# 340、第一个网络通讯程序

## 一、网络通讯的流程



## 二、demo1.cpp

/\*

\* 程序名：demo1.cpp，此程序用于演示socket的客户端

\*/

#include <iostream>

#include <cstdio>

#include <cstring>

#include <cstdlib>

#include <unistd.h>

#include <netdb.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <arpa/inet.h>

using namespace std;

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

{

if (argc!=3)

{

cout << "Using:./demo1 服务端的IP 服务端的端口\nExample:./demo1 192.168.101.139 5005\n\n";

return -1;

}

// 第1步：创建客户端的socket。

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

if (sockfd==-1)

{

perror("socket"); return -1;

}

// 第2步：向服务器发起连接请求。

struct hostent\* h; // 用于存放服务端IP的结构体。

if ( (h = gethostbyname(argv[1])) == 0 ) // 把字符串格式的IP转换成结构体。

{

cout << "gethostbyname failed.\n" << endl; close(sockfd); return -1;

}

struct sockaddr\_in servaddr; // 用于存放服务端IP和端口的结构体。

memset(&servaddr,0,sizeof(servaddr));

servaddr.sin\_family = AF\_INET;

memcpy(&servaddr.sin\_addr,h->h\_addr,h->h\_length); // 指定服务端的IP地址。

servaddr.sin\_port = htons(atoi(argv[2])); // 指定服务端的通信端口。

if (connect(sockfd,(struct sockaddr \*)&servaddr,sizeof(servaddr))!=0) // 向服务端发起连接清求。

{

perror("connect"); close(sockfd); return -1;

}

// 第3步：与服务端通讯，客户发送一个请求报文后等待服务端的回复，收到回复后，再发下一个请求报文。

char buffer[1024];

for (int ii=0;ii<3;ii++) // 循环3次，将与服务端进行三次通讯。

{

int iret;

memset(buffer,0,sizeof(buffer));

sprintf(buffer,"这是第%d个超级女生，编号%03d。",ii+1,ii+1); // 生成请求报文内容。

// 向服务端发送请求报文。

if ( (iret=send(sockfd,buffer,strlen(buffer),0))<=0)

{

perror("send"); break;

}

cout << "发送：" << buffer << endl;

memset(buffer,0,sizeof(buffer));

// 接收服务端的回应报文，如果服务端没有发送回应报文，recv()函数将阻塞等待。

if ( (iret=recv(sockfd,buffer,sizeof(buffer),0))<=0)

{

cout << "iret=" << iret << endl; break;

}

cout << "接收：" << buffer << endl;

sleep(1);

}

// 第4步：关闭socket，释放资源。

close(sockfd);

}

## 三、demo2.cpp

/\*

\* 程序名：demo2.cpp，此程序用于演示socket通信的服务端

\*/

#include <iostream>

#include <cstdio>

#include <cstring>

#include <cstdlib>

#include <unistd.h>

#include <netdb.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <arpa/inet.h>

using namespace std;

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

{

if (argc!=2)

{

cout << "Using:./demo2 通讯端口\nExample:./demo2 5005\n\n"; // 端口大于1024，不与其它的重复。

cout << "注意：运行服务端程序的Linux系统的防火墙必须要开通5005端口。\n";

cout << " 如果是云服务器，还要开通云平台的访问策略。\n\n";

return -1;

}

// 第1步：创建服务端的socket。

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

if (listenfd==-1)

{

perror("socket"); return -1;

}

// 第2步：把服务端用于通信的IP和端口绑定到socket上。

struct sockaddr\_in servaddr; // 用于存放服务端IP和端口的数据结构。

memset(&servaddr,0,sizeof(servaddr));

servaddr.sin\_family = AF\_INET; // 指定协议。

servaddr.sin\_addr.s\_addr = htonl(INADDR\_ANY); // 服务端任意网卡的IP都可以用于通讯。

servaddr.sin\_port = htons(atoi(argv[1])); // 指定通信端口，普通用户只能用1024以上的端口。

// 绑定服务端的IP和端口。

if (bind(listenfd,(struct sockaddr \*)&servaddr,sizeof(servaddr)) != 0 )

{

perror("bind"); close(listenfd); return -1;

}

// 第3步：把socket设置为可连接（监听）的状态。

if (listen(listenfd,5) != 0 )

{

perror("listen"); close(listenfd); return -1;

}

// 第4步：受理客户端的连接请求，如果没有客户端连上来，accept()函数将阻塞等待。

int clientfd=accept(listenfd,0,0);

if (clientfd==-1)

{

perror("accept"); close(listenfd); return -1;

}

cout << "客户端已连接。\n";

// 第5步：与客户端通信，接收客户端发过来的报文后，回复ok。

char buffer[1024];

while (true)

{

int iret;

memset(buffer,0,sizeof(buffer));

// 接收客户端的请求报文，如果客户端没有发送请求报文，recv()函数将阻塞等待。

// 如果客户端已断开连接，recv()函数将返回0。

if ( (iret=recv(clientfd,buffer,sizeof(buffer),0))<=0)

{

cout << "iret=" << iret << endl; break;

}

cout << "接收：" << buffer << endl;

strcpy(buffer,"ok"); // 生成回应报文内容。

// 向客户端发送回应报文。

if ( (iret=send(clientfd,buffer,strlen(buffer),0))<=0)

{

perror("send"); break;

}

cout << "发送：" << buffer << endl;

}

// 第6步：关闭socket，释放资源。

close(listenfd); // 关闭服务端用于监听的socket。

close(clientfd); // 关闭客户端连上来的socket。

}

# 341、基于Linux的文件操作

## 一、demo3.cpp

// demo3.cpp，本程序演示了Linux底层文件的操作-创建文件并写入数据。

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <fcntl.h>

#include <unistd.h>

using namespace std;

int main()

{

int fd; // 定义一个文件描述符/文件句柄。

// 打开文件，注意，如果创建后的文件没有权限，可以手工授权chmod 777 data.txt。

fd=open("data.txt",O\_CREAT|O\_RDWR|O\_TRUNC);

if (fd==-1)

{

perror("open(data.txt)"); return -1;

}

printf("文件描述符fd=%d\n",fd);

char buffer[1024];

strcpy(buffer,"我是一只傻傻鸟。\n");

if (write(fd,buffer,strlen(buffer))==-1) // 把数据写入文件。

{

perror("write()"); return -1;

}

close(fd); // 关闭文件。

}

## 二、demo4.cpp

// demo4.cpp，本程序演示了Linux底层文件的操作-读取文件。

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <fcntl.h>

#include <unistd.h>

int main()

{

int fd; // 定义一个文件描述符/文件句柄。

fd=open("data.txt",O\_RDONLY); // 打开文件。

if (fd==-1)

{

perror("open(data.txt)"); return -1;

}

printf("文件描述符fd=%d\n",fd);

char buffer[1024];

memset(buffer,0,sizeof(buffer));

if (read(fd,buffer,sizeof(buffer))==-1) // 从文件中读取数据。

{

perror("write()"); return -1;

}

printf("%s",buffer);

close(fd); // 关闭文件。

}

# 342、socket函数详解

## 一、什么是协议

人与人沟通的方式有很多种：书信、电话、QQ、微信。如果两个人想沟通，必须先选择一种沟通的方式，如果一方使用电话，另一方也应该使用电话，而不是书信。

协议是网络通讯的规则，是约定。

## 二、创建socket

int socket(int domain, int type, int protocol);

成功返回一个有效的socket，失败返回-1，errno被设置。

全部网络编程的函数，失败时基本上都是返回-1，errno被设置。

只要参数没填错，基本上不会失败。

不过，单个进程中创建的socket数量与受系统参数open files的限制。（ulimit -a ）

### 1）domain 通讯的协议家族

**PF\_INET IPv4互联网协议族。**

PF\_INET6 IPv6互联网协议族。

PF\_LOCAL 本地通信的协议族。

PF\_PACKET 内核底层的协议族。

PF\_IPX IPX Novell协议族。

IPv6尚未普及，其它的不常用。

### 2）type 数据传输的类型

SOCK\_STREAM 面向连接的socket：1）数据不会丢失；2）数据的顺序不会错乱；3）双向通道。

SOCK\_DGRAM 无连接的socket：1）数据可能会丢失；2）数据的顺序可能会错乱；3）传输的效率更高。

### 3）protocol 最终使用的协议

在IPv4网络协议家族中，数据传输方式为SOCK\_STREAM的协议只有IPPROTO\_TCP，数据传输方式为SOCK\_DGRAM的协议只有IPPROTO\_UDP。

本参数也可以填0。

socket(**PF\_INET**, SOCK\_STREAM, IPPROTO\_TCP); // 创建tcp的sock

socket(**PF\_INET**, SOCK\_DGRAM, IPPROTO\_UDP); // 创建udp的sock

## 三、TCP和UPD

### 1）TCP和UDP的区别

**TCP**

a）TCP面向连接，通过三次握手建立连接，四次挥手断开连接； **面试的重点**

b）TCP是可靠的通信方式，通过超时重传、数据校验等方式来确保数据无差错，不丢失，不重复，并且按序到达；

c）TCP把数据当成字节流，当网络出现波动时，连接可能出现响应延迟的问题；

d）TCP只支持点对点通信；

e）TCP报文的首部较大，为20字节；

f）TCP是全双工的可靠信道。

**UDP**

a）UDP是无连接的，即发送数据之前不需要建立连接，这种方式为UDP带来了高效的传输效率，但也导致无法确保数据的发送成功；

b）UDP以最大的速率进行传输，但不保证可靠交付，会出现丢失、重复等等问题；

c）UDP没有拥塞控制，当网络出现拥塞时，发送方不会降低发送速率；

d）UDP支持一对一，一对多，多对一和多对多的通信；

e）UDP报文的首部比较小，只有8字节；

f）UDP是不可靠信道。

### 2）TCP保证自身可靠的方式

a）数据分片：在发送端对用户数据进行分片，在接收端进行重组，由TCP确定分片的大小并控制分片和重组；

b）到达确认：接收端接收到分片数据时，根据分片的序号向对端回复一个确认包；

c）超时重发：发送方在发送分片后开始计时，若超时却没有收到对端的确认包，将会重发分片；

d）滑动窗口：TCP 中采用滑动窗口来进行传输控制，发送方可以通过滑动窗口的大小来确定应该发送多少字节的数据。当滑动窗口为 0 时，发送方不会再发送数据；

e）失序处理：TCP的接收端会把接收到的数据重新排序；

f）重复处理：如果传输的分片出现重复，TCP的接收端会丢弃重复的数据；

g）数据校验：TCP通过数据的检验和来判断数据在传输过程中是否正确。

### 3）UDP不可靠的原因

没有上述TCP的机制，如果校验和出错，UDP会将该报文丢弃。

### 4）TCP和UDP使用场景

**TCP 使用场景**

TCP实现了数据传输过程中的各种控制，适合对可靠性有要求的场景。

**UDP 使用场景**

可以容忍数据丢失的场景：

* **视频、音频等多媒体通信（即时通信）；**
* 广播信息。

### 5）UDP能实现可靠传输吗？

这是个伪命题，如果用UDP实现可靠传输，那么，应用程序必须实现重传和排序等功能，非常麻烦，还不如直接用TCP。谁能保证自己写的算法比写TCP协议的人更牛？

# 343、主机字节序与网络字节序

## 一、大端序/小端序

如果数据类型占用的内存空间大于1字节，CPU把数据存放在内存中的方式有两种：

* 大端序（Big Endian）：低位字节存放在高位，高位字节存放在低位。
* 小端序（Little Endia）：低位字节存放在低位，高位字节存放在高位。

假设从内存地址0x00000001处开始存储十六进制数0x12345678，那么：

Bit-endian（按原来顺序存储）

0x00000001           0x12

0x00000002           0x34

0x00000003           0x56

0x00000004           0x78

Little-endian（颠倒顺序储存）

0x00000001           0x78

0x00000002           0x56

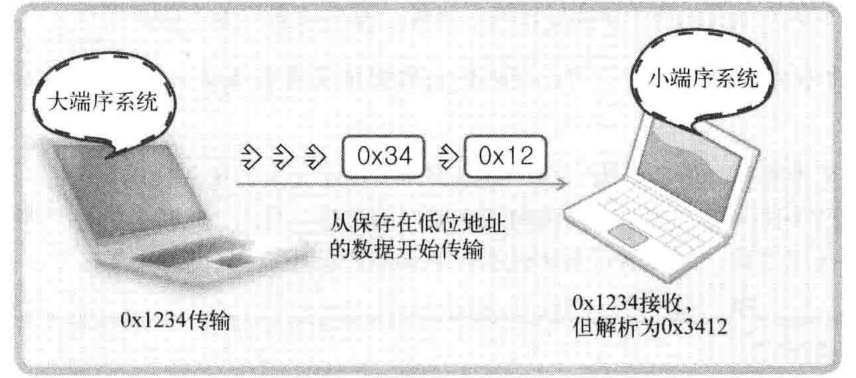
0x00000003           0x34

0x00000004           0x12

Intel系列的CPU以小端序方式保存数据，其它型号的CPU不一定。

操作文件的本质是把内存中的数据写入磁盘，在网络编程中，传输数据的本质也是把数据写入文件（socket也是文件描述符）。

这样的话，字节序不同的计算机之间传输数据，可能会出现问题。



## 二、网络字节序

为了解决不同字节序的计算机之间传输数据的问题，约定采用网络字节序（大端序）。

C语言提供了四个库函数，用于在主机字节序和网络字节序之间转换：

uint16\_t h to n s(uint16\_t hostshort); // uint16\_t 2字节的整数 unsigned short

uint32\_t htonl(uint32\_t hostlong); // uint32\_t 4字节的整数 unsigned int

uint16\_t ntohs(uint16\_t netshort);

uint32\_t n to h l(uint32\_t netlong);

h host（主机）；

to 转换；

n network（网络）；

s short（2字节，16位的整数）；

l long（4字节，32位的整数）；

## 三、IP地址和通讯端口

在计算机中，IPv4的地址用4字节的整数存放，通讯端口用2字节的整数（0-65535）存放。

例如：192.168.190.134 3232284294 255.255.255.255

192 168 190 134

大端：11000000 10101000 10111110 10000110

小端：10000110 10111110 10101000 11000000

## 四、如何处理大小端序

在网络编程中，数据收发的时候有自动转换机制，不需要程序员手动转换，只有向sockaddr\_in结体成员变量填充数据时，才需要考虑字节序的问题。

# 344、万恶的结构体

## 一、sockaddr结构体

存放协议族、端口和地址信息，客户端和connect()函数和服务端的bind()函数需要这个结构体。

struct sockaddr {

unsigned short sa\_family; // 协议族，与socket()函数的第一个参数相同，填AF\_INET。

unsigned char sa\_data[14]; // 14字节的端口和地址。

};

## 二、sockaddr\_in结构体

sockaddr结构体是为了统一地址结构的表示方法，统一接口函数，但是，操作不方便，所以定义了等价的sockaddr\_in结构体，它的大小与sockaddr相同，可以强制转换成sockaddr。

struct sockaddr\_in {

unsigned short sin\_family; // 协议族，与socket()函数的第一个参数相同，填AF\_INET。

unsigned short sin\_port; // 16位端口号，大端序。用htons(整数的端口)转换。

struct in\_addr sin\_addr; // IP地址的结构体。192.168.101.138

unsigned char sin\_zero[8]; // 未使用，为了保持与struct sockaddr一样的长度而添加。

};

struct in\_addr { // IP地址的结构体。

unsigned int s\_addr; // 32位的IP地址，大端序。

};

## 三、gethostbyname函数

根据域名/主机名/字符串IP获取大端序IP，用于网络通讯的客户端程序中。

struct hostent \*gethostbyname(const char \*name);

struct hostent {

char \*h\_name; // 主机名。

char \*\*h\_aliases; // 主机所有别名构成的字符串数组，同一IP可绑定多个域名。

short h\_addrtype; // 主机IP地址的类型，例如IPV4（AF\_INET）还是IPV6。

short h\_length; // 主机IP地址长度，IPV4地址为4，IPV6地址则为16。

char \*\*h\_addr\_list; // 主机的ip地址，以网络字节序存储。

};

#define h\_addr h\_addr\_list[0] // for backward compatibility.

转换后，用以下代码把大端序的地址复制到sockaddr\_in结构体的sin\_addr成员中。

memcpy(&servaddr.sin\_addr,h->h\_addr,h->h\_length);

## 四、字符串IP与大端序IP的转换

C语言提供了几个库函数，用于字符串格式的IP和大端序IP的互相转换，用于网络通讯的服务端程序中。

typedef unsigned int in\_addr\_t; // 32位大端序的IP地址。

// 把字符串格式的IP转换成大端序的IP，转换后的IP赋给sockaddr\_in.in\_addr.s\_addr。

in\_addr\_t inet\_addr(const char \*cp);

// 把字符串格式的IP转换成大端序的IP，转换后的IP将填充到sockaddr\_in.in\_addr成员。

int inet\_aton(const char \*cp, struct in\_addr \*inp);

// 把大端序IP转换成字符串格式的IP，用于在服务端程序中解析客户端的IP地址。

char \*inet\_ntoa(struct in\_addr in);

## 五、demo5.cpp

/\*

\* 程序名：demo5.cpp，此程序用于演示socket的客户端

\*/

#include <iostream>

#include <cstdio>

#include <cstring>

#include <cstdlib>

#include <unistd.h>

#include <netdb.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <arpa/inet.h>

using namespace std;

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

{

if (argc!=3)

{

cout << "Using:./demo5 服务端的IP 服务端的端口\nExample:./demo5 192.168.101.138 5005\n\n";

return -1;

}

// 第1步：创建客户端的socket。

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

if (sockfd==-1)

{

perror("socket"); return -1;

}

// 第2步：向服务器发起连接请求。

struct sockaddr\_in servaddr; // 用于存放协议、端口和IP地址的结构体。

memset(&servaddr,0,sizeof(servaddr));

servaddr.sin\_family = AF\_INET; // ①协议族，固定填AF\_INET。

servaddr.sin\_port = htons(atoi(argv[2])); // ②指定服务端的通信端口。

struct hostent\* h; // 用于存放服务端IP地址(大端序)的结构体的指针。

if ( (h = gethostbyname(argv[1])) == nullptr ) // 把域名/主机名/字符串格式的IP转换成结构体。

{

cout << "gethostbyname failed.\n" << endl; close(sockfd); return -1;

}

memcpy(&servaddr.sin\_addr,h->h\_addr,h->h\_length); // ③指定服务端的IP(大端序)。

//servaddr.sin\_addr.s\_addr=inet\_addr(argv[1]); // ③指定服务端的IP，只能用IP，不能用域名和主机名。

if (connect(sockfd,(struct sockaddr \*)&servaddr,sizeof(servaddr))==-1) // 向服务端发起连接清求。

{

perror("connect"); close(sockfd); return -1;

}

// 第3步：与服务端通讯，客户发送一个请求报文后等待服务端的回复，收到回复后，再发下一个请求报文。

char buffer[1024];

for (int ii=0;ii<10;ii++) // 循环3次，将与服务端进行三次通讯。

{

int iret;

memset(buffer,0,sizeof(buffer));

sprintf(buffer,"这是第%d个超级女生，编号%03d。",ii+1,ii+1); // 生成请求报文内容。

// 向服务端发送请求报文。

if ( (iret=send(sockfd,buffer,strlen(buffer),0))<=0)

{

perror("send"); break;

}

cout << "发送：" << buffer << endl;

memset(buffer,0,sizeof(buffer));

// 接收服务端的回应报文，如果服务端没有发送回应报文，recv()函数将阻塞等待。

if ( (iret=recv(sockfd,buffer,sizeof(buffer),0))<=0)

{

cout << "iret=" << iret << endl; break;

}

cout << "接收：" << buffer << endl;

sleep(1);

}

// 第4步：关闭socket，释放资源。

close(sockfd);

}

## 六、demo5.cpp

/\*

\* 程序名：demo6.cpp，此程序用于演示socket通信的服务端

\*/

#include <iostream>

#include <cstdio>

#include <cstring>

#include <cstdlib>

#include <unistd.h>

#include <netdb.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <arpa/inet.h>

using namespace std;

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

{

if (argc!=2)

{

cout << "Using:./demo6 通讯端口\nExample:./demo6 5005\n\n"; // 端口大于1024，不与其它的重复。

cout << "注意：运行服务端程序的Linux系统的防火墙必须要开通5005端口。\n";

cout << " 如果是云服务器，还要开通云平台的访问策略。\n\n";

return -1;

}

// 第1步：创建服务端的socket。

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

if (listenfd==-1)

{

perror("socket"); return -1;

}

// 第2步：把服务端用于通信的IP和端口绑定到socket上。

struct sockaddr\_in servaddr; // 用于存放协议、端口和IP地址的结构体。

memset(&servaddr,0,sizeof(servaddr));

servaddr.sin\_family=AF\_INET; // ①协议族，固定填AF\_INET。

servaddr.sin\_port=htons(atoi(argv[1])); // ②指定服务端的通信端口。

servaddr.sin\_addr.s\_addr=htonl(INADDR\_ANY); // ③如果操作系统有多个IP，全部的IP都可以用于通讯。

//servaddr.sin\_addr.s\_addr=inet\_addr("192.168.101.138"); // ③指定服务端用于通讯的IP(大端序)。

// 绑定服务端的IP和端口。

if (bind(listenfd,(struct sockaddr \*)&servaddr,sizeof(servaddr))==-1)

{

perror("bind"); close(listenfd); return -1;

}

// 第3步：把socket设置为可连接（监听）的状态。

if (listen(listenfd,5) == -1 )

{

perror("listen"); close(listenfd); return -1;

}

// 第4步：受理客户端的连接请求，如果没有客户端连上来，accept()函数将阻塞等待。

int clientfd=accept(listenfd,0,0);

if (clientfd==-1)

{

perror("accept"); close(listenfd); return -1;

}

cout << "客户端已连接。\n";

// 第5步：与客户端通信，接收客户端发过来的报文后，回复ok。

char buffer[1024];

while (true)

{

int iret;

memset(buffer,0,sizeof(buffer));

// 接收客户端的请求报文，如果客户端没有发送请求报文，recv()函数将阻塞等待。

// 如果客户端已断开连接，recv()函数将返回0。

if ( (iret=recv(clientfd,buffer,sizeof(buffer),0))<=0)

{

cout << "iret=" << iret << endl; break;

}

cout << "接收：" << buffer << endl;

strcpy(buffer,"ok"); // 生成回应报文内容。

// 向客户端发送回应报文。

if ( (iret=send(clientfd,buffer,strlen(buffer),0))<=0)

{

perror("send"); break;

}

cout << "发送：" << buffer << endl;

}

// 第6步：关闭socket，释放资源。

close(listenfd); // 关闭服务端用于监听的socket。

close(clientfd); // 关闭客户端连上来的socket。

}

# 345、封装socket

## 一、demo7.cpp

/\*

\* 程序名：demo7.cpp，此程序用于演示封装socket通讯的客户端

\*/

#include <iostream>

#include <cstdio>

#include <cstring>

#include <cstdlib>

#include <unistd.h>

#include <netdb.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <arpa/inet.h>

using namespace std;

class ctcpclient // TCP通讯的客户端类。

{

private:

int m\_clientfd; // 客户端的socket，-1表示未连接或连接已断开；>=0表示有效的socket。

string m\_ip; // 服务端的IP/域名。

unsigned short m\_port; // 通讯端口。

public:

ctcpclient():m\_clientfd(-1) {}

// 向服务端发起连接请求，成功返回true，失败返回false。

bool connect(const string &in\_ip,const unsigned short in\_port)

{

if (m\_clientfd!=-1) return false; // 如果socket已连接，直接返回失败。

m\_ip=in\_ip; m\_port=in\_port; // 把服务端的IP和端口保存到成员变量中。

// 第1步：创建客户端的socket。

if ( (m\_clientfd = socket(AF\_INET,SOCK\_STREAM,0))==-1) return false;

// 第2步：向服务器发起连接请求。

struct sockaddr\_in servaddr; // 用于存放协议、端口和IP地址的结构体。

memset(&servaddr,0,sizeof(servaddr));

servaddr.sin\_family = AF\_INET; // ①协议族，固定填AF\_INET。

servaddr.sin\_port = htons(m\_port); // ②指定服务端的通信端口。

struct hostent\* h; // 用于存放服务端IP地址(大端序)的结构体的指针。

if ((h=gethostbyname(m\_ip.c\_str()))==nullptr ) // 把域名/主机名/字符串格式的IP转换成结构体。

{

::close(m\_clientfd); m\_clientfd=-1; return false;

}

memcpy(&servaddr.sin\_addr,h->h\_addr,h->h\_length); // ③指定服务端的IP(大端序)。

// 向服务端发起连接清求。

if (::connect(m\_clientfd,(struct sockaddr \*)&servaddr,sizeof(servaddr))==-1)

{

::close(m\_clientfd); m\_clientfd=-1; return false;

}

return true;

}

// 向服务端发送报文，成功返回true，失败返回false。

bool send(const string &buffer) // buffer不要用const char \*

{

if (m\_clientfd==-1) return false; // 如果socket的状态是未连接，直接返回失败。

if ((::send(m\_clientfd,buffer.data(),buffer.size(),0))<=0) return false;

return true;

}

// 接收服务端的报文，成功返回true，失败返回false。

// buffer-存放接收到的报文的内容，maxlen-本次接收报文的最大长度。

bool recv(string &buffer,const size\_t maxlen)

{ // 如果直接操作string对象的内存，必须保证：1)不能越界；2）操作后手动设置数据的大小。

buffer.clear(); // 清空容器。

buffer.resize(maxlen); // 设置容器的大小为maxlen。

int readn=::recv(m\_clientfd,&buffer[0],buffer.size(),0); // 直接操作buffer的内存。

if (readn<=0) { buffer.clear(); return false; }

buffer.resize(readn); // 重置buffer的实际大小。

return true;

}

// 断开与服务端的连接。

bool close()

{

if (m\_clientfd==-1) return false; // 如果socket的状态是未连接，直接返回失败。

::close(m\_clientfd);

m\_clientfd=-1;

return true;

}

~ctcpclient(){ close(); }

};

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

{

if (argc!=3)

{

cout << "Using:./demo7 服务端的IP 服务端的端口\nExample:./demo7 192.168.101.138 5005\n\n";

return -1;

}

ctcpclient tcpclient;

if (tcpclient.connect(argv[1],atoi(argv[2]))==false) // 向服务端发起连接请求。

{

perror("connect()"); return -1;

}

// 第3步：与服务端通讯，客户发送一个请求报文后等待服务端的回复，收到回复后，再发下一个请求报文。

string buffer;

for (int ii=0;ii<10;ii++) // 循环3次，将与服务端进行三次通讯。

{

buffer="这是第"+to\_string(ii+1)+"个超级女生，编号"+to\_string(ii+1)+"。";

// 向服务端发送请求报文。

if (tcpclient.send(buffer)==false)

{

perror("send"); break;

}

cout << "发送：" << buffer << endl;

// 接收服务端的回应报文，如果服务端没有发送回应报文，recv()函数将阻塞等待。

if (tcpclient.recv(buffer,1024)==false)

{

perror("recv()"); break;

}

cout << "接收：" << buffer << endl;

sleep(1);

}

}

## 二、demo8.cpp

/\*

\* 程序名：demo8.cpp，此程序用于演示封装socket通讯的服务端

\*/

#include <iostream>

#include <cstdio>

#include <cstring>

#include <cstdlib>

#include <unistd.h>

#include <netdb.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <arpa/inet.h>

using namespace std;

class ctcpserver // TCP通讯的服务端类。

{

private:

int m\_listenfd; // 监听的socket，-1表示未初始化。

int m\_clientfd; // 客户端连上来的socket，-1表示客户端未连接。

string m\_clientip; // 客户端字符串格式的IP。

unsigned short m\_port; // 服务端用于通讯的端口。

public:

ctcpserver():m\_listenfd(-1),m\_clientfd(-1) {}

// 初始化服务端用于监听的socket。

bool initserver(const unsigned short in\_port)

{

// 第1步：创建服务端的socket。

if ( (m\_listenfd=socket(AF\_INET,SOCK\_STREAM,0))==-1) return false;

m\_port=in\_port;

// 第2步：把服务端用于通信的IP和端口绑定到socket上。

struct sockaddr\_in servaddr; // 用于存放协议、端口和IP地址的结构体。

memset(&servaddr,0,sizeof(servaddr));

servaddr.sin\_family=AF\_INET; // ①协议族，固定填AF\_INET。

servaddr.sin\_port=htons(m\_port); // ②指定服务端的通信端口。

servaddr.sin\_addr.s\_addr=htonl(INADDR\_ANY); // ③如果操作系统有多个IP，全部的IP都可以用于通讯。

// 绑定服务端的IP和端口（为socket分配IP和端口）。

if (bind(m\_listenfd,(struct sockaddr \*)&servaddr,sizeof(servaddr))==-1)

{

close(m\_listenfd); m\_listenfd=-1; return false;

}

// 第3步：把socket设置为可连接（监听）的状态。

if (listen(m\_listenfd,5) == -1 )

{

close(m\_listenfd); m\_listenfd=-1; return false;

}

return true;

}

// 受理客户端的连接（从已连接的客户端中取出一个客户端），

// 如果没有已连接的客户端，accept()函数将阻塞等待。

bool accept()

{

struct sockaddr\_in caddr; // 客户端的地址信息。

socklen\_t addrlen=sizeof(caddr); // struct sockaddr\_in的大小。

if ((m\_clientfd=::accept(m\_listenfd,(struct sockaddr \*)&caddr,&addrlen))==-1) return false;

m\_clientip=inet\_ntoa(caddr.sin\_addr); // 把客户端的地址从大端序转换成字符串。

return true;

}

// 获取客户端的IP(字符串格式)。

const string & clientip() const

{

return m\_clientip;

}

// 向对端发送报文，成功返回true，失败返回false。

bool send(const string &buffer)

{

if (m\_clientfd==-1) return false;

if ( (::send(m\_clientfd,buffer.data(),buffer.size(),0))<=0) return false;

return true;

}

// 接收对端的报文，成功返回true，失败返回false。

// buffer-存放接收到的报文的内容，maxlen-本次接收报文的最大长度。

bool recv(string &buffer,const size\_t maxlen)

{

buffer.clear(); // 清空容器。

buffer.resize(maxlen); // 设置容器的大小为maxlen。

int readn=::recv(m\_clientfd,&buffer[0],buffer.size(),0); // 直接操作buffer的内存。

if (readn<=0) { buffer.clear(); return false; }

buffer.resize(readn); // 重置buffer的实际大小。

return true;

}

// 关闭监听的socket。

bool closelisten()

{

if (m\_listenfd==-1) return false;

::close(m\_listenfd);

m\_listenfd=-1;

return true;

}

// 关闭客户端连上来的socket。

bool closeclient()

{

if (m\_clientfd==-1) return false;

::close(m\_clientfd);

m\_clientfd=-1;

return true;

}

~ctcpserver() { closelisten(); closeclient(); }

};

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

{

if (argc!=2)

{

cout << "Using:./demo8 通讯端口\nExample:./demo8 5005\n\n"; // 端口大于1024，不与其它的重复。

cout << "注意：运行服务端程序的Linux系统的防火墙必须要开通5005端口。\n";

cout << " 如果是云服务器，还要开通云平台的访问策略。\n\n";

return -1;

}

ctcpserver tcpserver;

if (tcpserver.initserver(atoi(argv[1]))==false) // 初始化服务端用于监听的socket。

{

perror("initserver()"); return -1;

}

// 受理客户端的连接（从已连接的客户端中取出一个客户端），

// 如果没有已连接的客户端，accept()函数将阻塞等待。

if (tcpserver.accept()==false)

{

perror("accept()"); return -1;

}

cout << "客户端已连接(" << tcpserver.clientip() << ")。\n";

string buffer;

while (true)

{

// 接收对端的报文，如果对端没有发送报文，recv()函数将阻塞等待。

if (tcpserver.recv(buffer,1024)==false)

{

perror("recv()"); break;

}

cout << "接收：" << buffer << endl;

buffer="ok";

if (tcpserver.send(buffer)==false) // 向对端发送报文。

{

perror("send"); break;

}

cout << "发送：" << buffer << endl;

}

}

# 349、多进程的网络服务端

## 一、demo10.cpp

/\*

\* 程序名：demo10.cpp，此程序用于演示多进程的socket服务端

\*/

#include <iostream>

#include <cstdio>

#include <cstring>

#include <cstdlib>

#include <unistd.h>

#include <netdb.h>

#include <signal.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <arpa/inet.h>

using namespace std;

class ctcpserver // TCP通讯的服务端类。

{

private:

int m\_listenfd; // 监听的socket，-1表示未初始化。

int m\_clientfd; // 客户端连上来的socket，-1表示客户端未连接。

string m\_clientip; // 客户端字符串格式的IP。

unsigned short m\_port; // 服务端用于通讯的端口。

public:

ctcpserver():m\_listenfd(-1),m\_clientfd(-1) {}

// 初始化服务端用于监听的socket。

bool initserver(const unsigned short in\_port)

{

// 第1步：创建服务端的socket。

if ( (m\_listenfd=socket(AF\_INET,SOCK\_STREAM,0))==-1) return false;

m\_port=in\_port;

// 第2步：把服务端用于通信的IP和端口绑定到socket上。

struct sockaddr\_in servaddr; // 用于存放协议、端口和IP地址的结构体。

memset(&servaddr,0,sizeof(servaddr));

servaddr.sin\_family=AF\_INET; // ①协议族，固定填AF\_INET。

servaddr.sin\_port=htons(m\_port); // ②指定服务端的通信端口。

servaddr.sin\_addr.s\_addr=htonl(INADDR\_ANY); // ③如果操作系统有多个IP，全部的IP都可以用于通讯。

// 绑定服务端的IP和端口（为socket分配IP和端口）。

if (bind(m\_listenfd,(struct sockaddr \*)&servaddr,sizeof(servaddr))==-1)

{

close(m\_listenfd); m\_listenfd=-1; return false;

}

// 第3步：把socket设置为可连接（监听）的状态。

if (listen(m\_listenfd,5) == -1 )

{

close(m\_listenfd); m\_listenfd=-1; return false;

}

return true;

}

// 受理客户端的连接（从已连接的客户端中取出一个客户端），

// 如果没有已连接的客户端，accept()函数将阻塞等待。

bool accept()

{

struct sockaddr\_in caddr; // 客户端的地址信息。

socklen\_t addrlen=sizeof(caddr); // struct sockaddr\_in的大小。

if ((m\_clientfd=::accept(m\_listenfd,(struct sockaddr \*)&caddr,&addrlen))==-1) return false;

m\_clientip=inet\_ntoa(caddr.sin\_addr); // 把客户端的地址从大端序转换成字符串。

return true;

}

// 获取客户端的IP(字符串格式)。

const string & clientip() const

{

return m\_clientip;

}

// 向对端发送报文，成功返回true，失败返回false。

bool send(const string &buffer)

{

if (m\_clientfd==-1) return false;

if ( (::send(m\_clientfd,buffer.data(),buffer.size(),0))<=0) return false;

return true;

}

// 接收对端的报文，成功返回true，失败返回false。

// buffer-存放接收到的报文的内容，maxlen-本次接收报文的最大长度。

bool recv(string &buffer,const size\_t maxlen)

{

buffer.clear(); // 清空容器。

buffer.resize(maxlen); // 设置容器的大小为maxlen。

int readn=::recv(m\_clientfd,&buffer[0],buffer.size(),0); // 直接操作buffer的内存。

if (readn<=0) { buffer.clear(); return false; }

buffer.resize(readn); // 重置buffer的实际大小。

return true;

}

// 关闭监听的socket。

bool closelisten()

{

if (m\_listenfd==-1) return false;

::close(m\_listenfd);

m\_listenfd=-1;

return true;

}

// 关闭客户端连上来的socket。

bool closeclient()

{

if (m\_clientfd==-1) return false;

::close(m\_clientfd);

m\_clientfd=-1;

return true;

}

~ctcpserver() { closelisten(); closeclient(); }

};

ctcpserver tcpserver;

void FathEXIT(int sig); // 父进程的信号处理函数。

void ChldEXIT(int sig); // 子进程的信号处理函数。

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

{

if (argc!=2)

{

cout << "Using:./demo10 通讯端口\nExample:./demo10 5005\n\n";

cout << "注意：运行服务端程序的Linux系统的防火墙必须要开通5005端口。\n";

cout << " 如果是云服务器，还要开通云平台的访问策略。\n\n";

return -1;

}

// 忽略全部的信号，不希望被打扰。顺便解决了僵尸进程的问题。

for (int ii=1;ii<=64;ii++) signal(ii,SIG\_IGN);

// 设置信号,在shell状态下可用 "kill 进程号" 或 "Ctrl+c" 正常终止些进程

// 但请不要用 "kill -9 +进程号" 强行终止

signal(SIGTERM,FathEXIT); signal(SIGINT,FathEXIT); // SIGTERM 15 SIGINT 2

if (tcpserver.initserver(atoi(argv[1]))==false) // 初始化服务端用于监听的socket。

{

perror("initserver()"); return -1;

}

while (true)

{

// 受理客户端的连接（从已连接的客户端中取出一个客户端），

// 如果没有已连接的客户端，accept()函数将阻塞等待。

if (tcpserver.accept()==false)

{

perror("accept()"); return -1;

}

int pid=fork();

if (pid==-1) { perror("fork()"); return -1; } // 系统资源不足。

if (pid> 0)

{ // 父进程。

tcpserver.closeclient(); // 父进程关闭客户端连接的socket。

continue; // 父进程返回到循环开始的位置，继续受理客户端的连接。

}

tcpserver.closelisten(); // 子进程关闭监听的socket。

// 子进程需要重新设置信号。

signal(SIGTERM,ChldEXIT); // 子进程的退出函数与父进程不一样。

signal(SIGINT ,SIG\_IGN); // 子进程不需要捕获SIGINT信号。

// 子进程负责与客户端进行通讯。

cout << "客户端已连接(" << tcpserver.clientip() << ")。\n";

string buffer;

while (true)

{

// 接收对端的报文，如果对端没有发送报文，recv()函数将阻塞等待。

if (tcpserver.recv(buffer,1024)==false)

{

perror("recv()"); break;

}

cout << "接收：" << buffer << endl;

buffer="ok";

if (tcpserver.send(buffer)==false) // 向对端发送报文。

{

perror("send"); break;

}

cout << "发送：" << buffer << endl;

}

return 0; // 子进程一定要退出，否则又会回到accept()函数的位置。

}

}

// 父进程的信号处理函数。

void FathEXIT(int sig)

{

// 以下代码是为了防止信号处理函数在执行的过程中再次被信号中断。

signal(SIGINT,SIG\_IGN); signal(SIGTERM,SIG\_IGN);

cout << "父进程退出，sig=" << sig << endl;

kill(0,SIGTERM); // 向全部的子进程发送15的信号，通知它们退出。

// 在这里增加释放资源的代码（全局的资源）。

tcpserver.closelisten(); // 父进程关闭监听的socket。

exit(0);

}

// 子进程的信号处理函数。

void ChldEXIT(int sig)

{

// 以下代码是为了防止信号处理函数在执行的过程中再次被信号中断。

signal(SIGINT,SIG\_IGN); signal(SIGTERM,SIG\_IGN);

cout << "子进程" << getpid() << "退出，sig=" << sig << endl;

// 在这里增加释放资源的代码（只释放子进程的资源）。

tcpserver.closeclient(); // 子进程关闭客户端连上来的socket。

exit(0);

}

# 350、实现文件传输功能

## 一、demo11.cpp

/\*

\* 程序名：demo11.cpp，此程序用于演示文件传输的客户端。

\*/

#include <iostream>

#include <fstream>

#include <cstdio>

#include <cstring>

#include <cstdlib>

#include <unistd.h>

#include <netdb.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <arpa/inet.h>

using namespace std;

class ctcpclient // TCP通讯的客户端类。

{

private:

int m\_clientfd; // 客户端的socket，-1表示未连接或连接已断开；>=0表示有效的socket。

string m\_ip; // 服务端的IP/域名。

unsigned short m\_port; // 通讯端口。

public:

ctcpclient():m\_clientfd(-1) {}

// 向服务端发起连接请求，成功返回true，失败返回false。

bool connect(const string &in\_ip,const unsigned short in\_port)

{

if (m\_clientfd!=-1) return false; // 如果socket已连接，直接返回失败。

m\_ip=in\_ip; m\_port=in\_port; // 把服务端的IP和端口保存到成员变量中。

// 第1步：创建客户端的socket。

if ( (m\_clientfd = socket(AF\_INET,SOCK\_STREAM,0))==-1) return false;

// 第2步：向服务器发起连接请求。

struct sockaddr\_in servaddr; // 用于存放协议、端口和IP地址的结构体。

memset(&servaddr,0,sizeof(servaddr));

servaddr.sin\_family = AF\_INET; // ①协议族，固定填AF\_INET。

servaddr.sin\_port = htons(m\_port); // ②指定服务端的通信端口。

struct hostent\* h; // 用于存放服务端IP地址(大端序)的结构体的指针。

if ((h=gethostbyname(m\_ip.c\_str()))==nullptr ) // 把域名/主机名/字符串格式的IP转换成结构体。

{

::close(m\_clientfd); m\_clientfd=-1; return false;

}

memcpy(&servaddr.sin\_addr,h->h\_addr,h->h\_length); // ③指定服务端的IP(大端序)。

// 向服务端发起连接清求。

if (::connect(m\_clientfd,(struct sockaddr \*)&servaddr,sizeof(servaddr))==-1)

{

::close(m\_clientfd); m\_clientfd=-1; return false;

}

return true;

}

// 向服务端发送报文（字符串），成功返回true，失败返回false。

bool send(const string &buffer) // buffer不要用const char \*

{

if (m\_clientfd==-1) return false; // 如果socket的状态是未连接，直接返回失败。

if ((::send(m\_clientfd,buffer.data(),buffer.size(),0))<=0) return false;

return true;

}

// 向服务端发送报文（二进制数据），成功返回true，失败返回false。

bool send(void \*buffer,const size\_t size)

{

if (m\_clientfd==-1) return false; // 如果socket的状态是未连接，直接返回失败。

if ((::send(m\_clientfd,buffer,size,0))<=0) return false;

return true;

}

// 接收服务端的报文，成功返回true，失败返回false。

// buffer-存放接收到的报文的内容，maxlen-本次接收报文的最大长度。

bool recv(string &buffer,const size\_t maxlen)

{ // 如果直接操作string对象的内存，必须保证：1)不能越界；2）操作后手动设置数据的大小。

buffer.clear(); // 清空容器。

buffer.resize(maxlen); // 设置容器的大小为maxlen。

int readn=::recv(m\_clientfd,&buffer[0],buffer.size(),0); // 直接操作buffer的内存。

if (readn<=0) { buffer.clear(); return false; }

buffer.resize(readn); // 重置buffer的实际大小。

return true;

}

// 断开与服务端的连接。

bool close()

{

if (m\_clientfd==-1) return false; // 如果socket的状态是未连接，直接返回失败。

::close(m\_clientfd);

m\_clientfd=-1;

return true;

}

// 向服务端发送文件内容。

bool sendfile(const string &filename,const size\_t filesize)

{

// 以二进制的方式打开文件。

ifstream fin(filename,ios::binary);

if (fin.is\_open() == false) { cout << "打开文件" << filename << "失败。\n"; return false; }

int onread=0; // 每次调用fin.read()时打算读取的字节数。 每次应搬砖头数。

int totalbytes=0; // 从文件中已读取的字节总数。 已搬砖头数。

char buffer[4096]; // 存放读取数据的buffer。 每次搬七块砖头。

while (true)

{

memset(buffer,0,sizeof(buffer));

// 计算本次应该读取的字节数，如果剩余的数据超过4096字节，就读4096字节。

if (filesize-totalbytes>4096) onread=4096;

else onread=filesize-totalbytes;

// 从文件中读取数据。

fin.read(buffer,onread);

// 把读取到的数据发送给对端。

if (send(buffer,onread)==false) return false;

// 计算文件已读取的字节总数，如果文件已读完，跳出循环。

totalbytes=totalbytes+onread;

if (totalbytes==filesize) break;

}

return true;

}

~ctcpclient(){ close(); }

};

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

{

if (argc!=5)

{

cout << "Using:./demo11 服务端的IP 服务端的端口 文件名 文件大小\n";

cout << "Example:./demo11 192.168.101.138 5005 aaa.txt 2424\n\n";

return -1;

}

ctcpclient tcpclient;

if (tcpclient.connect(argv[1],atoi(argv[2]))==false) // 向服务端发起连接请求。

{

perror("connect()"); return -1;

}

// 以下是发送文件的流程。

// 1）把待传输文件名和文件的大小告诉服务端。

// 定义文件信息的结构体。

struct st\_fileinfo{

char filename[256]; // 文件名。

int filesize; // 文件大小。

}fileinfo;

memset(&fileinfo,0,sizeof(fileinfo));

strcpy(fileinfo.filename,argv[3]); // 文件名。

fileinfo.filesize=atoi(argv[4]); // 文件大小。

// 把文件信息的结构体发送给服务端。

if (tcpclient.send(&fileinfo,sizeof(fileinfo))==false) { perror("send"); return -1; }

cout << "发送文件信息的结构体" << fileinfo.filename << "(" << fileinfo.filesize <<")。"<< endl;

// 2）等待服务端的确认报文（文件名和文件的大小的确认）。

string buffer;

if (tcpclient.recv(buffer,2)==false) { perror("recv()"); return -1; }

if (buffer!="ok") { cout << "服务端没有回复ok。\n"; return -1; }

// 3）发送文件内容。

if (tcpclient.sendfile(fileinfo.filename,fileinfo.filesize)==false)

{

perror("sendfile()"); return -1;

}

// 4）等待服务端的确认报文（服务端已接收完文件）。

if (tcpclient.recv(buffer,2)==false) { perror("recv()"); return -1; }

if (buffer!="ok") { cout << "发送文件内容失败。\n"; return -1; }

cout << "发送文件内容成功。\n";

}

## 二、demo12.cpp

/\*

\* 程序名：demo12.cpp，此程序用于演示文件传输的服务端。

\*/

#include <iostream>

#include <fstream>

#include <cstdio>

#include <cstring>

#include <cstdlib>

#include <unistd.h>

#include <netdb.h>

#include <signal.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <arpa/inet.h>

using namespace std;

class ctcpserver // TCP通讯的服务端类。

{

private:

int m\_listenfd; // 监听的socket，-1表示未初始化。

int m\_clientfd; // 客户端连上来的socket，-1表示客户端未连接。

string m\_clientip; // 客户端字符串格式的IP。

unsigned short m\_port; // 服务端用于通讯的端口。

public:

ctcpserver():m\_listenfd(-1),m\_clientfd(-1) {}

// 初始化服务端用于监听的socket。

bool initserver(const unsigned short in\_port)

{

// 第1步：创建服务端的socket。

if ( (m\_listenfd=socket(AF\_INET,SOCK\_STREAM,0))==-1) return false;

m\_port=in\_port;

// 第2步：把服务端用于通信的IP和端口绑定到socket上。

struct sockaddr\_in servaddr; // 用于存放协议、端口和IP地址的结构体。

memset(&servaddr,0,sizeof(servaddr));

servaddr.sin\_family=AF\_INET; // ①协议族，固定填AF\_INET。

servaddr.sin\_port=htons(m\_port); // ②指定服务端的通信端口。

servaddr.sin\_addr.s\_addr=htonl(INADDR\_ANY); // ③如果操作系统有多个IP，全部的IP都可以用于通讯。

// 绑定服务端的IP和端口（为socket分配IP和端口）。

if (bind(m\_listenfd,(struct sockaddr \*)&servaddr,sizeof(servaddr))==-1)

{

close(m\_listenfd); m\_listenfd=-1; return false;

}

// 第3步：把socket设置为可连接（监听）的状态。

if (listen(m\_listenfd,5) == -1 )

{

close(m\_listenfd); m\_listenfd=-1; return false;

}

return true;

}

// 受理客户端的连接（从已连接的客户端中取出一个客户端），

// 如果没有已连接的客户端，accept()函数将阻塞等待。

bool accept()

{

struct sockaddr\_in caddr; // 客户端的地址信息。

socklen\_t addrlen=sizeof(caddr); // struct sockaddr\_in的大小。

if ((m\_clientfd=::accept(m\_listenfd,(struct sockaddr \*)&caddr,&addrlen))==-1) return false;

m\_clientip=inet\_ntoa(caddr.sin\_addr); // 把客户端的地址从大端序转换成字符串。

return true;

}

// 获取客户端的IP(字符串格式)。

const string & clientip() const

{

return m\_clientip;

}

// 向对端发送报文，成功返回true，失败返回false。

bool send(const string &buffer)

{

if (m\_clientfd==-1) return false;

if ( (::send(m\_clientfd,buffer.data(),buffer.size(),0))<=0) return false;

return true;

}

// 接收对端的报文（字符串），成功返回true，失败返回false。

// buffer-存放接收到的报文的内容，maxlen-本次接收报文的最大长度。

bool recv(string &buffer,const size\_t maxlen)

{

buffer.clear(); // 清空容器。

buffer.resize(maxlen); // 设置容器的大小为maxlen。

int readn=::recv(m\_clientfd,&buffer[0],buffer.size(),0); // 直接操作buffer的内存。

if (readn<=0) { buffer.clear(); return false; }

buffer.resize(readn); // 重置buffer的实际大小。

return true;

}

// 接收客户端的报文（二进制数据），成功返回true，失败返回false。

// buffer-存放接收到的报文的内容，size-本次接收报文的最大长度。

bool recv(void \*buffer,const size\_t size)

{

if (::recv(m\_clientfd,buffer,size,0)<=0) return false;

return true;

}

// 关闭监听的socket。

bool closelisten()

{

if (m\_listenfd==-1) return false;

::close(m\_listenfd);

m\_listenfd=-1;

return true;

}

// 关闭客户端连上来的socket。

bool closeclient()

{

if (m\_clientfd==-1) return false;

::close(m\_clientfd);

m\_clientfd=-1;

return true;

}

// 接收文件内容。

bool recvfile(const string &filename,const size\_t filesize)

{

ofstream fout;

fout.open(filename,ios::binary);

if (fout.is\_open() == false) { cout << "打开文件" << filename << "失败。\n"; return false; }

int totalbytes=0; // 已接收文件的总字节数。

int onread=0; // 本次打算接收的字节数。

char buffer[4096]; // 接收文件内容的缓冲区。

while (true)

{

// 计算本次应该接收的字节数。

if (filesize-totalbytes>4096) onread=4096;

else onread=filesize-totalbytes;

// 接收文件内容。

if (recv(buffer,onread)==false) return false;

// 把接收到的内容写入文件。

fout.write(buffer,onread);

// 计算已接收文件的总字节数，如果文件接收完，跳出循环。

totalbytes=totalbytes+onread;

if (totalbytes==filesize) break;

}

return true;

}

~ctcpserver() { closelisten(); closeclient(); }

};

ctcpserver tcpserver;

void FathEXIT(int sig); // 父进程的信号处理函数。

void ChldEXIT(int sig); // 子进程的信号处理函数。

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

{

if (argc!=3)

{

cout << "Using:./demo12 通讯端口 文件存放的目录\n";

cout << "Example:./demo12 5005 /tmp\n\n";

cout << "注意：运行服务端程序的Linux系统的防火墙必须要开通5005端口。\n";

cout << " 如果是云服务器，还要开通云平台的访问策略。\n\n";

return -1;

}

// 忽略全部的信号，不希望被打扰。顺便解决了僵尸进程的问题。

for (int ii=1;ii<=64;ii++) signal(ii,SIG\_IGN);

// 设置信号,在shell状态下可用 "kill 进程号" 或 "Ctrl+c" 正常终止些进程

// 但请不要用 "kill -9 +进程号" 强行终止

signal(SIGTERM,FathEXIT); signal(SIGINT,FathEXIT); // SIGTERM 15 SIGINT 2

if (tcpserver.initserver(atoi(argv[1]))==false) // 初始化服务端用于监听的socket。

{

perror("initserver()"); return -1;

}

while (true)

{

// 受理客户端的连接（从已连接的客户端中取出一个客户端），

// 如果没有已连接的客户端，accept()函数将阻塞等待。

if (tcpserver.accept()==false)

{

perror("accept()"); return -1;

}

int pid=fork();

if (pid==-1) { perror("fork()"); return -1; } // 系统资源不足。

if (pid> 0)

{ // 父进程。

tcpserver.closeclient(); // 父进程关闭客户端连接的socket。

continue; // 父进程返回到循环开始的位置，继续受理客户端的连接。

}

tcpserver.closelisten(); // 子进程关闭监听的socket。

// 子进程需要重新设置信号。

signal(SIGTERM,ChldEXIT); // 子进程的退出函数与父进程不一样。

signal(SIGINT ,SIG\_IGN); // 子进程不需要捕获SIGINT信号。

// 子进程负责与客户端进行通讯。

cout << "客户端已连接(" << tcpserver.clientip() << ")。\n";

// 以下是接收文件的流程。

// 1）接收文件名和文件大小信息。

// 定义文件信息的结构体。

struct st\_fileinfo{

char filename[256]; // 文件名。

int filesize; // 文件大小。

}fileinfo;

memset(&fileinfo,0,sizeof(fileinfo));

// 用结构体存放接收报文的内容。

if (tcpserver.recv(&fileinfo,sizeof(fileinfo))==false) { perror("recv()"); return -1; }

cout << "文件信息结构体" << fileinfo.filename << "(" << fileinfo.filesize <<")。"<< endl;

// 2）给客户端回复确认报文，表示客户端可以发送文件了。

if (tcpserver.send("ok")==false) { perror("send"); break; }

// 3）接收文件内容。 string char \* + const char \* + char \*

if (tcpserver.recvfile(string(argv[2])+"/"+fileinfo.filename,fileinfo.filesize)==false)

{

cout << "接收文件内容失败。\n"; return -1;

}

cout << "接收文件内容成功。\n";

// 4）给客户端回复确认报文，表示文件已接收成功。

tcpserver.send("ok");

return 0; // 子进程一定要退出，否则又会回到accept()函数的位置。

}

}

// 父进程的信号处理函数。

void FathEXIT(int sig)

{

// 以下代码是为了防止信号处理函数在执行的过程中再次被信号中断。

signal(SIGINT,SIG\_IGN); signal(SIGTERM,SIG\_IGN);

cout << "父进程退出，sig=" << sig << endl;

kill(0,SIGTERM); // 向全部的子进程发送15的信号，通知它们退出。

// 在这里增加释放资源的代码（全局的资源）。

tcpserver.closelisten(); // 父进程关闭监听的socket。

exit(0);

}

// 子进程的信号处理函数。

void ChldEXIT(int sig)

{

// 以下代码是为了防止信号处理函数在执行的过程中再次被信号中断。

signal(SIGINT,SIG\_IGN); signal(SIGTERM,SIG\_IGN);

cout << "子进程" << getpid() << "退出，sig=" << sig << endl;

// 在这里增加释放资源的代码（只释放子进程的资源）。

tcpserver.closeclient(); // 子进程关闭客户端连上来的socket。

exit(0);

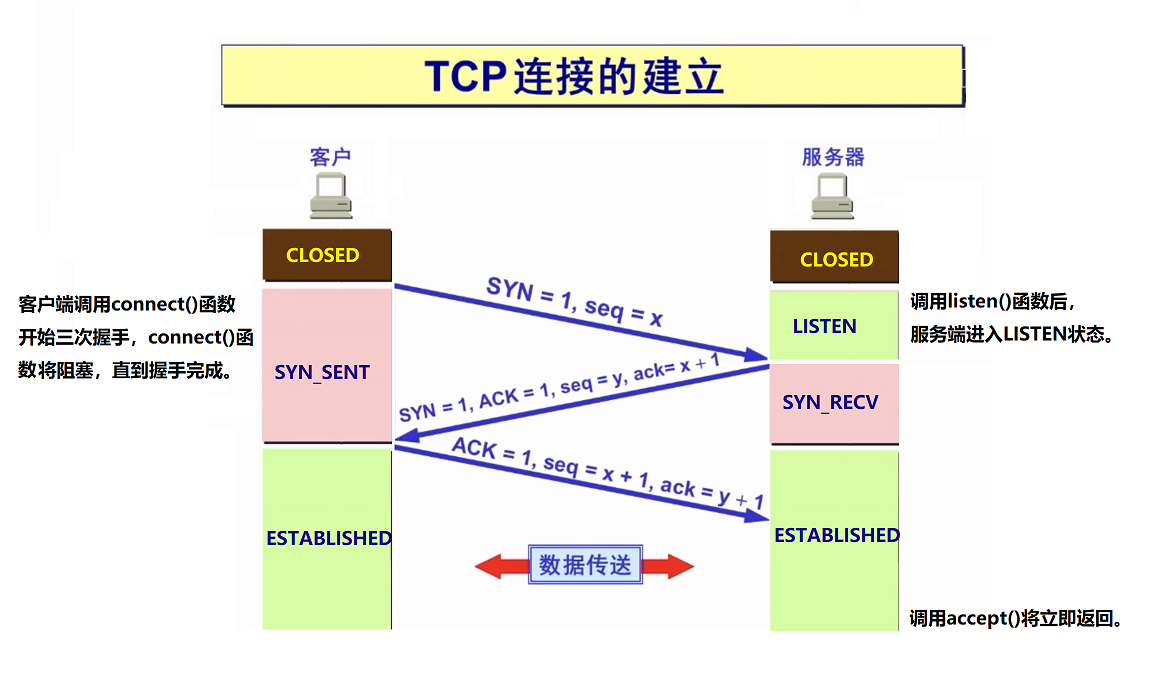
}

# 351、三次握手与四次挥手

TCP是面向连接的、可靠的协议，建立TCP连接需要三次对话（三次握手），拆除TCP连接需要四次对话（四次握/挥手）。

## 一、三次握手

服务端调用listen()函数后进入监听（等待连接）状态，这时候，客户端就可以调用connect()函数发起TCP连接请求，connect()函数会触发三次握手，三次握手完成后，客户端和服务端将建立一个双向的传输通道。



情景类似：

1、客户端对服务端说：我可以给你发送数据吗？

2、服务端回复：ok，不过，我也要给你发送数据。（这时候，客户端至服务端的单向传输通道已建立）

3、客户端回复：ok。（这时候，服务端至客户端的单向传输通道已建立）

**细节：**

1）客户端的socket也有端口号，对程序员来说，不必关心客户端socket的端口号，所以系统随机分配。（socket通讯中的地址包括ip和端口号，但是，习惯中的地址仅指ip地址）

2）服务端的bind()函数，普通用户只能使用1024以上的端口，root用户可以使用任意端口。

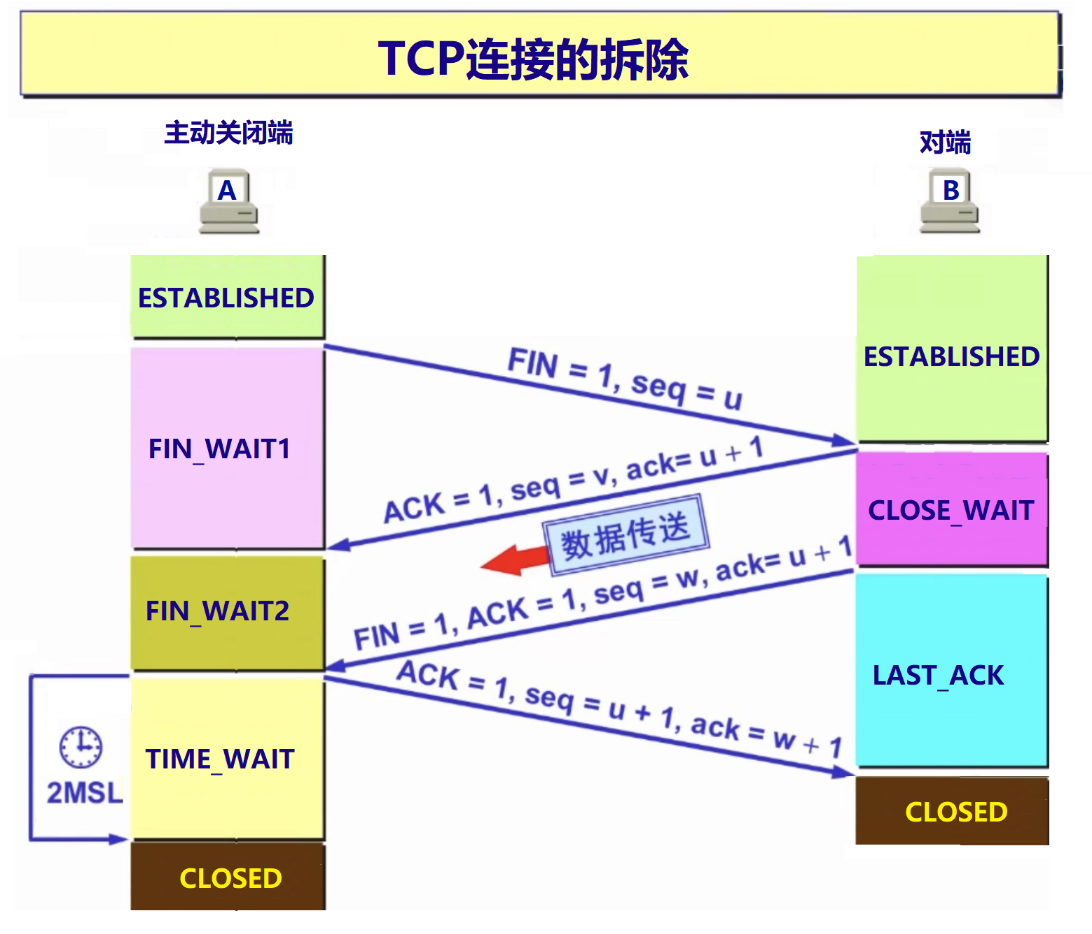
3）listen()函数的第二个参数+1为已连接队列（ESTABLISHED状态，三次握手已完成但是没有被accept()的socket，只存在于服务端）的大小。（在高并发的服务程序中，该参数应该调大一些）

4）SYN\_RECV状态的连接也称为半连接。

5）CLOSED是假想状态，实际上不存在。

## 二、四次挥手（握手）

断开一个TCP连接时，客户端和服务端需要相互总共发送四个包以确认连接的断开。在socket编程中，这一过程由客户端或服务端任一方执行close()函数触发。



情景类似：

1）一端（A）对另一端（B）说：我不会给你发数据了，断开连接吧。

2）B回复：ok。（这时候A不能对B发数据了，但是，B仍可以对A发数据）

3）B发完数据了，对A说：我也不会给你发数据了。（这时候B也不能对A发数据了）。

4、A回复：ok。

**细节：**

1）主动断开的端在四次挥手后，socket的状态为TIME\_WAIT，该状态将持续2MSL（30秒/1分钟/**2分钟**）。 MSL（Maximum Segment Lifetime）报文在网络上存在的最长时间，超过这个时间报文将被丢弃。

2）如果是客户端主动断开，TIME\_WAIT状态的socket几乎不会造成危害。a）客户端程序的socket很少，服务端程序的socket很多（成千上万）；b）客户端的端口是随机分配的，不存在重用的问题。

3）如果是服务端主动断开，有两方面的危害：a）socket没有立即释放；b）端口号只能在2MSL后才能重用。

在服务端程序中，用setsockopt()函数设置socket的属性（一定要放在bind()之前）。

int opt = 1;

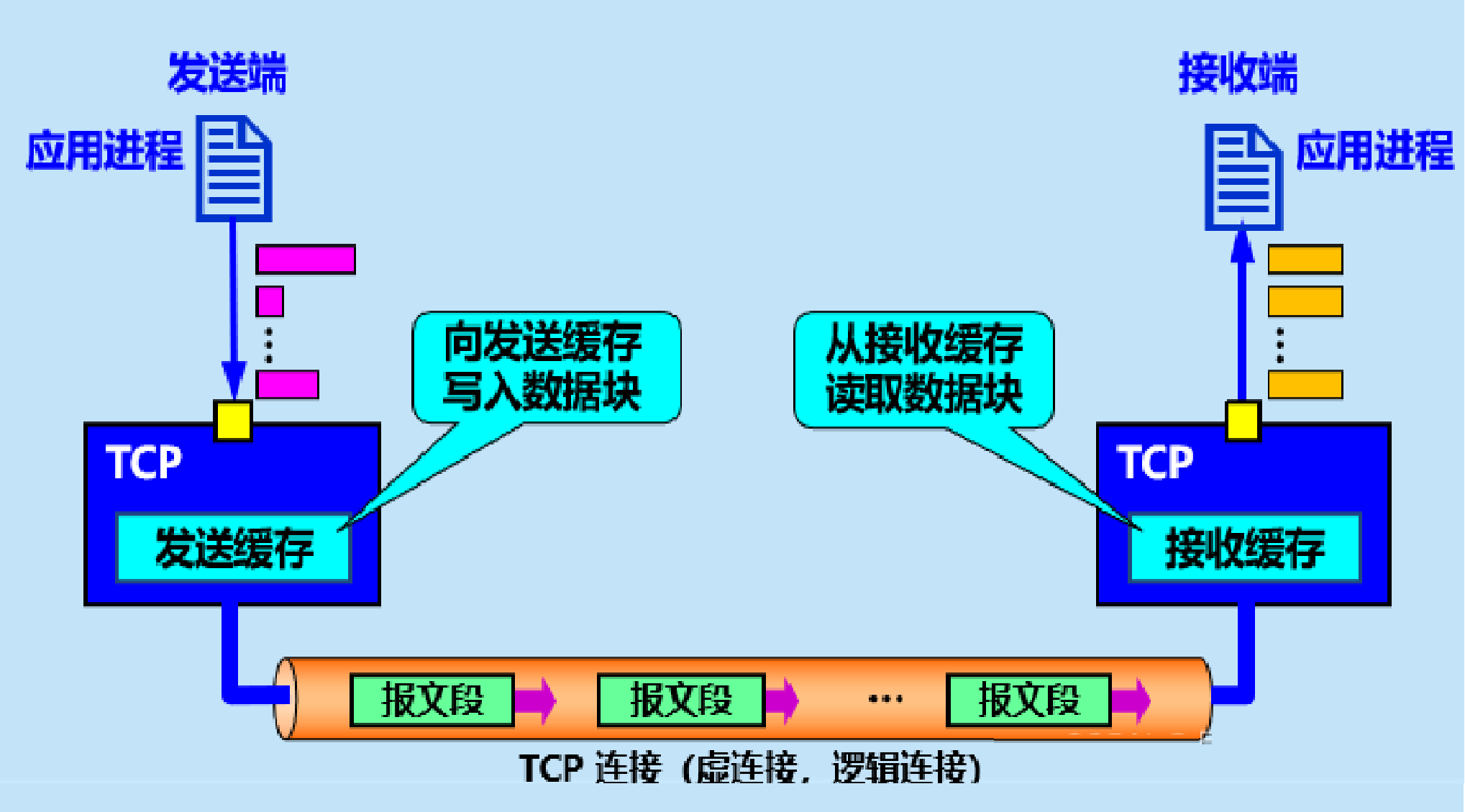
setsockopt(m\_listenfd,SOL\_SOCKET,SO\_REUSEADDR,&opt,sizeof(opt));

# 352、TCP缓存

系统为每个socket创建了发送缓冲区和接收缓冲区，应用程序调用send()/write()函数发送数据的时候，内核把数据从应用进程拷贝socket的发送缓冲区中；应用程序调用recv()/read()函数接收数据的时候，内核把数据从socket的接收缓冲区拷贝应用进程中。

发送数据即把数据放入发送缓冲区中。

接收数据即从接收缓冲区中取数据。



查看socket缓存的大小：

int bufsize = 0;

socklen\_t optlen = sizeof(bufsize);

getsockopt(sockfd, SOL\_SOCKET, SO\_SNDBUF, &bufsize, &optlen); // 获取发送缓冲区的大小。

cout << "send bufsize=" << bufsize << endl;

getsockopt(sockfd, SOL\_SOCKET, SO\_RCVBUF, &bufsize, &optlen); // 获取接收缓冲区的大小。

cout << "recv bufsize=" << bufsize << endl;

问题：

1）send()函数有可能会阻塞吗？ 如果自己的发送缓冲区和对端的接收缓冲区都满了，会阻塞。

2）向socket中写入数据后，如果关闭了socket，对端还能接收到数据吗？

**Nagle算法**

在TCP协议中，无论发送多少数据，都要在数据前面加上协议头，同时，对方收到数据后，也需要回复ACK表示确认。为了尽可能的利用网络带宽，TCP希望每次都能够以MSS（Maximum Segment Size，最大报文长度）的数据块来发送数据。

Nagle算法就是为了尽可能发送大块的数据，避免网络中充斥着小数据块。

Nagle算法的定义是：任意时刻，最多只能有一个未被确认的小段，小段是指小于MSS的数据块，未被确认是指一个数据块发送出去后，没有收到对端回复的ACK。

举个例子：发送端调用send()函数将一个int型数据（称之为A数据块）写入到socket中，A数据块会被马上发送到接收端，接着，发送端又调用send()函数写入一个int型数据（称之为B数据块），这时候，A块的ACK没有返回（已经存在了一个未被确认的小段），所以B块不会立即被发送，而是等A块的ACK返回之后（大概40ms）才发送。

TCP协议中不仅仅有Nagle算法，还有一个ACK延迟机制：当接收端收到数据之后，并不会马上向发送端回复ACK，而是延迟40ms后再回复，它希望在40ms内接收端会向发送端回复应答数据，这样ACK就可以和应答数据一起发送，把ACK捎带过去。

如果TCP连接的一端启用了Nagle算法，另一端启用了ACK延时机制，而发送的数据包又比较小，则可能会出现这样的情况：发送端在等待上一个包的ACK，而接收端正好延迟了此ACK，那么这个正要被发送的包就会延迟40ms。

**解决方案**

开启TCP\_NODELAY选项，这个选项的作用就是禁用Nagle算法。

#include <netinet/tcp.h> // 注意，要包含这个头文件。

int opt = 1;

setsockopt(sockfd, IPPROTO\_TCP, TCP\_NODELAY,&opt,sizeof(opt));

对时效要求很高的系统，例如联机游戏、证券交易，一般会禁用Nagle算法。

# 351、IO多路复用

多进程、多线程的网络服务器，为每个客户端连接创建一个进程/线程，不够用。

IO多路复用技术，可以用一个进程/线程处理多个TCP链接，减少系统开销

Select(1024)

Poll(数千)

Epoll（百万）

## 一、tcpselect.cpp

/\*

/\*

 \* 程序名：tcpselect.cpp，此程序用于演示采用select模型实现网络通讯的服务端。

 \* 作者：吴从周

\*/

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <string.h>

#include <sys/socket.h>

#include <arpa/inet.h>

#include <sys/fcntl.h>

// 初始化服务端的监听端口。

int initserver(int port);

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

{

    if (argc != 2) { printf("usage: ./tcpselect port\n"); return -1; }

    // 初始化服务端用于监听的socket。

    int listensock = initserver(atoi(argv[1]));

    printf("listensock=%d\n",listensock);

    if (listensock < 0) { printf("initserver() failed.\n"); return -1; }

    // 读事件：1）已连接队列中有已经准备好的socket（有新的客户端连上来了）；

    //               2）接收缓存中有数据可以读（对端发送的报文已到达）；

    //               3）tcp连接已断开（对端调用close()函数关闭了连接）。

    // 写事件：发送缓冲区没有满，可以写入数据（可以向对端发送报文）。

    fd\_set readfds;                         // 需要监视读事件的socket的集合，大小为16字节（1024位）的bitmap。

    FD\_ZERO(&readfds);                // 初始化readfds，把bitmap的每一位都置为0。

    FD\_SET(listensock,&readfds);  // 把服务端用于监听的socket加入readfds。

    int maxfd=listensock;              // readfds中socket的最大值。

    while (true)        // 事件循环。

    {

        // 用于表示超时时间的结构体。

        struct timeval timeout;

        timeout.tv\_sec=10;        // 秒

        timeout.tv\_usec=0;        // 微秒。

        fd\_set tmpfds=readfds;      // 在select()函数中，会修改bitmap，所以，要把readfds复制一份给tmpfds，再把tmpfds传给select()。

        // 调用select() 等待事件的发生（监视哪些socket发生了事件)。

        int infds=select(maxfd+1,&tmpfds,NULL,NULL,0);

        // 如果infds<0，表示调用select()失败。

        if (infds<0)

        {

            perror("select() failed"); break;

        }

        // 如果infds==0，表示select()超时。

        if (infds==0)

        {

            printf("select() timeout.\n"); continue;

        }

        // 如果infds>0，表示有事件发生，infds存放了已发生事件的个数。

        for (int eventfd=0;eventfd<=maxfd;eventfd++)

        {

            if (FD\_ISSET(eventfd,&tmpfds)==0) continue;   // 如果eventfd在bitmap中的标志为0，表示它没有事件，continue

            // 如果发生事件的是listensock，表示已连接队列中有已经准备好的socket（有新的客户端连上来了）。

            if (eventfd==listensock)

            {

                struct sockaddr\_in client;

                socklen\_t len = sizeof(client);

                int clientsock = accept(listensock,(struct sockaddr\*)&client,&len);

                if (clientsock < 0) { perror("accept() failed"); continue; }

                printf ("accept client(socket=%d) ok.\n",clientsock);

                FD\_SET(clientsock,&readfds);                      // 把bitmap中新连上来的客户端的标志位置为1。

                if (maxfd<clientsock) maxfd=clientsock;    // 更新maxfd的值。

            }

            else

            {

                // 如果是客户端连接的socke有事件，表示接收缓存中有数据可以读（对端发送的报文已到达），或者有客户端已断开连接。

                char buffer[1024];                      // 存放从接收缓冲区中读取的数据。

                memset(buffer,0,sizeof(buffer));

                if (recv(eventfd,buffer,sizeof(buffer),0)<=0)

                {

                    // 如果客户端的连接已断开。

                    printf("client(eventfd=%d) disconnected.\n",eventfd);

                    close(eventfd);                         // 关闭客户端的socket

                    FD\_CLR(eventfd,&readfds);     // 把bitmap中已关闭客户端的标志位清空。

                    if (eventfd == maxfd)              // 重新计算maxfd的值，注意，只有当eventfd==maxfd时才需要计算。

                    {

                        for (int ii=maxfd;ii>0;ii--)    // 从后面往前找。

                        {

                            if (FD\_ISSET(ii,&readfds))

                            {

                                maxfd = ii; break;

                            }

                        }

                    }

                }

                else

                {

                    // 如果客户端有报文发过来。

                    printf("recv(eventfd=%d):%s\n",eventfd,buffer);

                    // 把接收到的报文内容原封不动的发回去。

                    send(eventfd,buffer,strlen(buffer),0);

                }

            }

        }

    }

    return 0;

}

// 初始化服务端的监听端口。

int initserver(int port)

{

    int sock = socket(AF\_INET,SOCK\_STREAM,0);

    if (sock < 0)

    {

        perror("socket() failed"); return -1;

    }

    int opt = 1; unsigned int len = sizeof(opt);

    setsockopt(sock,SOL\_SOCKET,SO\_REUSEADDR,&opt,len);

    struct sockaddr\_in servaddr;

    servaddr.sin\_family = AF\_INET;

    servaddr.sin\_addr.s\_addr = htonl(INADDR\_ANY);

    servaddr.sin\_port = htons(port);

    if (bind(sock,(struct sockaddr \*)&servaddr,sizeof(servaddr)) < 0 )

    {

        perror("bind() failed"); close(sock); return -1;

    }

    if (listen(sock,5) != 0 )

    {

        perror("listen() failed"); close(sock); return -1;

    }

    return sock;

}

## 二、client.cpp

// 网络通讯的客户端程序。

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <errno.h>

#include <string.h>

#include <netinet/in.h>

#include <sys/socket.h>

#include <arpa/inet.h>

#include <time.h>

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

{

    if (argc != 3)

    {

        printf("usage:./client ip port\n"); return -1;

    }

    int sockfd;

    struct sockaddr\_in servaddr;

    char buf[1024];

    if ((sockfd=socket(AF\_INET,SOCK\_STREAM,0))<0) { printf("socket() failed.\n"); return -1; }

    memset(&servaddr,0,sizeof(servaddr));

    servaddr.sin\_family=AF\_INET;

    servaddr.sin\_port=htons(atoi(argv[2]));

    servaddr.sin\_addr.s\_addr=inet\_addr(argv[1]);

    if (connect(sockfd, (struct sockaddr \*)&servaddr,sizeof(servaddr)) != 0)

    {

        printf("connect(%s:%s) failed.\n",argv[1],argv[2]); close(sockfd);  return -1;

    }

    printf("connect ok.\n");

    // printf("开始时间：%d",time(0));

    for (int ii=0;ii<200000;ii++)

    {

        // 从命令行输入内容。

        memset(buf,0,sizeof(buf));

        printf("please input:"); scanf("%s",buf);

        if (send(sockfd,buf,strlen(buf),0) <=0)

        {

            printf("write() failed.\n");  close(sockfd);  return -1;

        }

        memset(buf,0,sizeof(buf));

        if (recv(sockfd,buf,sizeof(buf),0) <=0)

        {

            printf("read() failed.\n");  close(sockfd);  return -1;

        }

        printf("recv:%s\n",buf);

    }

    // printf("结束时间：%d",time(0));

}

## 三、tcppoll.cpp

/\*

 \* 程序名：tcppoll.cpp，此程序用于演示采用poll模型实现网络通讯的服务端。

 \* 作者：吴从周

\*/

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <string.h>

#include <poll.h>

#include <sys/socket.h>

#include <arpa/inet.h>

#include <sys/fcntl.h>

// 初始化服务端的监听端口。

int initserver(int port);

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

{

    if (argc != 2) { printf("usage: ./tcppoll port\n"); return -1; }

    // 初始化服务端用于监听的socket。

    int listensock = initserver(atoi(argv[1]));

    printf("listensock=%d\n",listensock);

    if (listensock < 0) { printf("initserver() failed.\n"); return -1; }

    pollfd fds[2048];                 // fds存放需要监视的socket。

    // 初始化数组，把全部的socket设置为-1，如果数组中的socket的值为-1，那么，poll将忽略它。

    for (int ii=0;ii<2048;ii++)

        fds[ii].fd=-1;

    // 打算让poll监视listensock读事件。

    fds[listensock].fd=listensock;

    fds[listensock].events=POLLIN;        // POLLIN表示读事件，POLLOUT表示写事件。

    // fds[listensock].events=POLLIN|POLLOUT;

    int maxfd=listensock;        // fds数组中需要监视的socket的实际大小。

    while (true)        // 事件循环。

    {

        // 调用poll() 等待事件的发生（监视哪些socket发生了事件)。

        int infds=poll(fds,maxfd+1,10000);      // 超时时间为10秒。

        // 如果infds<0，表示调用poll()失败。

        if (infds < 0)

        {

            perror("poll() failed"); break;

        }

        // 如果infds==0，表示poll()超时。

        if (infds == 0)

        {

            printf("poll() timeout.\n"); continue;

        }

        // 如果infds>0，表示有事件发生，infds存放了已发生事件的个数。

        for (int eventfd=0;eventfd<=maxfd;eventfd++)

        {

            if (fds[eventfd].fd<0) continue;                               // 如果fd为负，忽略它。

            if ((fds[eventfd].revents&POLLIN)==0)  continue;  // 如果没有读事件，continue

            // 如果发生事件的是listensock，表示已连接队列中有已经准备好的socket（有新的客户端连上来了）。

            if (eventfd==listensock)

            {

                struct sockaddr\_in client;

                socklen\_t len = sizeof(client);

                int clientsock = accept(listensock,(struct sockaddr\*)&client,&len);

                if (clientsock < 0) { perror("accept() failed"); continue; }

                printf ("accept client(socket=%d) ok.\n",clientsock);

                // 修改fds数组中clientsock位置的元素。

                fds[clientsock].fd=clientsock;

                fds[clientsock].events=POLLIN;

                if (maxfd<clientsock) maxfd=clientsock;    // 更新maxfd的值。

            }

            else

            {

                // 如果是客户端连接的socke有事件，表示有报文发过来了或者连接已断开。

                char buffer[1024]; // 存放从客户端读取的数据。

                memset(buffer,0,sizeof(buffer));

                if (recv(eventfd,buffer,sizeof(buffer),0)<=0)

                {

                    // 如果客户端的连接已断开。

                    printf("client(eventfd=%d) disconnected.\n",eventfd);

                    close(eventfd);               // 关闭客户端的socket。

                    fds[eventfd].fd=-1;        // 修改fds数组中clientsock位置的元素，置为-1，poll将忽略该元素。

                    // 重新计算maxfd的值，注意，只有当eventfd==maxfd时才需要计算。

                    if (eventfd == maxfd)

                    {

                        for (int ii=maxfd;ii>0;ii--)  // 从后面往前找。

                        {

                            if (fds[ii].fd!=-1)

                            {

                                maxfd = ii; break;

                            }

                        }

                    }

                }

                else

                {

                    // 如果客户端有报文发过来。

                    printf("recv(eventfd=%d):%s\n",eventfd,buffer);

                    send(eventfd,buffer,strlen(buffer),0);

                }

            }

        }

    }

    return 0;

}

// 初始化服务端的监听端口。

int initserver(int port)

{

    int sock = socket(AF\_INET,SOCK\_STREAM,0);

    if (sock < 0)

    {

        perror("socket() failed"); return -1;

    }

    int opt = 1; unsigned int len = sizeof(opt);

    setsockopt(sock,SOL\_SOCKET,SO\_REUSEADDR,&opt,len);

    struct sockaddr\_in servaddr;

    servaddr.sin\_family = AF\_INET;

    servaddr.sin\_addr.s\_addr = htonl(INADDR\_ANY);

    servaddr.sin\_port = htons(port);

    if (bind(sock,(struct sockaddr \*)&servaddr,sizeof(servaddr)) < 0 )

    {

        perror("bind() failed"); close(sock); return -1;

    }

    if (listen(sock,5) != 0 )

    {

        perror("listen() failed"); close(sock); return -1;

    }

    return sock;

}

## 四、tcpepoll.cpp

/\*

 \* 程序名：tcpepoll.cpp，此程序用于演示采用epoll模型实现网络通讯的服务端。

 \* 作者：吴从周

\*/

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <string.h>

#include <errno.h>

#include <sys/socket.h>

#include <sys/types.h>

#include <arpa/inet.h>

#include <sys/fcntl.h>

#include <sys/epoll.h>

// 初始化服务端的监听端口。

int initserver(int port);

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

{

    if (argc != 2) { printf("usage: ./tcpepoll port\n"); return -1; }

    // 初始化服务端用于监听的socket。

    int listensock = initserver(atoi(argv[1]));

    printf("listensock=%d\n",listensock);

    if (listensock < 0) { printf("initserver() failed.\n"); return -1; }

    // 创建epoll句柄。

    int epollfd=epoll\_create(1);

    // 为服务端的listensock准备读事件。

    epoll\_event ev;              // 声明事件的数据结构。

    ev.data.fd=listensock;   // 指定事件的自定义数据，会随着epoll\_wait()返回的事件一并返回。

    // ev.data.ptr=(void\*)"超女";   // 指定事件的自定义数据，会随着epoll\_wait()返回的事件一并返回。

    ev.events=EPOLLIN;      // 打算让epoll监视listensock的读事件。

    epoll\_ctl(epollfd,EPOLL\_CTL\_ADD,listensock,&ev);     // 把需要监视的socket和事件加入epollfd中。

    epoll\_event evs[10];      // 存放epoll返回的事件。

    while (true)        // 事件循环。

    {

        // 等待监视的socket有事件发生。

        int infds=epoll\_wait(epollfd,evs,10,-1);

        // 返回失败。

        if (infds < 0)

        {

            perror("epoll() failed"); break;

        }

        // 超时。

        if (infds == 0)

        {

            printf("epoll() timeout.\n"); continue;

        }

        // 如果infds>0，表示有事件发生的socket的数量。

        for (int ii=0;ii<infds;ii++)       // 遍历epoll返回的数组evs。

        {

            // printf("ptr=%s,events=%d\n",evs[ii].data.ptr,evs[ii].events);

            // 如果发生事件的是listensock，表示有新的客户端连上来。

            if (evs[ii].data.fd==listensock)

            {

                struct sockaddr\_in client;

                socklen\_t len = sizeof(client);

                int clientsock = accept(listensock,(struct sockaddr\*)&client,&len);

                printf ("accept client(socket=%d) ok.\n",clientsock);

                // 为新客户端准备读事件，并添加到epoll中。

                ev.data.fd=clientsock;

                ev.events=EPOLLIN;

                epoll\_ctl(epollfd,EPOLL\_CTL\_ADD,clientsock,&ev);

            }

            else

            {

                // 如果是客户端连接的socke有事件，表示有报文发过来或者连接已断开。

                char buffer[1024]; // 存放从客户端读取的数据。

                memset(buffer,0,sizeof(buffer));

                if (recv(evs[ii].data.fd,buffer,sizeof(buffer),0)<=0)

                {

                    // 如果客户端的连接已断开。

                    printf("client(eventfd=%d) disconnected.\n",evs[ii].data.fd);

                    close(evs[ii].data.fd);            // 关闭客户端的socket

                    // 从epollfd中删除客户端的socket，如果socket被关闭了，会自动从epollfd中删除，所以，以下代码不必启用。

                    // epoll\_ctl(epollfd,EPOLL\_CTL\_DEL,evs[ii].data.fd,0);

                }

                else

                {

                    // 如果客户端有报文发过来。

                    printf("recv(eventfd=%d):%s\n",evs[ii].data.fd,buffer);

                    // 把接收到的报文内容原封不动的发回去。

                    send(evs[ii].data.fd,buffer,strlen(buffer),0);

                }

            }

        }

    }

  return 0;

}

// 初始化服务端的监听端口。

int initserver(int port)

{

    int sock = socket(AF\_INET,SOCK\_STREAM,0);

    if (sock < 0)

    {

        perror("socket() failed"); return -1;

    }

    int opt = 1; unsigned int len = sizeof(opt);

    setsockopt(sock,SOL\_SOCKET,SO\_REUSEADDR,&opt,len);

    struct sockaddr\_in servaddr;

    servaddr.sin\_family = AF\_INET;

    servaddr.sin\_addr.s\_addr = htonl(INADDR\_ANY);

    servaddr.sin\_port = htons(port);

    if (bind(sock,(struct sockaddr \*)&servaddr,sizeof(servaddr)) < 0 )

    {

        perror("bind() failed"); close(sock); return -1;

    }

    if (listen(sock,5) != 0 )

    {

        perror("listen() failed"); close(sock); return -1;

    }

    return sock;

}

## 五、client1.cpp

// 网络通讯的客户端程序。

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <errno.h>

#include <string.h>

#include <netinet/in.h>

#include <sys/socket.h>

#include <arpa/inet.h>

#include <poll.h>

#include <fcntl.h>

// 把socket设置成非阻塞。

int setnonblocking(int fd)

{

    int  flags;

    // 获取fd的状态。

    if  ((flags=fcntl(fd,F\_GETFL,0))==-1)

        flags = 0;

    return fcntl(fd,F\_SETFL,flags|O\_NONBLOCK);

}

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

{

    if (argc != 3)

    {

        printf("usage:./client1 ip port\n"); return -1;

    }

    int sockfd;

    struct sockaddr\_in servaddr;

    char buf[1024];

    if ((sockfd=socket(AF\_INET,SOCK\_STREAM,0))<0) { printf("socket() failed.\n"); return -1; }

    setnonblocking(sockfd);          // 把sockfd设置成非阻塞。

    memset(&servaddr,0,sizeof(servaddr));

    servaddr.sin\_family=AF\_INET;

    servaddr.sin\_port=htons(atoi(argv[2]));

    servaddr.sin\_addr.s\_addr=inet\_addr(argv[1]);

    if (connect(sockfd, (struct sockaddr \*)&servaddr,sizeof(servaddr)) != 0)

    {

        if (errno!=EINPROGRESS)

        {

            printf("connect(%s:%s) failed.\n",argv[1],argv[2]); close(sockfd);  return -1;

        }

    }

    pollfd fds;

    fds.fd=sockfd;

    fds.events=POLLOUT;

    poll(&fds,1,-1);

    if (fds.revents==POLLOUT)

        printf("connect ok.\n");

    else

        printf("connect failed.\n");

return 0;

    // printf("开始时间：%d",time(0));

    for (int ii=0;ii<200000;ii++)

    {

        // 从命令行输入内容。

        memset(buf,0,sizeof(buf));

        printf("please input:"); scanf("%s",buf);

        // strcpy(buf,"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabbbbbbbbbbbbbbccccccccccccccccddddddddddddd");

        if (send(sockfd,buf,strlen(buf),0) <=0)

        {

            printf("write() failed.\n");  close(sockfd);  return -1;

        }

        memset(buf,0,sizeof(buf));

        if (recv(sockfd,buf,sizeof(buf),0) <=0)

        {

            printf("read() failed.\n");  close(sockfd);  return -1;

        }

        printf("recv:%s\n",buf);

    }

    // printf("结束时间：%d",time(0));

}

## 六、tcpepoll1.cpp

/\*

 \* 程序名：tcpepoll1.cpp，此程序用于临时的演示。

 \* 作者：吴从周

\*/

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <string.h>

#include <errno.h>

#include <sys/socket.h>

#include <sys/types.h>

#include <arpa/inet.h>

#include <sys/fcntl.h>

#include <sys/epoll.h>

// 把socket设置成非阻塞。

int setnonblocking(int fd)

{

    int  flags;

    // 获取fd的状态。

    if  ((flags=fcntl(fd,F\_GETFL,0))==-1)

        flags = 0;

    return fcntl(fd,F\_SETFL,flags|O\_NONBLOCK);

}

// 初始化服务端的监听端口。

int initserver(int port);

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

{

    if (argc != 2) { printf("usage: ./tcpepoll1 port\n"); return -1; }

    // 初始化服务端用于监听的socket。

    int listensock = initserver(atoi(argv[1]));

    printf("listensock=%d\n",listensock);

    setnonblocking(listensock);     // 把监听的socket设置为非阻塞。

    while (true)

    {

        if (accept(listensock,0,0)==-1)

        {

            if (errno!=EAGAIN)

            {

                perror("accept:");   return -1;

            }

        }

        else

            break;

    }

    printf("客户端已连接。\n");

return 0;

    if (listensock < 0) { printf("initserver() failed.\n"); return -1; }

    // 创建epoll句柄。

    int epollfd=epoll\_create(1);

    // 为服务端的listensock准备读事件。

    epoll\_event ev;              // 声明事件的数据结构。

    ev.data.fd=listensock;   // 指定事件的自定义数据，会随着epoll\_wait()返回的事件一并返回。

    // ev.data.ptr=(void\*)"超女";   // 指定事件的自定义数据，会随着epoll\_wait()返回的事件一并返回。

    ev.events=EPOLLIN;      // 打算让epoll监视listensock的读事件。

    epoll\_ctl(epollfd,EPOLL\_CTL\_ADD,listensock,&ev);     // 把需要监视的socket和事件加入epollfd中。

    epoll\_event evs[10];      // 存放epoll返回的事件。

    while (true)        // 事件循环。

    {

        // 等待监视的socket有事件发生。

        int infds=epoll\_wait(epollfd,evs,10,-1);

        // 返回失败。

        if (infds < 0)

        {

            perror("epoll() failed"); break;

        }

        // 超时。

        if (infds == 0)

        {

            printf("epoll() timeout.\n"); continue;

        }

        // 如果infds>0，表示有事件发生的socket的数量。

        for (int ii=0;ii<infds;ii++)       // 遍历epoll返回的数组evs。

        {

            // printf("ptr=%s,events=%d\n",evs[ii].data.ptr,evs[ii].events);

            // 如果发生事件的是listensock，表示有新的客户端连上来。

            if (evs[ii].data.fd==listensock)

            {

                struct sockaddr\_in client;

                socklen\_t len = sizeof(client);

                int clientsock = accept(listensock,(struct sockaddr\*)&client,&len);

                printf ("accept client(socket=%d) ok.\n",clientsock);

                // 为新客户端准备读事件，并添加到epoll中。

                ev.data.fd=clientsock;

                ev.events=EPOLLIN;

                epoll\_ctl(epollfd,EPOLL\_CTL\_ADD,clientsock,&ev);

            }

            else

            {

                // 如果是客户端连接的socke有事件，表示有报文发过来或者连接已断开。

                char buffer[1024]; // 存放从客户端读取的数据。

                memset(buffer,0,sizeof(buffer));

                if (recv(evs[ii].data.fd,buffer,sizeof(buffer),0)<=0)

                {

                    // 如果客户端的连接已断开。

                    printf("client(eventfd=%d) disconnected.\n",evs[ii].data.fd);

                    close(evs[ii].data.fd);            // 关闭客户端的socket

                    // 从epollfd中删除客户端的socket，如果socket被关闭了，会自动从epollfd中删除，所以，以下代码不必启用。

                    // epoll\_ctl(epollfd,EPOLL\_CTL\_DEL,evs[ii].data.fd,0);

                }

                else

                {

                    // 如果客户端有报文发过来。

                    printf("recv(eventfd=%d):%s\n",evs[ii].data.fd,buffer);

                    // 把接收到的报文内容原封不动的发回去。

                    send(evs[ii].data.fd,buffer,strlen(buffer),0);

                }

            }

        }

    }

  return 0;

}

// 初始化服务端的监听端口。

int initserver(int port)

{

    int sock = socket(AF\_INET,SOCK\_STREAM,0);

    if (sock < 0)

    {

        perror("socket() failed"); return -1;

    }

    int opt = 1; unsigned int len = sizeof(opt);

    setsockopt(sock,SOL\_SOCKET,SO\_REUSEADDR,&opt,len);

    struct sockaddr\_in servaddr;

    servaddr.sin\_family = AF\_INET;

    servaddr.sin\_addr.s\_addr = htonl(INADDR\_ANY);

    servaddr.sin\_port = htons(port);

    if (bind(sock,(struct sockaddr \*)&servaddr,sizeof(servaddr)) < 0 )

    {

        perror("bind() failed"); close(sock); return -1;

    }

    if (listen(sock,5) != 0 )

    {

        perror("listen() failed"); close(sock); return -1;

    }

    return sock;

}

## 七、tcpepoll2.cpp

/\*

 \* 程序名：tcpepoll2.cpp，此程序用于演示采用epoll模型的边缘触发。

 \* 作者：吴从周

\*/

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <string.h>

#include <errno.h>

#include <sys/socket.h>

#include <sys/types.h>

#include <arpa/inet.h>

#include <sys/fcntl.h>

#include <sys/epoll.h>

// 把socket设置成非阻塞。

int setnonblocking(int fd)

{

    int  flags;

    // 获取fd的状态。

    if  ((flags=fcntl(fd,F\_GETFL,0))==-1)

        flags = 0;

    return fcntl(fd,F\_SETFL,flags|O\_NONBLOCK);

}

// 初始化服务端的监听端口。

int initserver(int port);

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

{

    if (argc != 2) { printf("usage: ./tcpepoll2 port\n"); return -1; }

    // 初始化服务端用于监听的socket。

    int listensock = initserver(atoi(argv[1]));

    printf("listensock=%d\n",listensock);

    if (listensock < 0) { printf("initserver() failed.\n"); return -1; }

    setnonblocking(listensock);             // 把listensock设置为非阻塞。

    // 创建epoll句柄。

    int epollfd=epoll\_create(1);

    // 为服务端的listensock准备读事件。

    epoll\_event ev;              // 声明事件的数据结构。

    ev.data.fd=listensock;   // 指定事件的自定义数据，会随着epoll\_wait()返回的事件一并返回。

    // ev.data.ptr=(void\*)"超女";   // 指定事件的自定义数据，会随着epoll\_wait()返回的事件一并返回。

    ev.events=EPOLLIN;      // 打算让epoll监视listensock的读事件，LT（水平）模式。

    // ev.events=EPOLLIN | EPOLLET;      // 打算让epoll监视listensock的读事件，ET（边缘）模式。

    epoll\_ctl(epollfd,EPOLL\_CTL\_ADD,listensock,&ev);     // 把需要监视的socket和事件加入epollfd中。

    epoll\_event evs[10];      // 存放epoll返回的事件。

    while (true)        // 事件循环。

    {

        // 等待监视的socket有事件发生。

        int infds=epoll\_wait(epollfd,evs,10,-1);

        // 返回失败。

        if (infds < 0)

        {

            perror("epoll() failed"); break;

        }

        // 超时。

        if (infds == 0)

        {

            printf("epoll() timeout.\n"); continue;

        }

        // 如果infds>0，表示有事件发生的socket的数量。

        for (int ii=0;ii<infds;ii++)       // 遍历epoll返回的数组evs。

        {

            // printf("ptr=%s,events=%d\n",evs[ii].data.ptr,evs[ii].events);

            // 如果发生事件的是listensock，表示有新的客户端连上来。

            if (evs[ii].data.fd==listensock)

            {

                while (true)

                {

                    struct sockaddr\_in client;

                    socklen\_t len = sizeof(client);

                    int clientsock = accept(listensock,(struct sockaddr\*)&client,&len);

                    if ( (clientsock<0) && (errno==EAGAIN) ) break;

                    printf ("accept client(socket=%d) ok.\n",clientsock);

                    // 为新客户端准备读事件，并添加到epoll中。

                    setnonblocking(clientsock);              // 把客户端连接的socket设置为非阻塞。

                    ev.data.fd=clientsock;

                    // ev.events=EPOLLOUT;                       // LT-水平触发。

                    ev.events=EPOLLOUT|EPOLLET;        // ET-边缘触发。

                    epoll\_ctl(epollfd,EPOLL\_CTL\_ADD,clientsock,&ev);

                }

            }

            else

            {

                printf("触发了写件事。\n");

                for (int ii=0;ii<10000000;ii++)

                {

                    if (send(ev.data.fd,"aaaaaaaaaaaaaaaaaaaaaaaaaaabbbbbbbbbbbbbb",30,0)<=0)

                    {

                        if (errno==EAGAIN)

                        {

                            printf("发送缓冲区已填懣。\n"); break;

                        }

                    }

                }

                /\*

                // 如果是客户端连接的socke有事件，表示有报文发过来或者连接已断开。

                char buffer[1024];       // 存放从客户端读取的数据。

                memset(buffer,0,sizeof(buffer));

                int    readn;                 // 每次调用recv()的返回值。

                char \*ptr=buffer;        // buffer的位置指针。

                while (true)

                {

                    if ( (readn=recv(evs[ii].data.fd,ptr,5,0))<=0 )

                    {

                        if ( (readn<0) && (errno==EAGAIN) )

                        {   // 如果数据被读取完了，把接收到的报文内容原封不动的发回去。

                            send(evs[ii].data.fd,buffer,strlen(buffer),0);

                            printf("recv(eventfd=%d):%s\n",evs[ii].data.fd,buffer);

                        }

                        else

                        {

                            // 如果客户端的连接已断开。

                            printf("client(eventfd=%d) disconnected.\n",evs[ii].data.fd);

                            close(evs[ii].data.fd);            // 关闭客户端的socket

                        }

                        break;        // 跳出循环。

                    }

                    else

                        ptr=ptr+readn;                    // buffer的位置指针后移。

                }

                \*/

            }

        }

    }

  return 0;

}

// 初始化服务端的监听端口。

int initserver(int port)

{

    int sock = socket(AF\_INET,SOCK\_STREAM,0);

    if (sock < 0)

    {

        perror("socket() failed"); return -1;

    }

    int opt = 1; unsigned int len = sizeof(opt);

    setsockopt(sock,SOL\_SOCKET,SO\_REUSEADDR,&opt,len);

    struct sockaddr\_in servaddr;

    servaddr.sin\_family = AF\_INET;

    servaddr.sin\_addr.s\_addr = htonl(INADDR\_ANY);

    servaddr.sin\_port = htons(port);

    if (bind(sock,(struct sockaddr \*)&servaddr,sizeof(servaddr)) < 0 )

    {

        perror("bind() failed"); close(sock); return -1;

    }

    if (listen(sock,5) != 0 )

    {

        perror("listen() failed"); close(sock); return -1;

    }

    return sock;

}

## 八、client2.cpp

// 网络通讯的客户端程序。

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <errno.h>

#include <string.h>

#include <netinet/in.h>

#include <sys/socket.h>

#include <arpa/inet.h>

#include <time.h>

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

{

    if (argc != 3)

    {

        printf("usage:./client2 ip port\n"); return -1;

    }

    int sockfd;

    struct sockaddr\_in servaddr;

    char buf[1024];

    if ((sockfd=socket(AF\_INET,SOCK\_STREAM,0))<0) { printf("socket() failed.\n"); return -1; }

    memset(&servaddr,0,sizeof(servaddr));

    servaddr.sin\_family=AF\_INET;

    servaddr.sin\_port=htons(atoi(argv[2]));

    servaddr.sin\_addr.s\_addr=inet\_addr(argv[1]);

    if (connect(sockfd, (struct sockaddr \*)&servaddr,sizeof(servaddr)) != 0)

    {

        printf("connect(%s:%s) failed.\n",argv[1],argv[2]); close(sockfd);  return -1;

    }

    printf("connect ok.\n");

    // printf("开始时间：%d",time(0));

    for (int ii=0;ii<200000;ii++)

    {

        // 从命令行输入内容。

        memset(buf,0,sizeof(buf));

        printf("please input:"); scanf("%s",buf);

        // strcpy(buf,"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabbbbbbbbbbbbbbccccccccccccccccddddddddddddd");

        if (send(sockfd,buf,strlen(buf),0) <=0)

        {

            printf("write() failed.\n");  close(sockfd);  return -1;

        }

        memset(buf,0,sizeof(buf));

        if (recv(sockfd,buf,sizeof(buf),0) <=0)

        {

            printf("read() failed.\n");  close(sockfd);  return -1;

        }

        printf("recv:%s\n",buf);

    }

    // printf("结束时间：%d",time(0));

}