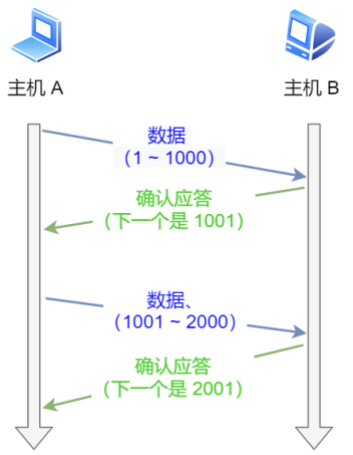
TCP--超时重传、流量控制、拥塞控制

重传机制

TCP实现可靠传输的方式之一，是通过序列号与确认应答。在TCP中，当发送端的数据到达接收主机的时候，接收端主机会返回一个确认应答消息ACK，表示已收到消息



但是错综复杂的网络中，并不能总是进行正常的数据传输，当数据在传输过程中丢失了，TCP就会启动重传机制来解决数据包丢失的问题

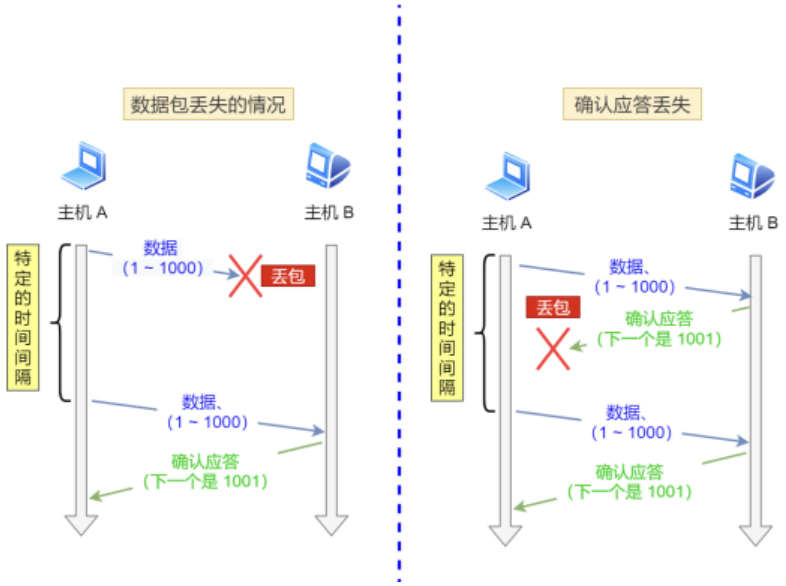
常见的重传机制有四种：超时重传、快速重传、SACK、D-SACK

超时重传

在发送数据时，设定一个定时器，当超过指定时间后，没有收到对方的ACK确认应答报文，发送方就会重新发送数据。

TCP会在下面两种情况下超时重传

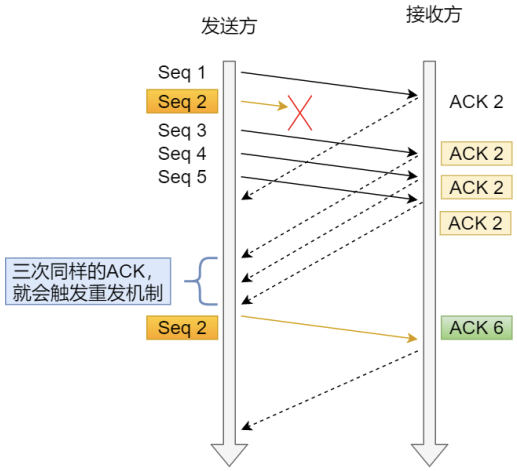
- 数据包丢失
- 确认应答丢失



缺点：超时重传存在的问题是，超时周期可能会比较长，导致重传时间较长
这时候就需要使用【快速重传】机制来解决问题

快速重传

快速重传不以时间为驱动，而是以数据驱动重传

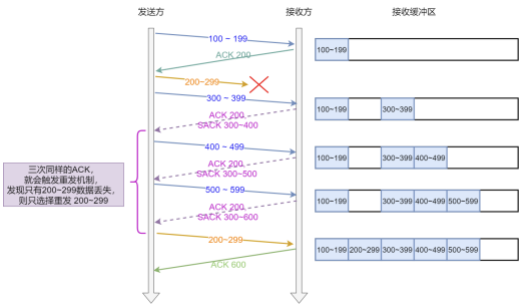


从图中可以很明显看到，发送方收到3个ACK2的回复，这时候发送方就会知道seq2没收到，在超时时间之前就会重传丢失的报文段。
不过快速重传只解决了一个问题，就是超时时间的问题，**但是它依然面临着另一个问题，就是重传的时候，是重传之前的一个还是重传所有的问题**

> 比如上面是重传2，还是重传2，3，4，5呢，因为发送方不知道这个连续的三个ACK2是谁传回来的，根据TCP的不同实现，这都是有可能的
> 为了解决不知道该重传哪些报文的问题，有了【SACK】方法

SACK

选择性确认：需要在TCP头部的【选项】字段里加一个SACK的东西，**它可以将缓存的地图(??信息)发送给发送方，这样发送方就知道哪些数据收到了，哪些数据没收到**，知道了这些信息，**就可以只重传丢失的数据。（SACK Block：记录丢失块后面收到的数据）**

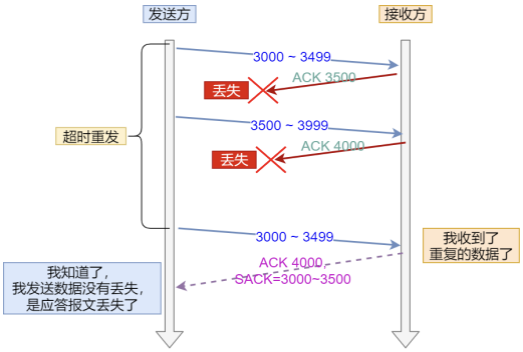


如图，发送方收到了三次同样的ACK报文，就会触发快速重传机制，通过SACK信息发现只有【200-299】这一段数据丢失，那么重发时只需要对这个TCP段进行重传就行

> 如果要支持SACK，需要双方都支持

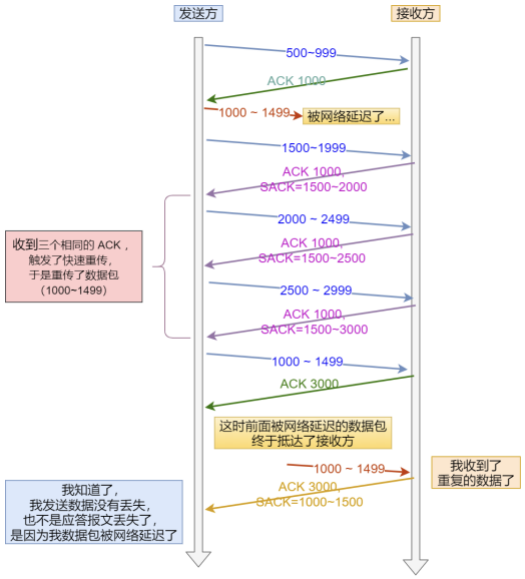
D-SACK

Duplicate-SACK，主要是使用SACK来告诉发送方哪些数据被重复接收了，举两个例子



ACK丢包

- 接收方发给发送方的两个 ACK 确认应答都丢失了，所以发送方超时后，重传第一个数据包（3000~3499）
- 于是接收方发现数据是重复收到的，于是回了一个 SACK = 3000~3500，告诉发送方3000~3500的数据早已被接收了，因为 ACK 都到了 4000 了，已经意味着 4000 之前的所有数据都已收到，所以这个SACK 就代表着 D-SACK 。
- 这样发送方就知道了，数据没有丢，是接收方的 ACK 确认报文丢了。



网络延时

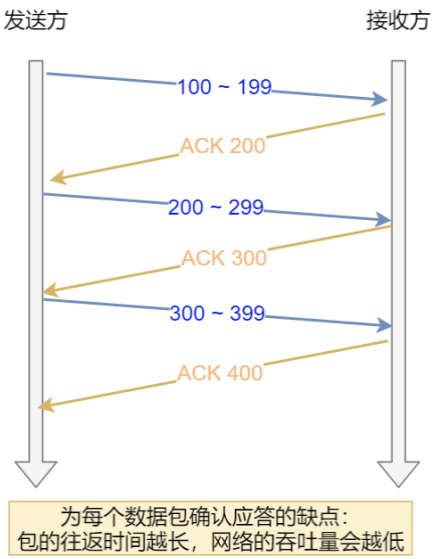
- 数据包（1000~1499） 被网络延迟了，导致发送方没有收到 ACK=1500 的确认报文。

- 而后面报文到达的三个相同的 ACK 确认报文，就触发了快速重传机制，但是在重传后，被延迟的数据包（1000~1499）又到了接收方；
- 所以「接收方」回了一个 SACK=1000~1500，因为 ACK 已经到了 3000，所以这个 SACK 是 D-SACK，表示收到了重复的包。
- 这样发送方就知道快速重传触发的原因不是发出去的包丢了，也不是因为回应的 ACK 包丢了，而是因为网络延迟了。

> 可见， D-SACK 有这么几个好处：
> 1. 可以让「发送方」知道，是发出去的包丢了，还是接收方回应的 ACK 包丢了；
> 2. 可以知道是不是「发送方」的数据包被网络延迟了；
> 3. 可以知道网络中是不是把「发送方」的数据包给复制了；

滑动窗口

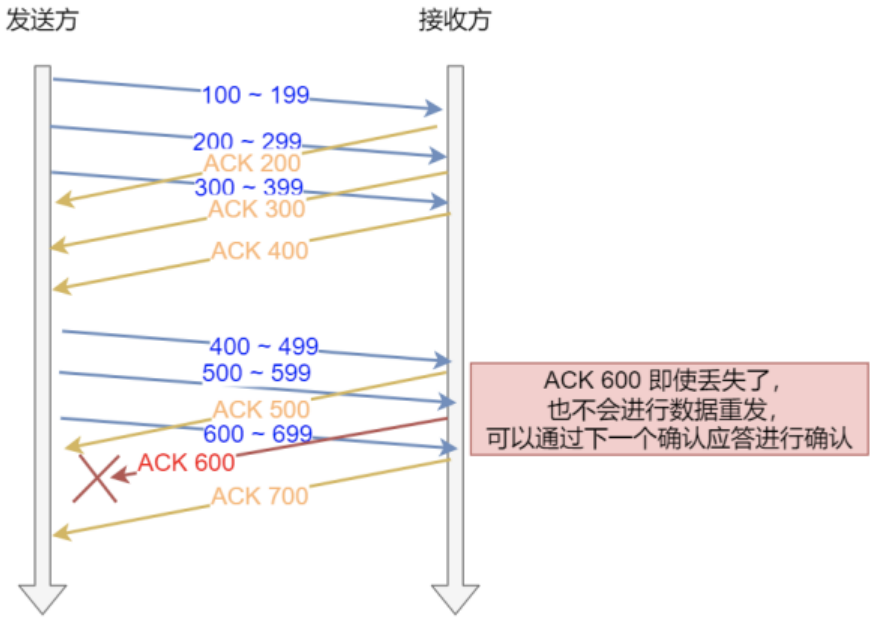
> 引入窗口概念的原因?
> 当TCP每发送一个数据时，都要进行一次确认应答，当收到一个应答后，再发送下一个TCP段，这样数据包的往返时间很长，通信的效率也会变得很低，所以引入窗口概念，类似消息队列或者缓冲区的概念



窗口大小就是指的是不需要等待确认应答，而可以继续发送数据的最大值
窗口的实现实际上是操作系统开辟的一个缓存空间，发送方主机在等待确认应答返回之前，必须在缓冲区中保留已发送数据，如果按期收到确认应答，那么此时数据就可以从缓存区清除

累计确认

假设窗口大小为 3 个 TCP 段，那么发送方就可以「连续发送」 3 个 TCP 段，并且中途若有 ACK 丢失，可以通过「下一个确认应答进行确认」。如下图：



> 如果TCP是每次发送一个数据都要进行一次应答确认，收到了上一个数据的ACK，再发送下一个数据，这种效率会比较低
> 这样的传输就有一个缺点：数据包的往返时间越长，通信的效率就越低。要解决这个问题，我们可以有累计确认这个方法

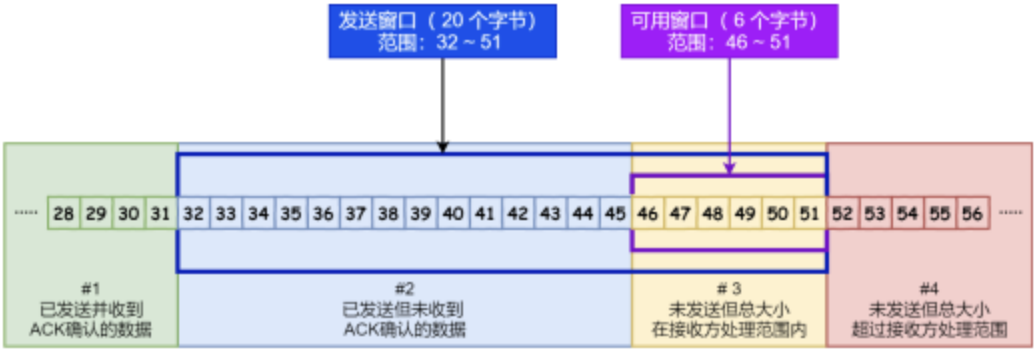
图中的 ACK 600 确认应答报文丢失，也没关系，因为可以通过下一个确认应答进行确认，只要发送方收到了 ACK700 确认应答，就意味着 700 之前的所有数据「接收方」都收到了。这个模式就叫**累计确认或者累计应答**。

窗口大小由哪一方决定?

> 在TCP头中有一个字段叫Window，就是窗口大小
> **这个字段是接收端告诉发送端自己还有多少缓冲区可以接收数据，于是发送端就可以根据这个接收端的处理能力来发送数据，而不会导致接收端处理不过来**
> **所以这个窗口大小由接收大窗口大小来决定。发送方数据不能超过接收方窗口大小，否则接收方就无法正常接收到数据**

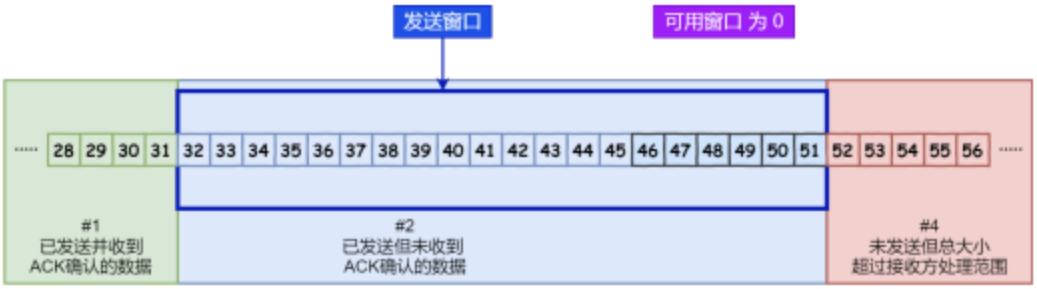
发送方的滑动窗口

我们先来看看发送方的窗口，下图就是发送方缓存的数据，根据处理的情况分成四个部分，其中深蓝色方框是发送窗口，紫色方框是可用窗口：

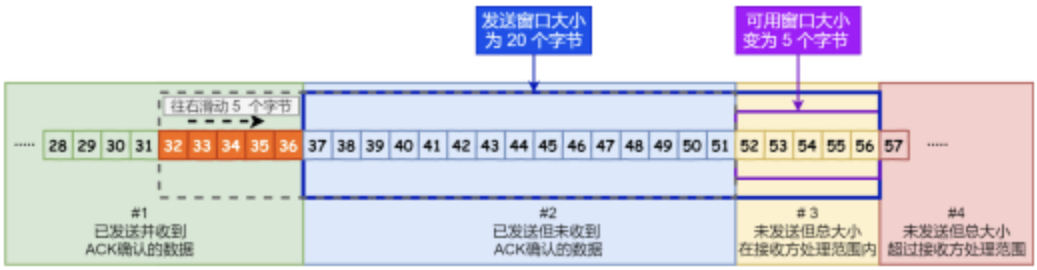


- #1 是已发送并收到 ACK确认的数据： 1~31 字节
- #2 是已发送但未收到 ACK确认的数据： 32~45 字节
- #3 是未发送但总大小在接收方处理范围内（接收方还有空间）： 46~51字节
- #4 是未发送但总大小超过接收方处理范围（接收方没有空间）： 52字节以后

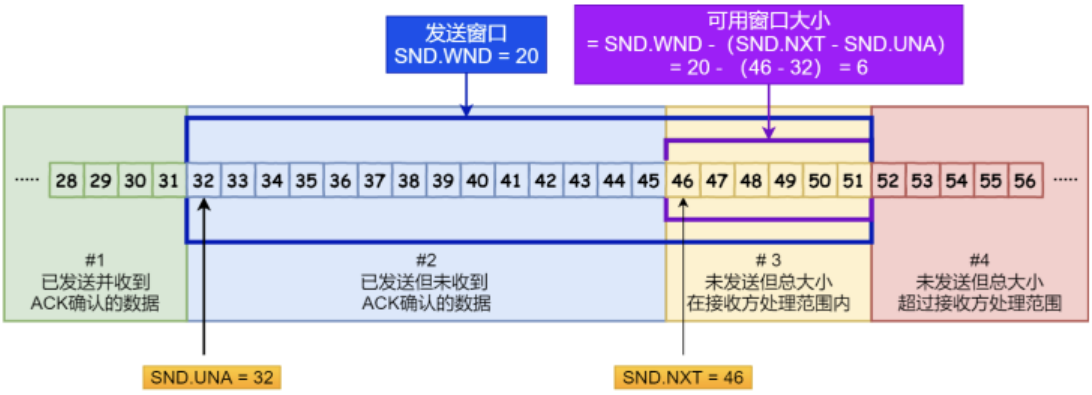
在下图，当发送方把数据「全部」都一下发送出去后，可用窗口的大小就为 0 了，表明可用窗口耗尽，在没收到ACK 确认之前是无法继续发送数据了。



在下图，当收到之前发送的数据 32~36 字节的 ACK 确认应答后，如果发送窗口的大小没有变化，则滑动窗口往右边移动 5 个字节，因为有 5 个字节的数据被应答确认，接下来 52~56 字节又变成了可用窗口，那么后续也就可以发送 52~56 这 5 个字节的数据了。



> 程序如何标识发送方的四个部分？？
> TCP 滑动窗口方案使用三个指针来跟踪在四个传输类别中的每一个类别中的字节。其中两个指针是绝对指针（指特定的序列号），一个是相对指针（需要做偏移）。



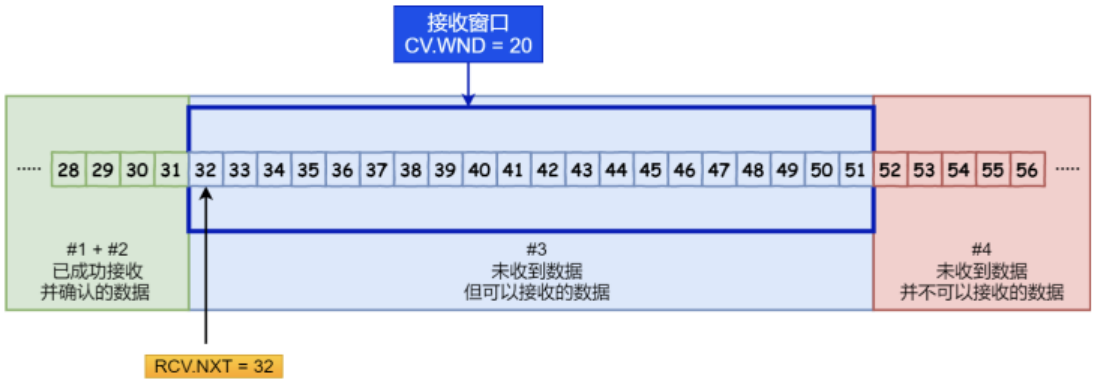
- SND.WND ： 表示发送窗口的大小（大小是由接收方指定的）；
- SND.UNA ： 是一个绝对指针，它指向的是已发送但未收到确认的第一个字节的序列号，也就是 #2 的第一个字节。
- SND.NXT ： 也是一个绝对指针，它指向未发送但可发送范围的第一个字节的序列号，也就是 #3 的第一个字节。
- 指向 #4 的第一个字节是个相对指针，它需要 SND.UNA 指针加上 SND.WND 大小的偏移量，就可以指向#4 的第一个字节了。

那么可用窗口大小的计算就可以是：可用窗口大小 = SND.WND –（SND.NXT – SND.UNA）

接收方的滑动窗口

接下来我们看看接收方的窗口，接收窗口相对简单一些，根据处理的情况划分成三个部分：

- #1 + #2 是已成功接收并确认的数据（等待应用进程读取）；
- #3 是未收到数据但可以接收的数据；
- #4 未收到数据并不可以接收的数据；



其中三个接收部分，使用两个指针进行划分：

- RCV.WND ： 表示接收窗口的大小，它会通告给发送方。

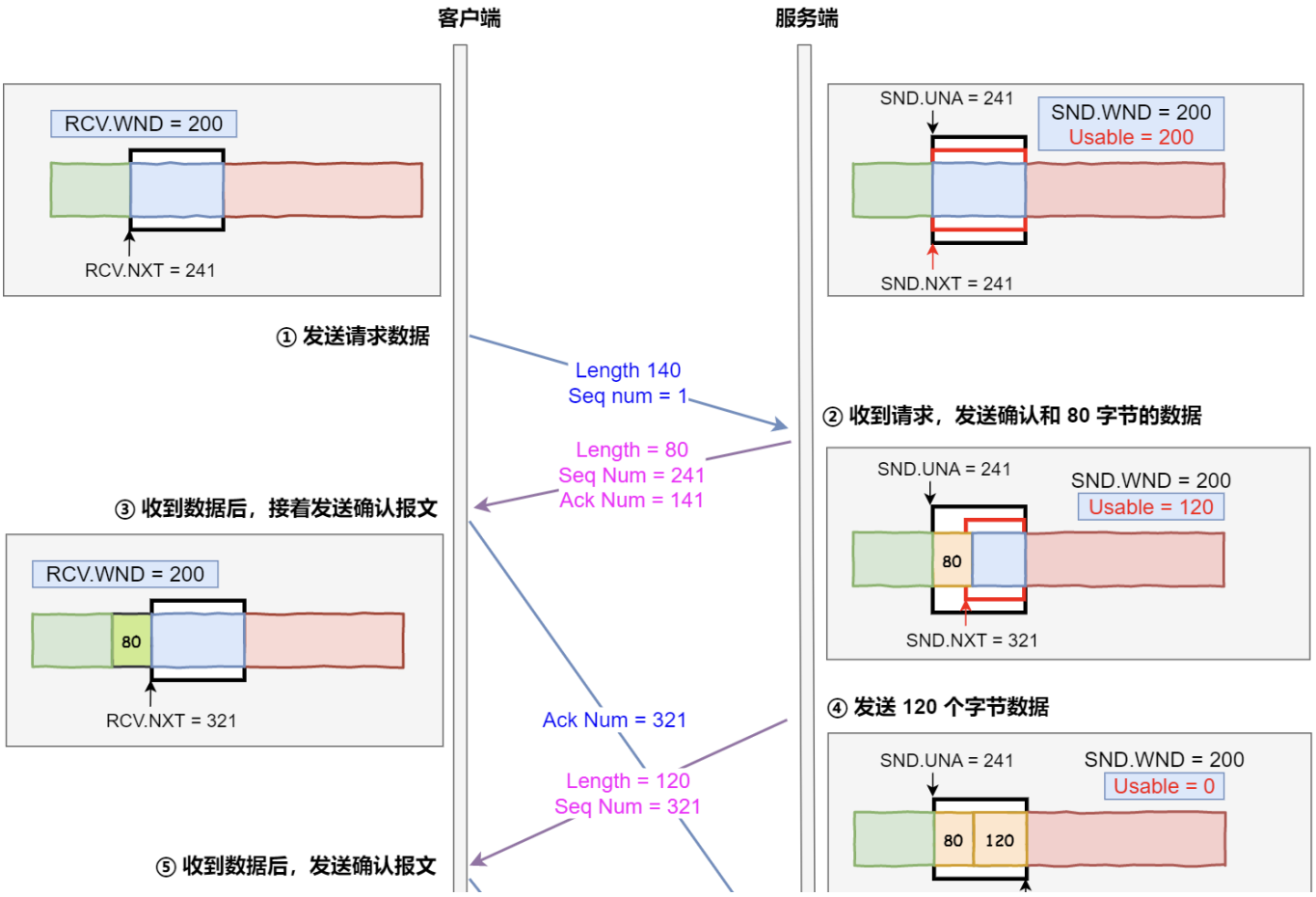
- RCV.NXT ： 是一个指针，它指向期望从发送方发送来的下一个数据字节的序列号，也就是 #3 的第一个字节。
- 指向 #4 的第一个字节是个相对指针，它需要 RCV.NXT 指针加上 RCV.WND 大小的偏移量，就可以指向#4 的第一个字节了。

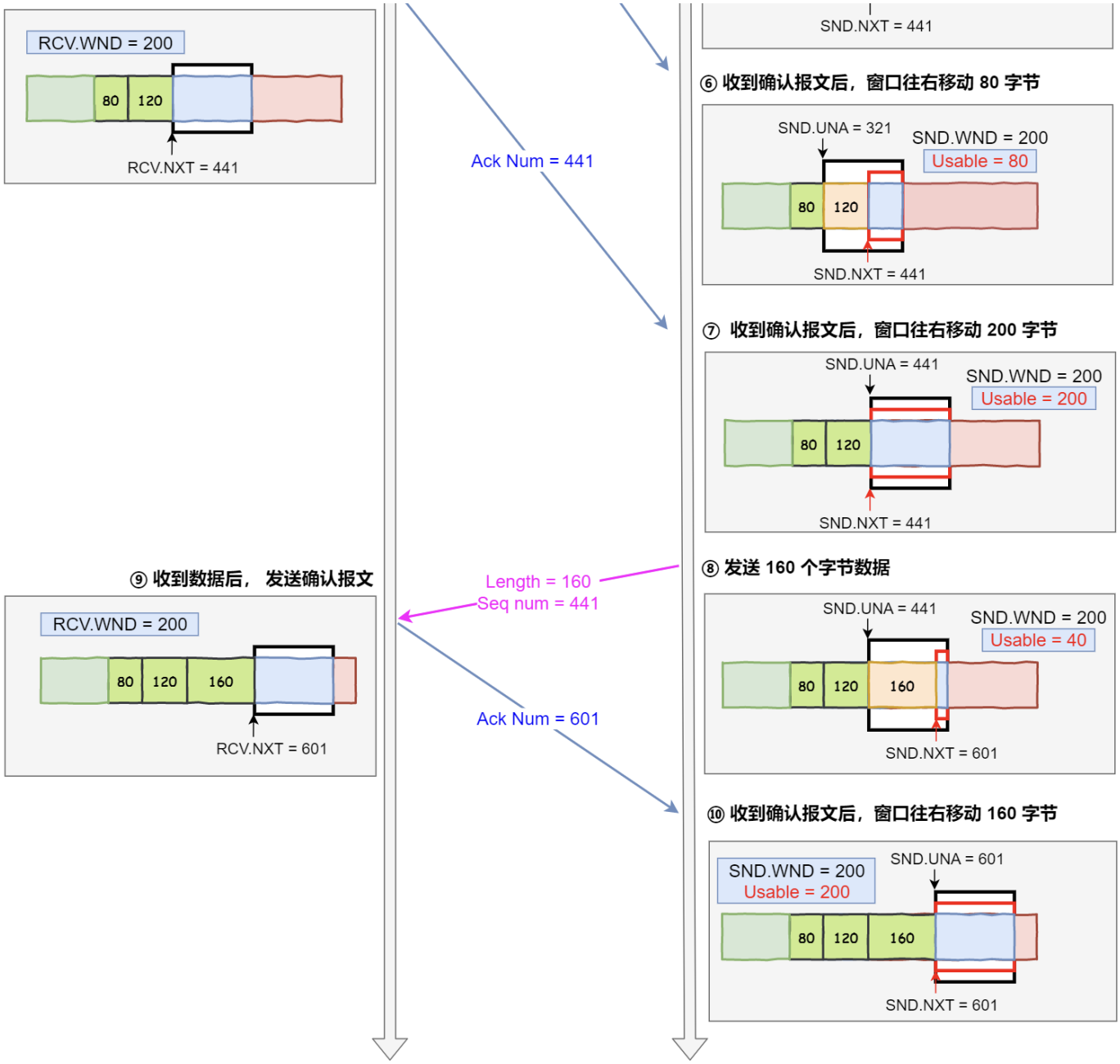
TCP保证每一个报文都能抵达对方，机制是这样的：报文发出后，必须接收到对方返回的确认报文ACK，如果很久都没收到，就会超时重传该报文，直到收到对方的ACK为止。**所以，TCP报文发出去后，并不会立马从内存中删除，因为在重传中还需要用到它**

> 接收窗口和发送窗口的大小是相等的吗?
> 并不是完全相等，是约等于，因为滑动窗口大小并不是一成不变的，当接收方的应用进程读取数据的速度非常快的话，这样接收窗口就可以很快空缺出来，那么新的接收窗口大小是通过TCP报文头中Window字段告诉发送方，这个传输存在延时，所以不是等于而是约等于

流量控制

发送方不能无脑发数据给接收方，需要考虑接收方处理能力，如果一直无脑发数据给对方，但对方处理不过来，那么就会导致触发重发机制，从而导致网络流量无端浪费
为了解决这个问题，**TCP提供一种机制可以让发送方根据接收方的实际接收能力控制发送的数据量，也就是流量控制**





根据上图的流量控制，说明下每个过程：

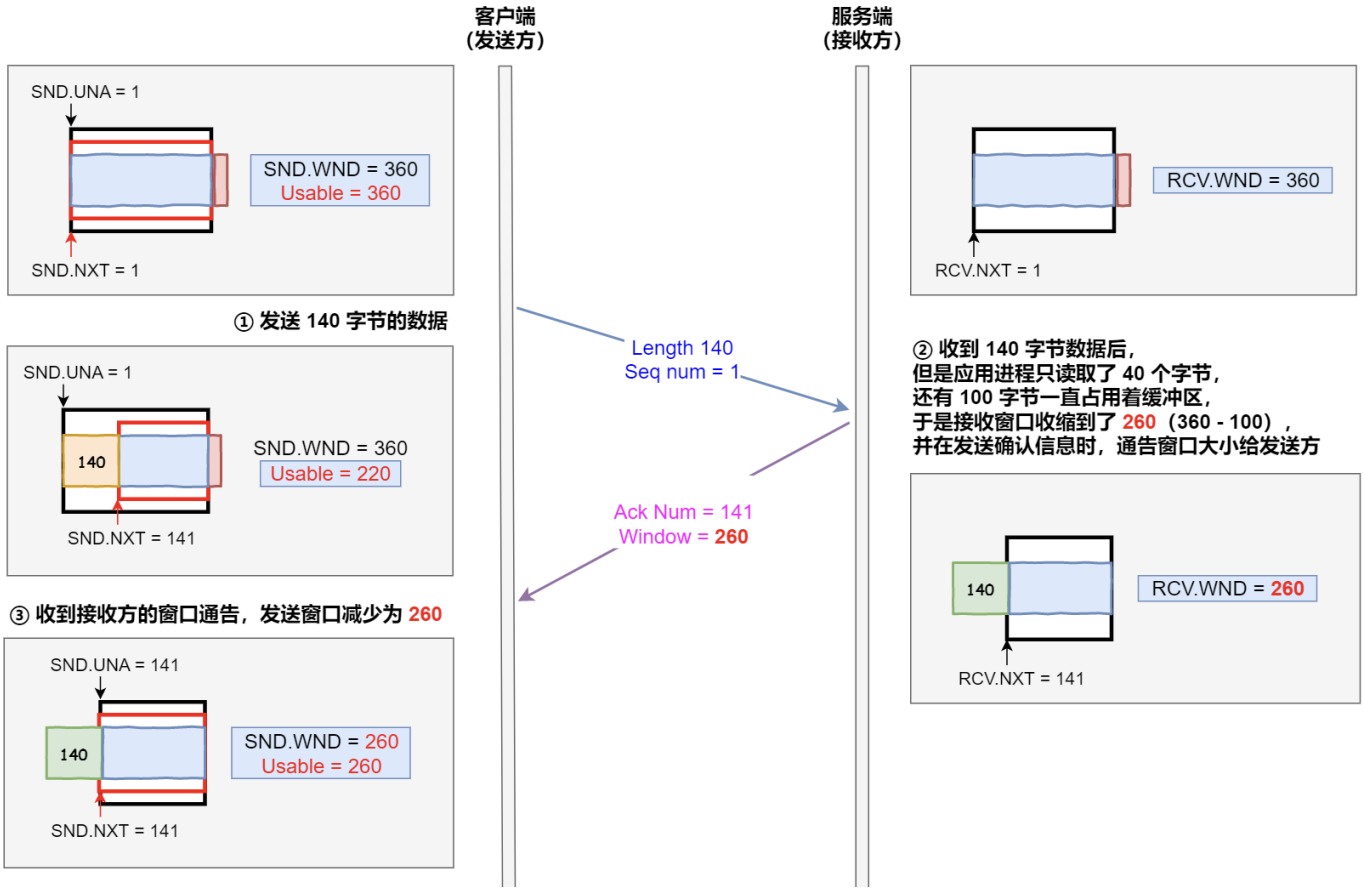
1. 客户端向服务端发送请求数据报文。这里要说明下，本次例子是把服务端作为发送方，所以没有画出服务端的接收窗口。
2. 服务端收到请求报文后，发送确认报文和 80 字节的数据，于是可用窗口 Usable 减少为 120 字节，同时SND.NXT 指针也向右偏移 80 字节后，指向 321， 这意味着下次发送数据的时候，序列号是 321。
3. 客户端收到 80 字节数据后，于是接收窗口往右移动 80 字节， RCV.NXT 也就指向 321， 这意味着客户端期望的下一个报文的序列号是 321，接着发送确认报文给服务端。
4. 服务端再次发送了 120 字节数据，于是可用窗口耗尽为 0，服务端无法再继续发送数据。
5. 客户端收到 120 字节的数据后，于是接收窗口往右移动 120 字节， RCV.NXT 也就指向 441，接着发送确认报文给服务端。

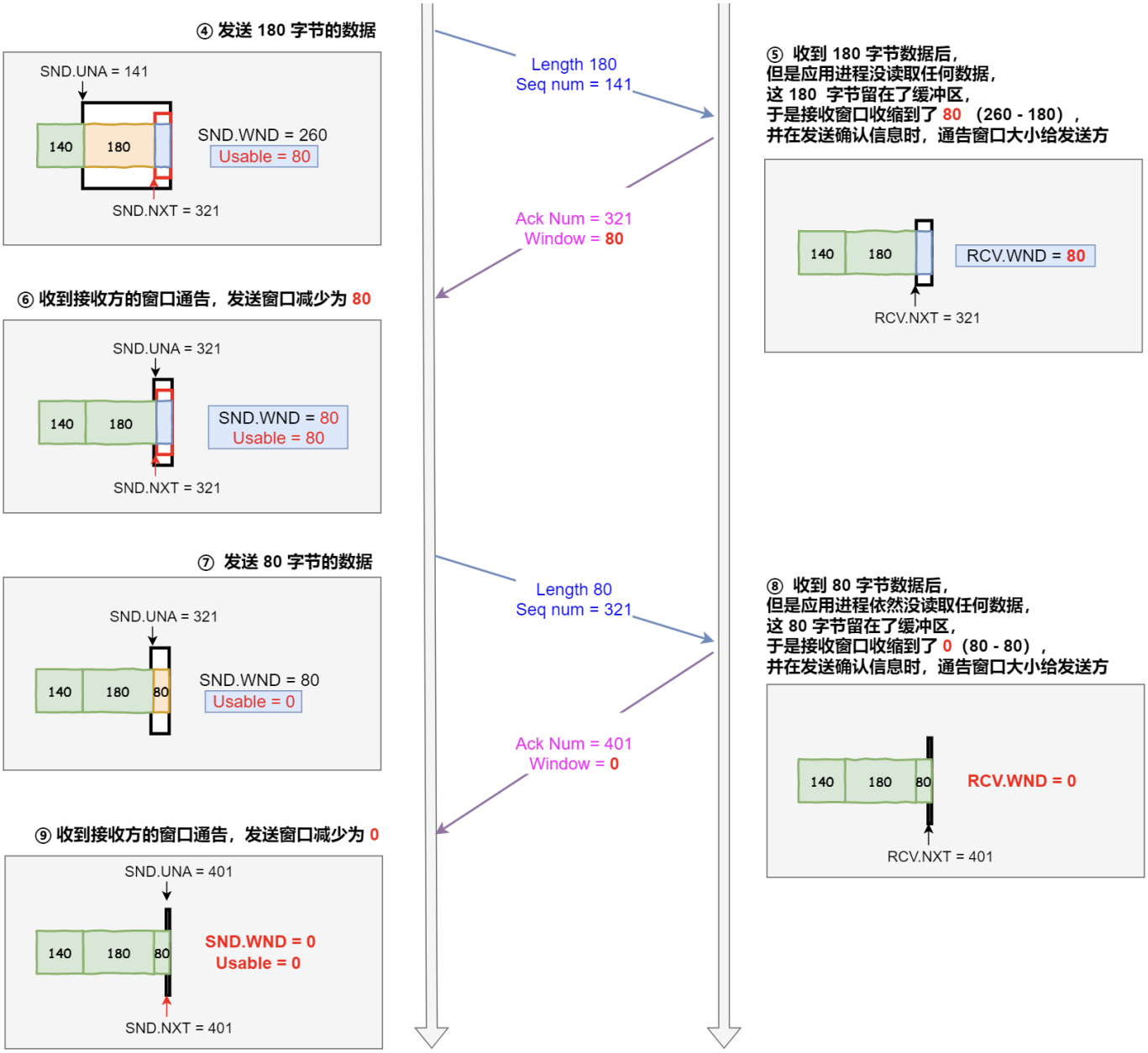
6. 服务端收到对 80 字节数据的确认报文后， SND.UNA 指针往右偏移后指向 321，于是可用窗口 Usable增大到 80。
7. 服务端收到对 120 字节数据的确认报文后， SND.UNA 指针往右偏移后指向 441，于是可用窗口 Usable增大到 200。
8. 服务端可以继续发送了，于是发送了 160 字节的数据后， SND.NXT 指向 601，于是可用窗口 Usable 减少到 40。
9. 客户端收到 160 字节后，接收窗口往右移动了 160 字节， RCV.NXT 也就是指向了 601，接着发送确认报文给服务端。
10. 服务端收到对 160 字节数据的确认报文后，发送窗口往右移动了 160 字节，于是 SND.UNA 指针偏移了160 后指向 601，可用窗口 Usable 也就增大至了 200。

操作系统缓冲区与滑动窗口的关系

上面例子中，假设发送窗口和接收窗口是不变的，但是实际上，发送窗口和接收窗口中所存放的字节数，都是存放在操作系统的内存缓冲区中的，而操作系统的缓冲区，会被操作系统调整，当应用进程无法及时读取缓冲区的内容时，也会对我们的缓冲区造成影响

> 例1：当应用程序没有及时读取缓存时，发生窗口和接收窗口的变化
> 考虑以下场景：
> - 客户端作为发送方，服务端作为发送方，发送窗口和接收窗口初始大小为360
> - 服务端非常地繁忙，当收到客户端数据时，应用层不能及时读取数据。





【这里还有没记录的】

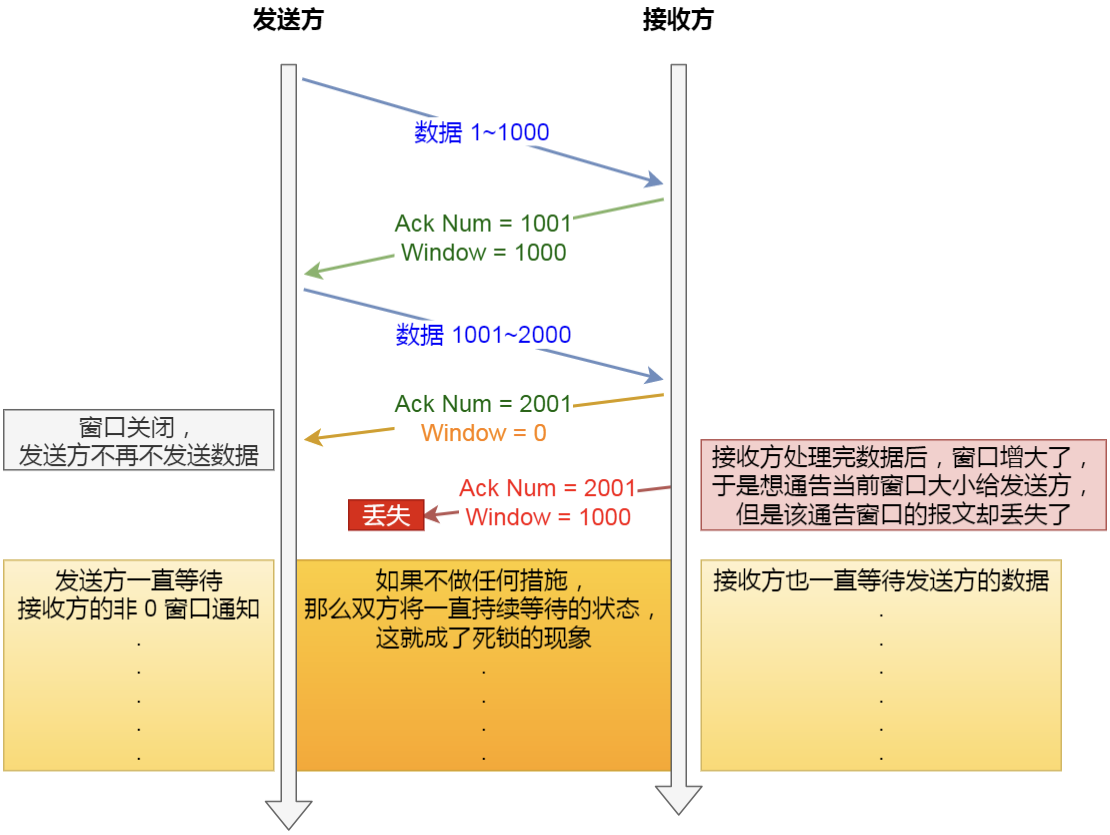
TCP规定不允许减少缓存的同时又收缩窗口的，而是采用先收缩窗口，过段时间再减少缓存，这样可以避免丢包情况

窗口关闭(零窗口通知)

如果窗口大小为0时，就会阻止发送方给接收方传递数据，这就是窗口关闭。直到窗口变成不是0，才会继续发送

> 窗口关闭潜在的危险
>
> 接收方向发送方通告窗口大小时，是通过ACK报文来通告的

> 当发生窗口关闭时，如果接收方已经处理完数据后，回复了一个非0ACK报文，如果这个ACK报文丢失，那么就很麻烦



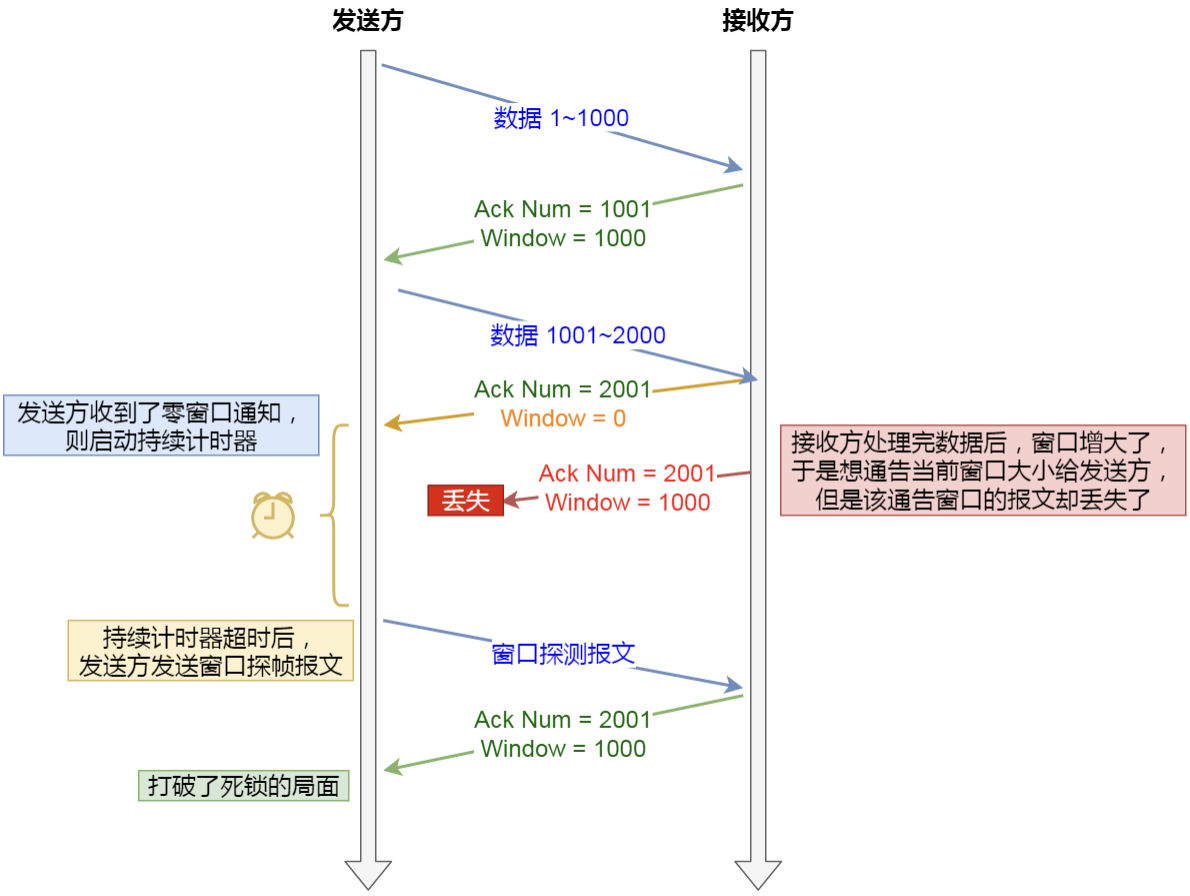
这就导致发送方一直在等待接收方的非0窗口通知，接收方也一直等待发送方的数据，如果不采取措施，这种相互等待的过程，就会造成死锁的现象。

窗口探测

> 如何解决窗口关闭时潜在的死锁现象？？

为了解决这个问题，TCP为每个连接设置一个持续定时器，只要TCP连接一方收到对方的零窗口通知，就启动持续计时器

如果持续计时器超时，就会发送窗口探测报文（window probe）报文，而对方在确认这个探测报文时，给出自己现在的接收窗口大小。



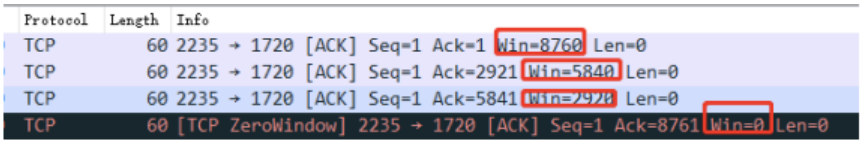
- 如果接收窗口仍然是0，那么收到这报文的一方就会重新启动持续计数器
- 如果接收窗口不是0，死锁就破解了

窗口探测次数一般为3次，每次大约30-60秒。如果**三次**过后还是收到接收窗口为0的话，有的TCP实现就会发RST报文来中断连接

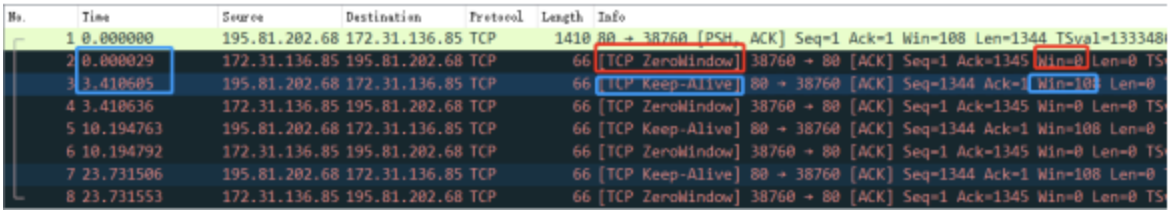
举例说明：

假如接收方处理数据的速度跟不上接收数据的速度，缓存就会被占满，从而导致接收窗口为0，当发送方接收到零窗口通知时，就会停止发送数据。

例子：如下图可以看到，接收方的窗口大小在不断收缩到0



接下来，发送方就会定时发送窗口大小的探测报文，以便及时指导接收方窗口的大小的变化。



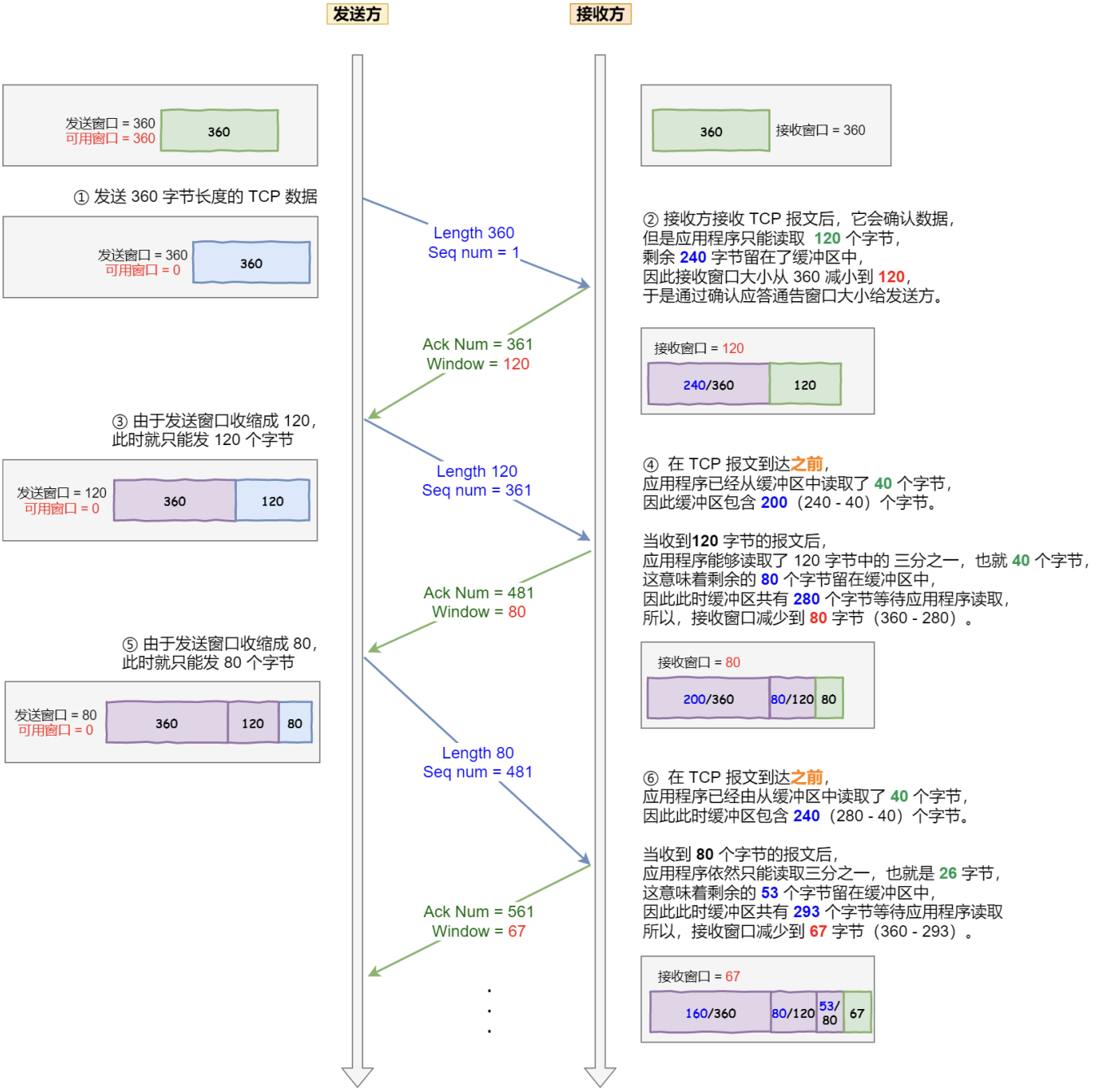
a. 发送方发送了一个数据包给接收方，接收方收到后，由于缓冲区被占满，回复了Win=0（零窗口通知）；
b. 发送方收到零窗口通知后，就不再发送数据了，过了3.4秒后，发送了一个TCP Keep-Alive（窗口大小探测报文）；
c. 接收方收到探测报文后，立马回复一个窗口通知，但此时还是Win=0；
d. 发送方知道接收窗口还是0之后，继续等待了6.8（翻倍）秒，再次发送Keep-Alive报文，接收方还是回复了Win=0；
e. 同样，等待13.5（继续翻倍）秒后，再次发送Keep-Alive报文，接收方还是回复了Win=0；

糊涂窗口综合征

如果接收方太忙了，来不及取走接收窗口里的数据，那么就会导致发送方的发送窗口越来越小。到最后，如果接收方腾出几个字节并告诉发送方现在有几个字节的窗口，而发送方会义无反顾发送这几个字节，这就是糊涂窗口综合征。
因为TCP+IP头部有40个字节的开销，如果每次传输的字节太小，总的传输代价太大，就相当于，一个可以装50人的大巴车，每次上两个人就发车，不划算
不过大巴车司机可以设定如果到达了25人，就选择发车，这就解决了这个问题。

现举个糊涂窗口综合症的栗子，考虑以下场景：
接收方的窗口大小是 360 字节，但接收方由于某些原因陷入困境，假设接收方的应用层读取的能力如下：

- 接收方每接收 3 个字节，应用程序就只能从缓冲区中读取 1 个字节的数据；
- 在下一个发送方的 TCP 段到达之前，应用程序还从缓冲区中读取了 40 个额外的字节；



可以发现，由于应用程序没法及时使用数据，每个阶段窗口都在不断减少，发送的数据也越来越小

所以，糊涂窗口综合征现象是可以发生在发送方和接收方的

- 接收方可以通告一个小窗口
- 发送方可以发送小数据

所以解决方法可以是

- 接收方不通告小窗口
- 发送方不发送小数据

接收方通常策略是：当窗口大小小于min（MSS，缓存空间/2），也就是小于MSS与1/2缓存大小中的最小值时，就会向发送方通告窗口为0，也就阻止了发送方再发数据过来，等到接收方处理了一些数据

之后，窗口大小>=MSS，或者接收方缓存空间有一半可以使用，就可以把窗口打开让发送方发送数据过来。

发送方策略是使用Nagle算法，思路是延时处理，满足下面其中之一才可继续传输

- 要等到窗口大小 >= MSS 或是 数据大小 >= MSS
- 收到之前发送数据的 ack 回包

没满足时，发送方就一直囤积数据，直到满足条件

延迟确认与Nagle算法

当TCP报文承载的数据非常小的时候，可能只有几字节，由于每个TCP报文的头部有20字节，还有20字节的IP头部，真正有效数据比重非常低。

两种处理方案：

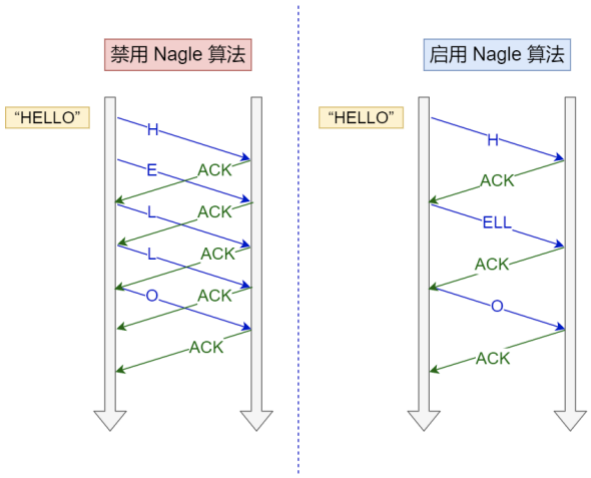
- Nagle算法
- 延迟处理

Nagle算法（发送方）

Nagle如何避免大量TCP小数据报文的传输？？策略：

- **没有已发送未确认**报文时，立刻发送数据；
- 存在已发送未确认报文时，直到**没有已发送未确认报文**或者**数据长度达到MSS**，再发送数据。

只要没满足上面条件中的一条，发送方一直在囤积数据，直到满足上面的发送条件。



简单说明一下右侧发送数据的过程：

1. 最开始没有已发送未确认的报文，立刻发送H字符
2. 然后，在还没有收到对H字符的确认报文，发送方就一直在囤积数据，直到收到了确认报文后，这时候没有了已发送未确认报文，于是将囤积的ELL字符一起发给了接收方
3. 最后，等到收到了ELL字符的ACK，将最后的O字符发送出去

小结：Nagle算法一定会有一个小报文，在最开始进行数据发送的时候。由于Nagle算法有囤积数据的过程，如果需要小数据包交互的场景的程序，比如，telnet或ssh这样的交互性比较强的程序，需要将Nagle算法关闭（默认是打开的），可以在Socket中设置`TCP_NODELAY`选项来关闭这个算法（关闭Nagle没有全局参数，需要根据每个应用自己的特点来关闭）

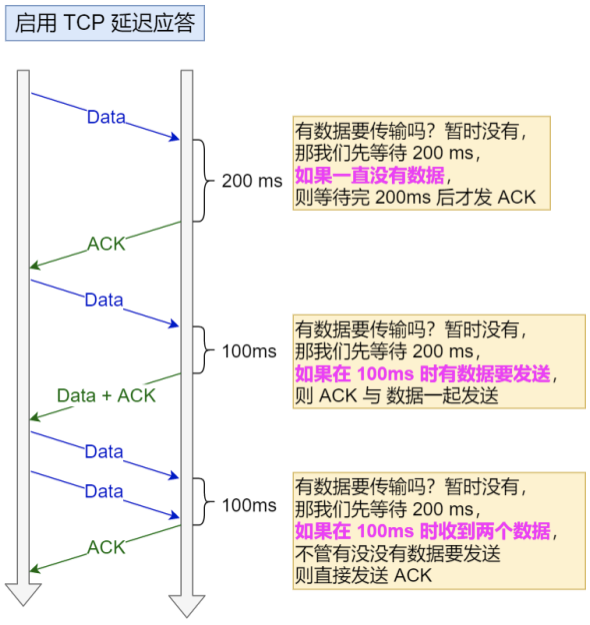
```
# 关闭 Nagle 算法
setsockopt(sock_fd, IPPROTO_TCP, TCP_NODELAY, (char *)&value, sizeof(int));
```

延迟确认（接收方）

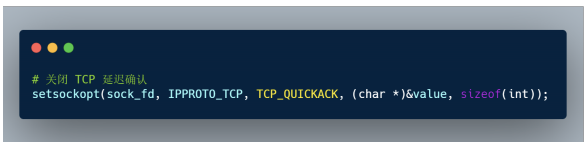
其实不仅仅是发送报文会有承载数据比重低的问题，事实上当没有携带数据的ACK，它的网络效率也是很低的，只有IP头部和TCP头部40字节，但是却没有携带数据报文，为了解决ACK传输效率低的问题，所以就衍生出了TCP**延迟确认**。

策略：

- 当有响应数据要发送时，ACK会随着响应数据一起立刻发送给对方
- 当没有响应数据要发送时，**ACK会延迟一段时间**，来等待是否有响应数据可以一起发送
- 如果在延迟等待时，对方第二个数据报文到达了，此时立刻发送ACK



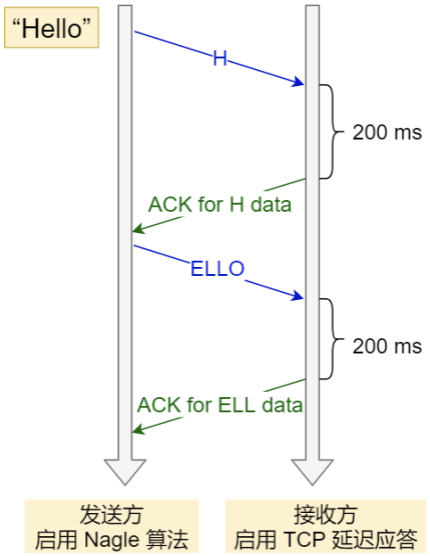
TCP延迟确认可以在Socket设置TCP_QUICKACK选项来启闭这个算法



```
# 关闭 TCP 延迟确认
setsockopt(sock_fd, IPPROTO_TCP, TCP_QUICKACK, (char *)&value, sizeof(int));
```

混合使用

当TCP延迟确认和Nagle算法混合使用时，会导致时耗增长



当发送方使用Nagle算法，接收方使用延迟确认会发生如下过程：

1. 开始由于没有已发送未确认的报文，发送方会立刻发送一个小报文；
2. 接收方接收到了后，由于接受方没有需要发送的数据，所以一直等待发送方的下一个报文到达；
3. 此刻的发送方由于没有等到前一个报文的ACK，而其他的数据还是小数据时，必须等待ACK否则无法发送下一个报文；
4. 接收方在等待了200ms后，回复ACK报文，发送方收到第一个报文的确认报文后，开始后续的发送

小结：由于两个算法混合使用，会导致发送方和接收方增加了额外的时延，会使得网络变得很慢，解决方案只有两种方法：

1. 发送方关闭Nagle算法；
2. 接收方关闭TCP延迟确认

拥塞控制

> 为什么有了流量控制了还需要需要流量控制?
> 流量控制是避免发送方数据填满接收方缓存，但是并不知道网络中具体发生了什么，一般来说，计算机网络都处在一个共享的环境中，因此也有可能会因为其他主机之间的通信导致网络拥堵

在网络出现拥堵的时候，如果继续发送大量的数据包，可能会导致数据包时延、丢失等，这时TCP就会重传数据，但是一旦重传数据就会导致网络中的负担更重，于是导致了更大的延迟和更多的丢包，这情况就会进入恶性循环被不断放大，所以TCP不能忽略网络上发生的事情，TCP设计者将其设计成一个无私的协议，当**网络发送拥塞时，TCP会自我牺牲，降低发送的数据量。**

于是，就有了**拥塞控制，控制的目的就是为了避免发送方的数据填满整个网络**

为了在**发送方**调节需要发送数据的量，定义了一个叫做【拥塞窗口】的概念（流量控制是接收方控制滑动窗口大小，发送方被动调整，拥塞控制是发送方主动控制）

拥塞窗口

> 什么是拥塞窗口，和发送窗口有什么关系？？

拥塞窗口cwnd是发送方维护的一个状态变量，它会根据网络的拥塞程度动态进行变化的

前面的发送窗口swnd和接收窗口rwnd是约等于关系，加入拥塞窗口概念后，发送窗口swnd的值为swnd=min(cwnd,rwnd)，也就是取拥塞窗口和接收窗口中的最小值。

拥塞窗口变化规则是：

- 只要网络中没有出现拥塞，cwnd就会增大
- 网络中出现了拥塞，cwnd减小

当发送方没有在规定时间内收到ACK应答报文，也就是发生了超时重传，就会认为网络出现了拥塞

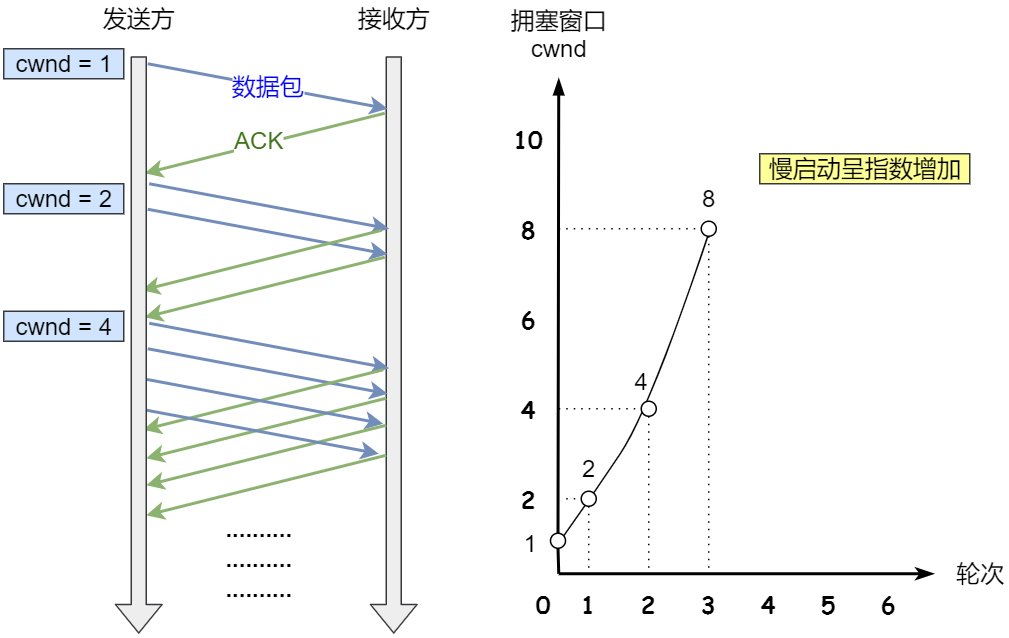
拥塞控制的主要四个算法如下

慢启动

在TCP连接刚建立是，首先是慢启动的过程，慢启动的意思就是一点一点提高发送数据包的数量（如果一开始就发大量数据，直接就会堵住网络）

慢启动规则：当发送方每收到一个ACK后，拥塞窗口cwnd的大小就会加1

这里假设拥塞窗口和发送窗口相等，举个例子



- 连接建立后，一开始初始化cwnd=1，表示可以传一个MSS（最大报文段长度）大小的数据

- 当收到1个ACK确认应答之后，cwnd增加1，于是一次就可以发送两个
- 当收到2个ACK确认应答之后，cwnd增加2，于是下次可以比这次增加两个，也就是四个
- 当收到4个ACK确认应答之后，每个ACK会让cwnd增加1，4个确认cwnd会增加4，所以下一次就可以发8个

可以看出，慢启动算法，发送方发送包的个数会指数增长（就记住1个ACK会让cwnd增加1）

> 那么慢启动涨到什么时候是个头?

这里有个慢启动门限ssthresh（slow start threshold）状态变量

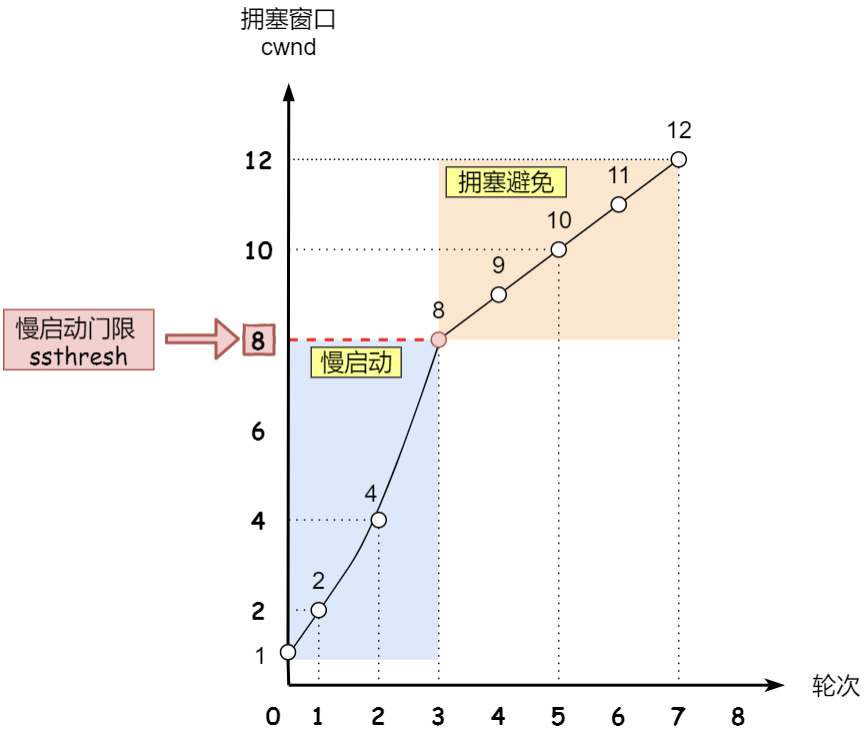
- 当cwnd<ssthresh，使用慢启动算法
- 当cwnd≥ssthresh，就需要使用下面的【拥塞避免算法】

拥塞避免

一般来说ssthresh的大小是65535字节。

那么进入拥塞避免算法后，它的规则是：每收到一个ACK时，cwnd增加1/cwnd

继续接慢启动的例子，假设ssthresh为8



- 当8个ACK应答确认来的时候，每个确认增加1/8，8个ACK确认会让cwnd增加1，于是下一次发送可以发送窗口就是8+1=9个MSS，变成了线性增长
- 此时要记得拥塞避免算法中，拥塞窗口cwnd还是在增加的，但变成了线性增长，一直增长，网络就会慢慢进入拥塞状态，**于是就会出现丢包现象，这时候就会触发重传机制**，这时候就会进入【拥塞发生算法】

拥塞发生

当网络出现拥塞时，也就是会发生数据包重传，这里重传主要是两种

- 超时重传
- 快速重传

对于这两种重传来说，拥塞发送算法是不同的，下面分别来说

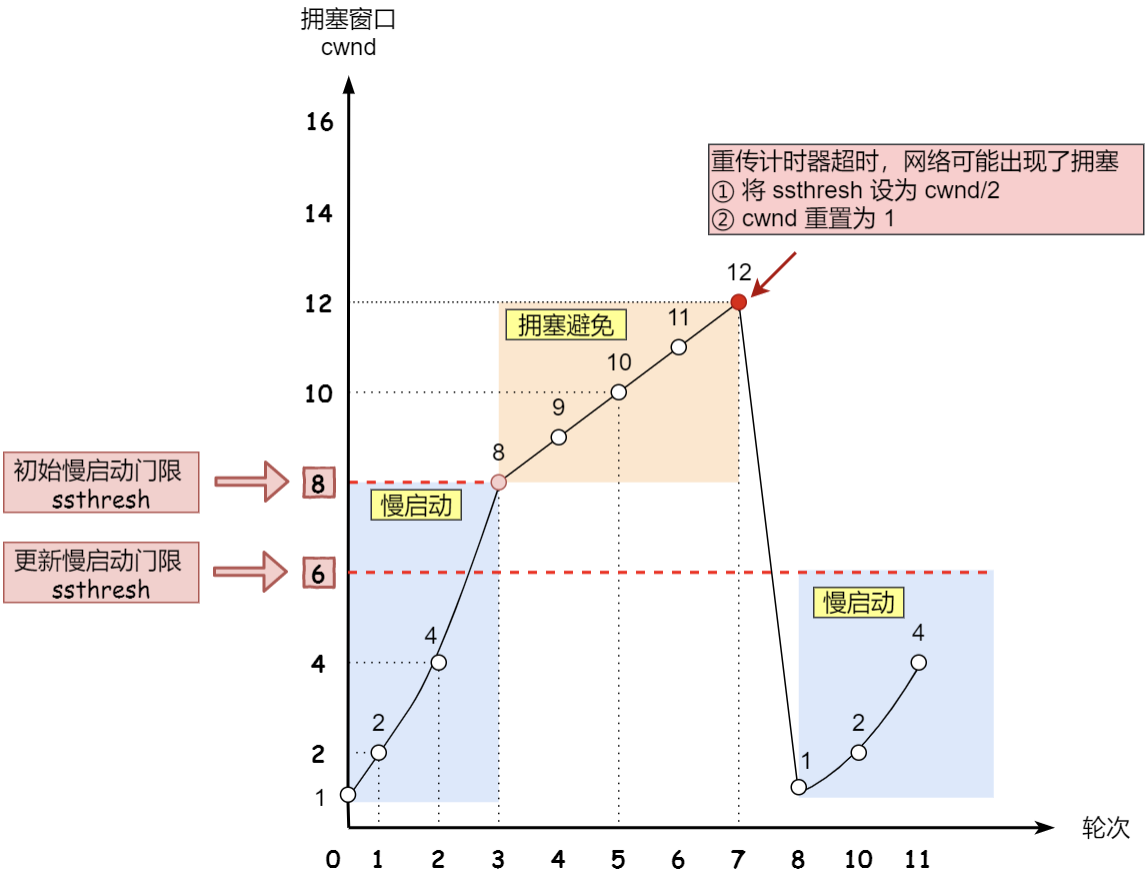
超时重传时的拥塞发生算法

发生超时重传后，就会使用拥塞发生算法。

这时候，ssthresh和cwnd都会发生变化

- ssthresh会变成ssthresh/2
- cwnd会被重置为1

继续举上面的例子



- 接着重新继续慢启动，这里慢启动是突然减少数据流的，这方式太过激进，反应也很强烈，会造成网络卡顿。

快速重传的拥塞发生算法

快速重传是在接受方发现丢了一个包后，发三次前一个包的ACK来表示下一个想收到包，于是发送方会快速进行重传，不必等到超时重传

TCP认为这种情况不严重，因为大部分没丢，只是丢了一部分，所以ssthresh和cwnd变化如下

- cwnd=cwnd/2 （= 6）
- ssthresh = cwnd （= 6）

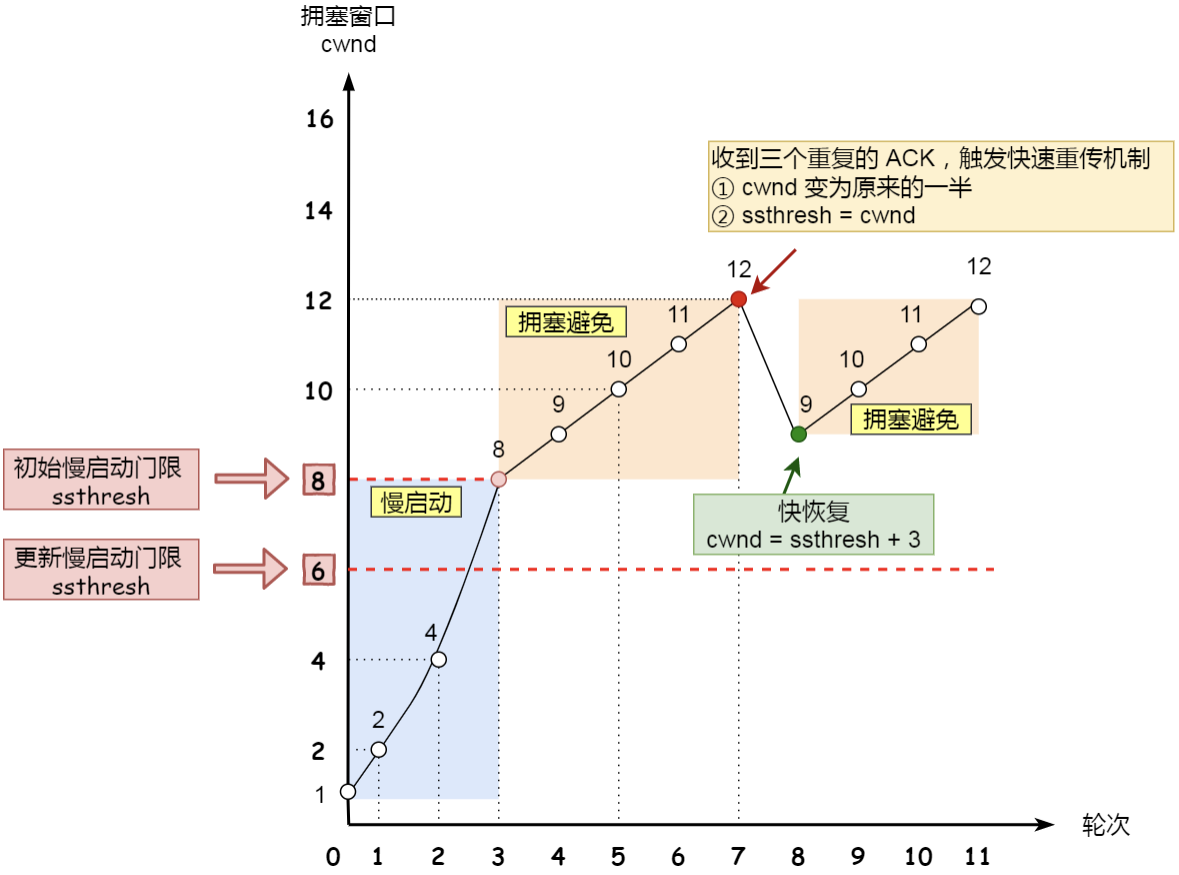
然后进入【快速恢复算法】

快速恢复

快速重传和快速重传算法一般同时使用

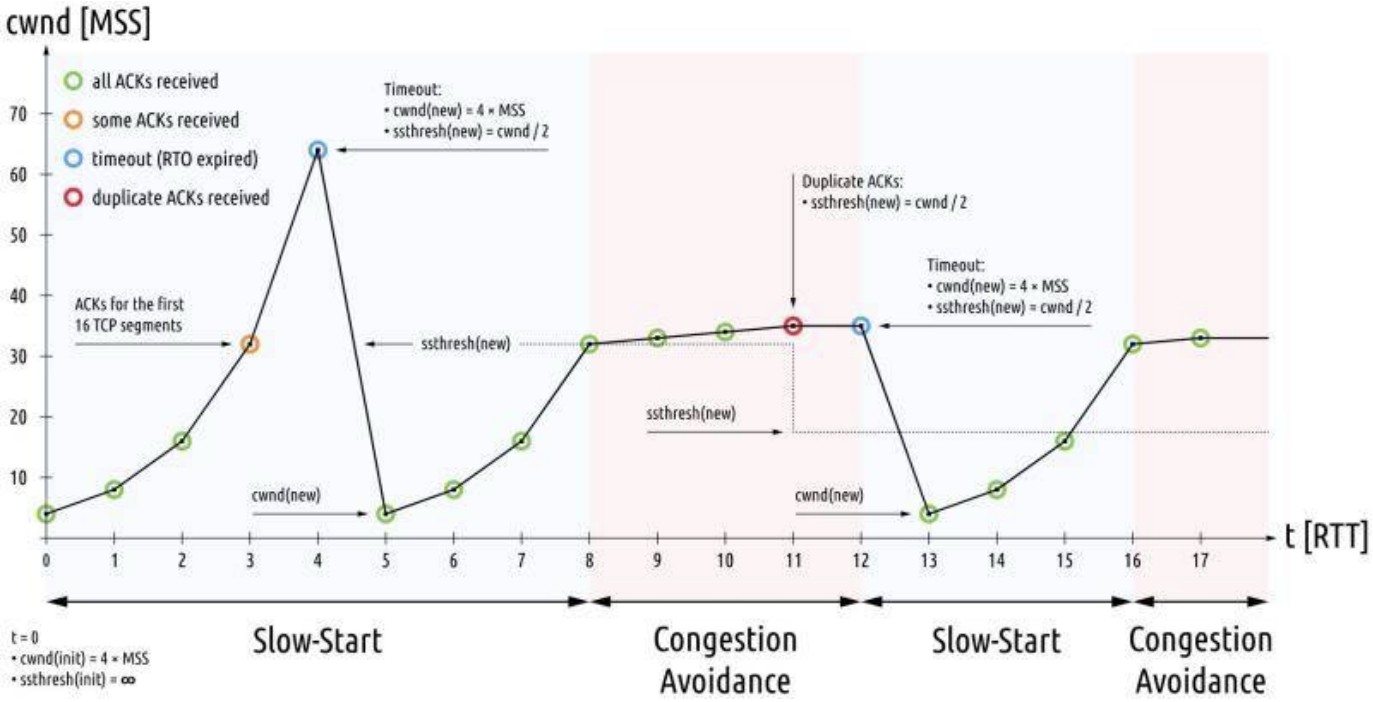
快恢复算法是认为，发送方还能收到三个重复ACK，说明网络也不至于那么糟糕，所以没必要和超时重传时那么夸张，直接将cwnd重置

还是继续上面例子来讲解快恢复算法



- cwnd = ssthresh+3（3的意思是有那三个ACK，会增加3）
- 重传丢失的数据包
  - 如果再收到重复的ACK，那么cwnd+1
  - 如果收到新数据的ACK，将cwnd设置为第一步中ssthresh的值，原因是该ACK确认了新的数据，说明从重复ACK后的数据都已收到，该恢复过程结束，可以回到恢复之前的状态了，也就是在此进入拥塞避免状态

整个拥塞控制的图如下


cwnd [MSS]
all ACKs received
some ACKs received
timeout (RTO expired)
duplicate ACKs received
Timeout:
• cwnd(new) = 4 × MSS
• ssthresh(new) = cwnd / 2
Duplicate ACKs:
• ssthresh(new) = cwnd / 2
Timeout:
• cwnd(new) = 4 × MSS
• ssthresh(new) = cwnd / 2
ACKs for the first
16 TCP segments
ssthresh(new)
ssthresh(new)
cwnd(new)
cwnd(new)
t [RTT]
t = 0
• cwnd(init) = 4 × MSS
• ssthresh(init) = ∞
Slow-Start
Congestion Avoidance
Slow-Start
Congestion Avoidance