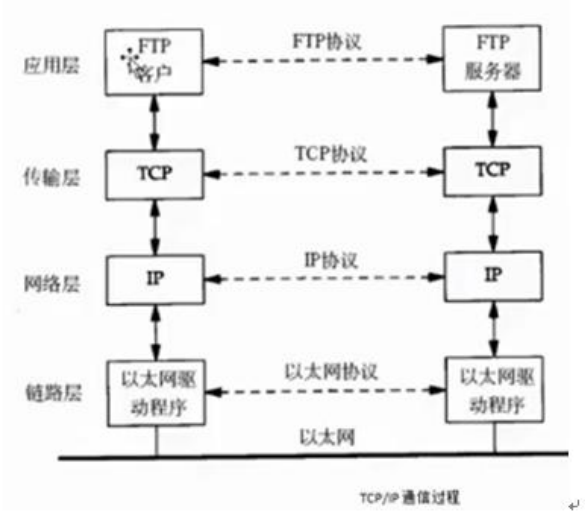
【Socket 基础】

1. 协议的概念
2. b/s c/s
3. 分层模型　7　4
4. 协议格式
   - 数据包基本格式
   - 以太网帧格式
   - arp数据包格式
   - IP段格式
   - TCP/UDP
5. NAT映射　打洞机制
6. 套接字
7. TCP　C/S模型

C/S：服务器/客户端模式，应用于数据量大、稳定性高的场景
B/S：浏览器，只需要开发服务器，但是支持的协议有限制
4 层模型：

应用层：http、ssh、ftp 等其他协议
传输层：TCP/UDP 协议
网络层：IP 协议
网络接口层：链路层，以太网帧协议



无特殊要求，用户只需要完成应用层协议，其他协议由系统进行转换封装
IPv4：32 位的目的地址和源地址
TCP/IP：16 位的端口号
每个路由/交换机：有 NAT 公网 ip 映射表 && 路由表（局域网 ip）
c/s 模式一般端口：8000
b/s 模式一般端口：8080

tcp 流式协议，udp 报式协议

Socket 编程

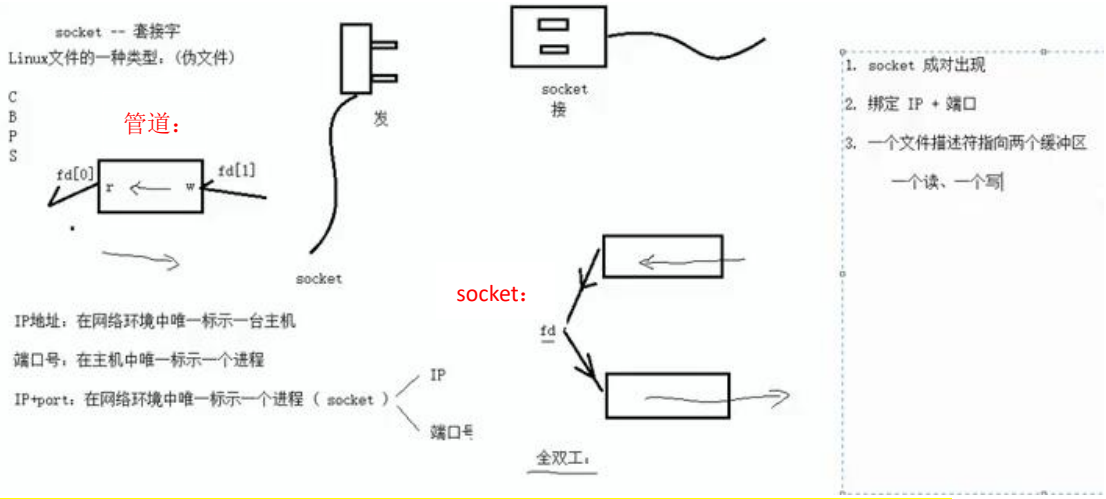
套接字概念

Socket 本身有“插座”的意思，在 Linux 环境下，用于表示进程间网络通信的特殊文件类型。本质为内核借助缓冲区形成的伪文件。

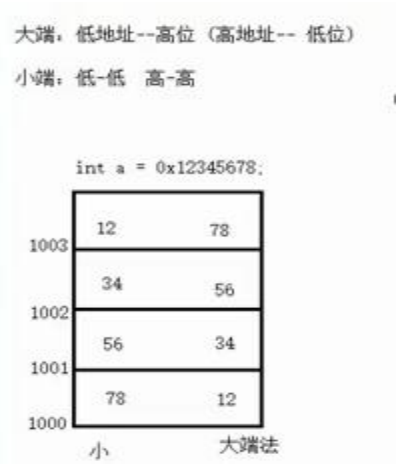
既然是文件，那么理所当然的，我们可以使用文件描述符引用套接字。与管道类似的，Linux 系统将其封装成文件的目的是为了统一接口，使得读写套接字和读写文件的操作一致。区别是管道主要应用于本地进程间通信，而套接字多应用于网络进程间数据的传递。

套接字的内核实现较为复杂，不宜在学习初期深入学习。

在 TCP/IP 协议中，“IP 地址+TCP 或 UDP 端口号”唯一标识网络通讯中的一个进程。“IP 地址+端口号”就对应一个 socket。欲建立连接的两个进程各自有一个 socket 来标识，那么这两个 socket 组成的 socket pair 就唯一标识一个连接。因此可以用 Socket 来描述网络连接的一对一关系。



计算机采用小端数据存储，而 TCP/IP 协议规定的网络数据流采用大端存放!!!



为使网络程序具有可移植性，使同样的 C 代码在大端和小端计算机上编译后都能正常运行，可以调用以下库函数做**网络字节序和主机字节序的转换**。

```
#include <arpa/inet.h>

uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

h to n：host 主机字节序，to， net 网络字节序，
转端口方便

h 表示 host，n 表示 network，l 表示 32 位长整数，s 表示 16 位短整数。

如果主机是小端字节序，这些函数将参数做相应的大小端转换然后返回，如果主机是大端字节序，这些函数不做转换，将参数原封不动地返回。

【IP 地址转换函数】：

inet_p：网络字节序的 ip ，to， n：net，
转 IP 方便

现在：

```
#include <arpa/inet.h>
int inet_pton(int af, const char *src, void *dst);
const char *inet_ntop(int af, const void *src, char *dst, socklen_t size);
```

支持 IPv4 和 IPv6
可重入函数

其中 inet_pton 和 inet_ntop 不仅可以转换 IPv4 的 in_addr，还可以转换 IPv6 的 in6_addr。

因此函数接口是 void *addrptr。

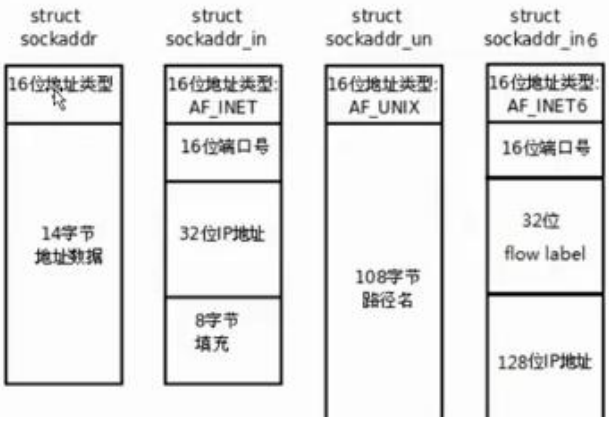
192.168.1.24 --》 unsigned int -》 htonl() -》 网络字节序

192.168.1.24 ----------> 网络字节序 inet_pton();

网络字节序 ----------> 点分十进制字符串 inet_ntop();

sockaddr 数据结构

strcut sockaddr 很多网络编程函数诞生早于 IPv4 协议，那时候都使用的是 sockaddr 结构体,为了向前兼容，现在 sockaddr 退化成了（void *）的作用，传递一个地址给函数，至于这个函数是 sockaddr_in 还是 sockaddr_in6，由地址族确定，然后函数内部再强制类型转化为所需的地址类型。



现在 struct sockaddr 结构体无法定义，只能定义 struct sockaddr_in 或者 struct sockaddr_in6，但是函数 bind、accept、connect，传参时，必须强制类型转换为 struct sockaddr!!!

查看结构体：man 7 ip

【网络套接字函数：】

socket 函数

```
#include <sys/types.h> /* See NOTES */
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
domain:
    AF_INET 这是大多数用来产生 socket 的协议，使用 TCP 或 UDP 来传输，用 IPv4 的地址
    AF_INET6 与上面类似，不过是来用 IPv6 的地址
    AF_UNIX 本地协议，使用在 Unix 和 Linux 系统上，一般都是当客户端和服务器在同一台及其上的时候使用
type:
    SOCK_STREAM 这个协议是按照顺序的、可靠的、数据完整的基于字节流的连接。这是一个使用最多的 socket 类型，这个 socket
是使用 TCP 来进行传输。
    SOCK_DGRAM 这个协议是无连接的、固定长度的传输调用。该协议是不可靠的，使用 UDP 来进行它的连接。
    SOCK_SEQPACKET 该协议是双线路的、可靠的连接，发送固定长度的数据包进行传输。必须把这个包完整的接受才能进行读取。
    SOCK_RAW socket 类型提供单一的网络访问，这个 socket 类型使用 ICMP 公共协议。（ping、traceroute 使用该协议）
    SOCK_RDM 这个类型是很少使用的，在大部分的操作系统上没有实现，它是提供给数据链路层使用，不保证数据包的顺序
protocol:
    传 0 表示使用默认协议。
返回值：
    成功，返回指向新创建的 socket 的文件描述符，失败，返回-1，设置 errno
```

套接字分为网络套接字和本地套接字，使用次关键字就是使用本地套接字

bind 函数

```
#include <sys/types.h> /* See NOTES */
#include <sys/socket.h>
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
sockfd:
    socket 文件描述符
addr:
    构造出 IP 地址加端口号
addrlen:
    sizeof(addr)长度
返回值：
    成功返回 0，失败返回-1，设置 errno
```

服务器程序所监听的网络地址和端口号通常是固定不变的，客户端程序得知服务器程序的地址和端口号后就可以向服务器发起连接，因此服务器需要调用 bind 绑定一个固定的网络地址和端口号。

bind()的作用是将参数 sockfd 和 addr 绑定在一起，使 sockfd 这个用于网络通讯的文件描述符监听 addr 所描述的地址和端口号。前面讲过，struct sockaddr *是一个通用指针类型，addr 参数实际上可以接受多种协议的 sockaddr 结构体，而它们的长度各不相同，所以需要第三个参数 addrlen 指定结构体的长度。如：

绑定端口+ip。

listen 函数

```
#include <sys/types.h> /* See NOTES */
#include <sys/socket.h>
int listen(int sockfd, int backlog);
sockfd:
    socket文件描述符
backlog:
    排队建立3次握手队列和刚刚建立3次握手队列的链接数和

    查看系统默认 backlog

cat /proc/sys/net/ipv4/tcp_max_syn_backlog
```

典型的服务器程序可以同时服务于多个客户端，当有客户端发起连接时，服务器调用的 accept()返回并接受这个连接，如果有大量的客户端发起连接而服务器来不及处理，尚未 accept 的客户端就处于连接等待状态，listen()声明 sockfd 处于监听状态，并且最多允许有 backlog 个客户端处于连接待状态，如果接收到更多的连接请求就忽略。listen()成功返回 0，失败返回-1。

允许同时有 backlog 个客户端请求连接，多的请求忽略。

accept 函数

```
#include <sys/types.h>      /* See NOTES */
#include <sys/socket.h>
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
sockdf:
    socket文件描述符
addr:
    传出参数，返回链接客户端地址信息，含IP地址和端口号
addrlen:
    传入传出参数（值-结果），传入sizeof(addr)大小，函数返回时返回真正接收到地址结构体的大小
返回值，
    成功返回一个新的socket文件描述符，用于和客户端通信，失败返回-1，设置errno
```

注意返回的是客户端的文件描述符，用于数据传输。此函数一般服务器调用。

三方握手完成后，服务器调用 accept()接受连接，如果服务器调用 accept()时还没有客户端的连接请求，就阻塞等待直到有客户端连接上来。addr 是一个传出参数，accept()返回时传出客户端的地址和端口号。addrlen 参数是一个传入传出参数（value-result argument），传入的是调用者提供的缓冲区 addr 的长度以避免缓冲区溢出问题，传出的是客户端地址结构体的实际长度（有可能没有占满调用者提供的缓冲区）。如果给 addr 参数传 NULL，表示不关心客户端的地址。

connect 函数

```
#include <sys/types.h>              /* See NOTES */
#include <sys/socket.h>
int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
sockdf:
    socket文件描述符
addr:
    传入参数，指定服务器端地址信息，含IP地址和端口号
addrlen:
    传入参数,传入sizeof(addr)大小
返回值：
    成功返回0，失败返回-1，设置errno
```

客户端需要调用 connect()连接服务器，connect 和 bind 的参数形式一致，区别在于 bind 的参数是自己的地址，而 connect 的参数是对方的地址。connect()成功返回 0，出错返回-1。

客户端调用。

C/S 设计流程：

server.c

1. socket() 建立套接字
2. bind() 绑定IP 端口号 (struct sockaddr_in addr 初始化)
3. listen() 指定最大同时发起连接数
4. accept() 阻塞等待客户端发起连接
5. read()
6. 小--大
7. write 给 客户端
8. close();

client.c

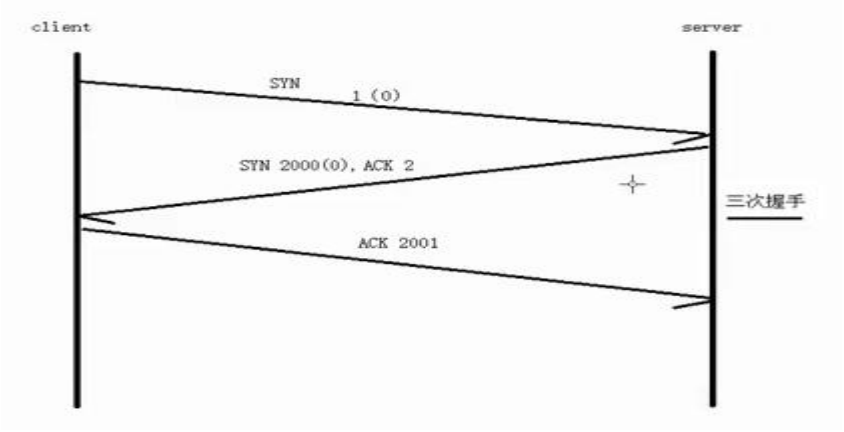
1. socket();
2. bind(); 可以依赖“隐式绑定”
3. connect();发起连接
4. write();
5. read();
6. close();

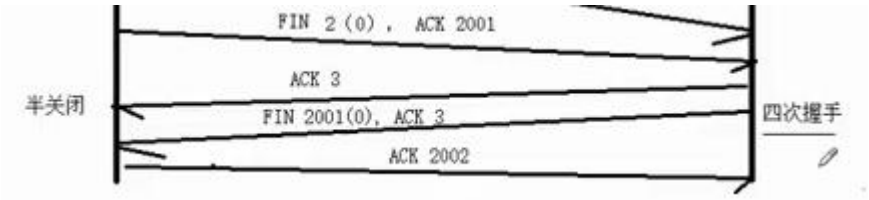
netstat -apn | grep 端口号 ：查看端口，可以 kill 掉

以太网帧格式，决定了 socket 一次包最多传输 1500Byte，若要求传输的字节数>1500，而 read 函数读一次 socket 包最多读完 1500 就返回了，系统慢速调用函数，也可能会被信号中断。

1. > 0 实际读到的字节数
2. **= 0 已经读到结尾（对端已经关闭）**
3. -1 需要进一步判断errno的值
   errno = EAGAIN or EWOULDBLOCK 设置了非阻塞方式读，并且没有数据到达
   errno = EINTR 慢速系统调用被中断
   errno = ECONNRESET 说明收到RST标志，连接被重置。需要close
   errno = “其他” 异常

注意第 2 点：由于以太网帧协议限制，socket 一个数据包最多 1500 字节，read 读取一次 socket 最多 1500 字节，假设要接受的大于 1500 字节，就会拆成多个包发过来，而 read 函数读 socket 一次最多只能 1500 字节，就返回 0 了。故需要多次调用，才能读到自己想要的数据包大小。或自己封装函数。ECONNREST 说明连接双方有一端退出但是未关闭连接，此时需要 close。





3 次握手建立连接，4 次握手断开连接。

3. MTU、mss、半关闭

MTU：最大传输单元　受协议限制　以太网1500　IP 65535

mss：受MTU 标示一个数据包携带数据的上限数。

win：滑动窗口——当前本端 能接收的数据上限值。（单位：字节）

netstat -apn | grep portname：端口状态向下转换的。

TCP 状态：服务器开启（listen 状态）、客户端连接（服务器、客户端都处于 establish 状态）、相互发送数据（状态不变）、关闭客户端（服务器：close_wait 状态、客户端：fin_wait2 状态）、再关闭服务器（无 listen 端口建立）

主动关闭处于 fin_wait2，被动关闭处于 close_wait（主动关闭连接的一方为实现可靠的关闭会先处于 time_wait 状态，2MSL 时间后会无 listen 端口建立）

半关闭

当 TCP 链接中 A 发送 FIN 请求关闭，B 端回应 ACK 后（A 端进入 FIN_WAIT_2 状态），B 没有立即发送 FIN 给 A 时，A 方处在半链接状态，此时 A 可以接收 B 发送的数据，但是 A 已不能再向 B 发送数据。

从程序的角度，可以使用 API 来控制实现半连接状态。

```
#include <sys/socket.h>
int shutdown(int sockfd, int how);
sockfd: 需要关闭的 socket 的描述符
how:    允许为 shutdown 操作选择以下几种方式:
    SHUT_RD(0):     关闭 sockfd 上的读功能，此选项将不允许 sockfd 进行读操作。
                    该套接字不再接受数据，任何当前在套接字接受缓冲区的数据将被无声的丢弃掉。
    SHUT_WR(1):     关闭 sockfd 的写功能，此选项将不允许 sockfd 进行写操作。进程不能在对此套接字发出写操作。
    SHUT_RDWR(2):   关闭 sockfd 的读写功能。相当于调用 shutdown 两次，首先是以 SHUT_RD，然后以 SHUT_WR。
```

使用 close 中止一个连接，但它只是减少描述符的引用计数，并不直接关闭连接，只有当描述符的引用计数为 0 时才关闭连接。

端口复用

在 server 的 TCP 连接没有完全断开之前不允许重新监听是不合理的。因为，TCP 连接没有完全断开指的是 connfd（127.0.0.1:6666）没有完全断开，而我们重新监听的是 lis-tenfd（0.0.0.0:6666），虽然是占用同一个端口，但 IP 地址不同，connfd 对应的是与某个客户端通讯的一个具体的 IP 地址，而 listenfd 对应的是 wildcard address。解决这个问题的方法是使用 setsockopt()设置 socket 描述符的选项 SO_REUSEADDR 为 1，表示允许创建端口号相同但 IP 地址不同的多个 socket 描述符。

在 server 代码的 socket()和 bind()调用之间插入如下代码：

```
int opt = 1;
setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));
```

允许服务器在 timewait 状态下，也能重新开启复用原端口。

【多 IO 服务器高效并发】

此之前有多线程、多进程并发

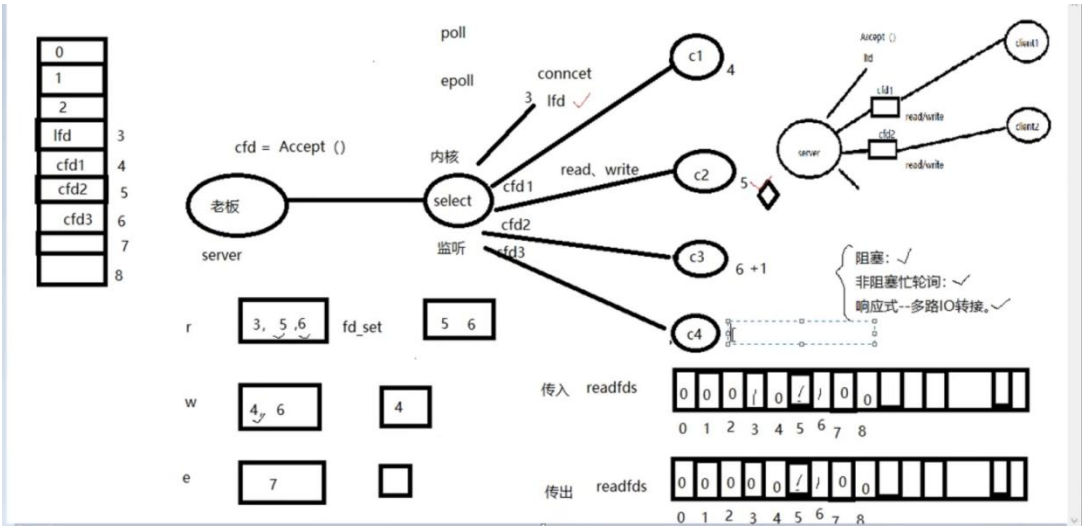
【1、select】

多路 I/O 转接服务器

多路 IO 转接服务器也叫做多任务 IO 服务器。该类服务器实现的主旨思想是，不再由应用程序自己监视客户端连接，取而代之由内核替应用程序监视文件。

主要使用的方法有三种

自己不在阻塞等待连接，而由内核监听请求，在反馈给应用 cx。借助内核的 select 函数进行监听。




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

    nfds:       监控的文件描述符集里最大文件描述符加 1，因为此参数会告诉内核检测前多少个文件描述符的状态
    readfds:    监控有读数据到达文件描述符集合，传入传出参数
    writefds:   监控写数据到达文件描述符集合，传入传出参数
    exceptfds:  监控异常发生达文件描述符集合,如带外数据到达异常，传入传出参数
    timeout:    定时阻塞监控时间，3 种情况
                1.NULL，永远等下去，阻塞
                2.设置 timeval，等待固定时间
                3.设置 timeval 里时间均为 0，检查描述字后立即返回，轮询
```

传入传出：传入要监听的集合，传出的就只有发生了动作的描述符集合

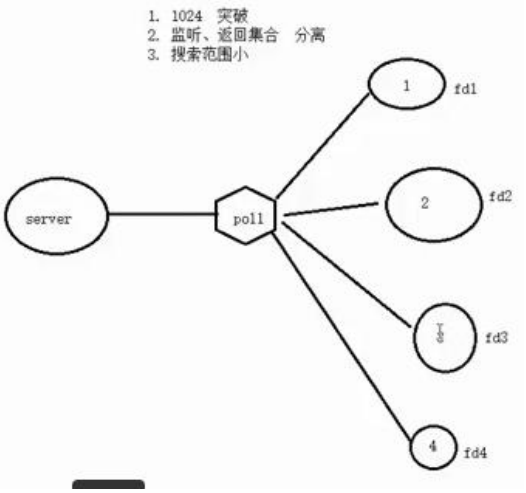
返回值：返回读、写、异常集合中发生了动作的总的描述符个数

连接请求属于读集合（你发来连接请求，我来批准，就要去读），读集合就是对方要发数据/请求过来。

```
struct timeval {
    long tv_sec; /* seconds */
    long tv_usec; /* microseconds */
};
void FD_CLR(int fd, fd_set *set);   //把文件描述符集合里 fd 清 0
int FD_ISSET(int fd, fd_set *set); //测试文件描述符集合里 fd 是否置 1
void FD_SET(int fd, fd_set *set);   //把文件描述符集合里 fd 位置 1
void FD_ZERO(fd_set *set);          //把文件描述符集合里所有位清 0
```

【2、poll】

```
int poll(struct pollfd *fds, nfds_t nfds, int timeout);

                {
        int   fd;       /* file descriptor */
        short events;   /* requested events */
        short revents;  /* returned events */
    };
fds: 数组的首地址。

struct pollfd fds[5000];

fds[0].fd = listenfd;
fds[0].events = POLLIN    /POLLOUT/POLLERR

fds[1].fd = fd1
fds[1].events = POLLOUT

fds[2].fd = fd3
fds[2].events = POLLIN

fds[3].fd = fd3
fds[3].events = POLLIN

poll(fds, 5, -1);
```




```
int poll(struct pollfd *fds, nfds_t nfds, int timeout);
    struct pollfd {
        int fd; /* 文件描述符 */
        short events; /* 监控的事件 */
        short revents; /* 监控事件中满足条件返回的事件，操作系统赋值
                        每一 bit 对应一种  events，判断是否发生该 events：
                        （client[0].revents & POLLIN ） */
    };
events:
    POLLIN          普通或带外优先数据可读
    POLLOUT         普通或带外数据可写
    POLLERR         发生错误
nfds            监控数组中有多少文件描述符需要被监控
timeout         毫秒级等待
        -1：阻塞等，#define INFTIM -1                Linux 中没有定义此宏
        0：立即返回，不阻塞进程
        >0：等待指定毫秒数，如当前系统时间精度不够毫秒，向上取值
```

如果不再监控某个文件描述符时，可以把 pollfd 中，fd 设置为-1，poll 不再监控此 pollfd，下次返回时，会把 revents 设置为 0。返回值也是发生动作的 fd 个数

【3、epoll】

优势比较：最大连接上限数，select 要重新编译内核、poll 需要修改配置文件，且两者只会告诉你有几个请求，并不告诉你是哪几个请求，epoll 具有此优势。

基础 API

1. 创建一个 epoll 句柄，参数 size 用来告诉内核监听的文件描述符的个数，跟内存大小有关。

```
#include <sys/epoll.h>
int epoll_create(int size)      size：监听数目
```

只是内核参考数目

返回值文件描述符，该文件描述符指向一个二叉树（红黑树）的树根

2. 控制某个 epoll 监控的文件描述符上的事件：注册、修改、删除。

往树根上添加/修改/删除节点。

```
#include <sys/epoll.h>
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event)
    epfd：  为 epoll_creat 的句柄
    op：    表示动作，用 3 个宏来表示：
        EPOLL_CTL_ADD (注册新的 fd 到 epfd)，
        EPOLL_CTL_MOD (修改已经注册的 fd 的监听事件)，
        EPOLL_CTL_DEL (从 epfd 删除一个 fd)；
    event： 告诉内核需要监听的事件

    struct epoll_event {
        __uint32_t events; /* Epoll events */
        epoll_data_t data; /* User data variable */
    };
```

```
typedef union epoll_data {
    void *ptr;
    int fd;
    uint32_t u32;
    uint64_t u64;
} epoll_data_t;
```

```
events: EPOLLIN ：    表示对应的文件描述符可以读（包括对端 SOCKET 正常关闭）
        EPOLLOUT:     表示对应的文件描述符可以写
        EPOLLERR:     表示对应的文件描述符发生错误
```

函数第二个参数 fd 与结构体的参数 fd 要对应一致

epoll ET：边沿触发 ：x.events = **EPOLLIN | EPOLLET**，在有动作来临时，通知你一次，一般用于很大数据包分析：先读取包头信息检验是否需要，不需要直接清空缓冲区；

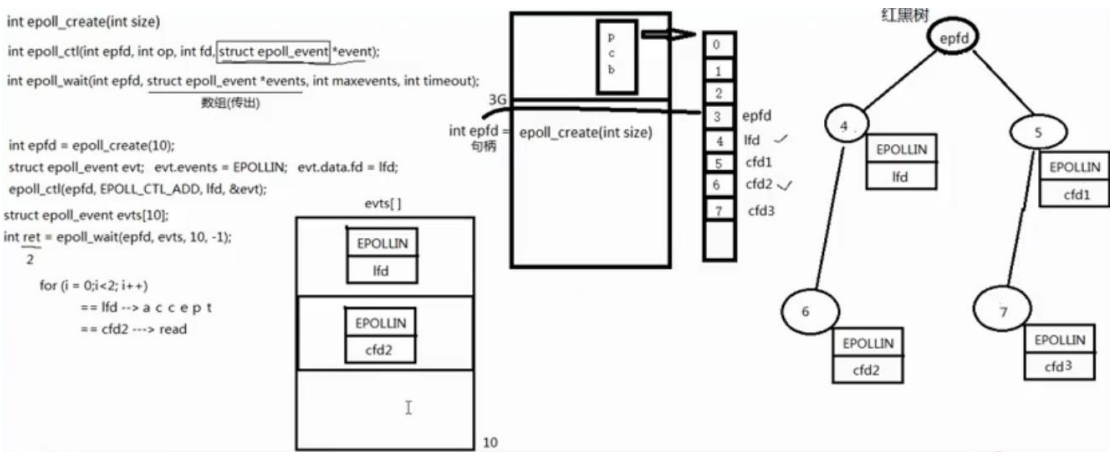
epoll LT：水平触发（默认），该动作的持续期间，你若未处理完，就一直通知你；

因此，使用 fcntl 函数设置 read 的非阻塞读+ epoll ET 模型，效率高！

3. 等待所监控文件描述符上有事件的产生，类似于 select()调用。

```
#include <sys/epoll.h>
int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout)
    events：    用来存内核得到事件的集合，
    maxevents： 告之内核这个 events 有多大，这个 maxevents 的值不能大于创建 epoll_create()时的 size，
    timeout：   是超时时间
        -1：阻塞
        0：  立即返回，非阻塞
        >0：指定毫秒
    返回值：  成功返回有多少文件描述符就绪，时间到时返回 0，出错返回-1
```

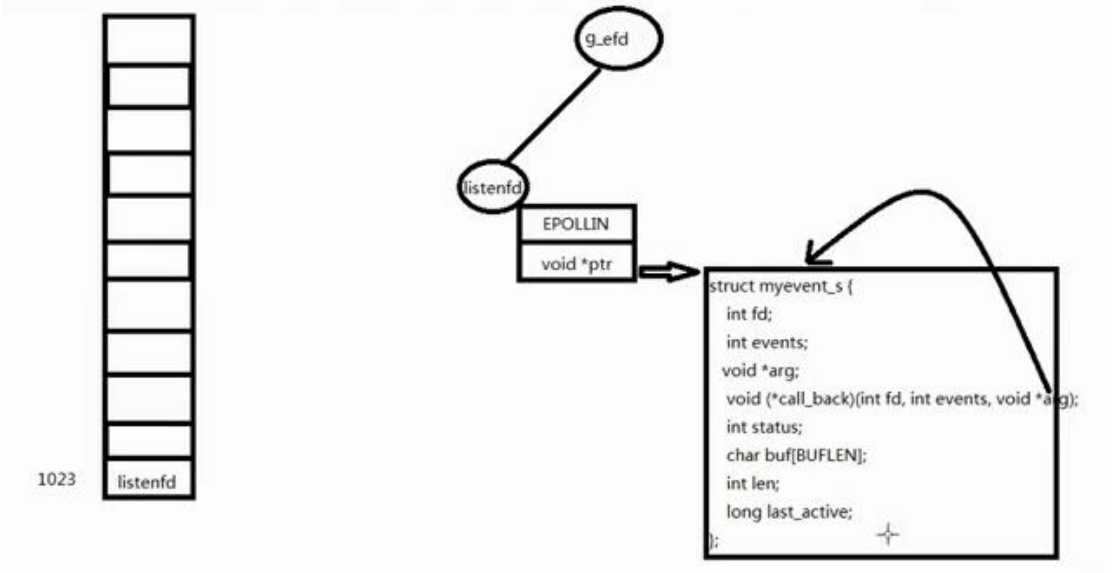
函数返回值为多少，events 数组就会得到几个元素，events 是 struct epoll_ events 类型的



【epoll 反应堆-libevent 库的核心思想】

```
epoll 反应堆模型 (libevent 核心思想实现)

        libevent  -- 跨平台   精炼--epoll  回调

        1. epoll --- 服务器 --- 监听 --- fd ----可读 ---- epoll返回 ---- read --- 小写转大写 --- write ---- epoll继续监听。

        2. epoll 反应堆模型：

        1) epoll --- 服务器 --- 监听 --- cfd ---- 可读 ---- epoll返回 ---- read -- cfd从树上摘下 --- 设置监听cfd写事件， 操作
                                                                                                    ("滑动窗口")
        --- 小写转大写 -- 等待epoll_wait 返回 --- 回写客户端 -- cfd从树上摘下 ----- 设置监听cfd读事件， 操作 -- epoll继续监听。

        2) evt[i].events = EPOLLIN, evt[I].data.fd == cfd       *ptr     struct {int fd, void (*func)(void *arg), void *arg}

struct myevent_s {
    int fd;                                         //要监听的文件描述符
    int events;                                     //对应的监听事件
    void *arg;                                      //泛型参数
    void (*call_back)(int fd, int events, void *arg);   //回调函数
    int status;                                     //是否在监听:1->在红黑树上(监听), 0->不在(不监听)
    char buf[BUFLEN];
    int len;
    long last_active;                               //记录每次加入红黑树 g_efd 的时间值
};
```

解释：假设已经链接，监听读事件，然后都事件发生，服务器读取数据，摘下客户端 cfd 修改为写事件，服务器进行数据的操作，等待写事件设置成功并返回，服务器再回写客户端，再摘下设置 cfd 读事件，继续监听。**(保证了服务器回写数据时，客户端一定是可以接收的！)**
反应堆模型在非阻塞 io 基础上建立。



利用 void *ptr 实现回调函数机制。

【线程池】

内核进行监听+外加线程处理（条件变量、队列、预先创建线程池，再从队列分配任务）