<https://blog.csdn.net/weiyuefei/article/details/52242880>

**5.1 ET模式下的读写**

    经过前面几节分析，我们可以知道，当epoll工作在ET模式下时，对于读操作，如果read一次没有读尽buffer中的数据，那么下次将得不到读就绪的通知，造成**buffer中已有的数据无机会读出，除非有新的数据再次到达**。对于写操作，主要是因为ET模式下fd通常为非阻塞造成的一个问题——**如何保证将用户要求写的数据写完。**

要解决上述两个ET模式下的读写问题，我们必须实现：

**a. 对于读，只要buffer中还有数据就一直读；**

**b. 对于写，只要buffer还有空间且用户请求写的数据还未写完，就一直写。**

要实现上述a、b两个效果，我们有两种方法解决。

方法一

(1) 每次读入操作后（read，recv），用户主动epoll\_mod IN事件，此时只要该fd的缓冲还有数据可以读，则epoll\_wait会返回读就绪。

(2) 每次输出操作后（write，send），用户主动epoll\_mod OUT事件，此时只要该该fd的缓冲可以发送数据（发送buffer不满），则epoll\_wait就会返回写就绪（有时候采用该机制通知epoll\_wai醒过来）。

这个方法的原理我们在之前讨论过：**当buffer中有数据可读（即buffer不空）且用户对相应fd进行epoll\_mod IN事件时ET模式返回读就绪，当buffer中有可写空间（即buffer不满）且用户对相应fd进行epoll\_mod OUT事件时返回写就绪。**

所以得到如下解决方式：

if(events[i].events&EPOLLIN)//如果收到数据，那么进行读入

{

    cout << "EPOLLIN" << endl;

    sockfd = events[i].data.fd;

    if ( (n = read(sockfd, line, MAXLINE))>0)

{

line[n] = '/0';

        cout << "read " << line << endl;

**if(n==MAXLINE)**

{

ev.data.fd=sockfd;

ev.events=EPOLLIN|EPOLLET;

epoll\_ctl(epfd,EPOLL\_CTL\_MOD,sockfd,&ev); //数据还没读完，重新MOD IN事件

}

else

{

ev.data.fd=sockfd;

ev.events=EPOLLIN|EPOLLET;

epoll\_ctl(epfd,EPOLL\_CTL\_MOD,sockfd,&ev); //buffer中的数据已经读取完毕MOD OUT事件

}

}

 else if (n == 0)

{

close(sockfd);

    }

}

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

{

    sockfd = events[i].data.fd;

    write(sockfd, line, n);

    ev.data.fd=sockfd; //设置用于读操作的文件描述符

    ev.events=EPOLLIN|EPOLLET; //设置用于注测的读操作事件

    epoll\_ctl(epfd,EPOLL\_CTL\_MOD,sockfd,&ev);  //修改sockfd上要处理的事件为EPOLIN

}

注：对于write操作，由于sockfd是工作在阻塞模式下的，所以没有必要进行特殊处理，和LT使用一样。

分析：这种方法存在几个问题：

(1) 对于read操作后的判断——**if(n==MAXLINE)，**不能说明这种情况buffer就一定还有没有读完的数据，试想万一buffer中一共就有MAXLINE字节数据呢？这样继续 MOD IN就不再得到通知，而也就没有机会对相应sockfd MOD OUT。

(2) 那么如果服务端用其他方式能够在适当时机对相应的sockfd MOD OUT,是否这种方法就可取呢？我们首先思考一下为什么要用ET模式，因为ET模式能够减少epoll\_wait等系统调用，而我们在这里每次read后都要MOD IN,之后又要epoll\_wait，势必造成效率降低，这不是适得其反吗？

综上，此方式不应该使用。

方法二

**读: 只要可读, 就一直读, 直到返回 0, 或者 errno = EAGAIN**

**写: 只要可写, 就一直写, 直到数据发送完, 或者 errno = EAGAIN**

if (events[i].events & EPOLLIN)

  {

  n = 0;

**while ((nread = read(fd, buf + n, BUFSIZ-1)) > 0)**

**{**

**n += nread;**

**}**

 if (nread == -1 && errno != EAGAIN)

  {

 perror("read error");

      }

      ev.data.fd = fd;

      ev.events = events[i].events | EPOLLOUT;

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

  }

  if (events[i].events & EPOLLOUT)

  {

  int nwrite, data\_size = strlen(buf);

      n = data\_size;

      while (n > 0)

  {

**nwrite = write(fd, buf + data\_size - n, n);**

**if (nwrite < n)**

**{**

**if (nwrite == -1 && errno != EAGAIN)**

**{**

**perror("write error");**

**}**

**break;**

**}**

**n -= nwrite;**

        }

    ev.data.fd=fd;

    ev.events=EPOLLIN|EPOLLET;

    epoll\_ctl(epfd,EPOLL\_CTL\_MOD,fd,&ev);  //修改sockfd上要处理的事件为EPOLIN

 }

注：使用这种方式一定要使每个连接的套接字工作于**非阻塞模式**，**因为读写需要一直读或写直到出错（对于读，当读到的实际字节数小于请求字节数时就可以停止），而如果你的文件描述符如果不是非阻塞的，那这个一直读或一直写势必会在最后一次阻塞。这样就不能在阻塞在epoll\_wait上了，造成其他文件描述符的任务饿死。**

**综上：方法一不适合使用，我们只能使用方法二，所以也就常说“ET需要工作在非阻塞模式”,当然这并不能说明ET不能工作在阻塞模式，而是工作在阻塞模式可能在运行中会出现一些问题。**

**方法三**

仔细分析方法二的写操作，我们发现这种方式并不很完美,因为写操作返回EAGAIN就终止写，但是返回EAGAIN只能说名当前buffer已满不可写，并不能保证用户（或服务端）要求写的数据已经写完。**那么如何保证对非阻塞的套接字写够请求的字节数才返回呢**（阻塞的套接字直到将请求写的字节数写完才返回）?

我们需要封装socket\_write()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\_write()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.

ssize\_t socket\_write(int sockfd, const char\* buffer, size\_t buflen)

{

  ssize\_t tmp;

  size\_t total = buflen;

  const char\* p = buffer;

  while(1)

  {

    tmp = write(sockfd, p, total);

    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;//返回已写字节数

}

分析：这种方式也存在问题，因为在理论上可能会长时间的阻塞在socket\_write()内部（buffer中的数据得不到发送，一直返回EAGAIN）,但暂没有更好的办法。

**不过看到这种方式时，我在想在socket\_write中将sockfd改为阻塞模式应该一样可行，等再次epoll\_wait之前再将其改为非阻塞。**

**5.2 ET模式下的accept**

    考虑这种情况：多个连接同时到达，服务器的 TCP 就绪队列瞬间积累多个就绪

连接，由于是边缘触发模式，epoll 只会通知一次，accept 只处理一个连接，导致 TCP 就绪队列中剩下的连接都得不到处理。

**解决办法是用 while 循环抱住 accept 调用，处理完 TCP 就绪队列中的所有连接后再退出循环。**如何知道是否处理完就绪队列中的所有连接呢？ accept  返回 -1 并且 errno 设置为 EAGAIN 就表示所有连接都处理完。

的正确使用方式为：

while ((conn\_sock = accept(listenfd,(struct sockaddr \*) &remote, (size\_t \*)&addrlen)) > 0) {

    handle\_client(conn\_sock);

}

if (conn\_sock == -1) {

     if (errno != EAGAIN && errno != ECONNABORTED

            && errno != EPROTO && errno != EINTR)

        perror("accept");

}

**扩展**：**服务端使用多路转接技术（select，poll，epoll等）时，accept应工作在非阻塞模式。**

原因：如果accept工作在阻塞模式，考虑这种情况： TCP 连接被客户端夭折，即在服务器调用 accept 之前（此时select等已经返回连接到达读就绪），客户端主动发送 RST 终止连接，导致刚刚建立的连接从就绪队列中移出，如果套接口被设置成阻塞模式，服务器就会一直阻塞在 accept 调用上，直到其他某个客户建立一个新的连接为止。但是在此期间，服务器单纯地阻塞在accept 调用上（实际应该阻塞在select上），就绪队列中的其他描述符都得不到处理。

    解决办法是把监听套接口设置为非阻塞， 当客户在服务器调用 accept 之前中止

某个连接时，accept 调用可以立即返回 -1， 这时源自 Berkeley 的实现会在内核中处理该事件，并不会将该事件通知给 epoll，而其他实现把 errno 设置为 ECONNABORTED 或者 EPROTO 错误，我们应该忽略这两个错误。（具体可参看UNP v1 p363）