网络编程之 TCP

下面将通过一种**问题驱动**的模式学习 TCP 网络编程。我们将通过编写一个 **echo 回射服务器**学习以下内容：

1.基本的 TCP 连接

2.readn、writen 以及 readline 函数的编写

3.使用 select 改进客户端

4.服务端处理 SIGPIPE 信号，客户端采用 shutdown 改进程序，close 和 shutdown 的区别

5.客户端采用 poll、epoll 改进程序

服务器并发的三种基本思路

1.采用 select、poll、epoll 编写服务器端

2.采用多进程编写服务器端

3.采用多线程编写服务器端

一、基本的 TCP 连接

在我们编写 TCP 程序之前，我们先回顾下 read 和 write 的用法。

```
ssize_t read(int fd, void *buf, size_t count);
```

read 的返回值有以下几种情况：

-1 表示读取错误，errno 置为相应的错误代码，如果是中断造成的 read 失败，errno 为 EINTR

0 表示读到了末尾，在 socket 中表示对方已经关闭了连接，这里接收到一个 FIN 请求，从而返回 0

>0 表示实际读到的字节数，这个返回值小于 n 并不是错误，因为 TCP 连接中可能并没有那么多的数据

ssize_t write(int fd, const void *buf, size_t count);

write 的返回值与 read 基本相同。

下面我们编写一个实际的 TCP 连接。

server 端：

```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define ERR_EXIT(m) \
    do\
    {\
        perror(m);\
        exit(EXIT_FAILURE);\
    }while(0)

#define MAXLINE 1023



static void do_serviece(int peerfd) {
    char recvbuf[MAXLINE + 1];
```

```
        memset(recvbuf, 0x00, sizeof recvbuf);

        while (1) {
            int nread = read(peerfd, recvbuf, MAXLINE);
            if (nread < 0) {
                ERR_EXIT("read");
            }
            if (nread == 0) {
                fprintf(stdout, "peer close\n");
                break;
            }
            fprintf(stdout, "receive: %s", recvbuf);
            int nwrite = write(peerfd, recvbuf, nread);
            if (nwrite < 0) {
                ERR_EXIT("write");
            }
            memset(recvbuf, 0x00, sizeof recvbuf);
        }
    }

    int main(int argc, char **argv) {

        int listenfd = socket(AF_INET, SOCK_STREAM, 0);
        if (listenfd < 0) {
            ERR_EXIT("socket");
        }

        int on = 1;
        if (setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on))
    < 0)
            ERR_EXIT("setsockopt");

        struct sockaddr_in servaddr;
        servaddr.sin_family = AF_INET;
        servaddr.sin_port = htons(8989);
        servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
        socklen_t len = sizeof servaddr;
        int ret = bind(listenfd, (struct sockaddr*) &servaddr, len);
        if (ret < 0) {
            ERR_EXIT("bind");
        }

        ret = listen(listenfd, SOMAXCONN);
        if (ret < 0) {
```

```
            ERR_EXIT("listen");
        }

        struct sockaddr_in peeraddr;
        bzero(&peeraddr, sizeof peeraddr);
        len = sizeof peeraddr;
        int peerfd = accept(listenfd, (struct sockaddr*) &peeraddr, &len);
        if (peerfd < 0) {
            ERR_EXIT("accept");
        }
        fprintf(stdout, "IP= %s, port= %d\n", inet_ntoa(peeraddr.sin_addr),
                ntohs(peeraddr.sin_port));

        do_serviece(peerfd);

        close(peerfd);
        close(listenfd);

        return 0;
    }
```

client 端：

```
    #include <errno.h>
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
    #include <unistd.h>

    #include <sys/types.h>
    #include <sys/socket.h>
    #include <netinet/in.h>
    #include <arpa/inet.h>

    #define ERR_EXIT(m) \
        do\
        {\
            perror(m);\
            exit(EXIT_FAILURE);\
        }while(0)

    #define MAXLINE 1023
```

```
static void do_service(int fd) {
    char recvbuf[MAXLINE + 1];
    char sendbuf[MAXLINE + 1];
    memset(recvbuf, 0x00, sizeof recvbuf);
    memset(sendbuf, 0x00, sizeof sendbuf);

    while (fgets(sendbuf, MAXLINE, stdin) != NULL) {

        int nwrite = write(fd, sendbuf, strlen(sendbuf));
        if (nwrite < 0) {
            ERR_EXIT("write");
        }
        int nread = read(fd, recvbuf, MAXLINE);
        if (nread < 0) {
            ERR_EXIT("read");
        }
        if (nread == 0) {
            fprintf(stdout, "peer close\n");
            break;
        }
        fprintf(stdout, "receive: %s", recvbuf);
        memset(recvbuf, 0x00, sizeof recvbuf);
        memset(sendbuf, 0x00, sizeof sendbuf);
    }
}

int main(int argc, char **argv) {

    int fd = socket(AF_INET, SOCK_STREAM, 0);
    if (fd < 0) {
        ERR_EXIT("socket");
    }

    struct sockaddr_in servaddr;
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(8989);
    servaddr.sin_addr.s_addr = inet_addr("127.0.0.1");
    socklen_t len = sizeof servaddr;

    int ret = connect(fd, (struct sockaddr *) &servaddr, len);
    if (ret < 0) {
        ERR_EXIT("connect");
    }
```

```
        do_service(fd);

        close(fd);

        return 0;
    }
```

我们需要搞清楚一个问题：TCP 在传输数据的时，**什么数据需要处理大小端问题，什么数据不需要？**

这里要明白，UDP 是面向报文的协议，只要缓冲区足够，那么一次就可以接收到完整的报文。但是 TCP 是**面向字节流**的，发送的数据都是以字节为单位。

如果我们要发送字符串，“abc....xyz”，因为字符串是按照字节顺序拼接的，TCP 能够保证对方按照相同的顺序接收这些字节，所以对方接收到的也是”abc..xyz”。

但是数字就不一样，例如 int i = 0x12345678 ， 如果不考虑大小端的处理，假设 TCP 发送的顺序是 12 34 56 78，糟糕的是，接收方的字节序与发送方恰好相反，于是接收方收到数字为 0x78563412，从而出现了问题。

问题的根源在哪里？在于字符串在任何主机上都是顺序拼接，而 int 的 4 个字节在不同的主机上拼凑的方式不一样。

所以，**凡是以字节为单位的，不需要处理字节序的问题，以多个字节为基本单位的，就需要转化成网络字节序再发送**。

目前我们编写的 echo 程序存在这样一个问题：客户端关闭，服务器能及时感应到，也随之关闭，但是服务器关闭的时候，**客户端无法感知**。问题出在哪里？

原因： server 和 client 均有两个 fd，server 是阻塞在 fd 上，所以 client 关闭的时候，**server 的 read 函数迅速返回 0**，从而获知 client 已经关闭。但是 **client 是阻塞在 fgets 函数**上，所以 server 关闭的时候，client 无法知道，必须敲击回车，运行到 read 函数时才能获知 server 关闭。

二、readn、writen 以及 readline 函数的实现

TCP 存在一个所谓的**粘包**问题。

粘包问题是指：当发送数据过快时，例如快速发送两个报文，分别为 400、600 字节，此时我们接收方的缓冲区为 1024，那么我们可能接收到一个长度为 1000 的消息。我们即使知道这 1000 字节是由两条消息组成的，**也无法区分它们的边界**。此时就称两条报文粘合在了一起。处理的方式就是，我们可以在每条报文的前面加上要发送的字节数（**这个字节数要转化成网络序**），然后使用 writen 函数写入相应字节。接收方只需要先读取前面的字节数获取消息的长度为 n，然后**使用 readn 函数接收**即可。

readn 函数的返回值含义要与 read 相同。

实现如下：

```
ssize_t readn(int fd, void *buf, size_t count) {
    size_t nleft;
    ssize_t nread;          //BUG
    char *ptr;

    ptr = buf;
    nleft = count;

    while (nleft > 0) {
        nread = read(fd, ptr, nleft);
        if (nread < 0) {
            if (errno == EINTR) {
                nread = 0;
            } else {
                return -1;
            }
        } else if (nread == 0) {
            break;
        }
        nleft -= nread;
        ptr += nread;
    }
    return count - nleft;
}
```

writen 函数**必须返回写入的字节数，**否则就是错误。

writen 的实现如下：

```
ssize_t writen(int fd, const void *buf, size_t count) {
    size_t nleft;
    ssize_t nwritten;     //这里必须是有符号数!!!
    const char *ptr;

    ptr = buf;
    nleft = count;
```

```
    while (nleft > 0) {
        nwritten = write(fd, ptr, nleft);
        if (nwritten < 0) {
            if (errno == EINTR) {
                //nwritten = 0;
                continue;
            } else {
                return -1;
            }
        } else if (nwritten == 0) {
            continue;
        }
        nleft -= nwritten;
        ptr += nwritten;
    }

    return count;
}
```

此时我们用 readn 和 writen 改进我们的程序如下：

需要添加一个结构体

```
#define MAXLINE 1023

struct pack {
    int len;
    char data[MAXLINE + 1];
};
```

server 端：

```
static void do_service(int peerfd) {
//  char recvbuf[MAXLINE + 1];
//  memset(recvbuf, 0x00, sizeof recvbuf);

    struct pack recvpac;
    memset(&recvpac, 0x00, sizeof recvpac);

    while (1) {
        int nread = readn(peerfd, &recvpac.len, 4);
        if (nread < 0) {
```

```
            ERR_EXIT("nread");
        }
        if (nread < 4) {
            fprintf(stdout, "peer close\n");
            break;
        }
        int nlen = ntohl(recvpac.len);
        fprintf(stdout, "len: %d\n", nlen);

        nread = readn(peerfd, recvpac.data, nlen);
        if (nread < 0) {
            ERR_EXIT("nread");
        }
        if (nread < nlen) {
            fprintf(stdout, "peer close\n");
            break;
        }
        fprintf(stdout, "receive: %s", recvpac.data);


        writen(peerfd, &recvpac, 4 + nlen);

        memset(&recvpac, 0x00, sizeof recvpac);
    }
}
```

client 端：

```
static void do_service(int fd) {

    struct pack recvbuf;
    struct pack sendbuf;
    memset(&recvbuf, 0x00, sizeof recvbuf);
    memset(&sendbuf, 0x00, sizeof sendbuf);

    while (fgets(sendbuf.data, MAXLINE, stdin) != NULL) {

        int nlen = strlen(sendbuf.data);
        sendbuf.len = htonl(nlen);
        writen(fd, &sendbuf, nlen + sizeof (int));

        int nread = readn(fd, &recvbuf.len, 4);
        if(nread < 0){
```

```
                ERR_EXIT("nread");
            }
            if(nread < 4){
                fprintf(stdout, "peer close\n");
                break;
            }

            fprintf(stdout, "len: %d\n", nlen);
            nread = readn(fd, recvbuf.data, nlen);
            if (nread < 0) {
                ERR_EXIT("nread");
            }
            if (nread == 0) {
                fprintf(stdout, "peer close\n");
                break;
            }
            fprintf(stdout, "receive: %s", recvbuf.data);
            memset(&recvbuf, 0x00, sizeof recvbuf);
            memset(&sendbuf, 0x00, sizeof sendbuf);
        }
    }
```

我们的粘包问题，有了初步的解决方案。但是在现实中的很多应用层协议，在发送消息的时候是**以\r\n 作为一条消息边界**的，例如HTTP，而不是采用固定的字节数。

所以我们可以编写一个 readline 函数，每次读取消息，直到遇见\n 为止。

那么该如何实现这个函数呢？

我们平时读取消息后，数据立刻从缓冲区清除，但是 TCP 中有一个函数 recv，它的声明如下：

ssize_t recv(int sockfd, void *buf, size_t len, int flags);

最后一个参数是标志位，如果我们设置为 MSG_PEEK，那么在调用 recv 的时候，数据读取到缓冲区，但是**缓冲区中并不擦除数据**。

这里可以把这个 recv 函数调用，**看做一个斥候，为我们后续的数据读取探路**。

所以这里我们给出 readline 函数的实现，我们需要编写另外一个辅助函数 recv_peek：

```
ssize_t recv_peek(int sockfd, void *buf, size_t len) {
    int nread;
    while (1) {
        nread = recv(sockfd, buf, len, MSG_PEEK);
        if (nread < 0 && errno == EINTR) {    //被中断则继续读取
            continue;
        }
        if (nread < 0) {
            return -1;
        }
        break;
    }
    return nread;
}

ssize_t readline(int sockfd, void *buf, size_t maxline) {
    int nread;     //一次 IO 读取的数量
    int nleft;//还剩余的字节数
    char *ptr;     //存放数据的指针的位置
    int ret;   //readn 的返回值
    int total = 0; //目前总共读取的字节数

    nleft = maxline;
    ptr = buf;

    while (nleft > 0) {
        ret = recv_peek(sockfd, ptr, nleft);
        //注意这里读取的字节不够，绝对不是错误!!!
        if (ret <= 0) {
            return ret;
        }

        nread = ret;
        int i;
        for (i = 0; i < nread; ++i) {
            if (ptr[i] == '\n') {
```

```
                ret = readn(sockfd, ptr, i + 1);
                if (ret != i + 1) {
                    return -1;
                }
                total += ret;
                return total;  //返回此行的长度  '\n'包含在其中
            }
        }

        ret = readn(sockfd, ptr, nread);
        if (ret != nread) {
            return -1;
        }
        nleft -= nread;
        total += nread;
        ptr += nread;
    }
    return maxline;
}
```

所以我们此时再次改进程序，每次发送数据使用 writen，**因为我们用 fgets 获取数据，所以数据末尾总是\n，**接收方采用 readline 就可以了。

我们再次改进 echo 回射服务器的核心代码：

```
static void do_service(int fd) {
    char recvbuf[MAXLINE + 1];
    memset(recvbuf, 0x00, sizeof recvbuf);

    while (1) {
        int ret = readline(fd, recvbuf, MAXLINE);
        if (ret == -1) {
            ERR_EXIT("readline");
        }
        if(ret == 0){
            fputs("peer close", stdout);
            break;
        }
//      fputs(recvbuf, stdout);
        fprintf(stdout, "receive: %s", recvbuf);
```

```
            writen(fd, recvbuf, strlen(recvbuf));
            memset(recvbuf, 0x00, sizeof recvbuf);
        }
    }
```

客户端代码如下：

```
    static void do_service(int fd) {
        char recvbuf[MAXLINE + 1];
        char sendbuf[MAXLINE + 1];
        memset(recvbuf, 0x00, sizeof recvbuf);
        memset(sendbuf, 0x00, sizeof sendbuf);

        while (fgets(sendbuf, MAXLINE, stdin) != NULL) {
            writen(fd, sendbuf, strlen(sendbuf));
            int ret = readline(fd, recvbuf, MAXLINE);
            if (ret == -1) {
                ERR_EXIT("readline");
            }
            if (ret == 0) {
                fputs("peer close", stdout);
                break;
            }
            fprintf(stdout, "receive: %s", recvbuf);
            memset(recvbuf, 0x00, sizeof recvbuf);
            memset(sendbuf, 0x00, sizeof sendbuf);
        }

    }
```

三、用 select 改进客户端程序

回顾我们之前遇到的问题：客户端阻塞在 fgets 上，导致不能及时接收到 server 发来的 FIN 请求，下面我们尝试着用 select 解决这个问题。

我们目前遇到的难题在于 client 只能阻塞在一个 fd 上，如果阻塞在标准输入上，那么服务器关闭我们无法感知，如果阻塞在 read（或者 readn、readline）函数上，那么我们无法及时输入。

打一个比方：老师给学生布置了课上习题，然后要检查他们做的正确与否。我们目前的程序是这样的，老师走到学生 A 处，等待他做完，然后检查，再去找学生 B，**如果某个学生在那里消耗时间，老师和后面的同学也只能耐心等待**。

这样显然不行，稍微聪明点的人就知道应该这样做：**老师在台上等待，谁做完谁举手，然后老师过去检查**，这样就很大程度上提高了效率。

这就是我们所要讲述的 IO 复用模型。

select 用法参考：

http://www.cnblogs.com/Anker/archive/2013/08/14/3258674.html

这里我们采用 select 模型来改进我们的客户端。

```
void do_service(int fd) {
    char recvbuf[MAXLINE + 1];
    char sendbuf[MAXLINE + 1];
    memset(recvbuf, 0x00, sizeof recvbuf);
    memset(sendbuf, 0x00, sizeof sendbuf);

    fd_set rset;
    int maxfd;
    int nready;
    int stdin_fd = fileno(stdin);
    maxfd = (stdin_fd > fd) ? stdin_fd : fd;
    FD_ZERO(&rset);
    int stdin_eof = 0;

    while (1) {
        if (stdin_eof == 0) {
            FD_SET(stdin_fd, &rset);
        }
        FD_SET(fd, &rset);
```

```
            nready = select(maxfd + 1, &rset, NULL, NULL, NULL);
            if (nready < 0) {
                ERR_EXIT("select");
            }
            if (nready == 0) {
                continue;
            }
            if (FD_ISSET(stdin_fd, &rset)) {
                if (fgets(sendbuf, MAXLINE, stdin) == NULL) {
                    //break;
                    close(fd);
                    sleep(4);
                    exit(EXIT_FAILURE);


                } else {
                    if(strcmp(sendbuf, "\n") == 0){
                        continue;
                    }
                    writen(fd, sendbuf, strlen(sendbuf));
                    memset(sendbuf, 0x00, sizeof sendbuf);
                }
            }
            if (FD_ISSET(fd, &rset)) {
                int ret = readline(fd, recvbuf, MAXLINE);
                if (ret == -1) {
                    ERR_EXIT("readline");
                }
                if (ret == 0) {
                    fputs("server close\n", stdout);
                    break;
                }
                fprintf(stdout, "receive: %s", recvbuf);
                memset(recvbuf, 0x00, sizeof recvbuf);
            }
        }

    }
```

四、SIGPIPE 信号以及 shutdown 与 close 的区别

OK，现在我们的客户端可以及时获取 server 关闭的消息了。

我们进行一些破坏活动。

server 的 read 和 writen 之间加入一个 sleep(4)，代表处理数据

依次启动 server/client，在 client 中迅速发送两个字符串，然后 Ctrl+D，然后你就看到，回射的消息没有收到，服务器崩溃。

让我们分析其中的原因：

1.client 发送两条数据，然后 ctrl+D，这时 client 调用 close **同时关闭了读写方向**。所以 client 发送了一个 **FIN** 请求。

2.server 收到第一条信息，然后睡眠，4s 后，发送数据，因为此时对方已经无法接收，所以 server **收到一个 RST 报文**。

3.server 收到第二条消息，4s 后再次发送，此时**往已经接收到 RST 的连接中发送数据，server 端收到一个 SIGPIPE 信号**，从而崩溃。

原因分析清楚了，如何解决呢？

客户端可以**改用 shutdown**。这里先解释一下 close 和 shutdown 的区别，有两点：

1.close 同时关闭读和写两个方向，而 shutdown 可以选择只关闭其中一个方向。

2.close 仅当某 fd 的引用计数为 0 时才真正关闭，而 shutdown 是立刻关闭。

这里我们主要利用第一点，client 按下 Ctrl+D 后，只关闭发送端，接收端依然开启，所以 server 不会遇到 RST 请求以及 SIGPIPE 信号。

代码如下：

```
//

    static void do_service(int fd) {
        char recvbuf[MAXLINE + 1];
        char sendbuf[MAXLINE + 1];
        memset(recvbuf, 0x00, sizeof recvbuf);
        memset(sendbuf, 0x00, sizeof sendbuf);

        fd_set rset;
        int maxfd;
        int nready;
        int stdin_fd = fileno(stdin);
        maxfd = (stdin_fd > fd) ? stdin_fd : fd;
        FD_ZERO(&rset);
        int stdin_eof = 0;

        while (1) {
            if (stdin_eof == 0) {
                FD_SET(stdin_fd, &rset);
            }
            FD_SET(fd, &rset);
            nready = select(maxfd + 1, &rset, NULL, NULL, NULL);
            if (nready < 0) {
                ERR_EXIT("select");
            }
            if (nready == 0) {
                continue;
            }
            if (FD_ISSET(stdin_fd, &rset)) {
                if (fgets(sendbuf, MAXLINE, stdin) == NULL) {
                    //break;
                    /*close(fd);
                    sleep(4);
                    exit(EXIT_FAILURE); */
                    stdin_eof = 1; //停止监听该描述符
                    shutdown(fd, SHUT_WR);

                } else {
                    if(strcmp(sendbuf, "\n") == 0){
                        continue;
                    }
                    writen(fd, sendbuf, strlen(sendbuf));
```

```
                    memset(sendbuf, 0x00, sizeof sendbuf);
                }
            }
            if (FD_ISSET(fd, &rset)) {
                int ret = readline(fd, recvbuf, MAXLINE);
                if (ret == -1) {
                    ERR_EXIT("readline");
                }
                if (ret == 0) {
                    fputs("server close\n", stdout);
                    break;
                }
                fprintf(stdout, "receive: %s", recvbuf);
                memset(recvbuf, 0x00, sizeof recvbuf);
            }
        }

    }
```

我们似乎解决了问题，但是这种**依靠 client 端修改代码的方式并不稳妥！**服务器不能把自身的安全交给客户端来保证。

所以稳妥的方式是 **server 端必须处理 SIGPIPE 信号**。

如下：

```
signal(SIGPIPE, SIG_IGN);
```

五、用 poll 改进客户端

前面我们使用 select 改进了客户端，但是 select 存在一系列的缺点，

比如：

于是我们采用另一种模型：poll

poll 的用法参见：

http://www.cnblogs.com/Anker/archive/2013/08/15/3261006.html

使用 poll 编写的客户端程序如下：

```
static void do_service(int fd) {
    char recvbuf[MAXLINE + 1];
    char sendbuf[MAXLINE + 1];
    memset(recvbuf, 0x00, sizeof recvbuf);
    memset(sendbuf, 0x00, sizeof sendbuf);


    int maxi = 0;//最大可用的下标
    struct pollfd client[2];
    int nready;
    client[0].fd = fd;
    client[0].events = POLLIN;
    int stdin_fd = fileno(stdin);
    client[1].fd = stdin_fd;
    client[1].events = POLLIN;
    maxi = 1;

    while (1) {
        nready = poll(client, maxi + 1, -1);
        if (nready < 0) {
            if (errno == EINTR) {
                continue;
            }
            ERR_EXIT("poll");
        }
        if (nready == 0) {
            continue;
        }

        //if (FD_ISSET(stdin_fd, &rset)) {
        if (client[1].revents & POLLIN) {
            if (fgets(sendbuf, MAXLINE, stdin) == NULL) {
                //break;
                //stdin_eof = 1; //停止监听该描述符
                client[1].fd = -1; //停止监听
                shutdown(fd, SHUT_WR);

            } else {
                if (strcmp(sendbuf, "\n") == 0) {
                    continue;
                }
                writen(fd, sendbuf, strlen(sendbuf));
                memset(sendbuf, 0x00, sizeof sendbuf);
            }
```

```
            }
            //if (FD_ISSET(fd, &rset)) {
            if (client[0].revents & POLLIN) {
                int ret = readline(fd, recvbuf, MAXLINE);
                if (ret == -1) {
                    ERR_EXIT("readline");
                }
                if (ret == 0) {
                    fputs("server close\n", stdout);
                    client[0].fd = -1;
                    break;
                }
                fprintf(stdout, "receive: %s", recvbuf);
                memset(recvbuf, 0x00, sizeof recvbuf);
            }
        }

    }
```

六、用 epoll 改进客户端

select 和 poll 都存在这样一些缺点：

epoll 用法如下：

http://www.cnblogs.com/Anker/archive/2013/08/17/3263780.html

用 epoll 改写客户端如下：

```
static void do_service(int fd) {
    char recvbuf[MAXLINE + 1];
    char sendbuf[MAXLINE + 1];
    memset(recvbuf, 0x00, sizeof recvbuf);
    memset(sendbuf, 0x00, sizeof sendbuf);

    int epollfd = epoll_create(2);
    if (epollfd == -1) {
```

```
        ERR_EXIT("epoll_create");
    }
    struct epoll_event events[2];
    struct epoll_event ev;
    ev.data.fd = fd;
    ev.events = EPOLLIN;
    int ret = epoll_ctl(epollfd, EPOLL_CTL_ADD, fd, &ev);
    if (ret == -1) {
        ERR_EXIT("epoll_ctl");
    }
    int stdin_fd = fileno(stdin);
    ev.data.fd = stdin_fd;
    ev.events = EPOLLIN;
    ret = epoll_ctl(epollfd, EPOLL_CTL_ADD, stdin_fd, &ev);
    if (ret == -1) {
        ERR_EXIT("epoll_ctl");
    }
    int nready;

    while (1) {

        //nready = poll(client, maxi + 1, -1);
        nready = epoll_wait(epollfd, events, 2, -1);
        if (nready < 0) {
            if (errno == EINTR) {
                continue;
            }
            ERR_EXIT("epoll_wait");
        }
        if (nready == 0) {
            continue;
        }

        int i;
        for (i = 0; i < nready; ++i) {

            //if (client[1].revents & POLLIN) {
            if (events[i].data.fd == stdin_fd) {

                if (fgets(sendbuf, MAXLINE, stdin) == NULL) {
                    //break;
                    //stdin_eof = 1; //停止监听该描述符
                    //client[1].fd = -1; //停止监听
                    ev.data.fd = stdin_fd;
```

```
                    epoll_ctl(epollfd, EPOLL_CTL_DEL, stdin_fd, &ev); // 从
events 中删除该 fd
                    shutdown(fd, SHUT_WR);

                } else {
                    if (strcmp(sendbuf, "\n") == 0) {
                        continue;
                    }
                    writen(fd, sendbuf, strlen(sendbuf));
                    memset(sendbuf, 0x00, sizeof sendbuf);
                }
            }

            //if (client[0].revents & POLLIN) {
            if(events[i].data.fd == fd){
                int ret = readline(fd, recvbuf, MAXLINE);
                if (ret == -1) {
                    ERR_EXIT("readline");
                }
                if (ret == 0) {
                    fputs("server close\n", stdout);
                    //client[0].fd = -1;
                    ev.data.fd = fd;
                    epoll_ctl(epollfd, EPOLL_CTL_DEL, fd, &ev); //delete
                    close(epollfd);
                    exit(EXIT_SUCCESS);  //直接退出程序
                }
                fprintf(stdout, "receive: %s", recvbuf);
                memset(recvbuf, 0x00, sizeof recvbuf);
            }

        }
    }

}
```

在我们改进客户端的过程中，select、poll、epoll 的大概模型是相似的。

大概的流程都是这样的：

1.添加监听的描述符和时间

2.监听，等待返回

3.处理读事件

4.处理写事件

5.继续下次循环