<http://blog.csdn.net/hnlyyk/article/details/48974749>

版权声明：【博主微信公众号：不忘初心的行者】【本文为博主原创,未经博主允许不得转载】 http://blog.csdn.net/u010193457/article/details/48974749

目录[(?)[+]](http://blog.csdn.net/hnlyyk/article/details/48974749)

## 1.简介

Linux I/O多路复用技术在比较多的TCP网络服务器中有使用，即比较多的用到select函数。Linux 2.6内核中有提高网络I/O性能的新方法，即epoll 。   
epoll是什么？按照man手册的说法是为处理大批量句柄而作了改进的poll。要使用epoll只需要以下的三个系统函数调用： epoll\_create(2)，epoll\_ctl(2)，epoll\_wait(2)。

## 2.select模型的缺陷

(1) 在Linux内核中，select所用到的FD\_SET是有限的   
内核中有个参数\_\_FD\_SETSIZE定义了每个FD\_SET的句柄个数：#define \_\_FD\_SETSIZE 1024。也就是说，如果想要同时检测1025个句柄的可读状态是不可能用select实现的；或者同时检测1025个句柄的可写状态也是不可能的。   
(2) 内核中实现select是使用轮询方法   
每次检测都会遍历所有FD\_SET中的句柄，显然select函数的执行时间与FD\_SET中句柄的个数有一个比例关系，即select要检测的句柄数越多就会越费时

## 3.Windows IOCP模型的缺陷

windows完成端口实现的AIO，实际上也只是使用内部用线程池实现的，最后的结果是IO有个线程池，你的应用程序也需要一个线程池。很多文档其实已经指出了这引发的线程context-switch所带来的代价。

## 4.EPOLL模型的优点

(1) 支持一个进程打开大数目的socket描述符(FD)   
epoll没有select模型中的限制，它所支持的FD上限是最大可以打开文件的数目，这个数字一般远大于select 所支持的2048。下面是我的小PC机上的显示：   
pt@ubuntu:~$ cat /proc/sys/fs/file-max   
6815744   
那么对于服务器而言,这个数目会更大。   
(2) IO效率不随FD数目增加而线性下降   
传统select/poll的另一个致命弱点就是当你拥有一个很大的socket集合，由于网络得延时，使得任一时间只有部分的socket是”活跃”的，而select/poll每次调用都会线性扫描全部的集合，导致效率呈现线性下降。但是epoll不存在这个问题，它只会对”活跃”的socket进行操作：这是因为在内核实现中epoll是根据每个fd上面的callback函数实现的。于是，只有”活跃”的socket才会主动去调用callback函数，其他idle状态的socket则不会。在这点上，epoll实现了一个”伪”AIO”，因为这时候推动力在os内核。在一些 benchmark中，如果所有的socket基本上都是活跃的，比如一个高速LAN环境，epoll也不比select/poll低多少效率，但若过多使用的调用epoll\_ctl，效率稍微有些下降。然而一旦使用idle connections模拟WAN环境，那么epoll的效率就远在select/poll之上了。   
(3) 使用mmap加速内核与用户空间的消息传递   
无论是select,poll还是epoll都需要内核把FD消息通知给用户空间，如何避免不必要的内存拷贝就显得很重要。在这点上，epoll是通过内核于用户空间mmap同一块内存实现。

## 5.EPOLL模型的工作模式

(1) LT模式   
LT：level triggered，这是缺省的工作方式，同时支持block和no-block socket，在这种模式中，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。   
(2) ET模式   
LT：edge-triggered，这是高速工作方式，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，内核就通过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而导致那个文件描述符不再是就绪状态(比如你在发送，接收或是接受请求，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)，内核就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。

## 6.EPOLL模型的使用方法

epoll用到的所有函数都是在头文件sys/epoll.h中声明的，下面简要说明所用到的数据结构和函数：   
(1) epoll\_data、epoll\_data\_t、epoll\_event   
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 \*/   
};

结构体epoll\_event 被用于注册所感兴趣的事件和回传所发生待处理的事件。epoll\_event 结构体的events字段是表示感兴趣的事件和被触发的事件，可能的取值为：   
EPOLLIN： 表示对应的文件描述符可以读；   
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；   
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： 表示对应的文件描述符有事件发生；

联合体epoll\_data用来保存触发事件的某个文件描述符相关的数据。例如一个client连接到服务器，服务器通过调用accept函数可以得到于这个client对应的socket文件描述符，可以把这文件描述符赋给epoll\_data的fd字段，以便后面的读写操作在这个文件描述符上进行。

(2)epoll\_create   
函数声明：intepoll\_create(intsize)   
函数说明：该函数生成一个epoll专用的文件描述符，其中的参数是指定生成描述符的最大范围。

(3) epoll\_ctl函数   
函数声明：intepoll\_ctl(int epfd,int op, int fd, struct epoll\_event \*event)   
函数说明：该函数用于控制某个文件描述符上的事件，可以注册事件、修改事件、删除事件。   
epfd：由 epoll\_create 生成的epoll专用的文件描述符；   
op：要进行的操作，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；   
如果调用成功则返回0，不成功则返回-1。

(4) epoll\_wait函数   
函数声明：int epoll\_wait(int epfd, structepoll\_event \* events, int maxevents, int timeout)   
函数说明：该函数用于轮询I/O事件的发生。   
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；   
返回发生事件数。

## 7 设计思路及模板

首先通过create\_epoll(int maxfds)来创建一个epoll的句柄，其中maxfds为你的epoll所支持的最大句柄数。这个函数会返回一个新的epoll句柄，之后的所有操作都将通过这个句柄来进行操作。在用完之后，记得用close()来关闭这个创建出来的epoll句柄。   
然后在你的网络主循环里面，调用epoll\_wait(int epfd, epoll\_event events, int max\_events,int timeout)来查询所有的网络接口，看哪一个可以读，哪一个可以写。基本的语法为：   
nfds = epoll\_wait(kdpfd, events, maxevents, -1);   
其中kdpfd为用epoll\_create创建之后的句柄，events是一个epoll\_event\*的指针，当epoll\_wait函数操作成功之后，events里面将储存所有的读写事件。max\_events是当前需要监听的所有socket句柄数。最后一个timeout参数指示 epoll\_wait的超时条件，为0时表示马上返回；为-1时表示函数会一直等下去直到有事件返回；为任意正整数时表示等这么长的时间，如果一直没有事件，则会返回。一般情况下如果网络主循环是单线程的话，可以用-1来等待，这样可以保证一些效率，如果是和主循环在同一个线程的话，则可以用0来保证主循环的效率。epoll\_wait返回之后，应该进入一个循环，以便遍历所有的事件。   
对epoll 的操作就这么简单，总共不过4个API：epoll\_create, epoll\_ctl,epoll\_wait和close。以下是man中的一个例子。

struct epoll\_event ev, \*events;

for(;;)

{

nfds = epoll\_wait(kdpfd, events, maxevents, -1); //等待IO事件

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

{

//如果是主socket的事件，则表示有新连接进入，需要进行新连接的处理。

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

{

client = accept(listener, (struct sockaddr \*) &local, &addrlen);

if(client < 0)

{

perror("accept error");

continue;

}

// 将新连接置于非阻塞模式

setnonblocking(client);

ev.events = EPOLLIN | EPOLLET;

//注意这里的参数EPOLLIN | EPOLLET并没有设置对写socket的监听，

//如果有写操作的话，这个时候epoll是不会返回事件的，

//如果要对写操作也监听的话，应该是EPOLLIN | EPOLLOUT | EPOLLET。

// 并且将新连接也加入EPOLL的监听队列

ev.data.fd = client;

// 设置好event之后，将这个新的event通过epoll\_ctl

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

{

//加入到epoll的监听队列里，这里用EPOLL\_CTL\_ADD

//来加一个新的 epoll事件。可以通过EPOLL\_CTL\_DEL来减少

//一个epoll事件，通过EPOLL\_CTL\_MOD来改变一个事件的监听方式。

fprintf(stderr, "epoll set insertion error: fd=%d"0, client);

return -1;

}

}

else

// 如果不是主socket的事件的话，则代表这是一个用户的socket的事件，

// 则用来处理这个用户的socket的事情是，比如说read(fd,xxx)之类，或者一些其他的处理。

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

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39

## 8 EPOLL模型的简单实例

#include <iostream>

#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>

#define MAXLINE 10

#define OPEN\_MAX 100

#define LISTENQ 20

#define SERV\_PORT 5555

#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 i, maxi, listenfd, connfd, sockfd, epfd, nfds;

ssize\_t n;

char line[MAXLINE];

socklen\_t clilen;

//声明epoll\_event结构体的变量, ev用于注册事件, events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, 指定生成描述符的最大范围为256

epfd = epoll\_create(256);

struct sockaddr\_in clientaddr;

struct sockaddr\_in serveraddr;

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

setnonblocking(listenfd); //把用于监听的socket设置为非阻塞方式

ev.data.fd = listenfd; //设置与要处理的事件相关的文件描述符

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

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

bzero(&serveraddr, sizeof(serveraddr));

serveraddr.sin\_family = AF\_INET;

char \*local\_addr = "200.200.200.204";

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

serveraddr.sin\_port = htons(SERV\_PORT); //或者htons(SERV\_PORT);

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

listen(listenfd, LISTENQ);

maxi = 0;

for( ; ; )

{

nfds = epoll\_wait(epfd, events, 20, 500); //等待epoll事件的发生

for(i = 0; i < nfds; ++i) //处理所发生的所有事件

{

if(events[i].data.fd == listenfd) //监听事件

{

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

if(connfd < 0)

{

perror("connfd<0");

exit(1);

}

setnonblocking(connfd); //把客户端的socket设置为非阻塞方式

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

std::cout << "connect from " << str <<std::endl;

ev.data.fd=connfd; //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\_ADD, connfd, &ev);

//注册ev事件

}

else if(events[i].events&EPOLLIN) //读事件

{

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

{

continue;

}

if ( (n = read(sockfd, line, MAXLINE)) < 0) // 这里和IOCP不同

{

if (errno == ECONNRESET)

{

close(sockfd);

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

}

else

{

std::cout<<"readline error"<<std::endl;

}

}

else if (n == 0)

{

close(sockfd);

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

}

ev.data.fd=sockfd; //设置用于写操作的文件描述符

ev.events=EPOLLOUT | EPOLLET; //设置用于注测的写操作事件

//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)//写事件

{

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; //设置用于注册的读操作事件

//修改sockfd上要处理的事件为EPOLIN

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

}

}

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47
* 48
* 49
* 50
* 51
* 52
* 53
* 54
* 55
* 56
* 57
* 58
* 59
* 60
* 61
* 62
* 63
* 64
* 65
* 66
* 67
* 68
* 69
* 70
* 71
* 72
* 73
* 74
* 75
* 76
* 77
* 78
* 79
* 80
* 81
* 82
* 83
* 84
* 85
* 86
* 87
* 88
* 89
* 90
* 91
* 92
* 93
* 94
* 95
* 96
* 97
* 98
* 99
* 100
* 101
* 102
* 103
* 104
* 105
* 106
* 107
* 108
* 109
* 110
* 111
* 112
* 113
* 114
* 115
* 116
* 117
* 118
* 119
* 120

## 9.epoll进阶思考

9.1. 问题来源   
最近学习EPOLL模型，介绍中说将EPOLL与Windows IOCP模型进行比较，说其的优势在于解决了IOCP模型大量线程上下文切换的开销，于是可以看出，EPOLL模型不需要多线程，即单线程中可以处理EPOLL逻辑。如果引入多线程反而会引起一些问题。但是EPOLL模型的服务器端到底可以不可以用多线程技术，如果可以，改怎么取舍，这成了困扰我的问题。上网查了一下，有这样几种声音：   
(1) “要么事件驱动(如epoll)，要么多线程，要么多进程，把这几个综合起来使用，感觉更加麻烦。”；   
(2) “单线程使用epoll，但是不能发挥多核；多线程不用epoll。”；   
(3) “主通信线程使用epoll所有需要监控的FD，有事件交给多线程去处理”；   
(4) “既然用了epoll, 那么线程就不应该看到fd, 而只看到的是一个一个的业务请求/响应； epoll将网络数据组装成业务数据后, 转交给业务线程进行处理。这就是常说的半同步半异步”。   
我比较赞同上述(3)、(4)中的观点   
EPOLLOUT只有在缓冲区已经满了，不可以发送了，过了一会儿缓冲区中有空间了，就会触发EPOLLOUT，而且只触发一次。如果你编写的程序的网络IO不大，一次写入的数据不多的时候，通常都是epoll\_wait立刻就会触发 EPOLLOUT；如果你不调用 epoll，直接写 socket，那么情况就取决于这个socket的缓冲区是不是足够了。如果缓冲区足够，那么写就成功。如果缓冲区不足，那么取决你的socket是不是阻塞的，要么阻塞到写完成，要么出错返回。所以EPOLLOUT事件具有较大的随机性，ET模式一般只用于EPOLLIN, 很少用于EPOLLOUT。   
9.2. 具体做法   
(1) 主通信线程使用epoll所有需要监控的FD，负责监控listenfd和connfd，这里只监听EPOLLIN事件，不监听EPOLLOUT事件；   
(2) 一旦从Client收到了数据以后，将其构造成一个消息，放入消息队列中；   
(3) 若干工作线程竞争，从消息队列中取出消息并进行处理，然后把处理结果发送给客户端。发送客户端的操作由工作线程完成。直接进行write。write到EAGAIN或EWOULDBLOCK后，线程循环continue等待缓冲区队列   
发送函数代码如下：

bool send\_data(int connfd, char \*pbuffer, unsigned int &len,int flag)

{

if ((connfd < 0) || (0 == pbuffer))

{

return false;

}

int result = 0;

int remain\_size = (int) len;

int send\_size = 0;

const char \*p = pbuffer;

time\_t start\_time = time(NULL);

int time\_out = 3;

do

{

if (time(NULL) > start + time\_out)

{

return false;

}

send\_size = send(connfd, p, remain\_size, flag);

if (nSentSize < 0)

{

if ((errno == EAGAIN) || (errno == EWOULDBLOCK) || (errno == EINTR))

{

continue;

}

else

{

len -= remain\_size;

return false;

}

}

p += send\_size;

remain\_size -= send\_size;

}while(remain\_size > 0);

return true;

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42

## 10 epoll 实现服务器和客户端例子

最后我们用C++实现一个简单的客户端回射，所用到的代码文件是

net.h server.cpp client.cpp

* 1

服务器端：epoll实现的，干两件事分别为：1.等待客户端的链接，2.接收来自客户端的数据并且回射；

客户端：select实现，干两件事为:1.等待键盘输入，2.发送数据到服务器端并且接收服务器端回射的数据；

/\*\*\*\*\*\*\*\*\*\*\*

net.h

\*\*\*\*\*\*\*\*\*\*\*/

#include <stdio.h>

#ifndef \_NET\_H

#define \_NET\_H

#include <iostream>

#include <vector>

#include <algorithm>

#include <stdio.h>

#include <sys/types.h>

#include <sys/epoll.h> //epoll ways file

#include <sys/socket.h>

#include <fcntl.h> //block and noblock

#include <stdlib.h>

#include <error.h>

#include <unistd.h>

#include <arpa/inet.h>

#include <netinet/in.h>

#include <string.h>

#include <signal.h>

using namespace std;

#define hand\_error(msg) do{perror(msg); exit(EXIT\_FAILURE);}while(0)

#endif

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32

/\*\*\*\*\*\*\*\*\*\*\*

server.c

\*\*\*\*\*\*\*\*\*\*\*/

#include "net.h"

#define MAX\_EVENTS 10000

int setblock(int sock)

{

int ret = fcntl(sock, F\_SETFL, 0);

if (ret < 0 )

hand\_error("setblock");

return 0;

}

int setnoblock(int sock) //设置非阻塞模式

{

int ret = fcntl(sock, F\_SETFL, O\_NONBLOCK );

if(ret < 0)

hand\_error("setnoblock");

return 0;

}

int main()

{

signal(SIGPIPE,SIG\_IGN);

int listenfd;

listenfd = socket( AF\_INET, SOCK\_STREAM,0 ); //create a socket stream

if( listenfd < 0 )

hand\_error( "socket\_create");

setnoblock(listenfd);

int on = 1;

if( setsockopt( listenfd, SOL\_SOCKET, SO\_REUSEADDR, &on, sizeof(on))< 0)

hand\_error("setsockopt");

struct sockaddr\_in my\_addr;

memset(&my\_addr, 0, sizeof(my\_addr));

my\_addr.sin\_family = AF\_INET;

my\_addr.sin\_port = htons(18000); //here is host sequeue

my\_addr.sin\_addr.s\_addr = inet\_addr("127.0.0.1");

if( bind( listenfd, (struct sockaddr \*)&my\_addr, sizeof(my\_addr)) < 0)

hand\_error("bind");

int lisId = listen(listenfd, SOMAXCONN);

if( lisId < 0) //LISTEN

hand\_error("listen");

struct sockaddr\_in peer\_addr; //用来 save client addr

socklen\_t peerlen;

//下面是一些初始化，都是关于epoll的。

vector<int> clients;

int count = 0;

int cli\_sock = 0;

int epfd = 0; //epoll 的文件描述符

int ret\_events; //epoll\_wait()的返回值

struct epoll\_event ev\_remov, ev, events[MAX\_EVENTS]; //events 用来存放从内核读取的的事件

ev.events = EPOLLET | EPOLLIN; //边缘方式触发

ev.data.fd = listenfd;

epfd = epoll\_create(MAX\_EVENTS); //create epoll，返回值为epoll的文件描述符

//epfd = epoll\_create(EPOLL\_CLOEXEC); //新版写法

if(epfd < 0)

hand\_error("epoll\_create");

int ret = epoll\_ctl(epfd, EPOLL\_CTL\_ADD, listenfd, &ev); //添加时间

if(ret < 0)

hand\_error("epoll\_ctl");

while(1)

{

ret\_events = epoll\_wait(epfd, events, MAX\_EVENTS, -1); //类似于select函数，这里是等待事件的到来。

if(ret\_events == -1)

{

cout<<"ret\_events = "<<ret\_events<<endl;

hand\_error("epoll\_wait");

}

if( ret\_events == 0)

{

cout<<"ret\_events = "<<ret\_events<<endl;

continue;

}

// cout<<"ret\_events = "<<ret\_events<<endl;

for( int num = 0; num < ret\_events; num ++)

{

cout<<"num = "<<num<<endl;

cout<<"events[num].data.fd = "<<events[num].data.fd<<endl;

if(events[num].data.fd == listenfd) //client connect

{

cout<<"listen sucess and listenfd = "<<listenfd<<endl;

cli\_sock = accept(listenfd, (struct sockaddr\*)&peer\_addr, &peerlen);

if(cli\_sock < 0)

hand\_error("accept");

cout<<"count = "<<count++;

printf("ip=%s,port = %d\n", inet\_ntoa(peer\_addr.sin\_addr),peer\_addr.sin\_port);

clients.push\_back(cli\_sock);

setnoblock(cli\_sock); //设置为非阻塞模式

ev.data.fd = cli\_sock;// 将新连接也加入EPOLL的监听队列

ev.events = EPOLLIN | EPOLLET ;

if(epoll\_ctl(epfd, EPOLL\_CTL\_ADD, cli\_sock, &ev)< 0)

hand\_error("epoll\_ctl");

}

else if( events[num].events & EPOLLIN)

{

cli\_sock = events[num].data.fd;

if(cli\_sock < 0)

hand\_error("cli\_sock");

char recvbuf[1024];

memset(recvbuf, 0 , sizeof(recvbuf));

int num = read( cli\_sock, recvbuf, sizeof(recvbuf));

if(num == -1)

hand\_error("read have some problem:");

if( num == 0 ) //stand of client have exit

{

cout<<"client have exit"<<endl;

close(cli\_sock);

ev\_remov = events[num];

epoll\_ctl(epfd, EPOLL\_CTL\_DEL, cli\_sock, &ev\_remov);

clients.erase(remove(clients.begin(), clients.end(), cli\_sock),clients.end());

}

fputs(recvbuf,stdout);

write(cli\_sock, recvbuf, strlen(recvbuf));

}

}

}

return 0;

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47
* 48
* 49
* 50
* 51
* 52
* 53
* 54
* 55
* 56
* 57
* 58
* 59
* 60
* 61
* 62
* 63
* 64
* 65
* 66
* 67
* 68
* 69
* 70
* 71
* 72
* 73
* 74
* 75
* 76
* 77
* 78
* 79
* 80
* 81
* 82
* 83
* 84
* 85
* 86
* 87
* 88
* 89
* 90
* 91
* 92
* 93
* 94
* 95
* 96
* 97
* 98
* 99
* 100
* 101
* 102
* 103
* 104
* 105
* 106
* 107
* 108
* 109
* 110
* 111
* 112
* 113
* 114
* 115
* 116
* 117
* 118
* 119
* 120
* 121
* 122
* 123
* 124
* 125
* 126
* 127
* 128
* 129
* 130
* 131

/\*\*\*\*\*\*\*\*\*\*\*

client.c

\*\*\*\*\*\*\*\*\*\*\*/

#include "net.h"

int main()

{

signal(SIGPIPE,SIG\_IGN);

int sock;

sock = socket( AF\_INET, SOCK\_STREAM,0 ); //create a socket stream

if( sock< 0 )

hand\_error( "socket\_create");

struct sockaddr\_in my\_addr;

//memset my\_addr;

memset(&my\_addr, 0, sizeof(my\_addr));

my\_addr.sin\_family = AF\_INET;

my\_addr.sin\_port = htons(18000); //here is host sequeue

// my\_addr.sin\_addr.s\_addr = htonl( INADDR\_ANY );

my\_addr.sin\_addr.s\_addr = inet\_addr("127.0.0.1");

int conn = connect(sock, (struct sockaddr \*)&my\_addr, sizeof(my\_addr)) ;

if(conn != 0)

hand\_error("connect");

char recvbuf[1024] = {0};

char sendbuf[1024] = {0};

fd\_set rset;

FD\_ZERO(&rset);

int nready = 0;

int maxfd;

int stdinof = fileno(stdin);

if( stdinof > sock)

maxfd = stdinof;

else

maxfd = sock;

while(1)

{

//select返回后把原来待检测的但是仍没就绪的描述字清0了。所以每次调用select前都要重新设置一下待检测的描述字

FD\_SET(sock, &rset);

FD\_SET(stdinof, &rset);

nready = select(maxfd+1, &rset, NULL, NULL, NULL);

cout<<"nready = "<<nready<<" "<<"maxfd = "<<maxfd<<endl;

if(nready == -1 )

break;

else if( nready == 0)

continue;

else

{

if( FD\_ISSET(sock, &rset) ) //检测sock是否已经在集合rset里面。

{

int ret = read( sock, recvbuf, sizeof(recvbuf)); //读数据

if( ret == -1)

hand\_error("read");

else if( ret == 0)

{

cout<<"sever have close"<<endl;

close(sock);

break;

}

else

{

fputs(recvbuf,stdout); //输出数据

memset(recvbuf, 0, strlen(recvbuf));

}

}

if( FD\_ISSET(stdinof, &rset)) //检测stdin的文件描述符是否在集合里面

{

if(fgets(sendbuf, sizeof(sendbuf), stdin) != NULL)

{

int num = write(sock, sendbuf, strlen(sendbuf)); //写数据

cout<<"sent num = "<<num<<endl;

memset(sendbuf, 0, sizeof(sendbuf));

}

}

}

}

return 0;

}