# 计算机网络

# Http：

## 状态码：

200：请求成功

201：请求成功已创建

301:重定向-永久移动

302:重定向-临时移动

400：一般是参数错误

401：未授权

403：请求禁止

404：资源不存在

405：禁止请求的方法

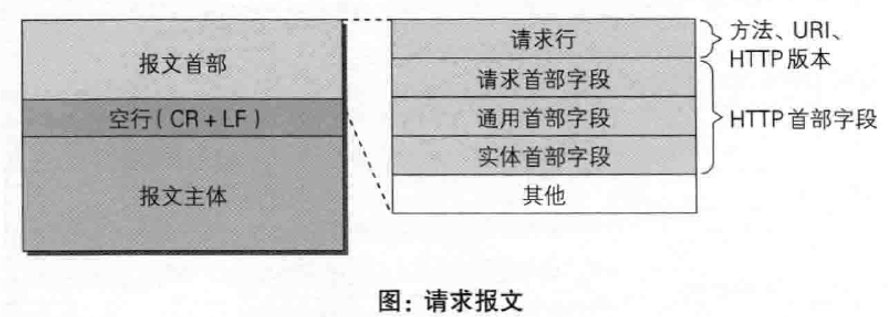
500：服务器内部错误

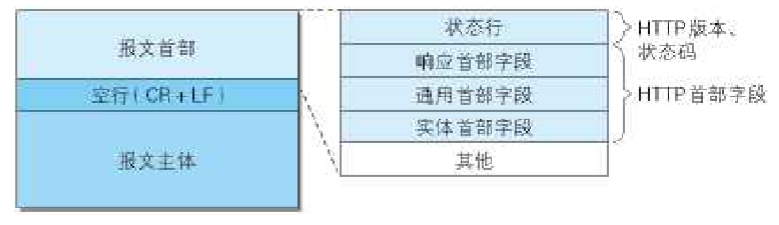
502：网关收到无效的响应

503：服务端停机维护

504：网关连接服务器超时

## Http报文结构：





## 首部字段：

<https://cloud.tencent.com/developer/article/1407959>

### 通用首部字段：

Cache-Control：max-age=5 缓存行为控制，5s内浏览器不要请求新的内容

Connection：keep-alive 连接类型：控制连接是否是长连接

Date：创建报文时间

Pragma：报文指令

Transfer-Encoding：报文主体编码格式：

### 请求首部头部：

Host：请求的服务器ip

Accept：接受的媒体类型

Accept-Charset：支持的字符集

Accept-Encoding：支持的编码集

Accept-Language：支持的语言

Referer：请求中的原始获取方

User-Agent：客户端信息

If-Match：{eTag}当请求的资源的eTag与给定的eTage相同时才响应请求

If-No-Match：{eTag}当请求的资源的eTag与给定的eTage不同时才响应请求

If-Modified-Since：在给定时间点后修改了数据，服务端才返回响应/与If-No-Match一起使用的时候，以If-Modified-Since为准

### 响应首部字段：

Accept-Range：是否接受请求范围

ETag：资源的ETag信息

Location：客户端重定向到指定的url

Server：服务器相关信息

Vary：代理服务器的管理信息

### 实体首部字段：

allow：资源支持的Http方法

Content-Encoding：内容编码

Content-Length：内容长度

Content-Language：内容预览

Content-Type：内容类型

Content-Location：替代对应资源的URL

Expires：实体过期日期

Last-Modified：资源最后修改时间

# Http1/Http2/Http3的区别

<https://network.51cto.com/article/634943.html>

<https://blog.fundebug.com/2019/03/07/understand-http2-and-http3/>

## Http 1.x存在的问题点：

1:每个资源的请求都需要建立一个tcp连接，虽然有keep-alive，但是还是会有很多连接

2:浏览器也会对同一个域名同时存在的连接数有限制，如果有请求阻塞，很可能会影响其他的请求，导致页面加载慢。

3:header数据大（cookie等信息），导致传输速度慢

## Http2.0：

2.0解决了一些1.0存在的痛点问题，2.0完全支持1.x的语法

1:http2.0每个域名开启一个tcp连接，连接时可以复用的，使用多路复用器管理连接。（如果连接可复用，那么它肯定是把请求做成了有状态的。）

2:在数据传输层面，将2.0的包的header进行二进制压缩，减少传输数据的大小

3:将数据包拆分成帧，分帧发送

### 问题：

http1和http2都存在头部阻塞的问题（tcp丢包重传），对于http1来说，因为这个问题只会影响一次http请求，但是对于http2而言，因为同一个域名复用一个连接，一个数据丢失，会阻塞之后的所有的连接。（因为一个线程处理，这个请求没有完成，怎么会处理下一个请求）

## Http3.0

底层不使用TCP协议，使用UDP来实现，从协议层面解决TCP给Http带来的头部阻塞问题。

# Https：

## Http存在以下问题：

1:明文通信，裸奔

2:无法验证身份

3:信息可能会被篡改

Https解决了上述问题  HTTPS = HTTP + SSL / TLS。

## 明文通信问题：

https发送的消息都是加密的，其过程如下

非对称加密过程：

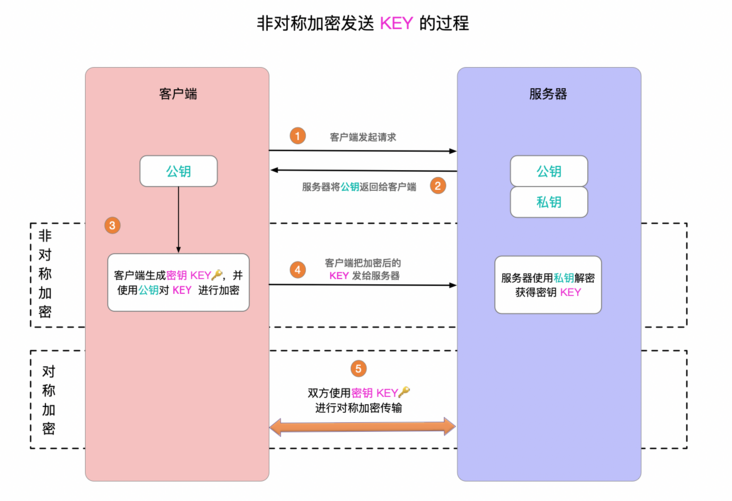
前提：使用RSA加密算法协商 对称加密密钥的key；RSA由服务端提供

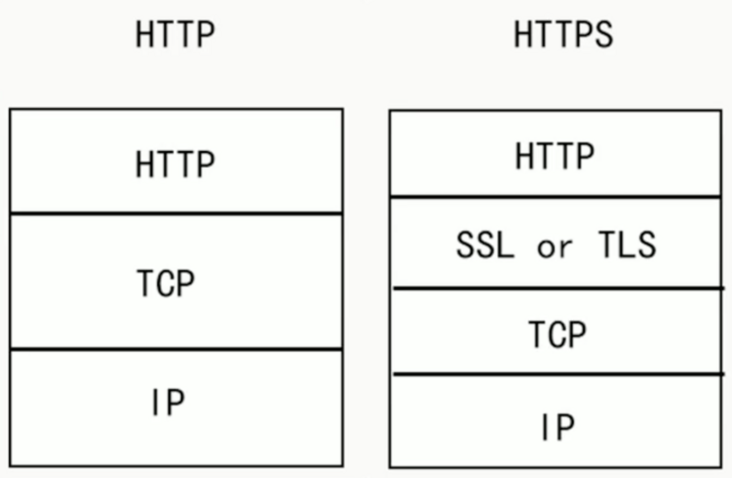
1:服务端返回自己的公钥

2:客户端生成对称加密的key，使用公钥加密返回给服务端

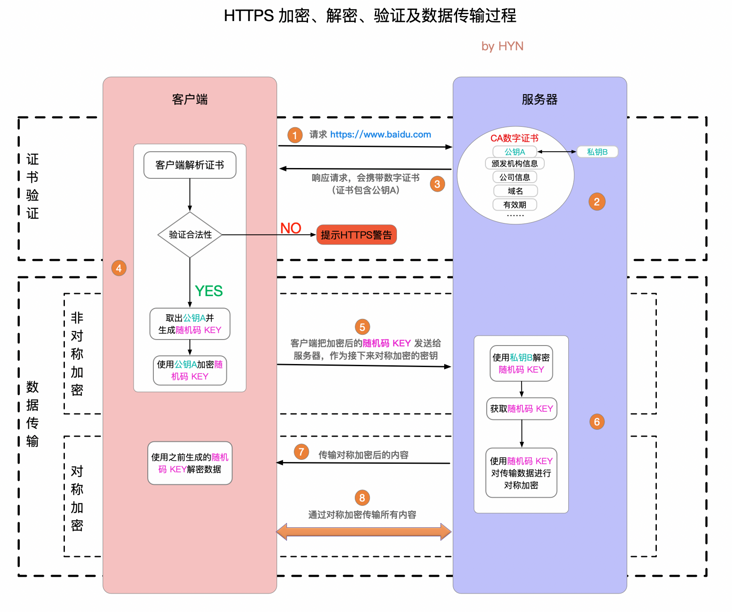
3:服务端使用私钥解密key

4:之后双方使用key进行对称加密通信





## Https原理：



1:客户端发送请求资源

2:收到请求之后，tls层发送服务端的 CA签发的证书。

3:客户端验证服务端证书

3-1:取出client端维护的根证书列表（根证书包含了根证书机构的公钥），使用根证书验证 ca机构的私钥加密的hash值，验证是否过期，域名是否匹配

3-2:验证失败，浏览器提示https警告

4:验证成功，使用服务端证书中的公钥进行下面的非对称加密协商key

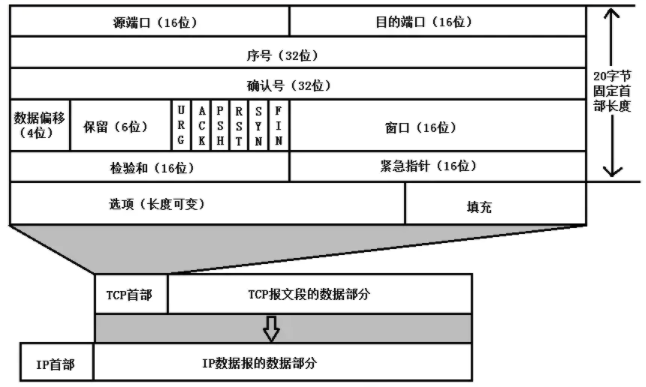
5:对称加密发送数据。

# TCP常用知识：

<https://www.jianshu.com/p/7b83a0070588>

## 包结构

TCP首部+TCP报文数据段



### TCP首部：

源端口/目的端口

序号：当前包的序列号

确认号：收到对方的包的序号+1

数据偏移：数据段的对于整个包的offset

控制字段：

SYN：请求建立连接字段，SYN=1，ACK=0代表client请求建立连接（client建立连接时控制字段ACK才会为0，其他都会为1）

ACK：代表确认号 有效

PSH：数据推送阶段标识位

FIN：结束标志位

URG：紧急标识位，让这个包有限发出和 处理，需配合紧急指针一起使用

RST：TCP连接遇到错误，需要重新创建连接

校验和：校验报文完整性

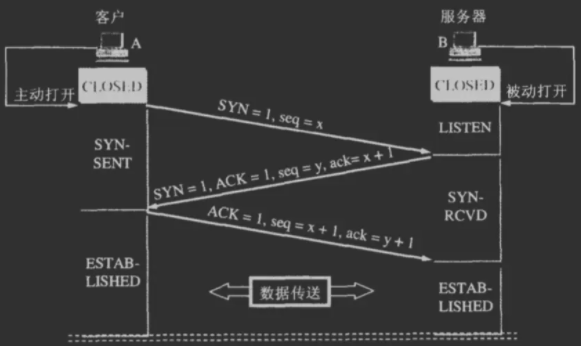
紧急指针：紧急数据的字节数（紧急数据之后就是普通数据）

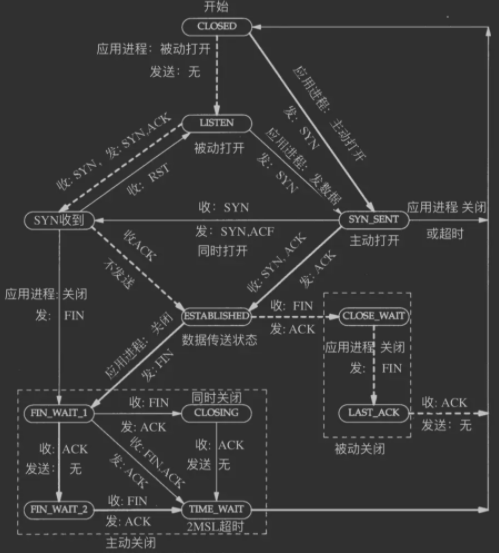
### TCP报文

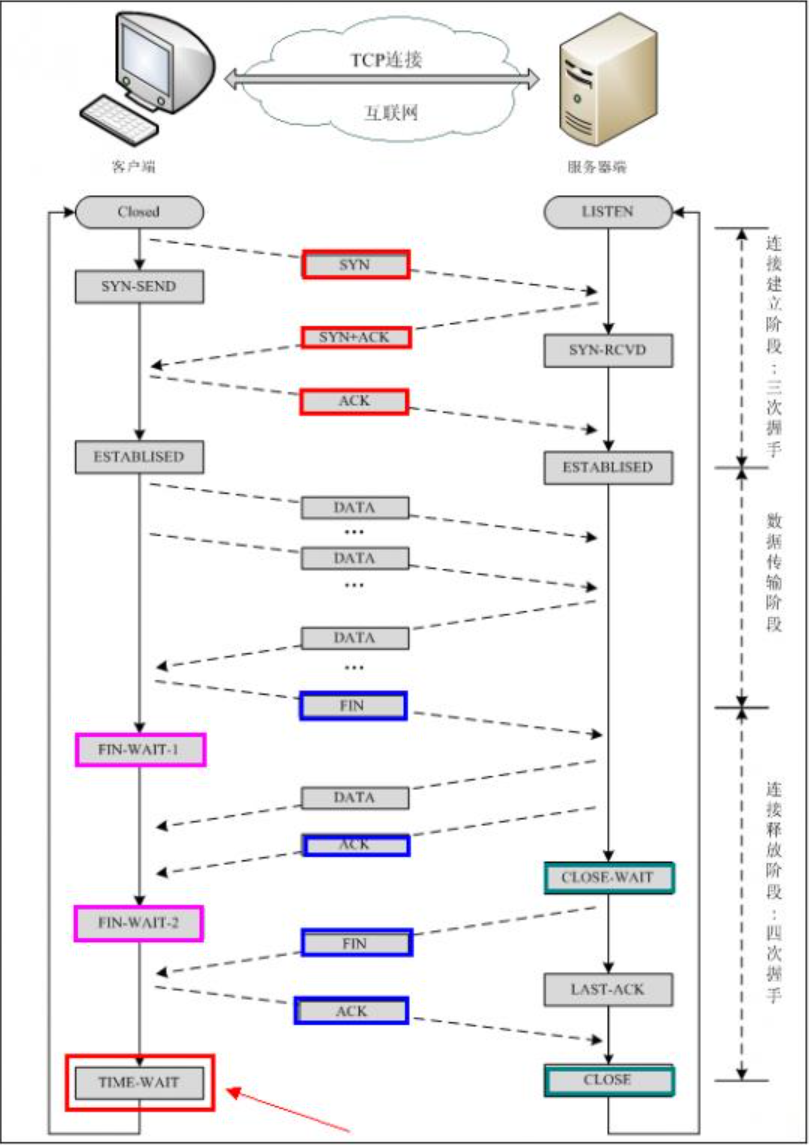
存储数据的地方

## 状态图

三次握手：







# TCP协议详解

https://www.cnblogs.com/xiaolincoding/p/12732052.html

###### 前置知识：

因为网卡MTU的限制，每次发送的tcp包+网络层协议头+数据链路层协议头大小不能超过MTU，所以发送大数据的时候，一个tcp包是放不下我们需要发送的消息的。所以引入了

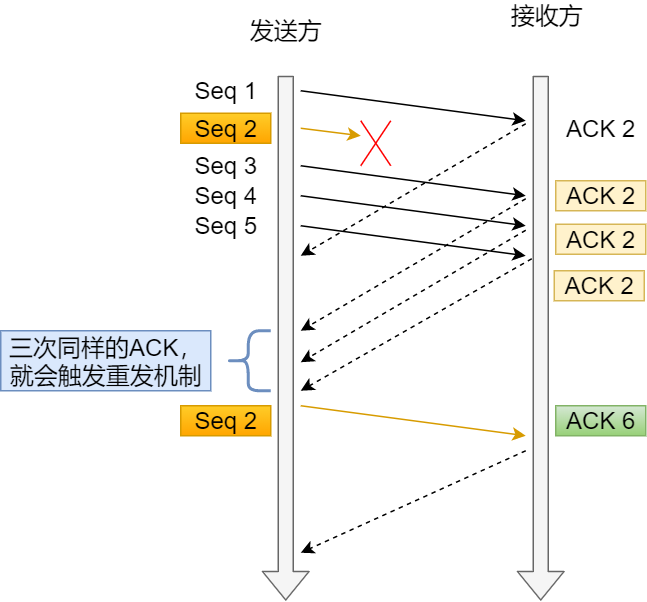
### 重传协议：

###### 背景：

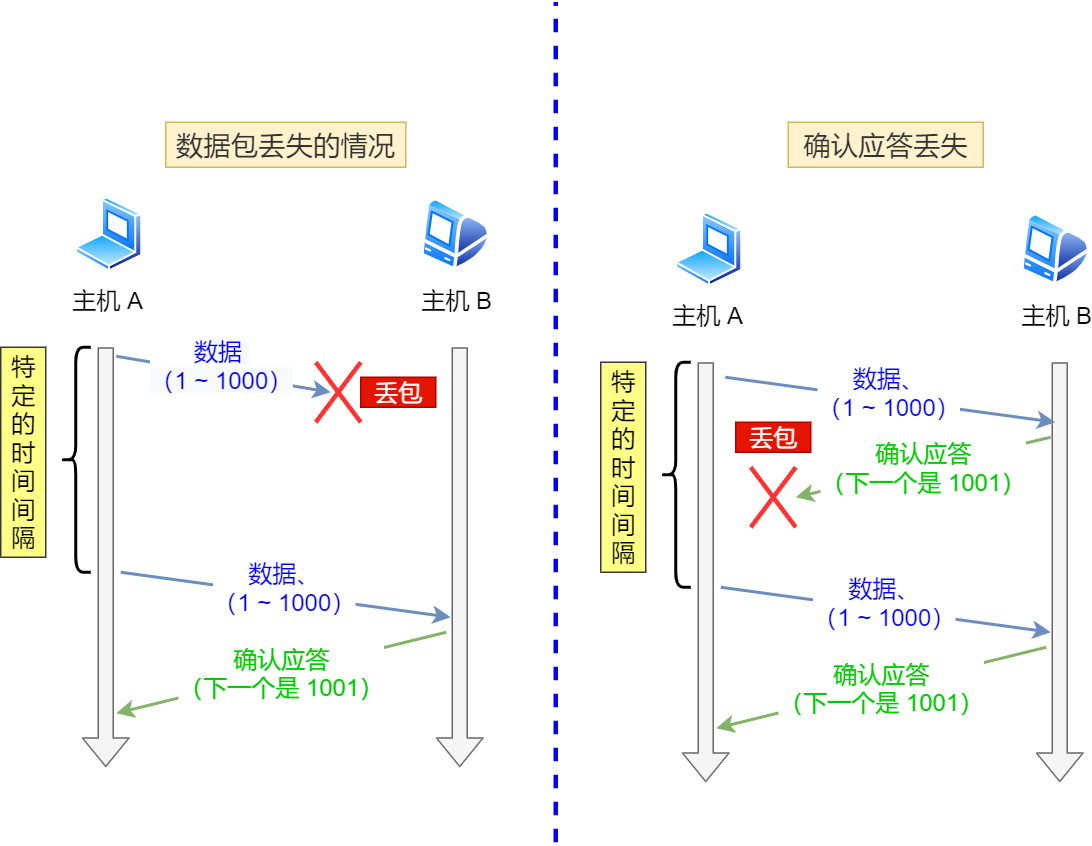
在网络中发送包，很可能会丢，因为tcp是面向连接的，稳定的协议，所以包一定不能丢。所以需要引入算法保证包不会丢失。

###### 重传算法：

**快速重传**：在接收端发现收到的包不是预期的，就会发送一个上一个接受的到的包的ack（里面的ack=x+1），表示我需要你再传一下x+1序号的包。



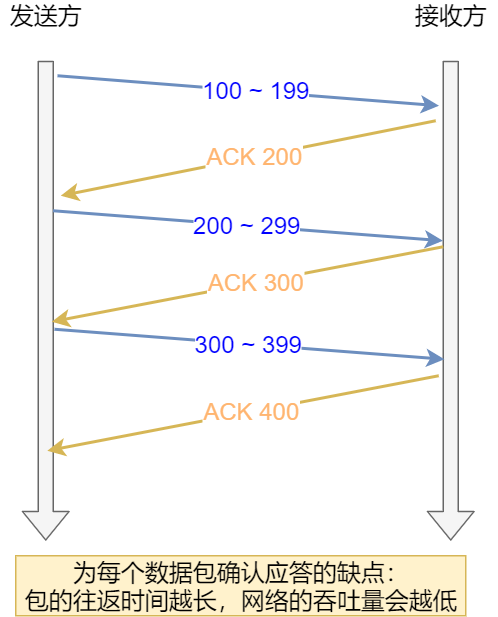
**超时重传**：这个是发送方的算法，发送方发现server端一直没有返回 ack包，client就会认为这个包丢了，就会重传



## 滑动窗口：

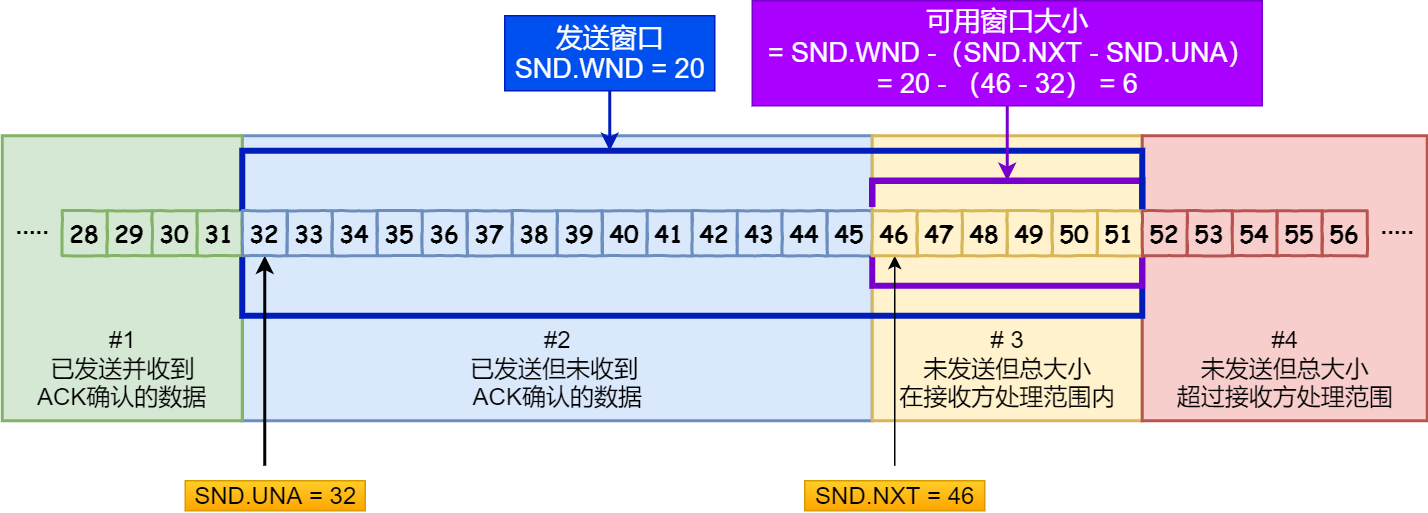
###### 背景：

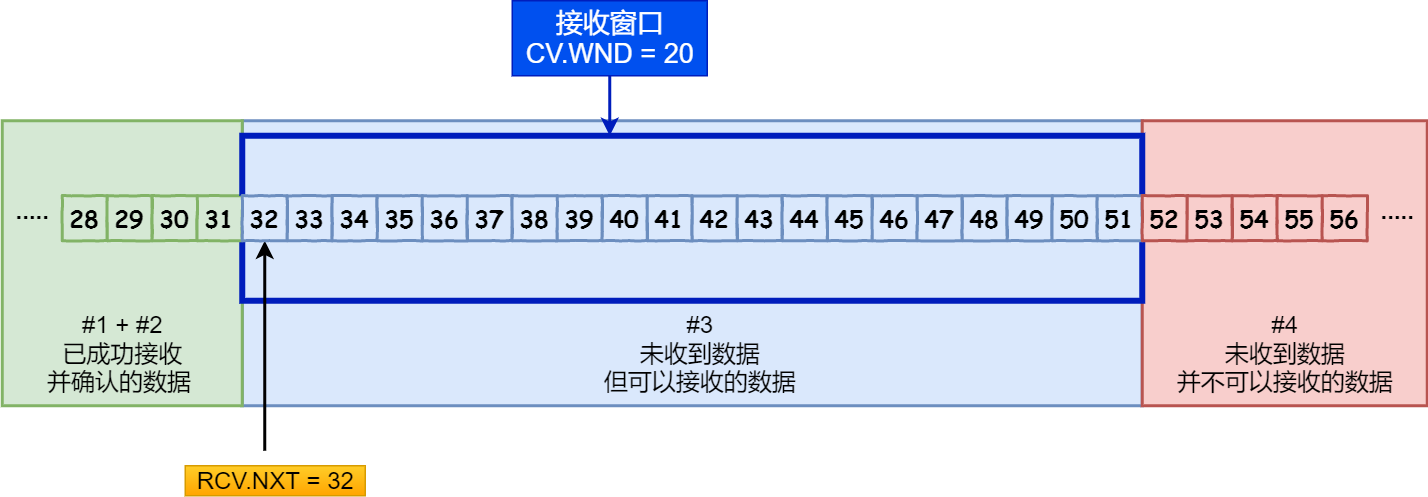
如果没有滑动窗口，那么发送接受必须是串行，你发一个，我接受一个，不能是异步的。



为了解决传输中存在的性能问题，引入了滑动窗口，，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。**

**窗口分为两种，发送窗口和接收窗口：**





## 流量控制：

###### 背景：

接收端接收能力有限，所以发送方不能一直发送

###### 大致原理：

发送方和接收方有固定滑动窗口的大小，发送方只有收到ack之后才能 可用窗口才能右移。

## 拥塞控制：

需要引入拥塞窗口

###### 背景：

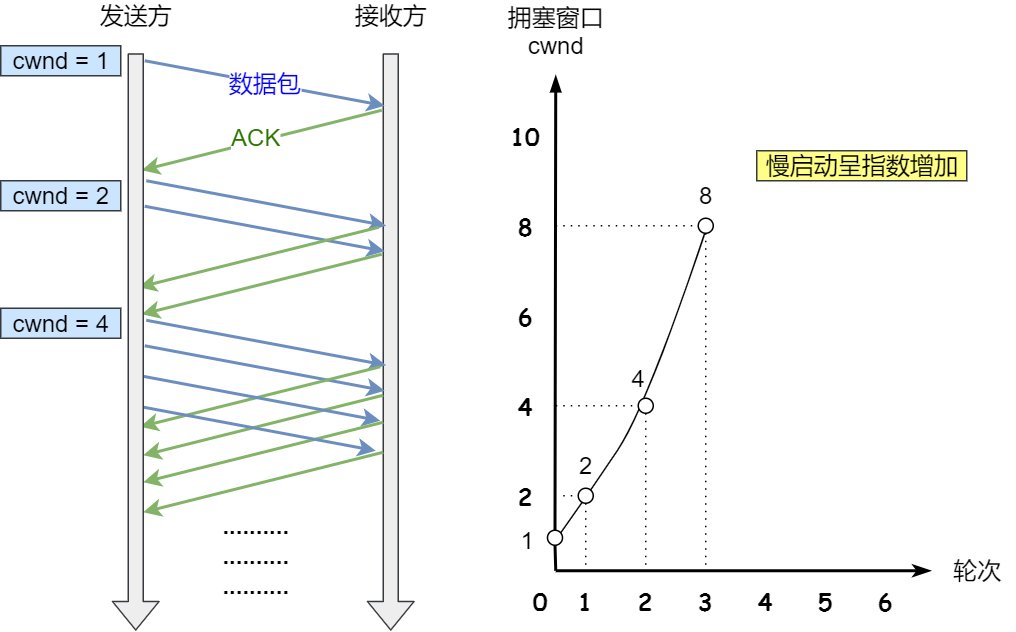
有时候网络可能不好，发送的包就容易丢失。

判断网络好不好是根据是否发生了超时重传，如果发生了，就代表了有拥塞。

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。

###### 拥塞控制算法：

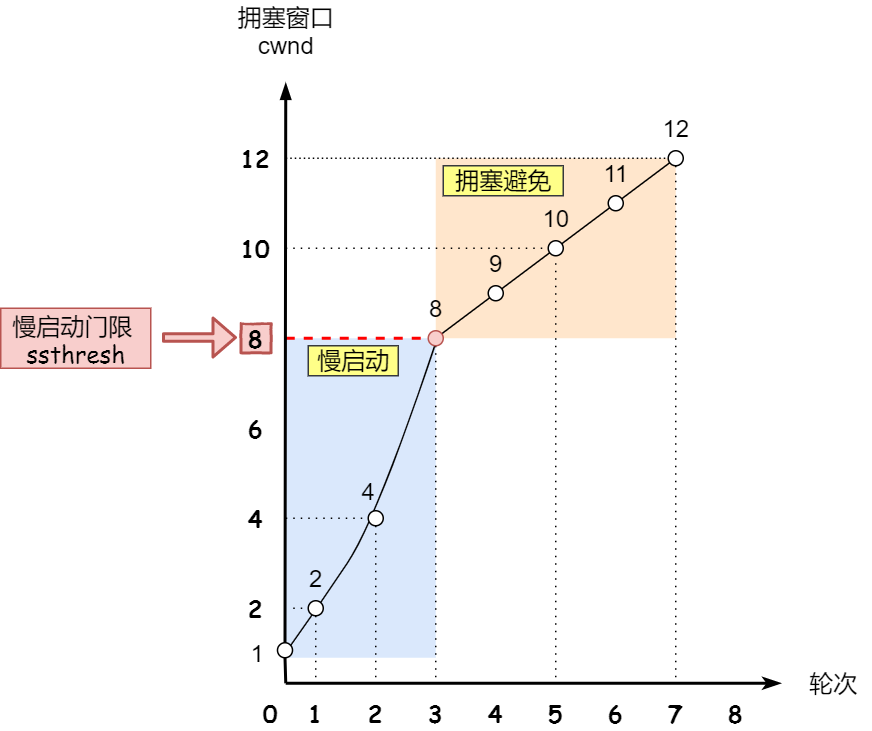
1:慢启动



刚建立连接，窗口大小为1，之后每收到一个ack之后，窗口就\*2，一直等到达到满启动阀值，然后使用拥塞避免算法。

###### 拥塞避免算法：

前面说到，当拥塞窗口cwnd超过慢启动门限ssthresh之后，就会进入拥塞避免算法。拥塞避免算法每收到一个ack，就增加 1/cwnd



# 当网络拥塞发生时，也就是数据包发生了重传，会涉及到

1:超时重传

2:快速重传

当发生了超时重传（严重）：

ssthresh = cwnd/2

cwnd = 1

当发生了快速重传（一般）：

sshresh = cwnd/2

cwnd = sshresh + 3

然后进入快速恢复算法：

收到一个 重复ACK，那么cwnd+1

如果收到新的ACK，那么cwnd = sshresh，进入拥塞避免阶段

# time\_wait/close\_wait过多问题：

这里说的问题都是指**服务端的**time-wait/close-wait过多的问题

### time\_wait:

###### 问题原因：

服务端主动关闭连接，进入到FAIT\_WAIT2，当client端发送FIN后，服务端收到，并且发送了ACK之后，就会变成TIME\_WAIT，然后等待2个TTL时间，结束；一般服务端的请求处理非常快，然后并发高，并且是服务端主动关闭会有这样的问题，我平时开发都是client主动关闭，没有遇到过类似这种的问题。

###### 导致的问题：

会导致连接过多，服务端不能处理更多的请求

###### 解决办法：

time\_wait2个ttl的存在的意义就是，为了让最后发送FIN的一侧能够正常释放连接。在服务端主动释放的场景，就是让client侧能够正常释放连接。办法只能有两个：

1:当渣男，不配置等待时间，发送完ACK直接关闭连接，不管client有没有收到，这样就可能会导致client端内存泄漏。

2:复用连接，解决高并发问题；长链接或者 连接复用。

注意：不能通过修改timeout的时间，如果小于2个ttl时间，那么就等于没有啊。。。。。还不如直接干掉

3:修改进程能打开的文件数量（治标不治本）

### close\_wait

###### 问题原因：

客户端主动关闭，但是服务端一只没有处理请求，所以没有发送完data，就没有发送fin字段给client，导致server端是close\_wait状态。一般是业务处理太是以下几个原因导致的：

1:比如忘记释放数据库连接，导致其他线程访问数据库一直阻塞，或则是业务处理流程本身过长，处理时间慢。

2:发送完数据，忘记执行socket的close。

###### 导致的问题：

服务端报错，too many open files，导致不能处理新的连接

###### 解决办法：

分析出问题的点

1:如果是业务处理本身比较久，优化业务

2:如果是编码问题，比如忘记socket.close了那么进行close

3:如果是数据库查询比较慢，那么查看是不是语句不合理，或者并发量太高

4:修改内核参数，增大进程打开的文件数（治标不治本）

# TCP/IP协议其他层概揽

