

TCP

首先还是先来解答这个常考面试题关于 TCP 部分的内容，然后再详细去学习这个协议。

常考面试题：UDP 与 TCP 的区别是什么？

TCP 基本是和 UDP 反着来，建立连接断开连接都需要先需要进行握手。在传输数据的过程中，通过各种算法保证数据的可靠性，当然带来的问题就是相比 UDP 来说不那么的高效。

头部

从这个图上我们就可以发现 TCP 头部比 UDP 头部复杂的多。

对于 TCP 头部来说，以下几个字段是很重要的

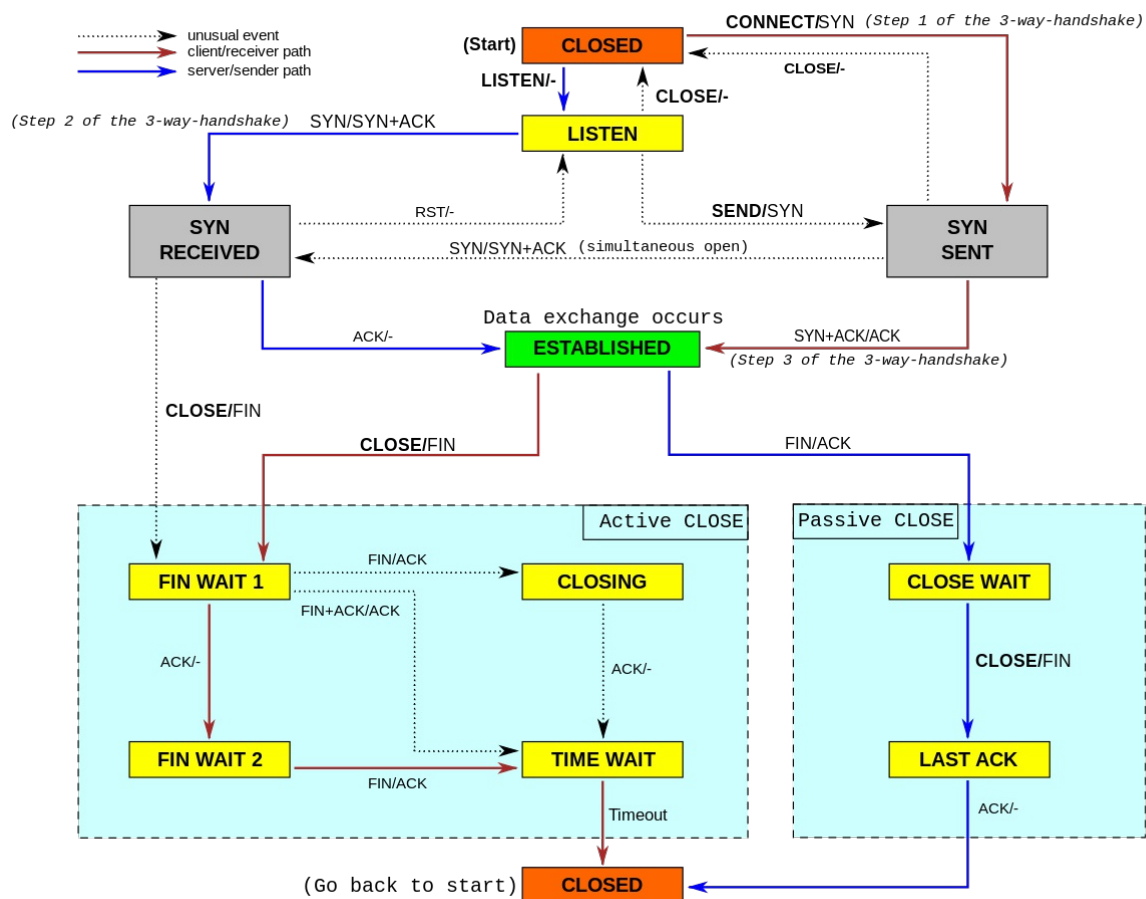
- Sequence number，这个序号保证了 TCP 传输的报文都是有序的，对端可以通过序号顺序的拼接报文
- Acknowledgement Number，这个序号表示数据接收端期望接收的下一个字节的编号是多少，同时也表示上一个序号的数据已经收到
- Window Size，窗口大小，表示还能接收多少字节的数据，用于流量控制
- 标识符
 - URG=1：该字段为一表示本数据报的数据部分包含紧急信息，是一个高优先级数据报文，此时紧急指针有效。紧急数据一定位于当前数据包数据部分的最前面，紧急指针标明了紧急数据的尾部。
 - ACK=1：该字段为一表示确认号字段有效。此外，TCP

还规定在连接建立后传送的所有报文段都必须把 ACK 置为一。

- PSH=1：该字段为一表示接收端应该立即将数据 push 给应用层，而不是等到缓冲区满后再提交。
- RST=1：该字段为一表示当前 TCP 连接出现严重问题，可能需要重新建立 TCP 连接，也可以用于拒绝非法的报文段和拒绝连接请求。
- SYN=1：当SYN=1，ACK=0时，表示当前报文段是一个连接请求报文。当SYN=1，ACK=1时，表示当前报文段是一个同意建立连接的应答报文。
- FIN=1：该字段为一表示此报文段是一个释放连接的请求报文。

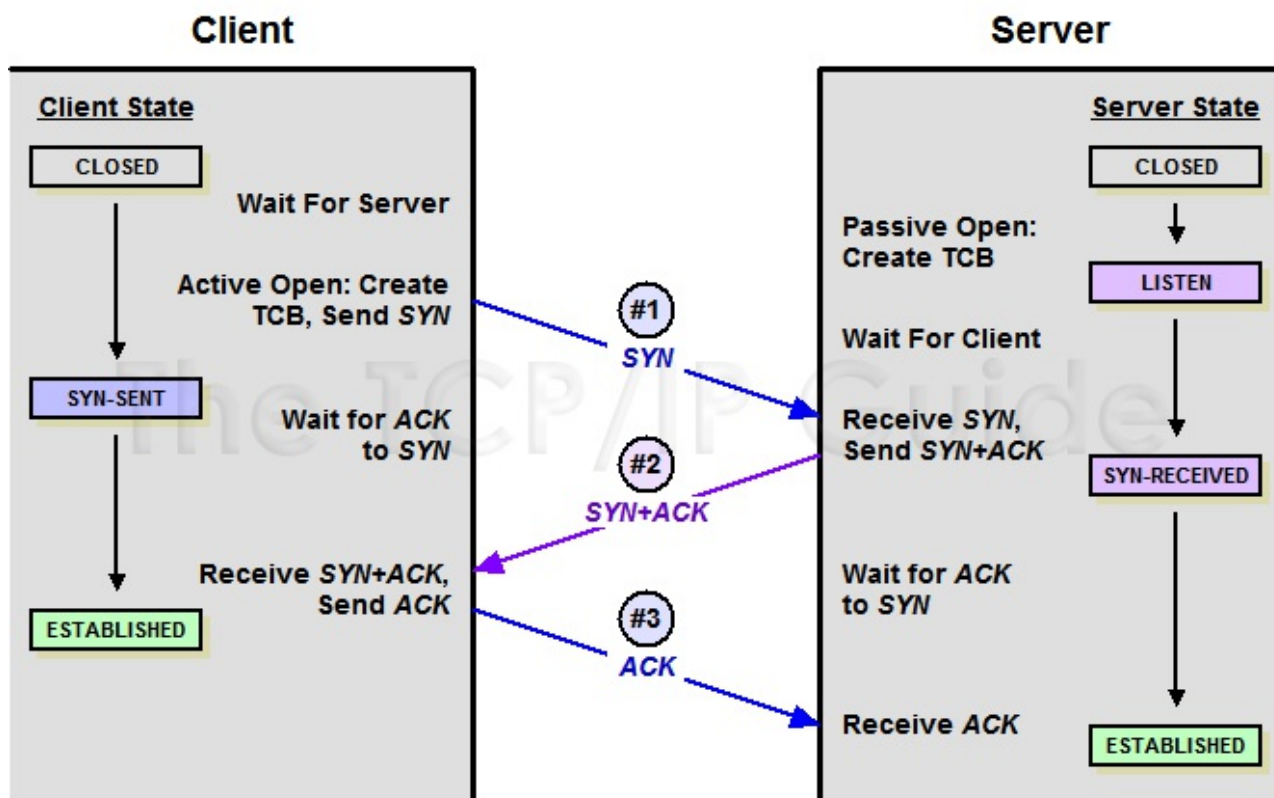
状态机

TCP 的状态机是很复杂的，并且与建立断开连接时的握手息息相关，接下来就来详细描述下两种握手。



在这之前需要了解一个重要的性能指标 RTT。该指标表示发送端发送数据到接收到对端数据所需的往返时间。

建立连接三次握手



首先假设主动发起请求的一端称为客户端，被动连接的一端称为服务端。不管是客户端还是服务端，TCP 连接建立完后都能发送和接收数据，所以 TCP 是一个全双工的协议。

起初，两端都为 CLOSED 状态。在通信开始前，双方都会创建 TCB。服务器创建完 TCB 后便进入 LISTEN 状态，此时开始等待客户端发送数据。

第一次握手

客户端向服务端发送连接请求报文段。该报文段中包含自身的数据通讯初始序号。请求发送后，客户端便进入 SYN-SENT 状态。

第二次握手

服务端收到连接请求报文段后，如果同意连接，则会发送一个应答，该应答中也会包含自身的数据通讯初始序号，发送完成后便进入 SYN-RECEIVED 状态。

第三次握手

当客户端收到连接同意的应答后，还要向服务端发送一个确认报文。客户端发完这个报文段后便进入 ESTABLISHED 状态，服务端收到这个应答后也进入 ESTABLISHED 状态，此时连接建立成功。

PS：第三次握手中可以包含数据，通过快速打开（TFO）技术就可以实现这一功能。其实只要涉及到握手的协议，都可以使用类似 TFO 的方式，客户端和服务端存储相同的 cookie，下次握手时发出 cookie 达到减少 RTT 的目的。

常考面试题：为什么 TCP 建立连接需要三次握手，明明两次就可以建立起连接

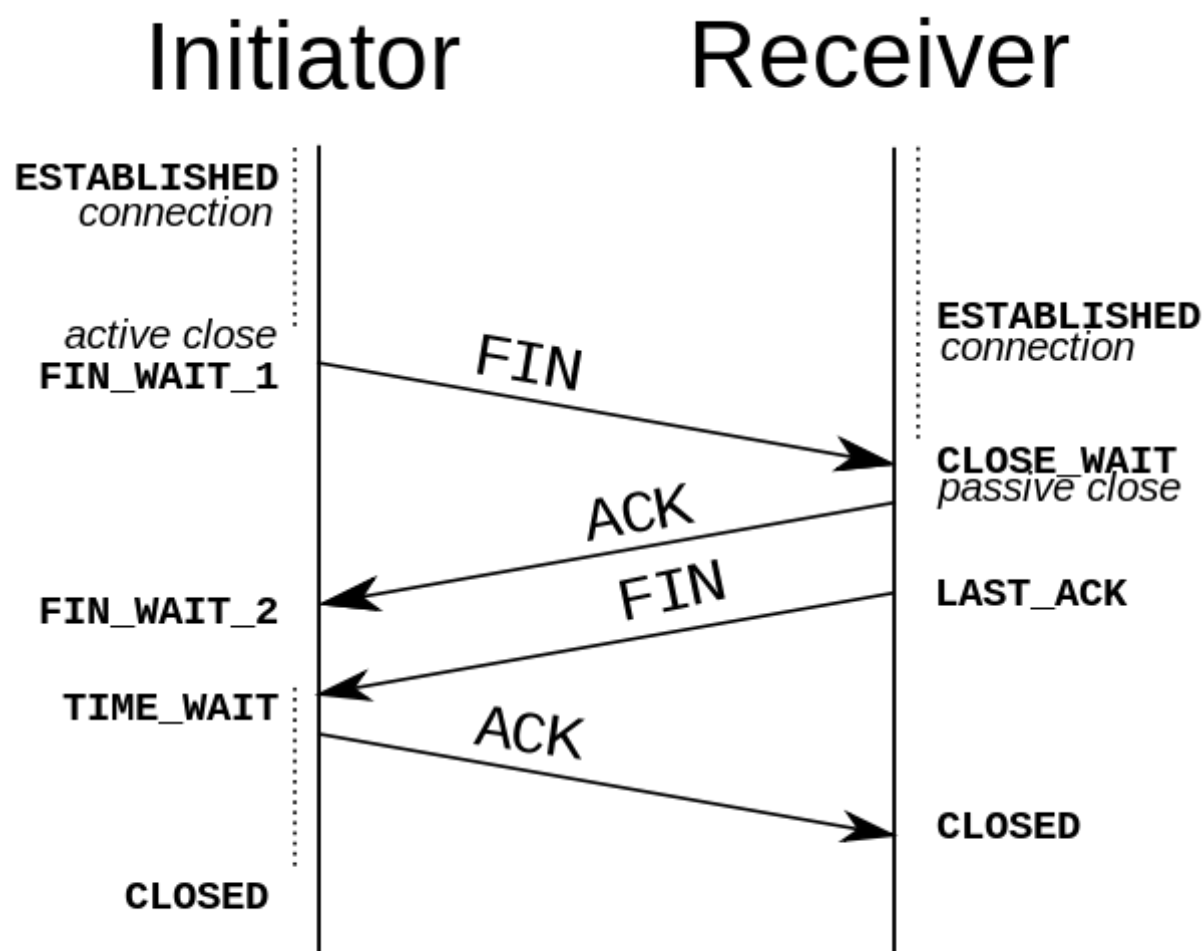
因为这是为了防止出现失效的连接请求报文段被服务端接收的情况，从而产生错误。

可以想象如下场景。客户端发送了一个连接请求 A，但是因为网络原因造成了超时，这时 TCP 会启动超时重传的机制再次发送一个连接请求 B。此时请求顺利到达服务端，服务端应答完就建立了请求，然后接收数据后释放了连接。

假设这时候连接请求 A 在两端关闭后终于抵达了服务端，那么此时服务端会认为客户端又需要建立 TCP 连接，从而应答了该请求并进入 ESTABLISHED 状态。但是客户端其实是 CLOSED 的状态，那么就会导致服务端一直等待，造成资源的浪费。

PS：在建立连接中，任意一端掉线，TCP 都会重发 SYN 包，一般会重试五次，在建立连接中可能会遇到 SYN Flood 攻击。遇到这种情况你可以选择调低重试次数或者干脆在不能处理的情况下拒绝请求。

断开链接四次握手



TCP 是全双工的，在断开连接时两端都需要发送 FIN 和 ACK。

第一次握手

若客户端 A 认为数据发送完成，则它需要向服务端 B 发送连接释放请求。

第二次握手

B 收到连接释放请求后，会告诉应用层要释放 TCP 链接。然后会发送 ACK 包，并进入 CLOSE_WAIT 状态，此时表明 A 到 B 的连接已经释放，不再接收 A 发的数据了。但是因为 TCP 连接是双向的，所以 B 仍旧可以发送数据给 A。

第三次握手

B 如果此时还有没发完的数据会继续发送，完毕后会向 A 发送连接释放请求，然后 B 便进入 LAST-ACK 状态。

PS：通过延迟确认的技术（通常有时间限制，否则对方会误认为需要重传），可以将第二次和第三次握手合并，延迟 ACK 包的发送。

第四次握手

A 收到释放请求后，向 B 发送确认应答，此时 A 进入 TIME-WAIT 状态。该状态会持续 2MSL（最大段生存期，指报文段在网络中生存的时间，超时会被抛弃）时间，若该时间段内没有 B 的重发请求的话，就进入 CLOSED 状态。当 B 收到确认应答后，也便进入 CLOSED 状态。

为什么 A 要进入 TIME-WAIT 状态，等待 2MSL 时间后才进入 CLOSED 状态？

为了保证 B 能收到 A 的确认应答。若 A 发完确认应答后直接进入 CLOSED 状态，如果确认应答因为网络问题一直没有到达，那么会造成 B 不能正常关闭。

ARQ 协议

ARQ 协议也就是超时重传机制。通过确认和超时机制保证了数据的正确送达，ARQ 协议包含停止等待 ARQ 和连续 ARQ 两种协议。

停止等待 ARQ

正常传输过程

只要 A 向 B 发送一段报文，都要停止发送并启动一个定时器，等待对端回应，在定时器时间内接收到对端应答就取消定时器并发送下一段报文。

报文丢失或出错

在报文传输的过程中可能会出现丢包。这时候超过定时器设定的时间就会再次发送丢失的数据直到对端响应，所以需要每次都备份发送的数据。

即使报文正常的传输到对端，也可能出现在传输过程中报文出错的问题。这时候对端会抛弃该报文并等待 A 端重传。

PS：一般定时器设定的时间都会大于一个 RTT 的平均时间。

ACK 超时或丢失

对端传输的应答也可能出现丢失或超时的情况。那么超过定时器时间 A 端照样会重传报文。这时候 B 端收到相同序号的报文会丢弃该报文并重传应答，直到 A 端发送下一个序号的报文。

在超时的情况下也可能出现应答很迟到达，这时 A 端会判断该序号是否已经接收过，如果接收过只需要丢弃应答即可。

从上面的描述中大家肯定可以发现这肯定不是一个高效的方式。假设在良好的网络环境中，每次发送数据都需要等待片刻肯定是不能接受的。那么既然我们不能接受这个不那么高效的协议，就来继续学习相对高效的协议吧。

连续 ARQ

在连续 ARQ 中，发送端拥有一个**发送窗口**，可以在没有收到应答的情况下持续发送窗口内的数据，这样相比停止等待 ARQ 协议来说减少了等待时间，提高了效率。

累计确认

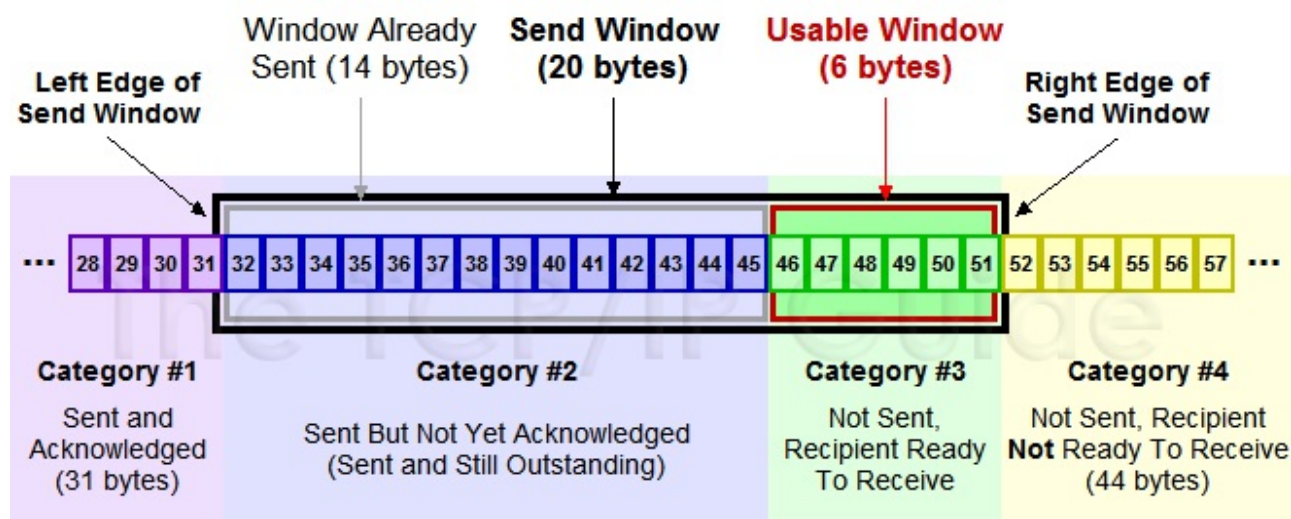
连续 ARQ 中，接收端会持续不断收到报文。如果和停止等待 ARQ 中接收一个报文就发送一个应答一样，就太浪费资源了。通过累计确认，可以在收到多个报文以后统一回复一个应答报文。报文中的 ACK 标志位可以用来告诉发送端这个序号之前的数据已经全部接收到了，下次请发送这个序号后的数据。

但是累计确认也有一个弊端。在连续接收报文时，可能会遇到接收到序号 5 的报文后，并未接收到序号 6 的报文，然而序号 7 以后的报文已经接收。遇到这种情况时，ACK 只能回复 6，这样就会造成发送端重复发送数据的情况。

滑动窗口

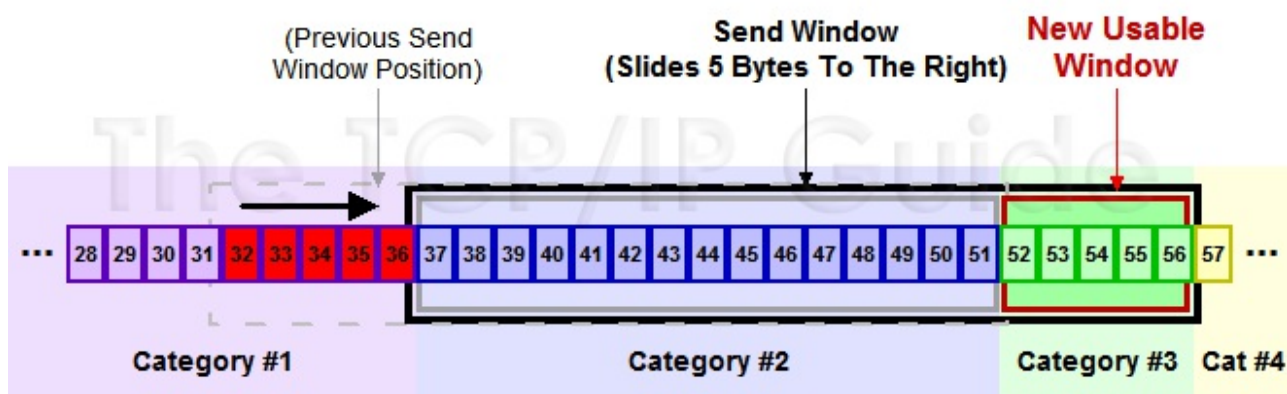
在上面小节中讲到了发送窗口。在 TCP 中，两端其实都维护着窗口：分别为发送端窗口和接收端窗口。

发送端窗口包含已发送但未收到应答的数据和可以发送但是未发送的数据。



发送端窗口是由接收窗口剩余大小决定的。接收方会把当前接收窗口的剩余大小写入应答报文，发送端收到应答后根据该值和当前网络拥塞情况设置发送窗口的大小，所以发送窗口的大小是不断变化的。

当发送端接收到应答报文后，会随之将窗口进行滑动



滑动窗口是一个很重要的概念，它帮助 TCP 实现了流量控制的功能。接收方通过报文告知发送方还可以发送多少数据，从而保证接收方能够来得及接收数据，防止出现接收方带宽已满，但是发送方还一直发送数据的情况。

Zero 窗口

在发送报文的过程中，可能会遇到对端出现零窗口的情况。在该情况下，发送端会停止发送数据，并启动 persistent timer。该定时器会定时发送请求给对端，让对端告知窗口大小。在重试次数超过一定次数后，可能会中断 TCP 链接。

拥塞处理

拥塞处理和流量控制不同，后者是作用于接收方，保证接收方来得及接受数据。而前者是作用于网络，防止过多的数据拥塞网络，避免出现网络负载过大的情况。

拥塞处理包括了四个算法，分别为：慢开始，拥塞避免，快速重传，快速恢复。

慢开始算法

慢开始算法，顾名思义，就是在传输开始时将发送窗口慢慢指数级扩大，从而避免一开始就传输大量数据导致网络拥塞。想必大家都下载过资源，每当我们开始下载的时候都会发现下载速度是慢慢提升的，而不是一蹴而就直接拉满带宽。

慢开始算法步骤具体如下

1. 连接初始设置拥塞窗口（Congestion Window）为 1 MSS（一个分段的最大数据量）
2. 每过一个 RTT 就将窗口大小乘二
3. 指数级增长肯定不能没有限制的，所以有一个阈值限制，当窗口大小大于阈值时就会启动拥塞避免算法。

拥塞避免算法

拥塞避免算法相比简单点，每过一个 RTT 窗口大小只加一，这样能够避免指数级增长导致网络拥塞，慢慢将大小调整到最佳值。

在传输过程中可能定时器超时的情况，这时候 TCP 会认为网络拥塞了，会马上进行以下步骤：

- 将阈值设为当前拥塞窗口的一半
- 将拥塞窗口设为 1 MSS
- 启动拥塞避免算法

快速重传

快速重传一般和快恢复一起出现。一旦接收端收到的报文出现失序的情况，接收端只会回复最后一个顺序正确的报文序号。如果发送端收到三个重复的 ACK，无需等待定时器超时而是直接启动快速重传算法。具体算法分为两种：

TCP Tahoe 实现如下

- 将阈值设为当前拥塞窗口的一半
- 将拥塞窗口设为 1 MSS
- 重新开始慢开始算法

TCP Reno 实现如下

- 拥塞窗口减半
- 将阈值设为当前拥塞窗口
- 进入快恢复阶段（重发对端需要的包，一旦收到一个新的 ACK 答复就退出该阶段），这种方式在丢失多个包的情况下就不那么好了
- 使用拥塞避免算法

TCP New Ren 改进后的快恢复

TCP New Reno 算法改进了之前 **TCP Reno** 算法的缺陷。在之前，快恢复中只要收到一个新的 ACK 包，就会退出快恢复。

在 **TCP New Reno** 中，TCP 发送方先记下三个重复 ACK 的分段的最大序号。

假如我有一个分段数据是 1 ~ 10 这十个序号的报文，其中丢失了序号为 3 和 7 的报文，那么该分段的最大序号就是 10。发送端只会收到 ACK 序号为 3 的应答。这时候重发序号为 3 的报文，接收方顺利接收的话就会发送 ACK 序号为 7 的应答。这时候 TCP 知道对端是有多个包未收到，会继续发送序号为 7 的报文，接收方顺利接收并会发送 ACK 序号为 11 的应答，这时发送端认为这个分段接收端已经顺利接收，接下来会退出快恢复阶段。

小结

这一章节内容很多，充斥了大量的术语，适合大家反复研读，已经把 TCP 中最核心最需要掌握的内容全盘托出了，如有哪里不明白的欢迎提问。

总结一下这一章节的内容：

- 建立连接需要三次握手，断开连接需要四次握手
- 滑动窗口解决了数据的丢包、顺序不对和流量控制问题
- 拥塞窗口实现了对流量的控制，保证在全天候环境下最优的传递数据