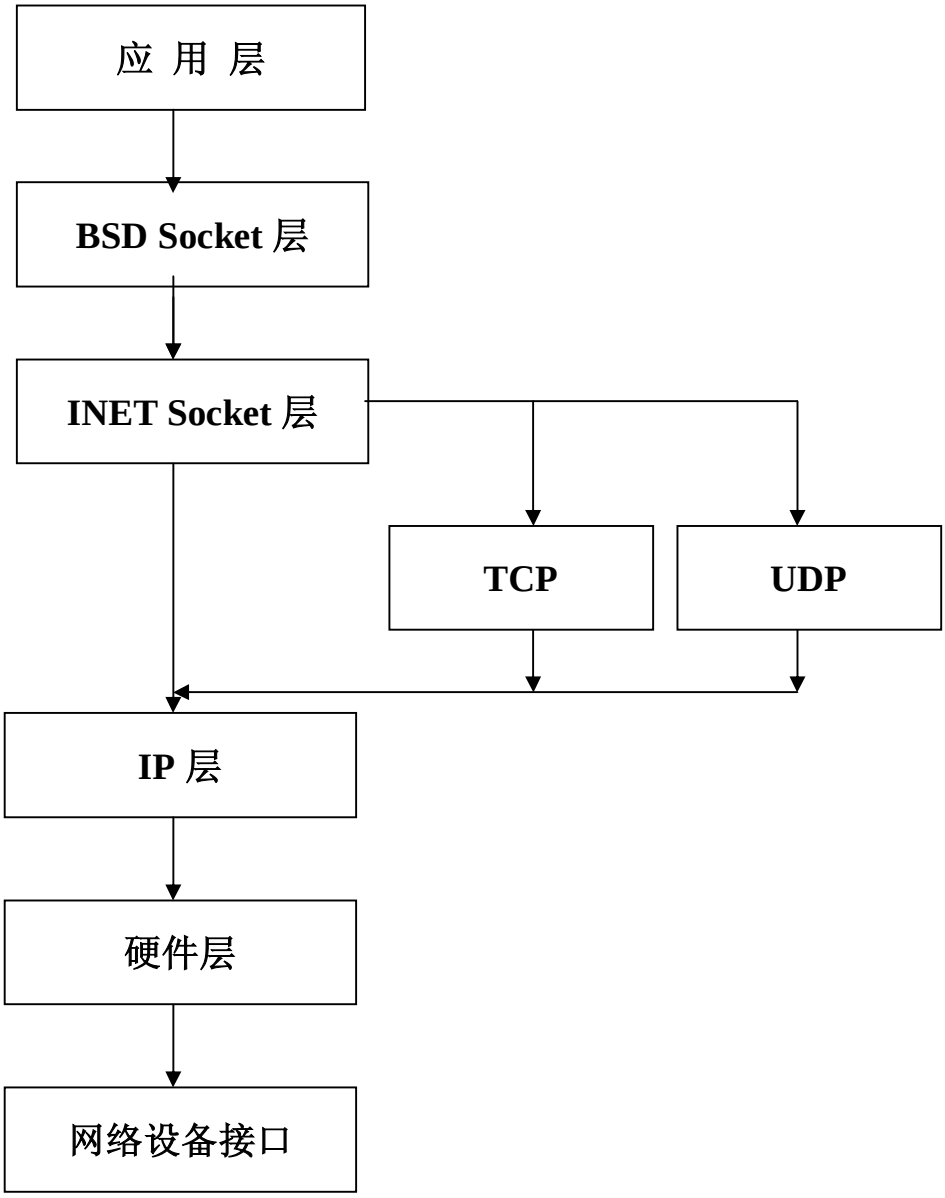
LINUX 网　络　程　序　设　计

第一部分：LINUX 网络程序设计技术（基础）

一、基于 Linux Socket(BSD Socket)的网络程序设计技术（Linux/Unix 平台）

1.1 Linux 协议栈

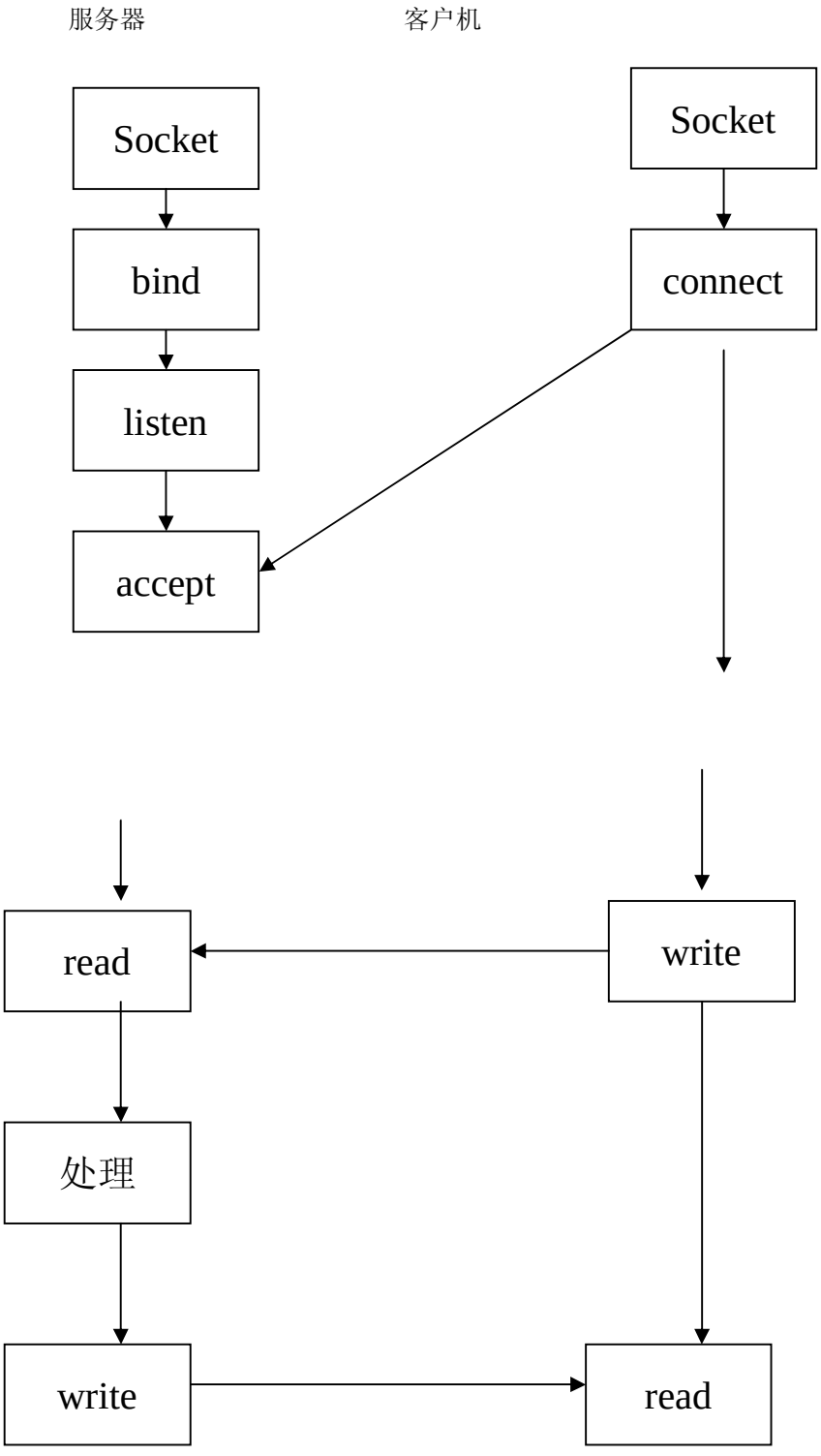


Linux 网络系统基本可以分为硬件层／数据链路层、IP 层、INET Socket 层、BSD Socket 层和应用层五个部分。其中在 LINUX 内核中包括了前四个部分，在应用层和 BSD Socket 层之间的应用程序接口以 4.4BSD 为模板。INETSocket 层实现比 IP 协议层次高，实现对 IP 分组排序、控制网络系统效率等功能。而 IP 层就是在 TCP／IP 网络协议栈中心的互联网　层实现。硬件层在

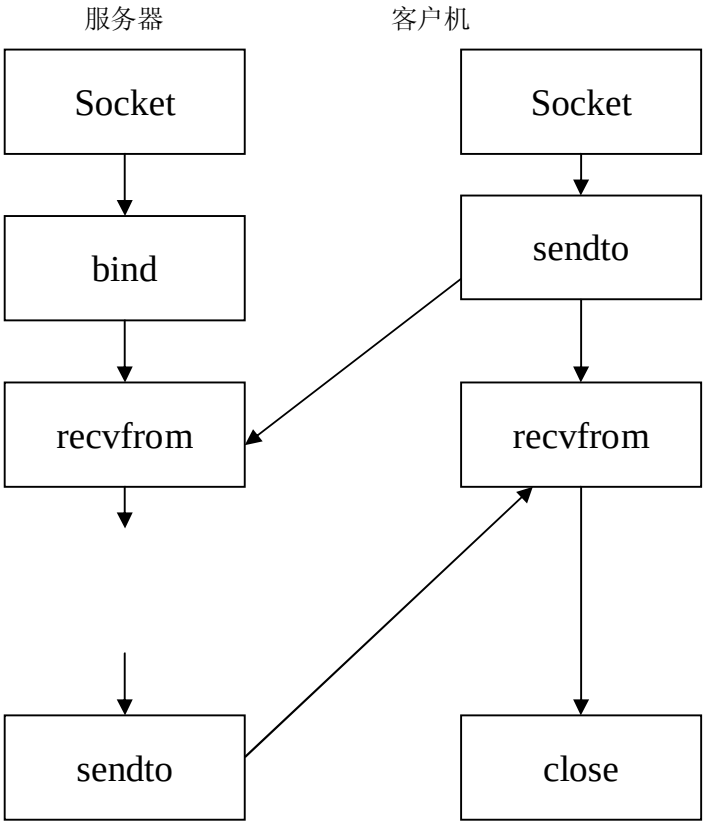
TCP/IP 协议栈本身中就和数据链路层区分不明确，暂时称这个包括了硬件驱动和硬件发送组织工作的层次为硬件层。

1.2 C/S 的通信模型

1.2.1 TCP 的 C/S 通信模型



1.2.2 UDP 的 C/S 通信模型



1.3 Socket 的基本函数

1.3.1 Internet 标准服务

著名（TCP、UDP）端口号（至 2003/9/15 为止）列表：

端口号分三类： 著名端口号；已注册端口号；动态或私用端口号。

著名端口号：0 到 1023.（通常要求有超级用户的权限）

已注册端口号：1024 到 49151

动态或私用端口号：49152 到 65535

第三类：动态或私用端口号：

从 49152 到 65535

1.3.2 基本的数据结构

1）socket 地址

头文件：sys/socket.h 和 netinet/in.h

见 linux 的/usr/include 下的内容。

注：或在 LINUX 下用：man in.h 来查看。这是在 in.h 中定义的。

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

/* Supported address families. */
```

```
#define AF_UNSPEC   0
#define AF_UNIX     1   /* Unix domain sockets          */
#define AF_LOCAL    1   /* POSIX name for AF_UNIX       */
#define AF_INET     2   /* Internet IP Protocol         */
#define AF_AX25     3   /* Amateur Radio AX.25          */
#define AF_IPX      4   /* Novell IPX                   */
#define AF_APPLETALK    5   /* AppleTalk DDP            */
#define AF_NETROM   6   /* Amateur Radio NET/ROM        */
#define AF_BRIDGE   7   /* Multiprotocol bridge         */
#define AF_ATMPVC   8   /* ATM PVCs                     */
#define AF_X25      9   /* Reserved for X.25 project    */
#define AF_INET6 10 /* IP version 6                 */
#define AF_ROSE     11  /* Amateur Radio X.25 PLP       */
#define AF_DECnet   12  /* Reserved for DECnet project  */
#define AF_NETBEUI  13  /* Reserved for 802.2LLC project*/
#define AF_SECURITY 14  /* Security callback pseudo AF */
#define AF_KEY      15      /* PF_KEY key management API */
#define AF_NETLINK  16
#define AF_ROUTE    AF_NETLINK /* Alias to emulate 4.4BSD */
#define AF_PACKET   17  /* Packet family                */
#define AF_ASH      18  /* Ash                          */
#define AF_ECONET   19  /* Acorn Econet                 */
```

```
#define AF_ATMSVC   20   /* ATM SVCs             */
#define AF_SNA      22   /* Linux SNA Project (nutters!) */
#define AF_IRDA     23   /* IRDA sockets         */
#define AF_PPPOX    24   /* PPPoX sockets        */
#define AF_MAX      32   /* For now.. */
```

这个sockaddr是SocketAPI中通用的。即尽管所有不同协议的地址（格式当然不同），但都用同一套Socket API。

2）Internet 的ＩＰ地址：netinet/in.h

```
/* Internet address. 32 位网络字节顺序的二进制形式的IP地址*/
/* Internet address.    */
typedef    uint32_t in_addr_t;
struct in_addr
   {
      in_addr_t          s_addr;
   };
```

```
/* Structure describing an Internet (IP) socket address. */
#define __SOCK_SIZE__    16
/* sizeof(struct sockaddr)    */
```

```
struct sockaddr_in {
sa_family_t       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)];
};

(2)结构sockaddr_in中的TCP或UDP端口号sin_port和IP地址sin_addr都是以网络字节顺序存储的。

(3)32 位的IP地址可以用两种不同的方法引用。例如，假设定义变量servaddr为Internet套接字地址结构，那么可以用servaddr.sin_addr或servaddr.sin_addr.s_addr来引用 这个IP地址，需要注意的是前一种引用是结构类型(struct in_addr)的数据，而后一种引用是整数类型(unsigned long)的数据。当将IP址作为函数参数使用时，需要明确使用哪种类型的数据，因为编译器对结构类型参数和整数类型参数的处理方式不一样。

(4)sin_zero成员未被使用，它是为了和通用套接字地址(struct sockaddr)保持一致而引入的。在编程时，一般将它设置为 0。通常的做法是在填充结构sockaddr__in的内容之前将整个结构变量清零。

(5)套接字地址结构仅供本机TCP协议记录套接字信息而用，这个结构变量本身是不在网络上传输的。但是它的某些内容，如IP址和端口号是在网络上传输的，这也是为什么这两部分数据需要转换成网络字节顺序的原因。

在设置结构sockaddr_in中的IP地址时，需要将字符串形式表示的IP地址转换成二进制形式。以下函数可以处理这个问题：

#include<arpa/inet.h)

注：主要是在arpa/inet.h中定义如下内容：（可用man inet.h查看）

uint32_t htonl(uint32_t);

uint16_t htons(uint16_t);

uint32_t ntohl(uint32_t);

uint16_t ntohs(uint16_t);

~~in_addr_t inet_addr(const char *);//建议用inet_aton()。~~

char *inet_ntoa(struct in_addr);

int inet_aton(const char *cp，struct in_addr *inp);//非零，转换成功（地址有效），否则为 0。

将字符串形式的IP地址转换成 32 位网络字节顺序的二进制形式的IP地址．成功时返回非 0，否则返回 0，转换后的IP地址存储在参数inp中。

char * inet_ntoa(struct in_addr in);

将 32 位二进制形式的IP地址转换为数字点形式的IP地址，结果在函数返回值中返回。

这 2 个函数将数字点形式表示的字符中IP地址与进行转换，如数字点形式表示的IP地址 192．168．0．10，对应于二进制形式表示的IP地址COA8000A。

[实例：Addr_Trans.cpp]

注：详情请查看：man inet_aton　或 man　inet_ntoa

3) 主机字节顺序与网络字节顺序

主机字节顺序：主机存储数据的顺序方式。网络中存在多种类型的机器，如基于Intel芯片的PC机和基于RISC芯片的工作站。这些不同类型的机器表示数据的字节顺序是不同的。

设有一个 16 位的整数A103 它由 2 个字节组成，高位字节是Al，低位字节是 03。在内存中可以有两种方式来存储这个整数：

little-endian方式：　低位字节存储在这个整数的开始地址位置，基于Intel芯片的机器采用的是这种方式。

big-endian方式：高位字节存储在开始地址位置，大多数基于RISC芯片的机器采用的是这种方式。

网络协议中的数据只有采用统一的字节顺序，才能在不同类型的机器之间正确地发送和接收数据。Intemet规定的网络字节顺序采用big-endian方式。例如TCP协议数据段中的 16 位端口号和 32 位IP地址就是使用网络字节顺序进行传送的。

Linux系统提供 4 个库函数来进行转换字节转换：

#include<arpa/inet.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);

注：可用 man htonl等查看。或 man inet.h查看

以上 4 个函数中h代表host，n代表network，s代表short，l代表long。Short是16 位整数，long是 32 位整数。前两个函数将主机字节顺序转换成网络字节顺序，而后两个函数则刚好相反。编程中，在需要使用网络字节顺序时，应该使用这几个函数来进行转换，绝对不要依赖于具体机器的表示方式。

4) 字节处理函数

套接字地址是多字节数据，不是以空字符结尾的，这和c语言中的字符串是不同的。

Linux提供两组函数来处理多字节数据，一组函数以b(byte)开头，是和BSD系统兼容的函数；另一组函数以mem(内存)开头，是ANSI C提供的函数。 以b开头的函数有：

#include<string.h>

（a） ~~void bzero(void *s,int n);~~

建议用memset().可用：man memset 或 man string.h 查看。

void *memset(void *s, int c, size_t n)

该函数将参数s指定的内存的前n个字节设置为字节c。通常用它来将套接字地址清零，如：

memset(&servaddr，0,sizeof(servaddr))；

注：其它常用的字节处理函数请查看：man string.h

1.3.3 基本套接字函数

本节主要讨论TCP套接字使用这些函数的情形 。

前面描述了TCP C/S间的交互过程，这是使用TCP协议进行网络通信的程序的基本模型。

注：在LINUX下进行网络编程是十分方便与强大。TCP／IP及SOCKET API在LINUX内核直接得到支持。完全不需要像WIN平台要先进行ws2_32.dll库的装载与卸载。它的错误处理方式是，当出现错误时，设置全局变量errnor的值为错误号。通过strerror(errno)函数获取错误的字符信息。Strerror()在string.h中定义。errno在errno.h中定义。

1．函数socket

函数socket创建一个套接字描述符。其定义如下：

注：man socket 查看。或：man 7 ip 或 man 7 tcp

#include<sys/types.h>

#include<sys/socket.h>

int socket(int domain， int type， int protocol)；

参数domain指定要创建的套接字的协议簇；

见前面的 Supported address families.

AF_INET用于TCP/IP。

参数type指定套接字类型：

字节流型：SOCK_STREAM

数据报型：SOCK_DGRAM

原始SOCKET：SOCK_RAW

参数protocol指定使用哪种协议。通常设置为 0，表示使用该型SOCKET的默认协议。对字节流型的SOCK_STREAM意味着是TCP，对数据报型的SOCK_DGRAM意味着是UDP，对原始SOCKET的SOCK_RAW则必须指明（如：ICMP或IGMP等）。

函数socket成功执行时，返回一个正整数，称为套接字描述符，标识这个套接字；否则，返回-1，并设置全局变量errno为相应的错误类型。

例：创建一个TCP套接字的操作一般如下：

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

if(sockfd<O){

fprintf(stderr， "socket error：%s\n"， strerror(errno))；//stdio.h

exit(1)；//stdlib.h

}

2、函数bind

函数bind将本地地址与套接字绑定在一起。其定义如下：

#include<sys/types.h>

#include<sys/socket.h>

int bind(int sockfd，struct sockaddr *myaddr，int addrlen);

注：man bind查看

参数sockfd是函数socket返回的套接字描述符；参数myaddr是本地地址；参数addrlen是套接字地址结构的长度。函数bind成功执行时，返回 0；否则，返回-1，并设置全局量errno为错误类型EADDRINUSER。

服务器和客户机都可以调用函数bind来绑定套接字地址，但一般是服务器调用函数

bind来绑定自己的公认端口号。绑定操作一般如下：

注：struct sockaddr_in myaddr;

而不是：struct sockaddr myaddr;

```
memset(&myaddr，0,sizeof(myaddr))；
myaddr.sin_family = AF_INET；
myaddr.sin_port = htons(PORT)；
myaddr.sin_addr.s_addr = htonl(INADDR_ANY)；
if( bind(sockfd，(struct sockaddr *)&myaddr，sizeof(myaddr))<0)
{
fprintf(stderr，"Bind to port %d error\n"，PORT)；
exit(1)；
}
```

绑定操作一般有如下 5 种组合方式:

程序类型	IP址	端口号	说 明
服务器	INADDR_ANY	非零值	指定服务器的公认端口号
服务器	本地IP地址	非零值	指定服务器的IP地址和公认端口号
客户机	INADDR_ANY	非零值	指定客户机的连接端口号
客户机	本地IP地址	非零值	指定客户机的IP地址和连接端口号
客户机	本地IP地址	零	指定客户机的IP地址

下面详细说明上表中列出的 5 种方式：

(1)服务器指定套接字地址的公认端口号，不指定IP地址。

服务器调用函数bind时，如果设置套接字的IP地址为特殊的INADDR_ANY，表示它愿意接收来自任何网络设备接口的客户机连接。这是服务器最经常使用的绑定方式。

(2)服务器指定套接字地址的公认端口号和IP地址。

服务器调用函数bind时，如果设置套接字的IP地址为某个本地IP地址，这表示服务器只接收来自对应于这个IP地址的特定网络设备接口的客户机连接。如果这台机器只有一个网络设备接口，这和第一种情况是没有区别的，但当这台机器有多个网络设备接口时，我们可以用这种方式来限制服务器的接收范围。

(3)客户机指定套接字地址的连接端口号。

在一般情况下，客户机不用指定自己的套接字地址的端口号，当客户机调用函数connect进行TCP连接时，系统会自动为它选择一个未用的端口号，并且用本地的IP地址来填充套接字地址中的相应项。但在有的情况下，客户机需要使用特定端口号，如Linux系统中的rlogin命令(见RFC1282)，因为rlogin命令需要使用保留端口号，而系统不会为客户机自动分配一个保留端口号，所以需要调用函数bind来和一个未用的保留端口号绑定。

(4)指定客户机的IP地址和连接端口号。

表示客户机使用指定的网络设备接口和端口号进行通信。

(5)指定客户机的IP地址。

表示客户机使用指定的网络设备接口进行通信，系统自动为客户机选择一个未用的端口号。一般只有在主机有多个网络设备接口时使用。

在编写客户机程序时，一般不要使用固定的客户机端口号，除非是在必须使用特定端口号的情况下。

3．函数listen

函数listen将一个套接字转换为倾听套接字(listening socket)。其定义如下：

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

参数sockfd指定要转换的套接字描述符；参数backlos设置请求队列的最大长度。函数listen成功执行时，返回 0；否则返回-1。

服务器需要调用函数listen将套接字转换成倾听套接字，以便接收客户机请求。

函数listen的功能有两个：

(1)函数socket创建的套接字是主动套接字，可以用它来进行主动连接(调用函数connect)，但是不能接收连接请求，而服务器的套接字必须能够接收客户机的请求。函数listen将一个尚未连接的主动套接字转换成为—个被动套接字：告诉TCP协议，这个套接字可以接收连接请求。执行函数listen之后，服务器的TCP状态由CLOSED状态转换成LISTEN状态。

(2)TCP协议将到达的连接请求排队，函数listen的第二个参数指定这个队列的最大长度。 要创建一个倾听套接字，必须首先调用函数socket创建一个主动套接字，然后调用函数bind将它与服务器套接字地址绑定在一起，最后调用函数listen进行转换。这 3 步操作是所有TCP服务器所必须的操作。

4．函数accept

函数accept从倾听套接字的完成连接队列中接收—个连接。如果完成连接队列为空，那么这个进程睡眠。其定义如下：

#include<sys/socket.h>

int accept(int sockfd，struct sockaddr *addr，int *addrlen)；

参数sockfd指定倾听套接字描述符；参数addr为指向一个Internet套接字地址结构的指针（即 struct sockaddr_in *）；参数addrlen为指向一个整型变量的指针。函数accept成功执行时，返回 3 个结果：

函数返回值为一个新的套接字描述符，标识这个接收的连接；参数addr指向的结构变量中存储客户机地址；参数addlen指向的整型变量中存储客户机地址的长度。如果对客户机的地址和长度不感兴趣，可以将参数addr和addrlen设置为NULL。函数accept执行失败时，返回-1.

函数accept从倾听套接字的完成连接队列中接收一个已经建立起来的TCP连接，因为倾听套接字是专为接收客户机连接请求，完成 3 次握手操作而用的，所以TCP协议不能使用倾听套接字描述符来标识这个连接，于是TCP协议创建一个新的套接字来标识这个要接收的连接，并将它的描述符返回给应用程序。现在有两个套接字，一个是调用函数accept时使用的倾听套接字，另一个是函数accept返回的连接套接字(connected socket)。这两个套接字的作用是完全不同的：一个服务器进程通常只需创建一个倾听套接字，在服　务器进程的整个活

动期间，用它来接收所有客户机的连接请求，在服务器进程终止前关闭这个倾听套接字；而对于每个接收的连接，TCP协议都创建一个新的连接套接字，来标识这个连接，服务器使用这个连接套接字与客户机进行通信操作，当服务器处理完这个客户机请求时，关闭这个连接套接字。

当函数accept阻塞等待已经建立的连接时，如果进程捕获到信号，那么函数将以错误返回，错误类型为EINTR。对于这种错误，一般重新调用函数accept来接收连接。

5．函数close

函数close关闭一个套接字描述符。套接字描述符的close操作与文件描述符的close操作类似。其定义如下：

```
#include<unistd.h>
int close(int sockfd);
```

参数sockfd指定要关闭的套接字描述符。函数close成功执行时，返回 0；否则，返回-1。

套接字描述符的close操作和文件描述符的close操作一样：函数close将套接字描述符的引用计数减 1，如果描述符的引用计数大于 0，则表示还有进程引用这个描述符，函数close正常返回；如果描述符的引用计数变为 0，则表示再没有进程引用这个描述符，于是启动清除套接字描述符的操作，函数close立即正常返回。清除套接字描述符的操作是：

将这个套接字描述符标记为关闭状态(不同于TCP协议状态转换图中的CLOSED状态)，然后立即返回进程。调用了函数close之后，进程将不再能够访问这个套接字，但是这不表示TCP协议删除了这个套接字。TCP协议将继续使用这个套接字，将尚未发送的数据传递到对方，然后发送FIN数据段，执行关闭操作，一直等到这个TCP连接完全关闭之后，TCP协议才删除这个套接字。

6．函数connect

函数connect与服务器建立一个连接。其定义如下：

```
# include<sys/types.h>
# include<sys/socket.h>
```

int connect(int sockfd，struct sockaddr *servaddr, int addrlen)；

参数sockfd是函数socket返回的套接字描述符；参数servaddr指定远程服务器的套接字地址，包括服务器的IP地址和端口号；参数addrlen指定这个套接字地址的长度。函数connect成功执行时，返回O；否则返回-1，并设置全局变量errno为以下任何一种错误类型：

ETIMEOUT、ECONNREFUSED、EHOSTUNREACH或ENETUNREACH。

在调用函数connect之前，客户机需要指定服务器进程的套接字地址。建立一个TCP连接的操作一般如下：

```
struct sockaddr_in    servaddr;
memset(&servaddr，0，sizeof(servaddr))；
servaddr.sin_family = AF_INET；
Servaddr.sin_port = htons(SERVER_PORT)；
if(inet_aton("192．168．0．1",&servaddr.sin_addr)<0) )
{
fprintf(stderr，"inet_aton error\n")；
exit(1)；
}
if(connect(sockfd，(struct sockaddr *)&servaddr,sizeof(servaddr)) <0)
{
fprintf(stderr，"connect error：%s"，strerror(errno))；
exit(1)；
}
```

7．函数send和recv

函数send和recv从套接字发送和接受数据。其定义如下：

注：man send　或　man recv　查看

ssize_t send(int s, const void *buf, size_t len, int flags);

功能：在已建立连接的套接字上发送数据。

参数：

1）s:[输入] 已建立连接的套接字，将在这个套接字上发送数据。

2）b u f：[输入] 字符缓冲区，其中包含即将发送的数据。

3）l e n ：[输入]指定即将发送的缓冲区内的字符数。

4）f l a g s：[输入] 可为 0 、M S G _ D O N T R O U T E 或M S G _ O O B 。另外，f l a g s 还可以是对那些标志进行按位“或运算”的一个结果。M S G _ D O N T R O U T E 标志要求传送层不要将它发出的包路由出去。由基层的传送决定是否实现这一请求（例如，若传送协议不支持该选项，这一请求就会被忽略）。M S G _ O O B 标志预示数据应该被带外发送。

返回值：返回实际发送的字节数；若发生错误，就返回-1。

2．接受数据：recv（）

对在已连接套接字上接受数据来说，r e c v 函数是最基本的方式。它的定义如下：

```
ssize_t recv(int s, void *buf, size_t len, int flags);
```

功能：在已连接套接字上接受数据。

参数：

1）s：[输入]准备接收数据的那个套接字。

2）b u f ：[输出]即将收到数据的字符缓冲区。

3）l e n：[输入] 准备接收的字节数或b u f 缓冲的长度。

4）f l a g s ：[输入]可以是下面的值：0 、M S G _ P E E K 或M S G _ O O B 。另外，还可对这些标志中的每一个进行按位和运算。当然，0 表示无特殊行为。M S G _ P E E K 会使有用的数据复制到所提供的接收端缓冲内，但是没有从系统缓冲中将它删除。

返回值：返回已接受的字节数。若socket已关闭，则返回 0，若出错返回：-1。

1.3.4 格式化数据的网络传输

网络可以传递任意格式的数据，但是可能需要采取一些特殊的处理方法。

1．文本字符串

文本字符串的发送和接收不需特殊的处理。网络传输是以字节为单位进行的，一个字符对应—个字节数据。

2．二进制数据

二进制数据的发送和接收不需特殊的处理。二进制数据以字节为单位发送、接收。

能够按顺序正确地接收这些数据。

3．整数

整数的发送和接收需要进行一些处理。前面已经提到，网络中不同类型的主机可能采用不同的字节顺序存储整数，所以发送整数前必须将它们转换为网络字节顺序，而接收方必须将它们再从网络字节顺序转换为主机字节顺序。对于16位的短整数，使用函数htons和ntohs进行转换；对于32的长整数，使用函数htonl和ntohl进行转换；对于64位的整数，需要使用其他方法。

下面这两个函数分别实现发送和接收一个32位长整数的操作：

```
void sender(int sockfd，long data)
{
    long nd；
    nd   = htonl(data)；
    send(sockfd，&nd，sizeof(nd),0)；
}
void receiver(int sockfd，lonq *data)
{
    long      nd；
    recv(sockfd，&nd，sizeof(nd),0)；
    *data = ntohl(nd)；
}      .
```

第二种发送和接收整数的方法：使用字符串发送和接收整数。这种方法可以处理任意类型的整数，包括64位整数。下面这两个函数分别实现发送和接收整数：

```
void sender(int sockfd，long data)
{
    char      buf[5]；
```

```
    sprintf(buf，"%ld"，data)；
    send(sockfd，buf，strlen(buf),0)；
}
void receiver(int sockfd，long *data)
{
    char    bur[5];
    recv(sockfd，buf，sizeof(buf),0)；
    sscanf(buf，"%ld"，data)；
}
```

4．结构数据

结构数据的发送和接收必须进行转换。不同类型的主机一般采取不同的顺序存储结构数据中的各个成员变量，所以必须处理这种不同。一般的处理方法是：发送方依次传送结构中各个成员变量，而接收方使用相同的顺序接收各个成员变量。

一个发送和接收结构数据的例子：

```
    struct multi_type{
    char    sd_str[10]；
    int     sd_int；
    }；
    void sender(int sockfd，struct multi_type data)
    {
    int  len；
    len =  strlen(data.sd_str)；
    len = htonl(len)；
    send(sockfd，&len，sizeof(len),0)；
   send(socked，&data.sd_str，len,0)；
    len = htonl(data.sd_int)；
    send(sockfd，&len，sizeof(len),0);
    }
```

```
void receiver(int sockfd，struct multi_type *data)
{
int       len；
recv(sockfd，&len，sizeof(len),0)；
len = ntohl(len)；          ·
recv(sockfd.data->sd_str，len,0)；
recv(sockfd，&len，sizeof(len),0)；
data->sd_int = ntohl(len)；
}
```

函数sender发送一个结构数据：这个结构的第一个成员变量是一个字符串，因为字符串的长度不固定，所以在发送字符串的内容之前，必须将字符串的长度通知对方，函数将这个长度转换成网络字节顺序后发送。第二个成员变量是一个整数，函数将这个整数转换成网络字节顺序后发送。

函数receiver接收一个结构数据：函数首先从套接字中读取一个整数，这个整数指定结构multi-type的第一个成员变量的长度，函数然后根据这个长度，从套接字读取字符串。因为结构multi-type的第二个成员变量是一个整数，长度固定，函数直接从套接字中读取这个整数。

5．浮点数

浮点数的发送和接收也必须进行处理。一种简单的处理方法是将这个浮点敷以字符串形式发送和接收。

6．指针数据

直接发送和接收指针数据是没有意义的。只能传送和接收指针所指的具体内容。在网络上传送格式化数据是比较复杂的。为了解决这个问题，Sun公司设计了一种外部数据表示，它规定了在网络上传送数据时如何表示成公共形式的数据。Sun公司的外部数据表示XDR(externalDataRepresentation)已经成为大多数客户机—服务器应用的一个事实上的标准。

1）XDR库函数完成数据项的转换

2）XDR 标准（RFC1014）(略)

［综合实例］设计一个TCP（单用户、单线程）程序，当客户机发送一个字符串时，返回该字符串中字母个数。当客户方发送”quit”命令时，比方结束运行。

[LINUX__TCPServer.cpp]

[LINUX__TCPClient.cpp]

1. LINUX多线程网络程序设计

注：在使用LINUX多线程时，创建项目后，在project->properties->c/c++builder->GCC C Linker->Libraries->Libraries(-l)下填入：pthread后，单击Apply后，单击OK。

1、LINUX线程

一个LINUX线程，即是如下格式的一个函数：

```
void * MyThread(void *)
{//线程代码}
```

其中：void *为传递给线程的参数。

如：定义一个线程，打印 1，2，3，。。。，10。每隔 1 秒打印一个数。

```
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>

 void *MyThread(void *arg) {
   int i;
   for ( i=1; i<=10; i++) {
      printf("线程ID %d :打印：%d \n",(int)pthread_self(),i);
      sleep(1);
   }
   return NULL;
}
```

2、创建一个LINUX线程

在头文件pthread.h中，定义了如下的方法，用于创建一个LINUX线程：

```
int pthread_create(pthread_t * ID,const pthread_attr_t * attr,
```

void * (* thread_addr)(void *), void * arg);

功能：创建一个LINUX线程。

参数：

1)ID［输出］返回线程的ID。这是一个pthread_t类型。被定义为：unsigned long int。

2)attr[输入]线程的属性。若为NULL，则是默认的属性。

3）线程的方法。

4）arg[输入]传给线程的参数。

返回值：若成功则返回 0，否则返回非零。

如：pthread_t mythread;

```
if ( pthread_create( &mythread, NULL, thread_function, NULL) ) {
    printf("pthread_create出错！.\n");
    abort();
}
```

3、等待一个线程的终止

使用pthread_join()等待一个线程的终止。实现简单的线程同步。定义如下：

int pthread_join(pthread_t thread, void **value_ptr);

功能：调用pthread_join()的线程（调用者线程）进入睡眠状态，等待参数thread指定的线程运行结束后才继续运行。

参数：

thread:[输入]被等待的线程。

value_ptr:[输入]线程运行结束时的出口号。传递给pthread_exit()方法。

返回值：若成功则返回 0，否则返回非零值（即出错号）

4、使线程睡眠

定义如下 ： #include <unistd.h>

unsigned int sleep(unsigned int seconds);

功能：使线程睡眠。

参数：seconds[输入]睡眠秒数。

返回值：时间到，线程唤醒 时返回 0。若时间不到，线程由信号打断则返回剩余的秒数。

5、终止一个线程

定义如下：

```
void pthread_exit(void *value_ptr);
```

线程自身主动停止运行。value_ptr是出口码。

6、LINUX线程互斥锁

1)定义LINUX线程互斥锁

互斥锁类型：pthread_mutex_t 。在pthread.h中定义。见：man pthread.h

定义一个互斥锁如下：

```
pthread_mutex_t      mymutex;
```

2）初始化一个互斥锁

在使用互斥之前一定要初始化。

```
int    pthread_mutex_init(
         pthread_mutex_t *restrict mutex,
         const pthread_mutexattr_t *restrict attr);
```

对互斥锁进行初始化。进行必要的资源分配。最后一定要用：
pthread_mutex_destroy(mymutex)进行资源的释放。
参数：mutex:[输入]互斥锁
attr:[输入]锁属性。若为NULL，则使用默认属性。
返回值：成功返回 0，否则返回非 0（错误号）

3）加锁

在进入临界区之前先要加锁。若成功才能进入到临界区中。否则线程在该锁等待队列中睡眠，直到锁被释放为止。
定义如下：

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

参数：
mutex:[输入]锁
返回值：成功返回 0，否则返回非 0 的错误号。

4）解锁
当线程从临界区操作完成，应尽快解锁，给其它线程执行的机会。
定义如下：

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

参数：

mutex:[输入]锁
返回值：成功返回 0，否则返回非 0 的错误号。

5）释放锁资源
当一个锁不再使用了，则应释放锁所占用的资源。一旦锁资源被释放，则锁不能再使用了，除非再用pthread_mutex_init()再次初始化分配资源为止。
定义如下：
int pthread_mutex_destroy(pthread_mutex_t *mutex);
mutex:[输入]锁
返回值：成功返回 0，否则返回非 0 的错误号。

互斥锁使用方式：

a)对每一个临界区资源，程序员人为地关联一个互斥锁。
即：定义一个互斥锁，并进行初始化。

b)在该临界区中，提供对临界区进行操作的程序代码（任何要访问临界区的线程，通过调用这些函数进行数据操作），代码结构如下：
pthread_mutex_lock(&mymutex);
临界区操作代码
pthread_mutex_unlock(&mymutex);

c)在释放临界区资源的代码中，进行锁的资源释放。

7、LINUX线程的同步

pthread_mutex_t互斥锁只能进行LINUX线程的互斥。但更多是时候同时会发生线程的同步。如何进行线程的同步？
1)定义同步对象
pthread_cond_t　　mycond;

2)初始化同步对象
分配同步对象所需要的资源。在使用之前一定要初始化。定义如下：
int pthread_cond_init(pthread_cond_t *restrict cond,
const pthread_condattr_t *restrict attr);
参数：cond:[输入]同步对象
attr:[输入]属性。若为NULL，则为默认属性。
返回值：成功 0 否则返回错误号。
注：若不再使用cond对象则一定要用pthread_cond_destroy(&mycond)释放资源。

3)等待一个资源条件
等待一个条件的发送。此时该线程处理睡眠状态。定义如下：

int pthread_cond_wait(pthread_cond_t *restrict cond,
pthread_mutex_t *restrict mutex);
参数：cond:[输入]同步对象。
Mutex:[输入]睡眠之前要先释放的锁（见：后边的使用说明）
注：将一直睡眠下去，直到有另外一个线程调用：pthread_cond_broadcast() 或 pthread_cond_signal() 为止。

int pthread_cond_timedwait(pthread_cond_t *restrict cond,
pthread_mutex_t *restrict mutex,
const struct timespec *restrict abstime);

含义同上。但时间上有限制。若abstime所表示的时间一到，就返回不再睡眠下去。

4)条件具备唤醒等待该条件的线程
int pthread_cond_broadcast(pthread_cond_t *cond);
功能：唤醒所有等待该条件对象的线程。

int pthread_cond_signal(pthread_cond_t *cond);
功能：唤醒所有等待该条件对象线程中的某一个线程。建议使用第一种。

5）释放条件对象的资源
当一个条件对象不再使用时要释放它的资源。定义如下：
int pthread_cond_destroy(pthread_cond_t *cond);
参数：cond:[输入]要释放资源的条件对象。

使用说明：

线程的同步设计是十分重要的。设计线程的同步代码时，基本的设计原则是：

1）pthread_cond_wait(...) 与 pthread_cond_broadcast(...)必须放在互斥锁内。这一点十分重要！

2）pthread_cond_wait(...)必须放在循环内。常用代码结构如下：

```
while(条件不满足)
{
        pthread_cond_wait(...)
}
//此时条件已满足
```

3）pthread_cond_wait(...) 与 pthread_cond_broadcast(...)必须配套使用。一个线程使用pthread_cond_wait(...)进入睡眠，另一个线程（协作者）在做完操作，使得条

件发生变化时，一定要使用pthread_cond_broadcast(...)来唤醒等待该条件的线程，否则很容易发生死锁。

4）典型 的代码结构如下：

定义互斥锁与条件对象：

```
pthread_mutex_t    my_lock;
pthread_mutex_init(&my_lock,NULL);//在该临界区上有互斥
pthread_cond_t    my_cond1;//代表条件 1 .如：栈不满。若满就wait;
pthread_cond_init(&my_cond1,NULL);//在该临界区上同时有同步
pthread_cond_t    my_cond2;//代表条件 2.如：栈不空。若空就wait;
pthread_cond_init(&my_cond2,NULL);//在该临界区上同时有同步
```

线程 1:（生产者线程）

访问临界区的代码结构是：

```
pthread_mutex_lock (&count_lock);//要进入临界区了

while(条件不满足时) //如：top>=MAX（含义：栈满了，放不下了）
{
    pthread_cond_wait( &my_cond1, &my_lock);//睡眠，让消费者取走后唤醒
}
//此时栈不满，即条件已满足
//对临界区进行操作的代码：放入元素

pthread_cond_broadcast(&my_cond2);//很重要。配套使用。唤醒消费者线程。
pthread_mutex_unlock (&count_lock);//最后离开临界区
```

线程 2:（消费者线程）

访问临界区的代码结构是：

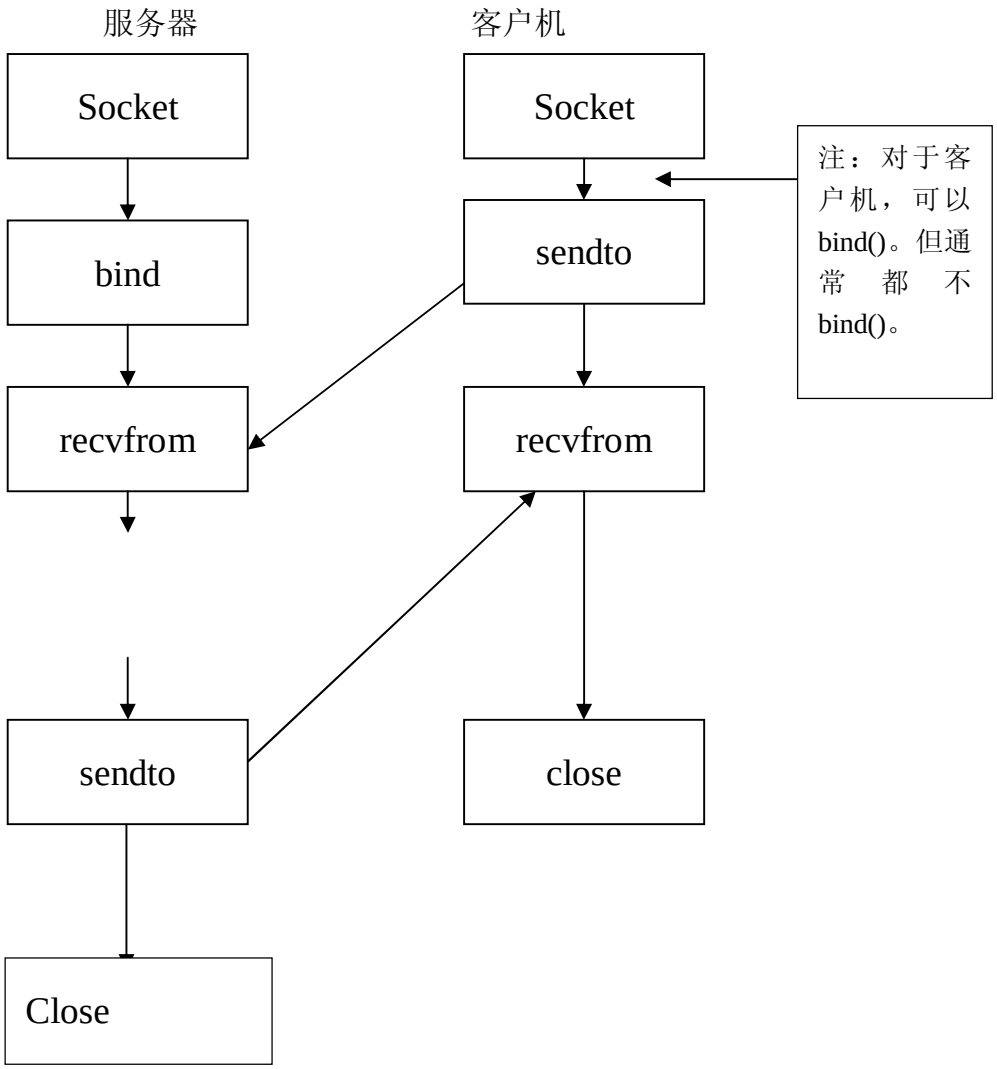
```
pthread_mutex_lock (&count_lock);//要进入临界区了

while(条件不满足时) //如：top＜0（含义：栈空了，取不了元素）
{
    pthread_cond_wait( &my_cond2, &my_lock);//睡眠，让生产者放入后唤醒
}
//此时栈不空，即条件已满足
//对临界区进行操作的代码：取走元素

pthread_cond_broadcast(&my_cond1);//很重要。配套使用。唤醒生产者线程。
pthread_mutex_unlock (&count_lock);//最后离开临界区
```

8、［LINUX下多线程TCP服务器示例:LINUX_MultiThreadTCPServer.cpp］

2.7.5 UDP 的 C/S 通信模型



1) UDPSocket的建立：

注：man 7 udp查看

创建UDPSocket，方式如下：

```
int    udps = socket(AF_INET,SOCK_DGRAM,0);
If( udps == -1)
{//出错处理！
```

}

2）UDP数据报的发送：sendto()

Sendto()函数用于发送一个UDP包。其定义的格式如下：

```
ssize_t sendto(
int s,
const void *buf,
size_t len,
int flags,
const      struct sockaddr *to,
socklen_t tolen
)
```

功能：用于发送一个UDP包。

参数：　1) s:[输入]要发送UDP包的socket。

2) buf:[输入]要发送的数据的缓冲区。

3) len:[输入]要发送的数据的字节数。

4) flags:[输入] 可为0 、MSG_DONTROUTE 或MSG_OOB 。另外，flags 还可以是对那些标志进行按位“或运算”的一个结果。MSG_DONTROUTE 标志要求传送层不要将它发出的包路由出去。由基层的传送决定是否实现这一请求（例如，若传送协议不支持该选项，这一请求就会被忽略）。MSG_OOB 标志预示数据应该被带外发送。

5)to:[输入]接受方的IP地址与端口号。

6) tolen:[输入]参数to所指的数据区的大小。

返回值：返回实际发送的字节数；若发生错误，就返回-1.

3）接受一个UDP包：recvfrom（）

接受一个UDP包，使用recvfrom() 函数是最基本的方式。它的定义如下：

ssize_t recvfrom(int s, void *buf, size_t len, int flags, struct sockaddr *from, socklen_t *fromlen);

功能：接受一个UDP包。

参数：

1) s：[输入]准备接收UDP包的那个套接字。

2) b u f ：[输出]即将收到数据的字符缓冲区。

3) l e n：[输入] 准备接收的字节数或b u f 缓冲的长度。

4) f l a g s ：[输入]可以是下面的值：0 、M S G _ P E E K 或M S G _ O O B 。另外，还可对这些标志中的每一个进行按位和运算。当然，0 表示无特殊行为。M S G _ P E E K 会使有用的数据复制到所提供的接收端缓冲内，但是没有从系统缓冲中将它删除。

5)from:[输出]发送方的地址与端口号。

6) fromlen:[输入]from所指数据区的大小。

返回值：返回已接受的字节数。若socket已关闭，则返回 0，若出错返回：-1。

UDP实例：

设计一个基于UDP的多客户机的网络程序，完成：客户端向服务器发送字符串，服务器向客户端返回其中包含的英文字母的个数。当客户端发送“quit”命令时，客户机结束运行。

（A） UDP 服务器方程序基本设计(多客户机工作模型)

根据上述所讨论的BSD SOCKET 的基本函数及UDP下C/S的工作模型，基本的UDP服务器(多客户机)程序的设计步骤如下：

S1、调用socket()创建一个UDP式的socket。
S2、调用bind()绑定服务器的IP和PORT。
while(UDP服务器继续运行)
{
S3、调用recvfrom()按应用协议进行UDP包的读取。
S4、调用sendto()将处理结果应答给客户机。
}
S5、调用close()关闭相应的UDP socket。

服务器方：[LINUX__UDPServer.cpp]

（B） UDP 客户机方程序基本设计

根据上述所讨论的BSD SOCKET的基本函数及UDP下C/S的工作模型，基本的UDP客户机程序的设计步骤如下：
S1、调用socket()创建一个socket。
S2、调用sendto()/recvfrom()按应用协议进行UDP网络通信。
S3、调用close()关闭相应的socket。
客户方：[LINUX__UDPClient.cpp]

（C）连接型 UDP

只用于UDP客户机。通常不用于UDP服务器。原因是要求内核进行UDP包的过滤。
基本思路：（UDP客户机：先用connect()向UDP服务器进行虚拟的“连接”，告之内核史对该UDP服务器进行UDP包的读取与发送。之后可直接使用recv()或send()进行UDP包的处理。）
[LINUX__connectUDPClient.cpp]

2.7.6 SOCKET 选项(第一部分)

组播与广播需要设置SOCKET选项。使用Socket选项函数，用于查看或设置Socket的参数和状态。分三种：应用层选项（SOL_SOCKET）、TCP选项（IPPROTO_TCP）、IP选项（IPPROTO_IP）。

```
#include <sys/types.h>
#include <sys/socket.h>

    int   getsockopt(int   s, int level, int optname, void *optval, socklen_t
    *optlen);

    int setsockopt(int s, int   level,   int   optname,   const   void   *optval,
    socklen_t optlen);
```

描述见：man setsockopt 或 man getsockopt

其中参数：
s：[输入] Socket
level[输入]: 哪一层的选项
optname[输入]: 该层中的选项名字
optval[输入/输出]:选项值的首地址。

optlen[输入/输出]: 选项值的缓冲区的长度
返回值：
成功： 0　不成功：-1

程序实例：查看常用的SOCKET选项的值。
[查看SOCKET选项的实例：LINUX_GetSockopt.cpp]

选项的详细讨论见后面。此处首先用于讨论广播与组播。

2.7.7 广播与组播

广播和组播是网络通信的重要手段。IP协议提供了点到点的数据传输通道，但这并不意味着在IP协议下的通信只能是一对一的。通过广播和组播，可以实现多对多的通信。广播和组播在实际中都应用得相当广泛。

1） 广播的基本原理

TCP／IP协议的一个显著优点就是在IP地址中定义了广播地址。 一个IP地址由(网络号，主初：号)组成。所有主机号部分为全 1 的IP地址都是广播地址。如A类地址 110.255.255.255、B类地址 166．111．255. 255、C类地址 202.112．58．255 等.这样的IP地址被称为定向广播地址。然而一般来说，主机号 部分还要划分出若干位用于描述子网号。例如 166．111．1．X。在子网中也有一个IP地址用于广播，同样这个地址的主机号部分为全 1 子网 166．111．1 这个子网中，广播地址为 166.111.1.255。通过这种广播地址而进行广播被称为子网定向广播(Subnet-directed broadcast)。划分了子网号部分的IP地址由(网络号，子网号，主机号)组成。由于一个网络号下通常有多个子网，因此定向广播地址也被称为全子网广播地址(All-subnets-directed broadcast address)-通过子网广播地址或是全子网广播地址发布广播信息，统称为定向广播。定向广播方式允许将一个目的IP地址为另一网络广播地址的分组发送出去。
一般来说，路由器发现一个分组的目的地址为广播地址，不会将其转发，但是可以通过改变路由器的配置来允许发送广播分组。

定向广播需要知道广播的目的网络。有限广播(Limited broadcast)则是一种与IP地址无关的广播方式。即 255 255．255．255。这种广播只限于在本地的MAC地址进行的。首先当发送方主机发现发送的分组目的地址为本子网广播地址，就将MAC地址设为全 1，即FF：FF：FF：FF：FF：FF。带有这样MAC地址的帧在经过任何该子网上的主机时，都将被数据链路层接收。在去掉帧信息后传到IP层，IP层处理过IP头信息后，如果本机有应用程序处理广播数据报，则发送到相应端口，否则抛弃该分组。需要注意的是，发送广播方也同样会收到它所发送的广播。

IP层在发送时，可能将较大的数据报分成若干分组发送。但是在广播方式中是不允许数据报拆分的，这是为了减少广播所占用的网络带宽。对于以太网来说，最大帧长不能超过 1518 字节。如果广播数据报所形成的帧超过这个限制，将不予发送。

2）广播程序实例

一个应用程序要发送广播包，需要用到**socket**选项：**SO_BROADCAST**。若该选项没有打开，则内核将拒绝进行广播包的发送。打开该选项的程序代码如下：

```
int on=1;
setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on));
```

程序实例：每 **3** 秒发送一次广播，将本机时间通知本子网内所有主机。
［**LINUX__TimeBroadcast.cpp**］
接受广播包的程序：即普通的**UDP**包接受程序。
［**LINUX__TimeRecvBroadcast.cpp**］

3）组播的基本原理

组播是通过D类IP地址进行的。D类地址的前 4 位为 1110，后面 28 位为组播的组标识。因此，D类地址的范围是 224．0．0．0 到 239．255．255．255。由于IP地址是整个因特网通用的，因此组播地址也有一些“周知”地址，这些地址只允许特定的应用。　下面列出一些“周知”的组播地址。

地址	用途
224．0．0．0	保留，不分配给任何组
224．0．0．1	本子网上所有主机
224．0．0．2	本子网上所有网关
224．0．1．1	NTP(网络时间协议)组

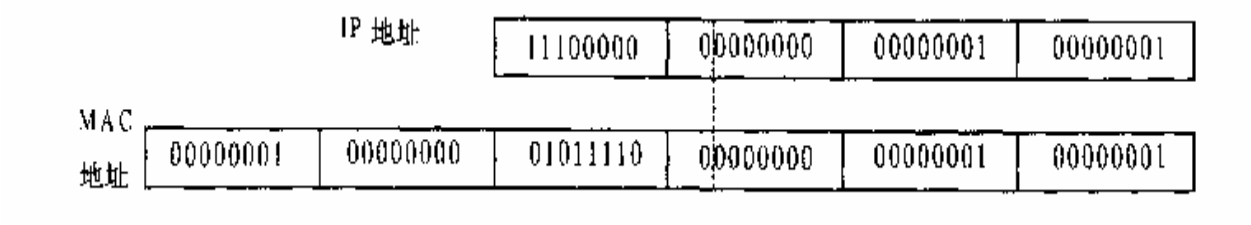
如果系统支持组播，那么ping 224．0．0．1 将会收到所有主机的回应，效果和ping有限广播地址(255．255．255．255)和子网广播地址一样。

组播地址一般由因特网的权威机构分配，未被分配的地址可以用于建立临时的组。这些组在需要时建立，在成员数为 0 时撤消。

作为与广播的对比，仍然以以太网为例介绍组播在子网中的发送和接收。当一个组播分组到达一个以太网时，形成帧后它的MAC地址为 01：00：5e：XX：XX：XX，其后 23 位由组播组标识的后 23 位映射而成。对于目的地址为 224．0．1．1 的组播分组，在以太网上帧的 MAC地址就为 01：00：5e：00：01：01，如下图所示。

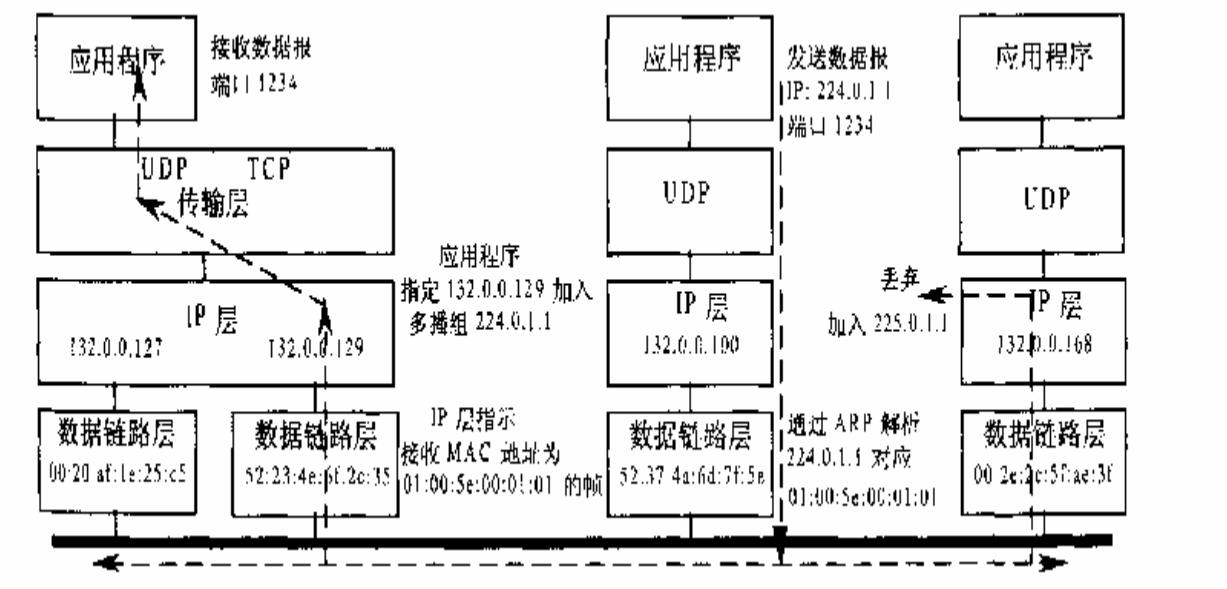
如果一台主机加入了 224．0．1．1 的组，该主机的数据链路层就会接收所有目的MAC 地址为 01：00：5e：00：01：01 的帧。但是，组播的IP地址到MAC地址的映射并不是一对一的。

由于组播IP地址中组标识有 28 位，而映射到MAC地址的只有 23 位，因此有 32 个组将映射成相同的MAC地址，例如 224．0．1．1，225．0．1．1，239．128．1．1 等都将映射到MAC地 址 01：00：5e：00：01：01。因此，要由IP层来检验到达的组播分组是否是自己所加入的组播组，　如果不是，则抛弃该分组。



组播IP地址到MAC地址的映射

下图显示了组播数据报在子网中的传输过程。左边的多归宿主机指定了它的一个接口加入组播组 224．0．1，1，而另一接口没有指定，因此组播帧只被一个接口接收。这与广播方式是不同的。广播方式中所有的接口都要收到广播帧。最右边的主机加入了组播组 225．0．1．1，因此它的链路层也接收了组播组 224．0．1．1 的帧，但IP层发现不是本组的组播分组后将之丢弃。



那么组播是如何发送到广域网的?这需要路由器对组播的支持。当子网中有一台主机加入了某一个组播组，它将通知本子网中的路由器。以后只要路由器接到目的地址为该组播地址的分组，就将复制该分组。复制的分组将在该子网中发送，原来的分组则继续转发到其他的路由器。如果某路由器所在的子网没有主机加入该组播组，那么这个路由器仅仅转发该分组，而不会复制。组播分组中的TTL域限制了它被转发的次数。每转发一次，TTL域减 1，当到达某路由器时分组的TTL域为 0，则被抛弃。

例：

从组播的原理可以看出，要接收组播的数据报需要经过如下步骤：

- 加入一个组播组。
- 在指定的端口上进行侦听。
- 接收并处理到达的组播数据报。
- 离开该组播组。

- 关闭socket。

注：向组中发送UDP包，没有任何特别要求。只要是普通的UDP包发送动作即可。
组中的成员当然可以向组发送UDP包。
以下IPV4 选项用于组播
IPv4 选项 数据类型 描述

IP_ADD_MEMBERSHIP struct ip_mreq 加入到组播组中
IP_DROP_MEMBERSHIP struct ip_mreq 从组播组中退出
IP_MULTICAST_IF struct ip_mreq 指定接受组播报文的接口
IP_MULTICAST_TTL u_char 指定提交组播报文的TTL
IP_MULTICAST_LOOP u_char 使组播报文环路有效或无效
定义的ip_mreq结构：

```
struct ip_mreq {
struct in_addr imr_multiaddr; /* IP multicast address of group */
struct in_addr imr_interface; /* local IP address of interface */
};
```

IP_ADD_MEMBERSHIP
若进程要加入到一个组播组中，用soket的setsockopt()函数设置该选项。该选项类型是ip_mreq结构，它的第一个字段imr_multiaddr指定了组播组的地址，第二个字段imr_interface指定了接口的IPv4 地址。

IP_DROP_MEMBERSHIP
该选项用来从某个组播组中退出。数据结构ip_mreq的使用方法与上面相同。
IP_MULTICAST_IF
该选项可以修改网络接口，在结构ip_mreq中定义新的接口。
IP_MULTICAST_TTL
设置组播报文的数据包的TTL（生存时间）。默认值是 1，表示数据包只能在本地的子网中传送。
IP_MULTICAST_LOOP
组播组中的成员自己也会收到它向本组发送的报文。这个选项用于选择是否激活这种状态。

组播方式程序实例.以组播方式来接受每 3 秒发送的时间。发送方向组中发即可。