第一部分　传统网络API

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

第1章　NetBIOS

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

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

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

OSI 网络模型

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

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

图1-1 OSI网络模型

对应OSI模型，NetBIOS主要在会话和传输层发挥作用。

1.1 Microsoft NetBIOS

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

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

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

1.1.1 LANA编号

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

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

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

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

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

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

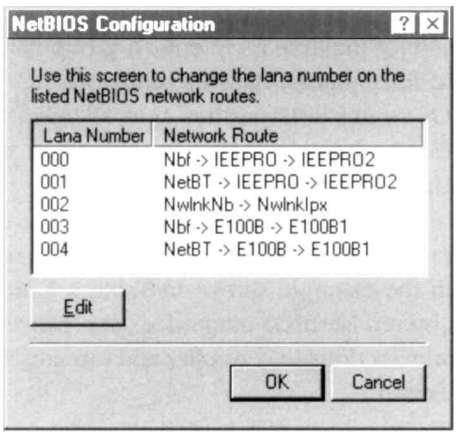

图1-2 NetBIOS配置对话框。这是一部多宿主机器，安装了两张网卡和三种传输协议：TCP/IP（NetBT）、NetBEUI（Nbf）以及IPX/SPX（NwlnkNb）

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

1.1.2 NetBIOS名字

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

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

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

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

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

表1-1 NetBIOS名字表

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

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

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

表1-2 唯一名字标识符

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

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

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

表1-3 组名标识符

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

(续)

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

1.1.3 NetBIOS特性

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

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

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

1.2 NetBIOS编程基础

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

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

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

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

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

表1-4 NCB结构成员

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

(续)

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

同步与异步

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

1.3 常规NetBIOS例程

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

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

程序清单1-1 常规NetBIOS例程(Nbcommon.c)

```
// Nbcommon.c

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

#include "nbcommon.h"

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

    retcode = Netbios(&ncb);

    return retcode;
}

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

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

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

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

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

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

    retcode = Netbios(&ncb);

    return retcode;
}

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.3.1 会话服务器：异步回调模型

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

程序清单1-2 异步回调服务器（Cbnbsvr.c）

```
// Cbnbsvr.c

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

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

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

DWORD WINAPI ClientThread(PVOID lpParam);

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

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

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

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

    return;
}

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

    FormatNetbiosName(pncb->ncb_callname, szClientName);

    while (1)
    {
        dwBufferLen = MAX_BUFFER;

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

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

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

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

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

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

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

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

//
```

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

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

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

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

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

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

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

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

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

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

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

1.3.2 会话服务器：异步事件模型

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

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

1) 列举LANA编号。

2) 重设每个LANA。

3) 为每个LANA增加服务器的名字。

4) 为每个LANA都执行（投放）一个监听命令。

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

程序清单1-3 异步事件服务器（Evnbsvr.c）

```
// Evnbsvr.c

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

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

#define MAX_SESSIONS    254
#define MAX_NAMES       254

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

    return 0;
}
```

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

异步服务器策略

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

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

1.3.3 NetBIOS会话客户机

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

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

程序清单1-4 异步事件客户机（Nbclient.c）

```
// Nbclient.c

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

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

#define MAX_SESSIONS      254
#define MAX_NAMES         254

#define MAX_BUFFER        1024

char    szServerName[NCBNAMSZ];

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

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

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

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

```
    return NRC_GOODRET;
}

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

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

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

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

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

        return 0;
    }
```

1.4 数据报的工作原理

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

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

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

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

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

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

到相同的消息。

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

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

程序清单1-5 NetBIOS数据报示例（Nbdgram.c）

```
// Nbdgram.c

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

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

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

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

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

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

    for(i = 1; i < argc; i++)
    {
        if (strlen(argv[i]) < 2)
            continue;
        if ((argv[i][0] == '-') || (argv[i][0] == '/'))
        {
            switch (tolower(argv[i][1]))
            {
                case 'n':        // Use a unique name
                    bUniqueName = TRUE;
                    if (strlen(argv[i]) > 2)
                        strcpy(szLocalName, &argv[i][3]);
                    break;
                case 'g':        // Use a group name
                    bUniqueName = FALSE;
                    if (strlen(argv[i]) > 2)
                        strcpy(szLocalName, &argv[i][3]);
                    break;
                case 's':        // Send datagrams
                    bSender = TRUE;
                    break;
                case 'c':        // # of datagrams to send or receive
                    if (strlen(argv[i]) > 2)
                        dwNumDatagrams = atoi(&argv[i][3]);
                    break;
                case 'r':        // Recipient's name for datagrams
                    if (strlen(argv[i]) > 2)
                        strcpy(szRecipientName, &argv[i][3]);
                    break;
                case 'b':        // Use broadcast datagrams
                    bBroadcast = TRUE;
                    break;
                case 'a':        // Receive datagrams on any name
                    bRecvAny = TRUE;
                    break;
                case 'l':        // Operate on this LANA only
                    bOneLana = TRUE;
                    if (strlen(argv[i]) > 2)
                        dwOneLana = atoi(&argv[i][3]);
                    break;
                case 'd':        // Delay (millisecs) between sends
                    if (strlen(argv[i]) > 2)
                        dwDelay = atoi(&argv[i][3]);
                    break;
                default:
                    printf("usage: nbdgram ?\n");
                    break;
```

```
            }
        }
    }
    return;
}

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

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

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

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

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

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

```
    ncb.ncb_length = buflen;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

    return 0;
}
```

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

表1-5 Nbdgram.c的命令

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

表1-6 Nbdgram.c的命令参数

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

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

1.5 其他NetBIOS命令

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

1.5.1 适配器状态

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

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

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

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

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

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

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

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

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

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

1.5.2 查找名字

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

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

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

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

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

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

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

1.5.3 将传送协议同LANA编号对应起来

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

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

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

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

1.6 平台问题

在下述平台上实施NetBIOS时，请留意一些特殊的限制。

1.6.1 Windows CE

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

1.6.2 Windows 9x

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

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

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

1.6.3 常规问题

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

1.7 小结

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