[TCP/IP基础 1](#_Toc513552957)

[1.OSI模型 1](#_Toc513552958)

[2.TCP/IP四层模型 2](#_Toc513552959)

[4.端口 7](#_Toc513552960)

[5.MTU 8](#_Toc513552961)

[6. ICMP协议 9](#_Toc513552962)

[7 ARP协议 10](#_Toc513552963)

[8 数据在网络中传输过程 11](#_Toc513552964)

[9 IP协议 12](#_Toc513552965)

[10网际校验和 13](#_Toc513552966)

[11 路由 14](#_Toc513552967)

[12 滑动窗口协议 15](#_Toc513552968)

[13 UDP协议 15](#_Toc513552969)

[Socket编程 17](#_Toc513552970)

[1 地址结构 17](#_Toc513552971)

[1.1 IPv4套接口地址 17](#_Toc513552972)

[1.2 通用地址结构 17](#_Toc513552973)

[2 网络字节序 17](#_Toc513552974)

[2.1 字节序 17](#_Toc513552975)

[2.2 主机字节序 18](#_Toc513552976)

[2.3 网络字节序 18](#_Toc513552977)

[3 字节序转换函数 18](#_Toc513552978)

[4 地址转换函数 19](#_Toc513552979)

[5 套接字类型 20](#_Toc513552980)

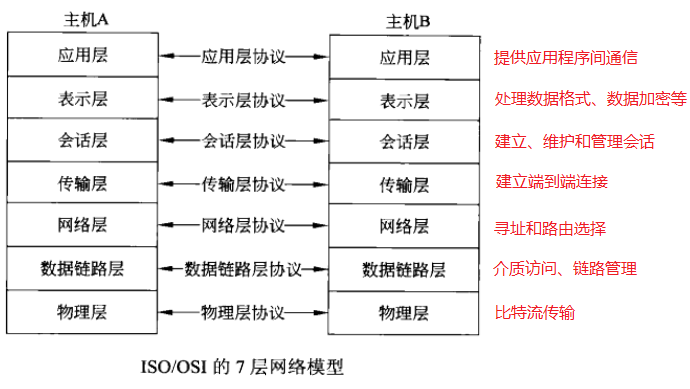
[6 TCP 客户端/服务端模型 20](#_Toc513552981)

[6.1 socket函数 21](#_Toc513552982)

# TCP/IP基础

## 1.OSI模型

OSI模型是开放系统互联模型，由ISO制定，的网络分层模型，共七层



物理层：

物理层定义了所有电子及物理设备的规范，为上层的传输提供了一个物理介质，本层中w数据传输的单位为比特（bit）。属于本层定义的规范有EIA/TIA RS-232、EIA/TIA RS-449、V.35、RJ-45等，实际使用中的设备如网卡等属于本层

数据链路层（Data Link Layer）：对物理层收到的比特流进行数据成帧。提供可靠的数据传输服务，实现无差错数据传输。在数据链路层中数据的单位为帧（frame）。属于本层定义的规范有SDLC、HDLC、PPP、STP、帧中继等，实际使用中的设备如switch交换机属于本层。

网络层（Network Layer）：网络层负责将各个子网之间的数据进行路由选择，分组与重组。本层中数据传输的单位为数据包（packet）。属于本层定义的规范有IP、IPX、RIP、OSPF、ICMP、IGMP等。实际使用中的设备如路由器属于本层。

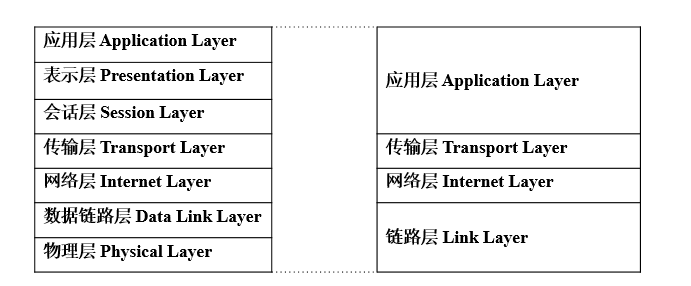
传输层（Transport Layer）：提供可靠的数据传输服务，它检测路由器丢弃的包，然后产生一个重传请求，能够将乱序收到的数据包重新排序。

会话层（Session Layer）：管理主机之间会话过程，包括会话建立、终止和会话过程中的管理。

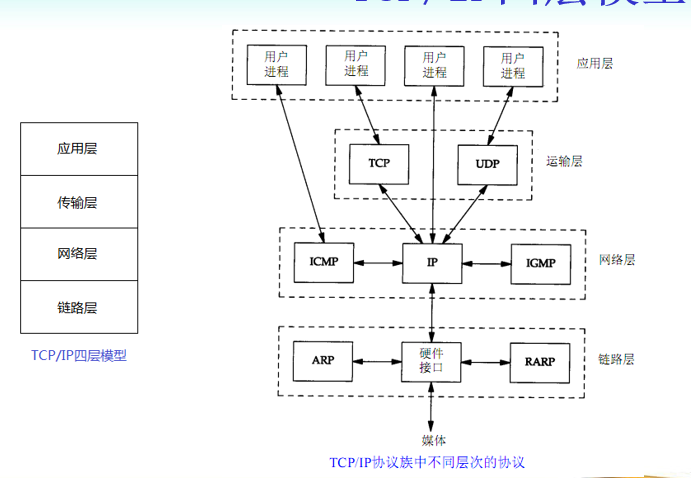
表示层（Presentation Layer）：表示层对网络传输的数据进行变换，使得多个主机之间传送的信息能够互相理解，包括数据的压缩、加密、格式转换等。

应用层（Application Layer）：应用层与应用程序界面沟通，以达至展示给用户的目的。 在此常见的协定有: HTTP，HTTPS，FTP，TELNET，SSH，SMTP，POP3等

## 2.TCP/IP四层模型



以上即为TCP/IP的四层模型，它讲OSI模型进行了简化



如上图所示，不同的层次有不同的网络协议作用

在链路层：

ARP协议：它能够根据IP地址获取物理地址。

RARP协议：反向地址转换协议，它和ARP的效果刚好相反

说明链路层主要是实现IP和MAC地址的交换

在网络层层：

ICMP协议:用于在IP主机和路由器之间传递控制消息，包括报告错误，交换受限控制和状态信息。它是面向无连接的协议

IGMP协议：所有IP组播系统(包括主机和路由器)都需要支持IGMP协议

IP协议：将多个包交换网络连接起来，它在源地址和目的地址之间传送一种称之为数据包的东西。

在运输层：

TCP协议：

TCP协议头部：



源端口号和目的端口号用于区别主机中的不同进程，而IP地址用来区分不同主机，源端口号和目的端口号配合IP首部中的源IP地址和目的地址IP就能确定一个唯一的TCP连接

32位序号用来标识从TCP发端向TCP收端发送的数据字节流，它表示在这个报文段中的第一个数据字节在数据流中的序号，主要用来解决网络报乱序的问题

32位确认序号包含发送确认的一端所期望收到的下一个序号，因此确认序号应该是上次已成功收到的数据字节序号加1，不过只有当标志位中的ACK标志位1才确认序列号的字段有效，主要解决不丢包的问题

首部长度：给出首部中32bit字的数目，需要这个值是因为任选字段的长度是可变的，这个字段占4bit(最多能表示15个32bit的字，即4\*15=60个字节的首部长度)，因此TCP最多有60字节的首部

标志位：

URG：此标志表示TCP包的紧急指针域(后面马上就要说到)有效，用来保证TCP连接不被中断，并且督促 中间层设备要尽快处理这些数据;

ACK：此标志表示应答域有效，就是说前面所说的TCP应答号将会包含在TCP数据包中;有两个取值：0和1， 为1的时候表示应答域有效，反之为0;

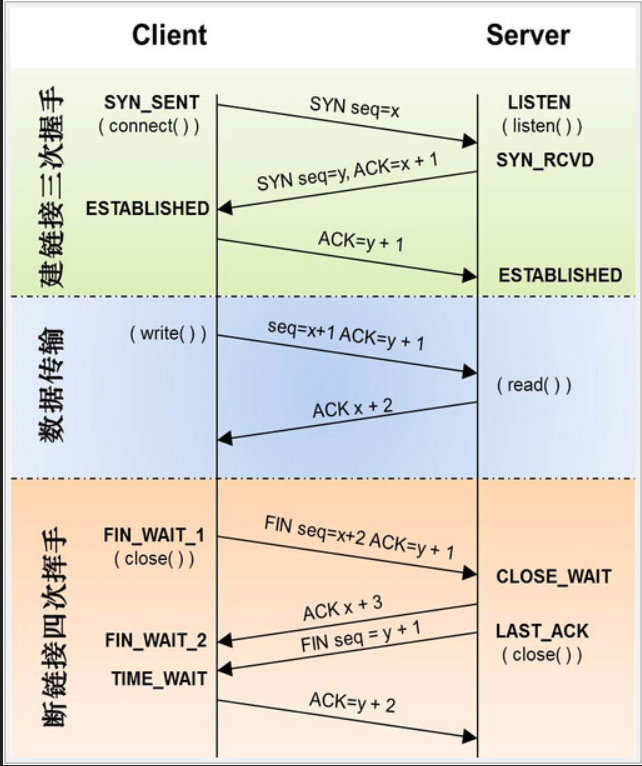
PSH：这个标志位表示Push操作。所谓Push操作就是指在数据包到达接收端以后，立即传送给应用程序， 而不是在缓冲区中排队

RST：这个标志表示连接复位请求。用来复位那些产生错误的连接，也被用来拒绝错误和非法的数据包;

SYN：表示同步序号，用来建立连接。SYN标志位和ACK标志位搭配使用，当连接请求的时候，SYN=1， ACK=0;连接被响应的时候，SYN=1，ACK=1;这个标志的数据包经常被用来进行端口扫描。扫描者发送 一个只有SYN的数据包，如果对方主机响应了一个数据包回来 ，就表明这台主机存在这个端口;但是由于这 种扫描方式只是进行TCP三次握手的第一次握手，因此这种扫描的成功表示被扫描的机器不很安全，一台安全 的主机将会强制要求一个连接严格的进行TCP的三次握手

FIN： 表示发送端已经达到数据末尾，也就是说双方的数据传送完成，没有数据可以传送了，发送FIN标志 位的TCP数据包后，连接将被断开。这个标志的数据包也经常被用于进行端口扫描。

窗口大小，也就是有名的滑动窗口，用来进行流量控制



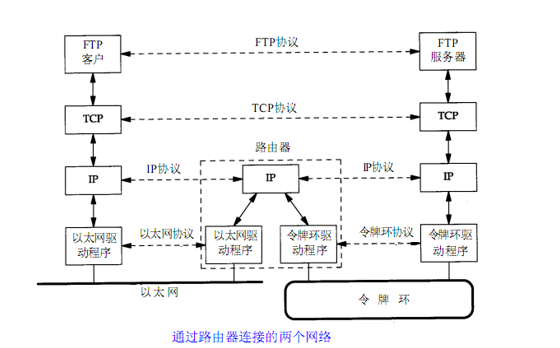
三次握手：TCP是面向连接的，无论哪一方向发送数据之前，都必须在双方之间建立一条连接。在TCP/IP协议中，TCP协议提供可靠的连接服务。连接通过三次握手初始化。

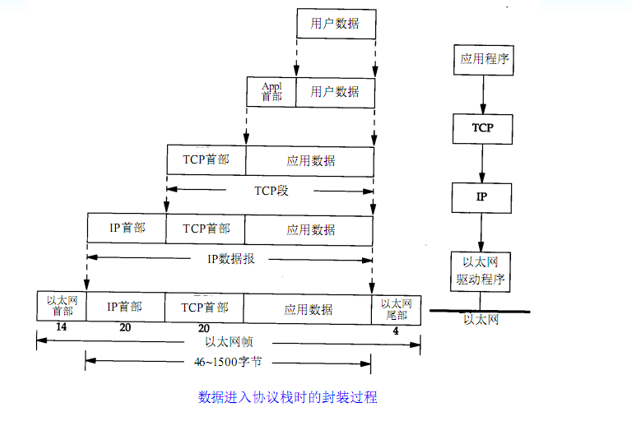
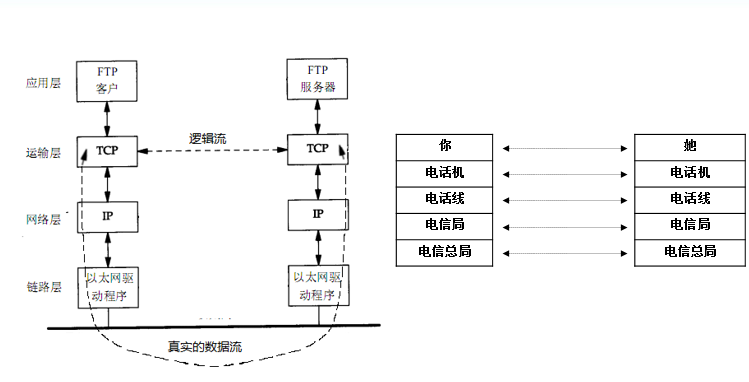
三次握手的目的是同步连接双方的序列号和确认号并交换TCP窗口的大小信息。

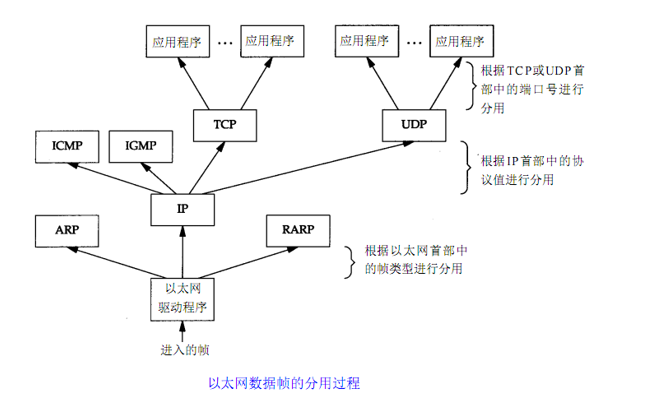
1.第一次握手：建立连接。客户端发送连接请求报文段，将SYN位置为1，Sequence Number为x;然后，客户端进入SYN\_SEND状态，等待服务器的确认;

2.第二次握手：服务器收到SYN报文段。服务器收到客户端的SYN报文段，需要对这个SYN报文段进行确认，设置Acknowledgment Number为x+1(Sequence Number+1);同时，自己自己还要发送SYN请求信息，将SYN位置为1，Sequence Number为y;服务器端将上述所有信息放到一个报文段(即SYN+ACK报文段)中，一并发送给客户端，此时服务器进入SYN\_RECV状态;

3.第三次握手：客户端收到服务器的SYN+ACK报文段。然后将Acknowledgment Number设置为y+1，向服务器发送ACK报文段，这个报文段发送完毕以后，客户端和服务器端都进入ESTABLISHED状态，完成TCP三次握手。







## 4.端口

众所周知端口（Well Known Ports）：从0到1023，这些端口由IANA分配和控制它们紧密绑定于一些服务。通常这些端口的通讯明确表明了某种服务的协议。例如：21端口为ftp服务端口。

注册端口（Registered Ports）：从1024到49151。它些端口不受IANA控制，但由IANA登记并提供使用情况清单。它们松散地绑定于一些服务。也就是说有许多服务绑定于这些端口，这些端口同样用于许多其它目的。例如：1433 Microsoft SQL服务端口

动态或私有端口（Dynamic or Private Ports）：从49152到65535。IANA不管这些端口。实际上，机器通常从1024起分配动态端口。但也有例外：SUN的RPC端口从32768开始

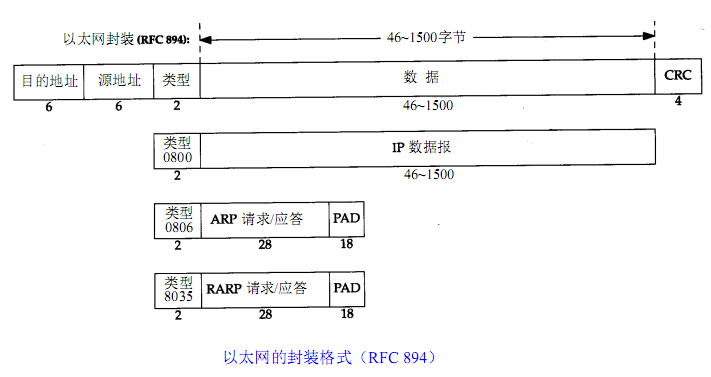
## 5.MTU

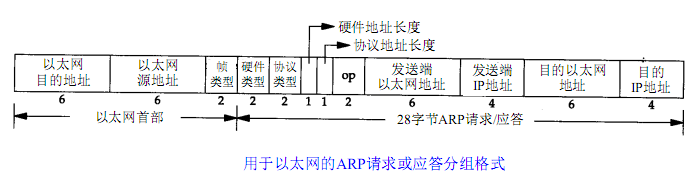
以太网和IEEE 802.3对数据帧的长度都有限制，其最大值分别是1500和1492字节，将这个限制称作最大传输单元（MTU）

如果IP层有一个数据报要传，而且数据的长度比链路层的MTU还大，那么IP层就要进行分片,把数据报分成若干片，这样每一片都小于MTU

当网络上的两台主机相互之间进行通信时，两台主机之间要经过多个网络，每个网络的链路层可能有不同的MTU，其中两台通信主机路径中的最小MTU称为路径MTU

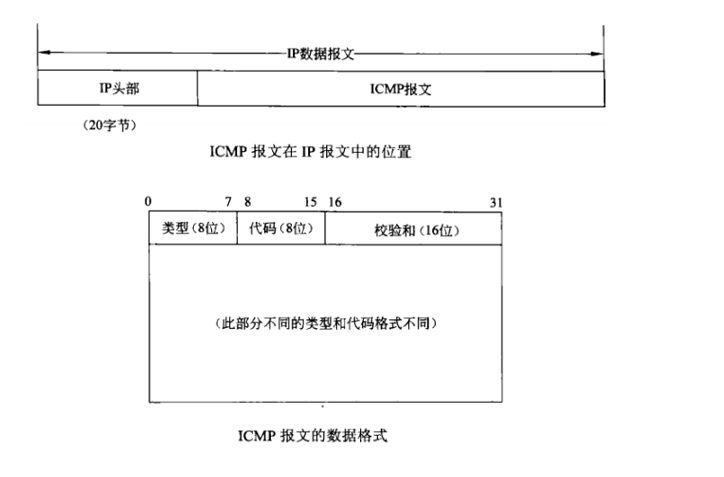
以太网帧格式





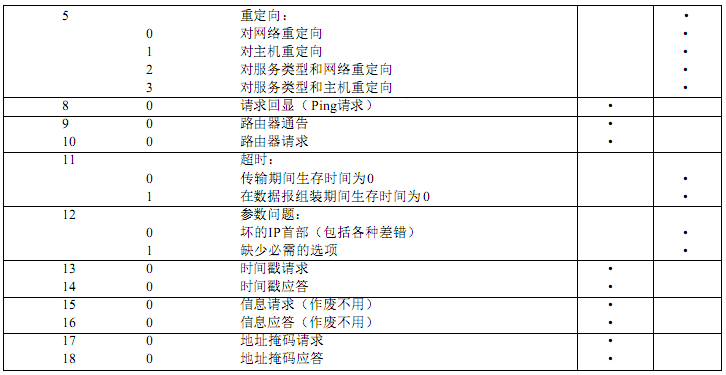
## 6. ICMP协议

ICMP协议用于传递差错信息，时间，回显，网络，信息等控制数据



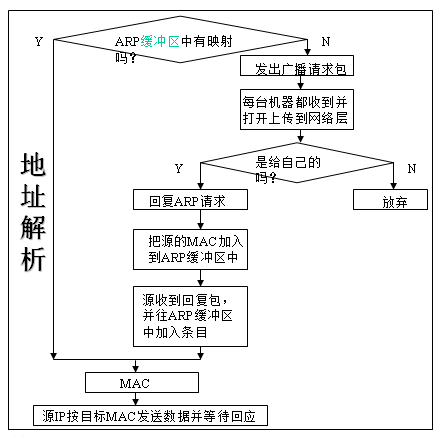
差错代码



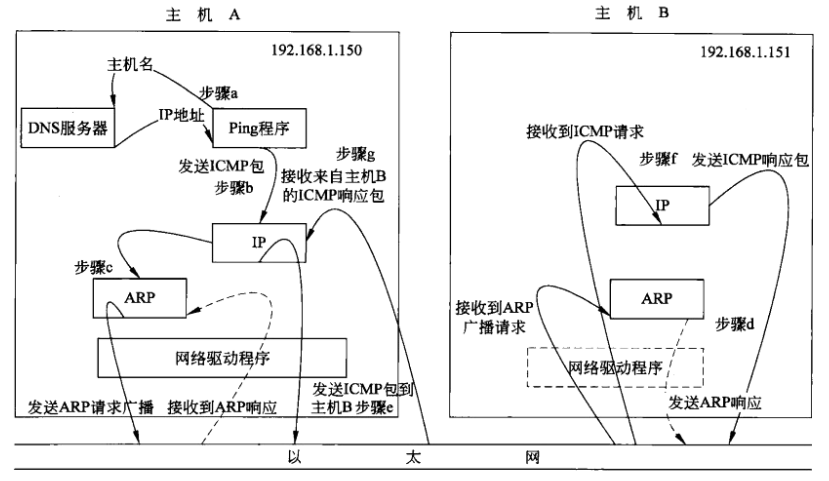


## 7 ARP协议

即地址解析协议，用来映射IP到MAC地址，解析过程



## 8 数据在网络中传输过程



a.应用程序ping会判断发送的是主机名还是IP地址，调用函数gethostbyname()解析主机B，将主机名转换成一个32的IP地址，这个过程叫DNS的域名解析。

b.ping程序向目的IP地址发送一个ICMP的ECHO包

c.将目标主机的IP地址转换为48位硬件地址，在局域网内发送ARP的请求广播，查找主机B的硬件地址

d.主机B的ARP的协议层接收到主机A的ARP请求后，将本机的硬件地址填充到应答包，发送ARP应答到主机A

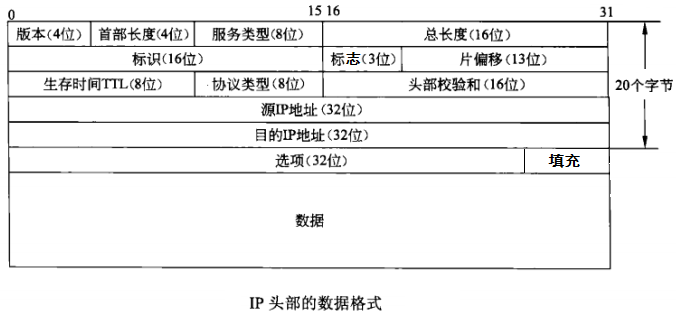
e.发送ICMP数据包到主机B

f.主机B接收到主机A的ICMP包，发送响应包

g.主机A接收到B的ICMP包响应包

## 9 IP协议

IP数据格式



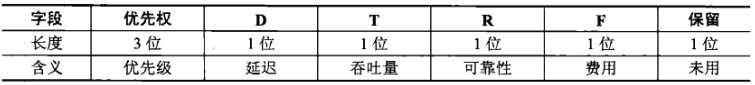
版本：IP协议版本号，长度为4位，IPv4此字段值为4，IPv6此字段值为6

首部长度：以32位的字为单位，该字段长度为4位，最大值为15，所以首部长度最大为60字节

服务类型(TOS)

长度为8位。此字段包含3位的优先权(已忽略),4位的服务类型子字段和1位的保留位(必须置0)

4位的服务类型分别位最小延迟(D),最大吞吐量(T),最高可靠性(R),最小费用(F)



TTL:

表示数据报最多可以经过的路由器的数量，数据报每经过一个路由器，TTL减1，减位0时丢弃，并发送ICMP报文通知源主机。TTL可以避免数据报在路由器之间不断循环

协议类型：

表示IP层上承载哪个高级协议，在封装和分用的过程中，协议栈知道改交给哪个层的协议处理。1 ICMP ; 2 IGMP ; 6 TCP; 17 UDP

头部校验和

保证数据报头部的数据完整性，但校验不包括数据部分。这样做的目的有二：一是所有将数据封装在IP数据包中的高层协议均含有覆盖整个数据的校验和，因此IP数据报没有必要再对其所承载的数据部分进行校验。二是每经过一个路由器，IP数据报的头部要发生改变，而数据部分不变，这样只对发生改变的头部进行校验，显然不会浪费太多的时间。为了减少计算时间，一般不用CRC校验码，而是采用更简单的网际校验和。

源IP地址

发送数据的主机IP地址

目的IP地址

收数据的主机IP地址

选项与填充（选项为4字节整数倍，否则用0填充）

安全和处理限制

路径记录：记录所经历路由器的IP地址

时间戳：记录所经历路由器的IP地址和时间

宽松源站路由：指定数据报文必须经历的IP地址，可以经过没有指定的IP地址。

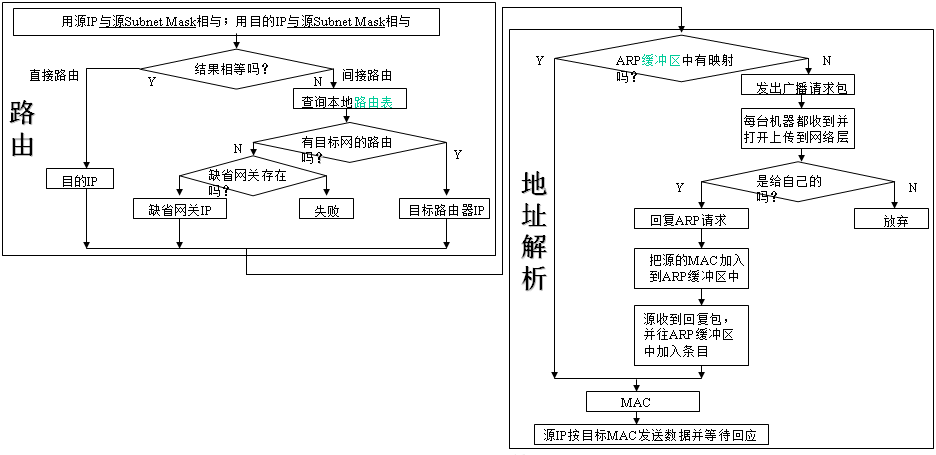
严格的源站路由：指定数据报文必须经历的IP地址，不能经过没有指定的IP地址。

## 10网际校验和

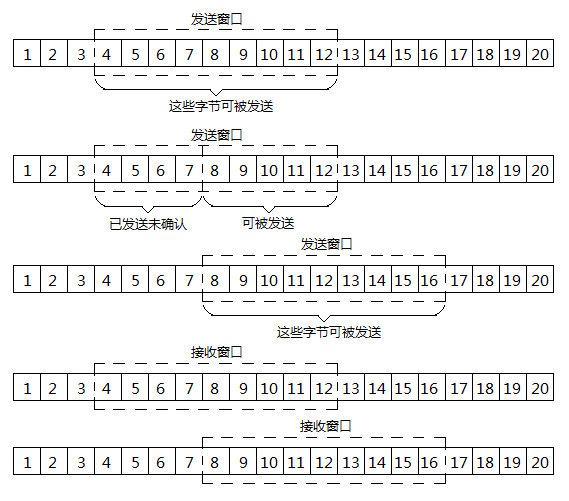
发送方对要发送的数据划分为许多16位字序列（如果数据的字节数为奇数，则在末尾补一字节0凑成偶数。）对这些序列反码求和，便得到校验和。

如果数据在传输过程中没有发生任何差错，那么接收方计算校验和的结果应该为全1。如果结果不是全1（即校验和错误）。

## 11 路由



## 12 滑动窗口协议



通告接收窗口（rwnd）：预防应用程序发送的数据超过对方的缓冲区。接收方使用的流量控制

拥塞窗口（cwnd）：预防应用程序发送的数据超过网络所能承受的能力。发送方使用的流量控制

发送窗口取两者较小值

慢启动阀值（ssthresh：slow start threshold）

慢启动阶段：cwnd从1开始按指数增长直到ssthresh

拥塞避免阶段：cwnd按线性增长，直到拥塞，将cwnd=1，ssthresh减半。

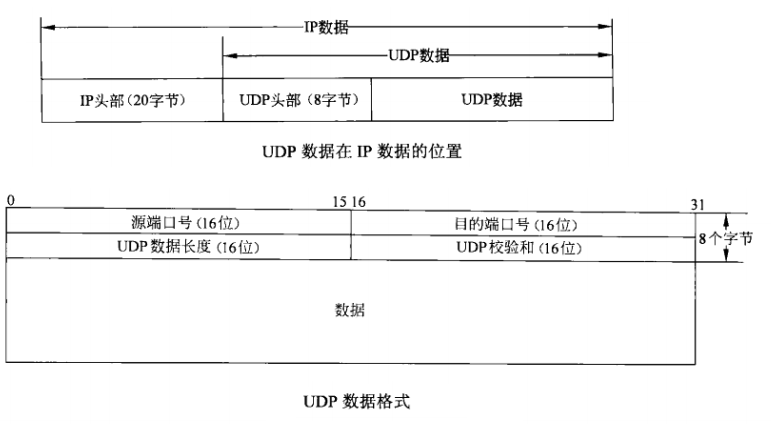
## 13 UDP协议

无连接

不可靠

一般情况下UDP更加高效

报文格式



# Socket编程

socket可以看做是用户进程与内核网络协议栈的编程接口

socket不仅可以用于本机的进程间通信，还可以用于网络上不同主机的进程间通信

## 1 地址结构

### 1.1 IPv4套接口地址

IPv4套接口地址结构通常也称为”网际套接字地址结构”,它以sockaddr\_in命名，定义在头文件<netinet/in.h>中

struct sockaddr\_in{

uint8\_t sin\_len;//整个sockaddr\_in结构体的长度

sa\_family sin\_family;//指定该地址家族，在这里必须设位AF\_INET

in\_port\_t sin\_port;//端口

struct in\_addr sin\_addr;//IPv4的地址

char sin\_zero[8];//暂不使用，一般将其设置为0

}

### 1.2 通用地址结构

通用地址结构用来指定与套接字关联的地址

struct sockaddr{

uint8\_t sin\_len;//整个sockaddr结构体的长度

sa\_family\_t sin\_family;//指定该地址家族

char sa\_data[14]//由sin\_family决定它的形式

}

## 2 网络字节序

### 2.1 字节序

大端字节序

最高有效位存储于最低内存地址处

最低有效位存储于最高内存地址处

小端字节序

最高有效位存储于最高内存地址处，最低有效位存储最低内存地址处

### 2.2 主机字节序

不同的主机有不同的字节序，如X86为小端字节序，Motorola 6800为大端字节序，ARM字节序是可配置的

### 2.3 网络字节序

网络字节序规定为大端字节序

## 3 字节序转换函数

uint32\_t htonl(uint32\_t hostlong)//将主机字节序转换为长整型的网络字节序(32)

uint16\_t htons(uint16\_t hostshort)//16位

uint32\_t ntohl(uint32\_t netlong)//将网络字节序转换为长整型的主机字节序(32)

uint16\_t ntohs(uint16\_t netshort)//16位

在上述函数中，h代表host，n代表network s代表short，l代表long

1 #include <stdio.h>

2 #include <arpa/inet.h>

3

4 int main(void)

5 {

6 unsigned int x = 0x12345678;

7 unsigned char \*p = (unsigned char\*)&x;

8 printf("%0x %0x %0x %0x\n", p[0], p[1], p[2], p[3]);

9

10 unsigned int y = htonl(x);

11 p = (unsigned char\*)&y;

12 printf("%0x %0x %0x %0x\n", p[0], p[1], p[2], p[3]);

13

14 return 0;

15 }

输出： 78 56 34 12

12 34 56 78

## 4 地址转换函数

#include<netinet/in.h>

#include<arpa/inet.h>

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

//将一个字符串IP地址转换为32位的网络系列IP地址，成功返回非零，如果输入的地址不正确会返回零

in\_addr\_t inet\_addr(const char\*cp)//将一个点分十进制转换为二进制的数，用于ipv4的IP转换

char\* inet\_ntoa(struct in\_addr in)//返回点分十进制的字符串在静态内存中的指针

1 #include <stdio.h>

2 #include <arpa/inet.h>

3

4 int main(void)

5 {

6 unsigned long addr = inet\_addr("192.168.0.100");

7 printf("addr=%u\n", ntohl(addr));

8

9 struct in\_addr ipaddr;

10 ipaddr.s\_addr = addr;

11 printf("%s\n", inet\_ntoa(ipaddr));

12 return 0;

13 }

输出：addr=3232235620

192.168.0.100

## 5 套接字类型

流式套接字(SOCK\_STREAM)

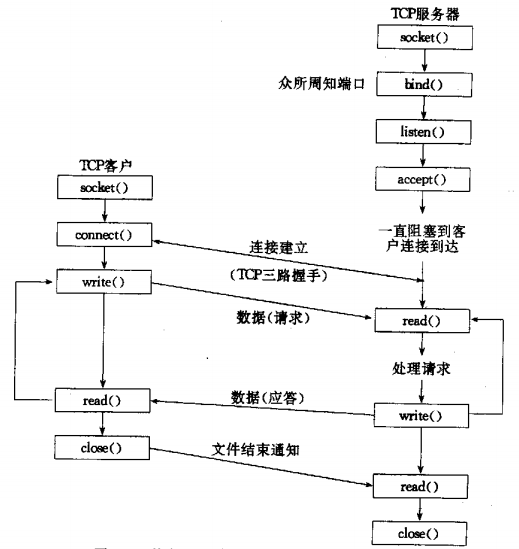
提供面向连接的，可靠的数据传输服务，数据无差错，无重复的发送，且按发送顺序接收

数据报式套接字(SOCK\_DGRAM)

数据无连接服务，不提供无错保证，数据可能丢失或重复，并且接收顺序混乱

原始套接字(SOCK\_RAW)

## 6 TCP 客户端/服务端模型



### 6.1 socket函数

包含头文件<sys/socket.h>

功能：创建一个套接字用于通信

函数原型：

#include <sys/types.h> /\* See NOTES \*/

#include <sys/socket.h>

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

参数

domain:指定通信协议族

如 AF\_UNIX,AF\_LOCAL

AF\_INET ipv4

AF\_INET6 ipv6

AF\_IPX ipx等等

type:指定socket类型

SOCK\_STREAM 流式套接字

SOCK\_DGRAM 数据报套接字

SOCK\_RAW 原始套接字

protocol：协议类型，一般为0

成功返回非负整数，它与文件描述符类似，我们称它为套接口描述字，失败返回-1.

### 6.2 bind函数

这个函数主要用于绑定一个本地地址到套接字

函数原型

#include <sys/types.h> /\* See NOTES \*/

#include <sys/socket.h>

int bind(int sockfd, const struct sockaddr \*addr,

socklen\_t addrlen);

参数：sockfd socket返回的套接字

addr 要绑定的地址

addrlen 地址长度

成功返回0，失败返回-1

### 6.3 listen函数

这个函数将套接字用于监听进入的连接

原型：

#include <sys/types.h> /\* See NOTES \*/

#include <sys/socket.h>

int listen(int sockfd, int backlog);

参数：

sockfd: socket函数返回的套接字

backlog 规定内核为此套接字排队的最大连接个数

成功返回0，失败返回-1

一般来说，listen函数应该在调用socket和bind函数之后，调用函数accept之前调用。

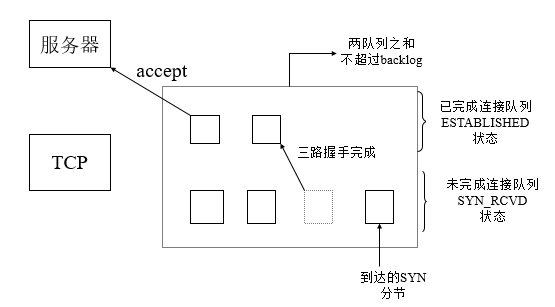
对于给定的监听套接口，内核要维护两个队列

1.已由客户发出并到达服务器，服务器正在等待完成的TCP三路握手过程

2.已完成连接的队列

它主要是将套接字变为被动套接字，让它等着客户端的连接，不再去主动连接，它将套接字和套接字对应的连接队列长度告诉系统内核，然后就结束不会阻塞，这样当有一个客户端主动连接(connect())主动连接，Linux内核就自动完成TCP三次握手，将建立好的连接自动存储到队列中，如此重复

如下面的连接过程：



### 6.4 accept()函数

这个函数从已连接的队列返回第一个连接，如果已完成的连接队列为空，

则阻塞

原型：

#include <sys/types.h> /\* See NOTES \*/

#include <sys/socket.h>

int accept(int sockfd, struct sockaddr \*addr, socklen\_t \*addrlen);

参数：

sockfd 服务器套接字

addr 返回对等方的套接字地址

addrlen 返回对等方的套接字地址长度

返回值：成功返回非负整数，失败返回-1

这个函数会从已连接的连接队列头部返回一个已经完成的连接，如果没有已经完成的连接，这个函数就会阻塞，直到取出队列中已完成的用户连接为止

### 6.5 connect函数

建立一个连接至addr所指定的套接字

原型：

#include <sys/types.h> /\* See NOTES \*/

#include <sys/socket.h>

int connect(int sockfd, const struct sockaddr \*addr,

socklen\_t addrlen);

参数：

sockfd 未连接套接字

addr 要连接的套接字地址

addrlen 第二个参数addr的长度

返回值：成功返回0，失败返回-1

connect()函数会为客户端主动连接服务器，通过三次握手建立连接，而这个过程由内核完成，最后把连接的结果返回给函数

实例:

windows客户端

#pragma comment(lib, "ws2\_32.lib")

#pragma warning(disable:4996)

#include<stdio.h>

#include<WinSock2.h>

#include<stdlib.h>

#include<memory.h>

#include<string.h>

int main()

{

//初始化编程环境

WORD myVersionRequest;

WSADATA wsaData;

myVersionRequest = MAKEWORD(1, 1);

int err;

err = WSAStartup(myVersionRequest, &wsaData);//打开套接字

if (!err)

{

printf("已打开套接字\n");

}

else

{

printf("嵌套字未打开!");

return 0;

}

int sock = socket(AF\_INET, SOCK\_STREAM,0);//创建TCP套接字

if (!sock)

{

printf("创建套接字出错\n");

return -1;

}

struct sockaddr\_in servaddr;

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

servaddr.sin\_addr.S\_un.S\_addr = inet\_addr("192.168.174.129");

servaddr.sin\_family = AF\_INET;

servaddr.sin\_port = htons(5100);

int ret = connect(sock, (struct sockaddr\*)&servaddr, sizeof(servaddr));

char sendbuf[1024] = { 0 };

char recvbuf[1024] = { 0 };

//获取stdin输入流，并将输入的内容写入sendbuf

while (fgets(sendbuf, sizeof(sendbuf), stdin) != NULL)

{

//向服务器发送数据

send(sock, sendbuf, sizeof(sendbuf), 0);

//获取从服务器发回的数据

recv(sock, recvbuf, sizeof(recvbuf), 0);

//输出获取的数据

fputs(recvbuf, stdout);

//清空接收和发送的内容

memset(sendbuf, 0, sizeof(sendbuf));

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

}

//关闭套接字

closesocket(sock);

return 0;

}

Linux服务端

#include <unistd.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <stdlib.h>

#include <stdio.h>

#include <errno.h>

#include <string.h>

//错误处理

#define ERR\_EXIT(m) \

do \

{ \

perror(m); \

exit(EXIT\_FAILURE); \

} while(0)

int main(void)

{

int listenfd;//监听套接字

//

if ((listenfd = socket(PF\_INET, SOCK\_STREAM, IPPROTO\_TCP)) < 0)

/\* if ((listenfd = socket(PF\_INET, SOCK\_STREAM, 0)) < 0)\*/

ERR\_EXIT("socket");

//设置服务端的地址信息

struct sockaddr\_in servaddr;

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

servaddr.sin\_family = AF\_INET;

servaddr.sin\_port = htons(5100);

servaddr.sin\_addr.s\_addr = htonl(INADDR\_ANY);//表示任意地址，让它绑定本机的任意地址

//绑定套接字

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

ERR\_EXIT("bind");

//监听套接字

if (listen(listenfd, SOMAXCONN) < 0)

ERR\_EXIT("listen");

struct sockaddr\_in peeraddr;

socklen\_t peerlen = sizeof(peeraddr);

int conn;

//等待对方连接，如果收到客户端的连接请求，就将套接字从客户端

if ((conn = accept(listenfd, (struct sockaddr\*)&peeraddr, &peerlen)) < 0)

ERR\_EXIT("accept");

char recvbuf[1024];

while (1)

{

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

//接收信息

int ret = read(conn, recvbuf, sizeof(recvbuf));

//输出接收的信息

fputs(recvbuf, stdout);

//向客户端发送信息

write(conn, recvbuf, ret);

}

//关闭套接字

close(conn);

close(listenfd);

return 0;

}

# 多客户连接

## 1.SO\_REUSEADDR

一般一个端口释放后会等待两分钟之后才能被使用，SO\_REUSERADDR是让端口释放后就可以被再次使用。

SO\_REUSEADDR用于对TCP套接字处于TIME\_WAIT状态下的socket，才可以重复绑定重复使用。server程序总是应该在调用bind()之前设置SO\_REUSEADDR套接字。TCP先调用close()的一方会进入TIME\_WAIT状态

SO\_REUSEADDR 提供的功能：

1.允许启动一个监听服务器并捆绑其众所周知的端口，即使以前建立的将此端口作为他们的本地端口的连接存在，如不设置此项，bind将出错

2. SO\_REUSEADDR允许在同一端口上启动同一服务器的多个实例，只要每个实例捆绑一个不同的本地IP地址即可。对于TCP，我们根本不可能启动捆绑相同IP地址和相同端口号的多个服务器。

3. SO\_REUSEADDR允许单个进程捆绑同一端口到多个套接口上，只要每个捆绑指定不同的本地IP地址即可。这一般不用于TCP服务器。

4. SO\_REUSEADDR允许完全重复的捆绑：当一个IP地址和端口绑定到某个套接口上时，还允许此IP地址和端口捆绑到另一个套接口上。一般来说，这个特性仅在支持多播的系统上才有，而且只对UDP套接口而言（TCP不支持多播）。

一般使用一个进程来处理一个进程。

setsockopt函数：

函数原型：

#include <sys/types.h> /\* See NOTES \*/

#include <sys/socket.h>

int getsockopt(int sockfd, int level, int optname,

void \*optval, socklen\_t \*optlen);

int setsockopt(int sockfd, int level, int optname,

const void \*optval, socklen\_t optlen);

参数说明：

sock :将要被设置或获取选项的套接字

level:选项所在的协议层

optname:需要访问的选项名

optval:对于getsockopt()，指向返回选项值的缓冲，对于setsockopt()，指向包含新选项值的缓冲

optlen: 对于getsockopt()，作为入口参数时，选项值的最大长度。作为出口参数时，选项值的实际长度。对于setsockopt()，现选项的长度。

成功执行时，返回0。失败返回-1，errno被设为以下的某个值    
EBADF：sock不是有效的文件描述词  
EFAULT：optval指向的内存并非有效的进程空间  
EINVAL：在调用setsockopt()时，optlen无效  
ENOPROTOOPT：指定的协议层不能识别选项  
ENOTSOCK：sock描述的不是套接字

level指定控制套接字的层次.可以取三种值:  
1)SOL\_SOCKET:通用套接字选项.  
2)IPPROTO\_IP:IP选项.  
3)IPPROTO\_TCP:TCP选项.　  
optname指定控制的方式(选项的名称),我们下面详细解释　  
  
  
optval获得或者是设置套接字选项.根据选项名称的数据类型进行转换

服务端(处理多个连接)：

#include <unistd.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <stdlib.h>

#include <stdio.h>

#include <errno.h>

#include <string.h>

#define ERR\_EXIT(m) \

do \

{ \

perror(m); \

exit(EXIT\_FAILURE); \

} while(0)

void do\_service(int conn)//服务端对连接所做的处理

{

char recvbuf[1024];

while (1)

{

memset(recvbuf, 0, sizeof(recvbuf));//

int ret = read(conn, recvbuf, sizeof(recvbuf));

if (ret == 0)

{

printf("client close\n");

break;

}

else if (ret == -1)

ERR\_EXIT("read");

fputs(recvbuf, stdout);//将客户端发送来的内容输出

write(conn, recvbuf, ret);//发送接收的内容到服务端

}

}

int main(void)

{

int listenfd;

if ((listenfd = socket(PF\_INET, SOCK\_STREAM, IPPROTO\_TCP)) < 0)

/\* if ((listenfd = socket(PF\_INET, SOCK\_STREAM, 0)) < 0)\*/

ERR\_EXIT("socket");

struct sockaddr\_in servaddr;

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

servaddr.sin\_family = AF\_INET;

servaddr.sin\_port = htons(5188);

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

/\*servaddr.sin\_addr.s\_addr = inet\_addr("127.0.0.1");\*/

/\*inet\_aton("127.0.0.1", &servaddr.sin\_addr);\*/

int on = 1;//开启地址重复利用，可以跳过TIME\_WAIT状态

if (setsockopt(listenfd, SOL\_SOCKET, SO\_REUSEADDR, &on, sizeof(on)) < 0)//设置套接字选项

ERR\_EXIT("setsockopt");

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

ERR\_EXIT("bind");

if (listen(listenfd, SOMAXCONN) < 0)

ERR\_EXIT("listen");

struct sockaddr\_in peeraddr;

socklen\_t peerlen = sizeof(peeraddr);

int conn;

pid\_t pid;

while (1)

{

//从已连接的队列中返回一个连接

if ((conn = accept(listenfd, (struct sockaddr\*)&peeraddr, &peerlen)) < 0)

ERR\_EXIT("accept");

//输出连接的信息

printf("ip=%s port=%d\n", inet\_ntoa(peeraddr.sin\_addr), ntohs(peeraddr.sin\_port));

pid = fork();//获取一个连接就创建一个进程，然后让子进程来进程通信的处理，这样就可以实现多个客户端的连接

if (pid == -1)

ERR\_EXIT("fork");

if (pid == 0)

{

close(listenfd);//子进程不需要处理监听

do\_service(conn);

exit(EXIT\_SUCCESS);

}

else

close(conn);

}

return 0;

}

点对点的聊天程序

#include <unistd.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <signal.h>

#include <stdlib.h>

#include <stdio.h>

#include <errno.h>

#include <string.h>

#define ERR\_EXIT(m) \

do \

{ \

perror(m); \

exit(EXIT\_FAILURE); \

} while(0)

void handler(int sig)

{

printf("recv a sig=%d\n", sig);

exit(EXIT\_SUCCESS);

}

int main(void)

{

int listenfd;

if ((listenfd = socket(PF\_INET, SOCK\_STREAM, IPPROTO\_TCP)) < 0)//监听套接字

/\* if ((listenfd = socket(PF\_INET, SOCK\_STREAM, 0)) < 0)\*/

ERR\_EXIT("socket");

struct sockaddr\_in servaddr;

memset(&servaddr, 0, sizeof(servaddr));//请空服务端的地址

//设置服务端地址

servaddr.sin\_family = AF\_INET;

servaddr.sin\_port = htons(5188);

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

/\*servaddr.sin\_addr.s\_addr = inet\_addr("127.0.0.1");\*/

/\*inet\_aton("127.0.0.1", &servaddr.sin\_addr);\*/

int on = 1;

//设置监听套接字

if (setsockopt(listenfd, SOL\_SOCKET, SO\_REUSEADDR, &on, sizeof(on)) < 0)

ERR\_EXIT("setsockopt");

//绑定监听套接字

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

ERR\_EXIT("bind");

if (listen(listenfd, SOMAXCONN) < 0)

ERR\_EXIT("listen");

struct sockaddr\_in peeraddr;

socklen\_t peerlen = sizeof(peeraddr);

int conn;

if ((conn = accept(listenfd, (struct sockaddr\*)&peeraddr, &peerlen)) < 0)

ERR\_EXIT("accept");

printf("ip=%s port=%d\n", inet\_ntoa(peeraddr.sin\_addr), ntohs(peeraddr.sin\_port));

pid\_t pid;

pid = fork();//创建子进程

if (pid == -1)

ERR\_EXIT("fork");

if (pid == 0)//在子进程中处理发送

{

signal(SIGUSR1, handler);

char sendbuf[1024] = {0};

while (fgets(sendbuf, sizeof(sendbuf), stdin) != NULL)

{

write(conn, sendbuf, strlen(sendbuf));

memset(sendbuf, 0, sizeof(sendbuf));

}

printf("child close\n");

exit(EXIT\_SUCCESS);

}

else

{

char recvbuf[1024];

while (1)

{

memset(recvbuf, 0, sizeof(recvbuf));//在父进程中处理接收

int ret = read(conn, recvbuf, sizeof(recvbuf));

if (ret == -1)

ERR\_EXIT("read");

else if (ret == 0)

{

printf("peer close\n");

break;

}

fputs(recvbuf, stdout);

}

printf("parent close\n");

kill(pid, SIGUSR1);

exit(EXIT\_SUCCESS);

}

return 0;

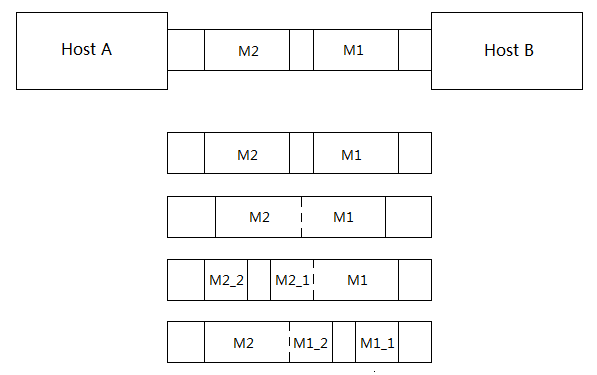
}

# 粘包和流协议

## 1.粘包原因

TCP是基于字节流的传输协议，传输的数据没有边界，而UDP是基于消息的传输协议，有边界

字节流不能保证一次读取操作返回的是一次消息。



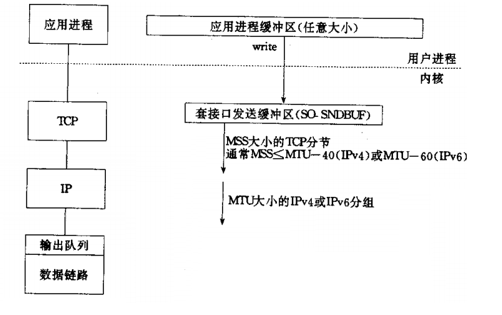
如上图所示，主机A向主机B发送M2,M2两个数据包：

1.两次读取分别返回M2,M1

2.一次读取M2,M1两个数据包

3.一次读取M2的一部分，一次读取M1和M2的另外一部分

这样就导致了粘包



应用发送过程要通过write函数将应用程序的缓冲区复制到套接口缓冲区，而套接口大缓冲区有大小，当应用程序的待发送的数据超过套接口的发送缓冲区的大小就会产生粘包

## 2 解决方案

本质上是要在应用层维护消息与消息的边界

定长包

包尾加\r\n（ftp）

包头加上包体长度

更复杂的应用层协议

## 3.读写定长的数据包

//接收count个字节数

ssize\_t readn(int fd, void \*buf, size\_t count)

{

size\_t nleft = count;//剩余的接收字节数

ssize\_t nread;//接收到的字节数

char \*bufp = (char\*)buf;

while (nleft > 0)//只要剩余的字节数大于0，就应该继续读取

{

if ((nread = read(fd, bufp, nleft)) < 0)//如果读取字节数小于0，说明读取出错

{

if (errno == EINTR)

continue;

return -1;

}

else if (nread == 0)//说明发送端关闭了发送操作

return count - nleft;

bufp += nread;// 对读取的缓冲进行偏移

nleft -= nread;// 还剩余的读取数

}

return count;

}

//发送count个字符

ssize\_t writen(int fd, const void \*buf, size\_t count)

{

size\_t nleft = count;//待发送的字节数

ssize\_t nwritten;//接收的字节数

char \*bufp = (char\*)buf;

while (nleft > 0)

{

if ((nwritten = write(fd, bufp, nleft)) < 0)//说明发送出错

{

if (errno == EINTR)//被其他信号中断

continue;

return -1;

}

else if (nwritten == 0)//什么都没发送过

continue;

bufp += nwritten;//指针偏移

nleft -= nwritten;//已发送的字节数

}

return count;

}

## 4.对回射服务端/客户端的完善

客户端：

#include <unistd.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <stdlib.h>

#include <stdio.h>

#include <errno.h>

#include <string.h>

#define ERR\_EXIT(m) \

do \

{ \

perror(m); \

exit(EXIT\_FAILURE); \

} while(0)

struct packet//定义数据包结构

{

int len;

char buf[1024];

};

//接收count个字节数

ssize\_t readn(int fd, void \*buf, size\_t count)

{

size\_t nleft = count;//剩余的接收字节数

ssize\_t nread;//接收到的字节数

char \*bufp = (char\*)buf;

while (nleft > 0)//只要剩余的字节数大于0，就应该继续读取

{

if ((nread = read(fd, bufp, nleft)) < 0)//如果读取字节数小于0，说明读取出错

{

if (errno == EINTR)

continue;

return -1;

}

else if (nread == 0)//说明发送端关闭了发送操作

return count - nleft;

bufp += nread;// 对读取的缓冲进行偏移

nleft -= nread;// 还剩余的读取数

}

return count;

}

//发送count个字符

ssize\_t writen(int fd, const void \*buf, size\_t count)

{

size\_t nleft = count;//待发送的字节数

ssize\_t nwritten;//接收的字节数

char \*bufp = (char\*)buf;

while (nleft > 0)

{

if ((nwritten = write(fd, bufp, nleft)) < 0)//说明发送出错

{

if (errno == EINTR)//被其他信号中断

continue;

return -1;

}

else if (nwritten == 0)//什么都没发送过

continue;

bufp += nwritten;//指针偏移

nleft -= nwritten;//已发送的字节数

}

return count;

}

int main(void)

{

int sock;

if ((sock = socket(PF\_INET, SOCK\_STREAM, IPPROTO\_TCP)) < 0)

ERR\_EXIT("socket");

struct sockaddr\_in servaddr;

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

servaddr.sin\_family = AF\_INET;

servaddr.sin\_port = htons(5188);

servaddr.sin\_addr.s\_addr = inet\_addr("127.0.0.1");

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

ERR\_EXIT("connect");

struct packet sendbuf;

struct packet recvbuf;

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

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

int n;

while (fgets(sendbuf.buf, sizeof(sendbuf.buf), stdin) != NULL)

{

n = strlen(sendbuf.buf);

sendbuf.len = htonl(n);

writen(sock, &sendbuf, 4+n);

int ret = readn(sock, &recvbuf.len, 4);//获取数据包长度

if (ret == -1)

ERR\_EXIT("read");

else if (ret < 4)

{

printf("client close\n");

break;

}

n = ntohl(recvbuf.len);

ret = readn(sock, recvbuf.buf, n);

if (ret == -1)

ERR\_EXIT("read");

else if (ret < n)

{

printf("client close\n");

break;

}

fputs(recvbuf.buf, stdout);

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

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

}

close(sock);

return 0;

}

客户端

#include <unistd.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <stdlib.h>

#include <stdio.h>

#include <errno.h>

#include <string.h>

#define ERR\_EXIT(m) \

do \

{ \

perror(m); \

exit(EXIT\_FAILURE); \

} while(0)

struct packet//定义数据包结构

{

int len;

char buf[1024];

};

//接收count个字节数

ssize\_t readn(int fd, void \*buf, size\_t count)

{

size\_t nleft = count;//剩余的接收字节数

ssize\_t nread;//接收到的字节数

char \*bufp = (char\*)buf;

while (nleft > 0)//只要剩余的字节数大于0，就应该继续读取

{

if ((nread = read(fd, bufp, nleft)) < 0)//如果读取字节数小于0，说明读取出错

{

if (errno == EINTR)

continue;

return -1;

}

else if (nread == 0)//说明发送端关闭了发送操作

return count - nleft;

bufp += nread;// 对读取的缓冲进行偏移

nleft -= nread;// 还剩余的读取数

}

return count;

}

//发送count个字符

ssize\_t writen(int fd, const void \*buf, size\_t count)

{

size\_t nleft = count;//待发送的字节数

ssize\_t nwritten;//接收的字节数

char \*bufp = (char\*)buf;

while (nleft > 0)

{

if ((nwritten = write(fd, bufp, nleft)) < 0)//说明发送出错

{

if (errno == EINTR)//被其他信号中断

continue;

return -1;

}

else if (nwritten == 0)//什么都没发送过

continue;

bufp += nwritten;//指针偏移

nleft -= nwritten;//已发送的字节数

}

return count;

}

int main(void)

{

int sock;

if ((sock = socket(PF\_INET, SOCK\_STREAM, IPPROTO\_TCP)) < 0)

ERR\_EXIT("socket");

struct sockaddr\_in servaddr;

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

servaddr.sin\_family = AF\_INET;

servaddr.sin\_port = htons(5188);

servaddr.sin\_addr.s\_addr = inet\_addr("127.0.0.1");

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

ERR\_EXIT("connect");

struct packet sendbuf;

struct packet recvbuf;

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

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

int n;

while (fgets(sendbuf.buf, sizeof(sendbuf.buf), stdin) != NULL)

{

n = strlen(sendbuf.buf);

sendbuf.len = htonl(n);

writen(sock, &sendbuf, 4+n);

int ret = readn(sock, &recvbuf.len, 4);

if (ret == -1)

ERR\_EXIT("read");

else if (ret < 4)

{

printf("client close\n");

break;

}

n = ntohl(recvbuf.len);

ret = readn(sock, recvbuf.buf, n);

if (ret == -1)

ERR\_EXIT("read");

else if (ret < n)

{

printf("client close\n");

break;

}

fputs(recvbuf.buf, stdout);

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

}

close(sock);

return 0;

}

# readline的实现

## 1. read,write和recv,send

#include <sys/types.h>

#include <sys/socket.h>

ssize\_t recv(int sockfd, void \*buf, size\_t len, int flags);

主要用来接收远端主机指定的socket传来的数据，并把数据存到由参数buf指向的内存空间，参数len为可接收数据的最大长度

flags一般设置为0,其他数值定义为：

MSG\_OOB 接收以out-of-band送出的数据

MSG\_PEEK 返回来的数据不会在系统内删除，如果再调用recv()会返回相同的数据内容

MSG\_WAITALL 强迫接收到len大小的数据后才返回，除非错误或有信号产生。

MSG\_NOSIGNAL此操作不愿被SIGPIPE信号中断返回值成功则返回收到的字符数，失败返回-1，错误原因存于errno中。

错误代码

EBADF 参数s非合法的socket处理代码

EFAULT 参数中有一指针指向无法存取的内存空间  
ENOTSOCK 参数s为一文件描述词，非socket。  
EINTR 被信号所中断  
EAGAIN 此动作会令进程阻断，但参数s的socket为不可阻断  
ENOBUFS 系统的缓冲内存不足。  
ENOMEM 核心内存不足  
EINVAL 传给系统调用的参数不正确。

|  |
| --- |
|  |
|  | send（经socket传送数据） |
| 相关函数 | sendto，sendmsg，recv，recvfrom，socket |
| 表头文件 | #include<sys/types.h> #include<sys/socket.h> |
| 定义函数 | int send(int s,const void \* msg,int len,unsigned int falgs); |
| 函数说明 | send()用来将数据由指定的socket 传给对方主机。参数s为已建立好连接的socket。参数msg指向欲连线的数据内容，参数len则为数据长度。参数flags一般设0，其他数值定义如下 MSG\_OOB 传送的数据以out-of-band 送出。 MSG\_DONTROUTE 取消路由表查询 MSG\_DONTWAIT 设置为不可阻断运作 MSG\_NOSIGNAL 此动作不愿被SIGPIPE 信号中断。 |
| 返回值 | 成功则返回实际传送出去的字符数，失败返回-1。错误原因存于errno |
| 错误代码 | EBADF 参数s 非合法的socket处理代码。 EFAULT 参数中有一指针指向无法存取的内存空间 ENOTSOCK 参数s为一文件描述词，非socket。 EINTR 被信号所中断。 EAGAIN 此操作会令进程阻断，但参数s的socket为不可阻断。 ENOBUFS 系统的缓冲内存不足 ENOMEM 核心内存不足 EINVAL 传给系统调用的参数不正确。 |

## 2.readline的实现

ssize\_t recv\_peek(int sockfd, void \*buf, size\_t len)

{

while (1)

{

int ret = recv(sockfd, buf, len, MSG\_PEEK);//返回的接收数据不会被删除，下次调用recv还能接收到已经返回的数据

if (ret == -1 && errno == EINTR)//如果被信号中断，重新接收

continue;

return ret;

}

//这个函数主要是判断是否有数据可以接收，可接收的缓冲有多大

}

//按行读取

ssize\_t readline(int sockfd, void \*buf, size\_t maxline)

{

int ret;

int nread;//读取的字节数

char \*bufp = buf;

int nleft = maxline;//可读取的最大字节数

while (1)

{

ret = recv\_peek(sockfd, bufp, nleft);

if (ret < 0)//说明没有数据可以接收

return ret;

else if (ret == 0)//对方关闭了套接口

return ret;

nread = ret;//接收到的字节数

int i;

for (i=0; i<nread; i++)//判断接收到的内容中是否有'\n'

{

if (bufp[i] == '\n')

{

ret = readn(sockfd, bufp, i+1);//将数据从缓冲区移除

if (ret != i+1)

exit(EXIT\_FAILURE);

return ret;

}

}

//如果没有'\n'说明一条消息没有读取结束

if (nread > nleft)//不允许超过剩余的字节数

exit(EXIT\_FAILURE);

nleft -= nread;

ret = readn(sockfd, bufp, nread);

if (ret != nread)

exit(EXIT\_FAILURE);

bufp += nread;//移动指针，下次读取的内容放到末尾

}

return -1;

}

## 3. getsockname,getpeername

#include <sys/socket.h>

int getsockname(int sockfd, struct sockaddr \*addr, socklen\_t \*addrlen);

这个函数用于返回在内核赋予该连接的本地IP地址和本地端口号协议簇等连接信息

#include <sys/socket.h>

int getpeername(int sockfd, struct sockaddr \*addr, socklen\_t \*addrlen);

这个函数用于获取客户端的IP地址和端口号等连接信号

故前面的客户端和服务端可以修改为：

客户端

#include <unistd.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <stdlib.h>

#include <stdio.h>

#include <errno.h>

#include <string.h>

#define ERR\_EXIT(m) \

do \

{ \

perror(m); \

exit(EXIT\_FAILURE); \

} while(0)

ssize\_t readn(int fd, void \*buf, size\_t count)

{

size\_t nleft = count;

ssize\_t nread;

char \*bufp = (char\*)buf;

while (nleft > 0)

{

if ((nread = read(fd, bufp, nleft)) < 0)

{

if (errno == EINTR)

continue;

return -1;

}

else if (nread == 0)

return count - nleft;

bufp += nread;

nleft -= nread;

}

return count;

}

ssize\_t writen(int fd, const void \*buf, size\_t count)

{

size\_t nleft = count;

ssize\_t nwritten;

char \*bufp = (char\*)buf;

while (nleft > 0)

{

if ((nwritten = write(fd, bufp, nleft)) < 0)

{

if (errno == EINTR)

continue;

return -1;

}

else if (nwritten == 0)

continue;

bufp += nwritten;

nleft -= nwritten;

}

return count;

}

ssize\_t recv\_peek(int sockfd, void \*buf, size\_t len)

{

while (1)

{

int ret = recv(sockfd, buf, len, MSG\_PEEK);//返回的接收数据不会被删除，下次调用recv还能接收到已经返回的数据

if (ret == -1 && errno == EINTR)//如果被信号中断，重新接收

continue;

return ret;

}

//这个函数主要是判断是否有数据可以接收，可接收的缓冲有多大

}

//按行读取

ssize\_t readline(int sockfd, void \*buf, size\_t maxline)

{

int ret;

int nread;//读取的字节数

char \*bufp = buf;

int nleft = maxline;//可读取的最大字节数

while (1)

{

ret = recv\_peek(sockfd, bufp, nleft);

if (ret < 0)//说明没有数据可以接收

return ret;

else if (ret == 0)//对方关闭了套接口

return ret;

nread = ret;//接收到的字节数

int i;

for (i=0; i<nread; i++)//判断接收到的内容中是否有'\n'

{

if (bufp[i] == '\n')

{

ret = readn(sockfd, bufp, i+1);//将数据从缓冲区移除

if (ret != i+1)

exit(EXIT\_FAILURE);

return ret;

}

}

//如果没有'\n'说明一条消息没有读取结束

if (nread > nleft)//不允许超过剩余的字节数

exit(EXIT\_FAILURE);

nleft -= nread;

ret = readn(sockfd, bufp, nread);

if (ret != nread)

exit(EXIT\_FAILURE);

bufp += nread;//移动指针，下次读取的内容放到末尾

}

return -1;

}

int main(void)

{

int sock;

if ((sock = socket(PF\_INET, SOCK\_STREAM, IPPROTO\_TCP)) < 0)

ERR\_EXIT("socket");

struct sockaddr\_in servaddr;

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

servaddr.sin\_family = AF\_INET;

servaddr.sin\_port = htons(5188);

servaddr.sin\_addr.s\_addr = inet\_addr("127.0.0.1");

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

ERR\_EXIT("connect");

struct sockaddr\_in localaddr;

socklen\_t addrlen = sizeof(localaddr);

if (getsockname(sock, (struct sockaddr\*)&localaddr, &addrlen) < 0)//获取本地连接信息

ERR\_EXIT("getsockname");

printf("ip=%s port=%d\n", inet\_ntoa(localaddr.sin\_addr), ntohs(localaddr.sin\_port));

char sendbuf[1024] = {0};

char recvbuf[1024] = {0};

while (fgets(sendbuf, sizeof(sendbuf), stdin) != NULL)

{

writen(sock, sendbuf, strlen(sendbuf));

int ret = readline(sock, recvbuf, sizeof(recvbuf));

if (ret == -1)

ERR\_EXIT("readline");

else if (ret == 0)

{

printf("client close\n");

break;

}

fputs(recvbuf, stdout);

memset(sendbuf, 0, sizeof(sendbuf));

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

}

close(sock);

return 0;

}

服务端

#include <unistd.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <stdlib.h>

#include <stdio.h>

#include <errno.h>

#include <string.h>

#define ERR\_EXIT(m) \

do \

{ \

perror(m); \

exit(EXIT\_FAILURE); \

} while(0)

ssize\_t readn(int fd, void \*buf, size\_t count)

{

size\_t nleft = count;

ssize\_t nread;

char \*bufp = (char\*)buf;

while (nleft > 0)

{

if ((nread = read(fd, bufp, nleft)) < 0)

{

if (errno == EINTR)

continue;

return -1;

}

else if (nread == 0)

return count - nleft;

bufp += nread;

nleft -= nread;

}

return count;

}

ssize\_t writen(int fd, const void \*buf, size\_t count)

{

size\_t nleft = count;

ssize\_t nwritten;

char \*bufp = (char\*)buf;

while (nleft > 0)

{

if ((nwritten = write(fd, bufp, nleft)) < 0)

{

if (errno == EINTR)

continue;

return -1;

}

else if (nwritten == 0)

continue;

bufp += nwritten;

nleft -= nwritten;

}

return count;

}

ssize\_t recv\_peek(int sockfd, void \*buf, size\_t len)

{

while (1)

{

int ret = recv(sockfd, buf, len, MSG\_PEEK);

if (ret == -1 && errno == EINTR)

continue;

return ret;

}

}

ssize\_t readline(int sockfd, void \*buf, size\_t maxline)

{

int ret;

int nread;

char \*bufp = buf;

int nleft = maxline;

while (1)

{

ret = recv\_peek(sockfd, bufp, nleft);

if (ret < 0)

return ret;

else if (ret == 0)

return ret;

nread = ret;

int i;

for (i=0; i<nread; i++)

{

if (bufp[i] == '\n')

{

ret = readn(sockfd, bufp, i+1);

if (ret != i+1)

exit(EXIT\_FAILURE);

return ret;

}

}

if (nread > nleft)

exit(EXIT\_FAILURE);

nleft -= nread;

ret = readn(sockfd, bufp, nread);

if (ret != nread)

exit(EXIT\_FAILURE);

bufp += nread;

}

return -1;

}

void do\_service(int conn)

{

char recvbuf[1024];

while (1)

{

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

int ret = readline(conn, recvbuf, 1024);

if (ret == -1)

ERR\_EXIT("readline");

if (ret == 0)

{

printf("client close\n");

break;

}

fputs(recvbuf, stdout);

writen(conn, recvbuf, strlen(recvbuf));

}

}

int main(void)

{

int listenfd;

if ((listenfd = socket(PF\_INET, SOCK\_STREAM, IPPROTO\_TCP)) < 0)

/\* if ((listenfd = socket(PF\_INET, SOCK\_STREAM, 0)) < 0)\*/

ERR\_EXIT("socket");

struct sockaddr\_in servaddr;

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

servaddr.sin\_family = AF\_INET;

servaddr.sin\_port = htons(5188);

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

/\*servaddr.sin\_addr.s\_addr = inet\_addr("127.0.0.1");\*/

/\*inet\_aton("127.0.0.1", &servaddr.sin\_addr);\*/

int on = 1;

if (setsockopt(listenfd, SOL\_SOCKET, SO\_REUSEADDR, &on, sizeof(on)) < 0)

ERR\_EXIT("setsockopt");

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

ERR\_EXIT("bind");

if (listen(listenfd, SOMAXCONN) < 0)

ERR\_EXIT("listen");

struct sockaddr\_in peeraddr;

socklen\_t peerlen = sizeof(peeraddr);

int conn;

pid\_t pid;

while (1)

{

if ((conn = accept(listenfd, (struct sockaddr\*)&peeraddr, &peerlen)) < 0)

ERR\_EXIT("accept");

printf("ip=%s port=%d\n", inet\_ntoa(peeraddr.sin\_addr), ntohs(peeraddr.sin\_port));

pid = fork();

if (pid == -1)

ERR\_EXIT("fork");

if (pid == 0)

{

close(listenfd);

do\_service(conn);

exit(EXIT\_SUCCESS);

}

else

close(conn);

}

return 0;

}

## 4.gethostname ,gethostbyname,gethostbyaddr

#include <unistd.h>

int gethostname(char \*name, size\_t len);

获取主机名

#include <netdb.h>

extern int h\_errno;

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

由主机名找到获取主机上的所有ip地址

#include <sys/socket.h> /\* for AF\_INET \*/

struct hostent \*gethostbyaddr(const void \*addr,socklen\_t len, int type);

#include <unistd.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <netdb.h>

#include <stdlib.h>

#include <stdio.h>

#include <errno.h>

#include <string.h>

#define ERR\_EXIT(m) \

do \

{ \

perror(m); \

exit(EXIT\_FAILURE); \

} while(0)

int getlocalip(char \*ip)

{

char host[100] = {0};

if (gethostname(host, sizeof(host)) < 0)

return -1;

struct hostent \*hp;

if ((hp = gethostbyname(host)) == NULL)

return -1;

strcpy(ip, inet\_ntoa(\*(struct in\_addr\*)hp->h\_addr));

return 0;

}

int main(void)

{

char host[100] = {0};

if (gethostname(host, sizeof(host)) < 0)

ERR\_EXIT("gethostname");

struct hostent \*hp;

if ((hp = gethostbyname(host)) == NULL)

ERR\_EXIT("gethostbyname");

int i = 0;

while (hp->h\_addr\_list[i] != NULL)

{

printf("%s\n", inet\_ntoa(\*(struct in\_addr\*)hp->h\_addr\_list[i]));//转换为点分十进制

i++;

}

char ip[16] = {0};

getlocalip(ip);

printf("localip=%s\n", ip);

return 0;

}

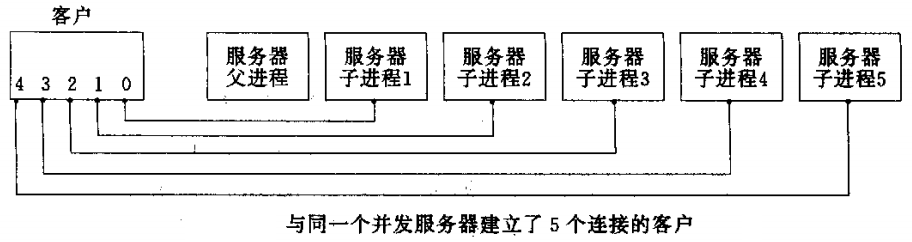
# 并发连接

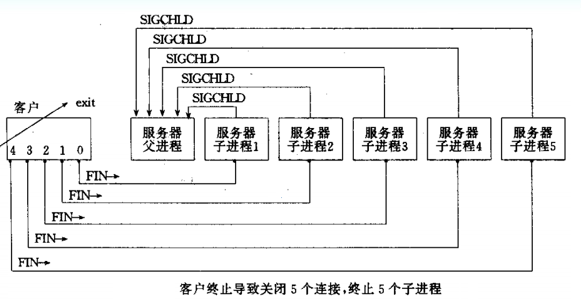
## 1.TCP流

TCP是基于字节流传输的，只维护发送出去多少，确认了多少，没有维护消息与消息之间的边界，因而可能导致粘包问题

粘包问题的解决方法是在应用层维护消息边界

## 2.僵尸进程与SIGCHLD信号





这种时候便会产生僵尸进程

#include <sys/types.h>

#include <sys/wait.h>

pid\_t wait(int \*status);

进程一旦调用了wait，就会立即阻塞自己，由wait自动分析当前进程的某个子进程是否已经退出，如果它找到了这样一个已经变成僵尸的子进程，wait就会收集这个子进程的信息，并把它彻底销毁后返回，如果没有找到这样的一个子进程，wait就会阻塞在这里，直到有一个出现为止

参数status用来保存被收集的进程退出时的一些状态，它是一个指向int类型的指针，如果只想消灭僵尸进程，可以设置为NULL，成功就会返回被收集的子进程的进程ID，如果没有子进程，就会调用失败，返回-1

pid\_t waitpid(pid\_t pid, int \*status, int options);

从本质上来说，调用waitpid也是为了消除僵尸进程只是提供了两个更灵活的两个控制参数

pid：进程ID

>0时，只等待进程ID为pid的子进程，不管其他已经有多少子进程运行结束，只要指定的子进程还没结束，waitpid就会一直等下去

=-1时，等待任何一个子进程退出，没有任何限制，此时waitpid和wait的作用一样

=0时，等待同一个进程组中的任何子进程，如果子进程已经加入别的进程组，waitpid不会对它做任何理踩

<-1时，等待一个指定进程组中的任何子进程，这个进程组的id等于pid的绝对值。

options:

options提供了一些额外的选项来控制waitpid，目前在Linux中只支持WNOHANG和WUNTRACED两个选项，这是两个常数，可以用"|"运算符把它们连接起来使用, ret=waitpid(-1,NULL,WNOHANG | WUNTRACED);

如果使用了WNOHANG参数，那么即使子进程没有退出，它也会立即返回，不会无限等待下去。

客户端：

#include <unistd.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <stdlib.h>

#include <stdio.h>

#include <errno.h>

#include <string.h>

#define ERR\_EXIT(m) \

do \

{ \

perror(m); \

exit(EXIT\_FAILURE); \

} while(0)

ssize\_t readn(int fd, void \*buf, size\_t count)

{

size\_t nleft = count;

ssize\_t nread;

char \*bufp = (char\*)buf;

while (nleft > 0)

{

if ((nread = read(fd, bufp, nleft)) < 0)

{

if (errno == EINTR)

continue;

return -1;

}

else if (nread == 0)

return count - nleft;

bufp += nread;

nleft -= nread;

}

return count;

}

ssize\_t writen(int fd, const void \*buf, size\_t count)

{

size\_t nleft = count;

ssize\_t nwritten;

char \*bufp = (char\*)buf;

while (nleft > 0)

{

if ((nwritten = write(fd, bufp, nleft)) < 0)

{

if (errno == EINTR)

continue;

return -1;

}

else if (nwritten == 0)

continue;

bufp += nwritten;

nleft -= nwritten;

}

return count;

}

ssize\_t recv\_peek(int sockfd, void \*buf, size\_t len)

{

while (1)

{

int ret = recv(sockfd, buf, len, MSG\_PEEK);

if (ret == -1 && errno == EINTR)

continue;

return ret;

}

}

ssize\_t readline(int sockfd, void \*buf, size\_t maxline)

{

int ret;

int nread;

char \*bufp = buf;

int nleft = maxline;

while (1)

{

ret = recv\_peek(sockfd, bufp, nleft);

if (ret < 0)

return ret;

else if (ret == 0)

return ret;

nread = ret;

int i;

for (i=0; i<nread; i++)

{

if (bufp[i] == '\n')

{

ret = readn(sockfd, bufp, i+1);

if (ret != i+1)

exit(EXIT\_FAILURE);

return ret;

}

}

if (nread > nleft)

exit(EXIT\_FAILURE);

nleft -= nread;

ret = readn(sockfd, bufp, nread);

if (ret != nread)

exit(EXIT\_FAILURE);

bufp += nread;

}

return -1;

}

//客户端的执行程序

void echo\_cli(int sock)

{

char sendbuf[1024] = {0};

char recvbuf[1024] = {0};

while (fgets(sendbuf, sizeof(sendbuf), stdin) != NULL)//从输入流获取要发送的内容

{

writen(sock, sendbuf, strlen(sendbuf));//发送

int ret = readline(sock, recvbuf, sizeof(recvbuf));//读取一行数据

if (ret == -1)

ERR\_EXIT("readline");

else if (ret == 0)//说明客户端已经关闭

{

printf("client close\n");

break;

}

fputs(recvbuf, stdout);//输出接受的数据

memset(sendbuf, 0, sizeof(sendbuf));

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

}

close(sock);

}

int main(void)

{

int sock;

if ((sock = socket(PF\_INET, SOCK\_STREAM, IPPROTO\_TCP)) < 0)

ERR\_EXIT("socket");

//设置服务端的地址信息

struct sockaddr\_in servaddr;

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

servaddr.sin\_family = AF\_INET;

servaddr.sin\_port = htons(5188);

servaddr.sin\_addr.s\_addr = inet\_addr("127.0.0.1");

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

ERR\_EXIT("connect");

struct sockaddr\_in localaddr;

socklen\_t addrlen = sizeof(localaddr);

if (getsockname(sock, (struct sockaddr\*)&localaddr, &addrlen) < 0)//获取本地主机

ERR\_EXIT("getsockname");

printf("ip=%s port=%d\n", inet\_ntoa(localaddr.sin\_addr), ntohs(localaddr.sin\_port));

//处理客户端的信息

echo\_cli(sock);

return 0;

}

服务端：

#include <unistd.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <signal.h>

#include <sys/wait.h>

#include <stdlib.h>

#include <stdio.h>

#include <errno.h>

#include <string.h>

#define ERR\_EXIT(m) \

do \

{ \

perror(m); \

exit(EXIT\_FAILURE); \

} while(0)

ssize\_t readn(int fd, void \*buf, size\_t count)

{

size\_t nleft = count;

ssize\_t nread;

char \*bufp = (char\*)buf;

while (nleft > 0)

{

if ((nread = read(fd, bufp, nleft)) < 0)

{

if (errno == EINTR)

continue;

return -1;

}

else if (nread == 0)

return count - nleft;

bufp += nread;

nleft -= nread;

}

return count;

}

ssize\_t writen(int fd, const void \*buf, size\_t count)

{

size\_t nleft = count;

ssize\_t nwritten;

char \*bufp = (char\*)buf;

while (nleft > 0)

{

if ((nwritten = write(fd, bufp, nleft)) < 0)

{

if (errno == EINTR)

continue;

return -1;

}

else if (nwritten == 0)

continue;

bufp += nwritten;

nleft -= nwritten;

}

return count;

}

ssize\_t recv\_peek(int sockfd, void \*buf, size\_t len)

{

while (1)

{

int ret = recv(sockfd, buf, len, MSG\_PEEK);

if (ret == -1 && errno == EINTR)

continue;

return ret;

}

}

ssize\_t readline(int sockfd, void \*buf, size\_t maxline)

{

int ret;

int nread;

char \*bufp = buf;

int nleft = maxline;

while (1)

{

ret = recv\_peek(sockfd, bufp, nleft);

if (ret < 0)

return ret;

else if (ret == 0)

return ret;

nread = ret;

int i;

for (i=0; i<nread; i++)

{

if (bufp[i] == '\n')

{

ret = readn(sockfd, bufp, i+1);

if (ret != i+1)

exit(EXIT\_FAILURE);

return ret;

}

}

if (nread > nleft)

exit(EXIT\_FAILURE);

nleft -= nread;

ret = readn(sockfd, bufp, nread);

if (ret != nread)

exit(EXIT\_FAILURE);

bufp += nread;

}

return -1;

}

void echo\_srv(int conn)

{

char recvbuf[1024];

while (1)

{

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

int ret = readline(conn, recvbuf, 1024);

if (ret == -1)

ERR\_EXIT("readline");

if (ret == 0)

{

printf("client close\n");

break;

}

fputs(recvbuf, stdout);

writen(conn, recvbuf, strlen(recvbuf));

}

}

void handle\_sigchld(int sig)

{

/\* wait(NULL);\*/

while (waitpid(-1, NULL, WNOHANG) > 0)

;

}

int main(void)

{

/\* signal(SIGCHLD, SIG\_IGN);\*/

signal(SIGCHLD, handle\_sigchld);//为SIGCHLD注册处理消息

int listenfd;

if ((listenfd = socket(PF\_INET, SOCK\_STREAM, IPPROTO\_TCP)) < 0)

/\* if ((listenfd = socket(PF\_INET, SOCK\_STREAM, 0)) < 0)\*/

ERR\_EXIT("socket");

struct sockaddr\_in servaddr;

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

servaddr.sin\_family = AF\_INET;

servaddr.sin\_port = htons(5188);

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

/\*servaddr.sin\_addr.s\_addr = inet\_addr("127.0.0.1");\*/

/\*inet\_aton("127.0.0.1", &servaddr.sin\_addr);\*/

//设置地址可复用

int on = 1;

if (setsockopt(listenfd, SOL\_SOCKET, SO\_REUSEADDR, &on, sizeof(on)) < 0)

ERR\_EXIT("setsockopt");

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

ERR\_EXIT("bind");

if (listen(listenfd, SOMAXCONN) < 0)

ERR\_EXIT("listen");

struct sockaddr\_in peeraddr;

socklen\_t peerlen = sizeof(peeraddr);

int conn;

pid\_t pid;

while (1)

{

if ((conn = accept(listenfd, (struct sockaddr\*)&peeraddr, &peerlen)) < 0)

ERR\_EXIT("accept");

printf("ip=%s port=%d\n", inet\_ntoa(peeraddr.sin\_addr), ntohs(peeraddr.sin\_port));

pid = fork();

if (pid == -1)

ERR\_EXIT("fork");

if (pid == 0)

{

close(listenfd);

echo\_srv(conn);

exit(EXIT\_SUCCESS);

}

else

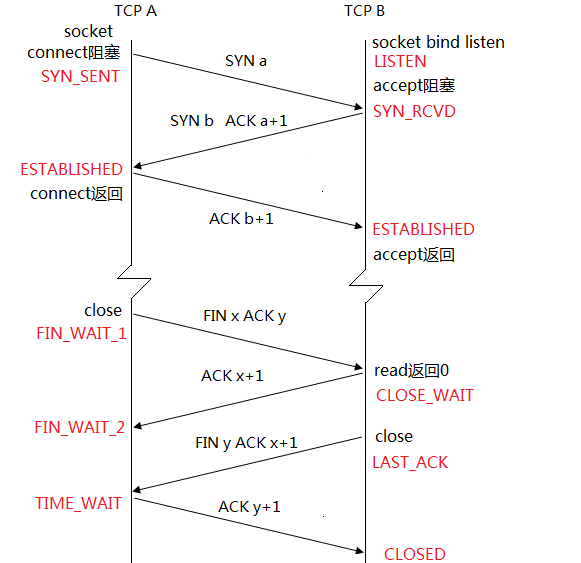
close(conn);

}

return 0;

}

# TCP状态



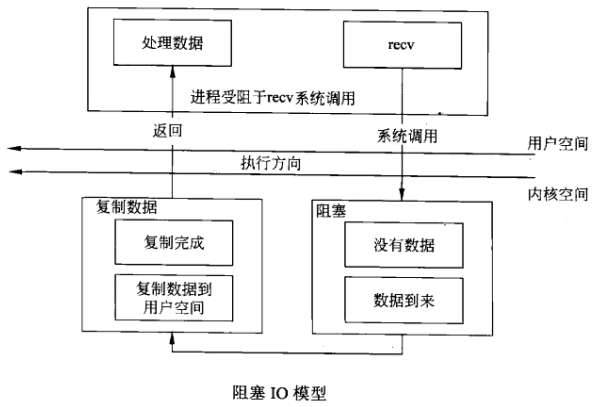
往一个已经接收FIN的套接中写是允许的，接收到的FIN仅仅代表对方不再发生数据。

在收到RST段之后，如果再调用write就会产生SIGPIPE信号，对于这个信号，可以选择忽略

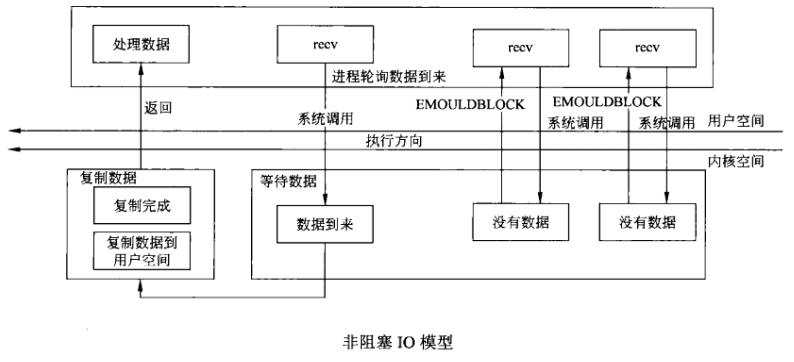
signal(SIGPIPE,SIG\_IGN)

# I/O模型

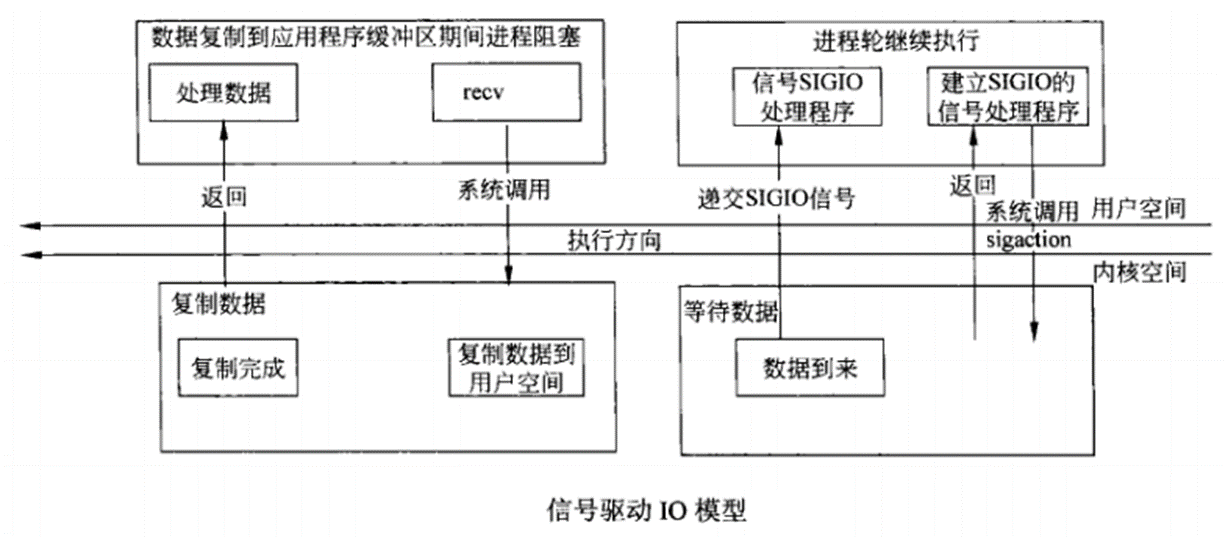
## 1.阻塞I/O



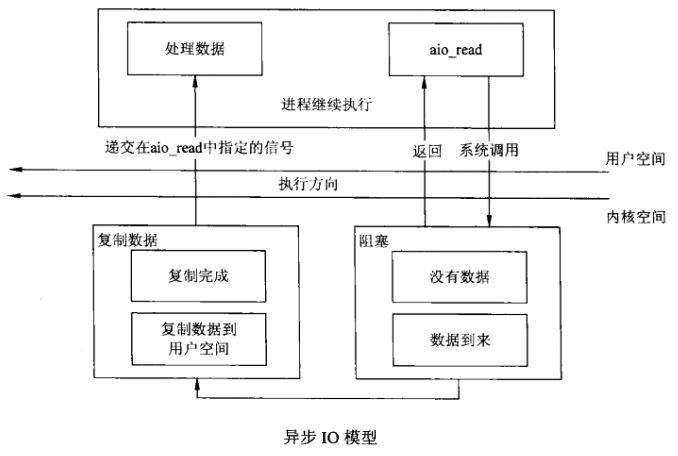
## 2.非阻塞I/O



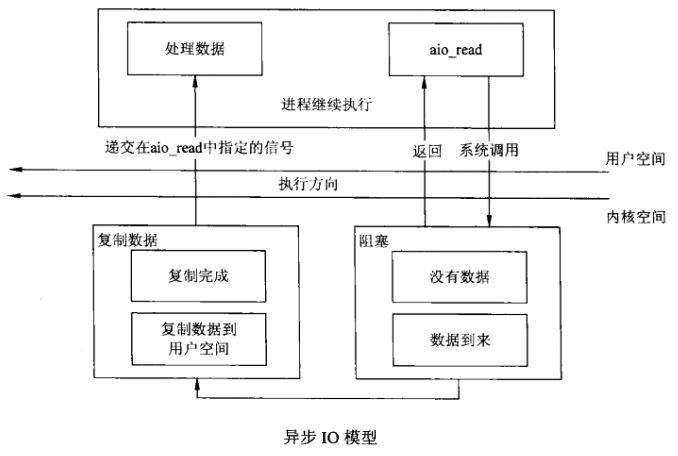
## 3.信号驱动I/O



## 4.异步I/O



## 5.I/O复用



## 6.select

#include <sys/time.h>

#include <sys/types.h>

#include <unistd.h>

int select(int nfds, fd\_set \*readfds, fd\_set \*writefds,

fd\_set \*exceptfds, struct timeval \*timeout);

select函数允许进程指示内核等待多个事件中的任何一个发生，并仅在有一个或多个事件发生或经历一段指定的时间唤醒它

nfds：select监视的套接字数，视进程中打开的文件数而定，一般设置为要监视的总数的最大套接字号加1

readfds:要监视的可读套接字集合

writefds:要监视的可写套接字集合

execptfds:要监视的异常套接字集合

timeout指定一个时间它告诉指定的描述字集合中任何一个就绪可花多少时间：

struct timeval {

long tv\_sec; /\* seconds \*/

long tv\_usec; /\* microseconds \*/

};

如果这个参数为空指针，将永远等待下去，仅在有一个描述字准备号I/O才返回

等待一段固定时间，在有一个描述字准备好就返回，但是不超过改参数指定的时间

不等待，检查描述字后立即返回，这称为轮询，为此，该参数必须指向一个timeval结构，而且其中的定时器限值必须为0

void FD\_CLR(int fd, fd\_set \*set);//清除fd与套接字集合的关系

int FD\_ISSET(int fd, fd\_set \*set);//检查与set联系的fd是否可读写

void FD\_SET(int fd, fd\_set \*set);//建立fd与set的联系

void FD\_ZERO(fd\_set \*set);//清空set与所有套接字的联系

客户端

#include <unistd.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <signal.h>

#include <stdlib.h>

#include <stdio.h>

#include <errno.h>

#include <string.h>

#define ERR\_EXIT(m) \

do \

{ \

perror(m); \

exit(EXIT\_FAILURE); \

} while(0)

ssize\_t readn(int fd, void \*buf, size\_t count)

{

size\_t nleft = count;

ssize\_t nread;

char \*bufp = (char\*)buf;

while (nleft > 0)

{

if ((nread = read(fd, bufp, nleft)) < 0)

{

if (errno == EINTR)

continue;

return -1;

}

else if (nread == 0)

return count - nleft;

bufp += nread;

nleft -= nread;

}

return count;

}

ssize\_t writen(int fd, const void \*buf, size\_t count)

{

size\_t nleft = count;

ssize\_t nwritten;

char \*bufp = (char\*)buf;

while (nleft > 0)

{

if ((nwritten = write(fd, bufp, nleft)) < 0)

{

if (errno == EINTR)

continue;

return -1;

}

else if (nwritten == 0)

continue;

bufp += nwritten;

nleft -= nwritten;

}

return count;

}

ssize\_t recv\_peek(int sockfd, void \*buf, size\_t len)

{

while (1)

{

int ret = recv(sockfd, buf, len, MSG\_PEEK);

if (ret == -1 && errno == EINTR)

continue;

return ret;

}

}

ssize\_t readline(int sockfd, void \*buf, size\_t maxline)

{

int ret;

int nread;

char \*bufp = buf;

int nleft = maxline;

while (1)

{

ret = recv\_peek(sockfd, bufp, nleft);

if (ret < 0)

return ret;

else if (ret == 0)

return ret;

nread = ret;

int i;

for (i=0; i<nread; i++)

{

if (bufp[i] == '\n')

{

ret = readn(sockfd, bufp, i+1);

if (ret != i+1)

exit(EXIT\_FAILURE);

return ret;

}

}

if (nread > nleft)

exit(EXIT\_FAILURE);

nleft -= nread;

ret = readn(sockfd, bufp, nread);

if (ret != nread)

exit(EXIT\_FAILURE);

bufp += nread;

}

return -1;

}

void echo\_cli(int sock)

{

/\*

char sendbuf[1024] = {0};

char recvbuf[1024] = {0};

while (fgets(sendbuf, sizeof(sendbuf), stdin) != NULL)

{

writen(sock, sendbuf, strlen(sendbuf));

int ret = readline(sock, recvbuf, sizeof(recvbuf));

if (ret == -1)

ERR\_EXIT("readline");

else if (ret == 0)

{

printf("client close\n");

break;

}

fputs(recvbuf, stdout);

memset(sendbuf, 0, sizeof(sendbuf));

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

}

close(sock);

\*/

fd\_set rset;

FD\_ZERO(&rset);

int nready;

int maxfd;

int fd\_stdin = fileno(stdin);

if (fd\_stdin > sock)

maxfd = fd\_stdin;

else

maxfd = sock;

char sendbuf[1024] = {0};

char recvbuf[1024] = {0};

while (1)

{

FD\_SET(fd\_stdin, &rset);

FD\_SET(sock, &rset);

nready = select(maxfd+1, &rset, NULL, NULL, NULL);

if (nready == -1)

ERR\_EXIT("select");

if (nready == 0)

continue;

if (FD\_ISSET(sock, &rset))

{

int ret = readline(sock, recvbuf, sizeof(recvbuf));

if (ret == -1)

ERR\_EXIT("readline");

else if (ret == 0)

{

printf("server close\n");

break;

}

fputs(recvbuf, stdout);

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

}

if (FD\_ISSET(fd\_stdin, &rset))

{

if (fgets(sendbuf, sizeof(sendbuf), stdin) == NULL)

break;

writen(sock, sendbuf, strlen(sendbuf));

memset(sendbuf, 0, sizeof(sendbuf));

}

}

close(sock);

}

void handle\_sigpipe(int sig)

{

printf("recv a sig=%d\n", sig);

}

int main(void)

{

/\*

signal(SIGPIPE, handle\_sigpipe);

\*/

signal(SIGPIPE, SIG\_IGN);

int sock;

if ((sock = socket(PF\_INET, SOCK\_STREAM, IPPROTO\_TCP)) < 0)

ERR\_EXIT("socket");

struct sockaddr\_in servaddr;

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

servaddr.sin\_family = AF\_INET;

servaddr.sin\_port = htons(5188);

servaddr.sin\_addr.s\_addr = inet\_addr("127.0.0.1");

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

ERR\_EXIT("connect");

struct sockaddr\_in localaddr;

socklen\_t addrlen = sizeof(localaddr);

if (getsockname(sock, (struct sockaddr\*)&localaddr, &addrlen) < 0)

ERR\_EXIT("getsockname");

printf("ip=%s port=%d\n", inet\_ntoa(localaddr.sin\_addr), ntohs(localaddr.sin\_port));

echo\_cli(sock);

return 0;

}

服务端

#include <unistd.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <signal.h>

#include <sys/wait.h>

#include <stdlib.h>

#include <stdio.h>

#include <errno.h>

#include <string.h>

#define ERR\_EXIT(m) \

do \

{ \

perror(m); \

exit(EXIT\_FAILURE); \

} while(0)

ssize\_t readn(int fd, void \*buf, size\_t count)

{

size\_t nleft = count;

ssize\_t nread;

char \*bufp = (char\*)buf;

while (nleft > 0)

{

if ((nread = read(fd, bufp, nleft)) < 0)

{

if (errno == EINTR)

continue;

return -1;

}

else if (nread == 0)

return count - nleft;

bufp += nread;

nleft -= nread;

}

return count;

}

ssize\_t writen(int fd, const void \*buf, size\_t count)

{

size\_t nleft = count;

ssize\_t nwritten;

char \*bufp = (char\*)buf;

while (nleft > 0)

{

if ((nwritten = write(fd, bufp, nleft)) < 0)

{

if (errno == EINTR)

continue;

return -1;

}

else if (nwritten == 0)

continue;

bufp += nwritten;

nleft -= nwritten;

}

return count;

}

ssize\_t recv\_peek(int sockfd, void \*buf, size\_t len)

{

while (1)

{

int ret = recv(sockfd, buf, len, MSG\_PEEK);

if (ret == -1 && errno == EINTR)

continue;

return ret;

}

}

ssize\_t readline(int sockfd, void \*buf, size\_t maxline)

{

int ret;

int nread;

char \*bufp = buf;

int nleft = maxline;

while (1)

{

ret = recv\_peek(sockfd, bufp, nleft);

if (ret < 0)

return ret;

else if (ret == 0)

return ret;

nread = ret;

int i;

for (i=0; i<nread; i++)

{

if (bufp[i] == '\n')

{

ret = readn(sockfd, bufp, i+1);

if (ret != i+1)

exit(EXIT\_FAILURE);

return ret;

}

}

if (nread > nleft)

exit(EXIT\_FAILURE);

nleft -= nread;

ret = readn(sockfd, bufp, nread);

if (ret != nread)

exit(EXIT\_FAILURE);

bufp += nread;

}

return -1;

}

void echo\_srv(int conn)

{

char recvbuf[1024];

while (1)

{

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

int ret = readline(conn, recvbuf, 1024);

if (ret == -1)

ERR\_EXIT("readline");

if (ret == 0)

{

printf("client close\n");

break;

}

fputs(recvbuf, stdout);

writen(conn, recvbuf, strlen(recvbuf));

}

}

void handle\_sigchld(int sig)

{

/\* wait(NULL);\*/

while (waitpid(-1, NULL, WNOHANG) > 0)

;

}

int main(void)

{

/\* signal(SIGCHLD, SIG\_IGN);\*/

signal(SIGCHLD, handle\_sigchld);

int listenfd;

if ((listenfd = socket(PF\_INET, SOCK\_STREAM, IPPROTO\_TCP)) < 0)

/\* if ((listenfd = socket(PF\_INET, SOCK\_STREAM, 0)) < 0)\*/

ERR\_EXIT("socket");

struct sockaddr\_in servaddr;

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

servaddr.sin\_family = AF\_INET;

servaddr.sin\_port = htons(5188);

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

/\*servaddr.sin\_addr.s\_addr = inet\_addr("127.0.0.1");\*/

/\*inet\_aton("127.0.0.1", &servaddr.sin\_addr);\*/

int on = 1;

if (setsockopt(listenfd, SOL\_SOCKET, SO\_REUSEADDR, &on, sizeof(on)) < 0)

ERR\_EXIT("setsockopt");

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

ERR\_EXIT("bind");

if (listen(listenfd, SOMAXCONN) < 0)

ERR\_EXIT("listen");

struct sockaddr\_in peeraddr;

socklen\_t peerlen = sizeof(peeraddr);

int conn;

pid\_t pid;

while (1)

{

if ((conn = accept(listenfd, (struct sockaddr\*)&peeraddr, &peerlen)) < 0)

ERR\_EXIT("accept");

printf("ip=%s port=%d\n", inet\_ntoa(peeraddr.sin\_addr), ntohs(peeraddr.sin\_port));

pid = fork();

if (pid == -1)

ERR\_EXIT("fork");

if (pid == 0)

{

close(listenfd);

echo\_srv(conn);

exit(EXIT\_SUCCESS);

}

else

close(conn);

}

return 0;

}

## 7.用select改进服务端程序

### 7.1读，写，异常事件发生条件

可读: 套接口缓冲区有数据可读

连接的读一半关闭，即接收到FIN段，读操作将返回0

如果是监听套接口，已完成连接队列不为空时

套接口上发生了一个错误待处理，错误可以通过getsockopt指定

SO\_ERROR选项来获取

可写: 套接口发送缓冲区有空间容纳数据

连接的写一半关闭，即收到RST段之后，再次调用write操作

套接口发生了一个错误待处理，错误可以通过getsockopt指定

SO\_ERROR选项来获取

异常:

套接口存在带外数据

用select改进服务器

#include <unistd.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <signal.h>

#include <sys/wait.h>

#include <stdlib.h>

#include <stdio.h>

#include <errno.h>

#include <string.h>

#define ERR\_EXIT(m) \

do \

{ \

perror(m); \

exit(EXIT\_FAILURE); \

} while(0)

ssize\_t readn(int fd, void \*buf, size\_t count)

{

size\_t nleft = count;

ssize\_t nread;

char \*bufp = (char\*)buf;

while (nleft > 0)

{

if ((nread = read(fd, bufp, nleft)) < 0)

{

if (errno == EINTR)

continue;

return -1;

}

else if (nread == 0)

return count - nleft;

bufp += nread;

nleft -= nread;

}

return count;

}

ssize\_t writen(int fd, const void \*buf, size\_t count)

{

size\_t nleft = count;

ssize\_t nwritten;

char \*bufp = (char\*)buf;

while (nleft > 0)

{

if ((nwritten = write(fd, bufp, nleft)) < 0)

{

if (errno == EINTR)

continue;

return -1;

}

else if (nwritten == 0)

continue;

bufp += nwritten;

nleft -= nwritten;

}

return count;

}

ssize\_t recv\_peek(int sockfd, void \*buf, size\_t len)

{

while (1)

{

int ret = recv(sockfd, buf, len, MSG\_PEEK);

if (ret == -1 && errno == EINTR)

continue;

return ret;

}

}

ssize\_t readline(int sockfd, void \*buf, size\_t maxline)

{

int ret;

int nread;

char \*bufp = buf;

int nleft = maxline;

while (1)

{

ret = recv\_peek(sockfd, bufp, nleft);

if (ret < 0)

return ret;

else if (ret == 0)

return ret;

nread = ret;

int i;

for (i=0; i<nread; i++)

{

if (bufp[i] == '\n')

{

ret = readn(sockfd, bufp, i+1);

if (ret != i+1)

exit(EXIT\_FAILURE);

return ret;

}

}

if (nread > nleft)

exit(EXIT\_FAILURE);

nleft -= nread;

ret = readn(sockfd, bufp, nread);

if (ret != nread)

exit(EXIT\_FAILURE);

bufp += nread;

}

return -1;

}

void echo\_srv(int conn)

{

char recvbuf[1024];

while (1)

{

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

int ret = readline(conn, recvbuf, 1024);

if (ret == -1)

ERR\_EXIT("readline");

if (ret == 0)

{

printf("client close\n");

break;

}

fputs(recvbuf, stdout);

writen(conn, recvbuf, strlen(recvbuf));

}

}

void handle\_sigchld(int sig)

{

/\* wait(NULL);\*/

while (waitpid(-1, NULL, WNOHANG) > 0)

;

}

int main(void)

{

/\* signal(SIGCHLD, SIG\_IGN);\*/

signal(SIGCHLD, handle\_sigchld);

int listenfd;

if ((listenfd = socket(PF\_INET, SOCK\_STREAM, IPPROTO\_TCP)) < 0)

/\* if ((listenfd = socket(PF\_INET, SOCK\_STREAM, 0)) < 0)\*/

ERR\_EXIT("socket");

struct sockaddr\_in servaddr;

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

servaddr.sin\_family = AF\_INET;

servaddr.sin\_port = htons(5188);

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

/\*servaddr.sin\_addr.s\_addr = inet\_addr("127.0.0.1");\*/

/\*inet\_aton("127.0.0.1", &servaddr.sin\_addr);\*/

int on = 1;

if (setsockopt(listenfd, SOL\_SOCKET, SO\_REUSEADDR, &on, sizeof(on)) < 0)

ERR\_EXIT("setsockopt");

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

ERR\_EXIT("bind");

if (listen(listenfd, SOMAXCONN) < 0)

ERR\_EXIT("listen");

struct sockaddr\_in peeraddr;

socklen\_t peerlen;

int conn;

/\*

pid\_t pid;

while (1)

{

if ((conn = accept(listenfd, (struct sockaddr\*)&peeraddr, &peerlen)) < 0)

ERR\_EXIT("accept");

printf("ip=%s port=%d\n", inet\_ntoa(peeraddr.sin\_addr), ntohs(peeraddr.sin\_port));

pid = fork();

if (pid == -1)

ERR\_EXIT("fork");

if (pid == 0)

{

close(listenfd);

echo\_srv(conn);

exit(EXIT\_SUCCESS);

}

else

close(conn);

}

\*/

int i;

int client[FD\_SETSIZE];

int maxi = 0;

for (i=0; i<FD\_SETSIZE; i++)

client[i] = -1;

int nready;

int maxfd = listenfd;

fd\_set rset;

fd\_set allset;

FD\_ZERO(&rset);

FD\_ZERO(&allset);

FD\_SET(listenfd, &allset);

while (1)

{

rset = allset;

nready = select(maxfd+1, &rset, NULL, NULL, NULL);

if (nready == -1)

{

if (errno == EINTR)

continue;

ERR\_EXIT("select");

}

if (nready == 0)

continue;

if (FD\_ISSET(listenfd, &rset))

{

peerlen = sizeof(peeraddr);

conn = accept(listenfd, (struct sockaddr\*)&peeraddr, &peerlen);

if (conn == -1)

ERR\_EXIT("accept");

for (i=0; i<FD\_SETSIZE; i++)

{

if (client[i] < 0)

{

client[i] = conn;

if (i > maxi)

maxi = i;

break;

}

}

if (i == FD\_SETSIZE)

{

fprintf(stderr, "too many clients\n");

exit(EXIT\_FAILURE);

}

printf("ip=%s port=%d\n", inet\_ntoa(peeraddr.sin\_addr), ntohs(peeraddr.sin\_port));

FD\_SET(conn, &allset);

if (conn > maxfd)

maxfd = conn;

if (--nready <= 0)

continue;

}

for (i=0; i<=maxi; i++)

{

conn = client[i];

if (conn == -1)

continue;

if (FD\_ISSET(conn, &rset))

{

char recvbuf[1024] = {0};

int ret = readline(conn, recvbuf, 1024);

if (ret == -1)

ERR\_EXIT("readline");

if (ret == 0)

{

printf("client close\n");

FD\_CLR(conn, &allset);

client[i] = -1;

}

fputs(recvbuf, stdout);

writen(conn, recvbuf, strlen(recvbuf));

if (--nready <= 0)

break;

}

}

}

return 0;

}

# close和shutdown

## 1.区别

close终止了数据传输的两个方向

shutdown可以选择的终止某个方向的数据传送或终止数据传送的两个方向

shutdown how=1可以保证对等方接收到一个EOF字符，而不管其他进程是否已经打开了套接字。而close不能保证，直到套接字引用计数减为0时才发送，也就是说直到所有的进程都关闭了套接字

2.客户端和服务器

客户端

#include <unistd.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <signal.h>

#include <stdlib.h>

#include <stdio.h>

#include <errno.h>

#include <string.h>

#define ERR\_EXIT(m) \

do \

{ \

perror(m); \

exit(EXIT\_FAILURE); \

} while(0)

ssize\_t readn(int fd, void \*buf, size\_t count)

{

size\_t nleft = count;

ssize\_t nread;

char \*bufp = (char\*)buf;

while (nleft > 0)

{

if ((nread = read(fd, bufp, nleft)) < 0)

{

if (errno == EINTR)

continue;

return -1;

}

else if (nread == 0)

return count - nleft;

bufp += nread;

nleft -= nread;

}

return count;

}

ssize\_t writen(int fd, const void \*buf, size\_t count)

{

size\_t nleft = count;

ssize\_t nwritten;

char \*bufp = (char\*)buf;

while (nleft > 0)

{

if ((nwritten = write(fd, bufp, nleft)) < 0)

{

if (errno == EINTR)

continue;

return -1;

}

else if (nwritten == 0)

continue;

bufp += nwritten;

nleft -= nwritten;

}

return count;

}

ssize\_t recv\_peek(int sockfd, void \*buf, size\_t len)

{

while (1)

{

int ret = recv(sockfd, buf, len, MSG\_PEEK);

if (ret == -1 && errno == EINTR)

continue;

return ret;

}

}

ssize\_t readline(int sockfd, void \*buf, size\_t maxline)

{

int ret;

int nread;

char \*bufp = buf;

int nleft = maxline;

while (1)

{

ret = recv\_peek(sockfd, bufp, nleft);

if (ret < 0)

return ret;

else if (ret == 0)

return ret;

nread = ret;

int i;

for (i=0; i<nread; i++)

{

if (bufp[i] == '\n')

{

ret = readn(sockfd, bufp, i+1);

if (ret != i+1)

exit(EXIT\_FAILURE);

return ret;

}

}

if (nread > nleft)

exit(EXIT\_FAILURE);

nleft -= nread;

ret = readn(sockfd, bufp, nread);

if (ret != nread)

exit(EXIT\_FAILURE);

bufp += nread;

}

return -1;

}

void echo\_cli(int sock)

{

/\*

char sendbuf[1024] = {0};

char recvbuf[1024] = {0};

while (fgets(sendbuf, sizeof(sendbuf), stdin) != NULL)

{

writen(sock, sendbuf, strlen(sendbuf));

int ret = readline(sock, recvbuf, sizeof(recvbuf));

if (ret == -1)

ERR\_EXIT("readline");

else if (ret == 0)

{

printf("client close\n");

break;

}

fputs(recvbuf, stdout);

memset(sendbuf, 0, sizeof(sendbuf));

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

}

close(sock);

\*/

fd\_set rset;

FD\_ZERO(&rset);

int nready;

int maxfd;

int fd\_stdin = fileno(stdin);

if (fd\_stdin > sock)

maxfd = fd\_stdin;

else

maxfd = sock;

char sendbuf[1024] = {0};

char recvbuf[1024] = {0};

int stdineof = 0;

while (1)

{

if (stdineof == 0)

FD\_SET(fd\_stdin, &rset);

FD\_SET(sock, &rset);

nready = select(maxfd+1, &rset, NULL, NULL, NULL);

if (nready == -1)

ERR\_EXIT("select");

if (nready == 0)

continue;

if (FD\_ISSET(sock, &rset))

{

int ret = readline(sock, recvbuf, sizeof(recvbuf));

if (ret == -1)

ERR\_EXIT("readline");

else if (ret == 0)

{

printf("server close\n");

break;

}

fputs(recvbuf, stdout);

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

}

if (FD\_ISSET(fd\_stdin, &rset))

{

if (fgets(sendbuf, sizeof(sendbuf), stdin) == NULL)

{

stdineof = 1;

/\*

close(sock);

sleep(5);

exit(EXIT\_FAILURE);

\*/

shutdown(sock, SHUT\_WR);

}

else

{

writen(sock, sendbuf, strlen(sendbuf));

memset(sendbuf, 0, sizeof(sendbuf));

}

}

}

}

void handle\_sigpipe(int sig)

{

printf("recv a sig=%d\n", sig);

}

int main(void)

{

/\*

signal(SIGPIPE, handle\_sigpipe);

\*/

signal(SIGPIPE, SIG\_IGN);

int sock;

if ((sock = socket(PF\_INET, SOCK\_STREAM, IPPROTO\_TCP)) < 0)

ERR\_EXIT("socket");

struct sockaddr\_in servaddr;

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

servaddr.sin\_family = AF\_INET;

servaddr.sin\_port = htons(5188);

servaddr.sin\_addr.s\_addr = inet\_addr("127.0.0.1");

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

ERR\_EXIT("connect");

struct sockaddr\_in localaddr;

socklen\_t addrlen = sizeof(localaddr);

if (getsockname(sock, (struct sockaddr\*)&localaddr, &addrlen) < 0)

ERR\_EXIT("getsockname");

printf("ip=%s port=%d\n", inet\_ntoa(localaddr.sin\_addr), ntohs(localaddr.sin\_port));

echo\_cli(sock);

return 0;

}

服务器

#include <unistd.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <signal.h>

#include <sys/wait.h>

#include <stdlib.h>

#include <stdio.h>

#include <errno.h>

#include <string.h>

#define ERR\_EXIT(m) \

do \

{ \

perror(m); \

exit(EXIT\_FAILURE); \

} while(0)

ssize\_t readn(int fd, void \*buf, size\_t count)

{

size\_t nleft = count;

ssize\_t nread;

char \*bufp = (char\*)buf;

while (nleft > 0)

{

if ((nread = read(fd, bufp, nleft)) < 0)

{

if (errno == EINTR)

continue;

return -1;

}

else if (nread == 0)

return count - nleft;

bufp += nread;

nleft -= nread;

}

return count;

}

ssize\_t writen(int fd, const void \*buf, size\_t count)

{

size\_t nleft = count;

ssize\_t nwritten;

char \*bufp = (char\*)buf;

while (nleft > 0)

{

if ((nwritten = write(fd, bufp, nleft)) < 0)

{

if (errno == EINTR)

continue;

return -1;

}

else if (nwritten == 0)

continue;

bufp += nwritten;

nleft -= nwritten;

}

return count;

}

ssize\_t recv\_peek(int sockfd, void \*buf, size\_t len)

{

while (1)

{

int ret = recv(sockfd, buf, len, MSG\_PEEK);

if (ret == -1 && errno == EINTR)

continue;

return ret;

}

}

ssize\_t readline(int sockfd, void \*buf, size\_t maxline)

{

int ret;

int nread;

char \*bufp = buf;

int nleft = maxline;

while (1)

{

ret = recv\_peek(sockfd, bufp, nleft);

if (ret < 0)

return ret;

else if (ret == 0)

return ret;

nread = ret;

int i;

for (i=0; i<nread; i++)

{

if (bufp[i] == '\n')

{

ret = readn(sockfd, bufp, i+1);

if (ret != i+1)

exit(EXIT\_FAILURE);

return ret;

}

}

if (nread > nleft)

exit(EXIT\_FAILURE);

nleft -= nread;

ret = readn(sockfd, bufp, nread);

if (ret != nread)

exit(EXIT\_FAILURE);

bufp += nread;

}

return -1;

}

void echo\_srv(int conn)

{

char recvbuf[1024];

while (1)

{

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

int ret = readline(conn, recvbuf, 1024);

if (ret == -1)

ERR\_EXIT("readline");

if (ret == 0)

{

printf("client close\n");

break;

}

fputs(recvbuf, stdout);

writen(conn, recvbuf, strlen(recvbuf));

}

}

void handle\_sigchld(int sig)

{

/\* wait(NULL);\*/

while (waitpid(-1, NULL, WNOHANG) > 0)

;

}

void handle\_sigpipe(int sig)

{

printf("recv a sig=%d\n", sig);

}

int main(void)

{

/\*signal(SIGPIPE, SIG\_IGN);\*/

signal(SIGPIPE, handle\_sigpipe);

/\* signal(SIGCHLD, SIG\_IGN);\*/

signal(SIGCHLD, handle\_sigchld);

int listenfd;

if ((listenfd = socket(PF\_INET, SOCK\_STREAM, IPPROTO\_TCP)) < 0)

/\* if ((listenfd = socket(PF\_INET, SOCK\_STREAM, 0)) < 0)\*/

ERR\_EXIT("socket");

struct sockaddr\_in servaddr;

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

servaddr.sin\_family = AF\_INET;

servaddr.sin\_port = htons(5188);

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

/\*servaddr.sin\_addr.s\_addr = inet\_addr("127.0.0.1");\*/

/\*inet\_aton("127.0.0.1", &servaddr.sin\_addr);\*/

int on = 1;

if (setsockopt(listenfd, SOL\_SOCKET, SO\_REUSEADDR, &on, sizeof(on)) < 0)

ERR\_EXIT("setsockopt");

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

ERR\_EXIT("bind");

if (listen(listenfd, SOMAXCONN) < 0)

ERR\_EXIT("listen");

struct sockaddr\_in peeraddr;

socklen\_t peerlen;

int conn;

/\*

pid\_t pid;

while (1)

{

if ((conn = accept(listenfd, (struct sockaddr\*)&peeraddr, &peerlen)) < 0)

ERR\_EXIT("accept");

printf("ip=%s port=%d\n", inet\_ntoa(peeraddr.sin\_addr), ntohs(peeraddr.sin\_port));

pid = fork();

if (pid == -1)

ERR\_EXIT("fork");

if (pid == 0)

{

close(listenfd);

echo\_srv(conn);

exit(EXIT\_SUCCESS);

}

else

close(conn);

}

\*/

int i;

int client[FD\_SETSIZE];

int maxi = 0;

for (i=0; i<FD\_SETSIZE; i++)

client[i] = -1;

int nready;

int maxfd = listenfd;

fd\_set rset;

fd\_set allset;

FD\_ZERO(&rset);

FD\_ZERO(&allset);

FD\_SET(listenfd, &allset);

while (1)

{

rset = allset;

nready = select(maxfd+1, &rset, NULL, NULL, NULL);

if (nready == -1)

{

if (errno == EINTR)

continue;

ERR\_EXIT("select");

}

if (nready == 0)

continue;

if (FD\_ISSET(listenfd, &rset))

{

peerlen = sizeof(peeraddr);

conn = accept(listenfd, (struct sockaddr\*)&peeraddr, &peerlen);

if (conn == -1)

ERR\_EXIT("accept");

for (i=0; i<FD\_SETSIZE; i++)

{

if (client[i] < 0)

{

client[i] = conn;

if (i > maxi)

maxi = i;

break;

}

}

if (i == FD\_SETSIZE)

{

fprintf(stderr, "too many clients\n");

exit(EXIT\_FAILURE);

}

printf("ip=%s port=%d\n", inet\_ntoa(peeraddr.sin\_addr), ntohs(peeraddr.sin\_port));

FD\_SET(conn, &allset);

if (conn > maxfd)

maxfd = conn;

if (--nready <= 0)

continue;

}

for (i=0; i<=maxi; i++)

{

conn = client[i];

if (conn == -1)

continue;

if (FD\_ISSET(conn, &rset))

{

char recvbuf[1024] = {0};

int ret = readline(conn, recvbuf, 1024);

if (ret == -1)

ERR\_EXIT("readline");

if (ret == 0)

{

printf("client close\n");

FD\_CLR(conn, &allset);

client[i] = -1;

close(conn);

}

fputs(recvbuf, stdout);

sleep(4);

writen(conn, recvbuf, strlen(recvbuf));

if (--nready <= 0)

break;

}

}

}

return 0;

}

# 套接字I/O超时

## 1.套接字I/O超时设置方法

### 1.1 alarm