**跨域相关，同源策略，跨域办法 - jsonp以及缺点 / webSocket协议 / CORS以及分类，预检请求 --24--**

跨域问题是由于javascript语言安全限制中的同源策略造成的

js跨域是指通过js在不同的域之间进行数据传输或通信，比如用ajax向一个不同的域请求数据，或者通过js获取页面中不同域的框架中(iframe)的数据。只要协议、域名、端口有任何一个不同，都被当作是不同的域。

1. **jsonp跨域：**在js中，不能直接用XMLHttpRequest请求不同域上的数据。但是，在页面上（script标签）引入不同域上的js脚本文件却是可以的，jsonp正是利用这个特性来实现的（script标签不受同源策略的限制。）。

通过动态生成script标签引入一个js文件，这个js文件载入成功后会执行我们在url参数中指定的函数（url最后加上的callback=xxx），并且会把我们需要的json数据作为参数传入。所以jsonp是需要服务器端的页面进行相应的配合的。

jquey中封装的$.getJSON方法会自动判断是否跨域，不跨域的话，就调用普通的ajax方法；跨域的话，则会以异步加载js文件的形式来调用jsonp的回调函数。

缺点：只支持get请求、安全问题(请求代码中可能存在安全隐患)、要确定jsonp请求是否失败并不容易

1. **跨域资源共享（CORS）：**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。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。

服务器端对于CORS的支持，主要就是通过设置Access-Control-Allow-Origin来进行的。如果浏览器检测到相应的设置，就可以允许Ajax进行跨域的访问。

请求路径需要是绝对路径

与JSONP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、使用window.name来进行跨域：**window对象有个name属性，该属性有个特征：即在一个窗口(window)的生命周期内,窗口载入的所有的页面都是共享一个window.name的，每个页面对window.name都有读写的权限，window.name是持久存在一个窗口载入过的所有页面中的，并不会因新页面的载入而进行重置。

**4、HTML5的window.postMessage方法来跨域传送数据:**

window.postMessage(message,targetOrigin)  方法是html5新引进的特性，可以使用它来向其它的window对象发送消息，无论这个window对象是属于同源或不同源

调用postMessage方法的window对象是指要接收消息的那一个window对象，该方法的第一个参数message为要发送的消息，类型只能为字符串；第二个参数targetOrigin用来限定接收消息的那个window对象所在的域，如果不想限定域，可以使用通配符 \*  。

需要接收消息的window对象，可以通过监听自身的message事件来获取传过来的消息，消息内容储存在该事件对象的data属性中。

**5、web sockets：**web sockets是一种浏览器的API，它的目标是在一个单独的持久连接上提供全双工、双向通信。(同源策略对web sockets不适用)

web sockets原理：在JS创建了web socket之后，会有一个HTTP请求发送到浏览器以发起连接。取得服务器响应后，建立的连接会使用HTTP升级从HTTP协议交换为web sockt协议。

只有在支持web socket协议的服务器上才能正常工作。

1. **通过修改document.domain来跨子域：**浏览器都有一个同源策略，其限制之一就是第一种方法中我们说的不能通过ajax的方法去请求不同源中的文档。 它的第二个限制是浏览器中不同域的框架之间是不能进行js的交互操作的。

有一点需要说明，不同的框架之间（父子或同辈），是能够获取到彼此的window对象的，但却不能使用获取到的window对象的属性和方法

比如，有一个页面，它的地址是http://www.example.com/a.html  ， 在这个页面里面有一个iframe，它的src是http://example.com/b.html, 很显然，这个页面与它里面的iframe框架是不同域的，所以我们是无法通过在页面中书写js代码来获取iframe中的东西的：

这个时候，document.domain就可以派上用场了，我们只要把http://www.example.com/a.html 和 http://example.com/b.html这两个页面的document.domain都设成相同的域名就可以了。但要注意的是，document.domain的设置是有限制的，我们只能把document.domain设置成自身或更高一级的父域，且主域必须相同。

修改document.domain的方法只适用于不同子域的框架间的交互。

**垂直居中 --19--**

1. **行内元素：text-align + line-height**(等于父元素的高度)
2. **定宽定高：absolute + margin。**直接计算，或者top、bottom设为0，margin:auto 0
3. **不定高：absolute + translate。**百分比，top:50%,transform:translateY(-50%)
4. **不定高：flex。**给父元素display:flex;而子元素align-self:center;align-self属性允许单个项目有与其他项目不一样的对齐方式，可覆盖align-items属性
5. **table-cell布局**：****给父元素display:table，子元素display：table-cell 和vertical-align:middle
6. **通过vertical-align:middle**实现CSS垂直居中是最常使用的方法，但是有一点需要格外注意，vertical生效的前提是元素的display：inline-block。

**水平垂直居中：**

1. **absolue + margin：**top、right...设为0，margin: auto
2. **absolue + translate：**top: 50% left: 50% transform: translate(-50%, -50%)
3. **flex布局：**display: flex; justify-content: center;//使子项目水平居中 align-items: center;//使子项目垂直居中
4. **table-cell布局：**给父元素display:table，子元素display：table-cell 和vertical-align:middle以及text-align: center

**强缓存，协商缓存 - 相关的头部、状态码 --19--**

浏览器缓存分为强缓存和协商缓存，浏览器加载一个页面的简单流程如下：

1、浏览器先根据这个资源的http头信息来判断强缓存是否过期。如果没有过期则直接加载缓存中的资源，并不会将请求发送到服务器。状态码为200

2、如果强缓存过期，则浏览器会将资源加载请求发送到服务器。服务器来判断浏览器本地缓存是否失效（协商缓存）。若可以使用，则服务器并不会返回资源信息，浏览器继续从缓存加载资源。状态码为304

3、如果未命中协商缓存，则服务器会将完整的资源返回给浏览器，浏览器加载新资源，并更新缓存。状态码为200

**强缓存**

命中强缓存时，浏览器并不会将请求发送给服务器。在Chrome的开发者工具中看到http的返回码是200，但是在Size列会显示为(from cache)。

强缓存是利用http的返回头中的Expires或者Cache-Control两个字段来控制的，用来表示资源的缓存时间。

**Expires**

缓存过期时间，用来指定资源到期的时间，是服务器端的具体的时间点。也就是说，Expires=max-age + 请求时间，需要和Last-modified结合使用。

**Cache-Control**

Cache-Control是一个相对时间，例如Cache-Control:3600，代表着资源的有效期是3600秒。由于是相对时间，并且都是与客户端时间比较，所以服务器与客户端时间偏差也不会导致问题。  
Cache-Control与Expires可以在服务端配置同时启用或者启用任意一个，同时启用的时候Cache-Control优先级高。

Cache-Control 可以由多个字段组合而成，主要有以下几个取值：

1. **max-age**指定一个时间长度，在这个时间段内缓存是有效的，单位是s。

2. **s-maxage** 同 max-age，覆盖 max-age、Expires，但仅适用于共享缓存，在私有缓存中被忽略。

3. **public** 客户端和代理服务器都可以缓存该资源

4. **private** 只让客户端可以缓存该资源；代理服务器不缓存

5. **no-cache**强制所有缓存了该响应的用户，在使用已缓存的数据前，发送带验证器的请求到服务器。不是字面意思上的不缓存。（不使用强缓存）

6**. no-store**禁止缓存，每次请求都要向服务器重新获取数据。（强缓存和协商缓存都不用）

**协商缓存**

若未命中强缓存，则浏览器会将请求发送至服务器。服务器根据http头信息中的Last-Modify/If-Modify-Since或Etag/If-None-Match来判断是否命中协商缓存。如果命中，则http返回码为304，浏览器从缓存中加载资源。

先验证Etag/If-None-Match，相同的情况下再验证Last-Modify/If-Modify-Since。

**Last-Modify/If-Modify-Since**

浏览器第一次请求一个资源的时候，服务器返回的header中会加上Last-Modify，Last-modify是一个时间标识该资源的最后修改时间

当浏览器再次请求该资源时，发送的请求头中会包含If-Modify-Since，该值为缓存之前返回的Last-Modify。服务器收到If-Modify-Since后，根据资源的最后修改时间判断是否命中缓存。如果命中缓存，则返回状态码304，并且不会返回资源内容（直接从本地缓存获取），并且不会返回Last-Modify。

**ETag/If-None-Match**

与Last-Modify/If-Modify-Since不同的是，Etag/If-None-Match返回的是一个校验码（ETag: entity tag）。ETag可以保证每一个资源是唯一的，资源变化都会导致ETag变化\*。ETag值的变更则说明资源状态已经被修改。服务器根据浏览器上发送的If-None-Match值来判断是否命中缓存。

HTTP1.1中Etag的出现主要是为了解决几个Last-Modified比较难解决的问题：

1. Last-Modified标注的最后修改只能精确到秒级，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修改时间

2. 如果某些文件会被定期生成，当有时内容并没有任何变化，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

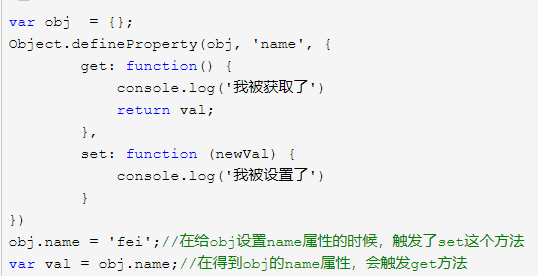
Etag是服务器自动生成或者由开发者生成的对应资源在服务器端的唯一标识符，能够更加准确的控制缓存。

Last-Modified与ETag是可以一起使用的，服务器会优先验证ETag，一致的情况下，才会继续比对Last-Modified，最后才决定是否返回304。

**Vue的双向绑定原理 --16--**

vue.js 是采用数据劫持结合发布者-订阅者模式的方式，通过Object.defineProperty()来劫持各个属性的setter，getter，在数据变动时发布消息给订阅者，触发相应的监听回调。

待深入了解···



**实现两栏（左固定 右自适应）/三栏（左右固定 中间自适应）布局 --16--**

**两栏布局：**

1. **使用浮动：**左边定宽高浮动，右边加上margin-left:左元素的宽度
2. **使用absolute：**父元素设为relative，左元素absolute，右元素加上margin-left
3. **使用table布局**：父元素设置display: table，两个子元素都设为display: table-cell
4. **使用flex布局：**父元素设置display: flex，右元素设置flex: 1(占据剩余空间)

**三栏布局：**

1. **使用浮动：**左右两边加上浮动，中间加上margin-left和margin-right
2. **使用absolute：**父元素relative，三个子元素都加上absolute，左边的设置left: 0右边的right: 0 中间的left和right为这两个元素的宽度
3. **使用table布局：**父元素试着display: table，三个子元素都设置display: table-cell，两边的元素设置宽度，中间不设置，宽度自适应
4. **使用flex布局：**父元素设置display: flex，左右两边设置宽度，中间设置flex: 1，会自动占据剩余空间
5. **使用网格布局（gird）**：不熟悉···

**状态码1XX-5XX --15--**

****HTTP状态码****表示客户端HTTP请求的返回结果、标记服务器端的处理是否正常或者是出现的错误，能够根据返回的状态码判断请求是否得到正确的处理很重要。



****各类别常见状态码****：

****2xx （3种）****

****200 OK：****表示从客户端发送给服务器的请求被正常处理并返回；

****204 No Content：****表示客户端发送给客户端的请求得到了成功处理，但在返回的响应报文中不含实体的主体部分（没有资源可以返回）；

****206 Patial Content：****表示客户端进行了范围请求，并且服务器成功执行了这部分的GET请求，响应报文中包含由Content-Range指定范围的实体内容。

****3xx （5种）****

****301 Moved Permanently：****永久性重定向，表示请求的资源被分配了新的URL，之后应使用更改的URL；

****302 Found：****临时性重定向，表示请求的资源被分配了新的URL，希望本次访问使用新的URL；

       301与302的区别：前者是永久移动，后者是临时移动（之后可能还会更改URL）

****303 See Other：****表示请求的资源被分配了新的URL，应使用GET方法定向获取请求的资源；

      302与303的区别：后者明确表示客户端应当采用GET方式获取资源

****304 Not Modified：****表示客户端发送附带条件（是指采用GET方法的请求报文中包含if-Match、If-Modified-Since、If-None-Match、If-Range、If-Unmodified-Since中任一首部）的请求时，服务器端允许访问资源，但是请求为满足条件的情况下返回改状态码；

****307 Temporary Redirect：****临时重定向，与303有着相同的含义，307会遵照浏览器标准不会从POST变成GET；（不同浏览器可能会出现不同的情况）；

****4xx （4种）****

****400 Bad Request：****表示请求报文中存在语法错误；

****401 Unauthorized：****未经许可，需要通过HTTP认证；

****403 Forbidden：****服务器拒绝该次访问（访问权限出现问题）

****404 Not Found：****表示服务器上无法找到请求的资源，除此之外，也可以在服务器拒绝请求但不想给拒绝原因时使用；

****5xx （2种）****

****500 Inter Server Error：****表示服务器在执行请求时发生了错误，也有可能是web应用存在的bug或某些临时的错误时；

****503 Server Unavailable：****表示服务器暂时处于超负载或正在进行停机维护，无法处理请求；

**Eventloop，可以顺便说一下浏览器渲染进程，哪些是宏任务微任务 - 经典看代码写输出 --15--**

首先，JS是单线程的语言，所以就是说在一个任务执行的过程中，必然不会存在同时执行的另一个任务。Event Loop就是javascript的执行机制，用来决定任务执行的顺序。

所有任务可以大致分为两种：同步任务和不能立即得到结果的异步任务。

当JS执行过程遇到同步任务时会直接在主线程执行，如果是异步任务则是执行后将其挂起，等待该异步任务返回结果后，将这个任务放入事件队列，但还不会执行。而是等待主线程的所有任务都执行完毕后再将事件队列里的任务放到主线程执行。期间遇到的异步任务会继续放入事件队列，等待当前主线程任务执行完毕后再执行，如此反复的循环过程就是事件循环。

异步任务还能再细分为宏任务和微任务，它们之间的执行顺序也不同。

根据这个异步事件的类型，这个事件实际上会被放到对应的宏任务队列或者微任务队列中去。并且在当前执行栈为空的时候，主线程会查看微任务队列是否有事件存在。如果不存在，那么再去宏任务队列中取出一个事件并把对应的回到加入当前执行栈；如果存在，则会依次执行队列中事件对应的回调，直到微任务队列为空，然后去宏任务队列中取出最前面的一个事件，把对应的回调加入当前执行栈...如此反复，进入循环。

当当前执行栈执行完毕时会立刻先处理所有微任务队列中的事件，然后再去宏任务队列中取出一个事件。同一次事件循环中，微任务永远在宏任务之前执行。

宏任务一般是：包括整体代码script，**setTimeout**，**setInterval**。setImmediate, requestAnimationFrame, I/O, UI rendering

微任务：**Promise**，process.nextTick。

整体代码script作为第一个宏任务执行结束，然后看看有哪些微任务

async/await本质上还是基于Promise的一些封装，而Promise是属于微任务的一种。所以在使用await关键字与Promise.then效果类似。async函数在await之前的代码都是同步执行的，可以理解为await之前的代码属于new Promise时传入的代码，await之后的所有代码都是在Promise.then中的回调

**TCP和UDP的区别，TCP拥塞控制是如何实现的 --14--**

TCP全称为传输控制协议，为应用最广的网络通信协议。UDP全称为用户数据报协议，它是在TCP之后推出的，与TCP互为补充，UDP与TCP相比并没有增加什么新的功能，反倒是取消了如顺序保证、丢包重发等几项功能，这也是UDP的优势所在。

再具体地讲的话：TCP和UDP的区别在于

1. TCP是面向连接的，连接之前需要进行三次握手，关闭连接时还得四次挥手，耗时较长。而UDP是无连接的，也就是发送数据之前不需要建立连接

2、TCP提供的是可靠的连接。通过TCP连接传送的数据无差错、不丢失、不重复且按序到达。UDP则没有这些保证，只管发送，最终接收到的数据什么样不确定

3、UDP具有更好的实时性，工作效率比TCP高，适用于对高速传输和实时性有较高的通信或广播通信。

4、TCP的连接只能一对一，而UDP可以一对一、一对多、多对多的交互通信

5、TCP占用系统资源较多、UDP占用资源较少

**[TCP字节流和UDP数据报区别](https://www.cnblogs.com/hoohack/p/4772365.html)**

两者的区别在于TCP接收的是一堆数据，而每次取多少由主机决定;而UDP发的是数据报，客户发送多少就接收多少。

**为什么UDP有时比TCP更有优势?**

UDP以其简单、传输快的优势，在越来越多场景下取代了TCP,如实时游戏。

（1）网速的提升给UDP的稳定性提供可靠网络保障，丢包率很低，如果使用应用层重传，能够确保传输的可靠性。

（2）TCP为了实现网络通信的可靠性，使用了复杂的拥塞控制算法，建立了繁琐的握手过程，由于TCP内置的系统协议栈中，极难对其进行改进。

采用TCP，一旦发生丢包，TCP会将后续的包缓存起来，等前面的包重传并接收到后再继续发送，延时会越来越大，基于UDP对实时性要求较为严格的情况下，采用自定义重传机制，能够把丢包产生的延迟降到最低，尽量减少网络问题对游戏性造成影响。

**JS数据类型 --11-**

基本数据类型：Null、Undefined、String、Number、Boolean、Symbol

引用数据类型：Object

**作用域链和原型链 --10--**

作用域就是变量和函数的可访问范围，控制着变量和函数的可见性与生命周期

作用域有全局作用域和局部作用域之分。全局作用域在任何地方都能访问到，局部作用域是只有特定的环境内才能访问到。

当一个新的作用域创建时（函数作用域和块作用域），会保存其父级作用域执行环境。在读取变量时，会先读取当前作用域中的值，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

JS的作用域为词法作用域而不是动态作用域，所以函数的作用域在定义时就已经决定

每一个JavaScript对象(null除外)在创建的时候就会与之关联另一个对象，这个对象就是我们所说的原型，每一个对象都会从原型"继承"属性。

查找属性或方法，如果本身没有，则会去\_\_proto\_\_中查找，也就是构造函数的显式原型中查找，如果原型对象中也没有该属性和方法，因为原型对象也是对象，也有\_\_proto\_\_，那么会继续向上查找原型，一直到null，如果没有则返回undefined。

通过\_\_proto\_\_形成原型链而非protrotype

以下三点需要谨记

1.每个实例对象都具有一个名为\_\_proto\_\_（隐式原型）的属性，指向自身构造函数的prototype

2.每个构造函数（构造函数标准为大写开头，如Function()，Object()等等JS中自带的构造函数，以及自己创建的）都具有一个名为prototype（显式原型）的方法（注意：既然是方法，那么就是一个对象（JS中函数同样是对象），所以prototype同样带有\_\_proto\_\_属性）；

3.每个原型对象都有一个 constructor 属性指向关联的构造函数

也就是实例对象与原型对象间有直接连接，构造函数与原型对象有直接连接，但构造函数与其实例对象之间没有直接连接。

**cookie、session、localStorage、sessionStorage --9--**

HTTP协议是无状态的协议。一旦数据交换完毕，客户端与服务器端的连接就会关闭，再次交换数据需要建立新的连接。这就意味着服务器无法从连接上跟踪会话。但从连接无法知道客户的身份。所以给客户端们颁发一个通行证吧，每人一个，无论谁访问都必须携带自己通行证。这样服务器就能从通行证上确认客户身份了。这就是Cookie的工作原理。

Cookie实际上是一小段的文本信息。客户端请求服务器，如果服务器需要记录该用户状态，就使用response向客户端浏览器颁发一个Cookie。客户端浏览器会把Cookie保存起来。当浏览器再请求该网站时，浏览器把请求的网址连同该Cookie一同提交给服务器。服务器检查该Cookie，以此来辨认用户状态。服务器还可以根据需要修改Cookie的内容。

每次HTTP请求的时候，客户端都会发送相应的Cookie信息到服务端。

会话cookie一般不存储在硬盘而是保存在内存里，当然这个行为并不是规范规定的。若设置了过期时间，浏览器就会把cookie保存到硬盘上，

Session是另一种记录客户状态的机制，不同的是Cookie保存在客户端浏览器中，而Session保存在服务器上。客户端浏览器访问服务器的时候，服务器把客户端信息以某种形式记录在服务器上。这就是Session。客户端浏览器再次访问时只需要从该Session中查找该客户的状态就可以了。

Session机制决定了当前客户只会获取到自己的Session，而不会获取到别人的Session。各客户的Session也彼此独立，互不可见。Session的使用比Cookie方便，但是过多的Session存储在服务器内存中，会对服务器造成压力。

session的运行一般还需要依赖cookie，服务器在创建时会发送给客户端session id的值，之后每次客户端发送请求，服务器都会读取请求头中的cookie信息来获取session id以辨识用户的身份

URL地址重写是对客户端不支持Cookie的解决方案。URL地址重写的原理是将该用户Session的id信息重写到URL地址中。服务器能够解析重写后的URL获取Session的id。这样即使客户端不支持Cookie，也可以使用Session来记录用户状态。

session中保存的是对象，cookie中保存的是字符串

**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. 设置cookie时间可以使cookie过期。但是使用session-destory（），我们将会销毁会话。  
   4、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。  
   5、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。(Session对象没有对存储的数据量的限制，其中可以保存更为复杂的数据类型)

**注意:**

      session很容易失效,用户体验很差;

      虽然cookie不安全,但是可以加密 ;

      cookie也分为永久和暂时存在的;

      浏览器 有禁止cookie功能 ,但一般用户都不会设置;

     一定要设置失效时间,要不然浏览器关闭就消失了;

两者最大的区别在于生存周期，一个是IE启动到IE关闭.(浏览器页面一关 ,session就消失了)，一个是预先设置的生存周期，或永久的保存于本地的文件。(cookie)

Web Storage的概念和cookie相似，区别是它是为了更大容量存储设计的，cookie的大小是受限的，并且每次请求一个新的页面的时候cookie都会被发送过去，这样无形中浪费了带宽，另外cookie还需要指定作用域，不可跨域调用。

webstorage是本地存储，存储在客户端，包括localStorage和sessionStorage  
 localStorage生命周期是永久，这意味着除非用户显示在浏览器提供的UI上清除localStorage信息，否则这些信息将永远存在。存放数据大小为一般为5MB,而且它仅在客户端（即浏览器）中保存，不参与和服务器的通信  
 sessionStorage仅在当前会话下有效，关闭页面或浏览器后被清除。存放数据大小为一般为5MB,而且它仅在客户端（即浏览器）中保存，不参与和服务器的通信。源生接口可以接受，亦可再次封装来对Object和Array有更好的支持

**Web Storage带来的好处：**  
 1、减少网络流量：一旦数据保存在本地之后，就可以避免再向服务器请求数据，因此减少不必要的数据请求，减少数据在浏览器和服务器间不必要的来回传递   
 2、快速显示数据：性能好，从本地读数据比通过网络从服务器上获得数据快得多，本地数据可以及时获得，再加上网页本身也可以有缓存，因此整个页面和数据都在本地的话，可以立即显示   
 3、临时存储：很多时候数据只需要在用户浏览一组页面期间使用，关闭窗口后数据就可以丢弃了，这种情况使用sessionStorage非常方便

 作用域不同，sessionStorage不在不同的浏览器窗口中共享，即使是同一个页面；localstorage在所有同源窗口中都是共享的；cookie也是在所有同源窗口中都是共享的

 localStorage 接替了 Cookie 管理购物车的工作，同时也能胜任其他一些工作。比如HTML5游戏通常会产生一些本地数据，localStorage 也是非常适用的。如果遇到一些内容特别多的表单，为了优化用户体验，我们可能要把表单页面拆分成多个子页面，然后按步骤引导用户填写。这时候 sessionStorage 的作用就发挥出来了

**flex的属性 --9--**

flex-direction属性：决定主轴方向

flex-wrap属性：控制如何换行

 justify-content属性：在主轴上的对齐方式

align-items属性：在交叉轴上的对齐方式

order属性：定义项目的排列顺序，数值越小越靠前，默认为0

flex-grow属性：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。

flex-shrink属性：定义了项目的缩小比例，默认为1，即如果空间不足，该项目将缩小。

flex-basis属性：定义了在分配多余空间之前，项目占据的主轴空间（main size）。浏览器根据这个属性，计算主轴是否有多余空间。它的默认值为auto，即项目的本来大小。

flex属性：是flex-grow, flex-shrink 和 flex-basis的简写，默认值为0 1 auto。后两个属性可选。

当 flex 取值为 none，则计算值为 0 0 auto；

当 flex 取值为 auto，则计算值为 1 1 auto；

flex: 1相当于 1 1 0%

**从输入URL到浏览器完成页面渲染发生了什么 - 被问烂了，越详细越好。这里又有web优化问题 --9--**

每个步骤都展开讲

1、浏览器查找当前URL是否存在缓存，并比较缓存是否过期。

2、DNS解析URL对应的IP。

3、根据IP建立TCP连接（三次握手）。

4、HTTP发起请求。

5、服务器处理请求，浏览器接收HTTP响应。

6、渲染页面，构建DOM树。

7、关闭TCP连接（四次挥手）。

**DNS解析过程（简易）**

域名解析的过程实际是将域名还原为IP地址的过程。

1. 解析这个域名，查看本地hosts文件是否有这个网址映射关系，如果有就调用这个IP地址映射
2. 浏览器发送请求到本地域名服务器查询本地缓存，如果有就直接返回，该过程以递归方式查询。
3. 最后迭代查询，按根域服务器 ->顶级域,.com->第二层域，baidu.com ->子域，www.baidu.com的顺序找到IP地址。
4. 本地域名服务器返回结果给客户端，并保存到缓存

所有网址真正的解析过程为: . -> .com -> google.com. -> www.google.com.。最后面并不是多打了一个.，这个.对应的就是根域名服务器，默认情况下所有的网址的最后一位都是.

**页面渲染：**

1、解析HTML，构建DOM树

2、解析CSS，生成CSS规则树

3、合并DOM树和CSS规则，生成render树

4、布局render树（Layout/reflow），负责各元素尺寸、位置的计算

5、绘制render树（paint），绘制页面像素信息

6、浏览器会将各层的信息发送给GPU，GPU会将各层合成（composite），显示在屏幕上

**什么是闭包 --9--**

****闭包指的是：能够访问另一个函数作用域的变量的函数****

****创建闭包最常见的方式是在一个函数内创建另一个函数，通过另一个函数访问这个函数的局部变量。闭包跟JS的垃圾清理机制有关，在原函数执行完后它的执行环境理应被销毁的，但因为新函数中还保留着对原函数中的变量的引用，所以JS就不会对其进行销毁。新函数也就能继续通过引用来访问原函数中的变量。****

**HTTP1.0、1.1、2.0的区别 --8--**

**1.1新增**

1. 持久连接：默认开启keep-alive。在一个TCP连接上可以传送多个HTTP请求和响应，减少了建立和关闭连接的消耗
2. 节省带宽：在请求头引入了range头域，可以只请求资源的某个需要的部分，节省了多余的开销
3. 增加缓存处理：1.1增加了如Etag、If-Match等更多的缓存头来控制缓存策略
4. 增加Host字段：使得一个服务器能够用来创建多个Web站点。HTTP1.1的请求消息和响应消息都应支持Host头域，且请求消息中如果没有Host头域会报告一个错误
5. 错误通知的管理：新增了24个错误状态响应码

**2.0新增**

1. 二进制分帧：通过在应用层和传输层之间增加了一个二进制分帧层，突破了HTTP1.1的性能限制，改善传输性能
2. 多路复用：它能够在一个TCP上进行任意数量的HTTP请求，多个请求在一个连接上并行执行，某个请求任务耗时严重，不会影响到其他连接的正常执行
3. 头部压缩：通过算法将头部数据进行压缩，这样数据体积小了，在网络上传输就能更快了
4. 服务器推送：在之前的版本中，只能由客户端向服务器单向通信。2.0版本使得服务器能够主动向客户端推送信息，并且可以进行缓存，当客户端需要的资源已缓存时就可以直接从本地加载这些资源而不用再发送请求

**进程线程的概念 --8--**

**进程：**一个在内存中运行的应用程序。每个进程都有自己独立的一块内存空间，一个进程可以有多个线程，比如在Windows系统中，一个运行的xx.exe就是一个进程。

**线程：**进程中的一个执行任务（控制单元），负责当前进程中程序的执行。一个进程至少有一个线程，一个进程可以运行多个线程，多个线程可共享数据。

与进程不同的是同类的多个线程共享进程的堆和方法区资源，但每个线程有自己的程序计数器、虚拟机栈和本地方法栈，所以系统在产生一个线程，或是在各个线程之间作切换工作时，负担要比进程小得多，也正因为如此，线程也被称为轻量级进程。

**区别：**

**根本区别：**进程是操作系统资源分配的基本单位，而线程是处理器任务调度和执行的基本单位

**资源开销：**每个进程都有独立的代码和数据空间（程序上下文），程序之间的切换会有较大的开销；线程可以看做轻量级的进程，同一类线程共享代码和数据空间，每个线程都有自己独立的运行栈和程序计数器（PC），线程之间切换的开销小。

**包含关系：**如果一个进程内有多个线程，则执行过程不是一条线的，而是多条线（线程）共同完成的；线程是进程的一部分，所以线程也被称为轻权进程或者轻量级进程。

**内存分配：**同一进程的线程共享本进程的地址空间和资源，而进程之间的地址空间和资源是相互独立的

**影响关系：**一个进程崩溃后，在保护模式下不会对其他进程产生影响，但是一个线程崩溃整个进程都死掉。所以多进程要比多线程健壮。

**执行过程：**每个独立的进程有程序运行的入口、顺序执行序列和程序出口。但是线程不能独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制，两者均可并发执行

**移动端适配 - 这里涉及到的东西可就多了：媒体查询，viewport，em/rem，vw/vh --8--**

1. **百分比方案（流式布局）**

使用 百分比% 定义 宽度，高度 用px固定，根据可视区域实时尺寸进行调整，尽可能适应各种分辨率，通常使用max-width/min-width控制尺寸范围过大或者过小。

优势：原理简单，不存在兼容性问题

缺点：如果屏幕尺度跨度太大，相对设计稿过大或者过小的屏幕不能正常显示。设置盒模型的不同属性时，其百分比设置的参考元素不唯一，容易使布局问题变得复杂

1. **rem方案（弹性布局）**

rem是相对长度单位，rem方案中的样式设计为相对于根元素font-size计算值的倍数。根据 屏幕宽度 设置html标签的font-size，在布局时使用 rem 单位布局，达到自适应的目的，是 弹性布局 的一种实现方式。

优势：兼容性较好，自适应效果比百分比方案要好

缺点：不是纯css移动适配方案，需要引入js脚本 在头部内嵌一段 js脚本 监听分辨率的变化来动态改变根元素的字体大小

1. **vh/vw方案**

****视口****是浏览器中用于呈现网页的区域，移动端的视口通常指的是****布局视口****

· **vw** : **1vw** 等于 **视口宽度** 的 1%

· **vh** : **1vh** 等于 **视口高度** 的 1%

· **vmin** : 选取 **vw** 和 **vh** 中 **最小** 的那个

· **vmax** : 选取 **vw** 和 **vh** 中 **最大** 的那个

使用 css 预处理器把设计稿尺寸转换为 vw 单位，包括 文本，布局高宽，间距 等，使得这些元素能够随视口大小自适应调整。  
 优势：纯 css 移动端适配方案，不存在脚本依赖问题，根据视口尺寸的百分比 来定义 元素宽度，视觉效果更好

缺点：存在一些兼容性问题，Android4.4以下不支持

**4、基于媒体查询的响应式设计**

响应式设计 使得一个网站同时适配 多种设备 和 多个屏幕，让网站的布局和功能随用户的使用环境（屏幕大小、输出方式、设备/浏览器能力）而变化，使其视觉合理，交互方式符合习惯。

主要实现是通过 媒体查询，通过给不同分辨率的设备编写不同的样式实现响应式布局，用于解决不同设备不同分辨率之间兼容问题，一般是指PC、平板、手机设备之间较大的分辨率差异。

优势：能够使网页在不同设备、不同分辨率屏幕上呈现合理布局，不仅仅是样式伸缩变换  
 缺点：要匹配足够多的设备与屏幕，一个web页面需要多个设计方案，工作量比较大

对于上述的各种移动端web页面自适应方案来说，都存在着一些优势和不足。对于国内的一些互联网站，通过查看网页源代码发现，它可能不是某一种方案的单独使用，而是几种方案的结合。

物理像素又被称为设备像素，他是显示设备中一个最微小的物理部件。每个像素可以根据操作系统设置自己的颜色和亮度。所谓的一倍屏、二倍屏(Retina)、三倍屏，指的是设备以多少物理像素来显示一个CSS像素

设备像素比简称为dpr，其定义了物理像素和设备独立像素的对应关系。它的值可以按下面的公式计算得到：

设备像素比 ＝ 物理像素 / 设备独立像素

对于安卓，所有设备缩放设为1,对于IOS，根据dpr不同，设置其缩放为dpr倒数。设置页面缩放可以使得1个CSS像素(1px)由1个设备像素来显示，从而提高显示精度；因此，设置1/dpr的缩放视口，可以画出1px的边框。

**BFC（块级格式上下文）的概念，触发条件，渲染规则，应用场景 --7--**

BFC是一个独立的布局环境，其中的元素布局是不受外界的影响，并且在一个BFC中，块盒与行盒（行盒由一行中所有的内联元素所组成）都会垂直的沿着其父元素的边框排列。

通俗一点来讲，可以把 BFC 理解为一个封闭的大箱子，箱子内部的元素无论如何翻江倒海，都不会影响到外部。

在BFC中，每一个盒子的左外边缘（margin-left）会触碰到容器的左边缘(border-left)（对于从右到左的格式来说，则触碰到右边缘）

只要元素满足下面任一条件即可触发 BFC 特性：

·body 根元素

·浮动元素：float 除 none 以外的值

·绝对定位元素：position (absolute、fixed)

·display 为 inline-block、table-cells、flex

·overflow 除了 visible 以外的值 (hidden、auto、scroll)

**BFC的特性**

      1.内部的Box会在垂直方向上一个接一个的放置。  
      2.垂直方向上的距离由margin决定  
      3.bfc的区域不会与float的元素区域重叠。  
      4.计算bfc的高度时，浮动元素也参与计算  
      5.bfc就是页面上的一个独立容器，容器里面的子元素不会影响外面元素。

**BFC的应用**

1. 同一个 BFC 下外边距会发生折叠，如果想要避免外边距的重叠，可以将其放在不同的 BFC 容器中。（防止margin塌陷）
2. BFC 可以包含浮动的元素（让父元素触发BFC，清除浮动）

3、BFC 可以阻止元素被浮动元素覆盖

**引用script标签时涉及到的defer async --7--**

**async和defer的相同点：**

都是为了异步加载js，如果不加这两个属性，把script放在body结尾的话，js会等html先解析，才会解析js脚本，但是加了这两个属性，解析html的同时也会解析js脚本。虽然async和defer都是对js脚本异步加载，但是他俩还是有些许不同。

**async和defer的不同点：**

  主要的区别是执行的时机不同，defer会等整个页面都解析完毕之后再运行，延迟到浏览器遇到</html>标签再执行。

async如果解析完了就会执行，不能保证脚本的执行顺序，要看这个js的解析速度了，所以如果和别的脚本有依赖性或者需要操作dom最好不要用async。async一般用于百度统计、谷歌统计那些关联性不强的脚本。

1.defer和async在网络加载过程是一致的，都是异步执行的；

　　2.两者的区别在于脚本加载完成之后何时执行，可以看出defer更符合大多数场景对应用脚本加载和执行的要求；

　　3.如果存在多个有defer属性的脚本，那么它们是按照加载顺序执行脚本的；而对于async，它的加载和执行是紧紧挨着的，无论声明顺序如何，只要加载完成就立刻执行，它对于应用脚本用处不大，因为它完全不考虑依赖。

总结，我们应该尽量使用传统的方法来使用javascript，不要引入defer或者async来改变其执行的时机，防止非预期行为的产生。

**VUE组件通信的方法 --7--**

**1、props和$emit**

父组件向子组件传递数据是通过prop传递的，子组件传递数据给父组件是通过$emit触发事件来做到的

**2、$parent和$children**

      在组件内部可以直接通过子组件$parent对父组件进行操作，父组件通过$children对子组件进行操作.

**3、v-model**

父组件通过v-model传递值给子组件时，会自动传递一个value的prop属性，在子组件中通过this.$emit(‘input',val)自动修改v-model绑定的值

**4、中央事件总线（bus）**

通过一个空的Vue实例作为中央事件总线（事件中心），用它来$emit触发事件和$on监听事件,巧妙而轻量地实现了任何组件间的通信，包括父子、兄弟、跨级。当我们的项目比较大时，可以选择更好的状态管理解决方案vuex。

**5、vuex**

如果业务逻辑复杂，很多组件之间需要同时处理一些公共的数据，这个时候才有上面这一些方法可能不利于项目的维护，vuex的做法就是将这一些公共的数据抽离出来，然后其他组件就可以对这个公共数据进行读写操作，这样达到了解耦的目的。

**6、$attrs和$listeners**

多级组件嵌套需要传递数据时，通常使用的方法是通过vuex。但如果仅仅是传递数据，而不做中间处理，使用 vuex 处理，未免有点大材小用。为此Vue2.4 版本提供了另一种方法----$attrs/$listeners（父组件A下面有子组件B，组件B下面有组件C,这时组件A想传递数据给组件C）

**7、provide和inject**

在 Vue.js 的 2.2.0+ 版本中添加加了 provide 和 inject 选项。他们成对出现，用于父级组件向下传递数据。

父组件中通过provider来提供变量，然后在子组件中通过inject来注入变量。不论子组件有多深，只要调用了inject那么就可以注入provider中的数据。而不是局限于只能从当前父组件的prop属性来获取数据，只要在父组件的生命周期内，子组件都可以调用。

**常见使用场景可以分为三类：**

·父子通信：

父向子传递数据是通过 props，子向父是通过 events（$emit）；通过父链 / 子链也可以通信（$parent/$children）；ref 也可以访问组件实例；provide / inject API；$attrs/$listeners

·兄弟通信：

Bus；Vuex

·跨级通信：

Bus；Vuex；provide / inject API、$attrs/$listeners

**介绍一下Https，为什么要结合对称加密和非对称加密 --7--**

简单的说HTTPS就是HTTP 加上了SSL（安全套接层）或TSL（安全传输层协议），主要是为了解决http通信中的安全性问题

在使用http协议时，HTTP直接与TCP通信，当使用SSL时，则需要先与SSL通信，通过SSL或TLS提供加密处理数据、验证对方身份以及数据完整性保护。然后再由SSL和TCP通信。

在这里引用图解HTTP的一句话：

HTTP + 加密 + 认证 + 完整性保护 = HTTPS

**HTTPS有如下特点：**

1、内容加密：采用混合加密技术，中间者无法直接查看明文内容

2、验证身份：通过证书认证客户端访问的是自己的服务器

3、保护数据完整性：防止传输的内容被中间人冒充或者篡改

对称加密：就是加密和解密都是用同一个秘钥，这种方式优点就是速度快，缺点就是在管理和分配秘钥的时候不安全。

非对称加密算法，非对称加密有一个秘钥对，叫做公钥和私钥，私钥自己持有，公钥可以公开的发送给使用的人。使用公钥进行加密的信息，只有和其配对的私钥可以解开。非对称的加密算法的优点是安全，因为他不需要把私钥暴露出去。

在正式的使用场景中一般都是对称加密和非对称加密结合使用，使用非对称加密完成秘钥的传递，然后使用对称秘钥进行数据加密和解密（在两个节点间通过便捷的公开密码加密技术建立起安全通信，然后再用安全的通信产生并发送临时的随机对称密钥，通过更快的对称加密技术对剩余的数据进行加密。）

**安全问题，XSS、CSRF概念，出现的场景，解决的办法 --7--**

XSS名为跨站脚本攻击，黑客将恶意脚本代码植入到页面中从而实现盗取用户信息等操作。其特点是不对服务器端造成任何伤害，而是通过一些正常的站内交互途径，例如发布评论，提交含有 JavaScript 的内容文本。这时服务器端如果没有过滤或转义掉这些脚本，作为内容发布到了页面上，其他用户访问这个页面的时候就会运行这些脚本。

**XSS攻击可以分为3类：反射型（非持久型）、存储型（持久型）、基于DOM。**

·反射型 XSS 只是简单地把用户输入的数据 “反射” 给浏览器，这种攻击方式往往需要攻击者诱使用户点击一个恶意链接，或者提交一个表单，或者进入一个恶意网站时，注入脚本进入被攻击者的网站。

·存储型 XSS 会把用户输入的数据 “存储” 在服务器端，当浏览器请求数据时，脚本从服务器上传回并执行。这种 XSS 攻击具有很强的稳定性。

·基于 DOM 的 XSS 攻击是指通过恶意脚本修改页面的 DOM 结构，是纯粹发生在客户端的攻击。

**防范：**

1. 设置HttpOnly 防止劫取 Cookie：浏览器将禁止页面的Javascript 访问带有 HttpOnly 属性的Cookie

2、输入检查：不要相信用户的任何输入。 对于用户的任何输入要进行检查、过滤和转义。

3、输出检查：用户的输入会存在问题，服务端的输出也会存在问题。一般来说，除富文本的输出外，在变量输出到 HTML 页面时，可以使用编码或转义的方式来防御 XSS 攻击。

CSRF名为跨站请求伪造，是劫持受信任用户向服务器发送非预期请求的攻击方式。通常情况下，CSRF 攻击是攻击者借助受害者的 Cookie 骗取服务器的信任，可以在受害者毫不知情的情况下以受害者名义伪造请求发送给受攻击服务器，从而在并未授权的情况下执行在权限保护之下的操作。

简单来说，CSRF就是利用用户的登录态发起恶意请求。

**防范：**

1、验证码：CSRF攻击往往是在用户不知情的情况下发起了网络请求。而验证码会保证用户必须与应用进行交互，才能完成请求。

2、Referer Check：在HTTP头中有一个字段叫做Referer,它记录了该HTTP请求的来源地址。通过Referer Check,可以检查是否来自合法的"源".

3、添加 token 验证(token==令牌)：可以在 HTTP 请求中以参数的形式加入一个随机产生的 token，并在服务器端建立一个拦截器来验证这个 token，如果请求中没有 token 或者 token 内容不正确，则认为可能是 CSRF 攻击而拒绝该请求。

**CSRF和XSS的区别：**

1、CSRF需要登陆后操作，XSS不需要

2、CSRF是请求页面api来实现非法操作，XSS是向当前页面植入js脚本来修改页面内容。

**new 实例化一个对象 / new一个对象时发生了什么？四个步骤 --7--**

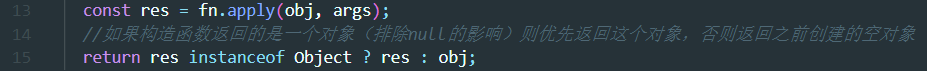
1、创建空对象：  
　　var obj = {};

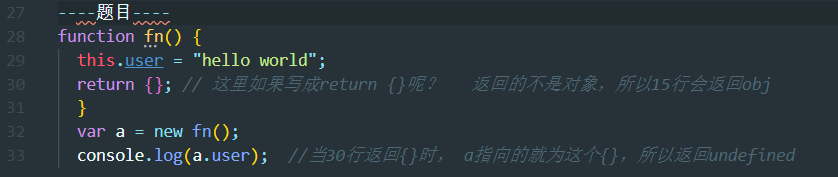
2、设置新对象的constructor属性为构造函数的名称，设置新对象的\_\_proto\_\_属性指向构造函数的prototype对象；  
　　obj.\_\_proto\_\_ = ClassA.prototype;

3、使用新对象调用函数，函数中的this被指向新实例对象：  
　　ClassA.call(obj);　　//{}.构造函数();

4、将初始化完毕的新对象地址，保存到等号左边的变量中

注意：若构造函数中返回this或返回值是基本类型（number、string、boolean、null、undefined）的值，则返回新实例对象；若返回值是引用类型的值，则实际返回值为这个引用类型。





**TCP3次握手和4次挥手 --6--**

****三次握手：****

1. ****客户端向服务器发送连接请求报文，同时发送SYN连接报文****
2. ****服务器接到请求后会发送给客户端一个SYN连接报文和一个ACK应答报文，ACK的序列号为客户端发送的SYN的序列号+1****
3. ****客户端收到后服务器发送的两个报文后会再发送一个ACK应答报文给服务器表示已收到，ACK的序列号为服务器发送的SYN的序列号+1****

****四次挥手：****

1. ****客户端发送终止连接的请求，同时发送FIN报文，客户端进入终止等待1状态****
2. ****服务器收到连接释放报文后，向发出ACK确认报文，服务器进入关闭等待状态。客户端接收到确认请求后进入终止等待2状态。此时服务器可能还有一部分数据还需要继续发送给客户端****
3. ****等待服务器将最后的数据发送完后，就会向客户端发送Fin连接释放报文。服务器此时进入最后确认状态****
4. ****客户端收到服务器的连接释放请求后，会向服务器发出ACK确认报文，同时进入时间等待状态，经过2\*MSL(最长报文段寿命)后没有问题则进入关闭状态，而服务器只要收到了客户端的确认报文就立即进入关闭状态。****

【问题1】为什么不能用两次握手进行连接？

答：3次握手完成两个重要的功能，既要双方做好发送数据的准备工作(双方都知道彼此已准备好)，也要允许双方就初始序列号进行协商，这个序列号在握手过程中被发送和确认。

       现在把三次握手改成仅需要两次握手，死锁是可能发生的。作为例子，考虑计算机S和C之间的通信，假定C给S发送一个连接请求分组，S收到了这个分组，并发 送了确认应答分组。按照两次握手的协定，S认为连接已经成功地建立了，可以开始发送数据分组。可是，C在S的应答分组在传输中被丢失的情况下，将不知道S 是否已准备好，不知道S建立什么样的序列号，C甚至怀疑S是否收到自己的连接请求分组。在这种情况下，C认为连接还未建立成功，将忽略S发来的任何数据分 组，只等待连接确认应答分组。而S在发出的分组超时后，重复发送同样的分组。这样就形成了死锁。

【问题2】如果已经建立了连接，但是客户端突然出现故障了怎么办？

TCP还设有一个保活计时器，显然，客户端如果出现故障，服务器不能一直等下去，白白浪费资源。服务器每收到一次客户端的请求后都会重新复位这个计时器，时间通常是设置为2小时，若两小时还没有收到客户端的任何数据，服务器就会发送一个探测报文段，以后每隔75秒钟发送一次。若一连发送10个探测报文仍然没反应，服务器就认为客户端出了故障，接着就关闭连接。

****【问题3】为什么连接的时候是三次握手，关闭的时候却是四次握手？****  
答：因为当Server端收到Client端的SYN连接请求报文后，可以直接发送SYN+ACK报文。其中ACK报文是用来应答的，SYN报文是用来同步的。但是关闭连接时，当Server端收到FIN报文时，很可能并不会立即关闭SOCKET，所以只能先回复一个ACK报文，告诉Client端，"你发的FIN报文我收到了"。只有等到我Server端所有的报文都发送完了，我才能发送FIN报文，因此不能一起发送。故需要四步握手。

****【问题4】为什么TIME\_WAIT状态需要经过2MSL(最大报文段生存时间)才能返回到CLOSE状态？****

答：虽然按道理，四个报文都发送完毕，我们可以直接进入CLOSE状态了，但是我们必须假象网络是不可靠的，有可以最后一个ACK丢失。所以TIME\_WAIT状态就是用来重发可能丢失的ACK报文。

**css元素样式，选择器优先级 --6--**

行内样式 > id选择器 >  class/伪类/属性选择器 > 标签选择器 > 通配符选择器



**CSS的position --6--**

1. static：静态定位，为position的默认值，元素以正常文档流排列
2. relative：相对定位，处于文档流中，相对于自己原本位置定位，因为元素原本的位置会被保留，其他元素的位置不受影响
3. absolute：绝对定位，脱离文档流，相对于static定位以外的最近的祖先元素进行定位。如果都是static定位最终会相对于body进行定位
4. fixed：固定定位，脱离文档流，总是相对于浏览器窗口定位
5. inherit：继承父元素的position属性
6. sticky：粘性定位，是一个还处于试验性的属性。不脱离文档流。当元素在容器被滚动超过指定的偏移值时，元素就会被固定在指定的位置

**CSS三角形、扇形、圆形 --6--**

如果将盒容器的width和height设置为0，并为每条边设置一个较粗的width值和彼此不同的颜色，最终会得到四个被拼接在一起三角形，它们分别指向不同的颜色。



当我们对四个三角形的其中三个设置颜色为透明即transparent，即可得到一个三角形



大家应该知道css3中引入了圆角属性（border-radius），一旦设置这个值，边框即会出现圆角。同样，我们对正方形设置圆角，即可得到饼状图



同样我们对其中三个边框设置透明色即可得到扇形



**前端性能优化 --6--**

**网络：**

·合并JS、CSS等外链文件，减少HTTP请求数

·压缩文件，减少文件大小

·合理使用缓存

·使用CDN内容分发网络，减少延迟

·使用DNS预解析，DNS预解析就是根据浏览器定义的规则，提前解析之后可能会用到的域名，使解析结果缓存到系统缓存中，缩短DNS解析时间，来提高网站的访问速度

**CSS:**

·将图片等静态资源使用懒加载，用到时再静态加载

·使用精灵图（CSS Sprite）

·将更精确的选择符放在右边，因为CSS匹配过程是从右到左的，应尽量让第一次匹配到的数量最少

·不要设置图片的尺寸，制作使用符合大小的图片，利用样式缩放图片会增加CPU计算负担，渲染时间，还会占用更多的带宽

·CSS放Head标签中

·尽量所使用的字体图标或者SVG图标来代替传统png图，因为字体图标或者SVG是矢量图，代码编写出来的，方大不会变形，而且渲染速度快

·不使用CSS @import

**JS:**

·重点：减少DOM的操作，特别是容易影响重绘和重排

·把可能性更高的条件判断语句放到前面，减少进行条件判断的次数

·将部分不影响首页展示的JS文件延迟到页面加载后加载

·将所有script标签放到body标签底部，确保脚本执行前页面已经完成渲染。以免直接开始执行JS脚本影响页面的渲染和其他文件的下载

·将变量放到离自身作用域最近的作用域中，每多一层作用域，访问时读写的速度就更慢，对象的属性也如此，如果需要多次读取对象属性，最好将其存入局部变量，尤其是嵌套对象成员

·尽量将一个动画元素单独设置为一个图层（避免重绘或者回流的大小），注意：图层不要过多设置，否则不但效果没有达到反而更差了

事件节流、防抖

·使用事件委托，为每个元素添加事件处理程序内存占用多，新增的元素需要重新添加事件处理程序。注销时也麻烦

·改善循环性能的最佳方式是减少每次迭代的运算量和减少循环迭代次数

·代码末尾加分号

**HTTP请求的类型，常见方法有哪些 --6--**

1. GET 请求指定页面信息，并返回实体主体（获取资源）；  
2. POST 向指定资源提交数据并进行处理请求，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或已有资源的修改（传输实体主体）；  
3. PUT 向指定资源位置上传其最新内容(资源更新)。  
4. DELETE 请求服务器删除指定的内容；  
5. Options     两种场景: 1. 获取资源支持的http方法 2. 跨域预检请求，检测实际请求是否可被浏览器接受  
6. Head        获取http的头部信息，与GET请求头部一致，只是不包含消息体，常用来检测url的有效性  
7. Trace        回显服务收到的请求，用于测试或诊断。一般禁用，防止被攻击或盗取信息  
8. Connect    http1.1预留给支持管道化的服务器  
9. Patch        与PUT方法一样，用于资源更新。区别，1.Patch是对资源部分更新，PUT是整体更新 2. 资源不存在时，Patch会创建新资源，PUT不会

**get / post 的区别 --5--**

**1、url可见性：**

get，参数url可见；

post，url参数不可见

**2、数据传输上：**

get，通过拼接url进行传递参数；

post，通过body体传输参数

**3、缓存性：**

get请求是可以缓存的

post请求不可以缓存

**4、后退页面的反应**

get请求页面后退时，不产生影响

post请求页面后退时，会重新提交请求

**5、传输数据的大小**

get一般传输数据大小不超过2k-4k（根据浏览器不同，限制不一样，但相差不大）

post请求传输数据的大小根据php.ini 配置文件设定，也可以无限大。

**6、安全性**

这个也是最不好分析的，原则上post肯定要比get安全，毕竟传输参数时url不可见，但也挡不住部分人闲的没事在那抓包玩。安全性个人觉得是没多大区别的，防君子不防小人就是这个道理。对传递的参数进行加密，其实都一样。

**TCP / IP 七层模型 --5--**



**http和https的区别 --5--**

　　1、https协议需要到ca申请证书，一般免费证书较少，因而需要一定费用。

　　2、http是超文本传输协议，信息是明文传输，如果攻击者截取了Web浏览器和网站服务器之间的传输报文，就可以直接读懂其中的信息。https则是具有安全性的ssl加密传输协议。

　　3、http和https用的端口也不一样，前者是80，后者是443。

　　4、http的连接很简单，是无状态的；HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，比http协议安全。

**各种排序 稳定性，时间复杂度 --5--**

　排序算法大体可分为两种：

　　　　一种是**比较排序**，时间复杂度O(nlogn) ~ O(n^2)，主要有：**冒泡排序**，**选择排序**，**插入排序**，**归并排序**，**堆排序**，**快速排序**等。

　　　　另一种是**非比较排序**，时间复杂度可以达到O(n)，主要有：**计数排序**，**基数排序**，**桶排序**等。



**清除浮动 --5--**

1、**额外标签法**（在最后一个浮动标签后，新加一个标签，给其设置clear：both；）（不推荐）

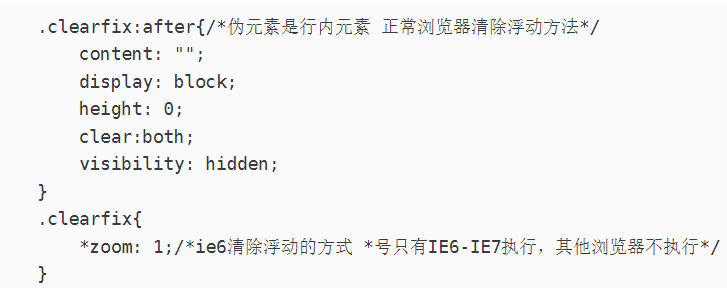
缺点：添加无意义标签，语义化差

2、**父级添加overflow属性**（父元素添加overflow:hidden）（不推荐）

通过触发BFC方式，实现清除浮动

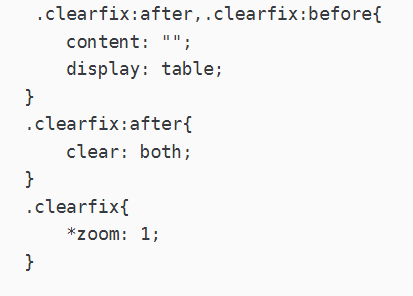
缺点：内容增多的时候容易造成不会自动换行导致内容被隐藏掉，无法显示要溢出的元素

3、**使用after伪元素清除浮动**（推荐使用）



缺点：ie6-7不支持伪元素：after，使用zoom:1触发hasLayout.

4、**使用before和after双伪元素清除浮动**（推荐使用）



缺点：用zoom:1触发hasLayout.

**diff算法的实现 --5--**

**http常用请求头和返回头有哪些，和缓存有关的有哪些 --5--**

****http请求中的常用头（请求头）的含义：****

Accept：告诉服务器，客户端支持的数据类型。

Accept-Charset：告诉服务器，客户端采用的编码。

Accept-Encoding：告诉服务器，客户机支持的数据压缩格式。

Accept-Language：告诉服务器，客户机的语言环境。

Host：客户机通过这个头告诉服务器，想访问的主机名。

If-Modified-Since:客户机通过这个头告诉服务器，资源的缓存时间。

Referer:客户机通过这个头告诉服务器，它是从哪个资源来访问服务器的。（一般用于防盗链）

User-Agent:客户机通过这个头告诉服务器，客户机的软件环境。

Cookie：客户机通过这个头告诉服务器，可以向服务器带数据。

Connection：客户机通过这个头告诉服务器，请求完后是关闭还是保持链接。

Date：客户机通过这个头告诉服务器，客户机当前请求时间。

****http请求中常用的响应头的含义：****

Location:这个头配合302状态码使用，告诉用户端找谁。

Server:服务器通过这个头，告诉浏览器服务器的类型

。  
Content-Encoding:服务器通过这个头，告诉浏览器数据采用的压缩格式。

Content-Length:服务器通过这个头，告诉浏览器回送数据的长度。

Content-Language：服务器通过这个头，告诉服务器的语言环境。

Content-Type:服务器通过这个头，回送数据的类型

Last-Modified:服务器通过这个头，告诉浏览器当前资源的缓存时间。

Refresh:服务器通过这个头，告诉浏览器隔多长时间刷新一次。

Content-Disposition:服务器通过这个头，告诉浏览器以下载的方式打开数据。

Transfer-Encoding:服务器通过这个头，告诉浏览器数据的传送格式。

ETag:与缓存相关的头。

Expires:服务器通过这个头，告诉浏览器把回送的数据缓存多长时间。-1或0不缓存。

Cache-Control和Pragma：服务器通过这个头，也可以控制浏览器不缓存数据。

Connection:服务器通过这个头，响应完是保持链接还是关闭链接。

Date:告诉客户机，返回响应的时间。

**em和rem --4--**

在css中单位长度用的最多的是px、em、rem，这三个的区别是：

　　·px是固定的像素，一旦设置了就无法因为适应页面大小而改变。

　　·em和rem相对于px更具有灵活性，他们是相对长度单位，意思是长度不是定死了的，更适用于响应式布局。

对于em和rem的区别一句话概括：em相对于父元素（错误），rem相对于根元素。

有一个比较普遍的误解，认为 em 单位是相对于父元素的字体大小。 事实上，根据W3标准 ，它们是相对于使用em单位的元素（自身）的字体大小。

父元素的字体大小可以影响 em 值，但这种情况的发生，纯粹是因为继承。

**回流和重绘 --4--**

**浏览器渲染过程如下：**

1、解析HTML，生成DOM树，解析CSS，生成CSSOM树

2、将DOM树和CSSOM树结合，生成渲染树(Render Tree)

3、Layout(回流):根据生成的渲染树，进行回流(Layout)，得到节点的几何信息（位置，大小）

4、Painting(重绘):根据渲染树以及回流得到的几何信息，得到节点的绝对像素

5、Display:将像素发送给GPU，展示在页面上。（这一步其实还有很多内容，比如会在GPU将多个合成层合并为同一个层，并展示在页面中。而css3硬件加速的原理则是新建合成层）

reflow:当render树中的一部分或者全部因为大小边距等问题发生改变而需要重建的过程叫做回流

repaint:当元素的一部分属性发生变化，如外观背景色不会引起布局变化而需要重新渲染的过程叫做重绘

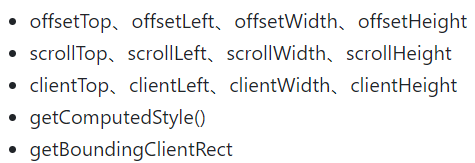
笼统来说当页面的布局和几何属性发生变化的时候就会引起回流。具体来说大概分别5大类：

1.首当其冲自然是dom树结构变化，比如你删除或者添加某个node.

2.元素几何属性变化，包括margin,padding,height,width,border等

3.页面渲染初始化

4.获取某些属性。虽然浏览器引擎可能会针对重排做了优化，比如Opera，它会等到有足够 数量的变化发生，或者等到一定的时间，或者等一个线程结束，再一起处理，这样就只发生一次重排。但除了render树的直接变化，当获取一些属性时，浏览器为取得 正确的值也会触发回流。这样就使得浏览器的优化失效了。这些属性包括：

等···

所以，在多次使用这些值时应进行缓存。

5.浏览器窗口发生变化-resize事件发生时

回流与重绘两者之间的联系在于: 触发回流一定会触发重绘, 而触发重绘却不一定会触发回流。

**如何减少回流、重绘**

1、直接改变className，如果动态改变样式，则使用cssText（考虑没有优化的浏览器）

2、让要操作的元素进行”离线处理”，处理完后一起更新

a) 使用DocumentFragment进行缓存操作,引发一次回流和重绘；  
 b) 使用display:none技术，只引发两次回流和重绘；  
 c) 使用cloneNode(true or false) 和 replaceChild 技术，引发一次回流和重绘；

3、不要经常访问会引起浏览器flush队列的属性（offsetLeft等），如果你确实要访问，利用缓存

4、让元素脱离文档流(absolute)，减少回流的Render Tree的规模

**防抖节流的使用，区别，手写 --4--**

在开发中，我们常常会去监听滚动事件或者用户输入框验证事件，如果事件处理没有频率限制，就会加重浏览器的负担，影响用户的体验感，

因此，我们可以采取防抖（debounce）和节流（throttle）来处理，减少调用事件的频率，达到较好的用户体验。

**防抖（debounce）：**

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，重新出发定时器。

**相当于就是多次执行，只执行一次。**

**节流（throttle）：**

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。

总结，看完了防抖和节流的分别介绍，我们来看看他们的区别：

函数防抖和函数节流都是防止某一时间频繁出发，但是这两兄弟之间的远离却不一样

**防抖**是将多次执行变为只执行一次，**节流**是将多次执行变为每隔一段时间执行。

### **结合应用场景**

　　防抖(debounce)

　　　　search搜索联想，用户在不断输入值时，用防抖来节约请求资源。

　　　　window触发resize的时候，不断的调整浏览器窗口大小会不断的触发这个事件，用防抖来让其只触发一次

　　节流(throttle)

　　　　鼠标不断点击触发，mousedown(单位时间内只触发一次)

　　　　监听滚动事件，比如是否滑到底部自动加载更多，用throttle来判断

**继承的方法 --4--**

1. **原型链模式**

function Person() {

Person.prototype.name = ‘jack’;

Person.prototype.age = 22;

Person.prototype.sayName = function() {

console.log(this.name);

}

}

使用这种原型模式定义属性和方法是由所有实例共享的，在不同实例访问到的都是相同的属性和方法

**2.构造函数模式**

依照惯例，构造函数名称的首字母都是要大写的，非构造函数则以小写字母开头

function Person(name, age) {

this.name = name;

this.age = age;

this.sayName = function () {

console.log(this.name);

}

}

let person1 = new Person(‘jack’, 29);

构造函数的主要问题在于：其定义的方法会在每个实例上都创建一遍

**3.组合继承（常用）**

基本的思路是使用原型链继承原型上的属性和方法，而通过盗用构造函数继承实例属性，这样既可以把方法定义在原型上以重现重用，又可以让每个实例都有自己的属性

function SuperType(name) {

this.name = name;

this.colors = [“red”, “blue”];

}

SuperType.prototype.sayName = function() {

console.log(this.name);

}

function SubType(name, age) {

//继承属性

SuperType.call(this, name);

this.age = age;

}

//继承方法

SubType.prototype = new SuperType();

组合继承弥补了原型链和盗用构造函数的不足，是JS中使用最多的继承模式。而且组合继承也保留instanceof操作符和isPrototypeOf()方法识别合成对象的能力

**4.原型式继承**

出发点是即使不自定义类型也可以通过原型实现对象之间的信息共享

function object(o) {

function F() {}

1. prototype = o;

return new F();

}

本质上，object()是对传入的对象执行了一次浅复制

适用情况：你有一个对象，想在它的基础上再创建新对象。你需要把这个对象先传给object()，然后再对返回的对象进行适当修改

原型式继承非常适合不需要单独创建构造函数但仍然需要在对象间共享信息的场合。但要记住，属性中包含的引用值始终会在相关对象间共享，跟使用原型模式是一样的

**5.寄生式继承**

背后的思路类似于寄生构造函数和工厂模式：创建一个实现继承的函数，以某种方式增强对象，然后返回这个对象

function createAnother(original) {

let clone = object(original); //通过调用函数创建一个新对象

clone.sayHi = function() { //以某种方式增强这个对象

console.log(“hi”);

};

return clone;

}

寄生式继承同样适合主要关注对象，而不在乎类型和构造函数的场景。object()函数不是寄生式继承所必须的，任何返回新对象的函数都可以在这里使用

**6.寄生式组合继承（常用）**

组合继承也存在效率问题，最主要的效率问题是父类构造函数始终会被调用两次：一次是在创建子类原型时调用，另一次是在子类构造函数中调用

寄生式组合继承通过盗用构造函数继承属性，但是用混合式原型链继承方法。基本思路是不通过调用父类构造函数给子类原型赋值，而是取得父类原型的一个副本。说到底就是使用寄生式继承来继承父类原型，然后将返回的新对象赋值给子类原型

function inheritPrototype(subType, superType) {

let prototype = object(superType.prototype); //创建对象

prototype.constructor = subType; //增强对象

subType.prototype = prototype; //赋值对象

}

寄生式组合继承可以算是引用类型继承的最佳模式

**this的指向 --4--**

this并不指向函数自身

需要明确的是,this在任何情况下都不指向函数的词法作用域

this实际上是在函数被调用时发生的绑定，它指向什么完全取决于函数在哪里被调用

**this的四种绑定规则:**

**1.默认绑定：**无法应用其他规则时的默认规则

**2.隐式绑定：**考虑调用位置是否有上下文对象，或者说是否被某个对象拥有或者包含（引用）

在分析隐式绑定时，我们必须在一个对象内部包含一个指向函数的属性，并通过这个属性间接引用函数，从而把this间接（隐式）绑定到这个对象上

一个最常见的this绑定问题就是被隐式绑定的函数会丢失绑定对象，也就是说它会应用默认绑定，从而把this绑定到全局对象（严格模式下是undefined）

**3.显式绑定：**使用call()/apply()方法，它们的第一个参数是一个对象，在调用函数时将其绑定到this。因为可以直接指定this的绑定对象，因而称为显式绑定。

如果传入了一个原始值（字符串、布尔或数字）来当做this的绑定对象，这个原始值会被转换成它的对象形式（new String()···）,这通常被称为装箱

**4.new绑定：**使用new来调用foo()时，会构造一个新对象并把它绑定到foo()调用中的this上。

使用new来调用函数，或者说发生构造函数调用时，会自动执行下面操作：

1. 创建（或者说构造）一个全新的对象
2. 这个新对象会被执行[[Prototype]]连接
3. 这个新对象会绑定到函数调用的this
4. 如果函数没有返回其他对象 ，那么new表达式中的函数调用会自动返回这个新对象

**优先级：new绑定>显示绑定>隐式绑定>默认绑定**

归纳：判断this

1.函数是否在new中调用（new绑定）？如果是的话this绑定的是新创建的对象

var bar = new foo();

2.函数是否通过call、apply（显式绑定）或者硬绑定调用？如果是的话，this绑定的是指定的对象

var bar = foo.call(obj2);

3.函数是否在某个上下文对象中调用（隐式绑定）？如果是的话，this绑定的是那个上下文对象

var bar = obj1.foo();

4.如果都不是的话，使用默认绑定。this绑定到全局对象，在严格模式下绑定到undefined

var bar = foo()

**箭头函数中的this**

箭头函数没有自己的this, 它的this是继承而来; 默认指向在定义它时所处的对象(宿主对象),而不是执行时的对象, 定义它的时候,可能环境是window; 箭头函数可以方便地让我们在 setTimeout ,setInterval中方便的使用this

**meta标签 --4--**

<meta> 标签提供关于 HTML 文档的元数据。它不会显示在页面上，但是对于机器是可读的。可用于浏览器（如何显示内容或重新加载页面），搜索引擎（关键词），或其他 web 服务。

meta里的数据是供机器解读的，告诉机器该如何解析这个页面，还有一个用途是可以添加服务器发送到浏览器的http头部内容

meta的必需属性是content，当然并不是说meta标签里一定要有content，而是当有http-equiv或name属性的时候，一定要有content属性对其进行说明。

在W3school中，对于meta的可选属性说到了三个，分别是http-equiv、 name和scheme。考虑到scheme不是很常用，所以就只说下前两个属性

http-equiv属性是添加http头部内容，对一些自定义的，或者需要额外添加的http头部内容，需要发送到浏览器中，我们就可以是使用这个属性。

<meta http-equiv="参数" content="参数变量值">；

**其中http-equiv属性主要有以下几种参数：**

**A、Expires(期限)**

说明：可以用于设定网页的到期时间。一旦网页过期，必须到服务器上重新传输。

**B、Pragma(cache模式)**

说明：禁止浏览器从本地计算机的缓存中访问页面内容。

**C、Refresh(刷新)**

说明：自动刷新并指向新页面。

**D、Set-Cookie(cookie设定)**

说明：如果网页过期，那么存盘的cookie将被删除。

**E、Window-target(显示窗口的设定)**

说明：强制页面在当前窗口以独立页面显示。

**F、content-Type(显示字符集的设定)**

说明：设定页面使用的字符集。

**G、content-Language（显示语言的设定）**

name属性主要用于描述网页，与之对应的属性值为content，content中的内容主要是便于搜索引擎机器人查找信息和分类信息用的。

<meta name="参数" content="具体的参数值">。

**其中name属性主要有以下几种参数：**

**A、Keywords(关键字)**

说明：keywords用来告诉搜索引擎你网页的关键字是什么。

**B、description(网站内容描述)**

说明：description用来告诉搜索引擎你的网站主要内容。

**C、robots(机器人向导)**

说明：robots用来告诉搜索机器人哪些页面需要索引，哪些页面不需要索引。

**D、author(作者)**

说明：标注网页的作者

1. **generator**

说明网站的采用的什么软件制作。

1. **copyright**

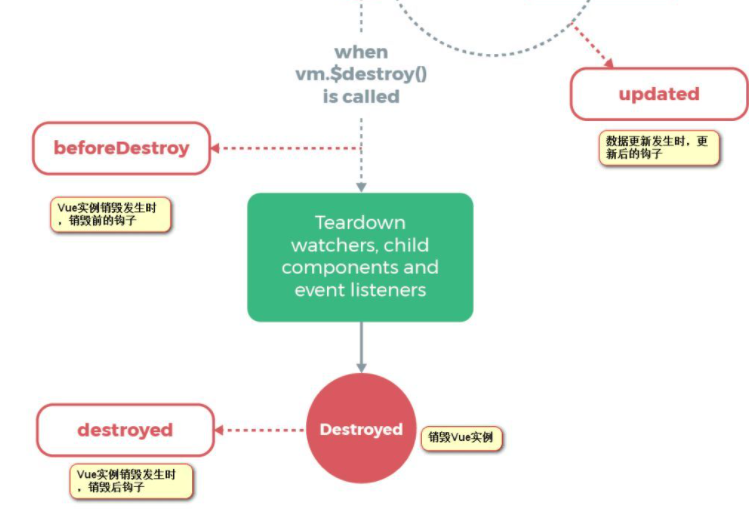
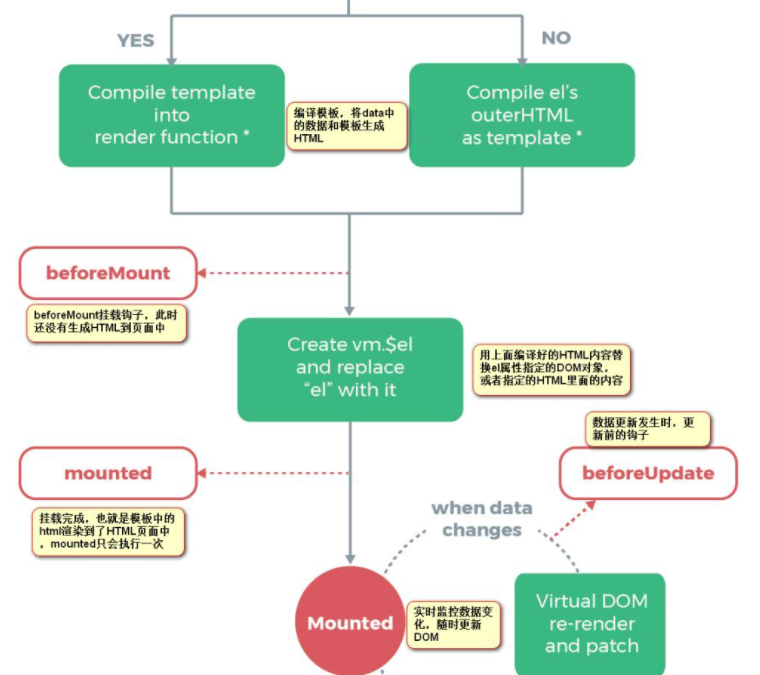
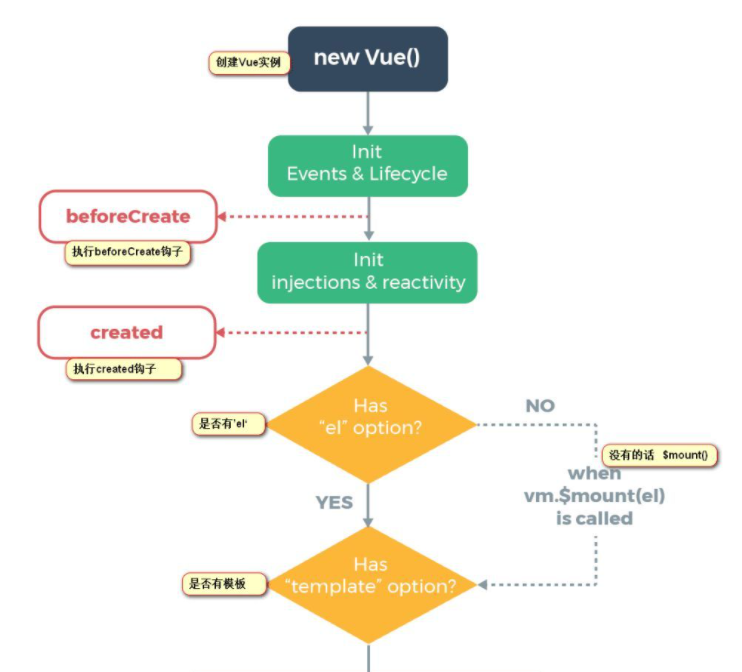
说明网站版权信息。

还有个设置字符集的charset

charset是声明文档使用的字符编码，解决乱码问题主要用的就是它，值得一提的是，这个charset一定要写第一行，不然就可能会产生乱码了。



**vue的生命周期 --4--**



1、beforeCreated（）

在实例初始化之后，数据观测(data observer) 和 event/watcher 事件配置之前被调用。

2、created（）

实例已经创建完成之后被调用。在这一步，实例已完成以下的配置：数据观测(data observer)，属性和方法的运算， watch/event 事件回调。然而，挂载阶段还没开始，$el 属性目前不可见。

3、beforeMount（）

在挂载开始之前被调用：相关的 render 函数首次被调用。

4、mounted（）

el 被新创建的 vm.$el 替换，并挂载到实例上去之后调用该钩子。如果 root 实例挂载了一个文档内元素，当 mounted 被调用时 vm.$el 也在文档内。

5、beforeUpdate（）

数据更新时调用，发生在虚拟 DOM 重新渲染和打补丁之前。 你可以在这个钩子中进一步地更改状态，这不会触发附加的重渲染过程。

6、updated（）

由于数据更改导致的虚拟 DOM 重新渲染和打补丁，在这之后会调用该钩子。

当这个钩子被调用时，组件 DOM 已经更新，所以你现在可以执行依赖于 DOM 的操作。然而在大多数情况下，你应该避免在此期间更改状态，因为这可能会导致更新无限循环。

7、beforeDestroy（）

实例销毁之前调用。在这一步，实例仍然完全可用。

8、destroyed（）

Vue 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。 该钩子在服务器端渲染期间不被调用。

**什么是虚拟DOM --3--**

当你用传统的源生api或jQuery去操作DOM时，浏览器会从构建DOM树开始从头到尾执行一遍流程。比如当你在一次操作时，需要更新10个DOM节点，理想状态是一次性构建完DOM树，再执行后续操作。但浏览器没这么智能，收到第一个更新DOM请求后，并不知道后续还有9次更新操作，因此会马上执行流程，最终执行10次流程。显然例如计算DOM节点的坐标值等都是白白浪费性能，可能这次计算完，紧接着的下一个DOM更新请求，这个节点的坐标值就变了，前面的一次计算是无用功。

虚拟DOM就是为了解决这个浏览器性能问题而被设计出来的。如果一次操作中有10次更新DOM的操作，虚拟DOM不会立即操作DOM，而是将这10次更新的diff内容保存在本地的一个js对象中，最终将这个js对象一次性attach到DOM树上，通知浏览器去执行绘制工作，这样可以避免大量的无谓的计算量。

**虚拟dom原理流程，简单概括有三点：**

1、用JavaScript模拟DOM树，并渲染这个DOM树

2、比较新老DOM树，得到比较的差异对象

3、把差异对象应用到渲染的DOM树。

虚拟DOM与真实DOM的区别？

1、虚拟DOM不会进行重排与重绘操作；

2、虚拟DOM进行频繁修改，然后一次性比较并修改真实DOM中需要修改的部分，最后进行重排和重绘，减少过多DOM节点重排和重绘损耗。

3、虚拟DOM有效降低大面积（真实DOM节点）的重排和重绘，因为最终与真实DOM比较差异，可以局部渲染。

**块元素、行内元素 --3--**

**块状元素**代表性的就是div，其他如p、nav、aside、header、footer、section、article、ul-li、address等等，都可以用div来实现。不过为了可以方便程序员解读代码，一般都会使用特定的语义化标签，使得代码可读性强，且便于查错。

　　块状元素特征：(1)能够识别宽高

　　　　　　　　　(2)margin和padding的上下左右均对其有效

　　　　　　　　　(3)可以自动换行

　　　　　　　　　(4)多个块状元素标签写在一起，默认排列方式为从上至下

**行内元素**最常使用的就是span，其他的只在特定功能下使用，修饰字体<b>和<i>标签，还有<sub>和<sup>这两个标签可以直接做出平方的效果，而不需要类似移动属性的帮助，很实用。

　　行内元素特征：(1)设置宽高无效

　　　　　　　　　(2)对margin仅设置左右方向有效，上下无效；padding设置上下左右都 有效，即会撑大空间

　　　　　　　　　(3)不会自动进行换行

**行内块状元素**综合了行内元素和块状元素的特性，但是各有取舍。因此行内块状元素在日常的使用中，由于其特性，使用的次数也比较多。

　　行内块状元素特征：(1)不自动换行

　　　　　　　　　　　(2)能够识别宽高

　　　　　　　　　　　(3)默认排列方式为从左到右

**总结：**

不管块级元素还是行内元素，区别主要是三个方面:一是排列方式，二是宽高边距设置，三是默认宽度。

·块级元素会独占一行，而内联元素和内联块元素则会在一行内显示。

·块级元素和内联块元素可以设置 width、height 属性，而内联元素设置无效。

·块级元素的 width 默认为 100%，而内联元素则是根据其自身的内容或子元素来决定其宽度。

而行内块级元素又同时拥有块级元素和行内元素的特点。

**行内替换元素和非替换元素**

替换元素就是浏览器根据标签的元素、属性来决定如何显示，比如img、video、input、textarea  
 非替换元素就是div、span、p、h1-h6、这些

行内替换元素是可以设置宽高的，行内非替换元素无法设置宽高。

行内替换元素width、 height、 margin的四个方向、 padding的四个方向都正常显示，遵循标准的css盒模型。 例如：img

**为什么浏览器的请求有两次，一次options，第二次才是真正请求 --3--**

在有很多情况下,当我们在js里面调用一次ajax请求时,在浏览器那边却会查询到两次请求,第一次的Request Method参数是OPTIONS,第二次才是我们真正的请求,比如get或是post请求方式

查阅相关的资料之后发现,这是浏览器对复杂跨域请求的一种处理方式,在真正发送请求之前,会先进行一次预请求,就是我们刚刚说到的参数为OPTIONS的第一次请求,他的作用是用于试探性的服务器响应是否正确,即是否能接受真正的请求,如果在options请求之后获取到的响应是拒绝性质的,例如500等http状态,那么它就会停止第二次的真正请求的访问

**大致说明一下,有三种方式会导致这种现象:**

1.请求的方法不是GET/HEAD/POST

2.POST请求的Content-Type并非application/x-www-form-urlencoded, multipart/form-data, 或text/plain

3.请求设置了自定义的header字段

比如我的我的Content-Type设置为“application/json;charset=utf-8”并且自定义了header选项导致了这种情况。