一、理解socket

1、socket即为套接字，在TCP/IP协议中，“IP地址+TCP或UDP端口号”唯一的标识网络通讯中的一个进程，“IP地址+TCP或UDP端口号”就为socket。

2、在TCP协议中，建立连接的两个进程（客户端和服务器）各自有一个socket来标识，则这两个socket组成的socket pair就唯一标识一个连接。

3、socket本身就有“插座”的意思，因此用来形容网络连接的一对一关系，为TCP/IP协议设计的应用层编程接口称为socket API。

二、网络字节序

内存中的多字节数据都有大小端之分，磁盘文件中的多字节数据相对于文件中的偏移地址也有大小端之分，同样，网络数据流也有大小端之分。

网络数据流的地址规定：先发出的数据时低地址，后发出的数据是高地址。发送主机通常将发送缓冲区中的数据按内存地址从低到高的顺序发出，为了不使数据流乱序，接收主机也会把从网络上接收的数据按内存地址从低到高的顺序保存在接收缓冲区中。

TCP/IP协议规定：网络数据流应采用大端字节序，即低地址高字节。

（PS：如果对大端字节序小端字节序不明白的童鞋们，可以看这篇文章参考一下：http://blog.csdn.net/qq_33951180/article/details/60767876）

由于两端的两个主机的大小端不一定相同，因此为了使这些网络数据具有更强的可移植性，使相同的代码在大端和小端主机上都能正常运行，我们可以调用以下库函数进行网络字节序和主机字节序的相关转换：

```
#include<arpa/inet.h>

//将主机字节序转换为网络字节序

uint32_t htonl(uint32_t
hostlong);//将32长整数从主机字节序转换为网络字节序，

 //如果主机字节序是小端，则函数会做相应大小
```

//端转换后返回；如果主机字节序是大端，则函

//数不做转换，将参数原封不动返回。。。下同

uint16_t htons(uint16_t hostshort);

//将网络字节序转换为主机字节序

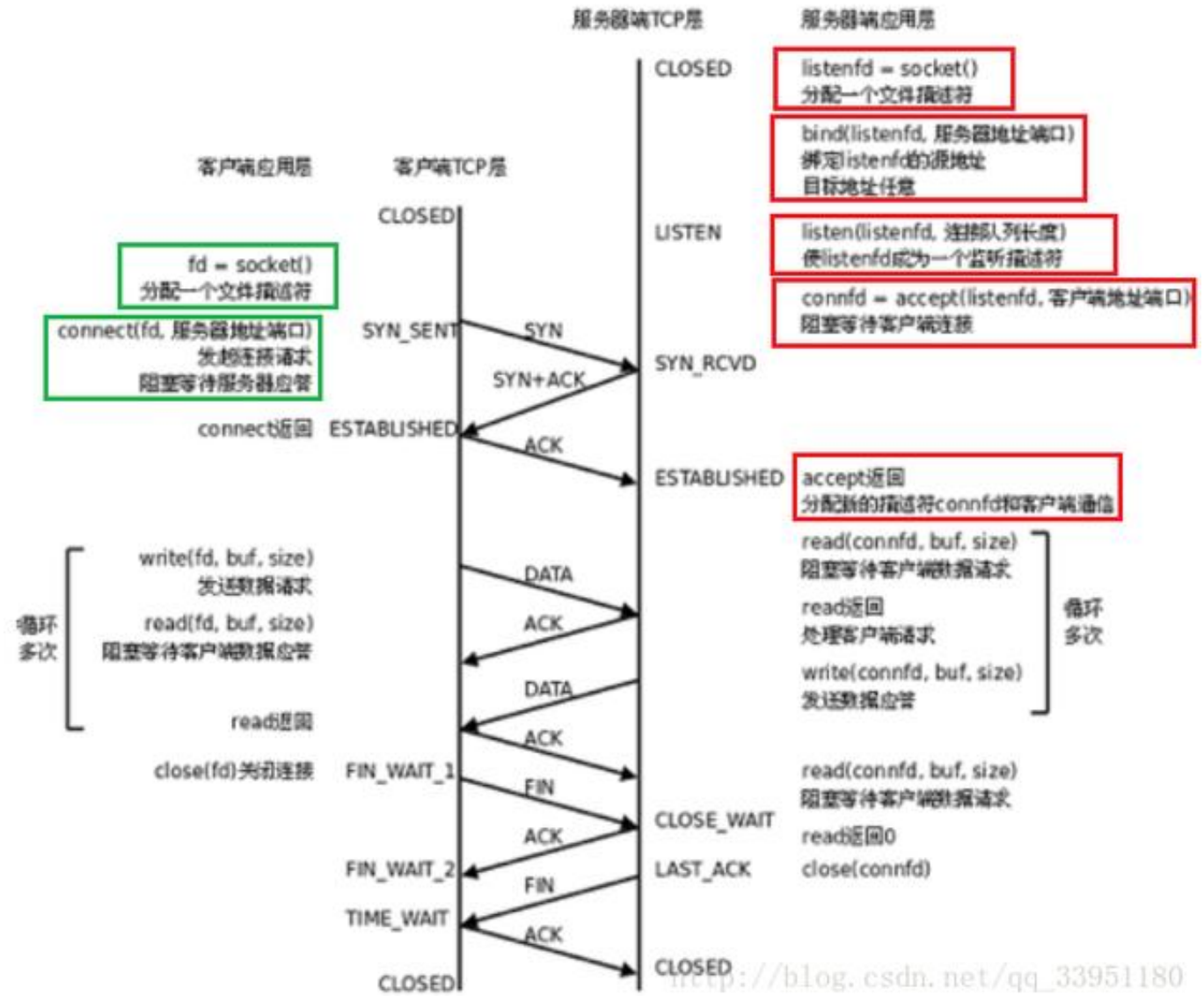
uint32_t ntohl(uint32_t netlong);

uint16_t ntohs(uint16_t netshort);

// h表示主机（host），n表示网络（net），l表示32位长整数，s表示16短整数。

三、TCP协议通讯的实现

TCP协议通讯流程：



这里写图片描述

我们先介绍几个函数：

1、创建套接字

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

//domain:该参数一般被设置为AF_INET，表示使用的是IPv4地址。还有更多选项可以利用man查看该函数

//type:该参数也有很多选项，例如SOCK_STREAM表示面向流的传输协议，SOCK_DGRAM表示数据报，我们这里实现的是TCP，因此选用SOCK_STREAM，如果实

现UDP可选SOCK_DGRAM

//protocol:协议类型，一般使用默认，设置为0

该函数用于打开一个网络通讯接口，出错则返回-1，成功返回一个socket(文件描述符)，应用进程就可以像读写文件一样调用read/write在网络上收发数据。

2、绑定

int bind(int sockfd,const struct sockaddr*addr,socklen_t addrlen);

//sockfd：服务器打开的sock

//后两个参数可以参考第四部分的介绍

服务器所监听的网络地址和端口号一般是固定不变的，客户端程序得知服务器程序的地址和端口号后就可以向服务器发起连接，因此服务器需要调用bind来绑定一个固定的网络地址和端口号。bind成功返回0，出错返回-1。

bind()的作用：将参数sockfd和addr绑定在一起，是sockfd这个用于网络通讯的文件描述符监听addr所描述的地址和端口号。

3、监听

int listen(int sockfd,int backlog);

//sockfd的含义与bind中的相同。

//backlog参数解释为内核为次套接口排队的最大数量，这个大小一般为5~10，不宜太大（是为了防止SYN攻击）

该函数仅被服务器端使用，listen()声明sockfd处于监听状态，并且最多允许有backlog个客户端处于连接等待状态，如果收到更多的连接请求就忽略。listen()成功返回0，失败返回-1。

4、接收连接

int accept(int sockfd,struct sockaddr* addr,socklen_t* addrlen);

//addrlen是一个传入传出型参数，传入的是调用者的缓冲区cliaddr的长度，以避免缓冲区溢出问题；传出的是客户端地址结构体的实际长度（有可能没有占满调用者提供的缓冲区）。如果给cliaddr参数传NULL，表示不关心客户端的地址。

典型的服务器程序是可以同时服务多个客户端的，当有客户端发起连接时，服务器就调用accept()返回并接收这个连接，如果有大量客户端发起请求，服务器来不及处理，还没有accept的客户端就处于连接等待状态。

三次握手完成后，服务器调用accept()接收连接，如果服务器调用accept()时还没有客户端的连接请求，就阻塞等待直到有客户端连接上来。

5、请求连接

int connect(int sockfd,const struct sockaddr* addr,socklen_t addrlen);

这个函数只需要有客户端程序来调用，调用该函数后表明连接服务器，这里的参数都是对方的地址。connect()成功返回0，出错返回-1。

了解这些函数后，我们来捋一捋客户端程序和服务器程序建立连接的过程：

服务器：首先调用socket()创建一个套接字用来通讯，其次调用bind()进行绑定这个文件描述符，并调用listen()用来监听端口是否有客户端请求来，如果有，就调用accept()进行连接，否则就继续阻塞式等待直到有客户端连接上来。连接建立后就可以进行通信了。

客户端：调用socket()分配一个用来通讯的端口，接着就调用connect()发出SYN请求并处于阻塞等待服务器应答状态，服务器应答一个SYN-ACK分段，客户端收到后从connect()返回，同时应答一个ACK分段，服务器收到后从accept()返回，连接建立成功。客户端一般不调用bind()来绑定一个端口号，并不是不允许bind()，服务器也不是必须要bind()。

思考题：为什么不建议客户端进行bind()？

答：当客户端没有自己进行bind时，系统随机分配给客户端一个端口号，并且在分配的时候，操作系统会做到不与现有的端口号发生冲突。但如果自己进行bind，客

户端程序就很容易出现问题，假设在一个PC机上开启多个客户端进程，如果是用户自己绑定了端口号，必然会造成端口冲突，影响通信。

进行一番理论知识后我们就可以写代码了：

“server.c”

```
#include<stdio.h>

#include<sys/types.h>

#include<sys/socket.h>

#include<stdlib.h>

#include<unistd.h>

#include <netinet/in.h>

#include <arpa/inet.h>

int startup(int _port,const char* _ip)

{

 int sock = socket(AF_INET,SOCK_STREAM,0);

 if(sock < 0)

 {

 perror("socket");

 exit(1);

 }

 struct sockaddr_in local;
```

```
 local.sin_family = AF_INET;

 local.sin_port = htons( _port);

 local.sin_addr.s_addr = inet_addr(_ip);

 socklen_t len = sizeof(local);

 if(bind(sock,(struct sockaddr*)&local , len) < 0)

 {

 perror("bind");

 exit(2);

 }

 if(listen(sock, 5) < 0) //允许连接的最大数量为5

 {

 perror("listen");

 exit(3);

 }

 return sock;

}

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

{

 if(argc != 3)

 {
```

```
printf("Usage:%s [loacl_ip] [loacl_port]\n",argv[0]);

return 1;

}

int listen_sock = startup(atoi(argv[2]),argv[1]);//初始化

//用来接收客户端的socket地址结构体

struct sockaddr_in remote;

socklen_t len = sizeof(struct sockaddr_in);

while(1)

{

int sock = accept(listen_sock, (struct sockaddr*)&remote, &len);

if(sock < 0)

{

perror("accept");

continue;

}

printf("get a client, ip:%s,
port:%d\n",inet_ntoa(remote.sin_addr),ntohs(remote.sin_port));

char buf[1024];

while(1)

{
```

```
 ssize_t _s = read(sock, buf, sizeof(buf)-1);

 if(_s > 0)

 {

 buf[_s] = 0;

 printf("client:%s",buf);

 }

 else

 {

 printf("client is quit!\n");

 break;

 }

 }

 }

 return 0;

}
```

” client.c “

```
#include<stdio.h>

#include<sys/types.h>

#include<sys/socket.h>
```

```
#include<unistd.h>

#include<stdlib.h>

#include<netinet/in.h>

#include<arpa/inet.h>

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

{

 if(argc != 3)

 {

 printf("Usage:%s [ip] [port]\n",argv[0]);

 return 0;

 }

 //创建一个用来通讯的socket

 int sock = socket(AF_INET,SOCK_STREAM, 0);

 if(sock < 0)

 {

 perror("socket");

 return 1;

 }

 //需要connect的是对端的地址，因此这里定义服务器端的地址结构体

 struct sockaddr_in server;
```

```
server.sin_family = AF_INET;

server.sin_port = htons(atoi(argv[2]));

server.sin_addr.s_addr = inet_addr(argv[1]);

socklen_t len = sizeof(struct sockaddr_in);

if(connect(sock, (struct sockaddr*)&server, len) < 0 )

{

perror("connect");

return 2;

}

//连接成功进行收数据

char buf[1024];

while(1)

{

printf("send###");

fflush(stdout);

ssize_t _s = read(0, buf, sizeof(buf)-1);

buf[_s] = 0;

write(sock, buf, _s);

}

close(sock);
```

```
 return 0;

}
```

但是这样实现只能进行单进程通信，也就是说每次只能使一个客户端连接上进行数据通讯，这显然不符合服务器的基本要求。我们可以想办法修改服务器端的代码，每次accept成功之后就创建一个子进程，让子进程去处理读写数据，父进程继续监听并accept。

具体代码：https://github.com/lybb/Linux/tree/master/TCP_pro

修改的代码在服务器程序的socket()和bind()之间加入了如下的代码：

```
int opt=1;

setsockopt(sock,SOL_SOCKET,SO_REUSEADDR,&opt,sizeof(opt));

//设置sockfd的选项为SO_REUSEADDR为1，表示允许创建端口号相同但IP不同的多个

//socket描述符
```

但是如果是用创建子进程的方法比较浪费资源，我们可以修改为创建线程的方法

四、sockaddr数据结构

IPv4 和 IPv6 的地址格式定义在“netinet/in.h”中，IPv4用sockaddr_in结构体表示，包括16位端口号和32位IP地址；IPv6用sockaddr_in6结构体表示，包括16位端口号、128位IP地址和一些控制字段。

UNIX Domain Socket的地址格式定义在sys/un.h中，用sockaddr_un结构体表示。各种socket地址结构体的开头都是相同的，前16位表?示整个结构体的长度(并不是所有UNIX的实现都有长度字段，如Linux就没有)，后16位表示地址类型。IPv4、IPv6和UNIX Domain Socket的地 址类型分别定义为常数AF_INET、AF_INET6、AF_UNIX。这样，只要取得某种sockaddr结构体的首地址，不需要知道具体是哪种类型的sockaddr结构体，就可以根据地址类型字段确定结构体中的内容。因此socket API可以接受各种类型的sockaddr结构体指针做参数，例如bind、accept、connect等函数，这些函数的参数应该设计成void

类型以便接受各种类型的指针，但是sock API的实现早于ANSI C标准化，那时还没有void 类型,因此这写函数的参数都用struct sockaddr*类型表示，在传参之前需要强制类型转换（在bind函数中就有用到）。

sockaddr_in中的成员struct in_addr sin_addr表示32位的IP地址。但是我们通常用点分十进制的字符串表示IP地址，以下函数可以在字符串表示和in_addr表示之间转换。

字符串转in_addr的函数:

这里写图片描述

```
#include <arpa/inet.h>

int inet_aton(const char *strptr, struct in_addr *addrptr);
in_addr_t inet_addr(const char *strptr);
int inet_pton(int family, const char *strptr, void *addrptr);
```


原文链接：https://blog.csdn.net/qq_33951180/article/details/68066634