Socket 编程

参考

UNP 第一卷

chinaunix 论坛

一、 基本知识

主机字节序和网络字节序

主机字节序即内存中存储字节的方法有：

1. Little endian：将低序字节存储在起始地址
2. Big endian：将高序字节存储在起始地址

网络字序表示网络协议在处理多字节时的顺序，一律为 big endian

主机字节序和网络字节序转换的函数：

```
#include <netinet/in.h>
uint16_t htons(uint16_t <16 位的主机字节序>)
uint32_t htonsl(uint32_t <32 位的主机字节序>) //转换为网络字节序
uint16_t ntohs(uint16_t <16 位的网络字节序>)
uint32_t ntohl(uint32_t <32 位的网络字节序>) //转换为主机字节序
```

缓冲区

每个 TCP SOCKET 有一个发送缓冲区和一个接收缓冲区，TCP 具有流量控制，所以接收缓冲区的大小就是通知另一端的窗口的大小，对方不会发大于该窗口大小的数据；而 UDP SOCKET 只有一个接收缓冲区无流量控制，当接收的数据报溢出时就会被丢弃

通信域（地址族）

套接字存在于特定的通信域(即地址族)中，只有隶属于同一地址族的套接字才能建立对话。Linux 支持 AF_INET(IPv4 协议)、AF_INET6(IPv6 协议)和 AF_LOCAL(Unix 域协议)。

套接口(socket)＝网络地址+端口号。,要建立一个套接口必须调用 socket 函数,套接口有三种类型，即字节流套接口(SOCK_STREAM),数据报套接口(SOCK_DGRAM)和原始套接口(SOCK_RAW)。定义一个连接的一个端点的两元组，即 IP 地址和端口号，称为一个套接口。在网络连接中，两个端点所组成的四元组（即本地 IP、本地 PORT、远程 IP 和远程 PORT）称为 socket pair，该四元组唯一的标识了一个网络连接。该情况可通过 netstat 验证。

二、 socket 地址结构

1. IPv4 的 Socket 地址结构（定长）

```
Struct in_addr{
    In_addr_t s_addr; // 32 位 IP 地址，网络字节序
}
Struct sockaddr_in{
    Uint8_t sin_len; //IPv4 为固定的 16 字节长度
    Sa_family_t sin_family;  //地址簇类型，为 AF_INET
    In_port_t sin_port; //16 位端口号，网络字节序
```

```
    Struct in_addr sin_addr; // 32 位 IP 地址
    Char sin_zero[8]; //未用
}
```

2. IPv6 的 socket 地址结构（定长）

```
struct in6_addr{
    uint8_t s6_addr[16]; //128 位 IP 地址，网络字节序
}
struct sockaddr_in6{
    uint8_t sin6_len; //IPv6 为固定的 24 字节长度
    sa_family_t sin6_family; //地址簇类型，为 AF_INET6
    in_port_t sin6_port; //16 位端口号，网络字节序
    uint32_t sin6_flowinfo; //32 位流标签
    struct in6_addr sin6_addr; //128 位 IP 地址
}
```

3. UNIX 域 socket 地址结构（变长）

Struct sockaddr_un，地址簇类型为 AF_LOCAL

4. 数据链路 socket 地址结构（变长）

struct sockaddr_dl，地址簇类型为 AF_LINK

5. 通用的 socket 地址结构

```
struct sockaddr{
    uint8_t sa_len;
    sa_family_t sa_family;
    char sa_data[14];
}
```

三、 C/S 网络编程

初始化 sock 连接符：

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

函数返回 socket 描述符，返回-1 表示出错

domain 参数只能取 AF_INET, protocol 参数一般取 0

应用示例：

TCP 方式：sockfd = socket(AF_INET,SOCK_STREAM,0);

UDP 方式：sockfd =socket(AF_INET, SOCK_DGRAM,0);

绑定端口：

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

函数返回-1 表示出错，最常见的错误是该端口已经被其他程序绑定。

需要注意的一点：在 Linux 系统中，1024 以下的端口只有拥有 root 权限的程序才能绑定。

连接网络（用于 TCP 方式）：

int connect(int sockfd, struct sockaddr *servaddr, int addrlen);
函数返回-1 表示出错，可能是连接超时或无法访问。返回 0 表示连接成功，可以通过 sockfd 传输数据了。

监听端口（用于 TCP 方式）：
int listen(int sockfd, int queue_length);
需要在此前调用 bind()函数将 sockfd 绑定到一个端口上，否则由系统指定一个随机的端口。
接收队列：一个新的 Client 的连接请求先被放在接收队列中，直到 Server 程序调用 accept 函数接受连接请求。
第二个参数 queue_length，指的就是接收队列的长度 也就是在 Server 程序调用 accept 函数之前最大允许的连接请求数，多余的连接请求将被拒绝。

响应连接请求（用于 TCP 方式）：
int accept(int sockfd,struct sockaddr *addr,int *addrlen);
accept()函数将响应连接请求，建立连接并产生一个新的 socket 描述符来描述该连接，该连接用来与特定的 Client 交换信息。
函数返回新的连接的 socket 描述符，错误返回-1

addr 将在函数调用后被填入连接对方的地址信息，如对方的 IP、端口等。
addrlen 作为参数表示 addr 内存区的大小，在函数返回后将被填入返回的 addr 结构的大小。
accept 缺省是阻塞函数，阻塞直到有连接请求
应用示例：
struct sockaddr_in their_addr; /* 用于存储连接对方的地址信息*/
int sin_size = sizeof(struct sockaddr_in);
… …（依次调用 socket(), bind(), listen()等函数）
new_fd = accept(sockfd, &their_addr, &sin_size);
printf(”对方地址: %s\n″, inet_ntoa(their_addr.sin_addr));
… …

关闭 socket 连接：
int close(int sockfd);
关闭连接将中断对该 socket 的读写操作。
关闭用于 listen()的 socket 描述符将禁止其他 Client 的连接请求。

部分关闭 socket 连接：
int shutdown(int sockfd, int how);
Shutdown()函数可以单方面的中断连接，即禁止某个方向的信息传递。
参数 how：
0 - 禁止接收信息
1 - 禁止发送信息
2 - 接收和发送都被禁止，与 close()函数效果相同

socket 轮询选择：
int select(int numfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct

timeval *timeout);
应用于多路同步 I/O 模式（将在同步工作模式中详细讲解）
FD_ZERO(*set) 清空 socket 集合
FD_SET(s, *set) 将 s 加入 socket 集合
FD_CLR(s, *set) 从 socket 集合去掉 s
FD_ISSET(s, *set) 判断 s 是否在 socket 集合中

常数 FD_SETSIZE：集合元素的最多个数
等待选择机制：
int poll(struct pollfd *ufds, unsigned int nfds, int timeout);
是 select 机制的一个变种，应用于多路同步 I/O 模式（将在同步工作模式中详细讲解）
ufds 是 pollfd 结构的数组，数组元素个数为 nfds。
struct pollfd {
int fd; /* 文件描述字 */
short events; /* 请求事件集合 */
short revents; /* 返回时间集合 */
};

接收/发送消息：
TCP 方式：
int send(int s, const void *buf, int len, int flags);
int recv(int s, void *buf, int len, int flags);
函数返回实际发送/接收的字节数，返回-1 表示出错，需要关闭此连接。
函数缺省是阻塞函数，直到发送/接收完毕或出错
注意：如果 send 函数返回值与参数 len 不相等，则剩余的未发送信息需要再次发送

UDP 方式：
int sendto(int s, const void *buf, int len, int flags, const struct sockaddr *to, int tolen);
int recvfrom(int s,void *buf, int len, int flags, struct sockaddr *from, int *fromlen);
与 TCP 方式的区别：
需要指定发送/接收数据的对方（第五个参数 to/from）
函数返回实际发送/接收的字节数，返回-1 表示出错。
函数缺省是阻塞函数，直到发送/接收完毕或出错
注意：如果 send 函数返回值与参数 len 不相等，则剩余的未发送信息需要再次发送

注意，网络字节流的读写不同于文件的读写，由于 socket 缓冲的因素，可能读写的字节数小于所指定的字节数。所以可以使用如下函数：

```
ssize_t readn(int fd, void * buf,size_t n)
{
    ssize_t nleft;
    ssize_t nread;
    char *ptr;
    ptr = buf;
```

```
    nleft = n;
    while (nleft > 0)
    {
        if ((nread = read(fd, ptr, nleft)) < 0){
            if (errno == EINTR)
                nread = 0;
            else
                return (-1);
        }
        else if (nread == 0)  //EOF
            break;
        nleft -= nread;
        ptr += nread;
    }
    return (n - nleft);
}

ssize_t written(int fd,const void *buf,size_t n)
{
    ssize_t nleft;
    ssize_t nwrite;
    const char *ptr;
    ptr = buf;
    nleft = n;
    while (nleft > 0) {
        if (nwrite = write(fd, ptr, left) <= 0) {
            if (errno == EINTR) {
                nwrite = 0;
            else
                return (-1);
        }
        nleft -= nwrite;
        ptr += nwrite;
    }
    return (n);
}
```

基于消息的方式：

```
int sendmsg(int s, const struct msghdr *msg, int flags);
int recvmsg(int s, struct msghdr *msg, int flags);
```

标志位：

上面这六个发送/接收函数均有一个参数 flags，用来指明数据发送/接收的标志，常用的标

志主要有：
MSG_PEEK 对数据接收函数有效，表示读出网络数据后不清除已读的数据
MSG_WAITALL 对数据接收函数有效，表示一直执行直到 buf 读满、socket 出错或者程序收到信号。
MSG_DONTWAIT 对数据发送函数有效，表示不阻塞等待数据发送完后返回，而是直接返回。（只对非阻塞 socket 有效）
MSG_NOSIGNAL 对发送接收函数有效，表示在对方关闭连接后出错但不发送 SIGPIPE 信号给程序。
MSG_OOB 对发送接收都有效，表示读/写带外数据(out-of-band data)

IP 地址字符串和网络字节序的二进制 IP 地址相互转换的函数：
#inlcude <arpa/inet.h>
int inet_aton(const char *<IP 地址字符串>, struct in_addr *<32 位的网络字节序形式的 IP 地址>)
成功—1
失败—0

[通用地址函数]int inet_pton(int <地址簇类型>，可以是 AF_INET/AF_INET6>,const char *<IP 地址字符串>,void *<32 位的网络字节序形式的 IP 地址>)
成功—1
格式错误—0
失败—0

in_addr_t inet_addr(const char *<IP 地址字符串>)
返回 32 位网络字节序的 IP 地址，失败—INADDR_NONE

char *inet_ntoa(struct in_addr <32 位的网络字节序形式的 IP 地址>) 返回 IP 地址字符串

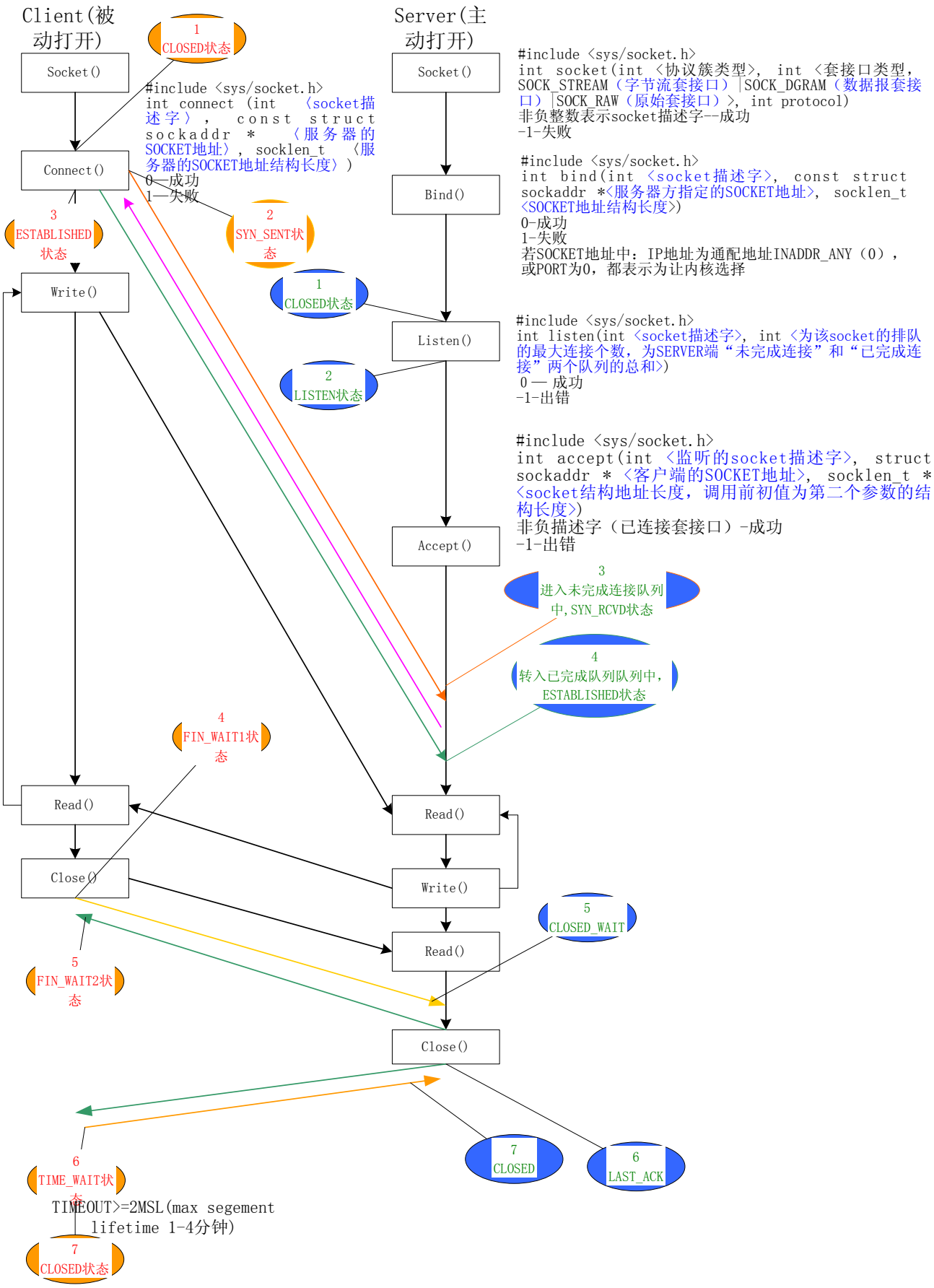
const char * inet_ntop(int <地址簇类型>, const void *<32 位的网络字节序形式的 IP 地址>, char *<IP 地址字符串>, size_t <IP 地址字符串的最大长度>)
返回指向结果<IP 地址字符串>的指针

字节顺序转换
htons()--"Host to Network Short"
htonl()--"Host to Network Long"
ntohs()--"Network to Host Short"
ntohl()--"Network to Host Long"

连接过程是通过一系列状态表示的，这些状态有：LISTEN，SYN-SENT，SYN-RECEIVED，ESTABLISHED，FIN-WAIT-1，FIN-WAIT-2，CLOSE-WAIT，CLOSING，LAST-ACK，TIME-WAIT 和 CLOSED。CLOSED 表示没有连接，各个状态的意义如下：
LISTEN - 侦听来自远方 TCP 端口的连接请求；
SYN-SENT - 在发送连接请求后等待匹配的连接请求；

SYN-RECEIVED - 在收到和发送一个连接请求后等待对连接请求的确认；
ESTABLISHED - 代表一个打开的连接，数据可以传送给用户；
FIN-WAIT-1 - 等待远程 TCP 的连接中断请求，或先前的连接中断请求的确认；
FIN-WAIT-2 - 从远程 TCP 等待连接中断请求；
CLOSE-WAIT - 等待从本地用户发来的连接中断请求；
CLOSING - 等待远程 TCP 对连接中断的确认；
LAST-ACK - 等待原来发向远程 TCP 的连接中断请求的确认；
TIME-WAIT - 等待足够的时间以确保远程 TCP 接收到连接中断请求的确认；
CLOSED - 没有任何连接状态；

Tcp 面向连接 如下图所示：


Client(被动打开)
Server(主动打开)
1 CLOSED状态
Socket()
#include <sys/socket.h>
int connect (int 〈socket描述字〉, const struct sockaddr * 〈服务器的SOCKET地址〉, socklen_t 〈服务器的SOCKET地址结构长度〉)
0—成功
1—失败
Connect()
3 ESTABLISHED状态
Write()
2 SYN_SENT状态
1 CLOSED状态
2 LISTEN状态
Socket()
#include <sys/socket.h>
int socket(int 〈协议簇类型〉, int 〈套接口类型, SOCK_STREAM（字节流套接口）|SOCK_DGRAM（数据报套接口）|SOCK_RAW（原始套接口）>, int protocol)
非负整数表示socket描述字--成功
-1-失败
Bind()
#include <sys/socket.h>
int bind(int 〈socket描述字〉, const struct sockaddr *〈服务器方指定的SOCKET地址〉, socklen_t 〈SOCKET地址结构长度〉)
0-成功
1-失败
若SOCKET地址中：IP地址为通配地址INADDR_ANY（0），或PORT为0，都表示为让内核选择
Listen()
#include <sys/socket.h>
int listen(int 〈socket描述字〉, int 〈为该socket的排队的最大连接个数，为SERVER端“未完成连接”和“已完成连接”两个队列的总和〉)
0 — 成功
-1-出错
Accept()
#include <sys/socket.h>
int accept(int 〈监听的socket描述字〉, struct sockaddr * 〈客户端的SOCKET地址〉, socklen_t * 〈socket结构地址长度，调用前初值为第二个参数的结构长度〉)
非负描述字（已连接套接口）-成功
-1-出错
3 进入未完成连接队列中,SYN_RCVD状态
4 转入已完成队列队列中，ESTABLISHED状态
4 FIN_WAIT1状态
Read()
Close()
5 FIN_WAIT2状态
Read()
Write()
Read()
5 CLOSED_WAIT
Close()
6 TIME_WAIT状态
TIMEOUT>=2MSL(max segement lifetime 1-4分钟)
7 CLOSED状态
7 CLOSED
6 LAST_ACK

TIME_WAIT 状态

一个 tcp 协议的 socket 编程例子：

[Server]

```
/* 主函数 */
int listenfd, connfd;
struct sockaddr_in servaddr, cliaddr;
struct hostent *hp;
struct servent *sp;
struct in_addr **pptr;

if ((hp = gethostbyname(argv[1])) == NULL)
    //error
if ((sp = getservbyname(argv[2], “tcp”)) == NULL)
    //error

pptr = (struct in_addr **)hp->h_addr_list;
listenfd = socket(AF_INET, SOCK_STREAM, 0);
bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family = AF_INET;
//servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
memcpy(&servaddr.sin_addr, *pptr, sizeof(struct in_addr));
servaddr.sin_port = htons(portnum);
bind(listenfd, (struct sockaddr *)&servaddr, sizeof(servaddr));
listen(listenfd, listenqueuenum);
signal(SIGCHLD,sig_chld);
for (;;)
{
    len = sizeof(cliaddr);
    if ((connfd = accept(listenfd, (struct sockaddr *)&cliaddr, &len)) < 0) {
        if (errorno==EINTR)
            continue;
        else
            perror(“accept error”);
    }
    if (fork()==0) //子进程，复制父进程的所有描述字，所以 listenfd 和 connfd 被父子进程所共享，描述字的访问记数值都累计为 2
    {
        close(listenfd);
        printf(“connection from %s, port %d\n”, inet_ntop(AF_INET, &cliaddr,.sin_addr, buf, sizeof(buf)), ntohs(cliaddr.sin_port));
        snprintf(buf, ……);
        write(connfd, buf, strlen(buf));
            … //process
```

```
        close(connfd); //处理结束后关闭 socket 连接
        exit(0); //子进程退出
    }
    close(connfd); //父进程用不到 connfd，可以将其关闭，由于描述字的访问记数值由
2 减 1，不会触发 FIN 分节，（和 shundown 不同）只有为 0 时，才真正关闭连接
}
```

自选端口应该大于 1023（不要是保留端口），小于 49152（临时端口）

```
/* SIDCHLD 信号处理函数 */
vod sig_chld(int signo)
{
    pid_t pid;
    int stat;

    while ((pid = waitpid(-1, &stat,  WNOHANG)) > 0);
    return;
}
```

[client]

```
sockfd=socket(AF_INET,SOCK_STREAM,0);
bzero(&servaddr, sizeof(servaddr));
bzero(&servaddr,sizeof(servaddr));
servaddr.sin_familf=AF_INET;
servaddr.sin_port=htons(SERV_PORT);
inet_pton(AF_INET, SERV_IP_STRING, &servaddr.sin_addr);
connect(sockfd, (struct sockaddr *)&serv_addr, sizeof(servaddr);
….//process
exit(0);
```

终止网络连接的正常方法是调用 close()，但是有两个限制：

1. 将描述字的访问计数-1，到 0 时才激发 TCP 连接的终止序列（丢弃接收缓冲区里的数据，发送发送缓冲区里的数据，如果描述字的访问计数为 0，在数据之后发出 FIN）
2. 终止了数据传输的两个方向，读和写

shutdown 可以关闭一半连接，而且在发完发送缓冲区里的数据后立即发出 FIN 序列

UDP 无连接 编程时，在 SERVER 端少了 accept()和 listen()，客户端少了 connect()，因为是无连接的，所以不用 close()，读写一般用 sendto()和 recvfrom()。如果 client 发送的数据被路由丢弃或者服务器的应答信息丢失，那么 client 将一直阻塞，直到应用设置的超时到达，UDP 需要应用来做接受确认、传输超时或者重传的机制。

一般的，UDP 服务属于迭代的，而 TCP 服务大多数是并发的。

用 I/O 复用方式（select()）实现(可以避免为每个客户端连接创建一个进程的开销)：

[server]

```
int client[FD_SETSIZE];
```

```
fd_set allset, rset;
…
maxfd = listenfd;
maxi = -1;
for (I = 0; I < FD_SETSIZE; i++)
    client[i] =  -1;
FD_ZERO(&allset);
FD_SET(listenfd, &allset);
For (;;){
  rset = allset;
  nready = select(maxfd+1, &rset, NULL, NULL, NULL);
  if (FD_ISSET(listenfd, &rset)) { //新连接
        clilen = sizeof(cliaddr);
        connfd = accept(listenfd, (struct sockaddr *)&cliaddr, &clilen);
        for (i = 0; I < FD_SETSIZE; i++) {
            if (client[i] < 0) { //找出 client 数组中第一个-1 的单元存放已经连接
的 socket
                client[i] = connfd;
                break;
        }
    if (I == FD_SETSIZE)
        fprintf(stderr, “too many clients\n”);
    FD_SET(connfd, &allset);
    If (connfd > maxfd)
        Maxfd = connfd;
    If (I > maxi)
        Maxi = I;
    If (--nready <= 0)
        Continue;
    }
    for (I = 0; I <= maxi; i++) {
        if ((sockfd = client[i]) < 0)
            continue;
        if (FD_ISSET(sockfd, &rset)) {
            if ((n = read(sockfd, …)) == 0){ //客户端已经关闭连接
                close(sockfd);
                FD_CLR(sockfd, &allset);
                Client[i] = -1;
            } else
            {
             … //process
                if (--nready <=0 )
                    break;
            }
```

```
        }
    }
}
```

用这种方法有一个问题：

容易受 DDOS 攻击。如一个客户端发一个字节，就睡眠，该程序将阻塞在 read()上，无法再为其他合法的客户端服务。解决方法：

1. 非阻塞 I/O
2. 为每个连接创建进程或线程
3. 对 I/O 操作设置操时

四、 常用函数

函数中均使用通用地址结构，所以一般要将特定协议的地址转换成通用地址结构。

1. 从进程到内核
   bind
   connect
   sendto
   sendmsg
2. 从内核到进程
   Accept
   Recvfrom
   Recvmsg
   Getpeername
   Getsockname
3. .解析器函数：
   ethostbyname
   gethostbyaddr
   getservbyname 端口号是以网络字节序返回的
   getservbyport 参数 port 必须是网络字节序，所以要调用 htons(port num)
4. I/O 流和描述字
   fdopen()：将描述字（int）转换为 I/O 流（FILE *）
   fileno()：将 I/O 流转换为描述字

注意：虽然一个 I/O 流也是全双工的，但是读和写之间必须要有一些如 fflush(),fseek(),fsetpos(),rewind()函数。最好的办法是对一个描述字创建两个流，读和写分开。

```
FILE *fin, *fout;
Fin = fdopen(sockfd, “r);
Fout = fdopen(sockfd, “w”);
Fgets(…);
Fputs(…);
```

但是这种标准 I/O 库的编程必须注意当前的缓冲方式：

完全缓冲：只有缓冲区满、fflush()或进程 exit()时才输出 I/O

行缓冲

不缓冲

五、 僵尸进程

Zombie<defunct>进程：一个子进程终止，系统内核会发出 SIGCHLD 信号，如果程序中没有用 singal、sigaction 进行 SIG_IGN 处理，也没有用 wait、waitpid 进行等待，结束的子进程就会变为僵尸进程。

僵尸进程产生的目的是保存子进程的信息，以便父进程在以后某个时刻需要取回
如果一个进程终止，但子进程有 Zombie 状态，这时他们的父进程将由 pid=1 的 init 进程接管，用 wait 来等待负责收回僵尸进程的资源。
僵尸进程占用内核空间，

六、 I/O 模式

5 个 I/O 模式：
1. 阻塞 I/O
2. 非阻塞 I/O

设置该标志代码：

```
int flags;
if ((flags = fcntl(fd, F_GETFL, 0)) < 0)
        //出错处理
flags |= O_NONBLOCK;
if ((fcntl(fd, F_SETFL, flags)) < 0)
        //出错处理
```

关闭该标志代码：

```
…
flags &= ~O_NONBLOCK;
…
```

非阻塞 connect 的典型应用：

```
…//设置 sockfd 为非阻塞，如上述代码
if ((n = connect(…)) < 0)
    if (errno != EINPROCESS) //正常情况下，connect 连接有一定时间，应该返回 EINPROCESS
        return(-1);
if (n == 0) //特殊情况，如 CLIENT 和 SERVER 在同台主机上，连接快速返回
    goto done;
FD_ZERO(&rset);
FD_SET(sockfd &rset);
West = rset;
Struct timeval val;
Val.tv_sec = …;
Val.tv_usec = 0;
If ((n = select(sockfd+1, &rset, &west, NULL, &val)) == 0) { //超时
    Errno = ETIMEOUT;
    Close(sockfd);
```

```
    Return(-1);
}
int error;
if (FD_ISSET(sockfd, &rset) || FD_ISSET(sockfd, &west)) {
    len = sizeof(error);
    if (getsockopt(sockfd, SOL_SOCKET, SO_ERROR, &error, &len) < 0) //检查 SOCKET
上待处理的错误
        return(-1);
}else
    //出错处理，表示在这段时间内没连接上 SERVER
done:
    fcntl(sockfd, FD_SETFL, flags); //恢复为原来的模式
    if (error){
        close(sockfd);
        errno = error;
        return(-1);
}
return(0);
```

3. I/O 复用
4. 信号驱动 I/O（SIGIO）
5. 异步 I/O

前 4 个 I/O 模式都属于同步 I/O，因为它们都阻塞于 I/O 的操作（即 I/O 数据准备好时，从内核空间往用户空间拷贝数据）

七、 守护进程

在后台运行并且与终端脱离联系的进程，只要系统不停止就一直运行着。

有几种方式启动：

1. /etc/rcx.d 下的启动脚本
2. Inetd 启动
3. Cron 或者 at 启动

守护进程的输出信息一般通过 SYSLOG 输出

函数示例：

```
#include <syslog.h>
#define MAXFD 64

void daemon_init()
{
    int I;
    pid_t pid;
    if ((pid = fork()) != 0)
        exit(0);
    setsid():
    signal(SIGHUP, SIG_IGN):
```

```
    if ((Pid = fork()) != 0)
        exit(0):
    chdir(“/”);
    umask(0);
    for (i=0; i<MAXFD; i++)
        close(i);
    openlog(logname, LOG_PID, facility);
}
```

八、 I/O 超时

方法 1：alarm()

例如：

```
static void connect_alarm(int signo)
{
    return;
}

signal(SIGALARM, connect_alarm);
if (alarm(timeout_sec)!=0)
    //error
if (connect(sockfd, (struct sockaddr *)servaddr, servlen) < 0)
{
    close(sockfd);
    if (errno = EINTR)
        errno = ETIMEDOUT;
}
alarm(0);
```

方法 2：select()

例如：

```
int readable_timeout(int fd, int sec)
{
    fd_set rset;
    struct timeval tv;
    FD_ZERO(&rset);
    FD_SET(fd, &rset):
    Tv.tv_sec = sec;
    Tv.tv_usec = 0;
    Return(select(fd+1, &rset, NULL, NULL, &tv));
}

…
sendto(sockfd, …);
if (readable_timeout(sockfd, 5)==0)
```

```
    //error
else
    recvfrom(sockfd,…); //可以读
```

方法 3：SO_RCVTIMEO 和 SO_SNDTIMEO 套接口选项

例如：

```
struct timeval tv;
tv.tv_sec=5;
tv.tv_usec=0;
setsockopt(sockfd, SOL_SOCKET, SO_RCVTIMEO, &tv, sizeof(tv)):
while (fgets(sendline…)!=NULL){
    sendto(sockfd, …);
    n=recvfrom(sockfd, …);
    if (n<0)
        if (errno == EWOULEBLOCK) {
            fprintf(stderr, “socket,timeout”);
            continue;
        }
        else
            //error;
…
```

九、 辅助数据的应用

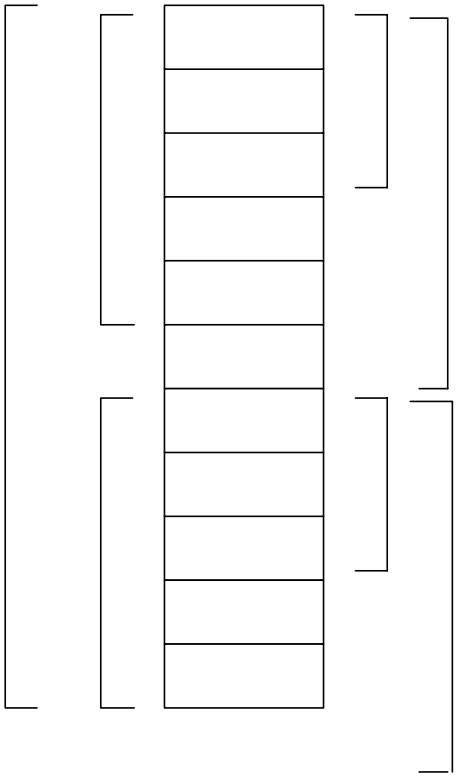
可以运用于任何进程之间传递描述字的应用

sendmsg 和 recvmsg 可以用来传送辅助数据（控制信息）。辅助数据由若干个辅助对象组成，每个辅助对象由一个 cmsghdr 结构开头。

```
    struct msghdr {
            void    *msg_name;              //用于 UDP 协议,发送方可以存放目的地址，接收方可以存放发送地址
            size_t  msg_namelen;            //同上
            struct   iovec *msg_iov;          //输入输出缓冲区数据，两个成员 iov_base 和 iov_len
            int     msg_iovlen;
            void    *msg_control;           // 辅助数据
            size_t  msg_controllen;         //辅助数据大小
            int     msg_flags;              //只对 recvmsg 有用
    };
    Struct cmsghdr{
        Socklen_t cmsg_len;
        Int cmsg_level; //IPV4 是 IPPROTO_IP，UNIX 域是 SOL_SOCKET
        Int Cmsg_type; //在 IPV4 中，IP_RECVDSTADDR 接受 UDP 数据报的目的地址
                  //IP_RECVIF 接受 UDP 数据报的接口索引
                  //UNIX 域中是 SCM_RIGHTS 传送描述字
                  //SCM_CREDS 传送用户凭证
```

}
//char cmsg_data[];

msghdr.msg_control 指向的辅助数据必须按 cmsghdr 结构对齐。



利用 UNIX 域 SOCKET 方式：

```
my_open(const char *pathname, int mode)
{
int sockfd[2];
…

socketpair(AF_LOCAL, SOCK_STREAM, 0, sockfd); //创建了一个流管道
if ((childpid = fork()) == 0) //子进程
    close(socdfd[0]);
    snprintf(argsockfd, sizeof(argsocdfd), “%d”, sockfd[1]);
    snprintf(argmode, sizeof(mode), “%d”, mode);
    execl(“./openfile”, “openfile”, argsockfd, pathname, argmode, (char
*)NULL);
    perror(“exec error\n”);
}
close(sockfd[1]);
```

Cmsg_len

msg_con

Cm

Cm

Cm

P

```
waitpid(chidpid, &status, 0);
if (WIFEXITED(status) == 0)
    perror(“chid process did not terminate\n”);
if ((status = WEXITSTATUS(status)) == 0) //把终止状态转换为退出状态 0-255
    read_fd(sockfd[0], &c, 1, &fd);
else{
    errno = status;
    fd = -1;
}
close(sockdf[0]);
return(fd);
}

ssize_t read_fd(int fd, void *ptr, ssize_t nbytes, int *recvfd)
{
    struct msghdr msg; // 辅助数据对象
    struct iovec iov[1];
    union{
        struct cmsghdr unit; //这里定义辅助数据对象的目的是为了让 msg_control 缓冲区和辅助数据单元对齐,所以不是简单地定义一个 char control[…],而是定义一个 union
        char control[CMSG_SPACE(sizeof(int))]; //辅助数据缓冲区
    }control_buf;
    struct cmsghdr *unitptr;
    int n;

    msg.msg_control = control_buf.control; //辅助数据缓冲区
    msg.msg_controllen = sizeof(control_buf.control); //辅助数据大小
    msg.msg_name = NULL;
    msg.msg_namelen = 0;
    iov[0].iov_base = ptr;
    iov[0].iov_len = nbytes;
    msg.msg_iov = iov;
    msg.msg_iovlen = 1;

    if ((n = recvmsg(fd, &msg, 0) <= 0)
        return(n);

    if ((unitptr = CMSG_FIRSTHDR(&msg) != NULL && unitptr->cmsg_len == CMSG_LEN(sizeof(int))) { //cmsg_len 应该=CMSG_LEN 宏得出的结果
    if (unitptr->cmsg_level != SOL_SOCKET) //UNIX DOMAIN 用 SOL-SOCKET
        //error
    if (unitptr->cmsg_type != SCM_RIGHTS)
        //error
    *recvd = *((int *)CMSG_DATA(unitptr)); //获得该辅助对象的数据
```

```
}
else
    *recvd = -1;
    return(n);

}
```

子进程写程序摘录：

```
msg.msg_control = control_buf.control;
msg.msg_controllen = sizeof(control_buf.control);

unitprt = CMSG_FIRSTHDR(&msg);
unitptr->cmsg_len = CMSG_LEN(sizeof(int));
unitptr->cmsg_level = SOL_SOCKET;
unitptr->cmsg_level = SCM_RIGHTS;
*((int *)CMSG_DATA(unitptr)) = sendfd; //CMSG_DATA()返回与 cmsghdr 相关联的数据的第一个字节的指针
msg.msg_name = NULL;
msg.msg_namelen = 0;
iov[0].iov_base = ptr;
iov[0].iov_len = nbytes;
msg.msg_iov = iov;
msg.msg_iovlen = 1;
sendmsg(fd, &msg, 0);
```

用户凭证

```
#include <sys/ucred.h>
struct fcred{
    uid_t fc_ruid; //真实 UID
    gid_t fc_rgid; //真实 GID
    char fc_login[MAXLOGNAME]; //LOGIN 名字
    uid_t fc_uid; //有效 UID
    short fc_ngroups; //组数
    gid_t fc_groups[NGROUPS];
}
#define fc_gid fc_groups[0]; //有效 GID
```

需要将 LOCAL _CREDS 这个 SOCKET 选项设置为 1，在 TCP 中是 CLIENT 在连接后第一次发送数据时由内核一起发送的，UDP 是每次发送数据都产生

十、 网络参数的设置和获取

获得主机名存到 hostname 中

```
int gethostname(char *hostname, size_t size);
```

取得本地的 SOCKET 信息

```
int getsockname(int sockfd, struct sockaddr *addr, int *addrlen);
```

取得对方主机的 SOCKET 信息

```
int getpeername(int sockfd, struct sockaddr *addr, int *addrlen);
```

获得 DNS 信息：

```
struct hostent *gethostbyname(const char *name);
struct hostent *gethostbyaddr(const char *addr, int len, int type);
```

返回了一个指向 struct hostent 的指针，struct hostent 定义如下：

```
struct hostent {
char *h_name; /* 主机域名 */
char **h_aliases; /* 别名*/
int h_addrtype; /* 地址类型 */
int h_length; /* IP 地址长度 */
char **h_addr_list; /* IP 地址链*/
};
#define h_addr h_addr_list[0]
```

gethostbyname 通过设置 h_errno 代表出错号，对应的错误函数有 hstrerror()和 herror()，分别对应于 strerror()和 perror()这两个普通的错误函数。

获得或改变 socket 属性

```
int getsockopt(int sockfd, int level, int name, char *value, int *optlen);
int setsockopt(int sockfd, int level, int name, char *value, int *optlen);
```

level:(级别)： 指定选项代码的类型。
SOL_SOCKET: 基本套接口
IPPROTO_IP: IPv4 套接口
IPPROTO_IPV6: IPv6 套接口
IPPROTO_TCP: TCP 套接口
level 一般为常数 SOL_SOCKET
name 选项名称
value 选项值:是一个指向变量的指针,变量可以是整形，套接口结构， 其他结构类型:linger{}, timeval{ }
optlen optval 的大小

常用选项的有：
[SOL_SOCKET]
SO_BROADCAST 允许发送广播数据 int
适用於 UDP socket。其意义是允许 UDP socket 「广播」（broadcast）讯息到网路上。

SO_DEBUG 允许调试 int

SO_DONTROUTE 不查找路由 int

SO_ERROR 获得套接字错误 int

SO_KEEPALIVE 保持连接 int
检测对方主机是否崩溃，避免（服务器）永远阻塞于 TCP 连接的输入。 设置该选项后，如果 2 小时内在此套接口的任一方向都没有数据交换，TCP 就自动给对方 发一个保持存活探测分节(keepalive probe)。这是一个对方必须响应的 TCP 分节.它会导致以下三种情况： 对方接收一切正常：以期望的 ACK 响应。2 小时后，TCP 将发出另一个探测分节。 对方已崩溃且已重新启动：以 RST 响应。套接口的待处理错误被置为 ECONNRESET，套接 口本身则被关闭。 对方无任何响应：源自 berkeley 的 TCP 发送另外 8 个探测分节，相隔 75 秒一个，试图得到 一个响应。在发出第一个探测分节 11 分钟 15 秒后若仍无响应就放弃。套接口的待处理错 误被置为 ETIMEOUT，套接口本身则被关闭。如 ICMP 错误是"host unreachable(主机不 可达)"，说明对方主机并没有崩溃，但是不可达，这种情况下待处理错误被置为 EHOSTUNREACH。

SO_DONTLINGER 若为真，则 SO_LINGER 选项被禁止。
SO_LINGER 延迟关闭连接 struct linger
上面这两个选项影响 close 行为
选项 间隔 关闭方式 等待关闭与否
SO_DONTLINGER 不关心 优雅 否
SO_LINGER 零 强制 否
SO_LINGER 非零 优雅 是
若设置了 SO_LINGER（亦即 linger 结构中的 l_onoff 域设为非零，参见 2.4，4.1.7 和 4.1.21 各节），并设置了零超时间隔，则 closesocket()不被阻塞立即执行，不论是否有排队数据未发送或未被确认。这种关闭方式称为"强制"或"失效"关闭，因为套接口的虚电路立即被复位，且丢失了未发送的数据。在远端的 recv()调用将以 WSAECONNRESET 出错。
若设置了 SO_LINGER 并确定了非零的超时间隔，则 closesocket()调用阻塞进程，直到所剩数据发送完毕或超时。这种关闭称为"优雅的"关闭。请注意如果套接口置为非阻塞且 SO_LINGER 设为非零超时，则 closesocket()调用将以 WSAEWOULDBLOCK 错误返回。
若在一个流类套接口上设置了 SO_DONTLINGER（也就是说将 linger 结构的 l_onoff 域设为零；参见 2.4，4.1.7，4.1.21 节），则 closesocket()调用立即返回。但是，如果可能，排队的数据将在套接口关闭前发送。

SO_OOBINLINE 带外数据放入正常数据流,在普通数据流中接收带外数据 int

SO_RCVBUF 接收缓冲区大小 int
设置接收缓冲区的保留大小
与 SO_MAX_MSG_SIZE 或 TCP 滑动窗口无关，如果一般发送的包很大很频繁，那么使用这个选项

SO_SNDBUF 发送缓冲区大小 int
设置发送缓冲区的保留大小
与 SO_MAX_MSG_SIZE 或 TCP 滑动窗口无关，如果一般发送的包很大很频繁，那么使用这个选项
每个套接口都有一个发送缓冲区和一个接收缓冲区。 接收缓冲区被 TCP 和 UDP 用来将接收到的数据一直保存到由应用进程来读。 TCP：TCP 通告另一端的窗口大小。 TCP 套接口接收缓冲区不可能溢出，因为对方不允许发出超过所通告窗口大小的数据。 这就是 TCP 的流量控制，

如果对方无视窗口大小而发出了超过宙口大小的数据，则接 收方 TCP 将丢弃它。 UDP：当接收到的数据报装不进套接口接收缓冲区时，此数据报就被丢弃。UDP 是没有 流量控制的；快的发送者可以很容易地就淹没慢的接收者，导致接收方的 UDP 丢弃数据报。

SO_RCVLOWAT 接收缓冲区下限 int
SO_SNDLOWAT 发送缓冲区下限 int
每个套接口都有一个接收低潮限度和一个发送低潮限度。它们是函数 selectt 使用的， 接收低潮限度是让 select 返回"可读"而在套接口接收缓冲区中必须有的数据总量。——对于一个 TCP 或 UDP 套接口，此值缺省为 1。发送低潮限度是让 select 返回"可写" 而在套接口发送缓冲区中必须有的可用空间。对于 TCP 套接口，此值常缺省为 2048。 对于 UDP 使用低潮限度， 由于其发送缓冲区中可用空间的字节数是从不变化的，只要 UDP 套接口发送缓冲区大小大于套接口的低潮限度，这样的 UDP 套接口就总是可写的。 UDP 没有发送缓冲区，只有发送缓冲区的大小。

SO_RCVTIMEO 接收超时 struct timeval
SO_SNDTIMEO 发送超时 struct timeval
SO_BROADCAST：获得或设置 socket 状况，使之可以广播发送数据报。(只能用于 UDP 方式)。
SO_REUSEADDR：设置该 socket 绑定的端口可以被重用。
注意：在 Linux 系统中，如果一个 socket 绑定了某个端口，该 socket 正常关闭或程序退出后，在一段时间内该端口依然保持被绑定的状态，其他程序（或者重新启动的原程序）无法绑定该端口。可以通过调用以下语句避免该问题：

```
opt = 1;
len = sizeof(opt);
setsockopt(sockfd,SOL_SOCKET,SO_REUSEADDR,&opt,&len);
```

SO_EXCLUSIVEADDRUSE
独占模式使用端口,就是不充许和其它程序使用 SO_REUSEADDR 共享的使用某一端口。
在确定多重绑定使用谁的时候，根据一条原则是谁的指定最明确则将包递交给谁，而且没有权限之分，也就是说低级权限的用户是可以重绑定在高级权限如服务启动的端口上的,这是非常重大的一个安全隐患,
如果不想让自己程序被监听，那么使用这个选

获得或改变 socket 的 I/O 属性：

```
int ioctl(int sockfd,long cmd,unsigned long* argp);
```

cmd 属性类型，argp 属性的参数。
常用的有：
FIONREAD，返回 socket 缓冲区中未读数据的字节数
FIONBIO，argp 为零时为阻塞模式，非零时为非阻塞模式
SIOCATMARK ，判断是否有未读的带外数据（仅用于 TCP 协议），返回 true 或 false

```
int fcntl(int fd, int cmd, long arg);
```

F_SETFL，arp 为 O_NONBLOCK 时进入非阻塞模式，为 0 时进入阻塞模式。
F_GETFL，获得属性。

调用 ioctl 获取系统网络接口的信息：

```
struct ifconf{
    int ifc_len; //为 ifc_buf 缓冲区的大小
    union{
        caddr_t ifcu_buf;  //被定义为 ifc_buf , 用户定义传递到内核
        struct ifreq *ifcu_req; //被定义为 ifc_req, 内核返回到用户
    }ifc_ifcu;
};
sturct ifreq{
    char ifr_name[IFNAMSIZ]; //网卡名字，如 hme0，IFNAMSIZ=16
    union {
        struct sockaddr ifru_addr; //定义为 ifr_addr,主 IP 地址;
        struct sockaddr ifru_dstaddr; //定义为 ifr_dstaddr;
        struct sockaddr ifru_broadaddr; //定义为 ifr_broadaddr;
        short ifru_flags; //定义为 ifr_flags，参考<net/if.h>
        int ifru_flags; //定义为 ifr_flags
        int ifru_metric; //定义为 ifr_metric
        caddr_t ifru_data; //定义为 ifr_data
     }ifr_ifru;
    }
```

例子：（在 SOLARIS 环境下通过）
输入 IP 地址,返回 MAC 地址:

```
#include <sys/types.h>
#include <sys/sockio.h>
#include <sys/param.h>
#include <errno.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <netdb.h>
#include <sys/types.h>
#include <unistd.h>
#include <stropts.h>
#include <net/if.h>
#include <stdio.h>
#include <sys/ioctl.h>
#include <net/if_arp.h>

int main(int argc, char **argv)
{
  int sockfd;
  char *ptr;
  struct arpreq arp;
  struct sockaddr_in *arptr;
```

```
         sockfd = socket(AF_INET, SOCK_STREAM, 0); //create any type of socket
        arptr = (struct sockaddr_in *)&arp.arp_pa;
        bzero(arptr, sizeof(struct sockaddr_in));
        arptr->sin_family = AF_INET;
        arptr->sin_addr.s_addr = inet_addr(argv[1]);
        ioctl(sockfd, SIOCGARP, &arp);
        printf("addr = %s, arp mac = %x %x %x %x %x %x %x %x\n",
inet_ntoa(arptr->sin_addr), arp.arp_ha.sa_data[0], arp.arp_ha.sa_da
ta[1], arp.arp_ha.sa_data[2], arp.arp_ha.sa_data[3], arp.arp_ha.sa_data[4],
arp.arp_ha.sa_data[5], arp.arp_ha.sa_data[6], arp.arp_ha
.sa_data[7], arp.arp_ha.sa_data[8]);
        ptr = arp.arp_ha.sa_data;
        printf("mac = %x:%x:%x:%x:%x:%x\n", *(ptr)&0xff, *(ptr+1)&0xff,
*(ptr+2)&0xff, *(ptr+3)&0xff, *(ptr+4)&0xff, *(ptr+5)&0xff);
}
```

根据网卡名字输出 MAC 地址：

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <netinet/in.h>
#include <net/if.h>
#include <net/if_arp.h>
#include <arpa/inet.h>
#include <errno.h>
#include <sys/sockio.h>

#define ETH_NAME   "eri0"

int main()
{
   int sock;
   struct sockaddr_in sin;
   struct ifreq ifr;

   sock = socket(AF_INET, SOCK_DGRAM, 0);
   if (sock == -1)
   {
      perror("socket");
      return -1;
   }

   strncpy(ifr.ifr_name, ETH_NAME, IFNAMSIZ);
```

```
    ifr.ifr_name[IFNAMSIZ - 1] = 0;

    if (ioctl(sock, SIOCGIFADDR, &ifr) < 0)
    {
        perror("ioctl");
        return -1;
    }

    memcpy(&sin, &ifr.ifr_addr, sizeof(sin));
    fprintf(stdout, "hme0: %s\n", inet_ntoa(sin.sin_addr));

    return 0;
}
```

输出当前系统所有 UP 状态的网络接口的 IP 地址,MAC 地址等信息

```
#include <sys/types.h>
#include <sys/sockio.h>
#include <sys/param.h>
#include <errno.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <netdb.h>
#include <sys/types.h>
#include <unistd.h>
#include <stropts.h>
#include <net/if.h>
#include <stdio.h>
#include <sys/ioctl.h>
#include <net/if_arp.h>

#define INTERFACE_NUM 5

struct my_inte{
        char inte_name[16];
        char mac[8];
        short mac_len;
        short flags;
        struct sockaddr primary_addr;
        struct sockaddr broad_addr;
        struct sockaddr dest_addr;
        struct sockaddr netmask;
        struct my_inet *next;

};
```

```
void list(struct my_inte* ptr)
{
        struct sockaddr_in *addr;

        printf("inteface_name = %s\n", ptr->inte_name);
        printf("mac = %x:%x:%x:%x:%x:%x\n", *(ptr->mac)&0xff, *(ptr->mac+1)&0xff,
*(ptr->mac+2)&0xff, *(ptr->mac+3)&0xff, *(ptr->mac
+4)&0xff, *(ptr->mac+5)&0xff);
        printf("flags = %d      ", ptr->flags);
        if (ptr->flags & IFF_UP) printf("up ");
        if (ptr->flags & IFF_BROADCAST) printf("broadcast ");
        if (ptr->flags & IFF_MULTICAST) printf("multicast ");
        if (ptr->flags & IFF_LOOPBACK)  printf("loopback ");
        if (ptr->flags & IFF_POINTOPOINT) printf("point to point ");
        printf("\n");

        addr = (struct sockaddr_in*)&ptr->primary_addr;
        printf("primary_addr = %s\n", inet_ntoa(addr->sin_addr));
        addr = (struct sockaddr_in*)&ptr->netmask;
        printf("netmask = %s\n", inet_ntoa(addr->sin_addr));
        addr = (struct sockaddr_in*)&ptr->broad_addr;
        printf("broad_addr = %s\n", inet_ntoa(addr->sin_addr));
        addr = (struct sockaddr_in*)&ptr->dest_addr;
        printf("dest_addr = %s\n\n", inet_ntoa(addr->sin_addr));
}

int main(int argc, char **argv)
{
  int sockfd, len, lastlen, flags;
  char *buf, *ptr;
  struct ifconf interfaces;
  struct ifreq *inteptr, intecopy;
  struct my_inte *head, *tail, *my_inte_ptr, *tmpptr;
  struct arpreq arp;
  struct sockaddr_in *arptr;

  sockfd = socket(AF_INET, SOCK_STREAM, 0); //create any type of socket
  lastlen = 0;
  len = INTERFACE_NUM * sizeof(struct ifreq);
  for (;;) {
        buf = (char *)malloc(len);
        interfaces.ifc_len = len;
        interfaces.ifc_buf = buf;
```

```
        if (ioctl(sockfd, SIOCGIFCONF, &interfaces) < 0){
                if (errno != EINVAL || lastlen != 0){ //if return EINVAL and not
return succeed
                        perror("ioctl error");
                        return;
                }
        }
        else
        {
                if (interfaces.ifc_len == lastlen)
                        break;
                lastlen = interfaces.ifc_len;
        }
        len += INTERFACE_NUM * sizeof(struct ifreq);
        free(buf);

  }

  head = tail = NULL;
  for (ptr = buf; ptr < buf + interfaces.ifc_len; ){
        inteptr = (struct ifreq *)ptr;
        ptr += sizeof(inteptr->ifr_name) + sizeof(struct sockaddr); //ptr pioint
to next ifreq struct
        intecopy = *inteptr;
        ioctl(sockfd, SIOCGIFFLAGS, &intecopy);
        flags = intecopy.ifr_flags;
        if (flags & IFF_UP == 0) //ignore down network card
                continue;

        my_inte_ptr = (struct my_inte_ptr*)calloc(1, sizeof(struct my_inte));
//create a self-defined struct
        my_inte_ptr->next = NULL;
        my_inte_ptr->flags = flags;
        bzero(&(my_inte_ptr->inte_name), 16);
        strncpy(my_inte_ptr->inte_name, inteptr->ifr_name, 16-1);
        memcpy(&my_inte_ptr->primary_addr, &(*inteptr).ifr_addr, sizeof(struct
sockaddr)); //IP addr

        arptr = (struct sockaddr_in *)&arp.arp_pa;
        bzero(arptr, sizeof(struct sockaddr_in));
        arptr->sin_family = AF_INET;
        arptr->sin_addr           =           ((struct           sockaddr_in
*)(&my_inte_ptr->primary_addr))->sin_addr;
        ioctl(sockfd, SIOCGARP, &arp);
```

```
        printf("addr = %s, arp mac = %x %x %x %x %x %x %x %x\n",
inet_ntoa(arptr->sin_addr), arp.arp_ha.sa_data[0], arp.arp_ha.sa_da
ta[1],  arp.arp_ha.sa_data[2],  arp.arp_ha.sa_data[3],  arp.arp_ha.sa_data[4],
arp.arp_ha.sa_data[5], arp.arp_ha.sa_data[6], arp.arp_ha
.sa_data[7], arp.arp_ha.sa_data[8]);
        memcpy(my_inte_ptr->mac, arp.arp_ha.sa_data, 8); //MAC

        ioctl(sockfd, SIOCGIFNETMASK, &intecopy);
        my_inte_ptr->netmask = intecopy.ifr_addr;

        if (flags & IFF_BROADCAST){
                ioctl(sockfd, SIOCGIFBRDADDR, &intecopy);
                my_inte_ptr->broad_addr = intecopy.ifr_broadaddr;

        }

        if (flags & IFF_POINTOPOINT){
                ioctl(sockfd, SIOCGIFDSTADDR, &intecopy);
                my_inte_ptr->dest_addr = intecopy.ifr_dstaddr;
        }

        if (head == NULL)
                head = tail = my_inte_ptr;
        else
                tail->next = my_inte_ptr;
  }
  free(buf);

//list the list
  for (my_inte_ptr = head; my_inte_ptr != NULL; ){
        tmpptr = my_inte_ptr;
        my_inte_ptr= my_inte_ptr->next;
        list(tmpptr);
        free(tmpptr);
  }

}
```

ARP 高速缓存：
由 ioctl 维护

```
stuct arpreq{
    struct sockaddr apr_pa;//IP 地址
    struct sockaddr arp_ha;//MAC 地址
    int arp_flags;
```

}

相关的操作有：

1. SIGCSARP：增加或修改一个 ARP 高速缓冲区的条目
2. SIGCDARP：删除一个条目
3. SIGCGARP：取出一个条目

路由套接口：

通过路由套接口交换的结构有 3 种：

1. Struct rt_msghdr{

```
U_short rtm_msglen;//消息长度，包括头和其后的 SOCKET 地址结构
U_char rtm_version;
U_char rtm_type; //操作类型，如 RTM_GET
U_short rtm_index;
int rtm_flags;
int rtm_addrs; //位掩码，如 RTA_DST,表示头结构后的 SOCKET 地址结构的类型
pid_t rtm_pid;
int rtm_seq;
int rtm_errno;
int rtm_inits;
struct rt_metrics rtm_rmx;
```

}

2. Struct if_msghdr{…}
3. Struct ifa_msghdr{…}

以上消息结构由用户填写部分信息发往内核，返回的数据在 struct …_msghdr 结构之后将最多有 8 个 SOCKET 结构，为：DST、GATEWAY、NETMASK、GENMASK、IFP、IFR、AUTHOR、BRD 套接字结构

十一、

十二、

十三、（待续）