第 14 章　WinSock 网络通信开发

Windows应用程序可以实现无限的网络功能，这些功能都建立在WinSock接口的基础上。WinSock是Windows Sockets的简称，也称为Windows套接字，是微软根据BSD UNIX操作系统中流行的Berkeley套接字规范而实现的一套Windows系统环境下的网络编程接口。

本章将具体介绍在Visual C++中，基于Winsock接口进行网络通信程序开发的基础知识。

14.1　网络通信与 WinSock 基础

网络通信是指应用程序与网络中其他系统的应用程序之间进行的通讯。在介绍网络通信程序的开发之前，首先简单介绍一下网络通信和WinSock的基本概念。

14.1.1　WinSock 的基本概念

Windows环境下网络编程的规范——Windows Sockets（简称WinSock）是Windows环境下应用广泛的、开放的、支持多种协议的网络编程接口。经过不断完善，在Intel、Microsoft、Sun、SGI、Informix、Novell等公司的全力支持下，它已成为Windows网络编程事实上的标准规范。

Windows Sockets规范用于提供给应用程序开发者一套简单的API，并让各家网络软件供应商共同遵守。任何能够与WinSock兼容实现协同工作的应用程序就被认为是具有WinSock接口，并称这种应用程序为WinSock应用程序。WinSock规范定义并记录了如何使用API与Internet协议族（IPS，通常指的是TCP/IP）连接。尤其要指出的是，所有的WinSock实现都支持流套接字和数据报套接字。

应用程序通过调用WinSock提供的API实现相互之间的通讯，而WinSock又利用下层的网络通讯协议功能和操作系统调用实现实际的通讯工作。

在ISO的OSI网络7层协议中，WinSock主要负责控制数据的输入和输出，也就是传输层和网络层。它屏蔽了数据链路层和物理层，给Windows环境下的网络编程带来了巨大的变化。

14.1.2　TCP/IP 协议与 WinSock

Internet建立在TCP/IP协议基础之上，采用了TCP/IP的网络体系结构。TCP/IP不是一个简单的协议，而是一组小的、专业化的协议，包括TCP、IP、UDP、ARP、ICMP以及其他的一些被称为子协议的协议。大部分网络管理员将整组协议称为TCP/IP，有时简称为IP。其中的几个重要协议介绍如下。

- TCP（Transmission Control Protocol，传送控制协议）：提供给用户进程可靠的全双工字节流面向连接的协议，主要为用户进程提供虚电路服务，并为数据可靠传输建立检查机制。大多数网络通信程序使用TCP。
- UDP（User Datagram Protocol，用户数据报协议）：提供给用户进程的无连接协议，用于传送数据而不执行正确性检查。
- IP（Internet Protocol，网间协议）：负责主机间数据的路由和网络上数据的存储，同时为ICMP、TCP、UDP提供分组发送服务。用户进程通常不会涉及这一层。

TCP/IP协议的核心部分是传输层协议（TCP、UDP）、网络层协议（IP）和物理接口层。这三层通常是在操作系统内核中实现的，因此用户一般不涉及。

编程界面一般有两种形式：一是由内核心直接提供的系统调用；二是使用以库函数方式提供的各种函数。前者为核内实现，后者为核外实现。用户服务要通过核外的应用程序才能实现，所以要使用套接字（WinSock）来实现。TCP/IP协议核心与应用程序的关系如图14-1所示。

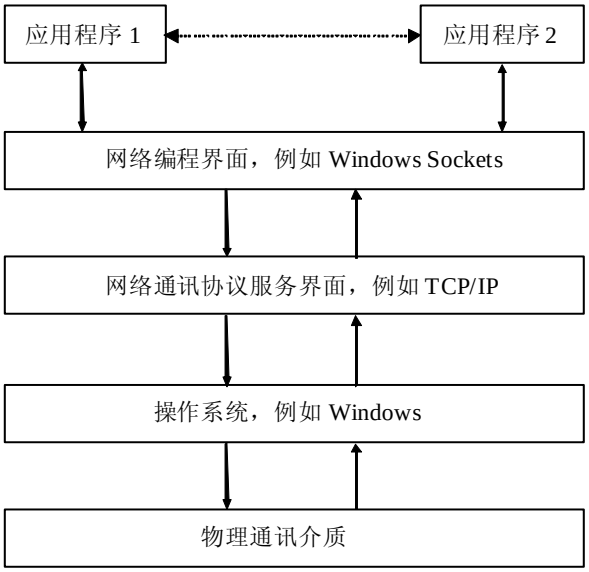

图 14-1　TCP/IP 协议核心与应用程序的关系

14.1.3　WinSock 通信与 C/S 结构

Windows Sockets通信的基础是套接字（Socket）。与文件操作类似，当要读写一个文件时，必须用一个文件对象（文件指针或文件句柄）执行这个文件。而Socket就是一个在应用程序之间用于读（接收信息）或写（发送信息）的网络对象。

Windows Sockets支持的套接字类型包括流式套接字（SOCK_STREAM）和数据报套接字（SOCK_DGRAM）。

- 流式套接字定义了一种可靠的面向连接的服务，实现了无差错无重复的顺序数据传输。

- 数据报套接字定义了一种无连接的服务。数据通过相互独立的报文进行传输，是无序的，并且不保证可靠、无差错。

对于要求精确传输数据的Windows Sockets通信程序，一般采用流式套接字。采用流式套接字通信的一个最典型的应用就是客户机/服务器（C/S）模型。这也是在TCP/IP网络中，两个进程间相互作用的主要模式。客户机/服务器模式在工作过程中采取的是主动请示方式，具体工作流程如14-2所示。

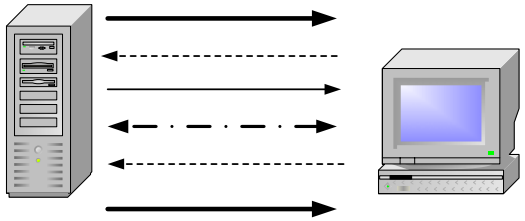

图 14-2　客户机/服务器模式工作流程

（1）首先启动服务器端，并根据请示提供相应服务，具体过程如下。

1）打开一个通信通道（Socket）并告知本地主机，准备在某一个地址上接收客户的连接请求。

2）进入监听状态，等待客户请求到达该端口。

3）接收到客户服务请求，处理该请求并发送应答信号。

4）返回第2步，等待另一客户请求。

5）关闭服务器。

（2）客户端的工作过程如下。

1）打开一个通信通道（Socket），并连接到服务器所在主机的特定端口。

2）向服务器发送服务请求报文，等待并接收应答，继续提出请求。

3）请求结束后关闭通信通道并终止。

应用程序A

开通通信通道，进入

向服务器发送连接

服务器接受连接请求

双向信息交互

关闭连接，向服务器

服务器一直处于监

服务端

14.1.4　MFC 中 WinSock 的封装类

在Visual C++ 6.0中，程序员可以直接使用Windows Sockets API函数进行WinSock网络通信程序的开发。

Windows Sockets API是以Berkeley Sockets API为模型的，提供了一个标准的API，Windows程序员可以使用它来编写网络应用程序。Windows Sockets API函数包括套接字函数、数据库函数以及针对Windows环境的的扩展函数。

MFC为套接字提供了封装类CAsyncSocket和CSocket。它们封装了Windows Sockets的API，使程序员可以用面向对象的方法调用Socket。

CAsyncSocket类在一个较低的层次上封装了Windows Sockets API。它封装了异步、非阻塞Socket的基本功能，提供了Socket的基本操作，可以用来很方便地编写常用网络通信软件。它提供的主要成员函数及其功能如表14-1所示（“*”表示为可重载函数）。

表 14-1　CAsyncSocket 类主要成员函数及其功能

函数名称	实现功能
CAsyncSocket	构造函数，构造 CAsyncSocket 对象
Create	创建套接字
Accept	接受套接字上的连接
AsyncSelect	请求对于套接字的事件通知
Bind	与套接字有关的本地地址
Close	关闭套接字
Connect	对对等套接字建立连接
IOCtr	控制套接字模式
Listen	建立套接字，侦听即将到来的连接请求
Receive	从套接字接收数据
ReceiveFrom	恢复数据报并且存储资源地址
Send	给连接套接字发送数据
SendTo	给特定目的地发送数据
ShutDown	使套接字上的 Send 和/或 Receive 调用无效
OnAccept*	通知侦听套接字，它可以通过调用 Accept，接受挂起连接请求
OnClass*	通知套接字，关闭对它的套接字连接
OnConnect*	通知连接套接字，连接尝试已经完成，无论成功或失败
OnOutOfBandData*	通知接收套接字，在套接字上有带外数据读入，通常是忙消息
OnReceive*	通知侦听套接字，通过调用 Receive 恢复数据
OnSend*	通知套接字，通过调用 Send，它可以发送数据

CSocket类由CAsyncSocket派生，是Windows Sockets API的高层抽象，提供了更高层次的功能——阻塞式的访问方式。CSocket类新提供的主要成员函数及其功能如表14-2所示。

表 14-2　CSocket 类新提供的主要成员函数及功能

函数名称	实现功能
CSocket	构造函数，构造一个 CSocket 对象
Create	创建一个套接字
IsBlocking	确定一个阻塞调用是否在进行中
FromHandle	返回一个指向 CSocket 对象的指针，给出一个套接字句柄
Attach	将一个套接字句柄与一个 CSocket 对象连接
CancelBlockingCall	取消一个当前在进行中的套接字调用
OnMessagePending*	当等待完成一个阻塞调用时调用此函数来处理悬而未决的消息

有关函数的原型及使用，在下面将结合具体的实例开发进行介绍。

14.1.5 WinSock 网络编程的常用术语

对于许多初学者来说，网络通信程序的开发非常难以入手。许多概念，诸如同步（Sync）、异步（Async）、阻塞（Block）、非阻塞（Unblock）等，初学者往往迷惑不清。这里对一些常用术语进行简单介绍。

1．套接字

套接字，也就是前面多次提到的Socket，是可以被命名和寻址的通信端点，是网络的基本构件。实际上，套接字可以理解为计算机提供的一个通信端口，通过这个端口可以与任何一个具有Socket接口的计算机通信。应用程序在网络上传输、接收的信息都通过此Socket接口来实现的。

使用中的每一个套接字都有其类型和一个与之相连的进程。WINDOWS SOCKET 1.1版本支持的套接字类型包括流套接字（SOCK_STREAM）和数据报套接字（SOCK_DGRAM）。

2．同步/异步

同步方式指的是发送方不等待接收方响应，便接着发送下个数据包的通信方式；异步方式指发送方发出数据后，等待收到接收方发回的响应，才发送下一个数据包的通信方式。

3．阻塞模式/非阻塞模式

CSocket类创建的套接字支持阻塞模式。阻塞模式简单来说就是服务器端与客户端之间的通信处于同步状态下。所谓阻塞套接字是指执行此套接字的网络调用时，直到成功才返回，否则一直阻塞在此网络调用上。例如，调用Receive函数读取网络缓冲区中的数据，如果没有数据到达，程序将一直停止在Receive这个函数调用上，直到读取到一些数据，此函数调用才返回。

在非阻塞模式下利用Socket事件的消息机制，服务器端与客户端之间的通信处于异步状态下。即执行非阻塞套接字的网络调用时，不管是否执行成功，都立即返回。例如，调用Receive函数读取网络缓冲区中的数据，不管是否读取到数据都立即返回，而不会一直停止在此函数调用上。

14.2 无连接通信开发

使用Socket进行通信主要有面向连接的流方式和无连接的数据报文方式。进行无连接通信时，通信的两台计算机就好像是把数据放在一个信封里，通过网络寄给对方。信在传送的过程中有可能会损坏，也有可能后发的先到。该种方式支持双向的数据流，但并不保证是可靠、有序、无重复的。使用Socket进行无连接通信开发时，需要创建数据报套接字 ——SOCK_DGRAM。

14.2.1 Socket 无连接通信机制

在IP中，无连接通信是通过UDP/IP协议完成的。UDP不能保证可靠的数据传输，但能将

数据发送到多个目标，或者接收多个源数据。如果一个客户端向服务器发送数据，则该数据会立刻被传输，不管服务器是否准备接收这些数据。如果服务器接收到客户端发送的数据，也不需要向客户端发送消息以确认数据被收到。UDP/IP的数据传输使用数据报，即离散的信息包。

使用无连接通信时，服务器端和客户端之间差别不大，通信双方处于完全对等状态，不存在监听与连接过程。这里就分别以发送端和接收端来区分双方，通过CSocket类实现Socket，对无连接通信的通信流程进行简单介绍。

1．接收端

对于在一个无连接的套接字上接收数据，具体流程如下。

（1）初始化WinSock的动态连接库后，首先构造CSocket套接字对象，使用Creat函数创建数据报格式的套接字。

（2）通过Bind函数将该套接字与准备接收数据的接口绑定在一起。

（3）使用Receive函数接收对方发送的数据。此时程序处于阻塞状态，直到接收到数据自动返回。

（4）接收数据完毕，使用Close函数关闭套接字。

2．发送端

对于在一个无连接的套接字上发送数据，具体流程如下。

（1）首先构造CSocket套接字对象，使用Creat函数创建数据报格式的套接字。

（2）通过Bind函数将该套接字与准备发送数据的接口绑定在一起。

（3）使用Send函数向对方发送数据。

（4）发送完毕，使用Close函数关闭套接字。

具体的通信流程如图14-3所示。

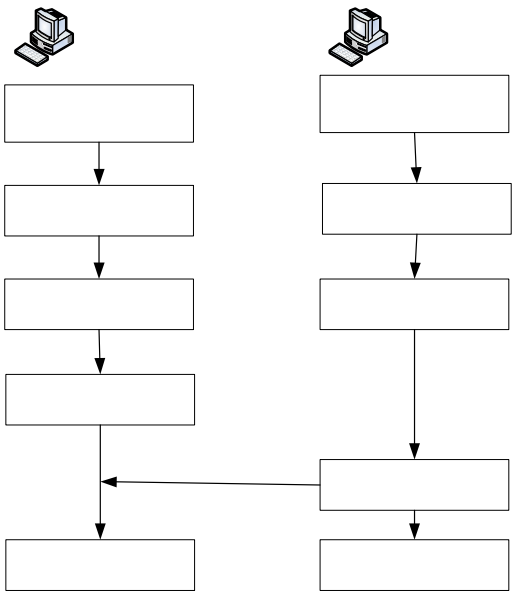

图 14-3　使用 CSocket 进行无连接通信的流程

14.2.2 主要功能函数介绍

14.1.4节已经介绍过，在MFC中，CAsyncSocket和CSocket类封装了Windows Sockets API函数。本节将结合套接字的创建和使用过程，对无连接通信中使用CAsyncSocket类的主要成员函数（CSocket类由CAsyncSocket类派生）进行详细介绍。

1．WinSock 环境的初始化

在使用WinSock MFC类之前，必须为应用程序初始化WinSock环境。其实现只要调用全局函数AfxSocketInit即可，代码如下。

```
BOOL CsocketApp::InitInstance( )
{
    if (!AfxSocketInit( ))
    {
        AfxMessageBox(IDP_SOCKETS_INIT_FAILED);
        return FALSE;
    }
    CWinApp::InitInstance( );
…
}
```

同时，在stdafx.h文件中添加如下代码。

```
#include <afxsock.h>
```

如果在使用MFC AppWizard[EXE]创建MFC工程时，在【MFC AppWizard – Step 2 of 4】窗口中勾选【Windows Sockets】复选框，则程序会自动添加上面的代码，实现WinSock的初始化。

2．创建 Socket

创建Socket，首先需要构造Socket对象，而后调用Create函数创建Socket。Create函数的调用格式如下。

```
BOOL CSocket::Create(UINT nSocketPort = 0,int nSocketType = SOCK_STREAM,LPCTSTR lpszSocketAddress = NULL)
```

各参数含义如下。

- nSocketPort：表示使用的端口号。默认为0，表示由系统自动选择，通常客户端都使用这个选择。
- nSocketType：表示使用的协议族。默认为SOCK_STREAM，表示使用面向连接的流服务；为SOCK_DGRAM，表示使用无连接的数据报服务。
- lpszSocketAddress：本地的IP地址，可以使用点分法表示，例如“127.0.0.1”。

也可以通过Bind函数设置Socket的地址和端口号，代码如下。

```
m_sock.Bind(4800,”168.0.0.1”);
```

上述代码表示该Socket对象的地址为168.0.1，端口为4800。

通过Socket提供的send()函数和Receive()函数可以实现任何类型数据的发送和接收。

3．发送和接收数据

通过Socket连接发送和接收数据比较简单，可以用Socket发送任何类型的数据，只需要一

个指向存放数据的缓冲区指针即可。发送时，缓冲区存放待发送的数据；接收时，接收的数据将拷贝到缓冲区。

（1）发送数据。可以使用Send函数通过Socket连接发送数据，函数的调用格式如下。

```
int CAsyncSocket::Send( const void* lpBuf, int nBufLen, int nFlags = 0 )
```

各参数含义如下。

- lpBuf：表示指向发送数据缓冲区的指针。如果数据为CString变量，可使用LPCTSTR操作符把CString变量作为缓冲区传送。
- nBufLen：指明缓冲区中要发送数据的长度。
- nFlags：该参数是可选的，用于控制消息的发送方式。

函数执行成功，返回发送到对方应用程序的数据总量。如果有错误产生，则函数返回SOCKET_ERROR。

（2）接收数据。Socket接收数据时，就需要调用Receive函数，函数的调用格式如下。

```
int CAsyncSocket::Receive( void* lpBuf, int nBufLen, int nFlags = 0 );
```

Receive函数的参数与Send函数基本相同。lpBuf为缓冲区指针，指明接收数据存储的位置；参数nBufLen是缓冲区的长度，指明Socket能存储多少数据；nFlags为标记位，收发双方需要指明相同的标记。

执行成功后，Receive函数也返回接收到数据的数据量；如果产生错误，则函数返回SOCKET_ERROR。

有一点需要说明，在接收数据时，最后一个字符后面最好设置一个NULL字。因为缓冲区中可能会有一些垃圾数据，如果接收的数据后面不加NULL，应用程序可能会把这些垃圾数据作为接收数据的一部分。例如下面的实现代码。

```
char *pBuf=new char[256];                          //创建接收数据缓冲区
int iRecv;                                         //接收数据总量
CString strReceive;                                //存储接收的数据
iRecv =m_sock.Receive(pBuf,256);                   //接收数据
if(iRecv ==SOCKET_ERROR);
{
                                                   //发送不成功错误处理代码
}
else
{
      pBuf[iRecv]=NULL;
      strReceive=pBuf;                             //将数据拷贝到 strReceive 变量中
……
}
```

对于无连接通信，即数据报类型的Socket，发送和接收数据还可以使用SendTo函数和ReceiveFrom函数，其功能与使用Send函数和Receive函数基本相同。

4．关闭 Socket 连接

当应用程序完成与对端的所有通信之后，就可以调用Close函数关闭这次连接。Close函数不带任何参数，调用格式如下。

```
m_sock.Close();
```

有时可能需要在Socket关闭之前就让其停止运行，这时可调用Shutdown函数。该函数调用格式如下。

```
BOOL ShutDown(int nHow = sends );
```

该函数只需要一个整形参数nHow，指明是否关闭此Socket数据的发送或接收。nHow参数的取值及含义如表14-3所示。

表 14-3　nHow 参数的取值及含义

取值	含义
0	停止通过 Socket 接收数据
1	停止通过 Socket 发送数据
2	停止通过 Socket 接收数据和发送数据

需要注意的是，调用Shutdown函数并不会关闭网络连接，也不能释放Socket所占用的任何资源，程序完成后，仍然需要调用Close函数关闭Socket。

14.2.3　无连接通信接收端的实现

无连接通信的发送方和接收方处于相同的地位，没有主次之分，常用于对数据可靠性要求不高的通信，例如实时的语音、图像传送和广播消息等。下面通过具体的实例来介绍使用CSocket类实现无连接通信。

本节将给出一个无连接通信接收端的实例，实现创建Socket从发送端固定的端口定时（间隔1.5秒）接收数据，并显示当前接收数据和本次连接所接收数据的总量。具体开发步骤如下。

1．创建 MFC 工程

启动Visual C++ 6.0，利用AppWizard创建一个基于窗口的MFC工程，工程名为UDPRevPort，在【MFC AppWizard – Step 2 of 4】窗口中勾选【Windows Sockets】复选框，如图14-4所示，其余选项采用默认设置。

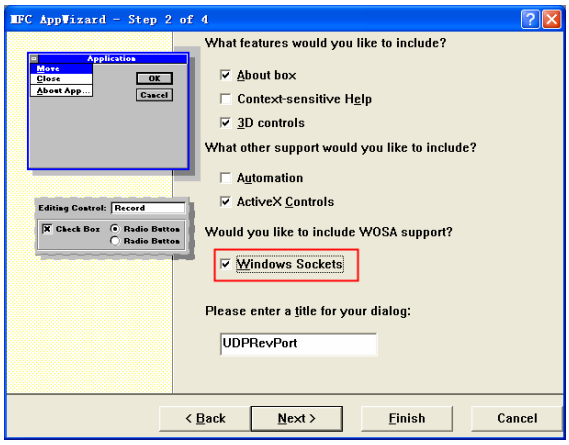

图 14-4　勾选【Windows Sockets】复选框

此时会发现，在CUDPRevPort App类的InitInstance函数中，自动添加了WinSock初始化代码，代码如下。

```
BOOL CUDPRevPortApp::InitInstance()
{
	if (!AfxSocketInit())
	{
		AfxMessageBox(IDP_SOCKETS_INIT_FAILED);
		return FALSE;
	}
	AfxEnableControlContainer();
……
```

如果用户在使用向导创建工程时，没有勾选【Windows Sockets】复选框，也可在程序中手工添加上述代码。

2．添加资源

在窗口资源编辑器中设计窗口，如图14-5所示。

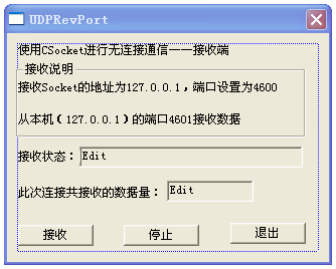

图14-5　设计窗口资源

使用ClassWizard为【接收状态】文本框添加CString类型的数据变量m_ReceiveStatus，为【此次连接共接收的数据量】文本框添加int型的数据变量m_nReCount，为按钮控件添加BN_CLICKED消息响应函数。

3．创建接收数据 Socket，进行接收数据操作

在头文件UDPRevPortDlg.h中声明一个CSocket类对象，代码如下。

```
class CUDPRevPortDlg : public CDialog
{
//Construction
public:
	CUDPRevPortDlg(CWnd* pParent = NULL);                    //standard constructor
	CSocket m_sockReceive;                                   //声明一个接收数据的 Socket 对象
……
```

在【接收】按钮的响应函数中，使用Creat函数创建套接字，使用Bind函数将其绑定到指定接收接口，启动定时器，定时接收数据。代码如下。

```
void CUDPRevPortDlg::OnReceive()
{
	//TODO: Add your control notification handler code here
	m_ReceiveStatus="没有收到数据!";
	UpdateData(false);
	if(m_sockReceive.Create(4600,SOCK_DGRAM,NULL))    //创建套接字
	{
		m_nReCount=0;
		m_sockReceive.Bind(4600,"127.0.0.1");                //绑定本地 IP
		SetTimer(1,1500,NULL);                               //创建一个定时器定时接收
		GetDlgItem(IDC_RECEIVE)->EnableWindow(false); //接收按钮无效
		GetDlgItem(IDC_STOP)->EnableWindow(true);        //停止按钮生效
	}
	else
	{
```

```
            AfxMessageBox("Socket 创建失败！");
        }
}
```

使用ClassWizard重载窗口类WM_TIMER消息响应函数OnTimer，使用CSocket类的ReceiveFrom函数定时接收数据。代码如下。

```
void CUDPRevPortDlg::OnTimer(UINT nIDEvent)
{
        //TODO: Add your message handler code here and/or call default
        char szRecv[10];

        CString m_SendIP="127.0.0.1";                                   //数据的发送端地址
        unsigned int m_SendPort=4601;                                   //数据发送端的端口
        int iRecv =m_sockReceive.ReceiveFrom(szRecv,6,m_SendIP,m_SendPort,0);
//接收 UDP 数据
        if(iRecv>0)
        {
                m_nReCount+=iRecv;                                      //接收数据的字节数
                szRecv[iRecv]=NULL;
                m_ReceiveStatus="当前接收数据："+(CString)szRecv;
        }
        else
        {
                m_ReceiveStatus="没有收到数据!";
        }
        UpdateData(false);
        CDialog::OnTimer(nIDEvent);
}
```

在【停止】按钮响应函数OnStop中，实现销毁计数器和关闭套接字操作。代码如下。

```
void CUDPRevPortDlg::OnStop()
{
        //TODO: Add your control notification handler code here
        KillTimer(1);                                                   //关闭定时器
        m_sockReceive.Close();                                          //关闭套接字
        m_ReceiveStatus="停止接收数据!";
        UpdateData(false);
        GetDlgItem(IDC_RECEIVE)->EnableWindow(true);                    //接收按钮生效
        GetDlgItem(IDC_STOP)->EnableWindow(false);                      //停止按钮无效
}
```

图14-6　程序运行结果

4．编译运行

编译运行程序，单击【接收】按钮，程序便会创建接收Socket，通过其提供的ReadFrom函数每隔1.5秒从本机（127.0.0.1）的4601端口接收数据。由于该端口此时并没有向接收Socket发送数据，所以程序就处于阻塞状态，即ReadFrom函数一直不返回。窗口就处于“挂起”状态，不能进行任何操作，只能通过任务管理器将其关闭。运行结果如图14-6所示。

14.2.4　无连接通信发送端的实现

要在一个无连接的套接字上发送数据，最简单的方法就是创建一个套接字，然后调用SendTo函数向指定接口发送数据即可。

本节将给出一个无连接通信发送端的实例，实现通过创建发送Socket，向上节实例的接收端口定时（间隔1.5秒）发送数据，并显示当前发送的数据和所发送的数据总量。具体开发步骤如下。

1．创建 MFC 工程

启动Visual C++ 6.0，利用MFC AppWizard[EXE]创建一个新的MFC工程，工程名为UDPSendPort，在【MFC AppWizard – Step 2 of 4】窗口中勾选【Windows Sockets】复选框。

2．添加资源

在窗口资源编辑器中设计窗口，如图14-7所示。

使用ClassWizard为【发送状态】文本框添加CString类型的数据变量m_strSendStatus，为【发送的数据总量】文本框添加int型的数据变量m_nSendCount，为按钮控件添加BN_CLICKED消息响应函数。

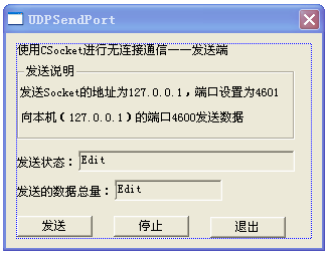

图14-7　设计窗口资源

3．创建发送数据 Socket，进行发送数据操作

在头文件UDPSendPortDlg.h中，声明一个CSocket类对象，代码如下。

```
class CUDPSendPortDlg : public CDialog
{
//Construction
public:
	CUDPSendPortDlg(CWnd* pParent = NULL);   //standard constructor
	CSocket m_sockSend;                      //声明一个发送 Socket 对象
……
```

在【发送】按钮的响应函数中，使用Creat函数创建套接字，使用Bind函数将其绑定到指定发送接口（与上节例程的接收位置相对应），启动定时器，定时发送数据，代码如下。

```
void CUDPSendPortDlg::OnSend()
{
	//TODO: Add your control notification handler code here
	if(m_sockSend.Create(4601,SOCK_DGRAM,NULL))          //创建套接字
	{
		m_sockSend.Bind(4601,"127.0.0.1");               //绑定本地套接字
		SetTimer(1,1500,NULL);                           //创建一个定时器
		GetDlgItem(IDC_SEND)->EnableWindow(false);       //发送按钮无效
		GetDlgItem(IDC_STOP)->EnableWindow(true);        //停止按钮生效
	}
	else
	{
		AfxMessageBox("Socket 创建失败！");
	}
}
```

使用ClassWizard重载窗口类WM_TIMER消息响应函数OnTimer，使用CSocket类的SendTo函数定时发送数据。代码如下。

```
void CUDPSendPortDlg::OnTimer(UINT nIDEvent)
{
	//TODO: Add your message handler code here and/or call default
	static iIndex=0;                                          //静态变量，用于计数
	char szSend[10];
	sprintf(szSend,"%06d",iIndex++);                          //发送的数据置入缓冲区
	CString m_ReceiveIP="127.0.0.1";                          //数据的接收端地址
	unsigned int m_RecievePort=4600;                          //数据接收端的端口
	int iSend= m_sockSend.SendTo(szSend,6,m_RecievePort,m_ReceiveIP,0);
                                                              //发送 UDP 数据
	m_strSendStatus="正在发送数据："+(CString)szSend;
                                                              //当前发送的数据
	m_nSendCount+=iSend;                                      //记录发送数据的字节数
	UpdateData(false);
	CDialog::OnTimer(nIDEvent);
}
```

在【停止】按钮响应函数OnStop中，实现销毁计数器和关闭套接字操作。代码如下。

```
void CUDPSendPortDlg::OnStop()
{
	//TODO: Add your control notification handler code here
	KillTimer(1);                                             //销毁定时器
	m_sockSend.Close();                                       //关闭套接字
	GetDlgItem(IDC_SEND)->EnableWindow(true);                 //发送按钮生效
	GetDlgItem(IDC_STOP)->EnableWindow(false);                //停止按钮无效
	m_strSendStatus="停止发送数据";
	UpdateData(false);
}
```

4．编译运行

编译运行程序，单击【发送】按钮，程序即会创建发送Socket，向指定接口定时发送数据，而不管对方是否开始接收或者是否正确接收。此时，若打开上节创建的接收端例程，单击【接收】按钮，即可接收到其发送的数据。运行结果如图14-8所示。

图 14-8　程序运行结果

14.3　面向连接通信开发

与无连接通信不同，面向连接通信要求两个通信的应用程序之间首先建立一条连接链

路，而后数据才能被正确接收和发送。面向连接通信的特点是通信可靠，对数据有重发和校验机制，通常用来做数据文件的传输，例如FTP、Telnet等。使用Socket进行面向连接通信开发时，需要创建流式套接字（SOCK_STREAM）。

14.3.1　Socket 面向连接通信机制

在IP中，面向连接的通信是通过TCP/IP协议完成的，TCP提供两个计算机间可靠无误的数据传输。应用程序使用TCP进行通信时，在源计算机和目标计算机之间会建立起一个虚拟连接。建立连接之后，计算机之间便能以双向字节流的方式进行数据交换。

与无连接通信不同，面向连接通信中，必须有一方扮演服务器的角色，等待另一方（客户端）的连接请求。服务器方需要首先创建一个监听套接字，在此套接字上等待连接。当连接建立后，会生成一个新的套接字用于与客户端通信。下面以CSocket类实现Socket通信为例，对面向连接通信的通信流程进行简单介绍。

1．服务器端

面向连接通信服务器端的具体实现流程如下。

（1）创建监听Socket对象。初始化WinSock的动态连接库后，需要在服务器端建立一个监听的Socket。即首先构造一个CSocket对象，而后通过调用Create函数创建一个流套接字。

（2）绑定监听Socket的端口。使用Bind函数为服务器端定义的监听Socket指定一个地址及端口，这样客户端才知道要连接哪一个地址的哪个端口。

（3）进入监听状态。使用Listen函数使服务器端的Socket进入监听状态，并设定可以建立的最大连接数。

（4）接受用户的连接请求。服务器进入到监听模式后，便已经做好了可以接受客户端连接的准备，下面就可以通过Accept函数来接受用户的连接请求。

（5）与客户端进行通信。Accept函数执行后，将新建一个通信Socket与客户端的Socket相通。原先的监听Socket继续维持监听状态，等待他人的连接要求。通信Socket就可以通过Read函数和Write函数与客户端进行通信。

（6）关闭服务。当要关闭服务器时，使用Close函数关闭监听套接字和通信套接字即可。

2．客户端

面向连接通信中，客户端网络连接的创建相对服务器端要简单，其具体实现流程如下。

（1）创建客户端Socket。初始化WinSock的动态连接库后，首先构造CSocket套接字对象，使用Creat函数创建套接字。与服务器端的Socket不同，客户端的Socket可以调用Bind函数，由自己来指定IP地址及端口号，也可以不调用Bind，而由Winsock来自动设定IP地址和端口。

（2）提出连接请求。客户端的Socket使用Connect函数来提出与服务器端的Socket建立连接申请。

（3）与服务器通信。服务器端接受客户端Socket的连接请求后，便可以通过Read函数和Write函数与服务器端进行通信。

（4）断开连接。使用Close函数关闭客户端Socket即可实现断开与服务器的连接。

具体的通信流程如图14-9所示。

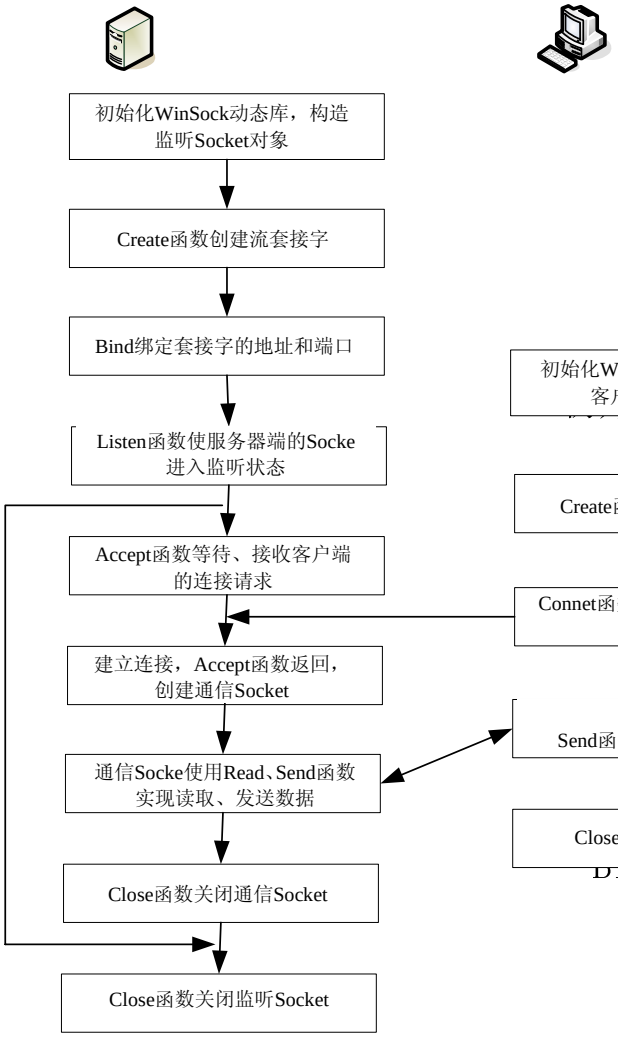

图 14-9　使用 CSocket 进行面向连接通信的流程

14.3.2　主要功能函数

上节介绍了面向连接通信程序开发的基本流程和使用的Socket类相关的成员函数。其中大部分函数在14.2.2节已有详细介绍，本节将对面向连接通信的Listen函数进行详细介绍。

1．监听函数 Listen

服务器端的监听Socke通过调用Listen成员函数，使Socket处于监听状态，监听对方（客户端）的连接请求。Listen函数的调用格式如下。

```
BOOL CAsyncSocket::Listen( int nConnectionBacklog = 5 );
```

其中，参数nConnectionBacklog为Socket可以同时接受的连接数，默认值为5，也是最大值。函数成功执行则返回一个非0值。

Listen函数的典型调用格式如下。

```
if(m_sock.Listen())
{
        ┆                                                    //Socket 监听成功后的代码
}
else
 ┆                                                           //错误处理代码
```

2．连接函数 Connect

客户端Socket通过调用Connect函数连接服务器，其调用格式如下。

```
BOOL CAsyncSocket::Connect( LPCTSTR lpszHostAddress, UINT nHostPort );
```

两个参数分别为要连接的计算机（服务器）的IP地址和端口号。

Connect函数的典型调用格式如下。

```
if (m_sock.Connect("168.0.0.1",4800));                      //发起连接请求
{
        ┆                                                    //Socket 连接成功后的代码
}else
 ┆                                                          //错误处理代码
```

当执行CSocket类的Connect函数时，在连接成功之前不会返回，即程序处于阻塞状态，只有成功连接或者出了故障不能进行连接才会返回。

3．接受连接函数 Acceptt

服务器端Socke通过调用Accept函数接受客户端连接请求，其调用格式如下。

```
BOOL CAsyncSocket::Accept( CAsyncSocket& rConnectedSocket, SOCKADDR* lpSockAddr = NULL, int* lpSockAddrLen = NULL )
```

各参数含义如下。

- rConnectedSocket：为一个新的套接字，用于与连接方通信。
- lpSockAddr：为SOCKADDR结构的指针，用于记录连接方（客户端）的IP地址信息。
- lpSockAddrLen：用于存储lpSockAddr信息的长度。

当连接建立后，一个新的套接字将被创建，该套接字用于与对方的应用程序连接。

Accept函数的调用格式如下。

```
CSocket m_sock2;                                            //声明一个 Socket 对象
if(m_sock.Accept(m_sock2));                                 //等待连接请求
{
      ……                                                    //连接请求成功后的代码
}else
……                                                          //连接请求失败后的代码
```

当使用CSocket类时，直到收到连接请求并接受后，Accept函数才返回。

14.3.3　面向连接通信服务器端的实现

与无连接通信的实例相对应，本节将给出一个面向连接通信服务器端的实例。实例将实现在固定端口创建监听套接字，等待客户端的连接。当有客户端连接时，通过新建的通信套

接字向客户套接字定时（间隔1.5秒）发送数据，并显示当前的发送状态和发送的数据量。具体开发步骤如下。

1．创建 MFC 工程

启动Visual C++ 6.0，利用AppWizard创建一个基于窗口的MFC工程，工程名为TCPSeverPor，在【MFC AppWizard – Step 2 of 4】窗口中勾选【Windows Sockets】复选框。

图14-10 设计窗口资源

2．添加资源

在窗口资源编辑器中设计窗口，如图14-10所示。

使用ClassWizard为【当前状态】文本框添加CString类型的数据变量m_strSendStatus，为【已经发送的数据量】文本框添加int型的数据变量m_nSendCount，为按钮控件添加BN_CLICKED消息响应函数。

3．创建监听 Socket 和通信 Socket

在头文件TCPSeverPortDlg.h中声明监听Socket和通信Socket对象，代码如下。

```
class CTCPSeverPortDlg : public CDialog
{
//Construction
public:
	CTCPSeverPortDlg(CWnd* pParent = NULL);  //standard constructor
	CSocket m_sockListen;                                    //声明一个监听套接字
	CSocket m_sockSend;                                      //声明一个通信套接字
……
```

在【启动】按钮的响应函数OnStar中，创建并绑定监听套接字，使套接字处于监听状态，等待客户程序的连接。当有客户程序连接后，创建一个定时器，通过新创建的通信套接字定时向客户套接字发送数据。代码如下。

```
void CTCPSeverPortDlg::OnStar()
{
	//TODO: Add your control notification handler code here
	if(m_sockListen.Create(5200,SOCK_STREAM,NULL))          //创建监听套接字
	{
		GetDlgItem(IDC_STAR)->EnableWindow(false);          //开始按钮无效
		GetDlgItem(IDC_STOP)->EnableWindow(true);           //停止按钮生效
		m_sockListen.Bind(5600,"127.0.0.1");                //绑定本地套接口
		if(m_sockListen.Listen(1))
		{
			m_strSendStatus="服务器处于监听状态";
			UpdateData(false);
			//等待连接请求，m_sockSend 为发送套接字，用于通信
			if(m_sockListen.Accept(m_sockSend))             //阻塞，当有连接进入时才返回
			{
			SetTimer(1,1500,NULL);                          //创建一个定时器定时发送
			}
		}
	}
	else
	{
		AfxMessageBox("监听 Socket 创建失败！");
```

```
    }
}
```

使用ClassWizard重载窗口类WM_TIMER消息响应函数OnTimer。代码如下。

```
void CTCPSeverPortDlg::OnTimer(UINT nIDEvent)
{
    //TODO: Add your message handler code here and/or call default
    static iIndex=0;                                    //静态变量，用于计数
    char szSend[10];
    sprintf(szSend,"%06d",iIndex++);                    //把发送的数据置入缓冲区
    //发送 TCP 数据
    int iSend= m_sockSend.Send(szSend,6,0);
    if(iSend>0)
    {
    m_nSendCount+=iSend;                                //发送字节数
    CString str=szSend;
    m_strSendStatus="正在发送数据"+str;
    UpdateData(false);
    }
    else
    {
        m_strSendStatus="发送数据失败";
        UpdateData(false);
    }
    CDialog::OnTimer(nIDEvent);
}
```

在【停止】按钮的响应函数OnStop中，实现销毁计数器和关闭套接字操作。代码如下。

```
void CTCPSeverPortDlg::OnStop()
{
    //TODO: Add your control notification handler code here
    KillTimer(1);                                       //销毁定时器
    m_sockSend.Close();                                 //关闭发送套接字
    m_sockListen.Close();                               //关闭监听套接字
    m_strSendStatus="服务器停止发送数据";
    UpdateData(false);
    GetDlgItem(IDC_STAR)->EnableWindow(true);           //开始按钮生效
    GetDlgItem(IDC_STOP)->EnableWindow(false);          //停止按钮无效
}
```

4．编译运行

编译运行程序，单击【启动】按钮，程序便会创建监听Socket，通过Accept函数等待客端户程序的连接。但由于此时没有客户端程序连接监听Socket绑定的接口，所示程序就处于阻塞状态，即Accept函数一直不返回。窗口就处于“挂起”状态，不能进行任何操作，只能通过任务管理器将其关闭。运行结果如图14-11所示。

图14-11　程序运行结果

14.3.4　面向连接通信客户端的实现

本节将给出一个面向连接通信客户端的实例。实例通过创建Socket连接上节创建的服务器接口，定时（间隔1.5秒）接收服务器发送的数据。具体开发步骤如下。

1．创建 MFC 工程

启动Visual C++ 6.0，利用AppWizard创建一个基于窗口的MFC工程，工程名为UDPReceivePort，在【MFC AppWizard – Step 2 of 4】窗口中勾选【Windows Sockets】复选框。

2．添加资源

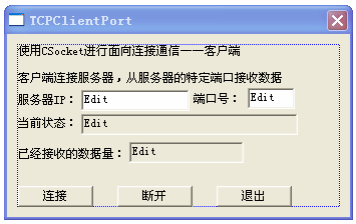

图14-12　设计窗口资源

在窗口资源编辑器中设计窗口，如图14-12所示。

使用ClassWizard为【服务器IP】文本框添加CString类型数据变量m_strIP，为【端口号】文本框添加short类型数据变量m_nPort，为【当前状态】文本框添加CString类型数据变量m_strReceiveStatus，为【已经接受的数据量】文本框添加int类型数据变量m_nReceiveCount，为按钮控件添加BN_CLICKED消息响应函数。

3．创建客户端 Socket，连接服务器并接收数据

在头文件TCPClientPortDlg.h中声明客户端Socket对象。代码如下。

```
class CTCPClientPortDlg : public CDialog
{
//Construction
public:
	CTCPClientPortDlg(CWnd* pParent = NULL);   //standard constructor
	CSocket m_sockReceive;                     //客户端 CSocket 对象
……
```

在【连接】按钮的响应函数OnConnect中，创建客户端Socket，连接服务器，创建定时器，定时接收数据。代码如下。

```
void CTCPClientPortDlg::OnConnect()
{
	//TODO: Add your control notification handler code here
	UpdateData();
	if(m_sockReceive.Create())                                    //创建套接字
	{
		//发起连接请求
		if(m_sockReceive.Connect(m_strIP,m_nPort))                //连接服务器，成功后返回
		{
		SetTimer(1,1500,NULL);                                    //创建一个定时器定时接收
		m_strReceiveStatus="成功连接服务器!";
		UpdateData(false);
		GetDlgItem(IDC_CONNECT)->EnableWindow(false);      //连接按钮无效
		GetDlgItem(IDC_DISCONNECT)->EnableWindow(true);   //断开按钮生效
		}
		else
		{
			m_strReceiveStatus="连接服务器失败!";
			UpdateData(false);
		}
	}
	else
	{
		AfxMessageBox("Socket 创建失败！");
	}
```

```
}
```

使用ClassWizard重载窗口类WM_TIMER消息响应函数OnTimer。代码如下。

```
void CTCPClientPortDlg::OnTimer(UINT nIDEvent)
{
	//TODO: Add your message handler code here and/or call default
	char szRecv[10];
	int iRecv =m_sockReceive.Receive(szRecv,6,0);                    //接收 TCP 数据
	if(iRecv>=0)
	{
		szRecv[iRecv]=NULL;
		m_strReceiveStatus=szRecv;                                   //记录接收数据
		m_nReceiveCount+=iRecv;                                      //接收字节数
		m_strReceiveStatus="当前接收的数据："+m_strReceiveStatus;
	}
	else                                                             //如果没有接收到任何数据
	{
		m_strReceiveStatus="没有收到数据!";
	}
	UpdateData(false);
	CDialog::OnTimer(nIDEvent);
}
```

在【断开】按钮的响应函数OnDisconnect中，实现销毁计数器和关闭套接字操作。代码如下。

```
void CTCPClientPortDlg::OnDisconnect()
{
	//TODO: Add your control notification handler code here
	KillTimer(1);                                                    //关闭定时器
	m_sockReceive.Close();                                           //关闭套接字
	m_strReceiveStatus="断开与服务器的连接!";
	UpdateData(false);
	GetDlgItem(IDC_CONNECT)->EnableWindow(true);                     //连接按钮生效
	GetDlgItem(IDC_DISCONNECT)->EnableWindow(false);                 //断开连接按钮无效
}
```

4．编译运行

编译运行程序，输入服务器的IP地址和端口号，单击【连接】按钮，程序即会创建客户端Socket，连接服务器，并处于阻塞状态。若先启动上节创建的服务器，再单击【连接】按钮，即可实现服务器与客户端的连接，进行收发数据操作。运行结果如图14-13所示。

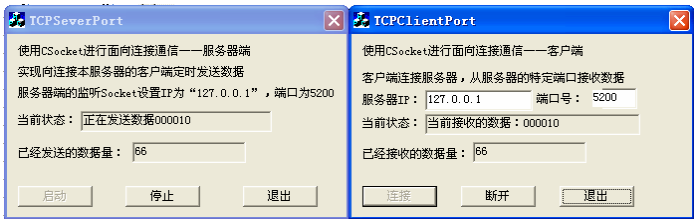

图 14-13　程序运行结果

可见，CSocket类支持阻塞模式，所以当执行Accept和Connect函数时，会阻塞自身的运行，挂起线程，直到建立连接为止。这种方式简化了面向连接网络程序的开发，但其弊端也

是显而易见的。

14.4 Socket 非阻塞模式及开发

当有多个客户端Socket与服务器端Socket连接和通信时，服务器端采用阻塞模式就显得不适合了。此时应该采用非阻塞模式，利用Socket事件的消息机制来接受多个客户端Socket的连接请求，并进行通信。本节将重点介绍Socket事件的处理机制，并通过一个网络聊天室的开发实例进行详解。

14.4.1 CSocket 阻塞模式

在14.3.3节介绍的面向连接通信服务器端程序开发中，监听套接字采用的是CSocket的阻塞模式。很显然，如果没有来自客户端Socket的连接请求，Socket就会不断调用Accept函数产生循环阻塞，直到有来自客户端Socket的连接请求才解除阻塞。阻塞解除，表示服务器端Socket和客户端Socket已成功连接，服务器端与客户端彼此可相互调用Send和Receive方法开始通信。

如果服务器端用于监听的Socket在主线程中运行，这将导致主线程的阻塞。因此，当有多个客户连接服务器时，需要分别为其创建一个新的线程，以运行Socket服务，如图14-14所示，但采用这种方式将会增加程序设计的复杂性。

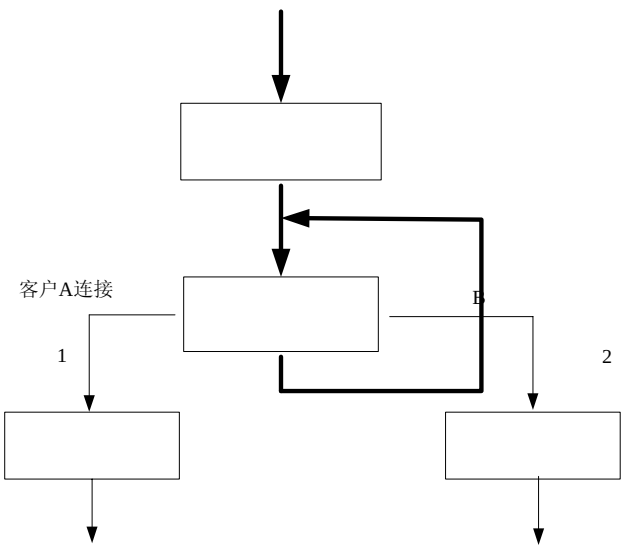

图 14-14　阻塞模式的通信过程

主线程

CSocket对

14.4.2　CSocket 非阻塞模式——事件处理

非阻塞模式是指服务器端与客户端之间的通信处于异步状态下，可以通过Socket事件的消息机制来实现。

使用CSocket（或者CAsyncSocket）类创建的客户端Socket与服务器端Socket进行连接通信时，会触发许多事件。例如，当客户端Socket成功连接服务器时，会触发FD_CONNECT事件，而服务器端接受用户连接，会触发FD_ACCEPT事件等。为了响应这些事件，在CAsyncSocket类中定义了一系列可重载的函数，例如OnConnect、OnAccept等。各事件及其对应的响应函数如表14-4所示。

表 14-4　CAsyncSocket 类的事件及其可重载事件函数

事件标记	响应函数	事件描述
FD_ACCEPT	OnAccept	通知侦听套接字，对方程序的连接请求正在等待被接受
FD_CLOSE	OnClose	表示连接的另一端应用程序已经关闭它的 Socket 或者连接已丢失，收到此通知的 Socket 应该关闭
FD_CONNECT	OnConnect	通知连接套接字，对方的连接已经完成，可以通过 Socket 发送或接收消息
FD_OOB	OnOutOfBandData	表示收到带外数据，带外数据在一个逻辑上独立的通道上发送，用户紧急数据通信，不是常规通信的一部分。Send 和 Receive 函数的第三个参数用户带外数据的发送和接收
FD_READ	OnReceive	表示 Socket 连接的数据已经接收到，可调用 Receive 函数接收
FD_WRITE	OnSend	表示通过 Socket 已经准备好发送数据，连接完成即可调用此函数

这样，用户就可以从CSocket类（或者CAsyncSocket）派生一个新类，通过重载这些Socket事件的响应函数，实现非阻塞模式的面向连接通信。例如，14.3.3节给出的面向连接的服务器端例程，采用的是CSocket的阻塞模式，其【启动】按钮响应函数如下。

```
void CTCPSeverPortDlg::OnStar()
{
    //TODO: Add your control notification handler code here
    if(m_sockListen.Create(5200,SOCK_STREAM,NULL))          //创建监听套接字
    {
    ……
        m_sockListen.Bind(5600,"127.0.0.1");                //绑定本地套接口
        if(m_sockListen.Listen(1))
        {
            m_strSendStatus="服务器处于监听状态";
            UpdateData(false);
            //等待连接请求，m_sockSend 为发送套接字，用于通信
            if(m_sockListen.Accept(m_sockSend))//阻塞，当有连接进入时才返回
            {
            ……
```

Accept函数阻塞了主线程，等待用户的连接。如果这里采用Socket事件机制，便可以实现非阻塞方式。简单来说，就是从CSocket类派生新的监听Socket类CListenSocket，在其中重

载OnAccept函数，在OnAccept函数中实现创建客户套接字与客户端进行通信。此时，【启动】按钮的响应函数可表示如下。

```
CListenSocket   m_ListenSocket                                  //重载的监听套接字对象
void CTCPSeverPortDlg::OnStar()                                 //启动服务
{
    if(m_ListenSocket.Create(5200,SOCK_STREAM,NULL))            //创建监听套接字
    {
    ……
        m_ListenSocket.Bind(5600,"127.0.0.1");                  //绑定本地套接口
        if(m_ListenSocket.Listen(1))
        {
            m_strSendStatus="服务器处于监听状态";
            UpdateData(false);
        }
}
```

由于没有调用Accept函数，所以主线程没有阻塞。而当有客户连接m_ListenSocket套接字时，服务器端便会触发FD_ACCEPT事件，调用OnAccept函数，在其中创建新的通信Socket与客户端Socket进行通信。当然，如果有多个客户连接服务器，便可以在OnAccept函数中构造多个Socket，分别与不同的客户端Socket进行通信。

实际上，与阻塞模式相比，非阻塞模式本质上是通过在服务器端构造不同Socket来处理不同客户端的访问。

在实际的网络通信软件开发中，异步非阻塞套接字是用到最多的。平常所说的C/S（客户端/服务器）结构的软件就是异步非阻塞模式的。下面将通过具体的实例对使用CSocket非阻塞模式实现网络编程进行具体介绍。

14.4.3 非阻塞模式服务器端的实现

下面将开发一个基于C/S结构的网络聊天室，利用Socket事件机制，即CSocket非阻塞模式来实现。

这里将实现服务器端的开发。该服务器的基本功能是开启服务端口，等待客户端的连接。当有客户端连接后，接收客户发送的文本信息，并将该信息转发给所有与之连接的客户端。

基本思路是从CSocket类派生CListenSocket（用于创建监听Socket）和CCommSocket（用于创建通信Socket），并创建一个CCommSocketList类，用于记录所有的CCommSocket对象。在CListenSocket类中，重载OnAccept函数。在该函数中，实现创建CCommSocket对象与客户端Socket进行通信，从而实现非阻塞模式。具体开发步骤如下。

1．创建 MFC 工程

启动Visual C++ 6.0，利用AppWizard创建一个基于窗口的MFC工程，工程名为UnBlockChatServer，在【MFC AppWizard – Step 2 of 4】窗口中勾选【Windows Sockets】复选框。

2．添加资源

在窗口资源编辑器中设计窗口，如图14-15所示。

使用ClassWizard为【服务器状态】文本框添加CString类型的数据变量m_strStatus，为按钮控件添加BN_CLICKED消息响应函数。

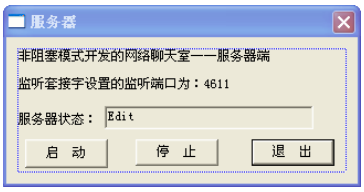

图14-15　设计窗口资源

3．创建 CCommSocket 类和 CCommSocketList 类

选择【Insert】→【New Class】命令，在弹出的【New Class】窗口中，以CSocket为基类，创建MFC Class类型的类CCommSocket。采用同样的方法，创建Generic Class类型的新类CCommSocketList。

CCommSocket类的头文件实现代码如下。

```
class CCommSocketList;
class CCommSocket : public CSocket
{
public:
//Operations
public:
        CCommSocket(CCommSocketList *);                    //带参数的/构造函数
        virtual ~CCommSocket();
//Overrides
public:

        CCommSocketList *List;                             //通信套接字链表
        CCommSocket * Front;                               //前一个套接字
        CCommSocket * Next;                                //后一个套接字
……
};
```

在构造函数中，实现变量的初始化，代码如下。

```
CCommSocket::CCommSocket(CCommSocketList *temp)
{
        Front=Next=0;
        List=temp;
}
```

使用ClassWizard，在CCommSocket类中重载CSocket类的OnReceive函数，实现与客户端Socket的通信，代码如下。

```
void CCommSocket::OnReceive(int nErrorCode)
{
        //TODO: Add your specialized code here and/or call the base class
        List->Sends(this);                                 //调用 CClientSocketList 的 send 函数
        CSocket::OnReceive(nErrorCode);
}
```

类CCommSocketList的头文件声明如下。

```
Class CCommSocketList
{
public:
        CCommSocketList();
        virtual ~CCommSocketList();
        BOOL Sends(CCommSocket *);                         //发送数据
        BOOL Add(CCommSocket *);                           //添加链表
        CCommSocket * Head;                                //链表头
};
```

其中，主要定义了两个函数Sends和Add，分别用于向客户端Socket发送数据和向链表中添加CCommSocket对象。具体实现代码如下。

```
BOOL CCommSocketList::Add(CCommSocket*add)
{
   CCommSocket*tmp=Head;                                    //客户套接字
   if (!Head)                                               //如果列表为空
   {
      Head=add;                                             //add 为表头
      return TRUE;
   }
   while (tmp->Next) tmp=tmp->Next;
   tmp->Next=add;                                           //添加列表元素
   return TRUE;
}
BOOL CCommSocketList::Sends(CCommSocket*tmp)
{
   char buff[1000];                                         //分配内存
   int n;
   CCommSocket*curr=Head;                                   //客户套接字
   n=tmp->Receive(buff,1000);                  //调用 CCommSocket 的 Receive 函数
   buff[n]=0;
   while (curr)
   {
      curr->Send(buff,n);                                   //向各个客户端发信息
      curr=curr->Next;
   }
   return TRUE;
}
```

4．创建 CListenSocket 类

采用同样的方法，以CSocket类为基类，派生新类CListenSocket，用于实现服务器端的监听套接字。使用ClassWizard，在CListenSocket类中重载CSocket类的OnAccept函数。当服务器端收到连接请求后，即调用该函数，在其中创建通信套接字，与客户端进行通信。OnAccept函数的实现代码如下。

```
void CListenSocket::OnAccept(int nErrorCode)
{
      //TODO: Add your specialized code here and/or call the base class
      CCommSocket *tmp=new CCommSocket(&CClientList);       //声明通信套接字
      Accept(*tmp);                                         //创建通信 Socket
      CClientList.Add(tmp);                                 //添加到通信套接字列表中
      CSocket::OnAccept(nErrorCode);
}
```

其中，CClientList在头文件中声明，代码如下。

```
public:
      CCommSocketList CClientList;                          //通信 socket 列表
```

5．实现服务器的启动与停止

【开始】按钮和【停止】按钮响应函数的实现代码如下。

```
void CUnBlockChatServerDlg::OnStart()
{
```

```
        //TODO: Add your control notification handler code here
        GetDlgItem(IDC_START)->EnableWindow(false);          //使启动按钮无效
        m_ListenSocket.Create(4611);                         //创建监听套接字端口为 4611
        m_ListenSocket.Listen(5);                            //开始监听
        m_strStatus="进入监听状态";
        UpdateData(false);
        GetDlgItem(IDC_STOP)->EnableWindow(true);            //将停止按钮激活
}
void CUnBlockChatServerDlg::OnStop()
{
        //TODO: Add your control notification handler code here
        GetDlgItem(IDC_STOP)->EnableWindow(FALSE);           //使停止按钮无效
        m_ListenSocket.Close();                              //关闭监听套接字
        m_strStatus="服务器关闭";
        UpdateData(false);
        GetDlgItem(IDC_START)->EnableWindow(TRUE);           //将启动按钮激活
}
```

其中，m_ListenSocket在头文件中定义，代码如下。

```
public:
        CListenSocket m_ListenSocket;
```

6．编译运行

此时编译运行程序，单击【启动】按钮，服务器即开启，进入监听状态。与前面的阻塞模式不同，由于实例采用的是非阻塞模式，所以主线程并没有阻塞，此时还可以在该窗口中进行其他操作。运行结果如图14-16所示。

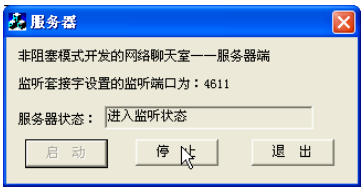

图14-16　程序运行结果

14.4.4　非阻塞模式客户端的实现

本节将介绍非阻塞模式聊天室客户端程序的开发。其基本的开发思想是，创建CSocket类的派生类CServerSocket，在其中重载OnReceive函数；当客户连接到服务器，服务器端有数据发送到本端时，将执行OnReceive函数，实现接收服务器端Socket发送的数据，并在窗口显示。具体开发步骤如下。

1．创建 MFC 工程

启动Visual C++ 6.0，利用AppWizard创建一个基于窗口的MFC工程，工程名为UnBlockChatClient，在【MFC AppWizard – Step 2 of 4】窗口中勾选【Windows Sockets】复选框。

2．添加资源

在窗口资源编辑器中设计窗口，如图14-17所示。选择【Insert】→【Resource】命令，向工程中添加一个窗口，实现登录窗口，其资源设计如图14-18所示。

使用ClassWizard为登录窗口创建关联类CConnectedDlg。

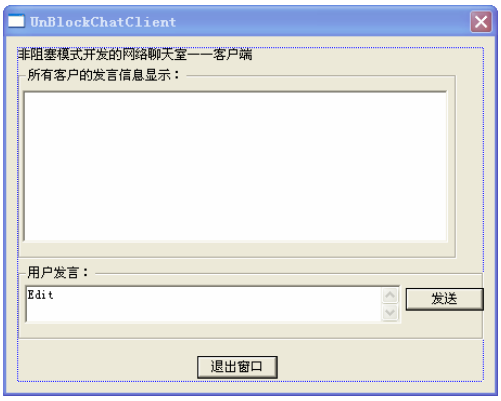

图 14-17　设计主窗口资源

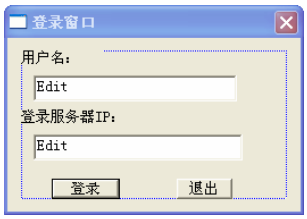

图 14-18　设计登录窗口资源

3．创建 CServerSocket 类

以CSocket类为基类，派生新类CServerSocket，通过该类创建的Socket对象实现连接服务器，向服务器发送消息，并从服务器接收信息，显示在窗口中的列表框中。

在CServerSocket类的头文件中，声明相关成员变量和函数，代码如下。

```
lass CUnBlockChatClientDlg;                          //主窗口类
class CServerSocket : public CSocket
{
//Attributes
public:
        CUnBlockChatClientDlg * myDlg;               //主窗口指针
        BOOL SetDlg(CUnBlockChatClientDlg *tmp);
        CString UserName;                            //用户名
……
```

SetDlg函数用户实现在Socket对象中获取主窗口的指针，代码如下。

```
BOOL CServerSocket::SetDlg(CUnBlockChatClientDlg*tmp)
{
        myDlg=tmp;
        return true;
}
```

使用ClassWizard，在CServerSocket类中重载CSocket类的OnReceive函数，调用主窗口的成员函数GetMessage，接收服务器的信息，并添加到主窗口的列表框中，代码如下。

```
void CServerSocket::OnReceive(int nErrorCode)
{
        //TODO: Add your specialized code here and/or call the base class
        myDlg->GetMessage();                         //接收服务器信息
        CSocket::OnReceive(nErrorCode);
}
```

主窗口类CUnBlockChatClientDlg中，GetMessage函数的实现代码如下。

```
BOOL CUnBlockChatClientDlg::GetMessage()
{
        char buff[1000];
        int count;
        count=myServerSocket->Receive(buff,1000);    //接收服务器消息
```

```
    buff[count]=0;
    m_IDC_LIST_CHATBOX_CONTROL.AddString(buff);      //添加到列表框
    return true;
}
```

而主窗口的myServerSocket又将如何获取要接收数据的Socket对象指针呢？其具体实现就是在CUnBlockChatClientDlg类中声明CServerSocket对象指针，并将该对象指针作为CUnBlockChatClientDlg类构造函数的一个参数。代码如下。

```
class CUnBlockChatClientDlg : public CDialog
{
//Construction
public:
    CServerSocket * myServerSocket;                  //声明 CServerSocket 对象指针
    BOOL GetMessage();
    CUnBlockChatClientDlg(CServerSocket *,CWnd* pParent = NULL);
                                                     //带参数的构造函数
……
```

构造函数的实现如下。

```
CUnBlockChatClientDlg::CUnBlockChatClientDlg(CServerSocket *tmp,CWnd* pParent /*=NULL*/)
    : CDialog(CUnBlockChatClientDlg::IDD, pParent)
{
    //{{AFX_DATA_INIT(CUnBlockChatClientDlg)
    m_IDC_EDIT_MESSAGE = _T("");
    //}}AFX_DATA_INIT
    //Note that LoadIcon does not require a subsequent DestroyIcon in Win32
    m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
    myServerSocket=tmp;                              //获取 CServerSocket 对象
}
```

4．在 CConnectedDlg 类中实现连接服务器

在【登录】按钮的响应函数OnConnect中，使用CServerSocket创建Socket对象，连接服务器。连接成功后，退出登录窗口，代码如下。

```
void CConnectedDlg::OnConnect()
{
    //TODO: Add your control notification handler code here
    UpdateData(TRUE);
    char*address;
    int n;
    if (!myServerSocket->Create())                   //创建服务套接字
    {
        myServerSocket->Close();
        AfxMessageBox("Socket 创建错误！！");
        return;
    }
    n=m_strAddress.GetLength();
    address=new char(n+1);                           //获取服务器的 IP 地址
    sprintf(address,"%s",m_strAddress.GetBuffer(n));
    address[n]=0;
    if (!myServerSocket->Connect(address,4611))      //连接服务器
    {
        myServerSocket->Close();
        AfxMessageBox("网络连接错误！请重新检查服务器地址的填写是否正确？");
        return;
```

```
    }
    myServerSocket->UserName=m_strUserName;
    CDialog::OnOK();                                    //退出窗口
}
```

连接成功后，如果服务器端有发送到本端的数据，激发FD_READ事件，调用CConnectedDlg重载的OnReceive函数，读取数据，并显示在主窗口的列表框中。

5．在主窗口类 CUnBlockChatClientDlg 中实现向服务器发送数据

```
void CUnBlockChatClientDlg::OnButtonSend()
{
    //TODO: Add your control notification handler code here
    int n;
    char message[1000];
    UpdateData(TRUE);
    //在消息前添加用户名
    m_IDC_EDIT_MESSAGE=myServerSocket->UserName+": "+m_IDC_EDIT_MESSAGE;
    n=m_IDC_EDIT_MESSAGE.GetLength();
    sprintf(message,"%s",m_IDC_EDIT_MESSAGE.GetBuffer(n));
    message[n]=0;
    if (!myServerSocket->Send(message,n+1))             //向服务器发送数据
    {
        AfxMessageBox("网络传输错误!");
    }
}
```

至此，与通信相关的连接服务器、发送数据、接收数据的功能均实现了。为了在主窗口出现之前先启动登录窗口窗口，需要在CUnBlockChatClientApp::InitInstance中添加代码。代码如下。

```
CServerSocket curSocket;
BOOL CUnBlockChatClientApp::InitInstance()
{
    if (!AfxSocketInit())
    {
        AfxMessageBox(IDP_SOCKETS_INIT_FAILED);
        return FALSE;
    }
……
    CConnectedDlg connectdlg(&curSocket);               //构造登录窗口
    if (connectdlg.DoModal()==IDCANCEL)                 //显示登录窗口
        return FALSE;
    CUnBlockChatClientDlg dlg(&curSocket);
    m_pMainWnd = &dlg;
    curSocket.SetDlg(&dlg);
    int nResponse = dlg.DoModal();
    ……
}
```

6．编译运行

此时编译运行程序，在登录窗口中设置用户名和服务器IP地址后，单击【登录】按钮，即可连接服务器（前提是服务器端已启动）。用户可以向服务器发送信息，并实时接收服务器发送的数据。因为采用了非阻塞模式，虽然可以实时接收服务器发送的数据，却并没有阻塞主线程。用户A和用户B同时与服务器通信的演示结果如图14-19所示。

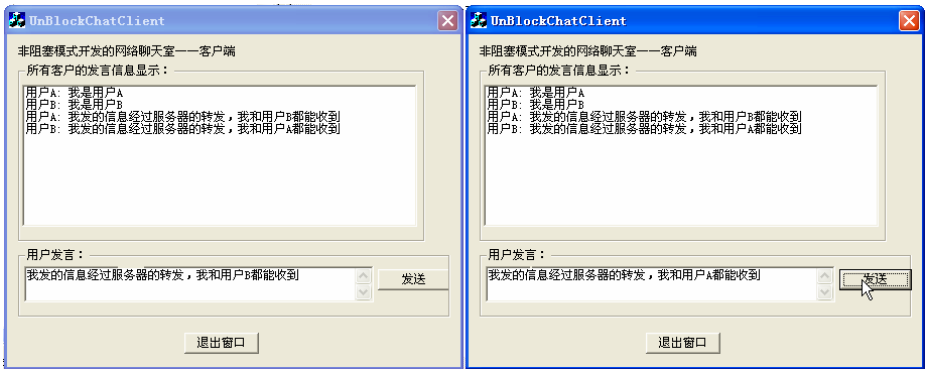

图 14-19　程序运行结果

14.5　小结

WinSock是Windows下应用广泛的、开放的、支持多种协议的网络编程接口。通过使用WinSock接口，可以开发各种网络功能。MFC为WinSock接口提供了封装类——CAsyncSocket和CSocket。在Visual C++中，通过它们可以方便灵活地开发网络应用程序。

本章主要介绍了使用CSocket进行基本网络通信——基于UDP协议的无连接通信和基于TCP协议的面向连接通信的基本流程和具体实现。本章最后详细介绍了使用CSocket的非阻塞模式进行C/S结构的网络应用程序开发，并给出了具体的实例。在学习本章时，读者需要重点注意阻塞模式和非阻塞模式的概念及实现方法，这是开发大型网络程序的基础，也是本章的难点。