epoll的接口非常简单，一共就三个函数：

### epoll\_create

int epfd = epoll\_create(intsize);

创建epoll句柄

创建一个epoll的句柄，size用来告诉内核这个监听的数目一共有多大。这个参数不同于select()中的第一个参数，给出最大监听的fd+1的值。需要注意的是，当创建好epoll句柄后，它就是会占用一个fd值，在linux下如果查看/proc/进程id/fd/，是能够看到这个fd的，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。

函数声明：int epoll\_create(int size)

该 函数生成一个epoll专用的文件描述符。它其实是在内核申请一空间，用来存放你想关注的socket fd上是否发生以及发生了什么事件。size就是你在这个epoll fd上能关注的最大socket fd数。随你定好了。只要你有空间。可参见上面与select之不同

### epoll\_ctl

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

将被监听的描述符添加到epoll句柄或从epool句柄中删除或者对监听事件进行修改。

该函数用于控制某个epoll文件描述符上的事件，可以注册事件，修改事件，删除事件。

参数：

epfd：由 epoll\_create 生成的epoll专用的文件描述符；

op：要进行的操作例如注册事件，可能的取值EPOLL\_CTL\_ADD 注册、EPOLL\_CTL\_MOD 修改、EPOLL\_CTL\_DEL 删除

fd：关联的文件描述符；

event：指向epoll\_event的指针；

如果调用成功返回0,不成功返回-1

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

epoll的事件注册函数，它不同与select()是在监听事件时告诉内核要监听什么类型的事件，而是在这里先注册要监听的事件类型。

第一个参数是epoll\_create()的返回值，

第二个参数表示动作，用三个宏来表示：

EPOLL\_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；

第三个参数是需要监听的fd，

第四个参数是告诉内核需要监听什么事件，structepoll\_event结构如下：

[cpp] view plain copy

1. typedef union epoll\_data {

2. void \*ptr;

3. int fd;

4. \_\_uint32\_t u32;

5. \_\_uint64\_t u64;

6. } epoll\_data\_t;

7.

8. struct epoll\_event {

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

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

11. };

events可以是以下几个宏的集合：

EPOLLIN：触发该事件，表示对应的文件描述符上有可读数据。(包括对端SOCKET正常关闭)；

EPOLLOUT：触发该事件，表示对应的文件描述符上可以写数据；

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。

如：

struct epoll\_event ev; //设置与要处理的事件相关的文件描述符

ev.data.fd=listenfd;

ev.events=EPOLLIN|EPOLLET; //设置要处理的事件类型

//注册epoll事件

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

### epoll\_wait

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

等待事件触发，当超过timeout还没有事件触发时，就超时。

等待事件的产生，类似于select()调用。参数events用来从内核得到事件的集合，maxevents告之内核这个events有多大(数组成员的个数)，这个maxevents的值不能大于创建epoll\_create()时的size，参数timeout是超时时间（毫秒，0会立即返回; -1将不确定，也有说法说是永久阻塞）。

该函数返回需要处理的事件数目，如返回0表示已超时。

返回的事件集合在events数组中，数组中实际存放的成员个数是函数的返回值。返回0表示已经超时。

函数声明:int epoll\_wait(int epfd,struct epoll\_event \* events,int maxevents,int timeout)

该函数用于轮询I/O事件的发生；

参数：

epfd:由epoll\_create 生成的epoll专用的文件描述符；

epoll\_event:用于回传代处理事件的数组；

maxevents:每次能处理的事件数；

timeout:等待I/O事件发生的超时值(单位我也不太清楚)；-1相当于阻塞，0相当于非阻塞。一般用-1即可

返回发生事件数。

epoll\_wait运行的原理是

等侍注册在epfd上的socket fd的事件的发生，如果发生则将发生的sokct fd和事件类型放入到events数组中。并且将注册在epfd上的socket fd的事件类型给清空，所以如果下一个循环你还要关注这个socket fd的话，则需要用epoll\_ctl(epfd,EPOLL\_CTL\_MOD,listenfd,&ev)来重新设置socket fd的事件类型。这时不用EPOLL\_CTL\_ADD,因为socket fd并未清空，只是事件类型清空。这一步非常重要。

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

从man手册中，得到ET和LT的具体描述如下

EPOLL事件有两种模型：

Edge Triggered(ET) //高速工作方式，错误率比较大，只支持no\_block socket (非阻塞socket)

LevelTriggered(LT) //缺省工作方式，即默认的工作方式,支持blocksocket和no\_blocksocket，错误率比较小。

假如有这样一个例子：(LT方式，即默认方式下，内核会继续通知，可以读数据，ET方式，内核不会再通知，可以读数据)

1.我们已经把一个用来从管道中读取数据的文件句柄(RFD)添加到epoll描述符

2. 这个时候从管道的另一端被写入了2KB的数据

3. 调用epoll\_wait(2)，并且它会返回RFD，说明它已经准备好读取操作

4. 然后我们读取了1KB的数据

5. 调用epoll\_wait(2)......

Edge Triggered工作模式：

如果我们在第1步将RFD添加到epoll描述符的时候使用了EPOLLET标志，那么在第5步调用epoll\_wait(2)之后将有可能会挂起，因为剩余的数据还存在于文件的输入缓冲区内，而且数据发出端还在等待一个针对已经发出数据的反馈信息。只有在监视的文件句柄上发生了某个事件的时候ET工作模式才会汇报事件。因此在第5步的时候，调用者可能会放弃等待仍在存在于文件输入缓冲区内的剩余数据。在上面的例子中，会有一个事件产生在RFD句柄上，因为在第2步执行了一个写操作，然后，事件将会在第3步被销毁。因为第4步的读取操作没有读空文件输入缓冲区内的数据，因此我们在第5步调用epoll\_wait(2)完成后，是否挂起是不确定的。epoll工作在ET模式的时候，必须使用非阻塞套接口，以避免由于一个文件句柄的阻塞读/阻塞写操作把处理多个文件描述符的任务饿死。最好以下面的方式调用ET模式的epoll接口，在后面会介绍避免可能的缺陷。(LT方式可以解决这种缺陷)

i 基于非阻塞文件句柄

ii 只有当read(2)或者write(2)返回EAGAIN时(认为读完)才需要挂起，等待。但这并不是说每次read()时都需要循环读，直到读到产生一个EAGAIN才认为此次事件处理完成，当read()返回的读到的数据长度小于请求的数据长度时(即小于sizeof(buf))，就可以确定此时缓冲中已没有数据了，也就可以认为此事读事件已处理完成。

Level Triggered工作模式 (默认的工作方式)

相反的，以LT方式调用epoll接口的时候，它就相当于一个速度比较快的poll(2)，并且无论后面的数据是否被使用，因此他们具有同样的职能。因为即使使用ET模式的epoll，在收到多个chunk的数据的时候仍然会产生多个事件。调用者可以设定EPOLLONESHOT标志，在epoll\_wait(2)收到事件后epoll会与事件关联的文件句柄从epoll描述符中禁止掉。因此当EPOLLONESHOT设定后，使用带有EPOLL\_CTL\_MOD标志的epoll\_ctl(2)处理文件句柄就成为调用者必须作的事情。

然后详细解释ET, LT:

//没有对就绪的fd进行IO操作，内核会不断的通知。

LT(leveltriggered)是缺省的工作方式，并且同时支持block和no-blocksocket。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。传统的select/poll都是这种模型的代表。

//没有对就绪的fd进行IO操作，内核不会再进行通知。

ET(edge-triggered)是高速工作方式，只支持no-blocksocket。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，直到你做了某些操作导致那个文件描述符不再为就绪状态了(比如，你在发送，接收或者接收请求，或者发送接收的数据少于一定量时导致了一个EWOULDBLOCK错误）。但是请注意，如果一直不对这个fd作IO操作(从而导致它再次变成未就绪)，内核不会发送更多的通知(only once),不过在TCP协议中，ET模式的加速效用仍需要更多的benchmark确认（这句话不理解）。

另外，当使用epoll的ET模型(epoll的非默认工作方式)来工作时，当产生了一个EPOLLIN事件后，

读数据的时候需要考虑的是当recv()返回的大小如果等于要求的大小，即sizeof(buf)，那么很有可能是缓冲区还有数据未读完，也意味着该次事件还没有处理完，所以还需要再次读取：

while(rs) //ET模型

{

buflen = recv(activeevents[i].data.fd, buf, sizeof(buf), 0);

if(buflen < 0) {

//由于是非阻塞的模式,所以当errno为EAGAIN时,表示当前缓冲区已无数据可读

// 在这里就当作是该次事件已处理处.

if(errno== EAGAIN || errno == EINT) //即当buflen<0且errno=EAGAIN时，表示没有数据了。(读/写都是这样)

break;

else

return; //真的失败了。

}

elseif(buflen == 0) {

//这里表示对端的socket已正常关闭.

}

if(buflen== sizeof(buf)

rs = 1; //需要再次读取(有可能是因为数据缓冲区buf太小，所以数据没有读完)

else

rs =0; //不需要再次读取(当buflen

}

非常重要::

还有，假如发送端流量大于接收端的流量(意思是epoll所在的程序读比转发的socket要快),由于是非阻塞的socket,那么send()函数虽然返回,但实际缓冲区的数据并未真正发给接收端,这样不断的读和发，当缓冲区满后会产生EAGAIN错误(参考mansend),同时,不理会这次请求发送的数据.所以,

需要封装socket\_send()的函数用来处理这种情况,该函数会尽量将数据写完再返回，返回-1表示出错。在socket\_send()内部,当写缓冲已满(send()返回-1,且errno为EAGAIN),那么会等待后再重试.这种方式并不很完美,在理论上可能会长时间的阻塞在socket\_send()内部,但暂没有更好的办法.

这种方法类似于readn和writen的封装(自己写过，在《UNIX环境高级编程》中也有介绍)

ssize\_t socket\_send(int sockfd, const char\* buffer, size\_tbuflen)

{

ssize\_t tmp;

size\_t total = buflen;

const char \*p = buffer;

while(1)

{

tmp =send(sockfd, p, total, 0);

if(tmp <0)

{

// 当send收到信号时,可以继续写,但这里返回-1.

if(errno == EINTR)

return -1;

//当socket是非阻塞时,如返回此错误,表示写缓冲队列已满,

//在这里做延时后再重试.

if(errno == EAGAIN)

{

usleep(1000);

continue;

}

return -1;

}

if((size\_t)tmp == total)

return buflen;

total -=tmp;

p +=tmp;

}

return tmp;

}

总结：

[cpp] view plain copy

1. man中给出了epoll的用法，example程序如下：

2. for(;;) {

3. nfds = epoll\_wait(kdpfd, events, maxevents, -1);

4.

5. for(n = 0; n < nfds; ++n) {

6. if(events[n].data.fd == listener) {

7. client = accept(listener, (struct sockaddr \*) &local,

8. &addrlen);

9. if(client < 0){

10. perror("accept");

11. continue;

12. }

13. setnonblocking(client);

14. ev.events = EPOLLIN | EPOLLET;

15. ev.data.fd = client;

16. if (epoll\_ctl(kdpfd, EPOLL\_CTL\_ADD, client, &ev) < 0) {

17. fprintf(stderr, "epoll set insertion error: fd=%d\n",

18. client);

19. return -1;

20. }

21. }

22. else

23. do\_use\_fd(events[n].data.fd);

24. }

25. }

此时使用的是ET模式，即，边沿触发，类似于电平触发，epoll中的边沿触发的意思是只对新到的数据进行通知，而内核缓冲区中如果是旧数据则不进行通知，所以在do\_use\_fd函数中应该使用如下循环，才能将内核缓冲区中的数据读完。

[cpp] view plain copy

1. while (1) {

2. len = recv(\*\*\*\*\*\*\*);

3. if (len == -1) {

4. if(errno == EAGAIN)

5. break;

6. perror("recv");

7. break;

8. }

9. do something with the recved data........

10. }

在上面例子中没有说明对于listen socket fd该如何处理，有的时候会使用两个线程，一个用来监听accept另一个用来监听epoll\_wait，如果是这样使用的话，则listen socket fd使用默认的阻塞方式就行了，而如果epoll\_wait和accept处于一个线程中，即，全部由epoll\_wait进行监听，则，需将listen socket fd也设置成非阻塞的，这样，对accept也应该使用while包起来（类似于上面的recv），因为，epoll\_wait返回时只是说有连接到来了，并没有说有几个连接，而且在ET模式下epoll\_wait不会再因为上一次的连接还没读完而返回，这种情况确实存在，我因为这个问题而耗费了一天多的时间，这里需要说明的是，每调用一次accept将从内核中的已连接队列中的队头读取一个连接，因为在并发访问的环境下，有可能有多个连接“同时”到达，而epoll\_wait只返回了一次。