

OSI 七层模型

OSI (Open System Interconnection) 是理想化的模型，将网络进行分层，其目的是将复杂的流程简单化，从而实现分而治之。（专人干专事）

一.网络分层的含义？

下层是为了上层提供服务的。

- 应用层：用户最终使用的接口
- 表示层：数据的表示、安全、压缩
- 会话层：建立和管理会话的
- 传输层：（主要提供安全及数据完整性保障）网络层不可靠，保证可靠的传输
- 网络层：（主要关心的是寻址），进行逻辑寻址，定位到对方，找到最短的路
- 数据链路层：（主要关心两个设备之间传递数据），建立逻辑链接，将数据组合成数据帧进行传递（差错检测，可靠传输）
- 物理层：（核心是传输数据比特流），不关心具体的传输媒体（双绞线、光纤、同轴电缆、无线...）

举例：写给女朋友信的过程

- 1.应用层：你心里有很多想对女朋友说的话。这个就是应用层中的数据
- 2.表示层：将你想说的话进行整合，有调理的表示出来
- 3.会话层：我希望我的信只能我的女朋看到别人不行（非女朋友偷看者死）
- 以上这三个就是我们完整信的内容。
- 4.传输层：我自己不好意思亲手交给她，找个快递来。告诉他我家504她家301，你发吧~
- 5.网络层：快递说这不是开玩笑吗？你得给我个能找到他的地址 xxx 省 xxx 市 xxx 街道 xxx 小区。还得添上你的地址，原地址和目标地址。
- 6.数据链路层：信件到了快递总部，会进行分类增加标识，快递需要中转，先找到第一个中转站发过去，之后根据目的地地址依次进行中转发送。
- 7.物理层：通过飞机、卡车将信邮寄到过去。

信件邮寄到目的地后，邮局会分配到对应的小区，找到对应的门牌号，我的女朋友就会拿到对应的信件了。



二.地址

通信是通过 `ip` 地址查找对应的 `mac` 来进行通信的。`IP地址` 是可变的（类似我们收件地址）`MAC地址` 是不可变的。

1. IP 地址

IPv4 网际协议版本4，地址由 32 位二进制数值组成 例如：192.168.1.1，大概42亿个

IPv6 网际协议版本6，地址由 8个16位块的128位组成。 例如：2408:8207:788b:2370:9530:b5e7:9c53:ff87
大约 3.4×10^{38}

2. MAC 地址

设备通信都是由内部的网卡设备来进行的，每个网卡都有自己的mac地址（原则上唯一）

三.物理设备

1.物理层

- 中继器：双绞线最大传输距离 100M，中继器可以延长网络传输的距离，对衰减的信号有放大在生的功能。
- 集线器：多口的中继器，目的是将网络上的所有设备连接在一起，不会过滤数据，也不知道将收到的数据发给谁。（采用的方式就是广播给每个人）

可以实现局域网的通信，但是会有安全问题，还会造成不必要的流量浪费。 傻，你就不能记住来过的人嘛？每次都发送？

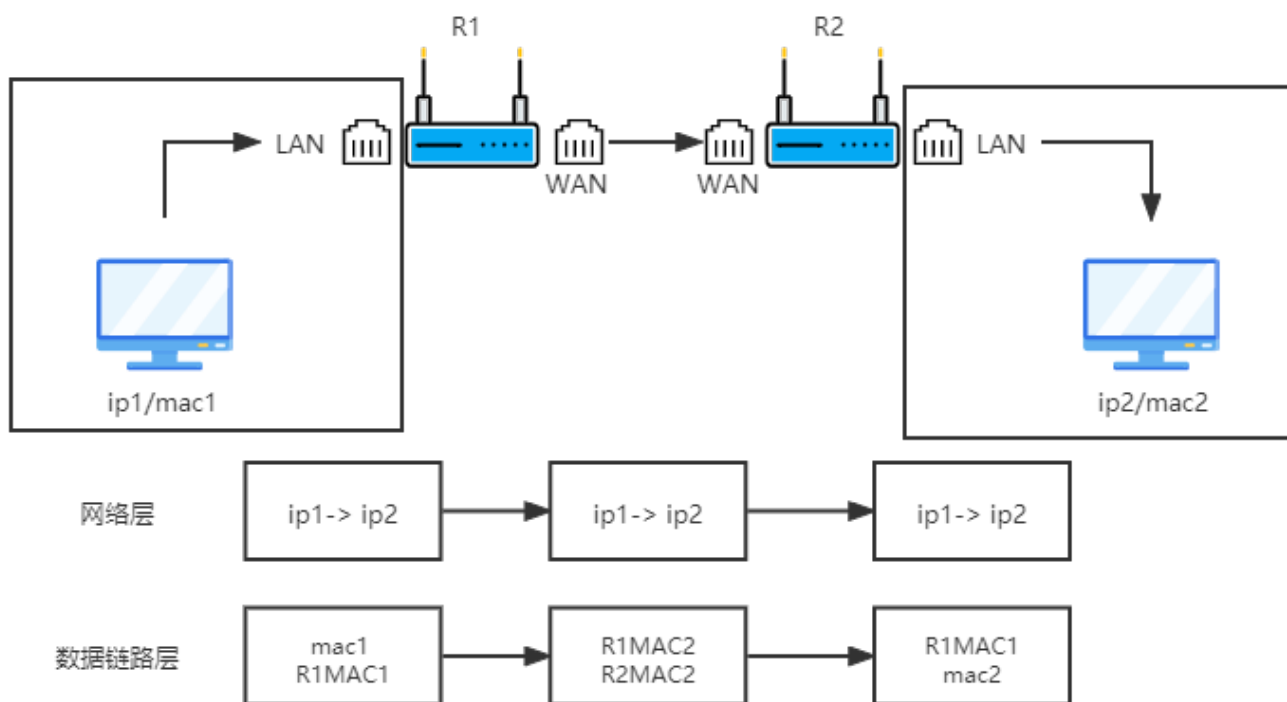
2.数据链路层

- 交换机：交换机可以识别已经连接设备的物理地址（MAC地址）。可以将数据传递到相应的端口上

3.网络层

- 路由器：检测数据的 ip 地址是否属于自己网络，如果不是会发送到另一个网络。没有 wan 口的路由器可以看成交换机。路由器一般充当网关，路由器会将本地 IP 地址进行NAT

网关：两个子网之间不可以直接通信，需要通过网关进行转发



四. TCP/IP 参考模型

Transmission Control Protocol/Internet Protocol，传输控制协议/网际协议。TCP/IP 协议实际上是一系列网络通信协议的统称，最核心的两个协议是 TCP 和 IP

1.什么是协议?

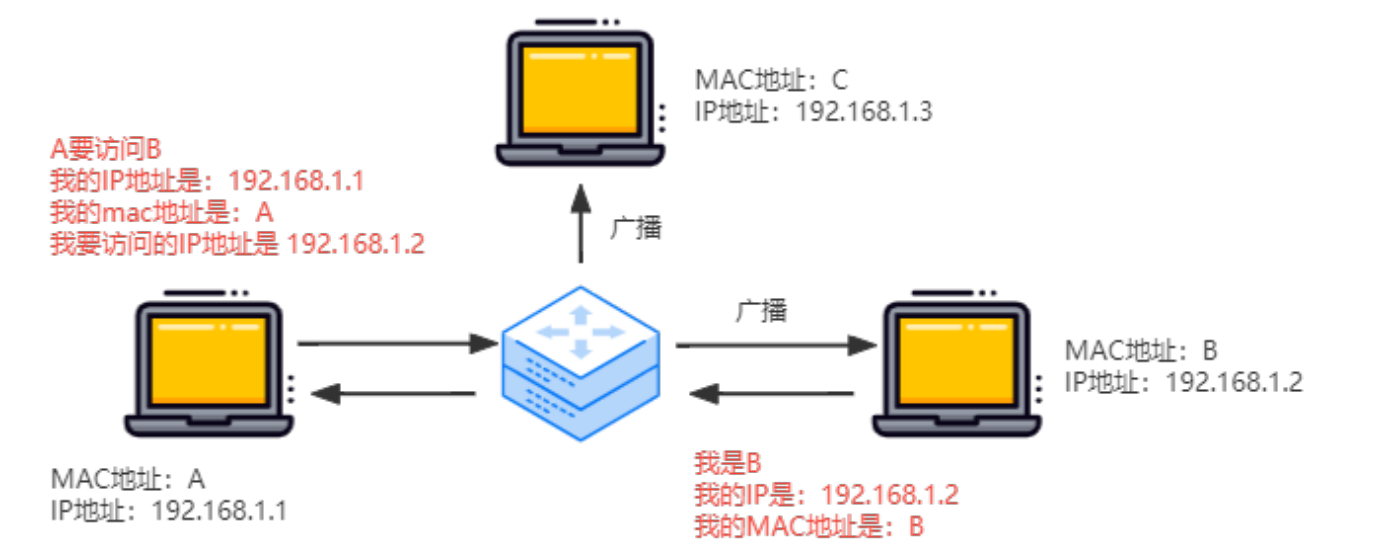
协议就是约定和规范。

数据链路层、物理层：物理设备 (在五层模型中能称之为协议的都在三层以上)

- 网络层：
 - IP 协议：寻址通过路由器查找，将消息发送给对方路由器，通过 ARP 协议,发送自己的mac地址
 - ARP 协议：Address Resolution Protocol 从 ip 地址获取 mac地址 （局域网）
- 传输层
 - TCP、UDP
- 应用层：
 - HTTP、DNS、FTP、TFTP、SMTP、DHCP

2. ARP 协议

根据目的 IP 地址，解析目的 mac 地址



ARP 缓存表		交换机MAC地址表	
Internet 地址	物理地址	端口号	物理地址
192.168.1.2	B	1	A
		2	B
		3	C

有了源mac地址和目标mac地址，就可以传输数据包了

3. DHCP 协议

通过 DHCP 自动获取网络配置信息（动态主机配置协议Dynamic Host Configuration Protocol）我们无需自己手动配置 IP

4. DNS 协议

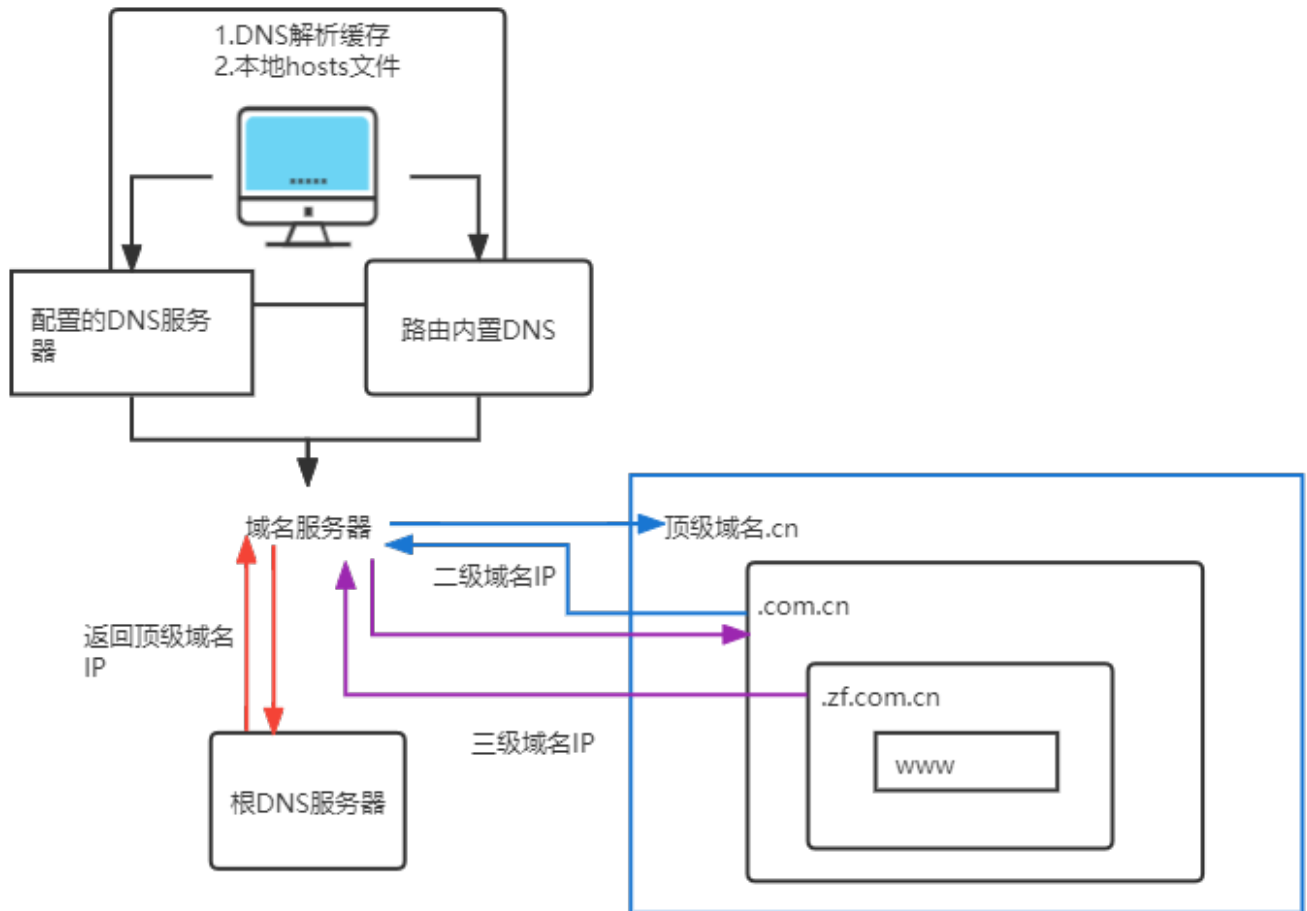
DNS 是Domain Name System的缩写，DNS 服务器进行域名和与之对应的 IP 地址转换的服务器

- 顶级域名 .com、
- 二级域名 .com.cn、三级域名 www.zf.com.cn，有多少个点就是几级域名

访问过程：我们访问 zf.com.cn

- 操作系统里会对 DNS 解析结果做缓存，如果缓存中有直接返回 IP 地址

- 查找 `C:\WINDOWS\system32\drivers\etc\hosts` 如果有直接返回 IP 地址
- 通过 **DNS 服务器** 查找离自己最近的根服务器，通过根服务器找到 `.cn` 服务器，将 ip 返回给 DNS 服务器
- DNS 服务器会继续像此 ip 发送请求，去查找对应 `.cn` 下 `.com` 对应的 ip...
- 获取最终的 ip 地址。缓存到 DNS 服务器上



DNS 服务器会对 ip 及 域名 进行缓存

五. TCP 和 UDP

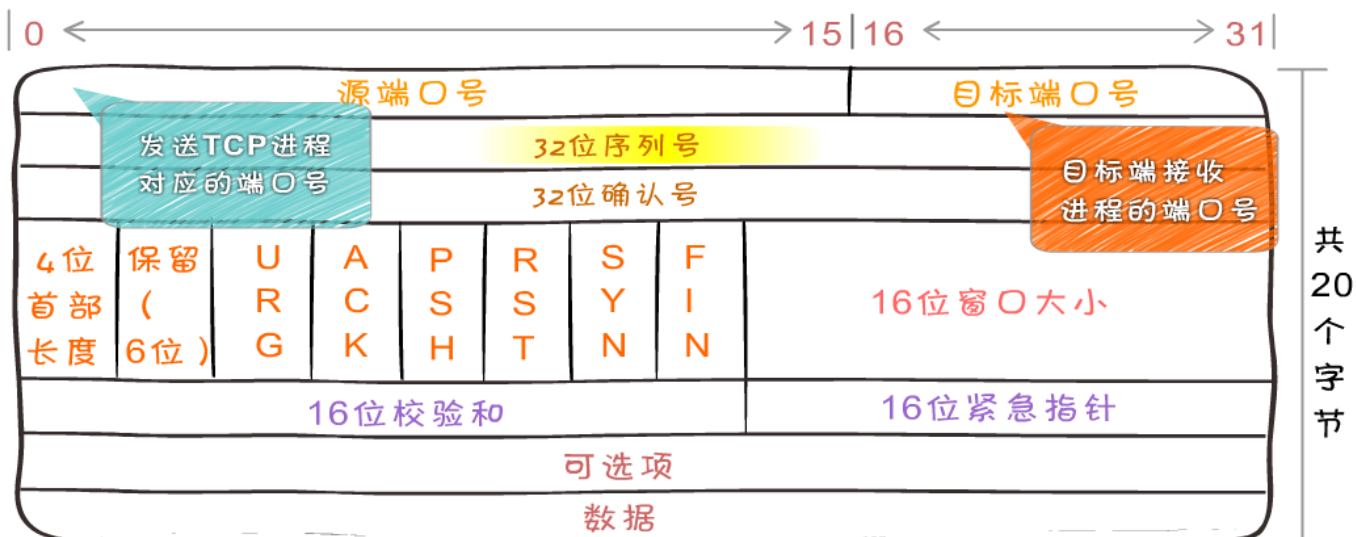
两个协议都在传输层，我们常说 TCP 是面向连接的而 UDP 是面向无连接的。

- UDP 发出请求后，不考虑对方是否能接收到、内容是否完整、顺序是否正确。收到数据后也不会进行通知。
- 首部结构简单，在数据传输时能实现最小的开销

1. TCP

tcp 传输控制协议 Transimission Control Protocol 可靠、面向连接的协议,传输效率低 (在不可靠的 IP 层上建立可靠的传输层)。TCP提供全双工服务，即数据可在同一时间双向传播。

1) TCP数据格式



- 源端口号、目标端口号，指代的是发送方随机端口，目标端对应的端口
- 序列号：32位序列号是用于对数据包进行标记，方便重组
- 确认序列号：期望发送方下一个发送的数据的编号
- 4位首部长度：单位是字节，4位最大能表示15，所以头部长度最大为60
- URG:紧急新号、ACK:确认信号、PSH:应该从TCP缓冲区读走数据、RST: 断开重新连接、SYN:建立连接、FIN:表示要断开
- 窗口大小：当网络通畅时将这个窗口值变大加快传输速度，当网络不稳定时减少这个值。在TCP中起到流量控制作用。
- 校验和：用来做差错控制，看传输的报文段是否损坏
- 紧急指针：用来发送紧急数据使用

TCP 对数据进行分段打包传输，对每个数据包编号控制顺序。

2. TCP 抓包

client.js

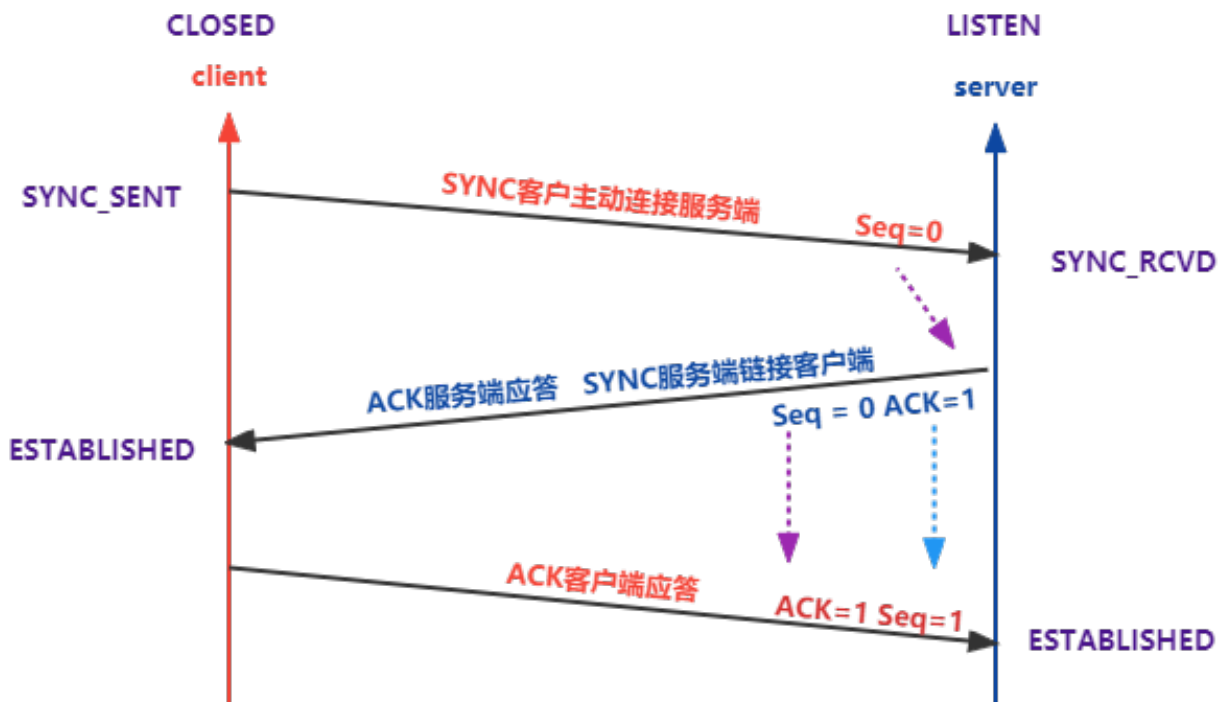
```
const net = require('net');
const socket = new net.Socket();
// 连接8080端口
socket.connect(8080, 'localhost');
// 连接成功后给服务端发送消息
socket.on('connect', function(data) {
  socket.write('hello'); // 浏览器和客户端说 hello
  socket.end()
});
socket.on('data', function(data) {
  console.log(data.toString())
})
socket.on('error', function(error) {
```

```
console.log(error);  
});
```

server.js

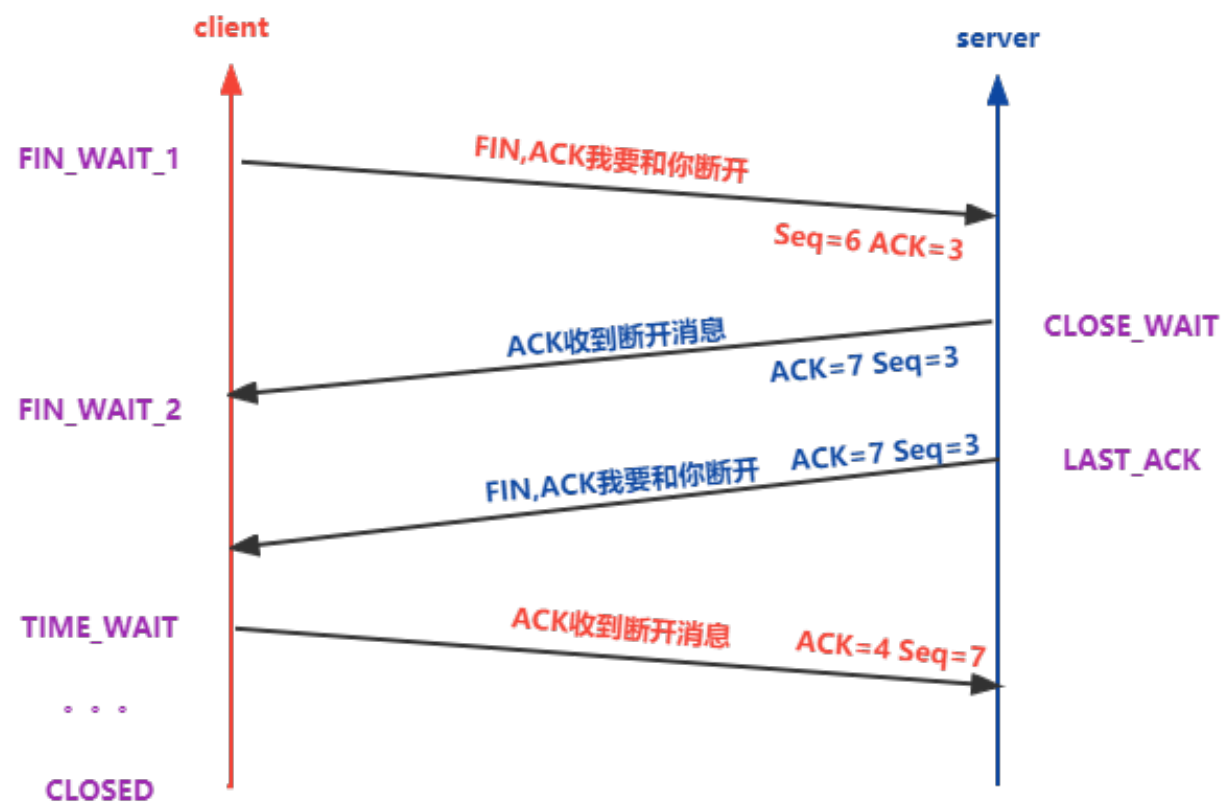
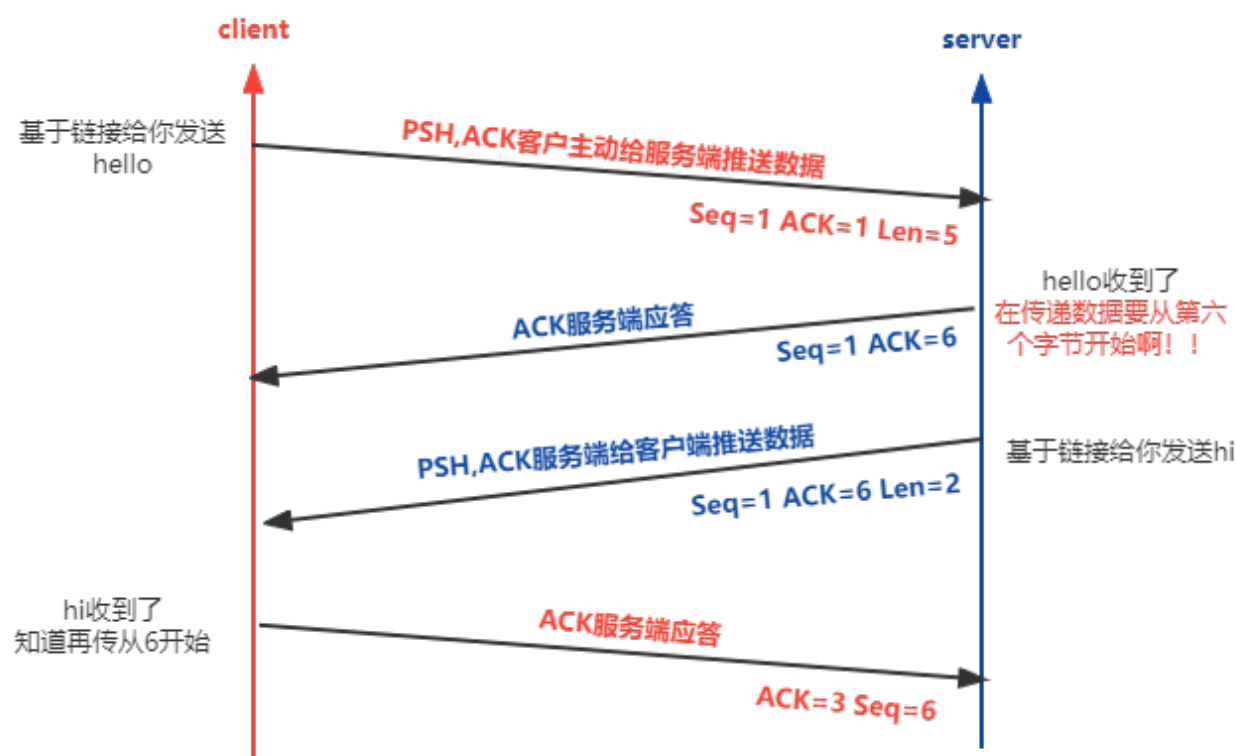
```
const net = require('net');  
const server = net.createServer(function(socket){  
  socket.on('data',function (data) { // 客户端和服务端  
    socket.write('hi'); // 服务端和客户端说 hi  
  });  
  socket.on('end',function () {  
    console.log('客户端关闭')  
  })  
})  
server.on('error',function(err){  
  console.log(err);  
})  
server.listen(8080); // 监听8080端口
```

1) 建立连接



- 1) 我能主动给你打电话吗? 2) 当然可以啊! 那我也能给你打电话吗?
- 3) 可以的呢, 建立连接成功!

4) 数据传输



- 四次挥手

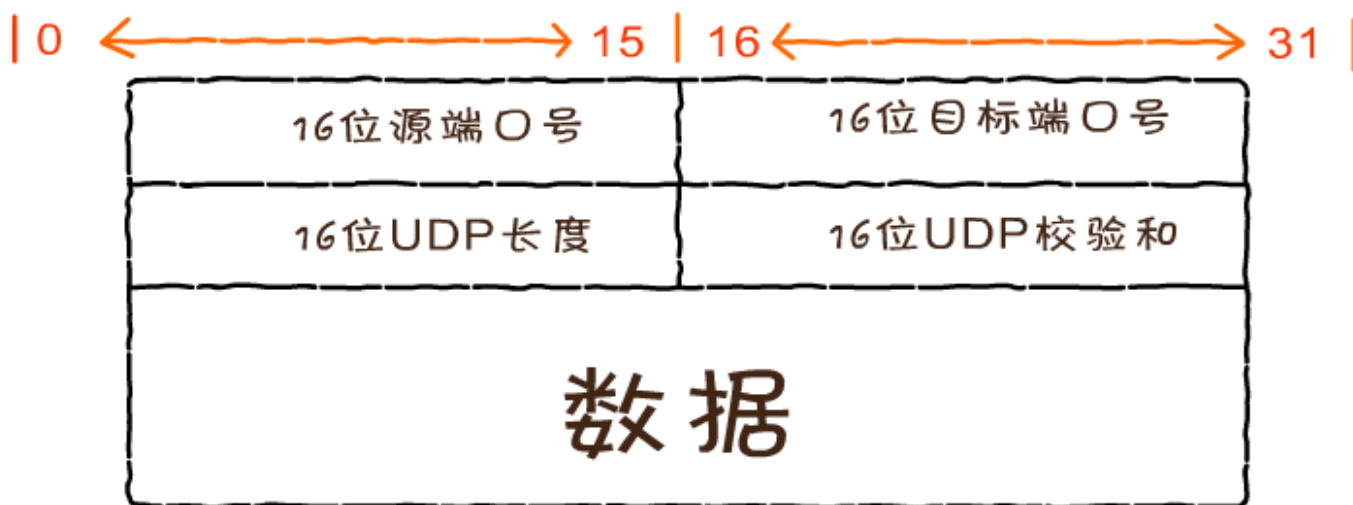
- 1) 我们分手吧 2) 收到分手的信息
- 3) 好吧，分就分吧 4) 行，那就到这里了

为了防止最终的 `ACK` 丢失，发送 `ACK` 后需要等待一段时间，因为如果丢包服务端需要重新发送 `FIN` 包，如果客户端已经 `closed`，那么服务端会将结果解析成错误。 从而在高并发非长连接的场景下会有大量端口被占用。

3. UDP

`udp` 用户数据报协议 `User Datagram Protocol`，是一个无连接、不保证可靠性的传输层协议。你让我发什么就发什么！

- 使用场景：`DHCP` 协议、`DNS` 协议、`QUIC` 协议等 (处理速度快，可以丢包的情况)



4. UDP 抓包

`server.js`

```
var dgram = require("dgram");
var socket = dgram.createSocket("udp4");
socket.on("message", function (msg, rinfo) {
  console.log(msg.toString());
  console.log(rinfo);
  socket.send(msg, 0, msg.length, rinfo.port, rinfo.address);
});
socket.bind(41234, "localhost");
```

`client.js`

```
var dgram = require('dgram');
var socket = dgram.createSocket('udp4');
socket.on('message', function(msg, rinfo){
    console.log(msg.toString());
    console.log(rinfo);
});
socket.send(Buffer.from('helloworld'), 0, 5, 41234, 'localhost', function(err, bytes){
    console.log('发送了个%d字节', bytes);
});
socket.on('error', function(err){
    console.error(err);
});
```

`udp.dstport == 41234`

5. 滑动窗口

发送连续的数据

- 滑动窗口：TCP是全双工的，所以发送端有发送缓存区；接收端有接收缓存区，要发送的数据都放到发送者的缓存区，发送窗口（要被发送的数据）就是要发送缓存中的哪一部分
- 核心是流量控制：在建立连接时，接收端会告诉发送端自己的窗口大小（`rwnd`），每次接收端收到数据后都会再次确认（`rwnd`）大小，如果值为0，停止发送数据。（并发送窗口探测包，持续监测窗口大小）

6. 粘包

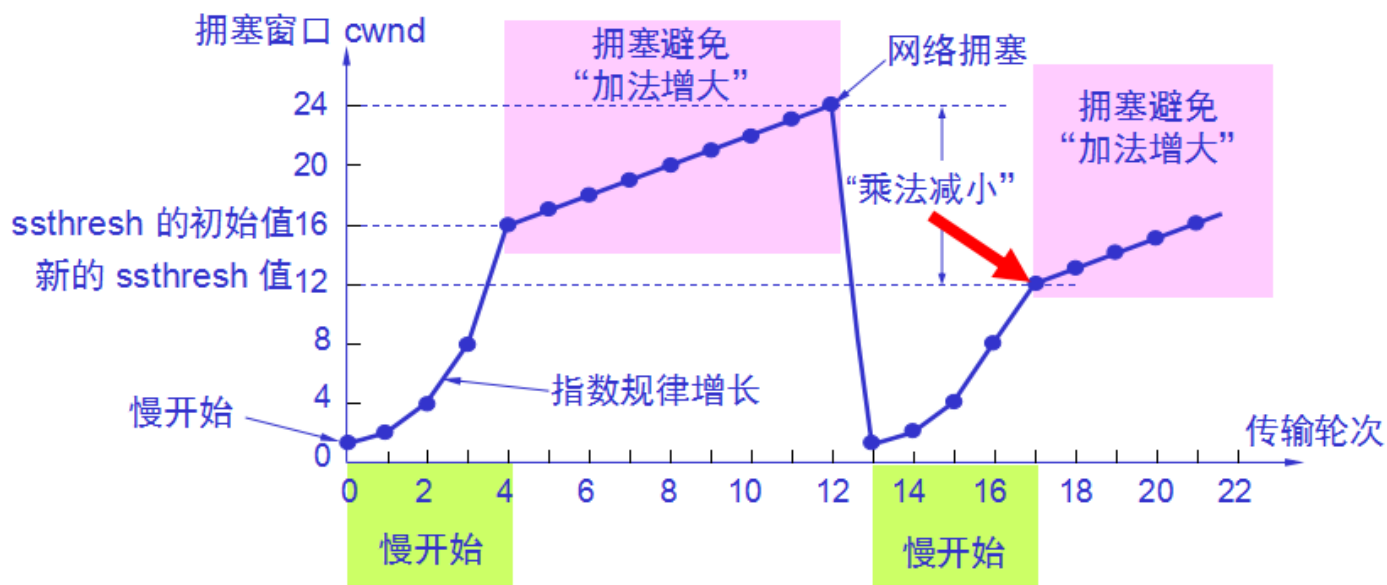
`Nagle` 算法的基本定义是任意时刻，最多只能有一个未被确认的小段（TCP内部控制）

`Cork`算法 当达到 `MSS` (Maximum Segment Size) 值时统一进行发送（此值就是帧的大小 - `ip` 头 - `tcp` 头 = 1460 个字节）理论值

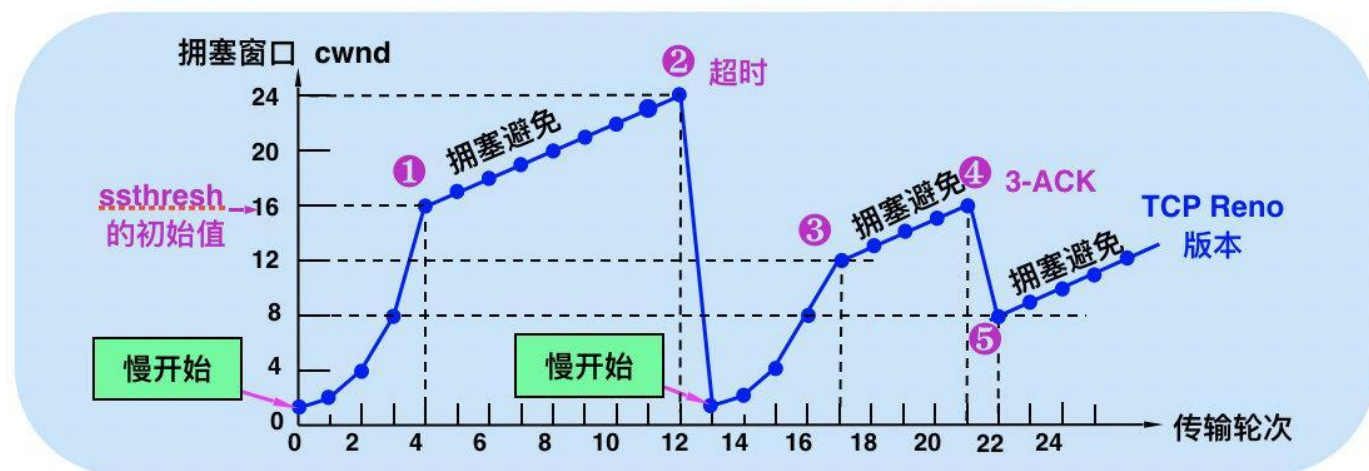
7. TCP 拥塞处理（队头阻塞，慢启动，短连接）

举例：假设接收方窗口大小是无限的，接收到数据后就能发送 `ACK` 包，那么传输数据主要是依赖于网络带宽，带宽的大小是有限的。

- TCP 维护一个拥塞窗口 `cwnd`（congestion window）变量，在传输过程正没有拥塞就将此值增大。如果出现拥塞（超时重传 `RTO`(Retransmission TimeOut)）就将窗口值减少。
- `cwnd < ssthresh` 使用慢开始算法
- `cwnd > ssthresh` 使用拥塞避免算法
- `ROT`时更新 `ssthresh` 值为当前窗口的一半，更新 `cwnd = 1`



- 传输轮次: RTT (Round-trip time), 从发送到确认信号的时间
- cwnd 控制发送窗口的大小。



快重传，可能在发送的过程中出现丢包情况。此时不要立即回退到慢开始阶段，而是对已经收到的报文重复确认，如果确认次数达到3此，则立即进行重传 快恢复算法(减少超时重传机制的出现)，降低重置 cwnd 的频率。

HTTP

一.HTTP 发展历程

1990年 HTTP/0.9 为了便于服务器和客户端处理，采用了“纯文本”格式，只运行使用GET请求。在响应请求之后会立即关闭连接。

1996年 HTTP/1.0 增强了 0.9 版本，引入了 HTTP Header（头部）的概念，传输的数据不再仅限于文本，可以解析图片音乐等，增加了响应状态码和 POST, HEAD 等请求方法。（内容协商）

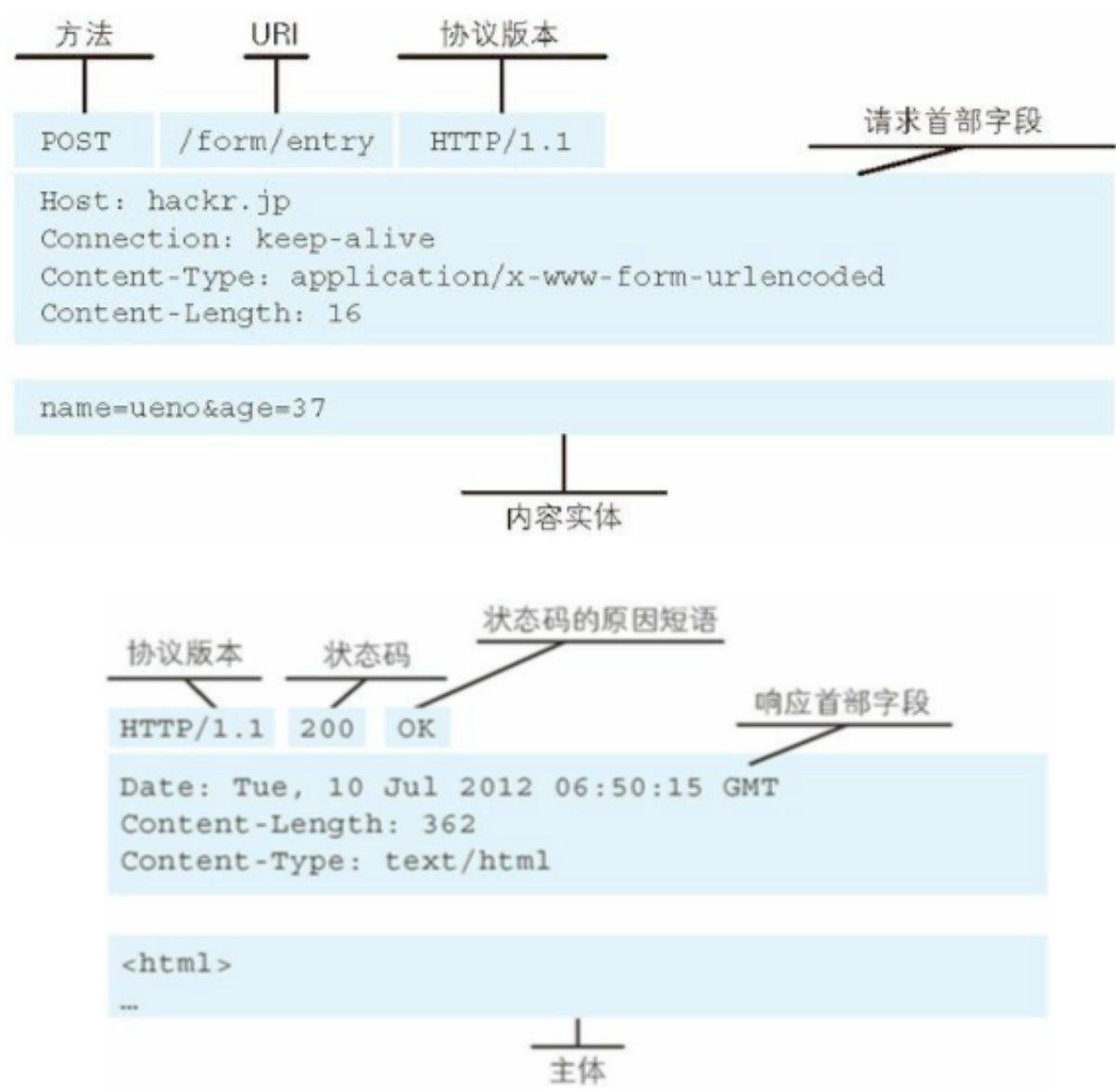
1999年广泛使用 HTTP/1.1，正式标准，允许持久连接，允许响应数据分块，增加了缓存管理和控制，增加了 PUT、DELETE 等新的方法。(问题 多个请求并发 http`队头阻塞的问题)

2015年 HTTP/2，使用 HPACK 算法压缩头部，减少数据传输量。允许服务器主动向客户端推送数据，二进制协议可发起多个请求，使用时需要对请求加密通信。

2018年 HTTP/3 基于 UDP 的 QUIC 协议。

二.HTTP/1.1

- HTTP/1.1 是可靠传输协议，基于 TCP/IP 协议；
- 采用应答模式，客户端主动发起请求，服务器被动回复请求；
- HTTP是无状态的每个请求都是互相独立
- HTTP 协议的请求报文和响应报文的结构基本相同，由三部分组成。



我始终认为，学好HTTP就是掌握HTTP中Header的使用

```
const http = require('http')
const server = http.createServer((req, res) => {
  res.end('hello')
})
server.listen(3000)
```

1.内容协商

客户端和服务端进行协商，返回对应的结果

客户端Header	服务端Header	
Accept	Content-Type	我发送给你的数据是什么类型
Accept-encoding	Content-Encoding	我发送给你的数据是用什么格式压缩 (gzip、deflate、br)
Accept-language		根据客户端支持的语言返回 (多语言)
Range	Content-Range	范围请求数据 206

2.长连接

TCP 的连接和关闭非常耗时间，所以我们可以复用 TCP 创建的连接。HTTP/1.1响应中默认会增加 Connection:keep-alive

3.管线化

如果值创建一条 TCP 连接来进行数据的收发，就会变成 "串行" 模式，如果某个请求过慢就会发生阻塞问题。**Head-of-line blocking** 管线化就是不用等待响应亦可直接发送下一个请求。这样就能够做到同时并行发送多个请求

同一个域名有限制，那么我就多开几个域名 域名分片

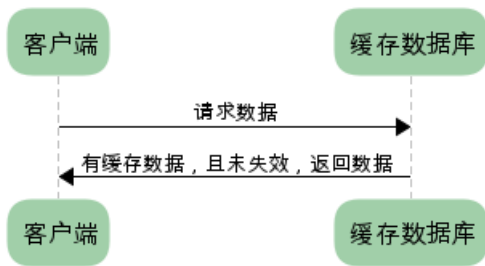
4.Cookie

Set-Cookie/Cookie用户第一次访问服务器的时候，服务器会写入身份标识，下次再请求的时候会携带 cookie。通过Cookie可以实现有状态的会话

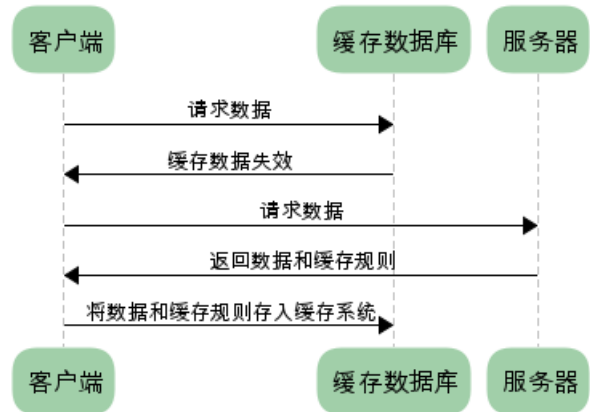
5.HTTP 缓存

强缓存 服务器会将数据和缓存规则一并返回，缓存规则信息包含在响应header中。 Cache-Control

强制缓存规则下，缓存命中



强制缓存规则下，缓存未命中



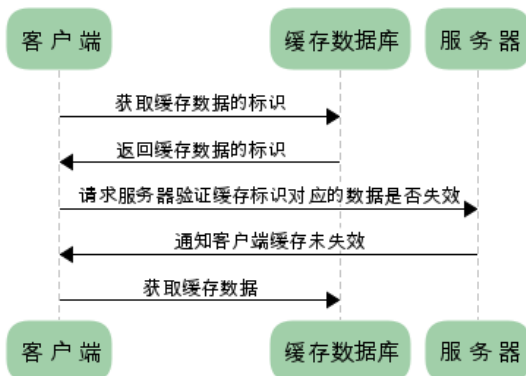
强制缓存存在有效期，缓存期内不会向服务端发送请求。超过时间后需要去服务端验证是否是最新版本。

对比缓存

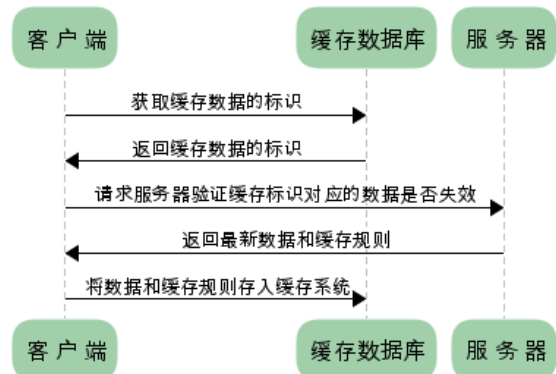
if-Modified-Since/if-None-Match (最后修改时间)、**Last-modified/Etag** (指纹)

- 最后修改时间是秒级的，一秒内修改多次无法监控
- 最后修改时间修改了，但是内容没有发生变化

对比规则下，缓存命中

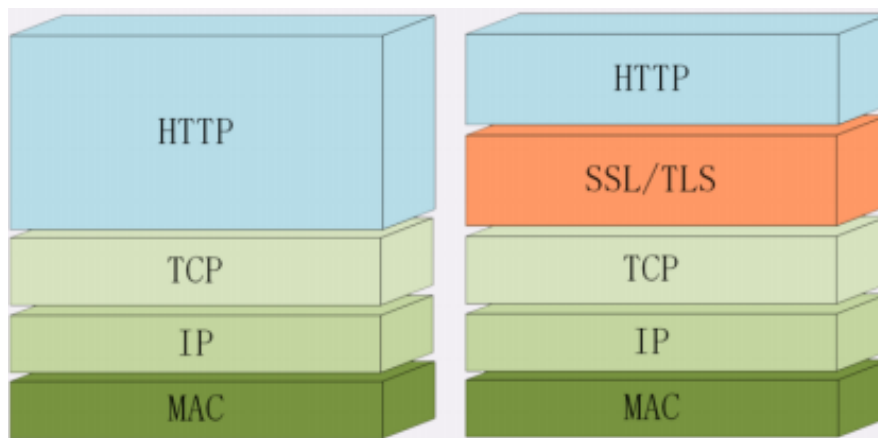


对比缓存规则下，缓存未命中



三 HTTPS (保证密文 防止篡改)

HTTP采用明文传输，中间人可以获取到明文数据（从而实现对数据的篡改）。这时候 HTTPS 就登场了！HTTPS 是什么呢？**HTTPS = HTTP + SSL/TLS**，SSL 安全套接层（Secure Sockets Layer）发展到 v3 时改名为 TLS 传输层安全(Transport Layer Security)，主要的目的是提供数据的完整性和保密性。



一.数据完整性

1.摘要算法

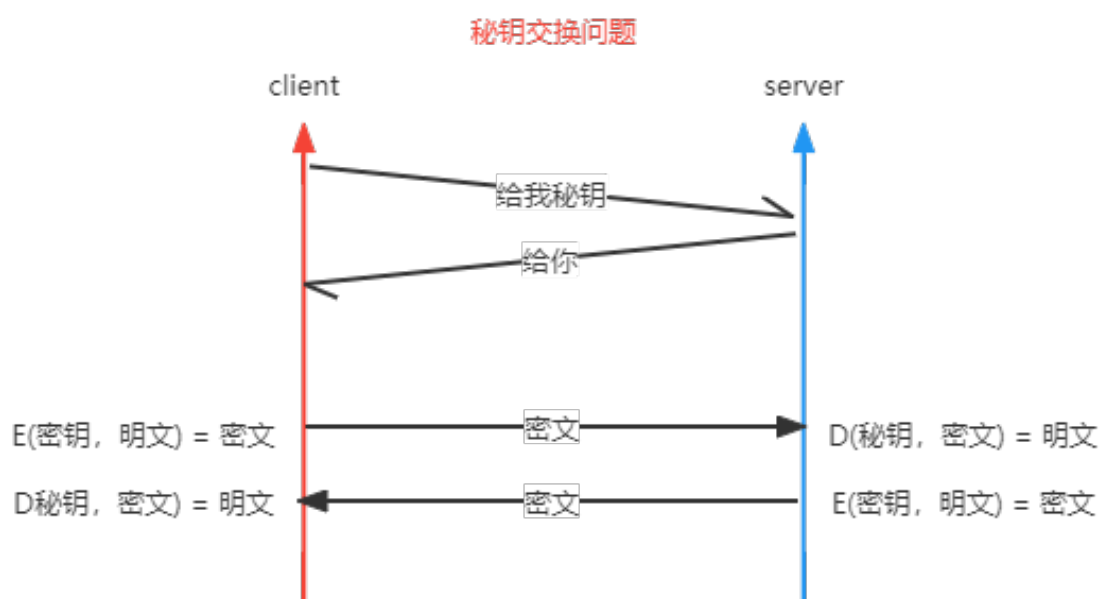
- 把任意长度的数据压缩成固定的长度
- 输入不同输出的结果发生剧烈的变化“雪崩效应”，相同的内容摘要后结果相同
- 不能从结果反推输入

我们可以在内容后面增加hash值进行传输，服务端收到后通过hash值来校验内容是否完整。数据是明文的显然不安全

二.数据加密

1.对称加密

加密和解密时使用的密钥都是同一个， 通信过程使用密钥加密后的密文传输。只有自己和网站才能解密。



M

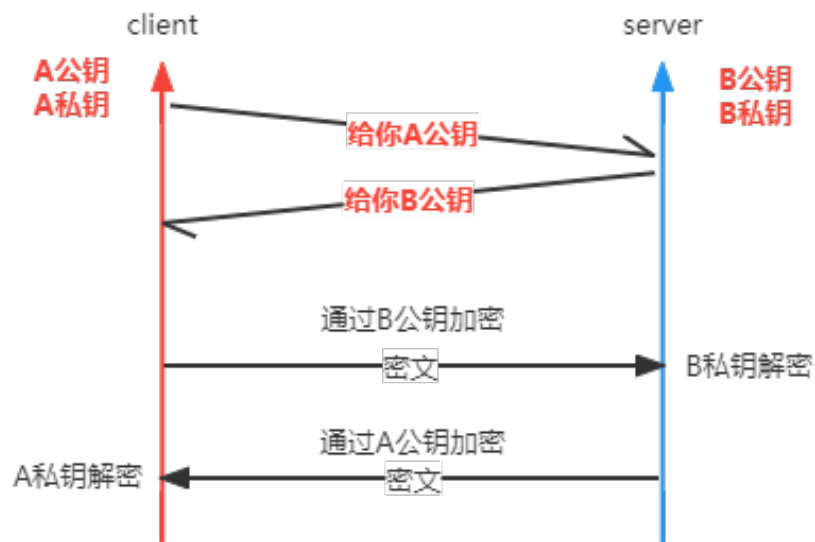
目前 AES (Advanced Encryption Standard) ChaCha20 为最常见的对称加密算法。

2.非对称加密

非对称加密可以解决“密钥交换”的问题。非对称加密有两个密钥，**公钥**、**私钥**，所以称之为非对称。公钥加密私钥解密。

并不能完全采用非对称加密算法，由于算法本身耗时远高于对称加密。

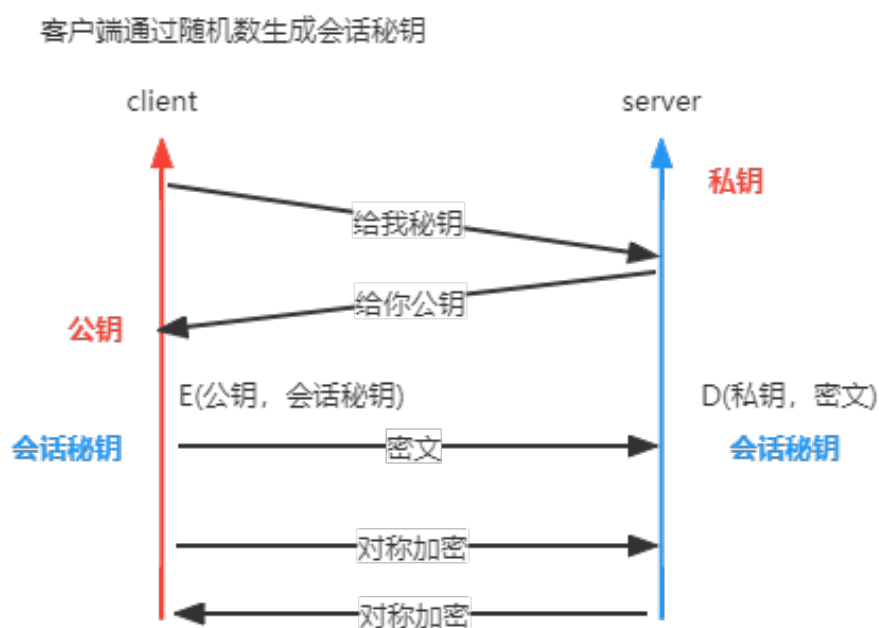
使用 `RSA`、`ECDHE` 算法解决密钥交换的问题



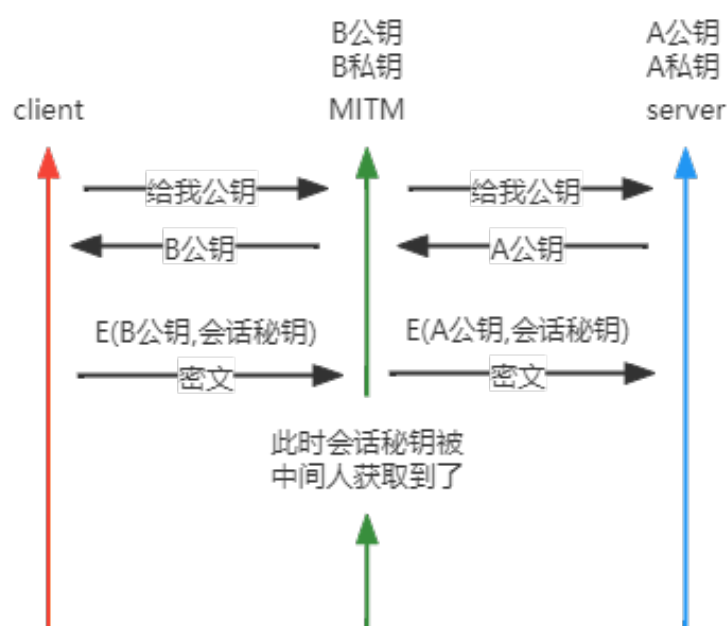
最常听到的非对称加密算法是 `RSA`、`ECC` (子算法 `ECDHE` 用于密钥交换, `ECDSA` 用于数字签名)(性能和安全略胜一筹) `HTTPS` 中目前广泛使用 `ECC` 。

3.混合加密

通信刚开始的时候使用非对称算法，交换密钥。在客户端生成**会话密钥**后传送给服务端，后续通信采用对称加密的方式

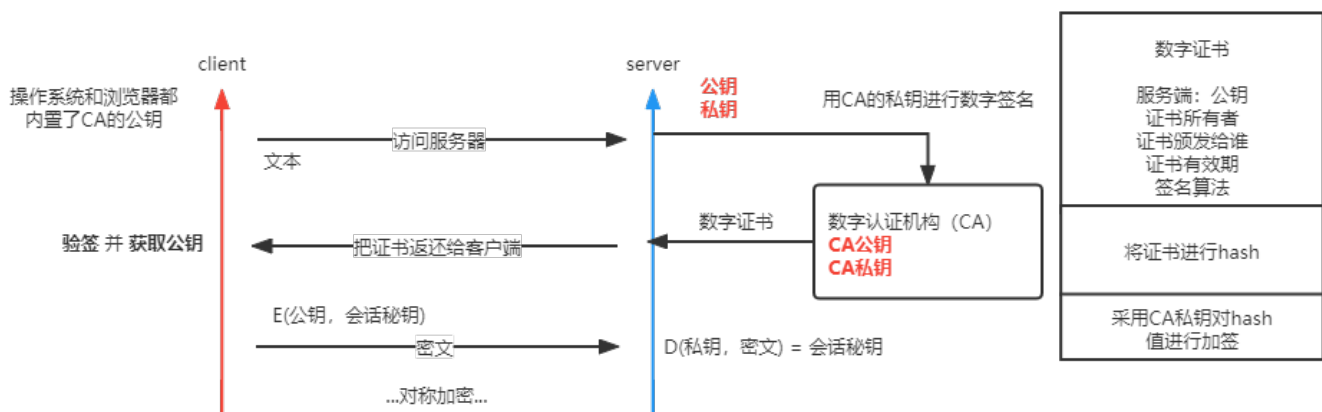


这里还并不安全，还涉及到中间人攻击。（指攻击者与通讯的两端分别创建独立的联系,并交换其所收到的数据）



4. 数字证书和CA

因为谁都可以发布公钥，所以我们需要验证对方身份。防止中间人攻击



客户端会判断有效期、颁发者、证书是否被修改及证书是否被吊销。每份签发证书都可以根据验证链查找到对应的根证书，操作系统、浏览器会在本地存储权威机构的根证书，利用本地根证书可以对对应机构签发证书完成来源验证。

- 加密：对传输的数据进行加密。
- 数据一致性：保证传输过程中数据不会被篡改。
- 身份认证：确定对方的真实身份。

三.HTTPS 过程

1.第一阶段

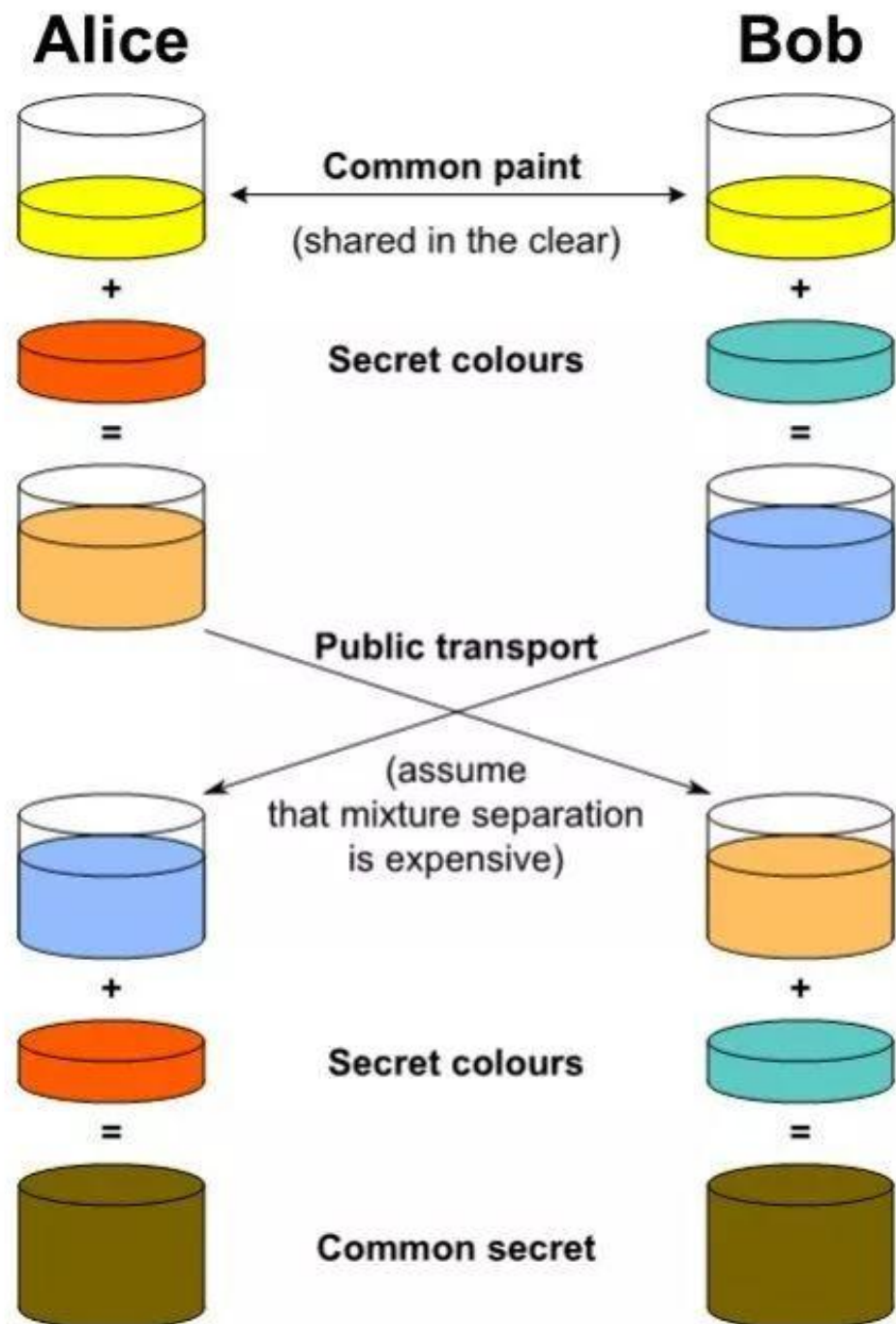
- 客户端会发送 `handshake Protocol: client hello`
 - Cipher Suites 密钥交换算法 + 签名算法 + 对称加密算法 + 摘要算法 套件列表
 - Random 客户端随机数
 - Version: TLS 1.2
- 服务端会发送 `handshake Protocol: Server Hello`
 - Version: TLS 1.2
 - Random 服务端随机数
 - Cipher Suites: 选择的套件

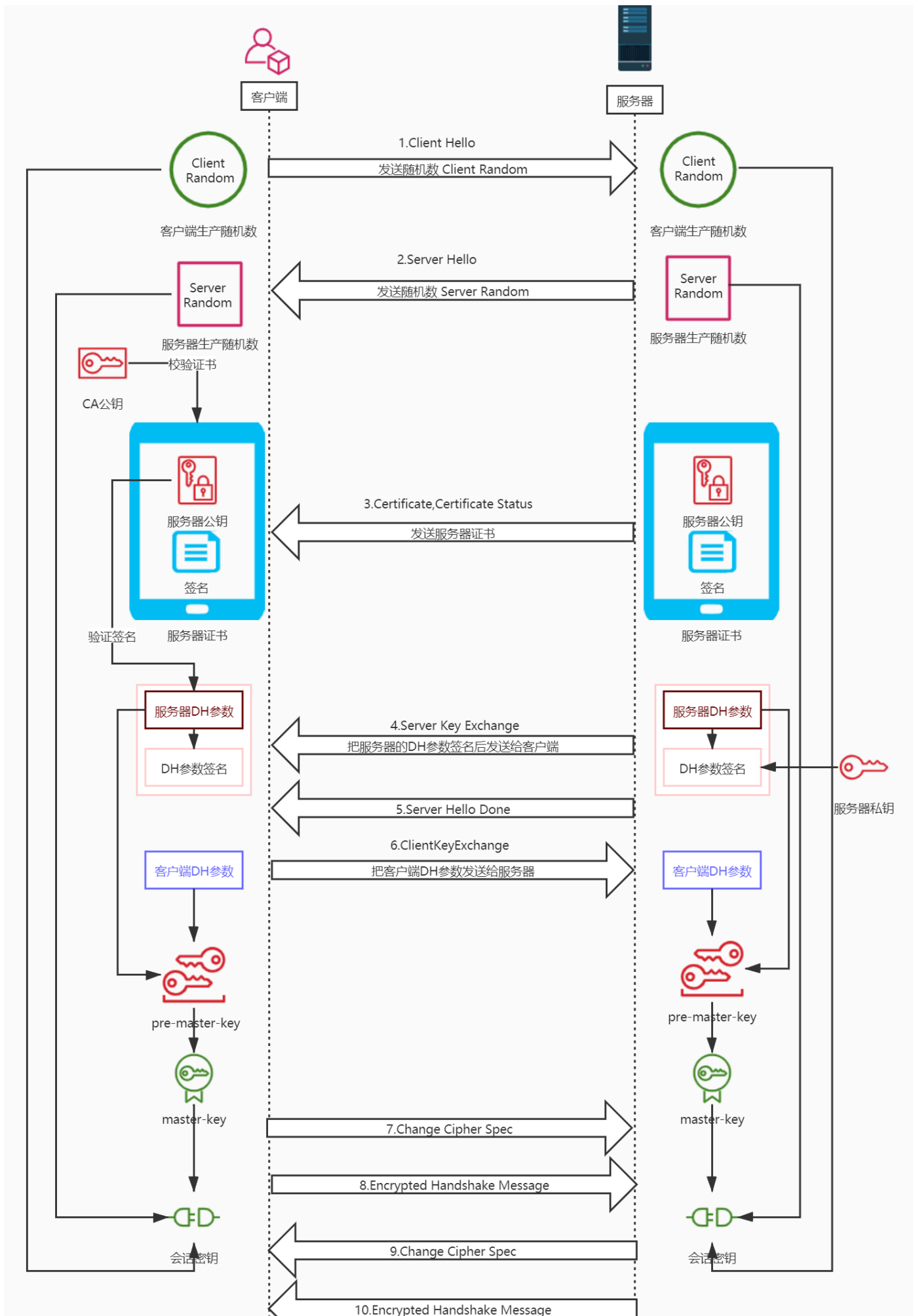
双方选择 TLS 版本，确定加密算法，生成两个随机数。

2.第二阶段

- 服务端发送证书 `certificate`
- 服务端发送 `ECDHE` 参数，服务端Hello完成
 - `Server Key Exchange`
 - `Server Hello Done`
- 客户端发送 `ECDHE` 参数，以后使用密钥进行通信吧，加密握手消息发送给对方
 - `Client Key Exchange`
 - `Change Cipher Spec`
 - `Encrypted HandleShake Message`
- 服务端发送会话凭证，以后使用密钥进行通信吧，加密握手消息发送给对方

- new Session Ticket
- Change Cipher Spec
- Encrypted Handshake Message





珠峰架构

SSL 协议组成

SSL 握手协议、SSL 密钥变化协议、SSL 警告协议、SSL 记录协议等

四.HTTP/2

HTTP/2主要的目标就是改进性能，兼容HTTP/1.1

1. One TCP connection

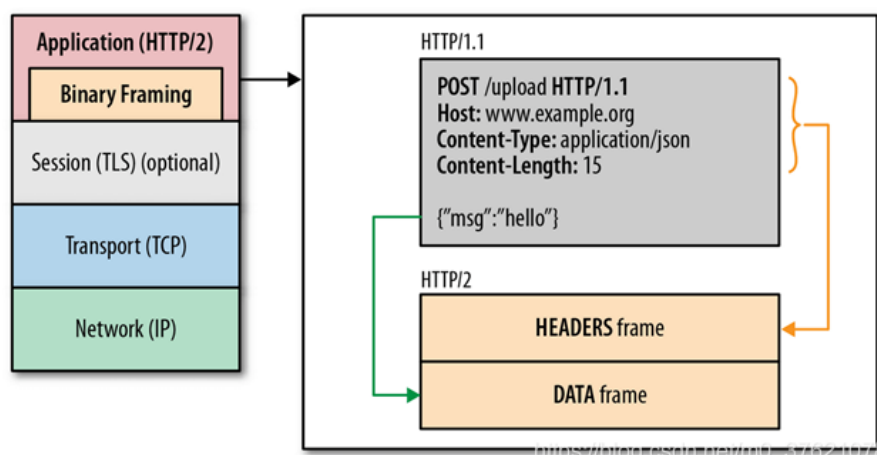
2. Request → Stream

- Streams are multiplexed
- Streams are prioritized

3. Binary framing layer

- Prioritization
- Flow control
- Server push

4. Header compression (HPACK)

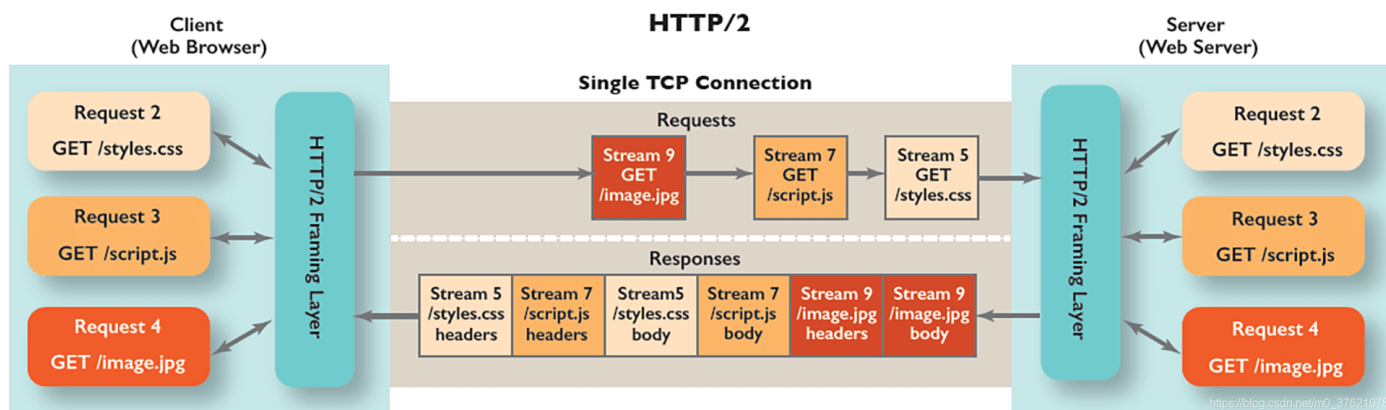
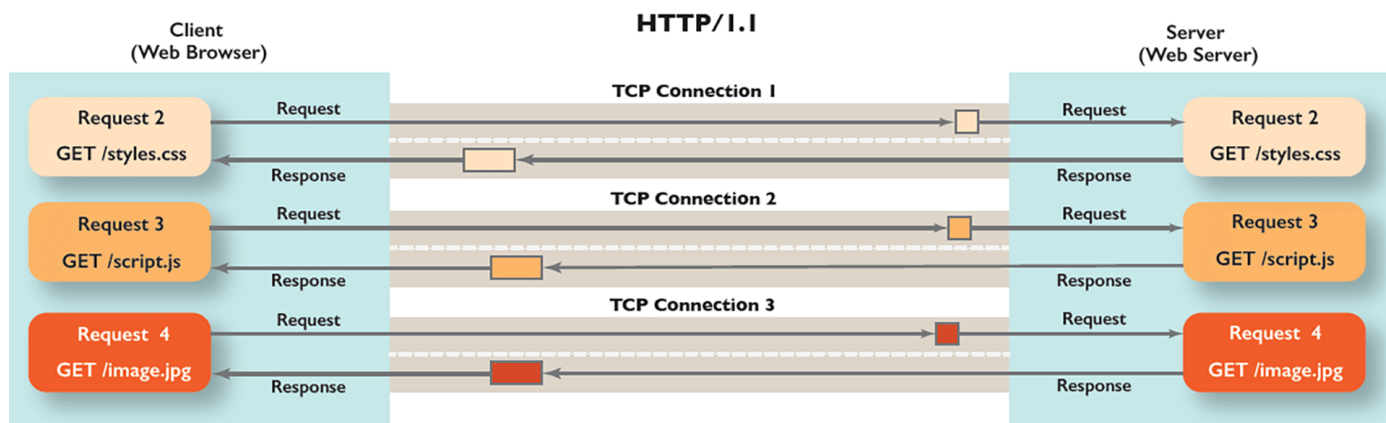


问题1：HTTP/1.1 中只优化了 body（gzip 压缩）并没有对头部进行处理

问题2：HTTP/1.1 问题在于当前请求未得到响应时，不能复用通道再次发送请求。需要开启新的TCP连接发送请求这就是我们所谓的管线化，但是后续的响应要遵循FIFO原则，如果第一个请求没有返回会被阻塞 HTTP队头阻塞问题。（最多并发的请求是6个）

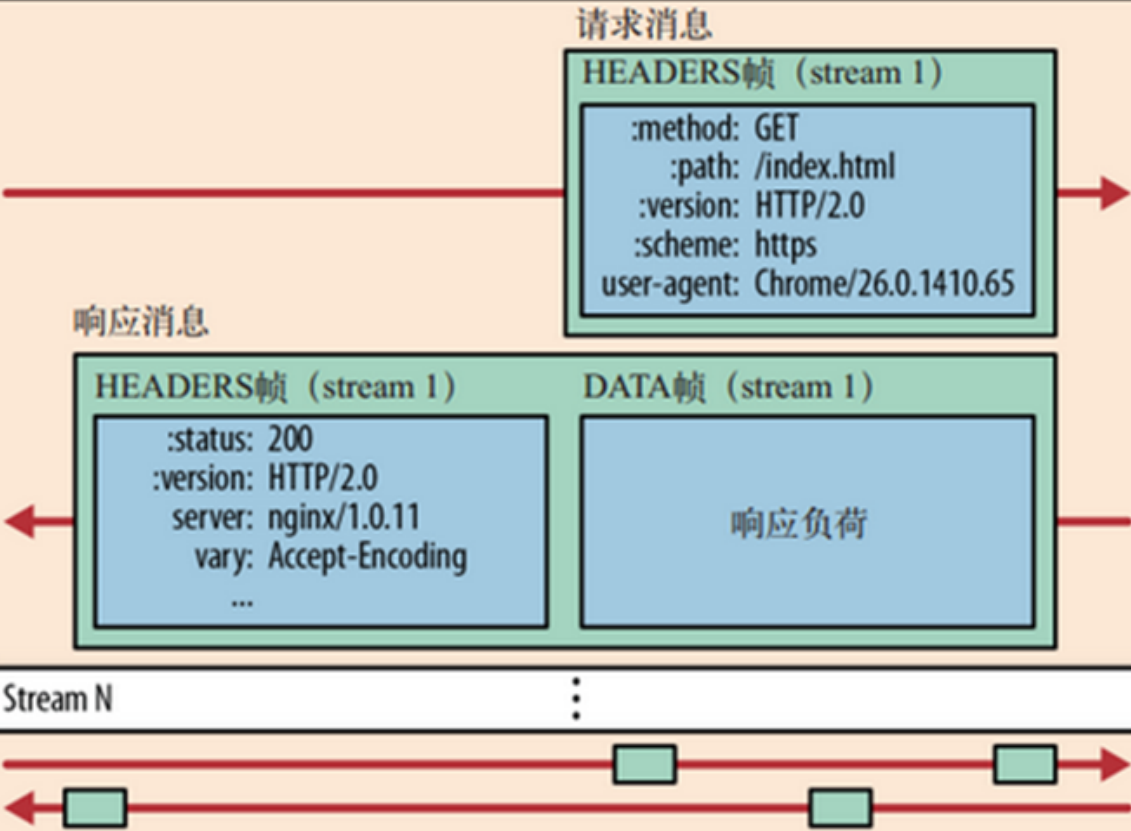
1.多路复用

在一条TCP链接上可以乱序收发请求和响应，多个请求和响应之间不再有顺序关系



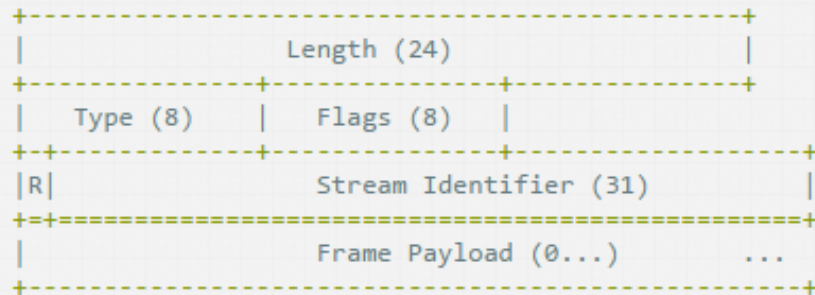
- 同域下采用一个TCP链接传输数据
- 采用二进制格式， HTTP/1.1采用的是纯文本需要处理空行、大小写等。文本的表现形式有多样性，二进制则只有0和1的组合不会有歧义而且体积更小。把原来的 Header+body 的方式转换为二进制帧。

Stream 1



- **HTTP/2** 虚拟了流的概念（有序的帧），给每帧分配一个唯一的流ID，这样数据可以通过 ID 按照顺序组合起来

帧的组成及大小



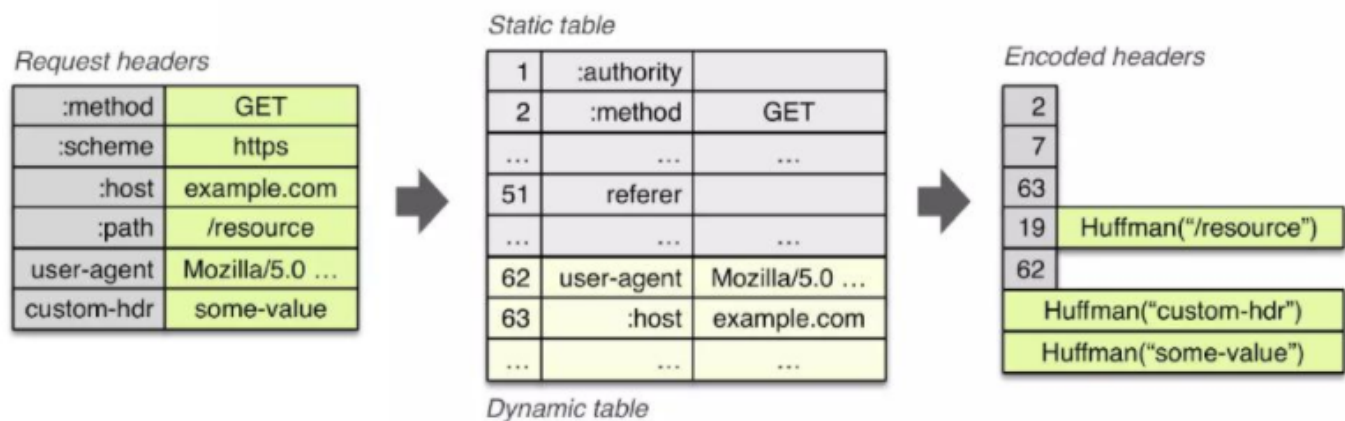
- Length帧的大小，`2^24` 帧最大不能超过 `16M`
- Type帧的类型：常用的就是 `HEADERS`，`DATA`
- Flags标志位：常用的是 `END_HEADERS`，`END_STREAM`，`PRIORITY`
- Stream Identifier 流的标号

2.头部压缩

使用 **HPACK** 算法压缩HTTP头

- 废除起始行，全部移入到Header中去，采用**静态表**的方式压缩字段
- 如果是自定义Header，在发送的过程中会添加到**静态表**后，也就是所谓的**动态表**
- 对内容进行哈夫曼编码来减小体积

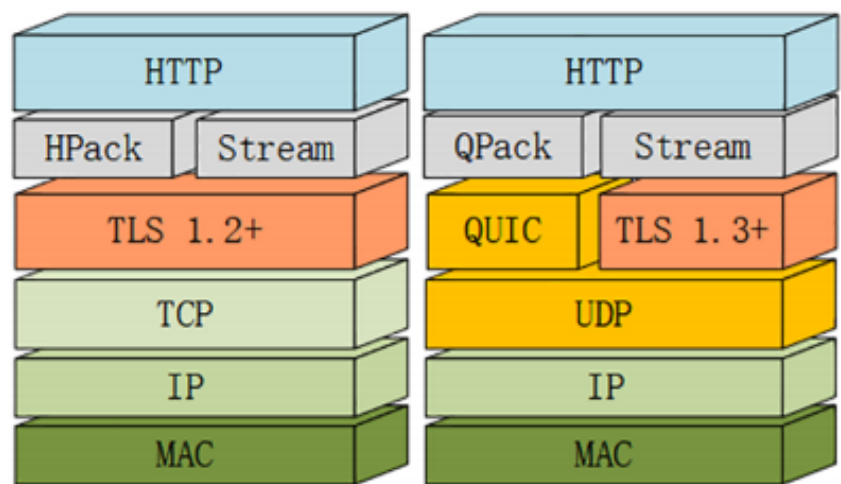
HPACK header compression



3.服务端推送

服务端可以提前将可能会用到的资源主动推送到客户端。

五.HTTP/3



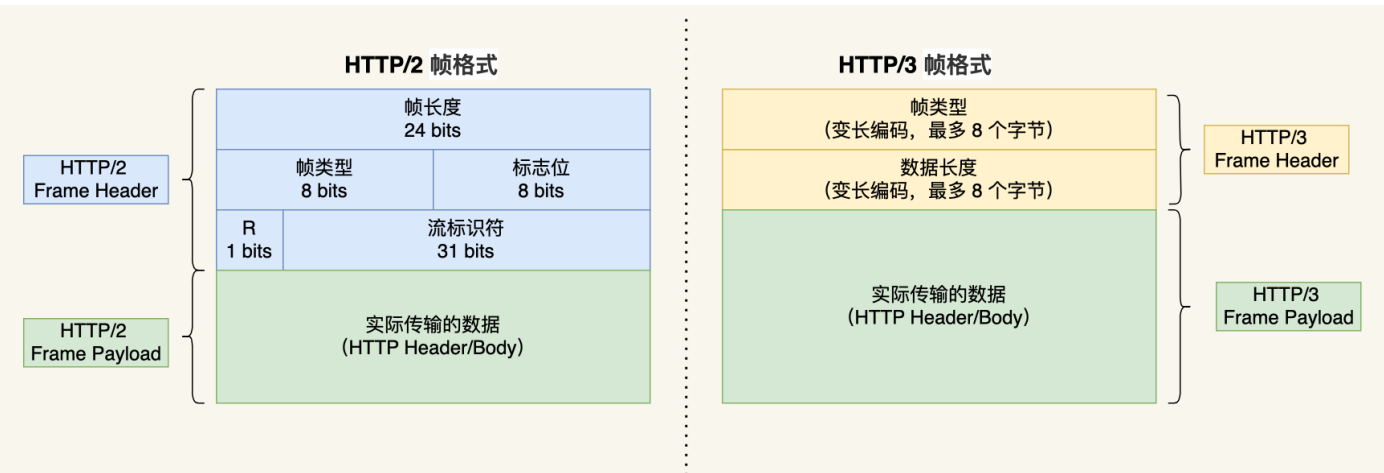
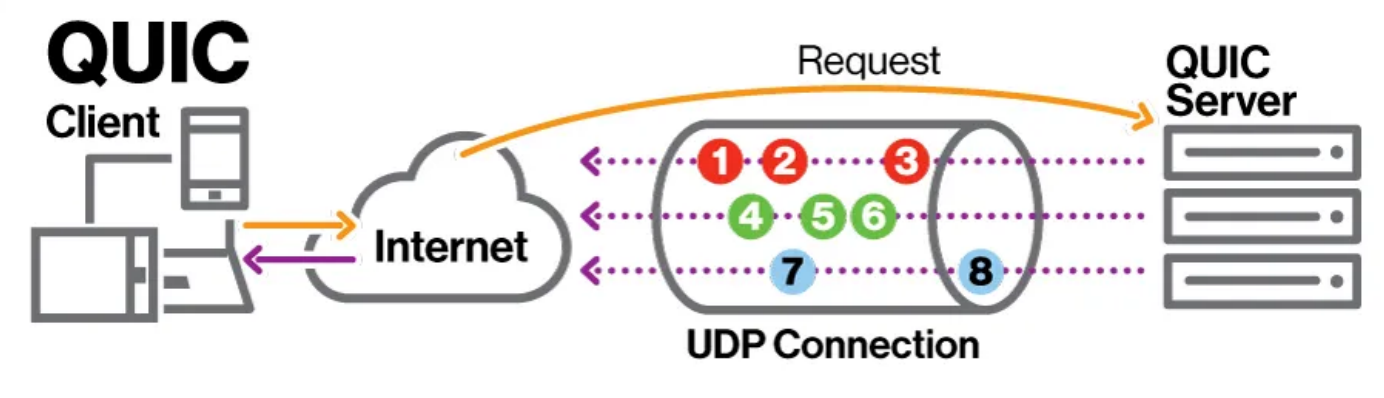
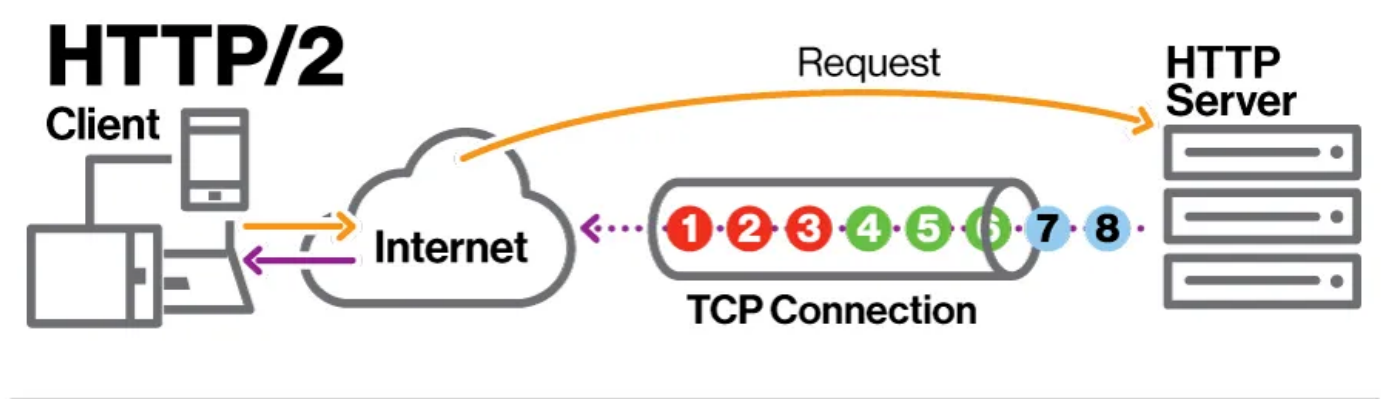
解决TCP中队头阻塞问题

TCP为了保证可靠传输，如果在传输的过程中发生丢包，可能此时其他包已经接受完毕，但是仍要等待客户端重传丢失的包。这就是TCP协议本身**队头阻塞**的问题。

HTTP/3 目前还处于草案阶段

1.QUIC 协议

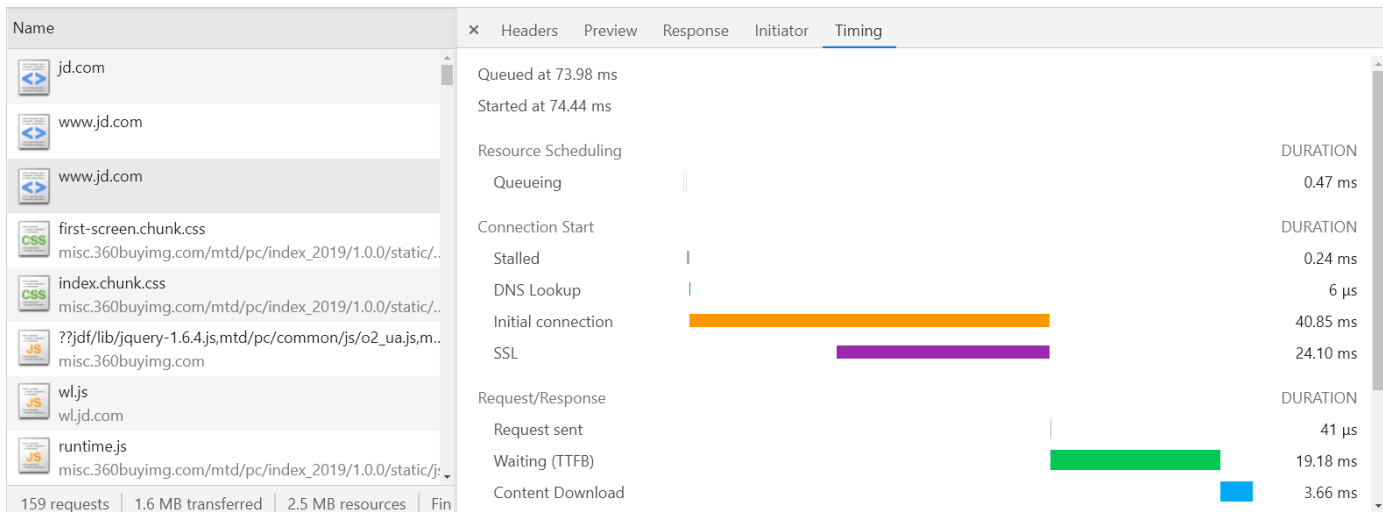
- HTTP/3中关键的改变，那就是把下层的 TCP 换成了 UDP。UDP 无序从而解决了队头阻塞的问题
 - QUIC 基于 UDP 之前说过 UDP 是无连的，接速度比 TCP 快
- QUIC 基于 UDP 实现了可靠传输、流量控制，引入流和多路复用
- QUIC 全面采用加密通信,QUIC 使用了 TLS 1.3, 首次连接只需要 1RTT
- 支持链接迁移，不受 IP 及 port 影响而发生重连，通过 ConnectionID 进行链接
- 使用 QPACK 进行头部压缩，HPACK 要求传输过程有序（动态表），会导致队头阻塞。



HTTP2 帧中需要封装流，HTTP3 则可以直接使用 Quic 里的stream

HTTP中的优化

一. Timing



- **Queueing**: 请求发送前会根据优先级进行排队，同时每个域名最多处理6个TCP链接，超过的也会进行排队，并且分配磁盘空间时也会消耗一定时间。
- **Stalled**: 请求发出前的等待时间（处理代理，链接复用）
- **DNS lookup**: 查找DNS的时间
- **initial Connection**: 建立TCP链接时间
- **SSL**: SSL握手时间（SSL协商）
- **Request Sent**: 请求发送时间（可忽略）
- **Waiting (TTFB)**: 等待响应的时间，等待返回首个字符的时间
- **Content Downloaded**: 用于下载响应的时间

二. 优化

- 减少网站中使用的域名 域名越多，DNS解析花费的时间越多。
- 减少网站中的重定向操作，重定向会增加请求数量。
- 选用高性能的Web服务器 Nginx 代理静态资源。
- 资源大小优化：对资源进行压缩、合并（合并可以减少请求，也会产生文件缓存问题），使用 gzip/br 压缩。
- 给资源添加强制缓存和协商缓存。
- 升级 HTTP/1.x 到 HTTP/2
- 付费、将静态资源迁移至 CDN

三. CDN

CDN 的全称是Content Delivery Network，受制于网络的限制，访问者离服务器越远访问速度就越慢

核心就是离你最近的服务器给你提供数据（代理 + 缓存）

- 先在全国各地架设 CDN 服务器
- 正常访问网站会通过 DNS 解析，解析到对应的服务器
- 解析1：我们通过 CDN 域名访问时，会被解析到 CDN 专用 DNS 服务器。并返回 CDN 全局负载均衡服务器的 IP 地址。
- 解析2：向全局负载均衡服务器发起请求，全局负载均衡服务器会根据用户 IP 分配用户所属区域的负载均衡

服务器。并返回一台 CDN 服务器 IP 地址

- 用户向 CDN 服务器发起请求。如果服务器上不存在此文件。则向上一级缓存服务器请求，直至查找到源服务器，返回结果并缓存到 DNS 服务器上。

