在前面的章节**网络编程（3）中演示示例**中已经介绍过TCP最简单的代码流程，本章节在此基础上进行拓展，先改造成服务端能不断接收客户端的消息，客户端也能反复发消息给服务端；之后，对服务端继续改进，简单使用线程处理并发的每一个客户端的一段时间的连接请求，断开后关闭当前线程；最后使用地址转换相关函数，识别对端用户的协议地址结构。

**1、TCP示例改进**

这里直接在前面TCP示例的基础上，服务端能够不断的接收客户端的消息，客户端也能不间断读取控制台输入的向服务端发送消息。代码也添加了出错的机制处理。

1. 服务端

注意服务端使用的函数:

**accept阻塞等待客户端的连接，返回一个接收客户端的已连接的套接字，并且能够获取对端的套接字协议地址结构（也可以不需要，指定参数为NULL）**；

**read阻塞等待过程中，若接收的长度为0，说明对端关闭了连接**。

见前述站章节。。。。。。[网络编程（4）socket主要函数接口、跨平台处理](https://blog.csdn.net/wanggao_1990/article/details/107884421)

#include <stdio.h>

#include <sys/socket.h>

#include <arpa/inet.h> // sockaddr\_in, inet\_addr

#include <unistd.h> // close

#include <cstring> // memset, bezro

#include <errno.h>

constexpr const char\* SRV\_ADDR = "127.0.0.1";

constexpr int SRV\_PORT = 8080;

int main()

{

/// 1、创建socket

int socket\_fd = socket(AF\_INET, SOCK\_STREAM, 0); // tcp

if(socket\_fd == -1) {

printf("%s: create socket failed. %s\n", \_\_func\_\_, strerror(errno));

return 1;

}

else {

printf("%s: create socket (fd = %d) success.\n", \_\_func\_\_, socket\_fd);

}

/// 2、绑定到本地端口

sockaddr\_in servaddr;

servaddr.sin\_family = AF\_INET;

servaddr.sin\_addr.s\_addr = inet\_addr(SRV\_ADDR);

servaddr.sin\_port = htons(SRV\_PORT);

int ret = ::bind(socket\_fd, (const sockaddr\*)&servaddr, sizeof(servaddr));

if(ret == -1) {

printf("%s: bind %s:%d failed. %s\n",

\_\_func\_\_, SRV\_ADDR, SRV\_PORT, strerror(errno));

return 1;

}

else {

printf("%s: bind %s:%d success.\n", \_\_func\_\_, SRV\_ADDR, SRV\_PORT);

}

/// 3、监听

ret = ::listen(socket\_fd, 5);

if(ret == -1) {

printf("%s: listen failed. %s\n", \_\_func\_\_, strerror(errno));

return 1;

}

else {

printf("%s: listening ...\n", \_\_func\_\_);

}

// 4、等待客户端连接

// sockaddr\_in clientaddr;

// socklen\_t sock\_len = sizeof(clientaddr);

// int sock\_id = ::accept(socket\_fd, (sockaddr\*)&clientaddr, &sock\_len);

int sock\_id = ::accept(socket\_fd, NULL, NULL); //不需要 对端地址结构

if(sock\_id < 0) {

printf("%s: accept error. %s\n", \_\_func\_\_, strerror(errno));

return 1;

}

// 5、持续回话，回显

int len;

char buf[1024];

while(true)

{

len = read(sock\_id, buf, sizeof(buf));

if(len < 0) {

printf("%s: recv failed. %s\n", \_\_func\_\_, strerror(errno));

break;

}

else if(len == 0) { // tcp 读取长度为0,表示对端关闭连接

printf("%s: client socket %d disconnet. recv len = 0. \n",\_\_func\_\_, sock\_id);

break;

}

buf[len]=0;

printf("recv from socket %d (%d): %s\n", sock\_id, len, buf);

// echo

len = ::write(sock\_id, buf, strlen(buf));

if(len < 0) {

printf("%s: send failed. %s\n", \_\_func\_\_, strerror(errno));

break;

}

}

::close(sock\_id);

/// 6、关闭连接

::close(socket\_fd);

}

1. 客户端

#include <stdio.h>

#include <sys/socket.h>

#include <arpa/inet.h> // sockaddr\_in, inet\_addr

#include <unistd.h> // close

#include <cstring> // memset, bezro

#include <errno.h>

constexpr const char\* SRV\_ADDR = "127.0.0.1";

constexpr int SRV\_PORT = 8080;

int main()

{

/// 1、创建socket

int socket\_fd = socket(AF\_INET,SOCK\_STREAM, 0); // tcp

if(socket\_fd == -1) {

printf("%s: create socket failed. %s\n", \_\_func\_\_, strerror(errno));

return 1;

}

else {

printf("%s: create socket (fd = %d) success.\n", \_\_func\_\_, socket\_fd);

}

/// 2、连接到服务端

sockaddr\_in servaddr;

servaddr.sin\_family = AF\_INET;

inet\_pton(AF\_INET,SRV\_ADDR, &servaddr.sin\_addr);

servaddr.sin\_port = htons(SRV\_PORT);

int ret = ::connect(socket\_fd, (sockaddr\*)&servaddr, sizeof(servaddr));

if(ret < 0) {

printf("%s: connect failed. %s\n", \_\_func\_\_, strerror(errno));

return 1;

}else{

printf("%s: connect %s:%d success.\n", \_\_func\_\_, SRV\_ADDR, SRV\_PORT);

}

// 3、 发送到服务端

int len ;

char buf[1024];

while (1)

{

printf("%s: send: ", \_\_func\_\_);

fgets(buf, sizeof(buf), stdin); // 等待用户输入

if (strcmp(buf, "exit \n") == 0)

break;

// 发送

len = ::write(socket\_fd, buf, strlen(buf));

if(len < 0){

printf("%s: send failed.Exit. %s\n", \_\_func\_\_, strerror(errno));

break;

}

// echo

len = ::read(socket\_fd, buf, sizeof(buf));

if(len < 0){

printf("%s: recv failed. %s\n", \_\_func\_\_, strerror(errno));

break;

}else if(len == 0){

printf("%s: server closed.\n", \_\_func\_\_);

break;

}

printf("%s: recv (%d): %s\n", \_\_func\_\_, len, buf);

}

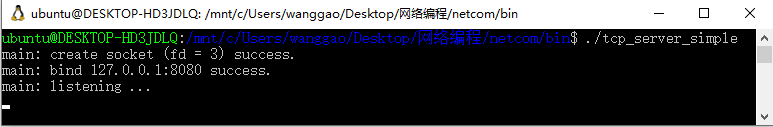
/// 4、关闭连接

::close(socket\_fd);

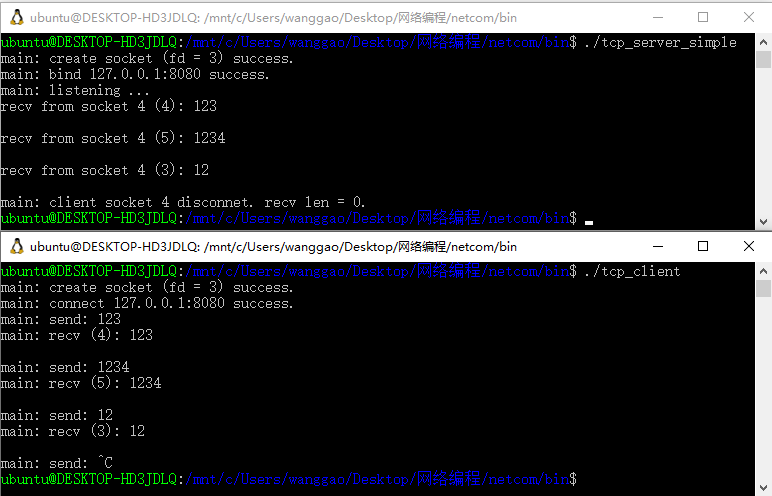
}

1. 测试结果

先启动服务器，陈成功进入等到客户端连接状态

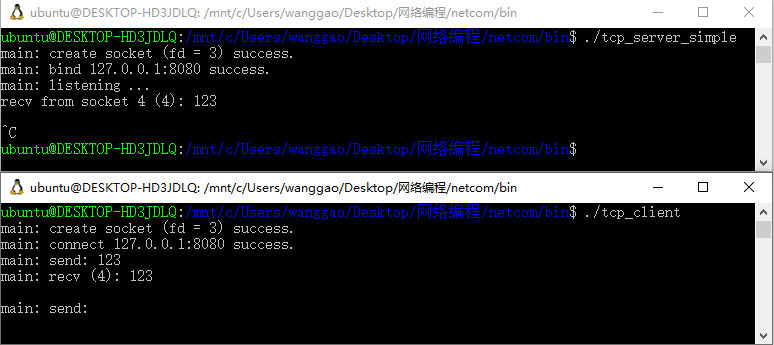


之后启动客户端，并多次输入内容并发送，之后Ctrl+C或输入exit退出客户端。

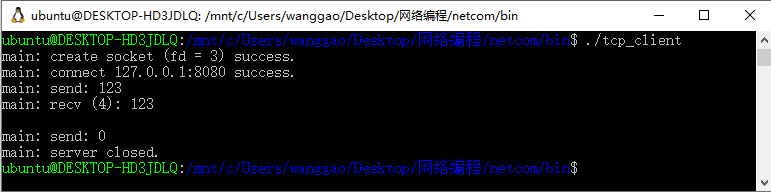


当客户端退出时，服务端read函数的返回值为0，并退出服务端。

当连接建立后，服务端先退出时，客户端并没有提示退出，如下：



因为客户端当前阻塞再fgets函数处，输入任意字符，客户端read函数返回0，服务端断开连接，客户端退出。



**2、TCP并发服务器**

继续改进服务器程序，使得服务端能够不断为不同客户端服务，最简单的做法就是直接将accept代码移入while循环中，和客户端交互结束后关闭当前已连接套接字，等待新的客户端连接。给出主体代码：

int main()

{

/// 1、创建socket

/// 2、绑定到本地端口

/// 3、监听

int len;

char buf[1024];

while(true)

{ // 5、等待连接

int sock\_id = ::accept(socket\_fd, NULL, NULL); //不需要 对端地址结构

if(sock\_id < 0) {

break 1;

}

while(true) {

len = read(sock\_id, buf, sizeof(buf));

if(len < 0) {

break;

}

else if(len == 0) { // tcp 读取长度为0,表示对端关闭连接

break;

}

// echo

len = ::write(sock\_id, buf, strlen(buf));

}

::close(sock\_id);

}

/// 5、关闭连接

::close(socket\_fd);

}

经过改进后的服务器是一个迭代服务器，对于一个简单的TCP连接请求(例如获取一次时间)，这样实现就足够。然而，当服务的一个客户请求时间花费较长或者交互次数较多时，并不希望整个服务器被单个客户长时间占，而是希望同时服务多个客户。

因此，这里简单使用多线程的方法来实现并发，对每一个客户连接创建一个线程来服务客户，能够不断接收新的TCP客户端连接。

这里直接使用c++11 thread库。首先添加的客户服务线程函数：

#include <thread>

void client\_service(int sock\_id)

{

int len;

char buf[1024];

while (1)

{

len = read(sock\_id, buf, sizeof(buf));

if(len < 0){

printf("%s: recv failed. %s\n", \_\_func\_\_, strerror(errno));

break;

}else if(len == 0){ // tcp 读取长度为0,表示对端关闭连接

printf("%s: client socket %d disconnet. recv len = 0.\n", \_\_func\_\_, sock\_id);

break;

}

buf[len] = 0; // //避免接收数据较上一次短，导致输出显示错误

if (strcmp(buf, "exit\n") == 0)break;

printf("recv from socket %d (%d): %s\n", sock\_id, len, buf );

// echo

len = ::write(sock\_id, buf, strlen(buf));

if(len < 0){

printf("%s: send failed. %s\n", \_\_func\_\_, strerror(errno));

break;

}

}

::close(sock\_id);

}

while(true){

// 等待连接

sockaddr\_in clientaddr; // 可以满足IPv4和IPv6的内存需求

socklen\_t sock\_len = 0;

int sock\_id = ::accept(socket\_fd, (sockaddr\*)&clientaddr,&sock\_len);

if(sock\_id < 0){

printf("%s: accept error. %s\n", \_\_func\_\_, strerror(errno));

return 1;

}

// 为当前连接创建一个线程

std::thread(client\_service,sock\_id,clientaddr,sock\_len).detach();

}

/// 4、关闭连接

::close(socket\_fd);

}

再继续修改while部分的代码，对新创建的线程detach后为客户端进行服务。

while(true){

// 等待连接

sockaddr\_storage\_in clientaddr; // 可以满足IPv4和IPv6的内存需求

socklen\_t sock\_len = sizeof(clientaddr);

int sock\_id = ::accept(socket\_fd, (sockaddr\*)&clientaddr,&sock\_len);

if(sock\_id < 0){

printf("%s: accept error. %s\n", \_\_func\_\_, strerror(errno));

return 1;

}

// 为当前连接创建一个线程

std::thread(client\_service).detach();

}

至此，上述代码就有一个典型的并发服务器程序的轮廓了。服务端可以接受多个客户端的请求，互不干扰。下一节中同时给出多客户端连接且同时显示客户端信息的演示截图。

**3、TCP多客户端地址**

在UDP套接字编程已经给出了通过recvfrom()函数获取对端的地址结构信息。在TCP编程中，服务端可以通过accept()函数获取客户端信息，还可以将accept()函数返回值——客户端已连接套接字作为getpeername()函数参数获取客户端信息。**相关获取地址信息的代码，另开一章，连接。**

对于服务端，我们可以直接给出使用getpeername方法获取客户端信息，修改客户服务线程函数void client\_service(int sock\_id)，添加代码

void client\_service(int sock\_id)

{

sockaddr\_storage clientaddr;

sock\_len = sizeof(clientaddr); // 128

getpeername(sock\_id, (sockaddr\*)&clientaddr, &sock\_len);

char client\_ip[INET6\_ADDRSTRLEN]; // 足够存IPv4和IPv6地址

int port;

if(sock\_len == sizeof(sockaddr\_in)){ // 128-> 16

sockaddr\_in \*cliAddr = (sockaddr\_in \*)&clientaddr;

//char\* client\_ip = inet\_ntoa(cliAddr->sin\_addr);

inet\_ntop(AF\_INET, &cliAddr->sin\_addr, client\_ip, sock\_len);

port = ntohs(cliAddr->sin\_port);

printf("new socket id = %d, clent %s:%d\n", sock\_id, client\_ip, port);

}

else if(sock\_len == sizeof(sockaddr\_in6)){ // 128 -> 28

sockaddr\_in6 \*cliAddr = (sockaddr\_in6 \*)&clientaddr;

inet\_ntop(AF\_INET6, &cliAddr->sin6\_addr, client\_ip, sock\_len);

port = ntohs(cliAddr->sin6\_port);

printf("new socket id = %d, clent %s:%d\n", sock\_id, client\_ip, port);

}

int len;

char buf[1024];

//

// 后面保持不变，省略

///

}

对于客户端，不需要外获取服务端的地址信息的。TCP客户端亦不需要，但是connect之后，我们可以利用这个已连接的套接字，使用getpeername获取服务端地址信息，使用getsockname获取本地的地址信息，两个函数代码使用方法全部，仅函数名不同而已。

这里简写，以INET\_IPV4协议为例

int ret = ::connect(socket\_fd, (sockaddr\*)&servaddr, sizeof(servaddr));

sockaddr\_in localaddr;

socklen\_t sock\_len = sizeof(localaddr); // 必须给初值

// 必须给初始长度, getsockname会根据实际情况重新赋值，这里 16 -> 16

// 若出现初始值<0， 会报错，提示 Invalid argument

ret = getsockname(socket\_fd, (sockaddr\*)&localaddr, &sock\_len);

if(ret == -1){

printf("%s: getsockname failed. %s\n", \_\_func\_\_, strerror(errno));

}else{

printf("%s: local addr: %s:%d\n",

\_\_func\_\_, inet\_ntoa(localaddr.sin\_addr), ntohs(localaddr.sin\_port));

}

}

分别运行服务端，连接多个客户端，客户端交替发送信息，最后断开。演示结果如下：

