如何通过 socket 实现网络通信

为了方便网络编程，90 年代初，由 Microsoft 联合了其他几家公司共同制定了一套 WINDOWS 下的网络编程接口，即 Windows Sockets 规范，它不是一种网络协议,而是一套开放的、支持多种协议的 Windows 下的网络编程接口。现在的 Winsock 已经基本上实现了与协议无关，你可以使用 Winsock 来调用多种协议的功能，但较常使用的是 TCP/IP 协议。Socket 实际在计算机中提供了一个通信端口，可以通过这个端口与任何一个具有 Socket 接口的计算机通信。应用程序在网络上传输，接收的信息都通过这个 Socket 接口来实现。

先介绍几个基本概念，同步(Sync)/异步(Async)，阻塞(Block)/非阻塞(Unblock)。同步方式指的是发送方不等接收方响应，便接着发下个数据包的通信方式；而异步指发送方发出数据后，等收到接收方发回的响应，才发下一个数据包的通信方式。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，比如调用 recv()函数读取网络缓冲区中的数据，如果没有数据到达，将一直挂在 recv()这个函数调用上，直到读到一些数据，此函数调用才返回；而非阻塞套接字是指执行此套接字的网络调用时，不管是否执行成功，都立即返回。比如调用 recv()函数读取网络缓冲区中数据，不管是否读到数据都立即返回，而不会一直挂在此函数调用上。在实际 Windows 网络通信软件开发中，异步非阻塞套接字是用的最多的。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。

创建 TCP 通信的过程及相关函数

服务器端

一、创建服务器套接字（socket）。
二、服务器套接字进行信息绑定（bind），并开始监听连接（listen）。
三、接受来自用户端的连接请求（accept）。
四、开始数据传输(send/receive)。
五、关闭套接字（closesocket）。

客户端

一、创建用户套接字（socket）。
二、与远程服务器进行连接（connect），如被接受则创建接收进程。
三、开始数据传输(send/receive)。
四、关闭套接字（closesocket）。

微软为 VC 定义了 Winsock 类如 CAsyncSocket 类和派生于 CAsyncSocket 的 CSocket 类，它们简单易用，可以使用这些类来实现自己的网络程序，但是为了更好的了解 Winsock API 编程技术，我们这里探讨怎样使用底层的 API 函数实现简单的 Winsock 网络应用程序设计，分别说明如何在 Server 端和 Client 端操作 Socket，实现基于 TCP/IP 的数据传送，最后给出部分源代码。

在 VC 中进行 WINSOCK 的 API 编程开发的时候，需要在项目中使用下面三个文件，否则会出现编译错误。

1. WINSOCK.H: 这是 WINSOCK API 的头文件，需要包含在项目中。可在 stdafx.h 中加入#include "winsock2.h"。

2. WSOCK32.LIB: WINSOCK API 连接库文件。在使用中，一定要把它作为项目的非缺省的连接库包含到项目文件中去。打开选择菜单 Project->Setting (ALT+F7),进入 Project Setting 对话框,在 Link 下的 Object/library modules 输入 ws2_32.lib，然后点 OK，或者在头文件中添加：#pragma comment(lib, "ws2_32.lib")。

3. WINSOCK.DLL: WINSOCK 的动态连接库，位于 WINDOWS 的安装目录下。

一、服务器端操作 socket（套接字）

1)在初始化阶段调用 WSAStartup()

此函数在应用程序中初始化 Windows Sockets DLL ，只有此函数调用成功后，应用程序才可以再调用其他 Windows Sockets DLL 中的 API 函数。在程式中调用该函数的形式如下：WSAStartup(0x0202，（LPWSADATA）&WSAData)，其中 0x0202 表示我们用的是 WinSocket2.0 版本，WSAata 用来存储系统传回的关于 WinSocket 的资料。

2)建立 Socket

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，为此可以调用 Socket()函数用来建立这个监听的 Socket,并定义此 Socket 所使用的通信协议。此函数调用成功返回 Socket 对象，失败则返回 INVALID_SOCKET(调用 WSAGetLastError()可得知原因，所有 WinSocket 的函数都可以使用这个函数来获取失败的原因)。

SOCKET PASCAL FAR socket(int af, int type, int protocol)
参数: af:目前只提供 PF_INET(AF_INET)；
type：Socket 的类型 (SOCK_STREAM、SOCK_DGRAM)；
protocol：通讯协定(如果使用者不指定则设为 0)；

如果要建立的是遵从 TCP/IP 协议的 socket,第二个参数 type 应为 SOCK_STREAM，如为 UDP（数据报）的 socket，应为 SOCK_DGRAM。

3)绑定端口

接下来要为服务器端定义的这个监听的 Socket 指定一个地址及端口

（Port），这样客户端才知道待会要连接哪一个地址的哪个端口，为此我们要调用 bind()函数，该函数调用成功返回 0，否则返回 SOCKET_ERROR。
int PASCAL FAR bind(SOCKET s, const struct sockaddr FAR *name,int namelen);

参 数： s：Socket 对象名，即通过 Socket 函数创建的 Socket 对象；
name：Socket 的地址值，这个地址必须是执行这个程式所在机器的 IP 地址，这个地址为地址结构，其中包含了本机的 IP 地址和监听端口号；
namelen：name 的长度，即地址结构的长度；

如果使用者不在意地址或端口的值，那么可以设定地址为 INADDR_ANY，及 Port 为 0，Windows Sockets 会自动将其设定适当之地址及 Port (1024 到 5000 之间的值)。此后可以调用 getsockname()函数来获知其被设定的值。

4）监听

当服务器端的 Socket 对象绑定完成之后,服务器端必须建立一个监听的队列来接收客户端的连接请求。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(目前最大值限制为 5, 最小值为 1)。该函数调用成功返回 0，否则返回 SOCKET_ERROR。

int PASCAL FAR listen(SOCKET s, int backlog);
参 数： s：需要建立监听的 Socket；
backlog：最大连接个数；

服务器端的 Socket 调用完 listen（）后，如果此时客户端调用 connect（）函数提出连接申请的话，Server 端必须再调用 accept() 函数，这样服务器端和客户端才算正式完成通信程序的连接动作。为了知道什么时候客户端提出连接要求,从而服务器端的 Socket 在恰当的时候调用 accept()函数完成连接的建立，我们就要使用 WSAAsyncSelect（）函数，让系统主动来通知我们有客户端提出连接请求了。该函数调用成功返回 0，否则返回 SOCKET_ERROR。

int PASCAL FAR WSAAsyncSelect(SOCKET s, HWND hWnd,unsigned int wMsg, long lEvent);

参数： s：Socket 对象；
hWnd ：接收消息的窗口句柄；
wMsg：传给窗口的消息；
lEvent：被注册的网络事件，也即是应用程序向窗口发送消息的网路事件，该值为下列值 FD_READ、FD_WRITE、FD_OOB、FD_ACCEPT、FD_CONNECT、FD_CLOSE 的组合，各个值的具体含意为 FD_READ：希望在套接字 S 收到数据时收到消息；FD_WRITE：希望在套接字 S 上可以发送数据时收到消息；FD_ACCEPT：希望在套接字 S 上收到连接请求时收到消息；FD_CONNECT：希望在套接字 S 上连接成功时收到消息；FD_CLOSE：希望在套接字 S 上连接关闭时收到消息；FD_OOB：希望在套接字 S 上收到带外数据时收到消息。

具体应用时，wMsg 应是在应用程序中定义的消息名称，而消息结构中的 lParam 则为以上各种网络事件名称。所以，可以在窗口处理自定义消息函数中使用以下结构来响应 Socket 的不同事件：

```
switch(lParam)

{

case FD_READ:
       …
   break;
case FD_WRITE、
       …
   break;
       …
}
```

5）服务器端接受客户端的连接请求

当 Client 提出连接请求时，Server 端 hwnd 视窗会收到 Winsock Stack 送来我们自定义的一个消息，这时，我们可以分析 lParam，然后调用相关的函数来处理此事件。为了使服务器端接受客户端的连接请求，就要使用 accept() 函数，该函数新建一 Socket 与客户端的 Socket 相通，原先监听之 Socket 继续进入监听状态，等待他人的连接要求。该函数调用成功返回一个新产生的 Socket 对象，否则返回 INVALID_SOCKET。

```
SOCKET PASCAL FAR accept( SCOKET s, struct sockaddr FAR *addr,int FAR *addrlen );
```

参数：s：Socket 的识别码；
addr：存放来连接的客户端的地址；
addrlen：addr 的长度

6）结束 socket 连接

结束服务器和客户端的通信连接是很简单的，这一过程可以由服务器或客户机的任一端启动，只要调用 closesocket()就可以了，而要关闭 Server 端监听状态的 socket，同样也是利用此函数。另外，与程序启动时调用 WSAStartup() 憨数相对应，程式结束前，需要调用 WSACleanup() 来通知 Winsock Stack 释放 Socket 所占用的资源。这两个函数都是调用成功返回 0，否则返回 SOCKET_ERROR。

```
int PASCAL FAR closesocket( SOCKET s );
```

参 数：s：Socket 的识别码；

```
int PASCAL FAR WSACleanup( void );
```

参 数： 无

二、客户端 Socket 的操作

1）建立客户端的 Socket

客户端应用程序首先也是调用 WSAStartup() 函数来与 Winsock 的动态连接库建立关系，然后同样调用 socket() 来建立一个 TCP 或 UDP socket（相同协定的 sockets 才能相通，TCP 对 TCP，UDP 对 UDP）。与服务器端的 socket 不同的是,客户端的 socket 可以调用 bind() 函数,由自己来指定 IP 地址及 port 号码；但是也可以不调用 bind()，而由 Winsock 来自动设定 IP 地址及 port 号码。

2）提出连接申请

客户端的 Socket 使用 connect()函数来提出与服务器端的 Socket 建立连接的申请，函数调用成功返回 0，否则返回 SOCKET_ERROR。

int PASCAL FAR connect(SOCKET s, const struct sockaddr FAR *name, int namelen);
参 数：s：Socket 的识别码；
name：Socket 想要连接的对方地址；
namelen：name 的长度

三、数据的传送

虽然基于 TCP/IP 连接协议（流套接字）的服务是设计客户机/服务器应用程序时的主流标准，但有些服务也是可以通过无连接协议（数据报套接字）提供的。先介绍一下 TCP socket 与 UDP socket 在传送数据时的特性：Stream (TCP) Socket 提供双向、可靠、有次序、不重复的资料传送。Datagram (UDP) Socket 虽然提供双向的通信，但没有可靠、有次序、不重复的保证，所以 UDP 传送数据可能会收到无次序、重复的资料，甚至资料在传输过程中出现遗漏。由于 UDP Socket 在传送资料时，并不保证资料能完整地送达对方，所以绝大多数应用程序都是采用 TCP 处理 Socket，以保证资料的正确性。一般情况下 TCP Socket 的数据发送和接收是调用 send() 及 recv() 这两个函数来达成，而 UDP Socket 则是用 sendto() 及 recvfrom() 这两个函数，这两个函数调用成功返回发送或接收的资料的长度，否则返回 SOCKET_ERROR。

int PASCAL FAR send(SOCKET s, const char FAR *buf,int len, int flags);
参数：s：Socket 的识别码
buf：存放要传送的资料的暂存区
len buf：的长度
flags：此函数被调用的方式

对于 Datagram Socket 而言，若是 datagram 的大小超过限制，则将不会送出任何资料，并会传回错误值。对 Stream Socket 言，Blocking 模式下，若是

传送系统内的储存空间不够存放这些要传送的资料，send()将会被 block 住，直到资料送完为止；如果该 Socket 被设定为 Non-Blocking 模式，那么将视目前的 output buffer 空间有多少，就送出多少资料，并不会被 block 住。flags 的值可设为 0 或 MSG_DONTROUTE 及 MSG_OOB 的组合。

int PASCAL FAR recv(SOCKET s, char FAR *buf, int len, int flags);
参数：s：Socket 的识别码
buf：存放接收到的资料的暂存区
len buf：的长度
flags：此函数被调用的方式

对 Stream Socket 言，我们可以接收到目前 input buffer 内有效的资料，但其数量不超过 len 的大小。

四、自定义的 CMySocket 类的实现代码：

根据上面的知识，我自定义了一个简单的 CMySocket 类，下面是我定义的该类的部分实现代码：

```
    ////////////////////////////////////
CMySocket::CMySocket() : //file://类的构造函数
{
  WSADATA wsaD;
  memset( m_LastError, 0, ERR_MAXLENGTH );
  // m_LastError 是类内字符串变量,初始化用来存放最后错误说明的字符串;
  // 初始化类内 sockaddr_in 结构变量，前者存放客户端地址，后者对应于服务器端地址;
  memset( &m_sockaddr, 0, sizeof( m_sockaddr ) );
  memset( &m_rsockaddr, 0, sizeof( m_rsockaddr ) );
  int result = WSAStartup(0x0202,  &wsaD);//初始化 WinSocket 动态连接库;
  if( result != 0 ) // 初始化失败;
  { set_LastError( "WSAStartup failed!", WSAGetLastError() );
    return;
  }
}

////////////////////////////
CMySocket::~CMySocket() { WSACleanup(); }//类的析构函数;
//////////////////////////////////////////////////
int CMySocket::Create( void )
  {// m_hSocket 是类内 Socket 对象，创建一个基于 TCP/IP 的 Socket 变量，并将值赋给该变量;
    if ( (m_hSocket = socket( AF_INET, SOCK_STREAM, IPPROTO_TCP )) == INVALID_SOCKET )
    {
      set_LastError( "socket() failed", WSAGetLastError() );
      return ERR_WSAERROR;
    }
```

```
    return ERR_SUCCESS;
  }
//////////////////////////////////////////////
int CMySocket::Close( void )//关闭 Socket 对象;
{
  if ( closesocket( m_hSocket ) == SOCKET_ERROR )
  {
    set_LastError( "closesocket() failed", WSAGetLastError() );
    return ERR_WSAERROR;
  }
  //file://重置 sockaddr_in 结构变量;
  memset( &m_sockaddr, 0, sizeof( sockaddr_in ) );
  memset( &m_rsockaddr, 0, sizeof( sockaddr_in ) );
  return ERR_SUCCESS;
}
//////////////////////////////////////////
int CMySocket::Connect( char* strRemote, unsigned int iPort )//定义连接函数;
{
  if( strlen( strRemote ) == 0 || iPort == 0 )
    return ERR_BADPARAM;
  hostent *hostEnt = NULL;
  long lIPAddress = 0;
  hostEnt = gethostbyname( strRemote );//根据计算机名得到该计算机的相关内容;
  if( hostEnt != NULL )
  {
    lIPAddress = ((in_addr*)hostEnt->h_addr)->s_addr;
    m_sockaddr.sin_addr.s_addr = lIPAddress;
  }
  else
  {
    m_sockaddr.sin_addr.s_addr = inet_addr( strRemote );
  }
  m_sockaddr.sin_family = AF_INET;
  m_sockaddr.sin_port = htons( iPort );
  if( connect( m_hSocket, (SOCKADDR*)&m_sockaddr, sizeof( m_sockaddr ) ) == SOCKET_ERROR )
  {
    set_LastError( "connect() failed", WSAGetLastError() );
    return ERR_WSAERROR;
  }
  return ERR_SUCCESS;
}
//////////////////////////////////////////////////////
int CMySocket::Bind( char* strIP, unsigned int iPort )//绑定函数;
{
```

```
  if( strlen( strIP ) == 0 || iPort == 0 )
    return ERR_BADPARAM;
  memset( &m_sockaddr,0, sizeof( m_sockaddr ) );
  m_sockaddr.sin_family = AF_INET;
  m_sockaddr.sin_addr.s_addr = inet_addr( strIP );
  m_sockaddr.sin_port = htons( iPort );
  if ( bind( m_hSocket, (SOCKADDR*)&m_sockaddr, sizeof( m_sockaddr ) ) == SOCKET_ERROR )
  {
    set_LastError( "bind() failed", WSAGetLastError() );
    return ERR_WSAERROR;
  }
  return ERR_SUCCESS;
}
////////////////////////////////////////
int CMySocket::Accept( SOCKET s )//建立连接函数，S 为监听 Socket 对象名；
{
  int Len = sizeof( m_rsockaddr );
  memset( &m_rsockaddr, 0, sizeof( m_rsockaddr ) );
  if( ( m_hSocket = accept( s, (SOCKADDR*)&m_rsockaddr, &Len ) ) == INVALID_SOCKET )
  {
    set_LastError( "accept() failed", WSAGetLastError() );
    return ERR_WSAERROR;
  }
  return ERR_SUCCESS;
}
////////////////////////////////////////////////
int CMySocket::asyncSelect( HWND hWnd, unsigned int wMsg, long lEvent )
//file://事件选择函数；
{
  if( !IsWindow( hWnd ) || wMsg == 0 || lEvent == 0 )
    return ERR_BADPARAM;
  if( WSAAsyncSelect( m_hSocket, hWnd, wMsg, lEvent ) == SOCKET_ERROR )
  {
    set_LastError( "WSAAsyncSelect() failed", WSAGetLastError() );
    return ERR_WSAERROR;
  }
  return ERR_SUCCESS;
}
////////////////////////////////////////////////
int CMySocket::Listen( int iQueuedConnections )//监听函数；
{
  if( iQueuedConnections == 0 )
    return ERR_BADPARAM;
  if( listen( m_hSocket, iQueuedConnections ) == SOCKET_ERROR )
```

```
  {
    set_LastError( "listen() failed", WSAGetLastError() );
    return ERR_WSAERROR;
  }
  return ERR_SUCCESS;
}
////////////////////////////////////////////////
int CMySocket::Send( char* strData, int iLen )//数据发送函数;
{
  if( strData == NULL || iLen == 0 )
    return ERR_BADPARAM;
  if( send( m_hSocket, strData, iLen, 0 ) == SOCKET_ERROR )
  {
    set_LastError( "send() failed", WSAGetLastError() );
    return ERR_WSAERROR;
  }
  return ERR_SUCCESS;
}
////////////////////////////////////////////////
int CMySocket::Receive( char* strData, int iLen )//数据接收函数;
{
  if( strData == NULL )
    return ERR_BADPARAM;
  int len = 0;
  int ret = 0;
  ret = recv( m_hSocket, strData, iLen, 0 );
  if ( ret == SOCKET_ERROR )
  {
    set_LastError( "recv() failed", WSAGetLastError() );
    return ERR_WSAERROR;
  }
  return ret;
}
void CMySocket::set_LastError( char* newError, int errNum )
//file://WinSock API 操作错误字符串设置函数;
{
  memset( m_LastError, 0, ERR_MAXLENGTH );
  memcpy( m_LastError, newError, strlen( newError ) );
  m_LastError[strlen(newError)+1] = '\0';
}
```

有了上述类的定义，就可以在网络程序的服务器和客户端分别定义CMySocket对象，建立连接，传送数据了。例如，为了在服务器和客户端发送数

据，需要在服务器端定义两个 CMySocket 对象 ServerSocket1 和 ServerSocket2，分别用于监听和连接，客户端定义一个 CMySocket 对象 ClientSocket，用于发送或接收数据，如果建立的连接数大于一，可以在服务器端再定义 CMySocket 对象，但要注意连接数不要大于五。

由于 Socket API 函数还有许多，如获取远端服务器、本地客户机的 IP 地址、主机名等等，读者可以再此基础上对 CMySocket 补充完善，实现更多的功能。