# 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
```

```
);
```

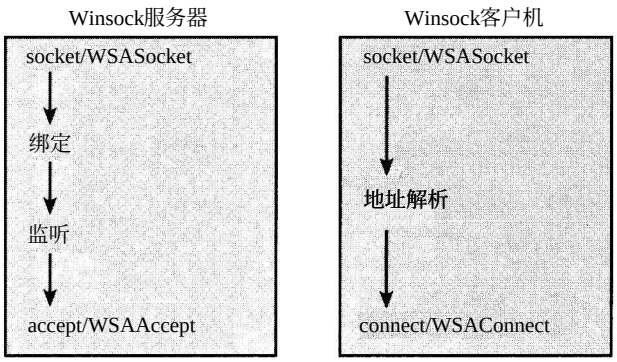


图7-1 服务器和客户机的Winsock基础

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。

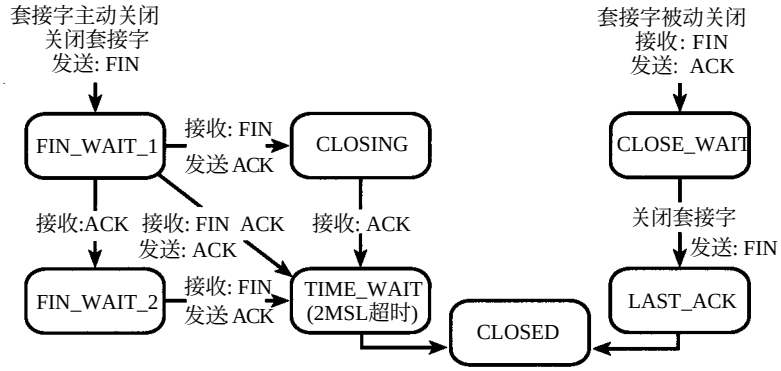


图7-2 TCP套接字的关闭状态

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。