# 浏览器

## 16ms渲染帧

屏幕刷新率60HZ，1秒刷新60次，则1000/60=16ms

## 重绘和重排（回流）

* 重排：是改变DOM元素的位置和大小（即改变了几何属性）时，会导致浏览器重新生成渲染树，这个过程叫重排。
* 重绘：当重新生成渲染树后，就要将渲染树每个节点绘制到屏幕，这个过程叫重绘。

不是所有的动作都会导致重排，比如改变字体颜色，只会导致重绘。

重排会导致重绘，重绘不会导致重排

重绘和重排这两个操作都是非常昂贵的，因为JavaScript引擎线程和GUI渲染线程是互斥的，他们同时只能一个在工作。

**如何减少重排和重绘？**

1. 用JavaScript修改样式时，最好不要直接写样式，而是替换class来改变样式
2. 如果要将DOM元素执行一系列操作，可以将DOM元素脱离文档流，修改完成后，再将它带回文档。

推荐使用隐藏元素（display：none）或文档碎片（DocumentFragement），都能很好的实现这个方案。

## 进程和线程

多线程可以并行处理任务，但是线程不能单独存在，它是有进程来启动和管理的；

一个进程就是一个程序的运行实例。启动程序的时候，操作系统会为该程序创建一块内存，用来存放代码，运行中的数据和一个执行任务的主线程，我们把这样的一个环境叫线程。

线程是依附于进程的，而进程中使用多线程并行处理能提升运算效率。

进程和线程之间的关系有如下4个特点：

1. 进程中的任意一线程执行出错，都会导致整个进程的崩溃；
2. 线程之间共享进程中的数据；
3. 当一个进程关闭之后，操作系统会回收进程所占用的内存
4. 进程之间的内容相互隔离

## 浏览器渲染机制

**9.1 浏览器是多进程的，包含哪些进程？**

* 浏览器主线程
* 渲染进程
* GPU进程
* 网络进程
* 插件进程

**9.2 浏览器的渲染进程是多线程的，包含哪些线程？**

* **GUI渲染线程**
* **JS引擎线程**
* **事件触发线程**
* **定时触发器线程**
* **异步http请求线程**

**9.3 浏览器的渲染流程**

## **渲染引擎和JavaScript引擎是怎么通信的**

## **常见的浏览器兼容性问题**

1. **不同浏览器的默认margin和padding不一样**
2. **图片的默认间距不一样**
3. **获取视口的宽高window.innerHeight/width**
4. **css的动画、过渡、渐变、flex也有，grid**
5. **Canvas、SVG**
6. **IE9以下不能的opacity，使用filter：alpha（opacity = 50）**
7. **event.offsetX/Y**
8. **绑定事件（addEventListener）IE9才支持**

## **web安全XSS、CSRF**

xss：跨站脚本攻击，xss的重点不在于跨站点，而在于脚本的执行

原理：恶意攻击者在web页面中会插入一些恶意的script代码；

当用户浏览该页面的时候，那么嵌入在web页面的script代码会执行；

以此达到恶意攻击用户的目的。

分类：

反射型XSS：在url后面的参数中加入恶意攻击代码

存储型XSS：将恶意代码存储在服务器上

防范：后端需要对提交的数据进行过滤

DOM-based型XSS：客户端的js对页面dom节点进行动态操作

SQL注入：

防范：cookie安全策略

http-only：只允许http或https请求读取cookie，js代码是无法读取cookie的；

secure-only：只允许https请求读取 [sɪˈkjʊə(r)]

host-only：只允许主机域名和domain设置完成一致的网站才能访问该cookie

防御HTML、HTML Attribute、javascript、URL、CSS编码

CSRF：跨域请求伪造； 用户登录一个正常的网站后，由于没有退出该正常网站，cookie信息还保留，然后用户去点击一个危险的网站页面，那么这个时候危险网站就可以拿到你之前登录的cookie信息，然后使用cookie信息去做其他事情。

防范：

验证HTTP Referer字段：判断来源域名是否是本地网站

加验证码

使用Token

CSRF攻击并没有违反同源策略

网页不能请求不同源服务器上的资源，也不能读取不同源网页的cookie、localStorage中的数据，但是以下资源的请求不受同源策略的限制：

* <script src=”…”></script>
* <link rel=”stylesheet” href=”…”>
* <img src=”…” />
* <video>和<audio>嵌入的多媒体资源
* <object>、<embed>和<applet>的插件
* @font-face引入的字体。一般浏览器允许跨域字体(cross-origin fonts),一些需要同源字体(same-origin fonts)
* <frame>和<iframe>载入的任何资源(可以使用X-Frame-Options阻止这种形式的跨域交互)

我们平常说的最多的跨域受限的方式是采用ajax的显示请求数据。

而CRSF可以利用上叙不受跨域资源影响的标签对目标服务器进行攻击。

## **浏览器缓存策略（强缓存、协商缓存）**

<https://www.cnblogs.com/bill-shooting/archive/2018/07/21/9347441.html>

**Web缓存的好处**

* 减少网络延迟，加快页面打开速度
* 减少网络带宽消耗
* 降低服务器压力
* ……

**Etag与IF-None-Match**

* Etag与IF-None-Match是一对报文头，属于HTTP 1.1
* Etag是一个响应首部字段，它是根据实体内容生成的一段hash字符串，标识资源的状态，由服务器产生。
* IF-None-Match是一个条件式的请求首部字段。如果请求资源时在请求首部加上这个字段，值为之前服务器返回的Etag。

**Etag能解决什么问题？**

1. Last-modified标注的最后修改时间只能精确到秒级，如果某些文件在1秒钟以内，被修改多次的话，它将不能准确标注文件的新鲜度；
2. 某些文件也许会周期性的更改，但是它的内容不改变（仅仅改变的是修改时间），但last-modified却改变了，导致文件没法使用缓存。
3. 有可能存在服务器没有准确获取文件修改时间，或者与代理服务器时间不一致等情形。

**缓存位置**

浏览器可以再内存、硬盘中开辟一个空间用于保存请求资源副本。

请求一个资源时，会按照优先级（service worker -> memory cache -> disk cache -> push cache）依次查找缓存。

**200 from memory cache**

表示不访问服务器，直接从内存中读取缓存。因为缓存的资源保存在内存中，所以读取速度快，但是关闭进程后，缓存资源也会随之销毁，一般来说，系统不会给内存分配比较大的容量，因此内存缓存一般用于存储较小的文件。同是内存缓存再有效性要求的场景下也很有用（比如浏览器的隐私模式）。

**200 from disk cache**

**C:\Users\shuanggui.yu\AppData\Local\Google\Chrome\User Data\Default\Cache**

表示不访问服务器，直接从硬盘中读取缓存。与内存相比，硬盘的读取速度相对较慢，但硬盘缓存持续的时间更长，关闭进程之后，缓存的资源依然存在。由于硬盘的容量较大，因此一般用于存储大文件。

**200 from prefetch cache**

在preload或prefetch的资源加载时，两者也是均存储在http cache，当资源加载完成后，如果资源是可以缓存的，那么其被存储在http cache中等待后续使用，如果资源不可被缓存，那么其在被使用前均存储在memory cache。

**CDN Cache**

* **浏览器缓存**

当浏览器访问过后的资源，会被浏览器缓存在本地，当下次再访问页面的时候，如果没有过期，就直接读取资源，加快浏览器的加载速度

http缓存机制：

1. Expires：设置最大缓存时间，当时间超过了就去服务器下载
2. http1.1 ，cache-control： max-age =time ， 当time过期后，检测etag 带上etag往服务器发请求，如果etag没变，直接告诉浏览器读本地缓存， 如果没有etag就会检测last-modified，判断 如果上一次更改的时候，距离本次访问时间比较久，说明文件没有发生变化，返回304

* **协商缓存和强制缓存**

协商缓存的标识有2种：ETag/if-None-Match和Last-Modified/if-Modify-Since

浏览器第一次请求一个资源的时候，服务器会告知浏览器一个最后修改时间，当浏览器再次请求的时候，request的请求头会加上if-Modify-Since，服务器收到浏览器发送来的最后修改时间和当前自己的最后修改时间进行对比，一致的话就会返回304，否则，当前自己的最后修改时间大于浏览器携带的最后时间，将返回200，并给浏览器传递新的最后修改时间。

Etag是唯一标识，

文中"只要资源有变化，ETag会重新生成的"。也不能说作者大大说错。而是不严谨。Etag生成规则是由服务器的生成算法决定的，像nginx服务器的Etag生成规则就是由response的 Last-Modified 与 Content-Length 表示为十六进制组合而成，而last-modified又是由mtime生成。当编辑文件但是没修改内容时资源是没有变化的，但是此时mtime已经改变所以Etag也会跟着改变，但是资源文件并没有改变，Etag也会重新生成

协商缓存有两种，一种是Last-Modified，就是第一次请求资源的时候，服务器会在响应头里面设置该字段，表示该资源的最后修改时间，浏览器再第二次请求该资源的时候，会在请求头里面加上一个字段If-Modified-Since，值为第一次请求的时候服务器返回的Last-Modified的值，服务器会判断当前资源的最后修改时间和请求头里面的If-Modified-Since字段是否相同，如果相同，则告诉客户端使用缓存，否则重新下载资源。

另外一种协商缓存是使用Etag，原理与Last-Modified类似，就是第一次请求的时候，服务器会根据资源的内容或者最好修改时间生成一个标识，然后再响应头里面设置Etag返回给客户端，客户端第二次请求的时候会在请求头里面带上这个Etag，也就是再请求头里面加上If-None-Match字段，服务器接收到Etag之后会判断是否与原来第一次的标识相同，如果相同，则告诉客户端使用缓存。

强制缓存：

浏览器在第一次请求资源的时候，服务器响应头里可以设置expires字段，该字段表示该资源的缓存过期时间，第二次请求的时候，如果时间还在该缓存时间之内，则会直接使用缓存，否则重新加载资源。这个expires字段有一个缺陷，就是它必须服务器和客户端的时间严格同步才能生效，所以现在很多人不会使用该方案，另外一种方案是第一次请求资源的时候，服务器设置响应头cache-control：max-age，这样设置的意思是告诉浏览器，这个资源什么时候过期，等第二次请求资源的时候，判断是否超过过期时间，如果没有超出，直接使用缓存。

Cache-control是服务器设置的。

cache-control的其他值： public 客户端和服务端都可以缓存

private 只能客服端缓存

no-store 不使用缓存

no-cache 使用协商缓存

# **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；

## 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
6. 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、

## cache缓存策略、

## 长链接Connection: keep-alive、多路复用

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

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，服务器就可以并行的传输数据。

HTTP2对统一域名下所有请求都是基于流，也就是说统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

## **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

## 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。

# Html

## Contenteditable与textarea

区别：contenteditbale随着内容增加自动增高；textarea不能自动增高，需要配合js监听文本高度的变化来实现；

## line-height与height

## Node和Element的区别

Element是node的一种element\_node

Node是一个基类，DOM中的Element、Text、Comment都继承它

**parentNode 和 parentElement**

parentNode是w3c标准，parentElement只ie支持

当父节点的nodeType不是1，即不是element节点的话，他的parentElement就会是null

一般情况parentNode可以取代parentElement的所有功能

ParentElement匹配的是parent为element的情况，而parentNode匹配的是parent为node的情况。Element是包含在node里的，它的nodeType是1。

## transition

<transition  name="move" mode="out-in">

</transition>

.move-enter-active, .move-leave-active {

  transition: opacity .5s

}

.move-enter, .move-leave-active {

  opacity: 0

}

## clientHeight、offsetHeight、scrollHeight、offsetTop、scrollTop

scrollHeight：网页正文全文高

clientHeight：网页可见区域高；包括padding；不包括border、水平滚动条、margin

offsetHeight：网页可见区域，包括padding、border、水平滚动条、margin

scrollTop：网页被卷去的高

offsetTop：当前元素顶部具体最近父元素顶部的距离，和滚动没有关系

## script标签的defer和async

async：可选属性，表示应该立即下载脚本，但不应该妨碍页面中的其他操作，比如下载其他资源和等待加载其他脚本。只对外部脚本文件生效。

Defer：可选属性，表示脚本可以延迟到文档完全被解析和显示之后再执行。只对外部脚本文件有效。

# CSS

## \*\*\*盒子模型

* margin 外边距
* border 边框
* padding 内边距
* content（内容区域）

width属性等于是content的宽度

总宽度 = 左右外边距 + 左右边框 + 左右内边距 + 内容宽度

总高度 = 上线外边距 + 上下边框 + 上下内边距 + 内容高度

IE5.X和6的width 属性等于 内边距+边框+content的宽度

## \*\*\*Flex弹性布局

布局的传统解决方案，基于盒装模型，依赖display属性 + position属性 + float属性。它对于那些特殊布局非常不方便，比如，垂直居中就不容易实现。

使用flex布局以后，子元素的float、clear和vertical-align属性将失效。

主轴：X轴

交叉轴：Y轴

**Flex容器的属性**

**1、flex-directio**n属性决定X周的方向，row | row-reverse | column | column-reverse

**2、flex-wrap** 默认情况下，项目都排在一条线上，如果一条轴线排不下，如何换行。

nowrap（默认），不换行

wrap 换行，第一行再上方

wrap-reverse 换行，第一行在下方

**3、flex-flow** 是flex-direction和flex-wrap的简写形式，默认值为row || nowrap

**4、justify-content** 定义项目再主轴上的对齐方式

flex-start（默认值）左对齐

flex-end 右对齐

center 居中

space-between 两端对齐，项目之间的间隔都相等

space-around 每个项目两侧的间隔相等。

所以，项目之间的间隔比项目与边框的间隔大一倍。

**5、align-items** 定义项目在y轴上如何对齐

flex-start 起点对齐

flex-end 终点对齐

center 终点对齐

baseline 项目的第一行文字的基线对齐

stretch（默认值）如果项目未设置高度或者设置为auto，将占满整个容器的高度

**6、aligin-content** 定义了很多轴线的对齐方式。

如果项目只有一根轴线，该属性不起作用。

**Flex项目的属性**

1. **order** 定义项目的排列顺序，数值越小，排列越靠前，默认为0
2. **flex-grow** 定义项目的放大比例，默认为0，即如果存在剩余空间，也不放大

如果所有项目的flex-grow属性都为1，则它们将等分剩余空间（如果有的话）；

如果一个项目的flex-grow属性为2，其他都为1，则前者占据的剩余空间比其他项多一倍。

1. **flex-shrink** 定义项目的缩小比例，默认为1，即如果空间不足，将项目缩小
2. **flex-basis** 定义了在分配多余空间之前，项目占据的主轴空间。浏览器根据这个属性，计算主轴是否有多余空间。它的默认值是auto，即项目的未来大小。

它可以设置为跟width或height属性一样的值（比如350px），则项目将占据固定空间。

1. **flex** 是flex-grow、flex-shrink和flex-basis的缩写，默认值为0 1 auto。后两个属性可选
2. **flex-self** 允许单个项目有与其他项目不一样的水平对齐方式，可以覆盖align-items属性。

默认为auto，表示继承父元素的align-items属性。

如果没有父元素，则等同于stretch

## flex什么时候不相对浏览器定位？

当元素祖先的transform属性非none时，定位容器有视口改为该祖先。

并不是所有能够生成层叠上下文的元素都会是的position:fixed失效，但也不止transform

## 如何触发一个元素形成层叠上下文

1. 根元素<html>
2. z-index值不为auto的绝对和相对定位
3. 一个z-index值不为auto的flex项目（flex item），即父元素display：flex|inline-flex
4. opacity [əʊˈpæsəti] 属性值小于1的元素
5. transform属性值不为none的元素
6. mix-blend-mode [mɪks] [məʊd] 属性值不为normal的元素

mix-blend-mode属性设置图片元素与父容器背景（黄色）进行混合，图片会变暗

1. filter值不为none的元素

filter属性定义了元素（通常是img的）可视效果（如：模糊与饱和度）

1. position：fixed
2. ……..

## px、em、rem，VW，VH

em相对长度单位，相对于当前对象内文字的字体尺寸，如当前对行内文字的字体尺寸未被人为设置，则相对于浏览器的默认字体尺寸。

rem是css3新增的一个相对单位（root em，根em），rem为元素设定字体大小时，依然是相对大小，但相对的只是HTML根元素。

VM：视口的宽度为100VM，相对于把视口分为100份；

VH：视口的高度为100VH，同理。

## 交互过程中布局layout [ˈleɪaʊt]的优化

## Js执行的优化

## line-height：2 /2em/2%

浏览器有一个默认文字大小是16px，

 1、line-height:2em、line-height:200%

　　　  根据父元素的字体大小（font-size）计算行高

         子元素继承父元素计算后的行高，如果子元素的字体很大，而父元素字体小，这样计算出来的行高有可能无法容量子元素的显示，会被截掉(或是重叠)。

     2、line-height:2（推荐）

　　    根据子元素的font-size计算行高（取最大字体值\*2）

如何父元素和子元素都没有设置font-size，就按照浏览器默认的16px\*2

## \*\*\*响应式布局实现方案

* **媒体查询**

css3媒体查询可以让我们针对不同的媒体类型定义不同的样式。

当你重置浏览器大小的过程中，页面也会根据浏览器的宽度和高度重新渲染页面。

* **百分比布局**
* **rem布局**

rem是css3新增的单位

rem单位是相对于根元素html的font-size来决定大小的。

根据视图容器的大小，动态的改变font-size即可。

rem布局是目前多屏幕适配的最佳方式。

**动态改变font-size的方式：**

1. 媒体查询方式
2. Js动态计算
3. Scss计算，指定rem基准值为37.5px，也有设置基准值为75px的

|  |
| --- |
| @function px2rem($px) {     $rem: 37.5px;      @return ($px / $rem) + rem;  } |

* **视口单位**

css中引入vm/vh，与视图窗口有关

vm表示相对于视图的宽度；vh表示相对于视图窗口的高度。

Vmin vw和vh中的较小值；Vmax vw和vh中的较大值

vw和vh很类似百分比单位，区别是%大部分相对于祖先元素，也有相对于自身的情况，比如border-radius、translate等；vm、vh是相对于视口的尺寸

* **图片响应式**

## 移动端适配

<https://www.runoob.com/w3cnote/moible-web-front-source.html>

meta篇

1、视图宽度

<meta name="viewport" content="width=device-width,initial-scale=1.0,minimum-scale=1.0,maximum-scale=1.0,user-scalable=no"/>

width=device-width 设置视图宽度为设备视图宽度，还可以固定宽度，例如：width=640则是640px的宽度（常见于微信）；

initial-scale=1.0 设置缩放比例为1.0

minimum-scale=1.0 和maximum-scale=1.0 最小缩放比例和最大缩放比例

user-scalable=no 禁止用户自由缩放，默认值为yes

提示：刚刚那个是带全部参数的，一般常用的，有user-scalable=no就不用使用minimum-scale=1.0和maximum-scale=1.0来强制禁止缩放了

2、自动识别格式

<meta name="format-detection" content="telephone=no"/>

Content里面的参数：telephone=no是禁止浏览器自动识别手机号码；email=no是禁止浏览器自动识别email。

Onorientationchange 手机屏幕旋转

Rezie 浏览器窗口缩放

document.documentElement 返回的是<html>……..</html>

document.documentElement.getBoundingClientRect().width （html的宽）

window.screen.width（设备的宽）

window.getComputedStyle(document.documentElement, null).getPropertyValue('font-size')

获取html标签设置的font-size

当浏览器没有最大化的时候，html的宽度小于设备的宽度

大多数浏览器的默认字号是16px，所以1rem=16px

设计稿750

Flexible将设计稿分为10份，可以理解为页面的width=10rem，即1rem=75px，所以根font-size（基准值）=75px

返利根节点data-screenwidth和font-size的设置

data-screenwidth（html的宽度）

data-screenwidth > 450时，font-size=60

data-screenwidth < 450时，font-size= data-screenwidth /750 \* 100

定时器实时监听屏幕大小，然后重新设置屏幕宽度和font-size

返利默认设置body的font-size=14px

当根html的font-size = 60px

则相当于1rem=60px

当设置某一个div的font-size=0.25rem时，font-size换算后为60\*0.25=15px

## Css样式权重

 !important > 行内 > 头部 > 引入

## line-height/font-size/height

## css伪类

input:checked 所有选中的表单元素

input:disabled 所有禁用的表单元素

input:enabled 所有启用的表单元素

input:in-range 元素指定范围内的值

input-invalid 无效的元素

p:empty 没有子元素的p元素

p:first-of-type 第一个p元素

p:last-child p元素下的最后一个子元素

## css三大预处理器（Sass、Less、stylus）

预处理器是一种用来为CSS增加一些编程特性的语言，无需考虑浏览器的兼容问题。

## BFC

## \*\*清除浮动的方法

[**https://www.cnblogs.com/plsmile/p/11125641.html**](https://www.cnblogs.com/plsmile/p/11125641.html)

标准流：盒子会各占整行位置。子盒子若是标准流，父盒子虽然没有高度，但是会撑开父盒子高度。

浮动：盒子浮起来，不会占据原来的位置，若父盒子没有定义高度，则不会撑开父盒子，父盒子高度为0。

浮动可以让多个块级元素在一行显示，且块与块之间没有间隙，但要注意给父盒子清除浮动，否则父盒子不会被撑开。

**为什么要清楚浮动呢？清除浮动的本质是什么?**

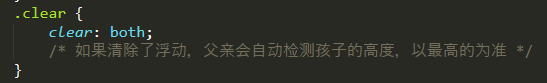
清除浮动主要是为了解决父级元素因为子级浮动引起的内部高度为0的问题。

**清除浮动的方法：**

* 额外标签法：给谁清除浮动，就在其后额外添加一个空白标签，空白标签添加样式clear:both

优点：通俗易懂，书写方便（不推荐使用）

缺点：添加需要无意义的标签，结构化比较差



* 父级添加overflow方法：可以通过触发BFC的方式，实现清除浮动效果。

优点：代码简洁（慎重使用，若该父盒子里还有position定位会引起麻烦）

缺点：内容增多的时候容易造成不会自动换行导致内容被隐藏掉，无法显示需要溢出的元素

Overflow:hidden;

注意：别加错位置，是给父亲加（并不是所有的浮动都需要清除，谁影响布局，才清除谁。）

* 使用after伪元素清除浮动

After方式是空元素的升级版，好处是不用单独加标签了。（较常用）

优点：符合闭合浮动思想，结构语义化正确

缺点：由于IE6-7不支持after，使用zoom：1触发hasLayout



* 使用before和after双伪元素清除浮动（较常用）



## \*\*\*垂直居中的实现

* **对单行元素进行垂直居中**

1、设置line-height等于父元素的height

* **对文本进行垂直居中**

设置display：flex和align-items：center实现文本居中

* **对已知高度块级元素进行垂直居中**

1. 绝对定位，配合top:50%和margin-top（元素高度一半）进行垂直居中
2. 绝对定位，配合top:0; bottom:0; 和margin:auto 进行垂直居中 margin [ˈmɑːdʒɪn]
3. 绝对定位，配合top:calc(50% - 元素本身高度一半) 函数进行垂直居中
4. ~~使用浮动float实现元素垂直居中~~

~~原理：通过再要进行垂直居中的元素a前面添加一个无内容的元素，并将该无内容元素的高设置为50%，再利用clear:both清除浮动，则元素a相对于父元素来说是垂直居中。~~

* **对未知高度块级元素进行垂直居中**

1. 绝对定位，配合top:50%和transform:translate(x,y)实现水平垂直居中

transform:translate(-50%,-50%); 针对元素本身向左以及向上移动50%

1. 居于视口单位的解决办法

通过使用margin-top:50vh配合transform:translateY(-50%),实现视口居中

1. 通过display:table-cell和vertical-aligin:middle，实现垂直居中

设置父块display:table，子块display:table-cell，vertical-aligin：middle

1. 基于flex的实现方案

设置父块display：flex，子块margin:auto

## transform

css transform属性允许你旋转、缩放、倾斜或平移给定的元素。

|  |  |  |
| --- | --- | --- |
| translate | 转换、平移 | 3D，x，y，z |
| scale [skeɪl] | 缩放 | 3D，x，y，z |
| rotate [rəʊˈteɪt] | 旋转 | 3D，x，y，z |
| skew [skjuː] | 倾斜 | x，y |
| perspective [pəˈspektɪv] | 3D元素定义透视视图 |  |

应用：

1. 画出小于1px的直线

transform: scaleY(0.2);

1. 实现box居中

 #box{

            width: 300px;

            background-color: green;

            top: 50%;

            transform: translateY(-50%);

            position: fixed;

        }

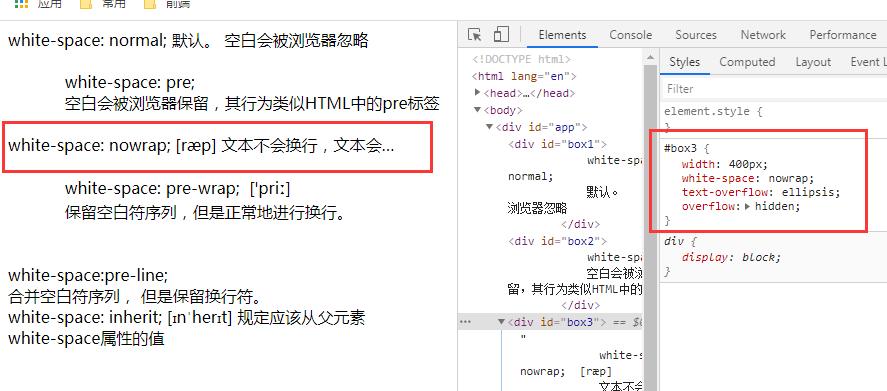
## white-space

white-space属性设置如何处理元素内的空白

## text-overflow

text-overflow属性规定当文本溢出包含元素时发生的事情。

## 一行显示多余显示省略号



## Zoom

Zoom这个属性是ie的专有属性，除了设置或者检索对象的缩放比例之外，它还可以触发ie的haslayout属性，清除浮动，清除margin重叠等作用。

## ie的haslayout属性

## link和@import的区别

# JavaScript

## Js执行机制

Js是单线程语言，同一时间内只能干一件事情；

**为什么JS不能是多线程？**

是否多线程取决于这个语言的用途。

一个简单的例子，如果同一时间，一个添加DOM，一个删除了DOM，这时候语言就不知道是该添还是删了。

**同步：**在主线程上排队执行的任务，只有前一个任务执行完毕，才能执行后一个任务。

**异步：**不进入主线程，而进入“任务队列”（task queue）的任务，只有“任务队列”通知主线程，该任务才会进入主线程执行

**运行机制**

1. 所有任务都在主线程上执行，形成一个执行栈
2. 主线程之外，还存在一个“任务队列”。只要异步任务有一个运行结果，就在“任务队列”之中放置一个事件。
3. 一旦“执行栈”中的所有同步任务执行完毕，系统就会读取“任务队列”，看看里面有哪些事件。于是那些对应的异步任务结束等待状态，进入执行栈，开始执行。
4. 主线程不断重复上面三步。

## 事件循环

js代码可以分为同步代码和异步代码；

Js引擎在执行代码时，首先会将同步代码加入到主线程中执行，形成一个执行栈；异步代码会放到event table中注册回调函数；

当异步任务处理完成时，JS引擎将异步任务的回调函数从event table移到“任务队列”中；

当主线程执行栈中的同步任务执行完毕，js引擎会从“任务队列”中读取最先进入队列中的任务，放入执行栈执行；

JS任务又可以分为宏任务和微任务

宏任务进入宏任务队列，每一个宏任务对应一个微任务队列；

宏任务有：主代码快、定时器（setTimeout、setInterval）、请求动画帧、http请求、IO请求等；

微任务有：promise.then()、promise.catch()等

一个宏任务执行过程中，遇到微任务会将其放入对应的微任务队列中；当这个宏任务执行完成，会将对应的微任务队列中所有的任务执行完成； 然后再去宏任务队列获取下一个宏任务。

SetTimeOut()的执行原理：setTimeOut第一个参数是回调函数，第二个参数是指定多少毫秒后执行这个回调函数；

首先：将这个回调函数放入event table中，

浏览器中有一个定时器用于计算等待时间，当到达了指定的时间，将回调函数移动到宏任务队列中。

http请求的执行过程：

首先将http请求任务放入event table，并注册回调函数；

浏览器主线程将http请求交给网络请求；

网络请求拿到http请求后，

首先判断浏览器是否有对它的返回结果做缓存，如果有，直接从缓存中获取返回结果交给浏览器主线程；

如果没有开始DNS解析

DNS解析完，开启TCP三次握手

三次握手成功，开始正式的请求；

服务器确认请求，并返回请求结果

再四次挥手；

网络请求拿到返回结果后，告知浏览器我已成功得到请求结果

浏览器为其开闭通道，将数据传给浏览器主线程；

主线程将http请求回调函数推入任务队列

## **JS变量提升**

函数和变量的声明都将被解释器提升到函数的最顶部

变量可以先使用再声明

变量初始化不会提升

## **作用域和作用域链**

作用域是可访问变量的集合。

在JavaScript中，对象和函数同样也是变量；

在JavaScript中，作用域为可访问变量、对象、函数的集合；

**局部作用域**

变量在函数内声明，变量为局部作用域；

局部变量：只能在函数内部访问；

局部变量在函数开始执行时创建，函数执行完后局部变量会自动销毁。

**全局作用域**

变量在函数外定义，即为全局变量

全局变量有全局作用域，网页中所有的脚本和函数均可使用；

如果变量在函数中没有声明（没有使用var关键字），该变量为全局变量。

**块级作用域**

ES6之前，没有块级作用域的概念

ES6可以使用let关键字来实现块级作用域

Let声明的变量只在let命令所在的代码块{}内有效，在{}之外不能访问

**变量生命周期**

Js变量生命周期在它声明时初始化；

局部变量在函数执行完毕后销毁；

全局变量在页面关闭后销毁。

**作用域链**

一般情况下，变量是在定义这个变量的函数作用域中取值，

但是如果在当前作用域中没有查到值，就会向上级作用域去查，直到查到全局作用域，这么一个查找过程形成的链条就叫做作用域链。

## this

this的指向在函数定义的时候是确定不了的，只有函数执行的时候才能确定this到底执行谁，实际上this的最终指向的是哪个调用它的对象。

this是当前执行上下文（global、function或eval）的一个属性，在非严格模式下，总是指向一个对象，在严格模式下可以是任意值。

* 全局上下文

无论是否在严格模式下，在全局执行环境中（在任何函数体外部）this都指向全局对象。

* 函数上下文

在函数内部，this的值取决于函数被调用的方式。

* 类上下文

在类的构造函数中，this是一个常规对象。

* 箭头函数中的this

定义时绑定this，继承的是父执行上下文里面的this

* 作为对象方法

当函数作为对象里的方法被调用，this被设置为调用该函数的对象。

* 构造函数中的this

当一个函数用作构造函数（使用new关键字），它的this被绑定到正在构建的新对象。

* 作为一个DOM事件处理函数

当函数被用作事件处理函数时，它的this指向触发事件的一个元素。

普通函数的this：指向它的调用者，如果没有调用者则默认指向window

箭头函数的this：指向箭头函数定义时所处的对象，而不是箭头函数使用时所在的对象，默认使用父级的this。

## Call()、apply()、bind()

## 继承

ES5实现继承的方式

* **构造函数实现继承**

原理：在子类的构造函数中指向父类的构造函数，并为其绑定子类的this，让父类的构造函数把成员属性和方法都挂到子类的this上去

缺点：继承不到父类原型上的属性和方法

* **原型链实现继承**

原理：将子类的原型对象指向父类的实例

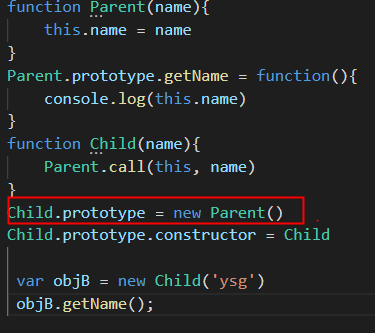
缺点：无法向父类构造函数传参

* **组合式继承**

原理：组合原型链继承和构造函数继承

用构造函数定义属性，用原型定义方法。

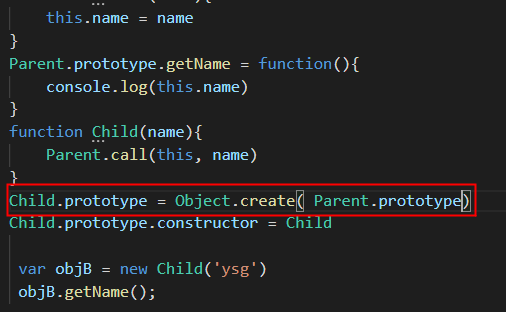
缺点：每次创建子类实例都执行了两次构造函数（Parent.call()和new Parent()），虽然这并不影响对父类的继承，但子类创建实例时，原型中会存在两份相同的属性和方法，这并不优雅。



* **寄生式组合继承**

为了解决构造函数被执行两次的问题，我们将执行父类实例改为指向父类原型，减去一次构造函数的执行。

但这种方式存在一个问题，由于子类原型和父类原型指向同一个对象，我们对子类原型的操作会影响到父类原型，为解决这个问题，我们给父类原型做一个浅拷贝。



寄生组合式继承，是目前ES5最成熟的继承方式，babel对ES6继承的转化也是使用寄生组合式继承。

## 原型和原型链

**普通对象和函数对象**

在javascript的世界中，全部都是对象，而对象之间也是存在区别的，我们一般区分普通对象和函数对象。

**原型**

原型也是一个对象，原型的作用，是实现对象的继承。

所有的js对象，都存在一个函数指针的属性，该属性指向实例对象的构造函数的原型。

Js的函数对象也是对象，所以它有一个原型指针，；除此之外，js的函数对象也有自己的原型。

普通对象的\_\_proto\_\_指向Object的prototype

函数对象的\_\_proto\_\_指向Function的prototype

函数对象的prototype的\_\_proto\_\_指向Object的prototype

Object的\_\_proto\_\_指向Function的prototype

Object的prototype的\_\_proto\_\_指向null

Function的\_\_proto\_\_指向Function的prototype

Function的prototype的\_\_proto\_\_指向Object的prototype

**原型链**

## Let、const和var区别

在HTML中，全局作用域是针对window对象；

使用var关键字声明的全局作用域变量属于window对象；

使用let关键字声明的全局作用域变量不属于window对象，使用window. 不能访问

**Let和const的异同点**

**相同点：**

二者都是块级作用域

都不能和它所有作用域内的其他变量或者函数拥有相同的名称

**区别：**

Const声明的常量必须初始化，而let声明的变量不用

Const定义变量的值不能通过再复制修改，也不能再次声明。而let定义的变量值可以修改

**Const并非真正的常量**

Const的本质：cost定义的变量并非常量，并非不可变，它定义了一个常量引用一个值。使用cost定义的对象和数据，其实是可变的

## 变量类型：原始类型、复合类型

6中基本数据类型：number、string、boolean、undefined、null、symbol

复合类型：Object、function、array

* **类型判断**

**Typeof**

Typeof xxx的结果有7种：

Number、string、boolean、undefined、object、function、symbol

注意：

1. Typeof null的结果是object，虽然null是原始类型，而非引用类型

这是因为在js的最初版本中，使用的是32位系统，为了性能考虑使用低位存储变量的类型信息。000开头代表的就是对象，另外null表示为全零。因此将null错误的判断为Objec。

1. typeof 复合类型，除了function类型的结果是function，其他的都是object，注意typeof xxx并没有array这一项，typeof [] 为object。因此使用typeof 并不能准确的判断变量的类型。
2. es6新增：typeof symbol 类型的值，结果是symbol。

* **Instanceof**

用于实例和构造函数的对应

Function fun(){}

Var f = new fun()

F instanceof fun

上面介绍过typeof [] 的结果是object，无法判断出数组类型，但是可以用instanceof来判断。

[] instanceof Array //true

## 值类型、引用类型

根据变量类型传递方法，变量又可以分为值类型和引用类型。

值类型：存储的是在栈中的数据。在参数传递方式上，值类型是按值传递。

引用类型：真实的数据存放在堆内存里，存储的是该对象在栈中引用。在参数传递方法上，引用类型是按共享传递。

Typeof 识别出来的类型中，出去object和function是引用类型，其他的都是值类型。

## 深拷贝/浅拷贝

* **概念**

1. 深拷贝和浅拷贝针对的是引用类型。基本类型的值存储在栈中，当复制时，栈内存会开辟一个栈内存。所以二者修改时，彼此不会影响。
2. 浅拷贝复制的是指向对象的指针，并没有开辟新的堆内存，原对象和新对象还是共享一块内存，修改新对象自然会影响原对象。

浅拷贝是按位拷贝对象，它会创建一个新对象，这个对象有着原始对象属性值的一份精确拷贝。如果属性是基本类型，拷贝的就是基本类型的值；如果属性是内存地址（引用类型），拷贝的就是内存地址

1. 深拷贝会开辟新的堆内存，原对象和新对象不共享同一块内存，修改新对象不会影响原对象。

* **浅拷贝的实现方式**

1、Object.assign()

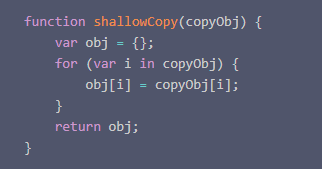
Object.assign()方法可以把任意多个的原对象自身可枚举属性拷贝给目标对象，然后返回目标对象。

2、Array.prototype.concat()

Contat()方法用于合并两个或多个数组。该方法不会更改现有数组，而是返回一个新数组

3、Array.prototype.slice()

4、循环复制



* **深拷贝的实现方式**

1. JSON.parse(JSON.stringify())

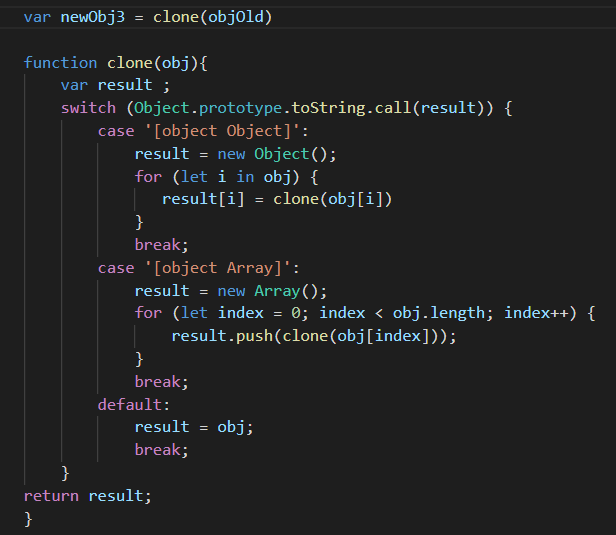
使用JSON.stringify将对象转成JSON字符串，在用JSON.parse()把字符串解析生对象。

此方法不能处理函数

1. 递归方法

遍历对象、数组，直到里面都是基本数据类型，然后再去复制，可实现深度拷贝。

该方法如果属性值是函数、undefined、symbol时，会忽略掉



1. jquery的$.extend()
2. js库loadsh中的\_.cloneDeep()

## setTimout和setInterval实现原理

setTimeout的主要思想是通过一个定时器，让函数在计时结束后再执行；

setInterval则是每隔一定的时间，就启动一次函数的执行；

setTimeout原理是：在指定的时间之后，把回调函数加入到任务队列末尾。

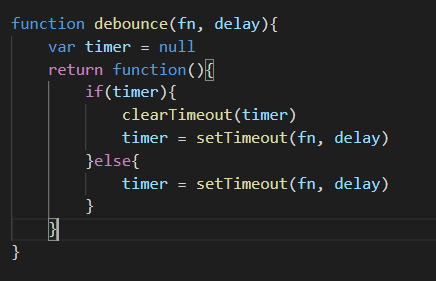
如果这个时间点任务队列中还存在其他的任务，那么该函数会在这些任务之后再执行，所以回调函数的实际执行时间大于等于指定的时间。

setInterval本质上就是每隔一定的时间向任务队列添加回调函数。但setInterval有一个原则：在向队列中添加回调函数时，如果队列中存在之前由其添加的回调函数，就放弃本次添加（不会影响之后的计时）。

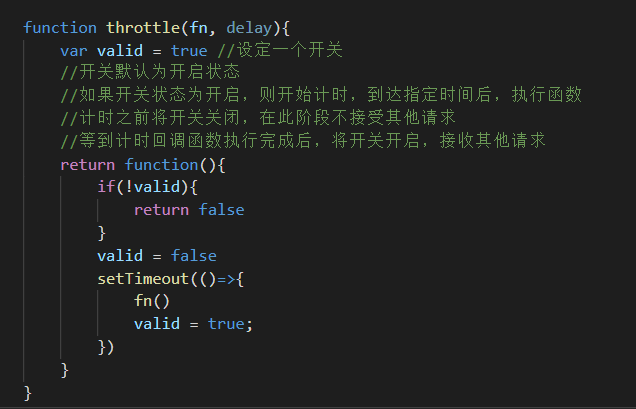
## 函数的防抖、节流

防抖和节流都是为了提升运行效率，减少函数的执行次数

防抖：把多次函数执行，合并成一次执行。给定一个间隔时间，当两次函数执行的间隔时间大于给定的间隔时间，就执行一次函数。



节流：减少函数执行的频率。规定一个单位时间，在单位时间内触发一个事件回调，触发时超过间隔则执行，否则不执行。



防抖和节流的应用场景：

1. 搜索框input事件
2. 页面滚动时

相关库：undercore.js

## requestAnimationFrame

window.requestAnimationFrame()该方法需要传入一个回调函数作为参数，该回调函数会在浏览器下一次重绘之前执行

注意：若你想在浏览器下次重绘之前继续更新下一帧动画，那么回调函数自身必须再次调用window.requestAnimationFrame

* **通过requestAnimationFrame获取帧率**

<body>

    <div id="text"></div>

    <script>

        var lastLoop =  (new Date()).getMilliseconds();

        var count = 1,arr = [],fps = 0, index = 0

        function step(){

          var currentLoop = (new Date()).getMilliseconds(); //获取毫秒

          if(lastLoop > currentLoop){  //从新开始新的一轮0-10000

              fps = count;

              arr.push(fps)

              count = 1;

          }else{//0-1000毫秒

              count += 1

          }

          document.getElementById('text').innerHTML = lastLoop + currentLoop

          console.log(count,fps)

          lastLoop = currentLoop

          index = index + 1

          if(index < 1000){

            window.requestAnimationFrame(step);

          }else{

              console.log('每秒的帧数')

              console.log(arr) //(16) [54, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60]

          }

        }

        step()

    </script>

</body>

* FPS 每秒传输帧数（Frames Per Second），刷新率 单位HZ；
* 浏览器的重绘频率一般会和显示器的刷新率保持同步。大多数采用W3C规范，浏览器的渲染页面的标准频率也为60FPS即每秒重绘60次，
* 当页面隐藏或最小化时，会被暂停，页面显示，会进行执行。节省了CPU开销。

## new操作符

1. 创建一个新的对象
2. 将对象的\_\_proto\_\_执行构造函数的prototype
3. 将构造函数的this指向这个新对象，执行构造函数的代码，为这个对象添加属性、方法等
4. 返回新对象

## 什么是polyfill？

Polyfill是用于实现浏览器并不支持的原生API的代码

比如说Object.assign()是很多现代浏览器都支持的原生Web API，但是古老的浏览器并不支持，那么假设有人写了一段代码来实现这个功能是这些浏览器也支持了这个功能，那么这久可以成为一个polyfill。

## 事件代理/委托

是JavaScript中常用的绑定事件的常用技巧。

即是把原来需要绑定在子元素的响应事件（click、keydown…）委托给父元素，

让父元素担当事件监听的职务

事件代理的原理是DOM元素的事件冒泡

## 事件冒泡

有两个fatherDiv和sonDiv，fatherDiv是sonDiv的父节点；

fatherDiv绑定fatherClickEvent，sonDiv绑定sonClickEvent;

当点击sonDiv后会先执行sonClickEvent再执行fatherClickEvent，

这是因为sonDiv将click事件传递给了fatherDiv；类似这种事件向上传递的现象，就叫做**事件冒泡；**

**如何取消事件冒泡？**

 function stopBubble(e){

        e = e?e:window.event

        if(e && e.stopPropagation){

            e.stopPropagation()

        }

        e.cancelBubble = true

    }

## Ajax、Axios、Fetch的核心区别

<http://www.axios-js.com/zh-cn/blogs/>

Ajax = Asynchronous JavaScriptand XML 异步网络请求。

AJAX带来的最大影响就是页面可以无刷新的请求数据。

1. **JavaScript原生实现AJAX**

所有现代浏览器均支持XMLHttpRequest对象（IE5和IE6使用ActiveObject）

* **构造函数**

XMLHttpRequest()

该构造函数用于初始化一个XMLHttpRequest实例对象。在调用下列任何其他方法之前，必须先调用该构造函数。

* **属性**

此接口继承了XMLHttpRequestEventTarget和EventTarget的属性。

|  |  |
| --- | --- |
| onreadystatechange |  |
| readyState | 代表请求状态码   1. unsent 代理被创建，但尚未调用open方法； 2. opened open方法已经调用； 3. headers\_recelived send()方法已经调用，并且头部和状态已经可获得； 4. loading 下载中，reponseText属性已经包含部分数据； 5. done 下载操作已完成。 |
| response | 返回一个ArrayBuffer、Blob、Document，或DOMString，具体是哪种类型取决于responseType的值。其中包含整个响应体。 |
| responseText | 返回一个DOMString，该DOMString包含对请求的响应，如果请求未成功或尚未放送，则返回null。 |
| responseType | 用于定义响应类型的枚举值  1、“” 空字符串，采用默认类型DOMString，与设置为text相同；  2、arraybuffer response是一个包含二进制数据的JavaScript ArrayBuffer；  3、blob response是一个包含二进制数据的blob对象；  4、document response是一个HTMLDocument或XML XMLDocument，这取决于接收到的数据是MIME类型。  5、json  6、text  7、ms-stream 下载流的一部分 |
| responseURL | 返回经过序列化的响应URL，如果该URL为空，则返回空字符串 |
| responseXML | 返回一个Document，其中包含该请求的响应，如果请求未成功，尚未发生或不能解析为XML或HTML，则返回null。 |
| status | 整形数字，代表请求的响应状态 |
| statusText | 包含完整的响应状态文本，如200 K |
| timeout | 表示该请求的最大请求时间（毫秒），若超出该请求，请求会自动终止 |
| ontimeout |  |
| upload | 代表上传进度 |
| withCredentials / krer den chuo s/ | 一个布尔值指定跨域请求是否应该带有授权信息，如cookie或授权header头 |

* **方法**

|  |  |
| --- | --- |
| abort() [əˈbɔːt] | 如果请求已被发出，则立即中止请求 |
| getAllResponseHeaders |  |
| getResponseHeader() | 返回包含指定响应头的字符串 |
| open() | 初始化一个请求 |
| overrideMimeType | 覆写有服务器返回的MIME类型 |
| end() | 发送请求，如果请求是异步的，那么该方法将在请求发送后立即返回 |
| setRequestHeader() | 设置HTTP请求头的值，必须在open之后，send之前调用该方法。 |

* **事件**

可以设置on\*属性，也提供标准的监听器addEventListener()API来监听事件。

|  |  |
| --- | --- |
| abort | 当request被停止时触发，如当程序调用XMLHttpRequest.abort()时，也可以使用onabort属性。 |
| error | 当request遭遇错误时触发。也可以使用onerror属性。 |
| load | XMLHttpRequest请求成功完成时触发。也可以使用onload属性。 |
| loadend | 当请求结束时触发，无论请求成功（load）还是失败（abort或eoor）。也可以使用onloadend属性。 |
| loadstart | 接收到响应数据时触发。  也可以使用onloadstart属性。 |
| progress [ˈprəʊɡres] | 当请求接收到更多数据时，周期性地触发。  也可以使用onprogress属性。 |
| progress事件会在请求接收到数据的时候被周期性触发。 |
| Timeout | 在预设时间内没有接收到应用时触发。  也可以使用ontimeout属性。 |

1. **jquery实现AJAX**

Jquery Ajax 是对原生XHR的封装，除此之外还增添了对JSONP的支持；

Ajax本身是针对MVC的编程，不符合现在前端MVVC的趋势；

很多情况下我们只需要使用ajax，但是却需要引入整个jquery，这是非常不合理的，于是便有了fetch的解决方案。

Jquery ajax 函数库

<https://www.w3school.com.cn/jquery/jquery_ref_ajax.asp>

1. **Axios是一个基于promise用于浏览器和nodejs的HTTP客户端；**

本质上也是对原生XML的封装，只不过它是promise的实现版本，符合最新的ES标准，有以下特点：

从浏览器中创建XMLHttpRequest

从node.js创建http请求

支持promise API

拦截请求和响应

转换请求数据和响应数据

取消请求

自动转换JSON数据

客户端支持防御XSRF

<http://www.axios-js.com/zh-cn/docs/#%E4%BB%80%E4%B9%88%E6%98%AF-axios%EF%BC%9F>

如何封装axios

* Axios的拦截器：interceptors [ˌɪntəˈsɛptəz]

添加请求拦截器：axios.interceptors.request.use()

1. **Fetch是ES6新增的通信方式，不是ajax，但是它本身实现数据通信，也是基于Promise管理的。**

Fetch 是一种HTTP数据请求的方式，是XML httpRequest的一种替代方案；

不是Ajax的进一步封装，而是原生JS，没有使用XML httpRequest对象。

串行：请求是异步的，需要等到上一个请求成功，才能执行下一个请求

并行：同时发送多个请求，HTTP请求可以同时进行，但是JS的操作都是一步步的，因为JS是单线程，等待所有请求都成功，我们再去做什么事情。

**Promise.all([**

**])**

**重写Promise.all()**

**同时并发几百个请求，怎么处理**

**Asyncpool**

## Onload和DomContentLoaded

当onload事件触发时，页面上所有的dom、样式表、脚本、图片、flash都已经加载完成了。

当DomContentLoaded事件出发时，仅当dom加载完成，不包括样式表、图片、flash

## CommonJs模块和Es6模块的区别

* **CommonJS**

1. 对于基本数据类型，属于复制。及会被模块缓存。同时，在另一个模块可以对该模块输出的变量重新复制。
2. 对于复杂数据类型，属于浅拷贝。由于两个模块引用的对象指向同一个内存空间，因此对该模块的值做修改时会影响另一个模块。
3. 当使用require命令加载某个模块时，就会运行整个模块的代码。
4. 当使用require命令加载同一个模块时，不会再执行该模块，而是取到缓存之中的值。也就是说，CommonJs模块无论加载多少次，就只会在第一次加载时运行一次，以后再加载，就返回第一次运行的结果，除非手动清除系统缓存。

* **ES6模块**

1. es6模块中的值属于【动态只读引用】
2. 对于只读来说，即不允许修改引用变量的值，import的变量是只读的，不论是基本数据类型还是复杂数据类型。当模块遇到import命令时，就会生成一个只读引用。等到脚本真正执行时，再根据这个只读引用，到被加载的那个模块里面去取值。
3. 对于动态来说，原始值发生变化，import加载的值也会发生变化。不论是基本数据类型还是复杂数据类型。

## **<script>标签属性defer和async**

当浏览器碰到script脚本的时候：

* 没有defer或者async，浏览器会立即加载并执行指定的脚本，“立即”指的是在渲染该script标签之下的文档元素之前，也就是说不等待后续载入的文档元素，读到就加载并执行。
* 有async，加载和渲染后续文档元素的过程和a.js的加载与执行并行进行（异步）。

执行a.js时，html解析暂停。

* 有defer，加载后续文档元素的过程和a.js的加载并行进行，但是a.js的执行要在所有元素解析完成之后，DOMContentLoaded事件触发之前完成。

HTML、JavaScript和css中，除了css，没有文件可以边下载、边解析。

## 懒加载（延迟加载）图片的原理

在图片没有进入可视化区域时，先不给src赋值，这样浏览器就不会发送请求了，

等到图片进入可视化区域再给src赋值，

图片的真实地址需要存储在data-src中，

图片没有进入可视区域，也就是说图片的offsetTop需要小于页面的可视高度，

当页面存在滚动条的时候，则需要满足img.scrollTop小于页面的可视化区域高度 + 页面滚动的高度。

## 异步解决方案

1. 回调函数
2. 发布订阅
3. Promise
4. Generator
5. Aysnc/await

## 柯里化和反柯里化

* 概念：柯里化就是把接收多个参数的函数变成接受一个单一参数（最初函数的第一个参数）的函数，并且返回接受余下参数而且返回结果的新函数的技术。
* 好处

1. 参数复用
2. 提前返回
3. 延迟计算/执行

# ES6

## 箭头函数

Es6标准新增了一种新的函数：Arrow Function（箭头函数）

箭头函数相当于匿名函数，并且简化了函数定义。箭头函数有两种格式，一种只包含一个表达式，可以省略{……}和return。还有一种可以包含多条语句，这时候不能省略{……}和return。

* 箭头函数this总是指向词法作用域，也就是外层调用者。

使用call、apply无法改变箭头函数的this。

* 箭头函数不能作为构造函数，即不可以使用new实例化。
* 箭头函数不绑定arguments。
* 箭头函数没有prototype属性。
* 箭头函数不能用作函数生成器，yield关键字不能在箭头函数中使用。

## Proxy

## Reflect

## Babel、polyfill

Babel默认只转换新的JavaScript句法（syntax），二不转化新的API，比如Iterator、Generator、Set、Map、Proxy、Reflect、Symbol、Promise等全局对象，以及一些定义在全局对象上的方法（比如Object.assign）都不会转码。

## promise

厄运回调金字塔、链式回调

Promise对象的then方法用来添加回调函数

Promise对象链呈流水线的模式进行作业，是因为在promise对象对自身的onFulfilled和onRejected响应器的处理中，会对其中返回的promise对象进行处理

其内部会将这个新的promise对象加入到promise对象链中，并将其暴露出来，使其继续接受新的promise对象的加入。只有当promise对象链中的上一个promise对象加入成功或失败阶段，写一个promise对象才会被激活，这就线程了流水线的作业模式；

* Promise的then是同步的；
* Promise的then内部的方法是异步微任务；
* 链式调用后面的then需要等待上一个then执行完成再注册

## promise.then()方法里面的函数为什么是异步的？

假设、promise.then()内部的方法是异步执行的；那么then内部的异步函数肯定在promise内部的异步函数之后执行。

Promise.then通过将内部方法push到一个数组里面进行订阅

当promise里面的异步函数执行完成，会调用上面的数组执行回调函数，由于then里面的方法还没有执行，只是数组则为空。

假设，promise.then内部的方法是同步执行的；then内部的方法需要等待promise内部的方法执行完成后才能够执行。如果promise内部的异步函数执行很久，那么then里面的函数就要一直等着，因为是同步的，所以会阻塞后面的同步代码执行。

**以下是手写promise**

const PENDING = 'pending'

const RESOLVED = "resolved"

const REJECT = "reject"

//因为所有的promise都遵循这个规范，所以

const resolvePromise = (promise2, x, resolve, reject) => {

   //判断X的值

   if(promise2 === x){ //此时会出现死循环，promise2不能return promise2

       return reject(new TypeError('promise2不能返回promise2'))

   }

   //判断数据类型 typeof contructor instanceof toString

   if(typeof x === 'object' && typeof x !== null || typeof x === 'function'){

        let called = false;

        try {

            let then = x.then

            if(typeof then === 'function'){ //认为它是promise

                then.call(x, yes=>{

                    if(called){

                        return

                    }

                    called = true

                    // resolve(yes) //如果yes还是一个promise

                    resolvePromise(promise2, yes, resolve, reject)

                }, no =>{

                    if(called){

                        return

                    }

                    called = true

                    reject(no)

                })

            }else{

                resolve(x)

            }

        }catch (e){

            if(called){

                return

            }

            called = true

            reject(e)

        }

   }else{

       // x是一个普通值

       resolve(x)

   }

}

class Promise{

    constructor(executor){

        this.status = PENDING;

        this.value = undefined;

        this.error = undefined;

        this.onfulfilledcallback = []

        this.onrejectedcallback = []

        let resolve = (value) =>{

            if(this.status == PENDING){

                this.status = RESOLVED

                this.value = value

                this.onfulfilledcallback.forEach(fn=>{

                    fn()

                })

            }

        }

        let reject = (error) => {

            this.status = REJECT

            this.error = error

            this.onrejectedcallback.forEach(fn=>{

                fn()

            })

        }

       try {

            executor(resolve, reject);

       } catch (error) {

           reject(error)

       }

    }

    then(onfulfilled, onrejected){

       //同步的情况

       //同步时resolve在then前面执行

        if(this.status == RESOLVED){

            onfulfilled(this.value)

        }

        if(this.status == REJECT){

            onrejected(this.error)

        }

        //异步的情况

        //异步时then在resolve前面执行

        if(this.status == PENDING){

            this.onfulfilledcallback.push(()=>{

                onfulfilled(this.value)

            })

            this.onrejectedcallback.push(()=>{

                onrejected(this.error)

            })

        }

    }

    //链式

    // 1、判断成功和失败函数的返回结果

    // 2、判断是不是promise，就采用他的状态

    // 3、如果不是promise

    then2(onfulfilled, onrejected){

        //onfulfilled, onrejected是可选参数

       onfulfilled = typeof onfulfilled === 'function'? onfulfilled: data => {

           return data

       }

       onrejected = typeof onrejected === 'function'?onrejected : err => {

           throw err

       }

        //同步的情况

       //同步时resolve在then前面执行

       let promise2 = new Promise((resolve, reject)=>{

            if(this.status === RESOLVED){

                setTimeout(() => {

                    try {

                        let x = onfulfilled(this.value)

                        // x可能是普通值，也可能是promise

                        // 此处使用promise2，需要使用异步的方式

                        // 加定时器后，异步代码没办法被外部try-catch捕获

                        resolvePromise( promise2, x, resolve, reject)

                    } catch (error) {

                        reject(error)

                    }

                },  0)

            }

            if(this.status === REJECT){

                setTimeout(() => {

                   try {

                        let x = onrejected(this.error)

                   } catch (error) {

                       reject(error)

                   }

                }, 0);

            }

            //异步的情况

            //异步时then在resolve前面执行

            if(this.status === PENDING){

                this.onfulfilledcallback.push(()=>{

                    setTimeout(() => {

                        try {

                            let x = onfulfilled(this.value)

                            resolvePromise( promise2, x, resolve, reject)

                        } catch (error) {

                            reject(error)

                        }

                    }, 0);

                })

                this.onrejectedcallback.push(()=>{

                    setTimeout(() => {

                        try {

                            let x = onrejected(this.error)

                            resolvePromise( promise2, x, resolve, reject)

                        } catch (error) {

                            console.log("异步出错")

                            reject(error)

                        }

                    }, 0);

                })

            }

       })

        return promise2

    }

}

const isPromise = (value) => {

    if((typeof value === 'object' && value !== null) || typeof value === 'function' ){

        if(typeof value.then === 'function'){

            return true

        }

    }

    return false

}

Promise.prototype.finally = function(cb){

    return p.then(data => {

        Promise.resolve(cb()).then(()=>data)

    }, err => {

       Promise.resolve(cb()).then(()=> {

           throw err

       })

       throw err

    })

}

Promise.all = function(values){

    return new Promise((resolve, reject)=>{

        let arr = []

        let index = 0

        function processData(key, value){

            arr[key] = value

            if(++index === values.length){

                resolve(arr)

            }

        }

        for(let i=0; i<values.length;i++){

            let current = values[i]

            if(isPromise(current)){

                current.then((data)=>{

                    processData(i,data)

                }, reject);

            }else{

                processData(i,current)

            }

        }

    })

}

Promise.all([1,2]).then((data)=>{

    console.log(data)

})

let p = new Promise(function(resolved, reject){

    setTimeout(() => {

        resolved('成功')

    }, 1000);

})

p.then2((data) =>{

    console.log('1是一个字符串：', data)

    return new Promise((resolve, reject) =>{

        resolve(123)

    });

}).then2((data)=>{  //订阅，先订阅再发布

   console.log('2是一个promise',data)

   return ()=>{

       console.log("函数输出")

   }

}, (error) => {

    console.log('---', error)

}).then2(data => {

    console.log('3是一个函数', data)

}).then2(data => {

    console.log(4, data)

})

## 协程

协程是一种程序运行的方式

协程是一种比线程更加轻量级的存在，协程存在线程的环境中，一个线程可以存在多个协程，可以将协程理解为线程中的一个个任务。不像进程和线程，协程不受操作系统的管理，而是被具体的应用程序代码所控制。一个线程一次只能执行一个协程，比如当前执行A协程，另外还有一个B协程，如果想要执行B的任务，就必须在A的协程中将JS线程的控制权转交给B协程，那么现在B执行，A就相当于处于暂停的状态。A将执行权交给B，也就是A启动B，我们也称A是B的父协程

## Generator

[ˈdʒenəreɪtə(r)]

Function\*

Function\*这种声明方式（function关键字后跟一个星号）会定义一个生成器函数(generator function)，它返回一个generator对象。

生成器函数在执行时能暂停，后面又能从暂停处继续执行。

调用一个生成器函数并不会马上执行它里面的语句，而是返回一个这个生成器的迭代器对象。

当这个迭代器的next()方法被首次（后续）调用时，其内的语句会执行到第一个（后续）出现yield [jiːld] 的位置为止，yield后紧跟迭代器要返回的值。或者如果用的是yield\*，则表示将执行权移交给另一个生成器函数（当前生成器暂停执行）。

next() 方法返回一个对象，这个对象包含两个属性：value和done，value属性表示本次yield表达式的返回值，done属性为布尔值，表示生成器后续是否还有yield语句，及生成器函数是否已经执行完毕并返回。

调用next()方法时，如果传入了参数，那么这个参数会传给上一条执行的yield语句左边的变量。

* **与Iterator的关系**

[yi ter rei ter]

Symbol [ˈsɪmbl]

* **for-of循环遍历generator实例对象**

function \* gen(){

    yield 1;

    yield 2;

    yield 3;

}

var g = gen()

for(let i of g){

    console.log(i)

}

输出为： 1、2、3

* **generator.prototype.throw**

[ˈprəʊtətaɪp]

Generator函数返回的遍历器对象，都有一个throw方法，可以在函数体外抛出错误，然后在generator函数体内捕获

* **generator.prototype.return**
* **yield\* 表达式**

[jiːld]

如果在generator函数内部，调用另一个generator函数。需要在前者的函数体内部，手动完成变量。

function\* genA(){

    yield 1;

    yield 2;

}

function\* genB(){

    yield 'a';

    for(let i of genA()){

        console.log(i)

    }

    yield 'b'

}

for(let i of genB()){

    console.log(i)

}

输出结果：

a

1

2

b

如果有多个generator函数嵌套，写起来就非常麻烦。ES6提供了yield\*表达式，作为解决办法，用来在一个generator函数里面执行另一个generator函数。

function\* genA(){

    yield 1;

    yield 2;

}

function\* genB(){

    yield 'a';

    yield genA();

    yield 'b'

}

for(let i of genB()){

    console.log(i)

}

输出结果：

a

Object [Generator] {}

b

## Java的class和JavaScript的class有什么区别？

javascript的类是基于原型的继承机制；

java的类是基于类的继承机制。

## for in 和 for of的区别

* for in 多用于对象的遍历，若是用来遍历数组，则拿到的是数组的索引。若是用来遍历对象，拿到的是属性名。

for of多用于遍历数组，拿到的是数组的每一项。若是用来遍历对象，则会报错（TypeError：obj is not iterable），因为对象不可迭代，没有迭代器。

# Nodejs

## http

http模块是node.js中非常重要的一个核心模块。

可以使用http.createServer方法创建一个http服务器；

可以使用http.reuqest方法来创建一个http客户端；

HTTP是一个用来传送数据的应用层协议，其底层使用TCP传输协议；

HTTP是互联网数据通信的基础，要相应客户端请求（如浏览器访问），首先需要部署HTTP服务器。

<https://itbilu.com/nodejs/core/N1okQ7Eh.html>

Nodejs如何创建HTTP服务器？

https://itbilu.com/nodejs/core/N1okQ7Eh.html

var http = require('http');

// 创建server对象，并添加request事件监听器

var server = http.createServer(function(req, res) {

    res.writeHeader(200,{'Content-Type':'text/plain'});

    res.end('http://erp.office.51fanli.com/');

});

server.listen(3000)

Nodejs对客户端的支持

var http = require('http');

var options = {

    host: '10.200.137.53',

    port: '8080',

    method: 'GET',

    path: '/Okr/OKpi/getProjectList?page=1&name=&type='

}

var req = http.request(options);

req.on('response', function(res){

    res.setEncoding('utf8');

    res.on('data', function(chunk){

        console.log('收到数据：%s', chunk);

    });

});

req.end();

返回结果：

# Webpack

## 四个核心概念（entry、output、loader、plugins）

入口（entry）：指示webpack应该使用哪个模块，来作为构建其内部依赖图的开始；

输出（output）：在哪里输出它所创建的bundles；

Loader：让webpack能够去处理那些非JavaScript文件；

插件（plugins）：用于执行范围更广的任务。

## 常用loader

**css-loader和style-loader**

css-loader用来解释@import和url();

style-loader用来将css-loader生成的样式表通过<style>标签，插入到页面中去。

**sass-loader、 less-loader和postcss-loader**

**postcss-loader**

主要功能：

* 把css解析成JS可以操作的抽象语法树AST
* 调用插件来处理AST并得到结果

配合autoprefixer插件，打包后的css自动加上-moz、-ms、-webkit等浏览器私有前缀

**babel-**loader

配合babel将高版本的ES6甚至ES7转化为ES5

**file-loader和url-loader**

都是用来处理图片、字体图标等文件

分两种情况：

1. 当文件大小小于limit参数，url-loader将文件转化为base-64编码，用于减少http请求；
2. 当文件大小大于limit参数，调用file-loader进行处理

vue-loader

## webpack构建过程

webpack的运行流程是一个串行的过程，从启动到结束会依次执行以下流程：

1. 初始化参数：从配置文件和shell语句中读取与合并参数，得到最终的参数。
2. 开始编译：用上一步得到的参数初始化compiler对象，加载所有配置的插件，执行对象的run方法开始执行编译。
3. 确定入口：根据配置中的entry找到所有的入口文件。
4. 编译模块：从入口文件出发，调用该所有配置的loader对模块进行翻译，再找出该模块依赖的模块，再递归本步骤直到所有入口依赖的文件都经过本步骤的处理、
5. 完成模块编译：在经过第4步使用loader翻译完所有模块后，得到了每个模块翻译后的最终内容以及它们之间的依赖关系。
6. 输入资源：根据入口和模块之间的依赖关系，组装成一个个包含多个模块的chunk，再把每个chunk转化成一个单独的文件加入到输出列表，这步是可以修改输出内容的最后机会。
7. 输出完成：再确定好输出内容以后，根据配置确认输出的路径和文件名，把文件内容写入到文件系统。

在以上过程中，webpack会在特定的时间点广播出特定的事件，插件在监听到感兴趣的事件后会执行特定的逻辑，并且插件可以调用webpack提供的API改变webpack的运行结果。

## 热更新HMR原理

1. 在启动webpack的时候，sockjs在服务器和浏览器端建立一个WebSocket长连接。
2. Webpack-dev-server调用webpack api监听compiler的done事件。当compile完成后，webpack-dev-server将编译打包的新模块hash值通过WebSocket长连接发送给浏览器端。
3. 浏览器端webpack-client接收到type为hash的数据后将hash值暂存起来。
4. Webpack-dev-server通过WebSocket长连接发送error、ok、warning类型的信息给浏览器。
5. 浏览器端webpack-dev-client接收到type为ok或warning的消息后，根据hot配置决定是刷新浏览器还是对浏览器进行热更新。

如果是热更新，基于node events的emit和on方法，将步骤3中获取到的hash提交给webpack，然后将控制权交给webpack客户端代码。

1. Webpack客户端client先判断模块是否有更新，即判断服务器传过来的最新hash值和客户端上一次缓存的hash是否一致。
2. 不一致的话，代表模块已更新，则webpack客户端client通过AJAX向服务器请求最新的文件，如果有将返回更新的文件hash列表。
3. webpack客户端client通过jsonp请求最新的代码块。
4. 新的代码块替换旧的模块代码。
5. 替换后，我们的业务代码并不知道代码已经发生变化，所以我们需要在业务代码中调用HMR的accept方法，添加模块更新后的处理函数

Module.hot.accept

我们手动在业务代码中添加容易犯错，也很麻烦，目前已经有很多的loader可以帮忙处理。

## devtool

# Vue

<https://segmentfault.com/a/1190000038848131>

## 生命周期

|  |  |  |
| --- | --- | --- |
| 生命周期钩子 | 组件状态 | 最佳实践 |
| BeforeCreate | 实例创建之后，this指向创建的实例，不能访问到data，computed、watch、methods上的方法和数据 | 常用于初始化非响应式变量 |
| Created  [kriˈeɪt] | 实例创建完成，可访问data、computed、watch、methods上的方法和数据，未挂载到DOM，不能访问$el属性，$ref属性内容为空数组。 | 常用于简单的ajax请求，页面的初始化 |
| BeforeMount | 在挂载开始之前被调用，beforeMount之前，会找到对应的template，并编译成render函数 |  |
| Mounted | 实例挂载到DOM上，此时可以通过DOM API访问到DOM节点，$ref属性可以访问 | 常用于获取VNode信息和操作，ajax请求 |
| BeforeUpdate | 响应式数据更新时调用，发生在虚拟DOM打补丁之前 | 适合更新之前访问现有的DOM，比如手动移除已添加的事件监听器 |
| Updated | 虚拟DOM重新渲染和打补丁之后调用，组件DOM已经更新，可执行依赖于DOM的操作 | 避免在这个钩子函数中操作数据，可能陷入死循环 |
| BeforeDestroy  [dɪˈstrɔɪ] | 实例销毁之前调用。这一步，实例依然可用，this仍能获取实例 | 常用于销毁定时器，解绑全局变量、销毁插件对象等操作 |
| Destroyed | 实例销毁后调用，调用后，vue实例指示的所有东西都会解绑定，所有的事件监听器会被移除，所有的子实例也会被销毁。 |  |

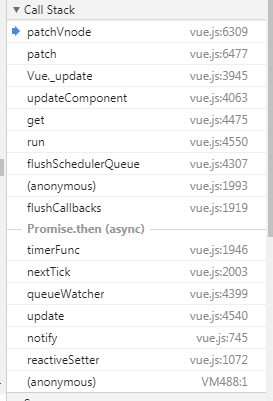
注意:  
created阶段的ajax请求与mounted请求的区别：前者页面视图未出现，如果请求信息过多，页面会长时间处于白屏状态。

## 父子组件各声明周期的执行顺序

## 虚拟dom，diff算法

virtual-dom完全使用js实现的，和宿主浏览器没有任何关系

* 1. vue的diff算法
     1. 调用栈



## 什么是单向数据流？

数据通常按照一个方向来传递，通常从父传递子组件，子组件要修改父组件数据需要通知父组件自己修改，目的是便于状态的管理和解耦。

如果不加以限制，任何子组件内部直接修改父组件数据就会影响到其他引用相同数据的组件的显示，从而造成数据混乱。

## Vue2实现数据的双向绑定/响应式原理

Vue2.0使用Object.defineProperty()实现数据响应

Vue遍历data中的数据，使用Object.defineProperty()劫持getter和setter

在getter中做数据依赖收集

在setter中监听数据的变化，并通知订阅了当前数据的地方

Vue通过数据劫持（Object.defineProperty）+ 订阅发布模式

监听器Observer：用来劫持并通过Object.defineProperty监听所有属性（转变成setter和getter形式），如果属性发生变化，就通知订阅者

订阅器Dep：用来收集订阅者，对监听器Observer和订阅者Watcher进行统一管理

订阅者Watcher：监听器Observer和解释器Compile之间通信的桥梁；如果收到属性变化通知，就会执行相应的方法，从而更新视图。

解析器Compile：可以解析每个节点的相关指令，对模板数据和订阅器进行初始化。

主要做的事情是：

1. 在自身实例化时往属性订阅器（Dep）里面添加自己
2. 自身有一个update方法
3. 待属性变动dep.notice()通知时，就触发自身的update方法，并触发解析器（compile）中绑定的回调。

## Vue3.0采用proxy实现双向绑定

Object.defineProperty()的问题有：

1. 检测不到对象属性的添加和删除

Vue的set是能让vue知道你添加了属性

1. vue2.0不监控数组下标的变化，导致直接通过数组的下标给数组设置值，不能实时响应。

Js可以通过Object.defineProperty()监控数据下标的变化，但vue的设计者出于性能考虑取消了这个功能，并不是原生不支持。

1. 只能劫持对象的属性，因此我们需要对每个对象的每个属性进行遍历，如果属性值也是对象，就需要深度遍历，当data中的数据比较多且层级很深的时候，会有性能问题，因为要遍历data中所有的数据并给其设置成响应式。

使用Proxy的优势：

1. 可直接监听数组类型的数据变化。
2. 监听的目标为对象本身，不需要像Object.definePropery一样遍历每一个属性，有一定的性能提升。
3. 可拦截apply、ownkeys、has等13中方法。
4. 直接实现对象属性的新增和删除。

## 异步更新队列

当触发数据更新通知时，dep通知watcher进行数据更新，这时watcher会将自己加入到一个异步的更新队列中；

然后更新队列会将传入的更新操作进行批量处理；

这样就达到了多次更新同时完成，提高了用户体验，减少了浏览器的开销，增强了性能。

采用微任务执行刷新队列函数。

## nextTick

[JS的运行机制](#_Js执行机制)

实现原理：基于语言执行机制实现，直接创建一个异步任务，那么nextTick自然就达到再同步任务后执行的目的。

nextTick是vue的更新策略，也是性能优化手段，基于JS执行机制实现。

Vue中我们改变数据时不会立即触发视图，如果需要实时获取最新的DOM，这个时候可以手动调用nextTick。

$nextTick()会将我们传入的函数加入到异步更新队列中，

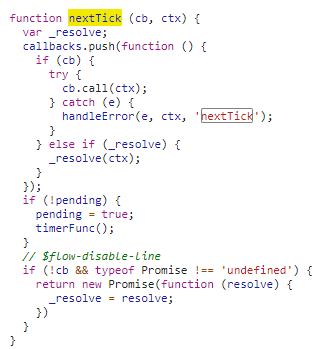
nextTick将函数放入一个callBacks的数组中；

然后推入到异步任务队列（微任务或者宏任务）中。

根据浏览器的渲染机制，渲染线程是在微任务执行完成之后运行的。渲染线程没运行，怎么拿到Dom呢？

因为，渲染线程只是把Dom树渲染成UI而已，vue更新Dom后，在Dm树中，新的Dom节点已经存在了，Js线程就可以拿到新的Dom了。除非开发者读取Dom的计算属性，触发了强制重流渲染线程才会打断js线程。

nextTick既是Vue的全局API，也是Vue的实例方法



## v-model的实现原理

|  |
| --- |
| <input v-model=”searchText” >  等价于:  <input v-bind:value=”searchText” v-on:input=”searchText = $event.target.value” > |

获取到页面表单上的v-model属性，拿到属性值val，为节点绑定input事件，同时把data中属性为val的属性值赋值给当前的表单value；

当前{{}}节点的时候，就创建一个观察者（Watcher），把这个观察者添加到当前val属性的发布者上。

## v-for的key为什么不建议使用数据下标

[**https://blog.csdn.net/qq\_28766729/article/details/98346565**](https://blog.csdn.net/qq_28766729/article/details/98346565)

## computed、watch、methods

computed是计算属性，它会根据所依赖的数据动态显示计算结果，该计算结果会被缓存起来。

computed应用场景

1. 适用于一些重复使用数据或复杂及费时的运算
2. 如果我们需要的数据依赖其他的数据的话

computed的成员可以只定义一个函数作为只读属性，也可以定义为get/set变成可读写属性

watch 是对data的数据监听回调，当依赖的data的数据变化时，会执行回调。在回调中传入newVal和oldVal两个参数。

Watch应用场景：

当在data中的某个数据发生变化时，我们需要做一些操作

watch 普通监听和深度监听

computed适合一个数据被多个数据影响，watch使用于一个数据影响多个数据。

Computed实现原理：

Watchers 也就是vm.\_computedWatchers对象的引用

存放每个计算属性的观察者watcher实例

## v-if和v-show的区别

v-if在进行切换时，会直接对标签进行创建和销毁，不显示的标签不会加载在DOM树中。

v-show在进行切换时，会对标签的display属性进行切换，通过display：none来隐藏元素。

一般来说，v-if的性能开销会比v-show大，切换频繁的标签更适合使用v-show。

## keep-alive

<keep-alive></keep-alive> 包裹动态组件时，会缓存不活动的组件实例,主要用于保留组件状态或避免重新渲染。

大白话: 比如有一个列表和一个详情，那么用户就会经常执行打开详情=>返回列表=>打开详情…这样的话列表和详情都是一个频率很高的页面，那么就可以对列表组件使用<keep-alive></keep-alive>进行缓存，  
这样用户每次返回列表的时候，都能从缓存中快速渲染，而不是重新渲染

<keep-alive>

<router-view v-if="$route.meta.alive"></router-view>

</keep-alive>

  <router-view v-if="!$route.meta.alive"></router-view>

结合vue-router使用

export default new Router({

    routes: [

        {

            path: '/banner',

            name: 'banner',

            component: Banner,

            meta:{

                alive: true,

            }

        },

]

})

**activated**::keep-alive组件激活时调用

## Vue全局API之Vue.extend

Vue.extend(options)

使用基础Vue构造器，创建一个“子类”。参数是一个包含组件选项的对象。

我们日常使用Vue编写组件是这个样子的：

Vue.component('MyComponent',{

    template: '<div>这是一个组件</div>'

})

这是全局组件的注册方法，但其实这是一个语法糖，真正的运行过程是这样的：

let  myComponent = Vue.extend({

    template: '<div>这是一个组件</div>'

})

console.log(myComponent)

Vue.component('MyComponent',myComponent)

Vue.extend()返回的是一个函数VueComponent，可以看做是Vue的一个子类

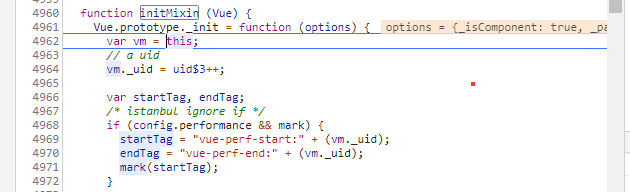


从源码上看VueComponent

1、有一个VueComponent函数



2、this.\_init调用的就是Vue原型上的\_init()



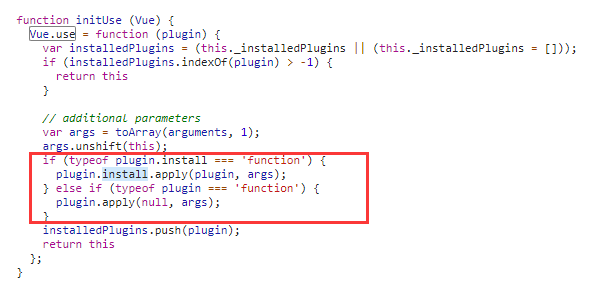
函数VueComponent可以new一个实例

let myComp = new myComponent()

console.log(myComp)

## Vue全局API之Vue.use

**源码**



1、\_installedPlugins 已安装插件列表

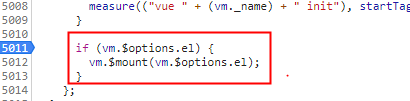
2、判断插件是否已经安装，已经安装就直接存缓存数组里面获取它

3、没有安装，则执行插件的install方法，没有install方法则直接执行插件

4、将新的插件放入数组，并返回

## Vue 选项/DOM之el

**如果没有el怎么处理**



如上，有el的时候才会挂载。

如果在实例化时存在这个选项，实例将立即进入编译过程，否则，需要显示调用vm.$mount()，手动启动编译。



手动挂载会判断options里面有没有template 和render，没有的话，会输出上面的警告。

## Vue选项name

类型String

限制：只有作为组件选项时起作用。

作用：1、允许组件模版递归调用自身。

2、便于调试。有名字的组件有更好的警告信息。另外，当在有vue-devtools，未命名组件将显示成<AnonymousComponent> [əˈnɒnɪməs] [kəmˈpəʊnənt]匿名组件，这很没有语义。通过提供 name选项，可以获取更有语义信息的组件树。

注意：组件在全局用Vue.component()注册时，全局ID自动作为组件的name。

## 什么是父组件，什么是子组件？

我们将某段代码封装成一个组件，而这个组件又在另一个组件中引入，而引入该封装组件的文件叫做父组件，被引用的组件叫做子组件。

## 父子组件，兄弟组件通信

1. 父组件向子组件传值

父组件引用子组件时，通过指令v-bind动态绑定一个属性，再在子组件中通过选项props接收来自父组件的数据。

* 父组件在子组件标签中绑定自定义属性；
* 子组件通过props[prɒps] 属性进行接收。

1. 子组件向父组件传值

子到父，在使用子组件的时候绑定自定义事件，在子组件内部拿到事件名称，再通过this.$emit(‘事件名’)触发并传递参数

* 父组件在子组件标签中绑定自定义事件；
* 子组件通过this.$emit()方法触发自定义事件，传值给父组件。

1. 兄弟组件之间传值

* 共同传给父组件，再有父组件分发（状态提升）；
* 使用Vuex
* 利用bus事件总线

其他的方式还有全局bus、provide/inject、refs/parent/children、attrs/$listeners

## 父组件有数据变更，如果通知到子组件

父组件data中的属性修改后，通过props传递给子组件的属性也会改变，且父组件和子组件的beforeUpdate和updated都会被触发，触发顺序为：

父组件beforeUpdate

子组件beforeUpdate

子组件updated

父组件updated

## 父组件子组件的生命周期顺序

父组件beforeCreate

父组件Created

父组件beforeMount

子组件beforeCreate

子组件Created

子组件beforeMount

子组件Mounted

父组件Mounted

## vuex

vuex是vue的一个插件，用来做状态管理

其实我们可以把vuex看做是一个仓库，我们把需要共享的数据统一存放在仓库中，谁需要，直接取就可以。然后提供好一个统一修改机制（提交mutation）来修改仓库里的数据。

仓库解决了数据传递层级和同级组件数据传递问题。

1、State：存放状态（数据）的地方

2、Mutation：修改状态的地方，用于变更store里面的数据

* 只能使用mutation修改state
* 为什么要使用mutation,方便后期的维护。
* 一个对象，里面可以定义多个函数。
* Mutation对象里面的函数不能执行异步操作

**触发mutation的第一种方式：**

* 通过this.$store.commit(‘add’)调用mutation对象里面的函数add；
* Mutation对象里面的函数的第一个参数是state，第二个参数是this.$store.commit(‘add’,2)传递过来的数据

**触发mutation的第二种方式：**

|  |
| --- |
| Import { mapMutation } from ‘vuex’;  …mapMutations([‘add’])  btnHandler(){  this.add()  } |

3、Action：做异步操作，并提交mutation修改状态的地方

* 如果通过异步操作更新数据，必须调用Action，而不能实现mutation,但是Action中还是要通过触发mutation的方式间接更改数据。

const store = new Vuex.Store({

    state: {

        conut: 0

    },

    mutations:{

        add(state){

            state.count ++

        }

    },

    Actions: {

        addAsync(context){

            setTimeout(()=>{

                context.commit('add')

            })

        }

    }

})

* 触发Actions的方式一：

this.$store.dispatch('addAsync')

* 触发Actions的方式二：

import { mapActions } from 'Vuex';

new Vue({

    methods:{

        ...mapActions(['addAsync']),

        btnHandler(){

            this.addAsync()

        }

    }

})

4 、Getters：派生状态的地方

对Store中的数据进行加工处理形成新的数据，类似Vue的计算属性。

Store中数据发生变化，Getter的数据也会跟着变化。

const store = new Vuex.Store({

    state: {

        conut: 0

    },

    getters: {

        showNum(state){

            return "当前最新的数量是" + state.count

        }

    }

}

* 调用getters的方式一：

this.$store.getters.showNum

vue v-cloak 指令

可以使用 v-cloak 指令设置样式，这些样式会在 Vue 实例编译结束时，从绑定的 HTML 元素上被移除。

当网络较慢，网页还在加载 Vue.js ，而导致 Vue 来不及渲染，这时页面就会显示出 Vue 源代码。我们可以使用 v-cloak 指令来解决这一问题。

在简单项目中，使用 v-cloak 指令是解决屏幕闪动的好方法。但在大型、工程化的项目中（webpack、vue-router）只有一个空的 div 元素，元素中的内容是通过路由挂载来实现的，这时我们就不需要用到 v-cloak 指令咯。

## **vue-rouder原理**

**什么是单页应用？**

单页应用的全称是 single-page application，简称 SPA，它是一种网站应用的模型，它可以动态重写当前的页面来与用户交互，而不需要重新加载整个页面。

**工作原理：**

SPA 的一个重要实现就是改变路由时，页面不刷新。实现这个功能，通常有两种方式：使用 window.history 对象或 location.hash。

1. hash模式：url后面的hash值的变化，浏览器既不会向服务器发出请求，浏览器也不会刷新，每次hash值的变化会触发hashchange事件。
2. history模式，利用HTML5中新增的pushState()和replaceState()方法，这两个方法应用于浏览器的历史记录栈，在当前已有的back、forward、go的基础之上，它们提供了对历史记录进行修改的功能。只是当它们执行修改时，虽然改变了当前的URL，但浏览器不会立即向后端发送请求。

* replaceState()方法使用state objects、title和URL作为参数，修改当前历史记录实体

语法：history.replaceState(stateObj, title[, url])

stateObject 状态对象是一个JavaScript对象；

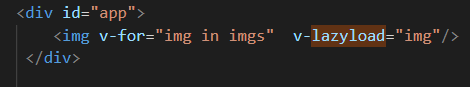
url可选，新的URL和当前的URL必须是同源的。

## Vue如何清除浏览器缓存

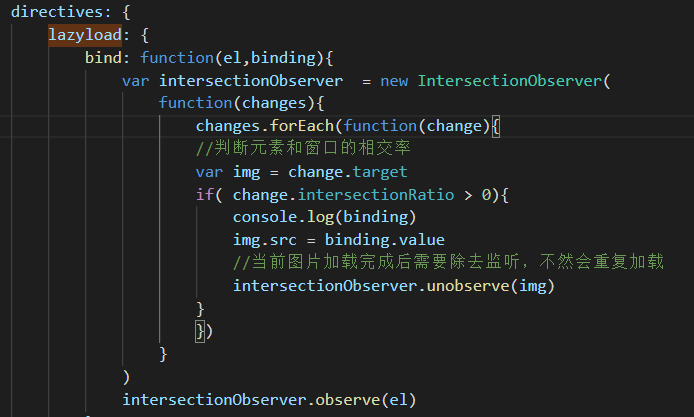
1. 项目打包的时候给每个打包文件加上hash值，一般是在文件后面加上时间戳；
2. 在html文件中加入meta标签，content属性设置为no-cache；
3. 在后端服务器中进行尽职缓存设置。

## ****vm.$refs****

## Vue自定义指令懒加载







## CSS scoped穿透

[https://vue-loader.vuejs.org/zh/guide/scoped-css.html#](https://vue-loader.vuejs.org/zh/guide/scoped-css.html)

**什么是scoped？**

在vue文件中的style标签上有一个特殊的属性，scoped。当一个style标签有scoped属性的时候，它的css样式只能用于当前的vue组件，可以使组件的样式不相互污染。如果一个项目的所有style标签都加上了scoped属性，相当于实现了样式的模块化。

**Scoped的实现原理**

vue中国的scoped属性的效果主要是通过postCSS实现的，

|  |
| --- |
| <style scoped>  .example {  color: red;  }  </style>  <template>  <div class="example">hi</div>  </template> |

通过postCSS转译后

|  |
| --- |
| <style>  .example[data-v-f3f3eg9] {  color: red;  }  </style>  <template>  <div class="example" data-v-f3f3eg9>hi</div>  </template> |

## Mixin

全局注册一个混入，影响注册之后所有创建的每个vue实例。

插件作者可以使用混入，像组件注入自定义的行为。

## 如何写一个插件

插件通常用来为Vue添加全局功能。插件的功能范围没有严格的限制——一般有下面几种：

1. 添加全局方法或者property。如：vue-custom-element
2. 添加全局资源：指令/过滤器/过渡等。如：vue-touch
3. 通过全局混入来添加一些组件选项。如vue-router
4. 添加vue实例方法，通过把它们添加到Vue.prototype上实现。
5. 一个库，提供自己的API，同时提供上面提到的一个或多个功能。如vue-router

**使用插件**

通过全局方法Vue.use()使用插件。它需要在你调用new Vue()启动应用之前完成：

**开发插件**

Vue.js的插件应该暴露一个install [ɪnˈstɔːl] 方法。这个方法的第一个参数是Vue构造器，第二个参数是一个可选的选项对象。

**大白话Vue.extends**

new Vue({}) options是一个空对象

new Vue({/\*something\*/}) 有【内容的实例】

Vue.extend({/\*something\*/}) 返回一个【有内容的类】，假设赋值给一个VueWithSoming，然后你就可以这么用：new VueWithSoming()。它返回的是【有内容的实例】，因为这个构造器本身带有【内容】。

## 编译之AST optimize优化

AST 优化（optimize）阶段标记静态节点和静态根节点

1. 递归遍历AST节点，判断每个节点是否是静态的；
2. 递归遍历父节点中所有子节点是否是静态的，如果所有子节点都为静态，则该父节点为静态根节点；
3. 在生成render函数阶段，判断一个节点是否是一个静态根节点，如果是静态根节点，生成静态render函数。
4. Render函数生成vnode阶段，如果是静态render函数，将其生成的vnode进行缓存，等到下次再执行render函数的时候直接从缓存中取出。

生成vnode的

标记静态根节点是为了缓存vnode，减少render函数生成vnode的时间。

## VUE单页面白屏问题

## VUE SSR 服务端渲染

# Vue3

## Composition （组合式）API

Vue2中，随着功能的增加，组件变得越来越复杂，越来越难维护，而难以维护的根本原因是Vue的API设计迫使开发者使用watch、computed、methods选项组织代码，而不是实际的业务逻辑。

另外vue2缺少一种较为简洁的低成本的机制来完成逻辑复用，虽然可以minxis完成逻辑复用，但是当minxi变多的时候，会使得难以找到对应的data、computed或者methods来源于哪个mixin，使得类型推断难以进行。

所以composition API的出现，主要是为了解决options API带来的问题，第一个是代码组织问题，composition API可以让开发者根据业务逻辑组织自己的代码，让代码具备更好的可读性和可拓展性。第二个是实现代码的逻辑提取和复用。

Composition API在setup函数中执行。

Mixins的最大的问题是命名冲突。



## 编译方面的优化

标记和提升所有的静态节点，diff的时候只需要对比动态节点的内容。

* **静态提升**

所有的静态节点都被提升到render方法之外，只会在应用启动的时候被创建一次，只会使用只需要应用提取的静态节点，随着每次的渲染被不断的复用。

## Patch flag

在动态标签末尾加上响应的标记，只有带patchFlag的节点才被认为是动态的元素，会被追踪属性的修改，能快速的找到动态节点，而不用逐个逐层遍历，提高了虚拟dom diff的性能。

# 前端性能优化

## 前端性能优化

1. 减少http请求

雪碧图

Background-position属性设置背景图像的起始位置。

1. 使用HTTP2
2. 使用服务器端渲染
3. 静态资源使用CDN

内容分发网络（CDN）是一组分布在多个不同地理位置的Web服务器。我们都知道，当服务器离用户越远时，延迟越高。CDN就是为了解决这一问题，在多个位置部署服务器，当用户离服务器更近，从而缩短请求时间。

1. 将CSS放在文件头部，JavaScript文件放在底部

CSS不会阻塞html生成DOM树，但会阻塞浏览器布局。

1. 使用字体图标iconfont替换图片图标
2. 善用缓存，不重复加载相同的资源
3. 压缩文件

JavaScript ： UglifyPlugin

css : MiniCssExtractPlugin

HTML : HtmlWebpackPlugin

Gzip : 可以通过向HTTP请求头中的Accept-Encoding头添加gzip标识来开启这一功能。当然，服务器也得支持这一功能。

1. 图片优化
2. 通过webpack按需加载代码，提取第三库代码，减少ES6转为ES5的冗余代码
3. 减少重绘重排
4. 使用事件委托/事件代理
5. 注意程序的局部性
6. If-else对比switch
7. 查找表
8. 避免页面卡顿
9. 使用requestAnimationFrame来实现视觉变化
10. 使用Web Workers
11. 使用位操作
12. 不要覆盖原生方法
13. 降低css选择器的复杂性
14. 使用flexbox而不是最早的布局模型
15. 使用transform和opacity [əʊˈpæsəti] 属性更改来实现动画

在css中，transforms和opacity这两个属性更改不会触发重排和重绘，它们是可以有合成器单独处理的属性。

1. 合理使用规则，避免过度优化

FCP（首屏渲染时间）

RTT（HTTP的往返时间）

前后端分离导致首屏需要请求更多的内容，导致更多HTTP的往返时间，这造成了白屏。

## 优化首屏渲染时间，减少白屏时间的一些方案

* 加速或减少HTTP请求耗损

1. 使用CDN加载公用库
2. 使用强缓存和协商缓存
3. 使用域名收敛
4. 小图片使用Base64代替
5. 使用Get请求代替Post请求
6. 设置Access-Control-Max-Age减少预检请求，
7. 页面内跳转其他域名或请求使用其他域名的资源时使用浏览器prefetch预解析等

延迟加载

* 减少请求内容的体积

非重要的库、非首屏图片延迟加载、SPA的组件懒加载等；

* 浏览器渲染原理

1. 开启服务器Gzip压缩
2. JS、CSS文件压缩合并
3. 减少cookies大小
4. SSR直接输出渲染后的HTML等

* 优化用户等待体验

白屏使用加载进度条、菊花图、骨架屏代替等。

## Css和JS的位置会影响页面效率

Css是页面渲染的关键因素之一，当页面存在外链css时，浏览器会等待全部的css下载及解析完成后再渲染页面。关键路径上的任何延迟都会影响首屏时间，因此我们需要尽快的将css传输到用户的设备，否则，用户只能看到一个空白的屏幕

Css在加载过程中不会影响到DOM树的生成，但是会影响render树的生成，进而影响到布局，所以一般来说，style的link标签需要尽量放在head里面，因为在解析DOM树的时候是自上而下的，而css样式又是通过异步加载的，这样的话，解析DOM树下的body节点和加载css样式能尽可能的并行，加快render树的生成速度。

Js脚本应该放在底部，原因在于js线程和GUI渲染线程是互斥的关系，如果js放在首部，当下载执行js的时候，会影响渲染线程绘制页面。

# Vue性能优化

* **代码层面的优化**

1、v-if和v-show区分使用场景

v-if适用于在运行时很少改变条件，不需要频繁切换条件的场景；

v-show则适用于需要非常频繁切换条件的场景。

2、computed和watch区分使用场景

3、v-for遍历必须为item添加key，且避免同时使用v-if

4、长列表性能优化

纯粹的数据展示，不会有任何改变，我们不需要vue来劫持我们的数据，在大量数据展示的情况下，这能够明显的减少组件初始化的时间。使用Object.freeze来冻结一个对象。

5、事件销毁

Vue组件销毁时，会自动清理它与其它实例的连接，解绑它的全部指令及事件监听器，但是仅限于组件本身的事件。如果在js内使用addEventListene等方式是不会自动销毁的，所以我们在组件销毁时需要手动移除这些事件的监听，以免造成内存泄露。

6、图片懒加载

7、路由懒加载

Vue是单页面应用，可能会有很多的路由引入，这样使用webpack打包后的文件很大，当进入首页时，加载的资源过多，页面会出现白屏的情况，不利于用户体验。

如果我们能把不同路由对应的组件分割成不同的代码块，然后当路由被访问的时候才加载对应的组件，这样就更加高效了。

1. 第三方插件按需引入
2. 优化无限列表性能

如果是非常长或者无限滚动的列表，那么需要采用窗口化的技术来优化性能，只需要渲染少部分区域dom节点的时间。

可参考以下开源项目：vue-virtual-scroll-list和vue-virtual-scroller

10、服务器渲染SSR或者预渲染

* **Webpack配置层面的优化**
* **基础的web技术层面的优化**

# 微信小程序weapp

<https://www.cnblogs.com/teahouse/p/11504361.html>

<https://developers.weixin.qq.com/community/develop/doc/d1421cd729a51548672430e544c458b2>

## weapp的运行环境

根据微信官方说明，weapp的运行环境有3个平台，IOS的webkit（苹果开源的浏览器内核）、android的X5（QQ浏览器内核），开发时用的nw.js（c++实现的web转桌面应用）。

## Weapp为什么不直接运行在浏览器（webview）中，而要绕过浏览器直接调用内核

因为运行在浏览器中的weapp是做不了监控的，而weapp的表现是半native app，半web app，而native app和web app的一个很重要的区别就是native app有自己的生命周期，在这之中，我们可以根据生命周期的不同时间段做出不同的调整，比如常驻内存，防止被系统杀掉、系统后台保存活度等等，而web app是没有这回事的，

# 项目介绍

<https://www.v2ex.com/t/672672>

<https://www.v2ex.com/t/645335>

## 人事管理系统

**项目难点**

* SKU
* 骨架屏（Skeleton Screen）
* 长列表优化
* Vue大文件上传和断点续传
* 前端监控系统
* 异常问题排查

**遇到的难点、问题及解决办法**

|  |  |
| --- | --- |
| **1** | * 大文件上传，如视频200M   <1>上传文件类型的判断；不是通过后缀名判断；而是转化为二进制流，判断头信息  <2>文件切片 Blob.slice  <3>断点续传  <4>文件算hash，用hash做唯一标识  方式1：webwork 防止卡顿线程  this.worker = new Worker(‘/’)  方法二：浏览器每一帧都有空闲时间  流畅的动画或者网页60FPS，也就是1秒渲染60次，1帧大概16.6ms  卡顿造成的原因：同步任务的耗时远大于16ms  我们可以利用空闲来计算，一旦有一个优先级更高的同步任务，返回浏览器控制权，等待下一次空闲，所谓浏览器的fiber架构，就是这个理念。  Fiber：   1. 虚拟dom从树拆成链表 易于中断 2. Diff过程用空闲时间来做   Time-slice  requestIdleCallback  方法三、  布隆过滤器 牺牲一点进度 获取更高的效率  <5>时间切片  <6>控制并发数，异步任务并发 + 报错重试  <7> |
| **2** | * 动态生成页面，页面input、select太多，直接传参数（组件类型，id、方法等），通过if判断生成，无需重复编写html代码 |
| **3** | * 页面使用easyUI的输入框、下拉框、按钮太多了，导致页面渲染非常的慢。   这个页面是一个员工信息管理页面，主要包含几十列的table列表、几十个筛选项、几十个编辑项；  解决办法：  方法一：按需加载  easyui是通过class来标识的，首先将所有的对应的easyui class换成与easyui无关的class；  其次，在每次需要使用的时候，再转化为easyui对应的class，比如：点击编辑以后，再将编辑框里面的所有项的class加上easyui对应的class；  方法二：能不是使用easyui的地方就不要使用，这样会导致整体的样式不一致，但我们可以将样式修改成一样的。 |
| **4** | * 我们这个系统很多的页面涉及到上传的功能，一般有图片、pdf、excel的上传   它存在的问题是，每一处的上传都由各自的代码，很多处的代码都是复制粘贴过去的；  我需要解决的是，去除重复代码，将上传功能提取到公共文件里面，前前后后有做过大约3次大的变动；  第一次就是将以前的上传代码保留一份，放入公共模块，供使用的地方调取；  第二次改动是因为发现前面写的上传公共方法不支持一个页面有多处上传的情况  第三次改动是因为ios的钉钉不支持图片的上传  再后面是写了vue版本的公共上传组件。 |
| **5** | * 一个vue页面，它引用了三个子组件；其中两个子组件的接口需要依赖另外一个子组件接口的返回值。   B组件需要依赖A组件里面的接口返回值；  A组件创建（created）和B组件挂载（mounted）时间间隔，比A组件接口返回时间短，  所以在A组件创建阶段请求接口，B组件挂载阶段请求接口不行。  Vuex设置一个uid，默认值为0，A组件接口接口返回后给uid赋值，  B组件接口判断uid是否为0，如果是0的话等待100ms，再判断，如此一直循环，直到uid不为0，再继续请求B接口 |
| **6** | * 异常问题排查   北京分公司的一位同事访问erp.office.51fanli.com的广告排期相关的页面返回空白页； |
| **7** | * 使用jOrgChart实现组织架构树 * Dom节点生成png，支持将架构树转化为图片下载，要熟悉下原理   <https://github.com/tsayen/dom-to-image/blob/master/src/dom-to-image.js> |

## 爱予亲子阅读微信小程序

在这个项目是我自己独立开发的，我做过的事情有：

购买服务器、域名、

搭建运行环境，linux系统，使用nginx部署，配置https，

在这个项目中使用到微信接口有：

1. 小程序更新
2. 获取用户信息
3. 微信支付功能
4. 微信模板发送信息

遇到的难点、问题及解决办法

* ios8卡顿问题

scroll-view的bindscrolltolower下拉刷新确实会存在异常卡顿

解决方式：使用view再使用onReachBottom（页面上拉触底事件的处理函数）

* 登录保持

## 广告排期系统

这个系统里面涉及到了一个排期面板，这个是比较复杂的

## 同事圈

详情页返回列表页保持

keep-alive

路由的地方增加meta

实现滚动行为的代码：router/index.js

|  |
| --- |
| scrollBehavior(to, from, savePosition){  if(savePosition){  return savePosition  }else{  return {x:0,y:0}  }  } |

表情插件

1. 使用css 雪碧图（精灵图）将所有的小表情合并为一张图片，使用background-position显示指定的小表情；
2. 数据库存储小表情英文别名，取出来后，替换为dom
3. 使用div的contenteditable=”true”来实现小表情的输入

# 网上学习链接

<https://blog.csdn.net/weixin_42623421/article/details/106856141>

# 场景题

## 登录

第一步：服务器配置

在php.ini文件中配置（session.name可以自定义）

|  |
| --- |
| Session.name =PHPSESSID22  Session.auto\_start=0 |

第二步：浏览器发送第一个http请求时，服务器返回一个sessionID给浏览器

具体形式为：http请求的response中设置set-cookie

则sessionID保存在了浏览器的cookie中

|  |
| --- |
| Set-Cookie: PHPSESSID22=aqu7hrh85rhsrc4qe821behvm6; path=/ |

第三步：浏览器发送第二个http请求，http请求的request中携带cookie

|  |
| --- |
| Cookie: PHPSESSID22=aqu7hrh85rhsrc4qe821behvm6 |

第四步：服务器验证是否登录，没有登录跳转到登录页

第五步：浏览器post参数携带用户名和MD5加密的密码，服务器通过用户名、密码及其他参数生成一个key存储在服务器的session中，验证是否登录就是判断这个session是否存在

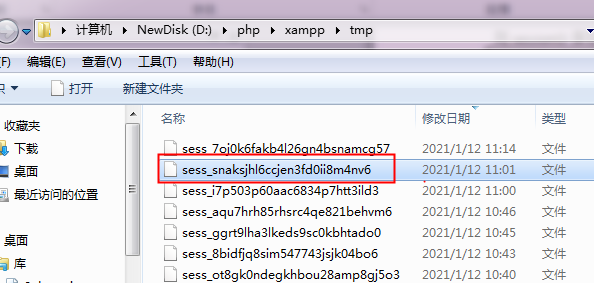
## PHP的session机制《PHPSESSID》

1. 默认机制，用磁盘文件来实现PHP会话。Php.ini配置如下

|  |
| --- |
| session.save\_handler=files  session.save\_path="D:\php\xampp\tmp" |

1. session\_start()

A、session\_start()是session机制的开始，



## JAVA的session机制 《JSESSIONID》

<https://www.cnblogs.com/barneywill/p/10116996.html>

## 单点登录（SSO）

上图是CAS官网上的标准流程，具体流程如下：

1. 用户访问app系统，app系统是需要登录的，但用户现在没有登录。
2. 跳转到CAS server，即SSO登录系统，**以后图中的CAS Server我们统一叫做SSO系统。** SSO系统也没有登录，弹出用户登录页。
3. 用户填写用户名、密码，SSO系统进行认证后，将登录状态写入SSO的session，浏览器（Browser）中写入SSO域下的Cookie。
4. SSO系统登录完成后会生成一个ST（Service Ticket），然后跳转到app系统，同时将ST作为参数传递给app系统。
5. app系统拿到ST后，从后台向SSO发送请求，验证ST是否有效。
6. 验证通过后，app系统将登录状态写入session并设置app域下的Cookie。

至此，跨域单点登录就完成了。以后我们再访问app系统时，app就是登录的。接下来，我们再看看访问app2系统时的流程。

1. 用户访问app2系统，app2系统没有登录，跳转到SSO。
2. 由于SSO已经登录了，不需要重新登录认证。
3. SSO生成ST，浏览器跳转到app2系统，并将ST作为参数传递给app2。
4. app2拿到ST，后台访问SSO，验证ST是否有效。
5. 验证成功后，app2将登录状态写入session，并在app2域下写入Cookie。

这样，app2系统不需要走登录流程，就已经是登录了。SSO，app和app2在不同的域，它们之间的session不共享也是没问题的。

**有的同学问我，SSO系统登录后，跳回原业务系统时，带了个参数ST，业务系统还要拿ST再次访问SSO进行验证，觉得这个步骤有点多余。他想SSO登录认证通过后，通过回调地址将用户信息返回给原业务系统，原业务系统直接设置登录状态，这样流程简单，也完成了登录，不是很好吗？**

**其实这样问题时很严重的，如果我在SSO没有登录，而是直接在浏览器中敲入回调的地址，并带上伪造的用户信息，是不是业务系统也认为登录了呢？这是很可怕的。**

总结

单点登录（SSO）的所有流程都介绍完了，原理大家都清楚了。总结一下单点登录要做的事情：

* **单点登录（SSO系统）是保障各业务系统的用户资源的安全 。**
* **各个业务系统获得的信息是，这个用户能不能访问我的资源。**
* **单点登录，资源都在各个业务系统这边，不在SSO那一方。 用户在给SSO服务器提供了用户名密码后，作为业务系统并不知道这件事。 SSO随便给业务系统一个ST，那么业务系统是不能确定这个ST是用户伪造的，还是真的有效，所以要拿着这个ST去SSO服务器再问一下，这个用户给我的ST是否有效，是有效的我才能让这个用户访问。**

# 智力题

**1、25匹马，5匹一组跑，跑几次可以找到前三名**

答：7 次

第一次： 5组各自跑一次，找到每组的第一名；

第二次： 5个第一名跑一次，找到前三名，此时已经知道最终的第1名

第三次：第1名组的第2和第3、第2组的第1和第2，第3组的第1 ，跑一次；

就可以知道最终的第2名和第3名

所以是5+1+1=7次

**2、5点15分，时针和分针的夹角是多少度？**

答：12个小时，每个小时的占360/12=30度，5小时=5\*30=150度

分针走一圈是1小时，1小时=60分钟，则1分钟占360/60=6度，15分钟=15\*6=90度

150度和90度的夹角是60度；

但分针走到15分钟时时针也有移动，1小时等于60分钟， 那么15分钟等于15/60=0.25小时；

1小时等于30度，那么0.25小时等于30\*0.25=7.5度；

所以5点15分，时针和分针的夹角是67.5度。

**3、如果你有两个桶，一个装红颜料，另一个装蓝颜料，两个桶的颜料一样多。你从蓝颜料里舀一杯，倒入红色颜料桶，再从红色颜料桶里舀一杯倒入蓝色颜料桶，假设红桶中的红色/蓝色 为 A，蓝桶中的蓝色/红色 为 B，A 和 B 哪个大？**

一样大；

假设红色和蓝色桶里面的颜料都为100毫升

蓝色往红色桶里倒入10毫升，则红色桶中有100毫升红色和10毫升蓝色

再从红色桶倒入蓝色桶10毫升，此时设置这10毫升颜料里面有x毫升红色和y毫升蓝色，则有以下公式：

X+y=10

X/Y = 10/100 = 1/10

计算出X=10/11 ; y=100/11

则现在蓝色桶有90 + 10/11= 1000/11的蓝色 ， 有100/11的红色

红色桶有100-100/11=1000/11的红色，有10-10/11=100/11的蓝色

蓝色桶 1000/11和100/11的比

红色桶1000/11和100/11的比

# 算法题

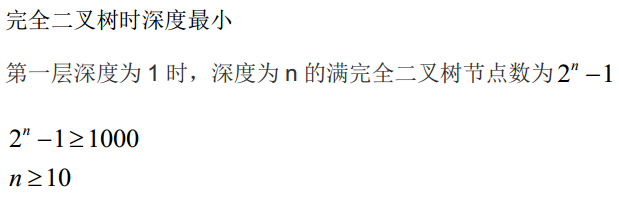
**1、一个二叉树有1000个节点，最小深度？**

答：最小深度为10

**2、一个树形结构，如果输入一个id找到他所在测children**

3、**二分查找算法**

二分查找也称折半查找，它是一种效率较高的查找方法。但是，折半查找要求线性表必须采用顺序存储结构，而且表中元素按关键字有序排列。



# 面试总结

众安：前后端开发 三面通过，薪资没谈拢

永辉 ：测试 一面挂，很惨

河姆渡：前端 两面通过，

掌门1对1：前端 1面挂

饿了吗：前端 1面挂

小红书：前端 1面挂

鱼泡泡/比心陪练： 1面挂

票易通：二面挂 2021-02-02~2021-02-09

众安前端面经

本周面了众安前端开发的岗位。 因为在职，HR安排了线上视频面试，比较方便，不耽误时间。 前端面试官是个小姐姐，声音特别亲切，面试时候也基本是在和你讨论的感觉。 问了VUE\React框架；ES6，CSS的一些应用场景；304是什么；node相关问题；还出了一个题目，有一个接口，里面ABC3种类型，现在想用接口A这个类型，如何捡出来 最后还发了一串代码让我写 虽然好几个问题都没有答上来，但是小姐姐会给提示，也会和你讨论 估计挂了，哎

对盒子模型的了解