5 MFC 套接字网络编程

Windows 应用程序可以有无限的网络功能，都是建立在 WinSock 接口的基础上。前面已经介绍了有关 Socket 的基础知识及基本开发技术。在实际 Windows 网络通信程序开发中，使用 Visual C++提供的 MFC 类库是十分方便和有效的。在本章和下一章中，将简单介绍如何使用 MFC 进行相关网络通信程序的开发。

5.1 MFC 基础与网络类库

MFC 的英文全称是 Microsoft Fundation Classes，即微软的基本类库。它封装了大部分 API 函数，并提供了一个应用程序框架，简化和标准了 Windows 程序设计，所以用 MFC 编写 Windows 应用程序也称为标准 Windows 程序设计。

5.1.1 MFC 基础

MFC 实际上可以理解为用来编写 Windows 应用程序的 C++类集。MFC 约有 200 个类，提供了 Windows 应用程序框架和创建应用程序的组件。它提供了大量的基类供程序员根据不同的应用环境进行扩充，同时允许在编程过程中自定义和扩展应用程序中的类，它还具有较好的移植性，可移植于众多的平台。

MFC 类库采用单一继承结构，从根类 CObject 层层派生出绝大多数 MFC 中的类，如图 5.1 所示。

基类 CObject 的最基本功能包括支持序列化（Serialization）、运行时（Run-time）类的信息获取，提供特定操作符，完成对象的建立与删除。

有关 MFC 的更多详细信息可参阅其他 MFC 程序设计的书籍、资料，这里不再详细介绍。

5.1.2 MFC 中的网络开发相关类

在 MFC 类库中，与网络开发相关的类主要有 WinSock 类、WinInet 类库和 ISAPI 类。下面对其分别进行简单介绍。

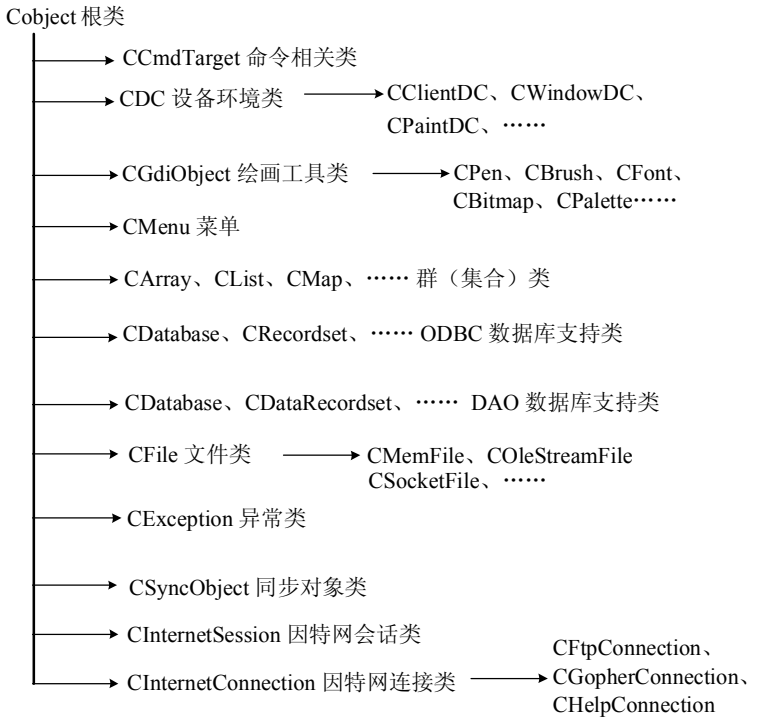

图 5.1　CObject 派生类层次示意图

1. WinSock 类

Microsoft Visual C++的 MFC 类库提供了两个 Socket 类：CAsyncSocket 类（封装了 Windows Sockets API）和 CSocket 类（从 CAsyncSocket 类派生的高级抽象，支持同步操作）。用户可以单独使用 CSocket 类，但通常和另一个相关的类 CSocketFile 一起使用。CSocketFile 类用于提供 Sockets 传送和接收数据的 CFile 对象。WinSock 类主要包含的类及其作用如表 5.1 所示。

表 5.1　WinSock 类

类名	功能简述
CAsyncSocket	实现对 WinSock API 的封装，层次较低
CSocket	派生于 CAsyncSocket 的高层次的抽象，支持同步操作
CSocketFile	为 WinSock 提供 CFile 接口

2. WinInet 类

为了简化 Internet 编程，MFC 提供了 WinInet 类库，它封装了 Win32 Internet（WinInet）和 ActiveX 技术，使 Internet 编程更加容易。表 5.2 列出了 MFC 提供的 13 个 WinInet 类。

表 5.2　WinInet 类及其说明

类名	说明
CInternetSession	创建并初始化一个或多个同时的 Internet 会话。如果需要，还可描述与代理服务器的连接

续表

类名	说明
CInternetConnection	管理应用程序与 Internet 服务器（HTTP 服务器、FTP 服务器、Gopher 服务器）建立的连接
CInternetFile	提供了用 Internet 协议访问远程服务器文件系统的方法
CHttpConnection	控制对 HTTP 服务器的连接
CHttpFile	提供查找和读取 HTTP 服务器上的文件功能
CGopherFile	提供查找和读取 Gopher 服务器上的文件功能
CFtpConnection	控制对 FTP 服务器的连接
CGopherConnection	控制对 Gopher 服务器的连接
CFileFind	进行本地的 Internet 文件查找
CFtpFileFind	在 FTP 服务器上查找 Internet 文件
CGopherFileFind	在 Gopher 服务器上查找 Internet 文件
CGopherLocator	从 Gopher 站点获取 Gopher 位标（locator），并提供给 CGopherFileFind 用来定位
CInternetException	描述与 Internet 操作有关的异常情况

3. ISAPI 类

ISAPI 类描述了 Internet 服务器的接口，例如运行 IIS（Internet Information Server，互联网信息服务）的 Windows NT 服务器就是一个 ISAPI 服务器，而 HTTP 过滤器则用于处理服务器请求。MFC 提供 5 个封装了 ISAPI 的类来创建和处理 Internet 服务器扩展和过滤器，如表 5.3 所示。

表 5.3　MFC 提供的封装了 ISAPI 的类

ISAPI 类名	说明
CHttpServer	创建和管理服务器扩展 DLL 的对象，也称为 Internet 服务器应用程序（ISA）。ISA 通过 EXTENSION_CONTROL_BLOCK 结构与服务器进行通信来处理客户端命令。每个客户端命令与 CHttpServer 对象的成员函数相对应，该命令由服务器的分析映射宏进行分析。另外，每个 ISAPI DLL 只能有一个 CHttpServer 对象，该对象包含由服务器提供的 EXTENSION_CONTROL_BLOCK 结构，并为每个单独的客户端请求创建一个新的 CHttpServerContext 对象
CHttpServerContext	由 CHttpServer 创建的对象，用于处理单一客户端对服务器对象的单一请求。服务器可能正在处理多个并发请求，因此可以将多个活动的 CHttpServerContext 对象与特定的 CHttpServer 实例关联
CHtmlStream	利用该类来管理客户机和服务器间的数据。无论何时，当 CHttpServer 类需要在客户机和服务器之间传递信息时，都会创建该类的对象
CHttpFilter	该对象可以管理过滤器的客户端数据。与 CHttpServer 对象一样，每个 ISAPI DLL 只能有一个 CHttpFilter 对象
CHttpFilterContext	CHttpFilter 对象将为每个活动的服务器事件创建一个 CHttpFilterContext 对象。该类表示在特定会话中由单个客户机形成的单个事件

CHttpServer 和 CHttpFilter 可以彼此独立存在，也就是说不必将 CHttpServer 与 CHttpFilter 一起使用，反之亦然。二者也可分别在单独的 ISAPI DLL 中。

本章主要介绍使用 MFC 提供的 WinSock 类进行基本的网络应用程序开发，WinInet 类和 ISAPI 类的使用将在下章讨论。

5.2 CAsyncSocket 类及其开发

前面已经介绍过，MFC 提供了两个 WinSocket 类：CAsyncSocket 和 CSocket，它们封装了 Windows Sockets 的 API，从而程序员可以用面向对象的方法调用 Socket。本节将简单介绍 CAsyncSocket 类及其开发。

5.2.1 CAsyncSocket 类

CAsyncSocket 类在较低的级别上封装了 Windows Sockets API，该类使用户可以使用面向对象的方式来进行 Sockets 编程，并且可以非常方便地调用其他 MFC 对象。

CAsyncSocket 类几乎是一一对应地封装了 Windows Sockets 的 API，因此具有直接调用 Sockets API 的灵活性。同时该类要求编程者对 Sockets 编程有较深入的了解。

CAsyncSocket 类提供的主要成员函数及其功能如表 5.4 所示。

表 5.4 CAsyncSocket 类的主要成员函数及其功能

函数	功能
构造函数	
CAsyncSocket	构造 CAsyncSocket 对象
Create	创建套接字
属性函数	
Attach	对 CAsyncSocket 对象附加套接字句柄
Detach	从 CAsyncSocket 对象除去套接字句柄
FromHandle	返回 CAsyncSocket 对象的指针，给出套接字句柄
GetLastError	获得上一次运行失败的状态
GetPeerName	获得与套接字连接的对等套接字的地址
GetSockName	获得套接字的本地名
GetSockOpt	获得套接字选项
SetSockOpt	设置套接字选项
操作函数	
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

续表

函数	功能
操作函数	
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
可重载函数	
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
数据成员	
m_hSocket	指定附加在此 CAsyncSocket 对象上的 Socket 句柄

5.2.2 CAsyncSocket 类的编程模式

在一个 MFC 应用程序中，要想轻松处理多个网络协议而又不牺牲灵活性时，可以考虑使用 CAsyncSocket 类，它的效率比 CSocket 类要高。

使用 CAsyncSocket 类的编程模式可简单表示如下：

（1）构造一个 CAsyncSocket 对象，并用这个对象的 Create 成员函数产生一个 Socket 句柄。

构造 CAsyncSocket 对象可以在栈上产生，也可以在堆上产生。在栈上创建一个流式套接字的代码可表示如下：

```
CAsyncSocket m_sock;                          //声明一个 Socket 对象
if(m_sock.Create(5900,SOCK_ STREAM,"127.0.0.1"))
{
    …                                         //Socket 创建成功后的代码
}else
…                                             //错误处理代码
```

而在堆上实现在指定端口号产生一个数据报套接字的代码可表示如下：

```
CAsyncSocket*pSocket=newCAsyncSocket
int nPort=5800                                //指定端口
pSocket->Create(nPort,SOCK-DGRAM)
```

Create 函数的原型如下：

```
BOOL CAsyncSocket::Create( UINT nSocketPort = 0, int nSocketType = SOCK_STREAM, long lEvent = FD_READ |
```

```
FD_WRITE | FD_OOB | FD_ACCEPT | FD_CONNECT | FD_CLOSE, LPCTSTR lpszSocketAddress = NULL );
```

各参数含义如下：

- nSocketPort：为使用的端口号，默认为 0，表示由系统自动选择，通常在客户端都使用这个选择。
- nSocketType：为使用的协议簇，默认为 SOCK_STREAM，表示使用有连接的流服务；为 SOCK_DGRAM 时，表示使用无连接的数据报服务。
- lEvent：通过指明 lEvent 所包含的标记来确定需要异步处理的事件，对于指明的相关事件的相关函数调用都不需要等待完成后才返回。各事件及其对应的可重载函数如表 5.5 所示。

表 5.5 CAsyncSocket 类可重载事件通知函数

事件	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 3 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，可以调用 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

- lpszSocketAddress：为本地的 IP 地址，可以使用点分法表示，如 127.0.0.1。

如果准备用 Socket 连接另一个应用程序，作为客户程序，不必给 Create 函数传送任何参数。而如果 Socket 作为服务程序，准备监听另一个应用程序的连接请求，则至少需要传送一个端口给 Create 函数。

（2）进行监听/连接操作。

如果是客户端程序，需要调用 CAsyncSocket::Connect()成员函数连接到服务端；而如果是服务端程序，用需要调用 AsyncSocket::Listen()成员函数开始监听，一旦接收到连接请求，则调用 CAsyncSocket::Accept()成员函数与客户端建立连接。

服务套接字监听函数 Listen()的原型如下：

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

该函数作为等待连接方时需要指明同时可以接受的连接数 nConnectionBacklog。

Listen()函数的典型调用方式如下：

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

服务套接字接收函数 Accept()的原型如下：

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

各参数含义如下：

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

该函数作为等待连接方将等待连接建立，当连接建立后，一个新的套接字 rConnectedSocket 将被创建，该套接字将会被用于通信。

CAsyncSocket::Accept()成员函数要用一个新的并且是空的 CAsyncSocket 对象作为它的参数，这里所说的“空的”指的是这个新对象还没有调用 Create()成员函数。

Accept()函数的典型应用代码如下：

```
CAsyncSocket 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
```

客户套接字连接函数 Connect()的原型如下：

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

该函数实现连接方（客户端）发起与等待连接方（服务端）的连接，需要指明对方（服务端）的 IP 地址和端口号。

Connect()函数的典型调用代码如下：

```
if (m_sock.Connect("127.0.0.1",5800));          //发起连接请求
{
    …                                           //Socket 连接成功后的代码
}else
…                                               //错误处理代码
```

（3）进行数据通信。

可以通过调用 CAsyncSocket 类的 Receive()、ReceiveFrom()成员函数接收数据；调用 Send()和 SendTo()成员函数发送数据。

Send()函数的原型如下：

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

各参数含义如下：

- lpBuf：指向发送数据缓冲区的指针，如果数据为 CString 变量，可以使用 LPCTSTR 操作符把 CString 变量作为缓冲区传送。
- nBufLen：指明缓冲区要发送数据的长度。
- 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，函数返回 SOCKET_ERROR。

典型的 Send 函数使用代码如下：

```
int iLen;                                          //要发送数据长度
int iSend;                                         //发送到对方应用程序的数据总量
iLen=str.GetLength();                              //str 为要发送的 CString 类型数据
iSend = m_sock.Send(LPCTSTR(str), iLen);           //发送 TCP 数据
if(iSend ==SOCKET_ERROR);
```

```
{
        //发送不成功的错误处理代码
}
else
{
        //发送成功的处理代码
}
```

（4）接收数据。

Socket 接收数据时，需要调用 Receive 函数。Receive 函数的原型如下：

```
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，指示 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 函数也返回接收到的数据的数据量；如果有错误产生，函数返回 SOCKET_ERROR。

典型的 Receive 函数使用代码如下：

```
char *pBuf=new char[1024];                  //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,1024);           //接收 TCP 数据
if(iRecv ==SOCKET_ERROR);
{
    //发送不成功的错误处理代码
}
else
{
    pBuf[iRecv]=NULL;
    strReceive=pBuf;                        //将数据拷贝到 strReceive 变量中
    …
}
```

在接收数据时，最后一个字符后面最好设置一个 NULL 字符，正如上面的代码。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。

如果采用无连接数据报通信方式，发送和接收数据使用 SendTo 和 ReceiveFrom 函数，其功能和使用与 Send 和 Receive 函数基本相同。

（5）销毁 CAsyncSocket 对象。

如果是在栈上产生的 CAsyncSocket 对象，则对象超出定义的范围时自动被析构；如果是在堆上产生，也就是用了 new 这个操作符，则必须使用 delete 操作符销毁 CAsyncSocket 对象。

5.2.3 CAsyncSocket 类的异步机制

从类名就知道，CAsyncSocket 是一个异步非阻塞 Socket 封装类。从上节的介绍可知，CAsyncSocket::Create()函数有一个参数指明了想要处理的 Socket 事件。用户关心的事件被指定以后，这个 Socket 默认就被用作了异步方式。

而 CAsyncSocket 内部到底是如何将事件交给用户的呢？CAsyncSocket 的 Create()函数执行后，

除了创建了一个 Socket 对象外，还创建了一个 CSocketWnd 窗口对象，并使用 WSAAsyncSelect()将创建的 Socket 对象与该窗口对象关联，以让该窗口对象处理来自 Socket 的事件（消息），然而 CSocketWnd 收到 Socket 事件之后只是简单地回调 CAsyncSocket::OnReceive()、CAsyncSocket::OnSend()、CAsyncSocket::OnAccept()、CAsyncSocket::OnConnect()等虚函数，所以 CAsyncSocket 的派生类只需要在这些虚函数里添加发送和接收的代码。

除了在 Create()函数中指定 Socket 需要处理的事件外，用户还可以通过调用 AsyncSelect()函数来指定。AsyncSelect()函数用组合标记定义激发哪些事件。AsyncSelect 函数的原型如下：

```
BOOL AsyncSelect(long lEvent = FD_READ | FD_WRITE | FD_OOB | FD_ACCEPT | FD_CONNECT | FD_CLOSE );
```

调用 AsyncSelect 函数的典型代码可表示如下：

```
if(m_sock.AsyncSelect (FD_READ| FD_CONNECT)== SOCKET_ERROR);
{
    …                          //错误处理
}else
…                              //成功后的代码
```

默认情况下，AsyncSelect 函数激发所有的事件函数。如果想要关闭激发所有的事件，可以采用如下代码：

```
m_sock..AsyncSelect (0);
```

需要注意的是，客户端在使用 CAsyncSocket::Connect()连接服务器时，往往会返回一个 WSAEWOULDBLOCK 错误（其他的某些函数调用也如此），实际上这不应该算作一个错误，它是 Socket 提醒我们，由于使用了非阻塞 Socket 方式，所以连接操作需要时间，不能瞬间建立。解决方案是，用户在 Connect()调用之后等待 CAsyncSocket::OnConnect()事件被触发。在 CAsyncSocket::OnConnect()被调用之后即可知道 Socket 连接是成功了还是失败了。

类似地，Send()如果返回 WSAEWOULDBLOCK 错误，用户可以在 OnSend()处等待；Receive()如果返回 WSAEWOULDBLOCK 错误，用户可以在 OnReceive()处等待，依此类推。

下面将通过一个具体的实例简单介绍使用 CAsyncSocket 类开发 C/S 结构网络服务程序的具体实现。

5.2.4 使用 CAsyncSocket 类实现信息转发器（服务端）

下面使用 CAsyncSocket 类的事件机制实现一个简单的 C/S 结构的网络信息转发器。该转发器实现的功能是当客户端与服务器建立连接后，可以向服务器端发送信息，而服务器收到消息后，将消息回发至客户端。本节将实现服务端的开发，其运行界面如图 5.2 所示。

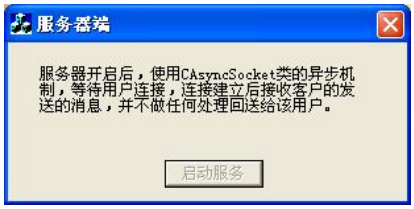

图 5.2 服务器端的运行界面

实例使用 Visual C++ 6.0 开发，使用 AppWizard 创建一个基于对话框的 MFC 工程，工程名为 ntTransInfoServer。在 MFC AppWizard Step 2 时，需要选中 Windows Sockets 复选项，如图 5.3 所示。

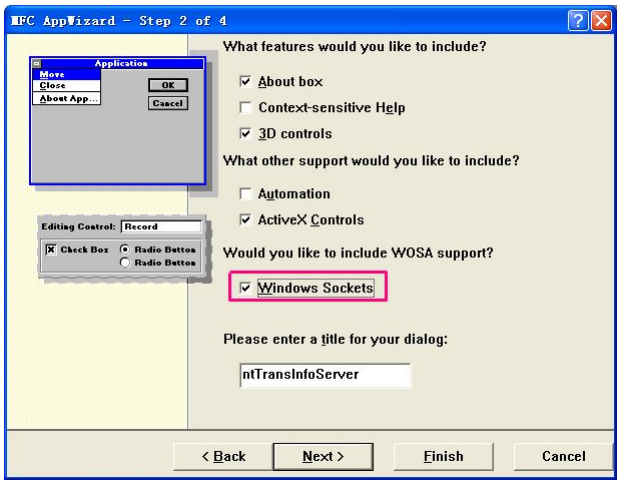

图 5.3　Windows Sockets 选项支持

然后在 CntTransInfoServerApp 类的 InitInstance 函数中自动添加了 WinSock 初始化代码，如下：

```
BOOL CntTransInfoServerApp::InitInstance()
{
    if (!AfxSocketInit())
    {
        AfxMessageBox(IDP_SOCKETS_INIT_FAILED);
        return FALSE;
    }
...
```

下面介绍实例的核心部分——监听 Socket 和通信 Socket 的实现。

1. 监听 Socket

以 CAsyncSocket 类为基类，派生新类 CListenSocket，用于实现服务器端的监听套接字。重载 CAsyncSocket 类的 OnAccept 函数，如果侦听到连接请求，调用 Accept 函数接受连接，并创建通信 Socket，触发通信 Socket 的 FD_READ 事件，进入读取数据状态。

实现代码如下：

```
void CListenSocket::OnAccept(int nErrorCode)
{
    //TODO：在此添加需要的代码或者调用基类的实现
    //侦听到连接请求，调用 Accept 函数
    CNewSocket* pSocket = new CNewSocket();
    if(Accept(*pSocket))                          //接受连接
    {
        pSocket->AsyncSelect(FD_READ);            //触发通信 Socket 的 Read 函数读数据
        m_pSocket=pSocket;                        //记录当前通信 Socket
    }
    else
    delete pSocket;                               //清除 Socket
    CAsyncSocket::OnAccept(nErrorCode);
}
```

2. 通信 Socket

以 CAsyncSocket 类为基类，派生新类 CNewSocket，用于实现与客户端的通信套接字。在 CNewSocket 类中，重载 CAsyncSocket 类的 OnReceive、OnSend 函数，实现从客户端接收消息和

发送消息至客户端。

实现代码如下：

```
void CNewSocket::OnReceive(int nErrorCode)                    //接收数据
{
    //TODO：添加需要的代码或者调用基类的实现
    m_nLength=Receive(m_szBuffer,sizeof(m_szBuffer),0);
    //直接转发消息
    AsyncSelect(FD_WRITE);                                    //触发 OnSend 函数
    CAsyncSocket::OnReceive(nErrorCode);
}

void CNewSocket::OnSend(int nErrorCode)                       //发送数据
{
    //TODO：添加需要的代码或者调用基类的实现
    char m_sendBuffer[4096];                                  //消息缓冲区
    memcpy(m_sendBuffer,"服务器转发：",24);
    strncat(m_sendBuffer,m_szBuffer,m_nLength);
    Send(m_sendBuffer,sizeof(m_sendBuffer));                  //发送数据
    AsyncSelect(FD_READ);                                     //触发 OnReceive 函数
    CAsyncSocket::OnSend(nErrorCode);
}
```

在主对话框启动服务按钮响应函数中实现创建监听套接字，使服务器进入监听状态，实现代码如下：

```
void CntTransInfoServerDlg::OnStar()
{
    //TODO：在此添加控制通知处理代码
    if(m_listenSocket.m_hSocket==INVALID_SOCKET)
    {
        //创建监听套接字，激发 FD_ACCEPT 事件，默认端口为 4088
        BOOL bFlag=m_listenSocket.Create(4088,SOCK_STREAM,FD_ACCEPT);
        if(!bFlag)                                            //创建失败
        {
            AfxMessageBox("Socket 创建失败！");
            m_listenSocket.Close();                           //关闭监听套接字
            PostQuitMessage(0);                               //退出窗口
            return;
        }
        GetDlgItem(IDC_STAR)->EnableWindow(false);
    }
    //“侦听”成功，等待连接请求
    if(!m_listenSocket.Listen(1))                             //如果监听失败
    {
        int nErrorCode = m_listenSocket.GetLastError();       //检测错误信息
        if(nErrorCode!=WSAEWOULDBLOCK)                        //如果不是线程被阻塞
        {
                AfxMessageBox("Socket 错误!");
            m_listenSocket.Close();                           //关闭套接字
                PostQuitMessage(0);                           //关闭窗口
            return;
        }
    }
}
```

在该实例中，在 FD_READ 事件响应函数 OnReceive 中，调用 Receive 函数从客户端接收数据

后立即触发 FD_WRITE 事件，即调用 OnSend 函数进入发送数据状态；同样，在 FD_WRITE 事件响应函数 OnSend 中，调用 Send 函数向客户端发送数据后立即触发 FD_READ 事件，调用 OnReceive 函数进入接收数据状态。这样，服务端就处于异步数据通信状态。

在实际应用中，程序员一般不直接使用 CAsyncSocket 类，而是自己定义他们的派生类，主要原因就是需要捕获 Socket 激活的事件，如 Socket 连接建立、数据接收完毕等。

5.2.5 使用 CAsyncSocket 类实现信息转发器（客户端）

在上节实现了信息转发器的服务端，本节将实现客户端的开发。客户端主要实现的功能是连接服务器，通过客户套接字向服务器发送信息，并接收服务器转发回来的信息。其运行界面如图 5.4 所示。

图 5.4 客户端的运行界面

实例使用 Visual C++ 6.0 开发，使用 AppWizard 创建一个基于对话框的 MFC 工程，工程名为 ntTransInfoClient。同样，在 MFC AppWizard Step 2 时，需要选中 Windows Sockets 复选项。

下面介绍实例的核心部分，即客户 Socket 的实现。

以 CAsyncSocket 类为基类，派生新类 CClientSocket，用于实现与服务器通信的客户套接字。在 CClientSocket 类中，重载 CAsyncSocket 类的 OnConnect、OnReceive、OnSend 函数。

OnConnect 函数实现连接到服务器后触发 FD_READ 事件，进入读数据状态，代码如下：

```
void CClientSocket::OnConnect(int nErrorCode)
{
    //TODO：添加需要的代码或者调用基类的实现
    if(nErrorCode==0)                              //连接成功
    {
        m_bConnected=TRUE;
        //下面两行代码用来获取对话框指针
        CntTransInfoClientApp* pApp=(CntTransInfoClientApp*)AfxGetApp();
        CntTransInfoClientDlg* pDlg=(CntTransInfoClientDlg*)pApp->m_pMainWnd;
        memcpy(m_szBuffer,"连接到： ",13);
        strncat(m_szBuffer,pDlg->m_szServerAdr,sizeof(pDlg->m_szServerAdr));
        pDlg->m_MsgR.InsertString(0,m_szBuffer);
        pDlg->GetDlgItem(IDC_SEND)->EnableWindow(true);          //激活发送按钮
        AsyncSelect(FD_READ);       //提请一个“读”的网络事件，准备接收
    }
```

```
    CAsyncSocket::OnConnect(nErrorCode);
}
```

在 OnReceive 函数中，使用 CAsyncSocket 类的 Receive()函数从服务器接收数据，并将收到的数据添加到列表框中，实现代码如下：

```
void CClientSocket::OnReceive(int nErrorCode)
{
    //TODO：添加需要的代码或者调用基类的实现
    m_nLength=Receive(m_szBuffer,sizeof(m_szBuffer),0);
    //下面两行代码用来获取对话框指针
    CntTransInfoClientApp* pApp=(CntTransInfoClientApp*)AfxGetApp();
    CntTransInfoClientDlg* pDlg=(CntTransInfoClientDlg*)pApp->m_pMainWnd;
    pDlg->m_MsgR.InsertString(0,m_szBuffer);                    //列表框添加接收的数据
    memset(m_szBuffer,0,sizeof(m_szBuffer));                    //清除缓冲区数据
    CAsyncSocket::OnReceive(nErrorCode);
}
```

在 OnSend 函数中，使用 CAsyncSocket 类的 Send()函数向服务器发送数据，然后触发 FD_READ 事件，进入读数据状态，实现代码如下：

```
void CClientSocket::OnSend(int nErrorCode)
{
    //TODO：添加需要的代码或者调用基类的实现
    Send(m_szBuffer,m_nLength,0);                               //发送数据
    m_nLength=0;
    memset(m_szBuffer,0,sizeof(m_szBuffer));                    //清除缓冲区数据
    //继续提请一个“读”的网络事件，接收 Server 消息
    AsyncSelect(FD_READ);
    CAsyncSocket::OnSend(nErrorCode);
}
```

最后介绍一个主对话框中连接和发送按钮响应函数的实现。

在连接按钮响应函数 OnConnect 中创建一个计数器，每间隔 1 秒尝试连接服务器，直至连接成功或者超时。实现代码如下：

```
void CntTransInfoClientDlg::OnConnect()                         //连接按钮响应函数
{
    //TODO：添加控制通知处理代码
    m_clientSocket.ShutDown(2);
    m_clientSocket.m_hSocket=INVALID_SOCKET;
    m_clientSocket.m_bConnected=FALSE;
    GetDlgItem(IDC_CONNECT)->EnableWindow(false);
    //默认端口为 4088
    m_szPort=4088;
    UpdateData(true);
    if(!m_ServerAddress.IsEmpty())
    {
        GetDlgItem(IDC_EDIT2)->EnableWindow(false);
        memcpy(m_szServerAdr,m_ServerAddress,sizeof(m_szServerAdr));
        //建立计时器，每 1 秒尝试连接一次，直到连上或 TryCount>10
        SetTimer(1,1000,NULL);
        TryCount=0;
    }
}
```

在定时器函数 OnTimer()中，创建套接字，连接服务器，实现代码如下：

```
void CntTransInfoClientDlg::OnTimer(UINT nIDEvent)
{
```

```
    //TODO：在此添加消息处理代码或者调用默认实现
    if(m_clientSocket.m_hSocket==INVALID_SOCKET)
    {
        BOOL bFlag=m_clientSocket.Create(0,SOCK_STREAM,FD_CONNECT);      //创建套接字
        if(!bFlag)
        {
            AfxMessageBox("Socket 创建错误");
            m_clientSocket.Close();
            GetDlgItem(IDC_EDIT2)->EnableWindow(true);
            return;
        }
    }
    m_clientSocket.Connect(m_szServerAdr,m_szPort);          //连接服务器
    TryCount++;                                              //连接次数加 1
    if(TryCount >=10 || m_clientSocket.m_bConnected)         //如果已经连接或连接超时
    {
        KillTimer(1);                                        //关闭定时器
        if(TryCount >=10)                                    //连接超时
        {
            AfxMessageBox("连接失败!");
            GetDlgItem(IDC_CONNECT)->EnableWindow(true);
            GetDlgItem(IDC_EDIT2)->EnableWindow(true);
        }
        return;
    }
    CDialog::OnTimer(nIDEvent);
}
```

在对话框发送按钮响应函数 OnSend()中，触发 Socket 的 FD_WRITE 事件，实现向服务器发送编辑框中的数据。代码如下：

```
void CntTransInfoClientDlg::OnSend()
{
    //TODO：添加控制通知处理代码
    if(m_clientSocket.m_bConnected)
    {
        m_clientSocket.m_nLength=m_MsgS.GetWindowText(m_clientSocket.m_szBuffer,
        sizeof(m_clientSocket.m_szBuffer));
        m_clientSocket.AsyncSelect(FD_WRITE);      //触发 Socket 的 OnSend 函数
    }
}
```

至此，客户端程序基本开发完毕。

5.3 CSocket 类及其开发

CSocket 是 MFC 在 CAsyncSocket 基础上派生的一个同步阻塞 Socket 的封装类，本节将简单介绍 CSocket 类及其开发实例。

5.3.1 CSocket 类

CSocket 类由 CAsyncSocket 派生，是 Windows Sockets API 的高级抽象，提供了更高层次的功能。CSocket 类提供了高级的 Socekt 支持，它运用了 MFC 的序列化类来提供和传输 Socket 对象。

CSocket 通常和 CSocketFile、CArchive 类混合使用。如可以将套接字上发送和接收的数据与一

个文件对象（CSocketFile）关联起来，通过读写文件来达到发送和接收数据的目的。另外，CSocket 对象提供阻塞模式，因为阻塞功能对于 CArchive 的同步操作是至关重要的，有关阻塞的概念在 5.3.3 节会有详细介绍。

CSocket 类是 CAsyncSocket 的派生类，继承了它封装的 WinSock API。CSocket 类除了继承了 CAsyncSocket 类的成员函数外，其他主要的成员函数及其功能如表 5.6 所示。

表 5.6　CSocket 类的主要成员函数及其功能

成员函数	功能
构造函数	
Csocket	构造一个 CSocket 对象
Create	创建一个套接字
属性	
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
操作函数	
CancelBlockingCall	取消一个当前正在进行中的套接字调用
可重载函数	
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

另外，进行 Socket 编程，不能不提到 CSocketFile 类。它并不是用来在 Socket 双方发送文件的，而是服务将需要序列化的数据的。如把一些结构体数据传给对方，程序的 CDocument()的序列化函数就完全可以和 CSocketFile 联系起来。

例如，有一个 CMyDocument 实现了 Serialize()，用户就通过如下的方式将文档数据传给 Socket 的另一方：

```
CSocketFile file( pSocket );
CArchive ar( &file, CArchive::store );
pDocument->Serialize( ar );
ar.Close();
```

同样，接收一方可以只改变上面的代码为 CArchive ar(&file, CArchive::load)即可。

需要注意的是，CArchive 类对象不能与数据报（UDP）套接字一起工作，因此对于数据报套接字，CAsyncSocket 和 CSocket 的使用方法是一样的。

5.3.2　CSocket 类的编程模式

使用 CSocket 类的编程模式可简单表示如下：

（1）构造一个 CSocket 对象，并用这个对象的 Create 成员函数产生一个 Socket 句柄。

在客户端程序中，除非需要数据报套接字，Create()函数一般情况下应该使用默认参数。而对于服务端程序，必须在调用 Create()时指定一个端口。

服务端的典型代码如下：

```
CSocket srvrSocket;                                    //构造一个 CSocket 对象
```

```
srvrSocket.Create(nPort);                    //nPort 为端口地址
```

客户端的典型代码如下：

```
CSocket clientSocket;                        //构造一个 CSocket 对象
clientSocket.Create();                       //缺省参数
```

（2）进行监听/连接操作。

如果是客户端程序，需要调用 CSocket::Connect()成员函数连接到服务端；而如果是服务端程序，则需要调用 AsyncSocket::Listen()成员函数开始监听，一旦接收到连接请求，则调用 CSocket::Accept()成员函数与客户端建立连接。

服务端的典型代码如下：

```
srvrSocket.Listen();                         //进入监听状态
CSocket SocketRecv;                          //创建一个新的 Socket 对象
srvrSocket. Accept(SocketRecv);              //接受客户端的连接请求
```

客户端的典型代码如下：

```
clientSocket.Connect (strAddr,nPort);        //strAddr 为机器名或 IP 地址
```

（3）进行数据通信。

这里 CSocket 类采用序列化的方法来进行通信，即与 CSocketFile 和 CArchive 类混合使用。首先需要产生一个 CSocketFile 对象，并把它与 CSocket 对象关联起来；接着为接收和发送数据各产生一个 CArchive 对象，把它们与 CSocketFile 对象关联起来；最后使用 CArchive 对象的 Read()、Write()等函数在客户端与服务端传送数据。

服务端的典型代码如下：

```
CSocketFile file(SocketRecv)                 //构造 CSocketFile 对象
CArchive arIn(&file, CArchive::load);        //构造 CArchive 对象
arIn>>dwValue;                               //利用 CArchive 对象传递数据
```

客户端的典型代码如下：

```
CSocketFile file(clientSocket)               //构造 CSocketFile 对象
CArchive arOut(&file, CArchive::store);      //构造 CArchive 对象
arOut<<dwValue;                              //利用 CArchive 对象传递数据
```

如果不使用 CSocketFile、CArchive 类的序列化方法，也可以通过调用 CSocket 类的成员函数 Receive()、ReceiveFrom()、Send()和 SendTo()进行数据通信，具体使用和 CAsyncSocket 类完全相同。

（4）通信完毕后，销毁 CArchive、CSocketFile 和 CSocket 对象。

5.3.3 CSocket 类的同步（阻塞）机制

Socket 有同步阻塞方式和异步非阻塞方式两种使用方法，下面对“同步（阻塞）”和“异步（非阻塞）”的概念进行简单介绍。

同步和异步往往都是针对一个函数来说的，“同步”就是函数直到其要执行的功能全部完成时才返回，而“异步”则是函数仅仅做一些简单的工作，然后马上返回，而它所要实现的功能留给其他线程或函数去完成。例如，SendMessage 就是“同步”函数，它不但发送消息到消息队列，还需要等待消息被执行完才返回；相反，PostMessage 就是一个异步函数，它只管发送一个消息，而不管这个消息是否被处理，就马上返回。

一个 Socket 可以处于“阻塞模式”或“非阻塞模式”。当一个套接字处于阻塞模式（同步操作）时，它的阻塞函数直到操作完成才会返回控制权，也就是说此套接字的阻塞函数在完成操作返回之前什么也不能做。如果一个 Socket 处于非阻塞模式（异步操作），则其调用函数立即返回。

CSocket 是 MFC 在 CAsyncSocket 基础上派生的一个同步阻塞 Socket 的封装类。它是如何把

CAsyncSocket 变成同步的，而且还能响应同样的 Socket 事件的呢？

其实很简单，CSocket 在 Connect()返回 WSAEWOULDBLOCK 错误时，并不是如 CAsyncSocket 类那样，在 OnConnect()、OnReceive()等这些事件的终端函数里去等待，而是马上进入一个消息循环，即从当前线程的消息队列里取它所关心的消息，如果取到了 WM_PAINT 消息，则刷新窗口；如果取到的是 Socket 发来的消息，则根据 Socket 是否有操作错误码调用相应的回调函数（如 OnConnect()等）。

其他的 CSocket 函数，如 Send()、Receive()、Accept()都在收到 WSAEWOULDBLOCK 错误时进入 PumpMessages()消息循环，这样一个原本异步的 CAsyncSocket，到了派生类 CSocket，就变成同步的了。

下面就以 CSocket 类的 Accept 函数为例，简单说明其阻塞模式的实现。CSocket::Accept 函数的源码实现如下：

```
while(!Accept(...))
{
    if (GetLastError( ) == WSAEWOULDBLOCK)
    PumpMessage(FD_ACCEPT);
    else
    return false;
}
```

它不断调用 CAsyncsocket::Accept（CSocket 派生自 CAsyncsocket 类）来判断服务端 socket 的事件队列中是否存在正在引入的连接事件 FD_ACCEPT，换句话说，就是判断是否有来自客户端 Socket 的连接请求。

如果当前服务端 Socket 的事件队列中存在正在引入的连接事件时，Accept 函数将返回一个非 0 值，跳出 while 循环体，解除阻塞；否则，Accept 返回 0，此时调用 GetLastError()将返回错误代码 WSAEWOULDBLOCK，表示队列中无任何连接请求，将执行 PumpMessage (FD_ACCEPT)，继续调用 CAsyncsocket::Accept 函数，等待连接。

PumpMessage 作为一个消息泵使得 Socket Window 中的消息能够维持在活动状态。

很显然，如果没有来自客户端 Socket 的连接请求，CSocket 就会不断地调用 Accept()产生循环阻塞，直到有来自客户端 Socket 的连接请求而解除阻塞。阻塞解除后，表示服务端 Socket 和客户端 Socket 已成功连接，服务端与客户端彼此可以开始通信。

如果服务端用于监听的 Socket 在主线程中运行，这将导致主线程的阻塞，因此当有客户端连接时需要创建一个新的线程以运行 Socket 服务。阻塞模式的通信过程可以表示为如图 5.5 所示。

在 Win32 环境下，如果要使用具有阻塞性质的套接字，应该放在独立的工作线程中处理，利用多线程的方法使阻塞不至于干扰其他线程，也不会把 CPU 时间浪费在阻塞上。多线程的方法既可以使程序员享受 CSocket 带来的简化编程的便利，也不会影响用户界面对用户的反应。

当然，CSocket 类也支持非阻塞模式，这是通过 Socket 事件的消息机制来实现的。通常需要从 CSocket 类派生一个新类，派生新类的目的是重载 Socket 事件的消息函数，然后在 Socket 事件的消息函数中添入合适的代码以完成客户端与服务端之间的通信。与阻塞模式相比，非阻塞模式无须创建一个新线程。

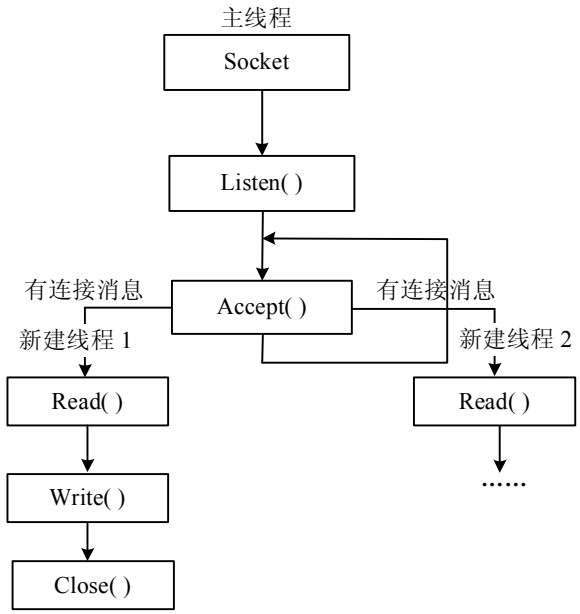

图 5.5　阻塞模式的通信过程

当有多个客户端 Socket 与服务端 Socket 连接及通信时，服务端采用阻塞模式就显得不适合了，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。

5.3.4　使用 CSocket 类的阻塞模式进行通信

本节将给出一个使用 CSocket 类实现有连接（基于 TCP/IP 协议）的数据通信的简单实例。在服务端，服务启动后，创建的 Socket 开始监听客户端的连接请求，并处于阻塞状态（无法进行其他操作），当收到客户端的连接请求时创建一个通信套接字，定时（间隔 2 秒）向外发送一个计数值，同时关闭监听套接字。

在客户端，启动接收数据时创建的客户 Socket 开始连接服务器，如果没有连接到服务器，它就处于阻塞状态（无法进行其他操作），直到与服务器建立了连接，开始定时（间隔 1 秒）接收数据，并显示数据。

实例的运行界面如图 5.6 所示。

图 5.6　实例的运行界面

下面分别介绍服务器端和客户端的开发过程。

1. 服务器端

服务器端程序实例使用 Visual C++ 6.0 开发，使用 AppWizard 创建一个基于对话框的 MFC 工程，工程名为 ntBlockServer。在 MFC AppWizard Step 2 时，需要选中 Windows Sockets 复选项。

在对话框窗口的启动按钮响应函数 OnSend()中，创建 CSocket 对象 m_sockListen，调用 Listen()函数进入监听状态，然后调用阻塞函数 Accept()进入阻塞模式，直到有客户连接服务器。然后创建定时器，定时向客户端发送数据。

OnSend()函数的实现代码如下：

```
void CntBlockServerDlg::OnSend()                          //启动服务按钮响应函数
{
    //TODO：在此添加控制通知处理代码
    UpdateData();
    if(m_sockListen.Create(5800,SOCK_STREAM,NULL))        //创建监听套接字
    {
        GetDlgItem(IDC_SEND)->EnableWindow(false);        //启动服务按钮无效
        GetDlgItem(IDC_STOP)->EnableWindow(true);         //停止服务按钮生效
        if(m_sockListen.Listen())
        {
            m_ServerStatus="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
            m_sockListen.Accept(m_sockSend);              //阻塞，当有连接进入时才返回
            m_sockListen.Close();
            SetTimer(1,2000,NULL);                        //创建一个定时器定时发送数据
        }
    }
    else
    {
        AfxMessageBox("Socket 创建失败！");
    }
}
```

WM_TIMER 消息响应函数 OnTimer()的实现代码如下：

```
void CntBlockServerDlg::OnTimer(UINT nIDEvent)
{
    //TODO：在此添加消息处理代码或者调用默认实现
    static iIndex=0;                                      //静态变量，用于计数
    char szSend[20];
    sprintf(szSend,"%010d",iIndex++);                     //发送的数据置入缓冲区
    //发送 TCP 数据
    int iSend= m_sockSend.Send(szSend,10,0);
    m_sendcount+=iSend;                                   //发送字节数
    CString str=szSend;
    m_ServerStatus="正在发送数据"+str;
    UpdateData(false);
    CDialog::OnTimer(nIDEvent);
}
```

2. 客户端

客户端程序实例使用 Visual C++ 6.0 开发，使用 AppWizard 创建一个基于对话框的 MFC 工程，工程名为 ntBlockClient。在 MFC AppWizard Step 2 时，需要选中 Windows Sockets 复选项。

在对话框窗口的接收数据按钮响应函数 OnReceive ()中创建 CSocket 对象 m_sockReceive，调

用阻塞函数 Connect ()连接服务器，进入阻塞模式，直到成功连接到服务器。然后创建定时器，定时从服务器接收数据。

OnReceive ()函数的实现代码如下：

```
void CntBlockClientDlg::OnReceive()
{
    //TODO：在此添加控制通知处理代码
    UpdateData();
    if(m_sockReceive.Create())                          //创建套接字
    {
        //发起连接请求
        m_sockReceive.Connect(m_ServerIP,m_ServerPort);  //连接服务器，成功后返回
        SetTimer(1,1000,NULL);                          //创建一个定时器定时接收
        m_ReceiveData="成功连接服务器！";
        UpdateData(false);
        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！");
    }
}
```

WM_TIMER 消息响应函数 OnTimer()的实现代码如下：

```
void CntBlockClientDlg::OnTimer(UINT nIDEvent)
{
    //TODO：在此添加消息处理代码或者调用默认实现
    char szRecv[20];
    int iRecv =m_sockReceive.Receive(szRecv,10,0);          //接收 TCP 数据
    if(iRecv>=0)
    {
        szRecv[iRecv]=NULL;
        m_ReceiveData=szRecv;                               //记录接收数据
        m_receivecount+=iRecv;                              //接收字节数
        m_ReceiveData=m_ReceiveData;
    }
    else                                                    //如果没有接收到任何数据
    {
        m_ReceiveData="没有收到数据！";
    }
    UpdateData(false);
    CDialog::OnTimer(nIDEvent);
}
```

该实例为典型的使用 CSocket 的阻塞模式实现基于 TCP 协议的有连接通信，其工作流程如图 5.7 所示。

CSocket 类提供了阻塞模式，即当执行 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。这种方式简化了网络程序的开发，但其弊端（阻塞主线程）也是显而易见的。在实际程序开发中，多采用的是非阻塞模式，即采用 Socekt 事件的方式处理程序的连接，在下节将给出详细的实例介绍。

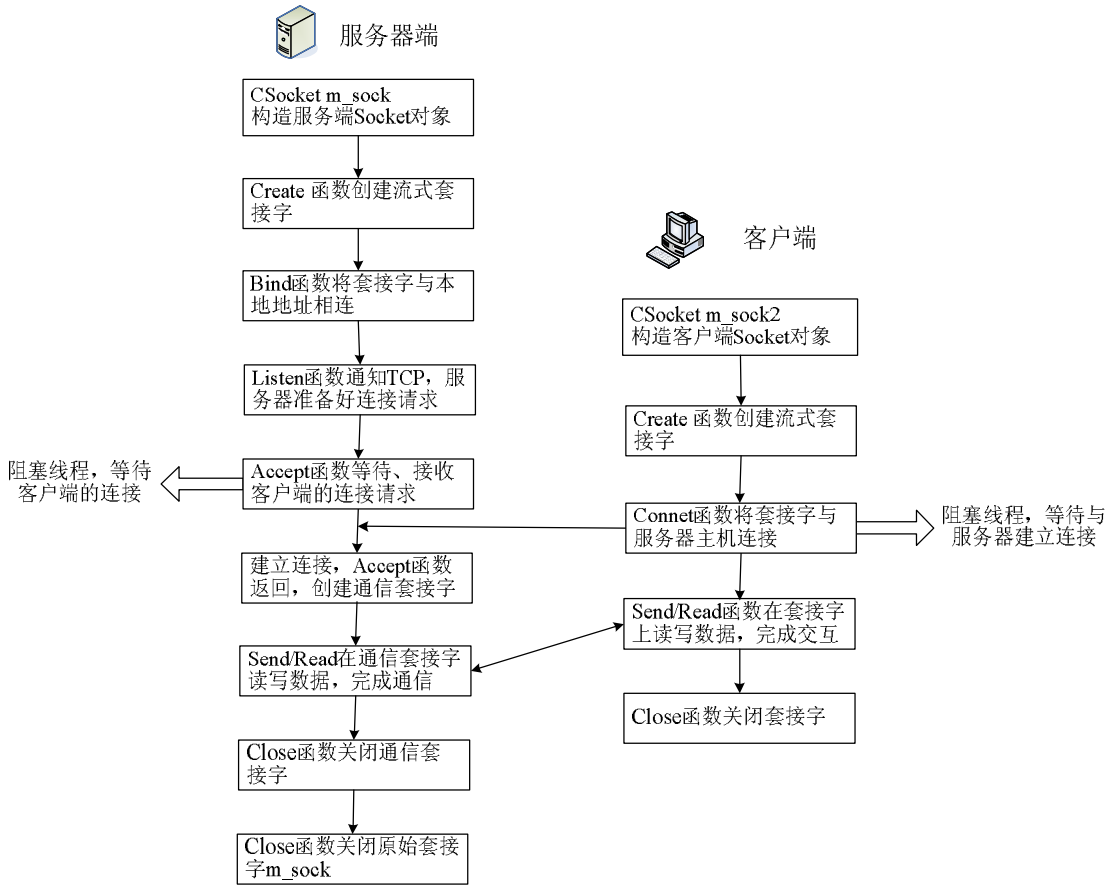

图 5.7　CSocket 类各函数的工作流程

5.4　网络聊天室开发实例

前面介绍了 MFC Socket 的基本开发技术，本节将基于 Visual C++，利用 MFC 实现一个简单的网络聊天室的开发。为了便于理解，这里只实现基本功能，即客户将聊天信息发送至服务器，而服务器将收到的信息分发给所有的客户，从而使各客户均能看得其他用户的聊天信息。在此基础上，读者可进行其他扩展开发。

5.4.1　服务器端程序的开发

服务器实现的基本功能是开启服务端口，等待客户的连接。当有客户连接后，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。

基本实现过程为从 CSocket 类派生两个 Socket 类：CListenSocket，用于创建监听 Socket；CCommSocket，用于创建通信 Socket。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，实现创建 CCommSocket 对象与客户 Socket 进行通信。具体开发过程如下：

（1）启动 Visual C++ 6.0，使用 AppWizard 创建一个基于对话框的 MFC 工程，工程名为

ntChatRoomServer。在 MFC AppWizard Step 2 时，需要选中 Windows Sockets 复选项。

（2）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。在其中重载 CSocket 类的 OnAccept()函数，收到连接请求后创建客户套接字，与客户端进行通信。实现代码如下：

```
void CListenSocket::OnAccept(int nErrorCode)
{
  //TODO：在此添加需要的代码或者调用基类的实现
  CClientSocket *tmp=new CClientSocket(&CCSL);          //创建客户套接字
  Accept(*tmp);                                         //接收客户数据
  CCSL.Add(tmp);                                        //添加到客户套接字列表中
  CSocket::OnAccept(nErrorCode);
}
```

（3）以 CSocket 类为基类派生新类 CCommSocket，用于创建客户 Socket 列表。重载 OnReceive()函数，实现将服务器接收的信息分发给所有连接至服务器的 Socket。

CCommSocket 类的头文件实现如下：

```
class CCommSocketList;
class CCommSocket : public CSocket
{
    public:
    //操作
    public:
        CCommSocket(CCommSocketList *);          //带参数的构造函数
        virtual ~CCommSocket();
    //覆盖
    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，如下：

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

OnReceive()函数的实现如下：

```
void CCommSocket::OnReceive(int nErrorCode)
{
    //TODO：在此添加需要的代码或者调用基类的实现
    List->Sends(this);                           //调用 CCommSocketList 的 send 函数
    CSocket::OnReceive(nErrorCode);
}
```

（4）创建 CCommSocketList 类，其头文件定义如下：

```
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
};
```

函数 Sends()用于向客户 Socket 发送数据，实现代码如下：

```
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;
}
```

函数 Add()用于向链表中添加 CCommSocket 对象，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;
}
```

（5）启动服务和停止服务按钮响应函数的实现代码如下：

```
void CntChatRoomServerDlg::OnStart()                    //启动服务
{
    //TODO：在此添加控制通知处理代码
    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 CntChatRoomServerDlg::OnStop()                     //停止服务
{
    //TODO：在此添加控制通知处理代码
    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
}
```

实例的运行结果如图 5.8 所示。

图 5.8　服务器端的运行结果

5.4.2　客户端程序的开发

当客户端程序启动后，首先弹出“连接”对话框，通过指定的地址和用户名连接服务器。连接成功后，客户就可以向服务器发信息，并接收服务器转发的信息。

基本的实现过程为创建 CSocket 类的派生类 CServerSocket，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 Socket 发送的数据，并在窗口中显示。具体开发过程如下：

（1）启动 Visual C++ 6.0，使用 AppWizard 创建一个基于对话框的 MFC 工程，工程名为 ntChatRoomClient。在 MFC AppWizard Step 2 时，需要选中 Windows Sockets 复选项。

（2）以 CSocket 类为基类派生新类 CServerSocket，实现连接服务器、向服务器发送消息，并从服务器接收信息的操作。

在 CServerSocket 类的头文件中声明相关成员变量和函数，如下：

```
class CntChatRoomClientDlg;                                    //主对话框类
class CServerSocket : public CSocket
{
    //属性
    public:
        CntChatRoomClientDlg * myDlg;                          //主对话框指针
        BOOL SetDlg(CntChatRoomClientDlg *tmp);
        CString UserName;                                      //用户名
        …
```

SetDlg()函数实现在 Socket 对象中获取主对话框的指针，代码如下：

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

重载 CSocket 类的 OnReceive 函数，调用主对话框的成员函数 GetMessage，接收服务器的信息，并添加到窗口的列表框中，代码如下：

```
void CServerSocket::OnReceive(int nErrorCode)
{
    //TODO：在此添加需要的代码或者调用基类的实现
    myDlg->GetMessage();                                       //接收服务器信息
    CSocket::OnReceive(nErrorCode);
}
```

主窗口类的 GetMessage()函数的实现代码如下：

```
BOOL CntChatRoomClientDlg::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;
}
```

（3）在“连接”对话框的“连接”按钮响应函数 OnConnect()中，实现使用 CServerSocket 对象连接服务器。当连接服务器成功后，如果服务器有发送到本端的数据，则会激发 FD_READ 事件，调用 OnReceive 函数，读取数据，并显示在主窗口的列表框中。实现代码如下：

```
void CConnectedDlg::OnConnect()
{
    //TODO：在此添加控制通知处理代码
    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，注意端口号
    {
        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
}
```

（4）在主窗口的“发送消息”按钮响应函数 OnButtonSend()中，实现使用 CServerSocket 对象向服务器发送数据，实现代码如下：

```
void CntChatRoomClientDlg::OnButtonSend()
{
    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、接收数据的功能均已实现了。有关程序的其他 MFC 相关开发可参阅本书所附的程序源代码。

客户端程序的运行结果如图 5.9 所示。

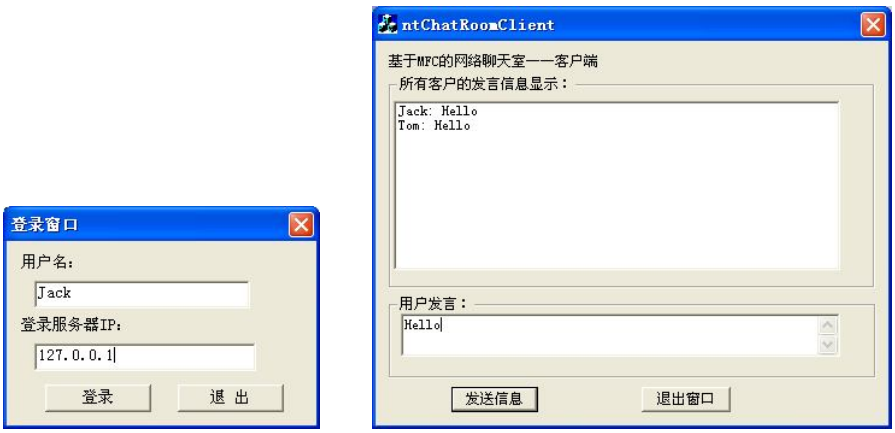

图 5.9　客户端的运行结果

5.5　本章小结

本章主要讲解使用 MFC 提供的 Socket 封装类 CAsyncSocket 和 CSocket 进行网络通信程序开发的方法，介绍了 CAsyncSocket 和 CSocket 类的主要函数、编程模式，并给出了具体的开发实例。本章的重点是有关 CSocket 类的同步（阻塞）机制和异步（非阻塞）机制的选择与使用。