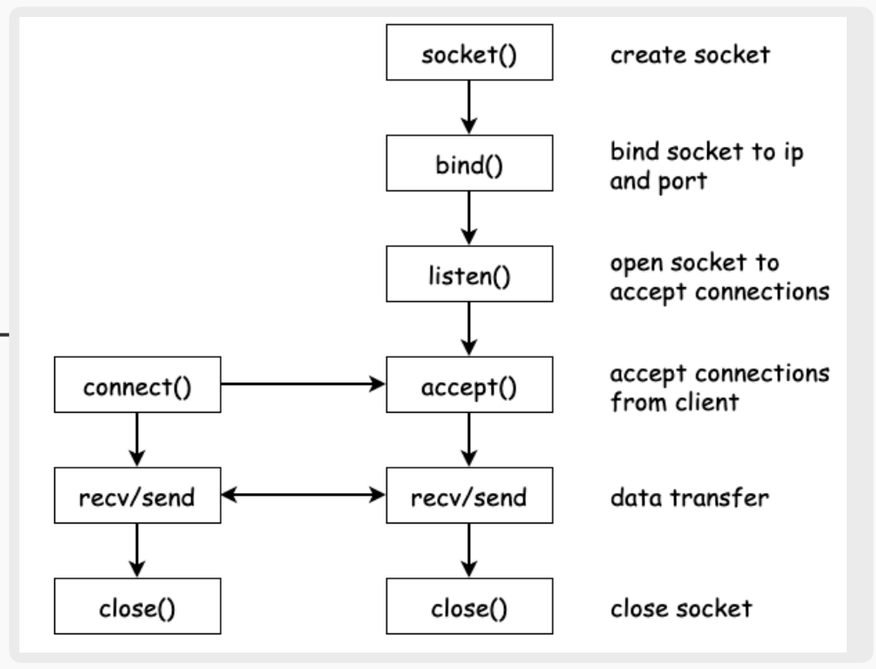
socket

基本流程



socket()

- **int socket(int domain, int type, int protocol);** — 创建一个 socket 并返回其描述符，该描述符和文件描述符没什么区别。如果我们的进程没有 open 其它文件的话，那么调用该函数应该会返回 3
- 形参
  - domain — 协议族或者是协议域，通常来说会有 3 个选择
    - AF_UNIX — Unix 域协议
    - AF_INET — IPv4 协议
    - AF_INET6 — IPv6 协议
    - 通常来说 AF_INET 使用更多
  - type — socket 类型，也就是我们熟知的 TCP 协议、UDP 协议
    - SOCK_STREAM — 流式数据协议，即 TCP
    - SOCK_DGRAM — 数据报数据协议，即 UDP
  - protocol — 通常为 0，表示由内核决定最合适的 protocol 值，当然这通常与前两个参数有关
- Example — int listenfd = socket(AF_INET, SOCK_STREAM, 0);

bind()

- **int bind(int sockfd, const struct sockaddr *addr, socketlen_t addrlen);** — bind() 系统调用将创建的 socket 绑定到一个地址和一个端口上
- 形参
  - sockfd — socket() 返回的套接字描述符
  - addr
    - 指向 socket 地址结构的一个指针，要么是 IPv4 地址，要么是 IPv6 地址

```
struct sockaddr {
    sa_family_t sin_family;
    char        sa_data[14];
};
```

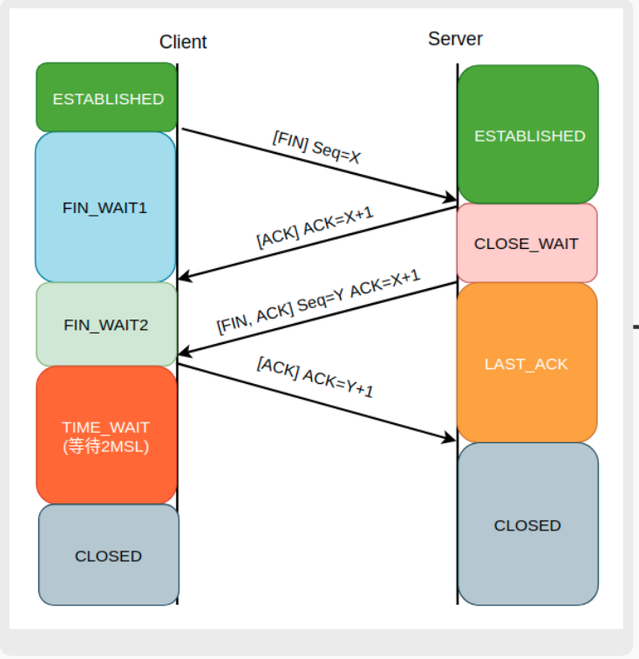
sockaddr 有一个比较蛋疼的问题就是把 IP 和 Port 混在了一起，所以大部分情况下会使用 sockaddr_in

- sockaddr_in

```
struct sockaddr_in {
    sa_family_t     sin_family;
    uint16_t        sin_port;
    struct in_addr  sin_addr;
    char            sin_zero[8];
};
```

```
struct sockaddr_in addr;
memset(&addr, 0, sizeof(addr));          // 不要使用 bzero
addr.sin_family = AF_INET;               // IPv4 网络协议
addr.sin_port = htons(9600);             // 将端口转化成网络字节序
addr.sin_addr.s_addr = INADDR_ANY;       // 绑定本机任意 IP 地址，通常为 0.0.0.0
```

- TIME_WAIT



改图是 TCP 连接拆除时的状态转移示意图，其它的我们可以不管，主要关注于 TIME_WAIT 状态

在 TCP 四次挥手过程中，第一个 FIN 和最后一个 ACK 包均由主动断开方发送，并且，TIME_WAIT 状态也只会出现在主动断开方

TIME_WAIT 状态的作用就是为了确保连接能够被正常的拆除，因为最后一个 ACK 包可能会丢失，那么此时需要主动断开方重传该 ACK 包。并且，确保老的重复的报文在网络中过期失效

TIME_WAIT 通常会等待 2MSL 的时间，MSL为报文的最大生存时间，Linux 系统下默认为 60s，所以 TIME_WAIT 默认持续 120s 的时间，也就是说，如果 server 端重启的话，仍然可能会存在原来的 IP + Port 的连接存在，那么重启后再次进行 bind 将失败，会提示 address already in used

- SO_REUSEADDR

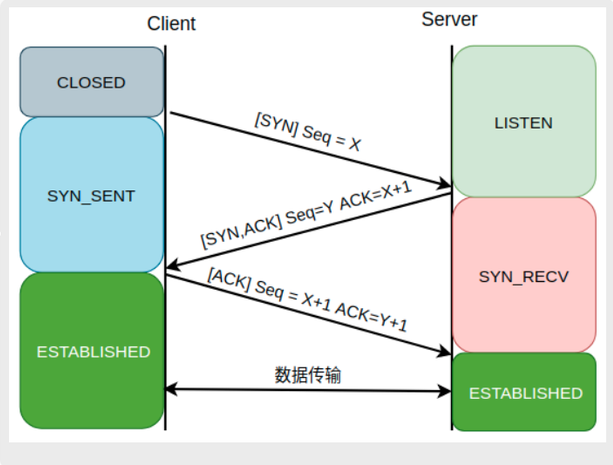
因此，我们创建的 socket 在 bind 地址之前，应该为其添加 SO_REUSEADDR 选项，以重用端口

该选项可以告诉内核，如果当前端口正在被使用，但是 TCP 状态处于 TIME_WAIT 的话，允许应用程序重用该端口。但是如果端口正在被使用且 TCP 状态为其它状态时，仍然返回错误，表明端口已经被使用

```
int  reuse_addr = 1;
setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, (const void *) &reuse_addr, sizeof(int));
```

listen()

- **int listen(int sockfd, int backlog);** — listen() 系统调用将文件描述符 sockfd 所引用的 socket 标记为被动，表示开始监听某一个端口



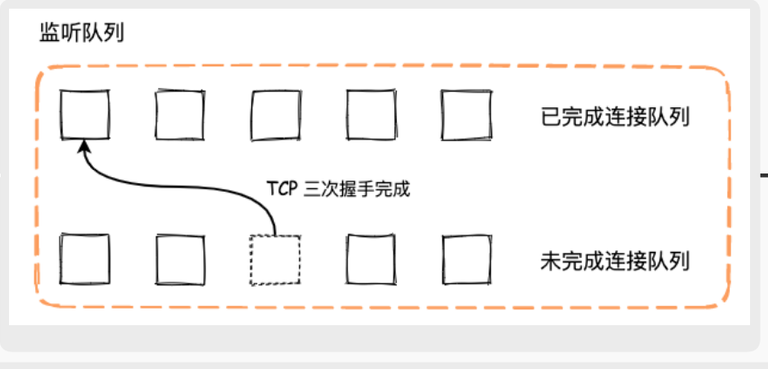
这是 TCP 三次握手的示意图。通常而言，都是由客户端主动发起连接

当服务端收到了 SYN 包以后，那么连接状态就会由 LISTEN 转变为 SYN_RECV，这个状态表示服务端三次握手还没有完成，所以也被称之为"半连接"

只有当再次收到客户端发来的 ACK 确认包以后，该连接才会有 SYN_RECV 状态转移至 ESTABLISHED 状态，表明连接已完全建立，双方均可收发数据

- 监听队列

那么，对于服务端来说，不管是半连接还是已建立的连接，都需要保存在内存中。内核使用两个队列来进行存储，分别为未完成连接队列，和已完成连接队列



当半连接收到了 ACK 确认包以后，内核将该连接从未完成连接队列移入至已完成连接队列中

在以往，backlog 参数的含义表示已完成连接队列大小 + 未完成连接队列的大小的最大值

但是后来改参数得到了修正，仅表示已完成连接队列的最大长度

将 backlog 的含义定义为 "已完成连接队列的最大长度" 可以在一定程度上防止 SYN 泛洪攻击，但是不能从根本上解决这个问题

accept()

- **int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen)**
  - accept() 系统调用将返回一个全新的 socket 文件描述符，用于表示与之通信的具体客户端
  - 客户端 IP 地址等信息通过 addr 以及 addrlen 参数写入

accept() 的本质就是从已完成连接队列的头部取出一个连接返回给应用程序，那么如果已完成连接队列中没有任何数据，并且 socket 被设置为阻塞的话，accept() 调用将一直阻塞

所以，accept() 返回一个新的 socket 描述符也就不那么令人吃惊了。通过 socket() 系统调用所创建的 socket 描述符主要的目的就是为了监听端口，而通过 accept() 获得的 socket 描述符才是和客户端通信的媒介

- accept4()

从内核 2.6.28 开始，Linux 内核支持使用非标准系统调用 accept4() 来接收新的连接，使用该函数可能会有移植性的问题，程序需要考虑到这一点

int accept4(int sockfd, struct sockaddr *addr, socklen_t *addrlen, int flags)

在使用 accept4() 时，我们可以直接指定 SOCK_NONBLOCK 参数，使得返回的 socket 后续操作为非阻塞，无需再使用 fcntl() 设置了