微软.NET程序员系列

Microsoft Press

- 欧美读者评价 ★★★★
- 微软网络专家更新再版
- 深入而详实的编程指导
- 丰富而全面的范例代码
- 覆盖传统网络编程接口
- 揭示Windows XP网络新特性

MICROSOFT

WINDOWS 网络编程(第2版)

Network Programming For Microsoft Windows, Second Edition

[美] **Anthony Jones, Jim Ohlund** 著

杨合庆 译

Visual Studio .NET产品组 审校

Microsoft .net

清华大学出版社

MICROSOFT

WINDOWS 网络编程(第2版)

Network Programming For Microsoft Windows, Second Edition

责任编辑：蔡　颖
封面设计：陈刘源

权威专家指导如何使用Windows XP的Winsock API和.NET套接字以及传统的Windows API 编写网络应用程序。

虽然为早期的Windows版本编写网络应用程序相对而言要容易，但是目前介绍Windows XP的网络特性的书籍还很少。作为更新的编程指南，本书着重于Windows XP中革新性的网络特性，同时介绍了为Visual C#编程语言提供的成功的网络支持。本书还介绍了最新的网际协议：IPv4、IPv6以及可靠IP多播协议。对于在网络技术方面需要明确实用的Microsoft网络API信息的开发者，或是寻求Microsoft网络操作的内部信息的管理员而言，这些内容都是较为理想的。如果您的编程或工作与当前的Microsoft Internet网络软件有密切的关系，本书也是很合适的学习和参考资料。

本书主题

- Winsock简介
- Winsock设计
- IPv4和IPv6网际协议
- Winsock支持的其他协议
- I/O方法
- 编写高性能可伸缩的应用程序
- 套接字选项和I/O控制命令
- 原始套接字
- Winsock服务提供程序接口
- Winsock Visual Basic编程
- 使用C#进行套接字编程
- RAS客户机
- NetBIOS
- Windows重定向器
- 邮槽
- 命名管道
- IP助手API

补充材料下载地址：http://www.wenyuan.com.cn /下载资源

阶段：分析商业需求 | 定义技术构架 | 设计解决方案 | 编码/实现 | 测试/调试 | 部署 | 支持/维护

读者服务邮箱：service@wenyuan.com.cn

作者介绍

本书的两位作者，Anthony Jones是微软核心Windows网络组的设计工程师，Jim Ohlund是微软网络和安全软件测试的一位领导工程师，他们在本书所讨论的知识领域内都是有深厚的理论和技术基础的专家。作为微软NetAPI开发支持小组的前任工程师和网络软件组的现任工程师，Anthony Jones和Jim Ohlund多年来一直在处理Windows网络的各种问题。

ISBN 7-302-05947-0
9 787302 059479 >
定价：78.00元

微软.NET 程序员系列

Windows 网络编程
(第 2 版)

[美] Anthony Jones 著
Jim Ohlund
杨合庆 译
Visual Studio .NET 产品组 审校

清华大学出版社

(京)新登字 158 号

内容简介

本书由权威专家编写，指导读者如何使用 Windows XP 的 Winsock API 和.NET 套接字以及传统的 Windows API 编写网络应用程序。作为更新的编程指南，本书着重于 Windows XP 中革新的联网特性，同时包含了对 C#编程语言的支持。本书还介绍了最新的网际协议：IPv4 和 IPv6，以及可靠 IP 多播协议。书中用大量的实例详细地描述了 Microsoft 网络 API 函数的应用，配套光盘也包含了所有的示例代码。对于在网络技术方面需要明确实用的 Microsoft 网络 API 信息的开发者，或是寻求 Microsoft 网络操作内部信息的管理员而言，这些内容都是较为理想的。对于在编程或工作中要用到当前的 Microsoft 或 Internet 联网软件的读者，本书也是很合适的学习和参考资料。

图书在版编目（CIP）数据

Windows 网络编程（第 2 版）/（美）琼斯，（美）欧伦德著；杨合庆译.—2 版.—北京：清华大学出版社，2002
（微软程序员系列）
书名原文：Network Programming for Microsoft Windows
ISBN 7-302-05947-0
Ⅰ.W… Ⅱ.①琼… ②欧… ③杨… Ⅲ.窗口软件，Windows—程序设计 Ⅳ.TP316.7

中国版本图书馆 CIP 数据核字（2002）第 076871 号

出 版 者：清华大学出版社(北京清华大学学研大厦，邮编 100084)
http://www.tup.tsinghua.edu.cn
责任编辑：蔡 颖
印 刷 者：世界知识印刷厂
发 行 者：新华书店总店北京发行所
开 本：787×960 1/16 **印张**：29.75 **彩插页**：4 **字数**：631 千字
版 次：2002 年 10 月第 1 版 2002 年 10 月第 1 次印刷
书 号：ISBN 7-302-05947-0/TP·3537
印 数：0001～4000
定 价：78.00 元

《微软.NET 程序员系列》序

自 2000 年 6 月微软宣布自己的.NET 战略以来，在不到两年的时间里，.NET 已经从战略变成现实。.NET 带来了全新的、快速而敏捷的企业计算能力，也给软件开发商和软件开发人员提供了支持未来计算的高效 Web 服务开发工具。作为微软.NET 战略的重要组成部分——Visual Studio .NET (中文版)已经于 2002 年 3 月 22 日正式在中国推出。

Visual Studio .NET 是一个功能强大、高效并且可扩展的编程环境。它充分展现了应用程序开发的潜能，并提供了生成应用程序所需的工具和技术。这些应用程序将给当今的企业、机构提供强大的支持，并推动下一代基于 XML Web 服务软件的发展。

有了 Visual Studio .NET，那些对全世界数百万的专业和业余程序员来说曾一度极端复杂、费时费力，甚至让人望而生畏的编程任务现在已不再神秘。更重要的是，Visual Studio .NET 使开发人员能运用既有的技能和知识来迎接新的编程挑战。

在 10 年前，Visual Basic 1.0 成为数以百万计的开发人员的革命性的应用程序开发语言。现在，Visual Studio .NET 为未来的 10 年做好了准备。

微软出版社为了配合 Visual Studio .NET 的推广以及.NET 技术的普及，邀请 Visual Studio .NET 项目开发组的核心开发人员和计算机图书专业作家精心编写了英文版《微软.NET 程序员系列》丛书；该丛书自面市以来，在美国图书销量排行榜上一直高居前列，颇受好评，成为程序开发人员和网络开发人员了解.NET 技术的权威工具书。尤其是《Microsoft .NET Framework 程序设计》一书，长期占据美国及欧洲此类书籍的排行榜冠军位置，程序开发人员不可不读此书。

清华大学出版社为了满足中国广大程序开发人员、网络开发人员学习最新技术的渴望，在微软出版社的配合下，从《微软.NET 程序员系列》这套丛书中精选了 40 本翻译成中文，以满足国内广大读者的需要。这套丛书阵容庞大，几乎涵盖了.NET 技术及其应用的各个方面；也正因为如此，翻译和编辑加工的工作量也大得惊人。但为了保持国外优秀技术图书的魅力，同时使读者领会新技术的真谛，本丛书的翻译和编辑都是经过严格筛选的、具有很高的翻译水平或丰富编辑经验的技术人员；另外，我们还聘请微软公司 Visual Studio .NET 产品组的技术专家审读每一本书，确保在技术上准确无误。

相信这套丛书定会帮助程序开发人员、网络开发人员以及那些具有一定编程基础的中、高级读者，快速、全面掌握.NET 技术，协助他们为技术生涯的下一个 10 年做好准备，为培养新一代软件人才，并推动中国软件产业的快速发展起到积极的作用！

这套丛书中的《C#技术内幕》和《C#编程技术》已于去年与读者见面，并得到读者的广泛好评。

目前，本从书中已出版和在编的共有 21 本，已从 6 月份起陆续和读者见面。这些书目包括：

- **《Microsoft .NET Framework 程序设计》**

  全面、详细地介绍了 Microsoft .NET Framework，可以帮助开发人员和设计人员轻松、高效地创建高性能且安全可靠的.NET 应用程序。

- **《应用程序升级——Visual Basic 6.0 到 Visual Basic .NET》**

  升级 Visual Basic 6.0 程序代码的最佳指导。涵盖了 Visual Basic .NET 的全部新功能和各种方案，这些方案能够以最小的中断进行代码的移植并维护混合环境。

- **《Visual C++ .NET 托管扩展程序设计》**

  Visual C++权威专家撰写。全面阐述了 Visual C++ .NET 托管扩展，讲解如何编写 .NET 库及应用程序。内容包括托管扩展的不同编程规则，以及 Visual C++等兼容.NET 的编程语言所具有的多种新特性等。

- **《Visual Basic .NET 实用标准》**

  包含针对面向对象编程、文件操作、解决方案分发及更多最佳经验，包括代码示例和将标准应用于 Visual Basic .NET 语法的建议。

- **《ASP .NET Web 应用程序开发新思维》**

  介绍了最新 Web 应用程序构建技术，带给您 Web 应用程序开发的新思想。帮助开发人员充分利用 Microsoft .NET，开发高效、安全的 Web 应用程序。

- **《构建 Web 解决方案——应用 ASP.NET 和 ADO.NET》**

  介绍 ADO.NET 与 ASP.NET 的综合应用，使开发人员可以将 Web 开发技术与数据库开发技术完美地结合起来，构建功能更加强大的 Web 应用程序和服务。

- **《Microsoft Windows 网络编程（第 2 版）》**

  主要讲述 Winsock 网络编程技术。详细介绍了如何编写高性能、可扩展的 Winsock 应用程序，还讲述了如何使用 C#开发 Winsock 程序。

- **《构建 XML Web 服务——基于 Microsoft .NET 平台》**

  详细深入地讲解了用于创建 XML Web 服务的主要协议和工具，并深入探讨了如何利用它们构建功能强大、高效的 Web 商业解决方案。还介绍了 Microsoft .NET My Services 以及其他基于 XML 的微软新技术。

- **《Microsoft .NET My Services 精解》**

  该书由 My Services 开发组的专家共同编写。详细解释了各项.NET 服务，并介绍了.NET My Services 消息界面模型、数据操纵语言、安全授权模型、系统文档结构以及管理模型，助您创建出充分利用.NET My Services 优点的 Web 服务。

- **《移动设备.NET 应用程序设计》**

  介绍了如何使用 ASP.NET、Visual Studio .NET 和移动通信 Internet 工具箱创建 Web 应用程序，在各种移动设备上以适当格式动态显示相同页面。有助于学习如何使用微软的信息服

务器，以及如何为移动设备提供 E-mail 访问。

- **《SQL Server 2000 与 Visual Basic .NET 编程》**

  介绍如何使用 SQL Server 2000、Visual Basic .NET、ADO.NET、ASP.NET、XML Web 服务和其他数据工具，开发数据库访问应用程序。

- **《SQL Server 2000 与 XML 编程》**

  本书内容是包含代码和指导的实用方法，介绍了如何为各行业、电子商务和 Web 创建强大的 XML 数据库应用程序。

- **《Microsoft .NET Server 企业解决方案》**

  介绍如何计划、开发和部署企业电子商务解决方案。将理论与实际紧密结合，详细讲解了如何解决电子商务集成时遇到的问题，并进一步探讨了使用 Microsoft .NET Server 的各种工具和技术解决具体的问题。

为了使读者的学习目标更为明确，在本丛书中，又划分出了两个子系列，现有 8 本书：

- **Core Reference 系列**

  目前共有 5 本：《Visual Basic .NET 核心编程》、《Visual C#核心编程》、《ADO.NET 核心编程》、《Microsoft .NET 核心编程》和《Visual J# .NET 核心编程》。已经列入我们的出版计划的还有另外两本：《Visual C++ .NET 核心编程》和《Microsoft .NET Framework 核心编程》。

  这些书分别介绍相应编程语言和.NET 框架类库的指令、示例代码、最佳编程惯例和基于案例的解决方案，内容详尽，讲解深入，集指导性与实用性于一体，最适合相关的开发人员用作专业读本。

- **Language Reference 系列**

  目前共有 3 本：《Visual Basic .NET 语言参考手册》、《Visual C#.NET 语言参考手册》和《Visual C++ .NET 语言参考手册》。以简洁、易于浏览和使用的形式，从 A 到 Z 列出了这 3 种编程语言的技术参考。

随着技术的发展，我们将根据读者的需要，不断增加新的书目。

丛书版式特色

本丛书在风格上力求文字精炼。并采用小 5 号字编排，内容紧凑，版面清晰美观，易于阅读。此外，书中还安排了一些特色段落，提供正文之外的一些细节知识：

注意：提醒阅读和操作过程中应注意的事项，避免出现错误或问题。

提示：指点一些操作捷径或实用技巧，使您少走弯路，阅读和操作更为高效。


要点：总结关键知识点或操作细节，助您适时掌握要领。


注：提示首次出现的编程元素，以及书中涉及到该元素的其他位置以供参考。

尽管我们倾心相注，精心而为，总有疏忽纰漏，恳请广大读者不吝赐教与指正，我们定会全力改进，以期在后续工作中得以完善。

本丛书在创作过程中得到了微软(中国)公司的大力支持。本丛书能够顺利出版，更是倾注了无数幕后人员的汗水和心力。在此，对他们的辛勤劳动一并表示衷心感谢！

编者

2002 年 6 月

前　言

欢迎阅读《Windows 网络编程(第 2 版)》。第 2 版不仅继承了第 1 版中相同的主题，而且增了许多新内容。本书主要讲述 Winsock 网络编程技术，还特别增加了两章，其中一章叙述编写高性能、可伸缩的 Winsock 应用程序；另一章叙述使用 C#编程语言进行 Winsock 编程，并用到了.NET 应用程序框架类库。另外，本书完全更新了讲述“Windows 服务提供程序接口”的那一章，介绍了附加的协议(如 IPv6 和可靠多播)，并揭示了 Windows XP 中的新功能。

本书覆盖大量的网络函数，它们适用于 Windows 95、Windows 98、Windows Me、Windows NT 4.0、Windows 2000、Windows XP 和 Windows CE。书中大部分内容涉及中级和高级网络主题，并重新整理了 Winsock 部分，使得本书更适合各个层次的程序设计人员阅读。

怎样使用本书

本书涉及下列 6 个技术领域：

- Winsock 应用程序编程接口(API)
- .NET 套接字(用 C#语言)
- Visual Basic Winsock 控件
- 客户端远程访问服务(RAS)
- IP 助手 API
- 传统网络——NetBIOS 和 Windows 重定向器

NetBIOS 和 Windows 重定向器技术在本书第 1 版发行后并未发生变化，为节省篇幅，相关章节(即第 17～22 章)作为补充材料放在我们的网站上(http://www.wenyuqn.com.cn/下载资源)，读者可根据需要自行下载。

第 2 版的大部分内容用来叙述 Winsock API。第 1 章一开始便介绍了 Winsock，对入门者而言这是很必要的。这一章涉及了所有基础知识，通过简单示例介绍了传输控制协议(TCP)和用户数据报协议(UDP)，并为后续章节的高级 Winsock 主题提供了一个粗略的指引。为简明起见，第 1 章仅叙述 IPv4 协议。

第 2 章讨论 Winsock 体系结构，如 Winsock 编录，以及 Winsock 如何适应总体网络堆栈。另外，这一章还叙述了怎样枚举 Winsock 编录，并找到系统中安装的每种协议的特性。

第 3 章介绍网际协议(IP)，其中介绍了 IPv4 和 IPv6，以及两者各自的寻址信息和名称解析。这一章的最后一个部分阐述了如何编写能在这两种协议之上无缝运行的应用程序。从 Winsock 中可以访问

的其余协议放在第 4 章叙述。这两章均展示了一些简单示例，来说明每种协议族。

第 5 章和第 6 章介绍了 Winsock 提供给 Winsock 高级程序员的各种输入/输出(I/O)模型。第 5 章给出了每种模型及其使用方法，第 6 章则详细阐述了如何编写高性能、可扩展的 Winsock 应用程序。第 6 章讨论了资源管理和不同 I/O 模型的优点，还讨论了性能指标。每种 I/O 模型都有相应的示例代码来阐释。

第 7 章叙述了从 Winsock 中可以访问的所有套接字选项和 I/O 控制命令，不仅包括常规 Winsock 选项，还包括各种协议专用的选项。针对每一种选项，都提供了成功访问该选项所需的预期输入输出参数。这一章的内容有所更新，包含了新协议(如 IPv6 和可靠多播)专用的选项。

第 8 章叙述 Winsock 注册和名称解析，介绍了可以执行查询的不同的命名空间，如域名系统(DNS)、服务声明协议(SAP)及 Active Directory 目录服务。这一章也讨论了新的“网络位置认识”(NLA)命名空间，它可以提供当前正连接到的网络的重要信息。

第 9 章的主题是多播。这一章阐述了 IPv4 多播和 IPv6 多播，还介绍了 Windows XP 中新引入的可靠多播传输。这一章也讨论了 ATM“点到多点”通信。第 10 章介绍服务质量(QOS)，这种技术能够为应用程序保证一部分网络带宽。第 11 章转到原始 IP 套接字，讨论如何建立协议头，并用它在 IP 网络(包括 IPv4 和 IPv6)之上直接通信。

第 12 章叙述 Winsock 服务提供程序接口(SPI)。通过这种接口，程序设计人员可以在 Winsock 和下层服务提供程序(如传输控制协议/网际协议，即 TCP/IP)之间安装一层，以便操纵套接字与协议行为，或名称注册和解析。这种功能使得软件开发者能够扩展 Winsock 的功能。SPI 一节彻底改写了一次，并提供了功能完全的、健壮的分层服务提供程序(LSP)示例代码。

第 13 章介绍.NET 应用程序框架的网络套接字对象。这一章介绍了如何从 C#语言访问新的套接字类。第 14 章讨论 Visual Basic Winsock 控件。之所以包括了这一章，是因为我们看到了有如此多的开发者信任 Visual Basic 和这个控件。这个控件在利用 Winsock 高级特性方面能力有限，但对于 Visual Basic 开发者而言，已经很有帮助了，毕竟他们需要的，只是简单易用的网络通信而已。

第 15 章介绍远程访问服务(RAS)客户端 API。由于 Internet、拨号通信和虚拟专用网(VPN)通信的流行，本书加入了关于 RAS 的一章。对程序设计人员而言，具有将拨号功能添加到网络应用程序的能力很有用，因为这可以让程序对用户而言更易于使用。也就是说，为了使用设计人员编写的网络应用程序，最终用户并不需要了解如何设置和建立拨号连接。

第 16 章讲述 IP 助手 API，用它可以获得有用的当前计算机网络配置信息。这一章介绍了一个新的函数，用它可以枚举 IPv6 特有的信息。

最后，第 1 版中叙述传统技术(NetBIOS、邮槽、命名管道及 Windows 重定向器)的相关章节、NetBIOS 命令以及 Winsock 错误代码参考，都作为本书的补充内容(第 17~22 章)放在我们的网站上(请

参阅封底的下载地址)。如果对英文文档的内容有不清楚的地方，可以参阅《Windows 网络编程(第 1 版)》中译本(机械工业出版社出版)中的相关章节。

我们有理由相信，本书是迄今为止最为全面的 Windows 网络编程指南，希望它能够成为您得力的学习工具和参考手册。

怎样使用补充材料

请访问 http://www.wenyuan.com.cn/下载资源，找到本丛书的相关链接，下载打包的补充材料。

本书每一章都展示了一系列示例代码，阐释如何运用书中介绍的网络 API。补充材料中也有这些示例代码。要安装它们，请将直接运行文件根目录下的 StartCD.exe。可以选择自启动菜单里的 Install Example Code 选项，安装示例代码，也可以从相应的文件夹直接访问每个示例(在 Samples\ChapterXX 目录下)。

注意 使用这些文件需要运行 32 位 Microsoft Windows 平台的计算机。

微软出版社支持信息

我们已经尽力保证本书及补充文件的内容准确无误。在 http://www.microsoft.com/mspress/support/ 上，微软出版社提供了对本书的更正。

本书展示的许多函数定义和表格都经过适当改编或翻印，这要感谢微软 Platform SDK 文档编制组的鼎力相助和许可。有些材料基于早期文档，因此可能会发生变化。要想了解最新的 Platform SDK 信息，或最近的更新和错误修正，请访问 MSDN 网址：http://www.microsoft.com/msdownload/platformsdk/sdkupdate/。

如果对本书或配书文件有任何建议、问题或想法，请通过下面的电子邮件发送到微软出版社：

MSPInput@Microsoft.com

或邮寄至：

Microsoft Press

Attn: *Network Programming for Microsoft Windows, Second Edition* Editor

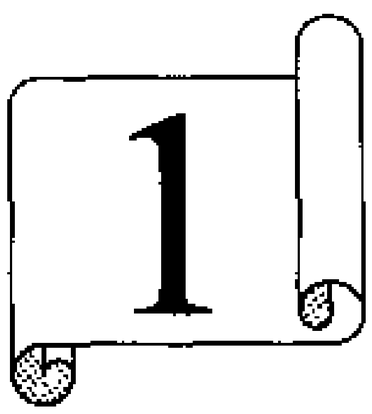
One Microsoft Way

Redmond, WA 98052-6399

请注意，上述地址并不提供软件产品的支持。

此外，也可向本书中文版编辑室发送电子邮件，以反馈意见。邮件地址请见本书封底上的“读者服务邮箱”。

目　录



Winsock 简介

本章专门讲解编写成功的 Winsock 应用程序的基本方法。Winsock 是一种标准 API(application programming interface，应用程序编程接口)，主要用于网络中的数据通信，它允许两个或者多个应用程序(或进程)在同一台机器上或通过网络相互通信。有一点我们必须明白：Winsock 是一种网络编程接口，而不是协议。使用 Winsock 编程接口，应用程序可通过普通网络协议如 TCP/IP(Transmission Control Protocol/Internet Protocol，传输控制协议/网际协议)或 IPX(Internet Packet Exchange, Internet 数据包交换)协议建立通信。Winsock 接口从在 UNIX 平台上实现的 BSD Socket(套接字)中继承了大量的特性。在 Windows 环境中，这种接口演变成一种真正独立于协议的接口，新发布的 Winsock 2 版本更是如此。

本章将讨论从网络上的一台机器到另一台机器建立通信的基本知识，以及如何收发数据。为了便于大家理解接受连接、建立连接和收发数据所需的 Winsock 调用，本章给出了多个示例。由于本章的目的是学习这些基本的 Winsock 调用，因而所举的示例均采用了直接阻塞的 Winsock 调用。第 5 章将讲述 Winsock 支持的非阻塞调用及其他各种 I/O 方法，其中包含示例代码。

除此以外，本章还将介绍各种 API 函数的 Winsock 1 和 Winsock 2 版本。通过前缀 WSA 可以区分该函数的两种版本。若 Winsock 2 在其规范中更新或增添了一个新的 API 函数，该函数名将带有 WSA 前缀。比如，建立套接字的 Winsock 1 函数只是被简单称为 socket，而 Winsock 2 引入该函数的新版本时，则将它命名为 WSASocket，该函数可以使用 Winsock 2 中出现的一些新特性。但请注意：这个命名规则有几个例外，如 WSAStartup、WSACleanup、WSARecvEx 及 WSAGetLastError 都属于 Winsock 1.1 规范的函数。

在使用 Winsock 开发应用程序前，必须了解创建应用程序时需要哪些文件和库。

1.1 Winsock 头文件及库文件

如前所述，Winsock 有两个主要版本，即 Winsock 1 和 Winsock 2，两者都能在除 Windows CE 之

外(Windows CE 只支持 Winsock 1)的所有 Windows 平台上运行。开发新的应用程序时，把 WINSOCK2.H 文件包含在应用程序中，该程序将使用 Winsock 2 规范。为了和其他旧的 Winsock 应用程序兼容以及保证 Windows CE 平台上的程序开发，可以使用 WINSOCK.H。另外，还有一个头文件 MSWSOCK.H，该头文件用于微软专用编程扩展，这些扩展通常用于高效 Winsock 应用程序的开发，第 6 章中将对此加以描述。

在编译采用了 WINSOCK2.H 的应用程序时，须链接到 WS2_32.LIB 库。使用 WINSOCK.H(比如在 Windows CE 中)时须使用 WSOCK32.LIB。如果从 MSWSOCK.H 中使用扩展 API，还必须链接 MSWSOCK.DLL。一旦包含了必需的头文件和链接环境，就可以开始编写应用程序代码了，这时需要初始化 Winsock。

1.2 Winsock 的初始化

每个 Winsock 应用都必须加载合适的 Winsock DLL 版本。如果调用一个 Winsock 函数之前没有加载 Winsock 库，这个函数就会返回一个 SOCKET_ERROR，错误信息是 WSANOTINITIALISED。加载 Winsock 库是通过调用 WSAStartup 函数实现的。这个函数的定义如下：

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

wVersionRequested 参数用于指定准备加载的 Winsock 库的版本。高位字节指定所需 Winsock 库的次版本，而低位字节则是主版本。可以使用宏 MAKEWORD(x,y)(其中，x 是高位字节，y 是低位字节)来方便地获得 wVersionRequested 的正确值。

lpWSAData 参数是指向 LPWSADATA 结构的指针，WSAStartup 用与其加载的库版本有关的信息填充这个结构：

```
typedef struct WSAData
{
    WORD                wVersion;
    WORD                wHighVersion;
    char                szDescription[WSADESCRIPTION_LEN +1];
    char                szSystemStatus[WSASYS_STATUS_LEN +1];
    unsigned short      iMaxSockets;
    unsigned short      iMaxUdpDg;
    char FAR*           lpVendorInfo;
}WSADATA,*LPWSADATA;
```

WSAStartup 将第一个字段 wVersion 设置为将要使用的 Winsock 版本。wHighVersion 参数包含了

现有 Winsock 库的最高版本。记住，这两个字段中，高位字节代表的是 Winsock 次版本，而低位字节代表的则是 Winsock 主版本。szDescription 和 szSystemStatus 这两个字段由特定的 Winsock 来实现和设置，它们并没有实际作用。不要使用下面这两个字段：iMaxSockets 和 iMaxUdpDg，它们分别表示可以同时打开的最大套接字数量，以及数据报的最大长度。通过 WSAEnumProtocols(请参阅第 2 章)查询协议信息，才能知道数据报的最大长度；可以同时打开的最大套接字数量不是固定的，它的值在很大程度上取决于可用的物理资源。最后的 lpVendorInfo 是一个保留字段，用于存储实现 Winsock 的特定供应商的信息。所有的 Windows 平台都没有使用这个字段。

表 1.1 列出了微软各种 Windows 平台支持的 Winsock 版本。必须注意这两个版本之间的差别，Winsock 1.x 不支持本书描述的很多高级 Winsock 特性。

表 1.1　各种 Windows 平台支持的 Winsock 版本

平　台	Winsock 版本
Windows 95	1.1(2.2)
Windows 98	2.2
Windows Me	2.2
Windows NT 4.0	2.2
Windows 2000	2.2
Windows XP	2.2
Windows CE	1.1

注意，即使某个平台支持 Winsock 2，也可以不使用这个 Winsock 的最新版本。也就是说，如果打算编写大多数平台均能支持的应用程序，应根据 Winsock 1.1 规范来编写。因为所有的 Winsock 1.1 调用都通过 Winsock 2 DLL 映射，所以所编程序可以在 Windows NT 4.0 平台上正确无误地运行。同时，如果市面上出现了您所用平台可以使用的 Winsock 库更新版本，那么您很可能要进行版本升级。因为这些新版本中包含了对错误的修正，所以从理论上来说，旧代码完全可以正常运行。在有些情况下，Winsock 堆栈的行为和规范中的定义有所不同，这样，许多程序员是根据特定目标平台的行为来编程，而不是根据规范来编写程序。

但是，在很多情况下，编写新应用程序时，程序员都想加载已发布的 Winsock 库的最新版本。当然，如果 Winsock 3 发布了，加载 2.2 版本的应用程序仍可一如既往地运行。但如果用户要求的 Winsock 版本比平台所能支持的版本新，WSAStartup 就会失败。如果 WSAStartup 正确返回，WSADATA 结构中的 wHighVersion 就是当前系统中的 Windsock 库能够支持的最新版本。

在使用 Winsock 接口编写好应用程序之后，应该调用 WSACleanup 函数。这个函数能够使 Winsock

释放所有由 Winsock 分配的资源，并取消这个应用程序挂起的 Winsock 调用。WSACleanup 函数的定义为：

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

因为操作系统将会自动释放资源，所以在退出应用程序时也可以不调用 WSACleanup 函数；然而如果这样做，您的应用程序就不再符合 Winsock 规范了。另外，每次调用 WSAStartup 后都应该调用 WSACleanup。

1.3 错误检查和处理

要想成功编写 Winsock 应用程序，检查和处理错误是至关重要的，所以这里首先对此进行介绍。事实上，对 Winsock 函数来说，返回错误是很常见的。但是，在有些情况下，这些错误是无关紧要的，通信仍可在那个套接字上进行。Winsock 调用失败时最常见的返回值是 SOCKET_ERROR。本书在详细介绍各个 API 调用时，都将指出和各种错误对应的返回值。SOCKET_ERROR 常量实际上是-1。如果调用 Winsock 函数时出现了错误，可以用 WSAGetLastError 函数来获得一段代码，这段代码专用来说明错误。该函数的定义如下：

```
int WSAGetLastError(void);
```

发生错误之后调用这个函数，返回的是所发生的错误的整数代码。WSAGetLastError 函数返回的这些错误代码都有已经预先定义的常量值；因 Winsock 版本的不同，这些值的声明可能在 WINSOCK1.H 中，也可能在 WINSOCK2.H 中。两个头文件的惟一差别是 WINSOCK2.H 中包含更多针对 Winsock 2 中引入的一些新 API 函数和功能的错误代码。为各种错误代码定义的常量(带有#define 指令)一般都以 WSAE 开头。相对于 WSAGetLastError 函数，另一个与此相关的函数是 WSASetLastError，使用该函数可以手动设置 WSAGetLastError 获取的错误代码。

下述程序演示了如何基于上述内容构建一个 Winsock 应用程序框架：

```
#include<winsock2.h>

void main(void)
{
   WSADATA wsaData;

   //初始化Winsock版本2.2

   if ((Ret = WSAStartup(MAKEWORD(2,2),&wsaData))!= 0)
   {
       //注意：因为Winsock没有加载，所以我们不能使用WSAGetLastError来确定导致故障的特定错
       //误。但我们将根据WSAStartup的返回状态进行判断
```

```
        printf("WSAStartup failed with error %d \n",Ret);
        return;
    }

    //当应用程序结束调用WSACleanup之后，设置Winsock通信代码

    if (WSACleanup() == SOCKET_ERROR)
    {
    printf("WSACleanup failed with error %d \n",WSAGetLastError());
    }
}
```

下面开始叙述如何使用网络协议建立通信。

1.4　协 议 寻 址

随着 Internet 的不断普及，IP 协议随处可得，当今大多数 Winsock 应用程序开发都使用 IP，因此，为了简便和避免重复，本章其余部分将只描述怎样使用 IP(Internet Protocol，网际协议)协议创建基本的 Winsock 调用来建立通信。前面已提到，Winsock 是一种独立于协议的接口，第 4 章将讲述使用其他协议(如 IPX)的具体内容。另外，本章对 IP 的讨论仅限于简单描述 IPv4(IP 第 4 版)。第 3 章将全面而详细地讲述所有 IP 版本，包括 IPv4 和 IPv6(IP 第 6 版)。

本章的其余部分将介绍使用 IPv4 协议建立 Winsock 通信的基本知识。IP 在大多数计算机操作系统中都能得到支持，并可在多数局域网(LAN)上使用，如办公室中的小型网络，也可在广域网(WAN)中使用，例如 Internet。从设计角度看，IP 是一种无连接协议，它不能确保数据传输的成功。两个高级协议——TCP(Transmission Control Protocol，传输控制协议)和 UDP(User Datagram Protocol，用户数据报协议) ——用通过 IP 进行面向连接和无连接的数据通信，这方面的内容将在以后讲述。TCP 和 UDP 都使用 IP 进行数据传输，通常被称作 TCP/IP 和 UDP/IP。如果需要在 Winsock 中使用 IPv4，需要知道怎样为 IPv4 寻址。

IPv4 寻址

在 IPv4 中，计算机都分配有一个地址，该地址用一个 32 位的数值来表示。客户机需要通过 TCP 或 UDP 和服务器通信时，必须指定服务器的 IP 地址和服务端口号。另外，服务器打算监听传入的客户机请求时，也必须指定一个 IP 地址和一个端口号。在 Winsock 中，应用程序通过 SOCKADDR_IN 结构来指定 IP 地址和服务端口信息，该结构的格式如下：

```
struct sockaddr_in
{
    short           sin_family;
```

```
    u_short        sin_port;
    struct in_addr     sin_addr;
    char           sin_zero[8];
};
```

sin_family 字段必须设为 AF_INET，以告知 Winsock 此时正在使用 IP 地址族。

用标识服务器服务的 TCP 或 UDP 通信端口由 sin_port 字段定义。因为有些可用端口号是为“已知的”服务保留的，如 FTP(文件传输协议，File Transfer Protocol)和 HTTP(Hypertext Transfer Protocol，超文本传输协议)，所以应用程序在选择端口时，必须特别小心。第 2 章中有更多关于选择端口的详细内容。

SOCKADDR_IN 结构的 sin_addr 字段把 IPv4 地址作为一个 4 字节的量存储起来，它是无符号长整数的数据类型。根据这个字段的不同用法，它还可表示一个本地或远程 IP 地址。IP 地址一般是用“Internet 标准点分表示法”像 a.b.c.d 一样指定的，其中每个字母代表一个字节的数字(用十进制、八进制或十六进制格式表示)，从左到右分配了一个无符号长整数的 4 个字节。最后一个字段 sin_zero，只充当填充项，以使 SOCKADDR_IN 结构和 SOCKADDR 结构的长度一样。

inet_addr 是一个很实用的支持函数，它可把一个点分 IP 地址转换成一个 32 位的无符号长整数。它的定义如下：

```
unsigned long inet_addr(
    const char FAR *cp
);
```

cp 字段是一个空终止字符串，用于接受点分表示法的 IP 地址。注意，这个函数把 IP 地址当作一个按网络字节顺序排列的 32 位无符号长整数返回(网络字节顺序在下面的“字节排序”小节中有简要说明)。

字节排序

不同的计算机处理器采用 big-endian 和 little-endian 形式进行编号，具体采用哪种表示方法，由各自的设计决定。比如，Intel86 处理器上，多字节编号用 little-endian 形式来表示：字节的排序是从最无意义的字节到最有意义的字节。在计算机中把 IP 地址和端口号指定成多字节数时，这个数就按**主机字节**(host-byte)顺序来表示。但是，如果在网络上指定 IP 地址和端口号，“Internet 联网标准”指定多字节值必须用 big-endian 形式来表示(从最有意义的字节到最无意义的字节)，一般称之为**网络字节**(network-byte)顺序。

有一系列函数可用于多字节数的转换，把后者从主机字节顺序转换成网络字节顺序，或进行反方向的转换。下面 4 个 API 函数使将一个数从主机字节顺序转换成网络字节顺序：

```
u_long htonl(u_long hostlong);
```

```
int WSAHtonl(
    SOCKET s,
    u_long hostlong,
    u_long FAR * lpnetlong
);

u_short htons(u_short hostshort);

int WSAHtons(
    SOCKET s,
    u_short hostshort,
    u_short FAR * lpnetshort
);
```

htonl 和 WSAHtonl 的 hostlong 参数是按主机字节顺序排序的一个 4 字节数。htonl 函数返回的数是网络字节顺序，而 WSAHtonl 函数通过 lpnetlong 参数返回的数也按网络字节顺序排列。htons 和 WSAHtons 的 hostshort 参数是按主机字节顺序排列的一个 2 字节数。htons 函数把这个数当作按网络字节顺序排列的一个 2 字节数值返回，而 WSAHtons 函数则通过 lpnetshort 参数返回这个数。

下面这 4 个函数是前面 4 个函数的逆向函数，它们把网络字节顺序转换成主机字节顺序：

```
u_long ntohl(u_long netlong);

int WSANtohl(
    SOCKET s,
    u_long netlong,
    u_long FAR * lphostlong
);

u_short ntohs(u_short netshort);

int WSANtohs(
    SOCKET s,
    u_short netshort,
    u_short FAR * lphostshort
);
```

现在，演示一下如何利用上面描述的 inet_addr 和 htons 函数来创建 SOCKADDR_IN 结构，并进行 IPv4 寻址。

```
SOCKADDR_IN InternetAddr;
INT nPortId = 5150;

InternetAddr.sin_family = AF_INET;
//将准备使用的点分 Internet 地址 136.149.3.29 转换为 4 字节整数，并把它分配给 sin_addr
```

```
InternetAddr.sin_addr.s_addr = inet_addr("136.149.3.29");
//nPortId 变量按存储主机字节顺序排列。将 nPortId 转换为网络字节顺序，并分配给 sin_port

InternetAddr.sin_port = htons(nPortId);
```

IP 地址不便于记忆，因此，大多数人更喜欢使用一个容易记忆且容易掌握的主机名。第 3 章将叙述一些有用的地址和名称解析函数，使用它们可以将主机名(如 www.somewebsite.com)解析为 IP 地址、服务名称(如 FTP)或端口号。这些函数有 getaddrinfo、getnameinfo、gethostbyaddr、gethostbyname、gethostname、getprotobyname、getprotobynumber、getservbyname 以及 getservbyport 等。同时还有这些函数的异步版本，如：WSAAsyncGetHostByAddr、WSAAsyncGetHostByName、WSAAsyncGetProtoByName、WSAAsyncGetProtoByNumber、WSAAsyncGetServByName 以及 WSAAsyncGetServByPort 等。

现在有了协议寻址的基础，诸如 IPv4，就可以准备通过创建套接字来建立通信了。

1.5 创建套接字

熟悉 Winsock 的人应该知道，API 是建立在套接字概念基础上的。套接字是传输提供程序的句柄。在 Windows 中，套接字和文件描述符不是一回事，因而是一个独立的类型，即 WINSOCK2.H 中的 SOCKET 类型。有两个函数可以用来创建套接字：socket 和 WSASocket。随后的 3 章将详细讲述如何创建每种可用协议的套接字。为简便起见，对套接字将仅作简要叙述：

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

第 1 个参数 af 是协议的地址族。由于本章仅使用 IPv4 来描述 Winsock，因此应将这个字段设为 AF_INET。第 2 个参数 type 是协议的套接字类型。如果使用 TCP/IP 创建套接字，应将该字段设为 SOCK_STREAM，而用 UDP/IP 时则应设为 SOCK_DGRAM。第 3 个参数是 protocol，用于在给定地址族和套接字类型具有多重入口时，对具体的传送作限定。对于 TCP，应将该字段设为 IPPROTO_TCP；而对于 UDP 则设为 IPPROTO_UDP。第 2 章将详细讲述如何创建所有协议的套接字，包括 WSASocket API 在内。

为控制各种套接字选项和套接字行为，Winsock 提供了 4 个有用的函数：setsockopt、getsockopt、ioctlsocket 及 WSAIoctl。在简单的 Winsock 编程中，没有必要特别使用这些函数。第 7 章将描述这些函数及其所有可用选项。在成功地创建了套接字之后，就可以开始在套接字上建立通信，并为收发数据做好准备。在 Winsock 中有两种基本的通信技术：面向连接的通信和无连接的通信。

1.6　面向连接的通信

本节讨论接受连接和建立连接所需要的 Winsock 函数。首先讨论的是如何通过监听客户机连接来开发服务器，并探讨接受或拒绝一个连接的过程。随后讨论的是怎样通过初始化与服务器的连接来开发客户机。最后讨论数据在面向连接会话中是如何传输的。

在 IP 中，面向连接的通信是通过 TCP/IP 协议完成的。TCP 提供两个计算机间可靠无误的数据传输。应用程序使用 TCP 通信时，在源计算机和目标计算机之间，会建立起一个虚拟连接。建立连接之后，计算机之间便能以双向字节流的方式进行数据交换。

1.6.1　服务器 API 函数

这里所说的服务器其实是一个进程，它需要等待任意数量的客户机与之建立连接，以便为它们的请求提供服务。服务器必须在一个已知的名称上监听连接。在 TCP/IP 中，这个名称就是本地接口的 IP 地址，再加上一个端口编号。每种协议都有一套不同的寻址方案，所以各自的命名方法也不同。在 Winsock 中，第一步是用 Socket 或 WSASocker 将给定协议的套接字绑定到它已知的名称上，这个过程是通过 bind API 调用来完成的。下一步是将套接字置为监听模式，这一步用 API 函数 listen 来完成的。最后，若一台客户机试图建立连接，服务器必须通过 accept 或 WSAAccept 调用来接受连接。接下来的几个小节将讨论绑定、监听和接受客户机连接所需的每个 API 调用。图 1.1 展示了建立通信信道时，服务器和客户机必须执行的基本调用。

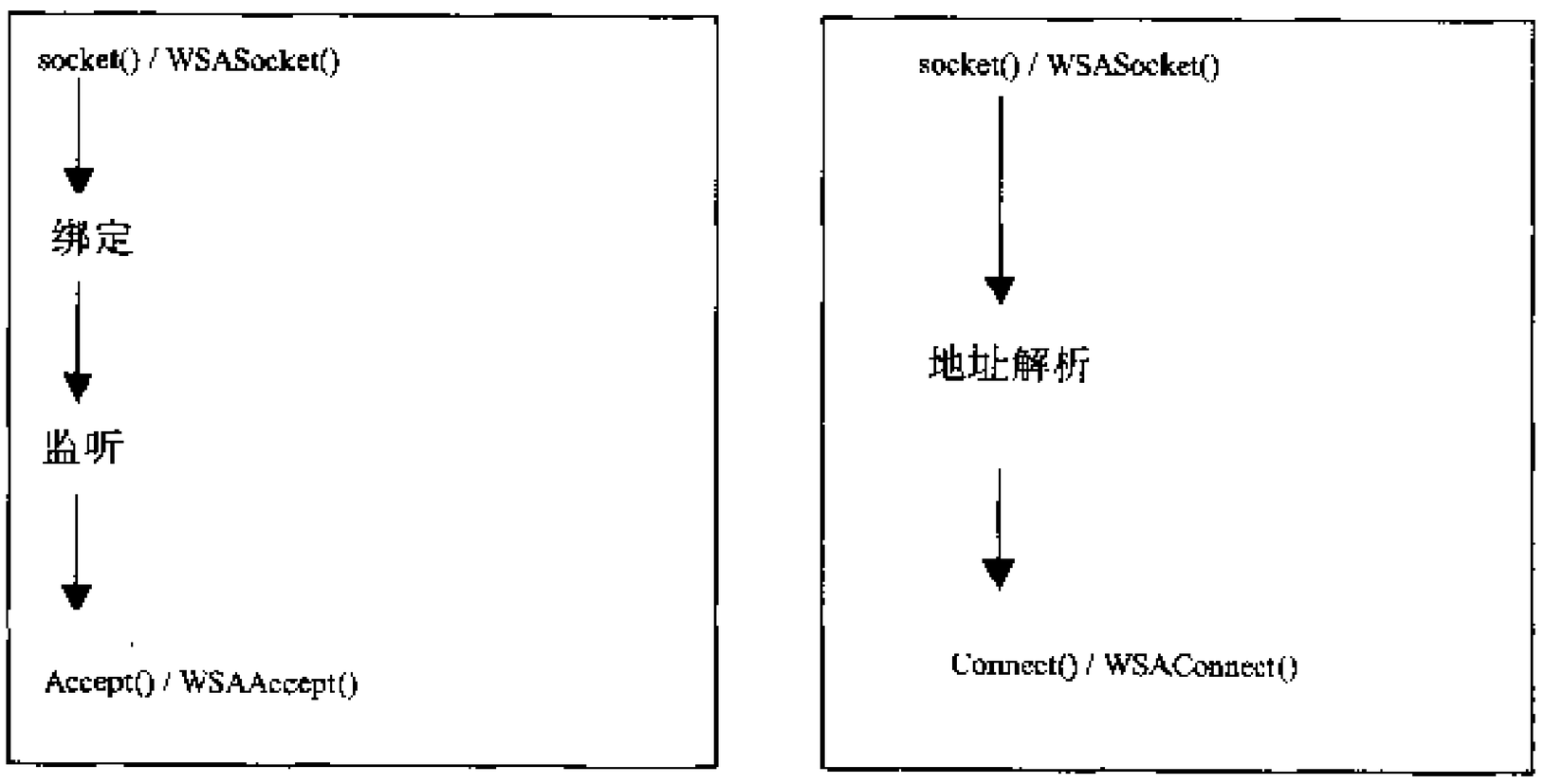


图 1.1 服务器和客户机的 Winsock 基础

1.6.1.1　绑定

一旦为某种协议创建了套接字，就必须将套接字绑定到一个已知地址上。bind 函数可将指定的套

接字同一个已知地址绑定到一起。该函数的声明如下：

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

其中，第 1 个参数 s 代表用来等待客户机连接的那个套接字。第 2 个参数的类型是 struct sockaddr，它的作用很简单，就是一个普通的缓冲区。根据所使用的那个协议，必须实际地填充一个地址缓冲区，并在调用 bind 时将其转换为一个 struct sockaddr。Winsock 头文件将 SOCKADDR 类型定义为 struct sockaddr。为简化起见，本章都将使用这个类型。第 3 个参数代表要传递的、由协议决定的地址结构的长度。举个例子来说，下列代码显示了在一个 TCP 连接上，如何来做到这一点：

```
SOCKET          s;
SOCKADDR_IN     tcpaddr;
int             port = 5150;

s = socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);

tcpaddr.sin_family = AF_INET;
tcpaddr.sin_port = htons(port);
tcpaddr.sin_addr.s_addr = htonl(INADDR_ANY);

bind(s,(SOCKADDR *)&tcpaddr,sizeof(tcpaddr));
```

可以看到这个例子创建了一个流套接字。随后，建立了 TCP/IP 地址结构，并打算在它上面接受客户机连接。在这种情况下，通过特殊 IP 地址 INADDR_ANY，套接字被绑定到默认的 IP 接口，并占据 5150 端口。可以指定系统中一个可用的显式 IP 地址，不过 INADDR_ANY 允许将套接字绑定到系统中所有可用的接口，以便将来传到任意接口上的客户机连接(必须在正确的端口上)都可以被监听套接字接受。bind 调用正式将套接字同 IP 接口及端口关联到了一起。

一旦出错，bind 就会返回 SOCKET_ERROR。对 bind 而言，最常见的错误是 WSAEADDRINUSE。如果使用的是 TCP/IP，那么 WSAEADDRINUSE 就表示另一个进程已经同本地 IP 接口及端口号绑定到了一起，或者那个 IP 接口和端口号处于 TIME_WAIT 状态。假如对一个已被绑定的套接字调用 bind，返回的将是 WSAEFAULT 错误。

1.6.1.2 监听

接下来要做的，是将套接字置入监听模式。bind 函数的作用只是将套接字和指定的地址关联在一起。指示套接字等候连接传入的 API 函数则是 listen，其定义如下：

```
int listen(
    SOCKET s,
```

```
    int     backlog
);
```

第 1 个参数同样是一个被绑定的套接字。backlog 参数指定了被搁置的连接的最大队列长度。因为完全可能同时出现几个服务器的连接请求，所以这个参数非常重要。例如，假定 backlog 参数为 2。如果有 3 个客户机同时发出请求，那么头两个会被放在一个“挂起”队列中，以便应用程序依次为它们提供服务。而第 3 个连接请求会失败，返回一个 WSAECONNREFUSED 错误。注意，一旦服务器接受了一个连接，那个连接请求就会从队列中删去，以便别人可继续发出请求。backlog 参数其实本身就存在着限制，这个限制是由下层的协议提供程序决定的。如果这个参数出现非法值，那么系统会用与之最接近的一个合法值来取代。另外，对于如何找出实际的 backlog 值，还不存在一种标准的方案。

与 listen 相关的错误是非常直观的。到目前为止，最常见的错误是 WSAEINVAL。该错误通常意味着，在调用 listen 之前没有调用 bind。另外，与 bind 调用相反，使用 listen 时可能接收到 WSAEADDRINUSE 错误。这个错误通常是在进行 bind 调用时发生的。

1.6.1.3　接受连接

现在，我们已做好了接受客户机连接的准备。这是通过 accept、WSAAccept 或 AcceptEx 函数来完成的(AcceptEx 是 accept 的扩展版本，第 6 章中对它作了详细描述)。Accept 的原型如下：

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

其中，参数 s 是一个被绑定的套接字，它处于监听模式。第 2 个参数应该是一个有效的 SOCKADDR_IN 结构的地址，而 addrlen 应该是 SOCKADDR_IN 结构的长度。对于属于另一种协议的套接字，应当用与那种协议对应的 SOCKADDR 结构来替换 SOCKADDR_IN。通过对 accpet 函数的调用，可以为被搁置的连接队列中的第 1 个连接请求提供服务。accpet 函数返回后，addr 结构中会包含发出连接请求的那个客户机的 IPv4 地址信息，而 addrlen 参数则指出 addr 结构的长度。此外，accpet 会返回一个新的套接字描述符，它对应于已经被接受的那个客户机连接。该客户机后续的所有操作都应该使用这个新的套接字。至于原来那个监听套接字，它仍然用于接受其他客户机连接，而且仍处于监听模式。

如果出错，INVALID_SOCKET 将被返回。在监听套接字为异步或非阻塞模式，并且没有连接被接受时，最常见的错误是 WSAEWOULDBLOCK。阻塞、非阻塞以及其它套接字模式将在第 5 章中作介绍。Winsock 2 引入了 WSAAccept 函数，根据条件函数的返回值，该函数可有条件地接受一个连接。第 10 章将详细地讲述 WSAAccept 函数。

到此为止，已经介绍过了创建简单的 Winsock TCP/IP 服务器应用程序所需的全部元素。下面的程序片断展示了如何编写一个能够接受 TCP/IP 连接的服务器。程序中没有对调用实施任何错误检查，这样可以使代码显得清晰易读。本应用程序的完整版本可在补充材料中的 TCPSERVER 文件中找到。

```
#include <winsock2.h>
void main(void)
{
    WSADATA        wsaData;
    SOCKET         ListeningSocket;
    SOCKET         NewConnection;
    SOCKADDR_IN    ServerAddr;
    SOCKADDR_IN    ClientAddr;
int Port = 5150;
//初始化 Winsock 版本 2.2

WSAStartup(MAKEWORD(2,2),&wsaData);

    //创建一个新的套接字来监听客户机连接

    ListeningSocket = socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);
    //建立一个 SOCKADDR_IN 结构，这个结构将告知 bind 我们想要在 5150 端口监听所有接口上
    //的连接
    //请留意这里是如何将端口变量从主机字节顺序转换为网络字节顺序的

    ServerAddr.sin_family = AF_INET;
    ServerAddr.sin_port = htons(Port);
    ServerAddr.sin_addr.s_addr = htonl(INADDR_ANY);

    //使用 bind 将这个地址信息和套接字关联起来
    bind(ListeningSocket,(SOCKADDR *)&ServerAddr,
    sizeof(ServerAddr));

    //监听客户机连接。这里使用 5 个 backlog，许多应用程序一般都使用这个数量

    listen(ListeningSocket,5);

    //连接到达时，接受一个新连接

    NewConnection = accept(ListeningSocket,(SOCKADDR *)
                        &ClientAddr,&ClientAddrLen));

    //此刻在这些套接字上可以做两件事。一是在 ListeningSocket 上再次调用 accept，
    //等候更多的连接到来。二是开始在 NewConnection 上收发数据。本章稍后将讲述怎
    //样进行数据收发
    //在 NewConnection 套接字上完成数据收发，以及在 ListeningSocket 套接字上
```

```
        //完成接受新连接后，应该用 closesocket API 关闭这些套接字
        //本章稍后将讲述套接字的关闭
        closesocket(NewConnection);
        closesocket(ListeningSocket);

        //应用程序完成对连接的处理后，调用 WSACleanup

        WSACleanup();
}
```

在了解到怎样创建一个能够接收客户机连接的服务器之后，下面接着讲述如何创建客户机。

1.6.2　客户端 API 函数

客户机的创建要简单得多，建立成功连接所需的步骤也要少得多。创建客户机只需 3 步操作：

1. 创建一个套接字。
2. 建立一个 SOCKADDR 地址结构，结构名称为准备连接到的服务器名(以下层协议为准)。对于 TCP/IP，这是客户机应用程序所监听的服务器的 IP 地址和端口号。
3. 用 connect 或 WSAConnect 初始化客户机与服务器的连接。

由于已经知道如何建立套接字和建立 SOCKADDR 结构，所以现在只有一步未做，那就是建立一个连接。

TCP 状态

作为一名 Winsock 程序员，通常没必要了解实际的 TCP 状态。但了解了 TCP 状态，就能更好地理解 Winsock API 调用如何对下层协议中的变化产生影响。此外，许多程序员在关闭套接字时，会碰到一个常见问题，那就是围绕套接字关闭时的 TCP 状态，这是我们目前最感兴趣的问题。

对每个套接字来说，它的初始状态都是 CLOSED。若客户机初始化了一个连接，就会向服务器发送一个 SYN 包，同时将客户机套接字状态置为 SYN_SENT。服务器收到 SYN 包后，会发出一个 SYN-ACK 包，客户机需要用一个 ACK 包对它做出响应。此时，客户机的套接字将处于 ESTABLISHED 状态。如果服务器一直不发送 SYN-ACK 包，客户机就会超时，并返回 CLOSED 状态。

若服务器的套接字同本地接口及端口绑定起来，并在它上面进行监听，那么套接字的状态便是 LISTEN。客户机试图与服务器连接时，服务器就会收到一个 SYN 包，并用一个 SYN-ACK 包做出回应。服务器套接字的状态就变成 SYN_RCVD。最后，客户机发出一个 ACK 包，它将使服务器套接字的状态变成 ESTABLISHED。

一旦应用程序处于 ESTABLISHED 状态，就可以通过两种方法来关闭它。如果由应用程序来关闭，便叫作“主动套接字关闭”；否则，套接字的关闭便是被动的。图 1.2 对两种关闭方法进行了解释。如主动关闭，应用程序便会发出一个 FIN 包。应用程序调用 closesocket 或 shutdown 时(把 SD_SEND 当作第 2 个参数)，会向通信对方发出一个 FIN 包，而且套接字的状态将变成 FIN_WAIT_1。正常情

况下，通信对方会用一个 ACK 包作为回应，套接字的状态随之变成 FIN_WAIT_2。如通信对方也关闭了连接，它会发出一个 FIN 包，我们的机器则会以一个 ACK 包作为回应，并将套接字的状态置为 TIME_WAIT。

TIME_WAIT 状态也叫作 2MSL 等待状态。其中，MSL 代表“分段最长生存时间”(Maximum Segment Lifetime)，表示一个数据包在被丢弃之前，可在网络上存在多长时间。每个 IP 包都含有一个 TTL(time-to-live，生存时间)字段，若它递减到 0，包便会被丢弃。一个包经过网络上的每个路由器时，TTL 值都会减 1，然后继续传递。一旦应用程序进入 TIME_WAIT 状态，那么就会一直持续两倍 MSL 的时间。这样，如果最后一个 ACK 包丢失，TCP 就可以重新发送它，也就是说，FIN 会被重新传送出去。两倍于 MSL 时间的等待状态结束之后，套接字便进入 CLOSED 状态。

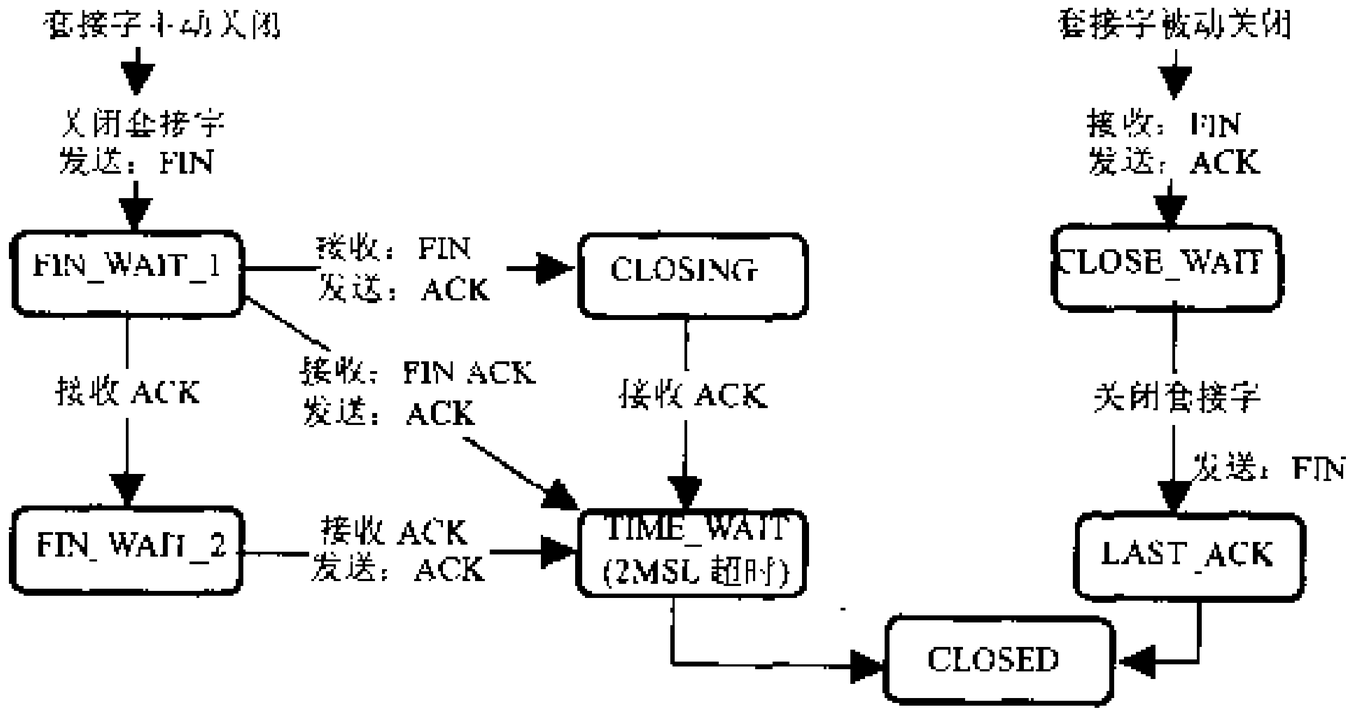


图 1.2 TCP 套接字的关闭状态

采取主动关闭措施时，沿两条路径可以进入 TIME_WAIT 状态。在以前的讨论中，只有一方发出一个 FIN，并接收一个 ACK 回应。然而，另一方仍然可以自由地发送数据，直到它也被关闭为止。因此，需要另外两条路径发挥作用。在一条路径中(即同步关闭)，一台计算机与之连接的通信对方会同时要求关闭：计算机向对方送出一个 FIN 数据包，并从它那里接收一个 FIN 数据包。随后，计算机会发出一个 ACK 数据包，对对方的 FIN 包作出回应，并将自己的套接字置为 CLOSING 状态。计算机从对方那里接收到最后一个 ACK 包之后，它的套接字状态会变成 TIME_WAIT。

主动关闭时，另一条路径其实就是同步关闭的变体：套接字从 FIN_WAIT_1 状态直接变成 TIME_WAIT 状态。若应用程序发出一个 FIN 数据包，但马上又从对方那里接收到一个 FIN-ACK 包时，这种情况就会发生。在这种情况下，对方会确认是否收到了应用程序的 FIN 包，并送出自己的 FIN 包。对于这个包，应用程序会以一个 ACK 包做出回应。

TIME_WAIT 状态的主要作用是，在 TCP 连接处于 2MSL 等待状态的时候，定义那个连接的一对套接字不可以被重新使用。这对套接字由本地 IP 端口以及远程 IP 端口组成。某些 TCP 实施方案不允许重新使用处于 TIME_WAIT 状态下的套接字对中的任何端口号。在微软实现的 TCP 中，不会存在这个问题。然而，若试图通过一对已处于 TIME_WAIT 状态的套接字建立连接，连接的建立就会失败，并返回 WSAEADDRINUSE 错误。要解决这一问题(除了等待使用本地端口的套接字对脱离

TIME_WAIT 状态之外)，一个办法是使用套接字选项 SO_REFUSEADDR，第 7 章将对这个选项进行详细讨论。

在被动关闭情况下，应用程序会从对方那里接收到一个 FIN 包，并用一个 ACK 包做出回应。此时，应用程序的套接字会变成 CLOSE_WAIT 状态。由于对方已关闭自己的终端，所以它不能再发送数据了。但应用程序却不同，它能一直发送数据，直到它关闭了自己的连接终端为止。要想连接终端，应用程序需要发出自己的 FIN，令应用程序的套接字状态变成 LAST_ACK。应用程序从对方接收到一个 ACK 包后，它的套接字就会回到 CLOSED 状态。

要想了解 TCP/IP 协议的有关详情，请访问 http://www.rfc-editor.org，查阅 RFC793 文件。

connect 函数

连接套接字是通过调用 connect、WSAConnect 或 ConnectEx 函数来完成的。先来看看 connect 函数的 Winsock 1 版本，其定义如下：

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

这个函数的参数含义是相当明显的：参数 s 是即将在其上面建立连接的那个有效 TCP 套接字；name 参数是 TCP 的套接字地址结构(SOCKADDR_IN)，表示要连接到的服务器；namelen 则是 name 参数的长度。

如果想连接的计算机没有进程用于监听给定端口，connect 调用就会失败，并产生 WSAECONNREFUSED。另一个错误可能是 WSAETIMEDOUT，这种情况一般发生在试图连接的计算机不可用时(也可能因为到主机之间的路由上出现硬件故障，或主机目前没有联网)。

下面的程序片断展示了如何编写一个能够连接到前述服务器应用程序的简单客户机。程序中没有对调用实施任何错误检查，以使代码显得清晰易读。本应用程序的完整版本可在补充材料中名为 TCPCLIENT 的文件中找到。

```
#include <winsock2.h>

void main(void)
{
    WSADATA        wsaData;
    SOCKET         s;
    SOCKADDR_IN    ServerAddr;
    int            Port = 5150;

    //初始化 Winsock 2.2 版本
```

```
    WSAStartup(MAKEWORD(2,2),&wsaData);

    //创建一个新的套接字来建立客户机连接

    s = socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);

    //建立一个 SOCKADDR_IN 结构，用它来连接到在 5150 端口的监听服务器
    //作为示范，这里假设我们的服务器 IP 地址是 136.149.3.29
    //显然，应该提示用户输入 IP 地址，并将用户数据填入这个字段

    ServerAddr.sin_family = AF_INET;
    ServerAddr.sin_port = htons(Port);
    ServerAddr.sin_addr.s_addr = inet_addr("136.149.3.29");

    //用套接字 s 创建一个到服务器的连接

    connect(s,(SOCKADDR *)&ServerAddr,sizeof(ServerAddr));

    //此时可以在套接字 s 上收发数据了。本章稍后将叙述怎
    //样进行数据收发
    //在套接字 s 上结束数据收发后，应该用 closesocket API 来关闭它
    //本章稍后将叙述套接字的关闭

    closesocket(s);

    //应用程序完成对连接的处理后，调用 WSACleanup

    WSACleanup();
}
```

既然已经能够为面向连接的服务器和客户机建立通信了，接下来便开始处理数据传输。

1.6.3 数据传输

收发数据是网络编程的主题。要在已建立连接的套接字上发送数据，可用这两个 API 函数：send 和 WSASend。第 2 个函数是 Winsock 2 中特有的。同样地，在已建立了连接的套接字上接收数据也有两个函数：recv 和 WSARecv。后者也是 Winsock 2 函数。必须牢牢记住这一点：所有关系到收发数据的缓冲区都属于简单的 char 类型，即面向字节的数据。事实上，它可能是一个包含任何原始数据的缓冲区，至于这个原始数据是二进制数据，还是字符型数据，则无关紧要。

另外，所有收发函数返回的错误代码都是 SOCKET_ERROR。一旦有错误返回，系统就会调用 WSAGetLastError 获得详细的错误信息。最常见的错误是 WSAECONNABORTED 和 WSAECONNRESET。两者均涉及到连接正在被关闭这一问题——要么由于超时被关闭，要么由于通

信方正在关闭连接。另一个常见错误是 WSAEWOULDBLOCK，一般出现在套接字处于非阻塞模式或异步状态时。这个错误意味着指定函数暂时不能完成。第 5 章将详细说明各种 WinsockI/O 方法，以避免出现此类错误。

1.6.3.1　send 和 WSASend

要在已建立连接的套接字上发送数据，第 1 个可用的 API 函数是 send，其原型为：

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

SOCKET 参数是已建立了连接、将用于发送数据的套接字。第 2 个参数 buf 则是指向字符缓冲区的指针，该缓冲区中包含即将发送的数据。第 3 个参数 len，指定即将发送的缓冲区内的字符数。最后，flags 可为 0、MSG_DONTROUTE 或 MSG_OOB。另外，flags 还可以是对这些标志进行按位“或”运算的一个结果。MSG_DONTROUTE 标志要求传输层不要将它发出的数据包路由出去。是否实现这一请求由下层的传输来决定(例如，若传输协议不支持该选项，这一请求就会被忽略)。MSG_OOB 标志表示数据应该进行带外发送。

在顺利的情况下 send 将返回发送的字节数；若发生错误，就返回 SOCKET_ERROR。常见的错误是 WSAECONNABORTED，这一错误一般在虚拟回路由于超时或协议有错而中断的时候发生。发生这种情况时，因为这个套接字不能再用了，所以应该将它关闭。当远程主机上的应用程序通过执行强行关闭或意外中断操作重新设置虚拟回路时，或远程主机重新启动时，发生的则是 WSAECONNRESET 错误。再次需要注意的是，发生这一错误时，应该关闭这个套接字。最后一个常见错误是 WSAETIMEOUT，它在连接由于网络故障或远程连接系统异常死机而导致连接中断时发生。

Send API 函数的 Winsock 2 版本是 WSASend，它的定义如下：

```
int WSASend(
    SOCKET s,
    LPWSABUF lpBuffers,
    DWORD dwBufferCount,
    LPDWORD lpNumberOfBytesSent,
    DWORD dwFlags,
    LPWSAOVERLAPPED lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);
```

这个套接字是一个连接会话的有效句柄。第 2 个参数是指向一个或多个 WSABUF 结构的指针。它既可是一个独立的结构，又可以是一个结构数组。第 3 个参数指明准备传递的 WSABUF 结构数量。

记住，每个 WSABUF 结构本身就代表了一个字符缓冲及其长度。为何打算同时发送多个缓冲呢？也许大家不太明白其中的原因。这就是稍后要讲到的散播—聚集 I/O 模式；但是，在一个已建立连接的套接字上利用多个缓冲区来发送数据时，每个缓冲区的发送顺序都是从数组中的第一个到最后一个 WSABUF 结构。lpNumberOfBytesSent 参数是指向 DWORD(是 WSASend 调用返回)的指针，其中包含已发送的字节总数。dwFlags 参数相当于它在 send 中的等效参数。最后两个参数——lpOverlapped 和 lpCompletionROUTINE——用于重叠 I/O。重叠 I/O 是 Winsock 支持的异步 I/O 模式的一种。关于重叠 I/O 在第 5 章将进行详细讲解。

WSASend 函数把 lpNumberOfBytesSent 设为写入的字节数。执行成功时，该函数就返回 0，否则就返回 SOCK_ERROR，该函数的常见错误通常与 send 函数一样。必须注意这最后一个发送函数：WSASendDisconnect。

1.6.3.2 WSASendDisconnect

这个函数非常特殊，一般不用。其原型是：

```
int WSASendDisconnect (
    SOCKET s,
    LPWSABUF lpOutboundDisconnectData
);
```

带外数据

对已建立连接的流套接字上的应用程序来说，如果需要发送的数据比流上的普通数据重要得多，便可将这些重要数据标记成 OOB(Out-of-band，带外)数据。位于连接另一端的应用程序可通过一个独立的逻辑信道(从理论上讲，该逻辑信道与数据流无关)来接收和处理 OOB 数据。

在 TCP 中，OOB 数据由一个紧急的 1 位标记(叫作 URG)和 TCP 分段头中的一个 16 位的指针组成。这里的标记和指针把特定的下行流字节当作紧急数据。实现紧急数据的两种特殊方法目前只能在 TCP.RFC793 中见到。该索引对 TCP 进行了描述，并引入了“紧急数据”这一概念，指明 TCP 头中的紧急指针是紧急数据字节之后那个字节的绝对偏移。在 RFC1122 中，紧急偏移被描述成指向紧急字节本身。

Winsock 规范中，独立于协议的 OOB 数据和 TCP 的 OOB 数据(紧急数据)实现均采用了 OOB 这一术语。要查看被搁置的数据中是否包含紧急数据，必须通过 SIOCATMARK 选项调用 ioctlsocket 函数。第 7 章将介绍 SIOCATMARK 的用法。

Winsock 提供了获得紧急数据的几种方法。可以内联紧急数据，使其出现在普通数据流中；也可以禁用内联，这样，对接收函数的不连续调用就只返回紧急数据。至于控制 OOB 数据行为的套接字选项 SO_OOBINLINE，本书也将在第 7 章详细讨论。

Telnet 和 Rlogin 使用紧急数据是有原因的，尽管如此，除非计划编写自己的 Telnet 和 Rlogin，否则就应该避开紧急数据。原因在于，它定义得不完善，而且它在其他平台上的实施情况可能和 Windows 有所不同。如果在紧急的情况下需要一种方法来发信号通知通信方，可以为紧急数据执行独立的控制

套接字，并为普通数据的传输保留主要的套接字连接。

函数 WSASendDisconnect 起初将套接字置为关闭状态，并发送断开的数据。当然，它只能用于支持从容关机和断开数据的传输协议。目前还没有传输提供程序支持断开数据。WSASendDisconnect 函数的行为和利用 SD_SEND 参数调用 shutdown 函数(后面将讲到该函数)差不多，但它另外还要发送包含在 boundDisconnectData 参数中的数据。之后的发送禁止在这个套接字上进行。如果调用失败，WSASendDisconnection 就会返回 SOCKET_ERROR。使用该函数可能会出现 send 函数中出现的某些错误。

1.6.3.3　recv 和 WSARecv

对于在已建立连接的套接字上接受数据的传入来说，recv 函数是最基本的方式。它的定义如下：

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

第 1 个参数 s，是准备用来接收数据的那个套接字。第 2 个参数 buf，是用于接收数据的字符缓冲区，而 len 则是准备接收的字节数或 buf 缓冲区的长度。最后的 flags 参数可以是下面的值：0、MSG_PEEK 或 MSG_OOB。另外，还可对这些标志中的每一个进行按位“或”运算。当然，0 表示没有特殊的行为。MSG_PEEK 表示要将可用的数据复制到所提供的接收端缓冲区内，但是并不从系统缓冲区中将这些数据删除。待发字节数也将被返回。

消息查看功能有一些缺点。它不仅导致性能下降(因为需要进行两次系统调用，一次是查看，另一次是无 MSG_PEEK 标志的真正删除数据的调用)，在某些情况下还可能不可靠。返回的数据可能没有反映出真正可用的总量。与此同时，把数据留在系统缓冲区内，可容纳传入数据的系统空间就会越来越少。结果使得系统减少各发送端的 TCP 窗口容量。这样，应用程序就不能获得最大的数据吞吐率。因此，最好是尽量把所有数据都复制到自己的缓冲区中，并在那里操作数据。

在基于消息或基于数据报的套接字(例如 UDP)上使用 recv 时，有几点应该注意。当挂起数据大于所提供的缓冲区时，缓冲区会尽量地让数据填满。这时，recv 调用会产生 WSAEMSGSIZE 错误。注意，消息大小的错误是在使用面向消息的协议时发生的。而流协议(如 TCP)则把传入的数据缓存下来，并尽量地返回应用程序所要求的数据，即使被挂起的数据量比缓冲大。因此，对流传输协议来说，就不会碰到 WSAEMSGSIZE 这个错误。

WSARecv 函数在 recv 的基础上增加了一些新特性。比如说重叠 I/O 和部分数据报通知。WSARecv 的定义如下：

```
int WSARecv(
```

```
    SOCKET s,
    LPWSABUF lpBuffers,
    DWORD dwBufferCount,
    LPDWORD lpNumberOfBytesRecvd,
    LPDWORD lpFlags,
    LPWSAOVERLAPPED lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);
```

参数 s 是已建立连接的套接字。第 2 和第 3 个参数是用来接收数据的缓冲。lpBuffers 参数是一个由 WSABUF 结构组成的数组，而 dwBufferCount 则表明这个数组中 WSABUF 结构的数量。如果接收操作立即完成，lpNumberOfBytesRceived 参数就会指向这个函数调用所收到的字节数。lpFlags 参数可以是下面任何一个值：MSG_PEEK、MSG_OOB、MSG_PARTIAL，或者是对这些值进行按位“或”运算之后的结果。MSG_PARTIAL 标志如果使用和出现的地方不同，其含义也不同。对支持部分消息的面向消息的协议(如 Apple Talk)来说，这个标志是在 WSARecv 调用返回后设置的(如果因为缓冲区空间不够，导致整条消息未能在这次调用中返回)。这时，后面的 WSARecv 调用都会设置这个标志，直到整条消息返回，才把这个 MASG_PARTIAL 标志清除。如果把这个标志当作一个输入参数传递，则接收操作应该一收到可用数据就结束，即使它收到的只是整条消息中的一部分。MSG_PARTIAL 标志只随面向消息的协议一起使用。另外，不是所有的协议都支持部分消息。每个协议的协议条目都包含一个标志，表明是否支持这一特性。有关详情参见第 2 章。lpOverlapped 和 lpCompletionROUTINE 参数用于重叠 I/O 的操作，第 5 章将对此进行详细讨论。此外，还必须注意另外一个特殊的接收函数 WSARecvDisconnect。

1.6.3.4 WSARecvDisconnect

这个函数与 WSASendDisconnect 函数相对应，其定义如下：

```
int WSARecvDisconnect(
    SOCKET s,
    LPWSABUF lpInboundDisconnectData
);
```

和 WSASendDisconnect 函数的参数一样，该函数的参数也是已建立连接的套接字句柄，以及一个有效的 WSABUF 结构(带有接收到的数据)。接收到的数据可以只是断开数据。这个断开数据是另一端的 WSASendDisconnect 调用发出的，它不能用于接收普通数据。另外，一旦收到数据，WSARecvDisconnect 函数就会取消接收远程通信方的数据，其作用和用 SD_RECEIVE 调用 shutdown 函数(稍后讲述)时相同。

1.6.4 流协议

由于大多面向连接的协议(如 TCP)同时也是流传输协议，所以，在此对流协议作一些介绍。在流

协议中，发送者和接收者可以将数据分解成小块数据，或将数据合并为大块数据。对于流套接字上收发数据所用的函数，需要了解的是：它们不能保证要求进行读取或写入的数据量。比如说，用 send 函数来发送一个有 2 048 字节的字符缓冲区，采用的代码是：

```
char sendbuff[2048];
int  nBytes = 2048;

//用 2 048 字节的数据填充 sendbuff
//假定 s 是一个有效的、已连接的流套接字

ret = send(s,sendbuff,nBytes,0);
```

对 send 函数而言，可能会返回已发出的少于 2 048 的字节。因为对每个收发数据的套接字来说，系统都为它们分配了相当充足的缓冲区空间，所以 ret 变量将被设为已发送的字节数。在发送数据时，内部缓冲区会将数据一直保留到可以将它发到线上为止。几种常见的情况都可导致这一情形的发生。比方说，传输大量的数据可以令缓冲区快速填满。同时，对 TCP/IP 来说，还有一个窗口大小的问题。接收端会对窗口大小进行调节，以指示它可以接收多少数据。如果有大量数据涌入接收端，接收端就会将窗口大小设为 0，为挂起的数据做好准备。对发送端来说，这样会强制它在收到一个新的大于 0 的窗口大小之前，不得再发送数据。在使用 send 调用时，缓冲区可能只能容纳 1024 个字节，这时，便有必要重新提交剩下的 1 024 个字节。要保证将所有的字节发出去，可采用下面的代码。

```
char sendbuff[2048];
int  nBytes = 2048,
nLeft,
idx;

//用 2048 字节的数据填充 sendbuff
//假定 s 是一个有效的、已连接的流套接字

nLeft = nBytes;
idx = 0;

while (nLeft >0)
{
    ret = send(s,&sendbuff[idx],nLeft,0);
    if (ret == SOCKET_ERROR)
    {
    //出错
    }
    nLeft -= ret;
    idx += ret;
}
```

在流套接字上接收数据时，这一原则照样适用，但意义不大。因为流套接字是一个不间断的数据

流，在读取它时，应用程序通常不会关心应该读多少数据。如果应用程序需要通过流协议获取离散消息，您需要做一些额外的工作。如果所有消息长度都一样，则处理起来比较简单，比如说，需要读取的 512 个字节的消息看起来就像下面这样：

```
char recvbuff[1024];
int  ret,
     nLeft,
     idx;

nLeft = 512;
idx = 0;

while (nLeft >0)
{
    ret = recv(s,&recvbuff[idx],nLeft,0);
    if (ret == SOCKET ERROR)
    {
    //出错
    }
    idx += ret;
    nLeft -= ret;
}
```

如果消息长度不同，处理起来就会麻烦一点。因此，有必要利用自己的协议来通知接收端，让它知道即将到来的消息长度是多少。比方说，写入接收端的前 4 个字节总是整数，大小为即将到来的消息的字节数。接收端每次开始读取时，会先查看前 4 个字节，把它们转换成一个整数，并查看构成消息的字节数是多少。

散播—聚集 I/O

散播—聚集(scatter-gather)支持是 Berkeley Socket 中首次随 Recv 和 Writev 这两个函数一起出现的概念。它随 WSARecv、WSARecvFrom、WSASend 和 WSASendTo 这几个 Winsock 2 函数一起使用。对那些收发特殊格式数据的应用程序来说，这个功能是非常有用的。比方说，客户机发到服务器的消息可能一直都是这样构成的：一个指定某种操作的固定的 32 字节的头，一个 64 字节的数据块和一个 16 字节的尾。这时，就可用一个由 3 个 WSABUF 结构组成的数组调用 WSASend，这 3 个结构分别对应 3 种消息类型。在接收端，则用 3 个 WSABUF 结构来调用 WSARecv，各个结构包含的数据缓冲区分别是 32 字节、64 字节和 16 字节。

在使用基于流的套接字时，散播—聚集 I/O 模式只是把 WSABUF 结构中提供的数据缓冲区当作一个连续的缓冲区。另外，接收调用可能在所有缓冲区填满之前就返回。在基于消息的套接字上，每次对接收操作的调用都会收到一条消息，其长度由所提供的缓冲区大小决定。如果缓冲空间不够，调用就会失败，并出现 WSAEMSGSIZE 错误，为了适应可用的缓冲空间，数据就会被截断。当然，如果使用支持部分消息的协议，就可用 MSG_PARTIAL 标志来避免数据的丢失。

1.6.5　中断连接

一旦完成了套接字连接，就必须将它关掉，并释放关联到那个套接字句柄的所有资源。要真正地释放与一个打开的套接字句柄关联的资源，执行 closesocket 调用即可。但要明白这一点，closesocket 可能会带来负面影响(和如何调用它有关)，即可能会导致数据的丢失。鉴于此，应该在调用 closesocket 函数之前，利用 shutdown 函数从容地终止连接。接下来讨论这两个 API 函数。

1.6.5.1　shutdown

为了保证通信方能够收到应用程序发出的所有数据，对一个好的应用程序来说，应该通知接收端“不再发送数据”。同样，通信对方也应该如此。这就是所谓的”正常关闭”方法，由 shutdown 函数来执行。shutdown 的定义如下：

```
int shutdown(
    SOCKET s,
    int    how
);
```

how 参数可以是后面的任何一个值：SD_RECEIVE、SD_SEND 或 SD_BOTH。如果是 SD_RECEIVE，就表示不允许再调用接收函数。这对底部的协议层没有影响。另外，对 TCP 套接字来说，不管数据在是等候接收，还是将随后抵达，都要重新设置连接。尽管如此，在 UDP 套接字上，仍然接受并排列传入的数据(因为对于无连接协议而言，shutdown 毫无意义)。如果选择 SE_SEND，表示不允许再调用发送函数。对 TCP 套接字来说，这样会在所有数据发出，并得到接收端确认之后，生成一个 FIN 包。最后，如果指定 SD_BOTH，则表示取消连接两端的收发操作。

应注意到，并非所有的面向连接协议都支持从容关闭，这是 shutdown API 执行的功能。对那些不支持从容关闭的协议来说(如 ATM)，仅调用 closesocket 便可结束进程。

1.6.5.2　closesocket

closesocket 函数用于关闭套接字，它的定义如下：

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

对 closesocket 的调用会释放套接字描述符，然后再利用套接字执行的调用就会失败，并出现 WSAENOTSOCK 错误。如果没有对该套接字的其他引用，那么所有与套接字描述符关联的资源都会被释放。其中包括丢弃所有队列中的数据。

对这个进程中任何一个线程来说，它们发出的被挂起的同步调用都会在未投递任何通知消息的情况下被删除。被挂起的的重叠操作也会被删除。与该重叠操作关联的任何事件、完成例程或完成端口能被执行，但最后都会失败，并出现 WSA_OPERATION_ABORTED 错误。套接字 I/O 模式在第 5 章中有详细讲解。另外，还有一点会对 closesocket 的行为产生影响：套接字选项 SO_LINGER 是否已经

设置。要了解这一点，请参考第 7 章中对 SO_LINGER 选项的描述。

1.7 无连接通信

和面向连接的协议比较起来，无连接协议的行为有很大不同，因此，收发数据的方法也会有所差别。和面向连接的服务器比较起来，无连接接收端改动不大，所以先讨论接收端(如果愿意，也可称之为服务器)。接下来再谈发送端。

在 IP 中，无连接通信是通过 UD/IP 协议完成的。UDP 不能确保可靠的数据传输，但能将数据发送到多个目标，或者接收的多个源数据。例如，如果一个客户机向一个服务器发送数据，则数据将立刻被传输，不管服务器是否准备接收这些数据。如果服务器接收来自客户机的数据，该服务器也不会发消息确认数据已收到。数据的传输使用数据报，即离散信息包。

1.7.1 接收端

对于在一个无连接套接字上接收数据的进程来说，操作过程并不复杂。先用 socket 或 WSASocket 创建套接字。再把这个套接字和准备接收数据的接口绑定在一起，这是通过 bind 函数(和面向会话的示例一样)来完成的。和面向连接不同的是，不必调用 listen 和 accept。相反，只需等待接收数据。由于没有连接，始发于网络上任何一台机器的数据报都可被接收端的套接字接收。最简单的接收函数是 recvfrom。它的定义如下：

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

前面 4 个参数和 recv 的参数是一样的，其中包括标志的可能值：MSG_OOB 和 MSG_PEEK。在使用无连接套接字时，和前面一样，仍然应该慎用 MSG_PEEK 标志。对监听套接字的给定协议来说，from 参数是一个 SOCKADDR 结构，带有指向地址结构长度的 fromlen。这个 API 调用的返回数据时，SOCKADDR 结构内便填入了发送数据的那个工作站的地址。

recvfrom 函数的 Winsock 2 版本是 WSARecvFrom。后者的原型是：

```
int WSARecvFrom(
    SOCKET s,
    LPWSABUF lpBuffers,
    DWORD dwBufferCount,
```

```
    LPDWORD lpNumberOfBytesRecvd,
    LPDWORD lpFlags,
    struct sockaddr FAR * lpFrom,
    LPINT lpFromlen,
    LPWSAOVERLAPPED lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);
```

两者的差别在于接收数据时 WSABUF 结构的用法。可以利用 dwBufferCount 为 WSARecvFrom 提供一个或多个 WSABUF 缓冲区。提供多个缓冲区后，就可以用散播—聚集 I/O 了。读取的字节总数放在 lpNumberOfBytesRecvd 中返回。在调用 WSARecvFrom 时，lpFlags 参数可以是 0(代表无选项)、MSG_OOB、MSG_PEEK 或 MSG_PARTIAL。这些标志还可以一起按位“或”运算。如果在调用这个函数时，指定了 MSG_PARTIAL，提供程序就知道返回数据，即使只收到了部分消息。当调用返回之后，如果只收到部分消息，应用程序就会设置 MSG_PARTIAL 标志。另外，调用返回后，WSARecvFrom 就会把 lpFrom 参数(它是一个指向 SOCKADDR 结构的指针)设置为发送端的地址。LpFromLen 同样指向 SOCKADDR 结构的长度，不过，在这个函数中，它还是一个指针，指向 DWORD。最后两个参数，lpOverlapped 和 lpCompletionROUTINE，用于重叠 I/O(第 5 章将就此展开讨论)。

在无连接套接字上接收(发送)数据的另一种方法是建立连接。听起来有些奇怪吧，但事实的确如此。无连接的套接字一旦建立，便可利用 SOCKADDR 参数(它被设为准备与之通信的远程接收端地址)调用 connect 或 WSAConnect。但事实上并没有建立真正的连接。传递到连接函数的套接字地址是与套接字关联在一起的，如此一来，才能够用 Recv 和 WSARecv 来代替 recvfrom 和 WSARecvFrom。为什么呢？其原因是数据的始发处是已知的。如果在应用程序中，一次只和一个端点进行通信，便能很容易地与数据报套接字建立连接。

下面的示例代码展示了如何创建一个简单的 UDP 接收端应用程序。本应用程序的完整版本可在补充材料中名为 UDPRECEIVER 的文件中找到。

```
#include <winsock2.h>
void main(void)
{
    WSADATA                 wsaData;
    SOCKET                  ReceivingSocket;
    SOCKADDR_IN             ReceiverAddr;
    int                         Port = 5150;
    char                    ReceiveBuf[1024];
    int                         BufLength = 1024;
    SOCKADDR_IN             SenderAddr;
    int                         SenderAddrSize = sizeof(SenderAddr);
    //初始化 Winsock 2.2 版本
    WSAStartup(MAKEWORD(2,2),&wsaData);
```

```
    //创建一个新的套接字来接收数据报
    ReceivingSocket = socket(AF_INET,SOCK_DGRAM,IPPROTO_UDP);
    //建立一个 SOCKADDR_IN 结构，这个结构将告知 bind 我们想要使用 5150 端口接收来自所
     //有接口的数据报
     ReceiverAddr.sin_family = AF_INET;
     ReceiverAddr.sin_port = htons(Port);
     ReceiverAddr.sin_addr.s_addr = htonl(INADDR_ANY);
     //使用 bind 将这个地址信息和套接字关联起来
     bind(ReceivingSocket,(SOCKADDR *)&SenderAddr,sizeof(SenderAddr));

    //此时可以在绑定套接字上接收数据报了
    recvfrom(ReceivingSocket,ReceiveBuf,BufLength,0,
             (SOCKADDR *)&SenderAddr,&SenderAddrSize);
        //应用程序结束接收数据报后，关闭套接字
        closesocket(ReceivingSocket);
        //应用程序结束后，调用 WSACleanup
        WSACleanup();
    }
```

在了解到如何创建一个能够接收数据报的接收端之后，下面叙述怎样创建发送端。

1.7.2 发送端

要在一个无连接的套接字上发送数据，有两种选择。最简单的一种，便是建立一个套接字，然后调用 sendto 或 WSASendTo。先讲解 sendto 函数，它的定义是这样的：

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

除了 buf 是发送数据的缓冲区，len 指明发送多少字节外，其余参数和 recvfrom 的参数一样。另外，to 参数是一个指向 SOCKADDR 结构的指针，该结构带有接收数据的那个工作站的目标地址。另外，也可以用 Winsock 2 函数 WSASendTo。它的定义如下：

```
int WSASendTo(
    SOCKET s,
    LPWSABUF lpBuffers,
    DWORD dwBufferCount,
    LPDWORD lpNumberOfBytesSent,
```

```
    DWORD dwFlags,
    const struct sockaddr FAR * lpTo,
    int iToLen,
    LPWSAOVERLAPPED lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);
```

同样，WSASendTo 和前一版本中的 sendto 函数类似。它把指针当作 lpBuffers 参数，该指针指向带有发给接收端数据的结构，而 dwBufferCount 参数则指明当前有多少个结构。可发送多个 WSABUF 结构来启用散播—聚集 I/O。在函数返回之前，WSASendTo 把第 4 个参数 lpNumberOfBytesSent 设为真正发送到接收端的字节数。lpTo 参数是针对具体协议的一个 SOCKADDR 结构，并带有接收端的地址。iToLen 参数是 SOCKADDR 结构的长度。最后两个参数，lpOverlapped 和 lpCompletionROUTINE，用于重叠 I/O(将在第 5 章中讨论)。

通过接收数据的方式，就可以把一个无连接的套接字连接到一个端点地址，并可以用 send 和 WSASend 发送数据。这种关联一旦建立，就不能再用带有地址的 sendto 和 WSASendTo，除非这个地址是传递到其中一个连接函数的地址，否则调用就会失败，出现 WSAEISCONN 错误。要取消套接字句柄与目标地址的关联，惟一的办法是在这个套接字句柄上以 INADDR_ANY 为目标地址调用 connect。

下面的示例代码展示了如何创建一个简单的 UDP 发送端应用程序。本应用程序的完整版本可在配套光盘中名为 UDPSENDER 的文件中找到。

```
#include <winsock2.h>

void main(void)
{
    WSADATA             wsaData;
    SOCKET              SendingSocket;
    SOCKADDR_IN         ReceiverAddr;
    int                 Port = 5150;
    char                SendBuf[1024];
    int                 BufLength = 1024;

    //初始化Winsock 2.2版本

    WSAStartup(MAKEWORD(2,2),&wsaData);

    //创建一个新的套接字来接收数据报

    SendingSocket = socket(AF_INET,SOCK_DGRAM,IPPROTO_UDP);

    //建立一个SOCKADDR_IN结构，来识别发送数据报的目的地
```

```
    //作为示范，假定接收者的IP地址是136.149.3.29，且接收者在5150端口等候数据报
    ReceiverAddr.sin_family = AF_INET;
    ReceiverAddr.sin_port = htons(Port);
    ReceiverAddr.sin_addr.s_addr = inet_addr("136.149.3.29");
    //把一个数据报发送到接收者

    sendto(SendingSocket,SendBuf,BufLength,0,
        (SOCKADDR *)&ReceiverAddr,sizeof(RecieverAddr));

    //应用程序完成发送数据报后，关闭套接字

    closesocket(SendingSocket);
    //应用程序结束后，调用WSACleanup

    WSACleanup();
}
```

1.7.3 基于消息的协议

正如面向连接的通信同时也是流协议，无连接通信几乎都是基于消息的。因此，在收发数据时，需要考虑这几点。首先，由于面向消息的协议保留了数据边界，所以提交给发送函数的数据在被发送完之前会形成阻塞。对非阻塞 I/O 模式而言，如果数据未能完全发送，发送函数就会返回 WSAEWOULDBLOCK 错误。这意味着下层的系统不能对不完整的数据进行处理，应该稍后再次调用发送函数。第 5 章将对此进行详述。需要记住的是，对于基于消息的协议而言，写入操作只能作为一种自动行为发生。

在连接的另一端，对接收函数的调用必须提供一个足够大的缓冲空间。如果提供的缓冲区不够大，接收调用就会失败，将出现错误 WSAEMSGSIZE。发生这种情况时，缓冲区会填满，但未接收完的数据会被丢弃。被截断的数据也无法恢复。唯一的例外是支持部分消息的协议，比方说 AppleTalkPAP 协议。当接收调用仅接收到部分消息时，WSARecv、WSARecvEx 或 WSARecvFrom 函数会在返回之前，将出入 flag 参数设为 MSG_PARTIAL。

对以支持部分消息的协议为基础的数据报来说，可考虑使用某个 WSARecv 函数。因为在调用 recv 或 recvfrom 时，不会有“读取的数据只是消息的一部分”这样一个通知。至于接收端怎样判断是否已读取完整条消息，具体方法则由程序员决定。随后的 recv/recvfrom 调用将返回这个数据报的其余部分。由于有这个限制，利用 WSARecvEx 函数就显得非常方便，因为它允许设置和读取 MSG_PARTIAL 标志，而 MSG_PARTIAL 标志指明整条消息是否已读取完毕。Winsock 2 函数 WSARecv 和 WSARecvFrom 也支持这一标志。关于这个标志的更多内容，请参见对 WSARecv、WSARecvEx 和 WSARecvFrom 这 3 个函数的描述。

最后要讲的便是在有多个网络接口的机器上发送 UDP/IP 消息。这方面的问题颇多，先来看一个最常见的问题：在一个 UDP 套接字被显式地绑定到一个本地 IP 接口，并发送数据报时，会发生什么情况？UDP 套接字并不会真正和网络接口绑定在一起，而是建立一种关联，即被绑定的 IP 接口成为发出去的 UDP 数据报的源 IP 地址。真正决定数据报在哪个物理接口上发送出去的，是路由表。如果不调用 bind，而是先调用 sendto/WSASendTo 或执行连接，网络堆栈就会根据路由表，自动选出最佳本地 IP 地址。这意味着，如果先执行显式绑定，源 IP 地址就可能有误。也就是说，源 IP 可能不是真正在它上面发送数据报的那个接口的 IP 地址。

1.7.4　释放套接字资源

因为无连接协议没有连接，所以也不会有对连接的正式关闭和从容关闭。在接收端或发送端完成收发数据时，它只需要在套接字句柄上调用 closesocket 函数，使可释放为套接字分配的所有相关资源。

1.8　其他 API 函数

本小节介绍其他几个 Winsock API 函数，它们在网络应用程序中非常有用。

1.8.1　getpeername

该函数用于获得通信方的套接字地址信息，该信息是关于已建立连接的那个套接字的。它的定义如下：

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

第 1 个参数是准备连接的套接字，后两个参数则是指向下层协议类型的 SOCKADDR 结构及其长度的指针。对数据报套接字来说，这个函数返回的是传递到连接调用的那个地址；但不会返回传递到 sendto 或 WSASendTo 调用的那个地址。

1.8.2　getsockname

该函数是对应 getpeername 的函数。它返回的是给定套接字的本地接口的地址信息。它的定义如下：

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

除了套接字 s 返回的地址信息是本地地址信息外，其参数和 getpeername 的参数都是一样的。在 TCP 协议中，这个地址与监听特定端口及 IP 接口的那个服务器套接字是一样的。

1.8.3 WSADuplicateSocket

WSADuplicateSocket 函数用来建立 WSAPROTOCOL_INFO 结构，该结构可传递到另一个进程，这样就可用另一个进程打开指向同一个下层套接字的句柄，如此一来，这个进程也能对该资源进行操作。注意，这一点只在进程之间的操作时才有必要；同一个进程中的线程可自由传递套接字描述符。该函数的定义如下：

```
int WSADuplicateSocket(
    SOCKET s,
    DWORD dwProcessId,
    LPWSAPROTOCOL_INFO lpProtocolInfo
);
```

第 1 个参数是准备复制的套接字句柄。第 2 个参数 dwProcessId，是打算使用所复制的套接字的进程的 ID。第 3 个参数 lpProtocolInfo，是一个指向 WSAPROTOCOL_INFO 结构的指针，该结构包含目标进程打开复制句柄时所需的信息。为了使当前的进程能够把 WSAPROTOCOL_INFO 结构传递给目标进程，然后目标进程再利用该结构建立一个指向指定套接字的句柄(利用 WSASocket 函数)，必须考虑进程间通信。

两个套接字的描述符都可独立使用 I/O；但 Winsock 没有提供访问控制，因此这要由程序员决定是否执行某种同步。因为是复制的套接字描述符，而不是实际的套接字，所以所有描述符中都可见到关联到某个套接字的所有状态信息。比方说，对于描述符上由 setsocketopt 函数设置的任何一个套接字选项，都可通过任何一个或所有描述符利用 getsockopt 函数来查看它们。如果一个进程在一个复制的套接字上调用 closesocket，就会导致该进程中的描述符被释放；但在最后留下的那个描述符上调用 closesocket 之前，下层套接字会保持打开状态。

另外，在使用 WSAAsyncSelect 和 WSAEventSelect 时，需要了解与共享套接字的通知有关的几个问题。这两个函数用于异步 I/O(第 5 章将对此进行讨论)。利用任何一个共享描述符执行这两个函数调用中的任何一个，都会删掉以前所有的套接字事件注册，不管用来注册的描述符是哪一个。例如，共享套接字不能把 FD_READ 事件传递给进程 A，也不能把 FD_WRITE 事件传递给进程 B。如果需要这两个描述符的事件通知，就应该重新设计应用程序，用线程来代替进程。

1.9 Windows CE

前面几个小节中的所有信息中也都可以应用于 Windows CE。惟一例外的是由于 Windows CE 是建立在 Winsock 1.1 规范基础上的，因此不能用 Winsock 2 中特有的函数，比如用于收发数据、建立

连接和接受连接的函数的 WSA 变体。Windows CE 平台上可用的 WSA 函数只有 WSAStartup、WSACleanup、WSAGetLastError 和 WSAIoctl。本章已经对前 3 个函数进行了讨论；最后一个留到第 7 章作全面讲解。

Windows CE 支持 TCP/IP 协议，这意味着可以同时访问 TCP 和 UDP。除了 TCP/IP 之外，它还支持红外线套接字。IrDA 协议只支持面向流的通信。对这两个协议来说，所有的常用 Winsock 1.1 API 函数调用都可用于创建和传输数据。惟一例外的操作是必须解决 Windows CE 2.0 内 UDP 数据报套接字中的错误：即每次调用 send 或 sendto，都会导致核心内存泄漏。这个错误在 Windows CE 2.1 中已得到修复，但由于内核分散在 ROM 中，因此，目前还没有更新的软件可以修复 Windows CE 2.0 中的这一错误。惟一的解决办法是不要在 Windows CE 2.0 中使用数据报。

由于 Windows CE 不支持控制台应用程序，而且只采用 UNICODE，所以本章介绍的示例都以 Windows 95、Windows 98、Windows NT 平台为目标平台。举出这些例子的目的是让大家直接了解 Winsock 的核心概念，用不着理会那些与 Winsock 无关的代码。除非此时正在编写 Windows CE 服务程序，否则几乎随时都需要一个用户界面。这样，便需要为窗口处理程序和其他用户界面元素编写若干新函数，而这样就违背了本书的初衷。除此以外，还必须了解 UNICODE 和非 UNICODE 的 Winsock 函数。至于传递给收发 Winsock 函数的指定字符串是 UNICODE 字符串还是 ANSI 字符串，则由程序员决定。只要传递的是一个有效的缓冲，Winsock 就不会注意其内容(当然，可能需要对缓冲类型进行转换以抑制编译警告)。需要记住，如果把一个 UNICODE 字符串转换成 char*，那么准备发送多少字节的长度就应该对参数做出相应的调整。因为其他的 Windows 系统函数都要求 UNICODE 字符串，所以在 Windows CE 中，如果打算把收到或发出的数据显示出来，必须考虑到它是不是 UNICODE，这样才能将它显示出来。总而言之，对于创建一个简单的 Winsock 应用程序而言，Windows CE 要复杂得多。

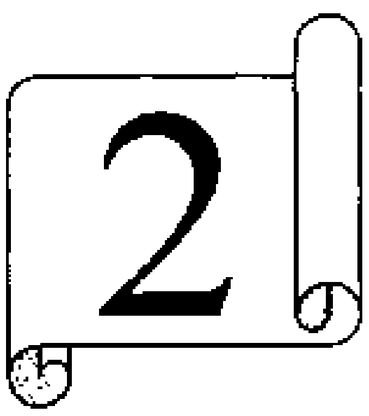
如果想在 Windows CE 上运行这些示例，只需要稍微修改 Winsock 代码即可。首先，头文件必须是 WINSOCK.H，而不是 WINSOCK2.H。因为 Winsock 1.1 是适用于 Windows CE 的最新 Winsock 版本，所以 WSAStartup 应该加载它。另外，Windows CE 不支持控制台应用程序，因而必须用 WinMain 来代替 main。注意，这并不意味着需要把一个窗口合并到应用程序中，只意味着不能用 printf 这一类的控制台文本 I/O 函数。

1.10　小　　结

本章介绍了如何具体使用 TCP 和 UDP 协议以建立面向连接通信和无连接通信时所需的核心 Winsock 函数。针对面向连接通信，本章演示了如何接受一个客户机连接，以及怎样建立一个到服务器的客户机连接。另外还介绍了面向会话的数据收发操作。针对无连接通信，本章也讲到了如何收发数据。由于本章的目的是介绍核心的 Winsock API，因此并未考虑涉及网络编程性能方面的问题。第

6 章将涉及性能方面的问题，并介绍微软 Winsock 的扩展，包括 TransmitFile、TransmitPackets、AcceptEx、GetAcceptExSockaddrs、ConnectEx、DisconnectEx 以及 WSARecvMsg 等。这些函数将有助于编写高性能、可升级的 Winsock 应用程序。

前面的讨论已经说明了如何用 Ipv4 协议来使用 Wonsock，随后 3 个章节将讲述 Winsock 结构的设计，并介绍其他可用协议的使用方法。



设计 Winsock

在第 1 章，我们介绍了 Winsock 的基础知识。本章我们将深入讲述系统体系结构，并讨论如何使 Winsock 适应总体系统设计的要求。然后讨论协议特性，并介绍应用程序如何枚举已安装的协议。最后讨论如何通过 socket 和 WSASocket 函数创建套接字，以及这些函数如何与 Winsock 编录进行交互。

2.1 系统体系结构

在了解系统体系结构之前，请注意：在将来的版本中，驱动程序名称和 DLL 等可能会改变，所以不能完全依赖于本章中讲述的具体细节设计应用程序，而应在全面了解系统总体设计的情况下，尽量遵守 Winsock 规范。下面所述内容适用于基于 Windows NT 的操作系统(本节最后将介绍 Windows NT 和 Windows 95、Windows 98 及 Windows Me 之间的一些区别)。

大部分 Winsock API 在 WS2_32.DLL 中实现，在 WINSOCK2.H 中声明。惟一的例外是 MSWSOCK.DLL 中的微软特有的 Winsock 扩展，如 TransmitFile 和 AcceptEx 等；因为这些扩展并不是 Winsock 正式规范的一部分，而是由微软所增加的，所以一些扩展 API 仅用在某些特定版本的 Windows 操作系统中(第 6 章将详细叙述这些 API)。

应用程序在调用 Winsock API 时，会调用 WS2_32.DLL。Winsock DLL 执行某种参数验证，进而确定应该将调用发送到哪一种协议服务提供程序。在 Winsock 编录中可能安装了多个提供程序，而 WS2_32.DLL 则确定由哪个提供程序处理该调用。

提供程序分为两种类型：基础提供程序和分层提供程序。基础提供程序位于传输协议顶端，如微软 TCP/IP 和 UDP/IP 提供程序，或实现 QOS(请见第 10 章)的 RSVP(Resource Reservation Protocol，资源保留协议)提供程序。微软基础提供程序由 MSAFD.DLL 和 MSWSOCK.DLL 组成，但实际上为 TCP/IP、IPX/SPX、NetBIOS、AppleTalk 等个体协议公开了一个或多个提供程序。本章稍后将叙述怎样编程列举系统中可用的提供程序。

分层提供程序位于 WS2_32.DLL 之下，基础提供程序之上；能截获并操纵 Winsock 调用。如果一个应用程序利用分层提供程序创建了一个套接字，分层提供程序将使用该套接字截获所有的 Winsock 调用。分层提供程序可能会阻塞、修改调用，也可能将未经修改的调用传递给底层的提供程序。第 12 章将详细介绍分层服务提供程序。

一旦 Winsock 调用到达基础提供程序，基础提供程序将接着依次调用 Winsock 核心模式组件。与其他操作系统不同，Windows NT 传输协议没有类似于 Winsock 的接口，这使得应用程序无法用这些接口同它们直接对话。Windows NT 传输协议使用一种更为通用的 API——传输驱动程序接口(Transport Driver Interface，TDI)，这种 API 所具有的通用性能，使 Windows NT 子系统不局限在特定旧版本的网络编程接口上。套接字模拟出 Winsock 核心模式驱动程序(目前在 AFD.SYS 中)实现。在为应用程序提供类似于套接字的接口时，这个驱动程序负责相关的连接和缓冲管理。AFD 接着将 TDI 传递给传输协议驱动程序(参见图 2.1)。

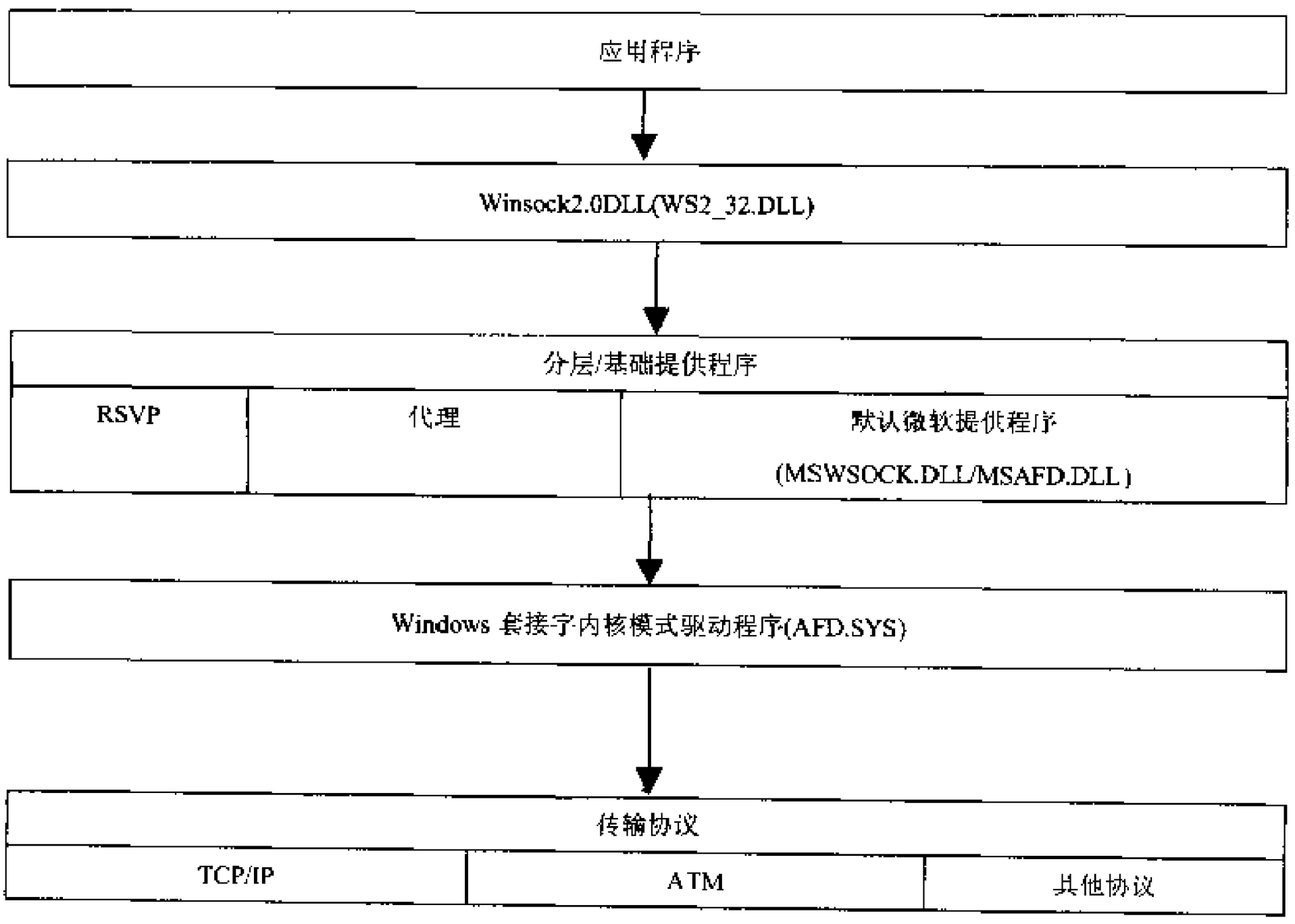


图 2.1　Winsock 系统体系结构

对 Windows 95、Windows 98 和 Windows Me 来说，除核心模式组件之外，总体结构和 Windows NT 相似。Windows 95、Windows 98 和 Windows Me 在 VXD 内实现驱动程序，所以没有 AFD.SYS 和 TCPIP.SYS。这样，它们驱动程序就为 AFVXD.VXD、WSOCK2.VXD 和 WSOCK.VXD 等。当然，正如前这所述，应用程序中不应该考虑实际的驱动程序或系统组件的文件名。Winsock API 规范在所有

版本的 Windows 操作系统中都可用。

2.2　协议的特征

在给出关于 Winsock 内核的一些信息之后，下面接着讨论可以访问的协议。前面已经提到，Winsock 提供程序实现的协议会表现出某种特征。Windows 支持各种不同的传输协议，如 TCP、UDP、IPX 及 SPX 等。每种协议各自有着不同的特性。有些要求在收发数据之前建立连接，有些则不能确保数据的可靠性或完整性。本节将叙述这些协议的各种特征。

2.2.1　面向消息

对每个离散的写命令来说，如果某个传送协议只将那些字节当做一条独立的消息在网上传输，我们就说这种协议是面向消息的。同时，这还意味着接收端发出数据请求后，接收到的数据是发送端写的一条离散消息。接收端只能收到一条消息。例如，在图 2.2 中，左边的工作站向右边的工作站提交了 3 条长度分别为 128、64 和 32 字节的消息。作为接收端的工作站发出 3 条 recv 调用，调用的缓冲区是 256 个字节。每次调用依次返回 128、64 和 32 个字节。第 1 次 recv 调用不会把所有的 3 个数据包都返回，即使它已经全部收到这些数据包。这种原理被称为“保留消息边界”(preserving message boundaries)，一般在交换结构化数据的时候出现。“保留消息边界”的一个范例是网络游戏。在网络游戏中，每个玩家均向别的玩家发出一个带有物理位置信息的数据包。这种通信后面隐含的代码很简单：一个玩家发出一个数据包请求，而且也从参与该游戏的另一的玩家处获得一个物理位置信息的数据包。

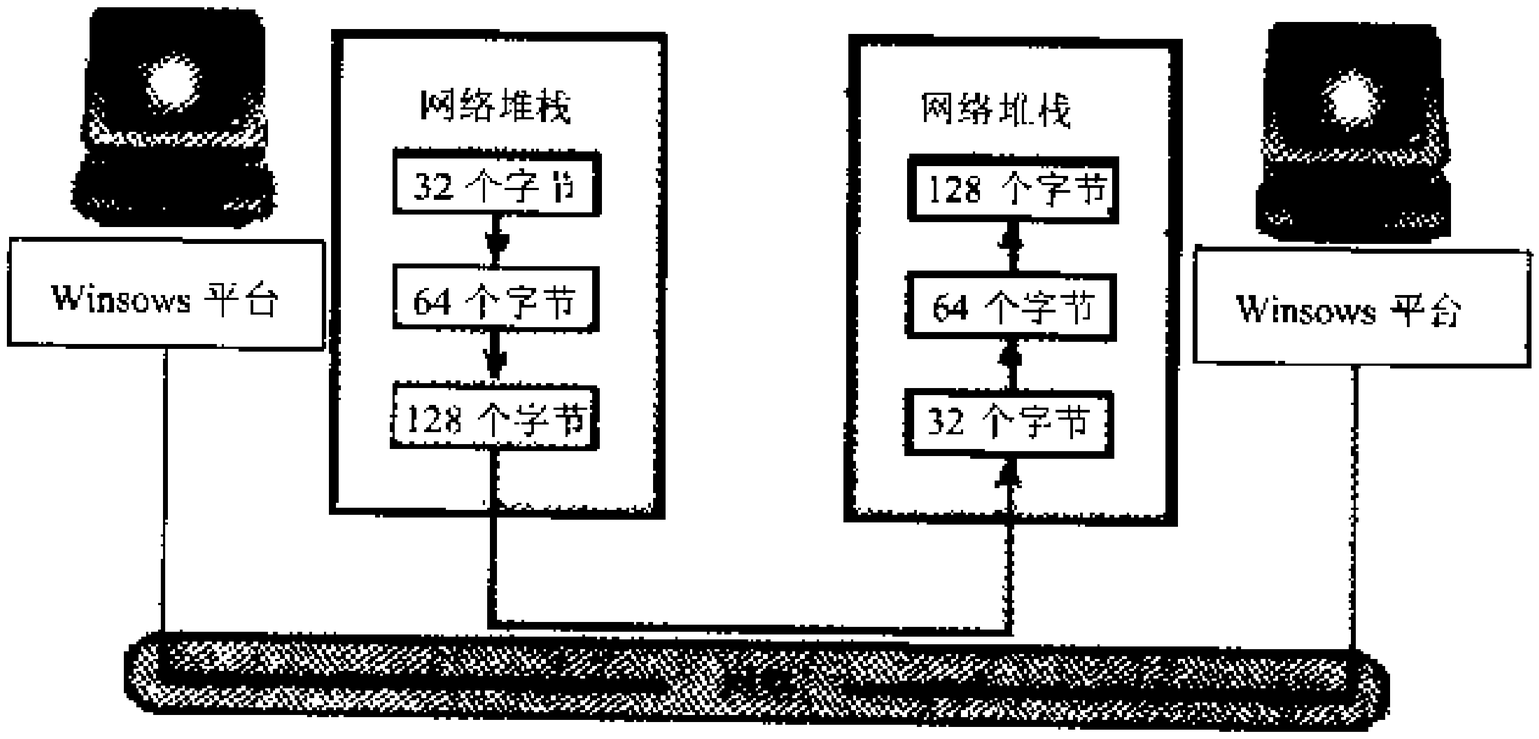


图 2.2　数据报服务

2.2.2　面向流

不保留消息边界的协议通常称作“基于流的协议”。要注意基于流这一术语常用来指代附加特性。

流服务的定义是连续的数据传输；不管消息边界是否存在，接收端都会尽量地读取有效数据。对发送端来说，流服务允许系统将原始消息分解成小消息或把几条消息积累在一起，并形成一个较大的数据包。对接收端来说，数据一到达网络堆栈，网络堆栈就开始读取它，并将它放在缓冲区中等候进程处理。在进程请求处理大量数据时，系统会尽量返回更多的数据。但有一个前提，那就是不能溢出为客户端调用提供的缓冲区。在图 2.3 中，发送端提交了 3 个数据包：长度分别为 128、64 和 32 个字节；本地系统堆栈把这些数据聚合在一起，形成一个更大的数据包。这种情况下，重组后的两个数据包就会一起传输。是否将各个独立的数据包累积在一起，要由许多因素决定，例如最大传输单元或 Nagle 算法。在 TCP/IP 中，Nagle 算法要求主机等待数据积累起来以后再将它们发到线上。这个主机将一直等下去，直到需要发送的数据积累到一定数量，或超过预定时间为止。实施 Nagle 算法时，主机的通信对方会在向主机发送确认之前，花费一段预定的时间来等候要传出的数据，这样，主机的通信对方就不必发送一个只有确认消息的数据包。发送小数据包不仅没有多少意义，而且还会徒增错误检查和确认的开销。

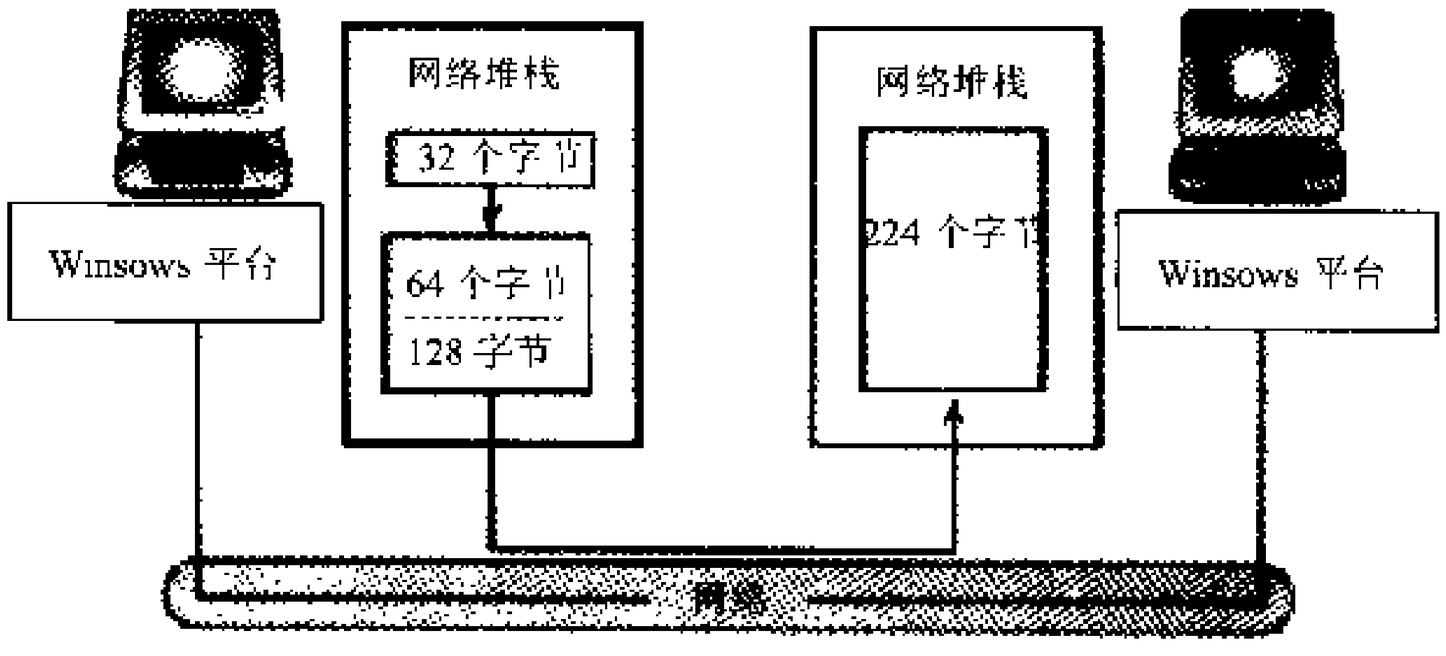


图 2.3 流服务

在接收端，网络堆栈会针对给定进程把所有传入的数据聚集在一起。我们来看看图 2.2。只要接收端一执行 256 字节缓冲区的 recv 调用，系统就会马上返回所有的 224 个字节。如果接收端只要求读取 20 个字节，则系统就只返回 20 个字节。

2.2.3 伪流

伪流(pseudo stream)这个术语常用于这样的系统中：这种系统使用的协议是基于消息的，该协议将数据放在离散的包中发送，接收端读取这些数据包并把它们放到缓冲池中，这样，接收应用程序便可读取任意大小的数据块。把图 2.2 中的发送端和图 2.3 中的接收端结合起来，便可说明伪流的工作原理。发送端必须分别发送每个独立的数据包，但接收端可以按任何适用的大小自由合并收到的数据包。多数情况下，可把伪流视作一个普通的“面向流的协议”。

2.2.4　面向连接和无连接

通常情况下，一个协议要么提供面向连接的服务，要么提供无连接的服务。在面向连接的服务中，进行数据交换之前，通信双方必须建立一条路径。这样既确保了通信双方之间存在路由，又保证了通信双方是活动的，可彼此响应。但这样做同时也意味着在通信双方之间建立一个通信信道需要大量的开销。除此以外，大部分面向连接的协议为保证投递无误，可能会执行额外的计算来验证正确性，从而进一步增加开销。而无连接协议却不保证接收端是否正在侦听。无连接服务类似于邮政服务：发信人把信装入邮箱即可。至于收信人是否盼望收到这封信，或邮局是否会因为暴风雨未能按时将信件投递到收信人处，发信人都不得而知。

应注意到，对于某些无连接协议(如 UDP)，Winsock 应用程序可能会使用目的地的 IP 地址来调用 connect 函数，但这并不意味着建立了任何物理连接。这仅仅是将目标地址和套接字关联起来的一种方便途径，以使可以使用 send 和 WSASend API 函数来代替 sendto 和 WSASendTo 函数。

2.2.5　可靠性和有序性

在设计一个使用某种协议的应用程序时，可靠性和有序性是必须注意的最关键的特性。可靠性(或有保证的交付)保证发送端发出的每个字节都能到达既定的接收端。不具备可靠性的协议，则不能保证每个字节都能到达接收端，也就不能保证数据的完整性。

有序性是指对数据到达接收端的顺序进行处理。如果一个协议保留了有序性，它就能够保证接收端所收到数据的顺序就是数据的发送顺序。显然，没有保留有序性的协议就没有次序保证。

大多数情况下，可靠性和有序性与协议是无连接的还是面向连接的特性密切相关。在面向连接的通信中，如果试图在通信双方之间建立一个清晰的通信信道，则一般希望所采用的协议能够保证数据的完整性和有序性。多数情况下，面向连接的协议总是能够保证数据的可靠性。注意，可靠性和有序性两者不能兼而得之，保证了数据包顺序，就不能自动保证数据的完整性。不言而喻，无连接协议的过人之处就是速度；因为它根本不必费心去建立一个指向接收端的虚拟连接。为什么错误检查降低了速度呢？这便是通常情况下无连接协议不能保证数据完整性和有序性，而面向连接的协议却能做到这一点的原因。数据报如此不完善，为什么大家都喜欢用它呢？一般来说，无连接协议比面向连接的通信要快得多。另外，它不必验证数据完整性和收到数据的确认信息——这两者即使在发送小量数据时也会大大增加复杂性。对于非关键的数据传输，数据报非常有用。而对于前面讨论过的网络游戏之类的应用程序而言，数据报简直就是为它们量身定做的：每个玩家都可利用数据报周期性地向别的玩家发送他/她在游戏中的位置。如果某一个客户机丢失了一个数据包，它很快就能收到另外一个，这就给玩家带来一种不间断通信的印象。

2.2.6 正常关闭

正常关闭只出现在面向连接的协议中。在这种关闭过程中，其中一方开始关闭通信会话，但此时另一方仍然可以读取线路上或网络堆栈上挂起的数据。如果面向连接的协议不支持正常关闭，则只要其中一方关闭了通信信道，就会出现连接立即中断、接收端丢失还未读取的数据这些情况。如果使用的是 TCP 协议，则连接双方都必须执行一次关闭，才能完全中断连接。发起方先向另一个通信方发出一个带有 FIN 控制标志的片段(数据报)。接收方则向发起方返回一个 ACK 控制标志，确认已收到 FIN，但此时它仍然可发送数据。FIN 控制标志表示，发起关闭的这一方再也不会发送数据。一旦对方决定不需要再发送数据，它也会发出一个 FIN 控制标志。随后，发起方用 ACK 控制标志加以确认。这时，连接就完全关闭了。

2.2.7 广播数据

所谓广播数据，就是从一个工作站发出数据，局域网内的其他所有工作站都能收到。因为局域网上的所有计算机都可获得并处理广播消息，所以，这一特征适用于无连接协议。使用广播消息的不利之处是，每台计算机都必须对该消息进行处理。例如，如果一个用户在局域网上广播一条消息，则每台计算机上的网卡都会收到这条消息，并把它上传到网络堆栈。然后，堆栈会在所有的网络应用程序中循环，看它们是否应该接收这条消息。通常，这个局域网上的多数计算机对该消息都不会感兴趣，最后会草草地将该消息一弃了之。但是，这些计算机仍然需要花时间来处理这个数据包，看是否有应用程序对它感兴趣。结果，每个工作站都会检查这个数据包，这样，高广播通信量就会致使局域网上的计算机陷入困境。一般情况下，路由器不会在网络间传送广播数据包。

2.2.8 多播数据

多播是指一个进程发送数据而一个或多个接收端可接收到数据的能力。进程加入一个多播会话所使用的方法和下层协议有关。例如，IP 协议下，多播就是广播的一种变体。IP 多播要求对收发数据感兴趣的所有主机加入一个特定的组。当进程希望加入某个多播组时，网卡上便会增添一筛选器，这样，只有被绑定多播组地址的数据才会被网络硬件接收，并被上传到网络堆栈进行适当处理。视频会议应用程序常常使用多播。第 9 章将详细论述 Winsock 的多播编程和其他一些多播的关键问题。

2.2.9 服务质量

服务质量(QOS)体现了应用程序请求独占网络带宽的能力。实时视频流就较好地利用了 QOS。对实时视频流应用程序的接收端而言，要显示平滑清晰的图像，发送的数据必须被限制在特定范围内。过去，应用程序是将数据缓存下来，再从缓冲区中重播帧，从而获得流畅的视频。这样，当某个时段接收数据的速度不够快时，重播例程也会拥有一定量缓存起来的帧可供播放。QOS 允许在网络上预留带宽，因此，可以在一系列有保证的限制条件下，离线读取帧。理论上讲，这意味着同一个实时视频

流应用程序可以使用 QOS，从而避免使用缓存帧。第 10 章将对 QOS 作详细讨论。

2.2.10　部分消息

部分消息只用于面向消息的协议。例如，一个应用程序想接收一条消息，但本地计算机仅收到了它的一部分数据。这种情况是可以出现的，特别是当发送端计算机正在传输大型消息，而本地计算机却没有足够多的资源来容纳整条消息的时候。实际上，多数面向消息的协议对数据报的最大长度都有一个合理限制，因此，这一特殊事件不会经常发生。但是，大多数数据报协议都支持大型消息——大得足以需要分解成许多小块才能在物理媒体上传输。如此一来，用户的应用程序请求读取这一消息时，就有这样一种可能性存在：用户系统可能只收到了部分消息。如果所用的协议支持部分消息，它就会通知读取方已返回的数据只是整条消息中的一部分。如果所用的协议不支持部分消息，下层网络堆栈就会继续接收其余的消息，直到收完为止。如果由于某种原因，导致整条消息不能全部到达，那么收到的数据报就会不完整，而多数不支持部分消息的不可靠协议都会把这个不完整的数据报丢弃。

2.2.11　路由选择的考虑

对应用程序开发者而言，一个需要考虑的重要方面，就是协议是否可路由。如果协议可路由，就可在两个工作站之间顺利地建立一条通信路径(要么是虚拟的面向连接的回路，要么是数据报通信的数据路径)，至于这两个工作站之间的网络硬件是什么，则无关紧要。例如，计算机 A 和计算机 B 分别在各自的子网中，连接两个子网的路由器 A 把这两台计算机分开了。可路由的协议将意识到计算机 B 和计算机 A 不在同一个子网上；因此它会把数据包定向到路由器，由路由器来决定将数据送达计算机 B 的最佳转发方式。非路由协议则没有这个能力——路由器会将它收到的非路由协议数据包统统丢弃。对发自非路由协议的数据包，即便它们的既定目的地是在路由器已连接的子网上，路由器也不会转发这些数据包。NetBEUI 是 Windows 平台支持的惟一一个不能路由的协议。

2.2.12　其他特征

Windows 平台上支持的各个协议都展现出一些特定的或特殊的特征。同时，还有大量不胜枚举的其他特征，例如字节序和最大传输单元等，都可用来描述目前网络上可用的各种协议。然而，在编写 Winsock 应用程序时，并非所有这些特征都是关键的。Winsock 2 提供了一种功能，可把每个实用的协议提供程序列举出来，还可以查询它们的特征。本章下一节对这一功能进行阐释，并给出了一个代码示例。

2.3　Winsock 编录

Winsock 编录是一个数据库，它包含系统中可用的各种不同的协议。Winsock 2 提供了一种办法，

可以确定在一个给定工作站上安装了哪些协议，并返回每种协议的各种特征。如果一个协议支持多种行为，则每一种不同的行为类型在系统中都会有各自的编录条目。例如，如果在系统中安装了 TCP/IP，系统中就会有两个 IP 条目：一个条目针对 TCP，是可靠的而且面向连接的；而另一个条目则针对 IP，是不可靠且又无连接的。

要想获得系统中安装的网络协议的相关信息，可以调用 WSAEnumProtocols 函数，这个函数的定义为：

```
int WSAEnumProtocols (
    LPINT lpiProtocols,
    LPWSAPROTOCOL_INFO lpProtocolBuffer,
    LPDWORD lpdwBufferLength
);
```

这个函数取代了 Winsock 1.1 中的 EnumProtocols 函数，EnumProtocols 函数是 Windows CE 中必不可少的。两者惟一的不同是：WSAEnumProtocols 返回的是 WSAPROTOCOLS_INFO 结构数组；而 EnumProtocols 返回的则是 PROTOCOL_INFO 结构数组，PROTOCOL_INFO 结构中包含的字段少于 WSAPROTOCOLS_INFO 结构中的字段(但包含的信息是差不多的)。WSAPROTOCOL_INFO 结构的定义为：

```
typedef struct _WSAPROTOCOL_INFO {
    DWORD dwServiceFlags1;
    DWORD dwServiceFlags2;
    DWORD dwServiceFlags3;
    DWORD dwServiceFlags4;
    DWORD dwProviderFlags;
    GUID ProviderId;
    DWORD dwCatalogEntryId;
    WSAPROTOCOLCHAIN ProtocolChain;
    int iVersion;
    int iAddressFamily;
    int iMaxSockAddr;
    int iMinSockAddr;
    int iSocketType;
    int iProtocol;
    int iProtocolMaxOffset;
    int iNetworkByteOrder;
    int iSecurityScheme;
    DWORD dwMessageSize;
    DWORD dwProviderReserved;
    TCHAR szProtocol[WSAPROTOCOL_LEN +1];
}WSAPROTOCOL_INFO,FAR *LPWSAPROTOCOL_INFO;
```

要调用 WSAEnumProtocols 函数，最简单的方法是令 lpProtocolBuffer 等于 NULL，并将

lpdwBufferLength 设置为 0，然后进行第 1 次调用。这个调用会失败，并返回 WSAENOBUFS 错误，但返回的 lpdwBufferLength 中将包含正确的缓冲区长度(这一长度是返回所有协议信息所必需的)。一旦分配了正确的缓冲区长度，并利用这个缓冲区进行第 2 次调用，WSAEnumProtocols 函数会返回 WSAPROTOCOL_INFO 结构的数量。这时，就可逐步深入各结构，查找自己所需要的协议条目了。补充材料各上的示例程序 ENUMCAT.C 列举了所有已安装的协议及其特性。

WSAPROTOCOL_INFO 结构最常用的字段是 dwServiceFlags1，它是代表不同协议属性的一个位字段。表 2.1 列出了可在字段中设置的各种位标志，并对各属性的含义进行了简要说明。

表 2.1　协议标志

属　性	含　义
XP1_CONNECTIONLESS	该协议提供无连接的服务。如果不设置该标志，则协议支持面向连接的数据传输
XP1_GUARANTEED_DELIVERY	该协议保证发送出去的所有数据都将到达既定接收端
XP1_GUARANTEED_ORDER	该协议保证数据按其发送顺序到达接收端，且数据不会重复。但是，该协议不能保证传送是否成功
XP1_MESSAGE_ORIENTED	该协议提供消息边界功能
XP1_PSEUDO_STREAM	该协议是面向消息的，但接收端会忽略消息边界
XP1_GRACEFUL_CLOSE	该协议支持两种关闭：通知各个通信方另一方打算关闭通信信道。如果不设置该标志，则只执行异常关闭
XP1_EXPEDITED_DATA	该协议提供带外数据(out-of-band 数据)
XP1_CONNECT_DATA	该协议支持带有连接请求的数据传输
XP1_DISCONNECT_DATA	该协议支持带有无连接请求的数据传输
XP1_SUPPORT_BROADCAST	该协议支持广播机制
XP1_SUPPORT_MULTIPOINT	该协议支持多点或多播机制。第 9 章将介绍多点通信
XP1_MULTIPOINT_CONTROL_PLANE	如果设置了这个协议标志，就会启动控制面板。反之，则不启动
XP1_MULTIPOINT_DATA_PLANE	如果设置了这个协议标志，就会启动数据面板。反之，则不启动
XP1_QOS_SUPPORTED	该协议标志支持 QOS 请求。第 10 章将介绍 QOS

续表

属 性	含 义
XP1_UNI_SEND	该协议在发送方向上是单向的
XP1_UNI_RECV	该协议在接收方向上是单向的
XP1_IFS_HANDLES	提供程序返回的套接字描述符是 IFS(Installable File System，可安装文件系统)句柄，可用于 ReadFile 及 WriteFile 之类的 API 函数中
XP1_PARTIAL_MESSAGE	WSASend 和 WSASendTo 中均支持 MSG_PARTIAL 标志
XP1_INTERRUPT	备用标志

这里的大多数协议标志都将在下面几章中作讨论，所以在此不会详细介绍各标志的全部含义。其他一些重要字段是 iProtocol、iSocketType 和 iAddressFamily。iProtocol 字段定义所在的条目属于哪种协议。对支持多种行为的协议来说(例如面向流的连接或数据报连接)，iSocketType 字段非常重要。iAddressFamily 字段用于为给定协议识别正确的寻址结构。

2.3.1 Winsock 编录和 Win64

在 64 位 Windows 中，可以在 WOW64(Windows on Windows，Windows 之上的 Windows)子系统中运行 32 位应用程序。因为 32 位和 64 位应用程序可能都要访问 Winsock 目录，因此系统将保留两个独立的目录。当运行 64 位 Winsock 应用程序并调用 WSAEnumProtocols 函数时，使用 64 位目录。而运行 32 位 Winsock 应用程序并调用 WSAEnumProtocols 函数时，则使用 32 位目录。在涉及到第 12 章中讨论的“Winsock 服务接供程序接口”时，Winsock 目录将变得尤为重要。

2.3.2 创建套接字

第 1 章的简单例子展示了如何使用 socket 函数来创建套接字。这个函数具有 3 个参数：地址族、套接字类型及协议。应用程序创建套接字时，它将参考 Winsock 目录，并试图将提供的参数和每个 WSAPROTOCOL_INFO 结构中包含的参数匹配。如果参数匹配成功，就根据参数的提供程序创建一个套接字。应注意到，某些情况下协议参数可以是 0。如果 WSAPROTOCOL_INFO 结构中的 dwProviderFlags(提供程序标志)字段是 PFL_MATCHES_PROTOCOL_ZERO，且请求的地址族及套接字类型与结构中的相应条目匹配，就使用相应的提供程序来创建套接字。例如，考虑下面的调用：

```
SOCKET s;
s =socket(AF_INET,SOCK_STREAM,0);
```

Winsock 将列举目录，并首先匹配后面紧随套接字类型的地址族。由于协议值是 0，而且提供程

序 MSAFD TCP 也包含了 PFL_MATCHES_PROTOCOL_ZERO 标志，所以，这个调用将根据 MSAFDTCP 提供程序创建一个 TCP 套接字。之后，系统不会再把这个请求与任何其他的提供程序匹配。

在某些情况下，多个提供程序可以共享同一个地址族、套接字类型及协议。RSVP 提供程序就存在这种情况。RSVP 提供程序为 TCP/IP 和 UDP/IP 都提供 QOS。因为多个提供程序可以共享同一个地址族、套接字类型及协议，所以不可能用 socket API 从 RSVP 提供程序创建套接字。为了解决这个问题，必须使用 Winsock 2 的 WSASocket 函数，该函数的定义为：

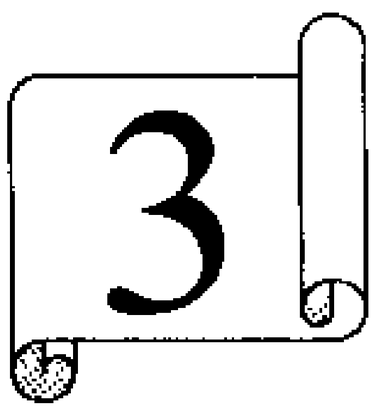
```
SOCKET WSASocket(
    int af,
    int type,
    int protocol,
    LPWSAPROTOCOL_INFO lpProtocolInfo,
    GROUP g,
    DWORD dwFlags);
```

前 3 个参数和 socket 函数的相应参数相同，而第 4 个参数采用了 WSAPROTOCOL_INFO 结构形式。如果 lpProtocolInfo 引用了一个提供程序条目，且前 3 个参数的值都是预先设定的 FROM_PROTOCOL_INFO，函数就根据给定提供程序创建一个套接字。应用程序使用这种方法便能创建一个 RSVP 套接字。

第 4 个参数处理套接字组，该套接字组在 Winsock 规范中有所叙述，但并不在 Windows 中实现。最后一个参数是可选标志，可以忽略。由此看来，仅仅 WSA_FLAG_OVERLAPPED 标志比较重要。如果打算使用重叠的 I/O(如第 5 章所述)，则创建套接字时，就需要给出这个标志。应注意到，使用 socket API 时，总是意味着同时也使用了重叠标志。其他套接字标志和多播有关，将在第 9 章中讲到。

2.4　小　　结

在这一章中，我们了解到 Winsock 如何适应总体系统结构，以及各种不同的协议如何插入到系统中。另外，我们看到了各种协议表现出来的特征，还了解到如何编程列举 Winsock 目录，以便获取这些特征。最后，我们了解到如何使用 WSASocket API 根据显式提供程序创建套接字。在下一章，我们将更详细地分析 IP 协议，包括 IPv4 和 IPv6。



网 际 协 议

本章讲述网际协议(IP)。第 1 章讨论到，要通过 Winsock 建立通信，必须了解如何针对特定协议(已在第 1 章中以 IPv4 为例作了说明)为工作站寻址。本章主要讨论 IPv4 和 IPv6，第 4 章将叙述 Windows 平台上最常见的协议。

IPv4 是一种用于 Internet 的网络协议，已经广为人知。IP 可广泛用于大多数计算机操作系统上，也可用于大多数局域网(LAN，例如办公室小型网络)和广域网(WAN，例如 Internet)。随着 Internet 中计算机数量的爆炸式增长，IPv4 的局限性越来越明显。自然而然地，下一代 IP 被开发出来，称为 IPv6。

本章将讨论 IPv4 及 IPv6 的背景、寻址方案、名称解析及 Winsock 细节。然后讨论怎样编写在两个版本的 IP 中都能够无缝运行的应用程序。

3.1 IPv4

IPv4 由美国国防部 ARPA(Advanced Research Project Agency，高级研究计划局)开发，该局在 20 世纪 60 年代建立了一个试验性的数据包开关网络。最初的网络协议很麻烦，这致使在 70 年代中期开发出了一种较好的协议。这项研究最终不仅发展到 TCP，还进一步发展到 IPv4。

3.1.1 寻址

IPv4 中，计算机都分配有一个 IP 地址，IP 地址用一个 32 位数来表示，正式的称呼是“IPv4 地址”。IPv4 地址通常表示为点分十进制格式，地址中的每 8 位字节被转换为一个十进制数值，并由句点分隔。

IPv4 地址被分为几个种类，分别描述地址被分配到网络的部分及分配到端点的部分。表 3.1 列出了各种不同的种类。

表 3.1　IPv4 地址类别

种类	网络部分	第 1 个数字	端点数字
A	8 位	0-127	16 777 216
B	16 位	128-191	65 536
C	24 位	193-223	256
D	N/A	224-239	N/A
E	N/A	240-255	N/A

指定一个 IP 地址时，表明网络部分位数的数字可以在反斜线“/”后附加到点分十进制地址后边。例如，地址 172.31.28.120/16 表明，地址的前面 16 位数字组成了地址的网络部分。这相当于使用 255.255.0.0 子网掩码起到的作用。

表 3.1 中的最后两项是 IPv4 地址的特殊种类。D 类地址是为 IPv4 多播预留的，E 类地址为试验性地址。另外，有几块地址是为专门用途预留的，不能被 Internet 中的系统使用。这些地址如下所示：

- 10.0.0.0–10.255.255.255(10.0.0.0/8)
- 172.16.0.0–172.31.255.255(172.16.0.0/12)
- 192.168.0.0–192.168.255.255(192.168.0.0/16)

最后，还有一个环回地址(127.0.0.1)，该地址为特殊地址，指向本地计算机。

要列出分配给本地接口的 IPv4 地址，可以使用 IPCONFIG.EXE 命令，该命令用来列出每个网络适配器及其分配到的 IPv4 地址。如果应用程序需要通过编程获取 IPv4 地址的列表，可以使用 SIO_ADDRESS_LIST_QUERY 命令调用 WSAIoctl 函数，这在第 7 章有所叙述。另外，IP 助手 API 提供这个函数，这在第 16 章中叙述。

我们已讨论了 IP4v 地址空间的细目分类，在这些互不相同的地址种类中，又有 3 种类型的 IPv4 地址：单播、多播和广播。随后的几个小节将对每种类型的地址展开叙述。

3.1.1.1　单播

分配到单个计算机接口上的地址称为单播地址。该地址仅可以分配到一个接口上。如果网络上其他计算机也配置了该地址，就会发生错误，导致数据的错误传输。A、B、C3 类地址组成 IPv4 的单播地址空间。

一般说来，为主机上的接口分配 IPv4(单播)地址时，要么静态地配置，要么由配置协议分配，例如用 DHCP(Dynamic Host Configuration Protocol，动态主机配置协议)。如果不能访问 DHCP 服务器，则系统将使用 APIPA(Automatic Private IP Addressing，自动专用 IP 寻址)自动分配一个 169.254.0.0/16

范围内的地址。

为了避免记忆大量的 IP 地址，可以使用 DNS(Domain Name System，域名系统)将 IPv4 地址和主机计算机名称相关联。稍后将讨论如何将主机名称解析为其 IPv4 地址(以及其 IPv6 地址)。

3.1.1.2 多播

多播地址未被分配到某个特定接口。相反，多个计算机可以“加入”一个多播组，监听某个特定的多播地址。加入该组的每个计算机都将收到发往该多播地址的任何数据。多播地址是 D 类地址。多播最大的一个好处是，能够将多播数据仅传送到对该数据感兴趣的那些计算机。IP 多播将在第 9 章详细讨论。

3.1.1.3 广播

IPv4 支持数据广播。这意味着发送到受限广播地址 255.255.255.255 的广播数据将被局域网内的每个计算机接收并处理。因为即使是那些对广播数据不感兴趣的计算机，也必须处理数据包，所以通常认为这种做法不好。

如果应用程序需要进行广播，最好使用子网直接广播的方式。这种方式也是广播数据，但顾名思义，这种方式仅将数据直接广播到某个特定子网内的计算机。例如，发送到 172.31.28.255 的一个数据报将只被同一个子网内的计算机接收到。

3.1.2 IPv4 管理协议

IPv4 需要依赖几个其他的协议才能实现其功能。其中我们感兴趣的 3 个支持协议为：ARP(Address Resolution Protocol，地址解析协议)、ICMP(Internet Control Message Protocol，Internet 控制消息协议)及 IGMP(Internet Group Management Protocol，Internet 组管理协议)。

ARP 用来将一个 32 位 IPv4 地址解析为一个物理地址或硬件地址，使得 IPv4 数据包能够被包装在适当的媒体帧(如以太网帧)中。在线路上发送数据之前，主机必须将下一个跃点的 IPv4 地址解析为对应的硬件地址。如果目标地址位于局域网内，则 ARP 请求将针对目的地的物理地址。如果一个或多个路由器将源和目的地分离开来，则 ARP 请求将针对缺省网关，数据包也将发往该网关。第 16 章叙述的 IP 助手 API 包含了一些 ARP 例程。

设计 ICMP 的目的是为了在 IPv4 主机之间发送状态信息和错误信息。信息类型包括回应请求和答复、目标不可抵达以及超时。ICMP 也被用来搜寻邻近路由器。第 11 章将对 ICMP 展开详细讨论，并说明如何发送 ICMP 信息。

IGMP 用来管理多播组的成员。主机上的应用程序加入多播组后，主机就会向外发送 IGMP 成员报表，通知该网络段的路由器，应该由哪些多播组来接收数据。路由器需要这种信息，以便将发往这些多播组的数据包向前发送——如果某网络段有对该数据感兴趣的接收者，则发往该网络段，否则不

发送。第 9 章将详细讨论 IGMP。

3.1.3　Winsock 中的 IPv4 寻址

在 Winsock 中，应用程序通过 SOCKADDR_IN 结构来指定 IPv4 地址和服务端口信息，该结构的定义为：

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

sin_family 字段必须设为 AF_INET，以告知 Winsock 我们此时正在使用 IP 地址族。sin_port 字段定义了用来标识服务器服务的 TCP 或 UDP 通信端口。应注意到，端口号其实没有应用到 IPv4 协议中，而是封装在 IPv4 报头内的传输层协议(如 TCP 或 UDP)的一个属性。

因为有些可用端口号是为已知的服务(例如 FTP 和 HTTP)保留的，所以应用程序选择端口时，必须特别小心。已知服务使用的端口是由 IANA(Internet Assigned Numbers Authority，Internet 编号授权委员会)分配的，这些端口列举在网页 http://www.iana.org/assignments/port-numbers 上。实质上，端口号分为下述 3 类：已知的、已注册的及动态和(或)专用端口。

- 0～1 023：由 IANA 控制，为已知服务所保留。
- 1 024～49 151：由 IANA 列出的已注册端口，由普通用户执行的普通用户进程或程序可以使用这些端口。
- 49 152～65 535：动态和(或)专用端口。

普通用户应用程序应在 1 024～49 151 范围内选用已注册端口，以避免可能使用其他应用程序或系统服务正在使用的端口。因为 IANA 没有在 49 152～65 535 范围内的端口上注册服务，所以这些端口也可随意使用。如果使用 API 函数 bind 时，应用程序绑定到了主机上另一个应用程序正在使用的端口上，则系统将返回 Winsock 出错信息 WSAEADDRINUSE。另外，即使没有显式地绑定到一个本地地址和端口上，客户机的发送或连接操作也会有效。在这种情况下，系统将把套接字隐式地绑定到 1024～5000 范围内的一个本地端口上。如果应用程序显式地绑定套接字，但指定一个为 0 的本地端口时，系统也会采取同样的行为。

SOCKADDR_IN 结构的 sin_addr 字段用于把 Ipv4 地址保存为 4 字节的以网络字节顺序排列的数值，它是一个无符号长整数的数据类型。根据不同的用法，这个字段还可表示一个本地或远程 IP 地址。IP 地址一般是用“Internet 标准点分表示法”像“a.b.c.d”一样指定的，其中每个字母代表一个字节数，并从左到右分配了一个 4 字节的无符号长整数。最后一个字段 sin_zero，只起到充当填充项的

作用，以使 SOCKADDR_IN 结构和 SOCKADDR 结构的长度一样。

这个套接字及其他套接字地址的结构中，所有字段都得按照网络字节顺序排列。然而，如果应用程序使用本章稍后将讨论到的名称解析和分配的 API，则所需的排序转换将自动实行。仅当应用程序从结构成员中显式地分配或获取数值时，才需要进行字节排序转换。字节排序已在第 1 章中进行了介绍。

3.2 IPv6

随着 Internet 中计算机数量的爆炸式增长，IPv4 的局限性越来越明显。首先，可用 IPv4 地址的数目不断地被消耗掉，这就导致了 NAT(network address translators，网络地址转换器)的使用。NAT 将多个专用地址映射到单个的公共 IP 地址上。对客户机—服务器应用程序而言，NAT 是有用的，不过在连接两个使用专用地址空间的组织时，NAT 就可能会出现问题。另外，有时 NAT 必须注意到下层协议，以便执行适当的地址转换。

其次，IPv4 寻址并非完全分等级的，这意味着 Internet 中枢路由器必须保留大量的路由表，以便将 IPv4 数据包正确地发送到 Internet 中任何地点。

开发 IPv6 的另一个诱因，是为了提供更简单的配置过程。用 IPv4 时，地址必须被静态地分配，或通过配置协议如 DHCP 进行分配。比较理想的情况是，主机不必依赖 DHCP 结构的管理，而应该能基于其所在的网络段对自身进行自动配置。

Windows XP 提供了 IPv6 的 developer-release 版本。对于 Windows 2000，可以从 http://www.microsoft.com/ipv6 下载 IPv6 协议的技术预览版。对于 Windows NT 4.0，也可从该网址获取“微软研究 IPv6 协议”。

本节将讨论 IPv6 地址的不同类型、IPv6 使用的支持协议，以及 IPv6 地址在 Winsock 中是被如何处理的。尽管我们将讨论寻址和名称解析，但并不会涉及 IPv6 的所有方面，如并没有讲述路由和建立 IPv6 网络。要了解更多的信息，请查阅 Windows XP 联机帮助或 Joseph Davies 的著作《*Understanding IPv6*》(微软出版社，2002)。

3.2.1 寻址

IPv4 和 IPv6 之间最显著的区别是，IPv6 地址是 128 位的，其大小是 IPv4 地址的四倍。使用这样大的地址空间，其中一个原因是需要将可用地址细分为一个路由域体系。路由域反映了 Internet 的布局。表 3.2 列出了地址空间的部分分配方法，同时也给出了每个部分的地址前缀。地址前缀指示了 IPv6 地址的高阶位。IPv6 寻址在 RFC2373 中有所叙述。

表 3.2　IPv6 地址分配

分　配	地址前缀	地址空间分数
保留地址	0000 0000	1/256
为 NSAP 分配保留的地址	0000 001	1/128
可聚合全球单播地址	001	1/8
链接—本地单播地址	1111 1110 10	1/1024
站点—本地单播地址	1111 1110 11	1/1024
多播地址	1111 1111	1/256

典型的 IPv6 地址由 16 位字节分段表示，显示为冒号分隔的十六进制数。下面是一个 IPv6 地址示例：

```
21DA:00D3:0000:2F3B:02AA:00FF:FE28:9C5A
```

每个 16 位块内部前面的零都可以去掉，如下所示：

```
21DA:D3:0:2F3B:2AA:FF:FE28:9C5A
```

很多 IPv6 地址包含一长串零，可以压缩这些零块，用两个冒号代替。例如，下述地址：

```
FE80:0:0:0:12:0:34:56
```

可以压缩为：

```
FE80::12:0:34:56
```

注意，只有单个邻近的一系列 16 位零块才可以压缩。

根据所在平台的不同，可以使用两种方法来获取分配到一个计算机上各个接口的 IPv6 地址列表。从网上下载的“微软研究”及 Windows 2000 技术预览版，以及 Windows XP 家庭版和 Windows XP 专业版，都使用 IPV6.EXE 命令。要列举 IPv6 接口，在启动命令行提示符之后执行 IPV6.EXE 即可。对所有版本的 Windows 2000 和 Windows XP(包括未来的 Windows)版本，也可以使用 NETSH.EXE 命令。这个命令的语法为：“NETSH.EXE 接口 IPv6 显示接口”。要编程获取本地接口的配置，可以使用 I/O 控制命令 SIO_ADDRESS_LIST_QUERY(第 7 章)及 IP 助手 API(第 16 章)。

IPv6 地址有 3 种基本类型：单播、任播和多播。应注意到，IPv6 并未定义广播地址(而是使用多播)。随后的几个小节将讨论每种地址类型。

3.2.1.1　单播

单播地址标识单个接口。然而，对于 IPv6 而言，一个接口往往分配有多个单播地址。一般会遇

到下列 4 种单播地址：

- 链接—本地地址
- 站点—本地地址
- 全球地址
- 兼容地址

每个接口始终分配有一个链接—本地地址——每个物理网络接口都自动配置了一个这样的地址。链接—本地地址用来仅和在同一个链接上的节点通信，并总是以 fe80::/64 为前缀。另外，由于链接—本地地址未持有路由信息，所以接口索引经常和地址一起显示。系统中的每个物理接口都分配到一个适配器索引号(也叫范围 ID)。当链接—本地地址被分配到某个接口时，链接号会被附加到该地址后边。下面的地址就是分配到物理适配器的一个链接—本地地址，该适配器的接口索引号为 5。

```
fe80::250:8bff:fea0:92ed%5
```

在 Winsock 中，如果使用链接—本地地址建立了一个连接，则必须给出接口索引，以指明从哪个链接可以访问远程主机。IPv6 链接—本地地址和本章前面讨论的 IPv4 APIPA 地址同义。

例如，考虑具有链接—本地地址 fe80::250:8bff:fea0:92ed%5 的主机 A 与具有链接—本地地址 fe80::250:daff:fec3:9e34%4 的主机 B。如果主机 A 发出一个到主机 B 的连接，则它会将自己的能够到达主机 B 的范围 ID 和 B 的目标地址一起使用。即要连接的地址是：fe80::250:daff:fec3:9e34%5。

IPv6 地址中，只有在本地网络环境中才能抵达地点—本地地址，例如在一个特定地点的公司网络中。因为不能从其他地点或 Internet 中访问这些地址，而且专用网络中的路由器不会将通信转发到本地地点之外，所以它们和 IPv4 专用地址空间相似。地点—本地地址使用 fec0::/48 作为前缀，必须用 IPv6 路由器或通过 DHCPv6 进行分配。目前，微软实现的 IPv6 还不支持 DHCPv6。支持 IPv6 的路由器将发送 RA(Router Advertisement，路由器广告)信息，这些信息将公布该地址的网络部分(例如，地址的前 64 位由 48 位地点—本地前缀与 16 位子网 ID 组成)，然后主机将根据这些信息把地点—本地地址分配给接收到 RA 的接口。

全球地址就是在 IPv6 互联网中都可访问的地址。该地址以 001 开始，前 64 位中的其余 61 位用来建立路由体系，后 64 位组成接口的标识符，用以惟一标识子网中的网络接口。全球地址也通过路由器广告或使用 DHCPv6 进行分配。

最后一种单播地址是兼容地址，用来支持从 IPv4 到 IPv6 的转换。Windows 支持 4 种兼容地址：ISATAP(Intrasite Automatic Tunnel Address-ing Protocol，现场自动隧道寻址协议)地址、6 到 4 地址、6 跨 4 地址以及 IPv4 兼容地址。ISATAP 地址可以由任何 IPv6 单播地址(如链接—本地地址、站点—本地地址及全球地址等)派生而得。通常情况下，ISATAP 地址由链接—本地地址派生而得。这种地址也包括嵌入式 IPv4 地址。例如，ISATAP 地址 fe80::5efe:172.17.7.2 是一个链接—本地地址，它包含了主机的 IPv4 地址(172.17.7.2)。当数据从这个接口送出时，IPv6 数据包被封装到一个 IPv4 报头中。

IPv4 目的地址从嵌入到 IPv6 ISATAP 目标地址的 v4 地址中获取。v4 地址必须是全球可访问的，以便两个端点能够通过自动隧道进行通信。目前，ISATAP 地址是一项 IETF(Internet Engineering Task Force，Internet 工程任务组)草案。

第 2 种兼容地址称为“6 到 4”，在 RFC3056 中有所叙述。6 到 4 地址使用全球前缀 2002:WWXX:YYZZ::/48，其中 WWXX:YYZZ 是 w.x.y.z(一种公共 IPv4 地址)的十六进制—冒号表示法。6 到 4 地址允许 IPv6/IPv4 主机之间通过 IPv4 路由结构进行通信。

Windows XP 支持 6 到 4 服务。这项服务不仅允许主机和使用 6 到 4 中继路由器的 IPv6 网络中的主机通信，还允许主机和同一站点的 6 到 4 主机、连接到 Internet 的 6 到 4 主机以及跨越 IPv4 网络的其他站点的主机通信。在 Windows XP 中，6 到 4 服务被配置为自动运行。如果某个接口分配有公共 IPv4 地址，则系统将创建一个 6 到 4 隧道接口(接口索引为 3)，并把 6 到 4 地址分配给该隧道接口。

第 3 种兼容地址是“6 跨 4”，这是一种使用 IPv4 多播的隧道技术。它使得 IPv4 和 IPv6 节点可以在 IPv4 下层结构之上使用 IPv6 进行通信。这种技术在 RFC2529 中有所叙述。

最后一种兼容地址是 IPv4 兼容地址。这种地址表现形式为 0:0:0:0:0:0:w.x.y.z(或::w.x.y.z)，其中 w.x.y.z 是公共 IPv4 地址的点分十进制表示。当应用程序使用 IPv4 兼容地址作为目的地址时，IPv6 的通信量将被自动封装到 IPv4 报头中，并通过 IPv4 网络送到目的地。

3.2.1.2　任播

任播是标识多接口的地址。使用这些地址的目的，是将指向一个任播地址的数据包路由到最近的分配有该地址的接口。当网络中存在几个节点提供某种特定的服务时，使用任播地址比较有利。每个计算机都可以分配同一个任播地址，有意联系这种服务的客户机将被路由到最近的提供服务的成员处。因为这种通信是一个对多个之中的一个，而不是一个对多个，所以它和多播不同。不过，目前任播地址仅分配于路由器。

3.2.1.3　多播

IPv6 中的多播和 IPv4 中的类似。只要进程在某个特定接口加入多播组，就可收到发往该多播地址的数据。IPv6 多播地址以 1111 1111(FF)作为开始部分。IPv6 多播和 IPv6 多播地址在第 9 章中有详细叙述。

3.2.2　IPv6 管理协议

IPv6 只需要一个助手协议：ICMPv6(Internet Control Message Protocol for IPv6，IPv6 Internet 控制信息协议)。RFC2463 中有对这个协议的定义。ICMPv6 提供和 ICMP 同种类型的服务，如：目标地址无法访问、回应及回应答复，但它还提供 MLD(Multicast Listener Discovery，多播监听者搜寻)机制和 ND(Neighbor Discovery，近邻发现)机制。在 ICMPv6 中，MLD 代替了 IGMP，ND 代替了 ARP。

3.2.3 Winsock 中的 IPv6 寻址

Winsock 应用程序中指定 IPv6 地址时使用下述结构：

```
struct sockaddr_in6 {
    short sin6_family;
    u_short sin6_port;
    u_long sin6_flowinfo;
    struct in6_addr sin6_addr;
    u_long sin6_scope_id;
};
```

第 1 个字段仅标识了地址族，该地址族为 AF_INET6，第 2 个字段是端口号。这个结构中的所有字段都必须按照网络字节顺序排列。应注意到，因为端口号是封装协议的一个属性，所以 IPv4 部分中讨论的所有有关端口号的知识同样适用于 IPv6，如 TCP 和 UDP 在 IPv6 中两者都可用。第 3 个字段 sin6_flowinfo 用于为连接标记通信量，但未在微软 IPv6 堆栈中使用。第 4 个字段是一个包含了二进制 IPv6 地址的 16 字节结构。最后一个字段 sin6_scope_id 指明地址所在的接口索引(或范围 ID)。应记住，对于链接—本地地址，必须指定目的地所在的本地范围 ID，sin6_scope_id 就用于这个目的。站点—本地地址可以引用地址号码作为范围 ID。全球地址不包含范围 ID。

最后应该注意的一个问题是，SOCKADDR_IN6 结构的长度为 28 字节，而 SOCKADDR 结构和 SOCKADDR_IN 结构的长度仅为 16 字节。

3.3 地址及名称解析

本节将讲述怎样分配两种 IP 的文字串地址，以及将两种 IP 的名称解析为地址专用结构。首先讲述新的名称解析 API：getaddrinfo 和 getnameinfo，这两个 API 代替了 IPv4 专用例程。然后讲述用于文字地址和套接字地址结构之间转换的常规 Winsock API，即 WSAAddressToString 和 WSAStringToAddress。注意，这些函数仅执行地址转换和分配，不执行名称解析。

接着将叙述传统 IPv4 专用例程。其中包括传统 API 的说明，以应付有必要保留原有代码的场合，不过任何新的工程项目都应该使用新的 API 函数。使用新的函数后，编写一个在 IPv4 和 IPv6 上都能够无缝运行的应用程序就比较容易了，这将是本章下一节的主题。

最后应注意到，本章叙述的所有名称解析函数仅仅解析名称，不能将名称注册为一个地址。这项功能由第 8 章中讨论的 Winsock RNR(Registration and Name Resolution，注册及名称解析)API 来完成。

3.3.1 名称解析例程

有几个随 IPv6 一起引进的新的名称解析函数能够处理 IPv4 和 IPv6 地址。原有的函数如

gethostbyname 和 inet_addr 只能处理 IPv4 地址。替代它们的函数名为 getnameinfo 和 getaddrinfo。

这两个新的名称解析函数在 WS2TCPIP.H 中定义。同时要注意到，尽管这些函数是 Windows XP 中的新函数，但它们也能在所有支持 Winsock 2 的平台上运行，这只要在包含头文件 WS2TCPIP.H 之前，包含头文件 WSPIAPI.H 即可。编译过的二进制文件则可以在所有支持 Winsock 2 的平台上运行，如 Windows 95、Windows 98、Windows Me、Windows NT 4.0 及 Windows 2000。

getaddrinfo 函数提供独立于协议的名称解析。函数原型为：

```
int getaddrinfo(
    const char FAR *nodename,
    const char FAR *servname,
    const struct addrinfo FAR *hints,
    struct addrinfo FAR *FAR *res
);
```

参数 nodename 指定以空字符结束的主机名和文字地址。servname 参数是一个包含端口号或服务名(如 ftp 或 telnet)的以空字符结束的字符串。第 3 个参数 hints 是一个结构，它能够传递一个或多个选项，这些选项将影响到名称解析的执行方式。最后，参数 res 返回 addrINFO 结构的一个链表，该结构包含了由字符串名称解析而来的地址。如果操作成功，则返回 0；否则返回 Winsock 错误信息。

addrINFO 结构的定义为：

```
struct addrinfo {
    int ai_flags;
    int ai_family;
    int ai_socktype;
    int ai_protocol;
    size_t ai_addrlen;
    char *ai_canonname;
    struct sockaddr *ai_addr;
    struct addrinfo *ai_next;
};
```

当要把 hints 结构传递到 API 时，结构应预先置零，且结构的前 4 个字段应相互关联。

- ai_flags 字段表示下述 3 个值中的一个：AI_PASSIVE、AI_CANONNAME 或 AI_NUMERICHOST。AI_PASSIVE 表示 nodename 是一个计算机名(如 www.microsoft.com)，AI_NUMERICHOST 表示 nodename 是一个文字字符串地址(如“10.10.10.1”)。稍后将讨论 AI_PASSIVE。
- ai_family 字段表示 AF_INET、AF_INET6 或 AF_UNSPEC。如果想解析到一个具体地址，请键入 AF_INET 或 AF_INET6。否则，如果给定 AF_UNSPEC，则返回地址可能是 IPv4，也可能是 IPv6，或者两个都返回。

- ai_socktype 字段指定要求的套接字类型，如 SOCK_DGRAM 或 SOCK_STREAM。当 servname 包含服务名称时，这个字段就会被使用。也就是说，根据使用的是 UDP，还是 TCP，有些服务具有不同的端口号。
- ai_protocol 字段指定要求的协议，如 IPPROTO_TCP。跟上面一样，当 servname 包含服务名称时，这个字段就有用。

如果没有 hints 结构传递给 getaddrinfo，函数将根据零 hints 结构执行，这时 ai_family 取为 AF_UNSPEC。

如果函数解析名称成功，则解析后的地址会通过 res 返回。如果名称被解析为多个地址，则返回结果为一个由 ai_next 字段形成的链表。每个由名称解析而来的地址在 ai_addr 中表示，其长度为 ai_addrlen 中给出的套接字地址结构的长度。可以将这两个字段直接传递给 bind、connect、sendto 等函数。

下面这一段代码示例展示了如何解析一个带有端口号的主机名，其中，解析操作是在与服务器建立一个 TCP 连接之前进行的。

```
SOCKET s;
struct addrinfo hints,
*result;
int rc;
memset(&hints,0,sizeof(hints));
hints.ai_flags = AI_CANONNAME;
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
hints.ai_protocol = IPPROTO_TCP;
rc =getaddrinfo("foobar","5001",&hints,&result);
if (rc != 0) {
//无法解析名称
}
s =socket(result->ai_family,result->ai_socktype,
result->ai_protocol);
if (s == INVALID_SOCKET) {
//套接字 API 失败
}
rc = connect(s,result->ai_addr,result->ai_addrlen);
if (rc == SOCKET_ERROR) {
//连接 API 失败
}
freeaddrinfo(result);
```

在这个示例中，应用程序试图解析主机名“foobar”，并希望在 5001 号端口与某项服务建立一个 TCP 连接。您还会注意到，代码并不在乎名称被解析为 IPv4 地址，还是 IPv6 地址。可能“foobar”

两种地址都注册了，在这种情况下，result 将包含由 ai_next 字段链接的额外的 addrINFO 结构。如果应用程序希望“foobar”仅注册了 IPv4 地址，则应将 hints.ai_family 设为 AF_INET。最后应注意，通过 res 返回的信息是动态分配的，一旦应用程序使用完这些信息之后，便需要调用 freeaddrinfo API 释放这些信息所占用的空间。

应用程序实施的另一个常见动作，是将一个文字字符串地址(如“172.17.7.1”或“fe80::1234”)分配到适当类型的套接字地址结构中。getaddrinfo 函数在 hints 结构中设置 AI_NUMERICHOST 标志，从而做到这一点。下面的代码展示了这一做法：

```
struct addrinfo hints,
*result;
int rc;
memset(&hints,0,sizeof(hints));
hints.ai_flags = AI_NUMERICHOST;
hints.ai_family = AI_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
hints.ai_protocol = IPPROTO_TCP;
rc =getaddrinfo("172.17.7.1","5001",&hints,&result);
if (rc !=0) {
//文字字符串地址无效
}
//使用 result
freeaddrinfo(result);
```

文字字符串地址“172.17.7.1”将被转换为必要的套接字地址结构，并通过 result 返回。因为传递了 AF_UNSPEC，API 将确定所需的正确套接字地址结构(SOCKADDR_IN 或 SOCKADDR_IN6)，并将地址作相应转换。如前所述，转换后生成的套接字地址结构的端口字段将被初始化为 5001。

注意，如果没有将任何标志作为 hints 结构的一部分进行传递，那么在解析文字字符串地址之后，被返回的含有转换后地址的结构 addrinfo 将设置 AI_NUMERICHOST 标志。同样，如果解析了主机名而没有传递任何 hints 结构，所返回的结构 addrinfo 将包含 AI_CANONNAME 标志。

getaddrinfo 可以使用的最后一个标志是 AI_PASSIVE，该标志用于获取能够传递给 bind 函数的地址。对于 IPv4 而言，这个地址可能是 INADDR_ANY(0.0.0.0)，而对于 IPv6 而言则可能是 IN6ADDR_ANY(::)。为获取绑定地址，hints 结构应该指明要为哪个地址族获取该无源地址(通过 ai_family)，nodename 应设为 NULL，servname 应设为非 NULL——指明应用程序要绑定到哪个端口号(可以是 0)。如果 AF_UNSPEC 被传递到 hints 结构中，则有两个 addrINFO 结构将被返回，一个带有 IPv4 绑定地址，另一个带有 IPv6 绑定地址。

在使用 getaddrinfo 解析了主机名之后，AI_PASSIVE 标志就可以派上用场了。一旦解析后的地址被返回，就可以在另一次调用 getaddrinfo 时使用初始结果中的 ai_family(地址族)，以便获取该地址族合适的绑定地址。这让应用程序避免了接触内部套接字地址结构的字段，也排除了使用两个独立的代

码路径(即根据地址将解析到哪个地址族来绑定套接字的两个独立路径)的必要。

另一个新的名称解析 API 是 getnameinfo，其功能和 getaddrinfo 函数相反。该函数接受已经初始化的套接字结构，并返回与地址及端口信息对应的主机和服务名。函数原型为：

```
int getnameinfo(
    const struct sockaddr FAR *sa,
    socklen_t salen,
    char FAR *host,
    DWORD hostlen,
    char FAR *serv,
    DWORD servlen,
    int flags
);
```

其中各个参数的含义是较为明显的。参数 sa 是套接字地址结构，名称信息将从这个参数中获取；而参数 salen 则是该地址结构的大小。参数 host 是接收主机名称的字符缓冲区，默认状态下将返回 FQDN(fully qualified domain name，完全合格的域名)。参数 hostlen 表示主机缓冲区的大小。参数 serv 是接收服务(或端口)信息的字符缓冲区，而参数 servlen 则表示该缓冲区的长度。最后，flags 参数指明将如何解析套接字地址。flags 可能的取值如下：

- NI_NOFQDN：表明仅返回 RDN(relative distinguished name，相对特异名)。例如，设置这个标志时，名为“mist.microsoft.com”的主机将只返回“mist”。
- NI_NUMERICHOST：表明将返回用字符串表示的地址，而不返回主机名。
- NI_NAMEREQD：表明如果地址不能被解析为 FQDN，则返回错误信息。
- NI_NUMERICSERV：表明将端口信息作为一个字符串返回，而不会将端口信息解析为一个已知的服务名(如“ftp”)。注意，如果在提供 serv 缓冲区时缺少这个标志，且端口号不能被解析为某种已知服务，则 genameinfo 将失败，出错信息为 WSANO_DATA(11004)。
- NI_DGRAM：用来将数据报服务从流服务中区分开来。对于少数几个为 UDP 和 TCP 定义了不同端口号的服务来说，这种区分是很有必要的。

在补充材料中有一个名为 RESOLVER.CPP 的文件，该文件具有执行名称解析的功能。主机名或地址随同服务或端口信息一起被传递给应用程序，并通过 getaddrinfo 函数进行解析。然后，针对每个返回的已解析地址，调用函数 getnameinfo，以获得计算机名，或获取字符串文字地址。

3.3.2 简单的地址转换

如果应用程序只是需要在字符串文字地址和套接字地址结构之间进行转换，可以使用 WSAStringToAddress 和 WSAAddressToString 这两个 API。使用 WSAStringToAddress 时，必须指定字符串地址所属的地址族，所以它不如 getaddrinfo“灵活”。这个 API 如下所示：

```
INT WSAStringToAddress(
```

```
    LPTSTR AddressString,
    INT AddressFamily,
    LPWSAPROTOCOL_INFO lpProtocolInfo,
    LPSOCKADDR lpAddress,
    LPINT lpAddressLength
);
```

第 1 个参数是需要转换的字符串，第 2 个参数指明该字符串所属的地址族(如 AF_INET、AF_INET6 或 AF_IPX)。第 3 个参数 lpProtocolInfo 是一个可选指针，它指向 WSAPROTOCOL_INFO 结构，这个结构定义了实施转换时所使用的协议提供程序。如果同时有多个提供程序执行一种协议，则这个参数可以用来指定一个显式的提供程序。第 4 个参数是一个适当的套接字地址结构，可以将字符串地址转换为这种地址结构，并把值赋给它。

应注意到，这个 API 可以转换包含端口号的字符串地址。例如，IPv4 字符串的注释允许在地址末尾的冒号后边紧跟端口号。例如，“157.54.126.42:1200”表示这个 IPv4 地址使用 1200 端口。在 IPv6 中，IPv6 地址串必须用方括号括起来，在方括号后边可以使用冒号和端口注释。例如，[fe80::250:8bff:fca0:92ed%5]:80 表示一个链接一本地地址，其范围 ID 后紧跟端口号 80。注意，只有端口号才会被解析，服务名(如 ftp)不会被解析。对这两个例子来说，如果使用 WSAStringToAddress 来转换这些字符串，则返回的套接字地址结构将被初始化，从而获得适当的二进制 IP 地址、端口号以及地址族。对 IPv6 而言，如果字符串地址的地址部分后边含有“%scope_ID”，则范围 ID 字段也会被初始化。

WSAAddressToString 提供从套接字地址结构到该地址字符串表示的一个映射。其原型为：

```
INT WSAAddressToString(
    LPSOCKADDR lpsaAddress,
    DWORD dwAddressLength,
    LPWSAPROTOCOL_INFO lpProtocolInfo,
    LPTSTR lpszAddressString,
    LPDWORD lpdwAddressStringLength
);
```

这个函数采用了一个 SOCKADDR 结构，并将二进制地址格式化为一个字符串，该字符串由 lpszAddressString 缓冲区表示。如果一个给定协议有多个传输提供程序，那么将该协议的 WSAPROTOCOL_INFO 结构作为 lpProtocolInfo 传递，便可选定一个具体的提供程序。注意，地址族是结构 SOCKADDR 的一个字段，该结构为 lpsaAddress 传递。

3.3.3　传统名称解析例程

因为新的应用程序可以使用 getaddrinfo 和 getnameinfo，所以本节讨论传统名称解析的目的，只是为了方便代码维护。还应该注意到，这两个新的 API 调用能够完成 8 个传统函数所具有的功能。

函数 inet_addr 把一个点分 IPv4 地址转换为一个 32 位无符号长整数，其定义为：

```
unsigned long inet_addr(
    const char FAR *cp
);
```

cp 字段是一个以空字符结束的字符串，它认可点分表示法的 IP 地址。注意，这个函数返回的 IPv4 地址是一个按网络字节顺序排列的 32 位无符号长整数，该长整数被分配给 SOCKADDR_IN 的 sin_addr 字段。网络字节顺序已在第 1 章中作了叙述。

与 inet_addr 相对的是 inet_ntoa，它获取一个 IPv4 网络地址，并将其转换为一个字符串。该函数的声明为：

```
char FAR *inet_ntoa(
    Struct in_addr in
);
```

下面的示例代码展示了如何使用 inet_addr 函数和 htons 函数创建一个 SOCKADDR_IN 结构。

```
SOCKADDR_IN InternetAddr;
INT nPortId = 5150;
InternetAddr.sin_family = AF_INET;
//将拟采用的点分 Internet 地址 136.149.3.29 转换为 4 字节整数，并把它分配给 sin_addr
InternetAddr.sin_addr.s_addr = inet_addr("136.149.3.29");
//nPortId 变量按主机字节顺序存储。将 nPortId 转换为网络字节顺序，并分配给 sin_port
InternetAddr.sin_port =htons(nPortId);
```

gethostbyname、WSAAsyncGetHostByName、gethostby-addr 和 WSAAsyncGetHostByAddr 这 4 个 Winsock 函数从主机数据库中获取与主机名或主机地址相对应的主机信息。前两个函数将主机名转换为其网络的 IPv4 地址，后两个函数则完成相反的操作——将 IPv4 网络地址映射回主机名。这些函数均返回 HOSTENT 结构，该结构的定义为：

```
struct hostent
{
    char FAR *h_name;
    char FAR *FAR *h_aliases;
    short h_addrtype;
    short h_length;
    char FAR *FAR *h_addr_list;
};
```

h_name 字段是主机的正式名称。如果网络使用了 DNS，则 FQDN 将促使名称服务器返回一条回复。如果网络使用一个本地“主机”文件，则该文件名称就是 IP 地址后的第一个条目。h_aliases 字段是一个以空字符结束的数组，表示主机的替代名称。h_addrtype 字段表示返回的地址族。h_length 字段定义 h_addr_list 字段中每个地址的字节长度。对 IPv4 地址而言，地址长度为 4 个字节。h_addr_list

字段是表示主机 IP 地址的一个以空字符结束的数组(可以为一个主机分配多个 IP 地址)，数组中的每个地址均以网络字节顺序返回。

一般情况下，应用程序使用数组中的第 1 个地址。然而，如果同时有多个地址被返回，则应用程序通常会随机选择一个可用的地址，而不会总是使用第 1 个。

这些函数的定义如下：

```
struct hostent FAR *gethostbyname (
    const char FAR *name
);

HANDLE WSAAsyncGetHostByName(
    HWND hWnd,
    unsigned int wMsg,
    const char FAR *name,
    char FAR *buf,
    int buflen
);

struct HOSTENT FAR *gethostbyaddr(
    const char FAR *addr,
    int len,
    int type
);

HANDLE WSAAsyncGetHostByAddr(
    HWND hWnd,
    unsigned int wMsg,
    const char FAR *addr,
    int len,
    int type,
    char FAR *buf,
    int buflen
);
```

对前两个函数而言，name 参数显示用户正在寻找的主机的一个友好名称。后两个函数将获取一个 IPv4 网络地址并将它分配给参数 addr，同时将地址长度指定为 len。type 指明操作过程中所传递的网络地址的地址族，其取值应为 AF_INET。这 4 个函数均通过一个 HOSTENT 结构返回结果。对两个同步函数(第 1 个和第 3 个)来说，HOSTENT 是一个系统分配的缓冲区，由于这个缓冲区是静态的，所以应用程序不能依赖它。而两个异步函数(第 2 个和第 4 个)将会把 HOSTENT 结构复制到由 buf 参数表示的缓冲区中，这个缓冲区大小应该等于 MAXGETHOSTSTRUCT。

最后，这些函数以及其他异步的名称和服务解析函数均返回一个 HANDLE，并用它来标识已发

动的操作。一旦操作完成，就会有一个由 wMsg 指明的窗口消息被发送到由 hWnd 给定的窗口。如果在某一时刻应用程序希望取消异步请求，则应该使用 WSACancelAsyncRequest 函数，其声明为：

```
int WSACancelAsyncRequest(
    HANDLE hAsyncTaskHandle
);
```

应牢记，同步 API 调用只有在查询结束或超时的情况下，才会发生阻塞，这可能会花费几秒钟的时间。

下一种类型的传统名称解析函数能够找出已知服务的端口号，也能够根据端口号找出服务项目。API 函数 getservbyname 和 WSAAsyncGetServByName 获取一个已知服务的名称(如 ftp)，之后返回该服务所使用的端口号。函数 getservbyport 和 WSAAsyncGetServByPort 则执行相反的操作，接收端口号，返回使用该端口的服务名。这些函数只简单地从一个文件名为“services”的文件中获取静态信息。在 Windows 95、Windows 98 及 Windows Me 中，services 文件位于%WINDOWS%目录下；在 Windows NT 中，该文件位于%WINDOWS%\System32\Drivers\Etc 下。

这 4 个函数用 SERVENT 结构返回服务信息。该结构的定义为：

```
struct servent {
    char FAR *s_name;
    char FAR *FAR *s_aliases;
    short s_port;
    char FAR *s_proto
};
```

字段 s_name 是服务名，字段 s_aliases 是表示字符串指针的一个以空字符结束的数组，数组中每个元素分别包含了服务的一个别名。s_port 是服务使用的端口号，s_proto 是服务使用的协议，如字符串“tcp”和“udp”。

这些函数的定义如下：

```
struct servent FAR *getservbyname(
    const char FAR *name,
    const char FAR *proto
);

HANDLE WSAAsyncGetServByName(
    HWND hWnd,
    unsigned int wMsg,
    const char FAR *name,
    const char FAR *proto,
    char FAR *buf,
    int buflen
);
```

```
struct servent FAR *getservbyport(
    int port,
    const char FAR *proto
);

HANDLE WSAAsyncGetServByPort(
    HWND hWnd,
    unsigned int wMsg,
    int port,
    const char FAR *proto,
    char FAR *buf,
    int buflen
);
```

name 参数表示所寻找的服务的名称。proto 参数有选择地指向一个字符串，该字符串指明 name 参数表示的服务是在哪个协议下注册的，如“tcp”和“udp”协议。后两个函数简单地接受端口号，并用它来和某服务名相匹配。两个同步的 API 函数将返回一个 SERVENT 结构，该结构是由系统分配的缓冲区。而两个异步的函数将采用一个由应用程序提供的缓冲区，其大小也应为 MAXGETHOSTSTRUCT。

最后一组传统的名称解析 API 函数在协议的字符串名称(如 tcp)与其协议号(tcp 将被解析为 IPPROTO_TCP)之间执行转换操作。这些函数有：getprotobyname、WSAAsyncGetProto-ByName、getprotobynumber 及 WSAAsyncGetProtoByNumber。前两个函数将字符串协议转换为协议号，而后两个则相反——将协议号映射回它的字符串名称。这些函数均返回一个 PROTOENT 结构，该结构的定义为：

```
struct protoent {
    char FAR *p_name;
    char FAR *FAR *p_aliases;
    short p_proto;
};
```

第 1 个字段 p_name 是协议的字符串名称，p_aliases 是表示字符串指针的一个以空字符结束的数组，该数组包含了协议的其他已经为人所知的名称。最后一个字段 p_proto 是协议号(如 IPPROTO_UDP 或 IPPROTO_TCP)。

这些函数的原型为：

```
struct protoent FAR *getprotbyname(
    const char FAR *name
);

HANDLE WSAAsyncGetProtoByName(
```

```
    HWND hWnd,
    unsigned int wMsg,
    const char FAR *name,
    char FAR *buf,
    int buflen
);

struct protoent FAR *getprotobynumber(
    int number
);

HANDLE WSAAsyncGetProtoByNumber(
    HWND hWnd,
    unsigned int wMsg,
    int number,
    char FAR *buf,
    int buflen
);
```

在同步及异步方面，这些函数和前述传统名称解析函数表现一致。

3.4 编写独立于 IP 版本的程序

如何在 IPv4 和 IPv6 上开发无缝运行的应用程序呢？本节将就此展开讨论。要做到这一点，首先要使用新的名称解析 API 函数 getaddrinfo 和 getnameinfo，还需重新调整 Winsock 的调用方式。

在介入具体内容之前，先要注意一些应该遵循的基本原则。首先，因为套接字地址结构的长度可能不相等，所以应用程序不能将这些结构具体分配到每种协议中(如将 SOCKADDR_IN 和 SOCKADDR_IN6 分别分配到 IPv4 和 IPv6 中)。针对这种情况，这里引进了一个新的套接字地址结构 SOCKADDR_STORAGE，它的大小和最大的协议专用地址结构相同。同时，这个结构还具有针对 64 位对齐问题的填充项。下列代码便使用了 SOCKADDR_STORAGE 结构来储存目标 IPv6 地址。

```
SOCKADDR_STORAGE  saDestination;
SOCKET            s;
int               addrlen,
                  rc;


s = socket(AF_INET6,SOCK_STREAM,IPPROTO_TCP);
if (s == INVALID_SOCKET) {
//套接字失败

}
```

```
addrlen = sizeof(saDestination);
rc = WSAStringToAddress(
    "3ffe:2900:d005:f28d:250:8bff:fea0:92ed",
    AF_INET6,
    NULL,
     (SOCKADDR *)&saDestination,
    &addrlen
    );
if (rc == SOCKET_ERROR) {
    //转换失败
}
rc = connect(s,(SOCKADDR *)&saDestination,sizeof(saDestination));
if (rc == SOCKET_ERROR) {
    //连接失败
}
```

其次，采用地址作为参数的函数应该传递整个套接字地址的结构，而不是传递协议专用的类型(如 structin_addr 或 structin6_addr)。对于 IPv6 而言，因为它需要范围 ID 信息，以使进行成功的连接，这一点就显得非常重要。所以，应该传递包含有地址的 SOCKADDR_STORAGE 结构。

最后，避免使用硬编码(hardcode)地址，不管这些地址是 IPv4，还是 IPv6。Winsock 头文件为所有硬编码地址(如用于绑定的环回地址和通配符地址)定义了常量。

弄清楚一些基本问题之后，下面开始讨论怎样组织一个应用程序，才能使其独立于 IP。讨论分为两个部分：客户机和服务器。

3.4.1　客户机

对 TCP 及 UDP 客户机来说，应用程序通常拥有服务器(或接受者)的 IP 地址或主机名。至于解析到 IPv4 地址，还是解析到 IPv6 地址，这一点并不重要。客户机应遵循下面的 3 个步骤：

1. 使用 getaddrinfo 函数解析地址。hints 结构不仅要包含套接字类型和协议，还应该包含 AF_UNSPEC。这取决于客户机是使用 TCP 通信，还是使用 UDP 通信。
2. 使用步骤 1 返回的 addrINFO 结构中的 ai_family、ai_socktype 和 ai_protocol 字段来创建套接字。
3. 使用 addrINFO 结构中的成员 ai_addr 来调用 connect 函数或 sendto 函数。

下列示例代码展示了上述规则。

```
SOCKET s;
struct addrinfo hints,
      *res = NULL
char *szRemoteAddress = NULL,
```

```
        *szRemotePort = NULL;
int rc;


//分析命令行，以获取远程服务器的主机名、地址及端口号，它们位于
//  szRemoteAddress 和 szRemotePort 中。
memset(&hints,0,sizeof(hints));
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
hints.ai_protocol = IPPROTO_TCP;
//首先在假定字符串是一个文字字符串地址的情况下进行解析
rc = getaddrinfo(
        szRemoteAddress,
        szRemotePort,
    &hints,
    &res
    );
if (rc == WSANO_DATA) {
        //无法解析名称—跳出
    }
s =socket(res->ai_family,res->ai_socktype,res->ai_protocol);
if (s == INVALID SOCKET) {
        //套接字失败
}
rc = connect(s,res->ai_addr,res->ai_addrlen);
if (rc == SOCKET_ERROR) {
        //连接失败
}
freeaddrinfo(res);
```

首先，应注意到，这里没有显式地引用 AF_INET 或 AF_INET6。同时，也没有必要操作下层的 SOCKADDR_IN 或 SOCKADDR_IN6 地址。getaddrinfo 调用将对返回的套接字地址结构进行完全初始化，使其获得所有必需的信息——地址族、二进制地址等，这对于连接或发送数据报而言，是非常必要的。

如果客户机应用程序需要在套接字创建之后，并要在调用 connect 或 sendto 之前，将套接字显式地绑定到一个本地端口，则可再次调用 getaddrinfo。这个调用将指定从第一次调用中返回的地址族、套接字类型、协议、AI_PASSIVE 标志以及所需的本地端口，并返回另一个套接字地址结构。这个返回的结构已被本次调用初始化，从而获得了必要的绑定地址(如 IPv4 的 0.0.0.0，以及 IPv6 的::)。

3.4.2 服务器

因为 Windows IPv6 堆栈是双堆栈，所以对服务器端进行编程比客户机端的编程牵涉的层面更多。

IPv4 和 IPv6 具有各自独立的堆栈，因此，如果一个服务器希望同时能够接受 IPv4 和 IPv6 连接，它就必须为 IPv4 和 IPv6 都创建套接字。要创建一个独立于 IP 的服务器，须遵循如下步骤：

1. 用 hints 结构调用 getaddrinfo 函数，该结构包含 AI_PASSIVE、AF_UNSPEC、所需的套接字类型、协议及所需的本地端口(这个端口用来监听或接收数据)。这个调用将返回两个 addrINFO 结构：一个包含用于 IPv4 的监听地址，另一个包含用于 IPv6 的监听地址。
2. 针对每个返回的 addrINFO 结构，创建一个包含 ai_family、ai_socktype 和 ai_protocol 字段的套接字，接着使用 ai_addr 和 ai_addrlen 成员调用函数 bind。

下述代码展示了具体的程序编写规则：

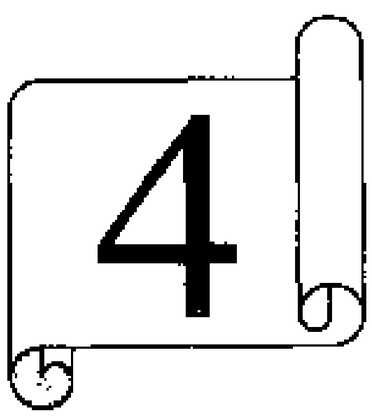
```
SOCKET slisten[16];
char *szPort = "5150";
struct addrinfo hints,
     *res = NULL,
     *ptr = NULL;
int count = 0,
     rc;
memset(&hints,0,sizeof(hints));
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
hints.ai_protocol = IPPROTO_TCP;
hints.ai_flags = AI_PASSIVE;
rc = getaddrinfo(NULL,szPort,&hints,&res);
if (rc ! = 0) {
       //因某种原因而失败
}
ptr = res;
while (ptr)
{
slisten[count]=socket(ptr->ai_family,
ptr->ai_socktype,ptr->ai_protocol);
if (slisten[count]==INVALID_SOCKET) {
    //套接字失败
}
rc = bind(slisten[count0,ptr->ai_addr,ptr->ai_addrlen);
if (rc == SOCKET_ERROR) {
    //绑定失败
}
rc = listen(slisten[count],7);
if (rc == SOCKET_ERROR) {
    //监听失败
}
count++;
```

```
        ptr = ptr->ai_next;
    }
```

创建并绑定套接字之后，应用程序需要等待与每个套接字建立的连接。第 5 章讨论了 Winsock 中可用的各种 I/O 模型，并提供了一些功能全面的客户机和服务器示例，这些示例都是依据本节叙述的规则编写的。

3.5 小　　结

本章讨论了 IPv4 和 IPv6，其中不仅叙述了每种地址族所必需的 Winsock 数据结构，还叙述了寻址及名称解析。在介绍完新的名称解析函数之后，本章接着讲述传统的名称解析函数。最后，本章还叙述了如何编写能在 IPv4 和 IPv6 上无缝运行的应用程序。第 4 章将讨论其余从 Winsock 中可以访问的协议，包括 IPX/SPX、AppleTalk、IrDA 和 ATM。



Winsock 支持的其他协议

要通过 Winsock 建立通信，就必须清楚如何用特定的协议为工作站寻址。第 3 章分别介绍了如何使用 IPv4 和 IPv6 地址族为工作站寻址。本章将讨论 Winsock 支持的其他协议——IrDA、IPX/SPX、NetBIOS、AppleTalk 及 ATM，还将讲述这些协议如何把指定家族的地址解析为网络上实际的计算机。本章只讲解一些构建各协议地址结构所需的基本知识。第 8 章将讨论注册和名称解析函数(这些函数公布给定协议族的服务)，以及直接名称解析、服务的公布与解析之间的差别。

在讲述各个地址族时，本章将首先探讨为网络中的计算机寻址的基本知识；然后讲述如何为每个地址族创建套接字；另外还将讨论协议特有的名称解析选项。补充材料为每个协议提供了一些基本的客户机和服务器示例。

4.1 红外线套接字

红外线套接字(infrared socket，IrSock)是一项新技术，它首先出现于 Windows CE 平台，目前主要运用于 Windows 98、Windows Me、Windows 2000 及 Windows XP。红外线套接字允许两台计算机通过红外线串行端口实现通信。红外线套接字不同于传统意义上的套接字，它的设计充分考虑了便携式计算设备的短暂性，并展示了一种新式的名称解析模型。下面将详细讲述红外线套接字。

4.1.1 寻址

传统解析方法假定的应用对象是诸如名称服务器之类的静态资源，但这些资源不能被正在运行网络客户任务的手提式电脑(或膝上型计算机)等移动设备使用。大多数带有 IrDA(Infrared Data Association，红外线数据联盟)设备的计算机都需要经常移动，因此使用传统的名称解析方案很难奏效。为了解决这个问题，IrDA 没有使用标准的 Winsock 命名服务函数和 IP 寻址(命名服务已并入通信流，并引入了一个新的地址族以支持与红外线串行端口绑定在一起的服务)，它用一种特殊方式浏览一定范围内(而不是整个大型网络内)的资源。IrSock 地址结构中包含一个服务名(它对 bind 和 connect

调用中所使用的应用程序进行描述)和一个设备标识符(描述运行服务的设备)。这里的服务名和设备标识符类似于传统 TCP/IP 套接字所用的 IP 地址和端口号字元组。IrSock 地址结构的定义如下:

```
typedef struct _SOCKADDR_IRDA
{
    u_short irdaAddressFamily;
    u_char irdaDeviceID[4];
    char irdaServiceName[25];
} SOCKADDR_IRDA, *PSOCKADDR_IRDA , FAR * LPSOCKADDR_IRDA;
```

irdaAddressFamily 字段总是被设置为 AF_IRDA。irdaDeviceID 是一个包含 4 个字符的字符串,它能惟一标识运行特定服务的设备。在建立 IrSock 服务器时,IrdaDeviceID 字段将被忽略;但这个字段对客户机来说非常重要,它指定的是准备连接的那个 IrDA 设备。irdaServiceName 字段是服务名(应用程序要么利用这项服务对自身进行注册,要么试图与这项服务建立连接)。

4.1.2 名称解析

可以将 LSAP-SEL(Logical Service Access Point Selectors,IrDA 逻辑服务访问点选择符)或注册到 IAS(Information Access Services,信息访问服务)的服务作为寻址的基础。IAS 从 LSAP-SEL 中提取一项服务,并把它放置到用户友好的文本服务名之中,方式和互联网域名服务器把名称映射到数字化的 IP 地址相近。要成功建立一个连接,既可以用 LSAP-SEL,也可以用用户友好名。不过,用户友好名需要名称解析。因为 IrDA 服务的地址空间有一定限制,所以大多数时候,都不要使用直接的 LSAP-SEL "地址"。Windows 实现方案允许使用 LSAP-SEL 整数标识符,标识符的范围是 1 到 127。因为 IAS 服务器把 LSAP-SEL 和一个文字化的服务名关联在一起,所以从本质上说,我们可把它当作一个 WINS 服务器。

实际的 IAS 条目有 3 个重要字段:类名、属性和属性值。举个例子来说,一个服务器希望在服务名 MyServer 下对其本身进行注册,服务器通过相应的 SOCKADDR_IRDA 结构执行绑定调用时将完成该项操作。这种情况一旦发生,IAS 条目就会增加一个,该新增条目中包括类名 MyServer、属性 IrDA:TinyTP:Lsap-SEL 和属性值 3。属性值就是下一个未用过的 LSAP-SEL,这个 LSAP-SEL 是系统根据注册信息来分配的。另一方面,客户机向 Connection 调用传递一个 SOCKADDR_IRDA 结构。随后 IAS 查找开始,查找的目标是带有类名 MyServer 和属性 IrDA:TinyTP:Lsap-SEL 的那项服务。这个 IAS 查询返回的值为 3。用户可以利用 getsocketopt 调用中的套接字选项 IRLMP_IAS_QUERY 来定制自己的 IAS 查询。

如果希望完全忽略 IAS(一般不建议使用),则可为客户机准备连接的服务名或终端直接指定一个 LASP-SEL 地址。忽略 IAS 的目的只是为了能与老的 IrDA 设备(例如红外线打印机)进行通信,这样的设备不提供任何 IAS 注册。把 SOCKADDR_IRDA 结构中的服务名指定为 LSAP-SEL-xxx(xxx 代表属性值,其范围在 1 到 127 之间)就可忽略 IAS 注册和查找。对服务器而言,这样做会直接为该服务器

分配特定的 LSAP-SEL 地址(假定这个 LSAP-SEL 地址尚未使用)。而对客户机而言，这样做则会忽略 IAS 查找，并促使客户机试图马上与在指定 LSAP-SEL 上运行的任何一项服务建立连接。

4.1.3　红外线设备列举

由于红外线设备的使用范围不固定，因此，必须有一种方法能够动态地列举出特定范围内所有可用的红外线设备。这里先从 Windows CE 的实现方案和 Windows 98、Windows Me、Windows 2000 及 Windows XP 实现方案之间的几点差别谈起。Windows CE 是最早支持 IrSock 的平台，它提供少量与红外线设备有关的信息。后来，Windows 98、Windows Me、Windows 2000 及 Windows XP 也开始支持 IrSock，但它们新增了另外的“提示”信息，该信息是由 enumeration(列举)请求返回的(关于提示信息，稍后将作简要论述)。因此，Windows CE 的 AF_IRDA.H 头文件中包含的是原始的、少量的结构定义；而其他平台提供的新的头文件中包含的，则是目前支持 IrSock 的各个平台的条件结构的定义。为了保持一致，本书建议大家采用较新的 AF_IRDA.H 头文件。

列举临近红外线设备的方法是采用 getsockopt 的 IRLMP_ENUM_DEVICE 命令。DEVICELIST 结构被当作 optval 参数传递。这里有两个 SEVICELIST 结构，一个用于 Windows 98、Windows Me、Windows 2000 及 Windows XP，另一个用于 Windows CE。这两个结构的定义如下：

```
typedef struct _WINDOWS_DEVICELIST
{
    ULONG                           numDevice;
    WINDOWS_IRDA_DEVICE_INFO   Device[1];
}WINDOWS_DEVICELIST,*PWINDOWS_DEVICELIST,FAR *LPWINDOWS_DEVICELIST;

typedef struct _WCE_DEVICELIST
{
    ULONG                           numDevice;
    WCE_IRDA_DEVICE_INFO        Device[1];
}WCE_DEVICELIST,*PWCE_DEVICELIST;
```

用于非 Windows CE 平台的结构与用于 Windows CE 的结构之间的惟一区别是：非 Windows CE 中包含一个 WINDOWS_IRDA_DEVICE_INFO 结构数组，该结构数组与用于 Windows CE 的 WCE_IRDA_DEVICE_INFO 结构数组相对应。条件性的#define 指令将根据目标平台把 DEVICELIST 声明为合适的结构。同样，对于 IRDA_DEVICE_INFO 结构也有两个声明：

```
typedef struct _WINDOWS_IRDA_DEVICE_INFO
{
    u_char irdaDeviceID[4];
    char    irdaDeviceName[22];
    u_char irdaDeviceHints1;
    u_char irdaDeviceHints2;
    u_char irdaCharSet;
```

```
}WINDOWS_IRDA_DEVICE_INFO,*PWINDOWS_IRDA_DEVICE_INFO,
    FAR *LPWINDOWS_IRDA_DEVICE_INFO;

typedef struct _WCE_IRDA_DEVICE_INFO
{
    u_char irdaDeviceID[4];
    char   irdaDeviceName[22];
    u_char Reserved[2];
}WCE_IRDA_DEVICE_INFO,*PWCE_IRDA_DEVICE_INFO;
```

同样#define 指令将根据目标平台，把 IRDA_DEVICE_INFO 声明为正确的结构定义。

正如前面提到的那样，实际用于列举红外线设备的函数是带有 IRLMP_ENUM_DEVICES 选项的 getsockopt 函数。下面这一段代码，可把邻近的所有红外线设备的 ID 列举出来：

```
SOCKET sock;
DEVICELIST devList;
DWORD dwListLen = sizeof(DEVICELIST);
sock = WSASocket(AF_IRDA,SOCK_STREAM,0,NULL,0,
    WSA_FLAG_OVERLAPPED);
...
devList.numDevice = 0;
dwRet = getsockopt(sock,SOL_IRLMP,IRLMP_ENUMDEVICES,
     (char *)&devList,&dwListLen);
```

在将 DEVICELIST 结构传递给 getsockopt 调用之前，应该把 numDevice 字段设成 0。一次成功的列举会把 numDevice 字段设成一个大于 0 的值，并把 Device 字段设置为和 IRDA_DEVICE_INFO 结构数量相等的一个值。同时，在实际的应用程序中，为检查进入使用范围内的新设备，可以多次执行 getsockopt。例如，在试图进行 5 次或少于 5 次的红外线设备查找时，使用的方法很简单：把 getsockopt 调用置入循环中，同时在每一次未成功的列举之后短期调用 Sleep 函数。

在知道如何列举红外线设备之后，创建客户机或服务器时就简单多了。同级的服务器端像一个“普通”服务器，它的创建更为简单，不需要额外的步骤。创建 IrSock 服务器的常见步骤如下：

1. 创建一个套接字，其地址族为 AF_IRDA，套接字类型为 SOCK_STREAM。
2. 用服务器的服务名填写 SOCKADDR_IRDA 结构。
3. 利用套接字句柄和 SOCKADDR_IRDA 结构调用 bind。
4. 利用套接字句柄和 backlog 限制调用 listen。
5. 阻止接受客户机接入的 accept 调用。

建立客户机时，因为必须先把红外线设备列举出来，所以步骤稍微有些复杂。创建 IrSock 客户机所需步骤如下：

1. 创建一个套接字，其地址族为 AF_IRDA，套接字类型为 SOCK-STREAM。

2. 调用带有 IRLMP_ENUM_DEVICES 选项的 getsockopt 函数，列举所有可用的红外线设备。
3. 针对返回的每个设备，利用设备 ID 和准备连接的服务名填写 SOCKADDR_IRDA 结构。
4. 利用套接字句柄和 SOCKADDR_IRDA 结构，调用 connect 函数。针对步骤 3 中所填写的结构，重复步骤 4 的操作，直到某 connect 成功。

4.1.4 查询 IAS

通过两种方法可以知道一项给定的服务是否在特定的设备上运行。第一种是尝试与给定服务建立连接；另一种是在 IAS 中查询给定的服务名。两种方法都要求列举所有的红外线设备，然后对每一个设备进行查询(或尝试连接)，直到查询或连接成功，或所有的设备都查完。调用带有 IRLMP_IAS_QUERY 选项的 getsockopt 函数，便可执行查找操作。需要注意的是，IAS_QUERY 结构有两个，一个用于 Windows 98、Windows Me、Windows 2000 及 Windows XP，另一个用于 Windows CE。各结构的定义如下：

```
typedef struct _WINDOWS_IAS_QUERY
{
    u_char irdaDeviceID[4];
    char    irdaClassName[IAS_MAX_CLASSNAME];
    char    irdaAttribName[IAS_MAX_ATTRIBNAME];
    u_long irdaAttribType;
    union
    {
        LONG irdaAttribInt;
        struct
        {
            u_long Len;
            u_char OctetSeq[IAS_MAX_OCTET_STRING];
        }irdaAttribOctetSeq;
        struct
        {
            u_long Len;
            u_long CharSet;
            u_char UsrStr[IAS_MAX_USER_STRING];
        }irdaAttribUsrStr;
    }irdaAttribute;
}WINDOWS_IAS_QUERY,*PWINDOWS_IAS_QUERY,FAR *LPWINDOWS_IAS_QUERY;

typedef struct _WCE_IAS_QUERY
{
    u_char  irdaDeviceID[4];
    char    irdaClassName[61];
    char    irdaAttribName[61];
    u_short irdaAttribType;
```

```
union
{
    int irdaAttribInt;
    struct
    {
        int     Len;
        u_char OctetSeq[1];
        u_char Reserved[3];
    }irdaAttribOctetSeq;
    struct
    {
    int   Len;
    u_char CharSet;
        u_char UsrStr[1];
        u_char Reserved[2];
        }irdaAttribUsrStr;
    }irdaAttribute;
}WCE_IAS_QUERY,*PWCE_IAS_QUERY;
```

可以看到，除了特定字符数组的长度不同之外，这两个结构的定义是差不多的。

要对特定服务的 LSAP-SEL 数量进行查询，方法很简单：把 irdaClassName 字段设为 LSAP-SEL 的属性字符串，即 IrDA:IrLMP:LsapSel，然后，把 irdaAttributeName 字段设为准备查询的那个服务名。除此以外，还必须根据位于使用范围内的有效设备来设置 irdaDeviceID 字段。

4.1.5 创建套接字

红外线套接字的创建很简单。因为 IrSock 只支持面向连接的数据流，所以它很少需要选项。下面的代码说明了如何利用 socket 或 WSASocket 调用来创建红外线套接字。由于 Winsock 1.1 的限制，必须采用 Windows CE 的 socket。

```
s = socket(AF_IRDA,SOCK_STREAM,0);
s = WSASocket(AF_IRDA,SOCK_STREAM,0,NULL,0,
WSA_FLAG_OVERLAPPED);
```

如果想创建特定的套接字，可以将 IRDA_PROTO_SOCK_STREAM 作为上面任何一个函数的协议参数进行传递。不过，该协议参数并不是并需的。因为传输目录中只有一个地址族条目 AF_IRDA，也就默认了将使用这个传输条目。

4.1.6 套接字选项

对 IrDA 来说，大多数 SO_socket 选项都是没有意义的。只有 SO_LINGER 和 SO_DONTLINGER 得到了特别的支持。IrSock 特有的套接字选项当然也得到了支持，不过其使用范围只限于地址族为

AF_IRDA 的套接字。第 7 章将全面论述这些选项，并对所有套接字选项及其参数进行了总结。

注意　在本书补充材料上，分别有一个基本的客户机和服务器的 IrSock 应用程序，文件名分别为 IRCLIENT.CPP 和 IRSERVER.CPP。

4.2　IPX/SPX

IPX 协议是一个常见协议，一般用于提供 Novell NetWare 客户机 / 服务器联网服务的计算机。IPX 提供两个进程间的无连接通信；这意味着，如果一个工作站发出一个数据包，该协议将无法保证这个数据包能够准确无误地传送到目标地点。如果应用程序需要有保障的数据传递，但仍坚持使用 IPX，它就应该选用一个 IPX 之上更高级的协议，例如 SPX(Sequence Packet Exchange，顺序分组交换)和 SPX II 协议。这两个协议中，SPX 数据包通过 IPX 发送。Winsock 为应用程序提供了在 Windows 平台上通过 IPX 进行通信的能力(这些平台是 Windows 95、Windows 98、Windows Me 及 Windows NT)，但没有提供在 Windows CE 平台上通过 IPX 进行通信的能力。

4.2.1　寻址

在 IPX 网络中，网段是用 IPX 路由器桥接在一起的。每个网段分配有惟一的 4 字节地址号。当更多的网段桥接在一起时，IPX 路由器便会通过惟一的网段号来管理网段之间的通信。连到一个网段的计算机是用一个惟一的 6 字节的节点号来标识的，这个节点号也往往是网络适配器的物理地址。一个节点(也就是一台计算机)一般有一个或多个进程用 IPX 通信。IPX 用套接字号来区分一个节点上的通信进程。

要用 IPX 进行 Winsock 客户机或服务器通信，必须建立 SOCKADDR_IPX 结构。该结构定义在 Wsipx.h 头文件中，在包含 Winsock2.h 文件之后，应用程序还必须包含该文件。SOCKADDR_IPX 结构的定义如下：

```
typedef struct sockaddr_ipx
{
    short              sa_family;
    char           sa_netnum[4];
    char           sa_nodenum[6];
    unsigned short     sa_socket;
}SOCKADDR_IPX,*PSOCKADDR_IPX,FAR *LPSOCKADDR_IPX;
```

sa_family 字段的值应总是设为 AF_IPX。sa_netnum 字段是 4 字节的数字，代表 IPX 网络上的网段号。sa_nodenum 字段是 6 字节的数字，代表计算机物理地址的节点号。sa_socket 字段代表一个套接字或一个端口，用来区分单个节点上的 IPX 通信。

4.2.2 创建套接字

利用 IPX 创建套接字时，有几种可能性存在。要打开 IPX 套接字时，只需调用带有 AF_IPX 地址族、SOCK_DGRAM 套接字类型以及 NSPROTO_IPX 协议的 socket 函数或 WSASocket 函数即可，过程如下：

```
s = socket(AF_IPX,SOCK_DGRAM,NSPROTO_IPX);
s = WSASocket(AF_IPX,SOCK_DGRAM,NSPROTO_IPX,
    NULL,0,WSA_FLAG_OVERLAPPED);
```

注意，第 3 个参数协议是必须指定的，而且不能为 0。该字段还可用于设置特定 IPX 包的类型，所以，这一点相当重要。

正如本节前面提到的那样，IPX 利用数据报提供不可靠的无连接通信。如果一个应用程序需要可靠的无连接通信，可采用比 IPX 高级的协议，例如 SPX 和 SPX II。要做到这一点，需要把 socket 和 WSASocket 调用中的类型字段设置成套接字类型 SOCK_SEQPACKET 或 SOCK_STREAM，并把协议字段设置成 NSPROTO_SPX 或 NSPROTO_SPXII 即可。

如果指定了 SOCK_STREAM，系统就会把数据当作连续不断的字节流发送出去，没有消息边界。这类似于 TCP/IP 中套接字的行为。另一方面，如果指定了 SOCK_SEQPACKET，数据传输时就有消息边界。在这种情况下，如果发送端发出 2 000 个字节，在这 2 000 个字节全部到达接收端之前，接收端是不会返回任何消息的。对 SPX 和 SPX II 来说，通过设置 SPX 头中的消息结束位，就可以做到这一点。指定 SOCK_STREAM 时，要注意这个位，因为在收到这个位之前，Winsock 的 recv 和 WSARecv 调用是不会终止的。如果指定 SOCK_STREAM 时，没有注意到这个消息结束位，一旦接收端收到数据，不管消息结束位设在哪里，recv 都会终止。从发送端这一方来说(使用 SOCK_SEQPACKET 类型)，如果发送的数据量小于单个的数据包，消息结束位就会随这些数据一起发送。如果发送的数据量大于单个的数据包，这些数据将被分包发送，消息结束位就会只设在最后发送的那个包中，而不是每个包都有。

4.2.1.1 绑定套接字

当 IPX 应用程序通过 bind 把本地地址和套接字关联在一起时，切记不要在 SOCKADDR_IPX 结构中指定网络编号和节点地址。这些字段会由 bind 函数利用系统中可用的第 1 个 IPX 网络接口来填充。如果一台计算机有多个网络接口(多重初始地址计算机)，就没有必要将它绑定到特定的接口。Windows 95、Windows 98、Windows Me 及 Windows NT 平台提供了一个虚拟内部网，该虚拟网中的每个接口都是可以访问的，与这些接口直接连接的物理网络无关。稍后将对内部网络编号进行详细论述。在成功绑定到本地接口之后，应用程序便可利用下述代码段中的 getsockname 函数获得本地网络编号和节点编号的信息。

```
SOCKET sdServer;
```

```
SOCKADDR_IPX IPXAddr;
int addrlen = sizeof(SOCKADDR_IPX);
if ((sdServer = socket (AF_IPX,SOCK_DGRAM,NSPROTO_IPX))
    == INVALID_SOCKET)
{
    printf("socket failed with error %d \n",
        WSAGetLastError());
    return;
}
ZeroMemory(&IPXAddr,sizeof(SOCKADDR_IPX));
IPXAddr.sa_family = AF_IPX;
IPXAddr.sa_socket = htons(5150);
if (bind(sdServer,(PSOCKADDR)&IPXAddr,sizeof(SOCKADDR_IPX))
    == SOCKET_ERROR)
{
    printf("bind failed with error %d \n",
        WSAGetLastError());
    return;
}
if (getsockname(sdServer,(PSOCKADDR)&IPXAddr,&addrlen)
    == SOCKET_ERROR)
{
    printf("getsockname failed with error %d",
        WSAGetLastError());
    return;
}
//输出从 getsockname()返回的 SOCKADDR_IPX 信息
```

4.2.2.2　网络编号与内部网络编号

网络编号(通常称作外部网络编号)用于标识 IPX 中的网络段，也用于 IPX 数据包在网络段之间的路由选择。Windows 95、Windows 98、Windows Me 及 Windows NT 平台都支持了内部网络编号，这个内部网络编号用于内部路由选择，并对网间网(几个网络桥接在一起构成)上的计算机进行惟一性标识。内部网络编号也称为“虚拟网号”，因为内部网络编号标识网间网中的另一个(虚拟的)网络段。因此，对一台正在运行 Windows 95、Windows 98、Windows Me 或 Windows NT 平台的计算机，如果为它配置了一个内部网络编号，NetWare 服务器或 IPX 路由器就会在自己与那台计算机的路由之间添加一个额外的中继。

对于多重初始地址计算机，内部虚拟网络还有特殊的用途。IPX 能够通过内部虚拟网络，把一个数据包从任何一个外部网络路由到任何一个本地网络接口。正因为如此，当应用程序绑定到本地网络接口时，不应该指定本地接口的信息，但需要把 SOCKADDR_IPX 结构的 sa_netnum 和 sa_nodenum 这两个字段设置为 0。举个例子，即使应用程序显式地绑定到网络 A 上的网络接口，而数据包却来自网络 B，内部网络编号也会使这个数据包在内部网络中运行路由，这样，应用程序就能收到这个数据

包了。

4.2.2.3 用 Winsock 设置 IPX 数据包类型

在利用 NSPROTO_IPX 协议规范建立套接字时，Winsock 允许应用程序指定 IPX 包的类型。IPX 包内的数据包类型字段表明这个 IPX 包提供或请求的服务类型。在 Novell 网中，下面的 IPX 包类型的定义是：

- **01h**：RIP(Routing Information Protocol，路由信息协议)包。
- **04h**：SAP(Service Advertising Protocol，服务公告协议)包。
- **05h**：SPX(Seqnenced Packet Exchange，顺序分组交换)包
- **11h**：NCP(NetWare Core Protocol，NetWare 核心协议)包。
- **14h**：Novell NetBIOS 的传输包。

要修改 IPX 包类型，只须把 NSPROTO_IPX+n 指定为 socket API 的协议参数，n 表示数据包类型的编号。例如，要打开一个把包类型设为 04h(SAP 包)的 IPX 套接字，可用下面的 socket 调用：

```
s = socket(AF_IPX,SOCK_DGRAM,NSPROTO_IPX +0x04);
```

4.2.2.4 名称解析

大家可能知道，在 Winsock 中进行 IPX 寻址时，必须提供构成地址的网络和节点的多字节编号，显得有些麻烦。IPX 支持应用程序通过用户友好名找到相应服务，并进而通过 SAP 协议获得 IPX 网络中的网络编号、节点编号和端口编号。正如第 8 章将要讲到的那样，Winsock 2 利用 WSASetService API 函数，提供了一个与协议无关的名称解析方法。通过 SAP 协议，IPX 服务器应用程序可利用 WSAAetService 通过用户友好名注册它们正在监听的网络编号、节点编号和端口编号。Winsock 2 还通过几个 API 函数提供了另外一种与协议无关的名称解析方法，这些函数是：WSALookupService-Begin、WSAlookupServiceNext 和 WSAlookupServiceEnd。

打开一个 IPX 套接字，并指定 SAP 包的类型，便可执行名称服务的注册和查找。打开套接字之后，便可开始向 IPX 网络广播 SAP 包，以便在网络上注册和定位服务。这就要求深入了解 SAP 协议，并处理对 IPX SAP 包进行解码的具体编程细节。

注意 在补充材料上，有一个名为 SOCKSPX.CPP 的 IPX 客户机和服务器应用程序，这个程序展示了如何在 IPX 上传输数据报，或在 SPX 上传输可靠的数据通信。

4.3 NetBIOS

NetBIOS 地址族也是另一种可从 Winsock 访问的协议族。NetBIOS 本身是一种网络编程接口(而不是网络协议)，它可在许多网络协议(如 TCP/IP)之上通信。Winsock 通过 NetBIOS 地址族与 NetBIOS

编程接口进行对接。从 Winsock 对 NetBIOS 寻址仍然需要得知 NetBIOS 名和 LANA 编号。补充材料上的第 17 章“NetBIOS API”，可以帮助大家了解许多 NetBIOS 的概念。这里不再讨论整个核心的 NetBIOS 接口概念，而是继续探讨通过 Winsock 访问 NetBIOS 的细节。

注意　NetBIOS 地址族由 Winsock 公开，只能用于 Windows NT 平台，不能用于 Windows 95、Windows 98 或 Windows Me 平台，也不能用于 Windows CE。

4.3.1　寻址

NetBIOS 下为计算机寻址的基础是 NetBIOS 名。一个 NetBIOS 名长度为 16 个字符，最后一个字符留给限定字符用，该字符定义这个名称属于哪类服务。NetBIOS 名有两种类型：惟一名和组名。惟一名在整个网络上只能由一个进程进行注册。例如，一个基于会话的服务器注册了一个名称 FOO，而打算和该服务器通信的客户机将尝试和 FOO 连接。组名允许一组应用程序注册同一个名称，如此一来，注册了这个组名的所有进程都可收到发给这个名称的数据报。

在 Winsock 中，NetBIOS 寻址结构是在 Wsanetbs.h 中定义的，定义如下：

```
#define NETBIOS_NAME_LENGTH 16
typedef struct sockaddr_nb
{
    short snb_family;
    u_short snb_type;
    char snb_name[NETBIOS_NAME_LENGTH];
}SOCKADDR_NB,*PSOCKADDR_NB,FAR *LPSOCKADDR_NB;
```

snb_family 字段指定这个结构的地址族，它应该始终被设置为 AF_NETBIOS。snb_type 字段用于指定一个惟一名或组名。下面的定义可用于这个 snb_type 字段：

```
#define NETBIOS_UNIQUE_NAME (0x0000)
#define NETBIOS_GROUP_NAME (0x0001)
```

最后，snb_name 字段就是实际的 NetBIOS 名。

如果大家已知道各字段的含义以及应该怎样设置它们，那么可以使用下面的宏，它定义于头文件中，将完成所有的设置：

```
#define SET_NETBIOS_SOCKADDR(_snb,_type,_name,_port)          \
{                                                             \
    int _i;                                                   \
    (_snb)->snb_family = AF_NETBIOS;                          \
    (_snb)->snb_type = (_type);                               \
    for (_i = 0;_i <NETBIOS_NAME_LENGTH -1;_i++){             \
    (_snb)->snb_name[_i]='';                                  \
    }                                                         \
```

```
    for (_i = 0;                                                    \
        *((_name)+_i)!='\0'                                         \
        && _i <NETBIOS_NAME_LENGTH -1;                              \
        _i++)                                                       \
    {                                                               \
     (_snb)->snb_name[_i]=*((_name)+_i);                            \
    }                                                               \
     (_snb)->snb_name[NETBIOS_NAME_LENGTH -1]=(_port);              \
;
```

这个宏的第一个参数_snb 是将要填写的 SOCKADDR_NB 结构的地址，它自动把 snb_family 字段设为 AF_NETBIOS。而这个宏的_type 参数指定的是 NETBIOS UNIQUE_NAME 或 NETBIOS_GROUP_NAME。_name 参数是 NetBIOS 名，这个宏认定_name 参数的长度至少是 NETBIOS_NAME_LENGTH 减 1，如果短于这一长度，它就会以空字符结束。注意，snb_name 字段是用空格来预填的。最后，这个宏把 snb_name 字符串的第 16 个字符设为_port 参数的值。

可以看到，在 Winsock 中 NetBIOS 名的结构是直截了当的，没有任何令人费解之处。名称解析是在后台执行的，因此，在执行操作之前，不必把名称解析成物理地址，这一点和 TCP 和 IrDA 相反。如果 NetBIOS 的实施依赖于多个协议，而各个协议都有自己的一套寻址方案时，这一点尤为明显。

4.3.2 创建套接字

创建套接字时，最重要的一点是 LANA 编号。就像使用原始 NetBIOS API 那样，这里必须知道哪些 LANA 编号和应用程序有关。NetBIOS 客户机和服务器之间要进行通信，它们必须有一个共用的传输协议，这样它们便可以通过该协议进行监听或连接。创建 NetBIOS 套接字有两种方法。第 1 种方法，是像下面这样调用 socket 或 WSASocket：

```
s = WSASocket(AF_NETBIOS,SOCK_DGRAM,-lana,
        NULL,0,WSA_FLAG_OVERLAPPED);
```

根据需要的是一个无连接数据报，还是一个面向连接的会话套接字，WSASocket 的 type 参数分别是 SOCK_DGRAM 和 SOCK_SEQPACKET。第 3 个参数 protocol 表示创建套接字时所依据的那个 LANA 编号。如果不想用它来表示该 LANA 编号，就得将它设置为负值。由于这里是在指定自己的参数，而不是使用 WSAPROTOCOL_INFO 结构，所以第 4 个参数是空值。第 5 个参数没有用。最后，把 dwFlags 参数设为 WSA_FLAG_OVERLAPPED；所有 WSASocket 调用都要指定 WSA_FLAG_OVERLAPPED。如果打算使用重叠 I/O(如下一章所述)，则创建套接字时就需要显示这个标志。

第 1 种套接字创建方法的不足之处是，必须知道从哪一个 LANA 编号着手才是有效的。不幸的是，目前 Winsock 中还没有一个简捷的方法能够把所有有效的 LANA 编号都列举出来。Winsock 提供了一个替代方案，即利用 WSAEnumProtocols 把所有传输协议列举出来。当然，也可像第 17 章叙述

的那样，利用 NCBENUM 命令调用 Netbios，从而获得有效的 LANA 编号。第 2 章对如何调用 WSAEnumProtocols 进行了说明。下面的示例列举所有的传输协议，对 NetBIOS 传输进行搜索，并为各个协议分别建立套接字。

```
dwNum = WSAEnumProtocols(NULL,lpProtocolBuf,&dwBufLen);
if (dwNum == SOCKET_ERROR)
{
    //出错
}
for (i = 0;i <dwNum;i++)
{
    //在 AF_NETBIOS 地址族中寻找这些条目
    if (lpProtocolBuf[i].iAddressFamily == AF_NETBIOS)
    {
        //寻找 SOCK_SEQPACKET 或 SOCK_DGRAM
        if (lpProtocolBuf[i].iSocketType == SOCK_SEQPACKET)
        {
            s[j++] = WSASocket(FROM_PROTOCOL_INFO,
                FROM_PROTOCOL_INFO,FROM_PROTOCOL_INFO,
                    &lpProtocolBuf[i],0,WSA_FLAG_OVERLAPPED);
        }
    }
}
```

在上面的伪代码中，我们列举了可用的传输协议，并通过它们对属于 AF_NETBIOS 地址族的协议进行反复查找。接下来，我们要检查套接字类型，这里要查找的是 SOCK_SEQPACKET 类型的条目。如果需要数据报，就检查 SOCK_DGRAM。如果找到的套接字类型和条目匹配，就有一个可以使用的 NetBIOS 传输协议了。如果需要 LANA 编号，就选用 WSAPROTOCOLS_INFO 结构中 iProtocol 字段的绝对值。不过当 LANA 编号为 0 时例外。因为 0 是 Winsock 为特殊用途保留的，这个 LANA 的 iProtocol 字段是 0x80000000。变量 j 包含有效的传输协议的数量。

注意　在补充材料上，有一个 NetBIOS 客户机和一个服务器应用程序，文件名分别为 WSNBCLNT.CPP 和 WSNBSVR.CPP。

4.4　AppleTalk

Winsock 中，对 AppleTalk 的支持已有一段时间了，但很少有人注意到。如果不与苹果公司的 MAC 机通信，您可能不会选择 AppleTalk 协议。因为 AppleTalk 与 NetBIOS 都是根据每一个进程来进行名称注册的，所以两者有些类似。也就是说，要使特定的名称为人所知，服务器必须对这个名称进行注册，客户机再用这个名称与服务器建立连接。不过，AppleTalk 名比 NetBIOS 名更为复杂。下一小节

将讨论如何为网络上使用 AppleTalk 协议的计算机进行寻址。

4.4.1 寻址

AppleTalk 名实际上是以 3 个独立的部分为基础的：名称、类型和区。每个名称的长度可达 32 个字符。这个名称标识计算机上的进程及与其关联的套接字。类型是区的子群机制。一般情况下，区是一个网络，它是由物理地定位于同一个循环圈内、使用 AppleTalk 协议的计算机构成的。微软实现的 AppleTalk 允许 Windows 兼容机对自己所在的默认区进行指定。多个网络可桥接在一起，各种易记的名称分别映射到一个套接字编号、一个节点编号和一个网络编号。在特定的类型和区内，AppleTalk 名必须是独一无二的。要满足这个要求，需要使用 NBP(Name Binding Protocol，名称绑定协议)。这个协议将在网上广播一个查询，以便知道这个名称是否已被使用。与此同时，AppleTalk 利用 RTMP(Routing Table Maintenance Protocol，路由表维护协议)，动态地对链接在一起的各个 AppleTalk 网络的路由进行查找。

下面的结构提供了从 Winsock 为 AppleTalk 主机寻址的基础：

```
typedef struct sockaddr_at
{
    USHORT sat_family;
    USHORT sat_net;
    UCHAR sat_node;
    UCHAR sat_socket;
}SOCKADDR_AT,*PSOCKADDR_AT;
```

注意，这个地址结构中只有字符或短整型数，没有友好名。SOCKADDR_AT 结构被传递到 bind、connect 和 WSAconnect 之类的 Winsock 调用中，如果要转换易于理解的名称，首先须在网络是对这个名称进行解析或注册。解析或注册分别通过 getsockopt 或 setsockopt 调用来完成。

4.4.1.1 注册 AppleTalk 名称

对一个服务器而言，如果它打算注册特定的名称，以便客户机可以很容易地与之建立连接，就需要利用 SO_REGISTER_NAME 选项来调用 setsockopt 函数。对于和 AplpleTalk 名有关的所有套接字选项，使用 WSH_NBP_NAME 结构即可，其定义为：

```
typedef struct
{
    CHAR ObjectNameLen;
    CHAR ObjectName[MAX_ENTITY];
    CHAR TypeNameLen;
    CHAR TypeName[MAX_ENTITY];
    CHAR ZoneNameLen;
    CHAR ZoneName[MAX_ENTITY];
}WSH_NBP_NAME,*PWSH_NBP_NAME;
```

许多类型(例如 WSH_REGISTER_NAME、WSH_DEREGISTER_NAME 和 WSH_REMOVE_NAME)都是在这个 WSH_NBP_NAME 结构的基础上定义的。使用哪个类型要根据具体情况而定，取决于所进行的操作是查找名称、注册名称还是删除名称。

下面的代码示例解释了如何注册 AppleTalk 名：

```
#define MY_ZONE "*"
#define MY_TYPE "Winsock-Test-App"
#define MY_OBJECT "AppleTalk-Server"
WSH_REGISTER_NAME atname;
SOCKADDR_AT ataddr;
SOCKET s;
//
//填入要注册的名称
//
strcpy(atname.ObjectName,MY_OBJECT);
atname.ObjectNameLen = strlen(MY_OBJECT);
strcpy(atname.TypeName,MY_TYPE);
atname.TypeNameLen = strlen(MY_TYPE);
strcpy(atname.ZoneName,MY_ZONE);
atname.ZoneNameLen = strlen(MY_ZONE);
s = socket(AF_APPLETALK,SOCK_STREAM,ATPROTO_ADSP);
if (s == INVALID_SOCKET)
{
//出错
}
ataddr.sat_socket = 0;
ataddr.sat_family = AF_APPLETALK;
if (bind(s,(SOCKADDR *)&ataddr,sizeof(ataddr))==SOCKET_ERROR)
{
//无法在AppleTalk网络中打开一个端点
}
if (setsockopt(s,SOL_APPLETALK,SO_REGISTER_NAME,
(char *)&atname,sizeof(WSH_NBP_NAME))==SOCKET_ERROR)
{
//名称注册失败!
}
```

首先要注意的是 MY_ZONE、MY_TYPE 和 MY_OBJECT 这 3 个字符串。记住，AppleTalk 名是 3 层的。注意，区是一个星号“*”。这是区字段中所用的特殊字符，表示计算机处在“当前”区。接下来，建立一个隶属于 ADSP(AppleTalk Data Stream Protocol，AppleTalk 数据流协议)的 SOCK_STREAM 类型的套接字。建立这个套接字之后，再利用一个地址结构 bind 进行调用，这个地址结构中有一个置零的 sat_socket 字段，并只设置了协议族字段。这个调用为函数用户的应用程序在网络上建立了一个发出请求的端点，所以非常重要。注意，虽然对 bind 的调用允许在网络上执行简

单操作，但它本身不允许应用程序接受客户机发出的连接请求。要接受客户机的连接请求，必须在网络上注册自己的名称，这是下一步工作要完成的。

注册 AppleTalk 名很简单。把 SOL_APPLETALK 作为 level 参数，把 SO_REGISTER_NAME 作 optname 参数进行传递，便可调用 setsockopt。后两个参数是一个指针，它指向 WSH_REGISTER_NAME 结构及其长度。如果 setsockopt 调用成功，服务器名便得以成功注册。如果调用失败，则表明所请求的名称大概已为他人所用。此时返回的 Winsock 错误是 WSAEADDRINUSE(10048)。注意，对同时面向数据报和面向流的 AppleTalk 协议来说，想接收数据的进程必须注册一个名称，以便客户机可向它发送数据报或与之建立连接。

4.4.1.2 解析 AppleTalk 名称

在对等的客户机这一端，应用程序通常通过服务器的友好名来得知服务器，并且必须把这个友好名解析成 Winsock 调用所需的网络编号、节点编号和套接字编号。通过 SO_LOOKUP_NAME 选项调用 getsockopt 函数，便可实现这一目的。执行 AppleTalk 名的查找依赖于 WSA_LOOKUP_NAME 结构，这个结构及与它相关的结构定义如下：

```
typedef struct
{
    WSH_ATALK_ADDRESS Address;
    USHORT Enumerator;
    WSH_NBP_NAME NbpName;
}WSH_NBP_TUPLE,*PWSH_NBP_TUPLE;
typedef struct _WSH_LOOKUP_NAME
{
    //NoTuple WSH_NBP_TUPLE 数组
    WSH_NBP_TUPLE LookupTuple;
    vULONG NoTuples;
}WSH_LOOKUP_NAME,*PWSH_LOOKUP_NAME;
```

在利用 SO_LOOKUP_NAME 选项调用 getsockopt 时，我们把一个缓冲区转换为 WSH_LOOKUP_NAME 结构传递出去，并在第 1 个 LookupTuple 成员内填充 WSH_NBP_NAME 字段。调用成功后，getsockopt 会返回一个由 WSH_NBP_TUPLE 元素组成的数组，其中包含所查找的 AppleTalk 名称的物理地址信息。下面的示例中包含了 ATALKNM.C 文件，该文件对如何查找 AppleTalk 名进行了说明。除此以外，示例还展示了如何列出所有“已找到的”AppleTalk 区，以及如何找到用户的默认区。区信息可通过 getsockopt 的选项 SO_LOOKUP_ZONES 和 SO_LOOKUP_MYZONE 来获得。

```
#include <winsock.h>
#include <atalkwsh.h>
#include <stdio.h>
#include <stdlib.h>
#define DEFAULT_ZONE "*"
```

```
#define DEFAULT_TYPE "Windows Sockets"
#define DEFAULT_OBJECT "AppleTalk-Server"
char szZone[MAX_ENTITY],
     szType[MAX_ENTITY],
     szObject[MAX_ENTITY];
BOOL bFindName = FALSE,
     bListZones = FALSE,
     bListMyZone = FALSE;
void usage()
{
    printf("usage:atlookup[options]\n");
    printf("Name Lookup:\n");
    printf("-z:ZONE-NAME \n");
    printf("-t:TYPE-NAME \n");
    printf("-o:OBJECT-NAME \n");
    printf("List All Zones:\n");
    printf("-lz \n");
    printf("List My Zone:\n");
    printf("-lm \n");
    ExitProcess(1);
}
void ValidateArgs(int argc,char **argv)
{
    int i;

    strcpy(szZone,DEFAULT_ZONE);
    strcpy(szType,DEFAULT_TYPE);
    strcpy(szObject,DEFAULT_OBJECT);

    for(i = 1;i <argc;i++)
    {
if (strlen(argv[i])<2)
    continue;
if ((argv[i][0]=='-')||(argv[i][0]=='/'))
{
    switch (tolower(argv[i][1]))
    {
        case 'z'://指定一个区域名
            if (strlen(argv[i])>3)
                strncpy(szZone,&argv[i][3],MAX_ENTITY);
            bFindName = TRUE;
            break;
        case 't'://指定一个类型名
            if (strlen(argv[i])>3)
                strncpy(szType,&argv[i][3],MAX_ENTITY);
```

```
            bFindName = TRUE;
            break;
        case 'o'://指定一个对象名
            if (strlen(argv[i])>3)
                strncpy(szObject,&argv[i][3],MAX_ENTITY);
            bFindName = TRUE;
            break;
        case 'l'://列出区信息
            if (strlen(argv[i])==3)
                //列出所有的区
                if (tolower(argv[i][2])=='z')
                    bListZones = TRUE;
                //列出用户区
                else if (tolower(argv[i][2])=='m')
                    bListMyZone = TRUE;
                break;
            default:
                usage();
            }
        }
    }
}
int main(int argc,char **argv)
{
    WSADATA wsd;
    char cLookupBuffer[16000],
    *pTupleBuffer = NULL;
    PWSH_NBP_TUPLE pTuples = NULL;
    PWSH_LOOKUP_NAME atlookup;
    PWSH_LOOKUP_ZONES zonelookup;
    SOCKET s;
    DWORD dwSize = sizeof(cLookupBuffer);
    SOCKADDR_AT ataddr;
    int i;
    //加载 Winsock 库
    //
    if (WSAStartup(MAKEWORD(2,2),&wsd)!=0)
    {
        printf("Unable to load Winsock library!\n");
        return 1;
    }
    ValidateArgs(argc,argv);
    atlookup = (PWSH_LOOKUP_NAME)cLookupBuffer;
    zonelookup = (PWSH_LOOKUP_ZONES)cLookupBuffer;
    if (bFindName)
```

```
{
    //填入要查找的名称
    //
    strcpy(atlookup->LookupTuple.NbpName.ObjectName,szObject);
    atlookup->LookupTuple.NbpName.ObjectNameLen =
        strlen(szObject);
    strcpy(atlookup->LookupTuple.NbpName.TypeName,szType);
    atlookup->LookupTuple.NbpName.TypeNameLen = strlen(szType);
    strcpy(atlookup->LookupTuple.NbpName.ZoneName,szZone);
    atlookup->LookupTuple.NbpName.ZoneNameLen = strlen(szZone);
}
//创建 AppleTalk 套接字
//
s = socket(AF_APPLETALK,SOCK_STREAM,ATPROTO_ADSP);
if (s == INVALID_SOCKET)
{
    printf("socket()failed:%d \n",WSAGetLastError());
    return 1;
}
//需要实施绑定，以便在 AppleTalk 网络上创建一个端点，并由此进行查询
//
ZeroMemory(&ataddr,sizeof(ataddr));
ataddr.sat_family = AF_APPLETALK;
ataddr.sat_socket = 0;
if (bind(s,(SOCKADDR *)&ataddr,sizeof(ataddr))==
        INVALID_SOCKET)
{
    printf("bind()failed:%d \n",WSAGetLastError());
    return 1;
}

if (bFindName)
{
    printf("Looking up:%s:%s@%s \n",szObject,szType,szZone);
    if (getsockopt(s,SOL_APPLETALK,SO_LOOKUP_NAME,
            (char *)atlookup,&dwSize)==INVALID_SOCKET)
    {
        printf("getsockopt(SO_LOOKUP_NAME)failed:%d \n",
            WSAGetLastError());
        return 1;
    }
    printf("Lookup returned:%d entries \n",
        atlookup->NoTuples);
    //
    //字符缓冲区现在包含了 WSH_NBP_TUPLE 结构的一个数组，其位置在 WSH_LOOKUP_NAME 结构
```

之后。

```
        //
        pTupleBuffer = (char *)cLookupBuffer +
            sizeof(WSH_LOOKUP_NAME);
        pTuples = (PWSH_NBP_TUPLE)pTupleBuffer;
        for(i = 0;i <atlookup->NoTuples;i++)
        {
            ataddr.sat_family = AF_APPLETALK;
            ataddr.sat_net = pTuples[i].Address.Network;
            ataddr.sat_node = pTuples[i].Address.Node;
            ataddr.sat_socket = pTuples[i].Address.Socket;
            printf("server address = %lx.%lx.%lx.\n",
                ataddr.sat_net,
                ataddr.sat_node,
                ataddr.sat_socket);
        }
    }
    else if (bListZones)
    {
        //应该为这个选项传递一个足够大的缓冲区，这一点非常重要。
        //如果缓冲区太小，Windows NT 4 SP3 就会出现蓝屏现象。
        //
        if (getsockopt(s,SOL_APPLETALK,SO_LOOKUP_ZONES,
                (char *)atlookup,&dwSize)==INVALID_SOCKET)
        {
            printf("getsockopt(SO_LOOKUP_NAME)failed:%d \n",
                WSAGetLastError());
            return 1;
        }
        printf("Lookup returned:%d zones \n",zonelookup->NoZones);
        //
        //字符缓冲区包含了一个以空字符分离的字符串列表，其位置在 WSH_LOOKUP_ZONES 结构之后。
        //
        pTupleBuffer = (char *)cLookupBuffer +
            sizeof(WSH_LOOKUP_ZONES);
        for(i = 0;i <zonelookup->NoZones;i++)
        {
            printf("%3d:'%s'\n",i+1,pTupleBuffer);
            while (*pTupleBuffer++);
        }
    }
    else if (bListMyZone)
    {
        //这个选项返回一个简单的字符串。
        //
```

```
    if (getsockopt(s,SOL_APPLETALK,SO_LOOKUP_MYZONE,
     (char *)cLookupBuffer,&dwSize)==INVALID_SOCKET)
    {
        printf("getsockopt(SO_LOOKUP_NAME)failed:%d \n",
        WSAGetLastError());
        return 1;
    }
    printf("My Zone:'%s'\n",cLookupBuffer);
}
else
    usage();
WSACleanup();
return 0;
}
```

在使用大多数 AppleTalk 套接字选项时(例如 SO_LOOKUP_MYZONE、SO_LOOKUP_ZONES 以及 SO_LOOKUP_NAME)，需要为 getsockopt 调用提供比较大的字符缓冲区。如果调用的选项要求提供一个结构，则必须得把这个结构放在所提供的字符缓冲区中开始的位置。如果调用 getsockopt 成功，这个函数就会把返回的数据放在字符缓冲区中所提供的结构之后。我们来看看上述示例中的 SO_LOOKUP_NAME 这一部分。变量 cLookupBuffer 是一个用在 getsockopt 调用中的简单字符数组。首先，我们把它转换成 PWSH_LOOKUP_NAME，并在里面填入想查找的名称信息。然后，把这个缓冲区传递给 getsockopt，再根据返回的结果，移动字符指针 pTupleBuffer，使其指向 WSH_LOOKUP_NAME 结构之后的那个字符。接下来，因为查找 AppleTalk 名的调用返回的数据是一个 WSH_NBP_TUPLE 结构数组，所以再把这个字符指针转换成一个 PWSH_NBP_TUPLE 变量。一旦有了正确的起始位置和字元组类型，名称查找就成功了。关于 AppleTalk 地址族特有的各种套接字选项，请参考第 7 章。

4.4.2　创建套接字

AppleTalk 可用于 Winsock 1.1 及稍后的版本，因此可以任选这些版本中的套接字创建例程。再次提醒大家注意，指定下层 AppleTalk 协议的方式有两种。第 1，可为自己需要的协议提供 Atalkwh.h 中的相应定义。第 2，利用 WSAEnumProtocols，并传递 WSAPROTOCOL_INFO 结构，使可列举协议了。直接用 socket 或 WSASocket 创建套接字时各 AppleTalk 协议所需的参数，可参见表 4.1。

表 4.1　AppleTalk 协议和参数

协　议	地址族	套接字类型	协议类型
MSAFDAppleTalk[ADSP]		SOCK_RDM	ATPROTO_ADSP
MSAFDAppleTalk[ADSP][Pseudo-Stream]		SOCK_STREAM	ATPROTO_ADSP

续表

协 议	地址族	套接字类型	协议类型
MSAFDAppleTalk[PAP]	AF_APPLETALK	SOCK_RDM	ATPROTO_PAP
MSAFDAppleTalk[RTMP]		SOCK_DGRAM	DDPPROTO_RTMP
MSAFDAppleTalk[ZIP]		SOCK_DGRAM	DDPPROTO_ZIP

注意 在补充材料中，有一个名为 ATALK.CPP 的 AppleTalk 应用程序，在 PAP 及 ADSP 协议上，该程序可以用作发送端或接收端应用程序。

4.5 ATM

ATM(Asynchronous Transfer Mode，异步传输模式)协议是目前的最新协议之一，Windows 98、Windows Me、Windows 2000 和 Windows XP 平台上的 Winsock 2 均支持 ATM。ATM 通常用于 LAN 和 WAN 上的高速联网，也用于各种类型的通信，例如要求高速通信的语音、视频和数据等。一般说来，ATM 利用网络上的 VC(Virtual Connections，虚拟连接)来提供有保证的 QOS。正如大家即将看到的那样，Winsock 能够通过 ATM 地址族来使用 ATM 网络上的虚拟连接。ATM 网络(如图 4.1 所示)一般由通过交换机(它们将 ATM 网络桥接在一起)相互连接的端点(或计算机)构成。

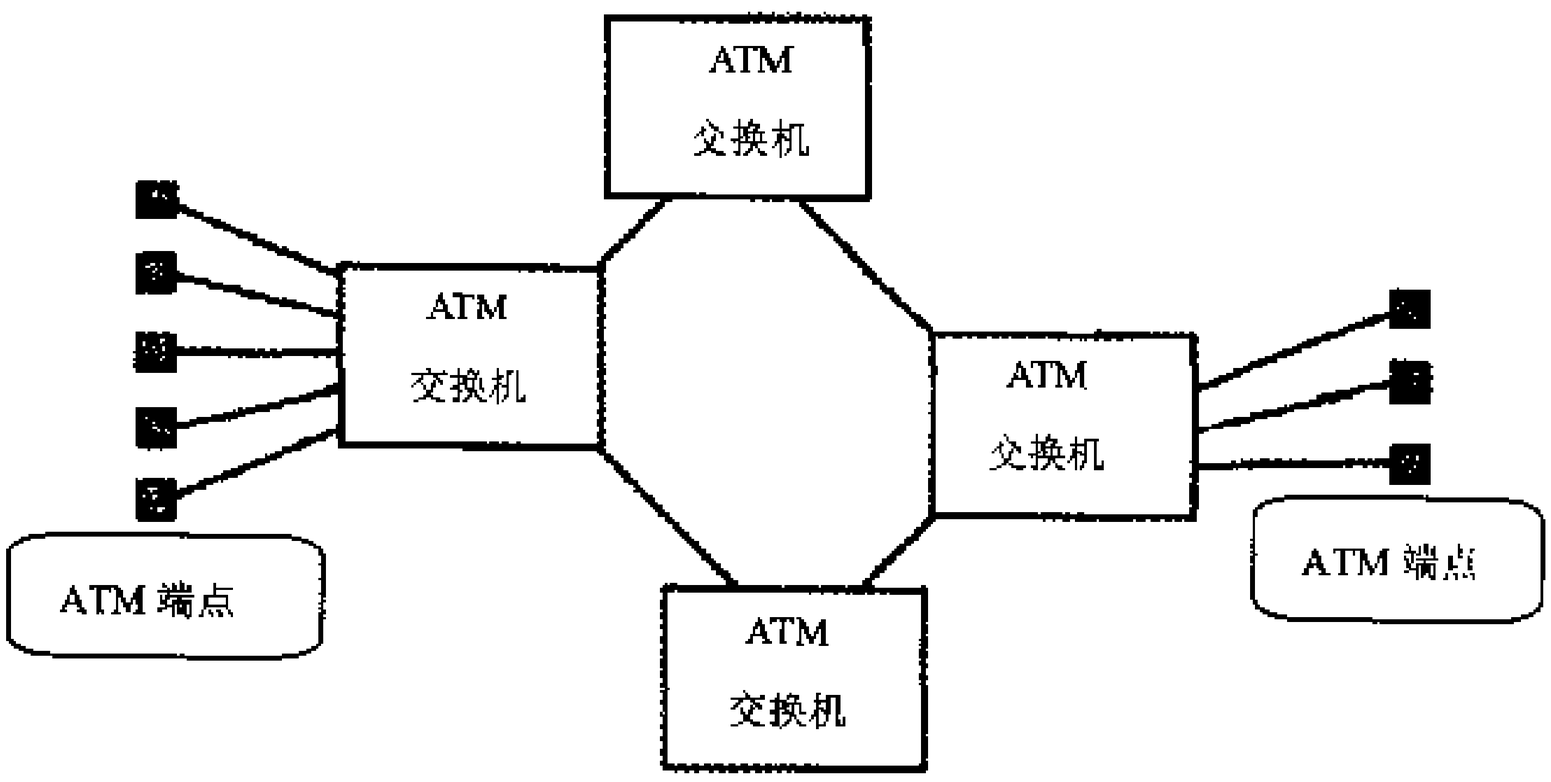


图 4.1 ATM 网络

在进行有关 ATM 协议的编程时，需要注意下面几点。首先，ATM 是一个媒体类型，而不是一种真正的协议。也就是说，ATM 类似于直接在以太网上写入以太帧。和以太网一样，ATM 协议没有提供流控制。它相当于一种面向连接的协议，要么提供消息模式，要么提供流模式。这还意味着，如果数据不能快速发送出去，发送应用程序就有可能造成本地缓冲溢出。同样，接收应用程序必须频繁投

递收到的数据：否则，接收缓冲被填满的时候，任何再传入的数据都可能被丢弃。如果应用程序需要流控制，可以采用这种方法，即在 ATM 上使用 IP 协议(它只是运行于 ATM 网络上的 IP 协议)。这样，应用程序便遵循在前面所描述的 IP 地址族(如第 3 章所述)。当名，ATM 确实提供了一些优于 IP 的优势，例如“根式多播方案”(第 9 章将对此进行说明)：不过，要根据应用程序需要来决定最适合的那种协议。

4.5.1　寻址

一个 ATM 网络有两个网络接口：UNI(user network interface，用户网络接口)和 NNI(network node interface，网络节点接口)。UNI 接口是在端点和 ATM 交换机之间建立的通信，而 NNI 接口则是在两个交换机之间建立的通信。各个接口都有自己相关的通信协议，具体说明如下：

- **UNI 信号协议**　允许端点在一个端点和一个 ATM 交换机之间发送设置及控制信息，从而在 ATM 网络上建立通信。注意，这个协议只限于终端点和 ATM 交换机之间的传输，不能直接通过交换机在 ATM 网络上传输。
- **NNI 信号协议**　允许 ATM 交换机在两个交换机之间交流路由选择和控制信息。

为达到通过 Winsock 设置 ATM 连接的目的，我们将只讨论 UNI 信号协议中的特定信息元素。目前，Windows XP、Windows 2000、Windows Me 及 Windows 98(Sernee Pack 1)上的 Winsock 均可支持 UNI 信号协议 3.1 版本。

Winsock 允许客户机 / 服务器应用程序通过设置 SAP 在 ATM 网络上进行通信。这需要使用 ATM UNI 信号协议。ATM 是一种面向连接的协议，为了进行通信，该协议要求端点在 ATM 网络上建立虚拟连接。对于 ATM 网络上的通信所使用的套接字接口而言，SAP 只是允许 Winsock 应用程序通过 SOCKADDR_ATM 地址结构对这个接口进行注册和标识。SAP 一旦建立，Winsock 就利用 UNI 信号协议，向 ATM 网络发起一系列调用。通过这种方式，Winsock 便用这个 SAP 在 ATM 网络上的 Winsock 客户机和服务器之间建立起一个虚拟连接。

SOCKADDR_ATM 结构的定义如下：

```
typedef struct sockaddr_atm
{
    u_short         satm_family;
    ATM_ADDRESS     satm_number;
    ATM_BLLI        satm_blli;
    ATM_BHLI        satm_bhli;
}sockaddr_atm,SOCKADDR_ATM,*PSOCKADDR_ATM,*LPSOCKADDR_ATM;
```

satm_family 应该始终为 AF_ATM。satm_number 字段利用一种基本的 ATM 寻址方案——E.164 和 NSAP(Network Service Access Point，网络服务访问点)中的一种，将实际的 ATM 地址表示成一个 ATM_ADDRESS 结构。NSAP 地址也被称为 NASP 式的 AESA(ATM Endsystem Address，ATM 终端

系统地址)。ATM_ADDRESS 结构的定义如下：

```
typedef struct
{
    DWORD AddressType;
    DWORD NumofDigits;
    UCHAR Addr[ATM_ADDR_SIZE];
}ATM_ADDRESS;
```

AddressType 字段定义指定的寻址方案。如果指定 E.164 寻址方案，这个字段就是 ATM_E164；若指定 NSAP 式的寻址方案，这个字段就是 ATM_NSAP。除此以外，当应用程序打算把套接字和一个 SAP 绑定在一起时，AddressType 字段还可设为表 4.2 中定义的其他值。本章稍后将对此进行详细讨论。NumofDigits 字段应该一直设为 ATM_ADDR_SIZE。Addr 字段表示 ATM 实际的 20 字节 E.164 或 NASP 地址。

SOCKADDR_ATM 结构的 satm_blli 和 satm_bhli 字段分别代表 ATM 信号协议中的 BLLI(Broadband Lower Layer Information，宽带基层信息)和 BLLI(Broadband Higher Layer Information，宽带高层信息)。一般说来，这些结构用于对 ATM 连接上运行的协议堆栈进行标识。对几个已知的 BLLI 和 BHIL 值的组合说明参见 ATM Form / IETF 文档(这些值的某一特定组合用于标识 ATM 网络上的 LAN 仿真，而另一种组合则用于标识 ATM 网络上的原始 IP，如此等等)。这些结构中字段值的完整范围都列在《ATM UNI 3.1 标准》一书中。ATM Form / IETF 文档可在网址 http://www.ietf.org 上找到。

表 4.2 ATM 套接字地址类型

ATM_ADDRESS AddressType 设置	地址类型
ATM_E164	E.164 地址，与 SAP 连接时使用
ATM_NSAP	NSAP 式的 ATM 终端系统地址(AESA)，与 SAP 连接时使用
SAP_FIELD_ANY_AESA_SEL	NSAP 式的 ATM 终端系统地址，带有通配的选择符 8 位位组。在将套接字绑定到 SAP 时使用
SAP_FIELD_ANY_AESA_REST	NSAP 式的终端系统地址，不包括通配的选择符 8 位位组。但其他的所有 8 位位组都有。在把套接字绑定到 SAP 时使用

BHLI 和 BLLI 数据结构的定义如下：

```
typedef struct
{
    DWORD HighLayerInfoType;
    DWORD HighLayerInfoLength;
    UCHAR HighLayerInfo[8];
```

```
}ATM_BHLI;

typedef struct
{
    DWORD Layer2Protocol;
    DWORD Layer2UserSpecifiedProtocol;
    DWORD Layer3Protocol;
    DWORD Layer3UserSpecifiedProtocol;
    DWORD Layer3IPI;
    UCHAR SnapID[5];
}ATM_BLLI;
```

这些字段的定义和用法不在本书讨论之列。如果只想在 ATM 网络上建立 Winsock 通信，应用程序就应该把 BHLI 和 BLLI 结构中的下列字段设为 SAP_FIELD_ABSENT 值：

- ATM_BLLI——第 2 层协议
- ATM_BLLI——第 3 层协议
- ATM_BHLI——高层信息类型(HighLayerInfoType)

在上述字段被设为 SAP_FIELD_ABSENT 值时，这两个结构中的其他字段都不会再被使用。下面的伪代码演示了应用程序如何用 SOCKADDR_ATM 结构为 NSAP 地址设置 SAP：

```
SOCKADDR_ATM atm_addr;
UCHAR MyAddress[ATM_ADDR_SIZE];
atm_addr.satm_family = AF_ATM;
atm_addr.satm_number.AddressType = ATM_NSAP;
atm_addr.satm_number.NumofDigits = ATM_ADDR_SIZE;
atm_addr.satm_blli.Layer2Protocol = SAP_FIELD_ABSENT;
atm_addr.satm_blli.Layer3Protocol = SAP_FIELD_ABSENT;
atm_addr.satm_bhli.HighLayerInfoType = SAP_FIELD_ABSENT;
memcpy(&atm_addr.satm_number.Addr,MyAddress,ATM_ADDR_SIZE);
```

ATM 地址一般用一个十六进制的 ASCII 字符串表示，该字符串由 40 个字符组成。这些字符与 ATM_ADDRESS 结构中组成 NSAP 式地址或 E.164 地址的 20 个字节相对应。例如，ATM NSAP 式地址可能像这样：

```
47000580FFE1000000F21A1D540000D10FED5800
```

把这个字符串转换成一个 20 字节的地址相当麻烦。不过，Winsock 提供了一个与协议无关的 API 函数 WSAStringToAddress，利用这个函数，便可把一个 40 个字符的 ATM 十六进制 ASCII 字符串转换成一个 ATM_ADDRESS 结构。本章将在最后着重讲一讲这个 API 函数。把十六进制 ASCII 字符串转换成十六进制(二进制)格式的另一种方法，是利用下列代码中定义的 AtoH 函数。该函数不属于 Winsock。但是，和前一个函数比较起来，它更容易开发，大家可看到在补充材料中的 ATM 示例(稍后将讲述)用到了它。

```
    //
    //功能：A 到 H
    //
    //说明：这个函数把用字符串(ASCII)格式指定的 ATM 地址转换为二进制(十六进制)格式
    void AtoH(CHAR *szDest,CHAR *szSource,INT iCount)
    {
        while (iCount--)
        {
            *szDest++ = (BtoH (*szSource++)<<4 )
            +BtoH (*szSource++);
        }
        return;
    }
    //
    //功能：B 到 H
    //
    //说明：这个函数返回与用 ASCII 格式指定的单个字符等同的二进制值
    //
    UCHAR BtoH(CHAR ch )
    {
        if (ch > = '0'&&ch <='9')
        {
            return (ch -'0');
        }
        if (ch >='A'&&ch <='F')
        {
            return (ch -'A'+0xA );
        }
        if (ch >='a'&&ch <='f')
        {
            return (ch -'a'+0xA );
        }
        //
        //地址中的非法值不会被接受
        //
        return -1;
    }
```

4.5.2 创建套接字

ATM 只允许通过虚拟连接进行通信，这就要求，在 ATM 中，应用程序只能建立面向连接的套接字。因此，数据的传输形式是字节流或面向消息的形式。要利用 ATM 协议打开一个套接字，需要先通过地址族 AF_ATM 和套接字类型 SOCK_RAW 调用 socket 函数或 WSASocket 函数，并把协议字段设为 ATMPROTO_AAL5。例如：

```
s = socket(AF_ATM,SOCK_RAW,ATMPROTO_AAL5);
s = WSASocket(AF_ATM,SOCK_RAW,ATMPROTO_AAL5,NULL,0,
WSA_FLAG_OVERLAPPED);
```

在默认状态下，打开一个套接字(例如在上面这个例子中)时，会建立一个面向流的 ATM 套接字。Windows 还实现了一个 ATM 提供程序用来执行面向消息的数据传输。使用这个面向消息的提供程序时，要求用户为 WSAS 函数显式地指定原始的 ATM 协议提供程序，使用 WSAPROTOCOL_INFO 结构可达到这一目的(关于这一结构，第 2 章曾详细讲述过)。因为 socket 和 WSASocket 这两个调用中的 3 个元素(地址族、套接字类型和协议)均和 Winsock 中可以使用的每一个 ATM 协议提供程序相匹配，所以指明 ATM 协议提供程序是非常必须的。在默认状态下，Winsock 会返回与这 3 个属性相匹配的协议条目，并把这个协议条目标记成默认设置(在这个例子中是面向流的提供程序)。下面的伪代码将演示如何获得 ATM 面向消息的协议提供程序，以及如何建立一个套接字：

```
dwRet = WSAEnumProtocols(NULL,lpProtocolBuf,&dwBufLen);
for (i = 0;i <dwRet;i++)
{
    if ((lpProtocolBuf[i].iAddressFamily == AF_ATM)&&
        (lpProtocolBuf[i].iSocketType == SOCK_RAW)&&
        (lpProtocolBuf[i].iProtocol == ATMPROTO_AAL5)&&
        (lpProtocolBuf[i].dwServiceFlags1 &
          XP1_MESSAGE_ORIENTED))
    {
          s = WSASocket(FROM_PROTOCOL_INFO,FROM_PROTOCOL_INFO,
              FROM_PROTOCOL_INFO,lpProtocolBuf[i],0,
              WSA_FLAG_OVERLAPPED);
    }
}
```

4.5.3　把套接字和 SAP 绑定在一起

事实上，因为 ATM 地址由 20 个字节组成，其中包含许多信息元素，所以它们是相当复杂的。除了最后一个字节外，这些元素的所有其他字节，Winsock 程序员均不用考虑细节问题。NSAP 式的地址和 E.164 地址中的最后一个字节均代表一个选择符的值，该值允许应用程序惟一地定义和指定端点上特定的 SAP。正如前面指出的那样，Winsock 利用 SAP 来建立 ATM 网络上的通信。

当 Winsock 应用程序打算在 ATM 网络上进行通信时，服务器应用程序必须在一个端点上注册一个 SAP，并等待客户机应用程序在注册的 SAP 上与服务器连接。对客户机应用程序来说，这个过程为：通过 ATM_E164 或 ATM_NSAP 地址类型设置 SOCKADDR_ATM 结构，并提供与服务器 SAP 关联在一起的那个 ATM 地址。要建立一个用于监听连接的套接字，应用程序必须先为指定的 AF_ATM 地址族建立一个套接字。这个套接字一旦建立，应用程序就必须利用表 4.2 中定义的 SAP_FIELD_ANY_AESA_SEL、SAP_FIELD_ANY_AESA_REST、ATM_E164 或 ATM_NSAP 地址类型来定义

SOCKADDR_ATM 结构。对 ATM 套接字而言，一旦应用程序调用 Winsock 的 bind API 函数(第 1 章对该函数作了介绍)，就会建立一个 SAP，而上述地址类型则定义了 Winsock 在端点上建立 SAP 的方式。

地址类型 SAP_FIELD_ANY_AESA_SEL 要求 Winsock 建立一个能对任何一种 ATM Winsock 连接进行监听的 SAP，这就是通常所说的对 ATM 地址和选择符进行通配。这意味着监听连接的这个端点只能绑定一个套接字——如果试图把另一个套接字与这个地址类型绑定在一起，就必定会失败，并出现 Winsock 错误 WSAEADDRINUSE。不过，也可以将另一个套接字和指定选择符上的端点显式地绑定在一起。而如果要将 SAP 显式地和端点上的指定选择符绑定在一起，则可以使用地址类型 SAP_FIELD_ANY_AESA_REST 来创建这个 SAP。这就是人们常说的只有 ATM 地址的通配，而没有选择符的通配。对端点上的一个特定选择符而言，一次只能绑定一个套接字，否则 bind 调用就会失败，并返回错误 WSAEADDRINUSE。在使用 SAP_FEILD_ANY_AESA_SEL 类型时，应该在 ATM_ADDRESS 结构中指定一个全部由 0 组成的 ATM 地址。如果使用的是 SAP_FIELD_ANY_AESA_REST 类型，就应该把这个 ATM 地址的前 19 个字节指定为 0，而最后一个字节则应该指明准备使用的选择符的编号。

和显式选择符(SAP_FIELD_ANY_AESA_REST)绑定在一起的套接字优先于和通配选择符(SAP_FIELD_ANY_AESA_SEL)绑定在一起的套接字。连接时，系统会优先采用和显式选择符(SAP_FIELD_ANY_AESA_REST)或显式接口(ATM_NSAP 和 ATM_E164)绑定在一起的套接字(也就是说，如果连接从一个套接字显式监听的指定端点和选择符进入，这个套接字就会获得连接)。只有在没有显式绑定套接字可用的情况下，系统才会用通配选择符套接字来获取连接。

注意 在补充材料中，有一个名为 WSOCKATM.CPP 的 ATM 客户机和服务器应用程序，该程序进一步演示了如何建立一个在 SAP 上监听连接的套接字。

最后，可通过一个名为 Atmadm.exe 的 Windows 实用程序，来获取所有 ATM 地址和端点上的虚拟连接信息。在开发 ATM 应用程序和需要了解端点上哪些接口可用时，这个程序相当有用。表 4.3 中列出的命令行选项都可用。

表 4.3 ATMADM.EXE 选项

参 数	说 明
-c	列出所有的连接(VC)。列出远程地址和本地接口
-a	列出所有已注册的地址(例如所有的本地 ATM 接口及其地址)
-s	输出统计信息(当前调用次数，收到或发出的传信和 ILMI 包数量)

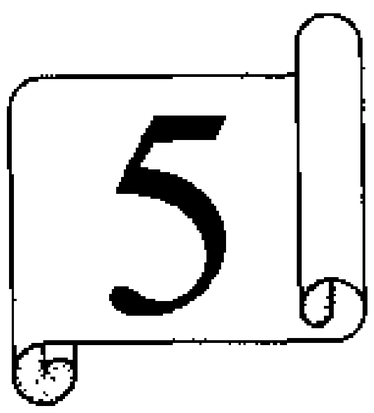
4.5.4　名称解析

目前，ATM 还没有可用于 Winsock 下的名称提供程序。不幸的是，要建立 ATM 网络上的套接字通信，应用程序就不得不对这个长达 20 个字节的 ATM 地址进行指定。第 8 章将讨论 Windows 2000 及 Windows XP 域名空间，一般可用它们来对带有用户易记的服务名的 ATM 地址进行注册。

4.6　小　　结

这一章叙述了 Winsock 支持的其余(非 IP)协议地址族，并说明了各个地址族特有的寻址属性。针对每个地址族，本章还讨论了如何创建套接字，以及如何设置套接字地址结构，以便开始通过协议进行通信。

到现在为止，本书已结束了有关 Winsock 基本通信技术的讨论，并已介绍了可用来创建简单 Winsock 应用程序的所有可用地址族。从第 5 章开始，本书将进入高级 Winsock 主题的讨论，并开始讨论高级的输入输出(I/O)方法。这些方法可用在 Winsock 应用程序中管理输入输出。



Winsock I/O 方法

本章的重点是如何在 Windows 套接字应用程序中对 I/O(输入 / 输出)操作进行管理。Winsock 分别提供了套接字模式和套接字 I/O 模型，可对一个套接字上的 I/O 行为加以控制。其中，套接字**模式**用于决定 Winsock 函数随套接字调用的行为。另一方面，套接字**模型**描述了一个应用程序如何对套接字上的 I/O 进行管理及操作。

Winsock 提供了两种套接字模式：**阻塞模式**和**非阻塞模式**。本章第 1 部分将详细介绍这两种模式，并阐释应用程序如何通过它们来管理 I/O。如本章的后面几部分所述，Winsock 提供了一些有趣的 I/O 模型，有助于应用程序通过某种异步方式，一次对一个或多个套接字上进行的通信加以管理。这些模型包括 select(选择)、WSAAsyncSelect(异步选择)、WSAEventSelect(事件选择)、Overlapped I/O(重叠 I/O)，以及 Completion port(完成端口)等等。本章结束时将对各种套接字模式和 I/O 模型的优缺点进行总结。同时，帮助大家判断到底哪一种最适合自己应用程序的要求。

所有 Windows 平台都支持套接字的阻塞或非阻塞操作模式。然而，并非每种平台都支持各种 I/O 模型。如表 5.1 所示，在当前版本的 Windows CE 中，仅提供了一个 I/O 模型。Windows 98 和 Windows 95(取决于安装的是 Winsock 1 还是 Winsock 2)则支持大多数 I/O 模型，惟一的例外是 I/O 完成端口。而到了 Windows NT 及其后期版本中，每种 I/O 模型都是支持的。

表 5.1　操作系统对套接字 I/O 模型的支持情况

平台	Blocking	Non-blocking Select	WSAAsync Select	WSAEvent Select	Overlapped	Completion Port
Windows CE	支持	支持	不支持	不支持	不支持	不支持
Windows 95 (Winsock 1)	支持	支持	支持	不支持	不支持	不支持
Windows 95 (Winsock 2)	支持	支持	支持	支持	支持	不支持

续表

平台	Blocking	Non-blocking Select	WSAAsync Select	WSAEvent Select	Overlapped	Completion Port
Windows 98	支持	支持	支持	支持	支持	不支持
Windows Me	支持	支持	支持	支持	支持	不支持
Windows NT	支持	支持	支持	支持	支持	支持
Windows 2000	支持	支持	支持	支持	支持	支持
Windows XP	支持	支持	支持	支持	支持	支持

5.1　套接字模式

就像前面提到的那样，Windows 套接字在两种模式下执行 I/O 操作：阻塞模式和非阻塞模式。在阻塞模式下，I/O 操作完成前，执行操作的 Winsock 调用(例如 send 和 recv)会一直等候下去，不会立即返回到程序中(将控制权交还给程序)。而在非阻塞模式下，Winsock 函数无论如何都会立即返回。由于 Windows CE 和 Windows 95(安装了 Winsock 1)平台仅支持极少的 I/O 模型，所以必须对这些平台上运行的应用程序采取一些适当的措施，让阻塞和非阻塞套接字能够满足各种场合的要求。

5.1.1　阻塞模式

因为在一个阻塞套接字上调用任何一个 Winsock API 函数，都会产生相同的后果——耗费或长或短的时间等待，所以对于处在阻塞模式的套接字，必须多加留意。大多数 Winsock 应用程序都是遵照一种“生产者—消费者”模型来编制的。在这种模型中，应用程序需要读取(或写入)指定数量的字节，然后以该数据为基础执行一些计算。下面展示的代码片断便是一个典型的例子：

```
SOCKET   sock;
char     buff[256];
int      done = 0,
         nBytes;
...
while(!done)
{
    nBytes = recv(sock,buff,65);
    if (nBytes == SOCKET_ERROR)
    {
        printf("recv failed with error %d \n",
            WSAGetLastError());
```

```
            Return;
        }
        DoComputationOnData(buff);
    }
    ...
```

这段代码的问题在于，假如没有处于挂起状态的数据，那么 recv 函数可能永远都无法返回。从语句中可以看出：只有从系统的输入缓冲区中读回一些内容，才允许返回。有些程序员可能会在 recv 中使用 MSG_PEEK 标志，或者调用 ioctlsocket(设置 FIONREAD 选项)，从而在系统的缓冲区中，事先查看是否存在必要的字节数量。然而，在不实际读入数据的前提下，仅仅查看数据(如实际读入数据，便会从系统缓冲区中将其删除)是一种不好的编程习惯，应尽力避免。为了检查有多少个字节可用，必须发出一个或者更多的系统调用，所以在查看数据的时候，对系统造成的开销是极大的。当然，还需要牵涉到进行实际的 recv 调用，将数据从系统缓冲区内删除的开销。为了避免这一情况，需要防止在没有连接查看系统网络缓冲的情况下，由于数据的缺乏(这可能是网络出了故障，也可能是客户机出了问题)而造成应用程序完全陷于凝固状态。为达此目的，一个办法是将应用程序划分为一个读线程，以及一个计算线程。两个线程都共享同一个数据缓冲区。对这个缓冲区的访问需要受到一定的限制，这是用一个同步对象来实现的，例如一个事件或者一个互斥体。读线程的职责是从网络上连续地读入数据，并将数据置入共享缓冲区内。读线程读取了计算线程开始工作所需要的最少数据量后，便会触发一个事件，通知计算线程开始工作。然后，计算线程从缓冲区中删除一个数据块，进行要求的计算。

下述程序段分别提供了两个函数，采取的便是上述办法。在这两个函数中，一个负责读取网络数据(ReadThread)，另一个则负责对数据执行计算(ProcessThread)。

```
#define MAX_BUFFER_SIZE 4096

//初始化关键部分(数据)，并在创建两个线程之前，创建一个自动重置的事件(hEvent)

CRITICAL_SECTION      data;
HANDLE                hEvent;
SOCKET                sock;
TCHAR                 buff[MAX_BUFFER_SIZE];
Int                   done = 0;

//创建并连接套接字
...
//读线程
void ReadThread(void)
{
    int nTotal = 0,
        nRead = 0,
        nLeft = 0,
```

```
    nBytes = 0;
while (!done)
{
    nTotal = 0;
    nLeft = NUM_BYTES_REQUIRED;
    //不管多少字节组成的数据，都足够用来进行处理(也就是说，只要非零即可)
    while (nTotal != NUM_BYTES_REQUIRED)
    {
        EnterCriticalSection(&data);
        nRead = recv(sock,&(buff[MAX_BUFFER_SIZE -nBytes]),
        nLeft,0);
        if (nRead == -1)
        {
            printf("error \n");
            ExitThread();
        }
        nTotal += nRead;
        nLeft -= nRead;
        nBytes += nRead;
        LeaveCriticalSection(&data);
    }
    SetEvent(hEvent);
  }
}

//计算线程
void ProcessThread(void)
{
    WaitForSingleObject(hEvent);
    EnterCriticalSection(&data);
    DoSomeComputationOnData(buff);
    //从输入缓冲区中删除已处理过的数据，并将余下的数据移到数组起始处
    nBytes -= NUM_BYTES_REQUIRED;
    LeaveCriticalSection(&data);
}
```

对阻塞套接字来说，它的一个缺点在于：应用程序很难同时通过多个建好连接的套接字通信。使用前述的方案，可对应用程序进行修改，令其为连接好的每个套接字都分配一个读线程，以及一个数据处理线程。尽管这仍然会增大一些开销，但的确是一种可行的方案。惟一的缺点便是伸缩性不好，以后想同时处理大量套接字时就不大方便了。

5.1.2　非阻塞模式

除了阻塞模式，还可考虑采用非阻塞模式的套接字。尽管这种套接字在使用上存在一些难度，但

只要排除了这个困难，它在功能上还是非常强大的。除具备阻塞套接字已有的各种优点之外，它还进行了少许扩充，功能更强。下面的示例展示了如何创建一个套接字，并将其置为非阻塞模式。

```
SOCKET              s;
unsigned long       ul = 1;
int                 nRet;

s = socket(AF_INET,SOCK_STREAM,0);
nRet = ioctlsocket(s,FIONBIO,(unsigned long * )&ul);
if (nRet == SOCKET_ERROR)
{
//未能将套接字置入非阻塞模式
}
```

将一个套接字置为非阻塞模式之后，处理收发数据或处理连接的 Winsock API 调用会立即返回。大多数情况下，这些调用在失败时会返回一个 WSAEWOULDBLOCK 错误，这意味着请求的操作在调用期间没有足够时间来完成。举个例子来说，假如在系统的输入缓冲区中，尚不存在被挂起的数据，那么 recv 调用就会返回 WSAEWOULDBLOCK 错误。通常，需要重复调用同一个函数，直至获得成功的返回代码。表 5.2 对常见的 Winsock 调用返回的 WSAEWOULDBLOCK 错误的含义进行了总结。

表 5.2 非阻塞套接字上的 WSAEWOULDBLOCK 错误

函数名	说 明
WSAAccept 和 accept	应用程序没有收到连接请求。再次调用，便可检查连接情况
Closesocket	在大多数情况下，这个错误意味着已用 SO_LINGER 选项调用了 setsockopt，而且已设定了一个非零的超时值
WSAConnect 和 connect	连接已初始化。再次调用，便可检查连接是否完成
WSARecv、recv、WSARecvFrom 和 recvfrom	没有收到数据。稍后再次检查
WSASend、send、WSASendTo 和 sendto	外出数据无缓冲区可用。稍后再试

由于非阻塞调用会频繁返回 WSAEWOULDBLOCK 错误，所以在任何时候，都应仔细检查所有返回代码，并作好失败的准备。许多程序员易犯的一个错误便是持续不停地调用一个函数，直到它返回成功的消息为止。例如，假定在一个紧凑的循环中不断地调用 recv，以读入 200 个字节的数据，那么与使用前述的 MSG_PEEK 标志来轮询一个阻塞套接字相比，前一种做法并没有优势可言。为此，Winsock 的套接字 I/O 模型可帮助应用程序判断一个套接字何时可供读写。

阻塞和非阻塞套接字模式都存在着优点和缺点。其中，从概念的角度说，阻塞套接字更容易使用，

但在应付建立连接的多个套接字时，或在数据的收发量不均、时间不定时，却显得极难管理。而另一方面，由于需要编写更多的代码，以便在每个 Winsock 调用中，对可能收到的 WSAEWOULDBLOCK 错误加以处理，所以非阻塞套接字便显得有些难于操作。在这些情况下，可考虑使用套接字 I/O 模型，它将帮助应用程序通过一种异步方式，同时对一个或多个套接字上进行的通信加以管理。

5.2　套接字 I/O 模型

共有 6 种类型的套接字 I/O 模型，可让 Winsock 应用程序对 I/O 进行管理，它们包括：blocking(阻塞)、select(选择)、WSAAsyncSelect(异步选择)、WSAEventSelect(事件选择)、overlapped(重叠)以及 completionport(完成端口)。这一节将向大家解释每种 I/O 模型的特点，同时讲述如何利用这些模型，来开发能同时管理一个或多个套接字请求的应用程序。在本书的补充材料中，针对每种 I/O 模型，大家都可以找到一个或者更多的示范应用程序。这些程序阐述了如何利用每种模型的原理，开发一个简单的 TCP 回应服务器。

应注意到，从技术角度讲，可以存在一个直接的非阻塞 I/O 模型——即用 ioctlsocket 函数将所有套接字置为非阻塞模式的一个应用程序。然而，因为应用程序会花费绝大部分时间在套接字处理和 I/O 操作中循环，直到这些处理和操作成功，所以这样做将变得很难管理。

5.2.1　阻塞模型

因为阻塞模型最简单，也最直接，所以大多数 Winsock 编程人员都从这种模型开始。第 1 章中 Winsock 示例使用的就是阻塞模型。前面已经说过，对于采用这个模型的应用程序，在处理 I/O 时，每个套接字连接通常使用一个或两个线程。之后每个线程都将发出阻塞操作，如 send 和 recv。

阻塞模型的优势是其简洁性。对于简单的应用程序和快速原型化，这个模型非常有用。因为创建多线程会消耗宝贵的系统资源，缺点是很难将它扩展到有很多连接的情况。

5.2.2　select 模型

select(选择)模型是 Winsock 中另一个应用广泛的 I/O 模型。之所以称其为“select 模型”，是由于它的工作原理是利用 select 函数，实现对 I/O 的管理。该模型最初设计是在不使用 Unix 操作系统的计算机上实现的，它们采用的是 Berkeley 套接字方案。select 模型已集成到 Winsock 1.1 中，它使那些想避免在套接字调用过程中被阻塞的应用程序，能够采取一种有序的方式，同时进行对多个套接字的管理。由于 Winsock 1.1 向后兼容于 Berkeley 套接字实施方案，所以假如有一个 Berkeley 套接字应用程序使用了 select 函数，那么从理论角度讲，毋需对其进行任何修改，便可正常运行。

select 函数可用于判断套接字上是否存在数据，或者能否向一个套接字写入数据。之所以要设计

这个函数，其目的是防止应用程序在套接字处于阻塞模式中时，在 I/O 绑定调用(如 send 或 recv)过程中进入阻塞状态；同时也防止在套接字处于非阻塞模式中时，产生 WSAEWOULDBLOCK 错误。除非满足事先用参数规定的条件，否则 select 函数在进行 I/O 操作时会阻塞。select 函数的原型如下：

```
int select(
    int nfds,
    fd_set FAR * readfds,
    fd_set FAR * writefds,
    fd_set FAR * exceptfds,
    const struct timeval FAR * timeout
);
```

其中，第 1 个参数 nfds 会被忽略。之所以仍然要提供这个参数，只是为了保持与早期的 Berkeley 套接字应用程序的兼容。大家注意到这里有 3 个 fd_set 参数：一个用于检查可读性(readfds)，一个用于检查可写性(writefds)，另一个用于带外数据(exceptfds)。从根本上说，fd_set 数据类型代表着一系列特定套接字的集合。其中，readfds 集合包括符合下述任何一个条件的套接字：

- 有数据可以读入
- 连接已经被关闭、重启或终止
- 假如已调用了 listen，而且有一个连接正处于挂置状态，那么 accept 函数调用会成功

writefds 集合包括符合下述任何一个条件的套接字：

- 有数据可以发出
- 如果正在对一个非阻塞连接调用进行处理，则连接就成功了

最后，exceptfds 集合包括符合下述任何一个条件的套接字：

- 假如正在对一个非阻塞连接调用进行处理，连接尝试就会失败
- 有 OOB(Out-of-band，带外)数据可供读取

假定我们想测试一个套接字是否可读，必须将这个套接字增添到 readfds 集合中，再等待 select 函数完成。当 select 调用完成之后，必须判断这个套接字是否仍为 readfds 集合的一部分。若答案是肯定的，便表明该套接字可读，可立即着手从它上面读取数据。在 3 个参数 (readfds、writefds 和 exceptfds)，任何两个都可以是空值(NULL)；但是，至少有一个不能为空值。在任何不为空的集合中，必须包含至少一个套接字句柄；否则，select 函数便没有任何东西可以等待。最后一个参数 timeout 对应的是一个指针，它指向一个 timeval 结构，用于决定 select 等待 I/O 操作完成时，最多等待多长的时间。如果 timeout 是一个空指针，那么 select 调用会无限期处于阻塞状态，直到至少有一个描述符与指定的条件相符后才结束。对 timeval 结构的定义如下：

```
struct timeval
{
    long tv_sec;
```

```
    long tv_usec;
};
```

其中，tv_sec 字段以秒为单位指定等待时间；tv_usec 字段则以毫秒为单位指定等待时间。若将超时值设置为{0,0}，表明 select 会立即返回，允许应用程序对 select 操作进行轮询。但出于对性能方面的考虑，应避免这样的设置。select 成功完成后，会在 fd_set 结构中，返回被挂起的 I/O 操作的所有套接字句柄的总量。若超过 timeval 设定的时间，便会返回 0。不管由于什么原因，假如 select 调用失败，都会返回 SOCKET_ERROR。

用 select 对套接字进行监听之前，应用程序必须将套接字句柄分配给一个集合，设置好一个或所有的读、写以及例外的 fd_set 结构。将一个套接字分配给任何一个集合后，再来调用 select，便可知道某个套接字上是否正在发生 I/O 活动。Winsock 提供了下列宏操作，可用来针对 I/O 活动，对 fd_set 集合进行处理与检查：

- **FD_ZERO(* set)**　将 set 初始化成空集合。集合在使用前都要清空
- **FD_CLR(s,* set)**　从 set 中删除套接字 s
- **FD_ISSET(s,* set)**　检查 s 是否为 set 集合的一名成员；如答案是肯定的，则返回 TRUE
- **FD_SET(s,* set)**　将套接字 s 加入集合 set 中

假定我们想知道是否可从一个套接字中安全地读取数据，同时不会陷于阻塞状态，便可使用 FD_SET 宏，将这个套接字分配给 fd_read 集合，再调用 select。要想检测这个套接字是否仍属 fd_read 集合的一部分，可使用 FD_ISSET 宏。采用下述步骤，使可完成用 select 操作一个或多个套接字句柄的全过程：

1. 使用 FD_ZERO 宏，初始化自己感兴趣的每一个 fd_set。
2. 使用 FD_SET 宏，将套接字句柄分配给自己感兴趣的每个 fd_set。
3. 调用 select 函数，然后等待直到 I/O 活动在指定的 fd_set 集合中设置好了一个或多个套接字句柄。select 完成后，会返回在所有 fd_set 集合中设置的套接字句柄总数，并对每个集合进行相应的更新。
4. 根据 select 的返回值，应用程序便可判断出哪些套接字存在着被搁置的 I/O 操作——具体的方法是使用 FD ISSET 宏，对每个 fd_set 集合进行检查。
5. 知道了每个集合中被挂起的 I/O 操作之后，对 I/O 进行处理，然后返回步骤 1，继续处理 select。

select 返回后，它会修改每个 fd_set 结构，删除那些不存在被挂起的 I/O 操作的套接字句柄。这正是在上述的步骤 4 中，为何要使用 FD_ISSET 宏来判断某个特定的套接字是否仍在集合中的原因。下面的示例代码阐述了为单个套接字设置 select 模型所需的一系列基本步骤。若想在这个应用程序中添加更多的套接字，只需为需要添加的套接字保留一个列表，或保留它们的一个数组。

```
SOCKET        s;
fd_set        fdread;
```

```
int             ret;

//创建套接字，接受连接

//在套接字上管理 I/O
while(TRUE)
{
    //在调用 select()之前始终清除读出集
    FD_ZERO(&fdread);

    //将 s 套接字添加到读出集
    FD_SET(s,&fdread);
    if ((ret = select(0,&fdread,NULL,NULL,NULL))
        == SOCKET_ERROR)
    {
        //条件有错
    }

    if (ret >0)
    {
        //在这个简单的例子中，select()的返回值为 1。处理多个套接字的应用程序可返回比 1 大的值。
        //在这里，应用程序应该检查一下，看看该套接字是否属于集合的一个部分。
        if (FD_ISSET(s,&fdread))
        {
            //套接字 s 上已发生了一个事件
        }
    }
}
```

使用 select 的优势是，能够从单个线程的多个套接字上进行多重连接及 I/O。这就避免了伴随阻塞套接字和多重连接的线程剧增。但可以加到 fd_set 结构中的最大套接字数量是一个不好的地方。默认状态下，最大数量由 FD_SETSIZE 定义，该值在 WINSOCK2.H 中定义为 64。为了提高这个上限，应用程序可将 FD_SETSIZE 定义得更大一些。这个定义必须在包含 WINSOCK2.H 之前出现。同时，下层提供程序强加了一个 fd_set 的最大值，通常情况下是 1 024，但不保证一定是。最后，针对一个较大的 FD_SETSIZE，在调用 select 之前，应考虑设置 1 000 个套接字的效果；在调用返回之后，检查这 1 000 个套接字中的每一个是否都被设置了，这样便可以看出效果如何。

5.2.3 WSAAsyncSelect 模型

Winsock 提供了一个有用的异步 I/O 模型。利用这个模型，应用程序可在一个套接字上，接收以 Windows 消息为基础的网络事件通知。具体的做法是在建好一个套接字后，调用 WSAAsyncSelect 函数。在继续讲解前，需要说明一个细微差别。WSAAsyncSelect 和 WSAEventSelect 模型提供了读写数据能力的异步通知。但它们不提供异步数据传送，而重叠及完成端口模型却提供异步数据传送。

WSAAsyncSelect 模型最早出现于 Winsock 的 1.1 版本中，用于帮助应用程序开发者面向一些早期的 16 位 Windows 平台(如 Windows for Workgroups)，适应其合作性的多任务消息环境。现在应用程序仍可从这种模型中得到好处，特别是在它们用一个标准的 Windows 过程(常称为“winproc”)对窗口消息进行管理的时候。该模型也得到了 MFC(Microsoft Foundation Class，微软基类)对象 CSocket 的采纳。

消息通知

要想使用 WSAAsyncSelect 模型，在应用程序中，首先必须用 CreateWindow 函数创建一个窗口，再为该窗口提供一个窗口过程支持函数(Winproc)。亦可使用一个对话框，为其提供一个对话过程来代替窗口过程，这是因为对话框本质也是窗口。考虑到我们的目的，我们用一个简单的窗口来演示这种模型，这个窗口带有一个简单的支持窗口的过程。设置好窗口的框架后，便可以开始创建套接字，并调用 WSAAsyncSelect 函数，打开窗口消息通知。该函数的定义如下：

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

其中，s 参数指定的是我们感兴趣的那个套接字。hWnd 参数指定的是一个窗口句柄，它标识的是网络事件发生之后，想要收到通知消息的那个窗口或对话框。wMsg 参数指定在发生网络事件时，打算接收的消息。该消息将被投递到由 hWnd 窗口句柄所标识的那个窗口。通常，应用程序需要将这个消息设为比 Windows 的 WM_USER 大的一个值，以避免网络窗口消息与预定义的标准窗口消息发生混淆。最后一个参数是 lEvent，它代表的是一个位掩码，指定一系列网络事件的组合(请参见表 5.3)，应用程序感兴趣的便是这一系列事件。大多数应用程序通常感兴趣的网络事件类型包括：FD_READ、FD_WRITE、FD_ACCEPT、FD_CONNECT 和 FD_CLOSE。当然，到底使用 FD_ACCEPT，还是使用 FD_CONNECT 类型，要取决于应用程序的身份到底是一个客户机还是一个服务器。如果应用程序同时对多个网络事件有兴趣，只需对各种类型执行一次简单的按位 OR(或)运算，然后将它们分配给 lEvent 就可以了。例如：

```
WSAAsyncSelect(s,hwnd,WM_SOCKET,
    FD_CONNECT | FD_READ | FD_WRITE | FD_CLOSE);
```

这样，应用程序便可在套接字 s 上，接收到有关连接、发送、接收以及关闭套接字这一系列网络事件的通知。特别要注意的是，多个事件务必在套接字上一次完成注册。另外还要注意的是，一旦在某个套接字上启用了事件通知，那么以后除非明确调用 closesocket 命令，或者由应用程序针对这个套接字调用了 WSAAsyncSelect，从而更改注册的网络事件类型，否则，事件通知总是有效。若将 lEvent 参数设为 0，则效果相当于停止在套接字上进行的所有网络事件通知。

若应用程序针对一个套接字调用了 WSAAsyncSelect，那么套接字的模式会从阻塞模式自动变成非阻塞模式，我们在前面已提到过这一点。这样，假如调用了像 WSARecv 这样的 WinsockI/O 函数，但当时却并没有数据可用，那么必然会造成调用的失败，并返回 WSAEWOULDBLOCK 错误。为防止这一点，应用程序应依赖于由 WSAAsyncSelect 的 uMsg 参数指定的用户自定义窗口消息，来指示网络事件类型何时在套接字上发生。

表 5.3 用于 WSAAsyncSelect 函数的网络事件类型

事件类型	含 义
FD_READ	应用程序想要接收有关是否可读的通知，以便读入数据
FD_WRITE	应用程序想要接收有关是否可写的通知，以便写入数据
FD_OOB	应用程序想接收是否有 OOB 数据抵达的通知
FD_ACCEPT	应用程序想接收与传入的连接有关的通知
FD_CONNECT	应用程序想接收一个已完成连接的通知或者一个多点 join 操作的通知
FD_CLOSE	应用程序想接收与套接字关闭有关的通知
FD_QOS	应用程序想接收套接字 QOS 发生变化的通知
FD_GROUP_QOS	应用程序想接收套接字组 QOS 发生变化的通知(为未来套接字组的使用所保留)
FD_ROUTING_INTERFACE_CHANGE	应用程序想接收在指定的方向上路由接口发生变化的通知
FD_ADDRESS_LIST_CHANGE	应用程序想接收本地地址列表针对套接字的协议家族发生变化的通知

应用程序在一个套接字上成功调用了 WSAAsyncSelect 之后，它会在与 hWnd 窗口句柄参数相关联的窗口过程中，以 Windows 消息的形式，接收网络事件通知。窗口过程通常定义如下：

```
LRESULT CALLBACK WindowProc(
    HWND hWnd,
    UINT uMsg,
    WPARAM wParam,
    LPARAM lParam
);
```

其中，hWnd 参数指定一个窗口的句柄，对窗口过程的调用正是由该窗口发出的。uMsg 参数指定

需要对哪些消息进行处理。我们感兴趣的是 WSAAsyncSelect 调用中定义的消息。wParam 参数指定的是一个套接字，该套接字上发生了一个网络事件。假若同时为这个窗口过程分配了多个套接字，这个参数的重要性便显示出来了。在 lParam 参数中，包含了两方面重要的信息。其中，lParam 的低字(低位字)指定了已经发生的网络事件，而 lParam 的高字(高位字)包含了可能出现的任何错误代码。

网络事件消息抵达窗口过程后，应用程序首先应检查 lParam 的高字位，以判断套接字上是否发生了网络错误。这里有一个特殊的宏：WSAGETSELECTERROR，可用它返回高字位包含的错误信息。若应用程序发现套接字上没有产生任何错误，接着便应调查到底是哪个网络事件类型造成了这条 Windows 消息的触发——具体的做法便是读取 lParam 的低字位内容。此时可使用另一个特殊的宏 WSAGETSELECTEVENT 返回 lParam 的低字部分。

下面的示例演示了如何使用 WSAAsyncSelect 这种 I/O 模型来实现窗口消息的管理。程序代码着重强调的是开发一个基本服务器应用程序所涉及到的主要步骤，所以忽略了开发完整的 Windows 应用程序需要涉及到的大量编程细节。

```
#define WM_SOCKET WM_USER +1

#include <winsock2.h>
#include <windows.h>

int  WINAPI WinMain(HINSTANCE hInstance,
HINSTANCE hPrevInstance,LPSTR lpCmdLine,
int nCmdShow)
{
    WSADATA wsd;
    SOCKET Listen;
    SOCKADDR_IN InternetAddr;
    HWND Window;

    //创建窗口，并将下面的 ServerWinProc 分配给它

    Window = CreateWindow();

    //启动并创建套接字

    WSAStartup(MAKEWORD(2,2),&wsd);
    Listen = socket (AF_INET,SOCK_STREAM,IPPROTO_TCP);

    //将套接字绑定到 5150 端口并开始监听连接

    InternetAddr.sin_family= AF_INET;
    InternetAdar.sin_addr.s_addr = htonl(INADDR_ANY);
    InternetAddr.sin_port = htons(5150);
```

```
    bind(Listen,(PSOCKADDR)&InternetAddr,
        sizeof(InternetAddr));

    //使用上面定义的 WM_SOCKET 在新套接字上设置窗口消息通知
    WSAAsyncSelect(Listen,Window,WM_SOCKET,
        FD_ACCEPT | FD_CLOSE);
    listen(Listen,5);
    //转换并分派窗口消息，直到应用程序终止

    while (1){
    //...
    }
}
BOOL CALLBACK ServerWinProc(HWND hDlg,UINT wMsg,
    WPARAM wParam,LPARAM lParam)
{
    SOCKET Accept;
    switch(wMsg)
    {
        case WM_PAINT:
            //处理窗口画图消息
            break;
            case WM_SOCKET:
            //使用 WSAGETSELECTERROR()宏来判断套接字上是否发生了错误
            if (WSAGETSELECTERROR(lParam))
            {
                //显示错误，关闭套接字
                closesocket((SOCKET)wParam);
                break;
            }
            //确定在套接字上发生了什么事件
            switch(WSAGETSELECTEVENT(lParam))
            {
                case FD_ACCEPT:
                //接受一个传入的连接
                Accept = accept(wParam,NULL,NULL);
                //让接受套接字为读、写及关闭通知做好准备
                WSAAsyncSelect(Accept,hDlg,WM_SOCKET,
                FD_READ | FD_WRITE | FD_CLOSE);
                break;
                case FD_READ:
                //从 wParam 中的套接字中检索数据
                break;
                case FD_WRITE:
                //wParam 中的套接字已准备好发送数据
```

```
                break;
                case FD_CLOSE:
                //连接已关闭
                closesocket((SOCKET)wParam);
                break;
            }
            break;
    }
    return TRUE;
}
```

最后一个特别有价值的问题是应用程序如何对 FD_WRITE 事件通知进行处理。只有在 3 种条件下，FD_WRITE 通知才会发出：

- 使用 connect 或 WSAConnect，一个套接字首次建立了连接
- 使用 accept 或 WSAAccept，套接字被接受以后
- 若 send、WSASend、sendto 或 WSASendTo 操作失败，返回了 WSAEWOULDBLOCK 错误，而且缓冲区的空间变得可用时

因此，作为一个应用程序，自收到首条 FD_WRITE 消息开始，便应认为必然能在一个套接字上发出数据，直至 send、WSASend、sendto 或 WSASendTo 返回了套接字错误 WSAEWOULDBLOCK。经过了这样的失败以后，要再用另一条 FD_WRITE 通知应用程序可以再次发送数据。

WSAAsyncSelect 模型有许多优点。最突出的一个方面是它可以在系统开销不大的情况下同时处理许多连接，而 select 模型需要建立 fd_set 结构。缺点是，即使应用程序不需要窗口(例如服务或控制台应用程序)，它也不得不额外使用一个窗口。同时，用一个单窗口程序来处理成千上万的套接字中的所有事件，很可能成为性能瓶颈(意味着这个模型伸缩性不大好了)。

5.2.4　WSAEventSelect 模型

Winsock 提供了另一个有用的异步事件通知 I/O 模型。和 WSAAsyncSelect 模型类似的是，它也允许应用程序在一个或多个套接字上，接收以事件为基础的网络事件通知。对于表 5.3 总结的、由 WSAAsyncSelect 模型采用的网络事件来说，它们均可原封不动地移植到新模型中。该模型最主要的差别在于网络事件通知是由事件对象句柄完成的，而不是通过窗口例程完成。

事件通知

事件通知模型要求应用程序针对打算使用的每一个套接字，首先创建一个事件对象。创建方法是调用 WSACreateEvent 函数，它的定义如下：

```
WSAEVENT WSACreateEvent(void);
```

WSACreateEvent 函数的返回值很简单，就是一个人工重设的事件对象句柄。一旦得到了事件对

象句柄之后，必须将它与某个套接字关联在一起，同时注册感兴趣的网络事件类型，如表 5.3 所示。要做到这一点，方法是调用 WSAEventSelect 函数，对它的定义如下：

```
int WSAEventSelect(
    SOCKET s,
    WSAEVENT hEventObject,
    long lNetworkEvents
);
```

其中，s 参数代表程序感兴趣的套接字。hEventObject 参数指定要与套接字关联在一起的事件对象(用 WSACreateEvent 取得的那一个)。而最后一个参数 lNetworkEvents 则对应一个位掩码，用于指定应用程序感兴趣的各种网络事件类型的组合(如表 5.3 所示)。要想获知对这些事件类型的详细说明，请参考已讨论过的 WSAAsyncSelect I/O 模型。

为 WSAEventSelect 创建的事件有两种工作状态和两种工作模式。其中，两种工作状态分别是**已传信** (signaled)和**未传信** (non-signaled)。工作模式则包括**人工重设** (manualreset)和**自动重设** (autoreset)。WSACreateEvent 最开始在一种未传信的工作状态中，并用一种人工重设模式，来创建事件句柄。若网络事件触发了与一个套接字关联在一起的事件对象，工作状态便会从未传信转变成已传信。由于事件对象是在一种人工重设模式中创建的，所以在完成了一个 I/O 请求的处理之后，应用程序需要负责将工作状态从已传信更改为未传信。要做到这一点，可调用 WSAResetEvent 函数，对它的定义如下：

```
BOOL WSAResetEvent(WSAEVENT hEvent);
```

该函数惟一的参数便是一个事件句柄；基于调用是成功还是失败，该函数会分别返回 TRUE 或 FALSE。应用程序完成了对某个事件对象的处理后，便应调用 WSACloseEvent 函数释放由事件句柄使用的系统资源。对 WSACloseEvent 函数的定义如下：

```
BOOL WSACloseEvent(WSAEVENT hEvent);
```

该函数也要将一个事件句柄作为自己惟一的参数，并会在成功后返回 TRUE，失败后返回 FALSE。

套接字同一个事件对象句柄关联在一起后，应用程序便可开始 I/O 处理；这就需要应用程序等待网络事件触发事件对象句柄的工作状态。WSAWaitForMultipleEvents 函数的设计宗旨便是用来等待一个或多个事件对象句柄，并在事先指定的一个或所有句柄进入已传信状态后，或在超过了一个规定的时间周期后，立即返回。下面是 WSAWaitForMultipleEvents 函数的定义：

```
DWORD WSAWaitForMultipleEvents(
    DWORD cEvents,
    const WSAEVENT FAR * lphEvents,
    BOOL fWaitAll,
    DWORD dwTimeout,
    BOOL fAlertable
);
```

其中，cEvents 和 lphEvents 参数定义了由 WSAEVENT 对象构成的一个数组。cEvents 指定的是这个数组中事件对象的数量，而 lphEvents 是一个指针，用于直接引用该数组。要注意的是，WSAWaitForMultipleEvents 只能支持由 WSA_MAXIMUM_WAIT_EVENTS 对象规定的一个最大值，在此这个最大值定义成 64。因此，对于发出 WSAWaitForMultipleEvents 调用的每个线程，该 I/O 模型一次最多都只能支持 64 个套接字。假如想让这个模型同时管理多于 64 个的套接字，必须创建额外的工作器线程，以便等待更多的事件对象。fWaitAll 参数指定了 WSAWaitForMultipleEvents 如何等待在事件数组中的对象。若将该参数设为 TRUE，那么只有等 lphEvents 数组内包含的所有事件对象都已进入已传信状态，函数才会返回；但若设为 FALSE，任何一个事件对象进入已传信状态时，函数就会返回。就后一种情况来说，返回值指出了到底是哪个事件对象造成了函数的返回。通常，应用程序应将该参数设为 FALSE，一次只为一个套接字事件提供服务。dwTimeout 参数规定了 WSAWaitForMultipleEvents 等待一个网络事件发生时，最多可等待多长时间，这是一项超时设定，以毫秒为单位。超过规定的时间，函数就会立即返回，即使由 fWaitAll 参数规定的条件尚未满足也如此。如超时值为 0，函数会检测指定的事件对象的状态，并立即返回。这样，应用程序实际使可实现对事件对象的轮询。假如没有可处理的事件，WSAWaitForMultipleEvents 便会返回 WSA_WAIT_TIMEOUT。如果 dwsTimeout 被设为 WSA_INFINITE，那么只有在网络事件传信了一个事件对象后，函数才会返回。最后一个参数是 fAlertable，在使用 WSAEventSelect 模型的时候，它可以被忽略，且应设为 FALSE。该参数主要用在重叠 I/O 模型中完成例程的处理过程中。本章后面将对重叠 I/O 模型进行详述。

应注意到，一次只服务一个已传信事件(将参数 fWaitAll 设为 FALSE)，就可能让套接字一直“挨饿”，且可能持续到事件数组的末尾。看看下面的代码：

```
WSAEVENT            HandleArray[WSA_MAXIMUM_WAIT_EVENTS];
Int                 WaitCount= 0,ret,index;

//将事件句柄分配到HandleArray
while (1){
  ret = WSAWaitForMultipleEvents(
          WaitCount,
          HandleArray,
          FALSE,
          WSA_INFINITE,
          TRUE);
  if ((ret != WSA_WAIT_FAILED)&&(ret != WSA_WAIT TIMEOUT)){
     index = ret -WSA_WAIT_OBJECT_0;

          //服务事件在HandleArray[index]上被传信

          WSAResetEvent(HandleArray[index]);
     }
```

```
}
```

如果和事件数组中索引 0 相关联的套接字连续地接收数据，以至于事件被重置之后，又有额外数据到达，并导致事件再次被传信，则数组中其余事件就会被闲置。这当然不是理想的状况。只要回路中有一个事件被传信并得到处理，就应该检查数组中的所有事件，看看它们是否也被传信。在有事件被传信之后，对每个事件的句柄使用 WSAWaitForMultipleEvents，并将 dwTimeOut 指定为 0，即可达到上述目的。

若 WSAWaitForMultipleEvents 收到一个事件对象的网络事件通知，便会返回一个值，指出造成函数返回的事件对象。这样，应用程序便可引用事件数组中已传信的事件，并检索与那个事件对应的套接字，判断到底是在哪个套接字上，发生了什么样的网络事件类型。对事件数组中的事件进行引用时，应该用 WSAWaitForMultipleEvents 的返回值，减去预定义值 WSA_WAIT_EVENT_0，从而得到具体的引用值(即索引位置)。如下例所示：

```
Index = WSAWaitForMultipleEvents(...);
MyEvent = EventArray[Index -WSA_WAIT_EVENT_0];
```

知道了造成网络事件的套接字后，接下来可调用 WSAEnumNetworkEvents 函数，调查发生了哪些网络事件。该函数的定义如下：

```
int WSAEnumNetworkEvents(
    SOCKET s,
    WSAEVENT hEventObject,
    LPWSANETWORKEVENTS lpNetworkEvents
);
```

s 参数对应于造成了网络事件的套接字。hEventObject 参数则是可选的；它指定了一个事件句柄，对应于打算重设的那个事件对象。由于我们的事件对象处在一种已传信状态，所以可将它传入，令其自动成为未传信状态。如果不想用 hEventObject 参数来重设事件，那么可使用 WSAResetEvent 函数对事件进行人工重设。最后一个参数是 lpNetworkEvents，代表一个指针，指向 WSANETWORKEVENTS 结构，用于检索套接字上发生的网络事件类型以及可能出现的任何相关的错误代码。下面是 WSANETWORKEVENTS 结构的定义：

```
typedef struct _WSANETWORKEVENTS
{
    long lNetworkEvents;
    int iErrorCode[FD_MAX_EVENTS];
}WSANETWORKEVENTS,FAR * LPWSANETWORKEVENTS;
```

lNetworkEvents 参数指定了一个值，对应于该套接字上发生的所有网络事件类型(参见表 5.3)。

注意 当一个事件进入传信状态时，可能会同时发生多个网络事件类型。例如，一个繁忙的服务器应用程序可能同时收到 FD_READ 和 FD_WRITE 通知。

iErrorCode 参数指定的是一个错误代码数组，这个数组同 lNetworkEvents 中的事件关联在一起。针对每个网络事件类型，都存在着一个特殊的事件索引，它与事件类型的名称类似，只是要在事件名称后面添加一个“_BIT”作为后缀字符串。例如，对 FD_READ 事件类型来说，iErrorCode 数组的索引标识符便是 FD_READ_BIT。下述代码段对此进行了阐释(针对 FD_READ 事件)：

```
//处理 FD_READ 通知
if (NetworkEvents.lNetworkEvents &FD_READ)
{
    if (NetworkEvents.iErrorCode[FD_READ_BIT]!= 0)
    {
        printf("FD_READ failed with error %d \n",
        NetworkEvents.iErrorCode[FD_READ_BIT]);
    }
}
```

完成了对 WSANETWORKEVENTS 结构中的事件的处理之后，应用程序应在所有可用的套接字上，继续等待更多的网络事件。下面的示例阐释了如何使用 WSAEventSelect 这种 I/O 模型，来开发一个服务器应用程序，同时对事件对象进行管理。这个程序主要着眼于开发一个基本的服务器应用程序要涉及到的步骤，这个应用程序要同时负责一个或多个套接字的管理。

```
SOCKET SocketArray[WSA_MAXIMUM_WAIT_EVENTS];
WSAEVENT EventArray[WSA_MAXIMUM_WAIT_EVENTS],
         NewEvent;
SOCKADDR_IN InternetAddr;
SOCKET Accept,Listen;
DWORD EventTotal = 0;
DWORD Index,i;

//创建一个 TCP 套接字在 5150 端口上进行监听
Listen = socket (PF_INET,SOCK_STREAM,0);
InternetAddr.sin_family= AF_INET;
InternetAddr.sin_addr.s_addr = htonl(INADDR_ANY);
InternetAddr.sin_port = htons(5150);

bind(Listen,(PSOCKADDR)&InternetAddr,
    sizeof(InternetAddr));

NewEvent = WSACreateEvent();

WSAEventSelect(Listen,NewEvent,
    FD_ACCEPT |FD_CLOSE);

listen(Listen,5);
```

```
SocketArray[EventTotal]= Listen;
EventArray[EventTotal]= NewEvent;
EventTotal++;

while(TRUE)
{
    //等候所有套接字上的网络事件
    Index = WSAWaitForMultipleEvents(EventTotal,
    EventArray,FALSE,WSA_INFINITE,FALSE);
    Index = Index -WSA_WAIT_EVENT_0;

    //遍历所有事件，查看被传信的事件是否多于一个
    for(i= Index;i <EventTotal ;i++
    {
        Index = WSAWaitForMultipleEvents(1,&EventArray[i],TRUE,1000,
        FALSE);
        if ((Index == WSA_WAIT_FAILED)||(Index == WSA_WAIT_TIMEOUT))
        continue;
        else
        {
            Index = i;
            WSAEnumNetworkEvents(
            SocketArray[Index],
            EventArray[Index],
            &NetworkEvents);

            //检查 FD_ACCEPT 消息
            if (NetworkEvents.lNetworkEvents &FD_ACCEPT)
            {
                if (NetworkEvents.iErrorCode[FD_ACCEPT_BIT]!= 0)
                {
                    printf("FD_ACCEPT failed with error %d \n",
                    NetworkEvents.iErrorCode[FD_ACCEPT_BIT]);
                    break;
                }

                //接受一个新连接，并将它添加到套接字及事件列表中
                Accept = accept(
                SocketArray[Index],
                NULL,NULL);

                //无法处理多于 WSA_MAXIMUM_WAIT_EVENTS 数量的套接字，故关闭接受套接字
                if (EventTotal >WSA_MAXIMUM_WAIT_EVENTS)
                {
```

```
        printf("Too manyconnections");
        closesocket(Accept);
        break;
    }

    NewEvent = WSACreateEvent();

    WSAEventSelect(Accept,NewEvent,
        FD_READ | FD_WRITE | FD_CLOSE);

    EventArray[EventTotal]= NewEvent;
    SocketArray[EventTotal]= Accept;
    EventTotal++;
        printf("Socket %d connected \n",Accept);
    }
    //处理 FD_READ 通知
    if (NetworkEvents.lNetworkEvents &FD_READ)
    {
        if (NetworkEvents.iErrorCode[FD_READ_BIT]!= 0)
        {
            printf("FD_READ failed with error %d \n",
            NetworkEvents.iErrorCode[FD_READ_BIT]);
            break;
        }

        //从套接字读入数据
        recv(SocketArray[Index -WSA_WAIT_EVENT_0],
        buffer,sizeof(buffer),0);
    }

    //处理 FD_WRITE 通知
    if (NetworkEvents.lNetworkEvents &FD_WRITE)
    {
        if (NetworkEvents.iErrorCode[FD_WRITE_BIT]!= 0)
        {
            printf("FD_WRITE failed with error %d \n",
            NetworkEvents.iErrorCode[FD_WRITE_BIT]);
            break;
        }
        send(SocketArray[Index -WSA_WAIT_EVENT_0],
        buffer,sizeof(buffer),0);
    }

    if (NetworkEvents.lNetworkEvents &FD_CLOSE)
    {
```

```
                    if (NetworkEvents.iErrorCode[FD_CLOSE_BIT]!= 0)
                    {
                        printf("FD_CLOSE failed with error %d \n",
                        NetworkEvents.iErrorCode[FD_CLOSE_BIT]);
                        break;
                    }
                    closesocket(SocketArray[Index]);
                    //从 Socket 和 Event 数组删除套接字及其关联的事件，并递减 EventTotal
                    CompressArrays(EventArray,SocketArray,&EventTotal);
                }
            }
        }
    }
```

WSAEventSelect 模型有几个方面的优势。它概念简单，不需要窗口环境。惟一的缺点是，它每次只等待 64 个事件，这一限制使得在处理多个套接字时，有必要组织一个线程池。同时，因为需要许多线程去处理大量套接字连接，这个模型的伸缩性不如后面讨论的重叠模型。

5.2.5 重叠模型

在 Winsock 中，相比迄今为止解释过的其他所有 I/O 模型，重叠 I/O(Overlapped I/O)模型使应用程序能达到更佳的系统性能。重叠模型的基本设计原理便是让应用程序使用重叠的数据结构，一次投递一个或多个 Winsock I/O 请求。针对那些提交的请求，在它们完成之后，应用程序可为它们提供服务。该模型适用于除 Windows CE 之外的各种 Windows 平台。模型的总体设计以 Windows 重叠 I/O 机制为基础。这种机制可通过 ReadFile 和 WriteFile 两个函数，在设备上执行 I/O 操作。

最开始的时候，Winsock 重叠 I/O 模型只能应用于 Windows NT 操作系统上运行的 Winsock 1.1 应用程序。应用程序可以针对套接字句柄调用 ReadFile 以及 WriteFile，同时指定重叠结构(稍后详述)，从而利用这个模型。自 Winsock 2 发布开始，重叠 I/O 便已集成到新的 Winsock 函数中，例如 WSASend 和 WSARecv。这样，重叠 I/O 模型便能适用于支持 Winsock 2 的所有 Windows 平台。

注意 在 Winsock 2 发布之后，重叠 I/O 仍可在 Windows NT 和 Windows 2000 这两个操作系统中，随 ReadFile 和 WriteFile 这两个函数使用。但是，Windows 95、Windows 98 和 Windows Me 均不具备这一功能。考虑到应用程序的跨平台兼容问题，应尽量避免使用 Windows 的 ReadFile 和 WriteFile 函数，分别换以 WSARecv 和 WSASend 函数。本节只打算讲述通过新的 Winsock 2 函数，来使用重叠 I/O 模型。

要想在一个套接字上使用重叠 I/O 模型，首先必须创建一个设置了重叠标志的套接字。有关重叠套接字的更多内容请参见第 2 章。

成功建好一个套接字，同时将它与一个本地接口绑定到一起后，便可开始进行重叠 I/O 的操作，

方法是调用下列的 Winsock 函数，同时指定一个可选的 WSAOVERLAPPED 结构：

- WSASend
- WSASendTo
- WSARecv
- WSARecvFrom
- WSAIoctl
- WSARecvMsg
- AcceptEx
- ConnectEx
- TransmitFile
- TransmitPackets
- DisconnectEx
- WSANSPIoctl

为了使用重叠 I/O，每个函数都把 WSAOVERLAPPED 结构作为参数。若用一个 WSAOVERLAPPED 结构一起调用这些函数，函数会立即完成并返回，而不管套接字是否设为阻塞模式(本章开头已详细讲过了)。这些函数依赖于 WSAOVERLAPPED 结构来管理一个 I/O 请求的完成。主要有两个方法可用来管理重叠 I/O 请求的完成情况：应用程序可等待**事件对象通知** (event object notification)，也可通过**完成例程** (completion routines)，对已经完成的请求加以处理。上面列出的函数(AcceptEx 除外)还有另一个常用的参数：WSAOVERLAPPED_COMPLETION_ROUTINE。该参数指定的是一个可选的指针，指向一个完成例程函数，该函数在重叠请求完成后被调用。接下来将探讨事件通知方法。本章稍后还会介绍如何使用可选的完成例程，来代替事件对完成的重叠请求进行处理。

5.2.5.1　事件通知

重叠 I/O 的事件通知方法要求将 Windows 事件对象与 WSAOVERLAPPED 结构关联在一起。若使用一个 WSAOVERLAPPED 结构，发出像 WSASend 和 WSARecv 这样的 I/O 调用，它们会立即返回。

通常，大家会发现这些 I/O 调用会以失败而告终，并返回 SOCKET_ERROR。使用 WSAGetLastError 函数，便可获得与 WSA_IO_PENDING 错误状态有关的一个报告。这个错误状态意味着 I/O 操作正在进行。稍后的某个时间，应用程序需要等候与 WSAOVERLAPPED 结构相关联的事件对象，判断某个重叠 I/O 请求何时完成。WSAOVERLAPPED 结构为重叠 I/O 请求的初始化及其后续的完成之间提供了一种通信媒介。下面是这个结构的定义：

```
typedef struct WSAOVERLAPPED
{
    DWORD Internal;
    DWORD InternalHigh;
```

```
    DWORD Offset;
    DWORD OffsetHigh;
    WSAEVENT hEvent;
}WSAOVERLAPPED,FAR * LPWSAOVERLAPPED;
```

其中，Internal、InternalHigh、Offset 和 OffsetHigh 字段均由系统在内部使用，不能由应用程序直接进行处理或使用。而另一方面，hEvent 字段则允许应用程序将事件对象的句柄同操作关联起来。

一个重叠 I/O 请求最终完成后，应用程序要负责获取重叠 I/O 操作的结果。一个重叠请求操作最终完成之后，在事件通知方法中，Winsock 会更改与 WSAOVERLAPPED 结构关联的事件对象的事件传信状态，将其从未传信变成已传信。由于已经有一个事件对象分配给 WSAOVERLAPPED 结构，所以只需简单地调用 WSAWaitForMultipleEvents 函数，便可判断出重叠 I/O 调用将在什么时候完成。该函数已在前面讲述 WSAEventSelect I/O 模型时介绍过了。WSAWaitForMultipleEvents 函数会等候一段指定的时间，等待一个或多个事件对象进入已传信状态。在此再次提醒大家注意：WSAWaitForMultipleEvents 函数一次最多只能等待 64 个事件对象。确认某个重叠请求完成之后，接着需要调用 WSAGetOverlappedResult 函数，判断这个重叠调用到底是成功，还是失败。该函数的定义如下：

```
BOOL WSAGetOverlappedResult(
    SOCKET s,
    LPWSAOVERLAPPED lpOverlapped,
    LPDWORD lpcbTransfer,
    BOOL fWait,
    LPDWORD lpdwFlags
);
```

其中，s 参数用于标识在重叠操作开始的时候被指定的那个套接字。lpOverlapped 参数是一个指针，对应于在重叠操作开始时，被指定的那个 WSAOVERLAPPED 结构。lpcbTransfer 参数也是一个指针，对应一个 DWORD 变量，该变量负责接收一次重叠发送或接收操作实际传输的字节数。fWait 参数用于决定函数是否应该等待挂起的重叠操作完成。若将 fWait 设为 TRUE，那么除非操作完成，否则函数不会返回；若设为 FALSE，而且操作仍然处于挂起状态，那么 WSAGetOverlappedResult 函数会返回 FALSE 值，同时返回一个 WSA_IO_INCOMPLETE 错误。但就目前的情况来说，由于需要在一个已传信事件上等候重叠操作完成，所以该参数无论采用什么设置，都没有任何效果。最后一个参数是 lpdwFlags 是一个指针，指向一个 DWORD，负责接收结果标志(假如原先的重叠调用是用 WSARecv 或 WSARecvFrom 函数发出的)。

如 WSAGetOverlappedResult 函数调用成功，返回值就是 TRUE。这意味着重叠 I/O 操作已成功完成，而且 lpcbTransfer 参数所指向的值已进行了更新。若返回值是 FALSE，那么可能是由下述任何一种原因造成的：

- 重叠 I/O 操作仍处在挂起状态
- 重叠操作已经完成，但含有错误

- 因为在提供给 WSAGetOverlappedResult 函数的一个或多个参数中，存在着错误，所以无法判断重叠操作的完成状态

失败后，lpcbTransfer 参数指向的值不会被更新，而且应用程序应调用 WSAGetLastError 函数，查看到底是何种原因造成了调用失败。

下面的示例代码阐述了如何编制一个简单的服务器应用程序，令其在一个套接字上对重叠 I/O 操作进行管理，该程序利用了前述的事件通知机制。

```
#define DATA_BUFSIZE 4096

void main(void)
{
    WSABUF DataBuf;
    char buffer[DATA_BUFSIZE];
    DWORD EventTotal = 0,
    RecvBytes= 0,
    Flags= 0;
    WSAEVENT EventArray[WSA_MAXIMUM_WAIT_EVENTS];
    WSAOVERLAPPED AcceptOverlapped;
    SOCKET ListenSocket,AcceptSocket;

    //第 1 步:
    //启动 Winsock，建立监听套接字
    ...

    //第 2 步:
    //接受一个入站连接
    AcceptSocket = accept(ListenSocket,NULL,NULL);

    //第 3 步:
    //建立一个重叠结构

    EventArray[EventTotal]= WSACreateEvent();

    ZeroMemory(&AcceptOverlapped,
    sizeof(WSAOVERLAPPED));
    AcceptOverlapped.hEvent = EventArray[EventTotal];

    DataBuf.len = DATA_BUFSIZE;
    DataBuf.buf = buffer;

    EventTotal++;

    //第 4 步:
```

```
//投递一个 WSARecv 请求，以便开始在套接字上接收数据

if (WSARecv(AcceptSocket,&DataBuf,1,&RecvBytes,
&Flags,&AcceptOverlapped,NULL)== SOCKET_ERROR)
{
    if (WSAGetLastError()!= WSA_IO_PENDING)
    {
        //出错
    }
}

//处理套接字上的重叠接收
while(TRUE)
{
    DWORD Index;
    //第 5 步:
    //等候重叠 I/O 调用结束
    Index = WSAWaitForMultipleEvents(EventTotal,
    EventArray,FALSE,WSA_INFINITE,FALSE);

    //索引应为 0，因为 EventArray 中仅有一个事件

    //第 6 步:
    //重置已传信事件
    WSAResetEvent(
        EventArray[Index -WSA_WAIT_EVENT_0]);
    //第 7 步:
    //确定重叠请求的状态
    WSAGetOverlappedResult(AcceptSocket,
        &AcceptOverlapped,&BytesTransferred,
        FALSE,&Flags);

    //先检查看通信对方是否已关闭连接，如果已关闭，则关闭套接字
    if (BytesTransferred == 0)
    {
        printf("Closing socket %d \n",AcceptSocket);
        closesocket(AcceptSocket);
        WSACloseEvent(
            EventArray[Index -WSA_WAIT_EVENT_0]);
        return;
    }

    //对接收到的数据进行某种处理
    //DataBuf 包含接收到的数据
```

```
...

//第 8 步:
//在套接字上投递另一个 WSARecv()请求

Flags = 0;
ZeroMemory(&AcceptOverlapped,
    sizeof(WSAOVERLAPPED));

AcceptOverlapped.hEvent = EventArray[Index -
    WSA_WAIT_EVENT_0];
DataBuf.len = DATA_BUFSIZE;
DataBuf.buf = buffer;
        if (WSARecv(AcceptSocket,&DataBuf,1,
            &RecvBytes,&Flags,&AcceptOverlapped,
            NULL)== SOCKET_ERROR)
        {
            if (WSAGetLastError()!= WSA_IO_PENDING)
            {
                //未预料到的错误
            }
        }
    }
}
```

对该程序采用的编程步骤总结如下:

1. 创建一个套接字，开始在指定的端口上监听连接请求。
2. 接受一个入站的连接请求。
3. 为接受的套接字新建一个 WSAOVERLAPPED 结构，并为该结构分配一个事件对象句柄。也将该事件对象句柄分配给一个事件数组，以便稍后由 WSAWaitForMultipleEvents 函数使用。
4. 将 WSAOVERLAPPED 结构指定为参数，在套接字上投递一个异步 WSARecv 请求。
5. 使用步骤 3 的事件数组，调用 WSAWaitForMultipleEvents 函数，并等待与重叠调用关联在一起的事件进入已传信状态。
6. 使用 WSAGetOverlappedResult 函数，判断重叠调用的返回状态是什么。
7. 函数完成后，针对事件数组，调用 WSAResetEvent 函数，从而重设事件对象，并对完成的重叠请求进行处理。
8. 在套接字上投递另一个重叠 WSARecv 请求。
9. 重复步骤 5～8。

这个例子极易扩展，从而提供对多个套接字的支持。方法是将代码的重叠 I/O 处理部分移至一个

独立的线程中，让主应用程序线程为额外的连接请求提供服务。

注意 如果用重叠方式(在 WSAOVERLAPPED 结构内部指定一个事件，或利用一个完成例程)调用 Winsock 函数，则调用操作有可能立即完成。例如，在数据已经被接收到并放入缓冲区之后调用 WSARecv，就会导致 WSARecv 返回 NO_ERROR。如果任何重叠函数失败并返回 WSA_IO_PENDING，或立刻成功，完成事件将总会被传信，而完成例程也将随时运行(如果已做了指定)。对于带完成端口的重叠 I/O 而言，这意味着完成通知将被传递到完成端口，使其开始服务。

5.2.5.2 完成例程

完成例程是应用程序用来管理完成的重叠 I/O 请求的另一种方法。完成例程其实就是一些函数，我们将这些函数传递给重叠 I/O 请求，以供重叠 I/O 请求完成时由系统调用。它们的基本设计宗旨，是通过调用者的线程，为已完成的 I/O 请求提供服务。除此以外，应用程序可通过完成例程，继续进行重叠 I/O 的处理。

如果希望用完成例程为重叠 I/O 请求提供服务，应用程序必须为一个绑定 I/O 的 Winsock 函数指定一个完成例程，同时指定一个 WSAOVERLAPPED 结构。一个完成例程必须拥有下述函数原型：

```
void CALLBACK CompletionROUTINE(
    DWORD dwError,
    DWORD cbTransferred,
    LPWSAOVERLAPPED lpOverlapped,
    DWORD dwFlags
);
```

用完成例程完成一个重叠 I/O 请求之后，参数中会包含下述信息：

- dwError 参数表明一个重叠操作(由 lpOverlapped 指定)的完成状态是什么
- cbTransferred 参数指明了在重叠操作期间，实际传输的字节量是多大
- lpOverlapped 参数指明传递到最初的 I/O 调用内的一个 WSAOVERLAPPED 结构
- dwFlags 参数返回操作结束时可能用的标志(如从 WSARecv 结束)

在用一个完成例程提交的重叠请求与用一个事件对象提交的重叠请求之间，存在着一个非常重要的区别。WSAOVERLAPPED 结构的事件字段 hEvent 并未被使用；也就是说，不可以将一个事件对象同重叠请求关联到一起。用完成例程发出重叠 I/O 调用之后，调用线程一旦完成，最终必须为完成例程提供服务。这样，便要求我们将调用线程置于一种**警觉的等待状态** (alertable wait state)。并在 I/O 操作完成后，对完成例程加以处理。WSAWaitForMultipleEvents 函数可用来将线程置于一种警觉的等待状态。这样做的缺点在于，我们还必须有一个事件对象可用于 WSAWaitForMultipleEvents 函数。假定应用程序用完成例程只对重叠请求进行处理，便不大可能有什么事件对象需要处理。作为一种变通方法，应用程序可用 Windows 的 SleepEx 函数将线程置为一种警觉的等待状态。当然，亦可创建一

个伪事件对象，它不与任何东西关联在一起。假如调用线程总是处于繁忙状态，而不是处于一种警觉的等待状态，那么根本不会有被投递的完成例程会得到调用。

如前所见，WSAWaitForMultipleEvents 通常会等待同 WSAOVERLAPPED 结构关联在一起的事件对象。该函数也用来将线程置入一种警觉的等待状态，并可为已经完成的重叠 I/O 请求进行完成例程的处理(前提是将 fAlertable 参数设为 TRUE)。用完成例程结束重叠 I/O 请求之后，返回值是 WSA_IO_COMPLETION，而不是事件数组中的一个事件对象的索引。SleepEx 函数的行为实际上和 WSAWaitForMultipleEvents 差不多，只是它不需要任何事件对象。对 SleepEx 函数的定义如下：

```
DWORD SleepEx(
    DWORD dwMilliseconds,
    BOOL bAlertable
);
```

其中，dwMilliseconds 参数定义了 SleepEx 函数的等待时间，以 ms(毫秒)为单位。如果将 dwMilliseconds 设为 INFINITE，那么 SleepEx 会无休止地等待下去。bAlertable 参数指定了完成例程的执行方式。假如将 bAlertable 设为 FALSE，而且进行了一次 I/O 完成回叫，那么 I/O 完成函数就不会执行，而且该函数也不会返回，除非超过由 dwMilliseconds 规定的时间。若设为 TRUE，那么完成例程会得到执行，同时 SleepEx 函数返回 WAIT_IO_COMPLETION。

下面的代码展示了如何构建一个简单的服务器应用程序，令其采用前述的方法，通过完成例程来实现对一个套接字请求的管理。

```
#define DATA_BUFSIZE 4096

SOCKET AcceptSocket,
       ListenSocket;
WSABUF DataBuf;
WSAEVENT EventArray[MAXIMUM_WAIT_OBJECTS];
DWORD Flags,
    RecvBytes,
    Index;
char buffer[DATA_BUFSIZE];

void main(void)
{
    WSAOVERLAPPED Overlapped;
    //第 1 步:
    //启动 Winsock，建立监听套接字
    ...

    //第 2 步:
    //接受一个新连接
    AcceptSocket = accept(ListenSocket,NULL,NULL);
```

```
//第 3 步:
//已经有了一个接受套接字之后，开始使用带有完成例程的重叠 I/O 来处理 I/O
//为了启动重叠 I/O 处理，先提交一个重叠 WSARecv()请求

Flags = 0;

ZeroMemory(&Overlapped,sizeof(WSAOVERLAPPED));

DataBuf.len = DATA_BUFSIZE;
DataBuf.buf = buffer;

//第 4 步:
//将 WSAOVERLAPPED 结构指定为一个参数，在套接字上投递一个异步 WSARecv()请求并提供下面的
//作为完成例程的 WorkerRoutine 函数

if (WSARecv(AcceptSocket,&DataBuf,1,&RecvBytes,
&Flags,&Overlapped,WorkerRoutine)
== SOCKET_ERROR)
{
    if (WSAGetLastError()!= WSA_IO_PENDING)
    {
        printf("WSARecv()failed with error %d \n",
        WSAGetLastError());
        return;
    }
}

//因为 WSAWaitForMultipleEvents() API 要求在一个或多个事件对象上等待，因此不得不
//创建一个伪事件对象。作为一种可选方案，也可以使用 SleepEx()作为替代。

EventArray[0]= WSACreateEvent();

while(TRUE)
{
    //第 5 步:
    Index = WSAWaitForMultipleEvents(1,EventArray,
    FALSE,WSA_INFINITE,TRUE);
    //第 6 步:
    if (Index == WAIT_IO_COMPLETION)
    {
        //一个重叠请求完成例程结束。继续为更多的完成例程服务
        continue;
    }
    else
```

```
        {
            //发生一个严重错误：停止处理
            //如果正在处理一个事件对象，那么这也就可能是事件数组的一个索引
            return;
        }
    }
}
void CALLBACK WorkerRoutine(DWORD Error,
                            DWORD BytesTransferred,
                            LPWSAOVERLAPPED Overlapped,
                            DWORD InFlags)
{
    DWORD SendBytes,RecvBytes;
    DWORD Flags;

    if (Error != 0 || BytesTransferred == 0)
    {
        //要么是套接字上发生了一个严重错误，要么套接字已被通信对方关闭
        closesocket(AcceptSocket);
        return;
    }

    //此刻，一个重叠 WSARecv()请求顺利完成
    //现在可以接受 DataBuf 变量中包含的已收到的数据了
    //处理完收到的数据后，需要投递另外一个重叠的 WSARecv()或 WSASend()请求
    //为简便起见，这里将投递另外一个 WSARecv()请求

    Flags = 0;

    ZeroMemory(&Overlapped,sizeof(WSAOVERLAPPED));

    DataBuf.len = DATA_BUFSIZE;
    DataBuf.buf = buffer;

    if (WSARecv(AcceptSocket,&DataBuf,1,&RecvBytes,
    &Flags,&Overlapped,WorkerRoutine)
    == SOCKET_ERROR)
    {
        if (WSAGetLastError()!= WSA_IO_PENDING )
        {
            printf("WSARecv()failed with error %d \n",
            WSAGetLastError());
            return;
        }
    }
```

}

该程序主要按下述编程步骤进行：

1. 新建一个套接字，开始在指定端口上监听传入的连接。
2. 接受一个入站的连接请求。
3. 为接受的套接字创建一个 WSAOVERLAPPED 结构。
4. 在套接字上投递一个异步 WSARecv 请求，需要将 WSAOVERLAPPED 指定为参数，同时提供一个完成例程。
5. 在将 fAlertable 参数设为 TRUE 的前提下，调用 WSAWaitForMultipleEvents，并等待一个重叠 I/O 请求完成。重叠请求完成后，完成例程会自动执行，而且 WSAWaitForMultipleEvents 会返回一个 WSA_IO_COMPLETION。在完成例程内，可用一个完成例程投递另一个重叠 WSARecv 请求。
6. 检查 WSAWaitForMultipleEvents 是否返回 WSA_IO_COMPLETION。
7. 重复步骤 5 和 6。

重叠模型提供高性能套接字 I/O。因为使用这种模型的应用程序通知缓冲区收发系统直接使用的数据，所以它和前面的几种不同。也就是说，如果应用程序投递了一个 10KB 大小的缓冲区来接收数据，且数据已到达套接字，则该数据将直接被拷贝到投递的缓冲区。在前面讲述的模型中，数据到达并被拷贝到单套接字接收缓冲区中，此时应用程序会被告知可以读入的容量。当应用程序调用接收函数之后，数据将从单套接字缓冲区拷贝到应用程序的缓冲区。第 6 章将讨论开发高性能、可扩展的 Winsock 应用程序的策略，还将详细讨论 WSARecvMsg、AcceptEx、ConnectEx、TransmitFile、TransmitPackets 及 DisconnectEx 等 API 函数。

在事件中使用重叠 I/O 的缺点，也是每次最多只能等待 64 个事件这一局限性。完成例程是一个不错的替代方式，但必须注意确保投递完成操作的线程进入警觉的等待状态，以便使完成例程能够圆满结束。同时，还要注意确保完成例程不要作过量的运算，以便在很重的负载之下，这些完成过程能够尽快开始运行。

5.2.6 完成端口模型

因为需要做出大量的工作以便将套接字添加到一个完成端口，而其他 I/O 方法的初始化步骤则省事多了，所以对新手来说，完成端口模型好像过于复杂了。然而，一旦弄明白是怎么回事，就会发现步骤其实并非那么复杂。同时，假若一个应用程序同时需要管理为数众多的套接字，那么采用这种模型，往往可以达到最佳的系统性能。但不幸的是，该模型只适用于 Windows NT、Windows 2000 及 Windows XP 操作系统。然而，和前面已介绍过的模型相比，完成端口模型提供了最好的伸缩性。这个模型非常适合用来处理数百乃至上千个套接字。

从本质上说，完成端口模型要求创建一个 Windows 完成端口对象，该对象通过指定数量的线程，

对重叠 I/O 请求进行管理，以便为已经完成的重叠 I/O 请求提供服务。要注意的是，所谓完成端口，实际是 Windows 采用的一种 I/O 构造机制，除套接字句柄之外，还可接受其他东西。然而，本节只打算讲述如何使用套接字句柄，来发挥完成端口模型的作用。使用这种模型之前，首先要创建一个 I/O 完成端口对象，用它面向任意数量的套接字句柄，管理多个 I/O 请求。要做到这一点，需要调用 CreateCompletionPort 函数。该函数的定义如下：

```
HANDLE CreateIoCompletionPort(
    HANDLE FileHandle,
    HANDLE ExistingCompletionPort,
    DWORD   CompletionKey,
    DWORD   NumberOfConcurrentThreads
);
```

在深入探讨其中的各个参数之前，首先要注意该函数实际用于两个截然不同的目的：

- 用于创建一个完成端口对象
- 将一个句柄同完成端口关联到一起

最开始创建一个完成端口时，我们惟一感兴趣的参数便是 NumberOfConcurrentThreads；前 3 个参数都不大重要。NumberOfConcurrentThreads 参数的特殊之处在于，它定义了在一个完成端口上，同时允许执行的线程数量。在理想情况下，我们希望每个处理器各自负责一个线程的运行，为完成端口提供服务，避免过于频繁的线程任务转换。若将该参数设为 0 则告诉系统，安装了多少个处理器，则允许同时运行多少个线程。可用下述代码创建一个 I/O 完成端口：

```
CompletionPort = CreatcIoCompletionPort(INVALID_HANDLE_VALUE,
    NULL,0,0);
```

该语句的作用是返回一个句柄，在为完成端口分配了一个套接字句柄后，用来对那个端口进行标识。

5.2.6.1　工作器线程与完成端口

成功创建一个完成端口后，便可开始将套接字句柄与对象关联到一起。但在关联套接字之前，首先必须创建一个或多个工作器线程，以便在套接字的 I/O 请求投递给完成端口对象后，为完成端口提供服务。在这个时候，或许大家会觉得奇怪，到底应创建多少个线程，以便为完成端口提供服务呢？因为服务 I/O 请求所需的线程数量取决于应用程序的总体设计情况，所以这实际正是完成端口模型显得颇为复杂的一个方面。在此要记住的一个重点在于，在调用 CreateIoComletionPort 时指定的并发线程数量，与打算创建的工作器线程数量相比，它们代表的并非同一件事情。以前，我们曾建议大家用 CreateIoCompletionPort 函数为每个处理器都指定一个线程(处理器的数量有多少，便指定多少线程)以避免频繁的线程上下文转换。CreateIoCompletionPort 函数的 NumberOfConcurrentThreads 参数明确地指示系统：在一个完成端口上，一次只允许 n 个工作器线程运行假如在完成端口上创建的工作器线程数量超出 n 个，那么在同一时刻，最多只允许 n 个线程运行(但实际上，在一段较短的时间内，系统

有可能超过这个值，但很快便会把它减少至事先在 CreateIoCompletionPort 函数中设定的值)。那么，为何实际创建的工作器线程数量有时要比 CreateIoCompletionPort 函数设定的多一些呢？这样做有必要吗？如先前所述，这主要取决于应用程序的总体设计情况。假定某个工作器线程调用了一个函数，例如 Sleep 或 WaitForSingleObject，但却进入了暂停状态，就允许另一个线程代替它的位置。换言之，我们希望随时都能执行尽可能多的线程；当然，最大的线程数量是事先在 CreateIoCompletonPort 调用里设定好的。这样，假如事先预计到线程有可能暂时处于阻塞状态，那么最好能够创建比 CreateIoCompletonPort 的 NumberOfConcurrentThreads 参数值更多的线程，以便到时候充分发挥系统的潜力。

一旦在完成端口上拥有足够多的工作器线程来为 I/O 请求提供服务，便可着手将套接字句柄同完成端口关联到一起。这需要在一个现有的完成端口上，调用 CreateIoCompletionPort 函数，同时为前 3 个参数——FileHandle、ExistingCompletionPort 和 CompletionKey——提供套接字的信息。其中，FileHandle 参数指定一个要同完成端口关联在一起的套接字句柄。ExistingCompletionPort 参数标识的是一个现有的完成端口套接字句柄已经与它关联在了一起。CompletionKey(完成键)参数则标识的是要与某个特定套接字句柄关联在一起的**单句柄数据** (per-handle data)；在这个参数中，应用程序可保存与一个套接字对应的任意类型的信息。之所以把它叫作单句柄数据，是由于它代表了与套接字句柄关联在一起的数据。可将它作为指向一个数据结构的指针；在这个结构中，同时包含了套接字的句柄，以及与该套接字有关的其他信息。正如本章稍后还会讲述的那样，为完成端口提供服务的线程的例程可通过这个参数，取得与套接字句柄有关的信息。

根据到目前为止已经讲到的东西，首先来构建一个基本的应用程序框架。下面的示例阐述了如何使用完成端口模型，来开发一个回应服务器应用程序。这个程序基本上按下述步骤进行：

1. 创建一个完成端口。第 4 个参数保持为 0，它指定在完成端口上每个处理器一次只允许执行一个工作器线程。
2. 判断系统内到底多少个处理器。
3. 创建工作器线程，根据步骤 2 得到的处理器信息，在完成端口上为已完成的 I/O 请求提供服务。在这个简单的例子中，我们为每个处理器都只创建一个工作器线程。这是由于事先已预计到，到时不会有任何线程进入暂停状态，造成由于线程数量的不足而使处理器空闲的局面(没有足够的线程可供执行)。调用 CreateThread 函数时，必须同时提供一个工作器例程，由线程在创建好后执行。本节稍后还会详细讨论线程的职责。
4. 准备好一个监听套接字，在端口 5150 上监听传入的连接请求。
5. 使用 accept 函数，接受入站的连接请求。
6. 创建一个数据结构，用于容纳单句柄数据，同时在结构中存入接受的套接字句柄。
7. 调用 CreateIoCompletionPort，将自 accept 返回的新套接字句柄同完成端口关联到一起。通过 CompletionKey 参数，将单句柄数据结构传递给 CreateIoCompletionPort。
8. 开始在已接受的连接上进行 I/O 操作。在此，我们希望通过重叠 I/O 机制，在新建的套接字

上投递一个或多个异步 WSARecv 或 WSASend 请求。这些 I/O 请求完成后，工作器线程会为 I/O 请求提供服务，同时继续处理以后的 I/O 请求。在步骤 3 指定的工作器例程中，我们将看到这一点。

9. 重复步骤 5~8，直至服务器终止。

```
HANDLE CompletionPort;
WSADATA wsd;
SYSTEM_INFO SystemInfo;
SOCKADDR_IN InternetAddr;
SOCKET Listen;
int i;
typedef struct _PER_HANDLE_DATA
{
    SOCKET Socket;
    SOCKADDR_STORAGE ClientAddr;
    //将和这个句柄关联的其他有用信息
}PER_HANDLE_DATA,* LPPER_HANDLE_DATA;

//加载 Winsock
StartWinsock(MAKEWORD(2,2),&wsd);

//第 1 步:
//创建一个 I/O 完成端口
CompletionPort = CreateIoCompletionPort(
INVALID_HANDLE_VALUE,NULL,0,0);

//第 2 步:
//确定系统中有多少个处理器
GetSystemInfo(&SystemInfo);

//第 3 步:
//基于系统中可用的处理器数量创建工作器线程
//对这个简单例子而言，我们为每个处理器都创建一个工作器线程

for(i = 0; i <SystemInfo.dwNumberOfProcessors; i++)
{
    HANDLE ThreadHandle;

    //创建一个服务器的工作器线程，并将完成端口传递到该线程
    //注意: ServerWorkerThread 进程并不是在这个列表中定义的
    ThreadHandle = CreateThread(NULL,0,
        ServerWorkerThread,CompletionPort,
        0,NULL;
    //关闭线程句柄
    CloseHandle(ThreadHandle);
```

```
}

//第 4 步:
//创建一个监听套接字

Listen = WSASocket(AF_INET,SOCK_STREAM,0,NULL,0,
    WSA_FLAG_OVERLAPPED);

InternetAddr.sin_family= AF_INET;
InternetAddr.sin_addr.s_addr = htonl(INADDR_ANY);
InternetAddr.sin_port = htons(5150);

bind(Listen,(PSOCKADDR)&InternetAddr,
    sizeof(InternetAddr));

//让套接字为监听做好准备
listen(Listen,5);

while(TRUE)
{
    PER_HANDLE_DATA * PerHandleData= NULL;
    SOCKADDR_IN saRemote;
    SOCKET Accept;
    int RemoteLen;

    //第 5 步:
    //接受连接,并分配到完成端口
    RemoteLen = sizeof(saRemote);
    Accept = WSAAccept(Listen,(SOCKADDR * )&saRemote,
    &RemoteLen);

    //第 6 步:
    //创建用来和套接字关联的单句柄数据信息结构
    PerHandleData = (LPPER_HANDLE_DATA)
        GlobalAlloc(GPTR,sizeof(PER_HANDLE_DATA));
    printf("Socket number %d connected \n",Accept);
    PerHandleData->Socket = Accept;
    memcpy(&PerHandleData->ClientAddr,&saRemote,RemoteLen);

    //第 7 步:
    //将接受套接字和完成端口关联起来
    CreateIoCompletionPort((HANDLE)Accept,
        CompletionPort,(DWORD)PerHandleData,0);

    //第 8 步:
```

```
    //开始在接受套接字上处理 I/O
    //使用重叠 I/O，在套接字上投递一个或多个 WSASend()或 WSARecv()调用
    WSARecv(...);
}
DWORD WINAPI ServerWorkerThread(LPVOID lpParam)
{
    //工作器线程的必备条件稍后讨论。
    return 0;
}
```

5.2.6.2　完成端口和重叠 I/O

将套接字句柄与一个完成端口关联在一起后，便能以套接字句柄为基础，投递重叠发送与接收请求，开始对 I/O 请求进行处理。之后，可开始依赖完成端口，接收有关 I/O 操作完成情况的通知。从本质上说，完成端口模型利用了 Windows 重叠 I/O 机制。在这种机制中，类似 WSASend 和 WSARecv 这样的 Winsock API 调用会立即返回。此时，需要由应用程序负责在以后的某个时间，通过 OVERLAPPED 结构来检索调用的结果。在完成端口模型中，要想做到这一点，需要使用 GetQueuedCompletionStatus 函数，让一个或多个工作器线程在完成端口上等待。该函数的定义如下：

```
BOOL GetQueuedCompletionStatus(
    HANDLE CompletionPort,
    LPDWORD lpNumberOfBytesTransferred,
    PULONG_PTR lpCompletionKey,
    LPOVERLAPPED * lpOverlapped,
    DWORD dwMilliseconds
);
```

其中，CompletionPort 参数对应于线程所在的完成端口。lpNumberOfBytesTransferred 参数负责在完成了一次 I/O 操作(如 WSASend 或 WSARecv)后，接收实际传输的字节数。lpCompletionKey 参数为原先传递到 CreateCompletionPort 函数的套接字返回单句柄数据。如前所述，大家最好将套接字句柄保存在这个键 (Key)中。lpOverlapped 参数用于接收已完成的 I/O 操作的 WSAOVERLAPPED 结构。因为可用它获取每个 I/O 操作的数据，所以这实际是一个相当重要的参数。而最后一个参数 dwMilliseconds，用于指明调用者等待一个完成数据包在完成端口上出现时，希望等候的毫秒数。假如将其设为 INFINITE，调用会无休止地等待下去。

5.2.6.3　单句柄数据和单 I/O 操作数据

当一个工作器线程从 GetQueuedCompletionStatus 这个 API 调用中接收到 I/O 完成通知后，在 lpCompletionKey 和 lpOverlapped 参数中，会包含一些必要的套接字信息。利用这些信息，可通过完成端口，继续在一个套接字上进行 I/O 处理。通过这些参数，可获得两种重要的套接字数据类型：单句柄数据以及单 I/O 操作数据。

因为在一个套接字首次与完成端口关联到一起的时候，单句柄数据便与一个特定的套接字句柄对

应起来了，所以 lpCompletionKey 参数也就包含了单句柄数据。这些数据正是在进行 CreateIoCompletionPortAPI 调用的时候，通过 CompletionKey 参数传递的。如早先所述，应用程序可通过该参数传递任意类型的数据。通常情况下，应用程序会将与 I/O 请求有关的套接字句柄保存在这里。

lpOverlapped 参数则包含了一个 OVERLAPPED 结构，在它后面跟随单 I/O 操作数据。工作器线程处理一个完成数据包时(回应数据、接受连接以及投递另一个线程等)，这些信息是它必须要知道的。单 I/O 操作数据是包含在一个结构内的、任意数量的字节，这个结构本身也包含了一个 OVERLAPPED 结构假如一个函数要求用到一个 OVERLAPPED 结构，我们便必须将这样的一个结构传递进去，以满足它的要求。要想做到这一点，一个简单的方法是定义一个结构，然后将 OVERLAPPED 结构作为新结构的第 1 个元素使用。举个例子来说，可定义下述数据结构，实现对单 I/O 操作数据的管理：

```
typedef struct
{
    OVERLAPPED     Overlapped;
    char           Buffer[DATA_BUFSIZE];
    int            BufferLen;
    int            OperationType;
}PER_IO_DATA;
```

该结构演示了通常要与 I/O 操作关联在一起的某些重要数据元素，例如刚才完成的那个 I/O 操作(发送或接收请求) 的类型。在这个结构中，我们认为提供给已完成的 I/O 操作的数据缓冲区是非常有用的。要想调用一个 Winsock API 函数，同时为其分配一个 OVERLAPPED 结构，只要简单地撤消对结构中的 OVERLAPPED 元素的引用即可。如下例所示：

```
PER_IO_OPERATION_DATA PerIoData;
WSABUF wbuf;
DWORD Bytes,Flags;

//初始化 wbuf ...

WSARecv(socket,&wbuf,1,&Bytes,&Flags,&(PerIoData.Overlapped),
NULL);
```

在工作器线程的后面部分，GetQueuedCompletionStatus 函数返回了一个重叠结构和完成键。获取单 I/O 数据应使用宏 CONTAINING_RECORD。例如：

```
PER_IO_DATA * PerIoData = NULL;
OVERLAPPED  * lpOverlapped = NULL;

ret = GetQueuedCompletionStatus(
    CompPortHandle,
    &Transferred,
    (PULONG_PTR)&CompletionKey,
    &lpOverlapped,
```

```
        INFINITE);
    //检查成功的返回
    PerIoData = CONTAINING_RECORD(lpOverlapped,PER_IO_DATA,Overlapped);
```

应该使用这个宏；否则，结构 PER_IO_DATA 的成员 OVERLAPPED 就始终不得不首先出现，这会成为一个危险的假设(多个开发者开发同一段代码时尤为严重)。

可以使用单 I/O 结构的一个字段来指示被投递的操作类型，从而可以确定到底是哪个操作投递到了句柄之上。在我们的例子中，OpdrationType 字段应设为可以指示读写等操作的值。对单 I/O 操作数据来说，它最大的一个优点便是允许我们在同一个句柄上，同时管理多个 I/O 操作(读 / 写、多个读、多个写等等)。大家此时或许会产生这样的疑问：在同一个套接字上，真的有必要同时投递多个 I/O 操作吗？答案在于系统的伸缩性，或者说扩展能力。例如，假定我们的计算机安装了多个中央处理器，每个处理器都在运行一个工作器线程，那么在同一个时候，完全可能有几个不同的处理器在同一个套接字上，进行数据的收发操作。

在继续讨论问题之前，需要强调一下和 Windows 完成端口有关的另一个重要方面。所有重叠操作可确保按照应用程序安排好的顺序执行。然而，不能确保从完成端口返回的完成通知也按照上述顺序执行。也就是说，对于两个重叠 WSARecv 操作，如果一个应用程序提供前者 10KB 的缓冲，而后者是 12KB 的缓冲，则 10KB 缓冲先被填充，然后 12KB 缓冲被填充。应用程序的工作器线程可能会在 10KB 操作完成事件之前从 GetQueuedCompletionStatus 接收到 12KB WSARecv 的通知。当然，在一个套接字上投递多重操作时，这个问题不会造成多大影响。

为了完成前述的简单回应服务器示例，需要提供一个 ServerWorkerThread 函数。下述代码展示了如何设计一个工作器线程的例程，令其使用单句柄数据以及单 I/O 操作数据为 I/O 请求提供服务。

```
DWORD WINAPI ServerWorkerThread(
LPVOID CompletionPortID)
{
    HANDLE CompletionPort = (HANDLE)CompletionPortID;
    DWORD BytesTransferred;
    LPOVERLAPPED Overlapped;
    LPPER_HANDLE_DATA PerHandleData;
    LPPER_IO_DATA PerIoData;
    DWORD SendBytes,RecvBytes;
    DWORD Flags;
    while(TRUE)
    {
        //等待和完成端口关联的任意套接字上的 I/O 完成

        ret = GetQueuedCompletionStatus(CompletionPort,
            &BytesTransferred,(LPDWORD)&PerHandleData,
            (LPOVERLAPPED * )&PerIoData,INFINITE);
```

```
        //先检查一下，看看是否在套接字上已有错误发生；
        //如果发生了，关闭套接字，并清除和这个套接字关联的单句柄数据和单 I/O 操作数据

        if (BytesTransferred == 0 &&
            (PerIoData->OperationType == RECV_POSTED ||
            PerIoData->OperationType == SEND_POSTED))
        {
            // BytesTransferred 为 0 时，表明套接字已被通信对方关闭，因此我们也要关闭套接字
            //注意：单句柄数据用来引用和 I/O 关联的套接字

            closesocket(PerHandleData->Socket);

            GlobalFree(PerHandleData);
            GlobalFree(PerIoData);
            continue;
        }

        //为完成的 I/O 请求服务。可以通过查看单 I/O 操作数据中包含的 OperationType 字段，
        //来确定刚完成的是哪个 I/O 请求
        if (PerIoData->OperationType == RECV_POSTED)
        {
            //对 PerIoData->Buffer 中接收到的数据施加某种操作
        }

        //投递另外一个 WSASend 或 WSARecv 操作
        //作为示范，这里将投递另一个 WSARecv() I/O 操作
        Flags = 0;

        //为下一个重叠调用建立单 I/O 操作数据。
        ZeroMemory(&(PerIoData->Overlapped),
            sizeof(OVERLAPPED));
            PerIoData->DataBuf.len = DATA_BUFSIZE;
            PerIoData->DataBuf.buf = PerIoData->Buffer;
            PerIoData->OperationType = RECV_POSTED;
            WSARecv(PerHandleData->Socket,
                &(PerIoData->DataBuf),1,&RecvBytes,
                &Flags,&(PerIoData->Overlapped),NULL);
        }
    }
```

对于一个给定的重叠操作，如果发生错误，则 GetQueuedCompletionStatus 将返回 FALSE。因为完成端口是 Windows 采用的一种 I/O 构造机制，所以，如果调用 GetLastError 或 WSAGetLastError，则错误代码极有可能是一个 Windows 错误代码，而非 Winsock 错误代码。要想找到对等的 Winsock 错误代码，可以在指定了套接字句柄和结构 WSAOVERLAPPED 的情况下，对已完成的操作调用

WSAGetOverlappedResult，之后 WSAGetLastError 将返回转换后的 Winsock 错误代码。

最后要注意的一处细节(在上面最后两个例子中没有提到)，是如何正确地关闭 I/O 完成端口——特别是同时运行了一个或多个线程，在几个不同的套接字上执行 I/O 操作的时候。要注意的一个主要问题是，在进行重叠 I/O 操作的同时，应避免强行释放 OVERLAPPED 结构。要想不出现这种情况，最好的办法是针对每个套接字句柄，调用 closesocket 函数，则任何尚未进行的重叠 I/O 操作都会完成。一旦所有套接字句柄都已关闭，使需在完成端口上终止所有工作器线程的运行。要想做到这一点，可以使用 PostQueuedCompletionStatus 函数，向每个工作器线程都发送一个特殊的完成数据包。该函数会指示每个线程立即结束并退出。下面是 PostQueuedCompletionStatus 函数的定义：

```
BOOL PostQueuedCompletionStatus(
    HANDLE CompletionPort,
    DWORD dwNumberOfBytesTransferred,
    ULONG_PTR dwCompletionKey,
    LPOVERLAPPED lpOverlapped
);
```

其中，CompletionPort 参数指明程序想向其发送一个完成数据包的完成端口对象。而就 dwNumberOfBytesTransferred、dwCompletionKey 和 lpOverlapped 这 3 个参数来说，每一个都允许指定一个值，直接传递给 GetQueuedCompletionStatus 函数中对应的参数。这样，一个工作器线程在收到传递过来的 3 个 GetQueuedCompletionStatus 函数参数后，便可根据这 3 个参数中的一个设置的特殊值，决定何时应该退出。例如，可用 dwCompletionPort 参数传递 0 值，而工作器线程会将其解释成终止指令。一旦所有工作器线程都已关闭，可使用 CloseHandle 函数关闭完成端口，最终安全退出程序。

完成端口 I/O 模型是目前为止在性能和伸缩性方面表现最好的一个。和完成端口相关联的套接字数目没有任何限制，而且仅需要少量线程为完成 I/O 进行服务。使用完成端口开发可伸缩、高性能的服务器的更多相关信息请参见第 6 章。

5.3　I/O 模型的问题

现在，对于如何挑选最适合自己应用程序的 I/O 模型，大家心中可能还有些问题。前面已经提到，每种模型都有自己的优点和缺点。同开发一个简单的阻塞模式的应用程序相比(运行许多服务线程)，其他每种 I/O 模型都需要更为复杂的编程工作。因此，针对客户机和服务器应用程序的开发，这里提供了下述建议。

客户机的开发

若打算开发一个客户机应用程序，令其同时管理一个或多个套接字，那么建议采用重叠 I/O 或

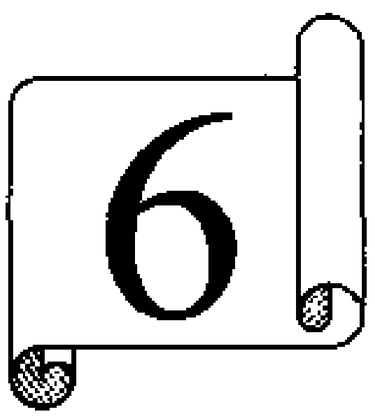
WSAEventSelect 模型，以便在一定程度上提升性能。然而，假如开发的是一个以 Windows 为基础的应用程序，要进行窗口消息的管理，那么因为 WSAAsyncSelect 本身便是从 Windows 消息模型借鉴来的，所以 WSAAsyncSelect 模型恐怕是一种更好的选择。若采用这种模型，程序一开始便具备了处理消息的能力。

服务器端开发

若开发的是一个服务器应用程序，要在一个给定的时间同时控制几个套接字，建议大家采用重叠 I/O 模型，这同样是从性能出发而考虑的。但是，如果预计到自己的服务器在任何给定的时间，都会为大量 I/O 请求提供服务，便应考虑使用 I/O 完成端口模型，从而获得更好的性能。

5.4 小　　结

至此，本书已全面介绍了可用于 Winsock 的各种 I/O 模型。这些模型包括简单的阻塞 I/O，到高性能的完成端口 I/O(获得最大的数据吞吐率)，使不同的应用程序可根据自己的要求，对 Winsock I/O 的性能进行充分的调整。到目前为止，大家已了解了可选用哪些传送协议、可设置哪些套接字的创建属性、如何创建基本的客户机 / 服务器应用程序以及与 Winsock 有关的其他常规主题。第 6 章将着重讨论如何编写高性能、可伸缩的服务器，其余章节将讲述一些更专门的 Winsock 主题。



可伸缩的 Winsock 应用程序

开发 Winsock 应用程序一直被被认为是很神秘、很难学的。实际上，只需要掌握几个基本原则，如创建套接字、连接套接字、接受连接及收发数据。真正的困难在于开发能够处理单个或成千上万个连接的可伸缩的 Winsock 应用程序。本章将叙述如何在 Windows NT 中编写可伸缩的 Winsock 应用程序。本章的介绍主要着重于客户机－服务器模型的服务器一侧，不过，有些主题对两侧都适用。

有关编写可伸缩应用程序的讨论适用于服务器应用程序，因此，自然也就仅仅适用于 Windows NT 4.0 及其后期版本。未包括 Windows NT 前期版本的原因是，很多将讨论到的功能需要 Winsock 2，而只有 Windows NT 4.0 及其后期版本才支持 Winsock 2。这里的讨论重点将集中在 TCP/IP 协议上，讨论的所有主题都能很容易地应用于其他面向连接、基于流的协议。有些主题也适用于 UDP/IP(如资源管理)，但无连接、基于消息的协议将不被提及。

本章将首先讨论一些不同的 Winsock API 函数，如 AcceptEx、TransmitFile 及 ConnectEx，这些函数用于可伸缩、高性能的应用程序。通常，因为原始的 Winsock 规范遗漏了几个关键的异步函数，所以这些函数附加了不同版本的操作系统的微软特有扩展。接着讨论的是实现可伸缩服务器的必要步骤，以及如何处理连接数变得非常大时发生的低资源状况。

6.1 API 及可伸缩性

在 Windows NT 平台上真正提供伸缩性的 I/O 模型只有重叠 I/O，它使用完成端口来执行通知。第 5 章讨论了各种不同的套接字 I/O 方法，并说明了对于较大数目的连接而言，完成端口提供了最大的灵活性和实现的简易性。类似 WSAAsyncSelect 和 select 的机制使得从 Windows 3.1 和 UNIX 进行端口操作更为容易(WSAAsyncSelect 用于 Windows 3.1，select 用于 UNIX)，但它们不适合进行伸缩。因为操作系统存在同时等待事件的局限，所以基于事件的模型不可伸缩。

重叠 I/O 其他主要的优势，是几个仅能以重叠方式调用的微软特有扩展。使用重叠 I/O 时，有几个选项可以决定接收通知的方式。因为操作系统存在只能同时等待 64 个对象的局限，所以基于事件

的模型不可伸缩，这就使得多个线程的使用成为必要。这种做法不仅没有效率，还需要大量的开销，以使将事件分派给可用工作器线程。由于几方面的原因，带回叫的重叠 I/O 并不是可有可无的。首先，许多微软特有扩展不允许对完成通知使用 APC(Asynchronous Procedure Call，异步程序调用)。其次，由于 APC 在 Windows 上的处理方式，应用程序的线程有可能会被闲置。一旦进程进入一种警觉的等待状态，所有挂起的 APC 将按照 FIFO(first in first out，先入先出)原则来处理。现在考虑这样一种情形，服务器建立了一个连接，并用完成函数投递了一个重叠 WSARecv。如果有数据需要接收，则完成例程将发动并投递另外一个重叠 WSARecv。根据计时环境和 APC 内执行的任务数量，另一个完成函数将排队等待(因为还有更多的数据需要读取)。这会导致服务器线程“挨饿”，直到该套接字上没有挂起的数据为止。

在对可伸缩的 Winsock 应用程序进行更深入的探讨之前，我们将讨论辅助我们开发可伸缩服务器的微软特有扩展。这些 API 是：TransmitFile、AcceptEx、ConnectEx、TransmitPackets、DisconnectEx 及 WSARecvMsg。还有一个相关的扩展函数 GetAcceptExSockaddrs，它一般和 AcceptEx 结合使用。

在描述这些函数之前，应该说明的一点是，这些函数是在 MSWSOCK.H 中定义的。另外，其中仅有 3 个函数(TransmitFile、AcceptEx 及 GetAcceptExSockaddrs)是真正从 MSWSOCK.DLL 导出的。不过，应用程序应该避免使用这 3 个函数，而应该动态加载扩展函数，所有其余的扩展 API 也都要求动态加载。并非所有提供程序都必须支持这些 API，因此最好根据当前使用的提供程序显式地加载这些 API。加载扩展 API 的示例请参见第 7 章及 SIO_GET_EXTENSION_FUNCTION_POINTER。

6.1.1 AcceptEx

对可伸缩的 TCP/IP 服务器而言最有用的扩展 API 也许就算 AcceptEx 了。利用这个函数，服务器可以投递一个异步调用，该调用将接受下一个传入的客户机连接。这个函数的定义为：

```
BOOL
PASCAL FAR
AcceptEx (
    IN SOCKET sListenSocket,
    IN SOCKET sAcceptSocket,
    IN PVOID lpOutputBuffer,
    IN DWORD dwReceiveDataLength,
    IN DWORD dwLocalAddressLength,
    IN DWORD dwRemoteAddressLength,
    OUT LPDWORD lpdwBytesReceived,
    IN LPOVERLAPPED lpOverlapped
    );
```

第 1 个参数是监听套接字，sAcceptSocket 参数是一个有效的、未绑定的套接字句柄，将被分配到下一个客户机连接上。因此需在投递 AcceptEx 调用前创建客户机的套接字句柄。因为套接字创建开销较大，所以这种做法很必要，如果一个服务器希望尽可能快地处理客户机连接，它就需要一个已创

建的套接字库，以便把新的连接分配进去。

sAcceptSocket 之后紧随的 4 个参数相互关联。lpOutputBuffer 参数中含有用于客户机连接的本地或远程地址，还有一个可选择的缓冲区，该缓冲区用来接收来自客户机的第 1 个数据块。dwReceiveDataLength 参数指明所提供的缓冲区需要多少字节数，该缓冲区用来接收客户机发送的数据。应用程序如果选择不接收数据，可将其指定为零。dwLocalAddressLength 参数指定套接字地址结构的大小，它相当于客户机套接字的地址族加上 16 个字节。客户机套接字连接的本地地址(如果已指定)放在 lpOutputBuffer 参数中紧随接收数据之后的地方。dwRemoteAddressLength 参数与 lpOutputBuffer 参数一样。客户机连接的远程地址(如果已指定)将被写入前一个参数中，紧随接收数据及本地地址之后。应注意，dwReceiveDataLength 参数可以是零，但 dwLocalAddressLength 参数和 dwRemoteAddress- Length 参数均不能为零。

lpdwBytesReceived 参数指明操作立刻成功时，新建的客户机连接上所收到的字节数。最后的 lpOverlapped 参数是用于这个重叠操作的 WSAOVERLAPPED 结构，该参数是必需的——如果想执行一个阻塞的接受调用，使用 accept 或 WSAAccept 就行了。

在继续讨论之前，我们用少许时间来看看一个使用了 AcceptEx 函数的示例。下列代码创建了一个 IPv4 套接字，并投递了一个单一的 AcceptEx。

```
SOCKET                  s,sclient;
HANDLE                  hCompPort;
LPFN_ACCEPTEX           lpfnAcceptEx = NULL;
GUID                    GuidAcceptEx = WSAID_ACCEPTEX;
//第12章将对WSAOVERLAPPEDPLUS类型展开详细叙述
//该类型包括重叠操作的上下文信息，还包括一个WSAOVERLAPPED结构
WSAOVERLAPPEDPLUS       ol;
SOCKADDR_IN             salocal;
DWORD                   dwBytes;
char                    buf[1024];
int                     buflen = 1024;

//创建完成端口
hCompPort = CreateIoCompletionPort(INVALID HANDLE_VALUE,
                          NULL,
                          (ULONG_PTR)0,
                          0
                          );

//创建监听套接字
s = socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);

//将监听套接字和完成端口关联起来
CreateIoCompletionPort((HANDLE)s,
```

```
                hCompPort,
                (ULONG_PTR)0,
                0
                );

//将套接字绑定到本地端口
salocal.sin_family = AF_INET;
salocal.sin_port = htons(5150);
salocal.sin_addr.s_addr = htonl(INADDR_ANY);
bind(s,(SOCKADDR *)&salocal,sizeof(salocal));

//将套接字置为监听
listen(s,200);

//加载AcceptEx函数
WSAIoctl(s,
        SIO_GET_EXTENSION_FUNCTION_POINTER,
        &GuidAcceptEx,
        sizeof(GuidAcceptEx),
        &lpfnAcceptEx,
        sizeof(lpfnAcceptEx),
        &dwBytes,
        NULL,
        NULL
        );

//为已接受的连接创建客户机套接字
sclient = socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);

//初始化"扩展的"重叠结构
memset(&ol,0,sizeof(ol));
ol.operation = OP_ACCEPTEX;
ol.client = sclient;

lpfnAcceptEx(s,
        sclient,
        buf,
        buflen -((sizeof(SOCKADDR_IN)+16)*2),
        sizeof(SOCKADDR_IN)+16,
        sizeof(SOCKADDR_IN)+16,
        &dwBytes,
        &ol.overlapped
        );

//在完成函数内部调用GetQueuedCompletionStatus
```

```
//在AcceptEx操作完成后，将已接受的客户机套接字和完成端口关联起来
```

这个示例做了一点简化，但展示了必要的步骤。它先表明如何建立一个监听套接字，而监听套接字的有关内容参见本书前面部分。然后再说明如何加载 AcceptEx 函数。因为对于每个调用中，导出扩展函数都只是简单地以加载同一个函数告终，所以应用程序应该始终加载扩展函数本身，以避免因使用从 MSWSOCK.DLL 导出的扩展函数而导致的性能损失。接下来要建立应用程序专用的重叠结构，该结构包含关于异步操作的必要信息，以便在异步操作完成时，服务器能够知晓发生了什么。为简便起见，这里并未包含该类型的实际声明，有关内容请参见第 5 章。一旦 AcceptEx 操作完成，新接受的客户机套接字将和完成端口关联起来。

同时应注意到，因为 AcceptEx 的高性能特性，监听套接字的套接字属性不会自动被客户机套接字继承。要继承属性，服务器必须用 SO_UPDATE_ACCEPT_CONTEXT 及客户机套接字句柄调用 setsockopt。更多内容请参见第 7 章。

另外一点要清楚，如果为 AcceptEx 指定了一个接收缓冲区(例如，dwReceiveDataLength 大于 0)，则重叠操作只有在连接上收到至少一个字节的数据之后才能完成，这点第 5 章中提到过。因此恶意的客户机可能投递许多连接，但决不发送任何数据。第 5 章讨论了避免这种情况的方法，即使用 SO_CONNECT_TIME 套接字选项。Windows NT 4.0 及其后期的版本均支持 AcceptEx 函数。

6.1.2　GetAcceptExSockaddrs

因为需要这个函数来对传递到接受调用的缓冲区内的本地及远程地址进行解码，所以这个函数真可算得上是 AcceptEx 函数的辅助函数。单个缓冲区不仅容纳连接的本地及远程地址，还容纳连接上收到的任何数据。指明要接收的任何数据将始终放在该缓冲区的开始部位，地址紧跟其后。不过，这些地址是打包形式的地址，GetAcceptExSockaddrs 函数将把它们拆开放到适当的相应地址族的结构中。该函数的定义为：

```
VOID
PASCAL FAR
GetAcceptExSockaddrs (
    IN PVOID lpOutputBuffer,
    IN DWORD dwReceiveDataLength,
    IN DWORD dwLocalAddressLength,
    IN DWORD dwRemoteAddressLength,
    OUT struct sockaddr **LocalSockaddr,
    OUT LPINT LocalSockaddrLength,
    OUT struct sockaddr **RemoteSockaddr,
    OUT LPINT RemoteSockaddrLength
    );
```

前 4 个参数与 AcceptEx 调用中的参数一样，而且它们必须和传递到 AcceptEx 中的数值相匹配。

也就是说，如果将 AcceptEx 中的 dwReceiveDataLength 参数指定为 1 024，则也要将 1 024 传递给 GetAcceptExSockaddrs。其余 4 个参数是本地及远程地址的 SOCKADDR 指针及其长度。这些参数都是输出参数。下面的代码展示了如何在前一个示例中的 AcceptEx 调用完成之后，调用 GetAcceptEx-Sockaddrs：

```
//buf和buflen在前面已经定义过了
SOCKADDR  *lpLocalSockaddr = NULL,
          *lpRemoteSockaddr = NULL;
int  LocalSockaddrLen = 0,
RemoteSockaddrLen = 0;
LPFN_GETACCEPTEXSOCKADDRS lpfnGetAcceptExSockaddrs=NULL;
//加载GetAcceptExSockaddrs函数

lpfnGetAcceptExSockaddrs(
    buf,
    buflen -((sizeof(SOCKADDR_IN)+16)*2),
    sizeof(SOCKADDR_IN)+16,
    sizeof(SOCKADDR_IN)+16,
    &lpLocalSockaddr,
    &LocalSockaddrLen,
    &lpRemoteSockaddr,
    &RemoteSockaddrLen
   );
```

函数完成之后，lpLocalSockaddr 和 lpRemoteSockaddr 参数指向指定的缓冲区中的某个位置，在那里套接字地址已被解包成正确的套接字地址结构。

6.1.3 TransmitFile

TransmitFile 是一个扩展的 API，它允许在套接字连接上发送一个打开的文件。这使得应用程序可以避免亲自去打开文件，重复地在文件执行读入操作，再将读入的那块数据写入套接字。相反，已打开的文件的句柄是各套接字连接一起给出的，在套接字上，文件数据的读入和发送都在核心模式下进行。这就避免了亲自执行文件读入时必需的多重内核变换。这个 API 的定义为：

```
BOOL
PASCAL FAR
TransmitFile (
    IN SOCKET hSocket,
    IN HANDLE hFile,
    IN DWORD nNumberOfBytesToWrite,
    IN DWORD nNumberOfBytesPerSend,
    IN LPOVERLAPPED lpOverlapped,
    IN LPTRANSMIT_FILE_BUFFERS lpTransmitBuffers,
```

```
    IN DWORD dwReserved
    );
```

第 1 个参数是连接套接字。hFile 参数是一个打开文件的句柄，这个参数可以是 NULL，此时将传输 lpTransmitBuffers。当然，使用 TransmitFile 来发送仅用于存储的缓冲区没有什么意义。NNumberOfBytesToWrite 参数是从文件发送的字节数，取值为零时表示发送整个文件。nNumberOfBytesPerSend 参数指明每个发送操作中所发送的每个数据块的大小，如果将其指定为零，系统就使用默认的发送大小。Windows NT 工作站上默认的发送大小是 4K，而在 Windows 服务器上是 64K。lpOVERLAPPED 结构可有可无，但得注意，如果忽略掉这个结构，文件传输将从当前文件指针的位置开始。LpTransmitBuffers 参数是一个 TRANSMIT_FILE_BUFFERS 结构，包含存储缓冲区，这个存储缓冲区将在文件被传输之前或之后传输。lpTransmitBuffers 参数是一个可选参数。最后一个参数是可选标志，它将影响文件操作的行为。表 6.1 包含了可能的标志及相应含义。另外，可以指定多个标志。

表 6.1　TransmitFile 标志

标　志	意　义
TF_DISCONNECT	在 TransmitFile 操作进入等待队列后，发起一个传输层断开动作
TF_REUSE_SOCKET	为套接字句柄的重新使用作好准备。在 TransmitFile 完成后，套接字句柄可用作 AcceptEx 中的客户机套接字。只有当 TF_DISCONNECT 也被指定时，这个标志才会生效
TF_USE_DEFAULT_WORKER	指示文件传输使用系统的默认线程，这对大型文件的发送很有用
TF_USE_SYSTEM_THREAD	这个选项也指示 TransmitFile 操作使用系统默认线程来进行
TF_USE_KERNEL_APC	指明应该使用核心异步过程调用来处理 TransmitFile 请求，而不用工作器线程。注意，如果应用程序处于等待状态(尽管不一定非得是警觉等待状态)，则核心 APC 只能由应用程序安排时间来运行
TF_WRITE_BEHIND	指明 TransmitFile 请求应该立即返回，即使远端可能还没有确认已收到数据。这个标志不能与 TF_DISCONNECT 或 TF_REUSE_SOCKET 标志同时使用

对基于文件的 I/O(如 Web 服务器)来说，TransmitFile 函数很有用。另外，它的一个有用的特性，是能够指定标志 TF_DISCONNECT 和 TF_REUSE_SOCKET。如果两个标志都指定，一旦发送操作完成，文件和(或)存储缓冲区都将被传输，套接字也将断开。同时，传递到 API 的套接字句柄将被用作 AccceptEx 中的客户机套接字，或用作 ConnectEx 中的连接套接字。因为套接字创建耗费非常之大，所以这一点极其有用。服务器可以用 AcceptEx 来处理客户机连接，然后用 TransmitFile 发送数据(指定上述标志)，过后套接字句柄可以在随后对的 AcceptEx 调用中使用。

注意，可以使用空文件句柄及空 lpTransmitBuffers 来调用 TransmitFile，不过仍然要指定 TF_DISCONNECT 和 TF_REUSE_SOCKET。这个调用不会发送任何数据，只是会让套接字在 AcceptEx 中重新使用。对于不支持 DisconnectEx API(本章稍后将叙述)的平台来说，这是一个很好的变通方法。最后，Windows NT 4.0 及后期版本均支持 TransmitFile 函数。同时，因为 TransmitFile 着重于服务器应用程序，因此只有在 Windows 的服务器版本上，其功能才能得到完全发挥。对于家庭版或专业版，在任何时候，只可以有两个未完成的 TransmitFile(或 TransmitPackets)调用。如果超过这个数目，则多余的将排队等候，直到正在执行的调用结束之后，才会被处理。

6.1.4 TransmitPackets

因为 TransmitPackets 扩展也是用来发送数据的，所以它和 TransmitFile 类似。两者的区别在于，TransmitPackets 可以按任意顺序发送任意数量的文件及存储缓冲。该函数的定义为：

```
BOOL
(PASCAL FAR *LPFN_TRANSMITPACKETS) (
    SOCKET hSocket,
    LPTRANSMIT_PACKETS_ELEMENT lpPacketArray,
    DWORD nElementCount,
    DWORD nSendSize,
    LPOVERLAPPED lpOverlapped,
    DWORD dwFlags
    );
```

第 1 个参数是已建立连接的套接字，数据将从它上边发送。另外，TransmitPackets 在数据报及面向流的协议(如 TCP/IP 和 UDP/IP)上边运行，这点和 TransmitFile 不同。lpPacketArray 参数是一个或多个 TRANSMIT_PACKETS_ELEMENT 结构组成的数组，后面将对该结构进行定义。nElementCount 参数只是用来指明 TRANSMIT_PACKETS_ELEMENT 数组中的成员个数。nSendSize 参数和 TransmtFile 的 nNumberOfBytesPerSend 参数一样。lpOverlapped 参数表示重叠结构是可选的。dwFlags 和 TransmtFile 中的那些标志一样，其选项参见表 6.1。惟一例外的是，标志名称以 TP 开头，而不是 TF——但含义完全一样。因为 TransmitPackets 在数据报上运行，所以 TP_DISCONNECT 和 TP_REUSE_SOCKET 对数据报来说毫无意义，指定它们将导致错误发生。

TRANSMIT_PACKETS_ELEMENT 结构的定义为：

```
typedef struct _TRANSMIT_PACKETS_ELEMENT {
    ULONG   dwElFlags;
    #define TP_ELEMENT_MEMORY  1
    #define TP_ELEMENT_FILE        2
    #define TP_ELEMENT_EOP         4
    ULONG   cLength;
    union {
        struct {
```

```
        LARGE_INTEGER   nFileOffset;
        HANDLE          hFile;
    };
    PVOID               pBuffer;
};
 TRANSMIT_PACKETS_ELEMENT, *PTRANSMIT_PACKETS_ELEMENT,
FAR  *LPTRANSMIT_PACKETS_ELEMENT;
```

第 1 个字段表明这个结构包含的缓冲区的类型，要么是存储缓冲区，要么是文件缓冲区，两者分别由 TP_ELEMENT_MEMORY 和 TP_ELEMENT_FILE 指定。TP_ELEMENT_EOP 标志可和另外两个标志中的其中一个一起作按位“或”运算。这样就表明在一个发送操作中，该结构不能与随后的结构联合在一起。这样使得应用程序可以推断出通信在线路上的分布方式。cLength 字段指明应该从文件存储缓冲区传输多少字节的数据。如果这个结构包含一个文件指针，则 cLength 为零时表明应传输整个文件。这个共用体要么包含一个指向内存缓冲区的指针，要么包含一个打开文件的句柄，以及该文件中的偏移值。在 TRANSMIT_PACKETS_ELEMENT 的多个元素中，引用同一个文件句柄是可能的。在这种情况下，偏移可以指定缓冲区从哪里开始。另外，偏移值为-1 时，表明从该文件当前的文件指针位置开始传输。

这里针对数据报套接字使用 TransmitPackets 的一个忠告：系统能够极其快地处理发送的请求并将它们放入等待队列，因此可能会有过量的数据报堆积在协议驱动程序处。此时，对于不可靠的协议而言，系统最好的做法是，在将数据包发送到线上之前把它们丢弃。

Windows XP 及其后期版本均支持 TransmitPackets 扩展 API，不过这个扩展也受到与 TransmitFile 同种类型的限制。在非服务器版本的 Windows NT 平台上，在任意给定时间，只能有两个未完成的 TransmitPackets(或 TransmitFile)调用。

6.1.5　ConnectEx

ConnectEx 扩展函数是一个极其必需的 API，Windows XP 及其后期版本均支持该函数。这个函数允许重叠的连接调用。以前，发动多个连接调用而又不是为每个连接使用一个线程的惟一途径，是使用多个非阻塞连接，但这样做十分麻烦，不好管理。函数 ConnectEx 的定义为：

```
BOOL
(PASCAL FAR *LPFN_CONNECTEX)(
    IN SOCKET s,
    IN const struct sockaddr FAR *name,
    IN int namelen,
    IN PVOID lpSendBuffer,
    IN DWORD dwSendDataLength,
   OUT LPDWORD lpdwBytesSent,
    IN LPOVERLAPPED lpOverlapped
    );
```

第 1 个参数是预先绑定的套接字。参数 name 表示将要连接到的远程地址，而 namelen 参数则是这个套接字地址结构的长度。lpSendBuffer 参数是一个可选指针，它指向建立连接之后将要发送的存储块，而 dwSendDataLength 参数表示要发送的字节数。lpdwBytesSent 参数被更新之后用来指明，在建立连接之后如果操作立刻完成的话，成功发送的字节数。lpOverlapped 参数是和该操作关联的 OVERLAPPED 结构。这个扩展函数只可以用重叠方式调用。

就像 AccceptEx 函数一样，由于 ConnectEx 是为执行而设计的，因此任何预先设定的套接字选项或属性都不会自动拷贝到连接套接字。为了达到这一目的，在建立连接之后，应用程序必须在套接字上调用 SO_UPDATE_CONNECT_CONTEXT。另外，就像 AcceptEx 一样，已经被 TransmitFile、TransmitPackets 或 DisconnectEx“断开及重新使用”的套接字句柄可以用作 ConnectEx 的套接字参数。

关于 ConnectEx API 没什么难题，仅有的要求是，传递到 ConnectEx 的套接字必须预先使用一个 bind 调用对之进行绑定。这里也没有什么特殊的标志，只是有一个重叠版本的连接，在建立连接后，可以选择性地多发送一个数据块。

6.1.6 DisconnectEx

这个扩展 API 很简单。它接受一个套接字句柄，执行一个传输层断开，再让套接字句柄准备好在随后的 AcceptEx 调用中重新使用。TransmitFile 和 TransmitFileAPI 均允许套接字断开，并在发送操作完成后重新使用，但这个独立的 API 不同，它是推荐给那些不采用上述两个 API 的应用程序在关闭前使用的。Windows XP 及其后期版本均支持这个扩展 API。不过，在 Windows 2000 或 Windows NT4.0 中，可以用空文件句柄及缓冲区来调用 TransmitFile，同时指定断开及重新使用标志，也能达到同样的效果。这个 API 的定义为：

```
typedef
BOOL
(PASCAL FAR *LPFN_DISCONNECTEX)(
IN SOCKET s,
IN LPOVERLAPPED lpOverlapped,
IN DWORD dwFlags,
IN DWORD dwReserved
);
```

前两个参数含义不言自明。dwFlags 参数可以指定为 0 或 TF_REUSE_SOCKET。如果标志是 0，则这个函数只是简单地断开连接。要想能够重新使用 AcceptEx 中的套接字，必须指定 TF_REUSE_SOCKET 标志。最后一个参数必须是 0，否则将返回 WSAEINVAL。如果这个函数被一个重叠结构调用，且套接字上还有挂起的操作，则 DisconnectEx 调用将返回 FALSE，错误信息为 WSA_IO_PENDING。一旦所有挂起的操作完成，且传输层断开操作已经发动，操作就会结束。否则，如果以阻塞方式调用这个函数，则只有所有挂起的 I/O 都完成，且断开操作已经发动之后，函数才会返回。注意，DisconnectEx 函数只能在面向连接的套接字上运行。

6.1.7　WSARecvMsg

这最后一个扩展函数和高性能、可伸缩 I/O 不太沾边，但它是 Windows XP(及后期版本)中出现的新成员，选出来讲是为了保持叙述的连贯性，并将它和其余的扩展 API 放在一起介绍。WSARecvMsg 基本上就相当于一个复杂的 WSARecv，仅点不同的就是前者能返回接收数据包的是哪个接口。对于一个被绑定到多重初始地址计算机的通配符地址上，并需要知道数据包到了哪个接口的数据报套接字而言，这一点很有用。这个函数的定义为：

```
typedef
INT
(PASCAL FAR *LPFN_WSARECVMSG)(
IN SOCKET s,
IN OUT LPWSAMSG lpMsg,
OUT LPDWORD lpdwNumberOfBytesRecvd,
IN LPWSAOVERLAPPED lpOverlapped,
IN LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);
```

大多数代码含义不言自明。其他扩展函数都不能用一个重叠式完成例程来调用，而这个函数可以。需要解释的参数是 lpMsg。这是一个 WSAMSG 结构，包含接收数据的缓冲区，还包含信息缓冲区，即容纳接收数据的相关信息的缓冲区。这个结构的定义为：

```
typedef struct _WSAMSG {
LPSOCKADDR        name;                 /*远程地址*/
INT               namelen;              /*远程地址长度*/
LPWSABUF          lpBuffers;                /*数据缓冲数组*/
DWORD             dwBufferCount;            /*数组中元素的数量*/
WSABUF            Control;                  /*控制缓冲区*/
DWORD             dwFlags;              /*标志*/
} WSAMSG, *PWSAMSG, *FAR LPWSAMSG;
```

第 1 个字段是一个容纳远程系统地址的缓冲区，namelen 字段则指定地址缓冲区的大小。lpBuffers 字段和 dwBuffeCount 字段与 WSARecv 中的一样。Control 字段指定一个将容纳可选控制数据的缓冲区。最后，dwFlags 也和 WSARecv 及 WSARecvFrom 中的字段一样。不过，dwFlags 还有一些额外的标志可以返回，提供所接收数据包的相关信息。这些新的标志如表 6.2 所述。

在默认情况下，调用 WSARecvMsg 不会返回控制信息。为了能够返回控制信息，套接字上必须设置一个或多个套接字选项，指明要返回的信息类型。目前，仅有一个选项支持，对于 IPv4 该选项为 IP_PKTINFO，对于 IPv6 则为 IPV6_PKTINFO。这些选项返回有关数据包被哪个本地接口接收这样的信息。关于这些选项的设置请参见第 7 章。

表 6.2 从 WSARecvMsg 返回的标志

标志	说明
MSG_BCAST	数据报被当作链路层广播接收，或它的目标地址是一个广播地址
MSG_TRUNC	数据报被截断。数据太多，不能全部拷贝到提供的接收缓冲区中
MSG_CTRUNC	控制数据被截断。在字段中提供的缓冲区太小，无法接收控制数据

只设置了适当的套接字选项，并且在 WSARecvMsg 完成之后，所请求的控制信息会通过 WSAMSG 参数中指定的 Control 缓冲区返回。所请求的每种类型的信息之前都有一个结构，这个结构指明信息的类型及其大小。这个报头结构的定义为：

```
typedef struct _WSACMSGHDR {
    SIZE_T    cmsg_len;
    INT       cmsg_level;
    INT       cmsg_type;
    /* UCHAR cmsg_data[ ]紧随其后*/
} WSACMSGHDR, *PWSACMSGHDR,  FAR  *LPWSACMSGHDR;
```

MSWSOCK.H 中定义了几个有用的宏，用它们可以抽取报文标题及其数据。

6.2 可伸缩的服务器体系结构

在介绍了微软特有扩展之后，下面将开始详细讨论可伸缩服务器的实现。因为本章注重于面向连接的协议，如 TCP/IP，所以先讨论如何接受连接，再讨论数据传输的管理。最后一节将比较详细地讨论资源管理。

6.2.1 接受连接

服务器实施的最常见的行为是接受连接。微软扩展的 AcceptEx 是惟一能够通过重叠 I/O 接受客户机连接的 Winsock 函数。前面已经提到，AcceptEx 函数需要事先通过调用 socket 创建客户机套接字。尽管有可能在调用 TransmitFile、TransmitPackets 或 DisconnectEx 之后重新使用套接字，也要求套接字必须是非绑定且无连接的。

一个回应的服务器必须一直有足够的未完成的 AcceptEx 调用，以使传入的客户机连接能够立刻得到处理。然而，不可能有大多的未完成 AcceptEx 调用来保证服务器可以立刻接受连接。记住，TCP/IP 堆栈会自动代表监听应用程序接受连接，直到达到储备极限。Windows NT 服务器目前的最大储备值为 200。如果一个服务器投递 15 个 AcceptEx 调用，而一下来了 50 个客户机连接，但没有一个客户机连接会受到拒绝。服务器的接受调用将满足前 15 个连接，然后系统将接受余下的连接——这需要消

耗储备数量，使得服务器还能接受 165 个额外的连接。然后当服务器投递另外的 AcceptEx 调用时，因为其中一个由系统排队等候的连接会返回，所以这些调用将会立刻成功。

在决定应该投递多少个 AcceptEx 操作时，服务器的特性扮演着一个重要的角色。例如，一个用来处理来自大量客户机的许多短时连接的服务器，相对于一个需要处理较少且寿命长的连接的服务器来说，前者会投递更多的并发 AcceptEx 操作。一种比较好的策略是允许 AcceptEx 调用的数目在一个下限值和上限值之间变动。应用程序可以记住挂起而未完成的 AcceptEx 操作的数量。之后，如果这些操作中的一个或多个完成之后，未完成的数量将减少到上限值以下，就可以投递其他 AcceptEx 调用。当然，如果某个时刻 AcceptEx 完成，但未完成的接受调用大于或等于上限值，则不会有额外的调用投递到当前的 AcceptE 处理中去。

在 Windows 2000 及其后期版本中，Winsock 提供了一个机制，可以决定一个应用程序在投递足够的 AcceptEx 调用时是否落后于时间表。创建监听套接字后，要通过 WSAEventSelect API 调用并注册 FD_ACCEPT 通知，将套接字与某个事件关联起来。如果没有挂起的 AcceptEx 操作(根据储备值由系统接收)，但有传入的客户机连接，则事件将被传信。这种机制甚至还可用作投递额外 AcceptEx 操作的指示。

使用 AcceptEx 的一个显著的好处，是在接受客户机连接的同时，还可以接收数据。对那些由客户机发送初始请求的服务器而言，这显得比较理想。然而，正如第 5 章所述，只有在至少接收到一个字节的数据之后，AcceptEx 操作才能完成。为了预防恶意攻击或过时连接，服务器应该循环通过未完成的 AcceptEx 操作中所有客户机套接字句柄，并用 SO_CONNECT_TIME 调用 getsockopt，不管套接字实际上是否已连接，该调用都会返回。如果套接字已连接，返回值就会大于 0。如果返回值是-1，则表明没有连接。如果实施了 WSAEventSelect 建议，则事件被传信的时候，就是检查未完的接受调用中的客户机套接字句柄是否已被连接的好机会。AcceptEx 调用接受一个传入的连接之后，就会开始等待接收数据，此刻未完成的接受调用就会少一个。如果没有剩余的接受调用，则事件将在下次客户机连接传入时被传信。这里有个警告，无论在何种情况下，应用程序都不应该关闭一个在 AcceptEx 调用中使用的、未被接受的客户机套接字句柄，原因是这样会导致内存泄漏。由于性能原因，在未连接的客户机句柄被关闭时，与 AcceptEx 调用相关联的核心模式结构不会被彻底清除干净，除非建立一个新的客户机连接或关闭监听套接字。

将 AcceptEx 请求投递到其中一个工作器线程(这些线程处理来自完成端口的通知)看起来好像更简单，也很符合逻辑，但因为创建套接字花费颇高，所以应该避免这样做。另外，工作器线程内部应该尽量避免任何复杂的计算，以便服务器可以尽可能快地处理完成通知。创建套接字花费高的一个原因，是 Winsock 2.0 的分层结构系。当服务器创建套接字时，在套接字创建好并返回应用程序之前，它可能会在多个提供程序中路由，而每个提供程序都要执行自己的任务。第 12 章将详细讨论分层提供程序。服务器应该从一个分离线程中创建客户机套接字和投递 AcceptEx 操作。当工作器线程中的一个重叠 AcceptEx 完成之后，可以用一个事件来传信发动接受操作的线程。

6.2.2 数据传输

与客户机连接上之后，服务器就需要传输数据。这个过程相当直接，所有的数据收发还应该用重叠 I/O 来执行。在默认情况下，每个套接字都有一个与发送和接收关联的缓冲区，用于缓冲待发或待收数据。大多数情况下都不用管这些缓冲区，但通过 SO_SNDBUF 或 SO_RCVBUF 选项调用 setsockopt，也可以改变它们，或将它们设置为 0。

下面看看当发送缓冲区的大小不是 0 时，系统怎样处理一个典型的发送调用。在应用程序作出一个发送调用时，如果有足够的缓冲空间，则数据将被拷贝到套接字的发送缓冲区中，之后调用立刻成功完成，且完成事件将被投递。相反，如果套接字的发送缓冲区已满，则应用程序的发送缓冲区将被锁定，发送调用失败，并返回 WSA_IO_PENDING。在处理完发送缓冲区内的数据之后(例如，将数据交给下层的 TCP 处理)，Winsock 将直接处理锁定的缓冲区。也就是说，应用程序将从缓冲区中把数据直接交给 TCP，完全绕过套接字的发送缓冲区。

接收数据时也是一样的道理。在实施重叠接收调用时，如果数据已在连接上被接收，则数据将被放到套接字的接收缓冲区中。之后数据将被直接拷贝到应用程序的缓冲区(直至填满)中，接收调用返回成功信息，接着投递完成通知。然而，如果套接字接收缓冲区是空的，则作出重叠接收调用时，应用程序的缓冲区将被锁定，调用将会失败，并返回 WSA_IO_PENDING。数据到达连接之后，将被直接拷贝到应用程序的缓冲区中，而绕过套接字的接收缓冲区。

因为只要始终有足够的重叠发送及接收操作被投递，就能避免额外的内存拷贝，所以将单套接字缓冲区设置为 0 通常不会使性能得到提升。因为应用程序的发送缓冲区将始终被锁定，直到可以被下传给 TCP 处理，所以停用套接字的发送缓冲区对性能的影响比停用接收缓冲区小。然而，如果接收缓冲被设为 0，且没有未完成的重叠接收调用，则任何传入的数据都只能停留在 TCP 层的缓冲区。TCP 驱动程序的缓冲区最多只能接收窗口的大小，即 17K——TCP 将根据需要增加这些缓冲区，直到达到最大限制；通常情况下，缓冲区比极限值少很多。这些 TCP 缓冲区(一个单连接)被定位在未分页内存池之中，这意味着，如果服务器有 1000 个连接，但根本没有投递接收，则将消耗 17MB 的未分页内存池。未分页内存池是一种很有限的资源，除非服务器能够保证始终有接收调用被投递给一个连接，否则不要轻易动用单套接字的接收缓冲区。

仅仅在少数几种特殊的情况下，不去动用接收缓冲区才会导致性能降低。考虑一下这种情况，服务器要处理成千上万的连接，却不能在每个连接上都投递一个接收(下一节您就会明白，这样做将导致巨大的花费)。同时，客户机零星地发送数据，传入的数据将被放入单套接字接收缓冲区中，如果此时服务器发动一个重叠接收，这个动作将没有任何实际意义。这个重叠操作发出了一个 IRP(I/O request packet，I/O 请求数据包)，动作结束之后，通知将立刻被送到完成端口。在这种情况下，服务器不能保持足够的接收调用来投递，因此最好执行简单的非阻塞接收调用来处理。

TransmitFile 和 TransmitPackets

发送数据时，服务器应该在合适的地方考虑使用 TransmitFile 和 TransmitPackets API 函数。这两个函数的好处是，在连接上大量数据可以被放入发送队列中，并导致一个 user-to-kernd 模式的传输。例如，如果服务器正向一个客户机发送文件数据，它仅需要打开指向该文件的句柄，并以发动一个 TransmitFile，而不是调用 ReadFile 和 WSASend，这将激发许多 user-to-kernd 模式的传输。同样，如果需要发送几个存储缓冲区，服务器也可以建立一个 TRANSMIT_PACKETS_ELEMENT 结构数组，并使用 TransmitPackets API。正如前面提到的，这些 API 允许断开连接并在随后的 AcceptEx 调用中重新使用套接字句柄。

6.3 资 源 管 理

如果一个计算机有足够的资源，则 Winsock 服务器就有能力处理成千上万的并发连接。然而，如果需要服务器处理的并发连接越来越多，则最终将会出现资源限制。最可能遇到的两个限制，一个是锁定页面的数量，一个是未分页内存池的使用。相对未分页内存池的消耗殆尽而言，锁定页面限制没那么严重，也比较容易避免。

对于每个重叠发送或接收操作而言，提交的数据缓冲都有可能被锁定。内存一旦被锁定，就不能从物理内存中分页出来。操作系统强加了一个可以被锁定的内存数量的上限。如果达到这个上限，重叠操作就会失败，并返回 WSAENOBUFS 错误。如果服务器在每个连接上投递很多重叠的接收操作，随着连接数的增加，这个上限就会达到。如果服务器希望处理极大数量的并发客户机，可以在每个连接上投递一个 0 字节的接收操作。因为没有缓冲和接收操作相关联，也就不需要锁定内存。因为一旦 0 字节接收操作完成，服务器就可以实施非阻塞接收，以获得套接字接收缓冲区中的所有数据，所以使用这种方法，单套接字接收缓冲就始终保留原封不动。一旦非阻塞接收以返回 WSAEWOULDBLOCK 而失败，就表示没有挂起的数据了。这个方案用于这样一种服务器，即要求尽可能大的并发连接，同时以牺牲在每个连接上的数据吞吐率为代价。

当然，对客户机如何与服务器交互了解得越透彻越好。在前一个例子中，一旦 0 字节接收完成，就执行非阻塞接收来获得缓冲数据。如果服务器获悉客户机突然发送了大量数据，则一旦 0 字节接收完成，服务器就可能会投递一个或多个重叠接收操作，以防客户机发送相当大数量的数据(大于单套接字接收缓冲区的默认值 8KB)。

另一个需要考虑的重要事项，是体系结构中服务器运行时所用的页面大小。系统在锁定传递给重叠操作的内存时，同时也会锁定页面边界。在 x86 体系结构中，页面是以 4KB 的倍数被锁定的。如果操作投递了一个 1KB 的缓冲区，则系统实际上将锁定一个 4KB 的数据块。为了避免这种浪费，重叠发送和接收缓冲区应该取为页面大小的倍数。可调用 WindowsAPIGetSystemInfo 来获取当前体系结构的页面大小。

触及未分页内存池的限制是一个更为严重的错误，并难以恢复。未分页内存池是内存的一个部分，它始终驻留于物理内存，永远不可能被分页出来。核心模式下的操作系统组件，如驱动程序，通常使用未分页内存池，其中包括 Winsock 及协议驱动程序，如 tcpip.sys。每个创建的套接字消耗未分页内存池的一小部分，用来保留套接字状态信息。当套接字被绑定到一个地址时，TCP/IP 堆栈会为本地地址信息分配额外的未分页内存池。套接字被连接时，远程地址结构也会由 TCP/IP 堆栈来分配。已建立连接的套接字总共消耗大约 2KB 的未分页内存池，从 accept 或 AcceptEx 返回的套接字则使用大约 1.5KB 的未分页内存池(因为已被接受的套接字仅需要存储远程地址)。另外，每个在套接字上发起的重叠操作需要分配一个 I/O 请求数据包，这大约要使用 500 个字节的未分页内存池。

可以看到，每个连接使用的未分页内存池数量并不大，然而，随着客户机连接数量的增加，服务器使用的未分页内存池就可能非常可观。例如，如果一个运行 Windows 2000(或更高版本)的服务器有 1GB 的物理内存，其中 256MB 就被设置为未分页内存池使用。通常，所分配的未分页内存池是物理内存的 4 分之一，在 Windows 2000 及其后期版本中上限为 256MB，而在 Windows NT 4.0 中上限为 128MB。未分页内存池为 256MB 时，可以处理 50 000 个或更多的连接，不过要留意限制那些排队等候接受新连接，或在已有连接上发送及接收的重叠操作的数量。在这个例子当中，单是已连接的套接字就消耗了未分页内存池上的 75MB(假设按前面说的每个套接字使用 1.5KB 未分页内存池)。因此，如果采用 0 字节重叠接收策略，则每个连接会分配到一个单个的 IRP，这又要使用 25MB 的未分页内存池。

如果系统用尽了未分页内存池，会有两种可能性。最好的可能是，Winsock 调用失败，返回 WSAENOBUFS。最坏的可能是系统出现终端错误而崩溃。这通常发生在核心模式组件(如第 3 方驱动程序)没能正确处理好一个失败的内存分配时。这种事情发生时，没有一种保险的途径可以恢复被耗尽的未分页内存池，而且，因为任意一个核心模式组件都能够耗尽未分页内存池，所以也没有可靠途径来监视可用未分页内存池的数量。这些讨论的重点，就是说不存在一种神奇的或编程的方法可以确定，究竟能够接受多少个并发连接及重叠操作。实质上也不可能确定，系统究竟是耗尽未分页内存池呢，还是超出锁定页面的数量，因为这两种情况都会导致 Winsock 调用失败，返回的错误信息都是 WSAENOBUFS。所以，必须在服务器上执行测试。因为这些原因，开发者必须用不同数量的并发连接及重叠操作对服务器性能进行测试，以便找到一个令人满意的方法。如果用编程强加限制来避免服务器耗尽未分页内存池，则任何 WSAENOBUFS 失败通常都是超出锁定页面限制而导致的，此时这种问题就可以通过编程方式来处理，例如，进一步限制未完成操作的数量，或关闭其中一些连接等。

6.4 服务器策略

本节将基于服务器特性来讨论处理资源的几个策略。同时，还要讨论加到客户机和服务器设计上的更多控制，这使得我们能够以避免前面讨论的诸多限制和瓶颈效应为前提来设计它们。当然，不存

在那种在各种环境下都能够打满分的简单方法。服务器可以粗略地分为两大类：高吞吐率的服务器和高连接的服务器。高吞吐率服务器更关心在少量连接上推进数据。当然，“少量连接”的意义和服务器上可用资源的数量有关。高连接服务器更关心处理大量连接，并非意图推进大量数据。

下面两节将讨论高吞吐率服务器和高连接服务器的策略。之后我们将查看从服务器示例收集的性能数，这些服务器示例在补充材料中可以找到。

6.4.1　高吞吐率

FTP 服务器就是高吞吐率服务器的一个例子，它主要关心大块数据的传输。在这种情况下，服务器把重点放在处理每个连接上，以减少传输数据所需的时间。因为并发连接越多，在每个连接上的数据吞吐率就会越低，所以为达到快速的目的，服务器必须限定并发连接的数量。FTP 服务器因为太繁忙而拒绝额外的连接就是一个例子。

这种策略的目标是 I/O。服务器应该保持足够的接收或发送操作用于投递，以提高数据吞吐率。因为每个重叠 I/O 需要内存被锁定，还需要未分页内存池的一小部分分配给和操作关联的每个 IRP，所以将 I/O 限制在较小的一组连接之内就显得很重要。服务器或许可以连续接受连接，并拥有一个相对较多的已建立的连接，但 I/O 必须被限定在一个较小的组内。

在这种情况下，服务器可以在已建立的客户机的子网上投递许多发送或接收操作。例如，服务器可以按照先入先出的方式处理客户机连接，并在前 100 个连接上投递许多重叠的发送和(或)接收操作。处理完这些客户机后，服务器就可以继续处理队列中的下一组客户机。在这个模型里面，未完成的发送和接收操作被限制在较小的一组连接之内。这避免了服务器在每个连接上盲目投递 I/O 操作而致使服务器资源很快被耗尽。

服务器应该留意监视每个连接上的未完成操作的数量，这样可以避免受到恶意客户机的攻击。例如，一个用来从客户机接收数据、处理数据并发送某种回应的服务器，应该留意未完成的发送操作有多少个。如果客户机仅仅是用数据对服务器进行狂轰滥炸，却不投递任何接收操作，服务器就有可能投递大量的重叠发送操作，而这些发送操作永远也完成不了，最终导致服务器死机。在这种情况下，一旦服务器发现未完成操作太多，就可以关掉连接。

6.4.2　最大化连接数

增大并发客户机连接的数量是两种策略中较难的一种。处理每个连接上的 I/O 变得很困难。服务器不能只是在每个连接上投递一个或多个发送或接收操作，原因是这样做要消耗大量的内存(在锁定页面方面以及在未分页内存池方面)。在这种策略中，服务器很乐意以每个连接上的数据吞吐率为代价来处理许多连接。飞毛腿信使服务器(instant messenger server)就是一个例子。服务器需要处理成千上万的连接，但每次仅需发送或接收少量字节。

用这种策略时，服务器没有必要在每个连接上投递一个重叠接收，因为这样做的话，每个接收缓冲区都会锁定许多页面。相反，服务器可以投递一个重叠的 0 字节接收操作。一旦该接收完成，服务器就执行一个非阻塞的接收操作，直到返回 WSAEWOUDLBLOCK。这使得服务器可以立即接收在该连接上接收到的所有缓冲数据。因为这个模型适合间歇式发送数据的客户机，所以它减少了锁定页面的数量，但仍然允许在每个连接上进行数据处理。

6.4.3 性能指标

本节叙述第 5 章和第 6 章提供的不同服务器的性能指标。所测试的服务器包括：使用阻塞套接字、使用带 select 的非阻塞套接字、使用 WSAAsyncSelect、使用 WSAEventSelect、带事件的重叠 I/O 以及使用带完成端口的重叠 I/O 的服务器。表 6.3 总结了这些测试的结果。每种 I/O 模型有几个条目。第 1 个条目是来自 3 个客户机的 7 000 个连接尝试。所有测试中的服务器都是回应服务器。也就是说，对每个被接受的连接而言，数据被接收之后，又会被送回客户机。每个 I/O 模型的第 1 个条目代表一个高数据吞吐率服务器，在这里，客户机总是尽可能快地将数据发往服务器。这些示例服务器都不限制并发连接的数量。其余的条目代表当客户机限制发送数据速度时的连接，限制速度是为了不超出网络可用带宽。每个 I/O 模型的第 2 个条目代表来自客户机的 12 000 个连接，客户机限制了发送数据的速度。如果服务器能够处理 12 000 个连接的大部分，则第 3 个条目将是服务器能够处理的客户机的最大数量。

前面已经提过，这里使用的服务器是第 5 章提供的，不过 I/O 完成端口服务器作了少许改动，这儿采用的 I/O 完成端口服务器限制了未完成操作的数量。这个数量被限制在 200，且在每个客户机连接上只投递一个接收操作。这个测试里使用的客户机是第 5 章所述的 I/O 完成端口客户机。连接是批量建立的，1 000 个客户机一批，这只要在客户机上指定“-c1000”选项即可。两个基于 x86 的客户机最多发动 12 000 个连接，其余的客户机用 Itanium 系统建立分批连接，每批 4 000。在已限制速度的测试中，每个客户机块被限制为每秒 200 000 字节(使用“-r200000”选项)。因此整块客户机的平均发送数据吞吐率就被限制为每秒 200 000 字节(不是每个客户机都限制到这个数量)。

表 6.3 I/O 方法性能比较

I/O 模型	尝试数/成功连接数	使用内存(KB)	未分页内存池	CPU 用量	线程	数据吞吐率(每秒发送/接收的字节)
阻塞	7 000/1 008	25 632	36 121	10%～60%	2 016	2 198 148/2 198 148
	12 000/1 008	25 408	36 352	5%～40%	2 016	404 227/402 227

续表

I/O 模型	尝试数/成功连接数	使用内存(KB)	未分页内存池	CPU 用量	线程	数据吞吐率(每秒发送/接收的字节)
非阻塞	7 000/4 011	4 208	135 123	95%～100%*	1	0/0
	12 000/5 779	5 224	156 260	95%～100%*	1	0/0
WSAAsyncSelect	7 000/1 956	3 640	38 246	75%～85%	3	1 610 204/1 637 819
	12 000/4 077	4 884	42 992	90%～100%	3	652 902/652 902
WSAEventSelect	7 000/6 999	10 502	36 402	65%～85%	113	4 921 350/5 186 297
	12 000/11 080	19 214	39 040	50%～60%	192	3 217 493/3 217 493
	46 000/45 933	37 392	80%～90%	121,624	791	3 851 059/3 851 059
重叠(事件)	7 000/5 558	21 844	34 944	65%～85%	66	5 024 723/4 095 644
	12 000/12 000	60 ,576	48 060	35%～45%	195	1 803 878/1 803 878
	49 000/48 997	241 208	155 480	85%～95%	792	3 865 152/3 834 511
重叠(完成端口)	7 000/7 000	36 160	31 128	40%～50%	2	6 282 473/3 893 507
	12 000/12 000	59 256	38 862	40%～50%	2	5 027 914/5 027 095
	50 000/49 ,997	242 272	148 192	55%～65%	2	4 326 946/4 326 496

服务器为 Pentium4 1.7GHz Xeon，内存 768MB。客户机建立在 3 台计算机上：Pentium II 233MHz，内存 128MB；Pentium II 350MHz，内存 128MB；Itanium 733MHz，内存 1GB。测试网络为 100MB 隔离集线器。所有测试计算机均安装了 Windows XP。

阻塞模型[illegible]中表现最差。阻塞服务器为每个客户机连接生成两个线程：一个发送数据，一个接收数据。在两种情形下服务器都有一小部分连接不能处理，因为它触及了系统在线程创建上的限制。于是调用以返回 ERROR_NOT_ENOUGH_MEMORY 而失败。其余的客户机连接失败时返回 WSAECONNREFUSED。

非阻塞模型只好那么一点点。它可以接受更多的连接，但达到了 CPU 的限制。非阻塞服务器把所有已连接的套接字放到 FD_SET 里边，并将它传递给 select。当 select 完成时，服务器使用 FD_ISSET 宏进行搜索，以确定该套接字是否已被传信。因为连接数的增加，使得搜索效率变低。这样只是确定一个套接字是否已被传信，就需要对数组进行线性搜索。为了部分地缓和这个问题，可以重新设计服务器，使它可以反复地单步执行从 select 返回的 FD_SET。惟一麻烦的问题是，服务器需要快速找到和该套接字句柄关联的 SOCKET_INFO 结构。在这种情况下，服务器可以提供一个较复杂的编目机制，如允许迅速查找的散列树。还要注意，因为服务器读取数据的速度不够快(正如 0 字节数据吞吐率所显示)，故 AFD 和 TCP 都在客户机连接上缓冲数据，所以未分页内存池使用量极高，这由大量的 CPU 使用率可以看出来。

WSAAsyncSelect 模型适用于小数目的客户机，但因为消息回路的额外开销使客户机迅速处理信息的能力大为减小，所以它的伸缩性不好。在两个测试中，服务器仅能处理大约三分之一的连接。客户机收到很多 WSAECONNREFUSED 错误信息，表明服务器不能足够快地处理 FD_ACCEPT 消息，因此监听储备未被用完。然而，即使是已经被接受的连接，也应该注意到，平均数据吞吐率是相当低的(即使是对于限制速度的客户机而言)。

出乎意料的是，WSAEventSelect 模型表现得十分优秀。在所有的测试中，服务器在极大程度上都能够处理所有传入的连接，同时获得非常好的数据吞吐率。这个模型的弊端，是需要用于管理新连接的线程库的额外开销。因为每个线程只能等待 64 个事件，新连接建立之后，必须创建新的线程来处理它们。同时，在最后一个建立了 45 000 多个连接的测试中，计算机变得非常迟缓。这极有可能是由于创建了用于服务许多连接的巨大数量的线程所导致。在 791 个线程中切换所需的总开销变得十分可观。因为产生了大量的 WSAENOBUFS 错误，所以服务器在某一处便再也不能接受任何连接。另外，客户机应用程序也达到其极限，再不能维持已经建立的连接(后面将对此进行详细讨论)。

带事件的重叠 I/O 模型在伸缩性方面和 WSAEventSelect 类似。两个模型的事件通知都依赖线程库，当线程切换的开销影响到模型处理客户机通信时，两个模型都会达到各自的极限。这个模型的性能指标几乎和 WSAEventSelect 的性能指标完全一样，直到线程数量增加之前，它的表现都十分良好。

最后一项是带完成端口的重叠 I/O 模型，它是所有 I/O 模型中表现最好的。它的内存使用(包括用

户内存和未分页内存池)及所接受的客户机与带事件的重叠I/O模型和WSAEventSelect模型都很相似，它们真正的差别在于CPU的使用。完成端口模型仅仅使用大约60%的CPU，而另外两个模型则需要更多的动力来维持同样数目的连接。另外一个明显的差别是，完成端口的数据吞吐率稍高一些。

进行这些测试时，有一个问题会变得明显起来，即根据客户机和服务器之间数据交互的特点，会出现相应的极限。服务器被设计成回应服务器，使得所有从客户机接收的数据都被返回。同时，每个客户机还连续向服务器发送数据(即使发送速度较低)。这导致数据总是在服务器的套接字上等待处理(要么在TCP缓冲区中，要么在AFD单套接字缓冲区中，两者都是未分页内存池)。3个表现较好的模型每次都只执行一个接收操作，然而，这意味着在大部分时间里，仍然有等待处理的数据。可以修改服务器，使得一旦连接上有数据，它就能够实施非阻塞接收。这样做会减少计算机上的缓冲数据。这种方法的弊端是，客户机持续不断地发送数据，以至于非阻塞接收可能返回大量的数据，致使其他连接"挨饿"(由于线程或完成线程不能处理其他事件或完成通知)。通常，调用一个非阻塞接收一直到WSAEWOULDBLOCK被返回，会对那些间断性地而非连续地传输数据的连接起作用。

从性能指标很容易推断出，WSAEventSelect和重叠I/O具有最好的性能。对于这两个基于事件的模型而言，为处理事件通知而建立一个线程库显得很麻烦，不过对于一个一般忙碌的服务器来说，仍然会表现出优秀的性能。因为线程间的任务切换会消耗更多的CPU，一旦连接和线程数量增加，伸缩性就会成为一个问题。因为客户机数量增加时，CPU用量的相对影响就会变小，所以完成端口模型仍旧提供了最佳的伸缩性。

6.5　Winsock直连及套接字直连协议

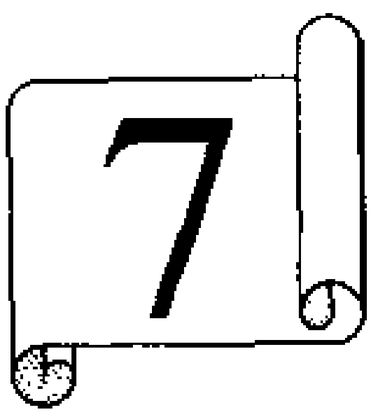
Winsock直连是在Windows 2000数据中心服务器上引进的一种高速互连，它是一个在由几家厂商(例如技嘉、康柏等)生产的特殊硬件上运行的协议。Winsock直连特殊的地方在于，它完全绕过TCP堆栈，直接奔向网络接口卡，这样就允许极端高速的数据通信。对TCP Winsock应用程序完全透明是Winsock直连的优势。也就是说，如果TCP应用程序在一个带有支持Winsock直连网卡的计算机上运行，程序就会透明地经过Winsock直连路线，而不是经由常规的以太网连接。

套接字直接协议是Winsock直连的下一代协议。它是设计来在未来版本的Windows操作系统的无限制兼容硬件上运行的。尽管这种线上协议和Winsock直连有少许差别，但它仍然对应用程序透明。

因为Winsock直连是透明的，所以在Winsock直连之上运行时，"常规"的Winsock应用程序原来遇到的问题仍然会出现。应用程序仍旧不得不管理未完成重叠操作的数量，以避免超过锁定页面或未分页内存池的限制。

6.6 小 结

本章的重点是为基于 Windows NT 的操作系统编写高性能、可伸缩 Winsock 服务器。本章讨论了几个微软特有的 Winsock 扩展，这些扩展为程序员开发这类服务器提供了有力的帮助。另外，本章不仅介绍了提高数据吞吐率的方法，还介绍了几个接受连接的途径，以便减小客户机收到拒绝连接的可能性。之后介绍了资源管理，这是编写高性能服务器所需的核心概念。最后比较了第 5 章中介绍的各种 I/O 模型的性能，看看有许多客户机连接请求时，它们的伸缩性是怎样的。



套接字选项和 I/O 控制命令

创建套接字后，通过套接字选项和 I/O 控制命令对各种属性进行操作，便可对套接字的行为产生影响。有的选项只用于信息的返回，而有的选项则可在应用程序中影响套接字的行为。ioctl 即 I/O 控制命令，也会对套接字的行为产生影响。本章着重讨论 4 个 Winsock 函数：getsockopt、setsocketopt、ioctlsocket 和 WSAIoctl。每个函数都有大量的命令，其中大多数尚未正式写入参考文档。本章将讨论各函数需要的参数和可以使用的选项，以及哪些平台对这些选项提供了支持。在此，我们假定各个选项都能在 Windows 平台上运行(Windows CE、Windows 95、Windows 98、Windows Me、Windows NT、Windows 2000 和 Windows XP)；如有例外，会专门注明。但选项在要求 Winsock 2 的支持时，情况可能有所不同。因为 Winsock 2 本身便不能在所有平台上通用，Winsock 2 的 I/O 控制命令和选项在 Windows CE 与 Windows 95 平台上均未获支持(除非已在 Windows 95 上安装了 Winsock 2 升级补丁程序)。另外还要记住：Windows CE 不支持除 TCP/IP 以外的其他专用协议的选项。

这些 I/O 控制命令和选项大多定义在 WINSOCK.H 或 WINSOCK2.H 内，这具体取决于它们到底从属于 WINSOCK 1，还是从属于 WINSOCK 2。但是，也有少数几个选项是 Microsoft 提供程序或某种传输协议所特有的。微软特有的一些扩展定义在 WINSOCK.H 和 MSWSOCK.H 这两个头文件内。而传输提供程序扩展定义在与其专用协议对应的头文件内。针对那些传输特有的选项，我们将随选项一道，指明正确的头文件是什么。要注意的是，若应用程序使用了微软特有的扩展，那么必须与 MSWOCK.LIB 建立链接。

7.1 套接字选项

getsocketopt 函数的常见用法是获得与给定套接字相关的信息。其原型如下：

```
int getsockopt(
    SOCKETs,
    int level,
    int optname,
```

```
    char FAR* optval,
    int FAR* optlen
    );
```

第 1 个参数 s 指定的是一个套接字，我们打算在这个套接字上执行指定的选项。对于使用中的具体协议来说，这个套接字必须是有效的。大多数选项都是某种特定的协议和套接字类型专有的，而其他选项适用于所有类型的套接字(特别是第 2 个参数 level)。如果 level 选项为 SOL_SOCKET，则表明它是一个通用选项，并不一定要与一种给定的协议有关。但之所以说“不一定”，是由于并非所有协议都实现了在 SOL_SOCKET 级别上定义的每一个套接字选项。例如，SO_BROADCAST 可将一个套接字置入广播模式，但并非所有协议都支持广播套接字的概念。optname 参数是我们在此真正感兴趣的选项。这些选项名均是在 Winsock 头文件内定义的常数值。最常见的独立于协议的选项(例如和 SOL_SOCKET 级别关联在一起的选项)是在 WINSOCK.H 和 WINSOCK2.H 这两个头文件中定义的。对于每种特定的协议来说，它们都有自己的头文件，其中定义了与之对应的特定选项。最后，optval 和 optlen 参数是两个变量，用于返回目标选项的值。大多数情况下，选项值都是一个整数(但也不是绝对的)。

setsocketopt 函数用于在一个套接字级别上或专用协议的级别上设置套接字选项。它的定义如下：

```
int  setsockopt (
    SOCKET s,
    int level,
    int optname,
    const char FAR * optval,
    int optlen
    );
```

该函数的参数和 getsocketopt 函数的参数相同，不过我们要为 optval 和 optlen 参数，将值传递进去。这些值是为特定的选项设定的。和 getsocketopt 函数一样，optval 大多数时候都是一个整数，但也并非总是如此。正式编程的时候，应查询对每个选项的说明，了解到底该将什么作为选项值传递进去。

调用 getsocketopt 或 setsocketopt 时，最常见的错误是试图获得一个套接字的信息，但那个套接字的下层协议却不具备某种指定的特征(或选项)。例如，一个 SOCK_STREAM 类型的套接字本身是不能对数据进行广播的；因此，若试图设置或获取 SO_BROADCAST 选项，便会造成 WSAENOPROTOOPT 错误。

7.1.1 SOL_SOCKET 选项级别

本节介绍可根据套接字本身的特征返回信息的一些套接字选项，但这些选项并非那个套接字协议所特有的(与它无关)。

7.1.1.1　SO_ACCEPTCONN

optval 类型	获取/设置	Winsock 版本	说　明
BOOL	只能获取	1+	如为 TRUE，表明套接字处于监听模式

如果已通过 Listen 函数将套接字置入监听模式，这个选项就会返回 TRUE。SOCK_DGRAM 类型的套接字不支持这一选项。

7.1.1.2　SO_BROADCAST

optval 类型	获取/设置	Winsock 版本	说　明
BOOL	两种均可	1+	如为 TRUE，表明套接字已配置为发送广播消息

如果指定的套接字已经配置为收发广播数据，对这个套接字选项进行查询，就会返回 TRUE。随 SO_BROADCAST 一起使用 setsockopt，便可在这个套接字上启用广播通信功能。对于非 SOCK_STREAM 类型的所有套接字来说，这个选项都是有效的。

广播是一种特殊的数据发送方法，使本地子网上的所有计算机都能收到相同的数据。当然，在监听传入的广播数据的计算机上，必须进行某些形式的处理。广播通信的缺点在于，假如同时有许多进程发送广播数据，网络立刻就会拥挤不堪，造成网络性能大打折扣。要想接收一条广播消息，首先必须启用广播选项，然后使用某个数据报接收函数，例如 recvfrom 或 WSARecvfrom。另外还有一种方法，即通过调用 connect 或 WSAConnect 把套接字与广播地址连接起来，再调用 recv 或 WSARecv 来实现。对 UDP 广播来说，必须指定一个端口号，以便向它发送数据报；类似地，接收端也必须请求在这个端口上接收广播数据。下面的示例代码显示了如何通过 UDP 发送广播数据：

```
SOCKET          s;
BOOL            bBroadcast;
char            *sMsg = "This is a test";
SOCKADDR_IN     bcast;

s = WSASocket(AF_INET,SOCK_DGRAM,0,NULL,0,WSA_FLAG_OVERLAPPED);
bBroadcast = TRUE;
setsockopt(s,SOL_SOCKET,SO_BROADCAST,(char *)&bBroadcast,
   sizeof(BOOL));
bcast.sin_family = AF_INET;
bcast.sin_addr.s_addr = inet_addr(INADDR_BROADCAST);
bcast.sin_port = htons(5150);
sendto(s,sMsg,strlen(sMsg),0,(SOCKADDR *)&bcast,sizeof(bcast));
```

对 UDP 来说，存在着一个特殊的广播地址，所有广播数据均应发至该地址。这个地址便是 255.255.255.255。为 INADDR_BROADCAST 提供一个#define 指令，可使整个程序更为简单，而且更加易读。

AppleTalk 是能够发送广播消息的另一种协议。AppleTalk 也提供了一个特殊地址供广播数据专用。通过第 4 章的学习，大家已经知道 AppleTalk 地址共由 3 部分组成：网络、节点和套接字(目的地)。要进行广播通信，需将目的地设为 ATADDR_BROADCAST(0xFF)，这样便会将数据报发给指定网络内的所有端点。

通常，在发送广播数据报时，只需要设置 SO_BROADCAST 选项。要想接收一个广播数据报，只需在那个指定的端口上，对传入的数据报进行监听即可。但在 Windows 95 操作系统中，在使用 IPX 协议的时候，接收套接字必须设置 SO_BROADCAST 选项，以便接收广播数据。这一点已在“微软知识库”中编号为 Q137914 的文章里作了说明，地址是 http://support.microsoft.com/ support/search。这是 Windows 95 的一个错误。

7.1.1.3 SO_CONDITIONAL_ACCEPT

optval 类型	获取/设置	Winsock 版本	说 明
BOOL	两种均可	2+	允许用 WSAAccept 接受或拒绝真实的连接

这个选项允许应用程序在协议层有条件地接受或拒绝传入的连接。例如，在默认情况下，这个选项针对 TCP 是关闭的，任何传入的 SYN 数据包都会用 ACK+SYN 确认收到，即使应用程序没有调用 accept、WSAAccept 或 AcceptEx 函数。如果打开这个选项，连接不会被确认，除非应用程序调用其中一个接受函数。如果调用 WSAAccept 和一个条件接受函数，则只有在条件接受函数返回 CF_ACCEPT 之后，连接才会被确认。这个选项必须在 listen 调用之前设置。

打开这个选项的弊端是，如果应用程序没有及时投递一个接受操作或返回 CF_ACCEPT，则客户机连接请求将会超时，并返回错误 WSAETIMEDOUT。对于 TCP/IP，默认情况下这个选项是关闭的，而对 ATM 而言在默认情况下则是打开的。Windows 2000 及其后期版本均支持这个选项。

7.1.1.4 SO_CONNECT_TIME

optval 类型	获取/设置	Winsock 版本	说 明
BOOL	只能获取	1+	返回套接字已建立连接的时间，以秒为单位

SO_CONNECT_TIME 是一个微软特有选项，用于返回连接已建立的秒数。该选项经常和 AcceptEx 函数一道使用。AcceptEx 要求为传入的客户机连接传递一个有效的套接字句柄。该选项可在客户机的 SOCKET 句柄上调用，以判断连接是否已经建立，以及建立了多久的时间。若套接字当前尚未建立连接，返回值便是 0xFFFFFFFF。

就 AcceptEx 来说，这个选项尤其关系重大。如果应用程序使用一个接收缓冲区投递 AcceptEx，则只有在客户机连接上收到数据之后，AcceptEx 才会完成。通过制造许多连接却不发送数据，恶意的应用程序就可以实施一种服务拒绝攻击。为了避免这种情况出现，服务器应该在 AcceptEx 调用中循环通过所有未完成的客户机套接字，看看情况是否为这些套接字已被连接上，但接收还未完成。更详

细的内容请参见第 6 章。

7.1.1.5　SO_DEBUG

optval 类型	获取/设置	Winsock 版本	说　明
BOOL	两者均可	1+	如果是 TRUE，就允许调试输出

应推荐 Winsock 服务提供程序在应用程序设置了 SO_DEBUG 选项的前提下输出调试信息。至于调试信息如何展示，要取决于服务提供程序的下层实现方式。要想打开调试信息，可设置 SO_DEBUG 并将布尔变量设为 TRUE 来调用 setsockopt 函数。若用 SO_DEBUG 调用 getsockopt，会返回 TRUE 或 FALSE，分别代表允许和禁止调试。不幸的是，目前尚无任何一种 Windows 平台支持 SO_DEBUG 选项，这在“微软知识库”中编号为 Q138965 的文章里已有说明。尽管设置这个选项后，就不会返回任何错误，但下层网络提供程序会将这个选项忽略。

7.1.1.6　SO_DONTLINGER

optval 类型	获取/设置	Winsock 版本	说　明
BOOL	两者均可	1+	如果是 TRUE，则禁用 SO_LINGER

某些协议支持套接字连接的从容关闭。对这些协议来说，它们实行了一种特殊的机制，可在通信一方或双方关闭了套接字的前提下，将尚未处理的数据发送或接收完毕。设置 SO_LINGER 选项，并调用 setsockopt 函数，就有可能改变这种行为，使得经过一段规定的时间后，套接字及其所有资源都被强行清除。与套接字关联在一起的所有未处理的或未传输完毕的数据都会被丢弃，同时重设与对方的连接(WSAECONNRESET)。可以检查 SO_DONTLINGER 选项，以确保不经历这样的一段拖延时间。若用 SO_DONTLINGER 调用 getsockopt，会返回一个布尔类型的 TRUE 或 FALSE 值，分别表示设置或没有设置拖延时间值。若在设置了 SO_DONTLINGER 的前提下，调用 setsockopt，便会禁止这种拖延。要注意的是，SOCK_DGRAM 类型的套接字不支持这一选项。

7.1.1.7　SO_DONTROUTE

optval 类型	获取/设置	Winsock 版本	说　明
BOOL	两者均可	1+	如果是 TRUE，便直接向网络接口发送消息，毋需查询路由表

SO_DONTROUTE 选项用于指示位于基层的网络堆栈，让它忽略路由表的存在而通过套接字绑定的那个接口直接将数据传送出去。例如，假定我们创建了一个 IPv4 UDP 套接字，并将它与接口 A 绑定到一起，然后发送一个数据包，目的地是同接口 B 连接的另一个网络上的某台计算机。此时，若采用默认设置，该数据包会经过一个路由过程，使其能通过接口 B 传入目标网络。但将这里的布尔值设为 TRUE，再来调用 setsockopt，便可避免这样做，数据包会从绑定的接口上直接发送出去。以后可

调用 getsockopt，判断是否启用了路由。要注意的是，在默认情况下，路由是启用的。

在一个 Windows 平台上调用这个选项总会成功。但是，Microsoft 提供程序会忽略这一请求，并总是通过路由表为外出数据确定一个适当的接口。

7.1.1.8 SO_ERROR

optval 类型	获取/设置	Winsock 版本	说 明
BOOL	只能获得	1+	返回错误状态

SO_ERROR 选项用于返回和重设基于单套接字的错误代码。这些错误与那些以具体线程为基础的错误代码不同，后者是用 WSAGetLastError 和 WSASetLastError 函数调用来处理的。若使用套接字进行了一次成功的调用，则不会重设由 SO_ERROR 选项返回的、基于单套接字的错误代码。尽管调用这个选项不会失败；但是，错误值并不是总能够及时得到更新的。因此，该选项有些时候会返回 0 值(表明没有错误)。一般情况下最好使用 WSAGetLastError。

7.1.1.9 SO_EXCLUSIVEADDRUSE

optval 类型	获取/设置	Winsock 版本	说 明
BOOL	两者均可	2+	如果是 TRUE，套接字绑定的那个本地端口就不能被另一个进程重新使用

这个选项是对 SO_REUSEADDR 的一个补充，后者很快就会讲到。这个选项的作用是禁止其他进程在一个本地地址上使用 SO_REUSEADDR(应用程序正在这个地址上运行)。例如，假定两个独立的进程都与同一个本地地址绑定到了一起(假定早先已设置了 SO_REUSEADDR)，那么两个套接字中到底由哪一个负责接收传入连接的请求通知呢？这一点并未定义。SO_EXCLUSIVEADDRUSE 选项的作用便是将一个本地地址锁定在与它绑定的那个套接字上。这样，假如有其他进程试图针对相同的本地地址使用 SO_REUSEADDR，那个进程调用便会失败。若想设置该选项，必须具有管理员的身份。而且要注意的是，仅有 Windows 2000 及其后期版本才支持这个选项。

7.1.1.10 SO_KEEPALIVE

optval 类型	获取/设置	Winsock 版本	说明
BOOL	两者均可	1+	如果是 TRUE，套接字就会进行配置，并在会话过程中发送“保持活跃”消息

对一个基于 TCP 的套接字来说，应用程序可请求下层服务提供程序允许在 TCP 连接上使用保持活跃(Keepalive)数据包，这是通过打开 SO_KEEPALIVE 套接字选项来做到的。在 Windows 平台上，“保持活跃”是根据 RFC1122 文件的 4.2.3.6 小节的规定而实现的。假如“保持活跃”造成了连接的中断，便会向套接字上正在进行的任何一个调用返回一个 WSAENETRESET 错误代码。而后续的任

何调用都会失败，并返回 WSAENOTCONN 错误。要想了解更多的实现细节，请参阅 RFC 文件。在此要注意的点是，“保持活跃”信号的发送要以不短于 2 小时的时间间隔进行。2 小时的“保持活跃”时间是通过注册表配置的；但是，一旦改变了这个默认值，同时也会影响系统中所有 TCP 连接的“保持活跃”行为，而这种后果通常是应当避免的。另一个办法是实行自己的“保持活跃”策略。要注意的是，SOCK_DGRAM 类型的套接字并不支持这一选项。

与“保持活跃”设置对应的注册表键是 KeepAliveInterval 和 KeepAliveTime。这两个键均用来保存类型为 REG_DWORD 的值，单位是毫秒。其中，前一个键指定每次“保持活跃”传输的间隔时间，直至收到一个回应。后一个键用于控制 TCP 以多大的频率尝试发送“保持活跃”数据包，以查验一个理想的连接是否仍然有效。在 Windows 95、Windows 98 及 Windows Me 操作系统中，这两个键都位于下述注册表路径中：

```
\HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\VxD\MSTCP
```

而在 Windows NT 中，这两个键位于下述位置：

```
\HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\TCPIP\Parameters
```

特别要指出的是，Windows 2000 操作系统引入了一个新的 I/O 控制命令：SIO_KEEPALIVE_VALS。可用它分别针对每一个套接字，更改其“保持活跃”值以及发送的间隔时间。再也不用像原来那样，只能在整个系统的范围内进行更改。本章稍后还会详细叙述这个 I/O 控制命令。

7.1.1.11　SO_LINGER

optval 类型	获取/设置	Winsock 版本	说　明
struct linger	两者均可	1+	设置或获取当前的拖延值 linger

SO_LINGER 用于控制当未发送的数据在套接字上排队等候时，一旦执行了 closesocket 命令，那么该采取什么样的动作。若在设置了该选项的前提下，调用 getsockopt，该函数便会在一个 linger 结构中，返回当前的拖延时间设定。该结构的定义如下：

```
struct linger {
   u_short  l_onoff;
   u_short  l_linger;
}
```

若 l_onoff 是一个非零值，意味着此时允许进行拖延，而 l_linger 对应的是一段拖延时间，以秒为单位。若超出规定的时间，便不再拖延，所以尚未发送或接收的数据都会被丢弃，同时重设与对方的连接。相反，可调用 setsockopt，在丢弃任何正在排队等候的数据之前，启用拖延功能，同时指定一段时间长度。要想达到这个目的，需要在类型为 struct linger 的一个变量中指定预期的时间长度。若使用 setsockopt 时设置了拖延时间值，那么必须将结构的 l_onoff 字段设为一个非零值。若先是允许拖延，又想将其禁止，可在设置 SO_LINGER 选项后，调用 setsockopt，同时将 linger 结构的 l_onoff 字

段设为 0。此外，亦可通过 SO_DONTLINGER 选项调用 setsockopt，并为 optval 参数传递一个 TRUE 值。但要注意的是，SOCK_DGRAM 类型的套接字不支持这一选项。

调用 closesocket 函数的时候，对拖延选项的设置会直接影响到一个连接的行为。在表 7.1 中，我们对不同的行为进行了总结。

表 7.1 拖延选项

选 项	间隔时间	关闭类型	是否等待关闭?
SO_DONTLINGER	不适用	从容关闭	否
SO_LINGER	0	强行关闭	否
SO_LINGER	非零值	从容关闭	是

假如将 SO_LINGER 的间隔时间设为 0(换言之，linger 结构的成员 l_onoff 不为 0，但 l_linger 为 0)，closesocket 调用就不会进入阻塞状态，即使队列中的数据尚未发送，或者尚未收到确认。我们称这种情况为“强行关闭”，或者“异常关闭”，因为套接字虚拟通信回路会立即重设，尚未发出的所有数据都会丢失。而在回路的远程端，正在运行的任何接收调用都会失败，同时返回 WSAECONNRESET 错误。

如果在一个阻塞套接字上，设置了 SO_LINGER，且间隔超时不为 0，那么 closesocket 调用也会在一个阻塞套接字上进入阻塞或暂停状态，直到剩余的数据全部发出，或者超过规定的时间。我们将这称为”正常关闭”。若超过规定的时间，数据仍未全部发出，Windows 套接字机制使会在 closesocket 返回之前，将连接终断。

7.1.1.12 SO_MAX_MSG_SIZE

optval 类型	获取/设置	Winsock 版本	说 明
unsigned int	只能获取	2+	对于一个面向消息的套接字来说，一条消息的最大长度

这是一种只能获取数据的套接字选项，用于针对一个由特定服务提供程序实现的、面向消息的套接字类型，设置一条消息的最大外出发送长度。而对那些面向字节流的套接字来说，该设定没有任何用处。

7.1.1.13 SO_OOBINLINE

optval 类型	获取/设置	Winsock 版本	说 明
BOOL	只能获取	1+	如果是 TRUE，OOB 数据就会在普通数据流中返回

在默认情况下，OOB 数据不是内嵌传送的。也就是说，若调用一个接收函数(同时相应地设置 MSG_OOB 标志)，那么 OOB 数据也会放在一个单独的调用中返回。若设置了这个选项，OOB 数据

会出现于从一个接收调用返回的数据流中。在设置了 SIOCATMARK 选项的前提下，若调用 ioctlsocket，便可决定哪个字节属于 OOB 数据。SOCK_DGRAM 类型的套接字不支持这一选项。要想了解 OOB 数据的详情，可参阅第 1 章。

7.1.1.14　SO_OPENTYPE

optval 类型	获取/设置	Winsock 版本	说　明
BOOL	两者均可	1+	如果设置，随后对 socket 的调用将返回非重叠句柄

在默认情况下，socket API 返回重叠的句柄。然而，如果希望在 C 运行库例程中使用套接字，则这个套接字被创建时不得带重叠标志。如果用 Winsock 2，可以在不指定 WSA_FLAG_OVERLAPPED 标志时调用函数 WSASocket。否则，这个套接字选项可能会被设置，这将影响随后所有当前线程对 socket 的调用，这样将返回非重叠句柄。设置这个选项后，INVALID_SOCKET 会被当作 SOCKET 参数传递到 setsockopt，且 optval 为非零值。要改回原来的设置，指定一个零值的 optval 即可。一个创建为非重叠的套接字不能以重叠方式使用(如完成例程或完成端口)——这样的操作将会失败。

7.1.1.15　SO_PROTOCOL_INFO

optval 类型	获取/设置	Winsock 版本	说　明
BOOL	只能获取	2+	返回 Winsock 套接字协议的条目

这又是一个只能获取或者只能用来收取数据的选项，可在指定的 WSAPROTOCOL_INFO 结构中，填充与套接字关联在一起的协议的特征信息。请参考第 2 章，了解 WSAPROTOCOL_INFO 结构的详细说明，及其各个成员字段的详情。

7.1.1.16　SO_RCVBUF

optval 类型	获取/设置	Winsock 版本	说　明
int	两者均可	1+	面向接收操作，为每个套接字分别获取或设置缓冲区的长度

这是一个非常简单的选项，用于返回或设置分配给套接字的缓冲区大小，这个缓冲区用于数据的接收。创建好一个套接字后，应用程序会为它分配一个发送缓冲区和一个接收缓冲区，分别用于数据的发送与接收。在请求将接收缓冲区的大小设为一个特定的值时，即便没有充分满足这个请求，也没有提供全部要求的空间，对 setsockopt 的调用也会成功。要想确保所请求的缓冲区空间都已分配完毕，可调用 getsockopt，调查实际分配了多大的空间。目前，除 Windows CE 以外，所有 Windows 平台都能获取或设置接收缓冲区的大小。在 Windows CE 中，我们不能更改这个值——只可以获取接收缓冲区大小。

之所以要更改缓冲区的大小，一个可能的原因是需要根据应用程序的行为，对缓冲区大小进行相

应地调整。例如，若编写一段代码来接收 UDP 数据报的接收，那么通常应将接收缓冲区的大小设为数据报本身长度的几倍。而对重叠 I/O 来说，若将缓冲区的大小设为 0，那么在某些情况下，可能会在一定程度上提升性能；若这些缓冲区的长度不为 0，必须牵涉到额外的内存复制操作，以便将数据从系统缓冲区移至用户指定的缓冲区。假如没有中间缓冲区，数据就会立即被复制到用户指定的缓冲区内。而这样的的操作只有在同时存在多个未完成的接收调用时，才显得有效。假如只进行一次接收，可能会对性能造成严重影响，因为除非已指定了一个缓冲区，而且做好了接收数据的准备，否则本地系统是不会接受任何传入数据的。对于性能方面的考虑第 6 章有所叙述。

7.1.1.17 SO_RCVTIMEO

optval 类型	获取/设置	Winsock 版本	说 明
int	两者均可	1+	获取或设置与套接字上数据接收对应的超时时间值(以毫秒为单位)

SO_RCVTIMEO 选项用于设置阻塞套接字上的接收超时值。这是一个以毫秒为单位的整数，用于指出一个 Winsock 接收函数在接收数据时，最多应阻塞多长时间。如需使用 SO_RCVTIMEO 选项，并用 WSASocket 函数创建套接字，那么必须将 WSA_FLAG_OVERLAPPED 指定为 WSASocket 的 dwFlags 参数的一部分。以后对任何 Winsock 接收函数的调用(包括 recv、recvfrom、WSARecv、WSARecvFrom 等等)都只会阻塞到指定的时间长度。假如在这段时间内，没有数据抵达，调用便会失败，并返回错误 10060(WSAETIMEDOUT)。如果接收操作超时，套接字就会处于不确定状态而不应再使用。

考虑到性能方面的因素，这一选项在 Windows CE 2.1 中已被禁用。如试图设置，它会被简单地忽略，且不会返回任何错误提示。但以前的 Windows CE 版本却实现了这一选项。

7.1.1.18 SO_REUSEADDR

optval 类型	获取/设置	Winsock 版本	说 明
BOOL	两者均可	1+	如果是 TRUE，套接字就可与一个正由另一个套接字使用的地址绑定到一起，或与处于 TIME_WAIT 状态的地址绑定到一起

在默认情况下，套接字不能同一个正在被其他套接字使用的本地地址绑定到一起。但在少数情况下，仍有必要以这种方式，来实现对某个地址的重复利用。每个连接都是通过它的本地及远程地址的组合来惟一标识的。针对我们想要连接的地址，只要能用细微的差异(例如 TCP/IP 中采用不同的端口号)，来维持这种惟一的特点，绑定便是允许的。

惟一例外的是监听套接字。两个独立的套接字不可与同一个本地接口(在 TCP/IP 的情况下，则是端口)绑定到一起来等待传入的连接通知。假定两个套接字都在同一个端口上进行监听，那么到底由

哪一个来接收传入的连接通知呢？这一点还没有作定义。在 TCP 的环境下，假如服务器关闭，或异常退出，造成本地地址和端口均进入 TIME_WAIT 状态，那么 SO_REUSEADDR 这个套接字选项便显得非常有用。因为在 TIME_WAIT 状态下，其他任何套接字都不能与那个端口绑定到一起。假若设置了该选项，服务器便可在重新启动之后，在相同的本地接口及端口上进行监听。

7.1.1.19　SO_SNDBUF

optval 类型	获取/设置	Winsock 版本	说　明
BOOL	两者均可	1+	如果是 TRUE(非零值)，意味着套接字被配置成可进行广播消息的发送

这也是一个非常简单的选项，要么返回、要么设置分配给套接字的用于数据发送的缓冲区的大小。创建好套接字后，要为它分配一个发送缓冲区和一个接收缓冲区，分别用于数据的发送及接收。若请求将发送缓冲区的大小设为一个特定的值，那么即使没有充分满足这个请求(没有提供要求的全部空间)，对 setsockopt 的调用也会成功。但是，如果想确认请求的缓冲区空间都已正确分配，可调用 getsockopt，调查目前实际分配了多大的空间。目前，除 Windows CE 以外，所有 Windows 平台都能获取或设置发送缓冲区的大小。在 Windows CE 中，我们不能更改这个值——只能获取接收缓冲区的大小。

和 SO_RCVBUF 一样，可用 SO_SNDBUF 选项将发送缓冲区的大小设为 0。对于阻塞发送调用来说，将缓冲区的大小设为 0 后，有个好处在于：一旦调用完成，便可肯定自己的数据正在传输途中。此外，和在接收操作中将缓冲区的长度设为 0 一样，不会涉及数据在系统缓冲区进行的额外的内存复制操作。但是，假如发送缓冲区的长度不为 0，那么通过默认的堆栈缓冲机制，可充分发挥数据流水线的优势。将这个长度变成 0 后，会失去这一优势，所以这是它的一个缺点。换言之，在默认情况下，假如要连续不停地执行数据的发送操作，那么本地网络堆栈可将我们的数据复制到一个系统缓冲区内，以便在某个恰当的时间发送出去(取决于当时采用的 I/O 模型)。而另一方面，假如应用程序需要其他方面的逻辑单元，那么将发送缓冲区屏蔽后，也可省下进行内存复制时的一些计算机指令的开销。更多内容请参见第 6 章。

7.1.1.20　SO_SNDTIMEO

optval 类型	获取/设置	Winsock 版本	说　明
int	两者均可	1+	获取或设置套接字上的数据发送的超时时间(以毫秒为单位)

在一个阻塞套接字上调用 Winsock 发送函数时，SO_SNDTIMEO 选项用于设定一个超时值。这是一个以毫秒为单位的整数值，设定发送函数在试图发送事件时，最多阻塞多久。假如需要使用 SO_SNDTIMEO 选项，并用 WSASocket 函数创建套接字，那么必须将 WSA_FLAG_OVERLAPPED

设为 WSASocket 的 dwFlags 参数的一部分。以后对任何 Winsock 发送函数的调用(send、sendto、WSASend、WSASendTo 等等)都只会阻塞到指定数量的时间。如发送操作在那段时间内不能完成，调用便会失败，并返回错误 10060(WSAETIMEDOUT)。如果发送操作超时，套接字就会处于不确定状态，而不应再使用它。

考虑到性能方面的原因，这个选项在 Windows CE 2.1 中已被禁止。如试图设置这个选项，那么它会被简单地忽略，而且不会返回任何错误提示。但以前版本的 Windows CE 却实现了这一选项。

7.1.1.21 SO_TYPE

optval 类型	获取/设置	Winsock 版本	说 明
int	只能获取	1+	返回指定套接字的类型(如 SOCK_DGRAM 和 SOCK_STREAM 等等)

SO_TYPE 选项是一个只能用来获取数据的选项，用于返回指定套接字的类型。可能的套接字类型包括 SOCK_DGRAM、SOCK_STREAM、SOCK_SEQPACKET、SOCK_RDM 和 SOCK_RAW 等等。

7.1.1.22 SO_UPDATE_ACCEPT_CONTEXT

optval 类型	获取/设置	Winsock 版本	说 明
SOCKET	两者均可	1+	用监听套接字的属性来更新客户机套接字上相应的属性

该选项属于 Microsoft 特有的一项扩展，经常与 AcceptEx 函数联合使用。该函数最显著的一个特点在于，它属于 Winsock 1 规范的一部分，允许通过重叠 I/O 操作进行数据接收调用。该函数将监听套接字设为自己的一个参数，同时还要指定一个相应的套接字句柄，这个句柄会成为一个被接受的客户机。为使监听套接字的特征能完整转移至客户机套接字，必须设置这个套接字选项。对于提供 QOS 功能的监听套接字来说，这一点尤为重要。要想使客户机套接字具有保障 QOS 的功能，必须设置这个选项。在一个套接字上设置该选项时，可将监听套接字作为 setsockopt 的 SOCKET 参数，并以 optval 的形式，传递接受套接字句柄(例如客户机句柄)。要注意的是，该选项是 Windows 操作系统特有的。

7.1.2 SOL_APPLETALK 选项级别

下述选项均为 AppleTalk 协议之专用套接字选项，只能用于通过 socket 或 WSASocket 函数创建的套接字(同时设置 AF_APPLETALK 标志)。这里列出的大多数选项都与 AppleTalk 名称的设置或获取有关。欲了解 AppleTalk 地址族更详细的信息，可参考第 4 章的内容。某些 AppleTalk 套接字选项(例如 SO_DEREGISTER_NAME)提供了不止一个选项名。在这种情况下，所有选项的名称均可互换使用。

7.1.2.1　SO_CONFIRM_NAME

optval 类型	获取/设置	Winsock 版本	说　明
WSH_NBP_TUPLE	只能获取	1	确认指定的 AppleTalk 名称和指定地址绑定在一起

SO_CONFIRM_NAME 选项用于检验一个指定的 AppleTalk 名称是否已和指定的地址绑定到了起。这样应用程序将向目标地址发送一个 NBP 检索名称绑定协议请求，以校验指定的名称。若此次调用以失败告终，并返回了一个 WSAEADDRNOTAVAIL 错误，便表明名称尚未与指定的地址绑定到一起。

7.1.2.2　SO_DEREGISTER_NAME，SO_REMOVE_NAME

optval 类型	获取/设置	Winsock 版本	说　明
WSH_REGISTER_NAME	只能设置	1	从网络中撤消对指定名称的注册

该选项用于将从网络中撤消对一个名称的注册。若指定的名称目前并未存在于网络中，调用返回后，仍会表明操作成功。请参阅第 4 章，了解 WSH_REGISTER_NAME 结构的详情，它实际是 WSH_NBP_NAME 结构的另一个名称。

7.1.2.3　SO_LOOKUP_MYZONE，SO_GETMYZONE

optval 类型	获取/设置	Winsock 版本	说　明
char*	只能获取	1	返回网络上的默认区域

该选项可返回网络上的默认区域。getsockopt 调用的 optval 参数应当是一个字符串，长度至少 33 个字符。要注意的是，NBP 名称的最大长度是由 MAX_ENTITY_LEN 规定的，后者定义成 32。若实际的名称长度不够，便需加上额外的空字符。

7.1.2.4　SO_LOOKUP_NAME

optval 类型	获取/设置	Winsock 版本	说　明
WSH_LOOKUP_NAME	只能获取	1	查找指定的 NBP 名称，并返回相符的名称及 NBP 信息字元组

该选项用于在网络上查找一个指定的名称(例如，当一个客户机想与服务器建立连接时)。连接建立之前，那些字面上易于理解的英语名称必须解析成具体的 AppleTalk 地址。大家可参考第 4 章的“解析 AppleTalk 名称”一节，那里提供了一段示范代码，可直接用来进行 AppleTalk 名称的解析。

在此要注意的一件事情是，一旦成功返回，WSH_NBP_TUPLE 结构便会占据指定缓冲区中 WSH_LOOKUP_NAME 信息之后的空间。换言之，当初调用 getsockopt 函数的时候，便应该提供一个足够大的缓冲区，以便在缓冲区的开头，完全装下 WSH_LOOKUP_NAME 信息，并在剩下的空间里，

装下大量的 WSH_NBP_TUPLE 结构。图 7.1 向大家展示了在调用之前，如何准备好这个缓冲区(首先容纳的是 WSH_LOOKUP_NAME 结构)，以及在返回后，WSH_NBP_TUPLE 结构放在什么位置。

图 7 1 SO_LOKUP_NAME 缓冲区

7.1.2.5 SO_LOOKUP_ZONES，SO_GETZONELIST

optval 类型	获取/设置	Winsock 版本	说 明
WSH_LOOKUP_ZONES	只能获取	1	返回来自 Internet 区域列表的区名

该选项要求提供一个足够大的缓冲区，以使在头部位置容下一个 WSH_LOOKUP_ZONES 结构。成功返回之后，在 WSH_LOOKUP_ZONES 结构之后的空间里，便会包含一系列以空字符结束的区域名称列表。下述代码向大家说明了如何利用这个 SO_LOOKUP_ZONES 选项：

```
PWSH_LOOKUP_NAME      atlookup;
PWSH_LOOKUP_ZONES     zonelookup;
char                      cLookupBuffer[4096],
                          *pTupleBuffer = NULL;

atlookup = (PWSH_LOOKUP_NAME)cLookupBuffer;
zonelookup = (PWSH_LOOKUP_ZONES)cLookupBuffer;
ret = getsockopt(s,SOL_APPLETALK,SO_LOOKUP_ZONES,(char *)atlookup,
   &dwSize);
pTupleBuffer = (char *)cLookupBuffer +sizeof(WSH_LOOKUP_ZONES);
for(i = 0;i <zonelookup->NoZones;i++)
{
    printf("%3d:'%s'\n",i +1,pTupleBuffer);
    while (*pTupleBuffer++);
}
```

7.1.2.6 SO_LOOKUP_ZONES_ON_ADAPTER，SO_GETLOCALZONES

optval 类型	获取/设置	Winsock 版本	说明
WSH_LOOKUP_ZONES	只能获取	1	为指定的适配器名返回一个区名列表

该选项类似于 SO_LOOKUP_ZONES，只是要求先指定一个适配器名，然后才以取得与这个适配器连接的网络对应的本地区名列表。同样地，在此必须提供一个足够大的缓冲区，在其头部装下一个 WSH_LOOKUP_ZONES 结构。而返回的以空字符结束的区名的列表接着 WSH_LOOKUP_ZONES 结构之后开始存放。此外，适配器的名称必须以 UNICODE 字符串(WCHAR)的形式传递。

7.1.2.7　SO_LOOKUP_NETDEF_ON_ADAPTER，SO_GETNETINFO

optval 类型	获取/设置	Winsock 版本	说　明
WSH_LOOKUP_NETDEF_ON_ADAPTER	只能设置	1	为指定网络返回种子值和默认区域

该选项可为指定的网络编号返回其种子值；同时返回一个以空字符结束的 ANSI 字符串，其中包含了与指定适配器连接的那个网络的默认区域。适配器是在结构之后，以一个 UNICODE(WCHAR)字符串的形式传递的，且会被函数返回的默认区域覆盖。假如网络并未进行“种子”设定，那么返回的是一个网络地址范围 1~0xFFFE，同时在以空字符结束的 ANSI 字符串中包含默认区域“*”。

7.1.2.8　SO_PAP_GET_SERVER_STATUS

optval 类型	获取/设置	Winsock 版本	说　明
WSH_PAP_GET_SERVER_STATUS	只能获取	1	Ww 指定服务器返回PAP状态

该选项可获取在 ServerAddr 指定的地址上注册的 PAP(Printer Access Protocol，打印机访问协议)状态。而那个地址通常是通过一次 NBP 检索操作获得的。该选项的 4 个保留字节分别对应于 PAP 状态包中的 4 个保留字节，均按网络字节顺序排列。而 PAP 状态字符串可以是任意的，它通过 SO_PAP_SET_SERVER_STATUS 选项进行设置；本章后面还会对详情进行解释。下面是 WSH_PAP_GET_SERVER_STATUS 结构的定义：

```
#define  MAX_PAP_STATUS_SIZE        255
#define  PAP_UNUSED_STATUS_BYTES    4

typedef struct _WSH_PAP_GET_SERVER_STATUS
{
    SOCKADDR_AT          ServerAddr;
    UCHAR                Reserved[PAP_UNUSED_STATUS_BYTES];
    UCHAR                ServerStatus[MAX_PAP_STATUS_SIZE +1];
} WSH_PAP_GET_SERVER_STATUS, *PWSH_PAP_GET_SERVER_STATUS;
```

下述代码段向大家展示了如何对 PAP 状态进行请求。状态字符串的长度是由 ServerStatus 字段的第 1 个字节指定的：

```
WSH_PAP_GET_SERVER_STATUS status;
Int                                nSize = sizeof(status);
status.ServerAddr.sat_family = AF_APPLETALK;
ret = getsockopt(s,SOL_APPLETALK,SO_PAP_GET_SERVER_STATUS,
    (char *)&status,&nSize);
```

7.1.2.9 SO_PAP_PRIME_READ

optval 类型	获取/设置	Winsock 版本	说 明
char[]	只能设置	1	这个调用会在一个 PAP 连接上发起一次读取操作，以使发送者能实际地发出数据

若在设置成 PAP 连接的一个套接字上调用这个选项，远程客户机便会在本地应用程序尚未调用 recv 或 WSARecvEx 的前提下，开始数据的发送。设置了该选项后，应用程序会在一次 select 调用中进入阻塞状态，然后进行实际的数据读取操作。该调用的 optval 参数指定的是一个缓冲区，用于数据的接收。该缓冲区的长度至少应为 MIN_PAP_READ_BUF_SIZE(4 096)个字节。通过这一选项，可在“出读驱动”的 PAP 协议上，实现对非阻塞套接字的支持。但要注意的是，对要读取的每个缓冲区，都必须在设置了 SO_PAP_PRIME_READ 选项的前提下，执行一次 setsockopt 调用。

7.1.2.10 SO_PAP_SET_SERVER_STATUS

optval 类型	获取/设置	Winsock 版本	说 明
char[]	只能设置	1	假如另一个客户机请求 PAP 状态，则设置要发送出去的状态

客户机可通过 SO_PAP_GET_SERVER_STATUS，请求获取 PAP 状态。可用该选项对状态进行设置，以便在客户机请求 PAP 状态时，向设置命令提交的缓冲区能在获取命令中返回。这里的“状态”实际是一个不超过 255 字节的缓冲区，其中包含了对应的那个套接字的状态信息。若在提供一个空缓冲区的前提下调用设置选项，则以前的状态值将会被清除掉。

7.1.2.11 SO_REGISTER_NAME

optval 类型	获取/设置	Winsock 版本	说 明
WSH_REGISTER_NAME	只能设置	1	在 AppleTalk 网络上注册指定的名称

该选项用于在 AppleTalk 网络中注册一个指定的名称。若那个名称已在网络中存在，便会返回 WSAEADDRINUSE 错误。大家可参考第 4 章，那里提供了对 WSH_REGISTER_NAME 结构的详细说明。

7.1.3 SOL_IRLMP 选项级别

SOL_IRLMP 级别与 IrDA 协议有着密切的联系，它的地址族为 AF_IRDA。使用 IrDA 套接字选项时，要记住的一个要点在于，在不同平台上，红外线套接字的具体实现方式是有所区别的。由于 Windows CE 是最早支持红外线通信的一个平台，所以没有包括后来在 Windows 98 及其后期版本中引入的一些新选项。本节除了对每个选项的含义进行解释外，也指明了它适用的平台。

7.1.3.1　IRLMP_9WIRE_MODE

optval 类型	获取/设置	Winsock 版本	说　明
BOOL	两者均可	1+	将 IrDA 套接字置入 IrCOMM 模式

这是另一个很少用的选项，用于通过 IrCOMM(红外线通信端口)建立与 Windows 98 的通信。它的操作级别要比 IrSock 正常的级别稍低一些。在 9 线模式下，每个 TinyTP 或 IrLMP 包都包含了一个附加的 1 字节的 IrCOMM 头。要想通过套接字接口做到这一点，首先需要使用 IRLMP_SEND_PDU_LEN 选项，取得 IrLMP 包的最大 PDU(Protocol Data Unit，协议数据单元)长度。随后，在建立连接或接受一个连接请求之前，使用 setsockopt 函数将套接字置入 9 线通信模式。这样便可告诉堆栈为外出的每个数据帧都添加一个 1 字节的 IrCOMM 头(总是将它设为 0)。每个 send 的长度都必须小于允许的最大 PDU 长度，以便为新增的 IrCOMM 字节腾出空间。IrCOMM 更深入的一些技术细节已超出了本书的范围，所以在此不再赘述。该选项仅适用于 Windows 98 及其后期版本。

7.1.3.2　IRLMP_ENUMDEVICES

optval 类型	获取/设置	Winsock 版本	说　明
DEVICELIST	只能获取	1+	为范围内具有红外线通信能力的设备返回一个 IrDA 设备的 ID 列表

由红外线通信的本质使然，具有这种通信能力的设备都是可移动的。换言之，它们可能移至有效通信范围之外。这个选项的作用便是对有效范围之内的 IR 设备进行查询。要想建立与另一个设备的连接，首先必须执行这一步操作，以取得自己想连接的每一个设备的设备 ID。

在支持 IrSock 这种套接字的各种不同的平台上，DEVIDELIST 结构也有所差异。这是由于新问世的一些平台除增加了这方面的支持以外，也引入了一些新的功能。前面已经说过，最早提供红外线通信能力的是 Windows CE，结果导致数据结构有些不同。下面是在 Windows 98 及其后期版本操作系统中，对 DEVIDELIST 结构的定义：

```
typedef struct _WINDOWS_DEVICELIST
{
   ULONG numDevice;
   WINDOWS_IRDA_DEVICE_INFO Device[1];
}WINDOWS_DEVICELIST,*PWINDOWS_DEVICELIST,FAR *LPWINDOWS_DEVICELIST;

typedef struct _WINDOWS_IRDA_DEVICE_INFO
{
   u_char  irdaDeviceID[4];
   char    irdaDeviceName[22];
   u_char  irdaDeviceHints1;
   u_char  irdaDeviceHints2;
```

```
   u_char  irdaCharSet;
}WINDOWS_IRDA_DEVICE_INFO,*PWINDOWS_IRDA_DEVICE_INFO,
FAR *LPWINDOWS_IRDA_DEVICE_INFO;
```

而在 Windows CE 中，DEVIDELIST 结构是这样定义的：

```
typedef struct _WCE_DEVICELIST
{
   ULONG numDevice;
   WCE_IRDA_DEVICE_INFO Device[1];
}WCE_DEVICELIST,*PWCE_DEVICELIST;

typedef struct _WCE_IRDA_DEVICE_INFO
{
   u_char irdaDeviceID[4];
   char irdaDeviceName[22];
   u_char Reserved[2];
}WCE_IRDA_DEVICE_INFO,*PWCE_IRDA_DEVICE_INFO;
```

每个结构都包含了一个名为 irdaDeviceID 的字段。这是一个长度为 4 字节的标志，用于惟一性地标识某个设备。我们需要用这个字段来填充 SOCKADDR_IRDA 结构，这个结构用于连接特定设备，或者通过 IRLMP_IAS_SET 及 IRLMP_IAS_QUERY 选项来处理或获取一个 IAS 条目。

在调用 getsockopt 函数，对红外线设备进行列举的时候，optval 参数必须是一个 DEVICELIST 结构。此时惟一的要求是在最开始的时候，将 numDevice 字段设为 0。假如没有发现任何红外线(IR)设备，对 getsockopt 的调用也不会返回任何错误。完成了一次调用后，应对 numbDevice 字段进行检查，看看它是否大于 0。若答案是肯定的，便意味着已找到了一个或多个红外线设备。Device 字段可返回与 numDevice 的值相等数量的大量结构。

7.1.3.3 IRLMP_EXECLUSIVE_MODE

optval 类型	获取/设置	Winsock 版本	说　明
BOOL	两者均可	1+	如果是 TRUE，表明套接字连接处于独占模式

因为该选项绕过了 IrDA 堆栈中的 TinyTP 层，而直接通过 IrLMP 进行通信，所以它通常并不在应用程序中直接使用。如果真的对使用这个选项有兴趣，请查阅位于 http://www.irda.org 的 IrDA 规范文件。该选项适用于 Windows CE 和 Windows 2000 或后期版本。

7.1.3.4 IRLMP_IAS_QUERY

optval 类型	获取/设置	Winsock 版本	说明
IAS_QUERY	只能获取	1+	在指定服务上查询 IAS，并查询与其属性对应的类名

这个套接字选项是对 IRLMP_IAS_SET 的一个补充，用于取得与一个类名及其服务有关的信息。

在发出对 getsockopt 的调用之前，首先必须填充好 irdaDeviceID 字段，将其指定为自己准备查询的那个设备的设备 ID。至于 irdaAttribName 字段，应设为想从中取得具体值的一个属性字符串。最常见的查询是针对 LSAP-SEL 号码进行的；它的属性字符串是“IrDA:IrLMP:LsapSel”。接下来，需要将 irdaClassName 字段设为将应用指定属性字符串的那个服务的名称。一旦填写好这些字段的内容，使可发出对 getsockopt 的调用。若成功返回，irdaAttribType 字段会指明从共用体中的哪个字段可以取得信息。可根据表 7.2 总结的标识符，对这个条目的含义进行诠释。最常见的返回错误是 WSASERVICE_NOT_FOUND。若在指定设备上没有找到相应的服务，便会返回这个错误。

7.1.3.5　IRLMP_IAS_SET

optval 类型	获取/设置	Winsock 版本	说　明
IAS_QUERY	只能设置	1+	为指定的类名和属性设置一个属性值

IAS 其实是一种动态服务注册实体，可对其进行查询和修改。利用 IRLMP_IAS_SET 选项，我们可在本地 IAS 内，为单个类设置一条属性。和 IRLMP_ENUMDEVICES 的情况一样，Windows CE 采用的是一种结构，而 Windows 98 及其后期版本采用的又是另一种结构。下面是 Windows 98 及其后期版本的结构定义：

```
typedef struct _WINDOWS_IAS_QUERY
{
   u_char irdaDeviceID[4];
   char   irdaClassName[IAS_MAX_CLASSNAME];
   char   irdaAttribName[IAS_MAX_ATTRIBNAME];
   u_long irdaAttribType;
   union
   {
      LONG irdaAttribInt;
      struct
      {
         u_long Len;
         u_char OctetSeq[IAS_MAX_OCTET_STRING];
      }irdaAttribOctetSeq;
      struct
      {
         u_long Len;
         u_long CharSet;
         u_char UsrStr[IAS_MAX_USER_STRING];
      }irdaAttribUsrStr;
   }irdaAttribute;
} WINDOWS_IAS_QUERY, *PWINDOWS_IAS_QUERY, FAR *LPWINDOWS_IAS_QUERY;
```

Windows CE 采用的 IAS 查询结构如下：

```
typedef struct _WCE_IAS_QUERY
{
   u_char     irdaDeviceID[4];
   char       irdaClassName[61];
   char       irdaAttribName[61];
   u_short    irdaAttribType;
   union
   {
      int irdaAttribInt;
      struct
      {
         int Len;
         u_char OctetSeq[1];
         u_char Reserved[3];
      }irdaAttribOctetSeq;
      struct
      {
         int     Len;
         u_char CharSet;
         u_char UsrStr[1];
         u_char Reserved[2];
      }irdaAttribUsrStr;
   }irdaAttribute;
}WCE_IAS_QUERY,*PWCE_IAS_QUERY;
```

在表 7.2 中，总结了 irdaAttribType 字段可采用的各种不同的常数，它指出了属性具体从属于哪种类型。最后两个条目不能由我们自行设置，而是通过对 getsockopt 的调用，同时设置一个 IRLMP_IAS_QUERY 套接字选项，在 irdaAttribType 字段中所返回的值。之所以也在表格中列出，只是为了完整起见。

表 7.2 IAS 属性类型

irdaAttribType 的可选值	要设置的字段
IAS_ATTRIB_INT	irdaAttribInt
IAS_ATTRIB_OCTETSEQ	irdaAttribOctetSeq
IAS_ATTRIB_STR	irdaAttribUsrStr
IAS_ATTRIB_NO_CLASS	无
IAS_ATTRIB_NO_ATTRIB	无

要想设置一个值，首先必须将 irdaDevicdID 设为目标红外线通信设备。在这个设备上，我们希望对其 IAS 条目进行修改。同样地，irdaAttribName 必须设为要在上面设置属性的那个类。而

irdaClassName 通常对应的是一种服务，我们要在上面设置属性。在此要记住的是，对 IrSock 类型的套接字来说，套接字服务器是随 IAS 注册的一种服务。客户机使用与之关联的一个 LSAP-SEL 编号，建立与服务器的连接。LSAP-SEL 编号是与这种服务关联在一起的属性。要想在服务的 IAS 条目中修改 LSAP-SEL 编号，请将 irdaDeviceID 字段设为服务正在上面运行的那个设备的 ID。irdaAttribName 字段应设为属性字符串“IrDA:IrLMP:LsapSel”，而 irdaClassName 字段应设为服务的名称(例如“MySocketServer”)。随后，将 irdaAttribType 设为 IAS_ATTRIB_INT，并将 irdaAttribInt 设为新的 LSAP-SEL 编号。当然，更改服务的 LSAP-SEL 编号并不是一种推荐的做法，本例只是为了阐述的方便。

7.1.3.6　IRLMP_IRLPT_MODE

optval 类型	获取/设置	Winsock 版本	说　明
BOOL	两者均可	1+	若为 TRUE，套接字就会配置成与支持 IR 的打印机通信

通过 Winsock，可与一台具有红外线功能的打印机进行通信，向其发送需要打印的数据。要做到这一点，在正式建立连接之前，首先要将套接字置入 IRLPT 模式。创建好套接字后，只需向这个选项传递一个布尔值 TRUE 即可。可利用 IRLMP_ENUMDEVICES 选项来寻找有效通信范围之内的红外线打印机。要注意的是，有些老式打印机不能向 IAS 注册自己的存在。此时，需要通过“LSAP-SEL-xxx”标识符，建立与它们的直接连接。大家可参考第 4 章对 IrSock 的讨论，了解具体如何绕过 IAS。该选项同时适用于 Windows CE、Windows 2000 及其后期版本。

7.1.3.7　IRLMP_SEND_PDU_LEN

optval 类型	获取/设置	Winsock 版本	说　明
int	只能获取	1+	取得最大的 PDU 长度

使用 IRLMP_9WIRE_MODE(9 线模式)选项时，可通过该选项查询最大的 PDU 长度。大家可参考对 IRLMP_9WIRE_MODE 的说明，了解这个选项的详情，它适用于 Windows CE、Windows 2000 及其后期版本。

7.1.4　IPPROTO_IP 选项级别

IPPROTO_IP 这一级的套接字选项与 IPv4 协议的特有属性存在密切联系，例如可用它们修改 IPv4 头内的特定字段，以及向 IPv4 多播组添加套接字等等。许多这样的选项在 Winsock.h 和 Winsock2.h 这两个头文件内都有声明，而且采用了不同的值。要注意的是，假如加载的是 Winsock 1，那么必须包含正确的头文件，同时建立与 WSOCK32.LIB 函数库的链接关系。若加载的是 Winsock 2，情况也是类似的，除了将 Winsock 2 的头文件包含进来外，还要建立与 WS2_32.LIB 的链接关系。因为两个

版本的 Winsock 均支持多播通信，所以在多播通信环境中，这一点尤其重要。除 Windows CE 的早期版本之外，其他所有 Windows 平台都提供了对多播通信的支持。而 Windows CE 自 2.1 版之后，也开始提供对这方面的支持。和 IGMPv3 相关的新多播选项是在 WS2TCPIP.H 中定义的。

7.1.4.1 IP_OPTIONS

Optval 类型	获取/设置	Winsock 版本	说　明
char[]	两者均可	1+	设置或获取 IP 头内的 IP 选项

通过该标志，可在 IP 头内设置各种 IP 选项。某些可能出现的选项包括：

- 安全和处理限制：RFC1108
- 记录路由：每个路由器都将自己的 IP 地址添加到头内(参见第 11 章的 Ping 程序示例)
- 时标(时间戳)：每个路由器都添加自己的 IPv4 地址及时间
- 松散源路由选择：数据包被要求去访问选项头内列出的每个 IPv4 地址
- 严格源路由选择：数据包被要求去访问仅在选项头内列出的那些 IPv4 地址

要注意的是，主机和路由器并不支持所有这些选项。

设置一个 IPv4 选项时，传至 setsockopt 调用内部的数据会遵循如图 7.2 所示的结构。IPv4 选项头最长可达 40 字节。

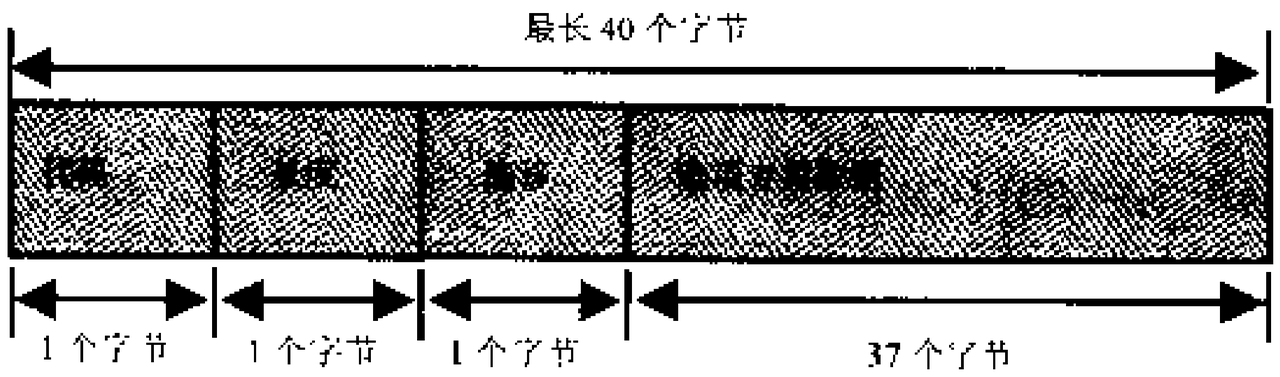


图 7.2　IP 选项头格式

其中，“代码”字段指出要提供的是什么类型的 IP 选项。例如，若将该值设为 0x7，便意味着是“记录路由”选项。而“长度”字段对应着选项头的长度，“偏移”是指头内数据部分的起始偏移位置。头的数据部分是由特定的选项来决定的。在下述代码段中，我们设置的是一个记录路由选项。请注意，我们首先声明了一个结构(struct ip_option_hdr)，它包含头 3 个选项值(代码、长度和偏移)。随后，我们将由具体选项决定的数据声明成一个数组，因为要记录的数据最多允许包含 9 个 IPv4 地址，所以数组由 9 个无符号长整数构成。在此要注意的是，IPv4 选项头允许的最大长度是 40 字节；然而，我们的结构只占用了 39 字节。系统会自动对这个头进行填充，将其长度保持为一个 32 位字的整数倍(最多 40 字节)。

```
struct ip_option_hdr
{
   unsigned  char  code;
```

```
    unsigned  char  length;
    unsigned  char  offset;
    unsigned  long  addrs[9];
}opthdr;
...
ZeroMemory((char *)&opthdr,sizeof(opthdr));
opthdr.code = 0x7;
opthdr.length = 39;
opthdr.offset = 4;//首地址(addrs)的偏移
ret = setsockopt(s,IPPROTO_IP,IP_OPTIONS,(char *)&opthdr,
      sizeof(opthdr));
```

选项设好之后，它便可应用于在指定套接字上发出的所有数据包。以后，我们可随 IP_OPTIONS 一道，调用 getsockopt 函数，以取得已设置了的选项。然而，这样做并不会返回填充到选项专用缓冲区内的任何数据。为取得在 IP 选项中设置的数据，要么将套接字创建成一个原始套接字(SOCK_RAW)，要么设置 IP_HDRINCL 选项。在后一种情况下，IP 头会在一个 Winsock 获取函数的调用之后，随数据一道返回。

7.1.4.2　IP_HDRINCL

optval 类型	获取/设置	Winsock 版本	说　明
BOOL	两者均可	2+	如果是 TRUE，IP 头就会随即将发送的数据一起提交到 Winsock 调用

若将 IP_HDRINCL 选项设为 TRUE，发送函数会将 IPv4 头包含在自己发送的数据前面，因此，在我们调用一个 Winsock 发送函数的时候，必须在数据前面包含完整的 IPv4 头，并对 IP 头的每个字段进行正确的填写。应注意到，设置这个选项之后，IPv4 网络堆栈将会在必要的时候对数据包的数据部分进行分段。这个选项仅对 SOCK_RAW 类型的套接字有效。在图 7.3 中，我们向大家展示了 IP 头实际看起来的样子。该选项仅适用于 Windows 2000 及其后期版本的操作系统。

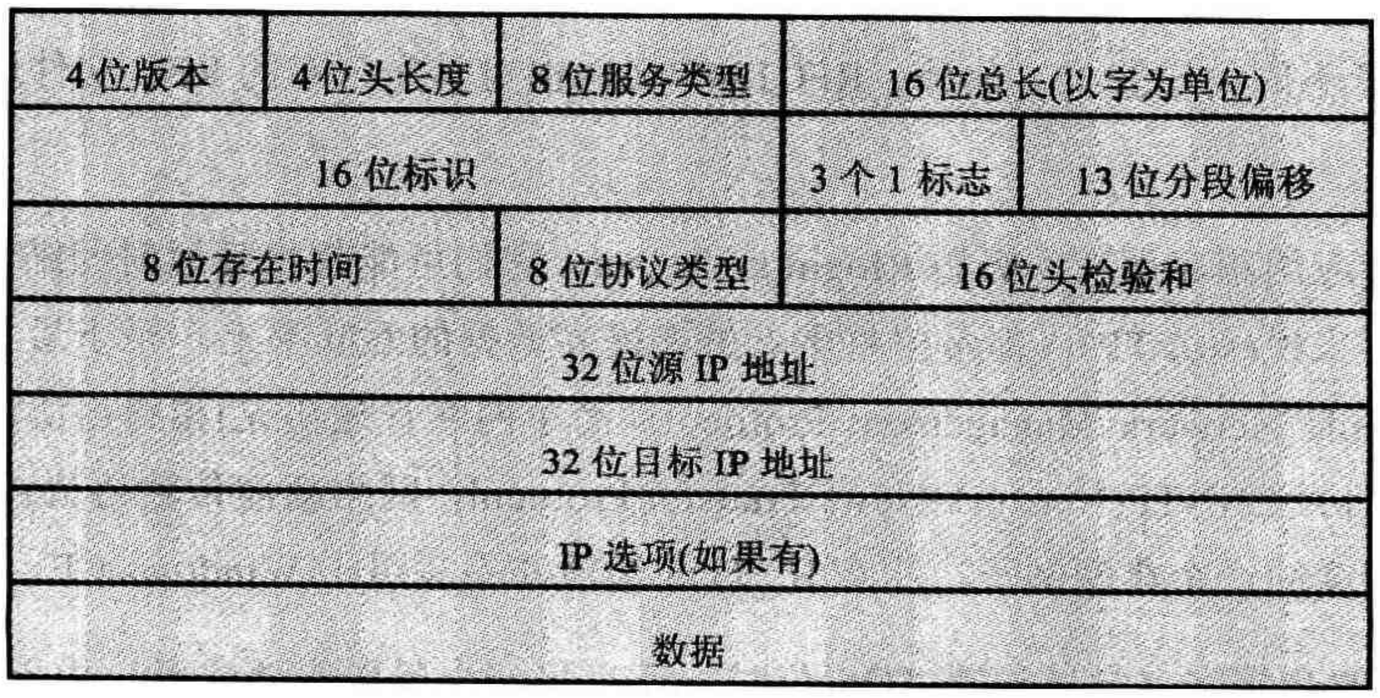


图 7.3　IPv4 头

其中，头的第 1 个字段指定的是 IPv4 版本，即 4。头长度是指在整个头中，32 位字一共有多少个(一个头的长度必须是 32 位的整数倍)。接下来的字段是服务的类型。对此，大家可参考下一节介绍的 IP_TOS 套接字选项，了解进一步的情况。总长字段是指 IP 头和数据加在一起，一共有多长(以字节计)。标识字段是一个特殊的值，用于对发出的每个 IPv4 包进行惟一性的标识。通常，每发出一个数据包，系统都会递增这个值。标志和分段偏移字段在 IPv4 包需要分割为较小的包时会用到。而 TTL 字段则用于限制数据包最多可通过多少个路由器。每遇到一个路由器对这个包进行转发的时候，TTL 的值都会递减 1。一旦 TTL 变成 0，便会将这个包丢弃。这样，便可限制一个包能在网络上生存的时间，避免它无休止地游荡下去。协议字段用于组装传入的数据包。采用了 IP 寻址机制的某些有效协议包括 TCP、UDP、IGMP 和 ICMP 等等。校验和是指对整个 IP 头进行 16 位 1 的求补的总和。要注意的是，这种计算仅针对头进行，不针对实际的数据。接下来的两个字段分别是 32 位的 IP 源和目标地址。IPv4 选项字段是一个长度不定的字段，包含了某些可选的信息，通常与安全性或路由选择有关。

要想将一个 IPv4 头包含在打算发送的数据中，最简单的办法便是定义一个结构，在其中包含 IP 头以及数据，再将整个结构传递给 Winsock 的 send 调用。大家可参考第 11 章的内容，那里除详细讲述了这方面的知识之外，还提供了这个选项的一个例子。要注意的是，该选项仅适用于 Windows 2000 及其后期版本。

7.1.4.3 IP_TOS

optval 类型	获取/设置	Winsock 版本	说　明
int	两者均可	1+	Ipv1 服务类型

TOS(type of service，服务类型)是在 IPv4 头中设置的一个字段，用于指出与一个包对应的具体特征。该字段总长度为 8 位，总共划分为 3 部分：一个 3 位长度的优先级字段(这方面的设置会被忽略)，一个 4 位长度的 TOS 字段，以及一个补足位(必须设为 0)。其中，4 个 TOS 位分别对应于最短延迟、最高数据吞吐率、最大可靠性以及最小费用。但要注意的是，每一次只能设置一个位。若这 4 个位的值均为 0，便暗示着采用普通服务。在 RFC1340 文件中，针对各种标准应用(包括 TCP、SMTP、NNTP 等等)，分别推荐了不同的设置位。此外，RFC1349 文件也对早先公布的 RFC 文件进行了一些补充及纠正。

对某些交互式应用程序而言，例如 Rlogin 或 Telnet，它们可能想将延迟时间缩至最短。而对任何一种文件传输机制来说(如 FTP)，都希望将重点放在提高数据的吞吐率上。至于最大可靠性，则是网络管理 SNMP(Simple Network Management Protocol，简单网络管理协议)和路由协议所强烈要求的。最后，Usenet 新闻协议 NNTP(Network News Transfer Protocol，网络新闻传输协议)是需要将金钱方面的开销降至最低程度的一个典型例子。注意 IP_TOS 选项并不适用于 Windows CE。

在提供 QOS 的套接字上尝试设置 TOS 位时，还有另一个问题需要注意。由于 IP 优先级设定已被 QOS 利用，以便对不同的服务等级加以区分，所以并不希望开发者任意更改这些值。因此，若在

具有 QOS 功能的套接字上调用 setsockopt 函数，同时又设置了 IP_TOS 选项，那么 QOS 服务提供程序会事先截获这一调用，查验是否对此项设定进行了修改。大家可参阅第 10 章，了解 QOS 的详情。

7.1.4.4　IP_TTL

optval 类型	获取/设置	Winsock 版本	说　明
int	两者均可	1+	IP TTL 参数

TTL 字段需要在 IP 头内进行设置。数据报利用 TTL 字段对数据报能够通过的最大路由器数量加以限制。这种限制的目的在于避免因路由循环而造成数据报永无休止地在网络中游荡。其背后的道理在于，数据报每经过一个路由器，都会由路由器将它的 TTL 值减去 1。若发现这个值变成了 0，数据报便会被丢弃。但要注意的是，Windows CE 不支持这个选项。

7.1.4.5　IP_MULTICAST_IF

optval 类型	获取/设置	Winsock 版本	说　明
unsigned long	两者均可	1+	获取或设置用于发出多播数据的本地接口

IP 多播接口(IF)选项用于设置一个本地接口，本地计算机以后发出的任何多播数据都会经由它传送出去。只有在那些同时安装了多个网络接口(如网卡或 Modem)的计算机上，此项设置才有意义。optval 参数应当是一个无符号长整数值，对应于本地接口的二进制 IP 地址。可用 inet_addr 函数将一个字符串形式的 IP 地址(点分十进制)转换成一个无符号长整数值，如下例所示：

```
DWORD mcastIF;

//先将套接字 s 加入到一个多播组中
mcastIF = inet_addr("129.113.43.120");
ret = setsockopt(s,IPPROTO_IP,IP_MULTICAST_IF,(char *)&mcastIF,
      sizeof(mcastIF));
```

7.1.4.6　IP_MULTICAST_TTL

optval 类型	获取/设置	Winsock 版本	说　明
int	两者均可	1+	为套接字获取或设置多播数据包的 TTL

和 IP 的 TTL 设置类似，这个选项执行相同的功能，只是它仅适用于通过指定套接字发出的多播数据。同样地，TTL 的目的是防止路由循环。但在多播通信环境中，设置 TTL 可缩小数据的传播范围。因此，多播组的各个成员必须在一个有效的范围之内，才能接收到数据报。多播数据报采用的默认 TTL 设定是 1。

7.1.4.7 IP_MULTICAST_LOOP

optval 类型	获取/设置	Winsock 版本	说 明
BOOL	两者均可	1+	如果是 TRUE，发至多播地址的数据将原封不动地反射回套接字的进入缓冲区

在默认情况下，在我们发送 IP 多播数据的时候，假如套接字也属于那个多播组的一名成员，数据便会原封不动地返回到该套接字。若将该选项设为 FALSE，则发出的任何数据都不会投递至该套接字的传入数据队列中。

7.1.4.8 IP_ADD_MEMBERSHIP

optval 类型	获取/设置	Winsock 版本	说 明
struct ip_mreq	只能设置	1+	在指定的 IP 多播组内为套接字赋与成员资格

该选项实际是由 Winsock 1 提供的一个方法，可将套接字加入一个指定的 IP 多播组。此时，需要用 socket 函数来创建地址族为 AF_INET 的一个套接字，同时将套接字的类型设为 SOCK_DGRAM。要想将套接字加入一个多播组，请使用下面的结构：

```
struct ip_mreq
{
   struct in_addr imr_multiaddr;
   struct in_addr imr_interface;
};
```

在 ip_mreq 结构中，imr_multiaddr 字段对应于打算加入的那个多播组的二进制地址；而 imr_interface 字段对应于加入多播组时所在的本地接口。关于有效多播地址的更多内容，大家可参考第 9 章。imr_interface 字段要么设为本地接口的一个二进制 IP 地址，要么设为 INADDR_ANY 值，表明选择的是默认接口(根据路由表决定)。

7.1.4.9 IP_DROP_MEMBERSHIP

optval 类型	获取/设置	Winsock 版本	说 明
struct ip_mreq	只能设置	1+	将套接字从指定的 IP 多播组内删去(撤消成员资格)

该选项正好与 IP_ADD_MEMBERSHIP 相反。如果用 ip_mreq 结构调用该选项，并在结构中放入与当初加入指定多播组时所用的相同的值，套接字 s 便会从那个组内被删去，脱离成员关系。同样地，第 9 章详细讲述了 IP 多播的详细知识。

7.1.4.10　IP_ADD_SOURCE_MEMBERSHIP

optval 类型	获取/设置	Winsock 版本	说　明
struct ip_mreq_source	只能设置	2+	加入一个多播组，但仅从给定源接受数据

这个多播选项在给定接口加入指定的多播组，但仅接受来自给定源 IPv4 地址(这称为包含列表)的数据。这个选项可以被多次调用，以建立一个具有多个可接受源的包含列表。输入结构的定义为：

```
struct ip_mreq_source {
   struct in_addr imr_multiaddr;
   struct in_addr imr_sourceaddr;
   struct in_addr imr_interface
};
```

第 1 个字段是 32 位多播地址，第 2 个字段是可接受源的 32 位 IPv4 地址，最后一个字段是用来从上面加入多播组的那个本地接口的 32 位 IPv4 地址。

Windows XP 及其后期版本支持这个选项，同时还需要局域网能够支持 IGMPv3。具体内容参见第 9 章。

7.1.4.11　IP_DROP_SOURCE_MEMBERSHIP

optval 类型	获取/设置	Winsock 版本	说　明
struct ip_mreq_source	只能设置	2+	从可接受源列表中删除给定 IPv4 源

这个选项是对 IP_ADD_SOURCE_MEMBERSHIP 选项的补充。我们一旦把一个或多个通过 IP_ADD_SOURCE_MEMBERSHIP 源加到某个特定的多播组后，就可以用这个选项来删除在包含列表中选定的源。使用这两个选项，应用程序就可以利用源列表从给定多播地址接受多播数据。这个选项要求 Windows XP 及支持 IGMPv3 的网络。具体内容参见第 9 章。

7.1.4.12　IP_BLOCK_SOURCE

optval 类型	获取/设置	Winsock 版本	说　明
struct ip_mreq_source	只能设置	2+	加入一个多播组，但接受来自给定 IPv4 源以外的每个地址的数据

这个选项及 IP_UNBLOCK_SOURCE 选项用于创建一个多播通信的源排除列表。IP_BLOCK_SOURCE 指定一个源后，来自该源的多播数据就不会被接受。可以多次调用这个选项来排除额外的源。这个选项要求 Windows XP 及支持 IGMPv3 的网络。具体内容参见第 9 章。

7.1.4.13 IP_UNBLOCK_SOURCE

optval 类型	获取/设置	Winsock 版本	说 明
BOOL	两者均可	1+	将给定 IPv4 源添加到可接受源列表中

这个选项将给定源从排除列表中删除，以便从这个被删除的源接收的多播数据可以传播给套接字。这个选项要求 Windows XP 及支持 IGMPv3 的网络。具体内容参见第 9 章。

7.1.4.14 IP_DONTFRAGMENT

optval 类型	获取/设置	Winsock 版本	说 明
BOOL	两者均可	1+	如果是 TRUE，就不对 IP 数据报进行分段

这个标志指示网络在传输过程中，不对 IPv4 数据报进行分段处理。然而，假如一个 IPv4 数据报的大小已经超过了 MTU(maximum transmission unit，最大传输单元)值，但未在 IPv4 头内设置分段标志，这个数据报就会被丢弃，同时向发送者返回一条 ICMP 错误消息("需要分段，但未设置分段位")。要注意的是，该选项并不适用于 Windows CE。

7.1.4.15 IP_PKTINFO

optval 类型	获取/设置	Winsock 版本	说 明
int	只能设置	2+	指明是否从 WSARecvMsg 返回数据包信息

这个选项指明，IPv4 数据包信息应该作为传给 WSARecvMsg API 的控制缓冲区的一个部分返回。这个函数在第 6 章中作了讨论。对 IPv4 而言，返回的结构是 IN_PKTINFO，其定义为：

```
typedef struct in_pktinfo {
   IN_ADDR    ipi_addr;
   UINT       ipi_ifindex;
}IN_PKTINFO;
```

第 1 个字段是 32 位的二进制 IPv4 地址，数据包从这个地址上接收。第 2 个字段是接口索引，可以用第 16 章讨论的 IP 助手 API 使其和特定的网络适配器关联起来。

7.1.5 IPPROTO_IPV6 选项级别

IPPROTO_IPV6 级别表示属于 IPv6 协议的套接字选项。其中的许多选项和 IPv4 选项对应。这些选项定义在 WS2TCPIP.H 中。

7.1.5.1　IPV6_HDRINCL

optval 类型	获取/设置	Winsock 版本	说　明
BOOL	两者均可	2+	在 AppleTalk 网络上注册指定的名称

这个选项指明，应用程序将为由 Winsock 发送调用所发送的每个数据包提供 IPv6 头。IPv6 堆栈不会对大于 MTU 的数据包实施分段操作，因此必须为每个发送的大于 MTU 的数据包建立适当的分段头。这个套接字选项仅对 SOCK_RAW 类型的套接字有效。在 IPv6 上，原始套接字才是真正意义上的原始。图 7.4 展示了 IPv6 头。

第 1 个字段的长度为 4 位，表示 IP 版本，在这里是 6。下一个 8 位字段定义通信类。这个字段现还没有定义明确的值。20 位流标号用于标记要求特殊处理的数据包序列。这个字段也没有预定义值。16 位有效负载字段是有效负载用 8 位字节表示的长度，其中包括除基本 IPv6 头本身外的任何扩展头。8 位的下一个头字段表示下一个头的协议值，可以是 IPv6 扩展头(如分段头)，也可以是下一个封装协议(如 UDP 或 TCP)。8 位的中继段限界字段表示数据包的 TTL 值。最后两个字段是 128 位源和目标 IPv6 地址。使用 IPV6_HDRINCL 选项的示例见第 11 章。

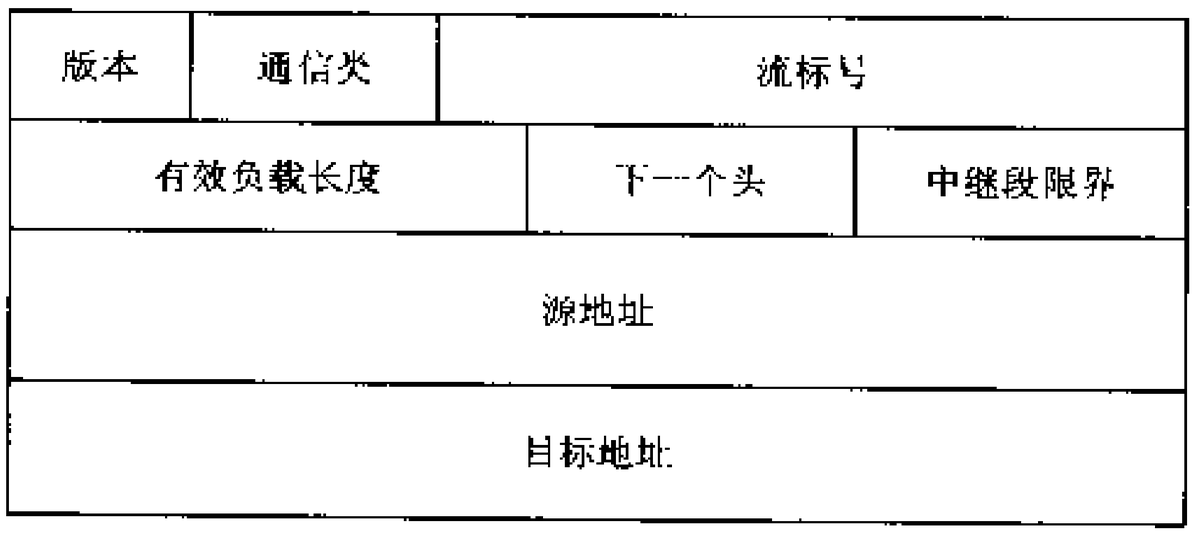


图 7.4　IPv6 头

7.1.5.2　IPV6_UNICAST_HOPS

optval 类型	获取/设置	Winsock 版本	说　明
int	两者均可	2+	IPv6 TTL 参数

这个选项用来设置 TTL 值，这个 TTL 值将用于套接字上发送的单播通信。这个选项和 IPv4 选项 IP_TTL 类似。

7.1.5.3　IPV6_MULTICAST_IF

optval 类型	获取/设置	Winsock 版本	说　明
int	两者均可	2+	发出多播数据的 IPv6 多播接口

这个选项和 IP_MULTICAST_IF 选项相似，不同之处在于这个选项设置 IPv6 多播数据的发出接

口。另外，它为外出通信指定接口的范围 ID，而不是指定本地 IPv6 接口地址。可以调用 IP 助手函数 GetAdaptersAddress 来获取本地接口的索引。IPv6 多播在第 9 章中有更详细的讨论，IP 助手 API 在第 16 章讨论。

7.1.5.4 IPV6_MULTICAST_HOPS

optval 类型	获取/设置	Winsock 版本	说 明
int	两者均可	1+	多播数据的 IPv6 TTL 参数

这个选项和 IP_MULTICAST_TTL 选项类似，不同之处在于它为 IPv6 多播通信设置 TTL 选项。

7.1.5.5 IPV6_MULTICAST_LOOP

optval 类型	获取/设置	Winsock 版本	说 明
int	两者均可	1+	如果是 TRUE，发至多播地址的数据将原封不动地反射回套接字的传入缓冲区

如果发送套接字也是多播组的一个成员的话，这个选项使得外出的 IPv6 多播通信能够反射回发送套接字。这个选项和 IPv4 多播选项 IP_MULTICAST_LOOP 类似。

7.1.5.6 IPV6_ADD_MEMBERSHIP,IPV6_JOIN_GROUP

optval 类型	获取/设置	Winsock 版本	说 明
struct ipv6_mreq	两者均可	2+	在指定接口上加入一个 IPv6 多播组

这个选项在一个特定的接口上加入 IPv6 多播组。输入参数的定义为：

```
typedef struct        ipv6_mreq {
   struct in6_addr    ipv6mr_multiaddr;
   unsigned int           ipv6mr_interface;
} IPV6_MREQ;
```

第 1 个字段是待加入的多播地址的 IPv6 地址。第 2 个字段是从它上边加入多播组的接口索引(或范围 ID)。更多内容参见第 9 章。

7.1.5.7 IPV6_DROP_MEMBERSHIP,IPV6_LEAVE_GROUP

optval 类型	获取/设置	Winsock 版本	说 明
struct ipv6_mreq	两者均可	2+	从给定接口上撤消多播组

这个选项从给定接口上撤消多播组成员。输入参数是 struct ipv6_mreq，它包含准备从给定接口(范围 ID)上撤消的多播组。更多内容参见第 9 章。

7.1.5.8　IPV6_PKTINFO

optval 类型	获取/设置	Winsock 版本	说　明
Int	两者均可	2+	指明是否从 WSARecvMsg 返回数据包信息

这个选项指明，IPv6 数据包信息应该作为传给 WSARecvMsg API 的控制缓冲区的一个部分返回。这个函数在第 6 章中有讨论。对 IPv6 而言，返回的结构是 IN6_PKTINFO，它的定义为：

```
typedef struct    in6_pktinfo {
    IN6_ADDR          ipi6_addr;
    UINT              ipi6_ifindex;
}IN6_PKTINFO;
```

第 1 个字段是二进制 IPv6 地址，应用程序在这个地址上接收数据包。第 2 个字段是接口索引，可以用第 16 章讨论的 IP 助手 API 使它和特定的网络适配器关联起来。

7.1.6　IPPROTO_RM 选项级别

IPPROTO_RM 级别的套接字选项用于可靠的多播传输。这些选项是在 wsrm.h 中声明的。Windows XP 支持的可靠多播协议的相关信息参见第 9 章。

7.1.6.1　RM_RATE_WINDOW_SIZE

optval 类型	获取/设置	Winsock 版本	说　明
RM_SEND_WINDOW	两者均可	2+	指定数据速率及窗口大小

这个选项设置发送窗口的大小，它决定了可以有效修复的数据的量，以及还有多少有效时间来修复数据。RM_SEND_WINDOW 结构的声明如下：

```
typedef struct _RM_SEND_WINDOW {
    ULONGRateKbitsPerSec;
    ULONGWindowSizeInMSecs;
    ULONG WindowSizeInBytes;
}RM_SEND_WINDOW
```

第 1 个字段表示数据速率，用千字节/每秒表示。第 2 个字段是用毫秒表示的窗口大小。最后一个字段是用字节表示的窗口大小。这 3 个字段的值必须满足等式：速率(千字节/每秒)—(字节表示的窗口大小×1024)/(毫秒表示的窗口大小×1000)。

7.1.6.2　RM_SET_MESSAGE_BOUNDARY

optval 类型	获取/设置	Winsock 版本	说　明
Int	两者均可	2+	指明应遵循的以字节表示的数据包的逻辑大小

这个选项用来以多数据块方式发送大型消息。例如，如果发送者希望将 2MB 缓冲区作为一个数据包发送，由于性能因素，在单个发送调用中提交 2MB 的缓冲区是不受欢迎的。设置这个选项就是为了指明传入数据包的逻辑大小。之后，发送者就可以以小数据块的方式发送这个 2MB 数据包。

7.1.6.3 RM_FLUSHCACHE

optval 类型	获取/设置	Winsock 版本	说 明
BOOL	只能设置	2+	刷新发送窗口的全部内容

这个选项刷新发送窗口的全部内容，使得不会再有 NAK(Negative Acknowledgement，否定应答)得到满足。这个选项当前还未实现。

7.1.6.4 RM_SENDER_WINDOW_ADVANCE_METHOD

optval 类型	获取/设置	Winsock 版本	说 明
eWINDOW_ADVANCE_METHOD	两者均可	2+	指明发送窗口推进的方式

这个选项指定发送窗口推进的方式。可能的值为枚举类型：

```
enum eWINDOW_ADVANCE_METHOD
{
    E_WINDOW_ADVANCE_BY_TIME = 1,     //默认模式
    E_WINDOW_USE_AS_DATA_CACHE
};
```

第 1 个值表示窗口随时间推进。也就是说，发送数据在 RM_SEND_WINDOW 结构中的 WindowSizeInMSecs 字段指定的时内量有效。第 2 个值表明仅在发送窗口根据 RM_SEND_WINDOW 结构中的 WindowSizeInBytes 字段所指定的发送窗口变满之后，才丢弃数据。

7.1.6.5 RM_SENDER_STATISTICS

optval 类型	获取/设置	Winsock 版本	说 明
RM_SENDER_STATS	只能获取	2+	返回对发送套接字的统计信息

这个选项获取一个 RM_SENDER_STATS 结构，这个结构中包含通话的各种网络统计信息，定义为：

```
typedef struct _RM_SENDER_STATS
{
    ULONGLONG DataBytesSent;            //#迄今为止已发送的客户机数据字节数
    ULONGLONG TotalBytesSent;           //SPM、OData 及 RData 字节数
    ULONGLONG NaksReceived;             //#迄今为止已收到的 NAK
    ULONGLONG NaksReceivedTooLate;      //#窗口推进之后收到的 NAK
    ULONGLONG NumOutstandingNaks;       //#待响应的 NAK
```

```
    ULONGLONG NumNaksAfterRData;          //#待响应的 NAK
    ULONGLONG RepairPacketsSent;          //#迄今为止已发送的 Repair (RDATA)
    ULONGLONG BufferSpaceAvailable;       //#部分丢弃的消息
    ULONGLONG TrailingEdgeSeqId;          //窗口中最小(最旧)的顺序 Id
    ULONGLONG LeadingEdgeSeqId;           //窗口中最大(最新)的顺序 Id
    ULONGLONG RateKBitsPerSecOverall;     //从开始在内部计算的发送速度
    ULONGLONG RateKBitsPerSecLast;        //对每个 INTERNAL_RATE_CALCULATION_
                                          //  FREQUENCY 计算的发送速度
}RM_SENDER_STATS;
```

头文件中每个字段的注释对结构作出了解释，这里就不再重复了。

7.1.6.6　RM_LATEJOIN

optval 类型	获取/设置	Winsock 版本	说　明
int	两者均可	2+	允许接受者对窗口中的任何数据包 NAK

如果设置了这个选项，只要通话一加入，接受者就可以 NAK 否定应答广告窗口内的任何顺序号。

7.1.6.7　RM_SET_SEND_IF

optval 类型	获取/设置	Winsock 版本	说　明
ULONG	两者均可	2+	设置可靠广播通信的外出接口

这个选项设置可靠广播通信的外出接口。输入参数为本地外出接口的 32 位二进制地址。

7.1.6.8　RM_ADD_RECEIVE_IF

optval 类型	获取/设置	Winsock 版本	说　明
ULONG	只能设置	2+	将给定接口添加为一个接收接口

在默认情况下，创建一个可靠多播的接收者之后，它就会监听默认接口(由路由表指明)上的通信。这个选项可以指定用来监听通信的本地接口的 32 位二进制 IPv4 地址。可以多次指定这个选项，以便在一个有多重初始地址的计算机中，添加多个监听接口。

7.1.6.9　RM_DEL_RECEIVE_IF

optval 类型	获取/设置	Winsock 版本	说　明
ULONG	只能设置	2+	将给定接口从接收接口列表中删除

这个选项将给定接口从可靠多播通信的有效接收接口列表中删除。所提供的地址是 32 位二进制 IPv4 地址。

7.1.6.10 RM_SEND_WINDOW_ADV_RATE

optval 类型	获取/设置	Winsock 版本	说 明
int	两者均可	2+	设置发送窗口增量的百分比

随着时间推移，发送窗口递增式推进，以便抛弃最旧的数据，为应用程序发送的新数据腾出空间。这个选项控制每次推进所抛弃数据的量。最大的可能值表示为整个窗口的百分比，定义为 MAX_WINDOW_INCREMENT_PERCENTAGE。

7.1.6.11 RM_USE_FEC

optval 类型	获取/设置	Winsock 版本	说 明
RM_FEC_INFO	两者均可	2-	启用通话上的正向纠错

这个选项启用通话上的 FEC(forward error correction，正向纠错)。RM_FEC_INFO 结构的定义为：

```
typedef struct _RM_FEC_INFO
{
    USHORT    FECBlockSize;
    USHORT    FECProActivePackets;
    UCHAR     FECGroupSize;
    BOOLEAN   fFECOnDemandParityEnabled;
}RM_FEC_INFO;
```

FECGroupSize 字段表示用来建立奇偶数据包的原始数据包数量，FECBlockSize 表示原始数据包加上生成的奇偶数据包的数量。FECProActivePackets 字段表示始终传输 FEC 数据包(而不只是为了修复数据才传输)。fFECOnDemandParityEnabled 字段允许接受者发出 FEC 修复数据请求的命令。

7.1.6.12 RM_SET_MCAST_TTL

optval 类型	获取/设置	Winsock 版本	说明
int	两者均可	2+	为发送的可靠多播数据设置 TTL

这个选项为一个可靠多播数据发送者设置 TTL 值。

7.1.6.13 RM_RECEIVER_STATISTICS

optval 类型	获取/设置	Winsock 版本	说 明
RM_RECEIVER_STATS	两者均可	2+	在一个接受者套接字上检索通话统计信息

这个选项获取一个 RM_SENDER_STATS 结构，这个结构中包含通话的各种网络统计信息，其定义为：

```
typedef struct _RM_RECEIVER_STATS
```

```
{
    ULONGLONG NumODataPacketsReceived; //#接收到的 OData 序列
    ULONGLONG NumRDataPacketsReceived; //#接收到的 RData 序列
    ULONGLONG NumDuplicateDataPackets; //#接收到的 RData 序列
    ULONGLONG DataBytesReceived;       //#迄今为止已接收的客户机数据字节数
    ULONGLONG TotalBytesReceived;      //SPM、OData 及 RData 字节数
    ULONGLONG RateKBitsPerSecOverall;      //从最初开始的在内部计算的接收速度
    ULONGLONG RateKBitsPerSecLast;     //对每个 INTERNAL_RATE_CALCULATION_
                                       //  FREQUENCY 计算的接收速度
    ULONGLONG TrailingEdgeSeqId;       //窗口中最小(最旧)的顺序 Id
    ULONGLONG LeadingEdgeSeqId;        //窗口中最大(最新)的顺序 Id
    ULONGLONG AverageSequencesInWindow;
    ULONGLONG MinSequencesInWindow;
    ULONGLONG MaxSequencesInWindow;
    ULONGLONG FirstNakSequenceNumber;  //#第 1 个未完成的 NAK
    ULONGLONG NumPendingNaks;              //等待 NCF 的序列
    ULONGLONG NumOutstandingNaks;      //已接收到 NCF 但没有收到 RDATA 的#序列
    ULONGLONG NumDataPacketsBuffered;  //#当前通过传输进行缓冲的数据包
    ULONGLONG TotalSelectiveNaksSent;      //#迄今为止已发送的选择性 NAK
    ULONGLONG TotalParityNaksSent;     //#迄今为止已发送的解析 NAK
}RM_RECEIVER_STATS;
```

头文件中每个字段的注释对结构作出了解释，这里就不再重复了。

7.1.7　IPPROTO_TCP 选项级别

仅有一个选项从属于 IPPROTO_TCP 级别。该选项只适用于流套接字(SOCK_STREAM)，其地址族为 AF_INET。这个选项可用在所有 Winsock 版本上，并得到了所有 Windows 平台的支持。

TCP_NODELAY

optval 类型	获取/设置	Winsock 版本	说　明
BOOL	两者均可	1+	若为 TRUE，就会在套接字上禁用 Nagle 算法

为减少网络通信的开销，提升性能以及数据吞吐率，在默认状态下系统采用 Nagle 算法。若应用程序请求发送一批数据，那么系统在接收了这些数据之后，可能会稍候一段时间，等其他数据累积进来，最后统一发送出去。但假如在一段规定的时间内，并无新数据加入，那么原先那些数据当然也会发送出去。这样做的好处便是：在单独一个 TCP 包内，数据量增大了。与之相反的则是：使用多个 TCP 包，但每个包携带的数据量比较少。如果是后一种情况，那么必然会涉及的一项开销在于，对每个包来说，其 TCP 头都必须占据 20 个字节的长度。例如，假定在此只发送几个字节，那么 20 个字节的头便显得有点儿多余。因此，在采用了 Nagle 算法后，可以更有效地利用数据包的可用空间。该算法的另一个功能是延迟发送“收到确认”消息。系统收到 TCP 数据之后，必须向对方反馈回一条

ACK 消息。但采用了该算法后，主机会暂时等待一段时间，看看是否有需要发给对方的数据，以便能随那些数据一道，将 ACK 消息反馈回去，从而节省一个数据包的通信量(这又是一项开销)。

因为在某些情况下， Nagle 算法的行为反而会产生不利的影响，本选项的目的便是禁止采用它。若网络应用程序通常只需发送数量相当少的数据，同时要求能得到极其迅速的回应，那么再使用这种算法，反而会明显影响性能。Telnet 便是这样的一个典型例子。Telnet 的本质是一种交互式应用，用户可通过它登录至一台远程计算机，然后向其传送命令。通常，用户每秒钟只会进行少量的键击。若 Nagle 在此仍要发挥作用，便会造成响应的迟钝，甚至造成对方主机不予应答的错觉。

7.1.8 NSPROTO_IPX 选项级别

这里列出的套接字选项均属 Microsoft 公司对 Windows IPX/SPX 套接字接口所作的特有扩展，用于确保与现有应用程序的兼容性。因为这些选项仅适用于 Microsoft IPX/SPX 堆栈，所以若非考虑到兼容性方面的原因，我们并不建议使用它们。若某个应用程序利用了这些扩展，那么完全有可能无法在其他 IPX/SPX 系统中正常工作。这些选项均定义在 WSNWLINK.H 头文件中。但在程序中定义时，应将该文件包括在 WINSOCK.H 及 WSIPS.H 这两个文件的后面。

7.1.8.1 IPX_PTYPE

optval 类型	获取/设置	Winsock 版本	说 明
int	两者均可	1+	获取或设置 IPX 包的类型

该选项用于设置或获取 IPX 数据包的类型。在自这个套接字发出的每个 IPX 包内，optval 参数中指定的值都对应于数据包的类型。optval 参数指定的是一个整数值。

7.1.8.2 IPX_FILTERPTYPE

optval 类型	获取/设置	Winsock 版本	说 明
int	两者均可	1+	获取或设置准备过滤的 IPX 包的类型

该选项用于获取或设置接收筛选器数据包的类型。注意在任何接收调用中，只有类型与 optval 参数中指定的值相符的 IPX 包才会返回；若类型与指定的数据包类型不符，数据包便会被丢弃。

7.1.8.3 IPX_STOPFILTERPTYPE

optval 类型	获取/设置	Winsock 版本	说 明
int	只能设置	1+	删除为指定 IPX 包设置的筛选器

可用该选项停止过滤当初用 IPX_FILTERTYPE 选项设置的数据包类型。

7.1.8.4　IPX_DSTYPE

optval 类型	获取/设置	Winsock 版本	说　明
int	两者均可	1+	获取或设置 SPX 头中的数据流字段值

针对发出的每个数据包的 SPX 头，该选项用于获取或设置数据流字段的值。

7.1.8.5　IPX_EXTENDED_ADDRESS

optval 类型	获取/设置	Winsock 版本	说　明
BOOL	两者均可	1+	如果是 TRUE，便允许对 IPX 包进行扩展寻址

该选项用于允许或禁止扩展寻址。发送数据时，它会在 SOCKADDR_IPX 结构中加入 unsigned char sa_ptype 元素，使结构的总长变成 15 字节；而在接收数据时，该选项会在 SOCKADDR_IPX 结构中同时添加 sa_ptype 和 unsigned char sa_flags 元素，使其总长变成 16 字节。目前在 sa_flags 中定义的位是：

- 0x01：接收的帧是以广播形式发送出去的
- 0x02：接收的帧是自这台计算机发送出去的

7.1.8.6　IPX_RECVHDR

optval 类型	获取/设置	Winsock 版本	说　明
BOOL	两者均可	1+	如果是 TRUE，就把 IPX 头随接收调用一起返回

若将该选项设为 TRUE，那么任何 Winsock 接收调用都会将 IPX 头随正式数据一道返回。

7.1.8.7　IPX_MAXSIZE

optval 类型	获取/设置	Winsock 版本	说　明
BOOL	只能获取	1+	返回 IPX 数据报的最大长度

调用 getsockopt 函数时利用该选项，便可返回允许的最大 IPX 数据报长度。

7.1.8.8　IPX_ADDRESS

optval 类型	获取/设置	Winsock 版本	说　明
IPX_ADDRESS_DATA	只能获取	1+	返回支持 IPX 的适配器的有关信息

该选项用于对绑定 IPX 的特定适配器信息进行查询。在同时安装了 n 个网络适配器(网卡、Modem 等等)的系统中，适配器的编号范围在 0 到 n-1 之间。如果想调查系统内到底安装了多少个支持 IPX 的适配器，可在设置 IPX_MAX_ADAPTER_NUM 选项的前提下，调用 getsockopt；或者重复调用

IPX_ADDRESS，同时递增 adapternum 的值，直至有错误返回。optval 参数指向的是一个 IPX_ADDRESS_DATA 结构。下面是该结构的定义：

```
typedef struct _IPX_ADDRESS_DATA
{
    INT        adapternum;        //输入：从 0 开始算起的适配器编号
    UCHAR          netnum[4];         //输出：IPX 网络编号
    UCHAR          nodenum[6];        //输出：IPX 节点地址
    BOOLEAN    wan;           //输出：值为 TRUE 时表明适配器是在一个 WAN 链接上
    BOOLEAN    status;            //输出：值为 TRUE 时表明 WAN 链接为上行(或适配器不是 WAN)
    INT        maxpkt;            //输出：最大数据包的大小，不包括 IPX 头
    ULONG          linkspeed;         //输出：用 100 字节/秒(100bps)表示的链接速度(如 96
表示 9600 bps)
}IPX_ADDRESS_DATA,*PIPX_ADDRESS_DATA;
```

7.1.8.9 IPX_GETNETINFO

optval 类型	获取/设置	Winsock 版本	说　明
IPX_NETNUM_DATA	只能获取	1+	返回与一个指定 IPX 网络编号有关的信息

这个选项获取与一个特定 IPX 网络编号有关的信息。若网络刚好位于 IPX 的高速缓存之中，那么该选项会直接将信息返回；否则的话，它会发出 RIP 请求进行查找。optval 参数指向的是一个有效的 IPX_NETNUM_DATA 结构。对这个结构的定义如下：

```
typedef struct _IPX_NETNUM_DATA
{
    UCHAR  netnum[4];  //输入：IPX 网络编号
    USHORT hopcount;   //输出：这个网络的跳跃总数，按计算机顺序
    USHORT netdelay;   //输出：这个网络的滴答总数，按计算机顺序
    INT     cardnum;           //输出：用来路由到这个网络的基于 0 的适配器编号；
                               //可作为 adapternum 输入到 IPX_ADDRESS
    UCHAR  router[6];          //输出：下一跳路由器的 MAC 地址，如果网络是直达的，则置为 0
}IPX_NETNUM_DATA,*PIPX_NETNUM_DATA;
```

7.1.8.10 IPX_GETNETINFO_NORIP

optval 类型	获取/设置	Winsock 版本	说　明
IPX_NETNUM_DATA	两者均可	1+	如果是 TRUE，则不对 IP 数据报进行分段

该选项类似于 IPX_GETNETINFO，只是它并不发出 RIP 请求。若网络正好在 IPX 的高速缓存之中，信息便可直接返回；否则，调用就会失败，而不会发出更多的请求。大家同时可参考 IPX_RERIPNETNUMBER，后者无论总是会发出 RIP 请求。和 IPX_GETNETINFO 相似，这个选项要求以 optval 参数的形式，传递一个 IPX_NETNUM_DATA 结构。

7.1.8.11　IPX_SPXGETCONNECTIONSTATUS

optval 类型	获取/设置	Winsock 版本	说　明
IPX_SPXCONNSTATUS_DATA	只能获取	1+	返回与一个已建立连接的 SPX 套接字有关的信息

该选项可返回与一个已建立连接的 SPX 套接字有关的信息。optval 参数指向的是一个 IPX_SPXCONNSTATUS_DATA 结构，定义见下。所有编号都按网络字节顺序排列(从高到低)。

```
typedef struct _IPX_SPXCONNSTATUS_DATA
{
    UCHAR   ConnectionState;
    UCHAR   WatchDogActive;
    USHORT LocalConnectionId;
    USHORT RemoteConnectionId;
    USHORT LocalSequenceNumber;
    USHORT LocalAckNumber;
    USHORT LocalAllocNumber;
    USHORT RemoteAckNumber;
    USHORT RemoteAllocNumber;
    USHORT LocalSocket;
    UCHAR   ImmediateAddress[6];
    UCHAR   RemoteNetwork[4];
    UCHAR   RemoteNode[6];
    USHORT RemoteSocket;
    USHORT RetransmissionCount;
    USHORT EstimatedRoundTripDelay;/*单位为毫秒*/
    USHORT RetransmittedPackets;
    USHORT SuppressedPacket;
}IPX_SPXCONNSTATUS_DATA,*PIPX_SPXCONNSTATUS_DATA;
```

7.1.8.12　IPX_ADDRESS_NOTIFY

optval 类型	获取/设置	Winsock 版本	说明
IPX_ADDRESS_DATA	只能获取	1+	若 IPX 适配器的状态发生变化，则发出异步通知

该选项可提交一个请求，以便在 IPX 绑定的那个适配器的状态发生变化后，及时收到通知。注意，状态的变化通常是在 WAN 线路建立或断开的时候发生的。该选项要求调用者提交一个 IPX_ADDRESS_DATA 结构，将其作为 optval 参数传递。然而，有一个例外是，在 IPX_ADDRESS_DATA 后面，要紧随着一个句柄，这个句柄对应于一个尚未进入传信状态的事件。下述伪代码阐释了调用该选项的一个方法：

```
char buff[sizeof(IPX_ADDRESS_DATA)+sizeof(HANDLE)];
IPX_ADDRESS_DATA *ipxdata;
HANDLE *hEvent;

ipxdata = (IPX_ADDRESS_DATA *)buff;
hEvent = (HANDLE *)(buff +sizeof(IPX_ADDRESS_DATA));
ipxdata->adapternum = 0;//设置为适当的适配器
*hEvent = CreateEvent(NULL,TRUE,FALSE,NULL);
setsockopt(s,NSPROTO_IPX,IPX_ADDRESS_NOTIFY,(char *)buff,
sizeof(buff));
```

在提交了 getsockopt 查询之后，它会成功完成任务。然而，optval 指向的 IPX_ADDRESS_DATA 结构在那时却不会被更新。相反，请求会在传输层内部进入排队队列；当适配器的状态发生变化时，IPX 就会找到队列中的一个 getsockopt 查询，并填好 IPX_ADDRESS_DATA 结构中的各个字段。随后，它会传信由 optval 缓冲区内的句柄指向的那个事件。若同时提交多个 getsockopt 调用，那么必须使用不同的事件。利用事件的目的是由于调用需要异步进行，而 getsockopt 目前尚不支持这一功能。

注意 以目前的实现状况来看，针对状态发生的每一种变化，传输层都只会传信队列中的一个查询。因此，在同一个时候，只能运行一个使用了排队查询的服务。

7.1.8.13 IPX_MAX_ADAPTER_NUM

optval 类型	获取/设置	Winsock 版本	说　明
int	只能获取	1+	返回存在的 IPX 适配器的个数

该选项可返回系统中支持 IPX 的适配器的数量。若该调用返回 n 个适配器，那么适配器的编号范围便在 0 到 n-1 之间。

7.1.8.14 IPX_RERIPNETNUMBER

optval 类型	获取/设置	Winsock 版本	说　明
IPX_NETNUM_DATA	只能获取	1+	返回一个网络编号的相关信息

该选项和 IPX_GETNETINFO 颇为类似，只是它会强迫 IPX 重发 RIP 请求，即便目标网络当前已位于它的缓冲区内(然而，假若它与目标网络直连，就不必重发)。和 IPX_GETNETINFO 类似，它也要求通过 optval 参数，传递一个 IPX_NETNUM_DATA 结构。

7.1.8.15 IPX_RECEIVE_BROADCAST

optval 类型	获取/设置	Winsock 版本	说　明
BOOL	只能设置	1+	如果是 TRUE，就不接收 IPX 广播包

默认情况下，IPX 套接字有能力接收广播数据包。若应用程序不需要收取广播包，则应将该选项

设为 FALSE，从而在一定程度上改善系统性能。但要注意的是，即使将选项设为 FALSE，也并不一定会为那个应用程序过滤掉广播数据。

7.1.8.16　IPX_IMMEDIATESPXZCK

optval 类型	获取/设置	Winsock 版本	说　明
BOOL	两者均可	1+	如果是 TRUE，就不在 SPX 连接上延迟发送 ACK

如将该选项设为 TRUE，那么确认包不会在 SPX 连接之上延迟发送。若应用程序不需要在 SPX 通信链路上进行双向通信，便可将其设为 TRUE。这样做尽管会增加发送的 ACK 包的数量，但却能防止应用程序由于延迟发送确认消息，造成对性能的影响。

7.2　IOCTLSOCKET、WSAIOCTL 和 WSANSPIoctl

套接字 ioctl 函数用于在套接字之上，控制套接字上的 I/O 行为，同时获取与那个套接字上挂起的 I/O 操作的有关信息。其中，第 1 个函数是 ioctlsocket，起源于 Winsock 1 规范，它的声明如下：

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

其中，参数 s 指定的是要在上面采取 I/O 操作的套接字描述符，而参数 cmd 是一个预定义的标志，用在将执行的 I/O 控制命令上。最后一个参数 argp 对应的是一个指针，指向与命令密切相关的一个变量。描述好每个命令之后，再给出所需变量的类型。Winsock 2 引入了一个新的 ioctl 函数，增添了数量较多的新选项。首先，它将单个 argp 参数分解成了一系列输入参数，用于容纳传递到函数内部的值；同时提供一系列输出参数，用于容纳从调用中返回的数据。此外，函数调用可使用重叠 I/O。这个新函数便是 WSAIoctl，它的定义如下：

```
int WSAIoctl(
    SOCKET s,
    DWORD dwIoControlCode,
    LPVOID lpvInBuffer,
    DWORD cbInBuffer,
    LPVOID lpvOutBuffer,
    DWORD cbOutBuffer,
    LPDWORD lpcbBytesReturned,
    LPWSAOVERLAPPED lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
```

```
);
```

头两个参数与 ioctlsocket 中的前两个参数相同。另两个参数 lpvInBuffer 和 cbInBuffer 则对输入参数进行了描述。其中，lpvInBuffer 参数是一个指针，指向传入的值，而 cbInBuffer 指定的是数据的多少，以字节为单位。类似地，lpvOutBuffer 和 cbOutBuffer 用于从调用中返回的数据。lpvOutBuffer 参数指向的是一个数据缓冲区，其中放置了返回的所有信息。而 cbOutBuffer 参数对应的是作为 lpvOutBuffer 传递进来的缓冲区的字节长度。要注意的是，某些调用可能只使用了输入或输出参数，而另一些调用两类参数都会用到。第 7 个参数是 lpcbBytesReturned，对应于实际返回的字节数。最后两个参数是 lpOverlapped 和 lpCompletionRoutine，在进行重叠 I/O 时调用这个函数就会用到。大家可参考第 5 章，了解重叠 I/O 的使用详情。

最后，Windows XP 中引入了一个新的 ioctl 函数 WSANSPIoctl。这个函数专门用来对命名空间提供程序作出 I/O 控制调用。该函数的定义为：

```
int WSANSPIoctl(
    HANDLE hLookup,
    DWORDdwControlCode,
    LPVOID lpvInBuffer,
    DWORDcbInBuffer,
    LPVOID lpvOutBuffer,
    DWORDcbOutBuffer,
    LPDWORD lpcbBytesReturned,
    LPWSACOMPLETION lpCompletion
);
```

除 hLookup 和 lpCompletion 参数之外，这个函数的其余参数与 WSAIoctl 的参数一样。传递给这个函数的句柄是从 WSALookupServiceBegin 返回的句柄。这个句柄标识的是对一个特定命名空间提供程序作出的命名空间查询。lpCompletion 参数指定，当命名空间提供程序或查询发生变化时，应该如何通知应用程序。这个新的 ioctl 函数由 NLA(Network Location Awareness，网络定位认知)服务使用，第 8 章将详细讲述该服务，因此这里就不再讨论了。

7.2.1 标准 I/O 控制命令

有 3 个 I/O 控制命令最为常用，它们是从 Unix 系统移植过来的，所有 Windows 平台都支持。另外，可以使用 ioctlsocket 或 WSAIoctl 调用这 3 个命令。

7.2.1.1 FIONBIO

函　数	输　入	输　出	Winsock 版本	说　明
ioctlsocket/WSAIoctl	unsigned int	无	1+	将套接字置入非阻塞模式

该命令可在套接字 s 上允许或禁止非阻塞模式。在默认情况下，所有套接字在创建好后，都会自

动成为阻塞套接字。若用 FIONBIO 这个 I/O 控制命令来调用 ioctlsocket，那么应设置 argp，用它来传递指向一个无符号长整数的指针；若打算启用非阻塞模式，应将那个长整数的值设为一个非零值。而若设为 0 值，意味着套接字将进入阻塞模式。如使用 WSAIoctl，只需在 lpvInBuffer 参数中简单地传递那个无符号长整数即可。

调用 WSAAsyncSelect 或 WSAEventSelect 函数的时候，套接字会被自动设为非阻塞模式。调用了其中任何一个函数之后，再有任何将套接字设回阻塞模式的企图，都会以失败告终，并返回 WSAEINVAL 错误。要想将套接字改回阻塞模式，应用程序首先必须禁止 WSAAsyncSelect。具体的做法是调用 WSAAsyncSelect，同时令 lEvent 参数等于 0。或者调用 WSAEventSelect，令 lNetworkEvents 参数等于 0，从而禁止 WSAEventSelect。

7.2.1.2　FIONREAD

函　数	输　入	输　出	Winsock 版本	说　明
两者均可	无	unsigned long	1+	返回准备在套接字上读取的数据量

该命令用于决定可从套接字上读入的数据量。对 ioctlsocket 来说，argp 值会返回一个无符号的整数，其中包含了打算读入的字节数。而若使用 WSAIoctl，那么无符号的整数是在 lpvOutBuffer 中返回的。若套接字 s 是一个面向数据流的套接字(类型为 SOCK_STREAM)，那么 FIONREAD 会返回在单独一次接收调用中，可读入的数据总量。要注意的是，若使用这种或其他形式的消息查看机制，并不一定能够保证返回正确的数据量。若在一个数据报套接字(类型为 SOCK_DGRAM)上使用 I/O 控制命令，返回值就是在套接字上排队的第 1 条消息的大小。

7.2.1.3　SIOCATMARK

函　数	输　入	输　出	Winsock 版本	说　明
两者均可	无	BOOL	1+	判断是否已读取了 OOB 数据

若将一个套接字配置成接收 OOB 数据，而且已设置成以内嵌方式读取这种 OOB 数据(通过设置 SO_OOBINLINE 套接字选项)，那么这个 I/O 控制命令就会返回一个布尔值，指出接下来是否准备接收 OOB 数据。如答案是肯定的，则返回 TRUE；否则，便返回 FALSE，而且下一次接收操作会返回 OOB 数据之前的所有或部分数据。对 ioctlsocket 来说，argp 会返回一个指向布尔变量的指针；而对 WSAIoctl 来说，会在 lpvOutBuffer 中返回指向布尔变量的指针。要记住的是，在同一个调用中，接收调用绝对不会将 OOB 数据和普通数据混和在一起。请参考第 1 章，那里讲述了 OOB 数据的详细情况。

7.2.2　其他 I/O 控制命令

除那些与 SSL(Secure Sockets Layer，加密套接字协议层)处理有关的命令之外，本小节列出的所有 I/O 控制命令都是 Winsock 2 特有的。前者只适用于 Windows CE 平台。如仔细观察 Winsock 2 的

头结构，便会发现其中实际还声明了另一些 I/O 控制命令。然而，本节只准备列出与用户的应用程序有关的那一部分 I/O 控制命令。此外，就像大家马上会看到的那样，并不是所有 I/O 控制命令都能在所有 Windows 平台上通用。相反，随着操作系统的升级换代，这些命令也必然会发生变化。对 Winsock 2 来说，其中大多数命令都是在 WINSOCK2.H 文件中定义的。而一些更新的命令则定义在 MSTCPIP.H 中。

7.2.2.1 SIO_ENABLE_CIRCULAR_QUEUEING

函 数	输 入	输 出	Winsock 版本	说 明
WSAIoctl	BOOL	BOOL	2+	如接收缓冲区队列溢出，则首先丢弃最早收到的消息

这个 I/O 控制命令在缓冲队列排满之后，用于控制下层服务提供程序如何处理传入的数据报消息。在默认情况下，若负责存放传入数据的缓冲区满了，那么以后新收到的任何消息都会被丢弃。若将该选项设为 TRUE，便表明新收到的消息绝对不会由于缓冲区的溢出而被丢弃；相反，队列中的那些最老的消息(最先收到的消息)会被丢弃，以便为新到的消息腾出地方。该命令仅适用于与不可靠的、面向消息的协议关联在一起的套接字。如果在其它类型的套接字上使用这个 I/O 控制命令(例如采用面向数据流协议的套接字)，或者服务提供程序本身不支持该命令，就会返回错误 WSAENOPROTOOPT。要注意的是，该选项目前只能在 Windows NT 平台上使用。

这个 I/O 控制命令既可用于打开和关闭循环队列，也可用于查询选项的当前状态。设置这个选项时，只需使用输入参数。若对选项的当前状态进行查询，那么只需提供输出布尔参数。

7.2.2.2 SIO_FIND_ROUTE

函 数	输 入	输 出	Winsock 版本	说 明
WSAIoctl	SOCKADDR	BOOL	2+	验证到指定地址的路由是否存在

该 I/O 控制命令用于核查是否能通过网络，访问到一个特定的地址。lpvInBuffer 参数指向与指定协议对应的一个 SOCKADDR 结构。若目标地址已存在于本地高速缓存中，lpvInBuffer 参数便是无效的。对 IPX 来说，该调用会引发一个 IPXGetLocalTarget 调用，以便在网络中查询一个指定的远程地址。但不幸的是，目前各个 Windows 平台上的 Microsoft 提供程序都还没有实现这个 I/O 控制命令。

7.2.2.3 SIO_FLUSH

函 数	输 入	输 出	Winsock 版本	说 明
WSAIoctl	无	无	2+	丢失发送缓冲区

这个 I/O 控制命令可丢弃当前与指定套接字关联的发送队列的内容。该选项没有输入或输出参数。目前，Windows NT 4 Service Pack 4、Windows 2000 及 Windows XP 提供了对它的支持。

7.2.2.4　SIO_GET_BROADCAST_ADDRESS

函　数	输　入	输　出	Winsock 版本	说　明
WSAIoctl	无	SOCKADDR	2+	为套接字地址族返回一个广播地址

这个 I/O 控制命令可返回一个 SOCKADDR 结构(通过 lpvOutBuffer)，其中包含了套接字 s 的地址族的广播地址，该地址可在 sendto 或 WSASendTo 中使用。这个命令仅适用于 Windows NT 操作系统。在 Windows 95、Windows 98 及 Windows Me 中均会返回 WSAEINVAL 错误。

7.2.2.5　SIO_GET_EXTENSION_FUNCTION_POINTER

函　数	输　入	输　出	Winsock 版本	说　明
WSAIoctl	GUID	函数指针	2+	取得下层提供程序特有的函数指针

这个 I/O 控制命令用于获取具体提供程序特有的函数(这类函数并不属于 Winsock 标准规范的一部分)。若选定了一种提供程序，程序员便可通过该 I/O 控制命令来利用这些函数，具体的做法是为每个函数都分配一个 GUID(globally unique identifier，全球唯一标识符)。随后，应用程序可使用 I/O 控制命令 SIO_GET_EXTENSION_FUNCTION_POINTER，取得指向函数的一个指针。在头文件 Mswsock.h 中，定义了由微软添加的一系列 Winsock 函数，包括各自对应的 GUID。例如，要查询目前的 Winsock 提供程序是否支持 TransmitFile 函数，可用它的 GUID 对提供程序进行查询。这个 GUID 是用下述代码定义的：

```
#define WSAID_TRANSMITFILE \
{0xb5367df0,0xcbac,0x11cf,{0x95,0xca,0x00,0x80,0x5f,0x48,0xa1,0x92}}
```

取得了像 TransmitFile 这样的一个扩展函数的指针后，便可直接调用它，而不必事先将应用程序同 Mswsock.lib 库文件链接起来。这样实际上可取消在 Mswsock.lib 中进行的一次中间函数调用。

大家可参阅 Mswsock.h 文件，了解还有哪些微软特有的扩展已定义了自己的 GUID。关于微软特有扩展的更多内容，请参见第 6 章。开发一个分层的服务提供程序时，I/O 控制命令是至关重要的一个环节。请参阅第 12 章，了解有关服务提供程序接口的详情。

7.2.2.6　SIO_CHK_QOS

函数	输入	输出	Winsock 版本	说明
WSAIoctl	DWORD	DWORD	2+	检查指定套接字的 QOS 属性

这个 I/O 控制命令可用来检查 QOS 内部的 6 种状态，目前只有 Windows 2000 及其后期版本才提供了对它的支持。以下 6 个标志分别对应于 6 种不同的状态：ALLOWED_TO_SEND_DATA、ABLE_TO_RECV_RSVP、LINE_RATE、LOCAL_TRAFFIC_CONTROL、LOCAL_QOSABILITY、END_TO_END_QOSABILITY。

其中，第 1 个标志 ALLOWED_TO_SEND_DATA 用在套接字上的 QOS 等级建立好之后(使用 SIO_SET_QOS 建立)，但又在接收到任何 RSVP 的 RESV(reservation request，预约请求)消息之前。若收到一条 RESV 消息，意味着要求的带宽已分配给了数据流。在收到 RESV 消息之前，与套接字对应的数据流只受到了“最大努力”(best-effort)的服务。在本书第 10 章，还会就 RSVP 协议以及网络资源的预约进行详尽、深入的探讨。利用 ALLOWED_TO_SEND_DATA 标志，可查看当前的“最大努力”的服务是否足够保证由 SIO_SET_QOS 请求的 QOS 服务等级。返回值要么是 1——意味着当前的“尽最大努力”的带宽已经足够使用；要么是 0——意味着目前的带宽尚不足以保证所请求的服务等级。若 ALLOWED_TO_SEND_DATA 标志的返回值是 0，那么作为发送方的应用程序，应在发出数据之前，等候一条 RESV 消息的抵达。

第 2 个标志是 ABLE_TO_RECV_RSVP，指出主机是否能够在指定套接字所绑定的那个接口上，接收和处理 RSVP 消息。返回值是 1 或 0，分别指出能还是不能接收 RSVP 消息。

下一个标志是 LINE_RATE，可返回“最大努力”的线路通信速率达到多少，单位是每秒多少千位(kbps)。若线路的通信速率未知，便返回一个 INFO_NOT_AVAIABLE 值。

最后 3 个标志指出在本地计算机或网络上，是否提供了特定的功能。对所有这 3 个选项来说，若返回值是 1，便表明对应的功能是支持的；若返回 0，则表明不支持。另外，若返回 INFO_NOT_AVAIABLE，表明没有办法检查。LOACAL_TRAFFIC_CONTROL 用于判断计算机上是否安装了通信控制组件，而且是否能够使用。LOCAL_QOSABILIT 用于判断本地计算机是否支持 QOS。最后，END_TO_END_QOSABILITY 指出整个本地网络是否启用了 QOS。

7.2.2.7 SIO_GET_QOS

函 数	输 入	输 出	Winsock 版本	说 明
WSAIoctl	无	QOS	2+	返回与套接字相关联的 QOS 结构

这个 I/O 控制命令负责接收同套接字关联在一起的 QOS 结构。提供的缓冲区必须足够大，以使容下整个结构。因为结构中可能包含了提供程序特有的一些信息，所以它有时会比 sizeof(QOS)稍大一些。欲了解 QOS 的详情，可参考第 10 章。假如套接字的地址族不支持 QOS，那么一旦在它上面使用该 I/O 控制命令，便会返回 WSAENOPROTOOPT 错误。该选项和 SIO_SET_QOS 都只能用在提供了支持 QOS 的传送协议的平台上，如 Windows 98、Windows Me 和 Windows 2000 及后期版本。

7.2.2.8 SIO_SET_QOS

函 数	输 入	输 出	Winsock 版本	说 明
WSAIoctl	QOS	无	2+	为指定套接字设置 QOS 属性

这个 I/O 控制命令与 SIO_GET_QOS 对应。该调用的输入参数是一个 QOS 结构，它定义了针对该套接字提出的带宽要求。这个调用不会返回任何输出值。大家可参考第 10 章，了解有关 QOS 的详

情。要注意的是，这个选项和 SIO_GET_QOS 都只能用在提供了支持 QOS 的传送协议的平台上，如 Windows 98、Windows Me 和 Windows 2000 及后期版本。

7.2.2.9　SIO_MULTIPOINT_LOOPBACK

函　数	输　入	输　出	Winsock 版本	说　明
WSAIoctl	BOOL	BOOL	2+	设置或调查多播数据是否循环返回到套接字

发送多播数据时，采取的默认行为是让发给多播组的任何数据都成为套接字接收队列内的传入数据。当然，只有在该套接字也属于目标多播组的一名成员时，这种循环返回行为才有意义。目前，这种行为只能在 IP 多播通信中见到，而 ATM 多播并不支持。要想禁止循环返回特性，可在输入参数 lpvInBuffer 中，传递一个布尔变量 FALSE。要想获取该选项当前的值，可将输入值保留为 NULL，并提供一个布尔变量，作为输出参数使用。

7.2.2.10　SIO_MULTICAST_SCOPE

函　数	输　入	输　出	Winsock 版本	说　明
WSAIoctl	int	int	2+	设置或获取多播数据的 TTL 值

这个命令控制着多播数据的存在时间，或者作用域。所谓作用域，实际是指数据最多允许路由至多少个网段。默认情况下，这个值的设定是 1。若多播数据包抵达一个路由器，其 TTL 值便会减 1。一旦 TTL 值变成了 0，那个包便会被丢弃。要想对这个值进行设定，可在 lpvInBuffer 中，传递一个目标 TTL 值(整数)；另外，可在调用 WSAIoctl 的同时，令 lpvOutBuffer 指向一个整数，就可以获取当前的 TTL 值。

7.2.2.11　SIO_KEEPLIVE_VALS

函　数	输　入	输　出	Winsock 版本	说　明
WSAIoctl	tcp_keepalive	tcp_keepalive	2+	针对每一个连接，分别将其 TCP“保持活跃”设置激活

这个 I/O 控制命令可针对每一个不同的连接，将其 TCP“保持活跃”设置激活，并且允许指定“保持活跃”的时间间隔。套接字选项 SO_KEEPALIVE 也允许启用 TCP“保持活跃”，但其发送间隔是在注册表(Registry)里设定的。若更改了注册表的值，同时也会更改计算机上所有进程“保持活跃”的间隔。但若使用这个 I/O 控制命令，便可分别针对每一个套接字，设置其“保持活跃”间隔。要想在指定的一个已建立连接的套接字上将 TCP“保持活跃”设置为有效，可将一个 tcp_keepalive 结构初始化，并将其作为输入缓冲区传递。下面是该结构的定义：

```
struct tcp_keepalive
{
    u_long onoff;
```

```
    u_long keepalivetime;
    u_long keepaliveinterval;
}
```

其中，结构字段 keepalivetime 和 keepaliveinterval 的含义和本章早些时候讨论 SO_KEEPALIVE 选项时介绍的注册表值是一样的。一旦设好“保持活跃”，便可对当前的设置值进行查询。方法是调用 WSAIoctl，同时设置 SIO_KEEPALIVE_VALS 这个 I/O 控制命令，并提供一个 tcp_keepalive 结构，作为输出缓冲区使用。这个 I/O 控制命令只能在 Windows 2000 及其后期版本上使用。

7.2.2.12 SIO_RCVALL

函 数	输 入	输 出	Winsock 版本	说 明
WSAIoctl	unsigned int	无	2+	接收网络上的所有数据包

使用这个 I/O 控制命令，同时将值设为 TRUE，便允许指定的套接字接收网络上的所有 IPv4 包。这个选项目前还未在 IPv6 套接字中实施，因此要求传递给 WSAIoctl 的套接字句柄的地址族必须是 AF_INET，套接字的类型必须是 SOCK_RAW，而协议必须是 IPPROTO_IP。除此以外，套接字必须同一个明确的本地接口建立绑定关系。换言之，我们不可将其绑定到 INADDR_ANY。一旦绑定好套接字，同时设好 I/O 控制命令后，便可调用 recv 或 WSARecv，以便返回 IPv4 数据报。要注意的是，由于这些数据是数据报，所以必须提供一个足够大的缓冲区。由于 IPv4 头表示总长的字段是一个 16 位的数值，所以理论上的最大长度为 65 535 字节。但在实际应用中，网络的最大传输单元(MTU)要小得多。这个 I/O 控制命令要求使用者在本地计算机上以管理员(Administrator)身份登录。此外，这个 I/O 命令仅适用于 Windows 2000 及其后期版本。

注意 在补充材料中，有一个名为 RCVALL.C 的示范文件，向大家阐述了如何使用这个命令，以及如何使用另外两个 SIO_RCVALL 命令。

7.2.2.13 SIO_RCVALL_MCAST

函 数	输 入	输 出	Winsock 版本	说 明
WSAIoctl	unsigned int	无	2+	接收网络上的所有多播数据包

这个 I/O 控制命令类似于上面讲述的 SIO_RCVALL。SIO_RCVALL 的使用规则也适用于这个命令，只是传递给 WSAIoctl 的套接字应当由 IPPROTO_UDP 对应的协议来创建。另一个区别是此处返回的只有 IPv4 多播数据，而非返回全部 IP 数据包。换言之，只有发至范围在 224.0.0.0 到 239.255.255.255 之间的目标地址的 IP 包才会被返回。该 I/O 控制命令仅适用于 Windows 2000 及其后期版本。

7.2.2.14　SIO_RCVALL_IGMPMCAST

函　数	输　入	输　出	Winsock 版本	说　明
WSAIoctl	unsigned int	无	2+	接收网络上的所有的 IGMP 数据包

同样，该命令类似于 SIO_RCVALL，只是传递给 WSAIoctl 的套接字应当由与 IPPROTO_IGMP 对应的协议创建。若设置这个选项，那么只会返回 IGMP 数据包。大家可参考前面对 SIO_RCVALL 的介绍，了解如何来使用这个选项。要注意的是，该 I/O 控制命令只能用于 Windows 2000 及其后期版本。

7.2.2.15　SIO_ROUTING_INTERFACE_QUERY

函　数	输　入	输　出	Winsock 版本	说　明
两者均可	SOCKADDR	无	2+	返回用于向目标地址发送数据的本地接口

利用这个 I/O 控制命令，可以找到用来向远程计算机发送数据的那个本地接口的地址。远程计算机的地址应当用 SOCKADDR 结构的形式提供，并通过 lpvInBuffer 参数来传递这个结构。除此以外，这里应该提供一个足够大的缓冲区作为 lpvOubBuffer 参数。在这个缓冲区内，将包含一个或多个 SOCKADDR 结构，对那些可供使用的接口进行描述。该命令既可用于单播端点，亦可用于多播端点，而且自该调用返回的接口可在后续的 bind 调用中使用。

Windows 2000 和 Windows XP 的即插即用特性是提供这样一个 I/O 控制命令的动机。用户可能自由地插拔 PCMCIA 网卡，从而影响应用程序能够使用的网络接口。为 Windows 2000 量身定造的应用程序应充分考虑到这方面的因素，并采取相应的对策。

因此，程序也不能仅仅依赖从 SIO_ROUTING_INTERFACE_QUERY 返回的信息。这些信息并不能保证长时期有效。要想应付这方面的情况，必须同时使用 SIO_ROUTING_INTERFACE_CHANGE 这个 I/O 控制命令，它负责在接口发生变化之后，向应用程序发出通知。一旦收到这样的通知，需要再次调用 SIO_ROUTING_INTERFACE_QUERY，获取最新信息。

注意　在补充材料中的 SIO_ROUTING_INTERFACE 目录下有一段示例代码，如何使用这些 I/O 控制命令作了说明。

7.2.2.16　SIO_ROUTING_INTERFACE_CHANGE

函　数	输　入	输　出	Winsock 版本	说　明
WSAIoctl	SOCKADDR	无	2+	与一个端点连接的接口发生变化后，发出通知

用这个 I/O 控制命令指示，在本地路由接口发生任何变化后，都向应用程序发出通知。那个接口

原本用于访问一个指定的远程地址。使用该命令时，需要在输入缓冲区中，提交一个 SOCKADDR 结构，对应于要查询的那个远程地址。对该命令的调用若成功完成，不会有任何数据返回。但是，假如同那个路由连接的接口已发生了某种形式的变化，应用程序就会及时地收到通知。随后，程序可调用 SIO_ROUTING_INTERFACE_QUERY，判断实际应使用哪一个接口。

有几种方式都可发出对这个命令的调用。假如套接字处在阻塞模式，那么除非接口发生变化，否则 WSAIoctl 调用不会完成并返回。若套接字处在非阻塞模式，便会返回 WSAEWOULDBLOCK 错误。随后，应用程序可通过 WSAEventSelect 或 WSAAsyncSelect，同时在网络事件位掩码中设置 FD_ROUTING_INTERFACE_CHANGE 标志，等候“路由变化”事件的发生。另外，重叠 I/O 也可用来进行这样的调用。通过这种方法，我们可为 WSAOVERLAPPED 结构提供一个事件句柄，它会在路由变化后收到通知。

输入结构 SOCKADDR 中指定的地址可以是一个特定的地址，亦可使用通配地址：INADDR_ANY，表明发生任何路由更改都会接到通知。要注意的是，由于路由信息相对来说比较稳定，不大会发生变化，所以提供程序具有一种这样的选项，忽略应用程序在输入缓冲区内提供的信息，只在接口发生变化后，才发出通知。因此，一种更好的做法，或许是先注册任意变化的通知，然后简单地调用 SIO_ROUTING_INTERFACE_QUERY，看看发生的改变是否会影响到应用程序。

7.2.2.17 SIO_ADDRESS_LIST_QUERY

函 数	输 入	输 出	Winsock 版本	说 明
WSAIoctl	无	SOCKET_ADDRESS_LIST	2+	返回套接字能够绑定的接口列表

这个 I/O 控制命令用于获取与套接字协议家族相匹配的本地传送地址列表，应用程序可与这些地址建立绑定关系。此时的输出缓冲区是一个 SOCKET_ADDRESS_LIST 结构，定义如下：

```
typedef struct _SOCKET_ADDRESS_LIST
{
    INT                iAddressCount;
    SOCKET_ADDRESS     Address[1];
}SOCKET_ADDRESS_LIST,FAR * LPSOCKET_ADDRESS_LIST;
typedef struct _SOCKET_ADDRESS
{
    LPSOCKADDR         lpSockaddr;
    INT                iSockaddrLength;
}SOCKET_ADDRESS,*PSOCKET_ADDRESS,FAR * LPSOCKET_ADDRESS;
```

其中，iAddressCount 字段用于返回列表中地址结构的数量，而 Address 字段对应的是一个数组，包含了与协议家族相符的一系列地址。

在 Windows 即插即用环境中，有效地址的数量可能会发生动态变化。因此，依赖这个 I/O 控制命

令返回的信息，应用程序不能始终保持恒定。要想将动态改变的因素考虑在内，应用程序首先应该调用 SIO_ADDRESS_LIST_QUERY，取得最新的接口信息，然后调用 SIO_ADDRESS_LIST_CHANGE，以便收到与未来的变化有关的通知。一旦地址列表发生改变，应用程序就应重新查询一遍。

假如提供的输出缓冲区不够大，WSAIoctl 调用便会失败，并返回 WSAEFAULT 错误。同时，lcbBytesReturned 参数会指出到底多大的缓冲区才够用。该 I/O 控制命令仅适用于 Windows 2000 及其后期版本。

注意　在补充材料的 SIO_ADDRESS_LIST 目录下有一段示例代码，对如何使用这些 I/O 控制命令作了说明。

7.2.2.18　SIO_ADDRESS_LIST_SORT

函数	输入	输出	Winsock 版本	说明
WSAIoctl	SOCKET_ADDRESS_LIST	SOCKET_ADDRESS_LIST	2+	按优先顺序对地址进行排序

这个选项接受从 SIO_ADDRESS_LIST_QUERY 命令返回的结构 SOCKET_ADDRESS LIST，对地址进行排序，如果是 IPv6 地址就填入适当的范围 ID。应注意到，如果列表中有一个全球地址且这个全球地址属于其中一个全球地点前缀，则范围 ID 仅为站点一本地地址而设置。该 I/O 控制命令仅适用于 Windows XP 及其后期版本。

7.2.2.19　SIO_ADDRESS_LIST_CHANGE

函　数	输　入	输　出	Winsock 版本	说　明
WSAIoctl	无	无	2+	本地接口发生变化时，发出通知

通过该命令，一旦给定的套接字协议家族的本地传送地址列表发生变化(应用程序可与这些地址建立绑定关系)，程序便会收到通知。若调用成功完成，在输出参数中，不会返回任何信息。

有几个办法可发出对该命令的调用。假如套接字处于阻塞模式，那么除非接口真的发生了某种变化，否则 WSAIoctl 调用是不会完成并返回的。若套接字处于非阻塞模式，则返回 WSAEWOULDBLOCK 错误。随后，应用程序可利用 WSAEventSelect 或者 WSAAsyncSelect 调用，同时在网络事件位掩码中设置 FD_ADDRESS_LIST_CHANGE 标志，从而等候“路由变化”事件的发生。另外，亦可通过重叠 I/O 发出这样的调用。采用这个方法时，需要为 WSAOVERLAPPED 结构提供一个事件句柄；一旦路由发生改变，那个句柄便会进入已传信状态。该 I/O 控制命令仅适用于 Windows 2000 及其后期版本。

7.2.2.20 SIO_GET_INTERFACE_LIST

函 数	输 入	输 出	Winsock 版本	说 明
WSAIoctl	无	INTERFACE_INFO[]	2+	返回本地接口列表

这个 I/O 控制命令定义于 Ws2tcpip.h 头文件内，用于返回与本地计算机上每个接口有关的信息。在输入方面，不需要设置任何东西；而在输出参数中，会返回由 INTERFACE_INFO 结构构成的一个数组。该结构的定义如下：

```
typedef struct _INTERFACE_INFO
{
    u_long          iiFlags;                    /*接口标志*/
    sockaddr_gen    iiAddress;                  /*接口地址*/
    sockaddr_gen    iiBroadcastAddress;         /*广播地址*/
    sockaddr_gen    iiNetmask;                  /*网络掩码*/
}INTERFACE_INFO,FAR *LPINTERFACE_INFO;

#define IFF_UP                  0x00000001          /*接口上行*/
#define IFF_BROADCAST       0x00000002          /*广播得到支持*/
#define IFF_LOOPBACK        0x00000004          /*这是环回接口*/
#define IFF_POINTTOPOINT    0x00000008          /*这是点到点接口*/
#define IFF_MULTICAST       0x00000010          /*多播得到支持*/
typedef union sockaddr_gen
{
    struct sockaddr Address;
    struct sockaddr_in AddressIn;
    struct sockaddr_in6 AddressIn6;
}sockaddr_gen;
```

其中，iiFlags 字段可返回一个标志掩码，指出接口是否可用(IFF_UP)，以及接口支持广播通信(IFF_BROADCAST)还是多播通信(IFF_MULTICAST)。此外，它也会指出接口是处于环回模式(IFF_LOOPBACK)，还是处于点到点通信模式(IFF_POINTTOPOINT)。其他 3 个字段，则分别包含了接口地址、广播地址以及相应的网络掩码。

7.2.2.21 SIO_GET_INTERFACE_LIST_EX

函 数	输 入	输 出	Winsock 版本	说 明
WSAIoctl	无	INTERFACE_INFO_EX[]	2+	返回本地接口列表

这个 I/O 控制命令和 SIO_GET_INTERFACE_LIST 相似，不过返回的结构包含一个嵌入的 SOCKET_ADDRESS 结构，这个嵌入结构用来描述每个本地接口，和 SOCKADDR_GEN 结构相对。它消除了套接字地址结构大小的从属性。INTERFACE_INFO_EX 结构的定义为：

```
typedef struct _INTERFACE_INFO_EX
```

```
{
    u_long            iiFlags;              /*接口标志*/
    SOCKET_ADDRESS    iiAddress;            /*接口地址*/
    SOCKET_ADDRESS    iiBroadcastAddress;   /*广播地址*/
    SOCKET_ADDRESS    iiNetmask;            /*网络掩码*/
} INTERFACE_INFO_EX, FAR *LPINTERFACE_INFO_EX;
```

所有字段含义与前面叙述的 INTERFACE_INFO 结构中的字段相同。这个 I/O 控制命令仅适用于 Windows XP 及其后期版本。

7.2.2.22　SIO_GET_MULTICAST_FILTER

函　数	输　入	输　出	Winsock 版本	说　明
WSAIoctl	无	struct ip_msfilter	2+	返回套接字上设置的多播筛选器

这个 I/O 控制命令获取给定套接字上设置的多播筛选器。多播状态由 SIO_SET_MULTICAST_FILTER I/O 控制命令设定。这个 I/O 控制命令要求一个支持 IGMPv3 的网络，且仅适用于 Windows XP。关于多播的更多内容请参见第 9 章。

7.2.2.23　SIO_SET_MULTICAST_FILTER

函　数	输　入	输　出	Winsock 版本	说　明
WSAIoctl	struct ip_msfilter	无	2+	在套接字上设置多播筛选器

这个 I/O 控制命令设置多播状态。输入参数是一个 ip_msfilter 结构，其定义为：

```
struct ip_msfilter {
    struct in_addr    imsf_multiaddr;
    struct in_addr    imsf_interface;
    u_long            imsf_fmode;
    u_long            imsf_numsrc;
    struct in_addr    imsf_slist[1];
};
```

第 1 个字段是待加入的多播地址。第 2 个字段是本地接口，从它上边加入多播组。imsf_fmode 字段指明筛选器状态是包含的还是排除的，这两种状态分别由 MCAST_INCLUDE 和 MCAST_EXCLUDE 定义。imsf_numsrc 字段表示 imsf_slist 数组中包含的 IPv4 源地址的数量。

对 IP_ADD_SOURCE_MEMBERSHIP、IP_DROP_SOURCE_MEMBERSHIP、IP_BLOCK_SOURCE 和 IP_UNBLOCK_SOURCE 的多个调用，可以用这个 I/O 控制命令代替，从而在单个调用中设置多播状态。这个命令要求网络支持 IGMPv3，且适用于 Windows XP。更多内容参见第 9 章。

7.2.2.24 SIO_INDEX_BIND

函 数	输 入	输 出	Winsock 版本	说 明
WSAIoctl	int	无	2+	将套接字绑定到给定接口索引

这个 I/O 控制命令将套接字绑定到接口索引，将接口索引作为一个输入参数，而不是将地址作为输入参数。这个 I/O 控制命令仅 Windows 2000 及其后期版本支持。

7.2.2.25 SIO_INDEX_MCASTIF

函 数	输 入	输 出	Winsock 版本	说 明
WSAIoctl	int	无	2+	将多播发送接口设置到指定索引

这个 I/O 控制命令为多播通信设置外出接口，它将接口索引作为一个输入参数，而不是将地址作为输入参数。这个 I/O 控制命令仅 Windows 2000 及其后期版本支持。

7.2.2.26 SIO_INDEX_ADD_MCAST

函 数	输 入	输 出	Winsock 版本	说 明
WSAIoctl	struct ip_mreq	无	2+	在指定接口索引上加入一个多播组

这个 I/O 控制命令通过一个接口索引而不是地址来加入一个多播组。输入参数是一个 ip_mreq 结构，不过 imr_interface 字段包含的是接口索引。更多内容参见第 9 章。这个 I/O 控制命令仅 Windows 2000 及其后期版本支持。

7.2.2.27 SIO_INDEX_DEL_MCAST

函 数	输 入	输 出	Winsock 版本	说 明
WSAIoctl	struct ip_mreq	无	2+	从指定接口索引上撤消一个多播组

这个 I/O 控制命令从选定的接口索引上撤消指定多播组的成员。这个 I/O 控制命令仅 Windows 2000 及其后期版本支持。更多内容参见第 9 章。

7.2.2.28 SIO_NSP_NOTIFY_CHANGE

函 数	输 入	输 出	Winsock 版本	说 明
WSANSPIoctl	无	无	2+	当一个命名空间查询不再有效时发出通知

这个 I/O 控制命令用于在可用网络改变时接收通知。这个 I/O 控制命令仅得到 Windows XP 及其后期版本的支持。关于这个 I/O 控制命令的更多内容参见第 9 章。

7.2.2.29　SIO_QUERY_TARGET_PNP_HANDLE

函　数	输　入	输　出	Winsock 版本	说　明
WSAIoctl	无	SOCKET	2+	返回下层提供程序的 SOCKET 句柄

这个 I/O 控制命令在下层提供程序中查询一个句柄，这个句柄可用来接收即插即用事件通知。这个 I/O 控制命令仅得到 Windows 2000 及其后期版本的支持。

7.2.2.30　SIO_UDP_CONNRESET

函　数	输　入	输　出	Winsock 版本	说　明
WSAIoctl	BOOL	无	2+	使 ICMP 错误可以广播到套接字

在默认情况下，任何在一个 UDP 套接字上发送的由通信产生的 ICMP 错误都不会被广播到套接字。例如，如果数据被送到一个没有套接字监听的端点，则会返回一个 ICMP 错误，如果把这个 I/O 控制命令设置为 TRUE，这些错误就会被广播到发送套接字，并且使用的是一种 WSAECONNRESET 错误的形式。这个 I/O 控制命令仅得到 Windows 2000 及其后期版本的支持。

7.2.3　加密套接字协议层的 I/O 控制命令

SSL(Secure Socket Layer，加密套接字协议层)命令目前只得到了 Windows CE 的支持。Windows 95、Windows 98、Windows Me、Windows NT、Windows 2000 及 Windows XP 均未包含支持 SSL 的提供程序。除仅适用于 Windows CE 之外，支持这些选项的惟一 Winsock 版本只有 Winsock 1.

7.2.3.1　SO_SSL_GET_CAPABILITIES

函　数	输　入	输　出	说　明
WSAIoctl	无	DWORD	返回 Winsock 加密提供程序的功能

该命令负责接收一系列特殊的标志，这些标志指出 Windows 套接字加密提供程序都具有哪些方面的能力。输出缓冲区必须是指向一个 DWORD(双字)位字段的一个指针。目前，仅定义了 SO_CAP_CLIENT 标志。

7.2.3.2　SO_SSL_GET_FLAGS

函　数	输　入	输　出	说　明
WSAIoctl	无	DWORD	返回与套接字关联的的 s 信道特有标志

这个命令可获取与一个特定套接字关联在一起的、s 信道所特有的标志。输出缓冲区必须是一个指针，指向一个 DWORD 位字段。请参考下述的 SO_SSL_SET_FLAGS，了解哪些才是有效标志。

7.2.3.3 SO_SSL_SET_FLAGS

函 数	输 入	输 出	说 明
WSAIoctl	DWORD	无	设置套接字的 s 信道特有的标志

这儿的输入缓冲区必须是一个指针，指向一个 DWORD 位字段。目前，仅设置了 SSL_FLAG_DEFER_HANDSHAKE 标志，它允许应用程序在切换到密文之前，先进行明文数据的收发。若通过代理服务器通信，便必须设置这个标志。

通常，Windows 套接字加密提供程序会在 Windows 套接字的 connect 函数中执行加密信号交换。然而，如果设置了这个标志，信号交换便会推迟到应用程序执行了 SO_SSL_PERFORM_HANDSHAKE 控制代码之后进行。完成信号交换后，这个标志会被重置。

7.2.3.4 SO_SSL_GET_PROTOCOLS

函 数	输 入	输 出	说 明
WSAIoctl	无	SSLPROTOCOLS	返回加密提供程序支持的协议列表

该命令可取得一系列协议构成的列表，所有协议均是这个套接字上的提供程序所支持的。输出缓冲区必须是一个指针，指向一个 SSLPROTOCOLS 结构。下面是该结构的定义：

```
typedef struct _SSLPROTOCOL
{
    DWORD dwProtocol;
    DWORD dwVersion;
    DWORD dwFlags;
}SSLPROTOCOL,*LPSSLPROTOCOL;
typedef struct _SSLPROTOCOLS
{
    DWORD       dwCount;
    SSLPROTOCOL ProtocolList[1];
}SSLPROTOCOLS, FAR *LPSSLPROTOCOLS;
```

对 dwProtocol 字段来说，有效的协议包括 SSL_PROTOCOL_SSL2、SSL_PROTOCOL_SSL3 和 SSL_PROTOCOL_PCT1。

7.2.3.5 SO_SSL_SET_PROTOCOLS

函数	输入	输出	说明
WSAIoctl	SSLPROTOCOLS	无	设置某层提供者应当支持的一个协议列表

这个 I/O 控制命令指定协议的一个列表，其中均是提供程序准备在这个套接字上支持的协议。输入缓冲区必须是一个指针，指向前述的 SSLPROTOCOLS 结构。

7.2.3.6　SO_SSL_SET_VALIDATE_CERT_HOOK

函　数	输　入	输　出	说　明
WSAIoctl	SSLVALIDATECERTHOOK	无	为 SSL 身份证书的接受设置校验函数

该 I/O 控制命令可设置一个指针，指向套接字的身份证书的验证挂钩。它用于指定一个回叫函数，从远程通信方收到一系列证书后，由 Windows 套接字加密提供程序调用该函数。在此，输入缓冲区必须是一个指针，指向 SSLVALIDATECERTHOOK 结构，如下所示：

```
typedef struct
{
    SSLVALIDATECERTFUNC    HookFunc;
    LPVOID                 pvArg;
}SSLVALIDATECERTHOOK,*PSSLVALIDATECERTHOOK;
```

其中，HookFunc 字段是指向一个用于证书验证的回叫函数的指针；pvArg 是指向应用程序专用数据的一个指针，可由应用程序用于任何目的。

7.2.3.7　SO_SSL_PERFORM_HANDSHAKE

函　数	输　入	输　出	说　明
WSAIoctl	无	无	在已建立连接的套接字上开始加密信号交换操作

这个 I/O 控制命令可在一个已连接的套接字上初始化加密信号交换序列；其中，SSL_FLAG_DEFER_HANDSHAKE 标志已在连接前设置好了。数据缓冲区并非是必需的，但 SL_FLAG_DEFER_HANDSHAKE 标志会被重置。

7.2.4　ATM I/O 控制命令

本小节列出的 I/O 控制命令是 ATM 协议家族特有的。它们非常基本，主要涉及对 ATM 设备数量的调查，以及获取本地接口的 ATM 地址，要了解 ATM 地址机制的详情请参阅第 4 章。

7.2.4.1　SIO_GET_NUMBER_OF_ATM_DEVICES

函　数	输　入	输　出	Winsock 版本	说　明
WSAIoctl	无	DWORD	2+	返回 ATM 适配器的数量

这个 I/O 控制命令可用一个 DWORD(双字)填写由 lpvOutBuffer 指向的输出缓冲区，该 DWORD 指出系统内总共包含了多少个 ATM 设备。每个设备都有一个惟一的 ID 标识，范围在 0～(该命令返回的数字-1)之间。

7.2.4.2 SIO_GET_ATM_ADDRESS

函 数	输 入	输 出	Winsock 版本	说 明
WSAIoctl	DWORD	ATM_ADDRESS	2+	为指定设备返回 ATM 地址

该 I/O 控制命令可获取与指定设备关联在一起的本地 ATM 地址。在输入缓冲区中，需要为这个命令指定类型为 DWORD 的一个设备 ID；而在由 lpvOutBuffer 参数指向的那个输出缓冲区中，会填入一个 ATM_ADDRESS 结构，其中包含了可用于后续 bind 调用的一个本地 ATM 地址。

7.2.4.3 SIO_ASSOCIATE_PVC

函 数	输 入	输 出	Winsock 版本	说 明
WSAIoctl	ATM_PVC_PARAMS	无	2+	将套接字与一个永久虚拟回路关联起来

这个 I/O 控制命令可将套接字与一个 PVC(Permanent Virtual Circuit，永久虚拟回路)关联到一起。该命令在输入缓冲区中指定，缓冲区包含了 ATM_PVC_PARAMS 结构。套接字应属于 AF_ATM 地址族。该函数成功返回之后，应用程序便可以开始数据的收发，好像连接已经建好(其实是“虚拟”的)一样。

ATM_PVC_PARAMS 结构的定义如下：

```
typedef struct
{
    ATM_CONNECTION_ID PvcConnectionId;
    QOS               PvcQos;
}ATM_PVC_PARAMS;

typedef struct
{
    DWORD DeviceNumber;
    DWORD VPI;
    DWORD VCI;
}ATM_CONNECTION_ID;
```

7.2.4.4 SIO_GET_ATM_CONNECTION_ID

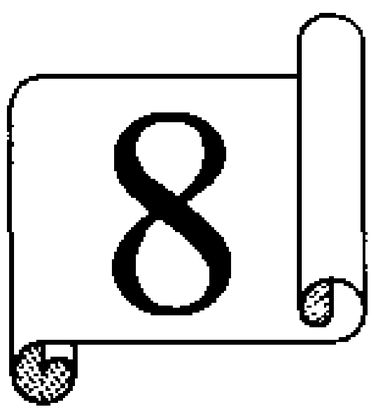
函 数	输 入	输 出	Winsock 版本	说 明
两者均可	无	ATM_CONNECTION_ID	2+	判断是否已经读取了 OOB 数据

这个 I/O 控制命令可获取同套接字关联在一起的 ATM 连接 ID。该函数成功返回后，由 lpvOutBuffer 参数指向的那个输出缓冲区会填入一个 ATM_CONNECTION_ID 结构，其中包含了设备

编号以及对应的 VPI/VCI(Virtual Path/Channel Identifier，虚拟路径/通道标识符)值，它们均是早先在 SIO_ASSOICATE_PVC 的条目中定义好的。

7.3　小　　结

从表面看，数量如此众多的套接字选项及 I/O 控制命令会使人觉得非常繁琐。但是，只需深入探索便会发现，通过它们，可访问具体协议独有的许多特性，同时还可利用它们对应用程序进行细致的调节。在某些情况下，程序必须使用一个或多个套接字选项及 I/O 控制命令，否则便无法继续工作，例如在使用 AppleTalk 或 IrDA 的场合。当然，大多数时候，应用程序每次只需使用少数几个选项便足够了。另外要注意的是，套接字选项及 I/O 控制命令最麻烦的一点在于，并非每个选项或 I/O 控制命令都能够适用于所有版本的 Windows 平台。这样，一旦涉及跨平台的兼容问题，应用程序的设计便会有不少困难。



名称注册和解析

本章将全面论述 Winsock 2 中引入的名称注册和解析模型，它们都是与协议无关的。由于现在已经丢弃了 Winsock 1 中引入的 GetService 和 SetService，所以我们将不再对它们进行讨论。这一章首先介绍名称注册和解析的重要性及其用法的背景知识，然后逐步深入讲解现有的各种不同的名称注册模型，接着介绍 Winsock 2 中用于名称解析的函数。另外，还将谈谈如何注册自己的服务以供他人查询，如何查询 DNS 名称及如何使用 NLA 服务查询 IP 网络名称等。

8.1 背 景 知 识

名称注册是一个过程，它把一个用户友好的名称和具体协议地址关联在一起。主机名及其 IP 地址便是例证。人们发现要记住一个工作站的地址(例如 157.54.185.186)非常麻烦。所以他们宁愿把自己的计算机命名为一个更容易记的地址，例如“MP3Server”。在使用 IP 的情况下中，DNS 会把 IP 地址映射成相应的名称。其他协议也提供了将特定地址注册为友好名称的方法。我们将在下一节详细地讨论命名空间。

人们不仅希望能够注册和解析主机名，还希望能够映射自己的 Winsock 服务器地址，以便于在客户机打算和服务器连接时，可获得服务器的地址。比方说，有一个服务器，它运行的计算机地址为 157.64.185.186，端口为 5000。如果它只在那台计算机上运行，就可以把这台服务器的地址放到客户机应用程序的代码中。但如果需要一个更为动态的服务器，让它在若干台计算机上运行时，就要考虑采用容错的分布式应用程序。如果一个服务器崩溃或过于繁忙，另一个就在某个地方开始接替它，为客户机提供服务。这种情况下，要找到服务器实际上在哪个地址运行，是非常令人头疼的。理想的情况是，用若干个地址来注册自己的服务器，如将它命名为“Fault-Tolerant Distributed Server”。另外，大家也许还希望动态更新已注册的服务及其地址。这便是名称注册和解析所涉及的内容，而本章将着重讨论 Winsock 提供的一些适用于分布式服务器注册和名称解析的程序。

8.2　命名空间模型

在深入探讨 Winsock 函数之前，需要讲讲大多数协议附带的各种命名空间模型。命名空间提供了一种功能，用一个友好名把具体的协议及其寻址属性关联在一起。最常见的命名空间是针对 IP 的 DNS 和 Novell 针对 IPX 开发的 NDS(NetWare Directory Services，NetWare 目录服务)。这些命名空间在组成和实现方面各不相同，但它们的有些属性对于理解如何通过 Winsock 注册和解析名称特别重要。

命名空间有 3 种类型：动态命名空间、静态命名空间和固定命名空间。动态命名空间允许人们实时注册服务。另外，还意味着客户机可以在运行时对这个服务进行查看。一般说来，动态命名空间需要周期性地广播服务信息，表示该服务可继续使用。用于 NetWare 环境的 SAP 和 AppleTalk 的 NBP 命名空间都属于动态命名空间。

在 3 类命名空间中，静态命名空间的灵活性最小。在静态命名空间内注册一个服务，需要在规定时间内进行手动注册。因为静态命名空间只有一种解析名称的方法，这意味着无法通过 Winsock 用它注册一个服务名。DNS 是一个静态命名空间。举个例子来说，可以用 DNS 手动地把 IP 地址和主机名输入一个文件，DNS 服务利用这个文件来处理解析请求。Windows XP 平台支持动态 DNS，但 Winsock 接口通常不提供更新 DNS 的方法。

固定命名空间和动态命名空间一样，允许实时注册服务。但和动态命名空间不同的是，固定命名空间把注册信息保留在固定的地方，例如磁盘上的一个文件中。只有在服务请求将自己删除时，固定命名空间才会把这项服务条目删除。固定命名空间的优点在于灵活，不会连续不断地广播任何一种类型的可用信息。缺点就是如果一个服务表现欠佳(或者说编得较差)，这个服务便会在不通知命名空间提供程序删除其服务条目的情况下，自动终结，从而导致客户机错误地认为该服务仍然可用。NDS 就是一种固定命名空间。

命名空间的列举

现在，大家已经知道命名空间的各种属性，现在来看一下如何得知一台计算机上可用哪些命名空间。多数预先定义的命名空间的声明都在 NSPADI.H 头文件中。每个命名空间都有一个分配所得的整数值。表 8.1 列出了一些比较常见的命名空间，它们已获得普遍支持，可用于 Windows 平台。返回什么样的命名空间取决于工作站上安装的协议。比方说，如果一个工作站上没有安装 IPX/SPX，就不会返回 NS_SAP 命名空间。

在一台计算机上安装 IPX/SPX 时，只支持 SAP 命名空间查询。如果想注册自己的服务，还需要安装“SAP 代理服务”(SAP Agent Service)。某些情况下，需要“NetWare 客户机服务”(Client Service for NetWare)把本地 IPX 接口地址准确无误地显示出来。如果没有这个服务，本地地址将全部以 0 的形式出现。另外还必须增加一个 NDS 客户机，以便利用 NDS 命名空间。所有这些协议和服务都可通

过“网络控制面板”(Network Control Panel)来添加。

表 8.1 已获支持的命名空间

命名空间	值	说 明
NS_SAP	1	SAP 命名空间：用于 IPX 网络
NS_NDS	2	NDS 命名空间：也用于 IPX 网络
NS_DNS	11	DNS 命名空间：多见于 TCP/IP 网络和 Internet
ND_NTDS	32	Windows NT 域名空间：运行于 Windows 2000 及 Windows XP 的与协议无关的命名空间

Winsock 2 提供了一种方法，可以通过编程获取系统上所有可用的命名空间的列表。这是通过调用 WSAEnumNameSpaceProviders 函数来完成的。该函数的定义如下：

```
INT WSAEnumNameSpaceProviders (
    LPDWORD                 lpdwBufferLength,
    LPWSANAMESPACE_INFO  lpnspBuffer
);
```

第 1 个参数是作为 lpnspBuffer 提交的缓冲区的长度，它是由多个 WSANAMESPACE_INFO 结构组成的一个大型数组。如果该函数是通过一个不够大的缓冲区调用的，就会失败，这将把 lpdwBufferLength 设为所需要的最小长度，并导致 WSAGetLastError 返回 WSAEFAULT 错误。这个函数将把已返回的 WSANAMESPACE_INFO 结构的数量返回，或在出现错误时返回 SOCKET_ERROR。WSANAMESPACE_INFO 结构描述在指定计算机上安装的一个独立命名空间，定义如下：

```
typedefstruct_WSANAME SPACE_INFO{
   GUIDN                SProviderId;
   DWORD                dwNameSpace;
   BOOL                 fActive;
   DWORD                dwVersion;
   LPTSTR                   lpszIdentifier;
}WSANAMESPACE_INFO,*  PWSANAMESPACE_INFO,
*LPWSANAMESPACE_INFO;
```

事实上，这个结构有两种形式——UNICODE 和 ANSI。根据建立项目时所采用的方式，Winsock 2 头文件类型会相应地将 WSANAMESPACE_INFO 定义为适当的结构。在实际应用中，所有结构和 Winsock 2 的注册和名称解析函数都有 UNICODE 和 ANSI 两个版本。该结构的第 1 个成员 NSProviderId，是一个通用的惟一标识符(GUID)，它对这个特殊的命名空间进行描述。dWNameSpace 字段是这个命名空间的整数常量，例如 NS_DNS 或 NS_SAP。fActive 成员是一个布尔值，若为真，就表明该命名空间可用，并准备开始查询；反之则表示提供程序未被激活，不能进行专门引用了该提

供程序的查询。dwVersion 字段只对这个提供程序的版本进行标识。最后的 lpszIdentifier 字段是该提供程序的一个描述性字符串标识符。

8.3　服务的注册

下一步便是如何设置自己的服务，使其可用，并让网络上的其他计算机都知道它。这就要利用命名空间提供程序注册一项服务，这样，打算与之通信的客户机就可以对它进行声明或查询。注册一项服务实际上只有两步。第 1 步是安装描述服务特征的**服务类**(service class)。

分清服务类和实际的服务之间的区别非常重要的。服务类和服务对应两个不相同的概念。比方说，服务类描述哪些命名空间可用来注册服务，该服务的特征有哪些(比如它是面向连接的，还是无连接的)。至于客户机如何才能建立连接，服务类是无法将其描述出来的。一旦完成了服务类的注册，便可注册实际的服务实例了，此时要引用服务所属的那个准确的服务类。完成这些事情后，客户机就发出请求，查找服务实例运行的地方，从而尝试与别的计算机进行通信。

8.3.1　安装服务类

在注册一个服务实例时，需要定义该服务属于哪个服务类。用哪个命名空间注册这个服务类中的服务，由服务类定义。注册服务类的 Winsock 函数是 WSAInstallServiceClass，它的定义如下：

```
INT WSAInstallServiceClass (LPWSASERVICECLASSINFO lpServiceClassInfo);
```

惟一的参数是 lpServiceClassInfo，它指向 WSASERVICECLASSINFO 结构，这个结构定义了服务类的属性，它的定义为：

```
typedef struct _WSAServiceClassInfo {
    LPGUID            lpServiceClassId;
    LPTSTR            lpszServiceClassName;
    DWORD             dwCount;
    LPWSANSCLASSINFO  lpClassInfos;
}WSASERVICECLASSINFO, *PWSASERVICECLASSINFO, *LPWSASERVICECLASSINFO;
```

第 1 个字段是 GUID，它对这个特殊的服务类进行惟一性标识。要在这里使用 GUID，有两种方法可以生成。一是利用公用程序 UNIDGEN.EXE 为这个服务器类建立一个 GUID。采用这种方法的问题是：如果需要再次引用这个 GUID，就必须把它的值硬编码到头文件中的某个地方。鉴于这一点，另一种方法应运而生。在头文件 SVCGUID.H 内，有几个宏可生成一个具有简单属性的 GUID。比方说，如果安装 SAP 服务类(用于广告 IPX 应用程序)，就可以使用宏 SVCID_NETWARE。惟一的参数是为应用程序类分配的 SAP ID 编号。注意 SAP ID 编号是在 NetWare 中预先定义好的，例如，0x4 表示文件服务器，而 0x7 则表示打印服务器。若采用这种方法，只需一个易于记忆的 SAP ID，利用它生成相应服务类的 GUID。另外，有几个宏把端口号当作一个参数来接收，并返回相应服务的 GUID。

现在来看看头文件 SUCGUIDE.H，其中包含一些针对逆向操作的宏协议——从 GUID 中取出服务端口号。GUID 是利用宏通过诸如端口号或 SAP ID 之类的属性生成的，表 8.2 列出了其中最常用的几个宏。头文件中还包括了一些已知端口号(为 FTP 和 Telnet 之类的服务预留的)的常量。

表 8.2 常用的服务 ID 宏

宏	说 明
SVCID_TCP(Port)	通过 TCP 端口号生成一个 GUID
SVCID_DNS(RecordType)	通过 DNS 记录类型生成一个 GUID
SVCID_UDP(Port)	通过 UDP 端口号生成一个 GUID
SVCID_NETWARE(SapId)	通过 SAP ID 号生成一个 GUID

WSASERVICECLASSINFO 结构的第 2 个字段是 lpszServiceClassName，它仅仅是这个特定服务类的一个字符串名。最后两个字段是相关的：dwCount 字段引用一个数值，表示传递给 lpClassInfos 字段的 WSANSCLASSINFO 结构的数量。这些结构定义的是在这个服务类下注册的服务所适用的命名空间和协议特征。该结构的定义为：

```
typedef struct _WSANSClassInfo{
LPSTR    lpszName;
DWORD    dwNameSpace;
DWORD    dwValueType;
DWORD    dwValueSize;
LPVOID   lpValue;
}WSANSCLASSINFO,*PWSANSCLASSINFO,*LPWSANSCLASSINFO;
```

lpszName 字段定义服务类具有的属性。表 8.3 列出了可用的各种属性。其中每个属性都有一个 REG_DWORD 数值类型。

表 8.3 服务类型

字符串值	定义的常量	命名空间	说 明
“SapId”	SERVICE_TYPE_VALUE_SAPID	NS_SAP	SAP 标识符
“ConnectionOriented”	SERVICE_TYPE_VALUE_CONN	任何一种	指明服务是面向连接的，还是无连接的
“TcpPort”	SERVICE_TYPE_VALUE_TCPPORT	NS_DNS NS_NTDS	TCP 端口

续表

字符串值	定义的常量	命名空间	说　明
“UdpPort”	SERVICE_TYPE_VALUE_UDPPORT	NS_DNS NS_NTDS	UDP 端口

dwNameSpace 是这个属性所用的命名空间。表 8.3 列出了各种服务类型通常使用的命名空间。后 3 个字段 dwValueType、dwValueSize 和 lpValue，都对与服务类型关联的值进行了描述。dwValueType 字段表示与这个条目关联的数据的类型，所以可算得上一种注册类型值。例如，这个值如果是 DWORD，那么值的类型就应该是 REG_DWORD。下一个字段 dwValueSize，仅仅是被当作 lpValue 传递的那些数据的大小，lpValue 是一个指向数据的指针。

至于如何安装一个名为“WidgetServerClass”的服务类，下面的代码示例对此进行了说明：

```
WSASERVICECLASSINFO   sci;
WSANSCLASSINFO        aNameSpaceClassInfo[4];
DWORD                 dwSapId = 200,
                      dwUdpPort= 5150,
                      dwZero = 0;
int                   ret ;

memset(&sci,0,sizeof(sci));

SET_NETWARE_SVCID(&sci.lpServiceClassId,dwSapId);
sci.lpszServiceClassName = (LPSTR)"WidgetServer Class";
sci.dwCount = 4;
sci.lpClassInfos = aNameSpaceClassInfo;

memset(aNameSpaceClassInfo,0,sizeof(WSANSCLASSINFO)*4);
//NTDS 命名空间设置
aNameSpaceClassInfo[0].lpszName = SERVICE_TYPE_VALUE_CONN;
aNameSpaceClassInfo[0].dwNameSpace = NS_NTDS;
aNameSpaceClassInfo[0].dwValueType = REG_DWORD;
aNameSpaceClassInfo[0].dwValueSize = sizeof(DWORD);
aNameSpaceClassInfo[0].lpValue = &dwZero;
aNameSpaceClassInfo[1].lpszName = SERVICE_TYPE_VALUE_UDPPORT;
aNameSpaceClassInfo[1].dwNameSpace = NS_NTDS;
aNameSpaceClassInfo[1].dwValueType = REG_DWORD;
aNameSpaceClassInfo[1].dwValueSize = sizeof(DWORD);
aNameSpaceClassInfo[1].lpValue = &dwUdpPort;

//SAP 命名空间设置
aNameSpaceClassInfo[2].lpszName = SERVICE_TYPE_VALUE_CONN;
aNameSpaceClassInfo[2].dwNameSpace = NS_SAP;
```

```
aNameSpaceClassInfo[2].dwValueType = REG_DWORD;
aNameSpaceClassInfo[2].dwValueSize = sizeof(DWORD);
aNameSpaceClassInfo[2].lpValue = &dwZero;
aNameSpaceClassInfo[3].lpszName = SERVICE_TYPE_VALUE_SAPID;
aNameSpaceClassInfo[3].dwNameSpace = NS_SAP;
aNameSpaceClassInfo[3].dwValueType = REG_DWORD;
aNameSpaceClassInfo[3].dwValueSize = sizeof(DWORD);
aNameSpaceClassInfo[3].lpValue = &dwSapId;

WSAInstallServiceClass(&sci);
```

首先要注意的是，这段代码采用了一个 GUID，上面的服务类将在这个 GUID 下注册。您设计的服务都属于 Widget Server Class 这个类，而这个服务类描述的则是常见的属性，这些属性属于这种服务的一个实例。在这个示例中，我们选用值为 200 的 NetWare SAP ID 来注册服务类。这样做只是为了方便。其实，我们可以用一个任意的 GUID 或基于 UDP 端口号的 GUID。另外，该服务可以使用 UDP 协议，这时客户机在端口 5150 上进行监听。

下一步应该注意的，是将 WSASERVICECLASSINFO 结构的 dwCount 字段设为 4。这个示例，将用 SAP 命名空间 (NS_SAP) 和 Windows NT 域名空间 (NS_NTDS) 来注册服务类。其余需要注意的是：尽管只用了两个命名空间来注册这个服务类，但这里却用了 4 个 WSANSCLASSINFO 结构。这样做的原因是，我们为每个命名空间都定义了两个属性，而每个属性都需要一个独立的 WSANSCLASSINFO 结构。针对每个命名空间，我们定义了服务是否将面向连接。在这个示例中，因为我们把 SERVICE_TYPE_VALUE_CONN 的值设成了一个布尔值 0，所以命名空间是无连接的。我们还针对 Windows NT 域名空间，使用服务类型 SERVICE_TYPE_VALUE_UDPPORT 设置了 UDP 端口号，这个服务通常在这个端口下运行。针对 SAP 命名空间，我们用服务类型 SERVICE_TYPE_VALUE_SAPID 设置了服务的 SAP ID。

对于每一个 WSANSCLASSINFO 条目，在为它设置值的类型和大小时，还必须设置服务类型所应用的命名空间的标识符。表 8.3 中包含了服务类型所要求的类型，在这一示例中所要求的类型都是 DWORD。最后一步是简单调用 WSAInstallServiceClass，并把它当作一个参数传递给 WSASERVICECLASSINFO 结构。如果 WSAInstallServiceClass 调用成功，就返回 0；反之，则返回 SOCKET_ERROR。如果 WSASERVICECLASSINFO 无效或格式有误，WSAGetLastError 就会返回 WSAEINCAL。如果这个服务类已经存在，WSAGetLastError 则返回 WSAEALREADY。在这种情况下，就可以调用 WSARemoveServiceClass 函数，删掉一个服务类。该函数的声明如下：

```
INT WSARemoveServiceClass(LPGUID lpServiceClassId );
```

这个函数的惟一参数是指向 GUID 的指针。该 GUID 就是定义具体服务类的 GUID。

8.3.2　服务的注册

一旦安装了服务类(说明您的服务有哪些常见属性)，就可以注册自己的服务实例，这样，远程计算机上的其他客户机就以查询到这一服务了。注册服务实例的 Winsock 函数是 WSASetService，定义如下：

```
INT WSASetService (
    LPWSAQUERYSET lpqsRegInfo,
    WSAESETSERVICEOP essOperation,
    DWORD dwControlFlags
);
```

第 1 个参数 lpqsRegInfo，是指向 WSAQUERYSET 结构的指针，该指针定义特定的服务。现在简要说明这个结构。essOperation 参数指明即将发生的行为，例如注册或取消注册。表 8.4 对这 3 个有效标志进行了说明。

表 8.4　设置服务标志

操作标志	含　义
RNRSERVICE_REGISTER	注册一个服务名。对动态名称提供程序而言，这个标志的意思是开始动态地广告这个服务。对固定的名称提供程序来说该标志表示更新数据库。对静态名称提供程序而言该标志则无意义
RNRSERVICE_DEREGISTER	从注册表中删除整个服务。对动态名称提供程序而言，意味着中止广告服务。对固定的名称提供程序来说，意味着从数据库中删除这个服务。对静态名称提供程序来说无意义
RNRSERVICE_DELETE	只将指定的服务实例从注册表中删除。一个注册服务可能包含若干个实例(这是在注册之后，紧接着利用 SERVICE_MULTIPLE 标志来完成的)，而这个命令只删除指定的实例(由 CSADDR_INFO 结构定义)。再次提醒大家注意，这个标志只能应用于动态和固定名称提供程序

第 3 个参数 dwControlFlags 不是 0 就是 SERVICE_MULTIPLE 这个标志。如果要在特定的服务实例下注册若干个地址，用这个标志即可。例如，有一个服务要在 5 台计算机上运行。传递给 WSAService 的 WSAQUERY 结构将引用 5 个 CSADDR_INFO 结构，一一对该服务的实例进行描述。这时便需要设置 SERVICE_MULTIPLE 标志。稍后，可取消单个服务实例注册，这是利用 RNRSERVICE_ DELETE 标志来完成的。表 8.5 给出了操作和控制标志的可能组合，并根据服务的存在与否，对命令结果进行了描述。

表 8.5 WSASetService 标志的组合

RNRSERVICE_REGISTER		
标志	含义	
	若服务已存在	若服务不存在
None	覆盖现有的服务实例	在给定的地址上增加一条新的服务条目
SERVICE_MULTIPLE	通过添加新地址的方式，更新服务实例	在给定的地址上增加一条新的服务条目

RNRSERVICE_DEREGISTER		
标志	含义	
	若服务已存在	若服务不存在
None	删除所有的服务实例，但不删除服务(一般说来，是保留 WSAQUERY,但 CSADDR_INFO 结构的数量是 0)	这是一个错误，将返回 WSASERVICE_NOT_FOUND
SERVICE_MULTIPLE	通过删除给定地址的方式，对这个服务进行更新。即便没有地址，这个服务仍然会存在	这是一个错误，并返回 WSASERVICE_NOT_FOUND

RNRSERVICE_DELETE		
标志	含义	
	若服务已存在	若服务不存在
None	把这个服务彻底从命名空间删除	这是一个错误，并返回 WSASERVICE_NOT_FOUND
SERVICE_MULTIPLE	通过删除具体地址的方式，对这个这服务进行更新。如果不保留地址，服务就会彻底从命名空间删除	这是一个错误，并返回 WSASERVICE_NOT_FOUND

至此，大家已知道了 WSASetService 的用法。接下来看看 WSAQUERYSET 结构。这个结构需要填充并传递给函数。这个结构的定义为：

```
typedef struct _WSAQuerySetW {
    DWORD                dwSize;
    LPTSTR               lpszServiceInstanceName;
    LPGUID               lpServiceClassId;
    LPWSAVERSION         lpVersion;
```

```
    LPTSTR              lpszComment;
    DWORD                   dwNameSpace;
    LPGUID              lpNSProviderId;
    LPTSTR              lpszContext;
    DWORD                   dwNumberOfProtocols;
    LPAFPROTOCOLS       lpafpProtocols;
    LPTSTR              lpszQueryString;
    DWORD                   dwNumberOfCsAddrs;
    LPCSADDR_INFO       lpcsaBuffer;
    DWORD                   dwOutputFlags;
    LPBLOB              lpBlob;
}WSAQUERYSETW, *PWSAQUERYSETW, *LPWSAQUERYSETW;
```

dwSize 字段应该设为 WSAQUERYSET 结构的长度。lpszServiceInstanceName 字段中包含一个为服务实例命名的字符串标识符。lpServiceClassId 字段是服务实例所属的那个服务类的 GUID。lpVersion 字段是可选的，当客户机请求一项服务时，可利用它获得有用的版本信息。lpszComment 字段也是可选的，可利用它指定一个任何类型的注释字符串。dwNameSpace 字段指定用来注册服务的命名空间。如果只用一个命名空间，就只能用指定的那个值；反之，就用 NS_ALL。另外，可以引用一个自定义的命名空间提供程序(关于怎样编写自定义的命名空间，将在第 12 章论述)。在使用自定义的命名空间提供程序时，要将 dwNameSpace 字段设为 0，而 lpNSProviderId 字段指定代表自定义的命名空间提供程序的 GUID。lpszContext 字段指定分级式命名空间(例如 NDS)中的查询起点。

dwNumberOfProtocols 和 lpafpProtocols 字段都是可选的参数，用于对搜索作限制，使其只返回所提供的协议。dwNumberOfProtocols 字段引用 AFPROTOCOLS 结构的数量，这些结构包含在 lpafpProtocols 数组中。这个结构的定义为：

```
typedef struct _AFPROTOCOLS {
    INT iAddressFamily;
    INT iProtocol;
}AFPROTOCOLS, *PAFPROTOCOLS, *LPAFPROTOCOLS;
```

第 1 个字段 iAddressFamily 是地址族常量，比如 AF_INET 和 AF_IPX。第 2 个字段 iProtocol 用于指定地址族的协议，例如 IPPROTO_TCP 或 NSPROTO_IPX。

WSAQUERYSET 结构中的下一个字段 lpszQueryString 是可选的，只提供给支持强化 SQL(Structured Query Language，结构化查询语言)的命名空间(例如 Whois++)使用。这个参数用来指定字符串。

注册一个服务时，接下来的这两个字段是最重要的。dwNumberOfCsAddrs 字段只提供了传递到 lpcsaBuffer 中的 CSADDR_INFO 结构的数量。CSADDR_INFO 结构定义地址族和服务所在的地址。如果出现多个结构，就可以用多个服务实例。该结构的定义如下：

```
typedef struct _CSADDR_INFO {
```

```
    SOCKET_ADDRESS     LocalAddr;
    SOCKET_ADDRESS  RemoteAddr;
    INT              SocketType;
    INT              iProtocol;
}CSADDR_INFO;
typedef struct _SOCKET_ADDRESS {
    LPSOCKADDR lpSockaddr;
    INT         iSockaddrLength;
}SOCKET_ADDRESS,*PSOCKET_ADDRESS,FAR *LPSOCKET_ADDRESS;
```

这个结构中还包括了 SOCKET_ADDRESS 的定义。在注册一个服务时，可指定本地和远程地址。本地地址字段(LocalAddr)用于指定这个服务的实例应该绑定的地址，而远程地址字段(RemoteAddr)则用于指定客户机在 connect 或 sendto 调用中，应该使用的那个地址。其他两个字段(iSocketType 和 iProtocol)则是为给定的地址指定套接字类型(比方说 SOCK_STREAM 和 SOCK_DGRAM)和协议家族(比方说 AF_INET 和 AF_IPX)。

WSAQUERYSET 结构的最后两个字段是 dwOutputFlags 和 lpBlob。一般说来，服务注册不需要这两个字段；不过在查询服务实例时，它们是非常有用的(将在下一节讨论)。只有命名空间提供程序才可以返回一个 BLOB 结构。也就是说，在注册一个服务时，不能添加自己的 BLOB 结构并令其在客户机查询中返回。

表 8.6 列出了 WSAQUERYSET 结构的字段，并根据执行查询还是注册，判断哪些字段是必要的，哪些又是可选的。

表 8.6 WSAQUERYSET 字段

字 段	查 询	注 册
dwSize	要求	要求
lpszServiceInstanceName	要求字符串或“*”	要求
lpServiceClassId	要求	要求
lpVersion	可选	可选
lpszComment	忽略	可选
dwNameSpace lpNSProvider ID	必须指定两者中的其中一个的字段	必须指定两者中的其中一个的字段
LpszContext	可选	可选
dwNumberOfProtocols	0 或 0 以上	0 或 0 以上

续表

字　段	查　询	注　册
LpafpProtocols	可选	可选
LpszQueryString	可选	忽略
dwNumberOfCsAddrs	忽略	要求
LpcsaBuffer	忽略	要求
DwOutputFlags	忽略	可选
LpBlob	忽略，可通过查询返回	忽略

8.3.3　服务注册示例

这一小节将介绍如何在 SAP 和 NTDS 这两个命名空间下注册自己的服务。Windows NT 域名空间相当有用，这就是我们把它包含到示例中的原因。不过，还必须了解它下面这些特性。首先，因为 Windows NT 域名空间是以活动目录 Active Directory 目录服务为基础的，所以它需要 Windows 2000 或 Windows XP。另外，对用于注册和(或)查找服务的 Windows 2000 或 Windows XP 工作站而言，还意味着在这个域内必须有一个账号，可以在这个域内访问 Active Directory。另一个需要注意的特性是 Windows NT 域名空间能够注册源于任何一个协议家族的套接字地址。这意味着 IP 和 IPX 服务均可以在同一个命名空间内注册。下列示例代码说明了注册一个服务实例所需要的基本步骤。为了简单起见，代码中没有执行错误检查。

```
SOCKET                  socks[2];
WSAQUERYSET             qs;
CSADDR_INFO         lpCSAddr[2];
SOCKADDR_IN         sa_in;
SOCKADDR_IPX            sa_ipx;
IPX_ADDRESS_DATA        ipx_data;
GUID                    guid = SVCID_NETWARE(200);
Int                     ret ,cb;

memset(&qs,0,sizeof(WSAQUERYSET));
qs.dwSize = sizeof(WSAQUERYSET);
qs.lpszServiceInstanceName = (LPSTR)"Widget Server";
qs.lpServiceClassId = &guid;
qs.dwNameSpace = NS_ALL;
qs.lpNSProviderId = NULL;
qs.lpcsaBuffer = lpCSAddr;
qs.lpBlob = NULL;
```

```
//
//设置服务的 IP 地址
//
memset(&sa_in,0,sizeof(sa_in));
sa_in.sin_family = AF_INET;
sa_in.sin_addr.s_addr = htonl(INADDR_ANY);
sa_in.sin_port= 5150;

socks[0]= socket(AF_INET,SOCK_DGRAM,IPPROTO_UDP);
ret = bind(socks[0],(SOCKADDR *)&sa_in,sizeof(sa_in));

cb = sizeof(sa_in);
getsockname(socks[0],(SOCKADDR *)&sa_in,&cb);

lpCSAddr[0].iSocketType = SOCK_DGRAM;
lpCSAddr[0].iProtocol = IPPROTO_UDP;
lpCSAddr[0].LocalAddr.lpSockaddr = (SOCKADDR *)&sa_in;
lpCSAddr[0].LocalAddr.iSockaddrLength = sizeof(sa_in);
lpCSAddr[0].RemoteAddr.lpSockaddr = (SOCKADDR *)&sa_in;
lpCSAddr[0].RemoteAddr.iSockaddrLength = sizeof(sa_in);
//
//设置服务的 IPX 地址
//
memset(sa_ipx.sa_netnum,0,sizeof(sa_ipx.sa_netnum));
memset(sa_ipx.sa_nodenum,0,sizeof(sa_ipx.sa_nodenum));
sa_ipx.sa_family = AF_IPX;
sa_ipx.sa_socket= 0;

socks[1]= socket(AF_IPX,SOCK_DGRAM,NSPROTO_IPX);

ret= bind(socks[1],(SOCKADDR *)&sa_ipx,sizeof(sa_ipx));

cb = sizeof(IPX_ADDRESS_DATA);
memset(&ipx_data,0,cb);
ipx_data.adapternum = 0;
ret = getsockopt(socks[1],NSPROTO_IPX,IPX_ADDRESS,
      (char *)&ipx_data,&cb);
cb = sizeof(SOCKADDR_IPX);
getsockname(socks[1],(SOCKADDR *)sa_ipx,&cb);

memcpy(sa_ipx.sa_netnum,ipx_data.netnum,sizeof(sa_ipx.sa_netnum));
memcpy(sa_ipx.sa_nodenum,ipx_data.nodenum,sizeof(sa_ipx.sa_nodenum));

lpCSAddr[1].iSocketType = SOCK_DGRAM;
lpCSAddr[1].iProtocol = NSPROTO_IPX;
```

```
lpCSAddr[1].LocalAddr.lpSockaddr = (struct sockaddr *)&sa_ipx;
lpCSAdar[1].LocalAddr.iSockaddrLength = sizeof(sa_ipx);
lpCSAddr[1].RemoteAddr.lpSockaddr = (struct sockaddr *)&sa_ipx;
lpCSAddr[1].RemoteAddr.iSockaddrLength = sizeof(sa_ipx);

qs.dwNumberOfCsAddrs = 2;

WSASetService(&qs,RNRSERVICE_REGISTER,0L);
```

示例代码说明了如何设置一个服务实例，这样，该服务的一个客户机便可找到自己需要与之通信的服务地址。第 1 个步骤是初始化 WSAQUERY 结构。另外，我们还需要为自己的服务实例命名。这里，我们简单地把它叫作 WidgetServer。另外一个关键性步骤是利用注册服务类时所用的一个 GUID。无论何时注册一个服务实例，这个服务都必须属于一个服务类。这里，我们用的是“Widget Service Class”(前一小节定义的)，它的 GUID 是 SVCID_NETWARE(200)。下一步便是设置我们感兴趣的命名空间。由于我们的服务运行于 IPX 和 UDP 上，因而指定的是 NS_ALL。因为我们要指定先前存在的命名空间，所以必须把 lpNSProviderId 设为 NULL。

下一步是设置 CSADDR_INFO 数组内的 SOCKADDR 结构。WSASetService 把这个数组当作 WSAQUERYSET 结构中的 lpcsaBuffer 字段传递。大家将注意到，示例的确建立了套接字，并在建立 SOCKADDR 结构之前，把它们绑定到了一个本地地址上。这是因为我们必需找到客户机需要连接的那个确切的本地地址。比方说，在为服务器建立 UDP 套接字时，我们绑定了 INADDR_ANY，直到调用 getsocketname 时，INADDR_ANY 才会返回实际的 IP 地址。利用 getsocketname 返回的信息，就可建立 SOCKADDR_IN 结构了。在 CSADDR_INFO 结构内，我们设置了套接字类型和协议。另两个字段是本地和远程地址信息。本地地址是服务器应该绑定的地址，而远程地址则是客户机应该用来连接到服务的地址。

为这个基于 UDP 的服务器设置了 SOCKADDR_INFO 结构之后，我们设置了基于 IPX 的服务。在第 4 章已经指出服务器应该和内部网络编号绑定在一起，这是通过把网络和节点编号设为 0 而完成的。再次提醒大家注意，这时并没有给出客户机所需的那个地址，因此，需要调用套接字选项 IPX_ADDRESS 来获得实际的地址。在填充 IPX 的 CSADDR_INFO 结构时，套接字类型用 SOCK_ DGRAM，协议用 NSPROTO_IPX 即可。因为客户机可以用两个地址(UDP 和 IPX)来建立连接，所以下一步是把 WSAQUERYSET 结构中的 dwNumberOfAddrs 字段设为 2。最后，调用 WSASetSetService 函数，它带有 WSAQUERYSET 结构和 RNRSERVICE_REGISTER 标志，但没有控制标志。不要指定 SERVICE_MULTIPLE 控制标志，这样，在选择取消注册服务时，这个服务的所有实例(IPX 和 UDP 地址)都会被取消注册。

另外，需要注意的是：上面的示例中没有考虑多重初始地址计算机。如果在一台有多重初始地址的计算机上建立一个绑定到 INADDR_ANY 的基于 UDP 的服务器，客户机就可以在任何一个可用接口上与这个服务器建立连接。对于 IP 而言，仅有 getsockname 是不够的；必须获得所有的本地 IP 接

口。获得这类信息的方法要根据使用的平台而定。对所有平台而言，常见的方法是调用 gethostbyname，返回具体名称的 IP 地址列表。在 Winsock 2 下，还可以调用 I/O 控制命令 SIO_GET_INTERFACE_LIST。针对 Windows 2000 及 Windows XP 平台，可使用 I/O 控制命令 SIO_ADDRESS_LIST_QUERY。最后，还可以用第 16 章中讨论的 IP 助手函数。简单的 TCP/IP 名称解析和对 gethostbyname 的介绍参见第 3 章，而 I/O 控制命令则参见第 7 章。

注意 除此以外，补充文件中的 RNRCS.CPP 是一个完整翔实的示例，它着重讨论了多重初始地址计算机。

8.4 服务的查询

现在，大家已经知道如何在命名空间内注册服务了，下面来看看客户机如何向命名空间查询具体的服务，进而获得通信服务的有关信息。尽管名称解析采用的查询函数有 3 个之多：WSALookupServiceBegin、WSALookupServiceNext 和 WSALookupServiceEnd，但和服务注册比起来，仍然要简单些。

执行查询的第 1 步是调用 WSALookupServiceBegin，这个函数通过设置查询限制来开始查询。它的原型如下：

```
INT  WSALookupServiceBegin (
    LPWSAQUERYSET lpqsRestrictions,
    DWORD dwControlFlags,
    LPHANDLE lphLookup
);
```

第 1 个参数是一个 WSAQUERY 结构，它对查询加以限制，例如限定查询哪些命名空间。第 2 个参数 dwControlFlags 决定查询的深度。表 8.7 中包含了各种可能用到的标志及其含义。这些标志都会影响查询的行为和查询返回的数据。最后一个参数是 HANDLE 类型的，它在函数返回时被初始化。若成功，函数返回的值就是 0；否则，就返回 SOCKET_ERROR。如果一个或多个参数无效，WSAGetLastError 就会返回 WSAEINVAL。如果在命名空间内找到该服务名，但没有和这些限制相匹配的数据，错误就是 WSANO_DATA。若指定服务不存在，则返回 WSASERVICE_NOT_FOUND 错误。

表 8.7 控制标志

标 志	含 义
LUP_DEEP	在分级式命名空间中，与第 1 级相对的请求深度
LUP_CONTAINERS	只获取容器对象。该标志只适合分级式命名空间
LUP_NOCONTAINERS	不返回任何容器。该标志只适合分级式命名空间

续表

标 志	含 义
LUP_FLUSHCACHE	忽略缓存的信息，直接查询命名空间。注意并非所有的名称提供程序都缓存查询
LUP_FLUSHPREVIOUS	让名称提供程序丢弃前一次返回的信息集。这个标志一般在 WSALookupServiceNext 返回 WSA_NOT_ENOUGH_MEMORY 之后才用。如果信息太多，或者所提供的缓冲区不够，信息集就会被丢弃，名称提供程序接着获取下一个信息集
LUP_NEAREST	按距离顺序获取结果。注意，因为注册服务时没有相关的具体规定，所以距离度量的计算由名称提供程序负责。不要求名称提供程序支持这一概念
LUP_RES_SERVICE	指定本地地址在 CSADDR_INFO 结构中返回
LUP_RETURN_ADDR	获取 lpcsaBuffer 形式的地址
LUP_RETURN_ALIASES	只获取别名信息。每个别名都会在对 WSALookupServiceNext 的连续调用中返回，并会设置 RESULT_IS_ALIAS 标志
LUP_RETURN_ALL	获取所有可用的信息
LUP_RETURN_BLOB	获取 lpBlob 形式的私有数据
LUP_RETURN_COMMENT	获取 lpszComment 形式的注释
LUP_RETURN_NAME	获取 lpszServiceInstanceName 形式的名称
LUP_RETURN_TYPE	获取 lpServiceClassId 形式的类型
LUP_RETURN_VERSION	获取 lpVersion 形式的版本号

在调用 WSALookupServiceBegin 时，返回的查询句柄是传递给 WSALookupServiceNext 的，它将返回我们需要的数据。该函数的定义如下：

```
INT WSALookupServiceNext(
    HANDLE hLookup,
    DWORD dwControlFlags,
    LPDWORD lpdwBufferLength,
    LPWSAQUERYSET lpqsResults
);
```

句柄 hLookup 通过 WSALookupServiceBegin 返回。对 dwControlFlags 参数而言，除了仅支持

LUP_FLUSHPREVIOUS 外，其含义和 WSALookupServiceBegin 是相同的。参数 lpdwBufferLength 是一个缓冲区长度，该缓冲区由 lpqsResults 传递。由于 WSAQUERYSET 结构中可能包含 BLOB(binary large object，二进制大型对象)数据，因此它通常要求传递一个大于结构本身的缓冲区。对即将返回的数据而言，如果提供的缓冲区长度不够，函数调用就会失败，并出现 WSA_NOT_ENOUGH_MEMORY 错误。

一旦利用 WSALookupServiceBegin 开始查询，就要在生成 WSA_E_NO_MORE(10110)错误之前，调用 WSALookupServiceNext。特别需要注意的是：早期的 Winsock 实现方案中，表示“没数据了”的错误代码是 WSAENOMORE(10102)，因此，对健壮的代码而言，应该还要对这两个错误都进行检查。一旦所有数据都已返回，或已经完成查询，再调用查询过程中所用的、带有 HANDLE 变量的 WSALookupServiceEnd。该函数的定义如下：

```
INT WSALookupServiceEnd (HANDLE hLookup );
```

8.4.1 怎样查询服务

如何对前一小节中注册的服务进行查询呢？首先是设置在定义查询时所用的 WSAQUERY 结构。来看看下面的代码：

```
WSAQUERYSET    qs;
GUID            guid = SVCID_NETWARE(200);
AFPROTOCOLS     afp[2]= {{AF_IPX,NSPROTO_IPX},{AF_INET,IPPROTO_UDP}};
HANDLE          hLookup;
int               ret ;

memset(&qs,0,sizeof(qs));
qs.dwSize = sizeof (WSAQUERYSET);
qs.lpszServiceInstanceName = "Widget Server";
qs.lpServiceClassId = &guid;
qs.dwNameSpace = NS_ALL;
qs.dwNumberOfProtocols = 2;
qs.lpafpProtocols = afp;

ret = WSALookupServiceBegin(&qs,LUP_RETURN_ADDR |LUP_RETURN_NAME,
      &hLookup);
if (ret == SOCKET_ERROR)
      //出错
```

记住，所有的服务查询都是在服务类 GUID 的基础上进行的，您查询的服务就是在这个服务类基础上注册的。变量 guid 设为服务器的服务类 ID。先把 qs 初始化为 0，再把 dwSize 字段设为结构的长度。下一步是给出准备查询的服务名。服务名可以是服务的真名，也可以指定一个通配符(*)，通配符将返回给定的服务类 GUID 的所有服务。接下来，让查询利用常量 NS_ALL 搜索所有的命名空间。最

后，设置协议(IPX 和 UDP/IP)，以便客户机能与之建立连接。这是利用山两个 AFPROTOCOLS 结构组成的数组来完成的。

现在，准备开始查询，首先必须调用 WSALookupServiceBegin。它的第 l 个参数是 WSAQUERYSET 结构，而后面的几个参数则是标志，这些标志定义找到相符的服务时应该返回哪些数据。这里，应该对 LUP_RETURN_ADDR 和 LUP_RETURN_NAME 这两个标志执行按位“和”运算，以此说明自己需要寻址信息和服务名；只有在用通配符(*)来指代服务名时，才需要 LUP_RETURN_NAME 标志；否则就表示已经知道服务名了。最后一个参数是一个标识这个特殊查询的 HANDLE 变量。函数成功返回之后，它将会被初始化。

一旦成功打开查询，就可在 WSA_E_NO_MORE 返回之前，调用 WSALookupServiceNext。每个成功的调用都会返回符合查询标准的那个服务的相关信息。这一步骤的代码如下：

```
char            buff[sizeof(WSAQUERYSET)+2000];
DWORD           dwLength,dwErr;
WSAQUERYSET   *pqs = NULL;
SOCKADDR       *addr;
int              i;

pqs = (WSAQUERYSET *) buff;
dwLength = sizeof(WSAQUERYSET)+2000;
while (1)
{
    ret = WSALookupServiceNext(hLookup,0,&dwLength,pqs);
    if (ret == SOCKET_ERROR)
    {
       if ((dwErr = WSAGetLastError()) == WSAEFAULT)
       {
          printf("Buffer too small;required size is:%d \n",dwLength);
          break;
       }
       else if ((dwErr == WSA_E_NO_MORE)||(dwErr = WSAENOMORE))
          break;
       else
       {
          printf("Failed with error:%d \n",dwErr);
          break;
       }
    }
for (i = 0;i <pqs->dwNumberOfCsAddrs;i++)
{
   addr = (SOCKADDR *)pqs->lpcsaBuffer[i]RemoteAddr.lpSockaddr;
   if (addr->sa_family == AF_INET)
   {
```

```
            SOCKADDR_IN *ipaddr = (SOCKADDR_IN *)addr;
            printf("IP address:port = %s:%d \n",
            inet_ntoa(ipaddr->sin_addr),
            ipaddr->sin_port);
        }
        else if (addr->sa_family == AF_IPX)
        {
            CHAR IPXString[64];
            DWORD IPXStringSize = sizeof(IPXString);
            if (WSAAddressToString((LPSOCKADDR)addr,
                sizeof(SOCKADDR_IPX),NULL,IPXString,
                &IPXStringSize) == SOCKET_ERROR)
                printf("WSAAddressToString returned error %d \n",
                    WSAGetLastError());
                else
                    printf("IPX address:%s \n",IPXString);
            }
        }
    }
    WSALookupServiceEnd(hLookup);
```

尽管这段示例代码有点简单，但非常直截了当。因为惟一可用于 WSALookupServiceNext 函数的标志只有一个：LUP_FLUSHPREVIOUS，所以该调用只需要一个有效的查询句柄、返回缓冲区和返回缓冲区的长度，不需要指定任何控制标志。如果提供的缓冲区太小，并且设置这个标志，该函数返回的结果就会被丢弃。然而，在示例中，我们没有采用 LUP_FLUSHPREVIOUS，如果我们提供的缓冲区太小，就会生成 WSAEFAULT 错误。发生这种情况时，就要把 lpdwBufferLength 设为所需要的长度。本例采用了一个固定长度的缓冲区，相当于 WSAQUERYSET 结构再加 2 000 个字节的长度。由于只要求了解所有的服务名和地址，所以这个长度绰绰有余。当然，如果要面向广大用户，应用程序中还应该能对 WSAEFAULT 错误进行处理。

一旦成功调用了 WSALookupServiceNext，缓冲区内就会填入一个 WSAQUERYSET 结构，这个结构中包含了返回的结果。在查询中，我们需要服务名和地址；那么 WSAQUERYSET 结构中，最重要的字段则是 lpszServiceInstanceName 和 lpcsaBuffer。前者包含服务名，后者则是一个 CSADDR_INFO 结构组成的数组，其中包含服务的寻址信息。dwNumberOfCsAddrs 参数准确地告诉我们已返回多少地址。上面的示例代码中，我们只是输出了地址。因为在打开查询时，只要求了这两个地址族，所以只检查并输出了 IPX 和 IP 地址。

在查询中，如果用通配符(*)来代表服务名，那么每次调用 WSALookupServiceNext，都会返回一个运行于网络上某个地方的特定的服务实例，当然，前提是实际上已注册了多个服务实例，并且都处于运行状态。一旦服务的所有实例都已返回，就会产生 WSA_E_NO_MORE 错误，这样就会中断循环。最后一件事是在查询句柄上调用 WSALookupServiceEnd，这样将释放为查询分配的所有资源。

8.4.2　查询 DNS

前面曾提过，DNS 命名空间是静态的，这意味着不能动态地注册服务；但仍可用 Winsock 名称解析函数来执行 DNS 查询。实际上，因为 DNS 命名空间提供程序是以 BLOB 的形式返回查询信息的，所以执行 DNS 查询比执行普通的已注册服务查询要复杂得多。为什么会这样呢？还记得第 3 章中对 gethostname 的讨论吗？那里已说过，名称检索返回的 HOSTNAME 结构中不仅包含了 IP 地址，还包含了别名。这些信息不太符合 WSAQUERYSET 结构的字段。

关于 BLOB 数据，需要注意的一点是：它的数据格式尚未很好地编入文档，这样就使得直接查询 DNS 显得有些困难。首先来看看如何打开查询。补充文件中的 DNSQUERY.CPP 文件包含了直接查询 DNS 所用的完整的示例代码；我们将逐段地对它们进行分析。下面的代码说明了如何初始化 DNS 查询：

```
WSAQUERYSET   qs;
AFPROTOCOLS   afp[2] = {{AF_INET,IPPROTO_UDP},{AF_INET,IPPROTO_TCP}};
GUID          hostnameguid = SVCID_INET_HOSTADDRBYNAME;
DWORD         dwLength = sizeof(WSAQUERYSET)+sizeof(HOSTENT)+2048;
char          buff[dwLength];
HANDLE        hQuery;
int           ret ;

qs = (WSAQUERYSET *)buff;
memset(&qs,0,sizeof(qs));
qs.dwSize = sizeof(WSAQUERYSET);
qs.lpszServiceInstanceName = argv[1];
qs.lpServiceClassId = &hostnameguid;
qs.dwNameSpace = NS_DNS;
qs.dwNumberOfProtocols = 2;
qs.lpafpProtocols = afp;

ret = WSALookupServiceBegin(&qs,LUP_RETURN_NAME ,LUP_RETURN_BLOB,
      &hQuery);
if (ret == SOCKET_ERROR)
      //出错
```

DNS 查询的设置与前一个例子非常类似。最显著的变化是，这里采用的是 GUID，即 SVCD_INET_HOSTADDRBYNAME。这是一个标识主机名查询的预定义的 GUID。lpszServiceInstanceName 是需要解析的主机名。由于要通过 DNS 解析主机名，所以只需为 dwNameSpace 指定 NS_DNS。最后，把 lpafProtocols 设成一个数组，该数组由两个 AFPROTOCOLS 结构组成，它把 TCP/IP 和 UDP/IP 协议定义成作为查询目标的协议。

查询一旦建立，就可调用 WSALookupServiceNext 来返回数据：

```
char          buff[sizeof(WSAQUERYSET)+sizeof(HOSTENT)+2048];
DWORD         dwLength = sizeof(WSAQUERYSET)+sizeof(HOSTENT)+2048;
WSAQUERYSET   *pqs;
HOSTENT           *hostent;

pqs = (WSAQUERYSET *)buff;
pqs->dwSize = sizeof(WSAQUERYSET);
ret = WSALookupServiceNext(hQuery,0,&dwLength,pqs);
if (ret == SOCKET_ERROR)
    //出错
WSALookupServiceEnd(hQuery);

hostent = (HOSTENT *)pqs->lpBlob->pBlobData;
```

因为 DNS 命名空间提供程序以 BLOB 的形式返回主机信息，所以需要提供一个足够大的缓冲区。这就是为什么这里用了一个非常大的缓冲区，其长度相当于一个 WSAQUERYSET 结构加上一个 HOSTENT 结构，再加上 2 048 个字节。再次提醒大家注意，如果长度还不够，函数调用就会失败，出现 WSAEFAULT 错误。在 DNS 查询中，所有的主机信息都放在 HOSTENT 结构内返回，因此有时主机名与多个 IP 地址关联在一起。这就是不必多次调用 WSALookupServiceNext 的原因。

有一点需要特别注意——对查询返回的 BLOB 结构进行解码。通过第 3 章的介绍，大家已知道 HOSTENT 结构的定义是这样的：

```
typedef struct hostent {
    char FAR * h_name;
    char FAR * FAR * h_aliases;
    short h_addrtype;
    short h_length;
    char FAR *FAR *h_addr_list;
}HOSTENT;
```

当 HOSTENT 结构以 BLOB 数据的形式返回时，结构内的指针对应的实际上是内存中数据存放时的偏移位置。这个偏移位置是自 BLOB 数据的起始处开始算起的。这样，就必须对指针作一番修正，使其指向绝对的内存位置，否则不能实际地访问到数据。图 8.1 展示了返回的 HOSTENT 结构以及内存的布局。DNS 查询是在主机名 riven 上执行的，该主机有一个对应的 IP 地址，但没有别名。结构中的每个字段都有一个偏移值。为纠正这个问题，使字段能引用正确位置，需要将偏移值加到 HOSTENT 结构的头地址上。这一操作需要针对 h_name、h_aliases 和 h_addr_list 字段进行。此外，h_aliases 和 h_addr_list 字段指定的是一个指针数组。一旦取得了指向指针数组的正确指针，在引用位置中的每个 32 位字段都会由偏移值构成。看看图 8.1 展示的 h_addr_list 字段，可以看出初始偏移值是 16 字节，它引用的是 HOSTENT 结构末尾之后的字节数。这是指向 4 字节 IP 地址的一个指针数组。不过，数组中的第一个指针的偏移值是 28 字节。要想指向正确的位置，需要先取得 HOSTENT 结构的地址，在它的基础上加 28 字节，指向一个实际的 4 字节位置。在那个位置上，含有数据 0x9D36B9BA，亦

即 IP 地址 157.54.185.186。随后，使可自那个位置开始，在 28 个字节的偏移值之后，取得总长为 4 个字节的数据(全部为 0)，它标志着指针数组的结束。假如该主机名同时关联着几个 IP 地址，那么必然存在着其他的偏移值。如法炮制，修正每一个指针，得到正确的数据。针对 h_aliases 指针以及它所引用的那个指针数组，也要采取同样的做法，将它们修正为正确的值。就本例来说，主机并未设置别名。数组的第 1 个条目是 0，表明不必再对那个字段采取任何多余的操作。最后一个字段是 h_name 字段，非常容易修正——只需将偏移值加到 HOSTENT 结构地址上面就可以了，它指向的是一个以空字符结束的字符串的起始处。

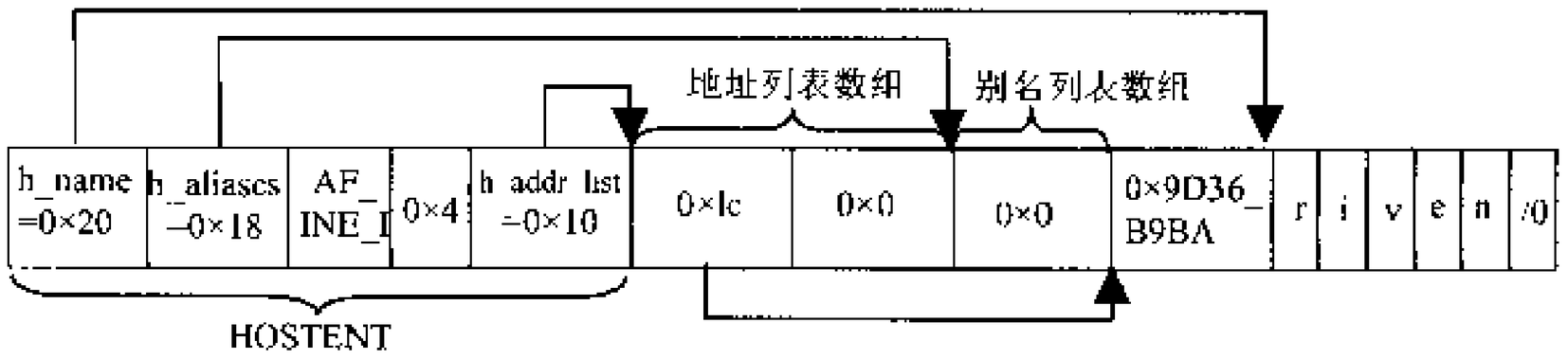


图 8.1　HOSTENT BLOB 数据

要把这些偏移修正为真正的地址，尽管涉及到大量的指针运算，但所需代码并不复杂。要修正 h_name 字段，进行简单的偏移调整即可，具体方法如下：

```
hostent->h_name = (PCHAR)((DWORD_PTR)hostent->h_name +
(PCHAR)hostent);
```

要修正指针数组(例如 h_aliases 和 h_addr_list 字段中的数组)，需要的代码则要多一些，但也只需要在这个数组中完整地循环一遍，依次修正每一个引用，直到碰到一个空条目为止。代码如下：

```
PCHAR *addr;
if (hostent->h_aliases)
{
    addr = hostent->h_aliases =
    (PCHAR)((DWORD_PTR)hostent->h_aliases +(PCHAR)hostent);
    while (addr)
    {
       addr = (PCHAR)((DWORD_PTR)addr +(PCHAR *)hostent);
       addr++;
    }
}
```

这段代码只是逐步浏览了各个数组条目，把 HOSTENT 结构的起始地址加到具体的偏移值上，这个偏移值即后来的条目的值。当然，一旦碰到了一个其值为 0 的数组条目，该过程就会停下来。对 h_addr_list 字段也要如法炮制。一旦偏移得到了修正，就可以正常地使用 HOSTENT 结构了。

8.4.3 查询 NLA

Windows XP 提供了一个名为 NLA 的新的名称服务，在任何给定时刻或当可用网络接口已经发生变化的时候，它能帮助应用程序识别所有连接到一个计算机的网络名称及连接属性。例如，您可能正在开发一个必须连接到 Internet 的应用程序，而 NLA 则可以确定一个 Internet 连接路径是否可用或何时能用。因为在移动计算领域，网络接口从哪里来和到哪里去由计算机的物理定位决定，例如在家里使用或工作时使用的一台膝上型电脑，所以 NLA 在这一领域特别有用。Winsock 允许通过命名空间提供程序接口来查询 NLA 服务。获取网络名称信息需要调用 WSALookupServiceBegin、WSALookupServiceNext 和 WSALookupServiceEnd。

在为 NLA 查询作准备时，应使用下述限制性的 WSAQUERYSET 参数及指导原则调用 WSALookupServiceBegin。dwSize 字段必须设为一个 WSAQUERYSET 结构的大小，dwNameSpace 字段必须设为 NS_NLA，lpServiceClassId 字段必须设为 GUID 定义 NLA_SERVICE_CLASS_GUID。其余字段——除 lpszServiceInstanceName 外——都被忽略。可以按需要选择设置 lpszServiceInstanceName 字段，将查询限制为一个特定的网络名。例如，假设有两个可用网络：一个家庭网络和一个 ISP 提供的 Internet 连接网络。假设 ISP 将 Internet 连接命名为“MyDSLProvider”，则可以通过指定类似“MyDSLProvider”的网络名称，将 NLA 查询限制在一个单一网络中。下面的代码片段展示了如何根据前面的讨论，建立一个限制性的 WSAQUERYSET 结构来为查询作准备：

```
WSAQUERYSET qs;
GUID NLANameSpaceGUID = NLA_SERVICE_CLASS_GUID;
//启动查询的必备字段
qs.dwSize = sizeof(WSAQUERYSET);
qs.dwNameSpace = NS_NLA;
qs.lpServiceClassId = &NLANameSpaceGUID;
//限制查询的可选字段
qs.lpszServiceInstanceName = "MyDSLProvider";
```

一只建立了限制性的结构，就必须确定如何使用为 WSALookupServiceBegin 而设的控制标志来控制 NLA 服务的查询。表 8.8 描述了 NLA 查询中专门使用的控制标志。如果单独使用 LUP_RETURN_NAME 或 LUP_RETURN_COMMENT，就只会接收到网络名称惟一的列表。如果还用了 LUP_RETURN_BLOB，就会接收到识别出来的每个网络的网络名称和网络特性。LUP_DEEP 标志允许接收最大数量的可用信息，不过，完成查询需要较长的时间。启动查询最普通的方式，是简单地对 LUP_RETURN_ALL 标志和 LUP_DEEP 标志取“或”即可，如下所示：

```
LPHANDLE hNLA;
WSALookupServiceBegin(&qs,LUP_RETURN_ALL | LUP_DEEP,&hNLA)
```

一旦成功地从 WSALookupServiceBegin 获得一个查找句柄 lphLookup，就可以开始对可用的网络名称进行 NLA 查询了。

表 8.8　NLA 专用控制标志

标　志	含　义
LUP_RETURN_ALL	相当于使用 LUP_RETURN_NAME、LUP_RETURN_COMMENT 和 LUP_RETURN_BLOB
LUP_RETURN_NAME	获取 WSAQUERYSET 的 lpszServiceInstanceName 字段中的网络名称
LUP_RETURN_COMMENT	获取 WSAQUERYSET 的 lpszComment 字段中的一个友好网络名称
LUP_RETURN_BLOB	获取网络特征信息，如连接速度、适配器标识及 Internet 连通性
LUP_DEEP	通知 NLA 去查询每个可用网络的所有特性
LUP_FLUSHPREVIOUS	仅用来调用 WSALookupServiceNext，以便跳过那些不能处理或不能为之分配内存的网络信息

因为 NLA 命名空间提供程序是作为一项服务实现的，因此，它的设计目的是在可用网络名称发生变化的时候，通知应用程序。第 1 次调用 WSALookupServiceBegin 时，可以立刻使用 WSALookupServiceNext 来查询当前网络信息，每次调用都将返回一个网络名称，直到函数失败，并返回错误 WSA_E_NO_MORE。

根据 WSALookupServiceBegin 中设置的控制标志，如果传递 LUP_RETURN_BLOB，就需要为传递到 WSALookupServiceNext 的 WSAQUERYSET 提供一个足够大的缓冲区，否则函数将失败，错误信息为 WSAEFAULT。LUP_RETURN_BLOB 可以允许查询返回一个 NLA_BLOB 数据结构的数组，该结构中包含特定的网络信息，例如从 WSALookupServiceNext 返回的每个网络的速度及连通性设置。

NLA_BLOB 可以由 0 或更多数据结构的数组组成，它的定义为：

```
typedef struct _NLA_BLOB {
   struct {
      NLA_BLOB_DATA_TYPE type;
      DWORD dwSize;
      DWORD nextOffset;
      }header;
   union {
      //header.type ->NLA_RAW_DATA
      CHAR rawData[1];
      //header.type ->NLA_INTERFACE
      struct {
         DWORD dwType;
         DWORD dwSpeed;
         CHAR adapterName[1];
      }interfaceData;
```

```
//header.type ->NLA_802_1X_LOCATION
struct {
   CHAR information[1];
}locationData;

//header.type ->NLA_CONNECTIVITY
struct {
   NLA_CONNECTIVITY_TYPE type;
   NLA_INTERNET internet;
}connectivity;
//header.type ->NLA_ICS
struct {
      struct {
         DWORD speed;
         DWORD type;
         DWORD state;
         WCHAR machineName[256];
         WCHAR sharedAdapterName[256];
      }remote;
   }ICS;

}data;

}NLA_BLOB,*PNLA_BLOB,*FAR LPNLA_BLOB;
```

在成功调用 WSALookupServiceNext 之后，首先应该检查 WSAQUERYSET 的 lpBlob 字段是否不为 NULL。如果不是 NULL，就可以开始查看从调用中返回的 NLA_BLOB 结构了。根据上述代码可以看出，一个 NLA_BLOB 结构有两个部分：头和数据。头部分描述 BLOB 的数据部分是哪种 NLA 类型，以及网络名称上是否还有更多的 NLA_BLOB(通过 nextOffset 字段)需要处理。应用程序应该在每个 NLA_BLOB 中的 nextOffset 字段所标识的所有数据 BLOB 中循环，直到该字段等于 0 为止。每个找到的 NLA_BLOB 可具有下述数据类型中的一种类型，这些数据类型定义在表 8.9 中。

表 8.9 NLA 数据类型

NLA_BLOB 数据类型	相应的 NLA_BLOB 数据段
NLA_802_1X LOCATION	存储在 locationData 结构中的无线网络信息
NLA_CONNECTIVITY	存储在 connectivity 结构中的基本网络连通性信息——有两个字段：type 及 Internet，分别指 NLA_CONNECTIVITY_TYPE 和 NLA_INTERNET 枚举类型——参见表 8.10 和表 8.11 对这些类型的描述
NLA_ICS	ICS 结构中共享 Internet 连接的内部网络

续表

NLA_BLOB 数据类型	相应的 NLA_BLOB 数据段
NLA_INTERFACE	常规网络信息，如媒体类型、连接速度及适配器名称，这些信息存储在 interfaceData 结构中。可能的媒体类型在 Ipifcons.h 的媒体类型部分定义
NLA_RAW_DATA	指 rawData 字段，但没有使用

表 8.10　NLA 连通性类型

NLA_CONNECTIVITY_TYPE	含　义
NLA_NETWORK_AD_HOC	代表一个未与任何其它网络连接的专用网络
NLA_NETWORK_MANAGED	代表一个用域控制器管理的网络
NLA_NETWORK_UNKNOWN	代表一个网络，这个网络属于一个 ICS 连接的专用(非 Internet)面
NLA_NETWORK_UNMANAGED	服务不能确定连接特征

表 8.11　NLA Internet 类型

NLA_INTERNET 数据类型	含　义
NLA_INTERNET_NO	表明没有到 Internet 的可用路径
NLA_INTERNET_UNKNOWN	表明服务不能确定是否存在一条到 Internet 的路径
NLA_INTERNET_YES	表明有一条到 Internet 的可用路径

能看出来 NLA_INTERFACE 和 NLA_CONNECTIVITY 数据类型提供了可用网络的一些最有用的信息。例如，如果获得了一个 NLA_CONNECTIVITY 数据 BLOB，就可以确定是否有一个到 Internet 的连接。下述代码段展示了如何通过调用 WSALookupServiceNext 获得每个网络名称，以及如何在每个名称中找到的 NLA_BLOB 数据中循环。

```
char   buff[16384];
DWORD  BufferSize;
while (1)
{
    memset(qs,0,sizeof(*qs));

    BufferSize = sizeof(buff);
    if (WSALookupServiceNext(hNLA,LUP_RETURN_ALL,
                        &BufferSize,qs) == SOCKET_ERROR)
```

```
    {
        int Err = WSAGetLastError();
        if (Err == WSA_E_NO_MORE)
        {
            //已无数据。停止询问。
            //
            break;
        }

        printf("WSALookupServiceNext failed with error %d \n",
        WSAGetLastError());

        WSALookupServiceEnd(hNLA);
        return;
    }

    printf("\nNetwork Name:%s \n",qs->lpszServiceInstanceName);
    printf("Network Friendly Name:%s \n",qs->lpszComment);
    if (qs->lpBlob != NULL)
    {
        //
        //在 BLOB 数据列表中循环
        //
        DWORD      Offset = 0;
        PNLA_BLOB  pNLA;

        do
        {
            pNLA = (PNLA_BLOB)&(qs->lpBlob->pBlobData[Offset]);
            switch (pNLA->header.type)
            {
            case NLA_RAW_DATA:
                printf("\tNLA Data Type:NLA_RAW_DATA \n");
                break;
            case NLA_INTERFACE:
                printf("\tNLA Data Type:NLA_INTERFACE \n");
                printf("\t \tType:%d \n",
                    pNLA->data.interfaceData.dwType);
                printf("\t \tSpeed:%d \n",
                    pNLA->data.interfaceData.dwSpeed);
                printf("\t \tAdapter Name:%s \n",
                    pNLA->data.interfaceData.adapterName);
                break;
            case NLA_802_1X_LOCATION:
                printf("\tNLA Data Type:NLA_802_1X_LOCATION \n");
```

```
        printf("\t \tInformation:%s \n",
            pNLA->data.locationData.information);
        break;
    case NLA_CONNECTIVITY:
        printf("\tNLA Data Type:NLA_CONNECTIVITY \n");
        switch(pNLA->data.connectivity.type)
        {
            case NLA_NETWORK_AD_HOC:
                printf("\t \tNetwork Type:AD HOC \n");
                break;
            case NLA_NETWORK_MANAGED:
                printf("\t \tNetwork Type:Managed \n");
                break;
            case NLA_NETWORK_UNMANAGED:
                printf("\t \tNetwork Type:Unmanaged \n");
                break;
            case NLA_NETWORK_UNKNOWN:
                printf("\t \tNetwork Type:Unknown \n");
        }
        switch(pNLA->data.connectivity.internet)
        {
            case NLA_INTERNET_NO:
                printf("\t \tInternet connectivity:No \n");
                break;
            case NLA_INTERNET_YES:
                printf("\t \tInternet connectivity:Yes \n");
                break;
            case NLA_INTERNET_UNKNOWN:
                printf("\t \tInternet connectivity:
                Unknown \n");
                break;
        }
        break;
        case NLA_ICS:
            printf("\tNLA Data Type:NLA_ICS \n");
            printf("\t \tSpeed:%d \n",
                pNLA->data.ICS.remote.speed);
            printf("\t \tType:%d \n",
                pNLA->data.ICS.remote.type);
            printf("\t \tState:%d \n",
                pNLA->data.ICS.remote.state);
            printf("\t \tMachine Name:%S \n",
                pNLA->data.ICS.remote.machineName);
            printf("\t \tShared Adapter Name:%S \n",
                pNLA->data.ICS.remote.sharedAdapterName);
```

```
                        break,
                    default:
                        printf("\tNLA Data Type:Unknown to this rogram \n");
                        break;
                }
                Offset = pNLA->header.nextOffset;
            }
            while (Offset != 0);
        }
    }
```

在取得所有的名称及从 WSALookupServiceNext 返回的 NLA_BLOB 数据之后，NLA 服务就有能力用 WSANSPIoctrl API 向应用程序通知在这些接口上所发生的改变了。WSANSPIoctrl 接受 WSALookupServiceBegin 返回的查找句柄，并将它和应用程序的一个完成方法关联起来，以便接收变化通知。WSANSPIoctl 的定义为：

```
int WSAAPI WSANSPIoctl(
    HANDLE hLookup,
    DWORD dwControlCode,
    LPVOID lpvInBuffer,
    DWORD cbInBuffer,
    LPVOID lpvOutBuffer,
    DWORD cbOutBuffer,
    LPDWORD lpcbBytesReturned,
    LPWSACOMPLETION lpCompletion
);
```

hLookup 参数应该被设为最初从 WSALookupServiceBegin 返回的句柄。dwControlCode 参数应该被设为 SIO_NSP_NOTIFY_CHANGE。NLA 不使用 lpvInBuffer 和 lpvOutBuffer 参数，因此两者都应该是 NULL。cbInBuffer 和 cbOutBuffer 参数也没有被使用，应该设为 0。lpcbBytesReturned 参数没什么用，不过必须为它提供一个缓冲区。lpCompletion 字段允许指定一个异步等待方法，如第 5 章中的重叠 I/O 模型所述。下面的代码段展示了如何建立一个基于事件的通知，来等待网络变化。

```
WSAOverlap.hEvent = Event;
WSAComplete.Type = NSP_NOTIFY_EVENT;
WSAComplete.Parameters.Event.lpOverlapped = &WSAOverlap;

if (WSANSPIoctl(hNLA,SIO_NSP_NOTIFY_CHANGE,NULL,0,NULL,0,
        &BytesReturned,&WSAComplete) == SOCKET_ERROR)
{
    intRet = WSAGetLastError();
    if (Ret! = WSA_IO_PENDING)
    {
        printf("WSANSPIoctrl failed with error %d \n",Ret);
        return;
```

```
    }
}

if (WSAWaitForMultipleEvents(1,&Event,TRUE,WSA_INFINITE,FALSE) ==
        WSA_WAIT_FAILED)
{
    printf("WSAWaitForMultipleEvents failed with error %d \n",
        WSAGetLastError());
    return;
}
WSAResetEvent(Event);

//使用同一个查找句柄查询当前所有可用的网络名称,这个句柄最初是从 WSALookupServiceBegin 返回
的
```

注意　在补充材料中，有一个名为 NLAQUERY.CPP 的示例，这个示例展示了如何根据前面已讨论的内容查询 NLA 名称。

最后，NLA 允许将附加说明信息和一个使用 WSASetService API 的查询所返回的每个网络名称关联起来。需要做的事情是，将一个有效的名称填入 WSAQUERYSET 结构的 lpszServiceInstanceName 字段，并为该结构的 lpszComment 字段中提供一个新的友好名称。设置好这两个字段之后，再调用 WSASetService，此时将第 1 个参数(lpqsRegInfo)设为 WSAQUERYSET 结构的一个指针，第 2 个参数(essOperation) 设为 RNRSERVICE_REGISTER 标志，第 3 个参数(dwControlFlags) 设为 NLA_FRIENDLY_NAME 值。

8.5 小　结

注册及名称解析(RNR)函数看起来过于复杂，但是它们为编写客户机/服务器应用程序提供了非常大的灵活性。名称注册的真正局限在于命名空间。尽管 TCP/IP 日渐风行，DNS 还是独领风骚了很长时间，虽然 DNS 欠缺灵活性。直到有了 Windows 2000、Windows XP 和 Windows NT 域名空间，我们才有了一个稳定的、与协议无关的名称解析方法，它为编写健壮的应用程序提供了充分的灵活性。另外，其他命名空间(例如 SAP)也可用在基于 IPX/SPX 的应用程序上，它们提供了许多类似于 Windows NT 域名空间的功能(与协议无关这一点除外)。

9 多　播

多播技术(有时也被称为点到多点)是一种让数据从一个组成员送出，然后复制给其他多个成员的技术。采用这种技术，可有效减轻网络通信的负担，避免资源的无谓浪费。最开始的时候，设计这一技术的目的是弥补广播通信的不足。假如过度地使用广播技术，将造成网络带宽的大幅占用，从而影响整个网络的通信效率。多播通信则不同。对一个网络内的工作站来说，只有在上面运行的进程表示自己对多播数据感兴趣时，多播数据才会复制给它们。然而，并非所有协议都支持多播通信，对Windows 平台而言，仅两种协议(IP 和 ATM)提供了对多播通信的支持。IP 多播包括 IPv4 及 IPv6。另外，存在着不同的 IP 多播类型。例如，Windows XP 支持 IPv4 的源多播，也支持可靠多播提供程序，这种提供程序使用 IP 多播，但增加了可靠性及面向会话的内容。本章将介绍多播通信的一些基本知识，同时说明如何在这两种协议中实现多播通信。

首先，我们将探讨多播网络的基本原理，其中涉及多播通信的各种可能形式，以及 IP 及 ATM 多播通信的基本特征。然后本章再分别用两个小节，讲解 IP 和 ATM 特有的一些问题。在每一小节，我们将讨论如何用 Winsock API 实现每种协议的多播特性。

所有 Windows 平台均支持一个或多个多播协议。所有 Windows 平台均支持 IPv4 多播(Windows CE 要求 2.1 版或更高版本)。IPv6 多播则在 Windows XP 上得到支持，在 Windows 2000 上为 IPv6 技术预览，在 Windows NT4.0 上为微软研究的 IPv6 堆栈。如果安装了 MSMQ(Microsoft Message Queue，微软信息队列)组件，则 Windows XP 可支持可靠多播。最后，Windows 98、Windows Me、Windows 2000 及 Windows XP 均支持本地 ATM 多播。

9.1　多播的含义

多播通信具有两个层面的重要属性：控制层面和数据层面。其中，**控制层面** (Control Plane)定义了组成员的组织方式；而**数据层面**(Data Plane)决定了在不同的成员之间，数据如何传播。这两方面的属性既可以是**有根的**(Rooted)，也可以是**无根的**(Non-rooted)。在一个“有根的”控制层面内，存在着

一个特殊的多播组成员，叫做 c_root(控制根，或根节点)。而剩下的每个组成员都叫做 c_leaf(控制叶，或叶节点)。大多数情况下，c_root 需负责多播组的建立，其间涉及到建立同任意数量的 c_leaf 连接。而在某些特殊情况下，c_leaf 则可在以后的某个时间申请加入一个特定的多播组(或者说，取得那个组的成员资格)。要注意的是，对任何一个给定的组来说，都只能存在一个根节点。ATM 协议便是“有根控制层面”的典型例子。

而一个“无根的”控制层面则允许随意加入一个组，其间不存在任何例外。在这种情况下，所有组成员均为 c_leaf 节点(叶节点)。每个成员都有权加入一个多播组。在一个无根控制层面内，我们可强行实施自己的组成员资格方案(实际效果类似于建立一个 c_root 根节点)，这需要实现自己的组成员资格协议来完成。然而，组成员资格方案实际仍然是在一个无根控制层面的基础上建立起来的。IP 多播和可靠多播便是无根控制层面的例子。图 9.1 展示了有根控制层面与无根控制层面的区别。在位于左边部分的有根控制层面中，c_root 必须明确邀请每个 c_leaf 都加入该组；而在位于右边部分的无根方案中，任何 c_leof 都能自由加入这个组。

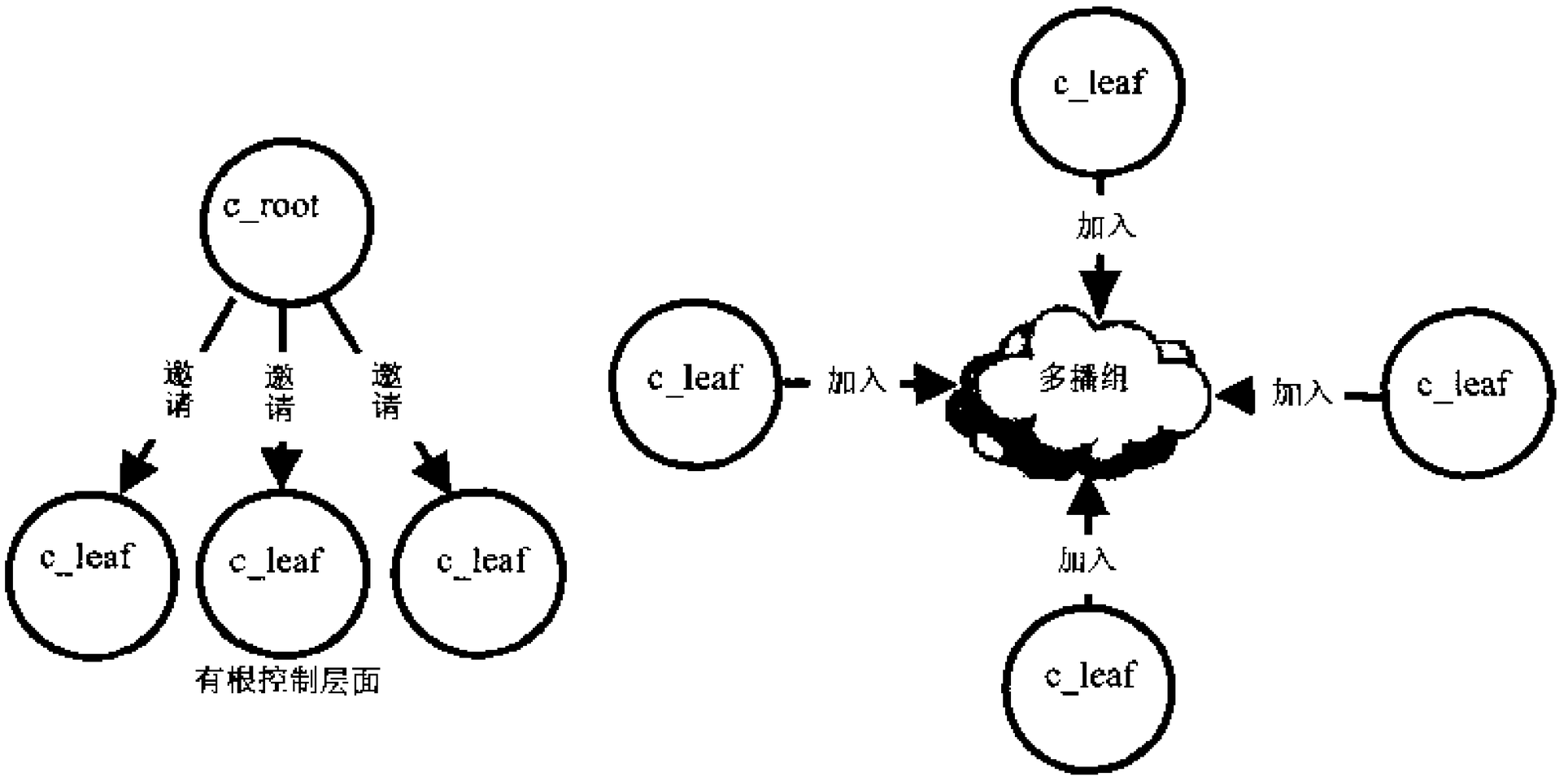


图 9.1　有根和无根控制层面

数据层面也存在着“有根的”和“无根的”两种形式。对一个有根数据层面而言，它有一个参与者叫做 d_root(数据根，或根节点)。数据传输只能在 d_root 和多播会话的其他成员之间进行。显然，那些成员是 d_leaf(数据叶，或叶节点)。这种传输既可单向进行，也可双向进行。但既然是一个有根数据层面，便暗示着出自一个 d_leaf 叶节点的数据只会被 d_root 根节点接收到；而自 d_root 发出的数据却可被每个 d_leaf 收到。ATM 和可靠多播便是“有根数据层面”的例子。在图 9.2 中，我们展示了有根及无根数据层面的区别。在位于左上部的有根数据层面中，自 d_root 发出的数据 abc 会传送给每一个 d_leaf；而自一个 d_leaf 发出的数据 xyz 却只会被 d_root 接收，不会传播到其他叶节点上。与此相反的是右下部分的无根示例。其中，不管最开始是由谁发出的数据，数据 abc 和 xyz 都会传播到

每个成员。

最后，在一个无根数据层面上，所有组成员都能将数据发给组内的其他所有成员。从一个组成员发出的数据块将被投递给其他所有成员，同时所有接收者都能将数据送回来。至于谁能接收或发送数据，则不存在任何限制。同样地，在数据层面上 IP 多播采用的是“无根的”通信方式。

由此我们看到 ATM 多播的控制层面和数据层面是有根的，而 IP 多播在两个层面都是无根的。可靠多播在控制层面是无根的，而在数据层面则是有根的。

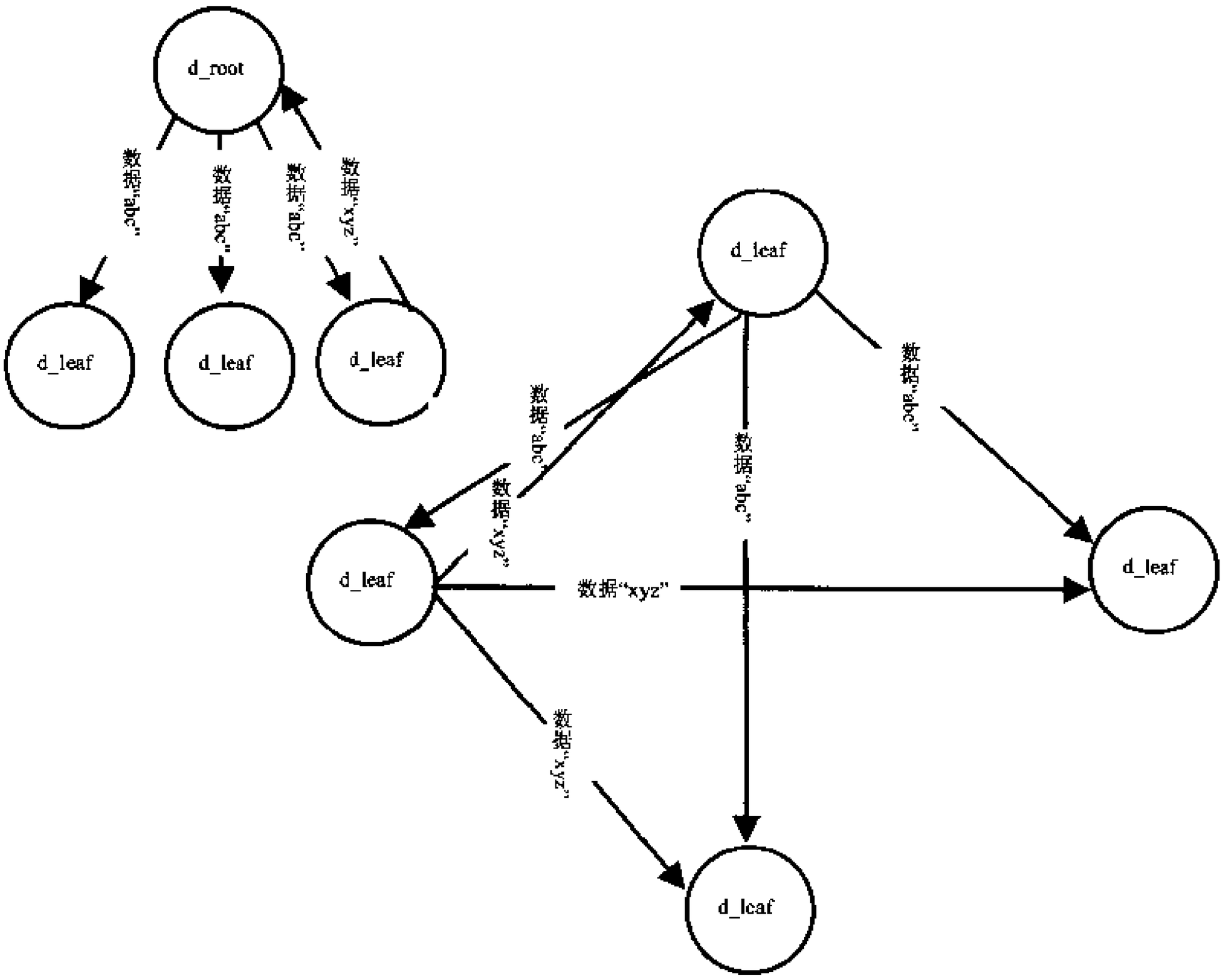


图 9.2 有根和无根数据层面

找到多播属性

第 2 章已讨论了如何列举协议条目，以及如何决定它们的属性。对一种协议来说，与多播有关的全部信息也可从编录的协议条目中取得。在由 WSAEnumProtocols 调用返回的 WSAPROTOCOL_INFO 结构中，dwServiceFlags1 条目包含了几个特殊的标志位，是我们特别感兴趣的。如果设置了 XP1_support_multipoint，表明该协议支持多播。假如设置了 XP1_MULTIPOINT_CONTROL_PLANE

位，表明协议支持的是一种有根控制层面；否则便是无根的。而假如设置了 XP1_MULTIPOINT_DATA PLANE 位，则表明协议支持的是一种有根数据层面。类似地，假如该标志位设被为 0，协议支持的使只有一个无根数据层面。

9.2　IP 多播

IP 多播通信需要依赖一组特殊的地址，名为“多播地址”。我们正是用这个组地址对一个指定的组进行命名。举个例子来说，假定 5 个节点都想通过 IP 多播实现彼此间的通信，它们便可加入同一个组地址。这些节点全部加入组中之后，由一个节点发出的任何数据均会一模一样地复制一份，发给组内的每个成员，甚至包括始发数据的那个节点。多播 IPv4 地址是一个 D 类 IP 地址，范围在 224.0.0.0 到 239.255.255.255 之间；IPv6 多播地址以前缀 FF(1111 1111)开始，这点第 3 章已作了讨论。但是，其中还有许多地址是为特殊用途而保留的。在 RFC1700 和 RFC2375 文件中，提供了所有保留地址的一个详细列表。“Internet 号码分配管理机构”(IANA)负责对这个列表的维护。表 9.1 总结了目前标记为“保留”的一些 IPv4 地址。在实际应用中，因为头 3 个保留多播地址是网络中的路由器专用的，所以可使用除它们之外的任何地址。请参考 RFC1700，了解确切的多播地址分配情况。

表 9.1　保留 IPv4 多播地址

多播地址	用　途
224.0.0.0	基本地址(保留)
224.0.0.1	这个子网上的所有节点
224.0.0.2	这个子网上的所有路由器
224.0.1.1	网络时间协议
224.0.0.9	RIP 第 2 版组地址
224.0.1.24	WINS 服务器组地址

IPv6 多播也将特定多播地址设置为特殊用途，其中的一些列在表 9.2 中。IPv4 及 IPv6 保留多播地址的一个区别，是 IPv6 根据要求的范围(如链接一本地及站点一本地)来提供不同的地址。

表 9.2　IPv6 多播地址

范　围	多播地址	用　途
节点一本地	FF01::1	所有节点地址
节点一本地	FF01::2	所有路由器地址

续表

范　围	多播地址	用　途
链接一本地	FF02::1	所有节点地址
链接一本地	FF02::2	所有路由器地址
链接一本地	FF02::9	RIP 路由器
站点一本地	FF05::2	所有路由器地址
站点一本地	FF05::1:3	所有 DHCP 服务器
站点一本地	FF05::1:4	所有 DHCP 中继

最初开发 TCP/IP 时，由于尚未考虑多播通信，所以后来不得不进行了大量改造，使 IP 能够提供对多播的支持。例如，我们已经发现 IP 要求保留　系列特殊地址，专门用于多播通信。除此以外，针对 IPv4，还专门引入了一个特殊的协议(IGMP)，以便管理多播客户机，以及它们在组内的成员关系。现在，设想位于不同子网的两个工作站想加入同一个多播组，如何通过 IP 来做到这一点呢？此时，不能简单地将数据广播到各处的多播地址了事，否则网络会立即由于数据泛滥而瘫痪。引入 IGMP 的目的正是为了通知路由器：网络中的一台计算机对发给一个指定组的数据有兴趣。最新版本的 IGMP(第 3 版)添加了一项支持，限制接受数据的源。应注意到，可靠多播提供程序仍旧依靠 IGMP 来完成组成员与网络元素的通信。

对于 IPv6，ICMPv6 将各种不同的支持协议如 ICMP、ARP 及 IGMP 合并成　个。多播成员用 MLD 消息来管理，MLD 是在 ICMPv6 协议上发送的一种消息。

9.2.1　支持协议

多播主机利用 IGMP 及 MLD 通知路由器：路由器所在子网的一台计算机想加入一个特定的多播组。IGMP 是 IPv4 多播方案的基础，MLD 则是 IPv6 的基础。要想使多播正常进行，两个多播节点之间的所有路由器都必须支持适当的多播支持协议。例如，对于 IPv4，假定计算机 A 和 B 加入了多播组 224.1.2.3，两者间总共存在着 3 个路由器。此时，3 个路由器都必须支持 IGMP，以保障通信的成功进行。若某个路由器不支持 IGMP，那么收到多播数据后，它会将数据丢弃了事。若一个应用程序加入了多播组，一条 “加入”(或汇报)消息便会发给加入多播所在的接口上的“所有路由器”地址。该命令用于通知所有路由器，有客户机对一个特定的多播地址产生了兴趣，即它想加入那个多播组。以后，假如路由器收到了发给那个多播地址的数据，便会将其转发给带有多播客户机的网络。

此外，若一个端点加入多播组，便会同时指定一个 TTL 参数。通过该参数，我们便知道对于在端点计算机上运行的多播应用程序来说，为了收发数据，中途需要经历多少个路由器。例如，假定我们编写了一个 IP 多播应用程序，令其加入组 X，同时将 TTL 值设为 2。然后将加入命令传给本地子

网上的“所有路由器”组。子网内的路由器会根据这一命令，指示自己以后应将多播数据转发给那个地址。随后，路由器会将 TTL 值减 1，再将一条加入命令传给与自己相邻的各个网络。那些网络上的路由器接到数据后会如法炮制，最后又将 TTL 值减去 1。就我们的例子来说，此时的 TTL 值已经变成了 0，所以“加入”命令不会再继续传递下去。从中可以看出，TTL 实际限制了多播数据能够传播多远。

若一个路由器拥有由工作站注册的一个或多个多播组，便会向“所有主机”地址定时发送一条“组查询”消息，查询当初以一条加入命令通知它的每个多播地址。假如网络上的客户机仍在使用那个多播地址，它们便会用另一条消息作出响应，让路由器继续转发与那个地址相关的数据。否则的话，路由器便会停止为那个地址转发任何数据。IGMP(第 2 版或更高)及 MLD 都支持明确地离开一个组的主机的概念。也就是说，每个接口都保留了一个有多少应用程序被加入到一个特定多播组的参考计数。当计数变为 0，则发出一条离开(或“完成”)消息，这将通知路由器停止向那个组转发数据。应注意到，对于 IGMPv1 而言，不存在显式的离开消息，因此路由器会继续转发数据，即使应用程序已经退出，这将导致不希望得到的结果。只有在它发送出一个组查询之后，路由器才会发现，原来已经没有客户机再收听那个特定的组了。

Windows XP 引入了对 IPv4 多播 IGMP 第 3 版的支持，允许应用程序加入 IPv4 多播组，并列出一个或多个源，可以从这些源接受数据，或拒绝这些源的数据。如果加入了 X 组，该组只指定源 A 及源 B 作为有效源，则只有起源自 A 或 B 的数据才会被传递给应用程序。类似地，如果加入了 X 组，且源 A 及源 B 被排除在外，则除了 A 或 B 外的每个源发送到多播组的数据都会被传递给应用程序。为了使 IGMPv3 能够正常地运行，网络上的路由器必须也支持 IGMPv3。因为路由器将为加入的多播组传播所有数据，不会仅限于来自列表所指定源的数据，所以如果它们不支持 IGMPv3，则不会产生任何实际效果。如果一个 Windows XP 主机所在的网络不支持 IGMPv3，则上机就会后退到现有的 IGMP 版本。

Windows 98、Windows Me 和 Windows 2000 本身便提供了对 IGMP 第 2 版的支持。而对 Windows 95 来说，在最新的 Winsock 2 升级中，也包括了 IGMP 第 2 版。在 Windows NT 4.0 中，自 ServicePack4(SP4)开始，也提供了对 IGMP 第 2 版的支持。以前的 SP 和操作系统本身仅支持第 1 版。如果想了解 IGMP 第 1 及第 2 版的详情情况，不妨分别参阅 RFC1112 以及 RFC2236 两份文档。目前，IGMP 第 3 版是一个 IETF 草案：draft-ietf-idmr-igmp-v3-07.txt。

9.2.2　用 Setsockopt 多播

最初，加入或离开一个多播组的惟一途径是使用 setsockopt API。Winsock 2 引入了一种独立于协议、使用 WSAJoinLeaf API(将在下一节讨论)的多播方法。但很快就可以看到，虽然 setsockopt 方法和使用的协议捆绑得更紧一些，但相对而言它却更灵活一点。

9.2.2.1 IPv4

有两个控制加入和离开组的套接字选项：IP_ADD_MEMBERSHIP 和 IP_DROP_MEMBERSHIP，套接字选项级别为 IPPROTO_IP。输入参数是一个 struct ip_mreq 结构，其定义为：

```
struct ip_mreq {
   struct in_addr imr_multiaddr;    /*组的 IP 多播地址*/
   struct in_addr imr_interface;    /*接口的本地 IP 地址*/
};
```

imr_multiaddr 字段是多播组的以网络字节顺序排序的 32 位 IPv4 地址，imr_interface 字段是本地接口的的 32 位 IPv4 地址(也是按网络字节顺序排序)，从这个地址加入多播组。下述代码片段展示了加入一个多播组的过程。

```
SOCKET s;
SOCKADDR_IN localif;
struct ip_mreq mreq;

s = socket(AF_INET,SOCK_DGRAM,IPPROTO_UDP);
localif.sin_family = AF_INET;
localif.sin_port = htons(5150);
localif.sin_addr.s_addr = htonl(INADDR_ANY);
bind(s,(SOCKADDR *)&localif,sizeof(localif));

mreq.imr_interface.s_addr = inet_addr("157.124.22.104");
mreq.imr_multiaddr.s_addr = inet_addr("234.5.6.7");

setsockopt(s,IPPROTO_IP,IP_ADD_MEMBERSHIP,(char *)&mreq,
sizeof(mreq));
```

应注意到，在加入组前，套接字应该被绑定到通配符地址(INADDR_ANY)。在这个示例中，套接字在本地接口 157.124.22.104 上加入到多播组 234.5.6.7 中。在同一个套接字的同一个接口或不同接口上，可以加入多个组。

在加入一个或多个多播组后，IP_DROP_MEMBERSHIP 选项用来离开一个特定的组。struct ip_mreq 结构再一次成为输入参数。结构的参数为待取消的本地接口和多播组。例如，针对上面的代码示例，下述代码将取消加入前面的多播组：

```
//按照上面的叙述加入组

mreq.imr_interface.s_addr = inet_addr("157.124.22.104");
mreq.imr_multiaddr.s_addr = inet_addr("234.5.6.7");

setsockopt(s,IPPROTO_IP,IP_DROP_MEMBERSHIP,(char *)&mreq,
sizeof(mreq));
```

最后，如果应用程序退出，或套接字关闭，则任何通过该进程或套接字加入的多播组将被清空。

注意　补充文件中的 IP-SETSOCKOPT 目录下提供了一个使用 setsockopt 的 IPv4 多播示例。

9.2.2.2　带多播源的 IPv4

IP 源多播在支持 IGMPv3 协议的系统上可用，它允许套接字在一个接口上加入多播组，同时指定一系列的源地址，来自这些地址的数据将被接受。套接字可在两种模式下加入一个组。第 1 种是 INCLUDE 模式，在这种模式下，套接字加入一个组的同时，指定 N 个有效源地址，来自这些有效源地址的数据将被接受。另一种是 EXCLUDE 模式，在这种模式下，套接字加入一个组的同时，指定 N 个源地址，除这些地址之外，来自其他源地址的数据将被接受。套接字选项根据使用的模式而不同。

如果使用 INCLUDE 模式加入一个多播组，套接字选项为 IP_ADD_SOURCE_MEMBERSHIP 和 IP_DROP_SOURCE_MEMBERSHIP。第 1 步是添加一个或多个源。两个选项都采用 struct ip_mreq_source 结构，它的定义为：

```
struct ip_mreq_source {
   struct in_addr imr_multiaddr;     /*组的 IP 多播地址*/
   struct in_addr imr_sourceaddr;    /*源 IP 地址*/
   struct in_addr imr_interface;      /*接口的本地 IP 地址*/
};
```

imr_multiaddr 和 imr_interface 字段与 struct ip_mreq 结构中的字段相同。而新的字段 imr_sourceaddr 指定 32 位 IP 源地址，从该地址发出的数据将被接受。如果有多个有效源，则使用同一个多播地址和接口连同其他有效源再次调用 IP_ADD_SOURCE_MEMBERSHIP。下述代码在一个本地接口加入一个多播组，带两个有效源：

```
SOCKET s;
SOCKADDR_IN localif;
struct ip_mreq_source mreqsrc;

s = socket(AF_INET,SOCK_DGRAM,IPPROTO_UDP);

localif.sin_family = AF_INET;
localif.sin_port = htons(5150);
localif.sin_addr.s_addr = htonl(INADDR_ANY);
bind(s,(SOCKADDR *)&localif,sizeof(localif));
mreqsrc.imr_interface.s_addr = inet_addr("157.124.22.104");
mreqsrc.imr_multiaddr.s_addr = inet_addr("234.5.6.7");
mreqsrc.imr_sourceaddr.s_addr = inet_addr("172.138.104.10");

setsockopt(s,IPPROTO_IP,IP_ADD_SOURCE_MEMBERSHIP,
       (char *)&mreqsrc,sizeof(mreqsrc));
```

```
mreqsrc.imr_sourceaddr.s_addr = inet_addr("172.141.87.101");

setsockopt(s,IPPROTO_IP,IP_ADD_SOURCE_MEMBERSHIP,
        (char *)&mreqsrc,sizeof(mreqsrc));
```

要从INCLUDE集合中删除一个源，要用多播组、本地接口及要删除的源调用IP_DROP_SOURCE_MEMBERSHIP。

要加入排除了一个或多个源的多播组，要使用 IP_ADD_MEMBERSHIP。使用 IP_ADD_MEMBERSHIP 加入多播组和在 EXCLUDE 模式下加入多播组效果一致，但和后者不同的是，它不排除任何源。送到多播组中的数据都会被接受，而不管源是怎么样的。在加入多播组后，可以调用 IP_BLOCK_SOURCE 来排除给定源。struct ip_mreq_source 结构再一次被当作输入参数，指定要阻塞的源。下述示例先加入一个多播组，再排除一个源：

```
SOCKET s;
SOCKADDR_IN localif;
struct ip_mreq mreq;
struct ip_mreq_source mreqsrc;

s = socket(AF_INET,SOCK_DGRAM,IPPROTO_UDP);

localif.sin_family = AF_INET;
localif.sin_port = htons(5150);
localif.sin_addr.s_addr = htonl(INADDR_ANY);
bind(s,(SOCKADDR *)&localif,sizeof(localif));

//加入一个组——筛选器是 EXCLUDE none
mreq.imr_interface.s_addr = inet_addr("157.124.22.104");
mreq.imr_multiaddr.s_addr = inet_addr("234.5.6.7");
setsockopt(s,IPPROTO_IP,IP_ADD_MEMBERSHIP,(char *)&mreq,
    sizeof(mreq));
mreqsrc.imr_interface = mreq.imr_interface;
mreqsrc.imr_multiaddr = mreq.imr_multiaddr;
mreqsrc.imr_sourceaddr.s_addr = inet_addr("172.138.104.10");
setsockopt(s,IPPROTO_IP,IP_BLOCK_SOURCE,(char *)&mreqsrc,
    sizeof(mreqsrc));
```

如果某个时候应用程序希望接受来自一个以前被阻塞的源的数据，则可以使用 IP_UNBLOCK_SOURCE 调用 setsockopt，将该源从排除集合中删除。输入参数为 struct ip_mreq_source，用来指定可接受的数据源。

注意 补充材料中的 IP-SOURCE 目录下提供了一个使用 setsockopt 的 IPv4 源多播示例。

9.2.2.3　IPv6

IPv6 多播和 IPv4 多播类似，但它的套接字选项名称不同，接受的输入参数也有少许差别。IPv6 的选项是 IPV6_ADD_MEMBERSHIP 和 IPV6_DROP_MEMBERSHIP，选项级别为 IPPROTO_IPV6。

指定多播组和接口的结构是一个 ipv6_mreq 结构，它的定义为：

```
typedef struct ipv6_mreq {
    struct      in6_addr ipv6mr_multiaddr;      /*IPv6 多播地址*/
    unsigned    int ipv6mr_interface;           /*接口索引*/
}IPV6_MREQ;
```

除了本地接口是接口索引，而不是完整的 IPv6 地址之外，这个结构的字段和 IPv4 结构 struct ip_mreq 一致，找到本地 IPv6 地址的接口索引的最容易的途径，是使用第 16 章介绍的 IP 助手 API 即 GetAdaptersAddresses。如果链接—本地地址被用作本地接口，则该链接的范围 ID 就是接口索引　下述示例代码展示了当给定的链接—本地地址为本地接口时，加入一个 IPv6 多播组的过程：

```
SOCKET                  s;
struct ipv6_mreq  mreq6;
struct addrinfo         *reslocal,
                        *resmulti,
                    hints;

s = socket(AF_INET6,SOCK_DGRAM,IPPROTO_UDP);

//获取要绑定到的本地通配符地址(也就是“::”)
memset(&hints,0,sizeof(hints));
hints.ai_family = AF_INET6;
getaddrinfo(NULL,"5150",&hints,&reslocal);
bind(s,reslocal->ai_addr,reslocal->ai_addrlen);
freeaddrinfo(reslocal);

//解析链接-本地接口
getaddrinfo("fe80::250:4ff:fe7c:7036%6",NULL,NULL,&reslocal);
//解析多播地址
getaddrinfo("ff12::1",NULL,NULL,&resmulti);

//加入多播组
mreq6.ipv6mr_multiaddr =
    ((SOCKADDR_IN6 *)resmulti->ai_addr)->sin6_addr;
mreq6.ipv6mr_interface =
    ((SOCKADDR_IN6 *)reslocal->ai_addr)->sin6_scope_id;
setsockopt(s,IPPROTO_IPV6,IPV6_ADD_MEMBERSHIP,
    (char *)&mreq6,sizeof(mreq6));

freeaddrinfo(resmulti);
```

```
freeaddrinfo(reslocal);
```

类似于 IPv4，在 IPv6 中，要离开一个多播组，同样是将 struct ipv6_mreq 传递到 setsockopt，但要使用 IPV6_DROP_MEMBERSHIP 选项。

注意 补充材料中的 IP-SETSOCKOPT 目录下提供了一个使用 setsockopt 的 IPv6 多播示例。这个示例就是 IPv4 多播的那个示例，这是因为示例会从命令行提供的地址推算出使用了哪个地址族(可使用 getaddrinfo API 来使得示例独立于 IP 版本，如第 3 章所述)。

9.2.2.4 用 IPv4 发送多播数据

前面的部分已经展示了如何加入及离开多播组，以使接收发送到该组的数据；然而，应用程序不需要加入多播组，就可向它发送数据。有一个问题必须明白：多重初始地址计算机。如果应用程序创建了一个 UDP 套接字，并用目标地址“234.5.6.7”调用 sendto，应该用哪个接口发送数据呢？基本上，列在路由表中的第 1 个接口就是用来发送数据的接口。为了改变这一状况，应用程序可以使用套接字选项 IP_MULTICAST_IF 来指定外出数据的接口。选项值是本地接口的 32 位 IPv4 地址。下述代码示例展示了如何在一个套接字上设置外出接口。

```
SOCKET          s;
SOCKADDR_IN     dest;
ULONG           localif;
char            buf[1024];
int             buflen = 1024;

s = socket(AF_INET,SOCK_DGRAM,IPPROTO_UDP);
localif = inet_addr("157.124.22.104");
setsockopt(s,IPPROTO_IP,IP_MULTICAST_IF,
     (char *)&localif,sizeof(localif));

dest.sin_family = AF_INET;
dest.sin_port = htons(5150);
dest.sin_addr.s_addr = inet_addr("234.5.6.7");

sendto(s,buf,buflen,0,(SOCKADDR *)&dest,sizeof(dest));
```

注意 补充材料中的多播示例代码也展示了这个套接字选项的使用。

9.2.2.5 用 IPv6 发送多播数据

IPv4 中存在的发送方面的问题 IPv6 多播也存在。发送到一个 IPv6 多播组的数据的外出接口用 IPV6_MULTICAST_IF 选项指定，其输入参数是一个整数，指定用于发送外出数据的适配器接口索引。同时，如果用 sendto 或 WSASendTo 发送数据到多播组，则为地址参数 to 指定的 SOCKADDR_IN6 结构的 sin6_scope_id 字段应该为 0。

注意　补充材料中的多播示例代码也展示了这个套接字选项的使用。

9.2.3　用 WSAIoctl 多播

对应 IPv4 源多播，有一个 I/O 控制命令可用来在调用中指定一个或多个要包含或排除的源地址，这个命令就是 SIO_SET_MULTICAST_FILTER。其输入参数是一个结构，名为 SIO_SET_MULTICAST_FILTER，它的定义为：

```
struct ip_msfilter {
    struct in_addr     imsf_multiaddr;/*组的 IP 多播地址*/
    struct in_addr     imsf_interface; /*接口的本地 IP 地址*/
    u_long          imsf_fmode;  /*筛选器模式——INCLUDE 或 EXCLUDE*/
    u_long          imsf_numsrc;  /* src_list 中的源数量*/
    struct in_addr     imsf_slist[1];
};
```

前两个字段的含义不言自明。第 3 个字段 imsf_fmode 指明，imsf_slist 数组中列出的源地址是多播组要包含的源呢，还是要排除的源。包含源则使用常数 MCAST_INCLUDE，反之，要排除源则使用 MCAST_EXCLUDE。imsf_numsrc 字段指明所提供的源的数量，imsf_slist 字段是源地址的数组。

下述代码示例展示了如何使用这个 I/O 控制命令：

```
SOCKET                    s;
Char                   buf[1024];
struct ip_msfilter     *msfilter;
DWORD                  bytes;
int                    filterlen;

s = socket(AF_INET,SOCK_DGRAM,IPPROTO_UDP);

//将套接字绑定到 INADDR_ANY
msfilter = (struct ip_msfilter *)buf;
msfilter->imsf_multiaddr.s_addr = inet_addr("234.5.6.7");
msfilter->imsf_interface.s_addr = inet_addr("157.124.22.104");
msfilter->imsf_fmode = MCAST_INCLUDE;
msfilter->imsf_numsrc = 3;
msfilter->imsf_slist[0].s_addr = inet_addr("172.138.104.10");
msfilter->imsf_slist[1].s_addr = inet_addr("172.138.104.11");
msfilter->imsf_slist[2].s_addr = inet_addr("172.138.104.12");
filterlen = sizeof(struct ip_msfilter)+
     ((msfilter->imsf_numsrc -1)*sizeof(struct in_addr));

WSAIoctl(s,SIO_SET_MULTICAST_FILTER,(void *)msfilter,filterlen,
    NULL,0,&bytes,NULL,NULL);
```

这个示例加入一个多播组，并指定 3 个有效的接受源。要检索多播筛选器的集合，需要使用 I/O 控制命令 SIO_GET_MULTICAST_FILTER。输入参数必须是一个 struct ip_msfilter 结构，该结构包含本地接口和多播组，从两者上可获得多播状态。如果成功，输出参数将返回一个 struct ip_msfilter 结构，该结构包含组及接口的源的全集。应注意到，必须指定足够大的缓冲区来容纳筛选器及所有的源。

注意 补充材料中的 IP-SOURCE 目录下提供了一个使用 WSAIoctl 的 IPv4 源多播示例。这个示例就是使用 setsockopt 的 IP 源多播示例。“-f”标志指明使用 SIO_SET_MULTICAST_FILTER 代替套接字选项。

9.2.4 用 WSAJoinLeaf 多播

Winsock 2 引入了 WSAJoinLeaf API，它是按照独立于协议的目的而设计的。这个 API 的定义为：

```
SOCKET WSAJoinLeaf(
    IN   SOCKET s,
    IN   const struct sockaddr FAR *name,
    IN   int namelen,
    IN   LPWSABUF lpCallerData,
    OUT LPWSABUF lpCalleeData,
    IN  LPQOS lpSQOS,
    IN  LPQOS lpGQOS,
    IN  DWORD dwFlags
);
```

在很多方面，WSAJoinLeaf 接受与 WSAConnect 一样的参数。不同之处在于，name 参数指明要加入的多播地址，dwFlags 指明套接字是否将在多播会话上收发数据。有效标志值为 JL_SENDER_ONLY、JL_RECEIVER_ONLY 和 JL_BOTH。

对于 IP 多播，必须使用 WSASocket API 创建一个 UDP 套接字，并指定标志 WSA_FLAG_MULTIPOINT_C_LEAF 和 WSA_FLAG_MULTIPOINT_D_LEAF。然后应将套接字绑定到具体的接口，并从该接口加入多播组。然后用多播组的多播地址调用 WSAJoinLeaf。这里有两点差别。首先，应该绑定到具体的本地接口，而不是绑定到通配符地址。其次，没有必要设置外出(发送)接口。加入多播组后，外出接口将被自动设为套接字所绑定到的接口。最后，在一个套接字上只能加入一个多播组。也就是说，跟使用 setsockopt 不一样，这里在一个套接字上 WSAJoinLeaf 只能被调用一次，而且只有关闭套接字时才能离开该多播组——除此以外，没有其他离开多播组的途径。

注意 补充材料中的 IP-WSAJOINLEAF 目录下提供了一个使用 WSAJoinLeaf 的 IP 多播示例，这个示例在 IPv4 和 IPv6 上都能运行。

9.3　可靠多播

安装微软消息队列 (MSMQ)组件之后，Windows XP 就可以支持可靠多播提供程序了。可靠多播是一种建立在 IPv4 多播之上的协议，目前还不支持 IPv6。Windows XP 上可靠多播的实现基于 PGM(Pragmatic General Multicasting，实用常规多播)规范。这个协议是基于 NAK 的，这意味着数据的所有接收者只有在探测到有数据包丢失的时候，才会通知发送者。如果协议是基于收到确认通知的，如 TCP，就可能发生如下情形，即数以百计或可能数以千计的接收者发送数据包给发送者，表示自己已经成功收到数据包，这样将使发送者的网络难以应付。本节只提供对如何实现这种协议的一个简单介绍，以使读者可以对如何提供可靠多播获得一个基本的印象。关于 PGM 协议的完整细节内容，请咨询 IETF 草案：draft-speakman-pgm-spec-06.txt。

可靠多播协议运行时，发送者储存数据，并在指定时限内发送给接收者，这个过程通常被称为发送窗口。这个窗口周期性地推进，窗口内的旧数据将被丢弃，以便为发送者发送的新数据腾出空间。组成发送窗口的变量有 3 个：发送速度、窗口按字节表示的大小及窗口用秒表示的大小(例如，被清除前数据在窗口中保留多久)。可靠多播驱动程序为窗口大小建立了默认值，该默认值可以被重写(因为默认值相当小)。

协议也在数据包中包含了一个顺序号，以便在接收者探测到它遗漏掉某个特定序列之后，可以将一个 NAK 传给发送者，接着发送者就会传输遗漏的数据。应注意到，接受者 NAK 那些刚从发送窗口清除掉的数据是有可能的。遇到这种情况时，因为数据不可恢复而使得接收者的“会话”将被重置——就像 TCP 会话因 ACK 超时或类似错误而被重置一样。

在可靠多播的 Winsock 提供程序方面，有两个重要的问题。首先，不仅存在一个可靠数据报(SOCK_RDM)提供程序，还存在一个流提供程序(SOCK_STREAM)。其次，传递到套接字创建函数的协议值为 IPPROTO_RM。这个协议特有的套接字选项包含在 WSRM.H 中。

下面来看如何创建一个可靠多播发送者及接收者。补充材料的 IP-RM 目录中有完整的代码示例。

9.3.1　可靠发送者

建立可靠多播的发送者是一个简单的过程。下面的清单即包含了必要的步骤。

1. 创建一个可靠多播套接字。
2. 将套接字绑定到 INADDR_ANY。
3. 使用 RM_SET_SEND_IF 来设置外出接口。
4. 将套接字连接到将发生数据传输的多播组地址。

对于 IP 多播，如果是多重初始地址计算机，就有必要设置外出接口。还应该注意，在发送者和

接收者之间，并未建立真正的连接。发送者调用的连接只是简单地将目标多播地址和套接字关联起来。下述代码示例展示了建立一个可靠多播发送者的过程。

```
SOCKET          s;
ULONG           sendif;
SOCKADDR_IN     localif,
                multi;
char              buf[1024];
int               buflen = 1024;

s = socket(AF_INET,SOCK_RDM,IPPROTO_RM);

//绑定到 INADDR_ANY
localif.sin_family = AF_INET;
localif.sin_port = htons(0);
localif.sin_addr.s_addr = htonl(INADDR_ANY);
bind(s,(SOCKADDR *)&localif,sizeof(localif));

//设置外出接口
sendif = inet_addr("157.124.22.104");
setsockopt(s,IPPROTO_RM,RM_SET_SEND_IF,(char *)&sendif,
    sizeof(sendif));

//将套接字连接到多播目标
multi.sin_family = AF_INET;
multi.sin_port = htons(5150);
multi.sin_addr.s_addr = inet_addr("234.5.6.7");
connect(s,(SOCKADDR *)&multi,sizeof(multi));

//发送数据
send(s,buf,buflen,0);

//关闭会话
closesocket(s);
```

这个例子创建了一个 SOCK_RDM 类型的套接字。因为它不仅提供面向消息含义，还显示了可靠性，所以和 SOCK_DGRAM 类似。同时，提供给绑定调用的本地端口并不是很重要。在创建好会话之后，发送者就可以开始发送数据了。

9.3.1.1 修改窗口大小

在默认状态下，由可靠多播驱动程序来设置用来缓冲待发送数据的窗口大小。每隔一段时间，发送窗口便会根据一个增量推进一次，以便抛弃最旧的数据，为新数据腾出空间。这个窗口在数据的可靠度上扮演了一个重要的角色。例如，一个高速率的数据发送者可能会从位于低带宽或拥挤的网络上的接收者那里接收到很多丢失数据的 NAK。如果发送窗口相当小的话，将导致接收者无法继续工作。

为了避免这种情况，发送者可以调整发送窗口的大小，以满足应用程序的需求。这就要通过套接字选项 RM_RATE_WINDOW_SIZE 来完成。其输入参数是一个结构，该结构的定义为：

```
typedef struct _RM_SEND_WINDOW
{
    ULONG RateKbitsPerSec;
    ULONG WindowSizeInMSecs;
    ULONG WindowSizeInBytes;
}RM_SEND_WINDOW;
```

RateKbitsPerSec 字段指定发送者发送数据的速率，在默认情况下是 56Kbps。WindowSizeInMSecs 字段指定在数据被清除以便为新数据腾出空间前，它可以在窗口中停留多长时间(用毫秒表示)。最后一个字段 WindowSizeInBytes 表示在旧数据被清除以便为新数据腾出空间前，可以存储多少旧数据。设置这个选项时，必须满足下述等式：WindowSizeInBytes＝(RateKbitsPerSec/8)×WindowSizeInMSecs。

这个选项必须在套接字连接到多播目标地址前设置——所有发送者的套接字选项必须在启动连接调用前设置。

9.3.1.2　FEC

FEC 是一种方法，这种方法可以从 K 个原始数据包当中创建出 H 个奇偶校验数据包(共计 N 个数据包)，以便于接收者可以用接收到的 N 个数据包中的任意 K 个数据包来重建 K 个原始数据包。例如，如果针对每 4 个原始数据包，就创建额外的 4 个奇偶校验数据包(共计 8 个)，则接收者只需要接收八个数据包中的任意 4 个，就可以重建原始数据。

算法采用数据包级别的里德·所罗门擦除(Reed Solomon Erasure)校正技术。意思是，当 FEC 功能启用后，由于会产生许多额外的奇偶校验数据包，因此会有更多数据包涌入线路，但接收者总是会丢失较小数量的数据包。如果接收者丢失的数据包是产生的 N 个数据包中的一个，则它仍然可以恢复原始数据包，这样就避免了丢失数据的 NAK，也避免了为重新传输而等待。

有两种操作方式可以启用 FEC：主动方式和随选方式。用前一种方式时，发送者始终为所有发送的数据计算奇偶校验数据包。其弊端是，奇偶校验数据包的计算可能是一种需要强化处理器的计算。第 2 种方式是随选的，意思是发送者正常发送数据，直到接收者返回一个 NAK，此时修复的数据将被作为 FEC 数据发送。当大量的接收者位于一个容易丢失数据的网络上，且丢失数据的预期重发量将会很巨大时，通常采用主动 FEC。随选方式是可靠性和网络超载之间的一种折衷方式。当位于同一个网络上的几个客户机丢失属于同一个 FEC 组的不同数据包时，这种方式很有用。在这种情况下，每个接收者为丢失的数据包向发送者发送 NAK，此时发送者将发送单个的 FEC 修复数据包，而这个数据包将满足所有客户机的 NAK 请求。

要在一个套接字上启用 FEC，要使用 RM_USE_FEC 套接字选项。这个选项必须在连接套接字之前设置。输入参数是一个结构，它的定义为：

```
typedef struct _RM_FEC_INFO
{
    USHORT  FECBlockSize;
    USHORT  FECProActivePackets;
    UCHAR   FECGroupSize;
    BOOLEAN fFECOnDemandParityEnabled;
}RM_FEC_INFO;
```

FECBlockSize 字段表示所产生数据包的最大数量。这个字段就是前面讨论的 N 个数据包(例如，原始数据包加上奇偶校验数据包)。然而，当前的实现方案忽略了这个字段，而依靠 FECProActivePackets 和 FECGroupSize 字段的和来确定块的大小。FECProActivePackets 字段表示要生成的奇偶校验数据包的数量——这就是生成的 H 个奇偶校验数据包。FECGroupSize 字段是用来生产奇偶信息的原始数据包的数量——即 K 个原始数据包。最后，fFECOnDemandParityEnabled 字段指明是否允许随选 FEC 请求。

9.3.2 可靠接收者

可靠多播接收者的创建由下面 5 个步骤组成：

1. 创建一个可靠多播套接字。
2. 绑定该套接字，该套接字指定多播组地址，会话将从这个地址上加入。
3. 如果接收者希望从某些特定接口监听(而不是从所有接口监听)，就调用 setsockopt 和 RM_ADD_RECEIVE_IF 来添加每一个接口。
4. 调用 listen()。
5. 用一个接受函数待命。

关于接收者有两个重要的区别需要注意。首先，绑定套接字时，没有指定本地接口。相反，将给出接收者希望加入的多播组。如果 SOCKADDR_IN 结构指定的端口号是 0，则指定多播组的任何可靠多播会话都会被接受。否则，如果给定了具体的端口，则只有在和多播组及端口号匹配时，它才会加入可靠多播会话。

第 2 个区别是，接收者必须添加用来监听传入会话的接口。由于绑定调用被用来指定要加入的多播会话，因此有必要指定用哪些本地接口来监听传入会话。在默认情况下，套接字将在任何本地接口上监听传入连接。如果应用程序希望从某个特定接口接受会话，就必须为每个本地接口调用 RM_ADD_RECEIVE_IF。当然，这一点只有在多重初始地址计算机上才比较重要。RM_ADD_RECEIVE_IF 的变量是用来监听的 32 位以网络字节顺序排序的本地接口。这个选项可被多次调用。RM_DEL_RECEIVE_IF 选项可用来从监听集合中删除一个接口，然而，该选项只有在 RM_ADD_RECEIVE_IF 被事先调用之后，才可以调用。

当一个接收者加入一个可靠多播会话之后，它不一定非要在发送者开始传输数据时就开始接收。

作为每个会话的一部分，可能请求修复的最旧的数据将会被广告。因此，如果客户机在发送者已经开始传输数据之后加入会话，则客户机可能会 NAK 那些已广告的序列号的数据。这通常被称为迟到加入。事实上发送者可以限制迟到加入者能够 NAK 的发送窗口数量(通过 RM_LATEJOIN 选项)。当然，这对应用程序完全透明。在会话被加入时，可靠多播提供程序将自动为所有可用数据 NAK。加入会话后，如果数据丢失且无法修复，会话也将被终止。相反，如果发送者关闭会话，则当所有余下的数据都被传送到应用程序之后，下一次接收操作将失败，错误信息为 WSEDISCONN。

下述代码示例展示了如何创建一个可靠多播接收者：

```
SOCKET              s,
                    ns;
SOCKADDR_IN         multi,
                    safrom;
ULONG               localif;
char                buf[1024];
int                 buflen = 1024,
                    fromlen,
                    rc;

s = socket(AF_INET,SOCK_RDM,IPPROTO_RM);

multi.sin_family = AF_INET;
multi.sin_port = htons(5150);
multi.sin_addr.s_addr = inet_addr("234.5.6.7");
bind(s,(SOCKADDR *)&multi,sizeof(multi));

listen(s,10);

localif = inet_addr("157.124.22.104");
setsockopt(s,IPPROTO_RM,RM_ADD_RECEIVE_IF,
          (char *)&localif,sizeof(localif));

fromlen = sizeof(safrom);
ns = accept(s,(SOCKADDR *)&safrom,&fromlen);
closesocket(s);//不再需要监听了

//开始接收数据...
while (1){
    rc = recv(ns,buf,buflen,0);
        if (rc = = SOCKET_ERROR){
        if (WSAGetLastError() = = WSAEDISCON)
            break;
        else {
            //一个未预料到的错误
```

```
        }
    }
}
closesocket(ns);
```

9.4 ATM 多播

通过 Winsock 实现的本地 ATM 通信也支持多播通信方式，它与 IP 多播在功能上有着显著的不同。前面已经说过，ATM 支持“有根的”控制层面及数据层面。换言之，若建立了一个多播服务器(或者 c_root)，便可由它控制谁能加入一个多播组，以及数据在组内如何传输。

与 IP 多播的一项重要区别在于，在 ATM 网络上，可通过 ATM 实现对 IP 的支持。采用这种配置方式，ATM 网络便能模拟一个 IP 网络，而 IP 地址实际上会被映射成为 ATM 特有的地址形式。若允许通过 ATM 来模拟 IP，那么可考虑继续使用 IP 多播通信。此时，IP 多播会相应地转换到 ATM 层。当然，亦可考虑使用原有的的 ATM 多播通信机制(本节讨论的主题)。如配置成通过 ATM 实现 IP 多播，那么 IP 多播通信的行为和在其他普通 IP 网络上应该一样，因为看起来好像就是在一个 IP 网络上。惟一的区别在于，因为所有多播调用都会被转换成 ATM 自己的命令，所以在此不必使用 IGMP。另外，我们或许能用一个或多个 LANE(LAN emulation，LAN 仿真)网络配置 ATM 网络。设计 LANE 最主要的目的便是让 ATM 表现得像一个“正规”网络，能够处理 IPX/SPX、NetBEUI、TCP/IP、IGMP、ICMP……等协议。在这种情况下，IP 多播与以太网上的 IP 多播通信真的是毫无差别；换言之，IGMP 亦可正常使用。

前面已经提到，ATM 支持有根控制层面和有根数据层面。这意味着在创建一个多播组的时候，需要建立一个根节点，由它负责“邀请”叶节点加入多播组。目前只支持由“根”发起的成员加入。换言之，“叶”不可自己请求加入一个组。除此以外，根节点(作为一个有根的数据层面)只能朝一个方向发送数据——从根到叶。

在 ATM 和 IP 多播之间，一个重大差别是 ATM 不需要特殊地址。惟一的要求是：作为一个“根”，它必须知道自己打算邀请的每一个叶的地址。此外，一个多播组内仅能有一个根节点存在。若另一个 ATM 端点发出了对相同的“叶”的邀请，它俩便会自动与原先的多播组脱离，成为一个单独的组。

用 WSAJoinLeaf 实现 ATM 多播

WSAJoinLeaf API 是使用 ATM 多播的惟一方法。另外，从语义上，建立 ATM 多播是面向客户机—服务器的，但从概念上却相反。服务器先创建一个套接字，然后用 WSAJoinLeaf 对每个它想邀请加入多播组的客户机，启动多播加入操作。每个客户机创建一个套接字，监听，并用一个接受函数待命。如果服务器邀请客户机，接受调用就可以完成。

下面两节叙述了建立一个 ATM 叶节点(客户机)并随后建立根节点(服务器)所需的确切步骤。在补充材料的 ATM 目录下提供了一个 ATM 多播示例。

ATM 叶节点

在多播组内创建一个叶节点是件轻而易举的事情。在 ATM 网络中，作为一个“叶”，必须随时随地监听来自一个“根”的、想要自己加入一个组的邀请。下面列出所需的 4 个步骤：

1. 使用 WSASocket 函数，创建 AF_ATM 地址族的一个套接字，同时设置 WSA_FLAG_MULTIPOINT_C_LEAF 和 WSA_FLAG_MULTIPOINT_D_LEAF 标志。
2. 将套接字同本地 ATM 地址及端口绑定到一起，这是用 bind 函数完成的。
3. 调用 listen。
4. 用 accept 或 WSAAccept 等待邀请。至于具体用哪个命令，要取决于我们使用的是什么 I/O 模型。要想全机了解 Winsock 的 I/O 模型，请参阅本书第 5 章。

建好连接后，叶节点便可开始接收来自根的数据。要记住的是，对 ATM 多播来说，数据流必须是单向的：由根至叶，不可逆反。

注意　目前，在任何给定时刻，Windows 98、Windows Me、Windows 2000 和 Windows XP 仅能支持一个 ATM 叶节点。对任何 ATM“点到多点”会话来说，在整个系统的范围之内，只可以有一个进程成为它的叶成员。

ATM 根节点

根节点的创建过程甚至还要比 ATM 叶节点的创建简单一些。基本步骤如下：

1. 使用 WSASocket 函数，创建 AF_ATM 地址族的一个套接字，同时设置 WSA_FLAG_MULTIPOINT_C_ROOT 和 WSA_FLAG_MULTIPOINT_D_ROOT 这两个标志。
2. 针对希望邀请的每一个端点，都随 ATM 地址一道调用 WSAJoinLeaf。

根节点可根据自己的需要，邀请任意数量的端点加入。然而，对每个叶节点，都必须单独执行一次 WSAJoinLeaf 调用。

9.5　小　　结

若应用程序要求同时与多个端点通信，同时又不想招致广播通信为网络带来的重大开销，那么多播便是很好的一种选择。本章对多播技术进行了定义，并展示了各种不同的多播模型。随后，探讨了不同的 IP 多播选项，包括 IPv4 和 IPv6 多播、IPv4 源多播及可靠多播。最后讨论了 ATM 点到多点通信。

10

常规服务质量

当今，随着多媒体技术的普遍应用和 Internet 的广泛流行，各种应用程序对带宽的要求越来越大，许多网络都越来越不堪重负。在共享媒体网络(如以太网)上，因为所有通信数据都有着相同的地位，所以这个问题尤其突出。即使一个非常简单的应用程序，也可能造成数据在网络上泛滥，造成整个网络的瘫痪。为此，人们提出了 QOS 的概念。QOS 实际是一系列组件，允许对网上的数据进行不同处理，并可为其分配不同的优先级。若一个网络具备 QOS 功能，便可对其进行配置，以便为程序员提供下述能力：

- 禁止非适应性协议(如 UDP)滥用网络资源
- 针对“最大努力”(best-effort)通信，以及高优先级或低优先级的通信，对资源进行明确划分
- 为冠名用户保留资源
- 根据用户分派资源访问的优先级

GQOS(Generic Quality of Service，常规服务质量)是微软对 QOS 的一种实现方案。目前，微软已提供了具有 QOS 能力的 TCP/IP 及 UDP/IP 提供程序，可在 Windows 98、Windows Me、Windows 2000 及 Windows XP 上使用。然而，在 Windows XP 中，IP QOS 提供程序的功能有所缩减，不支持完整的 GQOS 功能，这一点本章稍后将叙述。应注意到，IPv6 提供程序不支持 QOS。因为 ATM 本身使已提供了对 QOS 的支持，所以使用 ATM 协议的微软平台也能访问 QOS。

本章将介绍 QOS 的原理，以及它在 Windows 平台上的实现方式。首先要讨论的是，为了对不同的网络传输(网络通信)进行区别对待，哪些组件是必要的。随后将探讨如何利用 Winsock 接口来编写网络应用程序，使其能够利用这些组件，为一些对时间及网络带宽要求颇为严格的应用程序提供服务。本章的大部分内容都围绕 IP 网络上的 QOS 展开。本章末尾将讨论 QOS 在 ATM 网络上的情况，它与 IP 网络上的 QOS 稍有区别。

注意　贯穿全章，读者都完全可以假定本书讨论的 QOS 都是微软所实现的。

10.1　背 景 知 识

QOS 需要 3 个组件才能正常发挥作用：

- 网络上的设备：例如路由器和网关等等，它们可注意到这种服务上的区别。
- 本地工作站：可为自己放到网络上的通信分派相应的优先级。
- 策略组件：决定谁能使用可用的带宽，以及允许多少人使用。

在深入讨论这些组件之前，首先还是来看看 RSVP。这是在 QOS 发送者及 QOS 接收者之间使用的一种传输协议。RSVP 在 QOS 中扮演了一个非常重要的角色，它是 QOS3 个主要组件的集成。

10.1.1　RSVP

RSVP 类似于一种“粘合剂”，它负责将网络、应用程序以及策略组件粘合成一个整体。通过网络，RSVP 携带着资源预约请求信息，而这个网络可能由不同的媒体构成。沿着数据路径，RSVP 可将一名用户的 QOS 请求传播到配置了 RSVP 功能的所有网络设备上，并允许将资源从所有 RSVP 设备中“预约”出来。这样，网络节点便可根据网络的现状，判断出它是否能达到要求的服务级别。

RSVP 协议在预约网络资源时，需要贯穿全网，建立端到端的数据流。这里的数据流实际就是一条网络路径，它和一个或多个发送者、一个或多个接收者以及一个特定的 QOS 级别关联在一起。作为发送方主机，假如希望自己发出的数据分派到一个特定级别的 QOS，则需向目标接收者(可能不止一个)发送一条 PATH(路径)消息。在这条 PATH 消息中，主要包含了对带宽的要求。沿着这条路径，相关的参数便会传播至目标接收者。

作为一个接收方主机，假如对这个数据有兴趣，便可为数据流保留相应的资源(以及自发送者那里来的完整路径)。因此，它需要发回一条 RESV(即“预约”)消息，对发送者作出回应。此后，中间的 RSVP 设备便必须判断自己是否能提供要求的带宽，并核查发出资源请求的用户是否得到允许。假如提供请求的带宽毫无问题，而且根据用户的安全策略设置，表明用户有权发出这样的请求，那么位于中途的每个 RSVP 设备都会腾出需要的资源，并将 RESV 消息传回当初的发送者。

发送者收到 RESV 消息之后，QOS 数据便可开始流动了。在这个信息流内的每个端点都会定期发送 PATH 和 RESV 消息，重申资源预约，以及在可用带宽级别发生更改之后，提供相关的网络信息。此外，通过 PATH 和 RESV 消息的定期刷新，RSVP 协议便能一直保持动态。假如出现了一条更好(例如更快)的传输路径，刷新过的消息便能发现那条新路由。本章后面讨论 Winsock 提供的 QOS 机制时，还会将重点放在 RSVP 协议上，并讲解如何用 Winsock API 调用来调用它。

对于会话设置和 RSVP 来说，需要注意的一个重要方面是：资源的预约永远都是单向的。即便应用程序同时为数据发送及接收请求带宽，预约仍然是单向进行的。此时会为发送请求初始化一次会话，

并为接收请求初始化另一次会话。在本章后面，还会详细论述 RSVP 会话的初始化要求。

10.1.2 网络组件

为使“端到端”的 QOS 能正常运作，两个端点之间的网络设备必须能对数据通信的优先级加以区分。只有这样，它们才能在满足 QOS 要求的前提下，对应用程序要接收的数据进行正确的路由。除此以外，在应用程序发出带宽请求之后，这些网络设备必须能判断出网络中是否存在足够的带宽。为满足这一系列要求，下面这些组件已经被创建出来：

- 802.1p：在子网中区分数据包优先次序的一种标准，它在包的 MAC(media access control，介质访问控制)头中另行设置了 3 个位(二进制位)。
- IP 优先级：用于为 IP 包建立优先级的一种方法。
- 第 2 层传信：将 RSVP 对象映射成本地 WAN QOS 组件(位于网络的 OSI 第 2 层)的一种机制。
- SBM(Subnet Bandwidth Manager，子网带宽管理器)：用于对共享媒体网络带宽进行管理的组件。
- 资源预约协议(RSVP)：用于携带 QOS 请求以及相关信息的一种协议。沿着发送者与接收者之间的数据路径，这些信息会传递给配备了 QOS 能力的网络设备。注意发送者只能有一个，但接收者可以有多个。

10.1.2.1 802.1p

若要确保 QOS 要求的质量等级，同时避免对所有数据包进行同等对待，很大一部分因素取决于网络 HUB 和交换机。HUB 和交换机均位于 OSI 参考模型的第 2 层，所以它们只知道位于每个包开头的媒体访问控制头内的字段。

802.1p 是为网络数据包分配优先等级的一种标准，它需要在 MAC 头内设置一个总长 3 位的值(优先位)。在非 802.1p 网络中，假如一个子网变得堵塞，那么交换机和路由器便不能跟上数据的传输，同时会造成通信的延迟。而另一方面，在 802.1p 网络中，根据优先位的设定，交换机和路由器可对进入的数据进行优先级的分派，优先对待那些优先等级较高的数据包。

假如要为 QOS 实现 802.1p，那么硬件必须具备特殊的能力，可识别这个 3 位字段。NIC(Network interface card，网络接口卡)、网络驱动程序以及网络交换机都必须具备 802.1p 功能。

10.1.2.2 IP 优先级

IP 优先级(IP Precedence)是从一个比 802.1p 高的层次对优先值进行指定的方法。利用这种方法，通过 OSI 模型第 3 层设备(例如路由器)的数据包可以分别拥有不同的优先级设置。为实现 IP 优先级，需要在 IP 头内设置“服务类型”(TOS)字段，以建立不同的优先等级。这样，等级较高的通信会从路由器那里获得质量更好的服务。

和 802.1p 一样，为了使 IP 优先级正常工作，网络上的所有第 3 层设备都必须能够理解 IP 优先位

的设定，并以优先位为准，对传输的数据进行区别对待。

10.1.2.3　第 2 层传信

若数据通过一个广域网(WAN)传输，那么第 2 层传信是必要的。通常，一个 WAN 同时链接了数个网络，那些网络使用的硬件设备则种类繁多。因此，在数据传输的时候，WAN 应该能够同时处理第 1、第 2 以及第 3 层信息。为保障端到端的 QOS，WAN 链接必须理解 QOS 通信的优先等级。为达到这个目的，QOS 提供了一种方法，可将 RSVP 和其他 QOS 参数映射成 WAN 本身能够理解的第 2 层传信方法——WAN 技术利用这些方法来实施自己的 QOS。

10.1.2.4　SBM

SBM 负责对一个指定的共享媒体网络(如以太网)上的资源进行管理。SBM 也要面向 QOS 应用程序，负责以策略为基础的访问权限控制。在共享媒体网络上，SBM 是至关重要的一环，因为假如端点为某个应用程序请求 QOS，那么每个网络设备都要根据自己专用资源的分配情况，要么许可、要么拒绝这一请求。但网络设备不清楚共享媒体上的可用资源成为这些设备的一个调度程序后，SBM 使解决了这个问题。SBM 还与 ACS(Admission Control Service，许可控制服务)紧密联系在一起，该服务属于策略组件的一部分。SBM 必须通过检查，确保请求带宽的那个应用程序(或对应的用户)拥有足够的权限，有权作出这样的要求。要注意的是，一个网络的 SBM 完全可能是一个运行 Windows 2000 Server 的主机。

10.1.3　应用组件

现在，大家已经知道了如果要提供对 QOS 的支持，对网络有哪些要求。接下来，必须考虑本地系统如何根据应用程序请求的 QOS 等级，对数据的优先级进行分派。要想使本地系统支持 QOS，下述组件是必需的：

- GQOS 服务提供程序：负责调用其他 QOS 组件的一个服务提供程序。
- 通信控制模块：该模块负责控制离开计算机的通信数据。这个模块包括 GPC(Generic Packet Classifier，常规包分类器)、包调度器以及包封装器。
- RSVP：这个协议由 GQOS 服务提供程序激活，并携带网络上的预约请求信息。
- GQOS API 函数：提供 GQOS 的编程接口，如 Winsock。
- 通信控制(TC)API 函数：为通信控制组件提供编程接口，对本地主机上的数据传输进行管理。

10.1.3.1　GQOS 服务提供程序

GQOS 服务提供程序是一种 GQOS 组件，可调用几乎所有最终的 QOS 程序。GQOS 服务提供程序可初始化通信控制功能(如果合适的话)，同时为所有 GQOS 功能实施、维护以及控制 RSVP 传信。

要想在主机上找到支持 QOS 的服务提供程序，可用 WSAEnumProtocols 函数查询提供程序编录。用来核实某个提供程序是否支持 QOS 的标志位于 WSAPROTOCOL_INFO 结构之内，该结构正是自

WSAEnumProtocols 调用返回的。我们在此感兴趣的字段是 dwServiceFlags1，看看它的值是否为 XP1_QOS_SUPPORTED 就行了。如果是，便表明该提供程序支持 QOS。要想了解 WSAEnumProtocols 的详情，可参考第 2 章。

10.1.3.2 通信控制模块

TC(Traffic Control，通信控制)在 QOS 中扮演了一个至关重要的角色。在 TC 内，在启用 TC 的那个网络节点之内和之外的所有数据包都需要定义优先级。对数据包的优先级处理产生的影响随着它们流经系统和网络而最终传遍整个网络，从而直接对 QOS 的特征产生了影响。TC 模块通过 3 个模块来实现：GPC、包调度器以及包封装器。

1. GPC

GPC 的职责是对网络组件内的数据包进行分类，并分配它们的优先等级。GPC 定义优先级的依据是 CPU 时间或向网上传输数据这样的活动。为完成优先级的定义，GPC 需要创建索引表，对网络堆栈内的服务进行分类。这是为网络通信划分优先等级的第 1 个步骤。

2. 包调度器

包调度器(Packet Scheduler)控制着数据的传输方式，这是 QOS 的一项关键功能。包调度器实际是一个通信控制模块，规定一个应用程序或者一个数据流允许多少数据，本质上为一个特定的数据流强加了 QOS 参数集合。

包调度器采用由 GPC 提供的优先级分配方案，并为不同的优先级提供不同等级的服务。例如，由 GPC 分类为高优先级的数据会在包调度器内得到优先对待。

3. 包封装器

包封装器(Packet Shaper)的作用是调整数据从数据流到网络的传输。大多数应用程序都是以“突发”形式来读写数据的；然而，许多 QOS 应用程序都需要以固定的速度，来进行数据的发送。因此，包封装器需要在一个时间范围内，对数据的传输事先作好安排，使网络使用尽可能地平稳，营造出负载比较平衡的一个网络。

10.1.3.3 通信控制 API

通信控制 API 是本地主机上用来调整网络通信的那些组件的接口，其中包括用于处理常规包分类器、包调度器以及包封装器的方法。有些通信控制函数是通过对支持 GQOS 的 Winsock 函数的调用而被隐式调用的，此时由 GQOS 服务提供程序为这些支持 GQOS 的 Winsock 函数提供服务。然而，假如一个应用程序需要直接操作通信控制(TC)组件，便可考虑直接使用这些通信控件 API 函数。不过，对这些函数的深入探讨已超出了本书的范围(请参考 Platform SDK(Software Development Kit，软件开发工具包)，获得更多内容)。至于 Winsock GQOS API 的情况，本章稍后便会详加论述。

10.1.4　策略组件

策略(Policy)组件是 GQOS 的第 3 个也是最后一个组件，它负责将资源分配给支持 QOS 的应用程序。策略组件是系统管理员最感兴趣的东西，他们需要根据用户，或根据应用程序请求的带宽类别，对资源的分配加以控制。策略组件包括：

- ACS：一种 Windows 2000 服务器服务，用于截获 RSVP PATH 和 RESV 消息，对 QOS 客户机的访问加以控制，将访问划分为通过 QOS 提供的各个保证等级。
- LPM(Local Policy Module，本地策略模块)：面向 SBM，以通过 ACS 配置的策略为准，提供资源访问决策服务。
- PE(Policy Element，策略元素)：驻留于客户机，提供身份认证信息，以完成预约请求。

10.1.4.1　ACS

ACS 面向支持 QOS 的应用程序，对网络带宽的使用进行调整。这是通过 RSVP 协议来完成的。ACS 会同时截获 PATH 和 RESV 消息，校验发出请求的应用程序是否拥有足够的权限。截获到一条 RSVP 消息后，便会传递给 LPM，由后者进行实际的身份认证。

ACS 驻留于安装了 Windows 2000 操作系统的计算机上，可由系统管理员加以控制，系统管理员可分别针对用户、应用程序或者用户组，对它们使用的资源加以限制。

10.1.4.2　LPM

LPM 与前述的 ACS 存在着紧密的联系。因为先要由 ACS 截获 RSVP 消息，插入用户信息，再传递给 LPM。之后，LPM 会在 Active Directory 里找到用户，以验证其策略信息。假如申请的网络资源可用(由 SBM 决定)，而且身份资料正确无误(拥有足够的权限)，那么由 ACS 截获到的 RSVP 消息便会传给“下一跳”。当然，假如用户不够资格申请一个特定的 QOS 等级，便会生成一个错误，并在 RSVP 消息里返回。

注意　为保证策略的成功检查，用户必须是 Windows 2000 域的成员。

10.1.4.3　策略元素

在这个组件中，实际包含了 LPM 请求的策略信息。因为这些数据结构主要与网络资源的管理有关，而那并非本书的主题，所以本书不会讨论它们的详情。

10.2　QOS 和 Winsock

前一节已探讨了成功构建一个端到端 QOS 网络所需的各种组件。接下来，重点将转向 Winsock 2。利用这套 API 可从一个应用程序中实现对 QOS 的访问。首先讨论的是大多数 Winsock 调用都要用到

的顶级 QOS 结构。接下来，讨论那些能在套接字上调用 QOS 的 Winsock 函数，同时讲解在套接字上启用了 QOS 之后，如何再将 QOS 终止。我们要讨论的最后一个问题是提供程序特有的一些对象，可用来影响 QOS 服务提供程序的一些行为，或影响从它们那里返回的信息。

表面看，先讨论主要的 QOS 结构，再讨论 QOS 函数，最后再回过头来讨论提供程序特有的结构，似乎有些杂乱。但是，这样做是为了让大家首先全面理解那些主要结构如何与 Winsock API 调用进行交互，再来接触提供程序特有的一些选项的细节。

10.2.1 QOS 结构

对 QOS 编程来说，其中心结构便是 QOS 结构，该结构包括：

- 一个 FLOWSPEC 结构，用于描述应用程序用以发送数据的 QOS 等级。
- 一个 FLOWSPEC 结构，用于描述应用程序用以接收数据的 QOS 等级。
- 一个提供程序专用的缓冲区，用来指定各个提供程序具有的某些 QOS 特征(在讨论提供程序的一节里将介绍这些特征)。

10.2.1.1 QOS

QOS 结构同时为数据的发送及接收指定相应的 QOS 参数。它的定义如下：

```
typedef struct _QualityOfService
{
   FLOWSPEC SendingFlowspec;
   FLOWSPEC ReceivingFlowspec;
   WSABUF ProviderSpecific;
}QOS,FAR *LPQOS;
```

其中，FLOWSPEC 结构定义了通信的特征，以及每个传输方向的具体要求。而 ProviderSpecific 字段用于返回信息，并可更改 QOS 的行为。这些取决于不同提供程序的选项将在本章后面部分讲述。

10.2.1.2 FLOWSPEC

FLOWSPEC 是一种基本结构，用于对单个流进行描述。要记住的是，一个数据流描述的是朝一个方向传输的数据。该结构定义如下：

```
typedef struct _flowspec
{
   ULONG       TokenRate;
   ULONG       TokenBucketSize;
   ULONG       PeakBandwidth;
   ULONG       Latency;
   ULONG       DelayVariation;
   SERVICETYPE ServiceType;
   ULONG       MaxSduSize;
```

```
    ULONG       MinimumPolicedSize;
}FLOWSPEC,*PFLOWSPEC,FAR *LPFLOWSPEC;
```

接下来，让我们看看 FLOWSPEC 结构每个字段的含义。

TokenRate

TokenRate 字段定义了数据的传输速率，单位是“字节/秒”。Token 是“令牌”的意思。应用程序可以按此速度传输数据，但假如由于某种原因，它需要用一个较低的速度传输，便可积累下额外的令牌，以便更多的数据能在以后传输出去。然而，一个应用程序可累积的令牌数量要受 PeakBandwidth 字段的制约。同时，令牌本身的尺寸也要受到 TokenBucketSize 字段的限制。通过对令牌总量加以限制，便可防止不活动的数据流积下大量令牌，因为这样极易造成可用带宽的极大浪费。尽管数据流可在一定时间内，(以 TokenRate 值表示的速度)积累传输所需的“信用点”，但最多只能积累到由 TokenBucketSize 规定的上限，而且由于它们的“峰值传输”受到 PeakBandwidth 的限制，数据流不能一次发送任意多的数据。所以传输控制以及网络设备资源的完整性得到了有效保证。而网络设备资源完整性得到了保证，是由于这些设备不会受到突发性的“峰值”传输量的冲击。

由于采取了这些限制措施，应用程序只有在累积了足够的信用点之后，才能开始传输数据。假如没有达到对信用点数量的要求，那么对应用程序而言，要么等积累到足够多的信用点再发送数据，要么完全放弃数据的传输。通信控制(TC)模块负责决定对那些等待许久都没有发送出去的数据，该如何进行处理。因此，在设计应用程序的时候，就应注意将 TokenRate 请求设置成一个合理的数量。

假如应用程序不要求对传输速度进行安排，可将这一字段设为 QOS_NOT_SPECIFIED(-1)。

TokenBucketSize

如前所述，TokenBucketSize 字段面向一个指定的流，限制它能累积的“信用点”数量。例如，对一个视频应用程序而言，如果要求每个视频帧最好都能一次传送出去，可考虑将该字段设为准备传输的视频帧的大小。而对于那些要求数据传输速率固定的应用程序，应考虑将该字段设置成允许一些变化。类似于 TokenRate 字段，TokenBucketSize 字段采用的单位也是“字节/秒”。

PeakBandwidth

PeakBandwidth 字段规定了在指定时间范围内，最多能传送多少数据。事实上，这个值对应着“突发”传输允许的最大数据量。因为可用这个值防止应用程序积累数量众多的传输令牌，然后一起传送出去，造成网络上的数据泛滥，所以它是一个至关重要的值。PeakBandwidth 的采用的单位仍然是“字节/秒”。

Latency

Latency 字段规定了从一个位传送出去之后，在接收者(可能有多个)收到它之前，最多允许存在多长时间的延迟。至于具体怎样对该值进行解释，由 ServiceType 字段中请求的服务等级决定。Latency 字段采用的单位是毫秒。

DelayVariation

DelayVariation 字段指定一个数据包允许的最短及最长延迟之间的差距。通常，应用程序依据这个值决定应安排多大的缓冲区空间，来接收数据，同时仍然维持最初的数据传输形式。DelayVariation 的单位也为毫秒。

ServiceType

ServiceType(服务类型)字段定义了数据流要求的服务等级。可指定下述服务类型：

- SERVICETYPE_NOTRAFFIC：表明沿这个方向，不会有数据传输出去。
- SERVICETYPE_BESTEFFORT：指出服务类型取决于 FLOWSPEC 结构中规定的参数。系统会作出恰当的努力，来维持那个服务等级。然而，系统却不会对数据包是否能正常投递出去提供任何保障。
- SERVICETYPE_CONTROLLEDLOAD：表明数据传输的质量非常近似于在“无负担通信”的条件下，由“最大努力”服务所提供的质量。所谓“无负担通信”，通常可分为两种情况。第 1 种情况，数据包的丢失接近传输媒介正常的出错率；第 2 种情况，通信延迟不会大幅超过投递数据所经历的最小延迟时间。
- SERVICETYPE_GUARANTEED：在连接建立期间，严格保证数据传输以 TokenRate 字段规定的速度进行。不过，假如实际的数据传输速度超过 TokenSize，那么数据可能延迟，也可能被丢弃(具体由通信控制或 TC 配置决定)。除此以外，假如没有超过 TokenRate 的设定，也必须保证 Latency。
- SERVICETYPE_QUALITATIVE 和 SERVICETYPE_BESTEFFORT 表现类似。
- SERVICETYPE_NETWORK_CONTROL 具有最高的优先级，通常只用来控制网络。一般情况下不要使用。

除了这 6 种服务类型之外，另外还有几个标志，可用来提供返回给应用程序的信息。这些信息性标志可与任何有效的 ServiceType 标志进行“或”运算。表 10.1 列出了这些信息性的标志。

表 10.1 服务类型修改符标志

值	含 义
SERVICETYPE_NETWORK_UNAVAILABLE	指出在数据的发送或接收方向，服务不可用
SERVICETYPE_GENERAL_INFORMATION	指出一个数据流支持的所有服务类型
SERVICETYPE_NOCHANGE	指出请求的 QOS 服务等级没有发生变化。这个标志可由一次 Winsock 调用返回。另外，应用程序可通过与 QOS 的重新协商，来指定这个标志，表明在指定的方向上，QOS 等级没有发生变化

续表

值	含　义
SERVICETYPE_NONCONFORMING	这个标志未使用
SERVICE_NO_TRAFFIC_CONTROL	应用程序可用这个标志要求系统立即调用通信控制(TC)，而不是进行“最大努力”通信，除非一条 RESV 消息抵达
SERVICE_NO_QOS_SIGNALING	这个标志禁止发出任何 RSVP 传信消息。尽管可调用本地通信控制，但却不能发出 RSVP PATH 消息。这个标志亦可与一个负责接收的 FLOWSPEC 结构联合使用，从而禁止 RESV 消息的自动生成。应用程序会收到通知，知道一条 PATH 消息已经抵达，然后要执行 WSAIoctl(SIO_SET_QOS)，取消对这个标志的设置，允许 RESV 消息送出

MaxSduSize

MaxSduSize 字段指出在某个流内传输时，数据包的最大长度是多少。 MaxSduSize 以字节数为单位。

MinimumPolicedSize

MinimumPolicedSize 字段定义了某个流内允许的最小数据包长度。MinimumPolicedSize 的值也以字节数为单位。

10.2.2　QOS 调用函数

现在，假定让应用程序在网络上发出一个请求，提出特定的带宽要求。有 4 个函数可以初始化这个过程。一旦启动了一个 RSVP 会话之后，应用程序便可注册 FD_QOS 事件。QOS 状态信息和错误代码会以 FD_QOS 事件的形式，传输给应用程序。应用程序可注册以普通方式来接收这些事件：在 WSAAsyncSelect 或 WSAEventSelect 函数的事件字段内，设置 FD_QOS 标志。

假如用 FLOWSPEC 结构来建立一个连接，同时指定的是默认值(QOS_NOT_SPECIFIED)，亦即“没有指定 QOS”，那么 FD_QOS 通知就显得特别重要。应用程序发出了对 QOS 的请求之后，位于基层的提供程序便会定时更新 FLOWSPEC 结构，指出当前的网络状态，并会利用一个 FD_QOS 事件，向应用程序发出通知。通过这种信息，应用程序便可请求或修改 QOS 级别，以反映出可用的带宽大小。在此要注意的是，更新信息只指出了本地可用的带宽有多大，并不一定同时反映端到端通信

的带宽。

建立了一个数据流之后，可用的网络带宽可能发生变化，或者参与通信的某一方临时改变了主意，想更改要求的 QOS 服务等级。对已分配资源的一次重新协商，会导致 FD_QOS 事件的生成，向应用程序指出发生了什么改变。此时，应用程序应调用 SIO_GET_QOS，以取得新的资源等级。本章后面讲到 QOS 编程时，还会继续讲述 QOS 事件传信以及状态信息。

10.2.2.1 WSAConnect

客户机可使用 WSAConnect 函数，初始化与一个服务器的单播(也称为单点传送)QOS 连接。WSAConnect 的定义如下：

```
int WSAConnect (
   SOCKET s,
   const struct sockaddr FAR *name,
   int namelen,
   LPWSABUF lpCallerData,
   LPWSABUF lpCalleeData,
   LPQOS lpSQOS,
   LPQOS lpGQOS
);
```

要求的 QOS 值通过 lpSQOS 参数传递过去。目前，QOS 组尚未得以支持或实现；所以对 lpGQOS 这个参数而言，应该为其传递一个空值(NULL)。

其中，WSAConnect 调用既可用于面向连接的套接字，也可用于无连接的套接字。对面向连接的套接字来说，该函数除建立连接之外，还会生成适当的 PATH 以及(或者)RESV 消息。而对无连接的套接字来说，必须将一个端点的地址同套接字关联到一起，让服务提供程序知道应该将 PATH 和 RESV 消息发送到哪里。若在一个无连接的套接字上使用 WSAConnect，要注意的一个问题是，只有传给那个目标地址的数据才会由系统根据套接字的既定 QOS 等级进行封装。换句话说，WSAConnect 用于联系一个无连接套接字上的一个端点，在套接字的有效期内，数据只能在那两个端点之间传递。如果需要在保证 QOS 的前提下将数据发给多个端点，请用 WSAIoctl 和 SIO_SET_QOS 来指定每一个新端点。

10.2.2.2 WSAAccept

WSAAccept 函数负责接受一个具备 QOS 功能的客户机的连接。该函数的原型如下：

```
SOCKET WSAAccept(
   SOCKET s,
   struct sockaddr FAR *addr,
   LPINT addrlen,
   LPCONDITIONPROC lpfnCondition,
   DWORD dwCallbackData
```

```
);
```

如果想支持条件函数，必须建立如下原型：

```
int CALLBACK ConditionalFunc(
   LPWSABUF lpCallerId,
   LPWSABUF lpCallerData,
   LPQOS lpSQOS,
   LPQOS lpGQOS,
   LPWSABUF lpCalleeId,
   LPWSABUF lpCalleeData,
   GROUP FAR *g,
   DWORD dwCallbackData
);
```

QOS 服务提供程序的弊端是，并不保证能够返回客户机通过 lpSQOS 参数实际请求的 QOS 值。也就是说，要想在客户机套接字上启用 QOS，在 WSAAccept 调用之前以及之后，必须调用 WSAIoctl，同时设置 SIO_SET_QOS。假如在监听套接字上设置 QOS，在默认情况下，那些值将会被复制给客户机套接字。

但是，即使已经有一个 PATH 消息抵达，QOS 服务提供程序也不会将有效的 QOS 参数传递给条件函数。总之，请不要使用 WSAAccept 条件函数。

若在 Windows 98 上使用 WSAAccept，那么必须特别注意这个问题。但假如确实随 WSAAccept 使用了条件函数，同时 lpSQOS 参数不为 NULL，则必须设置 QOS(使用 SIO_SET_QOS)，否则 WSAAccept 调用会失败。

10.2.2.3　WSAJoinLeaf

WSAJoinLeaf 用于多点通信。第 9 章曾在一个更深入的层次上，探讨了多播的问题。该函数的定义如下：

```
SOCKET WSAJoinLeaf(
   SOCKET s,
   const struct sockaddr FAR *name,
   int namelen,
   LPWSABUF lpCallerData,
   LPWSABUF lpCalleeData,
   LPQOS lpSQOS,
   LPQOS lpGQOS,
   DWORD dwFlags
);
```

要想使一个应用程序加入多播会话，必须创建一个套接字，同时设置相应的标志(WSA_FLAG_MULTIPOINT_C_ROOT 、 WSA_FLAG_MULTIPOINT_C_LEAF 、 WSA_FLAG_

MULTIPOINT_D_ROOT 和 WSA_FLAG_MULTIPOINT_D_LEAF)。应用程序设置多点通信的时候，需要在 lpSQOS 参数中设定 QOS 参数。

用 WSAJoinLeaf 加入 IP 多播组时，多播组的加入操作以及 QOS RSVP 会话的设置是分开来进行的。事实上，一个多播组的加入也只是有可能成功。函数返回的时候，预约请求尚未完成。在以后的某个时间，应用程序会接收到一个 FD_QOS 事件，通知应用程序早先请求的资源分配操作到底是成功，还是失败。

在此请留意针对多播数据设置的 TTL。假如计划使用 SIO_MULTICAST_SCOPE 或 IP_MULTICAST_TTL 来设置 TTL，那么 TTL 必须在调用 WSAJoinLeaf 或者调用 I/O 控制命令 SIO_SET_QOS 对套接字的 QOS 进行设置之前进行设置。假如在 QOS 已经设好之后再来设置 TTL，那么除非 QOS 通过 SIO_SET_QOS 进行了重新协商，否则 TTL 不会产生任何效果。TTL 设置的值也会通过 RSVP 请求传递出去。

在套接字上设置 QOS 之前，便应该先完成 TTL 的设置，因为套接字上的 TTL 设置也会对 RSVP 消息的 TTL 产生影响，从而直接影响资源预约请求最终能传播到多少个网络中去，所以这是至关重要的一个事项。举个例子来说，如果希望在一个 IP 多播组内设置几个端点，而那个多播组同时跨越了 3 个网络。此时，最理想的设定是让 TTL = 3，使我们以后产生的任何网络通信都不会传播至其他不相干的网络。假如在调用 WSAJointLeaf 之前，没有设置 TTL，那么在 RSVP 消息送出的时候，会自动采用默认的 TTL 设置即 63。显然，这会造成主机从跨越许多个网络的地方预约资源，从而造成通信效率的严重下降。

10.2.2.4 WSAIoctl

在设置了 I/O 控制选项 SIO_SET_QOS 的前提下，如果调用 WSAIoctl 函数，那么既可用来在一个已建立连接或未建立连接的套接字上首次请求 QOS，也可用来在初次 QOS 请求完成以后，对 QOS 的要求进行重新协商。使用 WSAIoctl 的一个好处在于，假如 QOS 请求失败，那么经由提供程序所特有的一些信息，更详细的错误信息可在失败后返回，供我们参考，找出错误出在哪里。第 7 章已探讨了 WSAIoctl 函数的详情，并解释了具体如何调用它(即设置 SIO_SET_QOS 或 SIO_GET_QOS)。

SIO_SET_QOS 选项用于在一个套接字上设置或修改 QOS 参数。用 SIO_SET_QOS 调用 WSAIoctl 的一个好处在于，可指定提供程序所特有的一些对象，以便进一步确定 QOS 的行为。下一节将专门讲述提供程序所特有的全部对象。特别要指出的是，假如一个应用程序采用的是无连接套接字，而且不愿意使用 WSAConnect，那么可用 SIO_SET_QOS 调用 WSAIoctl，同时在提供程序专用缓冲区内，指定目标地址对象，从而与一个端点联系到一起，最终促使 RSVP 会话的建立。设定 QOS 参数时，应以 lpvInBuffer 的形式，传递 QOS 结构，同时用 cbInBuffer 指定传递的字节数量。

SIO_GET QOS 选项在接收到 FD_QOS 事件之后使用。若某个应用程序收到了这种事件通知，就应利用 SIO_GET_QOS，发出对 WSAIoctl 的一个调用，对产生该事件的原因进行调查。就像早先指

出的那样，产生 FD_QOS 事件的原因在于网络中可用带宽发生了变化，或者通信对方进行了重新协商。要想获得一个套接字的 QOS 值，以 lpvOutBuffer 的形式传递一个足够大的缓冲区，同时用 cbOutBuffer 指定缓冲区具体大小即可。输入参数可为 NULL，或为 0。调用 SIO_GET_QOS 时，要注意的一点是必须传递一个足够大的缓冲区，以便装下整个 QOS 结构，包括具体提供程序特有的对象。ProviderSpecific 字段(亦即一个 WASBUF 结构)位于 QOS 结构之内。如果将 len 字段设为 QUERY_PS_SIZE，同时 buf 字段为 NULL，那么从 WSAIoctl 返回后，len 字段便会更新，变成需要的大小。此外，假如缓冲区过小，造成调用失败，那么 len 字段的值也会更新为正确的大小。只有在 Windows 2000 和 Windows XP 操作系统中，才支持对缓冲区大小的查询。而对 Windows 98 来说，需要提前提供一个足够大的缓冲区——选好一个大缓冲区后，便不要另行变化。

利用 WSAIoctl，还可使用另一个 I/O 控制命令：SIO_CHK_QOS。该命令可用于查询如表 10.2 所示的 6 个值。调用这个命令时，lpvInBuffer 参数指向一个 DWORD(双字)，它会被设为 3 个标志之一。lpvOutBuffer 参数也应指向一个 DWORD；而且在返回之后，请求的值就会返回。最常用的标志是 ALLOWED_TO_SEND_DATA。假如发送者初始化了一条 PATH 消息，却没有收到任何指明 QOS 等级成功分配的 RESV 消息，便可使用该标志。若发送者在设置了 ALLOWED_TO_SEND_DATA 的前提下，使用 SIO_CHK_QOS 这个 I/O 控制命令，便会对网络进行查询，了解当前可用的“最大努力”通信是否足以发送在 QOS 结构中描述的那种数据(该结构已传递给一个发出 QOS 调用的函数)。欲知详情，请参阅第 9 章对该 I/O 控制命令的详细解释。

表 10.2　SIO_CHK_QOS 标志

SIO_CHK_QOS	标志说明	返回值
ALLOW_TO_SEND_DATA	指出数据的发送是立即开始，还是应该在应用程序接收到一条 RESV 消息后开始	BOOL
ABLE_TO_RECV_RSVP	指出发送者的接口是否支持 RSVP 功能	BOOL
LINE_RATE	返回接口的带宽容量	DWORD
LOCAL_TRAFFIC_CONTROL	返回传输控制(TC)是否已经安装，以及是否可以使用	BOOL
LOCAL_QOSABILITY	返回 QOS 是否可用	BOOL
END_TO_END_QOSABILITY	判断端到端的 QOS 是否可在网络上使用	BOOL

表 10.2 列出的返回值实际返回的是 1 或 0 值，分别对应“是”或“否”这两种值。假如系统当

前不能得到确切的值，表中最后 4 个选项可返回常数 INFO_NOT_AVAILABLE(信息不可用)。

10.3 终止 QOS

在前一节内，我们探讨了如何在一个套接字上调用 QOS。接下来，我们打算对 QOS 服务的中止进行讨论。下列每个事件都会造成 RSVP 的终止，同时停止为一个套接字处理 TC。

- 用 closesocket 函数关闭一个套接字。
- 用 shutdown 函数关闭一个套接字。
- 用一个空的通信对方地址调用 WSAConnect。
- 使用 SERVICETYPE_NOTRAFFIC 或者 SERVICE_TYPE_BESTEFFORT 服务类型，调用 WSAIoctl 和 SIO_SET_QOS。

除上述列表的第 2 项事件之外，其他各个事件都是很容易理解的。对于 shutdown 函数来说，请注意它既可能意味着数据发送的停止，也可能意味着接收的停止。发生两种情况的任意一个，都会造成在与之对应的那个方向上的数据流停止。换言之，假如用 SD_SEND 调用 shutdown，那么对于正在接收的数据，QOS 仍然是有效的。

10.3.1 提供程序特有的对象

本小节打算讲述那些提供程序特有的对象。它们会作为 QOS 结构的 ProviderSpecific 字段的一部分进行传递。这些对象要么通过 FD_QOS 事件，将 QOS 信息返回至应用程序；要么可随其他 QOS 参数一道，传递给 WSAIoctl，同时设定 SIO_SET_QOS 选项，以便对 QOS 的行为作进一步的确定。

在提供程序特有的对象中，每个都包含了一个 QOS_OBJECT_HDR 结构。该结构是这种对象的第 1 个成员。这个结构指明了提供程序特有对象的类型。因为这些提供程序对象通常会在经过了对 SIO_GET_QOS 的一次调用之后，在 QOS 结构中返回，所以该结构是至关重要的。使用 QOS_OBJECT_HDR，应用程序可以识别出每个对象，同时对其签名进行解码。对象头的定义如下：

```
typedef struct
{
   ULONG ObjectType;
   ULONG ObjectLength;
}QOS_OBJECT_HDR,*LPQOS_OBJECT_HDR;
```

其中，ObjectType 字段指出预设的提供程序特有对象的类型是什么。而 ObjectLength 字段指出整个对象的长度是多少。在这个长度中，同时包括了对象头，以及提供程序特有对象本身。表 10.3 对可选的对象类型标志进行了总结。

表 10 3　对象类型

提供程序对象	对象结构
QOS_OBJECT_SD_MODE	QOS_SD_MODE
QOS_OBJECT_SHAPING_RATE	QOS_SHAPING_RATE
QOS_OBJECT_DESTADDR	QOS_DESTADDR
RSVP_OBJECT_STATUS_INFO	RSVP_STATUS_INFO
RSVP_OBJECT_RESERVE_INFO	RSVP_RESERVE_INFO
RSVP_OBJECT_ADSPEC	RSVP_ADSPEC
RSVP_OBJECT_POLICY_INFO	RSVP_POLICY_INFO
QOS_OBJECT_END_OF_LIST	无，没有对象

10.3.1.1　QOS 封装丢弃模式

QOS 对象定义了 TC 的“包封装器”(Packet Shaper)如何处理一个指定流的数据。假如一个数据流与 FLOWSPEC 中指定的参数不符，那么在处理这个流时，通常就需要用到该属性。换言之，假如应用程序采用比发送 FLOWSPEC 的 TokenRate 字段设置的更快的速度来发送数据，我们便认为两者不相符。这个对象定义了本地系统应该如何应付这种情况。QOS_SD_MODE 结构的定义如下：

```
typedef struct _QOS_SD_MODE
{
   QOS_OBJECT_HDR ObjectHdr,
   ULONG          ShapeDiscardMode;
}QOS_SD_MODE,*LPQOS_SD_MODE;
```

其中，ShapeDiscardMode 字段的值可从表 10 4 中选择。

表 10 4　QOS 封装丢弃模式标志

标　志	说　明
TC_NONCONF_BORROW	为高优先等级的数据流提供了服务之后，用数据流接收剩余的资源。这种类型的流既不是“封装器”(Shaper)，也不是“排序器”(Sequencer)。假如为 TokenRate 指定了一个值，那么数据包可能出现不一致的情况，可能会降级到低于“最大努力”优先级

续表

标 志	说 明
TC_NONCONF_BORROW_PLUS	和 TC_NONCONF_BORROW 类似。但数据包在封装器中不会被标记为不相符
TC_NONCONF_SHAPE 必须为 TokenRate	指定一个值。不相符的数据包会保持在“包封装器”(Packet Shaper)内，直至相符为止
TC_NONCONF_DISCARD	必须为 TokenRate 指定一个值。不一致的数据包会被丢弃

大家或许会觉得奇怪，为何需要使用 TC_NONCONF_DISCARD 模式，在数据还没有通过线缆发送出去之前，便将其丢弃呢？事实上，在传送声音或影像数据时，有时便要采取这种行动。大多数情况下，在设置 FLOWSPEC 结构的时候，都会让一个数据包的大小正好等于一个影像帧的大小，或令其相当于一个小的声音片段。假如出于某个方面的原因，数据包发生了不相符的情况，那么对应用程序来说，采取哪种对策更好一些呢？是稍等一下，直至两者相符为止(就像 TC_NONCONF_SHAPE 的情况)呢？还是完全丢弃当前数据包，然后转移到下一个包的传递呢？通常，对那些对“实时性”要求较为严格的应用(例如影音文件的传输)，一种更好的做法是丢弃当前帧，立即转移到下一个帧。

10.3.1.2 QOS 目标地址

QOS_DESTADDR 结构用于为一个无连接的发送套接字指定目标地址，同时不必使用 WSAConnect 调用。此时，除非确切知道了无连接套接字的目标地址，否则不会有 RSVP PATH 或者 RESV 消息发送出去。可用 I/O 控制命令 SIO_SET_QOS 对目标地址进行设置。该结构的定义如下：

```
typedef struct _QOS_DESTADDR
{
   QOS_OBJECT_HDR ObjectHdr;
   const struct sockaddr *SocketAddress;
   ULONG          SocketAddressLength;
}QOS_DESTADDR,*LPQOS_DESTADDR;
```

其中，SocketAddress 字段指定的是一个 SOCKADDR 结构，它为给定的协议定义了端点地址。SocketAddressLength 则指定了 SOCKADDR 结构的长度。

10.3.1.3 RSVP 状态信息

RSVP 状态信息对象用于返回 RSVP 特有的一些错误和状态信息。该结构的定义如下：

```
typedef struct _RSVP_STATUS_INFO {
   QOS_OBJECT_HDR  ObjectHdr;
   ULONG           StatusCode;
   ULONG           ExtendedStatus1;
   ULONG           ExtendedStatus2;
}RSVP_STATUS_INFO,*LPRSVP_STATUS_INFO;
```

其中，StatusCode 字段对应于返回的 RSVP 消息。表 10.5 总结了可能用到的代码。另两个字段——ExtendedStatus1 和 ExtendedStatus2 字段则为提供程序特有的信息而保留。

表 10.5　RSVP 状态信息代码

标　志	含　义
WSA_QOS_RECEIVERS	至少有一条 RESV 消息抵达
WSA_QOS_SENDERS	至少有一条 PATH 消息抵达
WSA_QOS_NO_RECEIVERS	没有接收者
WSA_QOS_NO_SENDERS	没有发送者
WSA_QOS_REQUEST_CONFIRMED	预约已被确认
WSA_QOS_ADMISSION_FAILURE	由于缺乏资源，导致请求失败
WSA_QOS_POLICY_FAILURE	由于管理方面的原因，或者由于身份资料的错误，造成请求被拒绝
WSA_QOS_BAD_STYLE	未知的或冲突的样式
WSA_QOS_BAD_OBJECTRSVP_FILTERSPEC	结构的某个部分或与具体提供程序相关的缓冲区存在问题
WSA_QOS_TRAFFIC_CTRL_ERRORFLOWSPEC	结构的某个部分存在问题
WSA_QOS_GENERIC_ERROR	常规错误
ERROR_IO_PENDING	重复操作被取消

通常情况下，在收到一条 RSVP 消息后，应用程序需要接收一个 FD_QOS 事件，并调用 SIO_GET_QOS，以取得一个 QOS 结构，其中包含了一个 RSVP_STATUS_INFO 对象。例如，对于支持 QOS 的、以 UDP 为基础的接收者来说，会生成包含了一条 WSA_QOS_SENDERS 消息的 FD_QOS 事件，指出有人请求 QOS 服务将数据传给接收者。

10.3.1.4　RSVP 预约信息

RSVP 预约信息对象用于保存与 RSVP 有关的信息，以便通过 Winsock 2 QOS API 函数以及与特定提供程序对应的缓冲区，对双方之间的交互进行调整。RSVP_RESERVE_INFO 对象会覆盖默认的预约样式，并由一个 QOS 接收者使用。该对象的定义如下：

```
typedef struct _RSVP_RESERVE_INFO
```

```
{
   QOS_OBJECT_HDR     ObjectHdr;
   ULONG               Style,
   ULONG               ConfirmRequest;
   LPRSVP_POLICY_INFO  PolicyElementList;
   ULONG               NumFlowDesc;
   LPFLOWDESCRIPTOR    FlowDescList;
}RSVP_RESERVE_INFO,*LPRSVP_RESERVE_INFO;
```

其中，Style 字段指定了需要应用于该接收者的筛选器类型。在表 10 6 中，我们总结了目前可选的一些筛选器类型，以及不同类的接收者使用的默认筛选器类型。

表 10 6 默认筛选器样式

筛选器样式	默认用户
固定筛选器	单播接收者，已连接的 UDP 接收者
通配符	多播接收者，未连接的 UDP 接收者
共享显式	无

注意，每类筛选器稍后都会详加论述。假如 ConfirmRequest 字段不为零值，那么一旦接收应用程序收到 RESV 请求之后，就会发出通知。NumPolicyElements 与 PolicyElementList 字段是相互联系的。其中包含了 RSVP_POLICY 对象的数量，那些对象保存于 PolicyElementList 字段中。本章稍后还会对 RSVP_POLICY 进行定义。接下来，且让我们看看各种不同的筛选器样式，以及各自的特征。

10.3.1.5 RSVP_DEFAULT_STYLE

这个标志指明 QOS 服务提供程序使用默认样式。表 10 6 总结了各种不同接收者采用的默认样式。单播接收者使用固定筛选器，而通配符用于多播接收者。调用 WSAConnect 的 UDP 接收者亦可使用固定筛选器。

10.3.1.6 RSVP_FIXED_FILTER_STYLE

通常，这种样式会在接收者以及单独的一个数据源之间，建立起一个提供了 QOS 保障的数据流。对于单播接收者以及建立连接的 UDP 接收者来说，它们采用的便是这种样式：NumFlowDesc 设为 1，而 FlowDescList 包含了发送者的地址。但是，也完全可以设置采用了多个固定筛选器的样式，允许一个接收者从多个明确指定的数据源处，预约各自独立的数据流。举个例子来说，假如接收者想从 3 个发送者那里接收数据，并需要每一个都得保证 20Kbps 的带宽。此时，就需要考虑采用多固定筛选器样式。在这个例子中，NumFlowDesc 应设为 3，而 FlowDescList 应包含 3 个地址，每个 FLOWSPEC 一个。另外，也可为每个发送者分配不同的 QOS 等级；它们并非一定要完全相同。要注意的是，单播接收者和建立连接的 UDP 接收者不可使用多个固定的筛选器。图 10.1 展示了 FLOWDESCRIIPTOR 和 RSVP_FILTERSPEC 这两个结构间的关系。

10.3.1.7　RSVP_WILDCARD_STYLE

多播接收者和未连接的 UDP 接收者采用的是通配符样式。要想为 TCP 连接或者已经连接的 UDP 接收者使用这一样式，请务必将 NumFlowDesc 设为 0，并将 FlowDescList 设为 NULL。对于未建好连接的 UDP 接收者以及多播应用来说，因为发送者的地址当时是未知的，所以这是它们的默认筛选器样式。

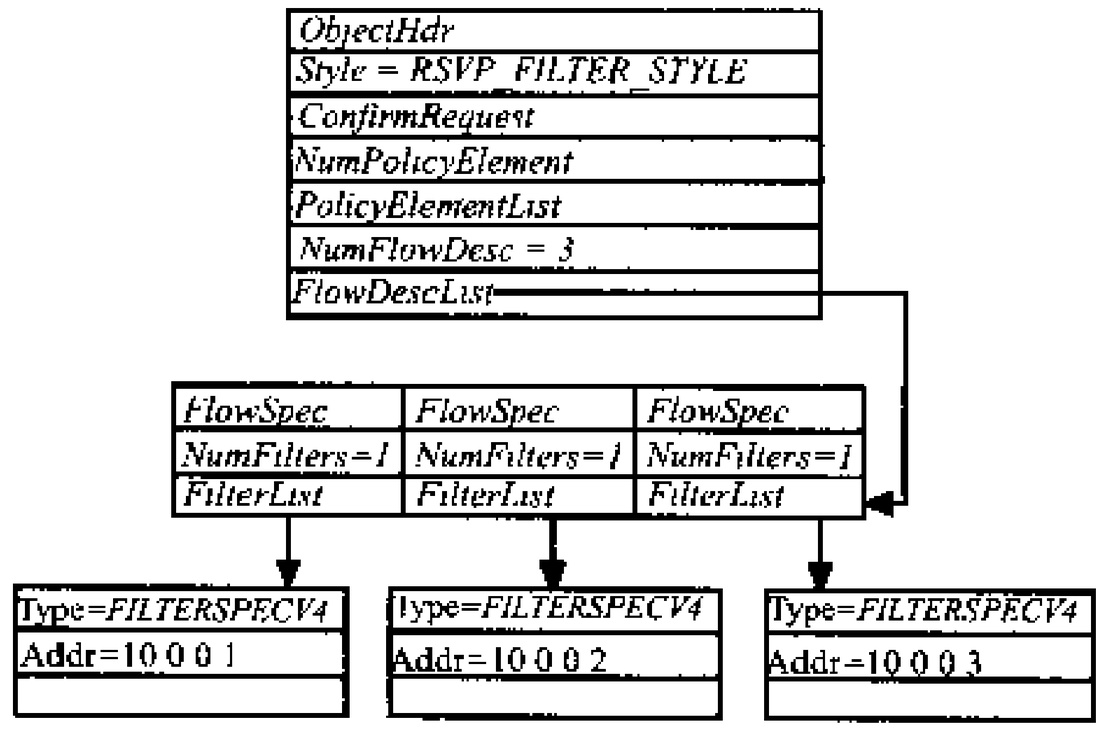


图 10 1　多固定筛选器样式

10.3.1.8　RSVP_SHARED_EXPLICIT_STYLE

在某种程度上，该样式类似于多固定筛选器样式，只是两者的网络资源有所区别。对前者来说，不再为每个发送者都分配网络资源。相反，网络资源需要在所有发送者之中共享。在这种情况下，NumFlowDesc 为 1，而 FlowDescList 包含了由发送者地址构成的一个列表。图 10.2 对这一样式进行了阐释。

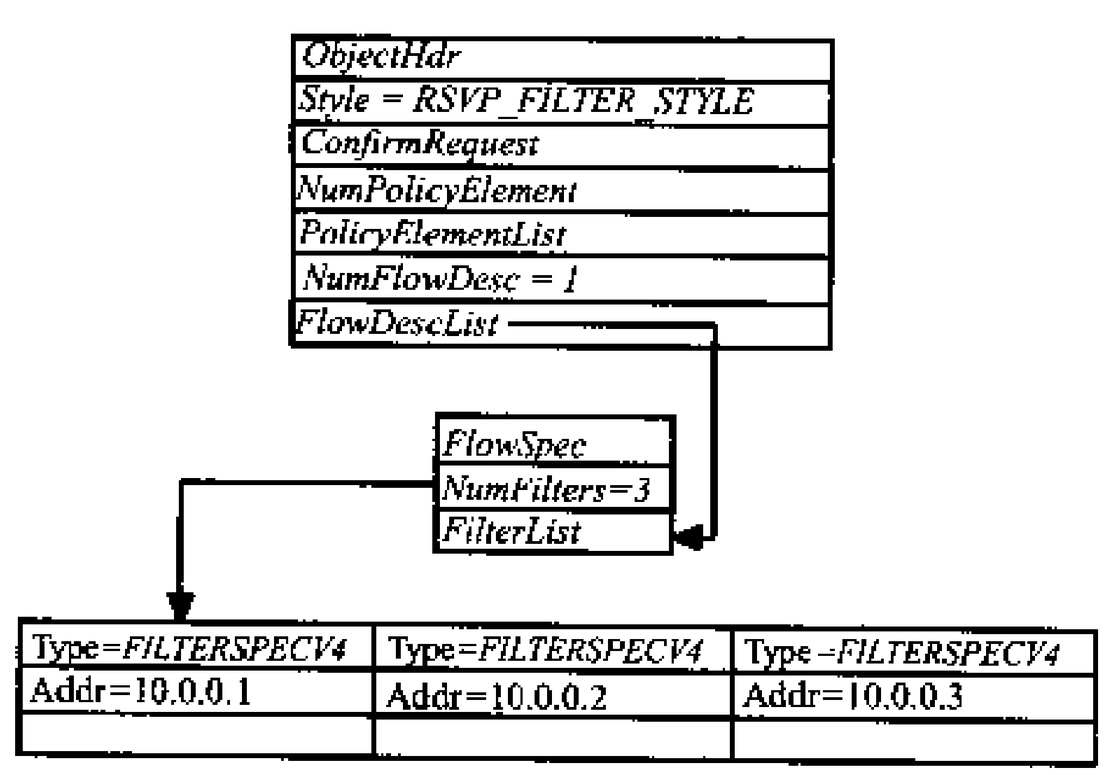


图 10.2　共享显式样式

在讨论 RSVP 样式时，已讲解了最后两个字段：NumFlowDesc 和 FlowDescList。至于具体如何使

用这两个字段，要取决于样式本身。其中，NumFlowDesc 定义了 FLOWDESCRIPTOR 结构在 FlowDescList 字段中的数量。该结构的定义如下：

```
typedef struct _FLOWDESCRIPTOR
{
   FLOWSPEC          FlowSpec;
   ULONG             NumFilters;
   LPRSVP_FILTERSPEC FilterList;
}FLOWDESCRIPTOR,*LPFLOWDESCRIPTOR;
```

这个对象用于定义由 FlowSpec 给定的每个 FLOWSPEC 的筛选器类型。同样地，在 NumFilters 字段中，包含了 FilterList 数组中存在的 RSVP_FILTERSPEC 对象的数量。RSVP_FILTERSPEC 对象的定义如下：

```
typedef struct _RSVP_FILTERSPEC {
   FilterType Type;
   union {
      RSVP_FILTERSPEC_V4       FilterSpecV4;
      RSVP_FILTERSPEC_V6       FilterSpecV6;
      RSVP_FILTERSPEC_V6_FLOW FilterSpecV6Flow;
      RSVP_FILTERSPEC_V4_GPI   FilterSpecV4Gpi;
      RSVP_FILTERSPEC_V6_GPI   FilterSpecV6Gpi;
      };
}RSVP_FILTERSPEC,*LPRSVP_FILTERSPEC;
```

其中，第 1 个字段 Type 是对下述值的简单列举：

```
typedef enum {
   FILTERSPECV4 = 1,
   FILTERSPECV6,
   FILTERSPECV6_FLOW,
   FILTERSPECV4_GPI,
   FILTERSPECV6_GPI,
   FILTERSPEC_END
}FilterType;
```

列举指定了共用体中存在的对象。每个筛选器的规格定义如下：

```
typedef struct _RSVP_FILTERSPEC_V4 {
   IN_ADDR_IPV4  Address;
   USHORT        Unused;
   USHORT        Port;
}RSVP_FILTERSPEC_V4,*LPRSVP_FILTERSPEC_V4;

typedef struct _RSVP_FILTERSPEC_V6 {
   IN_ADDR_IPV6  Address;
   USHORT        UnUsed;
```

```
   USHORT        Port;
}RSVP_FILTERSPEC_V6,*LPRSVP_FILTERSPEC_V6;

typedef struct _RSVP_FILTERSPEC_V6_FLOW {
   IN_ADDR_IPV6  Address;
   UCHAR         UnUsed;
   UCHAR         FlowLabel[3];
}RSVP_FILTERSPEC_V6_FLOW,*LPRSVP_FILTERSPEC_V6_FLOW;

typedef struct _RSVP_FILTERSPEC_V4_GPI {
   IN_ADDR_IPV4  Address;
   ULONG         GeneralPortId;
}RSVP_FILTERSPEC_V4_GPI,*LPRSVP_FILTERSPEC_V4_GPI;
```

```
typedef struct _RSVP_FILTERSPEC_V6_GPI {
   IN_ADDR_IPV6  Address;
   ULONG         GeneralPortId;
}RSVP_FILTERSPEC_V6_GPI,*LPRSVP_FILTERSPEC_V6_GPI;
```

10.3.1.9　RSVP 广告规范

RSVP_ADSPEC 对象定义了 RSVP“广告规范”(Adspec)中包含的信息。在该 RSVP 对象中，通常需要指出有哪些类型的服务可用(受控制负载，或者质量担保)，PATH 消息是否会碰到一个非 RSVP 跳跃，以及路径上最小的 MTU 是多大。下面是该结构的定义：

```
typedef struct _RSVP_ADSPEC
{
   QOS_OBJECT_HDR       ObjectHdr;
   AD_GENERAL_PARAMS  GeneralParams;
   ULONG                NumberOfServices;
   CONTROL_SERVICE      Services[1];
}RSVP_ADSPEC,*LPRSVP_ADSPEC;
```

我们感兴趣的第 1 个字段是 GeneralParams，它是类型为 AD_GENERAL_PARAMS 的一个结构。该结构用来定义一些常规的特征参数。这个结构的定义如下：

```
typedef struct _AD_GENERAL_PARAMS
{
   ULONG IntServAwareHopCount;
   ULONG PathBandwidthEstimate;
   ULONG MinimumLatency;
   ULONG PathMTU;
   ULONG Flags;
}AD_GENERAL_PARAMS,*LPAD_GENERAL_PARAMS;
```

其中，IntServAwareHopCount 字段指定的是符合 IntServ(Integrated Service，集成服务)要求的跳

跃数。PathBandwidthEstimate 字段指定的是从发送者到接收者，至少能使用多大的带宽。MinimumLatency 字段指定的是数据包通过路由器转发时，最小延迟时间的一个总和，以毫秒为单位。PathMTU 字段指定的是端到端不存在任何分段的最大传输单元。Flags 字段已经不再使用了。

10.3.1.10 RSVP 策略信息

要讨论的最后一个提供程序对象是 RSVP 策略信息。该对象的含义非常模糊——其中包含了来自 RSVP 的、尚未定义的任意数量的策略元素。该结构的定义如下：

```
typedef struct _RSVP_POLICY_INFO {
   QOS_OBJECT_HDR ObjectHdr;
   ULONG          NumPolicyElement;
   RSVP_POLICY    PolicyElement[1];
}RSVP_POLICY_INFO,*LPRSVP_POLICY_INFO;
```

其中，NumPolicyElement 字段给定 PolicyElement 数组中存在的 RSVP_POLICY 结构数量。下面是该结构的定义：

```
typedef struct _RSVP_POLICY {
   USHORT    Len;
   USHORT    Type;
   UCHAR     Info[4];
}RSVP_POLICY,*LPRSVP_POLICY;
```

RSVP_POLICY 结构是由 RSVP 传送的数据，对应于策略部件，与我们目前的需要关系不大。

10.4 QOS 编程

本节叙述的编程技术主要应用于 Windows 98、Windows Me 和 Windows 2000 平台。前面已经提到，Windows XP 不支持完整的 IP QOS 服务提供程序。因此，Windows XP 平台将不再支持 RSVP 传信和接纳控制层，但支持通信量控制 IP 数据包调度层，建立 FLOWSPEC 的技术也仍然适用，不过关于 QOS 通知和 RSVP 传信的信息不再适用。

QOS 的核心在于 RSVP 会话的初始化。在发送出 RSVP PATH 以及 RESV 消息，而且得到了处理后，进程才会成功预约到带宽。对应用程序来说，知道 RSVP 消息在什么时候建立是至关重要的。对发送者来说，在生成 PATH 消息之前，首先必须知道 3 个参数：

- 发送 FLOWSPEC 成员
- 源 IP 地址及端口
- 目标 IP 地址、端口及协议

只要调用了一个支持 QOS 的函数，那么 FLOWSPEC 成员便是已知的了。这些函数包括 WSAConnect、WSAJoinLeaf、WSAIoctl(设置 SIO_SET_QOS 选项)等等。源 IP 地址和端口号暂时无法知道，直到完成了套接字的本地绑定才清楚。这种绑定既可是隐式进行的，例如通过连接来绑定；也可是显示进行的。最后，应用程序需要知道数据的目的地。在完成了一个连接调用之后，或在建立了无连接的 UDP 连接之后，或者用 SIO_SET_QOS 这个 I/O 控制命令在与具体提供程序有关的数据中设置了 QOS_DESTADDR 之后，才能知道数据的目标地址是什么。类似地，要想生成一条 RSVPRESV 消息，事先必须知道 3 件事情：

- 接收 FLOWSPEC 成员
- 每个发送者的地址和端口
- 接收套接字的本地地址和端口

接收 FLOWSPEC 成员可通过任何支持 QOS 的 Winsock 函数获得。每个发送者的地址取决于筛选器的样式，后者可用提供程序特有的结构 RSVP_RESERVE_INFO 来人工设定(前面已经具体讲过了)。也可从一条 PATH 消息中取得这种信息。当然，取决于具体的套接字类型，有时为了生成 RESV 消息，并不一定非要事先收到一条 PATH 消息才能取得发送者的地址。这样的一个典型例子便是多播通信中采用的通配符筛选器。发出去的 RESV 消息会应用于一个会话中的所有发送者。对于单播和 UDP 接收者来说，本地地址与端口号不难取得。但多播接收者却不容易获取。在多播接收者的情况下，本地地址与端口分别是多播地址以及它对应的端口号。

下一节首先要介绍各种不同的套接字类型，以及它们如何与 QOS 服务提供程序及 RSVP 消息进行交互。然后会解释 QOS 服务提供程序如何向应用程序发出通知，告知它们发生了某种特定的事件。正确理解了这些概念之后，再来编写一个支持 QOS 的应用程序，便能起到事半功倍效果。要想编写出这样的应用程序，其实只需要知道如何取得 QOS 的质量保证，知道这些保证何时能够兑现，以及它们何时和怎样发生变化。

10.4.1　RSVP 和套接字类型

到目前为止，大家已对 PATH 及 RESVRSVP 消息如何生成有了一定的概念。后续的小节将讲解各类不同的套接字——UDP、TCP 以及多播 UDP。在这个过程中，大家会学习到它们如何与 QOS 服务提供程序沟通，以生成 PATH 和 RESV 消息。

10.4.1.1　单播 UDP

由于可选择使用已建立和未建立连接的两种 UDP 套接字，所以在单播 UDP 套接字上设置 QOS 时，有大量选项可供选择。对 UDP 发送者而言，发送 FLOWSPEC 是从 QOS 的某个调用函数中获得的。本地地址和端口号既可通过一个显式绑定调用取得，也可通过由 WSAConnect 完成的一个隐式绑定取得。最后要知道的是接收应用程序的地址及端口，这要么是在 WSAConnect 中指定的，要么是通过 QOS_DESTADDR 提供程序特有的结构来传递的(指定 SIO_SET_QOS 选项)。要注意的是，假如用

SIO_SET_QOS 来设置 QOS，那么必须事先绑定好套接字。

对 UDP 接收者来说，可调用 WSAConnect，将接收应用程序限制成单独的发送者。除此以外，使用 SIO_SET_QOS 这个 I/O 控制命令，应用程序可指定一个 QOS_DESTADDR 结构。也可以在调用 SIO_SET_QOS 的时候，不提供任何类型的目标地址。在这种情况下，会生成一条 RESV 消息，同时采用通配符筛选器样式。事实上，无论通过 WSAConnect 指定目标地址，还是通过 QOS_DESTADDR 结构来指定，都只有在希望应用程序只从一个采用固定筛选器样式的发送者那里接收数据的时候，才能采取这样的做法。

UDP 接收者实际可同时调用 WSAConnect 及 SIO_SET_QOS 这两个 I/O 控制命令，并可按任意顺序调用。假定在 WSAConnect 之前调用 SIO_SET_QOS，那么首先会创建一条采用通配符筛选器的 RESV 消息。一旦连接调用完毕，前一次 RESV 会话便会关闭，并采用固定筛选器样式，建立一次新的会话。另外，若在 WSAConnect 之后调用 SIO_SET_QOS，那么采用固定筛选器的 RESV 消息就不会中断 RSVP 会话，并生成一个通配筛选器样式。相反，它只是简单地更新与现有 RSVP 会话关联在一起的 QOS 参数。

10.4.1.2 单播 TCP

TCP 会话存在着两种可能性。首先，发送者可能是客户机，它建立与服务器的连接，并发送数据。第 2 种可能性则是与客户机连接的那个服务器才是发送者。在发送者是客户机的情况下，QOS 参数可直接在 WSAConnect 调用中指定，这会导致 PATH 消息的发出。调用连接之前，也可以调用 I/O 控制命令 SIO_SET_QOS，但除非其中的一个连接调用知道了目标地址，否则不会生成任何 PATH 消息。

假如发送者是服务器，那么服务器需要调用 WSAAccept 函数来接受客户机的连接请求。然而，这个函数并未提供，可供我们在接受的套接字上设置 QOS 的方式。假如通过 SIO_SET_QOS，在发出对 WSAAccept 的调用之间便已设好了 QOS，那么以后负责接受的套接字会继承监听套接字上设定的 QOS 等级。要注意的是，假如发送者在 WSAAccept 中使用条件函数，那么函数应同时传递在建立连接的那个客户机上设置的 QOS 值。但目前遇到的并不是这种情况。QOS 服务提供程序传递的可能只是一些垃圾——无论 Windows 98、Windows Me 还是 Windows 2000，都会产生这样的行为。例外在于，假如在 Windows 98 和 Windows Me 中，lpSQOS 参数不为 NULL，那么必须在条件函数内部，通过 I/O 控制命令 SIO_SET_QOS 设置某种 QOS 值。否则，即使返回了 CF_ACCEPT，WSAAccept 调用也会失败。在被接受之后，QOS 也可在客户机套接字上进行设置。

现在，来看看负责接收的 TCP 应用程序。第 1 种情况是调用 WSAConnect，同时设置一个接收 FLOWSPEC。在这种情况下，QOS 服务提供程序会创建一个 RESV(预约)请求。假如未将 QOS 参数提供给 WSAConnect，那么 SIO_SET_QOS 这个 I/O 控制命令可在以后的某个时间设置(结果便产生了一条 RESV 消息)。最后一种组合是服务器作为接收者使用，这和发送的情况差不多。发出一个 WSAAccept 调用之前，QOS 可在监听套接字上设置。此时，客户机套接字会继承相同的 QOS 等级。此外，也可在条件函数中设置 QOS，或在套接字已被接受之后设置。无论在哪种情况下，一旦有一条

PATH 消息抵达，QOS 服务提供程序都会生成一条 RESV 消息。

10.4.1.3　多播

多播发送者的行为和 UDP 发送者的行为差不多，惟一的例外在于，现在需要调用 WSAJoinLeaf，从而成为多播组的一名成员，而不是调用 WSAConnect，并指定一个目标地址。可用 WSAJoinLeaf 对 QOS 进行设置，或通过一个 SIO_SET_QOS 调用，对其进行单独设置。多播会话地址用于构成 RSVP 会话对象，并将该对象包含在 RSVP PATH 消息之中。

在多播接收者的情况下，除非已通过 WSAJoinLeaf 函数指定了多播地址，否则是不会生成 RESV 消息的。由于多播接收者没有指定一个对方地址，所以 QOS 提供程序在生成 RESV 消息的时候，会采用通配符筛选器样式。QOS 服务提供程序并不禁止一个套接字同时加入多个多播组。在这种情况下，服务提供程序会将 RESV 消息发给所有组，只要它们有一条匹配的 PATH 消息。为每个 WSAJoinLeaf 提供的 QOS 参数会使用在每条 RESV 消息中，但假如加入多个组之后，在套接字上调用 SIO_SET_QOS，那么新的 QOS 参数会应用于已经加入的所有多播组。

若发送者将数据发给一个多播组，那么只有在发送者已经加入多播组的前提下，才会使相应的 QOS 参数应用于那些数据。假如加入了一个多播组，但在使用 sendto 或 WSASendTo 的时候，却将另外某个多播组设为目的地，QOS 便不会应用于那些数据。除此以外，假如一个套接字加入了一个多播组，并指定了一个特定的方向(例如，在 WSAJoinLeaf 的 dwFlags 参数中使用 JL_SENDER_ONLY 或 JL_RECEIVER_ONLY)，那么 QOS 也会相应地产生作用。若某个套接字仅被设定为接收者，那么在其发送数据的时候，就不会享受到 QOS 提供的任何服务。

10.4.2　QOS 通知

迄今为止，本书已介绍了了如何为 TCP、UDP 以及多播 UDP 套接字调用 QOS。根据自己到底是接收还是发送数据，会有对应的 RSVP 事件产生。然而，这些 RSVP 消息的完成并非与调用它们的 API 调用严格地关联在一起。也就是说，假如为一个 TCP 接收套接字发出一个 WSAConnect 调用，那么会生成一条 RESV 消息，但 RESV 消息与实际的 API 调用却是无关的。这是由于在调用返回时，并不能保证预约已获得确认，也无法保证网络资源已正确分配。正是考虑到这方面的原因，这里增加了一个新的异步事件：FD_QOS，并将它投递给套接字。在典型情况下，发生下述事件时，便会投递一个 FD_QOS 事件通知：

- 通知应用程序的 QOS 请求被接受或被拒绝
- 网络提供的 QOS 发生明显改变(与以前协商的值相反)
- 状态表明，QOS 通信对方已经准备好发送或接收一个特定流的数据，或尚未准备好

10.4.2.1　注册 FD_QOS 通知

为利用这些通知带来的好处，应用程序首先必须注册，指明在发生 FD_QOS 事件的时候，自己

想收到通知。有两个办法可完成应用程序的注册。首先，可使用 WSAEventSelect 或者 WSAAsyncSelect，同时在事件标志的按位“或”运算中，将 FD_QOS 这个标志包含进来。只有已发出了对某个 QOS 调用函数的调用，应用程序才有权接收 FD_QOS 事件。要注意的是，在某些情况下，应用程序可能想接收 FD_QOS 事件，同时不想在套接字上设置 QOS 等级。要达到这个目的，可设置一个 QOS 结构，在它的发送及接收 FLOWSPEC 成员中包含 QOS_NOT_SPECIFIED 或者 SERVICETYPE_NOTRAFFIC 标志。这样做惟一的缺点在于，针对自己希望接收事件通知的那个 QOS 方向，SERVICE_NO_QOS_SIGNALING 标志必须同 SERVICETYPE_NOTRAFFIC 标志执行“或”运算。

如果需要了解如何调用两个异步选择函数，请参考第 5 章。这一章对其进行了更深入、全面地讲解。事件触发后，假如立即使用 WSAEventSelect，那么请调用 WSAEnumNetworkEvents 函数，获取附加的状态码(如果有的话)。只需要在调用中传递一个套接字句柄、一个事件句柄以及一个 WSANETWORKEVENTS 对象，以便在提供的结构中返回并设置事件信息。

10.4.2.2 RSVP 通知

早先已经提到，有两个办法可接收到 QOS 通知消息。这种信息实际与本节主题(获取 QOS 事件的结果)密切相关。如果已进行了注册，并用 WSAAsyncSelect 或 WSAEventSelect 来接收 FD_QOS 通知，而且实际也接收到了一个 FD_QOS 事件通知，那么必须执行对 WSAIoctl 的一个调用，同时设置 SIO_GET_QOS 这个 I/O 控制选项，以便知道具体是什么原因触发了事件。实际上并不一定需要为 FD_QOS 事件注册，可以通过重叠 I/O，简单地调用 WSAIoctl，同时设置 SIO_GET_QOS 选项。这同时也要求指定一个完成例程。一旦 QOS 服务提供程序检测到 QOS 发生了改变，便会调用这个例程。进行回叫的时候，在输出缓冲区内，应该能找到一个 QOS 结构。

无论在哪种情况下，一旦 QOS 内发生了变化，应用程序便能相应地得到通知，前提是已为 FD_QOS 进行了注册，或者使用了重叠 I/O 以及 SIO_GET_QOS。如果选择的是为 FD_QOS 进行注册，那么在收到事件通知之后，通常还需要调用 WSAIoctl，同时使用 SIO_GET_QOS 这个 I/O 控制命令。对这两种方法来说，在返回的 QOS 结构中，都只包含了针对一个单独方向的 QOS 信息。对于无效方向上的 FLOWSPEC 结构来说，在其 ServiceType 字段中，会设置 SERVICETYPE_NOCHANGE(服务类型未改变)。除此以外，同时可能发生多个 QOS 事件。在这种情况下，应循环调用 WSAIoctl 和 SIO_GET_QOS，直到返回 SOCKET_ERROR，同时 WSAGetLastError 返回一个 WSAEWOULDBLOCK 值为止。调用 SIO_GET_QOS 时，要注意的最后一个问题是缓冲区的大小。触发了一个 FD_QOS 事件之后，可能返回与具体提供程序有关的对象。事实上，只要缓冲区足够大，经常返回的是一个 RSVP_STATUS_INFO 结构。请参考以前讲述 WSAIoctl 的内容，了解具体如何判断一个缓冲区的正确大小。

假如应用程序采用了某个异步事件函数，那么特别要注意的一个问题是，一旦发生 FD_QOS 事件，则无论如何都必须执行一个 SIO_GET_QOS 操作，以使重新允许 FD_QOS 通知行为。

现在，大家已经知道了如何接收 QOS 事件通知，以及如何获取新的 QOS 参数(那些事件的结果)，

但到底会发生何种类型的通知呢？对一个 QOS 事件来说，触发它的第 1 个、也是最明显的一个原因是某个指定数据流的 FLOWSPEC 参数发生了改变。举个例子来说，假定将一个套接字设置成提供“最大努力”的服务。那么，作为 QOS 的服务提供程序，它会向应用程序定时发出通知，指出网络当前的状态是什么。除此以外，假如设置一个“受控制的负载”(Controlled Load)，同时设置其他一系列参数，那么一旦发生预约，与令牌大小及令牌速率对应的 QOS 参数可能会与我们请求的值稍有区别。在应用程序中，一旦收到一个 QOS 通知，便应立即将返回的 FLOWSPEC 同最初请求的 FLOWSPEC 进行对比，确定应用程序是否能够继续运行下去。还要记住的是，在支持 QOS 的一个套接字发挥作用期间，随时都可执行一个 SIO_SET_QOS 命令，更改任何参数。这也会为与当前 RSVP 会话关联在一起的一个或多个通信方生成一个 QOS 事件通知(也许不止一个)。作为一个健壮的应用程序，必须能够应付所有这些情况。

除更新 QOS 参数以外，QOS 事件通知还可以告诉我们发生的另外一些事情，例如出发送者或接收者发出的通知等等。在前文的表 10.5 中，已列出了所有可能的事件。有两个办法可获得这些状态代码。第 1 个办法是作为 RSVP_STATUS_INFO 对象的一部分来获取代码。发生了一个 QOS 事件，且已进行了对 SIO_GET_QOS 的一个调用之后，一个 RSVP_STATUS_INFO 对象便可能作为提供程序特有缓冲区的一部分而返回。第 2 个办法，如果用 WSAEventSelect 来注册事件，这些代码便可在自 WSAEnumNetworkEvents 返回的 WSANETWORKEVENTS 结构中返回。表 10.5 定义的代码可在 iErrorCode 数组中找到，该数组由 FD_QOS_BIT 进行索引。表中总结的头 5 个代码并非错误代码。通过这些代码，可知道与 QOS 连接状态有关的宝贵信息。表中列出的其他状态代码则全部属于 QOS 错误。虽然发生了这些错误，但它们并不能阻止数据的继续收发，它们的作用仅仅是指出在 QOS 会话中发生了一个错误。当然，在这种情况下发送的数据根本无法保证提供我们所要求的 QOS。

10.4.2.3　WSA_QOS_RECEIVERS 和 WSA_QOS_NO_RECEIVERS

进行单播通信的时候，一旦发送者开始运行，并接收到了第 1 条 RESV 消息，一个 WSA_QOS_RECEIVERS 便会上传给应用程序。假如接收者采取了任何操作来禁止 QOS，其结果便是一条 RESV 取消消息。发送者收到这条消息后，WSA_QOS_NO_RECEIVERS(无接收者)便会上传给应用程序。当然，在单播通信的情况下，许多接收者都只是简单地将套接字关闭，从而同时生成了 FD_CLOSE 以及 WSA_QOS_NO_RECEIVERS 事件。在大多数情况下，应用程序对此作出的响应也仅仅是将发送套接字关闭。

而在多播通信的情况下，只要接收者的数量发生了改变，而且不为零，那么作为发送应用程序，无论如何都会收到 WSA_QOS_RECEIVERS。换言之，每次有一个 QOS 接收者加入多播组时，以及每次有一个接收者脱离这个组时，只要组内至少还剩有一个接收者，多播发送者都会收到 WSA_QOS_RECEIVERS。

10.4.2.4　WSA_QOS_SENDERS 和 WSA_QOS_NO_SENDERS

发送者通知和接收者事件差不多，惟一的例外是它处理的是 PATH 消息的收到事件。对单播接收

者来说，启动之后，假若收到第 1 条 PATH 消息，便会生成一个 WSA_QOS_SENDERS；而 PATH 撤消消息会导致一条 WSA_QOS_NO_SENDERS 消息的生成。

类似地，一旦发送者的数量增加或减少，而且不为零，那么多播接收者就会收到一个 WSA_QOS_SENDERS 通知。一旦发送者的数量变为 0，WSA_QOS_NO_SENDERS 消息便会传递给应用程序。

10.4.2.5 WSA_QOS_REQUEST_CONFIRMED

这是最后一条状态消息。如果应用程序要求在确认了一个预约请求之后，便收到相应的通知，那么负责接收的 QOS 应用程序会收到这个消息。在 RSVP_STATUS_INFO 结构内，有一个名为 ConfirmRequest(确认请求)的字段。假如该字段的值不为零，便意味着在预约请求通过确认后，QOS 服务提供程序需要向应用程序发出一个通知。该对象是提供程序特有的一个选项，可随 QOS 结构传递给 SIO_SET_QOS 这个 I/O 控制命令。

10.4.3 QOS 模板

Winsock 提供了几个预先定义好的 QOS 结构，我们称其为模板，应用程序可按名称对这些模板进行查询。这些模板为某些常见的音频及视频编码 / 解码规范(如 G711 和 H263QCIF)定义了相应的 QOS 参数。查询模板的 WSAGetQOSByName 函数定义如下：

```
BOOL WSAGetQOSByName(
   SOCKET s,
   LPWSABUF lpQOSName,
LPQOS lpQOS
);
```

假如不知道已安装的模板的名称，首先可用该函数列举所有模板名。但要想真正成功，必须用 lpQOSName 提供一个足够大的缓冲区，并将它的第 1 个字符设为空字符，并为 lpQOS 传递一个空指针。如下述代码所示：

```
WSABUF    wbuf;
char          cbuf[1024];

cbuf[0] = '\0';
wbuf.buf = cbuf;
wbuf.len = 1024;
WSACetQOSByName(s,&wbuf,NULL);
```

返回之后，在字符缓冲内，会填入一个字符串数组，各字符串之间用一个空字符加以分隔，而且整个列表会用另一个空字符终止。换言之，最后一个字符串条目会有两个连续的空字符。根据这个缓冲区的内容，便能知道已安装的所有模板的名称，并可从中查询一个特定的模板。例如，下述代码可对 G711 模板进行查询：

```
QOS     qos;
WSABUF wbuf;

wbuf.buf = "G711";
wbuf.len = 4;
WSAGetQOSByName(s,&wbuf,&qos);
```

若请求的 QOS 模板不存在，那么查询会返回 FALSE，错误代码是 WSAEINVAL。假若成功，函数会返回 TRUE。

注意　补充材料中的示例 QOSTEMPLATE.CPP 阐述了如何列举已安装的 QOS 模板。

除此以外，还可以安装自己的 QOS 模板，使其他应用程序能根据名称对其进行查询。此时要用到两个函数：WSCInstallQOSTemplate 以及 WSCRemoveQOSTemplate。其中，第 1 个函数用于 QOS 模板的安装，而第 2 个用于删除。函数原型如下：

```
BOOL WSCInstallQOSTemplate(
   const LPGUID lpProviderId,
   LPWSABUF lpQOSName,
   LPQOS lpQOS
);

BOOL WSCRemoveQOSTemplate(
   const LPGUID lpProviderId,
   LPWSABUF lpQOSName
);
```

这两个函数的用法其实是一目了然的。要想安装一个模板，需要调用 WSCInstallQOSTemplate，同时指定 GUID、模板名以及 QOS 参数。GUID 是该模板一个独一无二的标识符，可由 UUIDGEN.EXE 这样的实用程序来生成。要想删除模板，只需调用 WSCRemoveQOSTemplate 并指定那个模板的名称，同时指定与安装过程相同的一个 GUID 就可以了。若成功，这两个函数都会返回 TRUE。

10.5　示　　例

本节将通过两个实际的例子，来展示 QOS 在 TCP 和 UDP 上的使用。第 1 个例子使用的是 TCP，将展示如何在 Windows 98、Windows Me 及 Windows 2000 平台上建立一个 FLOWSPEC，并对 RSVP 传信进行管理。第 2 个例子使用 UDP，主要为 Windows XP 而设计，只展示如何建立一个 FLOWSPEC，以便激活 IP 数据包调度器。这两个例子都要依赖于两个支持例程：PrintQos 以及 FindProtocolInfo，它们分别定义于 PRINTQOS.CPP 以及 PROVIDER.CPP 中。前一个例程用于打印 QOS 结构的内容，而后一个例程用于在提供程序编录中查找具有指定属性(如 QOS)的一种协议。

10.5.1 TCP

TCP 示例的的源码可在光盘上的一个名为 QOSTCP.CPP 的文件中找到。尽管该清单很长，但假如仔细观察，就会发现它并不是特别复杂。大多数代码都只不过是一些普通的 WSASelect 代码，这些内容已在第 5 章详细讲述过了。惟一的例外是在产生 FD_QOS 事件时做的事情。主函数只是完成一些普通的任务，解析出参数，建立起套接字，再调用 Server 或 Client 函数(具体取决于应用程序作为服务器还是客户机调用)。下面，首先来看看客户机连接的情况。

示例有一个命令行参数，告诉示例何时设置 QOS：连接之前、连接过程中、建立连接之后，还是在对方请求 QOS 进行 QOS 的本地设置之后。假如决定在连接建立之前设置 QOS(对客户机来说)，可将套接字绑定到一个随机端口，再调用 SIO_SET_QOS，同时设定一个 QOSFLOWSPEC。要注意的是，在调用 SIO_SET_QOS 之前，实际上并非真正需要进行这种绑定。这是由于，只有发出一个连接调用，才能知道对方的地址。而只有确切知道了对方的地址，才能建立起一个 RSVP 会话。

假如用户选择在连接过程中设置 QOS，那么示例代码会将 QOS 结构传递入 WSAConnect 调用。下述代码展示了如何随意地在连接之前或在连接过程中建立 QOS：

```
int iSetQos;
SOCKET s;
char szServerAddr[32];
...
const FLOWSPEC flowspec_notraffic = {QOS_NOT_SPECIFIED,
                                     QOS_NOT_SPECIFIED,
                                     QOS_NOT_SPECIFIED,
                                     QOS_NOT_SPECIFIED,
                                     QOS_NOT_SPECIFIED,
                                     SERVICETYPE_NOTRAFFIC,
                                     QOS_NOT_SPECIFIED,
                                     QOS_NOT_SPECIFIED};
const FLOWSPEC flowspec_g711 = {8500,
                                680,
                                17000,
                                QOS_NOT_SPECIFIED,
                                QOS_NOT_SPECIFIED,
                                SERVICETYPE_CONTROLLEDLOAD,
                                340,
                                340};

QOS    clientQos;    //QOS 客户机结构
QOS    *lpqos;
SOCKADDR_IN  server,
SOCKADDR_IN  local;
DWORD  dwBytes;
```

```
...
//设置客户机的 QOS flowspec

clientQos.SendingFlowspec = flowspec_g711;
clientQos.ReceivingFlowspec = flowspec_notraffic;
clientQos.ProviderSpecific.buf = NULL;
clientQos.ProviderSpecific.len = 0;

if (iSetQos == SET_QOS_BEFORE)
{
lpqos = NULL;

//绑定到本地接口并提供 flowspec
    local.sin_family = AF_INET;
    local.sin_port = htons(0);
    local.sin_addr.s_addr = htonl(INADDR_ANY);

    bind(s,(SOCKADDR *)&local,sizeof(local));

    WSAIoctl(s,SIO_SET_QOS,&clientQos,sizeof(clientQos),
            NULL,0,&dwBytes,NULL,NULL);
}
else if (iSetQos == SET_QOS_DURING)
{
    //在连接过程中使用 QOS 结构
    lpqos = &clientQos;
}
else if (iSetQos == SET_QOS_EVENT)
{
    lpqos = NULL;
    //当通信对方传信后，稍后便设置 QOS

    clientQos.SendingFlowspec.ServiceType |= SERVICE_NO_QOS_SIGNALING;
    clientQos.ReceivingFlowspec.ServiceType |= SERVICE_NO_QOS_SIGNALING;

WSAIoctl(s,SIO_SET_QOS,&clientQos,sizeof(clientQos),NULL,0,
       &dwBytes,NULL,NULL);
}

server.sin_family = AF_INET;
server.sin_port = htons(5150);
server.sin_addr.s_addr = inet_addr(szServerAddr);
```

```
printf("Connecting to:%s \n",inet_ntoa(server.sin_addr));

WSAConnect(s,(SOCKADDR *)&server,sizeof(server),NULL,NULL,
        lpqos,NULL);
...
```

这个 WSAConnect 调用会建立一个 RSVP 会话，并将客户机同指定的服务器连接起来。不然，用户需要指示这个例子应该让对方设置 QOS，而且没有 QOS 结构会传递给 WSAConnect。相反，代码会接受进行发送的 QOS 结构，用 SERVICE_NO_QOS_SIGNALING 标志同 FLOWSPEC 结构中的 ServiceType 字段进行或(OR)运算，并用 SIO_SET_QOS 命令调用 WSAIoctl。这样便可告诉 QOS 服务提供程序不要调用通信控制，而是依然等候 RSVP 消息。

设置好 QOS 后，客户机希望收到通知的事件(包括 FD_QOS)将被注册。注意，在应用程序要求接收 FD_QOS 前，QOS 事先必须在套接字上设置好。一旦收到 FD_QOS，客户机便会在 WSAWaitForMultipleEvents 的一个循环中等待。假如设定的某个事件被触发，循环便会中断。发生了一个事件之后，事件便会在 WSAEnumNetworkEvents 中被列举出来，同时附带已有的任何错误。下述代码片段展示了如何通过 WSAEventSelect 来处理 FD_QOS 事件：

```
wbuf.buf = databuf;
wbuf.len = DATA_BUFFER_SZ;

memset(databuf,'#',DATA_BUFFER_SZ);
databuf[DATA_BUFFER_SZ-1]= 0;

while (1)
{
    ret = WSAWaitForMultipleEvents(1,&hEvent,FALSE,
         WSA_INFINITE,FALSE);
    if (ret == WSA_WAIT_FAILED)
    {
       printf("WSAWaitForMulipleEvents()failed:%d \n",
            WSAGetLastError());
       return;
    }

    WSAEnumNetworkEvents(s,hEvent,&ne);

    if (ne.lNetworkEvents &FD_READ)
    {
       //读发生的通知
    }
    if (ne.lNetworkEvents &FD_WRITE)
    {
       if (ne.iErrorCode[FD_WRITE_BIT])
```

```
    printf("FD_WRITE error:%d \n",ne.iErrorCode[FD_WRITE_BIT]);
else
    printf("FD_WRITE \n");

if (!bWaitToSend)
{
    wbuf.buf = databuf;
    wbuf.len = DATA_BUFFER_SZ;
    //
    //如果网络不支持这个带宽，则不发送
    //
    if (!AbleToSend(s))
    {
        printf("Network is unable to provide "
            "sufficient best effort bandwidth \n");
        printf("before the reservation "
        "request is approved \n");
    }

    WSASend(s,&wbuf,1,&dwBytesSent,0,NULL,NULL);
    printf("Sent:%d bytes \n",dwBytesSent);
  }
}

if (ne.lNetworkEvents &FD_CLOSE)
{
  //关闭发生的通知
}

if (ne.lNetworkEvents &FD_QOS)
{
char buf[QOS_BUFFER_SZ];
QOS *lpqos = NULL;
DWORD dwBytes;
BOOL bRecvRESV = FALSE;

if (ne.iErrorCode[FD_QOS_BIT])
{
  printf("FD_QOS error:%d \n",ne.iErrorCode[FD_QOS_BIT]);
  if (ne.iErrorCode[FD_QOS_BIT]== WSA_QOS_RECEIVERS)
    bRecvRESV = TRUE;
}

else
  printf("FD_QOS \n");
```

```
lpqos = (QOS *)buf;
WSAIoctl(s,SIO_GET_QOS,NULL,0,
        buf,QOS_BUFFER_SZ,&dwBytes,NULL,NULL);
//
//检查一下，看看 QOS 结构中是否返回了状态对象，
//该结构中也可能会包含 WSA_QOS_RECEIVERS 标志
//
if (ChkForQosStatus(lpqos,WSA_QOS_RECEIVERS))
   bRecvRESV = TRUE;

if (iSetQos == SET_QOS_EVENT)
{
   lpqos->SendingFlowspec.ServiceType =
   clientQos.SendingFlowspec.ServiceType;
   WSAIoctl(s,SIO_SET_QOS,lpqos,dwBytes,
      NULL,0,&dwBytes,NULL,NULL);
   //
   //改变 iSetQcs，这样在接收到另一个事件时就不用再次设置 QOS 了
   //
   iSetQos = SET_QOS_BEFORE;
}

if (bWaitToSend &&bRecvRESV)
{
   wbuf.buf = databuf;
   wbuf.len = DATA_BUFFER_SZ;
   WSASend(s,&wbuf,1,&dwBytesSent,0,NULL,NULL);
   printf("Sent:%d bytes \n",dwBytesSent);
}
}
```

QOSTCP.CPP 的大部分都在与处理其他事件，例如 FD_READ、FD_WRITE 以及 FD_CLOSE 等等，其过程和第 5 章讲述过的 WSAEventSelect 示例代码差不多。惟一要注意的是 FD_WRITE 事件中的一个问题。我们提供的一个命令行选项，是等到接收到一条 RSVP PATH 消息之后，再将数据发送出去。假如要传输的数据可能超过网络上的可用带宽，那么这样做便显得颇为必要。AbleToSend 函数会调用 SIO_CHK_QOS，以判断请求的 QOS 参数是否在可用带宽的限制范围之内。如答案是肯定的，便可放心大胆地发出数据；否则，应等候发出数据的确认信号。

就客户机的情况来说，在此打算接收 WSA_QOS_RECEIVERS 消息，以指示对一条 RESV 消息的接收。可在收到一个 FD_QOS 事件后，再来做这些事情。此时，可调用 SIO_CHK_QOS 命令，取得相关的状态信息。该 WSA_QOS_RECEIVERS 标志可以两种方式返回。第 1 种方式，可在

WSANETWORKEVENTS 结构的 iErrorCode 字段中返回这个标志，它对应于由 FD_QOS_BIT 索引的某个数组元素。第 2 种方式，可在传递给 WSAIoctl 的缓冲区内返回一个 RSVP_STATUS_INFO 结构(使用 SIO_GET_QOS 这个 I/O 控制命令)。在这个结构中，也可能在其 StatusCode 字段中包含了 WSA_QOS_RECEIVERS 标志。如决定等待发送标志，那么需要检查来自 WSANETWORKEVENTS 的错误字段，了解是否返回了一个 RSVP_STATUS_INFO 结构。假如该标志位已设置，便可放心地发出数据。到此为止，所有工作结束。要想为客户机提供对 QOS 的支持，相应的代码是非常明了的。

在服务器那一边，这个例子便显得稍微有一些复杂，但也仅仅是因为它需要管理零个或多个客户机连接。监听套接字和客户机套接字都在一个名为 sc 的数组中进行处理。其中，数组元素 0 对应的是监听套接字，而剩下的元素均分配给可能的客户机连接。全局变量 nConns 包含了当前连接的客户机数量。只要完成了一个客户机连接，当时剩下的所有活动套接字都会自动向套接字数组的起始处靠拢。另外，还有一个对应的事件句柄数组。如用户决定在接受客户机连接之前，先设好 QOS，那么服务器首先会绑定监听套接字，再设置接收 QOS。监听套接字上设置的任何 QOS 参数都会被复制给客户机连接(除非服务器正在使用 AcceptEx)。监听套接字会对自己进行注册，要求只接收 FD_ACCEPT 事件通知。在服务器例程中，剩下的部分是一个大循环，等候在套接字句柄数组上的事件。最开始的时候，数组中惟一的套接字便是监听套接字。而随着越来越多的客户机连接建立，还会出现更多的套接字以及与它们对应的事件。假如由某个事件导致 WSAWaitForMultipleEvents 这个等待循环终止，而且指明事件句柄在数组元素 0 中，就表明那个事件是在监听套接字上发生的。如发生这种情况，程序便会调用 WSAEnumNetworkEvents，以调查具体发生的是什么事件。假如事件是在某个客户机套接字上发生的，那么代码会调用控制器例程 HandleClientEvents。

在监听套接字上，我们最感兴趣的事件是 FD_ACCEPT。若发生这种事件，便会用一个条件函数调用 WSAAccept。但是请记住，此时传递给条件函数的 QOS 参数是不可信任的。假如在 Windows 98 和 Windows Me 操作系统中，QOS 参数为非空值，那么必须设置某种形式的 QOS。Windows 2000 则不存在这方面的限制：QOS 可在任何时候设置。假如用户指出 QOS 要在接受调用的过程中设置，那么它会在条件函数中进行。一旦客户机套接字被接受，便会创建一个对应的事件句柄，而且为那个套接字注册恰当的事件。

函数 HandleClientEvents 可用来控制客户机套接字上发生的任何事件。读和写事件用不着多介绍，惟一要注意的是在发送之前，是否等待预约确认。假如用户要求在发送数据之前，等候预约确认的到达，那么客户机需要等待在一个 FD_QOS 事件中返回的 WSA_QOS_RECEIVERS 消息。该消息返回之后，除非接收到 FD_QOS，否则数据仍然不能发送出去。这个例子最重要的部分便是如何在套接字以及 FD_QOS 控制器中进行 QOS 的设置。

10.5.2　UDP

最后一个例子 QOSUDP.CPP 展示了如何在 UDP 上建立一个 QOS，并在不使用 RSVP 传信的情况

下激活 IP 数据包调度器。因为 Windows XP 不支持 RSVP 传信，所以这个例子主要为 Windows XP 而设计(尽管它可以在其他支持 QOS 的 Windows 平台上运行)。

这个例子就是一个发送者，它将数据报传输到一个指定的接收主机，该主机从 5150 端口接收数据报。在数据报被传输之前，使用一个 QOS_DESTADDR 对象调用 SIO_SET_QOS 便在套接字上建立了 QOS。要注意的一点是，由于标志 SERVICE_NO_QOS_SIGNALING 和字段 ServiceType FLOWSPEC 被实施“或”操作，因此，不管该平台是否支持 RSVP 传信，这个应用程序在所有平台上表现一样。

10.6 ATM 和 QOS

Windows 98(安装 SP1 的服务)、Windows Me、Windows 2000 和 Windows XP 都支持来自 Winsock 的、固有的 ATM 编程技术。QOS 是 ATM 网络天生就提供的一种功能；在 ATM 网络上，原先通过 IP 实现 QOS 必需的一系列网络、应用程序及策略组件均成了多余的东西，包括许可控制服务(Admission Control Service)以及 RSVP 协议。相反，ATM 交换机会自动进行带宽的分配，并可自动预防带宽的浪费。

除已经指出的这些差异之外，Winsock API 函数与 ATM QOS 配合使用时，其行为也与通过 IP 使用 QOS 时存在一些区别。第 1 个重要的区别是对于 QOS 带宽请求来说，对它的控制是作为连接请求的一部分来进行的。这和通过 IP 实现 QOS 时是不同的，后者要求 RSVP 会话的建立与正式的连接分离开。此外，假如在 ATM 环境中拒绝了一个带宽请求，那么连接就会失败。

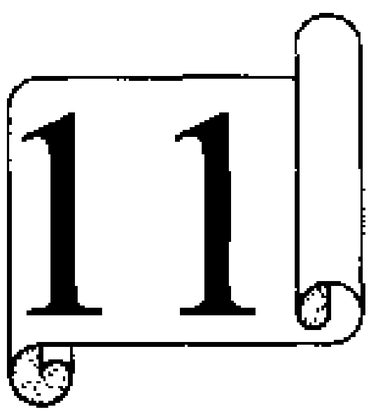
这样使引发了下一个论点：两个 ATM 提供程序均是面向连接的。因此，不必操心如何为一个无连接的套接字设置 QOS 等级，再来指定要与之通信的端点。另一个主要区别在于，对一个连接而言，只需由一方来设置 QOS 参数就可以了。也就是说，假如客户机希望对一个连接的 QOS 进行设定，那么在传给 WSAConnect 的 QOS 结构中，无论发送还是接收 FLOWSPEC 结构都会被设定。随后，这些值可立即应用于此次连接。而通过 IP 实现对 QOS 的支持时，发送者必须先请求一个特定的 QOS 等级，然后由接收者负责实际的预约。除此以外，还有可能需要用 WSAIoctl 及 SIO_SET_QOS 来设置监听套接字上的 QOS。这些值可应用于任何传入的连接。这同时意味着，在连接设置期间，必须设置好 QOS。注意不能在一个已经建好的连接上设置 QOS。

这样使引入了我们的最后一个论点：一旦为某个连接设好了 QOS，便不能通过调用 WSAIoctl 以及 SIO_SET_QOS，进行 QOS 的重新协商。若在一个连接上设好了 QOS，它的作用会一直持续下去，直到连接关闭。

注意 RSVP 在 ATM 的环境中已经不复存在，而且也不会发生任何传信事件。也就是说，表 10.5 总结的任何状态标志都不会生成。QOS 是在建立连接的过程中设置好的，除非这个连接关闭，否则永远都不会产生任何额外的通知或事件。

10.7　小　　结

QOS 为应用程序提供了更多的功能，使其可轻松地享用一个稳定的网络服务等级。建立 QOS 连接显得有些麻烦，但不要觉得困难。最需要掌握的是怎样以及何时生成 RSVP 消息；再以此为依据，相应地编写程序代码。尽管在 Windows 平台上 GQOS 的未来还不确定，但最新的 Windows XP 平台中仍然还有通信控制功能。



原始套接字

利用原始套接字可访问位于基层的传输协议。本章专门讲解如何运用这种原始套接字，来模拟 IP 的一些实用程序，例如 Traceroute 和 Ping 等等。使用原始套接字，也可对 IP 头信息进行操作。本章只关心 Ipv4 和 Ipv6 协议；至于如何针对其他协议使用原始套接字，这里不打算提及。而且，大多数协议(除 ATM 以外)根本就不支持原始套接字。所有原始套接字都是使用 SOCK_RAW 这个套接字类型来创建的，而且目前只有 Winsock 2 提供了对它的支持。因此，无论 Windows CE 还是老版本的 Windows 95(无 Winsock 2 升级)，都不能使用原始套接字。

此外，要想顺利地使用原始套接字，要求对基层的协议结构有一定程度的认识，而那已不是本书的重点。在这一章中，我们打算讨论 ICMP、ICMPv6 及 UDP。Ping 这个实用程序会用到 ICMP(包括两个版本)，以便探测到某个主机的路由是否有效和畅通，以及对方的计算机是否会作出响应。对程序开发者来说，经常都要用到一种编程方法来判断一台计算机是否活跃，能否访问它。至于 UDP 协议，我们打算把它同 IP_HDRINCL 这个套接字选项组合起来讨论。以它为例，讲述如何发送构造完整的 IP 包。对这些协议来说，我们都只会讲解与本章示例代码及示例程序密切相关的那些部分。更具体的内容可参考 W. Richard Stevens 关于 IP 的著作《TCP/IP Illustrated》第 1 卷(Addison-Wesley,1994)，也可参考每种协议的 RFC。

11.1 创建原始套接字

使用原始套接字的第 1 步便是创建套接字。可用 socket 命令或 WSASocket 调用来实现。注意，对 Windows 95、Windows 98 及 Windows Me 而言，在 Winsock 为 IP 列出的编录中，并不存在 SOCK_RAW 这一套接字类型。然而，这并不会妨碍我们创建此种类型的套接字。它的意思只是说，不能用 WSAPROTOCOL_INFO 结构来创建原始套接字。请参考第 2 章，那里详细讲述了如何用 WSAEnumProtocols 函数以及 WSAPROTOCOL_INFO 结构来列举协议条目。注意在套接字的创建过程中，必须手动设定 SOCK_RAW 标志。下述代码片断解释了如何将 ICMP 作为一种基层 IP 协议，来完成原始套接字的创建：

```
SOCKET s;

s = socket(AF_INET,SOCK_RAW,IPPROTO_ICMP);
//或
s = WSASocket(AF_INET,SOCK_RAW,IPPROTO_ICMP,NULL,0,
    WSA_FLAG_OVERLAPPED);
if (s == INVALID_SOCKET)
{
    //套接字创建失败
}
```

创建原始套接字时，套接字调用的协议参数变为 IP 头中的协议值。也就是说，如果一个原始的 AF_INET6 套接字是用协议值 66 创建的，则外出数据包的 IPv6 头将在下一个头字段中包含数值 66。

由于原始套接字使人们能对基层传输机制加以控制，所以有些人将其用于不法用途，从而成了 Windows NT 潜在的安全漏洞。因此，只有属于“管理员”(Administrators)组的成员，才有权创建 SOCK_RAW 类型的套接字。谁都可以在 Windows NT 上创建一个原始套接字，但非管理员就没法动用它，因为 bind API 会失败，并返回 WSAEACCES。而 Windows 95、Windows 98 和 Windows Me 均未施加这方面的限制。

要想在 Windows NT 中绕过这一限制，可考虑禁止对原始套接字的安全检查。方法是在注册表中创建如下变量，并将它的值设为 1(DWORD 类型)：

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet
   \Services\Afd\Parameters\DisableRawSecurity
```

更改了注册表后，注意重新启动计算机。

在上述示例代码中，我们采用的是 ICMP 协议。但假如想使用 IGMP、UDP、IP 或者原始 IP，只需分别设置 IPPROTO_IGMP、IPPROTO_UDP、IPPROTO_IP 或者 IPPROTO_RAW 即可。然而，请注意，在 Windows 95(安装了 Winsock 2)、Windows 98 以及 Windows NT4 操作系统中，创建原始套接字时，只能使用 IGMP 和 ICMP。协议标志 IPPROTO_UDP、IPPROTO_IP 以及 IPPROTO_RAW 均要求使用套接字选项 IP_HDRINCL，而该选项上述平台都是不支持的。然而，Windows Me 和 Windows 2000 及其后期版本提供了对 IP_HDRINCL 选项的支持，所以能够处理 IP 头(IPPROTO_RAW)、TCP 头(IPPROTO_TCP)以及 UDP 头(IPPROTO_UDP)。

采用恰当的协议标志完成了原始套接字的创建之后，便是在发送及接收调用中使用对应的套接字句柄。创建原始套接字时，IP 头会包含在任何接收到的返回数据中，无论是否设定了 IP_HDRINCL 选项。应用程序必须指定 IP 头的布局，并确定 IP 头的长度，以便找到接收缓冲区内的有效负载数据。

11.2 ICMP

ICMP 是使于不同主机间传递消息的一种机制，它有两个版本。原始 ICMP 和 IPv4 一起使用，用来在两个主机之间传递信息性消息，通常牵涉到主机间通信时发生的一些错误，如目标不可抵达或 TTL 超出等。对于 IPv6 则有一个新创建的版本 ICMPv6。ICMPv6 不仅包括指示性信息，还包括 ND 和 MLD。第 3 章已经讨论过，ND 等同于 ARP，MLD 等同于 IGMP。两个 ICMP 版本的讨论都将限于信息性消息。

前面已经提到过，ICMP 协议采用的是 IPv4 寻址机制，因为它本身就是直接封装在 IPv4 数据报内的一种协议。图 11.1 展示了 ICMP 消息的各个字段。ICMPv6 封装在一个 IPv6 数据报内，其结构和 ICMP 数据包一样(至少从前 4 个字节看是如此)。

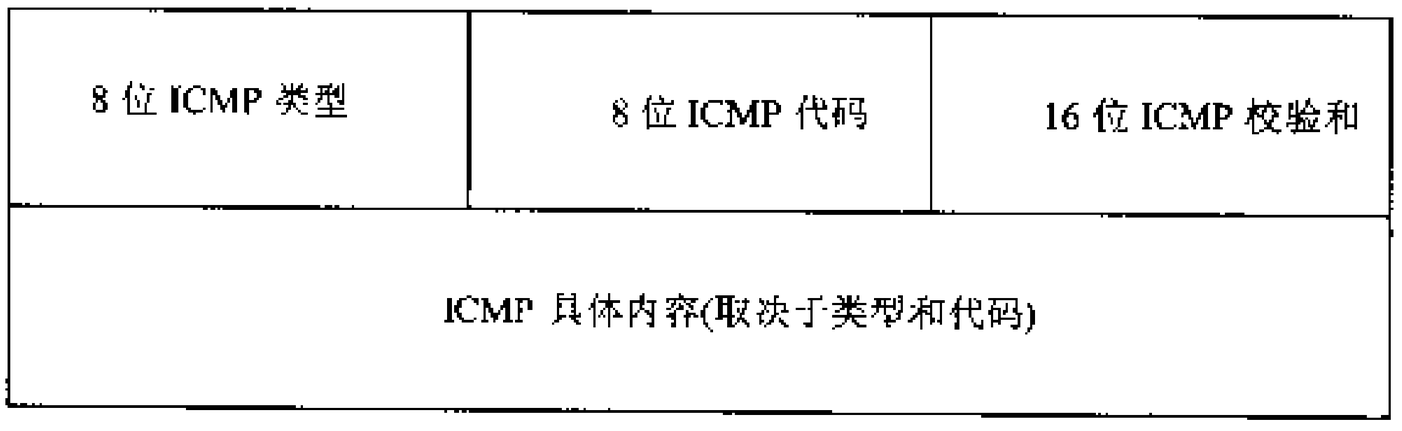


图 11.1 ICMP 头

其中，第 1 个字段指定的是 ICMP 消息类型，通常分为查询类型和错误类型两类。随后，代码字段进一步定义了查询或消息的类型。而校验和字段的长度为 16 位，是对 ICMP 头的求补和。应注意到，IPv4 和 IPv6 的校验和计算是有差别的。对 IPv4，校验和仅在 ICMP 头及其有效负载上计算，而对 IP6，检验和则在 IPV6 伪头、ICMPv6 头及其有效负载上计算。IPV6 伪头由下列字段组成：

- 128 位 IPv6 源地址
- 128 位 IPv6 目标地址
- 32 位上层协议数据包长度
- 24 位置零字段
- 8 位下一个头协议值

IPv6 要求通过一个上层协议对伪头进行检验和计算，该上层协议包含来自 IP 头的地址，包括 UDP 和 ICMPv6。如果上层协议包含了自己的数据包长度字段，则该长度值将被用于伪头的计算中。否则，就使用来自 IPV6 头的有效负载长度，再减去现有的 IPv6 扩展头的长度。本章后面的图 11.5 展示了 IPv6 伪头、UDP 头和有效负载。

最后，ICMP 的实际内容要依赖于前面设定的 ICMP 类型及代码。表 11.1 中列出了 ICMP 最常见的类型和代码，而表 11.2 中则列出了 ICMPv6 最常见的类型和代码。ICMP 数据包的类型和代码决定

着 ICMP 头后跟随的内容。

表 11.1　ICMP 消息类型

类　型	查询 / 错误(错误类型)	代　码	说　明
0	查询	0	回应答复
3	错误：目标不可抵达	0	网络无法访问
3	错误：目标不可抵达	1	无法访问主机
		2	无法访问协议
		3	无法访问端口
		4	数据包需要分段，但却设置了“不分段”位
		5	源路由失效
		6	目标网络未知
		7	目标主机未知
		8	源主机被隔离(作废)
		9	目标网络管理性禁止
		10	目标主机管理性禁止
		11	针对特定的 TOS(服务类型)，网络不可访问
		12	针对特定的 TOS，主机不可访问
		13	在筛选时通信被管理性地禁止
		14	违反主机优先级设定
		15	优先级失效
4	错误	0	源主机停止工作
5	错误：重定向	0	为网络重定向
		1	为主机重定向
		2	为 TOS 和网络重定向

续表

类 型	查询 / 错误(错误类型)	代 码	说 明
		3	为 TOS 和主机重定向
8	查询	0	回应请求
9	查询	0	路由器广告
10	查询	0	路由器请求
11	错误：超时	0	传输过程中 TTL 的值变成 0
		1	重新组合时 TTL 的值变成 0
12	错误：参数问题	0	IP 头损坏
		1	必需的选项丢失

若产生了一条 ICMP 错误消息，那么在消息中，在不超过 MTU 大小的情况下总是包含尽量多的 IP 头和 IP 有效负载，两者导致了那个 ICMP 错误发生。这样，收到 ICMP 错误的主机便可将消息与一种特定的协议及与错误相关的进程关联起来。就我们目前的情况来说，Ping 依赖于回应请求及回应答复这两种 ICMP 查询，而不仅仅是错误消息。在下一节里，我们打算讨论如何通过回应请求和回应答复消息，采取原始套接字，用 ICMP 协议来生成一个 Ping 请求。如果想知道关于 ICMP 错误或其他类型的 ICMP 查询，可参考讲述更透彻的资料，如 W. Richard Stevens 关于 IP 的著作《TCP/IP Illustrated》第 1 卷(Addison-Wesley,1994)。同时，关于 ICMP 和 ICMP6 的更多内容，分别参见 RFGs792 及 2463。

表 11.2 ICMPv6 消息类型

类 型	查询 / 错误(错误类型)	代 码	说 明
1	错误：目标不可抵达	0	没有到达目标的路线
		1	和目标的通信被管理性地禁止
		3	不能访问地址
		4	不能访问端口
2	错误：数据包过大	0	数据包比 MTU 大，不能继续转发
3	错误：超时	0	传输过程中超出中继限制
		1	片断组装超时
4	错误：参数问题	0	遭遇错误的头字段

续表

类　型	查询 / 错误(错误类型)	代　码	说　明
		1	遭遇的下一个头类型无法辨识
		2	遭遇无法辨识的 IPv6 选项
128	查询：回应请求	0	请求目标回应 ICMP 有效负载
129	查询：回应答复	0	答复一个回应请求的查询

11.2.1　Ping 示例

我们经常用 Ping 来判断一个特定的主机是否处于活动状态，以及是否可以通过网络访问它。如果生成一个 ICMP 回应请求，并将其定向至目标主机，便可知道是否能成功地访问到那台计算机。当然，这样做并不能担保一个套接字客户机能与那个主机上的某个进程顺利地建立连接(例如，远程服务器上的一个进程也许并没有进入监听模式)。若 Ping 成功，那么它只能证明一件事情：远程主机的网络层可对网络事件作出响应。大多数操作系统提供一种功能，可以停止对 ICMP 回应请求作出响应，一般运行有防火墙的计算机都会这样作。概括地说，这个 Ping 示例需要采取下面这些步骤：

1. 创建一个 AF_INET 地址族的 SOCK_RAW 类型的套接字，同时设定协议 IPPROTO_ICMP。对于 IPv6 而言，地址族为 AF_INET6，类型为 SOCK_RAW，协议值为 58。
2. 创建并初始化 ICMP 头。
3. 调用 sendto 或 WSASendto，将 ICMP 请求发给远程主机。
4. 调用 recvfrom 或 WSARecvfrom，以接收任何 ICMP 响应。

初始化 ICMP 头是一个简单的工作。首先，初始化 ICMP 头的类型和代码。记住，ICMP 和 ICMPv6 的头是一样的(如图 11.1 所示)。在类型和代码之后，必须紧跟回应请求头。这个头如图 11.2 所示。

16 位标识符	16 位顺序号
32 位时标	
有效负载	

图 11.2　回应请求头

第 1 个字段是一个 16 位标识符，用来惟一标识这个请求，并用来使该请求和它接收到的回应答复相关联，而不和其他进程的请求关联。通常会使用发送进程的进程标识符。下一个字段是顺序号，用来标识给定的请求数据包，以便和其他请求数据包区别开来。32 位时标字段仅用于 ICMP 请求(不用于 ICMPv6 请求)。跟随这个请求头的是任意的有效负载。下述代码展示了 IPv4 中 ICMP 回应请求的初始化和发送过程：

```
//定义 ICMP 头
typedef struct icmp_hdr
{
    unsigned char      icmp_type;
    unsigned char      icmp_code;
    unsigned short     icmp_checksum; .
    unsigned short     icmp_id;
    unsigned short     icmp_sequence;
    unsigned long      icmp_timestamp;
}ICMP_HDR,*PICMP_HDR,FAR *LPICMP_HDR;

ICMP_HDR           *icmp = NULL;
SOCKET              s;
SOCKADDR_STORAGE dest;
char                buf[sizeof(ICMP_HDR) + 32];

icmp = (ICMP_HDR *)buf;
icmp->icmp_type = 8;          //回应请求类型
icmp->icmp_code = 0;
icmp->icmp_id = GetCurrentProcessId();
icmp->icmp_checksum = 0;     //在计算校验和之前将字段置零
icmp->icmp_sequence = 0;
icmp->icmp_timestamp = GetTickCount();
//用一个随机字符填入有效负载
memset(&buf[sizeof(ICMP_HDR)],'@',32);
//在 ICMP 头和有效负载之上计算校验和
//checksum()函数在指定缓冲区上计算 16 位 1 求补
//其实现过程参见补充材料中的 Ping 代码示例
icmp->icmp_checksum = checksum(buf,sizeof(ICMP_HDR) + 32);

s = socket(AF_INET,SOCK_RAW,IPPROTO_ICMP);
//初始化目标 SOCKADDR_STORAGE
((SOCKADDR_IN *)&dest)->sin_family = AF_INET;
((SOCKADDR_IN *)&dest)->sin_port = htons(0);
//对于 ICMP，忽略端口
((SOCKADDR_IN *)&dest)->sin_addr.s_addr = inet_addr("1.2.3.4");
sendto(s,buf,sizeof(ICMP_HDR) + 32,0,(SOCKADDR *)&dest,
    sizeof(dest));
```

ICMP 和 ICMPv6 回应请求之间的另一个区别是对 ICMP 头中包含的检验和的计算。对 IPv4 而言，检验和仅在 ICMP 头和有效负载之上计算。因为 IPv6 要求检验和包括 ICMPv6 头和有效负载之前的 IPv6 伪头，所以 IPv6 更复杂一些。这意味着 Ping 应用程序必须知道 IPv6 头中的 IPv6 源地址和目标地址，用来计算任何外出 ICMPv6 请求的检验和。因为 IPv6 头不是我们建立的(如用 IPV6_HDRINCL 选项建立的 IPv6 头)，因此我们无法控制进入 IPv6 头的内容。但是，可以向传输查询，以便确定用哪

个本地接口来访问给定目标。这由 I/O 控制命令 SIO_ROUTING_INTERFACE_QUERY 来实施。查询完成之后，就有足够的信息来计算伪头校验和了。

发出 ICMP 回应请求以后，远程计算机会截获这个请求，然后生成一条回应答复消息，再传回给我们。假如出于某些方面的原因，不能访问目标主机生成，原先打算建立通信的那个路径上某处的一个路由器，就会返回对应的 ICMP 错误消息(例如“目标主机访问不到”)。假定与远程主机的物理网络连接并不存在问题，但远程主机已经关机或没有对网络事件作出响应，便需由自编的程序执行超时检定，侦测出这样的情况。因为回应请求中的时标已经被响应，收到这个答复之后，已用时间便很容易计算。补充材料中的 Ping.cpp 示例清晰地展示了如何创建一个套接字实现 ICMP 包的收发；以及如何通过 IP_OPTIONS 套接字选项，来实现记录路由选项(仅支持 IPv4)。

Ping 示例的一个显著特征，是使用了 IP_OPTIONS 套接字选项。记录路由 IPv4 选项的使用，使得只要 ICMP 数据包一遇到路由器，它的 IPv4 地址就被添加到 IPv4 选项头中，添加位置由 IPv4 选项头的偏移字段指明。路由器每添加一次地址，这个偏移值就增加 4。增量值是基于 IPv4 地址长度为 4 字节这一事实的。接收到回应答复之后，就可以将选项头解码，输出已访问的路由器的 IP 地址及主机名。关于其他类型的可用 IP 选项请参见第 7 章。

11.2.2　Traceroute 示例

对 IP 网络来说，另一个非常有价值的实用程序是 Traceroute。在 Windows 操作系统中，一般可以直接运行 Tracert.exe，来调用这个工具。利用它，我们可侦测出在抵达网络内任何一个指定的主机前，中途需经过哪些路由器，以及它们的 IP 地址是什么。当然，亦可利用 Ping，使用 IPv4 选项头内的记录路由选项，侦测中途经过的路由器 IPv4 地址。然而，Ping 最多只支持 9 次“跳跃”——这是在选项头内为地址分配的最大空间。同时，IPv6 不存在对等的选项。一个 IP 数据报需要穿越多个物理性的网络时，每通过一个路由器，我们就说进行了一次“跳跃”。

Traceroute 的设计原理是令其向目的地发送一个 UDP 数据包，并重复递增 TTL 值。最开始的时候，TTL 等于 1；也就是说，一旦它抵达路途中的第 1 个路由器，TTL 就会超时(变成 0)。这样路由器便会生成一个 ICMP 超时数据包。随后，最初的 TTL 值增加 1，以便 UDP 包能继续传到下一个路由器，而生成的 ICMP 超时包会自那个路由器返回。只需将返回的每一条 ICMP 消息都收集下来，便能为中途经过的路由器 IP 地址划出一条清晰的路线，直到最终抵达目标主机。所以一旦 TTL 的值递增得足够大，可以实际抵达目标位置，通常使会返回一条 ICMP“端口访问不到”消息，因为在接收端主机上，并没有进程在等待这条消息。

之所以认为 Traceroute 是一个有用的工具，是由于它为我们提供了与中途经历的路由器有关的大量信息。进行多播通信，或者在遇到路由问题的时候，这些资料便显得非常宝贵。对具体的应用程序来说，尽管需要执行 Traceroute 的机会比 Ping 少得多，但对某些特定的任务而言，却恐怕只有 Traceroute 才能胜任。

有两个办法可用来实现 Traceroute 程序。第 1 个办法，可使用 UDP 包和发送数据报，连续递增 TTL 的值。TTL 每一次超时，都会返回一条 ICMP 消息。这种方法要求使用 UDP 套接字来发送消息，同时用 ICMP 套接字来读取返回的消息。这个 UDP 套接字本身是一个极为普通的 UDP 套接字，如大家在第 1 章中所见。而 ICMP 套接字是原始套接字，前面已经讲述了创建方法。UDP 套接字的 TTL 值需要通过 IP_TTL 或 IPV6_UNICAST_HOPS 套接字选项加以控制。此外，也可以创建一个 UDP 套接字，并使用 IP_HDRINCL 选项(本章稍后还会详述)，在 IP 头内对 TTL 进行人工设置。当然，这要求程序员做更多的工作。

第 2 个办法，我们只需将 ICMP 数据包简单地发给目的地，同时连续递增 TTL 的值。在 TTL 超时的时候，这样做也会造成 ICMP 错误消息的返回。注意这种方法和 Ping 有着某些共通之处，它只要求使用一个 ICMP 套接字。在补充材料的示例代码文件夹下，大家可找到一个名为 TRACERT.CPP 的使用了 ICMP 数据包的 Traceroute 例子。该示例和 Ping 示例的结构和代码类似。惟一的区别是，Traceroute 示例中 TTL 值随着每次发送而增加。

11.3 使用 IP 头包含选项

对原始套接字来说，它存在的一项局限在于，我们只能对已经定义好的协议进行操作，例如 IGMP 和 ICMP 等等。不能用 IPPROTO_UDP 来创建一个新的原始套接字，也不能对 UDP 头进行操作；TCP 也是一样的。要想对 IP 头进行操作，同时也能操作 TCP 或 UDP 头(或封装在 IP 内的其他任何协议)，必须随原始套接字一道，使用 IP_HDRINCL 这个套接字选项。对于 IPv6 这个选项是 IPV6_HDRINCL。利用该选项，我们除了能自行构建 IP 头，亦能构建其他协议头。

除了操作已知协议如 UDP 之外，连同头包含选项一起使用原始套接字，就可以在应用程序中实现自己的协议方案，并将其封装到 IP 内。要达到这一效果，首先创建一个原始套接字，然后将协议类型设为 IPPROTO_RAW 即可。这样就可在 IP 头内人工设置协议字段，同时构建自定义的协议头。在本小节内，我们打算讲述如何构建自己的 UDP 数据包，使大家对其中牵涉到的步骤有一个比较全面的了解。一旦理解了如何操作 UDP 头，再来创建自己的协议头，或对 IP 内封装的其他协议进行处理，就比较容易了。

在讨论头包含选项的具体内容之前，需要弄清楚在 IPv4 和 IPv6 中使用这个选项的一个重要差别。对 IPv4，堆栈仍会验证 IPv4 头内部的字段。例如，IPv4 标识字段由堆栈设置，如果有必要，堆栈会将该数据包打散。也就是说，如果创建了一个原始 IPv4 数据包，设置了 IP_HDRINCL 并发送了一个比 MTU 大的数据包，堆栈就会将数据分割成多个数据包。对 IPv6，如果设置了 IPV6_HDRINCL 选项，计算所有必要的头和字段就得靠应用程序自身来进行。如果提交一个比 MTU 大的发送包，应用程序必须创建 IPv6 片断头，并正确计算偏移值，否则 IPv6 堆栈就会丢掉该数据包，而不发送了。

使用头包含选项对，必须针对每一次发送调用，动手向 IP 头内填充内容。同时，还要填写打包

在其中的其他协议头。IPv4 头和 IPv6 头的结构如第 7 章图 7.3、7.4 所示。与 IP 头相比，UDP 头要简单得多，长度仅为 8 个字节，而且只包含了 4 个字段，如图 11.3 所示。其中，头两个字段分别对应源和目标端口号，长度各自为 16 位。第 3 个字段对应 UDP 长度；以字节为单位，指定 UDP 头以及数据的总长。第 4 个字段则是校验和，稍后还会对此详加解释。UDP 包的最后一部分便是实际的数据。

16 位源端口	16 位目标端口
16 位 UDP 长度	16 位 UDP 校验和

图 11.3　UDP 头的格式

由于 UDP 是一种不能保证数据可靠传输的协议，所以校验和的计算是可选的。与 IPv4 校验和不同，UDP 校验和除包括了实际的数据，还包括 IPv4 头的一部分。而 IPv4 检验和只针对 IPv4 头进行计算。用于计算 UDP 校验和的附加字段叫作“伪头”。一个伪头由下述项目构成：

- 32 位源 IP 地址(IP 头)
- 32 位目标 IP 地址(IP 头)
- 8 位字段(置零)
- 8 位协议
- 16 位 UDP 长度

添加进这些项目的是 UDP 头以及数据。校验和的计算方法是得出 16 位 1 的求补总和。由于数据可能是奇数个字节，所以有时需要在数据末尾填充一个 0 字节，以便顺利计算出检验和。这个填充字段并不作为数据的一部分传递。图 11.4 展示了计算校验和所需的全部字段。头 3 个 32 位字构成了 UDP 伪头。紧接着的是 UDP 头及其数据。要注意的是，由于检验和是以 16 位值为基础计算出来的，所以或许需要用一个 0 字节对数据进行补位。

32 位源 IPv4 地址		
32 位目的 IPv4 地址		
0	8 位协议	16 位 UDP 长度
16 位源端口		16 位目标端口
16 位 UDP 长度		16 位 UDP 校验和
数据(以及任意填充字节)		

图 11.4　带 UDP 数据包和数据的 IPv4 伪头

对于 IPv6，我们已经知道如何计算 IPv6 伪头，这是计算 ICMPv6 数据包的校验和所必需的。UDP 的计算和 IPv6 伪头的计算一样，其中 IPv6 伪头在前面，紧跟着 UDP 头和有效负载(如果有必要，用 0 填充下一个 16 位边界)。IPv6 伪头如图 11.5 所示。

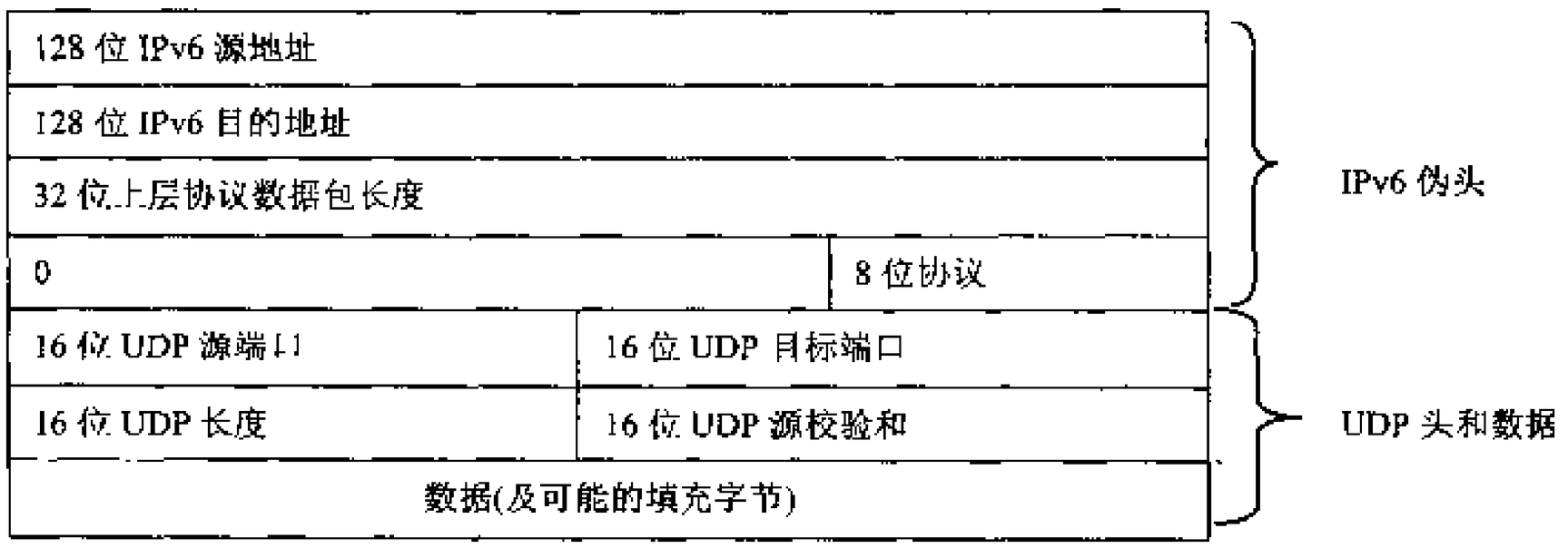


图 11.5 带 UDP 数据包和数据的 IPv6 伪头

注意 补充材料中的 IPHDRINC 目录下包含了一个功能全面的示例，完整地展示了在 IPv4 及 IPv6 上原始 UDP 数据包的创建。

下述代码片断展示了如何建立 IPv4 和 UDP 头：

```
//IPv4 头
typedef struct ip_hdr
{
   unsigned char   ip_verlen;      //4 位 IPv4 版本的 4 位头长度(用 32 位字表示)
   unsigned char   ip_tos;        //服务的 IP 类型
   unsigned short  ip_totallength;  //总长
   unsigned short  ip_id;         //惟一标识符
   unsigned short  ip_offset;      //片断偏移字段
   unsigned char   ip_ttl;        //生存时间
   unsigned char   ip_protocol;    //协议(TCP、UDP 等)
   unsigned short  ip_checksum;    //IP 校验和
   unsigned int    ip_srcaddr;     //源地址
   unsigned int    ip_destaddr;    //源地址
}IPV4_HDR, *PIPV4_HDR, FAR *LPIPV4_HDR;

//定义 UDP 头
typedef struct udp_hdr
{
   unsigned short src_portno;    //源端口号
   unsigned short dst_portno;    //目标端口号
   unsigned short udp_length;    //Udp 数据包长度
   unsigned short udp_checksum; //Udp 校验和(可选)
}UDP_HDR, *PUDP_HDR;

SOCKET     s;
char         buf[MAX_BUFFER],   //足够大的缓冲区
            *data = NULL;
IPV4_HDR   *v4hdr = NULL;
```

```
UDP_HDR   *udphdr = NULL;
USHORT    sourceport = 5000,
          Destport = 5001;
int         payload = 512,          //UDP 数据的大小
          optval;
SOCKADDR_STORAGE dest;

//初始化 IPv4 头
v4hdr = (IPV4_HDR *)buf;
v4hdr->ip_verlen = (4 <<4) | (sizeof(IPV4_HDR) / sizeof(ULONG));
v4hdr->ip_tos = 0;
v4hdr->ip_totallength = htons(sizeof(IPV4_HDR) + sizeof(UDP_HDR) +
   payload);
v4hdr->ip_id = 0;
v4hdr->ip_offset = 0;
v4hdr->ip_ttl = 8;//存活时间是 8
v4hdr->ip_protocol = IPPROTO_UDP;
v4hdr->ip_checksum = 0;
v4hdr->ip_srcaddr = inet_addr("1.2.3.4");
v4hdr->ip_destaddr = inet_addr("157.32.159.101");
//为 IPv4 头计算校验和
//checksum()函数在指定缓冲区上计算 16 位 1 求补
//其实现过程参见补充材料中的 IPHDRINC 代码示例
v4hdr->ip_checksum = checksum(v4hdr,sizeof(IPV4_HDR));

//初始化 UDP 头
udphdr = (UDP_HDR *)&buf[sizeof(IPV4_HDR)];
udphdr->src_portno = htons(sourceport);
udphdr->dst_portno = htons(destport);
udphdr->udp_length = htons(sizeof(UDP_HDR) + payload);
udphdr->udp_checksum = 0;
//将 UDP 有效负载初始化为某值
data = &buf[sizeof(IPV4_HDR) + sizeof(UDP_HDR)];
memset(data,'^',payload);

//计算 IPv4 和 UDP 伪头校验和——一这个例程从头中抽取出所有必要的字段，并在其上计算校验和。
// Ipv4PseudoHeaderChecksum()的实现过程参见示例代码
udphdr->udp_checksum = Ipv4PseudoHeaderChecksum(v4hdr,udphdr,data,
   sizeof(IPV4_HDR) + sizeof(UDP_HDR) + payload);

//创建原始 UDP 套接字
s = socket(AF_INET,SOCK_RAW,IPPROTO_UDP);

//设置头包含选项
optval = 1;
setsockopt(s,IPPROTO_IP,IP_HDRINCL,(char *)&optval,sizeof(optval));
```

```
//发送数据
((SOCKADDR_IN *)&dest)->sin_family = AF_INET;
((SOCKADDR_IN *)&dest)->sin_port = htons(destport);
((SOCKADDR_IN *)&dest)->sin_addr.s_addr = inet_addr("157.32.159.101");

sendto(s,buf,sizeof(IPV4_HDR) + sizeof(UDP_HDR) + payload,0,
       (SOCKADDR *)&dest,sizeof(dest));
```

这段代码直截了当，很容易弄懂。首先，用有效的条目初始化 IPv4 头。这里使用了一个假的源 IPv4 地址(1.2.3.4)，但提供了一个有效的目标地址。同时将 TTL 值设为 8。最后，仅对 IPv4 头校验和进行计算。之后是 UDP 头——由 IPv4 头的 ip_protocol 字段指明——被设为 IPPROTO_UDP。对这个 UDP 头，除设置 UDP 头及其有效负载的长度之外，还设置源端口和目标端口。最后一个事项，是计算伪头的校验和，这没有在代码中显示，不过计算很容易。从各种头中抽出必要的字段后，就可以计算校验和了。补充材料中有一个代码示例，其中的一个例程可来为 IPv4 和 IPv6 计算伪头校验和。

前面提到，针对 IPv4 使用头包含选项很容易，因为堆栈将执行任何必须的分割操作。然而，对于 IPv6，堆栈就不会那样作，这意味着如果应用程序需要发送有效负载超出 MTU 的原始数据，就不得不在发送数据包之前对它们进行手动分割操作。这要通过在 IPv6 头之后、剩余有效负载之前包含 IPv6 片断头来完成。要完成这一工作，IPv6 头的下一个头值应指明 IPv6 片断头(值为 44)。IPv6 片断头的下一个头值则应指明 IPPROTO_UDP。还得注意，UDP 头仅出现一次。第 1 个片断将包含 IPv6 头、IPv6 片断头、UDP 头及尽量多的和 MTU 相当的有效负载。接着的片断将只包含 IPv6 头、IPv6 片断头及剩余的有效负载。图 11.6 展示了这个示例。在这里，MTU 是 1500 字节，但发送的有效负载是 2000 字节。

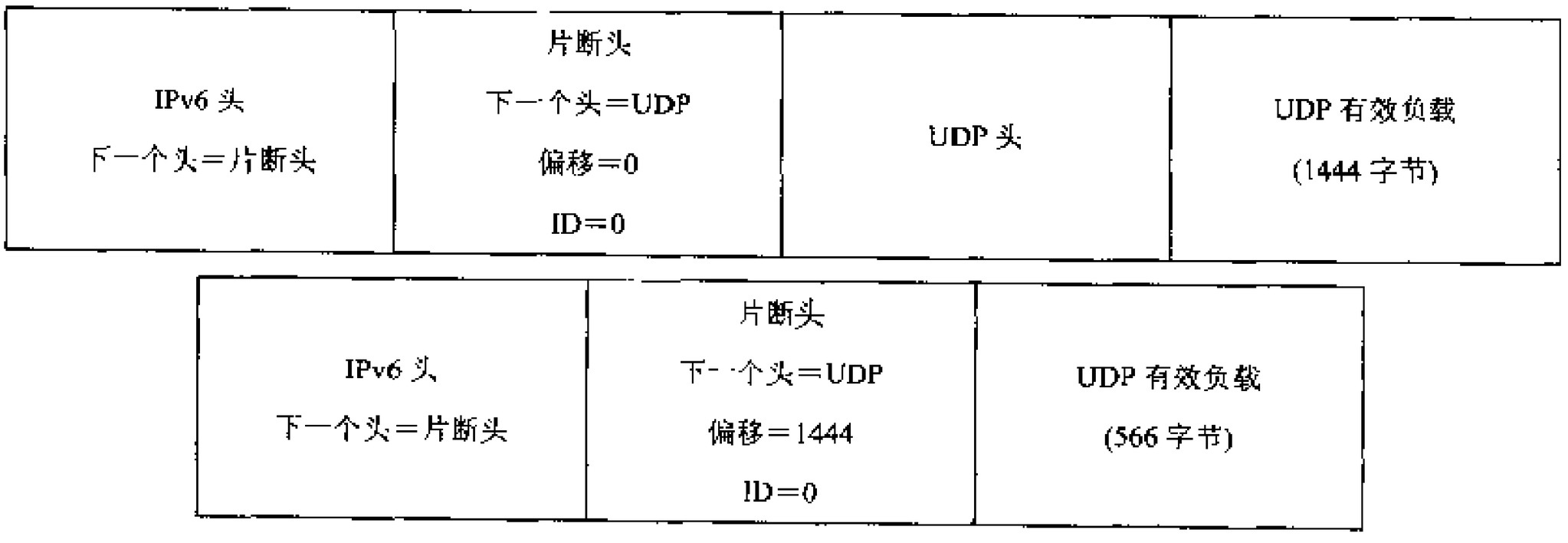


图 11.6 带片断的 IPv6UDP 数据包

补充材料中的示例 RAWUDP.CPP 展示如何在 IPv4 和 IPv6 中发送装配完成的 UDP 数据包。其中有两个相关的例程：PacketizeIpv4 和 PacketizeIpv6。因为必要时堆栈会分割数据，所以 V4 例程和这里没多大关系。然而，V6 例程会为每个必要的分割操作建立适当的 IPv6 头和片断头。

11.4　小　　结

原始套接字是一种强有力的机制，可供我们对基层协议进行操作。本章向大家阐述了如何通过 Winsock，利用原始套接字来创建 ICMP 和 ICMPv6 应用程序。但要注意的是，原始套接字也适用于其他许多应用程序，本章不可能全部都讲到。要想充分发挥原始套接字以及头包含选项(IP_HDRINCL 和 IPV6_HDRINCL)的功能，必须对 IP 协议以及 IP 内封装的其他所有协议有一个完整、透彻的理解。

12

Winsock 2 服务提供程序接口

Winsock 2 SPI(Service Provider Interface,服务提供程序接口)是对已经讨论过的 Winsock API 的补充。顾名思义，SPI 是一项对应用程序的服务，而非应用程序。服务被编写好之后便对应用程序公开，之后应用程序可以有意或无意地加载该服务。由于 SPI 是 Winsock 2 规范的一个部分，因此如果在 Windows 95 中，要求升级到 Winsock 2。图 12.1 展示了 Winsock 应用程序和 SPI 之间的联系。

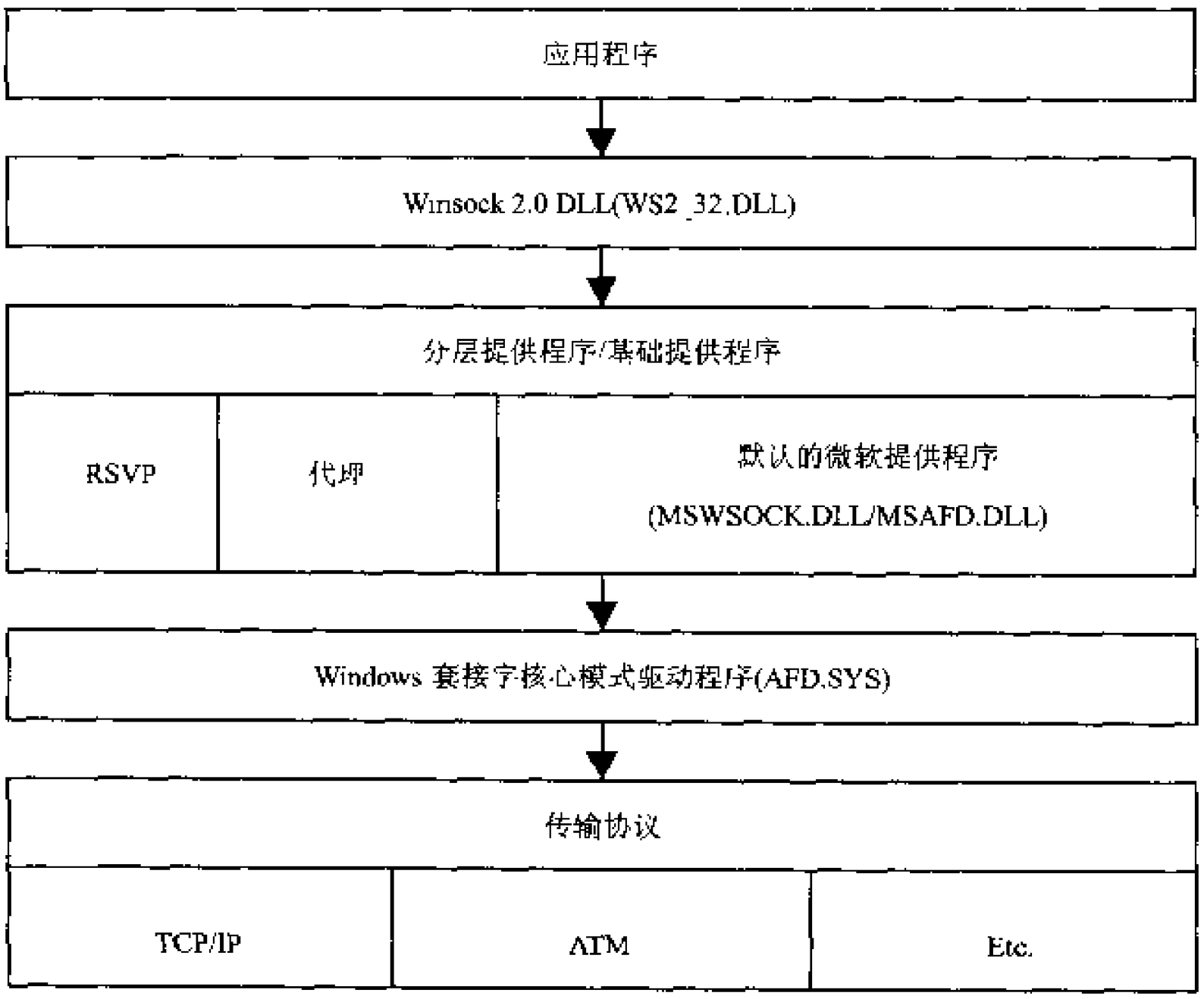


图 12.1 SPI 体系结构

SPI 分为两个部分：传输服务提供程序和命名空间提供程序。每个部分提供截然不同的功能。有两种类型的传输服务提供程序：分层服务提供程序和基础服务提供程序。分层服务提供程序将自己安

装到 Winsock 编录里边，位于基础提供程序之上，可能位于其他分层提供程序之间，并截获应用程序对 Winsock API 调用。基础提供程序公开一个 Winsock 接口，直接执行一种协议，如微软 TCP/IP 提供程序。本章仅讨论分层服务提供程序。如果应用程序创建的套接字和某个分层提供程序的特征匹配，则该分层服务提供程序就会被调用，之后就可以截获 Winsock 调用了。

命名空间提供程序和传输服务提供程序类似，只是它截获的是名称解析的 Winsock API 调用，如 gethostbyname 和 WSALookupServiceBegin。命名空间提供程序将自己安装在命名空间编录中，当应用程序执行的名称解析搜索和它匹配时，命名空间提供程序就会被调用。

在讲解具体内容之前，先介绍和分层服务提供程序及命名空间提供程序都有关的一些基本知识。Winsock 服务提供程序 API 包含在头文件 WS2SPI.H 中，SPI 应用程序和库文件 WS2_32.LIB 相链接。另外，SPI 中定义了 4 种类型的 API。表 12.1 列出了每种类型的前缀，以及它们是属于服务提供程序还是命名空间提供程序。

表 12.1　SPI 函数前缀

API 前缀	说　明
WSC	安装、删除或修改分层提供程序及命名空间提供程序
WSP	分层服务提供程序 API
WPU	分层提供程序使用的支持函数
NSP	命名空间提供程序 API

本章将先讨论分层提供程序接口，接着讨论命名空间提供程序接口。

12.1　分层服务提供程序

前面已经提过，LSP(layered service provider，分层服务提供程序)将自己安装到 Winsock 编录中，使得创建套接字的应用程序可以在不需要意识到 LSP 存在的情况下调用它。这在开发用来修改或监视 Winsock API 任何部分的系统组件时很有用。例如，一个实施 SSL 的安全套接字提供程序可以当作一个分层服务提供程序来执行。在这个例子中，当应用程序发起一个连接，并通过 Winsock 发送命令加密发送数据，同时解密从接收命令返回的数据时，LSP 将协调好 SSL 连接。分层服务提供程序的其他可能用途包括 Winsock 代理客户机和内容过滤。

通过安装一个模拟或扩展已有提供程序的全新 Winsock 提供程序，LSP 可以完成上述工作。例如，在开发一个过滤 HTTP 请求的 LSP 时，因为 HTTP 协议是在 TCP 之上运行的，所以就需要将提供程序放到微软 TCP 提供程序上面的层。因为我们希望任何使用 TCP 的应用程序都先通过提供程序，所

以这个新的提供程序和微软提供程序本质上应该难以区分(至少从应用程序的角度)看如此。当然，在已有 Winsock 提供程序之上，创建一个 LSP 使其能够执行完全不同的协议，是有可能的。

第 2 章讲述了在创建一个套接字的时候，Winsock 是如何选择并加载适当的提供程序的。当 LSP 创建好之后，就按照一定顺序放进编录中。应用程序创建套接字时，编录将按顺序列举，直到找到最好的匹配项，这时系统就会加载该提供程序。这样，分层提供程序就代替了默认的微软提供程序。

当一个根据分层提供程序创建套接字的应用程序作出一个 Winsock 调用时，系统会将调用传递到 LSP。这时 LSP 就可以执行其必要的任务了。如果需要更多的动作，它还可以将请求传递到下层的提供程序。例如，在对 HTTP 内容过滤的示例中，我们可能想截获 HTTP 请求，并在实际发出请求前修改它们。这需要 LSP 为发送数据的 Winsock API 作出一些动作。当应用程序调用 Winsock 发送函数时，调用将被路由到 LSP，LSP 将检查发送缓冲，并作出适当修改。当然，LSP 实际上并不知道如何发送 TCP 数据，它依赖于下层的 TCP 提供程序，该提供程序具有执行这种协议的核心模式驱动程序。LSP 必须知道这种协议在协议链的哪个位置，以便将修改过的发送请求传递给协议的下层提供程序。在多数情况下，该提供程序是基础提供程序，但是 LSP 也能被安装在其它 LSP 之上。最终，该请求将发向一个基础提供程序，由基础提供程序来执行适当的动作。下一节我们可以确切地看到这些协议链是如何实现的，因为安装好一个 LSP 后，这些链也必须建立。图 12.2 显示了应用程序、分层服务提供程序及基础提供程序之间的关系。

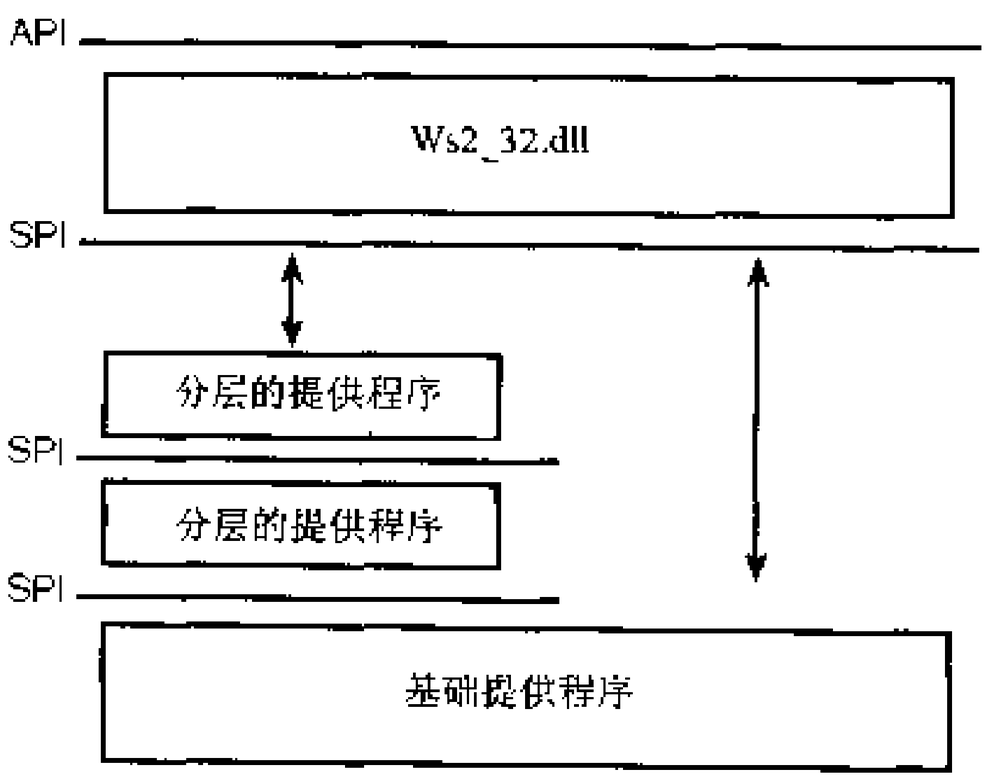


图 12.2 分层提供程序的体系结构

Winsock LSP 是作为一个标准的 Windows 动态链接库来执行的，必须将一个名为 WSPStartup 的单一函数条目导入到这个链接库里边。当系统调用分层提供程序的 WSPStartup 时，它必须通过一个作为参数传递的函数派遣表公开 30 个附加 SPI 函数，LSP 就是由这 30 个 SPI 函数组成的。表 12.2 列出了那些必须在 DLL 内部执行的 SPI 函数。

表 12.2　传输提供程序支持函数

API 函数	对应的 SPI 函数
WSAAccept(accept 也可间接映射成 WSPAccept)	WSPAccept
WSAAddressToString	WSPAddressToString
WSAAsyncSelect	WSPAsyncSelect
Bind	WSPBind
WSACancelBlockingCall	WSPCancelBlockingCall
WSACleanup	WSPCleanup
Closesocket	WSPCloseSocket
WSAConnect(connect 也可间接映射成 WSPConnect)	WSPConnect
WSADuplicateSocket	WSPDuplicateSocket
WSAEnumNetworkEvents	WSPEnumNetworkEvents
WSAEventSelect	WSPEventSelect
WSAGetOverlappedResult	WSPGetOverlappedResult
Getpeername	WSPGetPeerName
Getsockname	WSPGetSockName
Getsockopt	WSPGetSockOpt
WSAGetQOSByName	WSPGetQOSByName
WSAIoctl	WSPIoctl
WSAJoinLeaf	WSPJoinLeaf
Listen	WSPListen
WSARecv(recv 也可间接映射成 WSPRecv)	WSPRecv
WSARecvDisconnect	WSPRecvDisconnect
WSARecvFrom(recvfrom 也可间接映射成 WSPRecvFrom)	WSPRecvFrom

续表

API 函数	对应的 SPI 函数
Select	WSPSelect
WSASend(send 也可间接映射成 WSPSend)	WSPSend
WSASendDisconnect	WSPSendDisconnect
WSASendTo(sendto 也可间接映射成 WSPSendto)	WSPSendTo
Setsockopt	WSPSetSockOpt
Sbutdown	WSPSbutdown
WSASocket(socket 也可间接映射成 WSPSocket)	WSPSocket
WSAStringToAddress	WSPStringToAddress

大多数情况下，当一个应用程序调用 Winsock 函数时，WS2_32.DLL 就会调用一个对应的 Winsock SPI 函数并使用专门的服务提供程序来执行这个请求。例如，select 对应 WSPSelect，WSAConnect 对应 WSPConnect，WSAAccept 对应 WSPAccept。然而，并非所有的 Winsock 函数都有对应的 SPI 函数。下面的清单详细介绍了这些例外的函数。

- 支持函数如 htonl、htons、ntohl 和 ntohs 等在 WS2_32.DLL 中执行，不传递给下层的服务提供程序。这些函数的 WSA 版本也如此。
- IP 转换函数如 inet_addr 和 inet_ntoa 仅在 WS2_32.DLL 内执行。
- 所有 IP 专用名称转换和解析函数(如 WSAGetXbyY 函数)，以及 WSACancelAsyncRequest 和 gethostname 函数，均在 WS2_32.DLL 中执行。
- Winsock 编录函数和阻塞挂钩相关函数在 WS2_32.DLL 内执行。因此，WSAEnumProtocols、WSAIsBlocking、WSASetBlockingHook 和 WSAUnhookBlockingHook 没有相应的 SPI 函数。
- Winsock 错误代码在 WS2_32.DLL 内管理，因此 WSAGetLastError 和 WSASetLastError 未和服务提供程序对应。
- 事件对象操作和等待函数——包括 WSACreateEvent、WSACloseEvent、WSASetEvent、WSAResetEvent 和 WSAWaitForMultipleEvents——被直接映射为本地 Windows 操作系统的调用，在服务提供程序中不存在。

同时，补充材料中的 LSP 编录下有一个 LSP 示例。该 LSP 是一个传递通过的 LSP，不修改任何 Winsock API 调用，仅将调用传递到下面的层。贯穿整个分层提供程序的讨论都将引用同样的示例代码来阐述各个论点。

在具体讨论 LSP 的安装及执行前，先讨论错误处理。Winsock 应用程序经常使用

WSAGetLastError，有时用 WSASetLastError。然而，正如前面已经提到，这些函数没有对等的 SPI。相反，从参数方面看，一个 LSP 必须执行的每个 SPI 函数，都是它们的 API 等价物的确切镜像，惟一的例外是 SPI 函数多了一个额外的 lpErrno 参数。除了 lpErrno 之外，那些可以用一种重叠方式调用的 API 还有一个额外的参数，即调用线程的线程 ID(在“处理 I/O”部分叙述)。这是指向一个整数的指针，该整数应设置为正确的错误代码，以便在 LSP 函数失败时使用。要指示失败，LSP 函数应返回 SOCKET_ERROR，并设置 lpErrno。如果成功，将返回 NO_ERROR，lpErrno 值会被忽略掉。惟一的例外是 WSPStartup，它要么返回 NO_ERROR，要么返回导致启动失败的实际错误代码。

12.1.1　安装 LSP

在谈论 LSP 的实施前，第 1 步是将分层提供程序安装到 Winsock 编录中，过程将非常复杂。在第 2 章可以看到应用程序如何列举 Winsock 编录，并有一个代码示例来说明具体过程。安装 LSP 时先要安装一个 WSAPROTOCOL_INFOW 结构，该结构定义了分层提供程序的特征，以及 LSP 如何适应于“链”。由“分层服务提供程序”体现的字面意思看，提供程序是分层的，依次相互重叠，形成一个协议链，这个协议链的的定义为：

```
typedef struct _WSAPROTOCOLCHAIN {
    int ChainLen;
    DWORD ChainEntries[MAX_PROTOCOL CHAIN];
}WSAPROTOCOLCHAIN, FAR *LPWSAPROTOCOLCHAIN;
```

因为 ChainLen 字段指明条目的提供程序类型，所以它很重要。表 12.3 列出了可能的值。若 ChainLen 为 0 或 1 时，ChainEntries 数组中的数据就没有意义。ChainLen 值为 1 时表示一个基础提供程序，如微软 TCP 及 UDP 提供程序。通常，基础提供程序都有一个核心模式协议驱动程序和它相关联。例如，微软 TCP 及 UDP 提供程序需要 TCP/IP 的驱动程序 TCPIP.SYS 来最终实现其功能。还可以自行开发基础提供程序，不过这超出了本书讨论范围。关于基础提供程序的更多内容，请参考 Windows DDK(Driver Development Kit，驱动程序开发包)。

表 12.3　链长和提供程序类型

ChainLen 值	说　明
0	分层提供程序条目
1	基础提供程序
2 或更大	分层链条目

分层提供程序使用值为 0 或大于 1 的链长。那些链长为 0 的条目比较特殊。安装分层提供程序后，必须构造协议链来描述分层提供程序的位置。将链中每个协议的编录 ID 填进 ChainEntries 数组，即可达到此目的。编录 ID 就是 WSAPROTOCOL_INFOW 结构中所包含的 dwCatalogEntryId 字段。

这里来看看一个简单的例子。假定要开发一个在基础的微软 TCP 提供程序之上的 LSP，这将要求安装一个单个的提供程序，其 ChainLen 应为 2。ChainEntries 数组将包含两个条目：第 1 个是分层提供程序的编录 ID，第 2 个是微软 TCP 提供程序的编录 ID。现在的问题是，分层提供程序的编录 ID 该使用哪个值？构造描述 LSP 分层链的 WSAPROTOCOL_INFOW 结构时，dwCatalogEntryId 字段并未初始化，但不能简单地拼凑一个。只有通过 WSCInstallProvider 安装了一个提供程序之后，才会进行编录 ID 的分配。为了解决这个问题，首先安装一个虚提供程序条目，其 ChainLen 为 0。安装了这个虚提供程序后(也是分层提供程序)，系统就会分配编录 ID，用它就可以安装真实的分层链条目。

虚提供程序的 WSAPROTOCOL_INFOW 结构包含的数据基本上没有意义(除了通往提供程序 DDL 的路径之外，稍后将讨论)。而且，调用 WSAEnumProtocols 的应用程序不会获得任何链长为 0 的条目，只有调用 WSCEnumProtocols 时才会返回这些条目(和所有其他条目一起)。编写服务提供程序的安装(及删除)代码时，应该使用 WSCEnumProtocols，否则将看不到分层提供程序的虚条目，只能看到基础及分层链条目。

回到 LSP 示例上来。首先安装好虚 LSP 条目，之后列举编录，以便找到虚条目的提供程序 ID。然后建立描述分层链的 WSAPROTOCOL_INFOW 结构。在这个结构中 ChainLen 为 2，ChainEntries 包含两个值。第 1 个值是刚安装的虚条目的编录条目 ID，第 2 个数组索引包含基础 TCP 提供程序的编录条目 ID。图 12.3 展示了 3 个 WSAPROTOCOL_INFOW 结构。左边的结构是默认的微软 TCP 提供程序，中间的结构是虚 LSP 条目，右边的结构是 LSP 提供程序的分层链条目。应注意到，LSP 提供程序的协议链包含两个条目。还要注意，本图只展示了前 4 个协议链条目，而实际上 WSAPROTOCOL_INFOW 结构包含 MAX_PROTOCOL_CHAIN 个条目(7 个)。

.szProtocol = "MSAFD Tcpip [TCP/IP]"	.szProtocol = "My LSP"	.szProtocol = "My LSP over [TCP/IP]"
.iAddressFamily = AF_INET	.iAddressFamily = AF_INET	.iAddressFamily = AF_INET
.iSocketType = SOCK_STREAM	.iSocketType = SOCK_STREAM	.iSocketType = SOCK_STREAM
.iProtocol = IPPROTO_TCP	.iProtocol = IPPROTO_TCP	.iProtocol = IPPROTO_TCP
.dwCatalogEntryID = 1001	.dwCatalogEntryID = 1017	.dwCatalogEntryID = 1018
.ProtocolChain .ChainLen = 1 [0]	.ProtocolChain .ChainLen = 0	.ProtocolChain .ChainLen = 2 [1017] [1001]

图 12.3 基础微软 TCP 提供程序之上的 LSP 示例

12.1.1.1 安装提供程序条目

在讲述完基础知识后，现在来看看用于安装 Winsock 提供程序的 API，即 WSCInstallProvider。

这个 API 的定义为：

```
int WSPAPI
WSCInstallProvider(
    IN LPGUID lpProviderId,
    IN const WCHAR FAR *lpszProviderDllPath,
    IN const LPWSAPROTOCOL_INFOW lpProtocolInfoList,
    IN DWORD dwNumberOfEntries,
    OUT LPINT lpErrno
);
```

第 1 件要注意的事，就是这个 API 仅有一个 UNICODE 版本。参数列表也几乎不言自明。每个安装好的提供程序都需要一个 GUID 来惟一辨识该提供程序条目。使用命令行实用程序 UUIDGEN.EXE 或编程使用 UuidCreate，都可以生成一个 GUID。虚分层提供程序条目需要一个 GUID，分层链条目或多个分层链条目也需要一个(或多个)GUID。lpszProviderDllPath 参数是一个 UNICODE 字符串，包含了执行分层提供程序的 DDL 的访问路径，这个 DDL 路径可以容纳诸如%SYSTEMROOT%之类的环境变量。这个提供程序的路径对分层提供程序条目和分层链条目都应该是正确无误的。最后应注意，只有“管理员”(Administrators)组的成员才可以安装(及删除)Winsock 编录条目。

lpProtocolInfoList 参数是 WSAPROTOCOL_INFOW 结构的一个数组。数组中的每个条目都是将要安装的独立的提供程序条目。dwNumberOfEntries 参数指明数组中的条目数量。如果正安装的提供程序位于多个提供程序之上，这些程序可以全部一次性安装，或一次安装一个——稍后会发现，这是一个需要考虑的问题。当然，虚分层提供程序条目必须第 1 个自行安装，以便获得一个分层协议条目使用的编录 ID 条目。最后一个参数在失败的时候返回错误代码，此时 API 将返回 SOCKET_ERROR。

前面已经提到，分层提供程序条目没有任何意义，安装它只是为了获得一个编录 ID。对分层协议条目而言，WSAPROTOCOL_INFOW 结构通常是从将要分层的提供程序那里拷贝过来的，但有两点例外。首先，szProtocol 字段被修改，用来容纳新提供程序的名称。其次，如果 dwServiceFlags1 字段中有 XP1_IFS_HANDLES 标志，且这个标志将被删掉。如果设置了这个标志，表明这个提供程序返回的套接字句柄是真正的操作系统句柄，能被可交换地传递到并非专门接受套接字句柄(如 ReadFile 和 WriteFile)的 API，而不会导致性能打折扣。要想一个分层提供程序返回 IFS 句柄，必须得有一个相关联的核心模式组件，该组件能创建这些句柄，就像 TCPIP.SYS 对微软 TCP 及 UDP 提供程序所起的作用一样。本章稍后将更为详细地讨论套接字句柄。

当然，如果开发中的 LSP 是一个全新的协议，安装应用程序就必须得在 WSAPROTOCOL_INFOW 结构中设置正确的标志和字段，以便精确描述提供程序表现出来的行为。对协议结构的完整叙述请参见第 2 章。

最后，安装新的提供程序条目时，在列举的时候这些条目会默认地出现在 Winsock 编录的末端。因为系统总是会将套接字创建调用匹配到列举时出现在该 LSP 条目前面的 MSAFD TCP/IP 提供程序，

所以如果 LSP 模拟了 TCP/IP 提供程序，在默认情况下它将永远不会被调用(关于系统如何找到恰当的 Winsock 提供程序来加载，更多内容参见第 2 章)。因此，可能有必要对编录重新排序，以便新安装的 LSP 条目能够首先出现。可以通过 APIWSCWriteProviderOrder 来完成，该函数的定义为：

```
int WSPAPI WSCWriteProviderOrder(
    IN LPDWORD lpdwCatalogEntryId,
    iN DWORD dwNumberOfEntries
);
```

第 1 个参数是 DWORD 的一个数组，包含编录中每个提供程序的编录条目 ID，这些 ID 按照写入顺序排列。例如，如果 Winsock 编录(从 WSCEnumProtocols 返回的)中有 20 个条目，则该数组就包含 20 个条目，每个条目是一个已有提供程序的编录 ID。在 API 被调用之后，编录将被按照指定的顺序重新排序。同时这个数组不得包含任何重复的条目。应该注意到，这个 API 是在头文件 SPORDER.H 及库 SPORDER.LIB 中定义的。在近期发布的的 Platform SDK 中，这个函数的定义被移到 WS2_32.LIB 中，而 SPORDER.LIB 仅包含该定义的一个转发。

在成功安装 LSP 后，最好重启计算机。许多系统服务，如 LSASS(Local Security Authentication Server System，本地安全认证服务器系统)，它们的大多数套接字是在启动时创建的，其余套接字不需要在启动时创建。问题是，LSP 在这些服务使用的提供程序之上安装好之后，这些服务就会拥有一套来自多个提供程序的混合套接字。在这些应用程序使用 select API 时，上述情况可能会导致诸多问题。

现在总结一下。一个提供程序的安装要求首先安装链长为 0 的“虚”分层提供程序条目。要获得一个分层链条日将来可以引用的编录 ID，这是必不可少的步骤。安装好分层提供程序后，接着安装每个分层链，这些分层链将引用分层虚提供程序的编录 ID，作为它们的第 1 个链条目。之后的链条目是在这个虚提供程序之下的提供程序的编录 ID。然后，在大多数情况下，Winsock 编录都需要重新排序，以使大多数应用程序可以在 LSP 中调用，而不是在基础提供程序中调用。

应注意到，在 64 位 Windows 上操作 Winsock 编录时，应特别小心。为了使 32 位应用程序能够在 64 位 Windows 上运行，保留了两个独立的 Winsock 编录——一个为 32 位应用程序而保留，另一个为 64 位本地应用程序保留。为了操作这个编录，引入了几个新的 WSC 函数。针对每个 WSC 函数有一个新的 API，函数名后附有字符串“32”。但请注意，参数仍然一样。例如，函数 WSCInstallProvider 有一个对应的 WSCInstallProvider32 函数。“常规”函数(也就是结尾不是 32 的函数)在安装应用程序编译时的目标平台中的 Winsock 编录上运行。也就是说，如果 LSP 安装查询是为 64 位 Windows 编译的，则 WSCInstallProvider 函数会将 LSP 安装到 64 位编录中。类似地，如果是为 32 位 Windows 编译的，则将 LSP 安装到 32 位编录中。本地 64 位应用程序可以用以 32 结尾的新函数来操作 32 位编录。

惟一的问题是，如果需要将 LSP 安装到 32 位编录和 64 位编录中，应该如何进行？构造协议链的时候，协议链中就会出现虚提供程序的同一个编录 ID。为了解决这个问题，可以使用另外一个版本的安装函数，即 WSCInstallProvider64_32。这个函数将提供程序安装到两种编录中，以便将同一个编

录 ID 分配给 32 位条目和 64 位条目。还要注意，当安装到两个编录时，LSP 的两个版本的 DDL 应该出现。本地 64 位编译版本在%SYSTEMROOT%\system32 处，而%SYSTEMROOT%\syswow64 处则要放上 32 位编译版本。应注意到，一旦安装到两个编录中后，删除 LSP 就需要两个独立的调用——也就是说，不存在一个在两个编录上都能运行的等价卸载例程。

最后，Winsock 编录函数(WSC 的变种)也有一个以 32 结尾的新版本，遵循和上面讨论的一样的规则。如果一个本地 64 位应用程序需要列举 32 位编录和 64 位编录，则它必须调用 WSCEnumProtocols 来获得 64 位编录，然后调用 WSCEnumProtocols32 来获得 32 位编录。但是没有办法让 32 位应用程序获得 64 位编录，而且所有 32 位应用程序都这样(也就是说，32 位应用程序无法操作 64 位编录)。

12.1.1.2　删除 LSP

将 LSP 安装到 Winsock 编录之后，大多数情况下，删除提供程序是一件很容易的事情。函数 WSCDeinstallProvider 将删掉所有和给定 GUID 相关联的编录条目。这个 API 的定义为：

```
int WSPAPI WSCDeinstallProvider(
    IN LPGUID lpProviderId,
    OUT LPINT lpErrno
);
```

在一般情况下，LSP 需要两个 GUID：一个用于虚条目，另一个用于所有分层链条目。这种情况下要完全删除 LSP，应对每个 GUID 调用一次 WSCDeinstallProvider。当然，如果分层链提供程序是使用多个 GUID 安装的，则卸载代码就必须对每个 GUID 都要调用一次 WSCDeinstallProvider。

如果在安装了 LSP 后，这个 LSP 之上又安装了另外一个 LSP，则卸载原先的 LSP 就会变得极其复杂。第 2 个 LSP 的链条目将包含对第 1 个 LSP 的编录 ID 的引用。如果第一个 LSP 是盲目卸载的，则第 2 个 LSP 就会被破坏。如果第 2 个 LSP 作为 TCP/IP 及 UDP/IP 提供程序使用，则系统极有可能无法启动，或者导致无法访问网络。

在这种情况下，LSP 的卸载代码必须在引用自己编录 ID 的其他 Winsock 条目的协议链内进行检查。如果发现有其他条目引用自己的编录 ID，则卸载程序必须拷贝这些条目，然后卸载它们，并将自己的 LSP 条目从它的协议链内删除，最后再将拷贝的条目安装回编录中去。例如，考虑表 12.4 所示的情况。这个例子中有两个 LSP。LSP1 安装在基础 TCP/IP 和 UDP/IP 提供程序之上，LSP2 安装在 LSP1 的基础 TCP/IP 和 UDP/IP 提供程序之上。要删除 LSP1，同时也要修改 LSP2 的条目，以使得它们不再引用这两个编录 ID 即 1010 和 1011。要完成这个工作，LSP1 的卸载程序必须找到所有引用 LSP1 编录 ID 的条目，在这个例子中是条目 1021 和 1022。应先保存这些条目，然后再把它们卸载掉。之后应修改保存的条目，使得它们的链长是 2，并且它们的协议链中任何对 1010 或 1011 的引用都要删除。最后，这些条目应该安装到与前面相同的那个 GUID 之下。1021 的条目应该有一个 1020 的协议链，后面紧跟 1001；1022 的条目应该是 1020 和 1002。应注意到，在重新安装提供程序后，列举的条目将在编录末端出现。因此有必要重新对编录进行排序，将 LSP2 的条目放回原来的位置。

表 12.4 有多个 LSP 的 Winsock 编录

编录 ID	说 明	地址族/协议	链 长	协议链
1021	LSP2	UDP/IP	3	1020—>1010—>1001
1022	LSP2	UDP/IP	3	1020—>1011—>1002
1020	虚 LSP2	N/A	0	N/A
1010	LSP1	TCP/IP	2	1009—>1001
1011	LSP1	UDP/IP	2	1009—>1002
1009	虚 LSP1	N/A	0	N/A
1001	MSAFD TCP/IP	TCP/IP	1	N/A
1002	MSAFDUDP/IP	UDP/IP	1	N/A

12.1.1.3 修改 LSP 条目

我们已经看到，卸载一个已安装了其他提供程序的提供程序是一件极其可怕的事情——尤其是这种情况，该提供程序之上的那个提供程序同时还安装了许多其他的提供程序，而且它们都是用同一个 GUID 来安装的。为了解决这个问题，Windows XP 引入了一个新的 API，就是为了缓解上述卸载提供程序的困难而设计的。这个 API 允许在不卸载一个已有的提供程序的前提下对它进行修改。这个函数是 WSCUpdateProvider，它的定义为：

```
int WSPAPI WSCUpdateProvider(
    IN LPGUID lpProviderId,
    IN const WCHAR FAR *lpszProviderDllPath,
    IN const LPWSAPROTOCOL_INFOW lpProtocolInfoList,
    IN DWORD dwNumberOfEntries,
    OUT LPINT lpErrno
);
```

参数表和 WSCInstallProvider 的相同，不过这个函数不在编录中安装新的提供程序，而是更新 GUID 所引用的提供程序。用这个 API 可以更新一个已有提供程序的条目中的任何字段，但提供程序 ID(GUID)除外。因此对于表 12.4 中给出的例子，由于需要修改的只有协议链和链长，所以这个 API 使得 LSP2 的修改轻而易举。

12.1.2 编写分层提供程序

前面已提到，LSP 是在 DLL 中实现的。分层提供程序有 3 个重要部分：WSPStartup 函数、跟踪套接字句柄及处理各种 I/O 模型(如 select、WSAEventSelect、WSAAsyncSelect、重叠及完成端口等)。

当然，这不包括实现 LSP 提供的功能(如 HTTP 代理及 SSL 套接字)。

除了这3个重要的任务之外，还有处理微软特有 Winsock 扩展的问题，如 AcceptEx 和 TransmitFile。这个主题将在 3 个主要任务之后介绍。最后，LSP 还必须执行几个小任务以获得百分之百的兼容性。最后几个小节将详细讨论这些任务。

12.1.2.1　初始化提供程序

每个 LSP 都必须实现并输出 WSPStartup 函数。这个函数的原型为：

```
int WSPAPI WSPStartup(
    WORD wVersion,
    LPWSPDATA lpWSPData,
    LPWSAPROTOCOL_INFOW lpProtocolInfo,
    WSPUPCALLTABLE UpCallTable,
    LPWSPPROC_TABLE lpProcTable
);
```

第 1 个参数是应用程序所请求的 Winsock 版本。lpWSPData 参数是一个 WSPDATA 结构，这个结构是第 1 章中讲述的 WSADATA 结构的一个子集。LSP 必须填充这个结构，指明所支持的 Winsock 版本。lpProtocolINFO 结构是一个和正被加载的提供程序对应的 WSAPROTOCOL_INFOW 结构。对于一个 LSP 而言，这是它的其中一个协议结构。UpCallTable 参数是一个输入参数，包含指向各种 Winsock 支持例程的函数指针，Winsock 支持例程稍后将作讨论。最后，lpProcTable 参数是一个函数指针的结构，这些函数指针指向那些 LSP 执行的 Winsock 提供程序函数，在函数返回之前 LSP 必须完全填充这个结构。

在具体讨论初始化分层服务提供程序前，来看看当系统调用一个 LSP 的 WSPStartup 函数的时候，会有什么事情发生。首先，当应用程序调用 WSAStartup 时，系统不做任何动作。直到应用程序实际创建一个套接字时，提供程序的 WSPStartup 才被调用。应用程序创建套接字时，系统会在 Winsock 编录中搜索匹配的条目，如第 2 章所述。找到匹配条目之后，系统就会加载提供程序的 DLL，并调用提供程序的 WSPStartup 函数。

第 2 个问题是，您希望 LSP 的 WSPStartup 被调用多少次？这取决于 LSP 是如何安装的。例如，考虑一个具有两个分层协议条目的系统，如图 12.4 所示。图表的左边是两个分层协议条目：一个在 MSAFD TCP/IP 提供程序之上，另一个在 MSAFD IPX 提供程序之上。应注意到，因为两个条目都包含同一个提供程序的 GUID，所以它们都是在对 WSCInstallProvider 的同一个调用中安装的。如果应用程序创建一个 TCP/IP 套接字，此刻系统就会加载 LSP，并调用它的 WSPStartup 函数。之后，如果应用程序创建一个 IPX 套接字，因为 WSPStartup 函数已经被具有 GUID A 的提供程序而调用了，所以系统就不会再次调用这个函数了。此外，任何其他 TCP/IP 和 IPX 套接字的创建都不会激活对 WSPStartup 的再次调用。

然而，如果 TCP/IP 及 IPX 的分层协议条目是用对 WSCInstallProvider 的两次调用(故而有两个不同的 GUID)各自独立安装的，当应用程序创建第 1 个 TCP/IP 套接字时，LSP 的 WSPStartup 就会被调用。而当应用程序创建 IPX 套接字时，因为对应 GUID B 的提供程序还未被初始化，所以系统会再次调用 WSPStartup 函数。

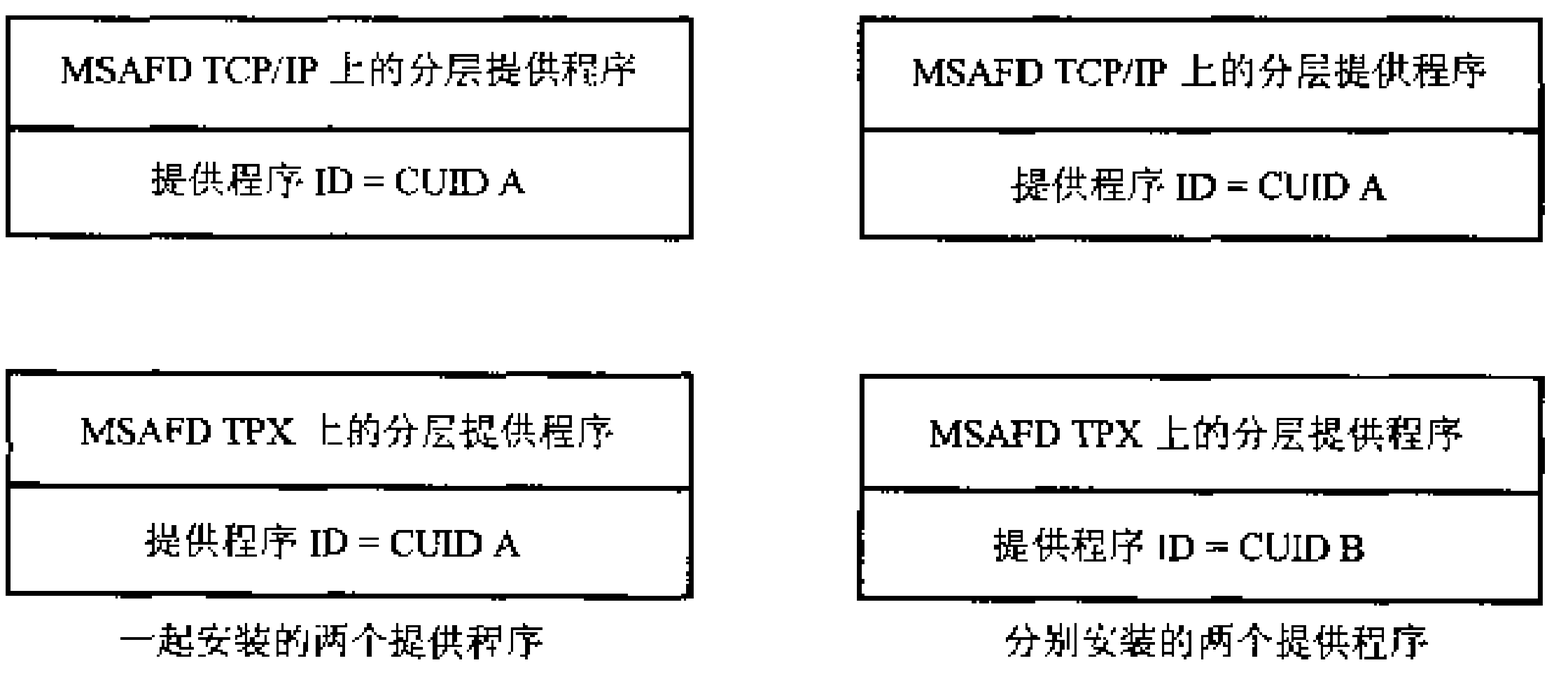


图 12.4 WSPStartup 调用行为

这是一个值得考虑的重要事项，原因是它指明了需要多少开销用于初始化工作。例如，考虑一个在所有已安装的协议之上的 LSP——例如一个内容筛选器。在这种情况下，可能会存在多种协议，且每种协议可能有多个提供程序条目。如果 LSP 的所有分层协议条目都是用同一个 GUID 一次性安装的，则当应用程序创建套接字时，LSP 的 WSPStartup 只能被调用一次。然而，大多数应用程序都倾向于从对应 2 个或 3 个提供程序条目的单个地址族创建套接字(例如，IPv4 有 3 个条目：TCP/IP、UDP/IP 和 RAW/IP)。在这种情况下，LSP 可以要求为每个提供程序建立多个内部的数据结构：IPv4、IPv6、IPX/SPX、NetBIOS、AppleTalk 及 IrDA。然而，如果 LSP 的分层协议条目是分组安装的，且各个组分别对应不同的地址族，则可以在调用应用程序决定使用某个特定的协议时，才分配必要的内部数据结构。用一个还是多个 GUID 安装，完全由 LSP 实施者决定，但是，知道 WSPStartup 被调用的频度却很有好处。

现在回到对初始化步骤的讨论，下面是 LSP 在它的 WSPStartup 函数中必须执行的 3 个任务：

1. 追踪 WSPStartup 已被调用的次数。
2. 初始化 lpWSPData 和 lpProcTable 参数。
3. 找到它自己在协议链中的位置，并初始化下面的层。

第 1 项要求很简单。LSP 应该记住 WSPStartup 已被调用的次数，以使每次 WSPStartup 被调用的时候，便让一个简单的参考计数增加 1。提供程序可能需要初始化一些内部数据结构，大多数情况下会发生在对 WSPStartup 的第 1 次调用中。同样，LSP 必须实现 WSPCleanup，此时参考计数的值将减少 1。一旦参考计数到达 0，任何内部数据结构及所有其他动态分配的资源都应该释放。

第 2 项要求也很简单。LSP 必须初始化 WSPDATA 和 WSPPROC_TABLE 参数。对于 WSPDATA

结构而，LSP 可以亲自验证 Winsock 版本是否正确，如果它不想验证这些参数，也可以在初始化下层提供程序(下一个要讨论的内容)时将 lpWSPData 参数传递到下层提供程序的 WSPStartup 函数中。WSPPROC_TABLE 是一个巨大的结构，包含所有在 LSP 中实现的函数的函数指针。分层服务提供程序必须初始化所有这些指针。

第 3 个要求稍微复杂一点。分层提供程序需要“加载”协议链中在它下面出现的提供程序。如果 LSP 位于多个下层提供程序之上，就得加载在 LSP 的每个提供程序条目之下的提供程序。LSP 如何找到自己在链中的位置呢？传递到 WSPStartup 的 lpProtocolInfo 参数是 LSP 的协议链条目其中一个的提供程序条目。前面已经讲过，这个条目将根据应用程序第 1 次创建的套接字的类型，匹配 LSP 的提供程序中的一个。

我们提到，系统传递了分层提供程序的一个 WSAPROTOCOL_INFOW 结构，该分层提供程序与应用程序根据 LSP 的一个分层协议条目所创建的套接相字对应。协议链数组中的第 1 个条目是 LSP 的编录 ID。给定这个值之后，Winsock 编录就可以被列举，且余下的分层链提供程序都能被找到。每个 LSP 条目的链数组的第 2 个索引包含需要加载的下层提供程序的编录 ID。

加载下层提供程序是一件简单的事情。对每一个下层提供程序，可调用 WSCGetProviderPath 函数来获得实现这个提供程序的 DDL 的位置。这个函数的原型为：

```
int WSCGetProviderPath(
    LPGUID lpProviderId,
    LPWSTR lpszProviderDllPath,
    LPINT lpProviderDllPathLen,
    LPINT lpErrno
);
```

在获得提供程序的 DDL 路径之后，在这个路径上调用 LoadLibrary，接着为该 DDL 的 WSPStartup 函数调用 GetProcAddress。初始化下层时只是简单的调用该 DDL 的 WSPStartup 即可。传递到 LSP 的 WSPStartup 中的 lpWSPData 可以被传递给下层提供程序的 WSPStartup 调用，以便该调用证实所要求的版本是正确的。

注意　在 Windows 95、Windows 98 及 Windows Me 中，所返回的提供程序的 DDL 路径总是为一个 UNICODE 字符串，因此必须将它转换为 ANSI，并必须使用加载库 A(LoadLibraryA)。

每个通过调用其 WSPStartup 来完成初始化的下层提供程序，都必须遵循应用到 LSP 的 WSPStartup 的规则。我们必须为该提供程序的条目传入一个 WSPDATA 及 WSAPROTOCOL_INFOW 结构，不管它是基础提供程序，还是其他分层提供程序。例如，如果下层提供程序是 MSAFD TCP/IP 提供程序，则传递的 WSAPROTOCOL_INFOW 结构是 MSAFD TCP/IP 提供程序，而非 LSP 的分层协议条目。每个下层提供程序应该用一列函数指针对传递给它的 WSPPROC_TABLE(函数表)进行初始

化。LSP 必须保存该函数表。如果应用程序在 LSP 中作了一个 Winsock 调用，则这个 LSP 最终必须调用下层提供程序相应的 Winsock 函数(除非 LSP 的目的是抑制或禁止该动作)。

在补充材料提供的示例 LSP 中，下述结构被分配给组成 LSP 的每个分层链条目中：

```
typedef struct _PROVIDER {
    WSAPROTOCOL_INFOW      NextProvider,
                               LayeredProvider;
    WSPPROC_TABLE              NextProcTable;
    EXT_WSPPROC_TABLE          NextProcTableExt;
    WCHAR                      ProviderPathW[MAX_PATH],
                               LibraryPathW[MAX_PATH];
    char                       ProviderPathA[MAX_PATH],
                               LibraryPathA[MAX_PATH];
    Int                        ProviderPathLen;
    HINSTANCE                  hProvider;
    LPWSPSTARTUP               lpWSPStartup;
    SOCK_INFO                  *SocketList;
}PROVIDER;
```

这个结构跟踪分层链条目的协议结构(LayeredProvider)以及下层提供程序的协议结构(NextProvider)。下层提供程序的过程表存储在 NextProcTable 字段中，而 NextProcTableExt 字段则是下层提供程序公开的微软特有 Winsock 函数的结构，是我们自己的。稍后将详细讨论这些条目。另外，提供程序路径(ProviderPath)的 UNICODE 和 ANSI 版本都被保存了。LibraryPath 字段包含和 ProviderPath 字段中大体相同的数据，不同的是它里面的系统变量通过 ExpandEnvironmentStrings API 做过扩充。hProvider 字段保存 LoadLibrary 返回的句柄，lpWSPStartup 字段包含该 DDL 的 WSPStartup 函数的地址。最后的 SocketList 字段是该提供程序创建的所有套接字的一个链表。这个字段将在以后变得重要。

再进一步深入讨论之前，先总结一下初始化 LSP 的步骤：

1. 确认要求的 Winsock 版本。
2. 增加参考计数。
3. 保存传入的 WSPUPCALLTABLE。
4. 找到位于该 LSP 的提供程序之下的 Winsock 提供程序(注意，如果该 LSP 的各个分层协议条目是在各自的 GUID 下安装的，则结果有可能是一个子集)。
5. 给每一个分层条目分配一个 PROVIDER 结构。
6. 加载每个下层提供程序的 DDL，并调用它们的 WSPStartup。
7. 保存从每个下层提供程序返回的 WSPPROC_TABLE。

如果这些步骤的其中一个失败，则 LSP 的 WSPStartup 将返回一个错误。有几个和初始化相关的 Winsock 错误代码，它们是：

- WSAEINVALIDPROCTABLE：表示下层提供程序返回了一个无效的程序表(例如，一个或多个条目为 NULL)。
- WSAEPROVIDERFAILEDINIT：表示 LSP 遇到一个妨碍它进行初始化的错误。

当然，如果遇到的错误类型有更为具体的 Winsock 错误代码，该错误代码就应当被使用。例如，如果在启动过程中，LSP 动态分配内存时失败了，就应返回 WSAENOBUFS。

12.1.2.2　创建套接字

分层服务提供程序的下一个重要任务是创建套接字句柄。到目前为止，已经讨论的事件顺序如下。首先，应用程序创建一个套接字，其参数和 LSP 的一个条目匹配。接着，系统通过调用 LSP 的 WSPStartup 加载 LSP，并获得 LSP 的函数调遣表，再调用 LSP 的 WSPSocket 函数创建一个套接字，并将这个套接字返回给应用程序。因为这里讨论的是分层提供程序，而非传输提供程序，因此 LSP 无法创建真正的操作系统句柄。“真正的”套接字句柄通过调用下层提供程序的 WSPSocket 函数而获得。记住，在用 LSP 的 WSPStartup 加载下层提供程序时，我们就获得了下层提供程序的 WSPPROC_TABLE。

在 LSP 的 WSPSocket 函数内部，必须验证地址族、套接字类型以及协议参数，并弄清楚该使用哪个下层提供程序——假设 LSP 安装在多个条目之上。记住，对于一些传输协议而言，套接字创建时 API 调用的协议参数可以是 0。例如，如果 LSP 位于 MSAFD TCP/IP 和 MSAFD UDP/IP 之上，并且应用程序作出了下述套接字调用：s = socket(AF_INET,SOCK_STREAM,0)；则 LSP 的函数调用也会采取同样的参数。随后 LSP 必须确定该套接字参数和 LSP 的位于 MSAFD TCP/IP 之上的条目匹配。之后必须找到对实现 MSAFD TCP/IP 的 DDL 所作的的启动调用返回的函数表，此时可以调用 LSP 的 WSPSocket。另外，调用应用程序可能会传入 WSAPROTOCOL_INFO 结构，这个结构将属于该 LSP。在根据下层提供程序创建套接字之前，必须将它的 WSAPROTOCOL_INFO 结构替换掉。

一旦根据下层提供程序创建好套接字，就有两个可选项。第 1 个是仅仅返回该套接字句柄。这样将产生的问题是，无法修改或监视在这个套接字上收发的数据。下一节将详细讨论个中原由。第 2 个可选项是返回一个“虚”句柄。之后 LSP 会将该虚句柄和下层提供程序返回的句柄关联起来。现在，无论应用程序何时用这个虚句柄调用 Winsock API，该句柄都会被传递进 LSP，此时 LSP 将找到和这个虚句柄关联的下层提供程序的句柄，并用下层提供程序句柄调用该下层提供程序的同一个 Winsock API。

这些虚句柄通过调用 WPUCreateSocketHandle 来创建。应注意到，LSP 不能直接调用这个 API。传递给 LSP 的 WSPStartup 例程的 WSPUPCALLTABLE 中，提供了指向这个 API 的函数指针。这个 API 的原型为：

```
SOCKET WPUCreateSocketHandle(
    DWORD dwCatalogEntryId,
    DWORD_PTR dwContext,
```

```
    LPINT lpErrno
);
```

第 1 个参数是 LSP 的分层协议链的编录 ID。第 2 个参数是希望用来和返回的套接字句柄相关联的任意数据块。最后一个参数表示这个 API 调用失败时将返回的错误代码。

通常，LSP 根据下层提供程序创建一个套接字，然后为该套接字分配一个包含上下文信息的数据结构。在示例 LSP 中使用了下述上下文结构：

```
typedef struct _SOCK_INFO
{
    SOCKET ProviderSocket;          //下层提供程序套接字句柄
    SOCKET LayeredSocket;           //应用程序的套接字句柄
    DWORD dwOutstandingAsync;       //未完成的异步操作计数
    BOOL bClosing;                  //应用程序关闭套接字了吗?
    volatile LONG RefCount;         //参考计数
    DWORD BytesSent;                //字节计数
    DWORD BytesRecv;
    HANDLE hIocp;                   //是否和一个 IOCP 关联?
    HWND  hWnd;                     //和套接字关联的窗口(如果有)
    UINT   uMsg;                    //套接字事件的消息
    CRITICAL_SECTION SockCritSec;//保护对这个对象的访问
    struct _PROVIDER *Provider;  //它属于哪个提供程序?
    struct _SOCK_INFO*prev,      //用来将这些结构链接到一起
                     *next;
}SOCK_INFO;
```

这是一大堆信息，不过对一个健壮的 LSP 来说是很有必要的。第 1 个字段是下层提供程序返回的套接字句柄，第 2 个字段是从 WPUCreateSocketHandle 返回的句柄。在调用 WPUCreateSocketHandle 时，我们传递一个 SOCK_INFO 结构的地址作为上下文信息。其余字段有一大部分处理套接字 I/O，这点将在下一节讨论。

现在可以构造 LSP 的 WSPSocket API 的基本框架了。它看起来应该比较像下面的示例：

```
SOCKET WSPAPI WSPSocket(
    int                             af,
    int                             type,
    int                          protocol,
    LPWSAPROTOCOL_INFOW          lpProtocolInfo,
    GROUP                           g,
    DWORD                           dwFlags,
    LPINT                           lpErrno)
{
    PROVIDER   *Provider = NULL;
    SOCK_INFO  *Context;
    SOCKET      ProviderSocket,
```

```
                LayeredSocket;

//首先验证参数

//找到和给定参数——该参数被设为 Provider——匹配的分层协议条目的 PROVIDER 结构
//用下层提供程序的 lpProtocolInfo(如果已经提供)来替换这里的 lpProtocolInfo

ProviderSocket = Provider->NextProcTable.lpWSPSocket(
    af,
    type,
    protocol,
    lpProtocolInfo,
    g,
    dwFlags,
    pErrno
    );
if (ProviderSocket != INVALID_SOCKET){
    Context = AllocateContext();   //分配一个 SOCK_INFO 结构

    LayeredSocket = MainUpCallTable.lpWPUCreateSocketHandle(
        Provider->LayeredProvider.ProtocolChain.ChainEntries[0],
        (DWORD_PTR)Context,
        lpErrno
        );
    if (LayeredSocket == INVALID_SOCKET){
        //句柄失效
    }
    Context->LayeredSocket = LayeredSocket;
    Context->ProviderSocket = ProviderSocket;
}
return LayeredSocket;
}
```

在 SOCK_INFO 结构中有两个重要的字段应该讲一下。bClosing 字段指明应用程序是否已经在虚套接字句柄上调用了 WSPCloseSocket。如果有未完成的 I/O 操作，则 LSP 绝对不能释放套接字的上下文信息，直到所有 I/O 都完成(大多数情况下都出错)为止。同时，要保持一个参考计数(使用 RefCount)，这样，如果执行调用的应用程序是多线程的，其中一个线程在使用套接字，而另一个线程关闭套接字，LSP 就不会释放第 1 个线程下面的 SOCK_INFO 结构，也就不会导致访问冲突。

LSP 必须实现表 12.2 列出的每个 SPI 函数。对于那些不创建套接字句柄(如除 WSPSocket、WSPAccept 和 WSPJoinLeaf 之外的函数)，但将套接字句柄用作一个参数的函数，有必要将应用程序的套接字句柄转换为下层句柄。记住，SOCK_INFO 结构是和每个虚套接字句柄关联的。上下文信息可以通过调用 WPUQuerySocketHandleContext 获得。这个函数在 WSPUPCALLTABLE 结构中可以找

到，它的定义为：

```
int WSPAPI WPUQuerySocketHandleContext(
    SOCKET s,
    LPDWORD_PTR lpContext,
    LPINT lpErrno
);
```

例如，来看看 LSP 可能会如何实现 WSPGetSockOpt 函数：

```
int WSPAPI WSPGetSockOpt(
    SOCKET s,
    int level,
    int optname,
    char FAR *optval,
    LPINT optlen,
    LPINT lpErrno
)
{
    SOCK_INFO *lpContext = NULL;
    int rc = NO_ERROR;

    rc = MainUpCallTable.lpWPUQuerySocketHandleContext(
        s,
        (LPDWORD_PTR)&lpContext,
        &err
    );
    if (rc == SOCKET_ERROR){
      *lpErrno = WSAENOTSOCK;
    }
    else
    {
        rc = lpContext->Provider->NextProcTable.lpWSPGetSockOpt(
            lpContext->ProviderSocket,
            level,
            optname,
            optval,
            optlen,
            &lpErrno
            );
    }
    return rc;
}
```

这个示例先查询上下文信息。如果找不到，就返回错误 WSAENOTSOCK。否则就用正确的套接字句柄简单地调用下层提供程序的 WSPGetSockOpt 函数。应注意到，在真正的实现过程中，当套接

字上下文被查出之后，参考计数应该增加 1，且在从 SPI 函数返回之前，参考计数应该减少 1。在这个示例 LSP 中，定义了两个助手例程来完成这一任务，即下面列出的 FindAndLockSocketContext 和 UnlockSocketContext：

```
SOCK_INFO *FindAndLockSocketContext(SOCKET s,int *lpErrno)
{
    SOCK_INFO  *SocketContext = NULL;
    Int          ret;

    EnterCriticalSection(&gCriticalSection);
    ret = MainUpCallTable.lpWPUQuerySocketHandleContext(
        s,
        (PDWORD_PTR)&SocketContext,
        lpErrno
        );
    if (ret == SOCKET_ERROR)
    {
       SocketContext = NULL;
    }
    else
    {
        InterlockedIncrement(&SocketContext->RefCount);
    }
    LeaveCriticalSection(&gCriticalSection);
    return SocketContext;
}

void UnlockSocketContext(SOCK_INFO *context)
{
    EnterCriticalSection(&gCriticalSection);
    InterlockedDecrement(&context->RefCount);
    LeaveCriticalSection(&gCriticalSection);
}
```

在这个代码示例中，gCriticalSection 是整个 DLL 的一个全局 CRITICAL_SECTION 对象。在一个 SPI 函数内，在为需要查询套接字上下文的 WSP 函数或支持函数使用任何套接字之前，通过调用 FindAndLockSocketContext，可以确保当其他操作正在进行过程中时，一个多线程应用程序能够关闭套接字但却不会导致访问冲突。当然，确保在从 SPI 函数返回前对 UnlockSocketContext 进行相应的调用，也是很重要的。

套接字创建的最后一个要考虑的事项，是在应用程序关闭套接字句柄时发生的。首先，查询所提供的句柄的套接字上下文。应注意到，这个动作将导致套接字参考计数增加 1(这意味着，如果参考计数比 1 大，就表示另一个线程正在访问该结构)。下一步是关闭下层提供程序的套接字句柄。这是必

要的，以使得任何未完成的 I/O 操作可以完成，并返回恰当的错误(将在下一节讨论)。

然而，如果套接字上下文未指明存在未完成的异步 I/O(通过上下文信息的 dwOutstandingAsync 字段指明)，或指明参考计数大于 0，则此时还不能关闭虚套接字句柄。相反，应该将套接字上下文结构标记为关闭(将 bClosing 设为 TRUE 即可)。如果不这样做，那么如果应用程序创建了另外一个套接字后，句柄值就可能会被重新使用，这可能导致微小的难以发觉的问题。例如，考虑这样一个情况，两个线程正访问一个句柄值为 0x300 的套接字。如果一个线程关闭套接字，第 2 个线程却正准备访问套接字，而套接字已被关闭，上下文信息被删除。这时第 3 个线程创建一个新的套接字，并给它分配了句柄值 0x300。正准备访问套接字的线程现在查找上下文，最后返回新套接字的上下文。此刻，新套接字可能正处于一种错误状态(例如，该连接的时候没有连接)，甚至可能是错误协议的一个套接字。无论使用这个套接字的是何种 API 都极有可能失败，返回一个令人出乎意料的错误代码，如 WSAENOTCONN 或 WSAEOPNOTSUPP。

只有在参考计数指明没有其他线程访问套接字上下文信息，且未完成操作计数是 0 的时候，才能关闭虚套接字，并释放上下文信息。在异步 I/O 完成之后，以及当参考计数递减时，应该检查套接字上下文的 bClosing 字段。如果它是 TRUE，则表明应用程序已经关闭套接字，这时需要关闭虚句柄，如果这样做安全的话。

如果已经确认关闭套接字是安全的，就可以通过 WPUCloseSocketHandle API 来完成，这是另一个包含在 WSPUPCALLTABLE 结构中的函数。这个函数的定义为：

```
int WSPAPI WPUCloseSocketHandle(
    IN SOCKET s,
    OUT LPINT lpErrno
);
```

最后要记住，函数 WSPAccept 和 WSPJoinLeaf 也返回套接字句柄，而且刚才为 WSPSocket 叙述的相同步骤也适用于它们。一旦一个新的套接字句柄从下层提供程序返回，一个应用程序套接字就会通过 WPUCreateSocketHandle 被创建出来，上下文信息也将和它相关联，之后这个应用程序套接字会被返回给调用者。然而，某些情况下，WSPJoinLeaf SPI 函数并不创建新的套接字(稍后详述)。

12.1.2.3 处理 I/O

创建 LSP 的最后一个主要任务，是处理应用程序可能在一个套接字上启动的各种类型的 I/O。应该记得，在第 5 章中，有大量的应用程序可用的 I/O 模型：阻塞套接字、select、WSAAsyncSelect、WSAEventSelect、重叠 I/O 及完成端口。前面已经提过，如果 LSP 希望修改或监视收发的数据，它必须用 WPUCreateSocketHandle 创建自己的套接字句柄，必须处理在套接字上可能发生的所有类型的 I/O 模型。本节我们来看看每种类型的 I/O 模型，并探讨让它们运行起来的必需步骤。

在具体探讨每种 I/O 模型之前，有必要提提一些适用于所有类型 I/O 的普遍规则。首先，如果 LSP 需要修改发送缓冲区，它不应该修改应用程序缓冲区内的数据，而应该制作一个自己的副本。另外，

LSP 不能和已启动的 I/O 的类型背道而驰。如果应用程序已经将一个套接字放进非阻塞模型，则在处理那些通常以返回 WSAEWOULDBLOCK 而失败的操作时，LSP 不应阻塞。

1. 阻塞与非阻塞

对于最基本的 I/O 阻塞与非阻塞套接字而言，真的没有太多要说的。对于那些收发数据的 SPI 函数，LSP 需要做的，仅仅是将套接字句柄转换为提供程序的套接字句柄，并调用下层提供程序的函数。例如，一个 LSP 的 WSPSend 函数看起来应该和下述代码类似：

```
int WSPAPI WSPSend(
    SOCKET                s,
    LPWSABUF              lpBuffers,
    DWORD                 dwBufferCount,
    LPDWORD               lpNumberOfBytesSent,
    DWORD                 dwFlags,
    LPWSAOVERLAPPED lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine,
    LPWSATHREADID   lpThreadId,
    LPINT                 lpErrno
   )
{
    SOCK_INFO *SocketContext = NULL;
    int          ret;

    //获取上下文信息
    SocketContext = FindAndLockSocketContext(s,lpErrno);

    if (lpOverlapped == NULL) //确保这不是重叠的
{
    SetBlockingProvider(SocketContext->Provider);
    ret = SocketContext->Provider->NextProcTable.lpWSPSend(
          SocketContext->ProviderSocket,
          lpBuffers,
          dwBufferCount,
          lpNumberOfBytesSent,
          dwFlags,
          NULL,
          NULL,
          lpThreadId,
          lpErrno
          );
    SetBlockingProvider(NULL);
}
UnlockSocketContext(SocketContext);
```

```
    return ret;
}
```

这是一个非常直接的过程：找到套接字上下文，并调用下层提供程序。为了和 16 位 Winsock 兼容，必须留意阻塞的是哪个提供程序，以防应用程序调用 WSACancelBlockingCall；这就是 SetBlockingProvider 函数所做的工作。接着保存 PROVIDER 结构的地址，该结构为该线程启动一个阻塞调用。应用程序调用 WSACancelBlockingCall 时，必须做的事情，就是调用下面阻塞层的 WSPCancelBlockingCall。SetBlockingProvider 例程使用线程本地存储器，来保存指向一个 PROVIDER 的指针，该 PROVIDER 当前正启动一个阻塞调用。

2. Select 和 WSPSelect

如果应用程序使用 select API 在一组套接字上等待事件，事情就会变得复杂一点。select API 将映射到 SPI 函数 WSPSelect，而且在将调用传递到下层提供程序之前，还有些工作需要做。有 3 个 FD_SET 结构将被传入，这些结构引用分层套接字，而不是引用下层提供程序的套接字。所以，必须获得 FD_SET 中包含的每个套接字的上下文，且必须建立一个新的 FD_SET 来容纳下层提供程序的套接字。

下述代码展示了如何转换传递到 WSPSelect 的 fdread FD_SET。

```
int WSPAPI WSPSelect(
    int nfds,
    fd_set FAR *readfds,
    fd_set FAR *writefds,
    fd_set FAR *exceptfds,
    const struct timeval FAR *timeout,
    LPINT lpErrno)
{
    FD_SET ReadFds,WriteFds,ExceptFds,
    int ret,HandleCount,count;

    //迅速将 LSP 套接字映射到提供程序套接字的简单结构
    struct {
        SOCKET LayeredSocket;
        SOCKET ProviderSocket;
    }Read[FD_SETSIZE],Write[FD_SETSIZE],Except[FD_SETSIZE];

//将 LSP 句柄转换为提供程序句柄
if (readfds){
        FD_ZERO(&ReadFds);
        for(i = 0;i <readfds->fd_count ;i + + ){
            SocketContext = FindAndLockSocketContext(
                (Read[i].LayeredSocket = readfds->fd_array[i]),
                lpErrno
                );
```

```
            Read[i].ProviderSocket = SocketContext->ProviderSocket;
            FD_SET(Read[i].ProviderSocket,&ReadFds);
            UnlockSocketContext(SocketContext);
        }
    }
    //对 writefds 进行相同处理

    //对 exceptfds 进行相同处理
    SetBlockingProvider(SocketContext->Provider);
    ret = SocketContext->Provider->NextProcTable.lpWSPSelect(
        nfds,
         (readfds ?&ReadFds :NULL),
         (writefds ?&WriteFds :NULL),
         (exceptfds ?&ExceptFds :NULL),
         timeout,
         lpErrno
         );
    SetBlockingProvider(NULL);
    HandleCount = ret;
    //将已传信的提供程序句柄映射回 LSP 句柄
if (readfds){
        count = readfds->fd_count;
        FD_ZERO(&readfds);
        for(i = 0;(i <count)&&HandleCount ;i + + ){
        5 if (MainUpCallTable.lpWPUFDIsSet(
                Read[i].ProviderSocket,
                &ReadFds)){
                FD_SET(Read[i].LayeredSocket,readfds);
                HandleCount--;
            }
        }
        //对 writefds 进行相同处理

        //对 exceptfds 进行相同处理
}
```

为了让这段代码运行，在传递到 select 的分层套接字和与它对应的提供程序套接字之间，始终维持着一种映射。因为在下层提供程序的 WSPSelect 被调用之后，LSP 必须只返回那些被传信的分层提供程序，所以这是很有必要的。

第 1 步是遍览 FD_SET 中的每个句柄，并找到它们的上下文信息。提供程序套接字到分层套接字的映射保留在 Read 数组中。然后我们将拥有第 2 个 FD_SET，即 ReadFds，它包含下层提供程序的套接字句柄，之后我们将把这些句柄传递到下层提供程序的 WSPSelect 函数中。只要这个函数一返回，

根据返回值，我们就会知道有多少句柄被传信了。然后就是查看被传信的是哪些已传递的提供程序句柄。要达到这一目的，对传入的每个提供程序套接字均调用助手函数 WPUFDIsSet 即可。在设置套接字之后，获取关联的分层套接字，并将它设置到已传递给该函数的 readfds FD_SET 中，以使 LSP 的 WSPSelect 一返回，应用程序就可以传信正确的句柄。对 writefds 和 exceptfds 也要进行同样的处理。当然，这个示例不执行错误处理，也不处理提供了一个超时值的情况。完整的实现过程参见 LSP 示例。

WPUFDIsSet 函数是另外一个在 WSPUPCALLTABLE 结构中传到 LSP 的助手函数。该函数的定义为：

```
int WSPAPI WPUFDIsSet(
    IN SOCKET s,
    IN fd_set FAR *fdset
    );
```

这个函数和第 5 章的 FD_ISSET 宏表现一致。

在实现一个 LSP 的 WSPSelect 时，经常会遇到的一个主要的问题是：如果传递到 FD_SET 的事件句柄其中一个未知，怎么办？如果 LSP 查询一个给定句柄的套接字上下文，但失败了，LSP 应该指明一个错误呢(如 WSAENOTSOCK)，还是简单地将该事件句柄不经修改地传到下层提供程序？WSPSelect API 惟一的问题是，它是惟一能够在一个单一调用中，接受多个套接字句柄的 Winsock 函数。对于其他 Winsock 函数，因为只有一个套接字句柄作为参数传递，所以系统明白哪个提供程序将处理该 API 调用。

尽管 Winsock 规范已经清楚地说明，只有来自同一个提供程序的套接字才可以被传递到 select，然而，许多应用程序(包括微软的 Internet Explorer)都不理会这一点，时常将 TCP 和 UDP 句柄一起下传。基础的微软提供程序不会验证是否所有套接字句柄都来自同一个提供程序。另外，微软提供程序会在单个的 select 调用中，正确地处理来自多个提供程序的套接字。因为某个 LSP 可能刚好被放置在单个的条目之上，如 TCP，所以这对于多个 LSP 来说是个问题。这种情况下，该 LSP 的 WSPSelect 被 FD_SET 调用，FD_SET 包含它们自己的套接字，以及来自其他提供程序的套接字。如果该 LSP 在转换套接字句柄的时候，不经意遭遇 UDP 句柄，则上下文查询将会失败。此刻，它可能会返回一个错误(WSAENOTSOCK)，或不经修改地将套接字传到下层。如果返回错误，且 LSP 仅仅位于 UDP/IPv4 (或 TCP/IPv4)之上，则 Internet Explorer 将不再运行。一个折衷方法是，总是将 LSP 安装到某个给定地址族的所有提供程序之上(如对于 IPv4，则安装在 TCP/IPv4、UDP/IPv4 和 RAW/IPv4 之上)。目前，尽管 Windows NT 4.0 上的 LSASS 被用来将 IPX 和 IPv4 套接字放一起传递(这已经在 Windows NT 4.0 最新的服务包中改正)，但还没有微软应用程序或服务将来自多个地址族的套接字句柄传递到单个的 select 调用中。

3. WSAAsyncSelect

通过 WSAAsyncSelect 来处理注册为事件通知的套接字，也需要额外的帮助。回想一下第 5 章中，

应用程序注册某些将被投递到给定窗口的事件的通知。这里的问题是，投递到应用程序窗口的 WPARAM 参数包含套接字句柄。这很糟糕，原因是 LSP 将转换传递到 WSPAsyncSelect 中的句柄，并利用转换后的套接字及其余参数调用下层提供程序的函数。结果，如果一个事件被投递，它将被直接投递到应用程序的窗口处理程序，且 WPARAM 参数将包含下层提供程序的套接字，而非 LSP 创建的套接字。

为了正确处理这种情况，LSP 必须创建自己的隐藏窗口，以便在窗口中接收来自下层提供程序的事件。然后通过 LSP 的窗口处理程序中，套接字就可以被转换回应用程序套接字，并被投递到应用程序的窗口处理程序。LSP 的 WSPAsyncSelect 必须保存窗口句柄，以及与应用程序套接字关联的信息。该信息被保存在套接字上下文中(例如，示例 LSP 的 SOCK_INFO 结构)。下述代码展示了如何处理这种情况：

```
int WSPAPI WSPAsyncSelect (
    SOCKET     s,
    HWND       hWnd,
    unsigned int   wMsg,
    long         lEvent,
    LPINT        lpErrno)
{
    SOCK_INFO *SocketContext;
    int ret;

    SocketContext = FindAndLockSocketContext(s,lpErrno);
    if (SocketContext != NULL)
    {
        SocketContext->hWnd = hWnd;
        SocketContext->uMsg = wMsg;

        //获取隐藏窗口的句柄
        if ((hWorkerWindow = GetWorkerWindow())!= NULL)
        {
            SetBlockingProvider(SocketContext->Provider);
            ret = SocketContext->Provider->NextProcTable.lpWSPAsyncSelect(
                    SocketContext->ProviderSocket,
                    hWorkerWindow,
                    WM_SOCKET,
                    lEvent,
                    lpErrno);
            SetBlockingProvider(NULL);
        }
    }
    UnlockSocketContext(SocketContext);
    return ret;
```

```
}
```

在这段代码中，套接字上下文被找到，窗口句柄和信息也被保存起来。然后 LSP 创建的隐藏异步窗口通过 GetWorkerWindow 调用被返回。这个例程简单地创建了窗口和线程来处理事件(完整的实现过程参见 ASYNCSELECT.CPP)。之后下层提供程序被下层提供程序套接字来调用，但窗口句柄是由我们的 LSP 助手窗口提供的。

隐藏窗口处理程序的代码很简单：

```
static LRESULT CALLBACK AsyncWndProc(
    HWND hWnd,
    UINT uMsg,
    WPARAM wParam,
    LPARAM lParam)
{
    SOCK_INFO *si;

    if (uMsg == WM_SOCKET)
    {
        if (si = GetCallerSocket(NULL,wParam))
        {
            MainUpCallTable.lpWPUPostMessage(
                si->hWnd,
                si->uMsg,
                si->LayeredSocket,
                lParam);
            return 0;
        }
    }
    return DefWindowProc(hWnd,uMsg,wParam,lParam);
}
```

这段代码只寻找套接字通知。惟一的难题是将提供程序套接字(表示为 wParam)映射回 LSP 创建的套接字，这由 GetCallerSocket 函数(在 SOCKINFO.CPP 中定义的)来完成。回想一下，PROVIDER 结构包含每个提供程序创建的所有 SOCK_INFO 的一个链表。GetCallerSocket 函数搜索所有的链表，以便寻找包含给定下层提供程序套接字句柄的 SOCK_INFO。由于没有其他方便的途径，可以将提供程序套接字映射回 LSP 套接字，所以，这是做必要的。

一旦找到 SOCK_INFO 之后，就要调用助手函数，来将事件投递到应用程序的窗口，该窗口具有正确的套接字句柄。记住，窗口句柄和消息是应用程序在句柄上调用 WSAAsyncSelect 函数之前就被保存好了的。这个函数位于 WSPUPCALLTABLE 中，它的定义为：

```
BOOL WSPAPI WPUPostMessage(
    IN HWND hWnd,
```

```
IN UINT Msg,
IN WPARAM wParam,
IN LPARAM lParam
);
```

4. WSAEventSelect

这个套接字模型不要求 LSP 做任何事情。当选择事件被传信后，会使用一个简单的事件句柄——没有套接字句柄返回，因此不需要额外的套接字转换。例如，应用程序用一个套接字、事件及事件屏蔽调用 WSAEventSelect。在 LSP 的 WSPEventSelect 内部，句柄被转换并跟随同一个事件及事件屏蔽传递到下层提供程序。如果在套接字上发生一个被请求的事件，下层提供程序就会将应用程序的事件设为已传信，之后应用程序就会调用一个发送或接收函数，如前面“阻塞和非阻塞”一节所述。

除非要求 LSP 截取这些事件通知(由 LSP 的实际目的决定)，否则没有必要替换我们自己的事件句柄来等待下层的通知。如果这样做了，一旦被替换的事件被传信，LSP 就会实施必要的计算，并用 WSASetEvent 传信应用程序的事件(这个事件必须保存在套接字上下文中)。

5. 重叠 I/O

取决于 LSP 所安装到的平台，处理重叠 I/O 是另外一个复杂的问题。最容易、最优雅的方法，是使用一个 I/O 完成端口来处理所有由应用程序初始化的重叠 I/O，而且不管应用程序是否使用了事件、回调或完成端口。然而，如果 LSP 准备在 Windows 95、Windows 98 或 Windows Me 上运行，就不可能这样做。这两种情况 LSP 提供程序示例都进行了处理。

对于 Windows NT 及 I/O 完成端口，LSP 会创建一个完成端口及一个工作器线程，该工作器线程通过调用 GetQueuedCompletionStatus 来为完成通知服务。如果应用程序作出一个重叠 I/O 调用，LSP 首先将检查一下，看看下层提供程序句柄是否已经和 LSP 的完成端口关联起来。这条信息作为 hIocp 字段包含在套接字上下文信息中。如果下层提供程序套接字已经被关联，这个字段就是非 NULL，否则，这个字段会包含 LSP 的完成端口的句柄。

一旦提供程序套接字和 LSP 的完成端口是关联的，I/O 就会在下层提供程序套接字句柄上投递。完成投递后，LSP 工作器线程将接收这个通知，LSP 就可以完成应用程序的请求了。在 LSP 收到完成通知后，应用程序的重叠 I/O 就会结束，以便应用程序可以通过事件、异步过程调用或它自己的完成端口收到通知。

每个能够以重叠方式使用的 Winsock SPI 函数(包括微软扩展函数)都需要特殊的处理。首先，LSP 必须留意应用程序传递给函数的 WSAOVERLAPPED 结构。这个结构保留了有用的信息，如指明进行中的 I/O、错误代码以及已传输的字节数。为了执行这个函数，LSP 定义了自己的 WSAOVERLAPPED 结构，来保留关于每个投递到下层提供程序套接字的重叠 I/O 操作的信息。这个结构的定义为：

```
typedef struct _WSAOVERLAPPEDPLUS
{
```

```
    WSAOVERLAPPED ProviderOverlapped;   //已传递到下层提供程序
    PROVIDER *Provider;                 //下层提供程序信息
    SOCK_INFO *SockInfo;                //这个操作的套接字信息
    SOCKET CallerSocket;                //应用程序(LSP)套接字
    SOCKET ProviderSocket;              //下层提供程序套接字
    HANDLE Iocp;                        //LSP 完成端口
    int Error;                          //错误代码?
    union                               //操作的参数
    {
        ACCEPTEXARGS AcceptExArgs;
        TRANSMITFILEARGS TransmitFileArgs;
        CONNECTEXARGS ConnectExArgs;
        TRANSMITPACKETSARGS TransmitPacketsArgs;
        DISCONNECTEXARGS DisconnectExArgs;
        WSARECVMSGARGS WSARecvMsgArgs;
        RECVARGS RecvArgs;
        RECVFROMARGS RecvFromArgs;
        SENDARGS SendArgs;
        SENDTOARGS SendToArgs;
        IOCTLARGS IoctlArgs;
    };
#define LSP_OP_IOCTL                1       //WSPIoctl
#define LSP_OP_RECV                 2       //WSPRecv
#define LSP_OP_RECVFROM             3       //WSPRecvFrom
#define LSP_OP_SEND                 4       //WSPSend
#define LSP_OP_SENDTO               5       //WSPSendTo
#define LSP_OP_TRANSMITFILE         6       //TransmitFile
#define LSP_OP_ACCEPTEX             7       //AcceptEx
#define LSP_OP_CONNECTEX            8       //ConnectEx
#define LSP_OP_DISCONNECTEX         9       //DisconnectEx
#define LSP_OP_TRANSMITPACKETS      10      //TransmitPackets
#define LSP_OP_WSARECVMSG           11      //WSARecvMsg
    int                     Operation;      //这个操作的类型
    LPWSATHREADID   lpCallerThreadId;       //调用者线程
    LPWSAOVERLAPPED lpCallerOverlapped;     //应用程序的 WSAOVERLAPPED 结构
    LPWSAOVERLAPPED_COMPLETION_ROUTINE      lpCallerCompletionRoutine;
    _WSAOVERLAPPEDPLUS          *next;
}WSAOVERLAPPEDPLUS,*LPWSAOVERLAPPEDPLUS;
```

可以看到，结构中为每个应用程序启动的重叠操作都保留了大量信息。因为很多字段的含义都是很明显的，所以这里不会对所有的字段都详细解释。先熟悉一下，在处理一个重叠调用时，需要做一些什么事情。步骤如下：

1. 分配一个 LSP 重叠上下文结构(例如，上一个清单里的 WSAOVERLAPPEDPLUS 对象)。
2. 保存 lpCallerOverlapped 结构中调用者的 WSAOVERLAPPED 指针。

3. 如果提供了完成例程，将其保存为 lpCallerCompletionRoutine。
4. 通过将 Internal 字段设置为 WSS_OPERATION_IN_PROGRESS(在 WS2SPI.H 中定义)，将调用者的 WSAOVERLAPPED 结构标记为待处理。
5. 确认下层提供程序套接字和 LSP 的完成端口相关联。
6. 在下层提供程序中，用下层提供程序套接字及 WSAOVERLAPPEDPLUS 结构中针对这个操作的 ProviderOverlapped 字段，调用同一个 SPI 函数。
7. 返回 SOCKET_ERROR，并将错误代码设为 WSA_IO_PENDING。

上面列出了所需的最少步骤。示例 LSP 有几个额外的步骤。首先，它保存了调用者的参数，如缓冲区指针及标志。这些被保存在 WSAOVERLAPPEDPLUS 结构中未命名的共用体中。针对每个支持重叠的 Winsock 函数，共用体都包含一个相应的结构——每个结构包含了对应于各自的 Winsock 函数参数表的字段。示例 LSP 没有使用保存的参数，不过对于一个实施特别任务的 LSP 而言，可能有必要使用。一个值得注意的重要事项是，有些提供的指针参数可能是基于堆栈的，从而导致 LSP 无法捕捉到指针值。例如，WSASend、WSASendTo、WSARecv 和 WSARecvFrom 获取一个结构数组，其中包含发送和接收缓冲。这个数组可能是基于堆栈的，这意味着应用程序一调用 Winsock 函数，该数组就可能从调用函数返回，或释放那段内存(如果是动态分配的)。因此，LSP 必须将缓冲区指针复制到自己已分配的 WSABUF 结构中。

在重叠 I/O 被投递到下层提供程序之后，剩下的事情，就是等待 LSP 的完成线程接收这个操作的通知了。完成线程接收通知时，从 GetQueuedCompletionStatus 返回的、指向这个操作的 WSAOVERLAPPED 结构的指针，实际上就是我们的 WSAOVERLAPPEDPLUS 结构。要完成这个操作，需执行下面 3 个步骤。

1. 调用下层提供程序的 WSPGetOverlappedResult 来获取已传输字节、标志及失败时的适当错误代码。
2. 设定 Offset 为错误(如果有发生)，OffsetHigh 为返回标志(如果有返回)，InternalHigh 为已传输字节，从而更新调用者的 WSAOVERLAPPED 结构。
3. 根据是否提供完成函数，确定使用 WPUQueueApc 或 WPUCompleteOverlappedRequest 来完成应用程序的重叠请求。

最后一步，是通知应用程序它的 I/O 操作已经完成了。如果提供了完成例程，LSP 需要运行这个例程。这由 WPUQueueApcWSPUCALLTABLE 函数来实施，这个函数是 WSPUCALLTABLE 结构的一个字段，它的定义为：

```
int WSPAPI WPUQueueApc(
    IN LPWSATHREADID lpThreadId,
    IN LPWSAUSERAPC lpfnUserApc,
    IN DWORD_PTR dwContext,
    OUT LPINT lpErrno
```

```
);
```

第 1 个参数是启动这个 I/O 的应用程序线程的线程 ID，因为完成例程必须在该线程的上下文内启动。回想一下会发现，这是在应用程序启动这个 I/O 时，WSAOVERLAPPEDPLUS 结构中保存的其中一个参数。第 2 个参数是要调用的应用程序的完成函数。参数 dwContext 是调用者的原始 WSAOVERLAPPED 结构，参数 lpErrno 在 WPUQueueApc 失败时返回一个错误。

如果应用程序不指定完成例程，仅提供一个 WSAOVERLAPPED 结构，则 LSP 将使用 WPUCompleteOverlappedRequest 来完成这个 I/O。有一点很让人感到奇怪，这个函数并非 WSPUPCALLTABLE 的成员。相反，它包含在 WS2_32.DLL 中，并按照常规方法被调用。该函数的定义为：

```
int WSPAPI WPUCompleteOverlappedRequest (
    SOCKET s,
    LPWSAOVERLAPPED lpOverlapped,
    DWORD dwError,
    DWORD cbTransferred,
    LPINT lpErrno
);
```

参数表很简单。SOCKET 参数是应用程序的套接字，lpOverlapped 参数是应用程序的 WSAOVERLAPPED 结构。dwError 参数是调用失败时返回的错误(若成功返回的则是 NO_ERROR)。cbTransferred 参数是操作中传递的字节数。lpErrno 参数在 WPUCompleteOverlappedRequest 调用失败时返回错误代码。

您会注意到，由 LSP 自动处理的重叠操作将失败，并返回 WSA_IO_PENDING，尽管当 LSP 向下层提供程序作出这个调用时，该重叠操作可能会立刻成功。因为不管操作是否立刻成功，通知始终都会被投递到完成等待队列中去，所以 LSP 没有这样作。如果始终在工作器线程中处理完成通知，代码除了会绝对符合 Winsock 规范之外，还会变得稍微简明一点。必须仔细确保针对每个 I/O 操作调用，应用程序只接收一个通知。在完成请求之前，提供的示例 LSP 始终返回“搁置”，并一直等待完成线程去接收通知。

在 Windows 95、Windows 98 及 Windows Me 上处理重叠 I/O 稍困难一点。有两种可能的途径。第 1 种，LSP 可以将重叠 I/O 发动到下层，并对完成通知使用事件。在第 5 章已经看到，这样作的弊端是，一个单个的线程只能等待 MAXIMUM_WAIT_OBJECTS 所表示的数量的事件句柄(目前是 64)。另外一种方法是使用完成函数，相对而言更容易实行。

如果作出调用的应用程序发动重叠请求，LSP 就会建立一个 WSAOVERLAPPEDPLUS 结构(如前所述)，之后这个对象被放到一个队列中。对于这个模型，仍旧使用工作器线程，其目的是等待将放到队列中的重叠请求。一旦工作器线程被通知有可用的工作项目，它就会从队列中删除一个项目，并实际地作出所请求的重叠操作(通过调用下层提供程序)。这些重叠操作是在 LSP 线程而非应用程序线

程的上下文中执行的，这点很重要。作出调用的线程必须处在一种警觉的等待状态，以便执行完成函数。由于作出调用的应用程序没有必要留意 Winsock 提供程序是否是分层的，因此它极有可能不会进入一种警觉的等待状态，除非应用程序使用完成函数(应用程序可以这样作)。结果，若 LSP 的工作器线程执行重叠请求的工作项目，且工作项目当时并非服务性的，该工作器线程就会始终维持一种警觉的等待状态。

应注意到，当 LSP 用完成函数发起重叠 I/O 时，提供的这个完成函数是 LSP 的，而不是应用程序的。一旦 LSP 的完成函数启动，LSP 便会使用应用程序提供的通知机制(如传信事件或启动完成函数)，将完成函数投递给应用程序。

12.1.2.4　Winsock 扩展函数

对于那些为自己创建套接字的 LSP，它们也必须处理 Winsock 专用扩展函数，这些函数将套接字句柄作为参数，包括 AcceptEx、TransmitFile、ConnectEx、DisconnectEx、TransmitPackets 和 WSARecvMsg 函数。这个任务是在 LSP 的 WSPIoctl 函数内完成的。当应用程序加载一个微软特有函数时，它将使用 I/O 控制命令代码 SIO_GET_EXTENSION_FUNCTION_POINTER 来调用 WSAIoctl。LSP 只须通过 InBuffer 参数(该参数包含所请求函数的 GUID)，便可确定加载了哪个函数。之后，LSP 便返回其扩展函数的地址。这个扩展函数接着会转换所有的套接字句柄，并加载下层将由 LSP 来调用的扩展函数。即使应用程序使用直接从 MSWSOCK.DLL 导出的 TransmitFile 和 AcceptEx 函数，上面所述仍行得通，因为只要用 SIO_GET_EXTENSION_FUNCTION_POINTER 调用 WSAIoctl，这些函数便可完成。

示例 LSP 将在 EXTENSION.CPP 中实现自己的扩展函数。这些函数的实现和其他 SPI 函数一样，LSP 必须转换句柄，验证参数，并处理可能的重叠 I/O。WSPIoctl 的代码包含在 SPI.CPP 中，应注意到，第 1 步要完成的，是检查一下，看看 I/O 控制命令代码是否为 SIO_GET_EXTENSION_FUNCTION_POINTER。

12.1.2.5　各种要求

本节叙述 LSP 为了正常运行而必须执行的各种任务。这一节将讨论每种服务提供程序 API，在每种 API 中，LSP 都必须实施一个特别的动作。

1. WSPGetSockOpt

当调用应用程序用 SO_PROTOCOL_INFOA 或 SO_PROTOCOL_INFOW 调用 LSP 的 WSPGetSockOpt 时，LSP 应该返回自己的协议信息结构，而不能转换句柄并传递到下层提供程序。如果真这样作了，调用就会返回下层提供程序的 WSAPROTOCOL_INFO 结构，而不是 LSP 的。应注意到，ANSI 和 UNICODE 都必须得到支持，以便 LSP 可以在返回的结构上执行适当的字符串转换。

2. WSPSetSockOpt

应用程序调用 AcceptEx 之后，它通常会用 SO_UPDATE_ACCEPT_CONTEXT 调用 setsockopt。

传递到 WSPSetSockOpt 的参数是接受套接字的套接字句柄。在将这个调用传递到下层提供程序之前，LSP 必须转换该句柄及监听套接字句柄。

3. WSPIoctl

有两个 I/O 控制命令代码 LSP 必须作出不同处理。前面已经提到，如果 LSP 在执行自己的扩展函数(如果返回自己的句柄，LSP 必须执行自己的扩展函数)时，它必须捕获 SIO_GET_EXTENSION_FUNCTION_POINTER。另外，它还必须捕获的命令是 SIO_QUERY_TARGET_PNP_HANDLE。WPUCreateSocketHandle 创建的句柄不是真正的即插即用句柄，因此不能接收通知。应用程序可以使用 SIO_QUERY_TARGET_PNP_HANDLE 来获得基础提供程序的套接字句柄。LSP 应该在返回缓冲区中返回下层提供程序的套接字句柄。

4. WSAJoinLeaf

WSAJoinLeaf 函数有点与众不同。取决于所用协议，返回值可能是一个新的套接字句柄(如在 ATM 中)，也可能是作为 s 参数传入的同一个句柄(如在 IP 多播中)。关于 WSAJoinLeaf 及其针对各种支持多播的协议的行为，参见第 9 章。目前，只有 IPv4 和 IPv6 两种协议在调用 WSAJoinLeaf 时不创建新的套接字句柄。如果准备将 LSP 放置于 IP 之上，应该考虑这一情况。否则，因为调用应用程序将只在其中一个句柄上调用 closesocket，所以如果真的创建了新的包括上下文信息的句柄，这些句柄就会被漏掉。

5. WSPAddressToString 和 WSPStringToAddress

这两个函数很独特，因为它们不接受套接字参数。相反，和给定地址匹配的 LSP 条目的 WSAPROTOCOL_INFOW 结构将被传入。LSP 会找到所提供的 WSAPROTOCOL_INFOW 结构下层的提供程序，并调用这个提供程序相应的 SPI 函数。惟一须遵守的规则是，如果下层提供程序不是一个基础提供程序，则不能对传递的 WSAPROTOCOL_INFOW 结构作任何修改。另一方面，如果下层提供程序是一个基础提供程序，LSP 就会替换这个基础提供程序的 WSAPROTOCOL_INFOW 结构。

12.1.3 调试 LSP

开发 LSP 是一个很复杂的任务，其中，只要出现一个错误，就极有可能使所有访问 Winsock 位于 LSP 下边的协议的应用程序崩溃。如果事情发生在 IP 上，则诸如 LSASS 之类的关键性服务都会停止。如果这种事情真的发生，须重启计算机，进入安全模式，并卸载 LSP，就可以将 Winsock 编录恢复回正常状态。另外，比较好的做法是在重启系统之前对 LSP 进行测试。Internet Explorer 一直是一个优秀的测试应用程序(当 LSP 位于 IP 之上时)。如果不用这种方式，就有必要编写一组测试应用程序来验证 LSP 的功能。

用应用程序跟踪小问题时，将调试信息输出到调试器非常有用。提供的示例 LSP 在几个地方使用了 OutputDebugString；通过定义 DEBUG 和 DEBUGSPEW，这个 LSP 还有能力为这个项目启动详细

的调试工作。为调试消息使用消息框是不大好的主意，因为在启动过程中，几个系统服务可能在用户接口子系统完全初始化之前加载 DDL，如果使用消息框，会导致 LSP DDL 在加载过程中失败。

对特别困难的问题，经常有必要使用调试器来确定失败点。对于交互式用户应用程序来说，Visual Studio 调试器及 NTSD(NT Symbolic Debugger，NT 符号调试器)——一个文本模式的调试器，Platform SDK 上有——都是很好的选择。在套接字上调用各种 API，来跟踪套接字创建的步骤，一般都能查到问题所在。在 NTSD 中，为每个正加载的 DDL 启动“加载断点”(break on load)功能(例如，NTSD 命令为 sxeld)，直到 LSP DDL 被加载为止。此刻，可以为应用程序感兴趣的 LSP 的函数(如 WSPStartup、WSPSocket 和 WSPConnect)设置断点。

如果在启动过程中诸如 LSASS 之类的系统服务出现问题，调试就会变得更困难一些。这需要将一个核心模式调试器附加到运行 LSP 的计算机上。然后就可以将 NTSD 附加到失败的系统服务上，再将 NTSD 控制台输出到在第 2 台计算机上运行的核心调试器。关于各种类型调试器的使用及设置，参见在 http://msdn.microsoft.com 上的联机 MSDN(Microsoft Developer Network，微软开发者网络)。

12.1.4　LSP 示例

本节整个讨论都引用了补充材料 LSP 编录下的示例 LSP。本节将简要叙述该项目的每个文件，以及如何安装 LSP。下面是文件清单及其实现的内容。

- ASYNCSELECT.CPP 实现用来处理 WSAAsyncSelect 的助手例程。包括为接收来自下层提供程序事件而创建隐藏窗口，以及创建为那些通知而服务的窗口程序。
- EXTENSION.CPP 实现所有可用的微软特有 Winsock 扩展，如 AcceptEx、TransmitFile、TransmitPackets、ConnectEx、DisconnectEx 和 WSARecvMsg 等。
- INSTLSP.CPP 实现安装及删除代码。这个文件要被编译为.EXE 文件，用来从 Winsock 编录安装和(或)删除 LSP。
- OVERLAP.CPP 为 LSP 实现处理重叠 I/O。对于 Windows NT 而言，这包括创建完成端口，以及为处理完成通知创建工作器线程。对于 Windows 95、Windows 98 和 Windows Me 而言，这包括建立一个工作项目队列，以及一个工作器线程，这个工作器线程将为队列中的 I/O 服务。
- PROVIDER.CPP 为 Winsock 编录列举以及 GUID(LSP 安装在 GUID 之下)定义实现公用例程。LSP DLL 和安装实用程序(INSTLSP.CPP)都使用了这些例程。
- SOCKINFO.CPP 实现公用例程，这些公用例程为 LSP 创建的套接字查找相关联的套接字上下文结构。这个文件中除了将 SOCK_INFO 结构插入到 PROVIDER 结构或从中删除的函数之外，还包含分配或释放 SOCK_INFO 结构的函数。
- SPI.CPP 这是 LSP 的“内脏”。它定义所有的 WSP*函数，包括 WSPStartup。

12.2 命名空间服务提供程序

第 8 章介绍了应用程序是怎样在命名空间内注册和解析服务的，这对网络上动态建的服务来说，是一个非常强人的特性。不幸的是，由于种种限制，现有的命名空间几乎不能发挥它们的重要作用。但是 Winsock 2 提供了一种方法，用于创建自己的命名空间。通过这个方法，可以采取任何一种自己喜欢的方式对名称注册和解析进行处理。

这是通过创建一个可实现 9 个命名空间函数的 DLL 来完成的。这些函数的前缀均是 NSP，与第 8 章中讲的 RNR 函数对应。比方说，等同于 WSASetService 的命名空间函数便是 NSPSetService。建立 DLL 之后，使利用标识这个命名空间的 GUID 把它安装到系统编录中。随后，应用程序可以在您的命名空间内注册和查询服务了。

Windows XP 增加了一个新的注册和名称解析函数：WSANSPIoctl。这个新的函数允许应用程序通过 WSALookupService API 初始化一次查找，然后使用一个对 WSANSPIoctl 的调用中返回的句柄，来接收关于该请求的信息。不要求命名空间提供程序实现自己的 NSPIoctl，但如果需要，它们也可以这样做。本节整个讨论将集中于必须实现的 9 个 NSP 函数，之后也会介绍 NSPIoctl。

本小节先介绍如何安装命名空间提供程序，然后对命名空间提供程序必须实现的函数进行说明。最后，除了注册和解析服务所用的一个示例应用程序外，还展示一个示例性的命名空间提供程序。

12.2.1 命名空间的安装

简单说来，命名空间只是一个 DLL，可以执行命名空间提供程序函数。在应用程序可使用命名空间之前，必须通过 WSCInstallNameSpace 函数，使系统知道有这个命名空间。反之，一旦安装了提供程序，就可以分别利用 WSAEnableNSProvider 和 WSAUnInstallNameSpace 这两个函数，取消这个提供程序或从系统编录中将它删除。接下来便是对这些函数的说明。

12.2.1.1 WSCInstallNameSpace

这个函数把一个提供程序安装到系统编录中。它的声明如下：

```
int WSCInstallNameSpace (
    LPWSTR lpszIdentifier,
    LPWSTR lpszPathName,
    DWORD dwNameSpace,
    DWORD dwVersion,
    LPGUID lpProviderId
);
```

首先应该注意到的是，所有字符串参数都是宽字符串。事实上，所有的命名空间提供程序都是用

宽字符串来实现的，这一点稍后将详细讨论。lpszIdentifier 参数是命名空间提供程序的名称。这个名称是在调用 WSAEnumNameSpaceProviders 时，被列举出来的(参见第 8 章)。lpszPathName 参数是 DLL 的位置。这个字符串中可包含%SystemRoot%之类的环境变量。dwNameSpace 参数是该命名空间的数字化标识符。例如，头文件 NSPAPI.H 定义了一些已知的命名空间，例如 IPX SAP 的 NS_SAP。dwVersion 参数则为该命名空间设置版本号。最后，lpProviderId 参数是一个标识该命名空间的 GUID。

如果调用成功，WSCInstallNameSpace 就返回 0；若失败，则返回 SOCKET_ERROR。最常见的失败是 WSAEINVAL，表示带有这个 GUID 标识的命名空间已经存在；如果是 WSAEACCESS，就表示调用进程不具备足够的权力。只有管理员(Administrator)组的用户才能安装命名空间。

12.2.1.2　WSCEnableNSProvider

这个函数用于修改命名空间提供程序的状态。可用于启用或取消提供程序。它的声明如下：

```
int WSCEnableNSProvider (
    LPGUID lpProviderId,
    BOOL fEnable
);
```

lpProviderId 参数是需要修改的命名空间的 GUID 标识符。fEnable 参数是一个布尔值，表示禁用或启用提供程序。已禁用的提供程序不能用于处理查询或注册服务。

如果调用成功，WSCEnableNSProvider 函数就会返回 0；若失败，便返回 SOCKET_ERROR。如果提供程序的 GUID 无效，就会返回 WSAEINVAL。

12.2.1.3　WSCUnInstallNameSpace

这个函数用于从编录中删除命名空间提供程序。它的定义如下：

```
int WSCUnInstallNameSpace (LPGUID lpProviderId );
```

lpProviderId 参数是将被删除的那个命名空间的 GUID。如果该 GUID 无效，函数调用就会失败，并返回 WSAEINVAL。

12.2.2　命名空间的实现

命名空间必须实现 9 个命名空间函数，这些函数与第 8 章中所讲的 RNR 函数相对应。除了实现这些函数外，还必须开发一种保持数据的方法。也就是说，还必须保持除了 DLL 实例之外的数据。每个加载 DLL 的进程都会接收自己的数据分段，这意味着保存在 DLL 内的数据(即固有数据)不能供其他实例共享(事实上，加载 DLL 的各个应用程序之间可以共享信息，不过实践中不提倡)。关于 DDL 的更多内容，参见 Jeffrey Richter 编著的《Programming Applications for Microsoft Windows》第 4 版(微软出版社，1999)。第 8 章提到了命名空间的 3 种类型：动态命名空间、固定命名空间和静态命名空间。显然，因为静态命名空间不允许对服务通过编程的方式进行注册，所以实现静态命名空间不可取。

稍后，我们将具体讲讲如何保持命名空间固有的数据。

另外，大家还必须知道这一点：在所有命名空间提供程序函数中，使用宽字符串是非常重要的。不仅要求函数中的字符串参数如此，而且要求 RNR 结构中的字符串也要如此，例如 WSAQUERYSET 和 WSASERVICECLASSINFO。大家也许有些迷惑不解：应用程序在注册或解析一个名称时，既可用 RNR 函数及结构的普通(ASCII)版本，又可以用它们的宽字符(UNICODE)版本，怎么可能呢？因为所有的 ASCII 调用都可通过一个中间层，该中间层将全部字符串统一转换成宽字符串，所以任何一个版本都可以。函数的调用和返回都如此，即如果 WSAQUERYSET 返回到调用应用程序(随同 WSALookupServiceNext)，则这个命名空间提供程序返回的全部数据起初都是 UNICODE 字符，但在函数调用返回之前，就被转换成了 ASCII 字符。换言之，如果应用程序使用的是 RNR 函数，那么调用它的宽字符版本就要快得多，因为无须进行转换。

在命名空间必须实现的这 9 个函数中，只有 7 个函数有相应的 Winsock 2 RNR 函数(参见表 12.5)。其余两个函数用于初始化和清除命名空间。系统中一旦安装了命名空间，应用程序通过指定用于安装的 GUID 或安装期间指定的命名空间标识符，便可以使用这个命名空间了。接下来，应用程序调用我们在第 8 章中讲到的标准 Winsock 2 RNR 函数。调用其中一个函数时，也会调用其相应的命名空间提供程序函数。例如，一个应用程序在调用 WSAInstallServiceClass(该函数引用了自定义命名空间的 GUID)时，也会调用那个提供程序的 NSPInstallServiceClass 函数。关于各个命名空间，我们将在下一小节一一说明。

表 12.5 RNR 函数与 NSP 函数之间的对应关系

Winsock 函数	等同(或对应)的命名空间提供程序函数
WSAInstallServiceClass	NSPInstallServiceClass
WSARemoveServiceClass	NSPRemoveServiceClass
WSAGetServiceClassInfo	NSPGetServiceClassInfo
WSASetService	NSPSetService
WSALookupServiceBegin	NSPLookupServiceBegin
WSALookupServiceNext	NSPLookupServiceNext
WSALookupServiceEnd	NSPLookupServiceEnd
WSANSPIoctl	NSPIoctl

12.2.2.1 NSPStartup

只要加载命名空间提供程序 DLL，NSPStartup 函数便会被调用。实现您的命名空间时必须包含

这个函数，而且必须将它从 DLL 中导出。为使命名空间提供程序正常工作，它所需的与 DLL 有关的数据结构都可在调用这个函数时得到分配。这个函数的原型定义如下：

```
int NSPStartup (
    LPGUID lpProviderId,
    LPNSP_ROUTINE lpnspRoutines
);
```

第 1 个参数是 lpProviderId，它是这个命名空间提供程序的 GUID。lpnspRoutines 参数是一个 NSP_ROUTINE 结构，这个结构是在实现这个函数时，必须填充的。这个结构提供了 8 个函数指针，分别指向提供程序的另外 8 个命名空间函数。NSP_ROUTINE 对象的定义如下：

```
typedef struct _NSP_ROUTINE
{
    DWORD                       cbSize;
    DWORD                       dwMajorVersion;
    DWORD                       dwMinorVersion;
    LPNSPCLEANUP                NSPCleanup;
    LPNSPLOOKUPSERVICEBEGIN     NSPLookupServiceBegin;
    LPNSPLOOKUPSERVICENEXT      NSPLookupServiceNext;
    LPNSPLOOKUPSERVICEEND       NSPLookupServiceEnd;
    LPNSPSETSERVICE              NSPSetService;
    LPNSPINSTALLSERVICECLASS     NSPInstallServiceClass;
    LPNSPREMOVESERVICECLASS     NSPRemoveServiceClass;
    LPNSPGETSERVICECLASSINFO     NSPGetServiceClassInfo;
    //NSPIoctl 是 Windows XP 中新加入的一个 API
    LPNSPIOCTL                   NSPIoctl;
}NSP_ROUTINE, FAR *LPNSP_ROUTINE;
```

这个结构的第 1 个字段 cbSize，表明 NSP_ROUTINE 结构的长度。下面两个字段，dwMajorVersion 和 dwMinorVersion，指示提供程序的版本信息，并不起其他作用。提供程序将其余条目设为它们各自的函数指针。例如，提供程序为 NSPSetService 字段分配了自己的 NSPSetService 函数地址(不管采用的是什么名称)。提供程序的函数名可以是任意的，但其参数和返回类型必须与提供程序的定义匹配。

实现 NSPStartup 时，惟一需要做的是填充 NSP_ROUTINE 结构。一旦提供程序完成填充和初始化例程，如果成功的话，函数就会返回 NO_ERROR。如果发生错误，就会返回 SOCKET_ERROR，并设置相应的 Winsock 错误代码。例如，如果一个提供程序尝试分配内存失败了，它就会将 WSAENOBUFS 作为参数，调用 WSASetLastError，然后再返回 SOCKET_ERROR。

现在，该好好讨论一下提供程序 DLL 中的错误处理了。为提供程序实现的所有函数调用若成功，便返回 NO_ERROR；若失败，便返回 SOCKET_ERROR。如果判断出调用失败了，则必须在返回之前，设置相应的 Winsock 错误代码。否则，任何试图利用您的命名空间进行的注册和查询服务的应用程序，都会报告 RNR 函数失败，但 WSAGetLastError 将返回 0。这样可能会为那些试图正常处理错

误的应用程序带来许多麻烦；因为 0 说明不了任何问题，所以它也不是我们希望返回的值。

12.2.2.2 NSPCleanup

卸载提供程序的 DLL 时，这个例程将被调用。有了这个函数，可以释放为 NSPStartup 例程分配的所有内存。这个例程的定义如下：

```
int NSPCleanup (LPGUID lpProviderId );
```

惟一的参数就是您的命名空间提供程序的 GUID。使用该函数作用不会清除任何动态分配的内存，事实上不需做任何事情。

12.2.2.3 NSPInstallServiceClass

NSPInstallServiceClass 函数的对应函数是 WSAInstallServiceClass，负责注册服务类。NSPInstallServiceClass 函数的定义如下：

```
int NSPInstallServiceClass (
    LPGUID lpProviderId,
    LPWSASERVICECLASSINFOW lpServiceClassInfo
);
```

第 1 个参数是提供程序的 GUID。lpServiceClassInfo 参数是正在注册的 WSASERVICECLASSINFOW 结构。提供程序必须保留一个服务类列表，确保在这个 WSASERVICECLASSINFOW 结构内，采用这个 GUID 的服务类只有一个。如果这个 GUID 已被采用，提供程序肯定会返回 WSAEALREADY。不然，提供程序就应该保留这个服务类，以便别的 RNR 操作能够引用它。

其余的大部分命名空间提供程序函数一般引用已安装的服务类。

12.2.2.4 NSPRemoveServiceClass

这个函数是对 NSPInstallServiceClass 函数的补充用来删除指定的服务类。这个命名空间函数的对应函数是 WSARemoveServiceClass。NSPRemoveServiceClass 函数的声明如下：

```
int NSPRemoveServiceClass (
    LPGUID lpProviderId,
    LPGUID lpServiceClassId
);
```

和上一个函数一样，第 1 个参数是提供程序的 GUID。第 2 个参数 lpServiceClassId，是即将被删除的服务类的 GUID。提供程序必须从存储设备上删除指定的服务类。如果没有找到 lpServiceClassId 指定的服务类，提供程序必然生成错误 WSATYPE_NOT_FOUND。

12.2.2.5 NSPGetServiceClassInfo

NSPGetServiceClassInfo 函数的对应函数是 WSAGetServiceClassInfo。它获取于一个和 GUID 关联

的 WSANAMESPACE_INFOW 结构。该函数的定义如下：

```
int NSPGetServiceClassInfo (
    LPGUID lpProviderId,
    LPDWORD lpdwBufSize,
    LPWSASERVICECLASSINFOW lpServiceClassInfo
);
```

和上一个函数一样，第 1 个参数仍然是提供程序的 GUID。lpdwBufSize 参数表明第 3 个参数 lpServiceClassInfo 中包含的字节数有多少。在输入端，第 3 个参数是一个 WSASERVICECLASSINFOW 结构，其中包含指定返回哪些服务类的搜索标准。这个结构中既可包含服务类名，又可包含即将返回的服务类的 GUID。如果提供程序找到了符合标准的服务类，肯定会在 lpServiceClassInfo 参数中返回 WSASERVICECLASSINFOW 结构，而且更新 lpdwBufSize 参数，从而指示返回了多少字节。

但是，如果没有找到与搜索标准相配的服务类，调用就会失败，并将 WSATYPE_FOUND 设置为错误。另外，如果找到了匹配的服务类，但提供的缓冲区太小，提供程序就应该更新 lpdwBufSize，指明需要多少缓冲区才足够，并把错误设为 WSAEFAULT。

12.2.2.6　NSPSetService

NSPSetService 函数的对应函数是 WSASetService，既可以用于注册服务，也可以用于从命名空间中删除服务。它的定义如下：

```
int NSPSetService (
   LPGUID lpProviderId,
   LPWSASERVICECLASSINFOW lpServiceClassInfo,
   LPWSAQUERYSETW lpqsRegInfo,
   WSAESETSERVICEOP essOperation,
   DWORD dwControlFlags
);
```

第 1 个参数是提供程序的 GUID。lpServiceClassInfo 参数是该服务所属的那个 WSASERVICECLASSINFOW 结构。lpqsRegInfo 参数是即将注册或删除(根据第 4 个参数 essOperation 中指定的操作来决定)的服务。最后一个参数是 dwControlFlags，指定标志 SERVICE_MULTIPLE，有了这个标志，便可修改指定的操作。可能的 essOperation 和 dwControlFlags 取值参见第 8 章表 8.4 及表 8.5 的叙述。

命名空间提供程序首先检查所提供的服务类是否真的存在。其次，根据指定的操作，采取相应的动作。关于有效的 essOperation 值的详情，以及 dwControlFlags 标志对这些值会产生什么样的影响，请参见第 8 章，其中着重讨论了 WSASetService。最后，提供程序的 NSPSetService 函数将对这些标志进行相应的处理。

如果服务提供程序在更新或删除一项不能找到的服务，就需要设置错误 WSASERVICE_NOT_

FOUND。如果提供程序在注册一项服务，WSAQUERYSETW 结构却是无效或未完成的，提供程序就会生成 WSAEINVAL 错误。

这个函数是最复杂的命名空间函数之一(仅次于 NSPLookupServiceNext)。提供程序必须维持一种方案，使其一直保持一项能被注册的服务，而且必须允许 NSPSetService 函数对数据进行更新。

12.2.2.7 NSPLookupServiceBegin

NSPLookupServiceBegin 函数和 NSPLookupServiceNext 及 NSPLookupServiceEnd 函数关联，用于初始化一个对命名空间的查询。这个函数对应于函数 WSALookupServiceBegin，用于建立搜索标准。它的原型如下：

```
int NSPLookupServiceBegin (
    LPGUID lpProviderId,
     (continued)LPWSAQUERYSETW lpqsRestrictions,
    LPWSASERVICECLASSINFOW lpServiceClassInfo,
    DWORD dwControlFlags,
    LPHANDLE lphLookup
);
```

和前面几个函数一样，第 1 个参数仍然是提供程序的 GUID。lpqsRestrictions 参数是定义查询参数的 WSAQUERYSETW 结构。第 3 个参数是 lpServiceClassInfo，它是一个 WSASERVICECLASSINFOW 结构，其中包含了指定服务类的简要信息，查询将在这个服务类中进行。dwControlFlags 参数采用了零个或若干个可对查询产生影响的标志。再次提醒大家注意，关于 WSALookupServiceBegin 和可以使用的标志，请参考第 8 章。注意，并不是所有的标志都能对每个提供程序产生影响。例如，如果您的命名空间不支持容器对象，便不必担心用于处理这些容器的标志(容器只是从概念上把服务组织在一起，至于容器内包含的究竟是什么，没有具体的限定)。最后的 lphLookup 参数是一个输出参数，它是一个定义特定查询的句柄。这个句柄用于 WSALookupServiceNext 和 WSALookupServiceEnd 之后的调用。

在实现 NSPLookupServiceBegin 时，要记住不能取消这个操作，且应该尽快完成它。因此，在打算开始网络查询时，就不用返回一个成功的响应。

提供程序本身应该保存查询参数，并将一个惟一的句柄关联到这个查询，以备日后引用。除此以外，提供程序还应该保持状态信息。我们将在下一节，深入讨论这样做的重要性和必要性。

12.2.2.8 NSPLookupServiceNext

一旦用 NSPLookupServiceBegin 启动了一个查询，应用程序便调用 NSPLookupServiceNext(该函数会依次调用命名空间提供程序函数 NSPLookupServiceNext)。事实上，这个调用是在搜索哪些服务符合查询标准。它的定义如下：

```
int NSPAPI WSALookupServiceNext (
```

```
    HANDLE hLookup,
    DWORD dwControlFlags,
    LPDWORD lpdwBufferLength,
    LPWSAQUERYSET lpqsResults
);
```

第 1 个参数 hLookup，是 WSALookupServiceBegin 函数返回的查询句柄。dwControlFlags 参数可以是 LUP_FLUSHPREVIOUS 标志，表明提供程序应该丢弃上一个结果集，转向下一个。一般说来，一个应用程序不能为结果提供足够大的缓冲区时，便请求丢弃上一个结果。下一个参数 lpdwBufferLength，表明被当作最后一个参数 lpqsResults 传递的缓冲区的长度。

NSPLookupServiceNext 被触发时，提供程序应该查找由句柄 hLookup 标识的查询参数。一旦获得了查询参数，便在已指定的服务类中开始搜索已注册的服务。就像上一小节中提到的那样，应该把查询状态保存下来。若发现了若干个匹配条目，调用进程便要调用若干次 WSALookupServiceNext，每次调用，提供程序都要返回一个数据集。如果没有找到匹配条目，提供程序就会返回 NSPLookup 错误。如果应用程序在调用 WSALookupServiceNext 的同时，又从另一个线程中调用了 WSALookupServiceEnd，正在进行中的查询就可能取消。在这种情况下，WSALookupServiceNext 调用将失败，并返回 WSA_E_CANCELLED 错误。

12.2.2.9　NSPLookupServiceEnd

完成查询之后，要调用 NSPLookupServiceEnd 函数结束查询，并释放所有命名空间资源。该函数的定义如下：

```
int NSPLookupServiceEnd (HANDLE hLookup );
```

这个函数只有一个参数——hLookup，该参数是一个句柄，指向将被关闭的查询。如果没有找到指定的句柄(或找到的句柄是无效句柄)，函数调用就会失败，并返回 WSA_INVALID_HANDLE 错误。

12.2.2.10　NSPIoctl

最后一个函数是 NSPIoctl，开发命名空间提供程序时不需要这个函数。如果 NSP 决定不实现这个函数，则 NSP_ROUTINE 的 cbSize 字段应该被设置为没有 LPNSPIOCTL 指针时的结构大小。下述代码可以完成这一任务：

```
lpnspRoutine->cbSize = FIELD_OFFSET(NSP_ROUTINE,NSPIoctl));
```

正如第 8 章所见，WSANSPIoctl API 是 Windows XP 中新出现的，目前仅有 NLA 命名空间使用它。这个 API 在当前网络发生变化时提供通知。

NSPIoctl 函数的定义为：

```
INT NSPIoctl(
    HANDLE              hLookup,
    DWORD               dwControlCode,
```

```
    LPVOID                  lpvInBuffer,
    DWORD                   cbInBuffer,
    LPVOID                  lpvOutBuffer,
    DWORD                   cbOutBuffer,
    LPDWORD                 lpcbBytesReturned,
    LPWSACOMPLETION         lpCompletion,
    LPWSATHREADID           lpThreadId
);
```

这个函数提供了一种从命名空间公开各种命令的方法。输入及输出参数怎么运行，完全由命名空间开发者决定。例如，如果 NSP 想打开一些统计信息(如已注册的实体数量或已执行的查询数量等)，可以执行 NSPIoctl，并允许应用程序使用 NSP 自己定义的 I/O 控制命令代码和输出缓冲结构，通过 WSANSPIoctl 来查询该 NSP。

这个函数的另外一个好处是，通过 lpCompletion 参数，它可以支持异步通知。这个结构允许发出调用的应用程序指定一个重叠结构、完成例程、窗口和完成端口，来接收完成的通知。同样，为通知注册什么样的信息，完全由 NSP 开发者确定，不过这个函数允许异步名称解析——这用 WSALookupService 函数是无法实现的。第 8 章讲述了当本地网络信息变化的时候，应用程序如何注册来接收通知。

如果命名空间选择执行这个函数，它必须保存输入和输出缓冲，以及 WSACOMPLETION 信息。然后，当 I/O 控制命令完成时(要么立刻完成，要么过一段时间之后完成)，就把输出信息复制到提供的缓冲区(如果缓冲区不够大，则返回错误)，且通知例程(如果已提供)也应该被传信。

根据实现服务的方式，有几种相应的方法可以处理这些异步完成事件。如果 NSP 是从 DDL 通知应用程序的，问题就很简单。要发送消息到一个窗口，使用 PostMessage 即可。要将 APC 放入队列，则使用 QueueUserApc。要传信一个事件，调用 SetEven。最后，要通知一个完成端口，使用 PostQueuedCompletionStatus。这些函数都是常规的 Windows API，关于它们的更多内容可以在 Platform SDK 中找到。

如果应用程序是从一个保持命名空间数据的服务或独立进程被通知的，情形就会困难得多。服务经常在一个不同的用户组下运行，因此可能无法访问某些资源。写窗口服务超出了本书的范围。更多的内容参考 Platform SDK 或 JeffreyRichter 和 JasonD.Clark 编著的《Programming Server-Side Applications for Windows 2000》(微软出版社，2000)。下一节将讨论示例 NSP，该 NSP 是作为一个独立的进程而实现的，但未公开 NSPIoctl 函数。

12.2.3 命名空间提供程序示例

关于如何创建命名空间以及创建时需要注意的重要问题(例如保持数据的方法)，前一节我们已讲了很多。但是，要开发一个完整的命名空间提供程序并不简单。接下来，为大家介绍一个命名空间示

例。该示例代码对一些需要注意的问题进行了解释。另外，为使大家易于理解，尽量使其简单化。

示例提供程序在补充材料的 NSP 编录下面的 3 个文件 MYNSP.H、MYNSP.CPP 和 MYNSP.DEF 中。示例命名空间的 DLL 便由这 3 个文件组成。除 DLL 外，大家还可以找到一个命名空间服务，它是一个 Winsock 服务器，负责处理 DLL 发出的请求。这个维护服务注册数据的服务器位于 MYNSP.CPP 文件中。另外两个文件，NSPSVC.CPP 和 PRINTOBJ.CPP，供 DLL 和服务所用，其中包含了支持例程，这些支持例程用于封送和打散在服务和 DLL 之间的套接字上发送的数据。数据的封送和打散稍后将具体说明。除了这两个文件，还有它们的头文件——NSPSVC.H 和 PRINTOBJ.H。这两个头文件中包含支持例程的函数原型。最后，RNRCS.C 是一个修改过的示例，对在我们定义的命名空间内注册和查询服务进行了解释，原来的示例参见第 8 章。应该注意，当我们将命名空间提供程序作为一种服务时，并不意味着示例代码是一个真正的 Windows 服务。

下面的小节将讨论怎样实现我们的命名空间。首先，大致谈谈所用的保持数据的方法。然后对实际的命名空间 DLL 的构成和命名空间服务的安装进行说明。接下来是命名空间服务的实现。最后展示应用程序如何对定义的命名空间进行注册和查询。

12.2.3.1　数据的保持

针对这里的命名空间，我们选用了一个独立的 Winsock 服务来保持命名空间信息。在 DLL 中执行的每一个命名空间函数中，要完成执行的话，需建立一个到该服务的连接，并对数据进行处理。为了简单起见，该服务运行于本地(服务在环回地址 127.0.0.1 上监听)计算机。在实现过程中，可通过注册或其他方法，来访问我们的命名空间服务的 IP 地址，这样，应用程序在调用这个命名空间时，无论它在什么地方运行，都可连接到这个命名空间服务。例如，以 DNS 为例，DNS 服务器的 IP 地址若不是静态设置的，就是在 DHCP 请求期间获得的。

当然，编写服务来维持这一信息并不是惟一的选择。还可在网络上维持一个文件，使其保留必要的信息。但是，因为性能受到了磁盘操作的限制，所以这也不是最佳途径。我们的命名空间示例所受到的一个性能限制在于：它建立了到这个服务的 TCP 连接。开发产品时，一般采用 UDP 之类的无连接数据报来提升系统性能。当然，要保证整体性能不受影响，还涉及到另外的编程要求，例如保证重新传输已丢弃的数据包。

12.2.3.2　命名空间 DLL

在实现命名空间服务之前，先来看看命名空间 DLL。每个命名空间提供程序都需要一个惟一的 GUID，我们的 GUID 定义在 MNSP.H 中。除了要有一个独一无二的标识符之外，我们的命名空间还需要一个简单的整数标识符。这个标识符可用于 WSAQUERYSET 结构的 dwNameSpace 字段中，就像大家在第 8 章所见的那样。我们的命名空间的 GUID 和标识符是这样的：

```
GUID MY_NAMESPACE_GUID = {0x55a2bd9e,0xbb30,0x11d2,
                          {0x91,0x66,0x00,0xa0,0xc9,0xa7,0x86,0xe8}
                         };
```

```
#define NS_MYNSP 66
```

如果要使用这个命名空间的应用程序，必须在它们的 Winsock 调用中指定这些值，所以这些值非常重要。当然，开发人员可以直接指定这些值，或通过 WSAEnumNameSpaceProviders 调用获得(详情参见第 8 章)这些值。同时要知道，一个应用程序在实现指定 NS_ALL 名称提供程序的操作时，应该在所有已安装名称提供程序上执行。这一点尤为重要，因为有几个 Windows 应用程序(例如 Microsoft Internet Explorer)需要在全部已安装的命名空间提供程序上执行查询。因此，在测试一个命名空间提供程序时，务必谨慎。编得不好的命名空间提供程序可能导致整个系统出问题。另外，因为安装命名空间提供程序需要 GUID 和命名空间标识符的值，所以它们也非常重要。

现在，来看看在 MYNSP.CPP 中实现的 NSP 函数。总的说来，这些函数非常相似，但启动和清除函数(NSPStartup 和 NSPCleanup)例外。启动函数只是利用我们定义的命名空间函数对 NSP_ROUTINE 结构进行了初始化。因为没必要进行清除，所以清除例程没有派上用场。

其余的函数需要和我们的服务进行交互，对数据进行查询或注册。需要和服务进行通信时，采取下列步骤即可：

1. 与服务建立连接(利用 NyNspConnect 函数)。
2. 写入一个 1 字节的行动代码，向服务指示即将采取的行动(服务注册、服务删除、查询等等)。
3. 封送参数，并把它们发给服务，参数类型由操作决定。例如，NSPLookupServiceNext 向服务发送了查询句柄，以便服务开始查询，而 NSPSetService 则发送一个完整的 WSAQUERYSET 结构。
4. 读取返回的代码。一旦服务有了执行操作请求所需的参数，就会返回操作的返回代码(不管是成功，还是失败)。所以 MYNSP.H 文件中，定义了两个常量：MYNSP_SUCCESS 和 MYNSP_ERROR。
5. 如果是请求查询，而且返回的代码是成功的话，就可以读取和打散返回结果了。举个例子来说，在已找到匹配事件中，NSPLookupService 返回的就是一个 WSAQUERYSET 结构。

由此看来，实现 DLL 并不复杂。NSP 函数必须获得参数，并对它们进行处理，在这里要做的是将相关操作的信息传递到命名空间服务。然后，由服务执行所请求的操作。但是，我们忽略了一个非常困难的但又必须执行的操作：通过套接字收发数据。一般说来，发送数据没有什么特殊要求，但在发送整个数据结构时，情况大不一样。大多数命名空间函数都把 WSAQUERYSET 或 WSASERVICECLASSID 当作参数。这个对象必须在连向服务的套接字连接上收发。难就难在这些结构不是连续不断的内存块。换言之，这些结构中包含字符串指针，而别的结构可以放在内存中的任何一个地方(如图 12.5 所示)。需要做的是封送这些分散的内存段(不管它们在哪里)并把它们逐个复制到一个独立的缓冲区内。这便是通常所说的“封送数据”(marshaling data)。在接收数据的这一端，必须反转这一进程。也就是说，读取的数据需要进行重组，还原成最初的结构，这些字符串指针必须进行“修复”，以便指向接收端上的有效内存位置。

针对这里的命名空间提供程序，我们提供了封送和打散 WSANAMESPACEINFO 和 WSAQUERYSET 结构的函数。这些函数位于 NSPSVC.CPP 文件中，供该命名空间的 DLL 和该命名空间服务使用(因为双方都需要能够收集和打散结构)。4 个函数都很简单，所以不打算在此赘述。

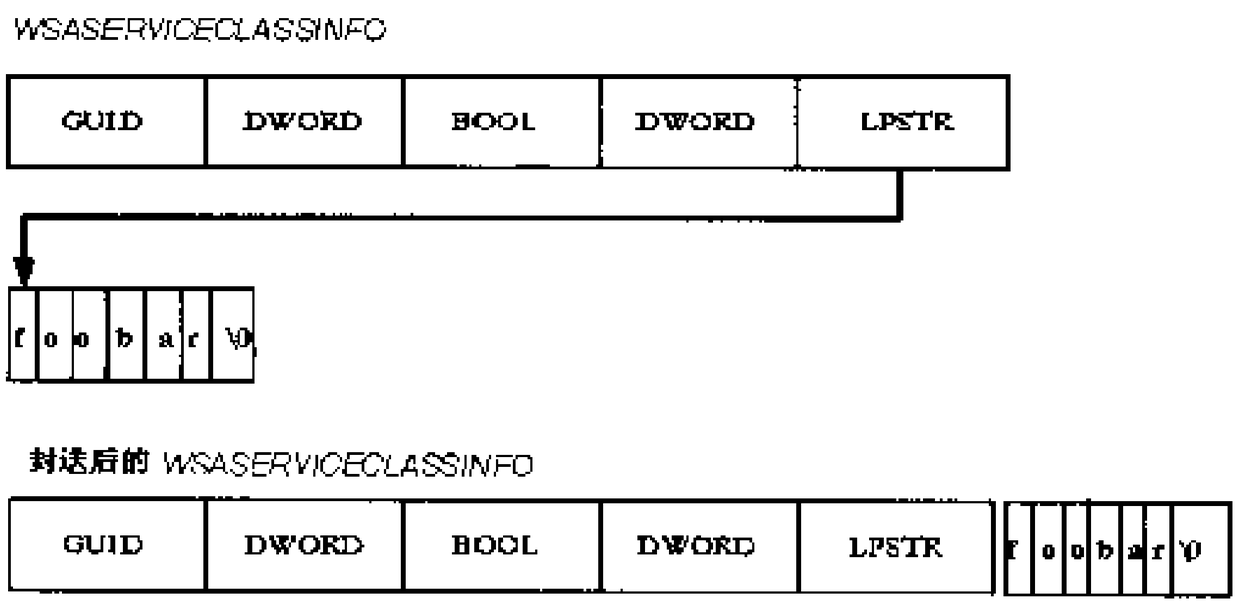


图 12.5　封送数据

12.2.3.3　命名空间的安装

整个过程中，命名空间提供程序的安装是最关键的一步。NSPINSTALL.C 文件是一个简单的安装程序。下面的代码便可安装我们的提供程序：

```
ret = WSCInstallNameSpace(L"Custom Name Space Provider",
    L"%SystemRoot%\\System32 \\Mynsp.dll",NS_MYNSP,1,
    &MY_NAMESPACE_GUID);
if (ret == SOCKET_ERROR)
{
    printf("Failed to install name space provider:%d \n",
    WSAGetLastError());
}
```

这个调用中的参数，只是提供程序的名称、DLL 的位置、整数标识符、版本以及 GUID。完成安装之后，惟一需要做的就是：确保提供的命名空间 DLL 的位置正确。真正可能出现的错误，便是试图安装的那个命名空间的 GUID 已经被另一个提供程序所用。

命名空间提供程序的删除更为简单。下面的代码取自我们的安装程序，用于删除我们的提供程序：

```
ret = WSCUnInstallNameSpace(&MY_NAMESPACE_GUID);
if (ret == SOCKET_ERROR)
{
    printf("Failed to remove provider:%d \n",WSAGetLastError());
}
```

12.2.3.4　命名空间服务

命名空间服务才是名称提供程序的精髓所在。这个服务可对所有已注册的服务类和服务实例进行追踪。命名空间 DLL 被用户的应用程序触发时，便连接到命名空间服务，开始执行操作。这个服务

很简单。在主函数内部，建立了一个监听套接字。然后，在一个循环内，命名空间 DLL 实例接受连接。为了简单起见，一次只处理一个连接。这样做还可以防止同步访问保持命名空间信息的那个数据结构。再次提醒大家注意，因为这样会降低性能，所以实际上提供程序不会如此执行，这样做的目的，是方便大家理解。一旦连接被接受，服务就从标识下一个行动的命名空间 DLL 中，读取一个单字节。

循环内部，对动作进行解码，并把命名空间 DLL 中的参数传递给服务。这时起开始执行所请求的动作。这些动作并不复杂，很容易弄明白，并且，可从针对每个可能的动作而采取的步骤中，看出命名空间服务的原理，这里没有必要赘述。但我们仍然要提到维持信息的各个结构。命名空间提供程序真正关心的数据类型只有两种：WSASERVICECLASSINFO 和 WSAQUERYSET。大家已知道，大部分 RNR 函数在其参数中，都引用了一个或多个这样的结构。如此一来，我们维持的是两大通用数组(分别对应这两个结构)和各数组的计数器。

DLL 请求安装服务类时，命名空间提供程序的主函数率先调用 LookupServiceClass(一个支持例程，定义 MYNSPSVC.CPP 中)。这个函数会遍历 ServiceClasses 数组，该数组由多个 WSASERVICECLASSINFO 结构构成。如果没有发现具有同样 GUID 的服务类，服务使返回一个错误(DLL 将它转换成 WSAEALREADY)。若不然，就会在这个数组后添加新的服务类，而且 dwNumServiceClasses 计数器会递增。

删除服务类的过程中(和安装服务类一样)，主函数调用了 LookupServiceClass。在这种情况下，如果服务类找到了，代码便将该数组的最后一个服务类移到已删除类的位置上。然后，代码使计数器递减。在 Winsock 2 规范中，对命名空间提供程序来说，没有特别指明这一点：在打算删除一个服务类时，如果已注册的服务仍然对该类进行了引用，会有什么情况发生。具体怎样处理这种情况，由您自己决定。但如果已注册服务引用了一个服务类，命名空间示例就不会允许将这个服务类删除。

维持 WSASERVICECLASSINFO 结构采取的原则同样适用于 WSAQUERY 结构的追踪。这里有一个结构数组(名为 Services)和一个计数器(名为 dwNumService)。服务的添加和删除处理方式和服务类的一样。

最后，服务必须为查询维持信息。应用程序发起一个查询时，必须维持查询期间所需的参数，并为查询分配一个惟一的句柄(即查询句柄)。这是非常必要的，因为 WSALookupServiceNext 只通过这个句柄来引用查询。必须保留的其他信息便是查询状态。也就是说，每次调用 WSALookupServiceNext，都能返回一个独立的信息集。代码必须记住 Services 数组中的最后一个位置，数据是在这个位置返回的。这样，以后对 WSALookupServiceNext 的调用，使从上一次调用离开的那个位置开始返回。

12.2.3.5 命名空间的查询

我们的命名空间示例中，最后一部分便是 RNRCS.C 文件。它是第 8 章的名称注册和解析示例的改版。为了使示例尽可能浅显易懂，我们只做了少许改动。第 1 个改动是，使代码列举已安装的命名空间提供程序，但只返回 NS_MYNSP 提供程序。第 2 个改动是，在注册一个服务时，RNRCS.C 只列

举本地 IP 接口，并会用它代替我们的服务地址。我们的服务提供程序支持任何 SOCKADDR 类型的注册。最后，该示例没有针对服务注册建立服务实例，只注册了服务名。其他的则和第 8 章中的示例一样。

12.2.3.6　运行示例

一旦已完成所有示例的编译，提供程序的安装和使用就非常简单了。可以利用下面的命令安装提供程序：

```
Nspinstall.exe install
```

当然，别忘了把 MYNSP.DLL 复制到%SystemRoot%\System32(通过下一个命令完成)。一旦安装了命名空间，服务实例便开始运行，以供查询和注册服务之用。

```
Mynspsvc.exe
```

现在，可以利用 RNRCS.EXE 查询和注册服务了。表 12.6 展示了一些应该执行的命令以及应该遵循的顺序。这个命令序列注册两个服务，随后执行一个通配符查询和一个指定查询。然后，再对这两个服务进行查询，并将它们删除。最后，我们执行了一个通配符查询，用以说明服务已经被删除。

表 12.6　运行示例命名空间

命　令	含　义
Rnrcs.exe-s:ASERVICE	注册服务 ASERVICE
Rnrcs.exe-s:BSERVICE	注册服务 BSERVICE
Rnrcs.exe-c:*	查询全部已注册的服务
Rnrcs.exe-c:BSERVICE	只查询名为 BSERVICE 的服务
Rnrcs.exe-c:ASERVICE-d	只查询名为 ASERVICE 的服务，若找到，便将其删除
Rnrcs.exe-c:BSERVICE-d	只查询名为 BSERVICE 的服务，若找到，便将其删除
Rnrcs.exe-c:*	查询全部已注册的服务

12.3　小　　结

Winsock 2 SPI 使软件开发人员能够对 Winsock 2 的性能进行扩展，这是通过开发服务提供程序来实现的。通过本章的学习，大家了解了如何开发传输服务提供程序和命名空间提供程序。本章结尾，还对 Winsock 连网技术进行了一番总结性讨论。下面两章，将介绍访问 Winsock 功能的两种另外的编程语言，即：Visual Basic 和 C#。

远程访问服务

迄今为止，本书已介绍了可在 Microsoft Windows 操作系统中使用的全部 Winsock API 函数。利用这些函数可以开发通过网络进行通信联系的应用程序。本章讲述一种重要的服务，名为 RAS(Remote Access Service，远程访问服务)，它允许用户将自己的计算机连接到一个远程网络如 ISP，或连接到一个企业通信网。一旦建立了连接，便可使用本书讲述的所有网络函数，即使您手上的计算机实际连接的是一个远程网络。

15.1 RAS 客户机

微软的所有 Windows 平台中都有 RAS 客户机，它允许我们将自己的计算机与另一个地方的远程计算机(其特性是一个远程访问服务器组件)相连。一般情况下，RAS 客户机利用连接了电话线的串行通信设备(如调制解调器)，通过拨号的方式呼叫远程计算机。因此，有时，RAS 客户机也被称为 DUN(dial-up networking，拨号联网)客户机。RAS 也支持通过隧道连接安全地利用 IP 网络(如 Internet)连接到远程网络，这被称为 VPN(Virtual Private Networking，虚拟专用网)。

在远程网络中，必须有一个等候 DUN 或 VPN 连接的服务器。RAS 客户机能够和多种类型的远程访问服务器建立通信。RAS 通过使用工业标准串行组帧协议及 IP 隧道协议建立通信。下面的协议是串行组帧协议，这里数据通信是在一个串行设备上进行的，如调制解调器：

- PPP(Point-to-Point Protocol，点到点协议) 可传输 IP、IPX 和 NetBEUI 通信协议。
- SLIP(Serial Line Internet Protocol，串行线路网际协议) 只能传输 IP 通信协议。
- 异步 NetBEUI((Microsoft Windows NT 3.1、Microsoft Windows for Workgroups 3.11) 只能传输 NetBEUI 通信协议。

RAS 使用下列 IP 隧道协议，这里数据通信是在一个已有 IP 连接上进行的：

- PPTP(Point-to-Point Tunneling Protocol，点到点隧道协议) 可安全地传输 IP 和 IPX 通信协议。
- L2TP(Layer 2 Tunneling Protocol，第 2 层隧道协议) 可安全地传输 IP 和 IPX 通信协议。

上述组帧协议描述了数据是怎样通过 RAS 通信链接进行传输的，并指明在 RAS 链接上可采用哪些网络通信协议(例如 TCP/IP 或 IPX)来通信。如果 RAS 服务器支持前面定义的任何一种组帧协议，RAS 客户机便可以与之建立一个链接。Windows 95、Windows 98、Windows Me 以及 Windows NT 均具有 RAS 服务器组件，能支持前面列出的任何一种串行组帧协议。Windows Me 以及所有的 Windows NT 平台也都支持 IP 隧道协议。

RAS 客户机和服务器之间的连接建立之后，网络协议堆栈(与所用的组帧协议和隧道协议有关)就通过这个 RAS 连接与远程计算机通信，就像通过 LAN 连接的一样。当然，如今许多调制解调器的数据通信速率明显比直接的 LAN 连接慢。

当一个 RAS 服务器接受串行组帧连接或 IP 隧道连接时，首先，它会通过协商前面列出的其中一种组帧协议或隧道协议，来建立与客户机的通信。协议连接一旦确立，RAS 服务器就会尝试对连接用户进行身份验证。本章描述的 RAS API 函数允许 RAS 客户机为这个 RAS 服务器指定用户名、口令和域登录凭证。当一个基于 Windows NT 的 RAS 服务器收到这条信息时，它就会利用 Windows NT 域安全访问控制验证登录凭证。注意，RAS 服务器没有让客户机登录到 Windows NT 域；相反地，它采用客户机凭证来验证是否允许用户建立 RAS 连接。RAS 连接过程和 Windows NT 域登录过程不一样。RAS 连接成功建立之后，计算机就可登录到 Windows NT 域。本章不详细说明整个登录过程。在 Windows 95、Windows 98、Windows Me、Windows XP 和 Windows .NET 服务器上，当一个 RAS 连接经过电话簿条目中可用选项的验证后，RAS 便可自动让计算机登录到一个域，这一点本章将在稍后讲到。

对串行通信而言，RAS 依赖 TAPI(Telephony Application Programming Interface，电话应用程序编程接口)来设置和控制调制解调器之类的串行通信设备，TAPI 控制这些拨号设备的硬件设置。在通过调制解调器建立 RAS 连接时，TAPI 会转向调制解调器，并从 RAS 向调制解调器发出拨号信息。这样，RAS 就会把调制解调器看成简单的 TAPI 调制解调器端口，这种端口具备拨号并与远程服务器建立电话连接的能力。正如大家稍后将看到的那样，在设置 RAS 连接信息时，有的 RAS API 函数引用了 TAPI 调制解调器端口。

本章还将介绍如何通过编程的方式，利用 RAS 建立与远程网络的通信。首先为大家介绍建立应用程序时，需要哪些头文件和库文件。接下来介绍关于拨号的基本知识——如何通过串行设备真正建立一个远程连接。然后介绍如何设置一个 RAS 电话簿，定义有关的 RAS 连接的详细通信属性。介绍完有关设置通信的基础知识之后，再为大家展示如何管理已建立的连接。最后叙述如何建立一个 VPN 连接。

15.2　编译和链接

在开发一个 RAS 应用程序时，需要包含以下几个头文件和库文件来建立自己的应用程序：

- RAS.H：包含 RAS API 函数所用的函数原型和数据结构。
- RASERROR.H：包含 RAS API 函数失败时所用的预先定义的错误代码。
- RASAPI32.LIB：所有 RAS API 函数的库。

RASERROR.H 列举了相当多预先定义的错误代码。这个文件中，大家将注意到一个错误说明字符串，这个字符串和 RAS 中所用的各种错误代码关联在一起。RAS 有一个名为 RasGetErrorString 的函数，它非常有用，允许我们通过编程来获得与特定 RAS 错误代码关联的错误字符串。RasGetErrorString 的定义如下：

```
DWORD RasGetErrorString(
    UINT uErrorValue,
    LPTSTR lpszErrorString,
    DWORD cBufSize
);
```

uErrorValue 参数接收一个特定的 RAS 错误代码，这个代码是一个 RAS 函数返回的。lpszErrorString 参数是一个由应用程序提供的缓冲区，将接收一个错误字符串，该字符串与 uErrorValue 参数中的错误代码关联到一起。这时应该有一个较大的缓冲区，以便能够容纳错误字符串；否则，该函数就会失败，并返回 ERROR_UNSUFFICIENT_BUFFER 错误。我们建议缓冲区长度至少为 256 个字符，对时下的任何一个 RAS 错误字符串来说，这个长度都是比较恰当的。最后一个参数 cBufSize 是为 lpszErrorString 提供的缓冲区的长度。

15.3 数据结构和平台兼容性问题

在编译和建立自己的应用程序时，大家会发现：RAS 函数所用的有些数据结构中，根据 WINVER 定义的值，会分别使用或不用一些特别的数据字段。但是，Windows CE SDK 没有定义 WINVER，因此，前面提到的字段不能应用于 Windows CE。RAS 数据结构中还有一个 dwSize 字段，必须把它设为您使用的 RAS 结构的字节长度。因为对使用这些结构的 RAS 函数而言，目标是针对特定平台的，所以这样做会对这些 RAS 函数产生影响。WINVER 标准适用于 Windows 95、Windows 98 和 Windows NT 平台。

- WINVER = 0X400：指明 RAS 应用程序的目标平台是 Windows 95、Windows 98 或没有服务包的 Windows NT 4.0。
- WINVER = 0X401：指明 RAS 应用程序的目标平台是有服务包的 Windows NT。
- WINVER = 0X500：指明 RAS 应用程序的目标平台是 Windows 2000。
- WINVER = 0X501：指明 RAS 应用程序的目标平台是 Windows XP 和 Windows .NET 服务器。

因为 RAS 数据结构在程序编译过程中会基于 WINVER 值而具有不同大小，所以 RAS 本身没有

可在所有平台上运行的一个可执行字段。但通过编程，利用一个可执行字段，仍然可能获得所有平台的支持(当然 Windows CE 除外)。尽管如此，我们还是强烈建议大家在编写 RAS 应用程序时，应该有一个特定的目标平台。

15.4　DUN1.3 升级和 Windows 95

在第 1 个版本到 OSR2 版本期间，Windows 95 发布了大量的版本。OSR2 版本是非卖品，只供 OEM(original equipment manufacturer，原始设备制造商)厂商安装在它们的产品上。Windows 95 的每次发布都具有不同的 RAS API 功能。因此，我们建议大家最好安装最新的 RAS 升级包(DUN1.3)以进一步发挥 RAS 的潜力。要获得 Windows 95 的 DUN1.3 升级版，可访问 http://www.microsoft.com/support。本章的前提是您的计算机上至少已经安装了 DUN1.3 升级版；至于以前的 DUN 版本中的 RAS，我们将不讨论。

15.5　RASDIAL

RAS 客户机应用程序准备和一个远程计算机建立通信时，必须调用 RasDial 函数。RasDial 函数相当复杂，要提供许多调用参数，这些参数用于拨号、身份验证以及和 RAS 服务器建立远程连接。该函数的定义如下：

```
DWORD RasDial(
    LPRASDIALEXTENSIONS lpRasDialExtensions,
    LPCTSTR lpszPhonebook,
    LPRASDIALPARAMS lpRasDialParams,
    DWORD dwNotifierType,
    LPVOID lpvNotifier,
    LPHRASCONN lphRasConn
);
```

基于 lpvNotifier 参数的值，RasDial 调用可以在两种操作模式中执行：同步和异步。同步模式中，在这个函数成功建立连接或以失败告终以前，RasDial 会处于阻塞状态，而异步模式中，RasDial 则立即完成连接，同时允许应用程序在连接期间执行别的操作。

15.5.1　同步模式

如果把 RasDial 的 lpvNotifier 参数设为 NULL，RasDial 就会进入同步模式。lpvNotifier 参数是 NULL 时，dwNotifierType 参数就会被忽略。同步调用 RasDial 是使用该函数的最简单的方式；美中不足的是，在同步模式下不能对连接进行监视，而在异步模式下则可以。稍后，我们将就此进行详述。

下述代码示例演示了如何同步调用 RasDial:

```
RASDIALPARAMS RasDialParams;
HRASCONN hRasConn;
DWORD Ret;

//始终设置 RASDIALPARAMS 结构的大小

RasDialParams.dwSize = sizeof(RASDIALPARAMS);
hRasConn = NULL;

//将这个字段设为空字符串，可让 RasDial 使用默认的拨号属性
lstrcpy(RasDialParams.szEntryName,"");

lstrcpy(RasDialParams.szPhoneNumber,"867-5309");
lstrcpy(RasDialParams.szUserName,"jenny");
lstrcpy(RasDialParams.szPassword,"mypassword");
lstrcpy(RasDialParams.szDomain,"mydomain");

//同步调用 RasDial(第 5 个参数设为 NULL)

Ret = RasDial(NULL,NULL,&RasDialParams,0,NULL,&hRasConn);
if(Ret != 0)
{
    printf("RasDial failed:Error = %d \n",Ret);
}
```

这个示例通过填充 lpRasDialParams 参数的字段来调用 RasDial。lpRasDialParams 参数是一个 RASDIALPARAMS 结构指针，定义了拨号和用户身份验证参数，RasDial 函数用这些参数建立一个远程连接。RASDIALPARAMS 结构的定义为:

```
typedefstruct _RASDIALPARAMS {
    DWORD dwSize;
    TCHAR szEntryName[RAS_MaxEntryName + 1];
    TCHAR szPhoneNumber[RAS_MaxPhoneNumber + 1];
    TCHAR szCallbackNumber[RAS_MaxCallbackNumber + 1];
    TCHAR szUserName[UNLEN + 1];
    TCHAR szPassword[PWLEN + 1];
    TCHAR szDomain[DNLEN + 1];
#if(WINVER >= 0x401)
    DWORD dwSubEntry;
    DWORD dwCallbackId;
#end if
}RASDIALPARAMS;
```

RASDIALPARAMS 结构的各个字段提供了建立一个连接的基础，描述如下：

- dwSize：应该被设为一个 RASDIALPARAMS 结构的长度(按字节算)。有了这个标志，RAS 使可对编译程序时所用的 WINVER 版本进行判断。
- szEntryName：一个字符串，允许您标识一个电话簿条目，该条目包含在 RasDial 函数的 lpszPhonebook 参数中列出的电话簿文件内。因为电话簿条目能够使您更好地指定 RAS 连接属性，例如选定一个 Modem 或选定一个分帧协议等等，所以这个参数非常重要。但是，为使用 RasDial 而指定一个电话簿条目是可选的。如果这个字段是空字符串(“”)，RasDial 就会选择系统上已安装的第 1 个可用的 Modem，并依据下一个参数 szPhoneNumber 来拨叫连接。稍后将详细叙述电话簿条目。
- szPhonebookNumber：一个字符串，代表一个电话号码，这个号码优先于 szEntryName 字段中指定的电话簿条目内包含的电话号码。
- szCallbackNumber：允许您指定一个电话号码，RAS 服务器可以根据这个号码回叫。如果 RAS 服务器允许有一个回叫号码，它就会中断原来的连接，利用指定的回叫号码回叫。这个特性非常不错，因为有了它，服务器就可知道连接的用户来自何处。
- szUserName：一个字符串，标识 RAS 服务器上的用户进行身份验证时所用的登录名。
- szPassword：一个字符串，标识 RAS 服务器上的用户进行身份验证时所用的密码。
- dwSubentry：标识用户账号所在的 Windows NT 域。
- szDomain：可选。允许指定最初的电话簿子条目，来拨叫一个 RAS 多链路连接。本章未叙述多链路特性。
- dwCallbackId：允许把一个应用程序定义的值传递到一个 RasDialFunc2 回叫函数中，这个字段没有被使用。

前面的示例将一个 RAS 需要的大量的拨号、用户名、密码及域信息填入一个 RASDIALPARAMS 结构中。之后 RASDIALPARAMS 结构将被传递给 RasDial 函数，该函数会同步地建立远程连接。

15.5.2　异步模式

比起同步调用 RasDial 函数来，异步调用要复杂得多，但它在建立连接时提供了巨大的灵活性。如果 RasDial 的 lpvNotifier 参数没有设为 NULL，RasDial 就会进入异步模式——调用会立即返回，但连接继续进行。在进行 RAS 连接时，异步调用 RasDial 是优选方法，因为可以对连接进程进行监视。lpcNotifier 参数既可以是一个指针，指向 RasDial 中发生连接活动时被调用的函数，又可以是一个通过 Windows 消息来接收进程通知的窗口句柄。RasDial 函数的 dwNotifierType 参数用于判断函数类型，或被传递到 lpvNotifier 的窗口句柄。至于 dwNotifierType 中可以指定哪些值，表 15.1 对此进行了说明。

表 15.1 RasDial 异步通知方法

通知类型	含 义
0	lpvNotifier 参数致使 RasDial 使用 RasDialFunc 函数指针管理连接事件
1	lpvNotifier 参数致使 RasDial 利用 RasDialFunc1 函数指针管理连接事件
2	lpvNotifier 参数致使 RasDial 利用 RasDialFunc2 函数指针管理连接事件
0xFFFFFFF	lpvNotifier 参数致使 RasDial 在连接事件期间发送一个窗口消息

表 15.1 展示了 lpvNotifier 参数中的 3 个回叫函数原型，可以把它们提供给 RasDial 函数，用来接收连接事件的通知。它们是：RasDialFunc、RasDialFunc1 和 RasDialFunc2。RasDialFunc 的原型如下：

```
VOID WINAPI RasDialFunc(
    UINT unMsg,
    RASCONNSTATE rasconnstate,
    DWORD dwError
);
```

unMsg 参数接收已发生的事件类型。当前的这个事件只可能是 WM_RASDIALEVENT，意味着这个参数没有用处。rasconnstate 参数接收 RasDial 函数即将发生的连接活动。表 15.2 定义了可能发生的连接活动。如果连接活动失败，dwError 参数就会收到 RAS 错误代码。

表 15.2 RAS 连接活动

活 动	状 态	说 明
RASCS_OpenPort	运行	即将打开一个通信端口
RASCS_PortOpened	运行	通信端口已打开
RASCS_ConnectDevice	运行	准备与一个设备连接
RASCS_DeviceConnected	运行	已成功与设备连接
RASCS_AllDevicesConnected	运行	物理链接已建立
RASCS_Authenticate	运行	RAS 身份验证进程已开始
RASCS_AuthNotify	运行	身份验证事件已发生
RASCS_AuthRetry	运行	服务器已请求尝试另一个身份验证
RASCS_AuthCallback	运行	服务器已请求一个回叫号码
RASCS_AuthChangePassword	运行	客户机已请求改变 RAS 账号上的密码

续表

活　动	状　态	说　明
RASCS_AuthProject	运行	协议发送准备进行
RASCS_AuthLinkSpeed	运行	链接速度正在计算过程中
RASCS_AuthAck	运行	身份验证据请求正在确认过程中
RASCS_ReAuthenticate	运行	回叫之后的身份验证进程准备开始
RASCS_Authenticated	运行	客户机已成功结束身份验证
RASCS_PrepareForCallback	运行	线路即将断开连接，准备回叫
RASCS_WaitForModemReset	运行	准备回叫之前，客户机等待 Modem 的重新设置
RASCS_WaitForCallback	运行	客户机等待服务器发出的进入拨号
RASCS_Projected	运行	协议发送已完成
RASCS_StartAuthentication	运行	用户身份验证已开始或已完成(只适用于 Windows 95、Windows 98 和 Windows Me)
RASCS_CallbackComplete	运行	已经回叫客户机(只适用于 Windows 95、Windows 98 和 Windows Me)
RASCS_LogonNetwork	运行	客户机正在登录远程网络(只适用于 Windows 95、Windows 98 和 Windows Me)
RASCS_SubEntryConnected	运行	多链路电话簿条目的子条目已连接。RasDialFunc2 的 dwSubEntry 参数中将包括一个已连接子条目的索引
RASCS_SubEntryDisconnected	运行	多链路电话簿条目已断开。RasDialFunc2 的 dwSubEntry 参数中将包含这个已断开连接的子条目的索引
RASCS_RetryAuthentication	暂停	RasDial 正在等待新的用户凭证
RASCS_CallbackSetByCaller	暂停	RasDial 正在等待客户机的回叫号码
RASCS_PasswordExpired	暂停	RasDial 希望用户提供一个新密码
RASCS_InvokeEapUI	暂停	Windows NT 平台上，RasDial 等待自定义用户接口，以便获得 EAP 信息

续表

活 动	状 态	说 明
RASCS_Connected	终止	RAS 连接成功，处于活动状态
RASCS_Disconnected	终止	RAS 连接失败或处于不活动状态

表 15.2 展示了异步 RasDial 调用中，连接活动有 3 种状态：运行、暂停和中止。运行状态表明 RasDial 调用仍处于进程中，每一个处于运行状态的活动都将提供进程状态信息。

暂停状态，是指 RasDial 需要更多信息来建立连接。默认设置下，禁用暂停状态。在基于 Windows NT 的操作系统中，在 RasDial 的 RASDIALEXTENSIONS 结构参数中设置 RDEOPT_PausedStates 标志，使可启用通知进程。连接活动处于暂停状态时，表明存在下述状况中的一种：

- 因为身份验证失败，所以用户需要提供新的登录凭证。
- 因为密码已到期，所以用户需要提供一个新的密码。
- 用户需要提供一个回叫号码。

这些活动与本章前面提到的 RASDIALPARAMS 结构中的信息相关。发生处于暂停状态的连接活动时，RasDial 就会向您的回叫函数(或窗口进程)发出通知。如果禁用暂停状态，RAS 就会向通知函数发送一个错误，RasDial 就会失败。如果启用暂停状态，RasDial 函数就会进入暂停状态，这样，应用程序就可以通过 RASDIALPARAMS 结构，提供必要的信息。RasDial 处于暂停状态时，通过原来的调用连接句柄(lphRasConn)和通知函数(lpvNotifier)，再次调用这个函数，便可恢复运行。亦可以调用 RasHangUp(稍后将讨论)，简单地终止暂停操作。恢复暂停连接时，需要通过传递给已恢复的 RasDial 调用的 RASDIALPARAMS 结构，来提供必要的用户输入。

注意 恢复暂停状态时，不要直接从一个通知处理程序函数(例如 RasDialFunc)调用 RasDial。RasDial 尚不能处理这种情况，所以应该直接从应用程序线程中恢复暂停状态。

最后一个状态(终止)表明 RasDial 连接不是已成功，就是已失败。还可以表示 RasHangUp 函数关闭了连接。

现在，对于监视异步 RasDial 调用的连接，大家已有初步的认识了。接下来，我们将演示如何设计一个简单的程序，使其能够异步调用 RasDial。下列代码展示了这一过程。在补充材料中可以找到一个异步调用 RasDial 示例。

```
void main(void)
{
    DWORD Ret;
    RASDIALPARAMS RasDialParams;
    HRASCONN hRasConn;
    //将调用参数填入 RASDIALPARAMS 结构，就像同步示例中所做的那样
```

```
    ...
    if((Ret = RasDial(NULL,NULL,&RasDialParams,0,
        &RasDialFunc,&hRasConn))!= 0)
    {
        printf("RasDial failed with error %d \n",Ret);
        return;
    }
    //如果 RasDial 成功，它就会离开完成，使得我们可以在 RasDial 进行时执行其他任务
    ...
}
//回叫函数 RasDialFunc()
void WINAPI RasDialFunc(UINT unMsg,RASCONNSTATE rasconnstate,
    DWORD dwError)
{
    char szRasString[256];//错误字符串的缓冲区
    if(dwError)
    {
        RasGetErrorString((UINT)dwError,szRasString,256);
        printf("Error:%d -%s \n",dwError,szRasString);
        return;
    }
    //映射 RasDial 的每个状态，并在屏幕上显示 RasDial 中进入的下一个状态

    switch (rasconnstate)
    {
        case RASCS_ConnectDevice:
            printf("Connecting device...\n");
            break;
        case RASCS_DeviceConnected:
            printf("Device connected.\n");
            break;
            //在这里添加其他连接活动
            ...
            default:
                printf("Unmonitored RAS activity.\n");
                break;
        }
}
```

RAS 的特征之一是独立函数 RasConnectionNotification，它允许应用程序决定何时建立或终止 RAS 连接。它的定义如下：

```
DWORD RasConnectionNotification(
    HRASCONN hrasconn,
    HANDLE hEvent,
    DWORD dwFlags
```

```
);
```

hrasconn 参数是 RasDial 返回的一个连接句柄。hEvent 参数是一个事件句柄，该句柄是应用程序利用 CreateEvent 函数创建的。dwFlags 参数可以设为下列连接活动标志的任何一种组合。其中最有用的是：

- RASCN_Connection：通知您 RAS 连接已经建立。如果 hrasconn 参数设为 INVALID_HANDLE_VALUE，则任何一次 RAS 连接都会传信这一事件。
- RASCN_Disconnection：通知您 RAS 连接已经终止。如果 hrasconn 参数设为 INVALID_HANDLE_VALUE，则任何连接终止都会传信这一事件。

注意，这些标志的表现和表 15.2 中描述的连接活动标志一样。如果连接期间发生了这些活动，您的事件就会被传信。因此，应用程序应该使用操作系统等待函数，比方说 WaitForSingleObject，利用它们来决定什么时候向对象发出通知。

15.5.3 关闭连接

关闭 RasDial 建立的连接很简单。只须调用 RasHangUp 即可。该函数的定义如下：

```
DWORD RasHangUp(
    HRASCONN hrasconn
);
```

hrasconn 参数是 hasDial 返回的一个句柄。尽管这个函数非常容易使用，但仍然需要了解 RAS 中，连接的内部管理是怎么回事。串行连接在使用一个调制解调器端口时，如果连接关闭，这个端口需要花时间重新设置这个连接。因此，应该一直等下去，直到端口连接完全关闭为止。要做到这一点，可调用 RasGetConnectionStatus 来判断连接何时被重置。RasGetConnectionStatus 的定义是这样的：

```
DWORD RasGetConnectStatus(
    HRASCONN hrasconn,
    LPRASCONNSTATUS lprasconnstatus
);
```

hrasconn 参数是 RasDial 返回的一个句柄。lprasconnstatus 参数是一个 RASCONNSTATUS 结构，用于接收当前的连接状态。RASCONNSTATUS 结构的定义如下：

```
typedefstruct _RASCONNSTATUS
{
    DWORD dwSize;
    RASCONNSTATE rasconnstate;
    DWORD dwError;
    TCHAR szDeviceType[RAS_MaxDeviceType + 1];
    TCHAR szDeviceName[RAS_MaxDeviceName + 1];
}RASCONNSTATUS;
```

下面这个结构的各个字段定义如下：

- dwSize：应该设置为 RASCONNSTATUS 结构的长度(按字节算)。
- rasconnstate：接收表 15.2 中定义的一种连接活动。
- dwError：若 RasGetConnectStatus 没有返回 0，就获取一个具体的 RAS 错误代码。
- szDeviceType：接收一个字符串，该字符串代表连接所用的设备类型。
- szDeviceName：接收当前的设备名。

我们建议在接收 RASCS_Disconnected 活动状态之前，先检查一下自己的连接状态。显而易见，在重新设置连接之前，可能需要多次调用 RasGetConnectionStatus。一旦连接被重新设置，就可以退出应用程序或建立另一个连接了。

至此已叙述了建立一个 RAS 连接的基础知识，接着将叙述如何建立电话簿条目，使用它可以为建立一个 RAS 连接设置高级通信属性。

15.6　电　话　簿

电话簿不过是一个 RASENTRY 结构的集合，这些结构中包含了电话号码、数据速率、用户身份验证信息、VPN 策略和其他连接信息。在 Windows 95、Windows 98、Windows Me 和 Windows CE 系统中，电话簿保存在系统的注册表内。在 Windows NT 系统中，电话簿存储在扩展名一般为.PBK 的文件中。RASENTRY 结构的定义如下：

```
typedefstruct tagRASENTRY
{
    DWORD dwSize;
    DWORD dwfOptions;
    DWORD dwCountryID;
    DWORD dwCountryCode;
    TCHAR szAreaCode[RAS_MaxAreaCode + 1];
    TCHAR szLocalPhoneNumber[RAS_MaxPhoneNumber + 1];
    DWORD dwAlternateOffset;
    RASIPADDR ipaddr;
    RASIPADDR ipaddrDns;
    RASIPADDR ipaddrDnsAlt;
    RASIPADDR ipaddrWins;
    RASIPADDR ipaddrWinsAlt;
    DWORD dwFrameSize;
    DWORD dwfNetProtocols;
    DWORD dwFramingProtocol;
    TCHAR szScript[MAX_PATH];
    TCHAR szAutodialDll[MAX_PATH];
```

```
    TCHAR szAutodialFunc[MAX_PATH];
    TCHAR szDeviceType[RAS_MaxDeviceType + 1];
    TCHAR szDeviceName[RAS_MaxDeviceName + 1];
    TCHAR szX25PadType[RAS_MaxPadType + 1];
    TCHAR szX25Address[RAS_MaxX25Address + 1];
    TCHAR szX25Facilities[RAS_MaxFacilities + 1];
    TCHAR szX25UserData[RAS_MaxUserData + 1];
    DWORD dwChannels;
    DWORD dwReserved1;
    DWORD dwReserved2;
#if(WINVER >= 0x401)
    DWORD dwSubEntries;
    DWORD dwDialMode;
    DWORD dwDialExtraPercent;
    DWORD dwDialExtraSampleSeconds;
    DWORD dwHangUpExtraPercent;
    DWORD dwHangUpExtraSampleSeconds;
    DWORD dwIdleDisconnectSeconds;
#endif
#if(WINVER >= 0x500)
    DWORD dwType;
    DWORD dwEncryptionType;
    DWORD dwCustomAuthKey;
    GUID guidId;
    TCHAR szCustomDialDll[MAX_PATH];
    DWORD dwVpnStrategy;
#endif
#if(WINVER >= 0x501)
    DWORD dwfOptions2;
    DWORD dwfOptions3;
    WCHAR szDnsSuffix[RAS_MaxDnsSuffix];
    DWORD dwTcpWindowSize;
    WCHAR szPrerequisitePbk[MAX_PATH];
    WCHAR szPrerequisiteEntry[RAS_MaxEntryName + 1];
    DWORD dwRedialCount;
    DWORD dwRedialPause;
#endif
}RASENTRY;
```

正如大家看到的那样，上面这个结构由许多字段组成，这里不加以叙述。这些字段的完整描述参考 Platform SDK。在调用 RAS API 时，如果它把一个电话簿文件当作参数(lpszPhonebook)，就可识别出到电话簿文件的路径了。正如我们前面提到的那样，Windows 95、Windows 98、Windows Me 和 Windows CE 系统中，因为电话簿条目是保存在系统注册表内的，所以这个参数必须是 NULL。而在 Windows NT 系统中，这是到一个电话簿文件的路径。通常，这个电话簿文件有一个扩展名.pbk。另

外，在 Windows NT 系统中默认电话簿位于%SystemRoot%\System32\Ras\RasPhone.pbk。如果将电话簿指定为 NULL，使使用系统默认电话簿文件。

15.6.1　添加电话簿条目

RAS 有 4 个函数，允许通过程序对电话簿 RASENTRY 结构进行管理。它们是：RasSetEntry、RasGetEntryProperties、RasRenameEntry 和 RasDeleteEntry。如要建立一个新条目或对一个现成的条目进行修改，可使用 RasSetEntryProperties 函数，它的定义如下：

```
DWORD RasSetEntryProperties(
    LPCTSTR lpszPhonebook,
    LPCTSTR lpszEntry,
    LPRASENTRY lpRasEntry,
    DWORD dwEntryInfoSize,
    LPBYTE lpbDeviceInfo,
    DWORD dwDeviceInfoSize
);
```

lpszPhonebook 参数是一个指针，指向电话簿文件名。lpszEntry 参数是一个指向字符串的指针，这个字符串用于标识已有的或新建的条目。如果 RASENTRY 结构以这个名称存在，其属性就会被修改；否则，便在这个电话簿中建立一个新条目。lpRasEntry 参数是一个指向 RASENTRY 结构的指针。可以把以空字符结束的字符串列表放在 RASENTRY 结构之后，定义备用电话号码。最后一个字符串的结尾是两个连续的空字符串。dwEntryInfoSize 参数是 lpRasEntry 参数中的那个结构的长度(按字节数算)。lpbDeviceInfo 参数是一个指向缓冲区的指针，该缓冲区中包含 TAPI 设备的配置信息。在 Windows 2000 和 Windows NT 平台上，没有使用这个参数，因此，应该设为 NULL。最后一个参数是 dwDeviceInfoSize，它代表 lpbDeviceInfo 缓冲区的长度(按字节数算)。

RasGetEntryProperties 函数，可用于获得一个已有的电话簿条目的属性，或一个新电话簿条目的默认值。它的定义如下：

```
DWORD RasGetEntryProperties(
    LPCTSTR lpszPhonebook,
    LPCTSTR lpszEntry,
    LPRASENTRY lpRasEntry,
    LPDWORD lpdwEntryInfoSize,
    LPBYTE lpbDeviceInfo,
    LPDWORD lpdwDeviceInfoSize
);
```

lpszPhonebook 参数是一个指向电话簿文件名的指针。lpszEntry 参数是指向一个字符串的指针，该字符串标识一个现成电话簿条目。如果把这个参数设为 NULL，lpRasEntry 和 lpbDeviceInfo 参数就会收到一个电话簿条目的默认值。获得这些默认值是非常有用的：如果需要建立一个新 RAS 电话簿

条目，可在调用 RasSetEntryProperties 函数之前，用恰当的系统信息填充 lpRasEntry 和 lpvDeviceInfo 这两个字段。

lpRasEntry 参数是指向一个缓冲区的指针，这个缓冲区是应用程序为接收 RASENTRY 结构而提供的。我们在讨论 RasSetEntryProperties 函数时曾经提过，这个结构的后面可以跟一个以空字符结束的字符串数组，这些字符串为请求的电话簿条目标识备用电话号码。因此，供接收结构用的缓冲区长度应该大于 RASENTRY 结构的长度。如果向它传递一个 NULL 指针，lpdwEntryInfoSize 参数就会收到一个总字节数，这个数是保存 RASENTRY 结构的所有元素和所有的备用电话号码所需要的。lpdwEntryInfoSize 参数是一个指针，指向包含接收缓冲区中字节数的 DWORD，这个缓冲区是应用程序为 lpRasEntry 参数提供的。RasSetEntryProperties 函数结束时，会把 lpdwEntryInfoSize 更新成 lpRasEntry 中实际上收到的字节数。我们强烈建议大家调用这个函数时，把 lpRasEntry 设为 NULL，把 lpdwENtryInfoSize 设为 0，以便获得缓冲区长度的有关信息。一旦有了正确的缓冲区长度，就可以再次调用这个函数，准确无误地获得所有的信息。

lpbDeviceInfo 参数是一个指针，指向一个应用程序提供的缓冲区，这个缓冲区用来接收这个电话簿条目的 TAPI 设备相关信息。如果把它设为 NULL，lpdwDeviceInfoSize 参数就会收到获得该信息所需的字节数。如果使用的是 Windows 2000、Windows NT 或 Windows XP，就应该把 lpbDeviceInfo 参数设为 NULL。最后一个参数是 lpdwDeviceInfoSize，它是一个指向 DWORD 的指针，DWORD 应该被设为字节数，即为 lpbDeviceInfo 提供的缓冲区中包含的字节数。RasGetEntryProperties 函数返回时，lpbDeviceInfoSize 函数就会返回 lpbDeviceInfo 缓冲区内返回的字节数。

下述代码示例演示了一个应用程序应该怎样利用 RasGetEntryProperties 和 RasSetEntryProperties 来建立一个新的电话簿条目。

```
#include <windows.h>
#include <ras.h>
#include <raserror.h>
#include <stdio.h>
void main(void)
{
    DWORD EntryInfoSize = 0;
    DWORD DeviceInfoSize = 0;
    DWORD Ret;
    LPRASENTRY lpRasEntry;
    LPBYTE lpDeviceInfo;
    //获得一个默认电话簿条目的缓冲区大小的信息
    if((Ret = RasGetEntryProperties(NULL,"",NULL,
        &EntryInfoSize,NULL,&DeviceInfoSize))!= 0)
    {
        if(Ret != ERROR_BUFFER_TOO_SMALL)
        {
```

```
            printf("RasGetEntryProperties sizing failed "
                "with error %d \n",Ret);
            return;
        }
    }
    lpRasEntry = (LPRASENTRY)Global_Alloc(GPTR,EntryInfoSize);

    if(DeviceInfoSize == 0)
        lpDeviceInfo = NULL;
    else
        lpDeviceInfo = (LPBYTE)GlobalAlloc(GPTR,DeviceInfoSize);

    //获得默认电话簿条目
    lpRasEntry->dwSize = sizeof(RASENTRY);

    if((Ret = RasGetEntryProperties(NULL,"",lpRasEntry,
        &EntryInfoSize,lpDeviceInfo,&DeviceInfoSize))!= 0)
    {
        printf("RasGetEntryProperties failed with error %d \n",
            Ret);
        return;
    }

//验证新的电话簿名称 "Testentry"

    if((Ret = RasValidateEntryName(NULL,"Testentry"))!=
        ERROR_SUCCESS)
    {
        printf("RasValidateEntryName failed with error %d \n",
            Ret);
        return;
    }

    //安装一个新的电话簿条目 "Testentry"，使用默认属性

    if((Ret = RasSetEntryProperties(NULL,"Testentry",
        lpRasEntry,EntryInfoSize,lpDeviceInfo,
        DeviceInfoSize))!= 0)
    {
        printf("RasSetEntryProperties failed with error %d \n",
            Ret);
        return;
    }
```

15.6.2 删除电话簿条目

删除一个电话簿条目很简单。调用 RasDeleteEntry 函数即可。这个函数的定义如下：

```
DWORD RasDeleteEntry(
    LPCTSTR lpszPhonebook,
    LPCTSTR lpszEntry
);
```

lpszPhonebook 参数是一个指向电话簿文件的指针。lpszEntry 参数是一个字符串，这个字符串代表一个现成的电话簿条目。如果调用这个函数成功，就会返回 ERROR_SUCCESS；否则，就返回 ERROR_INVALID_NAME。

15.6.3 管理用户凭据

RAS 客户机通过 RasDial 利用电话簿条目进行连接时，便将该用户的安全凭据保存下来，并把它们和电话簿条目关联起来。RasGetCredentials、RasSetCredentials、RasGetEntryDialParams 和 RasSetEntryDialParams 这 4 个函数允许对与一个电话条目关联的用户安全凭据进行管理。RasGetCredentials 和 RasSetCredentials 这两个函数是 Windows NT 4 中引入的(也可用于 Windows 2000)。这两个函数取代了 RasGetEntryDialParams 和 RasSetEntryDialParams。由于 RasGetCredentials 和 RasSetCredentials 不能用于 Windows 95、Windows 98、Windows Me 和 Windows CE，只有采用 RasGetEntryDialParams 和 RasSetEntryDialParams，它们可用于所有的平台。

使用 RasGetCredentials 函数，可获得与一个电话簿条目关联的用户凭据。它的定义如下：

```
DWORD RasGetCredentials(
    LPCTSTR lpszPhonebook,
    LPCTSTR lpszEntry,
    LPRASCREDENTIALS lpCredentials
);
```

lpszPhonebook 参数是一个指向电话簿文件的指针。lpszEntry 参数是一个代表现有的电话簿条目的字符串。lpCredentials 参数是一个指针，指向 RASCREDENTIALS 结构，这个结构可接收与指定电话簿条目相关的用户名、密码和域。RASCREDENTIALS 结构的定义如下：

```
typedefstruct {
    DWORD dwSize;
    DWORD dwMask;
    TCHAR szUserName[UNLEN + 1];
    TCHAR szPassword[PWLEN + 1];
    TCHAR szDomain[DNLEN + 1];
}RASCREDENTIALS,*LPRASCREDENTIALS;
```

它的字段定义如下：

- dwSize：指定 RASCREDENTIALS 结构的长度(按字节算)。应该始终将这个字段设为结构的长度。
- dwMask：是一个位掩码字段，通过预先定义的标志，来识别这个结构中接下来的哪 3 个字段是有效的。预先定义标志有：用于 szUserName 的 RASCM_UserName、用于 szPassword 的 RASCM_Password、用于 szDomain 的 RASCM_Domain。还有更多的 dwMask 标志，但这里不再叙述。
- szUserName：是一个以空字符结束的字符串，其中包含用户登录名。
- szPassword：是一个以空字符结束的字符串，其中包含用户密码。
- szDomain：是一个以空字符结束的字符串，其中包含用户登录的域。

若 RasGetCredentials 调用成功，就会返回 0。根据 lpCredentials 结构的 dwMask 字段中设置的标志，应用程序便可知道设置了哪些安全凭据。

RasSetCredentais 函数类似于 RasGetCredentials，但有一点除外，它还允许改变与一个电话簿条目关联的安全凭据。除了多一个 fClearCredentials 参数外，它的其余参数均和 RasGetCredentials 一样。RasSetCredentais 函数的定义如下：

```
DWORD RasSetCredentials(
    LPCTSTR lpszPhonebook,
    LPCTSTR lpszEntry,
    LPRASCREDENTIALS lpCredentials,
    BOOL fClearCredentials
);
```

fClearCredentials 参数是一个布尔运算符，如果设为 TRUE，就会导致 RasSetCredentials 把凭据改为一个空字符串("")值，该凭据是在 lpCredentials 结构中 dwMask 字段内标识的。例如，如果 dwMask 中包含 RASCM_Password 标志，原来保存的密码就会被替换成一个空字符串。如果 RasSetCredentials 函数调用成功，就会返回 0。

另外，还可以用 RasGetEntryDialParams 和 RasSetEntryDialParams 函数来管理与具体电话簿条目相关的用户安全凭据。RasGetEntryDialParams 的定义如下：

```
DWORD RasGetEntryDialParams(
    LPCTSTR lpszPhonebook,
    LPRASDIALPARAMS lprasdialparams,
    LPBOOL lpfPassword
);
```

lpszPhonebook 参数是一个指向电话簿文件名的指针，lprasdialparams 参数是指向一个 RASDIALPARAMS 结构的指针。lpfPassword 参数是一个布尔标志，如果在 lprasdialparams 结构中获

得了用户密码，就会返回 TRUE。

RasSetEntryDialParams 函数用于改变上一次的连接信息，该信息是 RasDial 调用时设置的，与特定的电话簿条目相关。RasSetEntryDialParams 的定义如下：

```
DWORD RasSetEntryDialParams(
    LPCTSTR lpszPhonebook,
    LPRASDIALPARAMS lprasdialparams,
    BOOL fRemovePassword
),
```

lpszPhonebook 和 lprasdialparams 参数与 RasGetEntryDialParams 函数中的前两个参数完全一样。fRemovePassword 参数是一个布尔标志，如果设为 TRUE，则是要求 RasSetEntryDialParams 删除与指定电话簿条目相关的密码，这个条目是在 lprasdialparams 结构中标识的。

15.7 连 接 管 理

RAS 有两个非常有用的函数。通过它们，可获得自己系统上已建立的各个连接的属性。它们是：RasEnumConnections 和 RasGetProjectionInfo。RasEnumConnections 函数可以获取系统上所有可用的活动 RAS 连接。当需要使用 RasGetProjectionInfo 来获取关于一个 RAS 连接的具体信息时，这个函数非常有用。RasGetProjectionInfo 将在本章稍后出现。RasEnumConnections 的定义如下：

```
DWORD RasEnumConnections(
    LPRASCONN lprasconn,
    LPDWORD lpcb,
    LPDWORD lpcConnections
);
```

lprasconn 参数是一个应用程序缓冲区，将收到一个 RASCONN 结构数组。RASCONN 结构的定义如下：

```
typedefstruct _RASCONN
{
    DWORD dwSize;
    HRASCONN hrasconn;
    TCHAR szEntryName[RAS_MaxEntryName + 1];
#if(WINVER > = 0x400)
    TCHAR szDeviceType[RAS_MaxDeviceType + 1];
    TCHAR szDeviceName[RAS_MaxDeviceName + 1];
#endif
#if(WINVER > = 0x401)
    TCHAR szPhonebook[MAX_PATH];
    DWORD dwSubEntry;
```

```
#endif
#if(WINVER >= 0x500)
    GUID guidEntry;
#endif
#if(WINVER >= 0x501)
    DWORD dwSessionId;
    DWORD dwFlags;
    LUID luid;
#endif
}RASCONN;
```

RASCONN 结构中最有用的字段是 hrasconn，这个字段接收最初由 RasDial 创建的连接句柄。可以用这个句柄来从 RasGetProjectionInfo 获取更多的连接信息，这一点将稍后叙述。对 RasEnumConnections 函数而言，需要把它投递到一个够大的缓冲区，使之能够容纳若干个 RASCONN 结构；不然的话，这个函数就会失败，并返回 ERROR_BUFFER_TOO_SMALL 错误。此外，缓冲区中，第 1 个 RASCONN 结构中的 dwSize 字段必须被设成一个 RASCONN 结构的字节长。下一个参数 lpcb 是一个指向一个变量的指针，必须把这个变量设为自己的 lprasconn 数组的长度(按字节算)。这个函数返回时，lpcb 就会包含列举所有连接时所需要字节数量。即使没有提供足够大的缓冲区，也可以用 lpcb 中返回的正确的缓冲区长度再试一次。lpcConnections 参数是一个变量指针，接收被写成 lprasconn 的 RASCONN 结构的个数。

有了取自 RasEnumConnections 和 RasGetSubEntryHandle 的 RAS 连接句柄，就可获得网络协议特有的信息，该信息用于一个已建立的 RAS 连接。这就是人们所说的**投影信息**(Projection information)。远程访问服务器利用投影信息来表示网络上的一个远程客户机。比方说，在一个组帧协议上建立一个连接时，这个连接采用的是 IP 协议，IP 配置信息(例如分配得到的 IP 地址)就会从 RAS 服务到客户机之间建立起来。要想获得 PPP 组帧协议上的协议投影信息，调用 RasGetProjectionInfo 函数即可，该函数的定义如下：

```
DWORD RasGetProjectionInfo(
    HRASCONN hrasconn,
    RASPROJECTION rasprojection,
    LPVOID lpprojection,
    LPDWORD lpcb
);
```

brasconn 参数是一个 RAS 连接句柄。rasprojection 参数是一个 RASPROJECTION 枚举类型，它允许指定一个协议来接收连接信息。lpprojection 参数接收一个和 rasprojection 中指定的枚举类型关联的数据结构。最后一个参数 lpcb 是一个变量指针，必须把它设为自己的 lpprojection 结构的长度。这个函数完成时，该变量中会包含获得投影信息所需要的缓冲区的长度。

下面的 RASPROJECTION 枚举类型能够取得连接信息：

- RASP_Amb
- RASP_PppNbf
- RASP_PppIpx
- RASP_PppIp
- RASP_PppIpx
- RASP_PppLcp
- RASP_Slip

要为一个 PPP 组帧协议连接获取 IP 地址信息，必须指定 RASP_PppIp 枚举类型。如果把 RASP_PppIpx 枚举类型指定为 RasGetProjectionInfo，收到的则是一个 RASPPPIPX 结构，该结构的定义如下：

```
typedef struct _RASPPPIP {
    DWORD dwSize;
    DWORD dwError;
    TCHAR szIpAddress[ RAS__MaxIpAddress + 1];
#ifndef WINNT35COMPATIBLE
    TCHAR szServerIpAddress[ RAS__MaxIpAddress + 1];
#endif
#if (WINVER >= 0x500)
    DWORD dwOptions;
    DWORD dwServerOptions;
#endif
}RASPPPIP;
```

它的字段定义如下：

- dwSize：应该设为一个 RASPPPIPX 结构的长度(按字节算)。
- dwError：从 PPP 协商进程中取得一个错误代码。
- szIpxAddress：取得一个字符串，该字符串代表客户机的 IPX 地址。
- szServerIpAddress：取得一个代表服务器 IP 地址的字符串。
- dwOptions：本地主机的压缩选项。
- dwServerOptions：远程主机的压缩选项。

下述代码展示了当在 RAS 之上建立了一个 PPP 连接时，如何调用 RasGetProjectionInfo 来获取分配到一个客户机的 IP 地址：

```
lpProjection = (RASPPPIP *)GlobalAlloc(GPTR,cb);
lpProjection->dwSize = sizeof(RASPPPIP);
cb = sizeof(RASPPPIP);

Ret = RasGetProjectionInfo(hRasConn,RASP_PppIp,
lpProjection,&cb);
```

```
if(Ret != ERROR_SUCCESS)
{
    printf("RasGetProjectionInfo failed with error %d",Ret);
    return;
}
else
{
    printf("\nRas Client IP address:%s \n",
        lpProjection->szIpAddress);
    printf("Ras Server IP address:%s \n",
        lpProjection->szServerIpAddress);
}
```

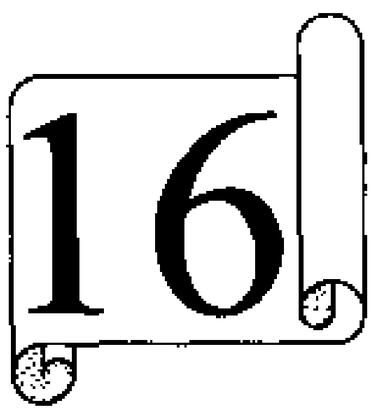
15.8　VPN

前面提到，RAS 也可以用来在一个 IP 网络之上创建安全的 VPN 连接。只要通过 RAS 或一个网络适配器在计算机上建立了 IP 连接，就可以这样做。现在有些计算机建立 IP 连接时，要么通过 DSL(Digital Subscriber Line，数字用户线)，要么通过电缆调制解调器，而不会使用 RAS 串行设备来连接到 Internet。如果没有建立 IP 连接，则可以先使用 RasDial 来在一个串行设备上建立 IP 连接，如前所述。一旦通过 RAS 或网络适配器建立了 IP 连接，就可以使用 RasDial 在已有 IP 连接上，建立一个到远程网络的安全链接。

使用 RAS 客户机来建立 VPN 连接，需要建立一个电话簿条目，该条目包含远程网络中远程 VPN 服务器的 IP 地址信息，而非电话拨号信息。在电话簿的 RASENTRY 结构中，必须将字段设为字符串“RASDT_Vpn”。设置了这个字段，就可以用 szLocalPhoneNumber 字段来指定远程 VPN 服务器的一个 DNS 名称或 IP 地址串，而非电话号码了。建立电话簿条目后，就可以使用电话簿条目来调用 RasDial，以建立 VPN 链接。

15.9　小　　结

本章介绍了利用 RAS 来扩展计算机连网能力的基础知识。还详细地说明了如何调用 RasDial 函数与远程网络进行通信。另外，我们还讨论了如何利用建立电话簿条目来充分挖掘 RAS 的潜力。



IP 助手函数

本章介绍一些新的 API 函数，有了这些函数，便可在自己的计算机上对 IP 协议特征进行查询和管理。它们将帮助您以编程方式获取下面的标准 IP 实用程序中可用的功能：

- IPCONFIG.EXE(或适用于微软 Windows 95、Windows 98 和 Windows Me 的 WINIPCFG.EXE)：显示 IPv4 配置信息，允许释放和更新 DHCP 分配的 IPv4 地址。
- IPV6.EXE：一个新引入的 API，用于列举 IPv6 地址，和 IPV6.EXE 或 NETSH.EXE 命令类似。这个实用程序将简单地被冠以名称 IPCONFIGV6.EXE。
- NETSTAT.EXE：显示 TCP 连接表、UDP 监听者表以及 IP 协议统计情况。
- ROUTE.EXE：显示并处理网络路由表。
- ARP.EXE：显示并修改供 ARP 使用的 IP 到物理地址转换表。

本章介绍的这些函数主要用于 Windows 98、Windows Me、Windows 2000 和 Windows XP 操作系统。有几个还可以用于 Windows NT SP4 及以后的 SP(服务包)版本，但所有这些函数都不能用于 Windows 95。接下来的讨论中，我们将一一指出各个函数适用于哪些平台。本附录中的所有函数原型均定义在 IPTYPES.H 文件中。在建立自己的应用程序时，必须把它链接到这个库文件 IPHLPAPI.LIB。

为本章提供的示例在补充材料的 SAMPLES\CHAPTER16 目录下，这个示例模拟了知名的系统实用程序。

应注意到，IP 助手 API 是在 Windows 平台支持 IPv6 之前开发的。因此，所有的 API 都返回关于 IPv4 的信息，惟一的例外是一个新的 IP 助手 API，即 GetAdaptersAddresses，下一节将对它进行讨论。

16.1 Ipconfig

IPCONFIG.EXE 程序展示了两条信息：IPv4 配置信息和 IPv4 配置参数，这些参数对应安装在计算机上的每个网络适配器。要获得 IPv4 配置信息，利用 GetNetworkParams 函数即可。它的定义如下：

```
DWORD GetNetworkParams(
PFIXED_INFO pFixedInfo,
PULONG pOutBufLen
);
```

pFixedInfo 参数取得一个缓冲区指针，该缓冲区接收 FIXED_INFO 数据结构，应用程序必须提供这个结构，以便获得 IPv4 配置信息。pOutBufLen 参数是一个变量指针，指定传递到 pFixedInfo 参数中的那个缓冲区的长度。如果提供的缓冲区不够大，GetNetworkParams 就会返回 ERROR_BUFFER_OVERFLOW 错误，并将 pOutBufLen 参数设为正确的缓冲区长度。

GetNetworkParams 中所用的 FIXED_INFO 结构定义如下：

```
typedef struct
{
    char                 HostName[MAX_HOSTNAME_LEN + 4];
    char                 DomainName[MAX_DOMAIN_NAME_LEN + 4];
    PIP_ADDR_STRING   CurrentDnsServer;
    IP_ADDR_STRING    DnsServerList;
    UINT                 NodeType;
    char                 ScopeId[MAX_SCOPE_ID_LEN + 4];
    UINT                 EnableRouting;
    UINT                 EnableProxy;
    UINT                 EnableDns;
}FIXED_INFO,*PFIXED_INFO;
```

它的各个字段定义如下：

- HostName：代表计算机名，这个名字由 DNS 进行识别。
- DomainName：说明计算机属于哪个 DNS 域。
- CurrentDnsServer：包含当前 DNS 服务器的 IPv4 地址。
- DnsServerList：是一个链接列表，其中包含计算机采用的 DNS 服务器。
- NodeType：说明 IPv4 网络上的系统是如何解析 NetBIOS 名的。表 16.1 包含了该字段可能的值。
- ScopeId：标识一个字符串值。这个值加在 NetBIOS 名中，通过逻辑方式把两个或两个以上的计算机组合在一起，在 TCP/IP 网络中进行通信。
- EnableRouting：说明系统是否会在它连接的网络中路由 IPv4 包。
- EnableProxy：说明系统是否充当网络上的 WINS 代理。WINS 代理通过 WINS，响应它已解析过的广播名字查询，并允许由 b 节点计算机组成的网络与其他已经用 WINS 注册的子网上的服务器建立连接。
- EnableDns：说明 NetBIOS 是否向 DNS 查询 WINS 不能解析的名称、广播或 LMHOST 文件。

表 16.1 节点类型可能的值

值	说 明
BROADCAST_NODETYPE	即 b 节点 NetBIOS 名称解析法，采用了这种解析方法，系统便利用 IP 广播来执行 NetBIOS 名称注册和名称解析
PEER_TO_PEER_ODETYPE	即 p 节点名称解析，采用了这种解析方法，系统便利用点到点通信与一个 NetBIOS 名称服务器(例如 WINS)通信，进而对 IP 地址进行注册和计算机名的解析
MIXED_NODETYPE	即 m 节点(mixed 节点)NetBIOS 名称解析法，采用了这种解析法，系统便同时采用前面的 b 节点和 p 节点方法。先用 b 节点方法；如果失败，再用 p 节点方法
HYBRID_NODETYPE	即 h 节点(hybrid 节点)NetBIOS 名称解析法，采用了这种解析法，系统便同时采用前面所讲的 b 节点和 p 节点。先用 p 节点；如果失败，再用 b 节点

FIXED_INFO 结构的 DnsServerList 字段是一个 IP_ADDR_STRING 结构，代表 IPv4 地址链接列表的起始处。这个结构定义为：

```
typedef struct _IP_ADDR_STRING
{
    struct _IP_ADDR_STRING* Next;
    IP_ADDRESS_STRING    IpAddress;
    IP_MASK_STRING       IpMask;
    DWORD                Context;
}IP_ADDR_STRING,*PIP_ADDR_STRING;
```

Next 字段标识列表中的下一个 DNS 服务器的 IPv4 地址。如果把它设为 NULL，它就表示列表的结尾。IpAddress 字段是一个字符串，以点分十进制字符串表示 IPv4 地址。IPMask 字段也是一个字符串，表示子网掩码，这个掩码与 IpAddress 中列出的 IPv4 地址关联在一起。最后一个字段是 Context，它用一个系统中惟一的值来标识 IPv4 地址。

另外，利用 IPCONFIG.EXE 程序，也可获得网络接口特有的 IP 配置信息。网络接口不仅可以是一个硬件以太网适配器，甚至还可以是一个 RAS 拨号适配器。调用 GetAdaptersInfo 命令，便可获得适配器信息。该函数的定义如下：

```
DWORD GetAdaptersInfo (
    PIP_ADAPTER_INFO pAdapterInfo,
    PULONG pOutBufLen
);
```

可以用 pAdapterInfo 参数来把一个指针投递给应用程序提供的缓冲区，这个缓冲区接收一个 ADAPTER_INFO 数据结构，该结构中含有这个适配器的配置信息。pOutBufInfo 参数是一个变量指针，这个变量指定传到 pAdapterInfo 参数中的缓冲区的长度。如果提供的缓冲区不够大，GetAdaptersInfo 就会返回 ERROR_BUFFER_OVERFLOW，并把 pOutBufLen 参数设为所需要的缓冲区长度。

事实上，IP_ADAPTER_INFO 结构是一个结构列表，计算机上所有可用的网络适配器的 IPv4 配置信息，都在这个列表中。IP_ADAPTER_INFO 的定义如下：

```
typedef struct _IP_ADAPTER_INFO
{
    struct _IP_ADAPTER_INFO *Next;
    DWORD                    ComboIndex;
    char                     AdapterName[MAX_ADAPTER_NAME_LENGTH + 4];
    char                     Description[MAX_ADAPTER_DESCRIPTION_LENGTH + 4];
    UINT                    AddressLength;
    BYTE                    Address[MAX_ADAPTER_ADDRESS_LENGTH];
    DWORD                   Index;
    UINT                     Type;
    UINT                     DhcpEnabled;
    PIP_ADDR_STRING         CurrentIpAddress;
    IP_ADDR_STRING          IpAddressList;
    IP_ADDR_STRING          GatewayList;
    IP_ADDR_STRING          DhcpServer;
    BOOL                    HaveWins;
    IP_ADDR_STRING          PrimaryWinsServer;
    IP_ADDR_STRING          SecondaryWinsServer;
    time_t                     LeaseObtained;
    time_t                     LeaseExpires;
}IP_ADAPTER_INFO,*PIP_ADAPTER_INFO;
```

各字段定义如下：

- Next：缓冲区内的下一个适配器。NULL 值表示列表的结尾。
- ComboIndex：未用，将被设为 0。
- AdapterName：适配器名。
- Description：是一个关于适配器的简单说明。
- AddressLength：指明 Address 字段中的那个适配器的物理地址由多少个字节组成。
- Address：该适配器的物理地址。
- Index：该适配器分配的惟一的网络接口内部索引编号。
- Type：将适配器类型指定为一个数值。表 16.2 定义了已获支持的适配器类型。所支持的适配器类型的完整列表可在 IPIFCONS.H 中找到。

表 16.2 适配器类型

适配器类型的值	说 明
MIB_IF_TYPE_ETHERNET	以太网适配器
MIB_IF_TYPE_FDDI	FDDI(Fiber Distributed Data Interface，光纤分布数据接口)适配器
MIB_IF_TYPE_LOOPBACK	回叫适配器
MIB_IF_TYPE_OTHER	其他类型的适配器
MIB_IF_TYPE_PPP	PPP 适配器
MIB_IF_TYPE_SLIP	Slip 适配器
MIB_IF_TYPE_TOKENRING	令牌环适配器

- DhcpEnabled：说明这个适配器上是否启用 DHCP。
- CurrentIpAddress：未用，将被设为 NULL 值。
- IpAddressList：指定为这个适配器分配的 IPv4 地址列表。
- GatewayList：指定代表默认网关的 IPv4 地址列表。
- DhcpServer：指定一个列表，表中只有一个元素，代表 DHCP 服务器所用的 IPv4 地址。
- HaveWins：说明该适配器是否使用 WINS 服务器。
- PrimaryWinsServer：指定一个列表，表中只有一个元素，代表主 WINS 服务器所用的 IPv4 地址。
- SecondaryWinsServer：指定一个列表，表中只有一个元素，代表辅助 WINS 服务器所用的 IPv4 地址。
- LeaseObtained：标识何时开始租用 DHCP 服务器提供的 IPv4 地址。
- LeaseExpires：标识 DHCP 服务器提供的 IPv4 地址何时到期。

GetAdaptersInfo 返回关于物理适配器及分配给它的 IPv4 地址的大量信息，但不返回任何 IPv6 信息。一个新的 API 即 GetAdaptersAddresses 已被引入来填补这一空隙，这个 API 返回 IPv4 及 IPv6 的地址信息，定义为：

```
DWORD WINAPI GetAdaptersAddresses(
    ULONG Family,
    DWORD Flags,
    PVOID Reserved,
    PIP_ADAPTER_ADDRESSES pAdapterAddresses,
    PULONG pOutBufLen
);
```

Family 参数指明应该被列举的地址族，有效值为：AF_INET、AF_INET6 或 AF_UNSPEC，这取决于想要 IPv4 信息、IPv6 信息还是所有的 IP 信息。Flags 参数控制返回地址的类型。表 16.3 列出了可能的值及其含义。应注意到，可以通过对多个标志一起“或”操作来指定多个标志。在默认的情况下，所有的地址都会被返回。最后两个参数是缓冲区及缓冲区长度，IP 信息将返回到缓冲区内。

表 16.3　GetAdaptersAddresses 标志

标　志	说　明
GAA_FLAG_SKIP_UNICAST	排除单播地址
GAA_FLAG_SKIP_ANYCAST	排除任播地址
GAA_FLAG_SKIP_MULTICAST	排除多播地址
GAA_FLAG_SKIP_DNS_SERVER	排除 DNS 服务器地址

IP 信息以一个 IP_ADAPTER_ADDRESSES 结构的形式返回。这个结构的定义为：

```
typedef struct _IP_ADAPTER_ADDRESSES {
    union {
        ULONGLONG Alignment;
        struct {
            ULONG Length;
            DWORD IfIndex;
        }
    }
    struct _IP_ADAPTER_ADDRESSES *Next;
    PCHAR AdapterName;
    PIP_ADAPTER_UNICAST_ADDRESS     FirstUnicastAddress;
    PIP_ADAPTER_ANYCAST_ADDRESS     FirstAnycastAddress;
    PIP_ADAPTER_MULTICAST_ADDRESS   FirstMulticastAddress;
    PIP_ADAPTER_DNS_SERVER_ADDRESS  FirstDnsServerAddress;
    PWCHAR DnsSuffix;
    PWCHAR Description;
    PWCHAR FriendlyName;
    BYTE    PhysicalAddress[MAX_ADAPTER_ADDRESS_LENGTH];
    DWORD  PhysicalAddressLength;
    DWORD  Flags;
    DWORD  Mtu;
    DWORD  IfType;
    IF_OPER_STATUS OperStatus;
    DWORD  Ipv6IfIndex;
    DWORD  ZoneIndices[16];
    PIP_ADAPTER_PREFIX FirstPrefix;
}IP_ADAPTER_ADDRESSES,*PIP_ADAPTER_ADDRESSES;
```

- Length：指定结构长度。
- IfIndex：指定接口索引，可被 GetAdaptersInfo 返回的接口索引交叉引用。
- Next：指定下一个返回的 IP_ADAPTER_ADDRESSES 结构。
- AdapterName：指定这些地址分配到的适配器名。
- FirstUnicastAddress：指向一个 IP_ADAPER_UNICAST_ADDRESS 结构列表的指针，这些结构包含分配到这个适配器的每个单播地址信息。
- FirstAnycastAddress：指向一个 IP_ADAPER_ANYCAST_ADDRESS 结构列表的指针，这些结构包含分配到这个适配器的每个任播地址信息。
- FirstMulticastAddress：指向一个 IP_ADAPER_MULTICAST_ADDRESS 结构列表的指针，这些结构包含分配到这个适配器的每个任播地址信息。这极为有用，因为它列出了所有在每个物理接口上加入的多播地址。
- FirstDnsServerAddress：指向一个 IP_ADAPER_DNS_SERVER_ADDRESS 结构列表的指针，这些结构包含分配到这个适配器的每个 DNS 服务器信息。
- DnsSuffix：指定和这个适配器关联的 DNS 后缀 UNICODE 字符串。
- Description：包含一个适配器的 UNICODE 字符串说明。
- FriendlyName：包含一个适配器的 UNICODE 字符串说明，但比 Description 字段更容易读懂。
- PhysicalAddress：用字节数组指定适配器的物理地址。如果是以太网适配器，它将指定 MAC 地址。
- PhysicalAddressLength：指定组成物理地址的字节数， PhysicalAddress 字段中包含该物理地址。
- Flags：指定针对 DDNS、DNS 及 DHCP 的适配器状态。表 16.4 列出了可能的标志。

表 16.4 IP_ADAPTERS_ADDRESSES 标志

标 志	说 明
IP_ADAPTER_DDNS_ENABLED	这个适配器上启用了动态 DNS
IP_ADAPTER_REGISTER_ADAPTER_ SUFFIX	这个适配器的 DNS 后缀已注册
IP_ADAPTER_DHCP_ENABLED	这个适配器上启用了 DHCP
IP_ADAPTER_RECEIVE_ONLY	接口只能接收数据
IP_ADAPTER_NO_MULTICAST	接口不能接收多播数据
IP_ADAPTER_IPV6_OTHER_STATEFUL_CONFIG	指明在上一次收到的 IPv6 路由器广告中设置了“O”比特。这表明出现了诸如 DHCPv6 之类的状态性配置信息

- Mtu：指定适配器支持的最大传输单元。
- IfType：指定适配器类型，可能的类型参见表 16.2。
- OperStatus：指定适配器操作状态。关于这个字段的更多内容参见 RFC2863。
- Ipv6IfIndex：指定分配到这个接口的 IPv6 地址的适配器接口索引。应注意到，这个字段应该和 IPv6 地址一起使用，而非 IfIndex 字段。
- ZoneIndices：指定 16 个不同范围级别的范围 ID。最通用的范围级别是在 SCOPE_LEVEL 枚举类型中定义的。更多内容参考 RFC2373。
- FirstPrefix：结构的一个链表，这个结构包含了网前缀，这些前缀是这个接口的链接前提。

IP_ADAPTERS_ADDRESSES 结构的最后一个部分是适配器结构。这 4 个结构非常相似，它们的大部分包含了同一种信息。因为其余的地址结构可以根据这个结构的单播版本进行推断，所以这里只讨论这个单播版本。单播结构的定义为：

```
typedef struct _IP_ADAPTER_UNICAST_ADDRESS {
    union {
        ULONGLONG Alignment;
        struct {
            ULONG Length;
            DWORD Flags;
        };
    };
    struct _IP_ADAPTER_UNICAST_ADDRESS *Next;
    SOCKET_ADDRESS     Address;
    IP_PREFIX_ORIGIN     PrefixOrigin;
    IP_SUFFIX_ORIGIN     SuffixOrigin;
    IP_DAD_STATE         DadState;
    ULONG               ValidLifetime;
    ULONG               PreferredLifeTime;
    ULONG               LeaseLifeTime;
}IP_ADAPTER_UNICAST_ADDRESS,*PIP_ADAPTER_UNICAST_ADDRESS;
```

- Length：指定结构的长度，以字节表示。
- Flags：指定地址类型。表 16.5 包含了可能的标志值。

表 16.5　单地址标志

单地址标志	说　明
IP_ADAPTER_ADDRESS_DNS_ELIGIBLE	地址可以用 DNS 注册(如分配的 DHCP 或 RA)
IP_ADAPTER_ADDRESS_TRANSIENT	地址不是一个永久地址(如 IPv6 专用地址)

- Next：指定链表中的下一个地址结构。
- PrefixOrigin：指定网络前缀的获取方法。表 16.6 列出了可能的值，这些值是 IPTYPES.H 中

定义的一个枚举类型。

- SuffixOrigin：指定获取地址主机部分的方法。表 16.7 列出了这些值，也是一个枚举类型。

表 16.6 前缀初始值

前缀初始值	说 明
IpPrefixOriginOther	前缀从这个表中所列资源之外的其他地方获取
IpPrefixOriginManual	前缀为手动配置——例如，分配一个静态 IPv4 地址
IpPrefixOriginWellKnown	前缀是一个固定地址——例如环回地址
IpPrefixOriginDhcp	前缀由 DHCP 分配
IpPrefixOriginRouterAdvertisement	前缀由一个路由器广告分配——例如一个 IPv6 地点一本地或全球地址

表 16.7 后缀初始值

后缀初始值	说 明
IpSuffixOriginOther	后缀从这个表中所列资源之外的其他地方获取
IpSuffixOriginManual	后缀为手动配置——例如，一个静态分配的 IP 地址
IpSuffixOriginWellKnown	后缀是一个知名地址——例如环回地址
IpSuffixOriginDhcp	后缀由 DHCP 分配
IpSuffixOriginLinkLayerAddress	后缀从下层网络层获取。例如 IPv6 链接一本地地址
IpSuffixOriginRandom	后缀是一个随机分配的值。例如 IPv6 专用地址

- DadState：指明地址的状态。DAD(Duplicate address detection，重复地址检测)是验证地址的一个过程。表 16.8 列出了取值及其含义。

表 16.8 地址状态

地址状态	说 明
IpDadStateInvalid	地址正在被删除
IpDadStateTentative	重复地址检测正在进行
IpDadStateDuplicate	一个重复地址已被删除
IpDadStateDeprecated	地址不再为新连接优先选用

续表

地址状态	说　明
IpDatStatePreferred	地址为优先选用地址

- ValidLifeTime：指定地址的有效期还有多久，用秒表示。值为 0xFFFFFFFF 时表明地址还未过期。
- PreferredLifetime：指定地址为优先选用地址的时间还有多久，用秒表示。优先选用时期结束后，地址就会进入非优先选用状态。值为 0xFFFFFFFF 时表明地址作为优先选用地址还未过期。
- LeaseLifeTime：指定 DHCP 租用有效期还有多久，用秒表示。值为 0xFFFFFFFF 时表明租用还未过期。

16.1.1　释放和更新 IPv4 地址

IPCONFIG.EXE 程序的另一个特色是：能够释放和更新从 DCHP 服务器处获得的 IPv4 地址。这是通过指定/release 和/renew 命令行参数来完成的。如果想通过编程释放 IPv4 地址，则可 IPReleaseAddress 函数，该函数定义如下：

```
DWORD IpReleaseAddress (
    PIP_ADAPTER_INDEX_MAP AdapterInfo
);
```

如果打算更新 IP 地址，则调用 IPRenewAddress 函数，它的定义如下：

```
DWORD IpRenewAddress (
    PIP_ADAPTER_INDEX_MAP AdapterInfo
);
```

这两个函数均突出了 AdapterInfo 参数，这个参数是一个 IP_ADAPTER_INDEX_MAP 结构，该结构用来标识即将对其地址进行释放和更新的适配器。IP_ADAPTER_INDEX_MAP 结构定义如下：

```
typedef struct _IP_ADAPTER_INDEX_MAP
{
    ULONG Index;
    WCHAR Name[MAX_ADAPTER_NAME];
}IP_ADAPTER_INDEX_MAP,*PIP_ADAPTER_INDEX_MAP;
```

它的各个字段定义如下：

- Index：标识为适配器分配的内部网络接口索引。
- Name：标识适配器名。

调用 GetInterfaceInfo 函数，便可获得某一特定适配器的 IP_ADAPTER_INDEX_MAP 结构。该函

数的定义如下：

```
DWORD GetInterfaceInfo (
IN PIP_INTERFACE_INFO pIfTable,
OUT PULONG dwOutBufLen
);
```

pIfTable 参数是一个指针，指向一个 IP_INTERFACE_INFO 应用程序缓冲区，该缓冲区将接收接口信息。dwOutBufLen 参数是一个变量指针，这个变量指定传递到 pIfTable 参数中的缓冲区的长度。如果这个缓冲区内容纳不下这个接口信息，GetInterfaceInfo 便返回 ERROR_INSUFFICIENT_BUFFER 错误，并把 dwOutBufLen 参数设为合适的缓冲区长度。

IP_INTERFACE_INFO 结构的定义如下：

```
typedef struct _IP_INTERFACE_INFO
{
    LONG                        NumAdapters;
    IP_ADAPTER_INDEX_MAP  Adapter[1];
}IP_INTERFACE_INFO,*PIP_INTERFACE_INFO;
```

它的各个字段定义如下：

- NumAdapters：Adapter 字段中的适配器编号。
- Adapter：是一个 IP_ADAPTER_INDEX_MAP 结构(见前面的定义)的数组。

一旦获得了特定适配器的 IP_ADAPTER_INDEX_MAP 结构，便可利用前面的 IPReleaseAddress 和 IPRenewAddress 这两个函数，释放或更新 DHCP 分配的 IPv4 地址了。

16.1.2 改变 IPv4 地址

IPCONFIG.EXE 程序不允许改变网络适配器的 IP 地址(DHCP 分配的除外)。但是，有两个函数允许增添或删除特定适配器的 IP 地址，它们是：AddIpAddress 和 DeleteIpAddress IP 助手函数。使用这两个函数时，需要知道网络适配器的索引编号和 IP 上下文编号。Windows 中，每个网络适配器都有一个惟一的索引 ID(前面已讲过)，而且，每个 IP 地址都有一个惟一的上下文 ID。适配器索引 ID 和 IP 上下文编号都可通过 GetAdaptersInfo 获得。AddIpAddress 函数的定义如下：

```
DWORD AddIPAddress (
    IPAddr  Address,
    IPMask  IpMask,
    DWORD IfIndex,
    PULONG NTEContext,
    PULONG NTEInstance
);
```

Address 参数把准备增添的 IPv4 地址指定为一个无符号的长整数值。IpMask 参数把 IP 地址的子

网掩码指定为一个无符号的长整数值。IfIndex 参数指定准备增添地址的适配器索引。NTEContext 参数取得与所增添的 IPv4 地址相关联的上下文取值。NTEInstance 参数取得与一个 IPv4 地址相关联的实例值。

如果想通过编程删除适配器的 IPv4 地址，调用 DeleteIpAddress 函数即可。它的定义如下：

```
DWORD DeleteIPAddress (
    ULONG NTEContext
);
```

NTEContext 参数是与 IPv4 地址关联的值，这个值可从 GetAdaptersInfo(前面已介绍)中得到。

应注意到，通过 AddIpAddress 函数添加的 IPv4 地址将保持不变，直到重启为止。

16.2　Netstat

NETSTAT.EXE 程序显示了计算机上的 TCP 连接表、UDP 监听者表以及 IP 协议统计。用于获得这一信息的函数可用于 Windows NT4 SP4(以及稍后的版本)、Windows 98 和 Windows Me。

16.2.1　取得 TCP 连接表

利用 GetTcpTable 函数，可获得 TCP 连接表。获得的信息和带上-p tcp –a 选项执行 NETSTAT.EXE 程序时看到的信息一样。GetTcpTable 函数的定义如下：

```
DWORD GetTcpTable(
    PMIB_TCPTABLE pTcpTable,
    PDWORD pdwSize,
    BOOL bOrder
);
```

pTcpTable 参数是一个指针，指向一个 MIB_TCPTABLE 应用程序缓冲区，该缓冲区接收 TCP 连接信息。pdwsize 参数是一个变量指针，这个变量指定传递到 pTcpTable 参数中的缓冲区长度。如果提供的缓冲区容纳不下这个 TCP 信息，函数就会把这个参数设为合适的缓冲区长度。bOrder 参数指定是否对返回信息进行分类。

GetTcpTable 函数返回的 MIB_TCPTABLE 结构的定义如下：

```
typedef struct _MIB_TCPTABLE
{
    DWORD dwNumEntries;
    MIB_TCPROW table[ANY_SIZE];
}MIB_TCPTABLE,*PMIB_TCPTABLE;
```

它的各个字段定义如下：

- dwNumEntries：说明 table 字段中有多少个条目。
- table：是一个 MIB_TCPROW 结构数组，其中包含 TCP 连接信息。

MIB_TCPROW 结构中包含构成一个 TCP 连接的 IPv4 地址对。这个结构的定义为：

```
typedef struct _MIB_TCPROW
{
    DWORD dwState;
    DWORD dwLocalAddr;
    DWORD dwLocalPort;
    DWORD dwRemoteAddr;
    DWORD dwRemotePort;
}MIB_TCPROW,*PMIB_ICPROW;
```

它的各个字段定义如下：

- dwState：指定 TCP 连接的状态，参见表 16.9 中的说明。关于 TCP 状态的更多内容参见第 1 章。

表 16.9 TCP 连接的状态

连接状态	RFC 说明
MIB_TCP_STATE_CLOSED	“关闭”状态
MIB_TCP_STATE_CLOSING	“正在关闭”状态
MIB_TCP_STATE_CLOSE_WAIT	“关闭等待”状态
MIB_TCP_STATE_DELETE_TCB	“删除”状态
MIB_TCP_STATE_ESTAB	“已建立”状态
MIB_TCP_STATE_FIN_WAIT1	“FIN wait1”状态
MIB_TCP_STATE_FIN_WAIT2	“FIN wait2”状态
MIB_TCP_STATE_LAST_ACK	“最后一次确认”状态
MIB_TCP_STATE_LISTEN	“正在监听”状态
MIB_TCP_STATE_SYN_RCVD	“同步接收”状态
MB_TCP_STATE_SYN_SENT	“同步发送”状态
MIB_TCP_STATE_TIME_WAIT	“时间等待”状态

- dwLocalAddr：为连接指定一个本地 IPv4 地址。
- dwLocalPort：为连接指定一个本地端口。
- dwRemoteAddr：为连接指定远程 IPv4 地址。
- dwRemotePort：为连接指定远程端口。

16.2.2　取得 UDP 监听者表

利用 GetUdpTable 函数，可获得 UDP 监听者表。获得的信息和带-p udp -a 选项的执行 Netstat.exe 程序时看到的信息一样。GetUdpTable 函数的定义如下：

```
DWORD GetUdpTable(
    PMIB_UDPTABLE pUdpTable,
    PDWORD pdwSize,
    BOOL bOrder
);
```

pUdpTable 参数是一个指针，指向 MIB_UDPTABLE 应用程序缓冲区，该缓冲区将接收 UDP 监听者信息。pdwSize 参数是一个变量指针，这个变量指定传递到 pUdpTable 参数内的缓冲区的长度。如果缓冲区容纳不下这个 UDP 信息，函数便把这个参数设为合适的缓冲区长度。bOrder 参数指定是否对返回信息进行分类。

GetUdpTable 函数返回的 MIB_UDPTABLE 结构的定义如下：

```
typedef struct _MIB_UDPTABLE
{
    DWORD dwNumEntries;
    MIB_UDPROW table[ANY_SIZE];
}MIB_UDPTABLE,*PMIB_UDPTABLE;
```

它的各个字段的定义如下：

- dwNumEntries：说明 table 字段中有多少条目。
- table：是一个 MIB_UDPROW 结构数组，其中包含 UDP 监听者的信息。

MIB_UDPROW 结构中包含 IPv4 地址，UDP 正在这些地址上监听数据报。这个结构的定义为：

```
typedef struct _MIB_UDPROW
{
    DWORD dwLocalAddr;
    DWORD dwLocalPort;
}MIB_UDPROW,*PMIB_UDPROW;
```

该结构的各个字段的定义如下：

- dwLocalAddr：指定本地 IPv4 地址。

- dwLocalPort：指定本地端口。

16.2.3 获取 IP 协议统计情况

用于获取 IP 统计情况的 4 个函数是：GetIpStatistics、GetIcmpStatistics、GetTcpStatistics 以及 GetUdpStatistics。这些函数产生的信息和带上-s 选项执行 Netstat.exe 程序时返回的信息一样。利用第 1 个统计函数 GetIpStatistics，可获得当前计算机上的 IP 统计情况，它的定义如下：

```
DWORD GetIpStatistics(
    PMIB_IPSTATS pStats
);
```

pStats 参数是一个指针，指向 MIB_IPSTATA 结构，这个结构接收计算机当前的 IPv4 统计情况。MIB_IPSTATS 结构的定义如下：

```
typedef struct _MIB_IPSTATS
{
    DWORD dwForwarding;
    DWORD dwDefaultTTL;
    DWORD dwInReceives;
    DWORD dwInHdrErrors;
    DWORD dwInAddrErrors;
    DWORD dwForwDatagrams;
    DWORD dwInUnknownProtos;
    DWORD dwInDiscards;
    DWORD dwInDelivers;
    DWORD dwOutRequests;
    DWORD dwRoutingDiscards;
    DWORD dwOutDiscards;
    DWORD dwOutNoRoutes;
    DWORD dwReasmTimeout;
    DWORD dwReasmReqds;
    DWORD dwReasmOks;
    DWORD dwReasmFails;
    DWORD dwFragOks;
    DWORD dwFragFails;
    DWORD dwFragCreates;
    DWORD dwNumIf;
    DWORD dwNumAddr;
    DWORD dwNumRoutes;
}MIB_IPSTATS,*PMIB_IPSTATS;
```

该结构的各个字段的定义如下：

- dwForwarding：说明计算机上是启用，还是禁止转发 IPv4 包。

- dwDefaultTTL：为计算机所发出的数据报指定初始 TTL 的值。
- dwInReceives：说明已收到多少数据报。
- dwInHdrErrors：说明已收到多少报头有误的数据报。
- dwInAddrErrors：说明已收到多少地址有误的数据报。
- dwForwDatagrams：说明已转发多少数据报
- dwInUnknownProtos：说明已收到多少协议不明的数据报。
- dwInDiscards：说明已收到多少已丢弃的数据报。
- dwInDelivers：说明已收到多少已投递的数据报。
- dwOutRequests：说明 IPv4 请求传输多少数据报
- dwRoutingDiscards：说明已丢弃的外出数据报有多少。
- dwOutDiscards：说明丢弃的传输数据报有多少。
- dwOutNoRoutes：说明没有路由目标的数据报有多少。
- dwReasmTimeout：说明分段数据报完全到达的最长时间。
- dwReasmReqds：说明需要重组的数据报有多少。
- dwReasmOks：说明已成功重组的数据报有多少。
- dwFragFails：说明不能进行分段的数据报有多少。
- dwFragCreates：说明可被分段的数据报有多少。
- dwNumIf：说明计算机上可用的 IPv4 接口有多少。
- dwNumAddr：说明计算机上标识的 IPv4 地址有多少。
- dwNumRoutes：说明路由表中可用的路由有多少。

第 2 个统计函数：GetIcmpStatistics，用于获得 ICMP 统计情况。它的定义如下：

```
DWORD GetIcmpStatistics(
    PMIB_ICMP pStats
);
```

pStats 参数是一个指针，指向 MIB_ICMP 结构，这个结构接收计算机上当前的 ICMP 统计情况。MIB_ICMP 结构的定义如下：

```
typedef struct  _MIB_ICMP
{
    MIBICMPINFO stats;
}MIB_ICMP,*PMIB_ICMP;
```

一眼可见，这个 MIB_ICMP 结构中又包含一个 MIBICMPINFO 结构，后者的定义如下：

```
typedef struct _MIBICMPINFO
{
    MIBICMPSTATS icmpInStats;
    MIBICMPSTATS icmpOutStats;
```

```
}MIBICMPINFO;
```

MIBICMPINFO 结构通过 MIBICMPSTATS 结构接收传入或外出的 ICMP 信息。icmpInStats 参数接收传入数据，而 icmpOutStats 参数则接收外出数据。MIBICMPSTATS 结构的定义如下：

```
typedef struct _MIBICMPSTATS
{
    DWORD dwMsgs;
    DWORD dwErrors;
    DWORD dwDestUnreachs;
    DWORD dwTimeExcds;
    DWORD dwParmProbs;
    DWORD dwSrcQuenchs;
    DWORD dwRedirects;
    DWORD dwEchos;
    DWORD dwEchoReps;
    DWORD dwTimestamps;
    DWORD dwTimestampReps;
    DWORD dwAddrMasks;
    DWORD dwAddrMaskReps;
}MIBICMPSTATS;
```

它的各个字段的定义如下：

- dwMsgs：说明已收发多少消息。
- dwErrors：说明已收发多少错误。
- dwDestUnreachs：说明已收发多少“目标不可抵达”消息。
- dwTimeExcds：说明已收发多少 TTL 超出的消息。
- dwParmProbs：说明已收发多少表明数据报内有错误 IP 信息的消息。
- dwSrcQuenchs：说明已收发多少源结束消息。
- dwRedirects：说明已收发多少重定向消息。
- dwEchos：说明已收发多少 ICMP 回应请求。
- dwEchoReps：说明已收发多少 ICMP 回应答复。
- dwTimestamps：说明已收发多少时间戳请求。
- dwTimestampReps：说明已收发多少时间戳答复。
- dwAddrMasks：说明已收发多少地址掩码。
- dwAddrMaskReps：说明已收发多少地址掩码答复。

用于获得 TCP 统计情况的第 3 个统计函数是 GetTcpStatistcs，它的定义如下：

```
DWORD GetTcpStatistics(
    PMIB_TCPSTATS pStats
);
```

pStats 参数是一个指针，指向 MIB_TCPSTATS 结构，这个结构接收计算机当前的 IP 统计信息。MIB_TCPSTATS 结构的定义如下：

```
typedef struct _MIB_TCPSTATS
{
    DWORD dwRtoAlgorithm;
    DWORD dwRtoMin;
    DWORD dwRtoMax;
    DWORD dwMaxConn;
    DWORD dwActiveOpens;
    DWORD dwPassiveOpens;
    DWORD dwAttemptFails;
    DWORD dwEstabResets;
    DWORD dwCurrEstab;
    DWORD dwInSegs;
    DWORD dwOutSegs;
    DWORD dwRetransSegs;
    DWORD dwInErrs;
    DWORD dwOutRsts;
    DWORD dwNumConns;
}MIB_TCPSTATS,*PMIB_TCPSTATS;
```

它的各个字段的定义如下：

- dwRtoAlgorithm：说明即将采用哪种重传输算法。有效值包括：MIB_TCP_RTO_CONSTANT、MIB_TCP_RTO_RSRE、MIB_TCP_RTO_VANJ 以及针对其他类型的 MIB_TCP_RTO_OTHER。
- dwRtoMin：说明重传输超时的最小值，以毫秒计。
- dwRtoMax：说明重传输超时的最大值，以毫秒计。
- dwMaxConn：说明最多能接受多少连接。
- dwActiveOpens：说明计算机向服务器发起了多少次连接。
- dwPassiveOpens：说明计算机监听了多少次客户机发出的连接。
- dwAttemptFails：说明尝试连接失败的次数是多少。
- dwEstabResets：说明对已建立的连接实行了多少次重设。
- dwCurrEstab：说明目前已建立的连接有多少。
- dwInSegs：说明收到了多少分段数据报。
- dwOutSegs：说明传输了多少分段数据报(除已重新传输的分段外)。
- dwRetransSegs：说明重传输了多少分段数据报。
- dwInErrs：说明收到多少错误。
- dwOutRsts：说明重设标志后，又传输了多少分段数据报。
- dwNumConns：说明连接的总数是多少。

最后一个统计函数 GetUdpStatistics，用于获得计算机上的 UDP 统计情况。它的定义如下：

```
DWORD GetUdpStatistics(
    PMIB_UDPSTATS pStats
);
```

pStats 参数是一个指针，指向 MIB_UDPSTATS 结构，这个结构接收计算机当前的 IP 统计信息。MIB_UDPSTATS 结构的定义如下：

```
typedef struct _MIB_UDPSTATS
{
    DWORD dwInDatagrams;
    DWORD dwNoPorts;
    DWORD dwInErrors;
    DWORD dwOutDatagrams;
    DWORD dwNumAddrs;
}MIB_UDPSTATS,*PMIB_UDPSTATS;
```

它的各个字段定义如下：

- dwInDatagrams：说明已收到多少数据报。
- dwNoPorts：说明因为端口号有误而丢弃了多少数据报。
- dwInErrors：说明已收到多少错误数据报(除 dwNoPorts 中统计的数目之外)。
- dwOutDatagrams：说明已传输多少数据报。
- dwNumAddrs：说明监听者表中有多少 UDP 条目。

16.3 Route

利用 ROUTE.EXE 命令，我们可以打印和修改路由表。路由表用于决定连接请求或收发数据报在哪个 IPv4 接口上进行。“IP 助手库”(IP Helper Library)提供了几个函数，用于对路由表进行处理。这几个函数可用于 Windows 98，Windows Me 及 Windows NT 4.0(SP4 或以后的版本)。

首先，为大家讲讲 ROUTE.EXE 命令的作用。该命令最基本的用法便是打印路由表。一个路由的组成有这几个要素：一个目标地址、一个网络掩码、一个网关、一个本地 IP 接口和一个跃点。另外的用法便是增添和删除路由。若想增添路由，必须指定前面提到的全部参数。若想删除路由，只指定目标地址即可。在本小节，我们准备看看这几个 IP 助手函数是怎样打印路由表的。随后，再为大家讲讲怎样增添或删除路由。

16.3.1 获得路由表

ROUTE.EXE 执行的最常见操作便是打印路由表。这是通过 GetIpForwardTable 函数来完成的。该

函数的定义如下：

```
DWORD GetIpForwardTable (
    PMIB_IPFORWARDTABLE pIpForwardTable,
    PULONG pdwSize,
    BOOL bOrder
);
```

第 1 个参数 pIpForwardTable 中包含了函数返回的路由表信息。在调用这个函数时，该参数应该引用一个恰当的缓冲区。如果 pIpForwardTable 参数被设为 NULL(或者缓冲区长度不够)，pdwSize 参数便返回成功调用该函数所需的缓冲区的长度。最后一个参数 bOrder，表明是否对返回结果进行分类。默认的分类顺序是：

- 目标地址
- 生成路由的协议
- 多路径路由策略
- 下一跳的地址。

路由信息包含在 MIB_IPFORWARDROW 结构中，这个结构的定义如下：

```
typedef struct _MIB_IPFORWARDTABLE
{
    DWORD dwNumEntries;
    MIB_IPFORWARDROW table[ANY_SIZE];
}MIB_IPFORWARDTABLE,*PMIB_IPFORWARDTABLE;
```

这个结构内还有一个 MIB_IPFORWARDROW 结构数组。dwNumEntries 字段表明这个数组中有几个结构。MIB_IPFORWARDROW 结构的定义如下：

```
typedef struct _MIB_IPFORWARDROW
{
DWORD dwForwardDest;
DWORD dwForwardMask;
DWORD dwForwardPolicy;
DWORD dwForwardNextHop;
DWORD dwForwardIfIndex;
DWORD dwForwardType;
DWORD dwForwardProto;
DWORD dwForwardAge;
DWORD dwForwardNextHopAS;
DWORD dwForwardMetric1;
DWORD dwForwardMetric2;
DWORD dwForwardMetric3;
DWORD dwForwardMetric4;
DWORD dwForwardMetric5;
```

```
}MIB_IPFORWARDROW,*PMIB_IPFORWARDROW;
```

它的各个字段的定义如下：

- dwForwardDest：是目标主机的 IPv4 地址。
- dwForwardMask：是目标主机的子网掩码。
- dwForwardPolicy：指定影响多路径路由选择的一系列条件。这些条件通常采用 IP TOS 的形式。关于 TOS 的详情，可参考第 7 章和 IP_TOS 选项。关于多路径路由的详情，参考 RFC 1354。
- dwForwardNextHop：是路由表中下一跳的 IPv4 地址。
- dwForwardIfIndex：表明针对该路由的接口的索引。
- dwForwardType：表示 RFC1354 中定义的路由类型。表 16.10 列出了该字段可能的值及其含义。
- dwForwardProto：是生成这个路由的协议。表 16.11 列出了该字段可能的值。IPX 协议值定义在 ROUTPROT.H 中，而 IP 条目则包含在 IPRTRMIB.H 中。
- dwForwardAge：表明路由持续时间，以毫秒计。
- dwForwardNextHopAS：是下一跳的自治系统编号。
- dwForwardMetric1：是路由协议专用的跃点值。详情参见 RFC1354。这个字段中包含路由跃点值，这个值是在执行 ROUTE.EXE 打印命令时经常看到的。若不使用这个条目，对这个字段以及下面 4 个字段而言，其值均为 MIB_IPROUTE_METRIC_UNUSED(-1)。
- dwForwardMetric2：是路由协议专用的跃点值。详情参见 RFC1354。
- dwForwardMetric3：是路由协议专用的跃点值。详情参见 RFC1354。
- dwForwardMetric4：是路由协议专用的跃点值。详情参见 RFC1354。
- dwForwardMetric5：是路由协议专用的跃点值。详情参见 RFC1354。

表 16.10 路由表条目中可能的路由类型

转发类型	说 明
MIB_IPROUTE_TYPE_INDIRECT	下一跳不是最终目的地(远程路由)
MIB_IPROUTE_TYPE_DIRECT	下一跳是最终目的地(本地路由)
MIB_IPROUTE_TYPE_INVALID	路由无效
MIB_IPROUTE_TYPE_OTHER	其他路由

表 16.11 路由协议标识符

协议标识符	说 明
MIB_IPPROTO_OTHER 未	未列出的协议

续表

协议标识符	说　明
MIB_IPPROTO_LOCAL	堆栈生成的路由
MIB_IPPROTO_NETMGMT	ROUTE.EXE 程序或 SNMP 添加的路由
MIB_IPPROTO_ICMP	ICMP 重定向的路由
MIB_IPPROTO_EGP	外部网关协议
MIB_IPPROTO_GGP	网关网关协议
MIB_IPPROTO_HELLO 未	HELLO 路由协议
MIB_IPPROTO_RIP 未未	路由信息协议
MIB_IPPROTO_IS_IS	IP 中间系统到中间系统协议
MIB_IPPROTO_ES_IS	IP 终端系统到中间系统协议
MIB_IPPROTO_CISCO	IP Cisco 协议
MIB_IPPROTO_BBN	BBN 协议
MIB_IPPROTO_OSPF	先打开最短路径(OSPF)路由协议
MIB_IPPROTO_BGP	边界网关协议
MIB_IPPROTO_NT_AUTOSTATIC 未	最初由一个路由协议添加的、非静态路由
MIB_IPPROTO_NT_STATIC	路由用户接口或 Routemon.exe 程序添加的路由
MIB_IPPROTO_STATIC_NON_DOD	等同于 PROTO_IP_NT_STATIC，但这些路由不会引发 DOD(Dial on Demand，按需拨号)
IPX_PROTOCOL_RIP	IPX 的路由信息协议
IPX_PROTOCOL_SAP	服务声明协议
IPX_PROTOCOL_NLSP	NetWare 链接服务协议

16.3.2　增加路由

路由命令的另一个作用是添加路由。记住，要添加一个路由，必须指定其目标 IPv4、网络掩码、网关、本地 IPv4 接口以及跃点。添加路由时，应该保证所给的本地 IPv4 接口是有效的。除此以外，因为路由将添加到这个接口上，所以还需要根据本地 IPv4 接口的内部索引值引用这个接口。调用

GetIpAddrTable 函数，便可获得这一信息。该函数的定义如下：

```
DWORD GetIpAddrTable (
    PMIB_IPADDRTABLE pIpAddrTable,
    PULONG pdwSize,
    BOOL bOrder
);
```

第 1 个参数 pIpAddrTable，是将返回结构 MIB_IPADDRTABLE 的缓冲区的正确长度。pdwSize 参数是被当作第 1 个参数传递的缓冲区的长度。最后一个参数 bOrder，说明是否通过升序 IP 地址，来返回本地 IPv4 接口。要找到正确的缓冲区长度，可指定 pIpAddrTable 参数为 NULL。函数返回之后，pdwSize 便指明正确的缓冲区长度。MIB_IPADDRTABLE 结构的定义如下：

```
typedef struct _MIB_IPADDRTABLE
{
    DWORD dwNumEntries
    MIB_IPADDRROW table[ANY_SIZE];
}MIB_IPADDRTABLE,*PMIB_IPADDRTABLE;
```

这个结构内还有一个 MIB_IPADDROW 结构。dwNumEntries 字段表明 table 字段数组中有多少个 MIB_IPADDROW 条目。MIB_IPADDRROW 结构的定义如下：

```
typedef struct _MIB_IPADDRROW
{
    DWORD dwAddr;
    DWORD dwIndex;
    DWORD dwMask;
    DWORD dwBCastAddr;
    DWORD dwReasmSize;
    unsigned short unused1;
    unsigned short unused2;
}MIB_IPADDRROW,*PMIB_IPADDRROW;
```

它的各个字段的定义如下：

- dwAddr：是给定接口的 IPv4 地址。
- dwIndex：与 IPv4 地址关联在一起的接口索引。
- dwMask：是 IPv4 地址的子网掩码。
- dwBCastAddr：是广播地址。
- dwReasmSize：是已收到的数据报重装后的最大长度。
- unused1andunused2：目前尚未使用。

利用 GetIpAddrTable 这个函数，便可验证针对指定路由的本地 IPv4 地址是否有效。在路由表中添加路由的函数是 SetIpForwardEntry，它的定义如下：

```
DWORD SetIpForwardEntry (
    PMIB_IPFORWARDROW pRoute
);
```

该函数中，惟一的参数 pRoute，是一个指向 MIB_IPFORWARDROW 结构的指针。这个结构定义了建立一个新路由所需要的元素。我们曾对这个结构及其成员字段进行了讨论。要添加一个新路由，必须指定下面几个字段的值：dwForwardIfIndex、dwForwardDest、dwForwardMask、dwForwardNextHop 以及 dwForwardPolicy。

16.3.3　删除路由

路由程序的最后一个动作是删除路由，这个命令最简单。调用路由命令删除路由时，必须指定准备删除的目标地址。然后，才能搜索与目标地址对应的 MIB_IPFORWARDROW 结构，这个结构是 GetIpForwardTable 返回的。接下来，再将 MIB_IPFORWARDROW 结构传递给 DeleteForwardEntry 函数，便可删除指定的路由条目了。DeleteForwardEntry 函数的定义如下：

```
DWORD DeleteIpForwardEntry (
    PMIB_IPFORWARDROW pRoute
);
```

删除路由的另一种可选办法是自己动手指定 pRoute 字段。删除路由所需的字段有：dwForwardIfIndex、dwForwardDest、dwForwardMask、dwForwardNextHop 以及 dwForwardPolicy。

16.4　ARP

ARP.EXE 程序用于查看和处理 ARP 缓存。通过 IP 助手函数模拟 ARP.EXE 程序的 Platform SDK 示例名为 Iprap.exe。ARP(也许大家还记得，它代表名称解析协议)负责把一个 IPv4 地址解析成一个物理性的 MAC 地址。为不致影响性能，计算机将这一信息缓存下来，并可能通过 ARP.EXE 程序对该信息进行访问。利用这个程序，选择-a 选项，便显示 ARP 表；选择-d 选项，便删除一个条目；选择-s 选项，则添加一个条目。下一小节中，我们将谈谈如何打印 ARP 缓存，在 ARP 表中添加条目以及删除 ARP 条目。

本小节中讨论的 IP 助手函数适用于 Windows 98、Windows Me 以及 Windows NT4.0(SP4 以及稍后的版本)。

用于获得 ARP 表的这个 IP 助手函数最简单。它便是 GetIpNetTable，其定义如下：

```
DWORD GetIpNetTable (
    PMIB_IPNETTABLE pIpNetTable,
    PULONG pdwSize,
    BOOL bOrder
```

```
);
```

第 1 个参数 pIpNetTable，是一个指向 MIB_IPNETTABLE 结构的指针，这个结构返回的是 ARP 信息。在调用这个函数时，必须提供一个足够大的缓冲区长度。和其他 IP 助手函数一样，如该参数是 NULL，就会返回 pdwSize 参数所需的正确的缓冲区长度，和 ERROR_INSUFFICIENT_BUFFER 错误。反之，pdwSize 则表示被当作 pIpNetTable 传递的缓冲区的长度。最后一个参数 bOrder，表明是否按升序对返回的 IP 条目进行分类。

MIB_IPNETTABLE 结构内还有一个 MIB_IPNETROW 结构数组，后者的定义如下：

```
typedef struct _MIB_IPNETTABLE
{
    DWORD dwNumEntries;
    MIB_IPNETROW table[ANY_SIZE];
}MIB_IPNETTABLE,*PMIB_IPNETTABLE;
```

dwNumEntries 字段表明 table 字段中有多少数组条目。MIB_IPNETROW 结构中包含真正的 ARP 条目信息。这个结构的定义为：

```
typedef struct _MIB_IPNETROW {
    DWORD dwIndex;
    DWORD dwPhysAddrLen;
    BYTE bPhysAddr[MAXLEN_PHYSADDR];
    DWORD dwAddr;
    DWORD dwType;
}MIB_IPNETROW,*PMIB_IPNETROW;
```

该结构的各个字段的定义如下：

- dwIndex：指定适配器的索引
- dwPhysAddrLen：表明 bPhysAddrs 字段内包含的物理接口的长度，按字节数计
- bPhysAddr：是一个字节数组，其中包含适配器的物理地址
- dwAddr：指定适配器的 IP 地址
- dwType：表明 ARP 条目的类型。表 16.12 展示了这个字段可能的值

表 16.12 ARP 条目类型可能的值

ARP 类型	含义
MIB_IPNET_TYPE_STATIC	静态条目
MIB_IPNET_TYPE_DYNAMIC	动态条目
MIB_IPNET_TYPE_INVALID	无效条目
MIB_IPNET_TYPE_OTHER	其他条目

16.4.1　添加 ARP 条目

ARP 另一个作用是在 ARP 缓存中添加条目，这个操作比较简单。用于添加 ARP 条目的 IP 助手函数是 SetIpNetEntry，它的定义如下：

```
DWORD SetIpNetEntry (
    PMIB_IPNETROW pArpEntry
);
```

惟一的参数是 MIB_IPNETWORK 结构，我们已在前一小节中提过这个结构。要添加一个 ARP 条目，只须用新的 ARP 信息来填充这个结构。首先，把 dwIndex 参数设为一个本地 IPv4 地址的索引，这个索引表明添加的 ARP 条目将用于哪个网络。记住，如果给出的是 IP 地址，就可利用 GetIpAddrTable 函数把它映射到索引中。下一个字段 dwPhysAddrLen，一般设为 6(多数物理地址，例如 ETHEENET MAC 地址，长度都是 6 个字节)。bPhysAddr 字节数组必须设为物理地址。多数 MAC 地址都用 12 个字符来表示，例如 00-A0-C9-A7-86-E8。这些字符需要被硬编码到 bPhysAddr 字段的恰当的字节数组位置内。例如，MAC 地址示例就会硬编码到下面的字节中：

```
00000000 10100000 11001001 10100111 10000110 11101000
```

编码方法和编码 IPX 及 ATM 地址所采用的方法一样(详情参见第 4 章)。dwAddr 字段必须设为从属于远程主机及指定 MAC 地址的 IP 地址。最后一个字段 dwType，设为表 16.12 中所列的 ARP 条目类型之一。MIB_IPNETWORK 结构填充好之后，就调用 SetIpNetEntry 函数，在缓存中添加 ARP 条目。如果成功，就会返回 NO_ERROR。

16.4.2　删除 ARP 条目

删除 ARP 条目类似于添加条目，只不过删除时需要的信息有所不同，这里需要的是接口索引 dwIndex，和准备删除的 ARP 条目的 IPv4 地址，即 dwADDR。用于删除 ARP 条目的函数是 DeleteIpNetEntry，它的定义如下：

```
DWORD DeleteIpNetEntry (
    PMIB_IPNETROW pArpEntry
);
```

其惟一的参数是一个 MIB_IPNETROW 结构，删除 ARP 条目时所需的惟一的信息是本地 IPv4 索引和准备删除的 IPv4 地址。记住，本地 IPv4 接口的索引编号可通过 GetIpAddrTable 函数获得。如果成功，便返回 NO_ERROR。

16.4.3　发送 ARP 请求

发送一个用目标物理地址填充 ARP 缓存的 ARP 请求有时很有用。例如，发送一个比链接 MTU

大的 UDP 数据报到一个不在 ARP 缓存内的目标，总是会失败；在 IPv4 的失败时不会有什么表示，在 IPv6 上的失败则会指明一个错误。SendArp 函数会试图把给定 IPv4 地址解析为它的 MAC 地址。没有和 SendArp 函数对应的 IPv6 函数，但 IPv6 至少会指明一个错误已经发生，从而应用程序可以在物理地址解析出来后，重传数据包。SendArp 函数的定义为：

```
DWORD SendArp(
    IPAddr DestIP,
    IPAddr SrcIP,
    PULONG pMacAddr,
    PULONG PhyAddrLen
);
```

DestIP 参数表示 ARP 试图解析的 IPv4 目标地址。SrcIP 参数是用来发送 ARP 请求的可选 IPv4 接口，如果值为 0，则由路由表来决定使用哪个接口。pMacAddr 参数是接收目标物理地址的数据缓冲。最后，PHyAddrLen 参数指明在缓冲区中返回的物理地址长度。

16.5 小　　结

本章以几个知名的系统实用程序为代表，介绍了 IP 助手 API。通过本章的学习，您会了解到如何进行编程，从 IPv4 物理堆栈获取有用的信息。除了新的 IP 助手 API 即 GetAdaptersAddresses 包含 IPv6 地址信息外，IP 助手 API 目前只列举了 IPv4 信息。

[General Information]
书名=Windows网络编程　第2版(原文版)
作者=
页数=458
SS号=0
出版日期=

封面
书名
版权
前言
目录
正文