# TODO

ioctl

<https://www.cnblogs.com/tdyizhen1314/p/4896689.html>

# EAGAIN

在linux进行非阻塞的socket接收数据时经常出现Resource temporarily unavailable，errno代码为11(EAGAIN)，这是什么意思？

　这表明你在非阻塞模式下调用了阻塞操作，在该操作没有完成就返回这个错误，这个错误不会破坏socket的同步，不用管它，下次循环接着recv就可以。对非阻塞socket而言，EAGAIN不是一种错误。在VxWorks和Windows上，EAGAIN的名字叫做EWOULDBLOCK。

# send函数MSG\_NOSIGNAL异常消息

服务端自然关闭进程，可是 client 端也竟然出乎意料的关闭掉。

Linux 下当连接断开，还发送数据的时候，不仅 send() 的返回值会有反映，而且还会向系统发送一个异常消息，如果不作处理，系统会出 BrokePipe，程序会退出，这对于服务器提供稳定的服务将造成巨大的灾难。为此，send() 函数的最后一个参数可以设置为 MSG\_NOSIGNAL，禁止 send() 函数向系统发送常消息。

<https://www.cnblogs.com/liwei0526vip/p/5019413.html>

# [epoll](https://www.cnblogs.com/skyfsm/p/7102367.html)

## epoll为何比select好

现在有这么一个场景：我是一个很忙的大老板，我有100个手机，手机来信息了，我的秘书就会告诉我“老板，你的手机来信息了。”我很生气，我的秘书就是这样子，每次手机来信息就只告诉我来信息了，老板赶紧去看。但是她从来不把话说清楚：到底是哪个手机来信息啊！我可有100个手机啊！于是，我只能一个一个手机去查看，来确定到底是哪几个手机来信息了。这就是IO复用中select模型的缺点！老板心想，要是秘书能把来信息的手机直接拿到我桌子上就好了，那么我的效率肯定大增（这就是epoll模型）。

那我们先来总结一下select模型的缺点：

1. 单个进程能够监视的文件描述符的数量存在最大限制，通常是1024，当然可以更改数量，但由于select采用轮询的方式扫描文件描述符，文件描述符数量越多，性能越差；(在linux内核头文件中，有这样的定义：#define \_\_FD\_SETSIZE 1024)
2. 内核 / 用户空间内存拷贝问题，select需要复制大量的句柄数据结构，产生巨大的开销；  
   select返回的是含有整个句柄的数组，应用程序需要遍历整个数组才能发现哪些句柄发生了事件；
3. select的触发方式是水平触发，应用程序如果没有完成对一个已经就绪的文件描述符进行IO操作，那么之后每次select调用还是会将这些文件描述符通知进程。

设想一下如下场景：有100万个客户端同时与一个服务器进程保持着TCP连接。而每一时刻，通常只有几百上千个TCP连接是活跃的(事实上大部分场景都是这种情况)。如何实现这样的高并发？

粗略计算一下，一个进程最多有1024个文件描述符，那么我们需要开1000个进程来处理100万个客户连接。如果我们使用select模型，这1000个进程里某一段时间内只有数个客户连接需要数据的接收，那么我们就不得不轮询1024个文件描述符以确定究竟是哪个客户有数据可读，想想如果1000个进程都有类似的行为，那系统资源消耗可有多大啊！

针对select模型的缺点，epoll模型被提出来了！

## epoll模型的优点

* 支持一个进程打开大数目的socket描述符
* IO效率不随FD数目增加而线性下降
* 使用mmap加速内核与用户空间的消息传递

## epoll的两种工作模式

* LT(level triggered，水平触发模式)是缺省的工作方式，并且同时支持 block 和 non-block socket。在这种做法中，内核告诉你一个文件描述符是否就绪了，然后你可以对这个就绪的fd进行IO操作。如果你不作任何操作，内核还是会继续通知你的，所以，这种模式编程出错误可能性要小一点。比如内核通知你其中一个fd可以读数据了，你赶紧去读。你还是懒懒散散，不去读这个数据，下一次循环的时候内核发现你还没读刚才的数据，就又通知你赶紧把刚才的数据读了。这种机制可以比较好的保证每个数据用户都处理掉了。
* ET(edge-triggered，边缘触发模式)是高速工作方式，只支持no-block socket。在这种模式下，当描述符从未就绪变为就绪时，内核通过epoll告诉你。然后它会假设你知道文件描述符已经就绪，并且不会再为那个文件描述符发送更多的就绪通知，等到下次有新的数据进来的时候才会再次出发就绪事件。简而言之，就是内核通知过的事情不会再说第二遍，数据错过没读，你自己负责。这种机制确实速度提高了，但是风险相伴而行。

## epoll模型API

#include <sys/epoll.h>

/\* 创建一个epoll的句柄，size用来告诉内核需要监听的数目一共有多大。当创建好epoll句柄后，

它就是会占用一个fd值，所以在使用完epoll后，必须调用close()关闭，否则可能导致fd被耗尽。\*/

int epoll\_create(int size);

/\*epoll的事件注册函数\*/

int epoll\_ctl(int epfd, int op, int fd, struct epoll\_event \*event);

/\*等待事件的到来，如果检测到事件，就将所有就绪的事件从内核事件表中复制到它的第二个参数events指向的数组\*/

int epoll\_wait(int epfd, struct epoll\_event \*events, int maxevents, int timeout);

epoll的事件注册函数epoll\_ctl，第一个参数是 epoll\_create() 的返回值，第二个参数表示动作，使用如下三个宏来表示：

POLL\_CTL\_ADD //注册新的fd到epfd中；

EPOLL\_CTL\_MOD //修改已经注册的fd的监听事件；

EPOLL\_CTL\_DEL //从epfd中删除一个fd；

struct epoll\_event 结构如下：

typedef union epoll\_data

{

void \*ptr;

int fd;

\_\_uint32\_t u32;

\_\_uint64\_t u64;

} epoll\_data\_t;

struct epoll\_event

{

\_\_uint32\_t events; /\* Epoll events \*/

epoll\_data\_t data; /\* User data variable \*/

};

epoll\_event结构体中的events 可以是以下几个宏的集合：

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；

EPOLLPRI //表示对应的文件描述符有紧急的数据可读（这里应该表示有带外数据到来）；

EPOLLERR //表示对应的文件描述符发生错误；

EPOLLHUP //表示对应的文件描述符被挂断；

EPOLLET //将EPOLL设为边缘触发(Edge Triggered)模式，这是相对于水平触发(Level Triggered)来说的。

EPOLLONESHOT//只监听一次事件，当监听完这次事件之后，如果还需要继续监听这个socket的话，需要再次把这个socket加入到EPOLL队列里。

## epoll的一个简单使用范例

#include <sys/socket.h>

#include <sys/epoll.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <fcntl.h>

#include <unistd.h>

#include <stdio.h>

#include <errno.h>

#include <stdlib.h>

#include <string.h>

#define MAXLINE 5

#define OPEN\_MAX 100

#define LISTENQ 20

#define SERV\_PORT 5000

#define INFTIM 1000

void setnonblocking(int sock)

{

int opts;

opts=fcntl(sock,F\_GETFL);

if(opts<0)

{

perror("fcntl(sock,GETFL)");

exit(1);

}

opts = opts|O\_NONBLOCK;

if(fcntl(sock,F\_SETFL,opts)<0)

{

perror("fcntl(sock,SETFL,opts)");

exit(1);

}

}

int main(int argc, char\* argv[])

{

int i, maxi, listenfd, connfd, sockfd,epfd,nfds, portnumber;

ssize\_t n;

char line[MAXLINE];

socklen\_t clilen;

if ( 2 == argc )

{

if( (portnumber = atoi(argv[1])) < 0 )

{

fprintf(stderr,"Usage:%s portnumber/a/n",argv[0]);

return 1;

}

}

else

{

fprintf(stderr,"Usage:%s portnumber/a/n",argv[0]);

return 1;

}

//声明epoll\_event结构体的变量,ev用于注册事件,数组用于回传要处理的事件

struct epoll\_event ev,events[20];

//生成用于处理accept的epoll专用的文件描述符

epfd=epoll\_create(256);

struct sockaddr\_in clientaddr;

struct sockaddr\_in serveraddr;

listenfd = socket(AF\_INET, SOCK\_STREAM, 0);

//把socket设置为非阻塞方式

//setnonblocking(listenfd);

//设置与要处理的事件相关的文件描述符

ev.data.fd=listenfd;

//设置要处理的事件类型

ev.events=EPOLLIN|EPOLLET;

//ev.events=EPOLLIN;

//注册epoll事件

epoll\_ctl(epfd,EPOLL\_CTL\_ADD,listenfd,&ev);

bzero(&serveraddr, sizeof(serveraddr));

serveraddr.sin\_family = AF\_INET;

char \*local\_addr="127.0.0.1";

inet\_aton(local\_addr,&(serveraddr.sin\_addr));//htons(portnumber);

serveraddr.sin\_port=htons(portnumber);

bind(listenfd,(struct sockaddr \*)&serveraddr, sizeof(serveraddr));

listen(listenfd, LISTENQ);

maxi = 0;

for ( ; ; ) {

//等待epoll事件的发生

nfds=epoll\_wait(epfd,events,20,500);

//处理所发生的所有事件

for(i=0;i<nfds;++i)

{

if(events[i].data.fd==listenfd)//如果新监测到一个SOCKET用户连接到了绑定的SOCKET端口，建立新的连接。

{

connfd = accept(listenfd,(struct sockaddr \*)&clientaddr, &clilen);

if(connfd<0){

perror("connfd<0");

exit(1);

}

//setnonblocking(connfd);

char \*str = inet\_ntoa(clientaddr.sin\_addr);

printf("accapt a connection from\n ");

//设置用于读操作的文件描述符

ev.data.fd=connfd;

//设置用于注测的读操作事件

ev.events=EPOLLIN|EPOLLET;

//ev.events=EPOLLIN;

//注册ev

epoll\_ctl(epfd,EPOLL\_CTL\_ADD,connfd,&ev);

}

else if(events[i].events&EPOLLIN)//如果是已经连接的用户，并且收到数据，那么进行读入。

{

printf("EPOLLIN\n");

if ( (sockfd = events[i].data.fd) < 0)

continue;

if ( (n = read(sockfd, line, MAXLINE)) < 0) {

if (errno == ECONNRESET) {

close(sockfd);

events[i].data.fd = -1;

} else

printf("readline error\n");

} else if (n == 0) {

close(sockfd);

events[i].data.fd = -1;

}

if(n<MAXLINE-2)

line[n] = '\0';

//设置用于写操作的文件描述符

ev.data.fd=sockfd;

//设置用于注测的写操作事件

ev.events=EPOLLOUT|EPOLLET;

//修改sockfd上要处理的事件为EPOLLOUT

//epoll\_ctl(epfd,EPOLL\_CTL\_MOD,sockfd,&ev);

}

else if(events[i].events&EPOLLOUT) // 如果有数据发送

{

sockfd = events[i].data.fd;

write(sockfd, line, n);

//设置用于读操作的文件描述符

ev.data.fd=sockfd;

//设置用于注测的读操作事件

ev.events=EPOLLIN|EPOLLET;

//修改sockfd上要处理的事件为EPOLIN

epoll\_ctl(epfd,EPOLL\_CTL\_MOD,sockfd,&ev);

}

}

}

return 0;

}

## REF

Linux编程之epoll

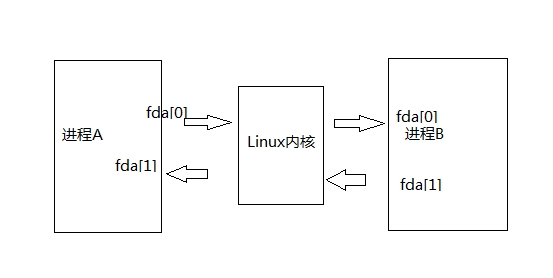
<https://www.cnblogs.com/skyfsm/p/7102367.html>

# pipe()函数

管道是一种把两个进程之间的标准输入和标准输出连接起来的机制，从而提供一种让多个进程间通信的方法，当进程创建管道时，每次

都需要提供两个文件描述符来操作管道。其中一个对管道进行写操作，另一个对管道进行读操作。对管道的读写与一般的IO系统函数一

致，使用write()函数写入数据，使用read()读出数据。



#include<unistd.h>

int pipe(int filedes[2]);

返回值：成功，返回0，否则返回-1。参数数组包含pipe使用的两个文件的描述符。fd[0]:读管道，fd[1]:写管道。

必须在fork()中调用pipe()，否则子进程不会继承文件描述符。两个进程不共享祖先进程，就不能使用pipe。但是可以使用命名管道

## 参考

https://www.cnblogs.com/kunhu/p/3608109.html

# INotify介绍与使用epoll

<http://zhoujinjian.cc/2017/12/01/Android-7-1-2-Android-N-Android-%E8%BE%93%E5%85%A5%E5%AD%90%E7%B3%BB%E7%BB%9F-Input-System/index.html>

## （二）、必备Linux知识\_双向通信(scoketpair)

<http://zhoujinjian.cc/2017/12/01/Android-7-1-2-Android-N-Android-%E8%BE%93%E5%85%A5%E5%AD%90%E7%B3%BB%E7%BB%9F-Input-System/index.html>

## 必备Linux知识\_实现任意进程间双向通信(scoketpair+binder)