# **http**

## 什么是http？

http是应用层的协议；

应用层协议是客户端和服务端通信的一个约束；

http是基于TCP协议之上

## 什么是TCP ？

面向**连接**的、**可靠的**传输协议；

什么是连接？

三次握手：

TCP是传输控制层的协议；

TCP协议由操作系统实现，操作系统对上提供Socket接口；

## TCP和UDP

TCP 传输控制协议；

UDP 用户数据报协议；

TCP和UDP都是传输层的协议；

TCP协议是一种可靠的、一对一的、面向有连接的通信协议；

UDP协议是一种不可靠的、面向无连接、可以实现多对一、一对多和一对一连接的通信协议。

## TCP是如何保证数据传输的可靠性？

TCP主要提供了校验和、序列号/确认应答、超时重传、最大消息长度、滑动窗口控制等方法实现。

## TCP数据包

**一、TCP数据包的大小**

* TCP是以太网协议和IP协议的上层协议，也是应用层协议的下层协议。
* 最底层的以太网协议规定了电子信号如何组成数据包；
* 以太网数据包的大小是固定的，最初是1518字节，后来增加到1522字节。

其中1500字节是负载，22字节是头信息；

* IP数据包在以太网数据包的负载里面，它有自己的头信息，最少需要20字节，所以IP数据包的负载最多为1480字节；
* TCP数据包在IP数据包的负载里面，他的头信息最少也需要20字节，因此TCP数据包的最大负载是1460字节。由于IP和TCP协议往往有额外的头信息，所以TCP负载实际为1400字节左右。
* 因此1500字节的信息需要两个TCP数据包。
* HTTP2协议的一大改进，就是压缩HTTP协议的头信息，使得一个HTTP请求可以放在一个TCP数据包里面，而不是分成多个，这样就提高了速度。

**二、TCP数据包的编号**

* 1个包1400字节，那么一次性发送大量数据，就必须分成多个包。比如，一个10MB的文件，需要发送7100多个包。发送的时候，TCP协议为每个包编号（sequence number，简称seq），以便接收的一方按照顺序还原。万一发送丢包，也可以知道丢失的是哪一个包。
* 每一个包的编号是一个随机数。为了便于理解，这里就把它成为1号包。假定这个包的负载长度时100字节，那么可以推算出下一个包的编号应该是101。这就是说，每个数据包都可以得到两个编号；自身的编号，以及下一个包的编号。接收方由此知道，应该按照什么顺序将他们还原成原始文件。

## 什么是socket?

## https工作原理

1. **客户端发送HTTPS请求、传送加密方式列表、传递client\_random**

SSL连接总是由客户端启动，在SSL会话开始时，执行SSL握手。用户在浏览器里输入一个https网址，然后连接到server的443端口。

客户端发送以下：

* 列出客户端密支持的加密方式列表（以客户端首选项顺序排列），如SSL的版本、客户端支持的加密算法和客户端支持的数据压缩方法（hash算法）；
* 包含28字节的随机数，client\_random

1. **服务端回应算法、传送server\_random、传送SSL证书**

服务器端返回以下：

* 服务器端选出的一套加密算法和Hash算法；
* 服务器生成的随机数server\_random
* SSL数字证书（服务器使用带有SSL的X.509 V3数字证书），这个证书包含网站地址、公匙public\_key，证书的颁发机构、过期时间等。

**服务端的配置**

采用HTTPS协议的服务器必须要有一套数据证书，可以自己制作，也可以像组织申请。区别是自己颁发的证书需要客户端验证通过，才可以继续访问，而使用受信任公司申请的证书则不会弹出提示页面。这台证书其实就是一对公匙和私匙。

\*\*如果对公匙和私匙不太理解，可以想象成一把钥匙和一个锁，只是全世界只有你一个人有这把钥匙，你可以把锁给别人，别人可以用这个锁把重要的东西锁起来，然后发给你，因为只有你一个人有这把钥匙，所以只有你才能看到被这把锁锁起来的东西。

**使用非对称加密算法生成公匙和私匙。**

**非对称私密算法有：RSA、DSA/DSS**

1. **客户端TLS 解析证书并传送加密信息**

* **解析证书**

这部分工作是由客户端的TLS来完成的

首先会验证证书是否有效，这是对服务端的一种认证，比如颁发机构、过期时间等，如果发现异常，则会弹出一个警告框，提示证书存在问题。

* **根据server\_random生成随机值premaster\_secre**

如果证书没有问题，那么浏览器根据步骤3的server\_random生成一个随机值premaster\_secret（前2个字节是协议版本号，后26字节是用在对称加密密匙的随机数字）

* **根据premaster\_secre、client\_random、server\_random生成master\_secret**

master\_secret的生成需要premaster\_secret，并需要client\_random和server\_random作为种子。

master\_secret = PRF(premaster\_secret,”master secret”, client\_random + server\_random)

* **发送公匙public\_key加密的premaster\_secret给服务端**

客户端发送公匙public\_key加密的premaster\_secret。目的就是让服务器得到这个随机数，以后客户端和服务器的通信就可以通过这个随机数进行加密解密来的。

**4、服务器解密信息并传输加密后的信息**

服务器用私匙private\_key解密后，得到客户端传过来的随机数premaster\_secret，又由于服务器在步骤1中收到了client\_random，所以服务器根据相同的生成算法，在相同输入参数的情况下，得到相同他的master\_secret。然后把内容通过该值进行对称加密。

非对称加密算法：RSA、DSA/DSS

对称加密算法：AES、RC4/3DES

HASH算法：MD5、SHA1、SHA256

非对称加密算法生成公匙和私匙； 对称加密算法用于对真正传输的数据进行加密；HASH算法用于验证数据的完整性。

对称加密：就是指加密和解密时使用的密匙都是同一个，是“对称”的

url上的路径、大部分header、和全部body会加密

## 一个完整的http请求包含：

* 缓存查找
* DNS查找

1. 浏览器检查本地hosts文件是否有这个网址的映射关系，如果有就调用这个IP地址映射，完成域名解析；
2. 如果没有找到则会查找本地DNS解析器缓存，如果查找到则返回；
3. 如果没有找到则会查找本地DNS服务器，如果查找到则返回；
4. 如果没有找到则会查找根域名服务器，如果查找到则返回
5. 如果没有找到则会查找顶级域名服务器（cn/com），如果查找到则返回
6. 依次类推，直到本地域名服务器得到IP地址并把它缓存到本地，共供下次查询使用。

域名的解析是从右到左的过程 ./com/baidu.com/www.baidu.com

真正的网址是[www.baidu.com](http://www.baidu.com).

根域名服务器就是一个点，默认情况下所有的网址最后一位都是一个点，故省略。

* TCP握手
* 浏览器发出http请求
* 服务器接收请求
* 服务器处理请求并发回响应
* 浏览器接收响应
* 四次挥手

请求到资源后开始渲染

## 浏览器的渲染过程

* 解析HTML生成DOM树
* 样式计算

1. 将css转换为stylesheets
2. 转换样式表的属性值，使其标准化，如rem->px,red->rgb(255,0,0)
3. 计算出dom树中每个节点的样式属性

css继承规则和层叠规则

层叠是css的一个基本特征，它是一个定义了如何合并来自多个源的属性值的算法。

这个阶段最终输出的内容是每个DOM节点的样式，并被保存在ComputedStyle的结构中。

* 布局

计算dom树中可见元素的几何位置

1. 创建布局树layoutTree

遍历dom树中的所有可见节点，并将这些节点加到布局树中；不可见的节点忽略。

如head标签下面的全部内容，如display:none的节点都忽略。

1. 布局计算

计算出每个元素的具体位置

* 分层

渲染引擎为特定的节点生成专门的图层，并生成一棵对应的图层树Layerout。

满足以下两点中的任意一点的元素可以被提升为单独的图层：

第一点：拥有层叠上下文属性的元素

第二点：需要剪裁的地方

* 图层绘制
* 栅格化

绘制列表只是用来记录绘制顺序和绘制指令的列表，

实际上绘制操作是由渲染引擎的合成线程来完成的。

1. 主线程将绘制列表提交给合成线程；
2. 合成线程将图层划分为图块，大小256\*256/512\*512；
3. 优先将视口附近的图块生成视图。

栅格化过程会使用GPU来加速生成

使用GPU生成位图的过程叫做快速栅格化

* 合成和显示

所有的图块都被栅格化以后，合成线程生成一个绘制图块的命令，然后将该命令提交给浏览器进程，浏览器进程根据该命令，将页面内容绘制到内存中，最后将内存显示到屏幕中。

[重排和重绘](#_重绘和回流)

抓取一个HTTP请求

Queueing：在请求队列中的时间

Stalled：从TCP链接建立完成，到真正可以传输数据之间的时间差，此时间包括代理协商时间。

Proxy negotiation：与代理服务器连接进行协商所花费的时间

DNS Lookup：执行DNS查找所花费的时间，页面上每个不同的域都需要进行DNS查找

Initial Connection/Connecting：建立连接所花费的时间，包括TCP握手/重试和协商SSl

SSL：完成SSL握手所花费的时间

Request sent：发出网络请求所花费的时间，通常为1ms的时间

Waiting（TFFB）：TFFB是发出页面请求到接收应答数据第一个字节的时间总和，它包含了DNS解析时间和获取响应消息第一个字节的时间。

Content Download：接收响应数据所花费的时间。

## 三次握手、四次挥手

**三次握手**

1. 客户端发送一个SYN包告诉服务端我的初始序列号是X；
2. 服务端收到SYN包后回复给客户端一个ACK确认包，告诉客户端我收到了；

接着服务端也需要告诉客户端自己的初始序列号，于是服务端也发送一个SYN包告诉客户端我的初始序列号是Y；

1. 客户端收到后，回复服务端一个ACK确认包说我知道了。

三次握手的本质是确认通信双方收发数据的能力。

首先，我让信使运输一份信件给对方，对方收到了，那么他就知道了我的发件能力和他的收件能力是可以的；

于是他给我回信，我若收到了，我便知我的发件能力和他的收件能力是可以的，并且他的发件能力和我的收件能力是可以的；

然而此时他还不知道他的发件能力和我的收件能力到底可不可以，于是我最后反馈一次，他若收到了，他便清楚他的发件能力和我的收件能力是可以的。

|  |  |
| --- | --- |
| 我 | 发件能力 |
| 收件能力 |
| 对方 | 发件能力 |
| 发件能力 |

**问题1：初始序列号X、Y是可以写死固定的吗？为什么不能？**

不能，会造成新连接到初始序列号和旧连接的初始序列号产生冲突。

**问题2：假如客户端发送一个SYN包给服务端后就挂了或者不管了，这个时候这个连接处于什么状态？会超时吗？为什么？**

客户端发送SYN包给服务端后挂了，服务端回给客户端的SYN-ACK一直没有收到客户端的ACK确认，这个时候这个连接既没建立起来，也不算失败。这就需要一个超时时间让服务端将这个连接断开，否则这个连接就会一直占用服务端的SYN连接队列中的一个位置，大量这样的连接就会将服务端的SYN连接队列耗尽，让正常的连接无法处理。

目前，Linux下默认会进行5次重发SYN-ACK包，重试的间隔时间从1S开始，下次的重试间隔时间是前一次的双倍，5次的重试间隔时间为1s、2s、4s、8s、16s，总共31s，第5次发出后还要等32s都知道第5次也超时了，所以总共需要63s，TCP才会把断开这个连接。

由于，SYN超时需要63秒，那么就给攻击者一个攻击服务器的机会，攻击者在短时间内发送大量的SYN包给服务端，用于耗尽服务端的SYN队列对于应对SYN过多的问题，linux提供了几个TCP参数来调整。

**问题3：没有第三次握手会怎么样？**

为了防止已失效的连接请求报文段突然又传送到了服务端，因而产生错误。

客户端发出的第一个连接请求报文段并没有丢失，而是再某个网络节点长时间的滞留了，以致于延误到连接释放以后的某个时间才到达服务器端。

本来这是一个早已生效的报文段。但服务器收到此失效的连接请求报文段后，就误认为是客户端再次发出的一个新的连接请求。于是就像客户端发出确认报文段，同意建立连接。

假设不采用“三次握手”，只要服务器端发出确认，新的连接就建立的。

由于现在客户端并没有发出建立连接的请求，因此不会理睬服务器端的确认，也不会向server发送确认包。但服务器端却以为新的运输连接已经建立，并一直等到客户端发来数据。这样，服务端的很多资源就白白浪费了。

**问题四：服务器何时分配资源？**

第二次握手：服务器的TCP收到连接请求报文段后，如同意建立连接，就像客户端发回确认，并在OS内核中为该TCP连接分配TCP缓存和变量。

**四次挥手**

1. 客户端发送一个FIN包告诉服务端我已经没数据需要发送服务端了；
2. 服务端收到后回复一个ACK确认包说我知道了；
3. 然后服务端在自己也没数据发送客户端后，服务端也发送一个FIN包给客户端告诉客户端我也已经没有数据发送给客户端；
4. 客户端收到后，就会回复一个ACK确认包说我知道了。

**问题1：客户端和服务端同时发起断开连接的FIN包会怎么样呢，TCP状态是怎么转移的？**

TCP的两端同时发起FIN包进行断开连接，那么两端可能出现完全一样的状态转移FIN\_wait1—>closeing->time\_wait，也就会客户端和服务端最后同时进入time\_wait状态

**问题2：四次挥手过程中，服务端的ACK确认包能不能和接下来的FIN包合并成一个包呢，这样四次挥手就变成了三次挥手。**

答案是可能的。TCP是全双工通信，客户端在自己已经不会再有新的数据发送给服务端后，可以发送FIN信号告知服务端，这边已经终止客户端对服务端的数据传输。

但是，这个时候服务端还是可以进行往客户端发送数据包，于是两端数据传输的终止时序上是独立并且可能翔哥比较长的时间，这个时候就必须最少需要4次挥手。

但是，如果服务端在收到客户端的FIN包后，在也没数据需要发送给客户端了，那么对客户端的ACK包和自己的FIN包就可以合并成为一个包发送过去，这样四次挥手就变成个了3次（linux协议栈似乎就是这样实现的）

**问题3：四次挥手过程中，首先断开连接的一端，在回复最后一个ACK后，为什么要进行time\_wait呢（超时设置是2\*MSL，RFC793定义了MSL是2分钟，Linux设置成了30秒）？在time\_wait的时候又不能释放资源，白白让资源占用那么长时间，能不能省了time\_wait呢，为什么？**

## http请求方法GET/POST/PUT…

**客户端发送一个HTTP请求到服务器的请求消息包括以下格式：请求行（request line）、请求头部（header）、空行和请求数据四个部分组成，**



http中定义了以下几种请求方法

GET、POST、PUT、DELETE、HEAD、TRACE、OPTIONS

根据 HTTP 标准，HTTP 请求可以使用多种请求方法。

HTTP1.0 定义了三种请求方法： GET, POST 和 HEAD方法。

HTTP1.1 新增了六种请求方法：OPTIONS、PUT、PATCH、DELETE、TRACE 和 CONNECT 方法。

|  |  |  |
| --- | --- | --- |
| **序号** | **方法** | **描述** |
| 1 | GET | 请求指定的页面信息，并返回实体主体。 |
| 2 | HEAD | 类似于 GET 请求，只不过返回的响应中没有具体的内容，用于获取报头 |
| 3 | POST | 向指定资源提交数据进行处理请求（例如提交表单或者上传文件）。数据被包含在请求体中。POST 请求可能会导致新的资源的建立和/或已有资源的修改。 |
| 4 | PUT | 从客户端向服务器传送的数据取代指定的文档的内容。 |
| 5 | DELETE | 请求服务器删除指定的页面。 |
| 6 | CONNECT | HTTP/1.1 协议中预留给能够将连接改为管道方式的代理服务器。 |
| 7 | OPTIONS | 允许客户端查看服务器的性能。 |
| 8 | TRACE | 回显服务器收到的请求，主要用于测试或诊断。 |
| 9 | PATCH | 是对 PUT 方法的补充，用来对已知资源进行局部更新 。 |

1. GET：对这个资源的查操作
2. DELETE：对这个资源的删操作，请求服务器删除指定的页面。但要注意：客户端无法保证删除操作一定会执行，因为HTTP规范允许服务器在不通知客户端的情况下撤销请求；
3. HEAD：与GET方法的行为很类似，但服务器在相应时不会返回消息体。一个HEAD请求的响应中，HTTP头中包含的元信息应该和一个GET请求的响应消息相同。这种方法可以用来获取请求中隐含的信息，而不用传输实体本身。也经常用来测试超链接的有效性、可用性和最近的修改。
4. TRACE：回显服务器收到的请求，主要用于测试或诊断
5. OPTIONS：用于获取当前URL所支持的方法。若请求成功，则它会把HTTP头中包含一个名为“Allow”的头，值是所支持的方法，如“GET、POST”。
6. PUT 从客服端像服务器传输的数据取代指定的文档内容。

**GET和POST的区别**

1. GET参数通过URL传递，POST放在Request body中。
2. GET请求会被浏览器主动cache，而POST不会，除非手动设置。
3. GET请求参数会被完整保留在浏览器历史记录里，而POST中的参数不会被保留
4. GEI请求中有非ASCII字符，会在请求之前进行转码，POST不会，因为POST在Request body中，通过MIME，也就可以传递非ASCII字符
5. 浏览器通常都会限制url长度在2K个字节，而（大多数）服务器最多处理64K大小的url；POST请求理论上无长度限制。
6. GET提交的数据大小，不同浏览器的限制不同，一般在2k-8k之间；POST提交的数据比较大，大小靠服务器的设定值限制，而且某些数据只能用POST方法，比如file
7. GET产生一个TCP数据包；POST产生两个TCP数据包。对给GET方式的请求，浏览器会把http header和data一并发送出去，服务器响应200（返回数据）；而对于POST，浏览器先发送header，服务器响应100 continue，浏览器再发送data，服务器响应200 ok（返回数据）。

## http常见状态码

<https://www.runoob.com/http/http-status-codes.html>

200 请求成功，并返回数据

204 无内容。服务器成功处理，但未返回内容。在未更新网页的情况下，可确保浏览器继续显示当前文档。

206 部分内容。

301 永久重定向

302 临时移动，可使用原有URL

304 资源未修改，可使用缓存

401 请求要求用户的身份认证

402 保留，将来使用

403 没有权限访问等

404 找不到资源

405客户端请求中的方法被禁止

500 服务器内部错误

## 断点续传、

## HTTP/2

HTTP/2协议的主要目的是提高网页性能。

头信息（header）原来是直接传输文本，现在是压缩后传输。

原来同一个TCP连接里面，上一个回应（response）发送完了，服务器才能发送下一个，现在可以多个回应一起发送。

## 长链接Connection: keep-alive、管道、多路复用

* **长连接**

HTTP/1.0中默认使用Connection:close；在HTTP/1.1中已经默认使用Connection:keep-alive。

Keep-alive模式更加高效，因为避免了链接建立和释放的开销。

如果一个连接不会断开，那么多个请求之间如何进行区分？

* Content-length

Content-length表示实体内容的长度。当服务器能明确知道返回内容的长度时，可以设置content-length，浏览器通过这个字段来判断当前请求的数据是否已经全部接受。

* transfer-encoding

当服务器无法知道实体内容的长度时，就可以通过指定transfer-encoding:chunked来告知浏览器当前的编码是将数据分成一块一块传递的。还可以指定transfer-encoding:gzip,

Chunked表明实体内容不仅是gzip压缩的，还是分块传递的。最后，当浏览器接收到一个长度为0的chunked时，知道当前请求内容已全部接收。

Http1.1基于串行文件传输数据，因此这些请求必须是有序的，所以实际上我们只是节省了建立连接的时间，而获取数据的时间并没有减少

最大并发数问题，假设我们在Apache中设置了最大并发数300，而因为浏览器本省的限制，最大请求数为6，那么服务器能承载的最高并发数是50

* **管道（pipeline）连接**

HTTP/1.1的新特性，允许在持久连接上可选的使用请求管道。

管道连接：将多个HTTP请求放到一个TCP连接中发送，在发送的过程中不需要等待服务器对上一个请求的响应；但是，客户端还是要按照发送请求的顺序来接收响应。及服务器需要按照顺序处理请求的，如果前一个请求非常耗时，那么后续请求都会收到影响，这就是所谓的线头阻塞。

另外，只有幂等请求（get、head等）可以进行管线化，而POST则有所限制。

此外，初次创建连接时也不应启动管线机制，因为服务器不一定支持HTTP1.1版本的协议。

* **长连接和管线化的区别**

长连接的请求和响应是顺序执行的，只有请求1的响应收到之后，才会发送请求2；

而管线化不需要等到上一次请求得到响应就可以进行下一次请求，实现并行发送请求。

HTTP协议的长连接和TCP的长连接的区别

HTTP的长连接是为了维持连接，以便复用连接，以减少TCP连接建立次数；

TC的长连接是检查TCP连接状态的机制。当网络两端建立了TCP连接后，空闲时，服务器会向客户端发送侦测包，直到收到对方的回答（ask），如果一直不回答，则会丢弃该TCP连接。TCP连接默认闲置时间是2小时，一般设置为30分钟足够了。

* **多路复用**

而HTTP2引入二进制数据帧和流的概念，其中帧对数据进行顺序标识、这样浏览器收到数据之后，就可以按照序列对数据进行合并，而不会出现合并后数据错乱的情况。同样是因为有了序列，服务器也可以并行的传输数据。

HTTP2对同一域名下所有请求都是基于流，也就是说同一域名不管访问多少文件，也只建立一路连接。同样Apache的最大连接数为300，因为有了这个新特性，最大的并发就可以提升到300，比原来提升了6倍。

## 服务器推送 websocket

<http://www.ruanyifeng.com/blog/2017/05/websocket.html>

http://www.ruanyifeng.com/blog/2018/03/http2\_server\_push.html

服务器推送是HTTP/2协议里面，唯一需要开发者自己配置的功能。其他功能都是服务器和浏览器自动实现，不需要开发者关心。

WebSocket对象提供了用于创建和管理WebSocket连接，以及可以通过该连接发送和接收数据的API。

服务器端

客户端

## **HTTP cookies**

HTTP cookies（也叫Web Cookie或浏览器Cookie）是服务器发送到用户浏览器并保存在本地的一小块数据，它会在浏览器下次像同一服务器再发起请求时被携带并发送到服务器。

Cookie主要用于以下三个方面：

* 会话状态管理（如用户登录状态、购物车、游戏分数或其它需要记录的信息）
* 个性化设置（如用户自定义设置、主题等）
* 浏览器行为跟踪（如跟踪分析用户行为）
* 创建Cookie

当服务器收到HTTP请求时，服务器可以在响应头里面添加一个Set-Cookie选项。浏览器收到响应后通常会保存下Cookie，之后对该浏览器每一次请求中都通过Cookie请求头部将Cookie信息发送给服务器。

* 定义Cookie的声明周期

Cookie的生命周期可以通过两种方式定义：

1. 会话期Cookie是最简单的Cookie：浏览器关闭之后它会被自动删除，也就是说它仅在会话期内有效。
2. 持久性Cookie的生命周期取决于过期时间（Expires）或有效期（Max-Age）指定的一段时间。

* 限制访问Cookie

1. HttpOnly

Document.cookie API无法访问带有HttpOnly属性的cookie，此类Cookie仅作用于服务器。

1. Secure

Cookie只应通过被HTTPS协议加密过的请求发送给服务端；

* Cookie的作用域

Domain和Path标识定义了cookie的作用域，即允许Cookie应该发送给哪些URL。

1. Domain指定哪些主机可以接受Cookie。如果不指定，默认为origin，不包含子域名。

如果指定了Domain，则一般包含子域名。因此，指定Domain比省略它的限制要少。

但是，当子域需要共享有关用户的信息时，这可能有所帮助。

1. Path标识制定了主机下的哪些路径可以接受Cookie
2. SameSite 服务器要求某个cookie在跨站请求时不会被发送

## **cookie、sessionStorage、localStorage**

<https://www.cnblogs.com/cencenyue/p/7604651.html>

storage [ˈstɔːrɪdʒ]

cookie是在HTML4中使用的给客户端保存数据的，也可以和session配合实现跟踪浏览器用户身份；

webstorage（包括：localStorage和sessionStorage）是在HTML5提出来的，纯粹为了保存数据，不会与服务器端通信。WebStorage两个主要目标：（1）提供一种在cookie之外存储会话数据的路径。（2）提供一种存储大量可以跨会话存在的数据的机制。

相同点：

都是保存在浏览器端，且同源的。  
cookie，localStorage，sessionStorage都是在客户端保存数据的，存储数据的类型：都是字符串。

不同点：

1、生命周期：  
1).cookie如果不设置有效期，那么就是临时存储（存储在内存中），是会话级别的，会话结束后，cookie也就失效了，如果设置了有效期，那么cookie存储在硬盘里，有效期到了，就自动消失了。  
2).localStorage的生命周期是永久的，关闭页面或浏览器之后localStorage中的数据也不会消失。localStorage除非主动删除数据，否则数据永远不会消失。  
3）sessionStorage仅在当前会话下有效。sessionStorage引入了一个“浏览器窗口”的概念，sessionStorage是在同源的窗口中始终存在的数据。只要这个浏览器窗口没有关闭，即使刷新页面或者进入同源另一个页面，数据依然存在。但是sessionStorage在关闭了浏览器窗口后就会被销毁。同时独立的打开同一个窗口同一个页面，sessionStorage也是不一样的。

2、网络流量：cookie的数据每次都会发给服务器端，而localstorage和sessionStorage不会与服务器端通信，纯粹为了保存数据，所以，webstorage更加节约网络流量。

3、大小限制：cookie大小限制在4KB，非常小；localstorage和sessionStorage在5M

4、安全性：WebStorage不会随着HTTP header发送到服务器端，所以安全性相对于cookie来说比较高一些，不会担心截获。

5、使用方便性上：WebStorage提供了一些方法，数据操作比cookie方便;  
6. 数据与服务器之间的交互方式  
cookie的数据会自动的传递到服务器，服务器端也可以写cookie到客户端  
sessionStorage和localStorage不会自动把数据发给服务器，仅在本地保存。

## cookie和session的区别

1）cookie数据存放在客户的浏览器上，session数据放在服务器上  
2）cookie不是很安全，别人可以分析存放在本地的cookie并进行cookie欺骗，考虑到安全应当使用session  
3）session会在一定时间内保存在服务器上，当访问增多，会比较占用你服务器的性能，考虑到减轻服务器性能方面，应当使用cookie  
4）单个cookie保存的数\*据不能超过4K，很多浏览器都限制一个站点最多保存20个cookie  
5）建议将登录信息等重要信息存放为session，其他信息如果需要保留，可以放在cookie中  
6）session保存在服务器，客户端不知道其中的信心；cookie保存在客户端，服务器能够知道其中的信息  
7）session中保存的是对象，cookie中保存的是字符串  
8）session不能区分路径，同一个用户在访问一个网站期间，所有的session在任何一个地方都可以访问到，而cookie中如果设置了路径参数，那么同一个网站中不同路径下的cookie互相是访问不到的

## Token

* Token的引入：Token是客户端频繁向服务器请求数据，服务器频繁的去数据库查询用户名和密码并进行对比，判断用户名和密码正确与否，并作出相应提示，在这样的背景下，Token便应运而生。
* Token的定义：Token是服务器生成的一串字符串，以作为客户端进行请求的一个令牌，当第一次登录后，服务器生成一个Token并将此Token返回给客户端，以后客户端只需要携带上这个Token前来请求数据即可，无需再次带上用户名和密码。
* 使用Token的目的：Token的目的是为了减轻服务器的压力，减少频繁的查询数据库，使服务器更加健壮。

## 跨域解决方案

* **JSONP**

最大特点是简单适用，兼容性好，缺点是只支持get请求，不支持post请求。

核心思想：网页通过添加一个<script>元素，向服务器请求JSON数据，服务器收到请求后，将数据放在一个指定名称的回调函数的参数位置传回来。

1、原生实现

<script src=”http://test.com/data.php?callback=dosomething”></script>

<script type=”text/javascript”>

function dosomething(res){

console.log(res.data)

}

</script>

* jQuery ajax

$.ajax({

url:’’,

type: ‘get’,

datatype: ‘jsonp’,

jsonpCallback: ‘handleCallback’,

data: {}

})

3、vue.js

this.$http.jsonp(‘’,{

params: {},

jsonp: ‘handleCallback’

}).then((res)=>{

console.log(res)

})

* **CORS（跨域资源分享）**

**一、简介**

CORS需要浏览器和服务器同时支持。目前，所有浏览器都支持该功能，IE浏览器不能低于IE10。

整个CORS通信过程，都是浏览器自动完成，不需要用户参与。对于开发者来说，CORS通信与同源的AJAX通信没有差别，代码完全一致。浏览器一旦发现AJAX请求跨源，就会自动添加一些附加的头信息，有时还会多出一次附加的请求，但用户不会有感觉。

因此，实现CORS通信的关键是服务器。只要服务器实现了CORS接口，就可以跨源通信。

二、两种请求

浏览器将CORS请求分成两类：简单请求和非简单请求

1. 普通跨域请求：只需要服务器端设置Access-Control-Allow-Origin
2. 带cookie跨域请求，前后端都需要进行设置

【前端设置】根据xhr.withCredentials字段判断是否带有cookie

【后端设置】Access-Control-Allow-Credentials 默认情况下，Cookie不包含再CORS请求中，设置为true，即表示服务器明确许可，cookie可以包含在请求中，一起发给服务器。

CORS支持所有类型的HTTP请求，是跨域HTTP请求的根本解决方案

JSONP只支持GET请求，JSONP的优势在于支持老式浏览器，以及可以向不支持CORS  
CORS的网站请求数据。

* **PostMessage**

Window.postMessage的功能是运行程序员跨域在两个窗口/frames间发送数据信息。基本上，它就像跨域的AJAX，但不是浏览器跟服务器之间的交互，而是两个客户端之间通信。

**数据发送端**

首先我们要做的是创建通信发起端，也就是数据源“source”。作为发起端，我们可以open一个新窗口，或创建一个frame，往新窗口里发送数据。

* **Nodejs中间件代理**

重定向

Proxy就是利用nodejs中间件代理做跨域的

原理：浏览器是禁止跨域的，但是服务器不禁止

代理服务器：

1. 接收客户端请求

代理服务器，直接和浏览器直接交互，需要设置CORS的首部字段

1. 将请求转发给服务器
2. 拿到服务器响应数据
3. 将响应转发给客户端

* **Nginx代理**

## DNS解析

1. 浏览器缓存
2. 操作系统缓存
3. Hosts文件
4. 本地域名服务器（LDNS）
5. 根域名服务器，根域名服务器返回给本地域名服务器一个所查询域的主域名服务器（.com/.cn这些）地址
6. 本地域名服务器再发送请求给上一步返回的主域名服务器
7. 主域名服务器返回这个域名对应的

将域名解析为IP

1，检查本机DNS缓存 可以是使用 ipconfig /displaydns 查看du

2. 检查本机 hosts文件，可以使用 CMD %windir%\System32\drivers\etc\hosts 查看

3. 向DNS查询 nslookup

ipconfig /displaydns 显示系统中已经缓存的DNS域名

ipconfig /flushdns 这是清除DNS缓存用的

## DNS预解析 dns-prefetch

DNS prefetch是一种DNS预解析技术，当你浏览网页时，浏览器会在加载网页时对网页中的域名进行解析缓存，这样在你单击当前网页中的链接时，就无需进行DNS解析，减少用户等待时间，提高用户体验。

目前每次DNS解析，通常在200ms以下。

针对DNS解析耗时问题，一些浏览器通过DNS Prefetch来提高访问的流畅性

* **实现步骤及原理**

1、<meta>信息高速浏览器，当前页面要做DNS预解析

<meta http-equiv=”x-dns-prefetch-control” content=”on” />

2、<head>使用<link>标签来强制对DNS预解析

<link rel=”dns-prefetch” href=”//static.huishenghuo.com” />

* **注意实现**

Dns-prefetch需慎用，多页面重复DNS预解析会增加重复DNS查询次数

* **浏览器对网站第一次的域名DNS解析查找流程：**

浏览器缓存 -> 系统缓存 -> 路由器缓存 -> ISP -> DNS缓存 -> 递归搜索

* **禁止隐式的DNS prefetch**

<meta http-equiv=”x-dns-prefetch-control” content=”off” />

## TCP拥塞控制

TCP

1、慢开始

2、拥塞避免

3、快重传

4、快恢复

快开始

发送方：

接收方:

传输轮次

往返时间RTT，不是恒定的值

拥塞窗口cwnd

发送窗口swnd

慢开始门限ssthresh

cwnd初始值1

慢开始门限Ssthresh初始值=16

1、开始慢启动

发送方发送1个报文段，发送方收到1个确认，swnd+1=2

发送方发送2个报文段，发送方收到2个确认， swnd+1=3

发送方发送3个报文段，发送方收到3个确认，swnd+1=4

………

2、当swnd == cwnd == 16 时，开始执行拥塞避免算法，此时cwnd按照指数增长

此时swnd=16，cwnd=1，此时只能发送一个报文段

发送方发送1个报文段，发送方收到1个确认，cwnd + 1 =2

发送方发送2个报文段，发送方说到2个确认，cwnd+2=4

发送方发送4个报文段，发送方收到4个确认，cwnd + 4 = 8

发送方发送8个报文段，发送方收到8个确认，cwnd + 8 = 16

3、此时cwnd == ssthresh，开始执行拥塞避免算法，此时cwnd每次只加1

发送方发送16个报文段，发送方收到16个确认，cwnd + 1 = 17

发送方发送17个报文段，发送方收到17个确认，cwnd + 1 = 18

发送方发送18个报文段，发送方收到18个确认，cwnd + 1 = 19

….

4、当发送方发送24个报文段，但只收到20个报文段，此时重新设置慢开始门限ssthresh=24/2=12，拥塞窗口cwnd减少为1，开始执行慢启动

总结：

1、TCP发送方一开始使用慢开始算法，让拥塞窗口cwnd的值从1开始按照指数规律增长

2、当拥塞窗口cwnd的值增长到初始的慢开始门限值ssthresh时，停止使用慢开始算法，转而执行拥塞避免算法，让拥塞窗口cwnd的值按线性加1的规律增长

3、当发生超时重传时，就判断网络可能出现拥塞，采取相应的措施

一方面将慢开始门限值更改为发生拥塞时的拥塞窗口cwnd值的一半

另一方面将拥塞窗口cwnd的值减少为1

并重新开始执行慢开始算法，拥塞窗口cwnd 的值又从1开始按照指数规律增长

“慢开始”是指一开始向网络注入的报文段少，而不是值拥塞窗口cwnd增长速度慢

“拥塞避免”并非完全能够避免拥塞，而是指在拥塞避免阶段将拥塞窗口控制为按线性规律增长，比较不容易出现拥塞。

有时，个别报文段在网络中丢失，但实际上网络并没有发生阻塞

这将导致发送方超时重传，并误认为网络发生了阻塞

发送方错误的启动慢开始算法，并把阻塞窗口cwnd又设置为最小值1，因而降低了传输的效率

采用快重传算法可以让发送方尽快进行重传，而不是等待超时重传计时器超时再重传。

要求接收方不要等待自己发送数据时才进行捎带确认，而是要立即发送确认；

即使收到了失序的报文段也要立即发出对已收到的报文段的重复确认

发送方一旦收到3个连续的重复确认，就将相应的报文段立即重传，而不是等该报文段的超时重传计时器超时再重传。

对于个别丢失的报文段，发送方不会出现超时重传，也就不会误认为出现了拥塞（进而降低了拥塞窗口cwnd为1）。使用快重传可以使整个网络的吞吐量提高约20%。

发送方一旦收到了3个重复确认，就知道现在只是丢失了个别的报文段，于是不启动慢开始算法，而是执行快恢复算法

发送方将慢开始门限ssthresh值和拥塞窗口cwnd值调整为当前窗口的一半，开始执行拥塞避免算法

也有的快恢复实现是把快恢复开始时的拥塞窗口cwnd再增大一些，及等于新的ssthresh+3

理由是：既然发送方收到了3个重复确认，就表明有3个数据报文段已经离开网络；

这三个报文段不再消耗网络资源而是停留在接收方的接收缓存中；

可见现在网络中不是堆积了报文段而是减少了3个报文段，由此可以适当的拥塞窗口扩大些。