# 总结

|  |
| --- |
| socket()： 创建socket  bind()： 绑定socket到本地地址和端口，通常由服务端调用  listen()： TCP专用，开启监听模式  accept()： TCP专用，服务器等待客户端连接，一般是阻塞态  connect()： TCP专用，客户端主动连接服务器  send()： TCP专用，发送数据  recv()： TCP专用，接收数据  sendto()： UDP专用，发送数据到指定的IP地址和端口  recvfrom()： UDP专用，接收数据，返回数据远端的IP地址和端口  closesocket()： 关闭socket  **int socket(int domain, int type, int protocol);**  **domain**：  即协议域，又称为协议族（family）。常用的协议族有，**AF\_INET、AF\_INET6、AF\_LOCAL**（或称AF\_UNIX，Unix域socket）、**AF\_ROUTE**等等。协议族决定了socket的地址类型，在通信中必须采用对应的地址，如AF\_INET决定了要用ipv4地址（32位的）与端口号（16位的）的组合、AF\_UNIX决定了要用一个绝对路径名作为地址。  **type**：  指定socket类型。常用的socket类型有，S**OCK\_STREAM、SOCK\_DGRAM、SOCK\_RAW、SOCK\_PACKET、SOCK\_SEQPACKET**等等（socket的类型有哪些？）。  **protocol**：  协议。如**IPPROTO\_TCP、IPPTOTO\_UDP、IPPROTO\_SCTP、IPPROTO\_TIPC**等，它们分别对应TCP传输协议、UDP传输协议、STCP传输协议、TIPC传输协议。当protocol为0时，会自动选择type类型对应的默认协议。 |

# 概念

**TCP：**传输控制协议，一种面向连接的协议，给用户进程提供可靠的全双工的字节流，TCP套接口是字节流套接口(stream socket)的一种。

**UDP：**用户数据报协议。UDP是一种无连接协议。UDP套接口是数据报套接口(datagram socket)的一种。

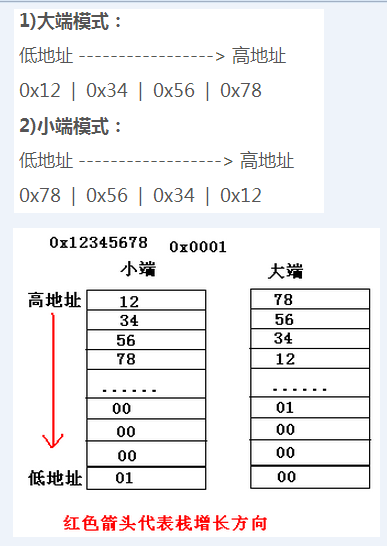
**主机字节序**就是我们平常说的大端和小端模式：不同的CPU有不同的字节序类型，这些字节序是指整数在内存中保存的顺序，这个叫做主机序。引用标准的Big-Endian和Little-Endian的定义如下：

**a) Little-Endian就是低位字节排放在内存的低地址端，高位字节排放在内存的高地址端。**

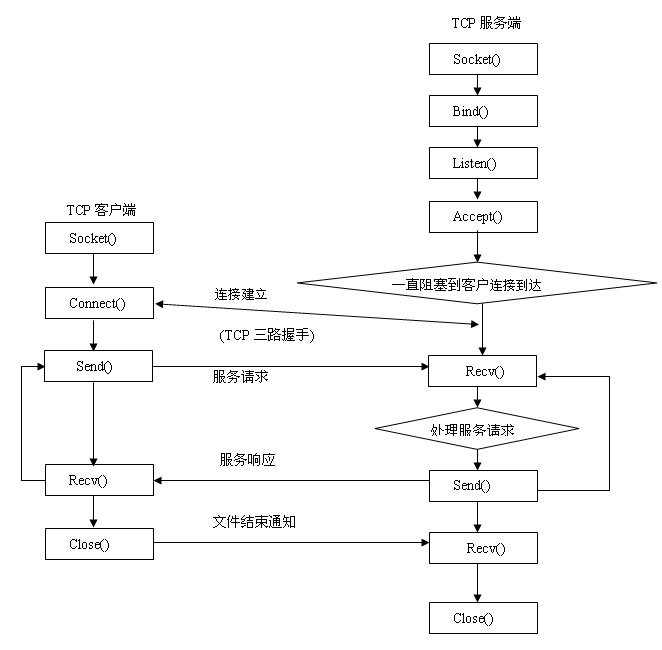
**b) Big-Endian就是高位字节排放在内存的低地址端，低位字节排放在内存的高地址端。**

**网络字节序**：4个字节的32 bit值以下面的次序传输：首先是0～7bit，其次8～15bit，然后16～23bit，最后是24~31bit。这种传输次序称作**大端字节序**。由于TCP/IP首部中所有的二进制整数在网络中传输时都要求以这种次序，因此它又称作**网络字节序**。字节序，顾名思义字节的顺序，就是大于一个字节类型的数据在内存中的存放顺序，一个字节的数据没有顺序的问题了。

所以：在将一个地址绑定到socket的时候，请先将主机字节序转换成为网络字节序，而不要假定主机字节序跟网络字节序一样使用的是Big-Endian。由于这个问题曾引发过血案！公司项目代码中由于存在这个问题，导致了很多莫名其妙的问题，所以请谨记对主机字节序不要做任何假定，务必将其转化为网络字节序再赋给socket。



# TCP框架

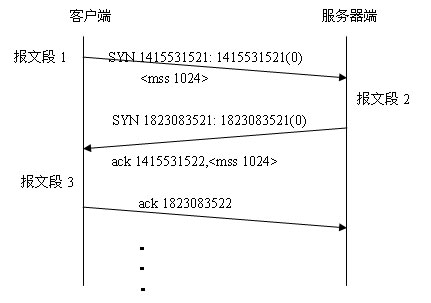


## 三路握手：

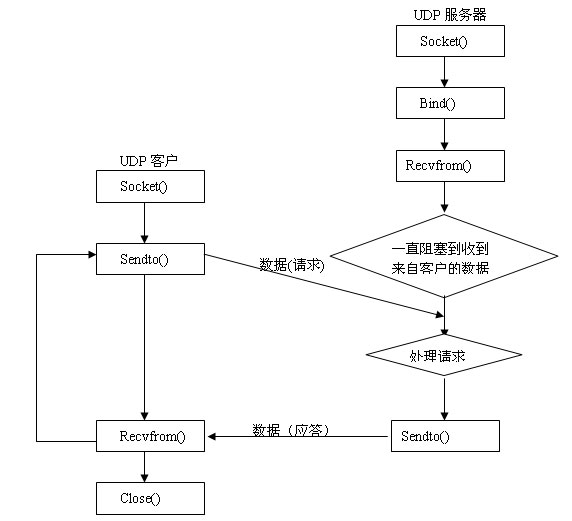
1.**客户端**发送一个**SYN**段（同步序号）指明客户打算连接的服务器端口，以及初始化序号(ISN) 。

2.**服务器**发回包含服务器的初始序号的**SYN**报文段作为应答。同时，将确认序号(ACK)设置为客户的ISN加1以对客户的SYN 报文段进行确认。一个SYN将占用一个序号。

3.**客户**必须将确认序号设置为服务器的ISN加1以对服务器的SYN报文段进行确认。



# UDP框架



# UDP和TCP的对比：

从上面的流程图比较我们可以很明显的看出**UDP没有三次握手过程**。

简单点说。UDP处理的细节比TCP少。**UDP不能保证消息被传送到**（它也报告消息没有传送到）目的地。UDP也**不保证数据包的传送顺序**。UDP把数据发出去后只能希望它能够抵达目的地。

**TCP优缺点：**

优点：

1．TCP提供以认可的方式显式地创建和终止连接。

2．TCP保证可靠的、顺序的（数据包以发送的顺序接收）以及不会重复的数据传输。

3．TCP处理流控制。

4．允许数据优先

5．如果数据没有传送到，则TCP套接口返回一个出错状态条件。

6．TCP通过保持连续并将数据块分成更小的分片来处理大数据块。—无需程序员知道

缺点：

TCP在转移数据时必须创建（并保持）一个连接。这个连接给通信进程增加了开销，让它比UDP速度要慢。

**UDP优缺点：**

1．UDP不要求保持一个连接

2．UDP没有因接收方认可收到数据包（或者当数据包没有正确抵达而自动重传）而带来的开销。

3．设计UDP的目的是用于短应用和控制消息

4．在一个数据包连接一个数据包的基础上，UDP要求的网络带宽比TDP更小。

## 最优的 TCP socket 缓冲区大小

|  |
| --- |
| BDP = 带宽 \* 延迟 (单位： 字节)  带宽\*延时/8\*2^20 = \*\*\*B  带宽: Mebibites 延时:s （RTT）    注意：调整tcp前，必须调整core，对于tcp,udp来说不能超出core的限制    默认值：  net.core.rmem\_max = 131071 128K  net.core.rmem\_default = 124928  net.core.wmem\_max = 131071  net.core.wmem\_default = 124928    net.ipv4.tcp\_rmem = 4096 87380 2691072  net.ipv4.tcp\_wmem = 4096 16384 2691072    示例：  带宽：1.5M 100M  延时：500ms 3000ms  1.5 \* 0.5 / 8 \* 2^20 ======= net.ipv4.tcp\_wmem  100 \* 3 / 8 \* 2^20 ======= net.ipv4.tcp\_wmem |

## 常见网络参数整理

|  |
| --- |
| **1.tcp连接保持管理：**  net.ipv4.tcp\_keepalive\_time = 7200  如果在该参数指定时间内某条连接处于空闲状态，则内核向远程主机发起探测  net.ipv4.tcp\_keepalive\_intvl = 75  内核向远程主机发送的保活探测的时间间隔  net.ipv4.tcp\_keepalive\_probes = 9  内核发送保活探测的最大次数，如果探测次数大于这个数，则断定远程主机不可达，则关闭该连接并释放本地资源  一个连接7200s空闲后，内核会每隔75秒去重试，若连续9次则放弃。这样就导致一个连接经过2h11min的时间才能被丢弃，降低该值能够尽量减小  失效连接所占用的资源，而被新的连接所使用。      **2.tcp连接管理：**  net.core.netdev\_max\_backlog=3000  该文件表示在每个网络接口接收数据包的速率比内核处理这些包的速率快时，允许送到队列的数据包的最大数目。  net.ipv4.tcp\_max\_syn\_backlog = 1024  控制每个端口的tcp syn的队列长度，来自客户端的连接请求需要排队，直至服务器接受，如果连接请求数大于该值，则连接请求会被丢弃，客户端无法  连接服务器，一般服务器要提高此值  net.ipv4.tcp\_synack\_retries = 5  控制内核向某个socket的ack,syn段重新发送响应的次数，降低此值可以尽早检测到来自远程主机的连接失败尝试  net.ipv4.tcp\_retries2 = 15  控制内核向已建立连接的远程主机重新发送数据的次数，降低此值，可以尽早的检测连接失效    **3.net.ipv4.tcp\_syncookies = 1**  SYN Cookie是对TCP服务器端的三次握手协议作一些修改，专门用来防范SYN Flood攻击的一种手段。它的原理是，在TCP服务器收到TCP SYN包并返回TCP SYN+ACK包时，不分配一个专门的数据区，而是根据这个SYN包计算出一个cookie值。在收到TCP ACK包时，TCP服务器在根据那个cookie值检查这个TCP ACK包的合法性。如果合法，再分配专门的数据区进行处理未来的TCP连接。      **4.net.ipv4.tcp\_tw\_reuse = 1**  表示开启重用。允许将TIME-WAIT sockets重新用于新的TCP连接，默认为0  net.ipv4.tcp\_tw\_recycle = 1  表示开启TCP连接中TIME-WAIT sockets的快速回收，默认为0    **5.net.ipv4.tcp\_fin\_timeout = 60**  缩短默认的timeout的时间    **6. /proc/sys/net/ipv4/conf/\*/accept\_redirects**  　　 如果主机所在的网段中有两个路由器，你将其中一个设置成了缺省网关，但是该网关在收到你的ip包时发现该ip包必须经过另外一个路由器，这时这个路由器就会给你  发一个所谓的“重定向”icmp包，告诉将ip包转发到另外一个路由器。参数值为布尔值，1表示接收这类重定向icmp 信息，0表示忽略。在充当路由器的linux主机上缺  省值为0，在一般的linux主机上缺省值为1。建议将其改为0以消除安全性隐患。 |

# Socket编程 – TCP

## 服务器

工作流程：

1. 创建一个Socket，

2. 调用bind函数将其与本机地址以及一个本地端口号绑定。

3. 调用listen在相应的socket上监听，

4. 调用accept()等待

5. 当客户端调用connect()后， accpet()接收到一个连接服务请求时，将生成一个新的socket。该socket包含客户端的IP和port信息。

5. 服务器调用send()函数，通过新的socket向客户端发送字符串" hi,I am server!"。

6. 关闭该socket。

|  |
| --- |
| #include <stdio.h>  #include <string.h>  #include <stdlib.h>  #include <arpa/inet.h>  #include <sys/types.h> /\* See NOTES \*/  #include <sys/socket.h>  #include <errno.h>  #define SERVER\_PORT (12345)  #define MAX\_MSG\_SIZE (1024)  int main()  {  int sock\_fd, cli\_sockfd; /\*sock\_fd：监听socket；client\_fd：数据传输socket \*/  struct sockaddr\_in cli\_addr; /\* 客户端地址信息 \*/  char msg[MAX\_MSG\_SIZE];/\* 缓冲区\*/  //Step1. 创建TCP的SOCKET  int ser\_sockfd=**socket**(AF\_INET,SOCK\_**STREAM**,0);  if(ser\_sockfd<0)  {  fprintf(stderr,"socker Error:%s\n",strerror(errno));  exit(1);  }  //Step2. 将本机的Ip和port绑定到这个ser\_sockfd中  struct sockaddr\_in ser\_addr;  int addrlen=sizeof(struct sockaddr\_in);  bzero(&ser\_addr,addrlen);  ser\_addr.sin\_family=AF\_INET;  ser\_addr.sin\_addr.**s\_addr**=htonl(**INADDR**\_**ANY**);  ser\_addr.**sin\_port**=htons(**SERVER**\_**PORT**);  if(**bind**(ser\_sockfd,(struct sockaddr\*)&ser\_addr,sizeof(struct sockaddr\_in))<0)  { /\*绑定失败 \*/  fprintf(stderr,"Bind Error:%s\n",strerror(errno));  exit(1);  }  //Step3. 启动ser\_sockfd的监听功能  if(**listen**(ser\_sockfd, 20)<0) //20表示最大允许20个用户请求排队  {  fprintf(stderr,"Listen Error:%s\n",strerror(errno));  close(ser\_sockfd);  exit(1);  }  while(1)  {  //Step4. 等待接收客户连接请求\*/  cli\_sockfd=**accept**(ser\_sockfd,(struct sockaddr\*) &cli\_addr,&addrlen);  if(cli\_sockfd<=0)  {  fprintf(stderr,"Accept Error:%s\n",strerror(errno));  }  else  {  //Step5. 有客户端连接，接收客户端信i息  //ssize\_t **recv**(int sockfd, void \*buf, size\_t len, int flags);  **recv**(cli\_sockfd,msg,MAX\_MSG\_SIZE,0);  printf("received a connection from %sn", inet\_ntoa(cli\_addr.sin\_addr));  printf("%s\n",msg);/\*在屏幕上打印出来 \*/  //Step6. 发送信息给客户端  strcpy(msg,"hi,I am server!");  **send**(cli\_sockfd,msg,sizeof(msg),0);  //Step7. 关闭客户端的连接  **close**(cli\_sockfd);  }  }  //Step8. 关闭服务端的连接  **close**(ser\_sockfd);  } |

## 客户端

工作流程：

1. 调用socket函数创建一个Socket，

2. 调用bind函数将其与本机地址以及一个本地端口号绑定，

3. 调用connect()请求连接服务器，

4. 调用send()向新的socket向客户端发送字符串" hi,I am client!"。

5.关闭该socket。

|  |
| --- |
| #include <stdio.h>  #include <string.h>  #include <stdlib.h>  #include <arpa/inet.h>  #include <sys/types.h> /\* See NOTES \*/  #include <sys/socket.h>  #include <errno.h>  #define SERVER\_PORT (12345)  #define MAX\_MSG\_SIZE (1024)  int GetServerAddr(char \* addrname)  {  printf("please input server addr:\n");  scanf("%s",addrname);  printf("addrname = %s\n", addrname);  return 1;  }  int main()  {  struct sockaddr\_in cli\_addr;/\* 客户端的地址\*/  char msg[MAX\_MSG\_SIZE];/\* 缓冲区\*/  char seraddr[14];  struct sockaddr\_in ser\_addr;  //Step1. 创建TCP的SOCKET  int cli\_sockfd=**socket**(AF\_INET,SOCK\_**STREAM**,0);  if(cli\_sockfd<0)  {/\*创建失败 \*/  fprintf(stderr,"socker Error:%s\n",strerror(errno));  exit(1);  }  //Step2. 绑定客户端地址  bzero(&ser\_addr, sizeof(ser\_addr));  cli\_addr.sin\_family=AF\_INET;  cli\_addr.sin\_addr.**s\_add**r=htonl(**INADDR**\_**ANY**);  cli\_addr.sin\_port=0;  if(**bind**(cli\_sockfd,(struct sockaddr\*)&cli\_addr, sizeof(ser\_addr))<0)  {  fprintf(stderr,"Bind Error:%s\n",strerror(errno));  exit(1);  }  //Step3. 连接到服务器地址  GetServerAddr(**seraddr**); /\* 串口输入服务器的地址\*/  bzero(&ser\_addr, sizeof(ser\_addr));  ser\_addr.sin\_family=AF\_INET;  ser\_addr.sin\_addr.s\_addr=inet\_addr(**seraddr**);  ser\_addr.sin\_port=htons(**SERVER**\_**PORT**);  if(**connect**(cli\_sockfd,(struct sockaddr\*)&ser\_addr, sizeof(ser\_addr))!=0)/\*请求连接\*/  {  /\*连接失败 \*/  fprintf(stderr,"Connect Error:%s\n",strerror(errno));  close(cli\_sockfd);  exit(1);  }  //Step4. 发送消息到服务器  strcpy(msg,"hi,I am client!");  **send**(cli\_sockfd,msg,sizeof(msg),0);/\*发送数据\*/  //Step5. 从服务器接收消息  **recv**(cli\_sockfd,msg,MAX\_MSG\_SIZE,0); /\* 接受数据\*/  printf("%s\n",msg);/\*在屏幕上打印出来 \*/  //Step6. 关闭客户端本地连接  **close**(cli\_sockfd);  } |

# Socket编程 – UDP

## 服务器

工作流程：首先调用socket函数创建一个Socket，然后调用bind函数将其与本机地址以及一个本地端口号绑定，接收到一个客户端时，服务器显示该客户端的IP地址，并将字串返回给客户端。

|  |
| --- |
| #include <stdio.h>  #include <string.h>  #include <stdlib.h>  #include <arpa/inet.h>  #include <sys/types.h> /\* See NOTES \*/  #include <sys/socket.h>  #include <errno.h>  #define SERVER\_PORT (12345)  int main(int argc,char \*\*argv)  {  int ser\_sockfd;  int len;  socklen\_t addrlen;  char seraddr[100];  struct sockaddr\_in ser\_addr;  //Step1. 建立UDP类型的socket  ser\_sockfd=**socket**(AF\_INET,SOCK\_**DGRAM**,0);  if(ser\_sockfd<0)  {  printf("I cannot socket success\n");  return 1;  }    //Step2. 绑定本机的IP和port到socket中  addrlen=sizeof(struct sockaddr\_in);  bzero(&ser\_addr,addrlen);  ser\_addr.sin\_family=AF\_INET;  ser\_addr.sin\_addr.s\_addr=htonl(**INADDR**\_**ANY**);  ser\_addr.sin\_port=htons(**SERVER**\_**PORT**);  /\*绑定客户端\*/  if(**bind**(ser\_sockfd,(struct sockaddr \*)&ser\_addr,addrlen)<0)  {  printf("connect");  return 1;  }  while(1)  {  //Step3. 等待客户端连接并接收客户端数据  bzero(seraddr,sizeof(seraddr));  len=**recvfrom**(ser\_sockfd,seraddr,sizeof(seraddr),0,(struct sockaddr\*)&ser\_addr,&addrlen);  /\*显示client端的网络地址\*/  printf("receive from %s\n",inet\_ntoa(ser\_addr.sin\_addr));  /\*显示客户端发来的字串\*/  printf("recevce:%s",seraddr);  //Step4. 将字串返回给client端\*/  **sendto**(ser\_sockfd,seraddr,len,0,(struct sockaddr\*)&ser\_addr,addrlen);  }  return 0;  } |

## 客户端

工作流程：首先调用socket函数创建一个Socket，填写服务器地址及端口号，从标准输入设备中取得字符串，将字符串传送给服务器端，并接收服务器端返回的字符串。最后关闭该socket。

|  |
| --- |
| #include <stdio.h>  #include <string.h>  #include <stdlib.h>  #include <arpa/inet.h>  #include <sys/types.h> /\* See NOTES \*/  #include <sys/socket.h>  #include <errno.h>  #include <unistd.h>  #define SERVER\_PORT (12345)  int GetServerAddr(char \* addrname)  {  printf("please input server addr:");  scanf("%s",addrname);  return 1;  }  int main(int argc,char \*\*argv)  {  int cli\_sockfd;  int len;  socklen\_t addrlen;  struct sockaddr\_in cli\_addr;  char buffer[256];  //Step1. 建立UDP类型的socket  cli\_sockfd=**socket**(AF\_INET,SOCK\_**DGRAM**,0);  if(cli\_sockfd<0)  {  printf("I cannot socket success\n");  return 1;  }  //Step2. 绑定服务端地址  char seraddr[14];  GetServerAddr(**seraddr**);  addrlen=sizeof(struct sockaddr\_in);  bzero(&cli\_addr,addrlen);  cli\_addr.sin\_family=AF\_INET;  cli\_addr.sin\_addr.s\_addr=inet\_addr(**seraddr**);  **cli**\_**addr**.sin\_port=htons(**SERVER**\_**PORT**);  //Step3. 发送消息给服务端  printf("please input send contexts:\n");  bzero(buffer,sizeof(buffer));  /\* 从标准输入设备取得字符串\*/  len=**read**(STDIN\_FILENO,buffer,sizeof(buffer));  /\* 将字符串传送给server端\*/  **sendto**(cli\_sockfd,buffer,len,0,(struct sockaddr\*)&**cli**\_**addr**,addrlen);  //Step4.接收server端返回的字符串  len=**recvfrom**(cli\_sockfd,buffer,sizeof(buffer),0,(struct sockaddr\*)&cli\_addr,&addrlen);  printf("receive from %s\n",inet\_ntoa(cli\_addr.sin\_addr));  printf("receive: %s",buffer);  **close**(cli\_sockfd);  } |

# Makefile文件

|  |
| --- |
| CC=gcc  CFLAGS=-o  all:tcp\_server tcp\_client udp\_server udp\_client  tcp\_server: tcp\_server.c  $(CC) $(CFLAGS) $@ tcp\_server.c  tcp\_client: tcp\_client.c  $(CC) $(CFLAGS) $@ tcp\_client.c  udp\_server: udp\_server.c  $(CC) $(CFLAGS) $@ udp\_server.c  udp\_client: udp\_client.c  $(CC) $(CFLAGS) $@ udp\_client.c  clean:  rm -f tcp\_server tcp\_client udp\_server udp\_client |

# End