计算机网络

jask

2024 年 12 月 29 日

1 计算机网络基础部分

1.1 协议和服务的区别

- 协议：语法：交换的信息的格式语义：发送者或接收者所要完成的操作，即需要发出何种控制信息，完成何种动作以及做出何种响应同步：收发双方的时序关系，时间实现顺序的详细说明
- 服务：是纵向的底层到上层提供服务。但服务是“垂直的”，即服务是下层通过层间接口向上层提供的。上层使用下层所提供的服务必须通过与下层交换一些命令，这些命令在 OSI 中称为服务原语。
- 协议和服务的区别：
  - 协议的实现保证了能够像上一层提供服务。本层的服务用户只能看见服务而无法看见下面的协议。
  - 协议是“水平的”，即协议是控制两个对等实体之间交换的帧、报文、分组的格式和意义的规则，实体利用协议来实现他们的服务定义。
  - 服务通过协议完成，协议通过服务体现；比喻：服务代表功能、能力；协议代表标准化或者规则。

1.2 TCP/IP 协议

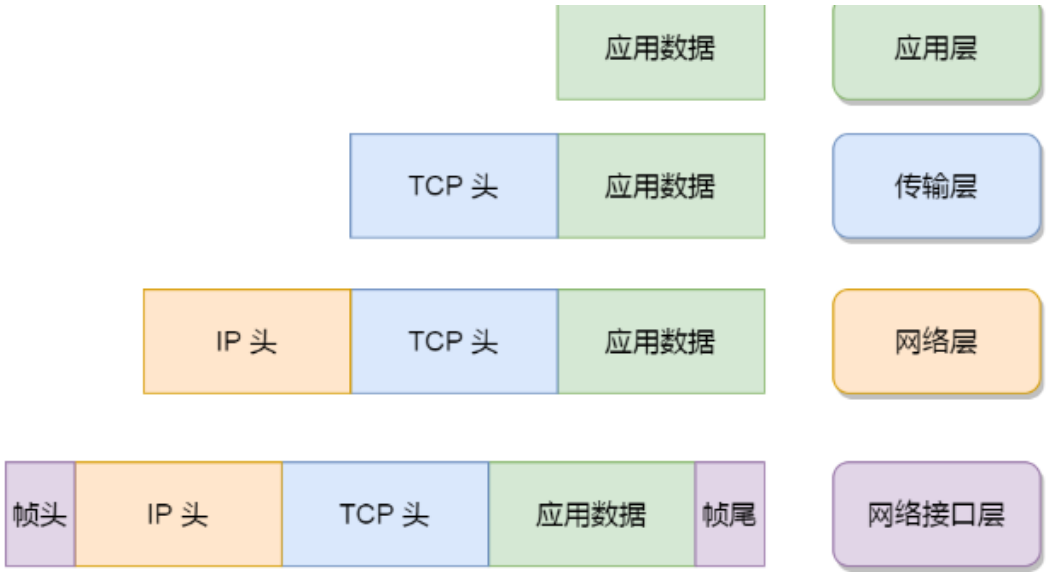
- TCP/IP 的分层

1. 应用层：只需要专注于给用户提供应用功能，如 HTTP, FTP, Telnet, DNS 等。应用层不关心数据是如何传输的，并且在操作系统的用户态运行，传输层及以下在操作系统内核态运行。
2. 传输层：为应用层提供网络支持，有 TCP 和 UDP 两个传输协议。TCP：Transmission Control Protocol, 拥有注入流量控制、超时重传、拥塞控制等特性。UDP：只负责发送数据包，不确保是否能够抵达，实时性相对较好，传输效率更高，也可以实现可靠传输。

   应用传输的数据可能会很大，直接传输不好控制，当数据包的大小超过 MSS 时，就需要将数据包分块，这样即使丢失了一个分块也只需要重新发送这一个分块，不需要重新发送整个数据包，每一个分块叫做一个 TCP 段。
3. 网络层：网络层负责将一个数据的设备传输给另一个设备，起到实际的传输功能。最常用的是 IP 协议，IP 协议会将传输层的报文作为数据部分，再加上 IP 包头组装成 IP 报文，如果 IP 报文大小超过 MTU(以太网中一般为 1500 字节) 就会再次分片，得到一个即将发送到网络的 IP 报文。IP 地址分成两种意义：
   - 网络号：负责标识该 IP 地址数据哪个子网；
   - 主机号：负责标识同一子网下的不同主机；

怎么区分就需要用到子网掩码。比如 10.100.122.0/24, 后面的/24 就是 255.255.255.0 子网掩码一共 24 个 1。知道了子网掩码就可以将 10.100.122.0 与 255.255.255.255 进行位运算，a 就可以得到网络号：10.100.122.0。将 255.255.255.255 取反后与 IP 地址进行与运算就可以得到主机号。在寻址过程中，先匹配到相同的网络号（标识找到同一个子网），才回去找对应的主机。IP 协议的另一个作用就是路由。路由器寻址工作中，就是要找到目标地址的子网，找到后进而把数据包转发给对应的网络内。**IP 协议的寻址作用是告诉我们去往下一个目的地该朝哪个方向走，路由则是根据「下一个目的地」选择路径。寻址更像在导航，路由更像在操作方向盘。**

4. 网络接口层：在网络层生成了 IP 头部 u 之后哦，需要在 Link Layer(网络接口层) 在 IP 头部加上 MAC 头部并封装成数据帧发送到网络上。以太网在判断网络包目的地时和 IP 的方式不同，因此必须采用相匹配的方式才能在以太网中将包发往目的地，而 MAC 头部就是干这个用的，所以，在以太网进行通讯要用到 MAC 地址。MAC 头部是以太网使用的头部，它包含了接收方和发送方的 MAC 地址等信息，我们可以通过 ARP 协议获取对方的 MAC 地址。所以说，网络接口层主要为网络层提供「链路级别」传输的服务，负责在以太网、WiFi 这样的底层网络上发送原始数据包，工作在网卡这个层次，使用 MAC 地址来标识网络上的设备。综上所述，TCP/IP 网络通常是由上到下分成 4 层，分别是应用层，传输层，网络层和网络接口层。



1.3 思考题：键入网址到网页显示发生的事情？

1. **第一步 HTTP** 浏览器首先解析 URL, 生成要发送的请求信息。解析后就获得 Web 服务器和文件名，然后要生成请求信息。

2. **第二步 DNS 服务器**通过浏览器解析 URL 并生成 HTTP 消息后，需要委托操作系统将消息发送给 Web 服务器。

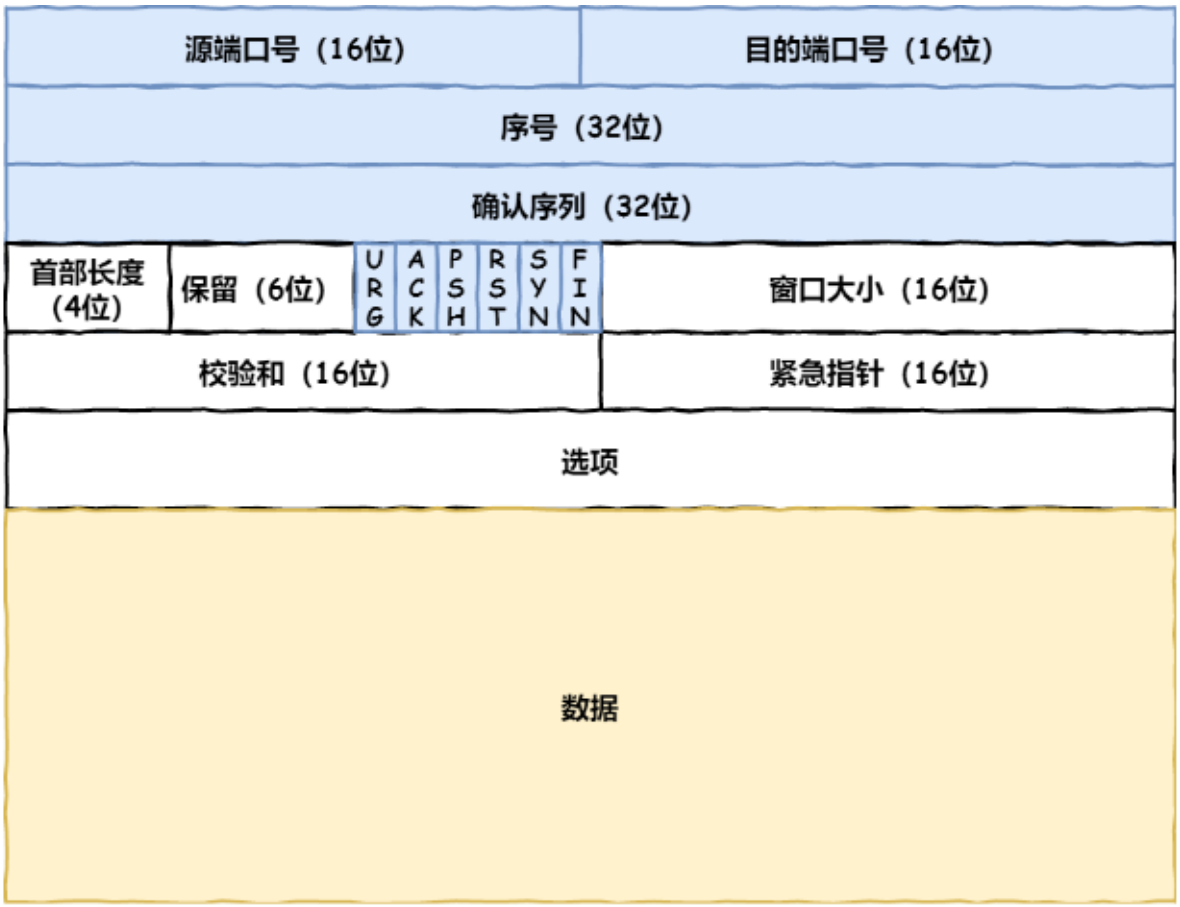
   但在发送之前，还有一项工作需要完成，那就是查询服务器域名对应的 IP 地址，因为委托操作系统发送消息时，必须提供通信对象的 IP 地址。DNS 中的域名都是用句点来分隔的，比如 www.server.com，这里的句点代表了不同层次之间的界限。在域名中，越靠右的位置表示其层级越高。域名解析流程: 客户端首先会发出一个 DNS 请求，问 www.server.com 的 IP 是啥，并发给本地 DNS 服务器（也就是客户端的 TCP/IP 设置中填写的 DNS 服务器地址）。本地域名服务器收到客户端的请求后，如果缓存里的表格能找到 www.server.com，则它直接返回 IP 地址。如果没有，本地 DNS 会去问它的根域名服务器：“老大，能告诉我 www.server.com 的 IP 地址吗？”根域名服务器是最高层次的，它不直接用于域名解析，但能指明一条道路。根 DNS 收到来自本地 DNS 的请求后，发现后置是.com，说：“www.server.com 这个域名归.com 区域管理”，我给你.com 顶级域名服务器地址给你，你去问问它吧。”本地 DNS 收到

顶级域名服务器的地址后，发起请求问“老二，你能告诉我 www.server.com 的 IP 地址吗?”顶级域名服务器说：“我给你负责 www.server.com 区域的权威 DNS 服务器的地址，你去问它应该能问到”。本地 DNS 于是转向问权威 DNS 服务器：“老三，www.server.com 对应的 IP 是啥呀?” server.com 的权威 DNS 服务器，它是域名解析结果的原出处。为啥叫权威呢？就是我的域名我做主。权威 DNS 服务器查询后将对应的 IP 地址 X.X.X.X 告诉本地 DNS。本地 DNS 再将 IP 地址返回客户端，客户端和目标建立连接。

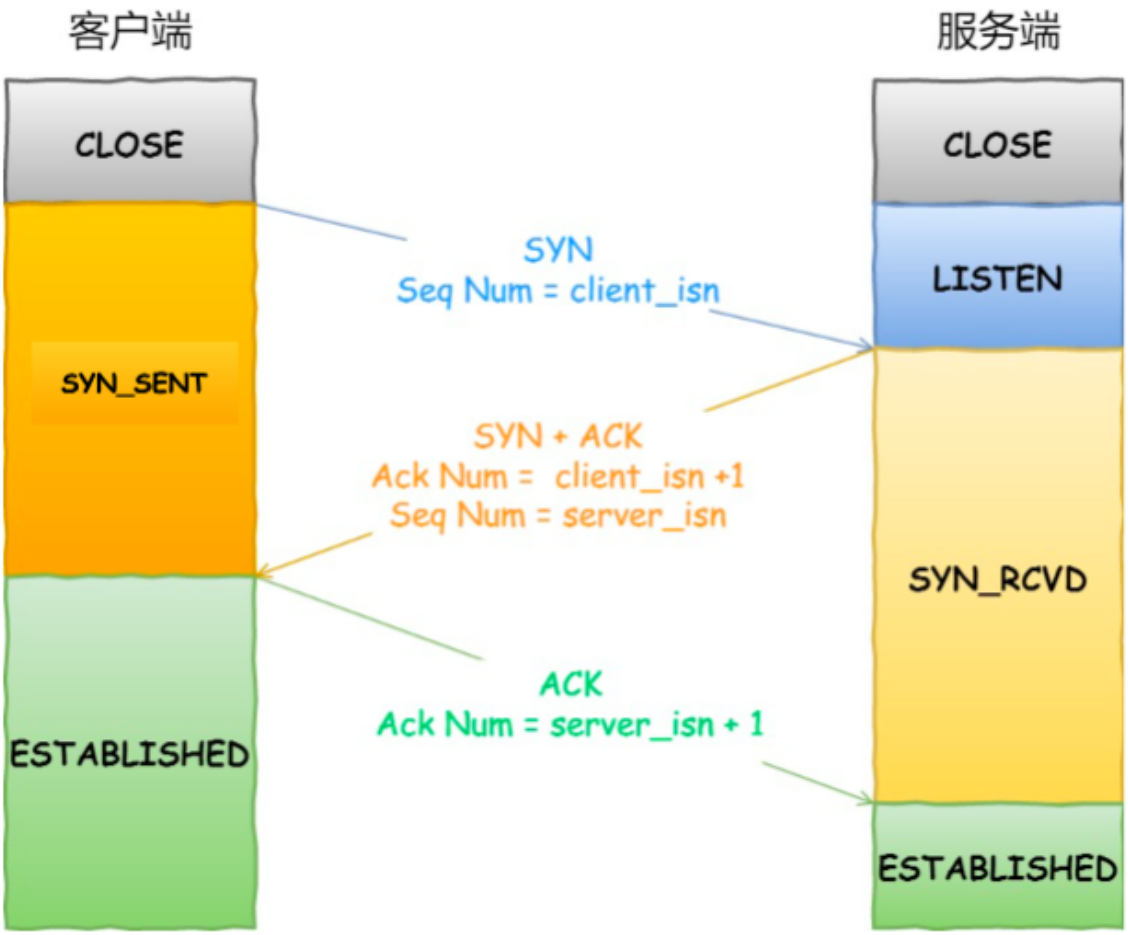
3. **第三步协议栈**通过 DNS 获取到 IP 后，就可以把 HTTP 的传输工作交给操作系统中的协议栈。协议栈的上半部分有两块，分别是负责收发数据的 TCP 和 UDP 协议，这两个传输协议会接受应用层的委托执行收发数据的操作。协议栈的下面一半是用 IP 协议控制网络包收发操作，在互联网上传数据时，数据会被切分成一块块的网络包，而将网络包发送给对方的操作就是由 IP 负责的。协议栈的下面一半是用 IP 协议控制网络包收发操作，在互联网上传数据时，数据会被切分成一块块的网络包，而将网络包发送给对方的操作就是由 IP 负责的。

   - ICMP 用于告知网络包传送过程中产生的错误以及各种控制信息。
   - ARP 用于根据 IP 地址查询相应的以太网 MAC 地址。

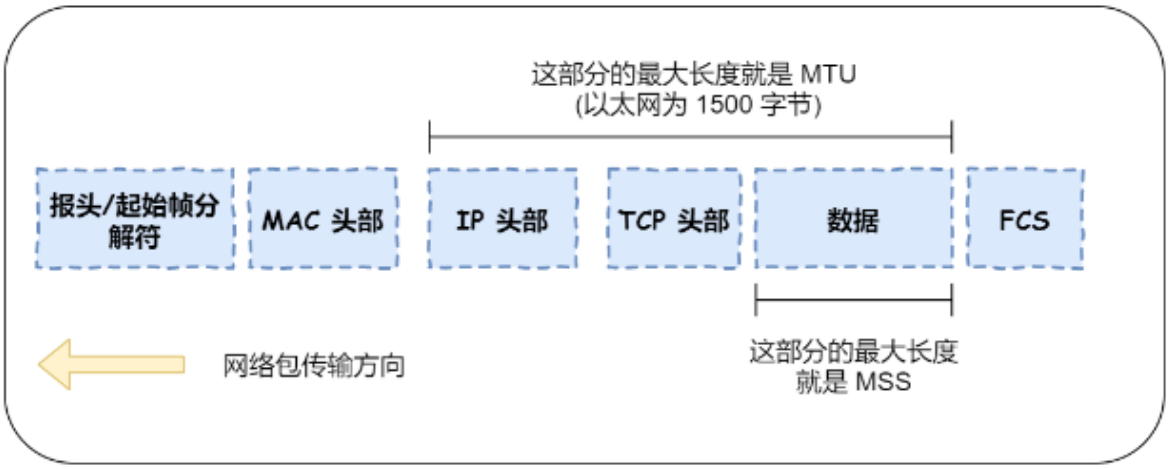
4. **TCP**



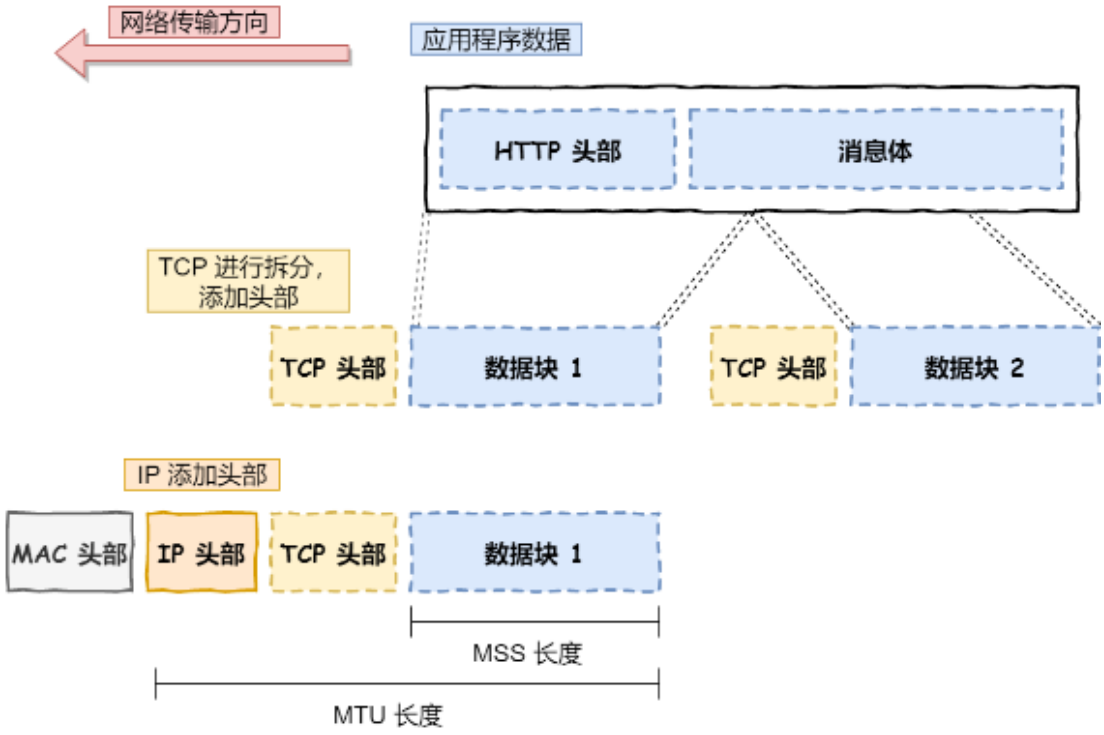
源端口号和目标端口号是不可少的。还有应该有的是确认号，目的是确认发出去对方是否有收到。如果没有收到就应该重新发送，直到送达，这个是为了解决丢包的问题。接下来还有一些状态位。例如 SYN 是发起一个连接，ACK 是回复，RST 是重新连接，FIN 是结束连接等。TCP 是面向连接的，因而双方要维护连接的状态，这些带状态位的包的发送，会引起双方的状态变更。还有一个重要的就是窗口大小。TCP 要做流量控制，通信双方各声明一个窗口（缓存大小），标识自己当前能够的处理能力。除了做流量控制以外，TCP 还会做拥塞控制，对于真正的通路堵车不堵车，它无能为力，唯一能做的就是控制自己，也即控制发送的速度。**三次握手**



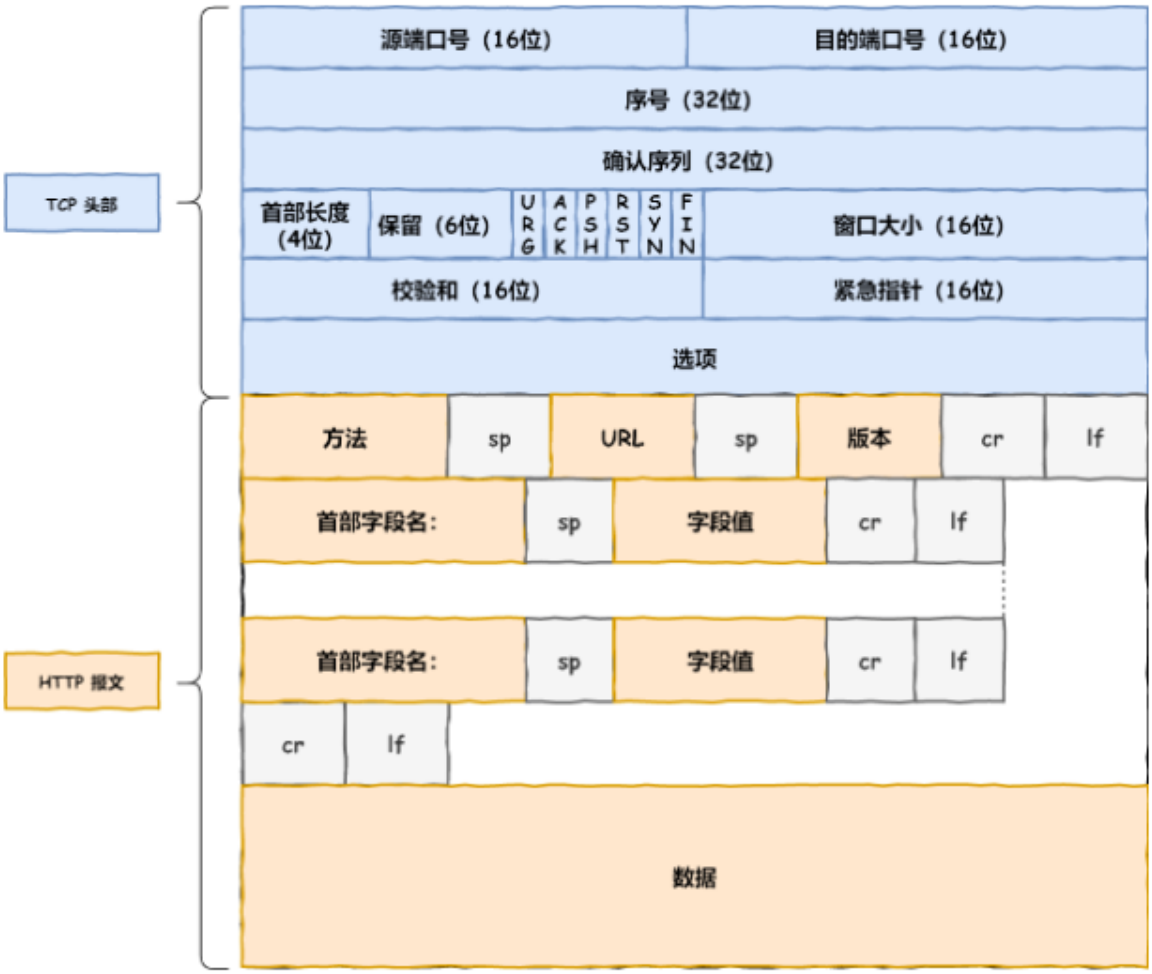
一开始，客户端和服务端都处于 CLOSED 状态。先是服务端主动监听某个端口，处于 LISTEN 状态。然后客户端主动发起连接 SYN，之后处于 SYN-SENT 状态。服务端收到发起的连接，返回 SYN，并且 ACK 客户端的 SYN，之后处于 SYN-RCVD 状态。客户端收到服务端发送的 SYN 和 ACK 之后，发送对 SYN 确认的 ACK，之后处于 ESTABLISHED 状态，因为它一发一收成功了。服务端收到 ACK 的 ACK 之后，处于 ESTABLISHED 状态，因为它也一发一收了。所以三次握手目的是保证双方都有发送和接收的能力。如果 HTTP 请求消息比较长，超过了 MSS 的长度，这时 TCP 就需要把 HTTP 的数据拆解成一块块的数据发送，而不是一次性发送所有数据。



MTU：一个网络包的最大长度，以太网中一般为 1500 字节。MSS：除去 IP 和 TCP 头部之后，一个网络包所能容纳的 TCP 数据的最大长度。数据会被以 MSS 的长度为单位进行拆分，拆分出来的每一块数据都会被放进单独的网络包中。也就是在每个被拆分的数据加上 TCP 头信息，然后交给 IP 模块来发送数据。



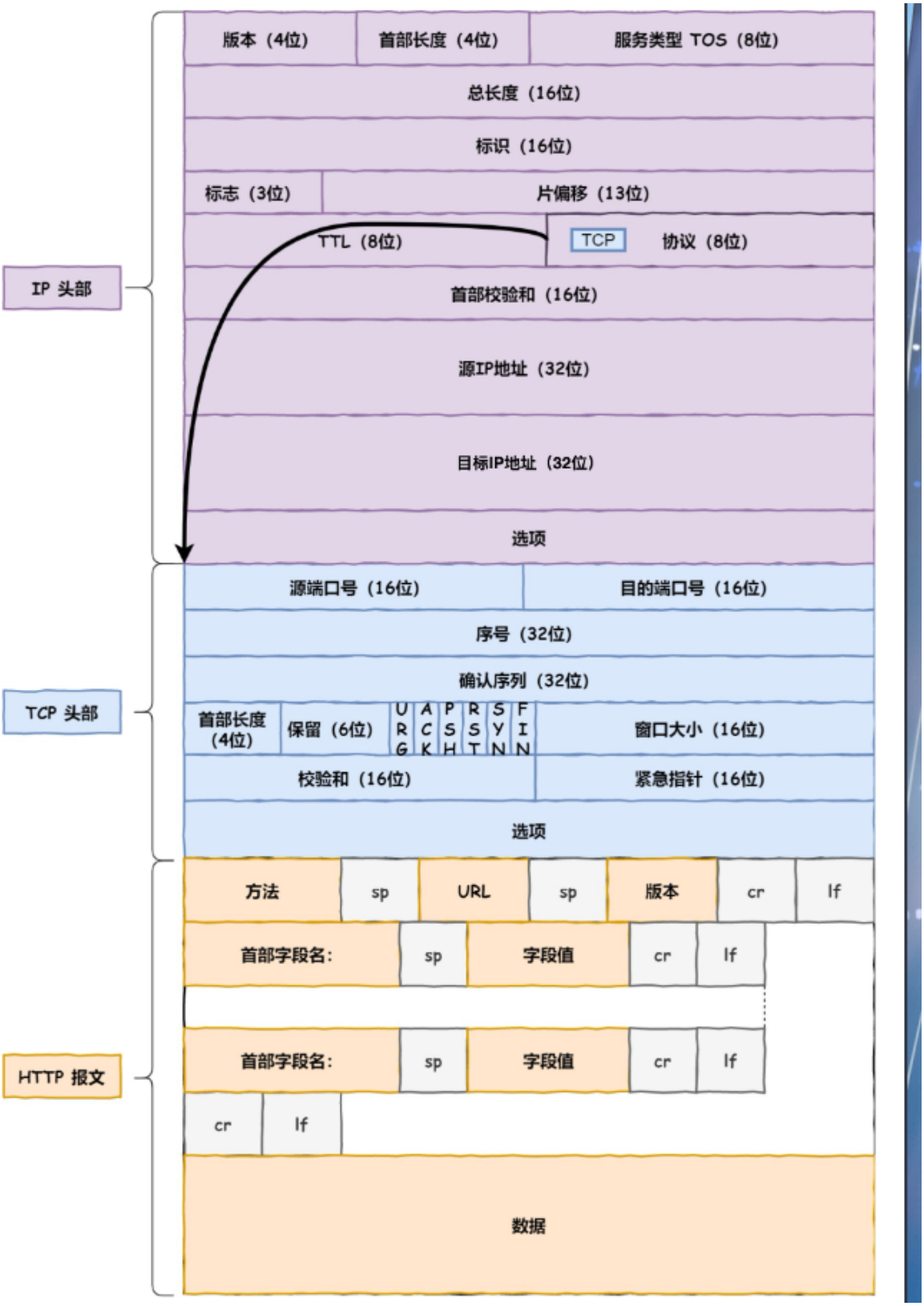
TCP 报文生成



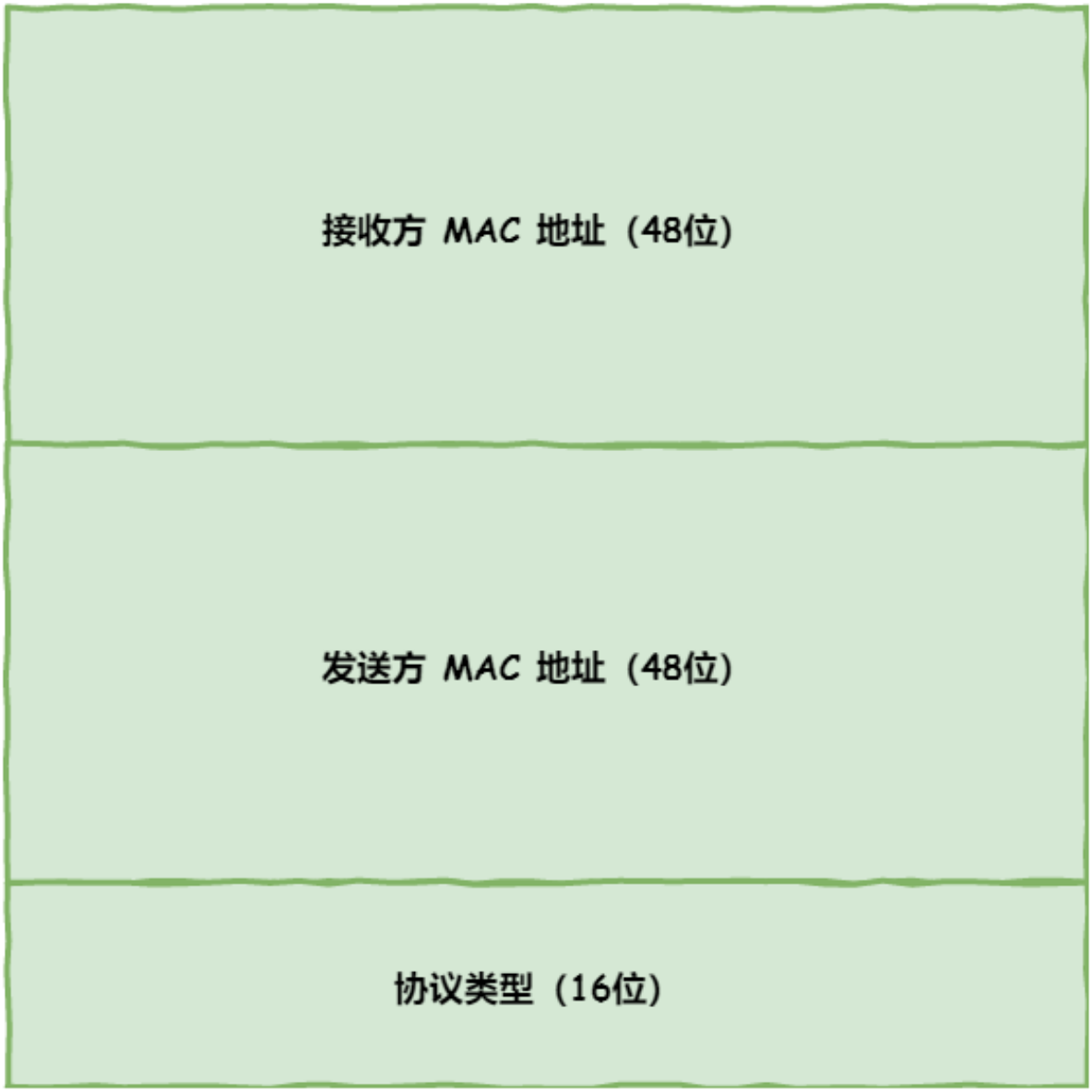
5. IP TCP 模块在执行连接、收发、断开等各阶段操作时，都需要委托 IP 模块将数据封装成网络包发送给通信对象。


版本（4位）
首部长度（4位）
服务类型 TOS（8位）
总长度（16位）
标识（16位）
标志（3位）
片偏移（13位）
TTL（8位）
协议（8位）
首部校验和（16位）
源IP地址（32位）
目标IP地址（32位）
选项
数据

在 IP 协议里面需要有源地址 IP 和目标地址 IP：因为 HTTP 是经过 TCP 传输的，所以在 IP 包头的协议号，要填写为 06（十六进制），表示协议为 TCP。当存在多个网卡时，在填写源地址 IP 时，就需要判断到底应该填写哪个地址。这个判断相当于在多块网卡中判断应该使用哪个一块网卡来发送包。这个时候就需要根据路由表规则，来判断哪一个网卡作为源地址 IP。这里将目标 IP 地址与子网掩码进行与运算，比较结果目标地址和子网掩码都是 0.0.0.0，这表示默认网关，如果其他所有条目都无法匹配，就会自动匹配这一行。并且后续就把包发给路由器，Gateway 即是路由器的 IP 地址。最终形成的数据报：



6. **两点传输-MAC** MAC 头部是以太网使用的头部，它包含了接收方和发送方的 MAC 地址等信息。



在 MAC 包头里需要发送方 MAC 地址和接收方目标 MAC 地址，用于两点之间的传输。一般在 TCP/IP 通信里，MAC 包头的协议类型只使用：

- 0800： IP 协议
- 0806： ARP 协议
- MAC 发送方和接收方如何确认？发送方的 MAC 地址获取就比较简单了，MAC 地址是在网卡生产时写入到 ROM 里的，只要将这个值读取出来写入到 MAC 头部就可以了。接收方的 MAC 地址就有点复杂了，只要告诉以太网对方的 MAC 的地址，以太网就会帮我们把包发送过去，那么很显然这里应该填写对方的 MAC 地址。所以先得搞清楚应该把包发给谁，这个只要查一下路由表就知道了。在路由表中找到相匹配的条目，然后把包发给 Gateway 列中的 IP 地址就可以了。
- 如何获取对方的 MAC 地址？需要 ARP 协议帮我们找到路由器的 MAC 地址。ARP 协议会在以太网中以广播的形式，对以太网所有的设备喊出：“这个 IP 地址是谁的？请把你的 MAC 地址告诉我”。在后续操作系统会把本次查询结果放到一块叫做 ARP 缓存的内存空间留着以后用，不过缓存的时间就几分钟。

至此，就要把数据包发送出去了

7. 网卡网络包只是存放在内存中的一串二进制数字信息，没有办法直接发送给对方。因此，我们需要将数字信息转换为电信号，才能在网线上传输，也就是说，这才是真正的数据发送过程。负责执行这一操作的

是网卡，要控制网卡还需要靠网卡驱动程序。网卡驱动获取网络包之后，会将其复制到网卡内的缓存区中，接着会在其开头加上报头和起始帧分界符，在末尾加上用于检测错误的帧校验序列。

8. 交换机交换机的设计是将网络包原样转发到目的地。交换机工作在 MAC 层，也称为二层网络设备。首先，电信号到达网线接口，交换机里的模块进行接收，接下来交换机里的模块将电信号转换为数字信号。然后通过包末尾的 FCS 校验错误，如果没问题则放到缓冲区。这部分操作基本和计算机的网卡相同，但交换机的工作方式和网卡不同。计算机的网卡本身具有 MAC 地址，并通过核对收到的包的接收方 MAC 地址判断是不是发给自己的，如果不是发给自己的则丢弃；相对地，交换机的端口不核对接收方 MAC 地址，而是直接接收所有的包并存放到缓冲区中。因此，和网卡不同，交换机的端口不具有 MAC 地址。交换机的 MAC 地址表主要包含两个信息：一个是设备的 MAC 地址，另一个是该设备连接在交换机的哪个端口上。交换机根据 MAC 地址表查找 MAC 地址，然后将信号发送到相应的端口。当 MAC 地址表中找不到指定的 MAC 地址时，交换机就无法判断应该把包发送到那里，就会把包转发到除了源端口之外的所有端口上。这样做不会产生什么问题，因为以太网的设计本来就是将包发送到整个网络的，然后只有相应的接收者才接收包，而其他设备则会忽略这个包。

9. **路由器**网络包经过交换机之后，现在到达了路由器，并在此被转发到下一个路由器或目标设备。

   · 与交换机的区别因为路由器是基于 IP 设计的，俗称三层网络设备，路由器的各个端口都具有 MAC 地址和 IP 地址；而交换机是基于以太网设计的，俗称二层网络设备，交换机的端口不具有 MAC 地址。

   · 基本原理路由器的端口具有 MAC 地址，因此它就能够成为以太网的发送方和接收方；同时还具有 IP 地址，从这个意义上来说，它和计算机的网卡是一样的。当转发包时，首先路由器端口会接收发给自己的以太网包，然后路由表查询转发目标，再由相应的端口作为发送方将以太网包发送出去。

   · 包接受操作首先，电信号到达网线接口部分，路由器中的模块会将电信号转成数字信号，然后通过包末尾的 FCS 进行错误校验。如果没问题则检查 MAC 头部中的接收方 MAC 地址，看看是不是发给自己的包，如果是就放到接收缓冲区中，否则就丢弃这个包。总的来说，路由器的端口都具有 MAC 地址，只接收与自身地址匹配的包，遇到不匹配的包则直接丢弃。

   · 查询路由表确定输出端口完成包接收操作之后，路由器就会去掉包开头的 MAC 头部。MAC 头部的作用就是将包送达路由器，其中的接收方 MAC 地址就是路由器端口的 MAC 地址。因此，当包到达路由器之后，MAC 头部的任务就完成了，于是 MAC 头部就会被丢弃。接下来，路由器会根据 MAC 头部后方的 IP 头部中的内容进行包的转发操作。转发操作分为几个阶段，首先是查询路由表判断转发目标。这里的路由匹配也使用的目标地址与条目的子网掩码进行按位与运算。

   · 包的发送操作首先，我们需要根据路由表的网关列判断对方的地址。如果网关是一个 IP 地址，则这个 IP 地址就是我们要转发到的目标地址，还未抵达终点，还需继续需要路由器转发。如果网关为空，则 IP 头部中的接收方 IP 地址就是要转发到的目标地址，也是就终于找到 IP 包头里的目标地址了，说明已抵达终点。

   知道对方的 IP 地址之后，接下来需要通过 ARP 协议根据 IP 地址查询 MAC 地址，并将查询的结果作为接收方 MAC 地址。路由器也有 ARP 缓存，因此首先会在 ARP 缓存中查询，如果找不到则发送 ARP 查询请求。接下来是发送方 MAC 地址字段，这里填写输出端口的 MAC 地址。还有一个以太类型字段，填写 0800（十六进制）表示 IP 协议。网络包完成后，接下来会将其转换成电信号并通过端口发送出去。这一步的工作过程和计算机也是相同的。发送出去的网络包会通过交换机到达下一个路由器。由于接收方 MAC 地址就是下一个路由器的地址，所以交换机会根据这一地址将包传输到下一个路由器。接下来，下一个路由器会将包转发给再下一个路由器，经过层层转发之后，网络包就到达了最终的目的地。在网络包传输的过程中，源 IP 和目标 IP 始终是不会变的，一直变化的是 MAC 地址，因为需要 MAC 地址在以太网内进行两个设备之间的包传输。

10. **服务器与客户端**数据包抵达服务器后，服务器会先扒开数据包的 MAC 头部，查看是否和服务器自己的 MAC 地址符合，符合就将包收起来。接着继续扒开数据包的 IP 头，发现 IP 地址符合，根据 IP 头中协议项，知道自己上层是 TCP 协议。于是，扒开 TCP 的头，里面有序列号，需要看一看这个序列包是不是我想要的，如果是就放入 **缓存**中然后返回一个 ACK，如果不是就丢弃。TCP 头部里面还有端口号，HTTP 的服务器正在监听这个端口号。

2 Linux 接受网络包的流程

网卡是计算机里的一个硬件，专门负责接收和发送网络包，当网卡接收到一个网络包后，会通过 DMA 技术，将网络包写入到指定的内存地址，也就是写入到 Ring Buffer ，这个是一个环形缓冲区，接着就会告诉操作系统这个网络包已经到达。

- 如何通知操作系统网络包已经达到？最简单的一种方式就是触发中断，也就是每当网卡收到一个网络包，就触发一个中断告诉操作系统。为了解决频繁中断带来的性能开销，Linux 内核在 2.6 版本中引入了 NAPI 机制，它是混合「中断和轮询」的方式来接收网络包，它的核心概念就是不采用中断的方式读取数据，而是首先采用中断唤醒数据接收的服务程序，然后 poll 的方法来轮询数据。因此，当有网络包到达时，会通过 DMA 技术，将网络包写入到指定的内存地址，接着网卡向 CPU 发起硬件中断，当 CPU 收到硬件中断请求后，根据中断表，调用已经注册的中断处理函数。硬件终端处理函数：
  - 需要先「暂时屏蔽中断」，表示已经知道内存中有数据了，告诉网卡下次再收到数据包直接写内存就可以了，不要再通知 CPU 了，这样可以提高效率，避免 CPU 不停的被中断。
  - 接着，发起「软中断」，然后恢复刚才屏蔽的中断。

  软中断的处理：内核中的 ksoftirqd 线程专门负责软中断的处理，当 ksoftirqd 内核线程收到软中断后，就会来轮询处理数据。ksoftirqd 线程会从 Ring Buffer 中获取一个数据帧，用 sk_{buff} 表示，从而可以作为一个网络包交给网络协议栈进行逐层处理。

3 HTTP 协议

- 状态码 1xx 类状态码属于提示信息，是协议处理中的一种中间状态，实际用到的比较少。2xx 类状态码表示服务器成功处理了客户端的请求，也是我们最愿意看到的状态。「200 OK」是最常见的成功状态码，表示一切正常。如果是非 HEAD 请求，服务器返回的响应头都会有 body 数据。「204 No Content」也是常见的成功状态码，与 200 OK 基本相同，但响应头没有 body 数据。「206 Partial Content」是应用于 HTTP 分块下载或断点续传，表示响应返回的 body 数据并不是资源的全部，而是其中的一部分，也是服务器处理成功的状态。3xx 类状态码表示客户端请求的资源发生了变动，需要客户端用新的 URL 重新发送请求获取资源，也就是重定向。「301 Moved Permanently」表示永久重定向，说明请求的资源已经不存在了，需改用新的 URL 再次访问。「302 Found」表示临时重定向，说明请求的资源还在，但暂时需要用另一个 URL 来访问。4xx 类状态码表示客户端发送的报文有误，服务器无法处理，也就是错误码的含义。「400 Bad Request」表示客户端请求的报文有错误，但只是个笼统的错误。「403 Forbidden」表示服务器禁止访问资源，并不是客户端的请求出错。「404 Not Found」表示请求的资源在服务器上不存在或未找到，所以无法提供给客户端。5xx 类状态码表示客户端请求报文正确，但是服务器处理时内部发生了错误，属于服务器端的错误码。
- Connection 字段最常用的就是 Keep-Alive 字段，用来表示长连接。
- Content-Type 字段用于服务器回应时，告知客户端本次数据是什么格式。

- Accept 声明自己可以接受哪些数据格式。

- Get 和 Post 的区别 GET 的语义是从服务器获取指定的资源，这个资源可以是静态的文本、页面、图片视频等。GET 请求的参数位置一般是写在 URL 中，URL 规定只能支持 ASCII，所以 GET 请求的参数只允许 ASCII 字符，而且浏览器会对 URL 的长度有限制（HTTP 协议本身对 URL 长度并没有做任何规定）。POST 的语义是根据请求负荷（报文 body）对指定的资源做出处理，具体的处理方式视资源类型而不同。POST 请求携带数据的位置一般是写在报文 body 中，body 中的数据可以是任意格式的数据，只要客户端与服务端协商好即可，而且浏览器不会对 body 大小做限制。

- Get 和 Post 方法都是安全和幂等的吗？在 HTTP 协议里，所谓的「安全」是指请求方法不会「破坏」服务器上的资源。所谓的「幂等」，意思是多次执行相同的操作，结果都是「相同」的 GET 方法就是安全且幂等的，因为它是「只读」操作，无论操作多少次，服务器上的数据都是安全的，且每次的结果都是相同的。所以，可以对 GET 请求的数据做缓存，这个缓存可以做到浏览器本身上（彻底避免浏览器发请求），也可以做到代理上（如 nginx），而且在浏览器中 GET 请求可以保存为书签。POST 因为是「新增或提交数据」的操作，会修改服务器上的资源，所以是不安全的，且多次提交数据就会创建多个资源，所以不是幂等的。所以，浏览器一般不会缓存 POST 请求，也不能把 POST 请求保存为书签。

  GET 的语义是请求获取指定的资源。GET 方法是安全、幂等、可被缓存的。

3.1 HTTP 缓存技术

实现方式：强制缓存和协商缓存

- 强制缓存强缓存指的是只要浏览器判断缓存没有过期，则直接使用浏览器的本地缓存，决定是否使用缓存的主动性在于浏览器这边。强缓存是利用下面这两个 HTTP 响应头部（Response Header）字段实现的，它们都用来表示资源在客户端缓存的有效期：Cache-Control，是一个相对时间；Expires，是一个绝对时间；如果 HTTP 响应头部同时有 Cache-Control 和 Expires 字段的话，Cache-Control 的优先级高于 Expires 。

- 协商缓存就是与服务端协商之后，通过协商结果来判断是否使用本地缓存。两种实现：

  1. 请求头部中的 If-Modified-Since 字段与响应头部中的 Last-Modified 字段实现请求头部中的 If-Modified-Since：当资源过期了，发现响应头中具有 Last-Modified 声明，则再次发起请求的时候带上 Last-Modified 的时间，服务器收到请求后发现有 If-Modified-Since 则与被请求资源的最后修改时间进行对比（Last-Modified），如果最后修改时间较新（大），说明资源又被改过，则返回最新资源，HTTP 200 OK；如果最后修改时间较旧（小），说明资源无新修改，响应 HTTP 304 走缓存。
  2. 请求头部中的 If-None-Match 字段与响应头部中的 ETag 字段响应头部中 Etag：唯一标识响应资源；请求头部中的 If-None-Match：当资源过期时，浏览器发现响应头里有 Etag，则再次向服务器发起请求时，会将请求头 If-None-Match 值设置为 Etag 的值。服务器收到请求后进行比对，如果资源没有变化返回 304，如果资源变化了返回 200。

  第一种实现方式是基于时间实现的，第二种实现方式是基于一个唯一标识实现的，相对来说后者可以更加准确地判断文件内容是否被修改，避免由于时间篡改导致的不可靠问题。如果在第一次请求资源的时候，服务端返回的 HTTP 响应头部同时有 Etag 和 Last-Modified 字段，那么客户端再下一次请求的时候，如果带上了 ETag 和 Last-Modified 字段信息给服务端，这时 Etag 的优先级更高，也就是服务端先会判断 Etag 是否变化了，如果 Etag 有变化就不用在判断 Last-Modified 了，如果 Etag 没有变化，然后再看 Last-Modified。

- 为什么 ETag 优先级更高?
  - 在没有修改文件内容情况下文件的最后修改时间可能也会改变，这会导致客户端认为这文件被改动了，从而重新请求；
  - 可能有些文件是在秒级以内修改的，If-Modified-Since 能检查到的粒度是秒级的，使用 Etag 就能够保证这种需求下客户端在 1 秒内能刷新多次；
  - 有些服务器不能精确获取文件的最后修改时间。

协商缓存这两个字段都需要配合强制缓存中 Cache-Control 字段来使用，只有在未能命中强制缓存的时候，才能发起带有协商缓存字段的请求。

- 使用 ETag 实现的协商缓存的过程：当浏览器第一次请求访问服务器资源时，服务器会在返回这个资源的同时，在 Response 头部加上 ETag 唯一标识，这个唯一标识的值是根据当前请求的资源生成的；当浏览器再次请求访问服务器中的该资源时，首先会先检查强制缓存是否过期：如果没有过期，则直接使用本地缓存；如果缓存过期了，会在 Request 头部加上 If-None-Match 字段，该字段的值就是 ETag 唯一标识；服务器再次收到请求后，会根据请求中的 If-None-Match 值与当前请求的资源生成的唯一标识进行比较：如果值相等，则返回 304 Not Modified，不会返回资源；如果不相等，则返回 200 状态码和返回资源，并在 Response 头部加上新的 ETag 唯一标识；如果浏览器收到 304 的请求响应状态码，则会从本地缓存中加载资源，否则更新资源。

3.2 HTTP 特性

- 简单 HTTP 基本的报文格式就是 header + body，头部信息也是 key-value 简单文本的形式，易于理解，降低了学习和使用的门槛。
- 灵活、易于拓展 HTTP 协议里的各类请求方法、URI/URL、状态码、头字段等每个组成要求都没有被固定死，都允许开发人员自定义和扩充。同时 HTTP 由于是工作在应用层（OSI 第七层），则它下层可以随意变化，比如说 HTTP3.0 改用 UDP 作为下层传输协议。
- 应用广泛和跨平台天然具有跨平台的优越性。

3.3 HTTP/1.1 的缺点

- 无状态无状态的好处，因为服务器不会去记忆 HTTP 的状态，所以不需要额外的资源来记录状态信息，这能减轻服务器的负担，能够把更多的 CPU 和内存用来对外提供服务。无状态的坏处，既然服务器没有记忆能力，它在完成有关联性的操作时会非常麻烦。
  - 解决方案：Cookie 技术：在客户端第一次请求后，服务器会下发一个装有客户信息的「小贴纸」，后续客户端请求服务器的时候，带上「小贴纸」，服务器就能认得了了。
- 明文传输明文意味着在传输过程中的信息，是可方便阅读的，比如 Wireshark 抓包都可以直接肉眼查看，为我们调试工作带了极大的便利性。但是这正是这样，HTTP 的所有信息都暴露在了光天化日下，相当于信息裸奔。
- 不安全通信使用明文（不加密），内容可能会被窃听。比如，账号信息容易泄漏，那你号没了。不验证通信方的身份，因此有可能遭遇伪装。比如，访问假的淘宝、拼多多，那你钱没了。无法证明报文的完整性，所以有可能已遭篡改。比如，网页上植入垃圾广告，视觉污染，眼没了。
  - 解决方案：引入 SSL/TLS 层。

3.4 HTTP/1.1 的性能？

HTTP 协议是基于 TCP/IP，并且使用了「请求 - 应答」的通信模式，所以性能的关键就在这两点里。

- 长连接早期 HTTP/1.0 性能上的一个很大的问题，那就是每发起一个请求，都要新建一次 TCP 连接（三次握手），而且是串行请求，做了无谓的 TCP 连接建立和断开，增加了通信开销。为了解决上述 TCP 连接问题，HTTP/1.1 提出了长连接的通信方式，也叫持久连接。这种方式的好处在于减少了 TCP 连接的重复建立和断开所造成的额外开销，减轻了服务器端的负载。特点：只要任意一段没有明确提出断开连接就保持 TCP 连接状态。

- 管道网络传输 HTTP/1.1 采用了长连接的方式，这使得管道（pipeline）网络传输成为了可能。即可在同一个 TCP 连接里面，客户端可以发起多个请求，只要第一个请求发出去了，不必等其回来，就可以发第二个请求出去，可以减少整体的响应时间。举例来说，客户端需要请求两个资源。以前的做法是，在同一个 TCP 连接里面，先发送 A 请求，然后等待服务器做出回应，收到后再发出 B 请求。那么，管道机制则是允许浏览器同时发出 A 请求和 B 请求。**但是服务器必须按照接收请求的顺序发送对这些管道化请求的响应**。如果服务端在处理 A 请求时耗时比较长，那么后续的请求的处理都会被阻塞住，这称为「队头堵塞」。**所以，HTTP/1.1 管道解决了请求的队头阻塞，但是没有解决响应的队头阻塞**。

- 队头阻塞「请求 - 应答」的模式会造成 HTTP 的性能问题。为什么呢？因为当顺序发送的请求序列中的一个请求因为某种原因被阻塞时，在后面排队的所有请求也一同被阻塞了，会招致客户端一直请求不到数据，这也就是「队头阻塞」。

3.5 HTTP 与 HTTPS

- 区别
  - HTTP 是超文本传输协议，信息是明文传输，存在安全风险的问题。HTTPS 则解决 HTTP 不安全的缺陷，在 TCP 和 HTTP 网络层之间加入了 SSL/TLS 安全协议，使得报文能够加密传输。
  - HTTP 连接建立相对简单，TCP 三次握手之后便可进行 HTTP 的报文传输。而 HTTPS 在 TCP 三次握手之后，还需进行 SSL/TLS 的握手过程，才可进入加密报文传输。
  - 两者的默认端口不一样，HTTP 默认端口号是 80，HTTPS 默认端口号是 443。
  - HTTPS 协议需要向 CA（证书权威机构）申请数字证书，来保证服务器的身份是可信的。

- HTTPS 解决了哪些问题?
  - 窃听风险，比如通信链路上可以获取通信内容，用户号容易没。
  - 篡改风险，比如强制植入垃圾广告，视觉污染，用户眼容易瞎。
  - 冒充风险，比如冒充淘宝网站，用户钱容易没。

- TLS/SSL 的特点
  - 信息加密：交互信息无法被窃取。
  - 校验机制：无法篡改通信内容，篡改了就不能正常显示。
  - 身份证书：证明淘宝是真的淘宝网，但你的钱还是会因为「剁手」而没。

- 如何解决三个风险?
  - 混合加密的方式实现信息的机密性，解决了窃听的风险。

  - 摘要算法的方式来实现完整性，它能够为数据生成独一无二的「指纹」，指纹用于校验数据的完整性，解决了篡改的风险。
  - 将服务器公钥放入到数字证书中，解决了冒充的风险。

- 混合加密
  - 在通信建立前采用非对称加密的方式交换「会话秘钥」，后续就不再使用非对称加密。
  - 在通信过程中全部使用对称加密的「会话秘钥」的方式加密明文数据。

- 为何采取混合加密
  - 对称加密只使用一个密钥，运算速度快，密钥必须保密，无法做到安全的密钥交换。
  - 非对称加密使用两个密钥：公钥和私钥，公钥可以任意分发而私钥保密，解决了密钥交换问题但速度慢。

- HTTPS 如何建立连接？SSL/TLS 协议基本流程：
  - 客户端向服务器索要并验证服务器的公钥。
  - 双方协商生产「会话秘钥」。
  - 双方采用「会话秘钥」进行加密通信。

- HTTPS 的应用数据如何保证完整性?
  - TLS 在实现上分为握手协议和记录协议两层：TLS 握手协议就是我们前面说的 TLS 四次握手的过程，负责协商加密算法和生成对称密钥，后续用此密钥来保护应用程序数据（即 HTTP 数据）；TLS 记录协议负责保护应用程序数据并验证其完整性和来源，所以对 HTTP 数据加密是使用记录协议；

HTTPS 协议本身到目前为止还是没有任何漏洞的，即使你成功进行中间人攻击，本质上是利用了客户端的漏洞（用户不够安全。

3.6 HTTP/1.1, HTTP/2, HTTP/3 的演进

3.6.1 HTTP/1.1 相比 HTTP/1.0 提高了什么性能？

- 使用长连接的方式改善了 HTTP/1.0 短连接造成的性能开销。
- 支持管道（pipeline）网络传输，只要第一个请求发出去了，不必等其回来，就可以发第二个请求出去，可以减少整体的响应时间。

但是仍然有性能瓶颈：

- 请求 / 响应头部（Header）未经压缩就发送，首部信息越多延迟越大。只能压缩 Body 的部分；
- 发送冗长的首部。每次互相发送相同的首部造成的浪费较多；
- 服务器是按请求的顺序响应的，如果服务器响应慢，会招致客户端一直请求不到数据，也就是队头阻塞；
- 没有请求优先级控制；
- 请求只能从客户端开始，服务器只能被动响应。

3.6.2 HTTP/2 做了什么优化？

- 头部压缩 HTTP/2 会压缩头（Header）如果你同时发出多个请求，他们的头是一样的或是相似的，那么，协议会帮你消除重复的部分。这就是所谓的 HPACK 算法：在客户端和服务器同时维护一张头信息表，所有字段都会存入这个表，生成一个索引号，以后就不发送同样字段了，只发送索引号，这样就提高速度了。
- 二进制格式 HTTP/2 不再像 HTTP/1.1 里的纯文本形式的报文，而是全面采用了二进制格式，头信息和数据体都是二进制，并且统称为帧（frame）：头信息帧（Headers Frame）和数据帧（Data Frame）。
- 并发传输我们都知道 HTTP/1.1 的实现是基于请求-响应模型的。同一个连接中，HTTP 完成一个事务（请求与响应），才能处理下一个事务，也就是说在发出请求等待响应的过程中，是没办法做其他事情的，如果响应迟迟不来，那么后续的请求是无法发送的，也造成了队头阻塞的问题。而 HTTP/2 就很牛逼了，引出了 Stream 概念，多个 Stream 复用在一条 TCP 连接。针对不同的 HTTP 请求用独一无二的 Stream ID 来区分，接收端可以通过 Stream ID 有序组装成 HTTP 消息，不同 Stream 的帧是可以乱序发送的，因此可以并发不同的 Stream ，也就是 HTTP/2 可以并行交错地发送请求和响应。
- 服务器推送 HTTP/2 还在一定程度上改善了传统的「请求 - 应答」工作模式，服务端不再是被动地响应，可以主动向客户端发送消息。客户端和服务器双方都可以建立 Stream，Stream ID 也是有区别的，客户端建立的 Stream 必须是奇数号，而服务器建立的 Stream 必须是偶数号。

3.6.3 HTTP/2 缺陷？

HTTP/2 通过 Stream 的并发能力，解决了 HTTP/1 队头阻塞的问题，看似很完美了，但是 HTTP/2 还是存在“队头阻塞”的问题，只不过问题不是在 HTTP 这一层面，而是在 TCP 这一层。

HTTP/2 是基于 TCP 协议来传输数据的，TCP 是字节流协议，TCP 层必须保证收到的字节数据是完整且连续的， 这样内核才会将缓冲区里的数据返回给 HTTP 应用，那么当「前 1 个字节数据」没有到达时，后收到的字节数据只能存放在内核缓冲区里，只有等到这 1 个字节数据到达时，HTTP/2 应用层才能从内核中拿到数据，这就是 HTTP/2 队头阻塞问题。

所以，一旦发生了丢包现象，就会触发 TCP 的重传机制，这样在一个 TCP 连接中的所有的 HTTP 请求都必须等待这个丢了的包被重传回来。

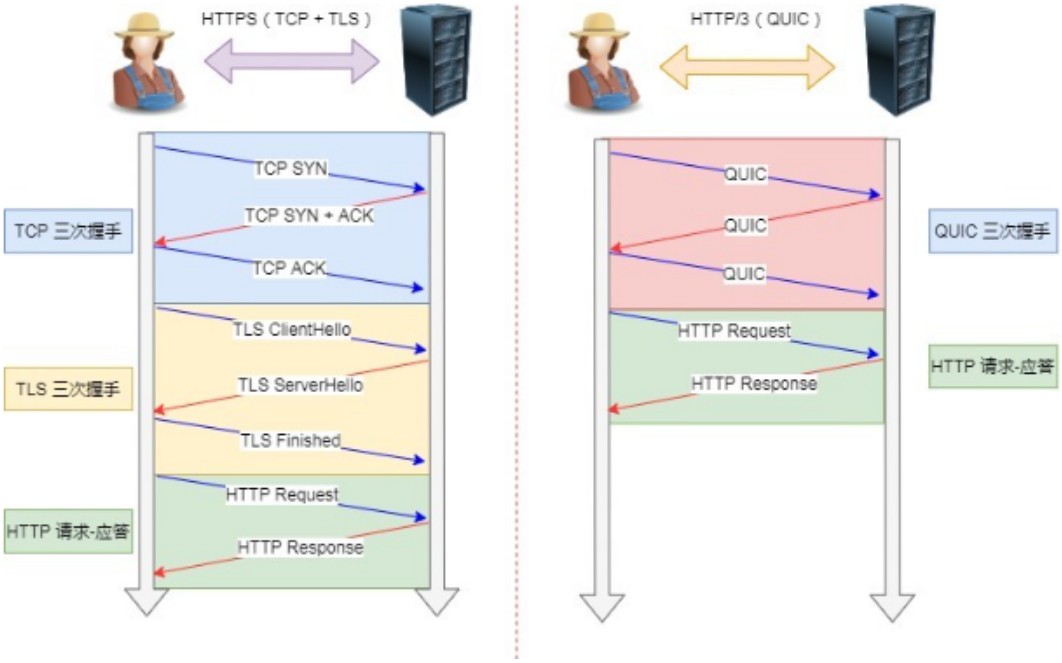
3.6.4 HTTP/3 的优化？

- 队头阻塞 HTTP/1.1 中的管道（pipeline）虽然解决了请求的队头阻塞，但是没有解决响应的队头阻塞，因为服务端需要按顺序响应收到的请求，如果服务端处理某个请求消耗的时间比较长，那么只能等响应完这个请求后，才能处理下一个请求，这属于 HTTP 层队头阻塞。HTTP/2 虽然通过多个请求复用一个 TCP 连接解决了 HTTP 的队头阻塞，但是一旦发生丢包，就会阻塞住所有的 HTTP 请求，这属于 TCP 层队头阻塞。

HTTP/2 队头阻塞的问题是因为 TCP，所以 HTTP/3 把 HTTP 下层的 TCP 协议改成了 UDP！UDP 发送是不管顺序，也不管丢包的，所以不会出现像 HTTP/2 队头阻塞的问题。大家都知道 UDP 是不可靠传输的，但基于 UDP 的 QUIC 协议可以实现类似 TCP 的可靠性传输。

- QUIC 的 3 个特点
  - 无队头阻塞 QUIC 协议也有类似 HTTP/2 Stream 与多路复用的概念，也是可以在同一条连接上并发传输多个 Stream，Stream 可以认为就是一条 HTTP 请求。当某个流发生丢包时，只会阻塞这个流，其他流不会受到影响，因此不存在队头阻塞问题。

- 更快地建立连接对于 HTTP/1 和 HTTP/2 协议，TCP 和 TLS 是分层的，分别属于内核实现的传输层、openssl 库实现的表示层，因此它们难以合并在一起，需要分批次来握手，先 TCP 握手，再 TLS 握手。HTTP/3 在传输数据前虽然需要 QUIC 协议握手，但是这个握手过程只需要 1 RTT，握手的目的是为确认双方的「连接 ID」，连接迁移就是基于连接 ID 实现的。



· 连接迁移基于 TCP 传输协议的 HTTP 协议，由于是通过四元组（源 IP、源端口、目的 IP、目的端口）确定一条 TCP 连接。那么当移动设备的网络从 4G 切换到 WIFI 时，意味着 IP 地址变化了，那么就必须要断开连接，然后重新建立连接。而建立连接的过程包含 TCP 三次握手和 TLS 四次握手的时延，以及 TCP 慢启动的减速过程，给用户的感觉就是网络突然卡顿了一下，因此连接的迁移成本是很高的。而 QUIC 协议没有用四元组的方式来“绑定”连接，而是通过连接 ID 来标记通信的两个端点，客户端和服务器可以各自选择一组 ID 来标记自己，因此即使移动设备的网络变化后，导致 IP 地址变化了，只要仍保有上下文信息（比如连接 ID、TLS 密钥等），就可以“无缝”地复用原连接，消除重连的成本，没有丝毫卡顿感，达到了连接迁移的功能。所以，QUIC 是一个在 UDP 之上的伪 TCP + TLS + HTTP/2 的多路复用的协议。

3.6.5 如何减少 HTTP 请求次数？

· 减少重定向请求次数重定向的工作交由代理服务器完成，就能减少 HTTP 请求次数了。而且当代理服务器知晓了重定向规则后，可以进一步减少消息传递次数。

· 合并请求如果把多个访问小文件的请求合并成一个大的请求，虽然传输的总资源还是一样，但是减少请求，也就意味着减少了重复发送的 HTTP 头部。如果把多个访问小文件的请求合并成一个大的请求，虽然传输的总资源还是一样，但是减少请求，也就意味着减少了重复发送的 HTTP 头部。另外由于 HTTP/1.1 是请求响应模型，如果第一个发送的请求，未收到对应的响应，那么后续的请求就不会发送（PS：HTTP/1.1 管道模式是默认不使用的，所以讨论 HTTP/1.1 的队头阻塞问题，是不考虑管道模式的），于是为了防止单个请求的阻塞，所以一般浏览器会同时发起 5-6 个请求，每一个请求都是不同的 TCP 连接，那么如果合并了请求，也就会减少 TCP 连接的数量，因而省去了 TCP 握手和慢启动过程耗费的时间。通过将多个小图片合并成一个大图片来减少 HTTP 请求的次数，从而减少网络的开销。合并请求的方式就是合并资源，以一个大资源的请求替换多个小资源的请求。但是这样的合并请求会带来新的问题，当大资源中的某一个小资源发生变化后，客户端必须重新下载整个完整的大资源文件，这显然带来了额外的网络消耗。

- 延迟发送请求请求网页的时候，没必要把全部资源都获取到，而是只获取当前用户所看到的页面资源，当用户向下滑动页面的时候，再向服务器获取接下来的资源，这样就达到了延迟发送请求的效果。

3.7 HTTP/2 的优化

- 头部压缩 1.1 版本中 Header 部分存在有：
  - 含很多固定的字段，比如 Cookie、User Agent、Accept 等，这些字段加起来也高达几百字节甚至上千字节，所以有必要压缩；
  - 大量的请求和响应的报文里有很多字段值都是重复的，这样会使得大量带宽被这些冗余的数据占用了，所以有必须要避免重复性；
  - 字段是 ASCII 编码的，虽然易于人类观察，但效率低，所以有必要改成二进制编码；

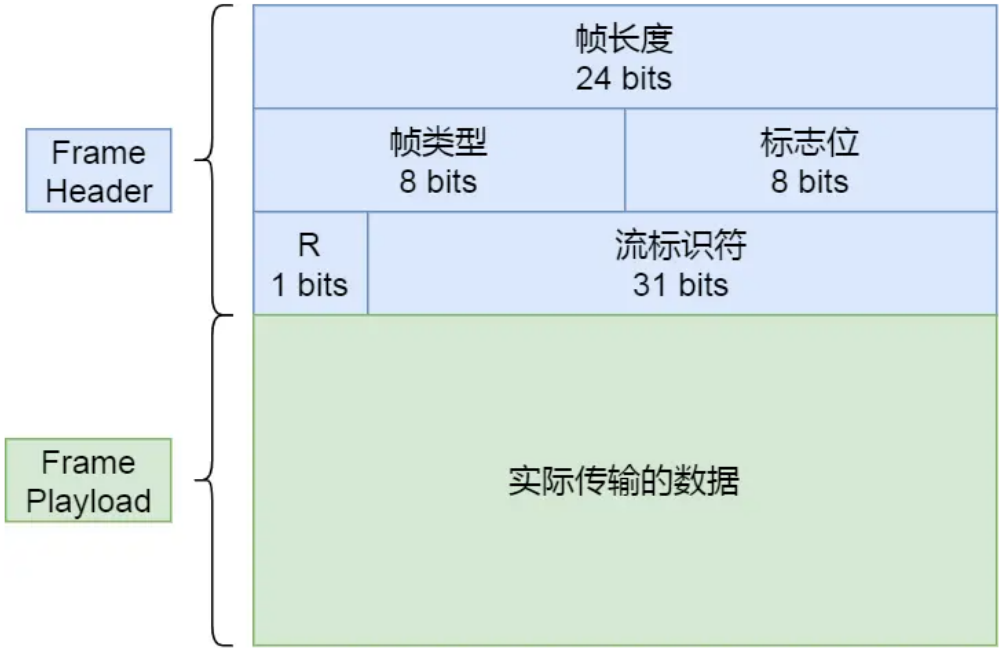
  2 版本没有用 gzip 进行压缩，而是采用 HPACK 算法，包含三个部分：
  - 静态字典 HTTP/2 为高频出现在头部的字符串和字段建立了一张静态表，它是写入到 HTTP/2 框架里的，不会变化的，静态表里共有 61 组。表中有的 Index 没有对应的 Header Value，这是因为这些 Value 并不是固定的而是变化的，这些 Value 都会经过 Huffman 编码后，才会发送出去。来看个具体的例子，下面这个 server 头部字段，在 HTTP/1.1 的形式如下：

    ```
    server: nghttpx\r\n
    ```

    算上冒号空格和末尾的换行符，共占用了 17 字节，而使用了静态表和 Huffman 编码，可以将它压缩成 8 字节，压缩率大概 47%。
  - 动态表编码（动态字典）静态表只包含了 61 种高频出现在头部的字符串，不在静态表范围内的头部字符串就要自行构建动态表，它的 Index 从 62 起步，会在编码解码的时候随时更新。

    比如，第一次发送时头部中的「User-Agent 」字段数据有上百个字节，经过 Huffman 编码发送出去后，客户端和服务器双方都会更新自己的动态表，添加一个新的 Index 号 62。那么在下一次发送的时候，就不用重复发这个字段的数据了，只用发 1 个字节的 Index 号就好了，因为双方都可以根据自己的动态表获取到字段的数据。

    所以，使得动态表生效有一个前提：必须同一个连接上，重复传输完全相同的 HTTP 头部。如果消息字段在 1 个连接上只发送了 1 次，或者重复传输时，字段总是略有变化，动态表就无法被充分利用了。
  - 二进制帧（哈夫曼编码）HTTP/2 把响应报文划分成了两类帧（**Frame** ），图中的 HEADERS（首部）和 DATA（消息负载）是帧的类型，也就是说一条 HTTP 响应，划分成了两类帧来传输，并且采用二进制来编码。



帧结构如上

– 并发传输通过 Stream 这个设计，多个 Stream 复用一条 TCP 连接，达到并发的效果，解决了 HTTP/1.1 队头阻塞的问题，提高了 HTTP 传输的吞吐量。多个 Stream 跑在一条 TCP 连接，同一个 HTTP 请求与响应是跑在同一个 Stream 中，HTTP 消息可以由多个 Frame 构成，一个 Frame 可以由多个 TCP 报文构成。在 HTTP/2 连接上，不同 Stream 的帧是可以乱序发送的（因此可以并发不同的 Stream ），因为每个帧的头部会携带 Stream ID 信息，所以接收端可以通过 Stream ID 有序组装成 HTTP 消息，而同一 Stream 内部的帧必须是严格有序的。

3.8 HTTP 和 RPC 的区别

· 服务方在 HTTP 中，你知道服务的域名，就可以通过 DNS 服务去解析得到它背后的 IP 地址，默认 80 端口。而 RPC 的话，就有些区别，一般会有专门的中间服务去保存服务名和 IP 信息，比如 Consul 或者 Etcd，甚至是 Redis。想要访问某个服务，就去这些中间服务去获得 IP 和端口信息。由于 DNS 也是服务发现的一种，所以也有基于 DNS 去做服务发现的组件，比如 CoreDNS。

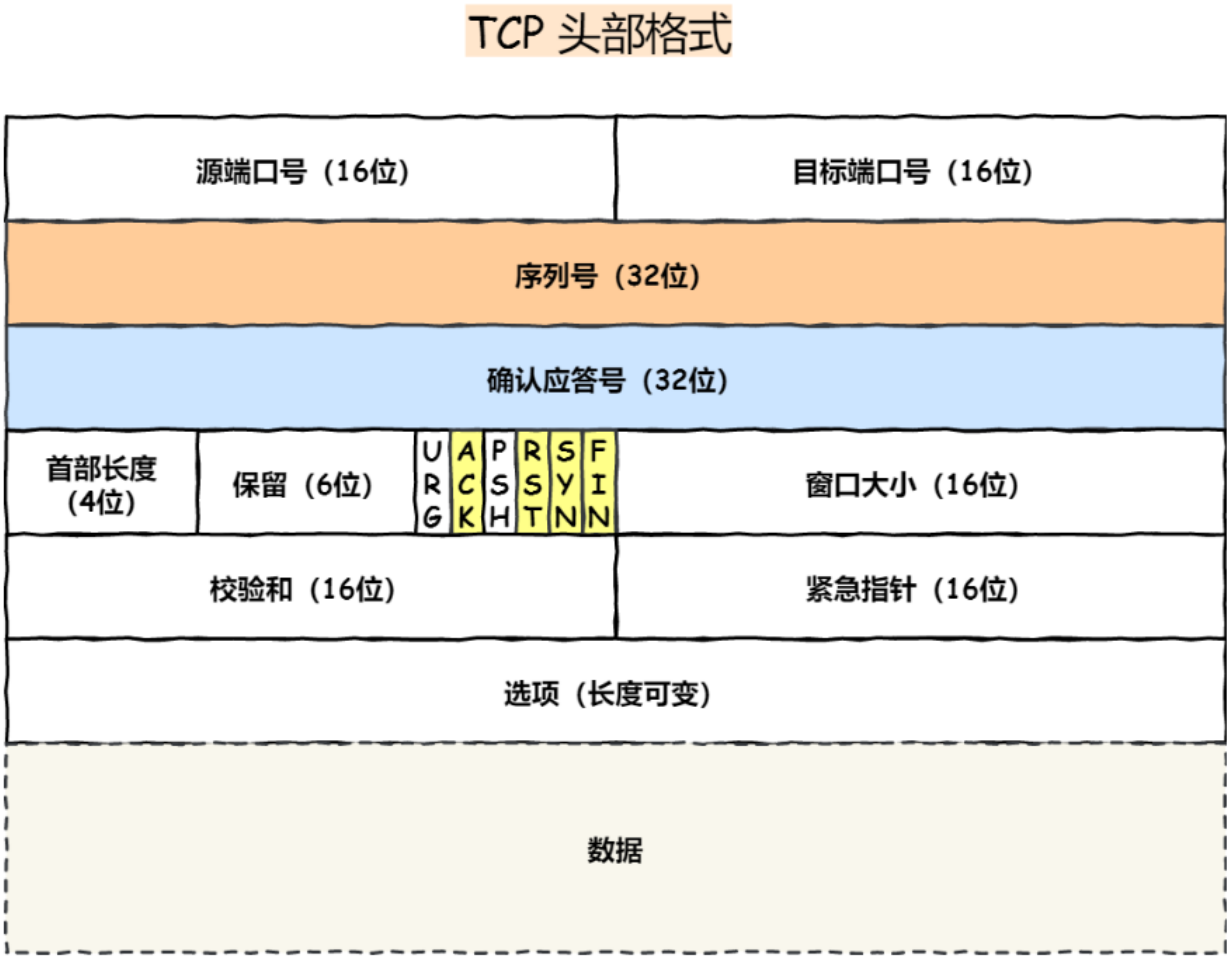
· 底层连接形式以主流的 HTTP/1.1 协议为例，其默认在建立底层 TCP 连接之后会一直保持这个连接（Keep Alive），之后的请求和响应都会复用这条连接。而 RPC 协议，也跟 HTTP 类似，也是通过建立 TCP 长链接进行数据交互，但不同的地方在于，RPC 协议一般还会再建个连接池，在请求量大的时候，建立多条连接放在池内，要发数据的时候就从池里取一条连接出来，用完放回去，下次再复用，可以说非常环保。

· 传输内容基于 TCP 传输的消息，说到底，无非都是消息头 Header 和消息体 Body。Header 是用于标记一些特殊信息，其中最重要的是消息体长度。Body 则是放我们真正需要传输的内容，而这些内容只能是二进制 01 串，毕竟计算机只认识这玩意。所以 TCP 传字符串和数字都问题不大，因为字符串可以转成编码再变成 01 串，而数字本身也能直接转为二进制。但结构体呢，我们得想个办法将它也转为二进制 01 串，这样的方案现在也有很多现成的，比如 Json，Protobuf。

4 TCP

4.1 基本认识

头部



序列号：在建立连接时由计算机生成的随机数作为其初始值，通过 SYN 包传给接收端主机，每发送一次数据，就「累加」一次该「数据字节数」的大小。用来解决网络包乱序问题。

确认应答号：指下一次「期望」收到的数据的序列号，发送端收到这个确认应答以后可以认为在这个序号以前的数据都已经被正常接收。用来解决丢包的问题。

- 什么是 TCP

  TCP 是面向连接的、可靠的、基于字节流的传输层通信协议。面向连接：一定是「一对一」才能连接，不能像 UDP 协议可以一个主机同时向多个主机发送消息，也就是一对多是无法做到的；可靠的：无论的网络链路中出现了怎样的链路变化，TCP 都可以保证一个报文一定能够到达接收端；字节流：用户消息通过 TCP 协议传输时，消息可能会被操作系统「分组」成多个的 TCP 报文，如果接收方的程序如果不知道「消息的边界」，是无法读出一个有效的用户消息的。并且 TCP 报文是「有序的」，当「前一个」TCP 报文没有收到的时候，即使它先收到了后面的 TCP 报文，那么也不能扔给应用层去处理，同时对「重复」的 TCP 报文会自动丢弃。

- 什么是 TCP 连接用于保证可靠性和流量控制维护的某些状态信息，这些信息的组合，包括 Socket、序列号和窗口大小称为连接。所以，建立一个 TCP 连接需要：

  - Socket：由 IP 地址和端口号组成
  - 序列号：用来解决乱序问题等

- 窗口大小：用来做流量控制

· 四元组源地址、源端口、目标地址、目标端口

源地址和目的地址的字段（32 位）是在 IP 头部中，作用是通过 IP 协议发送报文给对方主机。

源端口和目的端口的字段（16 位）是在 TCP 头部中，作用是告诉 TCP 协议应该把报文发给哪个进程。

· TCP 和 UDP 的区别

- 连接 TCP 是面向连接的传输层协议，传输数据前先要建立连接。UDP 是不需要连接，即刻传输数据。
- 服务对象 TCP 是一对一的两点服务，即一条连接只有两个端点。UDP 支持一对一、一对多、多对多的交互通信
- 可靠性 TCP 是可靠交付数据的，数据可以无差错、不丢失、不重复、按序到达。UDP 是尽最大努力交付，不保证可靠交付数据。但是我们可以基于 UDP 传输协议实现一个可靠的传输协议，比如 QUIC 协议。
- 拥塞控制、流量控制 TCP 有拥塞控制和流量控制机制，保证数据传输的安全性。UDP 则没有，即使网络非常拥堵了，也不会影响 UDP 的发送速率。
- 首部开销 TCP 首部长度较长，会有一定的开销，首部在没有使用「选项」字段时是 20 个字节，如果使用了「选项」字段则会变长的。UDP 首部只有 8 个字节，并且是固定不变的，开销较小。
- 传输方式 TCP 是流式传输，没有边界，但保证顺序和可靠。UDP 是一个包一个包的发送，是有边界的，但可能会丢包和乱序。
- 分片不同 TCP 的数据大小如果大于 MSS 大小，则会在传输层进行分片，目标主机收到后，也同样在传输层组装 TCP 数据包，如果中途丢失了一个分片，只需要传输丢失的这个分片。UDP 的数据大小如果大于 MTU 大小，则会在 IP 层进行分片，目标主机收到后，在 IP 层组装完数据，接着再传给传输层。

· 为什么 UDP 头部没有首部长度字段，而 TCP 首部有首部长度字段呢? TCP 有可变长的「选项」字段，而 UDP 头部长度则是不会变化的，无需多一个字段去记录 UDP 的首部长度。

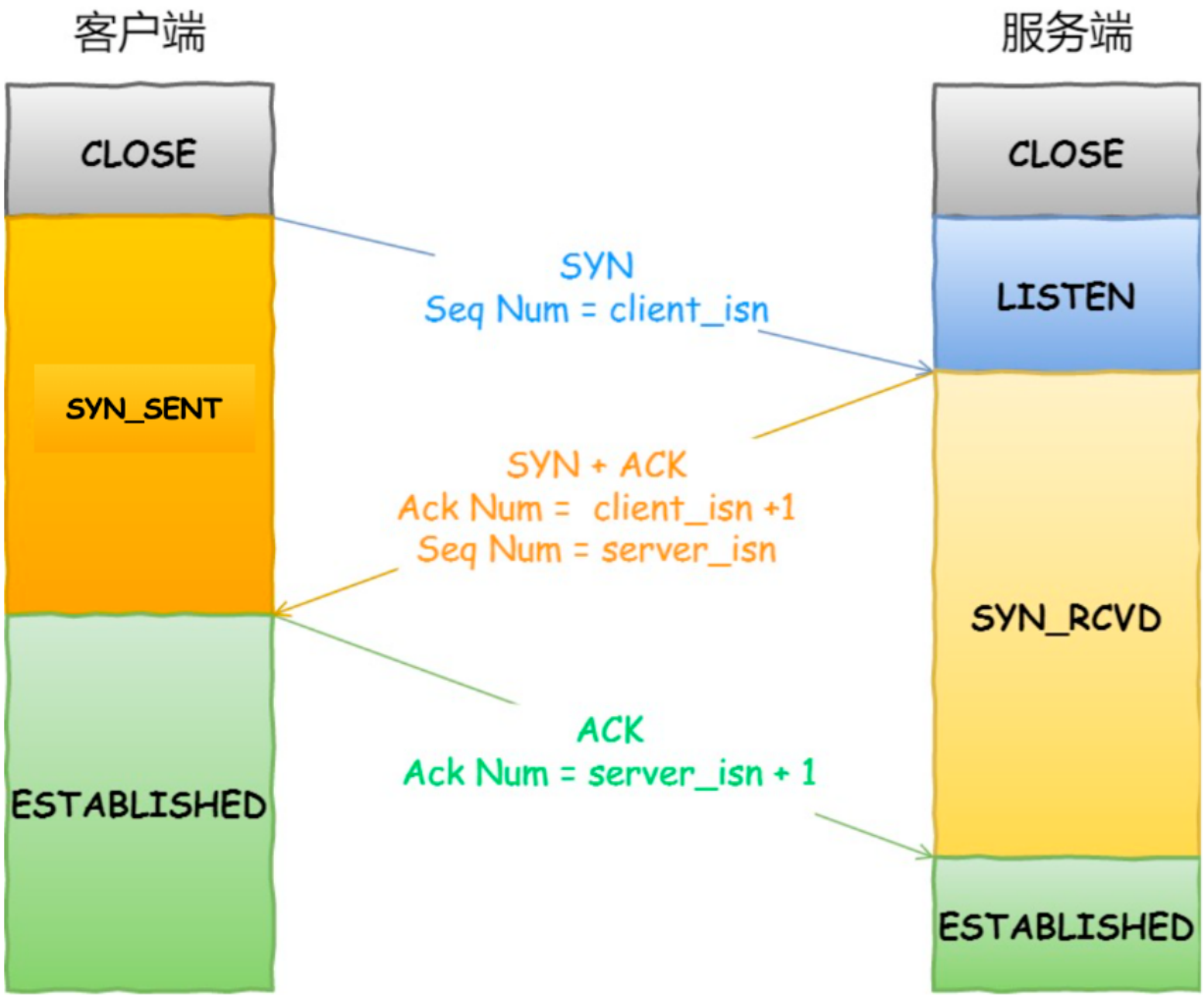
· 为什么 UDP 头部有包长度字段，而 TCP 没有呢?

TCP 计算负载长度：TCP 数据的长度 =IP 总长度-IP 首部长度-TCP 首部长度

其中 IP 总长度和 IP 首部长度，在 IP 首部格式是已知的。TCP 首部长度，则是在 TCP 首部格式已知的，所以就可以求得 TCP 数据的长度。

因为为了网络设备硬件设计和处理方便，首部长度需要是 4 字节的整数倍。如果去掉 UDP 的「包长度」字段，那 UDP 首部长度就不是 4 字节的整数倍了，所以我觉得这可能是为了补全 UDP 首部长度是 4 字节的整数倍，才补充了「包长度」字段。

4.2 三次握手



- 一开始，客户端和服务端都处于 CLOSE 状态。先是服务端主动监听某个端口，处于 LISTEN 状态
- 客户端会随机初始化序号（$client_{isn}$），将此序号置于 TCP 首部的「序号」字段中，同时把 SYN 标志位置为 1，表示 SYN 报文。接着把第一个 SYN 报文发送给服务端，表示向服务端发起连接，该报文不包含应用层数据，之后客户端处于 SYN-SENT 状态。
- 服务端收到客户端的 SYN 报文后，首先服务端也随机初始化自己的序号（$server_{isn}$），将此序号填入 TCP 首部的「序号」字段中，其次把 TCP 首部的「确认应答号」字段填入 $client_{isn} + 1$，接着把 SYN 和 ACK 标志位置为 1。最后把该报文发给客户端，该报文也不包含应用层数据，之后服务端处于 SYN-RCVD 状态。
- 客户端收到服务端报文后，还要向服务端回应最后一个应答报文，首先该应答报文 TCP 首部 ACK 标志位置为 1 ，其次「确认应答号」字段填入 $server_{isn} + 1$ ，最后把报文发送给服务端，这次报文可以携带客户到服务端的数据，之后客户端处于 ESTABLISHED 状态。
- 服务端收到客户端的应答报文后，也进入 ESTABLISHED 状态。

从上面的过程可以发现第三次握手是可以携带数据的，前两次握手是不可以携带数据的。具体流程请参考计网文件夹中的内容。

- 为什么每次建立 TCP 连接时，初始化的序列号都要求不一样呢?
  - 为了防止历史报文被下一个相同四元组的连接接收（主要方面）;
  - 为了安全性，防止黑客伪造的相同序列号的 TCP 报文被对方接收;

4.2.1 既然 IP 要分片，为什么 TCP 层还需要 MSS 呢？

MTU：一个网络包的最大长度，以太网中一般为 1500 字节；MSS：除去 IP 和 TCP 头部之后，一个网络包所能容纳的 TCP 数据的最大长度；

如果在 TCP 的整个报文（头部 + 数据）交给 IP 层进行分片，会有什么异常呢?

当 IP 层有一个超过 MTU 大小的数据（TCP 头部 + TCP 数据）要发送，那么 IP 层就要进行分片，把数据分片成若干片，保证每一个分片都小于 MTU。把一份 IP 数据报进行分片以后，由目标主机的 IP 层来进行重新组装后，再交给上一层 TCP 传输层。

这看起来井然有序，但这存在隐患的，那么当如果一个 IP 分片丢失，整个 IP 报文的所有分片都得重传。

因为 IP 层本身没有超时重传机制，它由传输层的 TCP 来负责超时和重传。

当某一个 IP 分片丢失后，接收方的 IP 层就无法组装成一个完整的 TCP 报文（头部 + 数据），也就无法将数据报文送到 TCP 层，所以接收方不会响应 ACK 给发送方，因为发送方迟迟收不到 ACK 确认报文，所以会触发超时重传，就会重发「整个 TCP 报文（头部 + 数据）」。

因此，可以得知由 IP 层进行分片传输，是非常没有效率的。

所以，为了达到最佳的传输效能 TCP 协议在建立连接的时候通常要协商双方的 MSS 值，当 TCP 层发现数据超过 MSS 时，则就先会进行分片，当然由它形成的 IP 包的长度也就不会大于 MTU ，自然也就不用 IP 分片了。

经过 TCP 层分片后，如果一个 TCP 分片丢失后，进行重发时也是以 MSS 为单位，而不用重传所有的分片，大大增加了重传的效率。

4.2.2 第一次握手丢失了，会发生什么？

当客户端想和服务端建立 TCP 连接的时候，首先第一个发的就是 SYN 报文，然后进入到 SYN_{SENT} 状态。

在这之后，如果客户端迟迟收不到服务端的 SYN-ACK 报文（第二次握手），就会触发「超时重传」机制，重传 SYN 报文，而且重传的 SYN 报文的序列号都是一样的。

不同版本的操作系统可能超时时间不同，有的 1 秒的，也有 3 秒的，这个超时时间是写死在内核里的，如果想要更改则需要重新编译内核，比较麻烦。

当客户端超时重传 3 次 SYN 报文后，由于 tcpsynretries 为 3，已达到最大重传次数，于是再等待一段时间（时间为上一次超时时间的 2 倍），如果还是没能收到服务端的第二次握手（SYN-ACK 报文），那么客户端就会断开连接。

4.2.3 第二次握手丢失了，会发生什么？

当服务端收到客户端的第一次握手后，就会回 SYN-ACK 报文给客户端，这个就是第二次握手，此时服务端会进入 SYN_{RCVD} 状态。

第二次握手的 SYN-ACK 报文其实有两个目的：

- 第二次握手里的 ACK，是对第一次握手的确认报文；
- 第二次握手里的 SYN，是服务端发起建立 TCP 连接的报文；

因为第二次握手报文里是包含对客户端的第一次握手的 ACK 确认报文，所以，如果客户端迟迟没有收到第二次握手，那么客户端就觉得可能自己的 SYN 报文（第一次握手）丢失了，于是客户端就会触发超时重传机制，重传 SYN 报文。

因为第二次握手报文里是包含对客户端的第一次握手的 ACK 确认报文，所以，如果客户端迟迟没有收到第二次握手，那么客户端就觉得可能自己的 SYN 报文（第一次握手）丢失了，于是客户端就会触发超时重传机制，重传 SYN 报文。

那么，如果第二次握手丢失了，服务端就收不到第三次握手，于是服务端这边会触发超时重传机制，重传 SYN-ACK 报文。

4.2.4 第三次握手丢失了，会发生什么？

客户端收到服务端的 SYN-ACK 报文后，就会给服务端回一个 ACK 报文，也就是第三次握手，此时客户端状态进入到 ESTABLISH 状态。

因为这个第三次握手的 ACK 是对第二次握手的 SYN 的确认报文，所以当第三次握手丢失了，如果服务端那一方迟迟收不到这个确认报文，就会触发超时重传机制，重传 SYN-ACK 报文，直到收到第三次握手，或者达到最大重传次数。

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。

4.3 四次挥手

客户端打算关闭连接，此时会发送一个 TCP 首部 FIN 标志位被置为 1 的报文，也即 FIN 报文，之后客户端进入 FIN_{WAIT1} 状态。

服务端收到该报文后，就向客户端发送 ACK 应答报文，接着服务端进入 $CLOSE_{WAIT}$ 状态

客户端收到服务端的 ACK 应答报文后，之后进入 FIN_{WAIT2} 状态。

等待服务端处理完数据后，也向客户端发送 FIN 报文，之后服务端进入 $LAST_{ACK}$ 状态。

客户端收到服务端的 FIN 报文后，回一个 ACK 应答报文，之后进入 $TIME_{WAIT}$ 状态。

服务端收到了 ACK 应答报文后，就进入了 CLOSE 状态，至此服务端已经完成连接的关闭。

客户端在经过 2MSL 一段时间后，自动进入 CLOSE 状态，至此客户端也完成连接的关闭。

每个方向都需要一个 FIN 和一个 ACK，因此通常被称为四次挥手。

主动关闭连接的，才有 $TIME_{WAIT}$ 状态。

4.3.1 为什么需要四次挥手？

关闭连接时，客户端向服务端发送 FIN 时，仅仅表示客户端不再发送数据了但是还能接收数据。

服务端收到客户端的 FIN 报文时，先回一个 ACK 应答报文，而服务端可能还有数据需要处理和发送，等服务端不再发送数据时，才发送 FIN 报文给客户端来表示同意现在关闭连接。

从上面过程可知，服务端通常需要等待完成数据的发送和处理，所以服务端的 ACK 和 FIN 一般都会分开发送，因此是需要四次挥手。

实际上，四次挥手可以变为三次，具体看计网下的内容。

4.3.2 第一次挥手丢失了，会发生什么？

当客户端（主动关闭方）调用 close 函数后，就会向服务端发送 FIN 报文，试图与服务端断开连接，此时客户端的连接进入到 FIN_{WAIT1} 状态。

正常情况下，如果能及时收到服务端（被动关闭方）的 ACK，则会很快变为 FIN_{WAIT2} 状态。

如果第一次挥手丢失了，那么客户端迟迟收不到被动方的 ACK 的话，也就会触发超时重传机制，重传 FIN 报文，重发次数由 $tcp_{orphanretries}$ 参数控制。

当客户端超时重传 3 次 FIN 报文后，由于 tcporphanretries 为 3，已达到最大重传次数，于是再等待一段时间（时间为上一次超时时间的 2 倍），如果还是没能收到服务端的第二次挥手（ACK 报文），那么客户端就会断开连接。

4.3.3 第二次挥手丢失了，会发生什么？

当服务端收到客户端的第一次挥手后，就会先回一个 ACK 确认报文，此时服务端的连接进入到 $\text{CLOSE}_{\text{WAIT}}$ 状态。

在前面我们也提了，ACK 报文是不会重传的，所以如果服务端的第二次挥手丢失了，客户端就会触发超时重传机制，重传 FIN 报文，直到收到服务端的第二次挥手，或者达到最大的重传次数。

当客户端超时重传 2 次 FIN 报文后，由于 tcporphanretries 为 2，已达到最大重传次数，于是再等待一段时间（时间为上一次超时时间的 2 倍），如果还是没能收到服务端的第二次挥手（ACK 报文），那么客户端就会断开连接。

当客户端收到第二次挥手，也就是收到服务端发送的 ACK 报文后，客户端就会处于 $\text{FIN}_{\text{WAIT2}}$ 状态，在这个状态需要等服务端发送第三次挥手，也就是服务端的 FIN 报文。

对于 close 函数关闭的连接，由于无法再发送和接收数据，所以 $\text{FIN}_{\text{WAIT2}}$ 状态不可以持续太久，而 $\text{tcp}_{\text{fintimeout}}$ 控制了这个状态下连接的持续时长，默认值是 60 秒。

但是注意，如果主动关闭方使用 shutdown 函数关闭连接，指定了只关闭发送方向，而接收方向并没有关闭，那么意味着主动关闭方还是可以接收数据的。

此时，如果主动关闭方一直没收到第三次挥手，那么主动关闭方的连接将会一直处于 $\text{FIN}_{\text{WAIT2}}$ 状态（$\text{tcp}_{\text{fintimeout}}$ 无法控制 shutdown 关闭的连接）。

4.3.4 第三次挥手丢失了，会发生什么？

当服务端（被动关闭方）收到客户端（主动关闭方）的 FIN 报文后，内核会自动回复 ACK，同时连接处于 $\text{CLOSE}_{\text{WAIT}}$ 状态，顾名思义，它表示等待应用进程调用 close 函数关闭连接。

此时，内核是没有权利替代进程关闭连接，必须由进程主动调用 close 函数来触发服务端发送 FIN 报文。

服务端处于 CLOSEWAIT 状态时，调用了 close 函数，内核就会发出 FIN 报文，同时连接进入 LASTACK 状态，等待客户端返回 ACK 来确认连接关闭。

如果迟迟收不到这个 ACK，服务端就会重发 FIN 报文，重发次数仍然由 $\text{tcp}_{\text{orphanretries}}$ 参数控制，这与客户端重发 FIN 报文的重传次数控制方式是一样的。

当服务端重传第三次挥手报文的次数达到了 3 次后，由于 tcporphanretries 为 3，达到了重传最大次数，于是再等待一段时间（时间为上一次超时时间的 2 倍），如果还是没能收到客户端的第四次挥手（ACK 报文），那么服务端就会断开连接。

客户端因为是通过 close 函数关闭连接的，处于 FINWAIT2 状态是有时长限制的，如果 tcpfintimeout 时间内还是没能收到服务端的第三次挥手（FIN 报文），那么客户端就会断开连接。

4.3.5 第四次挥手丢失了，会发生什么？

当客户端收到服务端的第三次挥手的 FIN 报文后，就会回 ACK 报文，也就是第四次挥手，此时客户端连接进入 $\text{TIME}_{\text{WAIT}}$ 状态。

在 Linux 系统，$\text{TIME}_{\text{WAIT}}$ 状态会持续 2MSL 后才会进入关闭状态。

然后，服务端（被动关闭方）没有收到 ACK 报文前，还是处于 LAST_{ACK} 状态。

如果第四次挥手的 ACK 报文没有到达服务端，服务端就会重发 FIN 报文，重发次数仍然由前面介绍过的 $\text{tcp}_{\text{orphanretries}}$ 参数控制。

当服务端重传第三次挥手报文达到 2 时，由于 tcporphanretries 为 2，达到了最大重传次数，于是再等待一段时间（时间为上一次超时时间的 2 倍），如果还是没能收到客户端的第四次挥手（ACK 报文），那么服务端就会断开连接。

客户端在收到第三次挥手后，就会进入 $TIME_{WAIT}$ 状态，开启时长为 2MSL 的定时器，如果途中再次收到第三次挥手（FIN 报文）后，就会重置定时器，当等待 2MSL 时长后，客户端就会断开连接。

4.3.6 为什么 $TIME_{WAIT}$ 等待的时间是 2MSL?

MSL 是 Maximum Segment Lifetime，报文最大生存时间，它是任何报文在网络上存在的最长时间，超过这个时间报文将被丢弃。因为 TCP 报文基于是 IP 协议的，而 IP 头中有一个 TTL 字段，是 IP 数据报可以经过的最大路由数，每经过一个处理他的路由器此值就减 1，当此值为 0 则数据报将被丢弃，同时发送 ICMP 报文通知源主机。

MSL 与 TTL 的区别：MSL 的单位是时间，而 TTL 是经过路由跳数。所以 MSL 应该要大于等于 TTL 消耗为 0 的时间，以确保报文已被自然消亡。

TTL 的值一般是 64，Linux 将 MSL 设置为 30 秒，意味着 Linux 认为数据报文经过 64 个路由器的时间不会超过 30 秒，如果超过了，就认为报文已经消失在网络中了。

$TIME_{WAIT}$ 等待 2 倍的 MSL，比较合理的解释是：网络中可能存在来自发送方的数据包，当这些发送方的数据包被接收方处理后又会向对方发送响应，所以一来一回需要等待 2 倍的时间。

比如，如果被动关闭方没有收到断开连接的最后的 ACK 报文，就会触发超时重发 FIN 报文，另一方接收到 FIN 后，会重发 ACK 给被动关闭方，一来一去正好 2 个 MSL。

可以看到 2MSL 时长这其实是相当于至少允许报文丢失一次。比如，若 ACK 在一个 MSL 内丢失，这样被动方重发的 FIN 会在第 2 个 MSL 内到达，$TIME_{WAIT}$ 状态的连接可以应对。

2MSL 的时间是从客户端接收到 FIN 后发送 ACK 开始计时的。如果在 TIME-WAIT 时间内，因为客户端的 ACK 没有传输到服务端，客户端又接收到了服务端重发的 FIN 报文，那么 2MSL 时间将重新计时。

4.3.7 为什么需要 $TIME_{WAIT}$ 状态？

- 防止历史连接中的数据，被后面相同四元组的连接错误的接收；为了理解原因，需要先了解序列号 SEQ 和初始序列号 ISN：

  1. 序列号，是 TCP 一个头部字段，标识了 TCP 发送端到 TCP 接收端的数据流的一个字节，因为 TCP 是面向字节流的可靠协议，为了保证消息的顺序性和可靠性，TCP 为每个传输方向上的每个字节都赋予了一个编号，以便于传输成功后确认、丢失后重传以及在接收端保证不会乱序。序列号是一个 32 位的无符号数，因此在到达 4G 之后再循环回到 0。
  2. 初始序列号，在 TCP 建立连接的时候，客户端和服务端都会各自生成一个初始序列号，它是基于时钟生成的一个随机数，来保证每个连接都拥有不同的初始序列号。初始化序列号可被视为一个 32 位的计数器，该计数器的数值每 4 微秒加 1，循环一次需要 4.55 小时。

  序列号和初始化序列号并不是无限递增的，会发生回绕为初始值的情况，这意味着无法根据序列号来判断新老数据。为了防止历史连接中的数据，被后面相同四元组的连接错误的接收，因此 TCP 设计了 $TIME_{WAIT}$ 状态，状态会持续 2MSL 时长，这个时间足以让两个方向上的数据包都被丢弃，使得原来连接的数据包在网络中都自然消失，再出现的数据包一定都是新建立连接所产生的。

- 保证「被动关闭连接」的一方，能被正确的关闭；TIME-WAIT 作用是等待足够的时间以确保最后的 ACK 能让被动关闭方接收，从而帮助其正常关闭。

  如果客户端（主动关闭方）最后一次 ACK 报文（第四次挥手）在网络中丢失了，那么按照 TCP 可靠性原则，服务端（被动关闭方）会重发 FIN 报文。

假设客户端没有 $TIME_{WAIT}$ 状态，而是在发完最后一次回 ACK 报文就直接进入 CLOSE 状态，如果该 ACK 报文丢失了，服务端则重传的 FIN 报文，而这时客户端已经进入到关闭状态了，在收到服务端重传的 FIN 报文后，就会回 RST 报文。

服务端收到这个 RST 并将其解释为一个错误（Connection reset by peer），这对于一个可靠的协议来说不是一个优雅的终止方式。

服务端收到这个 RST 并将其解释为一个错误（Connection reset by peer）进行异常终止，这对于一个可靠的协议来说不是一个优雅的终止方式。

为了防止这种情况出现，客户端必须等待足够长的时间，确保服务端能够收到 ACK，如果服务端没有收到 ACK，那么就会触发 TCP 重传机制，服务端会重新发送一个 FIN，这样一去一来刚好两个 MSL 的时间。

客户端在收到服务端重传的 FIN 报文时，$TIME_{WAIT}$ 状态的等待时间，会重置回 2MSL。

4.3.8 如何优化 $TIME_{WAIT}$?

1. net.ipv4.tcptwreuse 和 $tcp_{timestamps}$ 如下的 Linux 内核参数开启后，则可以复用处于 $TIME_{WAIT}$ 的 socket 为新的连接所用。

   有一点需要注意的是，tcptwreuse 功能只能用客户端（连接发起方），因为开启了该功能，在调用 connect() 函数时，内核会随机找一个 $time_{wait}$ 状态超过 1 秒的连接给新的连接复用。

   net.ipv4.$tcp_{twreuse}$ = 1 使用这个选项，还有一个前提，需要打开对 TCP 时间戳的支持，即

   net.ipv4.$tcp_{timestamps}$=1（默认即为 1）这个时间戳的字段是在 TCP 头部的「选项」里，它由一共 8 个字节表示时间戳，其中第一个 4 字节字段用来保存发送该数据包的时间，第二个 4 字节字段用来保存最近一次接收对方发送到达数据的时间。

   由于引入了时间戳，我们在前面提到的 2MSL 问题就不复存在了，因为重复的数据包会因为时间戳过期被自然丢弃。

2. net.ipv4.$tcpmaxtw_{buckets}$

   这个值默认为 18000，当系统中处于 TIMEWAIT 的连接一旦超过这个值时，系统就会将后面的 TIMEWAIT 连接状态重置，这个方法比较暴力。

3. 程序中使用 SO_{LINGER}

   我们可以通过设置 socket 选项，来设置调用 close 关闭连接行为。

   ```
   struct linger so_linger;
   so_linger.l_onoff = 1;
   so_linger.l_linger = 0;
   setsockopt(s, SOL_SOCKET, SO_LINGER, &so_linger,sizeof(so_linger));
   ```

   如果 $l_{onoff\ 为非}$ 0，且 $l_{linger\ 值为}$ 0，那么调用 close 后，会立该发送一个 RST 标志给对端，该 TCP 连接将跳过四次挥手，也就跳过了 $TIME_{WAIT\ 状态}$，直接关闭。

如果服务端要避免过多的 TIMEWAIT 状态的连接，就永远不要主动断开连接，让客户端去断开，由分布在各处的客户端去承受 TIMEWAIT。

4.3.9 服务端出现大量 $TIME_{WAIT}$ 状态的原因有哪些？

首先要知道 TIMEWAIT 状态是主动关闭连接方才会出现的状态，所以如果服务器出现大量的 TIMEWAIT 状态的 TCP 连接，就是说明服务器主动断开了很多 TCP 连接。

4.3.10 什么状态下，服务端会主动断开连接呢？

1. HTTP 没有使用长连接关闭 HTTP 长连接机制后，每次请求都要经历这样的过程：建立 TCP -> 请求资源 -> 响应资源 -> 释放连接，那么此方式就是 HTTP 短连接。只要任意一方的 HTTP header 中有 Connection:close 信息，就无法使用 HTTP 长连接机制，这样在完成一次 HTTP 请求/处理后，就会关闭连接。

   不过，根据大多数 Web 服务的实现，不管哪一方禁用了 HTTP Keep-Alive，都是由服务端主动关闭连接，那么此时服务端上就会出现 $TIME_{WAIT}$ 状态的连接。

   因此，当服务端出现大量的 $TIME_{WAIT}$ 状态连接的时候，可以排查下是否客户端和服务端都开启了 HTTP Keep-Alive，因为任意一方没有开启 HTTP Keep-Alive，都会导致服务端在处理完一个 HTTP 请求后，就主动关闭连接，此时服务端上就会出现大量的 $TIME_{WAIT}$ 状态的连接。

2. HTTP 长连接超时

   HTTP 长连接的特点是，只要任意一端没有明确提出断开连接，则保持 TCP 连接状态。

   HTTP 长连接可以在同一个 TCP 连接上接收和发送多个 HTTP 请求/应答，避免了连接建立和释放的开销。

   假设设置了 HTTP 长连接的超时时间是 60 秒，nginx 就会启动一个「定时器」，如果客户端在完后一个 HTTP 请求后，在 60 秒内都没有再发起新的请求，定时器的时间一到，nginx 就会触发回调函数来关闭该连接，那么此时服务端上就会出现 $TIME_{WAIT}$ 状态的连接。

3. HTTP 长连接的请求数量达到上限

   Web 服务端通常会有个参数，来定义一条 HTTP 长连接上最大能处理的请求数量，当超过最大限制时，就会主动关闭连接。

   比如 nginx 的 keepaliverequests 这个参数，这个参数是指一个 HTTP 长连接建立之后，nginx 就会为这个连接设置一个计数器，记录这个 HTTP 长连接上已经接收并处理的客户端请求的数量。如果达到这个参数设置的最大值时，则 nginx 会主动关闭这个长连接，那么此时服务端上就会出现 TIMEWAIT 状态的连接。

   $keepalive_{requests}$ 参数的默认值是 100 ，意味着每个 HTTP 长连接最多只能跑 100 次请求，这个参数往往被大多数人忽略，因为当 QPS (每秒请求数) 不是很高时，默认值 100 凑合够用。

   对于一些 QPS 比较高的场景，比如超过 10000 QPS，甚至达到 30000 , 50000 甚至更高，如果 keepaliverequests 参数值是 100，这时候就 nginx 就会很频繁地关闭连接，那么此时服务端上就会出大量的 TIMEWAIT 状态。

4.3.11 服务器大量出现 $CLOSE_{WAIT}$ 状态的原因有哪些呢？

CLOSEWAIT 状态是「被动关闭方」才会有的状态，而且如果「被动关闭方」没有调用 close 函数关闭连接，那么就无法发出 FIN 报文，从而无法使得 CLOSEWAIT 状态的连接转变为 $LAST_{ACK}$ 状态。

所以，当服务端出现大量 $CLOSE_{WAIT}$ 状态的连接的时候，说明服务端的程序没有调用 close 函数关闭连接。

4.3.12 如果已经建立了连接，但是客户端突然出现了故障怎么办？

客户端出现故障指的是客户端的主机发生了宕机，或者断电的场景。发生这种情况的时候，如果服务端一直不会发送数据给客户端，那么服务端是永远无法感知到客户端宕机这个事件的，也就是服务端的 TCP 连接将一直处于 ESTABLISH 状态，占用着系统资源。

为此，TCP 提出了保活机制

1. 保活机制

   定义一个时间段，在这个时间段内，如果没有任何连接相关的活动，TCP 保活机制会开始作用，每隔一个时间间隔，发送一个探测报文，该探测报文包含的数据非常少，如果连续几个探测报文都没有得到响应，则认为当前的 TCP 连接已经死亡，系统内核将错误信息通知给上层应用程序。

   如果开启了 TCP 保活，需要考虑以下几种情况：

   - 第一种，对端程序是正常工作的。当 TCP 保活的探测报文发送给对端，对端会正常响应，这样 TCP 保活时间会被重置，等待下一个 TCP 保活时间的到来。
   - 第二种，对端主机宕机并重启。当 TCP 保活的探测报文发送给对端后，对端是可以响应的，但由于没有该连接的有效信息，会产生一个 RST 报文，这样很快就会发现 TCP 连接已经被重置。
   - 第三种，是对端主机宕机（注意不是进程崩溃，进程崩溃后操作系统在回收进程资源的时候，会发送 FIN 报文，而主机宕机则是无法感知的，所以需要 TCP 保活机制来探测对方是不是发生了主机宕机），或对端由于其他原因导致报文不可达。当 TCP 保活的探测报文发送给对端后，石沉大海，没有响应，连续几次，达到保活探测次数后，TCP 会报告该 TCP 连接已经死亡。

4.3.13 如果已经建立了连接，但是服务端的进程崩溃会发生什么？

TCP 的连接信息是由内核维护的，所以当服务端的进程崩溃后，内核需要回收该进程的所有 TCP 连接资源，于是内核会发送第一次挥手 FIN 报文，后续的挥手过程也都是在内核完成，并不需要进程的参与，所以即使服务端的进程退出了，还是能与客户端完成 TCP 四次挥手的过程。

在 kill 掉进程后，服务端会发送 FIN 报文，与客户端进行四次挥手。

4.4 TCP 重传、滑动窗口、流量控制、拥塞控制

4.4.1 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时，接收端主机会返回一个确认应答消息，表示已收到消息。在网络环境复杂的情况下，可能会发生丢包，这时候就需要重传机制来工作。

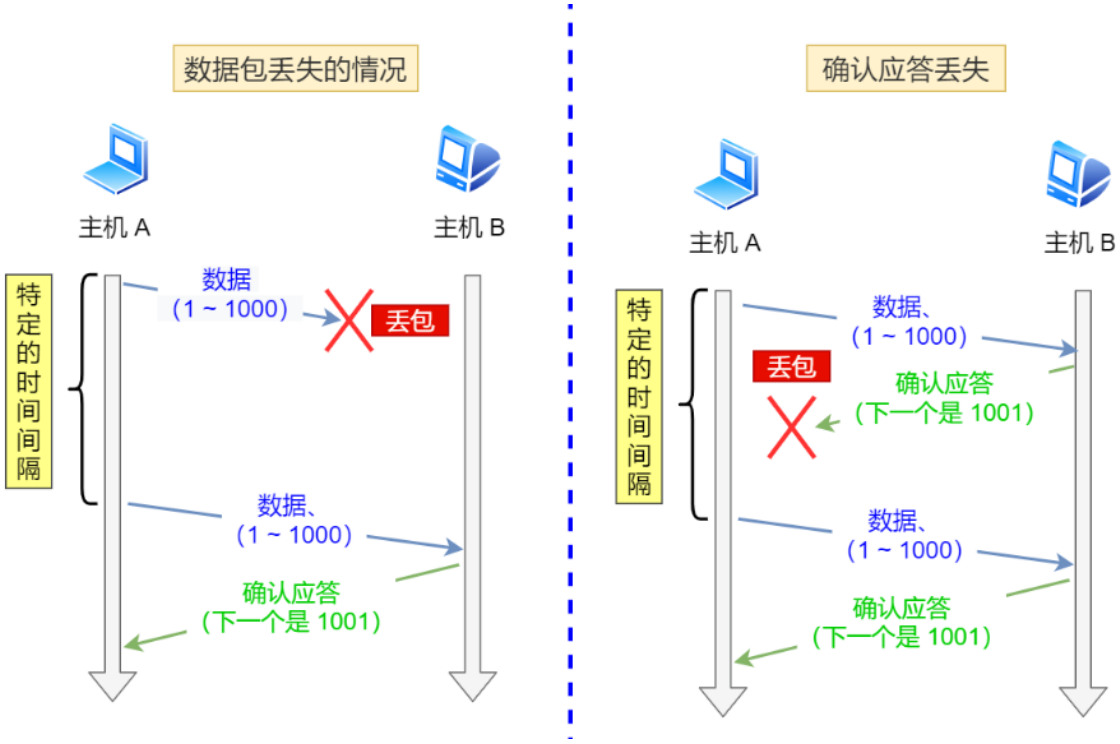
1. 超时重传

   重传机制的其中一个方式，就是在发送数据时，设定一个定时器，当超过指定的时间后，没有收到对方的 ACK 确认应答报文，就会重发该数据，也就是我们常说的超时重传。

   两种发生超时重传的情况

   (a) 数据包丢失

   (b) 确认应答丢失



两种情况

超时重传时间是以 RTO （Retransmission Timeout 超时重传时间）表示。

假设在重传的情况下，超时时间 RTO 「较长或较短」时，会发生什么事情呢?

- 当超时时间 RTO 较大时，重发就慢，丢了老半天才重发，没有效率，性能差；
- 当超时时间 RTO 较小时，会导致可能并没有丢就重发，于是重发的就快，会增加网络拥塞，导致更多的超时，更多的超时导致更多的重发。

超时重传时间 RTO 的值应该略大于报文往返 RTT 的值。

「报文往返 RTT 的值」是经常变化的，因为我们的网络也是时常变化的。也就因为「报文往返 RTT 的值」是经常波动变化的，所以「超时重传时间 RTO 的值」应该是一个动态变化的值。

如果超时重发的数据，再次超时的时候，又需要重传的时候，TCP 的策略是超时间隔加倍。

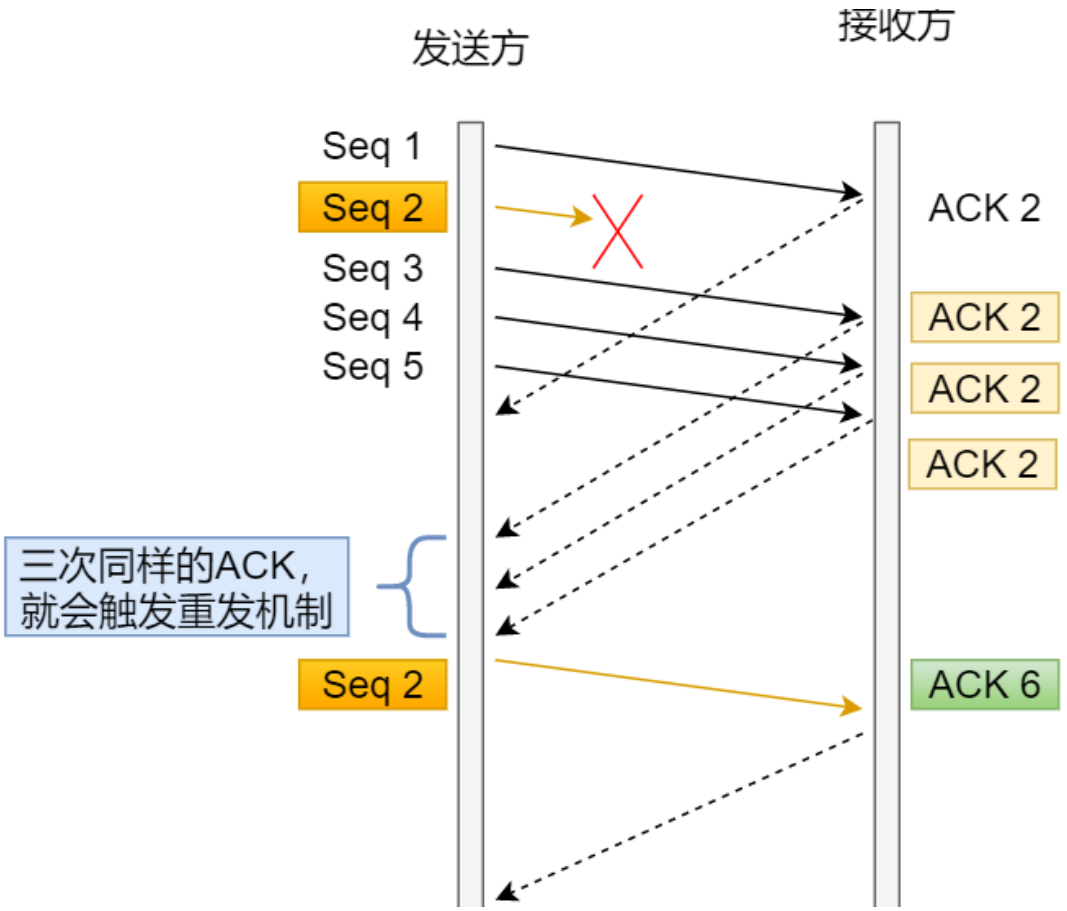
也就是每当遇到一次超时重传的时候，都会将下一次超时时间间隔设为先前值的两倍。两次超时，就说明网络环境差，不宜频繁反复发送。

超时重传存在的问题是，超市周期可能较长，有没有更快的方式?

于是就可以用「快速重传」机制来解决超时重发的时间等待。

2. 快速重传

快速重传（Fast Retransmit）机制，它不以时间为驱动，而是以数据驱动重传。



机制原理

发送端收到了三个 Ack = 2 的确认，知道了 Seq2 还没有收到，就会在定时器过期之前，重传丢失的 Seq2。

所以，快速重传的工作方式是当收到三个相同的 ACK 报文时，会在定时器过期之前，重传丢失的报文段。

快速重传机制只解决了一个问题，就是超时时间的问题，但是它依然面临着另外一个问题。**就是重传的时候，是重传一个，还是重传所有的问题。**

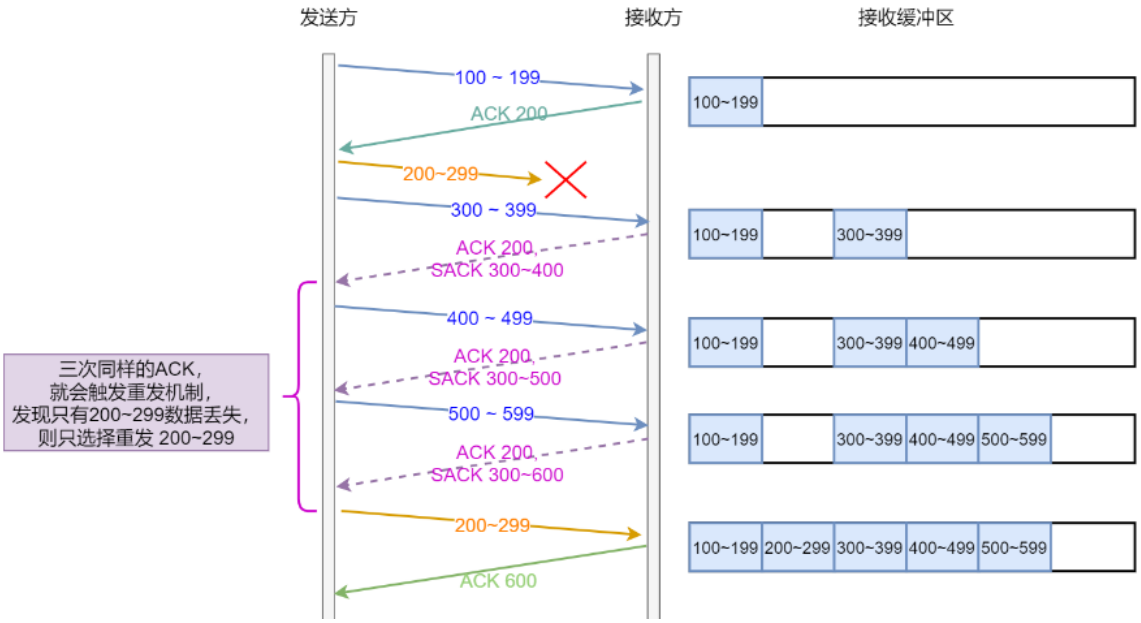
为了解决不知道该重传哪些 TCP 报文，于是就有 SACK 方法。

3. SACK 方法

SACK（Selective Acknowledgment），选择性确认。

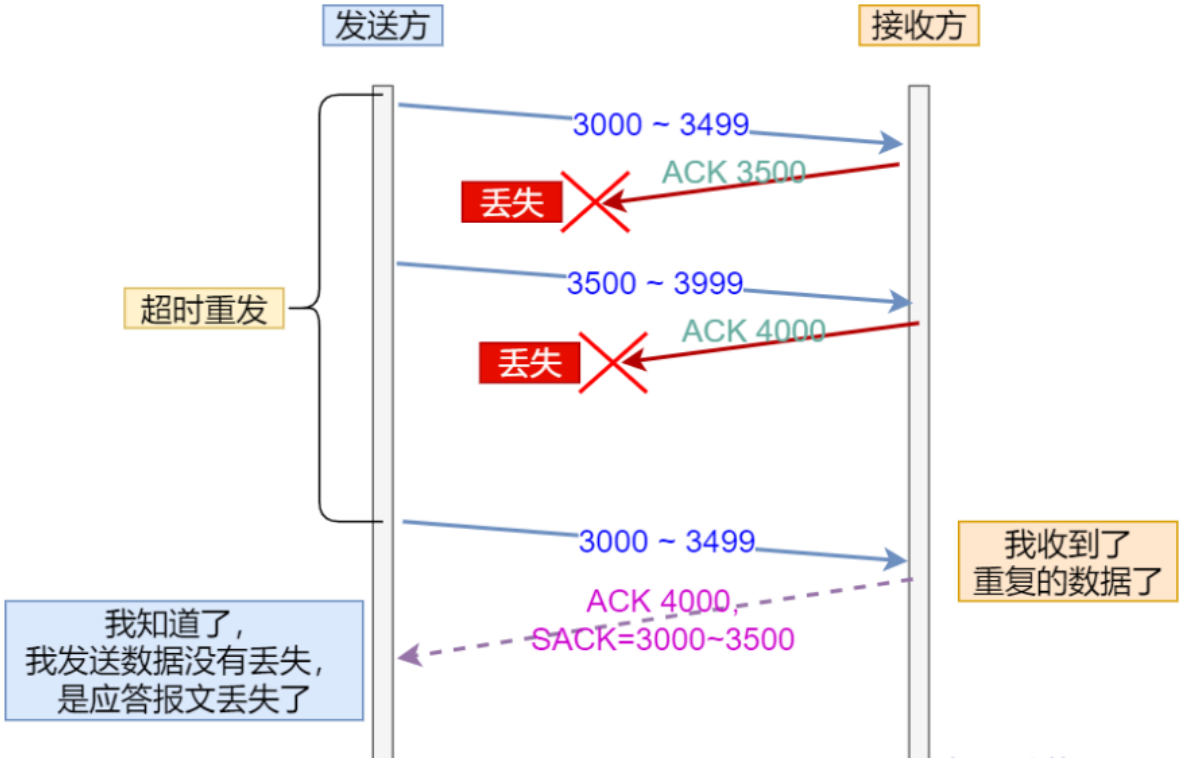
这种方式需要在 TCP 头部「选项」字段里加一个 SACK 的东西，它可以将已收到的数据的信息发送给「发送方」，这样发送方就可以知道哪些数据收到了，哪些数据没收到，知道了这些信息，就可以只重传丢失的数据。

如下图，发送方收到了三次同样的 ACK 确认报文，于是就会触发快速重发机制，通过 SACK 信息发现只有 200~299 这段数据丢失，则重发时，就只选择了这个 TCP 段进行重复。



4. Duplicate SACK 方法

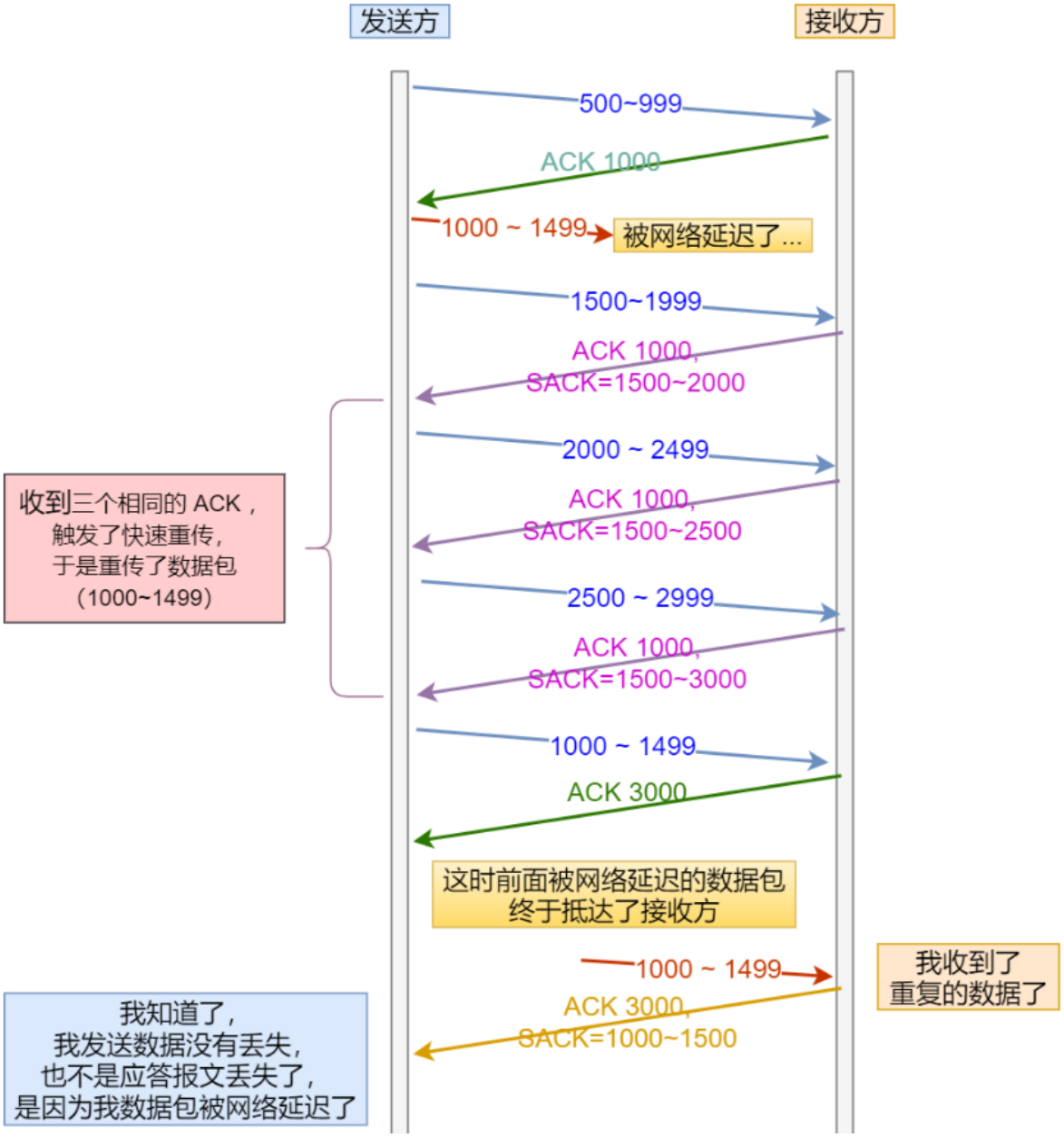
Duplicate SACK 又称 D-SACK，其主要使用了 SACK 来告诉「发送方」有哪些数据被重复接收了。



在这个例子中，「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。



在这个例子中，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 有这几个好处：

(a) 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;

(b) 可以知道是不是「发送方」的数据包被网络延迟了;

(c) 可以知道网络中是不是把「发送方」的数据包给复制了;

4.4.2 滑动窗口

TCP 是每发送一个数据，都要进行一次确认应答。当上一个数据包收到了应答了，再发送下一个。

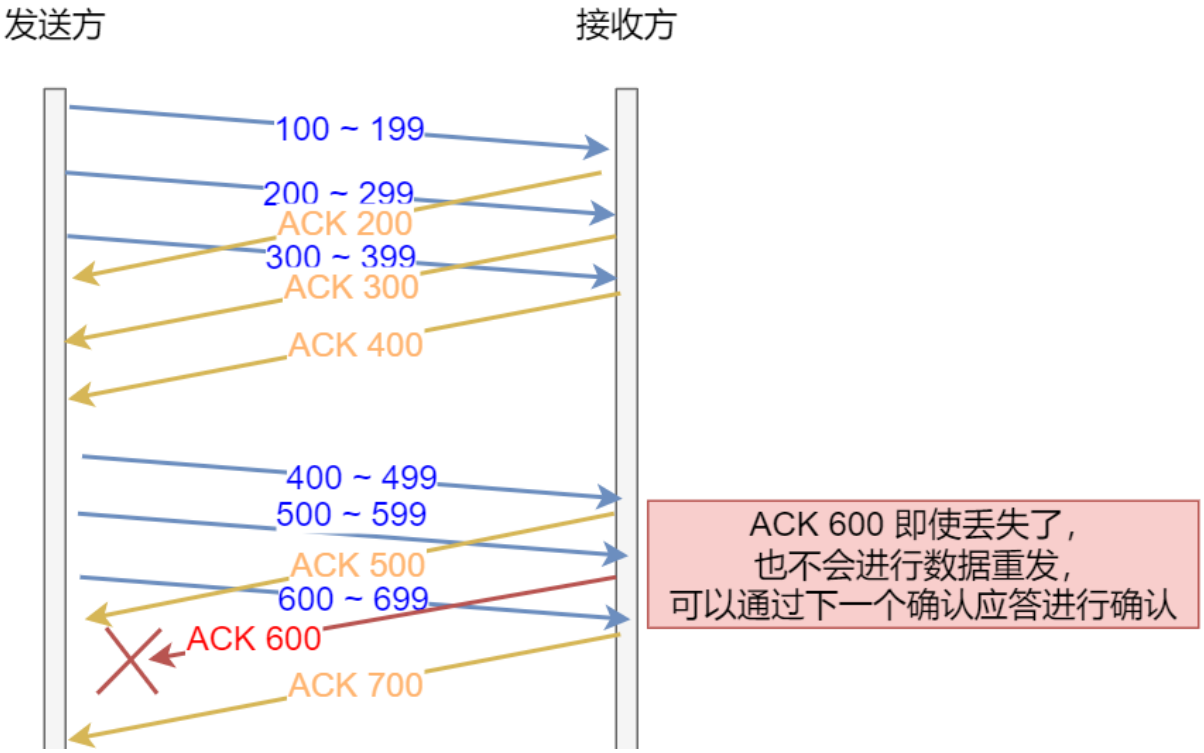
这样的传输方式有一个缺点：数据包的往返时间越长，通信的效率就越低。

为解决这个问题，TCP 引入了窗口这个概念。即使在往返时间较长的情况下，它也不会降低网络通信的效率。

那么有了窗口，就可以指定窗口大小，窗口大小就是指无需等待确认应答，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，发送方主机在等到确认应答返回之前，必须在缓冲区中保留已发送的数据。如果按期收到确认应答，此时数据就可以从缓存区清除。

如下，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」。



图中的 ACK 600 确认应答报文丢失，也没问题，因为可以通过下一个 e 确认应答进行确认，只要发送方收到了 ACK 700 确认应答，就意味着 700 之前的所有数据接收方都受到了，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？TCP 头里有一个字段叫 Window，也就是窗口大小。

  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，通常窗口的大小是由接收方的窗口大小来决定的。

  发送方发送的数据大小不能超过接收方的窗口大小，否则接收方就无法正常接收到数据。

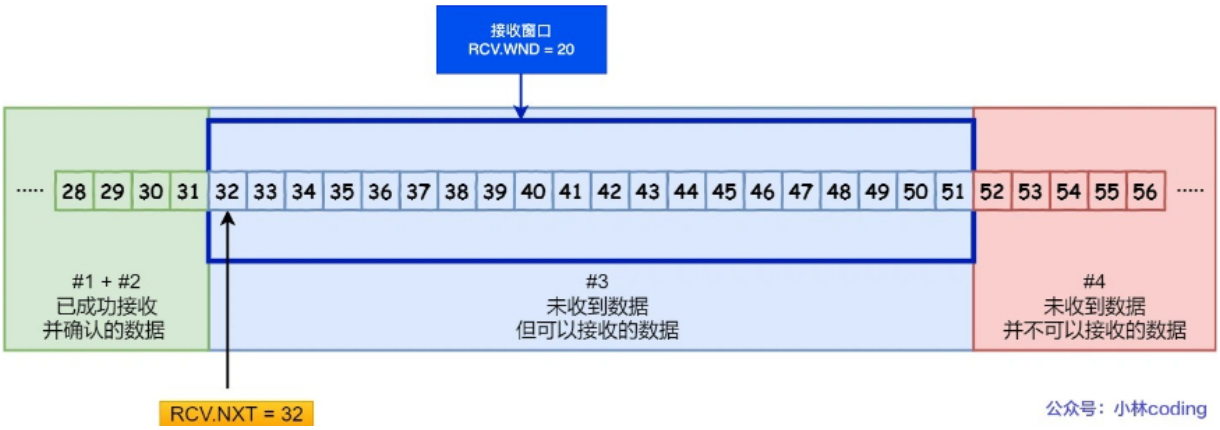
  TCP 滑动窗口方案使用三个指针来跟踪在四个传输类别中的每一个类别中的字节。其中两个指针是绝对指针（指特定的序列号），一个是相对指针（需要做偏移）。

  SND: WND 表示发送窗口的大小，USA(Send Unacknoledged) 绝对指针，指向已发送但还没确认的第一个字节的序列号，NXT 指向 o 未发生能够但可发送范围的第一个字节的序列号。

  可用窗口大小的计算：SND.WND-(SND.NXT-SND.USA)

  接收方的窗口：

- #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；



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。

- 接收窗口和发送窗口的大小相等吗？并不是完全相等，接收窗口的大小是约等于发送窗口的大小的。

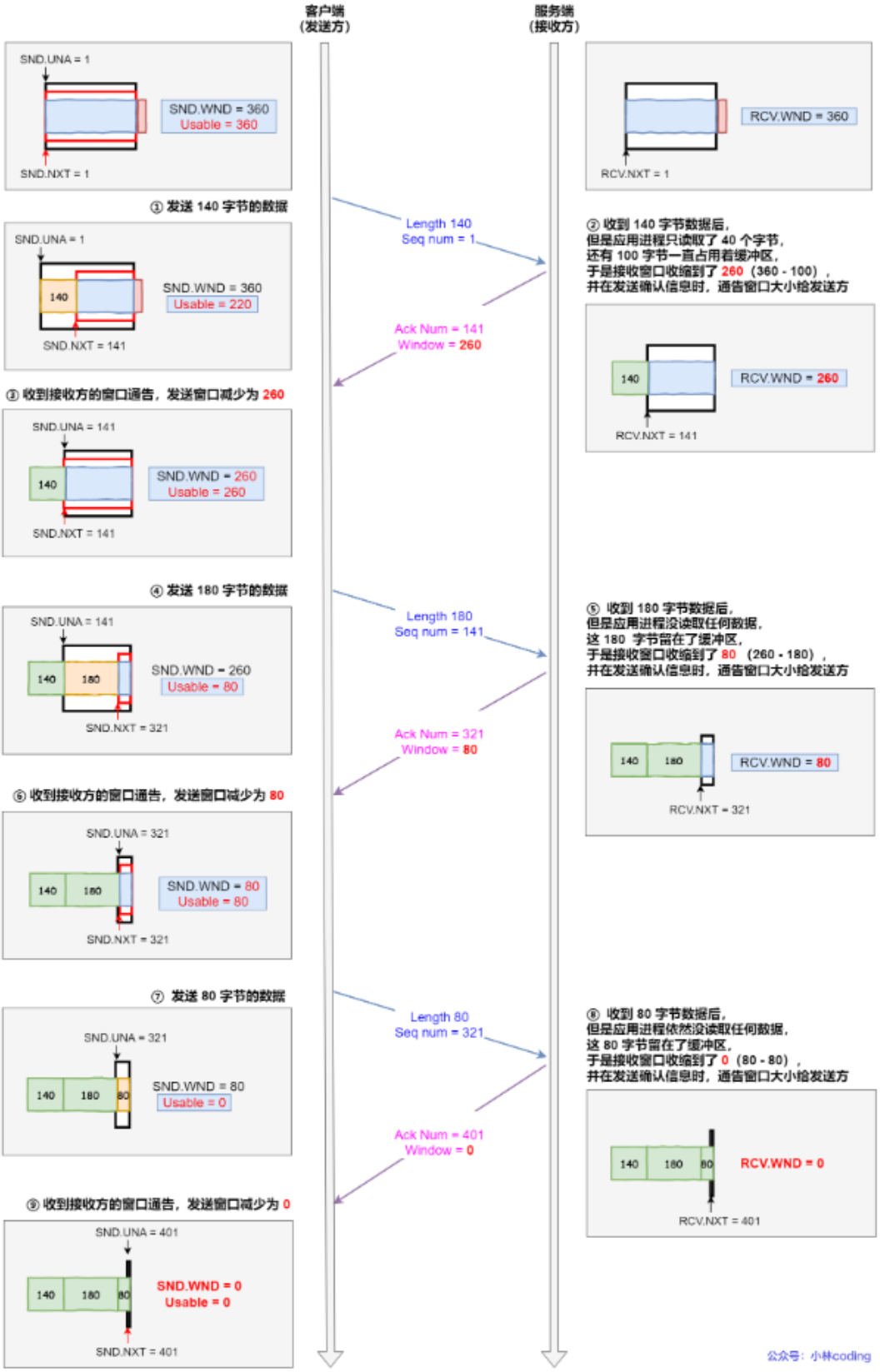
  因为滑动窗口并不是一成不变的。比如，当接收方的应用进程读取数据的速度非常快的话，这样的话接收窗口可以很快的就空缺出来。那么新的接收窗口大小，是通过 TCP 报文中的 Windows 字段来告诉发送方。那么这个传输过程是存在时延的，所以接收窗口和发送窗口是约等于的关系。

4.4.3 流量控制

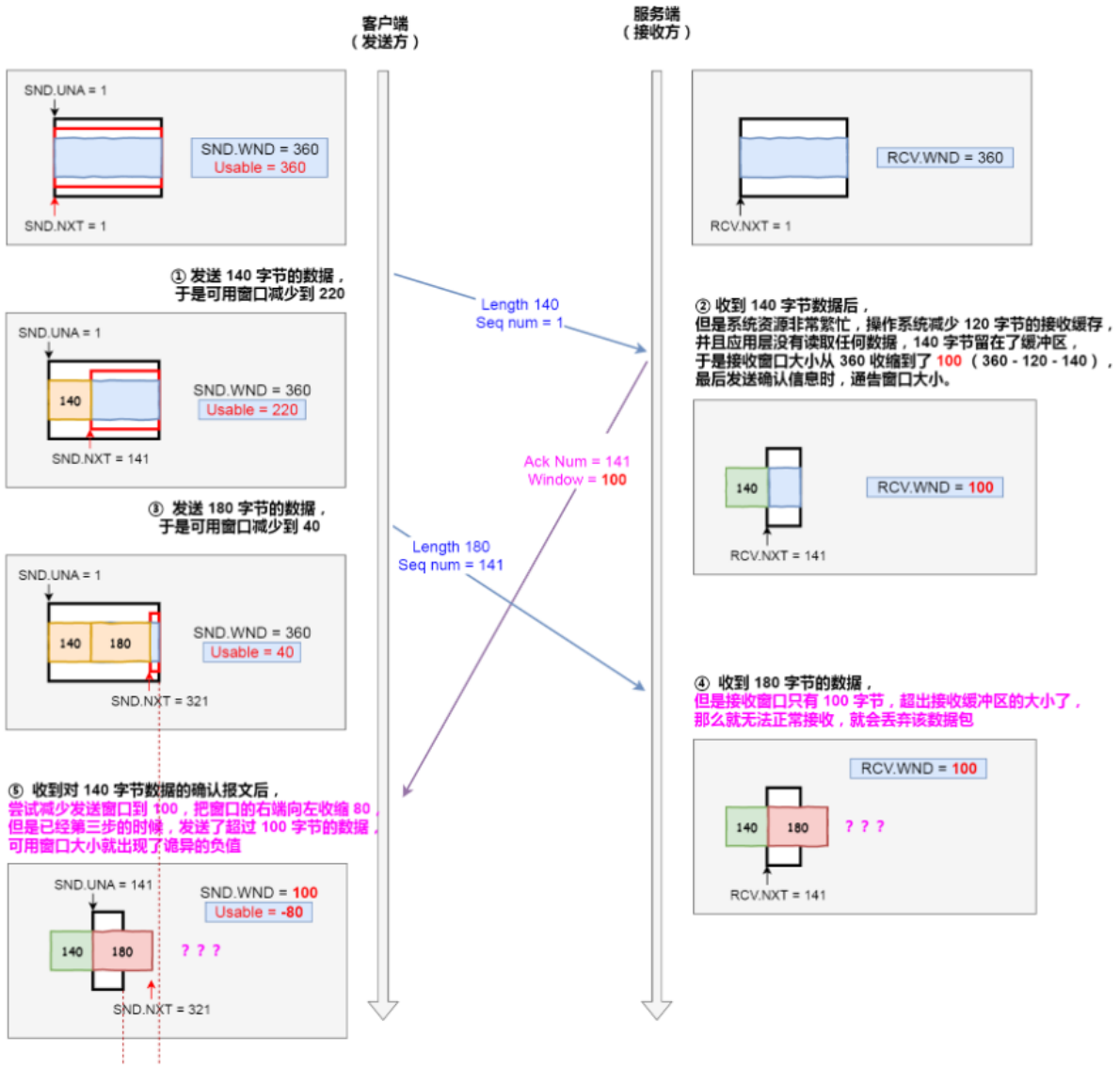
发送方不能无脑的发数据给接收方，要考虑接收方处理能力。

如果一直无脑的发数据给对方，但对方处理不过来，那么就会导致触发重发机制，从而导致网络流量的无端的浪费。

为了解决这种现象发生，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，这就是所谓的流量控制。



一个例子可见最后窗口都收缩为 0 了，也就是发生了窗口关闭。当发送方可用窗口变为 0 时，发送方实际上会定时发送窗口探测报文，以便知道接收方的窗口是否发生了改变。



当服务端系统资源非常紧张的时候，操作系统可能会直接减少了接收缓冲区大小，这时应用程序又无法及时读取缓存数据，那么这时候就有严重的事情发生了，会出现数据包丢失的现象。

为了防止这种情况发生，TCP 规定是不允许同时减少缓存又收缩窗口的，而是采用先收缩窗口，过段时间再减少缓存，这样就可以避免了丢包情况。

- 窗口关闭

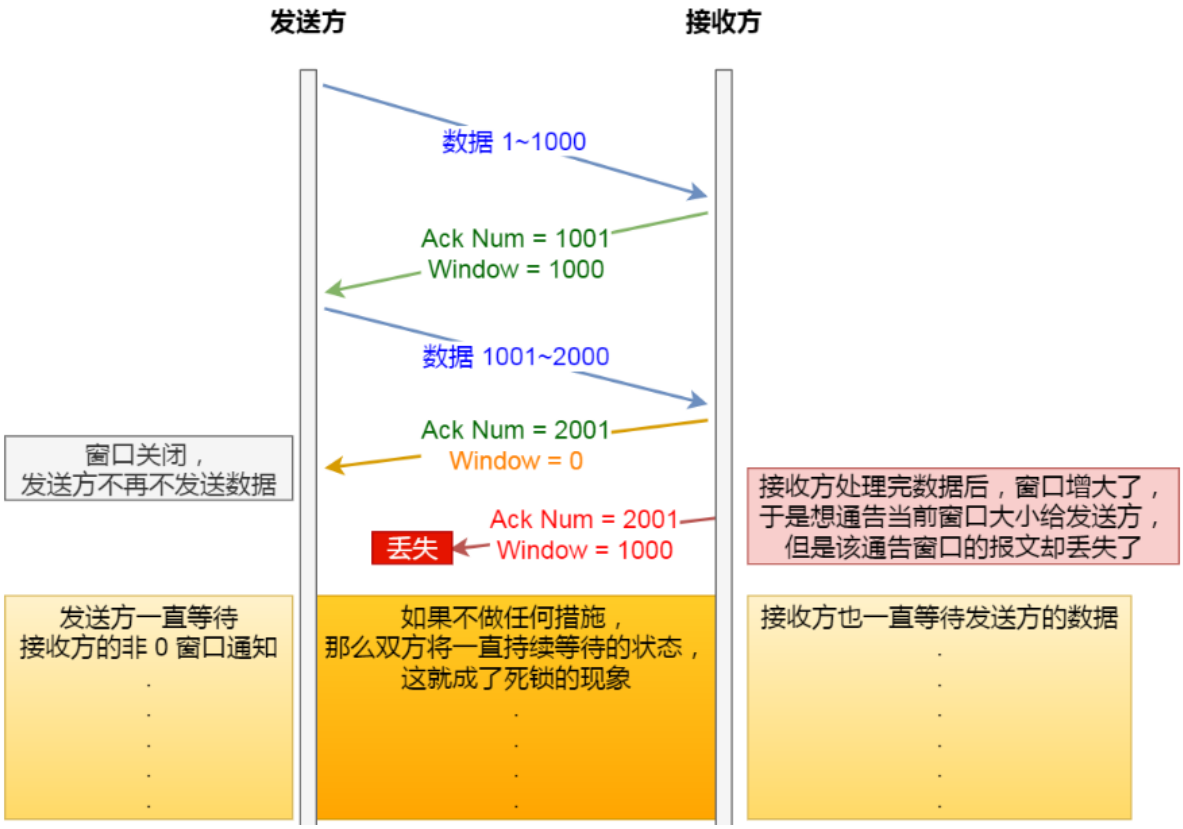
  TCP 通过让接收方指明希望从发送方接收的数据大小（窗口大小）来进行流量控制。

  如果窗口大小为 0 时，就会阻止发送方给接收方传递数据，直到窗口变为非 0 为止，这就是窗口关闭。

- 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。

  那么，当发生窗口关闭时，接收方处理完数据后，会向发送方通告一个窗口非 0 的 ACK 报文，如果这个通告窗口的 ACK 报文在网络中丢失了，那麻烦就大了。

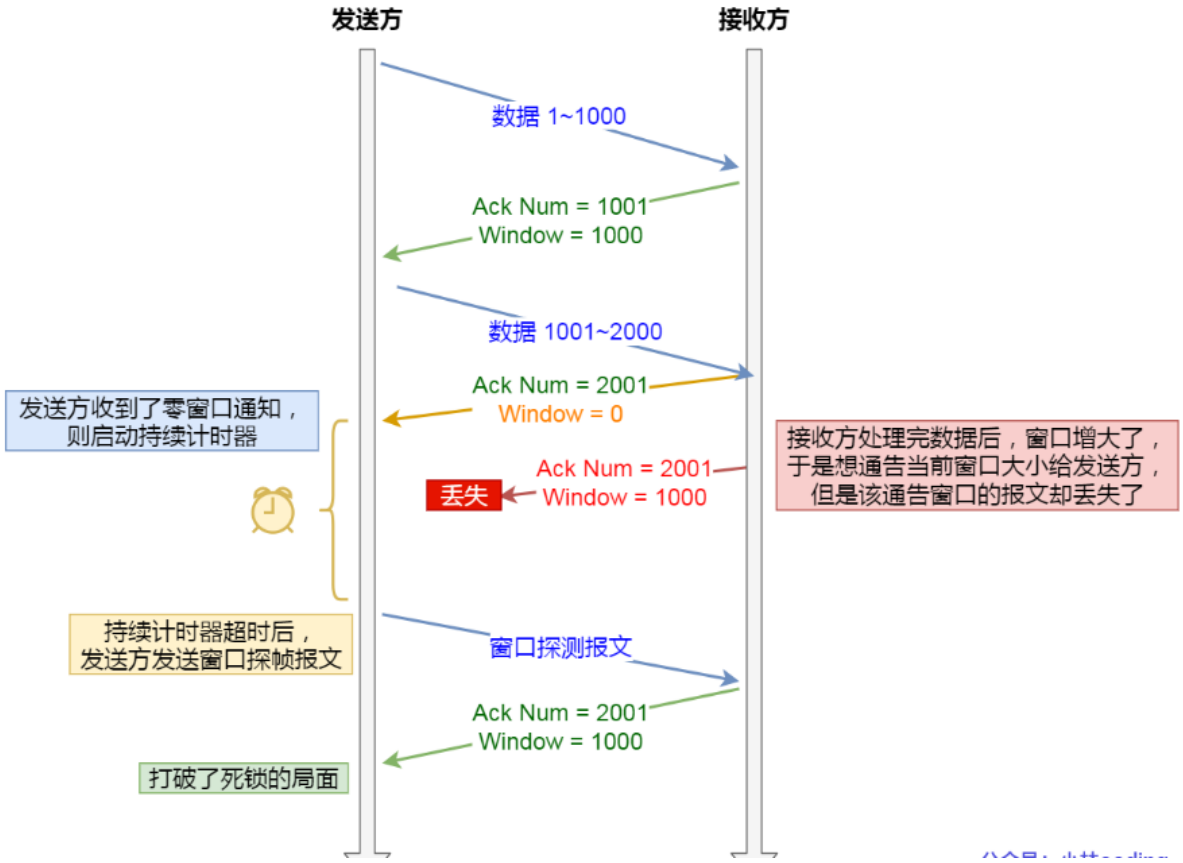


这会导致发送方一直等待接收方的非 0 窗口通知，接收方也一直等待发送方的数据，如不采取措施，这种相互等待的过程，会造成了死锁的现象。

- TCP 如何解决窗口关闭时，潜在的死锁现象呢?

  为了解决这个问题，TCP 为每个连接设有一个持续定时器，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 (Windowprobe) 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。



如果接收窗口仍然为 0，那么收到这个报文的一方就会重新启动持续计时器；

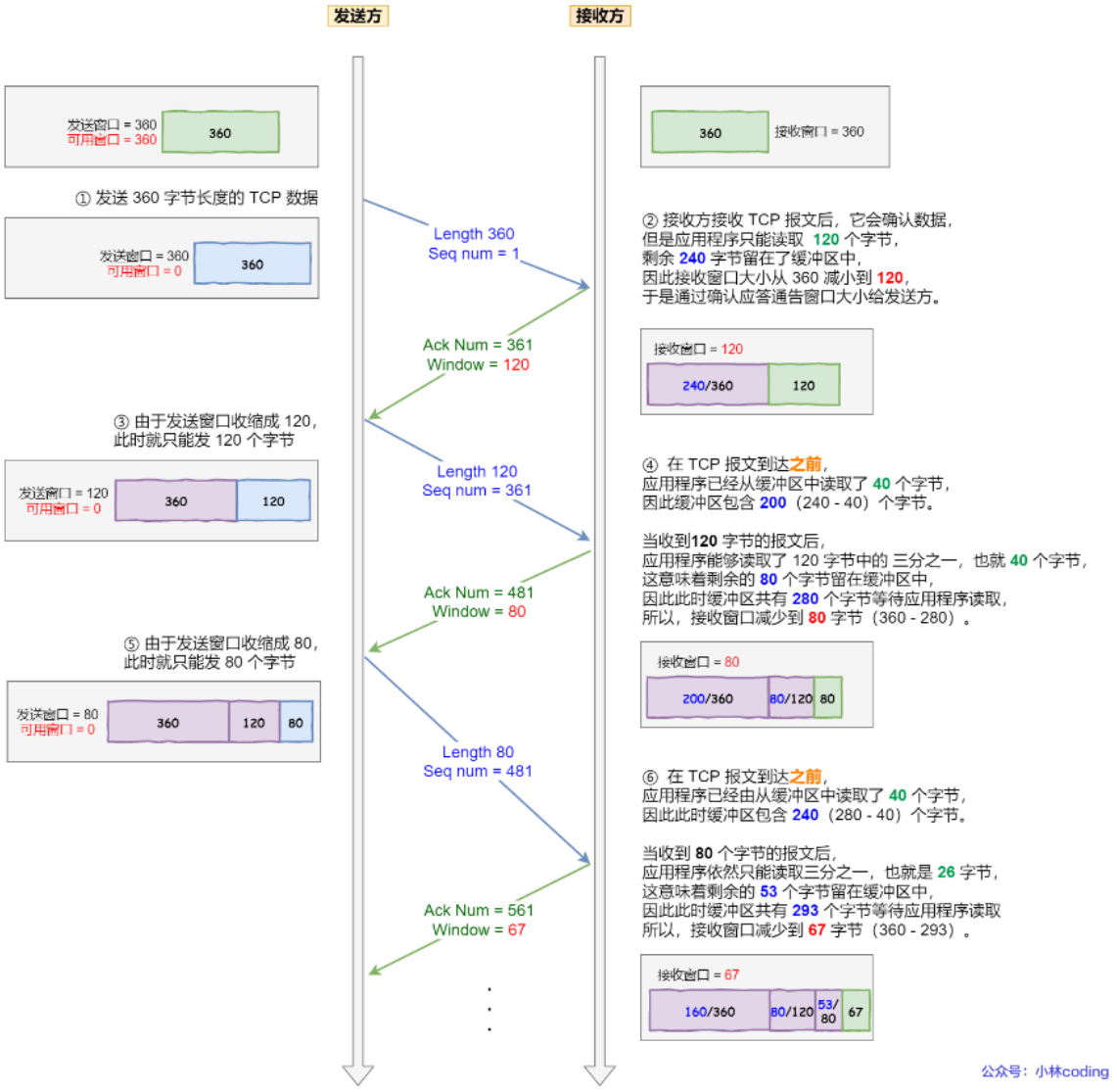
如果接收窗口不是 0，那么死锁的局面就可以被打破了。

窗口探测的次数一般为 3 次，每次大约 30-60 秒（不同的实现可能会不一样）。如果 3 次过后接收窗口还是 0 的话，有的 TCP 实现就会发 RST 报文来中断连接。

- 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地发送这几个字节，这就是糊涂窗口综合症。



糊涂窗口综合征例子

每个过程的窗口大小的变化，在图中都描述的很清楚了，可以发现窗口不断减少了，并且发送的数据都是比较小的了。

所以，糊涂窗口综合症的现象是可以发生在发送方和接收方：

- 接收方可以通告一个小的窗口
- 而发送方可以发送小数据

解决策略：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。

- 怎么让发送方避免发送小数据？发送方通常的策略如下：使用 Nagle 算法，该算法的思路是延时处理，只有满足下面两个条件中的任意一个条件，才可以发送数据：
  - 条件一：要等到窗口大小 >= MSS 并且数据大小 >= MSS；
  - 条件二：收到之前发送数据的 ack 回包；

只要上面两个条件都不满足，发送方一直在囤积数据，直到满足上面的发送条件。注意，如果接收方不能满足「不通告小窗口给发送方」，那么即使开了 Nagle 算法，也无法避免糊涂窗口综合症，因为如果对端

ACK 回复很快的话（达到 Nagle 算法的条件二），Nagle 算法就不会拼接太多的数据包，这种情况下依然会有小数据包的传输，网络总体的利用率依然很低。

所以，接收方得满足「不通告小窗口给发送方」+ 发送方开启 Nagle 算法，才能避免糊涂窗口综合症。

Nagle 算法默认是打开的，如果对于一些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，则需要关闭 Nagle 算法。

可以在 Socket 设置 $TCP_{NODELAY}$ 选项来关闭这个算法（关闭 Nagle 算法没有全局参数，需要根据每个应用自己的特点来关闭）

4.4.4 拥塞控制

- 为什么要有拥塞控制呀，不是有流量控制了吗？前面的流量控制是避免「发送方」的数据填满「接收方」的缓存，但是并不知道网络的中发生了什么。一般来说，计算机网络都处在一个共享的环境。因此也有可能会因为其他主机之间的通信使得网络拥堵。在网络出现拥堵时，如果继续发送大量数据包，可能会导致数据包时延、丢失等，这时 TCP 就会重传数据，但是一重传就会导致网络的负担更重，于是会导致更大的延迟以及更多的丢包，这个情况就会进入恶性循环被不断地放大.... 所以，TCP 不能忽略网络上发生的事，它被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，控制的目的就是避免「发送方」的数据填满整个网络。

  为了在「发送方」调节所要发送数据的量，定义了一个叫做「拥塞窗口」的概念。

- 什么是拥塞窗口？和发送窗口有什么关系呢？拥塞窗口 cwnd 是发送方维护的一个的状态变量，它会根据网络的拥塞程度动态变化的。

  我们在前面提到过发送窗口 swnd 和接收窗口 rwnd 是约等于的关系，那么由于加入了拥塞窗口的概念后，此时发送窗口的值是 swnd = min(cwnd, rwnd)，也就是拥塞窗口和接收窗口中的最小值。

  拥塞控制 cwnd 变化的规则：

  - 只要网络中没有出现拥塞，cwnd 就会增大；
  - 但网络中出现了拥塞，cwnd 就减少；

- 怎么知道当前网络中是否出现了拥塞呢？其实只要「发送方」没有在规定时间内接收到 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。

- 拥塞控制有哪些控制算法?

  - 慢启动
  - 拥塞避免
  - 拥塞发生
  - 快速恢复

1. 慢启动

   慢启动的算法记住一个规则就行：当发送方每收到一个 ACK，拥塞窗口 cwnd 的大小就会加一倍。

   这里假定拥塞窗口 cwnd 和发送窗口 swnd 相等，下面举个栗子：

   连接建立完成后，一开始初始化 cwnd = 1，表示可以传一个 MSS 大小的数据。

   当收到一个 ACK 确认应答后，cwnd 增加 1，于是一次能够发送 2 个。

   当收到 2 个的 ACK 确认应答后，cwnd 增加 2，于是就可以比之前多发 2 个，所以这一次能够发送 4 个。

当这 4 个的 ACK 确认到来的时候，每个确认 cwnd 增加 1，4 个确认 cwnd 增加 4，于是就可以比之前多发 4 个，所以这一次能够发送 8 个。

可以看出慢启动算法发包的个数是指数性的增长。

有一个叫慢启动门限 ssthresh （slow start threshold）状态变量。

当 cwnd < ssthresh 时，使用慢启动算法。

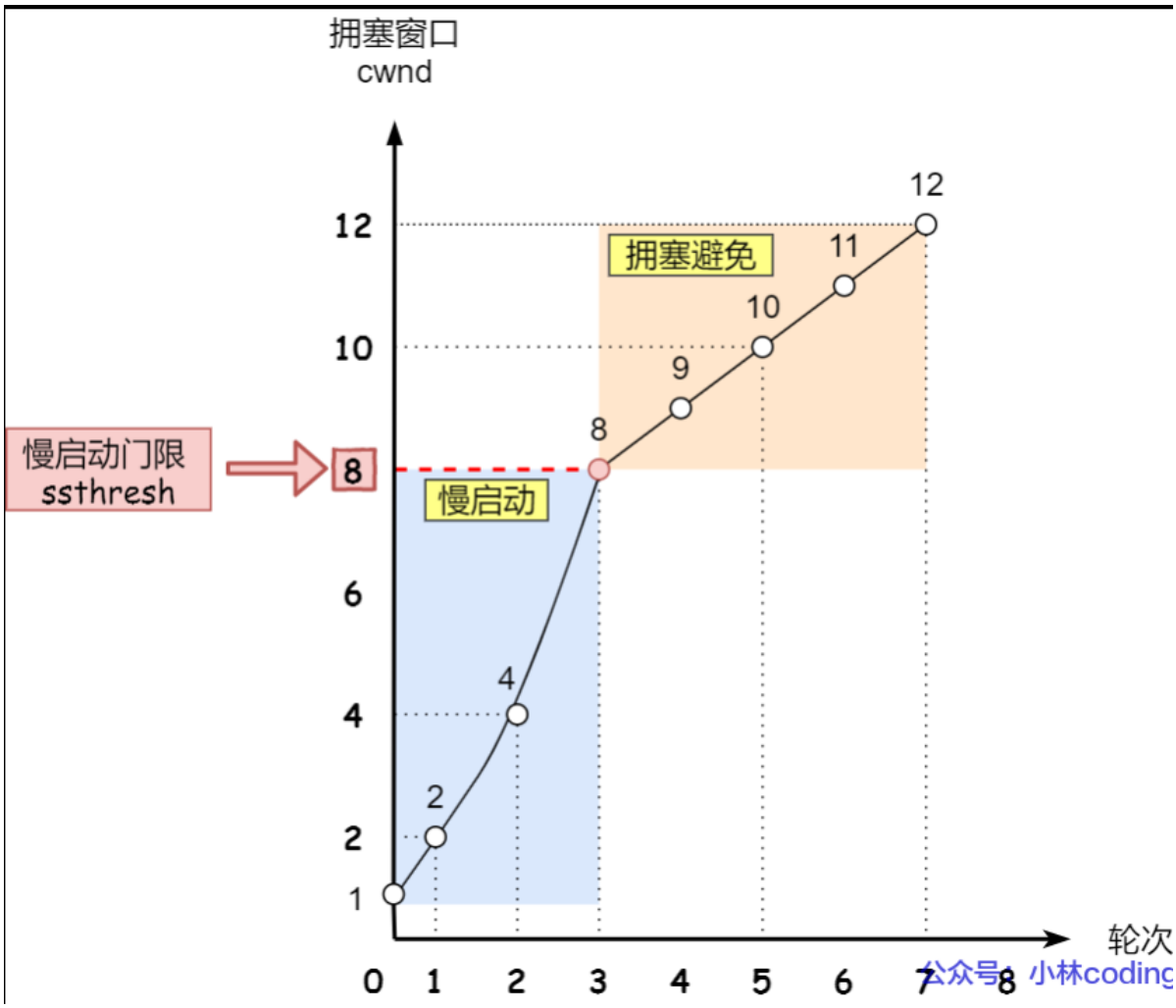
当 cwnd >= ssthresh 时，就会使用「拥塞避免算法」。

2. 拥塞避免算法

当拥塞窗口 cwnd 「超过」慢启动门限 ssthresh 就会进入拥塞避免算法。

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：每当收到一个 ACK 时，cwnd 增加 1/cwnd。



- 拥塞发生

  当网络出现拥塞，也就是会发生数据包重传，重传机制主要有两种：

  - 超时重传
  - 快速重传

- 发生超时重传的拥塞发生算法

当发生了「超时重传」，则就会使用拥塞发生算法。

这个时候，ssthresh 和 cwnd 的值会发生变化：

- ssthresh 设为 cwnd/2，
- cwnd 重置为 1 （是恢复为 cwnd 初始化值，我这里假定 cwnd 初始化值 1）
- 发生快速重传的拥塞发送算法

  当接收方发现丢了一个中间包的时候，发送三次前一个包的 ACK，于是发送端就会快速地重传，不必等待超时再重传。

  TCP 认为这种情况不严重，因为大部分没丢，只丢了一小部分，则 ssthresh 和 cwnd 变化如下：

  - cwnd = cwnd/2 ，也就是设置为原来的一半;
  - ssthresh = cwnd;
  - 进入快速恢复算法
- 快速恢复

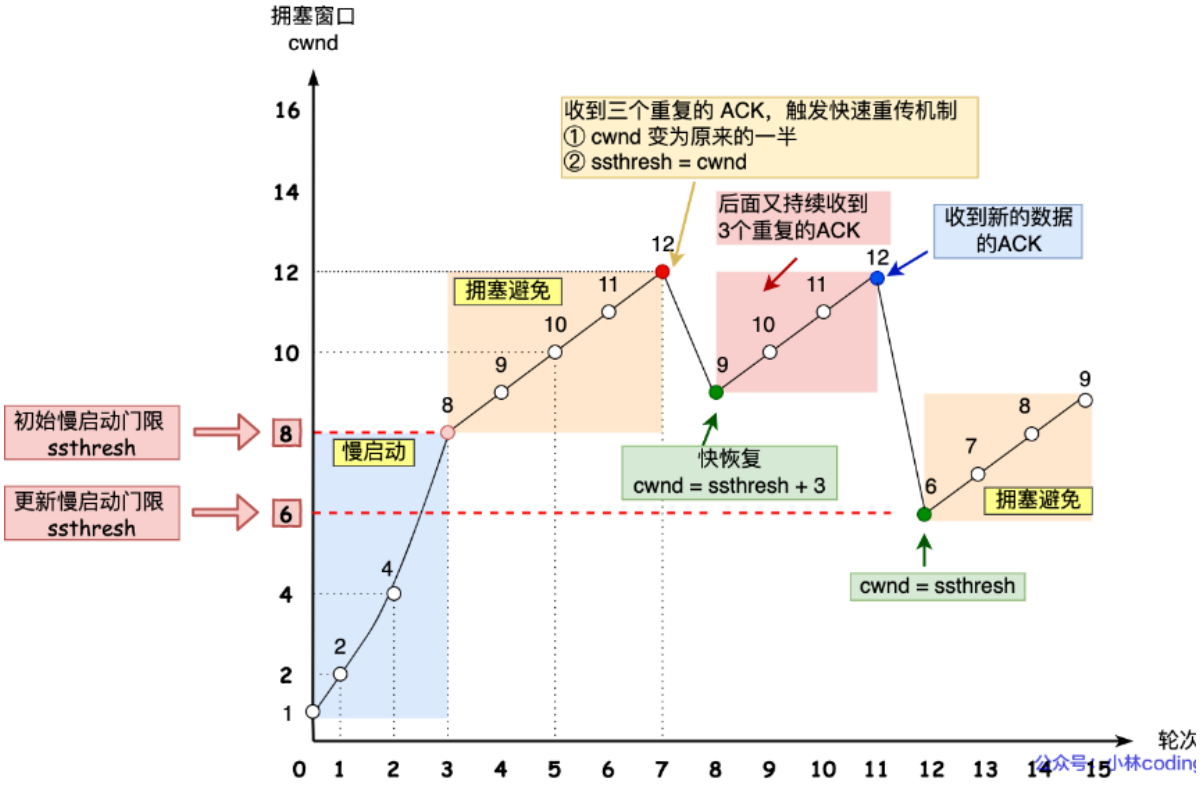
  快速重传和快速恢复算法一般同时使用，快速恢复算法是认为，你还能收到 3 个重复 ACK 说明网络也不那么糟糕，所以没有必要像 RTO 超时那么强烈。

  正如前面所说，进入快速恢复之前，cwnd 和 ssthresh 已被更新了：

  - cwnd = cwnd/2 ，也就是设置为原来的一半;
  - ssthresh = cwnd;

  然后进入快速恢复算法如下：

  - 拥塞窗口 cwnd = ssthresh + 3 （3 的意思是确认有 3 个数据包被收到了）;
  - 重传丢失的数据包;
  - 如果再收到重复的 ACK，那么 cwnd 增加 1;
  - 如果收到新数据的 ACK 后，把 cwnd 设置为第一步中的 ssthresh 的值，原因是该 ACK 确认了新的数据，说明从 duplicated ACK 时的数据都已收到，该恢复过程已经结束，可以回到恢复之前的状态了，也即再次进入拥塞避免状态;



也就是没有像「超时重传」一夜回到解放前，而是还在比较高的值，后续呈线性增长。

- 快速恢复算法过程中，为什么收到新的数据后，cwnd 设置回了 ssthresh？

  首先，快速恢复是拥塞发生后慢启动的优化，其首要目的仍然是降低 cwnd 来减缓拥塞，所以必然会出现 cwnd 从大到小的改变。

  其次，过程 2（cwnd 逐渐加 1）的存在是为了尽快将丢失的数据包发给目标，从而解决拥塞的根本问题（三次相同的 ACK 导致的快速重传），所以这一过程中 cwnd 反而是逐渐增大的。

4.5 TCP 半连接和全连接队列

在 TCP 三次握手的时候，Linux 内核会维护两个队列，分别是：

- 半连接队列，也称 SYN 队列；
- 全连接队列，也称 accept 队列；

服务端收到客户端发起的 SYN 请求后，内核会把该连接存储到半连接队列，并向客户端响应 SYN+ACK，接着客户端会返回 ACK，服务端收到第三次握手的 ACK 后，内核会把连接从半连接队列移除，然后创建新的完全的连接，并将其添加到 accept 队列，等待进程调用 accept 函数时把连接取出来。

Recv-Q：当前全连接队列的大小，也就是当前已完成三次握手并等待服务端 accept() 的 TCP 连接；

Send-Q：当前全连接最大队列长度，上面的输出结果说明监听 8088 端口的 TCP 服务，最大全连接长度为 128；