附录一 简明 Socket 编程指南

在本说明文档中，主要讲述了一些网络 SOCKET 编程的基本概念和有关函数说明，并给出了部分示例程序的源代码。在完成“TCP 和 IP 通信程序设计实验”、“ 实时声音传输实验”和“HTTP 代理实现实验”时，可以参考本文档的内容。

一、SOCKET 基本概念

1 Linux/Unix：Socket 函数库

Linux Socket 函数库是从 Berkeley 大学开发的 BSD UNIX 系统中移植过来的。BSD Socket 接口是在众多 Unix 系统中被广泛支持的 TCP/IP 通信接口，Linux 下的 Socket 程序设计，除了微小的差别之外，也适用于大多数其它 Unix 系统。

Socket 的使用，和文件操作比较类似。如同文件的读、写、打开、关闭等操作一样，TCP/IP 网络通信同样也有这些操作，不过它使用的接口不是文件描述符或者 FILE*，而是一个称做 Socket 的描述符。类似于文件操作，对于 Socket，也通过读、写、打开、关闭操作来进行网络数据传送。同时，还有一些辅助的函数，如域名/IP 地址查询、Socket 功能设置等。

2. DOS：WATTCP 函数库

关于 WATTCP 函数库，也就是 DOS 下 TCP/IP 协议驱动程序库是由加拿大 Waterloo 大学的 Erick Engelke 提供的。NCSA 的 telnet、ftp 等程序，都是利用 Erick Engelke 的 TCP/IP 程序库 WATTCP 开发出来的。WATTCP 是一个很容易使用的 C 语言函数库，相对 Unix 和其它环境下广泛使用的 BSD Socket 接口而言，它在 DOS 下提供了一个更为简单和直观的 TCP/IP 编程接口。

WATTCP 编程接口，相对于 Unix Socket 而言，作了相当的简化。Unix 下，网络操作和文件系统几乎融为一体，但 DOS 下不存在如 Unix 一样强大的网络文件系统功能。因此，在 DOS 下没有 Unix 中那种直接用文件系统调用来操作 Socket 的便利。DOS 下的 TCP/IP 通信和 DOS 系统几乎是完全独立的。WATTCP 支持 DOS TCP/IP 环境下的基本 Socket 接口，大致和 Unix Socket 兼容，包括面向连接的 TCP（SOCK_STREAM）和非连接的 UDP（SOCK_DGRAM）型 Socket。另外 WATTCP 提供一些专用的 SOCKET 接口函数。

3. Windows Sockets 规范

Windows Sockets 规范以 U.C. Berkeley 大学 BSD UNIX 中流行的 Socket 接口为范例定义了一套 Micosoft Windows 下网络编程接口。它不仅包含了人们所熟悉的 Berkeley Socket 风格的库函数；也包含了一组针对 Windows 的扩展库函数，以使程序员能充分地利用 Windows 消息驱动机制进行编程。

这一套 Windows Sockets API 能够在所有 3.0 以上版本的 Windows 和所有 Windows

Scokets 实现上使用，所以它不仅为 Windwos Sockets 实现和 Windows Sockets 应用程序提供了 16 位操作环境，而且也提供了 32 位操作环境。

Windows Sockets 也支持多线程的 Windows 进程。一个进程包含了一个或多个同时执行的线程。在 Windows 3.1 非多线程版本中，一个任务对应了一个仅具有单个线程的进程。而我们在本书中所提到的线程均是指在多线程 Windows 环境中的真正意义的线程。在非多线程环境中（例如 Windows 3.0）这个术语是指 Windows Sockets 进程。

Windows Sockets 规范中的针对 Windows 的扩展部分为应用程序开发者提供了开发具有 Windows 应用软件的功能。它有利于使程序员写出更加稳定并且更加高效的程序，也有助于在非占先 Windows 版本中使多个应用程序在多任务情况下更好地运作。除了 WSAStartup() 和 WSACleanup()两个函数除外，其他的 Windows 扩展函数的使用不是强制性的。

4. 套接口基本概念

通讯的基石是套接口，一个套接口是通讯的一端。在这一端上你可以找到与其对应的一个名字。一个正在被使用的套接口都有它的类型和与其相关的进程。套接口存在于通讯域中。通讯域是为了处理一般的线程通过套接口通讯而引进的一种抽象概念。套接口通常和同一个域中的套接口交换数据（数据交换也可能穿越域的界限，但这时一定要执行某种解释程序）。Windows Sockets 规范支持单一的通讯域，即 Internet 域。各种进程使用这个域互相之间用 Internet 协议族来进行通讯（Windows Sockets 1.1 以上的版本支持其他的域，例如 Windows Sockets 2）。

套接口可以根据通讯性质分类；这种性质对于用户是可见的。应用程序一般仅在同一类的套接口间通讯。不过只要底层的通讯协议允许，不同类型的套接口间也照样可以通讯。

用户目前可以使用两种套接口，即流套接口和数据报套接口。流套接口提供了双向的，有序的，无重复并且无记录边界的数据流服务。数据报套接口支持双向的数据流，但并不保证是可靠，有序，无重复的。也就是说，一个从数据报套接口接收信息的进程有可能发现信息重复了，或者和发出时的顺序不同。数据报套接口的一个重要特点是它保留了记录边界。对于这一特点，数据报套接口采用了与现在许多包交换网络（例如以太网）非常类似的模型。

二、SOCKET 编程原理

1. 套接口网络编程原理

套接口有三种类型：流式套接口，数据报套接口及原始套接口。

流式套接口定义了一种可靠的面向连接的服务，实现了无差错无重复的顺序数据传输。数据报套接口定义了一种无连接的服务，数据通过相互独立的报文进行传输，是无序的，并且不保证可靠，无差错。原始套接口允许对低层协议如IP或ICMP直接访问，主要用于新的网络协议实现的测试等。

无连接服务器一般都是面向事务处理的，一个请求一个应答就完成了客户程序与服务程序之间的相互作用。若使用无连接的套接口编程，程序的流程可以用图3-1表示。

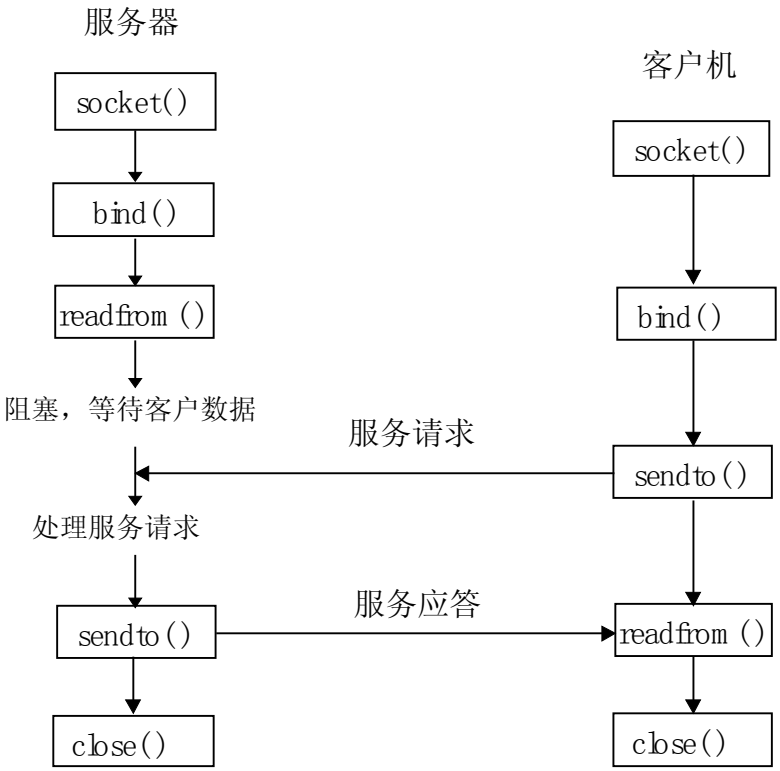

图 3-1 无连接套接口应用程序时序图

面向连接服务器处理的请求往往比较复杂，不是一来一去的请求应答所能解决的，而且往往是并发服务器。使用面向连接的套接口编程，可以通过图3-1来表示：其时序。

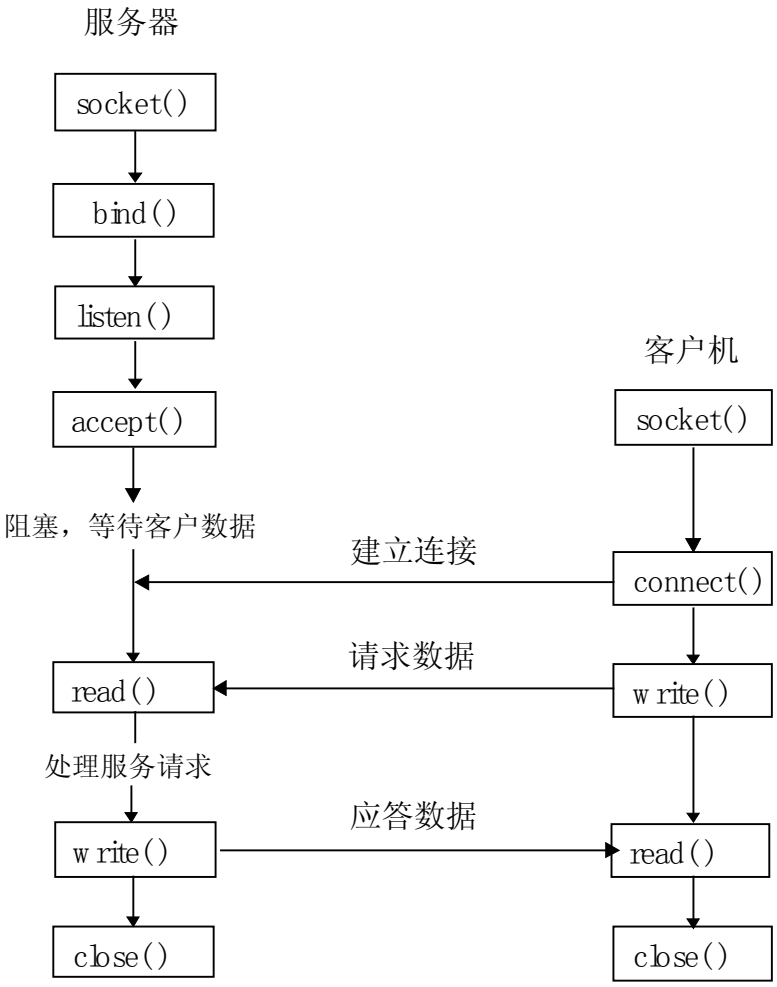

图 3-2 面向连接套接口应用程序时序图

套接口工作过程如下：服务器首先启动，通过调用socket()建立一个套接口，然后调用bind()将该套接口和本地网络地址联系在一起，再调用listen()使套接口做好侦听的准备，并规定它的请求队列的长度，之后就调用accept()来接收连接。客户在建立套接口后就可调用connect()和服务器建立连接。连接一旦建立，客户机和服务器之间就可以通过调用read()和write()来发送和接收数据。最后，待数据传送结束后，双方调用close()关闭套接口。

2. Windows Sockets 编程原理

由于Windows的基于消息的特点，WINSOCK和BSD套接口相比，有如下一些新的扩充：

1.异步选择机制

异步选择函数 WSAAsyncSelect()允许应用程序提名一个或多个感兴趣的网络事件，如FD_READ，FD_WRITE，FD_CONNECT，FD_ACCEPT 等等代表的网络事件。当被提名的网络事件发生时，Windows 应用程序的窗口函数将收到一个消息。这样就可以实现事件驱动了。

2.异步请求函数

异步请求函数允许应用程序用异步方式获得请求的信息，如 WSAAsyncGetXByY()类函数。 这些函数是对 BSD 标准函数的扩充。函数 WSACancelAsyncRequest()允许用户中止一个正在执行的异步请求。

3.阻塞处理方法

WINSOCK 提供了"钩子函数"负责处理 Windows 消息，使 Windows 的消息循环能够继续。WINSOCK 提供了两个函数(WSASetBlockingHook()和 WSAUnhookBlockingHook())让应用程序设置或取消自己的"钩子函数"。函数 WSAIsBlocking()可以检测是否阻塞，函数 WSACancelBlockingCall()可以取消一个阻塞的调用。

4.错误处理

WINSOCK 提供了两个 WSAGetLastError()和 WSASetLastError()来获取和设置最近错误号。

5.启动和终止

由于 Windows Sockets 的服务是以动态连接库 WINSOCK.DLL 形式实现的，所以必须要先调用 WSAStartup()函数对 Windows Sockets DLL 进行初始化，协商 WINSOCK 的版本支持，并分配必要的资源。在应用程序关闭套接口后，还应调用 WSACleanup()终止对 Windows Sockets DLL 的使用，并释放资源，以备下一次使用。

在这些函数中，实现 Windows 网络实时通信的关键是异步选择函数 WSAAsyncSelect()的使用。

三、SOCKET 函数库介绍

1. DOS：WATTCP 函数库

兼容型 Socket 函数

WATTCP 支持 DOS TCP/IP 环境下的基本 Socket 接口，大致和 Unix Socket 兼容，包括面向连接的 TCP（SOCK_STREAM）和非连接的 UDP（SOCK_DGRAM）型 Socket。其主要的函数有：

WATTCP 初始化：

sock_init()：初始化 TCP/IP 驱动程序，建立和 pkt driver 的调用关系；
sock_exit()：清除 TCP/IP 驱动程序和 pkt driver 的联系；
由于 DOS 环境下，TCP/IP 驱动程序是完全独立的，为了正常使用，必须首先初始化，使用完毕后，必须清除现场，以免影响其它程序的运行。

Socket 创建和释放：

socket()、n_close()

连接建立和撤销：

bind()、listen()、accept()、connect()

Socket 发送和接收：

n_read()、n_write()、select()

数据转换函数

gethostbyname()、inet_addr()、inet_ntoa()
这些函数中，n_read()、n_write()、n_close()分别对应于 Unix Socket 接口中的 read()、write()、close()函数，因为需要和 DOS 系统下的文件操作函数区分开，用了另外的名字。
以上这些函数，其含义和用法和标准的 Unix Socket 接口基本一致，是实验中应该掌握并学会使用的部分。

专用 Socket 接口函数

另外，WATTCP 函数库是相当开放的，上面的 Socket 接口是层次比较高的接口，对 Socket 的控制功能稍微弱一些，如，函数 n_read()和 n_write()都不是 non_blocking 型的，没有提供 non_blocking 选项。如果需要对 TCP Socket 进行更强的控制，可以完全弃之不用，而改用较为低级的 TCP/UDP 接口函数，但是，这要求直接使用 WATTCP 的内部数据结构 Tcp_Socket，对 WATTCP 的内部实现机制有比较清楚的了解，并在程序中间进行必要的协调。具体的函数有：

TCP/IP 接口初始化：

sock_init()：初始化 TCP/IP 驱动程序，建立和 pkt driver 的调用关系；
sock_exit()：清除 TCP/IP 驱动程序和 pkt driver 的联系；

Socket 操作函数：

tcp_listen()、tcp_accept()：被动等待建立 TCP 连接；
tcp_open()、udp_open()：主动建立 TCP/UDP 连接；
sock_close()：关闭 Socket；

Socket 读写函数：

sock_read()、sock_write()
非 non_blocking 的 Socket 读写函数；
sock_fastread()、sock_fastwrite()
sock_read()和 sock_write()的 non_blocking 版本；
sock_flush()
完全发送缓冲区中的数据；

Socket 等待型操作的容错处理：sock_wait_xxx 宏：

sock_wait_established　　tcp_open 之后，等待连接建立的完成；
sock_wait_input　　连接建立之后，等待接收网络数据；
sock_wait_closed　　sock_close 之后，等待连接撤销的完成；
关于这些函数的使用，请参见相应的示例。

2.Linux/Unix：Socket 函数库

Socket 操作：

Socket()：分配 Socket

#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain， int type， int protocol);

socket()函数分配一个 Socket 句柄，用于指定特定网络下、使用特定的协议和数据传送方式进行通信。Socket 接口是不仅仅局限于 TCP/IP 的，但是由于 TCP/IP 的广泛使用，它们几乎被完全等同起来了。

Socket 句柄分配以后，如果要开始 TCP 通信，还需要建立连接。根据需要，可以主动地建立连接（通过 connect()）和被动地等待对方建立连接（通过 listen()），在连接建立后才能使用读写操作通过网络连接进行数据交换。

参数说明：

domain　－domain 参数选择通信中使用的协议族，也就是网络的类型，可以是以下之一：
AF_UNIX　(UNIX 内部协议)
AF_INET　(ARPA Internet 协议，也就是 TCP/IP 协议族，亦即我们实验中所使用的)
AF_ISO　(ISO 协议)
AF_NS　(Xerox Network Systems 协议)
AF_IMPLINK (IMP "host at IMP" link layer)

type －数据传送的方式，可以是以下之一：
SOCK_STREAM：保证顺序的、可靠传送的双向字节数据流，最为常用，也是 TCP 连接所使用的方式。
SOCK_DGRAM：无连接的、不保证可靠的、固定长度（通常很小）的消息传送。
SOCK_SEQPACKET：顺序的、可靠的双向固定长度的数据包传送，只用于 AF_NS 类型的网络中。
SOCK_RAW：原始的数据传送，适用于系统内部专用的网络协议和接口，和 SOCK_RDM 一样，只能由超级用户使用。
SOCK_RDM：可靠的数据报传送，未实现。

protocol　－protocol 参数指定通信中使用的协议。在给定 Socket 的协议族和传送类型之后，一般情况下所使用的协议也就固定下来，如下表所示，此时 protocol 参数可使用缺省值'0'；但如果还有多个协议供选择，则必须使用 protocol 参数来标识。

协议族（仅考虑 IP	传送类型	protocol 参数常量	协议类型

协议族）		（/usr/include/linux/in.h）	
AF_INET	SOCK_STREAM SOCK_DGRAM SOCK_RAW SOCK_RAW	IPPROTO_TCP IPPROTO_UDP IPPROTO_ICMP IPRROTO_RAW	TCP UDP ICMP (raw)

返回值：
正常执行时，返回 Socket 描述符；否则，返回-1，错误状态在全局变量 errno 中。

close()：关闭 Socket

#include <unistd.h>
int close(int fd);
Socket 和文件描述符的关闭操作都是使用这个函数。
参数说明：fd －Socket 描述符。　　返回值：正常时返回 0，-1 表示出错。

bind()：给 Socket 指定本地地址

#include <sys/types.h>
#include <sys/socket.h>
int　bind(int sockfd，　struct sockaddr *my_addr，　int addrlen);
　　bind 函数给已经打开的 Socket 指定本地地址。这个函数的使用有以下两种情况：
－如果此 Socket 是面向连接的，而且此 Socket 在连接建立过程中处于被动的地位，即，己方程序使用 listen 函数等待对方建立连接，对方用 connect 函数来向此 Socket 建立连接，这种情况下，必须用 bind 给此 Socket 设定本地地址。在己方使用 listen 函数时，除指定 Socket 描述符之外，该 Socket 必须已经用 bind 函数设定好了本地地址（包括 IP 地址和端口号），这样，系统在收到建立连接的网络请求时，才能根据请求的目的地址，识别是通向哪个 Socket 的连接，从而己方才能用此 Socket 接收到发给此 Socket 地址的数据包。不指定 Socket 的本地地址，就无法将此 Socket 用于连接建立和数据接收。
　　如果此 Socket 用于无连接的情形，同样也要求给该 Socket 设定本地地址，这样，以后系统从网络中接收到数据后，才知道该送给哪个 Socket 及其相对应的进程。
参数说明：
sockfd　－Socket 描述符。
addrlen　－my_addr 结构的长度。
my_addr －用于侦听连接请求的本地地址。struct sockaddr 是一个通用型的结构，不仅包含 TCP/IP 协议的情况，同时也是为了适合于其它网络，如 AF_NS。由于它的这种通用性，它只是定义了一个一般意义上的存储空间，如/usr/include/linux/socket.h 中所示：

```
struct sockaddr
{
    unsigned short sa_family;    /* address family,  AF_xxx */
    char           sa_data[14]; /* 14 bytes of protocol address */
};
```

当使用 TCP/IP 协议（即：Internet 协议）时，可用如下的 struct 等价地代替 struct sockaddr（/usr/include/linux/in.h）：

```
#define __SOCK_SIZE__     16              /* sizeof(struct sockaddr)      */
struct sockaddr_in {
    short int          sin_family;  /* Address family */
    unsigned short int sin_port;    /* Port number */
    struct in_addr     sin_addr;    /* Internet address */
    /* Pad to size of `struct sockaddr'. */
    unsigned char      __pad[__SOCK_SIZE__ - sizeof(short int) -
                       sizeof(unsigned short int) - sizeof(struct in_addr)];
};
```

在 Socket 程序中，等待建立连接一方的准备过程请参见编程实例，以及 listen()、accept()的说明。
返回值：正常时返回 0，否则返回-1，同时 errno 是系统错误码。

listen()：准备接受连接请求。

```
#include <sys/socket.h>
int listen(int s,  int backlog);
```

在用 bind()给一个 Socket 设定本地地址之后，就可以将这个 Socket 用于接受连接请求，即 listen()。调用 listen()之后，系统将给此 Socket 配备一个连接请求的队列，暂存系统接收到的、申请向此 Socket 建立连接的请求，等待用户程序用 accept()正式接受该请求。队列长度，就由 backlog 参数指定。如下面的简图所示：

通信己方 Me		通信对方	
	Socket_Me	Peer Sockets	
连接请求暂存队列	[0]<--连接建立请求 1	<-- connect()	Socket_peer_1
	[1]<--连接建立请求 2	<-- connect()	Socket_peer_2
	……	……	……
	[backlog-1]<--连接建立请求 n	<-- connect()	Socket_peer_n

如果短时间内向己方建立连接的请求过多，己方来不及处理，那么排在 backlog 之后的请求将被系统拒绝。因此，backlog 参数实际上规定了己方程序能够容许的连接建立处理速度。至于己方程序使用此 Socket（及其指定的本地地址）实际建立连接的个数，由己方程序调用 accept()的次数来决定，参见 accept()的说明。
参数说明：s－Socket 描述符。backlog－连接请求暂存队列长度。
返回值：正常时返回 0；否则返回-1，同时 errno 是系统错误码。

accept：接受指定 Socket 上的连接请求

```
#include <sys/types.h>
#include <sys/socket.h>
int accept(int s,  struct sockaddr *addr,  int *addrlen);
```

在调用 listen()之后，系统就在 Socket 的连接请求暂存队列里存放每一个向该 Socket（及其本地地址）建立的连接请求。accept()函数的作用就是，从该暂存队列中取出一个连接请求，用该 Socket 的数据，创建一个新的 Socket：Socket_New，并为它分配一个文件描述符。

Socket_New 即标识了此次建立的连接，可被己方用来向连接的另一方发送和接收数据（write/read，send/recv）。同时，原 Socket 仍然保持打开状态不变，继续用于等待网络连接请求。

如果该 Socket 的暂存队列中没有待处理的连接请求，根据 Socket 的特征选项（是否 non_blocking），accept()函数将选择两种方式：如果该 Socket 不是 non_blocking 型的，accept()将一直等待，直到收到一个连接请求后才返回；如果该 Socket 是 non_blocking 型的，那么 accept()将立即返回，但如果没有连接请求，只返回错误信息，不创建新的 Socket_New。accept()返回后，如果创建了新的 Socket_New 来标识新建立的连接，那么参数 addr 指定的结构里面将会有对方的地址信息，addrlen 是地址信息的长度。

关于 accept()的进一步信息，如：如何检测某 Socket 有无待处理的连接请求、如何在使用 accept()接受连接请求之前先获取连接对方的地址、如何根据获取的对方地址信息拒绝该连接请求等，请参阅 Linux manual，此处不再累述。

参数说明：

s　　－Socket 描述符。

addr　　－accept()接受连接后，在 addr 指向的结构中存放对方的地址信息。如果是 AF_INET Socket，该地址信息就是对方的 IP 地址和端口号。

addrlen　－在调用 accept()之前，*addrlen 必须被设置为 addr 数据结构的合法长度。在 accept()返回之后，*addrlen 中是对方地址信息的长度。

返回值：如果正常创建了一个新的连接，那么返回非负的整数：即新连接的 Socket 描述符（注意，用于等待连接请求的原 Socket 保持打开状态不变，可用于接收新的连接请求。）；否则，返回-1，errno 是系统错误码。

connect：建立连接

```
#include <sys/types.h>
#include <sys/socket.h>
int   connect(int sockfd，  struct sockaddr *serv_addr，  int addrlen );
```

前面提到的函数，如 bind、listen、connect 等，都是用于被动地等待对方建立连接时需要使用的，而 connect()函数，则是主动地向对方建立连接时使用的。connect()使用一个事先打开的 Socket，和目的方（即通信对方，或称服务器一方）地址信息，向对方发出连接建立请求。一个完整的 Socket 通信发起过程可简单地图示为：

主动发起方（客户方）		被动接受方（服务方）
		listen(sock_w);等待连接建立请求
connect(sock_s)		
{		
发送连接建立请求	－>	
		sock_d = accept();接受连接建立请求
	<－	发送应答
收到连接建立应答		
}		

此时，在 sock_s 和 sock_d 之间，一个连接就建立完毕。

如果是 SOCK_STREAM 型的 Socket，通常只用 connect()建立一个正常的连接。但如果是

SOCK_DGRAM 型的 Socket，connect()函数并不象上图中那样向目的方发出连接建立请求，而只是简单地用给出的地址设置该 Socket 的目的地址，以后该 Socket 的无连接数据报就发往该目的地址。因此，对于 SOCK_DGRAM 型的 Socket，可以多次调用 connect()来改变该 Socket 的目的地址。SOCK_DGRAM 型的 Socket 与本实验关系不大，故不再详述。
参数说明：
sockfd　－Socket 描述符。
serv_addr－通信目的方的地址。其格式参见 bind()的说明。
addrlen　－目的地址长度。
返回值：连接正常建立时返回 0；否则，返回-1，系统错误码在 errno 中。

send/recv：用 Socket 发送和接收数据

```
#include <sys/types.h>
#include <sys/socket.h>
int send(int s， const void *msg， int len， unsigned int flags);
int sendto(int s， const void *msg， int len， unsigned int flags，
const struct sockaddr *to， int tolen);
int recv(int s， void *buf， int len， unsigned int flags);
int recvfrom(int s， void *buf， int len， unsigned int flags， struct sockaddr *from， int *fromlen);
```

在连接建立完成后，通信双方就可以使用以上这些函数来进行数据的发送和接收操作。其中，send 和 recv 用于连接建立以后的发送和接收；sendto 和 recvfrom 用于非连接的协议。对于非 non_blocking 型的 Socket，send 将等待数据发送完后才返回；对于 non_blocking 型的 Socket，send 将立即返回，用户程序需要用 select()函数决定网络发送是否结束。类似地，对于非 non_blocking 型的 Socket，若系统没有收到任何数据，recv 将等待接收数据到达后才返回；对于 non_blocking 型的 Socket，recv 将立即返回，并返回错误信息或者接收到的数据字节数。sendto 和 recvfrom 因为是非连接型的发送和接收，必须在参数中给出目的地址或者存放源地址的空间。
参数说明：
s　－Socket 描述符；
msg， buf　－存放接收或者发送数据的存储空间；
len　－接收或者发送数据的字节数；
to，from －sendto 和 recvfrom 所使用的，目的方地址和存放源地址的空间；
tolen，fromlen　－目的地址和源地址空间大小。
flag　－通常设为 0，详细说明请参见 Linux Manual。
返回值：
send/sendto 返回实际发送的数据字节数，或者-1，表示出错；
recv/recvfrom 返回实际接收到的数据字节数，或者-1，表示出错。

read/write：用系统文件操作进行 Socket 通信

```
#include <unistd.h>
ssize_t read(int fd， void *buf， size_t count);
ssize_t write(int fd， const void *buf， size_t count);
```

在连接建立完成后，对于连接建立过程中被动的一方，在 accept()正常返回后，它返回一个新的 Socket，并且为该 Socket 分配了一个文件描述符；对于连接请求发起方，connect()正常返回后，相应的 Socket 中也包含有已分配的文件描述符。因此，可以使用标准的 Unix 文件读写函数 read()/write()来进行 Socket 通信。要注意的是，由于网络数据和磁盘文件不一样，不是已经准备好的，因此，每次读写操作不一定能传送完指定长度的数据，需要由程序反复进行剩余部分的传送。

另外，文件描述符是较底层的文件操作参数，不同于 C 语言中常用的 FILE*。FILE*是使用 fread/fwrite 函数来进行读写操作的。

参数说明：

fd　　－文件或者 Socket 描述符。

buf　　－数据缓冲区。

count　　－数据字节数。

返回值：正常时，返回所读写的字节数（注意，可能小于 count 参数指定的数目）；否则，返回-1，errno 是系统错误码。

getsockopt/setsockopt：获取、设置 Socket 特征选项。

```
#include <sys/types.h>
#include <sys/socket.h>
int getsockopt(int　s，　int level，　int optname，　void *optval，　int *optlen);
int setsockopt(int s，　int level，　int optname，　const　void *optval，　int optlen);
```

由于在普通的 Socket 编程中很少涉及这些选项，在这里不作介绍。

non_blocking 特性：

由于前面多处提到 non_blocking 特性，这里介绍一下如何设置这种特性。在这里，我们又可以看到 Socket 和文件描述符在 Unix 系统中的相似性。实际上 non_blocking 特性也是通过 Unix 文件操作函数来设置的：

```
#include <unistd.h>
#include <fcntl.h>
fcntl(socket，　F_SETFL，　O_NONBLOCK);
```

数据格式转换

```
#include <netinet/in.h>
unsigned long int htonl(unsigned long int hostlong);
unsigned short int htons(unsigned short int hostshort);
unsigned long int ntohl(unsigned long int netlong);
unsigned short int ntohs(unsigned short int netshort);
```

数据格式转换函数提供和硬件平台无关的、主机数据字节顺序和网络字节顺序之间的转换。由于各种平台 CPU 结构的不同，在不同的硬件平台下，主机的字节顺序有两种情况：Intel 80x86 和 SUN Sparc CPU 的低位在前格式和 Motorola CPU（68000、PowerPC）等的高位在前格式。网络数据交换要求网络中所有的 int 型数据都有统一的字节顺序：高位在前格式，因此在 Socket 函数库中提供了以上统一的字节顺序转换函数。

在 Socket 程序中使用的地址数据，如端口号等，都必须遵循这样统一的字节顺序。因

此，在本实验的例子程序中，在 bind()函数、connect()函数等涉及 struct sockaddr_in 地址数据的场合，都可以看到以上转换函数的使用，以加强源程序的可移植性。

主机名字/地址数据查询

为配合 DNS 的使用，尽量方便 Internet 主机名字的记忆，避免使用烦琐的数字式 IP 地址，Socket 函数库提供了方便的主机名字查询函数。

struct hostent

struct hostent 是一个关于主机地址信息的数据结构，其中包含从 DNS 服务器得到的比较全面的主机信息。gethostbyname()和 gethostbyaddr()都返回这样的数据结构。实际使用时，可用此结构中的地址信息来设置 bind()和 connect()函数参数中的 struct sockaddr_in 中的地址，以支持 DNS 名字的使用。参见本实验的具体示例。

```
#include <netdb.h>
struct      hostent {
    char *h_name;       /* official name of host */
    char **h_aliases;   /* alias list */
    int    h_addrtype;  /* host address type */
    int    h_length;    /* length of address */
    char **h_addr_list; /* list of addresses from name server */
};
#define   h_addr    h_addr_list[0]    /* address,  for backward compatibility */
```

这其中最常用的是 h_addr，即主机的缺省地址（因为该主机名字可能对应多个地址）。

gethostbyname

```
#include <netdb.h>
struct hostent *gethostbyname(char *name);
```

根据 DNS 名字，查找主机地址信息。name 可以是 DNS 名字，如”www.ustc.edu.cn”，也可以是 IP 地址串：”202.38.64.2”。

gethostbyaddr

```
#include <netdb.h>
struct hostent *gethostbyaddr(char *addr; int len,   int type);
```

根据 IP 地址查找主机地址信息。addr 是整数格式的 IP 地址指针，如：unsigned char addr[4]＝{202，38，64，2}，在 Internet 协议中，len 必须为 4，type 必须为 AF_INET。

要注意的是：如果只知道主机的 IP 地址，而且 DNS 服务器中没有登记该主机，用 gethostbyname 总能得到适当的主机地址信息，它只需要简单地将 ASCII 形式的 IP 地址转换为二进制格式。但如果使用 gethostbyaddr，却得不到所需要的地址信息，因为此函数完全依靠 DNS 服务器进行 IP 到 DNS 名字的转换，不作其它的处理。

关于 Socket 编程接口，就介绍以上这些。作为普通的 TCP/IP 应用，它们已经足够了。在本实验的示例中，可以看到它们的具体应用。关于 Socket 接口的进一步资料，请参考更

详细的资料，如 Linux Manual 等。

3. Windows Socket 1.1 库函数

Winsock 编程接口也是直接从 Socket 接口移植过来的，以上介绍的 Linux Socket 函数，在 Winsock 环境下都可以直接使用。但是，由于 Windows 环境的特点，Microsoft 作了相当的扩充，增加了以下一些功能的函数：

－Winsock 驱动程序的检测、初始化、清除函数，和错误代码函数（因为 Windows 环境下没有全局的错误代码变量 errno），如 WSAStartup()、WSACleanup()、WSAGetLastError()、WSASetLastError()；

－针对 Windows 窗口界面的异步编程方式，提供了一些异步的主机数据查询函数，如 WSAAsyncGetHostByAddr()、WSAAsyncGetHostByName()等；

－对于数据接收和发送，也提供了异步的操作方式，如 WSAAsyncSelect()、WSAIsBlocking()、WSASetBlockingHook()、WSAUnhookBlockingHook()、WSACancelBlockingCall()、WSACancelAsyncRequest()等。

Windows 环境下的程序设计远不如 Unix 和 DOS 下简明，不利于集中体现 Socket 编程的特点，因此，我们的实验中对 Windows 下的 Socket 编程不作要求，对 Winsock 接口的介绍，也仅限于此。详细信息，可参阅 MS VC2.0 以上版本的联机帮助文件中关于 Winsock 的文档。在本实验室的 NT 服务器上存放有《Windows Sockets 规范及应用－Windows 网络编程接口》一书，可供查阅。

可以提一下的是，在 Windows95/NT 环境下，其 Console 应用程序（一种可以在 Win95/NT DOS 窗口下运行的 32 位程序，当然，它仍然有 DOS 程序的 1MB 内存限制）可以直接使用 Unix Socket 接口进行 Socket 程序设计（仅需要增加 WSAStartup 和 WSACleanup 的使用），在我们的校园网上，就有经过这样改造的 Win95 环境下的 BBS 客户程序。

四、程序示例

在这里，我们给出 Linux 和 DOS 环境下两个简单的 Socket 通信程序的例子，一个是服务程序，它打开一个端口等待接收，并原封不动地传回所收到的数据；另一个例子是客户程序，它建立 TCP 连接到服务器，将用户的输入一行一行地发送给服务器，并显示服务器发送回来的数据。

实际实验时，最好不要完全照搬这里的代码，那只能算是完成了实验的基本操作。这里强调的是大家对 TCP/IP 通信编程的理解和创意。

Linux Socket 通信程序示例一：服务程序

此服务程序等待客户建立一个到 7000 端口的连接，并原封不动地发回所有收到的数据，直到连接断开。

server.c

```
#include        <stdio.h>
#include        <string.h>
#include        <netinet/in.h>
```

```
#define PORT      7000

main ()
{
        struct sockaddr_in        client，  server;
        int                  s，  ns，  namelen，  pktlen;
        char      buf[256];

        s=socket(AF_INET，  SOCK_STREAM，  0);
        memset ((char *)&server，  sizeof(server)，  0);
        server.sin_family = AF_INET;
        server.sin_port = htons(PORT);
        server.sin_addr.s_addr = INADDR_ANY;
        bind(s，  (struct sockaddr *)&server，  sizeof(server));
        listen(s，1);
        namelen = sizeof (client);
        ns = accept (s，  (struct sockaddr *)&client，  &namelen);
        for (;;) {
                pktlen = recv (ns，  buf，  sizeof (buf)，  0);
                if (pktlen == 0)
                        break;
                printf ("Received line: %s\n"，  buf);
                send (ns，  buf，  pktlen，  0);
        }
        close(ns);
        close(s);
}
```

Linux Socket 通信程序实例二：直接使用 IP 地址的客户程序

client.c

```
#include        <stdio.h>
#include        <string.h>
#include        <errno.h>
#include        <netinet/in.h>

#define PORT              7000
#define HOST_ADDR         "202.38.75.33"

main ()
{
        struct sockaddr_in        server;
        int       s，  ns;
        int       pktlen，  buflen;
```

```
        char    buf1[256]， buf2[256];

        s=socket(AF_INET， SOCK_STREAM， 0);
        server.sin_family = AF_INET;
        server.sin_port = htons(PORT);
        server.sin_addr.s_addr = inet_addr (HOST_ADDR);
        if (connect(s， (struct sockaddr *)&server， sizeof(server)) < 0)
        {
                perror("connect()");
                return;
        }
        for (;;) {
                printf ("Enter a line: ");
                gets (buf1);
                buflen = strlen (buf1);
                if (buflen == 0)
                        break;
                send(s， buf1， buflen + 1， 0);
                recv(s， buf2， sizeof (buf2)， 0);
                printf("Received line: %s\n"， buf2);
        }
        close(s);
}
```

Linux Socket 通信程序实例三：使用 DNS 名字服务的客户程序

client_dns.c

```
#include        <stdio.h>
#include        <string.h>
#include        <errno.h>
#include        <netinet/in.h>
#include        <netdb.h>

#define PORT                7000
#define HOST_ADDR           "Beauty.eeis.ustc.edu.cn"

main ()
{
        struct sockaddr_in      server;
        struct hostent          *hp;

        int             s， ns;
        int             pktlen， buflen;
        char    buf1[256]， buf2[256];
```

```
        s=socket(AF_INET， SOCK_STREAM， 0);
        server.sin_family = AF_INET;
        server.sin_port = htons(PORT);
        hp=gethostbyname(HOST_ADDR);
        memcpy (hp->h_addr， &server.sin_addr， hp->h_length);
        if (connect(s， (struct sockaddr *)&server， sizeof(server)) < 0)
        {
                perror("connect()");
                return;
        }
        for (;;) {
                printf ("Enter a line: ");
                gets (buf1);
                buflen = strlen (buf1);
                if (buflen == 0)
                        break;

                send(s， buf1， buflen + 1， 0);
                recv(s， buf2， sizeof (buf2)， 0);

                printf("Received line: %s\n"， buf2);
        }
        close(s);
}
```

Windows sock 示例一：使用域名或 IP 地址发送数据

```
#include <winsock.h>
#include <stdio.h>
#include <conio.h>
#include <sys/types.h>
#include <sys/timeb.h>

void main(int argc，char **argv)
{
    char buf[65535];
    struct sockaddr_in client;
    struct hostent *hp;
    int s;
    unsigned long ip_addr;
    int sendlen;
    int pktlen;
    int i;
```

```
int waittm;
unsigned long sendtimes;
struct timeb starttime;
struct timeb endtime;
float timespers;
struct timeval tv;
fd_set rfds;
WSADATA    wsa_dat;

if(argc<4){
    printf("Usage:\n\tsend Hostname|hostaddr portNo. packetlen \n");
    exit(0);
}
if(0!=WSAStartup(0x0101，&wsa_dat)){
    perror("WSAStartup()");
    exit(0);
}
if((hp=gethostbyname(argv[1]))==NULL){
    ip_addr=inet_addr(argv[1]);
    hp=gethostbyaddr((char *)&ip_addr，4，PF_INET);
}
if(hp==NULL){
    perror("Gethostbyaddr()");
    exit(0);
}
memcpy(&client.sin_addr，hp->h_addr，hp->h_length);
if((s=socket(AF_INET，SOCK_STREAM，0))<0){
    perror("Socket()");
    exit(0);
}
client.sin_family=AF_INET;
client.sin_port=htons(atoi(argv[2]));
if(connect(s，(struct sockaddr*)&client，sizeof(client))<0){
    perror("Connect()");
    closesocket(s);
    exit(0);
}
waittm=atoi(argv[4]);
pktlen=atoi(argv[3]);
for(i=0;i<pktlen;i++)
    buf[i]=i;
sendtimes=0;
ftime(&starttime);
while(1){
```

```
            if((sendlen=send(s，buf，pktlen，0))<=0){
                perror("Send()");
                break;
            }
            sendtimes++;
            if(kbhit())
                break;
        }
        ftime(&endtime);
        closesocket(s);
        WSACleanup();
        printf("send times:%ul\n"，sendtimes);
        printf("time used:%d seconds and %d miliseconds\n"，endtime.time-starttime.time
                ，endtime.millitm-starttime.millitm);
        printf("%f times per second\n"，timespers=sendtimes/((float)(endtime.time- starttime.time)
                +((float)(endtime.millitm-starttime.millitm))/1000));
        printf("%d Bytes per packet\n"，pktlen);
        printf("%f Bits per second\n"，timespers*pktlen*8);
        return;
}
```

Windows socket 示例二：接收数据

```
#include <winsock.h>
#include <stdio.h>
#include <conio.h>
#include <sys/types.h>
#include <sys/timeb.h>

main(int argc，  char **argv)
{
        char buf[65535];
        struct sockaddr_in server;
        struct sockaddr_in client;
        int clientnamelen;
        int s;
        int clientfd;
        int pktlen;
        int pktlentmp;
        int tmp;
        int flag;
        unsigned long recvtimes;
        struct timeb starttime;
        struct timeb endtime;
```

```
float timespers;
fd_set rfds;
struct timeval tv;
int retval;
WSADATA wsa_dat;

if(argc<2){
    printf("Usage:\n\trecv portNo.\n");
    exit(0);
}
if(0!=WSAStartup(0x0101，&wsa_dat)){
    printf("winsock init error\n");
    exit(0);
}
if ((s=socket(AF_INET， SOCK_STREAM，0))<0){
    perror("Socket()");
    return;
}
server.sin_family = AF_INET;
server.sin_port = htons(atoi(argv[1]));
server.sin_addr.s_addr = INADDR_ANY;
if (bind(s， (struct sockaddr *)&server， sizeof(server))<0){
    perror("Bind()");
    return;
}
if(listen(s，1)!=0){
    perror("Listen()");
    exit(0);
}
tmp=sizeof(server);
flag=0;
clientnamelen=sizeof(client);
if((clientfd=accept(s，(struct sockaddr *)&client，&clientnamelen))==-1){
    perror("Accept()");
    exit(0);
    }
while(1){
    tv.tv_sec=0;
    tv.tv_usec=1;
    FD_ZERO(&rfds);
    FD_SET(clientfd，&rfds);
    if(retval=select(10，&rfds，NULL，NULL，&tv)){
        if ((pktlentmp = recv(clientfd，buf，65535，0))<=0){
            printf("error:Recv()\n");
```

```
                    break;
                    }
                if(!flag){
                    flag=1;
                    ftime(&starttime);
                    }
                pktlen=pktlentmp;
                recvtimes++;
                ftime(&endtime);
            }
            if(kbhit())
            break;
        }
        closesocket(clientfd);
        closesocket(s);
        WSACleanup();
        printf("recv times:%ul\n"，recvtimes);
        printf("time used:%d seconds and %d miliseconds\n"，endtime.time-
            starttime.time，endtime.millitm-starttime.millitm);
        printf("%f times per second\n"，timespers=recvtimes/((float)(endtime.time-
            starttime.time)+((float)(endtime.millitm-starttime.millitm))/1000));
        printf("%d Bytes per packet\n"，pktlen);
        printf("%f Bits per second\n"，timespers*pktlen*8);
}
```

DOS WATTCP 示例一：使用兼容型高层 Socket 的客户程序

```
#include        <stdio.h>
#include        <string.h>
#include        "tcp.h"

#define PORT                7000
#define HOST_ADDR           "Beauty.eeis.ustc.edu.cn"

main ()
{
        struct sockaddr_in          server;
        struct hostent              *hp;

        int                 s，  ns;
        int                 pktlen，  buflen;
        char        buf1[256]，  buf2[256];
```

```
    sock_init ();
    s=socket(AF_INET， SOCK_STREAM， 0);
    server.sin_family = AF_INET;
    server.sin_port = htons(PORT);
    hp=gethostbyname(HOST_ADDR);
    memcpy (&server.sin_addr， hp->h_addr， hp->h_length);
    if (connect(s， (struct sockaddr *)&server， sizeof(server)) < 0) {
        printf ("connect error.\n");
        exit (1);
    }
    for (;;) {
        printf ("Enter a line: ");
        gets (buf1);
        buflen = strlen (buf1);
        if (buflen == 0)
            break;

        n_write (s， buf1， buflen + 1);
        n_read (s， buf2， sizeof (buf2));
        printf("Received line: %s\n"， buf2);
    }
}
```

DOS WATTCP 示例二：使用兼容型 Socket 的服务程序

```
#include        <stdio.h>
#include        <string.h>
#include        "tcp.h"

#define PORT     7000

main ()
{
    struct sockaddr_in        client， server;
    int                s， ns， namelen， pktlen;
    char     buf[256];

    sock_init ();

    s=socket(AF_INET， SOCK_STREAM， 0);
    memset ((char *)&server， sizeof(server)， 0);
    server.sin_family = AF_INET;
    server.sin_port = htons(PORT);
    server.sin_addr.s_addr = INADDR_ANY;
```

```
    bind(s， (struct sockaddr *)&server， sizeof(server));
    listen(s，1);
    namelen = sizeof (client);
    ns = accept (s， (struct sockaddr *)&client， &namelen);
    for (;;) {
        pktlen = n_read (ns， buf， sizeof (buf));
        if (pktlen == 0)
            break;
        printf ("Received line: %s\n"， buf);
        n_write (ns， buf， pktlen);
    }
    n_close(ns);
    n_close(s);
}
```

DOS WATTCP 示例三：使用专用型 Tcp_Socket 的客户程序

```
#include <stdio.h>
#include <string.h>
#include “tcp.h”

#define PORT                    7000

tcp_Socket          Sock;
char                Buffer[ 513 ];

main ()
{
    char            *server="202.38.75.33"， buf[30];
    short           port;
    longword        host;
    tcp_Socket      *s;
    int             status， len;

    sock_init ();
    port = PORT;
    if (host = resolve( server )) {
        printf("\nResolved %s = %s \n\n"， server， w_inet_ntoa(buf， host));
    } else {
        printf("Could not resolve host '%s'\n"， server );
        exit( 3 );
    }
    s = &Sock;
    if (!tcp_open( s， 0， host， port， NULL )) {
```

```
            puts("Sorry，  unable to connect to that machine right now!");
            return;
        }
        printf("waiting...\r");
        sock_wait_established( (sock_type *)s，  sock_delay，  NULL，  &status );

        printf ("Enter a line: ");
        gets (Buffer);
        while ( Buffer[0] ) {
            sock_write ((sock_type *)s，  (byte *)Buffer，  strlen (Buffer) +1);
            sock_wait_input( (sock_type *)s，  30，  NULL，  &status );
            len = sock_fastread( (sock_type *)s，  Buffer，  512 );
            Buffer[ len ] = 0;
            printf( "Received line: %s\n"，  Buffer );
            printf ("Enter a line: ");
            gets (Buffer);
        }
        sock_close ( (sock_type *)s );

sock_err:
        switch (status) {
        case 1 : /* foreign host closed */
            break;
        case -1: /* timeout */
            printf("ERROR: %s\n"，  sockerr(s));
            break;
        case 2 : /* CLOSWT and no more data */
            sock_close( (sock_type *)s );
            /* printf("Closing case status 2\n"); */
            break;
        }
        printf("\n");
        exit( status );
}
```

五、编译环境的使用

为简化程序的编译，现以上面的 server.c、client.c 为例，简单介绍 Linux 和 DOS 环境下的 make/nmake 的使用。

Linux 系统：GNU C 2.7.2

为本次实验建立一个专门的子目录，在该目录中使用如下的 Makefile：
Filename: Makefile
Content:

```
CC = cc
all:    server client
server: server.c
        $(CC) -o server server.c
client: client.c
        $(CC) -o client client.c
```

编译命令：make<CR>。编辑源代码可选用 vi 或者 pico 编辑器，前者功能强大而方便，后者简单而易于使用。其它如 jed、joe 等也可供选用。

WindowsNT 系统

在 WindowsNT 系统下，用 Microsoft Visal C++ 5.0 作为编译环境（各人为自己的程序建立一个 sock 子目录）。操作如下：

1. 用 Msvc 打开源文件
2. 打开 Build 菜单，用 Build 命令对源文件进行编译和连接，Msvc 提示要求建立项目文件（project workspace），选择是(Y)。（这是会出现编译和连接错误）
3. 打开 project 菜单，打开 Settings 对话框，打开 Link 选项，在 Object/library modules 对话框中加入 wsock32.lib
4. 打开 Build 菜单，用 Rebuild all 命令重新对源文件进行编译和连接。

DOS 系统：Microsoft C 6.0（不作要求）

WATTCP 软件包（.H 头文件、.LIB 库文件、示例和库函数源代码）在 d:\wattcp 目录下。建议大家为本次实验建立一个专门的子目录，在该目录中使用如下的 Makefile：
Filename: Makefile
Content:

```
                        #Makefile for Waterloo TCP sample applications
CLIB=d:\wattcp\lib\wattcpl.lib      #wattcpld.lib 是包含调试信息的版本
WATTCP_INC=d:\wattcp\include #个别机器上 WATTCP 在 F:\wattcp
CFLAGS= /qc /Zi /AL -I$(WATTCP_INC)
CC= cl /F 8000 $(CFLAGS)
all:   client.exe clientd.exe server.exe
clientd.exe: clientd.c $(CLIB)
      $(CC) clientd.c $(CLIB)
client.exe: client.c $(CLIB)
      $(CC) client.c $(CLIB)
```

编译命令：nmake<CR>或者 nmk<CR>

六、参考文献

【1】 施炜 李铮 秦颍 著，Windows Sockets 规范及应用－Windows 网络编程接口（电子版），上海交通大学，1996.5

【2】 Phil Cornes 著，童寿彬等译，Linux 从入门到精通，北京：电子工业出版社，1998.7

【3】 David L.Stevens 等著，张娟 王海 译，用 TCP/IP 进行网际互连 第 1 卷，北京：电子工业出版社，1998.7

【4】 Visual C++ 5.0 联机帮助