# linux网络编程-----TCP连接及相关问题

c/s模型在建立连接时的流程如下

```
//服务器端
int sockfd = socket(AF_INET, SOCK_STREAM, 0);

struct sockaddr_in servaddr;
bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons(8080);
servaddr.sin_addr.s_addr = INADDR_ANY;

bind(sockfd, (struct sockaddr*)&servaddr, sizeof(servaddr));
listen(sockfd, 10);

struct sockaddr_in addr;
socklen_t len = sizeof(addr);
int fd = accept(sockfd, (struct sockaddr*)&addr, &len);

/* ... */

close(fd);
close(sockfd);12345678910111213141516171819 20
```

```
//客户端
int sockfd = socket(AF_INET, SOCK_STREAM, 0);

struct sockaddr_in servaddr;
bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons(8080);
inet_aton("127.0.0.1", &servaddr.sin_addr);

connect(sockfd, (struct sockaddr*)&servaddr, sizeof(servaddr);

/* ... */

close(sockfd);1234567891011121314
```

在客户端connect，服务器accept以及二者的close过程中，代表着TCP连接的建立和终止，也就是常说的TCP三路握手和四路挥手

# 三路握手

建立一个TCP连接时会发生如下情形

1. 服务器端必须准备好接受客户端的连接请求，通常是通过调用socket，bind，listen这三个函数来完成的，称之为被动打开
2. 客户端通过调用connect发起主动打开，主动连接到服务器端，这个过程中客户端发送一个SYN（同步）分节到服务器端，它告诉服务器将在连接中发送的数据的初始序列号（可以理解为数据的起始号码，只有号码在这个序列号之后的数据才会被认为是当前客户端发送的数据），**此时客户端状态为SYN_SENT**
3. 服务器必须确认（ACK）客户的SYN，将客户端的初始序列号记录下来。同时自己也需要发送一个SYN分节，它告诉客户端在连接中发送的数据的初始序列号（只有号码在这个序列号之后的数据才会被认为是当前服务器端发送的数据）。服务器端在一个分节中同时发送SYN和ACK。**此时服务器端状态为SYN_RCVD**
4. 客户端必须确认服务器端的SYN，将服务器端的初始序列号记录下来，发送确认ACK到服务器端，同时**connect函数返回，客户端状态变为ESTABLISHED**
5. 服务器端接受到来自客户端的确认（ACK），**accept函数返回，服务器端状态变为ESTABLISHED，建立TCP连接**

至此完成TCP的三路握手

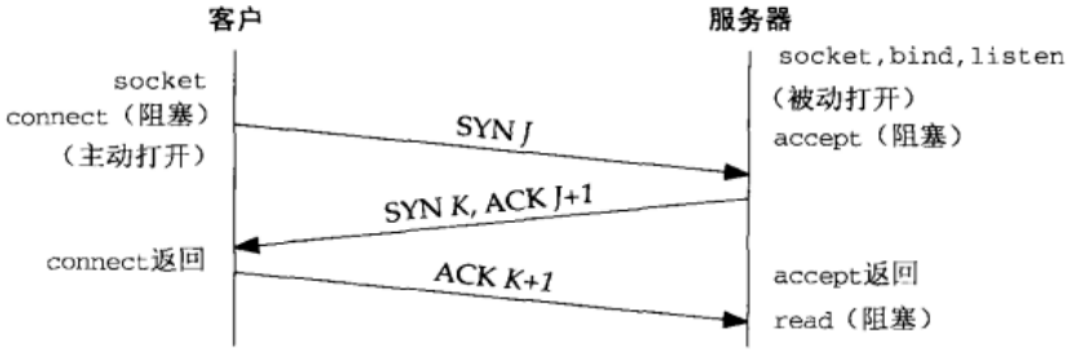
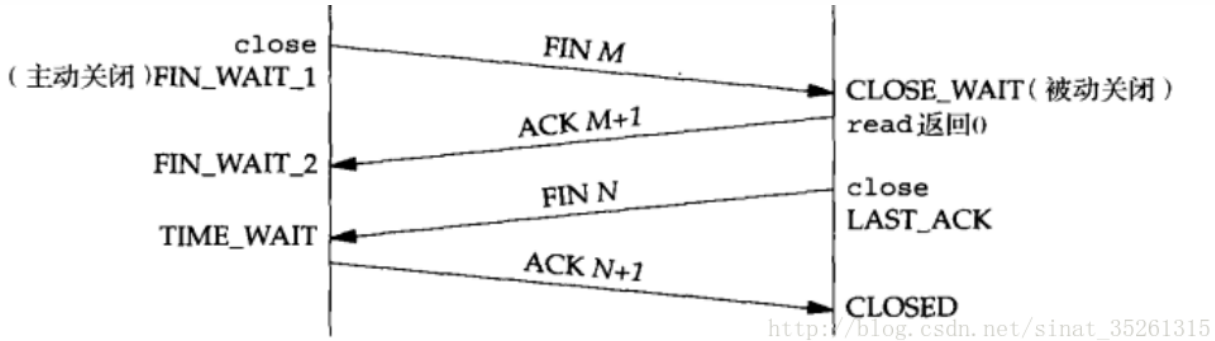



图2-2　TCP的三路握手



# 四路挥手

TCP建立连接需要3个分节，而终止连接却需要4个分节 **以下以客户端先调用close为例** \1. 客户端首先调用close函数关闭套接字，执行为主动关闭，该端的TCP发送一个FIN分节到服务器端，**表示数据发送完毕，客户端进入FIN_WAIT_1状态** \2. 接收到这个FIN的服务器端执行被动关闭。这个FIN由该端的TCP确认，发送确认分节（ACK）到客户端，FIN的接受也作为一个文件结束符传递给服务器端应用进程，也就是说如果服务器端进程调用recv/read函数，会读到相当于文件结束符的FIN，从而表示客户端已经发送完数据，执行了close。**至此服务器端进入CLOSE_WAIT状态，意思是等待调用close** \3. 客户端接收到来自服务器端的确认分节（ACK），**进入FIN_WAIT_2状态** \4. 一段时间之后（通常是recv/read到FIN后），服务器端执行close函数关闭套接字，这导致服务器端TCP也发送一个FIN到客户端，**此时服务器端进入LAST_ACK状态** \5. 客户端接收到服务器端发送的FIN并确认这个FIN，同时发送确认（ACK）到服务器端，**同时进入TIME_WAIT状态** \6. 服务器端接受到来自客户端的确认（ACK），**进入CLOSED状态，至此TCP连接终止**




# 相关问题

**为什么是三路握手，不是二路或者四路?**

1. 因为TCP连接具有高效，稳定等特性，而三路握手正好可以满足TCP的这种特性。
2. 如果是四路握手，即服务器端的SYN和ACK会分开发送。也就是客户端发送SYN到服务器端，服务器端记录客户端的数据的初始序列号，发送确认分节ACK给客户端，然后发送自己的数据初始序列号SYN到客户端。显然这两步可以合并在一个分节中发送，实现高效性
3. 如果是二路握手，即客户端收到服务器端的SYN和ACK后不发送确认（ACK）到服务器端，这就会出现如下问题：客户端知道服务器端能够收到自己的数据，而服务器端不知道客户端能不能收到自己的数据，无法实现稳定性
4. 综上，TCP的连接采用三路握手

---

**服务器端accept返回的套接字占用的端口和监听套接字监听的端口是否相同?** 相同，服务器端已建立的TCP连接套接字使用和监听套接字相同的端口。 直观的考虑，http服务器监听80端口的连接，服务器不可能给成千上万个访问分配不同的端口，所以通过accept返回的套接字使用的也是监听套接字的端口。 套接字由<源ip，源端口，目的ip，目的端口，协议>这个五元组唯一确定的，所以仅仅端口相同不能说明TCP连接是相同的

---

**套接字的结构是什么样的，为什么send时只传入一个套接字就知道往什么地址端口发送数据?** 套接字表面上使用int类型，其实内部由<源ip，源端口，目的ip，目的端口，协议>这个五元组唯一确定，在connect返回时，内核将套接字的五元组进行赋值，记录目的ip和目的端口，所以在之后的recv/send等io操作时只需要传入一个套接字遍知道究竟从什么地址，端口接受数据，往什么地址，端口发送数据了。 accept同理

---

**客户端套接字的端口是如何确定的，为什么没有手动绑定（使用bind函数）?** 在tcp连接中，客户端套接字的端口是由内核随机分配的，因为客户端不是客户端，需要知道具体端口供其他进程连接，所以也没必要使用bind绑定地址和端口，只需要手动connect到服务器的地址和端口即可

---

**如何理解主动关闭的一方会进入TIME_WAIT状态?** 在TCP连接终止时，主动关闭（首先调用close）的一方会进入这个状态，这个状态的持续时间是最长分节生命期的两倍，有时候称之为2MSL，有些实现是30s，有些则是2分钟不等。 TIME_WAIT状态有两个存在的理由

1. 可靠地实现TCP全双工连接的终止
2. 允许老的重复分节在网络中消逝

**第一个理由**可以假设四路挥手的最后一个确认分节（ACK）丢失，服务器端（仍然假设客户端主动关闭连接）在发送FIN一段时间后仍然没有收到来自客户端的确认（ACK），这就导致了服务器端认为客户端没有接受到自己的FIN分节，从而重新发送FIN分节到客户端。 因为客户端此时处于TIME_WAIT状态，仍然保留着TCP连接时的双方信息，所以收到FIN后也重新发送确认（ACK）到服务器端，从而确保服务器端可以收到ACK正常关闭。 如果客户端不进入TIME_WAIT状态而直接关闭，那么当最后的ACK分节丢失，服务器端重新发送FIN给客户端后，客户端已经没有了双方连接的信息，也就是说不能识别来自服务器端的这个FIN，就会发送一个RST（另一种类型的TCP分节），服务器端便认为TCP连接出现了错误 **第二个理由**可以假设在客户端12.106.32.254的1500端口和服务器端206.168.112.219的21端口之间有一个TCP连接被建立，客户端执行主动关闭，而后客户端又重新发起连接 如果客户端关闭后不进入TIME_WAIT状态，那么客户端内核会分配第一个可用的端口即1500（因为刚关闭，所以可用），这就导致了两次的TCP连接的客户端地址和端口是一样的，如果此时发送一段数据给服务器，服务器会误认为这个数据是第一次的连接还没有接受完的数据，也就是说会把本次连接误认为是刚开始的连接 如果客户端关闭后进入TIME_WAIT状态，那么内核不会分配一个处于TIME_WAIT状态的（ip，端口）给进程使用，就解决了上述的问题，又因为2msl足以让先前的数据报文消逝，所以2msl足矣

---

**服务器端accept完客户端请求后可不可以关闭监听套接字？** 可以，程序可以在任何时候关闭监听套接字，只是在关闭之后不能够再使用accept接受客户端请求。关闭监听套接字后不妨碍已经建立的tcp连接。另外，四路挥手是对于tcp连接而言的，对监听套接字使用close会立即关闭它，不需要四路挥手。

---

**传统iso协议和tcp/ip协议的具体内容？** iso七层模型由应用层，表示层，会话层，传输层，网络层，数据链路层，物理层构成 tcp/ip四层模型由应用层，传输层，网络层，数据链路层构成

- 应用层：各种应用程序协议，HTTP，FTP，TALENT等
- 传输层：TCP数据流传输，面向连接的稳定，高效，可靠的传输协议，支持重传。UDP数据报传输，面向无连接的传输协议，速度快，但是不可靠，适用于实时通信
- 网络层：ip协议
- 数据链路层：负责数据传输

---

**socket是什么，怎么理解socket的应用？** tcp/ip协议中用户程序直接接触的是应用层，而对于底层的tcp协议栈，需要有一种接口供应用程序调用，socket正是这样一种接口，供用户程序与tcp协议栈交互，可以理解socket为tcp协议的一种抽象接口

[原文链接]