

0326_select

```
● (base) [yufc@ALiCentos7:~]$ kill -l
 1) SIGHUP      2) SIGINT      3) SIGQUIT      4) SIGILL      5) SIGTRAP
 6) SIGABRT     7) SIGBUS      8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT    17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO      30) SIGPWR
31) SIGSYS     34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
○ (base) [yufc@ALiCentos7:~]$
```

信号驱动式其实用的比较少

当然我们可以打开一个选项

当底层进行IO的时候，发送SIGIO信号

如何进行非阻塞IO?

1. 让IO非阻塞，打开的时候，就可以指定非阻塞接口

2. 我们用统一的方式来进行非阻塞设置 fcntl();

select;

IO = 等 + 数据拷贝

1. 帮用户进行一次等待多个文件sock
2. 当哪些文件sock就绪了，select就要通知用户，对应就绪的sock有那些，然后，用户在调用recv/recvfrom/read等进行数据读取!

```

#include <iostream>
#include <unistd.h>

int main()
{
    // stdin == 0
    // 0号描述符就是很好的用来测试的描述符
    char buffer[1024];
    while (true)    scanf 这些天然就是阻塞式的
    {
        ssize_t s = read(0, buffer, sizeof(buffer) - 1);
        if(s > 0)
        {
            std::cout << "echo# " << buffer << std::endl;
        }
        else
        {
            // TODO
            std::cout << "read \"error\" " << std::endl;
        }
    }
    return 0;
}

bool SetNonBlock(int fd)
{
    int fl = fcntl(fd, F_GETFL); // 在底层获取当前fd对应文件的读写标记位
    if (fl < 0)
        return false;
    fcntl(fd, F_SETFL, fl | O_NONBLOCK); // 设置非阻塞
}

```

这个只需要设置一次，后续都是非阻塞了！



这个天然就是阻塞的
现在我们需要把从0号文件描述符读数据
改成非阻塞式的

```

(base) [yufc@ALiCentos7:~/Src/LinuxOS_And_ComputerNetwork/IO/0326]$ ./myfile
read "error"
read "error"
asread "error"
dfkajsdf
aajsdkfajecho# asdfkajsdf

read "error"
ksdfjadsf
adfjaksdecho# aajsdkfajksdfjadsf

read "error"
f
echo# adfjaksdf
sdfjadsf

```

现在就在非阻塞轮询

非阻塞的时候，我们是以出错的形式返回，告知上层没有就绪
a. 那我们如何甄别是真的出错了？
b. 还是因为没有就绪了呢？

```
23
24     char buffer[1024];
25     while (true)
26     {
27         errno = 0;
28         ssize_t s = read(0, buffer, sizeof(buffer) - 1); // 出错，不仅仅是错误返回值，errno变量也会被设置，表明出错
29         if (s > 0)
30         {
31             buffer[s - 1] = 0;
32             std::cout << "echo# " << buffer << " errno: " << errno << " errstring: " << strerror(errno) << std::endl;
33         }
34         else
35         {
36             // TODO
37             std::cout << "read \"error\""
38             << " errno: " << errno << " errstring: " << strerror(errno) << std::endl;
39             sleep(1);
        }
```

调试控制台 终端 端口

> < bash - 032

```
(base) [yufc@ALiCentos7:~/Src/LinuxOS_And_ComputerNetwork/IO/0326]$ make
g++ -o myfile myfile.cc -std=c++11
(base) [yufc@ALiCentos7:~/Src/LinuxOS And ComputerNetwork/IO/0326]$ ./myfile
read "error" errno: 11 errstring: Resource temporarily unavailable
read "error" errno: 11 errstring: Resource temporarily unavailable
ajdkfajread "error" errno: 11 errstring: Resource temporarily unavailable
dfa
echo# ajdkfajdfa errno: 0 errstring: Success
read "error" errno: 11 errstring: Resource temporarily unavailable
read "error" errno: 11 errstring: Resource temporarily unavailable
^C
(base) [yufc@ALiCentos7:~/Src/LinuxOS And ComputerNetwork/IO/0326]$
```

把错误码带上，我们就能看到现象了

那么11号errno码是什么？
我们接着来看看

```
        }
    else
    {
        // TODO
        std::cout << "read \\"error\\""
        |     |     << " errno: " << errno << " errstring: " << strerror(errno) << std::endl;
        if(errno == EWOULDBLOCK || errno == EAGAIN)
        {
            std::cout << "当前0号fd数据没有就绪, 请下一次再来试试吧" << std::endl;
            continue;
        }
        else if (errno == EINTR)
        {
            std::cout << "当前IO被信号中断了, 再试一试吧" << std::endl;
            continue;
        }
        else
        {
            //出错处理
        }
    }
```



这个就是非阻塞IO的代码框架

select

1. 帮用户进行一次等待多个文件sock
2. 当那些文件sock就绪了, select就要通知用户, 对应就绪的sock有哪些, 然后, 用户就可以调用recv/recvfrom/read等接口去读取

1. 快速认识一下select一个接口
2. fd_set
3. 接口挑一个重点参数，细致分析一下
4. 推而广之，理解剩余类似的参数
5. 快速编写代码
6. select的一般的编写代码的模式
7. 完成我们的代码

1. 快速认识一下select一个接口

```
/* According to POSIX.1-2001 */
#include <sys/select.h>
int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
```

nfds 是最大的等待文件描述符值+1，很好理解，毕竟底层用数组存

为什么select需要时间这些东西呢？

因为select等待多个fd的时候，策略可以选择

1. 阻塞式 -- 传nullptr
2. 非阻塞式 -- 传{0, 0}
3. 可以设置timeout时间，时间内阻塞，时间到立马返回
{5, 0} 这样传进去就是等五秒

至少只要有一个fd数据就绪or空间就绪，我们就可以进行返回了

至少只要有一个fd数据就绪or空间就绪，我们就可以进行返回了

全部都是输入&&输出型参数

```
struct timeval {
    time_t tv_sec; /* seconds */
    suseconds_t tv_usec; /* microseconds */
};
```

```
gettimeofday: 1685433086.577226
time: 1685433087
gettimeofday: 1685433087.577312
time: 1685433088
gettimeofday: 1685433088.577401
time: 1685433089
gettimeofday: 1685433089.577487
time: 1685433090
gettimeofday: 1685433090.577585
^C
```

就是这些东西

1. 快速认识一下select一个接口

```
/* According to POSIX.1-2001 */  
#include <sys/select.h>  
int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
```

等

至少只要有一个fd数据就绪or空间就绪，我们就可以进行返回了

这三个参数：

- a. 输入的时候：告诉内核，你要帮我关心哪一个sock的哪一种事件？
- b. 输出的时候：告诉用户，我所关心的sock中，哪些sock上的哪类事件已经就绪了

一般操作系统认为，一共有3种事件，分别对应三个参数

1. 读事件
2. 写事件
3. 异常事件

fd_set -> 文件描述符集

因为我们的文件描述符本质就是一个数字
所以fd_set本质就是一个位图！

```
void FD_CLR(int fd, fd_set *set);  
int FD_ISSET(int fd, fd_set *set);  
void FD_SET(int fd, fd_set *set);  
void FD_ZERO(fd_set *set);
```

这个位图不能通过用户按位与，按位或的形式去做修改，只能通过指定的函数！

现在我们挑一个参数，我们挑readfds这个参数，来详细分析一下select

readfds

- a. 输入时：用户->内核，我的比特位，比特位的位置，表示文件描述符值，比特位的内容表示：是否关心 --- 假设 0000 1010 表示 -> 只关心文件描述符1, 3上的读事件，其他文件描述符的读事件我不关心
- b. 输出时：内核->用户，我是OS，用户你让我关心的多个fd有结果了。比特位的位置依然表示文件描述符的值，但是内容表示->是否就绪 0000 1000 -> 表示3号文件描述符就绪了！-> 用户可以直接读取3号描述符，而不会被阻塞！

因为用户和内核都会去修改同一个位图。
所以fd_set这个东西用了一次之后要重新设置！



重要概念

接下来我们先快速编写一下select的代码来初步理解一下select

```
4
5  int main()
6  {
7      //1. fd_set是一个固定大小的位图，直接决定了select能关心的fd个数是有上限的!
8      fd_set fds;
9      std::cout << sizeof(fd_set)*8 << std::endl;
10 }
11 }
```

fd_set大小是固定的，所以select
能关心的fd个数是有上限的！

1024个！

调试控制台 终端 端口

> < terminal bas

(base) [yufc@ALiCentos7:~/Src/LinuxOS_And_ComputerNetwork/IO/0326/selectServer]\$./selectServer

1024

(base) [yufc@ALiCentos7:~/Src/LinuxOS_And_ComputerNetwork/IO/0326/selectServer]\$

```
fd_set rfds; // 读文件描述符集
FD_ZERO(&rfds);
struct timeval timeout = {5, 0};
while (true)
{
    // 如何看待listensock?
    // 获取新链接，我们依旧把它看作IO, input事件
    // 如果没有连接到来了呢？那就阻塞了！ -- 所以不能直接调用accept
    // int sock = Sock::Accept(__listen_sock, ...);
    FD_SET(__listen_sock, &rfds); // 将listensocket添加到读文件描述符集里面
    int n = select(__listen_sock + 1, &rfds, nullptr, nullptr, nullptr);
    // int n = select(__listen_sock + 1, &rfds, nullptr, nullptr, &timeout);
    switch (n)
    {
        case 0: //超时
            logMessage(DEBUG, "time out...");
            break;
        case -1: //等待失败
            logMessage(WARNING, "select error: %d : %s", errno, strerror(errno));
            break;
        default:
            //成功的
            logMessage(DEBUG, "get a new link event ..."); //
            break;
    }
}
```

起来之后我们用telnet去连
为什么有链接来的时候，这里会一直打印？
我不是只有一个链接吗？
因为当前链接来的时候，我们没有把它取走，
所以他一直提醒你
一直告诉我们有就绪的

```
void Start()
{
    private:
        void HandlerEvent(const fd_set &rfds)
        {
            std::string client_ip;
            uint16_t client_port;
            if(FD_ISSET(__listen_sock, &rfds))
            {
                //listensock上面的读事件就绪了，表示可以读取了！
                //可以获取新链接了
                int sock = Sock::Accept(__listen_sock, &client_ip, &client_port);
                //这里进行accept会不会阻塞？一定不会！
                if(sock < 0)
                {
                    logMessage(WARNING, "accept error");
                    return;
                }
                logMessage(DEBUG, "get a new line success : [%s:%d] : %d", client_ip.c_str(), client_port, sock);
                //TODO
                //能直接进行读取吗？不能！为什么不能？
            }
        }
    private:
}
```

```
logMessage(DEBUG, "get a new line success . [%s.%u] . %u , client_ip.c_str(), client_port, sock),
//TODO
//能直接进行读取吗？不能！为什么不能？
//谁最清楚数据什么时候就绪呢？select
//得到新连接的时候，此时我们应该考虑的是，将新的sock脱光select，让select帮我们进行检测sock上是否有新的数据
//有了数据，select读事件就绪，select就会通知我，我们再进行读取，我们就不会被阻塞了！！
```

所以这一步之后，我们应该把文件描述符托管给select，但是，
怎么托管呢？怎么把我们收到的文件描述符陆陆续续交给select
呢？ ---> 此时我们就卡住了！

为什么不能？

因为我们不知道select上面的数据什么时候来

select上面不是已经来了吗？
不是！

只是套接字搞定了，也就是三次握手完成了而已！

数据什么时候来？不知道
如果直接读取

数据没来，就会被阻塞住！

```
int select(int nfds, fd_set *readfds, fd_set *writefds,  
          fd_set *exceptfds, struct timeval *timeout);
```

1. nfds: 随着我们获取的sock越来越多，随着我们添加到select到sock越来越多，这注定了nfds每一次都可能要变化，我们需要对他动态计算
2. rfds/writefds/exceptfds: 都是输入输出型参数，注定了输入输出不一定是一样的，所以注定了我们每一次都要对rfds进行重新添加
3. timeout: 都是输入输出型参数，每一次都要进行重置，前提是你要的话
4. 1, 2 -->注定了我们必须将自己的合法的文件描述符需要单独全部保存起来，用来支持
 - a. 更新最大fd
 - b. 更新位图

Select服务器的编写规则：

1. 需要一个第三方数组用来保存所有的合法fd

```
while(true)  
{
```

1. 遍历数组，更新出最大值
2. 遍历数组，添加所有需要关心的fd到fd_set位图中
3. 调用select进行时间检测

```
}
```

```
85 }  
86 }  
87 private:  
88     uint16_t __port;  
89     int _listen_sock;  
90     int arr[NUM]; // 第三方位图  
91 };  
92 #endif
```

selectServer.hpp — yufc [SSH: AlibabaServer] 08

EXPLORER ... h++ selectServer.hpp ● h++ Sock.hpp C++ main.cc h++ Log.hpp

YUFC [SSH: ALIBABASERVER] Src > LinuxOS_And_ComputerNetwork > IO > 0326 > selectServer > h++ selectServer.hpp > SelectServer > HandlerEvent(const fd_set &)

```
    logMessage(WARNING, "accept error");
    return;
}
logMessage(DEBUG, "get a new line success : [%s:%d] : %d", client_ip.c_str(), client_port, sock);
// TODO
// 能直接进行读取吗? 不能! 为什么不能?
// 谁最清楚数据什么时候就绪呢? select
// 得到新连接的时候, 此时我们应该考虑的是, 将新的sock脱光选select, 让select帮我们进行检测sock上是否有新的数据
// 有了数据, select读事件就绪, select就会通知我, 我们再进行读取, 我们就不会被阻塞了!!!
// 要将sock添加给select, 只需要将fd添加到我们的数组中即可!!!
int pos = 1;
for(pos = 1; pos < NUM; pos++)
{
    if(__fd_array[pos] == FD_NONE)
    {
        //当前位置没有被占用
        break;
    }
}
if (pos == NUM)
{
    //数组满了, 不能再加了, 因为数组本来就是按照select监管的最大数量来定的
    logMessage(WARNING, "%s:%d", "select server already full, closs: %d", sock);
    close(sock);
}
else
{
    __fd_array[pos] = sock; // 此时, 我们就把accept到的sock放到数组里了, 其他不用管了!
}

private:
    uint16_t __port;
```

等到下一次外面的循环后
这个新的sock就会自动被添加到
select里了, select就开始帮忙监管了

DEBUG CONSOLE TERMINAL PORTS 9

> TERMINAL (base) [yufc@AliCentos7:~/Src/LinuxOS_And_ComputerNetwork/IO/0326/selectServer]\$ bash

bash selectS... bash

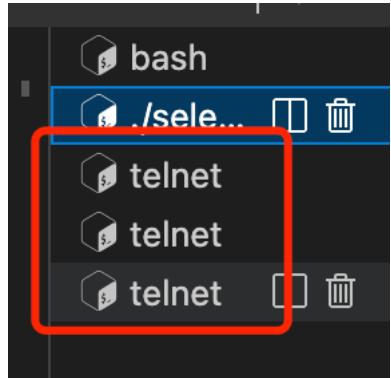
SSH: AlibabaServer master 0 0 9

Ln 122, Col 17 Spaces: 4 UTF-8 LF {} C++ Linux

```
void debugPrint()
{
    std::cout << "__fd_array[]:" << std::endl;
    for(int i = 0; i < NUM; i++)
    {
        std::cout << "__fd_array[i]" << " ";
    }
    std::cout << std::endl;
}
```

现在我就来看看
到底是不是每次获取新连接之后
__fd_array都会都一个值

```
30
31     // 规定 __fd_array[0] = __listen_sock --> 这个固定好就行，以后都不用动了，后面的再放后面接收的sock
32     __fd_array[0] = __listen_sock;
33 }
34 > -SelectServer() ...
35 void Start()
36 {
37     while (true)
38     {
39         // struct timeval timeout = {0, 0};
40         // 如何看待listensock?
41         // 获取新链接，我们依旧把它看作FIO, input事件
42         // 如果没有连接到来了呢？那就阻塞了！ -- 所以不能直接调用accept
43         // int sock = Sock::Accept(__listen_sock, ...);
44         // FD_SET(__listen_socl, &rfds); // 将listensocket添加到读文件描述符集里面
45         this->debugPrint();
46         rra_set_rras; // 读文件描述符集
47         FD_ZERO(&rfds); // 一定要保证一进来就要清空一次
48         int maxfd = __listen_sock;
49         for (int i = 0; i < NUM; i++)
50         {
51             if (__fd_array[i] == FD_NONE)
52                 continue;
53             FD_SET(__fd_array[i], &rfds); // 在数组里的都放进去
54             if (maxfd < __fd_array[i])
55             {
56                 maxfd = __fd_array[i];
57             }
58         }
59     }
60 }
```



The screenshot shows a terminal window with four tabs: bash, ./selectServer, telnet, and telnet. The telnet tab contains three entries, each with a red box around it:

- telnet
- telnet
- telnet

Below the terminal window is a code editor showing the file `selectServer.hpp`. The code is part of a C++ project named `YUFC` under the `Src` directory. The code implements a select server using `fd_set` and `select` system calls. It handles events for new links and prints debug information to the console.

```
73     break;
74     default:
75         // 成功的
76         logMessage(DEBUG, "get a new link event ..."); //
77         // 有事件到来了，我们就要去处理这个事件啊，所以要告诉HandlerEvent哪些事件就绪了
78         HandlerEvent(rfds);
79         break;
80     }
81 }
82
83 private:
84     > void HandlerEvent(const fd_set &rfds) ...
85     void debugPrint()
86     {
87         std::cout << "__fd_array[]:";
88         for (int i = 0; i < NUM; i++)
89         {
90             if (__fd_array[i] != -1)
91                 std::cout << __fd_array[i] << " ";
92         }
93         std::cout << std::endl;
94     }
95
96 private:
97     uint16_t port;
98 }
```

The terminal output shows the server's log messages, including the creation of a socket, initializing the TCP server, creating a base socket, and accepting multiple connections from telnet clients. The last three accepted connections are highlighted with red boxes.

```
[NORMAL] [1688544867] create socket success, sock: 3
[NORMAL] [1688544867] init TcpServer Success
[DEBUG] [1688544867] create base socket success
__fd_array[]:3
[DEBUG] [1688544875] get a new link event ...
[NORMAL] [1688544875] link success, serviceSock: 4 | 127.0.0.1 : -1870231538
[DEBUG] [1688544875] get a new line success : [127.0.0.1:44608] : 4
__fd_array[]:3 4
[DEBUG] [1688544927] get a new link event ...
[NORMAL] [1688544927] link success, serviceSock: 5 | 127.0.0.1 : -1870231538
[DEBUG] [1688544927] get a new line success : [127.0.0.1:44612] : 5
__fd_array[]:3 4 5
[DEBUG] [1688544940] get a new link event ...
[NORMAL] [1688544940] link success, serviceSock: 6 | 127.0.0.1 : -1870231538
[DEBUG] [1688544940] get a new line success : [127.0.0.1:44614] : 6
__fd_array[]:3 4 5 6
```

此时我们开启多个telnet
来连接

3, 4, 5, 6就加进去了

而3是listensock
后面的3个是accept到的
sock

写到了这个步骤，我们要清晰的认识到：

_fd_array里面有两类套接字

1. Listen套接字
2. Accept到的套接字

现在要清晰的认识到的事情：

1. 我们select中，就绪的fd会越来越多

```
private:  
    void HandlerEvent(const fd_set &rfds)  
{  
        for(int i = 0; i < NUM; i++)  
        {  
            //1. 去掉不合法的fd  
            if(_fd_array[i] == FD_NONE) continue;  
            //2. 合法的就一定就绪了？不一定！  
            if(FD_ISSET(_fd_array[i], &rfds))  
            {  
                if(_fd_array[i] == _listen_sock)  
                {  
                    //读事件就绪：连接事件到来，应该去accept  
                }  
                else  
                {  
                    //读事件就绪：INPUT事件到来，应该去recv, read  
                }  
            }  
        }  
    }  
}
```

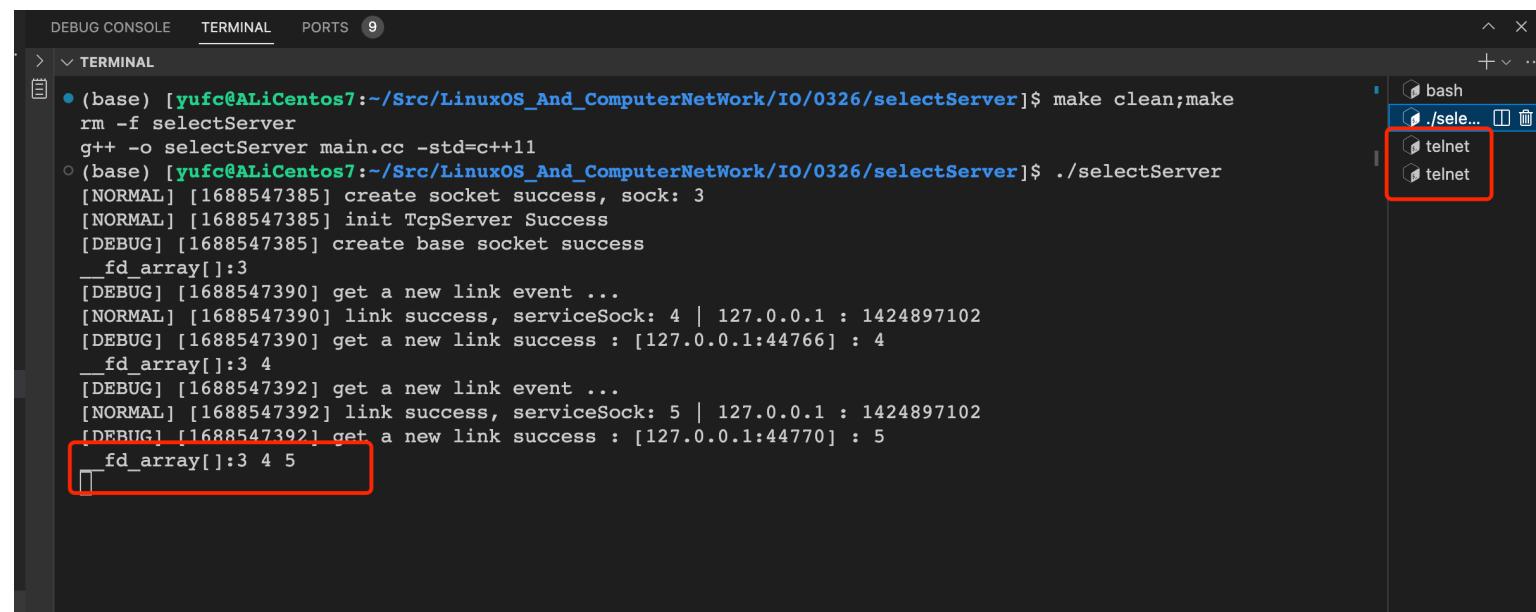


那么，现在这个代码的问题是什么？

1. 首先我们要明确，在Start里面，我们每过一个循环，都要调用一次select，都有以下两个目的：
 1. 把新的文件描述符告诉select，你要帮我看管这些rfds
 2. 问select，哪些文件描述符就绪，select怎么告诉我们？
2. 然后，调用完select，之后，就去调用HandlerEvent，去处理就绪的文件描述符，所以传rfds给HandlerEvent，但是现在的问题是。我们知道，我们需要关心的文件描述符，在_fd_array里面，而不是在rfds里面
3. rfds里面存的是，_fd_array里面，就绪的一部分，也就是说，rfds是_fd_array的子集。
4. ok现在的问题是什么？提问：每一个就绪的fd，要进行的动作难道是一样的？，我可以说，肯定不一样！如果我们对于每个就绪的fd，都去进行accept动作，这不是在瞎搞吗？只有listensock就绪，才去对这个fd进行accept啊，其他的不应该这样搞啊，所以我们要改代码结构！针对于每一个fd，他们要进行的动作是不一样的，如果是listensock就绪了，我们就去accept，如果是其他的读sock就绪了，我们就要去读！！！

```
private:  
    void Acceptor()  
    {  
        // 这个接口专门用来获取新连接, 即: listenSock就绪之后, HandlerEvent来调用Acceptor! !  
    }  
  
    void HandlerEvent(const fd_set &rfds)  
    {  
        for(int i = 0; i < NUM; i++)  
        {  
            //1. 去掉不合法的fd  
            if(_fd_array[i] == FD_NONE) continue;  
            //2. 合法的就一定就绪了? 不一定!  
            if(FD_ISSET(_fd_array[i], &rfds))  
            {  
                if(_fd_array[i] == _listen_sock)  
                {  
                    //读事件就绪: 连接事件到来, 应该去accept  
                }  
                else  
                {  
                    //读事件就绪: INPUT事件到来, 应该去recv, read  
                }  
            }  
        }  
    }  
}
```

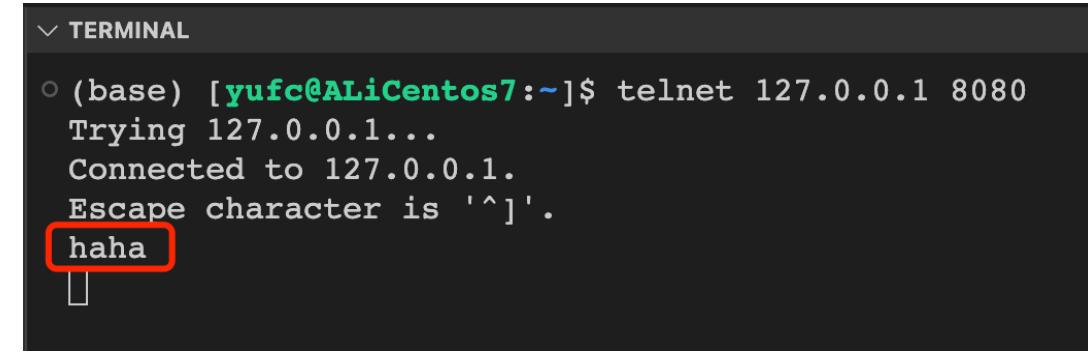
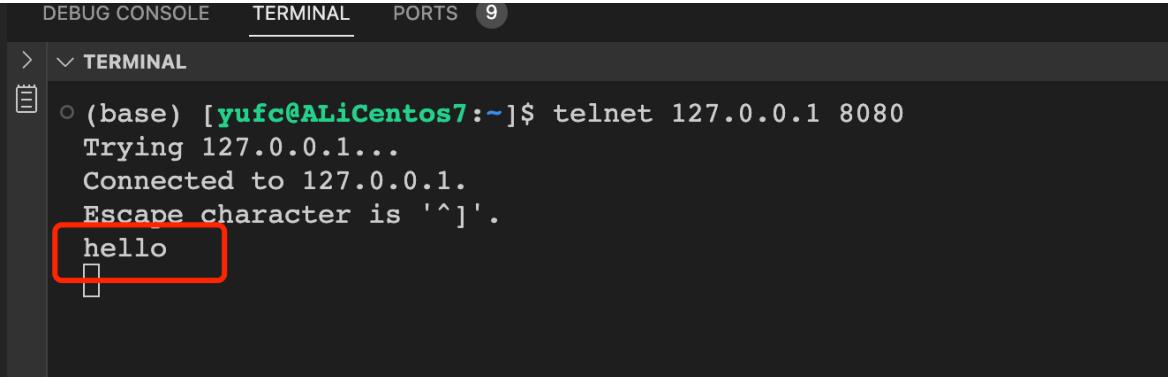
所以弄完这部分之后, 我们就可以测试一下了!



```
DEBUG CONSOLE TERMINAL PORTS 9  
> ✓ TERMINAL  
● (base) [yufc@AliCentos7:~/Src/LinuxOS_And_ComputerNetwork/IO/0326/selectServer]$ make clean;make  
rm -f selectServer  
g++ -o selectServer main.cc -std=c++11  
○ (base) [yufc@AliCentos7:~/Src/LinuxOS_And_ComputerNetwork/IO/0326/selectServer]$ ./selectServer  
[NORMAL] [1688547385] create socket success, sock: 3  
[NORMAL] [1688547385] init TcpServer Success  
[DEBUG] [1688547385] create base socket success  
_fd_array[]:3  
[DEBUG] [1688547390] get a new link event ...  
[NORMAL] [1688547390] link success, serviceSock: 4 | 127.0.0.1 : 1424897102  
[DEBUG] [1688547390] get a new link success : [127.0.0.1:44766] : 4  
_fd_array[]:3 4  
[DEBUG] [1688547392] get a new link event ...  
[NORMAL] [1688547392] link success, serviceSock: 5 | 127.0.0.1 : 1424897102  
[DEBUG] [1688547392] get a new link success : [127.0.0.1:44770] : 5  
_fd_array[]:3 4 5
```

这个肯定没问题

然后我们尝试发一下数据



在两个telnet里面都发一个信息

服务器便会疯狂提示我们

有10事件就绪了！！！

```
if (FD_ISSET(__fd_array[i], &rfds))
{
    if (__fd_array[i] == _listen_sock)
    {
        // 读事件就绪：连接事件到来，应该去accept
        Acceptor(); // 去accept
    }
    else
    {
        // 读事件就绪：INPUT事件到来，应该去recv, read
        logMessage(DEBUG, "message in, get IO event: %d", __fd_array[i]);
        // 在这一次读取的时候，不会被阻塞！！！
        // 此时select已经进行了时间检测，即：本次不会被阻塞！！！
        char buffer[1024];
        int n = recv(__fd_array[i], buffer, sizeof(buffer) - 1, 0);
        if (n > 0)
        {
            buffer[n] = 0;
            logMessage(DEBUG, "client[%d]# %s", __fd_array[i], buffer);
        }
    }
}
void debugPrint()
```

这样读取是有bug的！
有的！

面向字节流？数据报？黏包问题？

TCP是面向字节流的，怎么保证读到一个完整报文呢？

所以直接读取是有问题的！

怎么改，epoll的时候再说！

```
fd_array[]:3
[DEBUG] [1688550208] get a new link event ...
[NORMAL] [1688550208] link success, serviceSock: 4 | 127.0.0.1 : -95542738
[DEBUG] [1688550208] get a new link success : [127.0.0.1:44894] : 4
_fd_array[]:3 4
[DEBUG] [1688550215] get a new link event ...
[NORMAL] [1688550215] link success, serviceSock: 5 | 127.0.0.1 : -95542738
[DEBUG] [1688550215] get a new link success : [127.0.0.1:44896] : 5
_fd_array[]:3 4 5
[DEBUG] [1688550220] get a new link event ...
[DEBUG] [1688550220] message in, get IO event: 4
[DEBUG] [1688550220] client[4]# nihaod
收到消息了，处理（打印）完之后，挂了sock，也从数组中去掉了
_fd_array[]:3 4 5
[DEBUG] [1688550232] get a new link event ...
[DEBUG] [1688550232] message in, get IO event: 5
[DEBUG] [1688550232] client[5]# zaima
_fd_array[]:3 4 5
[DEBUG] [1688550314] get a new link event ...
[NORMAL] [1688550314] link success, serviceSock: 4 | 127.0.0.
[DEBUG] [1688550314] get a new link success : [127.0.0.1:4490
_fd_array[]:3 4
[DEBUG] [1688550323] get a new link event ...
[DEBUG] [1688550323] message in, get IO event: 4
[DEBUG] [1688550323] client[4] quit, me too...
_fd_array[]:3
```

这样其实代码就写完了

但是我们可以把一些细节上的东西优化一下

这个代码是多进程多线程吗？多进程吗？

但是他可以并发访问

就是select的功劳！！

Select服务器的编写规则：

1. 需要一个第三方数组用来保存所有的合法fd

```
while(true)
```

```
{
```

- 1. 遍历数组，更新出最大值

- 2. 遍历数组，添加所有需要关心的fd到fd_set位图中

- 3. 调用select进行时间检测

- 4. 遍历数组，找到就绪的事件，根据就绪事件，完成对应的多做

- a. accepter

- b. recver

```
}
```

扩展：如果我们要引入写入呢？学epoll的时候再说

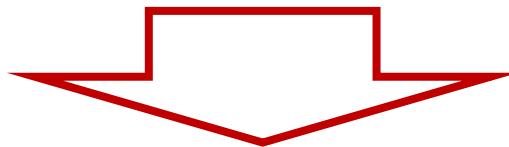
select的优缺点：

优点：（任何的多路转接方案都有这些优点）

1. 效率高（对比之前的，在多路转接这里，`select`只是弟弟） -> 因为他一直在HandlerEvent
2. 应用场景：有大量链接，但是只有少量的活跃的

缺点：

1. 为了维护第三方数组，所以`select`服务器会充满遍历操作
2. 每一次到要对`select`输出参数进行重新设定
3. `select`能够同时管理的fd个数是有上限的！
4. 因为几乎每一个参数都是输入输出型的，`select`一定会频繁的用户到内核，内核到用户的参数数据拷贝
5. 编码比较复杂



POLL