<https://blog.csdn.net/qicheng777/article/details/73642622>

## 一、关于socket通信

**服务器端工作流程：**

* 调用 socket() 函数创建套接字 用 bind() 函数将创建的套接字与服务端IP地址绑定
* 调用listen()函数监听socket() 函数创建的套接字，等待客户端连接 当客户端请求到来之后
* 调用 accept()函数接受连接请求，返回一个对应于此连接的新的套接字，做好通信准备
* 调用 write()/read() 函数和 send()/recv()函数进行数据的读写，通过 accept() 返回的套接字和客户端进行通信 关闭socket（close）

**客户端工作流程：**

* 调用 socket() 函数创建套接字
* 调用 connect() 函数连接服务端
* 调用write()/read() 函数或者 send()/recv() 函数进行数据的读写
* 关闭socket(close)

## 二、用select实现服务器端编程：

select函数楼主在之前文章中（[select函数用法](http://blog.csdn.net/qicheng777/article/details/70229456)）已经提及，不在多做缀述。下面贴上服务器端代码servce.c

#include <stdio.h>

#include <netinet/in.h> //for souockaddr\_in

#include <sys/types.h>

#include <sys/socket.h>

#include <errno.h>

#include <stdlib.h>

#include <arpa/inet.h>

//for select

#include <sys/time.h>

#include <sys/types.h>

#include <unistd.h>

#include <sys/select.h>

#include <strings.h> //for bzero

#include <string.h>

#define BUFF\_SIZE 1024

#define backlog 7

#define ser\_port 11277

#define CLI\_NUM 3

int client\_fds[CLI\_NUM];

int main(int agrc,char \*\*argv)

{

int ser\_souck\_fd;

int i;

char input\_message[BUFF\_SIZE];

char resv\_message[BUFF\_SIZE];

struct sockaddr\_in ser\_addr;

ser\_addr.sin\_family= AF\_INET; //IPV4

ser\_addr.sin\_port = htons(ser\_port);

ser\_addr.sin\_addr.s\_addr = INADDR\_ANY; //指定的是所有地址

//creat socket

if( (ser\_souck\_fd = socket(AF\_INET,SOCK\_STREAM,0)) < 0 )

{

perror("creat failure");

return -1;

}

//bind soucket

if(bind(ser\_souck\_fd, (const struct sockaddr \*)&ser\_addr,sizeof(ser\_addr)) < 0)

{

perror("bind failure");

return -1;

}

//listen

if(listen(ser\_souck\_fd, backlog) < 0)

{

perror("listen failure");

return -1;

}

//fd\_set

fd\_set ser\_fdset;

int max\_fd=1;

struct timeval mytime;

printf("wait for client connnect!\n");

while(1)

{

mytime.tv\_sec=27;

mytime.tv\_usec=0;

FD\_ZERO(&ser\_fdset);

//add standard input

FD\_SET(0,&ser\_fdset);

if(max\_fd < 0)

{

max\_fd=0;

}

//add serverce

FD\_SET(ser\_souck\_fd,&ser\_fdset);

if(max\_fd < ser\_souck\_fd)

{

max\_fd = ser\_souck\_fd;

}

//add client

for(i=0;i<CLI\_NUM;i++) //用数组定义多个客户端fd

{

if(client\_fds[i]!=0)

{

FD\_SET(client\_fds[i],&ser\_fdset);

if(max\_fd < client\_fds[i])

{

max\_fd = client\_fds[i];

}

}

}

//select多路复用

int ret = select(max\_fd + 1, &ser\_fdset, NULL, NULL, &mytime);

if(ret < 0)

{

perror("select failure\n");

continue;

}

else if(ret == 0)

{

printf("time out!");

continue;

}

else

{

if(FD\_ISSET(0,&ser\_fdset)) //标准输入是否存在于ser\_fdset集合中（也就是说，检测到输入时，做如下事情）

{

printf("send message to");

bzero(input\_message,BUFF\_SIZE);

fgets(input\_message,BUFF\_SIZE,stdin);

for(i=0;i<CLI\_NUM;i++)

{

if(client\_fds[i] != 0)

{

printf("client\_fds[%d]=%d\n", i, client\_fds[i]);

send(client\_fds[i], input\_message, BUFF\_SIZE, 0);

}

}

}

if(FD\_ISSET(ser\_souck\_fd, &ser\_fdset))

{

struct sockaddr\_in client\_address;

socklen\_t address\_len;

int client\_sock\_fd = accept(ser\_souck\_fd,(struct sockaddr \*)&client\_address, &address\_len);

if(client\_sock\_fd > 0)

{

int flags=-1;

//一个客户端到来分配一个fd，CLI\_NUM=3，则最多只能有三个客户端，超过4以后跳出for循环，flags重新被赋值为-1

for(i=0;i<CLI\_NUM;i++)

{

if(client\_fds[i] == 0)

{

flags=i;

client\_fds[i] = client\_sock\_fd;

break;

}

}

if (flags >= 0)

{

printf("new user client[%d] add sucessfully!\n",flags);

}

else //flags=-1

{

char full\_message[]="the client is full!can't join!\n";

bzero(input\_message,BUFF\_SIZE);

strncpy(input\_message, full\_message,100);

send(client\_sock\_fd, input\_message, BUFF\_SIZE, 0);

}

}

}

}

//deal with the message

for(i=0; i<CLI\_NUM; i++)

{

if(client\_fds[i] != 0)

{

if(FD\_ISSET(client\_fds[i],&ser\_fdset))

{

bzero(resv\_message,BUFF\_SIZE);

int byte\_num=read(client\_fds[i],resv\_message,BUFF\_SIZE);

if(byte\_num > 0)

{

printf("message form client[%d]:%s\n", i, resv\_message);

}

else if(byte\_num < 0)

{

printf("rescessed error!");

}

//某个客户端退出

else //cancel fdset and set fd=0

{

printf("clien[%d] exit!\n",i);

FD\_CLR(client\_fds[i], &ser\_fdset);

client\_fds[i] = 0;

// printf("clien[%d] exit!\n",i);

continue; //这里如果用break的话一个客户端退出会造成服务器也退出。

}

}

}

}

}

return 0;

}

select实现多路复用，多路复用，顾名思义，就是说各做各的事，标准输入事件到来，有相关函数处理。服务器处理服务器的事件，客户端到来时有相关函数对其进行处理，通过select遍历各fd的读写情况，就不用担心阻塞了。

## 三、用epoll实现客户端编程：

### 1、客户端程序（epoll\_client.c）：

#include<stdio.h>

#include<stdlib.h>

#include<netinet/in.h>

#include<sys/socket.h>

#include<arpa/inet.h>

#include<string.h>

#include<unistd.h>

#include <sys/epoll.h>

#include <errno.h>

#include <fcntl.h>

#define BUFFER\_SIZE 1024

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

{

int i,n;

int connfd,sockfd;

struct epoll\_event ev,events[20]; //ev用于注册事件,数组用于回传要处理的事件

int epfd=epoll\_create(256);//创建一个epoll的句柄，其中256为你epoll所支持的最大句柄数

struct sockaddr\_in client\_addr;

struct sockaddr\_in server\_addr;

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

server\_addr.sin\_port = htons(11277);

server\_addr.sin\_addr.s\_addr =INADDR\_ANY;

bzero(&(server\_addr.sin\_zero), 8);

int server\_sock\_fd = socket(AF\_INET, SOCK\_STREAM, 0);

ev.data.fd=server\_sock\_fd;//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,server\_sock\_fd,&ev);//注册epoll事件

if(server\_sock\_fd == -1)

{

perror("socket error");

return 1;

}

char recv\_msg[BUFFER\_SIZE];

char input\_msg[BUFFER\_SIZE];

if(connect(server\_sock\_fd, (struct sockaddr \*)&server\_addr, sizeof(struct sockaddr\_in)) == 0)

{

for(;;)

{

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

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

{

if(events[i].events&EPOLLOUT) //有数据发送，写socket

{

bzero(input\_msg, BUFFER\_SIZE);

fgets(input\_msg, BUFFER\_SIZE, stdin);

sockfd = events[i].data.fd;

write(sockfd, recv\_msg, n);

ev.data.fd=sockfd;

ev.events=EPOLLIN|EPOLLET;

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

}

else if(events[i].events&EPOLLIN)//有数据到来，读socket

{

bzero(recv\_msg, BUFFER\_SIZE);

if((n = read(server\_sock\_fd, recv\_msg, BUFFER\_SIZE)) <0 )

{

printf("read error!");

}

ev.data.fd=server\_sock\_fd;

ev.events=EPOLLOUT|EPOLLET;

printf("%s\n",recv\_msg);

}

}

}

}

return 0;

}

### 2、关于epoll函数：

相比于select，epoll最大的好处在于它不会随着监听fd数目的增长而降低效率。因为在内核中的select实现中，它是采用轮询来处理的，轮询的fd数目越多，自然耗时越多。并且，在linux/posix\_types.h头文件有这样的声明：   
#define \_\_FD\_SETSIZE 1024   
表示select最多同时监听1024个fd

一共三个函数：

1、 int epoll\_create (int size);   
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。

2、 int epoll\_ctl (int epfd , int op, 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，struct epoll\_event结构如下：

struct epoll\_event {

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

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

};

typedef union epoll\_data {

void \*ptr;

int fd;

\_\_uint32\_t u32;

\_\_uint64\_t u64;

} epoll\_data\_t;

**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

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

* 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，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。

#### 使用步骤：

<1>首先通过create\_epoll(int maxfds)来创建一个epoll的句柄，其中maxfds为你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。

<2>然后每一帧的调用epoll\_wait (int epfd, epoll\_event events, int max events, int timeout) 来查询所有的网络接口。

<3>kdpfd为用epoll\_create创建之后的句柄，events是一个epoll\_event\*的指针，当epoll\_wait这个函数操作成功之后，epoll\_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是 epoll\_wait的超时，为0的时候表示马上返回，为-1的时候表示一直等下去，直到有事件范围，为任意正整数的时候表示等这么长的时间，如果一直没有事件，则返回。一般如果网络主循环是单独的线程的话，可以用-1来等，这样可以保证一些效率，如果是和主逻辑在同一个线程的话，则可以用0来保证主循环的效率。

epoll\_wait返回之后应该是一个循环，遍历所有的事件。

**基本上都是如下的框架：**

for( ; ; )

{

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

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

{

if(events[i].data.fd==listenfd) //有新的连接

{

connfd = accept(listenfd,(sockaddr \*)&clientaddr, &clilen); //accept这个连接

ev.data.fd=connfd;

ev.events=EPOLLIN|EPOLLET;

epoll\_ctl(epfd,EPOLL\_CTL\_ADD,connfd,&ev); //将新的fd添加到epoll的监听队列中

}

else if( events[i].events&EPOLLIN ) //接收到数据，读socket

{

n = read(sockfd, line, MAXLINE)) < 0 //读

ev.data.ptr = md; //md为自定义类型，添加数据

ev.events=EPOLLOUT|EPOLLET;

epoll\_ctl(epfd,EPOLL\_CTL\_MOD,sockfd,&ev);//修改标识符，等待下一个循环时发送数据，异步处理的精髓

}

else if(events[i].events&EPOLLOUT) //有数据待发送，写socket

{

struct myepoll\_data\* md = (myepoll\_data\*)events[i].data.ptr; //取数据

sockfd = md->fd;

send( sockfd, md->ptr, strlen((char\*)md->ptr), 0 ); //发送数据

ev.data.fd=sockfd;

ev.events=EPOLLIN|EPOLLET;

epoll\_ctl(epfd,EPOLL\_CTL\_MOD,sockfd,&ev); //修改标识符，等待下一个循环时接收数据

}

else

{

//其他的处理

}

}

}