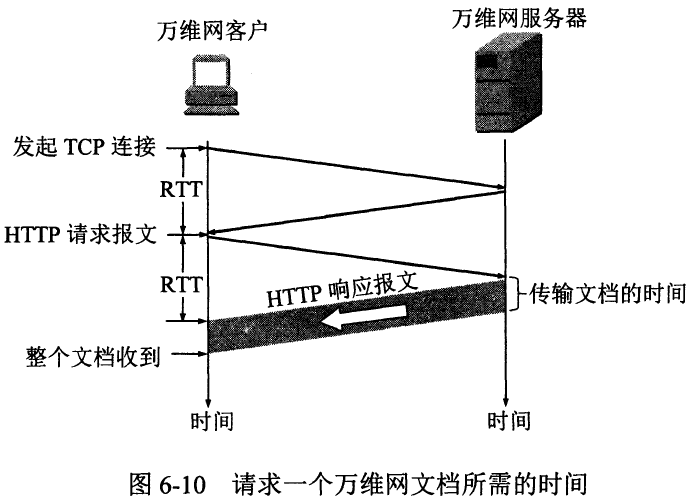
# HTTP的操作过程

HTTP协议本身是无连接的，虽然使用了TCP连接，但在通信双方交换HTTP报文之前不需要先建立HTTP连接。

HTTP协议是无状态的，是指协议对于交互性场景没有记忆能力。每次http请求都会返回同样的信息，因为这个是没有交互的，每一次的请求都是相互独立的。

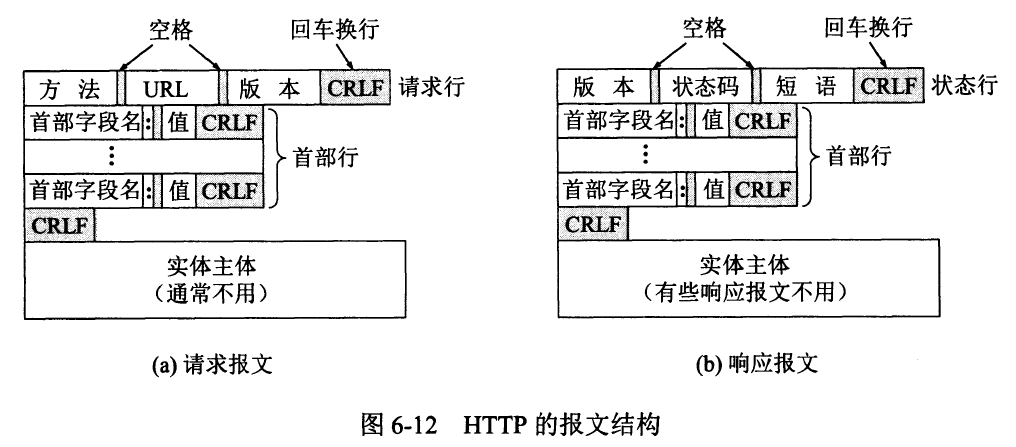
当建立TCP连接的三报文握手的前两部分完成后，即经过了一个RTT时间后，Web客户端就把HTTP请求报文作为TCP连接第三个报文的数据发送给Web服务器，服务器收到HTTP请求报完之后，就把所请求的文档作为响应报文返回给客户端。



从图6-10可以看出，请求一个web文档所需的时间是该文档的传输时间加上两倍往返时间RTT（一个RTT用于连接TCP连接，另一个RTT用于请求和接收Web文档。）。

# HTTP 的报文结构

HTTP 是 面向文本的(text-oriented)，因此在报文中的每一个字段都是一些 ASCII 码串，各个字段的长度都是不确定的。



## 报文组成

HTTP 请求报文和响应报文都是由三个部分组成的。可以看出，这两种报文格式的区别就是开始行不同。

### 开始行

* 在请求报文中的开始行叫做请求行(Request-Line)
* 在响应报文中的开始行叫做 状态行(Status-Line)
* 请求报文的第一行 “请求行” 只有三个内容：方法、请求资源的 URL 以及 HTTP 的版本。
* 响应报文的第一行 “状态行” 包括三项内容：HTTP 版本，状态码以及解释状态码的简单短语。

### 首部行

* 用来说明浏览器、服务器或报文主体的一些信息。首部可有好几行，但也可以不使用。在每一个首部行中都有首部字段名和它的值，每行在结束的地方都要有 “回车” 和 “换行”。整个首部行结束时，还有一空行将首部行和后面的实体主体分开。

### 实体主体(entity body)

* 在请求报文中一般都不用这个字段，而在响应报文中也可能没有这个字段。

## 响应状态码

1xx 表示通知信息，如请求收到了或正在进行处理。

2xx 表示成功，如接受或知道了。

3xx 表示重定向，如要完成请求还必须采取进一步的行动。

4xx 表示客户的差错，如请求中有错误的语法或不能完成。

5xx 表示服务器的差错，如服务器失效无法完成请求。

常见的状态码和相关的短语

200 OK：请求成功，信息在返回的响应报文中。

301 Moved Permanently：请求的对象已经被永久转移了，新的 URL 定义在响应报文的 Location：首部行中。客户软件将自动获取新的 URL。

400 Bad Request：一个通用差错代码，指示该请求不能被服务器理解。

404 Not Found：被请求的文档不在服务器上。

505 HTTP Version Not Supported：服务器不支持请求报文使用的 HTTP 协议版本。

# HTTP/1.0特性

## 优点

* 简单，易于理解
* 灵活，易于扩展

## 缺点

* 无状态
* 不安全，明文传输
* 传输效率低，每次请求响应都要建立TCP连接

# HTTP/1.1

## 新特性

* 默认开启长连接：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。持久连接的特点是，只要任意一端没有明确提出断开连接，则保持 TCP 连接状态。
* 流水线发送请求：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。
* 支持断点续传

## 缺点

* 请求 / 响应头部（Header）未经压缩就发送，首部信息越多延迟越大
* 队头阻塞：「请求 - 应答」的模式加剧了 HTTP 的性能问题。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」
* 没有请求优先级控制，导致关键请求被阻塞；
* 请求只能从客户端开始，服务器只能被动响应。

## 性能优化思路

1. 使用缓存避免发送重复HTTP请求

* 客户端会把第一次请求以及响应的数据保存在本地磁盘上，其中将请求的 URL 作为 key，而响应作为 value，两者形成映射关系。这样当后续发起相同的请求时，就可以先在本地磁盘上通过 key 查到对应的 value。同时服务器在发送 HTTP 响应时，会估算一个过期的时间，并把这个信息放到响应头部中，这样客户端在查看响应头部的信息时，一旦发现缓存的响应是过期的，则就会重新发送网络请求。
* 如果客户端从第一次请求得到的响应头部中发现该响应过期了，客户端重新发送请求，同时在请求的 Etag 头部带上第一次请求的响应头部中的摘要，来判断缓存中的资源和服务器中的资源是否一致。这个摘要是唯一标识响应的资源，当服务器收到请求后，会将本地资源的摘要与请求中的摘要做个比较。
* 如果不同，那么说明客户端的缓存已经没有价值，服务器在响应中带上最新的资源。
* 如果相同，说明客户端的缓存还是可以继续使用的，那么服务器仅返回不含有包体的 304 Not Modified 响应，告诉客户端仍然有效，这样就可以减少响应资源在网络中传输的延时

1. 减少重定向请求次数

* 如果重定向请求越多，那么客户端就要多次发起 HTTP 请求，每一次的 HTTP 请求都得经过网络，这无疑会越降低网络性能。

1. 合并请求

* 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。
* 比如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，还有服务端使用 webpack 等打包工具将 js、css 等资源合并打包成大文件，还可以将图片的二进制数据用 base64 编码后，以 URL 的形式潜入到 HTML 文件，跟随 HTML 文件一并发送等
* 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。

1. 延迟发送请求

* 当前不需要的资源，我们没必要也获取过来，于是可以通过「按需获取」的方式，来减少第一时间的 HTTP 请求次数。
* 请求网页的时候，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。

1. 压缩响应内容

* 通过压缩响应资源，降低传输资源的大小，从而提高传输效率，所以应当选择更优秀的压缩算法

# HTTP 与 HTTPS

## 区别

* 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。
* 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。
* HTTP 的端口号是 80，HTTPS 的端口号是 443。

## HTTPS解决的问题

### 明文传输

* 使用混合加密的方式实现信息的机密性，解决了窃听的风险。
* 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。

### 信息的完整性

* 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字符串，解决了篡改的风险。
* 客户端在发送明文之前会通过摘要算法算出明文的「指纹」，发送的时候把「指纹 + 明文」一同使用「会话密钥」加密成密文后，发送给服务器，服务器解密后，用相同的摘要算法算出发送过来的明文，通过比较客户端携带的「指纹」和当前算出的「指纹」做比较，若「指纹」相同，说明数据是完整的。

### 身份验证

* 将服务器公钥放入到数字证书中，解决了冒充的风险。
* 需要借助第三方权威机构 CA （数字证书认证机构），将服务器公钥放在数字证书（由数字证书认证机构颁发）中，只要证书是可信的，公钥就是可信的。



## HTTPS连接建立

### 先通过TCP建立三次握手，再通过SSL/TLS 的四次握手建立通信

### TLS第一次握手

ClientHello

• 首先，由客户端向服务器发起加密通信请求，也就是 ClientHello 请求。

• 在这一步，客户端主要向服务器发送以下信息：

• （1）客户端支持的 SSL/TLS 协议版本，如 TLS 1.2 版本。

• （2）客户端生产的随机数（Client Random），后面用于生产「会话秘钥」。

• （3）客户端支持的密码套件列表，如 RSA 加密算法。

### TLS第二次握手

SeverHello

• 服务器收到客户端请求后，向客户端发出响应，也就是 SeverHello。服务器回应的内容有如下内容：

• （1）确认 SSL/ TLS 协议版本，如果浏览器不支持，则关闭加密通信。

• （2）服务器生产的随机数（Server Random），后面用于生产「会话秘钥」。

• （3）确认的密码套件列表，如 RSA 加密算法。

Server Certificate

• （4）服务器的数字证书。

Server Hello Done

• 目的是告诉客户端，我已经把该给你的东西都给你了，本次打招呼完毕。

### 客户端验证证书有效性

客户端收到服务器的回应之后，首先通过浏览器或者操作系统中的 CA 公钥，确认服务器的数字证书的真实性。如果证书没有问题，客户端会从数字证书中取出服务器的公钥，然后使用它加密报文，向服务器发送其他信息。

### TLS第三次握手

Cipher Key Exchange

• 客户端就会生成一个新的随机数 (pre-master)，用服务器的 RSA 公钥加密该随机数，通过「Change Cipher Key Exchange」消息传给服务端。

• 服务端收到后，用 RSA 私钥解密，得到客户端发来的随机数 (pre-master)。

• 至此，客户端和服务端双方都共享了三个随机数，分别是 Client Random、Server Random、pre-master。

• 于是，双方根据已经得到的三个随机数，生成会话密钥（Master Secret），它是对称密钥，用于对后续的 HTTP 请求/响应的数据加解密。

Change Cipher Spec

• 生成完会话密钥后，然后客户端发一个「Change Cipher Spec」，表示随后的信息都将用「会话秘钥」加密通信。

Encrypted Handshake Message（Finishd）

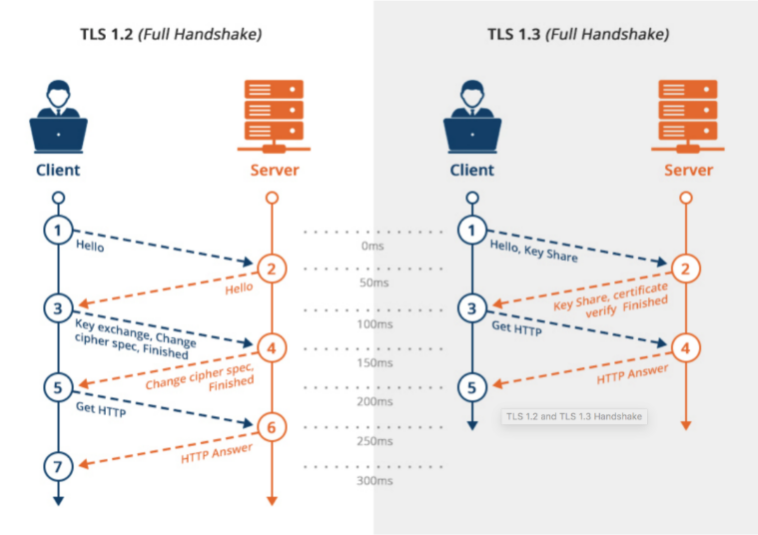
• 然后，客户端再发一个「Encrypted Handshake Message（Finishd）」消息，把之前所有发送的数据做个摘要，再用会话密钥（master secret）加密一下，让服务器做个验证，验证加密通信是否可用和之前握手信息是否有被中途篡改过。

### TLS第四次握手

服务器也是同样的操作，发「Change Cipher Spec」和「Encrypted Handshake Message」消息，如果双方都验证加密和解密没问题，那么握手正式完成。

## TLS 1.2 与 TLS 1.3

在 TLS 1.2 的握手中，一般是需要 4 次握手，下图的左边部分就是 TLS 1.2 的握手过程。上图的右边部分就是 TLS 1.3 的握手过程，可以发现 TLS 1.3 把 Hello 和公钥交换这两个消息合并成了一个消息，于是这样就减少到只需 1 RTT 就能完成 TLS 握手。



TLS 1.3 怎么合并的呢？具体的做法是，客户端在 Client Hello 消息里带上了支持的椭圆曲线，以及这些椭圆曲线对应的公钥。服务端收到后，选定一个椭圆曲线等参数，然后返回消息时，带上服务端这边的公钥。经过这 1 个 RTT，双方手上已经有生成会话密钥的材料了，于是客户端计算出会话密钥，就可以进行应用数据的加密传输了。

# HTTP/2、HTTP/3 演变

## 针对HTTP/1.1 的性能瓶颈，HTTP/2 的优化

### 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

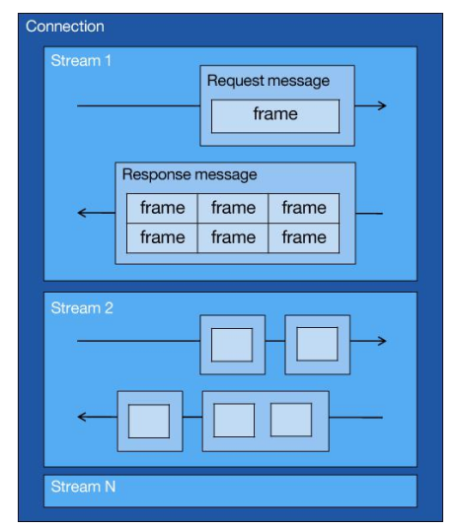
* 因为计算机只懂二进制，那么收到报文后，无需再将明文的报文转成二进制，而是直接解析二进制报文，这增加了数据传输的效率。

### 强化安全

* HTTP/2跑在HTTPS之上 TLS 1. 2

### 多路复用，并发传输

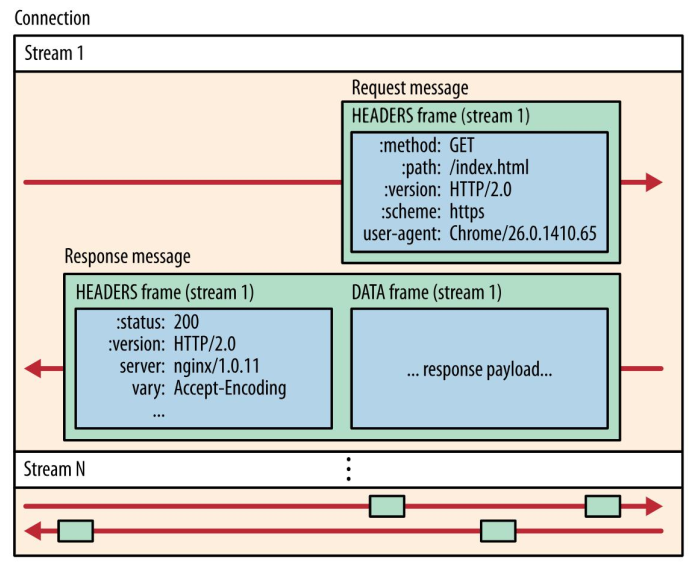
* HTTP/1.1 同一个连接中，HTTP 完成一个请求与响应，才能处理下一个事务，也就是说在发出请求等待响应的过程中，是没办法做其他事情的，如果响应迟迟不来，那么后续的请求是无法发送的，也造成了队头阻塞的问题。
* HTTP/2 通过 Stream 这个设计，多个 Stream 复用一条 TCP 连接，达到并发的效果，解决了 HTTP/1.1 队头阻塞的问题，提高了 HTTP 传输的吞吐量。



• Message 对应 HTTP/1 中的请求或响应，由 HTTP 头部和包体构成；

• Frame 是 HTTP/2 最小单位，以二进制压缩格式存放 HTTP/1 中的内容（头部和包体）；

* 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。



* 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。
* HTTP/2 还可以对每个 Stream 设置不同优先级，帧头中的「标志位」可以设置优先级，比如客户端访问 HTML/CSS 和图片资源时，希望服务器先传递 HTML/CSS，再传图片，那么就可以通过设置 Stream 的优先级来实现，以此提高用户体验。

### 服务端推送

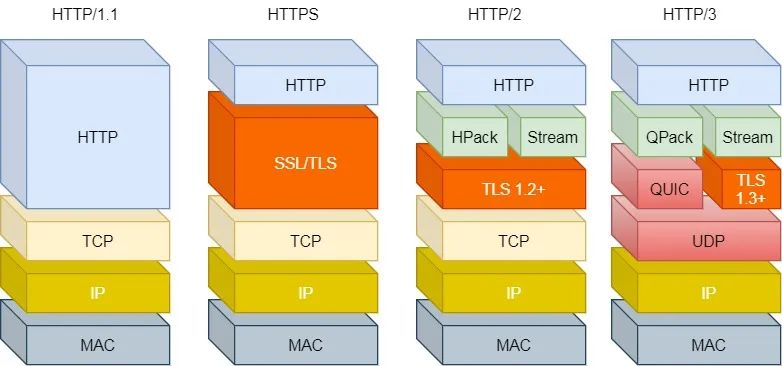
* 服务不再是被动地响应，也可以主动向客户端发送消息。
* 举例来说，在浏览器刚请求 HTML 的时候，就提前把可能会用到的 JS、CSS 文件等静态资源主动发给客户端，减少延时的等待，也就是服务器推送（Server Push，也叫 Cache Push）。

## HTTP/2 有哪些缺陷？HTTP/3 做了哪些优化？

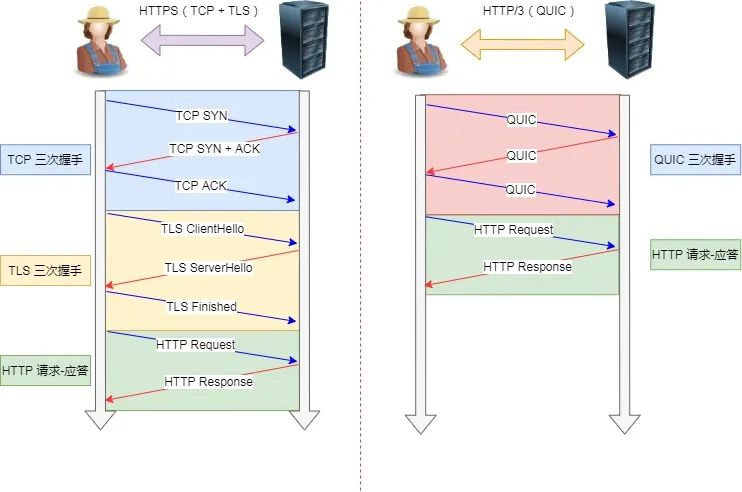
HTTP/2 主要的问题在于：多个 HTTP 请求在复用一个 TCP 连接，下层的 TCP 协议是不知道有多少个 HTTP 请求的。

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。

HTTP/3 把 HTTP 下层的 TCP 协议改成了 UDP！基于 UDP 的 QUIC 协议 可以实现类似 TCP 的可靠性传输。



HTTPS 要建立一个连接，要花费 6 次交互，先是建立三次握手，然后是 TLS/1.3 的三次握手。QUIC 直接把以往的 TCP 和 TLS/1.3 的 6 次交互合并成了 3 次，减少了交互次数。



# TCP基本认识

## 什么是TCP？

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；

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；

消息是「没有边界」的，所以无论我们消息有多大都可以进行传输。并且消息是「有序的」，当「前一个」消息没有收到的时候，即使它先收到了后面的字节已经收到，那么也不能扔给应用层去处理，同时对「重复」的报文会自动丢弃。

## 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。

建立一个 TCP 连接是需要客户端与服务器端达成上述三个信息的共识。

* 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

## UDP 和 TCP 有什么区别呢？分别的应用场景是？

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、按需到达。
* UDP 是尽最大努力交付，不保证可靠交付数据。

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。

由于 TCP 是面向连接，能保证数据的可靠性交付，因此经常用于：

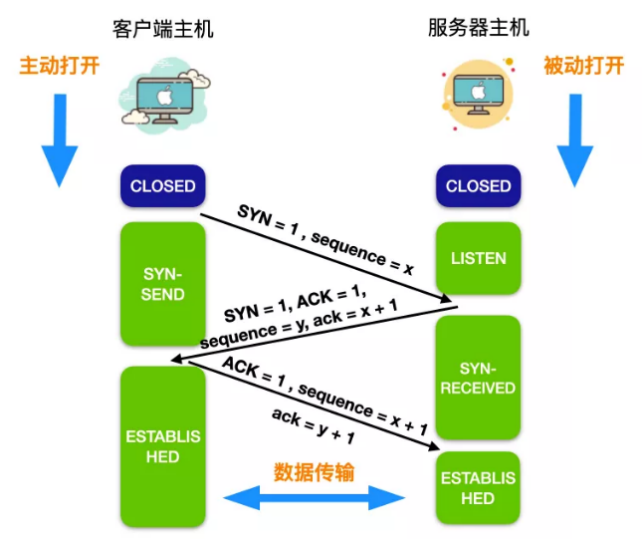
* FTP 文件传输
* HTTP / HTTPS

由于 UDP 面向无连接，它可以随时发送数据，再加上UDP本身的处理既简单又高效，因此经常用于：

* DNS 、SNMP 等
* 视、音频等媒体通信
* 广播通信

# TCP连接建立

## TCP 三次握手过程和状态变迁



三次握手的的目的不仅仅在于让通信双方知晓正在建立一个连接，也在于利用数据包中的选项字段来交换一些特殊信息，交换初始序列号。

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

一旦完成三次握手，双方都处于 ESTABLISHED 状态，此致连接就已建立完成，客户端和服务端就可以相互发送数据了。

## Linux 系统中查看 TCP 状态？

可以通过 netstat -napt 命令查看



## 为什么是三次握手？不是两次、四次？

### 为了防止旧的重复连接初始化造成混乱

* 客户端连续发送多次 SYN 建立连接的报文，在网络拥堵等情况下：
* 一个「旧 SYN 报文 seqnum = 90」比「最新的 SYN seqnum = 100 」 报文早到达了服务端；
* 那么此时服务端就会回一个 SYN + ACK 报文给客户端；
* 客户端收到后可以根据自身的上下文，判断这是一个历史连接（序列号过期或超时），那么客户端就会发送 RST 报文给服务端，表示中止这一次连接。
* 如果是两次握手连接，就不能判断当前连接是否是历史连接，三次握手则可以在客户端（发送方）准备发送第三次报文时，客户端因有足够的上下文来判断当前连接是否是历史连接：
* 如果是历史连接（序列号过期或超时），则第三次握手发送的报文是 RST 报文，以此中止历史连接；
* 如果不是历史连接，则第三次发送的报文是 ACK 报文，通信双方就会成功建立连接；

### 同步双方初始序列号

* TCP 协议的通信双方， 都必须维护一个「序列号」， 序列号是可靠传输的一个关键因素，它的作用：
* 接收方可以去除重复的数据；
* 接收方可以根据数据包的序列号按序接收；
* 可以标识发送出去的数据包中， 哪些是已经被对方收到的；
* 当客户端发送携带「初始序列号」的 SYN 报文的时候，需要服务端回一个 ACK 应答报文，表示客户端的 SYN 报文已被服务端成功接收，那当服务端发送「初始序列号」给客户端的时候，依然也要得到客户端的应答回应，这样一来一回，才能确保双方的初始序列号能被可靠的同步。
* 四次握手其实也能够可靠的同步双方的初始化序号，但由于第二步和第三步可以优化成一步，所以就成了「三次握手」。
* 而两次握手只保证了一方的初始序列号能被对方成功接收，没办法保证双方的初始序列号都能被确认接收。

### 避免资源浪费

* 如果只有「两次握手」，当客户端的 SYN 请求连接在网络中阻塞，客户端没有接收到 ACK 报文，就会重新发送 SYN ，由于没有第三次握手，服务器不清楚客户端是否收到了自己发送的建立连接的 ACK 确认信号，所以每收到一个 SYN 就只能先主动建立一个连接
* 那么服务器在收到请求后就会建立多个冗余的无效链接，造成不必要的资源浪费。
* 即两次握手会造成消息滞留情况下，服务器重复接受无用的连接请求 SYN 报文，而造成重复分配资源。

## 初始序列号 ISN 是如何随机产生的？

起始 ISN 是基于时钟的，每 4 毫秒 + 1，转一圈要 4.55 个小时。

ISN = M + F (localhost, localport, remotehost, remoteport)

M 是一个计时器，这个计时器每隔 4 毫秒加 1。

F 是一个 Hash 算法，根据源 IP、目的 IP、源端口、目的端口生成一个随机数值。要保证 Hash 算法不能被外部轻易推算得出，用 MD5 算法是一个比较好的选择。

## IP 层会分片，为什么 TCP 层还需要 MSS 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，在交给上一层 TCP 传输层。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。

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。

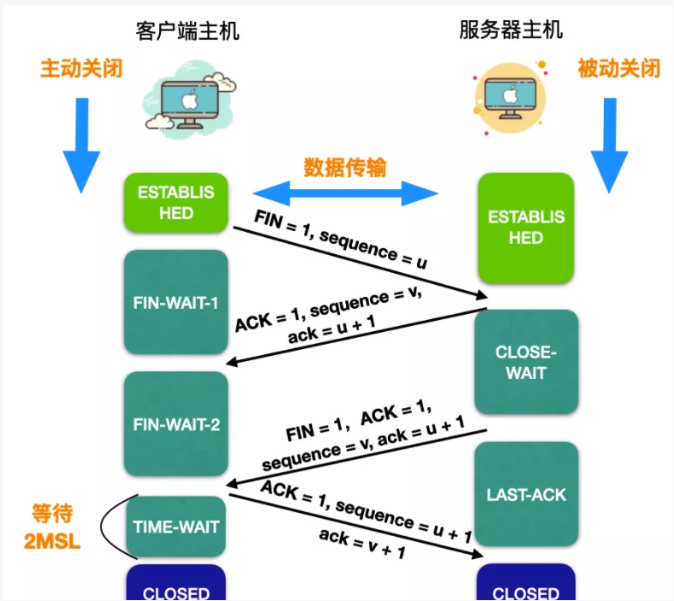
经过 TCP 层分片后，如果一个 TCP 分片丢失后，进行重发时也是以 MSS 为单位，而不用重传所有的分片，大大增加了重传的效率。

## 什么是 SYN 攻击？

TCP 连接建立是需要三次握手，假设攻击者短时间伪造不同 IP 地址的 SYN 报文，服务端每接收到一个 SYN 报文，就进入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，久而久之就会占满服务端的 SYN 接收队列（未连接队列），使得服务器不能为正常用户服务。

# TCP连接断开

## TCP 四次挥手



## 为什么挥手要四次？

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。

服务器收到客户端的 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，从而比三次握手导致多了一次。

## 为什么 TIME\_WAIT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。

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。

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。

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。

在 Linux 系统里 2MSL 默认是 60 秒，那么一个 MSL 也就是 30 秒。Linux 系统停留在 TIME\_WAIT 的时间为固定的 60 秒。

其定义在 Linux 内核代码里的名称为 TCP\_TIMEWAIT\_LEN：

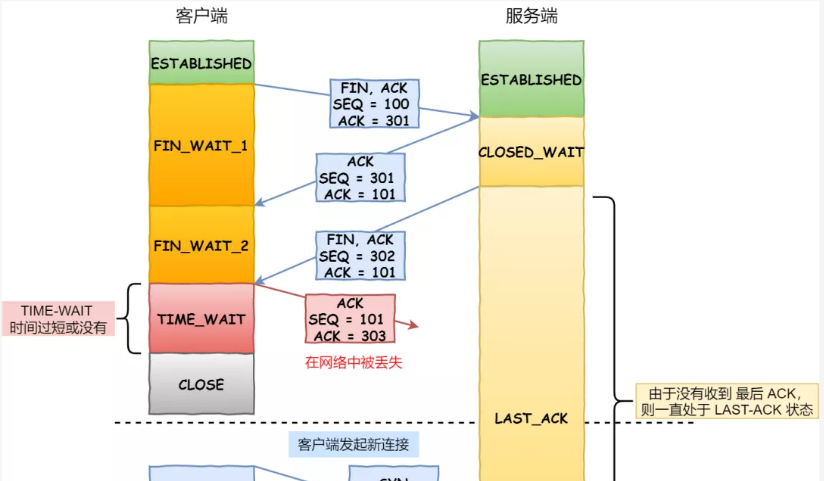
#define TCP\_TIMEWAIT\_LEN (60\*HZ)

如果要修改 TIME\_WAIT 的时间长度，只能修改 Linux 内核代码里 TCP\_TIMEWAIT\_LEN 的值，并重新编译 Linux 内核。

## 为什么需要 TIME\_WAIT？

### 保证连接正确关闭

* 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。



* 服务端没有收到四次挥手的最后一个 ACK 报文时，则会重发 FIN 关闭连接报文并等待新的 ACK 报文。
* 所以客户端在 TIME-WAIT 状态等待 2MSL 时间后，就可以保证双方的连接都可以正常的关闭。

### 防止旧连接的数据包

* 假设 TIME-WAIT 没有等待时间或时间过短，被延迟的数据包抵达后会有相同端口的 TCP 连接被复用后，被延迟的 SEQ = 301 抵达了客户端，那么客户端是有可能正常接收这个过期的报文，这就会产生数据错乱等严重的问题。
* 经过 2MSL 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。

## TIME\_WAIT 过多有什么危害？

如果服务器有处于 TIME-WAIT 状态的 TCP，则说明是由服务器方主动发起的断开请求。

过多的 TIME-WAIT 状态主要的危害有两种：

* 第一是内存资源占用；
* 第二是对端口资源的占用，一个 TCP 连接至少消耗一个本地端口；

如果服务端 TIME\_WAIT 状态过多，占满了所有端口资源，则会导致无法创建新连接。

## 已建立连接，但客户端出故障了怎么办？

TCP 有一个机制是保活机制。这个机制的原理是这样的：

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。

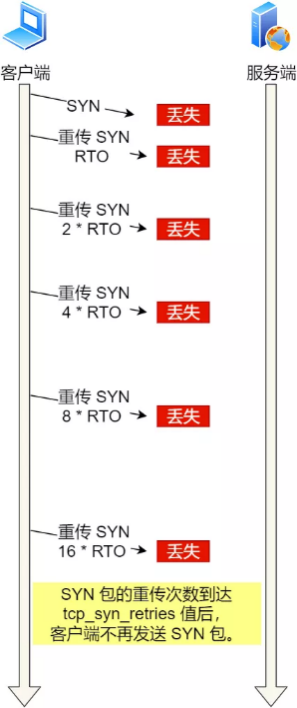
第二种，对端程序崩溃并重启。当 TCP 保活的探测报文发送给对端后，对端是可以响应的，但由于没有该连接的有效信息，会产生一个 RST 报文，这样很快就会发现 TCP 连接已经被重置。

第三种，是对端程序崩溃，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。

# TCP 三次握手异常情况

## 第一次握手丢包了

当客户端发起的 TCP 第一次握手 SYN 包，在超时时间内没收到服务端的 ACK，就会在超时重传 SYN 数据包，每次超时重传的 RTO 是翻倍上涨的，直到 SYN 包的重传次数到达 tcp\_syn\_retries 值后，客户端不再发送 SYN 包。

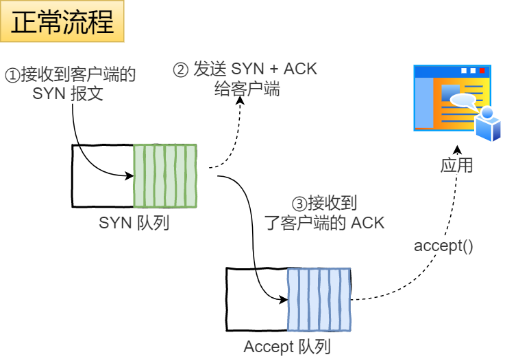


## 第二次握手丢包

当 TCP 第二次握手 SYN、ACK 包丢了后，客户端 SYN 包会发生超时重传，服务端收到客户的 SYN 包后，就会回 SYN、ACK 包，但是客户端一直没有回 ACK，服务端在超时后，重传了 SYN、ACK 包，接着一会，客户端超时重传的 SYN 包又抵达了服务端，服务端收到后，然后回了 SYN、ACK 包，但是SYN、ACK包的重传定时器并没有重置，还持续在重传，因为第二次握手在没收到第三次握手的 ACK 确认报文时，就会重传到最大次数。最后，客户端 SYN 超时重传次数达到了 5 次（tcp\_syn\_retries 默认值 5 次），就不再继续发送 SYN 包了。



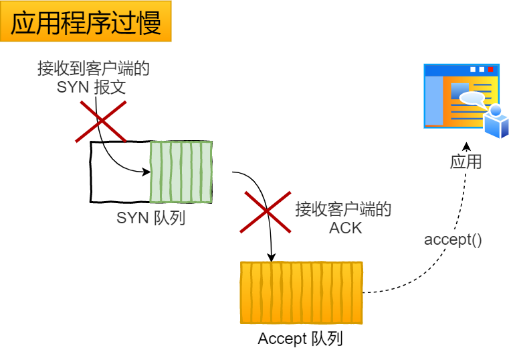
## 半连接队列与全连接队列



正常流程

* 当服务端接收到客户端的 SYN 报文时，会将其加入到内核的「 SYN 队列」；
* 接着发送 SYN + ACK 给客户端，等待客户端回应 ACK 报文；
* 服务端接收到 ACK 报文后，从「 SYN 队列」移除放入到「 Accept 队列」；
* 应用通过调用 accpet() socket 接口，从「 Accept 队列」取出连接。

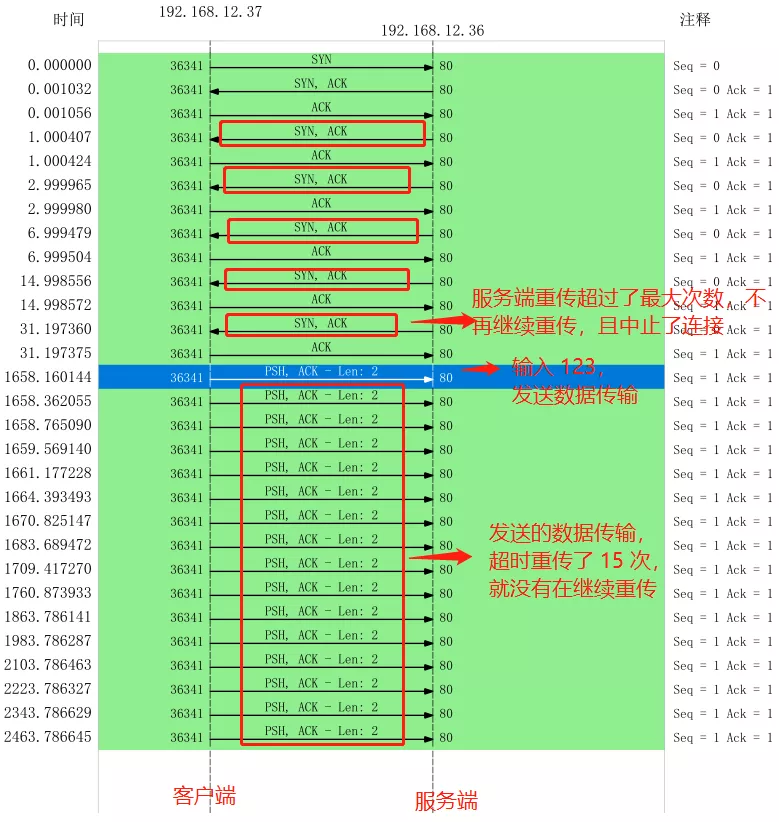
应用程序过慢



* 如果应用程序过慢时，就会导致「 Accept 队列」被占满。

## 第三次握手丢包

在建立 TCP 连接时，如果第三次握手的 ACK，服务端无法收到，则服务端就会短暂处于 SYN\_RECV 状态，而客户端会处于 ESTABLISHED 状态。



由于服务端一直收不到 TCP 第三次握手的 ACK，则会一直重传 SYN、ACK 包，直到重传次数超过 tcp\_synack\_retries 值（默认值 5 次）后，服务端就会断开 TCP 连接。

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。

而客户端则会有两种情况：

* 如果客户端没发送数据包，一直处于 ESTABLISHED 状态，然后经过 2 小时 11 分 15 秒才可以发现一个「死亡」连接，于是客户端连接就会断开连接。
* 如果客户端发送了数据包，一直没有收到服务端对该数据包的确认报文，则会一直重传该数据包，直到重传次数超过 tcp\_retries2 值（默认值 15 次）后，客户端就会断开 TCP 连接。

# 什么是 SYN 攻击，如何避免？

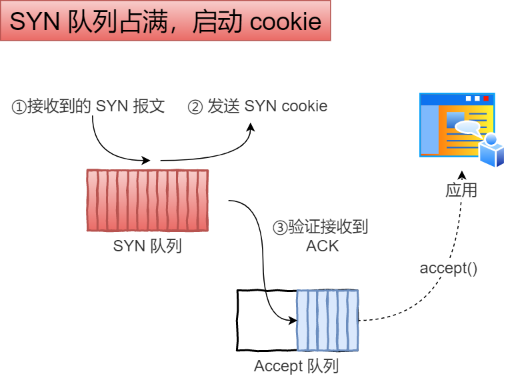
TCP 连接建立是需要三次握手，假设攻击者短时间伪造不同 IP 地址的 SYN 报文，服务端每接收到一个 SYN 报文，就进入SYN\_RCVD 状态，但服务端发送出去的 ACK + SYN 报文，无法得到未知 IP 主机的 ACK 应答，久而久之就会占满服务端的半连接队列，使得服务器不能为正常用户服务。

方式一

* 通过修改 Linux 内核参数，控制队列大小和当队列满时应做什么处理。
* 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。控制该队列的最大值如下参数：net.core.netdev\_max\_backlog
* SYN\_RCVD 状态连接的最大个数：net.ipv4.tcp\_max\_syn\_backlog
* 超出处理能时，对新的 SYN 直接回报 RST，丢弃连接：net.ipv4.tcp\_abort\_on\_overflow

方式二



* 如果不断受到 SYN 攻击，就会导致 SYN 队列（半连接队列）被占满，从而导致无法在建立新的连接。
* tcp\_syncookies 的方式可以应对 SYN 攻击的方法：net.ipv4.tcp\_syncookies = 1
* 
* 当 「 SYN 队列」满之后，后续服务器收到 SYN 包，不进入「 SYN 队列」；
* 计算出一个 cookie 值，再以 SYN + ACK 中的「序列号」返回客户端，
* 服务端接收到客户端的应答报文时，服务器会检查这个 ACK 包的合法性。如果合法，直接放入到「 Accept 队列」。
* 最后应用通过调用 accpet() socket 接口，从「 Accept 队列」取出的连接。

# TCP 四次挥手异常情况

## 第一次挥手丢失了

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\_orphan\_retries 参数控制。

当客户端重传 FIN 报文的次数超过 tcp\_orphan\_retries 后，就不再发送 FIN 报文，直接进入到 close 状态。

## 第二次挥手丢失了

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 CLOSE\_WAIT 状态。

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 FIN\_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FIN\_WAIT2 状态不可以持续太久，而 tcp\_fin\_timeout 控制了这个状态下连接的持续时长，默认值是 60 秒。

这意味着对于调用 close 关闭的连接，如果在 60 秒后还没有收到 FIN 报文，客户端（主动关闭方）的连接就会直接关闭。

## 第三次挥手丢失了

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 CLOSE\_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 CLOSE\_WAIT 状态时，调用了 close 函数，内核就会发出 FIN 报文，同时连接进入 LAST\_ACK 状态，等待客户端返回 ACK 来确认连接关闭。

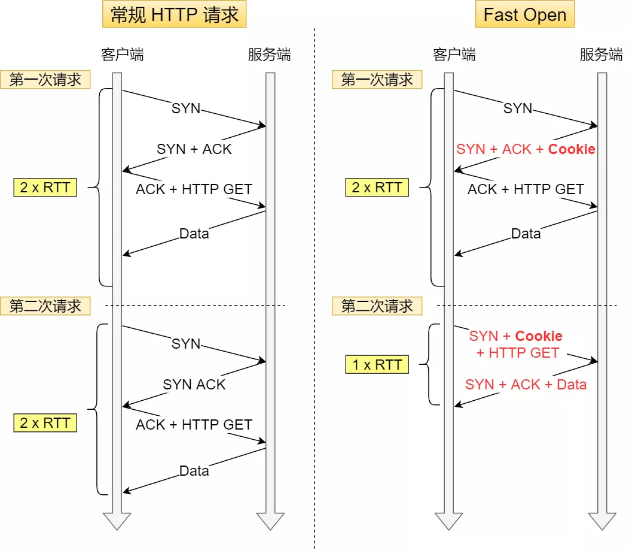
如果迟迟收不到这个 ACK，服务端就会重发 FIN 报文，重发次数仍然由 tcp\_orphan\_retries 参数控制，这与客户端重发 FIN 报文的重传次数控制方式是一样的。

## 第四次挥手丢失了

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 tcp\_orphan\_retries 参数控制。

# TCP 快速建立连接

在 Linux 3.7 内核版本中，提供了 TCP Fast Open 功能，这个功能可以减少 TCP 连接建立的时延。



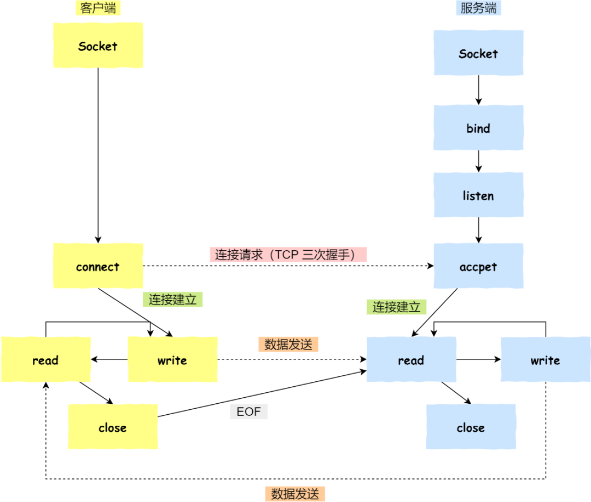
在第一次建立连接的时候，服务端在第二次握手产生一个 Cookie （已加密）并通过 SYN、ACK 包一起发给客户端，于是客户端就会缓存这个 Cookie，所以第一次发起 HTTP Get 请求的时候，还是需要 2 个 RTT 的时延；

在下次请求的时候，客户端在 SYN 包带上 Cookie 发给服务端，就提前可以跳过三次握手的过程，因为 Cookie 中维护了一些信息，服务端可以从 Cookie 获取 TCP 相关的信息，这时发起的 HTTP GET 请求就只需要 1 个 RTT 的时延；

客户端在请求并存储了 Fast Open Cookie 之后，可以不断重复 TCP Fast Open 直至服务器认为 Cookie 无效（通常为过期）

# Socket编程

## Socket 编程流程



* 服务端和客户端初始化 socket，得到文件描述符；
* 服务端调用 bind，将绑定在 IP 地址和端口;
* 服务端调用 listen，进行监听；
* 服务端调用 accept，等待客户端连接；
* 客户端调用 connect，向服务器端的地址和端口发起连接请求；
* 服务端 accept 返回用于传输的 socket 的文件描述符；
* 客户端调用 write 写入数据；服务端调用 read 读取数据；
* 客户端断开连接时，会调用 close，那么服务端 read 读取数据的时候，就会读取到了 EOF，待处理完数据后，服务端调用 close，表示连接关闭。

这里需要注意的是，服务端调用 accept 时，连接成功了会返回一个已完成连接的 socket，后续用来传输数据。

所以，监听的 socket 和真正用来传送数据的 socket，是「两个」 socket，一个叫作监听 socket，一个叫作已完成连接 socket。

成功连接建立之后，双方开始通过 read 和 write 函数来读写数据，就像往一个文件流里面写东西一样。

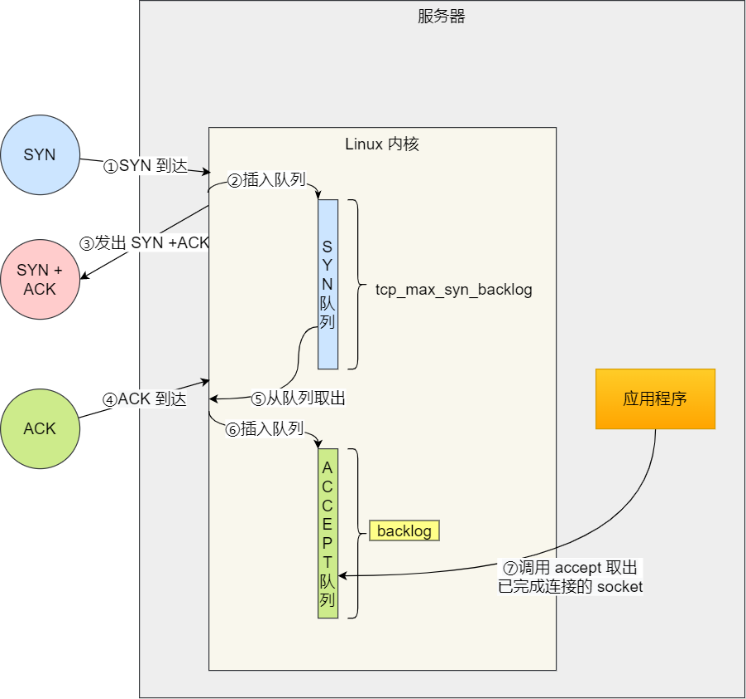
## listen 时候参数 backlog 的意义？

int listen (int socketfd, int backlog)

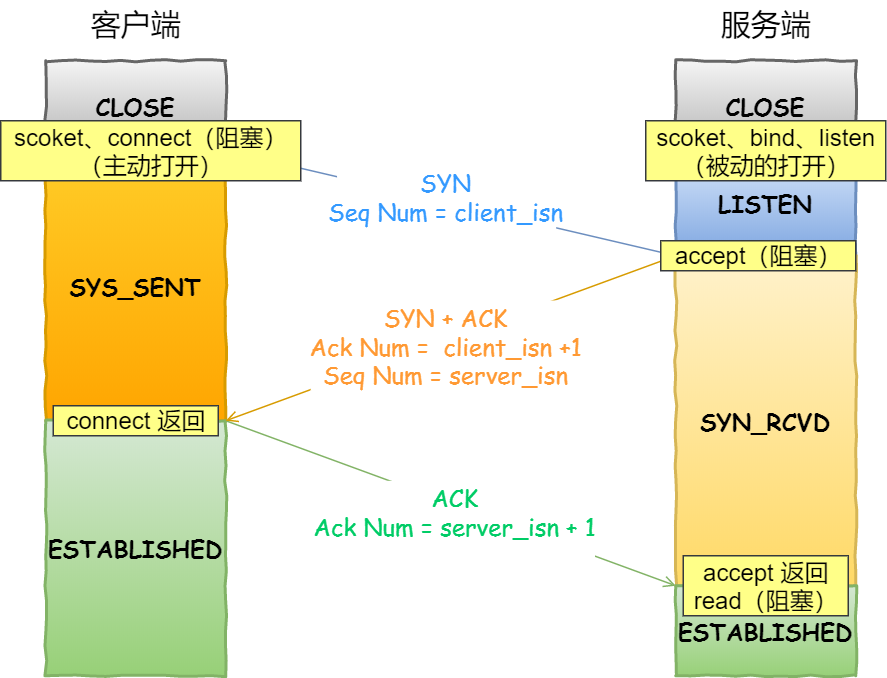
* 参数一 socketfd 为 socketfd 文件描述符
* 参数二 backlog，这参数在历史版本有一定的变化
  + 在早期 Linux 内核 backlog 是 SYN 队列大小，也就是未完成的队列大小。
  + 在 Linux 内核 2.2 之后，backlog 变成 accept 队列，也就是已完成连接建立的队列长度，所以现在通常认为 backlog 是 accept 队列。但是上限值是内核参数 somaxconn 的大小，也就说 accpet 队列长度 = min(backlog, somaxconn)。

Linux内核中会维护两个队列：

* 半连接队列（SYN 队列）：接收到一个 SYN 建立连接请求，处于 SYN\_RCVD 状态；
* 全连接队列（Accpet 队列）：已完成 TCP 三次握手过程，处于 ESTABLISHED 状态；



## accept 发生在三次握手的哪一步？



从上面的描述过程，我们可以得知客户端 connect 成功返回是在第二次握手，服务端 accept 成功返回是在三次握手成功之后。

## 客户端调用 close 了，连接是断开的流程是什么？



客户端调用 close，表明客户端没有数据需要发送了，则此时会向服务端发送 FIN 报文，进入 FIN\_WAIT\_1 状态；

服务端接收到了 FIN 报文，TCP 协议栈会为 FIN 包插入一个文件结束符 EOF 到接收缓冲区中，应用程序可以通过 read 调用来感知这个 FIN 包。这个 EOF 会被放在已排队等候的其他已接收的数据之后，这就意味着服务端需要处理这种异常情况，因为 EOF 表示在该连接上再无额外数据到达。此时，服务端进入 CLOSE\_WAIT 状态；

接着，当处理完数据后，自然就会读到 EOF，于是也调用 close 关闭它的套接字，这会使得服务端发出一个 FIN 包，之后处于 LAST\_ACK 状态；

客户端接收到服务端的 FIN 包，并发送 ACK 确认包给服务端，此时客户端将进入 TIME\_WAIT 状态；

服务端收到 ACK 确认包后，就进入了最后的 CLOSE 状态；

客户端经过 2MSL 时间之后，也进入 CLOSE 状态；

# 重传机制

## 超时重传

两种情况发生超时重传

* 数据包丢失
* 确认应答丢失

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

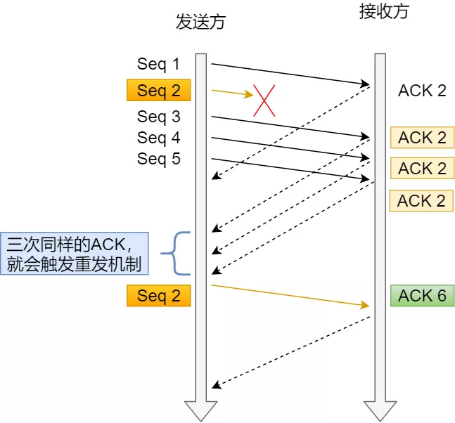
* 当超时时间 RTO 较大时，重发就慢，丢了老半天才重发，没有效率，性能差；
* 当超时时间 RTO 较小时，会导致可能并没有丢就重发，于是重发的就快，会增加网络拥塞，导致更多的超时，更多的超时导致更多的重发。

精确的测量超时时间 RTO 的值是非常重要的，这可让我们的重传机制更高效。

根据上述的两种情况，我们可以得知，超时重传时间 RTO 的值应该略大于报文往返 RTT 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。

## 快速重传



* 第一份 Seq1 先送到了，于是就 Ack 回 2；
* 结果 Seq2 因为某些原因没收到，Seq3 到达了，于是还是 Ack 回 2；
* 后面的 Seq4 和 Seq5 都到了，但还是 Ack 回 2，因为 Seq2 还是没有收到；
* 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。
* 最后，接收到收到了 Seq2，此时因为 Seq3，Seq4，Seq5 都收到了，于是 Ack 回 6 。

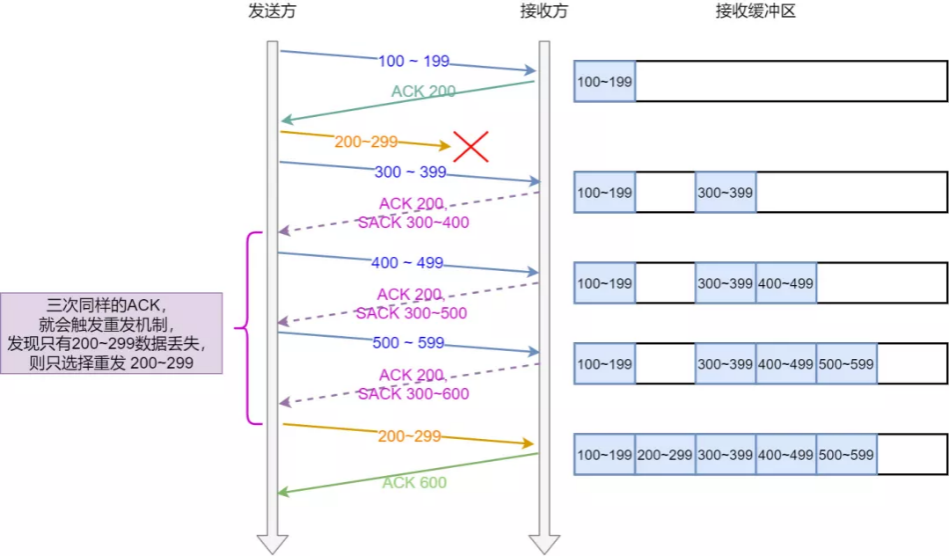
所以，快速重传的工作方式是当收到三个相同的 ACK 报文时，会在定时器过期之前，重传丢失的报文段。

## SACK

还有一种实现重传机制的方式叫：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，它可以将缓存的地图发送给发送方，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。

如上图，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。

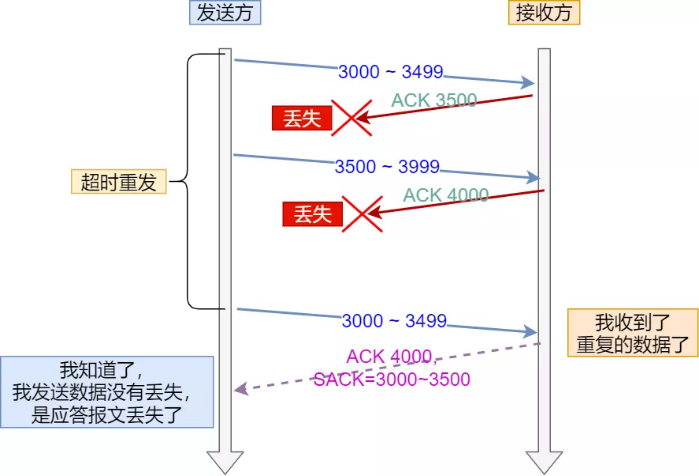


如果要支持 SACK，必须双方都要支持。在 Linux 下，可以通过 net.ipv4.tcp\_sack 参数打开这个功能（Linux 2.4 后默认打开）。

## D-SACK

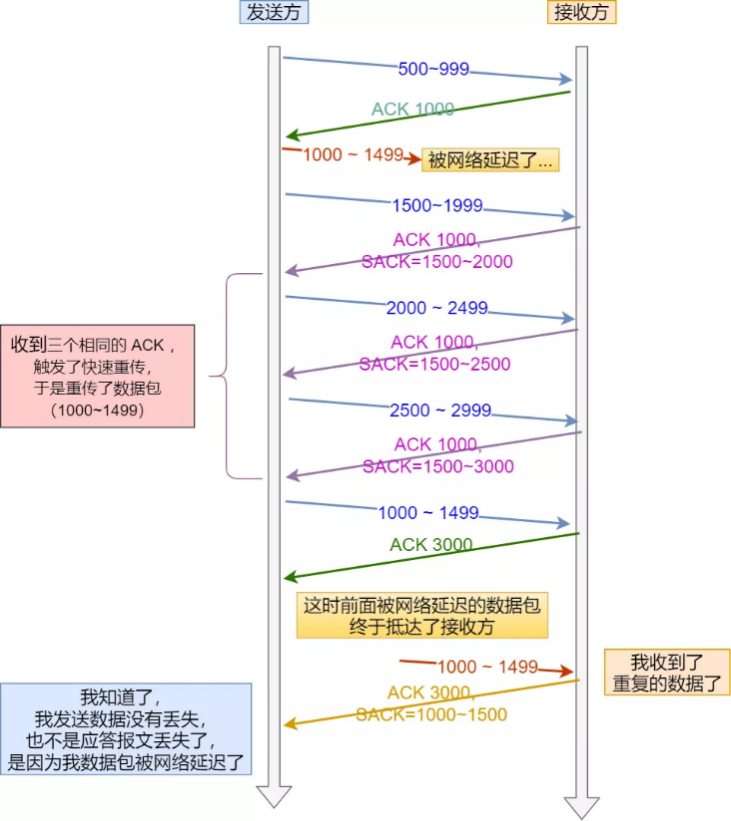
Duplicate SACK 又称 D-SACK，其主要使用了 SACK 来告诉「发送方」有哪些数据被重复接收了。

栗子一号：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。

栗子二号：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。

有这么几个好处

* 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;
* 可以知道是不是「发送方」的数据包被网络延迟了;
* 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

## 概念

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。

## 窗口大小由哪一方决定？

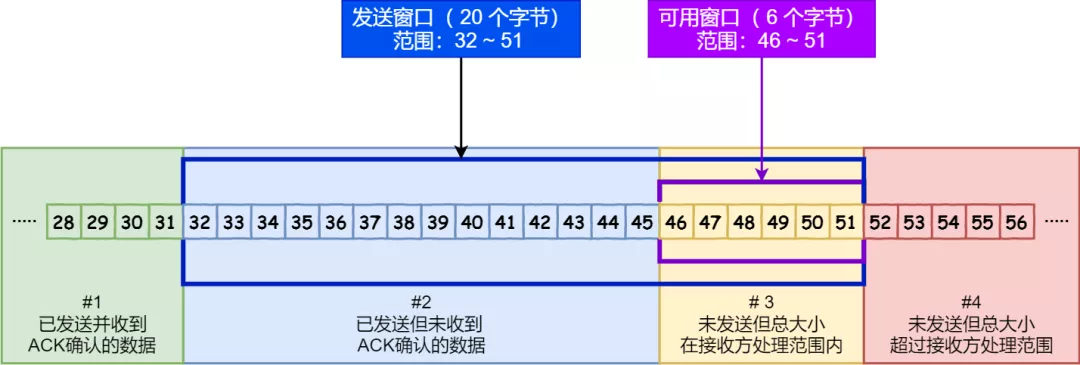
通常窗口的大小是由接收方的决定的。

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。

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。

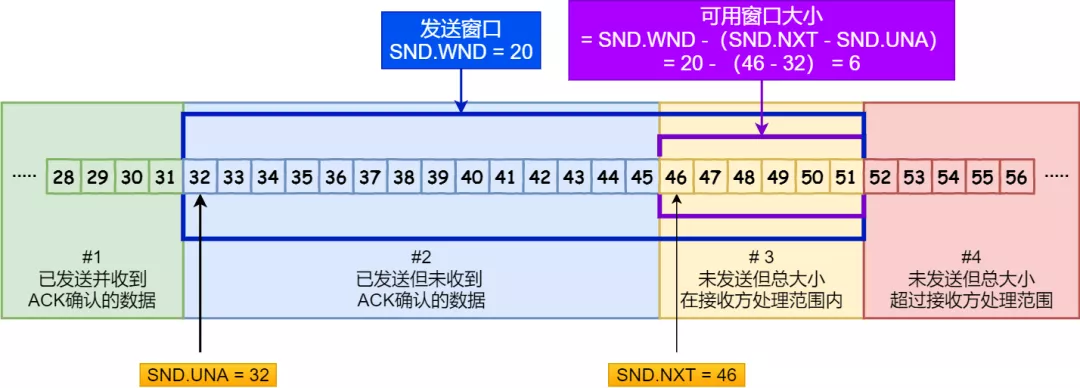
## 发送方的滑动窗口



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。

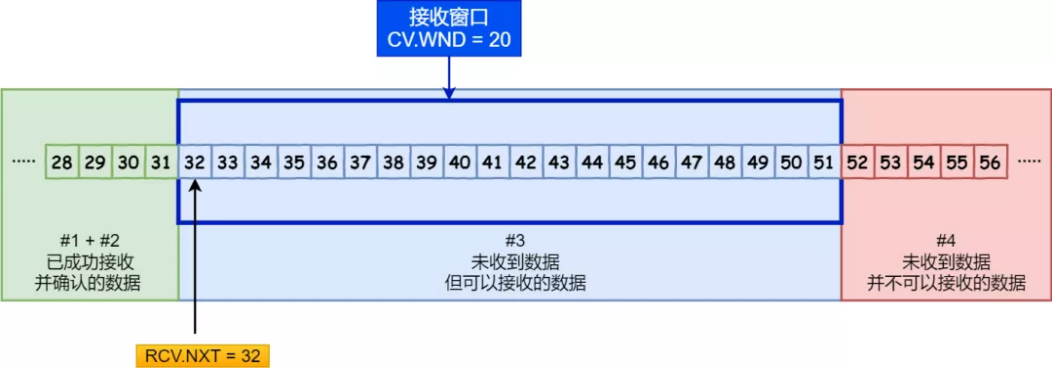
程序是如何表示发送方的四个部分的呢？

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。

## 接收方的滑动窗口



## 接收窗口和发送窗口的大小是相等的吗？

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。

# 流量控制

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。

流量控制是避免「发送方」的数据填满「接收方」的缓存

## 操作系统缓冲区与滑动窗口的关系

当应用程序没有及时读取缓存时，发送窗口和接收窗口的变化。

* 客户端作为发送方，服务端作为接收方，发送窗口和接收窗口初始大小为 360；
* 服务端非常的繁忙，当收到客户端的数据时，应用层不能及时读取数据。
* 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，操作系统会减少接收缓冲区大小，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，过段时间在减少缓存，这样就可以避免了丢包情况。

## 窗口关闭

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。

接收方向发送方通告窗口大小时，是通过 ACK 报文来通告的。

## TCP 是如何解决窗口关闭时，潜在的死锁现象呢？

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 ( 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。

窗口探查探测的次数一般为 3 此次，每次次大约 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。

## 糊涂窗口综合症

如果接收方腾出几个字节并告诉发送方现在有的小字节的窗口，而发送方会义无反顾地发送这几个小字节，这就是糊涂窗口综合症。

这个问题可以归结为小包的问题，就是由于发送端和接收端上的处理不一致，导致网络上产生很多的小包，之前也介绍过避免网络上产生过多小包的措施，比如Nagle算法。

# 拥塞控制

## 拥塞控制的作用

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….

控制的目的就是避免「发送方」的数据填满整个网络

## 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。

## 拥塞窗口 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

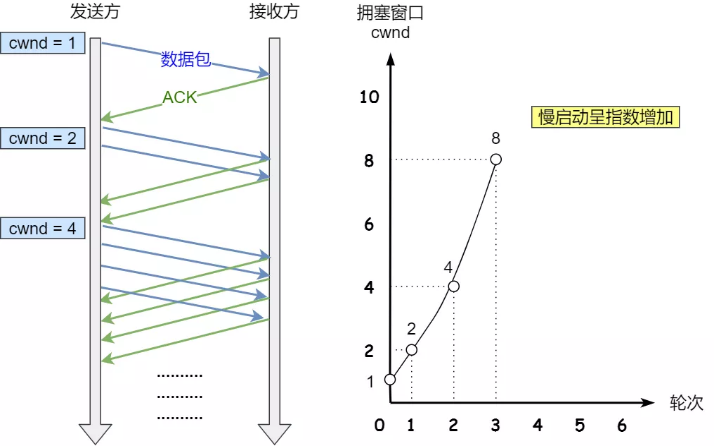
## 那么怎么知道当前网络是否出现了拥塞呢？

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用拥塞。

## 拥塞控制有哪些控制算法？

### 慢启动

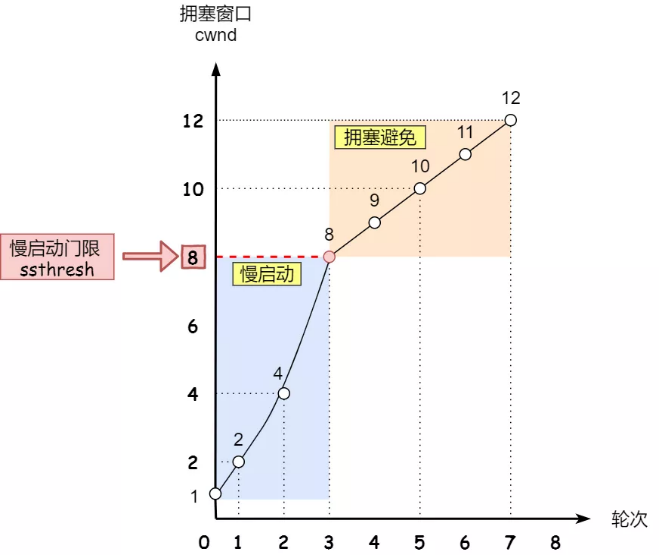
* 当发送方每收到一个 ACK，就拥塞窗口 cwnd 的大小就会加 1。



* 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」。

### 拥塞避免

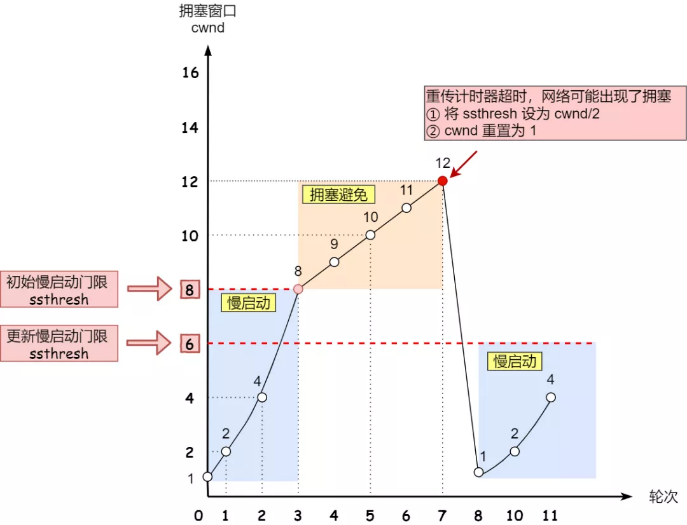
* 当拥塞窗口 cwnd 「超过」慢启动门限 ssthresh 就会进入拥塞避免算法。
* 它的规则是：每当收到一个 ACK 时，cwnd 增加 1/cwnd。



* 我们可以发现，拥塞避免算法就是将原本慢启动算法的指数增长变成了线性增长，还是增长阶段，但是增长速度缓慢了一些。
* 就这么一直增长着后，网络就会慢慢进入了拥塞的状况了，于是就会出现丢包现象，这时就需要对丢失的数据包进行重传。当触发了重传机制，也就进入了「拥塞发生算法」。

### 拥塞发生

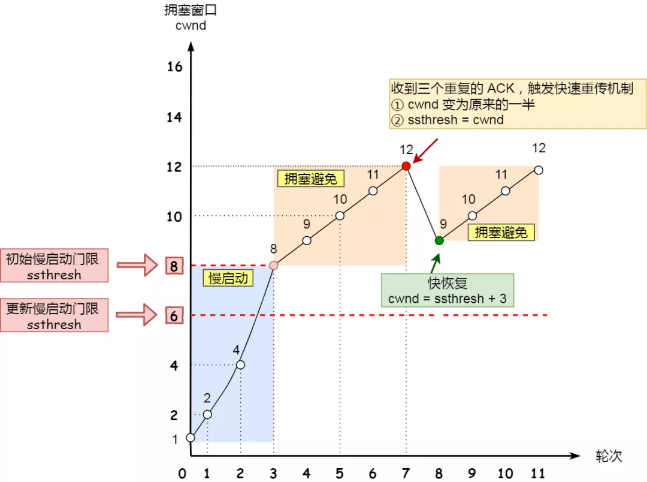
* 当网络出现拥塞，也就是会发生数据包重传，重传机制主要有两种：
* 超时重传
  + ssthresh 设为 cwnd/2
  + cwnd 重置为 1



* + 接着，就重新开始慢启动，慢启动是会突然减少数据流的。这真是一旦「超时重传」，马上回到解放前。但是这种方式太激进了，反应也很强烈，会造成网络卡顿。
* 快速重传
  + TCP 认为这种情况不严重，因为大部分没丢，只丢了一小部分
  + 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。
* 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，设置 cwnd 为 ssthresh，接着就进入了拥塞避免算法

# TCP 延迟确认

为了解决 ACK 传输效率低问题，所以就衍生出了 TCP 延迟确认。

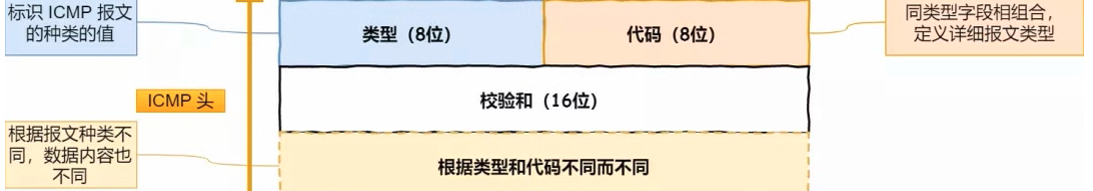
## TCP 延迟确认的策略：

1. 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
2. 当没有响应数据要发送时，ACK 将会延迟一段时间，以等待是否有响应数据可以一起发送
3. 如果在延迟等待发送 ACK 期间，对方的第二个数据报文又到达了，这时就会立刻发送 ACK

# ICMP协议功能

确认 IP 包是否成功送达目标地址、报告发送过程中 IP 包被废弃的原因和改善网络设置等。在 IP 通信中如果某个 IP 包因为某种原因未能达到目标地址，那么这个具体的原因将由 ICMP 负责通知。

# ICMP包头格式



# ICMP类型字段

查询报文类型：用于诊断的查询消息

* 回送应答（Echo Reply）—— 类型 0
* 回送请求（Echo Request）—— 类型8

Linux实现Ping相比原生的 ICMP，这里多了三个字段

* 标识符：用以区分是哪个应用程序发 ICMP 包，比如用进程 PID 作为标识符；
* 序号：序列号从 0 开始，每发送一次新的回送请求就会加 1， 可以用来确认网络包是否有丢失。
* 选项：ping 还会存放发送请求的时间值，来计算往返时间，说明路程的长短

差错报文类型：通知出错原因的错误消息

### 目标不可达消息 —— 类型为 3

* 网络不可达 —— 代码为 0
* 主机不可达 —— 代码为 1
* 协议不可达 —— 代码为 2
  + 当主机使用 TCP 协议访问对端主机时，能找到对端的主机了，可是对端主机的防火墙已经禁止 TCP 协议访问，那么会通过 ICMP 协议以协议不可达的原因告知主机。
* 端口不可达 —— 代码为 3
  + 当主机访问对端主机 8080 端口时，这次能找到对端主机了，防火墙也没有限制，可是发现对端主机没有进程监听 8080 端口，那么会通过 ICMP 协议以端口不可达的原因告知主机。
* 要分片但分不了片位 —— 代码为 4
  + 发送端主机发送 IP 数据报时，将 IP 首部的分片禁止标志位设置为1。根据这个标志位，途中的路由器遇到超过 MTU 大小的数据包时，不会进行分片，而是直接抛弃。随后，通过一个 ICMP 的不可达消息类型，代码为 4 的报文，告知发送端主机。

### 原点抑制消息 —— 类型 4

* 使用低速广域线路的情况下，路由器可能会遇到网络拥堵的问题。ICMP 原点抑制消息的目的就是为了缓和这种拥堵情况。

### 重定向消息 —— 类型 5

* 如果路由器发现发送端主机使用了「不是最优」的路径发送数据，那么它会返回一个 ICMP 重定向消息给这个主机。
* 在这个消息中包含了最合适的路由信息和源数据。这主要发生在路由器持有更好的路由信息的情况下。路由器会通过这样的 ICMP 消息告知发送端，让它下次发给另外一个路由器。

### 超时消息 —— 类型 11

* TTL减小到0时，IP 路由器将会发送一个 ICMP 超时消息给发送端主机，并通知该包已被丢弃。

# ICMP协议应用

## 查询报文类型

Ping

## 差错报文类型

traceroute

* 第一个作用就是故意设置特殊的 TTL，来追踪去往目的地时沿途经过的路由器。当差错报文类型是端口不可达时，说明发送方发出的 UDP 包到达了目的主机。
* 第二个作用是故意设置不分片，从而确定路径的 MTU。这样做的目的是为了路径MTU发现。因为有的时候我们并不知道路由器的 MTU 大小，以太网的数据链路上的 MTU 通常是 1500 字节，但是非以太网的 MTU 值就不一样了，所以我们要知道 MTU 的大小，从而控制发送的包大小。它的工作原理如下：  
  1. 首先在发送端主机发送 IP 数据报时，将 IP 包首部的分片禁止标志位设置为 1。根据这个标志位，途中的路由器不会对大数据包进行分片，而是将包丢弃。  
  2. 随后，通过一个 ICMP 的不可达消息将数据链路上 MTU 的值一起给发送主机，不可达消息的类型为「需要进行分片但设置了不分片位」。  
  3. 发送主机端每次收到 ICMP 差错报文时就减少包的大小，以此来定位一个合适的 MTU 值，以便能到达目标主机。

# 路由表字段

目的网络IP地址，子网掩码，下一跳IP地址，接口

# 静态路由算法

# 动态路由算法

距离-向量路由算法

* 路由器只掌握物理相连的邻居及链路费用
* 它为它的邻居提供从自己到网络中所有其他结点的最低费用估计
* RIP

## 链路状态路由算法

* 所有路由器掌握完整的网络拓扑和链路费用信息
* 每个结点通过广播的方式与所有其他结点交谈，但它仅告诉它们与它直接相连的链路的费用
* OSPF

# 层次路由

## 内部网关协议

RIP, OSPF

## 外部网关协议

BGP

