

第一章 网络模块

1.1 网络编程概述/基本 socket api

很多网络游戏通讯都是客户端/服务器结构。简写 Client/Server 或是 c/s 。
c/s 架构允许 N 个 Client 连接 1 个或多个 Server 来进行通讯。
做网页的朋友就会提出 还有 b/s 架构。其实 b/s 是在以 tcp 为通讯基础的数据封装协议（web 协议）。底层还是用的 tcp。

通常大部分 Server 都是基于 tcp 协议的(Transmission Control Protocol 传输控制协议) 是一种面向连接的、可靠的、基于字节流的传输层通信协议)
比如网络游戏，web 上网看视频，语音文字聊天 99%都是 tcp 协议通讯的，而且是 c/s 架构的。

基于 udp 协议的 Server 比较少，而且 udp 协议(User Datagram Protocol 提供面向事务的简单不可靠信息传送服务) 由于传输不可靠，所以在使用上需要再封装上一些技术来保证消息传输的可靠性，编码难度上升不少
udp 通常用于 p2p 架构(比如 bt 下载)，内网传输视频广播会议等。

本书大部分都是着重讲解 tcp 协议的通讯为主。

1.2 网络通信协议

开始需要先了解网络通信协议，都是一些相关知识，还是有必要了解一下的。

1.2.1 网络协议

要交换信息，必定有相关的规则，这些规则的集合就称为协议（Protocol）
那么常用的网络协议 TCP/IP 就是一个协议的集合。

1.2.2 ISO/OSI 七层参考模型

OSI 七层参考模型是 ISO 国际标准化组织制定的一个逻辑上的定义，一个规范，它把网络从逻辑上分为了 7 层。



OSI 的 7 层从上到下分别是

- 7 应用层
- 6 表示层
- 5 会话层
- 4 传输层
- 3 网络层
- 2 数据链路层
- 1 物理层

其中高层，即 7、6、5、4 层定义了应用程序的功能，下面 3 层，即 3、2、1 层主要面向通过网络的端到端的数据流。

OSI 七层参考模型 **仅供参考**，虽然它设计的很好，可惜出炉的太晚，我们实际网络通信中用的还是 TCP/IP 协议。

1.2.3 TCP/IP 协议

网络通讯协议，是 Internet 最基本的协议、Internet 国际互联网络的基础，由网络层的 IP 协议和传输层的 TCP 协议组成。

TCP/IP 定义了电子设备如何连入因特网，以及数据如何在它们之间传输的标准。协议采用了 4 层的层级结构：**网络访问层、互联网层、传输层和应用层**



网络访问层(Network Access Layer)在 TCP/IP 参考模型中并没有详细描述,只是指出主机必须使用某种协议与网络相连。

互联网层(Internet Layer)是整个体系结构的关键部分,其功能是使主机可以把分组发往任何网络,并使分组独立地传向目标。这些分组可能经由不同的网络,到达的顺序和发送的顺序也可能不同。高层如果需要顺序收发,那么就必须自行处理对分组的排序。互联网层使用因特网协议(IP, Internet Protocol)。TCP/IP 参考模型的互联网层和 OSI 参考模型的网络层在功能上非常相似。

传输层(Transport Layer)使源端和目的端机器上的对等实体可以进行会话。在这一层定义了两个端到端的协议:传输控制协议(TCP, Transmission Control Protocol)和用户数据报协议(UDP, User Datagram Protocol)。TCP 是面向连接的协议,它提供可靠的报文传输和对上层应用的连接服务。为此,除了基本的数据传输外,它还有可靠性保证、流量控制、多路复用、优先权 and 安全性控制等功能。UDP 是面向无连接的不可靠传输的协议,主要用于不需要 TCP 的排序和流量控制等功能的应用程序。

应用层(Application Layer)包含所有的高层协议,包括:文件传输协议(FTP, File Transfer Protocol)、电子邮件传输协议(SMTP, Simple Mail Transfer Protocol)、域名服务(DNS, Domain Name Service)和超文本传送协议(HTTP, HyperText Transfer Protocol)等。FTP 提供有效地将文件从一台机器上移到另一台机器上的方法;SMTP 用于电子邮件的收发;DNS 用于把主机名映射到网络地址;HTTP 用于在 WWW 上获取主页。

TCP/IP 结构对应 OSI

TCP/IP	OSI
应用层	应用层 表示层 会话层
主机到主机层（TCP）（又称传输层）	传输层
网络层（IP）(又称互联层)	网络层
网络接口层（又称链路层）	数据链路层 物理层

1.2.4 TCP/IP 协议族

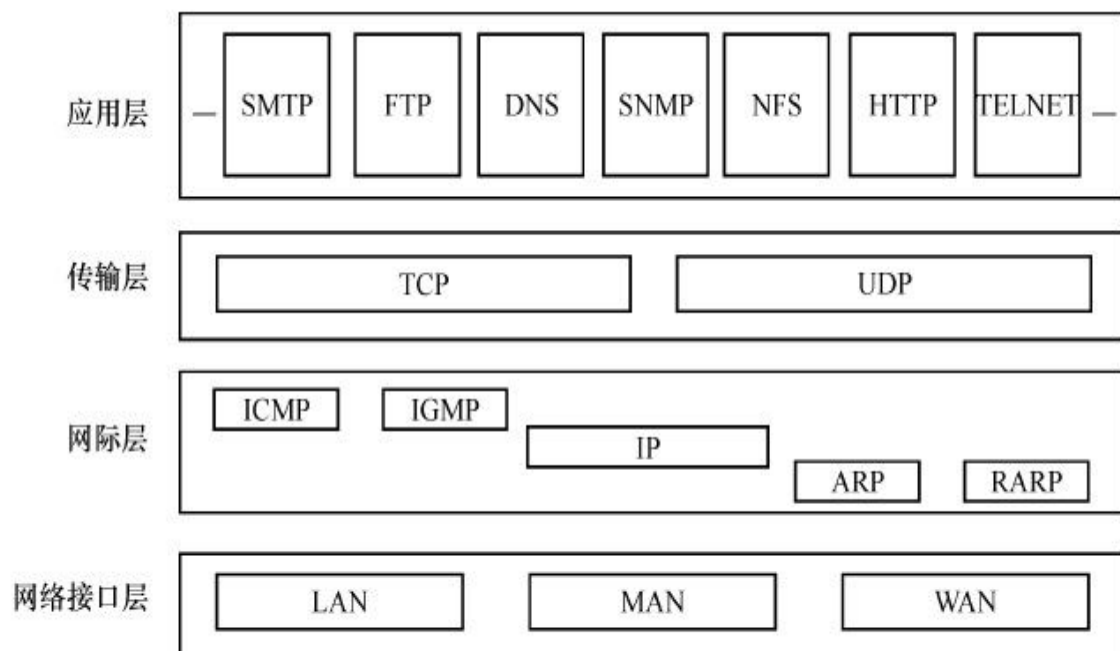


图 5-7 TCP/IP 协议族

虽然是 TCP/IP 协议，但 **TCP 协议和 IP 协议只是众多协议中的两个**，因为他们比较重要，所以就用 TCP/IP 来称呼整个协议族。除了 TCP/IP 其实还有其他协议的，比如 UDP, ICMP, IGMP, ARP。

我们网络编程主要就是使用 TCP/UDP 这两种协议。而本书主要讲解的就是 TCP。

1.2.5 总结

总结

OSI 中的层	功能	TCP/IP 协议族
应用层	文件传输，电子邮件，文件服务，虚拟终端	TFTP, HTTP, SNMP, FTP, SMTP, DNS, Telnet 等等
表示层	数据格式化，代码转换，数据加密	没有协议
会话层	解除或建立与别的接点的联系	没有协议
传输层	提供端对端的接口	TCP, UDP
网络层	为数据包选择路由	IP, ICMP, OSPF, EIGRP, IGMP
数据链路层	传输有地址的帧以及错误检测功能	SLIP, CSLIP, PPP, MTU
物理层	以二进制数据形式在物理媒体上传输数据	ISO2110, IEEE802, IEEE802.2

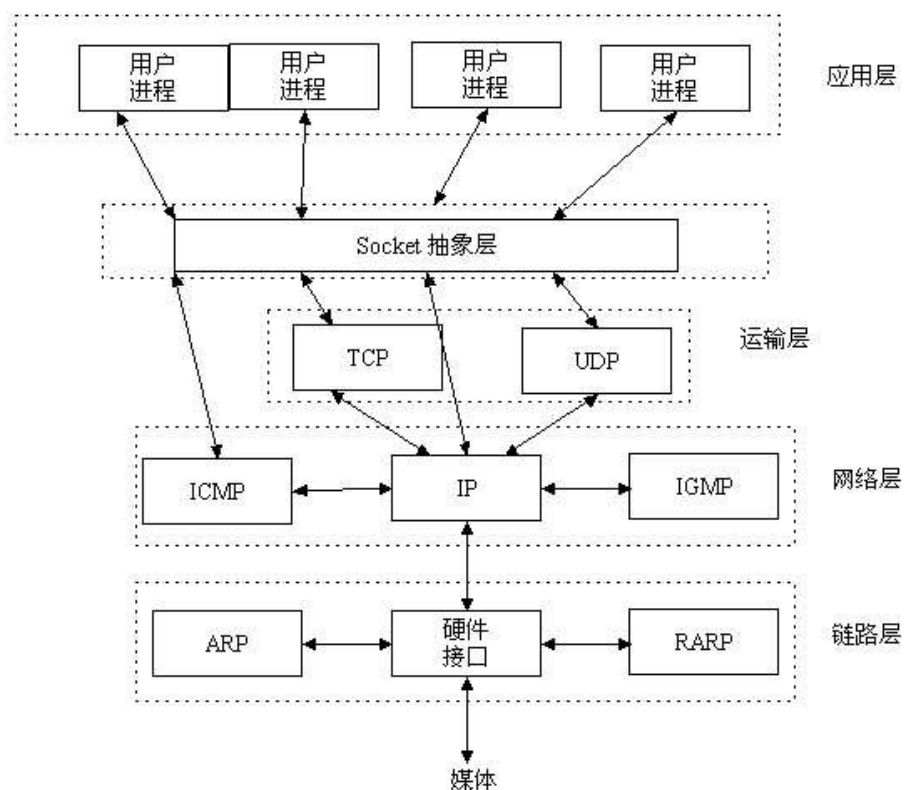
网络编程主要是在传输层，使用 TCP 或 UDP.网络层什么的了解一下就行了，除非你需要更底层的编程，比如发送自定义 IP 头数据包（无连接无端口，网络层级别）、发送 ICMP 包(ping 包)、网卡 sniffer 抓数据包(winpcap)。想要深入了解更多可以去看《Windows 网络编程技术》《TCP/IP 详解》

1.3 Socket 编程原理

TCP/IP 传输层采用的是 Socket 接口来实现，避免了开发人员直接面对复杂的多层网络协议。

1.3.1 套接字(Socket)

Socket 通常中文称作套接字。用于描述 IP 地址和端口，目的就是为了便于不同机器编程通信。 只要我们知道对方的 IP 和端口，我们就可以用 Socket 和对方相互通信。



Socket 主要分为两种，一种是流式 Socket（即是 TCP 协议用的 socket），一种是数据报式 Socket（即是 UDP 协议用的 Socket）。本书主要讲流式 Socket。

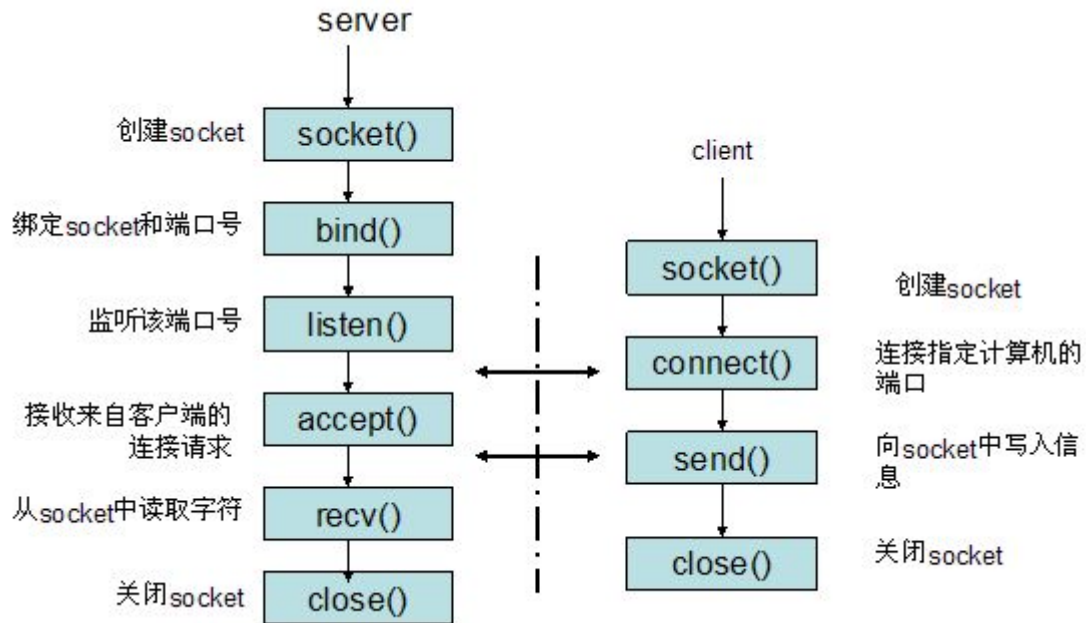
Socket 是网络 i/o 的基础，linux 和 windows 都提供了 socket 接口。

Linux 提供的是标准 Socket, 又称 Berkeley Socket。

Windows 提供了 winsock，是微软在标准 socket 基础上增加了一些 windows 的扩展，主要用于 windows 平台下开发。

1.3.2 Socket 通信流程

socket 是"打开—读/写—关闭"模式的实现，以使用 TCP 协议通讯的 socket 为例，其交互流程大概是这样子的



服务器根据地址类型（ipv4,ipv6）、socket 类型、协议创建 socket

服务器为 socket 绑定 ip 地址和端口号

服务器 socket 监听端口号请求，随时准备接收客户端发来的连接，这时候服务器的 socket 并没有被打开

客户端创建 socket

客户端打开 socket，根据服务器 ip 地址和端口号试图连接服务器 socket

服务器 socket 接收到客户端 socket 请求，被动打开，开始接收客户端请求，直到客户端返回连接信息。这时候 socket 进入阻塞状态，所谓阻塞即 `accept()` 方法一直到客户端返回连接信息后才返回，开始接收下一个客户端连接请求

客户端连接成功，向服务器发送连接状态信息

服务器 `accept` 方法返回，连接成功

客户端向 socket 写入信息

服务器读取信息

客户端关闭

服务器端关闭

先来看一个最简单 winsock c/s 例子。下面是 Server

Server.cpp

```
#include <stdio.h>
#include <winsock.h>
#pragma comment(lib, "Ws2_32")
#define MAXDATASIZE 100          /* 每次可以接收的最大字节 */

int _tmain(int argc, _TCHAR* argv[])
```

```

{
    int sockfd, new_fd;          /* 定义套接字 */
    struct sockaddr_in my_addr;  /* 本地地址信息 */
    struct sockaddr_in their_addr; /* 连接者地址信息 */
    int sin_size;

    WSADATA ws;
    WSAStartup(MAKEWORD(2,2), &ws); //初始化Windows Socket Dll

    //建立socket
    sockfd = socket(AF_INET, SOCK_STREAM, 0);

    //bind本机的端口
    my_addr.sin_family = AF_INET;      /* 协议类型是INET */
    my_addr.sin_port = htons(800);     /* 绑定 端口 */
    my_addr.sin_addr.s_addr = INADDR_ANY; /* 本机IP */

    bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr));

    //listen, 监听端口
    listen(sockfd, 5);
    printf("listen.....\n");
    //等待客户端连接
    sin_size = sizeof(struct sockaddr_in);
    new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &sin_size);

    //接收数据 并打印出来
    char buf[MAXDATASIZE];
    int numbytes = recv(new_fd, buf, MAXDATASIZE, 0);
    buf[numbytes] = '\0';
    printf("Received: [%s]", buf);

    //echo
    send(new_fd, buf, strlen(buf), 0);
    printf("send ok!\n");

    // 关闭套接字
    closesocket(sockfd);
    closesocket(new_fd);

    return 0;
}

```


下面是 Client.

Client.cpp

```
#include <stdio.h>
#include <stdio.h>
#include <winsock.h>
#pragma comment(lib, "Ws2_32")
#define MAXDATASIZE 100          /* 每次可以接收的最大字节 */

#define SERVER_IP "127.0.0.1"
#define SERVER_PORT 800

int _tmain(int argc, _TCHAR* argv[])
{

    WSADATA ws;
    WSAStartup(MAKEWORD(2,2), &ws);    //初始化Windows Socket Dll

    int sockfd = socket(AF_INET, SOCK_STREAM, 0);

    sockaddr_in their_addr; /* 对方的地址端口信息 */
    //连接服务器
    their_addr.sin_family = AF_INET; /* 协议类型是INET */
    their_addr.sin_port = htons(SERVER_PORT); /* 连接对方800端口 */
    their_addr.sin_addr.s_addr = inet_addr(SERVER_IP); /* 连接对方的IP */
    connect(sockfd, (struct sockaddr *)&their_addr, sizeof(struct sockaddr));

    //发送字符串给服务器
    char buf[MAXDATASIZE] = "hello socket!";
    send(sockfd, buf, strlen(buf), 0);
    printf("send ok!\n");

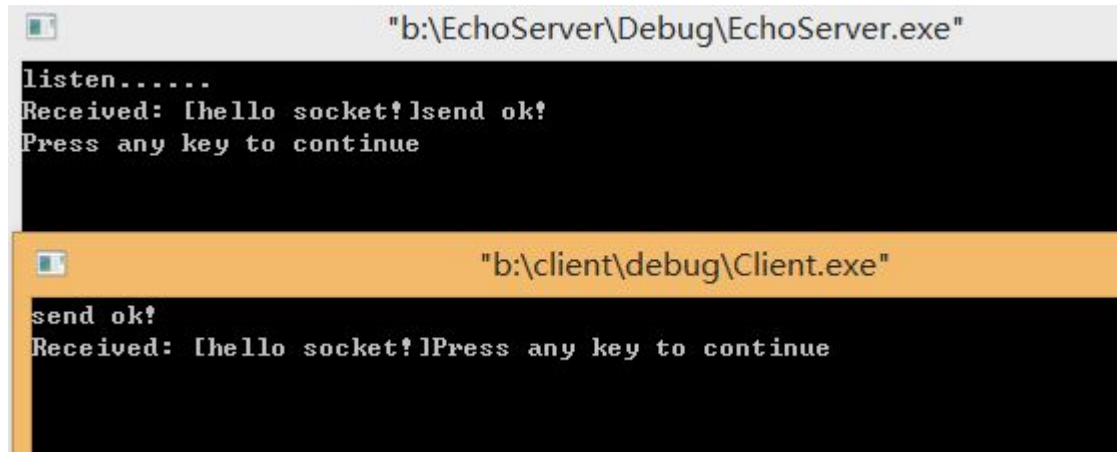
    //接收数据 并打印出来
    char buf2[MAXDATASIZE];
    int numbytes = recv(sockfd, buf2, MAXDATASIZE, 0);
    buf2[numbytes] = '\0';
    printf("Received: [%s]", buf2);

    //关闭套接字
```

```

    closesocket(sockfd);
    return 0;
}

```



先运行 Server，若运行成功，则会监听 800 端口，并等待客户连接。然后运行控制台，运行 Client 127.0.0.1，若成功，则会打印出 hello 的字符串，然后 server/client 都网络断开。

1.3.3 Socket 基本 api 介绍

WSAStartup 初始化 Windows Socket Dll

例：

```

WSADATA ws;
WSAStartup(MAKEWORD(2,2), &ws);

```

socket 创建 socket 句柄

SOCK_STREAM 参数表示创建流式 socket 即 tcp socket，SOCK_DGRAM 参数表示创建数据报 socket 即 udp socket。

例：

```

sockfd = socket(AF_INET, SOCK_STREAM, 0);

```

connect 连接指定的 ip 和端口

例：

```
their_addr.sin_family = AF_INET;           /* 协议类型是 INET    */
their_addr.sin_port = htons(800);          /* 连接对方 800 端口  */
their_addr.sin_addr.s_addr = inet_addr(argv[1]); /* 连接对方的 IP    */
connect(sockfd, (struct sockaddr *)&their_addr, sizeof(struct sockaddr));
```

bind 将一本地地址与一套接口捆绑。本函数适用于未连接的数据报或流类套接口，在 `connect()` 或 `listen()` 调用前使用。

简而言之：绑定端口，即绑定指定 ip 端口在的网卡上。如果 `bind` 失败可能是端口已经被占用了。

例：

```
my_addr.sin_family = AF_INET;           /* 协议类型是 INET    */
my_addr.sin_port = htons(800);          /* 绑定 端口    */
my_addr.sin_addr.s_addr = INADDR_ANY;   /* 本机 IP        */

bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr));
```

listen 创建一个套接口并监听申请的连接。简而言之：就是监听端口。`Listen` 调用成功后，端口就可以被其他 `socket connect` 了。如果 `listen` 失败可能是端口已经被占用了。

例：

```
listen(sockfd, 5);
```

accept 在一个套接口接受一个连接，后 2 个参数用来接收为通讯层所知的连接实体的地址（即对方的地址），参数可以为空，表示不关心（需要）对方 ip 端口相关信息。

例：

```
new_fd = accept(sockfd, NULL, NULL);
```

send 用来将数据由指定的 `socket` 传给对方主机。使用 `send` 时套接字必须已经连接

参数说明

第一个参数指定发送端套接字描述符；

第二个参数指明一个存放应用程式要发送数据的缓冲区；

第三个参数指明实际要发送的数据的字节数；

第四个参数一般置 0。

例：

```
send(new_fd, "hello\n", 6, 0);
```

recv 用于已连接的流式套接口进行数据的接收。 **参数说明** sockfd 参数是已建立连接的套接字，将在这个套接字上发送数据。第二个参数 buf，则是字符缓冲区，区内包含即将发送的数据。第三个参数 len，指定即将发送的缓冲区内的字符数。最后， flags 可为 0

例：

```
numbytes=recv(sockfd, buf, len, 0);
```

closesocket 关闭指定 socket 网络连接。

例：

```
closesocket(sockfd);
```

1.3.4 Socket api 总结

那么作为 Server 的 api 调用过程类似如下

```
WSAStartup(初始化环境);
```

```
Socket(申请 socket 句柄);
```

```
Bind(ip 端口绑定);
```

```
Listen(开始监听);
```

```
while(true)
```

```
{
```

```
    Accept(接受连接)
```

```
    Recv(接收) Send(发送)
```

```
    Closesocket(关闭对方连接)
```

```
}
```

Client api 调用过程更简单，就不多解释了。

那么有的同学会问了，Server 使用循环，在 while 里 accept 一个连接，一次只能给一个连接通讯，直到对方断开，才能接受新连接。这种代码模型称之为**串行**。当网络需求量不大的时候，这种模型其实够用了，但当大量连接同时通讯，它的弊端就出现了，无法及时响应了，有没有更好的办法来和大量的客户 socket 连接同时通讯呢？答案是有很多种！我们将在下一节里讲述。

1.4 并发 一线程一客户模型

接着上一节，我们讲到了 socket 在 while 循环里一次只能和一个客户端通讯，那么如何在一个线程里和大量的 socket 客户端通讯？答案就是并发！

Socket 并发，最简单的方法就是开线程。下面是一个代码例子

one thread one client io 模型

```
while(1)
{
    sockaddr_in adr;
    int nAddrLen = sizeof(adr);
    SOCKET c= accept(s, (struct sockaddr *)& adr, & nAddrLen);
    if(c!=INVALID_SOCKET )
    {
```

```

        //为一个socket client 开启单独的接收线程
        HANDLE h=::CreateThread(0,0, client_thread ,(LPVOID)c,0,NULL);
        ::CloseHandle(h);
    }
    .....
}

//客户线程
DWORD WINAPI client_thread(LPVOID lp)
{
    SOCKET c=(SOCKET)lp;

    可以在这里和 client 收发通讯。
}

```

下面贴出完整代码，ThreadEchoServer

ThreadEchoServer.cpp

```

#include <stdio.h>
#include <winsock.h>
#pragma comment(lib, "Ws2_32")
#define MAXDATASIZE 100          /* 每次可以接收的最大字节 */

//客户线程
DWORD WINAPI client_thread(LPVOID lp)
{
    SOCKET c=(SOCKET)lp;

    //接收数据 并打印出来
    char buf[MAXDATASIZE];
    int numbytes=recv(c, buf, MAXDATASIZE, 0);
    buf[numbytes] = '\0';
    printf("Received: [%s] \n",buf);

    //echo
    send(c,  buf,  strlen(buf) , 0);
    printf("send ok!\n");
}

```

```

// 关闭套接字
closesocket(c);

return 0;
}

int _tmain(int argc, _TCHAR* argv[])
{
    WSADATA ws;
    WSAStartup(MAKEWORD(2,2), &ws);    //初始化Windows Socket Dll

    //建立socket
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);

    sockaddr_in my_addr;    /* 本地地址信息 */
    //bind本机的端口
    my_addr.sin_family = AF_INET;    /* 协议类型是INET */
    my_addr.sin_port = htons(800);    /* 绑定 端口 */
    my_addr.sin_addr.s_addr = INADDR_ANY;    /* 本机IP */

    bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr));

    //listen, 监听端口
    listen(sockfd, 5);
    printf("listen.....\n");
    //等待客户端连接

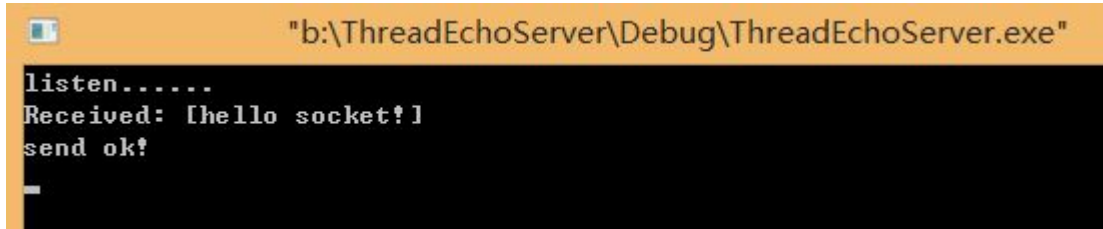
    while(1)
    {
        int sin_size;
        sockaddr_in their_addr;    /* 连接者地址信息 */
        sin_size = sizeof(struct sockaddr_in);
        SOCKET c = accept(sockfd, (struct sockaddr *)&their_addr, &sin_size);
        if(c != INVALID_SOCKET)
        {
            //为一个socket client 开启单独的接收线程
            HANDLE h = ::CreateThread(0, 0, client_thread, (LPVOID)c, 0, NULL);
            ::CloseHandle(h);
        }
    }

    // 关闭套接字

```

```
        closesocket(sockfd);  
  
        return 0;  
    }  
}
```

先运行ThreadEchoServer.exe，再运行Client.exe 运行结果如下



```
"b:\ThreadEchoServer\Debug\ThreadEchoServer.exe"  
listen.....  
Received: [hello socket!]  
send ok!  
_
```

现在，我们使用多线程实现了并发，有了可以同时和大量 socket 通讯的能力。

以上就是最简单的 one thread one client。优点简单高效适合中小型网络通讯情景，很多程序都是这种并发模型，比如 web 服务器。缺点也很明显，一个客户就要消耗一个线程资源，当大量客户成百上千连接，那么服务器明显会性能下降，创建线程变慢，甚至线程无法创建。总之一句话，太浪费资源！还有另一个更重要的缺点，那就是线程并发模型，如果客户端之间相互都没有交集，不和其他客户通讯，只和服务端交流，那么线程并发模型会工作的性能良好，但如果客户之间有数据交换，麻烦就来了，线程间通讯，是一个难题，线程通信就要同步，那么线程同步，会导致更多麻烦，效率下降，代码易出错造成死锁。 那么我们就要考虑其他更加高效的解决办法了。那就是 select io 模型！

但是在开始学习之前，还要补一点概念知识。

1.5 阻塞，非阻塞，同步，异步

同步和异步 关注的是通信机制。

所谓同步，就是发出一个‘调用’时，在还没有获得结果前，此‘调用’就不返回。 同步听的比较多的应该是线程同步，也就是多线程操作一个公共数据，但是问题来了，多线程同时写一个区域，那么会导致混乱，所以引用了线程同步这个概念，以及同步相关技术，比如线程锁，信号量，互斥，事件。

所谓异步，就是发出一个‘调用’时，提供一个接收结果回调接口(callback)或其他通知结果到达的方法，调用就返回了。

经常见到一些程序说明，同步调用，异步调用，都是说他的程序通信机制。比如 windows 提供文件读写操作，就有同步函数和异步函数。同步函数调用简单，运行流程一直往下走就行。但异步调用就比较繁琐，运行流程比较复杂，非直线型。但异步往往效率会更高，因为不会有同步等待结果造成时间消耗。

阻塞和非阻塞 关注的是 **调用中等待结果（消息，返回值）时的状态。**

阻塞调用是指在阻塞模式下，在 I/O 操作完成前，执行操作的函数（比如 `send` 和 `recv`）会一直等候下去，不会立即返回程序，当前线程会被挂起。调用线程只有在得到结果之后才会返回。

非阻塞调用指在非阻塞模式下，在不能立刻得到结果之前，调用不会阻塞当前线程，会立即返回。

看完了概念，我就来举一些例子说明。

`accept` 函数会阻塞线程，因为如果没有客户端连接监听的端口。那么 `accept` 就不会获得接受连接，会始终等待。

套接字有 2 种模式：阻塞非阻塞（有的书翻译成 锁定和非锁定）
`accept/recv/send` 等函数默认模式下是阻塞的，会阻塞线程。

对于处在锁定（阻塞）模式的套接字，我们必须多加留意，因为在一个锁定套接字上调用任何一个 `socket` 函数，都会产生相同的后果—耗费或长或短的时间“等待”。

相对的，非阻塞模式的套接字，调用会立即返回。大多数情况下，这些调用都会“失败”，并返回一个错误。

1.6 select io 模型

Winsock 提供的 I/O 模型一共有五种

`select`, `WSAAsyncSelect`, `WSAEventSelect`, `Overlapped`, `CompletionPort`。

Linux 下提供的 io 模型也有几种，其中也有 `select`。

这里先讲 `select` io 模型。因为它比较高效节能，至少比一客户一线程好多了，同时多平台都提供了 `select`，学会了它就可以编写出跨平台的网络程序（当然 `socket` 本身就可以跨平台了）

之所以称其为“`select` 模型”，是由于 它的“精髓”便是利用 `select` 函数，实现对 I/O 的管理！

```

int select(
int nfd,
fd_set FAR *readfds,
fd_set FAR *writefds,
fd_set FAR *exceptfds,
const struct timeval FAR *timeout
);

```

利用 select 函数，我们可以判断套接字上是否存在数据，或者能否向一个套接字写入数据。之所以要设计这个函数，唯一的目的是防止应用程序在套接字处于锁定模式中时，在一次 I/O 调用（如 send 或 recv）过程中，被迫进入“锁定”状态。

介绍下 windows 下的 select 参数含义。第一个 nfd 在 windows 里无用，默认 0 即可。

第二个参数表示 fd_set 读集,用于检查可读性（readfds），第三个参数用于检查可写性（writefds），第四个参数用于例外数据（exceptfds）。从根本上说，fd_set 数据类型代表着一系列特定套接字的集合。可以跟踪进去查看

```

#ifdef FD_SETSIZE
#define FD_SETSIZE      64
#endif /* FD_SETSIZE */

typedef struct fd_set {
    u_int    fd_count;           /* how many are SET? */
    SOCKET   fd_array[FD_SETSIZE]; /* an array of SOCKETs */
} fd_set;

```

看见 FD_SETSIZE 默认最大 64。即一次最大 select 调用检查 fd_set 集合，可以传递 64 个 socket 句柄。当然我们可以在包含头文件之前先定义，比如 `#define FD_SETSIZE 1024` 这样就可以一次检查 1024 个 socket 读或写有效性。

最后一个参数 timeout 对应的是一个指针，它指向一个 timeval 结构，用于决定 select 最多等待 I/O 操作完成多久的时间。如 timeout 是一个空指针，那么 select 调用会无限地“锁定”或停顿下去，直到至少有一个描述符符合指定的条件后结束。

Windows 提供了下列宏操作，可用来针对 I/O 活动，对 fd_set 进行处理与检查：

■ FD_CLR(s, *set): 从 set 中删除套接字 s。

- FD_ISSET(s, *set): 检查 s 是否 set 集合的一名成员；是，则返回 TRUE。
- FD_SET(s, *set): 将套接字 s 加入集合 set。
- FD_ZERO (*set): 将 set 初始化成空集合。

用下述步骤，便可完成用 select 操作一个或多个套接字句柄的全过程：

1. 使用 FD_ZERO，初始化一个 fd_set。
2. 使用 FD_SET，将套接字句柄分配给一个 fd_set。
3. 调用 select 函数，然后等待在指定的 fd_set 集合中，I/O 活动设置好一个或多个套接字句柄。select 完成后，会返回在所有 fd_set 集合中设置的套接字句柄总数，并对每个集合进行相应的更新。
4. 根据 select 的返回值，我们的应用程序便可判断出哪些套接字存在着尚未完成（待决）的 I/O 操作—具体的方法是使用 FD_ISSET 宏，对每个 fd_set 集合进行检查。
5. 知道了每个集合中“待决”的 I/O 操作之后，对 I/O 进行处理，然后返回步骤 1，继续进行 select 处理。

// select 模型处理过程 代码片段

```

SOCKET sock;
fd_set fdRead;
FD_ZERO(&fdRead); // FD_ZERO 初始化套接字集合
FD_SET(sock, &fdRead); // 添加要监听的套接字
// select 检查集合
int nRet = ::select(0, &fdRead, NULL, NULL, NULL);
if(nRet > 0)
{
    //测试 sock 是否有待决的 I/O
    if(FD_ISSET(sock, &fdRead))
    {
        //这里处理 I/O
        char szText[256];
        int nRecv = ::recv(sock, szText, strlen(szText), 0);
        if(nRecv > 0) // (2) 可读
        {
            szText[nRecv] = '\0';
            printf("接收到数据: %s\n", szText);
        }
        else // (3) 连接关闭
        {
            printf("%d 断开\n", sock);
            ::closesocket(sock);
        }
    }
}

```

```

        FD_CLR(sock , & fdRead);
    }
}
}else
{
    printf(" Failed select() \n");
}

```

下面贴出使用 select io 模型实现的 SelectEchoServer

SelectEchoServer.cpp

```

#include <stdio.h>
#include <winsock.h>
#pragma comment(lib,"Ws2_32")
#define MAXDATASIZE 100          /* 每次可以接收的最大字节 */

int nPort = 800; // 此服务器监听的端口号

int _tmain(int argc, _TCHAR* argv[])
{
    WSADATA ws;
    WSAStartup(MAKEWORD(2,2),&ws); //初始化Windows Socket Dll

    // 创建监听套节字
    SOCKET sListen = ::socket(AF_INET, SOCK_STREAM, 0);
    sockaddr_in sin;
    sin.sin_family = AF_INET;
    sin.sin_port = htons(nPort);
    sin.sin_addr.S_un.S_addr = INADDR_ANY;
    // 绑定套节字到本地机器
    if(::bind(sListen, (sockaddr*)&sin, sizeof(sin)) == SOCKET_ERROR)
    {
        printf(" Failed bind() \n");
        return -1;
    }
    // 进入监听模式
    ::listen(sListen, 5);
}

```

```

// select模型处理过程
// 1) 初始化一个套节字集合fdSocket, 添加监听套节字句柄到这个集合
fd_set fdSocket;    // 所有可用套节字集合
FD_ZERO(&fdSocket);
FD_SET(sListen, &fdSocket);
while(TRUE)
{
    // 2) 将fdSocket集合的一个拷贝fdRead传递给select函数,
    // 当有事件发生时, select函数移除fdRead集合中没有未决I/O操作的套节
    // 字句柄, 然后返回。
    fd_set fdRead = fdSocket;
    int nRet = ::select(0, &fdRead, NULL, NULL, NULL);
    if(nRet > 0)
    {

        // 3) 通过将原来fdSocket集合与select处理过的fdRead集合比较,
        // 确定都有哪些套节字有未决I/O, 并进一步处理这些I/O。
        for(int i=0; i<(int)fdSocket.fd_count; i++)
        {
            if(FD_ISSET(fdSocket.fd_array[i], &fdRead))
            {
                if(fdSocket.fd_array[i] == sListen)    // (1) 监听套节字接收
                到新连接
                {
                    if(fdSocket.fd_count < FD_SETSIZE)
                    {
                        sockaddr_in addrRemote;
                        int nAddrLen = sizeof(addrRemote);
                        SOCKET sNew = ::accept(sListen,
(SOCKADDR*)&addrRemote, &nAddrLen);
                        FD_SET(sNew, &fdSocket);
                        printf("接收到连接%d (%s) \n",
sNew, ::inet_ntoa(addrRemote.sin_addr));
                    }
                    else
                    {
                        printf(" Too much connections! \n");
                        continue;
                    }
                }
            }
            else
            {
                char szText[256];
                int nRecv = ::recv(fdSocket.fd_array[i], szText,

```

```

strlen(szText), 0);
        if(nRecv > 0)                                // (2) 可读
        {
            szText[nRecv] = '\0';
            printf("Received: [%s]\n",szText);
            //echo
            //echo
            send(fdSocket.fd_array[i],  szText ,  strlen(szText) ,
0);

            printf("send ok!\n");
        }
        else                                          // (3) 连接关闭
        {
            printf("%d断开 \n", fdSocket.fd_array[i]);
            ::closesocket(fdSocket.fd_array[i]);
            FD_CLR(fdSocket.fd_array[i], &fdSocket);
        }
    }
}
}
else
{
    printf(" Failed select() \n");
    break;
}
}

return 0;
}

```

先运行SelectEchoServer.exe ， 再运行Client.exe， 结果如下图

```

b:\selectechoserver\debug\SelectEchoServer.exe
接收到连接440 (127.0.0.1)
Received: [hello socket!]
send ok!
440断开
接收到连接444 (127.0.0.1)
Received: [hello socket!]
send ok!
444断开

```

现在，我们很好的只在一个线程里就实现了并发，可以同时和大量 socket 通讯的能力。

1.7 iocp/boost asio

IOCP (I/O Completion Port)，常称 I/O 完成端口。IOCP 模型属于一种通讯模型，适用于(能控制并发执行的)高负载服务器的一个技术，是 windows 上目前效率最高的 I/O 模型。使用 IOCP 可以实现成千上万的连接通讯，但是由于 IOCP 理解以及编码的复杂度较高，对使用者综合知识有一定要求，同步与异步，阻塞与非阻塞，重叠 I/O 技术，多线程，栈、队列，指针。对新手而言是个很大的障碍。于是我推荐使用 Boost Asio。

Boost 库是一个可移植、提供源代码的 C++库，作为标准库的后备，是 C++标准化进程的开发引擎之一。

Asio 则是 Boost 里的一个功能模块。提供了网络 I/O 功能。Asio 提供了同步和异步的 I/O 功能，其中异步 I/O（在 windows 平台下）的底层最终调用的就是 IOCP。

我们如果学会了 Asio，就等于学会了高效的网络编程，而且是跨平台的。你需要的只是下载，然后花一点时间学会它。

Boost 下载安装就不细说了。官网 <http://www.boost.org/> 目前最新的版本是 1.59，我开发用的版本是 1.46

还是以 echo 功能作为例子，用 asio 阻塞模式的 socket 来实现。

AsioBlockEchoServer.cpp

```
#include <cstdlib>
#include <iostream>
#include <boost/bind.hpp>
#include <boost/smart_ptr.hpp>
#include <boost/asio.hpp>
#include <boost/thread.hpp>

using boost::asio::ip::tcp;

const int max_length = 1024;

typedef boost::shared_ptr<tcp::socket> socket_ptr;

void session(socket_ptr sock)
```

```

{
    try
    {
        for (;;)
        {
            char data[max_length];

            boost::system::error_code error;
            size_t length = sock->read_some(boost::asio::buffer(data), error);
            if (error == boost::asio::error::eof)
                break; // Connection closed cleanly by peer.
            else if (error)
                throw boost::system::system_error(error); // Some other error.

            boost::asio::write(*sock, boost::asio::buffer(data, length));
        }
    }
    catch (std::exception& e)
    {
        std::cerr << "Exception in thread: " << e.what() << "\n";
    }
}

void server(boost::asio::io_service& io_service, short port)
{
    tcp::acceptor a(io_service, tcp::endpoint(tcp::v4(), port));
    for (;;)
    {
        socket_ptr sock(new tcp::socket(io_service));
        a.accept(*sock);
        boost::thread t(boost::bind(session, sock));
    }
}

int main(int argc, char* argv[])
{
    try
    {
        boost::asio::io_service io_service;
        server(io_service, 800);
    }
    catch (std::exception& e)
    {
        std::cerr << "Exception: " << e.what() << "\n";
    }
}

```



```

    }

    return 0;
}

```

可以看出 boost 的封装代码风格，纯面向对象，全部小写，用到了 bind, shared_ptr, 这些高级的 C++ 技术，对新手来说还是有些吃力。

这个例子是阻塞模式，接受一个连接就开启一个线程的。跟我们前面的一线程一客户的 io 模型是一个原理。

下面是异步 asio socket 实现的 Echo Server

AsioAsyncEchoServer.cpp

```

#include <cstdlib>
#include <iostream>
#include <boost/bind.hpp>
#include <boost/asio.hpp>

using boost::asio::ip::tcp;

class session
{
public:
    session(boost::asio::io_service& io_service)
        : socket_(io_service)
    {
    }

    tcp::socket& socket()
    {
        return socket_;
    }

    void start()
    {
        socket_.async_read_some(boost::asio::buffer(data_, max_length),
            boost::bind(&session::handle_read, this,
                boost::asio::placeholders::error,
                boost::asio::placeholders::bytes_transferred));
    }
}

```

```

void handle_read(const boost::system::error_code& error,
                 size_t bytes_transferred)
{
    if (!error)
    {
        boost::asio::async_write(socket_,
                                  boost::asio::buffer(data_, bytes_transferred),
                                  boost::bind(&session::handle_write, this,
                                                boost::asio::placeholders::error));
    }
    else
    {
        delete this;
    }
}

void handle_write(const boost::system::error_code& error)
{
    if (!error)
    {
        socket_.async_read_some(boost::asio::buffer(data_, max_length),
                                 boost::bind(&session::handle_read, this,
                                               boost::asio::placeholders::error,
                                               boost::asio::placeholders::bytes_transferred));
    }
    else
    {
        delete this;
    }
}

private:
    tcp::socket socket_;
    enum { max_length = 1024 };
    char data_[max_length];
};

class server
{
public:
    server(boost::asio::io_service& io_service, short port)
        : io_service_(io_service),
          acceptor_(io_service, tcp::endpoint(tcp::v4(), port))
    {

```

```

        session* new_session = new session(io_service_);
        acceptor_.async_accept(new_session->socket(),
                                boost::bind(&server::handle_accept, this, new_session,
                                              boost::asio::placeholders::error));
    }

    void handle_accept(session* new_session,
                       const boost::system::error_code& error)
    {
        if (!error)
        {
            new_session->start();
            new_session = new session(io_service_);
            acceptor_.async_accept(new_session->socket(),
                                    boost::bind(&server::handle_accept, this, new_session,
                                                  boost::asio::placeholders::error));
        }
        else
        {
            delete new_session;
        }
    }
}

private:
    boost::asio::io_service& io_service_;
    tcp::acceptor acceptor_;
};

int main(int argc, char* argv[])
{
    try
    {
        boost::asio::io_service io_service;
        server s(io_service, 800);
        io_service.run();
    }
    catch (std::exception& e)
    {
        std::cerr << "Exception: " << e.what() << "\n";
    }

    return 0;
}

```

1.8 asio 基本概念

asio 基于两个概念：I/O 服务和 I/O 对象

I/O 服务，抽象了操作系统的异步接口 这个对象是核心

`boost::asio::io_service`

I/O 对象，有多种对象

`boost::asio::ip::tcp::socket` socket 的 oop 封装

`boost::asio::ip::tcp::resolver`

`boost::asio::ip::tcp::acceptor` 接受器。功能就是 accept 客户的 connect 请求

`boost::asio::local::stream_protocol::socket` 本地连接

`boost::asio::posix::stream_descriptor` 面向流的文件描述符，比如 stdout, stdin

`boost::asio::deadline_timer` 定时器

`boost::asio::signal_set` 信号处理

所有 I/O 对象通常都需要一个 I/O 服务作为它们的构造函数的第一个参数，比如：

```
boost::asio::io_service io_service;
```

```
boost::asio::deadline_timer timer(io_service, boost::posix_time::seconds(5));
```

开始分析 AsioAsyncEchoServer 代码，首先定义了一个 `io_service`，然后创建了 `server` 对象的实例，`server` 构造传递了 `io_service` 和一个端口号，构造初始化列表又初始化了 `acceptor` 对象，用于接受连接，

`acceptor` 的构造传递了 `io_service` 和 `tcp::endpoint(tcp::v4(), port)` 用于创建一个 `ipv4` 上的监听端口。接着 `new` 了一个 `session`（表示一个连接/会话），然后 `acceptor.async_accept` 把这个 `session` 投递，并传递 `handle_accept` 用于实现 `async_accept` 事件完成后的回调。这样后，当客户连接了 `Server` 的 800 端口，`Server` 即会调用 `handle_accept` 来处理连接请求。`handle_accept` 里的处理也很简单，如果没出错，参数 `new_session` 调用 `start` 开始预读数据，接着创建一个新的 `session` 用 `acceptor.async_accept` 投递，从而达到不断接受新连接，如果 `handle_accept` 出错了，则 `delete` 这个会话对象。

`async_read_some` `async_write` 这两个异步读写方法，分别使用 `handle_read` `handle_write` 来处理 I/O 完成的结果。`handle_read` 代码是
如果没出错，则说明上次投递的读操作成功了，然后使用 `async_write` 把成功读取的数据再发送出去，出错了则 `delete` 会话自己。

1.9 数据协议 打包

看到这里，有的同学会问了，这些代码虽然都看懂了，但只是 Echo 发送接收字符串数据太简单了，想知道那些复杂的服务器发送各种消息数据是怎么实现的？别急，这一节就开始讲解数据协议与打包。

首先，网络服务端程序需要制定一个协议，规范，有了数据协议，才能双方通讯正常，比如 web server 遵守了 http 协议，浏览器才能按 http 协议正常的访问。

发送的数据并不局限于字符串，其实也可以发送任意数据类型，int float 数据类型 甚至是结构体，因为它们最终会被转化为二进制流传输。

比如

```
float f;  
int ret= recv( s, (char*)&f, sizeof(float),0 );  
这样就可以接收一个 float 类型的数据
```

```
send(s, (char*)&f, sizeof(float),0 ); 这样就可以发送一个 float 类型的数据
```

然后按顺序发送按顺序接收各种类型的数据，就是所谓的数据打包解包。

下面总结一下常见的数据打包方法

1. C 结构体大法，发送接收结构体，免去了数据打包解包。
2. XML 文件法，把数据序列化成 xml 文件字符串然后发送，另一端接收然后字符串解析成 xml 格式文件，读取 xml 数据。略微繁琐，数据是可视化字符，传输二进制类型时要转码，空间会增大，但避免了结构体类型的依赖，可以更广泛的支持其他平台。
3. Json 文件法，类型 xml 文件法。只是格式换成了 json，比 xml 节约空间。
4. 顺序写顺序读（序列化）法，即每个消息都要定义它的打包解包的具体数据读写顺序。略微繁琐。

方法各有各的优点，比如结构体法，如果写的程序只考虑 c/c++，那么结构体最简便。

接下来贴出一个结构体 Server/Client 的例子，让大家来学习。

Struct_Server.cpp

```
#include <stdio.h>  
#include <winsock.h>  
#pragma comment(lib,"Ws2_32")
```

```
struct MSG1
```

```

{
    int cmd;
    float f;
    char msg[50];
};

int _tmain(int argc, _TCHAR* argv[])
{
    int sockfd, new_fd;          /* 定义套接字 */
    struct sockaddr_in my_addr;  /* 本地地址信息 */
    struct sockaddr_in their_addr; /* 连接者地址信息 */
    int sin_size;

    WSADATA ws;
    WSAStartup(MAKEWORD(2,2), &ws); //初始化Windows Socket Dll

    //建立socket
    sockfd = socket(AF_INET, SOCK_STREAM, 0);

    //bind本机的端口
    my_addr.sin_family = AF_INET;      /* 协议类型是INET */
    my_addr.sin_port = htons(800);     /* 绑定 端口 */
    my_addr.sin_addr.s_addr = INADDR_ANY; /* 本机IP */

    bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr));

    //listen, 监听端口
    listen(sockfd, 5);
    printf("listen.....\n");
    //等待客户端连接
    sin_size = sizeof(struct sockaddr_in);
    new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &sin_size);

    //接收数据 并打印出来
    MSG1 m;
    int numbytes=recv(new_fd, (char*)&m, sizeof(MSG1), 0);

    printf("MSG1.cmd= %d \n", m.cmd);
    printf("MSG1.f= %f \n", m.f);
    printf("MSG1.msg= %s \n", m.msg);

    //echo

```

```

send(new_fd, (char*)&m, sizeof(MSG1), 0);
printf("send ok!\n");

// 关闭套接字
closesocket(sockfd);
closesocket(new_fd);

return 0;
}

```

Struct_Client.cpp

```

#include <stdio.h>
#include <stdio.h>
#include <winsock.h>
#pragma comment(lib, "Ws2_32")

#define SERVER_IP "127.0.0.1"
#define SERVER_PORT 800

struct MSG1
{
    int cmd;
    float f;
    char msg[50];
};

int _tmain(int argc, _TCHAR* argv[])
{
    WSADATA ws;
    WSStartup(MAKEWORD(2,2), &ws);    //初始化Windows Socket Dll

    int sockfd = socket(AF_INET, SOCK_STREAM, 0);

    sockaddr_in their_addr;    /* 对方的地址端口信息 */
    //连接对方
    their_addr.sin_family = AF_INET;    /* 协议类型是INET */
    their_addr.sin_port = htons(SERVER_PORT);    /* 连接对方

```

```

800端口    */
    their_addr.sin_addr.s_addr = inet_addr( SERVER_IP ); /* 连接对方的IP    */
    connect(sockfd, (struct sockaddr *)&their_addr,sizeof(struct sockaddr));

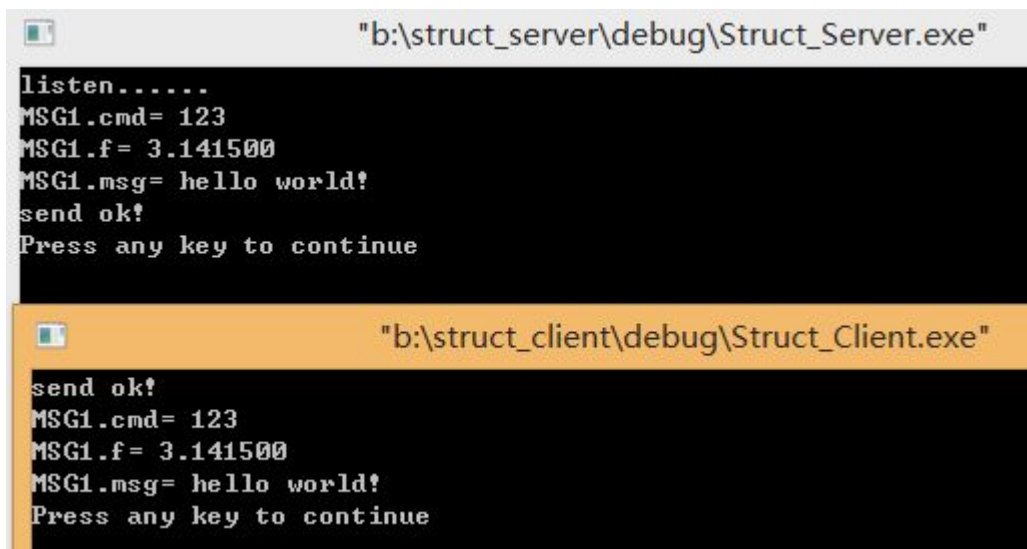
    //发送结构体给服务器
    MSG1 m;
    m.cmd=123;
    m.f=3.1415;
    strcpy(m.msg,"hello world!");
    send(sockfd, (char*)&m, sizeof(MSG1), 0);
    printf("send ok!\n");

    //接收数据 并打印出来
    MSG1 m2;
    int numbytes=recv(sockfd, (char*)&m2, sizeof(MSG1), 0);

    printf("MSG1.cmd= %d \n",m2.cmd);
    printf("MSG1.f= %f \n",m2.f);
    printf("MSG1.msg= %s \n",m2.msg);

    //关闭套接字
    closesocket(sockfd);
    return 0;
}

```



```

b:\struct_server\debug\Struct_Server.exe
listen.....
MSG1.cmd= 123
MSG1.f= 3.141500
MSG1.msg= hello world!
send ok!
Press any key to continue

b:\struct_client\debug\Struct_Client.exe
send ok!
MSG1.cmd= 123
MSG1.f= 3.141500
MSG1.msg= hello world!
Press any key to continue

```


如图所示，Server 和 Client 完美的使用了结构体通讯成功，Server 接收并打印出结构体的成员数据，然后 echo 结构体给 Client。

以此类推，设计好各种收发协议，定义好各种命令 id，结构体成员数据，通常首个成员定义成 int cmd;来表示命令 id。

1.10 网络引擎 模块化

结合前面所有知识的学习，我们这一节开始把网络模块封装化，成为一个独立的网络引擎，封装成引擎的好处有很多，把网络 io 代码和实际业务代码分离，不再强耦合，把代码模块化，定义好接口，即插即用。

我采用的是函数指针回调方式。

```
/*  
/*  
    服务端网络引擎接口  
  
*/  
/*  
//回调处理数据函数原型  
typedef VOID WINAPI ServerProcessRecvData( DWORD dwNetworkIndex ,  
BYTE *pMsg ,  WORD wSize  );  
  
class INetServer  
{  
public:  
    //是否已初始化监听  
    virtual BOOL IsListening()=0;  
  
    //网络初始化  
    virtual BOOL Init(  char* IP,  WORD Port ,  ServerProcessRecvData*  
pProcessRecvData ,  DWORD MaxConnectNum )=0;  
  
    //停止网络服务  
    virtual VOID Shutdown()=0;  
  
    //更新  
    virtual VOID Update()=0;  
  
    //单个断开
```

```

virtual BOOL Disconnect( DWORD dwNetworkIndex )=0;

//单个发送
virtual BOOL Send( DWORD dwNetworkIndex , BYTE *pMsg , WORD
wSize )=0;

//得到当前总连接数
virtual DWORD GetNumberOfConnections()=0;

//得到 ip
virtual char* GetIP(    DWORD dwNetworkIndex )=0;

};

```

INetServer 就是我们服务端网络引擎的接口定义。

ServerProcessRecvData 函数指针的参数的分别意义是 Client 会话索引，接收到的数据，接受到的数据长度。如果接收到的数据指针为空，则表示连接断开了。

下面是客户端网络引擎的接口定义

```

/*****
/*
    客户端网络引擎接口

*/
/*****
//回调处理数据函数原型
typedef VOID WINAPI ClientProcessRecvData(INetClient* p, BYTE *pMsg ,
WORD wSize  );

class INetClient
{

public:
    //连接服务器
    virtual BOOL Connect(  char* ServerIP,  WORD Port ,
ClientProcessRecvData* pProcessRecvData )=0;

    //断开连接
    virtual BOOL Disconnect( )=0;

```

```

//指定发送
virtual BOOL Send( BYTE *pMsg , WORD wSize )=0;

//更新
virtual VOID Update()=0;

};

```

1.11 网络引擎内部数据协议

为了网络引擎运行良好，我们还需要设置一个协议规范。比如有时接收数据，我们并不知道对方发送什么类型的数据，也不知道后续数据还有多长？怎么解决这个问题？答案就是要制定一个数据协议，我们规定网络引擎接口 **Send** 函数不仅要发送用户指定长度的数据，还要保证数据完整发送，另一端接收数据保证完整接收，要保证回调处理函数获得的数据无误，且知道这一次回调的数据长度大小。

我们需要在 **Send** 函数内部把用户数据封包，好比发快递包裹，需要给包贴上标签标明包的一些数据信息，**Send** 把用户数据封包，这个用户数据通常称之为‘包体’，而把用户数据额外信息的这一部分称之为‘包头’，**Send** 的数据被打包成 ‘包头’ + ‘包体’，然后发送出去。包头里最重要的信息当然是标明包体长度了，不然接收端不知道后续数据到底有多大。

包头通常用结构体表示。如下就是一个包头的定义

```

struct PACKET_HEADER
{
    WORD size; //待接收的数据包总大小
};

```

这样有了包头，我们网络引擎就能完美的解决数据包长度的相关问题。

ps:你也可以扩展包头的定义，比如加入校验位，**crc32** 数据包校验，加密 **key**，版本号等等

Send 的函数实现如下

```

BOOL Send (DWORD dw, BYTE *pMsg , WORD wSize )
{
    //包头
    PACKET_HEADER pack_hdr;
    pack_hdr.size=wSize;
}

```

```

//发送包头
int rt=send(dw, (char*)&pack_hdr, sizeof(PACKET_HEADER), 0);
if(rt == SOCKET_ERROR || rt == 0) return FALSE;

//发送包体
rt=send(dw, (const char*) pMsg, wSize, 0);
if(rt == SOCKET_ERROR || rt == 0) return FALSE;

return TRUE;
}

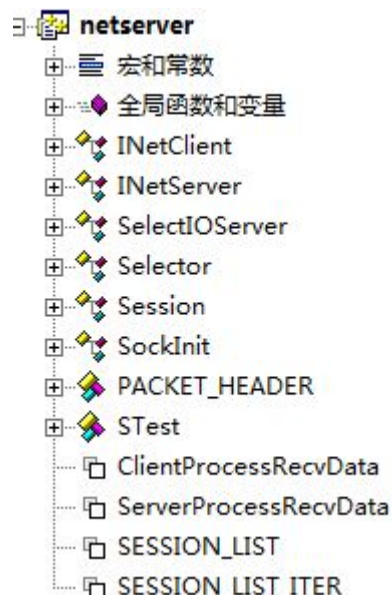
```

接收数据包时 思路就清晰了，先接收包头，得到包体长度，再接收包体，这样一包数据就接受成功了。

1.12 网络引擎

接下来我们开始网络引擎的封装代码编写工作。首先是基于 select io 模型的网络引擎编写。

NetServer 项目的代码结构如下



SelectIOServer 就是 INetServer 网络接口的实现类。实现了诸如 Init, Update, Send

这些重要的网络方法。

Selector 是 select io 模型的封装。简化操作。

Session 是客户端对象的封装类，每一个 Session 就表示一个 socket 客户端对象。

SocketInit 是一个初始化 winsock 库的类，很简单。

`typedef std::list<Session*> SESSION_LIST;` 是一个容器，链表，用于存放我们 Session 对象。

PACKET_HEADER 是我们包头的定义。

下面贴出 main.cpp 使用网络引擎 NetServer 的例子

```
SelectIOServer s;
```

```
struct STest
```

```
{
    int cmd;
    byte cc;
    int a;
    DWORD time;
    char name[100];
    char name2[10];
};
```

```
int sendcount=0;
```

```
int lastsendcount=0;
```

```
int recvcount=0;
```

```
int lastrecvcount=0;
```

```
//网络消息处理
```

```
void WINAPI RecvDataCall(DWORD dwNetworkIndex , BYTE *pMsg, WORD wSize)
```

```
{
    if( wSize == 0 || pMsg == NULL )//客户退出了
    {
        printf("[%d]退出\n",dwNetworkIndex );
        return;
    }else{

        ++recvcount;
        s.Send(dwNetworkIndex,pMsg,wSize);//简单echo
        ++sendcount;
    }
}
```

```
//-----
// Main
```

//-----

```
int _tmain(int argc, _TCHAR* argv[])
{
    int port=123;

    if(s.Init("",port,RecvDataCall,999))
    {
        printf("CreateNetServer    ok! listen [%d] ok\n",port);
    }else{
        printf("CreateNetServer    err! listen err\n");
        return 0;
    }

    printf("按任意键退出\n");

    DWORD last=::GetTickCount();
    while(s.IsListening() )
    {
        if(kbhit())
        {
            getch();
            // bd();
            break;
        }

        s.Update();

        if(GetTickCount()-last>1000)
        {
            last=GetTickCount();

            printf("有[%d]个客户连接   发包[%d/s] 收包
[%d/s]\n",s.GetNumberOfConnections()

                ,sendcount,recvcount );

            lastrecvcount=recvcount;
            lastsendcount=sendcount;
            sendcount=0;
            recvcount=0;
        }

        Sleep(1);
    }
}
```

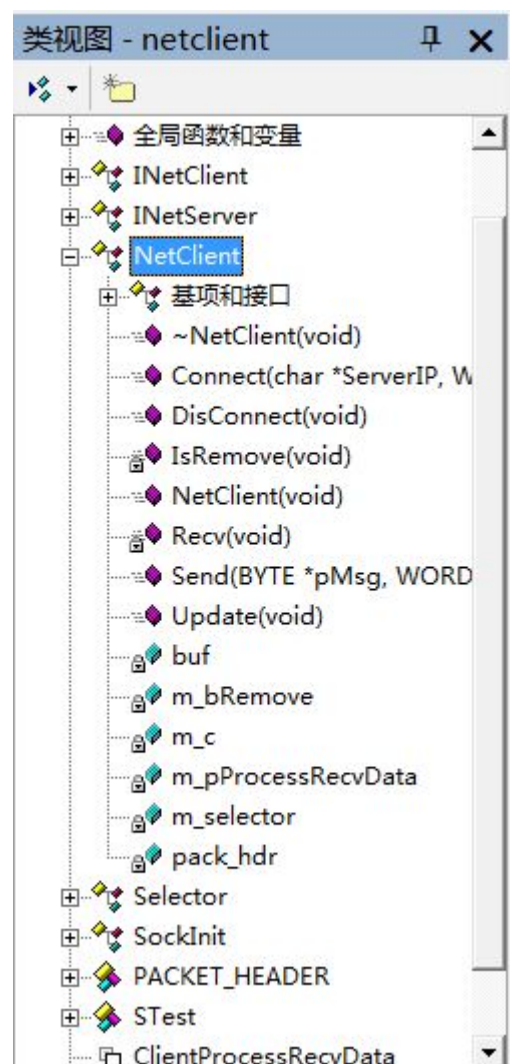
```

    }

    return 0;
}

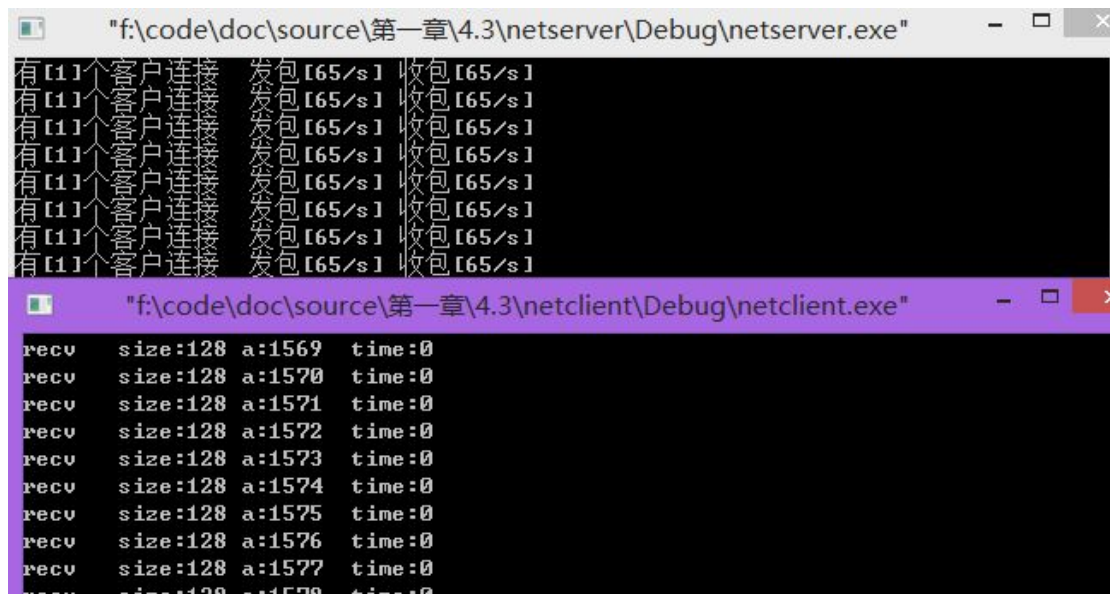
```

NetClient 更加简单就不多做介绍了。



NetClient 和 NetServer 主要的网络更新工作都在 Update 函数里进行。Update 里就是使用 select 函数测试当前 sock 是否可读，如果可读，就对应接收数据，然后回调给接口 ClientProcessRecvData 或 ServerProcessRecvData。

NetClient 和 NetServer 两个工程代码编写完毕，先运行 netserver.exe，再运行 netclient.exe 会看见双方开始收发数据了。如图所示



The image shows two overlapping Windows command prompt windows. The top window, titled "f:\code\doc\source\第一章\4.3\netserver\Debug\netserver.exe", displays a series of log messages indicating successful connections and data transmission. The bottom window, titled "f:\code\doc\source\第一章\4.3\netclient\Debug\netclient.exe", shows the corresponding receipt of data packets with their sizes and addresses.

```
"f:\code\doc\source\第一章\4.3\netserver\Debug\netserver.exe"
有[1]个客户连接 发包 [65/s] 收包 [65/s]
有[1]个客户连接 发包 [65/s] 收包 [65/s]
有[1]个客户连接 发包 [65/s] 收包 [65/s]
有[1]个客户连接 发包 [65/s] 收包 [65/s]
有[1]个客户连接 发包 [65/s] 收包 [65/s]
有[1]个客户连接 发包 [65/s] 收包 [65/s]
有[1]个客户连接 发包 [65/s] 收包 [65/s]
有[1]个客户连接 发包 [65/s] 收包 [65/s]

"f:\code\doc\source\第一章\4.3\netclient\Debug\netclient.exe"
recv size:128 a:1569 time:0
recv size:128 a:1570 time:0
recv size:128 a:1571 time:0
recv size:128 a:1572 time:0
recv size:128 a:1573 time:0
recv size:128 a:1574 time:0
recv size:128 a:1575 time:0
recv size:128 a:1576 time:0
recv size:128 a:1577 time:0
recv size:128 a:1578 time:0
```

1.13 本章总结

至此，我们本章的内容就讲完了。我们实现了一个网络模块。但是需要完善优化的还有很多地方。比如异常的处理，数据有效性检查，还有经常有人提到的“粘包”问题，这些就留给大家去研究，去自我提高吧。

2016.1.3 作者：司马威
作者博客 <http://blog.csdn.net/smwhotjay>

编程交流 q 群: 316641007