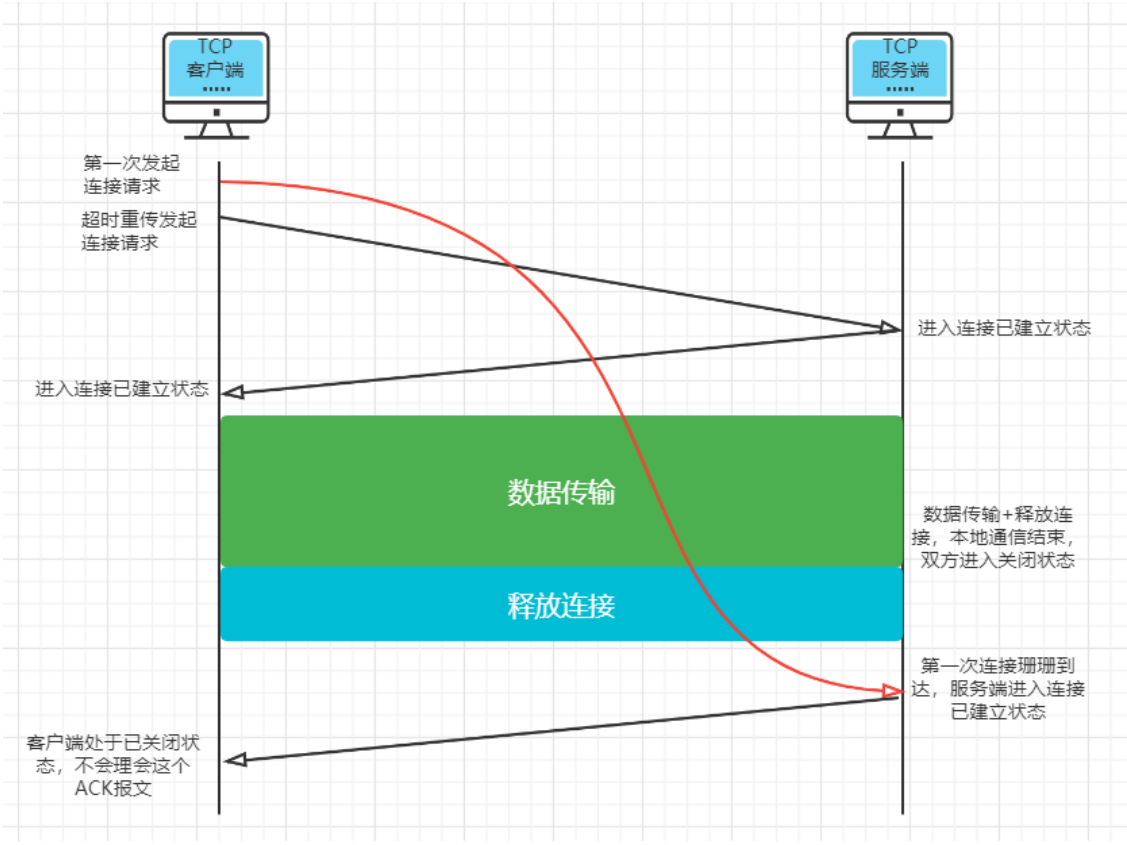
上一篇文章我们学习了TCP三次握手的详细过程，是不是还意犹未尽？下面我们来讨论几个有趣的问题，这也是面试中针对TCP三次握手喜欢讨论的问题。

一、最后一次ACK是否多余?

常见的面试问题：为什么是三次握手，而不是两次呢？最后一次ACK是否多余？是否可以简化为两报文握手?

坚定地告诉他：**不多余，不可简化为两报文握手！**

我们假设改用两报文握手，看看下面这种情况。



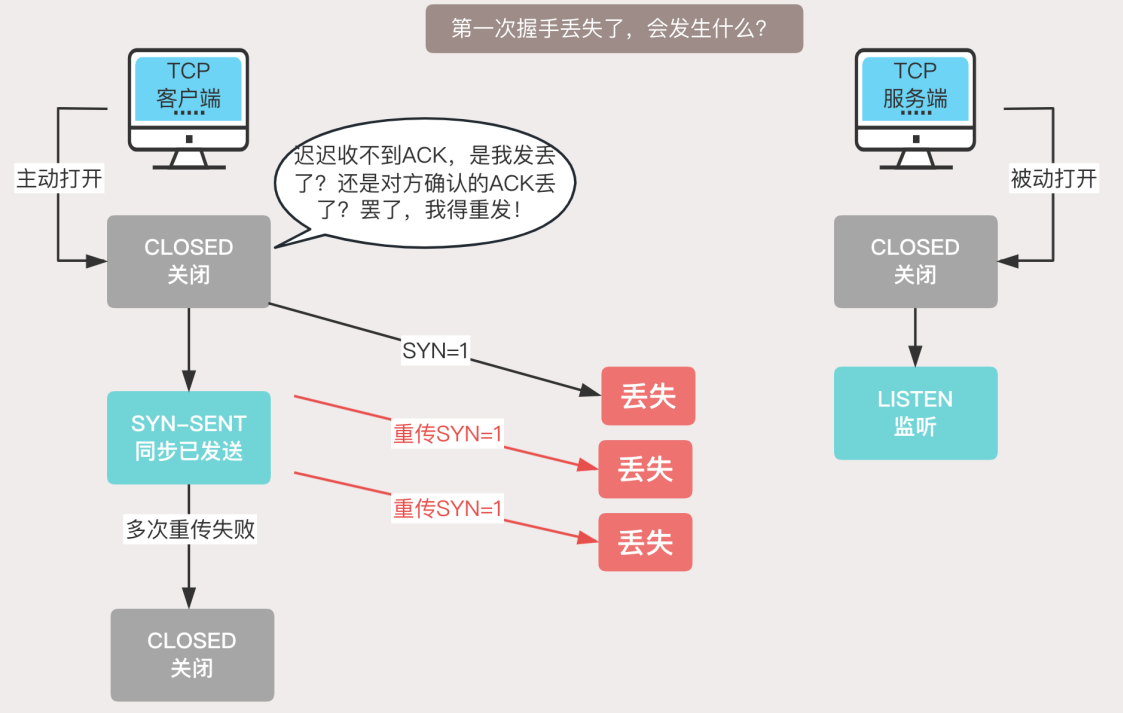
- ①、客户端进程发起一个TCP连接请求报文段，但此连接请求报文段由于某些原因在中间节点长时间滞留了，这必然会导致该报文段的超时重传；
- ②、假设重传的报文段被TCP服务端进程正常接收，TCP服务端进程给客户端发送一个TCP连接请求确认报文段，并进入连接已建立状态（请注意，我们这里是两报文握手，因此这一步服务端发送完连接请求确认报文后就会进入连接已建立状态）；
- ③、客户端进程收到TCP连接请求确认报文段后，进入连接已建立状态，不会再给服务端发送普通确认报文段；
- ④、此时，双方都已进入连接已建立状态，下面进行数据传输，在传输完毕后释放连接，双方又都进入关闭状态；

- ⑤、一段时间后，滞留在网络中的那个失效的TCP连接请求报文段到达了TCP服务端进程，此时服务端会误以为是客户端发来的新的TCP连接请求报文，针对此连接请求发送TCP连接请求确认报文，服务端进入连接已建立状态；
- ⑥、客户端收到此连接请求确认报文段，由于它自己并没有发起过新的TCP连接请求，并且处于关闭状态，因此不会理会该报文段；
- ⑦、不过此时服务端已经进入连接已建立状态，并且一直等待客户端发来数据，这将白白浪费服务端主机的很多资源。

综上所述，**采取三次握手而不是两次握手，原因是为了防止已失效的连接请求报文段突然又传送到了服务端，从而导致错误**。

那为什么不用四次握手呢？答案很简单，由于三次握手已经最少可靠连接建立，所以不需要使用更多的通信次数。

二、第一次握手丢失了，会发生什么？



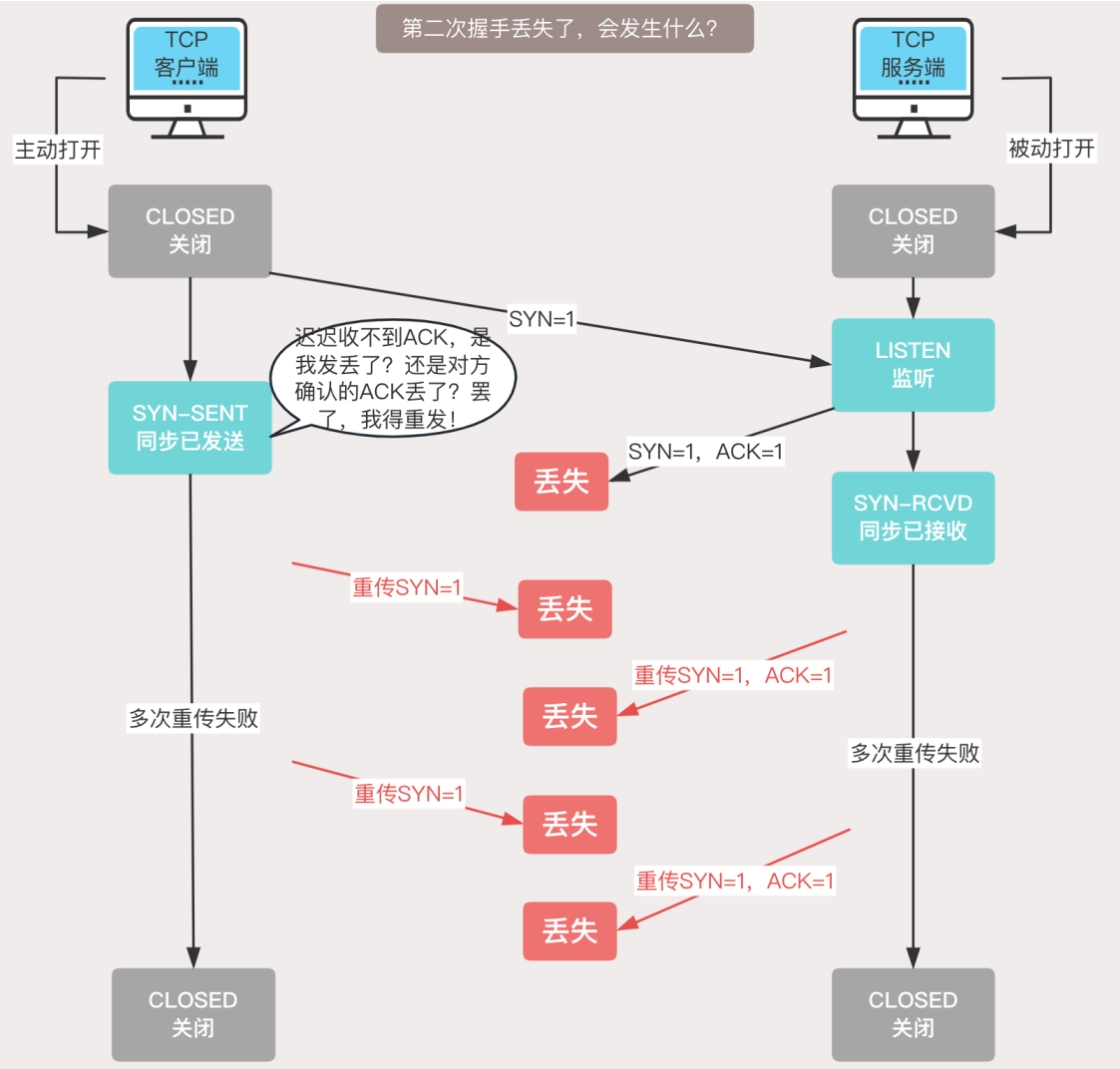
三次握手，客户端第一个发的是SYN=1报文，如果客户端迟迟收不到服务端的ACK报文，就会触发超时重传机制，重传的 SYN 报文的序列号都是一样的。

不同的操作系统实现的超时重传机制不一样，主流的重传机制是：第一次重传在1秒后，第二次重传在2秒后，第三次重传在4秒后，第四次重传在16秒后，第五次重传在32秒后，重传次数也各不相同，一般就是四五次。

如果达到重传次数上限后，客户端仍然收不到服务端ACK，客户端可以选择立即断开连接，也可以选择再等待一段时候后断开连接（比如等待上一次超时时间的2倍）。

这个重试机制对于软件开发有一定的启发，当我们自己在软件设计中遇到重传机制的设计，可以选择与TCP一样的指数级别增长的重试时间机制，而不是无脑快速重试。因为一般情况下，如果由于网络拥塞、服务端忙碌、路由器忙碌等原因导致丢包时，如果频繁重试，只会使得网络更加拥塞，事与愿违。

三、第二次握手丢失了，会发生什么？

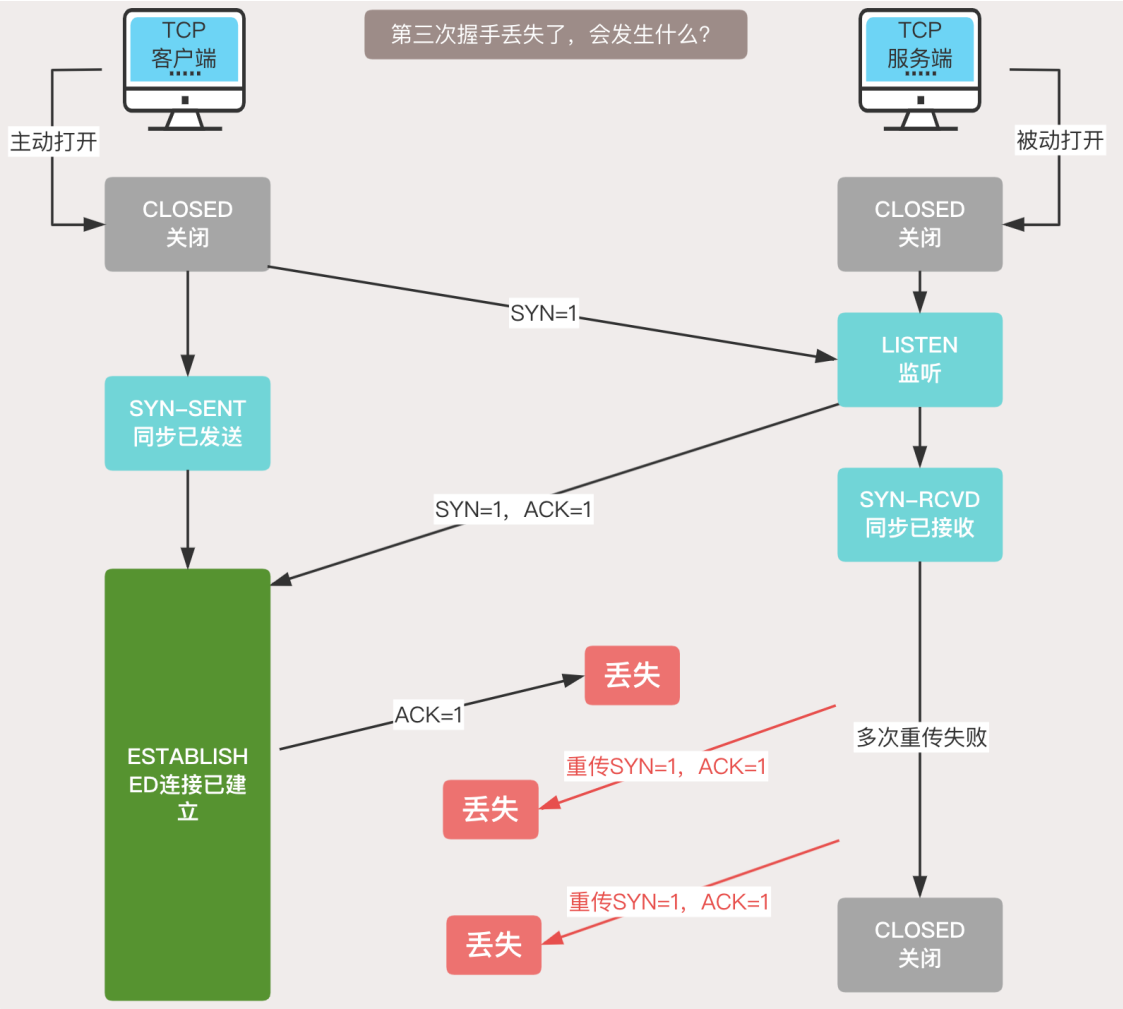


接下来，在服务端收到客户端第一次握手SYN报文后，需要回复SYN+ACK报文，本次报文的目的很显然，ACK是对第一次握手的确认，SYN是服务端发起向客户端建立连接的请求。所以我们说，TCP三次握手，是建立两个方向的连接。

如果第二次握手丢失，对于客户端来说，它会怀疑自己发出的第一个握手报文丢失，此时会触发客户端超时重传；而对于服务端来说，它会怀疑自己发出的第二个握手报文丢失，因为服务端需要客户端响应ACK，所以服务端这边也会触发超时重传。

所以说，如果第二次握手丢失，客户端和服务端都有可能触发超时重传机制，各自的重传次数可能不一样。

四、第三次握手丢失了，会发生什么?



最后，客户端收到了服务端SYN+ACK的第二次握手报文后，进入连接已建立状态，需要给服务端回复一个ACK报文，也就是第三次握手。

如果第三次握手报文ACK丢失了，怎么办呢?

此时服务端发现，自己的第二次握手报文迟迟得不到ACK响应，会触发服务端的超时重传SYN+ACK报文，直到收到客户端的第三次ACK握手报文，或者服务端达到最大重传次数而断开本次连接。

注意，**ACK 报文是不会有重传的，当 ACK 丢失了，就由对方重传对应的报文**。

五、SYN Flood攻击问题

我们将自己化身为一个顶级网络黑客，针对三次握手过程，想一想有没有漏洞可以攻击?

如果我们在极短时间，比如在0.001秒内，伪造大量不同IP地址的SYN报文发起三次握手连接，会发生什么?

服务端每收到一个SYN报文，会放入一个有限长度的队列，然后进入SYN_RCVD 状态，但服务端发送出去的 ACK + SYN 报文，无法得到未知 IP 主机的 ACK 应答，那么这个队列势必会被占满，那么服务端就无法接收来自新的客户端的握手连接请求了！

此外，在第二次握手时，服务端针对该连接会额外分配一些内存资源来存储缓存和变量，当遭遇SYN Flood攻击时，即便队列无限大，内存资源也是一个瓶颈，内存一旦爆满，服务也将不可用。

这就是SYN Flood攻击问题。

其实，在TCP建立连接过程中，连接有两种：一种是连接尚未完成，但是服务器已经接收到SYN报文。另一种是完成三次握手，但是连接没有被应用程序所接受。所以系统要为这两种情况维护两个队列。

以Linux系统来说，linux内核协议栈为一个TCP连接管理使用两个队列，一个是半连接队列，用来保存SYN_SENT和SYN_RECV状态请求。一个是全连接队列，用来保存处于established状态，但是该连接没有被应用层接受。

大概流程为：

- 当客户端发起一个连接请求的SYN报文段到达服务器端时，首先进入的是半连接队列，此时客户端连接状态为SYN_SENT；
- 服务器会响应客户端一个SYN报文段以及将客户端请求报文段中的序列号加1作为响应ACK返回，等待客户端进行确认并建立连接，此时服务端连接状态为SYN_RECV；
- 当服务器收到客户端确认报文段后，三次握手成功，此时连接状态为established，同时将该连接从半连接队列移除。

我们暂时只要知道，当发生SYN Flood攻击时，会直接将TCP 半连接队列打满，这样当 TCP 半连接队列满了，后续再在收到 SYN 报文就会丢弃，导致客户端无法和服务端建立连接。

那SYN Flood攻击问题如何避免呢?

针对恶意攻击，一方面我们可以在防火墙层面做限制单IP并发数，或者限制C类子网并发数，或者限制单位时间内连接数，总之就是结合实际业务做连接数的限制，在确保正常流量不受干扰的情况下，防止恶意流量产生的恶劣影响。

另外一方面就是通过调整各种参数做缓解，比如大促时，确实连接数较高，应对的方法比如有：

- 调大 netdev_max_backlog：当网卡接收数据包的速度大于内核处理的速度时，会有一个队列保存这些数据包，调大该队列的数值；
- 增大 TCP 半连接队列：既然半连接队列被打满，那就索性增大半连接队列咯；

- 开启 tcp_syncookies：开启 syncookies 功能就可以在不使用 SYN 半连接队列的情况下成功建立连接，相当于绕过了 SYN 半连接来建立连接。
- 减少 SYN+ACK 重传次数：当服务端受到 SYN 攻击时，就会有大量处于 SYN_REVC 状态的 TCP 连接，处于这个状态的 TCP 会重传 SYN+ACK ，当重传超过次数达到上限后，就会断开连接。那么针对 SYN 攻击的场景，我们可以减少 SYN-ACK 的重传次数，以加快处于 SYN_REVC 状态的 TCP 连接断开。
- ......

关于半连接队列和全连接队列，还有很多需要说明的，本篇文章仅仅作为一个引子，让读者朋友知道其概念存在，后续文章中我们详细再来探讨一番。

六、一台主机的TCP最大连接数是多少?

这也是一道极其经典的面试题，考验了我们对于TCP连接和操作系统的理解程度，下面我来尝试大体回答下该问题。

首先系统如何唯一标识一个TCP呢？这个问题之前说过，是通过一个四元组来唯一标识一个TCP连接，即{源IP、源PORT、目标IP、目标PORT}。

源IP	源PORT	目标IP	目标PORT

其次，我们需要明确一个事实，一般情况下server是会固定监听在某个端口上，而客户端一般是主动发起连接的一方，客户端端口哪来的呢？是让操作系统随机选择一个空闲的本地端口，可用的端口是1-65535，端口0有特殊含义，不能使用，这样可用端口最多只有65535个，但是实际上，操作系统也有可能有限制，可使用的端口少于这个范围，不过这个不重要，我们假设65535个端口都可以使用。

此时，一个极大的误区出现了：有人说最多只能创建 65535 个TCP连接。因为65535是指可用的端口总数，并不代表服务器同时只能接受65535个并发连接!

我们讨论这个问题时，先抛开其他干扰因素，只关注端口数量，需要从客户端和服务端两个角度来看待。

对于客户端来说，如果目标IP和目标端口是固定的一个，由于源IP一般也是只有一个，故而四元组的唯一性完全由源PORT决定，在这个情况下，一个client最大tcp连接数为65535。当然，这也是65535片面结论的由来。

但是如果不断变换目标 IP 和目标端口号，保证四元组不重复，那理论上支持建立的连接数量就非常非常多了，容易想象，IPV4是32位，目标端口也是16位，理论上支持建立2的（16+32+16）次方个连接了。

对于服务端来说，server通常固定在某个本地端口上监听，等待client的连接请求。即使server端有多个ip，本地监听端口也是独占的，因此server端tcp连接4元组中只有源IP和源PORT是可变的，因此最大tcp连接为客户端ip数×客户端port数，也就是说理论上服务端支持建立2的（32+16）次方个连接了。

可以看到，我们只要保证四元组唯一性，理论上是可以建立非常非常多TCP连接的。

但是事实是骨感的，在实际情况下，受到机器资源、操作系统等的限制，特别是sever端，其最大并发tcp连接数远不能达到理论上限。在unix/linux下限制连接数的主要因素是内存和允许的文件描述符个数（每个tcp连接都要占用一定内存，每个socket就是一个文件描述符），另外1024以下的端口通常为保留端口。

除了端口号限制问题外，其他几个限制有必要稍微聊一聊。

文件描述符限制问题：

> linux 下一切皆文件，为了更加方便管理文件，使用文件描述符来标识一个文件，比如一个TCP连接，操作系统将其简化为代号5，那么进程之需要针对5做读写即可。文件描述符的数量是有限的，我们可以修改大一点。

线程切换和CPU利用率问题：

> 引入一个著名的C10K问题，指的是当服务器连接数达到 1 万且每个连接都需要消耗一个线程资源时，操作系统就会不停地忙于线程的上下文切换，可能会导致整个系统不响应、响应慢甚至崩溃（CPU利用率达到100%）。不过，出现了IO多路复用模型，通过有限的线程来管理大量TCP连接，大大缓解了该问题。

内存限制问题：

> 这个很容易理解，每个TCP连接本身，以及这个连接所用到的缓冲区，都是需要占用一定内存的，内存是有限的，不可能无限支持TCP连接的创建。

所以，对server端，通过增加内存、修改最大文件描述符个数等参数，单机最大并发TCP连接数超过10万,甚至上百万是没问题的。我们在回答此问题时，不仅需要对网络有所了解，还需要对操作系统有所了解，是一个比较综合的问题。