sockaddr_in/sockaddr_un/sockaddr_in6
bind(sockaddr *addr----这个地址结构是为了实现接口统一) {
if (addr-> sin_ family == AF_INET) {ipv4地址结构的解析sockaddr. in}
else if (addr-> sin_family == AF_UNIX) {本地的地址结构解析sockaddr _un}
else.....
}

客户端程序中流程的特殊之处:
客户端通常不推荐用户自己绑定地址信息，而是让操作系统在发送数据的时候发现socket还没有绑定地址,然后自动选择一个合适的ip地址和端口进行绑定
1.如果不主动绑定，操作系统会选择-个合适的地址信息进行绑定(什么 地址就是合适的地址? --- 当前没有被使用的端口)
一个端口只能被一个进程占用， 若用户自己指定端口以及地址进行绑定有可能这个端口已经被使用了，则会绑定失败
让操作系统选择合适的端口信息，可以尽最大能力避免端口冲突的概率

对于客户端来说，其实并不关心使用什么源端地址将数据发送出去,只要能够发送数据并且接收数据就可以

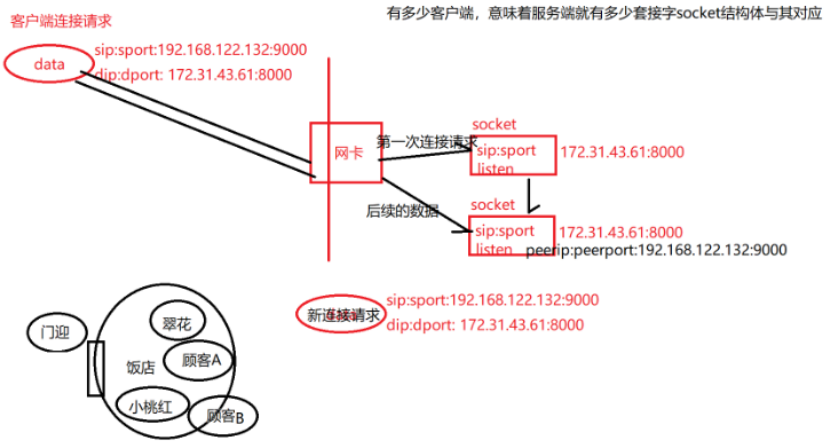
2.服务端可不可以也不主动绑定地址? --- 不可以的---
客户端所知道的服务端的地址信息，都是服务端告诉客户端的
一旦服务端不主动绑定地址，则会造成操作系统随意选择合适的地址进行绑定，服务端自己都不确定自己用了什么地址信息,如何告诉客户端
因此服务端通常必须主动绑定地址，并且不能随意改动

每个程序绑定的都是自己的网卡地址信息
客户端发送的对端地址信息一定是服务端绑定的地址信息
服务端绑定的地址信息，一定是自己主机上的网卡IP地址

tcp编程流程:面向连接,可靠传输，面向字节流

client
1.创建套接字
2.绑定地址信息
3.向服务端发起连接请求
server
1.创建套接字-创建socket结构体
2.为套接字绑定地址信息(告诉操作系统哪些数据放到我的socket缓冲区,发送数据使用哪个源端地址信息进行发送)
3.开始监听(tcp两端通信之前需要先建立连接--确保通信双方都具有收/发数据的能力;而建立连接的过程是操作系统完成的，用户不需要关心;开始监听就相当于告诉操作系统，可以开始接收客户端的连接请求了)
  1.服务的创建的套接字可以接收客户端的连接请求,
  2.当客户端连接请求到来之后，服务端会为这个客户端单独创建-个socketsocket结构体中描述(源端ip/源端port/对端ip/对端port/协议..) ;这些描述信息表示这个新的socket用于专门跟这个客户端进行通信
  服务端会为每一个客户端创建一 个socket实现与客户端通信

最早创建的套接字:就像门迎，永远只接收客户端的连接请求--并不与客户端进行后续的数据通信,
连接到来之后，创建新的套接字:就像-对-服务员与这个客户端进行数据通信后续的通信都是通过新的这个套接字完成的。




client

1.创建套接字

2.绑定地址信息(不推荐)

3.向服务端发起连接请求

4.收发数据

5.关闭套接字

server

1.创建套接字:在内核中创建socket结构体

2.绑定地址信息:通过socket描述源端地址信息

3.开始监听:告诉操作系统可以开始接收连接请求(tcp是面向连接,通信前先建立连接)

服务端接收新客户端连接请求，会为客户端创建一个新的socket, 这个套接字中既具有源端信息，也具有对端信息;这个新创建的套接字用于与这个客户端进行通信

最早的套接字:监听套接字---只用于接收新客户端连接请求

新创建套接字:通信套接字---用于后续与客户端进行数据通信

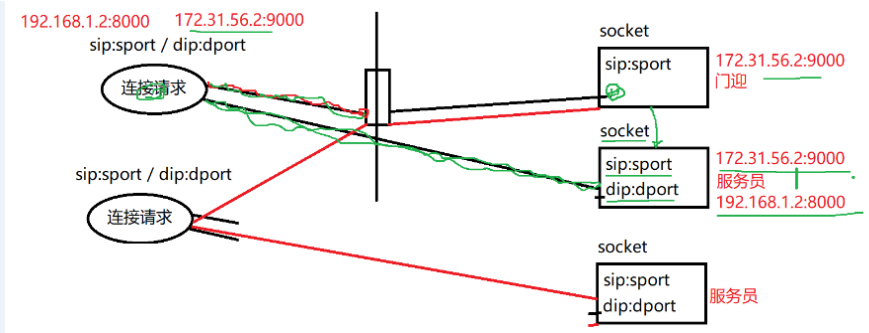
4.服务端程序中获取这个新建套接字的操作句柄描述符

因为后续与这个客户端的通信都是通过这个操作句柄完成的

最早的套接字描述符操作句柄只是用于建立连接，获取新连接的

5.收发数据:因为tcp的套接字socket中既描述了源端， 也描述了对端，因此收发数据不需要在获取/指定对端的地址信息了并且连接建立以后，谁先发送数据都可以

6.关闭套接字:释放资源



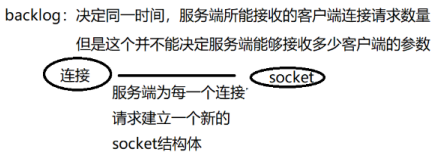

tcp编程socket接口介绍:

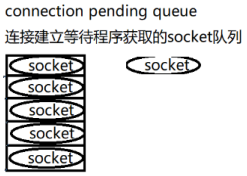
1.创建套接字: int socket(int domain, int type, int protocol) (AF_ INET, SOCK_ STREAM--流式套接字，IPPROTO_TCP);

2.绑定地址信息: int bind(int sockfd, struct sockaddr *addr, socklen. _t len); struct sockaddr_in;

3.服务端开始监听: int listen(int sockfd, int backlog); ---告诉操作系统开始接收连接请求

backlog:决定同一时间，服务端所能接收的客户端连接请求数量



SYN泛洪攻击:恶意主机不断的向服务端主机发送大量的连接请求

若服务端为每一个连接请求建立socket,则会瞬间资源耗尽,服务器崩溃

因此服务端有一个connection pending queue;存放为连接请求新建的socket节点; backlog参数决定了 这个队列的最大节点数量;若这个队列放满了，若还有新连接请求到来，则将这个后续请求丢弃掉

4.获取新建socket的操作句柄:从内核指定socket的pending queue中取出一个socket, 返回操作句柄

int accept(int sockfd, struct sockaddr *addr, socklen_t *len)

sockfd:监听套接字--指定要获取哪个pending queue中的套接字

addr:获取-个套接字, 这个套接字与指定的客户端进行通信，通过addr获取这个客户端的地址信息

len:输入输出型参数--指定地址信息想要的长度以及返回实际的地址长度

返回值:成功则返回新获取的套接字的描述符- -操作句柄;失败返回-1

5.通过新获取的套接字操作句柄(accept返回的描述符)与指定客户端进行通信

接收数据: ssize_ t recv(int sockfd, char *buf, int len, int flag);返回值:成功返回实际读取的数据长度;连接断开返回0;读取失败返回-1

还记得管道读取数据:所有写端被关闭,则read会返回0; tcp连接断开也是写端被关闭的-种体现;

发送数据: ssize_ t send(int sockfd, char *data, int len, int flag);返回值:成功返回实际发送的数据长度;失败返回-1;若连接断开触发异常

注意:连接若是断开了，recv会返回0; send会触发异常导致进程退出

6.关闭套接字:释放资源

int close(int fd)

7.客户端向服务端发送连接请求

int connect(int sockfd, int sockaddr *addr, socklen_ t len);

sockfd:客户端套接字---若还未绑定地址，则操作系统会选择合适的源端地址进行绑定

addr:服务端地址信息-- struct sockaddr_ in;这个地址信息经过connect之后也会描述到socket中

len: 地址信息长度

while(1) {

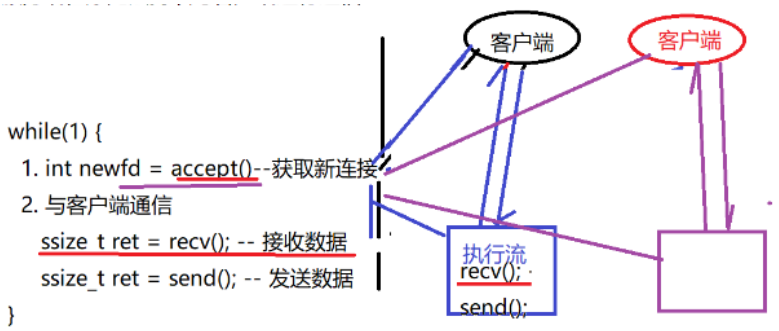
1.获取新建连接--- accept接口是一个阻塞函数--若没有新连接则阻塞等待

2.通过新建连接与客户端进行通信

}

1.服务端程序阻塞位置: accept获取新连接处

2.服务都安程序阻塞位置:与客户端通信处recv/send




因为我们不知道什么时候数据到来，也不知道什么时候新连接到来，因此流程只能写成先获取新连接,然后接收数据，相应数据，

然而这几个功能都有可能会导致程序流程阻塞;

1. accept这个函数是-个阻塞函数:功能是获取新连接，如果当前没有新连接,则阻塞等待直到有新连接

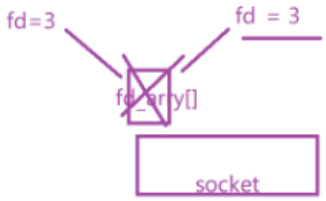
2. recv/send默认也是阻塞函数:接收缓冲区没有数据则recv阻塞 /发送缓冲区数据满了则send阻塞

你给幼儿园小朋友发糖--等待小朋友到来后给他发糖，看着小朋友吃糖，吃完后等待下一个小朋友
1.你等到了小朋友a,给他发了一个糖，看他吃完,然后开始等待下一个小朋友;然而这时候若小朋友a说还想吃糖
2.你等到了小朋友a,给他发了-一个糖，但是下朋友a现在不吃，因此你一直盯着他;这期间小朋友b过来
因为当前tcp服务端流程是固定的(因为我们无法获知什么时候有请求到来,然后调用accept, 什么时候有数据到来然后调用recv)，获取新连接，以及与客户端通信;
然而这两个功能都有可能阻塞，导致流程无法继续推进

解决方案:要防止流程阻塞,就要保证一个执行流只负责一个功能

一个执行流只管获取新连接，当新连接获取成功，然后创建新的执行流与客户端进行通信

多进程:
1.父进程创建子进程,数据独有，各自有一份cli_sock;然而子进程通过cli_sock通信，但是父进程不需要，因此父进程关闭自己的cli_sock
2.父进程要等待子进程退出，避免产生僵尸进程;为了 父进程只负责获取新连接，因此对于SIGCHLD信号 自定义处理回调等待
多线程:
1.主线程获取到新连接然后创建新线程与客户端进行通信，但是需要将套接字描述符传入线程执行函数中
2.但是传输这个描述符的时候，不能使用局部变量的地址传递(局部变量的空间在循环完毕就会被释放)，可以传描述符的值，也可以传入new的对象
3. C+ +中对于类型强转，将数据值当作指针传递有很多限制，我们想办法去克服就可以了
4.主线程中虽然不适用cli_sock,但是不能关闭cli_ sock,因为线程间共享资源，一个线程释放,另一个线程也就没法使用了




cli_ sock是一个TcpSocket类实例化的对象
_sock_ fd-是cli_ sock中的一个成员变量---用于保存创建套接字返回的文件描述符的值
cli_ sock.Recv(buf) 接收数据，Recv内部通过cli_ sock的sockfd接收数据的

连接断开在发送端与接收端上的表现:
1.接收端:连接断开，则recv返回0; 反之若recv返回0,表示的就是连接断开(套接字写端被关闭--双工通信)
2.发送端:连接断开，则send触发异常-SIGPIPE,导致进程退出

管道:
管道所有读端关闭，继续写入就会触发SIGPIPE异常
管道所有写端关闭，继续读取就会返回0

1. udp编程流程及接口:
客户端: socket / sendto / recvfrom / close
服务端: socket / bind / recvfrom / sendto / close
2. tcp编程流程及接口:
客户端: socket / connect / send / recv / close
服务端: socket / bind / listen / accept / recv / send / close
3.字节序转换接口:
htons / ntohs / htonl / ntohl / inet addr / inet ntoa / inet _ntop / inet pton

4.多执行流tcp服务端流程
多执行流思想
多进程流程: .
多线程流程:
线程池的使用:

# 复习:

udp编程流程:
1.创建套接字:在内核创建socket结构体并且描述通信所需信息(地址域，套接字传输类型，协议类型)
2.为套接字绑定地址信息:在socket中描述源端的地址信息(源端IP地址/源端端口信息)
3.收发数据
  客户端首先发送数据:将数据拷贝到内核态的socket发送缓冲区中
  服务端首先接收数据:从内核态socket接收缓冲区中拷贝出数据
4.关闭套接字:释放资源
udp编程使用的socket接口介绍:
1. int socket(int domain, int type, int protocol)
2. int bind(int sockfd, struct sockaddr *addr, socklen t len); struct sockaddr _in{ sin_ family; sin_ port; sin. addr.s_ addr)};
3. ssize_t sendto(int sockfd, char *data, int len, int flag, struct sockaddr *addr, socklen _t addrlen);
4. ssize _t recvfrom(int sockfd, char *buf, int len, int flag, struct sockaddr *addr, socklen_t *addrlen);
5. int close(int fd);

字节序转换接口:
1. uint32_ t htonl(uint32_ t) / uint32_ t ntoh(uint32_ t)
2. uint16. _t htons(uint16_ t) / uint16_ t ntohs(uint16_ t)
2. inet pton(int domain, char *src, void *dst); / inet_ntop(int domain, void *src, char *dst, int len);
3. in_addr _t inet_ addr(char *ip) / char *inet ntoa(struct in, addr);
inet_ pton(int domain, char *src, void *dst) ---- 将网络字节序的IP地址转换位网络字节序的整数IP地址IPV4/IPV6/..
sockaddr_ in addr; char buf[32];
例如IPV4地址的转换: inet_ pton(AF_ INET, *192.168.122.132", &addr.sin_addr.s_ addr)
inet_ ntop(AF_ INET, &addr.sin_ addr, buf, 32);将网络字节序数据转换成为字符串IP地址放到buf中

tcp通信流程:面向连接，可靠传输，面向字节流
tcp客户端与服务端流程:
客户端:创建套接字,描述地址信息，发起连接请求,连接建立成功，收发数据，关闭
服务端:创建套接字,描述地址信息，开始监听，接收连接请求，新建套接字，获取新建套接字描述符,通过这个描述符与客户端通信，关闭

tcp通信编程:
**tcp通信编程流程**
client
1.创建套接字
2.绑定地址信息(不推荐)
3.向服务端发起连接请求
4.与服务端进行通信
  1.发送数据
  2.接收数据
5.关闭套接字
server

1.创建套接字-在内核中创建socket结构体
2.为套接字绑定地址信息在socket中描述源端地址信息
3.开始监听告诉操作系统可以开始接收以及处理客户端的连接请求
  1.客户端连接请求到来:服务端创建新的套接字socket
  2.将这个新建的socket放到内核中创建的一个pending queue
4.获取连接建立完成的新建套接字
  1.从pending queue中取出一个socket
5.通过获取的新建套接字与客户端进行数据通信
  1.发送数据
  2.接收数据
6.关闭套接字

**tcp通信编程接口介绍:**
1.创建套接字
int socket(AF_ INET, SOCK_ STREAM, IPPROTO_TCP);
2.为套接字绑定地址信息
struct sockaddr_in{sin_ family=AF_INET; sin_port=htons(); sin_ addr.s _addr=inet addr()};
int bind(sockfd, &addr, len);
3.开始监听
int listen(int sockfd, int backlog);
sockfd:监听套接字-- -获取客户端连接请求的套接字
backlog:决定了内核中-个连接待处理队列中的最大节点数量, 决定了服务端同时能够接收多少个客户端连接请求

服务端同一时间的并发连接数
tcp是面向连接的，服务端会为每一个客户端的连接请求创建一个新的socket用于通信

万一有网络攻击: SYN泛洪攻击- -不断的向服务端发送大量的连接请求，若服务端不断的为每个新的连接请求创建套接字，则有可能瞬间资源耗尽服务器崩溃。
实际上服务端为客户端连接请求创建新的套接字也是有前提的--前提就是内核中这个监听套接字对应有一个连接待处理队列，这个队列中放的就是这些socket,如果放满了，则不再继续接收客户端的连接请求--- backlog参数决定了这个队列的最大节点数量
4.服务端程序中获取连接建立完成的socket: 从这个连接待处理队列中取出一个socket, 向程序返回-一个操作句柄
int accept(int sockfd--监听套接字, struct sockaddr *addr- -接收客户端的地址信息，socklen. t len -地址信息长度);
返回值:返回新获取的已完成连接的套接字描述符--后续与客户端通信使用的就是这个描述符
5.收发数据
ssize_ t recv(int sockfd--accept返回的新建套接字描述符，char *buf, int len, int flag);
ssize_ t send(int sockfd, char *data, int len, int flag);
6.关闭套接字: int close(int fd);
7.向服务端发起连接请求
int connect(int sockfd --- 客户端套接字描述符struct sockaddr *addr--服务端地址信息, socklen_t len--地址信息长度)

字节序转换接口:
uint16_ t htons(uint16_ t port); 2个字节的数据
uint16_ t ntohs(uint16_ t port);主机字节序与网络字节序

uint32_ t htonl(uint32 _t port); 4个字节的数据
uint32_ t ntohl(uint32_ _t port);主机字节序与网络字节序

in_addr _t inet_addr(char *ip);点分+进制字符串IP地址转换为网络字节序整数IP地址

char *inet _ntoa(struct in_ addr addr);网络字节序整数IP地址转换为点分十进制字符串

domain-地址域- AF_ INET-IPV4 AF_ INET6-IPV6
int inet_pton(int domain, char *ip, void *ip);字符串地址转换网络字节序整数地址
int inet_ntop(int domain, void *ip, char tip, int len);网络字节序整数IP地址转换为点分十进制字符串

```
struct sockaddr. in {
   sa _family_t sin_ family; -- 这个成员用于表示地址域----这是一个什么样的地址结构 - AF_INET
   in_port _t sin_ port; --这个成员是一个网络字节序的端口信息
   struct in_ addr {
      in_ addr_ t  s_addr; --- 这个成员是一个网络字节序的IP地址信息
   }sin_ addr;            sin_ addr.s_ addr
};
```

struct sockaddr_un - AF_UNIX/ struct sockaddr_in6- AF_INET6

当前的tcp服务端程序无法持续与客户端进行通信: