

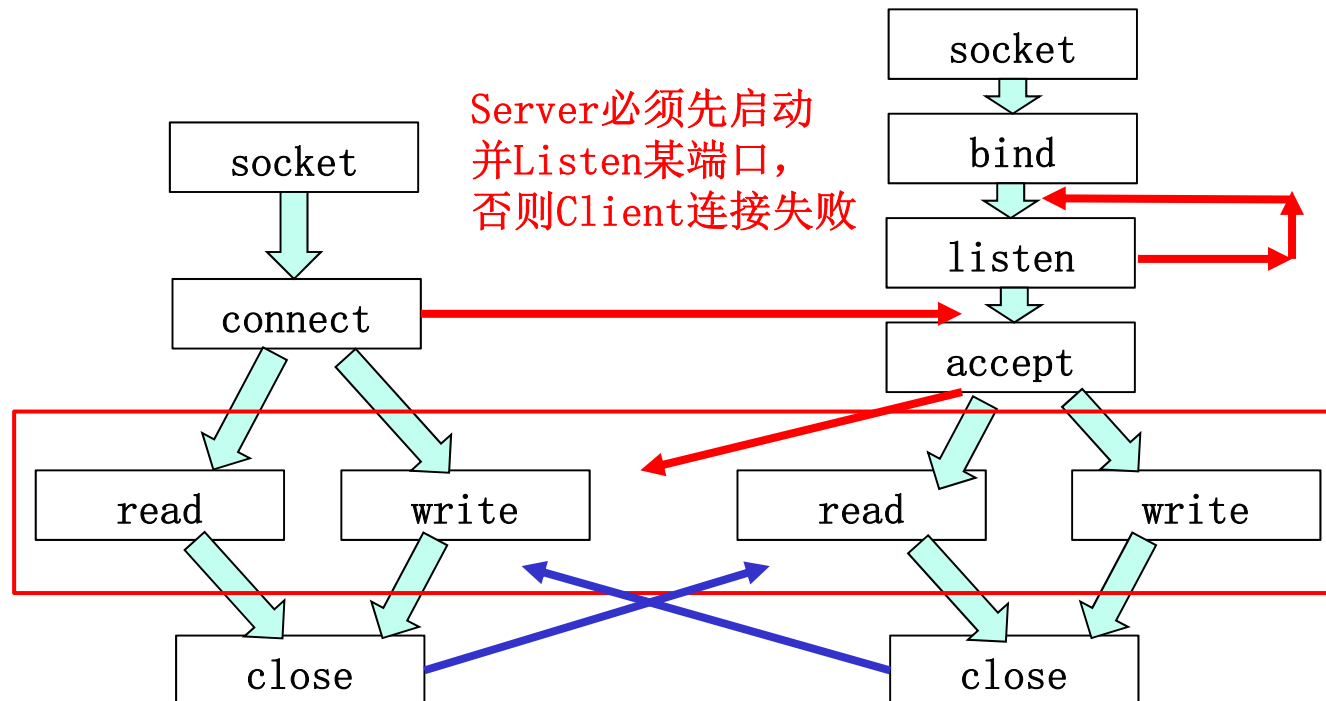
# §. TCP Socket的基本使用

## 1. Socket的基本概念

网络上的两个程序通过一个双向的通信连接实现数据的交换，这个连接的一端称为一个socket，也称作“套接字”，用于描述IP地址和端口，是一个通信链路的句柄，可以用来实现不同计算机之间的通信

在Internet上的主机一般会运行多个服务软件，同时提供几种服务。每种服务绑定一个或多个端口号，不同的服务对应不同的端口，一个端口号可以为多个客户提供相同的服务

## 2. Socket连接的基本过程



# § . TCP Socket的基本使用

## 3. Server端绑定监听端口

```
struct sockaddr_in sin;
int      sockfd;
int      opt;

/* 建立socket */
sockfd = socket(AF_INET, SOCK_STREAM, 0);
if (sockfd < 0)
    return -1;

/* 置端口重用(不设置则无法立即再次绑定端口) */
opt=1;
if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, (char *)&opt, sizeof(opt)) < 0) {
    printf("errno=%d(%s)\n", errno, strerror(errno));
    close(sockfd);
    return -2;
}

/* 绑定端口: 一台主机可能有几个IP地址, bind时, 若指定INADDR_ANY, 表示系统并不立即
   指定IP地址, 而是等客户请求连接时, 用客户请求的目的IP地址作为服务器的源地址 */
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = htonl(INADDR_ANY); /* 网络序 */
sin.sin_port = htons(bind_port); /* 网络序 */
if (bind(sockfd, (struct sockaddr *)&sin, sizeof(struct sockaddr)) < 0) {
    printf("errno=%d(%s)\n", errno, strerror(errno));
    close(sockfd);
    return -3;
}

/* 进入侦听状态: listen的第2个参数表示同时进入侦听队列的最大数量, 某些系统
   用0表示允许连接个数为任意, 但有些系统不识别, 因此统一用一个正数 */
if (listen(sockfd, 200) < 0) {
    printf("errno=%d(%s)\n", errno, strerror(errno));
    close(sockfd);
    return -4;
}

/* 执行到此, 代表Server端已准备好, 可以用来接受连接了 */

/* 进入 accept, 等待对方连接(阻塞, 等对方connect) */
struct sockaddr_in cliaddr;
int new_sock;

len=sizeof(struct sockaddr_in);
new_sock = accept(bind_sockfd,
                  (struct sockaddr *)&cliaddr, &len);
if (new_sock < 0) {
    printf("errno=%d(%s)\n", errno, strerror(errno));
    return -1;
}
```

## § . TCP Socket的基本使用

### 3. Server端绑定监听端口

另：怎么取得本机的所有IP地址？

```
int get_all_ip(int sock)
{
    struct ifconf    intf;
    struct in_addr   addr;
    char             buf[1024];
    struct ifreq      *ifreq;
    int              i;

    /* 初始化 intf */
    intf.ifc_len = 1024;          //buf的定义长度
    intf.ifc_buf = buf;

    /* 获取所有接口的信息 */
    ioctl(sock, SIOCGIFCONF, &intf);

    /* 取所有的IP地址 */
    ifreq = (struct ifreq*)buf;
    for(i=(intf.ifc_len/sizeof(struct ifreq));i>0;i--){
        addr = ((struct sockaddr_in*) &(ifreq->ifr_addr))->sin_addr;
        printf("本机IP地址: %s\n", inet_ntoa(addr));

        ifreq++;
    }

    return 0;
}
```

# § . TCP Socket的基本使用

## 4. Client端进行连接

```
struct sockaddr_in serv_addr;
int sockfd;

/* 建立socket */
if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    return -1;

/* 对源端口和源地址进行绑定（正常情况下不需要）*/
if (需要绑定源地址/源端口) {
    struct sockaddr_in my_addr;
    u_int local_ip = ****;
    u_short local_port = ****;
    bzero((char *)&my_addr, sizeof(struct sockaddr_in));
    my_addr.sin_family = AF_INET;
    my_addr.sin_addr.s_addr = htonl(local_ip);
    my_addr.sin_port = htons(local_port);
    if (bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr)) < 0) {
        close(sockfd); //可打印errno/streerror, 略
        return -2;
    }
} //可以只绑定源端口或源地址中的一个

/* 连接服务端 */
bzero((char *)&serv_addr, sizeof(struct sockaddr_in));
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = htonl(remote_ip);
serv_addr.sin_port = htons(remote_port);

if (connect(sockfd, (struct sockaddr *)&serv_addr, sizeof(struct sockaddr)) < 0) {
    close(sockfd); //可打印errno/streerror, 略
    return -3;
}

//若能执行到此, 则连接成功
```

绑定源端口是防止程序重入的常用方法之一，  
其余防止方法还有  
1、对某个特定文件加锁  
2、将自己的进程号写入文件  
3、...

## § . TCP Socket的基本使用

### 5. 连接成功后Socket的读

- 从文件描述符中读内容

```
ssize_t read(int fd, void *buf, size_t count);
```

- 从socket中读内容

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

- 常用写法:

```
#define MAX_READLEN    1000
```

```
char buf[MAX_READLEN];
```

```
int n;
```

```
n=read (sockfd, buf, sizeof(buf));
```

```
n=recv (sockfd, buf, sizeof(buf), 0);
```

- 返回值是实际读到的数据，-1代表出错(置errno)

- 参数3是缓冲区最大长度，和返回值不同，不要混淆，无论是同步还是异步方式，  
均不是返回最大长度

[ 同步：无数据则阻塞，读到任意数据或错误则返回

[ 异步：无数据立即返回-1（errno置值），读到任意数据或错误则返回  
=> 要select返回可读，才能去读(包括错误)

## § . TCP Socket的基本使用

### 5. 连接成功后Socket的读

- recv的第4个参数非0可以实现特殊功能

```
n=recv(sockfd, buf, sizeof(buf), MSG_OOB);
```

读TCP协议的 out-of-band 数据

```
n=recv(sockfd, buf, sizeof(buf), MSG_PEEK);
```

读到数据后，缓冲区中并不删除，仍可继续读

```
n=recv(sockfd, buf, sizeof(buf), MSG_DONTWAIT);
```

无论sockfd当前什么模式，临时当异步模式处理

```
n=recv(sockfd, buf, sizeof(buf), MSG_WAITALL);
```

读到第3个参数指定的长度才返回，对于异步模式无效（如果读的过程中有中断产生，仍可能返回小于指定长度的字节，因此仍要判断返回值）

## § . TCP Socket的基本使用

### 6. 连接成功后Socket的写

说明:

写的本质不是进行发送操作,而是把用户态的数据copy 到系统底层去,然后再由系统进行发送操作, send/write返回成功, 只表示数据已经copy 到底层缓冲,而不表示数据已经发出,更不表示对方端口已经收到数据

- 向文件描述符写内容

```
ssize_t write(int fd, const void *buf, size_t count);
```

- 向socket写内容

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

- 常用写法:

```
#define MAX_WRITELEN    1000
char buf[MAX_WRITELEN];
int n;
... //给buf赋内容, buflen为实际内容长度
n=write(sockfd, buf, buflen);
n=send (sockfd, buf, buflen, 0);
```

- 返回值是实际写入的数据, -1代表出错(置errno)

- 参数3是要写入的长度, 和返回值不完全相同

同步: 如果数据没写完(未达到参数3指定的值)则阻塞, 如出现错误则返回小于指定长度的值

异步: 若缓冲区满则可能返回小于参数3指定的值再次写则立即返回-1 (errno置值)

=> 要select返回可写, 才能去写(包括错误)

## § . TCP Socket的基本使用

### 6. 连接成功后Socket的写

- send的第4个参数非0可以实现特殊功能

```
n=send(sockfd, buf, buflen, MSG_OOB);
```

写TCP协议的 out-of-band 数据

```
n=send(sockfd, buf, buflen, MSG_DONTWAIT);
```

无论sockfd当前什么模式，临时当异步模式处理



## § . TCP Socket的基本使用

### 7. 连接的关闭

全部关闭:

```
int close(int fd);
```

半关闭:

```
int shutdown(int sockfd, int how);
```

- SHUT\_RD: 关闭socket的读这一半, 不再接收socket中的数据且收缓冲区的数据作废
- SHUT\_WR: 关闭socket的写这一半, 但发送缓冲区中的数据仍会被发送
- SHUT\_RDWR: 相当于close

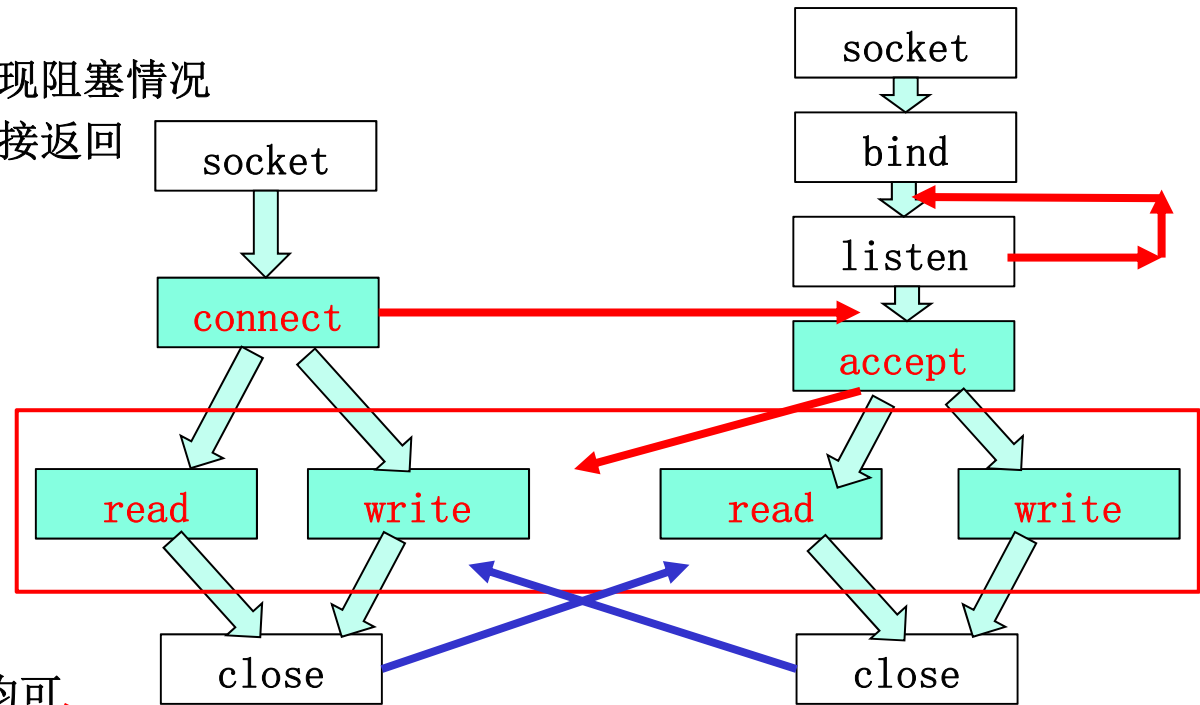
## §. TCP Socket的基本使用

### 8. 同异步的基本概念

同步：某些操作函数在使用时会出现阻塞情况

异步：任何操作函数均不阻塞，直接返回

#### ● 受影响的函数：



#### ● 设置异步的方法：任意socket均可

```
int sockfd, val;

if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    return -1;

/* 设为非阻塞模式 */
if ((val = fcntl(sockfd, F_GETFL, 0)) < 0) {
    close(sockfd); //可打印errno/strerro
    return -2;
}
if (fcntl(sockfd, F_SETFL, val|O_NONBLOCK) < 0) {
    close(sockfd); //可打印errno/strerro
    return -3;
}
```

connect  
listen  
accept

## § . TCP Socket的基本使用

### 9. select函数的使用

#### ● struct fd\_set结构

fd\_set可以理解为一个集合，在RHEL7 x86\_64下集合大小是128字节(即sizeof(fd\_set)=128)

该集合的每个bit位对应一个文件句柄(fd/sockfd)，由程序置初值，交由select函数阻塞后，如果某个文件句柄有事件发生，则对应置位后select函数返回，然后再由程序去判断哪个文件描述符有变化，进而处理

=> select函数能用极小的代价监视多个文件句柄上发生的事件

#### ● 与struct fd\_set相关的几个集合操作

void FD_ZERO(fd_set *set);	清空整个文件句柄集合
void FD_SET(int fd, fd_set *set);	将一个文件句柄添加到指定的fd_set中
int FD_ISSET(int fd, fd_set *set);	检查指定的fd_set中该文件句柄是否置位
void FD_CLR(int fd, fd_set *set);	清空指定的fd_set中该文件句柄的置位

## § . TCP Socket的基本使用

### 9. select函数的使用

#### ● select函数的工作流程

例：假设fd\_set长度为1个字节，现有的文件句柄是 0, 2, 5, 7（1字节最多8个fd，值0-7）

则：fd\_set rfd, wfd;

(1) FD\_ZERO(&rfd);

FD\_ZERO(&wfd);

=> rfd: 0000 0000      wfd: 0000 0000

(2) FD\_SET(fileno(stdin), &rfd) //假设0

FD\_SET(sock, &rfd) //假设2

FD\_SET(listensock, &rfd) //假设7

FD\_SET(sock, &wfd) //假设2

FD\_SET(sock1, &wfd) //假设5

=> rfd: 1000 0101      wfd: 0010 0100

=> 最大fd是7

(3) 执行select(8, &rfd, &wfd, NULL, NULL)

进入阻塞等待

(4) 假设0/2发生可读事件，2/5发生可写事件

=> select返回4

=> rfd: 0000 0101      wfd: 0010 0100

(5) 处理select函数的返回

=> 本次select处理完毕，回步骤1再次循环

```
if (FD_ISSET(fileno(stdin), &rfd)) { //0
    FD_CLR(fileno(stdin), &rfd);
    处理stdin的读 (本次有)
}
if (FD_ISSET(fileno(stdin), &rfd)) { //0
    FD_CLR(fileno(stdin), &rfd);
    处理stdin的读 (本次有)
}
if (FD_ISSET(sock, &rfd)) { //2
    FD_CLR(sock, &rfd);
    处理sock的读 (本次有)
}
if (FD_ISSET(listensock, &rfd)) { //7
    FD_CLR(listensock, &rfd);
    处理listen的读 (本次没有但仍要判断)
}
if (FD_ISSET(sock, &wfd)) { //2
    FD_CLR(sock, &wfd);
    处理sock的写 (本次有)
}
if (FD_ISSET(sock1, &wfd)) { //5
    FD_CLR(sock1, &wfd);
    处理sock1的写 (本次有)
}
```

# § . TCP Socket的基本使用

## 9. select函数的使用

### ● select函数的原型定义

```
int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
```

nfds : select监视的文件描述符数 (最大的fd+1)  
readfds : select监视的可读文件描述符集合  
writefds : select监视的可写文件描述符集合  
exceptfds : select监视的异常文件描述符集合 (不常用)  
timeout : 本次select的超时结束时间

select的返回值:  
正数: 读写置位的数量  
0 : 超时  
-1 : select被中断

### ● struct timeval结构

```
struct timeval {  
    long tv_sec;    //second  
    long tv_usec;  //microsecond  
};
```

理论精度可到百万分之1秒(us), 实际不可能

```
struct timeval tm;  
tm.tv_sec = 1;  
tm.tv_usec = 500*1000;  
sel = select(maxfd+1, &rfd, &wfd, NULL, &tm);
```

=> 每1.5秒超时一次

### ● 小技巧

```
struct timeval tm;  
tm.tv_sec = **;  
tm.tv_usec = **;  
select(0, NULL, NULL, NULL, &tm);  
=> 可以进行一个相对精度较高(ms级)的  
延时(虽然可置usec, 但us级做不到)
```

## § . TCP Socket的基本使用

### 9. select函数的使用

- select函数的原型定义
- struct timeval结构

```
struct timeval tm;  
tm.tv_sec  = 1;  
tm.tv_usec = 500*1000;  
sel = select(maxfd+1, &rfd, &wfd, NULL, &tm);
```

=> 每1.5秒超时一次

=> 超时指无任何事件发生时，每tm秒select返回0，若有读写事件发生，则select立即返回

=> 读写事件发生后，不重置tm而继续select，会继续等到剩余时间而超时

=> select(maxfd+1, &rfd, &wfd, NULL, NULL)则无限阻塞(有事件发生/中断返回，不返回0)

作业中：为什么要select+设置定时器(alarm)，  
而不直接用select超时？

原因：

- 1、select读写事件发生后，处理需要一定的时间，时间累积后，会导致超时时间不够准确（假设需要高精度时间）
- 2、若定时器到，select会因为 SIGALARM 信号而中断，此时select返回-1（想让大家了解 -1 不一定错误，还需要判断errno的值 errno = EINTR表示中断到）

应用场景：如果Server端同时维护多条连接，且每条连接需要用keepalive(心跳包)来维护状态，如何实现？

- 1、心跳包只有在本连接上无数据发送时才发送，因此每条连接的发送时间不一致，不适合select统一超时处理
- 2、常用的解决方案是定时器单位时间产生中断，各连接依据自身情况累加并判断是否需要发送心跳包

## § . TCP Socket的基本使用

### 9. select函数的使用

#### ● 异步方式connect的特殊处理(异步connect返回-1)

```
if (connect(sockfd, (struct sockaddr *)&serv_addr, sizeof(struct sockaddr))<0) {  
    fd_set wfd;  
    struct timeval tm;  
  
    FD_ZERO(&wfd);  
    FD_SET(sockfd, &wfd);  
    tm.tv_sec = ***;        // 秒  
    tm.tv_usec = ***;       // 微秒  
    sel = select(sockfd+1, NULL, &wfd, NULL, &tm);  
    if (sel<=0) {  
        //连接失败, 处理, 打印errno/strerrno等  
        return 0;  
    }  
    //能执行到此说明连接成功  
}
```

加<0时的返回值判断更好

#### ● select的适用范围

标准输入、标准输出、错误输出等

client端用于connect的sockfd

server端bind端口后用于listen的sockfd

server端由listen的sockfd接受连接后accept的fd

其它(管道等)

#### ● select后如果某个sockfd出错, 则处理关闭, 下次新accept/新socket建立的fd号可能与原来相同

#### ● CentOS7下, sizeof(fd\_set)=128 => select最多监视 128\*8 = 1024个fd且max=1023 => 超过怎么办? (poll函数/epoll函数及对应结构)

## § . TCP Socket的基本使用

### 10. Socket使用时的几个问题

- 两边同异步模式是否一定要相同？

不需要，但若想同时处理多个连接，必须异步方式

- ★ Server端用于listen的socket和已经accept的多个socket需要同时处理

- ★ Client端想同时发起多个连接时

- ★ 如果Server端想维护多连接但又想同步方式，可以fork子进程

- write/send一定每次成功吗？是否需要置select的写集？

- ★ 常规方式（有限数据写入）下，因为写缓冲区较大，一般可直接写

- ★ 极限情况下，填满了对端接收缓冲区+本方发送缓冲区，此时写会出错，因此需置select

- ★ 本方发送缓冲区未满的情况下，置select的writefds会立即返回，因此要按需置select

- server端到底应该是程序还是守护进程？

- ★ Server端一般无人值守，用守护进程


- server端是否应该为每个连接分裂一个子进程？

- ★ 根据实际情况，若多socket之间需要交换数据，应该一个进程维护多条连接，不需要fork

- ★ 如果socket仅用于和Client之间交换数据，则采用fork子进程的方式可以使程序简单，

可使用同步方式，程序不完善时不会导致全部崩溃

- 异步方式无法读到指定长度数据？



```
int total =0, len;

while(total < 希望长度) {
    select, 等待返回并判断sockfd可读
    len = read(sockfd, buf+total, 希望长度-total);
    if (len<=0) {
        错误处理
    }
    total += len;
}
```