

第4章 命名管道

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

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

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

4.1 命名管道的实施细节

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

4.1.1 命名管道命名规范

命名管道的标识是采用UNC格式进行的：

```
\\server\Pipe\[path]name
```

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

```
\\myserver\PIPE\mypipe
```

```
\\Testserver\pipe\cooldirectory\funtest\jim
```

```
\\.\Pipe\Easynamedpipe
```

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

4.1.2 字节模式及消息模式

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

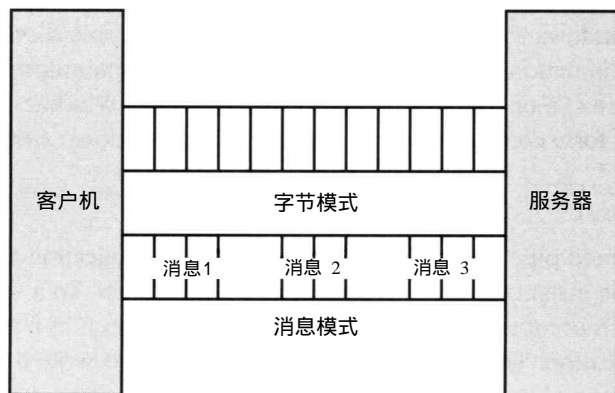


图4-1 字节模式和消息模式

4.1.3 应用程序的编译

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

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

4.1.4 错误代码

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

4.2 客户机与服务器的基础

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

4.2.1 服务器的细节

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

- 1) 使用 API 函数 CreateNamedPipe，创建一个命名管道实例句柄。
- 2) 使用 API 函数 ConnectNamedPipe，在命名管道实例上监听客户机连接请求。
- 3) 分别使用 ReadFile 和 WriteFile 这两个 API 函数，从客户机接收数据，或将数据发给客户机。
- 4) 使用 API 函数 DisconnectNamedPipe，关闭命名管道连接。
- 5) 使用 API 函数 CloseHandle，关闭命名管道实例句柄。

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

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

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

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

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

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

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

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

表 4-1 命名管道打开模式标志

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

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

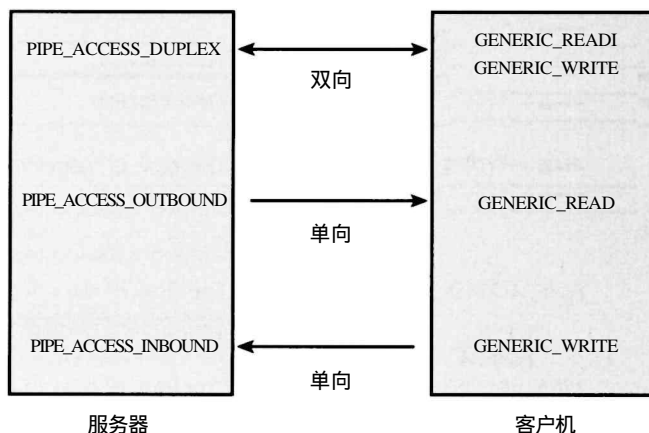


图4-2 模式标志以及数据流向

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

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

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

表4-2 命名管道的读写模式标志

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

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

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

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

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

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

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

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

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

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

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

程序清单4-1 简单的命名管道服务器

```
// Server.cpp  
  
#include <windows.h>  
#include <stdio.h>  
  
void main(void)  
{  
    HANDLE PipeHandle;  
    DWORD BytesRead;  
    CHAR buffer[256];  
    if ((PipeHandle = CreateNamedPipe("\\\\.\\Pipe\\Jim",  
        PIPE_ACCESS_DUPLEX, PIPE_TYPE_BYTE | PIPE_READMODE_BYTE, 1,  
        0, 0, 1000, NULL)) == INVALID_HANDLE_VALUE)  
    {  
        printf("CreateNamedPipe failed with error %d\n",  
            GetLastError());  
        return;  
    }  
  
    printf("Server is now running\n");  
  
    if (ConnectNamedPipe(PipeHandle, NULL) == 0)  
    {  
        printf("ConnectNamedPipe failed with error %d\n",  
            GetLastError());  
        CloseHandle(PipeHandle);  
        return;  
    }  
}
```

```

    }

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

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

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

    CloseHandle(PipeHandle);
}

```

如何构建空的授权访问控制列表 (NULL DACL)

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

```

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

```

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

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

- 1) 创建并初始化一个SECURITY_DESCRIPTOR结构，这是用API函数InitializeSecurityDescriptor来进行的。
- 2) 为SECURITY_DESCRIPTOR结构分配一个空的DACL，这是用API函数SetSecurityDescriptorDacl来进行的。

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


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

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

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

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

4.2.2 高级服务器的细节

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

1. 线程

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

程序清单 4-2 在 Win32 环境中使用线程技术开发的高级命名管道服务器

```
// Threads.cpp
```

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

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

#define NUM_PIPES 5

DWORD WINAPI PipeInstanceProc(LPVOID lpParameter);

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

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

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

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

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

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

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

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

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

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

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

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

2. 重叠式 I/O

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

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

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

程序清单4-3 使用Win32重叠式I/O技术开发的高级命名管道服务器

```
// Overlap.cpp

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

#define NUM_PIPES 5
#define BUFFER_SIZE 256

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

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

    DWORD Ret;
    DWORD Pipe;

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

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

```

    }

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

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

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

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

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

    Pipe = Ret - WAIT_OBJECT_0;

    ResetEvent(Event[Pipe]);

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

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

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

```



```

        return;
    }

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

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

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

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

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

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

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

```

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

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

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

3. 安全模拟

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

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

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

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

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

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

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

注意该函数无任何参数。

4.2.3 客户机的细节

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

- 1) 用API函数 `WaitNamedPipe`，等候一个命名管道实例可供自己使用。
- 2) 用API函数 `CreateFile`，建立与命名管道的连接。
- 3) 用API函数 `WriteFile` 和 `ReadFile`，分别向服务器发送数据，或从中接收数据。
- 4) 用API函数 `CloseHandle`，关闭打开的命名管道会话。

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

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

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

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

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

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

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

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

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

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

SECURITY_ANONYMOUS

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

SECURITY_IDENTIFICATION

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

SECURITY_IMPERSONATION

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

SECURITY_DELEGATION

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

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

SECURITY_CONTEXT_TRACKING

指出安全追踪模式是“动态”的。若未设定该标志，安全追踪模式便是“静态”的。

SECURITY_EFFECTIVE_ONLY

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

命名管道安全模拟的情况已在本章4.2.1节“服务器的细节”介绍过了。

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

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

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

程序清单 4-4 简单的命名管道客户机

```
// Client.cpp

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

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

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

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

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

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

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

    CloseHandle(PipeHandle);
}
```

4.3 其他API调用

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4.4 平台和性能问题

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

Q100291：命名管道名字的限制

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

Q119218：命名管道写操作限制在64K之内

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

Q110148：由WriteFile或ReadFile函数返回ERROR_INVALID_PARAMETER错误

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

Q180222：Windows 95的WaitNamedPipe和253号错误

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

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

Q141709：单台工作站最多只能建立49个命名管道连接

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

Q126645：从某项服务打开一个命名管道时，出现访问被拒的情况

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

4.5 小结

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