TCP可靠传输的技术保证

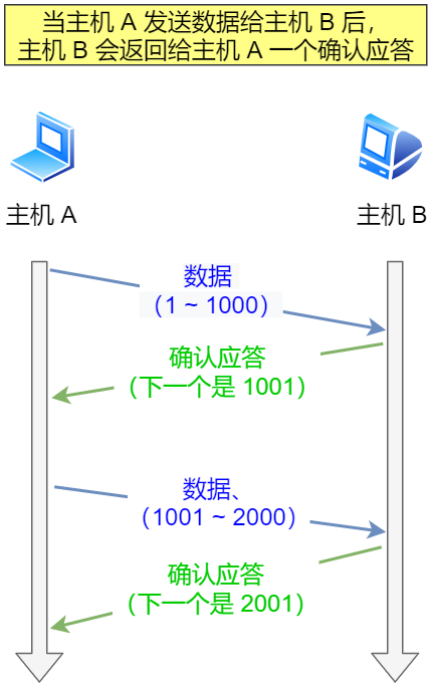
TCP的可靠传输包括：通过序列号、确认应答、重发控制、连接管理、窗口控制等来保证可靠的

部分直接在TCP的头部已经确定了，eg：序列号，能够保证有序，且丢失可查询；确认应答保证这个之前的报文已经正确收到，而不再需要关注

1. 重传机制

序列号 + 确认应答，能够保证正确发送的数据已经被正确接收，那么只需要关注这个之后的数据。

TCP中，发送端的数据带着序列号发送到接收端，接收端根据序列号排序，发现该序列号之前的数据已经有序，那么返回一个确认应答报文，表示已经正确接收该序列号之前的数据



但是，网络的情况是很复杂的，时有发生丢包的情况，那么等待的一方一直接收不到数据，那就需要**进行重传**——主要触发的机制是，发送方一直没有收到另一方的反馈，一定的时间之后就会进行重传

1.1 超时重传

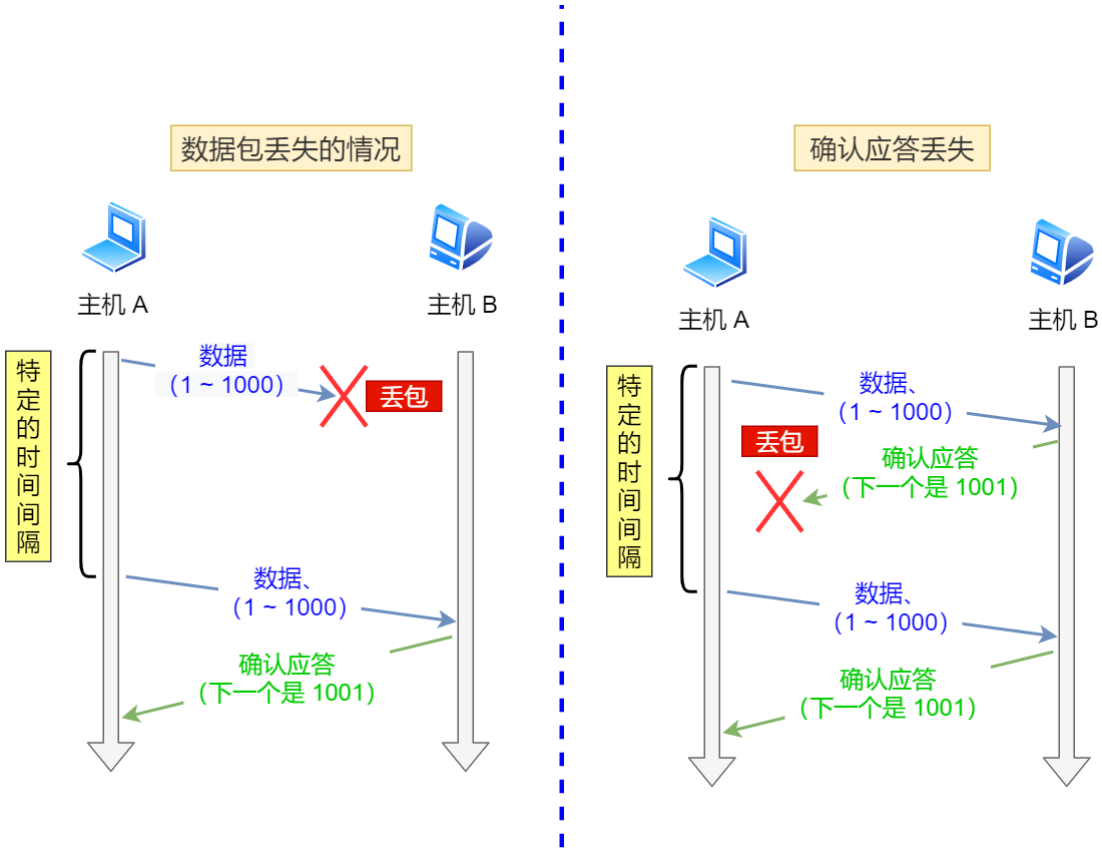
常见的，

概念：**发送数据**时，同时设定一个定时器，在指定时间内都没有收到对方的**ACK确认应答报文**，超时之后就会重新发送该数据

由于TCP两方都需要发送报文给对方，那么存在两种情况的超时重传

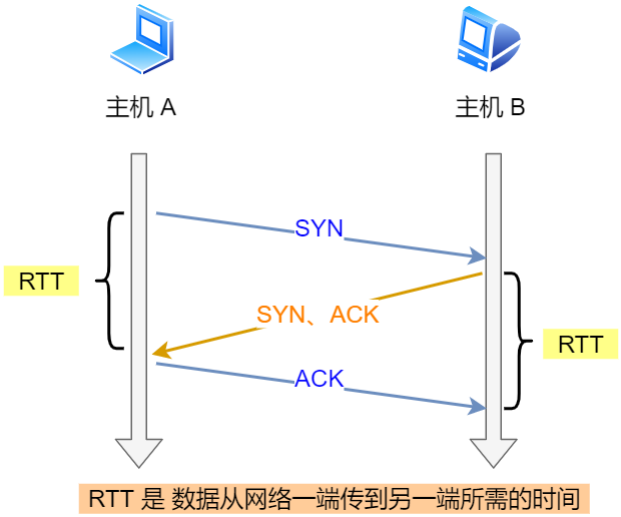
- 发送方的数据包丢失——发送方一直收不到ACK，发送方需要重传
- 接收方的ACK报文丢失——发送方一直收不到ACK，发送方需要重传，从而再次触发ACK报文的发送

——可以发现，不管是哪边丢包，都是发送方重新发送



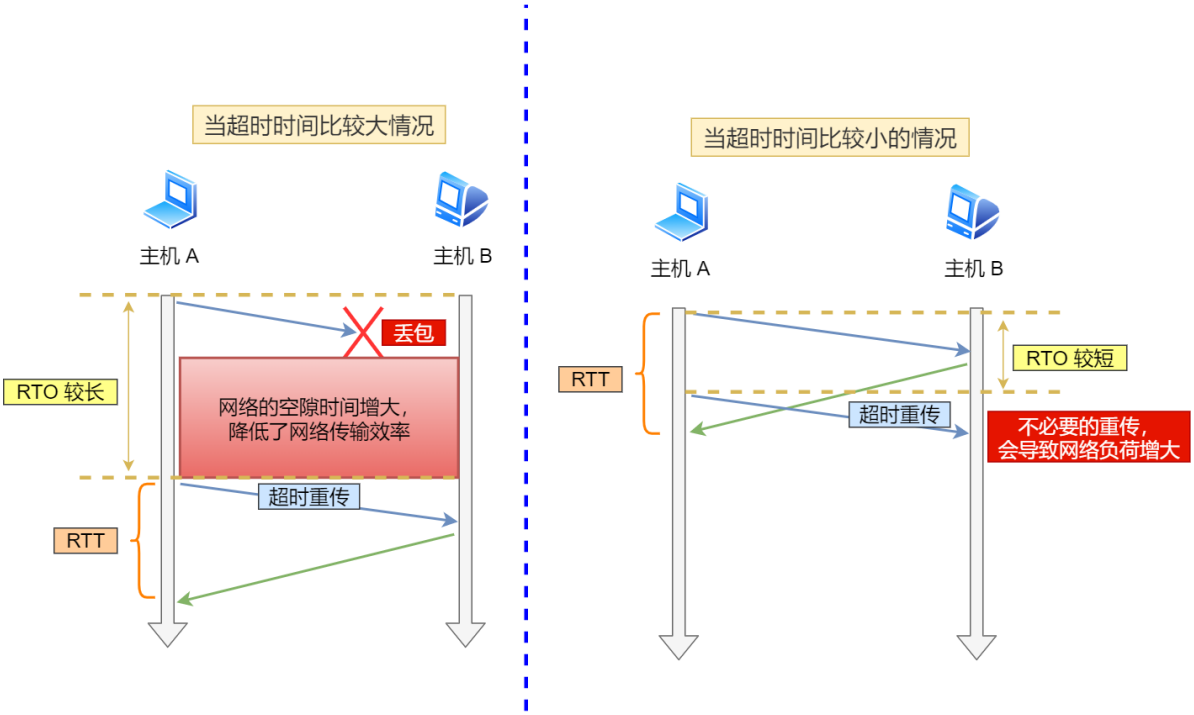
那么**超时时间**该如何设置呢：

RTT（往返时延）：**包的往返时间**，即一个包发送出去到应答响应收到的时间就是一个RTT



超时重传时间**RTO**：

如果设置的RTO不合适会出现啥情况呢?



- RTO太长，网络有很多空闲时间，利用率不高，性能差
- RTO太小，实际上没有丢包，但是超时时间到了，就重发，没有必要的重传，导致网络负载变大，更容易丢包了

所以需要合理设置RTO值——**RTO的值应该略大于RTT值**

可以看出RTO的取值看RTT，但是**RTT的值是会不断波动的**，受网络状态的影响而不是一个固定值，所以**RTO也是一个动态变化的值**

TCP采用了一种自适应算法

eg：Linux动态计算RTO的方法：

- TCP采样RTT的时间，然后加权平均，算出RTT的值，并且该值是不断变化的，即不断进行采样
- TCP还需要采样RTT的波动范围，防止出现大波动，而被采样频率掠过了

  （计算公式如下：）

① 首次计算 RTO ，其中 R1 为第一次测量的 RTT

$$SRTT = R1$$

$$DevRTT = R1/2$$

$$RTO = \mu * SRTT + \partial * DevRT = \mu * R1 + \partial * (R1/2)$$

② 后续计算 RTO ，其中 R2 为最新测量的 RTT

$$SRTT = SRTT + \alpha (RTT - SRTT) = R1 + \alpha * (R2 - R1)$$

$$DevRTT = (1-\beta)*DevRTT + \beta *(|RTT-SRTT|) =(1-\beta) * (R1/2) + \beta * (|R2 - R1|)$$

$$RTO = \mu * SRTT + \partial * DevRTT$$

（SRTT是平滑的RTT，即经过加权平均的值；DevRTT是平滑的RTT与最新RTT的差距）

Linux的参数是：α = 0.125，β = 0.25， μ = 1，∂ = 4（实验中获得的最好的参数）

如果超时重发的数据再次超时时，还是需要重传，这个的超时时间会被加倍

——即，**每当遇到超时重传的时候，下一次的超时重传时间间隔设置为之前的2倍。多次超时，说明网络环境差，不宜频繁反复发送**

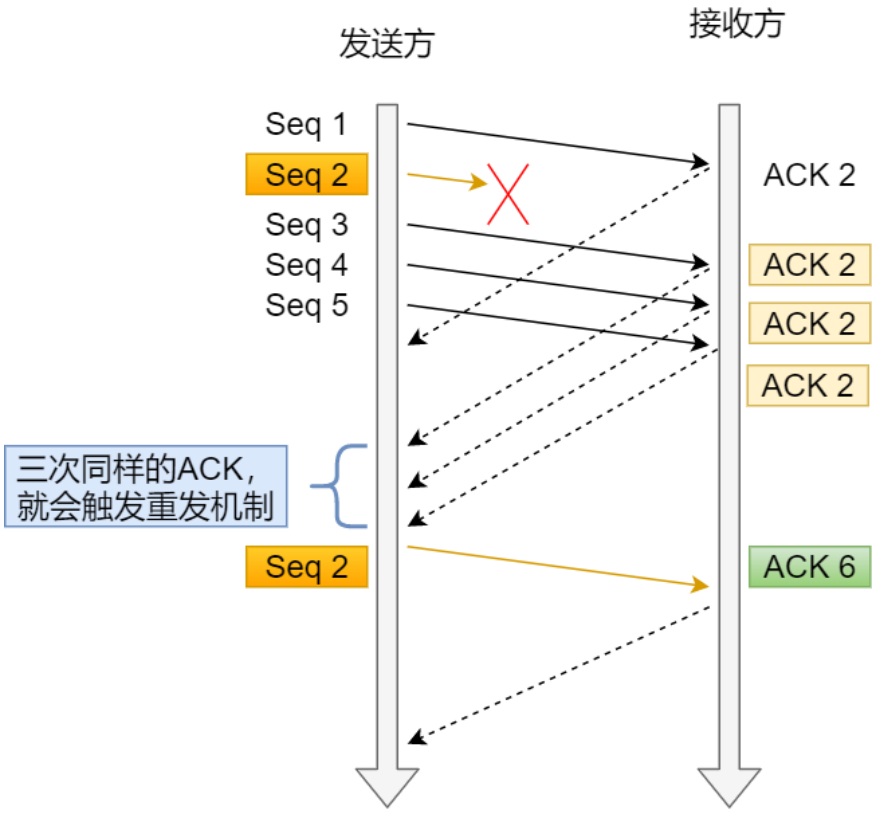
存在的问题：超时等待的周期可能较长，时间相对固定

1.2 快速重传

不以时间为驱动，而是以数据为驱动进行重传

首先要求：首先要求接收方**每收到一个失序的报文段后就立即发出重复确认**（为的是使发送方及早知道有报文段没有到达对方）而不要等待自己发送数据时才进行捎带确认

即下图，在收到seq3后，由于没有收到seq2，就认为是失序报文，所以立即发送ACK报文，且确认号=seq2



因为seq2丢包了，那么接收方即使收到seq3、4、5，但是缓存中缺少2，所以确认应答ack=2，**当连续收到3个相同的ack=2，会认为seq2没有收到，就立即重传**，这之后，返回的ack=6

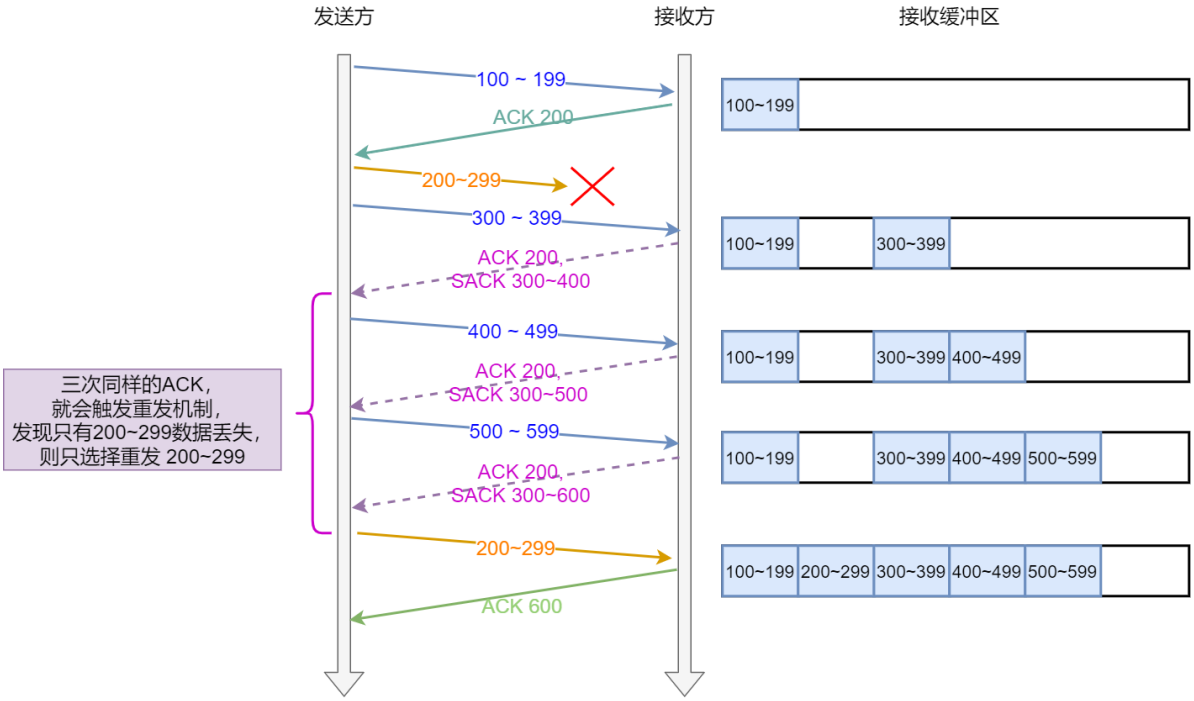
所以，快速重传的机制是：**当收到3个相同的ack报文时，会在定时器过期之前，重传丢失的报文**——可以发现，超时重传和快速重传可以同时存在，互相补充

存在的问题：重传的时候，重传ack确认应答号的报文，还是重传应答号之后的所有报文——因为不知道重复的几个ack是哪些个报文返回的，即丢包之后还存在多个丢包

1.3 SACK（选择性确认）

在TCP头部【选项】字段中，加一个SACK字段，能够将**接收方的缓存地图发送给发送方**，那么发送方就能知道哪些数据已经正确接收、哪些丢失了，那么只需要重传丢失的数据即可

eg：



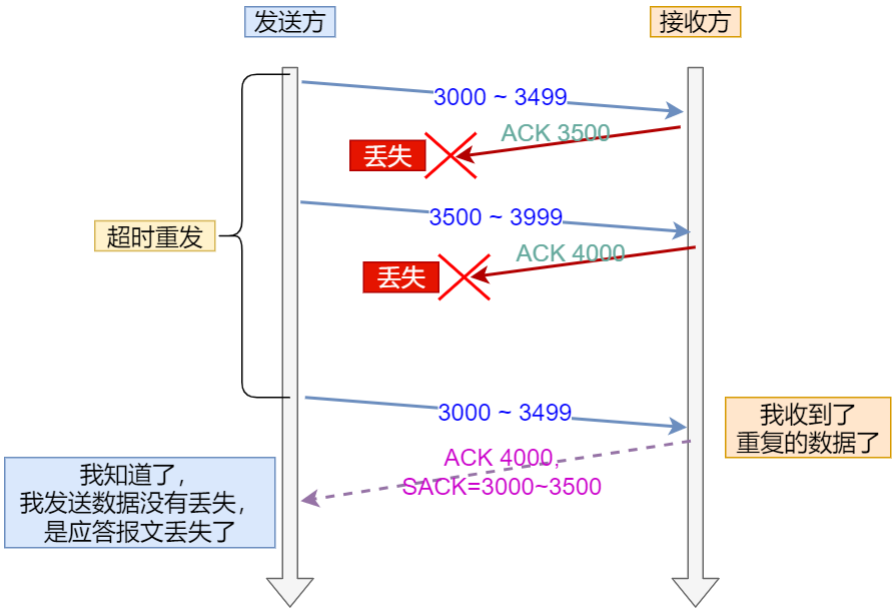
如果要使用SACK，需要双方都支持，Linux2.4之后默认打开的，可以通过`net.ipv4.tcp_sack`配置

可以发现SACK可以和超时重传、快速重传同时存在

1.4 D-SACK

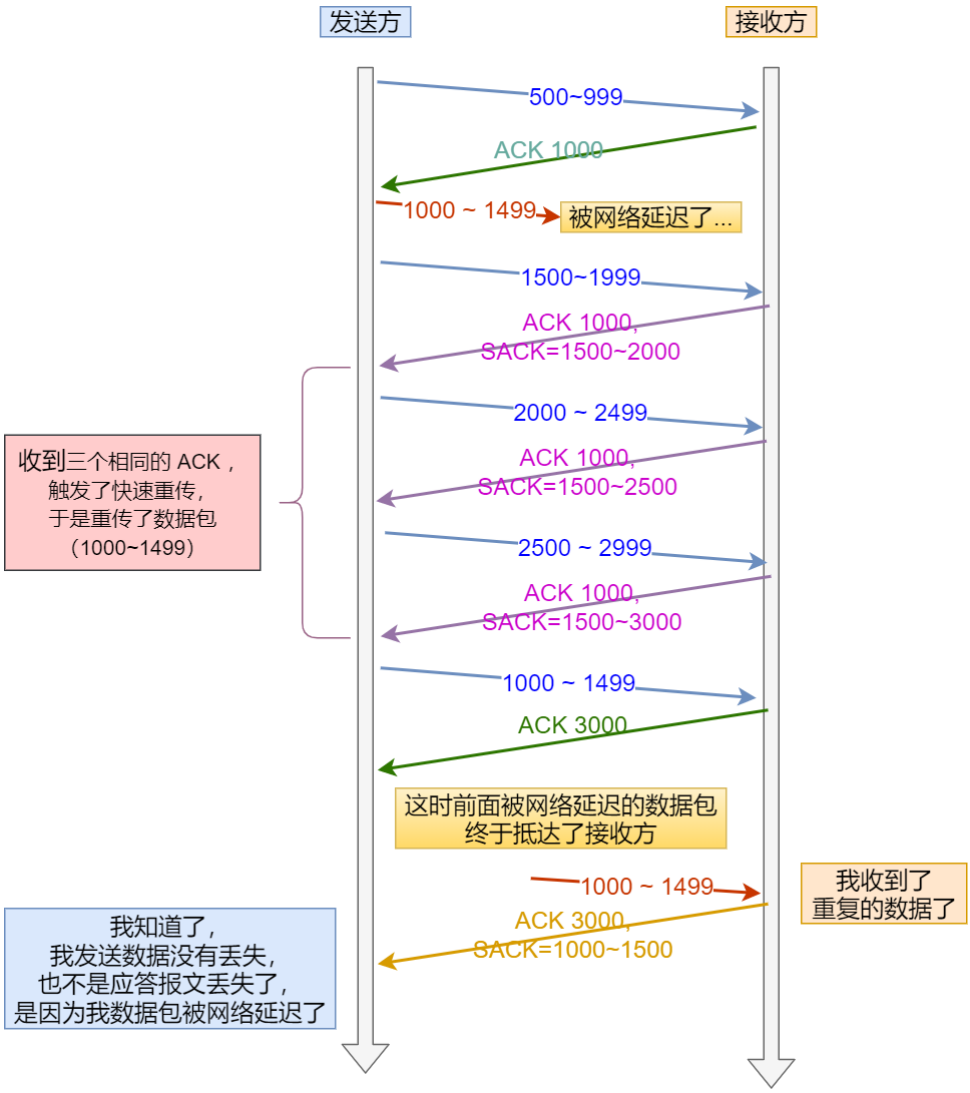
使用SACK告知发送方哪些数据被重复接收

如果是接收方的ACK丢失：



数据已经被正确接收，但是ACK丢失了，那么超时重传之前的数据，就会返回SACK提示

如果是网络延迟：



触发快速重传，此时有序的报文已经到了3000，那么延迟到的报文到了之后，发现是重复报文，则返回的ACK中提示是重复出现的报文

D-SACK的优势：

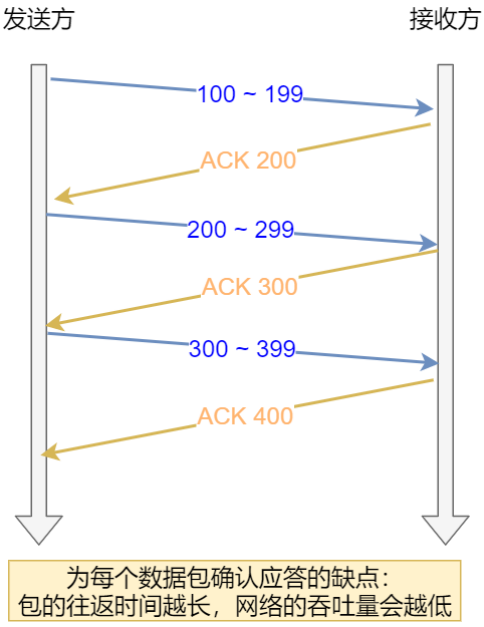
- 让发送方知道丢包的情况：是发送的数据丢失了 or ACK丢失?
- 如果是发送方的问题，那么是发送方的数据丢失了 or 网络延迟（并没有丢）？
- 可以知道，网络中是不是把发送方的数据给复制了？？？？？？？？？？？？

可以看出，D-SACK和超时重传、快速重传同时存在

Linux下可以通过`net.ipv4.tcp_dsack`配置是否开启该机制

2. 滑动窗口

引入的背景：TCP每发送一次数据，都要进行一次确认应答，类似于聊天的一问一答。但是这样对网络的利用率不高，**数据包的往返时间越长，网络吞吐量越低**



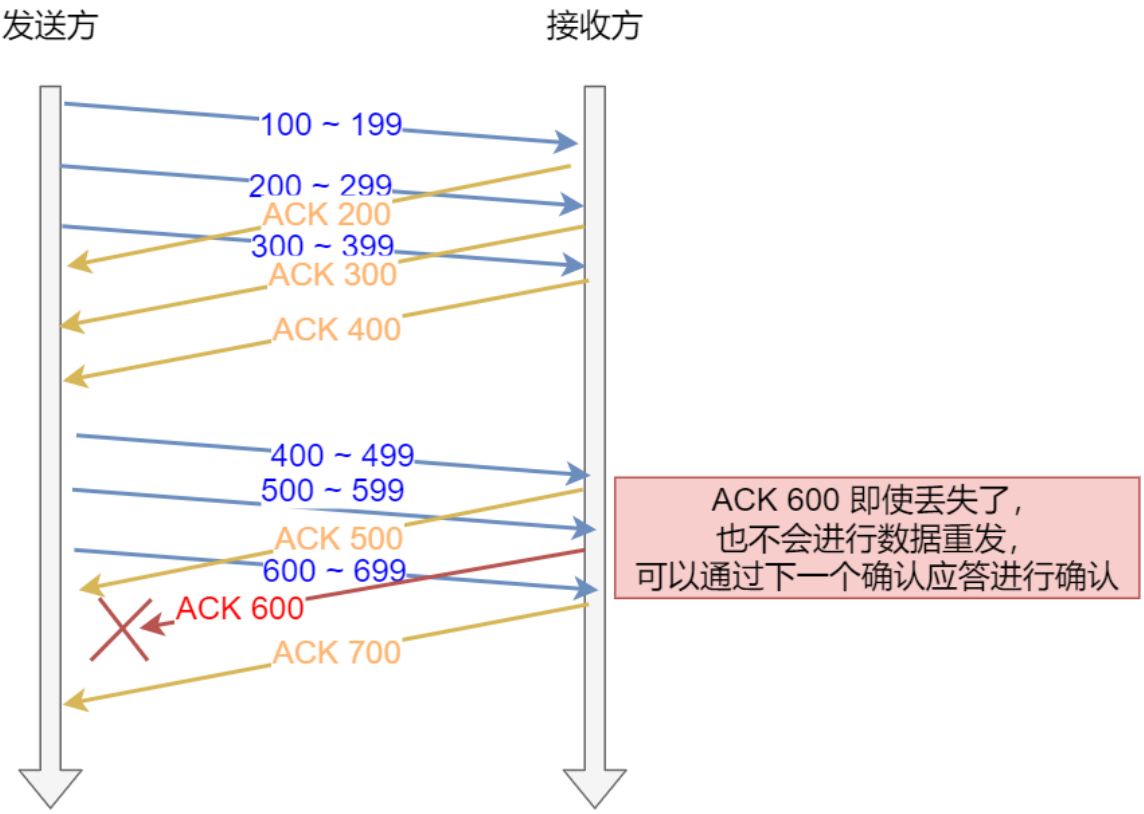
所以，TCP引入了窗口的概念，那么即使往返时间较长，也不怎么会降低效率，因为在上限内发送方可以不断地发送，接收方可以不断地接收和响应，从而网络上存在一串数据包传输。

窗口实际上就是**OS提供的一块缓存空间。**发送方在收到ACK之前（表示还未收到），必须要保存这些未收到ACK的缓冲数据。如果收到ACK，那么可以删除

窗口大小，即无需等待确认应答，还是可以继续发送的数据的最大值

eg：窗口大小为300，即可以连续发送300个数据段，而不需要等待ACK，如果收到ACK，那么窗口还会后移。

如果中途有ACK丢失/包丢失，会根据【下一个确认应答】来判断是否要进行重传



窗口大小的决定：**TCP头部的【窗口大小】字段——是接收端告知发送端自己还可以接受的缓冲区的大小。**然后发送端可以根据这个来设定自己的发送窗口大小，从而控制发送速度。

——所以，大小由接收端决定

ps：发送缓存存放的数据有：

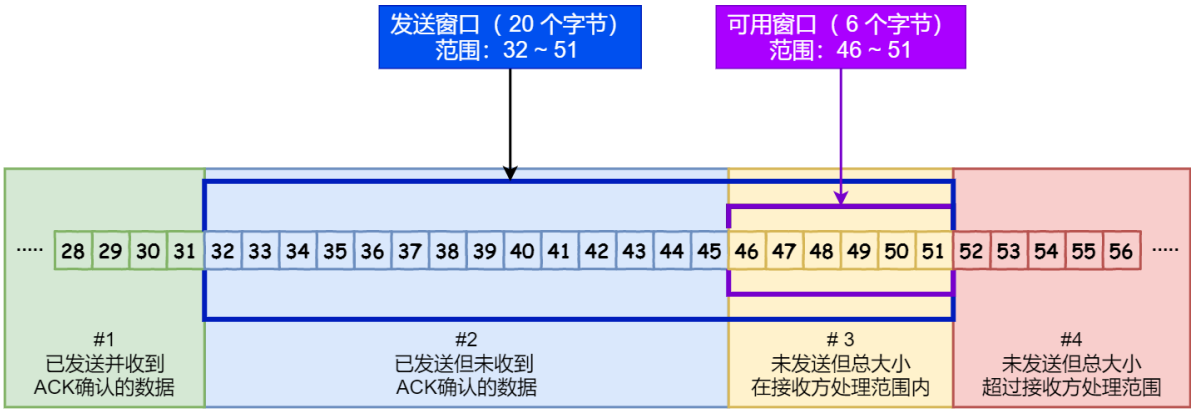
- 上面应用程序传递下来的待发送的数据（未发送）
- TCP已经发送出去，但是尚未收到确认的数据（作为副本需要保存，防止丢失需要重传）

对于已经发送出去，且已经收到确认的，可以删除副本

所以，实际上缓存的起始位置就是发送窗口的起始位置，即下面的#2的头部，#1部分实际上已经被删除了

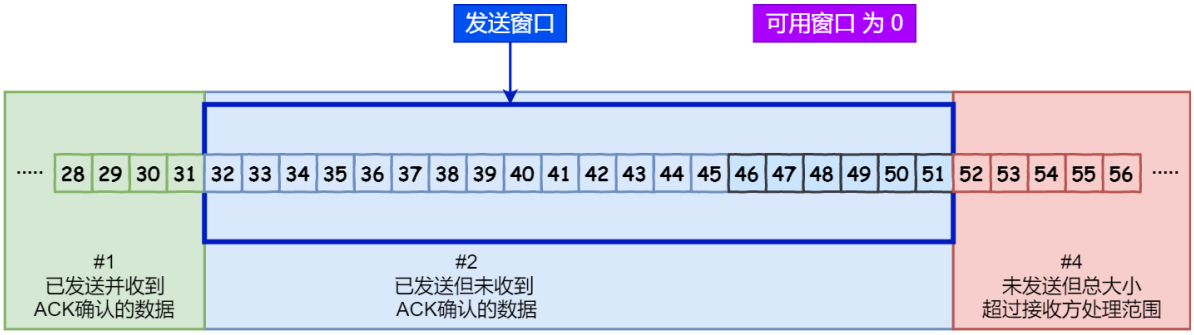
2.1 发送窗口

发送窗口的布局：



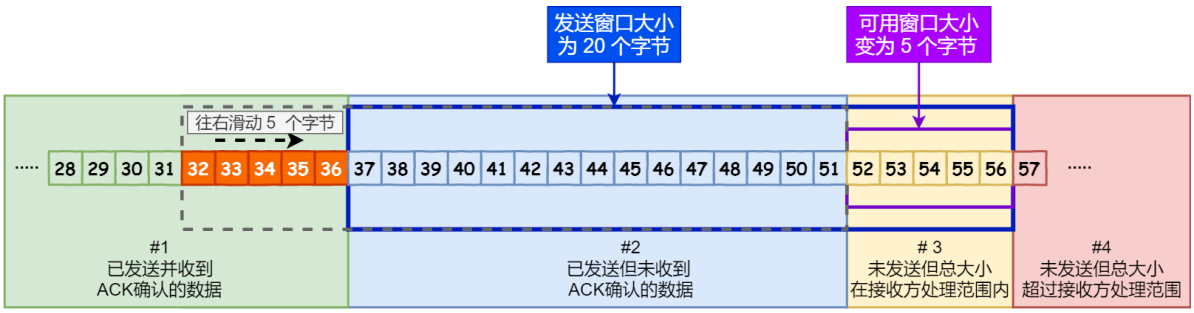
#3是还可以发送的范围，即在未收到#2相关的 ACK之前，还能发送这么多。如果ACK收到了，那么#3还会向后移动（正确来说，是整个深蓝色的框会后移）

临界情况：



已经将所有#3发送出去了，在这个过程中一直没有收到#2中任何的ACK，表示#2的32还未正确被收到，那么窗口不能移动，**窗口耗尽，在未收到ACK之前不能再发送任何数据了**

破局：

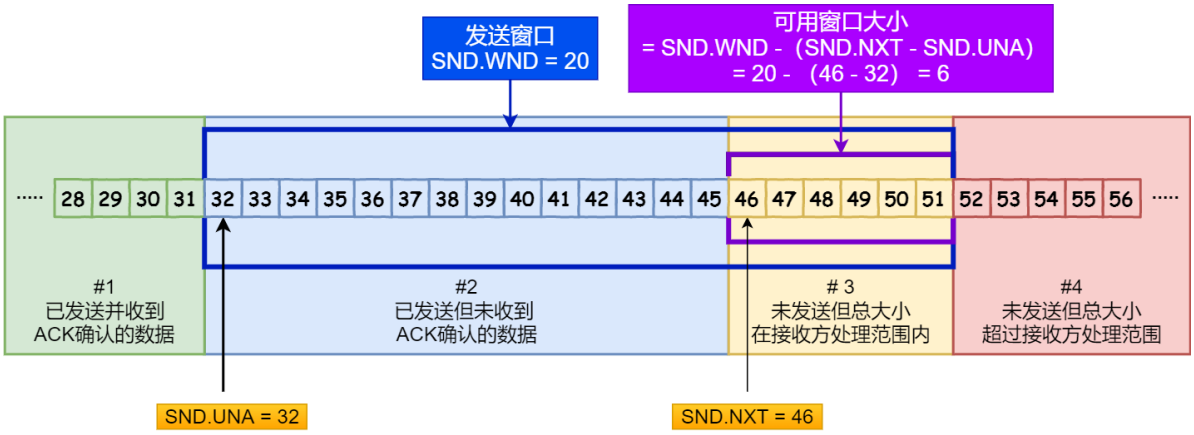


收到了ACK，那么整个窗口向右移动，#3又有数据可以发送了

（类似于算法中的sliding window）

数据结构：

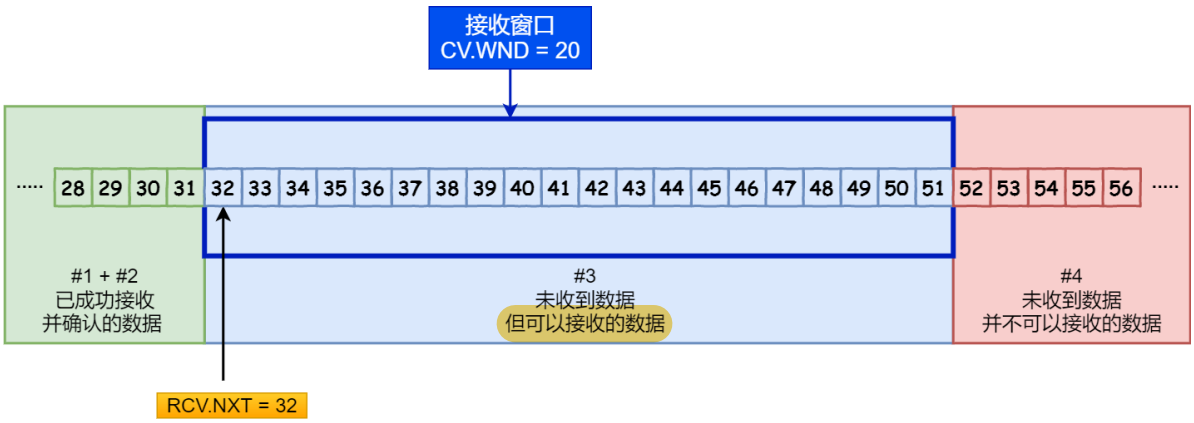
3指针法：2个指针表示绝对位置，1个指针表示相对位移（发送窗口的大小）



- SND.UNA：已发送，但是还未收到ACK的第一个数据，即最近一个ACK的确认应答号
- SND.NXT：未发送，但是在可发送范围的第一个数据
- SND.WND：发送窗口的大小，由接收方决定。可以用来计算还能发送的数据范围，即#4的第一数据 `SND.NXT + (SND.WND - (SND.NXT - SND.UNA))`

2.2 接收窗口

接受窗口的分布：



#1+#2 是在等待上层的应用进程读取，还存放在缓存中。

数据结构：双指针

- RCV.NXT：下一个等待接收的数据（那么窗口就能移动），即最近发送的一个ACK中的确认应答号
- RCV.WND：窗口大小，可以计算出#4的第一个数据值：`RCV.NXT + RCV.WND`

接收窗口和发送窗口不完全相等，是**约等于**

因为：窗口大小也是不断变化的，如果接收方的应用处理数据的速度比较快（#1 +#2的数据取得快），那么接收方的窗口会变大。而新的窗口大小是通过TCP报文的【Window】字段告知的，所以**存在一定的时延**

接收缓存存放的是：

- 按序到达，但是没有被上层应用读取的数据（如果读取就可以丢了）
- 未按序到达的数据

  （对于不按序到达的数据应如何处理，TCP标准并无明确规定，通常对不按序到达的数据是先临时存放在接收窗口中，等到字节流中所缺少的字节收到后，再按序交付上层的应用进程。）

接收方发送确认：**有累积确认的功能**，这样可以减小传输开销。接收方可以在合适的时候发送确认，也可以在自己有数据要发送时把确认信息**顺便捎带上**。

但是推迟确认有一定要求：

- 不应过分推迟发送确认，否则会导触发不必要的重传
- 捎带确认实际上并不经常发生，因为大多数应用程序不同时在两个方向上发送数据。

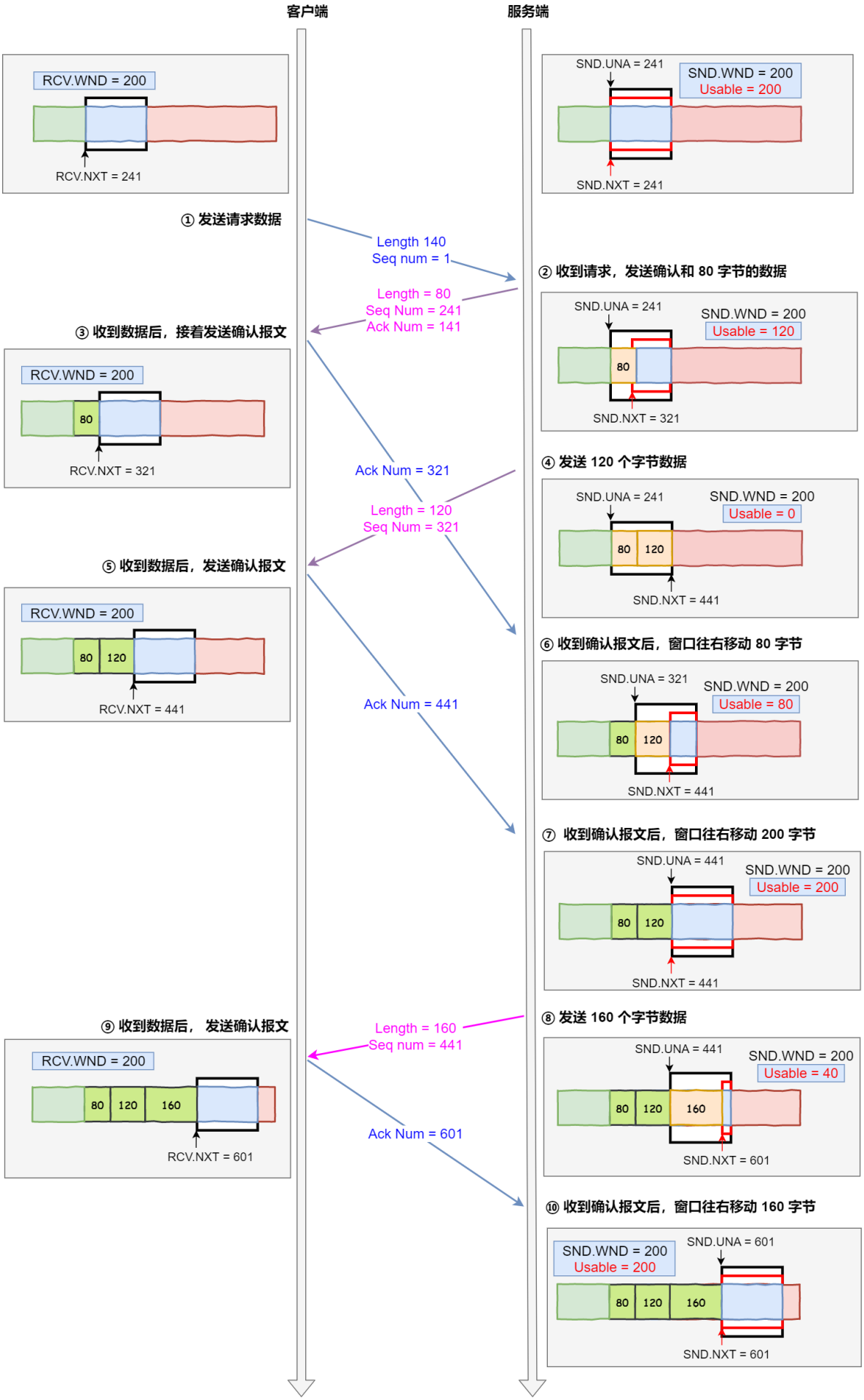
3. 流量控制

背景：发送方不能一直给接收方发送数据，接收方的处理能力有限，如果溢出了，引起接收端丢包，那么发送发会触发超时重传，而导致网络流量的浪费

所以引出：流量控制：可以让发送方根据接收方的实际接收能力，来**控制发送的速度**

eg：

- 客户端是接收方，服务端是发送方
- 接收窗口 = 发送窗口 = 200，且整个过程中窗口大小维持不变



流程梳理：（注意，这边是服务端向发送端发送数据，所以只关注客户端的接收窗口，和服务端的发送窗口）

1. 客户端请求数据，请求的长度为`Length = 140`，起始字段为`Seq Num = 1`（所以，等待对方的ACK的确认应答号为141）
2. 服务端收到请求后，发送ACK，并且总长度为80的数据，作为发送方可用窗口大小减少到120，且`SND.ACK=241`，`SND.NXT= 241 +80 = 321`

   该报文中，有确认应答号`ACK Num = 141`（表示客户端发送的140字节已经正确接收）。发送数据的长度为`Length = 80`，发送的数据的起始字段`Seq Num = 241`，（所以等待对方的ACK的确认应答号为321）
3. 客户端收到数据后，发送第一个ACK，并且作为接收方，接收窗口右移`RCV.NXT=241 + 80 = 321`

   报文中，确认应答号为`Ack Num = 321`
4. 在服务端未收到ACK之前，就已经发送新的数据了，发送的数据长度为120，那么`SND.NXT=321 + 120 = 441`，此时的可用窗口大小减少到0。而等待确认的数据起始仍为`SND.UNA=241`

   报文中，发送的数据长度为`Length = 120`，发送的数据的起始字段`Seq Num = 321`
5. 客户端收到数据后，发送第二个ACK，且接收窗口右移`RCV.NXT=321 + 120 = 441`

   报文中的确认应答号为`Ack Num = 441`
6. 此时，服务端已经接收到客户端发送的第一个ACK，那么发送窗口右移`SND.UNA = 321`，由于未发送新的数据，那么`SND.NXT = 441`，此时可用窗口大小为80
7. 此时，服务端又收到了客户端发送的第二个ACK，窗口右移`SND.UNA=441`，`SND.NXT=441`，此时可用窗口大小为200
8. 之后操作同上。

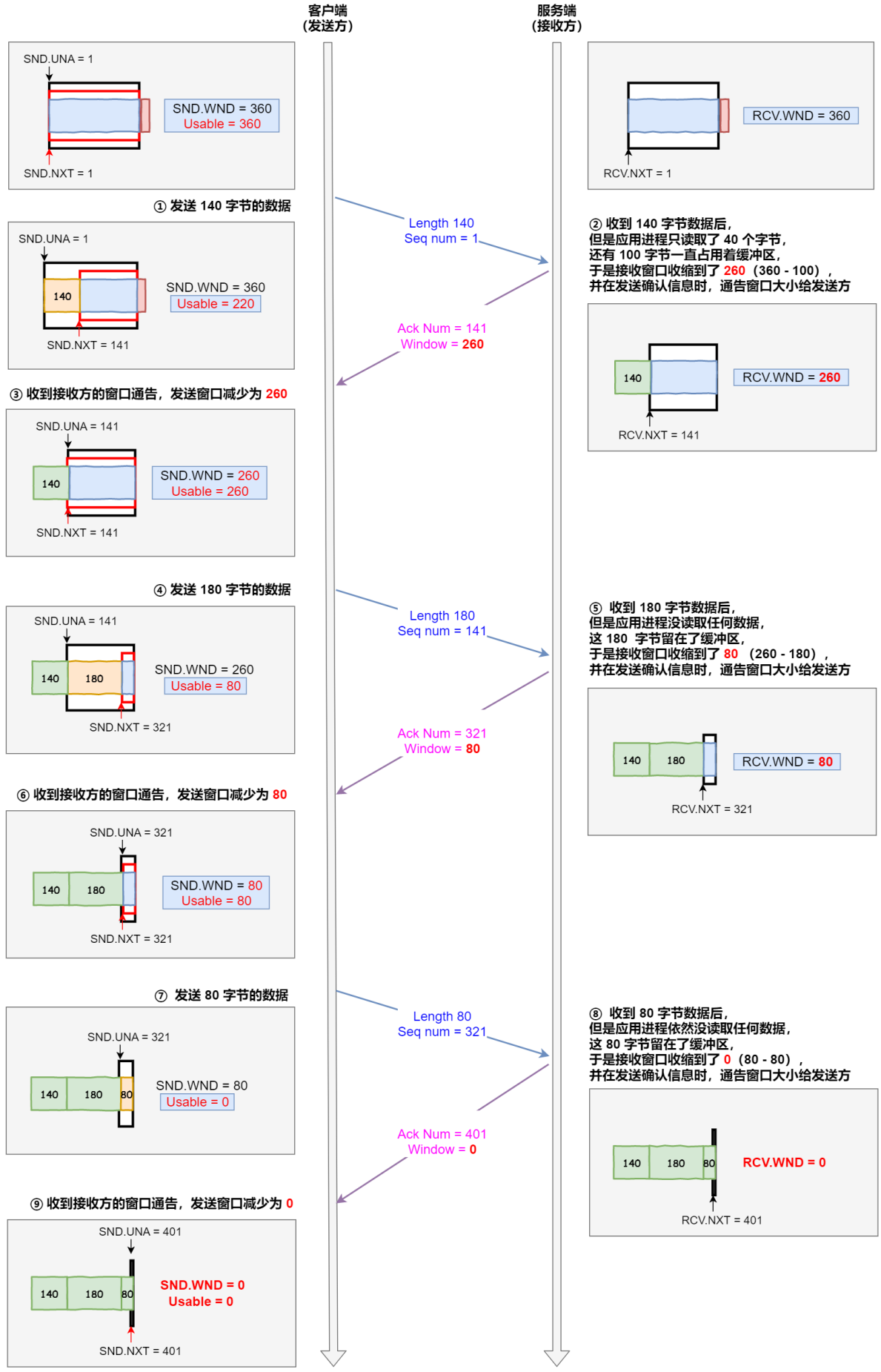
3.1 OS的缓冲区和滑动窗口的关系

OS的缓冲区是受OS控制的，而窗口中的内容都是存放在缓冲区中的。

所以发送和接收的速度，进程读取缓存的速度都是会影响缓冲区可存放的空间

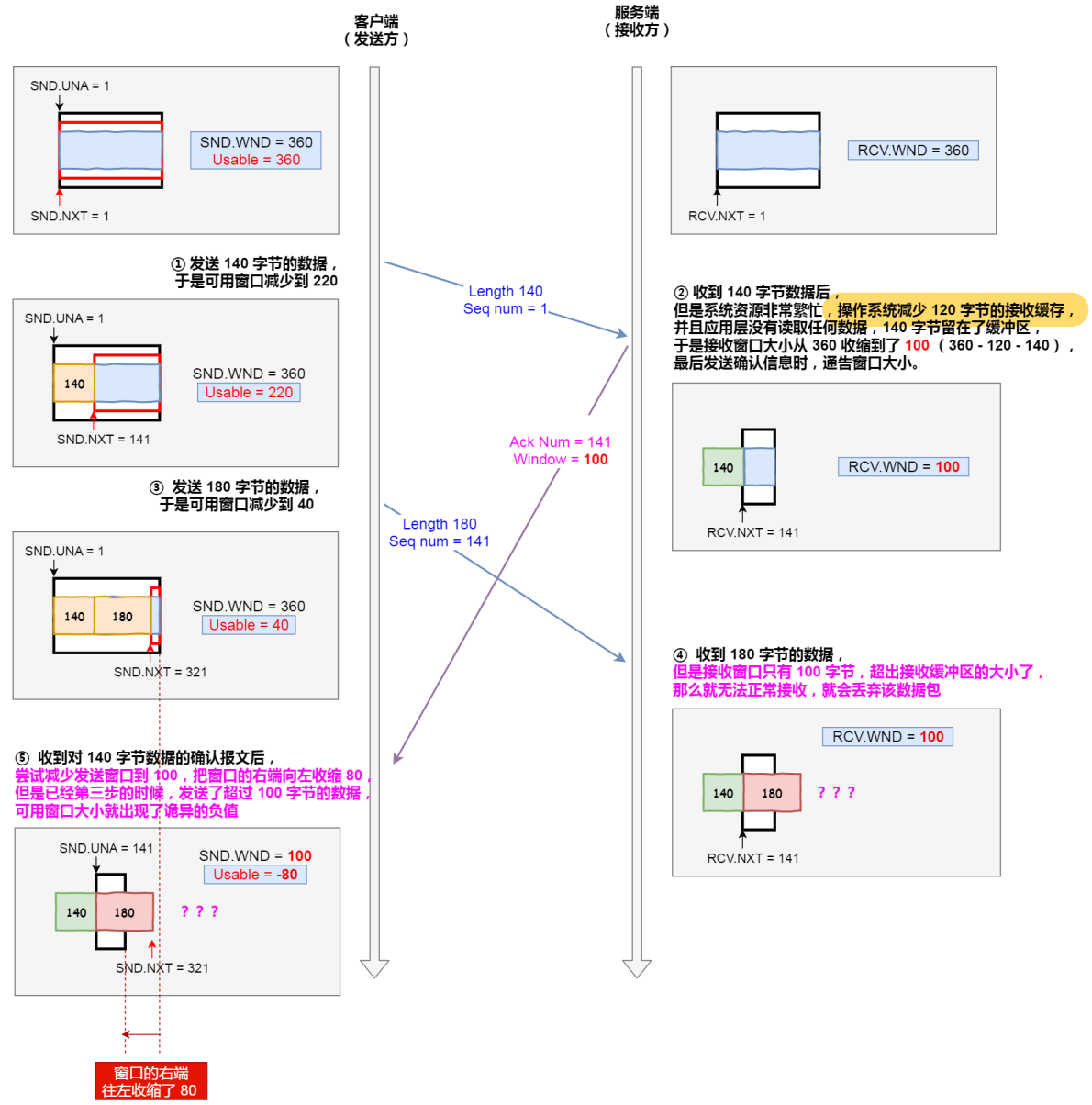
eg：如果应用程序没有及时读取缓存，看发送和接收窗口的变化：

- 客户端作为发送方，服务端作为接收方，窗口初始大小为360
- 服务端很繁忙，不能即使处理数据，即



窗口大小由于上面进程没有即时取数据而不断收缩，最后减少到0——就是窗口关闭，不能再发送了

eg：OS会直接减少接收缓存，而应用程序又无法及时读取缓存数据



主要原因是：**OS减小了缓存**，导致按照原窗口发送的数据存在丢包的情况。

所以，如果减少了缓存，然后再收缩窗口，就会出现丢包的情况。所以为了避免此种情况，TCP不允许同时减少缓存又收缩窗口的，而是先收缩窗口，过段时间再减少缓存，可以避免丢包

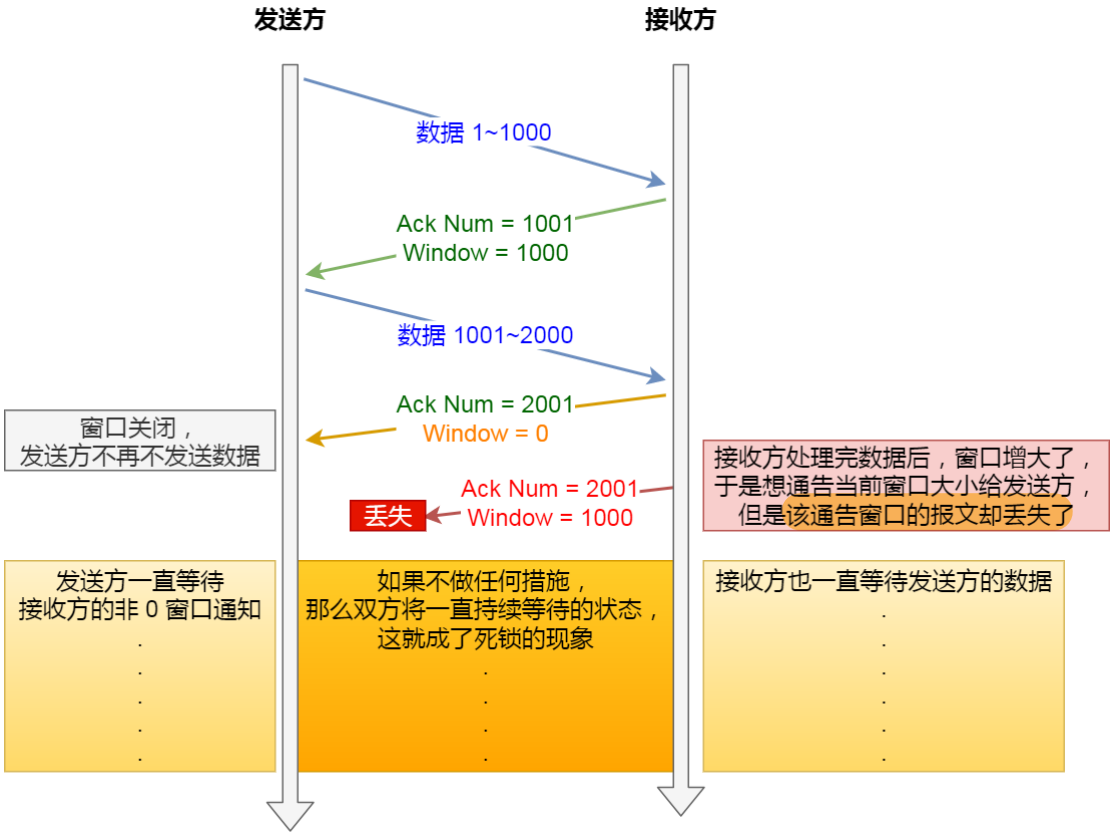
3.2 窗口关闭

概念：就是接收窗口大小减少到0，那么发送端无法再发送数据

潜在的问题：

因为接收方通知窗口大小是通过ACK报文中的【Window】来控制的。而如果从接收窗口从0变成非0的ACK通知报文，如果该报文丢失，那么会陷入死锁。

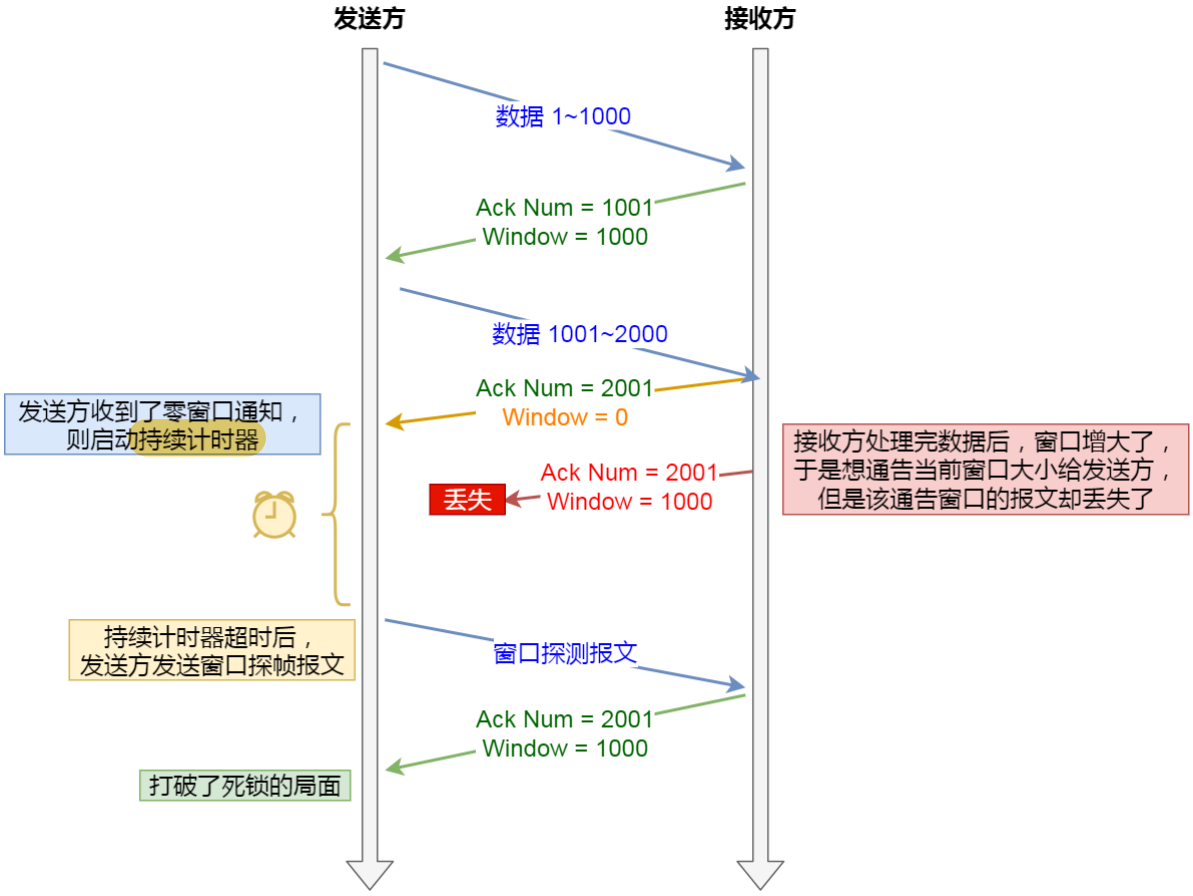
接收方以为发送方正确收到了ACK，等待发送发的数据报；而发送方一直没有收到窗口改变的报文，而一直等待窗口变成非0：



造成了**死锁**，所以需要避免这个情况：

TCP为每个连接设定了一个持续定时器，**只要TCP连接一方收到了对方的零窗口通知，就启动持续计时器。**如果持续计时器超时，发送方就会发送窗口探测报文，然后接收方会发送ACK报文，并且给出最新的接收窗口大小。那么就打破死锁的情况了。

- 如果接收窗口大小仍为0，那么发送方收到ACK之后，重置持续计时器
- 如果接收窗口大小非0，那么发送发收到ACK之后，可以对应修改发送窗口



——窗口探测的次数3次，每次间隔30s~60s，如果3次之后窗口仍为0，有些TCP可能会发RST来断开连接

3.3 糊涂窗口综合征

概念：接收方的上层线程没有及时取出数据，而导致窗口越来越小，而到最后，**接收方告知发送方窗口大小（很小，几个字节），那么根据前面的机制，发送方还是会发送几个字节的数据**——会出现：报文头部有很多字节（TCP + IP=40字节+），而传输的数据只有几个字节，**代价太大**

所以综合征的情况会发生在发送方和接收方：

- 接收方会通知，只剩下一个小窗口
- 发送方会根据小窗口，发送小数据

所以可以从上面两个方面避免综合征

- 接收方不通知小窗口

  当**接收窗口大小 < min(MSS，缓存空间 / 2)，直接关闭窗口**——通知接收窗口大小为0，那么对方不能再发送数据了。等窗口大小不满足之后，就可以打开窗口

- 发送方知道对方只有小窗口后，不发送小数据了

  使用Nagle算法，延时处理

  只有等到 `窗口大小 >= MSS` / `数据大小 >= MSS`；或者，得到了之前发送的数据的ACK。才会进行发送，否则就一直囤积数据

Nagle是默认打开的，只有在小数据交互的场景中，就关闭，eg：telnet、ssh等程序中

4. 拥塞控制

为了避免发送方的数据填满了接收方的缓存，而不管当前的网络情况——即拥塞控制是关注了当前的网络情况

主要是因为：网络是共享的，可能自己的TCP交互不是很繁忙，但是其他主机的通信导致当前网络繁忙。

网络出现拥堵时，如果继续发送大量数据包，可能会导致数据包时延、丢失等，那么容易触发超时重传，而一旦重传之后网络的负担就更重了，更容易导致时延、丢失等，如此恶性循环。所以合理的是，当网络发生拥塞时，TCP会执行降低发送的数据量

目的：**拥塞控制避免发送方的数据填满整个网络**

所以设定了一个【拥塞窗口】

拥塞窗口：发送方维护的一个变量，**会根据当前的网络情况动态变化**，所以此时发送窗口的大小 `SWND = min(cwnd, rwnd)`

规则：

- 网络中，没有出现拥塞，那么cwnd变大
- 网络中出现拥塞，cwnd变小

那么如何判断网络出现拥塞呢？——如果**触发了超时重传，就认为网络出现拥塞**

下面就是拥塞控制的算法：

4.1 慢启动

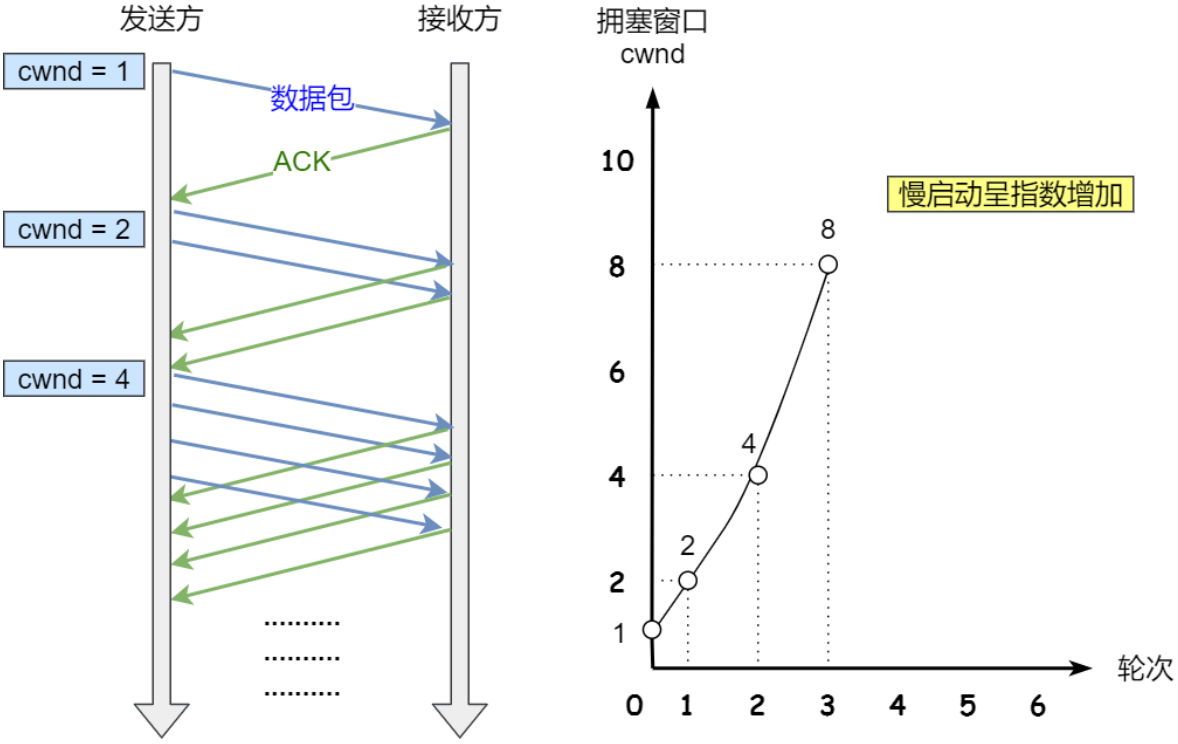
概念：一点点提高发送数据的量（即尝试着增加发送量，直到达到最优状态）

规则：当发送方收到一个ACK，拥塞窗口CWND+1，那么**每经过一个传输轮次，拥塞窗口cwnd就加倍。**

即：第1次，发送1个，收到一个ACK，窗口增大到2；第2次，可以发送2个，收到2个ACK，窗口+1+1；第3次，可以发送4个了.....（指数级增加）

效果如下：（图的窗口是以个为单位的，而不是以字节为单位，只是方便直观点）

一开始很缓慢，一个数据包发送，等到ACK才发送下一个；后面就是发送一批数据包，等待一批ACK



——**增加速度是指数级增加**

慢启动存在一个阈值（不能一直增加，后面增加的幅度越来越大，越来不精细，所以不再适合）

称为：**慢启动门限** `ssthresh` 状态变量：当cwnd < ssthresh，用慢启动算法；cwnd >= ssthresh，使用拥塞避免算法

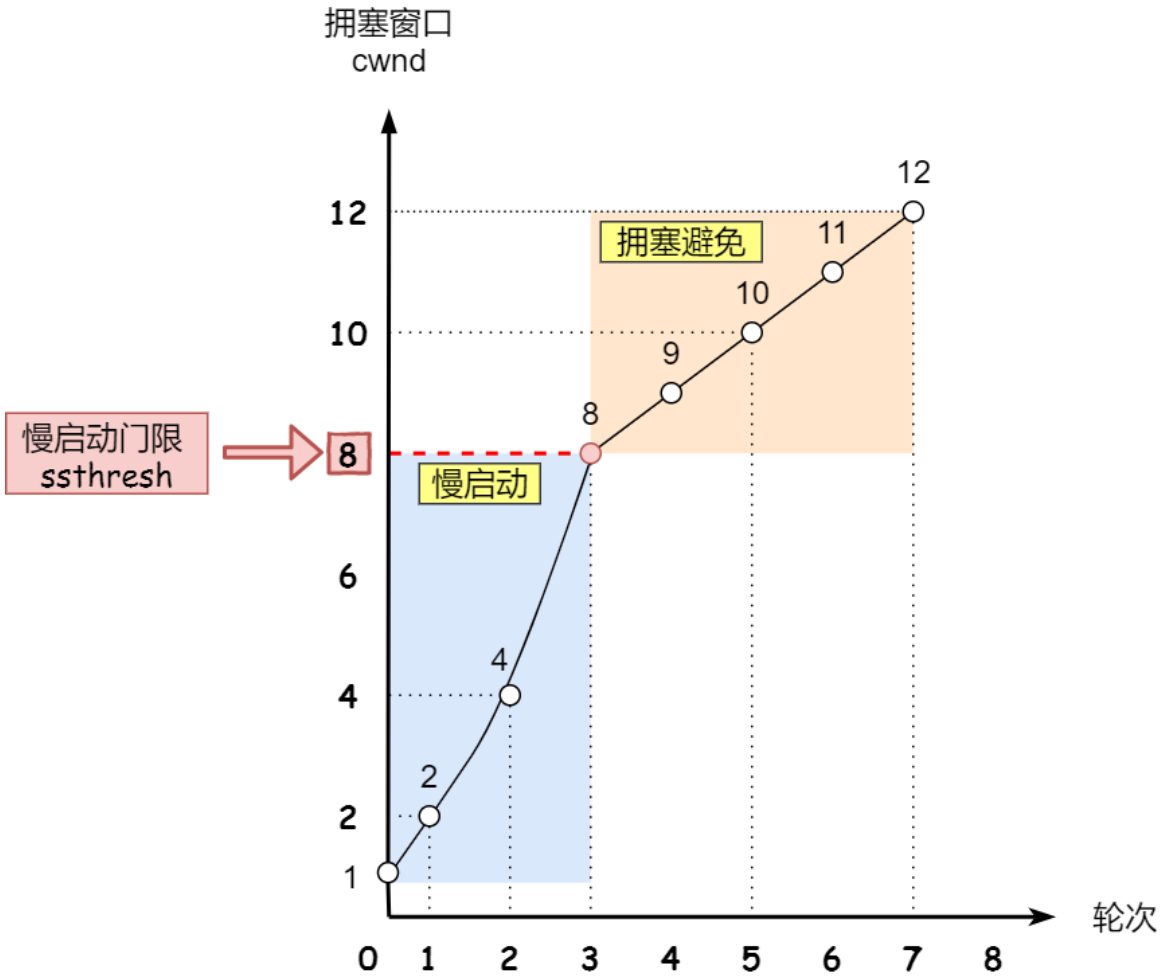
一般：ssthresh = 65536B

4.2 拥塞避免

规则：每当收到一个ACK，cwnd增加1/cwnd，即**每个轮次，cwnd增加1**

eg:

每一轮（收到了全部发送出去的数据的ACK，那么CWND增加1）——按照轮次，实现了**线性的增加**，但是还是处于增加状态的



4.3 拥塞发生

如果一直增加，就会导致网络拥塞，出现丢包的情况，出现重传的情况，那么就会触发【拥塞发生算法】

针对不同的重传机制，拥塞发生算法也是不同的

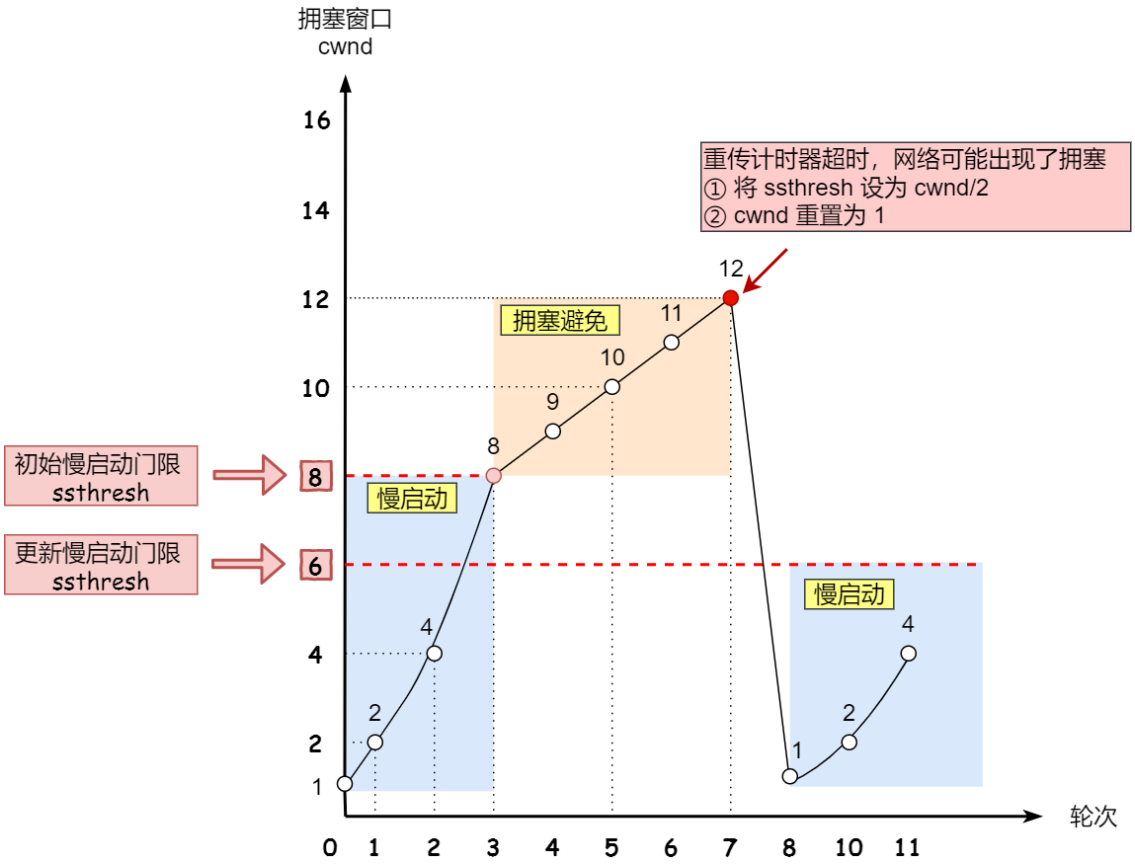
- 超时重传——认为网络已经出现拥塞情况

  当发生超时重传，那么ssthresh和cwnd发生变化：

  - ssthresh = cwnd / 2（初始ssthresh是提前设定的，而之后的ssthresh根据计算得到的）
  - cwnd = 1

  ——重新回到了慢启动的状态，且门限变小了

  措施剧变，对应的反应也剧烈，会造成网络卡顿



- 快速重传（3次重复的ACK，就快速重传之前的报文）——说明当前网络情况不是很糟糕

  出现快速重传之后，ssthresh和cwnd均变化

  - cwnd = cwnd / 2
  - ssthresh = cwnd ——即：ssthresh = cwnd = cwnd / 2，参数都减半
  - 进入快速恢复算法

4.4 快速恢复

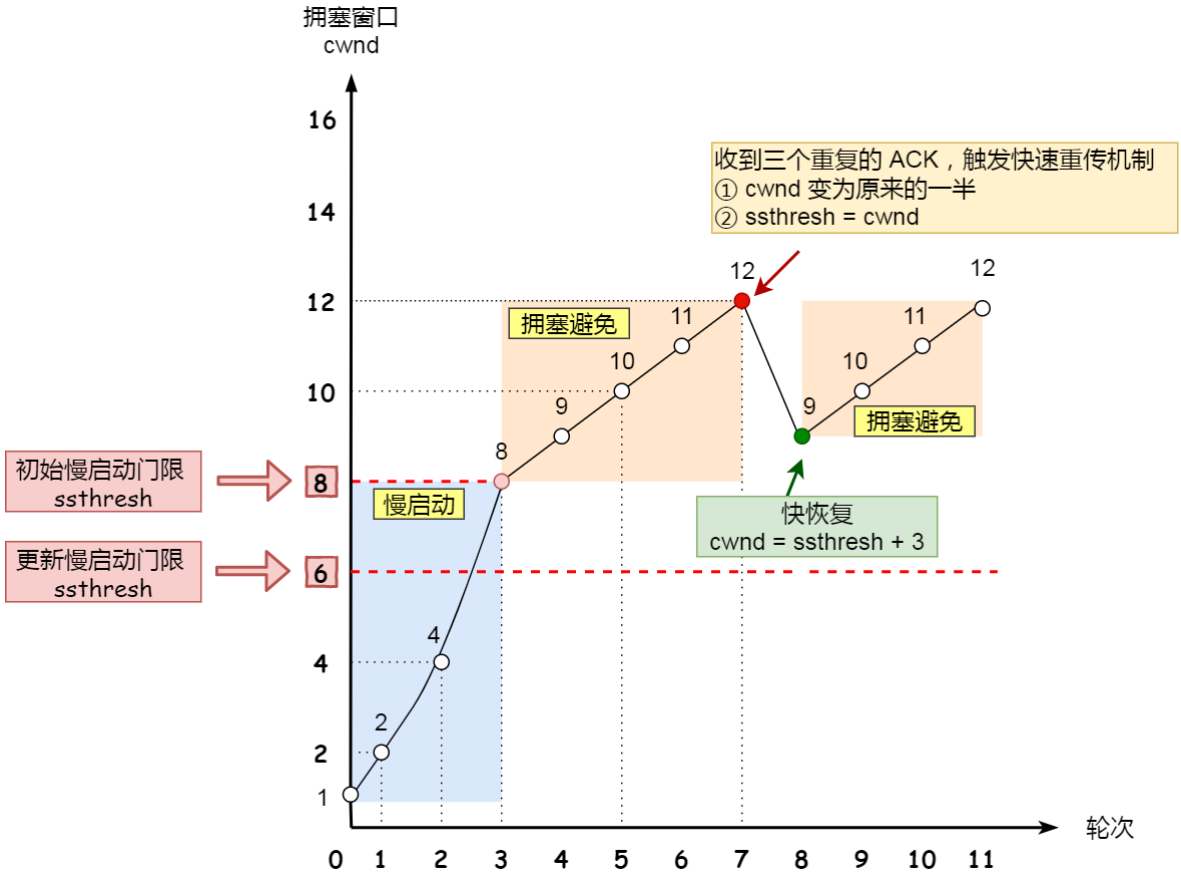
是在快速重传情况下触发的快速恢复算法

机制：

- cwnd = ssthresh + 3——即在未触发快速重传之前的cwnd，cwnd / 2+ 3（即下面的6+3）

  （主要是考虑到，能够收到3个ACK，说明接收方已经收到了后面的3个报文，网络中已经有3个报文离开了，所以它们只会出现在接收方的缓存中，可以适当将cwnd扩大一些，加快恢复原先状态）

- 重传丢失的数据包
- 如果再次收到重复的ACK，那么cwnd + 1（说明又有数据离开网络到达了接收方，所以窗口可以进一步增大）
- 如果收到新的ACK，表示该ACK已经确认了新的数据，那么重传的数据已经收到了，那么可以恢复之前的状态了，则将cwnd = ssthresh（即未触发快速重传之前的cwnd / 2），再次进入拥塞避免状态——即线性增加


拥塞窗口
cwnd
16
14
12
10
8
6
4
2
1
0 1 2 3 4 5 6 7 8 9 10 11
轮次
收到三个重复的 ACK，触发快速重传机制
① cwnd 变为原来的一半
② ssthresh = cwnd
初始慢启动门限
ssthresh
更新慢启动门限
ssthresh
慢启动
拥塞避免
拥塞避免
快恢复
cwnd = ssthresh + 3
1
2
4
8
9
10
11
12
9
10
11
12