级别阻止其他用户访问应用程序。

可以使用System.Net命名空间中的SocketPermission和SocketPermissionAttribute类给C#网络应用程序添加这项功能。这些类提供的技术允许基于客户机网络地址控制访问C#网络程序，这两个类可以实现Socket对象的说明性安全和强制性安全。

说明性安全 在使用SocketPermissionAttribute类时定义应用程序的安全属性。这种技术使用应用程序代码的元数据来定义应用程序的安全限制，每项限制定义为一个属性，可以用SocketPermissionAttribute类定义attribute属性。

强制性安全 在应用程序中生成SocketPermission对象以控制访问。对于强制性安全，必须生成SocketPermission对象并且给它分配一组安全权限。

到目前为止，说明性安全是二者中最容易以及最常用的技术。强制性安全需要生成新的SocketPermission对象，并且给对象分配正确的权限，比说明性安全复杂得多，不在本章讨论范围内。下一节讨论如何在网络程序中容易地使用说明性安全。

实现说明性安全

在网络应用程序中使用说明性安全很简单，只要在源代码中为每个需要的安全属性分别添加属性行即可。

安全属性的属性行必须以正确的格式出现。属性必须在应用程序代码中使用方括号（[]）括起来，例如在第16章中的[Serializable]属性。

SocketPermissionAttribute类属性在类构造函数中定义：

```
[SocketPermission(SecurityAction act, Access=acc, Host=host,
                  Port=port, Transport=trans)]
```

构造函数中的每个属性定义一个被安全属性检查的元素，属性中的所有元素必须满足属性才能生效。下面是类中定义五个参数：

SecurityAction对象 SecurityAction对象定义由属性执行的指定安全行为。表17.4列出可能的SecurityAction值。最常用的值是SecurityAction.Assert，允许访问资源，还有SecurityAction.Deny，拒绝访问资源。

表17.4 SecurityAction成员

成员	说明
Assert	可以访问资源，甚至更高级属性拒绝访问时也可以执行
Demand	调用这个类必须有足够权限
Deny	拒绝访问指定资源

(续表)

成员	说明
InheritanceDemand	必须有足够权限才能继承该类
LinkDemand	直接调用者必须有足够权限
PermitOnly	只能访问指定资源
RequestMinimum	请求运行代码需要的最小权限
RequestOptional	请求可选的附加权限
RequestRefuse	请求不授予那些有可能被滥用的权限

Access属性 Access属性定义安全属性允许的网络访问方法。只有两个值用在这里：

Accept 允许（或者拒绝）绑定到指定套接字

Connect 允许（或者拒绝）连接到远程套接字

警告：使用Accept Access属性要谨慎。这个属性的名字听起来好像可以根据远程地址限制收到的连接请求，实际并非如此。事实上它指定的是限制绑定的本地套接字地址。

Host属性 Host属性在安全属性中定义主机名或者地址。这个值是主机名字符串或者8位IP地址串。使用通配符可以定义更多网络地址：

```
Host="192.168.1.*"
```

Port属性 Port属性在安全属性中定义TCP或者UDP端口。这个值由某个端口号指定，或者使用特殊值All，代表指定传输端口和主机的全部端口。

Transport属性 Transport属性在安全属性中定义套接字类型。这个值为以下字符串值之一：

All 所有传输类型

Connectionless 无连接传输，例如UDP

ConnectionOriented 有连接传输，例如TCP

Tcp TCP传输

Udp UDP传输

应用程序代码中列出的属性按照它们的执行顺序排列。每个属性根据试图访问应用程序的对象分别检查。一旦某个属性拒绝访问，该对象就被禁止访问应用程序。

使用说明性安全

在网络中使用说明性安全很容易，只需在类文件中添加属性声明然后照常编译程序即可。当运行应用程序的时候，属性中设置的套接字权限将影响程序和本地套接字之间的连接或绑定。

客户机程序

程序清单17.1的PickyTcpClient.cs程序修改自程序清单7.1的TcpClientSample.cs程序，将说明性安全限制并入其中，限制客户机可以连接到的远程主机。

程序清单17.1 PickyTcpClient.cs程序

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Security;
using System.Security.Permissions;
using System.Text;

[SocketPermission(SecurityAction.Deny, Access="Connect", Host="127.0.0.1",
[SocketPermission(SecurityAction.Deny, Access="Connect", Host="192.168.0.2",
    Port="All", Transport="All"))
[SocketPermission(SecurityAction.Deny, Access="Connect", Host="192.168.1.100",
    Port="80", Transport="All"))

class PickyTcpClient
{
    public static void Main()
    {
        byte[] data = new byte[1024];
        string input, stringData;
        TcpClient server = null;

        Console.Write("Enter a host to connect to: ");
        string stringHost = Console.ReadLine();

        try
        {
            server = new TcpClient(stringHost, 9050);
        } catch (SocketException)
        {
            Console.WriteLine("Unable to connect to server");
            return;
        } catch (SecurityException)
        {
            Console.WriteLine(
                "Sorry, you are restricted from connecting to this server");
            return;
        }

        NetworkStream ns = server.GetStream();

        int recv = ns.Read(data, 0, data.Length);
        stringData = Encoding.ASCII.GetString(data, 0, recv);
```

```

        Console.WriteLine(stringData);

        while(true)
        {
            input = Console.ReadLine();
            if (input == "exit")
                break;
            ns.Write(Encoding.ASCII.GetBytes(input), 0, input.Length);
            ns.Flush();

            data = new byte[1024];
            recv = ns.Read(data, 0, data.Length);
            stringData = Encoding.ASCII.GetString(data, 0, recv);
            Console.WriteLine(stringData);
        }
        Console.WriteLine("Disconnecting from server...");
        ns.Close();
        server.Close();
    }
}

```

可以看出，网络程序代码本身与原来的TcpClientSample.cs程序很接近，区别在于程序开始定义的元数据。程序开始指定的安全属性限制了客户机访问的套接字。下面是一个属性例子：

```

[SocketPermission(SecurityAction.Deny, Access="Connect", Host="127.0.0.1",
    Port="All", Transport="All")]

```

这个属性限制客户机程序访问主机127.0.0.1（本地计算机）上的任何端口。

当套接字权限属性触发的时候会抛出一个SecurityException异常，catch语句会捕获这一错误信息并且生成明确的错误信息：

```

try
{
    server = new TcpClient(stringHost, 9050);
} catch (SocketException)
{
    Console.WriteLine("Unable to connect to server");
    return;
} catch (SecurityException)
{
    Console.WriteLine(
        "Sorry, you are restricted from connecting to this server");
    return;
}

```

这个程序的输出结果是：

```
C:\>PickyTcpClient
Enter a host to connect to: 127.0.0.1
Sorry, you are restricted from connecting to this server

C:\>
```

不出所料，连接尝试被拒绝，抛出了SecurityException异常。

服务器程序

说明性安全特性也适用于服务器，但是有微小差异。为服务器程序定义的安全属性也会限制本地套接字的访问，安全属性中定义的地址和端口与服务器本地地址和端口是对应的。

如果地址和端口受限，应用程序就不能在套接字上实现Bind()方法。下面是服务器安全属性的一个例子：

```
[SocketPermission(SecurityAction.Deny, Access="Accept", Host="0.0.0.0",
    Port="9050", Transport="All")]
```

安全属性阻止应用程序绑定到本地计算机的TCP端口9050。

说明：当指定主机安全属性为Accept时要小心。如果使用计算机内部的回环IP地址，应用程序可能仍然能够绑定到外部IP地址。特殊的IP地址IPAddress.Any (0.0.0.0)可以代表全部地址。

程序清单17.2的PickyTcpListener.cs程序源自程序清单7.2的TcpListenerSample.cs程序，其中加入了套接字权限属性。

程序清单17.2 PickyTcpListener.cs程序

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Security;
using System.Security.Permissions;
using System.Text;

[SocketPermission(SecurityAction.Deny, Access="Accept", Host="0.0.0.0",
    [SocketPermission(SecurityAction.Deny, Access="Accept", Host="0.0.0.0",
        Port="9051", Transport="All")])
[SocketPermission(SecurityAction.Deny, Access="Accept", Host="0.0.0.0",
    Port="9052", Transport="All")])

class PickyTcpListener
{
    public static void Main()
    {
        int recv;
```

```
TcpListener newsock = null;
byte[] data = new byte[1024];

Console.Write("Enter port number to use: ");
string stringPort = Console.ReadLine();
int port = Convert.ToInt32(stringPort);

try
{
    newsock = new TcpListener(port);
    newsock.Start();
} catch (SecurityException)
{
    Console.WriteLine("Sorry, that port is unavailable");
    return;
}

Console.WriteLine("Waiting for a client...");

TcpClient client = newsock.AcceptTcpClient();
NetworkStream ns = client.GetStream();

string welcome = "Welcome to my test server";
data = Encoding.ASCII.GetBytes(welcome);
ns.Write(data, 0, data.Length);

while(true)
{
    data = new byte[1024];
    recv = ns.Read(data, 0, data.Length);
    if (recv == 0)
        break;

    Console.WriteLine(
        Encoding.ASCII.GetString(data, 0, recv));
    ns.Write(data, 0, recv);
}
ns.Close();
client.Close();
newsock.Stop();
}
```

PickyTcpListener.cs程序限制应用程序访问3个端口（实际上是6个，3个TCP传输端口和3个UDP传输端口）。如果应用程序试图访问其中一个受限制的端口，就会抛出一个**SecurityException**异常，生成一个错误信息。

保护网络数据

通过因特网发送数据是有风险的，因特网节点之间传输的数据包很可能被恶意拦截。很多需要通过因特网通信的应用程序会采用加密系统来保护数据。虽然加密并非万无一失，它终究能为应用程序提供起码的数据保护。 .NET Framework 包含一些加密类，可以用来保护网络数据。

警告：加密并非牢不可破，无论加密方法多么复杂，它总能被破解。关键是当花费很长时间终于解密时，数据很可能已经过期失效了。

数据加密

在网络上发送数据的时候，任何人都可以截获数据包并且查看其内容。如果发送的是敏感数据，最好在打包发送前对其加密。

.NET库的System.Security.Cryptography命名空间包含很多加密数据的类，涉及多种加密算法，并且很难区分它们。基本上，它们分为两种加密/解密方案：对称加密和非对称加密。

对称加密

对称加密算法在加密和解密数据时使用同一个密钥，通信双方必须都持有该密钥。对称加密也称为私钥加密，只用一个私钥就可以完成加密和解密。

对称加密算法以块为单位加密数据，为了保证每个块大小一致，常常需要填充数据。每块加密数据与先前加密的数据块结成链（称为cipher block chaining），因此对称加密支持数据流，是加密网络数据的理想方式。

为保证每块加密数据都是惟一的（这样黑客就难以推算到数据模式），需要用到另一个重要的参数，名为初始向量（initialization vector, IV），与私钥一起使用。密钥和IV一起决定如何加密数据，以及如何将数据解密为原始格式。

.NET库提供了一些对称加密算法类，见表17.5。

表17.5 .NET对称加密类

类	说明
DESCryptoServiceProvider	实现DES加密运算
RC2CryptoServiceProvider	实现RC2加密运算
RijndaelManaged	实现Rijndael Managed加密运算
TripleDESCryptoServiceProvider	实现三重DES加密运算

不对称加密

不对称加密允许向其他用户提供密钥进行数据加密（称为公钥），自己保留配套的私

钥，可以对经公钥加密的数据进行解密。这种技术称为公钥加密。

公钥可以加密数据，但只有私钥才能解密数据。持有公钥的人不能解密数据，只能加密数据。这种不对称加密方式对于需要接受多方数据的人非常理想，不必担心数据被其他人解密。

不对称加密算法采用固定块加密技术，加密的数据块不能形成链。因此，不对称加密很难用在网络环境中，本章不提供这方面应用的例子。

说明：常常用对称加密算法加密网络数据，然后用不对称加密算法加密发送对称私钥。

.NET库提供两个不对称加密类，以私钥和公钥加密数据。

- `DSACryptoServiceProvider`
- `RSACryptoServiceProvider`

使用数据加密

在网络程序中使用对称加密比较容易出错，加密数据并通过网络发送数据需要很多步骤，稍不注意就可能出错。

加密数据

.NET对称加密类用流传递数据。处理流中的数据就很容易出错，各种常见的数据处理方面的问题都可能出现。

例如，`TripleDESCryptoServiceProvider`类使用三重DES加密运算提供加密功能，`CryptoStream`类将加密数据块传递给`Stream`对象，这个流可以是`FileStreams`、`MemoryStreams`或者`NetworkStreams`。

`CryptoStream`构造函数需要三个参数：

```
CryptoStream(Stream stream, ICryptoTransform transform, CryptoStreamMode mode)
```

第一个参数`stream`代表外部的`Stream`对象，加密数据将发送给它（或从中读取）。

第二个参数`transform`代表加密算法。要生成这个值，必须使用`TripleDESCryptoServiceProvider`类的`CreateEncryptor()`方法，这里`key`参数指私钥的值，`iv`参数就是初始向量：

```
CreateEncryptor(Key key, IV iv)
```

这两个值都是16字节的字节序列，这些值在通信双方必须完全相同。

构造函数最后一个参数是`mode`，定义流读（`CryptoStreamMode.Read`）还是写（`CryptoStreamMode.Write`）。`CryptoStream`对象不能既读又写。

使用对称加密方法加密数据的代码段如下：

```
FileStream fs = new FileStream("test.enc",  
    FileAccess.Create);  
TripleDESCryptoServiceProvider tdes = new TripleDESCryptoServiceProvider();  
CryptoStream csw = new CryptoStream(fs,  
    Tdes.CreateEncryptor(key, iv), CryptoStreamMode.Write);
```

```
byte[] data = Encoding.ASCII.GetBytes("Phrase to encrypt");
csw.Write(data, 0, data.Length);
csw.Close();
fs.Close();
```

一旦生成CryptoStream对象，写入流中的数据就被加密，然后传递给构造函数定义的流。

解密数据

解密过程类似于加密过程：数据经解密程序传递给指定的流对象。解密程序流用TripleDESCryptoServiceProvider类的CreateDecryptor()方法生成：

```
CryptoStream csr = new CryptoStream(memstrm,
    tdes.CreateDecryptor(key, iv), CryptoStreamMode.Read);
```

警告：必须使用加密数据时所用的密钥和iv值解密数据。通常这些值通过安全的传统邮件传递，或者通过机要挂号信传递。

对称解密的一个简单例子如下：

```
FileStream fs = new FileStream("test.enc", FileAccess.Open);
TripleDESCryptoServiceProvider tdes = new TripleDESCryptoServiceProvider();
CryptoStream csr = new CryptoStream(fs,
    tdes.CreateDecryptor(key, iv), CryptoStreamMode.Read);
byte[] data = new byte[1024];
int recv = csr.Read(data, 0, data.Length);
string phrase = Encoding.ASCII.GetString(data, 0, recv);
csr.Close();
fs.Close();
```

CryptoStream对象再次指向FileStream对象，但是这次定义的是解密程序，并且使用CryptoStreamMode.Read来从流中读取数据。当数据从流中读出来，就自动解密还原为原始文本。

简单加密/解密示例程序

为了了解加密技术的实现，这里给出一个简单的例子，先加密文本然后解密还原为原始文本。程序清单17.3的CryptoTest.cs程序是一个简单的例子，演示如何在C#程序中添加加密功能。

程序清单17.3 CryptoTest.cs程序

```
using System;
using System.IO;
using System.Security;
using System.Security.Cryptography;
using System.Text;
```



```

class CryptoTest
{
...public static void Main()
{
    Console.Write("Enter phrase to encrypt: ");
    string phrase = Console.ReadLine();
    MemoryStream memstrm = new MemoryStream();

    byte[] Key = {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
                  0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16};
    byte[] IV = {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
                0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16};

    TripleDESCryptoServiceProvider tdes = new TripleDESCryptoServiceProvider();
    CryptoStream csw = new CryptoStream(memstrm,
                                         tdes.CreateEncryptor(Key, IV), CryptoStreamMode.Write);

    csw.Write(Encoding.ASCII.GetBytes(phrase), 0, phrase.Length);
    csw.FlushFinalBlock();
    byte[] cryptdata = memstrm.GetBuffer();
    Console.WriteLine("Encrypted: {0}",
                      Encoding.ASCII.GetString(cryptdata, 0, (int)memstrm.Length));

    memstrm.Position = 0;
    byte[] data = new byte[1024];
    CryptoStream csr = new CryptoStream(memstrm,
                                         tdes.CreateDecryptor(Key, IV),
                                         CryptoStreamMode.Read);
    int recv = csr.Read(data, 0, data.Length);
    string newphrase = Encoding.ASCII.GetString(data, 0, recv);
    Console.WriteLine("Decrypted: {0}", newphrase);
    csr.Close();
    csw.Close();
    memstrm.Close();
}
}

```

CryptoTest.cs程序使用**MemoryStream**控制加密数据。密钥和IV值没有编到程序中，只随便选了16字节值。对于真正加密应用程序，应该为这些参数确定真实值并且保护起来。

字符串转换为字节序列之后，加密并且存储到**MemoryStream**中，调用**FlushFinalBlock()**方法。该方法确保全部加密的数据块都发送到流，没有留在加密类的缓冲空间内。

字符串被加密之后，**MemoryStream**内容转换为字节序列并且在控制台中以字符串方式显示（因为加密值包含非ASCII码文本，可能看到一些有意思的结果）。目的是为了看到数据确实被加密了。

加密数据被显示之后，**MemoryStream**对象的位置指针重置到流的开始，解密方法将加

密文本转换回原始字符串。下面是程序的输出结果：

```
C:\>CryptoTest
Enter phrase to encrypt: A test phrase.
Encrypted: ~/@'F?p{=. $q=R2U
Decrypted: A test phrase.

C:\>
```

网络数据加密

尽管用CryptoStream类处理FileStream和MemoryStream对象比较方便，但它不易处理NetworkStream对象。让CryptoStream类处理NetworkStream对象，却难以确定数据流中的数据边界。数据被加密后直接传递给NetworkStream对象，接收者很难确定何时接收到加密数据的末尾。如果解密程序在流全部到达之前解密数据，就会产生错误。

操控加密数据

数据范围问题的解决方案是采用第7章“使用C#套接字助手类”介绍过的一种流数据处理技术。如第7章所述，操控流数据可以采用预先规定流的长度，在发送流之前先发送长度信息，或者使用信息范围标记。

最简单的办法就是计算每个加密数据包的大小，并且在发送流数据之前先发送长度信息。接收程序可以从NetworkStream提取数据长度，并且确切知道从网络中读取多少字节数据才能正确形成加密信息。然后接收程序可以将完整的加密信息转发给相应的解密程序，提取原始数据。

要计算每个加密数据包的大小，必须在加密数据发送到NetworkStream之前将信息存储在缓存中。加密数据存入缓存以后，很容易计算数据包长度，然后将数据大小和数据全部发送到NetworkStream。CryptoTest.cs程序使用MemoryStream类，是很简单的解决方案。通过将CryptoStream转发到MemoryStream，可以使用GetBuffer()方法生成加密数据的字节序列。然后该序列可以像其他数据的字节序列一样发送到网络。

在接收端执行相反的步骤。从NetworkStream中读取数据送入CryptoStream对象解密，指向MemoryStream对象。解密数据全部进入MemoryStream对象以后，就可以使用GetBuffer()方法提取数据。

图17.2说明网络数据加密的发送过程和接收过程。

CryptoDataSender程序

程序清单17.4中的CryptoDataSender.cs程序演示了将加密的复杂数据发送到网络。CryptoDataSender源自BetterDataSender.cs程序（程序清单16.6），加入了三重DES加密确保数据安全发送。

说明：与原先的成员一样，CryptoDataSender程序必须与SerialEmployee.dll文件一同编译，这个文件在第16章讨论过，通过使用/r编译选项生成：csc /r:SerialEmployee.dll CryptoDataSender.cs。

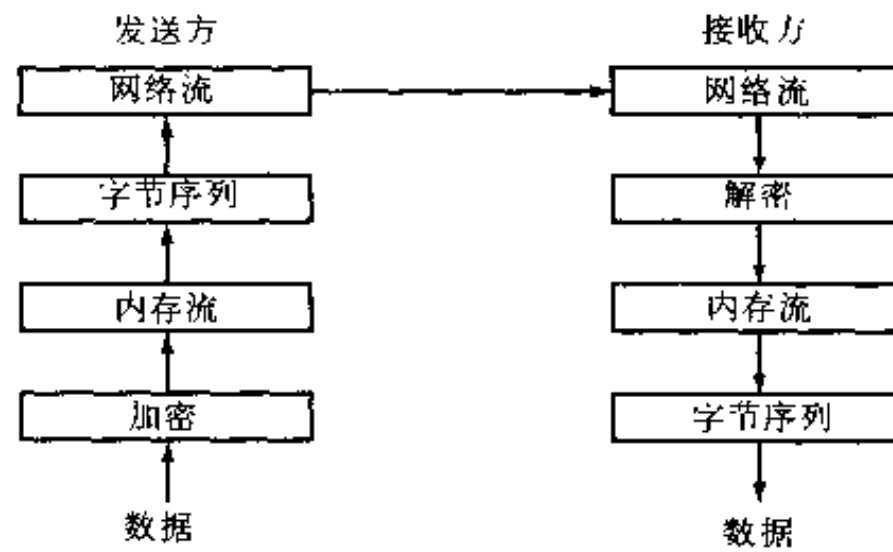


图17.2 加密网络数据

程序清单17.4 CryptoDataSender.cs程序

```

using System;
using System.IO;
using System.Net;
using System.Net.Sockets;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Soap;
using System.Security;
using System.Security.Cryptography;
using System.Text;

class CryptoDataSender
{
    private void SendData (NetworkStream strm, SerialEmployee emp)
    {
        IFormatter formatter = new SoapFormatter();
        MemoryStream memstrm = new MemoryStream();

        byte[] Key = {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x10,
0x11, 0x12, 0x13, 0x14, 0x15, 0x16};
        byte[] IV = {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x10,
0x11, 0x12, 0x13, 0x14, 0x15, 0x16};

        TripleDESCryptoServiceProvider tdes = new
TripleDESCryptoServiceProvider();
        CryptoStream csw = new CryptoStream(memstrm, tdes.CreateEncryptor(Key,
IV), CryptoStreamMode.Write);

        formatter.Serialize(csw, emp);
        csw.FlushFinalBlock();
        byte[] data = memstrm.GetBuffer();
        int memsize = (int)memstrm.Length;
        byte[] size = BitConverter.GetBytes(memsize);
        strm.Write(size, 0, 4);
    }
}

```

```
        strm.Write(data, 0, (int)memsize);
        strm.Flush();
        csw.Close();
        memstrm.Close();
    }

    public CryptoDataSender()
    {
        SerialEmployee emp1 = new SerialEmployee();
        SerialEmployee emp2 = new SerialEmployee();

        emp1.EmployeeID = 1;
        emp1.LastName = "Blum";
        emp1.FirstName = "Katie Jane";
        emp1.YearsService = 12;
        emp1.Salary = 35000.50;

        emp2.EmployeeID = 2;
        emp2.LastName = "Blum";
        emp2.FirstName = "Jessica";
        emp2.YearsService = 9;
        emp2.Salary = 23700.30;

        TcpClient client = new TcpClient("127.0.0.1", 9050);
        NetworkStream strm = client.GetStream();

        SendData(strm, emp1);
        SendData(strm, emp2);
        strm.Close();
        client.Close();
    }

    public static void Main()
    {
        CryptoDataSender cds = new CryptoDataSender();
    }
}
```

读者可能还记得，BetterDataSender程序使用SOAP串行功能对雇员数据对象进行串行化，然后通过网络发送到远程设备。由于使用SOAPFormatter，全部数据以文本格式通过网络传输。如果数据中包含敏感信息比如社会安全号码，那么任何人截获数据包后都可以看到这些信息，谁都不希望在应用程序中发生这样的事。

CryptoDataSender程序生成同样的SOAP数据串行，而在发送数据前通过CryptoStream对象加密信息：

```
formatter.Serialize(csw, emp);
csw.FlushFinalBlock();
```

```

byte[] data = memstrm.GetBuffer();
int memsize = (int)memstrm.Length;
byte[] size = BitConverter.GetBytes(memsize);
strm.Write(size, 0, 4);
strm.Write(data, 0, (int)memsize);

```

SOAPFormatter的Serialize()方法使用CryptoStream流对象而不是NetworkStream将雇员数据串行化成流。FlushFinalBlock()方法保证全部加密数据块都进入缓存。由于CryptoStream对象指向MemoryStream对象，对象包含全部加密的SOAP串行的SerialEmployee数据对象。然后MemoryStream对象转换为字节序列，测定数据大小，一同通过NetworkStream对象发送给远程接收者。

CryptoDataRcvr程序

现在来看接收程序，程序清单17.5的CryptoDataRcvr.cs程序演示如何从NetworkStream读取和加密数据。CryptoDataRcvr.cs程序仿照程序清单16.7的BetterDataRcvr程序编写。

说明：再次说明，CryptoDataRcvr程序必须用第16章的SerialEmployee.dll文件进行编译，使用/r:选项：
csc/r:SerialEmployee.dll CryptoDataRcvr.cs。

程序清单17.5 CryptoDataRcvr.cs程序

```

using System;
using System.IO;
using System.Net;
using System.Net.Sockets;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Soap;
using System.Security;
using System.Security.Cryptography;
using System.Text;

class CryptoDataRcvr
{
    private SerialEmployee RecvData (NetworkStream strm)
    {
        MemoryStream memstrm = new MemoryStream();

        byte[] Key = {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
                      0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16};
        byte[] IV = {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
                    0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16};

        TripleDESCryptoServiceProvider tdes = new
        TripleDESCryptoServiceProvider();

        CryptoStream csw = new CryptoStream(memstrm, tdes.CreateDecryptor(Key,
IV),
CryptoStreamMode.Write);

```

```

byte[] data = new byte[2048];
int recv = strm.Read(data, 0, 4);
int size = BitConverter.ToInt32(data, 0);
int offset = 0;
while(size > 0)
{
    recv = strm.Read(data, 0, size);
    csw.Write(data, offset, recv);
    offset += recv;
    size -= recv;
}
csw.FlushFinalBlock();
IFormatter formatter = new SoapFormatter();
memstrm.Position = 0;
SerialEmployee emp = (SerialEmployee)formatter.Deserialize(memstrm);
memstrm.Close();
return emp;
}

public CryptoDataRcvr()
{
    TcpListener server = new TcpListener(9050);
    server.Start();
    Console.WriteLine("Waiting for a client...");
    TcpClient client = server.AcceptTcpClient();
    NetworkStream strm = client.GetStream();

    SerialEmployee emp1 = RecvData(strm);
    Console.WriteLine("emp1.EmployeeID = {0}", emp1.EmployeeID);
    Console.WriteLine("emp1.LastName = {0}", emp1.LastName);
    Console.WriteLine("emp1.FirstName = {0}", emp1.FirstName);
    Console.WriteLine("emp1.YearsService = {0}", emp1.YearsService);
    Console.WriteLine("emp1.Salary = {0}\n", emp1.Salary);

    SerialEmployee emp2 = RecvData(strm);
    Console.WriteLine("emp2.EmployeeID = {0}", emp2.EmployeeID);
    Console.WriteLine("emp2.LastName = {0}", emp2.LastName);
    Console.WriteLine("emp2.FirstName = {0}", emp2.FirstName);
    Console.WriteLine("emp2.YearsService = {0}", emp2.YearsService);
    Console.WriteLine("emp2.Salary = {0}", emp2.Salary);
    strm.Close();
    server.Stop();
}

public static void Main()
{

```



```
CryptoDataRcvr cdr = new CryptoDataRcvr();  
}  
}
```

第16章给出的BetterDataRcvr程序中，从NetworkStream中读取数据直接反馈给SOAP Deserialize()方法。这里的CryptoDataRcvr程序明显包括其他步骤。由于新的数据被加密，所以必须在反串行之之前将其解密。

首先，从NetworkStream中读取加密的数据队列大小。知道数据队列大小之后，使用while循环保证从网络中读取全部加密数据：

```
while(size > 0)  
{  
    data = new byte[2048];  
    recv = strm.Read(data, 0, size);  
    csw.Write(data, offset, recv);  
    offset += recv;  
    size -= recv;  
}  
csw.FlushFinalBlock();
```

每个数据块都从网络中读取，反馈给CryptoStream流，解密数据并且存储在生成的MemoryStream对象中。当全部数据从网络中接收时，FlushFinalBlock()方法确保全部解密的数据块已经发送给了MemoryStream。

全部解密数据存入MemoryStream对象之后，重置指针，在Deserialzier()方法中使用MemoryStream对象生成原始雇员数据对象。

测试程序

编译CryptoDataSender.cs和CryptoDataRcvr.cs程序（使用SerialEmployee.dll文件）之后，可以在网络上测试它们。最后的结果应该是两个雇员记录的完整数据：

```
C:\>CryptoDataRcvr  
Waiting for a client...  
emp1.EmployeeID = 1  
emp1.LastName = Blum  
emp1.FirstName = Katie Jane  
emp1.YearsService = 12  
emp1.Salary = 35000.5  
  
emp2.EmployeeID = 2  
emp2.LastName = Blum  
emp2.FirstName = Jessica  
emp2.YearsService = 9  
emp2.Salary = 23700.3  
  
C:\>
```

如输出结果所示，发出的两个雇员数据对象被成功接收。然而，这不能证明加密确实起了作用，惟一的办法是监视数据在网络上的传输。如果是在网络中的不同设备上测试这些应用程序，可以使用WinDump或者Analyzer程序监视网络上的数据包。图17.3展示Analyzer程序中的一个数据包，可以看到数据包中的数据确实已经加密，保护数据不会被外界窥探。

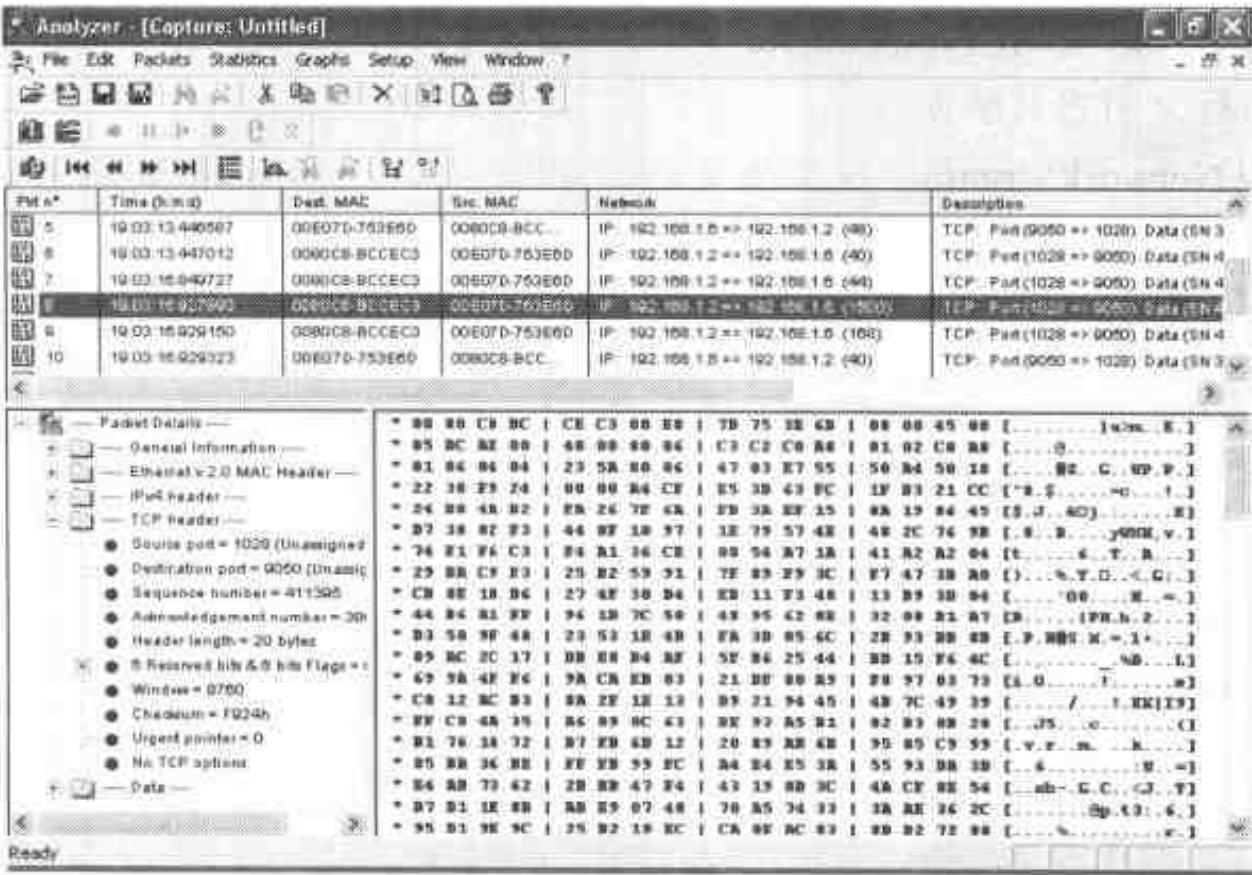


图17.3 在Analyzer程序中显示数据包

小结

数据的安全问题一直困扰着人们，所以需要在C#网络程序中并入安全特性。本章介绍了一些用起来很方便的.NET安全特性。

.NET CLR使用安全策略检查应用程序在系统中是否可以运行以及如何运行。每个应用程序是安全组的成员，安全组是基于应用程序特性组成的（例如应用程序的位置或者作者）。CLR系统使用安全权限集检查每个安全组对于本地系统资源的权限，例如文件和网络套接字。

可以使用.NET中的caspol程序检查计算机的安全设置，也可以用它来修改安全设置，赋予应用程序访问网络套接字的权限。通常，如果应用程序从远程文件服务器的共享磁盘运行，它们可能没有足够权限访问系统的网络套接字。要解决这个问题，可以向Everything权限集添加适当的组（通常是局域网）。

.NET网络库包含一些保护应用程序能访问到的网络套接字的类。SocketPermissionAttribute类允许在C#网络程序中定义属性命令，限制那些使用指定的套接字或者IP地址的用户。每个元数据命令定义一项在应用程序中实现的安全属性。如果安全属性指定的网络端口拒绝访问，端口被访问时应用程序就会抛出SecurityException异常。

通过网络向远程网络主机发送数据是有风险的。当数据在网络上传输时可能会遭到偷听。为保护数据，最好在通过网络发送数据之前使用加密技术。.NET库包含一些很好的加密类，帮助完成这一工作。

在网络程序中实现加密最简单的办法是使用对称加密类。对称加密允许数据加密为长度可变的数据块，而且可以结成链，形成流。加密数据流可以发送到任何类型的流，包括 `NetworkStream`。常常将数据存入 `MemoryStream` 并且从流数据中生成字节序列，这样可以很容易地操控数据。使用标准网络流技术，字节序列可以通过 `NetworkStream` 安全发送。当另一端收到数据时，必须使用与加密时使用的密钥相同的密钥将数据解密，还原为原始数据。

在本章中，讨论了很多网络问题。希望读者通过学习网络编程，已经可以编写自己的网络应用程序了。随着网络技术的日新月异，在网络编程领域总是出现新的技术需要学习。通过浏览网络新闻组以及持续关注最新的RFC，可以掌握网络发展的动向。

欢迎与我们联系

为了方便与我们联系，我们已开通了网站 (www.medias.com.cn)。您可以在本网站上了解我们的新书介绍，并可通过读者留言簿直接与我们沟通，欢迎您向我们提出您的想法和建议。也可以通过电话与我们联系，电话号码 (010) 68252397。

[G e n e r a l I n f o r m a t i o n]

书名= C # 网络应用编程

作者=

页数= 5 2 7

S S 号= 1 1 0 4 1 6 2 0

出版日期=

封面
书名
版权
前言
目录
正文