MFC Socket

Hanford

2016 年 12 月 04 日

目 录

第 1 章　同步 TCP 通讯

本文将说明如何使用 MFC 的 CSocket 和 CAsyncSocket 实现同步、异步 TCP/UDP 通讯。文中的代码已被上传至 git 服务器：

https://github.com/hanford77/Exercise

https://git.oschina.net/hanford/Exercise

在目录 Socket 里。

1.1　同步通讯与异步通讯

同步通讯与异步通讯就好比 SendMessage 与 PostMessage。

SendMessage 直接把消息提交给窗口过程进行处理。它返回时，消息已经被处理完毕。

PostMessage 只是把消息放入消息队列，在系统空闲的时候才会调用窗口过程处理消息。

同步通讯每调用一次读写函数，实际的读写工作都将被完成。因此，它的特点就是效率较低，但是代码逻辑结构较为简单。

异步通讯每调用一次读写函数，实际的读写工作可能并没有完成。因此，它的特点就是效率较高，但是代码逻辑结构较为复杂。

1.2　同步通讯类

同步通讯类 CSocketSync 派生自 CSocket，用于 TCP/UDP 的同步通讯。代码如下：

```
class CSocketSync : public CSocket
{
protected:
    enum { UDP_MAX_PKG = 65507 };  //UDP 数据包最大字节数
    ISocketEventHandler*m_pEventHandler; //套接字事件处理器
public:
    CSocketSync(ISocketEventHandler*pEvent = NULL)
```

```
    {
        m_pEventHandler    =    pEvent;
    }
    void  SetEventHandler(ISocketEventHandler*pEvent)
    {
        m_pEventHandler    =    pEvent;
    }
public:
    //用于发送 UDP 数据
    int  SendTo(const  void*  lpBuf,  int  nBufLen,UINT  nHostPort
            ,  LPCTSTR  lpszHostAddress  =  NULL,  int  nFlags  =  0)
    {
        SOCKADDR_IN  sockAddr;
        if(CSocketAsync::IpAddress(lpszHostAddress,nHostPort,sockAddr))
        {
            return  SendTo(lpBuf,nBufLen,(SOCKADDR*)&sockAddr
                    ,sizeof(sockAddr),nFlags);
        }
        return  0;
    }
    //用于发送 UDP 数据
    int  SendTo(const  void*pData,int  nLen,const  SOCKADDR*lpSockAddr
            ,int  nSockAddrLen,int  nFlags  =  0)
    {
        if(pData  &&  lpSockAddr  &&  nSockAddrLen  >  0)
        {
            if(nLen  <  0)
            {
                nLen    =    strlen((const  char*)pData);
            }
            if(nLen  >  0)
            {
                int  n          =    0;   //单次发送的字节数
                int  nSum       =    0;   //发送的累计字节数
                for(;;)
                {
                    n    =    nLen  -  nSum;      //待发送的字节数
                    if(n  >  UDP_MAX_PKG)
                    {//待发送的字节数不能太大
                        n    =    UDP_MAX_PKG;
                    }
                    n    =    CSocket::SendTo(pData,n,lpSockAddr
                                    ,nSockAddrLen,nFlags);
                    if(n  >  0)
```

```
                    {
                        nSum  +=   n;   //累计
                        if(nSum >= nLen)
                        {//发送完毕
                            return nLen;
                        }
                    }
                    else if(n == SOCKET_ERROR)
                    {
                        return nSum;
                    }
                }
            }
        }
        return 0;
    }
protected:
    virtual void OnReceive(int nErrorCode)
    {//接收到数据时，会触发该事件
        if(m_pEventHandler)
        {
            m_pEventHandler->OnReceive(this,nErrorCode);
        }
    }
    virtual void OnClose(int nErrorCode)
    {//连接断了，会触发该事件
        if(m_pEventHandler)
        {
            m_pEventHandler->OnClose(this,nErrorCode);
        }
    }
};
```

上述代码中，SendTo 函数用于 UDP 通讯，暂时不用管它。重点是事件处理，如：CSocket 接收到 OnReceive 事件时，会调用 virtual void OnReceive(int nErrorCode)函数。后者会调用 m_pEventHandler->OnReceive(this,nErrorCode);让 m_pEventHandler 来处理这个事件。

m_pEventHandler 是一个事件处理器（ISocketEventHandler*），接口 ISocketEventHandler 的定义如下：

```
class ISocketEventHandler
{
public:
    virtual void OnAccept (CAsyncSocket*pSender,int nErrorCode){}
```

```
    virtual  void  OnClose   (CAsyncSocket*pSender,int  nErrorCode){}
    virtual  void  OnConnect(CAsyncSocket*pSender,int  nErrorCode){}
    virtual  void  OnReceive(CAsyncSocket*pSender,int  nErrorCode){}
    virtual  void  OnSend    (CAsyncSocket*pSender,int  nErrorCode){}
};
```

1.3　同步 TCP 通讯客户端

1.3.1　界面

TCP 客户端界面如下：

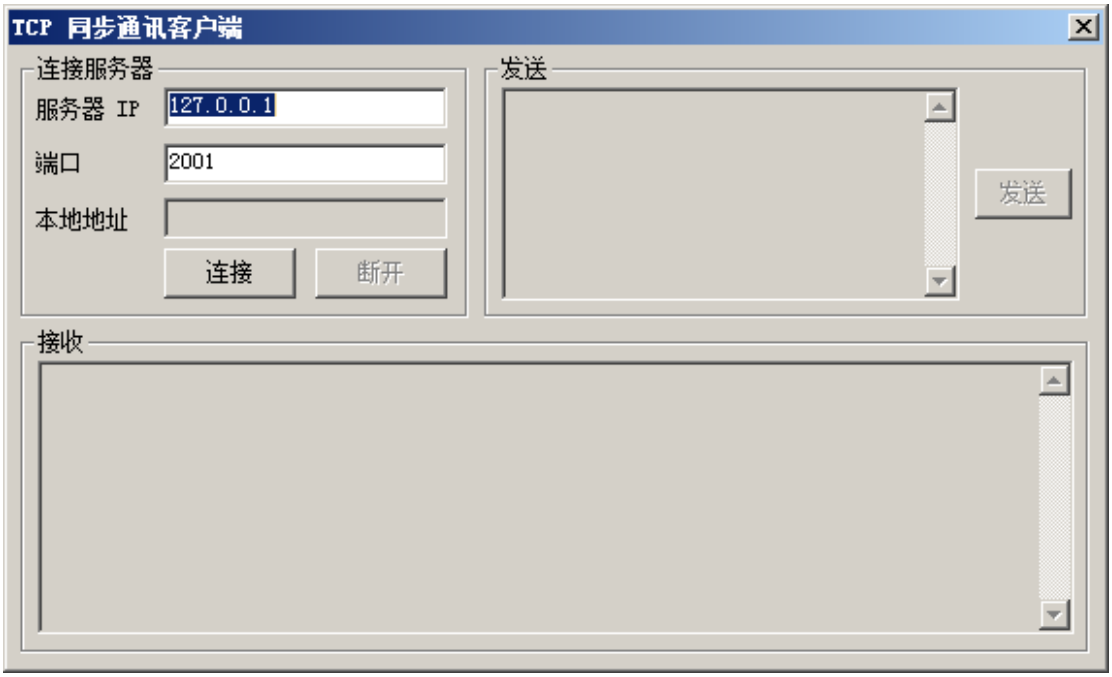

图 1.1

1.3.2　界面类声明

上面的界面，对应于类 CDlgTcpClientSync，其声明代码如下：

```
class  CDlgTcpClientSync  :  public  CDialog  ,  public  ISocketEventHandler
{
    ...    ...    ...
protected://ISocketEventHandler
    virtual  void  OnReceive(CAsyncSocket*pSender,int  nErrorCode);
```

```
    virtual void OnClose   (CAsyncSocket*pSender,int nErrorCode);
protected:
    CSocketSync m_Socket;
    std::string m_sRecv;
};
```

上述代码的重点：

1）定义了 CSocketSync m_Socket，用于套接字通讯；

2）成员变量 std::string m_sRecv 用来存储接收到的数据；

3）CDlgTcpClientSync 继承了 ISocketEventHandler，并重写了 OnReceive、OnClose 函数。m_Socket 的套接字事件，将由 OnReceive、OnClose 函数来处理。

1.3.3　界面类构造函数

界面类构造函数如下

```
CDlgTcpClientSync::CDlgTcpClientSync(CWnd* pParent /*=NULL*/)
: CDialog(CDlgTcpClientSync::IDD, pParent)
{
    m_Socket.SetEventHandler(this);
}
```

设置 m_Socket 的事件处理器为 this。其含义为：m_Socket 的 OnReceive 被调用时，就会调用函数 CDlgTcpClientSync::OnReceive。

1.3.4　连接服务器

单击图 1.1的“连接”按钮，将连接服务器。示例代码如下：

```
CString sIP      = _T("127.0.0.1");   //服务器 IP
int     nPort    = 2001;              //服务器端口
if(m_Socket.Socket()                  //创建套接字成功
&& m_Socket.Connect(sIP,nPort)   //连接服务器成功
)
{//连接成功
    ...    ...    ...
}
else
{//连接失败
    m_Socket.Close();      //关闭套接字
    ...    ...    ...
}
```

重点在于：

1）调用 m_Socket.Socket 创建套接字；

2）调用 m_Socket.Connect 连接服务器。

1.3.5　写数据

单击图 1.1的“发送”按钮，将发送数据给服务器。示例代码如下：

```
std::string  s(1024,,'1');
m_Socket.Send(s.c_str(),s.length());
```

重点就是调用 CSocket::Send 函数发送数据。上面的代码将发送 1024 字节的'1'给服务器。因为是同步的，所以数据未发送完毕之前，这个函数是不会返回的。

1.3.6　读数据

服务器发送数据过来时，CSocket::OnReceive 会被调用。m_Socket 所属的类 CSocketSync 重写了 OnReceive 函数，因此 CSocketSync::OnReceive 会被调用。代码 m_pEventHandler->OnReceive(this,nErrorCode);会被调用。这里的 m_pEventHandler 是一个 CDlgTcpClientSync*，后者重写了 ISocketEventHandler::OnReceive 函数，因此：程序一旦接收到服务器发来的数据，函数 CDlgTcpClientSync::OnReceive 将被调用。其代码如下：

```
char      buf[1024];
int       nRead     =     0;

while((nRead  =  m_Socket.CAsyncSocket::Receive(buf,sizeof(buf)))  >  0)
{//异步读取。读取到的数据加入变量 m_sRecv
      m_sRecv  +=    std::string((const  char*)buf,nRead);
}
```

重点在于：调用 CAsyncSocket::Receive 函数读取发送过来的数据。注意：CAsyncSocket::Receive 是异步读取的，它仅仅从套接字输入缓冲区内读取数据。如果输入缓冲区为空，它就直接返回-1。

如果把 m_Socket.CAsyncSocket::Receive 中的“.CAsyncSocket”删除，那么调用的就是 CSocket::Receive 函数。CSocket::Receive 是同步读取的，亦即在未读取到预定字节数（上面代码中的 sizeof(buf)）之前，它是不会返回的。结果就是整个程序会阻塞在这一行，将处于假死状态。

1.3.7　断开连接

单击图 1.1的“断开”按钮，将执行如下代码：

```
m_Socket.ShutDown(CAsyncSocket::both); //禁止 TCP 连接收、发数据
m_Socket.Close();   //关闭套接字
```

如果是服务器断开了此 TCP 连接，则 CDlgTcpClientSync::OnClose 会被调用。代码如下：

```
void  CDlgTcpClientSync::OnClose(CAsyncSocket*pSender,int  nErrorCode)
{
      m_Socket.ShutDown(CAsyncSocket::both);
      m_Socket.Close();
}
```

1.4　同步 TCP 通讯服务端

1.4.1　界面

TCP 服务端界面如下：

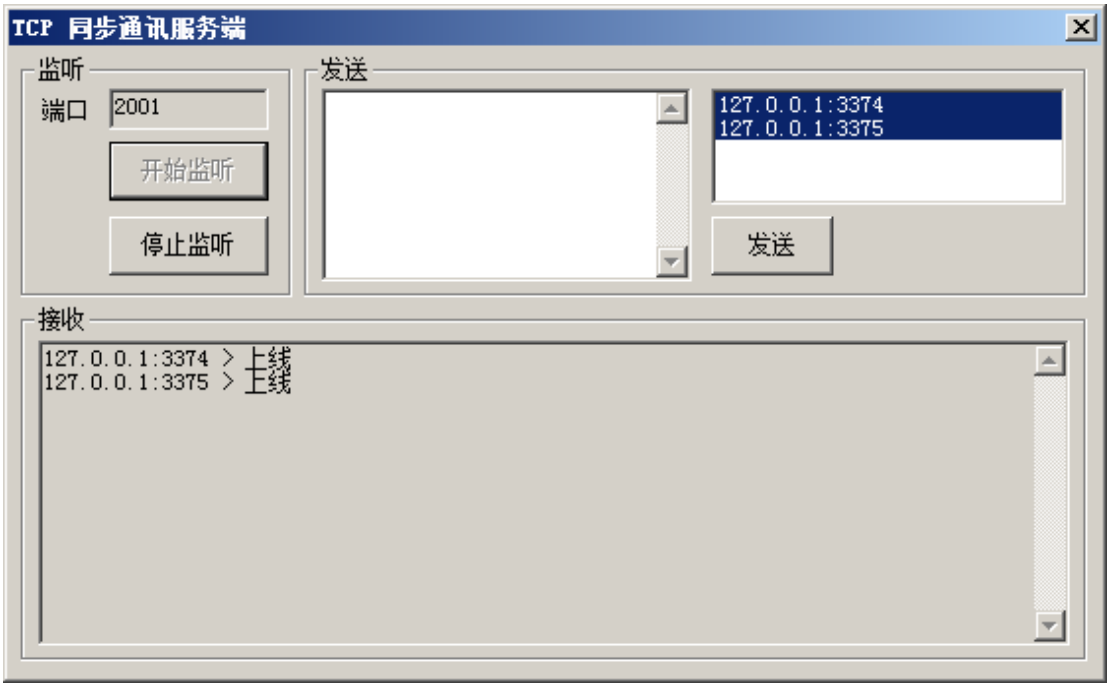

图 1.2

1.4.2　界面类声明

上面的界面，对应于类 CDlgTcpServerSync，其声明代码如下：

```
class CDlgTcpServerSync : public CDialog , public ISocketEventHandler
{
    ...    ...    ...
protected://ISocketEventHandler
    virtual void OnAccept (CAsyncSocket*pSender,int nErrorCode);
    virtual void OnReceive(CAsyncSocket*pSender,int nErrorCode);
    virtual void OnClose  (CAsyncSocket*pSender,int nErrorCode);
protected:
    typedef std::map<CSocketSync*,CString> MapClient;
    CSocketTcpListen  m_SocketListen;  //一个监听套接字
    MapClient  m_mapClient;       //一个客户端对应一个通讯套接字
    CString    m_sRecv;           //接收到的数据
};
```

上述代码的重点：

1）定义了 CSocketTcpListen m_SocketListen，该套接字用于监听客户端发送过来的连接请求；

2）定义了 std::map<CSocketSync*,CString> m_mapClient，用来存储多个通讯套接字，每个通讯套接字对应于一个客户端；

3）成员变量 m_sRecv 用来存储接收到的数据；

4）CDlgTcpServerSync 继承了 ISocketEventHandler，并重写了 Socket 事件处理函数。

1.4.3　CSocketTcpListen

CSocketTcpListen 继承自 CAsyncSocket，用来监听客户端发送过来的连接请求。其代码如下：

```
class CSocketTcpListen : public CAsyncSocket
{
public:
    CSocketTcpListen(ISocketEventHandler*pEvent = NULL)
    {
        m_pEventHandler    =    pEvent;
    }
    void SetEventHandler(ISocketEventHandler*pEvent)
    {
        m_pEventHandler    =    pEvent;
```

```
    }
protected:
    virtual void OnAccept(int nErrorCode)
    {//客户端发送连接请求时，会触发该事件
        if(m_pEventHandler)
        {
            m_pEventHandler->OnAccept(this,nErrorCode);
        }
    }
public:
    ISocketEventHandler*m_pEventHandler;  //套接字事件处理器
};
```

1.4.4　界面类构造函数

界面类构造函数如下

```
CDlgTcpServerSync::CDlgTcpServerSync(CWnd* pParent /*=NULL*/)
: CDialog(CDlgTcpServerSync::IDD, pParent)
{
    m_SocketListen.SetEventHandler(this);
}
```

设置 m_SocketListen 的事件处理器为 this。其含义为：m_SocketListen 的 OnAccept 被调用时，就会调用函数 CDlgTcpServerSync::OnAccept。

1.4.5　开始监听

单击图 1.2的“开始监听”按钮，示例代码如下：

```
int                 nPort    =    2001;  //监听该端口
CSocketTcpListen&   s        =    m_SocketListen;
if(s.Create(nPort)    //创建套接字，绑定端口
&& s.Listen())        //开始监听
{//监听成功
    EnableControls();
}
else
{//监听失败
    s.Close();
}
```

1.4.6　客户端上线

服务端接收到客户端的连接请求时，函数 CSocketTcpListen::OnAccept 和 CDlgTcpServerSync::OnAccept 会被依次调用。后者的代码如下：

```
void CDlgTcpServerSync::OnAccept(CAsyncSocket*pSender,int nErrorCode)
{
    if(0 == nErrorCode)
    {
        CSocketSync*pClient = new CSocketSync(this);
        if(m_SocketListen.Accept(*pClient))
        {//接受连接请求成功
            m_mapClient[pClient] = GetPeerName(pClient);
        }
        else
        {//接受连接请求失败
            delete pClient;
        }
    }
}
```

上面代码的重点在于：

1）调用 CAsyncSocket::Accept 函数，接受客户端的连接请求；

2）将客户端套接字加入到 m_mapClient 里。m_mapClient 的 Key 是 CSocketSync*pClient 可用于与客户端通讯；m_mapClient 的 Value 由函数 GetPeerName 获得，是客户端的 IP 地址和端口号，如：“127.0.0.1:5000”。函数 GetPeerName 的代码如下：

```
inline CString GetPeerName(CAsyncSocket*p)
{
    CString s;
    if(p)
    {
        CString sIP;
        UINT    nPort;
        if(p->GetPeerName(sIP,nPort))
        {
            s.Format(_T("%s:%u"),sIP,nPort);
        }
    }
    return s;
}
```

3）new CSocketSync(this) 将 Socket 事件处理器设置为 this，也就是 CDlgTcpServerSync。因此，当 CSocketSync::OnReceive、CSocketSync::OnClose 被调用时，CDlgTcpServerSync::OnReceive、CDlgTcpServerSync::OnClose 也会被

调用。

1.4.7　写数据

单击图 1.2的“发送”按钮，将发送数据给客户端。下面的代码将 1024 个'1'发送给所有的客户端：

```
std::string  s(1024,'1');
for(MapClient::iterator  it  =  m_mapClient.begin();
    it  !=  m_mapClient.end();++it)
{
     it->first->Send(s.c_str(),s.length());
}
```

重点就是调用 CSocket::Send 函数发送数据。因为是同步的，所以数据未发送完毕之前，这个函数是不会返回的。

1.4.8　读数据

某个客户端发送数据过来时，CDlgTcpServerSync::OnReceive 将被调用。代码如下：

```
void  CDlgTcpServerSync::OnReceive(CAsyncSocket*pSender
                                   ,int  nErrorCode)
{
     if(pSender)
     {
          CSocketSync*pClient      =     (CSocketSync*)pSender;
          char              buf[1024];
          int               nRead    =     0;
          std::string       s;

     while((nRead  =  pClient->CAsyncSocket::Receive(buf,sizeof(buf)))  >  0)
          {//异步读取
               s   +=     std::string((const  char*)buf,nRead);
          }
     }
}
```

要点如下：

1）多个客户端是通过 OnReceive 的第一个参数 pSender 进行区分的；

2）读取数据时使用的是 CAsyncSocket::Receive 函数，这个函数是异步读取数据的。

1.4.9　客户端下线

某个客户端下线时，CDlgTcpServerSync::OnClose 将被调用。代码如下：

```
void CDlgTcpServerSync::OnClose(CAsyncSocket*pSender,int nErrorCode)
{
    if(pSender)
    {
        CSocketSync*pClient = (CSocketSync*)pSender;
        MapClient::iterator it = m_mapClient.find(pClient);
        if(it != m_mapClient.end())
        {//从 m_mapClient 里删除该客户端
            m_mapClient.erase(pClient);
        }
        pClient->ShutDown(2);
        pClient->Close();
        delete pClient;
    }
}
```

要点如下：

1）从 m_mapClient 里删除该客户端；

2）销毁该客户端对应的 CSocketSync 对象。

1.4.10　停止监听

单击图 1.2的“停止监听”按钮，将执行如下代码：

```
CSocketSync*pClient = NULL;
for(MapClient::iterator it = m_mapClient.begin();
    it != m_mapClient.end();++it)
{//断开所有与客户端的 TCP 连接
    if(pClient = it->first)
    {
        pClient->ShutDown(2);
        pClient->Close();
        delete pClient;
    }
}
m_mapClient.clear();
//停止监听
if(m_SocketListen.m_hSocket != INVALID_SOCKET)
{
    m_SocketListen.ShutDown(2);
```

```
        m_SocketListen.Close();
}
```

要点如下：

1）遍历 m_mapClient，断开所有与客户端的 TCP 连接；

2）关闭监听套接字 m_SocketListen。

第 2 章　异步 TCP 通讯

同步通讯较为简单，但是执行效率较低。如：多个 TCP 客户端都是通过 GPRS 上网的，网速低得只有十几 KB/s。TCP 服务器给这些客户端发送数据时，将会等待很长时间，整个程序也可能会处于假死状态。

为了提高通讯效率，可以使用异步通讯。

2.1　异步通讯类

异步通讯类 CSocketAsync 派生自 CAsyncSocket，用于 TCP/UDP 的异步通讯。代码如下：

```
class CSocketAsync : public CAsyncSocket
{
protected:
    enum { UDP_MAX_PKG = 65507 };    //UDP 数据包最大字节数
    class SendToData
    {
    public:
        std::string sAddr;
        std::string sData;
    };
    ISocketEventHandler*m_pEventHandler;          //套接字事件处理器
    std::string       m_sSend;       //Send    函数缓存的发送数据
    std::list<SendToData>
                      m_lstSendTo;   //SendTo 函数缓存的发送数据
public:
    CSocketAsync(ISocketEventHandler*pEvent = NULL)
    {
        m_pEventHandler     =     pEvent;
    }
    void SetEventHandler(ISocketEventHandler*pEvent)
    {
        m_pEventHandler     =     pEvent;
    }
    /*************************************************************\
    转换 IP 地址格式
    szIP     [in]     字符串格式的 IP 地址。如：192.168.1.200
```

```
    nPort    [in]     端口，范围 [0,65535]。如：2001
    addr     [out]    地址
    \*********************************************************/
    static bool IpAddress(LPCTSTR szIP,UINT nPort
                          ,SOCKADDR_IN&addr)
    {
        USES_CONVERSION;
        memset(&addr,0,sizeof(addr));
        char*szIpA  =   T2A((LPTSTR)szIP);
        if(szIpA)
        {
            addr.sin_addr.s_addr    =   inet_addr(szIpA);
            if(addr.sin_addr.s_addr == INADDR_NONE)
            {
                LPHOSTENT lphost    =   gethostbyname(szIpA);
                if(lphost)
                {
                    addr.sin_addr.s_addr =
                        ((LPIN_ADDR)lphost->h_addr)->s_addr;
                }
                else
                {
                    return false;
                }
            }
        }
        else
        {
            addr.sin_addr.s_addr = htonl(INADDR_BROADCAST);
        }
        addr.sin_family =   AF_INET;
        addr.sin_port   =   htons((u_short)nPort);
        return true;
    }
public:
    //用于发送 TCP 数据
    virtual int Send(const void*pData,int nLen,int nFlags = 0)
    {
        if(pData)
        {
            if(nLen < 0)
            {
                nLen    =   strlen((const char*)pData);
            }
```

```
            if(nLen > 0)
            {
                if(m_sSend.empty())
                {//缓存数据 m_sSend 为空，发送数据
                    int n       =   0;  //单次发送的字节数
                    int nSum    =   0;  //发送的累计字节数
                    for(;;)
                    {
                        n   =   nLen - nSum;
      n   =   send(m_hSocket,(const char*)pData + nSum,n,0);
                        if(n > 0)
                        {
                            nSum    +=  n;
                            if(nSum >= nLen)
                            {
                                return nLen;
                            }
                        }
                        else if(n == SOCKET_ERROR)
                        {
                    if(WSAGetLastError() == WSAEWOULDBLOCK)
                            {//将数据加入缓存
  m_sSend += std::string((const char*)pData + nSum,nLen - nSum);
                                return nLen;
                            }
                            else
                            {
                                return nSum;
                            }
                        }
                    }
                }
                else
                {//缓存数据 m_sSend 不为空，直接将数据加入缓存
                    m_sSend += std::string((const char*)pData,nLen);
                    return nLen;
                }
            }
        }
        return 0;
    }
    //用于发送 UDP 数据
    int SendTo(const void* lpBuf, int nBufLen,UINT nHostPort
            , LPCTSTR lpszHostAddress = NULL, int nFlags = 0)
```

```
{
    SOCKADDR_IN sockAddr;
    if(IpAddress(lpszHostAddress,nHostPort,sockAddr))
    {
        return SendTo(lpBuf,nBufLen,(SOCKADDR*)&sockAddr
                ,sizeof(sockAddr), nFlags);
    }
    return 0;
}
//用于发送 UDP 数据
int SendTo(const void*pData,int nLen,const SOCKADDR*lpSockAddr
        ,int nSockAddrLen,int nFlags = 0)
{
    if(pData && lpSockAddr && nSockAddrLen > 0)
    {
        if(nLen < 0)
        {
            nLen    =    strlen((const char*)pData);
        }
        if(nLen > 0)
        {
            SendToData  data;
            if(m_lstSendTo.empty())
            {//无缓存数据，发送
                int n       =   0;  //单次发送的字节数
                int nSum    =   0;  //发送的累计字节数
                for(;;)
                {
                    n   =   nLen - nSum;    //待发送的字节数
                    if(n > UDP_MAX_PKG)
                    {//待发送的字节数不能太大
                        n   =   UDP_MAX_PKG;
                    }
                        n = CAsyncSocket::SendTo(
    (const char*)pData + nSum,n,lpSockAddr,nSockAddrLen,nFlags);
                    if(n > 0)
                    {
                        nSum  +=   n;   //累计
                        if(nSum >= nLen)
                        {//发送完毕
                            return nLen;
                        }
                    }
                    else if(n == SOCKET_ERROR)
```

```
                                {
                                    switch(GetLastError())
                                    {
                                  //case  WSAEMSGSIZE:  //超过  65507  字节
                                    case  WSAEWOULDBLOCK://操作被挂起
    data.sAddr.assign((const  char*)lpSockAddr,nSockAddrLen);      //地址
    data.sData.assign((const  char*)pData  +  nSum,nLen  -  nSum);    //数据
    m_lstSendTo.push_back(data);            //缓存地址和数据
                                        return  nLen;
                                    default:
                                        return  nSum;
                                    }
                                }
                            }
                        }
                        else
                        {//有缓存数据，直接缓存
    data.sAddr.assign((const  char*)lpSockAddr,nSockAddrLen);      //地址
    data.sData.assign((const  char*)pData,nLen);   //数据
    m_lstSendTo.push_back(data);          //缓存地址和数据
                            return  nLen;
                        }
                    }
                }
                return  0;
            }
protected:
            virtual  void  OnConnect(int  nErrorCode)
            {//连接上服务器了
                if(m_pEventHandler)
                {
                    m_pEventHandler->OnConnect(this,nErrorCode);
                }
            }
            virtual  void  OnReceive(int  nErrorCode)
            {//接收到数据
                if(m_pEventHandler)
                {
                    m_pEventHandler->OnReceive(this,nErrorCode);
                }
            }
            virtual  void  OnSend(int  nErrorCode)
            {//输出缓冲区空了，可以发送数据了
```

```
        if(m_sSend.length() > 0)
        {//发送缓存数据 m_sSend
            int n          =    0;              //单次发送的字节数
            int nSum       =    0;              //发送的累计字节数
            int nTotal     =    m_sSend.length();    //待发送的总字节数
            for(;;)
            {
                n    =    nTotal - nSum;   //待发送的字节数
                n    =    send(m_hSocket,m_sSend.c_str() + nSum,n,0);
                if(n > 0)
                {
                    nSum    +=    n;
                    if(nSum >= nTotal)
                    {
                        break;
                    }
                }
                else if(n == SOCKET_ERROR)
                {
                    //WSAGetLastError() == WSAEWOULDBLOCK
                    break;
                }
            }
            if(nSum > 0)
            {
                m_sSend =    m_sSend.substr(nSum);
            }
        }
        if(!m_lstSendTo.empty())
        {//发送缓存数据 m_lstSendTo
            for(std::list<SendToData>::iterator it = m_lstSendTo.begin();
                    it != m_lstSendTo.end();)
            {
                if(DoSendToData(*it))
                {
                    it   =    m_lstSendTo.erase(it);
                }
                else
                {
                    break;
                }
            }
        }
    }
```

```
    virtual void OnClose(int nErrorCode)
    {//连接断了
        if(m_pEventHandler)
        {
            m_pEventHandler->OnClose(this,nErrorCode);
        }
    }
protected:
    //发送一包数据
    bool DoSendToData(SendToData&data)
    {
        int nTotal  =   data.sData.length();    //总字节数
        int nSum    =   0;  //发送字节数的累计值
        int n       =   0;  //单次发送的字节数
        for(;;)
        {
            n   =   nTotal - nSum;
            if(n <= 0)
            {
                return true;    //这一包数据发送完毕了
            }
            if(n > UDP_MAX_PKG)
            {//每次发送的字节数不能过大
                n   =   UDP_MAX_PKG;
            }
            n   =   sendto(m_hSocket,data.sData.c_str() + nSum,n,0
                        ,(const struct sockaddr*)data.sAddr.c_str()
                        ,data.sAddr.length());
            if(n > 0)
            {
                nSum  +=   n;
            }
            else if(n == SOCKET_ERROR)
            {
                data.sData  =   data.sData.substr(nSum);
                //WSAGetLastError() == WSAEWOULDBLOCK
                break;
            }
        }
        return false;   //这一包数据没有发送完毕
    }
};
```

上述代码量是同步通讯类 CSocketSync 的近四倍。具体含义下文进行说明。

2.2　异步 TCP 通讯客户端

同步 TCP 通讯客户端使用的是 CSocketSync m_Socket；异步 TCP 通讯客户端使用的是 CSocketAsync m_Socket。两者的用法基本相同，不同之处如下：

2.2.1　连接服务器

单击图 1.1的“连接”按钮，将连接服务器。示例代码如下：

```
if(m_Socket.Socket())
{//创建套接字成功
    CString sIP      = _T("127.0.0.1");   //服务器 IP
    int     nPort    = 2001;              //服务器端口
    if(m_Socket.Connect(sIP,nPort))
    {//连接服务器成功
        OnConnect(&m_Socket,0);
    }
    else if(GetLastError() == WSAEWOULDBLOCK)
    {//连接操作被挂起，连接操作完成时会调用 OnConnect 函数
    }
    else
    {
        m_Socket.Close();
        SetDlgItemText(IDC_TXT_LOCAL,_T("连接失败"));
    }
}
```

```
void CDlgTcpClientAsync::OnConnect(CAsyncSocket*pSender
                                   ,int nErrorCode)
{//连接完成时的回调函数
    if(0 == nErrorCode)
    {//连接成功
    }
    else
    {//连接失败
    }
}
```

重点在于：

1）m_Socket.Connect(sIP,nPort)返回 TRUE，表示连接成功；

2）m_Socket.Connect(sIP,nPort)返回 FALSE，GetLastError() == WSAEWOULDBLOCK 表示连接操作被挂起。当连接操作完成时，会调用 CDlgTcpClient

Async::OnConnect 函数，该函数的第二个参数 nErrorCode 用来说明连接是否成功。

m_Socket.Connect 返回时，连接操作可能并没有完成，这就是异步操作。

2.2.2　写数据

异步写数据的代码与同步写数据的代码完全相同，如下所示：

```
std::string  s(1024,,'1');
m_Socket.Send(s.c_str(),s.length());
```

不过上面的 m_Socket.Send 调用的是 CSocketAsync::Send 函数。这个函数有些复杂，其精简代码如下：

```
virtual  int  Send(const  void*pData,int  nLen,int  nFlags  =  0)
{
    int  n          =    0;   //单次发送的字节数
    int  nSum     =    0;   //发送的累计字节数
    for(;;)
    {
        n     =     nLen  -  nSum;
        n     =     send(m_hSocket,(const  char*)pData  +  nSum,n,0);
        if(n  >  0)
        {
            nSum     +=     n;
            if(nSum  >=  nLen)
            {
                return  nLen;
            }
        }
        else  if(n  ==  SOCKET_ERROR)
        {
            if(WSAGetLastError()  ==  WSAEWOULDBLOCK)
            {//将数据加入缓存
                m_sSend  +=  std::string((const  char*)pData  +  nSum
                                         ,nLen  -  nSum);
                return  nLen;
            }
            else
            {
                return  nSum;
            }
        }
    }
```

```
}
```

重点如下：

1）send 函数的返回值大于零，表明发送成功了一些数据；

2）send 函数的返回值等于 SOCKET_ERROR 且 WSAGetLastError() == WSAEWOULDBLOCK，说明该写操作被挂起了。send 函数发送数据的实质是给套接字输出缓冲区内填入数据，当输出缓冲区满了，无法填入数据时就会这种情况，此即为写操作被挂起。那么，何时才能继续发送数据呢？当输出缓冲区为空时，系统会给套接字发送 OnSend 事件，在这里继续发送数据。代码如下：

```
virtual void OnSend(int nErrorCode)
{//输出缓冲区空了，可以发送数据了
    if(m_sSend.length() > 0)
    {//发送缓存数据 m_sSend
        int n = 0;                    //单次发送的字节数
        int nSum = 0;                 //发送的累计字节数
        int nTotal = m_sSend.length();  //待发送的总字节数
        for(;;)
        {
            n = nTotal - nSum;  //待发送的字节数
            n = send(m_hSocket,m_sSend.c_str() + nSum,n,0);
            if(n > 0)
            {
                nSum += n;
                if(nSum >= nTotal)
                {
                    break;
                }
            }
            else if(n == SOCKET_ERROR)
            {//输出缓冲区又满了，停止发送
                //WSAGetLastError() == WSAEWOULDBLOCK
                break;
            }
        }
        if(nSum > 0)
        {//舍弃掉已经发送的缓存数据
            m_sSend = m_sSend.substr(nSum);
        }
    }
}
```

上面的代码调用 send 函数把缓存数据 std::string m_sSend 发送出去。发送时输出缓冲区再次满的时候就停止发送。等输出缓冲区再次为空时，会再次调用 OnSend 函数，继续发送缓存数据……

可见：异步写数据需要缓存发送失败的数据，并且在 OnSend 函数里发送这些缓存数据。

2.3　异步 TCP 通讯服务端

“异步 TCP 通讯服务端”与“同步 TCP 通讯服务端”的不同之处在于 m_mapClient 的定义：

```
std::map<CSocketAsync*,CString>  m_mapClient;   //异步
std::map<CSocketSync*,CString>   m_mapClient;   //同步
```

与客户端的通讯，异步使用的是 CSocketAsync，同步使用的是 CSocketSync。其余代码基本相同。

第 3 章　同步 UDP 通讯

3.1　界面

UDP 通讯界面如下

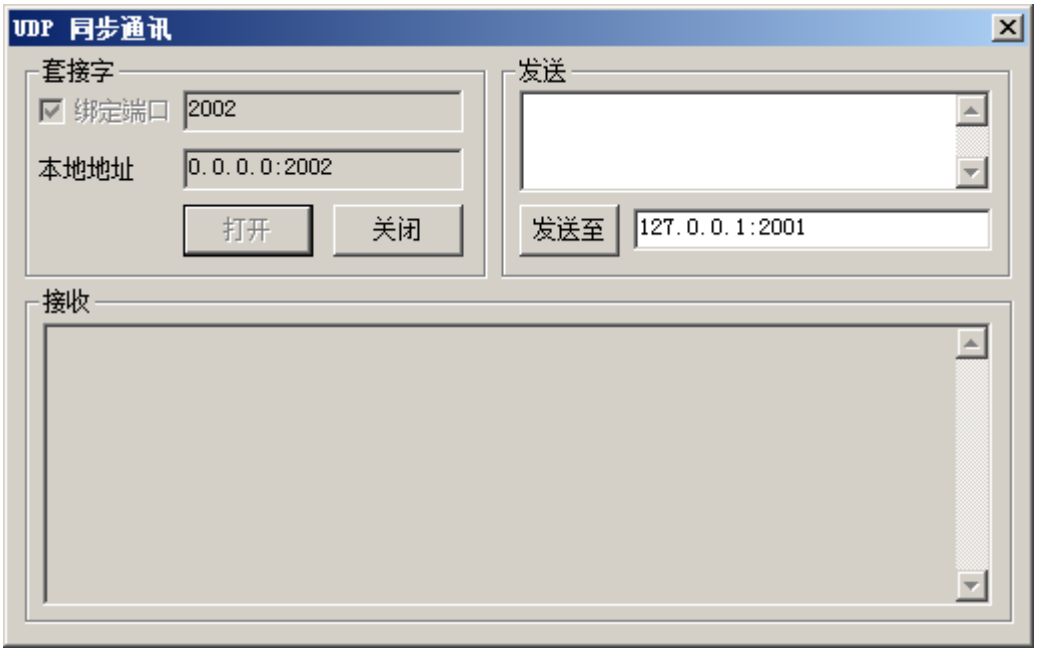

图 3.1

3.2　界面类声明

上面的界面，对应于类 CDlgTcpClientSync，其声明代码如下：

```
class CDlgUdpSync : public CDialog , public ISocketEventHandler
{
    ...    ...    ...
protected://ISocketEventHandler
    virtual void OnReceive(CAsyncSocket*pSender,int nErrorCode);
protected:
    CSocketSync m_Socket;   //套接字
    CString     m_sRecv;    //接收到的数据
};
```

上述代码的重点：

1）定义了 CSocketSync m_Socket，用于套接字通讯；

2）成员变量 std::string m_sRecv 用来存储接收到的数据；

3）CDlgUdpSync 继承了 ISocketEventHandler，并重写了 OnReceive 函数。m_Socket 的套接字事件，将由 OnReceive 函数来处理。

3.3　界面类构造函数

界面类构造函数如下

```
CDlgUdpSync::CDlgUdpSync(CWnd*  pParent  /*=NULL*/)
:  CDialog(CDlgUdpSync::IDD,  pParent)
{
      m_Socket.SetEventHandler(this);
}
```

设置 m_Socket 的事件处理器为 this。其含义为：m_Socket 的 OnReceive 被调用时，就会调用函数 CDlgUdpSync::OnReceive。

3.4　创建套接字

单击图 3.1中的“打开”按钮，将创建UDP套接字。示例代码如下：

```
BOOL  bOK      =     FALSE;
if(((CButton*)GetDlgItem(IDC_CHK_BIND))->GetCheck())
{//绑定某个端口
      int  nPort     =     2001;
      bOK  =    m_Socket.Create(nPort,SOCK_DGRAM);  //创建套接字并绑定
}
else
{//不绑定
      bOK  =     m_Socket.Socket(SOCK_DGRAM);    //创建套接字，不绑定
}
if(bOK)
{//创建套接字成功
}
else
{//创建套接字失败
      m_Socket.Close();
}
```

调用 m_Socket.Socket 或 m_Socket.Create 创建套接字，两者的区别在于：前者不绑定端口，后者绑定端口。

3.5　写数据

单击图 3.1中的“发送”按钮，即可给“127.0.0.1:2001”发送数据，其代码如下：

```
std::string  s(1024,'1');
CString      sIP        =     _T("127.0.0.1");
UINT         nPort      =     2001;
m_Socket.SendTo(s.c_str(),s.length(),nPort,sIP);
```

3.6　读数据

程序一旦接收到对方发来的数据，函数 CDlgTcpClientSync::OnReceive 将被调用。其代码如下：

```
void  CDlgUdpSync::OnReceive(CAsyncSocket*pSender,int  nErrorCode)
{
      char              buf[64  *  1024];
      int               nRead     =     0;
      CString           sIP;
      UINT              nPort     =     0;
      CString           sFrom;
      std::map<CString,std::string>
                         mapRecv;
      SOCKADDR_IN  sockAddr;
      int               nAddrLen       =     sizeof(sockAddr);
      memset(&sockAddr,0,sizeof(sockAddr));

      while((nRead  =  recvfrom(m_Socket.m_hSocket,buf,sizeof(buf),0
                              ,(struct  sockaddr*)&sockAddr,&nAddrLen))  >  0)
      {
            nPort     =     ntohs(sockAddr.sin_port);        //对方的端口
            sIP       =     inet_ntoa(sockAddr.sin_addr);  //对方的 IP
            sFrom.Format(_T("%s:%d"),sIP,nPort);
            mapRecv[sFrom]  +=     std::string((const  char*)buf,nRead);
      }
      if(!mapRecv.empty())
      {
```

```
            for(std::map<CString,std::string>::iterator it = mapRecv.begin();
                it != mapRecv.end();++it)
            {
                const std::string&s =    it->second;
                m_sRecv += it->first + _T(" > ")
                        + CString(s.c_str(),s.length()) + _T("\r\n");
            }
            if(m_sRecv.GetLength() > MAXRECV)
            {
                m_sRecv =    m_sRecv.Right(MAXRECV);
            }
            SetEditText(::GetDlgItem(m_hWnd,IDC_TXT_RECV),m_sRecv);
        }
}
```

重点在于：

1）调用 recvfrom 函数异步读取 UDP 数据包；

2）buf 要足够大，否则无法读取一包 UDP 数据。一包 UDP 数据最多 64KB，所以这里的 buf 也设置为 64KB；

3）读取到一包 UDP 数据后，即可获得对方的 IP 地址、端口号；

4）UDP 数据包可能是由不同的终端发送过来的，其 IP 地址、端口号不尽相同，因此使用 std::map<CString,std::string> mapRecv;来存储读取到的 UDP 数据；

5）最后一段代码把mapRecv里的数据显示到图 3.1中的“接收”文本框内。

3.7　关闭套接字

单击图 3.1中的“关闭”按钮，将关闭套接字。代码如下：

```
m_Socket.Close();
```

第 4 章　异步 UDP 通讯

“异步 UDP 通讯”与“同步 UDP 通讯”的写数据代码是相同的，如下：

```
std::string  s(1024,'1');
CString      sIP        =     _T("127.0.0.1");
UINT         nPort      =     2001;
m_Socket.SendTo(s.c_str(),s.length(),nPort,sIP);
```

不过上面的 m_Socket.SendTo 调用的是 CSocketAsync::SendTo 函数。这个函数与 CSocketAsync::Send 类似，也会将挂起的写数据缓存起来（std::list<SendToData> m_lstSendTo）。在输出缓冲区为空，系统调用 CSocketAsync::OnSend 函数时，将缓存数据发送出去。