# TODO

ioctl

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

# select/poll/epoll对比分析

*elect/poll/epoll都是IO多路复用机制，可以同时监控多个描述符，当某个描述符就绪(读或写就绪)，则立刻通知相应程序进行读或写操作。本质上select/poll/epoll都是同步I/O，即读写是阻塞的。*

## 一、select

原型：

int select (int maxfd, fd\_set \*readfds,

fd\_set \*writefds,

fd\_set \*exceptfds,

struct timeval \*timeout);

* maxfd：代表要监控的最大文件描述符fd+1
* writefds：监控可写fd
* readfds：监控可读fd
* exceptfds：监控异常fd
* timeout：超时时长
  + NULL，代表没有设置超时，则会一直阻塞直到文件描述符上的事件触发
  + 0，代表不等待，立即返回，用于检测文件描述符状态
  + 正整数，代表当指定时间没有事件触发，则超时返回

select函数监控3类文件描述符，调用select函数后会阻塞，直到描述符fd准备就绪（有数据可读、可写、异常）或者超时，函数便返回。 当select函数返回后，可通过遍历描述符集合，找到就绪的描述符。

**select缺点**

* 文件描述符个数受限：单进程能够监控的文件描述符的数量存在最大限制，在Linux上一般为1024，可以通过修改宏定义增大上限，但同样存在效率低的弱势;
* 性能衰减严重：IO随着监控的描述符数量增长，其性能会线性下降;

## 二、poll

原型：

int poll (struct pollfd \*fds, unsigned int nfds, int timeout);

其中pollfd表示监视的描述符集合，如下

struct pollfd {

int fd; //文件描述符

short events; //监视的请求事件

short revents; //已发生的事件

};

pollfd结构包含了要监视的event和发生的event，并且pollfd并没有最大数量限制。 和select函数一样，当poll函数返回后，可以通过遍历描述符集合，找到就绪的描述符。

**poll缺点：**从上面看select和poll都需要在返回后，通过遍历文件描述符来获取已经就绪的socket。同时连接的大量客户端在同一时刻可能只有很少的处于就绪状态，因此随着监视的描述符数量的增长，其性能会线性下降。

## 三、epoll

epoll是在内核2.6中提出的，是select和poll的增强版。相对于select和poll来说，epoll更加灵活，没有描述符数量限制。epoll使用一个文件描述符管理多个描述符，将用户空间的文件描述符的事件存放到内核的一个事件表中，这样在用户空间和内核空间的copy只需一次。epoll机制是Linux最高效的I/O复用机制，在一处等待多个文件句柄的I/O事件。

select/poll都只有一个方法，epoll操作过程有3个方法，分别是epoll\_create()， epoll\_ctl()，epoll\_wait()。

#### 3.1 epoll\_create

int epoll\_create(int size)；

功能：用于创建一个epoll的句柄，size是指监听的描述符个数， 现在内核支持动态扩展，该值的意义仅仅是初次分配的fd个数，后面空间不够时会动态扩容。 当创建完epoll句柄后，占用一个fd值.

ls /proc/<pid>/fd/ //可通过终端执行，看到该fd

使用完epoll后，必须调用close()关闭，否则可能导致fd被耗尽。

#### 3.2 epoll\_ctl

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

功能：用于对需要监听的文件描述符(fd)执行op操作，比如将fd加入到epoll句柄。

* epfd：是epoll\_create()的返回值；
* op：表示op操作，用三个宏来表示，分别代表添加、删除和修改对fd的监听事件；
  + EPOLL\_CTL\_ADD(添加)
  + EPOLL\_CTL\_DEL(删除)
  + EPOLL\_CTL\_MOD（修改）
* fd：需要监听的文件描述符；
* epoll\_event：需要监听的事件，struct epoll\_event结构如下：
* struct epoll\_event {
* \_\_uint32\_t events; /\* Epoll事件 \*/
* epoll\_data\_t data; /\*用户可用数据\*/
* };

events可取值：(表示对应的文件描述符的操作)

* + EPOLLIN ：可读（包括对端SOCKET正常关闭）；
  + EPOLLOUT：可写；
  + EPOLLERR：错误；
  + EPOLLHUP：中断；
  + EPOLLPRI：高优先级的可读（这里应该表示有带外数据到来）；
  + EPOLLET： 将EPOLL设为边缘触发模式，这是相对于水平触发来说的。
  + EPOLLONESHOT：只监听一次事件，当监听完这次事件之后就不再监听该事件

#### 3.3 epoll\_wait

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

功能：等待事件的上报

* epfd：等待epfd上的io事件，最多返回maxevents个事件；
* events：用来从内核得到事件的集合；
* maxevents：events数量，该maxevents值不能大于创建epoll\_create()时的size；
* timeout：超时时间（毫秒，0会立即返回）。

该函数返回需要处理的事件数目，如返回0表示已超时。

## 四、对比

1）在 select/poll中，进程只有在调用一定的方法后，内核才对所有监视的文件描述符进行扫描，而epoll事先通过epoll\_ctl()来注册一个文件描述符，一旦基于某个文件描述符就绪时，内核会采用类似callback的回调机制，迅速激活这个文件描述符，当进程调用epoll\_wait() 时便得到通知。(此处去掉了遍历文件描述符，而是通过监听回调的的机制。这正是epoll的魅力所在。)

**epoll优势**

1. 监视的描述符数量不受限制，所支持的FD上限是最大可以打开文件的数目，具体数目可以cat /proc/sys/fs/file-max查看，一般来说这个数目和系统内存关系很大，以3G的手机来说这个值为20-30万。

marlin:/ # cat proc/sys/fs/file-max

355,521

1. IO性能不会随着监视fd的数量增长而下降。epoll不同于select和poll轮询的方式，而是通过每个fd定义的回调函数来实现的，只有就绪的fd才会执行回调函数。

如果没有大量的空闲或者死亡连接，epoll的效率并不会比select/poll高很多。但当遇到大量的空闲连接的场景下，epoll的效率大大高于select/poll。

## 参考

select/poll/epoll对比分析

http://gityuan.com/2015/12/06/linux\_epoll/

# 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>

# 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)