第一部分　传统网络API

本书第一部分讲述的是传统的网络接口 NetBIOS、重定向器以及通过重定向器进行的各类网络通信。尽管本书大部分内容均围绕 Winsock编程这一主题展开，但是， API比起Winsock来，仍然具有某些独到之处。其中，第 1章探讨的是NetBIOS接口，它和Winsock类似，也是一种与协议无关的网络 API。NetBIOS提供了异步调用，同时兼容于较老的操作系统，如 OS/2和DOS等等。第2章讨论了重定向器的问题，它是接下去的两个新主题 —— 邮槽（第3章）和命名管道（第4章）的基础。重定向器提供了与传输无关的文件输入／输出方式。邮槽是一种简单的接口，可在 Windows机器之间实现广播和单向数据通信。最后，命名管道可建立一种双向信道，这种信道提供了对 Windows安全通信的支持。

第1章　NetBIOS

“ 网络基本输入／输出系统 ”（Network Basic Input/Output System, NetBIOS）是一种标准的应用程序编程接口（ API），1983年由Sytek公司专为IBM开发成功。NetBIOS为网络通信定义了一种编程接口，但却没有详细定义物理性的“ 帧 ”如何在网上传输。 1985年，IBM创制了NetBIOS扩展用户接口（NetBIOS Extended User Interface, NetBEUI），它同NetBIOS接口集成在一起，终于构成了一套完整的协议。由于 NetBIOS接口变得愈来愈流行，所以各大厂商也开始在其他如TCP/IP和IPX/SPX的协议上实施NetBIOS编程接口。到目前为止，全球已有许多平台和应用程序需要依赖于 NetBIOS，其中包括Windows NT、Windows 2000、Windows 95和Windows 98的许多组件。

注意　Windows CE并不支持NetBIOS API，只是用TCP/IP作为其传送协议，并同时支持NetBIOS的名字与名字解析。

Win32 NetBIOS接口向后兼容于早期的应用程序。本章要讨论的是 NetBIOS编程基础。首先向大家介绍的是NetBIOS的一些基本知识，从NetBIOS的名字及LANA编号开始，接着，我们围绕NetBIOS提供的基本服务展开讨论，比如面向会话和“ 无连接 ”通信等等。在每一节，都展示了一个简单的客户机和服务器示例。在本章最后，我们陈列了程序员需留意的一系列陷阱以及易犯的错误。在本书的附录 A中，大家可找到一份命令索引，其中对每个 NetBIOS命令都进行了总结，包括必要的参数，以及对其行为的简单说明。

OSI 网络模型

“ 开放系统互连 ”（OSI）模型从一个很高的层次对网络系统进行了描述。 OSI模型总共包含了七层。从最顶部的“ 应用层 ”开始，一直到最底部的“ 物理层 ”，这七个层完整阐述了最基本的网络概念。图 1-1展示的正是OSI模型的样子。

层	描 述
应用层	为用户提供相应的界面，以便使用提供的连网功能
表示层	完成数据的格式化
会话层	控制两个主机间的通信链路（开放、操作和关闭）
传输层	提供数据传输服务（可靠或不可靠）
网络层	在两个主机之间提供一套定址/寻址机制，同时负责数据包的路由选择
数据链路层	控制两个主机间的物理通信链路：同时还要负责对数据进行整形，以便在物理媒体上传输
物理层	物理媒体负责以一系列电子信号的形式，传出数据

图1-1 OSI网络模型

对应OSI模型，NetBIOS主要在会话和传输层发挥作用。

1.1 Microsoft NetBIOS

如前所述，NetBIOS API实施方案适用于为数众多的网络协议，使得编程接口“与协议无关”。换言之，假如根据NetBIOS规范设计了一个应用程序，它就能在TCP/IP、NetBIOS甚至IPX/SPX上运行。这是一项非常有用的特性，因为对一个设计得当的NetBIOS应用程序来说，它几乎能在任何机器上运行，无论机器连接的物理网络是什么。然而，我们也必须留意几个方面的问题。要想使两个NetBIOS应用（程序）通过网络进行正常通信，那么对它们各自运行的机器来说，至少必须安装一种两者通用的协议。举个例子来说，假定小张的机器只安装了TCP/IP，而小马的机器只安装了NetBEUI，那么对小张机器上的NetBIOS应用来说，便无法同小马机器上的应用进行通信。

除此以外，只有部分协议实施了NetBIOS接口。Microsoft TCP/IP和NetBEUI在默认情况下已提供了一个NetBIOS接口；然而，IPX/SPX却并非如此。为此，微软专门提供了一个IPX/SPX版本，在其中实现了该接口。在设计网络时，这个问题必须注意。安装协议时，具有NetBIOS能力的IPX/SPX协议通常会自动提醒你注意这方面的问题。例如，Windows 2000提供的协议本身就叫作“NWLink IPX/SPX/NetBIOS兼容传送协议”。而在Windows 95和Windows 98中，请留意IPX/SPX协议属性对话框，其中有一个特殊的复选框，名为“希望在IPX/SPX上启用NetBIOS”。

另外要注意的一个重要问题是NetBEUI并非是一种“可路由”协议。假定在客户机和服务器之间存在一个路由器，那么这种协议在两部机器上的应用便无法沟通。收到数据包后，路由器便会将其“无情地”地抛弃。TCP/IP和IPX/SPX则不同，它们均属“可路由”协议，不会出现这方面的问题。要注意的是，假如你需要在很大程度上依靠NetBIOS，那么在配置网络时，至少应安装一种可路由的传送协议。要想深入了解各种协议的特征以及相应的注意事项，请参阅第6章。

1.1.1 LANA编号

从编程角度思考，大家或许会觉得奇怪，传送协议与NetBIOS如何对应起来呢？答案便在于LAN适配器（LAN adapter, LANA）编号，它是我们理解NetBIOS的关键。在最初的NetBIOS实施方案中，每张物理网卡都会分配到一个独一无二的值：即LANA编号。但到Win32下，这种做法便显得有些问题。因为对一个工作站来说，它完全可能同时安装了多种网络协议，也可能安装了多张网卡。

每个LANA编号对应于网卡及传输协议的唯一组合。例如，假定某工作站安装了两张网卡，以及两种具有NetBIOS能力的传输协议（如TCP/IP和NetBEUI），那么总共就有四个LANA编号。下面是一种对应关系的例子：

0. TCP/IP —— 网卡1
1. NetBEUI —— 网卡1
2. TCP/IP —— 网卡2
3. NetBEUI —— 网卡2

通常，LANA编号的范围在0到9之间，除LANA 0之外，操作系统并不按某种固定的顺序来分配这些编号。那么，LANA 0有什么特殊含义呢？LANA 0代表的是“默认”LANA！NetBIOS问世早期，许多应用都采用硬编码的形式，只依赖LANA 0进行工作。在那时，大多数操作系统也只支持一个LANA编号。考虑到向后兼容的目的，我们可将LANA 0人工分配给一种特定的协议。

在Windows 95和Windows 98中，通过选择控制面板中的“网络”图标，可访问一种网络协议的“属性”对话框。在“网络”对话框中选择“配置”选项卡，再从网络组件列表中选择一种网络协议，按下“属性”按钮即可。对具有NetBIOS能力的每一种协议来说，其属性对话框的“高级”选项卡都有一个“设成默认的通信协议”复选框。若选中这个复选框，会重新安排协议的绑定，使默认协议能够分配到LANA 0。注意在任何时候，只能有一种协议才能选中这个复选框。由于Windows 95和Windows 98具有所谓的“即插即用”功能，所以我们没有其他办法可对协议的编号顺序进行更改。

Windows NT 4则允许用户在设置NetBIOS时拥有更大的灵活性。在“网络”对话框的“服务”选项卡中，可从“网络服务”列表框内选择NetBIOS接口，然后点按“属性”按钮。随后便会出现“NetBIOS配置”对话框，在这里可针对每一对网卡／传输协议的组合，分配各自的LANA编号。在这个对话框中，每张网卡都以其驱动程序的名字加以标识；但协议名称却显得有些暧昧。在图1-2中，我们展示了NetBIOS配置对话框的样子。单击其中的“Edit”（编辑）按钮，便可为每种协议单独分配LANA编号。Windows 2000也允许我们单独分配LANA编号。在控制面板中，双击“网络和拨号连接”图标。随后，从“高级”菜单中选择“高级设置”，然后在高级设置对话框中选择“LANA编号”选项卡。

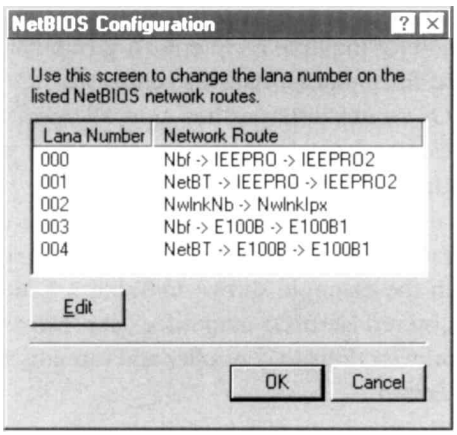

图1-2 NetBIOS配置对话框。这是一部多宿主机器，安装了两张网卡和三种传输协议：TCP/IP（NetBT）、NetBEUI（Nbf）以及IPX/SPX（NwlnkNb）

要想设计出一个“健壮”的NetBIOS应用，必然需要让自己的代码能对任意LANA编号上的连接进行控制。例如，假定小马编写了一个NetBIOS服务器应用，对LANA 2上的客户机进行监听。在小马的机器（即服务器）上，LANA 2正好对应于TCP/IP。后来，小张需要编写一个客户端应用，同小马的服务器通信，所以他决定让自己的程序通过工作站的LANA 2建立连接。然而，小张工作站上的LANA 2对应于NetBEUI。这样一来，两个应用相互间均无法通信——尽管两者都安装了TCP/IP和NetBEUI。为纠正协议的这种差异，小马的服务器应用程序必须对小马工作站上每个可能的LANA编号上的客户机连接进行“监听”。类似地，小张的客户机应用程序需要针对本机每个可能的LANA编号，尝试在其上面的连接。只有这样，小马和小张才能保证自己的应用尽最大可能成功通信。当然，尽管我们需要在代码中对任何LANA编号上的连接进行控制，但并不表示能够百分之百地成功。假如两台机器根本就没有安装一种共通的协议，那么无论如何都是不能成功的！

1.1.2 NetBIOS名字

现在，我们知道了LANA编号是什么，接着再来讨论NetBIOS名字（名称）的问题。对一个进程（或“应用”、“应用程序”）来说，它会注册自己希望与其通信的每个LANA编号。一个NetBIOS名字长度为16个字符，其中第16个字符是为特殊用途保留的。在名字表内添加一个名字时，应将名字缓冲区初始化成空白。在Win32环境中，针对每个可用的LANA编号，每个进程都会为其维持一张NetBIOS名字表。若为LANA 0增添一个名字，意味着你的应用程序只能在LANA 0上同客户机建立连接。对每个LANA来说，能够添加的名字的最大数量是254，编号从1到254（0和255由系统保留）。然而，每种操作系统都设置了一个低于254的最大默认值。重设每个LANA编号时，我们可对此默认值进行修改。

另外，NetBIOS名字共有两种类型：唯一名字和组名。“唯一名字”意味着它是独一无二的：网络上不能再有其他任何进程来注册这个名字。如果一台机器已注册了某名字，那么在你注册该名字时，便会收到一条“重复名字”出错提示。大家或许已经知道，微软网络中的机器名采用的便是NetBIOS名字。机器启动时，会将自己的名字注册到本地的“Windows互联网命名服务器”（WINS）。如果事前已有另一台机器注册了同样的名字，WINS服务器便会报错。WINS服务器维护着已注册的所有NetBIOS名字的一个列表。除此以外，随名字一道，还可保存协议特有的一些信息。比如在TCP/IP网络中，WINS同时维护着NetBIOS名字以及注册那个名字的IP地址（亦即相应的机器）。假如配置网络时未为其分配一个WINS服务器，那么如何检查名字是否重复呢？这时便要采用在整个网络内“发广播”的形式。当一名发送者向全网络发出一条特殊的广播消息时，如果没有其他机器回应这条消息，便允许发送者使用该名字。

而在另一方面，“组名”的作用是将数据同时发给多个接收者；或者相反，接收发给多个接收者的数据。组名并非一定要“独一无二”，它主要用于多播（多点发送）数据通信。

在NetBIOS名字中，第16个字符用于区分不同的微软网络服务。各种网络服务和组名需要用一个WINS服务器完成注册。要么由配置了WINS功能的计算机进行名字的直接注册，要么由那些尚未配置WINS功能的计算机，通过在本地子网内进行广播注册。Nbtstat命令是一个非常有用的工具，可用它获取与本地（或远程）计算机上注册的NetBIOS名字有关的信息。在表1-1展示的例子中，Nbtstat-n命令可针对用户“Davemac”，生成这个已注册的NetBIOS名字的列表。Davemac登录进入的那部计算机已被配置成一个主域控制器，而且运行的是

Windows NT Server操作系统，且已安装了Internet信息服务器（IIS）。

表1-1 NetBIOS名字表

名 字	第16个字节	名字类型	服 务
DAVEMAC1	<00>	唯一	工作站服务名
DAVEMAC1	<20>	唯一	服务器服务名
DAVEMACD	<00>	成组	域名
DAVEMACD	<1C>	成组	域控制器名
DAVEMACD	<1B>	唯一	主控浏览器名
DAVEMAC1	<03>	唯一	发信者名
Inet~Services	<1C>	成组	Internet信息服务器组名
IS~DAVEMAC1	<00>	唯一	Internet信息服务器唯一名
DAVEMAC1+++++++	<BF>	唯一	网络监视器名字

只有在安装了TCP/IP协议的前提下，才会安装Nbtstat命令。该工具亦可用来查询远程机器的名字表，方法是在远程机器的名字后面，接上一个 -a参数；或在远程机器的IP地址后，接上一个-A参数。

在表1-2中，我们总结了各种不同的Microsoft网络服务为唯一NetBIOS计算机名追加的默认第16个字节值。

表1-2 唯一名字标识符

第16个字节	含 义
<00>	工作站服务名。通常，它对应于NetBIOS计算机名
<03>	收发消息时采用的信使服务名。WINS服务器会将这个名字注册成WINS客户机上的信使服务，并通常追加到计算机名后面，以及当前登录到计算机的用户名的后面
<1B>	域主控浏览器名。这个名字用于标识主域控制器，并指出用什么客户机和其他浏览器同域主控浏览器取得联系
<06>	远程访问服务（RAS）服务器服务
<1F>	网络动态数据交换（NetDDE）服务
<20>	用于为文件共享提供“共享点”的服务器服务名
<21>	RAS客户机
<BE>	网络监视器代理
<BF>	网络监视器工具

表1-3则列出了在常用的一系列NetBIOS组名后，追加的默认第16个字节字符。

如此多的标识符很易使人产生混淆，很难真正记住。所以，请考虑把它作为一个“速查表”或“索引”使用。大家或许不应在自己的 NetBIOS名字中使用它们。为防止偶然同你的NetBIOS名字发生冲突，最好避免使用唯一名字标识符。对于组名，恐怕更要引起高度注意——假如你的名字同一个已有的组名相同，那么不会产生任何错误提示。若发生这种情况，结果就是会收到原本发给其他人的数据。

表1-3 组名标识符

第16个字节	含 义
<1C>	一个域组名，在这个组内包含了已注册域名的一系列计算机的特定地址。由域控制器来注册这个名字。WINS将它当作一个域组看待：组内每个成员必须单独更新自己的名字。域组最多只能包容 25个名字。若复制的一个静态 1C名字同另一个WINS服务器上的某个动态1C名字发生冲突，便会增加成员的一个“联合”，同时将记录标定为“静态”。假如记录是静态的，组内成员便不必定时刷新自己的IP地址

(续)

第16个字节	含 义
<1D>	指定一个主控浏览器的名字，客户机通过它访问主控浏览器。在一个子网上，只能有一个主控浏览器。WINS服务器会对域名注册作出“正”（肯定）响应，但却不会将域名保存在自己的数据库中。假如一台计算机向WINS服务器送出一个域名查询，则WINS服务器会返回一个“负”（否定）响应。若送出域名查询的那台计算机已被配置成h节点或m节点，便会随之广播那个查询，以解析出正确的名字。客户机解析名字的方法是由节点的类型决定的。如客户机配置成b节点解析，便会送出广播包，以便广告并解析出NetBIOS名字。p节点解析采用与WINS服务器的点到点通信方式。而m节点属于b及p节点的一种混合形式：首先使用的是b节点；如有必要，再接着使用p节点。最后一种解析方式是h节点，亦称“混合模式”。它无论如何都会先尝试使用p节点注册和解析，然后只有在解析失败的前提下，才会换用b节点。Windows操作系统默认为h节点
<1E>	一个普通组名。浏览器可向这个名字发送广播数据，并通过对它的监听来挑选一个主控浏览器。这些广播面向的是本地子网，绝对不应通过路由器传输
<20>	一个Internet组名。这种类型的名字由WINS服务器进行注册，以便为了管理方面的目的来标定特定的计算机组。例如，“printersg”可以是一个注册的组名，用于标定由打印服务器构成的一个管理性组
MSBROWSE	不再是单独一个追加的第16位字符，“_MSBROWSE_”需要追加到一个域名后面，并在本地子网上进行广播，向其他主控浏览器通告这个新增的域

1.1.3 NetBIOS特性

NetBIOS同时提供了“面向连接”服务以及“无连接”服务。面向连接的服务，是指它允许两个客户机相互间建立一个会话，或者说建立一个“虚拟回路”。这种“会话”实际是一种双向的通信数据流，通信的每一方都可向另一方发送消息。面向连接的服务可担保在两个端点之间，任何数据都能准确无误地传送。在这种服务中，服务器通常将自己注册到一个已知的名字下。客户机会搜寻这个名字，以便建立与服务器的通信。就拿NetBIOS的情况来说，服务器进程会针对想通过它建立通信的每一个LANA编号，将自己的名字加入与其对应的名字表。而对位于其他机器上的客户来说，就可将一个服务名解析成机器名，然后要求同服务器进程建立连接。大家可以看到，为建立这种虚拟回路，必须采取一些适当的步骤。而且在初次建立连接的时候，还会牵涉到一些额外的开销。“面向连接”或“面向会话”的通信可保证通信具有极高的可靠性，而且数据包的收发顺序亦能确保正确无误。然而，它仍然是一种“以消息为基础”的服务。也就是说，假如已连接好的某个客户机执行一个“读”命令，那么服务器在流中仍然只会返回一个数据包——尽管客户机此时提供了一个足够大的缓冲区，可同时容下几个包！

“无连接”或数据报服务中，服务器并不将自己注册到一个特定的名下，而只是由客户机收集数据，然后将其送入网络，事前不必先建好任何连接（即无连接）。对于数据的目的地址，客户机会将其定义成服务器相应进程对应的NetBIOS名字。这种类型的服务不提供任何保障，但同面向连接的服务相比，却可有更好的性能，如在使用数据报服务（无连接服务）时，省下了建立连接所需的开销。例如，客户机可能向服务器兴冲冲地一下子发出数千字节的数据，但那台服务器早在一两天前便已当机了。除非依赖自服务器传来的响应，否则客户机永远都收不到任何错误提示（在这种情况下，假如在一个特定的时间段内，没有收到任何响应，便

可认为服务器出了故障)。数据报服务既不能保证数据传输的可靠性，也不能保证数据包的传送顺序正确无误。

1.2 NetBIOS编程基础

现在，我们已理解了NetBIOS的一些基本概念，接下来要讨论的是NetBIOS API的设置，这其实非常简单，因为只有一个函数：

```
UCHAR Netbios(PNCB pNCB);
```

用于NetBIOS的所有函数声明、常数等等均是在头文件 Nb30.h内定义的。若想连接NetBIOS应用，唯一需要的库是Netapi32.lib。该函数最重要的特征便是pNCB这个参数，它对应于指向某个网络控制块（ NCB）的一个指针。在那个 NCB结构中，包含了为执行一个NetBIOS命令，相应的Netbios函数需要用到的全部信息。该结构的定义如下：

```
typedef struct _NCB
{
    UCHAR     ncb_command;
    UCHAR     ncb_retcode;
    UCHAR     ncb_lsn;
    UCHAR     ncb_num;
    PUCHAR    ncb_buffer;
    WORD      ncb_length;
    UCHAR     ncb_callname[NCBNAMSZ];
    UCHAR     ncb_name[NCBNAMSZ];
    UCHAR     ncb_rto;
    UCHAR     ncb_sto;
    void      (*ncb_post) (struct _NCB *);
    UCHAR     ncb_lana_num;
    UCHAR     ncb_cmd_cplt;
    UCHAR     ncb_reserve[10];
    HANDLE    ncb_event;
} * PNCB, NCB;
```

注意，并不是在对NetBIOS的每次调用中都需要用到该结构内的全部成员；有些数据字段对应的是输出参数（换言之，自Netbios调用返回之后才能设置)。在此提醒大家重要的一点：进行任何Netbios调用之前，不要一开始就填写结构内的各个成员，而应先将这个NCB结构清零！请看看表1-4的总结，其中解释了每个字段的用法。此外，本书附录 A的命令索引对每个NetBIOS命令都进行了详尽总结，并解释了它需要用到NCB结构中的哪些字段，以及哪些字段可选。

表1-4 NCB结构成员

字 段	定 义
ncb_command	指定要执行的NetBIOS命令。许多命令都可同步或异步与ASYNCH(0X80)标志以及命令进行按位OR（或）运算
ncb_retcodef	指定操作的返回代码。在一个异步操作进行期间，函数会将该值设为NRC_PENDING
ncb_lsn	对应一个本地会话编号，与当前环境内的一次会话有着唯一对应的关系。成功执行了一次NCBCALL或NCBLISTEN命令后，函数会返回一个新的会话编号
ncb_num	指定本地名字的编号。伴随NCBADDNAME或NCBADDGRNAME命令的每一次调用，都会返回一个新编号。针对所有数据报命令，都必须使用一个有效的编号
ncb_buffer	指向数据缓冲区。对那些需要发送数据的命令，该缓冲区包含了要送出的实际数据；而对那些需要接收数据的命令，则包含了要从 Netbios函数返回的数据。对其

(续)

字 段	定 义
	他命令来说，如NCBENUM，缓冲区便是预定义的结构LANA_ENUM
ncb_length	以字节数为单位，指定缓冲区的长度。对于接收命令来说， Netbios会将该值设为收到的字节数。若指定的缓冲区不够大， Netbios就会返回NRC_BUFLEN错误
ncb_callname	指定远程应用的名字
ncb_name	指定应用程序已知的名字
ncb_rto	设定接收操作的超时期限。该值应设为500毫秒的一个整数倍数。若为1，表示没有超时限制。该值是为NCBCALL和NCBLISTEN命令设置的，它们会影响后续的NCBRECV命令
ncb_sto	设定发送操作的超时期限。该值应设为500毫秒的一个整数倍数。若为1，表示不存在超时限制。该值是为NCBCALL和NCBLISTEN命令设置的，它们会影响后续的NCBSEND和NCBCHAINSEND命令
ncb_post	指定异步命令完成后需要调用的后例程的地址。函数定义为： void CALLBACK PostRoutine(PNCB pncb); 其中，pncb指向已完成命令的网络控制块
ncb_lana_num	指定要在上面执行命令的LANA编号
ncb_cmd_cpl	指定操作的返回代码。异步操作进行期间，Netbios会将这个值设为NRC_PENDING
ncb_reserve	保留；必须为0
ncb_event	指定设置为“未传信”（Nonsignaled）状态的一个Windows事件对象的句柄。完成一个异步命令后，事件便会设置成它的“传信”（Signaled）状态。只应使用人工重设事件。假若ncb_command未设置ASYNCH标志，或者ncb_post不为0，那么该字段必须为0。否则， Netbios会返回NRC_ILLCMD错误

同步与异步

调用Netbios函数时，可选择进行同步调用，还是进行异步调用。所有 NetBIOS命令本身均是同步的。换言之，完成命令以前，会一直调用 Netbios块。而对一个NCBLISTEN命令来说，当有一个客户机建立了连接，或发生某种类型的错误时，对 Netbios的调用才会返回。要想异步调用一个命令，需要让NetBIOS命令同ASYNCH标志进行一次逻辑OR（或）运算。如指定了ASYNCH标志，那么必须在ncb_post字段中指定一个后例程（Post Routine），或必须在ncb_event字段中指定一个事件句柄。执行一个异步命令时，从Netbios返回的值是NRC_GOODRET (0x00)，但ncb_cmd_cplt字段会设为NRC_PENDING(0xFF)。除此以外， Netbios函数还会将NCB结构的ncb_cmd_cplt字段设为NRC_PENDING（待决），直到命令完成为止。命令完成后，ncb_cmd_cplt字段会设为该命令的返回值。 Netbios也会在完成后将ncb_retcode字段设为命令的返回值。

1.3 常规NetBIOS例程

本节将讨论一个基本的NetBIOS服务器应用程序。之所以首先拿服务器开刀，是由于服务器的设计决定了客户机的行为。由于大多数服务器都要求同时为多个客户提供服务，所以异步NetBIOS模型是最适合的。展示这个服务器应用程序例子时，我们同时用到了异步回调（CallBack）例程以及事件模型。但在我们首先展示的源码中，必须实现大多数 NetBIOS应用程序、都要用到的一些常规函数。程序清单 1-1取自文件Nbcommon.c，它可在本书配套光盘

的\Examples\Chapter01\Common目录下找到。贯穿全书的示范代码都会用到来自本文件的一系列基本函数。

程序清单1-1 常规NetBIOS例程(Nbcommon.c)

```
// Nbcommon.c

#include <windows.h>
#include <stdio.h>
#include <stdlib.h>

#include "nbcommon.h"

//
// Enumerate all LANA numbers
//
int LanaEnum(LANA_ENUM *lenum)
{
    NCB                 ncb;

    ZeroMemory(&ncb, sizeof(NCB));
    ncb.ncb_command = NCBENUM;
    ncb.ncb_buffer = (PUCHAR)lenum;
    ncb.ncb_length = sizeof(LANA_ENUM);
    if (Netbios(&ncb) != NRC_GOODRET)
    {
        printf("ERROR: Netbios: NCBENUM: %d\n", ncb.ncb_retcode);
        return ncb.ncb_retcode;
    }
    return NRC_GOODRET;
}

//
// Reset each LANA listed in the LANA_ENUM structure. Also, set
// the NetBIOS environment (max sessions, max name table size),
// and use the first NetBIOS name.
//
int ResetAll(LANA_ENUM *lenum, UCHAR ucMaxSession,
             UCHAR ucMaxName, BOOL bFirstName)
{
    NCB                 ncb;
    int                 i;

    ZeroMemory(&ncb, sizeof(NCB));
    ncb.ncb_command = NCBRESET;
    ncb.ncb_callname[0] = ucMaxSession;
    ncb.ncb_callname[2] = ucMaxName;
    ncb.ncb_callname[3] = (UCHAR)bFirstName;

    for(i = 0; i < lenum->length; i++)
    {
        ncb.ncb_lana_num = lenum->lana[i];
        if (Netbios(&ncb) != NRC_GOODRET)
        {
            printf("ERROR: Netbios: NCBRESET[%d]: %d\n",
```

```
                ncb.ncb_lana_num, ncb.ncb_retcode);
            return ncb.ncb_retcode;
        }
    }
    return NRC_GOODRET;
}

//
// Add the given name to the given LANA number. Return the name
// number for the registered name.
//
int AddName(int lana, char *name, int *num)
{
    NCB                 ncb;

    ZeroMemory(&ncb, sizeof(NCB));
    ncb.ncb_command = NCBADDNAME;
    ncb.ncb_lana_num = lana;
    memset(ncb.ncb_name, ' ', NCBNAMSZ);
    strncpy(ncb.ncb_name, name, strlen(name));

    if (Netbios(&ncb) != NRC_GOODRET)
    {
        printf("ERROR: Netbios: NCBADDNAME[lana=%d;name=%s]: %d\n",
            lana, name, ncb.ncb_retcode);
        return ncb.ncb_retcode;
    }
    *num = ncb.ncb_num;
    return NRC_GOODRET;
}

//
// Add the given NetBIOS group name to the given LANA
// number. Return the name number for the added name.
//
int AddGroupName(int lana, char *name, int *num)
{
    NCB                 ncb;

    ZeroMemory(&ncb, sizeof(NCB));
    ncb.ncb_command = NCBADDGRNAME;
    ncb.ncb_lana_num = lana;
    memset(ncb.ncb_name, ' ', NCBNAMSZ);
    strncpy(ncb.ncb_name, name, strlen(name));

    if (Netbios(&ncb) != NRC_GOODRET)
    {
        printf("ERROR: Netbios: NCBADDGRNAME[lana=%d;name=%s]: %d\n",
            lana, name, ncb.ncb_retcode);
        return ncb.ncb_retcode;
    }
    *num = ncb.ncb_num;
    return NRC_GOODRET;
}
```

```
//
// Delete the given NetBIOS name from the name table associated
// with the LANA number
//
int DelName(int lana, char *name)
{
    NCB                 ncb;
    ZeroMemory(&ncb, sizeof(NCB));
    ncb.ncb_command = NCBDELNAME;
    ncb.ncb_lana_num = lana;
    memset(ncb.ncb_name, ' ', NCBNAMSZ);
    strncpy(ncb.ncb_name, name, strlen(name));

    if (Netbios(&ncb) != NRC_GOODRET)
    {
        printf("ERROR: Netbios: NCBADDNAME[lana=%d;name=%s]: %d\n",
            lana, name, ncb.ncb_retcode);
        return ncb.ncb_retcode;
    }
    return NRC_GOODRET;
}

//
// Send len bytes from the data buffer on the given session (lsn)
// and lana number
//
int Send(int lana, int lsn, char *data, DWORD len)
{
    NCB                 ncb;
    int                 retcode;

    ZeroMemory(&ncb, sizeof(NCB));
    ncb.ncb_command = NCBSEND;
    ncb.ncb_buffer = (PUCHAR)data;
    ncb.ncb_length = len;
    ncb.ncb_lana_num = lana;
    ncb.ncb_lsn = lsn;

    retcode = Netbios(&ncb);

    return retcode;
}

//
// Receive up to len bytes into the data buffer on the given session
// (lsn) and lana number
//
int Recv(int lana, int lsn, char *buffer, DWORD *len)
{
    NCB                 ncb;

    ZeroMemory(&ncb, sizeof(NCB));
    ncb.ncb_command = NCBRECV;
    ncb.ncb_buffer = (PUCHAR)buffer;
    ncb.ncb_length = *len;
```

```
    ncb.ncb_lana_num = lana;
    ncb.ncb_lsn = lsn;

    if (Netbios(&ncb) != NRC_GOODRET)
    {
        *len = -1;
        return ncb.ncb_retcode;
    }
    *len = ncb.ncb_length;

    return NRC_GOODRET;
}
//
// Disconnect the given session on the given lana number
//
int Hangup(int lana, int lsn)
{
    NCB                 ncb;
    int                 retcode;

    ZeroMemory(&ncb, sizeof(NCB));
    ncb.ncb_command = NCBHANGUP;
    ncb.ncb_lsn = lsn;
    ncb.ncb_lana_num = lana;

    retcode = Netbios(&ncb);

    return retcode;
}

//
// Cancel the given asynchronous command denoted in the NCB
// structure parameter
//
int Cancel(PNCB pncb)
{
    NCB                 ncb;

    ZeroMemory(&ncb, sizeof(NCB));
    ncb.ncb_command = NCBCANCEL;
    ncb.ncb_buffer = (PUCHAR)pncb;
    ncb.ncb_lana_num = pncb->ncb_lana_num;

    if (Netbios(&ncb) != NRC_GOODRET)
    {
        printf("ERROR: NetBIOS: NCBCANCEL: %d\n", ncb.ncb_retcode);
          return ncb.ncb_retcode;
     }
     return NRC_GOODRET;
 }

 //
 // Format the given NetBIOS name so that it is printable. Any
 // unprintable characters are replaced by a period. The outname
 // buffer is the returned string, which is assumed to be at least
```

```
// NCBNAMSZ + 1 characters in length.
//
int FormatNetbiosName(char *nbname, char *outname)
{
    int        i;

    strncpy(outname, nbname, NCBNAMSZ);
    outname[NCBNAMSZ - 1] = '\0';
    for(i = 0; i < NCBNAMSZ - 1; i++)
    {
        // If the character isn't printable, replace it with a '.'
        //
        if (!((outname[i] >= 32) && (outname[i] <= 126)))
            outname[i] = '.';
    }
    return NRC_GOODRET;
}
```

在Nbcommon.c中，出现的第一个常规例程是LanaEnum。这是几乎所有NetBIOS应用都会用到的一个最基本的例程。该函数可列举一个指定系统上可用的所有 LANA编号。函数会将一个NCB结构初始化成0，将ncb_command字段设为NCBENUM，为ncb_buffer字段分配一个LANA_ENUM结构，并将ncb_length字段设为LANA_ENUM结构的长度。在NCB结构正确初始化之后，为执行NCBENUM命令，LanaEnum函数需要采取的唯一行动便是调用 Netbios函数。如大家所见，一个 NetBIOS命令的执行异常简单。对同步命令来说，自 Netbios返回的值可告诉我们命令是否成功执行。注意常数 NRC_GOODRET肯定意味着“ 成功 ”。

使用当前机器上可用的 LANA编号数量，以及各个实际的 LANA编号，一次成功的NetBIOS调用会填充完善指定的LANA_ENUM结构。LANA_ENUM结构的定义如下：

```
typedef struct LANA_ENUM
{
    UCHAR   length;
    UCHAR   lana[MAX_LANA + 1];
} LANA_ENUM, *PLANA_ENUM;
```

其中，length成员指出本地机器共有多少个 LANA编号。lana字段代表由实际的 LANA编号构成的一个数组。而length值指出lana数组内有多少个元素会被填充LANA编号。

接下去的一个函数是ResetAll（全部重设）。同样，该函数会在所有 NetBIOS应用中用到。对一个编写风格良好的NetBIOS程序来说，必须重设计划使用的每个 LANA编号。一旦拥有一个LANA_ENUM结构，并有来自LanaEnum的LANA编号，便可针对结构中的每个LANA编号，调用NCBRESET命令来重设它们。这正是 ResetAll要帮我们达到的目的；函数的第一个参数是LANA_ENUM结构。重设只要求函数将 ncb_command设为NCBRESET，并将ncb_lana_num设为它需要重设的LANA。注意尽管某些平台（比如 Windows 95）并不要求我们对打算使用的每个LANA编号进行重设，但最好还是那样做。 Windows NT要求我们在正式使用前对每个LANA编号进行重设；否则，对 Netbios的其他调用就会返回错误代码 52（亦即 NRC_ENVNOTDEF）。

除此以外，重设一个 LANA编号时，可通过 ncb_callname的字符字段，设置特定的NetBIOS环境参数。ResetAll的其他参数与这些环境设定是对应的。函数用 ucMaxSession参数来设置ncb_callname的字符0，它用于指定可同时进行的最大会话数量。通常，操作系统会强

制使用一个比最大值小的默认值。举个例子来说， Windows NT 4的最大默认值为64个并发会话。ResetAll将ncb_callname的字符2（用于指定可为每个LANA增加的最大NetBIOS名字数量）设为ucMaxName参数的值。同样，操作系统也会强加一个默认的最大值。最后， ResetAll会将字符3（用于NetBIOS客户机）设为它的 bFirstName参数的值。通过将此参数设为 TRUE，一个客户机便能将机器名作为自己的 NetBIOS进程名使用。因此，那个客户机可与一个服务器建立连接，并在不允许任何进入连接的前提下，向其发送数据。这一选项有效缩短了初始化时间。而假若将一个NetBIOS名字加入本地名字表，那么必须为此付出相应的代价。

要想将名字加入本地名字表，必须用到另一个常规函数： AddName。需要的参数就是想添加的名字，以及将其加到哪个 LANA编号。请记住，每个LANA编号永远对应一个名字表。如果你的应用程序需要与每个可用的 LANA通信，便需为每个LANA增加进程名。用于增加一个唯一名字的命令是 NCBADDNAME。必须使用的其他字段包括要为其增加名字的那个LANA编号，以及要实际增加的名字，后者必须复制到 ncb_name中。AddName首先会将ncb_name缓冲区初始化成空白，然后假定 name参数指向一个空中止字串。成功添加一个名字后， Netbios会在ncb_num字段中，返回同新增名字对应的 NetBIOS名字编号。可随数据报一起使用该值，以标定始发的 NetBIOS进程。有关数据报更深入的情况，我们会在本章的后面进行详细讨论。增加一个独一无二的名字时，经常遇到的一个错误是 NRC_DUPNAME。若网络中的另一个进程已使用了要增加的名字，便会出现此类错误。

AddGroupName的工作原理同AddName大致相同，只是它执行的命令是NCBADDGRNAME，而且永远不会出现什么NRC_DUPNAME错误。

DelName是另一个有紧密联系的函数，用于从名字表中删除一个 NetBIOS名字。它只要求指定打算删除的名字，以及从哪个 LANA编号上删除那个名字。

在程序清单1-1中，接下去的两个函数是 Send和Recv，用于在一个已经建立的会话中，进行数据的收发。这两个函数在工作方式上几乎完全相同，除了 ncb_command字段的设置以外。这个命令字段可设为 NCBSEND或NCBRECV。当然，用于收发数据的 LANA编号以及相应的会话编号也是必需的参数。若成功执行了一个 NCBCALL或NCBLISTEN命令，便会返回相应的会话编号。客户机利用 NCBCALL命令同一个已知的服务建立连接；而服务器使用 NCBLISTEN“等候”进入的客户机连接。若两个命令中有一个成功， NetBIOS接口便会建立一个会话，并为其赋予独一无二的整数标识符。 Send和Recv还需要用到映射到 ncb_buffer和ncb_length的参数。发送数据时， ncb_buffer指向那个包含了要送出的数据的缓冲区。在长度（length）字段中，指定了缓冲区中应当送出的字符数量。而在接收数据时，缓冲区（ buffer）字段指向在向其中复制数据的一个内存块。而长度字段指定了该内存块的大小。 Netbios函数返回之后，它会用成功接收到的字节数来更新长度字段。在一个面向会话的连接中，对数据的发送来说，要注意的一个重要的问题是在调用 Send函数时，接收方执行一个Recv函数之前，Send函数会一直等待下去。这意味着假如发送方送出大量数据，但接收方还没有读取它，便会耗用大量本地资源对数据进行缓冲。因此，一个良好的编程习惯是同时只执行少数几个NCBSEND或NCBCHAINSEND命令。为进一步解决这个问题，请换用 Netbios命令NCBSENDNA和NCBCHAINSENDNA命令。通过这两个命令，便不必从接收方那里等待收到确认消息，而是直接送出数据了事。

在程序清单1-1中，最后的两个函数是 Hangup和Cancel，分别用于关闭已经建立的会话，

或者取消一个尚未进行（待决）的命令。我们可调用 NetBIOS命令NCBHANGUP来从容关闭一个建好的会话。执行该命令时，对于指定的会话来说，所有尚未进行的接收调用都会中止，并返回一个“会话关闭”错误： NRC_SCLOSED(0x0A)。如果还存在任何没有执行的发送命令，则Hangup命令会暂时停止封锁执行，等那些命令完成了再说。无论命令正在传输数据，还是正在等待远程端执行一个接收命令，这种延迟都会发生。

1.3.1 会话服务器：异步回调模型

现在，我们已掌握了进行后续工作所需的基本 NetBIOS函数。接下来，且让我们看看用于监听“进入”客户机连接的服务器。这只是一个简单的回应反射服务器；从客户机那里接收到的任何数据都会立即返还回去。在程序清单 1-2中，我们展示了具体的服务器代码，其中利用的是异步回调函数。在本书配套光盘上，亦可找到这个 Cbnbsvr.c程序，位于 /Examples/Chapter01/Server文件夹下。注意一下函数 main，便会发现我们首先用 LanaEnum来列举所有可用的LANA编号，然后用ResetAll来重设每个LANA。记住在几乎所有 NetBIOS应用中，都要先采取这两个步骤。

程序清单1-2 异步回调服务器（Cbnbsvr.c）

```
// Cbnbsvr.c

#include <windows.h>
#include <stdio.h>
#include <stdlib.h>

#include "..\Common\nbcommon.h"

#define MAX_BUFFER      2048
#define SERVER_NAME     "TEST-SERVER-1"

DWORD WINAPI ClientThread(PVOID lpParam);

//
// Function: ListenCallback
//
// Description:
//    This function is called when an asynchronous listen completes.
//    If no error occurred, create a thread to handle the client.
//    Also, post another listen for other client connections.
//
void CALLBACK ListenCallback(PNCB pncb)
{
    HANDLE      hThread;
    DWORD       dwThreadId;

    if (pncb->ncb_retcode != NRC_GOODRET)
    {
        printf("ERROR: ListenCallback: %d\n", pncb->ncb_retcode);
        return;
    }
    Listen(pncb->ncb_lana_num, SERVER_NAME);

    hThread = CreateThread(NULL, 0, ClientThread, (PVOID)pncb, 0,
```

```
        &dwThreadId);
    if (hThread == NULL)
    {
        printf("ERROR: CreateThread: %d\n", GetLastError());
        return;
    }
    CloseHandle(hThread);

    return;
}

//
// Function: ClientThread
//
// Description:
//    The client thread blocks for data sent from clients and
//    simply sends it back to them. This is a continuous loop
//    until the session is closed or an error occurs. If
//    the read or write fails with NRC_SCLOSED, the session
//    has closed gracefully--so exit the loop.
//
DWORD WINAPI ClientThread(PVOID lpParam)
{
    PNCB        pncb = (PNCB)lpParam;
    NCB         ncb;
    char        szRecvBuff[MAX_BUFFER];
    DWORD       dwBufferLen = MAX_BUFFER,
                dwRetVal = NRC_GOODRET;
    char        szClientName[NCBNAMSZ+1];

    FormatNetbiosName(pncb->ncb_callname, szClientName);

    while (1)
    {
        dwBufferLen = MAX_BUFFER;

        dwRetVal = Recv(pncb->ncb_lana_num, pncb->ncb_lsn,
            szRecvBuff, &dwBufferLen);
        if (dwRetVal != NRC_GOODRET)
            break;
        szRecvBuff[dwBufferLen] = 0;
        printf("READ [LANA=%d]: '%s'\n", pncb->ncb_lana_num,
            szRecvBuff);

        dwRetVal = Send(pncb->ncb_lana_num, pncb->ncb_lsn,
            szRecvBuff, dwBufferLen);
        if (dwRetVal != NRC_GOODRET)
            break;
    }
    printf("Client '%s' on LANA %d disconnected\n", szClientName,
        pncb->ncb_lana_num);

    if (dwRetVal != NRC_SCLOSED)
    {
        // Some other error occurred; hang up the connection
        //
```

```
        ZeroMemory(&ncb, sizeof(NCB));
        ncb.ncb_command = NCBHANGUP;
        ncb.ncb_lsn = pncb->ncb_lsn;
        ncb.ncb_lana_num = pncb->ncb_lana_num;

        if (Netbios(&ncb) != NRC_GOODRET)
        {
            printf("ERROR: Netbios: NCBHANGUP: %d\n", ncb.ncb_retcode);
            dwRetVal = ncb.ncb_retcode;
        }
        GlobalFree(pncb);
        return dwRetVal;
    }
    GlobalFree(pncb);
    return NRC_GOODRET;
}

//
// Function: Listen
//
// Description:
//    Post an asynchronous listen with a callback function. Create
//    an NCB structure for use by the callback (since it needs a
//    global scope).
//
int Listen(int lana, char *name)
{
    PNCB        pncb = NULL;

    pncb = (PNCB)GlobalAlloc(GMEM_FIXED | GMEM_ZEROINIT, sizeof(NCB));
    pncb->ncb_command = NCBLISTEN | ASYNCH;
    pncb->ncb_lana_num = lana;
    pncb->ncb_post = ListenCallback;
    //
    // This is the name clients will connect to
    //
    memset(pncb->ncb_name, ' ', NCBNAMSZ);
    strncpy(pncb->ncb_name, name, strlen(name));
    //
    // An '*' means we'll take a client connection from anyone. By
    // specifying an actual name here, we restrict connections to
    // clients with that name only.
    //
    memset(pncb->ncb_callname, ' ', NCBNAMSZ);
    pncb->ncb_callname[0] = '*';

    if (Netbios(pncb) != NRC_GOODRET)
    {
        printf("ERROR: Netbios: NCBLISTEN: %d\n", pncb->ncb_retcode);
        return pncb->ncb_retcode;
    }
    return NRC_GOODRET;
}

//
```

```
//
// Function: main
//
// Description:
//    Initialize the NetBIOS interface, allocate some resources, add
//    the server name to each LANA, and post an asynch NCBLISTEN on
//    each LANA with the appropriate callback. Then wait for incoming
//    client connections, at which time, spawn a worker thread to
//    handle them. The main thread simply waits while the server
//    threads are handling client requests. You wouldn't do this in a
//    real application, but this sample is for illustrative purposes
//    only.
//
int main(int argc, char **argv)
{
    LANA_ENUM   lenum;
    int         i,
                num;

    // Enumerate all LANAs and reset each one
    //
    if (LanaEnum(&lenum) != NRC_GOODRET)
        return 1;
    if (ResetAll(&lenum, 254, 254, FALSE) != NRC_GOODRET)
        return 1;
    //
    // Add the server name to each LANA, and issue a listen on each
    //
    for(i = 0; i < lenum.length; i++)
    {
        AddName(lenum.lana[i], SERVER_NAME, &num);
        Listen(lenum.lana[i], SERVER_NAME);
    }
   while (1)
   {
       Sleep(5000);
   }
}
```

main接下来要做的事情是将进程名增加到打算用来接收连接的每个 LANA编号。通过一次循环，服务器会将自己的进程名 TEST-SERVER-1增加到每个LANA编号。通过这个名字，客户机才能连接我们的服务器（当然要用空格填充）。试图建立或接受一个连接时， NetBIOS名字中的每个字符都必须明确指定。对于这个问题，大家必须特别留意。编写 NetBIOS客户机和服务器代码时，最常见的问题便是名字的错误匹配。一定要注意用空格或其他字符来填充名字。空格是最常用的填充字符，因为可以列举出来或打印出来，总之，人眼能够辨别它的存在。

对服务器来说，最后一个也是最关键的一个步骤是执行大量 NCBLISTEN命令。Listen函数首先会分配一个NCB结构。使用异步NetBIOS调用时，我们递交的NCB结构必须自执行调用之时开始，一直持续到调用结束为止。这便要求我们在执行命令前动态分配每一个 NCB结构，或者维持一个全局性的NCB结构池，以便在异步调用中使用。对NCBLISTEN来说，应设置希望通过它进行调用的那个 LANA编号。注意在程序清单 1-1列出的源代码清单中，

NCBLISTEN命令需要同ASYNCH命令进行逻辑或（OR）运算。指定ASYNCH命令时，对ncb_post和ncb_event这两个字段来说，其中任意一个必须设为非零值。否则， Netbios调用就会出错，报告NRC_ILLCMD（非法命令）错误。在程序清单1-2中，Listen函数会将ncb_post字段设成我们的回调函数： ListenCallback。接下来， Listen函数将把ncb_name字段设为服务器进程的名字。这正是客户机需要与之建立连接的那个名字。函数也会将 ncb_callname字段的第一个字符设为一个星号（*），指出服务器可从任何客户机接受连接请求。如果不这样做，亦可在ncb_callname字段中设置一个特定的名字，只允许注册了那个特定名字的客户机建立与服务器的连接。最后， Listen会发出对Netbios的一个调用。调用立即便会完成， Netbios函数将已提交的NCB结构的ncb_cmd_cplt字段设为NRC_PENDING(0xFF) —— 表示“待决”，直到命令执行完毕为止。

一旦main完成了重设，并为每个LANA编号都投放了一个NCBLISTEN命令，主线程会进入一个连续的循环中。

注意 由于这个服务器仅是一个简单的例子，所以在设计上非常简陋。在你编写自己的NetBIOS服务器应用时，还可在主循环中进行其他处理；或者在主循环中，为某个LANA编号投放一个同步NCBLISTEN命令。

只有在一个LANA编号上接受了一个进入的连接时，回调函数才会执行。 NCBLISTEN命令接受了一个连接后，会调用由 ncb_post指定的函数，并将最初的 NCB结构作为参数使用。随后， ncb_retcode会设为返回代码。请务必留意对这个值的检查，了解客户机连接是否成功建立。若连接成功，会在ncb_retcode字段中返回一个NRC_GOODRET(0x00)值。

连接成功后，需针对同一个 LANA编号执行另一个NCBLISTEN命令。之所以要这样做，是由于一旦原始的监听操作成功，则服务器会停止在那个 LANA编号上对客户机的连接进行监听，直到递交了另一个 NCBLISTEN为止。因此，假如服务器需要频繁地为客户提供服务，便需在同一个LANA上投放多个NCBLISTEN命令，以便能够同时接受多个客户机发出的连接请求。最后，回调函数会创建一个特殊的线程，为客户机提供服务。在我们的这个例子中，线程只是简单地循环，并调用一个成块读入命令（ NCBRECV），紧接着调用一个成块发送命令（NCBSEND）。所以，我们在此实现的是一个简单的回应服务器，用于从建立连接的客户机那里读入消息，再将其原封不动地“反射”回去。除非客户机中断连接，否则客户机线程会一直循环下去。连接中断时，客户机线程会执行一个 NCBHANGUP命令，以便在自己的这一端关闭当前连接。随后，客户机线程释放 NCB结构占用的空间，并正常退出。

对面向连接的会话来说，数据是由最基层的协议加以缓冲的，所以并非一定要发出“待决”的调用。发出一个接收命令后， Netbios函数会将可用的数据立即传给现成的缓冲区，而且调用会立即返回。而假若没有数据可用，接收调用便会暂停，直到有数据可用，或者会话断开为止。同样的道理也适用于发送命令：若网络堆栈能通过线缆立即送出数据，或者能将数据缓存在堆栈中，调用便会立即返回。而假若系统没有足够的缓冲区空间来立即送出数据，发送调用便会暂停，直到有可用的空间为止。要想避免这种形式的数据延误，可对数据的收发使用ASYNCH（异步）命令。若执行的是异步发送或接收命令，那么相应的缓冲区必须有一个较大的容量，超出调用进程本身的范围之外。避免收发延误的另一个办法是使用 ncb_sto和ncb_rto这两个字段。其中， ncb_sto字段用于设置发送延时。若为其指定一个非零值，便相当于为命令的执行规定了一个超时时限。注意时间长度是以 500毫秒为单位指定的。如命令超

时，便需要立即返回，数据则不会送出。接收超时的设置（ ncb_rto）道理是一样的：如在预先规定好的时间内没有数据到达，则调用返回，没有数据输入缓冲区。

1.3.2 会话服务器：异步事件模型

程序清单1-3展示了与程序清单1-2类似的一个回应服务器程序，只是采用了 Win32事件作为传信机制。事件模型与回调模型相似，唯一的区别在于对回调模型来说，系统会在异步操作完成后执行你的代码；而对事件模型来说，程序必须通过对事件状态的检查，来核实操作是否完成。由于这些属于标准的 Win32事件，所以可在此选用任何同步例程，比如WaitForSingleEvent和WaitForMultipleEvents等等。事件模型显得更有效率，因为程序员必须为程序规定一个恰当的结构，有意检查完成与否。

就我们的这个事件模型服务器程序来说，它最开头的几步与回调服务器是完全一致的：

1) 列举LANA编号。

2) 重设每个LANA。

3) 为每个LANA增加服务器的名字。

4) 为每个LANA都执行（投放）一个监听命令。

唯一区别在于我们需要跟踪每一个待决的（即尚未返回的）监听命令，因为必须将事件的完成同初始化某个特定命令的各个 NCB块对应起来。程序清单1-3分配了一个由系列NCB结构构成的数组， NCB结构的数量则等同于 LANA编号的数量（因为我们希望针对每个编号都执行一个NCBLISTEN监听命令。除此以外，代码还为每个 NCB结构都创建了一个事件，对应于命令的“ 完成 ”。Listen函数会从数组中取得某个NCB结构，作为自己的参数使用。

程序清单1-3 异步事件服务器（Evnbsvr.c）

```
// Evnbsvr.c

#include <windows.h>
#include <stdio.h>
#include <stdlib.h>

#include "..\Common\nbcommon.h"

#define MAX_SESSIONS    254
#define MAX_NAMES       254

#define MAX_BUFFER      2048
#define SERVER_NAME     "TEST-SERVER-1"

NCB    *g_Clients=NULL;        // Global NCB structure for clients

//
// Function: ClientThread
//
// Description:
//    This thread takes the NCB structure of a connected session
//    and waits for incoming data, which it then sends back to the
//    client until the session is closed
//
DWORD WINAPI ClientThread(PVOID lpParam)
```

```
{
    PNCB        pncb = (PNCB)lpParam;
    NCB         ncb;
    char        szRecvBuff[MAX_BUFFER],
                szClientName[NCBNAMSZ + 1];
    DWORD       dwBufferLen = MAX_BUFFER,
                dwRetVal = NRC_GOODRET;

    // Send and receive messages until the session is closed
    //
    FormatNetbiosName(pncb->ncb_callname, szClientName);
    while (1)
    {
        dwBufferLen = MAX_BUFFER;
        dwRetVal = Recv(pncb->ncb_lana_num, pncb->ncb_lsn,
            szRecvBuff, &dwBufferLen);
        if (dwRetVal != NRC_GOODRET)
            break;

        szRecvBuff[dwBufferLen] = 0;
        printf("READ [LANA=%d]: '%s'\n", pncb->ncb_lana_num,
            szRecvBuff);

        dwRetVal = Send(pncb->ncb_lana_num, pncb->ncb_lsn,
            szRecvBuff, dwBufferLen);
        if (dwRetVal != NRC_GOODRET)
            break;
    }
    printf("Client '%s' on LANA %d disconnected\n", szClientName,
        pncb->ncb_lana_num);
    //
    // If the error returned from a read or a write is NRC_SCLOSED,
    // all is well; otherwise, some other error occurred, so hang up the
    // connection from this side
    //
    if (dwRetVal != NRC_SCLOSED)
    {
        ZeroMemory(&ncb, sizeof(NCB));
        ncb.ncb_command = NCBHANGUP;
        ncb.ncb_lsn = pncb->ncb_lsn;
        ncb.ncb_lana_num = pncb->ncb_lana_num;

        if (Netbios(&ncb) != NRC_GOODRET)
        {
                printf("ERROR: Netbios: NCBHANGUP: %d\n",
                    ncb.ncb_retcode);
                GlobalFree(pncb);
                dwRetVal = ncb.ncb_retcode;
            }
        }
        // The NCB structure passed in is dynamically allocated, so
        // delete it before we go
        //
        GlobalFree(pncb);
        return NRC_GOODRET;
    }
```

```
//
// Function: Listen
//
// Description:
//    Post an asynchronous listen on the given LANA number.
//    The NCB structure passed in already has its ncb_event
//    field set to a valid Windows event handle.
//
int Listen(PNCB pncb, int lana, char *name)
{
    pncb->ncb_command = NCBLISTEN | ASYNCH;
    pncb->ncb_lana_num = lana;
    //
    // This is the name clients will connect to
    //
    memset(pncb->ncb_name, ' ', NCBNAMSZ);
    strncpy(pncb->ncb_name, name, strlen(name));
    //
    // An '*' means we'll accept connections from anyone.
    // We can specify a specific name, which means that only a
    // client with the specified name will be allowed to connect.
    //
    memset(pncb->ncb_callname, ' ', NCBNAMSZ);
    pncb->ncb_callname[0] = '*';

    if (Netbios(pncb) != NRC_GOODRET)
    {
        printf("ERROR: Netbios: NCBLISTEN: %d\n", pncb->ncb_retcode);
        return pncb->ncb_retcode;
    }
    return NRC_GOODRET;
}
//
// Function: main
//
// Description:
//    Initialize the NetBIOS interface, allocate some resources, and
//    post asynchronous listens on each LANA using events. Wait for
//    an event to be triggered, and then handle the client
//    connection.
//
int main(int argc, char **argv)
{
    PNCB        pncb=NULL;
    HANDLE      hArray[64],
                hThread;
    DWORD       dwHandleCount=0,
                dwRet,
                dwThreadId;
    int         i,
                num;
    LANA_ENUM   lenum;

    // Enumerate all LANAs and reset each one
    //
```

```
if (LanaEnum(&lenum) != NRC_GOODRET)
    return 1;
if (ResetAll(&lenum, (UCHAR)MAX_SESSIONS, (UCHAR)MAX_NAMES,
        FALSE) != NRC_GOODRET)
    return 1;
//
// Allocate an array of NCB structures (one for each LANA)
//
g_Clients = (PNCB)GlobalAlloc(GMEM_FIXED | GMEM_ZEROINIT,
        sizeof(NCB) * lenum.length);
//
// Create the events, add the server name to each LANA, and issue
// the asynchronous listens on each LANA.
//
for(i = 0; i < lenum.length; i++)
{
    hArray[i] = g_Clients[i].ncb_event = CreateEvent(NULL, TRUE,
        FALSE, NULL);

    AddName(lenum.lana[i], SERVER_NAME, &num);
    Listen(&g_Clients[i], lenum.lana[i], SERVER_NAME);
}
while (1)
{
// Wait until a client connects
//
dwRet = WaitForMultipleObjects(lenum.length, hArray, FALSE,
    INFINITE);
if (dwRet == WAIT_FAILED)
{
    printf("ERROR: WaitForMultipleObjects: %d\n",
        GetLastError());
    break;
}
// Go through all the NCB structures to see whether more than one
// succeeded. If ncb_cmd_plt is not NRC_PENDING, there
// is a client; create a thread, and hand off a new NCB
// structure to the thread. We need to reuse the original
// NCB for other client connections.
//
for(i = 0; i < lenum.length; i++)
{
    if (g_Clients[i].ncb_cmd_cplt != NRC_PENDING)
    {
        pncb = (PNCB)GlobalAlloc(GMEM_FIXED, sizeof(NCB));
        memcpy(pncb, &g_Clients[i], sizeof(NCB));
        pncb->ncb_event = 0;

        hThread = CreateThread(NULL, 0, ClientThread,
            (LPVOID)pncb, 0, &dwThreadId);
        CloseHandle(hThread);
        //
        // Reset the handle, and post another listen
        //
        ResetEvent(hArray[i]);
```

```
                Listen(&g_Clients[i], lenum.lana[i], SERVER_NAME);
            }
        }
    }
    // Clean up
    //
    for(i = 0; i < lenum.length; i++)
    {
        DelName(lenum.lana[i], SERVER_NAME);
        CloseHandle(hArray[i]);
    }
    GlobalFree(g_Clients);

    return 0;
}
```

其中，main函数的首次循环需要遍历每一个可用的LANA编号，为其增加服务器名字，执行NCBLISTEN命令，并同时构建由事件句柄构成的一个数组。接下来调用WaitForMultipleObjects。在其中一个句柄收到信号之前（即进入传信状态之前），这个调用会一直等待下去。一旦事件控制数组中的一个或多个句柄进入传信状态，WaitForMultipleObjects调用便会完毕，代码会构建一个线程，用它读取进入的消息，并将其原封不动回送给客户机。随后，代码会为传信状态的NCB结构创建一个副本，将其传递进入客户机线程。之所以要这样做，是由于我们希望沿用最初的NCB，以执行另一个NCBLISTEN监听命令。要达到这个目的，可重设事件，并针对那个结构再次调用Listen。注意此时没有必要将整个结构都复制下来。在实际应用中，只需用到本地会话编号（ncb_lsn）和LANA编号（ncb_lana_num）就可以了。然而，要想同时容纳两个值，打算将其都传递给同一个线程参数，那么NCB结构是一个非常不错的容器。事件模型使用的客户机线程与回调模型使用的那个几乎完全相同，只是这里采用了GlobalFree语句。

异步服务器策略

注意对前述的两种服务器工作模式来说，都可能存在拒绝为客户机提供服务的情况。NCBLISTEN完成后，在调用回调函数之前，或在事件收到信号之前，都存在少许的延时。只有在经历了几个语句之后，服务器才会执行另一个NCBLISTEN命令。例如，假定服务器在LANA 2上接受了一个客户机的连接。随后，在服务器针对同一个LANA编号执行另一个NCBLISTEN之前，假如又有一个客户机试图建立连接，便会收到一个名为NRC_NOCALL(0x14)的错误信息。这意味着，对指定的名字来说，目前尚未在它上面执行NCBLISTEN。要防止此类情况的出现，服务器必须为每个LANA都执行多个NCBLISTEN命令。

从前述两个服务器应用的例子可以看出，异步命令的执行其实是非常容易的。ASYNCH标志可应用于几乎所有NetBIOS命令。只是要记住，传递给Netbios的NCB结构有一个全局性的范围。

1.3.3 NetBIOS会话客户机

NetBIOS客户机在设计上类似于异步事件服务器。在程序清单1-4中，我们展示了客户机程序使用的源代码。按照名字，客户机首先采取大家或已熟知的一系列初始化步骤。它为每

个LANA编号的名字表增添自己的名字，然后执行一个异步连接命令。主循环会等候某个事件进入传信状态。随后，代码遍历与执行的那个连接命令对应的所有 NCB结构，每个LANA编号都对应一个结构。它会检查ncb_cmd_cplt的状态。如发现它设为NRC_PENDING（待决），代码就会取消异步命令；若命令完成（亦即建立了连接），但NCB并不与传信的那个NCB相符（由来自WaitForMultipleObjects调用的返回值指定），代码便会断开连接。假如服务器正在自己那一端对每个LANA进行监听扫描，同时客户机正在尝试在其每个 LANA上建立连接，那么就可能出现成功建立多个连接的情况。此时，代码会用 NCBHANGUP命令关掉多余的连接——它只需通过一个信道进行通信。由于允许双方都同时尝试建立一个连接，所以连接成功的机率大增。

程序清单1-4 异步事件客户机（Nbclient.c）

```
// Nbclient.c

#include <windows.h>
#include <stdio.h>
#include <stdlib.h>

#include "..\Common\nbcommon.h"

#define MAX_SESSIONS      254
#define MAX_NAMES         254

#define MAX_BUFFER        1024

char    szServerName[NCBNAMSZ];

//
// Function: Connect
//
// Description:
//    Post an asynchronous connect on the given LANA number to
//    the server. The NCB structure passed in already has the
//    ncb_event field set to a valid Windows event handle. Just
//    fill in the blanks and make the call.
//
int Connect(PNCB pncb, int lana, char *server, char *client)
{
    pncb->ncb_command = NCBCALL | ASYNCH;
    pncb->ncb_lana_num = lana;

    memset(pncb->ncb_name, ' ', NCBNAMSZ);
    strncpy(pncb->ncb_name, client, strlen(client));

    memset(pncb->ncb_callname, ' ', NCBNAMSZ);
    strncpy(pncb->ncb_callname, server, strlen(server));

    if (Netbios(pncb) != NRC_GOODRET)
    {
        printf("ERROR: Netbios: NCBCONNECT: %d\n",
            pncb->ncb_retcode);
        return pncb->ncb_retcode;
    }
```

```
    return NRC_GOODRET;
}

//
// Function: main
//
// Description:
//    Initialize the NetBIOS interface, allocate some resources
//    (event handles, a send buffer, and so on), and issue an
//    NCBCALL for each LANA to the given server. Once a connection
//    has been made, cancel or hang up any other outstanding
//    connections. Then send/receive the data. Finally, clean
//    things up.
//
int main(int argc, char **argv)
{
    HANDLE      *hArray;
    NCB         *pncb;
    char         szSendBuff[MAX_BUFFER];
    DWORD        dwBufferLen,
                 dwRet,
                 dwIndex,
                 dwNum;
    LANA_ENUM    lenum;
    int          i;

    if (argc != 3)
    {
        printf("usage: nbclient CLIENT-NAME SERVER-NAME\n");
        return 1;
    }
    // Enumerate all LANAs and reset each one
    //
    if (LanaEnum(&lenum) != NRC_GOODRET)
        return 1;
    if (ResetAll(&lenum, (UCHAR)MAX_SESSIONS, (UCHAR)MAX_NAMES,
            FALSE) != NRC_GOODRET)
        return 1;
    strcpy(szServerName, argv[2]);
    //
    // Allocate an array of handles to use for asynchronous events.
    // Also allocate an array of NCB structures. We need one handle
    // and one NCB for each LANA number.
    //
    hArray = (HANDLE *)GlobalAlloc(GMEM_FIXED,
        sizeof(HANDLE) * lenum.length);
    pncb   = (NCB *)GlobalAlloc(GMEM_FIXED | GMEM_ZEROINIT,
        sizeof(NCB) * lenum.length);
    //
    // Create an event, assign it into the corresponding NCB
    // structure, and issue an asynchronous connect (NCBCALL).
    // Additionally, don't forget to add the client's name to each
    // LANA it wants to connect over.
    //
    for(i = 0; i < lenum.length; i++)
```

```
{
    hArray[i] = CreateEvent(NULL, TRUE, FALSE, NULL);
    pncb[i].ncb_event = hArray[i];

    AddName(lenum.lana[i], argv[1], &dwNum);
    Connect(&pncb[i], lenum.lana[i], szServerName, argv[1]);
}
// Wait for at least one connection to succeed
//
dwIndex = WaitForMultipleObjects(lenum.length, hArray, FALSE,
    INFINITE);
if (dwIndex == WAIT_FAILED)
{
    printf("ERROR: WaitForMultipleObjects: %d\n",
        GetLastError());
}
else
{
    // If more than one connection succeeds, hang up the extra
    // connection. We'll use the connection that was returned
    // by WaitForMultipleObjects. Otherwise, if it's still pending,
    // cancel it.
    //
    for(i = 0; i < lenum.length; i++)
    {
        if (i != dwIndex)
        {
            if (pncb[i].ncb_cmd_cplt == NRC_PENDING)
                Cancel(&pncb[i]);
            else
                Hangup(pncb[i].ncb_lana_num, pncb[i].ncb_lsn);
        }
    }
    printf("Connected on LANA: %d\n", pncb[dwIndex].ncb_lana_num);
    //
    // Send and receive the messages
    //
    for(i = 0; i < 20; i++)
    {
            wsprintf(szSendBuff, "Test message %03d", i);
            dwRet = Send(pncb[dwIndex].ncb_lana_num,
                pncb[dwIndex].ncb_lsn, szSendBuff,
                strlen(szSendBuff));
            if (dwRet != NRC_GOODRET)
                break;
            dwBufferLen = MAX_BUFFER;
            dwRet = Recv(pncb[dwIndex].ncb_lana_num,
                pncb[dwIndex].ncb_lsn, szSendBuff, &dwBufferLen);
            if (dwRet != NRC_GOODRET)
                break;
            szSendBuff[dwBufferLen] = 0;
            printf("Read: '%s'\n", szSendBuff);
        }
        Hangup(pncb[dwIndex].ncb_lana_num, pncb[dwIndex].ncb_lsn);
    }
```

```
        // Clean things up
        //
        for(i = 0; i < lenum.length; i++)
        {
            DelName(lenum.lana[i], argv[1]);
            CloseHandle(hArray[i]);
        }
        GlobalFree(hArray);
        GlobalFree(pncb);

        return 0;
    }
```

1.4 数据报的工作原理

“数据报”（Datagram）属于一种“无连接”的通信方法。作为发送方，只需用目标NetBIOS名字为发出的每个包定址，然后简单地送出了事。此时，不会执行任何检查，以确保数据的完整性、抵达顺序或者传输的可靠性等等。

发出一个数据报共有三种方式。第一种是指挥数据报抵达一个特定的（或唯一的）组名。这意味着只能有一个进程负责数据报的接收——亦即注册了目标名字的那个进程。第二种是将数据报发给一个组名。只有注册了指定组名的那些进程才有权接收消息。最后，第三种方式是将数据报广播到整个网络。局域网内任何一个工作站上的任何进程均有权接收这种数据报消息。请用NCBDGSEND命令将数据报发给一个唯一的名字，或者发给一个组名；要想进行广播通信，请用NCBDGSENDBC命令。

任何数据报发送命令的使用都非常简单。首先，将ncb_num字段设为自一个NCBADDNAME或NCBADDGRNAME命令返回的名字编号。这个编号对应着消息的发送端。然后，请将ncb_buffer设为一个缓冲区的地址，那个缓冲内应包含了需要实际发出的数据。接下来，将ncb_lana_num字段设为一个特定的LANA编号，数据报将通过该LANA发送出去。最后，将ncb_callname设为目标NetBIOS名字。这既可是一个独一无二的名字，亦可是个组名。若想发送广播数据报，仍请执行上述所有步骤，只是剔除最后一步：因为所有工作站均需接收消息，不必设定ncb_callname的值。

当然，在前述的每种情况下，都必须有一个对应的数据报接收命令，以便实际地接收数据。数据报是“无连接”的；如数据报抵达了一个客户机，但对方却没有一个已处在“待决”状态的接收命令，数据便会被无情地抛弃，客户机也没有办法恢复那个数据（除非服务器重新发出数据）。这正是数据报通信最大的一个缺点。然而，这个缺点也换来了速度快的优点。同面向连接的方法相比，数据报的传送速度要快得多，因为完全省去了错误检查、连接设置之类的开销。

至于数据报的接收，也有三种方法可供选择。头两种方法要用到NCBDGRECV命令。首先，为目标设一个特定名字（唯一名字或组名）的消息执行一个数据报接收命令。其次，针对目标定为进程NetBIOS名字表内任何一个名字的任何一个数据报，为其执行一个数据报接收命令。第三，可用NCBDGRECVBC命令为一个广播数据报执行一个接收命令。

注意 若数据报的目标设为由一个不同的进程注册的名字，那么除非两个进程都注册了一个组名，否则不可能为其投放一个接收命令。在后一种情况下，两个进程都会接收

到相同的消息。

为执行一个接收命令，请将ncb_num字段设为自一次成功NCBADDNAME或NCBADDGRNAME调用返回的名字编号。这个编号指出我们打算在哪个名字上“监听”进入的数据报。若将该字段设为0xFF，表示可接收发给该进程之NetBIOS名字表内的任何名字的数据报。此外，还要创建一个用来接收数据的缓冲区，并将ncb_buffer设为该缓冲区的地址，将ncb_length设为该缓冲区的大小。最后，将ncb_lana_num设为要想在上面等候数据报的LANA编号。若通过NCBDGRECV命令（或NCBDGRECVBC命令）对Netbios的调用成功返回，那么在ncb_length字段中，就会包含接收到的实际字节数，而ncb_callname会包含发送进程的NetBIOS名字。

程序清单1-5的代码包含了基本的数据报函数。所有发送命令均属于封锁调用——一旦命令执行，而且数据进入线缆传输，函数便会返回，而且不会由于数据过载而造成执行暂停即锁定。接收调用属于异步事件，因为我们不知道数据在那个LANA编号上抵达。使用的代码类似于事件模型中的面向会话的服务器。针对每个LANA，代码都会投放一个异步的NCBDGRECV（或NCBDGRECVBC）命令。除非某个命令成功，否则会一直等候下去。成功后，它会检查投放的每个命令，为那些成功的打印出消息，并取消仍处在“待决”状态中的那些命令。这个例子同时为定向／广播式发送与接收提供了相应的函数。可将该程序编译成一个示范性应用，将其配置成负责数据报的发送或接收。这个程序可接收几个命令行参数，允许用户指定要发送或接收的数据报数量、连续两次发送之间的延迟、是否用广播数据报来取代定向数据报、用于任何名字的数据报收据等等。

程序清单1-5 NetBIOS数据报示例（Nbdgram.c）

```
// Nbdgram.c

#include <windows.h>
#include <stdio.h>
#include <stdlib.h>

#include "..\Common\nbcommon.h"

#define MAX_SESSIONS            254
#define MAX_NAMES               254
#define MAX_DATAGRAM_SIZE       512

BOOL    bSender = FALSE,                // Send or receive datagrams
        bRecvAny = FALSE,               // Receive for any name
        bUniqueName = TRUE,             // Register my name as unique?
        bBroadcast = FALSE,             // Use broadcast datagrams?
        bOneLana = FALSE;               // Use all LANAs or just one?
char    szLocalName[NCBNAMSZ + 1],      // Local NetBIOS name
        szRecipientName[NCBNAMSZ + 1]; // Recipient's NetBIOS name
DWORD   dwNumDatagrams = 25,            // Number of datagrams to send
        dwOneLana,                      // If using one LANA, which one?
        dwDelay = 0;                    // Delay between datagram sends

//
// Function: ValidateArgs
//
```

```
// Description:
//    This function parses the command line arguments
//    and sets various global flags indicating the selections
//
void ValidateArgs(int argc, char **argv)
{
    int                i;

    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 'n':        // Use a unique name
                    bUniqueName = TRUE;
                    if (strlen(argv[i]) > 2)
                        strcpy(szLocalName, &argv[i][3]);
                    break;
                case 'g':        // Use a group name
                    bUniqueName = FALSE;
                    if (strlen(argv[i]) > 2)
                        strcpy(szLocalName, &argv[i][3]);
                    break;
                case 's':        // Send datagrams
                    bSender = TRUE;
                    break;
                case 'c':        // # of datagrams to send or receive
                    if (strlen(argv[i]) > 2)
                        dwNumDatagrams = atoi(&argv[i][3]);
                    break;
                case 'r':        // Recipient's name for datagrams
                    if (strlen(argv[i]) > 2)
                        strcpy(szRecipientName, &argv[i][3]);
                    break;
                case 'b':        // Use broadcast datagrams
                    bBroadcast = TRUE;
                    break;
                case 'a':        // Receive datagrams on any name
                    bRecvAny = TRUE;
                    break;
                case 'l':        // Operate on this LANA only
                    bOneLana = TRUE;
                    if (strlen(argv[i]) > 2)
                        dwOneLana = atoi(&argv[i][3]);
                    break;
                case 'd':        // Delay (millisecs) between sends
                    if (strlen(argv[i]) > 2)
                        dwDelay = atoi(&argv[i][3]);
                    break;
                default:
                    printf("usage: nbdgram ?\n");
                    break;
```

```
            }
        }
    }
    return;
}

//
// Function: DatagramSend
//
// Description:
//    Send a directed datagram to the specified recipient on the
//    specified LANA number from the given name number to the
//    specified recipient. Also specified is the data buffer and
//    the number of bytes to send.
//
int DatagramSend(int lana, int num, char *recipient,
                 char *buffer, int buflen)
{
    NCB               ncb;

    ZeroMemory(&ncb, sizeof(NCB));
    ncb.ncb_command = NCBDGSEND;
    ncb.ncb_lana_num = lana;
    ncb.ncb_num = num;
    ncb.ncb_buffer = (PUCHAR)buffer;
    ncb.ncb_length = buflen;

    memset(ncb.ncb_callname, ' ', NCBNAMSZ);
    strncpy(ncb.ncb_callname, recipient, strlen(recipient));

    if (Netbios(&ncb) != NRC_GOODRET)
    {
        printf("Netbios: NCBDGSEND failed: %d\n", ncb.ncb_retcode);
        return ncb.ncb_retcode;
    }
    return NRC_GOODRET;
}

//
// Function: DatagramSendBC
//
// Description:
//    Send a broadcast datagram on the specified LANA number from the
//    given name number. Also specified is the data buffer and the number
//    of bytes to send.
//
int DatagramSendBC(int lana, int num, char *buffer, int buflen)
{
     NCB               ncb;

     ZeroMemory(&ncb, sizeof(NCB));
     ncb.ncb_command = NCBDGSENDBC;
     ncb.ncb_lana_num = lana;
     ncb.ncb_num = num;
     ncb.ncb_buffer = (PUCHAR)buffer;
```

```
    ncb.ncb_length = buflen;

    if (Netbios(&ncb) != NRC_GOODRET)
    {
        printf("Netbios: NCBDGSENDBC failed: %d\n", ncb.ncb_retcode);
        return ncb.ncb_retcode;
    }
    return NRC_GOODRET;
}

//
// Function: DatagramRecv
//
// Description:
//    Receive a datagram on the given LANA number directed toward the
//    name represented by num. Data is copied into the supplied buffer.
//    If hEvent is not 0, the receive call is made asynchronously
//    with the supplied event handle. If num is 0xFF, listen for a
//    datagram destined for any NetBIOS name registered by the process.
//
int DatagramRecv(PNCB pncb, int lana, int num, char *buffer,
                 int buflen, HANDLE hEvent)
{
    ZeroMemory(pncb, sizeof(NCB));
    if (hEvent)
    {
        pncb->ncb_command = NCBDGRECV | ASYNCH;
        pncb->ncb_event = hEvent;
    }
    else
        pncb->ncb_command = NCBDGRECV;
    pncb->ncb_lana_num = lana;
    pncb->ncb_num = num;
    pncb->ncb_buffer = (PUCHAR)buffer;
    pncb->ncb_length = buflen;

    if (Netbios(pncb) != NRC_GOODRET)
    {
        printf("Netbos: NCBDGRECV failed: %d\n", pncb->ncb_retcode);
        return pncb->ncb_retcode;
    }
    return NRC_GOODRET;
}

//
// Function: DatagramRecvBC
//
// Description:
//    Receive a broadcast datagram on the given LANA number.
//    Data is copied into the supplied buffer. If hEvent is not 0,
//    the receive call is made asynchronously with the supplied
//    event handle.
//
int DatagramRecvBC(PNCB pncb, int lana, int num, char *buffer,
                   int buflen, HANDLE hEvent)
```

```
{
    ZeroMemory(pncb, sizeof(NCB));
    if (hEvent)
    {
        pncb->ncb_command = NCBDGRECVBC | ASYNCH;
        pncb->ncb_event = hEvent;
    }
    else
        pncb->ncb_command = NCBDGRECVBC;
    pncb->ncb_lana_num = lana;
    pncb->ncb_num = num;
    pncb->ncb_buffer = (PUCHAR)buffer;
    pncb->ncb_length = buflen;

    if (Netbios(pncb) != NRC_GOODRET)
    {
        printf("Netbios: NCBDGRECVBC failed: %d\n", pncb->ncb_retcode);
        return pncb->ncb_retcode;
    }
    return NRC_GOODRET;
}

//
// Function: main
//
// Description:
//    Initialize the NetBIOS interface, allocate resources, and then
//    send or receive datagrams according to the user's options
//
int main(int argc, char **argv)
{
    LANA_ENUM   lenum;
int         i, j;
char        szMessage[MAX_DATAGRAM_SIZE],
            szSender[NCBNAMSZ + 1];
DWORD      *dwNum = NULL,
            dwBytesRead,
            dwErr;

ValidateArgs(argc, argv);
//
// Enumerate and reset the LANA numbers
//
if ((dwErr = LanaEnum(&lenum)) != NRC_GOODRET)
{
    printf("LanaEnum failed: %d\n", dwErr);
    return 1;
}
if ((dwErr = ResetAll(&lenum, (UCHAR)MAX_SESSIONS,
    (UCHAR)MAX_NAMES, FALSE)) != NRC_GOODRET)
{
    printf("ResetAll failed: %d\n", dwErr);
    return 1;
}
//
```

```
// This buffer holds the name number for the NetBIOS name added
// to each LANA
//
dwNum = (DWORD *)GlobalAlloc(GMEM_FIXED | GMEM_ZEROINIT,
        sizeof(DWORD) * lenum.length);
if (dwNum == NULL)
{
    printf("out of memory\n");
    return 1;
}
//
// If we're going to operate on only one LANA, register the name
// on only that specified LANA; otherwise, register it on all
// LANAs
//
if (bOneLana)
{
    if (bUniqueName)
        AddName(dwOneLana, szLocalName, &dwNum[0]);
    else
        AddGroupName(dwOneLana, szLocalName, &dwNum[0]);
}
else
{
    for(i = 0; i < lenum.length; i++)
    {
        if (bUniqueName)
            AddName(lenum.lana[i], szLocalName, &dwNum[i]);
        else
            AddGroupName(lenum.lana[i], szLocalName, &dwNum[i]);
    }
}
// We are sending datagrams
//
if (bSender)
{
    // Broadcast sender
    //
    if (bBroadcast)
    {
        if (bOneLana)
        {
            // Broadcast the message on the one LANA only
            //
            for(j = 0; j < dwNumDatagrams; j++)
            {
                wsprintf(szMessage,
                    "[%03d] Test broadcast datagram", j);
                if (DatagramSendBC(dwOneLana, dwNum[0],
                    szMessage, strlen(szMessage))
                    != NRC_GOODRET)
                    return 1;
                Sleep(dwDelay);
            }
        }
```

```
else
{
    // Broadcast the message on every LANA on the local
    // machine
    //
    for(j = 0; j < dwNumDatagrams; j++)
    {
        for(i = 0; i < lenum.length; i++)
        {
            wsprintf(szMessage,
                "[%03d] Test broadcast datagram", j);
            if (DatagramSendBC(lenum.lana[i], dwNum[i],
                szMessage, strlen(szMessage))
                != NRC_GOODRET)
                return 1;
        }
                Sleep(dwDelay);
            }
        }
    }
    else
    {
        if (bOneLana)
        {
            // Send a directed message to the one LANA specified
            //
            for(j = 0; j < dwNumDatagrams; j++)
            {
                wsprintf(szMessage,
                    "[%03d] Test directed datagram", j);
                if (DatagramSend(dwOneLana, dwNum[0],
                    szRecipientName, szMessage,
                    strlen(szMessage)) != NRC_GOODRET)
                    return 1;
                Sleep(dwDelay);
            }
        }
        else
        {
            // Send a directed message to each LANA on the
            // local machine
            //
            for(j = 0; j < dwNumDatagrams; j++)
            {
                for(i = 0; i < lenum.length; i++)
                {
                    wsprintf(szMessage,
                        "[%03d] Test directed datagram", j);
                    printf("count: %d.%d\n", j,i);
                    if (DatagramSend(lenum.lana[i], dwNum[i],
                        szRecipientName, szMessage,
                        strlen(szMessage)) != NRC_GOODRET)
                        return 1;
                }
                Sleep(dwDelay);
```

```
                    }
                }
            }
        }
        else                // We are receiving datagrams
        {
            NCB     *ncb=NULL;
char    **szMessageArray = NULL;
HANDLE  *hEvent=NULL;
DWORD   dwRet;

// Allocate an array of NCB structure to submit to each recv
// on each LANA
//
ncb = (NCB *)GlobalAlloc(GMEM_FIXED | GMEM_ZEROINIT,
            sizeof(NCB) * lenum.length);
//
// Allocate an array of incoming data buffers
//
szMessageArray = (char **)GlobalAlloc(GMEM_FIXED,
        sizeof(char *) * lenum.length);
for(i = 0; i < lenum.length; i++)
    szMessageArray[i] = (char *)GlobalAlloc(GMEM_FIXED,
            MAX_DATAGRAM_SIZE);
//
// Allocate an array of event handles for
// asynchronous receives
//
hEvent = (HANDLE *)GlobalAlloc(GMEM_FIXED | GMEM_ZEROINIT,
        sizeof(HANDLE) * lenum.length);
for(i = 0; i < lenum.length; i++)
    hEvent[i] = CreateEvent(0, TRUE, FALSE, 0);

if (bBroadcast)
{
    if (bOneLana)
    {
        // Post synchronous broadcast receives on
        // the one LANA specified
        //
        for(j = 0; j < dwNumDatagrams; j++)
        {
            if (DatagramRecvBC(&ncb[0], dwOneLana, dwNum[0],
                szMessageArray[0], MAX_DATAGRAM_SIZE,
                NULL) != NRC_GOODRET)
                return 1;
            FormatNetbiosName(ncb[0].ncb_callname, szSender);
            printf("%03d [LANA %d] Message: '%s' "
                "received from: %s\n", j,
                ncb[0].ncb_lana_num, szMessageArray[0],
                szSender);
        }
    }
    else
    {
```

```
            // Post asynchronous broadcast receives on each LANA
            // number available. For each command that succeeded,
            // print the message; otherwise, cancel the command.
            //
            for(j = 0; j < dwNumDatagrams; j++)
            {
                for(i = 0; i < lenum.length; i++)
                {
                    dwBytesRead = MAX_DATAGRAM_SIZE;
                    if (DatagramRecvBC(&ncb[i], lenum.lana[i],
                        dwNum[i], szMessageArray[i],
                        MAX_DATAGRAM_SIZE, hEvent[i])
                        != NRC_GOODRET)
                        return 1;
                }
                dwRet = WaitForMultipleObjects(lenum.length,
                    hEvent, FALSE, INFINITE);
                if (dwRet == WAIT_FAILED)
                {
                    printf("WaitForMultipleObjects failed: %d\n",
                        GetLastError());
                    return 1;
                }
                for(i = 0; i < lenum.length; i++)
                {
                    if (ncb[i].ncb_cmd_cplt == NRC_PENDING)
                        Cancel(&ncb[i]);
                    else
                    {
                        ncb[i].ncb_buffer[ncb[i].ncb_length] = 0;
                        FormatNetbiosName(ncb[i].ncb_callname,
                            szSender);
                        printf("%03d [LANA %d] Message: '%s' "
                            "received from: %s\n", j,
                            ncb[i].ncb_lana_num,
                            szMessageArray[i], szSender);
                    }
                    ResetEvent(hEvent[i]);
                }
            }
        }
    }
    else
    {
        if (bOneLana)
        {
        // Make a blocking datagram receive on the specified
        // LANA number
        //
        for(j = 0; j < dwNumDatagrams; j++)
        {
            if (bRecvAny)
            {
                // Receive data destined for any NetBIOS name
                // in this process's name table
```

```
            //
            if (DatagramRecv(&ncb[0], dwOneLana, 0xFF,
                szMessageArray[0], MAX_DATAGRAM_SIZE,
                NULL) != NRC_GOODRET)
                return 1;
        }
        else
        {
            if (DatagramRecv(&ncb[0], dwOneLana,
                dwNum[0], szMessageArray[0],
                MAX_DATAGRAM_SIZE, NULL)
                != NRC_GOODRET)
                return 1;
        }
        FormatNetbiosName(ncb[0].ncb_callname, szSender);
        printf("%03d [LANA %d] Message: '%s' "
               "received from: %s\n", j,
               ncb[0].ncb_lana_num, szMessageArray[0],
               szSender);
    }
}
else
{
    // Post asynchronous datagram receives on each LANA
    // available. For all those commands that succeeded,
    // print the data; otherwise, cancel the command.
    //
    for(j = 0; j < dwNumDatagrams; j++)
    {
        for(i = 0; i < lenum.length; i++)
        {
            if (bRecvAny)
            {
                // Receive data destined for any NetBIOS
                // name in this process's name table
                //
                if (DatagramRecv(&ncb[i], lenum.lana[i],
                    0xFF, szMessageArray[i],
                        MAX_DATAGRAM_SIZE, hEvent[i])
                        != NRC_GOODRET)
                        return 1;
                }
                else
                {
                    if (DatagramRecv(&ncb[i], lenum.lana[i],
                        dwNum[i], szMessageArray[i],
                        MAX_DATAGRAM_SIZE, hEvent[i])
                        != NRC_GOODRET)
                        return 1;
                }
            }
            dwRet = WaitForMultipleObjects(lenum.length,
                hEvent, FALSE, INFINITE);
            if (dwRet == WAIT_FAILED)
            {
```

```
                printf("WaitForMultipleObjects failed: %d\n",
                    GetLastError());
                return 1;
            }
            for(i = 0; i < lenum.length; i++)
            {
                if (ncb[i].ncb_cmd_cplt == NRC_PENDING)
                    Cancel(&ncb[i]);
                else
                {
                    ncb[i].ncb_buffer[ncb[i].ncb_length] = 0;
                    FormatNetbiosName(ncb[i].ncb_callname,
                        szSender);
                    printf("%03d [LANA %d] Message: '%s' "
                        "from: %s\n", j, ncb[i].ncb_lana_num,
                        szMessageArray[i], szSender);
                }
                ResetEvent(hEvent[i]);
            }
        }
    }
}
// Clean up
//
for(i = 0; i < lenum.length; i++)
{
    CloseHandle(hEvent[i]);
    GlobalFree(szMessageArray[i]);
}
GlobalFree(hEvent);
        GlobalFree(szMessageArray);
    }
    // Clean things up
    //
    if (bOneLana)
        DelName(dwOneLana, szLocalName);
    else
    {
        for(i = 0; i < lenum.length; i++)
            DelName(lenum.lana[i], szLocalName);
    }
    GlobalFree(dwNum);

    return 0;
}
```

编译好这个例子后，请进行下述测试，以便对数据报的工作原理心中有数。考虑到学习研究的目的，应运行该应用的两份“实例”，不过要在不同的机器上运行。如在同一台机器上运行，尽管仍然可行，但却无法体会一些重要的概念。在同一台机器上运行时，每一方的LANA编号都对应于相同的协议。但我们更感兴趣的是对应不同协议的情况。在表 1-5中，我们总结了应进行试验的命令。除此以外，后面的表 1-6还列出了对于这个示范程序来说，可考虑选用的命令行参数。

表1-5 Nbdgram.c的命令

客户机命令	服务器命令
Nbdgram /n:CLIENT01	Nbdgram /s /n:SERVER01 /r:CLIENT01
Nbdgram /n:CLIENT01 /b	Nbdgram /s /n:SERVER01 /b
Nbdgram /g:CLIENTGROUP	Nbdgram /s /r:CLIENTGROUP

表1-6 Nbdgram.c的命令参数

标 志	含 义
/n:my-name	注册唯一名字my-name
/g:group-name	注册组名group-name
/s	发送数据报（默认情况下，示例接收数据报）
/c:n	发送或接收n个数据报
/r:receiver	指定数据报的目标NetBIOS名字
/b	使用广播数据报
/a	为任何NetBIOS名字投放接收命令（将ncb_num设为0xFF）
/l:n	只在LANA n上执行所有操作（默认情况下，为每个LANA编号都会投放发送和接收命令）
/d:n	在连续两次发送之间，等待n毫秒的时间

针对第三个命令，请在不同的机器上运行多个客户机。这样可营造一个服务器将同一条消息发给一个组的环境，正在等候数据的每位组员都会接收到消息。此外，请用 /l:x命令行参数试验所列命令的不同组合。其中， x代表一个有效的LANA编号。这个参数的作用是将程序模式从在所有LANA上执行命令，切换为只在列出的LANA上执行命令。举个例子来说，命令Nbdgram /n:CLIENT01 /l:0的作用是让应用程序只监听LANA 0上的进入数据报，忽略抵达其他任何LANA上的任何数据。除此以外，参数 /a只对客户机才有意义。这个标志意味着接收命令需要取得发给任何 NetBIOS名字的数据报，只要那些名字已得到了进程的注册。在我们的例子中，这个参数意义不大，因为客户机仅注册了一个名字。尽管如此，大家至少可从中看出如何进行编程。大家可试着对代码进行修改，以便为命令行的每个“ /n:名字 ”选项注册一个名字。服务器启动时，接收标志只设为已由客户机注册的一个名字。尽管 NCBDGRECV命令没有指出一个特定的名字，客户机仍会接收到数据。

1.5 其他NetBIOS命令

迄今为止，我们讨论的所有命令都在某种程度上涉及一个会话的建立；通过会话或数据报进行数据的发送或接收；以及与之相关的一些主题。另外，还有少数几个命令专门负责信息的获取。其中包括适配器状态命令（ NCBASTAT）和查找名字命令（ NCBFINDNAME）。在后续的两个小节中，我们打算分别对这两个命令进行解释。最后一节则准备通过一种利于编程的形式，将LANA编号同它们的协议对应起来（实际并非一个 NetBIOS函数；之所以要讨论它，是由于可藉着它收集到大量有用的 NetBIOS信息）。

1.5.1 适配器状态

利用NCBASTAT命令，可取得与本地计算机及其 LANA编号有关的信息。要想通过程序，从Windows 95及Windows NT 4中查知机器的MAC地址，这个命令也是唯一可行的途径。但随着Windows 2000和Windows 98的问世，由于它们引入了新的 IP Helper（IP助手）函数，所以

使得MAC地址的查找变得更加容易。然而，对其他Win32平台来说，适配器（通常是“网卡”）状态命令仍是我们的唯一选择。

命令及其语法是很易理解的，但通过两种不同的方式来调用函数时，会对数据的返回产生影响。适配器状态命令会返回一个 ADAPTER_STATUS结构，紧接着是大量 NAME_BUFFER结构。对结构的定义如下：

```
typedef struct _ADAPTER_STATUS {
    UCHAR   adapter_address[6];
    UCHAR   rev_major;
    UCHAR   reserved0;
    UCHAR   adapter_type;
    UCHAR   rev_minor;
    WORD    duration;
    WORD    frmr_recv;
    WORD    frmr_xmit;
    WORD    iframe_recv_err;
    WORD    xmit_aborts;
    DWORD   xmit_success;
    DWORD   recv_success;
    WORD    iframe_xmit_err;
    WORD    recv_buff_unavail;
    WORD    t1_timeouts;
    WORD    ti_timeouts;
    DWORD   reserved1;
    WORD    free_ncbs;
    WORD    max_cfg_ncbs;
    WORD    max_ncbs;
    WORD    xmit_buf_unavail;
    WORD    max_dgram_size;
    WORD    pending_sess;
    WORD    max_cfg_sess;
    WORD    max_sess;
    WORD    max_sess_pkt_size;
    WORD    name_count;
} ADAPTER_STATUS, *PADAPTER_STATUS;
typedef struct _NAME_BUFFER {
    UCHAR   name[NCBNAMSZ];
    UCHAR   name_num;
    UCHAR   name_flags;
} NAME_BUFFER, *PNAME_BUFFER;
```

其中，特别值得注意的字段当属 MAC地址（adapter_address）、数据报最大长度（max_dgram_size）以及最大会话数（max_sess）。此外，name_count字段告诉我们总共返回了多少个NAME_BUFFER结构。对每个LANA来说，最多能支持的NetBIOS名字为254个，所以我们可选择提供一个足够大的缓冲区，容下所有名字；或将 ncb_length设为0，仅调用一次适配器状态命令。Netbios函数返回之后，它会提供必要的缓冲区大小设置。

要想调用NCBASTAT，需要设置的字段包括 ncb_command，ncb_buffer，ncb_length，ncb_lana_num以及ncb_callname。若ncb_callname的第一个字符是一个星号（*），那么尽管会执行一个状态命令，但只有那些由调用进程增添的 NetBIOS名字才会返回。然而，如果用一个适配器状态命令调用 Netbios，在当前进程的名字表内增添一个“唯一”名字，然后在ncb_callname字段中使用那个名字，那么所有 NetBIOS名字都会在本地进程的名字表内注册，

另外还要加上由系统注册的任何名字。除命令在其上面执行的那台机器以外，还可针对另一台机器，执行一镒适配器状态查询。要想做到这一点，请将 ncb_callname字段设为远程工作站的机器名。

注意 记住所有Microsoft机器名字都将其第16个字节设为0，应该用空格来替代它。

示范程序Astat.c是一个简单的适配器状态程序，可针对所有LANA适配器运行指定的查询。此外，通过/l:LOCALNAME标志，亦可在本地机器上执行命令，只是要求提供完整的名字表。/r:REMOTENAME标志则针对某个指定的机器名，执行一次远程查询。

使用适配器状态命令时，有几方面的问题需要注意。首先，对一台设置成“多主机”或“多宿主”的机器来说，拥有的 MAC地址不止一个。由于NetBIOS没办法调查一个LANA到底与哪个适配器以及协议绑定到一起，所以此时需要由我们负责，对返回的值进行过滤。此外，假如安装了远程访问服务（RAS），系统也会为那些连接分配对应的 LANA编号。若RAS连接已经断开（即未建立连接），那些LANA上的适配器状态会为MAC地址返回全零值。若RAS连接已经建立，MAC地址便与RAS分配给其所有虚拟网络设备的MAC地址相同。最后，在执行一次远程适配器状态查询时，必须通过两台机器均已正确安装的一种传送协议进行。举个例子来说，系统命令Nbtstat（亦即NCBASTAT的一个命令行版本）只能通过 TCP/IP传送协议执行它的查询。若远程机器尚未安装TCP/IP协议，命令便会失败。

1.5.2 查找名字

NCBFINDNAME命令只能在Windows NT及Windows 2000操作系统上使用，用于调查一个指定的NetBIOS名字是由谁注册的。要想进行一次成功的查找名字查询，进程必须在名字表内添加其唯一名字。该命令要求设置的字段包括命令本身、 LANA编号、缓冲区以及缓冲区的长度。查询返回的是一个 FIND_NAME_HEADER结构，以及数量不定的 FIND_NAME_BUFFER结构。结构定义如下：

```
typedef struct _FIND_NAME_HEADER {
    WORD    node_count;
    UCHAR   reserved;
    UCHAR   unique_group;
} FIND_NAME_HEADER, *PFIND_NAME_HEADER;

typedef struct _FIND_NAME_BUFFER {
    UCHAR   length;
    UCHAR   access_control;
    UCHAR   frame_control;
    UCHAR   destination_addr[6];
    UCHAR   source_addr[6];
    UCHAR   routing_info[18];
} FIND_NAME_BUFFER, *PFIND_NAME_BUFFER;
```

和适配器状态命令一样，假如 NCBFINDNAME命令在执行时将缓冲区的长度设为 0，那么Netbios函数会返回所需的长度，同时返回一个名为 NRC_BUFLEN的错误。

对一次成功的查询来说，参照返回的 FIND_NAME_HEADER结构，便可知道一个名字是注册成唯一名字，还是注册成组名。假如unique_group字段的值为0，表示它是一个“唯一”名字；值为1，则表示是个组名。node_count字段指出总共返回了多少个FIND_NAME_BUFFER结

构。通过FIND_NAME_BUFFER结构，我们可获知大量信息。在这些信息中，大多数只有在协议这一级才有用处。目前，我们最感兴趣的是 destination_addr（目标地址）和source_addr（源地址）两个字段。其中， source_addr字段包含了注册指定名字的那个网络适配器的 MAC地址；而destination_addr包含了执行查询的那个适配器的MAC地址。

针对本地机器的任何LANA编号，均可执行一次查找名字查询。对于本地网络的任何有效LANA编号来说，返回的数据应当是完全相同的。例如，我们可针对一个 RAS连接执行该命令，以判断一个名字是否在远程网络上进行了注册。在 Windows NT 4.0中，存在着一个不该存在的错误：若通过 TCP/IP执行一次查找名字查询，那么 Netbios会返回虚假的信息。因此，若计划在Windows NT 4.0下使用这种查询，请务必选择与另一种传送协议对应的 LANA编号，不要依赖TCP/IP。

1.5.3 将传送协议同LANA编号对应起来

本节打算讨论如何将TCP/IP和NetBEUI这样的传送协议同它们的LANA编号对应 起来。由于应用程序打算采用的传送协议不同，我们也需要解决不同系列的一些潜在问题，所以最好能够先以程序化的方式。来查找这些传送协议。用固有的 NetBIOS调用是无法做到这一点的，但在Windows NT 4和Windows 2000下却可通过Winsock 2做到。一个名为WSAEnumProtocols的Winsock 2函数可返回与可用的传送协议有关的信息（第5和第6章对WSAEnumProtocols进行了更详细的解释）。尽管Winsock 2可加到Windows 95中，而且也是Windows 98的默认安装选项，但令人遗憾的是，在这些平台保存的协议资料中，并不包括我们真正需要的任何NetBIOS信息。

对于Winsock 2，现在不打算作深入探讨，那属于本书第二部分的主题。牵涉到的基本步骤包括先用 WSAStartup函数装载 Winsock 2，调用WSAEnumProtocols，再对调用返回的WSAPROTOCOL_INFO结构进行检查核实。在本书配套 CD提供的Nbproto.c中，包含了用于执行这种查询的代码。

WSAEnumProtocols函数需要用到一个缓冲区地址参数，对应一个数据块；另外还需用到一个缓冲区长度参数。首先用一个空缓冲区地址和长度0来调用该函数。当然，调用会失败，但随后在缓冲区长度参数字段，会返回所需缓冲区的真实长度。拿到了正确的长度之后，再次调用这个函数即可。WSAEnumProtocols替返回它发现的协议数量。同时，WSAPROTOCOL_INFO会成为一个很大的结构，其中包含了大量字段，但我们在此感兴趣的只有szProtocol，iAddressFamily和iProtocol。假如iAddressFamily等于AF_NETBIOS，则iProtocol的绝对值便对应于由字串szProtocol指定的那种协议的数量。除此以外， ProvidedId GUID可用于将返回的协议同协议的预定义GUID对应起来。

采用这种方法，仅存在着一个方面的不足。在 Windows NT和Windows 2000下，对于LANA 0上安装的任何协议来说， iProtocol字段的值都为0x80000000。这是由于协议0是为特殊用途而保留的；分配了LANA 0的任何协议都有一个0x80000000的值，所以只需注意对这个值的检查就可以了。

1.6 平台问题

在下述平台上实施NetBIOS时，请留意一些特殊的限制。

1.6.1 Windows CE

NetBIOS接口未在Windows CE中实现。尽管重定向器提供了对 NetBIOS名字和名字解析的支持，但却没有提供相应的编程接口的支持。

1.6.2 Windows 9x

就普通用户广泛采用的Windows 95和Windows 98（含第二版）这两种操作系统来说，要注意几处特别的系统错误。在这些平台上，若要为任何 LANA增加任何NetBIOS名字，务必首先重设所有LANA编号。这是由于重设任何一个 LANA，都会完全破坏其他 LANA的名字表。所以，应避免写出像下面这样的代码：

```
LANA_ENUM    lenum;
// Enumerate the LANAs
for(i = 0; i < lenum.length; i++)
{
    Reset(lenum.lana[i]);
    AddName(lenum.lana[i], MY_NETBIOS_NAME);
}
```

此外，对Windows 95来说，它根本不会在对应于 TCP/IP协议的LANA上尝试执行一个异步NCBRESET命令。从最开始，便不应以异步形式来执行该命令，因为在通过 LANA进行任何后续的操作之前，首先必须完成一次重设。如试图强行以异步形式执行一个 NCBRESET命令，程序便会在NetBIOS TCP/IP虚拟设备驱动程序（VXD）中，导致一个严重错误，不得不重新启动计算机。

1.6.3 常规问题

进行面向会话的通信时（与“无连接”通信相对），其中一方可送出自己希望的、任意多的数据。但实际上，若非接收方投放一条接收命令，确认数据已经收到，否则作为发送方，会将数据缓存起来。对发送命令来说， NCBSENDNA和NCBCHAINSENDNA这两个NetBIOS命令是它们的“毋需确认”版本。如果不想让自己的发送命令等候来自接收方的确认信息，那么可考虑使用这两个命令。由于 TCP/IP在最基层的协议中提供了自己的收到确认机制，所以发送命令的这两个版本（毋需接收方确认）在其行为上，类似于要求确认的版本。

1.7 小结

尽管NetBIOS接口功能非常强大，但可惜的是，由于问世较早，它在功能上存在着诸多限制，已不适合现在的需要。它的优点之一是“与协议无关”——应用程序可通过 TCP/IP、NetBEUI以及SPX/IPX运行。NetBIOS同时提供了面向连接和无连接的通信模式。与 Winsock接口相比， NetBIOS接口的一项重要优点在于：它提供了一种统一的名字解析及注册方法。换言之，对一个NetBIOS应用来说，它只需一个NetBIOS名字便可工作。Winsock应用则不同，假如它利用了不同的协议，那么每种协议的定址方案都要考虑到（详见本书第二部分）。在第2章中，我们向大家引入“重定向器”的概念。重定向器实际是“邮槽”和“命名管道”密不可分的一部分。在第3和第4章将深入讨论“邮槽”和“命名管道”。

第2章 重 定 向 器

Windows使应用程序能通过操作系统内建的文件系统服务在网络上通信。有时候，我们将之称为“网络操作系统”（NOS）能力。本章准备利用Windows 95、Windows 98、Windows NT、Windows 2000和Windows CE等均含有的Windows文件系统组件，向大家展示这些网络连接能力。本章的目的是让大家理解这些能力与邮槽和命名管道连网技术的关系。邮槽和命名管道连网技术分别是第3章和第4章的主题。

若应用程序希望访问本地系统中的文件，需要依赖操作系统来满足 I/O（输入／输出）请求。我们通常把它称为“本地 I/O”。例如，在一个应用程序打开或关闭文件时，需要由操作系统来决定如何访问包含了指定文件内容的一个设备。找到设备后， I/O请求会被转发给一个本地设备驱动程序。通过网络来访问一个设备也同样。然而， I/O请求必须通过网络转发给对应的远程设备。我们将其称为“ I/O重定向”（I/O Redirection）。例如， Windows允许我们将一个本地磁盘标识符（如 E:）映射或重定向到远程计算机上的一个目录共享入口。应用程序若指出自己需要使用 E:时，操作系统便会将 I/O请求自动重定向至一个设备，那个设备叫作“重定向器”（Redirector）。重定向器会建立到远程计算机的一个网络信道，以便访问指定的远程目录。随后，应用程序可自由使用一些常规的文件系统 API函数，比如ReadFile（读文件）和WriteFile（写文件）等。虽然实际是通过网络访问的，但表面上却与访问本地文件无异。

本章着重讲解了如何通过重定向器将普通的 I/O请求“重定向”到远程设备。本章内容对于以后的学习异常重要，重定向机制是邮槽和命名管道技术的基础。首先，我们打算解释如何通过网络，使用“多UNC提供者”（ Multiple UNC Provider, MUP）资源定位符，通过“通用命名规范”（Universal Naming Convention, UNC）来引用远程文件。随后，我们讲解了MUP如何调用一个网络提供者，从而揭示出怎样通过一个重定向器，在“服务器消息块”（Server Message Block, SMB）协议的帮助下，在不同的计算机之间建立数据通信。最后，我们探讨了网络安全方面的一些问题。使用基本的文件 I/O操作，通过网络来访问文件时，这些安全问题是必须考虑到的。

2.1 通用命名规范

“ UNC路径” 为网络文件及设备的访问建立了一套统一的规范。它最大的特点便是不必指定或引用一个已映射到远程文件系统的本地驱动器字母。这一点非常重要，因为应用程序可变得“与驱动器字母无关”。在复杂的网络环境中，应用程序不必对此有任何顾忌。同引用本地驱动器字母的方法相比， UNC名字要优越得多，因为在访问共享资源时，不必担心用光有限的驱动器字母的问题。另外，驱动器字母的分配也和具体的用户有着密切的联系 —— 如果进程在你的用户环境中不能运行，便无法利用由你规定的驱动器字母映射关系。

UNC名字完全解决了这些问题，它的格式如下：

\\[服务器]\[共享名]\[路径]

第一部分是\\[服务器]，必须以两个反斜杠开头，紧跟着一个服务器名字。服务器的名字

代表着网络中的一台远程服务器，我们想访问的远程文件便位于其中。在 UNC名字中，第二部分是\[共享名]，它对应着远程服务器上的一个“ 共享入口 ”或者“ 共享位置 ”。所谓“ 共享位置 ”，实际就是文件系统中的一个目录（包括根目录），表示可共享的资源便放在这个位置下面，是其他机器获取共享资源的“ 入口 ”。而第三部分\[路径] 对应的是共享位置下的某个具体目录（或子目录）。举个例子来说，假定现在有一台名为 Myserver的服务器，在其本地硬盘驱动器D:\上设置了一个共享目录，名为 D:\Myfiles\CoolMusic，并将这一长串名字简化为“ Myshare ”这个易记的“ 共享名 ”。现在，假定该共享目录下含有一个名为 Sample.mp3文件。那么，假如网络中其他任何一台机器想引用（访问）这个 MP3音乐文件，只需像下面这样指定它的UNC名字即可：

\\Myserver\Myshare\Sample.mp3

可以看出，与其将一个本地驱动器映射到共享目录 Myshare，还不如通过网络用UNC名字来直接引用一个文件 —— 因为所有机器使用的都是同样的UNC名字！

若通过UNC名字在网络中引用文件，应用程序便不必关心通过网络建立连接的细节，这显然是一种非常出色的设计。使用 UNC名字，系统便可非常轻松地定位网络服务器共享目录以及文件路径。网络通信的所有细节都是由网络提供者的“ 重定向器 ”来负责控制的，本章稍后即会对此进行详细论述。完成了第 3和第4章的学习之后，大家便会知道邮槽和命名管道技术非常依赖UNC名字。

在图2-1中，我们展示了Windows环境下在网络操作系统上建立 UNC连接所需的一些常规组件。此图也显示了数据流在客户机与服务器的 NOS组件之间逐渐推进的情况。以前面的UNC路径\\Myserver\Myshare\Sample.mp3为基础，本章将对每一个组件进行细致讲解，并阐述通过一个网络来打开这个文件的过程。

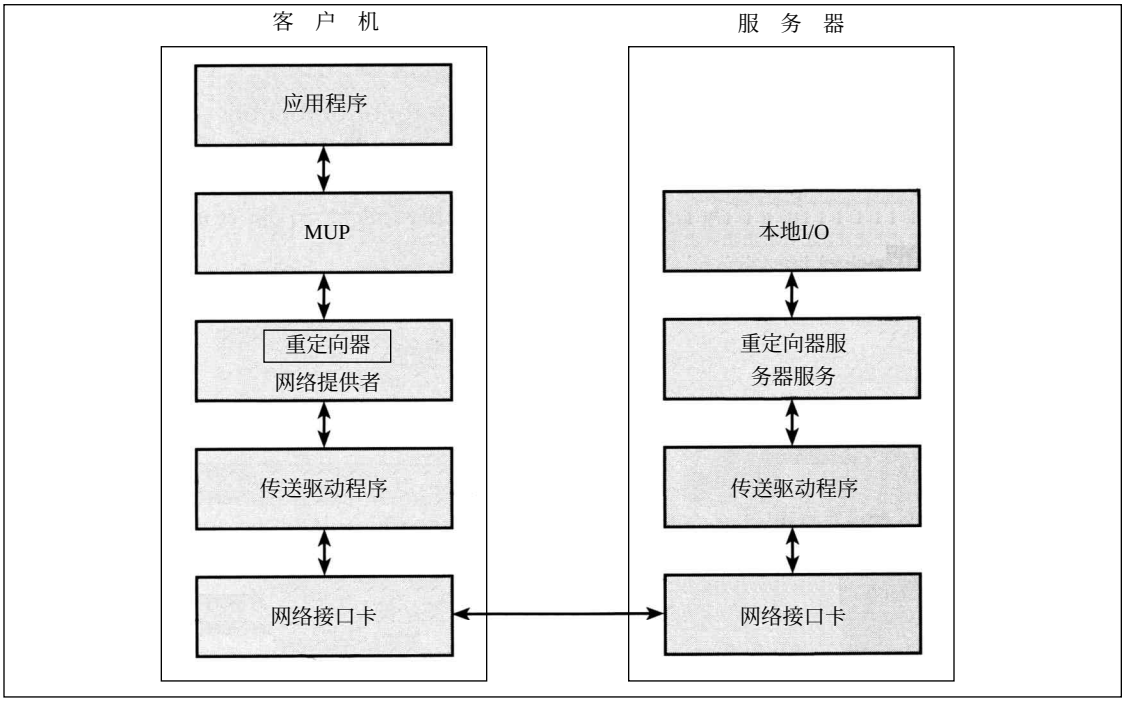

图2-1 重定向器的各个组件

2.2 多UNC提供者

MUP如图2-1是一种资源定位器，负责选择具体的网络提供者，来满足 UNC连接请求。所谓“网络提供者”（Network Provider），其实就是一种服务，可通过网络硬件访问位于一台远程计算机上的资源（如文件和打印机）。MUP需要通过一个网络提供者，在以 UNC名字为基础的所有文件及打印机I/O请求上提供服务，建立通信关系。

Windows NT、Windows 2000、Windows 95以及Windows 98均支持多个网络提供者的安装。举个例子来说， Windows平台支持一个名为“ Microsoft网络用户”（Client for Microsoft Network）的网络提供者。当然，亦可安装除微软以外的其他网络提供者，比如 Novell公司的Novell Client v3.01 for Windows 95/98等等。换句话说，一次可能有多个网络提供者为一个UNC请求提供服务。但必须注意的是， Windows CE目前仅支持一个网络提供者：“ Microsoft网络用户！”

MUP的基本任务便是决定具体由哪个网络提供者来满足一个 UNC请求。为作出这个决定，MUP需将请求中提到的 UNC名字发给已经安装好的每一个提供者（以并行方式）。若某个网络提供者表明自己能够提供 UNC名字牵涉到的那一种服务， MUP便会将请求中剩余的部分发给它。如果有多个提供者都表示能够服务一个 UNC请求，MUP便会根据优先级，挑选出最恰当的一个提供者。网络提供者的优先级是由各个提供者在系统内安装的顺序决定的。在Windows NT、Windows 2000、Windows 95和Windows 98中，要想对这种优先级进行改动，可考虑对注册表（Registry）中一个名为ProviderOrder的键值进行修改。它的位置在：

```
\HKEY_LOCAL_MACHINE
 \SYETEM
  \CurrentControlSet
   \Control
    \NetworkProvider
     \Order
```

根据优先级，已安装的各个网络提供者会按照先后顺序排列。由于 Windows CE只有一个网络提供者，所以根本就不会用 MUP来解析UNC名字。相反， UNC请求会直接进入那个唯一的提供者。

2.3 网络提供者

如前所述，网络提供者实际只是一种服务，通过网络硬件来访问位于远程计算机上的共享资源，比如文件和打印机等等。事实上，这正是网络操作系统的一种核心功能。网络提供者具备的最主要的功能之一便是将本地磁盘标识符（如 E:）重定向至远程机器上的一个磁盘目录。作为提供者，它必须能为 UNC连接请求提供服务。在 Windows环境中，网络提供者需要向操作系统展示出一个重定向器，从而做到这一点。

Windows最有特色的一个网络提供者称为“ Microsoft网络用户”，以前叫作“ Microsoft网络提供者”（MSNP）。打开前述的注册表目录，排在第一名的往往便是“ MSNP32”。通过MSNP，可在Windows NT 4、Windows 2000、Windows 95、Windows 98和Windows CE间自由地通信。但要注意的是， Windows CE不支持多个网络提供者，只提供了对 MSNP的内建客

户端支持。

2.4 重定向器简介

“重定向器”由网络提供者展示给用于接收和处理远程 I/O服务请求的操作系统。要做到这一点，它需要格式化服务请求消息，再将其发给远程计算机的重定向器服务器服务。远程机器的重定向器服务器服务收到这个请求之后， 会以发出本地I/O请求的方式，来满足这一请求。由于重定向器需要为较高级的服务（如 MUP）提供I/O服务，所以会在应用程序面前，将网络层的工作细节隐藏起来。这样一来，应用程序便毋需为重定向器提供协议特有的一些参数。因此，我们认为网络提供者是“与协议无关”的。换言之，在几乎任何网络配置中，应用程序都能正常运作。

MSNP提供了一个特殊的重定向器，可直接与网络传送层和 NetBIOS打交道，以便在客户机与服务器之间建立通信。本书第 1章讨论的NetBIOS API函数提供了一系列编程接口，正好可以做相同的事情。由 MSNP提供的这个重定向器有一个通俗的名字，叫作“ LAN管理器重定向器”（LAN Manager Redirector）。之所以叫这个名字，是由于当初设计它的时候，针对的便是老式的Microsoft LAN Manager软件，以便为MS-DOS应用程序提供网络操作系统能力（欲了解NetBIOS编程接口的详情，请参阅第 1章）。NetBIOS接口可通过大量网络协议进行通信。这便使得MSNP重定向器具有了“与协议无关”的特性，即应用程序不必关注某种网络协议的具体工作细节！若在自己的程序中使用了 MSNP重定向器，便能通过TCP/IP、NetBEUI甚至IPX/SPX进行通信。显然，这是一种非常有价值的特性，因为无论网络在物理上是如何构成的，程序都可以正常地实现通信。然而，我们仍应留意一个重要的问题。两个应用程序要想通过网络相互通信，那么对两个工作站来说，必须有一种协议是“通用”的。换言之，两者必须至少安装同一种通信协议。举个例子来说，假定工作站 A只安装了TCP/IP，而工作站B只安装了IPX。此时，尽管MSNP重定向器是“与协议无关”的，但仍会对此一筹莫展，无法通过一个网络在这两个工作站之间建立正常的通信。

MSNP重定向器与其他工作站通信时，需要向对方的“重定向器服务器”服务发送消息。这些消息采用一种固定的结构形式，称为 SMB。至于重定向器具体如何收发消息，要由“服务器消息块文件共享”（Server Message Block File Sharing）协议来决定，或简称SMB协议。

2.5 服务器消息块

SMB协议最早是由微软及 Intel于80年代末期联合开发成功的。当时的设计宗旨是让远程的文件系统能由 MS-DOS程序进行“透明”的访问。发展至今，这种协议的作用变得非常单一，就是以SMB数据结构的形式，让 Windows MSNP重定向器与远程工作站的 MSNP服务器服务进行通信。在 SMB数据结构中，总共包含了三个基本组件：命令代码、命令特有的以及用户数据。

SMB协议采用的是一个非常简单的“客户机请求／服务器响应”传输模型。 MSNP重定向器创建一个SMB结构时，需要在“命令代码”字段中指定一个特定的请求。若命令要求的是发送数据，比如一条 SMB写指令，数据便会随结构一道传送出去。随后， SMB结构会通过一种像TCP/IP这样的传送协议传给一个远程工作站的服务器服务。远程工作站的服务器服务会对收到的客户机请求进行处理，然后将一个 SMB响应数据结构传回客户机。

现在，大家知道了通过MSNP重定向器建立通信时，需要用到哪些基本组件。接下来，且让我们看看假定通过一个网络打开 \\Myserver\Myshare\Sample.mp3，那么各组件的通信情况是怎样的。如下所示：

1) 使用CreateFile这个API函数，应用程序向本地操作系统提交一个请求，要求打开\\Myserver\Myshare\Sample.mp3。

2) 根据从UNC路径描述中获得的信息，本地（本机）操作系统的文件系统判断出该 I/O（输入／输出）请求的目的地是一台远程机器，名为 \\Myserver，所以将此请求传递给MUP。

3) MUP调查出该I/O请求发给的是一个MSNP提供者，因为网上的\\Myserver机器正在使用NetBIOS名字解析机制。

4) I/O请求随即传给MSNP提供者的重定向器。

5) 重定向器将此请求格式化成一条 SMB消息，要求打开包含在远程 \Myshare目录下的Sample.mp3文件。

6) 格式化好的SMB消息终于通过一种网络传送协议，正式送入网络。

7) 名为\\Myserver的服务器从网上接收到这个SMB请求，并将请求传给服务器的MSNP重定向器服务器服务。

8) 服务器的重定向器服务提交一个本地I/O请求，希望打开位于\Myshare这个共享位置处的Sample.mp3文件。

9) 服务器的重定向器服务格式化好一条SMB响应消息，指出本地打开文件的I/O请求是成功，还是失败。

10) 通过一种网络传送协议，服务器的这条SMB响应消息返回客户机。

11) MSNP重定向器收到服务器的这条SMP响应消息，并向本机操作系统传递一个返回代码。

12) 本机操作系统再将该代码返回给当初应用程序的 CreateFile API请求。

从中可以看出，MSNP重定向器必须经历大量步骤，才能让一个应用程序访问到远程资源。当然，MSNP重定向器也需要对网络资源提供访问控制服务。这其实正是“网络安全”机制的一部分。

2.6 安全问题

这里对安全问题的讨论只限于通过网络对资源的访问。但在深入探讨如何保证通过网络安全访问一个资源之前，首先要对本地机器（本机）的安全机制有一定程度的了解。针对像文件和目录这样的系统资源，Windows NT和Windows 2000都同时提供了本地和远程访问控制的能力。在此，需要将这些资源看作需要“保密”的对象。若应用程序试图访问一个保密对象，操作系统便会检查该程序，看它是否有那个对象的“访问权限”。访问权限共分几种，最基本的包括读、写以及执行。Windows NT和Windows 2000是通过“安全描述符”以及“访问令牌”来实现访问控制的。

2.6.1 安全描述符

对需要保密的所有对象来说，都应包含一个特殊的“安全描述符”（Security Descriptor），规定自己特有的访问控制信息。在一个安全描述符内，包含了一个 SECURITY_DESCRIPTOR

结构，以及对应的安全信息，包括：

- 所有人安全标识符（Security Identifier，SID）：代表对象所有人。
- 组SID：指定对象的主组所有人。
- 授权访问控制列表（Discretionary Access Control List，DACL）：指定能由哪些人进行什么类型的访问。访问类型包括读、写及执行权限等。
- 系统访问控制列表（System Access Control List，SACL）：指定需要为哪些类型的访问企图生成审核记录，以便将来稽查。

就程序本身来说，无权直接对一个安全描述符结构的内容进行修改。然而，我们可利用一些Win32安全API函数，来进行这种形式的修改。通过这些 API，可进行安全信息的设置与获取。在本章结束的时候，会向大家展示具体如何操作。

1. 访问控制列表和访问控制实体

安全描述符的DACL和SACL字段指定的是访问控制列表（access control list, ACL）。这些字段可能包含了零值，或包含着更多的访问控制条目（ access control entities, ACE）。每个ACE都负责控制或监视着指定用户或用户组对一个对象的访问。在一个 ACE中，包含着下述类型的访问控制信息：

- 一个SID，标识着该ACE应用于哪个用户或用户组。
- 一个掩码，指出象读、写和执行这样的访问权限。
- 一个标志，指出ACE所属的类型 —— 允许访问、拒绝访问或者系统审核等。

注意系统审核ACE仅在SACL中使用，而允许访问和拒绝访问这两种 ACE类型在DACL中使用。在图2-2中，我们展示了一个文件对象的DACL设置情况。

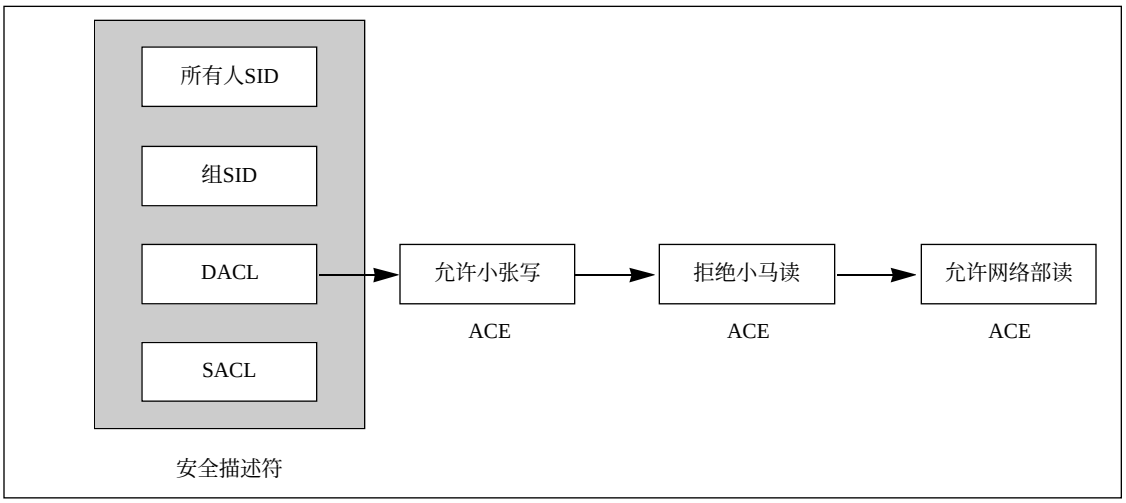

图2-2 设置了DACL的一个文件对象

对一个保密对象来说，假如没有 DACL（它的DACL已被API函数SetSecurityDescriptor Dacl设为一个空值），那么系统就允许所有人拥有对它的“完全”访问权限。若对象有一个DACL，便会参照DACL中的ACE设定，只允许进行那些明确允许的访问。假如 DACL中没有设置ACE，那么任何人都不能以任何形式访问该对象。类似地，假定 DACL设置了一些“允许访问”ACE，那么凡是ACE中未包括进来的所有用户及用户组，系统都会拒绝他们的访问。

大多数情况下，我们只需指定“允许访问” ACE就足够了。但下面这种情况是个例外：如果为某个用户组设置了一个“只允许访问” ACE，那么可能还要设置一些“拒绝访问”ACE，将那个组的某些特定成员排除在外。要想达到目的，必须将一个用户的“拒绝访问”ACE放在整个组的“允许访问”ACE之前。注意ACE的顺序是至关重要的，因为除非访问被允许或拒绝，否则系统会一直按序读取 ACE。用户的“拒绝访问”ACE必须放在最前面；否则的话，在系统读到用户组的“允许访问” ACE时，便会为那些受到限制的用户开放不该开放的权限。

图2-2展示了如何设置DACL，以便向一个名为“网络部”的用户组开放读权限。假定在这个组内，包含了小安、小张和小王三名用户。我们的目的是为组内除小安之外的所有人都授予“读”权限。为做到这一点，必须在为整个“网络部”设置“允许访问”权限之前，先为小安设置“拒绝访问”权限。在图 2-中，我们也为小张设置了一个允许访问 ACE，允许他拥有“写”的权力。记住应用程序并不能直接操作 ACL，它们必须通过一系列涉及安全保密的API函数，来做这些事情。

2. 安全标识符

我们知道，在保密对象的安全描述符及 ACE设置中，包含了一个安全标识符（ SID)。所谓SID，其实是一个独一无二的值，用于标定一个用户帐号、一个用户组帐号或者一次登录会话。对于像Windows NT服务器域这样的安全总监来说，它们需要在一个安全帐号数据库中对SID信息加以维护。用户登录进入后，系统便会从数据库中取得用户的 SID，并将其置于一个用户的“访问令牌”中。以后凡是在牵涉到 Windows NT安全的地方，系统都可根据令牌中包含的这个SID，正确判断出用户的身份。

2.6.2 访问令牌

用户登录进入一个Windows NT系统后，系统会对用户的帐号名和密码进行验证，两者统称为“登录凭据”。若用户登录成功（即验证通过），系统便会创建一个相应的访问令牌，并将该用户的SID分配给它。对面向这名用户执行的任何进程来说，都会拥有该访问令牌的一个拷贝。若一个进程试图访问一个受到保护（即保密）的对象，访问令牌中的这个 SID便会与DACL中分配给SID的访问权限进行对比。

2.7 网络安全

到目前为止，我们已简要解释了如何在本地机器上实施安全机制。接下来，打算来看看通过网络访问一个受安全保护的对象时，如何确保安全。如早先所述， MSNP重定向器负责着不同计算机之间的资源访问。此外， MSNP重定向器还要通过创建用户会话凭据的形式，在客户机与服务器之间建立一条安全通信链路。

会话凭据

共有两种类型的用户凭据：主登录凭据和会话凭据。若用户坐在一台工作站面前，登录进入机器，那么他／她的用户名和密码便成为主要的凭据集，并保存在一个访问令牌中。对任何用户来说，在任何时刻，只能存在一套主登录凭据。若用户尝试建立与一个远程资源的连接（无论通过映射驱动器的方式，还是用 UNC名字加以连接），便会对用户的主凭据进行验

证，判断他／她拥有对远程资源的何种访问权限。注意对 Windows NT和Windows 2000来说，用户可选择提供一套不同的凭据，以便在针对远程资源进行验证的时候使用。若用户提供的访问凭据是有效的，MSNP重定向器便会在用户的计算机同远程资源之间建立一个会话。重定向器会将会话与会话凭据关联到一起，后者包含了用户计算机用于对远程资源连接进行验证的那些凭据副本。在用户计算机与远程服务器之间，一次只能建立一套会话凭据。若机器 B提供两个共享位置（共享点）：\Hack和\Slash，而且假如机器 A的用户将\Hack映射成G，并将\Slash映射成H，那么两个会话都会共享同样的会话凭据，因为它们引用的都是同一个远程服务器。

MSNP重定向器服务器服务负责着远程服务器上的安全访问控制。若 MSNP重定向器服务器试图访问一个受到保护的对象（保密对象），便会用会话凭据来创建一个远程访问令牌。从此时开始，对安全机制的管理便和在本地机器上别无二致。在图 2-3中，我们展示了MSNP重定向器如何通过Windows NT域安全机制，来建立安全凭据。

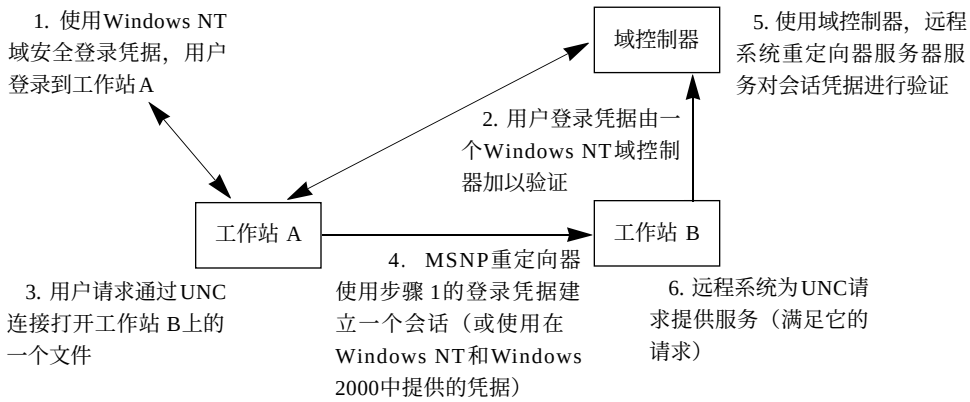

图2-3 安全凭据的演示

2.8 一个实例

使用MSNP重定向器，通过一个网络， Win32应用程序可分别使用CreateFile、ReadFile和WriteFile这三个API函数，来创建、访问以及修改文件。 Windows NT和Windows 2000是目前支持Win32安全机制的唯一平台。在程序清单 2-1中，我们演示了如何编写一个简单的应用程序，通过UNC连接来创建一个文件。这些代码可在配套光盘上找到，位于 \Examples\Chapter02目录下。

程序清单2-1 简单的文件创建示例

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

void main(void)
{
    HANDLE FileHandle;
    DWORD BytesWritten;

    // Open a handle to file \\Myserver\Myshare\Sample.txt
    if ((FileHandle = CreateFile("\\\\Myserver\\Myshare\\Sample.txt",
```

```
            GENERIC_WRITE | GENERIC_READ,
            FILE_SHARE_READ | FILE_SHARE_WRITE, NULL,
            CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL))
            == INVALID_HANDLE_VALUE)
        {
            printf("CreateFile failed with error %d\n", GetLastError());
            return;
        }

        // Write 14 bytes to our new file
        if (WriteFile(FileHandle, "This is a test", 14,
            &BytesWritten, NULL) == 0)
        {
            printf("WriteFile failed with error %d\n", GetLastError());
            return;
        }

        if (CloseHandle(FileHandle) == 0)
        {
            printf("CloseHandle failed with error %d\n", GetLastError());
            return;
        }
    }
```

2.9 小结

本章向大家简介了 Windows重定向器。利用它，应用程序可通过网络来访问远程的Windows文件系统资源。我们首先解释了重定向器在网络上的基本通信方法。随后，讨论了在应用程序使用重定向器时，由Windows NT和Windows 2000提供的一系列安全特性。下面两章将分别讨论邮槽和命令管道这两种非常有用的技术。在其涉及到的所有网络通信中，邮槽和命名管道都要依赖于“重定向器”。

第3章 邮 槽

Microsoft Windows NT、Windows 2000、Windows 95和Windows 98（含第二版）——但不包括Windows CE——提供了一种简单的单向“进程间通信”（interprocess communication, IPC）机制。这个机制的名字非常古怪，叫作“邮槽”（Mailslot）。用最简单的话来说，通过邮槽，客户机进程可将消息传送或广播给一个或多个服务器进程。在同一台计算机的不同进程之间，或在跨越整个网络的不同计算机的进程之间，协助进行消息的传输。用邮槽来开发应用程序是一件非常简单的事情，不要求对TCP/IP或IPX这样的基层网络传送协议有着非常深入的了解。由于邮槽是围绕一个广播通信体系设计出来的，所以当然不能指望能通过它实现数据的“可靠”传输。然而，在某些特殊类型的网络编程环境中，假如对数据传输的可靠性要求不高，那么邮槽仍然是一种非常有价值的技术。

例如，要想在自己办公室的所有人员之间建立一个简单的传信系统，便可考虑用邮槽来设计这个程序。想象自己的办公室环境拥有大量工作站。恰巧，办公室目前苏打短缺。每隔几分钟，每名工作站用户都有兴趣知道苏打机里还剩下多少可乐。利用邮槽，便可很轻易地设计出这种程序。我们可简单地编制一个邮槽客户端应用，用它监视苏打数量，并以五分钟为周期，将剩余的可乐数量广播给每一名感兴趣的工作站用户。由于邮槽并不能担保广播消息的可靠传输，所以有些工作站用户也许收不到所有的更新通知。但在这种情况下，少数的传输失败并不是个问题，因为消息每隔五分钟便会发送一遍，所以即使偶尔出错，收到信息的频率也相当高，足以让工作站用户跟上目前的最新情况。

邮槽最大的一个缺点便是只允许从客户机到服务器，建立一种不可靠的单向数据通信。而另一方面，邮槽最大的一个优点在于，它们使客户机应用能够非常容易地将广播消息发送给一个或多个服务器应用。

本章将向大家解释如何开发一个实际的邮槽客户机／服务器应用。在深入解释消息的大小问题之前，首先要介绍邮槽的一系列命名规范，它们对邮槽的总体行为进行着控制。接下来，我们将详细讲述如何开发一个基本的客户机／服务器应用。在本章结束时，则会告诉大家邮槽一些已知的问题，以及存在的各种局限，并提出相应的解决方案。

3.1 邮槽实施细节

邮槽是围绕Windows文件系统接口设计出来的。客户机和服务器应用需要使用标准的Win32文件系统I/O（输入／输出）函数，比如ReadFile和WriteFile等等，以便在邮槽上收发数据，同时利用Win32文件系统的命名规则。邮槽必须依赖Windows重定向器，通过一个“邮槽文件系统”（Mailslot File System, MSFS），来创建及标识邮槽。第2章已对Windows重定向器进行了比较详细的说明。

3.1.1 邮槽的名字

对邮槽进行标识时，需遵守下述命名规则：

\\server\Mailslot\[path]name

请将上述字串分为三段来看：\\server、\Mailslot和\[path]name。第一部分\\server对应于服务器的名字，我们要在上面创建邮槽，并在在上面运行服务器程序。第二部分 \Mailslot是一个“硬编码”的固定字串，用于告诉系统这个文件名从属于 MSFS。而第三部分\[path]name则允许应用程序独一无二地定义及标识一个邮槽名。其中，“path”代表路径，可指定多级目录。举个例子来说，对一个邮槽进行标识时，下面这些形式的名字都是合法的（注意 Mailslot不得变化，必须原文照输，亦即所谓的“硬编码”）：

\\Oreo\Mailslot\Mymailslot

\\Testserver\Mailslot\Cooldirectory\Funtest\Anothermailslot

\\.\Mailslot\Easymailslot

*\Mailslot\Myslot

服务器字串部分可表示成一个小数点（.）、一个星号（*）、一个域名或者一个真正的服务器名字。所谓“域”，其实就是一系列工作站和服务器的组合，它们共用一个相同的组名。本章稍后，在讲到一个简单的客户机的细节时，还会就邮槽名字作深入探讨。

由于邮槽要依赖Windows文件系统服务在网上来创建和传输数据，所以接口是“与协议无关”的。编制自己的应用程序时，便不必关心基层网络传送协议的细节，不必知道它在一个网络中如何在进程之间建立通信。邮槽通过一个网络与计算机进行远程通信时， Windows文件系统服务需要依赖Windows重定向器，使用“服务器消息块”（SMB）协议，将数据从客户机传给服务器。消息通常是通过“无连接”传输方式来发送的，但亦可强迫 Windows重定向器在Windows NT和Windows 2000上使用“面向连接”的传输方式。至于具体采用哪种方式，要由消息的长度决定。

3.1.2 消息的长度

邮槽通常用“数据报”（Datagram）在网络上传递消息。数据报实际是一些小数据包，只不过要以“无连接”的形式，通过网络进行传输。“无连接”意味着每个数据包在发给接收者之后，不要求对方提供包的收到确认信息。显然，这是一种“不可靠”的数据传输，因为无法保证消息肯定能正确送达。然而，通过无连接传输，我们确实可将消息从一个客户机广播给多个服务器。当然，上述情况也有例外。在 Windows NT和Windows 2000中，假如消息的长度超过424个字节，便会发生这种例外。

在Windows NT和Windows 2000中，若消息长度超过426个字节，便必须在一个SMB会话之上，通过一种“面向连接”的协议进行传输，而不再采用无连接的“数据报”形式。这样一来，在消息较大的情况下，便可保证它们的稳定、高效传输。然而，此时再也不能将一条消息从客户机广播给多个服务器。对于“面向连接”的传输来说，它必然是一种“一对一”通信：一个客户机对一个服务器！在不同的进程之间，使用“面向连接”的传输方式，往往可保证数据传输的可靠性。但在Windows NT和Windows 2000中，邮槽接口却不能保证一条消息肯定能够写入一个邮槽。举个例子来说，假如从客户机向服务器送出一条较大的消息，但指定的服务器在网络中并不存在，那么邮槽接口不会告诉你的客户机应用，数据到服务器的投递已经失败。由于Windows NT和Windows 2000要根据消息的长度，来更改它们的传输方法，所以假定一台机器运行的是Windows NT或Windows 2000，而另一台机器运行的是Windows 95

或者Windows 98，便会面临两种系统在沟通上的困难。

Windows 95和Windows 98只以“数据报”的形式来传送消息，无论消息长度如何。假如Windows 95或Windows 98客户机试图将一条长度超过 424字节的消息传给 Windows NT或Windows 2000服务器，Windows NT和Windows 2000便会只接受头424个字节，剩下的数据会被无情地“砍掉”。换言之，Windows NT和Windows 2000要求更大的消息通过一个“面向连接”的SMB会话进行传输。

消息从Windows NT或Windows 2000客户机传向Windows 95或Windows 98服务器时，也存在着类似的问题。请记住，Windows 95和Windows 98只通过数据报接收数据。由于对超过426个字节的消息来说，Windows NT和Windows 2000不会通过数据报来传输数据，所以在这种情况下，Windows 95和Windows 98不能从这些客户机收到消息。在表3-1中，我们为大家详细总结了这种消息长度上的限制。

注意 Windows CE已从表3-1中剔除出去了，因为它并未提供邮槽编程接口。另外要注意的是，由于Windows NT和Windows 2000重定向器的限制，长度为425或426字节的消息未在此表列出。

表3-1 邮槽消息的长度限制

传输方向	通过数据报进行“无连接”传输	进行“面向连接”的传输
Windows 95或Windows 98 ->Windows 95或Windows 98	消息长度高达64KB	不支持
Windows NT或Windows 2000 ->Windows NT或Windows 2000	消息长度必须为424字节或以下	消息必须大于426个字节
Windows NT或Windows 2000 ->Windows 95或Windows 98	消息长度必须为424字节或以下	不支持
Windows 95或Windows 98 ->Windows NT或Windows 2000	消息长度必须为424字节或以下；多余的字节会被截去	不支持

对Windows NT和Windows 2000这两种网络操作系统来说，还有另一项限制值得我们注意，因为它会对数据报数据的传输产生影响。Windows NT和Windows 2000重定向器不能收发长度正好为 425或426字节的一条数据报消息。例如，假定我们从 Windows NT或Windows 2000客户机向Windows 95、Windows 98、Windows NT或Windows 2000服务器发送这样的一条消息，Windows NT重定向器会在消息发给目标服务器之前，将其长度截短为424字节。

要想保证各种Windows平台之间能够完全正常地通信，强烈建议将消息长度限制在 424字节，或者更短。如果进行面向连接的传输，可考虑使用命名管道，而不是简单的邮槽。命名管道将在第4章讨论。

3.1.3 应用程序的编译

用Microsoft Visual C++编制一个邮槽客户机或服务器应用程序时，必须在程序文件中将Winbase.h这个包容文件包括在内。假如已经包含了 Windows.h（大多数应用程序都会这样），那么亦可将Winbase.h省去。此外，应用程序也要负责建立与 Kernel32.lib的链接，这通常需要用Visual C++链接器标志进行配置。

3.1.4 错误代码

开发邮槽客户机和服务器应用时，所有 Win32 API函数（CreateFile和CreateMailslot除外）在调用失败的情况下，都会返回 0值。CreateFile和CreateMailslot这两个 API却会返回INVALID_HANDLE_VALUE（无效句柄值）。若这些API函数调用失败，应用程序随即应调用GetLastError函数，来接收与此次失败有关的特殊信息。至于所有错误代码的一个完整列表，大家可参考本书附录 C提供的标准Windows错误代码列表，或者直接查询 Winerror.h这个头文件。

3.2 基本客户机／服务器

如前所述，邮槽建立了一个简单的客户机／服务器设计体系。在这个体系中，数据只能从客户机传到服务器，数据通信是单向进行的。服务器进程的职责是创建一个邮槽，而且是能从邮槽读取数据的唯一一个进程。邮槽客户机进程则负责打开邮槽的“ 实例 ”，该进程是能够向其中写入数据的唯一一种进程。

3.2.1 邮槽服务器的详情

若想实现一个邮槽，要求开发一个服务器应用，来负责邮槽的创建。下述步骤解释了如何编写一个基本的服务器应用：

1) 用CreateMailslot API函数创建一个邮槽句柄。

2) 调用ReadFile API函数，并使用现成的邮槽句柄，从任何客户机接收数据。

3) 用CloseHandle这个API函数，关闭邮槽句柄。

可以看出，要开发一个邮槽服务器程序，只需使用极少的 API调用。服务器进程是用CreateMailslot这个API调用来创建邮槽的。定义如下：

```
HANDLE CreateMailslot(
    LPCTSTR lpName,
    DWORD nMaxMessageSize,
    DWORD lReadTimeout,
    LPSECURITY_ATTRIBUTES lpSecurityAttributes
);
```

其中，第一个参数lpName指定邮槽的名字。名字的格式如下：

\\.\Mailslot\[path]name

要注意的是，服务器的名字用一个小数点来表示，亦即服务器就是本地机器。这样做是很有必要的，因为我们不能在远程计算机上创建邮槽。在 lpName参数中，名字必须以一种独一无二的形式表达。可将它设为一个独立的名字，也可以在它前面加上一个完整的目录路径。

nMaxMessageSize参数定义的是可写入邮槽的一条消息的最大长度（以字节为单位）。假如客户机写入的字节数多于nMaxMessageSize的设置，服务器便不会接收这条消息。若将它的值设为0，服务器便会接收任意长度的消息。

在一个邮槽上，读操作可以等待或不等待这两种模式进行，具体由 lReadTimeout参数决定。它以毫秒为单位，指定了读操作需要等候进入消息的时间。若将它的值设为MAILSLOT_WAIT_FOREVER，那么在进入的数据可以读取之前，读操作便会无限期地等待下去。若设为 0，读操作就会立即返回。本章稍后，还会就读操作的详情进行探讨。

lpSecurityAttributes参数决定了为邮槽施加的访问控制权限。在 Windows NT和Windows 2000中，这个参数只实现了一部分，所以同时还应指定一个 null（空） 参数。在邮槽上，唯一能够施加的安全措施是针对本地 I/O进行的 —— 客户机试图将服务器的名字设为小数点（ .），以打开一个邮槽。要想绕过这种安全机制，客户机可指定服务器的实际名字，而不是一个小数点（.），亦即相当于发出一个远程 I/O调用。在Windows NT和Windows 2000中，并未针对远程I/O而实现lpSecurityAttributes参数，因为假如每次发出一条消息时，都在客户机与服务器之间建立一个授权会话，那么效率会显得十分低下。因此，邮槽仅一部分符合标准文件系统采用的Windows NT和Windows 2000安全模型。结果便是，网络中的任何邮槽客户机都可将数据发给服务器。

用一个有效的句柄创建了邮槽之后，便可开始数据的实际读取。服务器是唯一能从邮槽读入数据的进程。服务器应使用 ReadFile这个Win32函数，来进行数据的读取。对 ReadFile的定义如下：

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

CreateMailslot会返回一个句柄hFile。lpBuffer和nNumberOfBytesToRead参数决定了可从邮槽读入多少数据。特别值得注意的是，这个缓冲区的大小应该比来自 CreateMailslot API调用的nMaxMessageSize参数的设置大。此外，缓冲区应该大于邮槽上的进入消息；如果不够大，ReadFile调用便会失败，并返回一个 ERROR_INSUFFICIENT_BUFFER错误。lpNumberOfBytesRead参数用于在ReadFile操作完成后，报告读入的实际字节数量。

利用lpOverlapped参数，我们可以通过异步方式，进行数据的读取。该参数采用的是Win32重叠I/O机制，第4章还会对这个问题进行深入的讲解。在默认情况下， ReadFile操作会处于暂停状态，直到有数据可以读入为止。重叠 I/O只能在Windows NT和Windows 2000上完成；如果使用的操作系统是 Windows 95或Windows 98，那么应将该参数设为 NULL。在程序清单3-1中，我们进一步阐释了如何编写一个简单的邮槽服务器应用。

程序清单3-1 服务器邮槽示例

```
// Server1.cpp

#include <windows.h>
#include <stdio.h>

void main(void)
{
    HANDLE Mailslot;
    char buffer[256];
    DWORD NumberOfBytesRead;

    // Create the mailslot
    if ((Mailslot = CreateMailslot("\\\\.\\Mailslot\\Myslot", 0,
        MAILSLOT_WAIT_FOREVER, NULL)) == INVALID_HANDLE_VALUE)
    {
```

```
        printf("Failed to create a mailslot %d\n", GetLastError());
        return;
    }

    // Read data from the mailslot forever!
    while(ReadFile(Mailslot, buffer, 256, &NumberOfBytesRead,
        NULL) != 0)
    {
        printf("%.*s\n", NumberOfBytesRead, buffer);
    }
}
```

3.2.2 邮槽客户机的详情

要想实现一个客户机，需要开发一个应用程序，对一个现有的邮槽进行引用和写入。下述步骤解释了如何编写一个基本的客户机应用：

1) 使用CreateFile这个API函数，针对想向其传送数据的邮槽，打开指向它的一个引用句柄。

2) 调用WriteFile这个API函数，向邮槽写入数据。

3) 完成了数据的写入后，用CloseHandle这个API函数，关闭打开的邮槽句柄。

如前所述，采用一种“无连接”的形式，邮槽客户机同邮槽服务器通信。客户机打开指向邮槽的一个引用句柄时，实际并不建立同邮槽服务器的一个连接。要想对一个邮槽进行引用，需要使用CreateFile这个API调用。对它的定义如下：

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

lpFileName参数用于描述一个或多个邮槽，我们可用本章早些时候介绍的邮槽命名格式，向其写入数据。在表3-2中，我们更深入地总结了邮槽的命名规范。 dwDesiredAccess参数必须设为GENERIC_WRITE，因为客户机只能向服务器写入数据。 dwShareMode参数必须设为FILE_SHARE_READ，允许服务器在邮槽上打开和进行读操作。 lpSecurityAttributes参数对于邮槽不会有什么效果，应将其设为NULL。dwCreationDisposition标志应设为OPEN_EXISTING。若一台机器既是客户机，也是服务器，这一设置便显得尤其重要——如果服务器没有创建邮槽，对API函数CreateFile的调用便会失败。如果服务器在远程工作，那么。dwCreationDisposition

表3-2 邮槽名字类型

名字格式	说 明
\\.\mailslot\name	标定同一台机器上的一个本地邮槽
\\servername\mailslot\name	标定名为servername的一个远程邮槽服务器
\\domainname\mailslot\name	标定在指定的domain（域）内，使用特定name（名字）的所有邮槽
*\mailslot\name	标定系统主域内，标定特定name（名字）的所有邮槽

参数便没什么意义 dwFlagsAndAttributes参数应定义成 FILE_ATTRIBUTE_NORMAL。hTemplateFile参数应设为NULL。

成功创建一个句柄后，便可开始向邮槽写入数据。请记住，作为客户机，只能将数据写入邮槽。这可以用Win32函数WriteFile来做到，定义如下：

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

其中，hFile参数是由CreateFile返回的一个引用句柄。lpBuffer和nNumberOfBytesToWrite参数决定了有多少字节从客户机发向服务器。一条消息的最大长度为 64KB。如果当初是用一个域或星号（*）格式来创建邮槽句柄，那么在 Windows NT和Windows 2000中，消息的长度限制在424字节之内；而在Windows 95和Windows 98中，限制在64KB之内。如客户机试图发送的消息超出了这一长度限制， WriteFile函数会便失败，而且 GetLastError函数会返回ERROR_BAD_NETPATH错误。之所以会出现这一情况，是由于需要以广播数据报的形式，把消息发送给网络中的所有服务器。lpNumberOfBytesWritten参数返回的是当WriteFile操作完成后，传给服务器的实际字节数量。

通过lpOverlapped参数，我们可采用异步形式，将数据写入一个邮槽。由于邮槽最大的特点便是“无连接”的数据传输，所以 WriteFile函数不会在I/O调用的时候暂停等候。在客户机上，这个参数应设为 NULL。在程序清单3-2中，我们进一步阐述了如何编写一个简单的邮槽客户端应用。

程序清单3-2 客户机邮槽实例

```
// Client.cpp

#include <windows.h>
#include <stdio.h>

void main(int argc, char *argv[])
{
    HANDLE Mailslot;
    DWORD BytesWritten;
    CHAR ServerName[256];

    // Accept a command line argument for the server to send
    // a message to
    if (argc < 2)
    {
        printf("Usage: client <server name>\n");
        return;
    }
        sprintf(ServerName, "\\\\%s\\Mailslot\\Myslot", argv[1]);

        if ((Mailslot = CreateFile(ServerName, GENERIC_WRITE,
            FILE_SHARE_READ, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
            NULL)) == INVALID_HANDLE_VALUE)
        {
```

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

        if (WriteFile(Mailslot, "This is a test", 14, &BytesWritten,
            NULL) == 0)
        {
            printf("WriteFile failed with error %d\n", GetLastError());
            return;
        }

        printf("Wrote %d bytes\n", BytesWritten);

        CloseHandle(Mailslot);
    }
```

3.3 其他邮槽API

对邮槽服务器应用来说，它可使用另外两个 API函数同邮槽打交道： GetMailslotInfo和SetMailslotInfo。其中，一旦邮槽上有消息可以传递， GetMailslotInfo函数便可负责获取消息的长度信息。利用这个函数，程序可针对长度不定的进入消息，动态地调节其缓冲区。GetMailslotInfo函数亦可用来对进入数据进行“ 轮询 ”。对GetMailslotInfo函数的定义如下：

```
BOOL GetMailslotInfo(
    HANDLE hMailslot,
    LPDWORD lpMaxMessageSize,
    LPDWORD lpNextSize,
    LPDWORD lpMessageCount,
    LPDWORD lpReadTimeout
);
```

其中，hMailslot参数指定自CreateMailslot API调用返回的一个邮槽。lpMaxMessageSize参数设置可将多大的一条消息写入邮槽（以字节为单位）。lpNextSize参数则以字节为单位，指出下一条消息的长度。 GetMailslotInfo可能会返回一个MAILSLOT_NO_MESSAGE值，指出在邮槽之上，目前没有等待接收的消息。利用这个参数，服务器便可在邮槽上轮询（不断地查询）是否有进入的消息，防止应用程序在 ReadFile函数调用过程中“ 凝结 ”，傻乎乎地一直等候下去。然而，用这种方式对数据进行轮询并不是一种很好的编程习惯。因为应用程序会连续不停地消耗宝贵的 CPU资源，以检查进入的数据 —— 即使根本没有需要处理的消息。这样一来，便会降低系统的总体性能。如果想防止 ReadFile“ 凝结 ”或暂停执行，建立你使用Win32的重叠I/O。lpMessageCount参数指定一个缓冲区，用于接收等候读入的消息的总量。lpReadTimeout参数则指定了另一个缓冲区，它以毫秒为单位，返回了一次读操作最多能等候多久的时间，让一条消息写入邮槽。

SetMailslotInfo函数用于设置一个邮槽的超时值。超过这个时间，读操作便不再等候进入消息。因此，应用程序完全有能力将读操作从“ 凝结 ”状态转变成“ 非凝结 ”状态；或者相反。对SetMailslotInfo的定义如下：

```
BOOL SetMailslotInfo(
    HANDLE hMailslot,
    DWORD lReadTimeout
);
```

bMailslot参数指定一个自 CreateMailslot API调用返回的邮槽。 lReadTimeout参数以毫秒为单位，指定一次读操作等候一条消息写入邮槽的最长时间。若将其设为 0，在不存在消息的前提下，读操作便会立即返回；若设为 MAILSLOT_WAIT_FOREVER，读操作便会永远等待下去。

3.4 平台和性能问题

在Windows 95（含OSR2）和Windows 98（含第二版）这两种平台上，邮槽存在着一些必须引起我们警惕的限制：“ 8.3字符名字限制 ”、不能取消 “ 凝结 ” 的I/O请求；以及由于超时引起的内存废弃。

3.4.1 8.3字符名字限制

Windows 95和Windows 98不声不响地将邮槽名字限制在8.3字符格式的范围之内。这样一来，有时候就会造成Windows 95/98和Windows NT/2000之间的 “ 不和 ”。举个例子来说，假定我们用\\.\Mailslot\Mymailslot这个名字来创建或打开一个邮槽，Windows 95实际会以\\.\Mailslot\Mymailsl的形式，来创建及引用邮槽。尽管会发生名字被截短的现象， CreateMailslot和CreateFile函数执行仍会成功完成。然而，假如一条消息从 Windows 2000发给Windows 95 —— 或者相反，便无法收到这条消息，因为邮槽的名字并不相符。假如客户机和服务器均在Windows 95机器上运行，则不会出现这种问题 —— 名字在客户机和服务器上都会被截短。要想防范这种互不兼容的问题，一定要将邮槽的名字限制在 8个字符或者更短。

3.4.2 不能取消 “ 凝结 ” 的I/O请求

Windows 95和Windows 98在取消暂停或 “ 凝结 ” 的I/O请求时，也会出现问题。邮槽服务器用ReadFile函数来接收数据。假如用 MAILSLOT_WAIT_POREVER标志创建了一个邮槽，读请求便会一直等待下去，直到有数据可用为止。假如在一个ReadFile请求尚未完成的前提下，服务器应用突然中止运行，应用程序便会被永远地 “ 挂起 ”，或者 “ 凝结 ”。要想取消，唯一的办法就是重新启动 Windows。为解决这个问题，可让服务器在单独一个线程中，打开一个句柄，令其指向自己的邮槽。并发送数据，以终止处于暂停状态的读操作。在程序清单 3-3中，我们详细阐述了这一方案。

程序清单3-3 修订过的邮槽服务器

```
// Server2.cpp

#include <windows.h>
#include <stdio.h>
#include <conio.h>

BOOL StopProcessing;

DWORD WINAPI ServeMailslot(LPVOID lpParameter);
void SendMessageToMailslot(void);

void main(void) {
```

```
    DWORD ThreadId;
    HANDLE MailslotThread;

    StopProcessing = FALSE;
    MailslotThread = CreateThread(NULL, 0, ServeMailslot, NULL,
        0, &ThreadId);

    printf("Press a key to stop the server\n");
    _getch();

    // Mark the StopProcessing flag to TRUE so that when ReadFile
    // breaks, our server thread will end
    StopProcessing = TRUE;

    // Send a message to our mailslot to break the ReadFile call
    // in our server
    SendMessageToMailslot();

    // Wait for our server thread to complete
    if (WaitForSingleObject(MailslotThread, INFINITE) == WAIT_FAILED)
    {
        printf("WaitForSingleObject failed with error %d\n",
            GetLastError());
        return;
    }
}

//
// Function: ServeMailslot
//
// Description:
//     This function is the mailslot server worker function to
//     process all incoming mailslot I/O
//
DWORD WINAPI ServeMailslot(LPVOID lpParameter)
{
    char buffer[2048];
    DWORD NumberOfBytesRead;
    DWORD Ret;
    HANDLE Mailslot;

    if ((Mailslot = CreateMailslot("\\\\.\\mailslot\\myslot", 2048,
        MAILSLOT_WAIT_FOREVER, NULL)) == INVALID_HANDLE_VALUE)
    {
        printf("Failed to create a MailSlot %d\n", GetLastError());
        return 0;
    }

    while((Ret = ReadFile(Mailslot, buffer, 2048,
        &NumberOfBytesRead, NULL)) != 0)
    {
        if (StopProcessing)
            break;

        printf("Received %d bytes\n", NumberOfBytesRead);
```

```
        }

        CloseHandle(Mailslot);

        return 0;
    }

    //
    // Function: SendMessageToMailslot
    //
    // Description:
    //     The SendMessageToMailslot function is designed to send a
    //     simple message to our server so we can break the blocking
    //     ReadFile API call
    //
    void SendMessageToMailslot(void)
    {
        HANDLE Mailslot;
        DWORD BytesWritten;

        if ((Mailslot = CreateFile("\\\\.\\mailslot\\myslot",
            GENERIC_WRITE, FILE_SHARE_READ, NULL, OPEN_EXISTING,
            FILE_ATTRIBUTE_NORMAL, NULL)) == INVALID_HANDLE_VALUE)
        {
            printf("CreateFile failed with error %d\n", GetLastError());
            return;
        }

        if (WriteFile(Mailslot, "STOP", 4, &BytesWritten, NULL) == 0)
        {
            printf("WriteFile failed with error %d\n", GetLastError());
            return;
        }

        CloseHandle(Mailslot);
    }
```

3.4.3 超时引起的内存废弃

对于Windows 95和Windows 98来说，值得注意的最后一个问题便是内存空间的“废弃”，或者说内存空间产生了“漏洞”。在邮槽上使用超时设定时，便有可能出现这一现象。假如用CreateMailslot函数创建一个邮槽，同时将超时时间设为大于0的一个值，那么一旦超过这一时间，ReadFile函数便会造成内存空间的废弃，生成所谓的“内存漏洞”。同时，函数会返回FALSE。经过多次ReadFile函数调用之后，系统就会变得极不稳定，以后超时的ReadFile调用会开始返回TRUE。因此，系统不能再执行其他MS-DOS程序。为解决这个问题，请将超时值设为0或者MAILSLOT_WAIT_FOREVER。这样便可禁止应用程序使用超时机制，从而避免了实际产生的内存“漏洞”。

在微软知识库中，描述了下述问题，以及它们存在的限制。这个知识库是完全公开的，网上地址在http: //support.microsoft.com/support/search。这里只对相关的主题进行简单介绍：

- Q139715：ReadFile为邮槽返回错误的“错误代码”

假如服务器用CreateMailslot打开一个邮槽，指定一个超时，然后用 ReadFile接收数据，那么假如没有数据可用（可以读取），ReadFile便会失败。此时用GetLastError函数会返回一个错误代码5（拒绝访问）。

■ Q192276：GetMailslotInfo返回不正确 lpNextSize值

如果在Windows 95 OEM Service Release 2（OSR2）或Windows 98中调用API函数GetMailslotInfo，同时没有安装一个网络客户机组件，那么对 lpNextSize参数来说，便会通过它接收到一个不确切的值（通常有百万之大），或者接收到一个负数。如再次调用该函数，则往往会恢复正常，返回正确的值。

■ Q170581：在Win95中创建的邮槽只允许4093个字节的消息

如调用WriteFile这个API函数，将多于4093个字节的数据写入已在Windows 95工作站上创建好的一个邮槽，操作便会失败。

■ Q131493：CreateFile和邮槽的问题

在API函数CreateFile的用户文档说明中，错误描述了在打开一个邮槽的客户机那一端时，CreateFile可能返回的值。

3.5 小结

本章讲解了邮槽（Mailslot）网络编程技术。利用这一技术，应用程序可以在 Windows重定向器的帮助下，实现简单的单向进程间数据通信。对邮槽来说，它最有价值的一项功能便是通过网络，将一条消息广播给一台或多台计算机。然而，邮槽并未提供对数据可靠传输的保障。假如希望用 Windows重定向器实现“ 可靠 ”的数据通信，请考虑使用命名管道，这是下一章的主题！

第4章 命 名 管 道

“ 命名管道 ”或“ 命名管线 ”（Named Pipes）是一种简单的进程间通信（ IPC）机制，Microsoft Windows NT，Windows 2000、Windows 95以及Windows 98均提供了对它的支持（但不包括Windows CE）。命名管道可在同一台计算机的不同进程之间，或在跨越一个网络的不同计算机的不同进程之间，支持可靠的、单向或双向的数据通信。用命名管道来设计应用程序实际非常简单，并不需要事先深入掌握基层网络传送协议（如 TCP/IP或IPX）的知识。这是由于命名管道利用了微软网络提供者（ MSNP）重定向器，通过一个网络，在各进程间建立通信。这样一来，应用程序便不必关心网络协议的细节。之所以要用命名管道作为自己的网络通信方案，一项重要的原因是它们充分利用了 Windows NT及Windows 2000内建的安全特性。

这里有一个可采纳命令管道的例子。假定我们要开发一个数据管理系统，只允许一个指定的用户组进行操作。想像在自己的办公室中，有一部计算机，其中保存着公司的秘密。我们要求只有公司的管理人员，才能访问及处理这些秘密。假定在自己的工作站机器上，公司内的每名员工都可看到网络上的这台计算机。然而，我们并不希望普通员工取得对机密材料的访问权。在这种情况下，命名管道可发挥出很好的作用，因为我们可开发一个服务器应用程序，令其以来自客户机的请求为准，对公司的秘密进行安全操作。服务器可将客户访问限制在管理人员身上，用Windows NT或新版Windows 2000自带的安全机制，便可非常轻松地做到这一点。

在此要记住的一个重点是，将命名管道作为一种网络编程方案使用时，它实际上建立一个简单的客户机／服务器数据通信体系，可在其中可靠地传输数据。本章将介绍如何来开发一个命名管道客户机及服务器应用。首先要解释的是命名管道的命名规范（约定），然后介绍基本的管道类型。随后，将向大家展示如何实现一个简单的服务器应用。然后以它为基础，深入探讨高级的服务器编程技术。接下来，讲解如何开发一个简单的客户机应用。到本章末，我们会对命名管道已知的所有问题及限制进行总结。

4.1 命名管道的实施细节

命令管道是围绕Windows文件系统设计的一种机制，采用“ 命名管道文件系统 ”（Named Pipe File System, NPFS）接口。因此，客户机和服务器应用可利用标准的 Win32文件系统API函数（如ReadFile和WriteFile）来进行数据的收发。通过这些 API函数，应用程序便可直接利用Win32文件系统命名规范，以及 Windows NT/Windows 2000文件系统的安全机制。 NPFS依赖于MSNP重定向器在网上进行命名管道数据的发送和接收。这样一来，便可实现接口的“ 与协议无关 ”特性：若在自己开发的应用程序中使用命名管道在网上不同的进程间建立通信，程序员不必关心基层网络传送协议（如 TCP和IPX等等）的细节。对 NPFS来说，命名管道是用“ 通用命名规范 ”（UNC）来标识的。在第 2章，我们已比较深入地探讨了 UNC、Windows重定向器以及安全机制。

4.1.1 命名管道命名规范

命名管道的标识是采用UNC格式进行的：

\\server\Pipe\[path]name

上述字串可分为三部分来观看：\\server、\Pipe和\[path]name。第一部分\\server指定一个服务器的名字。命名管道便是在那个服务器上创建的，而且要由它对进入的连接请求进行“监听”。第二部分\Pipe是一个不可变化的“硬编码”字串（必须原样照录，但不用区分大小写），用于指出该文件从属于NPFS。而第三部分\[path]name则使应用程序可以“唯一”定义及标定一个命名管道的名字，而且可在这里设置多级目录。举个例子来说，下述字串均是合法的命名管道名字：

\\myserver\PIPE\mypipe

\\Testserver\pipe\cooldirectory\funtest\jim

\\.\Pipe\Easynamedpipe

注意就服务器字串这一部分来说（第一部分），既可表达为一个小数点（.），亦可表达为一个实际的服务器名字。

4.1.2 字节模式及消息模式

命名管道提供了两种基本通信模式：字节模式和消息模式。在字节模式中，消息以一个连续的字节流的形式，在客户机与服务器之间流动。这意味着，对客户机应用和服务器应用来说，在任何一个特定的时间段内，它们不能准确知道有多少字节从管道中读入或者写入管道。因此，在一方写入某个数量的字节，并不表示在另一方会读出等量的字节。这样一来，客户机和服务器在传输数据的时候，便不必关心数据的内容。而在消息模式中，客户机和服务器则通过一系列不连续的数据单位，进行数据的收发。每次在管道上发出了一条消息后，它必须作为一条完整的消息读入。在图4-1中，我们对这两种管道模式进行了总结。

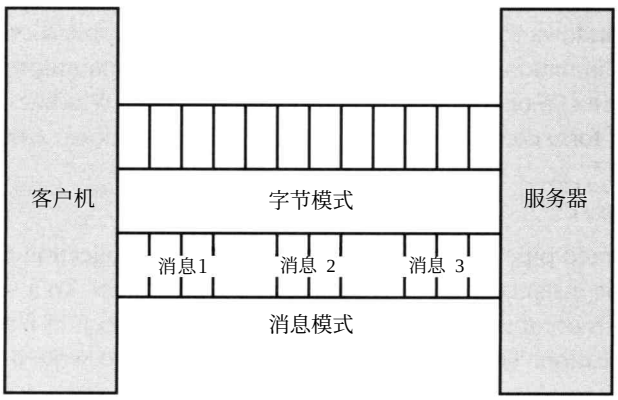

图4-1 字节模式和消息模式

4.1.3 应用程序的编译

用Microsoft Visual C++构建一个命名管道或服务器应用程序时，必须在自己的程序文件中加入Winbase.h这个头文件。但假如你的应用程序已经包括了 Windows.h —— 大多数情况下

都会如此，便可将Winbase.h忽略。此外，你的程序还需负责建立与 Kernel32.lib的链接关系，后者通常是用Visual C++的链接器标志来配置的。

4.1.4 错误代码

开发命名管道客户机和服务器应用时，所有Win32 API函数（CreateFile和CreateNamedPipe除外）都会在调用失败的前提下返回 0值。CreateFile和CreateNamedPipe返回的则是INVALID_HANDLE_VALUE。若这两个函数中任何一个调用失败，应用程序会调用GetLastError函数，取得与这一次失败有关的特定信息。在 Winerror.h这个头文件中，以及在本书附录C中，均提供了完整的错误代码清单。

4.2 客户机与服务器的基础

命名管道最大的特点便是建立一个简单的客户机／服务器程序设计体系。在这个体系结构中，在客户机与服务器之间，数据既可单向传递，亦可双向流动。这一点相当重要，因为我们可以自由地收发数据，无论应用程序是一个客户机还是一个服务器。对命名管道服务器和客户机来说，两者最大的区别在于：服务器是唯一一个有权创建命名管道的进程，也只有它才能接受管道客户机的连接请求。对一个客户机应用来说，它只能同一个现成的命名管道服务器建立连接。在客户机应用和服务器应用之间，一旦建好连接，两个进程都能对标准的Win32函数，在管道上进行数据的读取与写入。这些函数包括 ReadFile和WriteFile等等。要注意的是，命名管道服务器应用只能在 Windows NT或Windows 2000上工作——Windows 95和Windows 98不允许应用程序创建命名管道！正是由于存在这一限制，我们无法在两台Windows 95或Windows 98计算机之间直接建立通信。然而， Windows 95和Windows 98客户机可建立与Windows NT及Windows 2000计算机的正常连接。

4.2.1 服务器的细节

要想实现一个命名管道服务器，要求必须开发一个应用程序，通过它创建命名管道的一个或多个“实例”，再由客户机进行访问。对服务器来说，管道实例实际就是一个句柄，用于从本地或远程客户机应用接受一个连接请求。按下述步骤行事，便可写出一个最基本的服务器应用：

1) 使用API函数CreateNamedPipe，创建一个命名管道实例句柄。

2) 使用API函数ConnectNamedPipe，在命名管道实例上监听客户机连接请求。

3) 分别使用ReadFile和WriteFile这两个API函数，从客户机接收数据，或将数据发给客户机。

4) 使用API函数DisconnectNamedPipe，关闭命名管道连接。

5) 使用API函数CloseHandle，关闭命名管道实例句柄。

首先，我们的服务器进程需要使用 CreateNamedPipe这个API调用，创建一个命名管道实例。该调用的定义如下：

```
HANDLE CreateNamedPipe(
    LPCTSTR lpName,
    DWORD dwOpenMode,
    DWORD dwPipeMode,
```

```
    DWORD nMaxInstances,
    DWORD nOutBufferSize,
    DWORD nInBufferSize,
    DWORD nDefaultTimeOut,
    LPSECURITY_ATTRIBUTES lpSecurityAttributes
);
```

第一个参数是lpName，用于指定一个命名管道的名字。这个名字采用遵守下述 UNC格式：

```
\\.\Pipe\[path]name
```

注意服务器的名字在此是用一个小数点表示的。换言之，它对应于本地机器（本机）。注意不可在一台远程计算机上创建命名管道。参数的 [path]name部分必须指定一个独一无二的名字。它可以是一个单独的文件名，亦可在文件名前面加上完整的目录路径。

dwOpenMode参数用于指示一个管道创建好之后，它的传输方向、 I/O控制以及安全模式。在表4-1中，我们总结了可以选用的所有标志设定。创建一个管道时，需要将这些标志通过OR（或）运算组合到一起。

其中， PIPE_ACCESS_标志决定了在客户机与服务器之间，数据在管道上的流动方向。可用PIPE_ACCESS_DUPLEX标志以双向传输方式打开一个管道。也就是说，在客户机与服务器之间，数据可以双向传输。除此以外，亦可使用 PIPE_ACCESS_INBOUND或者PIPE_ACCESS_OUTBOUND标志， 以单向传输方式打开一个管道。也就是说，数据只能从客户机传向服务器，或从服务器传向客户机。图 4-2向大家进一步阐述了标志组合的情况，指出数据在客户机与服务器之间如何流动。

表4-1 命名管道打开模式标志

打开模式	标 志	说 明
双向	PIPE_ACCESS_DUPLEX	双向式管道：服务器和客户机进程都能在管道上读写数据
	PIPE_ACCESS_OUTBOUND	数据在管道中只能从服务器朝客户机流动
	PIPE_ACCESS_INBOUND	数据在管道中只能从客户机朝服务器流动
I/O控制	FILE_FLAG_WRITE_THROUGH	只适用于字节模式下的管道。对那些用来向命名管道写入数据的函数来说，除非写入的数据通过网络传送出去，而且进入远程计算机的管道缓冲区内，否则不会返回
	FILE_FLAG_OVERLAPPED	允许执行读、写和连接操作的函数使用重叠式 I/O
安全模式	WRITE_DAC	应用程序可对命名管道的DACL进行写操作
	ACCESS_SYSTEM_SECURITY	应用程序可对命名管道的SACL进行写操作
	WRITE_OWNER	应用程序可对命名管道的“所有人”及“组” SID,进行写操作

接下去的一系列dwOpenMode标志用于从服务器的角度出发，对一个命名管道的 I/O行为加以控制。FILE_FLAG_WRITE_THROUGH标志控制着写操作，除非写入的数据通过网络实际传送出去，而且进入远程计算机的管道缓冲区，否则负责数据写入的函数不会返回。要注意的是，该标志只对字节模式下的命名管道有用。此时，客户机和服务器分别位于不同的计算机上。FILE_FLAG_OVERLAPPED标志则允许执行读、写和连接操作的函数立即返回，即使那个函数可能会花较长的时间来完成操作。本章稍后设计一个高级的服务器程序时，还会对这种重叠式的I/O进行深入探讨。

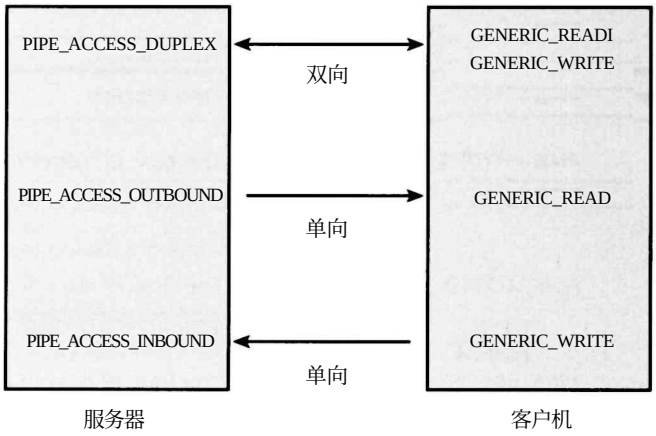

图4-2 模式标志以及数据流向

在表4-1中，最后一组dwOpenMode标志负责控制服务器访问安全描述符的能力，那些安全描述符是由命名管道创建的。一个管道建好之后，假如应用程序需要修改或更新管道的安全描述符，便应将这些标志相应地设为“允许访问”。其中，WRITE_DAC标志使我们的程序能够更新管道的授权访问控制列表（DACL）；而ACCESS_SYSTEM_SECURITY允许访问管道的系统访问控制列表（SACL）；最后，WRITE_OWNER标志允许我们更改管道的所有人及组安全ID（SID）。例如，对于已拥有管道访问权的一名用户，假定我们现在打算拒绝他的访问，便可使用安全API函数，修改管道的DACL。在本书第2章，我们已详细讨论了DACL、SACL以及SID。

在CreateNamedPipe这个API调用中，它的dwPipeMode参数指定了一个管道的读、写以及等待模式。在表4-2中，我们对可以选用的所有模式标志进行了总结。实际使用时，需要从每种模式类别中拿出一个标志，再将它们通过OR（或）运算合并到一起。假如当初使用PIPE_READMODE_BYTE | PIPE_TYPE_BYTE模式标志，以“面向字节”的形式来打开一个管道，那么数据只能以“字节流”的形式，在管道上进行读取和写入。也就是说，在管道上进行数据的读写时，不必保证读和写操作一一对应关系——因为数据并不包含任何消息边界。举个例子来说，假设一个发送者向管道写入500个字节，那么作为接收者，或许希望一次只读入100字节，直到收到所有数据为止。要想为消息建立明确的边界，对各条消息加以区分，便需事先将管道置为“面向消息”的模式，这是用PIPE_READMODE_MESSAGE | PIPE_TYPE_MESSAGE标志来做到的。处在这种模式下，每一次读必须有一次写与之对应！例如，假定发送者向管道写入 500字节的一条消息，那么在接收数据时，接收者必须为ReadFile函数提供一个500字节甚或更大的缓冲区。假如接收者没有这样做， ReadFile函数调用便会失败，并返回ERROR_MORE_DATA错误。亦可将PIPE_TYPE_MESSAGE和PIPE_READMORE_BYTE组合使用，允许发送者以消息形式向管道写入数据，同时让接收者一次可以读取任意数量的字节。在数据流中，消息的定界符会被忽略。然而，千万不可混合使用 PIPE_TYPE_BYTE标志以及PIPE_READMORE_MESSAGE标志。如果这样做，会造成 CreateNamedPipe函数调用失败，并返回一个 ERROR_INVALID_PARAMETER（参数无效）错误。这是由于数据以字节形式写入管道时， I/O流中不存在消息定界符的缘故。 PIPE_WAIT或PIPE_NOWAIT标志亦可同读与写模式标志组合使用。其中，PIPE_WAIT标志用于将管道置为“锁定”或“等待”模式，而 PIPE_NOWAIT标志将管道置为“非锁定”或“不等待”模式。在锁定模式中，像 ReadFile这样的I/O（输入／输出）操作会暂

停，直至I/O请求完成。假如不指定任何标志，这也是一种默认的行为。而使用非锁定模式标志（PIPE_NOWAIT），I/O操作无论如何都会立即返回。然而，在Win32应用程序中，不可用它来实现异步形式的I/O。之所以提供了这个标志，只是为了保证与早期Microsoft LAN Manager 2.0应用的向后兼容。使用ReadFile和WriteFile函数，应用程序可通过Win32重叠式I/O，来实现异步I/O。本章后面还会对此详述。

表4-2 命名管道的读写模式标志

模 式	标 志	说 明
写	PIPE_TYPE_BYTE	数据以字节流的形式，写入管道
	PIPE_TYPE_MESSAGE	数据以消息流的形式，写入管道
读	PIPE_READMODE_BYTE	数据以字节流的形式，从管道中读入
	PIPE_READMODE_MESSAGE	数据以消息流的形式，从管道中读入
等待	PIPE_WAIT	允许“锁定”模式
	PIPE_NOWAIT	允许“非锁定”模式

注意 PIPE_NOWAIT标志现已废弃不用，务必不要在Win32环境中用它来实现异步I/O。它之所以仍在本书出现，只是为了保证与早期的Microsoft LAN Manager 2.0软件的“向后兼容”。

nMaxInstances参数指定对一个命名管道来说，最多可创建多少个实例或管道句柄。所谓管道的“实例”，其实就是从本地或远程客户机应用到创建那个命名管道的服务器应用程序的一个连接。在此可接受的取值范围在1到PIPE_UNLIMITED_INSTANCES（无限实例）之间。例如，假定我们想设计一个服务器应用，令其一次最多只为五个客户连接提供服务，那么该参数便应设为5。假如将该参数设为PIPE_UNLIMITED_INSTANCES，管道实例的数量便基本上是“无限”的，当然，仍然要受可用的系统资源的限制。

CreateNamedPipe函数的nOutBufferSize和nInBufferSize参数分别指定了为内部输入及输出缓冲区长度保留的字节数量。每次创建一个命名管道实例时，都会动态决定这些长度。系统会使用未分页的内存池（由操作系统使用的物理内存）来设置“进入”以及（或者）外出”缓冲区。指定的缓冲区长度应当合理。首先不能过大，不至于将未分页的内存池都用光了。但另一方面，也不应过小，造成没有足够的空间来满足标准 I/O请求的需要。假如应用程序试图写入的数据量大于指定的缓冲区长度，系统就会自己试着扩充缓冲区，用未分页的内存池来装下数据。在实际使用中，应用程序应将这些内部缓冲区的长度设为一个合适的值，使之与调用ReadFile和WriteFile时的发送／接收缓冲区大小相符。

nDefaultTimeOut参数用于指定默认的超时时间（客户机等待同一个命令管道建立连接的最长时间），以毫秒为单位。注意这一设置只对特定的客户机应用才会产生影响——该应用通过WaitNamePipe函数，判断在什么时候，可用一个命名管道的实例来接受连接。本章稍后，在开发一个命名管道客户机应用时，还会更深入地讲述这一概念。

lpSecurityAttributes参数允许应用程序为命名管道指定一个安全描述符，并决定一个子进程是否能够继承新建的句柄。假如将该参数设为 NULL（空），命名管道便会获得一个默认的安全描述符，同时句柄不可继承。默认的安全描述符允许命名管道拥有与创建它的进程相同的安全限制以及访问控制——与第2章讲述的 Windows NT及Windows 2000安全模型相符。使用某些安全API函数，我们可在一个SECURITY_DESCRIPTOR结构中为特定的用户及用户组设

置访问权限，从而向一个管道施加访问控制的限制。假如服务器希望打开对任何客户机的访问权限，那么应该为SECURITY_DESCRIPTOR结构分配一个空的授权访问控制列表（DACL）。

从CreateNamedPipe调用成功接收到一个句柄之后（亦即一个管道实例），便必须等待来自一个命名管道客户机的连接。可使用 ConnectNamedPipe这个API函数，来建立这个连接。该函数的定义如下：

```
BOOL ConnectNamedPipe(
    HANDLE hNamedPipe,
    LPOVERLAPPED lpOverlapped
);
```

其中，hNamedPipe参数指定自 CreateNamedPipe调用返回的一个有效管道实例句柄。lpOverlapped参数则使这个API函数能以异步方式工作，或者说能将它置为“非锁定”模式，前提是当初使用FILE_FLAG_OVERLAPPED标志来创建管道。这正是大家所熟知的 Win32重叠式I/O。假如将该参数设为NULL，ConnectNamedPipe便会暂时进入“锁定”或“等候”状态，直到客户机建立了同服务器的连接。至于重叠式 I/O进一步的情况，还会在本章后面讲到如何创建一个更高级的命名管道服务器时讨论。

一个命名管道客户机成功建立了与服务器的连接之后，ConnectNamedPipe这个API调用便会结束。随后，服务器可用WriteFile函数，向客户机自由地发送数据；或者使用ReadFile函数，从客户机那里接收数据。服务器完成了与一个客户机的通信之后，便应调用DisconnctNamedPipe函数，以关闭此次通信会话。在程序清单 4-1中，我们向大家展示了如何编写一个简单的服务器应用，令其与客户机通信。

程序清单4-1 简单的命名管道服务器

```
// Server.cpp

#include <windows.h>
#include <stdio.h>

void main(void)
{
    HANDLE PipeHandle;
    DWORD BytesRead;
    CHAR buffer[256];
if ((PipeHandle = CreateNamedPipe("\\\\.\\Pipe\\Jim",
    PIPE_ACCESS_DUPLEX, PIPE_TYPE_BYTE | PIPE_READMODE_BYTE, 1,e
    0, 0, 1000, NULL)) == INVALID_HANDLE_VALUE)
{
    printf("CreateNamedPipe failed with error %d\n",
        GetLastError());
    return;
}

printf("Server is now running\n");

if (ConnectNamedPipe(PipeHandle, NULL) == 0)
      {
          printf("ConnectNamedPipe failed with error %d\n",
              GetLastError());
          CloseHandle(PipeHandle);
          return;
```

```
        }

        if (ReadFile(PipeHandle, buffer, sizeof(buffer),
            &BytesRead,  NULL) <= 0)
        {
            printf("ReadFile failed with error %d\n", GetLastError());
            CloseHandle(PipeHandle);
            return;
        }

        printf("%.*s\n", BytesRead, buffer);

        if (DisconnectNamedPipe(PipeHandle) == 0)
        {
            printf("DisconnectNamedPipe failed with error %d\n",
                GetLastError());
            return;
        }

        CloseHandle(PipeHandle);
    }
```

如何构建空的授权访问控制列表（NULL DACL）

在Windows NT或Windows 2000操作系统中，应用程序使用Win32 API函数创建诸如文件及命名管道这样的、要求安全保护的对象时，操作系统会赋予应用程序设置访问控制列表的权力，这是通过指定一个SECURITY_ATTRIBUTES结构来进行的。该结构的定义如下：

```
typedef struct _SECURITY_ATTRIBUTES {
    DWORD   nLength;
    LPVOID  lpSecurityDescriptor;
    BOOL    bInheritHandle
} SECURITY_ATTRIBUTES;
```

其中，lpSecurityDescriptor字段用于在一个SECURITY_DESCRIPTOR（安全描述符）结构中，为一个对象设定访问权限。在 SECURITY_DESCRIPTOR结构中，包含了一个DACL字段，它定义了哪些用户和用户组有权访问对象。假如将该字段设为 NULL，那么任何用户及用户组均能访问我们的资源。

要注意的是，应用程序不能直接访问一个 SECURITY_DESCRIPTOR结构，必须通过相关的Win32安全API函数来进行。如果想为SECURITY_DESCRIPTOR结构分配一个空的DACL，便需采取下述操作：

1) 创建并初始化一个SECURITY_DESCRIPTOR结构，这是用API函数InitializeSecurity Descriptor来进行的。

2) 为SECURITY_DESCRIPTOR结构分配一个空的DACL，这是用API函数SetSecurityDescriptorDacl来进行的。

成功建立一个新的 SECURITY_DESCRIPTOR结构后，必须将其分配给一个SECURITY_ATTRIBUTES结构。到这时为止，我们便已作好了准备，可开始调用像CreateNamedPipe这样的Win32函数，同时使用新建的SECURITY_ATTRIBUTES结构，其中包含了一个空的DACL。下述代码片断向大家展示了如何调用必要的安全API函数，来做到这一点：

```
    // Create new SECURITY_ATTRIBUTES and SECURITY_DESCRIPTOR
    // structure objects
    SECURITY_ATTRIBUTES sa;
    SECURITY_DESCRIPTOR sd;

    // Initialize the new SECURITY_DESCRIPTOR object to empty values
    if (InitializeSecurityDescriptor(&sd, SECURITY_DESCRIPTOR_REVISION)
        == 0)
    {
        printf("InitializeSecurityDescriptor failed with error %d\n",
            GetLastError());
        return;
    }

    // Set the DACL field in the SECURITY_DESCRIPTOR object to NULL
    if (SetSecurityDescriptorDacl(&sd, TRUE, NULL, FALSE) == 0)
    {
        printf("SetSecurityDescriptorDacl failed with error %d\n",
            GetLastError());
        return;
    }

    // Assign the new SECURITY_DESCRIPTOR object to the
    // SECURITY_ATTRIBUTES object
    sa.nLength = sizeof(SECURITY_ATTRIBUTES);
    sa.lpSecurityDescriptor = &sd;
    sa.bInheritHandle = TRUE;
```

4.2.2 高级服务器的细节

在前面的程序清单4-1中，我们展示了如何设计一个命名管道服务器应用，令其只负责对一个管道实例的控制。所有API调用都采用同步模式工作。在这种模式下，每个调用都会一直等到I/O请求完成，才会返回。命名管道服务器也能拥有多个管道实例，所以客户机能够建立同服务器的两个或更多的连接；管道实例的数量要受到 CreateNamedPipe这个API调用之nMaxInstances参数指定的数字的限制。要想同时控制不止一个的管道实例，服务器必须考虑使用多个线程，或者使用异步Win32 I/O机制（比如重叠式I/O以及完成端口等），分别为每个管道实例提供服务。采用异步I/O机制，服务器可从单独一个应用程序线程中，同时为所有管道实例提供服务。在此，我们将解释如何使用线程以及重叠式 I/O，来开发更高级的服务器应用。要想了解如何将“完成端口”应用于Windows套接字，请参考本书第8章。

1. 线程

要想开发一个高级服务器，令其使用线程，同时支持多个管道实例，整个过程是非常简单的。我们要做的唯一事情便是为每个管道实例都创建一个线程，并用前面（介绍简单服务器的开发时）讨论过的技术，为每个实例提供服务。在程序清单 4-2中，我们阐述了如何让一个服务器应用同时为五个管道实例提供服务。这个应用程序实际也是一个“反射”或“回应”（Echo）服务器，可从客户机读取数据，并将数据原封不动地回送给客户机。

程序清单4-2 在Win32环境中使用线程技术开发的高级命名管道服务器

```
// Threads.cpp

#include <windows.h>
```

```
#include <stdio.h>
#include <conio.h>

#define NUM_PIPES 5

DWORD WINAPI PipeInstanceProc(LPVOID lpParameter);

void main(void)
{
    HANDLE ThreadHandle;
    INT i;
    DWORD ThreadId;

    for(i = 0; i < NUM_PIPES; i++)
    {
        // Create a thread to serve each pipe instance
        if ((ThreadHandle = CreateThread(NULL, 0, PipeInstanceProc,
            NULL, 0, &ThreadId)) == NULL)
        {
            printf("CreateThread failed with error %\n",
                GetLastError());
            return;
        }
        CloseHandle(ThreadHandle);
    }

    printf("Press a key to stop the server\n");
    _getch();
}

//
// Function: PipeInstanceProc
//
// Description:
//     This function handles the communication details of a single
//     named pipe instance
//
DWORD WINAPI PipeInstanceProc(LPVOID lpParameter)
{
    HANDLE PipeHandle;
    DWORD BytesRead;
    DWORD BytesWritten;
    CHAR Buffer[256];

    if ((PipeHandle = CreateNamedPipe("\\\\.\\PIPE\\jim",
        PIPE_ACCESS_DUPLEX, PIPE_TYPE_BYTE | PIPE_READMODE_BYTE,
        NUM_PIPES, 0, 0, 1000, NULL)) == INVALID_HANDLE_VALUE)
    {
        printf("CreateNamedPipe failed with error %d\n",
            GetLastError());
        return 0;
    }

    // Serve client connections forever
    while(1)
    {
```

```
        if (ConnectNamedPipe(PipeHandle, NULL) == 0)
        {
            printf("ConnectNamedPipe failed with error %d\n",
                GetLastError());
            break;
        }

        // Read data from and echo data to the client until
        // the client is ready to stop
        while(ReadFile(PipeHandle, Buffer, sizeof(Buffer),
            &BytesRead,  NULL) > 0)
        {
            printf("Echo %d bytes to client\n", BytesRead);

            if (WriteFile(PipeHandle, Buffer, BytesRead,
                &BytesWritten, NULL) == 0)
            {
                printf("WriteFile failed with error %d\n",
                    GetLastError());
                break;
            }
        }

        if (DisconnectNamedPipe(PipeHandle) == 0)
        {
            printf("DisconnectNamedPipe failed with error %d\n",
                GetLastError());
            break;
        }
    }

    CloseHandle(PipeHandle);
    return 0;
}
```

要想使自己的服务器应用能够同时控制五个管道实例，首先要调用 API函数CreateThread（创建线程）。可令CreateThread同时启动五个执行线程，每个都负责执行一个 PipeInstanceProc函数（五个同时执行）。PipeInstanceProc函数的工作原理和程序清单4-1展示的那个基本服务器应用大致相同，只是它会调用DisconnectNamedPipe这个API函数，来重复使用一个命名管道句柄。该函数会关闭客户机同服务器的会话。应用程序调用了 DisconnectNamedPipe后，便可腾出手来，使用同样的管道实例句柄，调用ConnectNamePipe函数，为另一个客户提供服务。

2. 重叠式I/O

重叠式I/O是一种特殊的输入／输出机制，允许Win32 API函数（如ReadFile和WriteFile）在发出I/O请求之后，以异步方式工作。具体的工作原理是：向这些API函数传递一个OVERLAPPED（重叠式）结构，然后使用 API函数GetOverlappedResult，从原来那个OVERLAPPED结构中，取得一次I/O请求的结果。如果在使用重叠式结构的前提下，调用一个 Win32 API函数，那么调用无论如何都会立即返回！

要想通过重叠I/O机制， 开发一个高级的命名服务器，令其同时负责对多个命名管道实例的管理，首先需要调用 CreateNamedPipe函数，同时将它的 nMaxInstances参数设为大于1的一个值。此外，还必须将dwOpenMode标志设为FILE_FLAG_OVERLAPPED。在程序清单4-3中，

我们展示了具体如何开发这样的一个高级命名管道服务器。注意该应用实际是一个“反射”或“回应”服务器，用于从客户机读取数据，并将其原封不动地打回。

程序清单4-3 使用Win32重叠式I/O技术开发的高级命名管道服务器

```
// Overlap.cpp

#include <windows.h>
#include <stdio.h>

#define NUM_PIPES 5
#define BUFFER_SIZE 256

void main(void)
{
    HANDLE PipeHandles[NUM_PIPES];
    DWORD BytesTransferred;
    CHAR Buffer[NUM_PIPES][BUFFER_SIZE];
    INT i;
    OVERLAPPED Ovlap[NUM_PIPES];
    HANDLE Event[NUM_PIPES];

    // For each pipe handle instance, the code must maintain the
    // pipes' current state, which determines if a ReadFile or
    // WriteFile is posted on the named pipe. This is done using
    // the DataRead variable array. By knowing each pipe's
    // current state, the code can determine what the next I/O
    // operation should be.
    BOOL DataRead[NUM_PIPES];

    DWORD Ret;
    DWORD Pipe;

    for(i = 0; i < NUM_PIPES; i++)
    {
        // Create a named pipe instance
        if ((PipeHandles[i] = CreateNamedPipe("\\\\.\\PIPE\\jim",
            PIPE_ACCESS_DUPLEX | FILE_FLAG_OVERLAPPED,
            PIPE_TYPE_BYTE | PIPE_READMODE_BYTE, NUM_PIPES,
            0, 0, 1000, NULL)) == INVALID_HANDLE_VALUE)
        {
            printf("CreateNamedPipe for pipe %d failed "
                "with error %d\n", i, GetLastError());
            return;
        }

        // Create an event handle for each pipe instance. This
        // will be used to monitor overlapped I/O activity on
        // each pipe.
        if ((Event[i] = CreateEvent(NULL, TRUE, FALSE, NULL))
            == NULL)
        {
            printf("CreateEvent for pipe %d failed with error %d\n",
                i, GetLastError());
            continue;
```

```
        }

        // Maintain a state flag for each pipe to determine when data
        // is to be read from or written to the pipe
        DataRead[i] = FALSE;

        ZeroMemory(&Ovlap[i], sizeof(OVERLAPPED));
        Ovlap[i].hEvent = Event[i];

        // Listen for client connections using ConnectNamedPipe()
        if (ConnectNamedPipe(PipeHandles[i], &Ovlap[i]) == 0)
        {
            if (GetLastError() != ERROR_IO_PENDING)
            {
                printf("ConnectNamedPipe for pipe %d failed with"
                    " error %d\n", i, GetLastError());
                CloseHandle(PipeHandles[i]);
             return;
         }
    }
}

printf("Server is now running\n");


// Read and echo data back to Named Pipe clients forever
while(1)
{
    if ((Ret = WaitForMultipleObjects(NUM_PIPES, Event,
        FALSE, INFINITE)) == WAIT_FAILED)
    {
        printf("WaitForMultipleObjects failed with error %d\n",
            GetLastError());
        return;
    }

    Pipe = Ret - WAIT_OBJECT_0;

    ResetEvent(Event[Pipe]);

    // Check overlapped results, and if they fail, reestablish
    // communication for a new client; otherwise, process read
    // and write operations with the client

    if (GetOverlappedResult(PipeHandles[Pipe], &Ovlap[Pipe],
        &BytesTransferred, TRUE) == 0)
    {
        printf("GetOverlapped result failed %d start over\n",
            GetLastError());

        if (DisconnectNamedPipe(PipeHandles[Pipe]) == 0)
        {
            printf("DisconnectNamedPipe failed with error %d\n",
                GetLastError());
```

```
            return;
        }

        if (ConnectNamedPipe(PipeHandles[Pipe],
            &Ovlap[Pipe]) == 0)
        {
            if (GetLastError() != ERROR_IO_PENDING)
            {
                // Severe error on pipe. Close this
                // handle forever.
                printf("ConnectNamedPipe for pipe %d failed with"
                    " error %d\n", i, GetLastError());
                CloseHandle(PipeHandles[Pipe]);
            }
        }

        DataRead[Pipe] = FALSE;
    }
    else
    {
        // Check the state of the pipe. If DataRead equals
        // FALSE, post a read on the pipe for incoming data.
        // If DataRead equals TRUE, then prepare to echo data
        // back to the client.

        if (DataRead[Pipe] == FALSE)
        {
            // Prepare to read data from a client by posting a
            // ReadFile operation

            ZeroMemory(&Ovlap[Pipe], sizeof(OVERLAPPED));
            Ovlap[Pipe].hEvent = Event[Pipe];

            if (ReadFile(PipeHandles[Pipe], Buffer[Pipe],
                BUFFER_SIZE, NULL, &Ovlap[Pipe]) == 0)
            {
                if (GetLastError() != ERROR_IO_PENDING)
                {
                    printf("ReadFile failed with error %d\n",
                    GetLastError());
                }
            }

            DataRead[Pipe] = TRUE;
        }
        else
        {
            // Write received data back to the client by
            // posting a WriteFile operation
            printf("Received %d bytes, echo bytes back\n",
                BytesTransferred);

            ZeroMemory(&Ovlap[Pipe], sizeof(OVERLAPPED));
            Ovlap[Pipe].hEvent = Event[Pipe];
```

```
            if (WriteFile(PipeHandles[Pipe], Buffer[Pipe],
                BytesTransferred, NULL, &Ovlap[Pipe]) == 0)
                {
                    if (GetLastError() != ERROR_IO_PENDING)
                    {
                        printf("WriteFile failed with error %d\n",
                        GetLastError());
                    }
                }

                DataRead[Pipe] = FALSE;
            }
        }
    }
}
```

服务器应用要想同时为五个管道实例提供服务，必须调用五次 CreateNamedPipe函数，取得每个管道的实例句柄。服务器取得所有实例句柄后，便需针对每个管道，使用一个重叠式I/O结构，以异步方式调用五次 ConnectNamedPipe函数，开始对客户机连接请求的“监听”。客户机建好与服务器的连接后，所有 I/O都会以异步方式进行处理。若客户机断开连接，服务器便会先调用DisconnectNamedPipe函数，再重新执行一个ConnectNamedPipe调用，实现每一个管道实例句柄的重复利用。

3. 安全模拟

之所以会选择命名管道作为自己的网络编程方案，一个最好的理由便是它们依赖于Windows NT及Windows 2000的安全机制，在客户机试图建立同服务器的连接时，实现对访问的控制。Windows NT和Windows 2000安全机制具有“模拟”能力，允许一个命名管道服务器应用在客户机的安全环境中执行。执行一个命名管道服务器应用时，它通常会在用于启动该应用的那个进程的安全环境许可级别上工作。例如，假如拥有管理员权限的某人启动了一个命名管道服务器，服务器便有权访问 Windows NT或Windows 2000系统上的几乎任何资源。此时，假如在CreateNamedPipe中指定的SECURITY_DESCRIPTOR结构允许所有用户访问这个命名管道，就会埋下极大的安全隐患。

若服务器用 ConnectNamedPipe函数接受一个客户机连接请求，便可调用 API函数ImpersonateNamedPipeClient，令自己的执行线程在客户机的安全环境下工作。该函数的定义如下：

```
BOOL ImpersonateNamedPipeClient(
    HANDLE hNamedPipe
);
```

其中，hNamedPipe参数对应于自CreateNamedPipe返回的管道实例句柄。调用了这个函数之后，操作系统便会将服务器的线程安全环境变成客户机的安全环境。整个过程显得非常便利：假如设计自己的服务器时，目的便是访问像文件这样的资源，便会使用客户机的访问权限来进行。这样一来，服务器便能保持对资源的访问控制，无论由谁启动了这个进程。

若服务器的线程在客户机的安全环境中执行，它需要设置一个安全模拟级别。共有四个基本的模拟级别：匿名（Anonymous）、验证（Identification）、模拟（Impersonation）以及委派（Delegation）。通过设置不同的级别，便可控制服务器在多大的程度上“类似”于一个客户机，或者说，模拟的程度有多大。至于这些模拟级别更深入的情况，本章稍后在谈到一个

客户机应用的开发时，还会进一步地讲解。服务器完成了对一个客户机会话的处理之后，便应调用RevertToSelf，恢复成自己最初的线程执行安全环境。对 RevertToSelf这个API函数的定义如下：

```
BOOL RevertToSelf(VOID);
```

注意该函数无任何参数。

4.2.3 客户机的细节

实现一个命名管道客户机时，要求开发一个应用程序，令其建立与某个命名管道服务器的连接。注意客户机不可创建命名管道实例。然而，客户机可打开来自服务器的、现成的实例。下述步骤讲解了如何编写一个基本的客户机应用：

1) 用API函数WaitNamedPipe，等候一个命名管道实例可供自己使用。

2) 用API函数CreateFile，建立与命名管道的连接。

3) 用API函数WriteFile和ReadFile，分别向服务器发送数据，或从中接收数据。

4) 用API函数CloseHandle，关闭打开的命名管道会话。

建立一个连接之前，客户机需要用 WaitNamedPipe函数，检查是否存在一个现成的命名管道实例。对该函数的定义如下：

```
BOOL WaitNamedPipe(
    LPCTSTR lpNamedPipeName,
    DWORD nTimeOut
);
```

其中，lpNamedPipeName参数指定了试图与之建立连接的那个命名管道。 nTimeOut（超时）参数指定客户机需要等待一个管道的服务器进程多久的时间，让它在管道上完成一个待决的ConnectNamedPipe操作。

WaitNamedPipe成功完成后，客户机需要用 CreateFile这个API函数，打开指向服务器命名管道实例的一个句柄。 CreateFile的定义如下：

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

其中，lpFileName（文件名）参数指定希望打开的那个管道的名字；这个名字必须遵守第4章开头讲到的管道命名规范。

dwDesiredAccess参数定义了访问模式，应将其设为 GENERIC_READ（用于从管道上读取数据），或设为GENERIC_WRITE（用于将数据写入管道）。这些标志亦可统一起来设定，方法是对两个标志进行 OR（或）运算，得到最后的结果。注意访问模式必须兼容于管道当初在服务器上的创建方式。换言之，这个模式必须与 CreateNamedPipe的dwOpenMode参数中指定的那个相符（参见前文）。例如，假定服务器用 PIPE_ACCESS_INBOUND创建了一个管道，客户机便应指定GENERIC_WRITE。

dwShareMode参数应设为 0，因为一次只能有一个客户机访问一个管道实例。

lpSecurityAttributes参数应设为NULL，除非需要子进程继承客户机的句柄。之所以不能用这个参数来指定安全控制选项，完全是由于 CreateFile不能创建命名管道实例的缘故。dwCreationDisposition参数则应设为OPEN_EXISTING，意味着CreateFile函数会在命名管道不存在的情况下调用失败。

dwFlagsAndAttributes无论如何都应设为FILE_ATTRIBUTE_NORMAL。另外，该参数本来还可以选择 FILE_FLAG_WRITE_THROUGH、FILE_FLAG_OVERLAPPED以及SECURITY_SQOS_PRESENT标志，并将它们同FILE_ATTRIBUTE_NORMAL标志通过OR（或）运算合并到一起。其中，FILE_FLAG_WRITE_THROUGH和FILE_FLAG_OVERLAPPED的工作方式类似于在表4-1总结的服务器模式标志。SECURITY_SQOS_PRESENT标志则用于控制在一个命名管道服务器上，客户机的模拟安全级别。根据安全模拟级别，我们便可知道一个服务器进程与客户机进程在多大程度上类似。客户机可在建立与服务器的连接时，指定这一信息。若客户机设定了SECURITY_SQOS_PRESENT标志，那么必须使用下文总结的一个或多个安全标志：

■ SECURITY_ANONYMOUS

指定在“匿名”（Anonymous）安全级别上，对客户机加以模拟。服务器进程不可取得与客户机有关的身份验证信息，而且不能在客户机的安全环境中执行。

■ SECURITY_IDENTIFICATION

指定在“验证”（Identification）安全级别上，对客户机加以模拟。服务器进程可获取与客户机有关的一些信息，比如安全标识符以及优先权限等等。然而，却不能在客户机的安全环境中执行。假如命名管道客户机希望让服务器对客户机进行验证，但却不想让它完全扮演客户机，这一设定便非常恰当。

■ SECURITY_IMPERSONATION

指定在“模拟”（Impersonation）安全级别上，对客户机加以模拟。此时，客户机希望允许服务器进程获取与客户机有关的信息，并可在自己的本地系统中，在客户机的安全环境中执行。使用这一标志，服务器便能以客户机的身份，访问服务器上的任何本地资源。在这种情况下，服务器完全“模拟”或“扮演”了一个客户机。

■ SECURITY_DELEGATION

指定在“委派”（Delegation）安全级别上，对客户机加以模拟。服务器进程可取得与客户机有关的信息，并可同时在本地系统以及远程系统上，使用客户机的安全场景执行。

注意 只有服务器进程在Windows 2000上运行的时候，SECURITY_DELEGATION才可产生作用。Windows NT 4（包括最新的SP6）并未实现委派安全机制。

■ SECURITY_CONTEXT_TRACKING

指出安全追踪模式是“动态”的。若未设定该标志，安全追踪模式便是“静态”的。

■ SECURITY_EFFECTIVE_ONLY

指出在客户机安全环境中，只有已启用的那些部分才可由服务器使用。假如未设定该标志，客户机安全环境的所有部分均可使用，无论是否已经启用。

命名管道安全模拟的情况已在本章4.2.1节“服务器的细节”介绍过了。

CreateFile的最后一个参数是 hTemplateFile，它对命名管道无效，应设为 NULL。如CreateFile成功完成，未产生错误，客户机应用便可开始通过 ReadFile和WriteFile函数，在命

名管道上发送及接收数据。应用程序完成了数据的处理之后，可用 CloseHandle函数，将连接关闭（断开）。

程序清单4-4向大家展示了一个简单的命名管道客户机程序，阐述了成功开发一个基本命名管道客户机应用所需的各种 API调用。一旦该应用程序成功建立了同一个命名管道的连接，便会向服务器写入这样的一条消息： This is a test（这是一次测试）。

程序清单4-4 简单的命名管道客户机

```
// Client.cpp

#include <windows.h>
#include <stdio.h>

#define PIPE_NAME "\\\\.\\Pipe\\jim"

void main(void)
{
    HANDLE PipeHandle;
    DWORD BytesWritten;

    if (WaitNamedPipe(PIPE_NAME, NMPWAIT_WAIT_FOREVER) == 0)
    {
        printf("WaitNamedPipe failed with error %d\n",
            GetLastError());
        return;
    }

    // Create the named pipe file handle
    if ((PipeHandle = CreateFile(PIPE_NAME,
        GENERIC_READ | GENERIC_WRITE, 0,
        (LPSECURITY_ATTRIBUTES) NULL, OPEN_EXISTING,
        FILE_ATTRIBUTE_NORMAL,
        (HANDLE) NULL)) == INVALID_HANDLE_VALUE)
    {
        printf("CreateFile failed with error %d\n", GetLastError());
        return;
    }

    if (WriteFile(PipeHandle, "This is a test", 14, &BytesWritten,
        NULL) == 0)
    {
        printf("WriteFile failed with error %d\n", GetLastError());
        CloseHandle(PipeHandle);
        return;
    }

    printf("Wrote %d bytes", BytesWritten);

    CloseHandle(PipeHandle);
}
```

4.3 其他API调用

迄今为止，尚有几个特殊的命名管道函数是我们一直没有提及的。 CallNamedPipe和

TransactNamedPipe这两个API调用可有效减轻一个应用程序编码的复杂性。这两个函数均能在一次调用中，同时执行读和写操作。其中， CallNamedPipe函数允许客户机应用建立与一个消息类型的管道的连接（假如当时没有可用的管道实例，便会一直等候下去），然后在管道上读写数据，最后关闭这个管道。事实上，这几乎是一个完整的客户机应用，只是已在一个调用中全部写好了！对CallNamedPipe的定义如下：

```
BOOL CallNamedPipe(
    LPCTSTR lpNamedPipeName,
    LPVOID lpInBuffer,
    DWORD nInBufferSize,
    LPVOID lpOutBuffer,
    DWORD nOutBufferSize,
    LPDWORD lpBytesRead,
    DWORD nTimeOut
);
```

其中，lpNamedPipeName参数包含的是一个字串，采用 UNC名字格式，指定了一个命名管道。lpInBuffer和nInBufferSize参数分别指定一个缓冲区的地址与大小，应用程序打算用这个缓冲区将数据写入服务器。 lpOutBuffer和nOutBufferSize则指定了另一个缓冲区的地址和大小，不过应用程序用它从服务器那里接收数据。 在lpBytesRead参数中，包含了从管道读回的字节数。而nTimeOut指定的是一个时间长度，以毫秒为单位，规定了在一个命名管道可用之前，最长能等待多久的时间。

TransactNamedPipe函数既可在客户机应用中使用，亦可在服务器应用中使用。设计它的目的是为了将读操作与写操作整合到一个 API调用之中。 这样一来，由于减轻了MSNP重定向器收发数据的负担，所以能在一定程度上优化网络 I/O的性能。对TransactNamedPipe的定义如下：

```
BOOL TransactNamedPipe(
    HANDLE hNamedPipe,
    LPVOID lpInBuffer,
    DWORD nInBufferSize,
    LPVOID lpOutBuffer,
    DWORD nOutBufferSize,
    LPDWORD lpBytesRead,
    LPOVERLAPPED lpOverlapped
);
```

其中，hNamedPipe参数指定的是由CreateNamedPipe或CreateFile这两个API函数返回的命名管道。lpInBuffer和nInBufferSize参数指定一个缓冲区的地址和大小，应用程序用这个缓冲区将数据写入管道。lpOutBuffer和nOutBufferSize参数指定的则是另一个缓冲区的地址和大小，应用程序利用这个缓冲区从管道取得数据。 lpBytesRead参数用于接收自管道读回的实际字节数量。lpOverlapped参数允许这个TransactNamedPipe使用重叠式I/O，以异步形式工作。

接下来的三个函数（ GetNamedPipeHandleState， SetNamedPipeHandleState以及GetNamedPipeInfo）用于使命名管道客户机以及服务器通信在运行时间显得更加灵活。例如，可利用这些函数在运行时间更改一个管道的运行模式，从消息模式变成字节模式，或从字节模式变成消息模式。 GetNamedPipeHandleState用于接收与一个指定命名管道对应的信息，比如运行模式（消息或字节模式）、管道实例数以及缓冲区信息等等。在命令管道的一个实例的

“存在时间”内，不同时间由 GetNamedPipeHandleState函数返回的信息也可能会发生变化。对GetNamedPipeHandleState函数的定义如下：

```
BOOL GetNamedPipeHandleState(
    HANDLE hNamedPipe,
    LPDWORD lpState,
    LPDWORD lpCurInstances,
    LPDWORD lpMaxCollectionCount,
    LPDWORD lpCollectDataTimeout,
    LPTSTR lpUserName,
    DWORD nMaxUserNameSize
);
```

其中，hNamedPipe参数指定的是由CreateNamedPipe或CreateFile这两个API函数返回的一个命名管道。lpState参数是一个变量指针，那个变量负责接收管道句柄的当前工作模式。lpState参数可能返回两个值，一个是PIPE_NOWAIT，另一个是PIPE_READMODE_MESSAGE。lpCurInstances参数也是一个变量指针，那个变量负责当前的管道实例数量。lpMaxCollectionCount参数负责接收实际发送给服务器之前，打算在客户机上收集的最大字节数。 lpCollectDataTimeout参数接收的则是一个时间值，以毫秒为单位。超过这个时间，一个远程命名管道便必须通过网络传出信息。lpUserName和nMaxUerNameSize参数定义的是一个缓冲区，它负责接收一个“空中止”的字串，其中包含了客户机应用的用户名字串。

利用SetNamedPipeHandleState函数，我们可更改由 GetNamedPipeHandleState函数了解到的管道特征。SetNamedPipeHandleState函数的定义如下：

```
BOOL SetNamedPipeHandleState(
    HANDLE hNamedPipe,
    LPDWORD lpMode,
    LPDWORD lpMaxCollectionCount,
    LPDWORD lpCollectDataTimeout
);
```

其中，hNamedPipe参数指定的是由CreateNamedPipe或CreateFile这两个API函数返回的一个命名管道。lpMode参数设置管道的工作（运行）模式。 lpMaxCollectionCount参数指定的是客户机上收集的最大字节数量，随后数据便需发给服务器。 lpCollectDataTimeout参数以毫秒为单位指定了一个时间值，该值指明一个远程命名管道客户机通过网络传送信息之前，最多需等待的时间。

GetNamedPipeInfo这个API函数用于获得缓冲区大小以及管道实例最大数量信息。对它的定义如下：

```
BOOL GetNamedPipeInfo(
    HANDLE hNamedPipe,
    LPDWORD lpFlags,
    LPDWORD lpOutBufferSize,
    LPDWORD lpInBufferSize,
    LPDWORD lpMaxInstances
);
```

其中，hNamedPipe参数指定的是由CreateNamedPipe或CreateFile这两个API函数返回的一个命名管道。lpFlags参数取得命名管道的类型，并判断它到底是一个服务器，还是一个客户机，以及管道工作于字节模式，还是消息模式。 lpOutBufferSize参数以字节为单位，指定了

用来保存外出数据的内部缓冲区的大小；lpInBufferSize参数则以字节为单位，取得用于保存进入数据的内部缓冲区的大小。lpMaxInstance用于取得可以创建的管道实例的最大数量。

最后一个API函数是PeekNamedPipe，可用它对命令管道内的数据进行浏览，同时毋需将其从管道的内部缓冲区挪出。假如应用程序希望对进入的数据进行“轮询”，以免进行ReadFile这个API调用时发生“锁定”现象（执行暂停），便可考虑使用这一函数。另外，假若应用程序需要在数据实际接收之前，先作一番检查，这个函数也是相当有用的。例如，应用程序可能希望根据进入消息的长度，先对应用程序的缓冲区进行一番调节。 搞好后，再正式接收数据。PeekNamedPipe函数的定义如下：

```
BOOL PeekNamedPipe(
   HANDLE hNamedPipe,
   LPVOID lpBuffer,
   DWORD nBufferSize,
   LPDWORD lpBytesRead,
   LPDWORD lpTotalBytesAvail,
   LPDWORD lpBytesLeftThisMessage
);
```

其中，hNamedPipe参数用于指定由CreateNamedPipe或CreateFile这两个API函数返回的一个命名管道。lpBuffer和nBufferSize参数分别定义的是一个接收专用缓冲区的地址及大小，用于从管道取得数据。 lpBytesRead参数负责接收从管道实际读入缓冲区的字节数量。lpTotalBytesAvail参数用于接收可从管道发出的字节总数。lpBytesLeftThisMessage参数用于接收消息内尚存的字节数量（前提是管道用消息模式打开）。假如一条消息的实际长度大于由lpBuffer参数指定的那个缓冲区的长度，消息内剩下的字节便会返回。对于在字节模式下工作的命名管道而言，该参数则无论如何都会返回 0。

4.4 平台和性能问题

在微软知识库中，可找到下述问题及限制定义。这个知识库可在网上访问到，地址是http://support.microsoft.com/support。下面是对每一个问题的简要说明：

■ Q100291：命名管道名字的限制

假如创建了名为\\.\Pipe\Mypipes的管道，以后便不能创建名为 \\.Pipe\Mypipes\Pipe1的管道，因为\\.\Pipe\Mypipes已经是一个管道名，不可作为子目录使用。

■ Q119218：命名管道写操作限制在64K之内

若API函数WriteFile试图用一个大于64KB的缓冲区，向一个处于消息模式的命名管道写入数据，该函数便会返回FALSE，而GetLastError调用会返回ERROR_MORE_DATA。

■ Q110148：由WriteFile或ReadFile函数返回ERROR_INVALID_PARAMETER错误

假如在一个命名管道上工作，并使用重叠式 I/O，那么WriteFile或ReadFile函数调用都有可能失败，并返回 ERROR_INVALID_PARAMETER错误。这种失败一项可能的促因是OVERLAPPED结构的Offset以及OffsetHigh这两个成员未设为0。

■ Q180222：Windows 95的WaitNamedPipe和253号错误

在Windows 95中，假如将一个无效管道名作为第一个参数传递，那么在 WaitNamedPipe函数调用失败以后，用 GetLastError会返回一个错误253。对这个函数来说， 253并非一个标准的（事先定义的）错误代码。而假如换到 Windows NT 4上运行同样的代码，返回的错误代码

是161（ERROR_BAD_PATHNAME）。要想解决这种不一致的问题，请将253号错误解释成标准的161号错误：ERROR_BAD_PATHNAME（路径名错误）。

■ Q141709：单台工作站最多只能建立49个命名管道连接

假如命名管道服务器应用创建了49个以上的直接命名管道，那么在远程计算机上，一个客户机最多只能建立前面的49个命名管道连接，多余的只好忽略。

■ Q126645：从某项服务打开一个命名管道时，出现访问被拒的情况

假如一项服务采用“本地系统”帐号运行，同时试图打开在Windows NT上运行的一个命名管道，那么操作便会失败，并返回“拒绝访问”错误信息，亦即错误代码5。

4.5 小结

本章向大家介绍了命名管道网络编程技术，它为我们建立了一个简单的客户机／服务器数据通信体系，可确保数据进行可靠传输。接口依赖于Windows重定向器，以便通过一个网络来传送数据。对命令管道而言，它最大的一项好处便是直接利用了Windows NT及Windows 2000的安全机制，该机制是本书讲到的其他网络技术均不具备的一项好处！下面第二部分将向大家深入讲解Winsock技术，以便应用程序利用一种网络传输协议，进行“直接”通信。

第二部分 Winsock API

本书第二部分讲述的是在 Win32平台上的 Winsock编程。对于众多的基层网络协议，Winsock是访问它们的首选接口。而且在每个 Win32平台上，Winsock都以不同的形式存在着。Winsock是网络编程接口，而不是协议。它从 Unix平台的Berkeley（BSD）套接字方案借鉴了许多东西，后者能访问多种网络协议。在 Win32环境中，Winsock接口最终成为一个真正的“与协议无关”接口，尤其是在 Winsock 2发布之后。

接下来的三章讲解了协议和 Winsock的基本知识，其中包括每种协议的定址方案，以及一个简单的Winsock客户机／服务器示例。再后面的几章讲述的是 Winsock 2中的一些新特性，比如传送服务提供者、命名空间提供者和服务质量（ QoS）等等。就其中的一部分技术来说，可能会存在一些容易混淆的地方。原因是尽管它们都在 Winsock 2规范中得到了定义，而 Winsock 2在目前所有的Win32平台上都已得到支持（Windows CE除外，本部分还会对其详述），但落实到具体的平台，却并非所有这些特性都已真正地实现。必要时，我们会指出具体的限制在哪里。本部分开始之前，我们假定大家已具备了 Winsock（或BSD套接字）的基本知识，而且多少熟悉一些客户机／服务器的 Winsock基本知识。

第5章 网络原理和协议

建立Winsock 2规范的主要目的是提供一个与协议无关的传送接口。在各种不同的网络传送协议上，假若能为网络编程提供一个大家都很熟悉的接口，当然是一件不错的事情。但尽管如此，仍需注意各种网络协议的一些特征。本章将全面讲述使用特定协议时应该留意的一些特征，其中包括一些基本的网络连接原理。另外，我们还将讨论如何通过程序向 Winsock查询协议信息，并探讨针对一种具体协议创建套接字所需的基本步骤。

5.1 协议的特征

本章第一小节着重讲解目前常用网络传送协议的一些基本特征。通过这里的学习，大家可掌握与协议行为类型有关的一些背景知识。同时，作为程序员，可大致知道特定协议在程序中的行为方式。

5.1.1 面向消息

对每个离散写命令来说，如果传送协议把它们（而且只有它们）当做一条独立的消息在网上传送，我们就说该协议是面向协议的。同时，还意味着接收端在接收数据时，返回的数据是发送端写入的一条离散消息。接收端不能得到更多的消息，仅此而已。比如，在图 5-1中，左边的工作站向右边的工作站提交了三条分别是 128、64和32字节的消息。作为接收端的工作站发出三条读取命令，缓冲区是 256个字节。后来的各次调用返回的分别是 128、 64和32个字

节。第一次读取调用不会将这所有的三个数据包都返回，即使这些数据包已经收到也如此。这称为“保护消息边界”（preserving message boundaries），一般出现在交换结构化数据时。网络游戏是“保护消息边界”的较好范例。每个玩家均向别的玩家发出一个带有地图信息的数据包。这种通信后面的代码很简单：一个玩家请求一个数据包，另一个玩家又准确地从别的玩家处获得一个地图信息数据包。

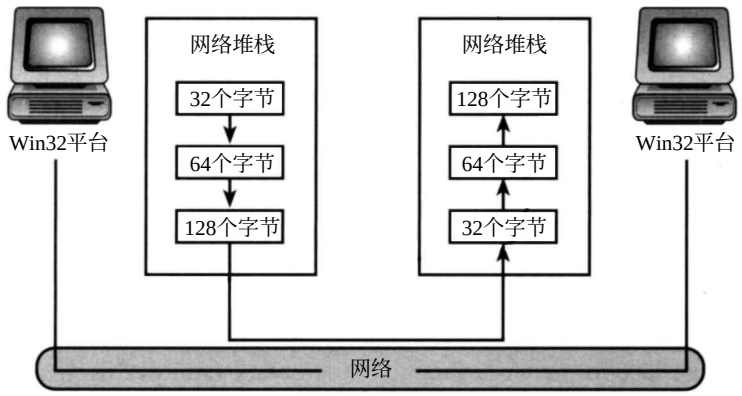

图5-1 数据报服务

无保护消息边界的协议通常称作“基于流的协议”。大家要知道“基于流的协议”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，意味着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，形成一个较大的数据包。对接收端来说，则是数据一到达网络堆栈，网络堆栈就开始读取它，并将它缓存下来等候进程处理。在进程请求处理大量数据时，系统会在不溢出为客户请求提供的缓冲区这一前提下，尽量返回更多的数据。在图 5-2中，发送端提交了三个数据包：分别是 128、64和32个字节；但是，本地系统堆栈自由地把这些数据聚合在一起，形成一个大的数据包。这种情况下，重组后的 2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受许多因素的影响，比如最大传输单元或 Nagle算法。在TCP/IP中，在将积累起来的数据发到线上之前， Nagle算法在等候数据积累的主机中进行。在需要发送的数据积累到一定数量或预定时间已超过之前，这个主机会一直等下去。实施Nagle算法时，主机的通信方在发送主机确认之前，会等一等外出数据，主机的通信方就不必发送一个只有确认的数据包。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，相当烦人。

在接收端，网络堆栈把所有进来的数据包聚集在一起，归入既定进程。我们来看看图 5-2。如果接收端执行一次256字节缓冲区的读取，系统马上就会返回 224个字节。如果接收端只要求读取20个字节，系统就会只返回20个字节。

伪流

伪流（pseudo-stream）这个术语常用于某种系统中，该系统使用的协议是基于消息的，发送的数据分别在各自独立的数据包内，接收端读取并把消息缓存在一起， 这样，接收应用程序便可读取任意大小的数据块。把图 5-1中的发送端和图5-2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各自独立的数据包，但接收端可以对收到的数据包自由组合。一般情况下，可把伪流视作一个普通的“面向流的协议”。

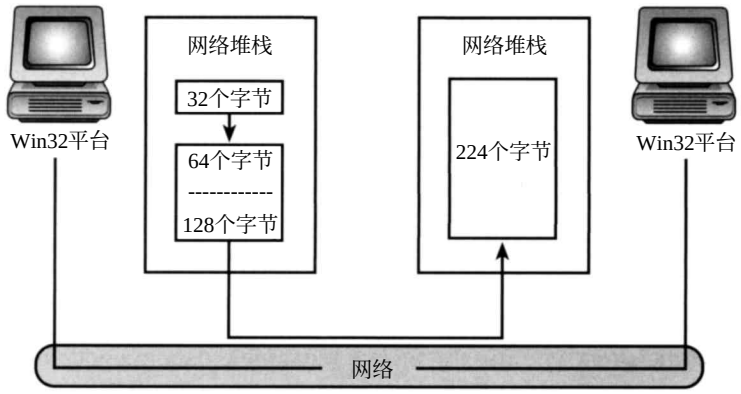

图5-2 流服务

5.1.2 面向连接和无连接

通常情况下，一个协议提供面向连接的服务，或提供无连接的服务。面向连接的服务中，进行数据交换之前，必须与通信方建立一条路径。这样既确定了通信方之间存在路由，又保证了通信双方都是活动的、都可彼此响应，但其特点是在通信双方之间建立一个通信信道需要很多开支。除此以外，大部分面向连接的协议为保证投递无误，可能会因为执行额外的计算来验证正确性，因此，进一步增加开支。而无连接协议却不保证接收端是否正在收听。无连接服务类似于邮政服务：发信人把信装入邮箱即可。至于收信人是否想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等等，发信人都不得而知。

5.1.3 可靠性和次序性

在设计用于特定协议的应用程序来说，可靠性和次序性是我们必须了解的最具决定性的特性。大多数情况下，可靠性和次序性与协议是无连接的，还是面向连接的密切相关。可靠性（或保证投递）保证了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，同样不能保证数据的完整性。

次序性是指对数据到达接收端的顺序进行处理。保护次序性的协议保证接收端收到数据的顺序就是数据的发送顺序。显然，没有保护次序性的协议就没有次序保证。

在面向连接的通信中，如果你一直试图在通信双方建立一个显式通信信道，一般希望协议能够保证数据的完整性和次序性。多数情况下，面向连接的协议的确能保证数据的可靠性。下一小节，我们将讨论真正的协议及其特征。注意，可靠性和次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。当然，无连接协议的过人之处就是速度：对接入接收端的虚拟连接来说，它们不会影响其建立。为什么这样反而降低了错误检查的速度呢？原来这便是无连接协议一般不保证数据完整性和次序性，而面向连接的协议却能做到的原因。数据报如此不完善，为什么大家都喜欢用它呢？原来，无连接协议按数量大小排序，比面向连接的通信快得多。另外，它不必去验证数据完整性和数据收到确认与否和发送小量数据时增加的复杂性等等。对非临界的数据传输来说，数据报非常有用。而前面讨论过的网络游戏示例之类的应用，数据报简直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，很快就能

收到另外一个，像无缝通信一样方便。

5.1.4 从容关闭

从容关闭只出现在面向连接的协议中。在这种关闭过程中，一方开始关闭通信会话，但另一方仍然可以读取线上或网络堆栈上已挂起的数据。如果面向连接的协议不支持从容关闭，只要其中一方关闭了通信信道，都会导致连接立即中断，数据丢失，接收端不能读取数据这些情况出现。如果使用 TCP协议，连接双方都必须执行一次关闭，以便完全中断连接。发起方向另一个通信方发出一个带有 FIN控制标志的分段（数据报）。接收端的通信方则向发起方返回一个ACK控制标志，确认收到FIN，但此时发起方仍然可发送数据。 FIN控制标志表示发起关闭的这一方不再发送数据。一旦通信方决定不需要发送数据，它就会发出一个 FIN控制标志，并带有发起方确认的ACK控制标志。这时，该连接已完全关闭。

5.1.5 广播数据

广播数据即数据从一个工作站发出，局域网内的其他所有工作站都能收到它。这一特征适用于无连接协议，因为 LAN上的所有机器都可获得并处理广播消息。使用广播消息的不利之处是每台机器都必须对该消息进行处理。比如，一用户在 LAN上广播一条消息，每台机器上的网卡都会收到这条消息，并把它上传到网络堆栈。然后，堆栈将这条消息在所有的网络应用中循环，看它们是否应该接收这条消息。通常，这个局域网上的多数机器对该消息都不感兴趣，草草地一弃了之。但是，各台机器仍需花时间来处理这个数据包，看是否有应用对它感兴趣。结果，高广播通信流使 LAN上的机器陷入困境，因为每个工作站都要检查这个数据包。一般情况下，路由器都不会传送广播包。

5.1.6 多播数据

多播是指一个进程发送数据的能力，这些数据即将由一个或多个接收端进行接收。进程加入一个多播会话的方法和采用的基层协议有关。比如， IP协议下，多播是广播的一种变形。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。进程希望加入多播组时，网卡上会增添一个过滤器，这样，只有绑定组地址的数据才会被网络硬件捡起，并上传到网络堆栈进行恰当处理。视频会议应用常常使用多播。第 11章将详细论述Winsock的多播编程和其他一些关键问题。

5.1.7 服务质量

服务质量（QoS）是应用的一种能力，用以请求针对专门用途分配特定的带宽。服务质量的好处可从实时视频流式传输中一见端倪。对实时视频流式传输应用的接收端而言，要显示平滑清晰的图像，发送的数据必须限定在特定范围内。过去，应用是将数据缓存下来，再从缓冲区中重播帧，进而获得流畅的视频。如果收到数据的时间太久，重播例程就可拥有大量可播放的、缓存起来的帧。 QoS允许网络上预留的带宽，因此，可以在保证的强制范围内，离线读取帧。理论上讲，这意味着同一个实时视频流式传输应用可用 QoS消除缓存帧的必要。我们将在第12章详细讨论QoS。

5.1.8 部分消息

部分消息只用于面向消息的协议。比如说，一个应用想接收消息，但本地计算机已收到了它的部分数据。这是相当常见的，特别是发送端计算机正在传输大型消息时。本地计算机可能没有足够多的资源来容纳整条消息。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 —— 大的足以需要根据物理方法来将其分解成许多小传输块。如此一来，用户的应用请求读取这条消息时，就有这种可能性存在 —— 用户系统可能只收到了部分消息。如果特定协议支持部分消息，就会通知读取方“已返回的数据只是整条消息中的一部分”。如果协议不支持部分消息，基层网络堆栈就会坚持接收其余的消息，直到收完为止。如果由于某种原因，整条消息不能全部到达，那么收到的数据报就会不完整，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简单地丢弃。

5.1.9 路由选择的考虑

一个重要考虑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建立一条成功的通信路径（要么是面向连接的回路，要么是数据报的数据路径），不管这两个工作站之间存在的网络硬件是什么。比如，机器 A和机器B分别在各自的子网中。连接两个子网的路由器A把这两台机器分开了。一个可路由的协议意识到机器 B和机器A不在同一个子网上：它就会把数据包定向到路由器，由路由器来决定数据的最佳转发方式，以便数据能抵达机器 B。非路由协议不能作出这样的规定：路由器会将它收到的非路由协议数据包统统丢弃。路由器不对发自非路由协议的数据包进行转发，即便数据包的既定目的地在其连接的子网上。NetBEUI是Win32平台支持的唯一的一个协议，它不能路由。

5.1.10 其他特征

Win32平台上支持的各个协议都展现了特定的或独特的特征。同时，还有大量不胜枚举的其他特性，比如字节序和最大传输单元，都可用来说明目前网络上所用的各种协议。然而，对编写Winsock应用程序来说，并不一定要面面俱到，囊括所有特征。 Winsock 2提供了一种功能，可把各个实用的协议提供者及其特征列举出来。本章 5.3节对这一功能进行解释，并展示了一个代码实例。

5.2 支持的协议

Win32平台提供的最有用的特征之一是能够同步支持多种不同的网络协议。正如大家在第2章中看到的那样， Windows重定向器保证将网络请求路由到恰当的协议和子系统；但是，有了Winsock，就可以编写可直接使用任何一种协议的网络应用程序了。第 6章将讨论利用适用于某一工作站的种种协议，以及如何为网络中的机器定址的问题。利用 Winsock编程接口的好处之一是因为它是一个与协议无关的接口。不管使用的是哪一种协议，它们的操作大多数是相通的。尽管如此，要为网络通信定位和连接通信方，仍然必须了解如何为工作站定址。

5.2.1 支持的Win32网络协议

Win32平台支持相当多的协议。通常，各个协议都同时具备多种行为。比如，网际协议

（IP）既支持面向连接的流服务，又支持无连接的数据报服务。表 5-1提供了大部分实用协议及各协议支持的一些行为。

表5-1 实用协议的特性

协 议	名 称	消息类型	连接类型	可靠性	数据包的次序性	从容关闭	广播支持	多点传输支持	服务质量	最大消息量(字节)
IP	MSAFD TCP	流	连接	有	有	有	无	无	无	无
	MSAFD UDP	消息	无连接	无	无	无	有	有	无	65467
	RSVP TCP	流	连接	有	有	有	无	无	有	无
	RSVP UDP	消息	无连接	无	无	无	有	有	有	65467
IPX/SPX	MSAFD nwlnkipx[IPX]	消息	无连接	无	无	无	有	有	无	576
	MSAFD nwlnkspx[SPX]	消息	连接	有	有	无	无	无	无	无
	MSAFD nwlnkspx[SPX][伪流]	消息	连接	有	有	无	无	无	无	无
	MSAFD nwlnkspx[SPXII]	消息	连接	有	有	有	无	无	无	无
	MSAFD nwlnkspx[SPXII][伪流]	消息	连接	有	有	有	无	无	无	无
NetBIOS	连续数据包	消息	连接	有	有	无	无	无	无	64KB (65535)
	数据报	消息	无连接	无	无	无	有① (SP25)	无	无	64KB (65535)
AppleTalk	MSAFD AppleTalk[ADSP]	消息	连接	有	有	有	无	无	无	64KB (65535)
	MSAFD AppleTalk[ADSP][伪流]	消息	连接	有	有	有	无	无	无	无
	MSAFD AppleTalk[PAP]	消息	连接	有	有	有	无	无	无	4096
	MSAFD AppleTalk[RTMP]	流	无连接	无	无	无	无	无	无	无
	MSAFD AppleTalk[ZIP]	流	无连接	无	无	无	无	无	无	无
ATM	MSAFD ATM AAL5	流	连接	无	有	无	无	有	有	无
	本地 ATM(AAL5)	消息	连接	无	有	无	无	有	有	无
红外线套接字	MSAFD lrda[lrDA]	流	连接	有	有	有	无	无	无	无

① NetBIOS支持发送到一个或一组NetBIOS客户机的数据报。但不支持全面广播。

5.2.2 Windows CE网络协议

Windows CE和其他Win32平台不同，它只支持 TCP/IP网络传输协议。除此以外，Windows CE只支持Winsock 1.1规格，这意味着本小节讨论的Winsock 2特性大多不能用于这个平台。Windows CE通过一个重定向器支持依赖于TCP/IP协议进行通信的 NetBIOS，但不会通过本地NetBIOS API或Winsock为NetBIOS提供编程接口。

5.3 Winsock 2协议信息

针对指定的工作站上安装哪种协议和各种协议特性的返回问题， Winsock 2提供了一种解

决办法。如果一个协议支持多种行为，每类行为在系统中都有各自的目录条目。比如，如果在自己的系统中安装了 TCP/IP，系统中就会有两个 IP条目：一个条目针对 TCP，是可靠的而面向连接的；而另一个条目针对 IP，是不可靠且有无连接的。

要想获得系统中安装的网络协议的相关信息，调用这个函数 WSAEnumProtocols即可，并像这样定义它：

```
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中必不可少的函数。唯一的不同是：WSAEnumProtocols返回的是一个WSAPROTOCOLS_INFO结构的数组；而EnumProtocols返回的则是PROTOCOL_INFO结构的数组，该数组中包含的字段少于WSAPROTOCOLS_INFO结构中的字段（但信息是差不多的）。WSAPROTOCOL_INFO结构的格式如下：

```
typedef struct _WSAPROTOCOL_INFOW {
    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;
    WCHAR               szProtocol[WSAPROTOCOL_LEN + 1];
} WSAPROTOCOL_INFOW, FAR * LPWSAPROTOCOL_INFOW;
```

打开Winsock

在可以调用一个Winsock函数之前，必须先加载一个版本正确的 Winsock库。Winsock启动例程是WSAStartup，它的定义是：

int WSAStartup(WORD wVersionRequested,LPWSADATA lpWSAData)

第一个参数是准备加载的 Winsock库的版本号。就目前的 Win32平台而言，Winsock 2库的最新版本是 2.2。唯一的例外是 Windows CE，它只支持 Winsock 1.1版。如果需要 Winsock 2.2版，指定这个值（0x0202）或使用宏MAKEWORD(2,2)即可。高位字节指定副版本，而低位字节则指定主版本。

第二个参数是 WSADATA结构，它是调用完成之后立即返回的。 WSADATA包含了

WSAStartup加载的关于Winsock版本的信息。表5-2列出了WSADATA结构的各个字段，WSADATA结构的真正定义是：

```
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, FAR * LPWSADATA;
```

大致说来，在WSADATA结构中，返回的唯一有用的信息是wVersion和wHighVersion。属于最大套接字和最大UDP长度的条目应该从自己正在使用的特定协议目录条目中获取。上一小节曾谈到这一点。

表5-2 WSADATA结构的成员字段

字 段	说 明
wVersion	调用者希望使用的Winsock版本号
wHighVersion	加载的Winsock库所支持的最高Winsock版本，通常和wVersion的值相同
szDescription	加载的Winsock库的文本说明
szSystemStatus	一个文字串，其中包含相应的状态或配置信息
iMaxSockets	套接字的最大编号（Winsock 2或稍后的版本忽略了该字段）
iMaxUdpDg	UDP数据报的最大容量（Winsock 2或稍后的版本忽略了该字段）
lpVendorInfo	厂商专有信息（Winsock 2或稍后的版本忽略了该字段）

在结束Winsock库，而且不再需要调用任何Winsock函数时，附带例程会卸载这个库，并释放资源。这个函数的定义是：

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

记住，每次调用WSAStartup，都需要调用相应的WSACleanup，因为每次启动调用都会增加对加载Winsock DLL的引用次数，它要求调用同样多次的WSACleanup，以此抵消引用次数。

注意，Winsock 2与Winsock 1.1规格中的所有函数调用完全兼容。因此，对Winsock 1.1规格编写的应用程序来说，在加载Winsock 2库的情况下，该应用程序也能够运行，因为Winsock 1.1函数通过Winsock 2对应函数得以映射。

要调用WSAEnumProtocols函数，最简单的方法是利用相当于NULL和lpdwBufferLength的lpProtocolBuffer，令初次调用为0。调用失败，返回WSAENOBUFS错误，但lpdwBufferLength中包含了恰如其分的缓冲区长度（这一长度是返回所有协议信息所需的）。一旦分配了恰当的缓冲区长度，便可利用这个缓冲区进行下一次调用，该函数返回WSAPROTOCOL_INFO结构所返回的号码。这时，就可逐步深入各结构，查找自己所需要的协议条目了。本书配套CD-ROM上的实例程序Enum.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5-3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说明。要查看特定属性，先选定相应的属性标志，然后对该属性和dwServiceFlags1字段进行按位和运

算。如果结果值非零，这个属性就会出现在指定协议中；反之，则不会出现。

表5-3 协议标志

属 性	含 义
XPl_CONNECTIONLESS	该协议提供无连接的服务。如果不设，则支持面向连接的数据传输
XPl_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
XPl_GUARANTEED_ORDER	保证数据按其发送顺序到达接收端，且数据不会重复。但是，该协议不能保证投递的准确性
XPl_MESSAGE_ORIENTED	实现消息边界
XPl_PSEUDO_STREAM	该协议是面向消息的，但接收端会忽略消息边界
XPl_GRACEFUL_CLOSE	支持二相关闭：通知各个通信方另一方打算关闭通信信道。如果不设，则只执行失败关闭
XPl_EXPEDITED_DATA	该协议提供紧急数据（带外数据）
XPl_CONNECT_DATA	该协议支持带有连接请求的数据传输
XPl_DISCONNECT_DATA	该协议支持带有无连接请求的数据传输
XPl_SUPPORT_BROADCAST	该协议支持广播机制
XPl_SUPPORT_MULTIPOINT	该协议支持多点或多播机制
XPl_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
XPl_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
XPl_QOS_SUPPORTED	该协议标志支持QoS请求
XPl_UNI_SEND	该协议在发送方向上是单向的
XPl_UNI_RECV	该协议在接收方向上是单向的
XPl_IFS_HANDLES	提供者返回的套接字描述符是Installable File System（可安装文件系统）句柄，可用于ReadFile WriteFile之类的API函数中
XPl_PARTIAL_MESSAGE	WSASend和WSASendTo中均支持MSG_PARTIAL标志

大多数协议标志，都将在下面的一章或几章中讨论，所以我们在此不详细介绍各标志的全部含义。其他重要字段是iProtocol、iSocketType和iAddressFamily。iProtocol字段定义该条目属于哪个协议。对支持多种行为的协议来说（比如说面向流的连接或数据报连接）iSocketType字段非常之重要。最后，iAddressFamily字段用于为指定协议区分正确的定址结构。在为指定协议建立套接字时，这三个条目都很重要，我们将在下一小节重点介绍。

5.4 Windows套接字

现在，大家对各种协议及其属性都比较熟悉了。接下来看看这些协议在Winsock中的用法。如果熟悉Winsock，就知道API是建立在套接字基础上的。所谓套接字，就是一个指向传输提供者的句柄。Win32中，套接字不同于文件描述符，所以它是一个独立的类型——SOCKET。套接字是由两个函数建立的：

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

SOCKET socket (
```

```
    int af,
    int type,
    int protocol
);
```

第一个参数af，是协议的地址家族。比如，如果想建立一个UDP或TCP套接字，可用常量AF_INET来指代互联网协议（IP）。第二个参数type，是协议的套接字类型。套接字的类型可以是下面五个值：SOCK_STREAM、SOCK_DGRAM、SOCK_SEQPACKET、SOCK_RAW和SOCK_RDM。第三个参数是protocol。指定的地址家族和套接字类型有多个条目时，就可用这个字段来限定使用特定传输。表5-4列出的值用于针对一个指定网络传输的地址家族、套接字类型和协议字段。

表5-4 套接字参数

协 议	地址家族	套接字类型		协议字段
Internet Protocol (IP)	*AF_INET*	TCP	*SOCK_STREAM*	*IPPROTO_IP*
		UDP	*SOCK_DGRAM*	*IPPROTO_UDP*
		Raw sockets	*SOCK_RAW*	*IPPROTO_RAW*
				IPPROTO_ICMP
IPX/SPX	*AF/NS*	MSAFD nwlnkipx[IPX]	*SOCK_DGRAM*	*NSPROTO_IPX*
	AF_IPX	MSAFD nwlnkspx[SPX]	*SOCK_SEQPACKET*	*NSPROTO_SPX*
		MSAFD nwlnkspx[SPX] [Pseudo-stream]	*SOCK_STREAM*	*NSPROTO_SPX*
		MSAFD nwlnkspx[SPXII]	*SOCK_STREAM*	*NSPROTO_SPXII*
		MSAFD nwlnkspx[SPXII] [Pseudo-stream]	*SOCK_STREAM*	*NSPROTO_SPXII*
NetBIOS	*AF_NETBIOS*	Sequential Packets	*SOCK_SEQPACKET*	*LANA* number
		Datagrams	*SOCK_DGRAM*	*LANA* number
AppleTalk	*AF_APPLETALK*	MSAFD AppleTalk[ADSP]	*SOCK_RDM*	*ATPROTO_ADSP*
		MSAFD AppleTalk[ADSP] [Pseudo-stream]	*SOCK_STREAM*	*ATPROTO_ADSP*
		MSAFD AppleTalk[PAP]	*SOCK_RDM*	*ATPROTO_PAP*
		MSAFD AppleTalk[RTMP]	*SOCK_DGRAM*	*DDPPROTO_RTMP*
		MSAFD AppleTalk[ZIP]	*SOCK_DGRAM*	*DDPPROTO_ZIP*
ATM	*AF_ATM*	MSAFD ATM AAL5	*SOCK_RAW*	*ATMPROTO_AAL5*
		Native ATM(AAL5)	*SOCK_RAW*	*ATMPROTO_AAL5*
Infrared Sockets	*AF_IRDA*	MSAFD Irda[IrDA]	*SOCK_STREAM*	*IRDA_PROTO _SOCK_STREAM*

建立套接字的前三个参数组织成三级。第一个同时也是最重要的参数是地址家族。它指定准备使用哪种协议，另外还为第二和第三个参数指定有效选项。比如，如果选择了 ATM地址家族（AF_ATM），那么在选用套接字类型时，就会限定只能采用原始套接字（SOCK_RAW）。同样，在选定一个地址家族和套接字类型后，也会限定只能用哪种协议。但是，投递协议参数为0是可行的。然后，系统再根据其他两个参数——af和type，选定一个传输提供者。列举协议条目时，要检查WSAPROTOCOL_INFO结构的dwProviderFlags条目，如果socket或WSASocket协议参数是0，它就是即将使用的默认传输。

如果在WSASocket函数时，已经利用WSAEnumProtocols列举了所有协议，就可选定一个WSAPROTOCOL_INFO结构，并将它当作lpProtocolInfo参数投递到 WSASocket。之后，若在前三个参数（af、type和protocol）中都指定常量FROM_PROTOCOL_INFO，就会转而采用WSAPROTOCOL_INFO结构中提供的那三个值。以上便是教大家如何指定一个准确无误的协议条目。

最后两个WSASocket标志很简单。组参数始终为 0，因为目前尚无可支持套接字组的Winsock版本。要指定一个或多个下列标志，可用dwFlags参数：

- WSA_FLAG_OVERLAPPED
- WSA_FLAG_MULTIPOINT_C_ROOT
- WSA_FLAG_MULTIPOINT_C_LEAF
- WSA_FLAG_MULTIPOINT_D_ROOF
- WSA_FLAG_MULTIPOINT_D_LEAF

第一个标志WSA_FLAG_OVERLAPPED，用于指定这个套接字具备重叠I/O（是适用于Winsock的可能实现的通信模式之一）。这个主题将在第8章详细讨论。调用socket建立一个套接字时，WSA_FLAG_OVERLAPPED便是默认设置。一般说来，在使用WSASocket时，最好始终保持设定该标志。后面四个标志用于处理多播套接字。

原始套接字

利用WSASocket建立套接字时，可向函数调用传送一个WSAPROTOCOL_INFO结构，以定义准备建立的那个套接字的类型；尽管如此，还是可建立一些套接字类型（在传输提供者目录中，它们没有相应的条目）。最佳示例是IP协议下的原始套接字。原始套接字一种通信，允许你把其他协议封装在UDP数据包中，比如说“互联网控制消息协议”（ICMP）。ICMP的目的是投递互联网主机间的控制、错误和信息型消息。由于 ICMP不提供任何数据传输功能，因此不能把它与UDP或TCP同等看待，但它和IP本身属于同一个级别。第13章将详细论述原始套接字。

5.5 具体平台的问题

已发布的Windows 95支持Winsock 1.1规格。微软已完成了一个免费的Winsock更新，可从他们的Web站点下载（http://www.microsoft.com/Windows 95/downloads/）。另外，Winsock 2 SDK很有用，其中包括了编译 Winsock 2应用程序所需要的头和库。自然， Windows 98、Windows NT 4和Windows 2000都支持Winsock2，无须任何必要的新增项。Windows CE只支持Winsock 1.1规格。

至于不同传输提供者提供的支持，应该提到的限制极少。 Windows CE只支持TCP/IP和红外线套接字。而对Windows 95和Windows 98而言， Winsock API中尚未揭示NetBIOS传输提供者（利用地址家族AF_NETBIOS的传输）。若调用WSAEnumProtocols函数，不会相应地列出NetBIOS提供者，即使机器上已安装了这些 NetBIOS提供者。尽管如此，通过利用原始NetBIOS接口，我们仍然可以使用NetBIOS，就象第1章所说的那样。最后，列出的RSVP（由它提供QoS）和ATM提供者当然可用于Windows 98和Windows 2000。

Winsock API和OSI模型

现在，我们来看看本章提及的一些概念是如何与 OSI模型关联的（参见第4页上的图1-1）。Winsock目录（通过WSAEnumProtocols列举出来的）中的传输提供者位于 OSI模型的传送层。也就是说，每个传输协议都会提供一种传输数据的方法；但是，它们本身又是另一个网络协议的成员，而网络协议位于网络层，因为它是为网络上各节点提供定址方法的协议。比如，UDP和TCP就是传输协议，但两者又都属于因特网协议（ IP）。

Winsock API安装在“ 会话层 ”和“ 传送层 ”之间。 Winsock提供了一种可为指定传输协议打开、计算和关闭会话的能力。在 Windows下，上面三层（应用层、呈现层和会话层）在很大程度上与用户的Winsock应用有关。换而言之，用户的 Winsock应用控制了会话的方方面面，必要时，还会根据程序的需要格式化数据。

5.6 选择适当的协议

在开发网络应用程序时，可从现有的大量协议中选出自己需要的，用来作为应用程序的基础。如果应用程序需要依靠特定协议进行通信，选择余地就有限；但是，如果想从头开发一个应用程序， TCP/IP就是首选协议之一，至少从支持能力和微软的赞助这一角度来看，是这样的。AppleTalk、NetBIOS和IPX/SPX都囊括在微软的操作系统中，以便提供与其他操作系统的兼容性，同时，它们也是网络编程的重要元素。由此可见，在网络计算机上安装Windows 95时，默认安装的协议便是NetBEUI和IPX/SPX。

随着互联网的爆炸性增长，许多大公司、教育部门以及其他部门都把 TCP/IP选作为自己的通信协议。Windows 2000中，微软还专门强调了TCP/IP。现在，大多数网络服务都以这个协议为基础，从而降低了对 NetBIOS的依赖性。除此以外，在微软的 TCP/IP实施方案中找到错误时，马上可以得到错误修复的响应，反之，其他协议中的错误要得到修复的话，可能就不那么幸运了（也许还是），这要看具体情况。

也就是说，在选择协议时，如果想把它用于网络应用， TCP/IP是你最安全的选择。除此以外，微软一直对 ATM网络提供强大的技术支持。如果有闲开发只运行于 ATM网络的应用程序，就应该在 Winsock利用原始ATM，这样兴许比较安全。当然， TCP/IP用户应该注意到ATM网络可配置成运行于 TCP/IP仿真模式，而且在这一模式中， TCP/IP同样可以正常运行。除了开发任何应用程序都需要的协议特性之外，这些只是需要考虑到的少数因素。

5.7 小结

通过本章的学习，大家已了解为应用程序选择网络传输时应该知道的基本特性。在为指

定的协议开发成功的网络应用程序时，了解这些特性是至关重要的。我们还有计划地深入探讨了如何获得安装在系统中的传输提供者列表和如何查询特定属性。最后，我们还学习了如何为指定的传输建立套接字，方法是为 WSASocket或socket函数指定正确的参数，再利用WSAEnumProtocols查询目录条目以及通过 WSAPROTOCOL_INFO结构，把函数投递到WSASocket。下一章，我们将进一步探讨各主要协议的定址方法。

第6章　地址家族和名字解析

要通过Winsock建立通信，必须了解如何利用指定的协议为工作站定址。本章将一一说明Winsock支持的协议以及各协议如何把一个指定家族的地址解析成网络上一台具体的机器。Winsock 2引入了几个新的、与协议无关的函数，它们可和任何一个地址家族一起使用；但是大多数情况下，各协议家族都有自己的地址解析机制，要么通过一个函数，要么作为一个投给getsockopt的选项。本章只讲解各协议组成地址结构时所需的一些基本知识。第 10章讨论注册和名字解析函数，这些函数对特定协议家族服务进行声明（这和简单的名字解析稍有不同）。关于直接名字解析、服务声明与解析之间的差别，可参见第 10章。

对已讲过的地址家族来说，我们将进一步探讨如何为网络上的一台机器定址。然后，再针对各个家族建立套接字。除此以外，还要讨论协议独有的名字解析选项。

6.1　IP

网际协议（Internet Protocol, IP）是一种用于互联网的网络协议，已经广为人知。它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（比如办公室小型网络）和广域网WAN（比如说互联网）。从它的设计看来，IP是一个无连接的协议，不能保证数据投递万无一失。两个比它高级的协议（TCP和UDP）用于依赖IP协议的数据通信。

6.1.1 TCP

面向连接的通信是通过“传输控制协议”（Transmission Control Protocol, TCP）来完成的。TCP提供两台计算机之间的可靠无错的数据传输。应用程序利用 TCP进行通信时，源和目标之间会建立一个虚拟连接。这个连接一旦建立，两台计算机之间就可以把数据当作一个双向字节流进行交换。

6.1.2 UDP

无连接通信是通过“用户数据报协议”（User Datagram Protocol, UDP）来完成的。UDP不保障可靠数据的传输，但能够向若干个目标发送数据，接收发自若干个源的数据。简单地说，如果一个客户机向服务器发送数据，这一数据会立即发出，不管服务器是否已准备接收数据。如果服务器收到了客户机的数据，它不会确认收到与否。数据传输方法采用的是数据报。

TCP和UDP两者都利用 IP来进行数据传输，一般称为 TCP/IP和UDP/IP。Winsock通过AF_INET地址家族为IP通信定址，这个地址家族的定义在 Winsock 1.h和Winsock 2.h中。

6.1.3 **定址**

IP中，计算机都分配有一个 IP地址，用一个32位数来表示，正式的称呼是“ IPv4地址”。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，

服务器打算监听接入客户机请求时，也必须指定一个 IP地址和一个端口号。Winsock中，应用通过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我们此时正在使用IP地址家族。

IP协议第6版

IP协议第6版对原来的IP协议规格进行了改进，将 IP地址扩展到16个字节。随着IPv4的退场，不久的将来， IPv6的地位显得越来越重要。许多 Winsock头文件中都包含针对IPv6结构的条件定义；但是当前的 Win32平台均没有提供IPv6网络堆栈（包括 Windows 2000在内）。“ 微软研究部 ”已开发出一个试验性的 IPv6堆栈，可从 http://research.microsoft.com/mstripv6/下载；然而，该堆栈未获支持，而且我们不打算深入第 6版协议中专有的特性。

准备使用哪个TCP或UDP通信端口来标识服务器服务这一问题，则由 sin_port字段定义。在选择端口时，应用必须特别小心，因为有些可用端口号是为“ 已知的 ”（即固定的）服务保留的（比如说文件传输协议和超文本传输协议，即 FTP和HTTP）。“ 已知的协议 ”，即固定协议，采用的端口由“ 互联网编号分配认证（ IANA）”控制和分配， RFC 1700中说明编号。从本质上说，端口号分为下面这三类：“ 已知 ”端口、已注册端口、动态和（或）私用端口。

- 0~1023由IANA控制，是为固定服务保留的。
- 1024~49151是IANA列出来的、已注册的端口，供普通用户的普通用户进程或程序使用。
- 49152~65535是动态和（或）私用端口。

普通用户应用应该选择 1024~49151之间的已注册端口，从而避免端口号已被另一个应用或系统服务所用。此外， 49152~65535之间的端口可自由使用，因为 IANA这些端口上没有注册服务。在使用bind API函数时，如果一个应用和主机上的另一个应用采用的端口号绑定在一起，系统就会返回Winsock错误WSAEADDRINUSE。第7章将深入阐述Winsock绑定进程。

SOCKADDR_IN结构的sin_addr字段用于把一个IP地址保存为一个4字节的数，它是无符号长整数类型。根据这个字段的不同用法，还可表示一个本地或远程 IP地址。IP地址一般是用“ 互联网标准点分表示法 ”（像a.b.c.d一样）指定的，每个字母代表一个字节数，从左到右分配一个 4字节的无符号长整数。最后一个字段 sin_ zero，只充当填充项的职责，以使SOCKADDR_IN结构和SOCKADDR结构的长度一样。

一个有用的、名为 inet_addr的支持函数，可把一个点式 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字段是一个空中止字符串，它认可点式表示法的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”小节中简要说明）。

1. 特殊地址

对于特定情况下的套接字行为，有两个特殊 IP地址可对它们产生影响。特殊地址INADDR_ANY允许服务器应用监听主机计算机上面每个网络接口上的客户机活动。一般情况下，在该地址绑定套接字和本地接口时，网络应用才利用这个地址来监听连接。如果你有一个多址系统，这个地址就允许一个独立应用接受发自多个接口的回应。

特殊地址INADDR_BROADCAST用于在一个IP网络中发送广播UDP数据报。要使用这个特殊地址，需要应用设置套接字选项SO_BROADCAST。第9章将对该选项进行详细论述。

2. 字节排序

针对“大头”（big-endian）和“小头”（little-endian）形式的编号，不同的计算机处理器的表示方法有所不同，这由各自的设计决定。比如， Intel 86处理器上，用“小头”形式来表示多字节编号：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，“互联网联网标准”指定多字节值必须用“大头”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。

有一系列的函数可用于多字节数的转换，把它们从主机字节顺序转换成网络字节顺序，反之亦然。下面四个API函数便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的一个4字节数。htonl函数返回的数顺序是网络字节顺序，而WSAHtonl函数通过lpnetlong参数返回的数顺序是网络字节顺序。htons和WSAHtons的hostshort参数是按主机字节顺序的一个2字节数。htons函数把这个数当作按网络字节顺序的一个2字节值返回，而WSAHtons函数通过lpnetshort参数把这个数返回。

下面这四个是前面四个函数的反向函数：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，我们打算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。

```
SOCKADDR_IN InternetAddr;
INT nPortId = 5150;

InternetAddr.sin_family = AF_INET;

// Convert the proposed dotted Internet address 136.149.3.29
// to a 4-byte integer, and assign it to sin_addr
InternetAddr.sin_addr.s_addr = inet_addr("136.149.3.29");

// The nPortId variable is stored in host-byte order. Convert
// nPortId to network-byte order, and assign it to sin_port.

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

6.1.4 创建套接字

创建一个IP套接字的好处是便于应用能够通过 TCP、UDP和IP协议进行通信。如要用 TCP协议打开一个 IP套接字，需调用带有地址家族 AF_INET和套接字类型 SOCK_STREAM的socket函数或WSASocket函数，并把协议字段设成0，方式如下：

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

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

要利用UDP协议打开IP套接字，只须指定套接字类型，用这个指定的套接字类型代替socket函数中的SOCK_STREAM和上面的WSASocket调用。还可以打开一个套接字通过 IP直接通信。这是把套接字类型设成 SOCK_RAW来完成的。第13章将对SOCK_RAW选项进行深入探讨。

6.1.5 名字解析

Winsock应用打算通过IP和主机通信时，必须知道这个主机的 IP地址。依用户看来， IP地址是不容易记的。在指定机器时，许多人更愿意利用一个易记的、友好的主机名而不是 IP地址。Winsock提供了两个支持函数，它们有助于用户把一个主机名解析成 IP地址。

Windows套接字gethostbyname和WSAAsynGetHostByName API函数从主机数据库中取回与指定的主机名对应的主机信息。两个函数均返回一个 HOSTENT结构，该结构的格式如下：

```
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字段是正式的主机名。如果网络采用了“域内命名系统”(DNS)，它就是导致命名服务器返回响应的“全限定域名”(FQDN)。如果网络使用一个本地“多主机”文件，主机名就是IP地址之后的第一个条目。 h_aliases字段是一个由主机备用名组成的空中止数组。h_addrtype表示即将返回的地址家族。 h_length字段则对h_addr_list字段中的每一个地址定义字节长度进行定义。h_addr_list字段是一个由主机IP地址组成的空中止数组（可以为一个主机分配若干个IP地址)。这个数组中的每个地址都是按网络字节顺序返回的。一般情况下，应用程序都采用该数组中的第一个地址。但是，如果返回的地址不止一个，应用程序就会相应地选择一个最恰当的，而不是一直都用第一个地址。

gethostbyname API函数的定义如下：

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

name参数表示准备查找的那个主机的友好名。如果这个函数调用成功，系统就会返回一个指向HOSTENT结构的指针。注意，保存HOSTENT结构的是系统内存。应用程序不应该依靠它来维护状态。由于该内存由系统维护，因此，你的应用程序不必释放这个已返回的结构。

WSAAsyncGetHostByName API函数是gethostbyname函数的异步版，后一个函数在结束时，利用Windows消息向应用程序发出通知。 WSAAsynGetHostByName的定义如下：

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

bWnd参数是窗口句柄，异步请求结束时，这个句柄将收到一条消息。 wMsg参数是异步请求结束时收到的窗口消息。 name参数代表我们正在查找的主机之用户友好名。 buf参数是一个指针，它指向接收HOSTENT数据的那个数据域。这个缓冲区必须大于 HOSTENT结构，应该设为MAXGETHOSTSTRUCT中定义的最大长度。

另外两个用于获得主机信息的函数是： gethostbyaddr和WSAAsynGetHostByName API函数，它们是为获得与IP网络地址相应的主机信息而设计的。在有了主机 IP地址，并打算查找其用户友好名时，这两个函数非常有用。 gethostbyaddr函数的定义如下：

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

addr参数是指向一个IP地址的指针，这个地址按网络字节顺序排列。 len参数用于指定addr参数的字节长度。 type参数将指定 AF_INET值，这个值表明指定类型是 IP地址。WSAAsyncGetHostByAddr API函数是gethostbyaddr函数的异步版。

端口号

应用打算与运行于本地或远程计算机上的服务进行通信时，除了要知道远程计算机的 IP地址外，还必须知道服务的端口号。在使用 TCP和UDP时，应用必须决定计划通过哪些端口进行通信。有几个“已知的端口号”是服务器服务保留的，这些服务支持比 TCP高级的协议（比如TCP和SPX）。举个例子来说，端口 21是为FTP预留的，端口80是为FTTP预留的。正如前面提到的那样，已知的服务一般都采用 1~1023之间的端口号来设置协议。如果你正在开发一个不使用任何一种已知服务的 TCP应用，就要考虑采用 1023以上的端口，以免重复。通过调用getservbyname和WSAAsyncGetServByName函数，便可获得已知服务的端口号。这两个函数只从名为services的文件中获得静态信息。 Windows 95和Windows 98中，服务文件位于%WINDOWS%下面；在 Windows NT和Windows 2000中，则位于% WINDOWS% \System32\Drivers\Etc下面。getservbyname函数的定义如下：

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

name参数代表准备查找的服务名。举个例子来说，如果你正在定位 FTP端口，就应该把name参数设成指向字串“ ftp ”。proto参数随便指向一个字串，这个字串表明 name中的服务是在这个参数中的协议下面注册的。 WSAAsyncGetServByName函数是getservbyname函数的异步版。

Windows 2000中有一个新的注册和请求 TCP及UDP服务信息的动态方法。服务器应用可利用WSASetService函数来注册服务名、IP地址和服务的端口号。客户机应用可利用这三个 API函数的组合请求这条服务信息，这三个函数是WSALookupServiceBegin、WSALookupServiceNext和WSALookupServiceEnd。第10章将对此进行讨论。

6.2 红外线套接字

红外线套接字称为IrSock，它是一个令人兴奋不已的新技术，首先在Windows平台上亮相。红外线套接字允许两台计算机通过红外线串行端口进行通信。目前，红外线套接字可在Windows 98和Windows 2000上使用。红外线套接字不同于传统意义上针对便携机的短暂性而设计的套接字。它们展示了一种新的名字解析模型，下一小节中将对此进行讨论。

6.2.1 定址

由于带有“红外线数据联盟”（Infrared Data Association, IrDA）设备的大多数计算机可能经常性地拆卸，传统的名字解析方案不能满足人们的需求。传统解析方法充当命名服务器之类的静态资源 —— 人们在移动正在运行网络客户任务的手提式电脑或膝上型计算机时，是不能使用这些资源的。为解决这一问题， IrDA就设计成了这样：无须在整个大型网络上，只须用一种特定的方式浏览范围内的资源，因此， IrDA没有使用标准的Winsock命名服务函数和IP定址。相反，命名服务已并入通信流，另外还引入了一个新的地址家族，以支持与红外线串行端口绑定在一起的服务。 IrSock地址结构中包含一个服务名（它对绑定和连接调用中所使用的应用进行描述）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;
```

irdaAddressFamily字段一直都是AF_IRDA。irdaDeviceID是一个4字符的字串，用于唯一性地标识特定服务所运行的设备。在建立IrSock服务器时，这个字段是忽略不计的。但是对客户机而言，却非常重要，因为它指定的是准备连接的那个IrDA设备（也可能有若干个）。最后，irdaServiceName字段是服务名，应用要么利用这项服务对其本身进行注册，要么试着与这项服务建立连接。

6.2.2 名字解析

定址可以利用“IrDA逻辑服务访问点选择符（LSAP-SEL）”或“信息访问服务”（IAS）注册的服务为基础。IAS从一个LSAP-SEL中摘出一项服务，并把它置入用户友好的文本服务名中，方式和互联网域名服务器把名字映射到数字化IP地址差不多。要成功建立一个连接，既可以用LSAP-SEL，又可以用用户友好名。不过，用户友好名需要名字解析。大多数时候，都不要使用直接的LSAP-SEL“地址”，因为IrDA服务的地址空间是限制了的。Win32实施方案允许LSAP-SEL整数标识符，标识符的范围是1到127。从本质上说，我们可把IAS服务器当作一个WINS服务器，因为它把LSAP-SEL和一个文字化的服务名关联在一起。

事实上的IAS条目有三个重要字段：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。这是服务器通过相应的SOCKADDR_IRDA结构执行绑定调用时完成的。这种情况一旦发生，就会增加一个IAS条目，该条目中包括类名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是系统根据注册来分配的。另一方面，客户机向连接调用投递一个SOCKADDR_IRDA结构。随后便开始IAS查找，查找带有类名MyServer和属性IrDA:TinyTP:Lsap-SEL的那项服务。IAS查询会返回3这个值。用户可利用getsocketopt调用中的套接字选项IRLMP_IAS_QUERY来定制自己的IAS查询。

如果打算完全忽略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后，就只能和不提供任何IAS注册的老IrDA设备（比如红外线打印机）进行通信。把SOCKADDR_IRDA结构中的服务名指定为LSAP-SEL-xxx，就可忽略IAS注册和查找。其中，xxx处是属性值，其范围在1到127之间。对服务器而言，这样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）。对客户机而言，这样会忽略IAS查找，并试图马上与运行于指定的LSAP-SEL上的任何一项服务建立连接。

6.2.3 红外线设备列举

由于红外线设备的使用地点不固定，因此，必须有一种方法，可以动态地把特定范围内的所有可用红外线设备列举出来。我们先从Windows CE实施方案和Windows 98及Windows 2000实施方案之间的几点差别谈起。Windows CE先于其他平台支持IrSock，并提供少量与红外线设备有关的信息。后来，Windows 98和Windows 2000也开始支持IrSock，但它们新增了另外的“提示”信息，该信息是由列举请求返回的（关于提示信息，我们稍后将简要论述）。

这样一来，Windows CE的AF_irda.h头文件中包含的是原始的、少量的结构定义；但是，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定义。为了保持一致，我们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参数投递。这里有两个结构，一个针对 Windows 98和Windows 2000，另一个针对Windows CE。这两个结构的格式如下：

```
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;
```

Windows 98和Windows 2000结构与Windows CE结构之间的唯一区别是 Windows 98和Windows 2000结构中包含一个 WINDOWS_IRDA_DEVICE_INFO数组，该数组与 WCE_IRDA_DEVICE_INFO结构的数组相对应。条件性的 #define指令根据目标平台，声明DEVICELIST结构是正确的。同样，也有对IRDA_DEVICE_INFO结构的两个声明：

```
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;
```

#define指令根据目标平台，向正确的结构定义声明 IRDA_DEVICE_INFO。

正如前面提到的那样，真正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 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);
```

在向getsockopt调用投递一个DEVICELIST结构之前，别忘了把numDevice字段设成0。一次成功列举会把numDevice字段设成一个大于0的值，并把Device字段中IRDA_DEVICE_INFO结构数量设为前一个值。同时，在实际应用中，为检查新使用的设备，可能会多次执行getsockopt。举个例子来说，一个很好的例子就是试着进行五次或少于五次的红外线设备查找。方法很简单：在未成功列举之后，利用短期调用 Sleep，把该调用置入循环即可。

现在，大家已知道如何列举红外线设备，创建一个客户机或服务器就更简单了。同级的服务器端更为简单，因为它像一个“普通”服务器。也就是说，不需要额外的步骤。创建IrSock服务器的常见步骤如下；

1) 建立一个地址家族AF_IRDA套接字和套接字类型SOCK_STREAM。

2) 用服务器的服务名填写一个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边限调用listen。

5) 为接入客户机锁定一个accept调用。

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

1) 建立地址家族AF_IRDA套接字和套接字类型SOCK-STREAM。

2) 调用有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一个 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）中所填的结构，重复步骤4），直到连接成功。

6.2.4 查询IAS

要知道特定服务是否在特定的设备上运行，有两种方法。第一种是真正与特定服务连接；另一种是向IAS查询特定的服务名。两种方法都要求列举红外线设备，然后对每一个设备进行查询直到达到目的或所有的设备都查完。执行查找是调用带有 IRLMP_IAS_QUERY选项的getsockopt函数来完成的。再次提醒大家注意， IAS_QUERY结构有两个，一个针对Windows 98和Windows 2000，另一个针对Windows CE。各结构的格式如下：

```
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;
```

大家可看到，除了特定字符数组的长度不同之外，这两个结构的格式是差不多的。

要对特定服务的LSAP-SEL数有多少进行查询，很简单：把irdaClassName字段设为LSAP-SEL的属性字串，即IrDA:IrLMP:LsapSel，然后，把irdaAttributeName字段设成准备查询的那个服务名。除此以外，还必须用范围内的有效设备来设置 irdaDeviceID字段。

6.2.5 创建套接字

红外线套接字的创建很简单。几乎不需要任何选项，这是因为 IrSock只支持面向连接的数据流。下面的代码说明了如何利用 socket或WSASocket调用来建立红外线套接字。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);
```

如果不想因循守旧，可把IRDA_PROTO_SICK_STREAM当作上面任何一个函数的协议参数投递出去。但系统不需要这个协议参数，因为传输目录中只有一个地址家族 AF_IRDA条目。

指定AF_IRDA会造成默认使用这个传输条目。

6.2.6 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，不过只限于地址家族 AF_IRDA套接字上。我们将在第9章全面论述这些选项，第9章还对所有套接字选项及其参数进行了总结。

6.3 IPX/SPX

“互联网包交换”（IPX）协议是一个常见协议，一般为承担 Novell NetWare客户机／服务器联网服务的计算机所用。IPX提供两个进程间的无连接通信；因此，如果一个工作站发出一个数据包，该协议无法保证这个数据包会准确无误地投递到目标地点。如果应用程序需要数据投递保证，但仍坚持使用IPX，它就会选用一个比IPX高级的协议，比如说“顺序分组交换”（SPX）和SPX II协议，这两个协议中， SPX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（它们是Windows 95、Windows 98、Windows NT以及Windows 2000），但没有提供Windows CE平台上通过IPX通信的能力。

6.3.1 编址

IPX网络、网段是用IPX路由器桥接在一起的。每个网段分配 4字节的唯一地址号。当更多的网段桥接在一起时， IPX路由器管理网段之间的通信，每个网段有唯一的网段号。连网时，使用唯一的6字节网段号，这个号也往往是转接器的物理地址。一个节点（也就是一台计算机）一般有一个或多个通信进程用 IPX通信。IPX用套接字号来区分一个节点上的通信。

要用IPX进行Winsock客户机或服务器通信，必须建立 SOCKADDR_IPX结构。该结构在Wsipx.h头文件中定义，应用程序在包括Winsock 2.h文件之后还必须包括该文件。SOCKADDR_IPX结构如下定义：

```
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字段应设为AF_IPX值，sa_netnum字段是4字节的地址，代表IPX网络上网段号，sa_nodenum字段是6字节的地址，代表节点计算机的物理地址， sa_socket字段代表一个节点区分IPX通信的套接字或接口。

6.3.2 创建套接字

利用IPX创建套接字提供了几种可能性。要打开 IPX套接字，调用带有地址家族 AF_IPX、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，第三个参数协议是必须指定的，而且不能为 0。这一点相当重要，因为该字段还可用于设置特定IPX包的类型。

正如我们前面提到的那样， IPX利用数据报提供不可靠的无连接通信。如果一个应用需要可靠的无连接通信，可采用比 IPX高级的协议，比说 SPX和SPX II。要做到这一点，把socket和WSASocket调用的类型和协议字段分别设置成套接字类型 SOCK_SEQPACKET或SOCK_STREAM和协议字段NSPROTO_SPX或NSPROTO_SPX II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，如果发送端发出 2000个字节，在这2000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，这是通过设置 SPX头中的消息结束位来完成的。指定 SOCK_ STREAM时，要注意这个位，Winsock recv和WSARecv调用在其收到这个位之前不会中止。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， recv就会中止，不管消息结束位设在哪里。从发送端这一方来说（使用 SOCK_SEQPACKET类型），如果发送的数据包小于一个完整的数据包，消息结束位就会随这个包一起发送。如果发送的包大于一个完整的数据包，消息结束位就只设在最后发送的那个包中，而不是每个包都有。

1. 绑定套接字

IPX应用通过bind把本地地址和套接字关联在一起时，不要在 SOCKADDR_IPX结构中指定网络号和节点地址。 bind函数利用系统上可用的第一个 IPX网络接口来填充这些字段。如果一台计算机有多个网络接口（多址计算机），它就不必绑定特定的接口。 Windows 95、Windows 98、Windows NT和Windows 2000提供一个虚拟内部网，不管它直接连接的物理网络如何，该虚拟网中的每个接口都是可以抵达的。我们稍后将对内部网络编号进行详细论述。应用成功绑定本地接口之后，便可利用下属代码段中的 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((unsigned) sdServer, (PSOCKADDR) &IPXAddr, &addrlen)
    == SOCKET_ERROR)
{
    printf("getsockname failed with error %d",
        WSAGetLastError());
    return;
}

// Print out SOCKADDR_IPX information returned from
// getsockname()
```

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中的网络段和IPX报在网络段之间的路由选择。Windows 95、Windows 98、Windows NT以及Windows 2000平台特别突出了内部网络编号，这个内部网络编号用于内部路由选择和对网间网（几个网络桥接在一起构成）上的计算机进行唯一性标识。内部网络编号也称为“虚拟网号”，即内部网络编号对网间网中的另一个（虚拟的）网络段进行标识。因此，如果一台计算机正在运行 Windows 95、Windows 98、Windows NT或Windows 2000，若打算为它配置一个内部网络编号， 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另增一次跳跃。

在多址计算机情形下，内部虚拟网络用于某一特定目的。应用程序绑定本地网络接口时，不应该指定本地接口信息，但需要把SOCKADDR_IPX结构的sa_netnum和sa_nodenum这两个字段设置为0。这是因为IPX能够通过内部虚拟网络的使用，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。比如，即使你的应用程序明显地绑定网络A上的网络接口，而数据包却来自网络B，内部网络编号就会使这个数据包在内部路由，这样，你的应用程序就收到这个数据包了。

3. 通过Winsock，设置IPX数据包的类型

在利用NSPROTO_IPX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：路由信息协议（RIP）包。
- 04h：服务声明协议（SAP）包。
- 05h：顺序分组交换（SPX）包。
- 11h：NetWare Core协议（NCP）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。比如，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. 名字解析

大家可能知道， Winsock中的IPX定址有些麻烦，因为必须提供构成地址的多字节的网络和节点编号。IPX使应用程序可以通过用户友好名找到相应服务，以此通过 SAP协议获得IPX网络中的网络编号、节点编号和端口编号。正如我们将在第 10章看到的那样， Winsock 2利用WSASetService API函数，提供了一个与协议无关的名字解析法。通过 SAP协议， IPX服务器应用可利用WSAAetService在用户友好名下注册它们正在监听的网络编号、节点编号和端口

编号。Winsock 2还通过下面几个API函数提供了另一个与协议无关的名字解析法，它们是：WSALookupServiceBegin、WSAlookupServiceNext和WSAlookupServiceEnd。

要执行自己的命名服务注册和查找，可以通过打开一个 IPX套接字并指定SAP类型的方法来完成。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。这要求大家深入了解SAP协议和掌握针对IPX SAP包解码的编程方面的基本知识。

6.4 NetBIOS

NetBIOS地址家族也是另一种可从 Winsock访问的协议家族。通过第 1章的NetBIOS讨论，大家将进一步熟悉这里提到的许多主题和注意事项。从 Winsock开始的NetBIOS定址仍需要得知NetBIOS名和LANA编号。我们假定大家已看过第 1章中的这些小节，然后，继续深入通过Winsock访问NetBIOS的特性。

注意 NetBIOS地址家族由Winsock剖析，只能用于Windows NT和Windows 2000。不能用于Windows 95、Windows 98和Windows CE。

6.4.1 定址

NetBIOS下机器的定址基础是 NetBIOS名，这个我们已在第 1章中讲过。NetBIOS名有16个字符长，最后一个字符留给定义这个名属于哪类服务的限定字符用。 NetBIOS名有两种类型：专有名和组名。专有名可只由整个网络上的一个进程注册。比如，一个基于会话的服务器将注册FOO名，而打算和该服务器沟通的客户机则试着和 FOO连接。组名允许一组应用注册同一个名字，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字的数据报。

Winsock中，NetBIOS定址结构是在Wsanetbs.h中定义的，格式如下：

```
#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家族，它应该始终为 AF_NETBIOS。snb_type字段用于指定一个专有名或组名。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字段就是事实上的NetBIOS名。

现在，大家已知道各字段的含义以及应该怎样设置它们了，下面设置的宏定义于头文件中，方便易行：

```
#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是此时正在填的SOCKADDR_NB结构的地址。正如大家所见的那样，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。这个宏认定其长度至少是 NETBIOS_NAME_LENGTH减1，如果短于这一长度，它就会空中止。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。

大家可看到，Winsock中NetBIOS名的结构是直接了当的，没有任何令人费解之处。名字解析是在悄然执行的，因此，它不像 TCP和IrDA那样，在执行操作之前，不必把名字解析成物理地址。在考虑到依赖于多个协议实施 NetBIOS，但各个协议都有自己的一套定址方案时，这一点更为明显。下一章中，我们将通过在 Winsock中使用NetBIOS接口，向大家展示一个简单的客户机／服务器范例。

6.4.2 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，必须知道哪些LANA编号和你的应用有关。必须记住， NetBIOS客户机和服务器要进行通信，必须有一个常用的传输协议，这样它们便可通过该协议进行监听或连接。创建 NetBIOS套接字有两种方法。其一是像下面这样调用 socket或WSASocket：

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

根据你需要的是一个无连接数据报，还是面向连接的会话套接字， WSASocket的type参数分别是SOCK_DGRAM和SOCK_SEQPACKET（不能两个同时指定）。第三个参数protocol，除非你必须将其取消，否则就是准备依据它来创建套接字的那个 LANA编号。第四个参数是空值，因为你此刻正在指定自己的参数，而不是正在使用 WSAPROTOCOL_INFO结构。第五个参数没有用。最后，把dwFlags参数设为WSA_FLAG_OVERLAPPED；根据所有WSASocket调用，指定WSA_FLAG_OVERLAPPED。

第一种套接字创建法的不足之处是必须知道从哪一个有效 LANA编号着手。不幸的是，Winsock目前没有妙方把所有有效LANA编号列举出来。备用Winsock可利用WSAEnumProtocols把所有传输协议列举出来。当然，也可利用 NCBENUM命令调用Netbios，从而获得有效的LANA编号。第5章对如何调用 WSAEnumProtocols进行了说明。下面的实例列举了所有的传输协议，搜查到一个NetBIOS传输，并针对各个协议分别建立了套接字。

```
dwNum = WSAEnumProtocols(NULL, lpProtocolBuf, &dwBufLen);
if (dwNum == SOCKET_ERROR)
{
```

```
        // Error
    }
    for (i = 0; i < dwNum; i++)
    {
        // Look for those entries in the AF_NETBIOS address family
        if (lpProtocolBuf[i].iAddressFamily == AF_NETBIOS)
        {
            // Look for either SOCK_SEQPACKET or 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地址家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，我们就有一个可以使用的NetBIOS传输协议了。如果还需要一个LANA编号，就选用WSAPROTOCOLS_INFO结构中iProtocol字段的绝对值。这个LANA的iProtocol字段是0x80000000，因为0是Winsock为特殊用途保留的。变量j将包含一个值，表示有效传输协议有多少个。

6.5 AppleTalk

Winsock中，对AppleTalk的支持已发布一段时间了，但鲜为人知。如果不与苹果公司的MAC机通信，你可能不会选择AppleTalk协议。从某种程度上来说，AppleTalk与NetBIOS类似，因为两者都是针对每一个进程来进行名字注册的。也就是说，要使特定的名字广为人知，服务器必须对这个名字进行注册。客户机再用这个名字与服务器建立连接。究其本质，AppleTalk名比NetBIOS名更为复杂。下一小节，我们将讨论如何为网络上使用 AppleTalk协议的计算机进行定址。

6.5.1 定址

AppleTalk名实际上是以三个独立的名字为基础的：名、类型和区。每个名字的长度可达32个字符。这个名字标识机器上的进程及其关联套接字。类型是区的子群机制。传统意义上，区是一个网络，它是由物理定位于同一个循环上、使用 AppleTalk协议的计算机构成的。微软的AppleTalk实施方案允许Windows兼容机对自己定位的默认区进行指定。多个网络可通过桥接联在一起。这些易记的名字分别映射一个套接字编号、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。这项要求由“名字绑定协议”（NBP）来执行，该协议将一次查询在网上广播，以便知道这个名字是否已使用。与此同时，AppleTalk利用“路由表维护协议”（RTMP），动态地对链接在一起的各个 AppleTalk网络的路由进行查找。

下一个结构提供了从Winsock为AppleTalk主机定址的基础：

```
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结构被投入bind、connect和WSAconnect之类的Winsock调用中，但如果要转换易于理解的名字，必须要求网络系统先对这个名字进行解析或注册。这是分别通过 getsockopt或setsockopt调用来完成的。

6.5.2 AppleTalk名的注册

对一个服务器而言，如果它打算注册特定名字，以便客户机可以很容易地与之建立连接，就要利用SO_REGISTER_NAME选项来调用setsockopt函数。牵涉到AplpleTalk名的所有套接字选项，都要使用WSH_NBP_NAME结构，它的格式如下：

```
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（比如说 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，看你是查找名字呢，还是注册名字或删除名字。

下面的代码实例解释了如何注册 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;
//
// Fill in the name to register
//
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)
{
```

```
    // Error
}
ataddr.sat_socket = 0;
ataddr.sat_family = AF_APPLETALK;
if (bind(s, (SOCKADDR *)&ataddr, sizeof(ataddr)) == SOCKET_ERROR)
{
    // Unable to open an endpoint on the AppleTalk network
}
if (setsockopt(s, SOL_APPLETALK, SO_REGISTER_NAME,
               (char *)&atname, sizeof(WSH_NBP_NAME)) == SOCKET_ERROR)
{
    // Name registration failed!
}
```

大家首先注意到的是 MY_ZONE、MY_TYPE和MY_OBJECT这三个字串。记住，AppleTalk名是三级式的。注意，区是一个星号“*”。这是区字段中所用的特殊字符，用来指定计算机处在“当前”区。接下来，我们建立一个隶属于AppleTalk协议ADSP的SOCK_STREAM类型的套接字。这个套接字建立之后，有一个利用一个地址结构进行的 bind调用，这个地址结构中有一个置零sat_socket字段和唯一设置的协议家族字段。这是非常重要的，因为它为发出请求的用户应用在网络上建立了一个端点。注意，虽然调用 bind允许你在网络上执行简单操作，但它本身不允许你的应用接受客户机发出的接入连接请求。要接受客户机连接，必须在网络上注册你自己的名字，这是下一步的工作。

注册AppleTalk名很简单。把SOL_APPLETALK当作“level”参数，SO_REGISTER_NAME当作optname参数投递出去，便可调用 setsockopt。后两个参数是一个指针，它指向我们的WSH_REGISTER_NAME结构及其长度。如果调用setsockopt成功，我们的服务器名便得以成功注册。如果调用失败，所请求的名字大概已为他人所用。返回的 Winsock错误是WSAEADDRINUSE（10048或0x02740h）。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字，以便客户机可向它发送数据报或与之建立连接。

6.5.3 AppleTalk名的解析

同等的客户机这一端，应用通常通过友好名来得知服务器，而且必须把这个友好名解析成Winsock调用所用的网络、节点和套接字编号。这是通过 SO_LOOKUP_ NAME选项调用getsockopt函数来完成的。执行AppleTalk名的查找依赖于WSA_LOOKUP_NAME结构。这个结构及其相关结构的格式如下：

```
typedef struct
{
    WSH_ATALK_ADDRESS   Address;
    USHORT              Enumerator;
    WSH_NBP_NAME        NbpName;
} WSH_NBP_TUPLE, *PWSH_NBP_TUPLE;

typedef struct _WSH_LOOKUP_NAME
{
    //  Array of NoTuple WSH_NBP_TUPLEs
    WSH_NBP_TUPLE       LookupTuple;
    ULONG               NoTuples;
} WSH_LOOKUP_NAME, *PWSH_LOOKUP_NAME;
```

在利用SO_LOOKUP_NAME选项调用getsockopt时，我们把一个缓冲造型当作 WSH_LOOKUP_NAME结构投递出去，并在第一个LookupTuple成员内填写WSH_NBP_NAME。调用成功后，getsockopt返回一个WSH_NBP_TUPLE元素组成的数组，其中包含那个 AppleTalk名的物理地址信息。程序清单 6-1中包含了Atalknm,c文件，该文件对如何查找 AppleTalk名进行了解释。除此以外，程序清单6-1还展示了如何列出所有的“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选项SO_LOOKUP _ZONES和SO_LOOKUP_MYZONE来获得。

程序清单6-1 AppleTalk名和区的查找

```
#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':        // Specify a zone name
                    if (strlen(argv[i]) > 3)
                        strncpy(szZone, &argv[i][3], MAX_ENTITY);
                    bFindName = TRUE;
                    break;
                case 't':        // Specify a type name
                    if (strlen(argv[i]) > 3)
                        strncpy(szType, &argv[i][3], MAX_ENTITY);
                    bFindName = TRUE;
                    break;
                case 'o':        // Specify an object name
                    if (strlen(argv[i]) > 3)
                        strncpy(szObject, &argv[i][3], MAX_ENTITY);
                    bFindName = TRUE;
                    break;
                case 'l':        // List zones information
                    if (strlen(argv[i]) == 3)
                        // List all zones
                        if (tolower(argv[i][2]) == 'z')
                            bListZones = TRUE;
                        // List my zone
                        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;

    // Load the Winsock library
    //
    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)
{
    // Fill in the name to look up
    //
    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);
}
// Create the AppleTalk socket
//
s = socket(AF_APPLETALK, SOCK_STREAM, ATPROTO_ADSP);
if (s == INVALID_SOCKET)
{
    printf("socket() failed: %d\n", WSAGetLastError());
    return 1;
}
// We need to bind in order to create an endpoint on the
// AppleTalk network to make our query from
//
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);
    //
    // Our character buffer now contains an array of
    // WSH_NBP_TUPLE structures after our WSH_LOOKUP_NAME
    // structure
    //
    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)
{
    // It is very important to pass a sufficiently big buffer
    // for this option. Windows NT 4 SP3 blue screens if it
    // is too small.
    //
    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);
    //
    // The character buffer contains a list of null-separated
    //  strings after the WSH_LOOKUP_ZONES structure
    //
    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)
{
    // This option returns a simple string
    //
         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;
}
```

在使用许多AppleTalk套接字选项时——比如SO_LOOKUP_MYZONE、SO_LOOKUP_ZONES以及SO_LOOKUP_NAME——需要为getsockopt调用提供更大的字符缓冲。如果调用要求你提供一个结构的选项，这个结构必须在所供字符缓冲之首。如果调用 getsockopt成功，这个函数就把返回的数据放在所供结构之后的字符缓冲中。我们来看看程序清单 6-1中的SO_LOOKUP_NAME这一部分。变量cLookupBuffer是一个用于getsockopt调用中的简单字符数组。首先，我们把它造型成 PWSH_LOOKUP_NAME，并在里面填入想查找的名字信息。然后，把这个缓冲投入 getsockopt，再根据返回的结果，增加字符指针 pTupleBuffer，这样，令其指向WSH_LOOKUP_NAME结构之后的那个字符。接下来，再把这个字符指针造型成一个PWSH_NBP_TUPLE WSH变量，因为查找AppleTalk名调用返回的数据是一个WSH_NBP_TUPLE结构组成的数组。一旦有了正确的起始位置和字元组类型，就可大功告成了。关于AppleTalk地址家族特有的各种套接字选项的详细资料，请参考第 9章。

6.5.4 创建套接字

AppleTalk可用于Winsock 1.1及稍后的版本，因此可任选套接字创建例程。再次提醒大家注意，指定基层AppleTalk协议的方式有两种。其一，可为自己需要的协议提供 Atalkwh.h中的相应定义。其二，利用 WSAEnumProtocols，然后投递WSAPROTOCOL_INFO结构，便可列举协议了。至于直接用 socket或WSASocket创建套接字时各AppleTalk协议所需的参数，均可参见表6-1。

表6-1 AppleTalk协议和参数

协 议	地址家族	套接字类型	协议类型
MSAFD AppleTalk[ADSP]	*AF_APPLETALK*	*SOCK_RDM*	*ATPROTO_ADSP*
MSAFD AppleTalk [ADSP][Pseudo-Stream]		*SOCK_STREAM*	*ATPROTO_ADSP*
MSAFD AppleTalk[PAP]		*SOCK_RDM*	*ATPROTO_PAP*
MSAFD AppleTalk[RTMP]		*SOCK_DGRAM*	*DDPPROTO_RTMP*
MSAFD AppleTalk [ZIP]		*SOCK_DGRAM*	*DDPPROTO_ZIP*

6.6 ATM

异步传输模式（ATM）协议是目前已有的最新协议之一， Windows 98和Windows 2000平台上的Winsock 2均支持它。ATM通常用于LAN和WAN上的高速联网，也用于各种类型的通信，比如说要求高速通信的语音、视频和数据等。一般说来， ATM利用网络上的虚拟连接（VC）来提供服务质量（QOS）保证。正如大家即将看到的那样， Winsock能够通过ATM地址家族来使用ATM网络上的虚拟连接。ATM网络（如图6-1所示）一般由通过交换机（它们将ATM网络桥接在一起）连接的端点（或计算机）构成。

针对ATM协议编程时，需要明白这几点。首先， ATM是一个媒体类型，而不是一个真正的协议。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。它是一个面向连接的协议，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，发送应用则可能溢出本地缓冲。同样地，接收应用必须频繁投递收到的数据：否则，接收缓冲填满之时，任何一个另外接入的数据都可能被丢弃。如果你的应用需要流控制，方法之一是在ATM上使用IP协议（它只是运行于ATM网络上的IP协议）。这样一来，应用便紧跟在上面描述的IP地址家族之后。当然，ATM的确提供了比IP好的一些好处，比如说“根式多播方案”（第12章将对此进行说明）；然而，要根据自己的应用需要来决定最适合你的那种协议。

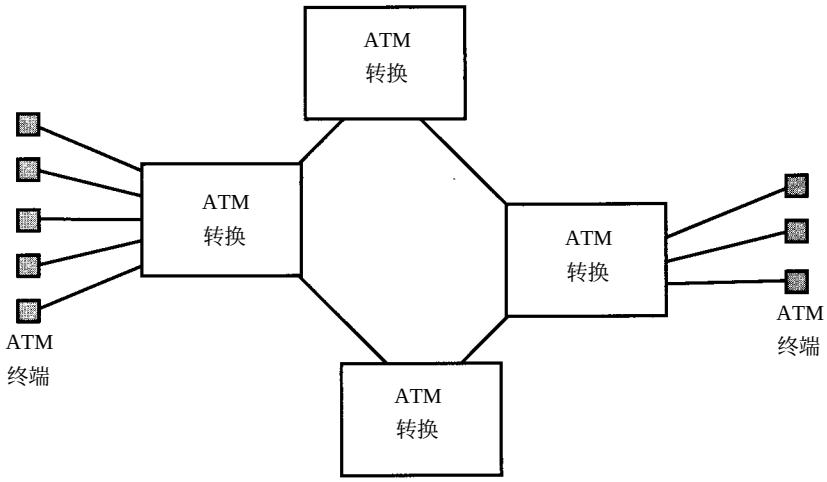

图6-1　ATM网络

6.6.1　定址

一个ATM网络有两个网络接口：用户网络接口（UNI）和网络节点接口（NNI）。UNI接口是在终端和ATM交换机之间建立的，而NNI接口则是在两个交换机之间建立的。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　允许终端在一个终端和一个ATM交换机之间发送设置和控制信息，从而在ATM网络上建立通信。注意，这个协议只限于一个终端和一个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。

若想通过Winsock设置ATM连接，那么我们只讨论UNI信号协议中的特定信息元素。目前，Windows 2000和Windows 98（SP1）上的Winsock可支持UNI信号协议3.1版本。

Winsock允许客户机／服务器通过设置“服务访问点”（SAP）应用在ATM网络上进行通信。这是利用ATM UNI信号协议设置“服务访问点”之后形成的连接来完成的。ATM是一个面向连接的协议，要求端点为进行通信，在ATM网络上建立虚拟连接。对ATM网络上通信用的套接字接口而言，SAP只是允许Winsock应用通过SOCKADDR_ATM地址结构对它进行注册和标识。SAP一旦建立，Winsock就利用UNI信号协议，向ATM网络发出调用，通过这种方式，便用这个SAP在ATM网络上的Winsock客户机和服务器之间建立一个虚拟连接。SOCKADDR_ATM结构的格式如下：

```
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应该一直为AF_ATM。satm_number字段利用其中一个基本ATM定址方案——E.164和“网络服务访问点”(NSAP)，将事实上的ATM地址表示成一个ATM_ADDRESS结构。NSAP也表示“NASP式的ATM终端系统地址”(AESA)。ATM_ADDRESS结构的格式如下：

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

AddressType字段定义特定的定址方案。如果是E.164定址方案，这个字段就是ATM_E164；若是NSAP式的定址方案，该字段就是 ATM_NSAP。除此以外，在应用打算把套接字和一个SAP绑定在一起时，还可将AddressType字段设为表6-2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字段表示事实上的ATM 20字节的E.164或NASP地址。

SOCKADDR_ATM结构的satm_blli和satm_bhli这两个字段分别代表ATM信号协议 中的“宽带基层信息”(BLLI)和“宽带高层信息”(BLLI)。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 BLLI和BHIL值已知组合的说明参见 ATM Form／IETF(互联网工程任务组)文档(这些值的特定组合用于标识 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)。这些结构中的字段值都列在ATM UNI 3.1标准一书中。ATM Form／IETF文档都可在网上地址http://www.ietf.org找到。

表6-2 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，带有通配的八个选择符。在绑定套接字和SAP时使用
SAP_FIELD_ANY_AESA_REST	NSAP式的终端系统地址，不包括通配的八个选择符，但其他的所有八个字符都有。在绑定套接字和SAP时使用

BHLI和BLLI数据结构的格式如下：

```
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;
```

关于这些字段的定义和用法，不在本书讨论之列。如果一个应用只想在 ATM网络上建立Winsock通信，就应该把BHLI和BLLI结构中的下列字段设为SAP_FIELD_ABSENT值：

- ATM_BLLI —— 第二层协议。
- ATM_BLLI —— 第三层协议。
- ATM_BHLI —— 高层信息类型（HighLayerInfoType）。

在上述字段被设为这个值时，不再使用这两个结构中的其他字段。下面的伪代码演示了一个应用如何用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地址一般表示成一个十六进制的 ASCII字符串，由 40个字符组成，与组成ATM_ADDRESS结构NSAP式或E.164地址的20个字节相对应。比如， ATM_NSAP式的地址可能像这样：

```
47000580FFE1000000F21A1D540000D10FED5800
```

把这个字串转换成一个20字节的地址相当麻烦。然而， Winsock提供了一个与协议无关的API函数， WSAStringToAddress，利用它，便可把一个40个字符的ATM十六进制ASCII字串转换成一个ATM_ADDRESS结构。我们将在本章最后着重讲一讲这个 API函数。把一个十六进制ASCII字串转换成十六进制（二进制）格式的另一种方法是利用程序清单 6-2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，大家可在第 7章中的实例中看到它。

程序清单6-2　用于转换ATM十六进制字串的函数

```
//
// Function: AtoH
//
// Description: This function coverts the ATM
// address specified in string (ASCII) format to
// binary (hexadecimal) format
//
void AtoH(CHAR *szDest, CHAR *szSource, INT iCount)
{
    while (iCount--)
    {
        *szDest++ = ( BtoH ( *szSource++ ) << 4 )
```

```
                + BtoH ( *szSource++ );
        }
        return;
    }
    //
    // Function: BtoH
    //
    // Description: This function returns the equivalent
    // binary value for an individual character specified
    // in ASCII format
    //
    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 );
        }
        //
        // Illegal values in the address will not be
        // accepted
        //
        return -1;
    }
```

6.6.2 创建套接字

ATM中，应用只能建立面向连接的套接字，因为 ATM只允许通过虚拟连接进行的通信。因此，数据的传输的形式是字节流或采用面向消息的形式。要利用 ATM协议打开一个套接字，先通过地址家族AF_ATM和套接字类型SOCK_RAW调用socket函数或WSASocket函数，然后再把协议字段设为ATMPROTO_AAL5即可。比如：

```
s = socket(AF_ATM, SOCK_RAW, ATMPROTO_AAL5);

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

默认状态下，打开套接字（比如上面这个例子）建立一个面向流的 ATM套接字。Windows还有一个富有特色的ATM提供者，可执行面向消息的数据传输。使用这个面向消息的提供者要求用户为WSAS函数显式指定原始ATM协议提供者，这是利用WSAPROTOCOL_INFO结构完成的（关于这一结构，我们曾在第5章详细讲过）。这一点是必须的，因为socket和WSASocket这两个调用中的三个元素（地址家族、套接字类型和协议）均和 Winsock中可以使用的每一个

ATM协议提供者相符。默认状态下， Winsock会返回与这三个属性相符的协议条目，并把这个协议条目标注成默认设置（在面向流的提供者情形下如此）。下面的伪代码将演示如何获得ATM面向消息协议提供者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);
    }
}
```

6.6.3 把套接字和SAP绑定在一起

事实上， ATM地址是相当复杂的，因为它们由 20个字节组成，其中包含许多信息元素。除了最后一个字节外，这些元素的所有其他字节， Winsock程序员均不用下功夫去记。 NSAP式地址和E.164地址中的最后一个字节均代表一个特定的选择符值，该值唯一允许应用定义和指定端点上特定的SAP。正如我们前面指出的那样， Winsock利用SAP建立ATM网络上的通信。

Winsock应用打算在ATM网络上进行通信时，服务器应用必须在一个端点上注册一个 SAP，并等待客户机应用根据已注册的 SAP接入。对客户机应用来说，这只包括通过 ATM_E164或ATM_NSAP地址类型设置SOCKADDR_ATM结构和提供与服务器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，应用必须先为指定的 AF_ATM地址家族建立一个套接字。这个套接字一旦建立，应用就必须像表 6-2中定义的那样，利用 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，应用一旦调用 Winsock的bind API函数（我们将在第7章对该函数进行讲述），就会建立一个SAP，这些地址类型对 Winsock在端点上建立SAP的方式进行了定义。

SAP_FIELD_ANY_AESA_SEL地址类型要求Winsock建立一个能对任何一种ATMWinsock连接进行监听的SAP，这就是通常所说的通配 ATM地址和选择符。这意味着监听连接的这个端点只能绑定一个套接字 —— 另一个套接字打算与这个地址类型绑定在一起，就会失败，并出现这样的Winsock错误WSAEADDRINUSE。然而，你也可以将另一个套接字和指定选择符上的端点显式绑定在一起。而显式和端点上的指定选择符绑定在一起的 SAP，则可用地址类型SAP_FIELD_ANY_AESA_REST来建立。这就是人们常说的只有 ATM地址，而没有选择符的“ 通配 ”。对端点上的一个指定选择符而言，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结构中指定一个均由零组成的 ATM地址。如果使用的是 SAP_FIELD_ANY_AESA_REST，就应该把这个 ATM地址的前19个字节指定为0，最后一个字节应该是准

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，如果一个连接接入套接字显式监听的指定端点和选择符，这个套接字就会获得连接）。只有在无显式绑定套接字可用的情况下，才会用通配选择符套接字来获得连接。第7章将进一步说明如何设置一个套接字，使其监听 SAP上的连接。

最后，可通过一个名为Atmadm.exe实用程序获得所有的ATM地址和端点上虚拟连接信息。在开发ATM应用程序和需要了解端点上哪些接口可用时，这个实例程序相当有用。下面表 6-3 中列出的命令行选项都可用。

表6-3 命令行选项

参数	说 明
-c	列出所有的连接（指的是虚伪连接， VC）。列出远程地址和本地接口
-a	列出所有已注册的地址（比如说所有的本地 ATM接口及其地址）
-s	打印特性（当前调用次数，收到或发出的传信和 ILMI包数，等等）

6.6.4 名字解析

目前，尚且没有命名提供者可用于 Winsock下的ATM协议。因此，不幸的便是要求应用对这个长达20个字节的 ATM地址进行指定，以便建立 ATM网络上的套接字通信。第 10章将讨论 Windows 2000域名空间（一般用于对带有友好服务名的 ATM地址进行注册）。

6.7 Winsock 2支持的其他函数

Winsock 2提供了两个有用的支持函数，它们是WSAAddressToString和WSAStringToAddress。这两个函数提供了一个与协议无关的转换方法，可以把 SOCKADDR结构转换成一个格式化的字符串，反之亦然。由于这两个函数都是与协议无关的，所以它们要求传输协议能支持字串转换。目前，它们只能用于 AF_INET和AF_ATM这两个地址家族。 WSAAddressToString函数的定义如下：

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

lpsaAddress参数代表特定协议（其中包括即将转换成字串的那个地址）的SOCKADDR结构。这个SOCKADDR结构的长度则由dwAddressLength参数指定。不同的协议，其长度也不相同。lpProtocolInfo参数是可选的，表示协议提供者。协议提供者可通过WSAEnumProtocols API函数获得，参见第 5章。如果指定了 NULL，对这个函数的调用就会采用第一个协议（它支持lpsaAddress中指定的协议家族）的提供者。lpszAddressString参数是一个缓冲，它收到的是易于理解的地址字串。lpdwAddressStringLength参数代表lpszAddressString的长度。根据结果，它返

回实际复制到lpszAddressString中的字串长度。如果提供的缓冲不够大，这个函数调用就会失败，并出现错误WSAEFAULT，而lpdwAddressStringLength参数也会根据所需要的字节长度而更新。

相反地，WSAStringToAddress API函数采用易于理解的地址串，并将它转换成一个SOCKADDR结构。WSAStringToAddress的定义如下：

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

AddressString参数是一个易于理解的地址串。表6-4对该字串的格式进行了说明。

表6-4 地址串的格式

地址家族	字串格式
IP	XXX.XXX.XXX.XXX:Y —— X代表IP地址串中的一组八字符，Y则表示端口号
ATM	NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN —— 40个N表示一个以十六进制表示的、由20个字节组成的ATM地址

Addressfamily参数代表AddressString参数的地址家族类型。lpProtocolInfo参数是可选的，代表一个协议提供者。如果把这个参数设为NULL，Winsock就会在第一个可用的协议提供者中进行搜索，查找Addressfamily参数所指定地址家族类型。如果想选定某个特定的协议提供者，WSAEnumProtocol API函数就可为你提供一个列表，已安装在系统中、可用的协议提供者都列在上面。Address缓冲参数选择的是收到信息的、地址串中的SOCKADDR结构。lpAddressLength参数代表所生成的SOCKADDR结构的长度。

6.8 小结

这一章论述了Winsock支持的协议地址家族，说明了各个家族特有的定址属性。针对每个地址家族，我们还讨论了如何创建套接字和如何设置套接字地址结构，以便开始通过协议进行通信。下一章，我们将描述适用于Winsock的基本通信技术，并把它们应用到本章讨论的所有地址家族上。

第7章 Winsock基础

本章专门讲解编写成功网络应用程序时所需的基本知识和 API调用。通过上一章的学习，大家已知道从 Winsock地址机和这些机器上的服务，可以很容易地访问协议。在这一章里，我们打算讨论如何从网络上的一台机器到另一台机器建立连接，以及如何收发数据。为使问题简单明了和免于重复，本章的讨论限定在 TCP/IP协议的范围之内。但是，针对第 6章中讲解的各个协议，本书配套光盘中包括了相应的客户机／服务器实例。需由具体协议决定的唯一一种操作是套接字的建立。而另一方面，建立连接和收发数据所需的其余大多数 Winsock调用都与基层的协议无关。而对某些例外的情况，在第 6章中，已在讨论各种协议时特地指出。

对于接受连接、建立连接和收发数据所需的 Winsock调用，本章展示的例子有助于大家对它们的理解。由于本章的目的是学习这些 Winsock调用，因而我们举的例子均采用了直接的成块Winsock调用。第8章则会展示Winsock可用的各种I/O模型，同时包括了示范代码。

除此以外，我们还打算在本章展示各种 API函数的Winsock 1和Winsock 2版本。可通过前缀WSA来把这两个版本的函数区分开。若 Winsock 2在其规格中更新或增添了一个新的 API函数，该函数名就会采用 WSA作为前缀。比如，建立套接字的 Winsock 1函数只简单称为socket，而Winsock 2引入了该函数的新版本，名为 WSASocket，它可以使用 Winsock 2中出现的某些新特性。

7.1 Winsock的初始化

每个Winsock应用都必须加载Winsock DLL的相应版本。如果调用 Winsock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。高位字节指定所需要的Winsock库的副版本，而低位字节则是主版本。然后，可用宏 MAKEWORD(X,Y)（其中，x是高位字节，y是低位字节）方便地获得 wVersionRequested的正确值。

lpWSAData参数是指向LPWSADATA结构的指针，WSAStartup用其加载的库版本有关的信息填在这个结构中：

```
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, FAR * LPWSADATA;
```

WSAStartup把第一个字段wVersion设成打算使用的Winsock版本。 wHighVersion参数容纳的是现有的Winsock库的最高版本。记住，这两个字段中，高位字节代表的是 Winsock副版本，而低位字段代表的则是Winsock主版本。szDescription和szSystemStatus这两个字段由特定的Winsock实施方案设定，事实上没有用。不要使用下面这两个字段： iMaxSockets和iMaxUdpDg，它们是假定的同时最多可打开多少套接字和数据报的最大长度。然而，要知道数据报的最大长度应该通过WSAEnumProtocols来查询协议信息。同时最多打开套接字的数目不是固定的，很大程度上和可用物理内存的多少有关。最后， lpVendorInfo字段是为 Winsock实施方案有关的指定厂商信息预留的。任何一个 Win32平台上都没有使用这个字段。

表7-1列出了各种微软Windows平台支持的Winsock最新版本。需要记住的重要点是两个版本之间的差别。 Winsock 1.x不支持本章描述的多数高级 Winsock特性。除此以外，对采用Winsock 1的应用而言，必须有Winsock.h包容文件，而对使用Winsock 2的应用而言，则需要Winsock 2.h包容文件。

表7-1 各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 NT 4.0	2.2
Windows 2000	2.2
Windows CE	1.1

注意 针对Windows 95的Winsock 2升级版可从http://www.microsoft.com/Windows 95/downloads/下载。

注意，即使一个平台支持 Winsock 2，仍可不需要Winsock的最新版本。也就是说，如果你打算编写主流平台均支持的应用程序，可根据 Winsock 1.1规格来编写。编出来的程序可以在Windows NT 4.0平台上正确无误地运行，因为所有的Winsock1.1调用都通过Winsock 2 DLL映射出来了。同时，若市面上出现了Winsock库的新版本，它是针对你正在使用的平台的，那么，你肯定想急于升级。这些新版本中包含了错误纠正，这样，老代码就可无故障运行了——至少理论上如此。某些情况下， Winsock堆栈的行为和规格中定义的不同。如此一来，许多程序员编程时根据特定目标平台的行为来进行编程，而不是根据规格来写程序。比如，在Windows NT 4.0平台下，一个程序正在用异步窗口事件模型，表明可以写入数据的每个成功的send或WSASend之后，才会投递FD_WRITE。然而，根据规定， FD_WRITE是在系统能够发送数据时（比如在应用程序启动时）投递，而且，投递的那个 FD_WRITE意味着在收到WSAEWOULDBLOCK错误之前，应该一直写入数据。事实上，在系统发送所有待发数据和可以处理更多的send WSAsend调用之后，将向用户的应用程序窗口投递一个FD_WRITE事件，这时，才又可以向网络写入数据（参见“微软知识库”文章 Q186245）。这一问题已在Windows NT 4.0的Service Pack4和Windows 2000中得到解决。

但是，在很多情况下，编写新应用程序时，用户都想加载已发布的Winsock库的最新版本。比如， 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。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。话又说回来， WSADATA结构的wHighVersion就是当前系统上的库支持的最新版本。

7.2 错误检查和控制

对编写成功的Winsock应用程序而言，错误检查和控制是至关重要的，因此，我们打算先为大家介绍错误检查和控制。事实上，对 Winsock函数来说，返回错误是非常常见的。但是，多数情况下，这些错误都是无关紧要的，通信仍可在套接字上进行。尽管其返回的值并非一成不变，但不成功的 Winsock调用返回的最常见的值是 SOCKET_ERROR。在详细介绍各个API调用时，我们打算指出和各个错误对应的返回值。实际上， SOCKET_ERROR常量是-1。如果调用一个Winsock函数，错误情况发生了，就可用 WSAGetLastError函数来获得一段代码，这段代码明确地表明发生的状况。该函数的定义如下：

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

发生错误之后调用这个函数，就会返回所发生的特定错误的完整代码。 WSAGetLastError函数返回的这些错误都已预定义常量值，根据 Winsock版本的不同，这些值的声明不在Winsock 1.h中，就会在Winsock 2.h中。两个头字段的唯一差别是Winsock 2.h中包含的错误代码（针对Winsock 2中引入的一些新的 API函数和性能）更多。为各种错误代码定义的常量（带有#定义指令）一般都以WSAE开头。

7.3 面向连接的协议

本节讨论针对接收连接和建立连接所需要的 Winsock函数。首先，讨论如何监听客户机连接，并探讨接受或拒绝一个连接所需的操作。随后，将讨论怎样初始化同服务器的一个连接。最后，讨论数据在连接会话中是如何传输的。

7.3.1 服务器API函数

“服务器”其实是一个进程，它需要等待任意数量的客户机连接，以便为它们的请求提供服务。对服务器监听的连接来说，它必须在一个已知的名字上。在 TCP/IP中，这个名字就是本地接口的IP地址，加上一个端口编号。每种协议都有一套不同的定址方案，所以有一种不同的命名方法。在Winsock中，第一步是将指定协议的套接字绑定到它已知的名字上。这个过程是通过API调用bind来完成的。下一步是将套接字置为监听模式。这时，用 API函数listen来完成的。最后，若一个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。在接下来的几个小节中，我们将讨论绑定和监听所需的每一个 API调用，另外还要讨论用于接受客户机连接所需的每个 API调用。在图7-1中，我们展示了为建立一个通信信道，服务器和客户机必须执行的基本调用。

1. bind

一旦为某种特定协议创建了套接字，就必须将套接字绑定到一个已知地址。 bind函数可将指定的套接字同一个已知地址绑定到一起。该函数声明如下；

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

```
);
```

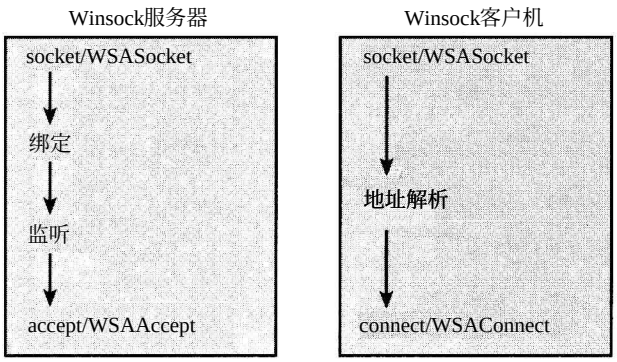

图7-1 服务器和客户机的Winsock基础

其中，第一个参数s代表我们希望在上面等待客户机连接的那个套接字。第二个参数的类型是struct sockaddr，它的作用很简单，就是一个普通的缓冲区。针对自己打算使用的那个协议，必须把该参数实际地填充一个地址缓冲区，并在调用 bind时将其造型为一个 struct sockaddr。为简化起见，本章都将使用这个类型。第三个参数代表要传递的、由协议决定的地址的长度。举个例子来说，下列代码阐述了在一个 TCP连接上，如何来做到这一点：

```
SOCKET              s;
struct 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));
```

假如你不清楚sockaddr_in结构的含义，请参考第 6章的TCP/IP定址小节。从这个例子中，大家可看到我们创建了一个流套接字。随后，我们设置了 TCP/IP地址结构，打算在它上面接受客户机连接。在这种情况下，套接字绑定到端口号 5150上的默认IP接口。之前的bind调用将套接字同IP接口／端口关联在一起。

一旦出错， bind就会返回 SOCKET_ERROR。对 bind来说，最常见的错误是WSAEADDRINUSE。如使用的是TCP/IP，那么WSAEADDRINUSE就表示另一个进程已经同本地IP接口和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。假如你针对一个套接字调用bind，但那个套接字已经绑定，便会返回 WSAEFFAULT错误。

2. listen

我们接下来要做的是将套接字置入监听模式。 bind函数的作用只是将一个套接字和一个指定的地址关联在一起。指示一个套接字等候进入连接的 API函数则是listen，其定义如下：

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

第一个参数同样是限定套接字。backlog参数指定了正在等待连接的最大队列长度。这个参数非常重要，因为完全可能同时出现几个服务器连接请求。例如，假定 backlog参数为2。如果三个客户机同时发出请求，那么头两个会被放在一个“待决”（等待处理）队列中，以便应用程序依次为它们提供服务。而第三个连接会造成一个 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，这个限制是由基层的协议提供者决定的。如果出现非法值，那么会用与之最接近的一个合法值来取代。除此以外，对于如何知道实际的 backlog值，其实并不存在一种标准手段。

与listen对应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，你忘记在 listen之前调用bind。否则，与bind调用相反，使用 listen时可能收到WSAEADDRINUSE。这个错误通常是在进行bind调用时发生的。

3. accept和WSAAccept

现在，我们已做好了接受客户连接的准备。这是通过 accept或WSAAccept函数来完成的。accept格式如下：

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

其中，参数 s是一个限定套接字，它处在监听模式。第二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，可为待决连接队列中的第一个连接请求提供服务。 accept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 IP地址信息，而addrlen参数则指出结构的长度。此外，accept会返回一个新的套接字描述符，它对应于已经接受的那个客户机连接。对于该客户机后续的所有操作，都应使用这个新套接字。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。

Winsock 2引入了一个名为WSAAccept的函数。它能根据一个条件函数的返回值，选择性地接受一个连接。这个新函数的定义如下：

```
SOCKET WSAAccept(
    SOCKET s,
    struct sockaddr FAR * addr,
    LPINT addrlen,
    LPCONDITIONPROC lpfnCondition,
    DWORD dwCallbackData
);
```

其中，头三个参数与accept的Winsock 1版本是相同的。lpfnCondition参数是指向一个函数的指针，那个函数是根据客户请求来调用的。该函数决定是否接受客户的连接请求，定义如下：

```
int CALLBACK ConditionFunc(
    LPWSABUF lpCallerId,
    LPWSABUF lpCallerData,
    LPQOS lpSQOS,
```

```
    LPQOS lpGQOS,
    LPWSABUF lpCalleeId,
    LPWSABUF lpCalleeData,
    GROUP FAR * g,
    DWORD dwCallbackData
);
```

lpCallerId是一个值参数，其中包含连接实体的地址。 WSABUF结构是许多Winsock 2函数常用的。对它的声明如下：

```
typedef struct __WSABUF {
    u_long      len;
    char FAR * buf;
} WSABUF, FAR * LPWSABUF;
```

根据它的用途， len字段要么指定由buf字段指向的那个缓冲区的长度，要么指定包含在数据缓冲区buf中的数据量。

对lpCallerId来说， buf指针指向的是一个地址结构。该结构针对的是建立连接的那种特定通信协议。为正确返回信息，只须将 buf指针建立为恰当的SOCKADDR类型。在TCP/IP的情况下，在SOCKADDR_IN结构中，当然应该包含建立连接的那个客户机的 IP地址。在连接请求期间，大多数网络协议都能提供对呼叫者 ID信息的支持。

lpCallerData参数中包含了随连接请求一道，由客户机发出的任何连接数据。若其中未指定呼叫者数据，那么该参数就默认为NULL。要注意的是，对大多数网络协议（如TCP）来说，它们并不提供对连接数据的支持。至于一种协议到底是支持连接数据，还是支持断开数据，可用WSAEnumProtocols函数对Winsock目录中相应的条目进行查询，从而得出正确的结论。这方面的详情可参考第5章。

lpSQOS和lpGQOS参数对客户机请求的任何一个服务质量（ QOS）参数进行指定，两个参数都引用了一个QOS结构，该结构中包含的信息是关于收发数据所需要的带宽。如果客户机没有要求QOS，这些参数都将是NULL。这两个参数的不同之处在于 lpSQOS指定的是一个独立的连接，而lpGQOS则用于套接字组。在Winsock 1或2中没有实施或支持套接字组（关于QOS的详情，可参考第12章）。

lpClalleeId属于另一种WSABUF结构，这一结构中包含已与客户机需要与之连接的本地地址。该结构的buf字段同样指向其相应地址家族的一个 SOCKADDR对象。对正在一个多主机的机器上运行的服务器来说，这种信息非常有用。记住，如果服务器和 INADDR_ANY地址绑定在一起，任何一个网络接口都可为连接请求提供服务。随后，该参数会返回实际建立连接的那个接口。

lpClalleeData参数是lpCallerData的补充。lpCalleeData参数指向一个WSABUF结构，服务器可利用这个结构把数据当作连接请求进程的一部分，返回客户机。如果服务提供者支持这一选项， len字段就会指出作为这个连接请求一部分，服务器最多可向客户机返回多少字节。这种情况下，服务器会根据这一数量，将尽可能多的字节复制到 WSABUF结构的buf部分，同时用len字段指出实际传输了多少个字节。如果服务器不想返回任何连接数据，那么，在返回之前，条件接受函数应将 len字段设为0。假如提供者不支持连接数据， len字段就会为0。大多数协议同样都不支持在接受连接之前进行数据交换。事实上， Win32平台当前支持的所有协议都不支持这一特性。

服务器将传递给条件函数的参数处理完之后，必须指出到底是接受、拒绝还是延后客户

机的连接请求。如果服务器打算接受连接，那么条件函数就应返回 CF_ACCEPT。如果拒绝，函数就应返回CF_REJECT。如果出于某种原因，现在还不能做出决定，就应返回 CF_DEFER。若服务器准备对这个连接请求进行处理，就应调用 WSAAcccept。要注意的是，条件函数在与WSAAccept函数相同的进程内运行，而且会立即返回。另外还要注意的是，对于当前的Win32平台支持的协议来说，条件接受函数并不意味着客户机的连接请求必须在从该函数返回一个值之后才会得到满足。大多数情况下，最基层的网络堆栈在条件接受函数调用的那一刻，就已经接受了连接。如果返回 CF_REJECT值，基层堆栈就会将连接简单地关闭了事。在此，我们对条件函数的用法不准备进行深入讲解，详情参见第 12章。

如发生错误，就会返回INVALID_SOCKET。最常见的错误是WSAEWOULDBLOCK。如果监听套接字处于异步状态或非暂停模式，同时没有要接受的连接时，就会产生此类的错误。若条件函数返回CF_DEFER，WSAAccept就会返回WSATRY_AGAIN错误。如果条件函数返回CF_REJECT，WSAAccept错误就是WSAECONNREFUSED。

7.3.2 客户机API函数

客户机要简单得多，建立成功连接所需的步骤也要少得多。客户机只需三步操作：

1) 用socket或WSASocket创建一个套接字。

2) 解析服务器名（以基层协议为准）。

3) 用connect或WSAConnect初始化一个连接。

通过第6章的学习，我们已知道了如何建立套接字和解析 IP主机名，所以现在只有一步未做，那就是建立一个连接。在第 6章中，我们还讨论了用于其他协议家族的各种名字解析方法。

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TCP状态，就能更好地理解Winsock API调用如何对基层协议中的改变产生影响。此外，许多程序员在关闭套接字时，会碰到一个常见问题；围绕套接字关闭的TCP状态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。作为客户机，需要用一个 ACK包对它做出反应。此时，客户机的套接字会变成 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。

若一个服务器的套接字同一个本地接口和端口绑定起来，并在它上面进行监听，那么套接字的状态便是LISTEN。客户机试图与之连接时，服务器就会收到一个 SYN包，并用一个SYN-ACK包做出响应。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，令服务器套接字的状态变成ESTABLISHED。

一旦应用处于ESTABLISHED状态，可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 7-2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当作第二个参数），会向对方发出一个FIN包，而且套接字的状

态则变成FIN_WAIT_1。正常情况下，通信对方会回应一个 ACK包，我们的套接字的状态随之变成FIN_WAIT_2。如对方也关闭了连接，便会发出一个 FIN包，我们的机器则会响应一个ACK包，并将己方套接字的状态置为 TIME_WAIT。

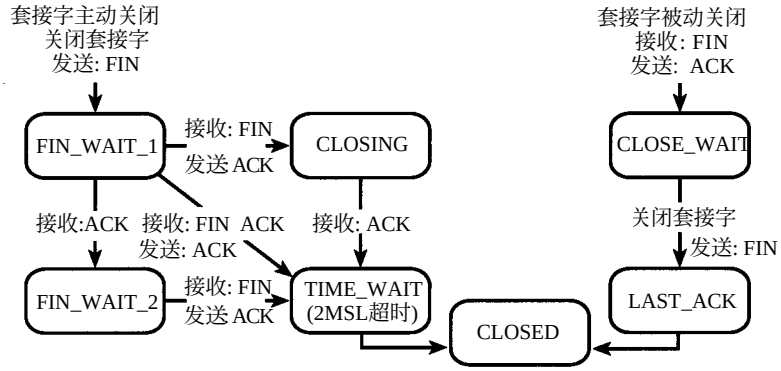

图7-2 TCP套接字的关闭状态

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），表示一个数据包在丢弃之前，可在网络上存在多长时间。每个IP包都含有一个“ 生存时间 ”（TTL）字段，若它递减为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 MSL时间的两倍之久。这样一来， TCP就可以在最后一个ACK丢失的前提下，重新发送它，也就是说， FIN会被重新传送出去。 MSL时间两倍之久的等待状态结束之后，套接字便进入 CLOSED状态。

采取主动关闭措施时，有两个路径会进入 TIME_WAIT状态。在我们以前的讨论中，只有一方发出一个 FIN，并接收一个 ACK响应。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，需要两个路径发挥作用。在一个路径中 (即同步关闭)，一台计算机和它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包做出响应，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，另一个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。若应用程序发出一个 FIN数据包，但几乎同时便从对方那里接收到一个FIN-ACK包，这种情况就会发生。在这种情况下，对方会确认收到应用程序的 FIN包，并送出自己的FIN包。对于这个包，应用程序会用一个 ACK包做出响应。

TIME_WAIT状态的主要作用是在TCP连接处于2MSL等待状态的时候，规定用于建立那个连接的一对套接字不可被拒绝。这对套接字由本地 IP端口以及远程IP端口组成。对某些TCP实施方案来说，它们不允许拒绝处于 TIME_WAIT状态下的套接字对中的任何端口号。在微软的方案中，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，就会失败，并返回 WSAEADDRINUSE错误。要解决这一问题（除了等待使用那个本地端口来脱离 TIME_WAIT状态的套接字对），一个办法是使用套接字选项SO_REFUSEADDR，我们将在第9章对这个选项进行详细讨论。

被动关闭情况下，应用程序会从对方那里接收一个 FIN包，并用一个ACK包做出响应。

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套接字，所以不能再发送数据了。但应用程序却不同，它能一直发送数据，直到对方的套接字已关闭为止。要想关闭对方的连接，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。应用程序从对方收到一个 ACK包后，它的套接字就会逆转成 CLOSED状态。

要想了解 TCP/IP协议的有关详情，请参阅 RFC 793文件。可在 http://www.rfc-editor.org那里找到这份文件。

connect函数和WSAConnect函数

最后一步就是连接。这是通过调用 connect函数或WSAConnect函数来完成的。我们先来看看该函数的Winsock 1版本，其定义如下：

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

该函数的参数是相当清楚的： s是即将在其上面建立连接的那个有效 TCP套接字；name是针对TCP（说明连接的服务器）的套接字地址结构（ SOCKADDR_IN）；namelen则是名字参数的长度。 Winsock 2版本中，它的定义是这样的：

```
int WSAConnect(
    SOCKET s,
    const struct sockaddr FAR * name,
    int namelen,
    LPWSABUF lpCallerData,
    LPWSABUF lpCalleeData,
    LPQOS lpSQOS,
    LPQOS lpGQOS
);
```

前三个参数和 connect API函数的参数是完全一样的。另外两个参数 —— lpCallerData和lpCalleeData，是字串缓冲区，用于收发请求连接时的数据。 lpCallerData参数是指向缓冲区的指针，缓冲区内包含客户机向服务器发出的请求连接的数据。 lpCalleeData参数则指向另一个缓冲区，区内包含服务器向客户机返回的建立连接时的数据。这两个参数都是 WSABUF结构，因此，若是lpCallerData，len字段应该设为buf字段中准备传输的数据长度。若是lpCalleeData，len字段则代表buf中的缓冲区长度，设为从服务器返回的数据长度。最后两个参数 —— lpSQOS和lpGQOS，表示QOS结构，该结构对即将建立的连接上收发数据所需要的带宽进行了定义。lpQOS参数用于指定套接字s需要的服务质量，而lpGQOS则用于指定套接字组所需要的服务质量。目前，尚未提供对套接字组的支持。若lpQOS是空值，则表明没有某应用专用的QOS。

如果你想连接的计算机没有监听指定端口这一进程， connect调用就会失败，并发生错误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能用时（亦可能因为到主机之间的路由上出现硬件故障或主机目前不在网上）。

7.3.3 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接收数据，可用这两个 API函数：

send和WSASend。第二个函数是Winsock 2中专有的。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都属于简单的 char类型。也就是说，这些函数没有“ Unicode ”版本。这一点对Windows CE来说尤为重要，因为Windows CE默认使用Unicode。使用Unicode时有一种选择，即把字符串当作 char*或把它造型为char*发送。需要注意的是，在利用字符串长度函数告诉 Winsock API函数收发的数据有多少字符时，必须将这个值乘以2，因为每个字符占用字串组的两个字节。另一种选择是在将字串数据投给 Winsock API函数之前，用WideCharToMultiByte把UNICODE转换成ASCII码。

另外，所有收发函数返回的错误代码都是 SOCKET_ERROR。一旦返回错误，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。两者均涉及到即将关闭连接这一问题 —— 要么通过超时，要么通过通信方关闭连接。另一个常见错误是 WSAEWOULDBLOCK，一般出现在套接字处于非暂停模式或异步状态时。这个错误主要意味着指定函数暂不能完成。第 8章，我们将详细说明各种Winsock I/O方法，以避免出现此类错误。

1. send和WSASend

要在已建立连接的套接字上发送数据，第一个可用的 API函数是send，其原型为：

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

SOCKET参数是已建立连接的套接字，将在这个套接字上发送数据。第二个参数 buf，则是字符缓冲区，区内包含即将发送的数据。第三个参数 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还可以是对那些标志进行按位“或运算”的一个结果。 MSG_DONTROUTE标志要求传送层不要将它发出的包路由出去。由基层的传送决定是否实现这一请求（例如，若传送协议不支持该选项，这一请求就会被忽略）。MSG_OOB标志预示数据应该被带外发送。

对返回数据而言， send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，这一错误一般发生在虚拟回路由于超时或协议有错而中断的时候。发生这种情况时，应该关闭这个套接字，因为它不能再用了。远程主机上的应用通过执行强行关闭或意外中断操作重新设置虚拟虚路时，或远程主机重新启动时，发生的则是WSAECONNRESET错误。再次提醒大家注意，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，它发生在连接由于网络故障或远程连接系统异常死机而引起的连接中断时。

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。第二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，又可以是一组结构。第三个参数指明准备投递的 WSABUF结构数。记住，每个WSABUF结构本身就是一个字符缓冲和缓冲长度。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。这就是我们稍后要讲的“分散集中 I/O模式”；但是，在一个已建立连接的套接字上利用多缓冲来发送数据时，顺序是从第一个到最后一个WSABUF结构。lpNumberOfBytesSent是指向DWORD（是WSASend调用返回的）的指针，其中包含字节总发送数。dwFlags参数相当于它在send中的等同物。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模式之一，关于这一点，我们将在第8章详细讲解。

WSASend函数把lpNumberOfBytesSent设为写入的字节数。成功的话，该函数就返回 0，否则就返回SOCK_ERROR，常见错误和send函数的情形一样。

2. WSASendDisconnect

该函数非常特殊，一般不用。其原型是：

```
int WSASendDisconnect (
    SOCKET s,
    LPWSABUF lpOUT boundDisconnectData
);
```

该函数起初将套接字置为关闭状态，发送无连接的数据。当然，它只能用于支持从容关机和无连接数据的传输协议。目前还没有传输提供者支持无连接的数据。 WSASendDisconnect函数的行为和利用 SD_SEND参数调用 shutdown函数差不多，但它另外还要发送包含在boundDisconnectData参数中的数据。后来的数据禁止在这个套接字上发送。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。

带外数据

对已建立连接的流套接字上的应用来说，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“带外数据”（Out-of-band, OOB）。位于连接另一端的应用可通过一个独立的逻辑信道（从概念上讲，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指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.RFC 793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，表明 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。但是在RFC 1122中，却将紧急偏移描述成指向紧急字节本身。

Winsock规格中，与协议无关的 OOB数据和TCP的OOB数据实施（紧急数据）均采用了OOB这一术语。要查看待发数据中是否包含紧急数据，必须通过 SIOCATMARK选项调用ioctlsocket函数。第9章将介绍SIOCATMARK的用法。

Winsock提供了获得紧急数据的几个方法。一是紧急数据一旦在线插入，它就会出现

在普通数据流中；二是可以关闭在线插入，这样，不连续调用接收函数就会只返回紧急数据。至于控制OOB数据行为的套接字选项SO_OOBINLINE，我们也将在第9章详细讨论。

Telnet和Rlogin使用紧急数据是有原因的。尽管如此，除非你计划编写自己的 Telnet和Rlogin，否则就应该远离紧急数据。因为它不容易定义，而且其他平台上的实施情况可能和Win32有所不同。在迫不得已的情况下使用紧急数据，必须发信号通知通信方为紧急数据执行一个独立的控制套接字，并为普通数据的传输保留主要的套接字连接。

3. recv和WSARecv

对在已连接套接字上接受接入数据来说， 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
);
```

第一个参数s，是准备接收数据的那个套接字。第二个参数 buf，是即将收到数据的字符缓冲，而len则是准备接收的字节数或 buf缓冲的长度。最后， 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和运算。当然， 0表示无特殊行为。MSG_PEEK会使有用的数据复制到所提供的接收端缓冲内，但是没有从系统缓冲中将它删除。另外，还返回了待发字节数。

消息取数不太好。它不仅导致性能下降（因为需要进行两次系统调用，一次是取数，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射出真正有用的数量。与此同时，把数据留在系统缓冲，可容纳接入数据的系统空间就会越来越少。其结果便是，系统减少各发送端的 TCP窗口容量。由此，你的应用就不能获得最大的流通。最好是把所有数据都复制到自己的缓冲中，并在那里计算数据。前面曾介绍过MSG_OOB标志。有关详情，参见前面“ 带外数据 ”的内容。

在面向消息或面向数据报的套接字上使用 recv时，这几点应该注意。在待发数据大于所提供的缓冲这一事件中，缓冲内会尽量地填充数据。这时， recv调用就会产生WSAEMSGSIZE错误。注意，消息长错误是在使用面向消息的协议时发生的。流协议把接入的数据缓存下来，并尽量地返回应用所要求的数据，即使待发数据的数量比缓冲大。因此，对流式传输协议来说，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。第二和第三个参数是接收数据的缓冲。 lpBuffers参数是

一个WSABUF结构组成的数组，而dwBufferCount则表明前一个数组中WSABUF结构的数目。如果接收操作立即完成，lpNumberOfBytesReceived参数就会指向执行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或者对这些值进行按位和运算之后的结果。MSG_PARTIAL标志使用和出现的地方不同，其含义也不同。对面向消息的协议来说，这个标志是WSARecv调用返回后设置的（如果因为缓冲空间不够导致整条消息未能在这次调用中返回的话）。这时，后面的WSARecv调用就会设置这个标志MASG_PARTIAL，直到整条消息返回，才把这个标志清除。如果这个标志当作一个输入参数投递，接收操作应该在一收到数据就结束，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。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 5章。lpOverlapped和lpCompletionROUTINE参数用于重叠I/O操作，第8章将对此详细讨论。

4. WSARecvDisconnect

这函数与WSASendDisconnect函数对应，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和一个有效的WSABUF结构（带有收到的数据）。收到的数据可以只是断开数据。这个断开数据是另一端执行WSASendDisconnect调用发出的，它不能用于接收普通数据。另外，一旦收到这个数据，WSARecvDisconnect函数就会取消接收远程通信方的数据，其作用和调用带有SD_RECV的shutdown函数相同。

5. WSARecvEx

WSARecvEx函数是微软专有的Winsock 1扩展，除了flags参数是按值引用外，其余和recv函数是一样的。它允许基层的提供者设置MSG_PARTIAL标志。该函数的原型如下：

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

如果收到的数据不是一条完整的消息，flags参数中就会返回MSG_PARTIAL标志。对面向消息的协议（即非流协议）来说，这个标志比较有用（即非流协议）。在MSG_PARTIAL标志被当作flags参数的一部分投递，而且收到的消息又不完整时，调用WSARecvEx，就会立即返回收到的那个数据。如果提供的接收缓冲容纳不下整条消息，WSARecvEx就会失败，并出现WSAEMSGSIZE错误，剩下的数据也会被截掉。注意，MSG_PARTIAL标志和WSAEMSGSIZE错误之间的确区别是：有了这个错误，即使整条消息到达接收端，但由于提供的数据缓冲太少，也不能对它进行接收。MSG_PEEK和MSG_OOB标志还可以和WSARecvEx一起使用。

7.3.4 流协议

由于大多面向连接的协议同时也是流式传输协议，所以，在此提一下流式协议。对于流

套接字上收发数据所用的函数，需要明白的是：它们不能保证对请求的数据量进行读取或写入。比如说，一个2048字节的字符缓冲，准备用send函数来发送它。采用的代码是：

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

// Fill sendbuff with 2048 bytes of data

// Assume s is a valid, connected stream socket
ret = send(s, sendbuff, nBytes, 0);
```

对send函数而言，可能会返回已发出的少于 2048的字节。ret变量将设为发送的字节数，这是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。在发送数据时，内部缓冲区会将数据一直保留到应该将它发到线上为止。几种常见的情况都可导致这一情形的发生。比方说，大量数据的传输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，以指出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，为待发数据做好准备。对发送端来说，这样会强令它在收到一个新的大于0的窗口大小之前，不得再发数据。在使用send调用时，缓冲区可能只能容纳1024个字节，这时，便有必要再提取剩下的 1024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;

// Fill sendbuff with 2048 bytes of data

// Assume s is a valid, connected stream socket
nLeft = nBytes;
idx = 0;
while (nLeft > 0)
{
    ret = send(s, &sendbuff[idx], nLeft, 0);
    if (ret == SOCKET_ERROR)
    {
        // Error
    }
    nLeft -= ret;
    idx += ret;
}
```

对在流套接字上接收数据来说，前一段代码有用，但意义不大。因为流套接字是一个不间断的数据流，应用程序在读取它时，和它应该读多少数据之间通常没有关系。如果应用需要依赖于流协议的离散数据，你就有别的事要做。如果所有消息长度都一样，则比较简单，也就是说，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)
    {
        // Error
    }
    idx += ret;
    nLeft -= ret;
}
```

消息长度不同，处理也可能不同。因此，有必要利用你自己的协议来通知接收端，即将到来的消息长度是多少。比方说，写入接收端的前 4个字节一直是整数，表示即将到来的消息有多少字节。然后，接收端先查看前 4个字节的方式，把它们转换成一个整数，然后判断构成消息的字节数是多少，通过这种方式，便开始逐次读取。

分散集合 I/O

分散集合支持是 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。对收发格式特别的数据这一类的应用来说，它是非常有用的。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，就可用一个由三个 WSABUF结构组成的数组调用WSASend，这三个结构分别对应的三种消息类型。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分别是 32字节、64字节和16字节。

在使用基于流的套接字时，分散集合 I/O模式只是把WSABUF结构中提供的数据缓冲当作一个连续性的缓冲。另外，接收调用可能在所有缓冲填满之前就返回。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决定。如果缓冲不够，调用就会失败，并出现 WSAEMSGSIZE错误，为了适应可用的缓冲数据就会被截断。当然，如果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。

7.3.5 中断连接

一旦完成任务，就必须关掉连接，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开着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函数从容中断连接。接下来，我们来谈谈这两个API函数。

1. shutdown

为了保证通信方能够收到应用发出的所有数据，对一个编得好的应用来说，应该通知接收端“ 不再发送数据 ”。同样，通信方也应该如此。这就是所谓的“ 从容关闭 ”方法，并由shutdown函数来执行。shutdown的定义如下：

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

);

how参数可以是下面的任何一个值： 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，不管数据在等候接收，还是数据接连到达，都要重设连接。尽管如此， UDP套接字上，仍然接受并排列接入的数据。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。

2. closesocket

closesocket函数用于关闭套接字，它的定义如下：

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

closesocket的调用会释放套接字描述符，再利用套接字执行调用就会失败，并出现WSAENOTSOCK错误。如果没有对该套接字的其他引用，所有与其描述符关联的资源都会被释放。其中包括丢弃所有等侯处理的数据。

对这个进程中任何一个线程来说，它们执行的待决异步调用都在未投递任何通知消息的情况下被删除。待决的重叠操作也被删除。与该重叠操作关联的任何事件，完成例程或完成端口能执行，但最后会失败，出现 WSA_OPERATION_ABORTED错误。异步和非封锁 I/O模式将在第 8章深入讲解。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。要得知其中缘由，参考第 9章中对SO_LINGER选项的描述。

7.3.6 综合分析

大家面对如此多的收发函数，可能有点无所适从，但事实上，对多数应用来说，接收数据一般用recv或WSARecv；发送数据则用send或WSASend。其他收发函数都是指定随某些罕见的特性一起使用（或由传送协议支持的）。在此，我们将利用前面所讲的原理和函数，对一个简单的客户机／服务器示例进行分析。程序清单 7-1中包含的代码是针对一个简单的回应服务器的。这个应用建立了一个套接字，绑定了一个本地 IP接口和端口，并监听客户机连接。在收到客户机连接请求之后，又建立了一个新套接字，再把这个新套接字投递到一个再生的客户机进程中。这个线程只读读数据，便把它发回客户机。

程序清单7-1 回应服务器代码

```
// Module Name: Server.c
//
// Description:
//    This example illustrates a simple TCP server that accepts
//    incoming client connections. Once a client connection is
//    established, a thread is spawned to read data from the
//    client and echo it back (if the echo option is not
//    disabled).
//
// Compile:
//    cl -o Server Server.c ws2_32.lib
//
// Command line options:
//    server [-p:x] [-i:IP] [-o]
//           -p:x      Port number to listen on
```

```
//           -i:str    Interface to listen on
//           -o        Receive only; don't echo the data back
//
#include <winsock2.h>

#include <stdio.h>
#include <stdlib.h>

#define DEFAULT_PORT        5150
#define DEFAULT_BUFFER      4096

int    iPort      = DEFAULT_PORT; // Port to listen for clients on
BOOL   bInterface = FALSE,    // Listen on the specified interface
       bRecvOnly  = FALSE;    // Receive data only; don't echo back
char   szAddress[128];        // Interface to listen for clients on

//
// Function: usage
//
// Description:
//    Print usage information and exit
//
void usage()
{
    printf("usage: server [-p:x] [-i:IP] [-o]\n\n");
    printf("       -p:x      Port number to listen on\n");
    printf("       -i:str    Interface to listen on\n");
    printf("       -o        Don't echo the data back\n\n");
    ExitProcess(1);
}

//
// Function: ValidateArgs
//
// Description:
//    Parse the command line arguments, and set some global flags
//    to indicate what actions to perform
//
void ValidateArgs(int argc, char **argv)
{
    int i;

    for(i = 1; i < argc; i++)
    {
        if ((argv[i][0] == '-') || (argv[i][0] == '/'))
        {
            switch (tolower(argv[i][1]))
            {
                case 'p':
                    iPort = atoi(&argv[i][3]);
                    break;
                case 'i':
                    bInterface = TRUE;
                    if (strlen(argv[i]) > 3)
                        strcpy(szAddress, &argv[i][3]);
```

```
                    break;
                case 'o':
                    bRecvOnly = TRUE;
                    break;
                default:
                    usage();
                    break;
            }
        }
    }
}

//
// Function: ClientThread
//
// Description:
//    This function is called as a thread, and it handles a given
//    client connection.  The parameter passed in is the socket
//    handle returned from an accept() call.  This function reads
//    data from the client and writes it back.
//
DWORD WINAPI ClientThread(LPVOID lpParam)
{
    SOCKET        sock=(SOCKET)lpParam;
    char          szBuff[DEFAULT_BUFFER];
    int           ret,
                  nLeft,
                  idx;

    while(1)
    {
        // Perform a blocking recv() call
        //
        ret = recv(sock, szBuff, DEFAULT_BUFFER, 0);
        if (ret == 0)        // Graceful close
            break;
        else if (ret == SOCKET_ERROR)
        {
            printf("recv() failed: %d\n", WSAGetLastError());
            break;
        }
        szBuff[ret] = '\0';
        printf("RECV: '%s'\n", szBuff);
        //
        // If we selected to echo the data back, do it
        //
        if (!bRecvOnly)
        {
            nLeft = ret;
            idx = 0;
            //
            // Make sure we write all the data
            //
            while(nLeft > 0)
            {
```

```
                ret = send(sock, &szBuff[idx], nLeft, 0);
                if (ret == 0)
                    break;
                else if (ret == SOCKET_ERROR)
                {
                    printf("send() failed: %d\n",
                        WSAGetLastError());
                    break;
                }
                nLeft -= ret;
                idx += ret;
            }
        }
    }
    return 0;
}

//
// Function: main
//
// Description:
//    Main thread of execution. Initialize Winsock, parse the
//    command line arguments, create the listening socket, bind
//    to the local address, and wait for client connections.
//
int main(int argc, char **argv)
{
    WSADATA       wsd;
    SOCKET        sListen,
                  sClient;
    int           iAddrSize;
    HANDLE        hThread;
    DWORD         dwThreadId;
    struct sockaddr_in local,
                       client;

    ValidateArgs(argc, argv);
    if (WSAStartup(MAKEWORD(2,2), &wsd) != 0)
    {
        printf("Failed to load Winsock!\n");
        return 1;
    }
// Create our listening socket
//
sListen = socket(AF_INET, SOCK_STREAM, IPPROTO_IP);
if (sListen == SOCKET_ERROR)
{
    printf("socket() failed: %d\n", WSAGetLastError());
    return 1;
}
// Select the local interface, and bind to it
//
if (bInterface)
{
    local.sin_addr.s_addr = inet_addr(szAddress);
```

```
        if (local.sin_addr.s_addr == INADDR_NONE)
            usage();
    }
    else
        local.sin_addr.s_addr = htonl(INADDR_ANY);
    local.sin_family = AF_INET;
    local.sin_port = htons(iPort);

    if (bind(sListen, (struct sockaddr *)&local,
            sizeof(local)) == SOCKET_ERROR)
    {
        printf("bind() failed: %d\n", WSAGetLastError());
        return 1;
    }
    listen(sListen, 8);
    //
    // In a continuous loop, wait for incoming clients. Once one
    // is detected, create a thread and pass the handle off to it.
    //
    while (1)
    {
        iAddrSize = sizeof(client);
        sClient = accept(sListen, (struct sockaddr *)&client,
                         &iAddrSize);
        if (sClient == INVALID_SOCKET)
        {
            printf("accept() failed: %d\n", WSAGetLastError());
            break;
        }
        printf("Accepted client: %s:%d\n",
            inet_ntoa(client.sin_addr), ntohs(client.sin_port));

        hThread = CreateThread(NULL, 0, ClientThread,
                    (LPVOID)sClient, 0, &dwThreadId);
            if (hThread == NULL)
            {
                printf("CreateThread() failed: %d\n", GetLastError());
                break;
            }
            CloseHandle(hThread);
        }
        closesocket(sListen);

        WSACleanup();
        return 0;
    }
```

这个示例的客户机代码（见程序清单 7-2）更简单。客户机建立一个套接字，并对投入应用的服务器名进行解析，然后与服务器建立连接。连接一旦建成，就可发送大量的消息了。每次发送数据之后，客户机都会等待服务器发回的回应。客户机把得自套接字的数据打印出来。

回应客户机和服务器不能完全说明 TCP协议的流式传输。这是因为读入取操作是在写操作之后进行的，至少客户机这一端是这样的。当然，对服务器来说，还有另一种方式。因此，

服务器每次调用读取函数，一般都会返回客户机发出的整条消息。但不要误会。如果客户机的消息大得超过了TCP的最大传输单元，在线上，它会被分成几个小的数据包，这种情况下，接收端需要多次执行接收调用，才能收完整条消息。为了更好地说明流式传输，运行客户机和服务器时带上-O选项即可。这样，客户机便只管发送数据，接收端只管读取数据。服务器如下执行：

```
server -p:5150 -o
```

而客户机如下执行：

```
client -p:5150 -s:IP -n:10 -o
```

大家最可能见到的是客户机进行了10次send调用，而服务器在一次或两次recv调用中，就读取了10条消息。

程序清单7-2 回应客户机代码

```
// Module Name: Client.c
//
// Description:
//    This sample is the echo client. It connects to the TCP server,
//    sends data, and reads data back from the server.
//
// Compile:
//    cl -o Client Client.c ws2_32.lib
//
// Command Line Options:
//    client [-p:x] [-s:IP] [-n:x] [-o]
//           -p:x      Remote port to send to
//           -s:IP     Server's IP address or host name
//           -n:x      Number of times to send message
//           -o        Send messages only; don't receive
//
#include <winsock2.h>
#include <stdio.h>
#include <stdlib.h>

#define DEFAULT_COUNT       20
#define DEFAULT_PORT        5150
#define DEFAULT_BUFFER      2048
#define DEFAULT_MESSAGE     "This is a test of the emergency \
broadcasting system"

char  szServer[128],          // Server to connect to
      szMessage[1024];        // Message to send to sever
int   iPort     = DEFAULT_PORT;  // Port on server to connect to
DWORD dwCount   = DEFAULT_COUNT; // Number of times to send message
BOOL  bSendOnly = FALSE;         // Send data only; don't receive

//
// Function: usage:
//
// Description:
//    Print usage information and exit
//
void usage()
```

```
{
    printf("usage: client [-p:x] [-s:IP] [-n:x] [-o]\n\n");
    printf("       -p:x      Remote port to send to\n");
    printf("       -s:IP     Server's IP address or host name\n");
    printf("       -n:x      Number of times to send message\n");
    printf("       -o        Send messages only; don't receive\n");
    ExitProcess(1);
}

//
// Function: ValidateArgs
//
// Description:
//    Parse the command line arguments, and set some global flags
//    to indicate what actions to perform
//
void ValidateArgs(int argc, char **argv)
{
    int               i;

    for(i = 1; i < argc; i++)
    {
        if ((argv[i][0] == '-') || (argv[i][0] == '/'))
        {
            switch (tolower(argv[i][1]))
            {
                case 'p':        // Remote port
                    if (strlen(argv[i]) > 3)
                        iPort = atoi(&argv[i][3]);
                    break;
                case 's':       // Server
                    if (strlen(argv[i]) > 3)
                        strcpy(szServer, &argv[i][3]);
                    break;
                case 'n':       // Number of times to send message
                    if (strlen(argv[i]) > 3)
                        dwCount = atol(&argv[i][3]);
                    break;
                case 'o':       // Only send message; don't receive
                    bSendOnly = TRUE;
                    break;
                default:
                    usage();
                    break;
            }
        }
    }
}

//
// Function: main
//
// Description:
//    Main thread of execution. Initialize Winsock, parse the
//    command line arguments, create a socket, connect to the
//    server, and then send and receive data
```

```
//
int main(int argc, char **argv)
{
    WSADATA       wsd;
    SOCKET        sClient;
    char          szBuffer[DEFAULT_BUFFER];
    int           ret,
                  i;
struct sockaddr_in server;
struct hostent    *host = NULL;

// Parse the command line, and load Winsock
//
ValidateArgs(argc, argv);
if (WSAStartup(MAKEWORD(2,2), &wsd) != 0)
{
    printf("Failed to load Winsock library!\n");
    return 1;
}
strcpy(szMessage, DEFAULT_MESSAGE);
//
// Create the socket, and attempt to connect to the server
//
sClient = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
if (sClient == INVALID_SOCKET)
{
    printf("socket() failed: %d\n", WSAGetLastError());
    return 1;
}
server.sin_family = AF_INET;
server.sin_port = htons(iPort);
server.sin_addr.s_addr = inet_addr(szServer);
//
// If the supplied server address wasn't in the form
// "aaa.bbb.ccc.ddd," it's a host name, so try to resolve it
//
if (server.sin_addr.s_addr == INADDR_NONE)
{
    host = gethostbyname(szServer);
    if (host == NULL)
    {
        printf("Unable to resolve server: %s\n", szServer);
        return 1;
    }
    CopyMemory(&server.sin_addr, host->h_addr_list[0],
        host->h_length);
}
if (connect(sClient, (struct sockaddr *)&server,
    sizeof(server)) == SOCKET_ERROR)
{
    printf("connect() failed: %d\n", WSAGetLastError());
    return 1;
}
// Send and receive data
//
```

```
        for(i = 0; i < dwCount; i++)
        {
            ret = send(sClient, szMessage, strlen(szMessage), 0);
            if (ret == 0)
                break;
            else if (ret == SOCKET_ERROR)
            {
                printf("send() failed: %d\n", WSAGetLastError());
                break;
            }
            printf("Send %d bytes\n", ret);
            if (!bSendOnly)
            {
                ret = recv(sClient, szBuffer, DEFAULT_BUFFER, 0);
                if (ret == 0)        // Graceful close
                    break;
                else if (ret == SOCKET_ERROR)
                {
                    printf("recv() failed: %d\n", WSAGetLastError());
                    break;
                }
                szBuffer[ret] = '\0';
                printf("RECV [%d bytes]: '%s'\n", ret, szBuffer);
            }
        }
        closesocket(sClient);

        WSACleanup();
        return 0;
    }
```

7.4 无连接协议

和面向连接的协议比较起来，无连接协议的行为极为不同，因此，收发数据的方式也会有所不同。由于在和面向会话的服务器比较时，无连接接收端改动不大，所以我们先谈谈接收端（如果你愿意，也可称之为服务器）。接下来再谈发送端。

7.4.1 接收端

对于在一个无连接套接字上接收数据的进程来说，步骤并不复杂。先用socket或WSASocket建立套接字。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。和面向会话不同的是，我们不必调用 listen和accept。相反，只需等待接收数据。由于它是无连接的，因此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
);
```

前面四个参数和recv是一样的，其中包括标志MSG_OOB和MSG_PEEK。在使用无连接套接字时，和前面一样，仍然提醒大家慎用MSG_PEEK标志。对监听套接字的具体协议来说，from参数是一个SOCKADDR结构，带有指向地址结构的长度的fromlen。这个API调用返回数据时，SOCKADDR结构内便填入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的WSABUF结构的用法上。你可以利用dwBufferCount为WSARecvFrom提供一个或多个WSABUF缓冲。提供多个缓冲，就可用发散集合了。读取的字节总数返回在lpNumberOfBytesRecvd中。在调用WSARecvFrom时，lpFlags参数可以是代表无选项的0、MSG_OOB、MSG_PEEK或MSG_PARTIAL。这些标志还可以累加起来。如果在调用这个函数时，指定MSG_PARTIAL，提供者就知道返回数据，即使只收到了部分消息。调用返回之后，如果只收到部分消息，就会设置MSG_PARTIAL标志。再次返回之后，WSARecvFrom就会把lpFrom参数（它是一个指向SOCKADDR结构的指针）设为发送端的地址。再次提醒大家注意，lpFromLen指向SOCKADDR结构的长度，另外，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（我们将在下一章就此展开讨论）。

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连接。投入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。如果在一次应用中，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。

7.4.2 发送端

要在一个无连接的套接字上发送数据，有两种选择。第一种，也是最简单的一种，便是建立一个套接字，然后调用sendto或WSASendTo。我们先来讲解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，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，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
);
```

再次提醒大家注意，WSASendTo和前一版本中的SendTo函数类似。它把指向带有发给接收端的数据的指针当作lpBuffers参数，dwBufferCount参数指明现在的结构是多少。我们可发送多个WSABUF结构启用发散集合I/O。在函数返回之前，WSASendTo把第四个参数lpNumberOfBytesSent设为真正发到接收端的字节数。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8章讨论）。

通过接收数据的方式，就可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发送数据了。这种关联一旦建立，就不能再用带有地址的 sendto和WSASendTo，除非这个地址是投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，唯一的办法是在这个套接字句柄上调用 closesocket，并建立一个新的套接字。

7.4.3 基于消息的协议

正由于面向连接的协议同时也是流式协议，无连接协议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对数据边界有保护，所以提交给发送函数的数据在被发送完之前累积成块。对异步或非块式 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。这意味着基层的系统不能对不完整的那个数据进行处理，你应该稍后再次调用发送函数。下一章将对此进行详述。主要需要记住的是，采用基于消息的协议时，对于写入数据来说，只能把它当作一个自治行为。

在连接另一端，对接收函数的调用必须提供一个足够大的缓冲空间。如果提供的缓冲不够，接收调用就会失败，出现 WSAEMSGSIZE。发生这种情况时，缓冲会尽力接收，但未收完的数据会被丢弃。被截断的数据无法恢复。唯一例外的是支持部分消息的协议却例外，比方说AppleTalk PAP协议。在WSARecvEx函数只收到部分消息时，它会在返回之前，便把自己的出入标志参数设为MSG_PARTIAL。

对以支持部分消息的协议为基础的数据报来说，可考虑使用一个 WSARecv函数。在调用recv时，不会有这一个通知“读取的数据只是消息的一部分”。至于接收端怎样判断是否已读取整条消息，具体方法则由程序员决定。后来的 recv调用返回这个数据报的其他部分。由于有这个限制，所以利用 WSARecvEx函数非常方便，它允许设置和读取 MSG_PARTIAL标志，

MSG_PARTIAL标志指明整条消息是否已读取。 Winsock 2函数WSARecv和WSARecvFrom也支持这一标志。关于这个标志的更多知识，请参见对WSARecv、WSARecvEx和WSARecvFrom这三个函数的描述。

我们最后要谈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，我们来看一个最常见的问题：在一个 UDP套接字明显绑定到一个本地 IP接口和发送数据报时，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。而是建立一种联系，即绑定的IP接口成为发出去的 UDP数据报的源IP地址。路由表才真正决定数据报在哪个物理接口上传出去。如果不调用 bind，而是先调用 sendto或WSASendTo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；如果你先执行明显绑定，源 IP地址就会有误。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。

7.4.4 释放套接字资源

因为无连接协议没有连接，所以也不会有正式的关闭和从容关闭。在接收端或发送端结束收发数据时，它只是在套接字句柄上调用 closesocket函数。这样，便释放了为套接字分配的所有相关资源。

7.4.5 综合分析

对于在无连接的套接字上收发数据的步骤，大家现在已经很清楚了。接下来，我们来看看执行这一进程的代码。程序清单 7-3展示了一个无连接的接收端。这段代码说明了如何在默认接口或指定的本地接口上接收数据报。

程序清单7-3 无连接的接收端

```
// Module Name: Receiver.c
//
// Description:
//    This sample receives UDP datagrams by binding to the specified
//    interface and port number and then blocking on a recvfrom()
//    call
//
// Compile:
//    cl -o Receiver Receiver.c ws2_32.lib
//
// Command Line Options:
//    sender [-p:int] [-i:IP][-n:x] [-b:x]
//           -p:int   Local port
//           -i:IP    Local IP address to listen on
//           -n:x     Number of times to send message
//           -b:x     Size of buffer to send
//
#include <winsock2.h>
#include <stdio.h>
#include <stdlib.h>

#define DEFAULT_PORT            5150
#define DEFAULT_COUNT           25
#define DEFAULT_BUFFER_LENGTH   4096
```

```
int   iPort    = DEFAULT_PORT;          // Port to receive on
DWORD dwCount  = DEFAULT_COUNT,         // Number of messages to read
      dwLength = DEFAULT_BUFFER_LENGTH; // Length of receiving buffer
BOOL  bInterface = FALSE;               // Use an interface other than
                                        // default
char  szInterface[32];            // Interface to read datagrams from

//
// Function: usage:
//
// Description:
//    Print usage information and exit
//
void usage()
{
    printf("usage: sender [-p:int] [-i:IP][-n:x] [-b:x]\n\n");
    printf("       -p:int   Local port\n");
    printf("       -i:IP    Local IP address to listen on\n");
    printf("       -n:x     Number of times to send message\n");
    printf("       -b:x     Size of buffer to send\n\n");
    ExitProcess(1);
}

//
// Function: ValidateArgs
//
// Description:
//    Parse the command line arguments, and set some global flags to
//    indicate what actions to perform
//
void ValidateArgs(int argc, char **argv)
{
    int                i;

    for(i = 1; i < argc; i++)
{
    if ((argv[i][0] == '-') || (argv[i][0] == '/'))
    {
        switch (tolower(argv[i][1]))
        {
            case 'p':   // Local port
                if (strlen(argv[i]) > 3)
                    iPort = atoi(&argv[i][3]);
                break;
            case 'n':   // Number of times to receive message
                if (strlen(argv[i]) > 3)
                    dwCount = atol(&argv[i][3]);
                break;
            case 'b':   // Buffer size
                if (strlen(argv[i]) > 3)
                    dwLength = atol(&argv[i][3]);
                break;
            case 'i':    // Interface to receive datagrams on
                if (strlen(argv[i]) > 3)
                {
                    bInterface = TRUE;
```

```
                    strcpy(szInterface, &argv[i][3]);
                    }
                    break;
                default:
                    usage();
                    break;
            }
        }
    }
}

//
// Function: main
//
// Description:
//    Main thread of execution. Initialize Winsock, parse the command
//    line arguments, create a socket, bind it to a local interface
//    and port, and then read datagrams.
//
int main(int argc, char **argv)
{
    WSADATA        wsd;
    SOCKET         s;
    char          *recvbuf = NULL;
    int            ret,
                   i;
    DWORD          dwSenderSize;
SOCKADDR_IN    sender,
               local;

// Parse arguments and load Winsock
//
ValidateArgs(argc, argv);

if (WSAStartup(MAKEWORD(2,2), &wsd) != 0)
{
    printf("WSAStartup failed!\n");
    return 1;
}
// Create the socket, and bind it to a local interface and port
//
s = socket(AF_INET, SOCK_DGRAM, 0);
if (s == INVALID_SOCKET)
{
    printf("socket() failed; %d\n", WSAGetLastError());
    return 1;
}
local.sin_family = AF_INET;
local.sin_port = htons((short)iPort);
if (bInterface)
    local.sin_addr.s_addr = inet_addr(szInterface);
else
    local.sin_addr.s_addr = htonl(INADDR_ANY);
if (bind(s, (SOCKADDR *)&local, sizeof(local)) == SOCKET_ERROR)
{
```

```
        printf("bind() failed: %d\n", WSAGetLastError());
        return 1;
    }
    // Allocate the receive buffer
    //
    recvbuf = GlobalAlloc(GMEM_FIXED, dwLength);
    if (!recvbuf)
    {
        printf("GlobalAlloc() failed: %d\n", GetLastError());
        return 1;
    }
    // Read the datagrams
    //
    for(i = 0; i < dwCount; i++)
    {
        dwSenderSize = sizeof(sender);
        ret = recvfrom(s, recvbuf, dwLength, 0,
                (SOCKADDR *)&sender, &dwSenderSize);
        if (ret == SOCKET_ERROR)
            {
                printf("recvfrom() failed: %d\n", WSAGetLastError());
                break;
            }
            else if (ret == 0)
                break;
            else
            {
                recvbuf[ret] = '\0';
                printf("[%s] sent me: '%s'\n",
                    inet_ntoa(sender.sin_addr), recvbuf);
            }
        }
        closesocket(s);

        GlobalFree(recvbuf);
        WSACleanup();
        return 0;
    }
```

接收数据报非常简单。先建立一个套接字。然后把它和本地接口绑定在一起。如果和默认接口绑定，便可利用getsockname函数，找到这个默认接口的IP地址。getsockname函数只返回SOCKADDR_IN结构，这个结构关联到投递给自己的具体套接字，指出绑定这个套接字的接口。之后，便是调用recvfrom，读取接入的数据。注意，我们此时用recvfrom函数，是因为我们不注重部分消息；UDP协议不支持部分消息。事实上，TCP/IP堆栈收到大型数据报片段时，会一直等到所有片段重组。如果各片段的顺序被打乱了或丢失了部分消息，堆栈就会把这个数据报丢弃。

程序清单7-4提供了无连接发送端的代码。这个示例比接收端的的选项丰富得多。必不可少的参数是IP地址和远程接收端的端口号。 -c选项代表是否先执行connect调用；默认行为是不执行。再次提醒大家注意，它的收发步骤也很简单。先建立套接字。如果出现 -c选项，就带上远程接收端的地址和端口号，调用 connect。这一步是在调用send之后进行的。如果不执行连接，只须在建立套接字之后，利用sendto函数，就可以向接收端发送数据了。

程序清单7-4 无连接的发送端

```
// Module Name: Sender.c
//
// Description:
//    This sample sends UDP datagrams to the specified recipient.
//    The -c option first calls connect() to associate the
//    recipient's IP address with the socket handle so that the
//    send() function can be used as opposed to the sendto() call.
//
// Compile:
//    cl -o Sender Sender.c ws2_32.lib
//
// Command line options:
//    sender [-p:int] [-r:IP] [-c] [-n:x] [-b:x] [-d:c]
//           -p:int   Remote port
//           -r:IP    Recipient's IP address or host name
//           -c       Connect to remote IP first
//           -n:x     Number of times to send message
//           -b:x     Size of buffer to send
//           -d:c     Character to fill buffer with
//
#include <winsock2.h>
#include <stdio.h>
#include <stdlib.h>

#define DEFAULT_PORT            5150
#define DEFAULT_COUNT           25
#define DEFAULT_CHAR            'a'
#define DEFAULT_BUFFER_LENGTH   64

BOOL  bConnect = FALSE;                 // Connect to recipient first
int   iPort    = DEFAULT_PORT;          // Port to send data to
char  cChar    = DEFAULT_CHAR;          // Character to fill buffer
DWORD dwCount  = DEFAULT_COUNT,         // Number of messages to send
      dwLength = DEFAULT_BUFFER_LENGTH; // Length of buffer to send
char  szRecipient[128];                 // Recipient's IP or host name

//
// Function: usage
//
// Description:
//    Print usage information and exit
//
void usage()
{
    printf("usage: sender [-p:int] [-r:IP] "
           "[-c] [-n:x] [-b:x] [-d:c]\n\n");
    printf("       -p:int   Remote port\n");
    printf("       -r:IP    Recipient's IP address or host name\n");
    printf("       -c       Connect to remote IP first\n");
    printf("       -n:x     Number of times to send message\n");
    printf("       -b:x     Size of buffer to send\n");
    printf("       -d:c     Character to fill buffer with\n\n");
    ExitProcess(1);
}
```

```
//
// Function: ValidateArgs
//
// Description:
//    Parse the command line arguments, and set some global flags to
//    indicate what actions to perform
//
void ValidateArgs(int argc, char **argv)
{
    int i;

    for(i = 1; i < argc; i++)
    {
        if ((argv[i][0] == '-') || (argv[i][0] == '/'))
        {
            switch (tolower(argv[i][1]))
            {
                case 'p':        // Remote port
                    if (strlen(argv[i]) > 3)
                        iPort = atoi(&argv[i][3]);
                    break;
                case 'r':        // Recipient's IP addr
                    if (strlen(argv[i]) > 3)
                        strcpy(szRecipient, &argv[i][3]);
                    break;
                case 'c':        // Connect to recipient's IP addr
                    bConnect = TRUE;
                    break;
                case 'n':        // Number of times to send message
                    if (strlen(argv[i]) > 3)
                        dwCount = atol(&argv[i][3]);
                    break;
                case 'b':        // Buffer size
                    if (strlen(argv[i]) > 3)
                        dwLength = atol(&argv[i][3]);
                    break;
                case 'd':        // Character to fill buffer
                    cChar = argv[i][3];
                    break;
                default:
                    usage();
                    break;
            }
        }
    }
}
//
// Function: main
//
// Description:
//    Main thread of execution. Initialize Winsock, parse the command
//    line arguments, create a socket, connect to the remote IP
//    address if specified, and then send datagram messages to the
//    recipient.
//
```

```
int main(int argc, char **argv)
{
    WSADATA        wsd;
    SOCKET         s;
    char          *sendbuf = NULL;
    int            ret,
                   i;
    SOCKADDR_IN    recipient;

    // Parse the command line and load Winsock
    //
    ValidateArgs(argc, argv);

    if (WSAStartup(MAKEWORD(2, 2), &wsd) != 0)
    {
        printf("WSAStartup failed!\n");
        return 1;
    }
    // Create the socket
    //
    s = socket(AF_INET, SOCK_DGRAM, 0);
    if (s == INVALID_SOCKET)
    {
        printf("socket() failed; %d\n", WSAGetLastError());
        return 1;
    }
    // Resolve the recipient's IP address or host name
    //
    recipient.sin_family = AF_INET;
    recipient.sin_port = htons((short)iPort);
    if ((recipient.sin_addr.s_addr = inet_addr(szRecipient))
        == INADDR_NONE)
    {
        struct hostent *host=NULL;

        host = gethostbyname(szRecipient);
        if (host)
            CopyMemory(&recipient.sin_addr, host->h_addr_list[0],
                host->h_length);
        else
    {
        printf("gethostbyname() failed: %d\n", WSAGetLastError());
        WSACleanup();
        return 1;
    }
}
// Allocate the send buffer
//
sendbuf = GlobalAlloc(GMEM_FIXED, dwLength);
if (!sendbuf)
{
    printf("GlobalAlloc() failed: %d\n", GetLastError());
    return 1;
}
memset(sendbuf, cChar, dwLength);
//
// If the connect option is set, "connect" to the recipient
```

```
// and send the data with the send() function
//
if (bConnect)
{
    if (connect(s, (SOCKADDR *)&recipient,
            sizeof(recipient)) == SOCKET_ERROR)
    {
        printf("connect() failed: %d\n", WSAGetLastError());
        GlobalFree(sendbuf);
        WSACleanup();
        return 1;
    }
    for(i = 0; i < dwCount; i++)
    {
        ret = send(s, sendbuf, dwLength, 0);
        if (ret == SOCKET_ERROR)
        {
            printf("send() failed: %d\n", WSAGetLastError());
            break;
        }
        else if (ret == 0)
            break;
        // Send() succeeded!
    }
}
else
{
    // Otherwise, use the sendto() function
    //
    for(i = 0; i < dwCount; i++)
    {
            ret = sendto(s, sendbuf, dwLength, 0,
                    (SOCKADDR *)&recipient, sizeof(recipient));
            if (ret == SOCKET_ERROR)
            {
                printf("sendto() failed; %d\n", WSAGetLastError());
                break;
            }
            else if (ret == 0)
                break;
            // sendto() succeeded!
        }
    }
    closesocket(s);
    GlobalFree(sendbuf);
    WSACleanup();
    return 0;
}
```

7.5 其他API函数

本小节介绍其他几个Winsock API函数，它们在实际网络应用中非常有用。

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
);
```

第一个参数是准备连接的套接字，后两个参数则是指向基层协议类型及其长度的指针。对数据报套接字来说，这个函数返回的是投向连接调用的那个地址；但不会返回投向 sendto或WSASendTo调用的那个地址。

2. getsockname

该函数是getsockname的对应函数。它返回的是指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返回的地址信息本地地址信息外，它的参数和getpeername的参数都是一样的。TCP协议中，这个地址和监听指定端口和 IP接口的那个服务器套接字是一样的。

3. WSADuplicateSocket

WSADuplicateSocket函数用来建立WSAPROTOCOL_INFO结构，该结构可投入另一个进程，这样就可用另一个进程打开一个指向同一个基层套接字的句柄，如此一来，另一个进程也能对该资源进行操作。注意，这一点只适用于两个进程之间；同一个进程中的线程可自由投递套接字描述符。该函数的定义如下：

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

第一个参数是准备复制的套接字句柄。第二个参数 dwProcessId，是打算使用复制套接字的进程之ID。第三个参数lpProtocolInfo，是一个指向WSAPROTOCOL_INFO结构的指针，将包含目标进程打开复制句柄时所需的信息。为了使目前的进程能够把 WSAPROTOCOL_INFO结构投到目标进程，然后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同步。所有描述符中都可见到关联到一个套接字的所有状态信息，这是因为复制的是套接字描述符，而不是事实上的套接字。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函数来看它们。如果一个进程在一个复制套接字上调用 closesocket，就会导致该进程中的描述符变成解除定位；但在最后留下的那个描述符上调用 closesocket之前，基层套接字会保持打开状态。

另外，在使用WSAAsyncSelect和WSAEventSelect时，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（我们将在第8章进行讨论）。利用任何一个共享描述符执行前两个函数的调用，都会删掉所有的套接字事件注册，不管注册所用的描述符究竟是哪一个。例如，共享套接字不能把 FD_READ事件投递给进程 A，不能把FD_WRITE投递给进程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。

4. TransmitFile

TransmitFile是微软专有的Winsock扩展，它允许从一个文件中传输高性能数据。这是非常有效的，因为整个数据传输可在内核模式中进行。也就是说，如果你的应用从指定的文件中读取一堆数据，然后用 send或WSASend时，涉及到“用户模式到内核模式传输”的发送调用就有若干个。有了 TransmitFile，整个读取和发送数据的进程就可在内核模式中进行。该函数的定义如下：

```
BOOL TransmitFile(
    SOCKET hSocket,
    HANDLE hFile,
    DWORD nNumberOfBytesToWrite,
    DWORD nNumberOfBytesPerSend,
    LPOVERLAPPED lpOverlapped,
    LPTRANSMIT_FILE_BUFFERS lpTransmitBuffers,
    DWORD dwFlags
);
```

hSocket参数用于识别已连接上的套接字（文件的传输便在该套接字上进行）。nFile参数是一个句柄，该句柄指向一个已打开的套接字（即即将发送的文件）。nNumberOfBytesToWrite表明写入多少指定文件中的字节。投递0表示将发送整个文件。nNumberOfBytesPerSend参数则表明写操作所用的发送长度。例如，指定 2048会引起TransmitFile在套接字上以2 KB数据块的形式发送指定文件。投递0表示采用默认的发送长度。 lpOverlapped参数指定一个OVERLAPPED结构，该结构用于重叠I/O模式（关于重叠I/O，可参见第8章）。

另一个参数lpTransmitBuffers，是一个TRANSMIT_FILE_BUFFERS结构，其中包含文件传输之前和之后准备发送的数据。该结构的格式如下：

```
typedef struct _TRANSMIT_FILE_BUFFERS {
    PVOID Head;
    DWORD HeadLength;
    PVOID Tail;
    DWORD TailLength;
} TRANSMIT_FILE_BUFFERS;
```

Head字段是一个指针，它指向文件传输之前准备发送的数据。 HeadLength表明预先准备发送的数据量。Tail字段则指向文件传输之后准备发送的数据。 TailLength是后来发送的数据量。

TransmitFile的最后一个参数是 dwFlags，用于指定即将对 TransmitFile行为产生影响的标志。表7-2列出了这些标志及其说明。

表7-2 TransmitFile标志

标 志	说 明
TF_DISCONNECT	数据发送完毕之后，开始执行套接字关闭
TF_REUSE_SOCKET	允许套接字句柄再次作为客户机套接字用于 AcceptEx中

(续)

标 志	说 明
TF_USE_DEFAULT_WORKER	表明传输应该在系统默认线程场景中进行。特别有利于长文件传输
TF_USE_SYSTEM_THREAD	表明传输应该在系统线程场景中进行。同样有利于长文件传输
TF_USE_KERNEL_APC	表明“内核异步进程调用”（APC）应该对文件传输进行处理。如果把指定文件读入缓冲区只需要一次读取，该标志便可提供一个重要的性能提升
TF_WRITE_BEHIND	表明TransmitFile应该在无须远程系统对所有的数据进行收到确认的情况下结束

7.6 Windows CE

前面几个小节中的所有信息中都提到了 Windows CE。唯一例外的是由于 Windows CE是建立在Winsock 1.1规格基础上的，因此不能用 Winsock 2中专有的函数，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。我们已经对前三个函数进行了讨论；最后一个留到第9章全面讲解。

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，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函数调用都可用于建立和传输数据。唯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，都会导致内核内存泄漏。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。唯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，而且只采用 Unicode，所以本章介绍的示例都以 Windows 95、Windows 98、Windows NT和Windows 2000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。除非你此时正在编写Windows CE服务，否则你随时都需要一个用户界面。这样，便需要为窗口处理器和其他用户界面元素编写若干新函数，而这些函数又会让大家误入歧途。除此以外，还必须了解Unicode和非Unicode的Winsock函数。至于投给收发 Winsock函数的指定字串是Unicode字串还是ASCII字串，则由程序员决定。只要它是一个有效的缓冲， Winsock就不会注意你投递的内容（当然，可能需要把缓冲造型成静听编译警告）。千万要记住，如果你把一个 Unicode字串造型成char*, 那么准备发送多少字节的长度参数就应该做出相应的调节！在 Windows CE中，如果你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。因为其他的 Win32函数的确要求 Unicode字串。总而言之，对执行一个简单的 Winsock应用而言， Windows CE要复杂得多。

如果想在Windows CE上运行这些示例，只需要稍微修订 Winsock代码即可。首先，头文件必须是 Winsock.h，而不是 Winsock 2.h。WSAStartup应该加载 1.1版，因为它是适用于 Windows CE的最新Winsock版。另外， Windows CE不支持控制台协议；因而必须用 WinMain来代替 main。注意，这并不意味着需要你把一个窗口合并到你的应用中；只意味着不能用 printf这一类的控制台文本I/O函数。

7.7 其他地址家族

本章中介绍的所有Winsock API函数都与协议无关。也就是说，这里列出的用法也可轻易用于Win32平台支持的其他协议。下面的小节只是针对配套光盘上的其他协议家族，说明一下光盘上的客户机／服务器实例代码。

7.7.1 AppleTalk

独立的AppleTalk实例对基本的客户机／服务器技术进行了解释。该实例均提供对AppleTalk PAP和ADSP协议的支持。PAP协议是面向消息的、无连接的、不可靠的协议，类似于UDP，但和UDP又有两点显著不同。首先，它支持部分消息，这意味着调用 WSARecvEX可能会返回数据报消息的一部分。这时必须检查返回的 MSG_PARTIAL标志，看是否还有要求获得完整消息的别的调用。第二点不同之处是在每次读入取操作之前，必须设置一个 PAP协议专有的套接字选项。SO_PRIME_READ这个选项随setsockopt函数一起使用，关于这一选项，我们将在第9章讨论。我们来看看光盘上的 Atalk.c实例，它对如何检查MSG_PARTIAL标志以及如何使用SO_PRIME_READ选项进行了解释。

ADSP协议是一个面向连接的、流式的、可靠的协议 —— 和TCP极其相似。基本的AppleTalk API调用仍然和本章介绍的UDP和TCP实例中的调用差不多。唯一的区别是名字解析。记住，在AppleTalk中，在查找和注册一个AppleTalk名之前，必须先绑定一个空地址。关于这一点，我们曾在第6章的“ AppleTalk定址 ”小节中详细讲过。

AppleTalk协议有一个限制。 Winsock 1.1中原来的AppleTalk支持，在Winsock2开发后，AppleTalk好像没有完全融入这些新函数。无论用 WSAsend，还是用WSARecv，都可能返回负字节计数之类的奇怪结果。关于这个问题的详细讨论可参考“ 微软知识库 ”文章 Q164565。唯一例外的是WSARecvEx，除了flags参数是输入／输出这一点不同外，它实际上是一个 recv调用，在返回之后被MSG_PARTIAL标志查询所用。

7.7.2 IrDA

红外线协议是近来新增的，可用于 Windows CE、Windows 98和Windows 2000。它只提供一个协议类型，是一个面向连接的、流式的、可靠的协议。再次提醒大家注意，代码中主要的不同之处是名字解析，它和 IP协议中采用的名字解析截然不同。大家还应该知道另一个不同之处是：由于Windows CE只支持Winsock 1.1规格，因此，在Windows CE平台的红外线套接字上，只能用Winsock 1.1规格中的函数。在Windows 98和Windows 2000平台上，就可以用Winsock 2规格中专有的函数了。光盘上的实例代码只采用了 Winsock 1.1函数。当然，在Windows 98和Windows 2000平台上，必须加载2.2版本或更新版本的Winsock库，这是因为2.2之前的版本不支持AF_IRDA地址家族。

至于红外线套接字的实例代码，可在下列文件中找到： Ircommon.h、Ircommon.c、Irclient.c以及Irserver.c。前面两个文件只定义了两个常见的函数，一个用于发送数据，另一个用于接收数据，两者均可用于服务器和客户机。客户机端的详细说明在 Irclient.c文件中，这个文件很简单。现列出范围内的红外线设备。然后用指定的服务名在各设备之间建立连接。从第一个接受连接请求的设备着手。随后，客户机发出数据，然后再读取返回的数据。而处于

连接另一端的服务器，其说明则在 Irserver.c文件中。服务器只建立一个红外线套接字，把指定的服务名和这个套接字绑定在一起，然后等待客户机连接请求。对每个客户机来说，就是生成一个线程来接收数据，并把它发回客户机。

注意，这些实例都是用Windows 98和Windows 2000编写的。和TCP/IP实例一样，只须做出少许修改，它们就可运行于 Windows CE。关于Windows CE，主要有两点要注意：一是不支持控制台应用，二是所有函数（除了 Winsock函数）采用的都是Unicode字串。

7.7.3 NetBIOS

我们曾介绍了几个 Winsock NetBIOS实例。通过第 1章的学习，大家已知道 NetBIOS和Winsock的情形差不多，具备使用几种不同传输协议的能力。第 6章中，我们曾学习了如何列举具有NetBIOS能力的传输协议，以及如何建立以其中任何一种协议为基础的套接字。每个协议与适配器组合都有两个条目： SOCK_DGRAM和SOCK_SEQPACKET。它们分别对应无连接的数据报与流套接字（此类套接字非常接近于 UDP和TCP套接字）。除名字解析之外，NetBIOS Winsock接口和本章前面提到的接口没什么区别。记住，一个编写得好的服务器应该在所有可用的LANA上监听，而且，客户机也可能与所有 LANA上的另一端建立连接。

光盘上的前两个示例是 Wsnbsvr.c和Wsnbclnt.c。它们采用的是 SOCK_SEQPACKET套接字类型。据程序员看来，这一类型是面向流的。服务器为每个 LANA一一建立套接字，可用WSAEnumProtocols函数把每个LANA都列出来。然后，服务器把已建立的套接字绑定到服务器的“已知”名。客户机连接一旦建立，服务器就会建立一个线程对该连接进行处理，至此，建立的这个线程只读取接入的数据，并把数据返射到客户机。类似地，客户机试着与所有的LANA建立连接。第一个连接成功之后，其他套接字就会关闭。然后，客户机向服务器发送数据，服务器向它返回数据。

另一个例子是Wsnbdgs.c，它是一个数据报或SOCK_DGRAM示例。该示例中包含用于收发数据报消息的代码。它是无连接的，因此，发给服务器的消息通过所有能用的传输协议进行发送（因为事先不知道哪个或哪些传输协议可以抵达服务器）。除此以外，这个示例还支持单一数据、组或广播数据（都在第 1章讨论过）。

7.7.4 IPX/SPX

IPX/SPX示例Sockspx.c解释了如何利用IPX协议以及流和序列包 SPXII。这个示例把这三个协议的发送端和接收端结合在一起。所用的特殊协议是通过 -P命令行选项来指定的。该示例简单易学。主函数对这个命令行参数进行解析，然后再调用 Server或Client函数。这一点对面向连接的SPXII协议来说，意味着在客户机试着与命令行上指定的服务器之间建立连接期间，服务器将套接字与内部网络地址绑定在一起，并等候客户机连续请求。连接一旦建立，就可以普通形式收发数据了。

对无连接的 IPX协议来说，这个示例更简单。服务器只绑定内部网络地址，通过调用recvfrom函数，等候接入数据。客户机通过 sendto函数，向命令行上指定的接收端发送数据。

这个示例的两部分都需要一些说明。其一是 FillIpxAddress函数，它负责把命令行上指定的ASCII IPX地址编码为SOCKADDR_IPX结构。正如大家在第 6章所见到的那样，IPX地址的表达方式是十六进制的字串，这意味着和 SOCKADDR_IPX结构中的地址字段比较起来， IPX

地址中的每个十六进制字符实际上要占 4位。FillIpxAddress选择了IPX地址，随后调用另一个函数 —— AtoH，即事实上执行地址转换的函数。

其二便是EnumerateAdapters函数，命令行中有 -m标志时，就会执行这个函数。它用套接字选项 IPX_MAX_ADAPTER_NUM来查对可用的本地 IPX地址有多少，随后，便调用 IPX_ADDRESS套接字选项来获得这些地址。这些套接字选项及其参数都将在第 9章讨论。我们之所以用这些选项，其原因是我们的示例采用直接 IPX地址，不执行任何名字解析。第 10章将对IPX可能采用的名字注册和名字解析一一剖析。

7.7.5 ATM

Windows 98和Windows 2000上，都可从 Winsock访问 ATM协议。 ATM示例包含在 Wsockatm.c、Support.c和Support.h这三个文件中。后面两个文件只包含了 Wsockatm.c所用的支持例程，比如说本地 ATM地址列举和ATM地址编码。ATM地址是十六进制编码，因此，就像对待 IPX地址那样，我们采用了同一个的 AtoH函数。另外，还用套接字 I/O控制命令 SIO_GET_NUMBER_OF_ATM_DEVICES来获得本地ATM接口的数量，然后再用I/O控制命令 SIO_GET_ATM_ADDRESS来获得真正的地址。这两个I/O控制命令将在第9章进行描述。

另外，客户机和服务器两端都是在 Wsockatm.c中实施的。由于ATM只支持面向连接的通信，所以这个示例不是很长，多数代码都在 main函数中给出。服务器准备绑定一个显式本地接口，并等待客户机连接，客户机连接的处理和监听套接字同在一个线程中进行。这意味着服务器一次只能为一个客户机提供服务。我们这样设计示例的目的是尽量使代码简单化。另一方面，客户机利用服务器的 ATM地址调用 connect函数。客户机和服务器之间的连接一旦建立，就可以在这个连接上发送数据了。

在使用ATM协议时，需要注意这一点：在服务器内执行 WSAAccept调用之后，客户机的地址就出来了。虽然如此，但在服务器收到连接请求时，客户机的地址却是未知的。这是因为在触发accept函数时，连接尚未完全建立。客户机端的情况也如此。客户机执行调用与服务器连接时，即便在连接未完全建立的情况下，调用也可获得成功。这意味着 connect或accept调用完成之后，在连接完全建立之前，试图立即发送数据可能会不成功。不幸的是，对应用而言，无法判断连接是否有效。另外， ATM只支持硬关闭。也就是说，在应用调用 closesocket时，连接会立即中断。对不支持从容关闭的协议来说，这时调用 closesocket，任何一端的套接字上的待发数据一般会被丢弃。这是一个颇受欢迎的行为；然而，对开发人员来说， ATM提供者较适合他们。数据在套接字上等待发送，而另一方上的套接字已被关闭了，这种情况下， Winsock仍然会返回等待对方套接字接收的数据。

7.8 小结

通过第6章的学习，大家已知道了如何为指定协议建立套接字以及如何为协议的地址家族解析主机名。本章对面向连接和无连接协议所需的一些基本的 Winsock函数进行了阐述。针对面向连接的协议，我们学习了如何接受客户机连接、如何建立客户机与服务器的连接。另外还掌握了面向会话的数据收发操作。针对无连接协议，我们学习了如何收发数据。当然，我们还只用了一个I/O模式来介绍了暂停套接字。下一章，我们将全面讨论可用于 Winsock的其他模型。

第8章　Winsock I/O方法

本章重点是如何在 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在随一个套接字调用时，那些 Winsock函数的行为。而另一方面，套接字模型描述了一个应用程序如何对套接字上进行的 I/O进行管理及处理。要注意的是，“套接字I/O模型”与“套接字模式”是无关的。套接字模型的出现，正是为了解决套接字模式存在的某些限制。

Winsock提供了两种套接字模式：锁定和非锁定。本章第一部分将详细介绍这两种模式，并阐释一个应用程序如何通过它们管理I/O。如大家在本章的后面部分所见， Winsock提供了一些有趣的I/O模型，有助于应用程序通过一种“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（重叠式I/O）以及Completion port（完成端口）等等。到本章结束时，我们打算对各种套接字模式以及 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平台都支持套接字以锁定或非锁定方式工作。然而，并非每种平台都支持每一种I/O模型。如表 8-1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，唯一的例外便是I/O完成端口。而到了Windows NT和最新发布的Windows 2000中，每种I/O模型都是支持的。

表8-1　操作系统对套接字I/O模型的支持情况

平　台	select	WSAAsync Select	WSAEvent Select	Overlapped	Completion Port
Windows CE	支持	不支持	不支持	不支持	不支持
Windows 95（Winsock 1）	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	不支持
Windows 98	支持	支持	支持	支持	不支持
Windows NT	支持	支持	支持	支持	支持
Windows 2000	支持	支持	支持	支持	支持

8.1　套接字模式

就像我们前面提到的那样， Windows套接字在两种模式下执行 I/O操作：锁定和非锁定。在锁定模式下，在I/O操作完成前，执行操作的Winsock函数（比如send和recv）会一直等候下去，不会立即返回程序（将控制权交还给程序）。而在非锁定模式下， Winsock函数无论如何都会立即返回。在Windows CE和Windows 95（安装Winsock 1）平台上运行的应用程序仅支持极少的I/O模型，所以我们必须采取一些适当的步骤，让锁定和非锁定套接字能够满足各种场合的要求。

8.1.1 锁定模式

对于处在锁定模式的套接字，我们必须多加留意，因为在一个锁定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”。大多数Winsock应用都是遵照一种“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，然后以它为基础执行一些计算。程序清单 8-1展示的代码片断便是一个典型的例子。

程序清单8-1 简单的锁定模式示例

```
SOCKET  sock;
char    buff[256];
int     done = 0;

...

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，假如没有数据处于“待决”状态，那么 recv函数可能永远都无法返回。这是由于从语句可以看出：只有从系统的输入缓冲区中读回点什么东西，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)，在系统的缓冲区中，事先“偷看”是否存在足够的字节数量。然而，在不实际读入数据的前提下，仅仅“偷看”数据（如实际读入数据，便会将其从系统缓冲区中将其删除），可不是一件光彩的事情。我们认为，这是一种非常不好的编程习惯，应尽全力避免。在“偷看”的时候，对系统造成的开销是极大的，因为仅仅为了检查有多少个字节可用，便发出一个或者更多的系统调用。以后，理所当然地，还需要牵涉到进行实际 recv调用，将数据从系统缓冲区内删除的开销。那么，如何避免这一情况呢？在此，我们的目标是防止由于数据的缺乏（这可能是网络出了故障，也可能是客户机出了问题），造成应用程序完全陷于“凝固”状态，同时不必连续性地检视系统网络缓冲！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，比如一个事件或者 Mutex（互斥体）。“读线程”的职责是从网络连续地读入数据，并将其置入共享缓冲区内。读线程将计算线程开始工作至少需要的数据量拿到手后，便会触发一个事件，通知计算线程：你老兄可以开始干活了！随后，计算线程从缓冲区取走（删除）一个数据块，然后进行要求的计算。

在程序清单8-2中，我们分别提供了两个函数，采取的便是上述办法。在两个函数中，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）。

程序清单8-2 多线程的锁定套接字示例

```
// Initialize critical section (data) and create
// an auto-reset event (hEvent) before creating the
// two threads
CRITICAL_SECTION data;
HANDLE           hEvent;
TCHAR            buff[MAX_BUFFER_SIZE];
int              nbytes;
...
// Reader thread
void ReadThread(void)
{
    int nTotal = 0,
        nRead = 0,
        nLeft = 0,
        nBytes = 0;

    while (!done)
    {
        nTotal = 0;
        nLeft = NUM_BYTES_REQUIRED;
        while (nTotal != NUM_BYTES_REQUIRED)
        {
            EnterCriticalSection(&data);
            nRead = recv(sock, &(buff[MAX_BUFFER_SIZE - nBytes]),
                nLeft);
            if (nRead == -1)
            {
                printf("error\n");
                ExitThread();
            }
            nTotal += nRead;
            nLeft -= nRead;

            nBytes += nRead;
            LeaveCriticalSection(&data);
        }
        SetEvent(hEvent);
    }
}

// Computation thread
void ProcessThread(void)
{
    WaitForSingleObject(hEvent);

    EnterCriticalSection(&data);
    DoSomeComputationOnData(buff);

    // Remove the processed data from the input
```

```
        // buffer, and shift the remaining data to
        // the start of the array
        nBytes -= NUM_BYTES_REQUIRED;

        LeaveCriticalSection(&data);
    }
```

对锁定套接字来说，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。使用前述的办法，我们可对应用程序进行修改，令其为连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。唯一的缺点便是扩展性极差，以后想同时处理大量套接字时，恐怕难以下手。

8.1.2 非锁定模式

除了锁定模式，我们还可考虑采用非锁定模式的套接字。尽管这种套接字在使用上存在着些许难度，但只要排除了这项困难，它在功能上还是非常强大的。除具备锁定套接字已有的各项优点之外，还进行了少许扩充，功能更强。程序清单 8-3向大家展示了如何创建一个套接字，并将其置为非锁定模式。

程序清单8-3 设置一个非锁定套接字

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

s = socket(AF_INET, SOCK_STREAM, 0);
nRet = ioctlsocket(s, FIOBIO, (unsigned long *) &ul);
if (nRet == SOCKET_ERROR)
{
    // Failed to put the socket into nonblocking mode
}
```

将一个套接字置为非锁定模式之后， Winsock API调用会立即返回。大多数情况下，这些调用都会“失败”，并返回一个WSAEWOULDBLOCK错误。什么意思呢？它意味着请求的操作在调用期间没有时间完成。举个例子来说，假如在系统的输入缓冲区中，尚不存在“待决”的数据，那么recv（接收数据）调用就会返回 WSAEWOULDBLOCK错误。通常，我们需要重复调用同一个函数，直至获得一个成功返回代码。在表 8-2中，我们对常见Winsock调用返回的WSAEWOULDBLOCK错误的含义进行了总结。

由于非锁定调用会频繁返回 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连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”一个锁定套接字相比，前一种做法根本没有任何优势可言。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。

锁定和非锁定套接字模式都存在着优点和缺点。其中，从概念的角度说，锁定套接字更易使用。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，假如需要编写更多的代码，以便在每个 Winsock调用中，对收到一个

WSAEWOULDBLOCK错误的可能性加以应付，那么非锁定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”，它有助于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。

表8-2 非锁定套接字上的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	大多数情况下，这个错误意味着已随SO_LINGER选项一道，调用了setsockopt，而且已设定了一个非零的超时值
WSAConnect和connect	应用程序已初始化。再次调用，便可检查是否完成
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

8.2 套接字I/O模型

共有五种类型的套接字I/O模型，可让Winsock应用程序对I/O进行管理，它们包括：select（选择）、WSAAsyncSelect（异步选择）、WSAEventSelect（事件选择）、overlapped（重叠）以及completion port（完成端口）。在这一节里，我们打算向大家解释每种I/O模型的特点，同时讲述如何利用这些模型，来开发自己的应用程序，以便同时管理一个或多个套接字请求。在本书的配套光盘上，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。

8.2.1 select模型

select（选择）模型是Winsock中最常见的I/O模型。之所以称其为“select模型”，是由于它的“中心思想”便是利用select函数，实现对I/O的管理！最初设计该模型时，主要面向的是某些使用Unix操作系统的计算机，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无辜“锁定”的应用程序，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。

利用select函数，我们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，唯一的目的便是防止应用程序在套接字处于锁定模式中时，在一次I/O绑定调用（如send或recv）过程中，被迫进入“锁定”状态；同时防止在套接字处于非锁定模式中时，产生WSAEWOULDBLOCK错误。除非满足事先用参数规定的条件，否则select函数会在进行I/O操作时锁定。select的函数原型如下：

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

其中，第一个参数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。大家可注意到三个 fd_set参数：一个用于检查可读性（readfds），一个用于检查可写性（writefds），另一个用于例外数据（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。
- 连接已经关闭、重设或中止。
- 假如已调用了listen，而且一个连接正在建立，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。
- 如果已完成了对一个非锁定连接调用的处理，连接就会成功。

最后，exceptfds集合包括符合下述任何一个条件的套接字：

- 假如已完成了对一个非锁定连接调用的处理，连接尝试就会失败。
- 有带外（Out-of-band，OOB）数据可供读取。

例如，假定我们想测试一个套接字是否“可读”，必须将自己的套接字增添到readfds集合，再等待select函数完成。select完成之后，必须判断自己的套接字是否仍为 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。在三个参数中（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最多等待I/O操作完成多久的时间。如timeout是一个空指针，那么select调用会无限期地“锁定”或停顿下去，直到至少有一个描述符符合指定的条件后结束。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”。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，返回刚好有未完成的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对套接字进行监视之前，在自己的应用程序中，必须将套接字句柄分配给一个集合，设置好一个或全部读、写以及例外 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，便可知道一个套接字上是否正在发生上述的 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进行处理与检查：

- FD_CLR(s, *set)：从set中删除套接字s。
- FD_ISSET(s, *set)：检查s是否set集合的一名成员；如答案是肯定的是，则返回 TRUE。
- FD_SET(s, *set)：将套接字s加入集合set。
- FD_ZERO(*set)：将set初始化成空集合。

例如，假定我们想知道是否可从一个套接字中安全地读取数据，同时不会陷于无休止的“锁定”状态，便可使用FD_SET宏，将自己的套接字分配给fd_read集合，再来调用select。要想检测自己的套接字是否仍属fd_read集合的一部分，可使用FD_ISSET宏。采用下述步骤，便可完成用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，然后等待在指定的fd_set集合中，I/O活动设置好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，我们的应用程序便可判断出哪些套接字存在着尚未完成（待决）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“待决”的I/O操作之后，对I/O进行处理，然后返回步骤1)，继续进行select处理。

select返回后，它会修改每个fd_set结构，删除那些不存在待决I/O操作的套接字句柄。这正是我们在上述的步骤(4)中，为何要使用FD_ISSET宏来判断一个特定的套接字是否仍在集合中的原因。在程序清单8-4中，我们向大家阐述了为一个（只有一个）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，只需为额外的套接字维护它们的一个列表，或维护它们的一个数组即可。

程序清单8-4 用select管理一个套接字上的I/O操作

```
SOCKET  s;
fd_set  fdread;
int     ret;

// Create a socket, and accept a connection
    -
// Manage I/O on the socket
while(TRUE)
{
    // Always clear the read set before calling
    // select()
    FD_ZERO(&fdread);

    // Add socket s to the read set
    FD_SET(s, &fdread);

    if ((ret = select(0, &fdread, NULL, NULL, NULL))
        == SOCKET_ERROR)
    {
        // Error condition
    }

    if (ret > 0)
    {
        // For this simple case, select() should return
        // the value 1. An application dealing with
        // more than one socket could get a value
        // greater than 1. At this point, your
        // application should check to see whether the
```

```
            // socket is part of a set.

            if (FD_ISSET(s, &fdread))
            {
                // A read event has occurred on socket s
            }
        }
    }
```

8.2.2 WSAAsyncSelect

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。该模型最早出现于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“落后”的多任务消息环境。应用程序仍可从这种模型中得到好处，特别是它们用一个标准的 Windows例程（常称为“winproc”），对窗口消息进行管理的时候。该模型亦得到了 Microsoft Foundation Class（微软基本类，MFC）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例程支持函数（ Winproc）。亦可使用一个对话框，为其提供一个对话例程，而非窗口例程，因为对话框本质也是“窗口”。考虑到我们的目的，我们打算用一个简单的窗口来演示这种模型，采用的是一个支持窗口例程。设置好窗口的框架后，便可开始创建套接字，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，它对应于网络事件发生之后，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。该消息会投递到由 hWnd窗口句柄指定的那个窗口。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，避免网络窗口消息与预定义的标准窗口消息发生混淆与冲突。最后一个参数是 lEvent，它指定的是一个位掩码，对应于一系列网络事件的组合（请参考表8-3），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呢，还是一个服务器。如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。举个例子来说：

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

这样一来，我们的应用程序以后便可在套接字 s上，接收到有关连接、发送、接收以及套接字关闭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注册！另外还要注意的是，一旦在某个套接字上允许了事件通知，那么以后除非明确调用closesocket命令，或者由应用程序针对那个套接字调用了 WSAAsyncSelect，从而更改了注册的网络事件类型，否则的话，事件通知会永远有效！若将 lEvent参数设为0，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“锁定”自动变成“非锁定”，我们在前面已提到过这一点。这样一来，假如调用了像WSARecv这样的Winsock I/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，来判断网络事件类型何时在套接字上发生；而不应盲目地进行调用。

表8-3 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带外（OOB）数据抵达的通知
FD_ACCEPT	应用程序想接收与进入连接有关的通知
FD_CONNECT	应用程序想接收与一次连接或者多点 join操作完成的通知
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“服务质量”（QoS）发生更改的通知
FD_GROUP_QOS	应用程序想接收套接字组“服务质量”发生更改的通知（现在没什么用处，为未来套接字组的使用保留）
FD_ROUTING_INTERFACE_CHANGE	应用程序想接收在指定的方向上，与路由接口发生变化的通知
FD_ADDRESS_LIST_CHANGE	应用程序想接收针对套接字的协议家族，本地地址列表发生变化的通知

应用程序在一个套接字上成功调用了WSAAsyncSelect之后，应用程序会在与hWnd窗口句柄参数对应的窗口例程中，以Windows消息的形式，接收网络事件通知。窗口例程通常定义如下：

```
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，对窗口例程的调用正是由那个窗口发出的。 uMsg参数指定需要对哪些消息进行处理。就我们的情况来说，感兴趣的是 WSAAsyncSelect调用中定义的消息。wParam参数指定在其上面发生了一个网络事件的套接字。假若同时为这个窗口例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。

网络事件消息抵达一个窗口例程后，应用程序首先应检查 lParam的高字位，以判断是否在套接字上发生了一个网络错误。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之低字位的内容。此时可使用另一个特殊的宏：WSAGETSELECTEVENT，用它返回lParam的低字部分。

在程序清单8-5中，我们向大家演示了如何使用WSAAsyncSelect这种I/O模型，来实现窗口消息的管理。在源程序中，我们着重强调的是开发一个基本服务器应用要涉及到的基本步骤，忽略了开发一个完整的Windows应用需要涉及到的大量编程细节。

程序清单8-5 WSAAsyncSelect服务器示范代码

```
#define WM_SOCKET WM_USER + 1
#include <windows.h>

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

    // Create a window and assign the ServerWinProc
    // below to it

    Window = CreateWindow();
    // Start Winsock and create a socket

    WSAStartup(...);
    Listen = Socket();

    // Bind the socket to port 5150
    // and begin listening for connections

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

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

    // Set up window message notification on
    // the new socket using the WM_SOCKET define
    // above

    WSAAsyncSelect(Listen, Window, WM_SOCKET,
        FD_ACCEPT | FD_CLOSE);

    listen(Listen, 5);

    // Translate and dispatch window messages
    // until the application terminates
}
```

```
BOOL CALLBACK ServerWinProc(HWND hDlg,WORD wMsg,
    WORD wParam, DWORD lParam)
{
    SOCKET Accept;

    switch(wMsg)
    {
        case WM_PAINT:
            // Process window paint messages
            break;

        case WM_SOCKET:

            // Determine whether an error occurred on the
            // socket by using the WSAGETSELECTERROR() macro

            if (WSAGETSELECTERROR(lParam))
            {
                // Display the error and close the socket
                closesocket(wParam);
                break;
            }

            // Determine what event occurred on the
            // socket

            switch(WSAGETSELECTEVENT(lParam))
            {
                case FD_ACCEPT:

                    // Accept an incoming connection
                    Accept = accept(wParam, NULL, NULL);

                    // Prepare accepted socket for read,
                    // write, and close notification

                    WSAAsyncSelect(Accept, hwnd, WM_SOCKET,
                        FD_READ | FD_WRITE | FD_CLOSE);
                    break;

                case FD_READ:

                    // Receive data from the socket in
                    // wParam
                    break;

                case FD_WRITE:

                    // The socket in wParam is ready
                    // for sending data
```

```
                    break;

                case FD_CLOSE:

                    // The connection is now closed
                    closesocket(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。只有在三种条件下，才会发出FD_WRITE通知：

- 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

因此，作为一个应用程序，自收到首条 FD_WRITE消息开始，便应认为自己必然能在一个套接字上发出数据，直至一个 send、WSASend、sendto或WSASendTo返回套接字错误WSAEWOULDBLOCK。经过了这样的失败以后，要再用另一条 FD_WRITE通知应用程序再次发送数据。

8.2.3 WSAEventSelect

Winsock提供了另一个有用的异步 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 8-3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。在用新模型开发的应用程序中，也能接收和处理所有那些事件。该模型最主要的差别在于网络事件会投递至一个事件对象句柄，而非投递至一个窗口例程。

事件通知

事件通知模型要求我们的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，就是一个创建好的事件对象句柄。事件对象句柄到手后，接下来必须将其与某个套接字关联在一起，同时注册自己感兴趣的网络事件类型，如表8-3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参数代表自己感兴趣的套接字。 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一个组合（如表 8-3所示）。要想获知对这些事件类型的详细说明，请参考早先讨论过的 WSAAsyncSelect I/O模型。

为WSAEventSelect创建的事件拥有两种工作状态，以及两种工作模式。其中，两种工作状态分别是“已传信”（signaled）和“未传信”（nonsignaled）。工作模式则包括“人工重设”（manual reset）和“自动重设”（auto reset）。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。随着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，我们的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);
```

该函数唯一的参数便是一个事件句柄；基于调用是成功还是失败，会分别返回 TRUE或FALSE。应用程序完成了对一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);
```

该函数也要拿一个事件句柄作为自己唯一的参数，并会在成功后返回 TRUE，失败后返回FALSE。

一个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；方法是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。在这个数组中，cEvents指定的是事件对象的数量，而lphEvents对应的是一个指针，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，在此定义成 64个。因此，针对发出 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不止 64个套接字，必须创建额外的工作者线程，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，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最多可

等待一个网络事件发生有多长时间，以毫秒为单位，这是一项“超时”设定。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。这样一来，应用程序实际便可实现对事件对象的“轮询”。但考虑到它对性能造成的影响，还是应尽量避免将超时值设为0。假如没有等待处理的事件，WSAWaitForMultipleEvents便会返回WSA_WAIT_TIMEOUT。如dwsTimeout设为WSA_INFINITE（永远等待），那么只有在一个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，在我们使用WSAEventSelect模型的时候，它是可以忽略的，且应设为FALSE。该参数主要用于在重叠式I/O模型中，在完成例程的处理过程中使用。本章后面还会对此详述。

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。这样一来，我们的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，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，调查发生了什么类型的网络事件。该函数定义如下：

```
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个“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，该函数早先已经讨论过了。最后一个参数是lpNetworkEvents，代表一个指针，指向WSANETWORKEVENTS结构，用于接收套接字上发生的网络事件类型以及可能出现的任何错误代码。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，对应于套接字上发生的所有网络事件类型（参见表8-3）。

注意 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，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，名字与事件类型的名字类似，只是要在事件名字后面添加一个“_BIT”后缀字串即可。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。下述代码片断对此进行了阐释（针对FD_READ事件）：

```
// Process FD_READ notification
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，我们的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。在程序清单 8-6中，我们阐释了如何使用WSAEventSelect这种I/O模型，来开发一个服务器应用，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要涉及到的步骤，令其同时负责一个或多个套接字的管理。

程序清单8-6 采用WSAEventSelect I/O模型的示范服务器源代码

```
SOCKET  Socket[WSA_MAXIMUM_WAIT_EVENTS];
WSAEVENT  Event[WSA_MAXIMUM_WAIT_EVENTS];
SOCKET Accept, Listen;
DWORD EventTotal = 0;
DWORD Index;
// Set up a TCP socket for listening on port 5150
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);

Socket[EventTotal] = Listen;
Event[EventTotal] = NewEvent;
EventTotal++;

while(TRUE)
{
    // Wait for network events on all sockets
    Index = WSAWaitForMultipleEvents(EventTotal,
        EventArray, FALSE, WSA_INFINITE, FALSE);

    WSAEnumNetworkEvents(
        SocketArray[Index - WSA_WAIT_EVENT_0],
        EventArray[Index - WSA_WAIT_EVENT_0],
```

```
        &NetworkEvents);

    // Check for FD_ACCEPT messages
    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;
        }

        // Accept a new connection, and add it to the
        // socket and event lists
        Accept = accept(
            SocketArray[Index - WSA_WAIT_EVENT_0],
            NULL, NULL);
        // We cannot process more than
        // WSA_MAXIMUM_WAIT_EVENTS sockets, so close
        // the accepted socket
        if (EventTotal > WSA_MAXIMUM_WAIT_EVENTS)
        {
            printf("Too many connections");
            closesocket(Accept);
            break;
        }

        NewEvent = WSACreateEvent();

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

        Event[EventTotal] = NewEvent;
        Socket[EventTotal] = Accept;
        EventTotal++;

        printf("Socket %d connected\n", Accept);
    }

    // Process FD_READ notification
    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;
        }

        // Read data from the socket
        recv(Socket[Index - WSA_WAIT_EVENT_0],
            buffer, sizeof(buffer), 0);
    }

    // Process FD_WRITE notification
```

```
    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(Socket[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(Socket[Index - WSA_WAIT_EVENT_0]);

        // Remove socket and associated event from
        // the Socket and Event arrays and decrement
        // EventTotal
        CompressArrays(Event, Socket, &EventTotal);
    }
}
```

8.2.4 重叠模型

在Winsock中，相比我们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一个重叠的数据结构，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Win32重叠I/O机制为基础。那个机制可通过 ReadFile和WriteFile两个函数，针对设备执行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。作为应用程序，它可针对一个套接字句柄，调用 ReadFile以及WriteFile，同时指定一个重叠式结构（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，比如WSASend和WSARecv。这样一来，重叠I/O模型便能适用于安装了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两个函数使用。但是，Windows 95和Windows 98均不具备这一功能。考虑到应用程序的跨平台兼容问题，同时考虑到性能方面的因素，应尽量避免使用Win32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，首先必须使用 WSA_FLAG_OVERLAPPED这个标志，创建一个套接字。如下所示：

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

创建套接字的时候，假如使用的是 socket函数，而非WSASocket函数，那么会默认设置WSA_FLAG_OVERLAPPED标志。成功建好一个套接字，同时将其与一个本地接口绑定到一起后，便可开始进行重叠 I/O操作，方法是调用下述的 Winsock函数，同时指定一个WSAOVERLAPPED结构（可选）：

- WSASend
- WSASendTo
- WSARecv
- WSARecvFrom
- WSAIoctl
- AcceptEx
- TrnasmitFile

大家现在或许已经知道，其中每个函数都与一个套接字上数据的发送、数据接收以及连接的接受有关。因此，这些活动可能会花极少的时间才能完成。这正是每个函数都可接受一个WSAOVERLAPPED结构作为参数的原因。若随一个 WSAOVERLAPPED结构一起调用这些函数，函数会立即完成并返回，无论套接字是否设为锁定模式（本章开头已详细讲过了）。它们依赖于WSAOVERLAPPED结构来返回一个I/O请求的返回。主要有两个方法可用来管理一个重叠I/O请求的完成：我们的应用程序可等待“事件对象通知”，亦可通过“完成例程”，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：lpCompletionROUTINE。该参数指定的是一个可选的指针，指向一个完成例程函数，在重叠请求完成后调用。接下去，我们将探讨事件通知方法。在本章稍后，还会介绍如何使用可选的完成例程，代替事件，对完成的重叠请求加以处理。

1. 事件通知

重叠I/O的事件通知方法要求将Win32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调用会以失败告终，返回 SOCKET_ERROR。使用WSAGetLastError函数，便可获得与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，我们的应用程序需要等候与 WSAOVERLAPPED结构对应的事件对象，了解一个重叠I/O请示何时完成。WSAOVERLAPPED结构在一个重叠I/O请求的初始化，及其后续的完成之间，提供了一种沟通或通信机制。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，不应由应用程序直接进行处理或使用。而另一方面， hEvent字段有点儿特殊，它允许应用程序将一个事件对象句柄同一个套接字关联起来。大家可能会觉得奇怪，如何将一个事件对象句柄分配给该字段呢？正如我们早先在WSAEventSelect模型中讲述的那样，可用WSACreateEvent函数来创建一个事件对象句柄。一旦创建好一个事件句柄，简单地将重叠结构的 hEvent字段分配给事件句柄，再使用重叠结构，调用一个Winsock函数即可，比如WSASend或WSARecv。

一个重叠I/O请求最终完成后，我们的应用程序要负责取回重叠 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会更改与一个WSAOVERLAPPED结构对应的一个事件对象的事件传信状态，将其从“ 未传信 ”变成“ 已传信 ”。由于一个事件对象已分配给WSAOVERLAPPED结构，所以只需简单地调用WSAWaitForMultipleEvents函数，从而判断出一个重叠I/O调用在什么时候完成。该函数已在我们前面讲述 WSAEventSelect I/O模型时介绍过了。 WSAWaitForMultipleEvents函数会等候一段规定的时间，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。发现一次重叠请求完成之后，接着需要调用 WSAGetOverlapped Result（取得重叠结构）函数，判断那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参数用于指定在重叠操作开始的时候，与之对应的那个套接字。 lpOverlapped参数是一个指针，对应于在重叠操作开始时，指定的那个 WSAOVERLAPPED结构。lpcbTransfer参数也是一个指针，对应一个 DWORD（双字）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一次待决（未决）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“ 待决 ”状态，那么 WSAGetOverlappedResult函数会返回FALSE值，同时返回一个WSA_IO_INCOMPLETE（I/O操作未完成）错误。但就我们目前的情况来说，由于需要等候重叠操作的一个已传信事件完成，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，它对应于一个指针，指向一个 DWORD（双字），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。这意味着我们的重叠 I/O操作已成功完成，而且由 lpcbTransfer参数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“ 待决 ”状态。
- 重叠操作已经完成，但含有错误。
- 重叠操作的完成状态不可判决，因为在提供给 WSAGetOverlappedResult函数的一个或多个参数中，存在着错误。

失败后，由 lpcbTransfer参数指向的值不会进行更新，而且我们的应用程序应调用WSAGetLastError函数，调查到底是何种原因造成了调用失败。

在程序清单8-7中，我们向大家阐述了如何编制一个简单的服务器应用，令其在一个套接字上对重叠I/O操作进行管理，程序完全利用了前述的事件通知机制。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) 接受一个进入的连接请求。

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) 在套接字上投递一个异步 WSARecv请求，指定参数为WSAOVERLAPPED结构。

注意 函数通常会以失败告终，返回 SOCKET_ERROR错误状态WSA_IO_PENDING (I/O操作尚未完成)。

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（换言之，等待那个事件的“触发”）。

6) WSAWaitForMultipleEvents函数完成后，针对事件数组，调用 WSAResetEvent（重设事件）函数，从而重设事件对象，并对完成的重叠请求进行处理。

7) 使用WSAGetOverlappedResult函数，判断重叠调用的返回状态是什么。

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，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内，让主应用程序线程为附加的连接请求提供服务。

程序清单8-7 采用事件机制的简单重叠I/O处理示例

```
void main(void)
{
    WSABUF DataBuf;
    DWORD EventTotal = 0;
    WSAEVENT EventArray[WSA_MAXIMUM_WAIT_EVENTS];
    WSAOVERLAPPED AcceptOverlapped;
    SOCKET ListenSocket, AcceptSocket;

    // Step 1:
    //  Start Winsock and set up a listening socket
    ...

    // Step 2:
    //  Accept an inbound connection
    AcceptSocket = accept(ListenSocket, NULL, NULL);

    // Step 3:
    //  Set up an overlapped structure

    EventArray[EventTotal] = WSACreateEvent();

    ZeroMemory(&AcceptOverlapped,
        sizeof(WSAOVERLAPPED));
```

```
AcceptOverlapped.hEvent = EventArray[EventTotal];

DataBuf.len = DATA_BUFSIZE;
DataBuf.buf = buffer;

EventTotal++;

// Step 4:
//  Post a WSARecv request to begin receiving data
//  on the socket

WSARecv(AcceptSocket, &DataBuf, 1, &RecvBytes,
    &Flags, &AcceptOverlapped, NULL);

// Process overlapped receives on the socket.

while(TRUE)
{
    // Step 5:
    //  Wait for the overlapped I/O call to complete
    Index = WSAWaitForMultipleEvents(EventTotal,
        EventArray, FALSE, WSA_INFINITE, FALSE);

    // Index should be 0 because we
    // have only one event handle in EventArray

    // Step 6:
    //  Reset the signaled event
    WSAResetEvent(
        EventArray[Index - WSA_WAIT_EVENT_0]);

    // Step 7:
    //  Determine the status of the overlapped
    //  request
    WSAGetOverlappedResult(AcceptSocket,
        &AcceptOverlapped, &BytesTransferred,
        FALSE, &Flags);

    // First check to see whether the peer has closed
    // the connection, and if so, close the
    // socket

    if (BytesTransferred == 0)
    {
        printf("Closing socket %d\n", AcceptSocket);

        closesocket(AcceptSocket);
            WSACloseEvent(
                EventArray[Index - WSA_WAIT_EVENT_0]);
            return;
        }

        // Do something with the received data.
        // DataBuf contains the received data.
        ...
```

```
            // Step 8:
            //  Post another WSARecv() request on the socket

            Flags = 0;
            ZeroMemory(&AcceptOverlapped,
                sizeof(WSAOVERLAPPED));

            AcceptOverlapped.hEvent = EventArray[Index -
                WSA_WAIT_EVENT_0];

            DataBuf.len = DATA_BUFSIZE;
            DataBuf.buf = Buffer;

            WSARecv(AcceptSocket, &DataBuf, 1,
                &RecvBytes, &Flags, &AcceptOverlapped,
                NULL);
        }
    }
```

在Windows NT和Windows 2000中，重叠I/O模型也允许应用程序以一种重叠方式，实现对连接的接受。具体的做法是在监听套接字上调用 AcceptEx函数。AcceptEx是一个特殊的Winsock 1.1扩展函数，位于Mswsock.h头文件以及Mswsock.lib库文件内。该函数最初的设计宗旨是在Windows NT与Windows 2000操作系统上，处理Win32的重叠I/O机制。但事实上，它也适用于Winsock 2中的重叠I/O。AcceptEx的定义如下：

```
BOOL AcceptEx (
    SOCKET sListenSocket,
    SOCKET sAcceptSocket,
    PVOID lpOutputBuffer,
    DWORD dwReceiveDataLength,
    DWORD dwLocalAddressLength,
    DWORD dwRemoteAddressLength,
    LPDWORD lpdwBytesReceived,
    LPOVERLAPPED lpOverlapped
);
```

sListenSocket参数指定的是一个监听套接字。 sAcceptSocket参数指定的是另一个套接字，负责对进入连接请求的“接受”。AcceptEx函数和accept函数的区别在于，我们必须提供接受的套接字，而不是让函数自动为我们创建。正是由于要提供套接字，所以要求我们事先调用socket或WSASocket函数，创建一个套接字，以便通过sAcceptSocket参数，将其传递给AcceptEx。lpOutputBuffer参数指定的是一个特殊的缓冲区，因为它要负责三种数据的接收：服务器的本地地址，客户机的远程地址，以及在新建连接上发送的第一个数据块。 dwReceiveDataLength参数以字节为单位，指定了在lpOutputBuffer缓冲区中，保留多大的空间，用于数据的接收。如这个参数设为0，那么在连接的接受过程中，不会再一道接收任何数据。 dwLocalAddressLength和dwRemoteAddressLength参数也是以字节为单位，指定在lpOutputBuffer缓冲区中，保留多大的空间，在一个套接字被接受的时候，用于本地和远程地址信息的保存。要注意的是，和当前采用的传送协议允许的最大地址长度比较起来，这里指定的缓冲区大小至少应多出 16字节。举个例子来说，假定正在使用的是TCP/IP协议，那么这里的大小应设为“SOCKADDR_IN结构的长度＋16字节”。lpdwBytesReceived参数用于返回接收到的实际数据量，以字节为单位。只有在操作以同步方式完成的前提下，才会设置这个参数。假如AcceptEx函数返回ERROR_IO_PENDING，

那么这个参数永远都不会设置，我们必须利用完成事件通知机制，获知实际读取的字节量。最后一个参数是lpOverlapped，它对应的是一个OVERLAPPED结构，允许AcceptEx以一种异步方式工作。如我们早先所述，只有在一个重叠I/O应用中，该函数才需要使用事件对象通知机制，这是由于此时没有一个完成例程参数可供使用。

一个名为GetAcceptExSockaddrs的Winsock扩展函数可从lpOutputBuffer中解析出本地和远程地址元素。GetAcceptExSockaddrs定义如下：

```
VOID GetAcceptExSockaddrs(
    PVOID lpOutputBuffer,
    DWORD dwReceiveDataLength,
    DWORD dwLocalAddressLength,
    DWORD dwRemoteAddressLength,
    LPSOCKADDR *LocalSockaddr,
    LPINT LocalSockaddrLength,
    LPSOCKADDR *RemoteSockaddr,
    LPINT RemoteSockaddrLength
);
```

lpOutputBuffer参数应设为自 AcceptEx返回的lpOutputBuffer。dwReceiveDataLength、dwLocalAddressLength以及dwRemoteAddressLength参数应设为与我们传递给 AcceptEx的dwReceiveDataLength、dwLocalAddressLength以及dwRemoteAddressLength参数相同的值。LocalSockaddr和RemoteSockaddr参数分别是指向一个包含了本地及远程地址信息的SOCKADDR结构的指针，负责从最初的 lpOutputBuffer参数接收一个指针偏移。这样一来，根据lpOutputBuffer中包含的地址信息，我们可以非常轻松地对一个 SOCKADDR结构中的元素进行引用。LocalSockaddrLength和RemoteSockaddrLength参数则分别负责接收本地及远程地址的长度。

2. 完成例程

“完成例程”是我们的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。最开始的时候，我们将其传递给一个重叠 I/O请求，在一个重叠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，为一个已完成的 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处理。

如果希望用完成例程为重叠I/O请求提供服务，在我们的应用程序中，必须为一个 I/O定界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
);
```

用完成例程完成了一个重叠I/O请求之后，参数中会包含下述信息：

- 参数dwError表明了一个重叠操作（由lpOverlapped指定）的完成状态是什么。
- cbTransferred参数指定了在重叠操作期间，实际传输的字节量是多大。
- lpOverlapped参数指定的是传递到最初的I/O调用内的一个WSAOVERLAPPED结构。
- dwFlags参数目前尚未使用，应设为0。

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，存在着一项非常重要的区别。WSAOVERLAPPED结构的事件字段hEvent并未使用；也就是说，我们不可将一个事件对象同重叠请求关联到一起。用完成例程发出了一个重叠 I/O调用之后，作为我们的调用线程，一旦完成，它最终必须为完成例程提供服务。这样一来，便要求我们将自己的调用线程置于一种“可警告的等待状态”。并在I/O操作完成后，对完成例程加以处理。WSAWaitForMultipleEvents函数可用来将我们的线程置于一种可警告的等待状态。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。假定应用程序只用完成例程对重叠请求进行处理，便不可能有任何事件对象需要处理。作为一种变通方法，我们的应用程序可用 Win32的SleepEx函数将自己的线程置为一种可警告等待状态。当然，亦可创建一个伪事件对象，不将它与任何东西关联在一起。假如调用线程经常处于繁忙状态，而且并不处在一种可警告的等待状态，那么根本不会有投递的完成例程会得到调用。

如早先所见，WSAWaitForMultipleEvents通常会等待同WSAOVERLAPPED结构关联在一起的事件对象。该函数也设计用于将我们的线程置入一种可警告等待状态，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）。用一个完成例程结束了重叠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，以毫秒为单位。假如将dwMilliseconds设为INFINITE，那么SleepEx会无休止地等待下去。bAlertable参数规定了一个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完成回调，那么I/O完成函数不会执行，而且函数不会返回，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。

在程序清单8-8中，我们向大家展示了如何构建一个简单的服务器应用，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。该程序的编码主要按下述步骤进行：

1) 新建一个套接字，开始在指定端口上，监听一个进入的连接。

2) 接受一个进入的连接请求。

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，方法是将WSAOVERLAPPED指定成为参数，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，可随一个完成例程一道，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)。

程序清单8-8 采用完成例程的简单重叠I/O处理示例

```
SOCKET AcceptSocket;
WSABUF DataBuf;

void main(void)
{
    WSAOVERLAPPED Overlapped;

    // Step 1:
    //  Start Winsock, and set up a listening socket
    ...

    // Step 2:
    //  Accept a new connection
    AcceptSocket = accept(ListenSocket, NULL, NULL);

    // Step 3:
    //  Now that we have an accepted socket, start
    //  processing I/O using overlapped I/O with a
    //  completion routine. To get the overlapped I/O
    //  processing started, first submit an
    //  overlapped WSARecv() request.

    Flags = 0;

    ZeroMemory(&Overlapped, sizeof(WSAOVERLAPPED));

    DataBuf.len = DATA_BUFSIZE;
    DataBuf.buf = Buffer;
    // Step 4:
    //  Post an asynchronous WSARecv() request
    //  on the socket by specifying the WSAOVERLAPPED
    //  structure as a parameter, and supply
    //  the WorkerRoutine function below as the
    //  completion routine

    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;
        }
    }

    // Since the WSAWaitForMultipleEvents() API
    // requires waiting on one or more event objects,
    // we will have to create a dummy event object.
    // As an alternative, we can use SleepEx()
    // instead.

    EventArray[0] = WSACreateEvent();
```

```
    while(TRUE)
    {
        // Step 5:
        Index = WSAWaitForMultipleEvents(1, EventArray,
            FALSE, WSA_INFINITE, TRUE);

        // Step 6:
        if (Index == WAIT_IO_COMPLETION)
        {
            // An overlapped request completion routine
            // just completed. Continue servicing
            // more completion routines.
            break;
        }
        else
        {
            // A bad error occurred--stop processing!
            // If we were also processing an event
            // object, this could be an index to
            // the event array.
            return;
        }
    }
}

void CALLBACK WorkerRoutine(DWORD Error,
                            DWORD BytesTransferred,
                            LPWSAOVERLAPPED Overlapped,
                            DWORD InFlags)
{
    DWORD SendBytes, RecvBytes;
    DWORD Flags;

    if (Error != 0 || BytesTransferred == 0)
    {
        // Either a bad error occurred on the socket
        // or the socket was closed by a peer
        closesocket(AcceptSocket);
        return;
    }

    // At this point, an overlapped WSARecv() request
    // completed successfully. Now we can retrieve the
    // received data that is contained in the variable
    // DataBuf. After processing the received data, we
    // need to post another overlapped WSARecv() or
    // WSASend() request. For simplicity, we will post
    // another WSARecv() request.

    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;
            }
        }
    }
```

8.2.5 完成端口模型

“完成端口”模型是迄今为止最为复杂的一种I/O模型。然而，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和Windows 2000操作系统。因其设计的复杂性，只有在你的应用程序需要同时管理数百乃至上千个套接字的时候，而且希望随着系统内安装的CPU数量的增多，应用程序的性能也可以线性提升，才应考虑采用“完成端口”模型。要记住的一个基本准则是，假如要为Windows NT或Windows 2000开发高性能的服务器应用，同时希望为大量套接字I/O请求提供服务（Web服务器便是这方面的典型例子），那么I/O完成端口模型便是最佳选择！

从本质上说，完成端口模型要求我们创建一个Win32完成端口对象，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Win32、Windows NT以及Windows 2000采用的一种I/O构造机制，除套接字句柄之外，实际上还可接受其他东西。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巨大威力。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。该函数定义如下：

```
HANDLE CreateIoCompletionPort(
    HANDLE FileHandle,
    HANDLE ExistingCompletionPort,
    DWORD CompletionKey,
    DWORD NumberOfConcurrentThreads
);
```

在我们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明显有别的目的：

- 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，唯一感兴趣的参数便是NumberOfConcurrentThreads（并发线程的数量）；前面三个参数都会被忽略。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。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“场景”切换。若将该参数设为0，表明系统内安装了多少个处理器，便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 = CreateIoCompletionPort(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定（引用）。

1. 工作者线程与完成端口

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“工作者线程”，以便在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，大家或许会觉得奇怪，到底应创建多少个线程，以便为完成端口提供服务呢？这实际正是完成端口模型显得颇为“复杂”的一个方面，因为服务I/O请求所需的数量取决于应用程序的总体设计情况。在此要记住的一个重点在于，在我们调用CreateIoComletionPort时指定的并发线程数量，与打算创建的工作者线程数量相比，它们代表的并非同一件事情。早些时候，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）以避免由于频繁的线程“场景”交换活动，从而影响系统的整体性能。CreateIoCompletionPort函数的NumberOfConcurrentThreads参数明确指示系统：在一个完成端口上，一次只允许n个工作者线程运行。假如在完成端口上创建的工作者线程数量超出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者线程数量有时要比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。假定我们的某个工作者线程调用了一个函数，比如Sleep或WaitForSingleObject，但却进入了暂停（锁定或挂起）状态，那么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。这样一来，假如事先预计到自己的线程有可能暂时处于停顿状态，那么最好能够创建比CreateIoCompletonPort的NumberOfConcurrentThreads参数的值多的线程，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者线程来为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。这要求我们在一个现有的完成端口上，调用CreateIo CompletionPort函数，同时为前三个参数——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参数指定的是一个现有的完成端口。CompletionKey（完成键）参数则指定要与某个特定套接字句柄关联在一起的“单句柄数据”；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”，是由于它只对应着与那个套接字句柄关联在一起的数据。可将其作为指向一个数据结构的指针，来保存套接字句柄；在那个结构中，同时包含了套接字的句柄，以及与那个套接字有关的其他信息。就象本章稍后还会讲述的那样，为完成端口提供服务的线程例程可通过这个参数，取得与套接字句柄有关的信息。

根据我们到目前为止学到的东西，首先来构建一个基本的应用程序框架。程序清单8-9向大家阐述了如何使用完成端口模型，来开发一个回应（或“反射”）服务器应用。在这个程序中，我们基本上按下述步骤行事：

1) 创建一个完成端口。第四个参数保持为0，指定在完成端口上，每个处理器一次只允许

执行一个工作者线程。

2) 判断系统内到底安装了多少个处理器。

3) 创建工作者线程，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者线程。这是由于事先已预计到，到时不会有任何线程进入“挂起”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者例程，由线程在创建好执行。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上监听进入的连接请求。

5) 使用accept函数，接受进入的连接请求。

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。通过完成键（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，一个工作者线程会为I/O请求提供服务，同时继续处理未来的I/O请求，稍后便会在步骤3)指定的工作者例程中，体验到这一点。

9) 重复步骤5)~8)，直至服务器中止。

程序清单8-9 完成端口的建立

```
StartWinsock();

// Step 1:
// Create an I/O completion port

CompletionPort = CreateIoCompletionPort(
    INVALID_HANDLE_VALUE, NULL, 0, 0);

// Step 2:
// Determine how many processors are on the system

GetSystemInfo(&SystemInfo);

// Step 3:
// Create worker threads based on the number of
// processors available on the system. For this
// simple case, we create one worker thread for each
// processor.

for(i = 0; i < SystemInfo.dwNumberOfProcessors;
    i++)
{
    HANDLE ThreadHandle;

    // Create a server worker thread, and pass the
    // completion port to the thread. NOTE: the
    // ServerWorkerThread procedure is not defined
    // in this listing.
```

```
        ThreadHandle = CreateThread(NULL, 0,
            ServerWorkerThread, CompletionPort,
            0, &ThreadID);

        // Close the thread handle
        CloseHandle(ThreadHandle);
    }

    // Step 4:
    // Create a listening socket

    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));

    // Prepare socket for listening

    listen(Listen, 5);

    while(TRUE)
    {
        // Step 5:
        // Accept connections and assign to the completion
        // port

        Accept = WSAAccept(Listen, NULL, NULL, NULL, 0);

        // Step 6:
        // Create per-handle data information structure to
        // associate with the socket
        PerHandleData = (LPPER_HANDLE_DATA)
            GlobalAlloc(GPTR, sizeof(PER_HANDLE_DATA));

        printf("Socket number %d connected\n", Accept);
        PerHandleData->Socket = Accept;

        // Step 7:
        // Associate the accepted socket with the
        // completion port

        CreateIoCompletionPort((HANDLE) Accept,
            CompletionPort, (DWORD) PerHandleData, 0);

        // Step 8:
        //  Start processing I/O on the accepted socket.
        //  Post one or more WSASend() or WSARecv() calls
        //  on the socket using overlapped I/O.
        WSARecv(...);
    }
```

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，便可以套接字句柄为基础，投递发送与接收请求，开始对I/O请求的处理。接下来，可开始依赖完成端口，来接收有关 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 Win32重叠I/O机制。在这种机制中，象 WSASend和WSARecv这样的Winsock API调用会立即返回。此时，需要由我们的应用程序负责在以后的某个时间，通过一个OVERLAPPED结构，来接收调用的结果。在完成端口模型中，要想做到这一点，需要使用 GetQueuedCompletionStatus（获取排队完成状态）函数，让一个或多个工作者线程在完成端口上等待。该函数的定义如下：

```
BOOL GetQueuedCompletionStatus(
    HANDLE CompletionPort,
    LPDWORD lpNumberOfBytesTransferred,
    LPDWORD lpCompletionKey,
    LPOVERLAPPED * lpOverlapped,
    DWORD dwMilliseconds
);
```

其中， CompletionPort参数对应于要在上面等待的完成端口。 lpNumberOfBytes Transferred参数负责在完成了一次I/O操作后（如WSASend或WSARecv），接收实际传输的字节数。lpCompletionKey参数为原先传递进入CreateCompletionPort函数的套接字返回“单句柄数据”。如我们早先所述，大家最好将套接字句柄保存在这个“键”（Key）中。lpOverlapped参数用于接收完成的I/O操作的重叠结果。这实际是一个相当重要的参数，因为可用它获取每个I/O操作的数据。而最后一个参数， dwMilliseconds，用于指定调用者希望等待一个完成数据包在完成端口上出现的时间。假如将其设为 INFINITE，调用会无休止地等待下去。

3. 单句柄数据和单I/O操作数据

一个工作者线程从 GetQueuedCompletionStatus这个API调用接收到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的 I/O处理。通过这些参数，可获得两方面重要的套接字数据：单句柄数据，以及单 I/O操作数据。

其中，lpCompletionKey参数包含了“单句柄数据”，因为在一个套接字首次与完成端口关联到一起的时候，那些数据便与一个特定的套接字句柄对应起来了。这些数据正是我们在进行CreateIoCompletionPort API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”。我们的工作者线程处理一个完成数据包时（将数据原封不动打转回去，接受连接，投递另一个线程，等等），这些信息是它必须要知道的。单 I/O操作数据可以是追加到一个OVERLAPPED结构末尾的、任意数量的字节。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一个元素使用。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;
    WSABUF     DataBuf;
    CHAR       Buffer[DATA_BUFSIZE];
    BOOL       OperationType;
} PER_IO_OPERATION_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，比如刚才完成的那个 I/O操作的类型（发送或接收请求）。在这个结构中，我们认为用于已完成 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，既可将自己的结构“造型”为一个 OVERLAPPED指针，亦可简单地撤消对结构中的OVERLAPPED元素的引用。如下例所示：

```
PER_IO_OPERATION_DATA  PerIoData;

//  可像下面这样调用一个函数
    WSARecv(socket, ..., (OVERLAPPED *)&PerIoData);
//  或像这样
    WSARecv(socket, ..., &(PerIoData.Overlapped));
```

在工作线程的后面部分，等 GetQueuedCompletionStatus函数返回了一个重叠结构（和完成键）后，便可通过撤消对 OperationType成员的引用，调查到底是哪个操作投递到了这个句柄之上（只需将返回的重叠结构造型为自己的 PER_IO_OPERATION_DATA结构）。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，假定我们的机器安装了多个中央处理器，每个处理器都在运行一个工作者线程，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。

为了完成前述的简单回应服务器示例，我们需要提供一个 ServerWorkerThread（服务器工作者线程）函数。在程序清单 8-10中，我们展示了如何设计一个工作者线程例程，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。

程序清单8-10 完成端口工作者线程

```
DWORD WINAPI ServerWorkerThread(
    LPVOID CompletionPortID)
{
    HANDLE CompletionPort = (HANDLE) CompletionPortID;
    DWORD BytesTransferred;
    LPOVERLAPPED Overlapped;
    LPPER_HANDLE_DATA PerHandleData;
    LPPER_IO_OPERATION_DATA PerIoData;
    DWORD SendBytes, RecvBytes;
    DWORD Flags;

    while(TRUE)
    {
        // Wait for I/O to complete on any socket
        // associated with the completion port

        GetQueuedCompletionStatus(CompletionPort,
            &BytesTransferred,(LPDWORD)&PerHandleData,
            (LPOVERLAPPED *) &PerIoData, INFINITE);
```

```
            // First check to see whether an error has occurred
            // on the socket; if so, close the
            // socket and clean up the per-handle data
            // and per-I/O operation data associated with
            // the socket

            if (BytesTransferred == 0 &&
                (PerIoData->OperationType == RECV_POSTED ||
                 PerIoData->OperationType == SEND_POSTED))
            {
                // A zero BytesTransferred indicates that the
                // socket has been closed by the peer, so
                // you should close the socket. Note:
                // Per-handle data was used to reference the
                // socket associated with the I/O operation.

                closesocket(PerHandleData->Socket);

                GlobalFree(PerHandleData);
                GlobalFree(PerIoData);
                continue;
            }

            // Service the completed I/O request. You can
            // determine which I/O request has just
            // completed by looking at the OperationType
            // field contained in the per-I/O operation data.
            if (PerIoData->OperationType == RECV_POSTED)
            {
                // Do something with the received data
                // in PerIoData->Buffer
            }

            // Post another WSASend or WSARecv operation.
            // As an example, we will post another WSARecv()
            // I/O operation.

            Flags = 0;

            // Set up the per-I/O operation data for the next
            // overlapped call
            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);
        }
    }
```

在程序清单8-9和程序清单8-10列出的简单服务器示例中（配套光盘也有），最后要注意的一处细节是如何正确地关闭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。要避免的一个重要问题是在进行重叠I/O操作的同时，强行释放一个OVERLAPPED结构。要想避免出现这种情况，最好的办法是针对每个套接字句柄，调用closesocket函数，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，便需在完成端口上，终止所有工作者线程的运行。要想做到这一点，需要使用PostQueuedCompletionStatus函数，向每个工作者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都“立即结束并退出”。下面是PostQueuedCompletionStatus函数的定义：

```
BOOL PostQueuedCompletionStatus(
    HANDLE CompletionPort,
    DWORD dwNumberOfBytesTransferred,
    DWORD dwCompletionKey,
    LPOVERLAPPED lpOverlapped
);
```

其中，CompletionPort参数指定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这三个参数来说，每一个都允许我们指定一个值，直接传递给GetQueuedCompletionStatus函数中对应的参数。这样一来，一个工作者线程收到传递过来的三个GetQueuedCompletionStatus函数参数后，便可根据由这三个参数的某一个设置的特殊值，决定何时应该退出。例如，可用dwCompletionPort参数传递0值，而一个工作者线程会将其解释成中止指令。一旦所有工作者线程都已关闭，便可使用CloseHandle函数，关闭完成端口，最终安全退出程序。

4. 其他问题

另外还有几种颇有价值的技术，可用来进一步改善套接字应用程序的总体I/O性能。值得考虑的一项技术是试验不同的套接字缓冲区大小，以改善I/O性能和应用程序的扩展能力。例如，假如某个程序只采用了一个比较大的缓冲区，仅能支持一个WSARecv请求，而不是同时设置三个较小的缓冲区，提供对三个WSARecv请求的支持，那么该程序的扩展能力并不是很好，特别是在转移到安装了多个处理器的机器上之后。这是由于单独一个缓冲区每次只能处理一个线程！除此以外，单缓冲区设计还会对性能造成一定的干扰：假如一次仅能进行一次接收操作，网络协议驱动程序的潜力便不能得到充分发挥（它经常都会很“闲”）。换言之，假如在接收更多的数据前，需要等等一次WSARecv操作的完成，那么在WSARecv完成和下一次接收之间，整个协议实际上处于“休息”状态。

另一个值得考虑的性能改进措施是用套接字选项SO_SNDBUF和SO_RCVBUF对内部套接字缓冲区的大小进行控制。利用这些选项，应用程序可更改一个套接字的内部数据缓冲区的大小。如将该设为0，Winsock便会在重叠I/O调用中直接使用应用程序的缓冲区，进行数据在协议堆栈里的传入、传出。这样一来，在应用程序与Winsock之间，便避免了进行一次缓冲区复制的必要。下述代码片断阐释了如何使用SO_SNDBUF选项，来进行setsockopt函数的调用：

```
int nZero = 0;

setsockopt(socket, SOL_SOCKET, SO_SNDBUF,
    (char *)&nZero, sizeof(nZero));
```

要注意的是，将这些缓冲区的大小设为0后，只有在一段给定的时间内，存在着多个I/O请求的前提下，才会产生积极作用。等到第9章，我们会向大家更深入地讲述套接字选项的知识。

提升性能的最后一项措施是使用AcceptEx这个API调用，来进行连接请求的处理，并投递

少量数据。这样一来，我们的应用程序只需通过一次 API调用，便可为一次接受请求和数据的接收提供服务，从而减少了单独进行 accept和WSARecv调用造成的开销。这样做还有另一个好处，我们可使用完成端口为 AcceptEx提供服务，因为它也提供了一个 OVERLAPPED结构。假如事先预计到自己的服务器应用在一个连接建好之后，只会进行少量的 recv-send（收发）操作，那么AcceptEx便显得相当有用（比如在设计一个 Web服务器的时候）。否则的话，接受一个连接后，假如程序要负责数百上千次数据传输操作，这样做对性能便没多大的助益。

最后提醒大家注意，在Winsock中，一个Winsock应用不应使用ReadFile和WriteFile这两个Win32函数，在一个完成端口上进行I/O处理。尽管这两个函数确实提供了一个 OVERLAPPED结构，而且可在完成端口上成功地使用，但就在 Winsock 2环境下进行 I/O处理来说，WSARecv和WSASend这两个函数却进行了更大程度的优化。若使用 ReadFile和WriteFile，需进行大量不必要的内核／用户模式进程调用、线程执行场景的频繁切换以及参数的汇集等等，使总体性能大打折扣。

8.3 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，大家心中可能还没什么数。前面已经提到，每种模型都有自己的优点和缺点。同开发一个简单的锁定模式应用相比（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的开发，我们分别提供了下述建议。

1. 客户机的开发

若打算开发一个客户机应用，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 WSAAsyncSelect模型恐怕是一种最好的选择，因为WSAAsyncSelect本身便是从Windows消息模型借鉴来的。若采用这种模型，我们的程序一开始便具备了处理消息的能力。

2. 服务器的开发

若开发的是一个服务器应用，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点考虑的。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。

8.4 小结

至此，我们已全面介绍了可用于 Winsock的各种I/O模型。这些模型从简单的锁定 I/O，到高性能的完成端口 I/O（获得最大的吞吐速度），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。本章也提供了对 Winsock一些常规用途的解释。到目前为止，大家已学习可选用哪些传送协议、可设置哪些套接字创建属性、如何创建基本客户机／服务器应用以及与 Winsock有关的其他常规主题。从第 9章开始，到第 14章，我们将讲述一些特殊的Winsock主题。首先在下一章（第9章），我们要解释可对套接字及基层协议的行为产生影响的一些套接字选项与 I/O控制命令。

第9章　套接字选项和I/O控制命令

套接字一旦建立，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。 I/O控制命令肯定会对套接字的行为产生影响。本章着重讨论四个 Winsock函数：getsockopt、setsocketopt、ioctlsocket和WSAIoctl。每个函数都有大量命令，其中大多数尚未正式或正确写入参考文档。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 Win32平台上运行（Windows CE、Windows 95、Windows 98、Windows NT和Windows 2000）；如有例外，我们会专门注明。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。因此， Winsock 2的I/O控制命令和选项在Windows CE与Windows 95平台上未获支持（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以外的其他协议的任何专用选项。

这些I/O控制命令和选项大多定义在Winsock.h或Winsock2.h内，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提供者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。而传输提供者扩展定义在与其协议对应的头文件内。针对那些传输特有选项，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，那么必须同Mswock.lib建立链接。

9.1　套接字选项

对getsocketopt（获得套接字选项）函数来说，它的常见用法是获得与指定套接字相关的信息。其原型如下：

```
int getsockopt (
    SOCKET s,
    int level,
    int optname,
    char FAR* optval,
    int FAR* optlen
);
```

第一个参数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。对你打算使用的具体协议来说，这个套接字必须是有效的。大多数选项都是一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（特别是第二个参数 level）。SOL_SOCKET的一个选项级别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级别SOL_SOCKET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，但并非所有支持的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。最常见的与协议无关选项（比如和 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，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或由协议决定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
);
```

它的参数和getsocketopt函数的参数相同，例外的是我们以 optval和optlen参数的形式，将值传递进去。这些值是为指定的选项设定的。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，但那个套接字的基层协议却不具备某种指定的特征（或选项）。例如，类型为SOCK_STREAM的一个套接字本身是不能对数据进行广播操作的；因此，若试图设置或获取 SO_BROADCAST选项，便会造成WSAENOPROTOOPT错误。

9.1.1 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，但这些信息并非那个套接字的基层协议所特有的（与它无关）。

1. SO_ACCEPTCONN

选项值类型	获取/设置	Winsock版本	说明
布尔值	只能获取	1+	如为TRUE（真），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。

2. SO_BROADCAST

选项值类型	获取/设置	Winsock版本	说明
布尔值	两种均可	1+	如为TRUE，表明套接字已配置成对广播消息进行发送

如果指定的套接字已经配置成收发广播数据，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机器都能收到相同的数据。当然，在监听进入广播数据的机器上，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，甚至有可能陷入瘫痪。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，比如recvfrom或WSARecvfrom。另外还有一种方法，即把套接字与广播地址连接起来，这是通过调用connect或WSAConnect，再调用recv或WSARecv来实现的。对UDP广播来说，必须指定一个端口号，以便向它发送数据报；类似的，接收端也必须请求在这个端口上接收广播数

据。下面的示范代码解释了如何通过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。通过为INADDR_BROADCAST提供一个#define引导命令，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，由广播数据专用。通过第6章的学习，大家已经知道AppleTalk地址共由三部分组成：网络、节点和套接字（目的地）。要进行广播通信，将目的地设为ATADDR_BROADCAST(0xFF)，这样便会促使数据报发给一个指定网络内的所有端点。

通常，只有在发送广播数据报时，才需要设置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，对进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”编号为Q137914的文章里得到了特别声明，地址是http://support.microsoft.com/support/search。这是Windows 95的一个错误。

3. SO_CONNECT_TIME

选项值类型	获取/设置	Winsock版本	说明
整数	只能获取	1+	返回套接字建立连接的时间，以秒为单位

SO_CONNECT_TIME是一个微软专用选项，用于返回连接建立的秒数。该选项最常见的用法是和AcceptEx函数一道使用。AcceptEx要求为进入的客户机连接请求，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。若套接字当前尚未连接，返回值便是0xFFFFFFFF。

4. SO_DEBUG

选项值类型	获取/设置	Winsock版本	说明
布尔值	两者均可	1+	如果是TRUE，就允许调试输出

推荐Winsock服务提供者在应用程序设置了SO_DEBUG选项的前提下，输出调试信息。至于调试信息如何展示，要取决于服务提供者的基层实施方式。要想打开调试信息，可设置SO_DEBUG，将布尔变量设为TRUE，调用setsockopt函数。若随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Win32平台支持SO_DEBUG选项，这在“微软知识库”编号为Q138965的文章里已有说明。尽管在设置了该选项的前提下，不会返回任何错误，但基层网络提供者会将其忽略。

5. SO_DONTLINGER

选项值类型	获取/设置	Winsock版本	说明
布尔值	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“安全”关闭。对这些协议来说，它们实现了一种特殊的机制，可防止在通信一方或双方关闭了套接字的前提下，造成尚未处理或尚未传输完毕的数据的丢失。要想改变这种行为，可设置SO_LINGER选项，然后调用setsockopt函数，从而改变这种行为。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待决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）。若设置了SO_DONTLINGER（不拖延）选项，可确保不经历这样的一段拖延时间。若随SO_DONTLINGER一起调用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一个拖延时间值。若在设置了SO_DONTLINGER的前提下，调用setsockopt，便会将这种拖延禁止。要注意的是，类型为SOCK_DGRAM的套接字不支持这一选项。

6. SO_DONTROUTE

选项值类型	获取/设置	Winsock版本	说明
布尔值	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（不路由）选项用于指示位于基层的网络堆栈，忽略路由表的存在，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UDP套接字，并将其与接口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机器。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，便可禁止这样做，数据包会从绑定的接口上直接发送出去。以后可调用getsockopt，判断是否允许了路由。要注意的是，在默认情况下，路由是允许的。

可在一个Win32平台上成功调用该选项。但是，Microsoft提供者会悄悄地忽略这一请求，并总是通过路由表为外出数据决定一个适当的接口。

7. SO_ERROR

选项值类型	获取/设置	Winsock版本	说明
布尔值	只能获得	1+	返回错误状态

SO_ERROR选项用于返回和重设以具体套接字为基础的错误代码。这些错误与那些以具体线程为基础的错误代码颇有不同，后者是用WSAGetLastError和WSASetLastError函数调用来控制的。若使用套接字，进行了一次成功的调用，那么不会重设由SO_ERROR选项返回的、以具体套接字为基础的错误代码。尽管调用该选项不会造成错误；但是，错误值并非肯定能够及时更新。因此，该选项有些时候会返回0值（表明没有错误）。除非绝对需要依赖单独的错误代码，否则最好还是使用WSAGetLastError。

8. SO_EXCLUSIVEADDRUSE

选项值类型	获取/设置	Winsock版本	说明
布尔值	两者均可	2+	如果是TRUE，套接字绑定的那个本地端口就不能重新被另一个进程使用

该选项是对SO_REUSEADDR（重复使用地址）的一个补充，后者很快就会讲到。该选项的作用是禁止其他进程在一个本地地址上使用SO_REUSEADDR（我们的应用程序正在这个地址上运行）。例如，假定两个单独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进入连接请求通知呢？

这一点并未定义！如果应用程序的要求非常苛刻（用于关键性任务的程序），这一点显然是非常不幸的。SO_EXCLUSIVEADDRUSE选项的作用便是将一个本地地址牢牢锁定在与它绑定的那个套接字上。这样一来，假如有其他进程试图针对相同的本地地址使用SO_REUSEADDR，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，它适用于Windows 2000！

9. SO_KEEPALIVE

选项值类型	获取/设置	Winsock版本	说明
布尔值	两者均可	1+	如果是TRUE，套接字就会进行配置，在会话过程中发送“保持活动”消息

对一个以TCP为基础的套接字来说，应用程序可请求基层服务提供者允许在TCP连接上使用“保持活动”（Keep Alive）数据包，这是通过打开SO_KEEPALIVE套接字选项来做到的。在Win32平台上，“保持活动”是根据RFC 1122文件的4.2.3.6小节的规定而实现的。假如由于“保持活动”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返回WSAENOTCONN错误。要想了解更多的实施细节，请大家参阅RFC文件。在此要注意的一个要点，“保持活动”信号的发送要以不短于2小时的时间间隔进行。2小时的“保持活动”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“保持活动”行为。而这种后果通常是应当避免的。另一个办法是实现自己的“保持活动”策略。在第7章，我们已讨论了这种类型的策略。要注意的是，SOCK_DGRAM类型的套接字并不支持这一选项。

与“保持活动”设置对应的注册表键是KeepAliveInterval（保持活动间隔）和KeepAliveTime（保持活动时间）。这两个键均用来保存类型为REG_DWORD的双字值，单位是“毫秒”。其中，前一个键指定每次“保持活动”传输的间隔时间，直至收到一个响应。后一个键用于控制TCP以多大的频率尝试一个“保持活动”数据包的发送，以查验一个理想的连接是否仍然有效。在Windows 95及Windows 98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和新出的Windows 2000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，在Windows 2000操作系统中，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“保持活动”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更改。本章稍后还会对这个I/O控制命令进行详解。

10. SO_LINGER

选项值类型	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是“拖延”的意思。SO_LINGER用于控制在未发送的数据排队等候于套接字上的时候，一旦执行了closesocket命令，那么该采取什么样的行动。若在设置了该选项的前提下，调用getsockopt，便会在一个linger结构中，返回当前的拖延时间设定。该结构定义如下：

```
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关闭拖延，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丢弃，同时重设与对方的连接。相反，我们可调用 setsockopt，在丢弃任何正在排队等候的数据之前，启用拖延功能，同时指定一个时间长度。要想达到这个目的，需要在类型为 struct linger的一个变量中指定希望的时间长度。

若随setsockopt一道设置了拖延时间值，那么必须将结构的 l_onoff字段设为一个不为零的值（非零值）。若允许拖延，后来又想将其禁止，可随 SO_LINGER选项一道，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（不拖延）选项调用setsockopt，为其optval参数传递一个TRUE值。但要注意的是，类型为SOCK_DGRAM的套接字不运行这一选项。

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 9-1中，我们对不同的行为进行了总结。

表9-1 拖延选项

选 项	间隔时间	关闭方式	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设为零（换言之，linger结构的成员l_onoff不为0，但l_linger为0），便不用担心closesocket调用会进入“锁定”状态（等待完成），即使队列中的数据尚未发送，或者尚未发出收到确认。我们称这种情况为“强行关闭”，或者“野蛮关闭”，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。

如果在一个锁定套接字上，设置了 SO_LINGER，那么closesocket调用也会在一个锁定套接字上进入锁定或暂停状态，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“从容关闭”。若超过规定的时间，数据仍未全部发出， Windows套接字机制便会在closesocket返回之前，将连接中断。

11. SO_MAX_MSG_SIZE

选项值类型	获取/设置	Winsock版本	说明
无符号整数	只能获取	2+	对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者实现的、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。在那些套接字上，没有办法知道最大的进入消息长度。

12. SO_OOBINLINE

选项值类型	获取/设置	Winsock版本	说明
布尔值	两者均可	1+	如果是TRUE，带外数据就会在普通数据流中返回

默认情况下，“带外”（Out-of-band，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数据只需单独一次调用便会返回。若设置了这个选项， OOB数据会出现于自一个接收调用返回的数据流中。而在设置了

SIOCATMARK选项的前提下，若调用 ioctlsocket，便可决定哪个字节属于 OOB数据。类型为SOCK_DGRAM的套接字同样不运行这一选项。不幸的是，在目前各个版本的 Win32平台上，均不支持这个套接字选项。要想了解 OOB数据的详情，可参阅第 7章。

13. SO_PROTOCOL_INFO

选项值类型	获取/设置	Winsock版本	说明
WSAPROTOCOL_INFO	只能获得	2+	套接字绑定的那种协议的特征

这又是一个“只能获得”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 6章，了解WSAPROTOCOL_INFO结构的详细说明，及其各个成员字段的详情。

14. SO_RCVBUF

选项值类型	获取/设置	Winsock版本	说明
整数	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长度

这是一个非常简单的选项，用于返回或设置分配给该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，会为其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。若请求将接收缓冲区的大小设为一个特定的值，那么即便没有充分满足这个请求，没有提供全部要求的空间，对 setsockopt的调用也会成功，不会返回错误。要想确保请求的缓冲区空间都已分配，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Win32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，只能“取得”它。

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，想进行 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由用户指定的缓冲区。假如没有中间缓冲区，数据就会立即复制到用户指定的缓冲区内。而这样的的操作只有在同时明确进行多个接收调用的前提下，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，因为除非指定一个缓冲区，而且准备好接收数据，否则本地系统是不会接受任何进入数据的。欲知详情，请参考第8章8.2.5节“完成端口模型”中“其他问题”小节。

15. SO_REUSEADDR

选项值类型	获取/设置	Winsock版本	说明
布尔值	两者均可	1+	如果是TRUE，套接字就可与一个正由其他套接字使用的地址绑定到一起，或与处在TIME_WAIT状态的地址绑定到一起

默认情况下，套接字不同一个正在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，来实现对一个地址的重复利用。通过第 7章的学习，大家已经知道，每个连接都是通过它的本地及远程地址的组合，“独一无二”地标识出来的。针对我们想要连接的地址，只要能用极其细微的差异（比如 TCP/IP中采用不同的端口号），来维持这种“独一无二”或者“唯一”的特点，绑定便是允许的。

唯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，以等待进入的连接通知。假定两个套接字都在同一个端口上进行监听，那么到底由谁来接收一个进入连接通知呢？对于这个问题，目前尚无一种正式规范提出了解决方案。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。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。但假若设置了该选项，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。

16. SO_SNDBUF

选项值类型	获取/设置	Winsock版本	说明
布尔值	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数据发送缓冲区的大小。创建好一个套接字后，会为其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，那么即便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，不会返回错误信息。但是，假如希望确定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Win32平台都能获取或设置发送缓冲区的大小。在Windows CE中，我们不可更改这个值，但能“取得”它。

和SO_RCVBUF一样，可用SO_SNDBUF选项将发送缓冲区的大小设为0。对于锁定发送调用来说，将缓冲区的大小设为0后，一个好处在于：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，不会涉及数据向系统缓冲区进行的额外内存复制操作。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，在以后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，假如应用程序需要的其他方面的优势，那么将发送缓冲区屏蔽后，亦可省下进行内存复制时的一些机器指令开销。欲知这方面的详情，可参阅第 8章8.2.5节“完成端口模型”中“其他问题”小节。

17. SO_TYPE

选项值类型	获取/设置	Winsock版本	说明
整数	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。

18. SO_SNDTIMEO

选项值类型	获取/设置	Winsock版本	说明
整数	两者均可	1+	获取或设置套接字上的数据发送超时时间（以毫秒为单位）

SO_SNDTIMEO选项用于在一个锁定套接字上调用Winsock发送函数时，设定一个“超时”值。这是一个以毫秒为单位的整数值，设定发送函数最多执行多久。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）都只会锁定指定数量的时间。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）。

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。

19. SO_RCVTIMEO

选项值类型	获取/设置	Winsock版本	说明
整数	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选项用于设置一个锁定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“锁定”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指定成为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）都只会锁定指定的时间长度。假如在那段时间内，没有数据抵达，调用便会失败，并返回错误 10060（WSAETIMEDOUT）。

考虑到性能方面的因素，这一选项在 Windows CE 2.1中已被禁止。如试图设置它，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。

20. SO_UPDATE_ACCEPT_CONTEXT

选项值类型	获取/设置	Winsock版本	说明
SOCKET	两者均可	1+	获取或设置与套接字上的数据接收对应的超时值（以毫秒计）

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，来进行数据接收调用的执行。该函数要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，它会成为一个已接受的客户机。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“服务质量”（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，这个选项是必须设置的。为在一个套接字上设置该选项，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（比如客户机句柄）。要注意的是，该选项仅适用于 Windows NT和Windows 2000这两种操作系统。

9.1.2 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名字的设置或获取有关。欲了解 AppleTalk地址家族更详细的信息，可参考第 6章的内容。某些AppleTalk套接字选项（比如 SO_DEREGISTER_NAME）提供了不止一个的选项名。在这种情况下，所有选项的名字均可互换使用。

1. SO_CONFIRM_NAME

选项值类型	获取/设置	Winsock版本	说明
WSH_NBP_TUPLE	只能获取	1	确定指定的AppleTalk名字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名字是否已和指定的地址绑定到了一起。这样可促使向目标地址发送一个“名字绑定协议”（NBP）检索请求，以校验指定的名字。若此次调用以失败告终，并返回了一个 WSAEADDRNOTAVAIL错误，便表明名字尚未与指定的地址绑定到一起。

2. SO_DEREGISTER_NAME，SO_REMOVE_NAME

选项值类型	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字的注册

该选项用于将一个名字从网络中撤消对它的注册。若指定的名字目前并未存在于网络中，调用返回后，仍会注明“操作成功”。请参阅第6章的6.5.2节“AppleTalk名的注册”小节，了解WSH_REGISTER_NAME结构的详情，它实际是WSH_NBP_NAME结构的另一个名字。

3. SO_LOOKUP_MYZONE，SO_GETMYZONE

选项值类型	获取/设置	Winsock版本	说明
字符	只能获取	1	返回网络上的默认网区

该选项可返回网络上的默认“网区”。getsockopt调用的optval参数应当是一个字串，长度至少33个字符。要注意的是，NBP名字的最大长度是由MAX_ENTITY_LEN规定的，后者定义成32。若实际的名字长度不够，便需加上额外的字符，形成“空终止符”。

4. SO_LOOKUP_NAME

选项值类型	获取/设置	Winsock版本	说明
WSH_LOOKUP_NAME	只能获取	1	查找指定的NBP名字，并返回相符的名字及NBP信息字元组

该选项用于在网络上查找一个指定的名字（比如，在一个客户机想建立与服务器的连接时）。连接建立之前，字面上易于理解的英语名字必须解析成一个具体的AppleTalk地址。大家可参考第6章6.5.3节“AppleTalk名的解析”，那里提供了一段示范代码，可直接用来进行AppleTalk名字的解析。

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WSH_NBP_TUPLE结构。图9-1向大家展示了在调用之初，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9-1 SO_LOKUP_NAME缓冲区

5. SO_LOOKUP_ZONES，SO_GETZONELIST

选项值类型	获取/设置	Winsock版本	说明
WSH_LOOKUP_ZONES	只能获取	1	返回来自Internet网区列表的区名

该选项要求提供一个足够大的缓冲区，以便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“空终止”的区名列表。下述代码向大家解释了如何利用这个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++);
}
```

6. SO_LOOKUP_ZONES_ON_ADAPTER，SO_GETLOCALZONES

选项值类型	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，只是要求我们先指定一个适配器名，再来取得与那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。而返回的空终止区名列表自 WSH_LOOKUP_ZONES结构之后开始存放。此外，适配器的名字必须以一个 UNICODE字串（WCHAR）的形式传递。

7. SO_LOOKUP_NETDEF_ON_ADAPTER，SO_GETNETINFO

选项值类型	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，以及默认网区

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”终止的ANSI字串，其中包含了与指定适配器连接的那个网络的默认网区。适配器是在结构之后，以一个 UNICODE（WCHAR）字串的形式传递的，并会由函数返回的默认网区覆盖。假如网络并未进行“种子”设定，那么会返回一个网络地址范围1~0xFFFE，同时在空终止的ANSI字串中包含默认网区“*”。

8. SO_PAP_GET_SERVER_STATUS

选项值类型	获取/设置	Winsock版本	说明
WSH_PAP_GET_SERVER_STATUS	只能获取	1	自指定服务器返回PAP状态

该选项可取得在ServerAddr指定之地址上注册的“打印机访问协议”（PAP）状态。而那个地址通常是通过一次NBP检索操作获得的。四个保留的字节分别对应于 PAP状态包中的四个保留字节，均按网络字节顺序排布。而一个 PAP状态字串可以是任意的，通过 SO_PAP_SET_SERVER_STATUS选项进行设置；详情还会在本章后面进行解释。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;
```

下述代码片断向大家展示了如何对 PAP状态进行请求。状态字串的长度是由 ServerStatus字段的第一个字节指定的：

```
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);
```

9. SO_PAP_PRIME_READ

选项值类型	获取/设置	Winsock版本	说明
char []	只能设置	1	这个调用会在一个PAP连接上填充一次读取操作，以便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“锁定”或“暂停”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（4096）个字节。通过这一选项，可在“由读驱动”的PAP协议上，实现对非锁定套接字的支持。但要注意的是，针对自己想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。

10. SO_PAP_SET_SERVER_STATUS

选项值类型	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，请求取得PAP状态。可用该选项对状态进行设置，以便在客户机请求状态的前提下，提交给设置命令的缓冲区能在获取命令中返回。这里的“状态”实际是一个最多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，则会将以前的状态值清除掉。

11. SO_REGISTER_NAME

选项值类型	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字

该选项用于在AppleTalk网络中注册一个指定的名字。若那个名字已在网络中存在，便会返回WSAEADDRINUSE错误。大家可参考第6章，那里提供了对WSH_REGISTER_NAME结构的详细说明。

9.1.3 SOL_IRLMP选项级别

SOL_IRLMP级别与IrDA（Infrared Data Association，红外线数据联盟）协议有着密切的联系，其地址家族为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施方式是有所区别的。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和Windows 2000中引入的一些新选项。在本节，我们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。

1. IRLMP_9WIRE_MODE

选项值类型	获取/设置	Winsock版本	说明
布尔值	两者均可	1+	获取或设置IP头内的IP选项

这是另一个非常罕见的选项，用于通过IrCOMM（红外线通信端口），建立与Windows 98的通信。它的工作级别要比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，取得一个IrLMP包的最大PDU长度。随后，在建立连接，或在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（即发送出去）的每个数据帧都添加一个长度为1字节的IrCOMM头（总是设为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（包括第二版）和Windows 2000。

2. IRLMP_ENUMDEVICES

选项值类型	获取/设置	Winsock版本	说明
DEVICELIST	只能获得	1+	针对范围内具有红外线通信能力的设备，返回一个 IrDA设备ID列表

由红外线通信的本质使然，具有这种能力的所有设备都注定是“机动”的，或者说是“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，而后来问世的Windows 98和Windows 2000更上一层楼，新增了一系列选项，可以更加完善地支持这种类型的通信。下面是在 Windows 98及Windows 2000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结构是象这样定义的：

```
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;
```

从中可以看出，设备信息结构也发生了变化： Windows CE使用的是 WCE_IRDA_DEVICE_INFO；而Windows 98和Windows 2000使用的是WINDOWS_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，用于唯一性地标识出那个设备。我们需要用这个字段来填充用于连接特定设备的 SOCKADDR_IRDA结构；或者通过IRLMP_IAS_SET及IRLMP_IAS_QUERY用于，分别用来处理或获取一个“信息访问服务”（IAS）条目。

调用getsockopt函数，对红外线设备进行列举的时候， optval参数必须对应一个DEVICELIST结构。此时唯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，那么对getsockopt的调用不会返回任何错误。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的值相等数量的一系列结构。

3. IRLMP_EXECLUSIVE_MODE

选项值类型	获取/设置	Winsock版本	说明
布尔值	两者均可	1+	如果是TRUE，表明套接字连接处于“独占”模式

该选项通常并不在应用程序中直接使用，因为它绕过了 IrDA堆栈中的TinyTP层，而直接通过IrLMP进行通信。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。

4. IRLMP_IAS_QUERY

选项值类型	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，首先必须设置好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，应设为想取得具体值的一个属性字串。最常见的查询是针对LSAP-SEL号码进行的；它的属性字串是“IrDA:IrLMP:LsapSel”。接下来，需要将irdaClassName字段设为指定属性字串即将应用于的那个服务的名字。一旦填写好这些字段的内容，便可发出对 getsockopt的调用。若成功返回，在irdaAttribType字段中，会指明需要同哪个字段联合，以便从中取得信息。可根据表 9-2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，便会返回这样的错误。该选项适用于Windows CE，Windows 98以及Windows 2000。

5. IRLMP_IAS_SET

选项值类型	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，为单个类设置一个属性。和 IRLMP_ENUMDEVICES一样，Windows CE采用的是一种结构，而 Windows 98和Windows 2000采用的又是另一种结构。下面是Windows 98和Windows 2000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9-2中，我们总结了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。最后两个条目不是可由我们自行设置的值，而是通过对 getsockopt的调用，同时设置一个IRLMP_IAS_QUERY套接字选项，而能在irdaAttribType字段中返回的值。之所以也在表格中列出，只是为了保证它们的完整性。

表9-2 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。在那个设备上，我

们想对其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，要在上面对属性进行设置。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。客户机使用与之对应的一个LSAP-SEL编号，建立与服务器的连接。 LSAP-SEL编号是与那种服务关联在一起的一种属性。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字段应设为属性字串“ IrDA:IrLMP:LsapSel ”，而irdaClassName字段应设为服务的名字（比如“ MySocketServer ”）。随后，将irdaAttribType设为IAS_ATTRIB_INT，而irdaAttribInt设为新的LSAP-SEL编号。当然，更改服务的LSAP-SEL编号并不是我们推荐的一种做法，本例只是为了阐述的方便。

6. IRLMP_IRLPT_MODE

选项值类型	获取/设置	Winsock版本	说明
布尔值	两者均可	1+	若为TRUE，套接字就会配置成与具有IR能力的打印机通信

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关系。大家可参考第 6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及Windows 2000。

7. IRLMP_SEND_PDU_LEN

选项值类型	获取/设置	Winsock版本	说明
整数	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，可通过该选项调查最大的“ 协议数据单元 ”（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和Windows 2000。

9.1.4 IPPROTO_IP选项级

IPPROTO_IP这一级的套接字选项与IP协议存在密切联系，比如可用它们 IP头内的特定字段，以及向IP多播组增添一个套接字等等。许多这样的选项都声明于 Winsock.h和Winsock2.h这两个头文件内，而且采用了不同的值。要注意的是，假如装载的是 Winsock 1，那么必须包括正确的头文件，同时建立与Wsock32.lib函数库的链接关系。若装载的是Winsock 2，那么情况也是类似的，除了将 Winsock 2的头文件包括进来外，还要建立与 Ws2_32.lib的链接关系。在多播通信环境中，这一点尤其重要，因为两个版本的 Winsock均支持多播通信。除Windows CE的早期版本之外，其他所有Win32平台都提供了对多播通信的支持。而 Windows CE自2.1版之后，也开始提供了这方面的支持。

1. IP_OPTIONS

选项值类型	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：

- 安全和控制限制：RFC 1108。
- 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13章的Ping程序示例）。

■ 时间标志（时间戳）：每个路由器都增添自己的IP地址及时间。
■ 松散源路由选择：为访问选项头内列出的每个 IP地址，便要求这个数据包。
■ 严格源路由选择：只有访问选项头内列出的那些 IP地址时，才要求用到这个数据包。

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IP选项时，传至setsockopt调用内部的数据会遵照如图 9-2所示的数据结构。 IP选项头最长可达40字节。

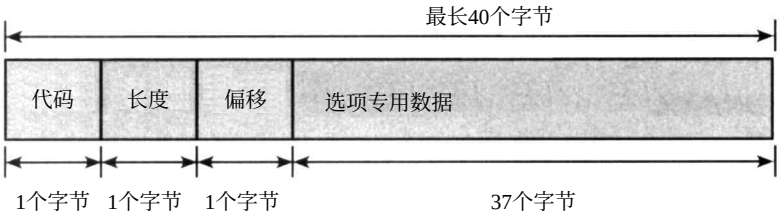

图9-2 IP选项头格式

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 ”部分必然是由特定的选项来决定的。在下述代码片断中，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），用它包含头三个选项值（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，由9个无符号（无正负号）的长整数（ long）构成，因为要记录的数据最多允许包含9个IP地址。在此要注意的是， IP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;  // Offset to first address (addrs)
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，要么必须将套接字创建成一个“ 原始套接字 ”（SOCK_RAW），要么应该设置IP_HDRINCL选项。在后一种情况下，IP头会在经过对一个Winsock获取函数的调用之后，随数据一道返回。

2. IP_HDRINCL

选项值类型	获取/设置	Winsock版本	说明
布尔值	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，并从读取的数据中返回

若将IP_HDRINCL选项设为TRUE，发送函数会将IP头包括在自己发送的数据前面，并会促使接收函数将IP头包括为自己的数据的一部分。因此，在我们调用一个 Winsock发送函数的时候，必须在数据前面包括完整的 IP头，并对IP头的每个字段进行正确的填写。在图 9-3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操作系统。

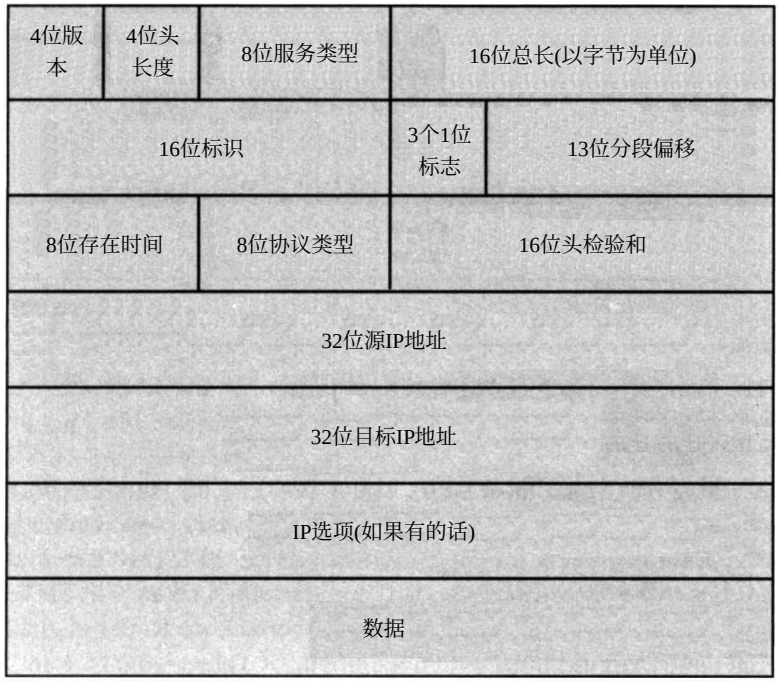

图9-3 IP头

其中，头的第一个字段指定的是 IP版本，目前通常是版本 4。头长度是指在整个头中， 32位字一共有多少个（一头的长度必须是 32位的整数倍）。接下来的字段是“ 服务类型 ”（TOS）。对此，大家可参考下文介绍的 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）。标识字段是一个独一无二的值，用于对发出的每个IP包进行“ 唯一性 ”的标定。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仅在IP包需要分割为较小的包时才会用到。而“ 存在时间 ”（Time to Live，TTL）字段，则用于限制数据包最多可穿越多少个路由器。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。这样一来，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。协议字段用于对进入的数据包组装。采用了IP定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的求余总和结果。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。 IP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 IP安全或路由选择有关。

要想将一个IP头和打算发送的数据组装到一起，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 13章的内容，那里除详细讲述了这方面的知识之外，还提供了与这个选项对应的一个例子。要注意的是，该选项仅适用于Windows 2000。

3. IP_TOS

选项值类型	获取/设置	Winsock版本	说明
整数	两者均可	1+	IP服务类型

服务类型（TOS）是在IP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，总共划分为三部分：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，四个TOS位分别对应于最短延迟、最高吞吐速度、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。若这四个位的值均为0，便暗示着采用“普通服务”。在RFC 1340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。此外， RFC 1349文件也对早先公布的RFC文件进行了一些补充及纠正。

对某些交互式应用而言，比如 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速度上。至于“最大可靠性”，则是网络管理（ SNMP）和路由协议所强烈要求的。最后， Usenet新闻协议（NNTP）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。

在提供服务质量（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，所以并不推荐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服务提供者会事先拦截这一调用，查验是否对此项设定进行了修改。大家可参阅第12章，了解QoS的详情。

4. IP_TTL

选项值类型	获取/设置	Winsock版本	说明
整数	两者均可	1+	IP协议的“存在时间”（TTL）参数

“存在时间”（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。之所以要进行这样的限制，目的在于避免由于路由循环，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，都会由路由器解析出它的 TTL值，并将其减去 1。若发现这个值变成了 0，数据报便会被“无情”地丢弃。但要注意的是， Windows CE不支持这个选项。

5. IP_MULTICAST_IF

选项值类型	获取/设置	Winsock版本	说明
无符号的长整数	两者均可	1+	获取或设置打算从它上面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，本地机器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（网卡、 Modem等等）的机器上，此项设置才有意义。optval参数应当是一个无符号（无正负号）的长整数值，对应于本地接口的二进制IP地址。可用inet_addr函数将一个字串形式的IP地址（点分十进制）转换成一个无符号的长整数值，如下例所示：

```
DWORD    mcastIF;
```

```
// First join socket s to a multicast group
mcastIF = inet_addr("129.113.43.120");
ret = setsockopt(s, IPPROTO_IP, IP_MULTICAST_IF, (char *)&mcastIF,
    sizeof(mcastIF));
```

6. IP_MULTICAST_TTL

选项值类型	获取/设置	Winsock版本	说明
整数	两者均可	1+	为套接字获取或设置多播数据包的存在时间

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可缩窄数据的“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. IP_MULTICAST_LOOP

选项值类型	获取/设置	Winsock版本	说明
布尔值	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”或“反弹”回套接字的进入缓冲区

默认情况下，在我们发送IP多播数据的时候，假如套接字亦属于那个多播组的一名成员，数据便会原封不动地回返至套接字。若将该选项设为 FALSE，发出的任何数据都不会投递至套接字的进入数据队列中。

8. IP_ADD_MEMBERSHIP

选项值类型	获取/设置	Winsock版本	说明
struc ip_mreq	只能设置	1+	在指定的IP组内为套接字赋予成员资格

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函数来创建地址家族为 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，请使用下述结构：

```
struct ip_mreq
{
    struct in_addr  imr_multiaddr;
    struct in_addr  imr_interface;
};
```

在ip_mreq结构中， imr_multiaddr对应于打算加入的那个多播组的二进制格式地址；而imr_interface对应于打算用来收发多播数据的那个本地接口。大家可参考第 11章，对哪些多播地址才算“有效”，做到心中有数。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，表明选择的是默认接口。

9. IP_DROP_MEMBERSHIP

选项值类型	获取/设置	Winsock版本	说明
struct ip_mre1	只能设置	1+	将套接字从指定的IP组内删去（撤消成员资格）

该选项正好与IP_ADD_MEMBERSHIP相反。随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相同的值，套接字 s便会从那个组内删去，脱离成员关系。同样地，第11章详细讲述了IP多播的知识。

10. IP_DONTFRAGMENT

选项值类型	获取/设置	Winsock版本	说明
布尔值	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，不要对 IP数据报进行分段处理。然而，假如一个 IP数据

报的大小已经超过了最大传输单元（MTU）值，但未在IP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。

9.1.5 IPPROTO_TCP选项级别

仅有一个选项从属于IPPROTO_TCP级别。该选项只适用于流式套接字（SOCK_STREAM），其地址家族为AF_INET。这个选项可用在所有Winsock版本上，并得到了所有Win32平台的支持，包括Windows CE。

TCP_NODELAY

选项值类型	获取/设置	Winsock版本	说明
布尔值	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，提升性能以及吞吐速度，系统默认采用Nagle算法。若应用程序请求发送一批数据（量较大），那么系统在接收了那些数据之后，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不管三七二十一地发送出去。这样造成的一个好结果便是：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2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收到确认消息的延迟发送。系统收到TCP数据之后，必须向对方反馈回一条ACK（收到确认）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）。

本选项的目的便是禁止采用Nagle算法，因为在某些情况下，它的行为反而会产生不利的影响。若网络应用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响应，那么再使用这种算法，反而会影响性能。Telnet便是这样的一个典型例子。Telnet的本质是一种“交互式”或“互动式”应用，用户可通过它登录至一台远程机器，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在此仍要不知好歹地“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。

9.1.6 NSPROTO_IPX选项级别

这儿列出的套接字选项均属Microsoft公司对Windows IPX/SPX套接字接口作出的特有扩展，用于确保与现有应用的兼容性。若非考虑到兼容性方面的原因，我们并不建议使用这些选项，因为它们仅适用于Microsoft IPX/SPX堆栈。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，应将该文件包括于Winsock.h及Wsipx.h这两个文件的后面。

1. IPX_PTYPE

选项值类型	获取/设置	Winsock版本	说明
整数	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。

2. IPX_FILTERPTYPE

选项值类型	获取/设置	Winsock版本	说明
整数	两者均可	1+	获取或设置准备过滤的IPX包之类型

该选项用于获取或设置接收过滤器数据包类型。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。

3. IPX_STOPFILTERPTYPE

选项值类型	获取/设置	Winsock版本	说明
整数	只能设置	1+	删除为指定IPX包设置的过滤器

可用该选项停止过滤当初用IPX_FILTERTYPE选项设置的数据包类型。

4. IPX_DSTYPE

选项值类型	获取/设置	Winsock版本	说明
整数	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。

5. IPX_EXTENDED_ADDRESS

选项值类型	获取/设置	Winsock版本	说明
布尔值	两者均可	1+	如果是TRUE，便允许对IPX包进行扩展定址

该选项用于允许或禁止扩展定址／寻址。发送数据时，它会在SOCKADDR_IPX结构中加入unsigned char sa_ptype，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：接收的帧是自这台机器发送出去的。

6. IPX_RECVHDR

选项值类型	获取/设置	Winsock版本	说明
布尔值	两者均可	1+	如果是TRUE，就随接收调用一起，返回IPX头

若将该选项设为“真”，那么任何Winsock接收调用都会随正式数据一道，返回IPX头。

7. IPX_MAXSIZE

选项值类型	获取/设置	Winsock版本	说明
整数	只能获取	1+	返回IPX数据报的最大长度

随该选项调用getsockopt函数，便可返回允许的最大IPX数据报长度。

8. IPX_ADDRESS

选项值类型	获取/设置	Winsock版本	说明
IPX_ADDRESS_DATA	只能获取	1+	返回具备IPX能力之适配器的有关信息

该选项用于对IPX绑定的一个特定适配器进行查询。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。如欲调查系统内到底安装了多少个提供IPX能力的适配器，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，直至返回错误。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;  // Input: 0-based adapter number
    UCHAR   netnum[4];   // Output: IPX network number
    UCHAR   nodenum[6];  // Output: IPX node address
    BOOLEAN wan;         // Output: TRUE = adapter is on a WAN link
    BOOLEAN status;      // Output: TRUE = WAN link is up (or adapter
                         // is not WAN)
    INT     maxpkt;      // Output: max packet size, not including IPX
                         // header
    ULONG   linkspeed;   // Output: link speed in 100 bytes/sec
                         // (i.e., 96 == 9600 bps)
} IPX_ADDRESS_DATA, *PIPX_ADDRESS_DATA;
```

9. IPX_GETNE TINFO

选项值类型	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

这个选项包含了与一个特定 IPX网络编号有关的信息。若网络刚好位于 IPX的缓冲之中，那么选项会直接将信息返回；否则的话，它会发出 RIP请求，以期找到目标网络。 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];  // Input: IPX network number
    USHORT hopcount;   // Output: hop count to this network, in machine
                       // order
    USHORT netdelay;   // Output: tick count to this network, in machine
                       // order
    INT    cardnum;    // Output: 0-based adapter number used to route
                       // to this net; can be used as adapternum input
                       // to IPX_ADDRESS
    UCHAR  router[6];  // Output: MAC address of the next hop router,
                       // zeroed if the network is directly attached
} IPX_NETNUM_DATA, *PIPX_NETNUM_DATA;
```

10. IPX_GETNETINFO_NORIP

选项值类型	获取/设置	Winsock版本	说明
IPX_ADDRESS_DATA	两者均可	1+	如果是TRUE，就不会对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的缓冲之内，信息便可直接返回；否则，调用就会失败，不会发出更多的请求。大家同时可参考IPX_RERIPNETNUMBER，后者无论如何都会发出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。

11. IPX_SPXGETCONNECTIONSTATUS

选项值类型	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。所有数字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; /* In milliseconds */
    USHORT RetransmittedPackets;
    USHORT SuppressedPacket;
} IPX_SPXCONNSTATUS_DATA, *PIPX_SPXCONNSTATUS_DATA;
```

12. IPX_ADDRESS_NOTIFY

选项值类型	获取/设置	Winsock版本	说明
IPX_ADDRESS_DATA	只能获取	1+	若IPX适配器的状态发生改变，则发出异步通知

该选项可提交一个请求，以便在IPX与之绑定的那个适配器的状态发生更新后，及时收到通知。注意状态的改变通常是在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，将其放在 optval参数中传递。然而，一个例外是在IPX_ADDRESS_DATA后面，要紧随着一个句柄，对应于一个尚未进入传信状态（即尚未触发）的事件。下述伪代码向大家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;  // Set to the appropriate adapter
*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，由 optval指向的那个IPX_ADDRESS_DATA结构在那时却不会马上发生更新。相反，请求会在传送层内部进入队列（排队）；以后若适配器的状态发生了变化， 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，它会传送由 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。之所以要利用事件，是由于调用需要“ 异步 ”进行； getsockopt目前尚不支持这一功能。

警告 以目前的实施方案来看，针对状态发生的每一种变化，传送层都只会传信队列中的一个查询。因此，在同一个时候，只能运行使用了排队查询的一个服务。

13. IPX_MAX_ADAPTER_NUM

选项值类型	获取/设置	Winsock版本	说明
整数	只能获取	1+	返回存在的IPX适配器个数

该选项可返回系统中具有 IPX功能的适配器的数量。若该调用返回 n个适配器，那么适配

器的范围编号便在0到n~1之间。

14. IPX_RERIPNETNUMBER

选项值类型	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，却不必重发）。和IPX_GETNETINFO类似，它也要求通过optval参数，传递一个IPX_NETNUM_DATA结构。

15. IPX_RECEIVE_BROADCAST

选项值类型	获取/设置	Winsock版本	说明
布尔值	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，也并不一定造成为那个应用过滤掉广播数据。

16. IPX_IMMEDIATESPXZCK

选项值类型	获取/设置	Winsock版本	说明
布尔值	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（ACK）不会在SPX连接之上推迟发送。若应用程序不需要在SPX通信链路上进行双向通信，便可尽情地将其设为 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推迟发送确认消息，造成对性能的影响。

9.2 IOCTLSOCKET和WSAIOCTL

一系列套接字I/O控制函数用于在套接字之上，控制I/O的行为，同时获取与那个套接字上进行的I/O操作有关的信息。其中，第一个函数是 ioctlsocket，起源于Winsock 1规范，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cmd是一个预定义的标志，用于打算执行的I/O控制命令。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，再给出要求变量的类型。 Winsock 2引入了一个新的ioctl函数，增添了数量多得多的新选项。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，用于容纳自调用返回的数据。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的相同。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，指向传递进入的值，而cbInBuffer指定的是数据的多少，以字节为单位。类似地，lpvOutBuffer和cbOutBuffer用于自调用返回的任何数据。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参数对应的是在lpvOutBuffer中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。第七个参数是lpcbBytesReturned，对应于实际返回的字节数。最后两个参数是lpOverlapped和lpCompletionRoutine，在随重叠I/O调用这个函数时使用。大家可参考第8章，了解重叠I/O的使用详情。

9.2.1 标准I/O控制命令

1. FIONBIO

函数	输入	输出	Winsock版本	说明
ioctlsocket/WSAIoctl	无符号	无	1+	将套接字置入非锁定模式

该命令可在套接字s上允许或禁止“非锁定”（Nonblocking）模式。默认情况下，所有套接字在创建好后，都会自动进入“锁定”套接字。若随FIONBIO这个I/O控制命令来调用ioctlsocket，那么应设置argp，令其传递指向一个“无符号”（无正负号）长整数的指针；若打算启用非锁定模式，应将那个长整数的值设为一个非零值。而若设为0值，意味着套接字进入锁定模式。如换用新的WSAIoctl命令，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，会将套接字自动设为非锁定模式。调用了其中任何一个函数之后，再有任何将套接字设回锁定模式的企图，都会以失败告终，并返回WSAEINVAL错误。要想将套接字改回锁定模式，应用程序首先必须禁止WSAAsyncSelect。具体的做法是调用WSAAsyncSelect，同时令其lEvent参数等于0。或者调用WSAEventSelect，令lNetworkEvents参数等于0，从而禁止WSAEventSelect。

2. FIONREAD

函数	输入	输出	Winsock版本	说明
两者均可	无	无符号长整数	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自动读入的数据量。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，可返回的数据总量。要注意的是，若使用这种或其他形式的消息预先“窥视”机制，并一定保证能够返回正确的数据量。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一条消息的大小。

3. SIOCATMARK

函数	输入	输出	Winsock版本	说明
两者均可	无	布尔值	1+	判断是否已读取了带外数据

若一个套接字配置成接收带外（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），那么本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，接收调用绝对不会将OOB数据和普通数据搅和在同一个调用中。请参考第7章，那里讲述了OOB数据的详细情况。

9.2.2 其他I/O控制命令

除那些与SSL处理有关的之外，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，就象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Win32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（Windows 2000专用），则在Mstcpip.h中定义。

1. SIO_ENABLE_CIRCULAR-QUEUEING

函数	输入	输出	Winsock版本	说明
WSAIoctl	布尔值	布尔值	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，用于控制基层服务提供者如何对进入的数据报消息加以控制。默认情况下，若负责存放进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“老”消息（最先收到的消息）会被丢弃，以便会新到的消息腾出地方。该命令仅适用于那些同不可靠的、面向消息的协议关联在一起的套接字。倘若在别的什么类型的套接字上使用这个I/O控制命令（比如采用一种面向数据流的协议的套接字），或者服务提供者本身不支持该命令，就会返回错误WSAENOPROTOOPT。要注意的是，该选项目前只能在Windows NT和Windows 2000平台上使用。

该I/O控制命令既可用于打开／关闭循环队列，亦可用于查询选项的最新状态。设置该选项时，只需使用输入参数。若对选项的最新状态进行查询，那么只需提供输出布尔参数。

2. SIO_FIND_ROUTE

函数	输入	输出	Winsock版本	说明
WSAIoctl	SOCKADDR	布尔值	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缓冲，首先便会使其无效。对IPX来说，该调用会引发一个IPX GetLocalTarget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Win32平台上的Microsoft提供者都还没有实现这个I/O控制命令。

3. SIO_FLUSH

函数	输入	输出	Winsock版本	说明
WSAIoct1	无	无	2+	判断是否已读取OOB数据

这个I/O控制命令可丢弃当前与指定套接字对应的发送队列的内容。该选项没有输入或输出参数。目前仅Windows 2000和Windows NT 4 Service Pack 4（SP4）才提供了对它的支持。

4. SIO_BROADCAST_ADDRESS

函数	输入	输出	Winsock版本	说明
WSAIoctl	无	SOCKADDR	2+	为套接字地址家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之地址家庭的广播地址，可在sendto或WSASendTo中使用。这个命令仅适用于Windows NT和Windows 2000操作系统。Windows 98及Windows 98均会返回WSAEINVAL错误。

5. SIO_GET_EXTENSION_FUNCTION_POINTER

函数	输入	输出	Winsock版本	说明
WSAIoctl	GUID	函数指针？	2+	取得基层提供者特有一个函数指针

这个I/O控制命令用于获取具体提供者特有的函数（并不属于Winsock标准规范的一部分）。若选定了一种提供者，程序员便可通过该I/O控制命令来利用其特有的函数，具体的做法是为每个函数都分配一个GUID。随后，应用程序可使用I/O控制命令SIO_GET_EXTENSION_FUNCTION_POINTER，取得指向该函数的一个指针。在头文件Mswsock.h中，定义了由微软自己增添的一系列Winsock函数，包括各自对应的GUID。例如，要查询目前安装的Winsock提供者是否支持TransmitFile函数，可用它的GUID对提供者进行查询。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，不必事先将自己的应用程序同Mswsock.lib库文件链接起来。这样实际可取消在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（微软也是一个“提供者”）。我们开发一个分层的服务提供者时，I/O控制命令是至关重要的一个环节。请参阅第14章，了解有关服务提供者接口的详情。

6. SIO_CHK_QOS

函数	输入	输出	Winsock版本	说明
WSAIoctl	DWORD	DWORD	2+	为指定的套接字设置QOS属性

这个I/O控制命令可用来检查QoS内部的六种状态，目前只有Windows 2000才提供了对它的支持。以下六个标志分别对应于六种不同的状态：

- ALLOWED_TO_SEND_DATA（允许发送数据）。
- ABLE_TO_RECV_RSVP（能够接收RSVP）。
- LINE_RATE（线路速率）。
- LOCAL_TRAFFIC_CONTROL（本地通信控制）。
- LOCAL_QOSABILITY（本地质量保证）。
- END_TO_END_QOSABILITY（端到端质量保证）。

其中，第一个标志ALLOWED_TO_SEND_DATA用在QoS等级建立好之后（使用SIO_SET_QOS建立），但又在接收到任何RSVP预约请求（RESV）消息之前。若收到一条RESV消息，意味着要求的带宽已分配给我们的数据流。在收到RESV消息之前，与套接字对应的数据流只会提供“尽最大努力”（Best-effort）服务。在本书第12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，可检视

当前的“尽最大努力”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”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。

第二个标志是ABLE_TO_RECV_RSVP, 指出主机是否能在指定套接字绑定的那个接口上，接收与处理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“尽最大努力”能使线路通信速率达到多少，单位是每秒多少千位（kbit/s或Kbps）。若线路的通信速率未知，便返回一个 INFO_NOT_AVAIABLE（信息不可用）值。

最后三个标志指出在本地机器或网络上，是否提供了特定的功能。对所有这三个选项来说，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用于判断机器上是否安装了通信控制组件，而且是否能够使用。 LOCAL_QOSABILITY判断本地机器是否支持QoS。最后，END_TO_END_QOSABILITY指出整个本地网络是否具有 QoS能力。

7. SIO_GET_QOS

函数	输入	输出	Winsock版本	说明
WSAIoctl	无	QOS	2+	返回与套接字关联在一起的 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，以便容下整个结构。它有时会比 sizeof(QOS)稍大一些，因为结构中可能包含了提供者特有的一些信息。欲了解QoS的详情，可参考第12章。假如一个套接字的地址家族不支持 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都只能在提供了 QoS相容传送协议的操作系统平台上使用，如 Windows 98和 Windows 2000。

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，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 12章，了解有关 QoS的详情。要注意的是，这个选项和 SIO_GET_QOS都只适用于那些提供了 QoS相容传送协议的平台，如Windows 98和Windows 2000。

9. SIO_MULTIPOINT_LOOPBACK

函数	输入	输出	Winsock版本	说明
WSAIoctl	布尔值	布尔值	2+	设置或调查多播数据是否循环返回套接字

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进入数据。当然，只有在该套接字亦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，可将输入值留为NULL，并提供一个布尔变量，作为输出参数使用。

10. SIO_MULTICAST_SCOPE

函数	输入	输出	Winsock版本	说明
WSAIoctl	整数	整数	2+	设置或获取多播数据的存在时间值

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“无情”地丢弃。要想对这个值进行设定，可在lpvInBuffer中，传递一个希望的TTL值（整数）；否则的话，可在调用 WSAIoctl的同时，令lpvOutBuffer指向一个整数，简单地取得当前的TTL设定。

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，分别设置其TCP“保持活动”周期

这个I/O控制命令可针对每一个不同的连接，设置 TCP的“保持活动”消息发送周期。套接字选项SO_KEEPALIVE也允许发送TCP“保持活动”消息，但其发送间隔时间是在注册表（Registry）里设定的。若更改了注册表的值，同时也会更改机器上所有进程的“保持活动”周期。但若使用这个I/O控制命令，便可分别针对每一个套接字，设置其消息发送周期。要想在指定的一个套接字（已建立连接）上设置“保持活动”消息的发送周期，可初始化一个tcp_keepalive结构，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完全一样的。一旦设好一个“保持活动”周期，便可对当前的设置进行查询。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上使用。

12. SIO_RCVALL

函数	输入	输出	Winsock版本	说明
WSAIoctl	无符号整数	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 IP包。这要求将套接字句柄传递给 WSAIoctl，同时套接字的地址家族必须是 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所谓的 INADDR_ANY。一旦绑定好套接字，同时设好I/O控制命令，便可调用recv或WSARecv，以便返回IP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 IP头的总长字段是一个16位的整数倍，所以理论上的最大长度为 65535字节。但在实际应用中，网络的最大传输单元（MTU）要小得多。若使用这个 I/O控制命令，要求在本地机器上以“管理员”的身份登录。此外，这个I/O命令仅适用于Windows 2000。在本书的配套光盘上，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。

13. SIO_RCVALL_MCAST

函数	输入	输出	Winsock版本	说明
WSAIoctl	无符号整数	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

里，只是传递给WSAIoctl的套接字应当由通过IPPROTO_IGMP指定的协议来创建。另一个区别是此时返回的只有多播数据，而非返回全部 IP数据包。换言之，只有发至目标地址范围224.0.0.0到239.255.255.255之间的IP包，才会返回。该I/O控制命令仅适用于Windows 2000。

14. SIO_RCVALL_IGMPMCAST

函数	输入	输出	Winsock版本	说明
WSAIoctl	无符号整数	无	2+	接收网络上的所有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。

15. SIO_ROUTING_INTERFACE_QUERY

函数	输入	输出	Winsock版本	说明
两者均可	SOCKADDR	无	2+	判断是否已读取OOB数据

利用这个I/O控制命令，我们可找到用来向远程机器发送数据的那个本地接口的地址。远程机器的地址应当以SOCKADDR结构的形式来传递，并通过lpvInBuffer参数来传递这个结构。除此以外，这里应该通过lpvOubBuffer参数，提供一个足够大的缓冲区。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量身定造的一个应用程序应充分考虑到这方面的因素，并相应地采取对策。

另外，程序也不能仅仅依赖自SIO_ROUTING_INTERFACE_QUERY返回的信息。这些信息并不能保证其长时期有效。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，请再次调用SIO_ROUTING_INTERFACE_QUERY，获取最新信息。

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改变后，发出通知

用这个I/O控制命令指出自己希望在本地路由接口发生任何变化后，都向自己发出通知，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，设置一个SOCKADDR结构，对应于自己想查询的那个远程地址。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“锁定”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“非锁定”模式，便会返回WSAEWOULDBLOCK错误。随后，应用程序可通过WSAEventSelect或WSAAsyncSelect，等候“路由更改”这一事件的发生，同时在网络事件们掩码中设置 FD_ROUTING_INTERFACE_CHANGE标志。另外，重叠 I/O亦可用来进行这样的调用。使用这个方法，我们可在WSAOVERLAPPED结构中提供一个事件句柄，它会在路由改变后收到通知。

输入SOCKADDR结构中指定的地址可以是一个具体的地址，亦可使用通配地址：INADDR_ANY，表明发生任何路由更改，自己都想收到通知。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，所以作为提供者（比如微软提供者），提供了忽略来自应用程序输入缓冲区的信息的选项，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收到关于任何变化的通知，然后简单地调用 SIO_ROUTING_INTERFACE_QUERY，看看发生的改变是否会影响到自己的应用程序。

17. SIO_ADDRESS_LIST_QUERY

函数	输入	输出	Winsock版本	说明
WSAIoctl	无	SOCKET_ADDRESS_LIST	2+	返回套接字绑定的一系列接口的列表

这个I/O控制命令用于取得与套接字协议家族相符的一系列本地传送地址的列表，应用程序可与之建立绑定关系。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Win32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，应用程序不可仅仅依赖自这个I/O控制命令返回的信息，来进行相关的操作。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，表明自己想收到与未来的变化有关的通知。一旦地址列表发生改变（收到了通知），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。

18. 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，一旦指定套接字之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。假如套接字处于锁定模式，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。若套接字处于非锁定模式，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“ 路由更

改”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，我们要在WSAOVERLAPPED结构中提供一个事件句柄；一旦路由发生改变，那个句柄便会进入“传信”状态。

19. 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，用于返回与本地机器上每个接口有关的信息。在输入这一块，不需要设置任何东西；而在输出参数中，会返回由一系列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;            /* Interface flags   */
    sockaddr_gen    iiAddress;          /* Interface address */
    sockaddr_gen    iiBroadcastAddress; /* Broadcast address */
    sockaddr_gen    iiNetmask;          /* Network mask      */
} INTERFACE_INFO, FAR * LPINTERFACE_INFO;

#define IFF_UP              0x00000001 /* Interface is up                 */
#define IFF_BROADCAST       0x00000002 /* Broadcast is supported          */
#define IFF_LOOPBACK        0x00000004 /* This is loopback interface      */
#define IFF_POINTTOPOINT    0x00000008 /* This is point-to-point interface*/
#define IFF_MULTICAST       0x00000010 /* Multicast is supported          */

typedef union sockaddr_gen
{
    struct sockaddr Address;
    struct sockaddr_in  AddressIn;
    struct sockaddr_in6 AddressIn6;
} sockaddr_gen;
```

其中，iiFlags成员字段可返回一个标志掩码，指出接口是否可用（IFF_UP），以及支持广播通信（IFF_BROADCAST），还是支持多播通信（IFF_MULTICAST）。此外，它也会指出接口处在“循环返回”模式（IFF_LOOPBACK），还是处在点到点通信模式（IFF_POINTTOPOINT）。在其他三个字段中，则分别包含了接口的地址、广播地址以及相应的网络掩码。

9.2.3 安全套接字层的I/O控制命令

“安全套接字层”（Secure Socket Layer，SSL）命令目前只得到了Windows CE的支持。Windows 95/98、Windows NT以及Windows 2000均未包含具有SSL能力的提供者。除仅适用于Windows CE之外，支持这些选项的唯一Winsock版本只有Winsock 1。

1. SO_SSL_GET_CAPABILITIES

函数	输入	输出	说明
WSAIoctl	无	DWORD	返回Winsock安全提供者的功能

该命令负责接收一系列特殊的标志，指出 Windows套接字安全提供者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）位字字段的一个指针。目前，仅定义了SO_CAP_CLIENT标志。

2. SO_SSL_GET_FLAGS

函数	输入	输出	说明
WSAIoctl	无	DWORD	返回与套接字对应的s信道特有标志

这个命令可返回同一个特定套接字关联在一起的、 s信道所特有的一系列标志。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。

3. SO_SSL_SET_FLAGS

函数	输入	输出	说明
WSAIoctl	DWORD	无	设置套接字s信道特有标志

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（推迟联络）标志，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位。

通常，Winsock安全提供者会在Winsock的connect函数执行“安全联络”操作。然而，假如发现设置了这个标志，联络便会推迟到应用程序执行了 SO_SSL_PERFORM_HANDSHAKE控制代码之后进行。完成联络后，这个标志会被重设。

4. SO_SSL_GET_PROTOCOLS

函数	输入	输出	说明
WSAIoctl	无	SSLPROTOCOLS	返回安全提供者支持的协议列表

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目前所支持的。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。

5. SO_SSL_SET_PROTOCOLS

函数	输入	输出	说明
WSAIoctl	SSLPROTOCOLS	无	设置基层提供者应当支持的一个协议列表

这个I/O控制命令指定一系列协议的列表，均是提供者准备在这个套接字上支持的。其中，输入缓冲区必须是一个指针，指向前述的 SSLPROTOCOLS结构。

6. SO_SSL_SET_VALIDATE_CERT_HOOK

函数	输入	输出	说明
WSAIoctl	SSLVALIDATECERTHOOK	无	为SSL身份凭据的接受设置校验函数

该I/O控制命令可设置一个指针，指向套接字的身份凭据验证挂钩。它用于指定一个回调函数，在自远程通信方收到一系列证书后，由 Windows套接字安全提供者加以调用。在此，输入缓冲区必须是一个指针，指向 SSLVALIDATECERTHOOK结构，如下所述：

```
typedef struct
```

```
{
    SSLVALIDATECERTFUNC     HookFunc;
    LPVOID                  pvArg;
} SSLVALIDATECERTHOOK, *PSSLVALIDATECERTHOOK;
```

其中，HookFunc字段是指向一个证书验证回调函数的指针； pvArg是指向应用程序特有数据的一个指针，可由应用程序用于任何目的。

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安全联络操作

这个I/O控制命令可在一个已连接的套接字上初始化安全联络序列；其中， SSL_FLAG_DEFER_HANDSHAKE标志已在连接前设置好了。数据缓冲区并非必需的，但SL_FLAG_DEFER_HANDSHAKE标志会被重设。

9.2.4 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，以及取得本地接口的 ATM地址，要了解ATM地址机制的详情请参阅第6章。

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，指出系统内总共包含了多少个 ATM设备。每个设备都有一个对应的、独一无二的 ID，范围在0~（该命令返回的数字-1）之间。

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控制命令可取得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，最后会填入一个 ATM_ADDRESS结构，其中包含了可由后续 bind调用使用的一个本地ATM地址。

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永久虚拟回路（ Permanent Virtual Circuit，PVC）关联到一起。PVC在输入缓冲区中指定，采用 ATM_PVC_PARAMS结构的形式。套接字应从属于AF_ATM地址家族。自该函数成功返回之后，应用程序便能开始数据的收发，好象连接已实际地建好（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;
```

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。自该函数成功返回后，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值，它们均是早先在 SIO_ASSOICATE_PVC的条目中定义好的。

9.3 小结

从表面看，数量如此众多的套接字选项及 I/O控制命令似乎会将人搞得晕头转向。但是，只需深入地探索，便会发现通过它们，应用程序的设计可得到极大的简化，可访问具体协议独有的许多特征。此外，还可利用它们对应用程序进行细致的调节。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，比如在 AppleTalk和IrDA（红外线）通信的前提下。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控制命令都能适用于所有版本的 Windows平台。这样一来，一旦涉及跨平台的兼容问题，应用程序的设计便务必非常小心。

第10章　名字注册和解析

本章，我们将全面论述Winsock 2中引入的名字注册和解析模型，它们都是与协议无关的。由于现在已经废弃了Winsock 1中引入的名字注册和解析方法，所以我们将不再对它进行讨论。首先，介绍名字注册和解析的重要性及其用法的背景知识，然后步步深入现有的各种不同的名字注册模型，最后说明Winsock 2中用于解析名字的函数。另外，还谈谈如何注册自己的服务，以供他人查询。

10.1　背景知识

名字注册是一个过程，把一个用户好用的名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（比如 157.54.185.186）非常麻烦。所以他们宁愿把自己的机器命名为一个更容易记的地址，比如“ ajones1 ”。在IP中，一项名为“ 域名命名系统 ”（DNS）会把IP地址映射成相应的名字。我们将在下一节详细地讨论名字空间 。

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，你有一个服务器，它运行的机器地址为157.64.185.186，端口为5000。如果它只在那台机器上运行，就可以把这台服务器的地址硬编码到客户机应用程序中。但如果你需要一个更为动态的方法，即在若干台机器运行的服务器时，就要考虑采用一个容错的分布式应用程序。如果一个服务器崩溃或过于繁忙，另一个应用就开始接替它，为客户机提供服务。这种情况下，要找到服务器事实上在哪个地址运行，是非常令人头疼的。最理想的情况是用若干个地址来注册自己的服务器 —— 命名为“ 容错分布式服务器 ”。另外，大家也许还希望动态更新一个已注册的服务及其地址。这便是名字注册和解析的核心，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字解析的设计。

10.2　名字空间模型

深入Winsock函数之前，需要为大家讲讲大多数协议附带的各种名字空间模型。名字空间提供了一种能力，用一个友好名把具体的协议及其定址属性关联在一起。最常见的名字空间是针对IP的DNS和Novell针对IPX开发的NetWare目录服务（NDS）。这些名字空间在组成和实施各不相同，但它们的有些属性特别有助于我们理解如何通过 Winsock注册和解析名字。

名字空间有三种类型：动态的、静态的和固定的。动态名字空间允许人们即时注册服务。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，动态名字空间依赖于周期性地广播服务信息，表示该服务可继续使用。动态命名空间有：“ 服务声明协议 ”（SAP）（用于NetWare环境）和AppleTalk的“ 名字绑定协议 ”（NBP）名字空间。

这三类名字空间中，静态名字空间的灵活性最小。在静态名字空间内注册一个服务，需要在规定时间内进行手工注册。这意味着无法通过 Winsock用静态名字空间注册一个服务名，因为它只有一种解析法。 DNS是一个静态名字空间。举个例子来说，你可以用 DNS手工把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。

固定名字空间和动态名字空间一样，允许即时注册服务。但和动态名字空间不同的是，固定名字空间把注册信息保留在固定的地方上，比如说磁盘上的一个文件中。只有在服务请求被删除时，固定名字空间才会把这项服务条目删除。它的优点在于灵活，不会连续不断地广播任何一种类型的有用信息。缺点就是如果一个服务行为不佳（或者说编得糟糕），该服务便在不通知名字空间提供者删除其服务条目的情况下，不知所终。从而导致客户机错误地认为该服务仍然可用。NDS是一个固定名字空间。

名字空间的列举

现在，大家已经知道名字空间的各种属性，但一台机器上可用哪些名字空间呢？我们来看看。多数预先定义的名字空间的声明都在Nspapi.h头文件中。每个名字空间都有一个分配所得的整数值。表10-1列出了一些比较常见的名字空间，它们已获支持，并可用于Win32平台。返回的名字空间由工作站上安装的协议决定。比方说，如果一个工作站上没有安装IPX/SPX，就不会返回NS_SAP名字空间。

表10-1 已获支持的名字空间

名字空间	值	说 明
NS_SAP	1	SAP名字空间；用于IPX网络
NS_NDS	2	NDS名字空间；也用于IPX网络
NS_DNS	11	DNS名字空间；多见于TCP/IP网络和互联网
ND_NTDS	32	Windows NT域名空间；运行于Windows 2000的与协议无关的命名空间

在一台机器上安装IPX/SPX时，只支持SAP名字空间查询。如果想注册自己的服务，还需要安装“SAP代理服务”。某些情况下，需要“NetWare的客户机服务”（Client Service of NetWare）把本地的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名字空间。所有这些协议和服务都可通过“控制面板”得以增添。

Winsock 2提供了一种方法，即如何通过程序获得一份列表，表上列出系统上所有可用的名字空间。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
);
```

第一个参数作为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充裕的缓冲区调用的，就会失败，把lpdwBufferLength设为所需要的最小长度，则会导致WSAGetLastError返回WSAEFAULT错误。这个函数将返回的WSANAMESPACE_INFO结构数的多少返回，或在错误之后出现SOCKET_ERROR。WSANAMESPACE_INFO结构描述指定机器上安装的一个独立名字空间。它的格式如下：

```
typedef struct _WSANAMESPACE_INFO {
GUID NSProviderId;
DWORD dwNameSpace;
BOOL fActive;
```

```
DWORD dwVersion;
LPTSTR lpszIdentifier;
} WSANAMESPACE_INFO, *PWSANAMESPACE_INFO,
LPWSANAMESPACE_INFO;
```

事实上，这个结构有两种格式——Unicode和ANSI。Winsock 2头文件针对具体需要，灵活地定义了为WSANAMESPACE_INFO的相应结构。实际应用中，所有结构和 Winsock 2注册和名字解析函数都有两个版本：Unicode和ANSI。该结构的第一个成员 NSProviderId，是一个通用的唯一识别符（GUID），它对这个特殊的名字空间进行描述。 dWNameSpace字段是这个名字空间的整数常量，比如 NS_DNS或NS_NAP。fActive成员是一个布尔值，若是真，就表明该名字空间可用，并准备发出请求；反之则表示提供者未激活，不能发出特别引用提供者的请求。dwVersion字段只对这个提供者的版本进行识别。最后， lpszIdentifier是该提供者的一个描述性的字串识别符。

10.3 服务的注册

下一步便是如何设置自己的服务，并使其有用，让网络上的其他机器都知道它。这就是利用名字空间提供者注册一个服务，这样一来，打算与之通信的客户机就可以对它进行声明或请求。注册一个服务实际上只有两步。第一步是安装一个描述服务特征的 service class（服务类）。

弄清楚服务类和事实上的服务本身之间的区别非常重要的。服务类和服务对两个不相同的概念。比方说，服务类描述哪些名字空间可用来注册自己的服务，该服务的特征有哪些（它是面向连接的，还是无连接的）。至于客户机如何才能建立一个连接，该服务类是无法将其描述出来的。服务类一旦注册，便可注册事实上的服务了，事实上的服务引用的是它所属的那个准确的服务类。一旦发生这种情况，客户机就可执行请求，在服务实例运行的地方进行查找，从而与别的机器通信。

10.3.1 安装服务类

在注册一个服务实例时，需要定义你的服务属于哪个服务类。用哪个名字空间注册这个服务类中的服务，由服务类定义决定。注册服务类的 Winsock函数是WSAInstallServiceClass，它的定义如下：

```
INT WSAInstallServiceClass (LPWSASERVICECLASSINFO lpServiceClassInfo);
```

唯一的参数是lpServiceClassInfo，它指向一个WSASERVICECLASSINFO结构，这个结构定义了这个服务类的属性。它的格式如下：

```
typedef struct _WSAServiceClassInfo {
    LPGUID              lpServiceClassId;
    LPTSTR              lpszServiceClassName;
    DWORD               dwCount;
    LPWSANSCLASSINFO    lpClassInfos;
} WSASERVICECLASSINFO, *PWSASERVICECLASSINFO, LPWSASERVICECLASSINFO;
```

第一个字段是GUID，它对这个特殊的服务类进行唯一性识别。要在这里使用 GUID，有两种方法可生成它。一是利用公用程序 Uuidgen.exe，并为这个服务器类建立一个 GUID。采用这种方法的问题就是：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。鉴于这一不便，第二种方法应运而生。在头文件 Svcguid.h内，有几个宏可生成基于一个简单属性的GUID。比方说，如果你安装SAP服务类（将声明你的IPX应用程序），就可用SVCID_NETWARE宏。唯一的参数便是你为自己的应用程序类分配的 SAP ID编号。SAP ID编号是在NetWare中预先定义好的，比如说，0x4表示文件服务器，而0x7则表示打印服务器。若采用后一种方法，只需一个易于记忆的 SAP ID，利用它生成指定服务类的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 Svcguide.h，其中包含一些针对逆向操作的宏，这些宏都是有用的 —— 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10-2列出了其中最常用的几个宏。头文件中还包括了一些众所周知的端口号（为 FTP和Telnet之类的服务预留的）的常量。

表10-2 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）	通过一个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编口号生成一个GUID

WSASERVICECLASSINFO结构的第二个字段是lpszServiceClassName，它仅仅是这个特定服务类的一个字串名。最后两个字段是描述性的： dwCount字段引用一个数值，表示投递给lpClassInfos字段的WSANSCLASSINFO结构有多少，这些结构对事实上服务所用的名字空间和协议特征进行定义。事实上的服务是指在这个服务类下注册的服务。它的格式如下：

```
typedef struct _WSANSClassInfo {
    LPSTR   lpszName;
    DWORD   dwNameSpace;
    DWORD   dwValueType;
    DWORD   dwValueSize;
    LPVOID  lpValue;
}WSANSCLASSINFO, *PWSANSCLASSINFO, *LPWSANSCLASSINFO;
```

lpszName字段定义服务类处理属性。表 10-3列出了可用的各种属性。其中每个属性都有一个REG_DWORD类的值。

表10-3 服务类型

字 串 值	定义的常量	名字空间	说 明
“SapId”	SERVICE_TYPE_VALUE_SAPID	NS_SAP	SAP ID
“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
“UdpPort”	SERVICE_TYPE_VALUE_UDPPORT	NS_DNS NS_NTDS	UDP端口

dwNameSpace是这个属性所用的名字空间。表 10-3列出了各种服务类型通常使用的名字空间。后三个字段dwValueType、dwValueSize和lpValue，都对真正与服务类型关联的那个值进行了描述。dwValueType字段标识与这个条目关联的数据的类型，所以称之为注册类型值。比如，这个值如果是DWORD，其类型就应该是REG_DWORD。下一个字段dwValueSize，仅

仅是被当作lpValue投递的数据之长度，lpValue是一个数据指针。

至于如何安装一个名为“Widget Server Class”的服务类，下面的代码示例对此进行了解释：

```
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)"Widget Server Class";
sci.dwCount = 4;
sci.lpClassInfos = aNameSpaceClassInfo;

memset(aNameSpaceClassInfo, 0, sizeof(WSANSCLASSINFO) * 4);
// NTDS name space setup
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 name space setup
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;

ret = WSAInstallServiceClass(&sci);
if (ret == SOCKET_ERROR)
{
    printf("WSAInstallServiceClass() failed %d\n", WSAGetLastError());
}
```

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。你设计的服务都属于Widget Server Class这个类，而这个服务类描述的则是常见的属性，这些属性又属于一个服务实例。这个示例中，我们选用 NetWareSAP ID of 200来注册服务类。这样做

只是为了方便。其实，应该用一个任意的 GUID或基于UDP端口号的GUID。另外，当客户机正在端口5150上监听时，该服务可以使用 UDP协议。

下一个注意点是WSASERVICECLASSINFO结构的dwCount字段被设为4。这个示例中，将用SAP名字空间（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：即使是在只用两个名字空间注册这个服务类，但这里却用了四个 WSANSCLASSINFO结构。因为我们为每个名字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。为每个名字空间定义了服务是否面向连接。这个示例中，名字空间是无连接的，因为我们把 SERVICE_TYPE_VALUE_CONN的值设成了一个布尔值 0。我们还针对Windows NT域名空间，通过使用服务类型SERVICE_TYPE_VALUE_UDPPORT，设置了UDP端口号，这个服务一般在这个端口下运行。针对SAP名字空间，我们用服务类型SERVICE_TYPE_VALUE_SAPID设置了自己的服务SAP ID。

针对每一个WSANSCLASSINFO条目，在设置服务类型和值长度时，还必须设置名字空间识别符，服务类型将在应用于这一名字空间。表 10-3中包括服务类型所需的类型，这一示例中结果都是DWORD。最后一步是简单调用 WSAInstallServiceClass，并把它当作一个参数投给WSASERVICECLASSINFO结构。如果WSAInstallServiceClass调用成功，就返回0；反之，则返回 SOCKET_ERROR。如果 WSASERVICECLASSINFO无效或排列有误，WSAGetLastError就会返回WSAEINCAL。如果这个服务类已经存在， WSAGetLastError则返回WSAEALREADY。这种情况下，就可以调用WSARemoveServiceClass，删掉一个服务器类。它的声明如下：

```
INT WSARemoveServiceClass( LPGUID lpServiceClassId );
```

这个函数的唯一参数是指向GUID的指针。这个GUID就是定义具体服务类的GUID。

10.3.2 服务的注册

服务类一旦安装（说明你的服务有哪些常见属性），就可以注册自己的服务实例，这样一来，远程机器上的其他客户机就可使用这一服务了。注册一个服务实例的 Winsock函数是WSASetService。

```
INT WSASetService (
    LPWSAQUERYSET lpqsRegInfo,
    WSAESETSERVICEOP essOperation,
    DWORD dwControlFlags
);
```

第一个参数lpqsRegInfo，是指向WSAQUERYSET结构的指针，该指针定义特定的服务。我们简要说明这个结构。 essOperation参数指定即将发生的行为，比如注册或取消注册。表 10-4对这三个有效标志进行了说明。

第三个参数dwControlFlags，不是0就是SERVICE_MULTIPLE这个标志。如果要在具体的服务实例下注册若干个地址，用这个标志即可。比如，你有一个服务，想在五台机器上运行它。投入WSAService的WSAQUERY结构将引用五个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，可取消注册一个单一的服务实例，这是通过利用 RNRSERVICE_DELETE标志来完成的。表10-5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10-4 设置服务标志

操作标志	含 义
RNRSERVICE_REGISTER	注册一个服务名。对动态名字提供者而言，这个标志的意思是开始动态地声明这个服务。对固定的名字提供者来说，无意义
RNRSERVICE_DEREGISTER	从注册表中删除整个服务。对动态名字提供者而言，意味着中止声明服务。对固定的名字提供者来说，意味着从数据库中删除这个服务。对静态名字提供者来说，无意义
RNRSERVICE_DELETE	只将具体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）。再次提醒大家注意，这个标志只能应用于动态和固定名字提供者

表10-5 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	覆盖现成的服务实例	在具体的地址上增加一条新的服务条目
SERVICE_MULTIPLE	通过增加新地址的方式，更新服务实例	在具体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结构数是0	这是一个错误，将返回WSASERVICE_NOT_FOUND
SERVICE_MULTIPLE	通过删除具体地址的方式，对这个服务进行更新。即便无地址，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	把这个服务彻底从名字空间删除	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，对这个服务进行更新。如果不保留地址，服务就会彻底从名字空间删除	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投入函数。它的格式如下：

```
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字段中包含一个名字服务实例的字串识别符。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。在客户机请求一个服务时，可利用它获得有用的版本信息。lpszComment字段也是可选的。可在这里指定一个任何类型的注释字串。 dwNameSpace字段指定准备用来注册服务的名字空间。如果你只用一个名字空间，就只能用那个值；反之，就用NS_ALL。另外，可以引用一个定制的名字空间提供者（关于怎样编写自己的名字空间，将在第14章论述）。在定制的命名空间提供者这种情况下， dwNameSpace字段设为0，而lpNSProviderId指定代表定制名字空间提供者的GUID。lpszContext字段指定分层式名字空间（NDS ）中的查询起点。

dwNumberOfProtocols和lpafpProtocols字段都是可选的参数，用于限制搜索，只返回所提供协议。dwNumberOfProtocols字段对AFPROTOCOLS结构的数目进行引用，这些结构包含在lpafpProtocols数组中。它的格式如下：

```
typedef struct _AFPROTOCOLS {
    INT iAddressFamily;
    INT iProtocol;
} AFPROTOCOLS, *PAFPROTOCOLS, *LPAFPROTOCOLS;
```

第一个参数iAddressFamily，是AF_INET和AF_IPX之类的地址家族常量。第二个参数iProtocol是源于指定地址家族的协议，比如IPPROTO_TCP和NSPROTO_IPX。

WSAQUERYSET结构中的下一个字段lpszQueryString是可选的，只供支持丰富结构化查询语言（SQL）的名字空间所用，比如说Whois++。这个参数用来指定字串。

注册一个服务时，接下来的这两个参数是最重要的。 dwNumberOfCsAddrs字段只提供了投到lpcsaBuffer中的CSADDR_INFO结构的数目。CSADDR_INFO结构定义地址家族和服务事实上定位的地址。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            iSocketType;
    INT            iProtocol;
} CSADDR_INFO;

typedef struct _SOCKET_ADDRESS {
    LPSOCKADDR lpSockaddr;
    INT        iSockaddrLength;
} SOCKET_ADDRESS, *PSOCKET_ADDRESS, FAR * LPSOCKET_ADDRESS;
```

这一结构中还还包括了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）则是为具体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）。只有名字空间提供者才可以返回一个 BLOB结构。也就是说，在注册一个服务时，你不能增加自己的BLOB结构，令其在客户机查询中返回。

表10-6列出了WSAQUERYSET结构的字段，并根据执行查询还是注册，判断哪些字段是要求的，哪些又是可选的。

表10-6 WSAQUERYSET字段

字 段	查 询	注 册
dwSize	要求	要求
lpszServiceInstanceName	要求字串或“*”	要求
lpServiceClassId	要求	要求
lpVersion	可选	可选
lpszComment	忽略	可选
dwNameSpace	二选一	二选一
lpNSProviderId	必须指定的字段	必须指定字段
lpszContext	可选	可选
dwNumberOfProtocols	0或0以上	0或0以上
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

10.3.3 服务注册示例

这一小节中，将向大家展示如何在 SAP和NTDS这两个名字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。尽管如此，必须了解它的下面这些特性。首先，Windows NT域名空间需要Windows 2000，因为它是以“活动目录”（Active Directory）为基础的。另外，对你打算在上面注册和(／或)查找服务的Windows 2000工作站而言，还意味着必须有一个账号，可以在这个域内访问“活动目录”。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。这意味着你的IP和IPX服务均可以注册在同一个名字空间内。程序清单 10-1解释了注册一个服务实例的所需要的基本步骤。为了简单起见，代码中没有执行错误检查。

程序清单10-1 WSASetService示例

```
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;
//
// Set the IP address of our service
//
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);
//
// Set up the IPX address for our service
//
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;
lpCSAddr[1].LocalAddr.iSockaddrLength = sizeof(sa_ipx);
lpCSAddr[1].RemoteAddr.lpSockaddr = (struct sockaddr *)&sa_ipx;
```

```
lpCSAddr[1].RemoteAddr.iSockaddrLength = sizeof(sa_ipx);

qs.dwNumberOfCsAddrs = 2;

ret = WSASetService(&qs, RNRSERVICE_REGISTER, 0L);
```

程序清单10-1中的示例代码解释了如何设置一个服务实例，这样一来，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。第一个步骤是初始化 WSAQUERY结构。另外，我们还需要为自己的服务实例名字。这里，我们简单地把它叫作 WidgetServer。另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“ WidgetService 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名字空间。由于我们的服务运行于IPX和UDP，因而指定的是 NS_ALL。因为我们指定的是先前存在的名字空间，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把这个结构当作WSAQUERYSET结构中的lpcsaBuffer字段投递。大家将注意到，我们的示例中的确建立了套接字，并在设置SOCKADDR结构之前，把它们绑定到一个本地地址。这是因为我们必需找到客户机打算连接的那个确切的本地地址。比方说，在为服务器建立我们的 UDP时，我们绑定了INADDR_ANY，直到调用 getsocketname，才会把事实上的 IP地址给我们。利用getsocketname返回的信息，我们就可建立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服务的地址。

在为这个基于UDP的服务器设置SOCKADDR_INFO结构之后，我们设置了基于 IPX的服务。在第6章中，大家已知道服务器应该和网络编号绑定在一起，这个网络编号是通过把网络和节点编号设为0而得的。再次提醒大家注意，它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来获得事实上的地址。在填充 IPX的CSADDR_INFO结构时，分别利用套接字类型和协议的 SOCK_DGRAM和NSPROTO_IPX即可。最后一步是把WSAQUERYSET结构中的dwNumberOfAddrs字段设为2，因为有两个地址（UDP和IPX）客户机用它们来建立一个连接。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，在选择注册我们的服务时，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址机。如果你在一台多址机上建立一个绑定INADDR_ANY的基于UDP的服务器，客户机就可以在任何一个接口上与这个服务器建立连接。在IP的情况下，getsockname是不充分的；你必须获得所有的本地 IP接口。获得这类信息的方法要根据各人使用的平台而定。对所有平台而言，常见的方法是调用 gethostbyname，返回一张我们名字的IP地址列表。在Winsock 2下，还可以调用ioctl命令SIO_GET_INTERFACE_LIST。针对Windows 2000平台，可使用ioctl命令SIO_ADDRESS_LIST_QUERY。最后，还可以用附录 B中讨论的IP助手函数。简单的 TCP/IP名字解析和gethostbyname的介绍参见第6章，而ioctl命令则参见第9章。除此以外，附带光盘上的 Rnrcs.c是一个完整翔实的、以多址机为重点的示例。

10.4 服务的查询

现在，大家已经知道如何在名字空间内注册服务了，下面来看看客户机是如何向名字空间查询具体服务，进而获得通信服务的有关信息的。尽管名字解析采用的查询函数有三个之多：WSALookupServiceBegin、WSALookupServiceNext和WSALookupServiceEnd，但和服务注册比起来，仍然要简单些。执行查询的第一步是调用 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
);
```

第一个参数是一个WSAQUERY结构，它把限制加到查询上，比如说限定查询哪些名字空间。第二个参数dwControlFlags，决定查询的深度。表10-7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，在函数返回之后利用。若成功，返回的值就是 0；否则，就返回SOCKET_ERROR。如果一个或一个以上的参数无效， WSAGetLastError就会返回WSAEINVAI。如果在名字空间内找到该服务名，但没有可与所给限制匹配的数据，错误就是 WSANO_DATA。若指定服务不存在，则返回WSASERVICE_NOT_FOUND错误。

表10-7 控制标志

标 志	含 义
LUP_DEEP	在分层式名字空间中，请求深度与第一级相对应
LUP_CONTAINERS	只获得容器对象。该标志只适合分层式名字空间
LUP_NOCONTAINERS	不返回任何容器。该标志只适合分层式名字空间
LUP_FLUSHCACHE	忽略隐藏信息，直接查询名字空间。注意并非所有的名字提供者都隐藏查询
LUP_FLUSHPREVIOUS	令名字提供者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。如果所提供的缓冲区不够，信息就会被丢弃，并检索下一个信息集
LUP_NEAREST	按距离顺序检索结果。注意，这由计算距离公制的名字提供者来决定，因为注册服务时没有为该信息作准备。不要求名字提供者支持这一概念
LUP_RES_SERVICE	指定本地地址在CSADDR_INFO结构中返回
LUP_RETURN_ADDR	获得lpcsaBuffer形式的地址
LUP_RETURN_ALIASES	只获得别名信息。每个别名都会在成功调用 WSALookupService中返回
LUP_RETURN_ALL	获得所有有用的信息
LUP_RETURN_BLOB	获得lpBlob形式的私有数据
LUP_RETURN_COMMENT	获得lpszComment形式的注释
LUP_RETURN_NAME	获得lpszServiceInstanceName形式的名字
LUP_RETURN_TYPE	获得lpServiceClassId形式的类型
LUP_RETURN_VERSION	获得lpVersion形式的版本号

在调用WSALookupServiceBegin时，返回的查询句柄是你投给WSALookupServiceNext的，它会返回你需要的数据。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，除了只支持LUP_FLUSHPREVIOUS外，其含义和 WSALookupServiceBegin是相同的。参数lpdwBufferLength被当作lpqsResults投递，是一个缓冲区长度。由于 WSAQUERYSET结构中应该包含二进制的大型对象（ BLOB）数据。它通常要求你投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)之前，调用WSALookupServiceNext。特别需要注意的是：早期的Winsock实施中，“ 无过多数据 ”的错误代码是WSAENOMORE(10102)，因此，对一个健壮的代码而言，应该还要对这两个错误代码都进行检查。一旦所有数据都返回，或已经完成查询，就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 );
```

10.4.1 怎样对服务进行查询

如何对前一小节中注册的服务进行查询呢？第一件事是设置定义查询所用的 WSAQUERY结构。我们来看看下面的代码：

```
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)
    // Error
```

记住，所有的服务查询都在服务类 GUID的基础上进行，你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初始化成0，在把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（*），通配将返回具体的服务类 GUID的所有服务。接下来，要求查询利用 NS_ALL常量搜索所有的名字空间。最后，设置协议（ IPX和UDP/IP），以便我们的客户机能与之建立连接。这是利用一个由两个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。它的第一个参数是我们的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呢数据。这里，你应该对 LUP_RETURN_ADDR和LUP_RETURN_NAME这两个标志执

行按位和运算，藉此说明自己需要定址和服务名；只有在用通配符（*）来指代服务名时，才需要LUP_RETURN_NAME标志；否则就是已经知道服务名了。最后一个参数是一个识别这个特殊查询的HANDLE变量。成功返回之后，它就会被初始化。

一旦成功打开查询，就可在WSA_E_NO_MORE返回之前，调用WSALookupServiceNext。每个成功调用都会返回符合查询标准的那个服务的相关信息。这一过程的代码如下：

```
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(addr->sin_addr),
            addr->sin_port);
        }
        else if (addr->sa_family == AF_IPX)
        {
            SOCKADDR_IPX *ipxaddr = (SOCKADDR_IPX *)addr;
            printf("%02X%02X%02X%02X.%02X%02X%02X%02X%02X%02X:%04X",
                (unsigned char)ipxaddr->sa_netnum[0],
                (unsigned char)ipxaddr->sa_netnum[1],
                (unsigned char)ipxaddr->sa_netnum[2],
                (unsigned char)ipxaddr->sa_netnum[3],
                (unsigned char)ipxaddr->sa_nodenum[0],
                (unsigned char)ipxaddr->sa_nodenum[1],
                (unsigned char)ipxaddr->sa_nodenum[2],
                (unsigned char)ipxaddr->sa_nodenum[3],
                (unsigned char)ipxaddr->sa_nodenum[4],
                (unsigned char)ipxaddr->sa_nodenum[5],
```

```
                ntohs(ipxaddr->sa_socket));
            }
        }
    }
    WSALookupServiceEnd(hLookup);
```

尽管这段示例代码有点简单，但非常明晰。调用 WSALookupServiceNext只需要一个有效的查询句柄、返回缓冲区的长度和返回缓冲区本身。不需要指定任何控制标志，因为唯一可用于该函数标志只有一个：LUP_FLUSHPREVIOUS。如果我们提供的缓冲区太小，若设置这个标志，该函数返回的结果就会被丢弃。然而，在我们的示例中，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。我们的例子中采用了一个固定长度的缓冲区，相当于WSAQUERYSET结构再加2000个字节的长度。由于你只要求了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，你的应用程序中还应该能对WSAEFAULT错误进行处理。

一旦成功调用 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。我们的查询中，需要服务名和地址； WSAQUERYSET结构中，最重要的字段是 lpszServiceInstanceName和lpcsaBuffer。前者包含服务名，后者则是一个CSADDR_INFO结构数组，其中包含服务的定址信息。 dwNumberOfCsAddrs参数会准确地告诉我们已返回多少地址。上面的示例代码中，我们只是打印了地址。只检查并打印了 IPX和IP地址，因为在你打开查询时，只要求了这两个地址家族。

查询中，如果用通配符（*）来代表服务名，那么每次调用 WSALookupServiceNext，都会返回一个特定的服务实例，它运行于网络上某个地方——当然，前提是实际上已注册了多个服务实例，并且正处于运行状态。一旦服务的所有实例都已返回，就会产生WSA_E_NO_MORE错误，这样你就会中断循环。最后一件事是在查询句柄上调用WSALookupServiceEnd。这样便可释放为查询分配的所有资源。

10.4.2 查询DNS

前面，我们曾提过 DNS名字空间是静态的，这意味着你不能动态地注册自己的服务；但仍可用Winsock名字解析函数来执行DNS查询。实际上，执行DNS查询比执行普通的已注册服务查询要复杂得多，因为DNS名字空间提供者是以BLOB的形式返回查询信息。为什么会这样呢？还记得第6章中对gethostname的讨论吗？“名字检索返回的 HOSTNAME结构中不仅包含了IP地址，还包含了别名”。WSAQUERYSET结构偏偏就例外。

关于BLOB数据，需要注意的一点是：它的格式尚未很好地编入帮助文档，这样就使得直接查询DNS显得有些困难。我们首先来看看如何打开查询。本书附带光盘上的 Dnsquery.c文件包含了直接查询 DNS所用的完整的示例代码；我们逐段地对它们进行分析。下面的代码解释了如何初始化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;
HANDLE       hQuery;

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.lpafProtocols = afp;

ret = WSALookupServiceBegin(&qs, LUP_RETURN_NAME | LUP_RETURN_BLOB,
    &hQuery);
if (ret == SOCKET_ERROR)
    // Error
```

DNS查询的设置非常类似于我们的前一个例子。最显著的变化是我们这里采用的是 GUID SVCD_INET_HOSTADDRBYNAME。这是一个识别主机名查询的GUID。lpszServiceInstanceName是我们准备解析的主机名。由于正在通过 DNS解析主机名，所以我们只需为 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协议定义成我们的查询感兴趣的协议。

查询一旦建立，就可调用WSALookupServiceNext，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)
    // Error
WSALookupServiceEnd(hQuery);

hostent = pqs->lpBlob->pBlobData;
```

因为DNS名字空间提供者以BLOB的形式返回主机信息，所以需要你提供一个足够大的缓冲区。这就是为什么要用一个非常大的缓冲区的原因，其长度相当于一个 WSAQUERYSET加上一个HOSTENT，再加上2048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，所有的主机信息都返回在 HOSTENT结构内，即使主机名与多个IP地址关联在一起。这就是不必多次调用 WSALookupServiceNext的原因。

这个地方需要特别注意——对查询返回的BLOB结构进行解码。通过第 6章的学习，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;
```

HOSTENT结构以BLOB数据的形式返回时，结构内的指针对应的是实际是内存中的偏移位置，实际数据是自那个位置起开始存放的。这个偏移位置是自 BLOB数据的起始处开始算起的。这样一来，我们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。在图10-1中，我们向大家展示了HOSTENT结构以及返回的内存布局。DNS查询是主机名“ 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，使字段能引用到正确位置，我们需要将偏移值加到HOSTENT结构的头地址上。这一计算需要针对 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，引用位置中的每个32位字段都会由偏移值构成。看看图 10-1展示的h_addr_list字段，便会发现初始偏移量是16字节，它指定的是 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。然而，数组中的每一个指针偏移距离是 28字节。要想令其指向正确的位置，需要先取得HOSTENT结构的地址，再在它的基础上加 28字节，指向一个实际的 4字节位置。在那个位置，含有数据 0x9D36B9BA，亦即IP地址157.54.185.186。随后，便可自那个位置开始，在第28个字节的偏移量之后，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。假如该主机名同时对应着几个 IP地址，那么必然存在着其他的偏移量。如法炮制，便能修正指针，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，我们的主机并未设置别名。数组的第一个条目是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，非常容易纠正 —— 只需将偏移量加到HOSTENT结构地址上面就可以了，它指向的是一个空中止字串的起始处。

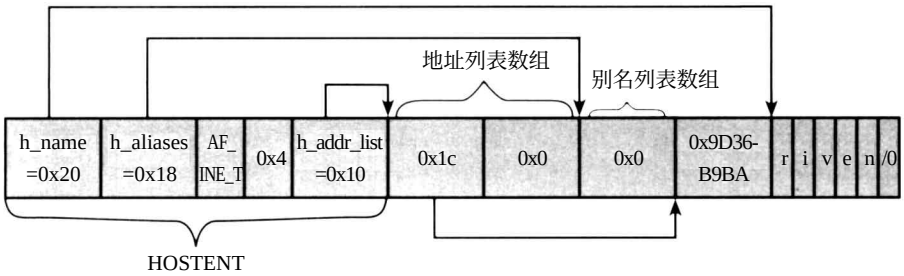

图10-1 HOSTENT BLOB数据

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，具体如下：

```
hostent->h_name = (PCHAR)((DWORD_PTR)hostent->h_name) + (PCHAR)hostent;
```

要修正指针数组（比如 h_aliases 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走一遍，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量上，这个偏移量即后来的条目的值。当然，一旦碰到了一个其值为 0的数组条目，你自然就会停下来。需要对h_addr_list字段如法炮制。一旦偏移得到了修正，就可以正常使用 HOSTENT结构了。

10.5 小结

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。名字注册的真正局限在于名字空间。随着 TCP/IP的风行，DNS曾一度独领风骚，但DNS又欠灵活。直到有了Windows 2000和Windows NT域名空间，我们才有了一个稳定的、与协议无关的名字解析方法，它为编写强健的应用程序提供了充分的灵活性。另外，其他名字空间（比如SAP）也可用于以IPX/SPX为基础的应用程序，它们提供了许多类似于 NTDS的能力（与协议无关这一点除外）。

第11章 多　　播

“多播”亦称“多点传送”（Multicasting），是一种让数据从一个成员送出，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”（Broadcasting）通信的不足。假如过度使用广播技术，极易造成网络带宽的大幅占用，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“有兴趣”，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 Win32平台而言，仅两种可从Winsock内访问的协议（IP和ATM）才提供了对多播通信的支持。通过本章的学习，大家可掌握多播通信的一些基本知识，同时理解如何在这两种协议中实现多播通信。

我们首先探讨多播网络的基本原理，其中涉及多播通信的各种形式，以及 IP及ATM多播通信的基本特征。掌握了最基本的知识后，再分别用两个小节，讲解 IP和ATM特有的一些问题。在本章最后，介绍一些 API调用，用于实现 Winsock 1和Winsock 2中的多播通信。Winsock提供对IP多播的支持已有不短的日子，自第一个版本的 Winsock（Winsock 1）便已开始。到了Winsock 2后，多播接口得到了全面的扩展，具有了“与协议无关”的特点。

支持多播通信的平台包括Windows CE 2.1、Windows 95、Windows 98、Windows NT 4和Windows 2000。自2.1版开始，Windows CE才开始实现对IP多播的支持。在以往的版本中，则没有实现。支持多播的所有平台都能进行 IP多播通信。然而，由于只有在 Windows 98和Windows 2000中，才固化了对ATM Winsock的支持，所以到目前为止，只有这两个平台（含Win98 SE）才能提供对ATM多播通信与生俱来的支持。当然，这并不妨碍我们开发一个 IP多播应用，令其在ATM网络上运行。它对我们造成的唯一限制是：不能编写“与生俱来”或者“固有”的ATM多播代码。这个问题的详情还会在11.2节“IP多播”中讲到。对多播支持来说，另一个需要注意的问题是，某些型号较老的网卡（NIC）不能使用IP多播地址进行数据的发送及接收。1998年之后制造的大多数网卡都提供了对 IP多播的支持，但这也不是绝对的。

11.1 多播的含义

多播通信具有两个层面的重要特征：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，数据如何传送。这两方面的特征既可以是“有根的”（Rooted），也可以是“无根的”（Nonrooted）。在一个“有根的”控制层面内，存在着一个特殊的多播组成员，叫作 c_root（控制根，或根节点）。而剩下的每个组成员都叫作 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的连接。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，对任何一个具体的组来说，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。

而对一个“无根的”控制层面来说，它则允许任何人加入一个组，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），这是通过实施自己的组成员资格协议来实现的。然而，我们的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多播便是无根控制层面的一个典型例子。在图 11-1中，我们向大家展示了有根与无根控制层面的区别。在位于左上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；而在位于右下部分的无根方案中，任何人都能自由加入这个组。

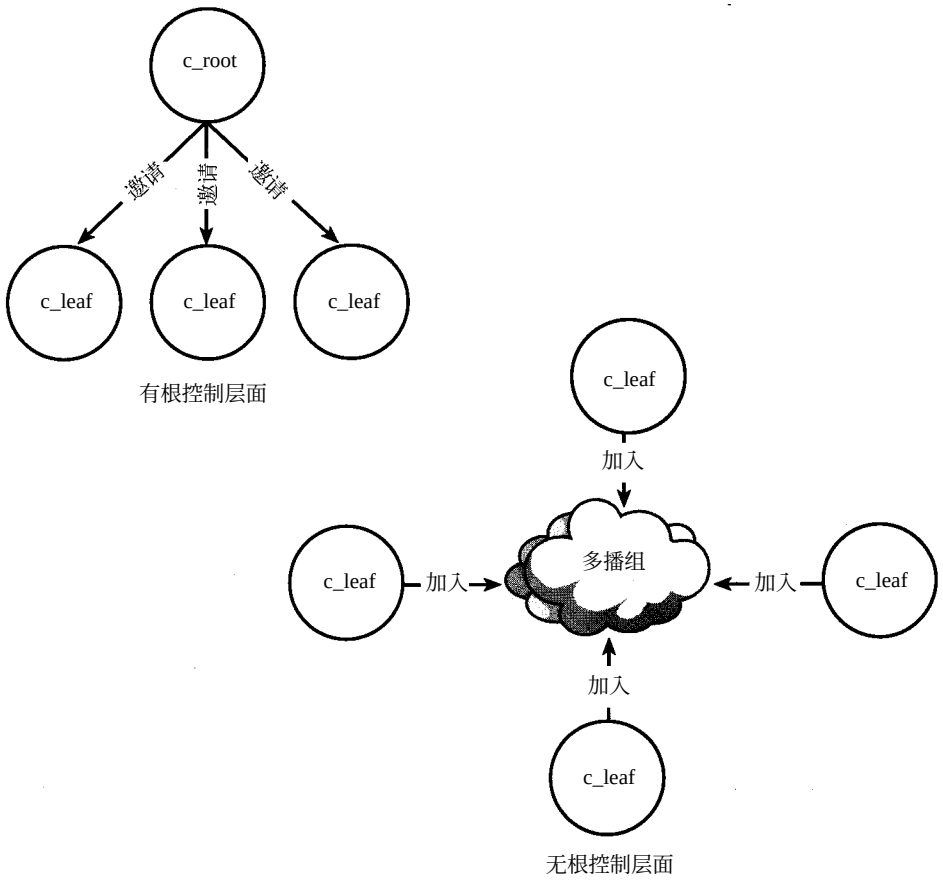

图11-1 有根和无根控制层面

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，它有一个参与者叫作 d_root（数据根，或根节点）。数据传输只能在 d_root和多播会话的其他所有成员之间进行。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，亦可双向进行。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发出的数据却可由每个 d_leaf收到。ATM也是“有根数据层面”的一个典型例子。在图 11-2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却只会由d_root接收到，不会“蔓延”到其他叶节点。与此相反的是右下部分的无根示例。其中，数据abc和xyz会“蔓延”到每个成员，无论最开始是由谁发出的数据。

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会投递给其他所有成员，同时所有接收者都能回送数据。至于谁能接收或发送数据，则不存在任何限制。同样地， IP多播采用的是数据层面上的“ 无根 ”通信方式。

我们现在知道， ATM多播是在控制及数据层面的一种有根通信方式，而 IP多播在两个层面上都是“ 无根 ”的。除此以外，还有可能存在另一些组合形式。例如，我们可能有一个有根控制层面，其中一个节点决定谁能加入组内；另外还有一个无根数据层面，自任何成员发出的数据都可被其他所有成员“ 看到 ”。只不过，在Winsock中，目前尚无任何一种已经支持的协议以这种形式工作。

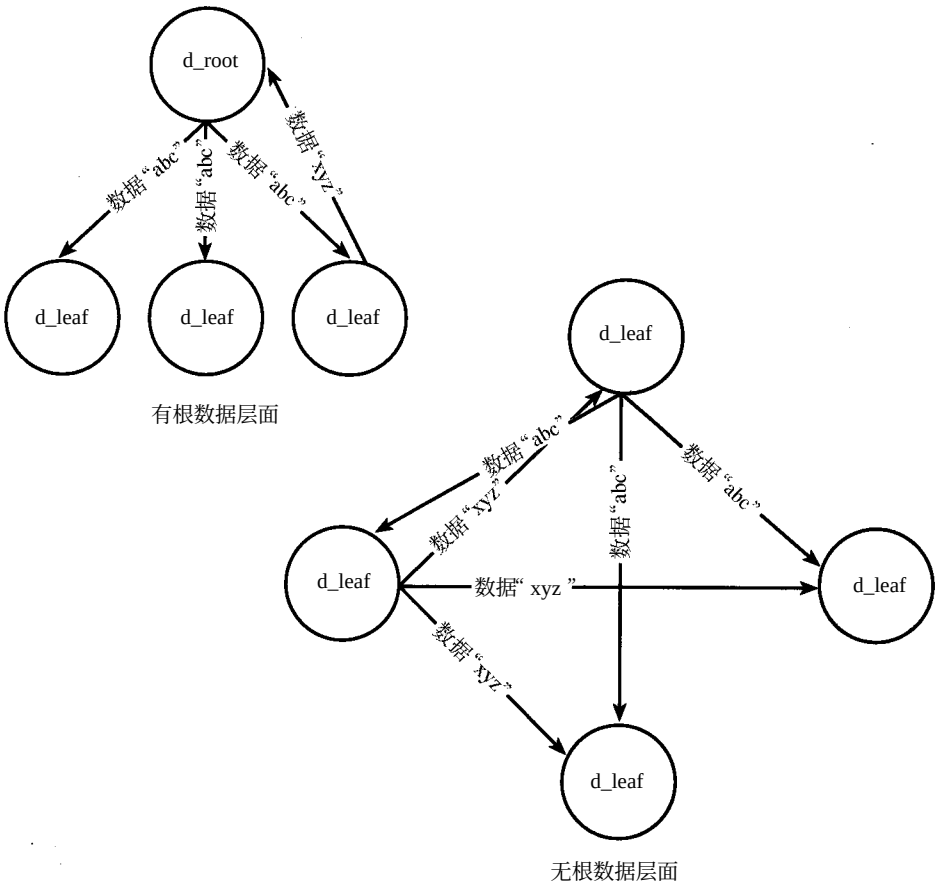

图11-2 有根和无根数据层面

调查多播属性

在第5章，我们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亦可从目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，是我们所特别感兴趣的。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 XP1_MULTIPOINT_DATA_PLANE位已被设置，表

明协议支持的是一种有根数据层面。类似地，假如标志位设为 0，协议支持的便只有一个无根数据层面。

11.2 IP多播

IP多播通信需要依赖一个特殊的地址组，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，假定五个节点都想通过 IP多播，实现彼此间的通信，它们便可加入同一个组地址。全部加入之后，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IP地址是一个D类IP地址，范围在224.0.0.0到239.255.255.255之间。但是，其中还有许多地址是为特殊用途而保留的。比如，224.0.0.0根本没有使用（也不能使用），224.0.0.1代表子网内的所有系统（主机），而224.0.0.2代表子网内的所有路由器。上述最后两个特殊地址只能由 IGMP协议使用，对此我们后面还会详细探讨。在 RFC 1700文件中，提供了所有保留地址的一个详细清单。该文件是为特殊用途保留的所有资源的一个列表，大家可以找来作为参考。“Internet分配数字专家组”（IANA）负责着这个列表的维护。在表 11-1中，我们总结了目前标定为“保留”的一些地址。在实际应用中，可使用除头三个保留多播地址之外的任何地址，因为它们是网络中的路由器专用的。请参考RFC 1700，了解准确的多播地址分配情况。

表11-1 多播地址

多播地址	用 途
224.0.0.0	基本地址（保留）
224.0.0.1	这个子网上的所有系统
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版本组地址
224.0.1.24	WINS服务器组地址

最开始设计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要求保留一系列特殊地址，专门用于多播数据的传输。除此以外，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，请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多播地址了事，否则网络会立即由于数据泛滥而瘫痪。开发 Internet网关管理协议（Internet Gateway Management Protocol, IGMP）的目的正是为了通知路由器：网络中的一台机器对发给一个指定组的数据有兴趣。

11.2.1 Internet网关管理协议

多播主机利用IGMP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IP多播方案的基础。要想使它正常工作，两个多播节点之间的所有路由器都必须提供对IGMP的支持。例如，假定机器 A和B加入了多播组224.1.2.3，两者间总共存在着三个路由器。此时，三个路由器都必须具有 IGMP能力，以保障通信的成功进行。若某个路由器不具备IGMP能力（即不支持IGMP），那么收到多播数据后，会将它草草地丢弃了事。若一个应用加入了多播组，一条 IGMP“加入”命令便会发给子网上那个特殊的“所有路由器”地址

(224.0.0.2)。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，即它们想加入那个多播组。以后，假如路由器收到了发给那个多播地址的数据，便会将其转发给所有多播客户机。

此外，若一个端点加入多播组，便会同时指定一个“存在时间”(TTL)参数。通过该参数，我们便知道对于在端点机器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，令其加入组X，同时TTL值设为2。此时，一个加入命令会传给本地子网上的“所有路由器”组。子网内的路由器会根据这一命令，判断出自己以后应将多播数据转发到那个地址。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(不再传给相邻的网络)。从中可以看出，TTL实际限制了多播数据能够蔓延得多“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”组(224.0.0.1)定时发送一条“组查询”消息，查询当初通过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，便会用另一条IGMP消息作出响应，让路由器放心，以便继续转发与那个地址对应的数据。否则的话，路由器便会停止为那个地址转发任何数据。即便客户机通过任何一种Winsock方法，明确离开了那个多播组，路由器上的过滤器设置仍然不会立即消除。

不幸的是，正是由于上述原因，有时才会产生错误：客户机可能在放弃了多播组A的成员资格后，马上便加入了组B。但另一方面，除非路由器执行了一次组查询，但没有接收到响应，否则会将发给多播组A和B的数据都转发到网上。假如这两个组的传输数据总量大于网络本身允许的带宽，岂不是要糟糕？为此，我们可考虑换用IGMP协议的第2版。这是目前的最新版本，允许客户机向路由器发送一条“离开”消息，明确告诉它停止转发指定多播地址的数据。当然，针对每个特定的地址，路由器都维持着一个参考性的客户机计数。因此，除非子网上的所有客户机都脱离了一个特定的地址，否则发给那个地址的数据仍会继续“蔓延”下去。

Windows 98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中，自Service Pack 4(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，不妨参阅RFC 1112以及RFC 2236这两份标准文档。

11.2.2 IP叶节点

加入IP多播组的过程非常简单，因为每个节点都是一个“叶”，所以在加入一个组的时候，采取的操作步骤都是一样的。由于IP多播在Winsock 1和Winsock 2中都可以实现，所以可通过两种API调用方法，来做成同样的事情。下面是Winsock 1中，实现IP多播通信所需的基本步骤：

1) 用socket函数创建一个套接字，注意要设为AF_INET地址家族以及SOCK_DGRAM套接字类型。要想初始化一个多播套接字，不必设置任何特殊标志，因为socket函数本身便没有提供什么标志参数。

2) 如果想从组内接收数据，请将套接字同一个本地端口绑定到一起。

3) 调用setsockopt函数，同时设置IP_ADD_MEMBERSHIP选项，指定想加入的那个组的

地址结构。

如使用的是Winsock 2，那么步骤1)和2)是相同的，只是步骤3)要改一改——换成调用WSAJoinLeaf函数，将自己加入那个组。至于这两个函数（另一个是setsockopt）以及它们的差别，则将在本章后面11.4节“多播与Winsock”内，进行深入探讨。

请注意，假如一个应用程序只是打算发送数据，便不必加入一个IP多播组。向多播组发送数据时，网络中传输的数据包与普通UDP包大致相同，只是目的地址换成了一个特殊的多播地址而已。但假如想接收多播数据，便必须加入一个组。但无论如何，除了对组成员资格的要求之外，IP多播通信与普通的UDP协议通信并无什么区别：两者都是“无连接”的、“不可靠”的……等等。

11.2.3 IP多播的实施

IP多播通信得到了所有Windows平台的支持（2.1版之前的Windows CE除外）。但另一方面，在各个平台上，IP多播的具体实施方式却稍有差异。我们早先已经指出，使用的网卡首先必须支持多播通信。为此，网卡需要以硬件形式，为接口增加多播过滤器。多播IP地址采用了一个特殊的MAC地址，其中包含了编好码的IP地址，使网卡能轻易判断出进入的数据包是否属于多播数据，同时调查这个包的目标多播IP地址是哪一个。所有这些信息只需检查MAC头便可取得。在RFC 1700中，对具体的编码方案进行了讨论。本书不打算再继续深入下去，因为它与Winsock API并无多大联系。

然而，Windows 95和Windows NT 4的多播实施方式却有所不同，这是通过将网卡置于“混杂”模式来进行的。换句话说，网卡会通过信号线缆取得传来的所有数据包，再一起转交给网络驱动程序，由后者负责检查MAC头的内容，看看是否有多播数据发给一个多播组，而且工作站上的一个进程属于那个组的成员。由于这种过滤实际是由软件（驱动程序）完成的，性能当然比不上用硬件来过滤数据包。因此，Windows 98和Windows 2000采取了不同的实施方式。两者均利用了网卡本身的能力，来实现过滤器的增添。由于在硬件一级执行，所以效能当然要高出许多。目前大多数网卡都能支持16或32个硬件过滤器。对Windows 98来说，唯一的限制是一旦设置了所有硬件过滤器，机器上运行的进程便无法再加入任何新的多播组。假如超出了硬件本身的限制，多播加入操作就会失败，并返回一个WSAENOBUFS错误。在这一方面，Windows 2000显得更为“健壮”。假如所有硬件过滤器都被占用，Windows 2000就会像Windows 95和Windows NT 4那样，将网卡置为“混杂”模式，通过软件实现MAC头的解析。

11.3 ATM多播

通过Winsock实现的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”控制及数据层面。换言之，若建立了一个多播服务器（或者c_root），便能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网络便能仿真一个IP网络，而IP地址实际上会映射成为ATM特有的地址形式。若允许通过ATM来仿真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“正宗”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网络上基本一般无异。唯一的区别在于，在此不必使用IGMP，因为所有多播调用都会转换成ATM自己的命令。另外，我们或许能将一个ATM网络配置成一个或多个“LAN仿真（LANE）”网络。设计LANE最主要的目的便是让ATM表现得像一个“标准”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。这意味着在我们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。在新版的Windows 2000中，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，一项重大差别是ATM不需要特殊地址。唯一的要求是：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，便会自动与原先的多播组脱勾。两者均转移到一个单独的组内。

11.3.1 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、要自己加入一个组的邀请。下面列出必要的一系列步骤：

1) 使用WSASocket函数，创建地址家族AF_ATM的一个套接字，同时设置WSA_FLAG_MULTIPOINT_C_LEAF和WSA_FLAG_MULTIPOINT_D_LEAF这两个标志。

2) 将套接字同本地ATM地址及端口绑定到一块儿，这是用bind函数完成的。

3) 调用listen（监听）命令。

4) 用accept或WSAAccept等候邀请。至于具体用哪个命令，要取决于我们使用的是什么I/O（输入／输出）模型。要想全盘了解Winsock I/O模型，请参阅本书第8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，在任何指定时刻，Windows 98和Windows 2000仅能支持一个ATM叶节点。换言之，对任何ATM“点到多点”会话来说，在整个系统的范围之内，只能有一个进程成为它的叶成员。

11.3.2 ATM根节点

根节点的创建过程甚至还要比ATM叶的创建简单一些。基本步骤如下：

1) 使用WSASocket函数，创建地址家族AF_ATM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。

11.4 多播与Winsock

现在，大家已基本熟悉了Windows平台的两类基本多播通信方式。接下来，且让我们来看看Winsock提供了哪些API调用，以便真正实现多播。针对IP多播，Winsock提供了两种不同的实现方法，具体取决于使用的是哪个版本的Winsock。第一种方法是Winsock 1提供的，要求通过套接字选项来加入一个组。Winsock 2则引入了一个新函数，专门负责多播组的加入。这个函数便是WSAJoinLeaf，它与基层协议是无关的。下面，我们首先要讲述的是Winsock 1方法，也是应用得最广的一种方法（特别由于它是自Berkeley套接字衍生出来的）。

11.4.1 Winsock 1多播

在Winsock 1设计的方法中，IP多播组的加入和离开是用setsockopt命令来完成的，同时要用到IP_ADD_MEMBERSHIP（加入组）和IP_DROP_MEMBERSHIP（脱离组）这两个选项。使用这两个套接字选项时，必须传递一个ip_mreq结构，它的定义如下：

```
struct ip_mreq
{
    struct in_addr  imr_multiaddr;
    struct in_addr  imr_interface;
};
```

其中，imr_multiaddr字段用于指定要加入的多播组，而imr_interface指定要通过它送出多播数据的本地（或本机）接口。如果将imr_interface设为INADDR_ANY，便会自动使用默认接口；否则，请指定本地接口具体要使用哪个IP地址（假如同时有多个IP地址可选）。下面示范代码向大家展示了如何加入一个多播组234.5.6.7：

```
SOCKET            s;
struct ip_mreq    ipmr;
SOCKADDR_IN       local;
int               len = sizeof(ipmr);

s = socket(AF_INET, SOCK_DGRAM, 0);

local.sin_family = AF_INET;
local.sin_addr.s_addr = htonl(INADDR_ANY);
local.sin_port = htons(10000);
ipmr.imr_multiaddr.s_addr = inet_addr("234.5.6.7");
ipmr.imr_interface.s_addr = htonl(INADDR_ANY);

bind(s, (SOCKADDR *)&local, sizeof(local));
setsockopt(s, IPPROTO_IP, IP_ADD_MEMBERSHIP,
    (char *)&ipmr, &len);
```

要离开一个多播组，只需调用setsockopt，同时设置IP_DROP_MEMBERSHIP。注意仍然要传递一个ip_mreq结构，其中的值与加入那个组时是一样的。如下例所示：

```
setsockopt(s, IPPROTO_IP, IP_DROP_MEMBERSHIP,
    (char *)&ipmr, &len);
```

Winsock 1 IP多播示例

在程序清单11-1中，我们提供了一个简单的IP多播示范程序。该程序首先加入一个指定的组，然后扮演一个发送者或接收者（具体由命令行参数决定）。写这个程序时，我们让一个发

送者只负责数据的发送，而一个接收者只是利用一个循环结构，简单地等候进入的数据。显然，按照这一设计，我们没有展示出假如发送者将数据发给多播组；同时也作为那个组的成员，对进入数据进行监听的情况。发送的数据会返还给发送者自己的“进入数据”队列。我们称这种情况为“多播返还”（Multicast Loopback）。要注意的是，只有在我们使用IP多播通信时，才会出现这一情况。本章稍后，还会讲到如何禁止进行这种“返还”。

程序清单11-1 Winsock 1.1多播示例

```
// Module name: Mcastws1.c

#include <windows.h>
#include <winsock.h>

#include <stdio.h>
#include <stdlib.h>

#define MCASTADDR      "234.5.6.7"
#define MCASTPORT       25000
#define BUFSIZE         1024
#define DEFAULT_COUNT   500

BOOL  bSender = FALSE,       // Act as sender?
      bLoopBack = FALSE;     // Disable loopback?

DWORD dwInterface,           // Local interface to bind to
      dwMulticastGroup,      // Multicast group to join
      dwCount;               // Number of messages to send/receive

short iPort;                 // Port number to use
//
// Function: usage
//
// Description:
//    Print usage information and exit
//
void usage(char *progname)
{
    printf("usage: %s -s -m:str -p:int -i:str -l -n:int\n",
        progname);
    printf(" -s     Act as server (send data); otherwise\n");
    printf("          receive data.\n");
    printf(" -m:str Dotted decimal multicast IP addres to join\n");
    printf("          The default group is: %s\n", MCASTADDR);
    printf(" -p:int Port number to use\n");
    printf("          The default port is: %d\n", MCASTPORT);
    printf(" -i:str Local interface to bind to; by default \n");
    printf("          use INADDRY_ANY\n");
    printf(" -l     Disable loopback\n");
    printf(" -n:int Number of messages to send/receive\n");
    ExitProcess(-1);
}

//
// Function: ValidateArgs
```

```
//
// Description:
//    Parse the command line arguments, and set some global flags,
//    depending on the values
//
void ValidateArgs(int argc, char **argv)
{
    int      i;

    dwInterface = INADDR_ANY;
    dwMulticastGroup = inet_addr(MCASTADDR);
    iPort = MCASTPORT;
    dwCount = DEFAULT_COUNT;

    for(i = 1; i < argc; i++)
    {
        if ((argv[i][0] == '-') || (argv[i][0] == '/'))
        {
            switch (tolower(argv[i][1]))
            {
                case 's':   // Sender
                    bSender = TRUE;
                    break;
                case 'm':   // Multicast group to join
                    if (strlen(argv[i]) > 3)
                        dwMulticastGroup = inet_addr(&argv[i][3]);
                    break;
                case 'i':   // Local interface to use
                    if (strlen(argv[i]) > 3)
                        dwInterface = inet_addr(&argv[i][3]);
                    break;
                case 'p':   // Port number to use
                    if (strlen(argv[i]) > 3)
                        iPort = atoi(&argv[i][3]);
                    break;
                case 'l':   // Disable loopback?
                    bLoopBack = TRUE;
                    break;
                case 'n':   // Number of messages to send/recv
                    dwCount = atoi(&argv[i][3]);
                    break;
                default:
                    usage(argv[0]);
                    break;
            }
        }
    }
    return;
}

//
// Function: main
//
// Description:
//    Parse the command line arguments, load the Winsock library,
```

```
//    create a socket, and join the multicast group. If this program
//    is started as a sender, begin sending messages to the
//    multicast group; otherwise, call recvfrom() to read messages
//    sent to the group.
//
int main(int argc, char **argv)
{
    WSADATA             wsd;
    struct sockaddr_in  local,
                        remote,
                        from;
    struct ip_mreq      mcast;
    SOCKET              sockM;
    TCHAR               recvbuf[BUFSIZE],
                    sendbuf[BUFSIZE];
int                 len = sizeof(struct sockaddr_in),
                    optval,
                    ret;
DWORD               i=0;

// Parse the command line, and load Winsock
//
ValidateArgs(argc, argv);

if (WSAStartup(MAKEWORD(1, 1), &wsd) != 0)
{
    printf("WSAStartup failed\n");
    return -1;
}
// Create the socket. In Winsock 1, you don't need any special
// flags to indicate multicasting.
//
if ((sockM = socket(AF_INET, SOCK_DGRAM, 0)) == INVALID_SOCKET)
{
    printf("socket failed with: %d\n", WSAGetLastError());
    WSACleanup();
    return -1;
}
// Bind the socket to the local interface. This is done so
// that we can receive data.
//
local.sin_family = AF_INET;
local.sin_port   = htons(iPort);
local.sin_addr.s_addr = dwInterface;

if (bind(sockM, (struct sockaddr *)&local,
    sizeof(local)) == SOCKET_ERROR)
{
    printf("bind failed with: %d\n", WSAGetLastError());
    closesocket(sockM);
    WSACleanup();
    return -1;
}

// Set up the im_req structure to indicate what group we want
```

```
// to join as well as the interface
//
remote.sin_family      = AF_INET;
remote.sin_port        = htons(iPort);
remote.sin_addr.s_addr = dwMulticastGroup;
mcast.imr_multiaddr.s_addr = dwMulticastGroup;
mcast.imr_interface.s_addr = dwInterface;

if (setsockopt(sockM, IPPROTO_IP, IP_ADD_MEMBERSHIP,
    (char *)&mcast, sizeof(mcast)) == SOCKET_ERROR)
{
    printf("setsockopt(IP_ADD_MEMBERSHIP) failed: %d\n",
        WSAGetLastError());
    closesocket(sockM);
    WSACleanup();
    return -1;
}
// Set the TTL to something else. The default TTL is 1.
//
optval = 8;
if (setsockopt(sockM, IPPROTO_IP, IP_MULTICAST_TTL,
    (char *)&optval, sizeof(int)) == SOCKET_ERROR)
{
    printf("setsockopt(IP_MULTICAST_TTL) failed: %d\n",
        WSAGetLastError());
    closesocket(sockM);
    WSACleanup();
    return -1;
}
// Disable the loopback if selected. Note that in Windows NT 4
// and Windows 95 you cannot disable it.
//
if (bLoopBack)
{
    optval = 0;
    if (setsockopt(sockM, IPPROTO_IP, IP_MULTICAST_LOOP,
        (char *)&optval, sizeof(optval)) == SOCKET_ERROR)
    {
        printf("setsockopt(IP_MULTICAST_LOOP) failed: %d\n",
            WSAGetLastError());
        closesocket(sockM);
        WSACleanup();
        return -1;
    }
}

if (!bSender)           // Client
{
    // Receive some data
    //
    for(i = 0; i < dwCount; i++)
    {
            if ((ret = recvfrom(sockM, recvbuf, BUFSIZE, 0,
                (struct sockaddr *)&from, &len)) == SOCKET_ERROR)
            {
```

```
                printf("recvfrom failed with: %d\n",
                    WSAGetLastError());
                closesocket(sockM);
                WSACleanup();
                return -1;
            }
            recvbuf[ret] = 0;
            printf("RECV: '%s' from <%s>\n", recvbuf,
                inet_ntoa(from.sin_addr));
        }
    }
    else                    // Server
    {
        // Send some data
        //
        for(i = 0; i < dwCount; i++)
        {
            sprintf(sendbuf, "server 1: This is a test: %d", i);
            if (sendto(sockM, (char *)sendbuf, strlen(sendbuf), 0,
                (struct sockaddr *)&remote,
                sizeof(remote)) == SOCKET_ERROR)
            {
                printf("sendto failed with: %d\n",
                    WSAGetLastError());
                closesocket(sockM);
                WSACleanup();
                return -1;
            }
            Sleep(500);
        }
    }
    // Drop group membership
    //
    if (setsockopt(sockM, IPPROTO_IP, IP_DROP_MEMBERSHIP,
        (char *)&mcast, sizeof(mcast)) == SOCKET_ERROR)
    {
        printf("setsockopt(IP_DROP_MEMBERSHIP) failed: %d\n",
            WSAGetLastError());
    }
    closesocket(sockM);

    WSACleanup();
    return 0;
}
```

用Winsock 1的方法来实现多播通信时，必须特别注意：使用正确的文件和库进行链接。如果装载的是Winsock 1.1库，便应将Winsock.h包括进来，并同Wsock32.lib建立链接关系。假如装载的是 Winsock 2或更高版本的库，便需同时包括 Winsock.h和Ws2tcpip.h，并同Ws2_32.lib建立链接关系。这样做是必需的，因为针对下述常数，同时存在着两套不同的值：IP_ADD_MEMBERSHIP、IP_DROP_MEMBERSHIP、IP_MULTICAST_IF和IP_MULTICAST_LOOP。而这些值原来的规定——由Stephen Deering制订——却从未正式集成到Winsock规范中来。所以到了 Winsock 2之后，这些值发生了改变。当然，假如使用的是老版本的Winsock，那么只需建立与 wsock32.lib的链接，便万事大吉。转到 Winsock 2机器上

运行时，常数会转换成正确的值。

11.4.2 Winsock 2多播

Winsock 2多播通信要比Winsock 1多播通信稍微复杂一些。但是，前者提供了一些有用的机制，可支持一些用于提供附加特性的协议，比如“服务质量”（QoS）等等。另外，它支持的一些协议还提供了“有根多播”能力。套接字选项在新版的 Winsock中不再用于对组成员关系进行初始化。相反，我们要用一个新增的函数（ WSAJoinLeaf）来做这件事情。该函数的原型定义如下：

```
SOCKET WSAJoinLeaf(
    SOCKET s,
    const struct sockaddr FAR * name,
    int namelen,
    LPWSABUF lpCallerData,
    LPWSABUF lpCalleeData,
    LPQOS lpSQOS,
    LPQOS lpGQOS,
    DWORD dwFlags
);
```

其中，第一个参数s代表一个套接字句柄，是自 WSASocket返回的。传递进来的这个套接字必须使用恰当的多播标志进行创建；否则的话， WSAJoinLeaf就会失败，并返回错误WSAEINVAL。必须记住的是，我们需要指定两个多点传送标志：一个用于指出该套接字在控制层面是“有根”的，还是“无根”的；另一个用于指出该套接字在数据层面是“有根”的，还是“无根”的。用于指定控制层面的标志是 WSA_FLAG_MULTIPOINT_C_ROOT和WSA_FLAG_MULTIPOINT_C_LEAF，而用于指定数据层面的标志是 WSA_FLAG_MULTIPOINT_D_ROOT和WSA_FLAG_MULTIPOINT_D_LEAF。第二个参数是SOCKADDR（套接字地址）结构，具体内容由当前采用的协议决定。对有根控制方案来说（比如 ATM），这个地址指定了打算邀请加入的客户机；而对无根控制方案来说（比如 IP），这个地址指定的是主机打算加入的那个多播组。 namelen（名字长度）参数的意义并不难猜：用于指定 name参数的长度，以字节为单位。 lpCallerData（呼叫者数据）参数的作用是在会话建好之后，将一个数据缓冲区传输给和自己通信的对方。而 lpCalleeData（被叫者数据）参数用于初始化一个缓冲区，在会话建好之后，接收来自对方的数据。注意在当前的 Windows平台上，这两个参数并未真正实现，所以均应设为 NULL。lpSQOS参数用于指定一个FLOWSPEC结构，指出应用程序要求多大的带宽（第 12章对QoS或服务质量进行了详细讲述；简单地说，带宽越大，网络应用享受的服务质量就越高）。lpGQOS参数在此会被忽略，因为当前没有任何一种Windows平台支持所谓的“套接字组”。而最后一个参数dwFlags指出该主机是发送数据、接收数据或收发兼并。为此，该参数的可选值分别是：JL_SENDER_ONLY、JL_RECEIVER_ONLY或者JL_BOTY。

针对同多播组绑定到一起的套接字，该函数会返回一个 SOCKET描述符。假如WSAJoinLeaf调用是通过一个异步（或非锁定）套接字进行的，那么除非加入操作完成，否则返回的套接字描述符是没有用的。例如，假定套接字处于异步模式（来自 WSAAsyncSelect或WSAEventSelect调用），那么若非原来的套接字s接收到一个对应的FD_CONNECT通知，否则描述符便是无效的。注意 FD_CONNECT通知只有在有根控制方案中才会生成。在这种方案

中，name参数指定了一个特定端点的地址。在表 11-2中，我们总结了在什么样的前提下，应用程序才会接收到一个 FD_CONNECT通知。通过在原来的套接字上调用 closesocket，我们可为这些非锁定模式取消一个“ 待决 ”的加入请求。而一个多点会话中的根节点可调用 WSAJoinLeaf一次或者多次，以便增加新的叶节点。但大多数时候，一次只能有一个多点连接请求处于“ 待决 ”状态。

如先前所述，对非锁定套接字来说，一次加入请求不能立即完成。同时，除非请求完成，否则返回的套接字描述符是无法使用的。假如我们通过 ioctlsocket命令FIONBIO将套接字置入非锁定模式，对 WSAJoinLeaf的调用便不会返回 WSAEWOULDBLOCK错误，因为函数实际已返回了一个“ 成功启动 ”的指示。注意在一个异步 I/O模型中，要想接收有关成功启动的通知，唯一的方法便是通过 FD_CONNECT消息。大家可参考第 8章，了解WSAAsyncSelect和 WSAEventSelect异步I/O模型的详情。对于处在锁定模式中的套接字来说，无论 WSAJoinLeaf操作是成功还是失败，都不能向应用程序发出相应的通知。换言之，我们或许应该避免直接的非锁定套接字，因为除非在后续的 Winsock调用中实际用到套接字，否则便无法确切判断一个多播加入操作是否成功（若加入操作失败，使用时也会失败）。

WSAJoinLeaf返回的套接字描述符是各不相同的，具体取决于输入套接字是一个根节点，还是一个叶节点。如果是根节点，那么根据 name参数，我们就可知道打算邀请加入多点会话的一个特定的“ 叶 ”的地址。为了使 c_root（根节点）能够对各个叶的成员关系（成员资格）进行维护， WSAJoinLeaf会为叶返回一个新的套接字句柄。这个新套接字与邀请时使用的根套接字完全相同。其中包括通过异步 I/O模型注册的任何异步事件，比如 WSAEventSelect和 WSAAsyncSelect。然而，这些新的套接字只能用来从“ 叶 ”那里收取 FD_CLOSE通知消息。发给多播组的任何数据都应通过 c_root套接字送出。某些情况下，我们可通过由 WSAJoinLeaf返回的套接字送出数据，但只有与那个套接字对应的“ 叶 ”才会真正收到数据。 ATM协议便允许这样干。最后，要想从多播组中删去一个叶节点，那么作为“ 根 ”，只需在与那个叶对应的套接字上调用closesocket就可以了。

另一方面，如果随一个叶节点调用 WSAJoinLeaf，那么name参数对应的要么是一个根节点的地址，要么是一个多播组的地址。前一种情况指定一次由叶发起的加入，目前尚无任何一种协议提供了对它的支持（然而，以后的 ATM UNI4.0规范会提供这方面的支持）。后一种情况指定的则是 IP多播通信的工作方式。但无论在哪种情况下，自 WSAJoinLeaf返回的套接字句柄与作为s传递的套接字句柄都是相同的。如果对 WSAJoinLeaf的调用是一次由叶节点发起的加入，那么根节点会用 bind，listen和accept/WSAAccept方法来“ 监听 ”进入的连接请求。上述几种方法对一个服务器来说，应该是大家早已耳熟能详的！若应用程序想把自己从多点会话中删去，那么需要在套接字上调用 closesocket，用它终止成员关系，同时也释放出被占用的套接字资源。在表 11-2中，我们也总结了根据控制层面的类型，根据套接字传递的参数，同时根据name参数，需采取什么样的行动。这些行动包括：函数调用成功之后，是否返回一个新的套接字描述符；以及应用程序是否会收到一个 FD_CONNECT通知等。

若应用程序调用accept或者WSAAccept函数，以便等候由根发起的一次邀请（此时自己的身份是“ 叶 ”），或者等候来自叶的加入请求（此时的身份是“ 根 ”），函数便会返回一个套接字，亦即一个c_leaf套接字描述符，与WSAJoinLeaf返回的类似。若协议既允许由根发起加入，也允许由叶发起加入，那么为了与这种协议适应，对已处在“ 监听 ”模式的一个 c_root套接字

来说，完全可将其当作WSAJoinLeaf的一个输入来使用。

发出了对WSAJoinLeaf的调用之后，会返回一个新的套接字句柄。该句柄并不用于数据的收发。对于原来的套接字句柄来说（自 WSASocket返回，传递进入WSAJoinLeaf），它会在所有的数据发送及接收操作中使用。新句柄指出我们是指定多播组的一名成员 —— 只要句柄有效。若针对新句柄调用 closesocket，会使我们的应用程序脱离多播组，失去它的成员资格。或在一个c_root套接字上调用closesocket，那么凡是采用了异步 I/O模型的c_leaf节点都会收到一个FD_CLOSE通知。

表11-2 WSAJoinLeaf行动总结

控制层面	s	name参数（名字）	行 动	是否收到FD_CONNECT通知？	返回的套接字描述符
有根的	c_root	叶地址	由根邀请一个叶加入	是	用于FD_CLOSE通知，以及用于将私人数据传给那个叶s的一个副本
	c_leaf	根地址	由叶发起与根的连接	是	
无根的	c_root	N/A	不可能的一种组合	N/A	N/A
	c_leaf	组地址	叶加入一个组	否	s的一个副本

1. Winsock 2 IP多播示例

在程序清单11-2中，我们向大家展示了Mcastws2.c的源码清单。该程序实际解释了如何使用WSAJoinLeaf命令，来加入及脱离一个IP多播组。这实际是在一个简单的Winsock 1 IP多播示例基础上改编而成的。那个例子叫作 Mcastws1.c。新程序的不同之处在于，加入／离开调用已进行了修改，改为使用本节讲述的 WSAJoinLeaf。

程序清单11-2 IP多播示例

```
// Module: Mcastws2.c
//
#include <winsock2.h>
#include <ws2tcpip.h>

#include <stdio.h>
#include <stdlib.h>

#define MCASTADDR     "234.5.6.7"
#define MCASTPORT      25000
#define BUFSIZE        1024
#define DEFAULT_COUNT  500

BOOL  bSender = FALSE,       // Act as a sender?
      bLoopBack = FALSE;     // Disable loopback?

DWORD dwInterface,           // Local interface to bind to
      dwMulticastGroup,      // Multicast group to join
      dwCount;               // Number of messages to send/receive

short iPort;                 // Port number to use

//
```

```
// Function: usage
//
// Description:
//    Print usage information and exit
//
void usage(char *progname)
{
    printf("usage: %s -s -m:str -p:int -i:str -l -n:int\n",
        progname);
    printf("   -s        Act as server (send data); otherwise\n");
    printf("              receive data.\n");
    printf("   -m:str    Dotted decimal multicast IP addres "
       "to join\n");
    printf("              The default group is: %s\n", MCASTADDR);
    printf("   -p:int    Port number to use\n");
    printf("              The default port is: %d\n", MCASTPORT);
    printf("   -i:str    Local interface to bind to; by default \n");
    printf("              use INADDRY_ANY\n");
    printf("   -l        Disable loopback\n");
    printf("   -n:int    Number of messages to send/receive\n");
    ExitProcess(-1);
}

//
// Function: ValidateArgs
//
// Description:
//    Parse the command line arguments, and set some global flags,
//    depending on the values
//
void ValidateArgs(int argc, char **argv)
{
    int      i;

    dwInterface = INADDR_ANY;
    dwMulticastGroup = inet_addr(MCASTADDR);
    iPort = MCASTPORT;
    dwCount = DEFAULT_COUNT;

    for(i=1; i < argc ;i++)
    {
        if ((argv[i][0] == '-') || (argv[i][0] == '/'))
        {
            switch (tolower(argv[i][1]))
            {
                case 's':  // Sender
                    bSender = TRUE;
                    break;
                case 'm':  // Multicast group to join
                    if (strlen(argv[i]) > 3)
                        dwMulticastGroup = inet_addr(&argv[i][3]);
                    break;
                case 'i':  // Local interface to use
                    if (strlen(argv[i]) > 3)
                        dwInterface = inet_addr(&argv[i][3]);
```

```
                    break;
                case 'p':  // Port to use
                    if (strlen(argv[i]) > 3)
                        iPort = atoi(&argv[i][3]);
                    break;
                case 'l':  // Disable loopback
                    bLoopBack = TRUE;
                    break;
                case 'n':  // Number of messages to send/recv
                    dwCount = atoi(&argv[i][3]);
                    break;
                default:
                    usage(argv[0]);
                    break;
            }
        }
    }
    return;
}

//
// Function: main
//
// Description:
//    Parse the command line arguments, load the Winsock library,
//    create a socket, and join the multicast group. If this program
//    is run as a sender, begin sending messages to the multicast
//    group; otherwise, call recvfrom() to read messages sent to the
//    group.
//
int main(int argc, char **argv)
{
    WSADATA             wsd;
    struct sockaddr_in  local,
                        remote,
                        from;
    SOCKET              sock, sockM;
    TCHAR               recvbuf[BUFSIZE],
                        sendbuf[BUFSIZE];
    int                 len = sizeof(struct sockaddr_in),
                        optval,
                        ret;
DWORD                   i=0;

// Parse the command line, and load Winsock
//
ValidateArgs(argc, argv);

if (WSAStartup(MAKEWORD(2, 2), &wsd) != 0)
{
    printf("WSAStartup() failed\n");
    return -1;
}
// Create the socket. In Winsock 2, you do have to specify the
// multicast attributes that this socket will be used with.
```

```
//
if ((sock = WSASocket(AF_INET, SOCK_DGRAM, 0, NULL, 0,
              WSA_FLAG_MULTIPOINT_C_LEAF
            | WSA_FLAG_MULTIPOINT_D_LEAF
            | WSA_FLAG_OVERLAPPED)) == INVALID_SOCKET)
{
    printf("socket failed with: %d\n", WSAGetLastError());
    WSACleanup();
    return -1;
}
// Bind to the local interface. This is done to receive data.
local.sin_family = AF_INET;
local.sin_port   = htons(iPort);
local.sin_addr.s_addr = dwInterface;

if (bind(sock, (struct sockaddr *)&local,
    sizeof(local)) == SOCKET_ERROR)
{
    printf("bind failed with: %d\n", WSAGetLastError());
    closesocket(sock);
    WSACleanup();
    return -1;
}
// Set up the SOCKADDR_IN structure describing the multicast
// group we want to join
//
remote.sin_family      = AF_INET;
remote.sin_port        = htons(iPort);
remote.sin_addr.s_addr = dwMulticastGroup;
//
// Change the TTL to something more appropriate
//
optval = 8;
if (setsockopt(sock, IPPROTO_IP, IP_MULTICAST_TTL,
    (char *)&optval, sizeof(int)) == SOCKET_ERROR)
{
    printf("setsockopt(IP_MULTICAST_TTL) failed: %d\n",
        WSAGetLastError());
    closesocket(sock);
    WSACleanup();
    return -1;
}
// Disable loopback if needed
//
if (bLoopBack)
{
    optval = 0;
    if (setsockopt(sock, IPPROTO_IP, IP_MULTICAST_LOOP,
        (char *)&optval, sizeof(optval)) == SOCKET_ERROR)
    {
        printf("setsockopt(IP_MULTICAST_LOOP) failed: %d\n",
            WSAGetLastError());
        closesocket(sock);
        WSACleanup();
        return -1;
```

```
    }
}
// Join the multicast group.  Note that sockM is not used
// to send or receive data. It is used when you want to
// leave the multicast group. You simply call closesocket()
// on it.
//
if ((sockM = WSAJoinLeaf(sock, (SOCKADDR *)&remote,
    sizeof(remote), NULL, NULL, NULL, NULL,
    JL_BOTH)) == INVALID_SOCKET)
{
    printf("WSAJoinLeaf() failed: %d\n", WSAGetLastError());
    closesocket(sock);
    WSACleanup();
    return -1;
}

if (!bSender)           // Receiver
{
    // Receive data
    //
    for(i = 0; i < dwCount; i++)
    {
            if ((ret = recvfrom(sock, recvbuf, BUFSIZE, 0,
                (struct sockaddr *)&from, &len)) == SOCKET_ERROR)
            {
                printf("recvfrom failed with: %d\n",
                    WSAGetLastError());
                closesocket(sockM);
                closesocket(sock);
                WSACleanup();
                return -1;
            }
            recvbuf[ret] = 0;
            printf("RECV: '%s' from <%s>\n", recvbuf,
                inet_ntoa(from.sin_addr));
        }
    }
    else                    // Sender
    {
        // Send data
        //
        for(i = 0; i < dwCount; i++)
        {
            sprintf(sendbuf, "server 1: This is a test: %d", i);
            if (sendto(sock, (char *)sendbuf, strlen(sendbuf), 0,
                (struct sockaddr *)&remote,
                sizeof(remote)) == SOCKET_ERROR)
            {
                printf("sendto failed with: %d\n",
                    WSAGetLastError());
                closesocket(sockM);
                closesocket(sock);
                WSACleanup();
                return -1;
```

```
            }
            Sleep(500);
        }
    }
    // Leave the multicast group by closing sock.
    // For nonrooted control and data plane schemes, WSAJoinLeaf
    // returns the same socket handle that you pass into it.
    //
    closesocket(sock);

    WSACleanup();
    return -1;
}
```

2. Winsock 2 ATM多播示例

在程序清单11-3中，我们列出了Mcastatm.c的源码清单，这是一个简单的ATM多播示范程序，用于阐释一次“由根发起的”加入。本列表并未包括文件 Support.c。在那个文件内，包括了在多播示例中用到的一些例程，但却并非多播通信所独有的，比如 GetATMAddress等等——用于返回本地接口的ATM地址。

看看程序清单11-3展示的代码，便会发现对于多播根来说，最重要的部分便是Server函数，它通过一个循环，为命令行指定的每个客户机都调用一次 WSAJoinLeaf。针对加入的每个客户机，服务器都维持了一个套接字数组。然而，在 WSASend调用中，使用的却是主套接字。假如ATM协议提供了对它的支持，而且感兴趣的是接收“秘密交谈”信息，便可选择在叶套接字上进行监听（比如自 WSAJoinLeaf返回的套接字）。换句话说，假定客户机在连好的套接字上发出数据，服务器会在自 WSAJoinLeaf返回的、相应的句柄上收到。其他客户机则收不到这种数据。

而对客户机来说，请注意观察 Client函数。可以发现，唯一的要求便是将客户机同一个本地接口绑定起来，并等候服务器通过 accept或WSAAccept调用发来的邀请。一旦邀请抵达，新建的套接字便可用来接收由“根”发出的数据。

程序清单11-3 ATM多播示例

```
// Module: Mcastatm.c

#include "Support.h"

#include <stdio.h>
#include <stdlib.h>


#define BUFSIZE                 1024
#define MAX_ATM_LEAF               4

#define ATM_PORT_OFFSET         ((ATM_ADDR_SIZE * 2) - 2)
#define MAX_ATM_STR_LEN         (ATM_ADDR_SIZE * 2)

DWORD   dwAddrCount = 0,
        dwDataCount = 20;
BOOL    bServer = FALSE,
        bLocalAddress = FALSE;
```

```
char   szLeafAddresses[MAX_ATM_LEAF][MAX_ATM_STR_LEN + 1],
       szLocalAddress[MAX_ATM_STR_LEN + 1],
       szPort[3];
SOCKET sLeafSock[MAX_ATM_LEAF];
// Module: usage
//
// Description:
//    Print usage information
//
void usage(char *progname)
{
    printf("usage: %s [-s]\n", progname);
    printf("     -s       Act as root\n");
    printf("     -l:str   Leaf address to invite (38 chars)\n");
    printf("              May be specified multiple times\n");
    printf("     -i:str   Local interface to bind to (38 chars)\n");
    printf("     -p:xx    Port number (2 hex chars)\n");
    printf("     -n:int   Number of packets to send\n");
    ExitProcess(1);
}

// Module: ValidateArgs
//
// Description:
//    Parse command line arguments
//
void ValidateArgs(int argc, char **argv)
{
    int      i;

    memset(szLeafAddresses, 0,
        MAX_ATM_LEAF * (MAX_ATM_STR_LEN + 1));
    memset(szPort, 0, sizeof(szPort));

    for(i = 1; i < argc; i++)
    {
        if ((argv[i][0] == '-') || (argv[i][0] == '/'))
        {
            switch (tolower(argv[i][1]))
            {
                case 's':     // Server
                    bServer = TRUE;
                    break;
                case 'l':     // Leaf address
                    if (strlen(argv[i]) > 3)
                    {
                        strncpy(szLeafAddresses[dwAddrCount++],
                            &argv[i][3], MAX_ATM_STR_LEN - 2);
                    }
                    break;
                case 'i':     // Local interface
                    if (strlen(argv[i]) > 3)
                    {
                        strncpy(szLocalAddress, &argv[i][3],
                            MAX_ATM_STR_LEN - 2);
```

```
                        bLocalAddress = TRUE;
                    }
                    break;
                case 'p':     // Port address to use
                    if (strlen(argv[i]) > 3)
                        strncpy(szPort, &argv[i][3], 2);
                    break;
                case 'n':     // Number of packets to send
                    if (strlen(argv[i]) > 3)
                        dwDataCount = atoi(&argv[i][3]);
                    break;
                default:
                    usage(argv[0]);
                    break;
            }
        }
    }
    return;
}

//
// Function: Server
//
// Description:
//    Bind to the local interface, and then invite each leaf
//    address that was specified on the command line.
//    Once each connection is made, send some data.
//
void Server(SOCKET s, WSAPROTOCOL_INFO *lpSocketProtocol)
{
    // Server routine
    //
    SOCKADDR_ATM  atmleaf, atmroot;
    WSABUF        wsasend;
    char          sendbuf[BUFSIZE],
                  szAddr[BUFSIZE];
    DWORD         dwBytesSent,
                  dwAddrLen = BUFSIZE,
                  dwNumInterfaces,
                  i;
    int           ret;
// If no specified local interface is given, pick the
// first one
//
memset(&atmroot, 0, sizeof(SOCKADDR_ATM));
if (!bLocalAddress)
{
    dwNumInterfaces = GetNumATMInterfaces(s);
    GetATMAddress(s, 0, &atmroot.satm_number);
}
else
    AtoH(&atmroot.satm_number.Addr[0], szLocalAddress,
        ATM_ADDR_SIZE - 1);
//
// Set the port number in the address structure
```

```
//
AtoH(&atmroot.satm_number.Addr[ATM_ADDR_SIZE-1], szPort, 1);
//
// Fill in the rest of the SOCKADDR_ATM structure
//
atmroot.satm_family                 = AF_ATM;
atmroot.satm_number.AddressType     = ATM_NSAP;
atmroot.satm_number.NumofDigits     = ATM_ADDR_SIZE;
atmroot.satm_blli.Layer2Protocol    = SAP_FIELD_ANY;
atmroot.satm_blli.Layer3Protocol    = SAP_FIELD_ABSENT;
atmroot.satm_bhli.HighLayerInfoType = SAP_FIELD_ABSENT;
//
// Print out what we're binding to, and bind
//
if (WSAAddressToString((LPSOCKADDR)&atmroot,
    sizeof(atmroot), lpSocketProtocol, szAddr,
    &dwAddrLen))
{
    printf("WSAAddressToString failed: %d\n",
        WSAGetLastError());
}
printf("Binding to: <%s>\n", szAddr);

if (bind(s, (SOCKADDR *)&atmroot,
    sizeof(SOCKADDR_ATM)) == SOCKET_ERROR)
{
    printf("bind() failed: %d\n", WSAGetLastError());
    return;
}
// Invite each leaf
//
for(i = 0; i < dwAddrCount; i++)
{
    // Fill in the SOCKADDR_ATM structure for each leaf
    //
    memset(&atmleaf, 0, sizeof(SOCKADDR_ATM));
    AtoH(&atmleaf.satm_number.Addr[0], szLeafAddresses[i],
        ATM_ADDR_SIZE - 1);
    AtoH(&atmleaf.satm_number.Addr[ATM_ADDR_SIZE - 1],
        szPort, 1);

    atmleaf.satm_family                 = AF_ATM;
    atmleaf.satm_number.AddressType     = ATM_NSAP;
    atmleaf.satm_number.NumofDigits     = ATM_ADDR_SIZE;
    atmleaf.satm_blli.Layer2Protocol    = SAP_FIELD_ANY;
    atmleaf.satm_blli.Layer3Protocol    = SAP_FIELD_ABSENT;
    atmleaf.satm_bhli.HighLayerInfoType = SAP_FIELD_ABSENT;
    //
    // Print out client's address, and then invite it
    //
    if (WSAAddressToString((LPSOCKADDR)&atmleaf,
        sizeof(atmleaf), lpSocketProtocol, szAddr,
        &dwAddrLen))
    {
        printf("WSAAddressToString failed: %d\n",
```

```
                WSAGetLastError());
        }
        printf("[%02d] Inviting: <%s>\n", i, szAddr);

        if ((sLeafSock[i] = WSAJoinLeaf(s,
            (SOCKADDR *)&atmleaf, sizeof(SOCKADDR_ATM), NULL,
            NULL, NULL, NULL, JL_SENDER_ONLY))
            == INVALID_SOCKET)
        {
            printf("WSAJoinLeaf() failed: %d\n",
                WSAGetLastError());
            WSACleanup();
            return;
        }
    }
    // Note that the ATM protocol is a bit different from TCP.
    // When the WSAJoinLeaf (or connect) call completes, the
    // peer has not necessarily accepted the connection yet,
    // so immediately sending data will result in an error;
    // therefore, we wait for a short time.
    //
    printf("Press a key to start sending.");
    getchar();
    printf("\n");

        //
        // Now send some data to the group address, which will
        // be replicated to all clients
        //
        wsasend.buf = sendbuf;
        wsasend.len = 128;
        for(i = 0; i < dwDataCount; i++)
        {
            memset(sendbuf, 'a' + (i % 26), 128);
            ret = WSASend(s, &wsasend, 1, &dwBytesSent, 0, NULL,
                NULL);
            if (ret == SOCKET_ERROR)
            {
                printf("WSASend() failed: %d\n", WSAGetLastError());
                break;
            }
            printf("[%02d] Wrote: %d bytes\n", i, dwBytesSent);
            Sleep(500);
        }

        for(i = 0; i < dwAddrCount; i++)
            closesocket(sLeafSock[i]);
        return;
    }

    //
    // Function: Client
    //
    // Description:
    //    First the client binds to the local interface (either one
    //    specified on the command line or the first local ATM address).
```

```
//    Next it waits on an accept call for the root invitation. It
//    then waits to receive data.
//
void Client(SOCKET s, WSAPROTOCOL_INFO *lpSocketProtocol)
{
    SOCKET       sl;
    SOCKADDR_ATM atm_leaf,
                 atm_root;
    DWORD        dwNumInterfaces,
                 dwBytesRead,
                 dwAddrLen=BUFSIZE,
                 dwFlags,
                 i;
    WSABUF       wsarecv;
    char         recvbuf[BUFSIZE],
                 szAddr[BUFSIZE];
int          iLen = sizeof(SOCKADDR_ATM),
             ret;

// Set up the local interface
//
memset(&atm_leaf, 0, sizeof(SOCKADDR_ATM));
if (!bLocalAddress)
{
    dwNumInterfaces = GetNumATMInterfaces(s);
    GetATMAddress(s, 0, &atm_leaf.satm_number);
}
else
    AtoH(&atm_leaf.satm_number.Addr[0], szLocalAddress,
        ATM_ADDR_SIZE-1);
AtoH(&atm_leaf.satm_number.Addr[ATM_ADDR_SIZE - 1],
    szPort, 1);
//
// Fill in the SOCKADDR_ATM structure
//
atm_leaf.satm_family                 = AF_ATM;
atm_leaf.satm_number.AddressType     = ATM_NSAP;
atm_leaf.satm_number.NumofDigits     = ATM_ADDR_SIZE;
atm_leaf.satm_blli.Layer2Protocol    = SAP_FIELD_ANY;
atm_leaf.satm_blli.Layer3Protocol    = SAP_FIELD_ABSENT;
atm_leaf.satm_bhli.HighLayerInfoType = SAP_FIELD_ABSENT;
//
// Print the address we're binding to, and bind
//
if (WSAAddressToString((LPSOCKADDR)&atm_leaf,
    sizeof(atm_leaf), lpSocketProtocol, szAddr,
    &dwAddrLen))
{
    printf("WSAAddressToString failed: %d\n",
        WSAGetLastError());
}
printf("Binding to: <%s>\n", szAddr);

if (bind(s, (SOCKADDR *)&atm_leaf, sizeof(SOCKADDR_ATM))
        == SOCKET_ERROR)
```

```
{
    printf("bind() failed: %d\n", WSAGetLastError());
    return;
}
listen(s, 1);
//
// Wait for the invitation
//
    memset(&atm_root, 0, sizeof(SOCKADDR_ATM));
    if ((sl = WSAAccept(s, (SOCKADDR *)&atm_root, &iLen, NULL,
        0)) == INVALID_SOCKET)
    {
        printf("WSAAccept() failed: %d\n", WSAGetLastError());
        return;
    }
    printf("Received a connection!\n");

    // Receive some data
    //
    wsarecv.buf = recvbuf;
    for(i = 0; i < dwDataCount; i++)
    {
        dwFlags = 0;
        wsarecv.len = BUFSIZE;
        ret = WSARecv(sl, &wsarecv, 1, &dwBytesRead, &dwFlags,
            NULL, NULL);
        if (ret == SOCKET_ERROR)
        {
            printf("WSARecv() failed: %d\n", WSAGetLastError());
            break;
        }
        if (dwBytesRead == 0)
            break;
        recvbuf[dwBytesRead] = 0;
        printf("[%02d] READ %d bytes: '%s'\n", i, dwBytesRead,
            recvbuf);
    }
    closesocket(sl);
    return;
}

//
// Function: main
//
// Description:
//    This function loads Winsock library, parses command line
//    arguments, creates the appropriate socket (with the right
//    root or leaf flags), and starts the client or the server
//    functions, depending on the specified flags
//
int main(int argc, char **argv)
{
    WSADATA             wsd;
    SOCKET              s;
    WSAPROTOCOL_INFO    lpSocketProtocol;
```

```
    DWORD               dwFlags;
ValidateArgs(argc, argv);
//
// Load the Winsock library
//
if (WSAStartup(MAKEWORD(2, 2), &wsd) != 0)
{
    printf("WSAStartup failed\n");
    return -1;
}
// Find an ATM-capable protocol
//
if (FindProtocol(&lpSocketProtocol) == FALSE)
{
    printf("Unable to find ATM protocol entry!\n");
    return -1;
}
// Create the socket using the appropriate root or leaf flags
//
if (bServer)
    dwFlags = WSA_FLAG_OVERLAPPED
              | WSA_FLAG_MULTIPOINT_C_ROOT
              | WSA_FLAG_MULTIPOINT_D_ROOT;
else
    dwFlags = WSA_FLAG_OVERLAPPED
              | WSA_FLAG_MULTIPOINT_C_LEAF
              | WSA_FLAG_MULTIPOINT_D_LEAF;

if ((s = WSASocket(FROM_PROTOCOL_INFO, FROM_PROTOCOL_INFO,
        FROM_PROTOCOL_INFO, &lpSocketProtocol, 0,
        dwFlags)) == INVALID_SOCKET)
{
    printf("socket failed with: %d\n", WSAGetLastError());
    WSACleanup();
    return -1;
}
// Start the correct driver, depending on which flags were
// supplied on the command line
//
if (bServer)
{
    Server(s, &lpSocketProtocol);
}
else
{
    Client(s, &lpSocketProtocol);
}
closesocket(s);
    WSACleanup();
    return 0;
}
```

11.4.3 常用的Winsock选项

无论 Winsock 1还是 Winsock 2，IP多播都有三个专用的套接字选项，分别是：

IP_MULTICAST_TTL，IP_MULTICAST_IF和IP_MULTICAST_LOOP。这些选项以往在Winsock 1中常常用于多播组的加入和离开。但对新版的 Winsock 2函数来说，它们仍然非常有用。当然，所有这三个套接字选项均是IP多播所独有的。

1. IP_MULTICAST_TTL

该选项用于设置多播数据的TTL（存在时间）值。在默认情况下， TTL值为1。也就是说，多播数据遇到第一个路由器，便会被它“无情”地丢弃，不允许传出本地网络之外，即只有同一个网络内的多播成员才会收到数据。假如增大 TTL的值，多播数据就可经历多个路由器传到其他网络。TTL的值是多少，最多便能经过多少个路由器。假如路由器收到一个数据包，并判断出自己应将这个包转发给相邻的网络，那么 TTL值随即会被减1。若减1之后，发现TTL的新值变成了0，路由器便会将这个包即时地丢弃，因为它的“存在时间”已经到了。若多播数据报的目标地址在224.0.0.0到224.0.0.255之间，那么多播路由器不会对其进行转发，无论它们的TTL有多大。这个地址范围是为路由协议以及其他低级拓扑查找和维护协议（比如网关查找和组成员关系报告）所保留的，不可擅用。

若调用setsockopt，level参数便是IPPROTO_IP，optname参数是IP_MULTICAST_TTL，而optval参数是一个整数，用于指定新的 TTL值。下述代码片断阐述了具体如何设置 TTL的值：

```
int    optval;

optval = 8;
if (setsockopt(s, IPPROTO_IP, IP_MULTICAST_TTL, (char *)&optval,
        sizeof(int)) == SOCKET_ERROR)
{
    // Error
}
```

除了这个套接字选项之外，对于 WSAIoctl或ioctlsocket这两个函数来说，还可以使用SIO_MULTICAST_SCOPE选项，它可对套接字采取同样的操作。

注意，在ATM多播环境中， TTL参数是不必要的。因为数据在 ATM上的传送只是一个单向过程，所有接收者都是已知的。由于控制层面是“有根”的，所以 c_root节点必须明确地邀请每个叶节点加入。这就意味着，我们不必对数据传输的范围进行限制，数据不会“流窜”到没有任何多播成员的网络上。

2. IP_MULTICAST_IF

这个选项用于设置自哪个IP接口发出多播数据。正常情况下（即非多播环境），只能参照路由表来决定一个数据报自哪个接口发出。此时，依据一个特定数据报本身以及它的目的地，系统可判断出哪个接口最适合用来发出这个数据报。然而，由于多播地址可由任何人使用，所以仅仅依靠路由表的自动判断是不够的。程序员必须多少施加一些自己的影响。当然，只有打算发出多播数据的那台机器已通过网卡建立了与多个网络的连接，才需要考虑到这方面的问题。对于这个选项， optval参数指定的是一个本地接口的地址，多播数据以后便会从这个接口发出。请参考下述代码：

```
DWORD    dwInterface;

dwInterface = inet_addr("129.121.32.19");
if (setsockopt(s, IPPROTO_IP, IP_MULTICAST_IF, (char *)&dwInterface,
```

```
    sizeof(DWORD)) == SOCKET_ERROR)
{
    // Error
}
```

可以看出，在上例中，我们将本地接口设为 129.121.32.19。对通过套接字s发出的任何多播数据而言，都会通过分配了那个IP地址的网络接口送出。

再次提醒大家， ATM并不需要单独用一个套接字选项来设置接口。调用 WSAJoinLeaf之前，c_root可同一个特定的接口明确绑定到一起。类似地，客户机必须同一个特定的 ATM接口绑定到一起，以便等候随accept或WSAAccept发出的加入邀请。

3. IP_MULTICAST_LOOP

最后一个套接字选项决定了应用程序是否接收自己的多播数据。如应用程序加入了一个多播组，并向那个组发送数据，应用程序便会接收到那些数据。数据送出的时候，假如有一个recvfrom调用正处于“ 待决 ”状态，调用便会返回那个数据的一个副本。要注意的是，若只是想让应用程序将数据发给一个多播组，那么它并不一定非要加入多播组。只有在希望接收发给那个组的数据时，才需要加入那个组。设计 IP_MUTLICAST_LOOP套接字选项的目的便是禁止将数据返还给本地接口。使用这个选项，我们可将一个整数值作为 optval参数传递，它其实仅仅是一个“ 布尔 ”值，用于指出到底允许还是禁止多播返还。请来看看下述示范代码：

```
int    optval;

optval = 0;     // Disable loopback
if (setsockopt(s, IPPROTO_IP, IP_MULTICAST_LOOP, (char *)&optval,
    sizeof(int)) == SOCKET_ERROR)
{
    // Error
}
```

另外，与 WSAIoctl或ioctlsocket配套使用的一个 ioctl命令（SIO_MULTIPOINT_LOOPBACK）亦可做到同样的事情。但不幸的是，这个套接字选项并未在 Windows 95、Windows 98或Windows NT 4中实现。而且在默认情况下，多播数据的“ 返还 ”是允许的。假如用这个套接字选项禁止了数据返还，便会收到一个 WSAENOPROTOOPT错误。

按照定义， ATM根节点（亦即允许向多播组发送数据的唯一成员）不会接收到自己发出的数据，因为根套接字并非一个叶套接字。在 ATM多播环境中，只有叶套接字才能接收数据。当初用来创建c_root的过程亦可用来创建一个独立的 c_leaf节点，并由c_root邀请加入。然而，这却并非一个真正意义上的数据“ 返还 ”（Loopback）。

11.4.4 拨号网络多播的一处限制

若试图在一个“ 远程访问服务 ”（RAS）接口或拨号接口上进行多播数据的收发，一处限制务必小心在意。这一限制实际是由拨号连接的那个服务器造成的。目前，大多数以Windows为基础的拨号服务器运行的都是 Windows NT 4操作系统，它们通常没有提供一个IGMP代理。这便意味着，我们发出的任何组成员加入请求都无法离开 Windows NT 4服务器。这样一来，我们的应用程序便不能加入任何多播组，所以也无法进行多播数据的收发。与Windows 2000配套提供的RAS服务器则包含了一个IGMP代理，只是在默认情况下并未开启。

但是，只要启用了这个代理，拨号客户机除了能加入多播组之外，还能进行多播数据的收发。

11.5 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，并向大家展示了各种不同的多播模型。随后，我们探讨了 IP多播和ATM点到多点通信如何应用于这些模型。最后，我们解释了Winsock API具体如何提供对多播通信的支持。同时介绍了 Winsock 1和Winsock 2的方法。前者使用的是套接字选项，而后者使用的是更为合理的 WSAJoinLeaf函数。

第12章　常规服务质量

当今，随着多媒体技术的普遍应用，同时由于 Internet的广泛流行，许多网络都越来越不堪重负。基本的原因便是各种应用（特别是多媒体应用）对带宽的要求越来越大，以至于出现了“ 供不应求 ”的局面。在所谓的共享媒体网络上（如以太网），这个问题尤其突出，因为所有通信数据都有着相同的地位。即使一个非常简单的应用，也可能造成数据在网络上泛滥成灾，造成整个网络的瘫痪。为此，人们提出了“ 服务质量 ”（Quality of Service，简称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，便可根据实际需要，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 ”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。
- 为用户分派资源访问的优先级。

常规服务质量（Generic Quality of Service，GQOS）是微软对QoS的一种实施方案。目前，微软已提供了具有QoS能力的TCP/IP及UDP/IP提供者，可在Windows 98及Windows 2000上使用。要注意的是，ATM本身便已提供了对QoS的支持。

本章将向大家介绍QoS的原理，以及它在Win32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）进行区分对待，哪些组件是必要的。随后，我们将探讨如何利用Winsock接口来写程序，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提供服务。本章的大部分内容都围绕 IP网络上的QoS展开。在本章末，我们将讨论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，我们都会把“ 服务质量 ”简称为QoS。此外，读者完全可以假定我们讨论的都是微软实施的这一套QoS。

12.1　背景知识

QoS需要三个组件才能正常发挥作用：

- 网络上的设备：比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：可为自己引入的网络传输分派相应的优先级。
- 策略组件：谁能使用可用的带宽，以及允许多少人使用。

然而，在我们深入讨论这些组件之前，首先还是来看看“ 资源预约协议 ”（Resource Reservation Protocol，RSVP）的问题。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，而且是 QoS之三个主要组件的大集成者。

12.1.1　资源预约协议

可将RSVP想象成一种“ 粘合剂 ”，它负责将网络、应用以及策略组件粘合成一个整体。

通过一个网络，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，允许将资源从所有RSVP设备中“预约”或“保留”出来。这样一来，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（或服务品质）。

RSVP协议在预约网络资源时，需要贯穿全网，建立端到点的“数据流”。这里的“数据流”实际就是一个网络路径，除了同一个或多个发送者联系在一起之外，还要同一个特定的QoS级别联系在一起。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，便需向目标接收者（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（就是“预约”的意思）消息，对发送者作出回应。此后，位于中途的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真的有权来做这样的事情。假如拿出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“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消息的定时刷新，RSVP协议便能“永葆青春”；或者说，一直保持“动态”。假如出现了一条更好（比如更快）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，还会将重点放回RSVP协议，并讲解如何用Winsock API调用提供对它的支持。

对于会话设置和RSVP来说，有个问题切不可掉以轻心——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个会话，并为接收请求初始化另一个会话。在本章后面，还会详细论述RSVP会话的初始化要求。

12.1.2 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（选择）。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，我们创建了下面这些必不可少的组件：

- 802.1p：在子网中区分数据包优先次序的一种标准，它在包的访问媒体控制（MAC）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对象映射成正宗WAN QoS组件（网络的位于OSI第2层）的一种机制。
- 子网带宽管理器（SBM）：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（负载）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。

1. 802.1p

这个标准用于确保QoS要求的质量等级，同时避免对存在于网络 HUB和交换机的所有数据包进行同等对待。HUB和交换机均位于OSI参考模型的第2层，所以只有在每个包开头的媒体访问控制头内进行设置，才有意义。

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，连接不通，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。网卡（NIC）、网络驱动程序以及网络交换机都必须具备 802.1p功能。

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。利用这个方法，通过OSI模型第3层设备（比如路由器）的数据包可以分别拥有不同的优先等级设置。为实现IP优先级，我们需要在IP头内设置“服务类型”（TOS）字段，以建立不同的优先等级。换言之，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，并以它为准，对传输的数据进行分别对待。

3. 第2层传信

若数据通过一个广域网（WAN）传输，那么第2层传信是颇为必要的。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五花八门，不一而足。总之，网络通信的环境非常复杂。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（服务质量），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技术利用这些方法为实现自己的 QoS。

4. 子网带宽管理器

子网带宽管理器（Subnet Bandwidth Manager，SBM）负责对一个指定共享媒体网络（如以太网）上的资源进行管理。 SBM也要面向QoS应用，施加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，因为假若端点为某个应用请求 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。那么， SBM是如何来解决这个问题的呢？它的办法是与“许可控制服务”（Admission Control Service，ACS）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（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完全可能是一个特殊的主机，正在运行 Windows 2000 Server网络操作系统。

12.1.3 应用组件

现在，大家已经知道了为提供对 QoS的支持，对网络提出了哪些要求。接下来，我们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服务提供者：负责调用其他QoS组件的一个服务提供者。
- 通信控制模块：该模块负责控制离开计算机的通信数据。这个模块包括常规包分类器、包调度器以及包封装器。
- 资源预约协议（RSVP）。
- 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。

1. GQOS服务提供者

GQOS服务提供者是一种GQOS组件，可调用几乎所有最终的QoS机制。GQOS服务提供者可初始化通信控制功能（如果可以的话），同时为所有GQOS功能实施、维护以及控制RSVP传信。

要想在自己的主机上找到具有QoS能力服务提供者，可用WSAEnumProtocols函数查询提供者目录。用来核实某个提供者是否支持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，便表明该提供者支持QoS。要想了解WSAEnumProtocols的详情，可参考第5章。

2. 通信控制模块

通信控制（Traffic Control，TC）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。正是由于在这里对不同的数据包分配了不同的优先级，所以随着它们流经系统和网络，最终传遍整个网络，所以会直接对QoS的特征产生影响。TC模块通过三个模块来实现：常规包分类器、包调度器以及包封装器。

(1) 常规包分类器

常规包分类器（Generic Packet Classifier，GPC）的职责是对网络组件内的数据包进行分类，并分配它的优先等级。GPC定义优先级的依据是CPU时间或在网上进行传输这样的活动。

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一个步骤。

(2) 包调度器

数据包调度器（Packet Scheduler）控制着数据的传输方式，这是QoS的一项关键功能。数据包调试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。换言之，它需要为一个特定的数据流规定相应的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）的作用是规划数据从数据流到网络的传输。大多数应用程序都是以“爆发”形式来读写数据的；然而，许多QoS应用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，营造出负担比较平衡的一个网络。

3. 通信控制API

通信控制API是本地主机上用来规划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在对 Winsock GQOS的调用过程中，“ 顺便 ”调用的。此时需要由 GQOS服务提供者为那些函数提供服务。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。然而，对这些函数的深入探讨已超出了本书的范围（请参考 Platform SDK，获得进一步的信息；本书配套光盘提供)。至于Winsock GQOS API的情况，本章稍后便会详加论述。

12.1.4 策略组件

“ 策略 ”（Policy）是GQOS的最后一个组件，它负责将资源分配给具有 QoS能力的应用程序。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：

- 许可控制服务（ACS）：一种Windows 2000服务器服务，用于拦截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。
- 本地策略模块（LPM）：面向SBM（子网带宽管理器），以通过ACS配置的策略为准，提供资源访问决策服务。
- 策略元素（PE）：驻留于客户机，提供身份证明信息，以完成预约请求。

1. 许可控制服务

许可控制服务（Admission Control Service，ACS）面向具有QoS功能的应用程序，对网络带宽的使用进行规划。这是通过 RSVP协议来完成的。 ACS会同时拦截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。拦截到一条 RSVP消息后，便会传递给本地策略模块（LPM），由后者进行实际的身份验证。

ACS驻留于安装了Windows 2000操作系统的机器上，可由系统管理员加以控制，后者可分别针对用户、应用程序或者用户组，对他们享受的资源加以限制。

2. 本地策略模块

本地策略模块（Local Policy Module，LPM）与前述的ACS存在着紧密的联系。因为先要由ACS拦截RSVP消息，插入用户信息，再传递给 LPM。之后， LPM会在“ 活动目录 ”（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拦截到的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域的一份子。

3. 策略元素

在这个组件中，实际包含了 LPM请求的策略信息。本书不会讨论这些数据结构的详情，因为它们主要与网络资源的管理有关，而那并非本书的主题。

12.2 QoS和Winsock

在前一节内，我们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，让我们把重点转向Winsock 2。利用这套API，我们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中止。我们要讨论的最后一个问题是“ 提供者 ”特有的一些对象，可用来影响 QoS服务提供者的一些行为，或影响自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者特有的结构，似乎有些“ 杂乱无章 ”。但是，我们之所以这样做，完全是为了让大家首先全面理解那些主要结构如何与Winsock API调用打交道，再来接触由特定提供者决定的一些选项的细节。

12.2.1 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。
- 一个提供者特有的缓冲区，用来指定各个提供者具有的某些 QoS特征（后文详述）。

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 Provider Specific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者的选项将在本章后面部分讲述。

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，让我们看看每个FLOWSPEC结构字段的含义。

(1) TokenRate

TokenRate字段定义了数据的传输速度，单位是“ 字节 /秒 ”。Token是“ 令牌 ”的意思。作为一个应用程序，它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，积累传输所需的“信用点”（令牌），但最多只能积累到由TokenBucketSize的大小，而且由于它们的“峰值传输”受到PeakBandwidth的限制，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。之所以说传输控制得到了保证，是由于数据流不能一次发送任意多的数据。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）模块负责决定对于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）。

(2) 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，可考虑将该字段设为准备传输的视频帧的大小，因为我们要求每个视频帧最好都能一次传送出去。而对于那些要求数据传输速度固定的应用，应考虑将该字段设置成允许一些变化。类似于TokenRate字段，TokenBucketSize采用的单位也是“字节／秒”。

(3) PeakBandwidth

PeakBandwidth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。这是一个至关重要的值，因为可用它防止应用程序积累数量众多的传输令牌，然后一古脑地传送出去，造成网络上的数据“泛滥成灾”。PeakBandwidth的原意便是“高峰带宽”，采用的单位仍然是“字节／秒”。

(4) Latency

Latency字段规定了从一个位传送出去之后，到接收者（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，要由在 ServiceType字段中请求的服务等级决定。Latency指定的是一个时间长度，采用的单位是“毫秒”。

(5) DelayVariation

DelayVariation指定一个数据包允许的最短及最长延迟时间的差距。通常，应用程序依据这个值来决定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样式。DelayVariation的单位也为毫秒。

(6) 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。系统会进行恰当的努力，来维持那个服务等级。然而，却不对数据包是否能正常投递出去，提供任何保障。这正是所谓的“最大努力”通信。
- 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。第一种情况，数据包的丢失接近传输媒介正常的出错率；第二种情况，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。然而，假如实际的数据传输速度超过 TokenSize，那么数据可能延误，也可能丢弃（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，Latency（延迟）时间也必须保证。

除了这四种服务类型之外，另外还有几个标志，可用来提供返回给应用程序的信息。这些信息性标志可通过与任何有效 ServiceType标志的OR（或）运算，结合到一起。表 12-1全面总结了这些信息性的标志。

表12-1 服务类型修改符标志

值	含 义
SERVICETYPE_NETWORK_UNAVAILABLE	指定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。这个标志可自一次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
SERVICE_IMMEDIATE_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_TRAFFIC_CONTROL	该标志可与其他 ServiceType标志通过OR（或）运算合并到一起，完全禁止通信控制
SERVICE_NO_QOS_SIGNALING	这个标志随前述的立即通信控制标志（ SERVICETYPE_IMMEDIATE_TRAFFIC_CONTROL）一道使用，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

(7) 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。

(8) MinimumPolicedSize

MinimumPolicedSize字段定义了某个流内允许的最小数据包长度。 MinimumPolicedSize的值也用字节数为单位。

12.2.2 QoS调用函数

现在，假定我们想让自己的应用程序在网络上发出一个请求，提出特定的带宽要求。有四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“提供者”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，并不一定同时反映着“端到端”通信的带宽。

建好一个数据流之后，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。此时，通过对已分配资源的一次重新协商，会造成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，QoS事件传信以及状态信息的详情还会继续深入下去。

1. WSAConnect

客户机可使用一个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。要求的QoS值通过lpSQOS参数传递过去。目前，成组QoS（Group QoS）尚未得以支持或实现；所以对lpGQOS这个参数而言，应为其传递一个空值（NULL）。

```
int WSAConnect (
    SOCKET s,
    const struct sockaddr FAR *name,
    int namelen,
    LPWSABUF lpCallerData,
    LPWSABUF lpCalleeData,
    LPQOS lpSQOS,
    LPQOS lpGQOS
);
```

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，让服务提供者知道将PATH和RESV消息发送哪里。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。

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
);
```

但令人非常遗憾的是，QoS服务提供者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，那些值在默认情况下，会复制给客户机套接字。

实际上，条件函数根本没什么用处。使用 TCP，我们实际根本不能拒绝一个客户机连接，因为到条件函数调用的时候，连接已在 TCP这一级建立起来了。除此以外，即使已经有一个PATH消息抵达，QoS服务提供者也不会将有效的 QoS参数传递给条件函数。总之，请不要使用WSAAccept条件函数。

注意 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参数不为空（NULL），便必须设置QOS（使用SIO_SET_QOS），否则WSAAccept调用会失败。

3. WSAJoinLeaf

WSAJoinLeaf用于多点通信。在第11章中，我们在一个更深入的层次，探讨了多播（多点传送）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，我们会接收到一个 FD_QOS事件，通知我们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“存在时间”（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

SIO_SET_QOS Ioctl命令对套接字的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，便应完成TTL的设置，这是至关重要的一个注意事项，因为套接上的TTL设置也会对RSVP消息的TTL产生影响，从而直接影响你的资源预约请求最终能传播到几个网络中去。举个例子来说，如果我们希望在一个 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“蔓延”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，会自动采用的默认的 TTL设置：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。

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连接或未连接的套接字上首次请求 QoS，亦可用来在初次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，那么经由提供者所特有的一些信息，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。在第9章中，我们已探讨了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。随SIO_SET_QOS调用WSAIoctl的一个好处在于，我们可指定由具体提供者决定（提供者所特有）的一些对象，以便进一步推敲、落实QoS的行为。在下一节内，我们打算专门讲述提供者所特有的全部对象。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，那么可随SIO_SET_QOS一道，来调用WSAIoctl，同时在提供者特有缓冲区内，指定目标地址对象，从而与一个“端点”联系到一起，最终促使一个 RSVP会话的正常建立。设定QoS参数时，请以lpvInBuffer的形式，传递QoS结构，同时用cbInBuffer来指定传递的字节量大小。

SIO_GET_QOS选项则在FD_QOS事件接收到之后使用。若某个应用程序收到了这种事件通知，就应随SIO_GET_QOS一道，发出对WSAIoctl的一个调用，对产生该事件的原因进行调查。就像我们早先指出的那样，之所以会产生 FD_QOS事件，原因不外乎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指定具体大小。输入参数可为 NULL（空值），或为0（零值）。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，包括由具体提供者决定的对象。 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（空），那么在自WSAIoctl返回后，len字段便会发生更新，换成需要的大小。此外，假如由于缓冲区过小，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操作系统中，才支持对缓冲区大小的查询。而对 Windows 98来说，请务必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。

随WSAIoctl一道，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12-2所示的6个值。调用这个命令时， lpvInBuffer参数会指向一个DWORD（双字），它会设

为三个标志之一。lpvOutBuffer参数也应指向一个DWORD；而且在返回之后，请求的值也会返回。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，但却没有收到任何RESV消息，指出QoS等级的成功分配，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“尽最大努力”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12-2 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（布尔值，下同）
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（双字）
LOCAL_TRAFFIC_CONTROL	返回传输控制（TC）是否已经安装，并可开始使用	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12-2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，表中最后四个选项可返回常数 INFO_NOT_AVAILABLE（信息不可用）。

12.3 QoS中止

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的中止，同时停止为一个套接字处理“通信控制”（TC）。

- 用closesocket函数关闭一个套接字。
- 用shutdown函数关闭一个套接字。
- 用一个空的远程地址（通信对方的地址）调用 WSAConnect。
- 使用SERVICETYPE_NOTRAFFIC或者SERVICE_TYPE_BESTEFFORT服务类型，调用WSAIoctl和SIO_SET_QOS。

除上述列表的第二项事件之外，其他各个事件都是很容易明白的。对于 shutdown函数来说，请务必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，数据流的停止。换言之，假如随 SD_SEND（发送）调用shutdown，那么对于正在接收的数据，QoS仍然是有效的。

由提供者决定的对象

本小节打算讲述那些由具体提供者所决定的对象。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信息返回至我们的应用程序；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的行为作进一步的落实。

在由提供者决定的对象中，每个都包含了一个 QOS_OBJECT_HDR（对象头）结构。该

结构是这种对象的第一个成员。这个结构指定了提供者特有对象的类型。该结构是至关重要的，因为这些提供者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。使用QOS_OBJECT_HDR，我们的应用程序可以标识出每个对象，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指出预设的提供者特有对象的类型是什么。而 ObjectLength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，以及提供者特有对象本身。表 12-3对可选的对象类型标志进行了总结。

表12-3 对象类型

提供者对象	对象结构
QOS_OBJECT_PRIORITY	QOS_PRIORITY
QOS_OBJECT_SD_MODE	QOS_SD_MODE
QOS_OBJECT_TRAFFIC_CLASS	QOS_TRAFFIC_CLASS
QOS_OBJECT_DESTADDR	QOS_DESTADDR
QOS_OBJECT_SHAPER_QUEUE_DROP_MODE	QOS_SHAPER_QUEUE_LIMIT_DROP_MODE
QOS_OBJECT_SHAPER_QUEUE_LIMIT	QOS_SHAPER_QUEUE_LIMIT
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。没有更多的对象

1. QoS优先级

QoS优先级定义了数据流的绝对优先级大小。优先级本身的取值范围在 0到7之间，从最低级到最高级，优先程度越来越高！ QOS_PRIORITY结构定义如下：

```
typedef struct _QOS_PRIORITY
{
    QOS_OBJECT_HDR  ObjectHdr;
    UCHAR           SendPriority
    UCHAR           SendFlags;
    UCHAR           ReceivePriority;
    UCHAR           Unused;
} QOS_PRIORITY, *LPQOS_PRIORITY;
```

这些优先级设定决定了本地对应数据流传输的优先等级大小（在负责发送数据的主机计算机内部），它相对于来自其他流的数据传输。对一个数据流来说，默认的优先等级是 3。这一优先级需要与FLOWSPEC的ServiceType（服务类型）参数联合使用，最终决定应将何种优先级应用于“ 包调度器 ”（Packet Scheduler）内部的数据流。 SendFlags和ReceivePriority目前尚未使用，但将来却不一定。

2. QoS封装丢弃模式

QoS对象定义了通信控制（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 ”。这个对象定义了作为本地系统，该如何应付这种情况。QOS_SD_MODE结构定义如下：

```
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12-4中选择。

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，有时便要采取这种看似“ 反常 ”的行动。大多数情况下， FLOWSPEC结构在设置的时候，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断。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）吗？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，比如影音文件的传输，那么一种更好的做法是丢弃当前帧，立即转移到下一个帧。

表12-4 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，可能会降级低于“ 尽最大努力 ”优先级
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无情地丢弃

3. QoS通信类别

QOS_TRAFFIC_CLASS结构可携带由一个第 2层网络设备提供给主机的 802.1p通信类参数。该结构的定义如下：

```
typedef struct _QOS_TRAFFIC_CLASS
{
    QOS_OBJECT_HDR   ObjectHdr;
    ULONG            TrafficClass;
} QOS_TRAFFIC_CLASS, *LPQOS_TRAFFIC_CLASS;
```

主机会用结构中指定的 TrafficClass值，填写对应的、已传送的数据包的 MAC头。尽管该结构已包容在Qos.h文件中，但却不允许应用程序对自己的优先级进行设置。

4. 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，同时不必使用一个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。我们可用 SIO_SET_QOS这个I/O控制命令对目标地址进行设置。结构定义如下：

```
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，它为指定的协议定义了端点地址。SocketAddressLength则指定了SOCKADDR这个结构的长度。

5. QoS封装器队列限制丢弃模式

这个结构定义了在抵达一个流的封装器队列长度限制后，用于代替数据包默认丢弃方案的任何一种自定义丢弃方案。请大家记住，对于“传输控制”的传输封装器（Traffic Shaper）模块来说，我们对其进行配置，在当前队列中的数据与FLOWSPEC结构的定义不相符时，丢弃任何多余的数据。结构定义如下：

```
typedef struct _QOS_SHAPER_QUEUE_LIMIT_DROP_MODE
{
    QOS_OBJECT_HDR   ObjectHdr;
    ULONG            DropMode;
}  QOS_SHAPER_QUEUE_LIMIT_DROP_MODE,
  *LPQOS_SHAPER_QUEUE_LIMIT_DROP_MODE;
```

对DropMode来说，可选的两个值如表12-5所示。

表12-5 封装器队列限制丢弃模式

标 志	含 义
QOS_SHAPER_DROP_FROM_HEAD	从队列头中丢弃数据包（此乃默认行为）
QOS_SHAPER_DROP_INCOMING	一旦抵达队列限制，便丢弃任何进入的数据包

6. QoS封装器队列限制

我们可用自定义的“封装器队列限制结构”来代替封装器队列的默认流限制。该结构的定义如下：

```
typedef struct _QOS_SHAPER_QUEUE_LIMIT
{
    QOS_OBJECT_HDR   ObjectHdr;
    ULONG            QueueSizeLimit;
} QOS_SHAPER_QUEUE_LIMIT, *LPQOS_SHAPER_QUEUE_LIMIT;
```

其中，QueueSizeLimit指定封装器队列的长度，以字节为单位。或设定一个较大的封装器队列长度，可防止数据的“无辜”丢弃，允许那些数据在不会用光缓冲区空间的前提下，正常地“排队”。

7. RSVP状态信息

RSVP状态信息对象用于返回RSVP特有的一些错误，以及一些状态信息。结构定义如下：

```
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。在表12-6中，我们总结了可能用到的代

码。另两个字段——ExtendedStatus1和ExtendedStatus2则为与具体提供者相关的信息保留。

表12-6 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_NO_QOS_RECEIVERS	没有接收者
WSA_NO_QOS_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
WSA_QOS_BAD_STYLE	未知或冲突的样式
WSA_QOS_BAD_OBJECT	RSVP_FILTERSPEC结构的某个部分或与具体提供者相关的缓冲区存在问题
WSA_QOS_TRAFFIC_CTRL_ERROR	FLOWSPEC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，对于具有QoS能力的，以UDP为基础的接收者来说，会生成包含了一条 WSA_QOS_SENDERS消息的一个FD_QOS事件，指出有“人”请求QoS服务，希望将数据传给接收者。

8. 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者对应的缓冲区，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。该对象定义如下：

```
typedef struct _RSVP_RESERVE_INFO
{
    QOS_OBJECT_HDR    ObjectHdr;
    ULONG             Style;
    ULONG             ConfirmRequest;
    ULONG             NumPolicyElements;
    LPRSVP_POLICY     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过滤器类型。在表 12-7中，我们总结了目前可选的一些过滤器类型，以及不同类的接收者使用的默认过滤器类型。注意每类过滤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字段是联系起来的。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，我们还会对RSVP_POLICY进行定义。接下来，且让我们看看各种不同的过滤器样式，以及各自的特征。

9. RSVP_DEFAULT_STYLE

这个标志指示QoS服务提供者使用默认样式。表12-7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过滤器，而通配符用于多播接收者。调用 WSAConnect的UDP接收者

亦可使用固定过滤器。

表12-7 默认过滤器样式

过滤器样式	默认用户
固定过滤器	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

10. RSVP_FIXED_FILTER_STYLE

通常，这种样式会在接收者以及单独一个数据源之间，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，我们也完全可以设置采用了多个固定过滤器的样式，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，假如我们的接收者想从三个发送者那里接收数据，并需要为每一个都保证（预约）20Kbps的带宽。此时，我们就需要考虑采用多固定过滤器样式。在这个例子中，NumFlowDesc应设为3，而FlowDescList应包含三个地址，每个FLOWSPEC一个。另外，我们亦可为每个发送者分配不同的 QoS等级；它们并一定要完全相同。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过滤器。在图 12-1中，我们展示了FLOWDESCRIIPTOR（流描述符）和RSVP_FILTERSPEC（RSVP过滤器规格）这两个结构间的关系。

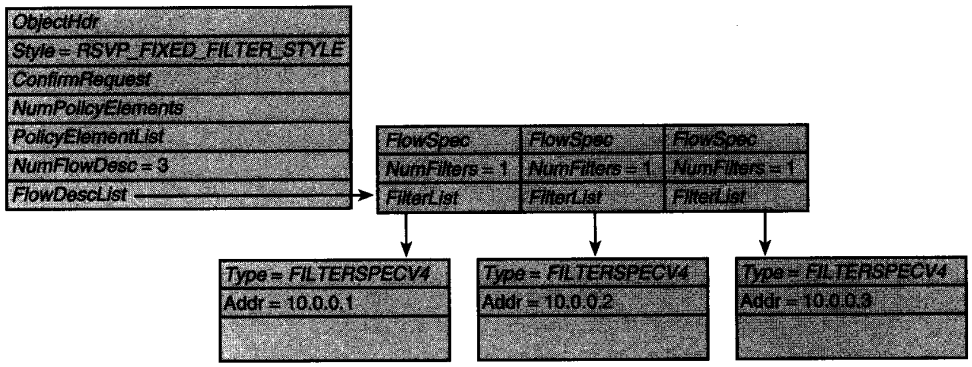

图12-1 多固定过滤器样式

11. 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（ WILDCARD）。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，这是它们的默认过滤器样式 —— 因为发送者的地址当时是未知的。

12. RSVP_SHARED_EXPLICIT_STYLE

在某种程度上，该样式类似于多固定过滤器样式，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12-2对这一样式进行了阐释。

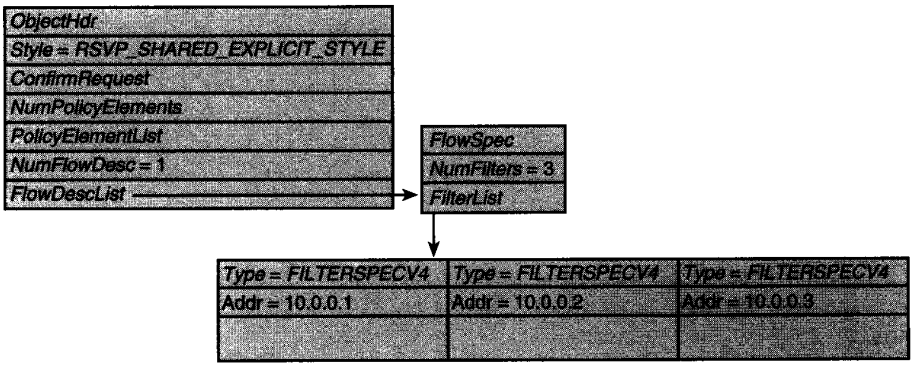

图12-2 共享显式样式

在我们讨论RSVP样式时，已向大家讲解了最后两个字段：NumFlowDesc（数据流的目标数量）和FlowDescList（数据流目标地址列表）。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（流描述符）结构在FlowDescList字段中的数量。该结构定义如下：

```
typedef struct _FLOWDESCRIPTOR
{
    FLOWSPEC              FlowSpec;
    ULONG                 NumFilters;
    LPRSVP_FILTERSPEC     FilterList;
} FLOWDESCRIPTOR, *LPFLOWDESCRIPTOR;
```

该对象用于定义由 FlowSpec给定的每个 FLOWSPEC的过滤器类型。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，第一个字段Type是对下述值的一个简单列举：

```
typedef enum {
        FILTERSPECV4 = 1,
        FILTERSPECV6,
        FILTERSPECV6_FLOW,
        FILTERSPECV4_GPI,
        FILTERSPECV6_GPI,
        FILTERSPEC_END
} FilterType;
```

列举指定了联盟中存在的对象。每个过滤器的规格（FILTERSPEC）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;
```

13. 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;
```

我们感兴趣的第一个字段是GeneralParams，它是类型为AD_GENERAL_PARAMS（常规广告参数）的一个结构。该结构用来定义一些常规的特征参数，正如其名。该对象的定义如下：

```
typedef struct _AD_GENERAL_PARAMS
{
    ULONG       IntServAwareHopCount;
    ULONG       PathBandwidthEstimate;
    ULONG       MinimumLatency;
    ULONG       PathMTU;
    ULONG       Flags;
} AD_GENERAL_PARAMS, *LPAD_GENERAL_PARAMS;
```

其中，IntServAwareHopCount指定的是符合“集成服务”（IntServ）要求的跳数(hop)。PathBandwidthEstimate指定的是从发送者到接收者，至少能使用多大的带宽。

MinimumLatency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计，以毫秒为单位。PathMTU指定的是“最大传输单元”（MTU），亦即一个数据包最多能在多大。如超过这一设定，数据包在传输过程中便可能发生分解。Flags字段目前尚未使用。

14. RSVP策略信息

我们要讨论的最后一个提供者对象是RSVP策略信息。该对象的含义非常暧昧——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字段指定了在PolicyElement数组中，总共存在多少个RSVP_POLICY结构。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，对应于策略元素，与我们目前的需要关系不大。

12.4 QoS编程

QoS的核心在于RSVP会话的建立。除非发送出RSVP PATH以及RESV消息，而且得到了处理，同时正确预约了带宽，否则这个会话是不会草率地建立起来的。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，首先必须知道三个参数：

- 发送FLOWSPEC成员。
- 源IP地址及端口。
- 目标IP地址、端口及协议。

只要调用了一个具有QoS功能的函数，那么FLOWSPEC成员便是已知的了。那些函数包括WSAConnect、WSAJoinLeaf、WSAIoctl（设置SIO_SET_QOS选项）等等。源IP地址和端口号却暂时无法知道，直至完成了套接字的本地绑定。这种绑定既可是暗示性进行的（隐式），比如通过连接来绑定；亦可是明确进行的。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者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RSVP RESV消息，事先必须知道三件事情：

- 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成员可通过任何具有QoS能力的Winsock函数获得。每个发送者的地址取

决于过滤器的“样式”，后者可用由具体提供者决定的结构 RSVP_RESERVE_INFO来人工设定（前面已经具体讲过了）。否则的话，亦可从一条 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，从而取得发送者的地址。这样的一个典型例子便是多播通信中采用的通配符过滤器。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用多费什么脑筋便能取得。然而，多播接收者却不然。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。

在本节内，我们首先要来看看各种不同的套接字类型，以及它们如何与 QoS服务提供者及RSVP消息打交道。然后，我们会解释 QoS服务提供者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，再来写一个提供 QoS功能的应用程序，便能起到事半功倍效果。要想写出这样的应用程序，其实只需知道如何取得 QoS的质量保证，知道这些保证何时能够兑现，并且知道它们何时和怎样发生变化。

12.4.1 RSVP和套接字类型

到目前为止，大家已对 PATH及RESV RSVP消息如何生成有了一定的概念。在后续的小节中，我们打算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服务提供者沟通，以生成PATH和RESV消息。

1. 单播UDP

由于我们可选择使用“已连接”和“未连接”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”绑定调用中取得，亦可通过由WSAConnect完成的一个“隐式”绑定中取得。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中明确指定的，要么是通过 QOS_DESTADDR这个由具体提供者决定的结构来传递的（指定 SIO_SET_QOS选项）。要注意的是，假如用 SIO_SET_QOS来设置QoS，那么套接字必须在事前绑定好。

对UDP接收者来说，可调用 WSAConnect，将接收应用程序限制成单独一个发送者。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（目标地址）结构。否则，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，同时采用通配过滤器样式。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，都只有在自己希望应用程序只从一个采用固定过滤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过滤器的RESV消息。一旦连接调用完毕，前一次 RESV会话便会关闭；并采用固定过滤器样式，建立一次新的会话。除此以外，若在 WSAConnect之后调用SIO_SET_QOS，那么采用固定过滤器的RESV消息不会中断 RSVP会话，并生成一个通配过滤器样式。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。

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。第二种可能性则是客户机与之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，这会造成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，否则根本不会生成任何PATH消息。

假如发送者不巧是服务器，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，这个函数并未提供一种方式，可供我们在接受的套接字上设置 QoS。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。然而，我们目前遇到的并不是这种情况。 QoS服务提供者传递的只是一些垃圾 —— 无论Windows 98，还是Windows 2000，都会产生这样的行为。例外在于，假如在 Windows 98中，lpSQOS参数不为“空”（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。在接受之后， QoS亦可在客户机套接字上进行设置。

现在，让我们来看看负责接收的TCP应用。第一种情况是调用WSAConnect，同时设置一个接收FLOWSPEC。在这种情况下， QoS服务提供者会创建一个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（结果便造成了一条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。否则，亦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，那么QoS服务提供者都会生成一条 RESV消息。

3. 多播

多播发送者的行为和 UDP发送者的行为差不多，唯一的例外在于，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，并将该对象包括在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（通信对方的地址），所以QoS提供者在生成RESV消息的时候，会采用通配过滤器样式。 QoS服务提供者并不禁止一个套接字同时加入多个多播组。在这种情况下，服务提供者会将 RESV消息发给所有组，只有它们有一条相符的PATH消息。为每个WSAJoinLeaf提供的QoS参数会在每条RESV消息中使用，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它的前提下，才会造成相应的QoS参数应用于那些数据。换言之，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，不会享受到 QoS提供的任何服务。

12.4.2 QoS通知

迄今为止，大家已学习了如何为TCP、UDP以及多播UDP套接字调用QoS。大家已经知道，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，也无法担保网络资源已正确分配。正是考虑到这方面的原因，我们增加了一个新的异步事件： FD_QOS，它需要投递给套接字。典型情况下，在发生下述事件时，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，还是被拒绝。
- 网络提供的QoS发生明显改变（与以前协商的值相反）。
- 指出一个QoS通信对方是否已作好准备，可开始收发一个特定流的数据。

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，我们可使用 WSAEventSelect或者WSAAsyncSelect，同时在事件标志的按位“ 或 ”运算中，将 FD_QOS这个标志包括进来。然而，只有已发出了对某个 QoS调用函数的一个调用，应用程序才有资格接收 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，我们可设置一个 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。这样做唯一的缺点在于，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标志通过“ 或 ”运算合并到一起。

如果需要准确地了解如何调用两个异步选择函数，请参考第 8章。在该章中，我们对其进行了更为深入、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）。这个函数也在第8章进行了详述。然而，我们在此仍然要稍微讨论一下，因为对 QoS应用来说，它非常简单、短小和重要。我们需要在调用中传递一个套接字句柄、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。该结构定义如下：

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

其中，lNetworkEvents字段可设为已经触发的所有事件标志的一个按位“ 或 ”运算的结果。要想知道是否发生了一个特定的事件，只需将这个字段的值与那个事件的标志进行一次简单的“ 与 ”运算 —— 使用&运算符。如结果不为零，表明那个事件已经触发。 iErrorCode数组指出遇到了什么错误，或者在 QoS的情况下，指出状态信息。假如触发了一个事件，与那个事件对应的一个标志便会用于索引这个数组。假如数组索引为 0，表明没有发生错误；否则，这

个值就是包含了错误代码的那个数组元素的索引位置。举个例子来说，假定触发了 FD_QOS事件，那么可用FD_QOS_BIT标志对 iErrorCode数组进行索引，检查是否存在错误，或取得相应的状态信息。其他所有Winsock异步事件（FD_READ_BIT和FD_WRITE_BIT等等）都定义了类似的索引标志。

2. RSVP通知

我们早先已经提到，有两个办法均可接收到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，希望随 WSAAsyncSelect或WSAEventSelect接收FD_QOS通知，而且实际接收到了一个 FD_QOS事件通知，那么我们必须执行对WSAIoctl的一个调用，同时设置 SIO_GET_QOS这个I/O控制选项，以便知道具体是谁触发了事件。实际上并不一定需要为 FD_QOS事件注册，可以通过重叠式 I/O，简单地调用WSAIoctl，同时设置SIO_GET_QOS选项。这同时也要求我们指定一个完成例程。一旦 QoS服务提供者检测到QoS发生了一个改变，便会调用这个例程。发生回调的时候，在输出缓冲内，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，我们的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。如果我们选择的是为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，都只包含了针对单独一个方向的 QoS信息。换言之，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，我们应循环调用 WSAIoctl和SIO_GET_QOS，直到返回SOCKET_ERROR，同时WSAGetLastError返回一个WSAEWOULDBLOCK值。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者有关的对象。事实上，经常都会返回一个 RSVP_STATUS_INFO结构，只要缓冲区足够大！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，那么无论如何都必须执行一个 SIO_GET_QOS操作，以便重新允许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，触发它的第一个、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，假定我们将一个套接字设置成提供“尽最大努力”的服务。那么，作为 QoS的服务提供者，它会向我们的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，假如我们设置一个“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度对应的 QoS参数可能会与我们请求的值稍有区别。在我们的应用程序中，一旦收到一个QoS通知，便应立即将返回的 FLOWSPEC同我们最初请求的进行对比，确定应用程序能够继续运行下去。还要记住的是，在提供 QoS能力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。这也会促使一个 QoS事件通知的生成，可知会与当前RSVP会话联系在一起的通信方（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了另外一些事情，比如由发送者或接收者发出的通知等等。在前文的表 12-6中，我们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。第一个办法是作为 RSVP_STATUS_INFO对象的一部分。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者特有缓冲区的一部分而返回。第二个办法，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12-6定义的代码可在iErrorCode数组中找到，该数组由 FD_QOS_BIT进行索引。表中总结的头五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承诺提供我们要求的 QoS保证。

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，并接收到了第一条 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了事，从而同时生成了 FD_CLOSE以及WSA_QOS_NO_RECEIVERS事件。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完事。

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。

4. WSA_QOS_SENDERS和WSA_QOS_NO_SENDERS

发送者通知和接收者事件差不多，唯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，假若收到第一条PATH消息，便会生成一个WSA_QOS_SENDERS；而PATH撤消消息会造成一条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。

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服务提供者需要向应用程序发出一个通知。该对象是一个需要由提供者决定的选项，可随QoS结构传递给SIO_SET_QOS这个I/O控制命令。

12.4.3 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 ”（Template）。作为应用

程序，可按名字对其进行查询。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。WSAGetQOSByName定义如下：

```
BOOL WSAGetQOSByName(
    SOCKET s,
    LPWSABUF lpQOSName,
    LPQOS lpQOS
);
```

假如不知道已安装的那些模板的名字，首先可用该函数“列举”出所有模板名。但要想真正成功，必须用 lpQOSName提供一个足够大的缓冲区，并将它的第一个字符设为空字符，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;
WSAGetQOSByName(s, &wbuf, NULL);
```

返回之后，在字符缓冲内，会填入一个字串数组，各字串之间用一个空字符（ NULL）加以分隔，而且整个列表会用另一个空字符加以中止。换言之，最后一个字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字，并可从中查询一个特定的。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。本书配套光盘上的示例 Qostemplate.c向大家阐述了如何列举已安装的 QoS模板。

除此以外，我们还可以安装自己的 QoS模板，使其他应用程序能根据名字对其进行查询。此时要用到两个函数：WSCInstallQOSTemplate以及WSCRemoveQOSTemplate。其中，第一个函数用于QoS模板的安装，而第二个用于删除。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 WSCInstallQOS Template，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这样的工具程序来生成。要想删除模板，只需调用 WSCRemoveQOSTemplate并指定那个模板的名字，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。

12.5 示例

在本节内，我们打算通过三个实际的例子，来展示QoS的使用。第一个例子使用的是TCP，它在三个例子中显得最“直接了当”，因为它是“面向连接”的。第二个例子使用UDP，而且没有使用任何连接调用。最后一个例子使用的则是多播UDP。在所有这三个例子中，我们都会使用WSAEventSelect，因为它比WSAAsyncSelect稍微简单一些。对于TCP单播的例子，我们在此给出了完整的源码。但对UDP以及多播示例来说，却只给出了一些重要的代码片断，这是由于它们的大多数概念都是相同的，无论使用的套接字是何种类型。要想观看完整源码，请参考本书的配套光盘。这三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.c以及Provider.c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者目录中查找具有指定属性（如QoS）的一种协议。

12.5.1 单播TCP

TCP QoS示例的完整源码是一份很长的清单，请见后文。本例的源码亦可在光盘上的第12章目录下找到，文件名为Qostcp.c。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，我们已在第8章详细讲述过了。唯一的例外是我们在产生FD_QOS事件时做的事情。主函数只是进行解析出参数，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，让我们首先来看看客户机连接的情况。

在这里的所有例子中，都可用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（在表12-8中，我们列出了Qostcp.c可以选用的命令行参数）。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QOS FLOWSPEC。要注意的是，在调用SIO_SET_QOS之前，实际并非真正需要进行这种绑定。这是由于除非发出一个连接调用，否则对方的地址是不知道的。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，那么示范代码会将QoS结构传递进入WSAConnect调用。这个调用会建立一个RSVP会话，并将客户机同指定的服务器连接起来。否则的话，用户需要指出这个例子应该等候对方设置QoS，而且没有QoS结构传递给WSAConnect。相反，代码会分析发送QoS结构，拿SERVICE_NO_QOS_SIGNALING标志同FLOWSPEC结构中的ServiceType字段进行或（OR）运算，并随SIO_SET_QOS命令一道调用WSAIoctl。这样便可告诉QoS服务提供者不要调用通信控制，而是依然等候RSVP消息。

设好QoS后，会注册客户机希望收到通知的事件，包括FD_QOS。注意为使应用程序正常接收到FD_QOS，QoS事先必须在套接字上设置好，要求接收FD_QOS。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。

Qostcp.c的大多数部分都在与其他事件打交道，比如FD_READ，FD_WRITE以及。

FD_CLOSE等等，其过程和第8章讲述过的WSAEventSelect示范代码差不多。唯一要注意的是FD_WRITE事件中的一个问题。我们提供的一个命令行选项是等候接收到一条 RSVP PATH消息，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，应等候确认发出数据的信号。

就客户机的情况来说，我们在此打算接收WSA_QOS_RECEIVERS消息，以初始化对一条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，我们可调用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。第一种方式，可在WSANETWORKEVENTS（网络事件）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。第二种方式，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（状态代码）字段中包含了 WSA_QOS_RECEIVERS标志。如决定等待发送标志，那么我们需要检查来自 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，相应的代码是非常直观的。

而换到服务器那一边，我们的这个例子便显得稍微有一些复杂，但也仅仅牵涉到它需要管理零个或多个客户机连接。对负责监听的套接字来说，它和客户机套接字一样，都在一个名为sc的数组中进行控制。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，数组中唯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起来，还会出现更多的套接字，以及与它们对应的事件。假如由某个事件造成 WSAWaitForMultipleEvents这个等待循环中止，而且在数组元素 0中指出了事件句柄是什么，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，我们的程序会调用 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，便会随一个条件函数，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操作系统中， QoS参数不为空值（NULL），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“接受”了客户机套接字，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说，唯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。

程序清单12-1 单播TCP示例（Qostcp.c）

```
// Module: Qostcp.c
//
#include <winsock2.h>
#include <windows.h>
#include <qos.h>
#include <qossp.h>

#include "provider.h"
#include "printqos.h"

#include <stdio.h>
#include <stdlib.h>

#define QOS_BUFFER_SZ       16000 // Default buffer size for
                                  // SIO_GET_QOS
#define DATA_BUFFER_SZ       2048 // Send/Recv buffer size

#define SET_QOS_NONE            0   // No QOS
#define SET_QOS_BEFORE          1   // Set QOS on listening socket
#define SET_QOS_DURING          2   // Set QOS in conditional accept
#define SET_QOS_AFTER           3   // Set QOS after accept
#define SET_QOS_EVENT           4   // Wait for FD_QOS and then set

#define MAX_CONN                10

int  iSetQos,           // When to set QOS?
     nConns;
BOOL bServer,           // Client or server?
     bWaitToSend,       // Wait to send data until RESV
     bConfirmResv;
char szServerAddr[64]; // Server's address
QOS  clientQos,         // QOS client structure
     serverQos;         // QOS server structure
RSVP_RESERVE_INFO  qosreserve;

//
// Set up some common FLOWSPEC structures
//
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};

const FLOWSPEC flowspec_guaranteed = {17000,
                                      1260,
                                      34000,
                                      QOS_NOT_SPECIFIED,
                                      QOS_NOT_SPECIFIED,
                                      SERVICETYPE_GUARANTEED,
                                      340,
                                      340};

//
// Function: SetReserveInfo
//
// Description:
//    For receivers, if a confirmation is requested this must be
//    done with an RSVP_RESERVE_INFO structure
//
void SetQosReserveInfo(QOS *lpqos)
{
    qosreserve.ObjectHdr.ObjectType = RSVP_OBJECT_RESERVE_INFO;
    qosreserve.ObjectHdr.ObjectLength = sizeof(RSVP_RESERVE_INFO);
    qosreserve.Style = RSVP_DEFAULT_STYLE;
    qosreserve.ConfirmRequest = bConfirmResv;
    qosreserve.NumPolicyElement  = 0;
    qosreserve.PolicyElementList = NULL;
    qosreserve.FlowDescList = NULL;

    lpqos->ProviderSpecific.buf = (char *)&qosreserve;
    lpqos->ProviderSpecific.len = sizeof(qosreserve);

    return;
}
//
// Function: InitQos
//
// Description:
//    Set up the client and server QOS structures. This is
//    broken out into a separate function so that you can change
//    the requested QOS parameters to see how that affects
//    the application.
//
void InitQos()
{
    clientQos.SendingFlowspec = flowspec_g711;
    clientQos.ReceivingFlowspec =  flowspec_notraffic;
    clientQos.ProviderSpecific.buf = NULL;
    clientQos.ProviderSpecific.len = 0;
```

```
    serverQos.SendingFlowspec = flowspec_notraffic;
    serverQos.ReceivingFlowspec = flowspec_g711;
    serverQos.ProviderSpecific.buf = NULL;
    serverQos.ProviderSpecific.len = 0;
    if (bConfirmResv)
        SetQosReserveInfo(&serverQos);
}

//
// Function: usage
//
// Description:
//    Print out usage information
//
void usage(char *progname)
{
    printf("usage: %s -q:x -s -c:IP\n", progname);
    printf("      -q:[b,d,a,e] When to request QOS\n");
    printf("          b        Set QOS before bind or connect\n");
    printf("          d        Set QOS during accept cond func\n");
    printf("          a        Set QOS after session setup\n");
    printf("          e        Set QOS only upon receipt of FD_QOS\n");
    printf("      -s           Act as server\n");
    printf("      -c:Server-IP Act as client\n");
    printf("      -w           Wait to send until RESV has arrived\n");
    printf("      -r           Confirm reservation request\n");
    ExitProcess(-1);
}

//
// Function: ValidateArgs
//
// Description:
//    Parse command line arguments and set global variables to
//    indicate how the application should act
//
void ValidateArgs(int argc, char **argv)
{
    int      i;

    // Initialize globals to a default value
    //
    iSetQos = SET_QOS_NONE;
    bServer = TRUE;
    bWaitToSend = FALSE;
    bConfirmResv = FALSE;

    for(i = 1; i < argc; i++)
    {
        if ((argv[i][0] == '-') || (argv[i][0] == '/'))
        {
            switch (tolower(argv[i][1]))
            {
                case 'q':        // When to set QOS
                    if (tolower(argv[i][3]) == 'b')
                        iSetQos = SET_QOS_BEFORE;
```

```
                else if (tolower(argv[i][3]) == 'd')
                    iSetQos = SET_QOS_DURING;
                else if (tolower(argv[i][3]) == 'a')
                    iSetQos = SET_QOS_AFTER;
                else if (tolower(argv[i][3]) == 'e')
                    iSetQos = SET_QOS_EVENT;
                else
                    usage(argv[0]);
                break;
            case 's':        // Server
                printf("Server flag set!\n");
                bServer = TRUE;
                break;
            case 'c':        // Client
                printf("Client flag set!\n");
                bServer = FALSE;
                if (strlen(argv[i]) > 3)
                    strcpy(szServerAddr, &argv[i][3]);
                else
                    usage(argv[0]);
                break;
            case 'w':       // Wait to send data until
                            // RESV has arrived
                bWaitToSend = TRUE;
                break;
            case 'r':
                bConfirmResv = TRUE;
                break;
            default:
                usage(argv[0]);
                break;
        }
    }
  }
  return;
}

//
// Function: AbleToSend
//
// Description:
//    Checks to see whether data can be sent on the socket before
//    any RESV messages have arrived. This function checks to see whether
//    the best-effort level currently available on the network is
//    sufficient for the QOS levels that were set on the socket.
//
BOOL AbleToSend(SOCKET s)
{
    int     ret;
    DWORD   dwCode = ALLOWED_TO_SEND_DATA,
            dwValue,
            dwBytes;

    ret = WSAIoctl(s, SIO_CHK_QOS, &dwCode, sizeof(dwCode),
                &dwValue, sizeof(dwValue), &dwBytes, NULL, NULL);
    if (ret == SOCKET_ERROR)
```

```
    {
        printf("WSAIoctl() failed: %d\n", WSAGetLastError());
        return FALSE;
    }
    return (BOOL)dwValue;
}

//
// Function: ChkForQosStatus
//
// Description:
//    Check for the presence of an RSVP_STATUS_INFO object and
//    determine whether the supplied flags are present in that
//    object
//
DWORD ChkForQosStatus(QOS *lpqos, DWORD dwFlags)
{
    QOS_OBJECT_HDR   *objhdr = NULL;
    RSVP_STATUS_INFO *status = NULL;
    char             *bufptr = NULL;
    BOOL              bDone = FALSE;
    DWORD             objcount = 0;

    if (lpqos->ProviderSpecific.len == 0)
        return 0;

    bufptr = lpqos->ProviderSpecific.buf;
    objhdr = (QOS_OBJECT_HDR *)bufptr;


    while (!bDone)
    {
        if (objhdr->ObjectType == RSVP_OBJECT_STATUS_INFO)
        {
            status = (RSVP_STATUS_INFO *)objhdr;
            if (status->StatusCode & dwFlags)
                return 1;
        }
        else if (objhdr->ObjectType == QOS_OBJECT_END_OF_LIST)
            bDone = TRUE;

        bufptr += objhdr->ObjectLength;
        objcount += objhdr->ObjectLength;
        objhdr = (QOS_OBJECT_HDR *)bufptr;

        if (objcount >= lpqos->ProviderSpecific.len)
            bDone = TRUE;
    }
    return 0;
}

//
// Function: HandleClientEvents
//
// Description:
```

```
//    This function is called by the Server function to handle
//    events that occurred on client SOCKET handles. The socket
//    array is passed in along with the event array and the index
//    of the client who received the signal. Within the function,
//    the event is decoded and the appropriate action occurs.
//
void HandleClientEvents(SOCKET socks[], HANDLE events[], int index)
{
    WSANETWORKEVENTS  ne;
    char              databuf[4096];
    WSABUF            wbuf;
    DWORD             dwBytesRecv,
                      dwFlags;
    int               ret,
                      i;

    // Enumerate the network events that occurred
    //
    ret = WSAEnumNetworkEvents(socks[index], events[index], &ne);
    if (ret == SOCKET_ERROR)
    {
        printf("WSAEnumNetworkEvents() failed: %d\n",
            WSAGetLastError());
        return;
    }
    // Data to be read
    //
    if ((ne.lNetworkEvents & FD_READ) == FD_READ)
    {
        wbuf.buf = databuf;
        wbuf.len = 4096;

        if (ne.iErrorCode[FD_READ_BIT])
            printf("FD_READ error: %d\n",
                ne.iErrorCode[FD_READ_BIT]);
        else
            printf("FD_READ\n");

        dwFlags = 0;
        ret = WSARecv(socks[index], &wbuf, 1, &dwBytesRecv,
            &dwFlags, NULL, NULL);
        if (ret == SOCKET_ERROR)
        {
            printf("WSARecv() failed: %d\n", WSAGetLastError());
            return;
        }
        wbuf.len = dwBytesRecv;

        printf("Read: %d bytes\n", dwBytesRecv);
    }
    // Able to write data; nothing to do here
    //
    if ((ne.lNetworkEvents & FD_WRITE) == FD_WRITE)
    {
        if (ne.iErrorCode[FD_WRITE_BIT])
```

```
            printf("FD_WRITE error: %d\n",
                ne.iErrorCode[FD_WRITE_BIT]);
        else
            printf("FD_WRITE\n");
    }
    // The client closed the connection. Close the socket on our
    // end and clean up the data structures.
    //
    if ((ne.lNetworkEvents & FD_CLOSE) == FD_CLOSE)
    {
        if (ne.iErrorCode[FD_CLOSE_BIT])
            printf("FD_CLOSE error: %d\n",
                ne.iErrorCode[FD_CLOSE_BIT]);
        else
            printf("FD_CLOSE ...\n");
        closesocket(socks[index]);
        WSACloseEvent(events[index]);

        socks[index] = INVALID_SOCKET;
        //
        // Remote the client socket entry from the array and
        // compact the remaining clients to the beginning of the
        // array
        //
        for(i = index; i < MAX_CONN - 1; i++)
            socks[i] = socks[i + 1];
        nConns--;
    }
    // Received an FD_QOS event. This could mean several things.
    //
    if ((ne.lNetworkEvents & FD_QOS) == FD_QOS)
    {
        char         buf[QOS_BUFFER_SZ];
        QOS         *lpqos = NULL;
        DWORD        dwBytes;

        if (ne.iErrorCode[FD_QOS_BIT])
            printf("FD_QOS error: %d\n",
                ne.iErrorCode[FD_QOS_BIT]);
        else
            printf("FD_QOS\n");

        lpqos = (QOS *)buf;
        lpqos->ProviderSpecific.buf = &buf[sizeof(QOS)];
        lpqos->ProviderSpecific.len = sizeof(buf) - sizeof(QOS);

        ret = WSAIoctl(socks[index], SIO_GET_QOS, NULL, 0,
                buf, QOS_BUFFER_SZ, &dwBytes, NULL, NULL);
        if (ret == SOCKET_ERROR)
        {
            printf("WSAIoctl(SIO_GET_QOS) failed: %d\n",
                WSAGetLastError());
            return;
        }
        PrintQos(lpqos);
        //
        // See whether we're set for receiving FD_QOS events only.
```

```
        // If so, we need to invoke QOS on the connection
        // now; otherwise, client will never receive a RESV message.
        //
        if (iSetQos == SET_QOS_EVENT)
        {
            lpqos->ReceivingFlowspec.ServiceType =
                serverQos.ReceivingFlowspec.ServiceType;

            ret = WSAIoctl(socks[index], SIO_SET_QOS, lpqos,
                dwBytes, NULL, 0, &dwBytes, NULL, NULL);
            if (ret == SOCKET_ERROR)
            {
                printf("WSAIoctl(SIO_SET_QOS) failed: %d\n",
                    WSAGetLastError());
                return;
            }
            //
            // Change iSetQos so we don't set QOS again if we
            // receive another FD_QOS event
            //
            iSetQos = SET_QOS_BEFORE;
        }
    }
    return;
}

//
// Function: SrvCondAccept
//
// Description:
//    This is the conditional function for WSAAccept. The QOS service
//    provider is limited in that the QOS values passed into here are
//    unreliable, so the option SET_QOS_DURING is useless unless we call
//    SIO_SET_QOS with our own values (as opposed to the values the client
//    is requesting since those values are supposed to be returned in
//    lpSQOS). Note that on Windows 98, if lpSQOS is not NULL you have to
//    set some QOS values (with SIO_SET_QOS) in the conditional
//    function; otherwise, WSAAccept will fail.
//
int CALLBACK SrvCondAccept(LPWSABUF lpCallerId,
    LPWSABUF lpCallerdata, LPQOS lpSQOS, LPQOS lpGQOS,
    LPWSABUF lpCalleeId, LPWSABUF lpCalleeData, GROUP *g,
    DWORD dwCallbackData)
{
    DWORD       dwBytes = 0;
    SOCKET      s = (SOCKET)dwCallbackData;
    SOCKADDR_IN client;
    int         ret;

    if (nConns == MAX_CONN)
        return CF_REJECT;

    memcpy(&client, lpCallerId->buf, lpCallerId->len);
    printf("Client request: %s\n", inet_ntoa(client.sin_addr));

    if (iSetQos == SET_QOS_EVENT)
```

```
    {
        printf("Setting for event!\n");
        serverQos.SendingFlowspec.ServiceType |=
            SERVICE_NO_QOS_SIGNALING;
        serverQos.ReceivingFlowspec.ServiceType |=
            SERVICE_NO_QOS_SIGNALING;

        ret = WSAIoctl(s, SIO_SET_QOS, &serverQos,
            sizeof(serverQos), NULL, 0, &dwBytes, NULL, NULL);
        if (ret == SOCKET_ERROR)
        {
            printf("WSAIoctl() failed: %d\n",
                WSAGetLastError());
            return CF_REJECT;
        }
    }
    return CF_ACCEPT;
}

//
// Function: Server
//
// Description:
//    This server routine handles incoming client connections.
//    First it sets up the listening socket, sets QOS when
//    appropriate, and waits for incoming clients and events.
//
void Server(SOCKET s)
{
SOCKET          sc[MAX_CONN + 1];
WSAEVENT        hAllEvents[MAX_CONN+1];
SOCKADDR_IN     local,
                client;
int             clientsz,
                ret,
                i;
DWORD           dwBytesRet;
WSANETWORKEVENTS ne;

// Initialize the arrays to invalid values
//
for(i = 0; i < MAX_CONN+1; i++)
{
    hAllEvents[i] = WSA_INVALID_EVENT;
    sc[i] = INVALID_SOCKET;
}
// Array index 0 will be our listening socket
//
hAllEvents[0] = WSACreateEvent();
sc[0]         = s;
nConns        = 0;

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

```
if (bind(s, (SOCKADDR *)&local, sizeof(local)) == SOCKET_ERROR)
{
    printf("bind() failed: %d\n", WSAGetLastError());
    return;
}
listen(s, 7);

if (iSetQos == SET_QOS_BEFORE)
{
    ret = WSAIoctl(sc[0], SIO_SET_QOS, &serverQos,
        sizeof(serverQos), NULL, 0, &dwBytesRet, NULL, NULL);
    if (ret == SOCKET_ERROR)
    {
        printf("WSAIoctl(SIO_SET_QOS) failed: %d\n",
            WSAGetLastError());
        return;
    }
    printf("Set QOS on listening socket:\n");
    PrintQos(&serverQos);
}
if (WSAEventSelect(sc[0], hAllEvents[0], FD_ACCEPT) ==
    SOCKET_ERROR)
{
    printf("WSAEventSelect() failed: %d\n", WSAGetLastError());
    return;
}

while (1)
{
    ret = WSAWaitForMultipleEvents(nConns+1, hAllEvents, FALSE,
        WSA_INFINITE, FALSE);
    if (ret == WSA_WAIT_FAILED)
    {
        printf("WSAWaitForMultipleObject() failed: %d\n",
            WSAGetLastError());
        return;
    }
    if ((i = ret - WSA_WAIT_EVENT_0) > 0)  // Client network event
        HandleClientEvents(sc, hAllEvents, i);
    else
    {
        ret = WSAEnumNetworkEvents(sc[0], hAllEvents[0],
            &ne);
        if (ret == SOCKET_ERROR)
        {
            printf("WSAEnumNetworkevents() failed: %d\n",
                WSAGetLastError());
            return;
        }
        if ((ne.lNetworkEvents & FD_ACCEPT) == FD_ACCEPT)
        {
            if (ne.iErrorCode[FD_ACCEPT_BIT])
                printf("FD_ACCEPT error: %d\n",
                    ne.iErrorCode[FD_ACCEPT_BIT]);
            else
```

```
                printf("FD_ACCEPT\n");

            clientsz = sizeof(client);
            sc[++nConns] = WSAAccept(s, (SOCKADDR *)&client,
                &clientsz, SrvCondAccept, sc[nConns]);
            if (sc[nConns] == SOCKET_ERROR)
            {
                printf("WSAAccept() failed: %d\n",
                    WSAGetLastError());
                nConns--;
                return;
            }
                hAllEvents[nConns] = WSACreateEvent();

                Sleep(10000);
                if (iSetQos == SET_QOS_AFTER)
                {
                    ret = WSAIoctl(sc[nConns], SIO_SET_QOS,
                        &serverQos, sizeof(serverQos), NULL, 0,
                        &dwBytesRet, NULL, NULL);
                    if (ret == SOCKET_ERROR)
                    {
                        printf("WSAIoctl() failed: %d\n",
                            WSAGetLastError());
                        return;
                    }
                }
                ret = WSAEventSelect(sc[nConns],
                    hAllEvents[nConns], FD_READ | FD_WRITE |
                    FD_CLOSE | FD_QOS);
                if (ret == SOCKET_ERROR)
                {
                    printf("WSAEventSelect() failed: %d\n",
                        WSAGetLastError());
                    return;
                }
            }
            if (ne.lNetworkEvents & FD_CLOSE)
                printf("FD_CLOSEn");
            if (ne.lNetworkEvents & FD_READ)
                printf("FD_READn");
            if (ne.lNetworkEvents & FD_WRITE)
                printf("FD_WRITEn");
            if (ne.lNetworkEvents & FD_QOS)
                printf("FD_QOS\n");
        }
    }
    return;
}

//
// Function: Client
//
// Description:
//    The client routine initiates the connection, sets QOS when
//    appropriate, and handles incoming events.
```

```
//
void Client(SOCKET s)
{
    SOCKADDR_IN  server,
                 local;
WSABUF           wbuf;
DWORD            dwBytes,
                 dwBytesSent,
                 dwBytesRecv,
                 dwFlags;
HANDLE           hEvent;
int              ret, i;
char             databuf[DATA_BUFFER_SZ];
QOS             *lpqos;
WSANETWORKEVENTS ne;

hEvent = WSACreateEvent();
if (hEvent == NULL)
{
    printf("WSACreateEvent() failed: %d\n", WSAGetLastError());
    return;
}

lpqos = NULL;
if (iSetQos == SET_QOS_BEFORE)
{
    local.sin_family = AF_INET;
    local.sin_port = htons(0);
    local.sin_addr.s_addr = htonl(INADDR_ANY);

    if (bind(s, (SOCKADDR *)&local, sizeof(local)) ==
        SOCKET_ERROR)
    {
        printf("bind() failed: %d\n", WSAGetLastError());
        return;
    }
    ret = WSAIoctl(s, SIO_SET_QOS, &clientQos,
        sizeof(clientQos), NULL, 0, &dwBytes, NULL, NULL);
    if (ret == SOCKET_ERROR)
    {
        printf("WSAIoctl(SIO_SET_QOS) failed: %d\n",
            WSAGetLastError());
        return;
    }
}
else if (iSetQos == SET_QOS_DURING)
    lpqos = &clientQos;
else if (iSetQos == SET_QOS_EVENT)
{
    clientQos.SendingFlowspec.ServiceType |=
        SERVICE_NO_QOS_SIGNALING;
    clientQos.ReceivingFlowspec.ServiceType |=
        SERVICE_NO_QOS_SIGNALING;

    ret = WSAIoctl(s, SIO_SET_QOS, &clientQos,
```

```
        sizeof(clientQos), NULL, 0, &dwBytes, NULL, NULL);
    if (ret == SOCKET_ERROR)
    {
        printf("WSAIoctl() failed: %d\n", WSAGetLastError());
        return;
    }
}

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));

ret = WSAConnect(s, (SOCKADDR *)&server, sizeof(server),
    NULL, NULL, lpqos, NULL);
if (ret == SOCKET_ERROR)
{
    printf("WSAConnect() failed: %d\n", WSAGetLastError());
    return;
}


ret = WSAEventSelect(s, hEvent, FD_READ | FD_WRITE |
    FD_CLOSE | FD_QOS);
if (ret == SOCKET_ERROR)
{
    printf("WSAEventSelect() failed: %d\n", WSAGetLastError());
    return;
}

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("WSAWaitForMultipleEvents() failed: %d\n",
        WSAGetLastError());
    return;
}

ret = WSAEnumNetworkEvents(s, hEvent, &ne);
if (ret == SOCKET_ERROR)
{
    printf("WSAEnumNetworkEvents() failed: %d\n",
        WSAGetLastError());
    return;
}
if (ne.lNetworkEvents & FD_READ)
```

```
{
    if (ne.iErrorCode[FD_READ_BIT])
        printf("FD_READ error: %d\n",
            ne.iErrorCode[FD_READ_BIT]);
    else
        printf("FD_READ\n");

    wbuf.len = 4096;
    dwFlags = 0;
    ret = WSARecv(s, &wbuf, 1, &dwBytesRecv, &dwFlags,
        NULL, NULL);
    if (ret == SOCKET_ERROR)
    {
        printf("WSARecv() failed: %d\n",
            WSAGetLastError());
        return;
    }
    printf("Read: %d bytes\n", dwBytesRecv);

    wbuf.len = dwBytesRecv;
    ret = WSASend(s, &wbuf, 1, &dwBytesSent, 0, NULL,
        NULL);
    if (ret == SOCKET_ERROR)
    {
        printf("WSASend() failed: %d\n",
            WSAGetLastError());
        return;
    }
    printf("Sent: %d bytes\n", dwBytesSent);
}
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;
        //
        // If the network can't support the bandwidth,
        // don't send
        //
        if (!AbleToSend(s))
        {
            printf("Network is unable to provide "
                "sufficient best-effort bandwidth\n");
            printf("before the reservation "
                "request is approved\n");
        }

        for(i = 0; i < 1; i++)
        {
```

```
            ret = WSASend(s, &wbuf, 1, &dwBytesSent, 0,
                NULL, NULL);
            if (ret == SOCKET_ERROR)
            {
                printf("WSASend() failed: %d\n",
                    WSAGetLastError());
                return;
            }
            printf("Sent: %d bytes\n", dwBytesSent);
        }
    }
}
if (ne.lNetworkEvents & FD_CLOSE)
{
    if (ne.iErrorCode[FD_CLOSE_BIT])
        printf("FD_CLOSE error: %d\n",
            ne.iErrorCode[FD_CLOSE_BIT]);
    else
        printf("FD_CLOSE ...\n");
    closesocket(s);
    WSACloseEvent(hEvent);
    return;
}
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;
ret = WSAIoctl(s, SIO_GET_QOS, NULL, 0,
        buf, QOS_BUFFER_SZ, &dwBytes, NULL, NULL);
if (ret == SOCKET_ERROR)
{
    printf("WSAIoctl(SIO_GET_QOS) failed: %d\n",
        WSAGetLastError());
    return;
}
PrintQos(lpqos);
//
// Check to see whether a status object is returned
// in the QOS structure that might also contain the
// WSA_QOS_RECEIVERS flag
//
if (ChkForQosStatus(lpqos, WSA_QOS_RECEIVERS))
```

```
    bRecvRESV = TRUE;

if (iSetQos == SET_QOS_EVENT)
{
    lpqos->SendingFlowspec.ServiceType =
        clientQos.SendingFlowspec.ServiceType;
    ret = WSAIoctl(s, SIO_SET_QOS, lpqos, dwBytes,
            NULL, 0, &dwBytes, NULL, NULL);
    if (ret == SOCKET_ERROR)
    {
        printf("WSAIoctl(SIO_SET_QOS) failed: %d\n",
            WSAGetLastError());
        return;
    }
    //
    // Change iSetQos so that we don't set QOS again if we
    // receive another FD_QOS event
    //
    iSetQos = SET_QOS_BEFORE;
}
            if (bWaitToSend && bRecvRESV)
            {
                wbuf.buf = databuf;
                wbuf.len = DATA_BUFFER_SZ;

                for(i = 0; i < 1; i++)
                {
                    ret = WSASend(s, &wbuf, 1, &dwBytesSent, 0,
                        NULL, NULL);
                    if (ret == SOCKET_ERROR)
                    {
                        printf("WSASend() failed: %d\n",
                            WSAGetLastError());
                        return;
                    }
                    printf("Sent: %d bytes\n", dwBytesSent);
                }
            }
        }
    }
    return;
}

//
// Function: main
//
// Description:
//    Initialize Winsock, parse command line arguments, create
//    a QOS TCP socket, and call the appropriate handler
//    routine depending on the arguments supplied
//
int main(int argc, char **argv)
{
    WSADATA           wsd;
    WSAPROTOCOL_INFO *pinfo = NULL;
```

```
    SOCKET            s;

    // Parse the command line
    ValidateArgs(argc, argv);
    if (WSAStartup(MAKEWORD(2,2), &wsd) != 0)
    {
        printf("Unable to load Winsock: %d\n", GetLastError());
        return -1;
    }
    pinfo = FindProtocolInfo(AF_INET, SOCK_STREAM, IPPROTO_TCP,
        XP1_QOS_SUPPORTED);
    if (!pinfo)
    {
        printf("unable to find suitable provider!\n");
        return -1;
    }
    printf("Provider returned: %s\n", pinfo->szProtocol);

    s = WSASocket(FROM_PROTOCOL_INFO, FROM_PROTOCOL_INFO,
        FROM_PROTOCOL_INFO, pinfo, 0, WSA_FLAG_OVERLAPPED);
    if (s == INVALID_SOCKET)
    {
        printf("WSASocket() failed: %d\n", WSAGetLastError());
        return -1;
    }
    InitQos();

    if (bServer)
        Server(s);
    else
        Client(s);

    closesocket(s);
    WSACleanup();
    return 0;
}
```

表12-8 Qostcp.c命令行参数

参 数	含 义
-q:[b,d,a,e]	指明在之前（b）、之间（d）、之后（a）或在设置前等待一个FD_QOS事件（e）设置QoS
-s	以服务器的身份运行
-c: 服务器IP地址	以客户机的身份运行，同时与指定IP地址的服务器建立连接
-w	收到一条RESV消息后，再开始发送数据
-r	设置一旦预约被确认便接收通知的选项

12.5.2 单播UDP

采用单播UDP通信，可比TCP多出几种可能性。配套光盘上的UDP示例名字叫作Qosudp.c。它兼具发送者及接收者的双重身份。如作为发送者使用，那么有两种方法可指示 QoS服务提供者将数据发到哪里。请记住，要想建立一个RSVP会话，那么对方的地址是必须提前知道的。要想做到这一点，可在指定QOS_DESTADDR（目标地址）对象的前提下，调用 WSAConnect

或SIO_SET_QOS命令。在我们提供的这个单播UDP示例中，还专门提供了一个参数，用于指示何时对QoS进行设置。假如用户要求QoS在绑定或建立连接之前设置，那么需要在WSAConnect调用中指定QoS参数。假如用户决定在完成了会话的建立之后再来设置QoS，那么在调用WSAConnect的时候，便不必同时设置QoS参数，而且以后调用SIO_SET_QOS的时候，不必提供QOS_DESTADDR对象。最后，假若用户要求只有在接收到一个FD_QOS事件通知的前提下，才对QoS进行设置，那么尽管需要用一个QOS_DESTADDR对象来调用SIO_SET_QOS，但SERVICE_QOS_NO_SIGNALING这个标志需要与ServiceType这个FLOWSPEC字段通过“或”运算合并到一起。

而作为接收端使用时，该程序的选项要少得多。用于指示何时设置QoS的标志在这里全都派不上用场。可在准备接收数据之前设置QoS，亦可要求接收者等待一个FD_QOS事件的发生。这是由于UDP并不接收任何连接请求，即在接受函数的调用过程中，或在会话建立起来之后，并不存在什么QoS设置。作为一个接收者，它亦可选择指定一个不同的过滤器样式，比如固定过滤器，或者“共享显式”过滤器等等。假如指定了一个不同的过滤器，那么必须使用-r:IP选项，明确设定一个IP地址。在RSVP_RESERVE_INFO（预约信息）结构中，SetQosReceivers函数需要用一个RSVP_FILTERSPEC（过滤器规格）结构的内容向其中填充。后者定义了发送者的IP地址。设置过滤器时，要特别留意的一个问题是必须指定发送者的端口号。也就是说，作为接收者，它在同发送者那一方建立绑定关系之前，必须先知道每个发送者的IP地址。

请注意，接收者有时亦可使用WSAConnect将发送者的IP地址同套接字联系到一起。然而，由于UDP接收者可能同时指定多个不同的过滤器样式，而且可能存在多个发送者，所以在此不能使用WSAConnect。要记住的是，假如用WSAConnect关联了一个端点的IP地址，那么发送与接收操作只能针对那个端点进行，同时QoS也会同它联系到一起。

将这个单播UDP的例子同前面的TCP示例比较，会发现两者其实很相似。唯一的例外便是它们采用了不同的方式在套接字上设置QoS。UDP程序要求发送者指定接收者的IP地址，以便调用RSVP，而QoS针对这一目的提供了两种方法。而究其本质，TCP程序默认是在连接调用过程中做到这一点的。对两个程序来说，它们的事件循环几乎完全相同。但请不要忘记UDP应用程序的一个重大“陷阱”：对一个套接字来说，假如使用的不是WSAConnect，那么用SIO_SET_QOS在它上面设置任何QoS选项之前，套接字本身必须先建好“本地绑定”关系！同INADDR_ANY和端口0建立绑定关系是完全合法的；当然，亦可指定一个具体的IP和端口。而假若使用的是WSAConnect调用，那么这种绑定的建立是“隐式”或自动进行的；换言之，毋需我们去干涉。因此，假如在那个时候设置QoS，那么事前并不一定要进行“显式”或明确绑定。

12.5.3 多播UDP

我们提供的最后一个例子是多播QoS，在本书配套光盘上对应的文件名是Qosmcast.c。该程序的核心函数便是大家或许早已熟悉的WSAJoinLeaf。如前一章所述，应用程序必须先通过对该函数的调用，来加入一个多播组。程序加入多播组后，接下来的事情便是传递QoS参数。这儿的多播例子采用的参数与单播UDP的例子完全相同。我们可自行决定何时在套接字上对QoS进行设置。如选择在接受条件函数的执行期间设置QoS，那么QoS会传递进入

WSAJoinLeaf调用之中。否则的话，可随SIO_SET_QOS一道，通过对WSAIoctl的调用，来完成QoS的设置。

对于接收者来说，它允许用户自行设定过滤器的样式：固定过滤器或“共享显式”过滤器。请大家记住，多播UDP在默认情况下采用的是一个通配样式。若指定了一个不同样式的过滤器类型，那么这种改变只会应用于接收者。而且假如对这两种过滤器类型之一感兴趣，可用一个“-r:发送者IP”命令行选项，指定每个发送者的地址。用户可用-f选项来决定采用什么过滤器，若设为se，表明使用“共享显式”过滤器；而若设为ff，表明使用固定过滤器。

对发送者和接收者来说，都可用-m选项来指定要加入的多播组。要注意的是，可重复使用这一选项，从而根据自己的需要，加入任意数量的多播组。-s选项指出当前的程序应作为“发送者”使用。-w选项用于指示发送者，令其在等到一个WSA_QOS_RECEIVERS通知之后，再开始数据的发送。最后，使用-q选项，可指出何时对QoS进行设置。无论什么时候进行QoS的设置，套接字都会与端口5150建立本地绑定关系。在实际应用中，你可选择任意端口；或干脆设为0，让程序为自己自动选择一个端口。然而，假如接收者打算设置固定或共享显式过滤器，那么它必须提供发送者的IP及端口。为使问题简化，我们在此使用了一个固定端口。和单播UDP的例子不同，这儿的接收者不要求事先建立与端口的本地绑定关系，再来设置QoS。原因在于，我们调用WSAJoinLeaf的时候，其实已自动绑定了套接字——如果它尚未绑定的话。大家或许会提出的另一个问题是：能不能用套接字选项命令IP_ADD_MEMBERSHIP（加入组）和IP_DROP_MEMBERSHIP（脱离组）来代替WSAJoinLeaf函数？答案是否定的，你不可这样做！假如使用那些命令，那么请求的QoS参数不会应用于套接字。

对于不同时候设置QoS会发生什么事情，我们在此不打算作深一步的探讨，因为这与单播UDP的情况非常相似。经过这一系列的学习，大家现在应已非常熟悉如何建立RSVP会话，以及生成PATH和RESV消息需要哪些必要条件！

12.6 ATM和QoS

如早先所述，QoS是ATM网络一种“与生俱来”能力。无论Windows 2000还是Windows 98（安装SP1的服务）都支持来自Winsock的、固有的ATM编程技术，这在第6章已进行了非常全面的论述。QoS是ATM网络默认提供的一种功能；换言之，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。这些组件包括许可控制服务（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。

除我们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。第一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。

这样便导致了第二项区别：两个ATM提供者均是“面向连接”的。因此，我们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进入的连接。这同时意味着，在连接设置期间， QoS必须设好！注意不能在一个已经建好的连接上设置QoS。

这样便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。换言之，表12.6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。

12.7 小结

QoS使应用程序如虎添翼，可放心大胆地享用一个稳定的网络服务等级，从而保障视频及音频信号等流畅地播放。建立 QoS连接显得多少有些麻烦，但千万不要畏难。最需要掌握的便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。

第13章　原始套接字

利用“原始套接字”（Raw Socket），我们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的一些实用工具，比如Traceroute和Ping程序等等。使用原始套接字，亦可对IP头信息进行实际的操作。本章只关心IP协议；至于如何针对其他协议使用原始套接字，我们不打算提及。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Microsoft Windows CE还是老版本的Windows 95（无Winsock 2升级）均不能利用原始套接字的能力。

此外，要想顺利使用原始套接字，要求对基层的协议结构有一定程度的认识，而那已超出了本书的范围。在这一章中，我们打算讨论Internet控制消息协议（ICMP）、Internet组管理协议（IGMP）以及用户数据报协议（UDP）。ICMP会由Ping这个实用程序用到，以便探测到某个主机的路由是否有效和畅通，看看对方的机器是否会作出响应。对程序开发者来说，经常都要用到一种程序化的方法，以便判断一台机器是否“活动”，网络数据能否抵达它。IP多播通信利用IGMP将多播组成员信息通告给路由器。大多数Win32平台目前都增加了对IGMP第2版的支持。但在某些情况下，我们也需要送出自己的IGMP数据包，以便脱离组成员关系。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，讲述如何发送自己的IGMP包。对这三种协议来说，我们都只会讲解与本章示范代码及示范程序密切相关的那些部分。

13.1　原始套接字的创建

要想使用原始套接字，第一步便是创建它。可用socket命令或WSASocket调用来做到这一点。注意在典型情况下，在Winsock为IP列出的目录中，并不存在SOCK_RAW这一套接字类型。然而，这并不能妨碍我们创建此种类型的套接字。它的意思只是说，我们不能用一个WSAPROTOCOL_INFO结构来创建一个原始套接字。请参考第5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，必须自行设定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，来完成一个原始套接字的创建：

```
SOCKET    s;

s = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
// Or
s = WSASocket(AF_INET, SOCK_RAW, IPPROTO_ICMP, NULL, 0,
    WSA_FLAG_OVERLAPPED);
if (s == INVALID_SOCKET)
{
    // Socket creation failed
}
```

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

而造成了Windows NT下一个潜在的安全漏洞。因此，只有属于“管理员”（Administrators）组的成员，才有权创建类型为SOCK_RAW的套接字。而Windows 95和Windows 98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。

在上述示范代码中，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，请注意其中存在的一处限制：在Windows NT 4、Windows 98以及Windows 95（安装Winsock 2）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，而该选项在上述平台下都是不支持的。然而，Windows 2000确实提供了对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，接下来的事情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头会包含在接收到的任何返回数据中，无论是否设定了IP_HDRINCL选项。

13.2 Internet控制消息协议

Internet控制消息协议（ICMP）是便于不同主机间传递简短消息的一种机制。大多数ICMP消息都牵涉到主机间通信时发生的一些错误；而其他ICMP消息用于对主机进行查询。ICMP协议采用的是IP定址机制，因为它本身就是封装在IP数据报内的一种协议。在图13-1中，我们展示了ICMP消息的各个字段。整条ICMP消息封装在一个IP头内。

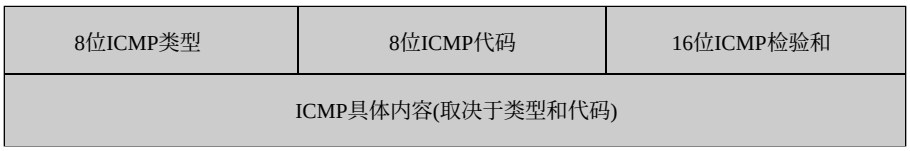

图13-1 ICMP头

其中，第一个字段指定的是ICMP消息类型，可分为查询或错误两类。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头内容的一个补余求和。最后，ICMP的实际内容要依赖于前面设定的ICMP类型及代码。在表13-1中，我们对那些类型及代码进行了详细总结。

若生成的是一条ICMP错误消息，那么在消息中，肯定包含了导致那个ICMP错误的IP头，以及IP数据报的头8个字节。这样一来，收到ICMP错误的主机便可将消息与一种特定的协议联系起来。并可根据实际情况，对那个错误作出处理。就我们目前的情况来说，Ping需要依赖于回应请求及回应答复这两种ICMP查询，而不是仅仅依赖一条错误消息。生成ICMP消息的主机会通过TCP或UDP，对问题作出响应；除此之外，ICMP的用处并不大。在下一节里，

我们打算讨论如何随原始套接字一道，用 ICMP协议来生成一个Ping请求， Ping请求是用来回应请求和回应答复消息的。

表13-1 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（Echo Reply）
3	错误：目标不可抵达	0	网络访问不到
		1	主机访问不到
		2	协议访问不到
		3	端口访问不到
		4	数据包需要分段（分解），但却没有设置分段位
		5	源路由失效
		6	目标网络未知
		7	目标主机未知
		8	源主机独立（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），网络不可抵达
		12	针对特定的TOS（服务类型），主机不可抵达
		13	由于过滤，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	和TOS和网络重定向
		3	和TOS和主机重定向
8	查询	0	回应请求（Echo Request）
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
13	查询	0	时间戳请求
14	查询	0	时间戳答复
15	查询	0	信息请求
16	查询	0	信息答复
17	查询	0	地址掩码请求
18	查询	0	地址掩码答复

13.2.1 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，并且是否可以通过网络访问到。通过生成一个ICMP“回应请求”（Echo Request），并将其定向至打算查询的目标主机，便可知道自己是否能成功地访问到那台机器。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！概括地说，我们的这个Ping示例需要采取下面这些步骤：

1) 创建类型为SOCK_RAW的一个套接字，同时设定协议IPPROTO_ICMP。

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。

发出ICMP回应请求以后，远程机器会拦截这个请求，然后生成一条回应答复消息，再通过网络传回给我们。假如出于某些方面的原因，不能抵达目标主机，就会生成对应的 ICMP错误消息（比如“目标主机访问不到”），由原先打算建立通信的那个路径上某处的一个路由器返回。假定与远程主机的物理性连接并不存在问题，但远程主机已经关机或没有设置对网络事件作出响应，便需由自己的程序来执行超时检定，来侦测出这样的情况。程序清单 13-1列出的Ping.c示例向大家清晰展示了如何创建一个特殊的套接字，以便实现 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。

程序清单13-1 Ping.c

```
// Module Name: Ping.c
//
// Command Line Options/Parameters:
//     Ping [host] [packet-size]
//
//     host         String name of host to ping
//     packet-size  Integer size of packet to send
//                  (smaller than 1024 bytes)
//
//#pragma pack(1)

#define WIN32_LEAN_AND_MEAN
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>
#include <stdlib.h>

#define IP_RECORD_ROUTE  0x7
//
// IP header structure
//
typedef struct _iphdr
{
    unsigned int   h_len:4;         // Length of the header
    unsigned int   version:4;       // Version of IP
    unsigned char  tos;             // Type of service
    unsigned short total_len;       // Total length of the packet
    unsigned short ident;           // Unique identifier
    unsigned short frag_and_flags; // Flags
    unsigned char  ttl;             // Time to live
    unsigned char  proto;           // Protocol (TCP, UDP, etc.)
    unsigned short checksum;        // IP checksum

    unsigned int   sourceIP;
    unsigned int   destIP;
} IpHeader;
```

```
#define ICMP_ECHO        8
#define ICMP_ECHOREPLY   0
#define ICMP_MIN         8 // Minimum 8-byte ICMP packet (header)

//
// ICMP header structure
//
typedef struct _icmphdr
{
    BYTE   i_type;
    BYTE   i_code;                 // Type sub code
    USHORT i_cksum;
    USHORT i_id;
    USHORT i_seq;
    // This is not the standard header, but we reserve space for time
    ULONG  timestamp;
} IcmpHeader;

//
// IP option header--use with socket option IP_OPTIONS
//
typedef struct _ipoptionhdr
{
    unsigned char        code;        // Option type
    unsigned char        len;         // Length of option hdr
    unsigned char        ptr;         // Offset into options
    unsigned long        addr[9];     // List of IP addrs
} IpOptionHeader;
#define DEF_PACKET_SIZE  32        // Default packet size
#define MAX_PACKET       1024      // Max ICMP packet size
#define MAX_IP_HDR_SIZE  60        // Max IP header size w/options

BOOL  bRecordRoute;
int   datasize;
char *lpdest;

//
// Function: usage
//
// Description:
//    Print usage information
//
void usage(char *progname)
{
    printf("usage: ping -r <host> [data size]\n");
    printf("       -r           record route\n");
    printf("        host        remote machine to Ping\n");
    printf("        datasize    can be up to 1 KB\n");
    ExitProcess(-1);
}

//
// Function: FillICMPData
//
// Description:
```

```
//    Helper function to fill in various fields for our ICMP request
//
void FillICMPData(char *icmp_data, int datasize)
{
    IcmpHeader *icmp_hdr = NULL;
    char       *datapart = NULL;

    icmp_hdr = (IcmpHeader*)icmp_data;
    icmp_hdr->i_type = ICMP_ECHO;        // Request an ICMP echo
    icmp_hdr->i_code = 0;
    icmp_hdr->i_id = (USHORT)GetCurrentProcessId();
    icmp_hdr->i_cksum = 0;
    icmp_hdr->i_seq = 0;

    datapart = icmp_data + sizeof(IcmpHeader);
    //
    // Place some junk in the buffer
    //
    memset(datapart,'E', datasize - sizeof(IcmpHeader));
}
//
// Function: checksum
//
// Description:
//    This function calculates the 16-bit one's complement sum
//    of the supplied buffer (ICMP) header
//
USHORT checksum(USHORT *buffer, int size)
{
    unsigned long cksum=0;

    while (size > 1)
    {
        cksum += *buffer++;
        size -= sizeof(USHORT);
    }
    if (size)
    {
        cksum += *(UCHAR*)buffer;
    }
    cksum = (cksum >> 16) + (cksum & 0xffff);
    cksum += (cksum >>16);
    return (USHORT)(~cksum);
}

//
// Function: DecodeIPOptions
//
// Description:
//    If the IP option header is present, find the IP options
//    within the IP header and print the record route option
//    values
//
void DecodeIPOptions(char *buf, int bytes)
{
```

```
    IpOptionHeader *ipopt = NULL;
    IN_ADDR         inaddr;
    int             i;
    HOSTENT        *host = NULL;

    ipopt = (IpOptionHeader *)(buf + 20);

    printf("RR:   ");
    for(i = 0; i < (ipopt->ptr / 4) - 1; i++)
    {
        inaddr.S_un.S_addr = ipopt->addr[i];
        if (i != 0)
            printf("      ");
        host = gethostbyaddr((char *)&inaddr.S_un.S_addr,
                    sizeof(inaddr.S_un.S_addr), AF_INET);
        if (host)
            printf("(%-15s) %s\n", inet_ntoa(inaddr), host->h_name);
        else
            printf("(%-15s)\n", inet_ntoa(inaddr));
    }
    return;
}

//
// Function: DecodeICMPHeader
//
// Description:
//    The response is an IP packet. We must decode the IP header to
//    locate the ICMP data.
//
void DecodeICMPHeader(char *buf, int bytes,
    struct sockaddr_in *from)
{
    IpHeader       *iphdr = NULL;
    IcmpHeader     *icmphdr = NULL;
    unsigned short  iphdrlen;
    DWORD           tick;
    static   int    icmpcount = 0;

    iphdr = (IpHeader *)buf;
    // Number of 32-bit words * 4 = bytes
    iphdrlen = iphdr->h_len * 4;
    tick = GetTickCount();

    if ((iphdrlen == MAX_IP_HDR_SIZE) && (!icmpcount))
        DecodeIPOptions(buf, bytes);

    if (bytes  < iphdrlen + ICMP_MIN)
    {
        printf("Too few bytes from %s\n",
            inet_ntoa(from->sin_addr));
    }
    icmphdr = (IcmpHeader*)(buf + iphdrlen);

    if (icmphdr->i_type != ICMP_ECHOREPLY)
```

```
        {
            printf("nonecho type %d recvd\n", icmphdr->i_type);
            return;
        }
        // Make sure this is an ICMP reply to something we sent!
        //
        if (icmphdr->i_id != (USHORT)GetCurrentProcessId())
        {
            printf("someone else's packet!\n");
            return ;
        }
        printf("%d bytes from %s:", bytes, inet_ntoa(from->sin_addr));
        printf(" icmp_seq = %d. ", icmphdr->i_seq);
        printf(" time: %d ms", tick - icmphdr->timestamp);
        printf("\n");

        icmpcount++;
        return;
    }

    void ValidateArgs(int argc, char **argv)
    {
        int                i;

        bRecordRoute = FALSE;
        lpdest = NULL;
        datasize = DEF_PACKET_SIZE;

        for(i = 1; i < argc; i++)
        {
            if ((argv[i][0] == '-') || (argv[i][0] == '/'))
            {
                switch (tolower(argv[i][1]))
                {
                    case 'r':        // Record route option
                        bRecordRoute = TRUE;
                        break;
                    default:
                        usage(argv[0]);
                        break;
                }
            }
            else if (isdigit(argv[i][0]))
                datasize = atoi(argv[i]);
            else
                lpdest = argv[i];
        }
    }

    //
    // Function: main
    //
    // Description:
    //    Set up the ICMP raw socket, and create the ICMP header. Add
    //    the appropriate IP option header, and start sending ICMP
```

```
//    echo requests to the endpoint. For each send and receive,
//    we set a timeout value so that we don't wait forever for a
//    response in case the endpoint is not responding. When we
//    receive a packet, decode it.
//
int main(int argc, char **argv)
{

    WSADATA            wsaData;
    SOCKET             sockRaw = INVALID_SOCKET;
    struct sockaddr_in dest,
                       from;
    int                bread,
                       fromlen = sizeof(from),
                       timeout = 1000,
                       ret;
    char               *icmp_data = NULL,
                       *recvbuf = NULL;
    unsigned int       addr = 0;
    USHORT             seq_no = 0;
    struct hostent     *hp = NULL;
    IpOptionHeader     ipopt;


    if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0)
    {
        printf("WSAStartup() failed: %d\n", GetLastError());
        return -1;
    }
    ValidateArgs(argc, argv);

    //
    // WSA_FLAG_OVERLAPPED flag is required for SO_RCVTIMEO,
    // SO_SNDTIMEO option. If NULL is used as last param for
    // WSASocket, all I/O on the socket is synchronous, the
    // internal user mode wait code never gets a chance to
    // execute, and therefore kernel-mode I/O blocks forever.
    // A socket created via the socket function has the over-
    // lapped I/O attribute set internally. But here we need
    // to use WSASocket to specify a raw socket.
    //
    // If you want to use timeout with a synchronous
    // nonoverlapped socket created by WSASocket with last
    // param set to NULL, you can set the timeout by using
    // the select function, or you can use WSAEventSelect and
    // set the timeout in the WSAWaitForMultipleEvents
    // function.
    //
    sockRaw = WSASocket (AF_INET, SOCK_RAW, IPPROTO_ICMP, NULL, 0,
                         WSA_FLAG_OVERLAPPED);
    if (sockRaw == INVALID_SOCKET)
    {
        printf("WSASocket() failed: %d\n", WSAGetLastError());
        return -1;
    }
    if (bRecordRoute)
```

```
{
    // Setup the IP option header to go out on every ICMP packet
    //
    ZeroMemory(&ipopt, sizeof(ipopt));
    ipopt.code = IP_RECORD_ROUTE; // Record route option
    ipopt.ptr  = 4;               // Point to the first addr offset
    ipopt.len  = 39;              // Length of option header

    ret = setsockopt(sockRaw, IPPROTO_IP, IP_OPTIONS,
        (char *)&ipopt, sizeof(ipopt));
    if (ret == SOCKET_ERROR)
    {
        printf("setsockopt(IP_OPTIONS) failed: %d\n",
            WSAGetLastError());
    }
}
// Set the send/recv timeout values
//
bread = setsockopt(sockRaw, SOL_SOCKET, SO_RCVTIMEO,
            (char*)&timeout, sizeof(timeout));
if(bread == SOCKET_ERROR)
{
    printf("setsockopt(SO_RCVTIMEO) failed: %d\n",
        WSAGetLastError());
    return -1;
}
timeout = 1000;
bread = setsockopt(sockRaw, SOL_SOCKET, SO_SNDTIMEO,
            (char*)&timeout, sizeof(timeout));
if (bread == SOCKET_ERROR)
{
    printf("setsockopt(SO_SNDTIMEO) failed: %d\n",
        WSAGetLastError());
    return -1;
}
memset(&dest, 0, sizeof(dest));
//
// Resolve the endpoint's name if necessary
//
dest.sin_family = AF_INET;
if ((dest.sin_addr.s_addr = inet_addr(lpdest)) == INADDR_NONE)
{
    if ((hp = gethostbyname(lpdest)) != NULL)
    {
        memcpy(&(dest.sin_addr), hp->h_addr, hp->h_length);
        dest.sin_family = hp->h_addrtype;
        printf("dest.sin_addr = %s\n", inet_ntoa(dest.sin_addr));
    }
    else
    {
        printf("gethostbyname() failed: %d\n",
            WSAGetLastError());
        return -1;
    }
}
```

```
//
// Create the ICMP packet
//
datasize += sizeof(IcmpHeader);

icmp_data = HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY,
              MAX_PACKET);
recvbuf = HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY,
              MAX_PACKET);
if (!icmp_data)
{
    printf("HeapAlloc() failed: %d\n", GetLastError());
    return -1;
}
memset(icmp_data,0,MAX_PACKET);
FillICMPData(icmp_data,datasize);
//
// Start sending/receiving ICMP packets
//
while(1)
{
    static int nCount = 0;
    int        bwrote;

    if (nCount++ == 4)
        break;
    ((IcmpHeader*)icmp_data)->i_cksum = 0;
    ((IcmpHeader*)icmp_data)->timestamp = GetTickCount();
    ((IcmpHeader*)icmp_data)->i_seq = seq_no++;
    ((IcmpHeader*)icmp_data)->i_cksum =
        checksum((USHORT*)icmp_data, datasize);

    bwrote = sendto(sockRaw, icmp_data, datasize, 0,
                (struct sockaddr*)&dest, sizeof(dest));
    if (bwrote == SOCKET_ERROR)
    {
        if (WSAGetLastError() == WSAETIMEDOUT)
        {
            printf("timed out\n");
            continue;
        }
        printf("sendto() failed: %d\n", WSAGetLastError());
        return -1;
    }
    if (bwrote < datasize)
    {
        printf("Wrote %d bytes\n", bwrote);
    }
    bread = recvfrom(sockRaw, recvbuf, MAX_PACKET, 0,
                (struct sockaddr*)&from, &fromlen);
    if (bread == SOCKET_ERROR)
    {
        if (WSAGetLastError() == WSAETIMEDOUT)
        {
            printf("timed out\n");
            continue;
```

```
            }
            printf("recvfrom() failed: %d\n", WSAGetLastError());
            return -1;
        }
        DecodeICMPHeader(recvbuf, bread, &from);

        Sleep(1000);
    }
    // Cleanup
    //
    if (sockRaw != INVALID_SOCKET)
        closesocket(sockRaw);
    HeapFree(GetProcessHeap(), 0, recvbuf);
    HeapFree(GetProcessHeap(), 0, icmp_data);

    WSACleanup();
    return 0;
}
```

对这个Ping示范程序来说，它最引人注目的一个特点便是采用了IP_OPTIONS套接字选项。之所以要使用记录路由IP选项，是由于这样一来，当我们的ICMP包抵达路由器时，它的IP地址便能自动添加到IP选项头内——具体插入位置则取决于事先在IP选项头内设好的偏移量字段。每次遇到一个路由器向其中加入自己的IP地址时，这个偏移距离都会自动递增4。之所以是“4”，而不是其他数字，是由于对IPv4地址来说，它的长度正好是4个字节。本书不打算对IPv6的情况作深入探讨，目前也没有任何一种Windows平台提供了对它的支持。一旦收到回应答复，便可对选项头进行解析，列印出中途访问过的那些路由器的IP地址及主机名。大家可参考第9章的内容，了解还能使用其他什么类型的IP选项。

13.2.2 Traceroute示例

对IP网络来说，另一个非常有用的工具是Traceroute（路由追踪）。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，我们可侦测出为抵达网络内任何一个指定的主机，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IP选项头内的记录路由选项，侦测中途经过的路由器IP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。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”。若网络较大（如Internet），穿过的路由器不止9个，便应换用Traceroute。

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IP的“存活时间”（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一个路由器，TTL首先会超时（变成0）。这样便会造成路由器生成一个ICMP“超时”数据包。随后，最初的TTL值递增1，以便UDP包能继续传到下一个路由器，而生成的ICMP超时包会自第一个路由器返回。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地址勾勒出一个清晰的轮廓，直到最终抵达目标主机。一旦TTL的值递增得足够大，可以实际抵达目标位置，通常便会返回一条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。第一个办法，可使用UDP包和发送数据报，连续递增更改TTL的值。TTL每一次“超时”，都会向我们返回一条ICMP消息。这种方法要求使用安装了UDP协议的一个套接字来发送数据，同时用安装了 ICMP协议的另一个套接字来接收数据（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 7章中所见。而 ICMP套接字的类型比较特殊，是 SOCK_RAW（原始套接字），并设定了IPPROTO_ICMP协议。UDP套接字的TTL值需要通过IP_TTL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，这要求程序员进行更多的工作。

第二个办法，我们只需将 ICMP数据包简单地发给目的地，同时连续递增更改 TTL的值。在TTL“超时”的时候，这样做也会造成一条 ICMP错误消息的返回。注意这种方法和 Ping有着某些共通之处，它也只要求使用一个套接字（安装 ICMP协议）。在本书配套光盘的示范代码文件夹下，大家可找到一个使用了 ICMP数据包的Traceroute例子。本章正文并不打算列出它的完整源码，因为它的大部分代码都与前面的 Ping例子相同。

13.3 Internet组管理协议

Internet组管理协议（IGMP）由IP多播通信专用，可对多播组的成员加入或脱离组进行管理。大家可参考第11章的内容，了解如何通过Winsock来加入及离开一个多播组。某个应用程序加入了一个多播组后，便会向本地网络的每个路由器都发出一条 IGMP消息，使用特殊地址224.0.0.2，该地址固定对应于“所有路由器”组。根据这条 IGMP消息，路由器便可知道以后发给指定多播地址的数据需要转发给这个新加入的成员。若没有这项能力，多播数据便和广播数据没什么区别了。

现在来澄清一些事实。IGMP协议共有两个版本：版本1（IGMPv1）和版本2（IGMPv2），分别由RFC 1112和RFC 2236文件加以定义。两者的主要区别在于：版本 1没有提供专门的方法，让一个主机指示路由器停止转发送给一个多播组的数据。换言之，若某个主机决定脱离组成员关系，便没有办法将这种情况通报给路由器，而路由器当然会坚持不懈地向其转发数据，直至在其发出一个组查询后，发现对方没有“应答”。在协议第2版中，则增加了一条能够明确表明自己要“离开”的消息。这样一来，一旦打算放弃成员资格，主机便可向路由器发出通知，使其能够立即知道这一情况。除此以外，两个版本的头格式也存在一些不同。图13-2展示了版本1的头结构，图13-3则是版本的2头结构。由于长度仅为8字节，所以这两个头其实都非常简单。

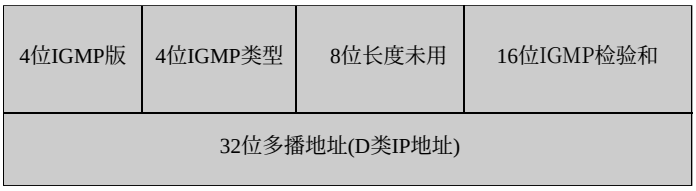

图13-2 IGMPv1头格式

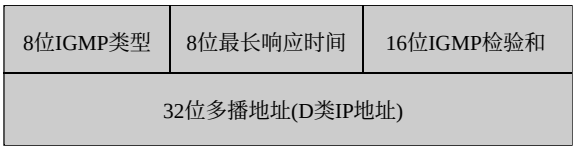

图13-3 IGMPv2头格式

其中，版本 1的头含有两个 4位字段。第一个字段指定 IGMP版本，而第二个字段指定IGMP消息类型。在版本 2中，单独一个 8位字段代替了这两个字段。此外，版本 1中未用的字段在新版本中派上了用场，用来指定最长的响应时间。只有在成员关系查询消息中，“ 最长响应时间 ”字段才有意义：它代表在发出一个响应报告之前，允许等候的最长时间；采用的单位是0.1秒。在其他所有消息中，发送都会将该字段设为 0，而接收者也会忽略它。

对IGMPv1来说，版本字段必然设为 1，而类型字段有两个可选的值，如表 13-2所示。使用主机成员资格查询（0x1）消息，路由器可判断一个主机正在使用哪些多播组。在这种情况下，组地址字段的值应设为 0。路由器会将查询数据包发给 “ 所有主机 ” 地址（ 224.0.0.1）。假如有主机仍是某个组的成员，那么各个主机都会将一个主机成员资格报告（ IGMP消息类型0x2）反馈回 “ 所有路由器 ” 地址，同时指出自己加入的是哪个多播地址。假如一个主机是首次加入一个组，那么一条组成员资格报告消息也会发给 “ 所有路由器 ” 组。

如表13-3所示，IGMP的第2版加入了两种新的消息类型 —— 版本2成员资格报告（0x16），以及离开组（0x17）消息。注意另两类消息在作用上与版本 1是相同的（主机成员资格查询和成员资格报告），尽管为其分配的值不同。然而，请注意版本 2的IGMP包已将版本及类型字段浓缩到单独一个字段里，而 0x11展开成二进制，便是 00010001。从表面看，完全相当于 “ 版本等于1，类型等于1 ” ！

表13-2 IGMP版本1的消息类型

类型	说 明
0x1	主机成员资格查询
0x2	主机成员资格报告

表13-3 IGMP版本2的消息类型

类型	说 明
0x11	成员资格查询
0x12	版本1成员资格报告
0x16	版本2成员资格报告
0x17	离开组

版本2的成员资格查询（0x11）与版本1的成员资格查询差别是极其微小的。前者不仅能进行版本1那样的标准查询，也能在一个网络上，查询在一个指定的组地址中，包括了哪些组成员。具体的做法是向 “ 所有路由器 ” 组发送一个成员资格查询包，同时在组地址字段中指定一个打算查询的目标组。版本 2的成员资格报告则是最新设计的，因为它允许使用 “ 最长超时 ” 字段，对主机响应查询的时间作出限制。如超过时间仍未作出响应，便会禁止向其转发特定多播组的数据。

13.4 IP_HDRINCL的使用

对原始套接字来说，它存在的一项限制在于，我们只能对已经定义好的协议进行操作，比如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。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。

此外，假如想在应用程序中实现自己的协议方案，并将其封装到IP内，那么首先可以创建一个原始套接字，然后将协议类型设为IPPROTO_RAW。这样一来，我们就可在IP头内人工设置协议字段，同时构建自己的定制协议头。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，便只是“小菜一碟”。

使用IP_HDRINCL选项时，必须针对每一次发送调用，向IP头内自行填充内容。同时，还要填写封装在其中的其他任何协议头。IP头的结构已在第9章进行了讲述。大家可参考图9-3，以及专门讲述IP_HDRINCL选项的那一小节。与IP相比，UDP头要简单得多。长度仅为8个字节，而且只包含了四个字段，如图13-4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。第三个字段对应UDP长度；以字节为单位，指定UDP头以及数据的总长。第四个字段则是校验和，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检验和

图13-4 UDP头的格式

由于UDP是一种不能保证数据可靠传输的协议（即“不可靠”协议），所以校验和的计算是可选的。然而，为保持大家知识系统的完整性，在此还是要对其进行解释。与IP校验和不同，UDP校验和除覆盖了UDP头之外，还同时覆盖了实际的数据，此外还包括IP头的一部分。而IP校验和只针对IP头进行计算。要求用于计算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（零除外）。
- 8位协议。
- 16位UDP长度。

加入这些项目的是UDP头以及数据。校验和的计算方法和IP与ICMP的方法是相同的：得出16位1的求余总和。由于数据可能是个奇数，所以有时需要在数据末尾填充一个0字节，以便顺利计算出校验和。这个填充字段并不作为数据的一部分传递。在图13-5中，我们向大家演示了计算校验和所需的全部字段。头三个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，由于校验和是以16位值为基础计算出来的，所以或许需要用一个0字节对数据进行填充。

32位源IP地址		
32位目的IP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检验和
数据(任意字节)		

图13-5 UDP伪头

程序清单13-2列出的示范程序作用很简单，从我们选择的任何IP及端口，它向任何指定的目标IP及端口地址发送一个UDP数据包。第一步是创建一个原始套接字，然后设置IP_HDRINCL标志：

```
SOCKET   s;
BOOL     bOpt;

s = WSASocket(AF_INET, SOCK_RAW, IPPROTO_UDP, NULL, 0,
        WSA_FLAG_OVERLAPPED);
ret = setsockopt(s, IPPROTO_IP, IP_HDRINCL, (char *)&bOpt,
        sizeof(bOpt));
```

注意，我们现已创建了一个协议为IPPROTO_UDP的原始套接字。这种做法只适用于Windows 2000，因为它同时也要求设置IP_HDRINCL选项。示范程序Iphdrinc.c向大家阐释了如何利用原始套接字和IP_HDRINCL选项，操作传出去的数据包的IP与UDP头。

程序清单13-2 原始UDP示例

```
// Module Name: Iphdrinc.c
//
#pragma pack(1)

#define WIN32_LEAN_AND_MEAN

#include <winsock2.h>
#include <ws2tcpip.h>

#include <stdio.h>
#include <stdlib.h>

#define MAX_MESSAGE         4068
#define MAX_PACKET          4096
//
// Set up some default values
//
#define DEFAULT_PORT        5150
#define DEFAULT_IP          "10.0.0.1"
#define DEFAULT_COUNT       5
#define DEFAULT_MESSAGE     "This is a test"

//
// Define the IP header. Make the version and length fields one
// character since we can't declare two 4-bit fields without
// the compiler aligning them on at least a 1-byte boundary.
```

```
//
typedef struct ip_hdr
{
    unsigned char  ip_verlen;        // IP version & length
    unsigned char  ip_tos;           // IP type of service
    unsigned short ip_totallength;   // Total length
    unsigned short ip_id;            // Unique identifier
    unsigned short ip_offset;        // Fragment offset field
    unsigned char  ip_ttl;           // Time to live
    unsigned char  ip_protocol;      // Protocol(TCP, UDP, etc.)
    unsigned short ip_checksum;      // IP checksum
    unsigned int   ip_srcaddr;       // Source address
    unsigned int   ip_destaddr;      // Destination address
} IP_HDR, *PIP_HDR, FAR* LPIP_HDR;
//
// Define the UDP header
//
typedef struct udp_hdr
{
    unsigned short src_portno;       // Source port number
    unsigned short dst_portno;       // Destination port number
    unsigned short udp_length;       // UDP packet length
    unsigned short udp_checksum;     // UDP checksum (optional)
} UDP_HDR, *PUDP_HDR;

//
// Global variables
//
unsigned long  dwToIP,               // IP to send to
               dwFromIP;             // IP to send from (spoof)
unsigned short iToPort,              // Port to send to
               iFromPort;            // Port to send from (spoof)
DWORD          dwCount;              // Number of times to send
char           strMessage[MAX_MESSAGE]; // Message to send

//
// Description:
//    Print usage information and exit
//
void usage(char *progname)
{
    printf("usage: %s [-fp:int] [-fi:str] [-tp:int] [-ti:str]\
        [-n:int] [-m:str]\n", progname);
    printf("       -fp:int   From (sender) port number\n");
    printf("       -fi:IP    From (sender) IP address\n");
    printf("       -fp:int   To (recipient) port number\n");
    printf("       -fi:IP    To (recipient) IP address\n");
    printf("       -n:int    Number of times to read message\n");
    printf("       -m:str    Size of buffer to read\n\n");
    ExitProcess(1);
}
//
// Function: ValidateArgs
//
// Description:
```

```
//    Parse the command line arguments, and set some global flags to
//    indicate the actions to perform
//
void ValidateArgs(int argc, char **argv)
{
    int                i;

    iToPort = DEFAULT_PORT;
    iFromPort = DEFAULT_PORT;
    dwToIP = inet_addr(DEFAULT_IP);
    dwFromIP = inet_addr(DEFAULT_IP);
    dwCount = DEFAULT_COUNT;
    strcpy(strMessage, DEFAULT_MESSAGE);

    for(i = 1; i < argc; i++)
    {
        if ((argv[i][0] == '-') || (argv[i][0] == '/'))
        {
            switch (tolower(argv[i][1]))
            {
                case 'f':        // From address
                    switch (tolower(argv[i][2]))
                    {
                        case 'p':
                            if (strlen(argv[i]) > 4)
                                iFromPort = atoi(&argv[i][4]);
                            break;
                        case 'i':
                            if (strlen(argv[i]) > 4)
                                dwFromIP = inet_addr(&argv[i][4]);
                            break;
                        default:
                            usage(argv[0]);
                            break;
                    }
                    break;
                case 't':        // To address
                    switch (tolower(argv[i][2]))
                    {
                        case 'p':
                            if (strlen(argv[i]) > 4)
                                iToPort = atoi(&argv[i][4]);
                            break;
                        case 'i':
                            if (strlen(argv[i]) > 4)
                                dwToIP = inet_addr(&argv[i][4]);
                            break;
                        default:
                            usage(argv[0]);
                            break;
                    }
                    break;
                case 'n':        // Number of times to send message
                    if (strlen(argv[i]) > 3)
                        dwCount = atol(&argv[i][3]);
                    break;
```

```
                case 'm':
                    if (strlen(argv[i]) > 3)
                        strcpy(strMessage, &argv[i][3]);
                    break;
                default:
                    usage(argv[0]);
                    break;
            }
        }
    }
    return;
}

//
// Function: checksum
//
// Description:
//    This function calculates the 16-bit one's complement sum
//    for the supplied buffer
//
USHORT checksum(USHORT *buffer, int size)
{
    unsigned long cksum=0;

    while (size > 1)
    {
        cksum += *buffer++;
        size  -= sizeof(USHORT);
    }
    if (size)
    {
        cksum += *(UCHAR*)buffer;
    }
    cksum = (cksum >> 16) + (cksum & 0xffff);
    cksum += (cksum >>16);

    return (USHORT)(~cksum);
}
//
// Function: main
//
// Description:
//    First parse command line arguments and load Winsock. Then
//    create the raw socket and set the IP_HDRINCL option.
//    Following this, assemble the IP and UDP packet headers by
//    assigning the correct values and calculating the checksums.
//    Then fill in the data and send to its destination.
//
int main(int argc, char **argv)
{
    WSADATA            wsd;
    SOCKET             s;
    BOOL               bOpt;
    struct sockaddr_in remote;       // IP addressing structures
    IP_HDR             ipHdr;
    UDP_HDR            udpHdr;
    int                ret;
    DWORD              i;
```

```
    unsigned short     iTotalSize,   // Lots of sizes needed to fill
                       iUdpSize,     // the various headers with
                       iUdpChecksumSize,
                       iIPVersion,
                       iIPSize,
                       cksum = 0;
    char               buf[MAX_PACKET],
                      *ptr = NULL;
    IN_ADDR            addr;

    // Parse command line arguments, and print them out
    //
    ValidateArgs(argc, argv);
    addr.S_un.S_addr = dwFromIP;
    printf("From IP: <%s>\n     Port: %d\n", inet_ntoa(addr),
        iFromPort);
    addr.S_un.S_addr = dwToIP;
    printf("To   IP: <%s>\n     Port: %d\n", inet_ntoa(addr),
        iToPort);
    printf("Message: [%s]\n", strMessage);
    printf("Count:   %d\n", dwCount);

    if (WSAStartup(MAKEWORD(2,2), &wsd) != 0)
    {
        printf("WSAStartup() failed: %d\n", GetLastError());
        return -1;
    }
//  Creating a raw socket
//
s = WSASocket(AF_INET, SOCK_RAW, IPPROTO_UDP, NULL, 0,0);
if (s == INVALID_SOCKET)
{
    printf("WSASocket() failed: %d\n", WSAGetLastError());
    return -1;
}

// Enable the IP header include option
//
bOpt = TRUE;
ret = setsockopt(s, IPPROTO_IP, IP_HDRINCL, (char *)&bOpt,
   sizeof(bOpt));
if (ret == SOCKET_ERROR)
{
    printf("setsockopt(IP_HDRINCL) failed: %d\n", WSAGetLastError());
    return -1;
}
// Initalize the IP header
//
iTotalSize = sizeof(ipHdr) + sizeof(udpHdr) + strlen(strMessage);

iIPVersion = 4;
iIPSize = sizeof(ipHdr) / sizeof(unsigned long);
//
// IP version goes in the high-order 4 bits of ip_verlen. The
// IP header length (in 32-bit words) goes in the lower 4 bits.
//
```

```
ipHdr.ip_verlen = (iIPVersion << 4) | iIPSize;
ipHdr.ip_tos = 0;                          // IP type of service
ipHdr.ip_totallength = htons(iTotalSize); // Total packet len
ipHdr.ip_id = 0;                  // Unique identifier: set to 0
ipHdr.ip_offset = 0;              // Fragment offset field
ipHdr.ip_ttl = 128;               // Time to live
ipHdr.ip_protocol = 0x11;         // Protocol(UDP)
ipHdr.ip_checksum = 0 ;           // IP checksum
ipHdr.ip_srcaddr = dwFromIP;      // Source address
ipHdr.ip_destaddr = dwToIP;       // Destination address
//
// Initalize the UDP header
//
iUdpSize = sizeof(udpHdr) + strlen(strMessage);

udpHdr.src_portno = htons(iFromPort) ;
udpHdr.dst_portno = htons(iToPort) ;
udpHdr.udp_length = htons(iUdpSize) ;
udpHdr.udp_checksum = 0 ;
//
// Build the UDP pseudo-header for calculating the UDP checksum.
// The pseudo-header consists of the 32-bit source IP address,
// the 32-bit destination IP address, a zero byte, the 8-bit
// IP protocol field, the 16-bit UDP length, and the UDP
// header itself along with its data (padded with a 0 if
// the data is odd length).
//
iUdpChecksumSize = 0;
ptr = buf;
ZeroMemory(buf, MAX_PACKET);

memcpy(ptr, &ipHdr.ip_srcaddr,  sizeof(ipHdr.ip_srcaddr));
ptr += sizeof(ipHdr.ip_srcaddr);
iUdpChecksumSize += sizeof(ipHdr.ip_srcaddr);

memcpy(ptr, &ipHdr.ip_destaddr, sizeof(ipHdr.ip_destaddr));
ptr += sizeof(ipHdr.ip_destaddr);
iUdpChecksumSize += sizeof(ipHdr.ip_destaddr);

ptr++;
iUdpChecksumSize += 1;

memcpy(ptr, &ipHdr.ip_protocol, sizeof(ipHdr.ip_protocol));
ptr += sizeof(ipHdr.ip_protocol);
iUdpChecksumSize += sizeof(ipHdr.ip_protocol);

memcpy(ptr, &udpHdr.udp_length, sizeof(udpHdr.udp_length));
ptr += sizeof(udpHdr.udp_length);
iUdpChecksumSize += sizeof(udpHdr.udp_length);

memcpy(ptr, &udpHdr, sizeof(udpHdr));
ptr += sizeof(udpHdr);
iUdpChecksumSize += sizeof(udpHdr);

for(i = 0; i < strlen(strMessage); i++, ptr++)
    *ptr = strMessage[i];
iUdpChecksumSize += strlen(strMessage);
```

```
    cksum = checksum((USHORT *)buf, iUdpChecksumSize);
    udpHdr.udp_checksum = cksum;
    //
    // Now assemble the IP and UDP headers along with the data
    // so we can send it
    //
    ZeroMemory(buf, MAX_PACKET);
    ptr = buf;

    memcpy(ptr, &ipHdr, sizeof(ipHdr));   ptr += sizeof(ipHdr);
    memcpy(ptr, &udpHdr, sizeof(udpHdr)); ptr += sizeof(udpHdr);
        memcpy(ptr, strMessage, strlen(strMessage));

        // Apparently, this SOCKADDR_IN structure makes no difference.
        // Whatever we put as the destination IP addr in the IP header
        // is what goes. Specifying a different destination in remote
        // will be ignored.
        //
        remote.sin_family = AF_INET;
        remote.sin_port = htons(iToPort);
        remote.sin_addr.s_addr = dwToIP;

        for(i = 0; i < dwCount; i++)
        {
            ret = sendto(s, buf, iTotalSize, 0, (SOCKADDR *)&remote,
                sizeof(remote));
            if (ret == SOCKET_ERROR)
            {
                printf("sendto() failed: %d\n", WSAGetLastError());
                break;
            }
            else
                printf("sent %d bytes\n", ret);
        }
        closesocket(s) ;
        WSACleanup() ;

        return 0;
    }
```

建好套接字，并设置了IP_HDRINCL选项之后，代码会开始IP头的填写。大家可注意到，代码将IP头声明为一个结构：IP_HDR。注意头两个4位字段已合并成单独一个字段，因为编译程序只能在最小1个字节的边界上对齐字段。正是由于这个原因，代码必须将 IP版本置于高4位。ip_protocol字段的值设为0x11，它对应于UDP。代码也将ip_srcaddr字段设为源IP地址（或者想让接收者认为的任何地址），并将ip_destaddr字段设为接收者的IP地址。网络堆栈会计算出IP校验和，所以代码中不必另行设置。

下一步是对UDP头进行初始化。这个操作是非常简单的，因为字段的数量本来就不多。设置了源和目标端口号之后，同时还要设置 UDP头的长度。校验和字段设为0。尽管在你编写的代码中，可能并不需要进行 UDP校验和的计算，但这里的例子却那样做了，目的只是为了阐述如何用UDP“伪头”来做这件事情。为使计算简单一些，我们将所有必要的字段都复制到一个临时性的字符缓冲区内：buf。然后，将该缓冲区传递给我们的校验和计算函数，同时

指出缓冲区的长度，完成最终的计算。

发出数据报之前，要做的最后一件事情是将消息的各个部分“组装”到一个邻近的缓冲区内。只需简单地使用memcpy函数，便可将IP、UDP头以及数据复制到一个邻近的缓冲区内。接着调用sendto函数，将数据发送出去。注意在使用 IP_HDRINCL选项的时候，sendto中的to参数会被忽略。这是由于数据无论如何都会发送给我们在 IP头内指定的那个主机！

下面来看看Iphdrinc.c程序实际运行时的情况。此时可使用第 7章讲述的那个UDP接收者应用程序：Receiver.c。例如，可用下述参数来启动UDP接收者应用程序：

```
Receiver.exe -p:5150 -i:xxx.xxx.xxx.xxx -n:5 -b:1000
```

如果机器只安装了一个网络接口（即只对应一个 IP地址），那么可考虑省略-i参数；否则的话，便请指定自己打算使用的那个接口的 IP地址。在Windows 2000机器中，以下述形式启动Iphdrinc.c程序：

```
Iphdrinc.exe -fi:1.2.3.4 -fp:10 -ti:xxx.xxx.xxx.xxx -tp:5150 -n:5150
```

注意，用-ti参数设定的IP地址必须是接收者正在上面监听的同样的一个 IP地址。随后，接收者应用程序应输出下述报告：

```
[1.2.3.4:10] sent me: 'This is a test'
[1.2.3.4:10] sent me: 'This is a test'
[1.2.3.4:10] sent me: 'This is a test'
[1.2.3.4:10] sent me: 'This is a test'
[1.2.3.4:10] sent me: 'This is a test'
```

结束了对 recvfrom的调用之后，Receiver.c除列印出收到的消息之外，还会显示出自SOCKADDR_IN结构返回的地址信息，该结构已传递进入 recvfrom。随后，可检验收到数据包的IP头是否与使用“Microsoft网络监视器”在网上捕捉数据包获得的结果相同。在表 13-4中，我们列出了Iphdrinc示例可选的各个命令行参数。注意可自行设定源和目标的 IP地址/端口编号。

表13-4 Iphdrinc.c参数

参 数	说 明
-fi:xxx.xxx.xxx.xxx	IP包的源IP地址
-fp:整数	IP包的源端口号
-ti:xxx.xxx.xxx.xxx	IP包的目标IP地址
-tp:整数	IP包的目标端口
-n:整数	要发送的UDP数据报的数量
-m:字串	要送出的消息

13.5 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IGMP应用程序。但要注意的是，原始套接字亦适用于其他许多应用，本章不可能全部都能讲到。要想充分发挥原始套接字以及IP_HDRINCL选项的功能，必须对IP协议以及IP内封装的其他所有协议有一个通盘、透彻的理解。

第14章 Winsock 2 服务提供者接口

Winsock 2服务提供者接口（Service Provider Interface, SPI）代表着另一端的Winsock编程（和Winsock 2API相对应）。Winsock的一端是API，另一端则是SPI。自第6章到第13章，我们已对 Winsock 2 API进行了详细讨论。 Winsock 2是围绕着 Windows开放系统架构（Windows Open System Architecture，WOSA）来设计的，WOSA在Winsock和Winsock应用程序之间有一个标准 API；在Winsock和Winsock服务提供者（比如 TCP/IP）之间有一个标准的SPI。图14-1展示了Ws2_32.dll，即Winsock 2支持的动态链接库（DLL）在Winsock应用程序和Winsock服务提供者之间的分布情况。本章详细地讲解了 Winsock 2 SPI。在结束本章的学习时，大家自然便理解如何开发服务提供者，进一步扩展 Winsock 2的能力。

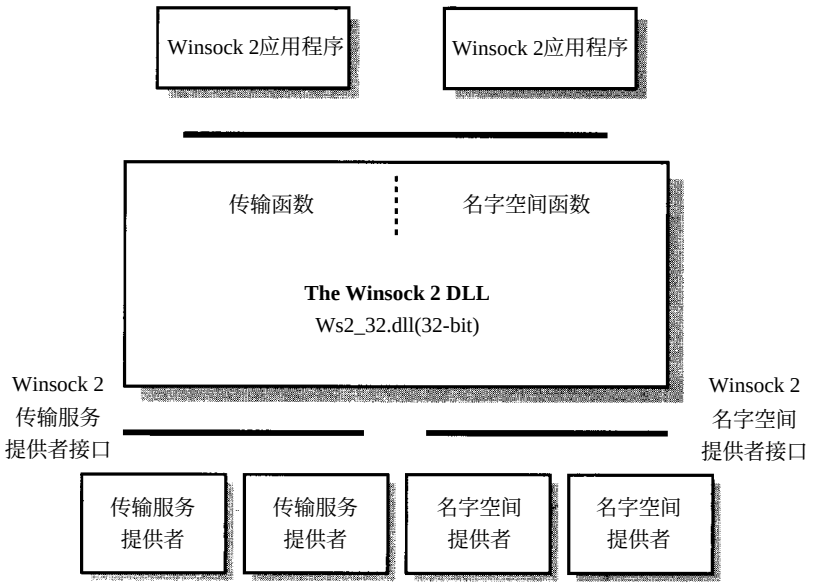

图14-1 Winsock 2的WOSA架构

14.1 SPI基础

Winsock 2 SPI允许开发两类服务提供者 —— 传输提供者和名字空间提供者。“传输提供者”（Transport providers，一般称作协议堆栈，比如 TCP/IP）即能够提供建立通信、传输数据、日常数据流控制和错误控制等等功能的服务。“名字空间提供者”（Name space providers）则把一个网络协议的定址属性和一个或多个用户友好名关联到一起，以便启用与协议无关的名字解析方案。服务提供者不外乎就是 Win32支持的DLL，挂靠在Winsock 2的Ws2_32.dll模块下。对Winsock2 API中定义的许多内部调用来说，这些服务提供者都提供了它们的运作方式。

14.1.1 SPI命名规则

Winsock 2 SPI函数原型采用下面的函数前缀命名规则：

- WSP（Winsock提供者）：用于传送服务提供者函数。
- NSP（名字空间提供者）：用于名字空间提供者函数。
- WPU（Winsock提供者上调）：供服务提供者调用的Ws2_32.dll支持函数使用。
- WSC（Winsock配置）：供在Winsock 2中安装服务提供者的函数使用。

举个例子来说，一个名为WSAInstallProvider的函数，仅仅是一个SPI配置函数。

14.1.2 Winsock 2 API和SPI函数之间的映射

多数情况下，一个应用程序在调用 Winsock 2函数时，Ws2_32.dll会调用相应的Winsock SPI函数，利用特定的服务提供者执行所请求的服务。举个例子来说， select对应WSPSelect，WSAConnect对应WSPConnect，而WSAAccept则对应WSPAccept。然而，并非所有的Winsock函数都有对应的SPI函数。下面列出了这些例外情况。

- htonl、htons、ntohl和ntohs这一类的支持函数在Ws2_32.dll内部执行，没向下传给服务提供者。这一点对这些函数的WSA版本来说，也不例外。
- 像inet_addr和inet_ntoa这样的IP转换函数只能在Ws2_32.dll内执行。
- Winsock 1.1中，所有由IP决定的名字转换和解析函数（比如getXbyY、WSAAsynGetXByY、WSACancelAsyncRequest以及gethostname）都在Ws2_32.dll内部执行。
- Winsock服务提供者列举和与锁定挂钩相关的函数都在 Ws2_32.dll内部执行。因此，WSAEnumProtocols、WSAIsBlocking、WSASetBlockingHook和WSAUnhookBlocking没有相应的SPI函数。
- Winsock错误代码的管理在Ws2_32.dll内部进行。而SPI中，不需要WSAGetLastEror和WSASetLastError。
- 事件对象处理和等待函数，其中包括WSACreateEvent、WSACloseEvent、WSASetEvent、WSAResetEvent和WSAWaitForMultipleEvents，这些函数直接映射成原始的 Win32操作系统调用，没有出现在SPI中。

现在，大家准备学习哪些 Winsock API映射成Winsock 2服务提供者。在开发服务提供者时，大家将看到定义在包容文件 Ws2spi.h中的所有SPI函数原型。

14.2 传输服务提供者

Winsock 2中使用的传输服务提供者有两类：基础服务提供者和分层服务提供者。基础服务提供者执行网络传输协议（比如 TCP/IP）的具体细节其中包括在网络上收发数据之类的核心网络协议功能。“ 分层式 ”（Layered）服务提供者只负责执行高级的自定义通信功能，并依靠下面基础服务提供者，在网络上进行真正的数据交换。举个例子来说，你可以在现成的基础TCP/IP提供者上执行一个数据安全管理者或带宽管理者。图 14-2展示了如何在一个Ws2_32.dll和一个基础提供者之间安装一个或多个分层式提供者。

本小节的重点在于如何开发分层式传输服务提供者。如果你此时正在开发一个基础服务提供者，就可以运用这里的规则。我们具体讲解如何从头执行特定 SPI函数。举个例子来说，我们不提供WSPSend SPI函数是如何把数据写入网络适配器这类的细节。相反，我们展示的

则是分层式提供者的WSPSend函数将如何调用低级提供者的 WSPSend函数，这是多数分层式服务提供者的一项基本要求。从本质上说，开发一个分层式服务提供者所涉及的许多操作都依赖于你的提供者对你的下一层提供者进行的 SPI调用来完成。容易出错的环节是通过Winsock I/O模型，怎样对I/O调用（参见第8章）进行处理，至于这一点，我们稍后将继续讨论。本书的配套光盘上，我们提供了一个示例，名为 LSP，向大家演示了如何执行分层式协议提供者，该提供者只计下了一个套接字上利用 IP传输协议传输了多少字节。微软平台 SDK特别提供了一个更高级的分层式协议提供者示例，名为 layered（分层式），可在MSDN平台SDK示例中找到它。至于 MSDN平台SDK示例则在 ftp://ftp.microsoft.com/bussys/WinSock/winsock2/layered.zip下载。

注意 上面我们常提到这些术语“ SPI客户机 ”和“ 低级别提供者 ”。SPI客户机既可以是Winsock 2的Ws2_32.dll，又可以是位于你的服务提供者之上的另一个分层式服务提供者。SPI客户机绝不可能是Winsock应用程序本身，因为Winsock应用程序必须使用Ws2_32.dll中导出的Winsock 2 API。“ 低级别提供者 ”这个术语只能在描述分层式服务提供者的开始时使用。低级别提供者既可以是另查个分层式服务提供者，又可以是基础服务器提供者。正如大家即将看到的那样，一台机器上可以安装若干个分层式服务提供者；因此，也可以在你的提供者下面，再安装一个分层式服务提供者。

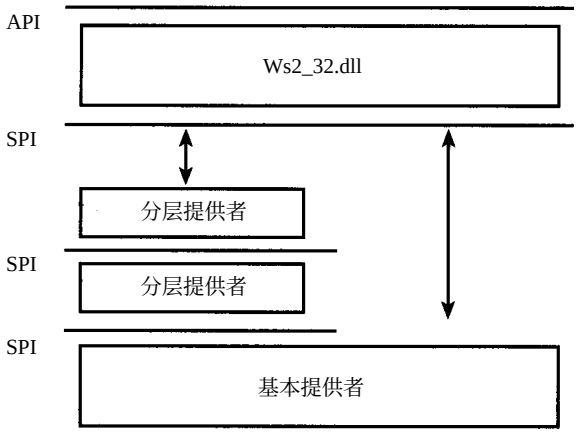

图14-2 分层提供者的架构

14.2.1 WSPStartup

Winsock 2传输服务提供者随标准的Windows动态链接库模块一起执行，你必须把 DllMain函数导入这个动态链接库模块中。除此以外，还必须导入一个名为 WSPStartup的单一函数条目。在调用者（如SPI客户机）调用WSPStartup时，便通过一个被当作参数投送的函数派遣表打开另外的30个SPI函数，传输服务提供者便由这 30个函数组成。函数派遣表参见表 14-1。你的服务提供者必须提供对WSPStartup函数和其他30个函数的支持。

何时以及怎样调用 WSPStarup呢？这个问题尤其重要。也许应用程序调用 WSAStartup API时，你也在考虑调用WSPStartup。这里的情况有所不同。调用 WSAStartup期间，Winsock不知道需要使用哪种服务提供者的类型。 Winsock根据WSASocket调用的地址家族、套接字类

型和协议参数，决定需要加载哪个服务提供者。只有在一个应用程序通过 socket或WSASocket API调用建立一个套接字时， Winsock才会调用一个服务提供者。举个例子来说，如果一个应用程序建立了一个采用地址家族 AF_INET、套接字类型SOCK_STREAM的套接字， Winsock就搜索并加载与之相应的、能够提供 TCP/IP能力的传输提供者。我们将在本章的 14.2.7节“传输服务提供者的安装”就此进行深入讨论。

表14-1 传输服务提供者的支持函数

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
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

14.2.2 参数

在初始化传输服务提供者时， 关键性的函数便是 WSAStartup。它的定义如下：

```
int WSPStartup(
    WORD wVersionRequested,
    LPWSPDATAW lpWSPData,
    LPWSAPROTOCOL_INFOW lpProtocolInfo,
    WSPUPCALLTABLE UpcallTable,
    LPWSPPROC_TABLE lpProcTable
);
```

wVersionRequested参数用于取得调用者能够使用的 Windows Sockets SPI支持函数的最新版本。你的服务提供者应该对这个值进行检查，以便得知它是否支持所请求的版本。你的提供者通过一个WSPDATA结构，利用lpWSPData参数返回关于它自己的版本信息。 WSPDATA结构的格式如下：

```
typedef struct WSPData
{
    WORD        wVersion;
    WORD        wHighVersion;
    WCHAR       szDescription[WSPDESCRIPTION_LEN + 1];
} WSPDATA, FAR * LPWSPDATA;
```

wVersion参数中，协议提供者必须返回调用者希望采用的 Winsock版本号。wHighVersion参数必须返回提供者支持的 Winsock的最高版本。这个值一般和 wVersion的值相同（关于Winsock版本信息，详见第 7章）。szDescription字段返回一个空中止 UNICODE字串，该字串把你的提供者视作SPI客户机。这个字段最多可容纳256个字符。

WSPStartup的lpProtocolInfo参数是一个指针，指向一个 WSAPROTOCOL_INFOW结构。该结构中包含了和提供者有关的特征信息（协议特征和这个结构的详细情况，参见第 5章“ Winsock协议信息 ”）。WSAPROTOCOL_INFOW结构中的信息是Ws2.32_dll从Winsock 2服务提供者目录中检索得来的。这个目录中包含了服务提供者的属性信息。我们将在“ 传输服务提供者的安装 ”这一小节，对 Winsock 2目录条目做进一步的探讨。

在开发分层式服务提供者时，需要以一种独特的方式来处理 lpProtocolInfo这个参数，因为它包含的信息关系到你的服务提供者在 Ws2_32.dll和基础服务提供者之间，是怎样分布的。这个参数用于决定你的服务提供者之下的那个服务提供者（既可以是另一个分层提供者，又可以是一个基础提供者）。从某个角度来说，你的提供者必须通过加载下一个提供者的 DLL模块和调用它的 WSPStartup函数这一方式，来加载下一个提供者。 lpProtocolInfo指向的WSAPROTOCOL_INFOW结构中包含一个字段ProtocolChain，该字段标志你的服务提供者和机器上的其他提供者是如何排序的。

ProtocolChain字段实际上是一个WSAPROTOCOLCHAIN结构，它的格式如下：

```
typedef struct _WSAPROTOCOLCHAIN
{
    int ChainLen;
    DWORD ChainEntries[MAX_PROTOCOL_CHAIN];
} WSAPROTOCOLCHAIN, FAR * LPWSAPROTOCOLCHAIN;
```

ChainLen字段标志Ws2_32.dll和基础的服务提供者之间，重叠了多少层（这个数包含基础提供者在内）。如果你有一台这样的机器，在一个协议之上只有一个重叠服务提供者，比如说TCP/IP，那么这个字段就应该是 2。ChainEntries字段是一数组，该数组由服务提供者目录标志号组成，这些编号唯一性地标志出重叠服务提供者，这些提供者因为某一特定协议而链接到一起。我们将在 14.2.7节“ 传输服务提供者的安装 ”中详细说明 WSAPROTOCOLCHAIN结构。分层式服务提供者的一项要求是搜索 ProtocolChain字段，以便决定它自己在服务提供者数组中所处的位置（通过搜索层目录条目这一方式），以及确定数组中的下一个提供者。如果下一个提供者是另一层，就必须把未经修改的 lpProtocolInfo结构投给下一层的WSPStartup函数。如果下一层提供者是本数组中的最后一位（即基础提供者），你的提供者就必须在调用基础提供者的WSPStartup函数之前，利用这个基础提供者的 WSAPROTOCOL_INFOW结构，在

lpProtocolInfo结构上执行交换。程序清单 14-1演示了分层提供者应该怎样通过编程来管理lpProtocolInfo结构。

程序清单14-1 为WSPStartup寻找相应的WSAPROTOCOL_INFOW结构

```
LPWSAPROTOCOL_INFOW ProtocolInfo;
LPWSAPROTOCOL_INFOW ProtoInfo = lpProtocolInfo;
DWORD ProtocolInfoSize = 0;
// Find out how many entries we need to enumerate
if (WSCEnumProtocols(NULL, ProtocolInfo, &ProtocolInfoSize,
    &ErrorCode) == SOCKET_ERROR)
{
    if (ErrorCode != WSAENOBUFS)
    {
        return  WSAEPROVIDERFAILEDINIT;
    }
}

if ((ProtocolInfo = (LPWSAPROTOCOL_INFOW) GlobalAlloc(GPTR,
    ProtocolInfoSize)) == NULL)
{
    return  WSAEPROVIDERFAILEDINIT;
}

if ((TotalProtocols = WSCEnumProtocols(NULL, ProtocolInfo,
    &ProtocolInfoSize, &ErrorCode)) == SOCKET_ERROR)
{
    return  WSAEPROVIDERFAILEDINIT;
}

// Find our layered provider's catalog ID entry
for (i = 0; i < TotalProtocols; i++)
    if (memcmp (&ProtocolInfo[i].ProviderId, &ProviderGuid,
        sizeof (GUID))==0)
    {
        gLayerCatId = ProtocolInfo[i].dwCatalogEntryId;
        break;
    }

// Save our provider's catalog ID entry
gChainId = lpProtocolInfo->dwCatalogEntryId;

// Find our catalog ID entry in the protocol chain
for(j = 0; j < lpProtocolInfo->ProtocolChain.ChainLen; j++)
{
    if (lpProtocolInfo->ProtocolChain.ChainEntries[j] ==
        gLayerCatId)
    {
        NextProviderCatId =
        lpProtocolInfo->ProtocolChain.ChainEntries[j+1];

        // Check whether next provider is the base provider
        if (lpProtocolInfo->ProtocolChain.ChainLen ==
            (j + 2))
        {
            for (i = 0; i < TotalProtocols; i++)
                if (NextProviderCatId ==
```

```
                    ProtocolInfo[i].dwCatalogEntryId)
                {
                    ProtoInfo = &ProtocolInfo[i];
                    break;
                }
            }
            break;
        }
    }

    // At this point, ProtoInfo will contain the appropriate
    // WSAPROTOCOL_INFOW structure
```

WSPStartup的UpcallTable取得Ws2_32.dll的SPI上调派遣表，表中包含了指向许多支持函数的指针，你的提供者可利用这些函数来管理提供者自身和 Winsock 2之间的I/O操作。本章的“ Winsock I/O模型支持 ”小节中，我们还定义许多此类的函数，并对它们的用法进行了描述。

WSPStartup的最后一个参数是lpProcTable，代表一个表，表内有30个SPI函数指针，你的服务提供者必须能够提供对这些函数的支持。这 30个函数列在表14-1中。每个SPI函数都遵照自己对应的API函数的参数说明。

- 每个函数都提供一个终极参数， lpErrno，在实施失败时，你的提供者必须用这个参数报告具体的Winsock错误代码。举个例子来说，如果你在实施 WSPSend，但不能为它分配内存，可能就会返回WSAENOBUF错误代码。
- SPI函数WSPSend、WSPSendTo、WSPRecv、WSPRecvFrom以及WSPIoctl，都突出了这个额外的参数：lpThreadId，它标志调用了SPI函数的应用程序线程。稍后，我们将看到，这个特性对支持完成例程时，发挥了很大的作用。

最后一个需要注意的是几个 Winsock 1.1函数，比如 send和recv，直接映射成相应的Winsock 2函数。我们之所以说这些 Winsock 1.1间接映射成相应的SPI函数，是因为它们事实上调用了提供类似功能的 Winsock 2函数。比如说， send函数实际上调用的是WSASend函数，WSASend函数的映射函数是WSPSend。表14-1中，我们特别注明了间接映射成 SPI函数的API函数。

14.2.3 实例计数

在Winsock规格中，应用程序调用 WSAStartup和WSACleanup函数的次数是没有限制的。你的服务提供者的WSPStartup和WSPCleanup函数也将和它们对应的 API函数一样，调用的次数相当。如此一来，你的服务提供者就应该保留一个实例计数，以便了解 WSPStartup函数被调用了多少次，再相应地调用 WSPCleanup多少次，以便抵消WSPStartup被调用的次数。保留实例次数的目的在于：允许你的服务提供者的启用和清除过程合乎情理。举个例子来说，只要你的实例计数大于 0，你的提供者就可以一直保存在内存中。当实例计数落到 0以下时，Ws2_32.dll最终从相应的内存中卸载你的提供者。

14.2.4 套接字句柄

当SPI客户机调用WSPSocket、WSPAccept和WSPJointLeaf函数时，服务提供者必须返回套接字句柄。返回 SPI客户机的套接字句柄既可以是可安装文件系统（ IFS）句柄，又可以是

非IFS句柄。如果一个服务提供者返回的是 IFS句柄，就会被视作 IFS提供者；反之，就是非IFS提供者。微软的基础传输提供者全都是 IFS句柄。

设计Winsock的目的是允许Winsock应用程序利用Win32 API函数ReadFile和WriteFile，在套接字句柄上收发数据。因此，必须考虑到如何在一个服务提供者内建立套接字句柄。如果在开发服务提供者的同时，还试图为套接字句柄上调用 ReadFile和WriteFile的Winsock应用程序提供服务，就要考虑另行开发一个 IFS提供者。但之前，必须先了解I/O的某些限制。

1. IFS 提供者

我们在前面曾提过，传输服务提供者可以是分层的或基础的。如果正在开发一个基础的IFS提供者，你的提供者就会有一个核心模式的操作系统组件，而这个组件启用了 Winsock提供者来建立句柄，与ReadFile和WriteFile函数中的文件系统句柄相同。核心模式软件的开发不在本书讨论之列。如果想了解如何开发核心模式操作系统组件这方面的详情，可参考 MSDN设备开发工具包（Device Development Kit，DDK）。

分层式服务提供者也可变成 IFS提供者，但前提是它必须位于现成的基础 IFS提供者顶部。这涉及到把低级IFS提供者的套接字句柄(在你的分层提供者中检索得来的)直接上传到你的SPI客户机。从一个低级提供者直接上传套接字句柄会限制了分层提供者的能力，主要表现在以下两方面：

- 如果在一个套接字上调用了 ReadFile和WriteFile，便不会调用分层提供者的 WSPSend和WSPRecv函数。这些函数将绕过分层提供者，直接调用基础 IFS提供者的实施。
- 分层提供者将不能后处理提交到一个完成端口的重叠 I/O请求。完成端口的后处理完成绕过了分层式提供者。

如果你的分层式提供者试图对通过这个提供者传递的所有 I/O操作进行监视， 还必须开发一个非IFS分层式提供者，我们稍后对此进行详细讨论。

只要一个IFS提供者（分层的或基础的）建立了一个新的套接字描述符，就需要要求它调用WPUModifyIFSHandle，而不是提供指向 SPI客户机的新句柄。 Win32API函数（比如ReadFile和WriteFile）在一个套接字上执行I/O时，这样便可使Winsock Ws2_32.dll合理地标识与指定套接字关联到一起的IFS服务提供者进程。 WPUModifyIFSHandle的定义如下：

```
SOCKET WPUModifyIFSHandle(
    DWORD dwCatalogEntryId,
    SOCKET ProposedHandle,
    LPINT lpErrno
);
```

dwCatalogEntryId参数标识你的服务提供者的目录ID。ProposedHandle参数代表一个IFS句柄，该句柄是你的服务提供者分配的（在基础提供者的情况下）。如果此时正在开发分层式IFS提供者，这个句柄就会从低级提供者上传。如果这个函数失败，返回错误 INVALID_SOCKET，lpErrno参数便取得特定的Winsock错误代码信息。

2. 非IFS提供者

如果你此时正在开发分层式提供者，并试图对一个套接字上发生的所有读取和写入操作进行监视，还必须开发一个 IFS提供者。非IFS提供者使用上调函数WPUCreateSocketHandle建立套接字句柄。 WPUCreateSocketHandle建立的套接字句柄类似于 IFS提供者句柄，两者均允许Winsock应用程序在套接字上使用 ReadFile和WriteFile函数。但是，这两个函数对 I/O性能有

一个非常明显的不利影响，因为 Winsock 2架构必须分别执行到服务提供者的 WSPRecv和WSPSend函数的重定向I/O操作。WPUCreateSocketHandle的定义如下：

```
SOCKET WPUCreateSocketHandle(
    DWORD dwCatalogEntryId,
    DWORD dwContext,
    LPINT lpErrno
);
```

dwCatalogEntryId标识你的服务提供者的目录 ID。dwContext参数允许你自由地把提供者的数据和一个套接字描述符关联到一起。在本书配套光盘上的 LSP示例中，我们利用了这个字段保存了收发数据的字节数。 Winsock提供了一个上调函数 WPUQuerySocketHandleContext，该函数用于取得保存在dwContext中关联的套接字提供者数据，它的定义如下：

```
int WPUQuerySocketHandleContext(
    SOCKET s,
    LPDWORD lpContext,
    LPINT lpErrno
);
```

s参数代表套接字句柄，该句柄从一个 SPI客户机（最初通过WPUCreateSocketHandle建立的）开始下传，并且你打算用这个套接字句柄取得套接字提供者的数据。 lpContext参数取得最初投入WPUCreateSocketHandle中的提供者数据。和前一个函数中的 lpErrno参数一样，若函数调用失败，就会收到相关的 Winsock错误代码。如果WPUCreateSocketHandle失败，便返回INVALID_SOCKET；若WPUQuerySocketHandleContext失败，则返回SOCKET_ERROR。

14.2.5 Winsock I/O模型支持

通过第8章的学习，大家已经知道Winsock的特别突出了几个I/O模型，这些模型可以用来管理套接字上的 I/O操作。从服务提供者角度来看，每个模型都要求使用某些 SPI上调函数，这些函数是Ws2_32.dll提供的，通过前面提到的 WSPStartup中的UpcallTable参数使用。如果此时正在开发一个简单的 IFS分层式提供者，下面几个小节中讲的 I/O模式遵守的原则就不适用，因为低级提供者便足以应付对各个模式的 I/O操作的管理了。这些原则只适用于其他类的提供者。我们将重点讨论如何开发非 IFS分层式服务提供者。

1. 锁定和非锁定模型

Winsock 2中，锁定I/O是最简单的。锁定套接字的任何一个 I/O操作都不会在操作出完成之前返回。因此，所有线程一次只能执行一个 I/O操作。举个例子来说，一个 SPI客户机以锁定方式调用 WSPRecv函数时，你的提供者只需要把这个调用直接传给下一个提供者的WSPRecv调用。对你的提供者而言，它的 WSPRecv函数只能在下一个提供者的 WSPRecv函数完成时，才能够返回。

尽管锁定I/O模型非常易于实现，但仍然需要考虑向后兼容Winsock 1.1锁定挂钩这一问题。Winsock 2 API规格中已经删除了WSASetBlockingCall和WSACancelBlocking API调用。但是，一个Winsock 1.1应用程序在调用 WSASetBlockingHook和WSACancelBlockingCall时，Ws2_32.dll仍然可以调用 WSPCancelBlockingHook。在分层式服务提供者中，只需要把WSPCancelBlockingHook调用传给基础提供者的相应调用即可。如果此时正在执行一个基础提供者和一个锁定调用，就必须执行周期性地调用 WPUQueryBlockingCallback函数这一机制。

WPUQueryBlockingCallback的定义如下：

```
int WPUQueryBlockingCallback(
    DWORD dwCatalogEntryId,
    LPBLOCKINGCALLBACK FAR *lplpfnCallback,
    LPDWORD lpdwContext,
    LPINT lpErrno
);
```

像我们在介绍WSPStartup函数时描述的那样，dwCatalogEntryId参数取得你的提供者的目录ID。lplpfnCallback参数是一个函数指针，指向应用程序的锁定挂钩函数，你必须周期性地调用这个函数（即回调函数），以避免该应用程序的锁定调用真正地被锁定。这个回调函数的形式如下：

```
typedef BOOL (CALLBACK FAR * LPBLOCKINGCALLBACK)(
    DWORD dwContext
);
```

你的提供者周期性调用 lplpfnCallback时，便把lpdwContext参数的值投给该回调函数的dwContext参数。WPUQueryBlockingCallback函数的最后一个参数是lpErrno。若该函数返回SOCKET_ERROR，这个参数便返回相应的Winsock错误代码信息。

2. select模型

select I/O模型要求提供者为 WSPSelect函数中的这几个参数（ readfds、writefds和exceptfds）管理fd_set结构。WSPSelect函数的定义如下：

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

从本质上来说，fd_set数据类型代表一个套接字集合。一个 SPI客户机利用WSPSelect调用你的提供者时，便把套接字句柄投给这个集合中的一个或多个套接字。你的提供者负责判断列出的套接字上何时发生了网络活动。

对非IFS分层提供者来说，便要求建立三个 fd_set数据字段，并将SPI客户机的套接字句柄映射到这个集合中的低层提供者的套接字句柄。一旦建立了所有的套接字集合，你的提供者便调用低层提供者上的 WSPSelect。低层提供者的调用完成时，你的提供者必须判断各个fd_set字段中，哪些套接字上有待发事件。此时，便可以用这个非常有用的上调函数WPUFDIsSet，来判断设置了哪些基础提供者套接字。这个上调函数类似于 FD_ISSET宏（参见第8章），它的定义如下：

```
int WPUFDIsSet(
    SOCKET s,
    FD_SET FAR *set
);
```

s参数代表你正在套接字集合中查找的套接字。 set参数是一个真正的套接字描述符集合。由于你的提供者将检查投向低层提供者的每个描述符集合中的内容，因此，你的提供者应该保存上层提供者到低层提供者之间的套接字映射。低层提供者完成 WSPSelect调用时，就可轻

松地向上层提供者反映哪些套接字上有I/O待发事件了。

一旦决定了哪些低层提供者套接字有待发的网络事件，就必须更新原来的 fd_set集合，这些集合是原来的SPI客户机投递的。 Ws2spi.h文件中提供了三个宏（ FD_CLR、FD_SET和FD_ZERO），它们可用于对原来的套接字集合进行管理。关于这些宏的说明，参见第 8章。程序清单14-2演示了怎样实施WSPSelect。

程序清单14-2 WSPSelect实施详解

```
int WSPAPI WSPSelect(
    int nfds,
    fd_set FAR * readfds,
    fd_set FAR * writefds,
    fd_set FAR * exceptfds,
    const struct timeval FAR * timeout,
    LPINT lpErrno)
{
    SOCK_INFO *SocketContext;
    u_int i;
    u_int count;
    int Ret;
    int HandleCount;

    // Build an Upper and Lower provider socket mapping table
    struct
    {
        SOCKET ClientSocket;
        SOCKET ProvSocket;
    } Read[FD_SETSIZE], Write[FD_SETSIZE], Except[FD_SETSIZE];

    fd_set ReadFds, WriteFds, ExceptFds;

    // Build the ReadFds set for the lower provider
    if (readfds)
    {
        FD_ZERO(&ReadFds);

        for (i = 0; i < readfds->fd_count; i++)
        {
            if (MainUpCallTable.lpWPUQuerySocketHandleContext(
                (Read[i].ClientSocket = readfds->fd_array[i]),
                (LPDWORD) &SocketContext, lpErrno) ==
                SOCKET_ERROR)
                return SOCKET_ERROR;
            FD_SET((Read[i].ProvSocket =
                SocketContext->ProviderSocket), &ReadFds);
        }
    }

    // Build the WriteFds set for the lower provider.
    // This is just like the ReadFds set above.
    ...

    // Build the ExceptFds set for the lower provider.
    // This is also like the ReadFds set above.
    ...
```

```
        // Call the lower provider's WSPSelect
        Ret = NextProcTable.lpWSPSelect(nfds,
            (readfds ? &ReadFds : NULL),
            (writefds ? &WriteFds : NULL),
            (exceptfds ? &ExceptFds : NULL), timeout, lpErrno);

        if (Ret != SOCKET_ERROR)
        {
            HandleCount = Ret;

            // Set up calling provider's readfds set
            if (readfds)
            {
                count = readfds->fd_count;
                FD_ZERO(readfds);

                for(i = 0; (i < count) && HandleCount; i++)
                {
                    if (MainUpCallTable.lpWPUFDIsSet(
                        Read[i].ProvSocket, &ReadFds))
                    {
                        FD_SET(Read[i].ClientSocket, readfds);
                        HandleCount--;
                    }
                }
            }

            // Set up calling provider's writefds set.
            // This is just like the readfds set above.
            ...
            // Set up calling provider's exceptfds set.
            // This is also like the readfds set above.
            ...
        }

        return Ret;
    }
```

3. WSAAsyncSelect模型

WSAAsyncSelect I/O模型涉及到的管理中，包括对套接字上网络事件的Windows消息通知进行管理。SPI客户机调用WSPAsyncSelect函数后，便可使用这个模型了。这个函数的定义如下：

```
int WSPAsyncSelect(
    SOCKET s,
    HWND hWnd,
    unsigned int wMsg,
    long lEvent,
    LPINT lpErrno
);
```

s参数代表SPI客户机的套接字，该套接字希望收到网络事件的窗口消息通知。 hWnd参数标志窗口句柄，在套接字s上发生IEvent参数中定义的网络事件时，这个窗口句柄便接收 wMsg参数中定义的消息。如果在执行这个函数时，返回了 SOCKET_ERROR，lpErrno参数便会收到一个Winsock错误代码。至于你的提供者必须支持哪些网络事件（ IEvent参数中定义的），

详情参见表8-3。

SPI客户机在调用WSPAsyncSelect时，你的提供者便利用客户机提供的窗口句柄和客户机提供的窗口消息（通过 WSPAsyncSelect调用投递的），负责通知SPI客户机，套接字上什么时候发生了网络事件。你的提供者调用 WPUPostMessage后，便可向SPI客户机通知网络事件了。WPUPostMessage的定义如下：

```
BOOL WPUPostMessage(
    HWND hWnd,
    UINT Msg,
    WPARAM wParam,
    LPARAM lParam
);
```

你的提供者利用hWnd参数标识即将通知的SPI客户机的窗口句柄。 Msg参数标识用户自定义的消息，该消息被投递到 WSPAsyncSelect的wMsg参数中。 wParam参数则接受发生网络事件的那个套接字句柄。最后一个参数是lParam，它接受两段信息。lParam的低位字（low word）接受的是已经发生的网络事件。在谈到 lParam低位字中标志的网络事件时，如果你的提供者实施中发生了错误，lParam的高位字（highword）则接受该错误的代码。

如果此时正在开发一个非 IFS分层式服务提供者，你的提供者就成为低层提供者的WSPAsyncSelect函数的一个客户机。这样一来，就需要你的提供者在调用 WSPAsyncSelect函数之前，利用WPUQuerySocketHandleContext，把一个SPI客户机的套接字句柄转换成你的提供者的套接字句柄。由于需要服务提供者利用 WPUPostMessage，向SPI客户机通知网络事件。你的提供者必须从低层提供者着手，拦截这些窗口信息。为什么呢？因为 WPUPostMessage将低层提供者的套接字句柄投回了 lParam的高位字里面。其结果是，你的提供者必须把 lParam中的高位字中的套接字句柄翻译为 SPI客户机的套接字句柄，这个句柄用在原来的WSPAsyncSelect调用中。

要想对低层提供者发出的窗口消息拦截进行管理，最好是建立一个“雇员”（worker）线程。这个线程将建立一个隐藏的窗口，对网络事件窗口消息进行管理。当你的提供者在低层套接字上调用WSPAsyncSelect时，你只须把“雇员”窗口句柄投给低层提供者的WSPAsyncSelect调用即可。从此，你的提供者的“雇员”窗口便会收到低层提供者发出的网络事件窗口，你的提供者便可利用WPUPostMessage，向SPI通知网络事件了。

4. WSAEventSelect模型

WSAEventSelect模型涉及套接字上发生网络事件时，怎样通知事件对象这方面的问题。调用WSPEventSelect函数之后，SPI客户机便可使用这个模型了。该函数的定义如下：

```
int WSPEventSelect(
    SOCKET s,
    WSAEVENT hEventObject,
    long lNetworkEvents,
    LPINT lpErrno
);
```

s参数代表SPI客户机的套装字，该套接字希望得到底网络事件通知。 hEventObject参数是一个WSAEVENT对象句柄，当套接字s上发生了lNetworkEvents中指定的网络事件时，你的提供者便向你通知这个句柄。 lpErrno参数收到一个Winsock错误代码，前提是实施该函数时，返

回了SOCKET_ERROR。你的提供者必须支持的网络事件（lNetworkEvents参数中标志的）和WSAAsyncSelect I/O模型中指定的一样。

非IFS分层提供者中的WSPEventSelect的实施实际上非常简单。SPI客户机下传一个事件对象，你的提倡者只须利用WPUQuerySocketHandleContext，把这个SPI客户机的套接字句柄翻译成低层提供者的套接字句柄即可。完成套接字句柄的翻译之后，便可利用这个事件对象，直接从这个SPI客户机中调用低层提供者的WSPEventSelect函数。这个SPI客户机的事件对象上发生I/O操作时，低层提供者便直接通知事件对象，以及向SPI客户机通知网络事件已到达。低层提供者向事件对象发出通知时，在事件完成之前，不会向SPI客户机返回套接字句柄。因此，你的提供者就无须为 SPI客户机执法套接字句柄的翻译。这一点与前面提到的WSAAsyncSelect模型不相同，在前一种模型中，因为套接字句柄被投入低层提供者发出的窗口消息中，因此你的提供者必须对一个已完成的网络事件执行套接字句柄的翻译。

如果你此时正在开发一个基础提供者，一个套接字上发生lNetworkEvents中指定的网络事件时，你的提供者便通过客户机提供的事件对象（通过WSPEventSelect调用传递的）负责向SPI客户机发出通知。你的提供者可利用上调函数WPUSetEvent向SPI客户机通知网络事件。该上调函数的定义如下：

```
BOOL WPUSetEvent (
    WSAEVENT hEvent,
    LPINT lpErrno
);
```

hEvent参数代表你的提供者必须通知的事件对象句柄，这个句柄是从WSPEventSelect开始投递的。如果该函数返回FALSE，lpErrno参数便随特定的Winsock错误代码一起返回。

5. OverlappedI/O模型

OverlappedI/O（重叠式I/O）模型要求你的提供者执行一个重叠式I/O管理器，同时为基于对象的事件和基于例程的完成重叠式I/O请求提供服务。下面的SPI函数采用了Winsock中的Win32重叠式I/O：

- WSPSend
- WSPSendTo
- WSPRecv
- WSPRecvFrom
- WSPIoctl

每个函数都突出了三个常见的参数：一个指向一个WSAOVERLAPPED结构的可选性指针、一个指向WSAOVERLAPPED_COMPLETION_ROUTINE函数的可选性指针以及一个指向WSATHREADID结构的指针（该指针标志执行调用的应用程序线程）。

重叠式I/O操作期间，服务提供者和SPI客户机之间是怎样进行通信的呢？其中的关键是WSAOVERLAPPED结构。该结构的格式如下：

```
typedef struct _WSAOVERLAPPED {
    DWORD    Internal;
    DWORD    InternalHigh;
    DWORD    Offset;
    DWORD    OffsetHigh;
    WSAEVENT hEvent;
} WSAOVERLAPPED, FAR * LPWSAOVERLAPPED;
```

服务提供者的重叠式I/O管理器负责管理SPI客户机WSAOVERLAPPED结构的Internal字段。开始重叠式处理时，服务提供者必须把 Internal字段设为WSS_OPERATION_IN_PROGRESS。这是非常重要的，因为如果SPI客户机调用WSPGetOverlappedResult，与此同时，你的提供者也在为一个待决重叠式操作服务，就可以把 WSS_OPERATION_IN_PROGRESS值用于你的提供者实施的WSPGetOverlappedResult调用中，以此判断重叠式操作是否处于处理过程中。

当一个I/O操作结束时，提供者便设置OffsetHigh和Offset字段。OffsetHigh设为该操作返回的结果Winsock错误代码，Offset则应该设为WSPRecv和WSPRecvFrom I/O操作返回的结果标志。在设置了这几个字段之后，提供者便通过一个事件对象或完成例程（根据上面的可选WSAOVERLAPPED结构所用的I/O函数来决定），通知已完成重叠操作请求的SPI客户机。

(1) Event（事件）

在基于事件的重叠式I/O模型中，SPI客户机随内含一个事件对象的WSAOVERLAPPED结构，调用了我们前面提到的一个I/O函数。另外，必须把WSAOVERLAPPED_COMPLETION_ROUTINE设为NULL（空）值。服务提供者负责管理重叠式I/O请求。请求完成之后，提供者必须提醒调用者的已完成的I/O请求线程。这是通过调用上调函数 WPUCompleteOverlapped Request来完成的。该函数的定义如下：

```
WSAEVENT WPUCompleteOverlappedRequest(
    SOCKET s,
    LPWSAOVERLAPPED lpOverlapped,
    DWORD dwError,
    DWORD cbTransferred,
    LPINT lpErrno
);
```

s参数和lpOverlapped参数代表最初的套接字，而WSAOVERLAPPED结构则始发于客户机。提供者应该把dwError参数设为重叠式I/O请求的完成状态，cbTransferred参数设为重叠式操作期间所传输的字节数。最后一个参数是lpErrno，如果这个函数调用返回SOCKET_ERROR的话，它就会报告相应的Winsock错误代码。WPUCompleteOverlapped Request完成时，便在SPI客户机的WSAOVERLAPPED结构中设置两个字段： InternalHigh设为cbTransferred字节数，Internal设为除了WSS_OPERATION_IN_PROGRESS之外的一个值。

在基于事件的重叠式I/O模型中，SPI客户机最终会调用. WSPGetOverlappedResult，以获得重叠式请求完成之后的结果。该函数的定义如下：

```
BOOL WSPGetOverlappedResult (
    SOCKET s,
    LPWSAOVERLAPPED lpOverlapped,
    LPDWORD lpcbTransfer,
    BOOL fWait,
    LPDWORD lpdwFlags,
    LPINT lpErrno
);
```

调用WSPGetOverlappedResult时，服务提供者必须报告原来的重叠式操作请求的操作状态。正如我们前面提到的那样，提供者负责管理 Internal、InternalHigh、Offset和OffsetHigh，这四个字段在SPI客户机的WSAOVERLAPPED结构中。调用WSPGetOverlappedResult时，提供者应该首先检查SPI客户机的WSAOVERLAPPED结构中的Internal字段。如果Internal字端被

设为WSS_OPERATION_IN_PROGRESS，则表示提供者仍然在处理一个重叠式请求。如果WSPGetOverlappedResult的fWait参数被设为TRUE（真），提供者就必须在返回结果之前，在投入SPI客户机的WSAOVERLAPPED结构的事件句柄上等候这个重叠操作结束。如果 fWait参数被设为FALSE（假），提供者就会返回 Winsock错误WSA_IO_INCOMPLETE。等待返回结果之后，这个重叠式操作一旦完成，提供者就应该这样设置 WSPGetOverlappedResult函数的各个参数：

- 把lpcsTransfer设为WSAOVERLAPPED结构中的InternalHigh字段的值，从中得知收发操作所传输的字节是多少。
- 把lpdwFlags设为WSAOVERLAPPED结构中的Offset字段的值，从中得知 WSPRecv或WSPRecvFrom操作的结果标志。
- 把lpErrno设为WSAOVERLAPPED结构中的OffsetHigh字段的值，从中得知结果化的错误代码。

程序清单14-3演示了怎样实施WSPGetOverlappedResult。

程序清单14-3 WSPGetOverlappedResult实施详解

```
BOOL WSPAPI WSPGetOverlappedResult(
    SOCKET s,
    LPWSAOVERLAPPED lpOverlapped,
    LPDWORD lpcbTransfer,
    BOOL fWait,
    LPDWORD lpdwFlags,
    LPINT lpErrno)
{
    DWORD Ret;

    if (lpOverlapped->Internal != WSS_OPERATION_IN_PROGRESS)
    {
        *lpcbTransfer = lpOverlapped->InternalHigh;
        *lpdwFlags = lpOverlapped->Offset;
        *lpErrno = lpOverlapped->OffsetHigh;

        return(lpOverlapped->OffsetHigh == 0 ? TRUE : FALSE);
    }
    else
    {
        if (fWait)
        {
            Ret = WaitForSingleObject(lpOverlapped->hEvent,
                INFINITE);
            if ((Ret == WAIT_OBJECT_0) &&
                (lpOverlapped->Internal !=
                WSS_OPERATION_IN_PROGRESS))
            {
                *lpcbTransfer = lpOverlapped->InternalHigh;
                *lpdwFlags = lpOverlapped->Offset;
                *lpErrno = lpOverlapped->OffsetHigh;

                return(lpOverlapped->OffsetHigh == 0 ? TRUE :
                    FALSE);
            }
            else
```

```
                *lpErrno = WSASYSCALLFAILURE;
            }
            else
                *lpErrno = WSA_IO_INCOMPLETE;
        }

        return FALSE;
    }
```

(2) Completion routine（完成例程）

在基于完成例程的重叠式I/O模型中，SPI客户机随一个WSAOVERLAPPED结构和一个WSAOVERLAPPED_COMPLETION_ROUTINE指针一起，调用了前面提到的I/O函数之一。服务提供者负责管理重叠式 I/O请求。请求完成时，提供者必须利用 Win32异步进程调用（APC）I/O机制，提醒调用者的完成I/O线程注意。APC机制要求调用者线程处于“警觉等待状态”（参见第8章）。服务提供者结束对基于完成例程的重叠式请求提供的服务时，必须提醒SPI客户机注意请求的操作已完成。这是通过调用上调函数 WPUQueueApc来完成的。该函数的定义如下：

```
int WPUQueueApc(
    LPWSATHREADID lpThreadId,
    LPWSAUSERAPC lpfnUserApc,
    DWORD dwContext,
    LPINT lpErrno
);
```

lpThreadId参数代表SPI客户机的WSATHREADID结构，这个结构是从一个提供完成例程的I/O调用开始投递的。lpfnUserApc参数代表一个指向WSAUSERAPC函数的指针，这个函数扮演的是一个中间函数的角色，提供者必须提供这个函数，以便对 SPI客户机的回调。作为一个中间函数，它必须调用SPI客户机的WSAOVERLAPPED_COMPLETION_ROUTINE，这个函数是最初的重叠式调用提供的。WSAUSERAPC中间函数的原型定义如下：

```
typedef void (CALLBACK FAR * LPWSAUSERAPC)(DWORD dwContext);
```

注意，这个函数的定义中只有一个参数——dwContext。SPI客户机调用这个回调函数时，dwContext参数中包含的信息原来是投入WPUQueueApc的dwContext参数中的。从本质上来说，在调用SPI客户机的WSAOVERLAPPED_COMPLETION_ROUTINE时，dwContext参数允许你投递一个数据结构，这个结构中包含自己可能需要的信息元素。它在第 8章中的定义是这样的：

```
void CALLBACK CompletionROUTINE(
    IN DWORD dwError,
    IN DWORD cbTransferred,
    IN LPWSAOVERLAPPED lpOverlapped,
    IN DWORD dwFlags
);
```

服务提供者应该通过WPUQueueApc的dwContext参数，投递下列信息：

- 当作Winsock错误代码的重叠式操作的状态。
- 通过重叠式操作传输的字节数是多少。
- 调用者的WSAOVERLAPPED结构。

■ 调用者投入I/O调用中的标志。

有了这些信息，提供者便可成功地通过中间完成例程，调用SPI客户机的WSAOVERLAPPED_COMPLETION_ROUTINE。

(3) Completion ports（完成端口）

在Winsock 2中，完成端口I/O模型的实施是在Ws2_32.dll模块中进行的。正如我们在第 8章中所讲的那样，完成端口模型是建立在使用基于事件的重叠式 I/O模型基础上的。因此，在管理完成端口模型时，服务提供者不必要考虑太多。

(4) 重叠I/O的管理

SPI客户机在利用前面提到的任何一个重叠式 I/O模型，调用你的分层提供者时，对你的提供者的重叠式管理器来说，它应该利用基于事件的或第 8章中所讲的完成端口重叠式 I/O方法，调用低层提供者。如果此时你的提供者运行于 Windows NT或Windows 2000，我们建议大家在低层提供者上采用完成端口来执行重叠式 I/O。在Windows 95和Windows 98平台上，只能使用基于事件的重叠式I/O。在处理重叠式I/O事件时，下面三个上调函数可供你的提供者使用：

```
WSAEVENT WPUCreateEvent(LPINT lpErrno);
BOOL WPUResetEvent(WSAEVENT hEvent, LPINT lpErrno);
BOOL WPUCloseEvent(WSAEVENT hEvent, LPINT lpErrno);
```

WPUCreateEvent函数建立并返回一个事件对象。这个函数完全和第 8章中讲的WSACreateEvent函数一样：建立了一个处于手工重设模式的事件对象。如果 WPUCreateEvent不能建立事件对象，就会返回 NULL，而且lpErrno中将包含特定的Winsock错误代码。WSPResetEvent函数则和WSAResetEvent函数一样：把一个事件对象（参数 hEvent）的signaled状态重设为unsignaled状态。WPUCloseEvent函数和WSACloseEvent函数一样，释放与事件对象句柄关联在一起的所有的操作资源。

本书配套光盘上，我们的LSP示例采用了基于事件的重叠式I/O来管理SPI客户机的所有的I/O活动。这样，我们的 LSP示例便可在 Windows 2000、Windows NT、Windows 98以及Windows 95上使用。但是，必须注意：这个示例只有一个服务重叠式 I/O操作的线程，这样一来，我们的提供者便不能同时利用基于事件的重叠式 I/O为多个WSA_MAXIMUM_WAIT_EVENTS(64)事件对象服务（参见第8章）。如果你的提供者希望为64个以上的事件对象提供服务，就可以建立多个服务线程，提供对多个事件对象的服务。如果此时正在针对Windows NT和Windows 2000开发提供者，我们建议大家采用完成端口，而不是基于事件的重叠式I/O，这样便可不受它的限制。

14.2.6 扩展函数

Winsock库Mswsock.lib为应用程序提供了扩展函数，增强了 Winsock的功能。表14-2对目前已获支持的扩展函数进行了定义。

表14-2 Winsock扩展函数

扩展函数	GUID
AcceptEx	WSAID_ACCEPTEX
GetAcceptExSockaddrs	WSAID_GETACCEPTEXSOCKADDRS
TransmitFile	WSAID_TRANSMITFILE
WSARecvEx	没有相关的GUID

链接到 Mswsock.lib的一个应用程序在使用 AcceptEx、GetAcceptExSockaddrs和TransmitFile时，利用SIO_GET_EXTENTION_FUNCTION_POINTER选项，隐式地调用了提供者的WSPIoctl函数。WSPIoctl的定义如下：

```
int WSPIoctl(
    SOCKET s,
    DWORD dwIoControlCode,
    LPVOID lpvInBuffer,
    DWORD cbInBuffer,
    LPVOID lpvOutBuffer,
    DWORD cbOutBuffer,
    LPDWORD lpcbBytesReturned,
    LPWSAOVERLAPPED lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine,
    LPWSATHREADID lpThreadId,
    LPINT lpErrno
);
```

调用这个函数时， dwIoControlCode参数被设为 SIO_GET_EXTENSION_FUNCTION_POINTER这个值。lpvInBuffer参数中包含指向一个通用唯一标志符（GUID）的指针，它表示Mswsock.lib正在寻找哪些扩展函数（表 14-2中所列的GUID值定义了目前已获支持的扩展函数）。如果这个GUID值与其定义的值匹配，提供者就必须通过 lpvOutBuffer，返回一个函数指针，指向提供者的扩展函数。其余的参数不适宜直接管理 SPI中的扩展函数。

特别注意，表 14-2中的WSARecvEx函数没有 GUID。这是因为 WSARecvEx没有调用WSPIoctl，而是直接调用了WSARecv。其结果是提供者不能直接监视 WSARecvEx函数。

14.2.7 传输服务提供者的安装

要安装传输服务提供者，应开发一个简单的应用程序，使其将一个分层或基础提倡者插入服务提供者的Winsock 2目录。传输提供者的安装方式决定了它是一个分层提供者，还是一个基础提供者。安装程序只在 Winsock 2系统配置数据库中，配置了传输提供者，这个系统配置数据库是一个目录，系统上已安装的服务提供者都在这个目录中。配置数据库让 Winsock 2得知服务提供者已存在，并定义了提供的服务类型。 Winsock应用程序建立套接字时，Winsock 2便利用这个数据库来判断需要加载哪些传输服务提供者。 WS2_32.DLL在数据库中搜索第一个与socket和WSASocket API调用的套接字输入参数（比如说地址家族、套接字类型和协议）匹配的提供者。一旦找到与之匹配的条目， Ws2_32.dll便加载相应的服务提供者之DLL（动态数据链接库），这个DLL是定义在该目录中的。

从本质上说，要在Winsock 2服务提供者数据库内成功安装和管理服务提供者条目，需要四个SPI函数。这四个函数的前缀均为WSC：

- WSCEnumProtocols
- WSCInstallProvider
- WSCWriteProviderOrder
- WSCDeInstallProvider

这些函数利用WSAPROTOCOL_INFOW结构，对服务提供者数据库进行查询和操作，关于这个结构，可参见第5章。由于我们的目的是安装传输服务提供者，因此主要讨论该结构的

这三个字段：ProviderId、dwCatalogEntryId和ProtocolChain。ProviderId字段是一个GUID，允许你在任何系统上定义和安装唯一一个提供者。 dwCatalogEntryId字段只用一个唯一的数字值标识这个数据库中的 WSAPROTOCOL_INFOW目录条目结构。 ProtocolChain字段则决定WSAPROTOCOL_INFOW结构是一个基础提供者目录条目，还是一个分层提供者亦或是一个提供者协议链。ProtocolChain字段是一个WSAPROTOCOLCHAIN结构，该结构的定义如下：

```
typedef struct {
    int ChainLen;
    DWORD ChainEntries[MAX_PROTOCOL_CHAIN];
} WSAPROTOCOLCHAIN, FAR * LPWSAPROTOCOLCHAIN;
```

ChainLen字段决定一个目录条目是代表一个分层式提供者（当 ChainLen = 0时），还是一个基础提供者（当 ChainLen = 1时），或者是一个协议链（当 ChainLen >1时）。协议链（protocol chain）是一个特殊的目录条目，定义分层式服务提供者在 Winsock和其他服务提供者之间的位置（参见图 14-2）。服务提供者数据库中，不管是分层提供者，还是基础提供者，每个提供者都只有一个目录条目。最后一个字段 ChainEntries1，是一个目录ID数组，用于说明服务提供者加入协议链的顺序。建立套接字期间， Ws2_32.dll在目录中搜索相应的服务提供者时，只查找协议链和基础提供者目录条目。分层式提供者目录条目（ ChainLen为0）则被Ws2_32.dll忽略，它存在于协议链目录条目中的唯一目的便是向一个协议链标识分层提供者。

1. 基础提供者的安装

要安装一个基础提供者，必须建立一个 WSAPROTOCOL_INFOW目录条目结构，用它来代表基础提供者。这样便要求用能够说明该基础提供者的协议属性信息，来填充该结构中的字段。记住，ProtocolChain结构中的ChainLen 字段应设为1。这个基础结构一旦定义，就需要利用WSCInstallProvider函数，把它安装在目录中。 WSCInstallProvider函数的定义如下：

```
int WSCInstallProvider(
    const LPGUID lpProviderId,
    const LPWSTR lpszProviderDllPath,
    const LPWSAPROTOCOL_INFOW lpProtocolInfoList,
    DWORD dwNumberOfEntries,
    LPINT lpErrno
);
```

lpProviderId参数是一个 GUID，允许你向 Winsock目录标识一个协议提供者。lpszProviderDllPath参数是一个字串，其中包括到该提供者之DLL的加载路径。这个字串中可包括%SystemRoot%之类的环境变量。 lpProtocolInfoList参数代表安装在这个目录中的一个WSAPROTOCOL_INFOW数据结构的数组。为了安装基础提供者，可将 WSAPROTOCOL_INFOW结构分配给该数组的第一个元素。从dwNumberOfEntries参数中，可得知lpProtocolInfoList数组中有多少条目。如果WSCInstallProvider函数失败，返回SOCKET_ERROR时，lpErrno参数中就会包含特定的错误代码信息。

2. 分层提供者的安装

要安装分层式服务提供者，需要建立两个 WSAPROTOCOL_INFOW目录条目结构。一个将代表分层提供者（比如协议链长度等于 0），另一个将代表一个协议链（比如，协议链长度大于1），该协议链把分层提供者与一个基础提供者链接起来。应该用一个现成服务提供者的WSPROTOCOL_INFOW目录条目结构的属性来初始化这两个结构。调用 WSCEnumProtocols函数（定义如下），便可获得这个现成服务提供者的WSPROTOCOL_INFOW目录条目结构了。

```
int WSCEnumProtocols(
    LPINT lpiProtocols,
    LPWSAPROTOCOL_INFOW lpProtocolBuffer,
    LPDWORD lpdwBufferLength,
    LPINT lpErrno
);
```

lpiProtocols参数是一个可选的值数组。如果lpiProtocols是NULL（真），就会返回所有有效协议的信息；否则，只能取得该数组中列出的协议信息。 lpProtocolBuffer参数是一个应用程序提供的缓冲区，用 Winsock 2中的 WSAPROTOCOL_INFOW结构来充填。输入端的lpdwBufferLength参数是字节数，表示投入WSCEnumProtocols中的lpProtocolBuffer缓冲区中的字节是多少。若是在输出端，则收到缓冲区的最小长度，这个长度可投入 WSCEnumProtocols内，获得全部所请求的信息。如果函数调用失败，并返回 SOCKET_ERROR，lpErrno参数中就会出现特定的错误信息。一旦有了准备进行重叠操作的提供者目录条目，就可以把这个提供者的WSAPROTOCOL_INFOW结构复制到新建立的结构中。

初始化之后，首先需要用 WSCInstallProvider来安装你的分层提供者目录条目，然后，利用WSCEnumProtocols列举出所有的目录条目，获得安装之后为这个结构分配的目录 ID。然后，用这个目录条目来设置一个协议链目录条目，通过它，便将你的分层提供者和另一个提供者链接起来。接下来，便调用 WSCInstallProvider来安装这个链式提供者。下面的伪代码对此进行了解释：

```
...
WSAPROTOCOL_INFOW LayeredProtocolInfoBuff,
                  ProtocolChainProtoInfo,
                  BaseProtocolInfoBuff;
...
// Retrieve BaseProtocolInfoBuff using WSCEnumProtocols()

memcpy (&LayeredProtocolInfoBuff , &BaseProtocolInfoBuff,
    sizeof(WSAPROTOCOL_INFO));
LayeredProtocolInfoBuff.dwProviderFlags = PFL_HIDDEN;
LayeredProtocolInfoBuff.ProviderId = LayeredProviderGuid;

// This entry will be filled in by the system
LayeredProtocolInfoBuff.dwCatalogEntryId = 0;

LayeredProtocolInfoBuff.ProtocolChain.ChainLen =
    LAYERED_PROTOCOL;
WSCInstallProvider(&LayeredProviderGuid, L"lsp.dll",
    &LayeredProtocolInfoBuff, 1, & install_error);

// Determine the catalog ID of the layered provider
// using the WSCEnumProtocols() function
for (i = 0; i < TotalProtocols; i++)
    if (memcmp (&ProtocolInfo[i].ProviderId, &ProviderGuid,
        sizeof (GUID))==0)
   {
      LayeredCatalogId = ProtocolInfo[i].dwCatalogEntryId;
      break;
   }

Memcpy(&ProtocolChainProtoInfo, &BaseProtocolInfoBuff,
       sizeof(WSAPROTOCOL_INFO));
ProtocolChainProtoInfo.ProtocolChain.ChainLen = 2;
```

```
ProtocolChainProtoInfo.ProtocolChain.ChainEntries[0] =
    LayeredProvideProtocolInfo.dwCatalogEntryId;
ProtocolChainProtoInfo.ProtocolChain.ChainEntries[1] =
    BaseProtocolInfoBuff.dwCatalogEntryId;

WSCInstallProvider(
    &ChainedProviderGuid,
    L"lsp.dll",                 // lpszProviderDllPath
    &ProtocolChainProtoInfo,    // lpProtocolInfoList
    1,                          // dwNumberOfEntries
    &install_error              // lpErrno
);
```

注意，PFL_HIDDEN标志是在WSAPROTOCOL_INFOW结构中，针对以上伪代码中的分层提供者指定的。这个标志确保 WSAEnumProtocols函数（参见第5章）返回的缓冲区内不会出现这个分层提供者的目录。

安装程序应该管理的另一个重要标志是 XP1_IFS_HANDLES。对任何一个非IFS的服务提供者来说，只要它建立自己的套接字句柄时，用的是 WPUCreateSocketHandle，就不应该在自己的WSAPROTOCOL_INFOW结构中设置这个 XP1_IFS_HANDLES标志。没有设置XP1_IFS_HANDLES标志， Winsock应用程序便会由于前面提到的性能下降而避免使用ReadFile和WriteFile。

3. 提供者的排序

系统上一旦安装了服务提供者，就必须考虑 Winsock 2怎样在数据库中搜索服务提供者这一问题。多数Winsock应用程序都通过socket或WSASocket函数中的参数，指定了自己需要的协议。举个例子来说，如果你的应用程序利用地址家族AF_INET和套接字类型SOCK_STEREAM建立了一个套接字， Winsock 2便在能够满足这一需要的数据库中搜索默认的TCP/IP协议链或基础提供者目录条目。在用 WSCInstallProvider安装一个服务提供者时，目录条目便自动成为数据库中的最后一个条目。要使服务提供者成为默认的 TCP/IP提供者，必须对数据库中的提供者进行重新排序，并把协议链目录条目放在 TCP/IP提供者之前，这是通过调用WSCWriteProviderOrder来完成的，它的定义如下：

```
int WSCWriteProviderOrder(
    LPDWORD lpwdCatalogEntryId,
    DWORD dwNumberOfEntries
);
```

lpwdCatalogEntryId参数接受一个由目录ID组成的数组，这些ID用于标志目录的排序。正如前面所讲的那样，调用WSCEnumProtocols，便可获得该目录中的目录ID了。dwNumberOfEntries参数是一个计数，表示这个数组中有多少目录条目。如果 WSCWriteProviderOrder调用成功，就返回ERROR_SUCCESS(0)；否则，就返回一个Winsock错误代码。

WSCWriteProviderOrder函数在 Ws2_32.dll库内。要使用它，应用程序必须链接到Sporder.lib。另外，这个库关联的 Sporder.lib模块也不在Windows操作系统中。其支持 DLL可在“微软开发者网络”（MSDN）库中找到。如果你打算在编程过程中使用它，必须将它包括到你的安装程序中。 MSDN库还提供了一个很方便的软件工具，名为 Sporder.exe，允许你对Winsock 2数据库中的目录条目进行查看和重新排序。图 14-3是一个示例，在Windows 2000系统上安装一个分层式提供者之后，对 Winsock 2进行的快速查看（利用Sporder.exe进行的）。

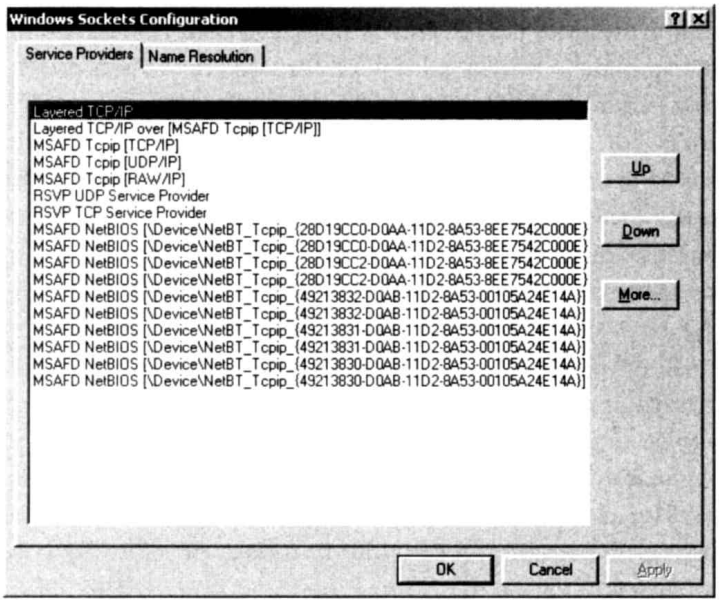

图14-3 Sporder.exe

4. 服务提供者的删除

从Winsock 2目录中删除一个服务提供者很简单。主要的任务便是调用WSCDeinstallProvider函数，该函数的定义如下：

```
int WSCDeinstallProvider(
   LPGUID lpProviderId,
   LPINT lpErrno
);
```

lpProviderId参数代表准备删除的服务提供者的 GUID。如果函数返回SOCKET_ERROR，lpErrno参数就会接收特定的Winsock错误代码。

在删除服务提供者时，必须考虑到服务提供者的目录 ID完全可能包含在一个分层式服务提供者中。如果是这样，应该从所有引用了提供者的协议链中，将你的目录 ID删除。

安装分层式提供者所涉及的问题

分层式服务提供者的确大大发挥了连网服务的潜能。但是，目前的 Winsock 2规格中，尚未解决一个问题：如果一个分层式服务提供者发现系统中已安装了另一个分层式服务提供者，怎样才知道自己应插在协议链中的何处呢？举个例子来说，若你打算在一个已安装了URL过滤提供者的系统上，安装一个数据加密提供者，显而易见，在一个现成的协议链中，数据加密提供者应该插在过滤提供者下面。但问题是提供者安装程序无法找到现成提供者提供的服务类型，因而也无从得知自己应该插在协议链中的什么地方。在由主管决定安装哪些提供者以及对这些提供者进行排序的联网环境中，这倒不是个大问题。但随着分层式提供者的日益普及，这个问题越来越明显。唯一安全的安装办法便是：直接在基础提供者上安装分层式提供者，并使新的协议链成为协议的默认提供者。这样，可以保证新提供者的服务，但删除了被当作默认提供者链的现成的分层式提供者。

除了分层式服务提供者在协议链中的排序问题外， Winsock 2规格中还有另一个相关

问题没有得到解决：现成的分层式提供者如何才能避免在发生时，被修改和警告呢？和前一个问题相比，这个问题倒不大。实际上，不修改一个分层式提供者协议链，分层式提供者的硬编码器照样可将协议链的顺序硬编码到 WSPStartup函数中，并在 LSP的WSAPROTOCOL_INFOW结构中的ProtocolChain.ChainLen指定成1，把这个分层式提供者当作一个基础提供者安装到系统。

14.3 命名空间服务提供者

通过第10章的学习，大家已经知道应用程序是怎样在名字空间内注册和解析服务的，这对网络上动态建立的服务来说，是一个非常关键的特性。不幸的是，由于种种限制，现有的名字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，供建立你自己的名字空间所用，通过这个方法，可以采取任何一种自己喜欢的方式对命字注册和解析进行处理。

这是通过建立一个可实施 9个名字空间函数的 DLL来完成的。这些函数的前缀均是 NSP，与第 10章中讲的 RNR函数对应。比方说，等同于 WSASetService的名字空间函数便是NSPSetService。DLL建立之后，便利用标志这个名字空间的 GUID安装到系统目录中。随后，应用程序便可以在你的名字空间内注册和查询服务了。

本小节，我们先为大家介绍如何安装名字空间提供者，然后对名字空间提供者必须实施的函数进行说明。最后，随注册和解析服务所用的一个示范应用程序，向大家展示一个示例性的名字空间提供者。

14.3.1 名字空间的安装

简单说来，名字空间只是一个 DLL，可以实施名字空间提供者函数。在应用程序可使用名字空间之前，必须通过 WSCInstallNameSpace函数，使系统知道这个名字空间。反之，一旦安装了提供者，就可以分别利用 WSAEnableNSProvider和WSAUnInstallNameSpace这两个函数，取消这个提供者或从系统目录中将它删除。接下来便是对这些函数的说明。

1. WSCInstallNameSpace

这个函数用于把一个提供者安装到系统目录中。它的声明如下：

```
int WSCInstallNameSpace (
    LPWSTR lpszIdentifier,
    LPWSTR lpszPathName,
    DWORD dwNameSpace,
    DWORD dwVersion,
    LPGUID lpProviderId
);
```

大家注意到的第一点便是所有字串参数都是宽位字符串。事实上，所有的名字空间提供者都是用宽位字符串来实施的。稍后将详细讨论。lpszIdentifier参数是名字空间提供者的名字。这个名字是在调用WSAEnumNameSpaceProviders时，列举得来的（参见第10章）。lpszPathName参数是DLL的位置。这个字串中可包含% SystemRoot%之类的环境变量。 dwNameSpace参数是该名字空间的数字化标志符。比如说，头文件 Nspapi.h定义了一些众所周知的名字空间，比如IPX SAP的NS_SAP。dwVersion参数则为该名字空间设置版本号。最后， lpProviderId参数是一个标志该名字空间的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标志的名字空间已经存在；如果是WSAEACCESS，就表示调用进程不具备安装特权。只有主管级的用户才能安装名字空间。

2. WSCEnableNSProvider

这个函数用于修改名字空间提供者的状态。可用于启用或取消提供者。它的声明如下：

```
int WSCEnableNSProvider (
    LPGUID lpProviderId,
    BOOL fEnable
);
```

lpProviderId参数是名字空间的GUID标识符，表示准备对这个名字空间进行修改。fEnable参数是一个布尔值，表示禁用或启用提供者。已禁用的提供者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。如果提供者的GUID无效，就会返回WSAEINVAL。

3. WSCUnInstallNameSpace

这个函数用于从目录中删除名字空间提供者。它的定义如下：

```
int WSCUnInstallNameSpace ( LPGUID lpProviderId );
```

lpProviderId参数是准备删除的那个名字空间的GUID。如果该GUID无效，函数调用就会失败，并返回WSAEINVAL。

14.3.2 名字空间的实施

名字空间必须实施9个名字空间函数，这些函数与第10章中所讲的RNR函数相对应。除了实施这些函数外，还必须开发一个保持数据的方法。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。不过练习中不提倡）。第10章，我们提到了名字空间的三种类型：动态、静态和固定状态。显然，实施静态名字空间不可取，因为它不允许对服务通过编程的方式进行注册。稍后，我们将具体讲讲如何保持名字空间固有的数据。

另外，大家还必须知道这一点：在所有名字空间提供者函数中，使用宽位字符串是非常重要的。不仅要求函数中的字串参数如此，要求RNR结构中的字串也要如此，比如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字时，既可用RNR函数及结构的普通（ASCII）版本，又可以用它们的宽位字符（Unicode）版本，怎么可能呢？答案是随便哪个版本都可以，因为所有的ASCII调用都可通过一个中间层，将全部字串统一转换成宽位字符串。函数调用并返回之后便如此，即如果WSAQUERYSET返回调用程序（随WSALookupServiceNext一起）这个名字空间提供者返回的全部数据起初都是Unicode字符，但在函数调用返回之前，就转换成了ASCII字符。换言之，如果你的应用程序使用的是RNR函数，那么调用它的宽位字符版本就要快得多，因为无须进行转换。

在名字空间必须实施的这9个函数中，只有7个函数有相应的Winsock 2 RNR函数（参见表14-3）。其余两个函数用于初始化和清除名字空间。系统中一旦安装了名字空间，应用程序通

过指定GUID或安装期间指定的名字空间标识符，便可以使用这个名字空间了。接下来，应用程序调用我们在第10章中讲到的标准Winsock2RNR函数。调用其中一个函数时，也会调用其相应的名字空间提供者函数。比如说，一个应用程序在调用 WSAInstallServiceClass（该函数引用了自定义名字空间的GUID）时，也会调用那个提供者的NSPInstallServiceClass函数。关于各个名字空间，我们将在下一小节一一说明。

表14-3 Winsock 2注册和名解析函数与名字空间提供者函数之间的对应关系

Winsock函数	等同（或对应）的名字空间提供者函数
WSAInstallServiceClass	NSPInstallServiceClass
WSARemoveServiceClass	NSPRemoveServiceClass
WSAGetServiceClassInfo	NSPGetServiceClassInfo
WSASetService	NSPSetService
WSALookupServiceBegin	NSPLookupServiceBegin
WSALookupServiceNext	NSPLookupServiceNext
WSALookupServiceEnd	NSPLookupServiceEnd

1. NSPStartup

只要加载名字空间提供者DLL，便会调用NSPStartup函数。你的名字空间实施中必须包括这个函数，而且必须从DLL中导出。为使名字空间提供者正常工作，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
);
```

第一个参数是lpProviderId，它是这个名字空间提供者的GUID。lpnspRoutines参数是一个NSP_ROUTINE结构，这个结构是你在实施这个函数时，必须填充的。这个结构提供了 8个函数指针，分别指向你的提供者的另8个名字空间函数。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;
} NSP_ROUTINE, FAR * LPNSP_ROUTINE;
```

这个结构的第一个字段 cbSize，表明NSP_ROUTINE结构的长度。下两个字段，dwMajorVersion和dwMinorVersion，包括你的提供者的版本信息（可以是任意的）。提供者条目的其余部分设为它们各自的函数指针。比如说，提供者为 NSPSetService字段分配了自己的NSPSetService函数地址（不管采用的是什么名字）。你的提供者函数名可以是任意的，但其参数和返回类型必须与提供者的定义匹配。

NSPStartup实施中，唯一需要做的是填充NSP_ROUTINE结构。一旦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和Winsock错误代码。比如，如果一个提供者尝试分配内存失败了，它就会随 WSA_NOT_ENOUGH_MEMORY参数一起，调用WSASetLastError，然后再返回SOCKET_ERROR。

现在，该好好讨论一下提供者的 DLL中的错误处理了。为提供者实施的所有函数调用若成功，便返回NO_ERROR；若失败，便返回 SOCKET_ERROR。如果判断调用失败了，必须在返回之前，设置相应的Winsock错误代码。否则，任何试图利用你的名字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，WSAGetLastError返回0。这样可能会为那些试图从容处理错误的应用程序带来许多麻烦； 0也不是我们希望返回的值，因为它说明不了任何问题。

2. NSPCleanup

卸载提供者的DLL时，便调用这个例程。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 );
```

唯一的参数就是你的名字空间提供者的 GUID。该函数的唯一作用便是清除全部动态分配的内存。

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
);
```

第一个参数是提供者的 GUID。lpServiceClassInfo参数是正在注册的 WSASERVICE-CLASSINFOW结构。你的提供者必须保持一个服务类列表，确保这个 WSASERVICE-CLASSINFOW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，提供者肯定会返回WSAEALREADY。不然，提供者就应该保持这个服务类，以便别的 RNR操作能够引用它。

其余的名字空间提供者函数一般引用已安装的服务类。

4. NSPRemoveServiceClass

这个函数和NSPInstallServiceClass函数对应（一个用来安装，一个用来删除）。这个名字空间函数的对应函数是WSARemoveServiceClass。它的声明如下：

```
int NSPRemoveServiceClass (
    LPGUID lpProviderId,
    LPGUID lpServiceClassId
);
```

和上一个函数一样，第一个参数是提供者的 GUID。第二个参数lpServiceClassId，是即将删除的服务类的GUID。提供者必须从存储设备上删除指定的服务类。如果没有找到 lpService ClassId指定的服务类，提供者必然生成错误 WSATYPE_NOT_FOUND。

5. NSPGetServiceClassInfo

NSPGetServiceClassInfo函数的对应函数是WSAGetServiceClassInfo。它获取于一个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，第一个参数仍然是提供者的 GUID。lpdwBufSize参数表明第三个参数lpService ClassInfo中包含的字节数有多少。在输入端，第三个参数是一个 WSASERVICE-CLASSINFOW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。如果提供者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，使之表明返回了多少字节。

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，并设置错误WSATYPE_FOUND。另外，如果找到了匹配的服务类，但提供的缓冲区太小，提供者就应该更新lpdwBufSize，使之表明需要多少缓冲区才足够，并把错误设为 WSAEFAULT。

6. NSPSetService

NSPSetService函数的对应函数是 WSASetService，既可以用于注册服务，也可以用于从名字空间中删除服务。它的定义如下：

```
int NSPSetService (
    LPGUID lpProviderId,
    LPWSASERVICECLASSINFOW lpServiceClassInfo,
    LPWSAQUERYSETW lpqsRegInfo,
    WSAESETSERVICEOP essOperation,
    DWORD dwControlFlags
);
```

第一个参数是提供者的GUID。lpServiceClassInfo参数是该服务所属的那个WSASERVICE-CLASSINFOW结构。lpqsRegInfo参数是即将注册或删除（根据第四个参数 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。

名字空间提供者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，采取相应的行动。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 10章的10.3节“服务的注册”小节，其中着重讨论了 WSASetService。你的提供者的NSPSetService函数对这些标志进行相应的处理。

如果你的服务提供者在更新或删除不能找到的服务，就需要设置错误 WSASERVICE_NOT_FOUND。如果提供者在注册一个服务， WSAQUERYSETW结构却是无效或未完成的，提供者就会生成WSAEINVAL错误。

这个函数是最复杂的名字空间函数之一（仅次于 NSPLookupServiceNext）。提供者必须维护一种方案，使其一直保持能被注册的服务之数据。而且，还必须允许 NSPSetService函数对这一数据进行更新。

7. NSPLookupServiceBegin

NSPLookupServiceBegin函数和NSPLookupServiceNext以及NSPLookupServiceEnd有关，用于开始对你的名字空间进行查询。该函数的对应函数是 WSALookupServiceBegin，用于建

立搜索标准。它的原型如下：

```
int NSPLookupServiceBegin (
    LPGUID lpProviderId,
    LPWSAQUERYSETW lpqsRestrictions,
    LPWSASERVICECLASSINFOW lpServiceClassInfo,
    DWORD dwControlFlags,
    LPHANDLE lphLookup
);
```

和前面几个函数一样，第一个参数仍然是提供者的 GUID。lpqsRestrictions参数是定义查询参数的WSAQUERYSETW结构。第三个参数是lpServiceClassInfo，它是一个WSASERVICE-CLASSINFOW结构，其中包含了指定的服务类的简要信息，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 10章。注意，并不是所有的标志都能对每个提供者产生影响。比如，如果你的名字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，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。

实施NSPLookupServiceBegin时，要记住这个操作不能取消，而应该尽快完成它。因此，在你打算开始网络查询时，若想成功返回，就不应该要求得到响应。

提供者本身应该保存查询参数，并将一个独一无二的句柄关联到这个查询，以备日后引用。除此以外，提供者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。

8. NSPLookupServiceNext

一旦用NSPLookupServiceBegin开始了查询，应用程序便调用 NSPLookupServiceNext（该函数会依次调用这个名字空间提供者函数 NSPLookupServiceNext）。事实上，这个调用是在找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
);
```

第一个参数hLookup，是WSALookupServiceBegin函数返回的查询句柄。 dwControlFlags参数可以是标志LUP_FLUSHPREVIOUS，表明提供者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投递的缓冲区的长度。

NSPLookupServiceNext被触发时，提供者应该查找由句柄 hLookup标志的查询参数。一旦获得了查询参数，便在开始指定的服务类中开始搜索已注册的服务。就像我们在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，调用进程便调用 WSALookupServiceNext若干次，每次调用，提供者都要返回一个数据集。如果没有找到匹配条目，提供者就会返回 NSPLookup错误。如果应用程序在进程中调用 WSALookupServiceNext的同时，又从另一个线程中调用了 WSALookupServiceEnd，可能会删除进程中的查询。这一

事件中，WSALookupServiceNext就会失败，并返回WSA_E_CANCELLED错误。

9. NSPLookupServiceEnd

完成查询之后，便调用NSPLookupServiceEnd函数结束查询，并释放所有名字空间资源。该函数的定义如下：

```
int NSPLookupServiceEnd ( HANDLE hLookup );
```

它只有一个参数——hLookup，该参数是一个句柄，指向行将关闭的查询。如果没有找到指定的句柄（例如是无效句柄的话），函数调用就会失败，并返回WSA_INVALID_HANDLE错误。

14.3.3 名字空间提供者示范

关于如何建立自己的名字空间及建立时需要注意的重要问题（比如保持数据的方法），我们已讲了很多。但是，要开发一个完整的名字空间提供者并不简单，接下来，为大家介绍一个名字空间示范。该示范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。

示范提供者在配套光盘的Examples\Charpter14\NSP目录中下面三个文件中：Mynsp.h、Mynsp.cpp和Mynsp.def。范例名字空间的DLL便由这三个文件组成。除DLL外，大家还可以找到一个名字空间服务，它是一个Winsock服务器，负责处理DLL发出的请求。这个维护服务注册信息的服务器位于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，这些支持例程用于收集和取消服务和 DLL之间的套接字上发送的数据。数据的收集和取消稍后将具体说明。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是一个修改过的范例，对在我们定义的名字空间内注册和查询服务进行了解释，原来的范例参见第10章。

下面的小节将讨论怎样实施我们的名字空间。首先，大致谈谈我们所用的保持数据的方法。然后对事实上的名字空间DLL的构成和名字空间服务的安装进行说明。接下来是名字空间服务的实施。最后展示应用程序如何对我们定义的名字空间进行注册和查询。

1. 数据的保持

针对这里的名字空间，我们选用了一个独立的Winsock服务来保持名字空间信息。在DLL中实施的每一个名字空间函数中，要完成实施的话，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（在loopback地址127.0.0.1上监听）。在实际实施过程中，可通过注册或其他方法访问我们的名字空间服务的 IP地址，这样一来，应用程序在调用这个名字空间时，无论它在什么地方运行，都可连接到这个名字空间服务。比如，以DNS为例，DNS服务器的IP地址不是静态设置的，就是在DHCP请求期间获得的。

当然，编一个服务来维护这一信息并不是唯一的选择。还可在网络上维护一个文件，使其保留必要的信息。但是，这也不是最佳途径，因为性能受到了磁盘操作的影响。我们的名字空间范例存在的性能限制在于：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，比如保证重新传输已丢弃的数据包。

2. 名字空间DLL

在实施名字空间服务之前，先来看看名字空间 DLL。每个名字空间提供者都需要一个独一无二的GUID，我们的GUID定义在Mnsp.h。除了要有一个独一无二的标识符之外，我们的名字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10章所见的那样。我们的名字空间的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
```

这些值非常重要，因为想使用这个名字空间的应用程序必须在它们的 Winsock调用中指定这些值。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10章）。同时，要知道一个应用程序在执行指定 NS_ALL名字提供者这样的操作时，应该在所有已安装名字提供者上执行。这一点尤为重要，因为有几个 Windows应用程序（比如说MicrosoftInternet Explorer）需要在全部已安装名字空间提供者上执行查询。因此，在测试一个名字空间提供者时，务必谨慎。编得糟糕的名字空间提供者可能导致整个系统出问题。另外， GUID和名字空间标识符的值也非常重要，因为安装名字空间提供者需要它们。

现在，来看看在Mynsp.cpp中实施的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名字空间函数对NSP_ROUTINE结构进行了初始化。清除例程则没有派上用场，因为没必要进行清除。

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。表示即将采取的行动（服务注册、服务删除、查询等等）。

3) 汇集参数，并把它们发给服务。参数类型由操作决定。比如， 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，就可以读取和取消返回结果了。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，实施DLL并不复杂。NSP函数必须获得参数，并对它们进行处理。这里的函数是向名字空间服务投递信息的操作。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。大多数名字空间函数都把WSAQUERYSET或WSASERVICECLASSID当作参数。而且，必须在连接了服务的套接字上收发这个对象。难就难在这些结构不是连续不断的内存块。换言之，这些结构中包含有字串指针，而别的结构可以放在内存中的任何一个地方（如图 14-4所示）。你需要做的是汇集这些分散的内存段（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“ 数据汇集 ”（marshaling data）。在接收数据的这一端，必须保持这个进程。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。

针对这里的名字空间提供者，我们提供了汇集和分散 WSANAMESPACEINFO和WSAQUERYSET结构的函数。这些函数位于Nspsvc.cpp文件中，供该名字空间的DLL和该名字空间服务使用（因为双方都需要能够收集和分散结构）。四个函数都很简单，我们不打算在此赘述。

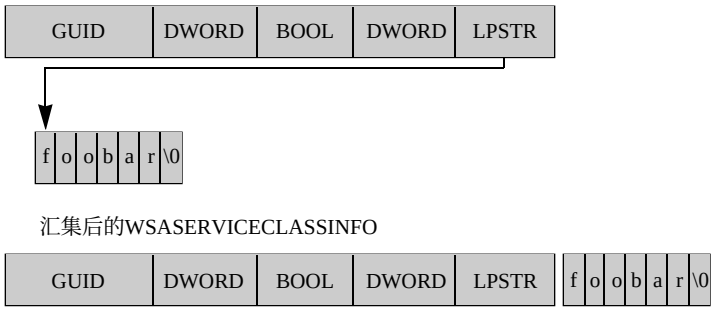

图14-4 数据的收集

3. 名字空间的安装

整个实施过程中，名字空间提供者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者：

```
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());
}
```

这个调用中参数是提供者的名称、 DLL的位置、整数识别符、版本以及GUID。完成安装之后，唯一需要的就是：确保你提供的名字空间 DLL的位置正确。真正可能出现的错误便是你试图安装的那个名字空间的GUID已经被另一个提供者所用。

名字空间提供者的删除更为简单。下面的代码取自我们的安装程序，用于删除我们的提供者：

```
ret = WSCUnInstallNameSpace(&MY_NAMESPACE_GUID);
if (ret == SOCKET_ERROR)
{
    printf("Failed to remove provider: %d\n", WSAGetLastError());
}
```

4. 名字空间服务

名字空间服务才是名字提供者的精髓所在。这个服务可对所有已注册的服务类和服务实例进行追踪。名字空间DLL被用户的应用程序触发时，便连上名字空间服务，开始执行操作。这个服务很简单。在main函数内部，建立了一个监听套接字。然后，在一个循环内，名字空间DLL实例接受连接。为了简单起见，一次只处理一个连接。这样还可以防止你同步访问保持名字空间信息的那个数据结构。再次提醒大家注意，实际上提供者不会如此，因为这样会

降低性能，我们这里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名字空间DLL中，读取一个单一字节。

循环内部，对行动进行解码，并把从名字空间 DLL中取得的参数投递给服务。这时起开始执行所请求的行动。这些行动并不复杂，采用的代码脉络相当清晰，并且，可从针对每个可能的行动而采取的步骤中，看出名字空间服务的原理，因此，不必赘述。但我们仍然要提到维护信息的各个结构。名字空间提供者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，我们维护的是两大通用数组（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，名字空间提供者的 main函数率先调用LookupServiceClass（一个支持例程，定义于 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，服务便返回一个错误（DLL将它翻译成WSAEALREADY）。若不然，就会在这个数组后添加新的服务类，而且dwNumServiceClasses计数器则会递增。

删除服务类的过程中（和安装服务类一样），main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，代码递减计数器。 Winsock 2规格中，对名字空间提供者来说，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，由你自己决定。但如果已注册服务引用了一个服务类，我们的名字空间范例不允许你将这个服务类删除。

维护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，服务必须维护的信息是查询。应用程序发起一个查询时，必须维护查询期间所需的参数，并为查询分配一个独一无二的句柄（即查询句柄）。这是非常必要的，因为WSALookup ServiceNext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。这样一来，以后对WSALookupServiceNext的调用，便从上一次调用留下的那个位置开始返回。

5. 名字空间的查询

我们的名字空间范例中，最后一部分便是 Rnrcs.c文件。它是第10章的名字注册和解析范例的改版。为了使示例尽可能浅显易懂，我们只做了少许改动。第一个改动是将代码改为列举已安装的名字空间提供者，但只返回 NS_MYNSP提供者。第二个改动是在注册一个服务时，Rnrcs.c只列举了本地 IP接口，将用它代替我们的服务地址。我们的服务提供者支持任何SOCKADDR类型的注册。最后，该范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10章中的范例一样。

6. 运行范例

一旦已完成所有范例的编译，提供者的安装和使用就非常简单了。利用下面的命令安装提供者：

```
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）。一旦安装了名字空间，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14-4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查询，用以说明服务已经被删除。

表14-4 使用示例名字空间注册和查询服务的命令序列

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

14.4 Winsock SPI函数的调试追踪

Winsock 2可对Ws2_32.dll中的API和SPI调用进行追踪。对开发传输服务提供者和名字空间提供者来说，这一点是非常有用的。MSDN平台SDK中有两个示例（Dt_dll和Dt_dll2）都是为追踪SPI调用而设计的。这两个示例的妙处在于，可以对它们进行修改，使之追踪自己感兴趣的API和SPI。

要使用这个调试特性，首先必须取得已通过你的目标平台核查的 Winsock 2的Ws2_32.dll版本（build号）。可在Mssdk\Bin\Debug\Winsock下的MSDN平台SDK中找到它。一旦有了核查过的版本，就可把它复制成% SystemRoot%System32下面的WS2_32.DLL，或把它放在自己的Winsock应用程序的工作目录中。后者是最佳选择，因为在以后的调试操作时，不必再去修改系统。已核查过的版本安装之后，必须编译并建立一个 MSDN Dt_dll示范。编译完成之后，就会收到一个支持DLL，名为Dt_dll.dll。然后，再把Dt_dll.dll复制到你的Winsock应用程序的工作目录中。一切搞定之后，便可开始对那些不是直接从你的应用程序中调用的 Winsock 2 API和SPI函数进行追踪了。

14.5 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者来实现的。通过本章的学习，大家了解了如何开发传输服务提供者和名字空间提供者。本章结尾，还对 Winsock连网技术进行了一番总结性讨论。下一章，将介绍 Microsoft Visual Basic Winsock控件。这些控件的确使用了 Winsock，但我们没有引入任何新的 Winsock概念，只指导大家如何使用VisualBasic的Winsock控件。如果你对Visual Basic不感兴趣，可转移到本书的最后一章，该章全面讲解了“ 远程访问服务 ”（RAS），这个技术大大增强了Winsock应用程序的灵活性。

第15章　微软Visual Basic Winsock控件

本章讲解微软Visual Basic Winsock控件的问题。这是一种非常新的控件，用于将 Winsock接口简化成易于使用的Visual Basic内部接口。在这种控件问世之前，要想通过 Visual Basic进行网络编程，唯一的办法便是将所有 Winsock函数都从DLL（动态链接库）中导入，然后重新定义必要的结构。而你可以想象，那些结构的数量有多少！这一过程会耗费程序员大量的时间，且极易出错。常见错误包括类型声明的错误配合等等。然而，假如你真的需要直接从Winsock导入Visual Basic，从而获取更大的灵活性，亦可参考一下本书第二部分一直都在讲述的各个Visual Basic示例。在每个Visual Basic示例中，都包含了一个文件：Winsock.bas，它负责着必要常数及函数的导入。本章的重点只放在 Visual Basic Winsock控件身上。在本章的开头，将概述控件的属性及方法，然后提供几个例子，引导大家具体使用这种控件。

第一个Winsock控件是随Visual Basic 5.0引入的。后来，随同Visual StudioService Pack 2（SP2），提供了一个经过修订的控件。后来又发布了 Service Pack 3（SP3），但该控件并未对SP2的那个版本进行什么改进。而在 Visual Basic 6.0中，我们拿到了最新的 Winsock控件。在本章要结束的时候，会对其各个版本的差异进行详细讲解。

要注意的是，Winsock控件只为Winsock API函数提供了一个基本接口。和Winsock不一样，Winsock并非一个“与协议无关”的接口，它只能使用 IP传送协议！除此以外，该控件建立在Winsock 1.1版本规范的基础上。控件同时支持 TCP以及UDP，但支持程度偏低，许多特性都无法使用。控件本身不能访问任何套接字选项。换言之，像多播及广播之类的特性便无法使用。总之，只有在你仅仅需要基本的网络数据通信的前提下，才可考虑使用 Winsock控件。它的性能无论如何都不是最好的，因为在数据传给系统之前，需要先将数据缓冲到控件内部，这样便增加了额外的开销，同时产生了一些不确定性的因素。

15.1　属性

现在，大家对控件具有的一些功能已有了一个基本概念。接下来，且让我来看看该控件提供了哪些属性。在表 15-1中，我们对可用的属性进行了一番总结。这些属性可对控件的行为产生影响，同时可用来获取与控件状态有关的信息。

通过第7章的阅读，大家应该早已熟悉了这些基本属性。它们明显类似于第　7章在客户机／服务器示例中讨论的各个基本 Winsock函数。但也有少数几个属性并非与 Winsock API存在严格的对应关系，应好好地设置它们，以便正确地使用该控件。首先要注意的是 Protocol属性，我们应设置它，告诉控件自己需要的是哪种类型的套接字 —— 要么是SOCK_STREAM（数据流），要么是SOCK_DGRAM（数据报）。在实际工作中，控件会自动执行实际的套接字创建工作，而该属性是我们唯一能对其施加控制的渠道。在连接成功建立之后，或在完成了服务器的绑定，令其等候一次连接之后，可读取 SocketHandle（套接字句柄）属性的内容。之所以要这样做，是由于在某些情况下，我们希望将句柄传递给自一个　DLL导入的其他Winsock API函数。利用State（状态）属性，我们可获取与控件目前正在做的工作有关的信息。

这种信息是至关重要的，因为控件本身是以“异步”方式工作的，事件可能在什么时刻“触发”。利用这个属性，便可确定套接字正处在一个有效的状态，可放心大胆地进行后续的操作。在表15-2中，我们对套接字各种可能的状态进行了总结，同时讲解了它们的含义。

表15-1 Winsock控件属性

属性名称	返回值	是否为只读	说 明
BytesReceived	Long	是	返回接收缓冲区内“待决”（等候处理）的字节数量。可用GetData方法来取回数据
LocalHostName	String	是	返回本地机器的名字
LocalIP	String	是	返回本地机器采用“点分十进制”格式的IP地址
LocalPort	Long	否	返回要使用的本地端口集合。若将端口设为0，表明系统需要随机性地挑选一个可用的端口。通常，只能一个客户机使用端口0
Protocol	Long	否	为控件返回或设置协议，注意控件本身支持TCP或UDP。需要设置的常数值包括sckTCPProtocol（TCP协议）以及sckUDPProtocol（UDP协议），分别对应于值0和1
RemoeHost	String	否	返回或设置远程机器的名字。既可使用字串形式的主机名，亦可使用采用“点分十进制”格式的字串表达形式
RemoteHostIP	String	是	返回远程机器的IP地址。对于TCP连接，该字段会在成功建立连接之后设置；而对于UDP连接，该字段会在产生了DataArrival（数据抵达）事件后设置，即设置之后，会包含负责发送数据的那台机器的IP地址
RemotePort	Long	否	返回或设置要连接的远程端口
SocketHandle	Long	是	返回与套接字句柄对应的一个值
State	Integer	是	返回控件状态，注意这是一个列举类型。请参考表15-2，了解各个不同的套接字常数

表15-2 套接字状态

常 数	值	含 义
sckClosed	0	默认值，表示关闭
sckOpen	1	打开
sckListening	2	正在监听连接
sckConnectionPending	3	连接请求已抵达，但尚未完成
sckResolvingHost	4	主机名正在解析
sckHostResolved	5	主机名解析已完成
sckConnecting	6	连接请求已经启动，但尚未完成
sckConnected	7	连接建立（完成）
sckClosing	8	对方正在初始化一次“关闭”
sckError	9	产生某个错误

15.2 方法

Winsock控件仅提供了少数几个方法。大多数方法名字都与它们的Winsock相似，其中仅存在两处例外，分别是GetData和SendData方法。其中，GetData用来读入待决数据。通常，一旦触发了DataArrival事件（通知我们数据已经抵达），便应调用GetData方法。另外，SendData显然用于数据的发送。除此以外，一个名为PeekData（偷看数据）的方法类似于调用Winsock函数recv，同时设定MSG_PEEK选项。但同样要注意的是，对消息的“偷看”是一种不好的

编程习惯，应尽量避免。在表15-3中，我们列出了所有可用的方法，以及它们的参数。至于这些方法本身的细节问题，将在本章稍后展示客户机和服务器示例的小节中，进行深入探讨。

表15-3 Winsock控件方法

方 法	参 数	返回值	说 明
Accept	RequestID	Void	只能用于TCP连接。处理一个ConnectionRequest事件时，用这个方法接受进入的连接请求
Bind	LocalPort，LocalIP	Void	将套接字同指定的本地端口和IP绑定到一起。假如安装有多个网络适配器（网卡），请使用Bind。Bind必须在Listen之前调用
Close	无	Void	选择连接或者监听套接字
Connect	RemoteHost，RemotePort	Void	在指定的RemotePort（远程端口）编号上，建立与指定RemoteHost（远程主机）的一个TCP连接
GetData	Data，Type，MaxLen	Void	取回当前待决的数据。Type和MaxLen参数均是可选的。Type参数定义要读入的数据的类型。MaxLen参数指定最多要取回多少字节或字符。对于除字节数组及字串以外的其他类型，GetData会忽略MaxLen参数的设定
Listen	无	Void	创建一个套接字，并将其置入监听模式。Listen只用于TCP连接
PeekData	Data，Type，MaxLen	Void	行为与GetData几乎完全一致，只是数据不会从系统缓冲区中删去
SendData	Data	Void	将数据发给远程计算机。假如传递了一个UNICODE字串，那么它会先转换成一个ANSI字串。对于二进制数据，无论如何都要使用一个字节数组

15.3 事件

“事件”是在发生了一个特定的动作后，以异步方式调用的例程。在我们编制的Visual Basic应用程序中，必须对Winsock控件可能生成的各种事件加以控制，以便顺利地使用该控件。通常，这些事件是由通信对方发起的某个行动而触发的。举个例子来说，如果TCP连接的某一方关闭了套接字，便会产生TCP“半关闭”（Half-close）事件。发起关闭的那一方会生成一个FIN信号，而另一方会用一个ACK信号作出响应，确认此次关闭请求。接收FIN的那一方会触发一个Close事件。这样一来，我们的Winsock应用程序便知道对方不再发送数据。随后，应用程序需要读入剩余的所有数据，并调用自己这一方的Close方法，将连接完全关闭掉。在表15-4中，我们总结了可能触发的各种Winsock事件，对每个事件进行了说明。

表15-4 Winsock控件事件

事 件	参 数	说 明
Close	无	远程计算机关闭连接时发生
Connect	无	Connect方法成功完成后发生
ConnectionRequest	RequestID	远程机器请求建立一个连接时发生
DataArrival	bytesTotal	新数据抵达时发生
Error	Number、Description、Scode、Source、HelpFile、HelpContext、CancelDisplay	产生一个Winsock错误时发生
SendComplete	无	发送操作完成后发生
SendProgress	bytesSent、bytesRemaining	数据正在发送时发生

15.4 UDP示例

现在，让我们来探讨一个简单的UDP应用程序。首先请看看本书配套光盘Chapter15目录下的SockUDP.vbp这个Visual Basic项目文件。该项目文件成功编译并运行之后，会看到和图15-1差不多的一个对话框。这个示范程序本身既是UDP消息的一个发送者，也是一个接收者，所以只需使用一个程序实例，便能同时实现消息的发送与接收。此外，在程序清单 15-1中，列出了窗体、按钮以及Winsock控件背后的所有代码。

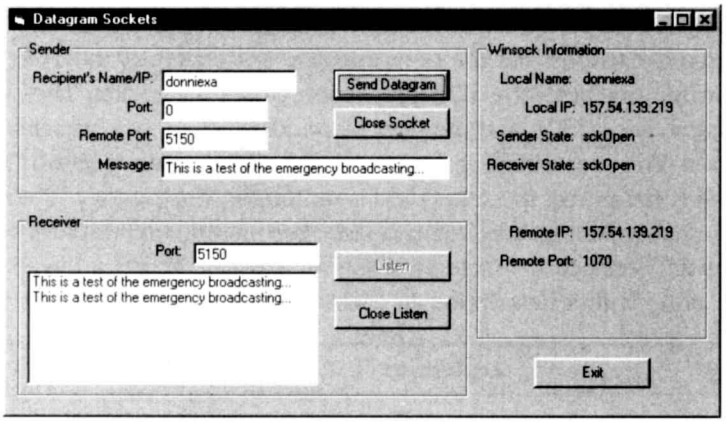

图15-1 示范UDP应用

请观察窗体，可注意到其中总共含有两个Winsock控件。一个控件用于发送数据报，另一个用于接收数据报。另外还有三个分组框：一个由发送者专用，一个由接收者专用，另一个（最右边的那个）则用于显示常规的Winsock信息。对发送者来说，需要一些地方来放置接收者的主机名或IP地址。设置RemoteHost属性时，既可使用机器文本形式的名字，亦可使用数字形式的、采用“点分十进制”格式表达的IP地址字串。如有必要，控件会自动解析出名字。另外，还需要一个远程端口，以便向其发送UDP数据包。除此以外，请注意用于本地端口的文字框（txtSendLocalPort）。对发送者来说，从哪个本地端口发出数据其实关系不大，只需注意将数据发到那个端口就可以了（目标端口）。如将本地端口设为0，系统会自动分配一个尚未使用的端口。最后一个文本框（txtSendData）用于输入要发送出去的字串。另外，请注意共有两个命令按钮：一个用于发送数据，另一个用于关闭套接字。要想发出数据报，首先必须将Winsock控件同一个远程地址、一个远程端口以及一个本地端口绑定起来，然后才能进行实际的数据发送操作。这意味着假如想更改这三个参数中的任何一个，首先便需要关闭套接字，再与新参数重新绑定到一起。这正是窗体中提供了一个Close Socket（关闭套接字）的原因。

程序清单15-1 UDP示范代码

```
Option Explicit

Private Sub cmdExit_Click()
    Unload Me
End Sub

Private Sub cmdSendDgram_Click()
```

```
    ' If the socket state is closed, we need to bind to a
    ' local port and also to the remote host's IP address and port
    If (sockSend.State = sckClosed) Then
        sockSend.RemoteHost = txtRecipientIP.Text
        sockSend.RemotePort = CInt(txtSendRemotePort.Text)
        sockSend.Bind CInt(txtSendLocalPort.Text)

        cmdCloseSend.Enabled = True
    End If
    '
    ' Now we can send the data
    '
    sockSend.SendData txtSendData.Text
End Sub

Private Sub cmdListen_Click()
    ' Bind to the local port
    '
    sockRecv.Bind CInt(txtRecvLocalPort.Text)
    '
    ' Disable this button since it would be an error to bind
    ' twice (a close needs to be done before rebinding occurs)
    '
    cmdListen.Enabled = False
    cmdCloseListen.Enabled = True
End Sub

Private Sub cmdCloseSend_Click()
    ' Close the sending socket, and disable the Close button
    '
    sockSend.Close
    cmdCloseSend.Enabled = False
End Sub

Private Sub cmdCloseListen_Click()
    ' Close the listening socket
    '
    sockRecv.Close
    ' Enable the right buttons
    '
    cmdListen.Enabled = True
    cmdCloseListen.Enabled = False
    lstRecvData.Clear
End Sub
Private Sub Form_Load()
    ' Initialize the socket protocols, and set up some default
    ' labels and values
    '
    sockSend.Protocol = sckUDPProtocol
    sockRecv.Protocol = sckUDPProtocol

    lblHostName.Caption = sockSend.LocalHostName
    lblLocalIP.Caption = sockSend.LocalIP

    cmdCloseListen.Enabled = False
    cmdCloseSend.Enabled = False
```

```
    Timer1.Interval = 500
    Timer1.Enabled = True
End Sub

Private Sub sockSend_Error(ByVal Number As Integer, _
        Description As String, ByVal Scode As Long, _
        ByVal Source As String, ByVal HelpFile As String, _
        ByVal HelpContext As Long, CancelDisplay As Boolean)
    MsgBox Description
End Sub

Private Sub sockRecv_DataArrival(ByVal bytesTotal As Long)
    Dim data As String

    ' Allocate a string of sufficient size, and get the data;
    ' then add it to the list box
    data = String(bytesTotal + 2, Chr$(0))
    sockRecv.GetData data, , bytesTotal
    lstRecvData.AddItem data
    ' Update the remote IP and port labels
    '
    lblRemoteIP.Caption = sockRecv.RemoteHostIP
    lblRemotePort.Caption = sockRecv.RemotePort
End Sub

Private Sub sockRecv_Error(ByVal Number As Integer, _
        Description As String, ByVal Scode As Long, _
        ByVal Source As String, ByVal HelpFile As String, _
        ByVal HelpContext As Long, CancelDisplay As Boolean)
    MsgBox Description
End Sub

Private Sub Timer1_Timer()
    ' When the timer goes off, update the socket status labels
    '
    Select Case sockSend.State
        Case sckClosed
            lblSenderState.Caption = "sckClosed"
        Case sckOpen
            lblSenderState.Caption = "sckOpen"
        Case sckListening
            lblSenderState.Caption = "sckListening"
        Case sckConnectionPending
            lblSenderState.Caption = "sckConnectionPending"
        Case sckResolvingHost
            lblSenderState.Caption = "sckResolvingHost"
        Case sckHostResolved
            lblSenderState.Caption = "sckHostResolved"
        Case sckConnecting
            lblSenderState.Caption = "sckConnecting"
        Case sckClosing
            lblSenderState.Caption = "sckClosing"
        Case sckError
            lblSenderState.Caption = "sckError"
        Case Else
            lblSenderState.Caption = "unknown"
    End Select
```

```
    Select Case sockRecv.State
        Case sckClosed
            lblReceiverState.Caption = "sckClosed"
        Case sckOpen
            lblReceiverState.Caption = "sckOpen"
        Case sckListening
            lblReceiverState.Caption = "sckListening"
        Case sckConnectionPending
            lblReceiverState.Caption = "sckConnectionPending"
        Case sckResolvingHost
            lblReceiverState.Caption = "sckResolvingHost"
        Case sckHostResolved
            lblReceiverState.Caption = "sckHostResolved"
        Case sckConnecting
            lblReceiverState.Caption = "sckConnecting"
        Case sckClosing
            lblReceiverState.Caption = "sckClosing"
        Case sckError
            lblReceiverState.Caption = "sckError"
        Case Else
            lblReceiverState.Caption = "unknown"
    End Select
End Sub
```

15.4.1 UDP消息的发送

现在，大家已知道了作为一个发送者，它具有哪些方面的常规能力。接下来，且让我们来看看幕后的代码。首先是 Form_Load_例程。第一步是把 socksendWinsock控件的Protocol（协议）属性设为UDP，这是利用sckUDPProtocol列举类型来完成的。除了禁用 cmdCloseSend命令按钮外，该例程中的其他命令都没有用于发送数据。我们这样做是为了完整起见，因为在一个已经关闭的控件上调用 Close（关闭）命令毫无意义。注意， Winsock控件的默认状态就是关闭的。

接下来，我们看看 cmdSendDgram_Click，单击“ 发送数据 ”按钮就会触发这一命令。这是发送UDP消息的关键。示范代码的第一步是检查套接字的状态。如果套接字处于关闭状态，代码就会把一个 UDP套接字和一个远程地址、远程端口以及本地端口绑定在一起。一旦把 UDP Winsock控件与这些参数绑定在一起，这个控件的状态就会从 sckClosed变成sckOpen。如果不执行代码检查，就试图在每一次的数据发送上绑定套接字，则会产生运行时错误 40020“ 当前状态下的操作无效 ”（Invalid operation at current state）。一旦绑定了套接字，这个套接字就会一直存在，直到关闭。这就是代码在一旦绑定 Winsock控件后，为发送套接字启用“ 关闭套接字 ”按钮的原因。最后一步是随用户打算发送的数据一起，调用 SendData方法。SendData方法返回后，代码便完成数据的发送了。

只有两个子例程和发送UDP消息相关。第一个是cmdCloseSend，名符其实的关照套接字，允许用户在再次发送数据之前，改变远程主机、远程端口或本地端口这几个参数。另一个是 sockSend_Error，它是一个Winsock事件，只要产生 Winsock错误，就会触发这一事件。由于UDP是不可靠的，所以几乎不会产生错误。即便出现了错误，代码也只是简单地显示错误说明。这个应用程序中，用户看到的仅仅只是“ 目标不可抵达 ”这个消息。

15.4.2 UDP消息的接收

众所周知，利用Winsock控件发送UDP包是非常简单而直接的。UDP包的接收更简单。我们回头看看Form_Load例程，以便了解接收UDP消息需要怎么做。从利用Winsock控件发送数据可以看出，代码把Protocol属性设为UDP。另外，还禁用了“ Close Listen ”（关闭监听）按钮。再次提醒大家注意，关闭一个已关闭的套接字毫无意义，但为了完整起见，示范代码中仍然执行了关闭。同时，在编程过程中，最好经常性地设想一下“如果这时调用 X方法，又会如何呢？”。许多程序员碰到的控件问题均由于在控件状态无效时调用方法。一个例子就是：在一个已经连接的Winsock控件上调用connect（连接）方法。

要监听接入的UDP包，先来看看cmdListen_Click例程。这是“ Listen ”（监听）按钮句柄。只需一步，便可监听接入 UDP包：在接收Winsock控件上调用Bind方法，投递用户打算在上面监听接入UDP数据报的本地端口。在监听接入 UDP数据包时，代码只需要本地端口，和发送数据的远程端口无关。在示范代码绑定接收 Winsock控件之后，就禁用了 cmdListen按钮，这样可防止用户再次单击“监听”按钮。试图绑定一个已绑定的控件会失败，出现运行时错误。

此时，sockRecv控件已注册为接收UDP数据。控件在与它绑定的端口上接收 UDP数据时，会触发DataArrival事件。这个事件是在sockRecv_DataArrival例程中执行的。投入该事件的参数，bytesTotal是可以读取的字节数。代码分配了一个字串，该字串的长度略大于正在阅读的数据长。然后，代码调用 GetGetData方法，把分配得到的字串当作第一个参数送出去。第二个参数默认为Visual Basic类型vbString，第三个参数指定需要读取的字节数，在这个例子中，是bytesTotal这个值。如果代码请求读取的字节数小于 bytesTotal参数指定的值，就会产生运行时错误。一旦把数据读入了字符缓冲区，代码就会把它增加到读取的消息列表框中。这个子例程的最后几个步骤是：为远程主机的 IP地址和端口号设置标签说明。每次收到 UDP包后，针对才收到的包，把RemoteHostIP和RemotePort属性设为远程主机的IP地址和端口号。因此，如果数据报收到多个主机发来的UDP数据包，这些属性值就会频繁发生变化。

与接收UDP消息有关的最后两个子例程是 cmdCloseListen_Click和sockRecv_Error。单击“关闭监听”按钮，就可以实行 cmdCloseListen_Click句柄了。这个例程只是在Winsock控件上简单地调用Close方法。关闭UDP控件会释放基层套接字描述符。只要出现 Winsock错误，就会触发sockRecv_Error事件。正如前一小节中提到的那样，由于 UDP本身的不可靠性，因此，有时会发生UDP错误。

15.4.3 获取Winsock信息

我们的UDP示范代码中，最后一部分是“ Winsock Information group box ”（Winsock信息组框）。本地名和本地IP标签都是在表单加载时设置的。表单加载和 Winsock控件一旦作为例示，LocalHostName和LocalIP属性就会被设为主机名和主机的 IP地址，而且，任何时候都可以读取。下面两个标签“发送端状态”和“接收端状态”，显示了应用程序所用的两个Winsock控件的当前状态。状态信息的刷新频率是每半秒一次。“计时器”控件也会进入信息中。每隔500毫秒，计时器控件便会触发一次“计时器句柄”，查询套接字状态和更新标签。我们只显示了套接字状态。最后两个标签，“远程IP”和“远程端口”，是在每次收到UDP消息时设置的，这一点，我们在前一段曾讲过。

15.4.4 运行UDP示例

现在，大家应该已经理解了如何收发 UDP消息。接下来，让我们来看看它的一个实际运行示例。要想对其进行测试，最好的办法是在三台不同的机器上分别运行该应用程序的一个实例。在其中一个程序中，点击“ Listen ”（监听）按钮。而在另外的两个程序中，请在“ Recipient’s Name/IP ”（接收者的名字／IP）文本框内输入第一个应用程序正在运行的那台机器的名字。你既可输入一个主机名，也可输入对应的 IP地址。接下来，点击几次“ Send Datagram ”（发送数据报）按钮，消息应该在接收者的消息窗口内显示出来。收到每一条消息之后，在“ Winsock Information ”（Winsock信息）区域，显示内容应发生更新，填上消息发出时所在的那个发送者的IP地址以及端口号。甚至可用与接收者相同的应用程序上的“ Sender ”（发送者）命令，在同一台机器上发送消息。

另一个有趣的测试是使用由子网定向的广播或广播数据报。假定我们的三台测试机器都位于同一个子网上，我们可将一个数据报发送到一个指定的子网，所有正在监听的应用程序都应该接收到消息。举个例子来说，在我们的测试机器中，有两个是单址机（单一宿主），IP地址分别是157.54.185.186和157.54.185.224。最后一个机器则是多址机（多宿主），它的IP地址是169.254.26.113以及157.54.185.206。可以看出，这三台机器都共享同一个子网：157.54.185.255。现在，让我们先转移一下重点，讨论一个重要的细节。如果你打算接收 UDP消息，那么在调用 Bind方法时，必须隐式地同网络绑定中保存的第一个 IP地址绑定到一起。如果你的机器只安装了一张网卡，这样做便足够了。但某些情况下，机器中可能安装了多张网卡（有多个网络接口），造成在一台机器内存在多个 IP地址。这些情况下， Bind方法的第二个参数必须设为准备绑定的具体 IP地址。不幸的是， Winsock控件属性LocalIP只能返回一个IP地址。此外，控件本身并没有提供其他方法来取得同本地机器关联在一起的其他 IP地址。

现在，且让我们来试试广播通信。首先关闭在每个应用程序运行实例上的发送或监听套接字。在两台单址机上，请点击“ Listen ”（监听）按钮，使每台机器都能接收到数据报消息。在此，我们不打算使用那台多址机，这是由于在代码中，我们尚未同任何具体的 IP地址绑定到一起。在第三台机器上，请输入接收者的地址：157.54.185.255，然后点击几次“ Send Data ”（发送数据）按钮。随后，应发现两个监听应用程序都会收到消息。假如负责发送的机器也是一个“ 多址机 ”，那么大家可能会感到不解，它如何决定数据报该从哪个网络接口发送出去呢？事实上，路由表在这里会发挥至关重要的作用，它可以决定用于发送消息的最佳接口，只要事先知道了消息的目标地址以及本地机器上每个接口的地址。我们要进行的最后一次测试是关闭第三台机器上的发送者套接字。请将接收者的地址设为 255.255.255.255，然后点击几次“ Send Datagram ”（发送数据报）按钮。其结果应该是相同的：另外两个监听程序会接收到消息。然而，对多址机来说，此时存在的一个区别是， UDP消息已经广播到同机器连接的每一个网络上了。

15.4.5 UDP状态

现在，对于到底采用哪种方法调用顺序，以便成功地收发数据报，大家可能有些迷惑不解。如前面所述，用 Winsock控件编程时，最容易犯的错误是：调用一个方法，但那个方法的操作对控件当前的状态来说却是无效的。为帮助大家避免此类错误，请看图 15-2。该图演示了使用 UDP消息时，套接字状态的变化情况。请注意，默认的起始状态无论如何都是

sckClosed。而且对于无效的主机名来说，不会产生任何错误。

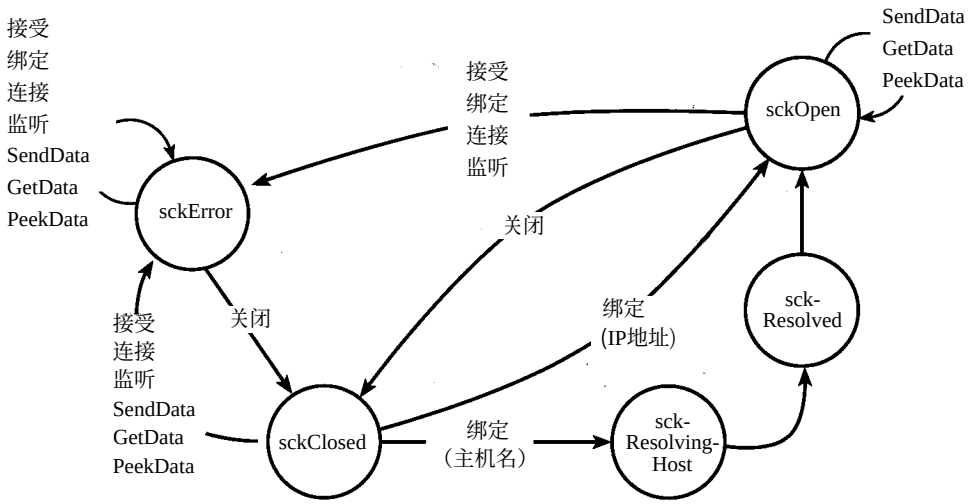

图15-2 UDP状态示意图

15.5 TCP示例

假如将Winsock控件用于TCP协议的处理，那么和处理UDP时比较起来，程序员需要做更多的工作，整个过程显得比较复杂。和我们处理 UDP时一样，首先会向大家展示一个示范性的TCP程序。接下来，讲述该程序的一些规格，以便对成功使用 TCP连接所涉及到的步骤有一个大致了解。在图 15-3中，我们展示了这个程序运行时的样子。程序清单 15-2则列出了该程序的源代码。在本书配套光盘的第 15章目录下，大家也可找到该Visual Basic项目文件的源码，文件名是SockTCP.vbp。

首先来观察一下图 15-3的窗体布局，对该程序的功能有一个大致的印象。同样，其中有三个分组框：TCP Server（TCP服务器）、TCP Client（TCP客户机）以及Winsock Information（Winsock信息）。首先让我们来看看该程序的 TCP服务器部分。服务器提供了一个文本框，名为txtServerPort，用于指定服务器打算与之绑定的本地端口，以便监听进入的客户机连接请求。此外，服务器还提供了两个按钮。一个用于将服务器置于监听模式，另一个用于关闭服务器，并停止接受进入连接。最后，服务器还提供了一个 Winsock控件，名为sockServer。观察一下属性页，便会发现Index属性被设为0。这意味着，该控件实际上是一个数组，可容纳 Winsock控件的多个实例。0表示在窗体载入时，只会创建它的一个实例（亦即数组的元素 0）。在任何时候，我们都可在一个数组元素中动态载入 Winsock控件的另一个实例。

Winsock控件数组实际上是发挥服务器功能至关重要的一个环节。请记住，每个 Winsock控件只有一个套接字句柄同它关联在一起。通过第 7章的学习，大家已经知道，服务器接受一个进入连接时，会新建一个套接字，以便对那个连接进行控制。对我们的应用程序来说，它的设计宗旨是：在建好一个客户机连接后，能够动态装载额外的 Winsock控件，以便连接能够传给新载入的控件，同时不会打断服务器套接字对连接的控制。要想达到这一目的，另一个方法是在程序设计的时候，将数量为 x的Winsock控件实际地置入窗体。然而，这样做的效率显然非常低下，以后也不易于扩展。应用程序启动时，必须耗费大量的时间来装载每个控件

所需的资源。此外，还必须解决具体打算使用多少个控件的问题。放置了 x个控件之后，能够同时连接的客户机数量便限定在 x之内。假如应用程序只允许同时建立固定数量的客户机连接，那么事先在窗体中设置固定数量的 Winsock控件，便显得较为合理，而且可能比使用数组简单得多。然而，对多数应用程序来说， Winsock控件数组是最佳的编程方案。

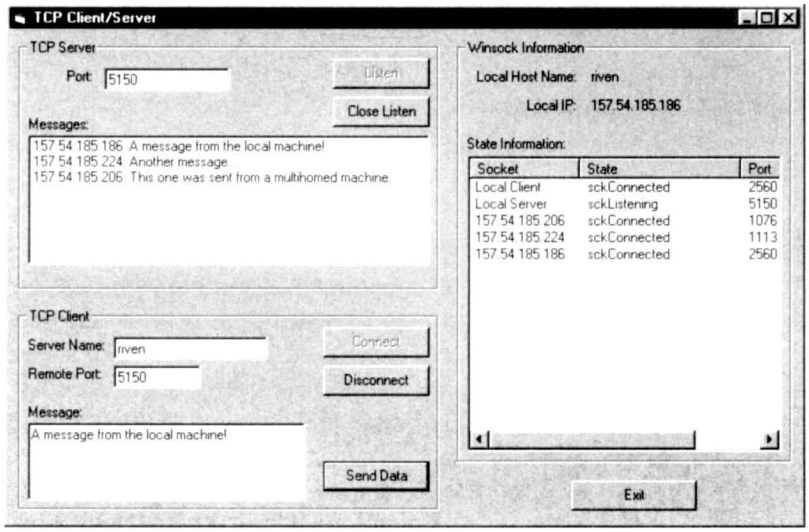

图15-3 示范TCP程序

程序清单15-2 示范TCP程序

```
Option Explicit

' The index value of the last Winsock control dynamically loaded
' in the sockServer array
Private ServerIndex As Long

Private Sub cmdCloseListen_Click()
    Dim itemx As Object
    ' Close the server's listening socket. No more
    ' clients will be allowed to connect.
    '
    sockServer(0).Close
    cmdListen.Enabled = True
    cmdCloseListen.Enabled = False

    Set itemx = lstStates.ListItems.Item(2)
    itemx.SubItems(2) = "-1"
End Sub

Private Sub cmdConnect_Click()
    ' Have the client control attempt to connect to the
    ' specified server on the given port number
    '
    sockClient.LocalPort = 0
    sockClient.RemoteHost = txtServerName.Text
    sockClient.RemotePort = CInt(txtPort.Text)
    sockClient.Connect
```

```
    cmdConnect.Enabled = False
End Sub

Private Sub cmdDisconnect_Click()
    Dim itemx As Object
    ' Close the client's connection and set up the command
    ' buttons for subsequent connections
    '
    sockClient.Close

    cmdConnect.Enabled = True
    cmdSendData.Enabled = False
    cmdDisconnect.Enabled = False
    ' Set the port number to -1 to indicate no connection
    '
    Set itemx = lstStates.ListItems.Item(1)
    itemx.SubItems(2) = "-1"
End Sub

Private Sub cmdExit_Click()
    Unload Me
End Sub
Private Sub cmdListen_Click()
    Dim itemx As Object
    ' Put the server control into listening mode on the given
    ' port number
    '
    sockServer(0).LocalPort = CInt(txtServerPort.Text)
    sockServer(0).Listen

    Set itemx = lstStates.ListItems.Item(2)
    itemx.SubItems(2) = sockServer(0).LocalPort

    cmdCloseListen.Enabled = True
    cmdListen.Enabled = False
End Sub

Private Sub cmdSendData_Click()
    ' If we're connected, send the given data to the server
    '
    If (sockClient.State = sckConnected) Then
        sockClient.SendData txtSendData.Text
    Else
        MsgBox "Unexpected error! Connection closed"
        Call cmdDisconnect_Click
    End If
End Sub

Private Sub Form_Load()
    Dim itemx As Object

    lblLocalHostname.Caption = sockServer(0).LocalHostName
    lblLocalHostIP.Caption = sockServer(0).LocalIP

    ' Initialize the Protocol property to TCP since that's
    ' all we'll be using
    '
    ServerIndex = 0
```

```
    sockServer(0).Protocol = sckTCPProtocol
    sockClient.Protocol = sckTCPProtocol
    ' Set up the buttons
    '
    cmdDisconnect.Enabled = False
    cmdSendData.Enabled = False
    cmdCloseListen.Enabled = False
    ' Initialize the ListView control that contains the
    ' current state of all Winsock controls created (not
    ' necessarily connected or being used)
    '
    Set itemx = lstStates.ListItems.Add(1, , "Local Client")
    itemx.SubItems(1) = "sckClosed"
    itemx.SubItems(2) = "-1"
    Set itemx = lstStates.ListItems.Add(2, , "Local Server")
    itemx.SubItems(1) = "sckClosed"
    itemx.SubItems(2) = "-1"
    ' Initialize the timer, which controls the rate of refresh
    ' on the above socket states
    '
    Timer1.Interval = 500
    Timer1.Enabled = True
End Sub

Private Sub sockClient_Close()
    sockClient.Close
End Sub

Private Sub sockClient_Connect()
    Dim itemx As Object

    ' The connection was successful: enable the transfer data
    ' buttons
    cmdSendData.Enabled = True
    cmdDisconnect.Enabled = True

    Set itemx = lstStates.ListItems.Item(1)
    itemx.SubItems(2) = sockClient.LocalPort
End Sub

Private Sub sockClient_Error(ByVal Number As Integer, _
        Description As String, ByVal Scode As Long, _
        ByVal Source As String, ByVal HelpFile As String, _
        ByVal HelpContext As Long, CancelDisplay As Boolean)
    ' An error occured on the Client control: print a message,
    ' and close the control. An error puts the control in the
    ' sckError state, which is cleared only when the Close
    ' method is called.
    MsgBox Description
    sockClient.Close
    cmdConnect.Enabled = True
End Sub

Private Sub sockServer_Close(index As Integer)
    Dim itemx As Object
    ' Close the given Winsock control
```

```
    '
    sockServer(index).Close

    Set itemx = lstStates.ListItems.Item(index + 2)
    lstStates.ListItems.Item(index + 2).Text = "---.---.---.---"
    itemx.SubItems(2) = "-1"

End Sub

Private Sub sockServer_ConnectionRequest(index As Integer, _
        ByVal requestID As Long)
    Dim i As Long, place As Long, freeSock As Long, itemx As Object

    ' Search through the array to see wether there is a closed
    ' control that we can reuse
    freeSock = 0
    For i = 1 To ServerIndex
        If sockServer(i).State = sckClosed Then
            freeSock = i
            Exit For
        End If
    Next i
    ' If freeSock is still 0, there are no free controls
    ' so load a new one
    '
    If freeSock = 0 Then
        ServerIndex = ServerIndex + 1
        Load sockServer(ServerIndex)

        sockServer(ServerIndex).Accept requestID
        place = ServerIndex
    Else
        sockServer(freeSock).Accept requestID
        place = freeSock
    End If
    '  If no free controls were found, we added one above.
    '  Create an entry in the ListView control for the new
    '  control.  In either case set the state of the new
    '  connection to sckConnected.
    '
    If freeSock = 0 Then
        Set itemx = lstStates.ListItems.Add(, , _
            sockServer(ServerIndex).RemoteHostIP)
    Else
        Set itemx = lstStates.ListItems.Item(freeSock + 2)
        lstStates.ListItems.Item(freeSock + 2).Text = _
            sockServer(freeSock).RemoteHostIP
    End If
    itemx.SubItems(2) = sockServer(place).RemotePort

End Sub

Private Sub sockServer_DataArrival(index As Integer, _
        ByVal bytesTotal As Long)
    Dim data As String, entry As String

    ' Allocate a large enough string buffer and get the
    ' data
    '
    data = String(bytesTotal + 2, Chr$(0))
    sockServer(index).GetData data, vbString, bytesTotal
    ' Add the client's IP address to the beginning of the
```

```
    ' message and add the message to the list box
    '
    entry = sockServer(index).RemoteHostIP & ": " & data
    lstMessages.AddItem entry
End Sub

Private Sub sockServer_Error(index As Integer, _
        ByVal Number As Integer, Description As String, _
        ByVal Scode As Long, ByVal Source As String, _
        ByVal HelpFile As String, ByVal HelpContext As Long, _
        CancelDisplay As Boolean)
    ' Print the error message and close the specified control.
    ' An error puts the control in the sckError state, which
    ' is cleared only when the Close method is called.
    MsgBox Description
    sockServer(index).Close
End Sub

Private Sub Timer1_Timer()
    Dim i As Long, index As Long, itemx As Object

    ' Set the state of the local client Winsock control
    '
    Set itemx = lstStates.ListItems.Item(1)
    Select Case sockClient.State
        Case sckClosed
            itemx.SubItems(1) = "sckClosed"
        Case sckOpen
            itemx.SubItems(1) = "sckOpen"
        Case sckListening
            itemx.SubItems(1) = "sckListening"
        Case sckConnectionPending
            itemx.SubItems(1) = "sckConnectionPending"
        Case sckResolvingHost
            itemx.SubItems(1) = "sckResolvingHost"
        Case sckHostResolved
            itemx.SubItems(1) = "sckHostResolved"
        Case sckConnecting
            itemx.SubItems(1) = "sckConnecting"
        Case sckConnected
            itemx.SubItems(1) = "sckConnected"
        Case sckClosing
            itemx.SubItems(1) = "sckClosing"
        Case sckError
            itemx.SubItems(1) = "sckError"
        Case Else
            itemx.SubItems(1) = "unknown: " & sockClient.State
    End Select
        ' Now set the states for the listening server control as
        ' well as any connected clients
        '
        index = 0
        For i = 2 To ServerIndex + 2
            Set itemx = lstStates.ListItems.Item(i)

            Select Case sockServer(index).State
                Case sckClosed
                    itemx.SubItems(1) = "sckClosed"
                Case sckOpen
                    itemx.SubItems(1) = "sckOpen"
                Case sckListening
```

```
                itemx.SubItems(1) = "sckListening"
            Case sckConnectionPending
                itemx.SubItems(1) = "sckConnectionPending"
            Case sckResolvingHost
                itemx.SubItems(1) = "sckResolvingHost"
            Case sckHostResolved
                itemx.SubItems(1) = "sckHostResolved"
            Case sckConnecting
                itemx.SubItems(1) = "sckConnecting"
            Case sckConnected
                itemx.SubItems(1) = "sckConnected"
            Case sckClosing
                itemx.SubItems(1) = "sckClosing"
            Case sckError
                itemx.SubItems(1) = "sckError"
            Case Else
                itemx.SubItems(1) = "unknown"
        End Select
        index = index + 1
    Next i
End Sub
```

15.5.1 TCP服务器

从现在开始，我们打算为大家讲解窗体后面的代码。先来看看程序清单 15-2显示的Form_Load例程。头两条语句的作用很简单，将两个标签分别设为本地机器的主机名和IP地址。这些标签显示于Winsock信息分组框内，它们的作用和UDP示例中的信息框完全相同。接下来，我们将服务器控件sockServer初始化成TCP协议。Winsock控件数组的元素0肯定对应于监听套接字。在这以后，进程会先将 Close Listen（关闭监听）按钮禁用。稍后，等服务器开始监听客户机连接时，该按钮又会启用。这个进程的最后部分负责 ListView控件（lstStates）的设置。该控件用于显示当前使用的每一个 Winsock控件的最新状态。代码会为客户机和服务器控件添加对应的条目，令其分别成为数组元素 1和2。以后假如动态载入了其他任何 Winsock控件，都会添加到这两个元素之后。与服务器控件对应的条目名为“ Local Server ”（本地服务器）。和在UDP例子中一样，进程也会设置一个计时器，对套接字状态的更新频率进行管理。在默认情况下，计时器每隔半秒钟，便刷新一道显示。

现在，让我们来看看服务器使用的两个按钮。第一个是 Listen（监听）按钮，它的用法非常简单。Listen按钮的控制模块 cmdListen负责将LocalPort（本地端口）属性设为用户在txtServerPort文本框内输入的值。对监听套接字来说，本地端口是最重要的一项设定。所有客户机都会通过这个端口尝试建立与服务器的连接。设好 LocalPort属性后，代码下一步要做的唯一的一件事情是调用 Listen方法。Listen按钮的控制模块将sockServer控件置于监听模式后，程序会开始等待在我们的sockServer控件上发生ConnectionRequest事件 —— 表明有一个客户机请求连接。用户可点击另一个按钮（Close Listen），以关闭sockServer控件。Close Listen按钮的控制模块会针对sockServer(0)调用Close方法，防止以后再进行任何客户机连接。

对TCP服务器来说，最重要的事件是 ConnectionRequest，它负责对进入的客户机请求的控制。若客户机请求建立一个连接，那么可选择两种方法来处理这一请求。第一种方法，可用服务器套接字本身来控制客户机。但这样做也有缺点，因为它会关闭监听套接字，并禁止

再为其他任何连接提供服务。要想采用这种方法，只需针对拥有特定 requestID的服务器控件调用Accept方法。那个requestID早已传递到事件控制模块内。控制客户机连接请求的另一个方法是将连接传递给某个单独的控件，这正是我们的示范 SockTCP程序所采用的。要记住的是，我们有一个由Winsock控件构成的数组，而且元素 0对应于监听套接字。要做的第一件事是搜索整个数组，在其中寻找状态为“ 关闭 ”的控件。举个例子来说，可搜索值为 sckClosed的State属性。当然，在第一次循环中，不可能找到空闲的控件，因为它们尚未载入。在这种情况下，大家会发现第一次循环根本没有执行。同时，变量 freeSock仍然为0，表明没有找到空闲的控件。接下来的步骤是动态载入一个 Winsock控件。具体做法是递增 ServerIndex计数器的值（对应于载入控件的数组位置），然后执行下述语句：

```
Load sockServer(ServerIndex)
```

现在，一个新的Winsock控件已经载入了。接下来，进程可随同指定的请求ID（requestID），调用Accept方法。剩下的语句会在lstStates这个ListView（列表视图）控件中加入一个新条目，使程序能够显示出新Winsock控件的最新状态。

假如已经存在一个加载的 Winsock控件，其状态已为“ 关闭 ”，那么进程只需简单地重新使用那个控件。具体说来，就是在它上面调用 Accept方法。经常加载和卸载控件并不是一种好的编程习惯，因为它会对性能产生一定的影响。某些时候，加载和卸载操作会让你付出惨重代价。在Winsock控件卸载时，还存在一个“ 内存渗漏 ”的问题。本章稍后将就此进行详述。

剩下的服务器函数非常简单。只要有客户机在自己那一端调用 Close方法，就会触发sockServer_Close事件。此时，服务器要做的事情只是关闭自己这一端的套接字，同时清除ListView控件内的 IP地址条目（将其设为“ ---.---.---.--- ”，并将条目的端口设为- 1）。sockServer_DataArrival函数会分配一个缓冲区，以便接收数据。随后，调用 GetData方法，来实际执行读操作。之后，消息会添加到 lstMessages列表框内。最后一个服务器函数是 Error事件控制模块。一旦发生错误，该控制模块就会显示出文本消息，随后关闭控件。

15.5.2 TCP客户机

目前为止，大家已知道了怎样实现服务器。接下来，看看客户机的情况。在 Form_Load进程中，客户机执行的唯一初始化操作是将 sockClient的协议设为TCP。除初始化代码之外，还有三个从属于客户机的命令按钮控制模块，以及几个事件控制模块。其中，第一个按钮是Connect（连接），它的控制模块名为cmdConnect_Click。LocalPort设为0，因为无论本地系统分配哪一个端口编号，都不重要。 RemoteHost和RemotePort则分别根据txtServerName和txtPort字段中的值来设定。这些便是与服务器建立 TCP连接所需的所有信息。万事俱备，剩下的事便是调用 Connect方法。调用这个方法后，控件状态便是“ 正在解析名字 ”或“ 连接 ”（控件的状态是sckResolvingHost、sckResolved或sckConnecting）。终于建立连接之后，状态变成“ sckConnected ”（已连接），并触发Connect（连接）事件。下一小节将全面讲解各种连接状态以及各状态之间的转换。

连接一旦建立，便调用 sockClient_Connect句柄。这个句柄只是启用了“ SendData ”（发送数据）和“ Disconnect ”（取消连接）两个按钮。除此以外，本地机器上建立了连接的那个端口编号已针对lstStates ListView控件中的“ 本地客户机条目 ”而得以更新。现在便可收发数

据。另外还有两个事件句柄：sockClient_Close和sockClient_Error。sockClient_Close事件句柄只是关闭客户机Winsock控件，而sockClient_Error事件句柄则显示一个具有错误说明的消息框，之后关闭这个控件。

关于客户机，最后需要提到的是这两个命令按钮： Send Data（发送数据）和Disconnect（取消连接）。子例程cmdSendData_Click对Send Data（发送数据）按钮进行控制。如果已连接了Winsock控件，这个例程便随txtSendData文本框内的字串一起，调用SendData方法。最后，Disconnect（取消连接）按钮由cmdDisconnect_Click控制。这个句柄只关闭客户机控件，重新把一系列按钮设置成初始状态，并更新lstStates ListView控件中的Local Client（本地客户机）条目。

15.5.3 获取Winsock信息

TCP示例的最后部分是Winsock信息。我们在以前曾对此进行了解释，但为了使大家心里更清楚，还需要简明扼要地讲一讲。在UDP示例中，对所有已加载Winsock控件来说，当前套接字状态的更新是计时器触发的。默认刷新时间是500毫秒。加载之后的lstStates ListView控件中增加了两个条目。第一个条目是Local Client（本地客户机），它对应于客户机Winsock控件sockClient。第二个条目是LocalServer（本地服务器），它引用正在监听的套接字。只要建立了新的客户机连接，就会动态加载一个新的Winsock控件，sckStates控件中也会增加一个新条目；条目名就是该客户机的IP地址。客户机取消连接时，把IP地址设为“ ---.---.---.--- ”，端口号设为-1，便将条目设为默认状态。当然，如果有另一个客户机连接时，便重新使用服务器数组中没有用过的控件。同时显示本地机器的IP地址和主机名。

15.5.4 运行TCP示例

运行TCP示例非常简单。启动三个应用程序实例，一台机器一个。利用TCP时，机器是否是多址机这一点无关紧要，因为路由表会判断哪个接口更适合这个具体的 TCP连接。其中一个TCP示例中，单击Listen（监听）按钮，启动监听套接字。大家将注意到 State Information ListView（状态信息列表视图）中的Local Server（本地服务器）条目从sckClosed变成了sckListening，列出的端口号是5150。现在起，服务器准备接受客户机连接。

其中一个客户机上，先把Server Name（服务器名）字段设为运行第一个应用程序实例的那台计算机（监听服务器）的名字，再单击Connect（连接）按钮。客户机应用程序上， State Information（状态信息列表）中的Local Client（本地客户机）条目现在处于sckConnected状态，建立连接的本地端口之端口号更新为一个“非负值”。另外，服务器这一端，状态信息列表中增加了一个条目，该列表名为刚连接的客户机的IP地址。新增条目的状态是sckConnected，其中还包含了服务器这一端上已建立连接的那个端口的编号。现在，可以向客户机上的Message（消息）文本域内输入文字，并单击几次Send Data（发送数据）按钮。大家将看到出现在服务器端的Message（消息）文本域内的消息。接下来单击 Disconnect按钮取消这个客户机连接。在客户机端， State Information（状态信息）列表框内的Local Client（本地客户机）条目设为原来的sckClosed，端口号设为-1。对服务器来说，与客户机对应的那个条目没有被删除；只用这样的方式来表示未用：一个名字设为虚线 IP地址，状态设为sckClosed，端口号值设为-1。

第三台机器上，在Server Name（服务器名）文本框内输入监听套接字的名字，建立一个客户机连接。其结果与第一个客户机一样，只不过这个服务器用同样的 Winsock控件，像第一台机器上那样，对这个客户机进行处理。如果 Winsock控件处于已关闭状态，就可用于接受任何一个接入连接。最后一步是利用服务器应用程序上的客户机建立本地连接。连接建立之后，Socket Information（套接字信息）列表中便增加一个新条目，这和前面的示例一样。唯一的区别是：这时列出的 IP和服务器的 IP地址一样。实战多客户机和一个服务器的确能深入了解它们之间是如何交互的，以其相关命令会触发什么样的结果。

15.5.5 TCP状态

使用TCP协议的Windows控件比UDP套接字复杂得多，因为TCP涉及到的套接字状态更多。图15-4是一个TCP套接字的状态图。默认的起始状态是 sckClosed。各个状态之间的转换非常直接，无需过多解释，但 sckClosing状态除外。由于TCP的半关闭特性，在使用 SendData方法时，便有两条转换途径。 TCP连接的一端执行一个 Close（关闭）方法，不能再发送数据了。连接的另一端收到了这个 Close事件，置入 sckClosing状态，但仍然可以接收数据。这就是SendData方法的sckClosing状态转换有两条途径的原因。如果执行 Close方法的这端试图调用SendData，就会产生错误，并且状态转为 sckError。收到Close事件的一端可自由发送数据和接收其余的数据。

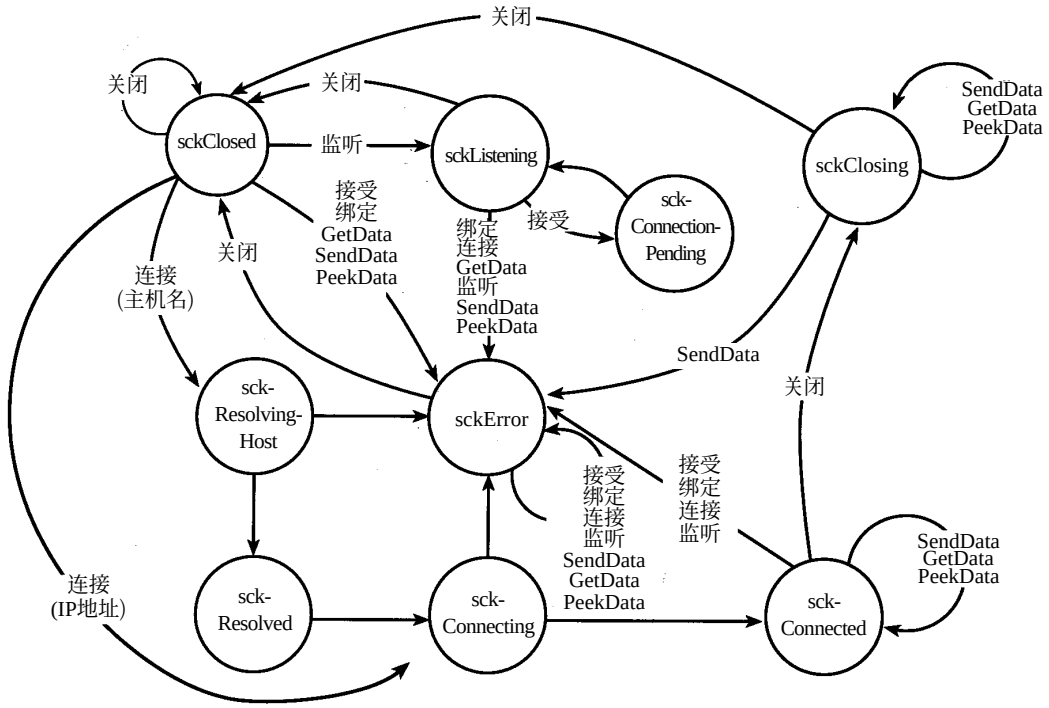

图15-4 TCP状态

15.6 存在的局限

Winsock控件用处颇为广泛，而且易于使用；但不幸的是，还存在几个错误，令控件不能

使用执行关键任务的应用程序。本小节中讨论的错误出现在最新版本的 Visual Basic 5.0控件，这个控件是在SP 2基础上更新的。

第一个错误问题不大，涉及到动态加载和卸载控件。卸载一个以前加载的控件时，会导致内存渗漏。这就是我们在服务器示例中进行客户机连接和取消连接时，没有加载和卸载控件的原因。控件一旦载入内存，我们会尽可能地不卸载它，留给另外的客户机使用。

第二个错误是在将队列中的所有数据发送进入网络之前，关闭套接字连接造成的。某些情况下，若在SendData事件之后调用Close方法（Close的处理先于SendData时），会导致数据的丢失，至少从接收者的角度来看时如此。通过捕捉 SendComplete事件（SendData完全将数据送入网络之后触发的）的方式，便可避免这个问题。另外一个解决办法是，可通过对发送／接收事务处理的特殊安排，使接收者在收到自己希望的所有数据时，能够先一步执行Close命令。随后便会触发发送者上的Close事件，后者随之通知自己发出的所有数据均已被对方收到。现在，可以非常安全地将连接关闭。

最后一个、同时也是最严重的错误是：提交一个大缓冲区时造成数据的丢失。假如一个足够大的数据块正在队列之中，等候进入网络传输，那么控件的内部缓冲区有可能变得一团糟，造成部分数据的丢失。不幸的是，没有一种完美的方案可有效解决这个问题。最好的办法是每次提交不超过1000字节的数据块。提交了一个缓冲区之后，便等待 SendComplete（发送完成）事件的发生，再提交下一个缓冲区。尽管这样做十分不便，但确实是提高控件可靠性的最佳方式。

随Visual Basic 6.0发布的最新Winsock控件修正了上述除第二个之外的所有问题。如在调用SendData后马上执行一个Close命令，那么套接字会立即关闭，根本不会等待全部数据都发送完毕。当然，如果能将所有问题都解决，便显得更加完美了。但是，剩下的这个问题是三个问题中最不严重的，而且最容易解决。

15.7 常见错误

通过第6到第14章的学习，大家已经知道，一个应用程序可能会经历大量 Winsock错误。在此，我们不打算将所有问题都总结一遍。在接下去的两个小节中，我们只准备介绍使用Winsock控件的时候，最常遇到的两个错误是“ 本地地址正在使用 ”以及“ 当前状态下的无效操作 ”。

1. 本地地址正在使用

“ 本地地址正在使用 ”错误是在同一个本地端口绑定时发生的。要想同本地端口建立绑定关系，要么可以使用 Bind方法，要么可以使用 Connect方法。它表明要绑定的那个端口正在使用。这种现象通常见于 TCP服务器中，那些服务器一直同一个本地端口绑定在一起，使客户机能够随时定位（找到）服务。假定在应用程序使用一个套接字之前，那个套接字被不正确地关闭，它就会短时间内处于TIME_WAIT状态，以确保所有数据都在那个端口上完成了收发。在这期间，若试图建立同那个端口的绑定，便会产生所谓的“ 本地地址正在使用 ”错误。另外，在客户机那一端，一个常邮的误操作也会造成这样的错误。假若将 LocalPort属性的值设为0，然后建立了一个连接，那么 LocalPort会自动更新为用于建立本地客户机连接的那个端口号。若计划重新使用相同的控件，来进行后续的连接，那么一定要记住重设 LocalPort属性，将其变成0。否则的话，一旦以前的连接没有正确关闭，便会遇到这样的错误。

2. 当前状态下的无效操作

“当前状态下的无效操作”（Invalid operation at current state）可能是最常见的一种错误。若调用一个Winsock控件方法，但取决于控件的当前状态，却要求禁止那样的操作，便会产生这样的错误。大家可参考一下图15-2和图15-4，观察UDP和TCP套接字的结构。要想写出真正“健壮”的代码，在调用一个方法之前，一定记住先检查一下套接字的状态。

Winsock错误会通过Error事件生成。这些错误等同于直接用Winsock编程时遇到的那些错误。要想了解Winsock错误的详情，请参考第7章，其中讲述了最常见的一系列错误；亦可参考附录C，那里列出了可能出现的所有Winsock错误代码。

15.8 Windows CE的Winsock控件

在Visual Basic Toolkit for Windows CE（VBCE）中，包含了一个Winsock控件，提供了与普通Visual Basic Winsock控件差不多的功能。两者最主要的差别在于，尽管UDP未获支持，但Windows CE的Winsock控件提供了IrDA（红外线通信联盟）协议。此外，由于两者存在的一些细微差异，促使我们在编程的时候，同非Windows CE的Winsock控件编程相比，需要作出一些变化。

通过第7章的学习，大家知道Windows CE没有提供异步Winsock模型。在此，Windows CE的Winsock控件也不例外。就编程来说，最主要的区别在于Connect方法工作于“锁定”模式。此时不再有Connect事件。一旦调用Connect，试图建立一个连接，调用便会被暂时挂起，直至建好连接，或者返回一个错误。

除此以外，VBCE 1.0没有提供对控件数组的支持。换言之，我们必须更改在程序清单15-2中显示的服务器设计。因此，要想同时控制多个连接，唯一的方法便是在窗体中置入大量Windows CE Winsock控件。在实际应用中，这样做便限制了能够同时建立的客户机连接的数量，造成整个应用程序的扩展能力大打折扣。

最后，ConnectionRequest事件不再有RequestID参数，这让人感觉似乎非常奇怪。为此，我们必须在控件上调用Accept方法。触发ConnectionRequest事件的连接请求是由负责接收连接请求的控件加以满足的。

15.8.1 Windows CE Winsock示例

本节介绍一个示范应用程序，它运用了Windows CEWinsock控件。除前面提到的区别之外，桌面机Winsock控件的设计原理同样适用于Windows CE控件的设计。在程序清单15-3中，我们展示了Windows CE Winsock控件背后的一系列代码。

程序清单15-3 Windows CE Winsock示例

```
Option Explicit

' This global variable is used to retain the current value
' of the radio buttons.  0 corresponds to TCP, while 2 means
' IrDA (infrared).  Note that UDP is not supported by the
' control currently.
Public SocketType

Private Sub cmdCloseListen_Click()
' Close the listening socket, and set the other buttons
```

```
' back to the start state
    WinSock1.Close

    cmdConnect.Enabled = True
    cmdListen.Enabled = True
    cmdDisconnect.Enabled = False
    cmdSendData.Enabled = False
    cmdCloseListen.Enabled = False
End Sub

Private Sub cmdConnect_Click()
' Check which type of socket type was chosen, and initiate
' the given connection

    If SocketType = 0 Then
        ' Set the protocol and the remote host name and port
        '
        WinSock1.Protocol = 0
        WinSock1.RemoteHost = txtServerName.Text
        WinSock1.RemotePort = CInt(txtPort.Text)
        WinSock1.LocalPort = 0

        WinSock1.Connect
    ElseIf SocketType = 2 Then
        ' Set the protocol to IrDA, and set the service name
        '
        WinSock1.Protocol = 2
        'WinSock1.LocalPort = 0
        'WinSock1.ServiceName = txtServerName.Text
        WinSock1.RemoteHost = txtServerName.Text

        WinSock1.Connect
    End If
    ' Make sure the connection was successful; if so,
    ' enable/disable some commands
    '
    MsgBox WinSock1.State
    If (WinSock1.State = 7) Then
        cmdConnect.Enabled = False
        cmdListen.Enabled = False
        cmdDisconnect.Enabled = True
        cmdSendData.Enabled = True
    Else
        MsgBox "Connect failed"
        WinSock1.Close
    End If
End Sub

Private Sub cmdDisconnect_Click()
' Close the current client connection, and reset the
' buttons to the start state
    WinSock1.Close

    cmdConnect.Enabled = True
    cmdListen.Enabled = True
```

```
    cmdDisconnect.Enabled = False
    cmdSendData.Enabled = False
    cmdCloseListen.Enabled = False
End Sub

Private Sub cmdListen_Click()
' Set the socket to listening mode for the given protocol
' type
'
    If SocketType = 0 Then
        WinSock1.Protocol = 0
        WinSock1.LocalPort = CInt(txtLocalPort.Text)
        WinSock1.Listen
    ElseIf SocketType = 2 Then
        WinSock1.Protocol = 2
        WinSock1.ServiceName = txtServerName.Text
        WinSock1.Listen
    End If
    ' If we're not in listening mode now, something
    ' went wrong
    '
    If (WinSock1.State = 2) Then
        cmdConnect.Enabled = False
        cmdListen.Enabled = False
        cmdCloseListen.Enabled = True
    Else
        MsgBox "Unable to listen!"
    End If
End Sub

Private Sub cmdSendData_Click()
' Send the data in the box on the current connection
'
    WinSock1.SendData txtSendData.Text
End Sub

Private Sub Form_Load()
' Set the initial values for the buttons, the timer, etc.
'
    optTCP.Value = True
    SocketType = 0

    Timer1.Interval = 750
    Timer1.Enabled = True

    cmdConnect.Enabled = True
    cmdListen.Enabled = True

    cmdDisconnect.Enabled = False
    cmdSendData.Enabled = False
    cmdCloseListen.Enabled = False

    lblLocalIP.Caption = WinSock1.LocalIP
End Sub
```

```
Private Sub optIRDA_Click()
' Set the socket type to IrDA
'
    optIRDA.Value = True
    SocketType = 2
End Sub

Private Sub optTCP_Click()
' Set the socket type to TCP
'
    optTCP.Value = True
    SocketType = 0
    cmdConnect.Caption = "Connect"
End Sub

Private Sub Timer1_Timer()
' This is the event that gets called each time the
' timer expires. Update the socket state label.
'
    Select Case WinSock1.State
        Case 0
            lblState.Caption = "sckClosed"
        Case 1
            lblState.Caption = "sckOpen"
        Case 2
            lblState.Caption = "sckListening"
        Case 3
            lblState.Caption = "sckConnectionPending"
        Case 4
            lblState.Caption = "sckResolvingHost"
        Case 5
            lblState.Caption = "sckHostResolved"
        Case 6
            lblState.Caption = "sckConnecting"
        Case 7
            lblState.Caption = "sckConnected"
        Case 8
            lblState.Caption = "sckClosing"
        Case 9
            lblState.Caption = "sckError"
    End Select
End Sub

Private Sub WinSock1_Close()
' The other side initiated a close, so we'll close our end.
' Reset the buttons to their initial state.
'
    WinSock1.Close

    cmdConnect.Enabled = True
    cmdListen.Enabled = True
    cmdDisconnect.Enabled = False
    cmdSendData.Enabled = False
    cmdCloseListen.Enabled = False
End Sub
```

```
Private Sub WinSock1_ConnectionRequest()
' We got a client connection; accept it on the listening
' socket
'
    WinSock1.Accept
End Sub

Private Sub WinSock1_DataArrival(ByVal bytesTotal)
' This is the event for data arrival.  Get the data, and
' add it to the list box.
'
    Dim rdata

    WinSock1.GetData rdata
    List1.AddItem rdata
End Sub

Private Sub WinSock1_Error(ByVal number, ByVal description)
' An error occurred; display the message, and close the socket
'
    MsgBox description
    Call WinSock1_Close
End Sub
```

在此，我们不打算就程序清单 15-3这个示范程序的规格进行详细说明，因为它其实和程序清单15-2展示的那个SockTCP示例是类似的。唯一的区别在于前一节提到的那些限制。在此要注意的是，Windows CE Winsock控件只是一种比较“初级”的控件。换句话说，它的设计并不像桌面版本那么完善。类型库并没有完全实现：必须通过一个整数，对协议类型加以区分；而不能对类型进行枚举。除此以外，就像下一节“已知的问题”要讲述的那样，套接字状态枚举类型也存在着问题。

红外线连接的控制其实并不难，和 TCP连接差不多。但在通过红外线端口建立一个监听套接字时，却会出现一个例外。对一个红外线服务器来说，我们用“服务名”对其进行引用，后者已在第6章的IrDA定址小节进行了详细解释。Windows CE Winsock控件新增了一个属性，名为ServiceName。我们可将该属性设为客户机试图连接的下一个服务名文本字串。举个例子来说，下述代码片断可将名为 CeWinsock的Windows CE Winsock控件置入监听模式，并赋予“MyServer”的名字：

```
CeWinsock.Protocol = 2              ' Protocol 2 is IrSock
CeWinsock.ServiceName = "MyServer"
CeWinsock.Listen
```

要想通过红外线套接字发布一项服务，除此以外便无其他特殊要求，只需指定服务名即可。

15.8.2 已知的问题

使用VBCE Winsock控件时，我们在为Winsock状态使用列举的值时，发现了一个奇怪的问题。不知由于什么原因，这些值是在开发环境中定义的，但每次在代码中引用一个 sck枚举值时，在远程设备上，都会弹出下述错误提示消息：“运行这个程序时遇到错误”（An error was encountered while running this program）。若将枚举换成各自对应的常数值，这种错误便会消失。我们认为这是VBCE的一个错误，并希望在未来版本的工具包中，能得以纠正。

15.9 小结

Visual Basic Winsock控件非常适用于简单的、不重要的网络应用程序。 Winsock控件的Visual Basic 5.0版本中存在着几个问题，造成难以成功地进行控件编程，但主要问题都在最新发布的Visual Basic版本中得到了解决。有了这个控件，便可在自己的 Visual Basic应用程序中自由增添网络通信能力。当然，它并不是万能的，应用程序如果需要与 Winsock进行大量沟通，便应考虑从Winsock DLL中手工导入必要的函数和常数。就像我们在前面提到的那样，在本书第二部分提供的各个Winsock Visual Basic示例中，实际都从Ws2_32.dll中导入了Winsock函数。例如，大家可参考本书配套光盘 Chapter07\VB目录下的SimpleTCP与SimpleUDP程序，以及它们对应的Winsock.bas文件。

第三部分　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网络 API函数。利用这些函数，我们的应用程序可通过网络，建立与其他程序的通信联系。在那些讨论中，我们在很大程度上将重点放在七层 OSI模型的应用层和表示层上面。在这个过程中，我们并没有就具体的某种网络协议进行剖析，大多数讨论都站在一个“与协议无关”的角度，解释如何来使用函数。

本书的最后这一部分讨论了一种重要的服务，名为“远程访问服务”（Remote Access Service, RAS），它允许用户从远程地点，将自己的计算机连接到一个本地计算机网络。一旦建立了连接，便可使用本书讲述的所有网络函数，即便你手上的计算机实际连接的是一个远程网络！

第16章　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（其特色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一个调制解调器，通过拨号的方式呼叫远程计算机。因此，有时， RAS客户机也称作“拨号联网（DUN）客户机”。

服务器这方面，必须有一项等候 DUN连接的服务。RAS客户机能够和多种类型的远程访问服务器建立通信。RAS通过使用工业标准分帧协议建立通信。比如这些协议：

- 点到点协议（PPP）可传输IP IPX NetBEUI通信协议。
- 串行线路网际协议（SLIP）只能传输IP通信协议。
- 异步NetBEUI（微软的Windows NT 3.1、微软的Windows for Workgroups3.11）只能传输NetBEUI通信协议。

这些分帧协议描述了数据是怎样通过 RAS链接进行传输的，指令 RAS链接上采用哪些网络通信协议（比如 TCP/IP或IPX）通信。如果RAS服务器支持前面定义的任何一种分帧协议，RAS客户机便可以与之建立一个链接。 Windows 95、Windows 98、Windows 2000以及Windows NT的特色便是：RAS服务器组件能支持前面列出的任何一种分帧协议。

RAS客户机和服务器之间的连接建立之后，网络协议堆栈（与所用的分帧协议有关）就通过这个RAS连接，与远程计算机通信，就象通过 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。

RAS服务器收到一次拨号连接时，处理前面列出的其中一种分帧协议之后，便与客户机开始通信。分帧协议一旦确立， 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、口令和域登录凭据。 Windows 2000或Windows NT RAS服务器收到这条信息时，利用 Windows NT域安全访问控制验证登录

凭据。注意，RAS服务器没有进入客户机登录的Windows NT域；相反地，它采用客户机凭据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。Windows 95和Windows 98平台上，RAS连接通过电话簿条目中可用的选项，经过验证后，RAS便可自动进入一台机器，登录到一个域，和我们将在本章稍后讲的一样。

RAS依赖“电话应用程序接口”（Telephony Application Programming Interface，TAPI）设置和控制调制解调器之类的电话通信设备，TAPI控制这些拨号设备的硬件设置。在利用调制解调器设置一个RAS连接时，TAPI会转向调制解调器，并自RAS向调制解调器发出拨号信息。这样一来，RAS就会把调制解调器看成简单的TAPI调制解调器端口，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。

本章还将讨论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自己的应用程序时，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——如何真正建立一个远程连接。然后介绍如何设一个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，我们再为大家展示如何管理已建立的连接。

16.1 编译和链接

在开发一个RAS应用程序时，需要用这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串，这个字串和RAS中所用的各错误代码关联在一起。RAS特别突出的是一个名为RasGetErrorString的函数，它非常有用，允许通过编程来获得与特定RAS错误代码关联的错误字串。RasGetErrorString的定义如下：

```
DWORD RasGetErrorString(
    UINT uErrorValue,
    LPTSTR lpszErrorString,
    DWORD cBufSize
);
```

uErrorValue参数取得一个特定的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，将接收一个错误字串，该字串与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错误字串来说，这个长度都比较恰当的。最后一个参数cBufSize是你为lpszErrorString提供的缓冲区的长度。

16.2 数据结构和平台兼容问题

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，分别使用或不用额外的数据字段。但是，Windows CESDK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，必须把它设为你使用的RAS结构的字节长度。这样做会对使用这些结构的RAS函数产生影响，因为对这些RAS函数而言，目标是针对某具体平台的。WINVER标准适用于Windows 95、Windows 98、Windows 2000和Windows NT平台。

■ WINVER = 0X400：指明你的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。

■ WINVER = 0X401：指明你的RAS应用程序的目标平台是有服务包的Windows NT。

■ WINVER = 0X500：指明你的RAS应用程序的目标平台是Windows 2000。

RAS本身没有可在所有平台上运行的一个可执行字段。但通过细致的编程，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应用程序时，应该有一个具体的目标平台。

16.3 DUN 1.3升级和Windows 95

在OSR的第一个版本到OSR 2期间，Windows 95发布了大量的版本。OSR 2是非卖品；只供OEM厂商安装在它们的产品上。每次发布都对RAS API的功能进行了沿伸。因此，我们建议大家最好安装最新的RAS升级包（DUN 1.3）以进一步发挥RAS的潜力。要获得Windows 95的DUN 1.3升级版，访问http://www.microsoft.com/support。本章的前提是你的机器上至少已经安装了DUN 1.3升级版；至于以前的DUN版本中的RAS，我们将不讨论。

16.4 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
);
```

lpRasDialExtensions参数是一个可选指针，指向一个RASDIALEXTENSIONS结构，有了这个结构，你的应用程序便可使用RasDial函数的扩展特性了。在Windows 95、Windows 98和Windows CE上，这个参数是被忽略的，必须把它设为NULL。RASDIALEXTENSIONS结构的格式如下：

```
typedef struct tagRASDIALEXTENSIONS {
   DWORD       dwSize;
   DWORD       dwfOptions;
   HWND        hwndParent;
   ULONG_PTR   reserved;
#if (WINVER >= 0x500)
   ULONG_PTR   reserved1;
   RASEAPINFO  RasEapInfo;
```

```
#endif
} RASDIALEXTENSIONS;
```

注意，前面已经讲过，编译期间，由于 WINVER值不相同，这个结构的长度也会不同。它的各个字段是这样定义的：

- dwSize：必须设为RASDIALEXTENSIONS结构的长度（按字节算）。
- dwfOptions：允许你为使用RasDial扩展特性而设置位标志。表 16-1对这些标志进行了说明。
- hwndParent：不用，应该设为NULL。
- reserved：不用，应该设为0。
- reserved1：保留，供以后的Windows 2000 RAS扩展特性使用。应该设为0。
- RasEapInfo：在Windows 2000上，允许你指定“扩展性身份验证协议”（EAP）信息。EAP不在本书讨论范围内。

表16-1 RasDial位标志

标 志	说 明
RDEOPT_UsePrefixSuffix	令RasDial使用与指定拨号设备关联的前缀和后缀
RDEOPT_PausedStates	允许RasDial进入暂停操作状态，这样一来，用户就可以重试登录、改变密码和设置回拨号码
RDEOPT_IgnoreModemSpeaker	令RasDial忽略RAS电话簿中的Modem喇叭设置
RDEOPT_SetModemSpeaker	如果设置了RDEOPT_IgnoreModemSpeaker标志，那么利用这个标志，就可以打开Modem喇叭
RDEOPT_IgnoreSoftwareCompression	令RasDial忽略软件压缩设置
RDEOPT_SetSoftwareCompression	在设置了RDEOPT_IgnoreSoftwareCompression标志的前提下，再用这个标志，便会打开软件压缩
RDEOPT_PauseOnScript	供RasDialDlg内部使用。不能设置这个标志

在Windows 2000和Windows NT中，RasDial函数的lpszPhonebook参数用于识别到一个电话簿文件的路径。如在Windows 95、Windows 98和Windows CE上，这个参数必须设为NULL，因为电话簿是保存在系统注册表内的。所谓电话簿，就是一个 RAS拨号属性的集合，这些属性对如何设置一个 RAS连接进行了定义。但并不要求你利用这个电话簿建立一个 RAS连接。RasDial的特色在于其丰富的拨号参数，允许你设置一个基本连接。我们稍后将详细讨论 RAS电话簿。

RASDIALPARAMS结构指针lpRasDialParams定义了拨号和用户身份验证参数， RasDial函数便用这些参数建立一个远程连接。该结构的格式如下：

```
typedef struct _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;
#endif
```

```
} RASDIALPARAMS;
```

RASDIALPARAMS结构的各个字段是这样定义的：

- dwSize：应该设为一个RASDIALPARAMS结构的长度（按字节算）。有了这个标志，RAS便可对你编辑程序时所用的WINVER版本进行内部判断。
- szEntryName：一个字串，允许你标识一个电话簿条目，该条目中包含在 RasDial函数的lpszPhonebook参数中列出的电话簿文件内。这个参数非常重要，因为电话簿条目能够使你更好地指定RAS连接属性，比如选定一个Modem或选定一个分帧协议等等。但是，为使用RasDial而指定一个电话簿条目并不是非用不可的，可选择使用。如果这个字段是空字串（“”），RasDial就会选择系统上已安装的第一个可用的 Modem，并依据下一个参数szPhoneNumber来拨叫连接。
- szPhonebookNumber：一个字串，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：允许你指定一个电话号码， RAS服务器可以根据这个号码回拨。如果RAS服务器允许有一个回拨号码，它就会中断原来的连接，利用你指定的回拨号码回拨。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 2000或Windows NT区域。
- szDomain：可选。允许你指定最初的电话簿子条目拨叫一个 RAS多链路连接（我们稍后将讨论RAS电话簿子条目和多链路连接）。
- dwCallbackId：允许你把一个应用程序定义的值投递到一个 RasDialFunc2回拨函数中，这个字段没有使用。

下面这两个RasDial参数——dwNotifier和lpvNotifier，用于对RasDial操作模式进行判断（可同步调用还是异步调用）。最后一个RasDial参数是lphRasConn，它是一个指针，指向HRASCONN类型的RAS连接句柄。在调用 RasDial之前，必须把这个参数设为 NULL。如果RasDial调用成功，就会返回指向RAS连接的引用句柄。

下面，我们将为大家演示如何调用 RasDial。正如我们前面提到的那样， 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会处于锁定状态，而异步模式中， RasDial则立即执行连接，同时允许你的应用程序在连接期间执行别的操作。

16.4.1 同步模式

如果把RasDial的lpvNotifier参数设为NULL，RasDial就会置入同步模式。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作用；美中不足的是，同步模式下不能对连接进行监视，稍后，我们将就此进行详述。程序清单16-1演示了如何同步调用 RasDial。注意，这段代码中没有指定电话簿或电话簿条目，而是演示如何简单地建立RAS连接。

程序清单16-1 同步调用RasDial

```
// Always set the size of the RASDIALPARAMS structure
```

```
RasDialParams.dwSize = sizeof(RASDIALPARAMS);
hRasConn = NULL;

// Setting this field to an empty string will allow
// RasDial to use default dialing properties

lstrcpy(RasDialParams.szEntryName, "");

lstrcpy(RasDialParams.szPhoneNumber, "867-5309");
lstrcpy(RasDialParams.szUserName, "jenny");
lstrcpy(RasDialParams.szPassword, "mypassword");
lstrcpy(RasDialParams.szDomain, "mydomain");

// Call RasDial synchronously (the fifth parameter
// is set to NULL)
Ret = RasDial(NULL, NULL, &RasDialParams, 0, NULL, &hRasConn);

if (Ret != 0)
{
    printf("RasDial failed: Error = %d\n", Ret);
}
```

16.4.2 异步模式

比起同步调用RasDial函数来，异步调用要复杂得多。如果 RasDial的lpvNotifier参数没有设为NULL，RasDial就会进入异步模式，意味着进行连接的同时，函数调用会立即返回。在进行RAS连接时，异步调用RasDial是优选方法，因为你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，又可以是一个窗口句柄（通过Windows消息接收进程通知）。RasDial函数的dwNotifierType参数用于判断函数类型或被投入lpvNotifier的窗口句柄。至于 dwNotifierType中可以指定哪些值呢？表 16-2对此进行了说明。

表16-2 RasDial异步通知方法

通知类型	含 义
0	lpvNotifier参数令RasDial使用RasDialFunc函数指针管理连接事件
1	lpvNotifier参数令RasDial利用RasDialFunc1函数指针管理连接事件
2	lpvNotifier参数令RasDial利用RasDialFunc2函数指针管理连接事件
0xFFFFFFF	lpvNotifier参数令RasDial在连接事件期间发送一个窗口消息

表16-2展示了lpvNotifier参数中的三个函数原型，你可以把它们提供给 RasDial函数，用来接收连接事件的回拨通知。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，它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 16-3

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16-3展示了异步RasDial调用中，连接活动有三种状态：运行、暂停和中断。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。默认设置是取消暂停状态。在RASDIALEXTENSIONS结构中设置RDEOPT_PausedStates标志，启用通知进程。关于这一点，我们在前面曾经讲过。连接活动处于暂停状态时，就会指明相应的情况。

- 用户需要提供新的登录凭据，因为身份验证失败。
- 用户需要提供一个新的密码，因为密码已到期。
- 用户需要提供一个回拨号码。

表16-3 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	已成功于设备连接
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拨号码
RASCS_AuthChangePassword	运行	客户机已请求改变RAS账号上的密码
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	回拨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	线路即将取消连接，准备回拨
RASCS_WaitForModemReset	运行	准备回拨之前，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接入拨号
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和Windows 98）
RASCS_CallbackComplete	运行	已经回拨客户机（只适用于Windows 95和Windows 98）
RASCS_LogonNetwork	运行	客户机正在登录远程网络（只适用于 Windows 95和Windows 98）
RASCS_SubEntryConnected	运行	多链路电话簿条目的子条目已接入。 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取消连接。 RasDialFunc2的dwSubEntry参数中将包含这个已取消连接的子条目的索引
RASCS_RetryAuthentication	暂停	RasDial正在等待新的用户凭据
RASCS_CallbackSetByCaller	暂停	RasDial正在等待客户机的回拨号码
RASCS_PasswordExpired	暂停	RasDial希望用户提供一个新密码
RASCS_InvokeEapUI	暂停	Windows 2000平台上，RasDial等待自定义用户接口，以便获得EAP信息
RASCS_Connected	中止	RAS连接成功，处于活动状态
RASCS_Disconnected	中止	RAS连接失败或处于不活动状态

这些活动与本章前面提到的RASDIALPARAMS结构中的信息息息相关。发生处于暂停状态的连接活动时，RasDial就会向你的回调函数（或窗口进程）发出通知。如果把暂停状态取消，RAS就会向你的通知函数发送一个错误，RasDial就会失败。如果启用暂停，RasDial函数就会进入暂停状态，这样，你的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），简单地中止暂停操作，恢复运行。恢复暂停连接时，需要通过投递给已恢复的RasDial调用的RASDIALPARAMS结构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句柄函数（比如RasDialFunc）调用RasDial。RasDial尚不能处理这种情况，所以应该直接从你的应用程序线程中恢复暂停状态。

最后一个状态(中止)表明RasDial连接不是已成功，就是失败。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建立的程序，使其能够异步调用RasDial。程序清单16-2展示了这一过程。大家可在本书配套光盘上找到一个异步调用RasDial示例。

程序清单16-2 异步调用RasDial

```
void main(void)
{
    DWORD Ret;
    RASDIALPARAMS RasDialParams;
    HRASCONN hRasConn;

    // Fill in the RASDIALPARAMS structure with call parameters
    // as was done in the synchronous example
    ...

    if ((Ret = RasDial(NULL, NULL, &RasDialParams, 0,
        &RasDialFunc, &hRasConn)) != 0)
    {
        printf("RasDial failed with error %d\n", Ret);
        return;
    }

    // Perform other tasks while RasDial is processing
    ...
}

// Callback function RasDialFunc()
void WINAPI RasDialFunc(UINT unMsg, RASCONNSTATE rasconnstate,
    DWORD dwError)
{
    char szRasString[256]; // Buffer for error string

    if (dwError)
    {
        RasGetErrorString((UINT)dwError, szRasString, 256);
```

```
            printf("Error: %d - %s\n",dwError, szRasString);
            return;
        }
        // Map each of the states of RasDial, and display on the
        // screen the next state that RasDial is entering

        switch (rasconnstate)
        {
            case RASCS_ConnectDevice:
                printf ("Connecting device...\n");
                break;
            case RASCS_DeviceConnected:
                printf ("Device connected.\n");
                break;

            // Add other connection activities here
            ...

            default:
                printf ("Unmonitored RAS activity.\n");
                break;
        }
    }
```

表16-2中，还提到了另外两个回调通知函数，它们是： RasDialFunc1和RasDialFunc2。它们的原型如下：

```
VOID WINAPI RasDialFunc1(
    HRASCONN hrasconn,
    UINT unMsg,
    RASCONNSTATE rascs,
    DWORD dwError,
    DWORD dwExtendedError
);

DWORD WINAPI RasDialFunc2(
    DWORD dwCallbackId,
    DWORD dwSubEntry,
    HRASCONN hrasconn,
    UINT unMsg,
    RASCONNSTATE rascs,
    DWORD dwError,
    DWORD dwExtendedError
);
```

RasDialFunc1函数和我们前面讨论的RasDialFunc函数极其相似，只不过增加了两个参数：hbrasconn和dwExtendedError。hbrasconn参数是一个指向连接的句柄。这个连接是 RasDial函数返回的。dwExtendedError参数，允许你在连接期间获得 dwError参数所发生的这些错误的扩展错误信息。

- ERROR_SERVER_NOT_RESPONDING：dwExtendedError收到一个NetBIOS特有的错误代码。
- ERROR_NETBIOS_ERROR：dwExtendedError收到一个NetBIOS特有的错误代码。
- ERROR_AUTH_INTERNAL：dwExtendedError收到一个内部诊断性错误代码。这些错误代码尚未归入文档。

■ ERROR_CANNOT_GET_LANA：dwExtendedError收到一个路由RAS特有的错误代码。

RasDialFunc2函数和RasDialFunc1函数极其相似，但它比后者多了两个参数：dwCallbackId和dwSubEntry。dwCallbackId参数中包含了一个应用程序定义的值，这个值最初是设在RASDIALPARAMS结构的dwCallbackId字段中的，我们前面曾讲过这个结构被投给了 RasDial调用。dwSubEntry参数接收子条目电话簿索引，这个索引会再次回调 RasDialFunc2函数。

16.4.3 状态通知

RAS的特征之一是单立函数 RasConnectionNotification。它允许你的应用程序决定何时建立或中断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，该句柄是你的应用程序利用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：

■ RASCN_Connection：通知你RAS连接已经建立。如果 hrasconn参数设为INVALID_HANDLE_VALUE，任何一次RAS连接都会触发这一事件。

■ RASCN_Disconnection：通知你RAS连接已经中止。如果hrasconn参数设为INVALID_HANDLE_VALUE，任何连接中止都会触发这一事件。

■ RASCN_BandwidthAdded：在一个多链路连接上，子条目连接会触发这一事件。

■ RASCN_BandwidthRemoved：在多链路连接上，子条目取消连接时，会触发这一事件。

注意，这些标志的活动方式和表 16-3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，就会触发你的事件。因此，你的应用程序应该使用 Win32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。

16.4.4 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。尽管这个函数非常易于使用，但仍然需要了解RAS中，连接的内部管理是怎么回事。连接在利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，你应该一直等下去，直到端口连接完全关闭为止。要做到这一点，在重新设置自己的连接时，可调用 RasGetConnectionStatus来判断连接是否完全关闭。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，取得当前的连接状态。RASCONNSTATUS结构的格式如下：

```
typedef struct _RASCONNSTATUS
{
    DWORD dwSize;
    RASCONNSTATE rasconnstate;
    DWORD dwError;
    TCHAR szDeviceType[RAS_MaxDeviceType + 1];
    TCHAR szDeviceName[RAS_MaxDeviceName + 1];
} RASCONNSTATUS;
```

上面这个结构的各个字段是这样定义的：

- dwSize：应该设置为RASCONNSTATUS结构的长度（按字节算）。
- rasconnstate：取得表16-3中定义的一种连接活动。
- dwError：若RasGetConnectStatus没有返回0，就取得一个具体的RAS错误代码。
- szDeviceType：取得一个字串，该字串代表连接所用的设备类型。
- szDeviceName：取得当前的设备名。

我们建议在收取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。

16.5 电话簿

利用电话簿条目对建立连接所需的属性进行保存和管理，通过这一方式， RAS便可以设置与远程服务器的通信。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、用户身份验证信息和其他连接信息。在 Windows 95、Windows 98和Windows CE平台上，电话簿保存在系统的“注册表”内。在 Windows NT和Windows 2000平台上，电话簿的扩展名一般是.PBK的文件中。RASENTRY结构的格式如下：

```
typedef struct 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
} RASENTRY;
```

正如大家看到的那样，上面这个结构由许多字段组成。这些字段的定义如下：

■ dwSize：指定一个RASENTRY结构的长度（按字节算）。

■ dwfOptions：可以设为一个或多个的标志，这些标志的描述可参见表 16-4。

表16-4 RASENTRY结构选项的标志

选项标志	描 述
RASEO_Custom	在Windows 2000平台上，使用自定义加密
RASEO_DisableLcpExtensions	RAS取消定义在RFC 1570中的PPP LCP扩展
RASEO_IpHeaderCompression	RAS将对PPP连接上的IP头压缩进行处理
RASEO_ModemLights	在Windows 2000上，这个任务栏将显示状态监视器
RASEO_NetworkLogon	在Windows 95和Windows 98上，只有在RAS连接通过身份验证之后，RAS才会将让用户登录到一个域
RASEO_PreviewDomain	在Windows 2000上，RAS将在拨号之前，显示用户域
RASEO_PreviewPhoneNumber	Windows 2000上，RAS将显示准备拨打的那个电话号码
RASEO_PromoteAlternates	如果利用备用号码进行的连接成功的话，RAS可以把备用号码转为首要号码
RASEO_RemoteDefaultGateway	RAS连接处于活动状态时，IP包的默认路由通过的是拨号适配器，而不是其他的网络适配器
RASEO_RequireCHAP	Windows 2000上，将采用“盘问联络身份验证协议”（CHAP）进行身份验证
RASEO_RequireDataEncryption	必须成功地协商数据加密；不然就应该丢弃这个连接。此外，还必须设置RASEO_RequireEncrytedPw标志
RASEO_RequireEAP	Windows 2000平台上，将采用EAP进行身份验证
RASEO_RequireEncryptedPw	对客户机进行身份验证时，设置这个标志可避免PPP使用“密码身份验证协议”（PAP），而是用CHAP和Shiva的密码身份验证协议（SPAP）

(续)

选项标志	描 述
RASEO_RequireMsCHAP	Windows 2000平台上，将采用微软的CHAP进行身份验证
RASEO_RequireMsCHAP2	Windows 2000平台上，将采用微软的CHAP第2版进行身份验证
RASEO_RequireMsEncryptedPw	这个标志优先于RASEO_RequireEncryptedPw，允许RAS使用微软的安全密码方案，比如说微软的CHAP
RASEO_RequirePAP	Windows 2000平台上，将采用PAP进行身份验证
RASEO_RequireSPAP	Windows 2000平台上，将采用SPAP进行身份验证
RASEO_RequireW95MSCHAP	Windows 2000平台上，采用老版本的微软CHAP（这是为Windows 95 RAS服务器设计的）进行身份验证
RASEO_ReviewUserPW	Windows 2000平台上，RAS将在拨号之前，显示用户名和密码
RASEO_SecureLocalFiles	Windows 2000和Windows NT平台上，RAS在与一个条目建立连接之前，对现成的远程文件系统和远程打印机的绑定进行检查
RASEO_SharedPboneNumbers	Windows 2000上，电话号码是共享的
RASEO_ShowDialingProgress	Windows 2000上，RAS将显示拨号过程
RASEO_SpecificIpAddr	这个标志要求RAS使用ipaddr字段中指定的IP地址
RASEO_SpecificNameServers	这个标志要求RAS使用ipaddrDns、ipaddrDnsAlt、ipaddrWins和ipaddrWinsAlt这几个字段中指定的IP信息
RASEO_SwCompression	这个标志让RAS对连接上发送的数据的软件压缩进行协商
RASEO_TerminalAfterDial	拨号连接后，RAS显示用户输入的终端窗口
RASEO_TerminalBeforeDial	拨号连接之前，RAS显示用户输入的终端窗口
RASEO_UseCountryAndAreaCodes	这个标志要求RAS利用dwCountryID、dwCountryCode和szAreaCode字段以及szLocalPhoneNumber字段，构建一个电话号码
RASEO_UseLogonCredentials	这个标志要求RAS在拨号时，采用用户名、密码和当前正在登录的用户所在区域。只有设置RASEO_RequiredMsEncryptedPw标志时，才会发生这种情况

- dwCountryID：如果设置了RASEO_UseCountryAndAreaCodes选项标志，就会指定一个TAPI国家标识符。调用RasGetCountryInfo函数之后，便可获取国家标识码信息（我们稍后将对此进行讨论）。
- CountryCode：如果设置了RASEO_UseCountryAndAreaCodes选项标志，就会指定一个与dwCountryID字段关联的国家代码。如果该字段为 0，就采用Windows中与dwCountryID关联的国家代码。
- szAreaCode：如果设置了RASEO_UseCountryAndAreaCodes标志，就会指定一个地区代码。
- szLocalPhoneNumber：指定准备拨打的电话号码。如果设置了RASEO_UseCountry-AndAreaCodes标志，RAS就会把dwCountryID、dwCountryCode和szAreaCode这三个字段的值和你的电话号码合并在一起。
- dwAlternateOffset：指定按字节算的偏移量，从这个结构的开始处算起，为RAS电话簿条目准备的备用电话号码就保存在这里。备用电话号码被保存为一个连续的空中止字串集。这个集合中的最后一个字串是以两个连续的空字符结尾的。
- ipaddr：如果设置了RASEO_SpecificIpAddr标志，就为这个连接指定一个IP地址，供它使用。
- ipaddrdns：如果设置了RASEO_SpecificNameServers标志，就为这个连接指定一个DNS服务器IP地址。

- ipaddrDnsAlt：如果设置了RASEO_SpecificNameServers标志，就为这个连接指定一个备用的DNS服务器IP地址。
- ipaddrDWins：如果设置了RASEO_SpecificNameServers标志，就为这个连接指定一个WINS服务器IP地址。
- ipaddrDWinsAlt：如果设置了RASEO_SpecificNameServers标志，就为这个连接指定一个备用的WINS服务器IP地址。
- dwFrameSize：若在dwFramingProtocol字段中设置RASFP_Slip标志，分帧协议的长度就会变为1006或1500个字节。
- dwfNetProtocols：指定标志，这些标志表示分帧协议上将使用哪些网络协议。可能有这几个标志；针对 NetBEUI的RASNP_NetBEUI、针对IPX的RASNP_Ipx、针对IP的RASNP_Ip。
- dwFramingProtocol：指定标志，这些标志表示RAS连接上将使用哪些分帧协议。可能有这几个标志；针对PPP的RASFP_Ppp、针对SLIP的RASFP_Slip、针对异步。
- szScript：指定到一个拨号Script的完整路径，一开始连接，就会执行这个拨号。
- szAutodialDll：指定一个定制的 DLL，该DLL可用于设置RAS的自动拨号（autodial）特性。本书没有讨论如何管理RAS自动拨号特性。
- szAutodialFunc：对szAutodialDll字段中定制DLL输出的函数名进行指定。
- szDeviceType：指定用于连接的设备类型。其值应该标识成字串。表 16-5列出了它可能的值。

表16-5 RAS设备类型

设备名字串	说 明
“RASDT_Modem”	COM端口上的调制解调器
“RASDT_Isdn”	ISDN适配器
“RASDT_X25”	X.25网卡
“RASDT_Vpn”	虚拟专用网（VPN）连接
“RASDT_Pad”	包汇编器/取消汇编器
“RASDT_Generic”	一般类型
“RASDT_Serial”	串口
“RASDT_FrameRelay”	帧中继设备
“RASDT_Atm”	ATM设备
“RASDT_Sonet”	同步光纤网络（SONET）设备
“RASDT_SW56”	交换56-Kbps访问
“RASDT_Irda”	红外线设备
“RASDT_Parallel”	并口

- szDeviceName：对连接所用的 TAPI设备进行识别。可利用 RasEnumDevices函数获取TAPI设备，稍后将对此进行讨论。
- szX25PadType：指定一个X.25 PAD类型。
- szX25Address：指定一个X.25地址。
- szX25Facilities：指定设备应答X.25主机发出的请求。
- szX25UserData：为X.25连接指定额外的信息。
- dwChannels：未使用。

- dwReserved1：未使用，但必须设为0。
- dwReserved2：未使用，但必须设为0。
- dwSubEntries：识别与该电话簿条目关联的多链路子条目有多少。应该把这个字段设为0，并令RasSetSubEntryProperties函数管理该字段的多链路子条目（关于这个函数，我们稍后将详述）。
- dwDialMode：Windows 2000中，利用RASEDM_DialAll和RASEDM_DialAsNeeded的定义值进行连接时，这个字段指定 RAS应该怎样拨多链路子条目。如果这个字段设为RASEDM_DialAll，就会拨叫所有的多链路子条目。如果设为RASEDM_DialAsNeeded，RAS就会利用dwDialExtraPercent、dwDialExtraSampleSeconds、dwHangUpExtraPercent和dwHangUpExtraSampleSeconds这四个字段来决定何时拨叫和取消新增的多链路子条目。
- dwDialExtraPercent：Windows 2000中，该字段指定一个连接当前总带宽的百分比。所用的总带宽超过这个百分比超过了 dwDialExtraSampleSeconds指定的时间时，RAS就会拨叫另一个子条目。
- dwDialExtraSampleSeconds：Windows 2000中，利用该字段，表明RAS拨叫另一个子条目之前，当前带宽的利用必须超过 dwDialExtraPercent中指定的带宽百分比多少秒。
- dwHangUpExtraPercent：Windows 2000中，从连接的子条目可用的总带宽的百分比。总带宽的使用率低于dwDialExtraSampleSeconds中指定的百分比时，RAS将中断子条目连接。
- dwHangUpExtraSampleSeconds：Windows 2000中，利用该字段，表明RAS在取消一个子条目之前，当前带宽的使用必须降到 dwHangUpExtraPercent中定义的带宽百分比以下的限定时间。
- dwIdleDisconnectSeconds：利用这个字段，表明在RAS中断连接之前，允许连接闲置多长时间。还可把这个字段设为RASIDS_Disabled（防止连接中断），或RASIDS_UseGlobalValue（利用系统的默认值）。
- dwType：Windows 2000中，指定电话簿条目的类型。表16-6列出了这些类型可能的值

表16-6 RASENTRY电话簿类型

类 型	说 明
RASET_Direct	直接的串行或并行连接
RASET_Internet	互联网连接服务（ICS）
RASET_Phone	电话线
RASET_Vpn	虚拟专用网

- dwEncryptionType：Windows 2000中，指定连接上投递的数据所使用的加密类型。表16-7列出了可能的值。

表16-7 RAS连接上所使用的数据加密值

值	说 明
ET_40Bit	40位数据加密
ET_128Bit	128位数据加密

- dwCustomAuthkey：Windows 2000平台上，指定一个身份验证密钥，这是为 EAP厂商

提供。

- guidId：Windows 2000平台上，标识与一个电话簿条目关联在一起的GUID。
- szCustomDialDll：Windows 2000平台上，指定到一个DLL的路径，这个DLL中包含了自定义RAS拨叫函数。如果这个字段是NULL，RAS将采用默认的系统拨叫者。至于如何为RAS开发自定义拨叫者的种种细节，本书没有涉及。
- dwVpnStrategy：Windows 2000平台上，指定VPN连接将采用的VPN拨叫方案表16-8列出了可能的值。

表16-8 RAS VPN拨号方案

值	说 明
VS_Default	RAS先拨叫点到点通道传输协议(PPTP)。如果失败，就尝试第2层通道传输协议(L2TP)。
VS_L2tpFirst	先拨叫L2TP
VS_L2tpOnly	只拨叫L2TP
VS_PptpFirst	先拨叫PPTP
VS_PptpOnly	只拨叫PPT

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和Windows CE平台上，这个参数必须是 NULL，因为电话簿条目是保存在系统注册表内的。而在 Windows 2000和Windows NT平台上，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， Windows 2000和Windows NT上的系统默认电话簿位于% SystemRoot% \System32\Ras\RasPhone.pbk。如果你将电话簿指定为NULL，便使用系统默认电话簿文件。

有三个支持函数可帮助你建立和管理电话簿条目。它们是： RasValidateEntry、RasEnumDevices和RasGetCountryInfo。下面展示的RaValidateEntryName函数，用来判断名字的格式是否正确，是否已包含在电话簿中。

```
DWORD RasValidateEntryName(
    LPCTSTR lpszPhonebook,
    LPCTSTR lpszEntry
);
```

lpszPhonebook参数是一个指针，指向电话簿文件名。 lpszEntry参数是一个字串，表示正在核对的电话簿条目名。电话簿中没有这个名字，但该名字格式无误时，便返回ERROR_SUCESS。若名字格式有错，这个函数就会失败，返回 ERROR_INVALID_NAME；如果电话簿中有这个名字，就会返回 ERROR_ALREADY_EXISTS。

对具体的计算机而言，可利用 RasEnumDevice函数获得所有具有RAS能力的设备名及类型。

```
DWORD RasEnumDevices(
    LPRASDEVINFO lpRasDevInfo,
    LPDWORD lpcb,
    LPDWORD lpcDevices
);
```

lpRasDevInfo参数是一个指针，指向一个你必须提供的应用程序缓冲区，这个缓冲区用来接收RASDEVINFO结构数组。RASDEVINFO结构的格式如下：

```
typedef struct tagRASDEVINFO {
    DWORD dwSize;
```

```
    TCHAR szDeviceType[RAS_MaxDeviceType + 1];
    TCHAR szDeviceName[RAS_MaxDeviceName + 1];
} RASDEVINFOW;
```

该结构的字段是这样定义的：

- dwSize：在调用RasEnumDevices之前，必须把它设为RASDEVINFO结构的长度（按字节算）。
- szDeviceType：接收对设备类型进行描述的字串，比如 RASDT_Modem。
- szDeviceName：接收TAPI设备的正式名称。

这时，必须确定提供的缓冲区足以容纳若干个结构；否则， RasEnumDevices就会失败，返回ERROR_BUFFER_TOO_SMALL。下一个参数lpcb，是一个变量指针，用于取得列举设备时所需的字节数。这个参数必须设为你自己的 lpRasDevInfo缓冲区的长度（按字节算）。最后一个参数lpcDevices，是一个变量指针，用于取得写入 lpRasDevInfo中的lpcDRASADEVINFO结构的个数。

RasGetCountryInfo函数允许从Windows中取得各个国家特有的TAPI拨叫信息。

```
DWORD RasGetCountryInfo(
    LPRASCTRYINFO lpRasCtryInfo,
    LPDWORD lpdwSize
);
```

lpRasCtryInfo参数是一个缓冲区，用于接收拨号的前缀和与指定国家相关的其他信息。这个缓冲区必须是一个 RASCTRYINFO结构，后面再跟上额外的字节（接收一个国家描述字串）。为了能够容纳RASCTRYINFO结构和描述性字串，我们建议至少分配 256个字节的缓冲区。RASCTRYINFO结构的格式如下：

```
typedef struct RASCTRYINFO
{
    DWORD dwSize;
    DWORD dwCountryID;
    DWORD dwNextCountryID;
    DWORD dwCountryCode;
    DWORD dwCountryNameOffset;
} RASCTRYINFO;
```

它的字段是这样定义的：

- dwSize：必须设为RASCTRYINFO结构的长度（按字节算）。
- dwCountryID：允许你在 Windows国家列表中指定一个 TAPI国家识别符（用于RASENTRY结构的dwCountryID字段）。如果把这个字段设为 1，就会收到列表中的第一个条目。
- dwNextCountryID：收到列表中的下一个 TAPI国家识别符。如果把这个字段设为 0，就会收到列表中的最后一个识别符。
- dwCountryCode：收到指定在dwCountryID参数中的国家相关的拨号前缀代码。
- dwCountryNameOffset：指定字节数，表示这个结构的起始处到空中止字串的起始处之间的字节是多少，这个空中止字串对 RASCTRYINFO结构后面的国家进行了描述。

RasGetCountryInfo的另一个参数是lpdwsize，它是一个变量指针，接收 lpRasCtryInfo放在lpRasCtryInfo缓冲区中的字节数。在调用这个函数之前，必须把这个参数设为你的应用程序的缓冲区长度。

16.5.1 电话簿条目的增添

RAS有四个函数，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参数是一个指向字串的指针，这个字串用于识别现成的或新建的条目。 RASENTRY结构以这个名字存在，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。可以把空中止字串列表放在定义备用电话号码的 RASENTRY结构之后。最后一个字串的结尾是两个连续的空字串。 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，可用于获得一个现成电话簿条目的属性或一个新电话簿条目的值。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参数是一个指向一个字串的指针，该字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，这个缓冲区是你的应用程序为接收 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空中止字串数组，这些字串为请求的电话簿条目识别备用电话号码。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的那个字节数的DWORD，这个缓冲区是你的应用程序为 lpRasEntry参数提供的。RasSetEntry Properties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。如果你使用的是 Windows 2000和Windows NT，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，从这个数可看出为lpbDeviceInfo提供的缓冲区中包含的字节有多少。RasGetEntryProperties返回时，lpbDeviceInfoSize就会返回lpbDeviceInfo缓冲区内返回的字节数。

程序清单16-3演示了一个应用程序应该怎样利用 RasGetEntryProperties和RasSetEntry Properties来建立一个新的电话簿条目。

程序清单16-3 利用默认属性建立一个新的RAS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;

    // Get buffer sizing information for a default phonebook entry

    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) GlobalAlloc(GPTR, EntryInfoSize);

    if (DeviceInfoSize == 0)
        lpDeviceInfo = NULL;
    else
        lpDeviceInfo = (LPBYTE) GlobalAlloc(GPTR, DeviceInfoSize);

    // Get default phonebook entry
    lpRasEntry->dwSize = sizeof(RASENTRY);

    if ((Ret = RasGetEntryProperties(NULL, "", lpRasEntry,
```

```
            &EntryInfoSize, lpDeviceInfo, &DeviceInfoSize)) != 0)
        {
            printf("RasGetEntryProperties failed with error %d\n",
                Ret);
            return;
        }

        // Validate new phonebook name "Testentry"

        if ((Ret = RasValidateEntryName(NULL, "Testentry")) !=
            ERROR_SUCCESS)
        {
            printf("RasValidateEntryName failed with error %d\n",
                Ret);
            return;
        }

        // Install a new phonebook entry, "Testentry", using
        // default properties

        if ((Ret = RasSetEntryProperties(NULL, "Testentry",
            lpRasEntry, EntryInfoSize, lpDeviceInfo,
            DeviceInfoSize)) != 0)
        {
           printf("RasSetEntryProperties failed with error %d\n",
               Ret);
           return;
       }
    }
```

16.5.2 电话簿条目的重命名

现在，大家已对建立和修改电话簿条目有了一定的认识。接下来看看 RasRenameEntry函数。RasRenameEntry只允许对电话簿条目进行重命名。它的定义如下：

```
DWORD RasRenameEntry(
    LPCTSTR lpszPhonebook,
    LPCTSTR lpszOldEntry,
    LPCTSTR lpszNewEntry
);
```

lpszPhonebook参数是一个指向电话簿文件的指针。 lpszOldEntry参数是一个指向字串的指针，这个字串用于识别准备进行重命名的那个现成的电话簿条目。 lpszNewEntry参数是指向一个字串的指针，这个字串中包含了电话簿条目的新名字。你的应用程序在调用RasRenameEntry之前，应该根据新名字来调用 RasValidateEntryName。如果RasRenameEntry调用成功，就会返回0。如果失败，就会返回下面这些错误：

- ERROR_INVALID_NAME：表示lpszNewEntry名字无效。
- ERROR_ALREADY_EXISTS：表示电话簿中已存在lpszNewEntry名字。
- ERROR_CANNOT_FIND_PHONEBOOK_ENTRY：表示电话簿中未找到lpszOldEntry名。

16.5.3 电话簿条目的删除

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参数是一个字串，这个字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。

16.5.4 电话簿条目的列举

RAS提供了一个非常方便的函数 —— RasEnumEntries，它可以获得电话簿文件中所有可用的电话簿条目。它的定义如下：

```
DWORD RasEnumEntries (
    LPCTSTR reserved,
    LPCTSTR lpszPhonebook,
    LPRASENTRYNAME lprasentryname,
    LPDWORD lpcb,
    LPDWORD lpcEntries
);
```

reserved参数没有采用，但必须把它设为 NULL。lpszPhonebook参数是一个指向电话簿文件名的指针。lprasentryname参数是一个指向应用程序缓冲区的指针，这个缓冲区是你必须提供的，用来接收RASENTRYNAME结构数组。RASENTRYNAME结构的格式如下：

```
typedef struct _RASENTRYNAME
{
    DWORD dwSize;
    TCHAR szEntryName[RAS_MaxEntryName + 1];
#if (WINVER >= 0x500)
    DWORD dwFlags;
    CHAR  szPhonebookPath[MAX_PATH + 1];
#endif
} RASENTRYNAME;
```

字段如下定义：

- dwSize：在调用 RasEnumEntries之前，必须把它设为 RASENTRYNAME结构的长度（按字节算）。
- szEntryName：接收电话簿条目的名字。
- dwFlags：在Windows 2000平台上，这个标志表示电话簿条目是在采用REN_AllUsers标志的系统默认电话簿（前面曾讲过）内还是在采用REN_User标志的一个用户配置电话簿内。
- szPhonebookPath：Windows 2000平台上，指定到电话簿文件的完整路径。

这时，必须提供足够大的缓冲区，以便能够容纳若干个结构；若不然， RasEnumEntries调用就会失败，出现ERROR_BUFFER_TOO_SMALL错误。下一个参数lpcb是一个变量指针，表示列举条目所需要的字节数有多少。这个参数必须设为你自己的 lprasentryname缓冲区的长度（按字节算）。最后一个参数是 lpcEntries，它是一个变量指针，表示写入 lprasentryname缓冲区中的RASENTRYNAME结构数有多少。

16.5.5 用户凭据的管理

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这四个函数允许你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Windows95、Windows 98和Windows CE，只有采用RasGetEntryDialParams和RasSetEntryDialParams，它们可用于所有的平台。

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参数是一个代表现成电话簿条目的字串。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、密码和区域。 RASCREDENTIALS结构的格式如下：

```
typedef struct {
    DWORD dwSize;
    DWORD dwMask;
    TCHAR szUserName[UNLEN + 1];
    TCHAR szPassword[PWLEN + 1];
    TCHAR szDomain[DNLEN + 1];
} RASCREDENTIALS, *LPRASCREDENTIALS;
```

它的字段是这样定义的：

- dwSize：指定RASCREDENTIALS结构的长度（按字节算）。这个字段应该始终设为结构的长度。
- dwMask：是一个位掩码字段，通过预先定义标志，识别这个结构中接下来的哪三个字段是有效的。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。
- szUserName：是一个空中止字串，其中包含用户登录名。
- szPassword：是一个空中止字串，其中包含用户密码。
- szDomain：是一个空中止字串，其中包含用户登录的区域。

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，你的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，它还允许你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串（“ ”）值，凭据是在lpCredentials结构中dwMask字段内进行识别的。比

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是一样的。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结构中识别的。

16.5.6 多链接电话簿的子条目

Windows 2000和Windows NT平台上，RAS允许你对多链路电话簿条目进行管理，以增强通信能力。多链路条目能够使一个RAS连接上关联更多的通信设备，进而增大连接的总带宽。RAS允许你利用RasGetSubEntryProperties和RasSetSubEntryProperties这两个函数对多链路电话簿条目进行管理。RasGetSubEntryProperties函数的定义如下：

```
DWORD RasGetSubEntryProperties(
    LPCTSTR lpszPhonebook,
    LPCTSTR lpszEntry,
    DWORD dwSubEntry,
    LPRASSUBENTRY lpRasSubEntry,
    LPDWORD lpdwcb,
    LPBYTE lpbDeviceConfig,
    LPDWORD lpcbDeviceConfig
);
```

lpszPhonebook参数是一个指向电话簿文件名的指针。lpszEntry参数是一个电话簿条目。dwSubEntry参数指定的是一个子条目的索引，这个子条目包含在电话簿条目内。lpRasSubEntry参数是一个指向缓冲区的指针，这个缓冲区将接收一个RASSUBENTRY结构和一个供选择的备用电话号码列表。RASSUBENTRY结构的格式如下：

```
typedef struct tagRASSUBENTRY
{
```

```
    DWORD dwSize;
    DWORD dwfFlags;
    TCHAR szDeviceType[RAS_MaxDeviceType + 1];
    TCHAR szDeviceName[RAS_MaxDeviceName + 1];
    TCHAR szLocalPhoneNumber[RAS_MaxPhoneNumber + 1];
    DWORD dwAlternateOffset;
} RASSUBENTRY;
```

上面这个结构的定义是这样的：

■ dwSize：必须设成一个RASSUBENTRY结构的长度（按字节算）。

■ dwFlags：未用。

■ szDeviceType：接收一个代表设备类型的字串，该设备用于具体的连接上。

■ szDeviceName：接收TAPI设备的真名。

■ szLocalPhoneNumber：识别该设备所用的电话号码。

■ dwAlternateOffset：指定字节数，表示RASSUBENTRY结构的开始处到跟在该结构后面的连续空中止字串字节表之间的字节数是多少。

lpRasSubEntry缓冲区必须足够大，以便能够容纳一个RASSUBENTRY结构和备用的电话号码串；否则，RasGetSubEntryProperties就会失败，并返回ERROR_BUFFER_TOO_SMALL错误。lpdwcb参数应该设为你自己的lpRasSubEntry缓冲区中的字节数。函数返回时，lpdwcb将收到总的字节数，这个数表示包含RASSUBENTRY结构和备用电话号码需要这么多字节。lpbDeviceConfig和lpcbDeviceConfig参数都没有使用，应该设为NULL。

你可以建立一个新的子条目或对一个指定电话簿条目的现成的子条目进行修改，这是通过调用RasSetSubEntryProperties函数来完成的。该函数的定义如下：

```
DWORD RasSetSubEntryProperties(
    LPCTSTR lpszPhonebook,
    LPCTSTR lpszEntry,
    DWORD dwSubEntry,
    LPRASSUBENTRY lpRasSubEntry,
    DWORD dwcbRasSubEntry,
    LPBYTE lpbDeviceConfig,
    DWORD dwcbDeviceConfig
);
```

这个函数的各个参数中，lpRasSubEntry指定准备添加到电话簿中的子条目。其余参数则和RasGetSubEntryProperties中的参数一样。

16.6 连接管理

RAS有三个非常有用的函数。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、RasGetSubEntryHandle和RasGetProjectionInfo。RasEnumConnections函数列出了所有活动的RAS连接，它的定义如下：

```
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结构的格式如下：

```
typedef struct _RASCONN
{
    DWORD dwSize;
    HRASCONN hrasconn;
    TCHAR szEntryName[RAS_MaxEntryName + 1];
#if (WINVER >= 0x400)
    CHAR szDeviceType[RAS_MaxDeviceType + 1];
    CHAR szDeviceName[RAS_MaxDeviceName + 1];
#endif
#if (WINVER >= 0x401)
    CHAR szPhonebook[MAX_PATH];
    DWORD dwSubEntry;
#endif
#if (WINVER >= 0x500)
    GUID guidEntry;
#endif
} RASCONN;
```

该结构的字段定义如下：

- dwSize：指出RASCONN结构的长度（按字节算）。
- brasconn：取得RasDial建立的连接句柄。
- szEntryName：取得用来建立连接的电话簿条目。如果采用了空字串，这个字段就会返回一个带有句号（.）的字串，后面还有全部电话号码。
- szDeviceType：取得一个字串，描述连接所用的设备类型。
- szDeviceName：取得一个带有设备名的字串，该设备用于建立连接。
- szPhoneName：为建立连接的条目取得电话簿的完整路径。
- dwSubEntry：取得一个多链接电话簿条目的子条目索引。
- guidEntry：在Windows 2000平台上，为建立连接所用的电话簿条目取得一个 GUID。

对RasEnumConnections函数而言，需要把它投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，你的缓冲区中，第一个 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的一切字节。即使你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，取得被写成 lprasconn的RASCONN结构的个数。

RasGetSubEntryHandle函数的定义在下一页。它允许你为多链接连接的一个具体的子条目取得一个连接句柄。

```
DWORD RasGetSubEntryHandle(
    HRASCONN hrasconn,
    DWORD dwSubEntry,
    LPHRASCONN lphrasconn
);
```

brasconn参数是一个多链接连接的RAS连接句柄。dwSubEntry参数是多链接连接中的一个

设备的子条目索引。lpbrasconn参数为子条目设备取得连接句柄。

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。这个新的网络协议特有的信息就是人们所说的“ Projection information ”（投影信息）。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，在一个分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（比如分配得到的IP地址）就会从RAS服务到客户机之间建立起来。要想获得 PPP分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列举类型，允许你指定一个协议来接收连接信息。 lpprojection参数取得一个数据结构，这个结构和rasprojection中指定的列举类型有关联。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。这个函数结束时，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列举类型允许你取得连接信息：

- RASP_Amb。
- RASP_PppNbf。
- RASP_PppIpx。
- RASP_PppIp。

如果指定一个RASP_Amb类型，就会收到一个RASAMB结构，该结构的格式如下：

```
typedef struct _RASAMB
{
    DWORD dwSize;
    DWORD dwError;
    TCHAR szNetBiosError[NETBIOS_NAME_LEN + 1];
    BYTE bLana;
} RASAMB;
```

它的各个字段是这样定义的：

- dwSize：应该设为RASAMB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。
- szNetBiosError：如果在 PPP身份验证进程中，发生了名字冲突，就会收到一个NetBIOS名。如果dwError返回ERROR_NAME_EXISTS_ON_NET，szNetBiosError就会收到导致这一错误的那个名字。
- bLana：对用于建立远程访问连接的NetBIOS LAN适配器（LANA）的编号进行识别。

如果把RASP_PppNbf指定为RasGetProjectionInfo，就会收到一个RASPPPNBF结构，该结构的定义如下：

```
typedef struct _RASPPPNBF
{
    DWORD dwSize;
    DWORD dwError;
```

```
    DWORD dwNetBiosError;
    TCHAR szNetBiosError[NETBIOS_NAME_LEN + 1];
    TCHAR szWorkstationName[NETBIOS_NAME_LEN + 1];
    BYTE bLana;
} RASPPPNBF;
```

RASPPPNBF的字段和RASAMB结构中的字段一样，但是，前者包含的字段比后者多了两个：szWorkstationName和 dwNetBiosError。szWorkstationName字段收到NetBIOS名，这个名用来识别网络上你正在连接的那个工作站。 dwNetBiosError字段收到的是发生的NetBIOS错误。

如果把RASP_PppIpx列举类型指定为RasGetProjectionInfo，收到的则是一个RASPPPIPX结构，该结构的定义如下：

```
typedef struct _RASPPPIPX
{
    DWORD dwSize;
    DWORD dwError;
    TCHAR szIpxAddress[RAS_MaxIpxAddress + 1];
} RASPPPIPX;
```

它的字段是这样定义的：

- 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：取得一个字串，该字串代表客户机的 IPX地址。

如果把RASP_PppIp列举类型指定为RasGetProjectionInfo，就会收到RASPPPIP结构。该结构的格式如下：

```
typedef struct _RASPPPIP
{
    DWORD dwSize;
    DWORD dwError;
    TCHAR szIpAddress[RAS_MaxIpAddress + 1];
    TCHAR szServerIpAddress[RAS_MaxIpAddress + 1];
} RASPPPIP;
```

它的各个字段是这样定义的：

- dwSize：应该设为一个RASPPPIP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：取得一个代表客户机IP地址的字串。
- szServerIpAddress：取得一个代表服务器IP地址的字串。

对在RAS基础上建立的IP连接来说，怎样才能获得为它分配的 IP地址呢？看看程序清单16-4，你就明白了。

程序清单16-4 在一个IP连接上使用RasGetProjectionInfo

```
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);
}
```

16.7 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充分挖掘RAS的潜力。本章最后，对RAS和本书第三部分进行了全面的总结。

第四部分 附 录

附录A NetBIOS命令索引

本附录陈列并解释了可为NCB结构的ncb_command字段选用的有效命令，必须将这个结构传递给Netbios函数。在每个命令说明中，我们都提供了一张表格。表中说明了必须设置的NCB字段，以及优先返回的由Netbios函数设置的字段。每个表的第二列说明指定的NCB结构字段是一个输入参数，还是一个输出参数。第三列说明执行NetBIOS调用时，是否必须设置这个字段。如果出现一个X，就必须提供一个值。否则，如果这个字段是一个输入参数并且没有X，则可以是任意值。关于Netbios的详情，可参阅第1章。

A.1 NCBADDGRNAME

这个命令用于在本地名字表添加一个组名。添加的组名必须是独一无二的，但任何人都可把它用作一个组名。组名最常见的用法是用作数据报接收端。它的名字编号在数据报操作的ncb_num字段中返回。字段设置参见表A-1。

表 A-1

字 段	输入/输出	是否要求
ncb_command	输入	X
ncb_retcode	输出	X
ncb_lsn		
ncb_num	输出	
ncb_buffer		
ncb_length		
ncb_callname		
ncb_name	输入	X
ncb_rto		
ncb_sto		
ncb_post	输入	
ncb_lana_num	输入	X
ncb_cmd_cplt	输出	
ncb_event	输入	

A.2 NCBADDNAME

这个命令用于在本地名字表添加一个唯一名中。在整个网络中，唯一名必须是独一无二的，否则，就会返回错误。它的名字编号在数据报操作所用的 ncb_num字段中返回。参见表A-2。

表 A-2

字 段	输入/输出	是否要求
ncb_command	输入	X
ncb_retcode	输出	
ncb_lsn		
ncb_num	输出	
ncb_buffer		
ncb_length		
ncb_callname		
ncb_name	输入	X
ncb_rto		
ncb_sto		
ncb_post	输入	
ncb_lana_num	输入	X
ncb_cmd_cplt	输出	X
ncb_event	输入	

A.3 NCBASTAT

这个命令用于取得本地或远程适配器的状态。在调用这个命令时，要把 ncb_buffer设为一个缓冲区，这个缓冲区中包括一个 ADAPTER_STATUS结构及跟在这个结构后面的一个NAME_BUFFER结构数组。参见表A-3。

表 A-3

字 段	输入/输出	是否要求
ncb_command	输入	X
ncb_retcode	输出	
ncb_lsn		
ncb_num		
ncb_buffer	输入/输出	X
ncb_length	输入/输出	X
ncb_callname	输入	X
ncb_name		
ncb_rto		
ncb_sto		
ncb_post	输入	
ncb_lana_num	输入	X
ncb_cmd_cplt	输出	
ncb_event	输入	

A.4 NCBCALL

这个命令用于建立会话与另一个进程（ncb_name字段中指明的）的连接。参见表A-4。

表 A-4

字 段	输入/输出	是否要求
ncb_command	输入	X

(续)

字 段	输入/输出	是否要求
ncb_retcode	输出	
ncb_lsn	输出	
ncb_num		
ncb_buffer		
ncb_length		
ncb_callname	输入	X
ncb_name	输入	X
ncb_rto	输入	
ncb_sto	输入	
ncb_post	输入	
ncb_lana_num	输入	X
ncb_cmd_cplt	输出	
ncb_event	输入	

A.5 NCBCANCEL

这个命令用于取消上一次未完成的命令。ncb_buffer字段指向NCB结构，这个结构中有准备取消的操作。取消NCBSEND或NCBCHAINSEND命令都会中止会话；然而，这两个函数的无确认变体函数都没有取消它们各自的会话。下面的命令是不能被取消的： NCBADDGRNAME、NCBADDNAME、NCBCANCEL、NCBDELINAME、NCBRESET、NCBDGSEND、NCBDGSENDBC和NCBSSTAT。参见表A-5。

表 A-5

字 段	输入/输出	是否要求
ncb_command	输入	X
ncb_retcode	输出	
ncb_lsn		
ncb_num		
ncb_buffer	输入	X
ncb_length		
ncb_callname		
ncb_name		
ncb_rto		
ncb_sto		
ncb_post		
ncb_lana_num	输入	X
ncb_cmd_cplt	输出	
ncb_event		

A.6 NCBCHAINSEND

这个命令向指定的接收端发送两个缓冲区的内容。可发送的数据量最大为 128KB（各个缓冲区的最大值是64KB）。在ncb_buffer和ncb_length中，指定第一个缓冲区并指定它的长度。利用ncb_callname的0~1字节指定第二个缓冲区的长度，并用2~5字节表示它。参见表A-6。

表 A-6

字 段	输入/输出	是否要求
ncb_command	输入	X
ncb_retcode	输出	
ncb_lsn	输入	X
ncb_num		
ncb_buffer	输入	X
ncb_length	输入	X
ncb_callname	输入	X
ncb_name		
ncb_rto		
ncb_sto		
ncb_post	输入	
ncb_lana_num	输入	X
ncb_cmd_cplt	输出	
ncb_event	输入	

A.7 NCBCHAINSENDNA

这个命令向指定的接收端发送两个缓冲区的内容，不等待接收端的收到确认。可发送的数据最大量为128KB（各个缓冲区的最大值是64KB）。分别在ncb_buffer和ncb_length中指定第一个缓冲区及其长度。利用ncb_callname的0~1字节指定第二个缓冲区的长度，并用2~5字节表示它。参见表A-7。

表 A-7

字 段	输入/输出	是否要求
ncb_command	输入	X
ncb_retcode	输出	
ncb_lsn	输入	X
ncb_num		
ncb_buffer	输入	X
ncb_length	输入	X
ncb_callname	输入	X
ncb_name		
ncb_rto		
ncb_sto		
ncb_post	输入	
ncb_lana_num	输入	X
ncb_cmd_cplt	输出	
ncb_event	输入	

A.8 NCBDELNAME

这个命令从本地名字表中删除一个名字。如果即将删除的名字与活动会话相关，就会返回NRC_ACTSES（0x0F）错误。如果活动会话命令尚未完成，就会收到 NRC_NAMERR（0x17）错误。参见表A-8。

表 A-8

字 段	输入/输出	是否要求
ncb_command	输入	X
ncb_retcode	输出	
ncb_lsn		
ncb_num		
ncb_buffer		
ncb_length		
ncb_callname		
ncb_name	输入	X
ncb_rto		
ncb_sto		
ncb_post	输入	
ncb_lana_num	输入	X
ncb_cmd_cplt	输出	
ncb_event	输入	

A.9 NCBDGRECV

这个命令收到一个直接指向本地名的数据报，本地名和 ncb_num值相关。如果ncb_num是0xFF，该命令就会收到直接指向任何一个本地名的数据报。本地名既可以是一个组名，又可以是一个唯一名。如果发送一个数据报时，没有待决的接收数据报命令，这个数据报就会丢失。如果提供的缓冲区太小，就会出现这条消息“未完成错误”，NRC_INCOMP（0x06），数据就会被截断，以适应缓冲区的长度。参见表 A-9。

表 A-9

字 段	输入/输出	是否要求
ncb_command	输入	X
ncb_retcode	输出	
ncb_lsn		
ncb_num	输入	X
ncb_buffer	输入	X
ncb_length	输入/输出	X
ncb_callname	输出	
ncb_name		
ncb_rto		
ncb_sto		
ncb_post	输入	
ncb_lana_num	输入	X
ncb_cmd_cplt	输出	
ncb_event	输入	

A.10 NCBDGRECVBC

这个命令接收源于执行发送数据报命令的名字的广播数据报。如果提供的缓冲区太小，就会产生“未完成错误”NRC_INCOMP（0x06），为了与缓冲区适应，便截断这个数据。参

见表A-10。

表 A-10

字 段	输入/输出	是否要求
ncb_command	输入	X
ncb_retcode	输出	
ncb_lsn		
ncb_num	输入	X
ncb_buffer	输入	X
ncb_length	输入/输出	X
ncb_callname	输出	
ncb_name		
ncb_rto		
ncb_sto		
ncb_post	输入	
ncb_lana_num	输入	X
ncb_cmd_cplt	输出	
ncb_event	输入	

A.11 NCBDGSEND

这个命令向指定名发送一个数据报。这个名字既可以是一个唯一名，又可以是一个组名。如果一个适配器有一个针对同一个名字的待发接收数据报命令，它就会收到自己的消息。数据报的最大长度与基层协议有关。要想得知数据报的最大长度，执行本地 NCBASTAT命令即可。从返回的 ADAPTER_STATUS结构中，便可得知基层传送协议要求的数据报最大长度。参见表A-11。

表 A-11

字 段	输入/输出	是否要求
ncb_command	输入	X
ncb_retcode	输出	
ncb_lsn		
ncb_num	输入	X
ncb_buffer	输入	X
ncb_length	输入	X
ncb_callname	输入	X
ncb_name		
ncb_rto		
ncb_sto		
ncb_post	输入	
ncb_lana_num	输入	X
ncb_cmd_cplt	输出	
ncb_event	输入	

A.12 NCBDGSENDBC

这个命令向LAN上的各台主机发送广播数据报。只有有未完成的接收数据报命令的机器

才能收到这条消息。同时，如果本地适配器中有一个待发接收数据报命令，它就会收到它自己的消息。广播数据报和NCBDGSEND条目中提到的长度限制是一样的。参见表 A-12。

表 A-12

字 段	输入/输出	是否要求
ncb_command	输入	X
ncb_retcode	输出	
ncb_lsn		
ncb_num	输入	X
ncb_buffer	输入	X
ncb_length	输入	X
ncb_callname		
ncb_name		
ncb_rto		
ncb_sto		
ncb_post	输入	
ncb_lana_num	输入	X
ncb_cmd_cplt	输出	
ncb_event	输入	

A.13 NCBENUM

这个命令列举LANA编号。在执行这个命令时，要把ncb_buffer设为一个LANA_ENUM结构。该命令返回时， LANA_ENUM的length字段会返回本地机器上的 LANA编号的数目。LANA_ENUM的lana字段内填入的是LANA编号。参见表A-13。

表 A-13

字 段	输入/输出	是否要求
ncb_command	输入	X
ncb_retcode	输出	
ncb_lsn		
ncb_num		
ncb_buffer	输入	X
ncb_length	输入	X
ncb_callname		
ncb_name		
ncb_rto		
ncb_sto		
ncb_post		
ncb_lana_num	输入	X
ncb_cmd_cplt	输出	
ncb_event		

A.14 NCBFINDNAME

这个命令在网络上查找一个名字的位置（即机器名）。在发出这个命令时， ncb_buffer中就会填入FIND_NAME_HEADER结构，以及一个或多个FIND_NAME_BUFFER结构。这个命

令是Windows NT专有的，尚未获得所有Win32平台的支持。参见表A-14。

表 A-14

字 段	输入/输出	是否要求
ncb_command	输入	X
ncb_retcode	输出	
ncb_lsn		
ncb_num		
ncb_buffer	输入/输出	X
ncb_length	输入	X
ncb_callname	输入	X
ncb_name		
ncb_rto		
ncb_sto		
ncb_post	输入	
ncb_lana_num	输入	X
ncb_cmd_cplt	输出	
ncb_event	输入	

A.15 NCBHANGUP

这个命令关闭一个指定的连接会话。这个会话的所有未完成的接收命令都将中断，并返回“会话已关闭”错误NRC_SCLOSED（0xOA）。如果发送或链式发送命令未完成，挂起命令就会延迟，直到命令结束。不管这些命令是在传输数据，还是等待远程会话方发出接收命令，都会产生这样的延迟。另外，如果同时存在若干个完成的 NCBRECVANY命令，那么在会话已经关闭时，则只能有一个命令返回错误代码。而对其他任何一个接收命令来说，每个未完成的接收命令都会返回错误。参见表A-15。

表 A-15

字 段	输入/输出	是否要求
ncb_command	输入	X
ncb_retcode	输出	
ncb_lsn	输入	X
ncb_num		
ncb_buffer		
ncb_length		
ncb_callname		
ncb_name		
ncb_rto		
ncb_sto		
ncb_post	输入	
ncb_lana_num	输入	X
ncb_cmd_cplt	输出	
ncb_event	输入	

A.16 NCBLANSTALERT

这是一个只适用于Windows NT的命令，它通知用户持续一分钟便会失败的LAN故障。然

而在测试过程中，这个命令通知几个常见的 LAN故障，比如说网络线缆已断开。参见表 A-16。

表 A-16

字 段	输入/输出	是否要求
ncb_command	输入	X
ncb_retcode	输出	
ncb_lsn		
ncb_num		
ncb_buffer		
ncb_length		
ncb_callname		
ncb_name		
ncb_rto		
ncb_sto		
ncb_post		
ncb_lana_num	输入	X
ncb_cmd_cplt	输出	
ncb_event		

A.17 NCBLISTEN

这个命令对来自另一个进程（包括本地的或远程的）的连接进行监听。如果 ncb_callname 的第一个字符是星号（*），就会用网络适配器（由它向本地名发出一个 NCBCALL）来建立一个会话。执行NCBCALL的名字在ncb_callname字段中返回。如果指定发送或接收超时，新会话上执行的所有发送和接收调用都将遵循这一超时设置。参见表 A-17。

表 A-17

字 段	输入/输出	是否要求
ncb_command	输入	X
ncb_retcode	输出	
ncb_lsn	输出	
ncb_num		
ncb_buffer		
ncb_length		
ncb_callname	输入/输出	X
ncb_name	输入	X
ncb_rto	输入	
ncb_sto	输入	
ncb_post	输入	
ncb_lana_num	输入	X
ncb_cmd_cplt	输出	
ncb_event	输入	

A.18 NCBRECV

这个命令接收源于指定会话名的数据。如果此时未完成的接收数据命令有若干个，就是按照下面的顺序对这些命令进行处理：

1) 接收（NCBBECV）。

2) 接收指定名的所有数据（NCBRECVANY）。

3) 接收任何一个名字的所有数据（NCBRECEANY）。

同一个优先级的命令则按照“ 先进先出 ”的顺序处理。如果缓冲区容纳不下整个数据，就会返回NRC_INCOMP（0x06）错误。发生这种情况时，就要执行另一个缓冲区较大的接收命令，或者用已经到期的或无确认的超时来执行发送命令 —— 这种情况下，会丢失数据。ncb_length字段设为实际上返回的数据量。参见表 A-18。

表 A-18

字 段	输入/输出	是否要求
ncb_command	输入	X
ncb_retcode	输出	
ncb_lsn	输入	X
ncb_num	输入	X
ncb_buffer	输入	X
ncb_length	输入/输出	X
ncb_callname		
ncb_name		
ncb_rto		
ncb_sto		
ncb_post	输入	
ncb_lana_num	输入	X
ncb_cmd_cplt	输出	
ncb_event	输入	

A.19 NCBRECVANY

这个命令接收的数据是与任何一个指定名相对应的会话发出的。另外，通过把 ncb_num字段设为0xFF的方式，这个命令还可以接收目标地址是本地名的数据。不然的话，将 ncb_num设为网络编号也行，这个网络编号是在本地名字表添加了一个名字之后返回的。然后，该命令把等待指定名的数据捡起。另外，如果同时有若干个未完成的接收命令，就按优先级顺序进行处理。详情参见NCBRECV。

如果一个会话的关闭是一个本地会话关闭命令，或远程会话方关闭会话或一个会话中止命令执行的，指定名所有未完成的 NRCRECVANY命令最终都会返回错误 NRC_SCLOSED（0x0A）；NCB结构的ncb__lsn字段设为已断开的本地会话编号。如果这个已关闭的会话中，NCBRECVANY命令没有等待指定的名字，却存在任意会话（ ncb_num是0xFF）的未完成的NCBRECVANY命令，那么该命令就会返回 NRC_SCLOSED错误，而且ncb_lsn字段则设为相应的会话编号。参见表A-19。

表 A-19

字 段	输入/输出	是否要求
ncb_command	输入	X
ncb_retcode	输出	
ncb_lsn	输出	

(续)

字 段	输入/输出	是否要求
ncb_num	输入/输出	X
ncb_buffer	输入	X
ncb_length	输入/输出	X
ncb_callname		
ncb_name		
ncb_rto		
ncb_sto		
ncb_post	输入	
ncb_lana_num	输入	X
ncb_cmd_cplt	输出	
ncb_event	输入	

A.20 NCBRESET

这个命令对指定的LANA编号进行重新设置，并对某些环境资源产生影响。

- 如果ncb_lsn不是0，与ncb_lana_num相关的所有资源都会被释放。
- 如果ncb_lsn是0，与ncb_lana_num相关的所有资源都会被释放，并分配新的资源。ncb_callname[0]字节指定会话最多能够有多少， ncb_callname[2]字节指定名字最多有多少， ncb_callname[3]则要求应用程序使用计算机名（其名编号为 1）。参见表A-20。

表 A-20

字 段	输入/输出	是否要求
ncb_command	输入	X
ncb_retcode	输出	
ncb_lsn	输入	X
ncb_num	输入	X
ncb_buffer		
ncb_length		
ncb_callname		
ncb_name		
ncb_rto		
ncb_sto		
ncb_post		
ncb_lana_num	输入	X
ncb_cmd_cplt	输出	
ncb_event		

A.21 NCBSEND

这个命令向指定的对话方发送数据。可传送的数据最多为 65 536（即64KB）。如果远程会话方发出一个挂起命令，所有的未完成的发送命令就会返回“会话已关闭”错误NRC_SCLOSED（0x0A）。如果若干个发送命令等待执行，就会按照“先来先出的顺序”进行处理。参见表A-21。

表 A-21

字 段	输入/输出	是否要求
ncb_command	输入	X
ncb_retcode	输出	
ncb_lsn	输入	X
ncb_num		
ncb_buffer	输入	X
ncb_length	输入	X
ncb_callname		
ncb_name		
ncb_rto		
ncb_sto		
ncb_post	输入	
ncb_lana_num	输入	X
ncb_cmd_cplt	输出	
ncb_event	输入	

A.22 NCBSENDNA

这个命令向指定会话发送数据，但不等待会话方返回的确认。否则，这个命令的行为就和NCBSEND的行为一样。参见表A-22。

表 A-22

字 段	输入/输出	是否要求
ncb_command	输入	
ncb_retcode	输出	
ncb_lsn	输入	X
ncb_num		
ncb_buffer	输入	X
ncb_length	输入	X
ncb_callname		
ncb_name		
ncb_rto		
ncb_sto		
ncb_post	输入	
ncb_lana_num	输入	X
ncb_cmd_cplt	输出	
ncb_event	输入	

A.23 NCBSSTAT

这个命令取得会话状态。在调用这个命令时， ncb_buffer被设为一个内存块，将用后跟一个或多个SESSION_BUFFER结构的一个SESSION_HEADER结构来填充这个内存块。如果ncb_name的第一个字节是一个星号（*），这个命令就可获得与本地名字表中所有名字相关的所有会话的状态。如果提供的缓冲区太小，就返回 NRC_INCOMP（0x06）错误。如果缓冲区的长度小于4，返回的错误就是NRC_BUFLEN（0x01）。参见表A-23。

表 A-23

字 段	输入/输出	是否要求
ncb_command	输入	X
ncb_retcode	输出	
ncb_lsn		
ncb_num	输出	
ncb_buffer	输入	X
ncb_length	输入	X
ncb_callname		
ncb_name	输入	X
ncb_rto		
ncb_sto		
ncb_post	输入	
ncb_lana_num	输入	X
ncb_cmd_cplt	输出	
ncb_event	输入	

A.24 NCBUNLINK

这个命令断开适配器连接，并为 NetBIOS的前期版本提供兼容能力。该命令不能用于Win32平台。

附录B　IP助手函数

本附录介绍一些新的API函数，有了这些函数，便可在自己的计算机上对 IP协议统计情况进行查询和管理。它们有助于获得下面的能力：

- Ipconfig.exe（或适用于微软Windows 95的Winipcfg.exe）：显示IP配置信息，允许释放和更新DHCP分配的IP地址。
- 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“地址解析协议”（ARP）使用的IP到物理地址翻译表。

本附录介绍的这些函数主要用于 Windows 98和Windows 2000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Iphlpapi,h文件中。在建立你自己的应用程序时，必须把它链接到这个库文件 Iphlpapi.lib。

B.1　IPCONFIG

Ipconfig.exe程序展示了两条信息：IP配置信息和IP配置参数，参数是由安装在机器上的网络适配器所决定的。要获得IP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，你的应用程序必须提供这个结构，以便获得 IP配置信息。pOutBufLen参数是一个变量指针，指定投入pFixedInfo参数中的那个缓冲区的长度。如果你提供的缓冲区不够大，GetNetworkParams就会返回ERROR_BUFFER_OVERFLOW，并将pOutBufLen参数设为正确的缓冲区长度。GetNetworkParams中所用的FIXED_INFO结构的格式如下：

```
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：代表你的计算机名，这个名字由DNS进行识别。

- DomainName：说明你的计算机属于哪个DNS域。
- CurrentDnsServer：包含当前DNS服务器的IP地址。
- DnsServerList：是一个链接列表，其中包含你的机器所采用的 DNS服务器。
- NodeType：说明IP网络上的系统是如何解析NetBIOS名的。表B-1包含了该字段可能的值
- ScopeId：识别一个字串值。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 IP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不能解析的名字、广播或 LMHOST文件。

表B-1　节点类型可能的值

值	说　明
BROADCAST_NODETYPE:	即b节点NetBIOS名字解析法，采用了这种解析方法，系统便利用IP广播来执行NetBIOS名字注册和名字解析
PEER_TO_PEER_NODETYPE:	即p节点名字解析，采用了这种解析方法，系统便利用点到点通信与一个NetBIOS名字服务器（比如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名字解析法，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名字解析法，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IP地址链接列表的起始处。它的格式如下：

```
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服务器之IP地址。如果把它设为NULL，就表明列表中的最后一个地址。IpAddress字段是一个字符串，以点式十进制字串表示 Ip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IP地址关联在一起。最后一个字段是Context，用一个独一无二的值来标识系统上的 IP地址。

另外，利用IpConfig.exe程序，也可获得网络接口专有的 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
);
```

通过pAdapterInfo参数，把一个指针投递给应用程序提供的缓冲区，这个缓冲区取得一个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，这个变量指定投入pAdapterInfo参数中的缓冲区的长度。如果你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，你的机器上所有可用的网络适配器的IP配置信息都在这个列表中。IP_ADAPTER_INFO的格式如下：

```
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值表明列表中的最后一个适配器。
- ComboIndex：未用，将设为0。
- AdapterName：适配器名。
- Description：是一个关于适配器的简单说明。
- AddressLength：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：该适配器分配的唯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字化的值。表 B-2定义了已获支持的适配器类型。
- DhcpEnabled：说明这个适配器上是否启用 DHCP。
- CurrentIpAddress：未用，但将设为NULL（空）值。
- IpAddressList：指定为这个适配器分配的 IP地址列表。
- GatewayList：指定代表默认网关的IP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S所用的IP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IP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IP地址。
- LeaseObtained：标识何时开始租用DHCP服务器提供的IP地址。
- LeaseExpires：标识DHCP服务器提供的IP地址何时期满。

表B-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（光纤分布数据接口）适配器
MIB_IF_TYPE_LOOPBACK	Loopback适配器
MIB_IF_TYPE_OTHER	其他类型的适配器
MIB_IF_TYPE_PPP	PPP（点到点协议）适配器
MIB_IF_TYPE_SLIP	Slip适配器
MIB_IF_TYPE_TOKENRING	令牌环适配器

B.1.1 释放和更新IP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IP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 IP地址，则可调用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参数是一个IP_ADAPTER_INDEX_MAP结构，该结构识别即将对其地址进行释放和更新的适配器。 IP_ADAPTER_INDEX_MAP结构的格式如下：

```
typedef struct _IP_ADAPTER_INDEX_MAP
{
    ULONG Index;
    WCHAR Name[MAX_ADAPTER_NAME];
}IP_ADAPTER_INDEX_MAP, *PIP_ADAPTER_INDEX_MAP;
```

它的各个字段是这样定义的：

- 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，这个变量指定投入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结构的格式如下；

```
typedef struct _IP_INTERFACE_INFO
{
    LONG                  NumAdapters;
    IP_ADAPTER_INDEX_MAP Adapter[1];
} IP_INTERFACE_INFO,*PIP_INTERFACE_INFO;
```

它的各个字段是这样定义的：

- 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IP地址了。

B.1.2 改变IP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，有两个函数允许你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场景编号。Windows中，每个网络适配器都有一个独一无二的索引ID（前面已讲过），而且，每个IP地址都有一个独一无二的场景ID。适配器索引ID和IP场景编号都可通过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IP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IP地址关联的场景值。NTEInstance参数取得与一个IP地址关联的实例值。

如果想通过编程删除适配器的IP地址，调用DeleteIpAddress函数即可。它的定义如下：

```
DWORD DeleteIPAddress (
    ULONG NTEContext
);
```

NTEContext参数是与IP地址关联的值，这个值可从GetAdaptersInfo（前面已介绍）中得到。

B.2 NETSTAT

Netstat.exe程序显示了你的计算机上的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不仅可用于Windows 98和Windows 2000，而且可用于Windows NT 4 SP4（以及稍后的版本）。

B.2.1 取得TCP连接表

利用GetTcpTable函数，可获得TCP连接表。获得的信息和你在执行带上 -p tcp-a选项的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，该缓冲区内接收TCP连接信息。pdwsize参数是一个变量指针，这个变量指定投入 pTcpTable参数中的缓冲区长度。如果你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结构的格式如下：

```
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IP地址对。它的格式如下：

```
typedef struct _MIB_TCPROW
{
    DWORD dwState;
    DWORD dwLocalAddr;
    DWORD dwLocalPort;
    DWORD dwRemoteAddr;
    DWORD dwRemotePort;
} MIB_TCPROW, *PMIB_TCPROW;
```

它的各个字段是这样定义的；

- dwState：指定TCP连接的状态，参见表B-3中的说明。

表B-3 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

(续)

连接状态	RFC说明
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
MIB_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IP地址。
- dwLocalPort：为连接指定一个本地端口。
- dwRemoteAddr：为连接指定远程IP地址。
- dwRemotePort：为连接指定远程端口。

B.2.2 取得UDP监听者表

利用GetUdpTable函数，可获得UDP监听者表。获得的信息和你在执行有一个 -pudp -a选项的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，这个变量指定投入 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结构格式如下：

```
typedef struct _MIB_UDPTABLE
{
    DWORD dwNumEntries;
    MIB_UDPROW table[ANY_SIZE];
} MIB_UDPTABLE, * PMIB_UDPTABLE;
```

它的各个字段是这样定义的：

- 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IP地址，UDP正在这些地址上监听数据报。它的格式如下：

```
typedef struct _MIB_UDPROW
{
    DWORD dwLocalAddr;
    DWORD dwLocalPort;
} MIB_UDPROW, * PMIB_UDPROW;
```

该结构的各个字段是这样定义的：

- dwLocalAddr：指定本地IP地址。
- dwLocalPort：指定本地IP端口。

B.2.3 取得IP协议统计情况

用于获得IP统计情况的四个函数是：GetIpStatistics、GetIcmpStatistics、GetTcpStatistics以及GetUdpStatistics。这些函数产生的信息和调用带上 -s参数的Netstat.exe程序时返回的信息一样。利用第一个统计函数 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，这个结构接收你的计算机当前的 IP统计情况。MIB_IPSTATS结构的格式如下：

```
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;
```

该结构的各个字段是这样定义的：

- dwForwarding：说明你的计算机上是启用，还是禁止转发 IP包。
- dwDefaultTTL：为你的计算机所发出的数据报指定最初的生存期（ 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IP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：说明你的计算机上可用的IP接口有多少。
- dwNumAddr：说明你的计算机上标识的IP地址有多少。
- dwNumRoutes：说明路由表中可用的路由有多少。

第二个统计函数：GetIcmpStatistics，用于获得“互联网控制协议”（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，这个结构接收你的计算机上当前的ICMP统计情况。MIB_ICMP结构的格式如下：

```
typedef struct _MIB_ICMP
{
    MIBICMPINFO stats;
} MIB_ICMP,*PMIB_ICMP;
```

一眼可见，这个MIB_ICMP结构中另包含一个MIBICMPINFO结构，后者的格式如下：

```
typedef struct _MIBICMPINFO
{
    MIBICMPSTATS icmpInStats;
    MIBICMPSTATS icmpOutStats;
} MIBICMPINFO;
```

MIBICMPINFO结构通过MIBICMPSTATS结构接收接入或外出的ICMP信息。icmpInStats参数接收接入数据，而icmpOutStats参数则接收外出数据。MIBICMPSTATS结构的格式如下：

```
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;
```

它的各个字段是这样定义的：

- 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：说明已收发多少生存期已过消息。
- 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响应请求。
- dwEchoReps：说明已收发多少ICMP响应应答。
- dwTimestamps：说明已收发多少时间戳请求。
- dwTimestampReps：说明已收发多少时间戳响应。
- dwAddrMasks：说明已收发多少地址掩码。
- dwAddrMaskReps：说明已收发多少地址掩码响应。

用于获得TCP统计情况的第三个统计函数是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，这个结构接收你的计算机当前的IP统计。MIB_TCPSTATS结构的格式如下：

```
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;
```

它的各个字段是这样定义的：

- 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：说明你的计算机向服务器发起了多少次连接。
- dwPassiveOpens：说明你的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：说明已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，用于获得你的计算机上的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，这个结构接收你的计算机当前的IP统计。MIB_UDPSTATS结构的格式如下：

```
typedef struct _MIB_UDPSTATS
{
    DWORD dwInDatagrams;
    DWORD dwNoPorts;
    DWORD dwInErrors;
    DWORD dwOutDatagrams;
    DWORD dwNumAddrs;
} MIB_UDPSTATS,*PMIB_UDPSTATS;
```

它的各个字段是这样定义的：

- 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。

B.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IP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 2000以及Windows NT 4 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

公制。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。

B.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
);
```

第一个参数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：

1) 目标地址。
2) 生成路由的协议。
3) 多路径路由策略。
4) 下一跳的地址。

路由信息包含在MIB_INFORWARDROW结构中，这个结构的格式如下：

```
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结构的格式如下：

```
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;
```

它的各个字段是这样定义的：

- dwForwardDest：是目标主机的IP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的服务类型”（TOS）的形式。关于TOS的详情，可参考第9章和IP_TOS选项。关于多路径路由的详情，参考1354。
- dwForwardNextHop：是路由表中下一跳的IP地址。
- dwForwardIfIndex：表明针对该路由的接口之索引。
- dwForwardType：表明RFC 1354中定义的路由类型。表B-4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 B-5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：是路由协议专有的公制值。详情参见RFC 1354。这个字段中包含路由公制值，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四个字段而言，其值均为MIB_IPROUTE_METRIC_UNUSED（-1）。
- dwForwardMetric2：是路由协议专有的公制值。详情参见 RFC 1354。
- dwForwardMetric3：是路由协议专有的公制值。详情参见 RFC 1354。
- dwForwardMetric4：是路由协议专有的公制值。详情参见 RFC 1354。
- dwForwardMetric5：是路由协议专有的公制值。详情参见 RFC 1354。

表B-4 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B-5 路由协议标识符

协议标识符	说 明
MIB_IPPROTO_OTHER	未列出的协议
MIB_IPPROTO_LOCAL	堆栈生成的路由
MIB_IPPROTO_NETMGMT	Route.exe程序或SNMP添加的路由器
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	HELLO路由协议
MIB_IPPROTO_RIP	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	IPCisco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

(续)

协议标识符	说 明
MIB_IPPROTO_NT_AUTOSTATIC	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“按需拨号”（DOD）
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

B.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 IP、网络掩码、网关、本地IP接口以及公制。添加路由时，应该保证所给的本地 IP接口是有效的。除此以外，还需要根据本地IP接口的内部索引值，引用这个接口，因为路由将添加到这个接口上。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
);
```

第一个参数pIpAddrTable，是将返回MIB_IPADDRTABLE结构的缓冲区的正确长度。pdwSize参数是被当作第一个参数投递的缓冲区的长度。最后一个参数 bOrder，说明是否返回以升序的方式处理IP地址的本地IP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便指定正确的缓冲区长度。MIB_IPADDRTABLE结构的格式如下：

```
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MIB_IPADDROW条目。MIB_IPADDRROW结构的格式如下：

```
typedef struct _MIB_IPADDRROW
{
    DWORD    dwAddr;
    DWORD    dwIndex;
    DWORD    dwMask;
    DWORD    dwBCastAddr;
    DWORD    dwReasmSize;
    unsigned short unused1;
    unsigned short unused2;
} MIB_IPADDRROW, *PMIB_IPADDRROW;
```

它的各个字段是这样定义的：

- dwAddr：是指定接口的IP地址。
- dwIndex：与IP地址关联在一起的接口之索引。

- dwMask：是IP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。
- unused1 and unused2：目前尚未使用。

利用这个函数，便可验证针对指定路由的本地 IP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，唯一的参数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。

B.3.3 删除路由

路由程序的最后一个行动是删除路由，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结构投递给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行指定pRo，ute字段。删除路由所需的字段有：dwForward-IfIndex、dwForwardDest、dwForwardMask、dwForwardNextHop以及dwForwardPolicy。

B.4 ARP

Arp.exe程序用于查看和处理 ARP缓存。通过IP助手函数仿真 Arp.exe程序的Platform SDK范例名为Iprap.exe。ARP（也许大家还记得，它代表名字解析协议）负责把一个 IP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 2000以及Windows NT4 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
);
```

第一个参数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则表明被当作pIpNetTable投递的缓冲区的长度。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，后者的格式如下：

```
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。它的格式如下：

```
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B-6展示了这个字段可能的值

表B-6 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

B.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
);
```

唯一的参数是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ARP信息来填充这个结构。首先，把 dwIndex参数设为一个本地IP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。下一个字段 dwPhysAddrLen，一般设为6（多数物理地址，比如说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，比如00-A0-C9-86-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。比如， MAC地址范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 6章）。dwAddr字段必须设为远程主机的IP地址，此时正在该机器上输入它的 MAC地址。最后一个字段dwType，设为表B-6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。

B.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条目之IP地址 —— dwADDR。用于删除 ARP条目的函数是DeleteIpNetEntry，它的定义如下：

```
DWORD DeleteIpNetEntry (
    PMIB_IPNETROW pArpEntry
);
```

其唯一的参数是一个MIB_IPNETROW结构，删除ARP条目时所需的唯一的信息是本地 IP索引和准备删除的IP地址。记住，本地IP接口的索引编号可通过GetIpAddrTable函数获得。如果成功，便返回NO_ERROR。

附录C　Winsock错误代码

本附录按错误编号列出了所有Winsock错误代码。但要注意的是，该列表没有包括标记为“BSD特有”的Winsock错误，也没有包括那些尚未正式列入规范的错误。此外，与Win32错误有着直接对应关系的Winsock错误列在本附录末尾。

10004——WSAEINTR

函数调用中断。该错误表明由于对WSACancelBlockingCall的调用，造成了一次调用被强行中断。

10009——WSAEBADF

文件句柄错误。该错误表明提供的文件句柄无效。在Microsoft Windows CE下，socket函数可能返回这个错误，表明共享串口处于“忙”状态。

10013——WSAEACCES

权限被拒。尝试对套接字进行操作，但被禁止。若试图在sendto或WSASendTo中使用一个广播地址，但是尚未用setsockopt和SO_BROADCAST这两个选项设置广播权限，便会产生这类错误。

10014——WSAEFAULT

地址无效。传给Winsock函数的指针地址无效。若指定的缓冲区太小，也会产生这个错误。

10022——WSAEINVAL

参数无效。指定了一个无效参数。例如，假如为WSAIoctl调用指定了一个无效控制代码，便会产生这个错误。另外，它也可能表明套接字当前的状态有错，例如在一个目前没有监听的套接字上调用accept或WSAAccept。

10024——WSAEMFILE

打开文件过多。提示打开的套接字太多了。通常，Microsoft提供者只受到系统内可用资源数量的限制。

10035——WSAEWOULDBLOCK

资源暂时不可用。对非锁定套接字来说，如果请求操作不能立即执行的话，通常会返回这个错误。比如说，在一个非暂停套接字上调用connect，就会返回这个错误。因为连接请求不能立即执行。

10036——WSAEINPROGRESS

操作正在进行中。当前正在执行非锁定操作。一般来说不会出现这个错误，除非正在开发16位Winsock应用程序。

10037——WSAEALREADY

操作已完成。一般来说，在非锁定套接字上尝试已处于进程中的操作时，会产生这个错误。比如，在一个已处于连接进程的非锁定套接字上，再一次调用connect或WSAConnect。另外，服务提供者处于执行回调函数（针对支持回调例程的Winsock函数）的进程中时，也会

出现这个错误。

10038——WSAENOTSOCK

无效套接字上的套接字操作。任何一个把 SOCKET句柄当作参数的Winsock函数都会返回这个错误。它表明提供的套接字句柄无效。

10039——WSAEDESTADDRREQ

需要目标地址。这个错误表明没有提供具体地址。比方说，假如在调用 sendto时，将目标地址设为INADDR_ANY（任意地址），便会返回这个错误。

10040——WSAEMSGSIZE

消息过长。这个错误的含义很多。如果在一个数据报套接字上发送一条消息，这条消息对内部缓冲区而言太大的话，就会产生这个错误。再比如，由于网络本身的限制，使一条消息过长，也会产生这个错误。最后，如果收到数据报之后，缓冲区太小，不能接收消息时，也会产生这个错误。

10041——WSAEPROTOTYPE

套接字协议类型有误。在socket或WSASocket调用中指定的协议不支持指定的套接字类型。比如，要求建立SOCK_STREAM类型的一个IP套接字，同时指定协议为 IPPROTO_UDP，便会产生这样的错误。

10042——WSAENOPROTOOPT

协议选项错误。表明在getsockopt或setsockopt调用中，指定的套接字选项或级别不明、未获支持或者无效。

10043——WSAEPROTONOSUPPORT

不支持的协议。系统中没有安装请求的协议或没有相应的实施方案。比如，如果系统中没有安装TCP/IP，而试着建立TCP或UDP套接字时，就会产生这个错误。

10044——WSAESOCKTNOSUPPORT

不支持的套接字类型。对指定的地址家族来说，没有相应的具体套接字类型支持。比如，在向一个不支持原始套接字的协议请求建立一个 SOCK_RAW套接字类型时，就会产生这个错误。

10045——WSAEOPNOTSUPP

不支持的操作。表明针对指定的对象，试图采取的操作未获支持。通常，如果试着在一个不支持调用Winsock函数的套接字上调用了 Winsock时，就会产生这个错误。比如，在一个数据报套接字上调用accept或WSAAccept函数时，就会产生这样的错误。

10046——WSAEPFNOSUPPORT

不支持的协议家族。请求的协议家族不存在，或系统内尚未安装。多数情况下，这个错误可与WSAEAFNOSUPPORT互换（两者等价）；后者出现得更为频繁。

10047——WSAEAFNOSUPPORT

地址家族不支持请求的操作。对套接字类型不支持的操作来说，在试着执行它时，就会出现这个错误。比如，在类型为 SOCK_STREAM的一个套接字上调用 sendto或WSASendTo函数时，就会产生这个错误。另外，在调用 socket或WSASocket函数的时候，若同时请求了一个无效的地址家族、套接字类型及协议组合，也会产生这个错误。

10048——WSAEADDRINUSE

地址正在使用。正常情况下，每个套接字只允许使用一个套接字地址（例如，一个 IP套接字地址由本地IP地址及端口号组成）。这个错误一般和bind、connect和WSAConnect这三个函数有关。可在setsockopt函数中设置套接字选项SO_REUSEADDR，允许多个套接字访问同一个本地IP地址及端口号（详情见第9章）。

10049 —— WSAEADDRNOTAVAIL

不能分配请求的地址。API调用中指定的地址对那个函数来说无效时，就会产生这样的错误。例如，若在bind调用中指定一个IP地址，但却没有对应的本地IP接口，便会产生这样的错误。另外，通过connect、WSAConnect、sendto、WSASendTo和WSAJoinLeaf这四个函数为准备连接的远程计算机指定端口0时，也会产生这样的错误。

10050 —— WSAENETDOWN

网络断开。试图采取一项操作时，却发现网络连接中断。这可能是由于网络堆栈的错误，网络接口的故障，或者本地网络的问题造成的。

10051 —— WSAENETUNREACH

网络不可抵达。试图采取一项操作时，却发现目标网络不可抵达（不可访问）。这意味着本地主机不知道如何抵达一个远程主机。换言之，目前没有已知的路由可抵达那个目标主机。

10052 —— WSAENETRESET

网络重设时断开了连接。由于“保持活动”操作检测到一个错误，造成网络连接的中断。若在一个已经无效的连接之上，通过setsockopt函数设置SO_KEEPALIVE选项，也会出现这样的错误。

10053 —— WSAECONNABORTED

软件造成连接取消。由于软件错误，造成一个已经建立的连接被取消。典型情况下，这意味着连接是由于协议或超时错误而被取消的。

10054 —— WSAECONNRESET

连接被对方重设。一个已经建立的连接被远程主机强行关闭。若远程主机上的进程异常中止运行（由于内存冲突或硬件故障），或者针对套接字执行了一次强行关闭，便会产生这样的错误。针对强行关闭的情况，可用 SO_LINGER套接字选项和setsockopt来配置一个套接字（欲知详情，请参阅第9章）。

10055 —— WSAENOBUFS

没有缓冲区空间。由于系统缺少足够的缓冲区空间，请求的操作不能执行。

10056 —— WSAEISCONN

套接字已经连接。表明在一个已建立连接的套接字上，试图再建立一个连接。要注意的是，数据报和数据流套接字均有可能出现这样的错误。使用数据报套接字时，假如事先已通过connect或WSAConnect调用，为数据报通信关联了一个端点的地址，那么以后试图再次调用sendto或WSASendTo，便会产生这样的错误。

10057 —— WSAENOTCONN

套接字尚未连接。若在一个尚未建立连接的“面向连接”套接字上发出数据收发请求，便会产生这样的错误。

10058 —— WSAESHUTDOWN

套接字关闭后不能发送。表明已通过对 shutdown的一次调用，部分关闭了套接字，但事

后又请求进行数据的收发操作。要注意的是，这种错误只会在已经关闭的那个数据流动方向上才会发生。举个例子来说，完成数据发送后，若调用 shutdown，那么以后任何数据发送调用都会产生这样的错误。

10060——WSAETIMEDOUT

连接超时。若发出了一个连接请求，但经过规定的时间，远程计算机仍未作出正确的响应（或根本没有任何响应），便会发生这样的错误。要想收到这样的错误，通常需要先在套接字上设置好SO_SNDTIMEO和SO_RCVTIMEO选项，然后调用connect及WSAConnect函数。要想了解在套接字上设置SO_SNDTIMEO和SO_RCVTIMEO选项的详情，可参考第9章。

10061——WSAECONNREFUSED

连接被拒。由于被目标机器拒绝，连接无法建立。这通常是由于在远程机器上，没有任何应用程序可在那个地址之上，为连接提供服务。

10064——WSAEHOSTDOWN

主机关闭。这个错误指出由于目标主机关闭，造成操作失败。然而，应用程序此时更有可能收到的是一条WSAETIMEDOUT（连接超时）错误，因为对方关机的情况通常是在试图建立一个连接的时候发生的。

10065——WSAEHOSTUNREACH

没有到主机的路由。应用程序试图访问一个不可抵达的主机。该错误类似于WSAENETUNREACH。

10067——WSAEPROCLIM

进程过多。有些Winsock服务提供者对能够同时访问它们的进程数量进行了限制。

10091——WSASYSNOTREADY

网络子系统不可用。调用WSAStartup时，若提供者不能正常工作（由于提供服务的基层系统不可用），便会返回这种错误。

10092——WSAVERNOTSUPPORTED

Winsock.dll版本有误。表明不支持请求的Winsock提供者版本。

10093——WSANOTINITIALISED

Winsock尚未初始化。尚未成功完成对WSAStartup的一次调用。

10101——WSAEDISCON

正在从容关闭。这个错误是由WSARecv和WSARecvFrom返回的，指出远程主机已初始化了一次从容关闭操作。该错误是在像ATM这样的“面向消息”协议上发生的。

10102——WSAENOMORE

找不到更多的记录。这个错误自WSALookupServiceNext函数返回，指出已经没有留下更多的记录。这个错误通常可与WSA_E_NO_MORE互换使用。在应用程序中，应同时检查这个错误以及WSA_E_NO_MORE。

10103——WSAECANCELLED

操作被取消。这个错误指出当WSALookupServiceNext调用仍在处理期间，发出了对WSALookupServiceEnd（服务中止）的一个调用。此时，WSALookupServiceNext便会返回这个错误。这个错误代码可与WSA_E_CANCELLED互换使用。作为应用程序，应同时检查这个错误以及WSA_E_CANCELLED。

10104——WSAEINVALIDPROCTABLE

进程调用表无效。该错误通常是在进程表包含了无效条目的情况下，由一个服务提供者返回的。欲知服务提供者的详情，可参考第14章。

10105——WSAEINVALIDPROVIDER

无效的服务提供者。这个错误同服务提供者关联在一起，在提供者不能建立正确的Winsock版本，从而无法正常工作的前提下产生。

10106——WSAEPROVIDERFAILEDINIT

提供者初始化失败。这个错误同服务提供者关联在一起，通常见于提供者不能载入需要的DLL时。

10107——WSASYSCALLFAILURE

系统调用失败。表明绝对不应失败的一个系统调用却令人遗憾地失败了。

10108——WSASERVICE_NOT_FOUND

找不到这样的服务。这个错误通常与注册和名字解析函数相关，在查询服务时产生（第10章对这些函数进行了详尽解释）。该错误表明，在给定的名字空间内，找不到请求的服务。

10109——WSATYPE_NOT_FOUND

找不到类的类型。该错误也与注册及名字解析函数关联在一起，在处理服务类（ Service Class）时发生。若注册好一个服务的实例，它必须引用一个以前通过 WSAInstallServiceClass 安装好的服务。

10110——WSA_E_NO_MORE

找不到更多的记录。这个错误是自 WSALookupServiceNext调用返回的，指出已经没有剩下的记录。该错误通常可与 WSAENOMORE互换使用。作为一个应用程序，应同时检查这个错误以及WSAENOMORE。

10111——WSA_E_CANCELLED

操作被取消。该错误指出在对 WSALookupServiceNext的调用尚未完成的时候，又发出了对WSALookupServiceEnd（中止服务）的一个调用。这样， WSALookupServiceNext就会返回该错误。这个错误代码可与 WSAECANCELLED互换使用。作为一个应用程序，应同时检查这个错误以及WSAECANCELLED。

10112——WSAEREFUSED

查询被拒。由于被主动拒绝，所以一个数据库查询操作失败。

11001——WSAHOST_NOT_FOUND

主机没有找到。这个错误是在调用 gethostbyname和gethostbyaddr时产生的，表明没有找到一个授权应答主机（Authoritative Answer Host）。

11002——WSATRY_AGAIN

非授权主机没有找到。这个错误也是在调用 gethostbyname和gethostbyaddr时产生的，表明没有找到一个非授权主机，或者遇到了服务器故障。

11003——WSANO_RECOVERY

遇到一个不可恢复的错误。这个错误也是在调用 gethostbyname和gethostbyaddr时产生的，指出遇到一个不可恢复的错误，应再次尝试操作。

11004——WSANO_DATA

没有找到请求类型的数据记录。这个错误也是在调用 gethostbyname和gethostbyaddr时产生的，指出尽管提供的名字有效，但却没有找到与请求类型对应的数据记录。

11005 —— WSA_QOS_RECEIVERS

至少有一条预约消息抵达。这个值同 IP服务质量（QoS）有着密切的关系，其实并不是一个真正的“ 错误 ”（QoS的详情见第12章）。它指出网络上至少有一个进程希望接收 QoS通信。

11006 —— WSA_QOS_SENDERS

至少有一条路径消息抵达。这个值同 QoS关联在一起，其实更像一种状态报告消息。它指出在网络上，至少有一个进程希望进行 QoS数据的发送。

11007 —— WSA_QOS_NO_SENDERS

没有QoS发送者。这个值同QoS关联在一起，指出不再有任何进程对 QoS数据的发送有兴趣。请参阅第12章，了解在发生这样的错误时，对所发生情况的一系列完整说明。

11008 —— WSA_QOS_NO_RECEIVERS

没有QoS接收者。这个值同QoS关联在一起，指出不再有任何进程对 QoS数据的接收有兴趣。请参阅第12章，查阅对这个错误的完整说明。

11009 —— WSA_QOS_REQUEST_CONFIRMED

预约请求已被确认。QoS应用可事先发出请求，希望在批准了自己对网络带宽的预约请求后，收到通知。若发出了这样的请求，一旦批准，便会收到这样的消息。请参阅第 12章，了解对此消息的详细说明。

11010 —— WSA_QOS_ADMISSION_FAILURE

缺乏资源致错。资源不够，以至于无法满足 QoS带宽请求。

11011 —— WSA_QOS_POLICY_FAILURE

证书无效。表明发出QoS预约请求的时候，要么用户并不具备正确的权限，要么提供的证书无效。

11012 —— WSA_QOS_BAD_STYLE

未知或冲突的样式。QoS应用程序可针对一个指定的会话，建立不同的过滤器样式。若出现这一错误，表明指定的样式类型要么未知，要么存在冲突。请参阅第 12章，了解对过滤器样式的详细说明。

11013 —— WSA_QOS_BAD_OBJECT

无效的FILTERSPEC结构或者提供者特有对象。假如为 QoS对象提供的FILTERSPEC结构无效，或者提供者特有的缓冲区无效，便会返回这样的错误，详见第 12章。

11014 —— WSA_QOS_TRAFFIC_CTRL_ERROR

FLOWSPEC有问题。假如通信控制组件发现指定的 FLOWSPEC参数存在问题（作为QoS对象的一个成员传递），便会返回这样的错误。

11015 —— WSA_QOS_GENERIC_ERROR

常规QoS错误。这是一个比较泛泛的错误；假如其他QoS错误都不适合，便返回这个错误。

6 —— WSA_INVALID_HANDLE

指定的事件对象无效。若使用与 Win32函数对应的Winsock函数，便有可能产生这样的Win32错误。它表明传递给WSAWaitForMultipleEvents的一个句柄是无效的。

8 —— WSA_NOT_ENOUGH_MEMORY

内存不够。这个Win32错误指出内存数量不足，无法完成指定的操作。

87 —— WSA_INVALID_PARAMETER

一个或多个参数无效。这个Win32错误表明传递到函数内部的参数无效。假若事件计数参数无效，那么在执行WSAWaitForMultipleEvents的时候，也会发生这样的错误。

258 —— WSA_WAIT_TIMEOUT

操作超时。这个Win32错误指出重叠I/O操作未在规定的时间内完成。

995 —— WSA_OPERATION_ABORTED

重叠操作被取消。这个Win32错误指出由于套接字的关闭，造成一次重叠 I/O操作的取消。除此以外，该错误也可能在执行SIO_FLUSH这个I/O控制命令时出现。

996 —— WSA_IO_INCOMPLETE

重叠I/O事件对象未处于传信状态。这个 Win32错误也和重叠I/O操作密切相关，在调用WSAGetOverlappedResults函数的时候产生，指出重叠I/O操作尚未完成。

997 —— WSA_IO_PENDING

重叠操作将在以后完成。用Winsock函数发出一次重叠I/O操作时，若出现这样的Win32错误，便表明操作尚未完成，而且会在以后的某个时间完成。有关重叠 I/O的深入讨论，可参阅第8章。