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。

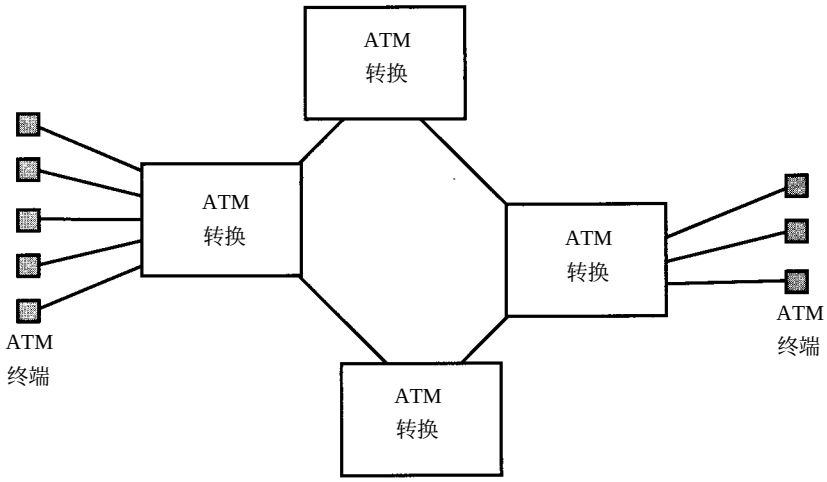

图6-1　ATM网络

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。