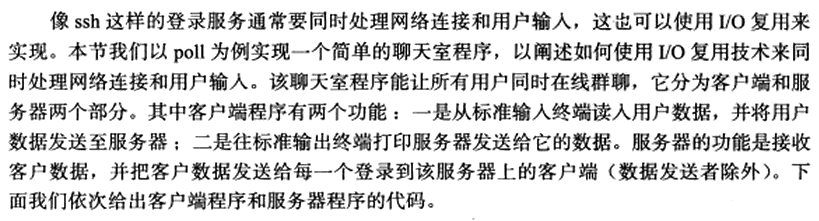
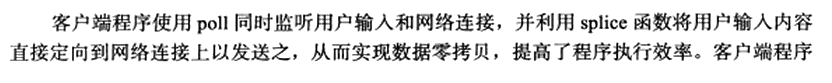
# 项目背景



# 客户端

## 关键技术



## 代码分解

### 参数检查与转换

if (argc <= 2)

{

printf("usage: %s ip host\n", basename(argv[0]));

return 1;

}

const char\* ip = argv[1];

int port = atoi(argv[2]);//字符串转整型

sockaddr\_in address;//tcp/ip专用socket地址

bzero(&address, sizeof(address));//bzero与memset效果相同，bzero是linux特有的函数

//memset(&address, '\0', sizeof(address));

address.sin\_family = AF\_INET;//设置地址协议族

address.sin\_port = htons(port);//htons:host to net short 字节序转换

inet\_pton(AF\_INET, ip, &address.sin\_addr);//ip地址转换函数，反过来是inet\_ntop

### 创建socket并连接到服务器

int sockfd = socket(PF\_INET, SOCK\_STREAM, 0);

ret = connect(sockfd, (sockaddr\*)&address, sizeof(address));

if (ret < 0)

{

printf("connection failure!errno:%d\n", errno);

return 1;

}

### 创建一个管道用于实现输入内容到socket的数据零拷贝

int p[2];

ret = pipe(p);

assert(ret != -1);

### 创建poll并设置监听fd和events

pollfd fds[2];

fds[0].fd = 0;

fds[0].events = POLLIN;

fds[0].revents = 0;

fds[1].fd = sockfd;

fds[1].events = POLLIN | POLLRDHUP;

fds[1].revents = 0;

### 监听输入与socket端口并处理事件

while (1)

{

ret = poll(fds, 2, -1);//返回就绪（可读、可写、异常）的文件描述符的总数，返回-1表示失败并设置了errno

if (ret < 0)

{

printf("poll failure!\n");

break;

}

//轮询每个文件描述符与事件

if (fds[1].revents & POLLRDHUP)//服务器关闭了socket连接

{

printf("connection closed by server!\n");

break;

}

else if (fds[1].revents & POLLIN)//socket端口有输入

{

memset(msg, '\0', BUF\_SIZE);

ret = recv(fds[1].fd, msg, BUF\_SIZE - 1, 0);

if (ret < 0)

{

if (errno != EAGAIN)

{

printf("Recv error!\n");

continue;

}

}

printf("%s", msg);

}

if (fds[0].revents & POLLIN)//用户直接输入终端

{

//splice()函数可以在两个文件描述符之间移动数据，且其中一个描述符必须是管道描述符

splice(0, NULL, p[1], NULL, 32768, SPLICE\_F\_MORE); //输入零拷贝到管道

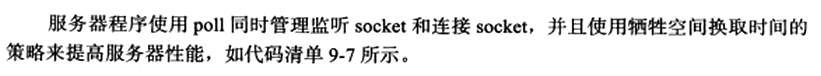
splice(p[0], NULL, sockfd, NULL, 32768, SPLICE\_F\_MORE);//管道零拷贝到socket

}

}

# 服务端

## 关键技术



## 代码分解

### 参数检查与转换与客户端一样

### 创建监听socket并命名（bind）

int ret;

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

assert(listen >= 0);

ret = bind(listenfd, (sockaddr\*)&address, sizeof(address));

assert(ret != -1);

### 创建用户数据空间

client\* USERS = new client[FD\_LIMIT];//保存用户数据，可以直接用fd获取对应的用户数据（以空间换时间）

int user\_count = 0;

### 创建poll并设置事件

pollfd fds[USER\_LIMIT + 1];//+1是listen fd

//添加并设置listen fd

fds[0].fd = listenfd;

fds[0].events = POLLIN | POLLERR;

fds[0].revents = 0;

//添加并设置用户socket

for (int i = 1; i < USER\_LIMIT + 1; i++)

{

fds[i].fd = -1;

fds[i].events = 0;

fds[i].revents = 0;

}

//开始监听listen fd

ret = listen(listenfd, 5);

assert(ret != -1);

### 监听输入与socket端口并处理事件

while (1)

{

//返回就绪（可读、可写、异常）的文件描述符的总数，返回-1表示失败并设置了errno

ret = poll(fds, USER\_LIMIT + 1, -1);

if (ret < 0)

{

printf("poll failure!\n");

break;

}

else if (ret == 0)

{

continue;

}

else

{

//轮询每个文件描述符与事件

for (int i = 0; i < user\_count + 1; i++)

{

//监听端口有新的连接

if ((fds[i].fd == listenfd) && (fds[i].revents & POLLIN))

{

//从accept队列中取出一个socket连接

int clientfd;

sockaddr\_in clientAddr;

bzero(&clientAddr, sizeof(clientAddr));

socklen\_t length = (socklen\_t)sizeof(clientAddr);

clientfd = accept(listenfd, (sockaddr\*)&clientAddr, &length);

//失败时返回-1并设置errno

if (clientfd < 0)

{

printf("errno:&d\n", errno);

continue;

}

//用户数量超出限制时发送错误信息后关闭连接

if (user\_count >= USER\_LIMIT)

{

const char\* info = "Sorry,too many users\n";

printf("%s", info);

send(clientfd, info, sizeof(info), 0);

close(clientfd);

continue;

}

//新增一个用户

user\_count++;

USERS[clientfd].address = clientAddr;

fds[user\_count].fd = clientfd;

fds[user\_count].events = POLLIN | POLLRDHUP | POLLERR;

fds[user\_count].revents = 0;

printf("comes a new user,now have %d users\n", user\_count);

}

//fds[i]发生错误

else if (fds[i].revents & POLLERR)

{

printf("get an error from %d\n", fds[i].fd);

char errors[100];

memset(errors, '\0', sizeof(errors));

socklen\_t length = sizeof(errors);

//取出对应socket的错误信息并清除错误，失败返回-1

if (getsockopt(fds[i].fd, SOL\_SOCKET, SO\_ERROR, &errors, &length) < 0)

{

printf("getsockopt failed,errno:%d\n", errno);

}

printf("error:%s\n", errors);

continue;

}

//客户端断开连接

else if (fds[i].revents & POLLRDHUP)

{

USERS[fds[i].fd] = USERS[fds[user\_count].fd];

printf("user %d left\n", fds[i].fd);

close(fds[i].fd);

fds[i].fd = fds[user\_count].fd;

user\_count--;

i--;

}

//客户端发送了信息

else if (fds[i].revents & POLLIN)

{

int clientfd = fds[i].fd;

memset(USERS[clientfd].buf, '\0', BUF\_SIZE);

//接收信息保存到该用户对应的buf

ret = recv(clientfd, USERS[clientfd].buf, BUF\_SIZE, 0);

if (ret < 0)//如果不是EAGAIN错误就关闭该客户端连接

{

if (errno != EAGAIN)

{

USERS[fds[i].fd] = USERS[fds[user\_count].fd];

printf("user %d left for error:%d\n", fds[i].fd, errno);

close(fds[i].fd);

fds[i].fd = fds[user\_count].fd;

user\_count--;

i--;

}

}

else if (ret == 0)

{

printf("code should not come to here\n");

}

//设置其它用户的可写事件并设置write指针指向发送消息的用户buf

else

{

printf("Get %d bytes from %d: %s ", ret, fds[i].fd, USERS[fds[i].fd].buf);

for (int j = 1; j < user\_count + 1; j++)

{

if (i == j)

continue;

fds[j].events |= ~POLLIN;

fds[j].events |= POLLOUT;

USERS[fds[j].fd].write = USERS[fds[i].fd].buf;

}

}

}

//挨个发送接收到的信息

else if (fds[i].revents & POLLOUT)

{

if (!USERS[fds[i].fd].write)

{

continue;

}

ret = send(fds[i].fd, USERS[fds[i].fd].write, strlen(USERS[fds[i].fd].write), 0);

if (ret < 0)

{

if (errno != EAGAIN)

{

USERS[fds[i].fd] = USERS[fds[user\_count].fd];

printf("user %d left for error:%d\n", fds[i].fd, errno);

close(fds[i].fd);

fds[i].fd = fds[user\_count].fd;

user\_count--;

i--;

}

}

//回复events

USERS[fds[i].fd].write = NULL;

fds[i].events |= POLLIN;

fds[i].events |= ~POLLOUT;

}

}

}

}

# Reference

1. Linux高性能服务器编程
2. <https://www.cnblogs.com/anker/p/3261006.html>