第14章 Winsock 2 服务提供者接口

Winsock 2服务提供者接口（Service Provider Interface, SPI）代表着另一端的Winsock编程（和Winsock 2API相对应）。Winsock的一端是API，另一端则是SPI。自第6章到第13章，我们已对 Winsock 2 API进行了详细讨论。 Winsock 2是围绕着 Windows开放系统架构（Windows Open System Architecture，WOSA）来设计的，WOSA在Winsock和Winsock应用程序之间有一个标准 API；在Winsock和Winsock服务提供者（比如 TCP/IP）之间有一个标准的SPI。图14-1展示了Ws2_32.dll，即Winsock 2支持的动态链接库（DLL）在Winsock应用程序和Winsock服务提供者之间的分布情况。本章详细地讲解了 Winsock 2 SPI。在结束本章的学习时，大家自然便理解如何开发服务提供者，进一步扩展 Winsock 2的能力。

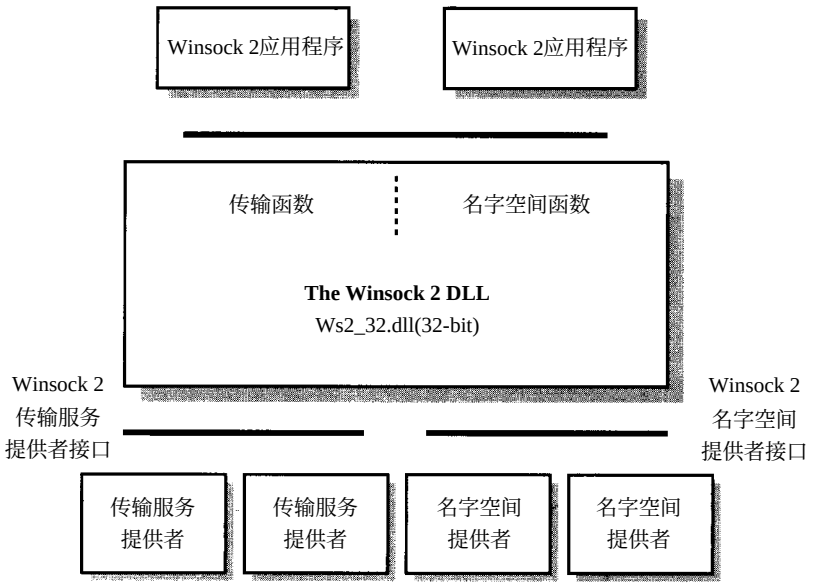

图14-1 Winsock 2的WOSA架构

14.1 SPI基础

Winsock 2 SPI允许开发两类服务提供者 —— 传输提供者和名字空间提供者。“传输提供者”（Transport providers，一般称作协议堆栈，比如 TCP/IP）即能够提供建立通信、传输数据、日常数据流控制和错误控制等等功能的服务。“名字空间提供者”（Name space providers）则把一个网络协议的定址属性和一个或多个用户友好名关联到一起，以便启用与协议无关的名字解析方案。服务提供者不外乎就是 Win32支持的DLL，挂靠在Winsock 2的Ws2_32.dll模块下。对Winsock2 API中定义的许多内部调用来说，这些服务提供者都提供了它们的运作方式。

14.1.1 SPI命名规则

Winsock 2 SPI函数原型采用下面的函数前缀命名规则：

- WSP（Winsock提供者）：用于传送服务提供者函数。
- NSP（名字空间提供者）：用于名字空间提供者函数。
- WPU（Winsock提供者上调）：供服务提供者调用的Ws2_32.dll支持函数使用。
- WSC（Winsock配置）：供在Winsock 2中安装服务提供者的函数使用。

举个例子来说，一个名为WSAInstallProvider的函数，仅仅是一个SPI配置函数。

14.1.2 Winsock 2 API和SPI函数之间的映射

多数情况下，一个应用程序在调用 Winsock 2函数时，Ws2_32.dll会调用相应的Winsock SPI函数，利用特定的服务提供者执行所请求的服务。举个例子来说， select对应WSPSelect，WSAConnect对应WSPConnect，而WSAAccept则对应WSPAccept。然而，并非所有的Winsock函数都有对应的SPI函数。下面列出了这些例外情况。

- htonl、htons、ntohl和ntohs这一类的支持函数在Ws2_32.dll内部执行，没向下传给服务提供者。这一点对这些函数的WSA版本来说，也不例外。
- 像inet_addr和inet_ntoa这样的IP转换函数只能在Ws2_32.dll内执行。
- Winsock 1.1中，所有由IP决定的名字转换和解析函数（比如getXbyY、WSAAsynGetXByY、WSACancelAsyncRequest以及gethostname）都在Ws2_32.dll内部执行。
- Winsock服务提供者列举和与锁定挂钩相关的函数都在 Ws2_32.dll内部执行。因此，WSAEnumProtocols、WSAIsBlocking、WSASetBlockingHook和WSAUnhookBlocking没有相应的SPI函数。
- Winsock错误代码的管理在Ws2_32.dll内部进行。而SPI中，不需要WSAGetLastEror和WSASetLastError。
- 事件对象处理和等待函数，其中包括WSACreateEvent、WSACloseEvent、WSASetEvent、WSAResetEvent和WSAWaitForMultipleEvents，这些函数直接映射成原始的 Win32操作系统调用，没有出现在SPI中。

现在，大家准备学习哪些 Winsock API映射成Winsock 2服务提供者。在开发服务提供者时，大家将看到定义在包容文件 Ws2spi.h中的所有SPI函数原型。

14.2 传输服务提供者

Winsock 2中使用的传输服务提供者有两类：基础服务提供者和分层服务提供者。基础服务提供者执行网络传输协议（比如 TCP/IP）的具体细节其中包括在网络上收发数据之类的核心网络协议功能。“ 分层式 ”（Layered）服务提供者只负责执行高级的自定义通信功能，并依靠下面基础服务提供者，在网络上进行真正的数据交换。举个例子来说，你可以在现成的基础TCP/IP提供者上执行一个数据安全管理者或带宽管理者。图 14-2展示了如何在一个Ws2_32.dll和一个基础提供者之间安装一个或多个分层式提供者。

本小节的重点在于如何开发分层式传输服务提供者。如果你此时正在开发一个基础服务提供者，就可以运用这里的规则。我们具体讲解如何从头执行特定 SPI函数。举个例子来说，我们不提供WSPSend SPI函数是如何把数据写入网络适配器这类的细节。相反，我们展示的

则是分层式提供者的WSPSend函数将如何调用低级提供者的 WSPSend函数，这是多数分层式服务提供者的一项基本要求。从本质上说，开发一个分层式服务提供者所涉及的许多操作都依赖于你的提供者对你的下一层提供者进行的 SPI调用来完成。容易出错的环节是通过Winsock I/O模型，怎样对I/O调用（参见第8章）进行处理，至于这一点，我们稍后将继续讨论。本书的配套光盘上，我们提供了一个示例，名为 LSP，向大家演示了如何执行分层式协议提供者，该提供者只计下了一个套接字上利用 IP传输协议传输了多少字节。微软平台 SDK特别提供了一个更高级的分层式协议提供者示例，名为 layered（分层式），可在MSDN平台SDK示例中找到它。至于 MSDN平台SDK示例则在 ftp://ftp.microsoft.com/bussys/WinSock/winsock2/layered.zip下载。

注意 上面我们常提到这些术语“ SPI客户机 ”和“ 低级别提供者 ”。SPI客户机既可以是Winsock 2的Ws2_32.dll，又可以是位于你的服务提供者之上的另一个分层式服务提供者。SPI客户机绝不可能是Winsock应用程序本身，因为Winsock应用程序必须使用Ws2_32.dll中导出的Winsock 2 API。“ 低级别提供者 ”这个术语只能在描述分层式服务提供者的开始时使用。低级别提供者既可以是另查个分层式服务提供者，又可以是基础服务器提供者。正如大家即将看到的那样，一台机器上可以安装若干个分层式服务提供者；因此，也可以在你的提供者下面，再安装一个分层式服务提供者。

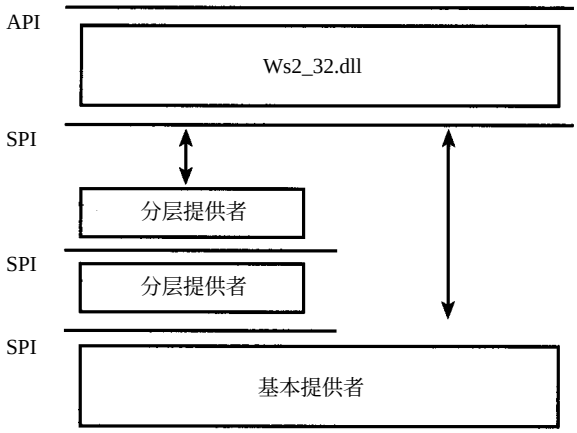

图14-2 分层提供者的架构

14.2.1 WSPStartup

Winsock 2传输服务提供者随标准的Windows动态链接库模块一起执行，你必须把 DllMain函数导入这个动态链接库模块中。除此以外，还必须导入一个名为 WSPStartup的单一函数条目。在调用者（如SPI客户机）调用WSPStartup时，便通过一个被当作参数投送的函数派遣表打开另外的30个SPI函数，传输服务提供者便由这 30个函数组成。函数派遣表参见表 14-1。你的服务提供者必须提供对WSPStartup函数和其他30个函数的支持。

何时以及怎样调用 WSPStarup呢？这个问题尤其重要。也许应用程序调用 WSAStartup API时，你也在考虑调用WSPStartup。这里的情况有所不同。调用 WSAStartup期间，Winsock不知道需要使用哪种服务提供者的类型。 Winsock根据WSASocket调用的地址家族、套接字类

型和协议参数，决定需要加载哪个服务提供者。只有在一个应用程序通过 socket或WSASocket API调用建立一个套接字时， Winsock才会调用一个服务提供者。举个例子来说，如果一个应用程序建立了一个采用地址家族 AF_INET、套接字类型SOCK_STREAM的套接字， Winsock就搜索并加载与之相应的、能够提供 TCP/IP能力的传输提供者。我们将在本章的 14.2.7节“传输服务提供者的安装”就此进行深入讨论。

表14-1 传输服务提供者的支持函数

API函数	对应的SPI函数
WSAAccept（accept也可间接映射成WSPAccept）	WSPAccept
WSAAddressToString	WSPAddressToString
WSAAsyncSelect	WSPAsyncSelect
bind	WSPBind
WSACancelBlockingCall	WSPCancelBlockingCall
WSACleanup	WSPCleanup
closesocket	WSPCloseSocket
WSAConnect（connect也可间接映射成WSPConnect）	WSPConnect
WSADuplicateSocket	WSPDuplicateSocket
WSAEnumNetworkEvents	WSPEnumNetworkEvents
WSAEventSelect	WSPEventSelect
WSAGetOverlappedResult	WSPGetOverlappedResult
getpeername	WSPGetPeerName
getsockname	WSPGetSockName
getsockopt	WSPGetSockOpt
WSAGetQOSByName	WSPGetQOSByName
WSAIoctl	WSPIoctl
WSAJoinLeaf	WSPJoinLeaf
listen	WSPListen
WSARecv（recv也可间接映射成WSPRecv）	WSPRecv
WSARecvDisconnect	WSPRecvDisconnect
WSARecvFrom（recvfrom也可间接映射成WSPRecvFrom）	WSPRecvFrom
select	WSPSelect
WSASend（send也可间接映射成WSPSend）	WSPSend
WSASendDisconnect	WSPSendDisconnect
WSASendTo（sendto也可间接映射成WSPSendto）	WSPSendTo
setsockopt	WSPSetSockOpt
sbutdown	WSPSbutdown
WSASocket（socket也可间接映射成WSPSocket）	WSPSocket
WSAStringToAddress	WSPStringToAddress

14.2.2 参数

在初始化传输服务提供者时， 关键性的函数便是 WSAStartup。它的定义如下：

```
int WSPStartup(
    WORD wVersionRequested,
    LPWSPDATAW lpWSPData,
    LPWSAPROTOCOL_INFOW lpProtocolInfo,
    WSPUPCALLTABLE UpcallTable,
    LPWSPPROC_TABLE lpProcTable
);
```

wVersionRequested参数用于取得调用者能够使用的 Windows Sockets SPI支持函数的最新版本。你的服务提供者应该对这个值进行检查，以便得知它是否支持所请求的版本。你的提供者通过一个WSPDATA结构，利用lpWSPData参数返回关于它自己的版本信息。 WSPDATA结构的格式如下：

```
typedef struct WSPData
{
    WORD        wVersion;
    WORD        wHighVersion;
    WCHAR       szDescription[WSPDESCRIPTION_LEN + 1];
} WSPDATA, FAR * LPWSPDATA;
```

wVersion参数中，协议提供者必须返回调用者希望采用的 Winsock版本号。wHighVersion参数必须返回提供者支持的 Winsock的最高版本。这个值一般和 wVersion的值相同（关于Winsock版本信息，详见第 7章）。szDescription字段返回一个空中止 UNICODE字串，该字串把你的提供者视作SPI客户机。这个字段最多可容纳256个字符。

WSPStartup的lpProtocolInfo参数是一个指针，指向一个 WSAPROTOCOL_INFOW结构。该结构中包含了和提供者有关的特征信息（协议特征和这个结构的详细情况，参见第 5章“ Winsock协议信息 ”）。WSAPROTOCOL_INFOW结构中的信息是Ws2.32_dll从Winsock 2服务提供者目录中检索得来的。这个目录中包含了服务提供者的属性信息。我们将在“ 传输服务提供者的安装 ”这一小节，对 Winsock 2目录条目做进一步的探讨。

在开发分层式服务提供者时，需要以一种独特的方式来处理 lpProtocolInfo这个参数，因为它包含的信息关系到你的服务提供者在 Ws2_32.dll和基础服务提供者之间，是怎样分布的。这个参数用于决定你的服务提供者之下的那个服务提供者（既可以是另一个分层提供者，又可以是一个基础提供者）。从某个角度来说，你的提供者必须通过加载下一个提供者的 DLL模块和调用它的 WSPStartup函数这一方式，来加载下一个提供者。 lpProtocolInfo指向的WSAPROTOCOL_INFOW结构中包含一个字段ProtocolChain，该字段标志你的服务提供者和机器上的其他提供者是如何排序的。

ProtocolChain字段实际上是一个WSAPROTOCOLCHAIN结构，它的格式如下：

```
typedef struct _WSAPROTOCOLCHAIN
{
    int ChainLen;
    DWORD ChainEntries[MAX_PROTOCOL_CHAIN];
} WSAPROTOCOLCHAIN, FAR * LPWSAPROTOCOLCHAIN;
```

ChainLen字段标志Ws2_32.dll和基础的服务提供者之间，重叠了多少层（这个数包含基础提供者在内）。如果你有一台这样的机器，在一个协议之上只有一个重叠服务提供者，比如说TCP/IP，那么这个字段就应该是 2。ChainEntries字段是一数组，该数组由服务提供者目录标志号组成，这些编号唯一性地标志出重叠服务提供者，这些提供者因为某一特定协议而链接到一起。我们将在 14.2.7节“ 传输服务提供者的安装 ”中详细说明 WSAPROTOCOLCHAIN结构。分层式服务提供者的一项要求是搜索 ProtocolChain字段，以便决定它自己在服务提供者数组中所处的位置（通过搜索层目录条目这一方式），以及确定数组中的下一个提供者。如果下一个提供者是另一层，就必须把未经修改的 lpProtocolInfo结构投给下一层的WSPStartup函数。如果下一层提供者是本数组中的最后一位（即基础提供者），你的提供者就必须在调用基础提供者的WSPStartup函数之前，利用这个基础提供者的 WSAPROTOCOL_INFOW结构，在

lpProtocolInfo结构上执行交换。程序清单 14-1演示了分层提供者应该怎样通过编程来管理lpProtocolInfo结构。

程序清单14-1 为WSPStartup寻找相应的WSAPROTOCOL_INFOW结构

```
LPWSAPROTOCOL_INFOW ProtocolInfo;
LPWSAPROTOCOL_INFOW ProtoInfo = lpProtocolInfo;
DWORD ProtocolInfoSize = 0;
// Find out how many entries we need to enumerate
if (WSCEnumProtocols(NULL, ProtocolInfo, &ProtocolInfoSize,
    &ErrorCode) == SOCKET_ERROR)
{
    if (ErrorCode != WSAENOBUFS)
    {
        return  WSAEPROVIDERFAILEDINIT;
    }
}

if ((ProtocolInfo = (LPWSAPROTOCOL_INFOW) GlobalAlloc(GPTR,
    ProtocolInfoSize)) == NULL)
{
    return  WSAEPROVIDERFAILEDINIT;
}

if ((TotalProtocols = WSCEnumProtocols(NULL, ProtocolInfo,
    &ProtocolInfoSize, &ErrorCode)) == SOCKET_ERROR)
{
    return  WSAEPROVIDERFAILEDINIT;
}

// Find our layered provider's catalog ID entry
for (i = 0; i < TotalProtocols; i++)
    if (memcmp (&ProtocolInfo[i].ProviderId, &ProviderGuid,
        sizeof (GUID))==0)
    {
        gLayerCatId = ProtocolInfo[i].dwCatalogEntryId;
        break;
    }

// Save our provider's catalog ID entry
gChainId = lpProtocolInfo->dwCatalogEntryId;

// Find our catalog ID entry in the protocol chain
for(j = 0; j < lpProtocolInfo->ProtocolChain.ChainLen; j++)
{
    if (lpProtocolInfo->ProtocolChain.ChainEntries[j] ==
        gLayerCatId)
    {
        NextProviderCatId =
        lpProtocolInfo->ProtocolChain.ChainEntries[j+1];

        // Check whether next provider is the base provider
        if (lpProtocolInfo->ProtocolChain.ChainLen ==
            (j + 2))
        {
            for (i = 0; i < TotalProtocols; i++)
                if (NextProviderCatId ==
```

```
                    ProtocolInfo[i].dwCatalogEntryId)
                {
                    ProtoInfo = &ProtocolInfo[i];
                    break;
                }
            }
            break;
        }
    }

    // At this point, ProtoInfo will contain the appropriate
    // WSAPROTOCOL_INFOW structure
```

WSPStartup的UpcallTable取得Ws2_32.dll的SPI上调派遣表，表中包含了指向许多支持函数的指针，你的提供者可利用这些函数来管理提供者自身和 Winsock 2之间的I/O操作。本章的“ Winsock I/O模型支持 ”小节中，我们还定义许多此类的函数，并对它们的用法进行了描述。

WSPStartup的最后一个参数是lpProcTable，代表一个表，表内有30个SPI函数指针，你的服务提供者必须能够提供对这些函数的支持。这 30个函数列在表14-1中。每个SPI函数都遵照自己对应的API函数的参数说明。

- 每个函数都提供一个终极参数， lpErrno，在实施失败时，你的提供者必须用这个参数报告具体的Winsock错误代码。举个例子来说，如果你在实施 WSPSend，但不能为它分配内存，可能就会返回WSAENOBUF错误代码。
- SPI函数WSPSend、WSPSendTo、WSPRecv、WSPRecvFrom以及WSPIoctl，都突出了这个额外的参数：lpThreadId，它标志调用了SPI函数的应用程序线程。稍后，我们将看到，这个特性对支持完成例程时，发挥了很大的作用。

最后一个需要注意的是几个 Winsock 1.1函数，比如 send和recv，直接映射成相应的Winsock 2函数。我们之所以说这些 Winsock 1.1间接映射成相应的SPI函数，是因为它们事实上调用了提供类似功能的 Winsock 2函数。比如说， send函数实际上调用的是WSASend函数，WSASend函数的映射函数是WSPSend。表14-1中，我们特别注明了间接映射成 SPI函数的API函数。

14.2.3 实例计数

在Winsock规格中，应用程序调用 WSAStartup和WSACleanup函数的次数是没有限制的。你的服务提供者的WSPStartup和WSPCleanup函数也将和它们对应的 API函数一样，调用的次数相当。如此一来，你的服务提供者就应该保留一个实例计数，以便了解 WSPStartup函数被调用了多少次，再相应地调用 WSPCleanup多少次，以便抵消WSPStartup被调用的次数。保留实例次数的目的在于：允许你的服务提供者的启用和清除过程合乎情理。举个例子来说，只要你的实例计数大于 0，你的提供者就可以一直保存在内存中。当实例计数落到 0以下时，Ws2_32.dll最终从相应的内存中卸载你的提供者。

14.2.4 套接字句柄

当SPI客户机调用WSPSocket、WSPAccept和WSPJointLeaf函数时，服务提供者必须返回套接字句柄。返回 SPI客户机的套接字句柄既可以是可安装文件系统（ IFS）句柄，又可以是

非IFS句柄。如果一个服务提供者返回的是 IFS句柄，就会被视作 IFS提供者；反之，就是非IFS提供者。微软的基础传输提供者全都是 IFS句柄。

设计Winsock的目的是允许Winsock应用程序利用Win32 API函数ReadFile和WriteFile，在套接字句柄上收发数据。因此，必须考虑到如何在一个服务提供者内建立套接字句柄。如果在开发服务提供者的同时，还试图为套接字句柄上调用 ReadFile和WriteFile的Winsock应用程序提供服务，就要考虑另行开发一个 IFS提供者。但之前，必须先了解I/O的某些限制。

1. IFS 提供者

我们在前面曾提过，传输服务提供者可以是分层的或基础的。如果正在开发一个基础的IFS提供者，你的提供者就会有一个核心模式的操作系统组件，而这个组件启用了 Winsock提供者来建立句柄，与ReadFile和WriteFile函数中的文件系统句柄相同。核心模式软件的开发不在本书讨论之列。如果想了解如何开发核心模式操作系统组件这方面的详情，可参考 MSDN设备开发工具包（Device Development Kit，DDK）。

分层式服务提供者也可变成 IFS提供者，但前提是它必须位于现成的基础 IFS提供者顶部。这涉及到把低级IFS提供者的套接字句柄(在你的分层提供者中检索得来的)直接上传到你的SPI客户机。从一个低级提供者直接上传套接字句柄会限制了分层提供者的能力，主要表现在以下两方面：

- 如果在一个套接字上调用了 ReadFile和WriteFile，便不会调用分层提供者的 WSPSend和WSPRecv函数。这些函数将绕过分层提供者，直接调用基础 IFS提供者的实施。
- 分层提供者将不能后处理提交到一个完成端口的重叠 I/O请求。完成端口的后处理完成绕过了分层式提供者。

如果你的分层式提供者试图对通过这个提供者传递的所有 I/O操作进行监视， 还必须开发一个非IFS分层式提供者，我们稍后对此进行详细讨论。

只要一个IFS提供者（分层的或基础的）建立了一个新的套接字描述符，就需要要求它调用WPUModifyIFSHandle，而不是提供指向 SPI客户机的新句柄。 Win32API函数（比如ReadFile和WriteFile）在一个套接字上执行I/O时，这样便可使Winsock Ws2_32.dll合理地标识与指定套接字关联到一起的IFS服务提供者进程。 WPUModifyIFSHandle的定义如下：

```
SOCKET WPUModifyIFSHandle(
    DWORD dwCatalogEntryId,
    SOCKET ProposedHandle,
    LPINT lpErrno
);
```

dwCatalogEntryId参数标识你的服务提供者的目录ID。ProposedHandle参数代表一个IFS句柄，该句柄是你的服务提供者分配的（在基础提供者的情况下）。如果此时正在开发分层式IFS提供者，这个句柄就会从低级提供者上传。如果这个函数失败，返回错误 INVALID_SOCKET，lpErrno参数便取得特定的Winsock错误代码信息。

2. 非IFS提供者

如果你此时正在开发分层式提供者，并试图对一个套接字上发生的所有读取和写入操作进行监视，还必须开发一个 IFS提供者。非IFS提供者使用上调函数WPUCreateSocketHandle建立套接字句柄。 WPUCreateSocketHandle建立的套接字句柄类似于 IFS提供者句柄，两者均允许Winsock应用程序在套接字上使用 ReadFile和WriteFile函数。但是，这两个函数对 I/O性能有

一个非常明显的不利影响，因为 Winsock 2架构必须分别执行到服务提供者的 WSPRecv和WSPSend函数的重定向I/O操作。WPUCreateSocketHandle的定义如下：

```
SOCKET WPUCreateSocketHandle(
    DWORD dwCatalogEntryId,
    DWORD dwContext,
    LPINT lpErrno
);
```

dwCatalogEntryId标识你的服务提供者的目录 ID。dwContext参数允许你自由地把提供者的数据和一个套接字描述符关联到一起。在本书配套光盘上的 LSP示例中，我们利用了这个字段保存了收发数据的字节数。 Winsock提供了一个上调函数 WPUQuerySocketHandleContext，该函数用于取得保存在dwContext中关联的套接字提供者数据，它的定义如下：

```
int WPUQuerySocketHandleContext(
    SOCKET s,
    LPDWORD lpContext,
    LPINT lpErrno
);
```

s参数代表套接字句柄，该句柄从一个 SPI客户机（最初通过WPUCreateSocketHandle建立的）开始下传，并且你打算用这个套接字句柄取得套接字提供者的数据。 lpContext参数取得最初投入WPUCreateSocketHandle中的提供者数据。和前一个函数中的 lpErrno参数一样，若函数调用失败，就会收到相关的 Winsock错误代码。如果WPUCreateSocketHandle失败，便返回INVALID_SOCKET；若WPUQuerySocketHandleContext失败，则返回SOCKET_ERROR。

14.2.5 Winsock I/O模型支持

通过第8章的学习，大家已经知道Winsock的特别突出了几个I/O模型，这些模型可以用来管理套接字上的 I/O操作。从服务提供者角度来看，每个模型都要求使用某些 SPI上调函数，这些函数是Ws2_32.dll提供的，通过前面提到的 WSPStartup中的UpcallTable参数使用。如果此时正在开发一个简单的 IFS分层式提供者，下面几个小节中讲的 I/O模式遵守的原则就不适用，因为低级提供者便足以应付对各个模式的 I/O操作的管理了。这些原则只适用于其他类的提供者。我们将重点讨论如何开发非 IFS分层式服务提供者。

1. 锁定和非锁定模型

Winsock 2中，锁定I/O是最简单的。锁定套接字的任何一个 I/O操作都不会在操作出完成之前返回。因此，所有线程一次只能执行一个 I/O操作。举个例子来说，一个 SPI客户机以锁定方式调用 WSPRecv函数时，你的提供者只需要把这个调用直接传给下一个提供者的WSPRecv调用。对你的提供者而言，它的 WSPRecv函数只能在下一个提供者的 WSPRecv函数完成时，才能够返回。

尽管锁定I/O模型非常易于实现，但仍然需要考虑向后兼容Winsock 1.1锁定挂钩这一问题。Winsock 2 API规格中已经删除了WSASetBlockingCall和WSACancelBlocking API调用。但是，一个Winsock 1.1应用程序在调用 WSASetBlockingHook和WSACancelBlockingCall时，Ws2_32.dll仍然可以调用 WSPCancelBlockingHook。在分层式服务提供者中，只需要把WSPCancelBlockingHook调用传给基础提供者的相应调用即可。如果此时正在执行一个基础提供者和一个锁定调用，就必须执行周期性地调用 WPUQueryBlockingCallback函数这一机制。

WPUQueryBlockingCallback的定义如下：

```
int WPUQueryBlockingCallback(
    DWORD dwCatalogEntryId,
    LPBLOCKINGCALLBACK FAR *lplpfnCallback,
    LPDWORD lpdwContext,
    LPINT lpErrno
);
```

像我们在介绍WSPStartup函数时描述的那样，dwCatalogEntryId参数取得你的提供者的目录ID。lplpfnCallback参数是一个函数指针，指向应用程序的锁定挂钩函数，你必须周期性地调用这个函数（即回调函数），以避免该应用程序的锁定调用真正地被锁定。这个回调函数的形式如下：

```
typedef BOOL (CALLBACK FAR * LPBLOCKINGCALLBACK)(
    DWORD dwContext
);
```

你的提供者周期性调用 lplpfnCallback时，便把lpdwContext参数的值投给该回调函数的dwContext参数。WPUQueryBlockingCallback函数的最后一个参数是lpErrno。若该函数返回SOCKET_ERROR，这个参数便返回相应的Winsock错误代码信息。

2. select模型

select I/O模型要求提供者为 WSPSelect函数中的这几个参数（ readfds、writefds和exceptfds）管理fd_set结构。WSPSelect函数的定义如下：

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

从本质上来说，fd_set数据类型代表一个套接字集合。一个 SPI客户机利用WSPSelect调用你的提供者时，便把套接字句柄投给这个集合中的一个或多个套接字。你的提供者负责判断列出的套接字上何时发生了网络活动。

对非IFS分层提供者来说，便要求建立三个 fd_set数据字段，并将SPI客户机的套接字句柄映射到这个集合中的低层提供者的套接字句柄。一旦建立了所有的套接字集合，你的提供者便调用低层提供者上的 WSPSelect。低层提供者的调用完成时，你的提供者必须判断各个fd_set字段中，哪些套接字上有待发事件。此时，便可以用这个非常有用的上调函数WPUFDIsSet，来判断设置了哪些基础提供者套接字。这个上调函数类似于 FD_ISSET宏（参见第8章），它的定义如下：

```
int WPUFDIsSet(
    SOCKET s,
    FD_SET FAR *set
);
```

s参数代表你正在套接字集合中查找的套接字。 set参数是一个真正的套接字描述符集合。由于你的提供者将检查投向低层提供者的每个描述符集合中的内容，因此，你的提供者应该保存上层提供者到低层提供者之间的套接字映射。低层提供者完成 WSPSelect调用时，就可轻

松地向上层提供者反映哪些套接字上有I/O待发事件了。

一旦决定了哪些低层提供者套接字有待发的网络事件，就必须更新原来的 fd_set集合，这些集合是原来的SPI客户机投递的。 Ws2spi.h文件中提供了三个宏（ FD_CLR、FD_SET和FD_ZERO），它们可用于对原来的套接字集合进行管理。关于这些宏的说明，参见第 8章。程序清单14-2演示了怎样实施WSPSelect。

程序清单14-2 WSPSelect实施详解

```
int WSPAPI WSPSelect(
    int nfds,
    fd_set FAR * readfds,
    fd_set FAR * writefds,
    fd_set FAR * exceptfds,
    const struct timeval FAR * timeout,
    LPINT lpErrno)
{
    SOCK_INFO *SocketContext;
    u_int i;
    u_int count;
    int Ret;
    int HandleCount;

    // Build an Upper and Lower provider socket mapping table
    struct
    {
        SOCKET ClientSocket;
        SOCKET ProvSocket;
    } Read[FD_SETSIZE], Write[FD_SETSIZE], Except[FD_SETSIZE];

    fd_set ReadFds, WriteFds, ExceptFds;

    // Build the ReadFds set for the lower provider
    if (readfds)
    {
        FD_ZERO(&ReadFds);

        for (i = 0; i < readfds->fd_count; i++)
        {
            if (MainUpCallTable.lpWPUQuerySocketHandleContext(
                (Read[i].ClientSocket = readfds->fd_array[i]),
                (LPDWORD) &SocketContext, lpErrno) ==
                SOCKET_ERROR)
                return SOCKET_ERROR;
            FD_SET((Read[i].ProvSocket =
                SocketContext->ProviderSocket), &ReadFds);
        }
    }

    // Build the WriteFds set for the lower provider.
    // This is just like the ReadFds set above.
    ...

    // Build the ExceptFds set for the lower provider.
    // This is also like the ReadFds set above.
    ...
```

```
        // Call the lower provider's WSPSelect
        Ret = NextProcTable.lpWSPSelect(nfds,
            (readfds ? &ReadFds : NULL),
            (writefds ? &WriteFds : NULL),
            (exceptfds ? &ExceptFds : NULL), timeout, lpErrno);

        if (Ret != SOCKET_ERROR)
        {
            HandleCount = Ret;

            // Set up calling provider's readfds set
            if (readfds)
            {
                count = readfds->fd_count;
                FD_ZERO(readfds);

                for(i = 0; (i < count) && HandleCount; i++)
                {
                    if (MainUpCallTable.lpWPUFDIsSet(
                        Read[i].ProvSocket, &ReadFds))
                    {
                        FD_SET(Read[i].ClientSocket, readfds);
                        HandleCount--;
                    }
                }
            }

            // Set up calling provider's writefds set.
            // This is just like the readfds set above.
            ...
            // Set up calling provider's exceptfds set.
            // This is also like the readfds set above.
            ...
        }

        return Ret;
    }
```

3. WSAAsyncSelect模型

WSAAsyncSelect I/O模型涉及到的管理中，包括对套接字上网络事件的Windows消息通知进行管理。SPI客户机调用WSPAsyncSelect函数后，便可使用这个模型了。这个函数的定义如下：

```
int WSPAsyncSelect(
    SOCKET s,
    HWND hWnd,
    unsigned int wMsg,
    long lEvent,
    LPINT lpErrno
);
```

s参数代表SPI客户机的套接字，该套接字希望收到网络事件的窗口消息通知。 hWnd参数标志窗口句柄，在套接字s上发生IEvent参数中定义的网络事件时，这个窗口句柄便接收 wMsg参数中定义的消息。如果在执行这个函数时，返回了 SOCKET_ERROR，lpErrno参数便会收到一个Winsock错误代码。至于你的提供者必须支持哪些网络事件（ IEvent参数中定义的），

详情参见表8-3。

SPI客户机在调用WSPAsyncSelect时，你的提供者便利用客户机提供的窗口句柄和客户机提供的窗口消息（通过 WSPAsyncSelect调用投递的），负责通知SPI客户机，套接字上什么时候发生了网络事件。你的提供者调用 WPUPostMessage后，便可向SPI客户机通知网络事件了。WPUPostMessage的定义如下：

```
BOOL WPUPostMessage(
    HWND hWnd,
    UINT Msg,
    WPARAM wParam,
    LPARAM lParam
);
```

你的提供者利用hWnd参数标识即将通知的SPI客户机的窗口句柄。 Msg参数标识用户自定义的消息，该消息被投递到 WSPAsyncSelect的wMsg参数中。 wParam参数则接受发生网络事件的那个套接字句柄。最后一个参数是lParam，它接受两段信息。lParam的低位字（low word）接受的是已经发生的网络事件。在谈到 lParam低位字中标志的网络事件时，如果你的提供者实施中发生了错误，lParam的高位字（highword）则接受该错误的代码。

如果此时正在开发一个非 IFS分层式服务提供者，你的提供者就成为低层提供者的WSPAsyncSelect函数的一个客户机。这样一来，就需要你的提供者在调用 WSPAsyncSelect函数之前，利用WPUQuerySocketHandleContext，把一个SPI客户机的套接字句柄转换成你的提供者的套接字句柄。由于需要服务提供者利用 WPUPostMessage，向SPI客户机通知网络事件。你的提供者必须从低层提供者着手，拦截这些窗口信息。为什么呢？因为 WPUPostMessage将低层提供者的套接字句柄投回了 lParam的高位字里面。其结果是，你的提供者必须把 lParam中的高位字中的套接字句柄翻译为 SPI客户机的套接字句柄，这个句柄用在原来的WSPAsyncSelect调用中。

要想对低层提供者发出的窗口消息拦截进行管理，最好是建立一个“雇员”（worker）线程。这个线程将建立一个隐藏的窗口，对网络事件窗口消息进行管理。当你的提供者在低层套接字上调用WSPAsyncSelect时，你只须把“雇员”窗口句柄投给低层提供者的WSPAsyncSelect调用即可。从此，你的提供者的“雇员”窗口便会收到低层提供者发出的网络事件窗口，你的提供者便可利用WPUPostMessage，向SPI通知网络事件了。

4. WSAEventSelect模型

WSAEventSelect模型涉及套接字上发生网络事件时，怎样通知事件对象这方面的问题。调用WSPEventSelect函数之后，SPI客户机便可使用这个模型了。该函数的定义如下：

```
int WSPEventSelect(
    SOCKET s,
    WSAEVENT hEventObject,
    long lNetworkEvents,
    LPINT lpErrno
);
```

s参数代表SPI客户机的套装字，该套接字希望得到底网络事件通知。 hEventObject参数是一个WSAEVENT对象句柄，当套接字s上发生了lNetworkEvents中指定的网络事件时，你的提供者便向你通知这个句柄。 lpErrno参数收到一个Winsock错误代码，前提是实施该函数时，返

回了SOCKET_ERROR。你的提供者必须支持的网络事件（lNetworkEvents参数中标志的）和WSAAsyncSelect I/O模型中指定的一样。

非IFS分层提供者中的WSPEventSelect的实施实际上非常简单。SPI客户机下传一个事件对象，你的提倡者只须利用WPUQuerySocketHandleContext，把这个SPI客户机的套接字句柄翻译成低层提供者的套接字句柄即可。完成套接字句柄的翻译之后，便可利用这个事件对象，直接从这个SPI客户机中调用低层提供者的WSPEventSelect函数。这个SPI客户机的事件对象上发生I/O操作时，低层提供者便直接通知事件对象，以及向SPI客户机通知网络事件已到达。低层提供者向事件对象发出通知时，在事件完成之前，不会向SPI客户机返回套接字句柄。因此，你的提供者就无须为 SPI客户机执法套接字句柄的翻译。这一点与前面提到的WSAAsyncSelect模型不相同，在前一种模型中，因为套接字句柄被投入低层提供者发出的窗口消息中，因此你的提供者必须对一个已完成的网络事件执行套接字句柄的翻译。

如果你此时正在开发一个基础提供者，一个套接字上发生lNetworkEvents中指定的网络事件时，你的提供者便通过客户机提供的事件对象（通过WSPEventSelect调用传递的）负责向SPI客户机发出通知。你的提供者可利用上调函数WPUSetEvent向SPI客户机通知网络事件。该上调函数的定义如下：

```
BOOL WPUSetEvent (
    WSAEVENT hEvent,
    LPINT lpErrno
);
```

hEvent参数代表你的提供者必须通知的事件对象句柄，这个句柄是从WSPEventSelect开始投递的。如果该函数返回FALSE，lpErrno参数便随特定的Winsock错误代码一起返回。

5. OverlappedI/O模型

OverlappedI/O（重叠式I/O）模型要求你的提供者执行一个重叠式I/O管理器，同时为基于对象的事件和基于例程的完成重叠式I/O请求提供服务。下面的SPI函数采用了Winsock中的Win32重叠式I/O：

- WSPSend
- WSPSendTo
- WSPRecv
- WSPRecvFrom
- WSPIoctl

每个函数都突出了三个常见的参数：一个指向一个WSAOVERLAPPED结构的可选性指针、一个指向WSAOVERLAPPED_COMPLETION_ROUTINE函数的可选性指针以及一个指向WSATHREADID结构的指针（该指针标志执行调用的应用程序线程）。

重叠式I/O操作期间，服务提供者和SPI客户机之间是怎样进行通信的呢？其中的关键是WSAOVERLAPPED结构。该结构的格式如下：

```
typedef struct _WSAOVERLAPPED {
    DWORD    Internal;
    DWORD    InternalHigh;
    DWORD    Offset;
    DWORD    OffsetHigh;
    WSAEVENT hEvent;
} WSAOVERLAPPED, FAR * LPWSAOVERLAPPED;
```

服务提供者的重叠式I/O管理器负责管理SPI客户机WSAOVERLAPPED结构的Internal字段。开始重叠式处理时，服务提供者必须把 Internal字段设为WSS_OPERATION_IN_PROGRESS。这是非常重要的，因为如果SPI客户机调用WSPGetOverlappedResult，与此同时，你的提供者也在为一个待决重叠式操作服务，就可以把 WSS_OPERATION_IN_PROGRESS值用于你的提供者实施的WSPGetOverlappedResult调用中，以此判断重叠式操作是否处于处理过程中。

当一个I/O操作结束时，提供者便设置OffsetHigh和Offset字段。OffsetHigh设为该操作返回的结果Winsock错误代码，Offset则应该设为WSPRecv和WSPRecvFrom I/O操作返回的结果标志。在设置了这几个字段之后，提供者便通过一个事件对象或完成例程（根据上面的可选WSAOVERLAPPED结构所用的I/O函数来决定），通知已完成重叠操作请求的SPI客户机。

(1) Event（事件）

在基于事件的重叠式I/O模型中，SPI客户机随内含一个事件对象的WSAOVERLAPPED结构，调用了我们前面提到的一个I/O函数。另外，必须把WSAOVERLAPPED_COMPLETION_ROUTINE设为NULL（空）值。服务提供者负责管理重叠式I/O请求。请求完成之后，提供者必须提醒调用者的已完成的I/O请求线程。这是通过调用上调函数 WPUCompleteOverlapped Request来完成的。该函数的定义如下：

```
WSAEVENT WPUCompleteOverlappedRequest(
    SOCKET s,
    LPWSAOVERLAPPED lpOverlapped,
    DWORD dwError,
    DWORD cbTransferred,
    LPINT lpErrno
);
```

s参数和lpOverlapped参数代表最初的套接字，而WSAOVERLAPPED结构则始发于客户机。提供者应该把dwError参数设为重叠式I/O请求的完成状态，cbTransferred参数设为重叠式操作期间所传输的字节数。最后一个参数是lpErrno，如果这个函数调用返回SOCKET_ERROR的话，它就会报告相应的Winsock错误代码。WPUCompleteOverlapped Request完成时，便在SPI客户机的WSAOVERLAPPED结构中设置两个字段： InternalHigh设为cbTransferred字节数，Internal设为除了WSS_OPERATION_IN_PROGRESS之外的一个值。

在基于事件的重叠式I/O模型中，SPI客户机最终会调用. WSPGetOverlappedResult，以获得重叠式请求完成之后的结果。该函数的定义如下：

```
BOOL WSPGetOverlappedResult (
    SOCKET s,
    LPWSAOVERLAPPED lpOverlapped,
    LPDWORD lpcbTransfer,
    BOOL fWait,
    LPDWORD lpdwFlags,
    LPINT lpErrno
);
```

调用WSPGetOverlappedResult时，服务提供者必须报告原来的重叠式操作请求的操作状态。正如我们前面提到的那样，提供者负责管理 Internal、InternalHigh、Offset和OffsetHigh，这四个字段在SPI客户机的WSAOVERLAPPED结构中。调用WSPGetOverlappedResult时，提供者应该首先检查SPI客户机的WSAOVERLAPPED结构中的Internal字段。如果Internal字端被

设为WSS_OPERATION_IN_PROGRESS，则表示提供者仍然在处理一个重叠式请求。如果WSPGetOverlappedResult的fWait参数被设为TRUE（真），提供者就必须在返回结果之前，在投入SPI客户机的WSAOVERLAPPED结构的事件句柄上等候这个重叠操作结束。如果 fWait参数被设为FALSE（假），提供者就会返回 Winsock错误WSA_IO_INCOMPLETE。等待返回结果之后，这个重叠式操作一旦完成，提供者就应该这样设置 WSPGetOverlappedResult函数的各个参数：

- 把lpcsTransfer设为WSAOVERLAPPED结构中的InternalHigh字段的值，从中得知收发操作所传输的字节是多少。
- 把lpdwFlags设为WSAOVERLAPPED结构中的Offset字段的值，从中得知 WSPRecv或WSPRecvFrom操作的结果标志。
- 把lpErrno设为WSAOVERLAPPED结构中的OffsetHigh字段的值，从中得知结果化的错误代码。

程序清单14-3演示了怎样实施WSPGetOverlappedResult。

程序清单14-3 WSPGetOverlappedResult实施详解

```
BOOL WSPAPI WSPGetOverlappedResult(
    SOCKET s,
    LPWSAOVERLAPPED lpOverlapped,
    LPDWORD lpcbTransfer,
    BOOL fWait,
    LPDWORD lpdwFlags,
    LPINT lpErrno)
{
    DWORD Ret;

    if (lpOverlapped->Internal != WSS_OPERATION_IN_PROGRESS)
    {
        *lpcbTransfer = lpOverlapped->InternalHigh;
        *lpdwFlags = lpOverlapped->Offset;
        *lpErrno = lpOverlapped->OffsetHigh;

        return(lpOverlapped->OffsetHigh == 0 ? TRUE : FALSE);
    }
    else
    {
        if (fWait)
        {
            Ret = WaitForSingleObject(lpOverlapped->hEvent,
                INFINITE);
            if ((Ret == WAIT_OBJECT_0) &&
                (lpOverlapped->Internal !=
                WSS_OPERATION_IN_PROGRESS))
            {
                *lpcbTransfer = lpOverlapped->InternalHigh;
                *lpdwFlags = lpOverlapped->Offset;
                *lpErrno = lpOverlapped->OffsetHigh;

                return(lpOverlapped->OffsetHigh == 0 ? TRUE :
                    FALSE);
            }
            else
```

```
                *lpErrno = WSASYSCALLFAILURE;
            }
            else
                *lpErrno = WSA_IO_INCOMPLETE;
        }

        return FALSE;
    }
```

(2) Completion routine（完成例程）

在基于完成例程的重叠式I/O模型中，SPI客户机随一个WSAOVERLAPPED结构和一个WSAOVERLAPPED_COMPLETION_ROUTINE指针一起，调用了前面提到的I/O函数之一。服务提供者负责管理重叠式 I/O请求。请求完成时，提供者必须利用 Win32异步进程调用（APC）I/O机制，提醒调用者的完成I/O线程注意。APC机制要求调用者线程处于“警觉等待状态”（参见第8章）。服务提供者结束对基于完成例程的重叠式请求提供的服务时，必须提醒SPI客户机注意请求的操作已完成。这是通过调用上调函数 WPUQueueApc来完成的。该函数的定义如下：

```
int WPUQueueApc(
    LPWSATHREADID lpThreadId,
    LPWSAUSERAPC lpfnUserApc,
    DWORD dwContext,
    LPINT lpErrno
);
```

lpThreadId参数代表SPI客户机的WSATHREADID结构，这个结构是从一个提供完成例程的I/O调用开始投递的。lpfnUserApc参数代表一个指向WSAUSERAPC函数的指针，这个函数扮演的是一个中间函数的角色，提供者必须提供这个函数，以便对 SPI客户机的回调。作为一个中间函数，它必须调用SPI客户机的WSAOVERLAPPED_COMPLETION_ROUTINE，这个函数是最初的重叠式调用提供的。WSAUSERAPC中间函数的原型定义如下：

```
typedef void (CALLBACK FAR * LPWSAUSERAPC)(DWORD dwContext);
```

注意，这个函数的定义中只有一个参数——dwContext。SPI客户机调用这个回调函数时，dwContext参数中包含的信息原来是投入WPUQueueApc的dwContext参数中的。从本质上来说，在调用SPI客户机的WSAOVERLAPPED_COMPLETION_ROUTINE时，dwContext参数允许你投递一个数据结构，这个结构中包含自己可能需要的信息元素。它在第 8章中的定义是这样的：

```
void CALLBACK CompletionROUTINE(
    IN DWORD dwError,
    IN DWORD cbTransferred,
    IN LPWSAOVERLAPPED lpOverlapped,
    IN DWORD dwFlags
);
```

服务提供者应该通过WPUQueueApc的dwContext参数，投递下列信息：

- 当作Winsock错误代码的重叠式操作的状态。
- 通过重叠式操作传输的字节数是多少。
- 调用者的WSAOVERLAPPED结构。

■ 调用者投入I/O调用中的标志。

有了这些信息，提供者便可成功地通过中间完成例程，调用SPI客户机的WSAOVERLAPPED_COMPLETION_ROUTINE。

(3) Completion ports（完成端口）

在Winsock 2中，完成端口I/O模型的实施是在Ws2_32.dll模块中进行的。正如我们在第 8章中所讲的那样，完成端口模型是建立在使用基于事件的重叠式 I/O模型基础上的。因此，在管理完成端口模型时，服务提供者不必要考虑太多。

(4) 重叠I/O的管理

SPI客户机在利用前面提到的任何一个重叠式 I/O模型，调用你的分层提供者时，对你的提供者的重叠式管理器来说，它应该利用基于事件的或第 8章中所讲的完成端口重叠式 I/O方法，调用低层提供者。如果此时你的提供者运行于 Windows NT或Windows 2000，我们建议大家在低层提供者上采用完成端口来执行重叠式 I/O。在Windows 95和Windows 98平台上，只能使用基于事件的重叠式I/O。在处理重叠式I/O事件时，下面三个上调函数可供你的提供者使用：

```
WSAEVENT WPUCreateEvent(LPINT lpErrno);
BOOL WPUResetEvent(WSAEVENT hEvent, LPINT lpErrno);
BOOL WPUCloseEvent(WSAEVENT hEvent, LPINT lpErrno);
```

WPUCreateEvent函数建立并返回一个事件对象。这个函数完全和第 8章中讲的WSACreateEvent函数一样：建立了一个处于手工重设模式的事件对象。如果 WPUCreateEvent不能建立事件对象，就会返回 NULL，而且lpErrno中将包含特定的Winsock错误代码。WSPResetEvent函数则和WSAResetEvent函数一样：把一个事件对象（参数 hEvent）的signaled状态重设为unsignaled状态。WPUCloseEvent函数和WSACloseEvent函数一样，释放与事件对象句柄关联在一起的所有的操作资源。

本书配套光盘上，我们的LSP示例采用了基于事件的重叠式I/O来管理SPI客户机的所有的I/O活动。这样，我们的 LSP示例便可在 Windows 2000、Windows NT、Windows 98以及Windows 95上使用。但是，必须注意：这个示例只有一个服务重叠式 I/O操作的线程，这样一来，我们的提供者便不能同时利用基于事件的重叠式 I/O为多个WSA_MAXIMUM_WAIT_EVENTS(64)事件对象服务（参见第8章）。如果你的提供者希望为64个以上的事件对象提供服务，就可以建立多个服务线程，提供对多个事件对象的服务。如果此时正在针对Windows NT和Windows 2000开发提供者，我们建议大家采用完成端口，而不是基于事件的重叠式I/O，这样便可不受它的限制。

14.2.6 扩展函数

Winsock库Mswsock.lib为应用程序提供了扩展函数，增强了 Winsock的功能。表14-2对目前已获支持的扩展函数进行了定义。

表14-2 Winsock扩展函数

扩展函数	GUID
AcceptEx	WSAID_ACCEPTEX
GetAcceptExSockaddrs	WSAID_GETACCEPTEXSOCKADDRS
TransmitFile	WSAID_TRANSMITFILE
WSARecvEx	没有相关的GUID

链接到 Mswsock.lib的一个应用程序在使用 AcceptEx、GetAcceptExSockaddrs和TransmitFile时，利用SIO_GET_EXTENTION_FUNCTION_POINTER选项，隐式地调用了提供者的WSPIoctl函数。WSPIoctl的定义如下：

```
int WSPIoctl(
    SOCKET s,
    DWORD dwIoControlCode,
    LPVOID lpvInBuffer,
    DWORD cbInBuffer,
    LPVOID lpvOutBuffer,
    DWORD cbOutBuffer,
    LPDWORD lpcbBytesReturned,
    LPWSAOVERLAPPED lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine,
    LPWSATHREADID lpThreadId,
    LPINT lpErrno
);
```

调用这个函数时， dwIoControlCode参数被设为 SIO_GET_EXTENSION_FUNCTION_POINTER这个值。lpvInBuffer参数中包含指向一个通用唯一标志符（GUID）的指针，它表示Mswsock.lib正在寻找哪些扩展函数（表 14-2中所列的GUID值定义了目前已获支持的扩展函数）。如果这个GUID值与其定义的值匹配，提供者就必须通过 lpvOutBuffer，返回一个函数指针，指向提供者的扩展函数。其余的参数不适宜直接管理 SPI中的扩展函数。

特别注意，表 14-2中的WSARecvEx函数没有 GUID。这是因为 WSARecvEx没有调用WSPIoctl，而是直接调用了WSARecv。其结果是提供者不能直接监视 WSARecvEx函数。

14.2.7 传输服务提供者的安装

要安装传输服务提供者，应开发一个简单的应用程序，使其将一个分层或基础提倡者插入服务提供者的Winsock 2目录。传输提供者的安装方式决定了它是一个分层提供者，还是一个基础提供者。安装程序只在 Winsock 2系统配置数据库中，配置了传输提供者，这个系统配置数据库是一个目录，系统上已安装的服务提供者都在这个目录中。配置数据库让 Winsock 2得知服务提供者已存在，并定义了提供的服务类型。 Winsock应用程序建立套接字时，Winsock 2便利用这个数据库来判断需要加载哪些传输服务提供者。 WS2_32.DLL在数据库中搜索第一个与socket和WSASocket API调用的套接字输入参数（比如说地址家族、套接字类型和协议）匹配的提供者。一旦找到与之匹配的条目， Ws2_32.dll便加载相应的服务提供者之DLL（动态数据链接库），这个DLL是定义在该目录中的。

从本质上说，要在Winsock 2服务提供者数据库内成功安装和管理服务提供者条目，需要四个SPI函数。这四个函数的前缀均为WSC：

- WSCEnumProtocols
- WSCInstallProvider
- WSCWriteProviderOrder
- WSCDeInstallProvider

这些函数利用WSAPROTOCOL_INFOW结构，对服务提供者数据库进行查询和操作，关于这个结构，可参见第5章。由于我们的目的是安装传输服务提供者，因此主要讨论该结构的

这三个字段：ProviderId、dwCatalogEntryId和ProtocolChain。ProviderId字段是一个GUID，允许你在任何系统上定义和安装唯一一个提供者。 dwCatalogEntryId字段只用一个唯一的数字值标识这个数据库中的 WSAPROTOCOL_INFOW目录条目结构。 ProtocolChain字段则决定WSAPROTOCOL_INFOW结构是一个基础提供者目录条目，还是一个分层提供者亦或是一个提供者协议链。ProtocolChain字段是一个WSAPROTOCOLCHAIN结构，该结构的定义如下：

```
typedef struct {
    int ChainLen;
    DWORD ChainEntries[MAX_PROTOCOL_CHAIN];
} WSAPROTOCOLCHAIN, FAR * LPWSAPROTOCOLCHAIN;
```

ChainLen字段决定一个目录条目是代表一个分层式提供者（当 ChainLen = 0时），还是一个基础提供者（当 ChainLen = 1时），或者是一个协议链（当 ChainLen >1时）。协议链（protocol chain）是一个特殊的目录条目，定义分层式服务提供者在 Winsock和其他服务提供者之间的位置（参见图 14-2）。服务提供者数据库中，不管是分层提供者，还是基础提供者，每个提供者都只有一个目录条目。最后一个字段 ChainEntries1，是一个目录ID数组，用于说明服务提供者加入协议链的顺序。建立套接字期间， Ws2_32.dll在目录中搜索相应的服务提供者时，只查找协议链和基础提供者目录条目。分层式提供者目录条目（ ChainLen为0）则被Ws2_32.dll忽略，它存在于协议链目录条目中的唯一目的便是向一个协议链标识分层提供者。

1. 基础提供者的安装

要安装一个基础提供者，必须建立一个 WSAPROTOCOL_INFOW目录条目结构，用它来代表基础提供者。这样便要求用能够说明该基础提供者的协议属性信息，来填充该结构中的字段。记住，ProtocolChain结构中的ChainLen 字段应设为1。这个基础结构一旦定义，就需要利用WSCInstallProvider函数，把它安装在目录中。 WSCInstallProvider函数的定义如下：

```
int WSCInstallProvider(
    const LPGUID lpProviderId,
    const LPWSTR lpszProviderDllPath,
    const LPWSAPROTOCOL_INFOW lpProtocolInfoList,
    DWORD dwNumberOfEntries,
    LPINT lpErrno
);
```

lpProviderId参数是一个 GUID，允许你向 Winsock目录标识一个协议提供者。lpszProviderDllPath参数是一个字串，其中包括到该提供者之DLL的加载路径。这个字串中可包括%SystemRoot%之类的环境变量。 lpProtocolInfoList参数代表安装在这个目录中的一个WSAPROTOCOL_INFOW数据结构的数组。为了安装基础提供者，可将 WSAPROTOCOL_INFOW结构分配给该数组的第一个元素。从dwNumberOfEntries参数中，可得知lpProtocolInfoList数组中有多少条目。如果WSCInstallProvider函数失败，返回SOCKET_ERROR时，lpErrno参数中就会包含特定的错误代码信息。

2. 分层提供者的安装

要安装分层式服务提供者，需要建立两个 WSAPROTOCOL_INFOW目录条目结构。一个将代表分层提供者（比如协议链长度等于 0），另一个将代表一个协议链（比如，协议链长度大于1），该协议链把分层提供者与一个基础提供者链接起来。应该用一个现成服务提供者的WSPROTOCOL_INFOW目录条目结构的属性来初始化这两个结构。调用 WSCEnumProtocols函数（定义如下），便可获得这个现成服务提供者的WSPROTOCOL_INFOW目录条目结构了。

```
int WSCEnumProtocols(
    LPINT lpiProtocols,
    LPWSAPROTOCOL_INFOW lpProtocolBuffer,
    LPDWORD lpdwBufferLength,
    LPINT lpErrno
);
```

lpiProtocols参数是一个可选的值数组。如果lpiProtocols是NULL（真），就会返回所有有效协议的信息；否则，只能取得该数组中列出的协议信息。 lpProtocolBuffer参数是一个应用程序提供的缓冲区，用 Winsock 2中的 WSAPROTOCOL_INFOW结构来充填。输入端的lpdwBufferLength参数是字节数，表示投入WSCEnumProtocols中的lpProtocolBuffer缓冲区中的字节是多少。若是在输出端，则收到缓冲区的最小长度，这个长度可投入 WSCEnumProtocols内，获得全部所请求的信息。如果函数调用失败，并返回 SOCKET_ERROR，lpErrno参数中就会出现特定的错误信息。一旦有了准备进行重叠操作的提供者目录条目，就可以把这个提供者的WSAPROTOCOL_INFOW结构复制到新建立的结构中。

初始化之后，首先需要用 WSCInstallProvider来安装你的分层提供者目录条目，然后，利用WSCEnumProtocols列举出所有的目录条目，获得安装之后为这个结构分配的目录 ID。然后，用这个目录条目来设置一个协议链目录条目，通过它，便将你的分层提供者和另一个提供者链接起来。接下来，便调用 WSCInstallProvider来安装这个链式提供者。下面的伪代码对此进行了解释：

```
...
WSAPROTOCOL_INFOW LayeredProtocolInfoBuff,
                  ProtocolChainProtoInfo,
                  BaseProtocolInfoBuff;
...
// Retrieve BaseProtocolInfoBuff using WSCEnumProtocols()

memcpy (&LayeredProtocolInfoBuff , &BaseProtocolInfoBuff,
    sizeof(WSAPROTOCOL_INFO));
LayeredProtocolInfoBuff.dwProviderFlags = PFL_HIDDEN;
LayeredProtocolInfoBuff.ProviderId = LayeredProviderGuid;

// This entry will be filled in by the system
LayeredProtocolInfoBuff.dwCatalogEntryId = 0;

LayeredProtocolInfoBuff.ProtocolChain.ChainLen =
    LAYERED_PROTOCOL;
WSCInstallProvider(&LayeredProviderGuid, L"lsp.dll",
    &LayeredProtocolInfoBuff, 1, & install_error);

// Determine the catalog ID of the layered provider
// using the WSCEnumProtocols() function
for (i = 0; i < TotalProtocols; i++)
    if (memcmp (&ProtocolInfo[i].ProviderId, &ProviderGuid,
        sizeof (GUID))==0)
   {
      LayeredCatalogId = ProtocolInfo[i].dwCatalogEntryId;
      break;
   }

Memcpy(&ProtocolChainProtoInfo, &BaseProtocolInfoBuff,
       sizeof(WSAPROTOCOL_INFO));
ProtocolChainProtoInfo.ProtocolChain.ChainLen = 2;
```

```
ProtocolChainProtoInfo.ProtocolChain.ChainEntries[0] =
    LayeredProvideProtocolInfo.dwCatalogEntryId;
ProtocolChainProtoInfo.ProtocolChain.ChainEntries[1] =
    BaseProtocolInfoBuff.dwCatalogEntryId;

WSCInstallProvider(
    &ChainedProviderGuid,
    L"lsp.dll",                 // lpszProviderDllPath
    &ProtocolChainProtoInfo,    // lpProtocolInfoList
    1,                          // dwNumberOfEntries
    &install_error              // lpErrno
);
```

注意，PFL_HIDDEN标志是在WSAPROTOCOL_INFOW结构中，针对以上伪代码中的分层提供者指定的。这个标志确保 WSAEnumProtocols函数（参见第5章）返回的缓冲区内不会出现这个分层提供者的目录。

安装程序应该管理的另一个重要标志是 XP1_IFS_HANDLES。对任何一个非IFS的服务提供者来说，只要它建立自己的套接字句柄时，用的是 WPUCreateSocketHandle，就不应该在自己的WSAPROTOCOL_INFOW结构中设置这个 XP1_IFS_HANDLES标志。没有设置XP1_IFS_HANDLES标志， Winsock应用程序便会由于前面提到的性能下降而避免使用ReadFile和WriteFile。

3. 提供者的排序

系统上一旦安装了服务提供者，就必须考虑 Winsock 2怎样在数据库中搜索服务提供者这一问题。多数Winsock应用程序都通过socket或WSASocket函数中的参数，指定了自己需要的协议。举个例子来说，如果你的应用程序利用地址家族AF_INET和套接字类型SOCK_STEREAM建立了一个套接字， Winsock 2便在能够满足这一需要的数据库中搜索默认的TCP/IP协议链或基础提供者目录条目。在用 WSCInstallProvider安装一个服务提供者时，目录条目便自动成为数据库中的最后一个条目。要使服务提供者成为默认的 TCP/IP提供者，必须对数据库中的提供者进行重新排序，并把协议链目录条目放在 TCP/IP提供者之前，这是通过调用WSCWriteProviderOrder来完成的，它的定义如下：

```
int WSCWriteProviderOrder(
    LPDWORD lpwdCatalogEntryId,
    DWORD dwNumberOfEntries
);
```

lpwdCatalogEntryId参数接受一个由目录ID组成的数组，这些ID用于标志目录的排序。正如前面所讲的那样，调用WSCEnumProtocols，便可获得该目录中的目录ID了。dwNumberOfEntries参数是一个计数，表示这个数组中有多少目录条目。如果 WSCWriteProviderOrder调用成功，就返回ERROR_SUCCESS(0)；否则，就返回一个Winsock错误代码。

WSCWriteProviderOrder函数在 Ws2_32.dll库内。要使用它，应用程序必须链接到Sporder.lib。另外，这个库关联的 Sporder.lib模块也不在Windows操作系统中。其支持 DLL可在“微软开发者网络”（MSDN）库中找到。如果你打算在编程过程中使用它，必须将它包括到你的安装程序中。 MSDN库还提供了一个很方便的软件工具，名为 Sporder.exe，允许你对Winsock 2数据库中的目录条目进行查看和重新排序。图 14-3是一个示例，在Windows 2000系统上安装一个分层式提供者之后，对 Winsock 2进行的快速查看（利用Sporder.exe进行的）。

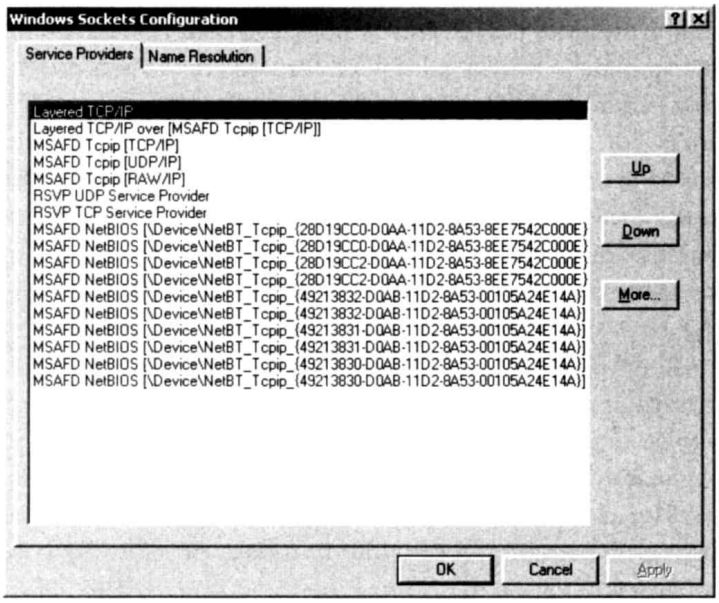

图14-3 Sporder.exe

4. 服务提供者的删除

从Winsock 2目录中删除一个服务提供者很简单。主要的任务便是调用WSCDeinstallProvider函数，该函数的定义如下：

```
int WSCDeinstallProvider(
   LPGUID lpProviderId,
   LPINT lpErrno
);
```

lpProviderId参数代表准备删除的服务提供者的 GUID。如果函数返回SOCKET_ERROR，lpErrno参数就会接收特定的Winsock错误代码。

在删除服务提供者时，必须考虑到服务提供者的目录 ID完全可能包含在一个分层式服务提供者中。如果是这样，应该从所有引用了提供者的协议链中，将你的目录 ID删除。

安装分层式提供者所涉及的问题

分层式服务提供者的确大大发挥了连网服务的潜能。但是，目前的 Winsock 2规格中，尚未解决一个问题：如果一个分层式服务提供者发现系统中已安装了另一个分层式服务提供者，怎样才知道自己应插在协议链中的何处呢？举个例子来说，若你打算在一个已安装了URL过滤提供者的系统上，安装一个数据加密提供者，显而易见，在一个现成的协议链中，数据加密提供者应该插在过滤提供者下面。但问题是提供者安装程序无法找到现成提供者提供的服务类型，因而也无从得知自己应该插在协议链中的什么地方。在由主管决定安装哪些提供者以及对这些提供者进行排序的联网环境中，这倒不是个大问题。但随着分层式提供者的日益普及，这个问题越来越明显。唯一安全的安装办法便是：直接在基础提供者上安装分层式提供者，并使新的协议链成为协议的默认提供者。这样，可以保证新提供者的服务，但删除了被当作默认提供者链的现成的分层式提供者。

除了分层式服务提供者在协议链中的排序问题外， Winsock 2规格中还有另一个相关

问题没有得到解决：现成的分层式提供者如何才能避免在发生时，被修改和警告呢？和前一个问题相比，这个问题倒不大。实际上，不修改一个分层式提供者协议链，分层式提供者的硬编码器照样可将协议链的顺序硬编码到 WSPStartup函数中，并在 LSP的WSAPROTOCOL_INFOW结构中的ProtocolChain.ChainLen指定成1，把这个分层式提供者当作一个基础提供者安装到系统。

14.3 命名空间服务提供者

通过第10章的学习，大家已经知道应用程序是怎样在名字空间内注册和解析服务的，这对网络上动态建立的服务来说，是一个非常关键的特性。不幸的是，由于种种限制，现有的名字空间几乎不能发挥它们的重要作用。但是 Winsock 2提供了一种方法，供建立你自己的名字空间所用，通过这个方法，可以采取任何一种自己喜欢的方式对命字注册和解析进行处理。

这是通过建立一个可实施 9个名字空间函数的 DLL来完成的。这些函数的前缀均是 NSP，与第 10章中讲的 RNR函数对应。比方说，等同于 WSASetService的名字空间函数便是NSPSetService。DLL建立之后，便利用标志这个名字空间的 GUID安装到系统目录中。随后，应用程序便可以在你的名字空间内注册和查询服务了。

本小节，我们先为大家介绍如何安装名字空间提供者，然后对名字空间提供者必须实施的函数进行说明。最后，随注册和解析服务所用的一个示范应用程序，向大家展示一个示例性的名字空间提供者。

14.3.1 名字空间的安装

简单说来，名字空间只是一个 DLL，可以实施名字空间提供者函数。在应用程序可使用名字空间之前，必须通过 WSCInstallNameSpace函数，使系统知道这个名字空间。反之，一旦安装了提供者，就可以分别利用 WSAEnableNSProvider和WSAUnInstallNameSpace这两个函数，取消这个提供者或从系统目录中将它删除。接下来便是对这些函数的说明。

1. WSCInstallNameSpace

这个函数用于把一个提供者安装到系统目录中。它的声明如下：

```
int WSCInstallNameSpace (
    LPWSTR lpszIdentifier,
    LPWSTR lpszPathName,
    DWORD dwNameSpace,
    DWORD dwVersion,
    LPGUID lpProviderId
);
```

大家注意到的第一点便是所有字串参数都是宽位字符串。事实上，所有的名字空间提供者都是用宽位字符串来实施的。稍后将详细讨论。lpszIdentifier参数是名字空间提供者的名字。这个名字是在调用WSAEnumNameSpaceProviders时，列举得来的（参见第10章）。lpszPathName参数是DLL的位置。这个字串中可包含% SystemRoot%之类的环境变量。 dwNameSpace参数是该名字空间的数字化标志符。比如说，头文件 Nspapi.h定义了一些众所周知的名字空间，比如IPX SAP的NS_SAP。dwVersion参数则为该名字空间设置版本号。最后， lpProviderId参数是一个标志该名字空间的GUID。

如果调用成功，WSCInstallNameSpace就返回0；若失败，则返回SOCKET_ERROR。最常见的失败是WSAEINVAL，表示带有这个GUID标志的名字空间已经存在；如果是WSAEACCESS，就表示调用进程不具备安装特权。只有主管级的用户才能安装名字空间。

2. WSCEnableNSProvider

这个函数用于修改名字空间提供者的状态。可用于启用或取消提供者。它的声明如下：

```
int WSCEnableNSProvider (
    LPGUID lpProviderId,
    BOOL fEnable
);
```

lpProviderId参数是名字空间的GUID标识符，表示准备对这个名字空间进行修改。fEnable参数是一个布尔值，表示禁用或启用提供者。已禁用的提供者不能用于处理查询或注册服务。

如果调用成功，WSCEnableNSProvider函数就会返回0；若失败，便返回SOCKET_ERROR。如果提供者的GUID无效，就会返回WSAEINVAL。

3. WSCUnInstallNameSpace

这个函数用于从目录中删除名字空间提供者。它的定义如下：

```
int WSCUnInstallNameSpace ( LPGUID lpProviderId );
```

lpProviderId参数是准备删除的那个名字空间的GUID。如果该GUID无效，函数调用就会失败，并返回WSAEINVAL。

14.3.2 名字空间的实施

名字空间必须实施9个名字空间函数，这些函数与第10章中所讲的RNR函数相对应。除了实施这些函数外，还必须开发一个保持数据的方法。也就是说，还必须保持除了DLL实例之外的数据。每个加载DLL的进程都会接收自己的数据分段，这意味着保存在DLL内的数据（即固有数据）不能供其他实例共享（事实上，加载DLL的各个应用程序之间可以共享信息。不过练习中不提倡）。第10章，我们提到了名字空间的三种类型：动态、静态和固定状态。显然，实施静态名字空间不可取，因为它不允许对服务通过编程的方式进行注册。稍后，我们将具体讲讲如何保持名字空间固有的数据。

另外，大家还必须知道这一点：在所有名字空间提供者函数中，使用宽位字符串是非常重要的。不仅要求函数中的字串参数如此，要求RNR结构中的字串也要如此，比如WSAQUERYSET和WSASERVICECLASSINFO。大家也许有些迷惑不解：应用程序在注册或解析一个名字时，既可用RNR函数及结构的普通（ASCII）版本，又可以用它们的宽位字符（Unicode）版本，怎么可能呢？答案是随便哪个版本都可以，因为所有的ASCII调用都可通过一个中间层，将全部字串统一转换成宽位字符串。函数调用并返回之后便如此，即如果WSAQUERYSET返回调用程序（随WSALookupServiceNext一起）这个名字空间提供者返回的全部数据起初都是Unicode字符，但在函数调用返回之前，就转换成了ASCII字符。换言之，如果你的应用程序使用的是RNR函数，那么调用它的宽位字符版本就要快得多，因为无须进行转换。

在名字空间必须实施的这9个函数中，只有7个函数有相应的Winsock 2 RNR函数（参见表14-3）。其余两个函数用于初始化和清除名字空间。系统中一旦安装了名字空间，应用程序通

过指定GUID或安装期间指定的名字空间标识符，便可以使用这个名字空间了。接下来，应用程序调用我们在第10章中讲到的标准Winsock2RNR函数。调用其中一个函数时，也会调用其相应的名字空间提供者函数。比如说，一个应用程序在调用 WSAInstallServiceClass（该函数引用了自定义名字空间的GUID）时，也会调用那个提供者的NSPInstallServiceClass函数。关于各个名字空间，我们将在下一小节一一说明。

表14-3 Winsock 2注册和名解析函数与名字空间提供者函数之间的对应关系

Winsock函数	等同（或对应）的名字空间提供者函数
WSAInstallServiceClass	NSPInstallServiceClass
WSARemoveServiceClass	NSPRemoveServiceClass
WSAGetServiceClassInfo	NSPGetServiceClassInfo
WSASetService	NSPSetService
WSALookupServiceBegin	NSPLookupServiceBegin
WSALookupServiceNext	NSPLookupServiceNext
WSALookupServiceEnd	NSPLookupServiceEnd

1. NSPStartup

只要加载名字空间提供者DLL，便会调用NSPStartup函数。你的名字空间实施中必须包括这个函数，而且必须从DLL中导出。为使名字空间提供者正常工作，它所需的与 DLL有关的数据结构都可在调用这个函数时得到分配。这个函数的原型定义如下：

```
int NSPStartup (
    LPGUID lpProviderId,
    LPNSP_ROUTINE lpnspRoutines
);
```

第一个参数是lpProviderId，它是这个名字空间提供者的GUID。lpnspRoutines参数是一个NSP_ROUTINE结构，这个结构是你在实施这个函数时，必须填充的。这个结构提供了 8个函数指针，分别指向你的提供者的另8个名字空间函数。NSP_ROUTINE对象的定义如下：

```
typedef struct _NSP_ROUTINE
{
    DWORD                    cbSize;
    DWORD                    dwMajorVersion;
    DWORD                    dwMinorVersion;
    LPNSPCLEANUP             NSPCleanup;
    LPNSPLOOKUPSERVICEBEGIN  NSPLookupServiceBegin;
    LPNSPLOOKUPSERVICENEXT   NSPLookupServiceNext;
    LPNSPLOOKUPSERVICEEND    NSPLookupServiceEnd;
    LPNSPSETSERVICE          NSPSetService;
    LPNSPINSTALLSERVICECLASS NSPInstallServiceClass;
    LPNSPREMOVESERVICECLASS  NSPRemoveServiceClass;
    LPNSPGETSERVICECLASSINFO NSPGetServiceClassInfo;
} NSP_ROUTINE, FAR * LPNSP_ROUTINE;
```

这个结构的第一个字段 cbSize，表明NSP_ROUTINE结构的长度。下两个字段，dwMajorVersion和dwMinorVersion，包括你的提供者的版本信息（可以是任意的）。提供者条目的其余部分设为它们各自的函数指针。比如说，提供者为 NSPSetService字段分配了自己的NSPSetService函数地址（不管采用的是什么名字）。你的提供者函数名可以是任意的，但其参数和返回类型必须与提供者的定义匹配。

NSPStartup实施中，唯一需要做的是填充NSP_ROUTINE结构。一旦完成填充和初始化例程，如果成功的话，函数就会返回 NO_ERROR。如果发生错误，就会返回 SOCKET_ERROR和Winsock错误代码。比如，如果一个提供者尝试分配内存失败了，它就会随 WSA_NOT_ENOUGH_MEMORY参数一起，调用WSASetLastError，然后再返回SOCKET_ERROR。

现在，该好好讨论一下提供者的 DLL中的错误处理了。为提供者实施的所有函数调用若成功，便返回NO_ERROR；若失败，便返回 SOCKET_ERROR。如果判断调用失败了，必须在返回之前，设置相应的Winsock错误代码。否则，任何试图利用你的名字空间进行的注册和查询服务的应用程序都会报告 RNR函数失败，WSAGetLastError返回0。这样可能会为那些试图从容处理错误的应用程序带来许多麻烦； 0也不是我们希望返回的值，因为它说明不了任何问题。

2. NSPCleanup

卸载提供者的DLL时，便调用这个例程。有了这个函数，可以释放为 NSPStartup例程分配的所有内存。这个例程的定义如下：

```
int NSPCleanup ( LPGUID lpProviderId );
```

唯一的参数就是你的名字空间提供者的 GUID。该函数的唯一作用便是清除全部动态分配的内存。

3. NSPInstallServiceClass

NSPInstallServiceClass函数的对应函数是 WSAInstallServiceClass，负责注册服务类。NSPInstallServiceClass函数的定义如下：

```
int NSPInstallServiceClass (
    LPGUID lpProviderId,
    LPWSASERVICECLASSINFOW lpServiceClassInfo
);
```

第一个参数是提供者的 GUID。lpServiceClassInfo参数是正在注册的 WSASERVICE-CLASSINFOW结构。你的提供者必须保持一个服务类列表，确保这个 WSASERVICE-CLASSINFOW结构内，采用这个GUID的服务类只有一个。如果这个GUID已被采用，提供者肯定会返回WSAEALREADY。不然，提供者就应该保持这个服务类，以便别的 RNR操作能够引用它。

其余的名字空间提供者函数一般引用已安装的服务类。

4. NSPRemoveServiceClass

这个函数和NSPInstallServiceClass函数对应（一个用来安装，一个用来删除）。这个名字空间函数的对应函数是WSARemoveServiceClass。它的声明如下：

```
int NSPRemoveServiceClass (
    LPGUID lpProviderId,
    LPGUID lpServiceClassId
);
```

和上一个函数一样，第一个参数是提供者的 GUID。第二个参数lpServiceClassId，是即将删除的服务类的GUID。提供者必须从存储设备上删除指定的服务类。如果没有找到 lpService ClassId指定的服务类，提供者必然生成错误 WSATYPE_NOT_FOUND。

5. NSPGetServiceClassInfo

NSPGetServiceClassInfo函数的对应函数是WSAGetServiceClassInfo。它获取于一个GUID

关联的WSANAMESPACE_INFOW结构。该函数的定义如下：

```
int NSPGetServiceClassInfo (
LPGUID lpProviderId,
LPDWORD lpdwBufSize,
LPWSASERVICECLASSINFOW lpServiceClassInfo
);
```

和上一个函数一样，第一个参数仍然是提供者的 GUID。lpdwBufSize参数表明第三个参数lpService ClassInfo中包含的字节数有多少。在输入端，第三个参数是一个 WSASERVICE-CLASSINFOW结构，其中包含指定返回哪些服务类的搜索标准。这个结构中既可包含服务类名，又可包含即将返回的服务类的 GUID。如果提供者找到了符合标准的服务类，肯定会在lpServiceClassInfo参数中返回WSASERVICECLASSINFOW结构，而且更新lpdwBufSize参数，使之表明返回了多少字节。

但是，如果没有找到与搜索标准相配的服务类，调用就会失败，并设置错误WSATYPE_FOUND。另外，如果找到了匹配的服务类，但提供的缓冲区太小，提供者就应该更新lpdwBufSize，使之表明需要多少缓冲区才足够，并把错误设为 WSAEFAULT。

6. NSPSetService

NSPSetService函数的对应函数是 WSASetService，既可以用于注册服务，也可以用于从名字空间中删除服务。它的定义如下：

```
int NSPSetService (
    LPGUID lpProviderId,
    LPWSASERVICECLASSINFOW lpServiceClassInfo,
    LPWSAQUERYSETW lpqsRegInfo,
    WSAESETSERVICEOP essOperation,
    DWORD dwControlFlags
);
```

第一个参数是提供者的GUID。lpServiceClassInfo参数是该服务所属的那个WSASERVICE-CLASSINFOW结构。lpqsRegInfo参数是即将注册或删除（根据第四个参数 essOperation中指定的操作来决定）的服务。最后一个参数是 dwControlFlags，指定标志SERVICE_MULTIPLE，有了这个标志，便可修改指定的操作。

名字空间提供者首先检查所提供的服务类是否真的存在。其次，根据指定的操作，采取相应的行动。关于有效的 essOperation值的详情以及dwControlFlags标志对这些值会产生什么样的影响，请参见第 10章的10.3节“服务的注册”小节，其中着重讨论了 WSASetService。你的提供者的NSPSetService函数对这些标志进行相应的处理。

如果你的服务提供者在更新或删除不能找到的服务，就需要设置错误 WSASERVICE_NOT_FOUND。如果提供者在注册一个服务， WSAQUERYSETW结构却是无效或未完成的，提供者就会生成WSAEINVAL错误。

这个函数是最复杂的名字空间函数之一（仅次于 NSPLookupServiceNext）。提供者必须维护一种方案，使其一直保持能被注册的服务之数据。而且，还必须允许 NSPSetService函数对这一数据进行更新。

7. NSPLookupServiceBegin

NSPLookupServiceBegin函数和NSPLookupServiceNext以及NSPLookupServiceEnd有关，用于开始对你的名字空间进行查询。该函数的对应函数是 WSALookupServiceBegin，用于建

立搜索标准。它的原型如下：

```
int NSPLookupServiceBegin (
    LPGUID lpProviderId,
    LPWSAQUERYSETW lpqsRestrictions,
    LPWSASERVICECLASSINFOW lpServiceClassInfo,
    DWORD dwControlFlags,
    LPHANDLE lphLookup
);
```

和前面几个函数一样，第一个参数仍然是提供者的 GUID。lpqsRestrictions参数是定义查询参数的WSAQUERYSETW结构。第三个参数是lpServiceClassInfo，它是一个WSASERVICE-CLASSINFOW结构，其中包含了指定的服务类的简要信息，查询将在这个服务类中进行。dwControlFlags参数采用了零或若干个可对查询产生影响的标志。再次提醒大家注意，关于WSALookupServiceBegin和可以使用的标志，请参考第 10章。注意，并不是所有的标志都能对每个提供者产生影响。比如，如果你的名字空间不支持容器对象，便不必担心用于处理这些容器的标志（容器只是从概念上把服务组织在一起，至于容器内包含的究竟是什么，没有具体的限定）。最后，lphLookup参数是一个输出参数，它是一个定义特定查询的句柄。这个句柄用于WSALookupServiceNext和WSALookupServiceEnd之后的调用。

实施NSPLookupServiceBegin时，要记住这个操作不能取消，而应该尽快完成它。因此，在你打算开始网络查询时，若想成功返回，就不应该要求得到响应。

提供者本身应该保存查询参数，并将一个独一无二的句柄关联到这个查询，以备日后引用。除此以外，提供者还应该保持状态信息。我们将在下一节，深入讨论这样做的重要性和必要性。

8. NSPLookupServiceNext

一旦用NSPLookupServiceBegin开始了查询，应用程序便调用 NSPLookupServiceNext（该函数会依次调用这个名字空间提供者函数 NSPLookupServiceNext）。事实上，这个调用是在找哪些服务符合查询标准。它的定义如下：

```
int NSPAPI WSALookupServiceNext (
    HANDLE hLookup,
    DWORD dwControlFlags,
    LPDWORD lpdwBufferLength,
    LPWSAQUERYSET lpqsResults
);
```

第一个参数hLookup，是WSALookupServiceBegin函数返回的查询句柄。 dwControlFlags参数可以是标志LUP_FLUSHPREVIOUS，表明提供者应该丢弃上一个结果集，转向下一个。一般说来，一个应用程序不能为结果提供足够大的缓冲区时，便请求丢弃上一个结果。下一个参数lpdwBufferLength，表明被当作最后一个参数 lpqsResults投递的缓冲区的长度。

NSPLookupServiceNext被触发时，提供者应该查找由句柄 hLookup标志的查询参数。一旦获得了查询参数，便在开始指定的服务类中开始搜索已注册的服务。就像我们在上一小节中提到的那样，应该把查询状态保存下来。若发现了若干个匹配条目，调用进程便调用 WSALookupServiceNext若干次，每次调用，提供者都要返回一个数据集。如果没有找到匹配条目，提供者就会返回 NSPLookup错误。如果应用程序在进程中调用 WSALookupServiceNext的同时，又从另一个线程中调用了 WSALookupServiceEnd，可能会删除进程中的查询。这一

事件中，WSALookupServiceNext就会失败，并返回WSA_E_CANCELLED错误。

9. NSPLookupServiceEnd

完成查询之后，便调用NSPLookupServiceEnd函数结束查询，并释放所有名字空间资源。该函数的定义如下：

```
int NSPLookupServiceEnd ( HANDLE hLookup );
```

它只有一个参数——hLookup，该参数是一个句柄，指向行将关闭的查询。如果没有找到指定的句柄（例如是无效句柄的话），函数调用就会失败，并返回WSA_INVALID_HANDLE错误。

14.3.3 名字空间提供者示范

关于如何建立自己的名字空间及建立时需要注意的重要问题（比如保持数据的方法），我们已讲了很多。但是，要开发一个完整的名字空间提供者并不简单，接下来，为大家介绍一个名字空间示范。该示范代码对一些需要注意的问题进行了解释。另外，为使大家易于理解，尽量使其简单化。

示范提供者在配套光盘的Examples\Charpter14\NSP目录中下面三个文件中：Mynsp.h、Mynsp.cpp和Mynsp.def。范例名字空间的DLL便由这三个文件组成。除DLL外，大家还可以找到一个名字空间服务，它是一个Winsock服务器，负责处理DLL发出的请求。这个维护服务注册信息的服务器位于Mynsp.cpp文件中。另外两个文件，Nspsvc.cpp和Printobj.cpp，供DLL和服务所用，其中包含了支持例程，这些支持例程用于收集和取消服务和 DLL之间的套接字上发送的数据。数据的收集和取消稍后将具体说明。除了这两个文件，还有它们的头文件——Nspsvc.h和Printobj.h。这两个头文件中包含支持例程的函数原型。最后， Rnrcs.c是一个修改过的范例，对在我们定义的名字空间内注册和查询服务进行了解释，原来的范例参见第10章。

下面的小节将讨论怎样实施我们的名字空间。首先，大致谈谈我们所用的保持数据的方法。然后对事实上的名字空间DLL的构成和名字空间服务的安装进行说明。接下来是名字空间服务的实施。最后展示应用程序如何对我们定义的名字空间进行注册和查询。

1. 数据的保持

针对这里的名字空间，我们选用了一个独立的Winsock服务来保持名字空间信息。在DLL中实施的每一个名字空间函数中，要完成实施的话，需建立一个到该服务的连接，并对数据进行处理。为了简单起见，该服务运行于本地（在loopback地址127.0.0.1上监听）。在实际实施过程中，可通过注册或其他方法访问我们的名字空间服务的 IP地址，这样一来，应用程序在调用这个名字空间时，无论它在什么地方运行，都可连接到这个名字空间服务。比如，以DNS为例，DNS服务器的IP地址不是静态设置的，就是在DHCP请求期间获得的。

当然，编一个服务来维护这一信息并不是唯一的选择。还可在网络上维护一个文件，使其保留必要的信息。但是，这也不是最佳途径，因为性能受到了磁盘操作的影响。我们的名字空间范例存在的性能限制在于：它建立了到这个服务的 TCP连接。开发产品时，一般采用UDP之类的无连接数据报来提升系统性能。当然，要保证整体性能不受影响，还涉及到另外的编程要求，比如保证重新传输已丢弃的数据包。

2. 名字空间DLL

在实施名字空间服务之前，先来看看名字空间 DLL。每个名字空间提供者都需要一个独一无二的GUID，我们的GUID定义在Mnsp.h。除了要有一个独一无二的标识符之外，我们的名字空间还需要一个简单的整数标识符。这个标识符可用于 WSAQUERYSET结构的dwNameSpace字段中，就像大家在第10章所见的那样。我们的名字空间的GUID和标识符是这样的：

```
GUID MY_NAMESPACE_GUID = {0x55a2bd9e,0xbb30,0x11d2,
                          {0x91,0x66,0x00,0xa0,0xc9,0xa7,0x86,0xe8}
                         };
#define NS_MYNSP 66
```

这些值非常重要，因为想使用这个名字空间的应用程序必须在它们的 Winsock调用中指定这些值。当然，开发人员可以直接指定这些值，或通过 WSAEnumNameSpaceProviders调用获得（详情参见第10章）。同时，要知道一个应用程序在执行指定 NS_ALL名字提供者这样的操作时，应该在所有已安装名字提供者上执行。这一点尤为重要，因为有几个 Windows应用程序（比如说MicrosoftInternet Explorer）需要在全部已安装名字空间提供者上执行查询。因此，在测试一个名字空间提供者时，务必谨慎。编得糟糕的名字空间提供者可能导致整个系统出问题。另外， GUID和名字空间标识符的值也非常重要，因为安装名字空间提供者需要它们。

现在，来看看在Mynsp.cpp中实施的NSP函数。总的说来，这些函数非常相似，但启动和清除函数（NSPStartup和NSPCleanup）例外。启动函数只是利用我们定义的名字空间函数对NSP_ROUTINE结构进行了初始化。清除例程则没有派上用场，因为没必要进行清除。

其余的函数需要和我们的服务进行交互，对数据进行查询或注册。需要和服务进行通信时，采取下列步骤即可：

1) 与服务建立连接（利用NyNspConnect函数）。

2) 写入一个1字节的行动代码。表示即将采取的行动（服务注册、服务删除、查询等等）。

3) 汇集参数，并把它们发给服务。参数类型由操作决定。比如， NSPLookupServiceNext向服务发送了查询句柄，以便服务开始查询，而 NSPSetService则发送一个完整的WSAQUERYSET结构。

4) 读取返回的代码。一旦服务有了执行操作请求所需的参数，就会返回操作的返回代码（不管是成功，还是失败）。所以Mynsp.h文件中，定义了两个常量： MYNSP_SUCCESS和MYNSP_ERROR。

5) 如果是请求查询，而且返回的代码是成功的话，就可以读取和取消返回结果了。举个例子来说，在已找到匹配事件中， NSPLookupService返回的就是一个WSAQUERYSET结构。

由此看来，实施DLL并不复杂。NSP函数必须获得参数，并对它们进行处理。这里的函数是向名字空间服务投递信息的操作。然后，由服务执行所请求的操作。但是，我们忽略了一个非常困难的但又必须执行的操作：通过套接字收发数据。一般说来，发送数据没有什么特殊要求，但在发送整个数据结构时，情况大不一样。大多数名字空间函数都把WSAQUERYSET或WSASERVICECLASSID当作参数。而且，必须在连接了服务的套接字上收发这个对象。难就难在这些结构不是连续不断的内存块。换言之，这些结构中包含有字串指针，而别的结构可以放在内存中的任何一个地方（如图 14-4所示）。你需要做的是汇集这些分散的内存段（不管它们在哪里）并把它们逐个复制到一个独立的缓冲区内。这便是通常所说的“ 数据汇集 ”（marshaling data）。在接收数据的这一端，必须保持这个进程。也就是说，

读取的数据需要进行重组，还原成最初的结构，这些字串指针必须进行“修复”，以便指向接收端上的有效内存位置。

针对这里的名字空间提供者，我们提供了汇集和分散 WSANAMESPACEINFO和WSAQUERYSET结构的函数。这些函数位于Nspsvc.cpp文件中，供该名字空间的DLL和该名字空间服务使用（因为双方都需要能够收集和分散结构）。四个函数都很简单，我们不打算在此赘述。

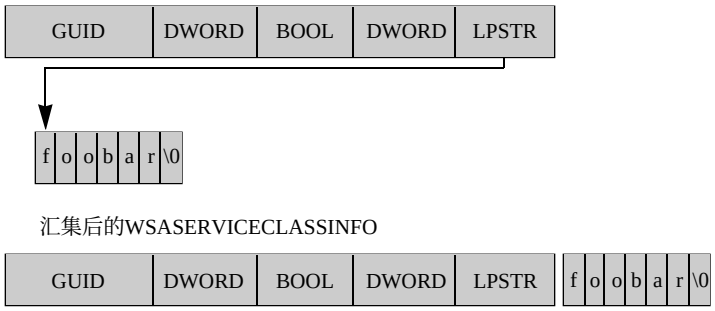

图14-4 数据的收集

3. 名字空间的安装

整个实施过程中，名字空间提供者的安装是最关键的一步。 Nspinstall.c文件是一个简单的安装程序。下面的代码便可安装我们的提供者：

```
ret = WSCInstallNameSpace(L"Custom Name Space Provider",
    L"%SystemRoot%\\System32\\Mynsp.dll", NS_MYNSP, 1,
    &MY_NAMESPACE_GUID);
if (ret == SOCKET_ERROR)
{
    printf("Failed to install name space provider: %d\n",
        WSAGetLastError());
}
```

这个调用中参数是提供者的名称、 DLL的位置、整数识别符、版本以及GUID。完成安装之后，唯一需要的就是：确保你提供的名字空间 DLL的位置正确。真正可能出现的错误便是你试图安装的那个名字空间的GUID已经被另一个提供者所用。

名字空间提供者的删除更为简单。下面的代码取自我们的安装程序，用于删除我们的提供者：

```
ret = WSCUnInstallNameSpace(&MY_NAMESPACE_GUID);
if (ret == SOCKET_ERROR)
{
    printf("Failed to remove provider: %d\n", WSAGetLastError());
}
```

4. 名字空间服务

名字空间服务才是名字提供者的精髓所在。这个服务可对所有已注册的服务类和服务实例进行追踪。名字空间DLL被用户的应用程序触发时，便连上名字空间服务，开始执行操作。这个服务很简单。在main函数内部，建立了一个监听套接字。然后，在一个循环内，名字空间DLL实例接受连接。为了简单起见，一次只处理一个连接。这样还可以防止你同步访问保持名字空间信息的那个数据结构。再次提醒大家注意，实际上提供者不会如此，因为这样会

降低性能，我们这里的目的是方便大家理解。一旦连接被接受，服务就从标识下一个行动的名字空间DLL中，读取一个单一字节。

循环内部，对行动进行解码，并把从名字空间 DLL中取得的参数投递给服务。这时起开始执行所请求的行动。这些行动并不复杂，采用的代码脉络相当清晰，并且，可从针对每个可能的行动而采取的步骤中，看出名字空间服务的原理，因此，不必赘述。但我们仍然要提到维护信息的各个结构。名字空间提供者真正关心的数据类型只有两种： WSASERVICECLASSINFO和WSAQUERYSET。大家已知道，大部分RNR函数在其参数中，都引用了一个或多个这样的结构。如此一来，我们维护的是两大通用数组（分别对应这两个结构）和各数组的计数器。

DLL请求安装服务类时，名字空间提供者的 main函数率先调用LookupServiceClass（一个支持例程，定义于 Mynspsvc.cpp中）。这个函数会遍历 ServiceClasses数组，该数组由多个WSASERVICECLASSINFO结构构成。如果没有发现具有同样 GUID的服务类，服务便返回一个错误（DLL将它翻译成WSAEALREADY）。若不然，就会在这个数组后添加新的服务类，而且dwNumServiceClasses计数器则会递增。

删除服务类的过程中（和安装服务类一样），main函数调用了LookupServiceClass。在这种情况下，如果服务类找到了，代码便将该数组的最后一个服务类移到已删除类的位置上。然后，代码递减计数器。 Winsock 2规格中，对名字空间提供者来说，没有特别指明这一点：在打算删除一个服务类时，如果已注册的服务仍然对该类进行了引用，会有什么情况发生。具体怎样处理这种情况，由你自己决定。但如果已注册服务引用了一个服务类，我们的名字空间范例不允许你将这个服务类删除。

维护WSASERVICECLASSINFO结构采取的原则同样适用于 WSAQUERY结构的追踪。这里有一个结构数组（名为 Services）和一个计数器（名为 dwNumService）。服务的添加和删除处理方式和服务类的一样。

最后，服务必须维护的信息是查询。应用程序发起一个查询时，必须维护查询期间所需的参数，并为查询分配一个独一无二的句柄（即查询句柄）。这是非常必要的，因为WSALookup ServiceNext只通过这个句柄来引用查询。必须保留的其他信息便是查询状态。也就是说，每次调用WSALookupServiceNext，都能返回一个独立的信息集。代码必须记住：Services数组中的最后一个位置，数据是在这个位置返回的。这样一来，以后对WSALookupServiceNext的调用，便从上一次调用留下的那个位置开始返回。

5. 名字空间的查询

我们的名字空间范例中，最后一部分便是 Rnrcs.c文件。它是第10章的名字注册和解析范例的改版。为了使示例尽可能浅显易懂，我们只做了少许改动。第一个改动是将代码改为列举已安装的名字空间提供者，但只返回 NS_MYNSP提供者。第二个改动是在注册一个服务时，Rnrcs.c只列举了本地 IP接口，将用它代替我们的服务地址。我们的服务提供者支持任何SOCKADDR类型的注册。最后，该范例没有针对服务注册建立服务实例，只注册了服务名。其他的则和第10章中的范例一样。

6. 运行范例

一旦已完成所有范例的编译，提供者的安装和使用就非常简单了。利用下面的命令安装提供者：

```
Nspinstall.exe install
```

当然，别忘了把Mynsp.dll复制到%SystemRoot%\System32（通过下一个命令完成）。一旦安装了名字空间，服务实例便开始运行，以供查询和注册服务之用。

Mynspsvc.exe

现在，可以利用Rnrcs.exe查询和注册服务了。表14-4展示了一些应该执行的命令以及应该遵循的顺序。这个命令序列注册两个服务，随后执行一个通配查询和一个指定查询。然后，再对这两个服务进行查询，并将它们删除。最后，我们执行了一个通配查询，用以说明服务已经被删除。

表14-4 使用示例名字空间注册和查询服务的命令序列

命 令	含 义
Rnrcs.exe -s:ASERVICE	注册服务ASERVICE
Rnrcs.exe -s:BSERVICE	注册服务BSERVICE
Rnrcs.exe -c:*	查询全部已注册的服务
Rnrcs.exe -c:BSERVICE	只查询名为BSERVICE的服务
Rnrcs.exe -c:ASERVICE -d	只查询名为ASERVICE的服务，若找到，便将其删除
Rnrcs.exe -c:BSERVICE -d	只查询名为BSERVICE的服务，若找到，便将其删除
Rnrcs.exe -c:*	查询全部已注册的服务

14.4 Winsock SPI函数的调试追踪

Winsock 2可对Ws2_32.dll中的API和SPI调用进行追踪。对开发传输服务提供者和名字空间提供者来说，这一点是非常有用的。MSDN平台SDK中有两个示例（Dt_dll和Dt_dll2）都是为追踪SPI调用而设计的。这两个示例的妙处在于，可以对它们进行修改，使之追踪自己感兴趣的API和SPI。

要使用这个调试特性，首先必须取得已通过你的目标平台核查的 Winsock 2的Ws2_32.dll版本（build号）。可在Mssdk\Bin\Debug\Winsock下的MSDN平台SDK中找到它。一旦有了核查过的版本，就可把它复制成% SystemRoot%System32下面的WS2_32.DLL，或把它放在自己的Winsock应用程序的工作目录中。后者是最佳选择，因为在以后的调试操作时，不必再去修改系统。已核查过的版本安装之后，必须编译并建立一个 MSDN Dt_dll示范。编译完成之后，就会收到一个支持DLL，名为Dt_dll.dll。然后，再把Dt_dll.dll复制到你的Winsock应用程序的工作目录中。一切搞定之后，便可开始对那些不是直接从你的应用程序中调用的 Winsock 2 API和SPI函数进行追踪了。

14.5 小结

Winsock 2 SPI使软件开发人员能够对Winsock 2的性能进行扩展，这是通过开发服务提供者来实现的。通过本章的学习，大家了解了如何开发传输服务提供者和名字空间提供者。本章结尾，还对 Winsock连网技术进行了一番总结性讨论。下一章，将介绍 Microsoft Visual Basic Winsock控件。这些控件的确使用了 Winsock，但我们没有引入任何新的 Winsock概念，只指导大家如何使用VisualBasic的Winsock控件。如果你对Visual Basic不感兴趣，可转移到本书的最后一章，该章全面讲解了“ 远程访问服务 ”（RAS），这个技术大大增强了Winsock应用程序的灵活性。