一、浏览器原理

1.浏览器内核总结

浏览器内核是多线程，在内核控制下各线程相互配合以保持同步，一个浏览器通常由以下常驻线程组成：GUI渲染线程、JavaScript引擎线程、事件触发线程、定时触发器线程、异步http请求线程

浏览器/RunTime	内核（渲染引擎）	JavaScript 引擎
Chrome	webkit->blink	V8
FireFox	Gecko	SpiderMonkey
Safari	Webkit	JavaScriptCore
Edge	EdgeHTML	Chakra(for JavaScript)
IE	Trident	JScript（IE3.0-IE8.0）
Opera	Presto->blink	Linear A（4.0-6.1）/ Linear B（7.0-9.2）/ Futhark（9.5-10.2）/ Carakan（10.5-）
Node.js	-	V8

- **Trident**：这种浏览器内核是 IE 浏览器用的内核，因为在早期 IE 占有大量的市场份额，所以这种内核比较流行，以前有很多网页也是根据这个内核的标准来编写的，但是实际上这个内核对真正的网页标准支持不是很好。但是由于 IE 的高市场占有率，微软也很长时间没有更新 Trident 内核，就导致了 Trident 内核和 W3C 标准脱节。还有就是 Trident 内核的大量 Bug 等安全问题没有得到解决，加上一些专家学者公开自己认为 IE 浏览器不安全的观点，使很多用户开始转向其他浏览器。
- **Gecko**：这是 Firefox 和 Flock 所采用的内核，这个内核的优点就是功能强大、丰富，可以支持很多复杂网页效果和浏览器扩展接口，但是代价是也显而易见就是要消耗很多的资源，比如内存。
- **Presto**：Opera 曾经采用的就是 Presto 内核，Presto 内核被称为公认的浏览网页速度最快的内核，这得益于它在开发时的天生优势，在处理 JS 脚本等脚本语言时，会比其他的内核快3倍左右，缺点就是为了达到很快的速度而丢掉了一部分网页兼容性。
- **Webkit**：Webkit 是 Safari 采用的内核，它的优点就是网页浏览速度较快，虽然不及 Presto 但是也胜于 Gecko 和 Trident，缺点是对于网页代码的容错性不高，也就是说对网页代码的兼容性较低，会使一些编写不标准的网页无法正确显示。WebKit 前身是 KDE 小组的 KHTML 引擎，可以说 WebKit 是 KHTML 的一个开源的分支。
- **Blink**：谷歌在 Chromium Blog 上发表博客，称将与苹果的开源浏览器核心 Webkit 分道扬镳，在 Chromium 项目中研发 Blink 渲染引擎（即浏览器核心），内置于 Chrome 浏览器之中。其实 Blink 引擎就是 Webkit 的一个分支，就像 webkit 是KHTML 的分支一样。Blink 引擎现在是谷歌公司与 Opera Software 共同研发，上面提到过的，Opera 弃用了自己的 Presto 内核，加入 Google 阵营，跟随谷歌一起研发 Blink

1): GUI渲染线程

GUI渲染线程负责渲染浏览器界面HTML元素，解析HTML，CSS，构建DOM树和渲染树，布局和绘制等。

当界面需要重绘(Repaint)或由于某种操作引发回流(重排)(reflow)时,该线程就会执行。

在Javascript引擎运行脚本期间,GUI渲染线程都是处于挂起状态的,也就是说被"冻结"了，GUI更新会被保存在一个队列中等到JS引擎空闲时立即被执行。

2): JavaScript引擎线程

JavaScript引擎，也可以称为JS内核，主要负责处理Javascript脚本程序，例如V8引擎。

JS引擎一直等待着任务队列中任务的到来，然后加以处理，一个Tab页（renderer进程）中无论什么时候都只有一个JS线程在运行JS程序（单线程）。

为什么js是单线程的？与异步冲突吗？

JS中其实是没有线程概念的，所谓的单线程也只是相对于多线程而言。JS的单线程是指一个浏览器进程中只有一个JS的执行线程，同一时刻内只会有一段代码在执行。

举个通俗例子，假设JS支持多线程操作的话，JS可以操作DOM，那么一个线程在删除DOM，另外一个线程就在获取DOM数据，这样子明显不合理，这算是证明之一。因此**为了防止渲染出现不可预期的结果**，浏览器设置GUI渲染线程与JavaScript引擎为互斥的关系，当JavaScript引擎执行时GUI线程会被挂起，GUI更新会被保存在一个队列中等到引擎线程空闲时立即被执行。如果JS执行的时间过长，这样就会造成页面的渲染不连贯，导致页面渲染加载阻塞的感觉。

JS的宿主环境（比如浏览器，Node）是多线程的，宿主环境通过某种方式（事件驱动）使得js具备了异步的属性。例：定时器触发(settimeout和setinterval) 是由**「浏览器的定时器线程」**执行的定时计数，然后在**定时时间把定时处理函数的执行请求插入到JS执行队列的尾端**（所以用这两个函数的时候，*实际的执行时间是大于或等于指定时间的*，不保证能准确定时的

3): 事件触发线程

当一个事件被触发时该线程会把事件添加到待处理队列的**队尾**，等待JS引擎的处理。

这些事件可以是当前执行的代码块如定时任务、也可来自浏览器内核的其他线程如鼠标点击、AJAX异步请求等，但由于JS的**单线程关系**所有这些事件都得排队等待JS引擎处理。

4): 定时触发器线程

`setInterval`与`setTimeout`所在线程

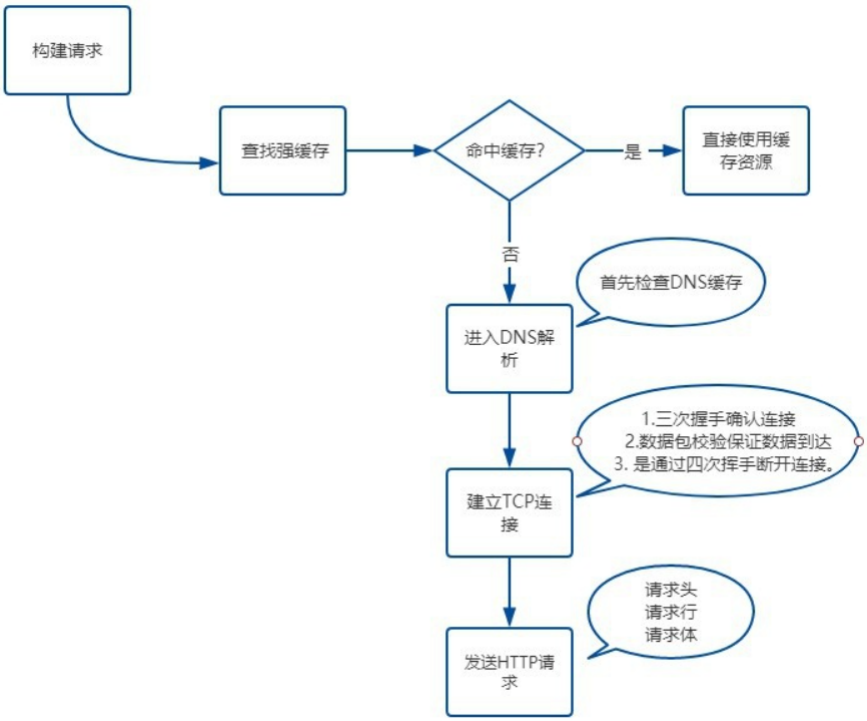
浏览器定时计数器并不是由JavaScript引擎计数的, 因为JavaScript引擎是单线程的, 如果处于阻塞线程状态就会影响记计时的准确。通过单独线程来计时并触发定时（计时完毕后，添加到事件队列中，等待JS引擎空闲后执行）

注意，W3C在HTML标准中规定，规定要求setTimeout中低于4ms的时间间隔算为4ms

5): 异步http请求线程

在XMLHttpRequest在连接后是通过浏览器新开一个线程请求。将检测到状态变更时，如果设置有回调函数，异步线程就产生状态变更事件，将这个回调再放入事件队列中，再由JavaScript引擎执行

2.输入URL到界面加载的过程



1、浏览器的地址栏输入URL并按下回车

2、浏览器查找当前URL是否存在缓存，并比较缓存是否过期（查找强缓存）。当浏览器发现请求的资源已经在浏览器缓存中存有副本（控制台的network中会标注该请求是在服务器中请求的还是浏览器缓存中的，缓存的查找顺序按照：Service Worker-->Memory Cache-->Disk Cache-->Push Cache，具体介绍看后面），它会拦截请求，返回该资源的副本，并直接结束请求，而不会再去源服务器重新下载。如果缓存查找失败，就会进入网络请求过程了。

3、DNS解析URL对应的IP，浏览器提供了**DNS数据缓存功能**

> DNS解析过程：

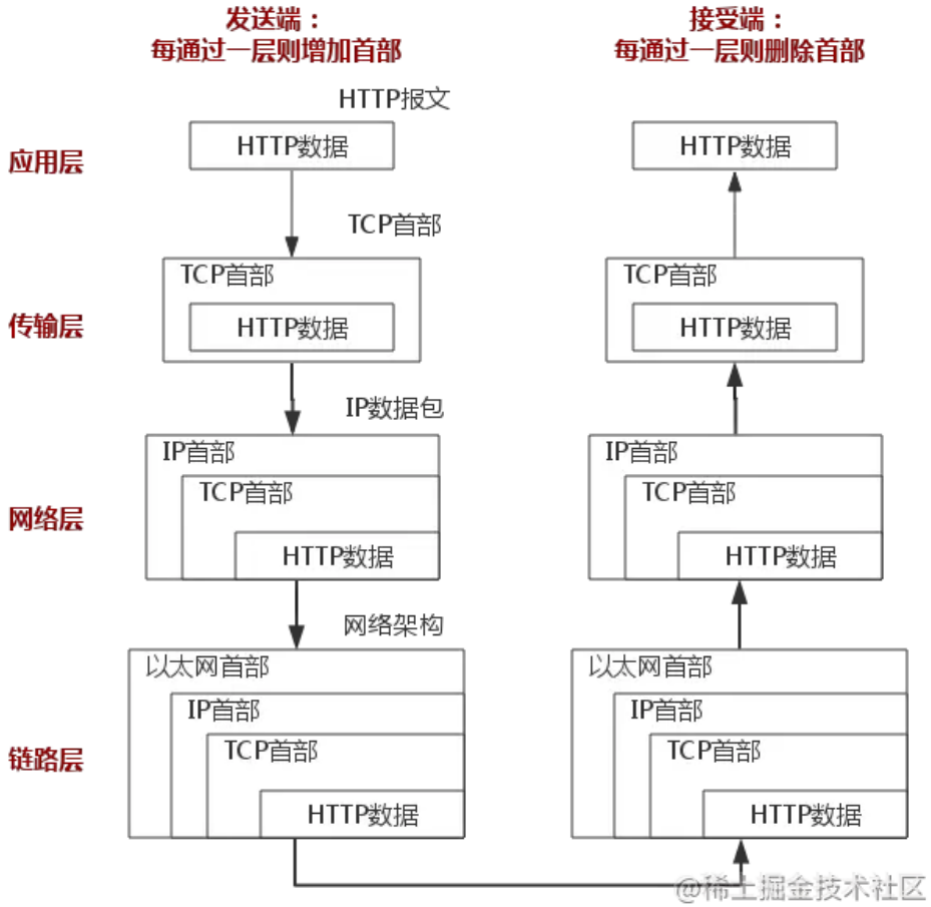
1. 用户主机上运行着DNS的客户端，就是我们的PC机或者手机客户端运行着DNS客户端。
2. 浏览器将接收到的url中抽取出域名字段，就是访问的主机名，比如www.feng.com, 并将这个主机名传送给DNS应用的客户端.
3. DNS客户机端向DNS服务器端发送一份查询报文，报文中包含着要访问的主机名字段（中间包括一些列缓存查询以及分布式DNS集群的工作）。
4. 该DNS客户机最终会收到一份回答报文，其中包含有该主机名对应的IP地址。
5. 一旦该浏览器收到来自DNS的IP地址，就可以向该IP地址定位的HTTP服务器发起TCP连接。

4、根据IP建立TCP连接（**三次握手**）。（Chrome 在同一个域名下要求同时最多只能有 6 个 TCP 连接，超过 6 个的话剩下的请求就得等待)

> PS：TCP 连接通过什么手段来保证数据传输的可靠性？一是`三次握手`确认连接，二是`数据包校验`保证数据到达接收方，三是通过`四次挥手`断开连接。

5、HTTP发起请求, 共发送请求头、请求体、请求行三种

http协议向服务器发送请求，发送请求的过程中，浏览器会向Web服务器以Stream(流)的形式传输数据，告诉Web服务器要访问服务器里面的哪个Web应用下的Web资源




```
// 请求行：由请求方法、请求URI和HTTP版本协议。请求方法是GET，路径为根路径，HTTP协议版本为1.1
GET / HTTP/1.1

//请求头：之前说的Cache-Control、If-Modified-Since、If-None-Match都由可能被放入请求头中作为缓存的标识信息
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3
Accept-Encoding: gzip, deflate, br
Accept-Language: zh-CN,zh;q=0.9
Cache-Control: no-cache
Connection: keep-alive
Cookie: /* 省略cookie信息 */
Host: www.baidu.com
Pragma: no-cache
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (iPhone; CPU iPhone OS 11_0 like Mac OS X)
AppleWebKit/604.1.38 (KHTML, like Gecko) Version/11.0 Mobile/15A372 Safari/604.1

//请求体：只有在Post方法下存在，常见的场景是表单提交
```

6、服务器处理请求，浏览器接收HTTP响应。其中，网络响应具有三个部分:**响应行**、**响应头**和**响应体**。

服务器接收到浏览器传输的数据后，开始解析接收到的数据，服务器解析请求里面的内容时知道客户端浏览器要访问的是应用里面的哪这个Web资源，然后服务器就去读取这个Web资源里面的内容，将读到的内容再以Stream(流)的形式传输给浏览器

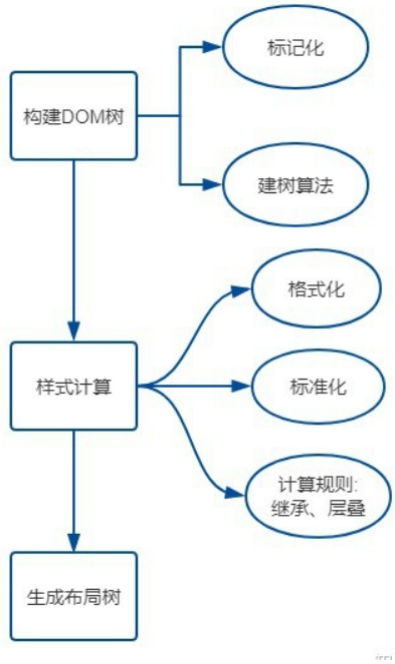
```
//响应行，由HTTP协议版本、状态码和状态描述组成
HTTP/1.1 200 OK

//请求头：包含了服务器及其返回数据的一些信息，服务器生成数据的时间、返回的数据类型以及对即将写入的Cookie信息
Cache-Control: no-cache
Connection: keep-alive
Content-Encoding: gzip
Content-Type: text/html;charset=utf-8
Date: Wed, 04 Dec 2019 12:29:13 GMT
Server: apache
Set-Cookie:
rsv_i=f9a0SIItKqzv7kqgAAgphbGyRts3RwTg%2FLyU3Y5Eh5LwyfOOrAsvdezbay0QqkDqFZ0DfQXby4wXKT8Au8O7ZT9UuMsBq2k; path=/; domain=.baidu.com
```

7、渲染页面，自上而下加载，构建DOM树（详见后面的解析）

8、关闭TCP连接（四次挥手）

```
PS:在相应完成后，不一定TCP连接断开，需要判断`Connection`字段。如果请求头或响应头中包含**Connection: Keep-Alive**，表示建立了持久连接，这样`TCP`连接会一直保持，之后请求统一站点的资源会复用这个连接；反之则断开连接
```

3.浏览器的渲染过程*




```
由于浏览器无法直接理解`HTML字符串`，因此将这一系列的字节流转换为一种有意义并且方便操作的数据结构，这种数据结构就是`DOM树`。`DOM树`本质上是一个以`document`为根节点的多叉树。
```

DOM树：解析器首先会创建一个`document`对象。标记生成器会把每个标记的信息发送给**建树器。建树器**接收到相应的标记时，会**创建对应的 DOM 对象。**创建这个`DOM对象`后会做两件事情: 1. 将`DOM对象`加入 DOM 树中。 2. 将对应标记压入存放开放(与`闭合标签`意思对应)元素的栈中。（即DOM树的构建过程是一个深度遍历过程，当前节点的所有子节点都构建好后才会构建当前节点的下一个兄弟节点）

DOM样式：先格式化与标准化，之后计算每个节点的具体样式信息，主要遵从两个规则：继承和层叠 -> 每个子节点都会默认继承父节点的样式属性，如果父节点中没有找到，就会采用浏览器默认样式，也叫`UserAgent样式`

生成布局树：1. 遍历生成的 DOM 树节点，并把他们添加到`布局树中`。 2. 计算布局树节点的坐标位置。

将CSS解析成CSS Rule Tree

Rendering Tree: 根据DOM树和CSSOM构造Rendering Tree，有了Render Tree，浏览器已经能知道网页中有哪些节点、各个节点的CSS定义以及他们的从属关系。下一步操作称之为Layout，顾名思义就是计算出每个节点在屏幕中的位置。

绘制：遍历render树，并使用UI后端层绘制每个节点

4.浏览器的解析过程 *

浏览器的渲染主要分为以下几个模块：

1、HTML解析器：解释`HTML文档`的解析器，主要作用是将HTML文本解释为DOM树。在此过程中，渲染引擎会尝试尽快的把内容显示出来。它不会等到所有HTML都被解析完才创建并布局渲染树。它会在处理后续内容的同时把处理过的局部内容先展示出来。

2、CSS解析器：它的作用是为DOM中的各个元素对象`计算出样式信息`，为布局提供`基础设施`，将CSS解析成CSS Rule Tree（CSSOM），CSS选择器的读取顺序是从右向左

```
#molly div.haha span{color:#f00}
```

例：先找到span然后顺着往上找到class为“haha”的div再找到id为“molly”的元素, 成功匹配到则加入结果集; 如果直到根元素html都没有匹配，则不再遍历这条路径，从下一个span开始重复这个过程。

3、JavaScript解析器：JavaScript引擎能够解释`JavaScript代码`，并通过`DOM接口和CSS接口`来修改`网页内容 和样式信息`，从而改变`渲染的结果`。JS解析器会找到js当中的所有变量、函数、参数等等，并且把变量赋值为未定义(undefined)。把函数取出来成为一个函数块，然后存放到仓库当中。这件事情做完了之后才开始逐行解析代码（由上向下，由左向右），然后再去和仓库进行匹配。

4、布局（layout）：在DOM创建之后，浏览器内核（WebKit）需要将其中的元素对象同样式信息`结合起来`，计算他们的`大小位置等布局信息`，形成一个能表达着所有信息的`内部表示模型`

5、绘图模块（paint）：使用`图形库`将布局计算后的各个网页的节点绘制成图像结果

渲染分为以下几个步骤:

1、浏览器会从上到下解析文档

2、遇见HTML标记，调用**HTML解析器**解析为对应的token(一个token就是一个标签文本的序列化)并构建DOM树(就是一块内存，保存着tokens，建立他们之间的关系)

3、遇见style/link标记，调用相应解析器处理CSS标记，并构建出**CSS样式树**

PS：

- style标签中的样式：由**HTML解析器**进行解析，**不会**阻塞浏览器渲染(可能会产生“闪屏现象”)，不会阻塞DOM解析
- link引入的CSS样式：
    - 由CSS解析器进行解析，**会**阻塞浏览器渲染：因为HTML和CSS是**并行解析**的，所以CSS不会阻塞HTML解析
    - **会阻塞整体页面的DOM渲染**：因为最后要渲染必须CSS和HTML一起解析完并合成一处（Render Tree））；

    - **会阻塞后面的js语句执行**.
- JavaScript 是可操纵 DOM 和 css 样式的, 如果在修改这些元素属性同时渲染界面（即 JavaScript 线程和 UI 线程同时运行）,那么渲染线程前后获得的元素数据就可能不一致了
- 脚本的内容是获取元素的样式，宽高等`css`控制的属性，浏览器是需要计算的，也就是依赖于`css`。浏览器也无法感知脚本内容到底是什么，为避免样式获取，因而只好等前面所有的样式下载完后，再执行`JS`
- **不阻塞DOM的解析**
- **优化**：使用CDN节点进行外部资源加速，对CSS进行压缩，优化CSS代码(不要使用太多层选择器)

4、遇见script标记，调用`JavaScript引擎`处理script标记，绑定事件，修改DOM树/CSS树等

PS:

- **js会阻塞后续DOM的解析**，原因是：浏览器不知道后续脚本的内容，如果先去解析了下面的DOM，而随后的js删除了后面所有的DOM，那么浏览器就做了无用功，浏览器无法预估脚本里面具体做了什么操作，例如像document.write这种操作，索性全部停住，等脚本执行完了，浏览器再继续向下解析DOM
- **js会阻塞页面渲染**，原因是：js中也可以给DOM设置样式，浏览器等该脚本执行完毕，渲染出一个最终结果，避免做无用功。
- **js会阻塞后续js的执行**，原因是维护依赖关系，例如：必须先引入jQuery再引入bootstrap
- 无论css阻塞，还是js阻塞，**都不会阻塞浏览器加载外部资源**（图片、视频、样式、脚本等）

  原因：浏览器始终处于一种：**先把请求发出去**的工作模式，只要是涉及到网络请求的内容，无论是：图片、样式、脚本，都会先发送请求去获取资源，至于资源到本地之后什么时候用，由浏览器自己协调。这种做法效率很高。

5、将DOM树与CSS合并成一个**渲染树**

6、根据渲染树来渲染，计算每个节点的**几何信息**(这一过程需要依赖GPU)

7、最终将各个节点**绘制**在屏幕上

5.缓存相关知识

A：为什么要缓存（好处）？简述一下缓存机制?

答：web缓存的优点：1.减少网络延迟，加快网页打开速度 2.减少服务器压力 3.减少网络带宽消耗

缓存机制主要分为强缓存 - 协商缓存 - 缓存位置

强缓存（本地缓存）：浏览器中的缓存作用分为两种情况，一种是需要发送`HTTP`请求，一种是不需要发送。首先是检查强缓存，这个阶段`不需要`发送HTTP请求。HTTP1.0使用的是Expires，用具体时间标识缓存是否过期。HTTP1.1使用Cache-Control，使用max-age过期时长控制缓存，同时Cache-Control可以配合很多命令一起使用

```
// HTTP1.0 Expires
Expires: Wed, 22 Nov 2019 08:41:00 GMT
//HTTP1.1 Cache-Control
Cache-Control:max-age=3600
//Cache-Control常见的配合命令
private:  这种情况就是只有浏览器能缓存了，中间的代理服务器不能缓存。

no-cache: 跳过当前的强缓存，发送HTTP请求，即直接进入`协商缓存阶段`。

no-store：非常粗暴，不进行任何形式的缓存。

s-maxage：这和`max-age`长得比较像，但是区别在于s-maxage是针对代理服务器的缓存时间。
```

协商缓存：强缓存失效之后，浏览器在请求头中携带相应的**缓存tag**来向服务器发请求，由服务器根据这个tag，来决定是否使用缓存，这就是协商缓存。主要分为**Last-Modified** 和 **ETag**，各有优劣：

1: Last-Modified：最后修改时间，浏览器会在请求头上携带`If-Modified-Since`的字段，和这个服务器中**该资源的最后修改时间**对比:

- 如果请求头中的这个值小于最后修改时间，说明是时候更新了。
- 返回新的资源，跟常规的HTTP请求响应的流程一样

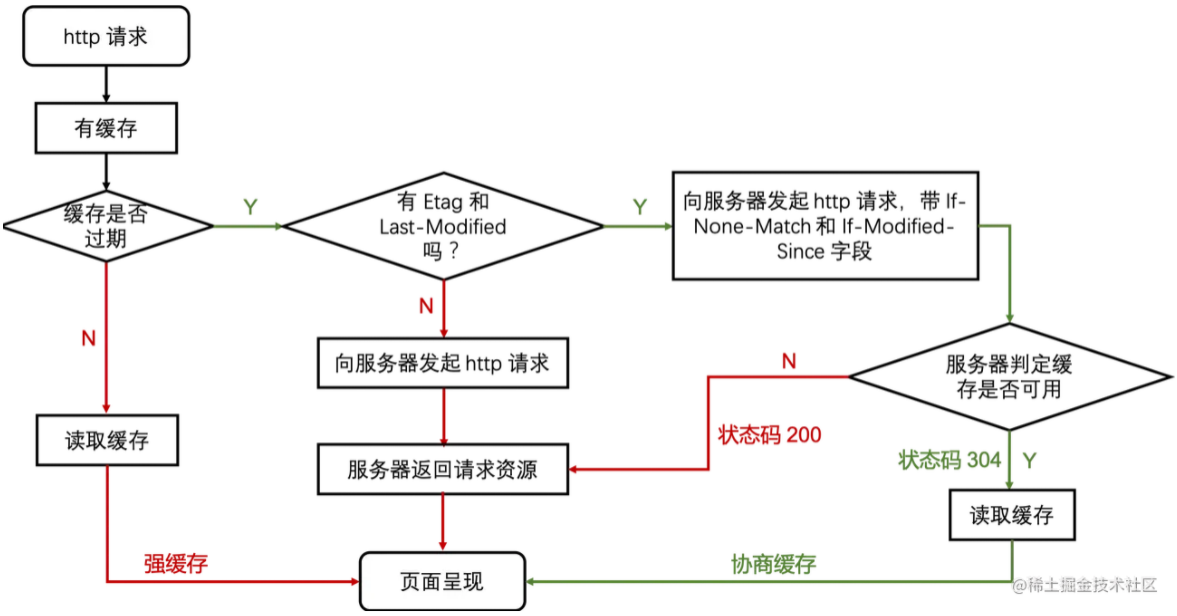
2：ETag：`ETag`是服务器根据当前文件的内容，给文件生成的唯一标识，只要里面的内容有改动，这个值就会变。浏览器请求头发送**If-None-Match**字段，服务器接收到**If-None-Match**后，会跟服务器上该资源的**ETag**进行比对:

- 如果两者不一样，说明要更新了。返回新的资源，跟常规的HTTP请求响应的流程一样。
- 否则返回304，告诉浏览器直接用缓存。

3：两者对比：

1. 在**精准度**上，`ETag`优于`Last-Modified`。 ETag 是按照内容给资源上标识，因此能准确感知资源的变化。而 Last-Modified 就不一样了，它在一些特殊的情况并不能准确感知资源变化，主要有两种情况:
    - 编辑了资源文件，但是文件内容并没有更改，这样也会造成缓存失效。
    - Last-Modified 能够感知的单位时间是秒，如果文件在 1 秒内改变了多次，那么这时候的 Last-Modified 并没有体现出修改了。
2. 在**性能**上，`Last-Modified`优于`ETag`，也很简单理解，`Last-Modified`仅仅只是记录一个时间点，而`Etag`需要根据文件的具体内容生成哈希值。

另外，如果两种方式都支持的话，服务器会优先考虑`ETag`



三种刷新操作对 http 缓存的影响

- 正常操作：地址栏输入 url，跳转链接，前进后退等。-> 强制缓存有效，协商缓存有效
- 手动刷新：f5，点击刷新按钮，右键菜单刷新。 -> 强制缓存失效，协商缓存有效
- 强制刷新：ctrl + f5，shift+command+r。 -> 强制缓存失效，协商缓存失效

B：缓存位置

浏览器中的缓存位置一共有四种，按优先级从高到低排列分别是： Service Worker -> Memory Cache -> Disk Cache -> Push Cache

- **Service Worker** 借鉴了 Web Worker的思路，即让 JS 运行在主线程之外，是一个服务器与浏览器之间的中间人角色。如果网站中注册了service worke，那么它可以拦截当前网站所有的请求，进行判断（需要编写相应的判断程序）。如果需要向服务器发起请求的就转给服务器，如果可以直接使用缓存的就直接返回缓存不再转给服务器。从而大大提高浏览体验。

  使用 Service Worker的话，传输协议必须为 HTTPS。因为 Service Worker 中涉及到请求拦截，所以必须使用 HTTPS 协议来保障安全。Service Worker 的缓存与浏览器其他内建的缓存机制不同，它可以让我们自由控制缓存哪些文件、如何匹配缓存、如何读取缓存，并且缓存是持续性的。

  由于它脱离了浏览器的窗体，因此无法直接访问 `DOM`。虽然如此，但它仍然能帮助我们完成很多有用的功能，比如 `离线缓存`、 `消息推送` 和 `网络代理` 等功能。其中的 `离线缓存` 就是 **Service Worker Cache**。
- **Memory Cache**指的是内存缓存，从效率上讲它是最快的。但是从存活时间来讲又是最短的，当渲染进程结束后，内存缓存也就不存在了。
- **Disk Cache**就是存储在磁盘中的缓存，从存取效率上讲是比内存缓存慢的，但是他的优势在于存储容量和存储时长。比较大的JS、CSS文件会直接被丢进磁盘，反之丢进内存 - 内存使用率比较高的时候，文件优先进入磁盘
- **Push Cache**即推送缓存，这是浏览器缓存的最后一道防线。它是 `HTTP/2` 中的内容，当以上三种缓存都没有命中时，才会被使用。它只在会话（Session）中存在，一旦会话结束就被释放，并且缓存时间也很短暂，在Chrome浏览器中只有5分钟左右。这种缓存设置了Last-Modifed但没有设置Cache-Control，也就是只拿到最后更新时间，但没有设置过期时间，这种情况下浏览器会有一个默认的缓存策略push cache，自动设置过期时间：(Date - Last-Modified)*0.1，也就是当前时间减去最后更新时间后再乘10%。

C：总结

对浏览器的缓存机制来做个简要的总结:

首先通过 `Cache-Control` 验证强缓存是否可用 - 如果强缓存可用，直接使用 - 否则进入协商缓存，即发送 HTTP 请求，服务器通过请求头中的`If-Modified-Since`或者`If-None-Match`字段检查资源是否更新 - 若资源更新，返回资源和200状态码 - 否则，返回304，告诉浏览器直接从缓存获取资源

6. 浏览器的本地缓存/存储（Storage）

浏览器的本地存储主要分为`Cookie`、`WebStorage`和`IndexedDB`，其中`WebStorage`又可以分为`localStorage`和`sessionStorage`

A: Cookie

`HTTP`协议是一个无状态协议，客户端向服务器发请求，服务器返回响应，故事就这样结束了，但是下次发请求如何让服务端知道客户端是谁呢？这种背景下，就产生了 **Cookie**。Cookie 本质上就是浏览器里面存储的一个很小的文本文件，内部以键值对的方式来存储。首先，客户端向服务器发送HTTP请求，服务器收到HTTP请求后，在相应头中添加一个set-cookiie字段。浏览器收到响应后保存cookie，之后对该服务器每一次请求中都通过Cookie字段将Cookie信息发送给服务器（向同一个域名下发送请求，都会携带相同的 Cookie，服务器拿到 Cookie 进行解析，便能拿到**客户端的状态**）。cookie的过期时间主要由**Last-Modified** 和 **ETag**设定。

Cookie 的作用很好理解，就是用来做**状态存储**的，但它也是有诸多致命的缺陷的：

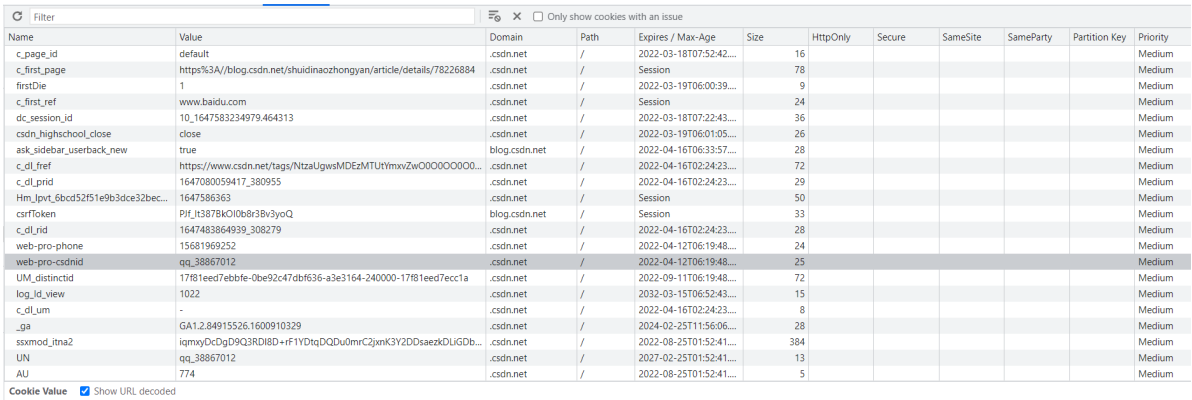
1. 容量缺陷。Cookie 的体积上限只有`4KB`，只能用来存储少量的信息。
2. 性能缺陷。Cookie 紧跟域名，不管域名下面的某一个地址需不需要这个 Cookie ，请求都会携带上完整的 Cookie，这样随着请求数的增多，其实会造成巨大的性能浪费的，因为请求携带了很多不必要的内容。
3. 安全缺陷。由于 Cookie 以纯文本的形式在浏览器和服务器中传递，很容易被非法用户截获，然后进行一系列的篡改，在 Cookie 的有效期内重新发送给服务器，这是相当危险的。另外，在`HttpOnly`为 false 的情况下，Cookie 信息能直接通过 JS 脚本来读取。

   防护方法：使用cookie的httponly属性。若此属性为true，则只有在http请求头中会带有此cookie的信息，而不能通过document.cookie来访问此cookie。

   1、如果在Cookie中设置了"HttpOnly"属性，那么通过后台程序读取，JS脚本将无法读取到Cookie信息，这样能有效的防止**XSS攻击。**
   2、但是设置HttpOnly属性，**Cookie盗窃**的威胁并没有彻底消除，因为cookie还是有可能**传递的过程中**被**监听捕获**后信息泄漏。

Cookie中的常用属性如下：

Name,Value(Cookie的名称、值)，Domain（域名）、Path、Expires/Max-age（过期时间）、Size、HttpOnly、Secure（指定cookie的值通过网络如何在用户和WEB服务器之间传递）



Secure字段：这个属性的值或者是“secure”，或者为空。缺省情况下，该属性为空，也就是使用**不安全的HTTP连接**传递数据。如果一个 cookie 标记为secure，那么，它与WEB服务器之间就通过**HTTPS**或者其它安全协议传递数据。不过，设置了secure属性不代表其他人不能看到你机器本地保存的cookie。换句话说，**把cookie设置为secure，只保证cookie与WEB服务器之间的数据传输过程加密**，而保存在本地的cookie文件并不加密。如果想让本地cookie也加密，得自己加密数据

B: localStorage

ocalStorage用于持久化的本地存储，除非主动删除数据，否则数据是永远不会过期的。localStorage有一点跟Cookie一样，就是针对一个域名，即在同一个域名下，会存储相同的一段localStorage。利用`localStorage`的较大容量和持久特性，可以利用`localStorage`存储一些内容稳定的资源，比如官网的`logo`，存储`Base64`格式的图片资源，因此利用`localStorage`

不过它相对Cookie还是有相当多的区别的:

- 容量：localStorage 的容量为2.5MB到10MB之间（各家浏览器不同），相比于Cookie的 4K 大大增加。当然这个 5M 是针对一个域名的，因此对于一个域名是持久存储的
- 只存在客户端：默认不参与与服务端的通信。这样就很好地避免了 Cookie 带来的性能问题和安全问题。
- 接口封装：通过localStorage暴露在全局，并通过它的 setItem 和 getItem等方法进行操作，非常方便。

C: sessionStorage

定义：sessionStorage用于本地存储一个会话（session）中的数据，这些数据只有在同一个会话中的页面才能访问
并且当会话结束后数据也随之销毁。因此sessionStorage不是一种持久化的本地存储，仅仅是**会话级别**的存储。只允许同一窗口访问

应用场景：对表单信息进行维护，将表单信息存储在里面，可以保证页面即使刷新也不会让之前的表单信息丢失；可以用它存储本次浏览记录。如果关闭页面后不需要这些记录，用`sessionStorage`就再合适不过了。事实上微博就采取了这样的存储方式。

sessionStorage和localStorage一下内容一致:

- 容量。容量上限也为 5M。
- 只存在客户端，默认不参与与服务端的通信。
- -接口封装。除了`sessionStorage`名字有所变化，存储方式、操作方式均和`localStorage`一样。

但`sessionStorage`和`localStorage`有一个本质的区别，那就是前者只是**会话级别的存储**，并不是持久化存储。会话结束，也就是页面关闭，这部分`sessionStorage`就不复存在了。

D: IndexedDB

`IndexedDB`是运行在浏览器中的`非关系型数据库`（不支持SQL查询语句），本质上是数据库, 允许储存大量数据，提供查找接口，还能建立索引。除了拥有数据库本身的特性，比如`支持事务`，`存储二进制数据`，IndexedDB有以下特点：

- 键值对存储： IndexedDB 内部采用对象仓库（object store）存放数据。所有类型的数据都可以直接存入，包括 JavaScript 对象。对象仓库中，数据以"键值对"的形式保存，**每一个数据记录都有对应的主键**，主键是**独一无二的**，不能有重复，否则会抛出一个错误。
- 异步操作：数据库的读写属于 I/O 操作, 浏览器中对异步 I/O 提供了支持，所以IndexedDB操作时不会锁死浏览器，用户依然可以进行其他操作，与LocalStorage形成鲜明对比（LocalStorage是同步的）。所以，IndexedDB的异步设计防止了大量数据的读写，不拖慢网页
- 受同源策略限制：即无法访问跨域的数据库。

- 支持事务： IndexedDB 支持事务（transaction），这意味着一系列操作步骤之中，只要有一步失败，整个事务就都取消，数据库回滚到事务发生之前的状态，不存在只改写一部分数据的情况
- 储存空间大： IndexedDB 的储存空间比 LocalStorage 大得多，一般来说不少于 250MB，甚至没有上限。
- 支持二进制储存：IndexedDB 不仅可以储存字符串，还可以储存二进制数据（ArrayBuffer 对象和 Blob 对象）。

E： WebSQL

WebSQL 与 IndexedDB 都是最新的 HTML5 本地缓存技术，相比于 Local Storage 和 Session Storage 来说，存储功能更强大，支持的数据类型也更多，比如图片、视频等。当前只有谷歌支持，ie和火狐均不支持

WebSQL 更准确的说是 WebSQL DB API，它是一种**操作本地数据库的网页 API 接口**，通过 API 可以完成客户端数据库的操作。当我们使用 WebSQL 的时候，可以方便地用 SQL 来对数据进行增删改查。而这些浏览器客户端，比如 Chrome 和 Safari 会用 SQLite 实现本地存储，微信就采用了 SQLite 作为本地聊天记录的存储。

F: Cookie、Session、Token

cookie存储在客户端，session存储在服务端，客户端只存储session id。如果将账户的一些信息都存入Cookie中的话，一旦信息被拦截，那么我们所有的账户信息都会丢失掉。所以就出现了Session，在一次会话中将重要信息保存在Session中，客户端存储的一个session id对应一次会话请求

Session：会话管理，每位用户收到一个随机的字符串（session id），当向服务器发送请求时，携带该session id。让服务器识别用户身份。浏览器客户端大多数采用cookie的方式存储session。服务器则将session保存在临时服务器中，当用户离开网站时，session会被销毁。但是当web服务器做了负载均衡，当下一个操作请求到另一台服务器时，session会丢失。

Token：

让服务器存储session id也不是不行，但是当服务器要存储所有人的session id时，太耗费服务器开销了（实在塞不下了233333），所以有大佬提出了token，原理如下：

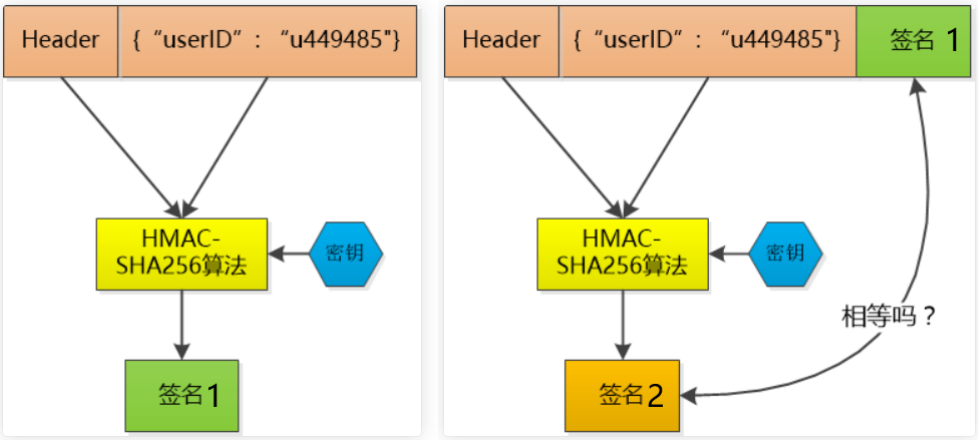
1：当用户A已经登录时，服务器给用户发送一个token（令牌），该token中包含了用户A的user id。重点是！！！服务器用了某种算法，已经对token进行了加密（形成了签名1），只有服务器才有密钥，所以别人不能伪造这个token了。服务器并不保存这个token，而是用户A保存。

2：每当用户A向服务器发送请求时，在请求头header中附带了自己的token（token中携带了user id和签名1）。

3：服务器用专属密钥解密token，得到user id后重新签名（签名2），并判断签名1和签名2是否相等。若相等，则告诉用户A可以成功登陆，反之，则需要用户重新认证。

这样服务器就不用存储session id，只要计算签名是否一致就可以。时间换空间

目前FaceBook、Twitter、Google+、Github的部分API和Web应用都使用到了tokens



token的几种特性（为什么会适用token）：

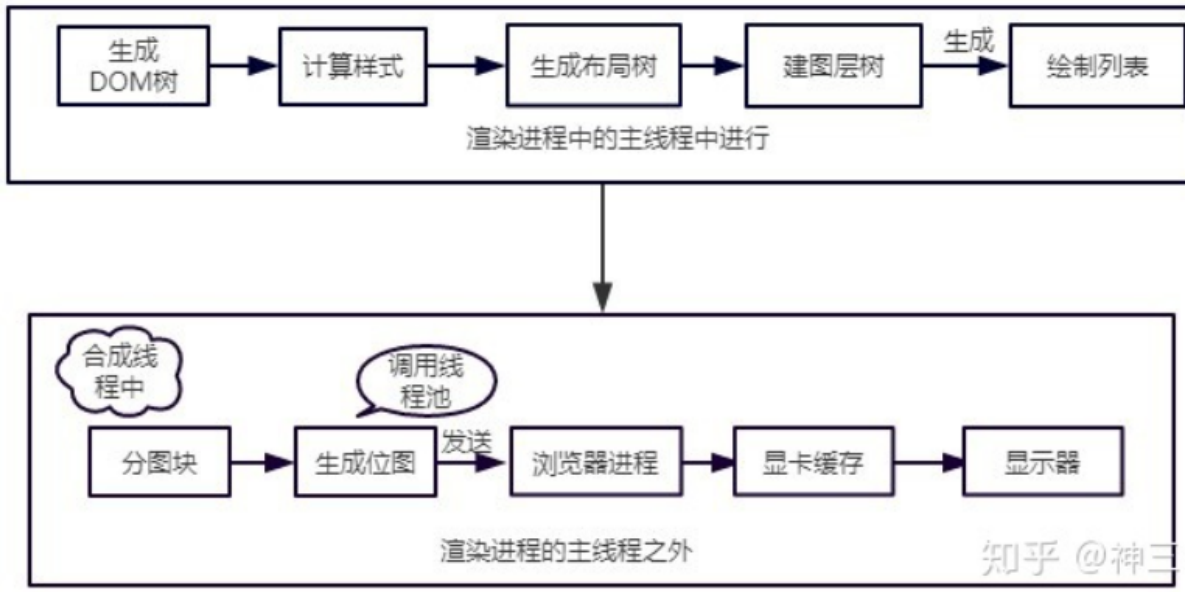
1. 无状态、可扩展
2. 支持移动设备
3. 跨程序调用
4. 安全

token可以理解为是基于客户端验证的一种验证方式，对比之下，服务器验证（服务器记录session id）有以下几种缺陷：

1. session：每次认证用户发起请求时，服务器需要创建一个记录来存储信息，当越来越多的用户发请求时，内存开销也会不断增加
2. 可扩展性：在服务器端内存中使用session存储登录信息，伴随的是可扩展问题
3. CORS：当跨平台or跨设备使用时，跨域资源共享很烦人，服务器与服务器之间可能出现禁止请求的问题
4. CSRF（跨站请求伪造）：用户被利用访问其他网站（后面开个专题详细写一下浏览器攻击）

7.重绘、重排与合成

渲染流水线流程如下：



A：重排

触发条件：当我们对 DOM 结构的修改引发 DOM 几何尺寸变化的时候，会发生`回流`的过程。相当于将**解析和合成**的过程重新又走了一篇，开销非常大的。有以下的操作会触发回流:

- 一个 DOM 元素的几何属性变化，常见的几何属性有`width`、`height`、`padding`、`margin`、`left`、`top`、`border`等等, 这个很好理解。
- 使 DOM 节点发生`增减`或者`移动`。当DOM 结构发生改变，则重新渲染 DOM 树，然后将后面的流程(包括主线程之外的任务)全部走一遍，包括生成DOM树、计算样式、生成布局树、建图层树生成绘制列表
- 读写`offset`族、`scroll`族和`client`族属性的时候，浏览器为了获取这些值，需要进行回流操作。
  调用`window.getComputedStyle`方法。
- 常见属性有：width、 top、 text-align 、height、 bottom、overflow-y 、 padding、 left、 font-weight 、margin、overflow 、 display、 position、font-family border-width、 float、 line-height 、 border

B：重绘

触发条件：当 DOM 的修改导致了样式的变化，并且没有影响几何属性的时候，会导致`重绘`(`repaint`)。只会修改**计算样式、绘制列表**两个阶段。

即有：重绘不一定导致回流，但回流一定发生了重绘。常见触发重绘的属性如下：color、 background、 outline-color 、border-style、outline 、border-radius、 outline-style 、 background-size、 box-shadow

C：合成

利用 CSS3 的`transform`、`opacity`、`filter`这些属性就可以实现合成的效果，也就是大家常说的**GPU加速**。在合成的情况下，会直接跳过布局和绘制流程，直接进入**非主线程**处理的部分，即直接交给**合成线程**处理。交给它处理有两大好处: 1. 能够充分发挥GPU的优势。合成线程生成位图的过程中会调用线程池，并在其中使用`GPU`进行加速生成，而GPU 是擅长处理位图数据的。2.没有占用主线程的资源，即使主线程卡住了，效果依然能够流畅地展示。

D： 指导意义/优化方案

1. 将多次改变样式属性的操作合为一次，避免频繁使用 style，而是采用修改`class`的方式，。例：元素位置移动变换时尽量使用CSS3的`transform`来代替`top，left`等操作，不使用table布局
2. 使用**createDocumentFragment**进行批量的 DOM 操作。
3. 对于 resize、scroll 等进行**防抖/节流**处理。
4. 添加 will-change: tranform ，让渲染引擎为其单独实现一个图层，当这些变换发生时，仅仅只是利用合成线程去处理这些变换，而不牵扯到主线程，大大提高渲染效率。当然这个变化不限于`tranform`, 任何可以实现合成效果的 CSS 属性都能用`will-change`来声明
5. 利用`文档素碎片（documentFragment）`，vue使用了该方式提升性能
6. 动画实现过程中，启用`GPU硬件加速：transform:tranlateZ(0)`。同时也可为动画元素`新建图层`，提高动画元素的`z-index`
7. 编写动画时，尽量使用`requestAnimationFrame`

8.为什么HTTPS数据传输更安全?

原理是在`HTTP`和`TCP`之间建立了一个中间层，当`HTTP`和`TCP`通信时并不是像以前那样直接通信，直接经过了一个中间层进行加密，将加密后的数据包传给`TCP`, 响应的，`TCP`必须将数据包解密，才能传给上面的`HTTP`。这个安全层称之为安全层，核心作用就是对数据进行加密。

在安全层中，主要有对称加密、非对称加密、对称与非对称加密结合、添加数字证书方法来保证数据安全。具体原理自行百度

9.事件的防抖和节流

A：节流（throttle）

节流的核心思想: 如果在定时器的时间范围内再次触发，则不予理睬，等当前定时器`完成`，才能启动**下一个定时器任务**。这就好比公交车，10 分钟一趟，10 分钟内有多少人在公交站等我不管，10 分钟一到我就要发车走人！通常利用函数节流，让函数有节制地执行，而不是毫无节制的触发一次就执行一次，规定在一个单位时间内，只能触发一次函数。如果这个单位时间内触发多次函数，只有一次生效。通常与setTimeout方法一起进行

适用场景：鼠标的点击事件，比如mousedown只触发一次；监听滚动事件，比如是否滑到底部加载更多；游戏中子弹发射的频率

```
const throttle = function(fn, interval) {
  let last = 0;
  return function (...args) {
    let context = this;
    let now = +new Date();
    // 还没到时间
    if(now - last < interval) return;
    last = now;
    fn.apply(this, args)
  }
}
```

B：防抖（debounce）

每次事件触发则删除原来的定时器，建立新的定时器。跟**王者荣耀**的**回城**功能类似，你反复触发回城功能，那么**只认最后一次**，从最后一次触发开始计时。

函数防抖是指在事件被触发n秒后再执行回调，如果在这n秒内又被触发，则重新计时。

适用场景：Search搜索，用户不断输入值时，用防抖来节约Ajax请求,也就是输入框事件；window触发resize时，不断调整浏览器大小会不断触发这个事件，用防抖让其只触发一次

C:加强版节流

原因：防抖有时候触发的太频繁会导致一次响应都没有，我们希望到了固定的时间必须给用户一个响应

```
function throttle(fn, delay) {
  let last = 0, timer = null;
  return function (...args) {
    let context = this;
    let now = new Date();
    if(now - last < delay){          //如果时间未到，则延迟执行
      clearTimeout(timer);
      setTimeout(function() {
        last = now;
        fn.apply(context, args);
      }, delay);
    } else {
      // 这个时候表示时间到了，必须给响应
      last = now;
      fn.apply(context, args);
```

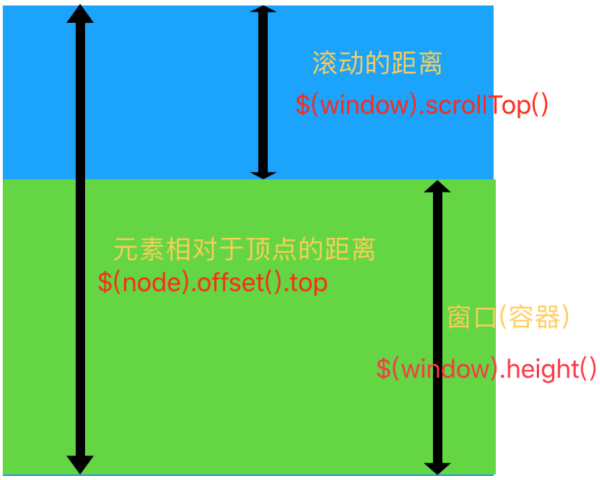
```
    }
  }
}
```

10.图片懒加载(建议看视频)

一张图片就是一个`<img>`标签，浏览器是否发起请求图片是根据`<img>`的src属性，所以实现懒加载的关键就是，在图片没有进入可视区域时，先不给`<img>`的src赋值，这样浏览器就不会发送请求了，等到图片进入可视区域再给src赋值。

A：方法一：滚动事件

```
//首先给图片一个占位资源:，data-src表示自定义属性，存放真正需要显示的图片路径，当页面滚动直至此图片出现在可视区域时，用js取到该图片的data-src的值赋给src，再加载src，渲染图片
<img src="default.jpg" data-src="http://www.xxx.com/target.jpg" />

//通过监听 scroll 事件（鼠标滚动）来判断图片是否到达视口。当图片距离视窗顶部的距离（imageTop）大于窗口显示区的高度（window.innerHeight），表明图片不能看到，不用加载。也要对scroll事件做节流，以免频繁触发：
window.addEventListener('scroll', throttle(lazyload, 200));

window.addEventListener('scroll',(e)=>{
    images.forEach(images => {
        const imageTop =image.getBoundingClientRect().top;          // 图片距离视窗顶部的距离
        if( imageTop < window.innerHeight){                        //当图片距离顶部的距离小于窗口显示区高度，则渲染该图片
            const data_src = image.getAttribute('data-src');
            image.sertAttribute('src',data_src);                  //将之前占位的图片（data-src）赋值给src，进行真正的图片渲染
        };
        console.log('scroll触发')
    })
})

页可见区域宽：  document.body.clientWidth;
网页可见区域高：  document.body.clientHeight;
网页可见区域宽：  document.body.offsetWidth (包括边线的宽);
网页可见区域高：  document.body.offsetHeight (包括边线的宽);
网页正文全文宽：  document.body.scrollWidth;
网页正文全文高：  document.body.scrollHeight;
网页被卷去的高：  document.body.scrollTop;
网页被卷去的左：  document.body.scrollLeft;
网页正文部分上：  window.screenTop;
网页正文部分左：  window.screenLeft;
屏幕分辨率的高：  window.screen.height;
屏幕分辨率的宽：  window.screen.width;
屏幕可用工作区高度：  window.screen.availHeight;
```



B：方法二：DOM元素的getBoundingClientRect方法

```
function lazyload() {
  for(let i = count; i <num; i++) {
    // 元素现在已经出现在视口中
    if(img[i].getBoundingClientRect().top <
document.documentElement.clientHeight) {
      if(img[i].getAttribute("src") !== "default.jpg") continue;
      img[i].src = img[i].getAttribute("data-src");
      count ++;
    }
  }
}
```

C：方法三：浏览器提供的构造函数IntersectionObserver（推荐）

这是浏览器内置的一个`API`，实现了`监听window的scroll事件`、`判断是否在视口中`以及`节流`三大功能。

```
let img = document.getElementsByTagName("img");

//IntersectionObserver是一个函数，所以需要创建一个实例对象
const observer = new IntersectionObserver(changes => {
  //changes 是被观察的元素集合，是个数组
  for(let i = 0, len = changes.length; i < len; i++) {
    let change = changes[i];
    if(change.isIntersecting) {             // 通过这个属性判断是否在视口中，即是否进行到
可视窗口，看到了图片
      const imgElement = change.target;   //如果看见了目标元素（targrt），则获取图表节
点
      imgElement.src = imgElement.getAttribute("data-src");   //修改自定义属性data-
src为常规的src属性
      observer.unobserve(imgElement);     //在图片加载出来后，停止观察元素以实现节流功能
    }
  }
})
Array.from(img).forEach(item => observer.observe(item));
```

11.浏览器的异步相关问题（详见JS文档）

A：defer和async的区别*

两者都是异步去加载外部JS文件，不会阻塞DOM解析

Async是在外部JS加载完成后，浏览器空闲时，Load事件触发前执行，标记为async的脚本并不保证按照指定他们的先后顺序执行，该属性对于内联脚本无作用 (即没有「src」属性的脚本）。

defer是在JS加载完成后，整个文档解析完成后，触发 `DOMContentLoaded` 事件前执行，如果缺少 `src` 属性（即内嵌脚本），该属性不应被使用，因为这种情况下它不起作用

B.DOMContentLoaded 与 load 的区别？

C：谈一谈你对requestAnimationFrame（rAF）理解

12.浏览器垃圾回收机制

由于字符串、对象和数组没有固定大小，所有当他们的大小已知时，才能对他们进行动态的存储分配。JavaScript程序每次创建字符串、数组或对象时，解释器都必须分配内存来存储那个实体。只要像这样动态地分配了内存，最终都要释放这些内存以便他们能够被再用，否则，JavaScript的解释器将会消耗完系统中所有可用的内存，造成系统崩溃。JavaScript的解释器可以检测到何时程序不再使用一个对象了，当他确定了一个对象是无用的时候，他就知道不再需要这个对象，可以把它所占用的内存释放掉了。

```
var a = "before";
var b = "override a";
var a = b; //重写a
```

这段代码运行之后，“before”这个字符串失去了引用（之前是被a引用），系统检测到这个事实之后，就会释放该字符串的存储空间以便这些空间可以被再利用。

浏览器通常用采用的垃圾回收有两种方法：标记清除、引用计数。

1)：标记清除

原理：这是JS中最常用的垃圾回收方式，当变量进入执行环境时，就标记这个变量为“进入环境”。从逻辑上讲，永远不能释放进入环境的变量所占用的内存，因为只要执行流进入相应的环境，就可能会用到他们。当变量离开环境时，则将其标记为“离开环境”。

垃圾收集齐首先会给所有变量添加标记，然后删除正在环境中的变量和被环境变量引用变量的标记。剩余有标记的变量被认为是要删除的变量，垃圾收集器销毁带标记的值，并回收他们占用的内存空间。

V8中的CG算法。。。。没看懂

2)：引用计次

原理：跟踪记录每个值被引用的次数，当一个变量A被赋值B时，这个值B的引用次数+1。相反，如果A又取得了另外一个值C，则这个值B的引用次数就-1。当引用次数变成0时，则说明没有办法再访问这个值了。但这个方法存在些许bug：

```
function problem() {
    var objA = new Object();
    var objB = new Object();

    objA.someOtherObject = objB;
    objB.anotherObject = objA;
}
```

在上述例子中，objA和objB通过各自的属性相互引用；也就是说这两个对象的引用次数都是2。在采用引用计数的策略中，由于函数执行之后，这两个对象都离开了作用域，函数执行完成之后，objA和objB还将会继续存在，因为他们的引用次数永远不会是0。这样的相互引用如果说很大量的存在就会导致大量的内存泄露。所以，**大多数浏览器已经放弃了这种回收方式。**

13.浏览器兼容问题

1：原因：因为不同的浏览器对同一段代码有不同的解析，造成页面显示效果不统一的情况。在大多数情况下，我们的需求是，无论用户用什么浏览器来查看我们的网站或者登陆我们的系统，都应该是统一的显示效果。版本越高的浏览器，支持的特性越多，我们用的某个插件使用的特性可能高版本的浏览器支持，低版本的不支持。

2：解决方法：

A: CSS HACK

CSS hack是通过在CSS样式中加入一些特殊的符号也就是浏览器前缀，让不同的浏览器识别不同的符号（什么样的浏览器识别什么样的符号是有标准的，CSS hack就是让你记住这个标准），以达到应用不同的CSS样式的目的

B: polyfill

polyfill 是一段代码(或者插件)，提供了那些开发者们希望浏览器原生提供支持的功能。程序库先检查浏览器是否支持某个API，如果不支持则加载对应的 polyfill。比如，html5的storage。不同浏览器，不同版本，有些支持，有些不支持。其实polyfill就是shim的一种。（Shim 指的是在一个旧的环境中模拟出一个新 API ，而且仅靠旧环境中已有的手段实现，以便所有的浏览 器具有相同的行为。）

C: PostCSS

PostCSS是一个利用JS插件来对CSS进行转换的工具，这些插件非常强大，强大到无所不能。其中，Autoprefixer就是众多PostCSS插件中最流行的一个。

Autoprefixer可以自动帮我们加上浏览器前缀。

D: Modernizr. js

Modernizr. js十分的强大，既能给老版本浏览器打补丁，又能保证新浏览器渐进增强的用户体验。Modernizr默认做的事情很少，除了（在你选择的情况下）给不支持html5的标签的浏览器，如IE6，7，8追加一点由Remy Sharp开发的html5垫片脚本，使其识别html5元素之外，它主要做的就是浏览器功能检测。因此，它知道浏览器是否支持各种html5和css3特性。

PS: Modernizr只是帮我们检测feature是否被支持，它并不能够给浏览器添加那些本来不支持的feature。

14.跨域问题

A：浏览器同源策略：

所谓同源是指：域名、协议、端口相同。核心就在于它认为自任何站点装载的信赖内容是不安全的。浏览器处于安全方面的考虑,只允许本域名下的接口交互, 不同源的客户端脚本,在没有明确授权的情况下,不能读写对方的资源。常用数据传输的方式有ajax与fetch。同源策略又分为以下两种：

1. DOM 同源策略：禁止对不同源页面 DOM 进行操作。这里主要场景是 iframe （iframe 元素会创建包含另外一个文档的内联框架（即行内框架））跨域的情况，不同域名的 iframe 是限制互相访问的。
2. XMLHttpRequest 同源策略：禁止使用 XHR 对象向不同源的服务器地址发起 HTTP 请求。

基于同源策略，**限制**了以下行为：

- Cookie、LocalStorage 和 IndexDB 无法读取
- DOM 和 JS 对象无法获取
- Ajax请求发送不出去

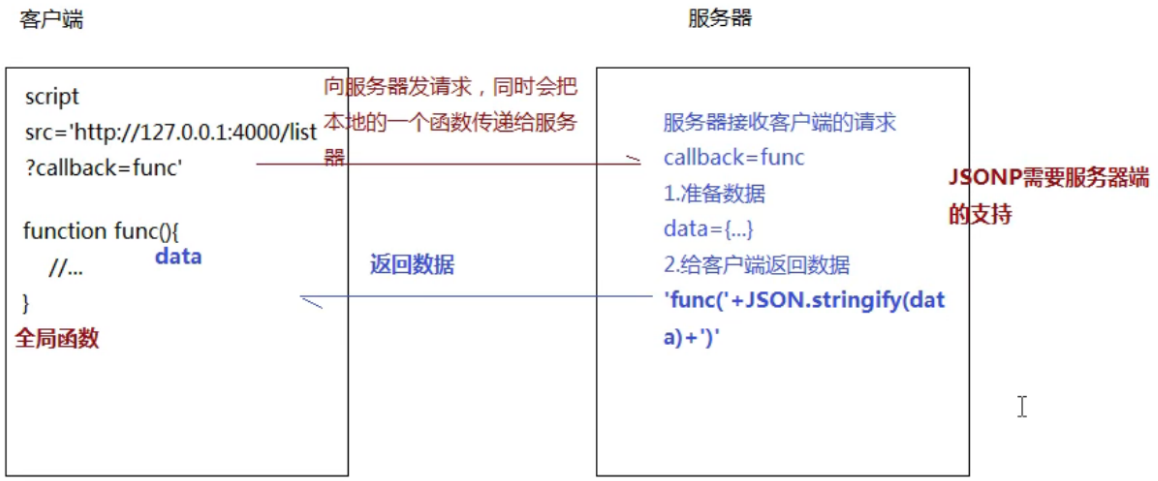
B：跨域的定义与解决方法：

跨域，是指浏览器不能执行其他网站的脚本。它是由浏览器的同源策略造成的，是浏览器对JavaScript实施的安全限制。

解决方法如下：

1.JSONP（过老，不常用）：

JSONP是一段参数是json格式(大多数情况)的JS代码。Web页面上调用js文件时不受跨域影响（不仅如此，别人还发现凡是拥有"src"这个属性的标签都拥有跨域的能力，比如script、img、iframe）。工作原理如下：




```
//客户端
$.ajax({
    url:'http://127.0.0.1:8001/list',
    method:'get',
    dataType:'jsonp',
    success: res=>{
        console.log(res)
    }
});

//服务端
let express = require('express'),
    app = express();
app.listen(8001, _ =>{                    // '_' 主要是为了占位
    console.log('ok!')
});
app.get('/list',(req,res) => {
    let {
        callback = Function.prototype  //设置默认值为一个空函数
    } = req.query;

    let data = {
        code:0,
        message:'test'
    };
```

```
    res.send(`${callback}(${JSON.stringify(data)    // 转义数据，  并发送给回调全局函数
中
    })`);
});
```

优点：不受同源策略的影响，兼容性更好，在古老的浏览器中皆可运行，不需要XMLHttpRequest或ActiveX的支持，并且在请求完成后可以通过调用callback的方式回传结果

缺点：只支持GET请求而不支持POST、PUT、DELETE等其他类型的HTTP请求。只支持跨域HTTP请求，不能解决不同域的两个页面之间如何进行JS调用的问题。有可能存在URL劫持问题，返回木马文件，即XSS攻击

2. CORS跨域资源共享（当前流行）：

CORS（Cross-Origin Resource Sharing）跨域资源共享，定义了必须在访问跨域资源时，浏览器与服务器应该如何沟通。CORS背后的基本思想就是**使用自定义的HTTP头部让浏览器与服务器进行沟通，从而决定请求或响应是应该成功还是失败。**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。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。（PS: CORS同时需要后端与浏览器的支持，只要后端实现了CORS，就实现了跨域）

浏览器看到服务器传回来的这个头部就知道能不能进行跨域请求了

注意：CORS分为简单请求与复杂请求，**简单请求**需满足以下条件：

1： 使用下列方法之一：GET、HEAD、POST

2：Content -Type的值仅限于下列三者之一： text/plain、multipart/form-data、application/x-www-form-urlencoded

复杂请求：

不符合以上条件的请求就肯定是复杂请求了。 复杂请求的CORS请求，会在正式通信之前，增加一次HTTP查询请求，称为"预检"请求,该请求是 option 方法的，通过该请求来知道服务端是否允许跨域请求。

```
// 一、基于node.js跨域
// 1. 客户端发送ajxa/fetch请求,首先进行axios默认配置
fetch( "http://目标资源网站", {methods:'GET'})
.then(response => response.json())
.then(data => console.log(data))

//2：服务器设置相关信息（需要处理options试探性请求--复杂请求）
app.use((req,res,next)=>{
```

```
    //允许哪一个源获取数据
    res.header("Access-Control-Allow-Origin","http://loaclhost:8000");
    //允许的客户段请求头，可以获取哪些数据
    res.header("Access-Control-Allow-Credentials","Content-
Type,Content,Length,Authorization                    , Accept,X-Requested-Width");
    // 允许携带cookie
    res.setHeader('Access-Control-Allow-Credentials', true)
    //运行客户端的请求方式，预检请求，查看客户端是否存在恶意方法methods等来操控数据
    res.header("Access-Control-Alloe-
Methods","PUT,POST,GET,DELETE.HEAD.OPTIONS");
    // 预检的存活时间
    res.setHeader('Access-Control-Max-Age', 6)
    // 允许返回的头
    res.setHeader('Access-Control-Expose-Headers', 'name')

    if(req.methods==='OPTIONS'){
        res.send('ok!');
        return;
    }
    next();
})
```

```
// python下的CORS

def after_request(response):
    # JS前端跨域支持，使用CORS方式
    response.headers['Cache-Control'] = 'no-cache'
    response.headers['Access-Control-Allow-Origin'] = '*'
    return response
```

JSONP与CORS的对比：

1: JSONP只能实现GET请求，而CORS支持所有类型的HTTP请求。

2: 使用CORS，开发者可以使用普通的XMLHttpRequest发起请求和获得数据，比起JSONP有更好的错误处理。

3: JSONP主要被老的浏览器支持，它们往往不支持CORS，而绝大多数现代浏览器都已经支持了CORS）

3. Nginx反向代理跨域：

Nginx 实现跨域请和proxy原理一致，只是Nginx帮我们做了服务器转发请求，我们在请求数据时,仍然写自己的网站的请求地址,而不是实际的请求网站的地址。

流程如下：

1：将自己的前端资源部署在Nginx的HTTP服务器

2：设置网站资源根目录

3：在 nginx.conf 配置文件中，设置监听端口与proxy_pass的转发规则

前端通过代理访问数据

```
server{
    listen          2222;
    server_name     localhost;      // 设置监听端口为http://localhost:2222

    location {
```

```
            root   /Users/relax/Desktop/9种跨域/nginx反向代理处理跨域/html;    //# 配置自己的静态网站路径,并指定网站的根目录
            index  index.html index.htm;
    }

    location /api{          //转发规则设置,也可使用正则表达式
        proxy_pass   http://127.0.0.1:3000
        add_header   Access-Control-Allow-Origin *;
    }
}
```

4. http proxy

CORS和JSONP解决跨域问题都有一个前提：**需要后台的支持**，Proxy可以解决这一问题。一般配合webpack和webpack-dev-server使用，具体流程如下：

1. 自己的网站端口号是: `http://127.0.0.1:12345`
2. 自己网站后台有一个 `/proxy` 的路由专门处理跨域数据请求.
3. 同时，有网站提供了一个数据接口,但是端口号是: `54321` -> `http://127.0.0.1:54321/data.json`，因为端口号不同，而产生了跨域问题
4. 在请求`data.json` 时，使用`/proxy`路由进行代理=> `http://127.0.0.1:12345/proxy?http://127.0.0.1:54321/data.json`
5. 经自己的服务器解析路径获取到真是的数据连接 `http://127.0.0.1:54321/data.json`
6. 自己的服务器由于没有同源策略的限制,所以可以直接发送这个请求,并或者数据返回值.
7. 最后经由自己的服务器返回给自己的前端浏览器

5. WebSocket协议跨域

Websocket是HTML5的一个持久化的协议，它实现了浏览器与服务器的全双工通信，同时也是跨域的一种解决方案。WebSocket和HTTP都是应用层协议，都基于 TCP 协议。但是 **「WebSocket 是一种双向通信协议，在建立连接之后，WebSocket 的 server 与 client 都能主动向对方发送或接收数据」**。同时，WebSocket 在建立连接时需要借助 HTTP 协议，连接建立好了之后 client 与 server 之间的双向通信就与 HTTP 无关了。

```
// 客户端
<script>
    let socket = new WebSocket('ws://localhost:3000');
    socket.onopen = function () {
      socket.send('我爱你');//向服务器发送数据
    }
    socket.onmessage = function (e) {
      console.log(e.data);//接收服务器返回的数据
    }
</script>

// 服务端
let WebSocket = require('ws'); //记得安装ws
let wss = new WebSocket.Server({port:3000});
wss.on('connection',function(ws) {
  ws.on('message', function (data) {
    console.log(data);
    ws.send('我不爱你')
  });
})
```

15.浏览器安全问题

1: XSS攻击

XSS 攻击是指浏览器中执行恶意脚本(无论是跨域还是同域)，从而拿到用户的信息并进行操作，主要分存储型、反射型和文档型。可以完成以下的坏事情，嘿嘿嘿：

a): 窃取Cookie

b):监听用户行为，比如输入账号密码后直接发送到黑客服务器

c): 修改DOM伪造登录表单

d): 在页面中生成浮窗广告

1.1 存储型XSS：

存储型的 XSS 将脚本存储到了服务端的数据库，然后在客户端执行这些脚本，从而达到攻击的效果。常见的场景是留言评论区提交一段脚本代码，如果前后端没有做好转义的工作，那评论内容存到了数据库，在页面渲染过程中`直接执行`，相当于执行一段未知逻辑的 JS 代码，是非常恐怖的。这就是存储型的 XSS 攻击。

1.2 反射型：

反射型是因为恶意脚本是通过作为网络请求的参数，经过服务器，然后再反射到HTML文档中，执行解析。直接看个链接，简单明了：

```
http://sanyuan.com?q=<script>alert("我是小坏蛋病毒，交出你的money~")</script>
```

浏览器将参数q中的内容当做html解析后，就被攻击了23333。

1.3 文档型：

作为中间人的角色，在数据传输过程劫持到网络数据包，然后**修改里面的数据**，也叫作**主动中间人攻击**（PS:被动中间人攻击只窃听，不修改数据）。主要的劫持方式包括**WIFI路由劫持**或**本地恶意软件**

1.4 防范手段

1. 不要相信用户的任何输出，都要对用户的输入进行**转码**或**过滤**，比如转义后，弹窗代码变为了：

```
&lt;script&gt;alert(&#39;我是小坏蛋病毒，交出你的money~&#39;)&lt;/script&gt;
```

无法被解析器执行

2. 利用CSP（浏览器中的内容安全策略），核心思想主要是由服务器决定浏览器加载哪些资源，具体来说可以完成以下功能:

   a): 限制其他域下的资源加载。

   b): 禁止向其它域提交数据。

   c): 提供上报机制，能帮助我们及时发现 XSS 攻击。

3. 利用HttpOnly：很多 XSS 攻击脚本都是用来窃取Cookie, 而设置 Cookie 的 HttpOnly 属性后，JavaScript 便无法读取 Cookie 的值。这样也能很好的防范 XSS 攻击。

2: CSRF攻击

CSRF(Cross-site request forgery), 即跨站请求伪造，指的是黑客诱导用户点击链接，打开黑客的网站，然后黑客利用用户**目前的登录状态**发起跨站请求。利用服务器的**验证漏洞**和**用户之前的登录状态**来模拟用户进行操作。利用用户登录状态，黑客可以做任何登录网站后的操作，比如银行转账等。主要有以下几种方式：

2.1 自动发GET请求 or诱导点击

黑客网页里面可能有一段这样的代码:

```
<img src="https://xxx.com/info?user=hhh&count=100">
```

进入页面后自动发送 get 请求，值得注意的是，这个请求会自动带上关于 http://xxx.com 的 cookie 信息(这里是假定你已经在 http://xxx.com 中登录过)。

假如服务器端没有相应的验证机制，它可能认为发请求的是一个正常的用户，因为携带了相应的 cookie，然后进行相应的各种操作，可以是转账汇款以及其他的恶意操作。

2.2 自动发送POST请求

黑客可能自己填了一个表单，写了一段自动提交的脚本。

```
<form id='hacker-form' action="https://xxx.com/info" method="POST">
  <input type="hidden" name="user" value="hhh" />
  <input type="hidden" name="count" value="100" />
</form>
<script>document.getElementById('hacker-form').submit();</script>
```

同样也会携带相应的用户 cookie 信息，让服务器误以为是一个正常的用户在操作，让各种恶意的操作变为可能。

这几种攻击方式的前提

- 目标站点一定要有 CSRF 漏洞
- 用户要登录过目标站点，并且在浏览器上保持有该站点的登录状态
- 需要用户打开一个第三方站点，可以是黑客的站点，也可以是一些论坛

2.3 Window.Opener（诱导点击）

带有**target="_blank"** 跳转的网页拥有了浏览器 `window.opener` 对象赋予的对原网页的**跳转权限**, 例如一个恶意网站在某 UGC 网站 Po 了其恶意网址，该 UGC 网站用户在新窗口打开页面时，恶意网站利用该漏洞将原 UGC 网站跳转到伪造的钓鱼页面，用户返回到原窗口时可能会忽视浏览器 URL 已发生了变化，`伪造页面` 即可进一步进行 `钓鱼` 或其他 `恶意行为`:

```
<a href="https://xxx/info?user=hhh&count=100" taget="_blank">点击进入修仙世界</a>
```

想象一下，你在浏览淘宝的时候，点击了网页聊天窗口的一条外链，出去看了一眼，回来之后淘宝网已经变成了另一个域名的 `高仿网站`，而你却未曾发觉，继续买东西把自己的钱直接打到骗子手里。

防范：为 target="_blank" 加上 rel="noopener noreferrer" 属性。

```
<a href="外跳的地址" rel="noopener noreferrer">外跳的地址a>
```

缺点: 为禁止了跳转带上 referrer，目标网址 `没办法检测来源地址`。

还有一种方法是，所有的外部链接都替换为内部的跳转连接服务，点击连接时，先跳到内部地址，再由服务器 redirect 到外部网址。

现在很多站点都是这么做的, 不仅可以规避风险，还可以控制非法站点的打开：

"/redirect?target=http%3A%2F%2Fxxx.yyy.com">

2.4 防范措施

1. 利用Cookie的SameSite属性

`CSRF攻击`中重要的一环就是自动发送目标站点下的 `Cookie`,然后就是这一份 Cookie 模拟了用户的身份。因此在`Cookie`上面下文章是防范的不二之选。恰好，在 Cookie 当中有一个关键的字段，可以对请求中 Cookie 的携带作一些限制，这个字段就是**SameSite**。SameSite可以设置为三个值，`Strict`、`Lax`和`None`。

```
**a.** 在`Strict`模式下，浏览器完全禁止第三方请求携带Cookie。比如请求`sanyuan.com`网站只能在`sanyuan.com`域名当中请求才能携带 Cookie，在其他网站请求都不能。

**b.** 在`Lax`模式，就宽松一点了，但是只能在 `get 方法提交表单`况或者`a 标签发送 get 请求`的情况下可以携带 Cookie，其他情况均不能。

**c.** 在`None`模式下，也就是默认模式，请求会自动携带上 Cookie。
```

2. 验证来源站点

这就需要要用到**请求头**中的两个字段: **Origin**和**Referer**。其中，**Origin**只包含域名信息，而**Referer**包含了具体的 URL 路径。当然，这两者都是可以伪造的，通过 Ajax 中自定义请求头即可，安全性略差。

3. Token

`Django`作为 Python 的一门后端框架，如果是用它开发过的同学就知道，在它的模板(template)中, 开发表单时，经常会附上这样一行代码:

```
{% csrf_token %}
```

这就是`CSRF Token`的典型应用。那它的原理是怎样的呢?

首先，浏览器向服务器发送请求时，服务器生成一个字符串，将其植入到返回的页面中。

然后浏览器如果要发送请求，就必须带上这个字符串，然后服务器来验证是否合法，如果不合法则不予响应。这个字符串也就是`CSRF Token`，通常第三方站点无法拿到这个 token, 因此也就是被服务器给拒绝

4 .其他

各个层级进行**权限验证**（例如现在的购物网站，只要涉及现金交易，肯定要输入密码或者扫码验证才行），也可以在敏感的接口尽量使用POST请求而不是GET

3. SQL注入

后台人员使用用户输入的数据来组装SQL查询语句的时候不做防范， 遇到一些恶意的输入， 最后生成的SQL就会有问题。比如：

```
// 返回id为1的文章
sql = "SELECT * FROM articles WHERE id =", $id

//恶意输入，1永远等于1，整个where相当于查询数据库中整张表，还有很多其他的SQL注入语句，在此不详细进行赘述
articlrs/index.php?id=-1 OR 1 = 1
```

现在的系统一般都会加入过滤和验证机制， 可以有效预防SQL注入问题。

4.点击劫持/界面操作劫持

也被成为UI-覆盖攻击，经常碰到，某些操作的按钮上加一层**透明的iframe**，点击一下， 就入坑了。防御有两种方法：

4.1. 使用 HTTP 头防御

通过配置 nginx 发送 **X-Frame-Options**响应头.这个 HTTP 响应头 就是为了防御用 iframe 嵌套的点击劫持攻击，使浏览器**阻止嵌入网页的渲染。**该响应头有三个值可选，分别是：

1. `DENY`，表示页面不允许通过 iframe 的方式展示。
2. `SAMEORIGIN`，表示页面可以在相同域名下通过 iframe 的方式展示。
3. `ALLOW-FROM`，表示页面可以在指定来源的 iframe 中展示。

4.2 使用 Javascript 防御

判断顶层视口的域名是不是和本页面的`域名一致`，如果不一致就让恶意网页自动跳转到我方的网页。

```
    if (top.location.hostname !== self.location.hostname) {
        alert("您正在访问不安全的页面，即将跳转到安全页面！")     //执意访问就没办法了23333
        top.location.href = self.location.href;
     }
```

5. 中间人攻击

中间人攻击（Man-in-the-Middle Attack, `MITM`）是一种由来已久的网络入侵手段.如 SMB 会话劫持、DNS 欺骗等攻击都是典型的MITM攻击。简而言之，所谓的MITM攻击就是**通过拦截正常的网络通信数据**，并进行**数据篡改**和**嗅探**来达到攻击的目的，而通信的双方却毫不知情。

5.1 如何防御中间人攻击

1. 确保当前你所访问的网站使用了HTTPS
2. 如果你是一个网站管理员，你应当执行HSTS协议
3. 不要在公共Wi-Fi上发送敏感数据
4. 如果你的网站使用了SSL，确保你禁用了不安全的SSL/TLS协议。
5. 不要点击恶意链接或电子邮件。