盒模型：

1. 从内到外分别是：content、padding、border、margin.
2. 两种模型：标准模型：width只包含content；IE模型：width包含content、padding、border；可以用box-sizing设置content-box（标准模型），border-box（IE模型）

链接：<https://www.cnblogs.com/chengzp/p/cssbox.html>

BFC (块格式化上下文):

渲染规则：默认根元素（html 元素）会创建一个 BFC，其块级盒子元素将会按照如下规则进行渲染：

● 块级盒会在垂直方向，一个接一个地放置，每个盒子水平占满整个容器空间

● 块级盒的垂直方向距离由上下 margin 决定，**同属于一个 BFC 中的两个或以上块级盒的相接的 margin 会发生重叠**

● BFC 就是页面上的一个隔离的独立容器，容器里面的子元素不会影响到外面的元素。反之也如此

● 计算 BFC 的高度时，浮动元素也参与计算

# 如何解决外边距叠加的问题？

根据 BFC 的定义，两个元素同属于一个 BFC 中的两个或以上块级盒的相接的 margin 会发生重叠。

● 1.如果不想重叠，将某一个设置成inline-block，让某一个不是BFC即可（不好）

● 2.把这两个 <div> 分别放进两个建立了 BFC 的父级容器里（用上面的BFC设置都行）

● 3.把子元素也设置成BFC，用overflow:hidden（没试过，试一下）

链接：<https://www.zhihu.com/question/19823139>

链接：<https://developer.mozilla.org/zh-CN/docs/Web/Guide/CSS/Block_formatting_context>

清除浮动：

1. clear:both;

<div class=“container clearfix”>

<div class=“box” style=“float:left;”></div>

</div>

.clearfix::after{

content:“”;

display:table;

clear:both;

}

1. overflow:hidden;(创建BFC)

<div class=“container”>

<div class=“box” style=“float:left;”></div>

</div>

.container{

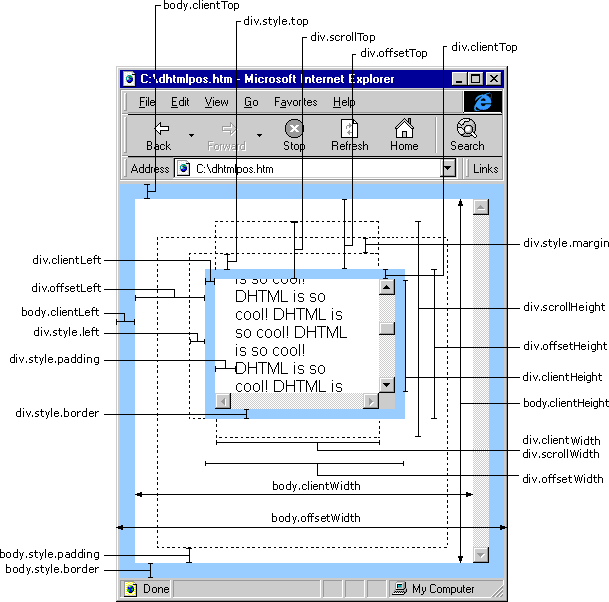
overflow:hidden;

}

圣杯布局&双飞翼布局：

链接：<https://blog.csdn.net/wangchengiii/article/details/77926868>

[clientHeight , scrollHeight , offsetHeight之间的区别](http://www.cnblogs.com/nanshanlaoyao/p/5964730.html)&Element.getBoundingClientRect()：



以下属性或方法会刷新渲染队列

* offsetTop、offsetLeft、offsetWidth、offsetHeight
* clientTop、clientLeft、clientWidth、clientHeight
* scrollTop、scrollLeft、scrollWidth、scrollHeight
* getComputedStyle()（IE中currentStyle）

我们在修改样式过程中，要尽量避免使用上面的属性

● InnerHeight/innerWidth与上面3者的区别：

只读属性，声明了窗口的文档显示区的高度和宽度，以像素为单位。这里的宽度和高度不包括菜单栏、工具栏以及滚动条等的高度。只有window有这个属性，用window.innerHeight/innerWidth获取。

● outerHeight/outerWidth同理：

outerHeight属性设置或返回一个窗口的外部高度，包括所有界面元素（如工具栏/滚动条）。outerWidth属性设置或返回窗口的外部宽度，包括所有的界面元素（如工具栏/滚动）。

移动端适配：rem、vw/vh/vmin/vmax、viewport、media query

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

一般写为<meta name="viewport" content="width=device-width, initial-scale=1">即可

media query：@media screen and (max-width:720px) and (min-width:320px)

链接：<https://www.jianshu.com/p/ac9cf34340b6>

### [DOMContentLoaded与load的区别](https://www.baidu.com/link?url=CBYc9dyXXy7ab6tVYdIMK_MNYCdDrZFLXB4r-h-PbLaWvsAlFbG8waKWbHMPWBDiO6OG0OnGudTSGIVXtjGe6a&wd=&eqid=ba9712cc0001d97f000000065ba5da94" \t "https://www.baidu.com/_blank)

1.当文档中没有脚本时，浏览器解析完文档便能触发 DOMContentLoaded 事件；如果文档中包含脚本，则脚本会阻塞文档的解析，而脚本需要等位于脚本前面的css加载完才能执行。在任何情况下，DOMContentLoaded 的触发不需要等待图片等其他资源加载完成。

2.load，页面上所有的资源（图片，音频，视频等）被加载以后才会触发load事件，简单来说，页面的load事件会在DOMContentLoaded被触发之后才触发。

模块化：防止全局污染、去耦合、方便维护。解决**命名冲突**和**模块间依赖管理**的问题。

工程化：因为开发环境需要模块化、可能使用Sass或TypeScript编程，因此会导致文件数过多，影响HTTP请求；而且，Sass和TypeScript浏览器也无法解析，需要将其转化为CSS和JavaScript。所以，需要进行**转化、压缩、打包**后将文件放在线上环境中进行，而**自动处理这个过程的行为，称为工程化**。

CSS优先级：（CSS2特异性）

● 第一优先级：无条件优先的属性只需要在属性后面使用！important。它会覆盖页面内任何位置定义的元素样式。ie6不支持该属性。

● 第二优先级：在html中给元素标签加style，即内联样式。该方法会造成css难以管理，所以不推荐使用。

● 第三优先级：由一个或多个id选择器来定义。例如，#id{margin:0;}会覆盖.classname{margin:3pxl}

● 第四优先级：由一个或多个类选择器、属性选择器、伪类选择器定义。如.classname{margin:3px}会覆盖div{margin:6px;}

● 第五优先级：由一个或多个元素选择器、伪元素选择器定义。如div{marigin:6px;}会覆盖\*{margin:10px；}

● 第六优先级：通配选择器，如\*{marigin:6px;}

问题：`#a .b{}` 与 `.a #b` 谁优先级高？

答案：优先级相同，谁写在后面，应用谁的样式。

HTTP2：

1. 二进制分帧：比起像HTTP/1.x这样的文本协议，二进制协议解析起来更高效、“线上”更紧凑，更重要的是错误更少。因为它们对如空白字符的处理、大小写、行尾、空链接等的处理很有帮助。

（在应用层(HTTP/2)和传输层(TCP or UDP)之间增加一个二进制分帧层在二进制分帧层中， HTTP/2 会将所有传输的信息分割为更小的消息和帧（frame）,并对它们采用二进制格式的编码。帧对数据进行顺序标识，数据以乱序并行传输，在客户端再进行排序，以此来实现多路复用，且只需要1条HTTP连接。（HTTP/1.x是文本协议只能串行传输，因此需要开多条HTTP连接，但浏览器会限制最大并发数，所以请求过多还是会被阻塞）其中 HTTP1.x 的首部信息会被封装到 HEADER frame，而相应的 Request Body 则封装到 DATA frame 里面。）

1. 多路复用：允许同时通过单一的 HTTP/2 连接发起多重的请求-响应消息（在 HTTP/1.1 协议中 「浏览器客户端在同一时间，针对同一域名下的请求有一定数量限制，一般并发数为6。超过限制数目的请求会被阻塞」）

链接：<https://segmentfault.com/a/1190000011172823>

1. 一个域只需要一个TCP连接：（HTTP/1.x是文本协议只能串行传输，因此需要开多条HTTP连接，但浏览器会限制最大并发数，一般为6，所以请求过多，有些请求还是会被阻塞）（HTTP/2 通过二进制分帧和多路复用，让所有数据流共用同一个连接，可以更有效地使用 TCP 连接，让高带宽也能真正的服务于 HTTP 的性能提升。）
2. 头部压缩：使用HPACK进行头部压缩（因为HTTP头部更替不是很频繁，所以正在client和server端做个缓存，http2.0使用encoder来减少需要传输的header大小，通讯双方各自cache一份header fields表，既避免了重复header的传输，又减小了需要传输的大小。）
3. 服务端推送（Server Push）：H2允许服务器，预先将网页所需要的资源PUSH到浏览器的内存中（如请求page.html，server会push给我们script.js和style.css，省去了requset的开销）

链接：<https://www.zhihu.com/question/34074946>

链接：<https://blog.csdn.net/jiayanhui2877/article/details/44957105>

Vue生命周期

beforecreated：el 和 data 并未初始化   
created:完成了 data 数据的初始化，el没有  
beforeMount：完成了 el 和 data 初始化   
mounted ：完成挂载

链接：<https://segmentfault.com/a/1190000008010666>

LocalStorage:



链接：<https://developer.mozilla.org/zh-CN/docs/Web/API/Window/localStorage>

设计模式：（待完善）

单例模式：用一个命名空间将自己的属性与方法包起来，防止与他人的属性和方法冲突；并且，可以用来管理私有变量；

同源策略：是一种约定。

● 同源政策的目的：是为了保证用户信息的安全，防止恶意的网站窃取数据。

● 同源策略应该是浏览器在接收加载资源之前对其来源进行了检查，然后限制加载。

● 目前，如果非同源，共有三种行为受到限制：

（1） Cookie、LocalStorage 和 IndexDB 无法读取。

（2） DOM 无法获得。

（3） AJAX 请求不能发送。

● AJAX的同源策略限制：可能限制发出（保护用户的cookie）或者限制接收（隔绝不信任的服务器返回的数据）【根据MDN：https://developer.mozilla.org/zh-CN/docs/Web/HTTP/Access\_control\_CORS，不同浏览器实现方式不同，看浏览器产商自己怎么做。一般是限制发出】

如果使用CORS，得跟服务器先协商好，（默认用户自己手动请求url的服务器可信），如果返回Access-control-allow-orign的话，说明此服务器可信，数据可接收（红宝书写：CORS定义了在必须访问跨域资源时，浏览器与服务器应该如何沟通。定义CORS是一种沟通方式）。

如果用户自己请求了某个恶意网站，这个网站返回Access-control-allow-orign:\*，就会接收恶意数据了。

（这是一种XSS+CSRF的攻击方式：诱导用户去点小广告，点完便执行请求返回恶意数据）

但不能获取用户的cookie：如果设置access-control-allow-origin为\*星号（任何域），则Access-Control-Allow-Credentials头是不能设置为true的。并且客户端也需要设置xhr.withCredentials = true;

一个面试问题：（如果不同源的话，浏览器会报错，不能获得服务器的数据。如果我是攻击者，我发送请求到后台以求获得数据，但这个行为还得征得服务器同意，那么同源策略是保护服务器还是浏览器的？）

（答：都有保护作用。主要保护用户客户端的信息不被其他域获取，服务器如果没有做过滤的话肯定也会被攻击。同源策略就像一个防火墙，保护同一域下的信息不会泄露出去。但是这个要配合很多技巧，不然也还是能获取的（各种攻击手段了解一下），只是不那么容易获取罢了）

AJAX原理：<https://www.cnblogs.com/jackson0714/p/AJAX.htm>

CORS(阮一峰)：<http://www.ruanyifeng.com/blog/2016/04/cors.html>

Git pull & git fecth：

● git fetch只会将本地库所关联的远程库的commit id更新至最新

● git pull会将本地库更新至远程库的最新状态

● git pull = git fetch + git merge



链接：<https://blog.csdn.net/a19881029/article/details/42245955>

# git版本回退(除了hard还有其他2种，待完善)

git reset --hard `commit-id`

# **[Form表单点击submit按钮提交，阻止页面跳转到action地址：](https://segmentfault.com/q/1010000006779322)**

方法1：监听 form 的 onsubmit 事件

$('form').on('submit', function (e) {

e.preventDefault()

// ....

}

方法2：让target指向隐藏的iframe

<https://blog.csdn.net/wobendiankun/article/details/38667379>

方法3：使用Ajax

**flex布局中align-items 和align-content的区别：**

1.align-items属性适用于所有的flex容器，它是用来设置每个flex元素在侧轴上的**默认对齐方式**。

2.align-content属性**只适用于多行**的flex容器，并且当侧轴上有多余空间使flex容器内的flex线对齐。

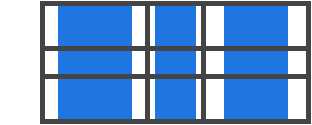
\*注：flex只有justify-content没有justify-items

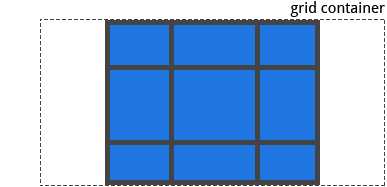
链接：<https://blog.csdn.net/sinat_27088253/article/details/51532992>

**Grid布局中align-items/justify-items 和align-content/justify-content的区别：**

1. 一般用align-items/justify-items，

某个item修改则用align-self/justify-self



2.align-content/justify-content：有时，网格的总大小可能小于其网格容器的大小。如果你的所有 grid items 都使用像px这样的非弹性单位来设置大小，则可能发生这种情况。此时，你可以设置网格容器内的网格的对齐方式。 此属性沿着行轴对齐网格（与之对应的是 align-content, 沿着列轴对齐）。

#### 链接：<https://segmentfault.com/a/1190000012889793#articleHeader5>

**强缓存：**状态码：200；相关字段：cache-control：max-age、expire

**协商缓存：**状态码：304(无改动)、200(有改动)；相关字段：Etag(if-None-Match)、last-modefied(If-Modified-Since)

# **状态码**

# **链接：<https://blog.csdn.net/qq_32963841/article/details/78365449>**

# **Promise的队列与setTimeout的队列有何关联（微任务与宏任务）**



原因：因为 setTimeout 是属于 macrotask 的，而整个 script 也是属于一个 macrotask, promise.then 回调 是 microtask ,执行过程大概如下：

* 由于整个 script 也属于一个 macrotask, 由于会先执行 macrotask 中的第一个任务，再加上**promise 构造函数因为是同步的**，所以会先打印出 1和2
* 然后继续同步执行末尾的 console.log(3) 打印出3
* 此时 setTimeout 被推进到 macrotask 队列中， promise.then 回调被推进到 microtask 队列中
* 由于在第一步中已经执行完了第一个 macrotask , 所以接下来会顺序执行所有的 microtask, 也就是 promise.then 的回调函数，从而打印出5
* microtask 队列中的任务已经执行完毕，继续执行剩下的 macrotask 队列中的任务，也就是 setTimeout, 所以打印出 4

“宏任务”在下一轮Event Loop执行，“微任务”在本轮Event Loop的所有任务结束后执行

链接：<https://www.zhihu.com/question/36972010>

**let & const & var :**

● let 声明的变量的作用域是块级的；

● let 不能重复声明已存在的变量；

● let 会被提升，只是有暂时死区（ 所谓暂时死区，就是不能在初始化之前，使用变量 ）

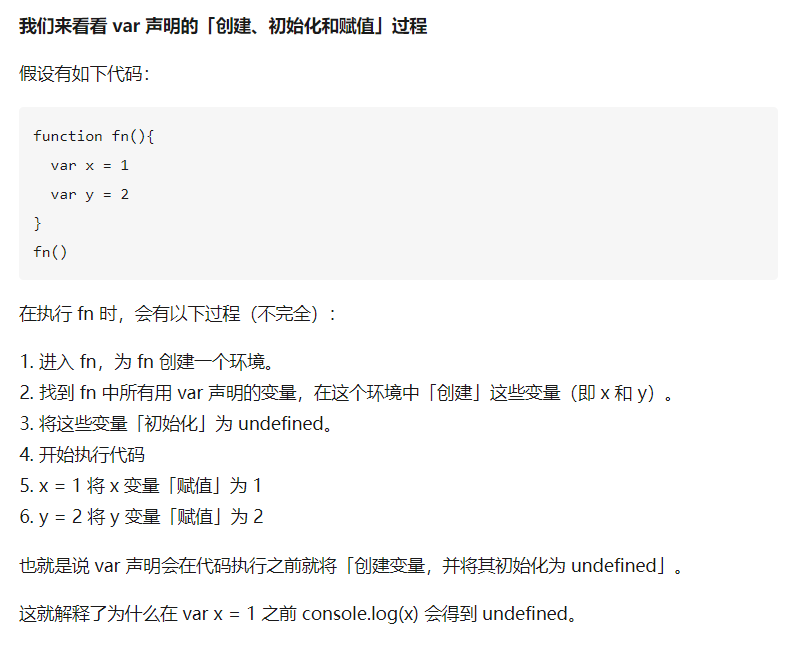
● 提升：

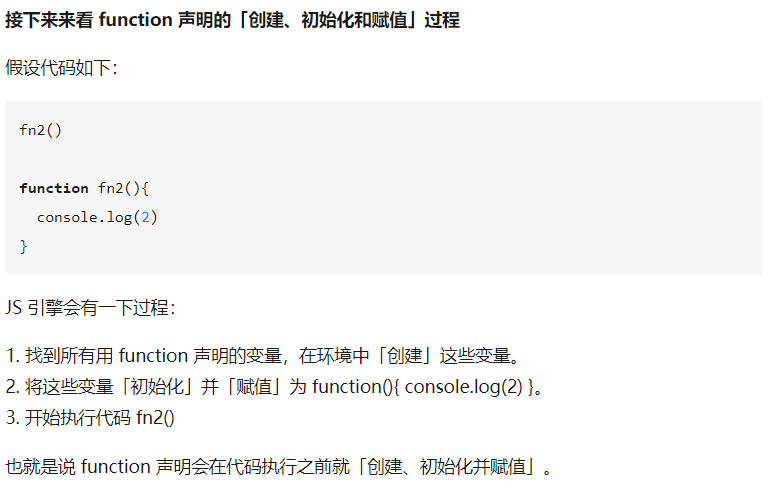
● let 的「创建」过程被提升了，但是初始化没有提升。（MDN说不会提升是错的）

● var 的「创建」和「初始化」都被提升了。

● function 的「创建」「初始化」和「赋值」都被提升了。

● const 和 let 只有一个区别，那就是 const 只有「创建」和「初始化」，没有「赋值」过程







链接：<https://zhuanlan.zhihu.com/p/28140450>

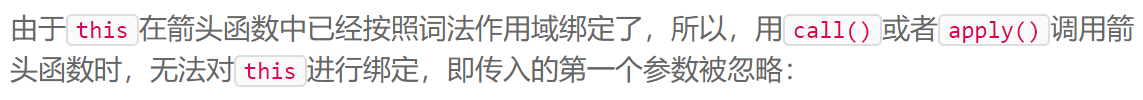
链接：https://zhuanlan.zhihu.com/p/27558914

**箭头函数的this（箭头函数与普通函数的区别）：**



● 箭头函数的this对象是绑定定义时时所在的作用域。

● 普通函数的this对象是绑定使用时所在的作用域。

链接：<https://www.liaoxuefeng.com/wiki/001434446689867b27157e896e74d51a89c25cc8b43bdb3000/001438565969057627e5435793645b7acaee3b6869d1374000>

链接：<http://es6.ruanyifeng.com/#docs/function#%E7%AE%AD%E5%A4%B4%E5%87%BD%E6%95%B0>

## **什么时候不能使用箭头函数：**

## 定义对象方法、定义原型方法、定义构造函数、定义事件回调函数。

## 链接：<https://juejin.im/post/58fda157570c350058e6afb8>

## **export与export default**

export跟export default 有什么区别呢？如下：

export与export default均可用于导出常量、函数、文件、模块等  
2、你可以在其它文件或模块中通过import+(常量 | 函数 | 文件 | 模块)名的方式，将其导入，以便能够对其进行使用  
3、在一个文件或模块中，export、import可以有多个，export default仅有一个  
4、通过export方式导出，在导入时要加{ }，export default则不需要

5、

链接：<https://www.cnblogs.com/xiaotanke/p/7448383.html>

链接：<https://segmentfault.com/q/1010000010126010>

链接：<https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Statements/export>

**模块化区别**

**‌conmonjs**

服务端模块的规范，**同步加载**

**‌AMD**

**异步加载，提前执行（依赖）**：当所有依赖被加载和执行完成后，执行传入的回调函数才被执行（加载>执行>回调）

**依赖前置**，在解析和执行当前模块之前，模块作者必须指明当前模块所依赖的模块。代码在一旦运行到此处，能立即知晓依赖。而无需遍历整个函数体找到它的依赖，因此性能有所提升，缺点就是开发者必须显式得指明依赖——这会使得开发工作量变大。

define(['./a','./b'],function(a,b){

a.doSomething()

b.doSomething()

})

**‌CMD**

**异步加载，延迟执行（依赖）**：回调函数内部依赖的js全部加载（暂不调用）完后，调用回调函数。当回调函数调用require('jquery')，即执行绑定在'jquery'这个id上的js文件，即刻执行，并传回返回值（加载>回调>执行）

**依赖就近**，可以把依赖写进你的代码中的任意一行。代码在运行时，首先是不知道依赖的，需要遍历所有的require关键字，找出后面的依赖。具体做法是将function toString后，用正则匹配出require关键字后面的依赖。显然，这是一种牺牲性能来换取更多开发便利的方法。

define(function(require, exports, module{

var a = require('./a')

a.doSomething()

var b = require('./b')

b.doSomething()

})

链接：<https://segmentfault.com/q/1010000008543797>

**‌ES6模块化与CommonJS**

**CommonJS** 输出是**值的拷贝**，即原来模块中的值改变不会影响已经加载的该值，**ES6**静态分析，动态引用，输出的是**值的引用**，值改变，引用也改变，即原来模块中的值改变则该加载的值也改变。

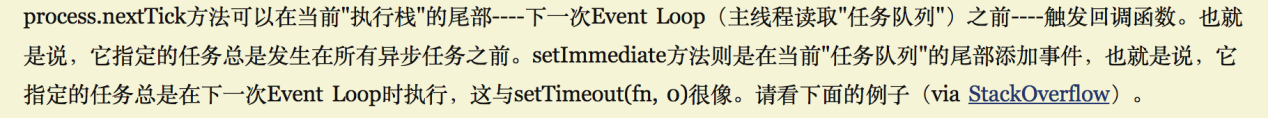
**CommonJS 模块是运行时加载，ES6 模块是编译时输出接口。**

**CommonJS 加载的是整个模块，即将所有的接口全部加载进来，ES6 可以单独加载其中的某个接口（方法），**

**CommonJS this 指向当前模块，ES6 this 指向undefined**

链接：<https://blog.csdn.net/lhjuejiang/article/details/80274212>

**setImmediate & process.nextTick:**



1.process.nextTick：添加到执行栈的尾部

2.setImmediate：添加到任务队列的尾部，浏览器端相当于setTimeout(f(){},0)，Node端是在IO callback之后

3.setImmediate&setTimeout这么写的话，顺序不确定,这得看cpu：

链接：<http://www.ruanyifeng.com/blog/2014/10/event-loop.html>

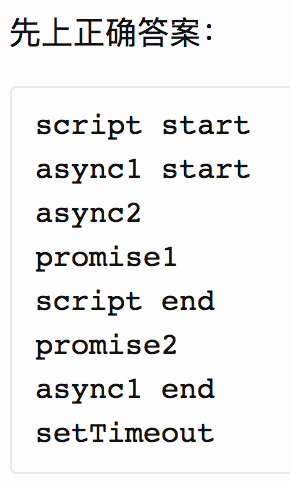
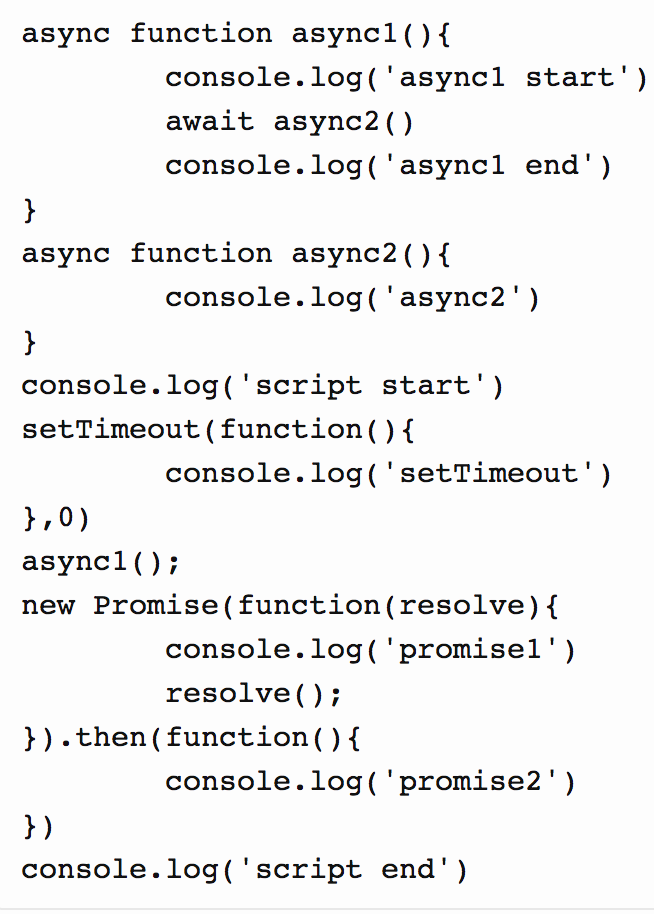
**宏任务与微任务：**

正常任务（task）与微任务（microtask）。它们的区别在于，“正常任务”在下一轮Event Loop执行，“微任务”在本轮Event Loop的所有任务结束后执行。

**正常任务**包括以下情况：**setTimeout、setInterval、setImmediate、I/O、各种事件（比如鼠标单击事件）的回调函数**

**微任务**目前主要是**process.nextTick**和 **Promise.then** 这两种情况

# 链接：<https://blog.csdn.net/shuidinaozhongyan/article/details/78162134>**promise、async和await之执行顺序的那点事:**



要点：

1.先执行调用栈的同步事件；

2.setTimeout里的函数是异步执行，放在宏任务队列（下一次event loop执行）；

3.Promise是立即执行（同步执行）；Promise里的resolve是异步执行，并且放在微任务队列（本次event loop末尾执行）；

4.async 函数返回一个 Promise 对象;

5.当函数执行的时候，一旦遇到 await 就会先返回，等到触发的异步操作完成，再接着执行函数体内后面的语句；

过程：

1. 执行一遍调用栈的同步事件，输出如上：script start-->...-->script end
2. 此时调用栈为空；微任务队列有（1）async2执行结束返回的Promise对象、（2）new Promise的resolve；宏任务队列有setTimeout里的函数。
3. 先调用“async2执行结束返回的Promise对象”，执行其Promise对象，执行到resolve之前（这是一个值为Undefined的resolve），将resolve放入微任务队列末尾；
4. 接着执行下一个微任务“new Promise的resolve”，输出‘promise2’；
5. 接着执行刚刚放进去的微任务“async2的Promise对象resolve”，无输出；
6. 接着因为await后面的函数执行结束，开始执行await下面的代码，输出’async1 end’;
7. 接着因为微任务队列为空，所以将宏任务队列的setTimeout放入的函数放入调用栈并执行，输出’setTimeout’
8. 结束

链接：<https://lvdingjin.github.io/tech/2018/05/27/async-and-await.html>

**JavaScript的Object查询时间复杂度是O(1)：**

V8的properties和elements都有“快速”和“慢速”模式。快速模式用密集的存储方式（FixedArray），慢速模式用哈希表（Dictionary）。对数组而言，稀疏数组一定是用慢速模式。

链接：<https://www.zhihu.com/question/30848981>

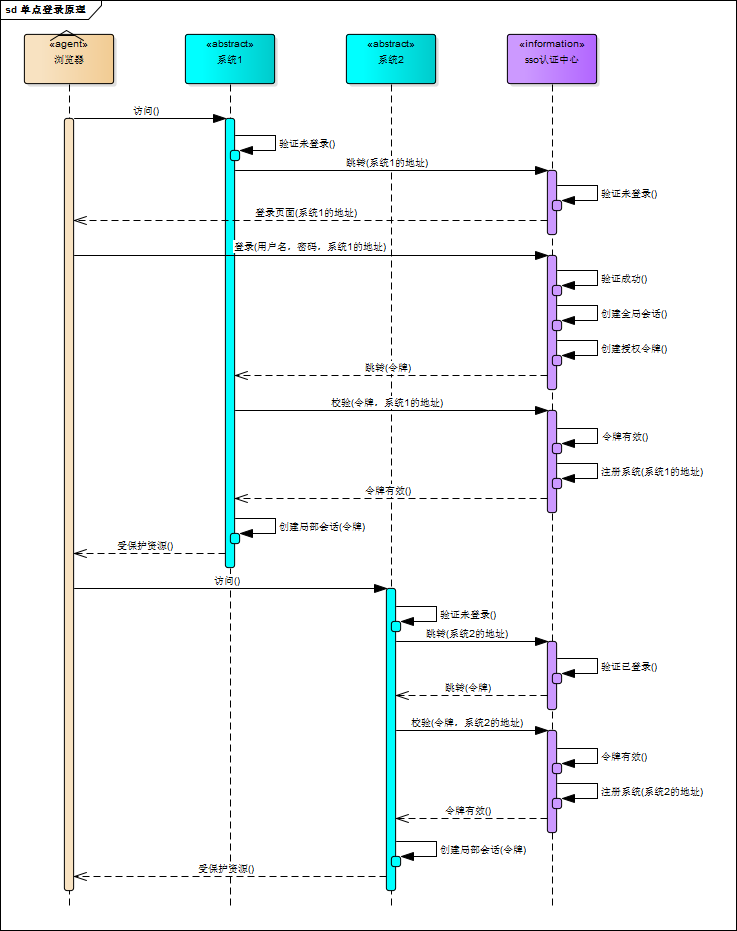
Element.getBoundingClientRect()

返回元素的大小及其相对于视口的位置

配合 window.innerHeight/document.documentElement.clientHeight，可以实现下拉刷新，监控曝光

链接：<https://developer.mozilla.org/zh-CN/docs/Web/API/Element/getBoundingClientRect>

# 单点登录之CAS原理和实现：





过程：（ssosv是sso的登陆页面）

1. 访问服务器a请求数据
2. 服务器a验证为未登陆状态，（携带服务器a的地址）请求sso服务器验证登陆状态
3. sso服务器（携带服务器a的地址）302跳转到ssosv页面
4. ssosv页面携带cookie请求ssosv网关
5. ssosv网关302后到ssosv登陆接口，并携带cookie请求
6. ssosv服务器验证通过，返回ticket并302到sso服务器
7. sso服务器验证ticket通过，200
8. 服务器a创建session回话

链接：

<https://blog.csdn.net/qq_24708791/article/details/78535565>

https://www.cnblogs.com/ywlaker/p/6113927.html

**React/Vue区别：**

MVC/MVVM，大型小型应用，JSX/模板，v-for/map，state状态管理/data数据

**响应式开发最佳实践（viewport+rem）:**

viewport只是改变document.documentElement.clientWidth或window.innerWidth

viewport的值取width与 initial-scale的最大值

链接：https://github.com/jawil/blog/issues/21

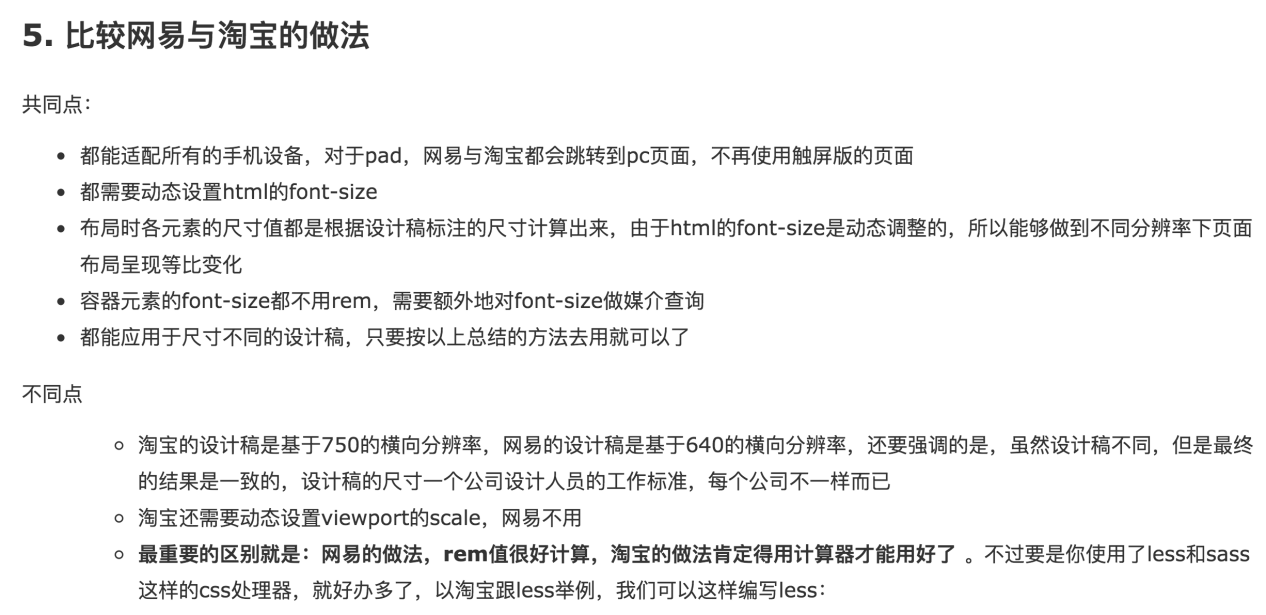
响应式开发最佳实践：<http://www.cnblogs.com/lyzg/p/4877277.html>

自己写的响应式博客文章：<https://www.jianshu.com/p/ac9cf34340b6>

**在实际项目中，把与元素尺寸有关的css，如width,height,line-height,margin,padding等都以rem作为单位，这样页面在不同设备下就能保持一致的网页布局。**

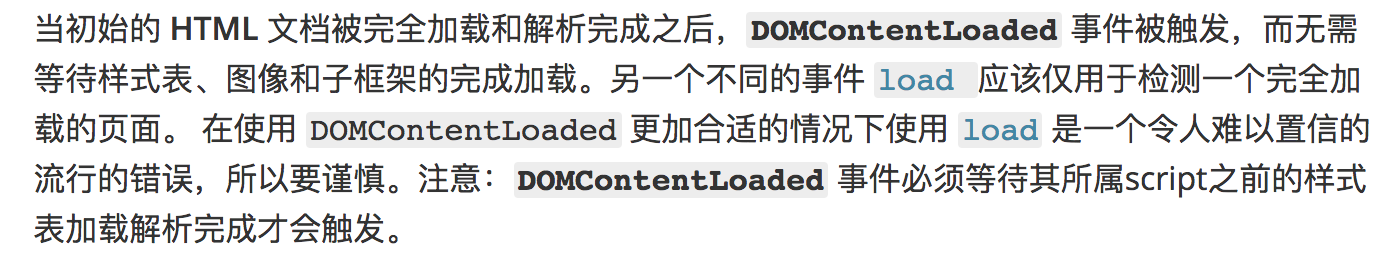
**Font-size用媒体查询改变px，不能设为rem。**





Domready: <https://www.jianshu.com/p/88b9d3874749>

Domready使用“DOMContentLoaded”事件





链接：<https://developer.mozilla.org/zh-CN/docs/Web/Events/DOMContentLoaded>

获取一个页面的节点类型数量：

```

new Set([...document.querySelectorAll('\*')].map(node => node.nodeName)).size

```

1. index原则：

一、

z-index只对设置了position(非static)的元素有用

二、

一般原则：

1. 写在后面优先级大
2. 设置了position优先级比无设置的大
3. 都设置了position，则比较z-index
4. Z-index相同，则写在后面优先级大

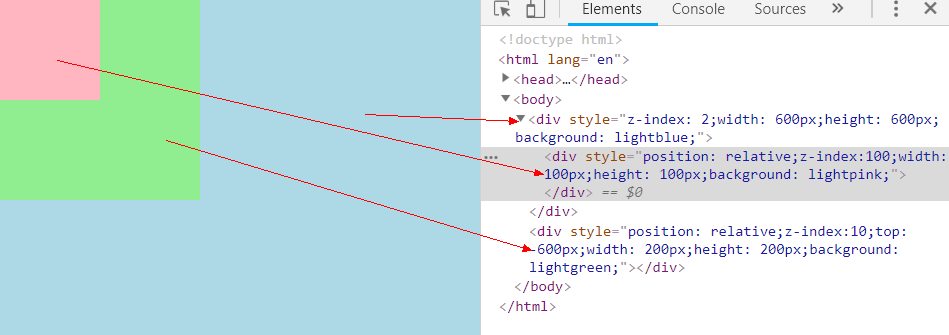
三、

子元素一定在父元素上面

四、

若父元素1无设置position，子元素2设置position，兄弟元素3设置了position

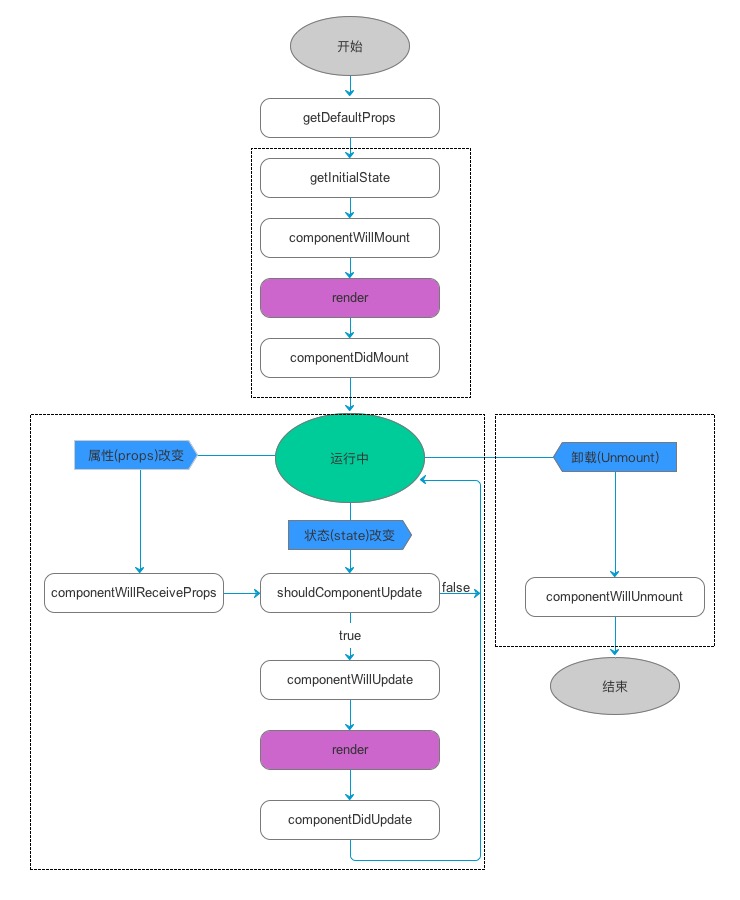
则3在1上面，再3与2比较z-index



1. 层叠等级

链接：<https://webdesign.tutsplus.com/zh-hans/articles/what-you-may-not-know-about-the-z-index-property--webdesign-16892>

React生命周期



* **初始化**

1、getDefaultProps()

设置默认的props，也可以用dufaultProps设置组件的默认属性.

2、getInitialState()

在使用es6的class语法时是没有这个钩子函数的，可以直接在constructor中定义this.state。此时可以访问this.props

3、componentWillMount()

组件初始化时只调用，以后组件更新不调用，整个生命周期只调用一次，此时可以修改state。

4、 render()

react最重要的步骤，创建虚拟dom，进行diff算法，更新dom树都在此进行。此时就不能更改state了。

5、componentDidMount()

组件渲染之后调用，只调用一次。

* **更新**

6、componentWillReceiveProps(nextProps)

组件初始化时不调用，组件接受新的props时调用。

7、shouldComponentUpdate(nextProps, nextState)

react性能优化非常重要的一环。组件接受新的state或者props时调用，我们可以设置在此对比前后两个props和state是否相同，如果相同则返回false阻止更新，因为相同的属性状态一定会生成相同的dom树，这样就不需要创造新的dom树和旧的dom树进行diff算法对比，节省大量性能，尤其是在dom结构复杂的时候

8、componentWillUpdata(nextProps, nextState)

组件初始化时不调用，只有在组件将要更新时才调用，此时可以修改state

9、render()

组件渲染

10、componentDidUpdate()

组件初始化时不调用，组件更新完成后调用，此时可以获取dom节点。

* **卸载**

11、componentWillUnmount()

组件将要卸载时调用，一些事件监听和定时器需要在此时清除。

链接：<https://www.cnblogs.com/qiaojie/p/6135180.html>

**栈内存与堆内存**

栈是为执行线程留出的内存空间。当函数被调用的时候，栈顶为局部变量和一些 bookkeeping 数据预留块。当函数执行完毕，块就没有用了，可能在下次的函数调用的时候再被使用。

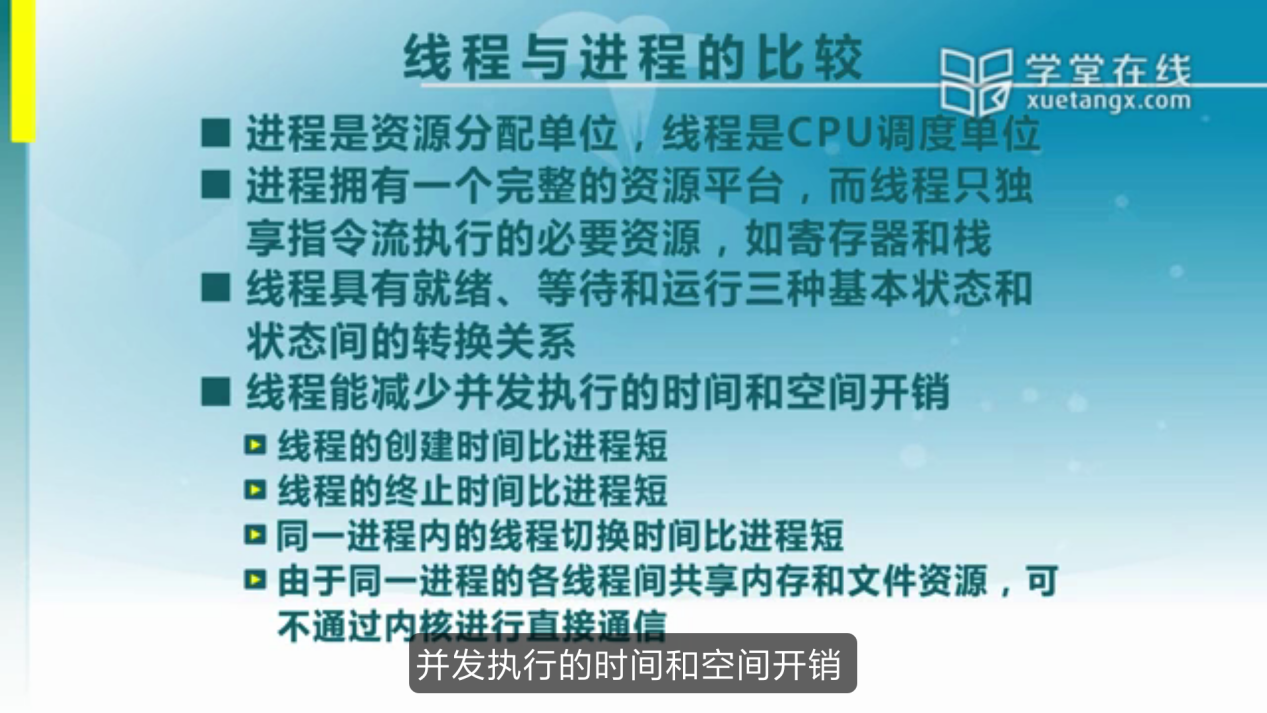
堆（heap）是为动态分配预留的内存空间。和栈不一样，从堆上分配和重新分配块没有固定模式；你可以在任何时候分配和释放它。

每一个线程都有一个栈，但是每一个应用程序通常都只有一个堆（尽管为不同类型分配内存使用多个堆的情况也是有的）。

1. 当线程创建的时候，操作系统（OS）为每一个系统级（system-level）的线程分配栈。通常情况下，操作系统通过调用语言的运行时（runtime）去为应用程序分配堆。 2. 栈附属于线程，因此当线程结束时栈被回收。堆通常通过运行时在应用程序启动时被分配，当应用程序（进程）退出时被回收。 3. 当线程被创建的时候，设置栈的大小。在应用程序启动的时候，设置堆的大小，但是可以在需要的时候扩展（分配器向操作系统申请更多的内存）。 4. 栈比堆要快，因为它存取模式使它可以轻松的分配和重新分配内存（指针/整型只是进行简单的递增或者递减运算），然而堆在分配和释放的时候有更多的复杂的 bookkeeping 参与。另外，在栈上的每个字节频繁的被复用也就意味着它可能映射到处理器缓存中，所以很快（译者注：局部性原理）。

链接：<http://blog.jobbole.com/75321/>

**线程与进程**





运行->等待（堵塞）：当进程请求某资源且必须等待这个资源加载完成时；

运行->就绪：时间片用完，或被更高优先级的进程抢占处理器等；

[牛客网-当一个进程时间片用完从CPU上退下来时，它的状态应变为()。](https://www.nowcoder.com/questionTerminal/420073242511483f99b9d56718551051)





**DNS：**

# DNS使用TCP和UDP的端口号**53**

**DNS在进行区域传输的时候使用TCP协议，其它时候则使用UDP协议。**

DNS的规范规定了2种类型的DNS服务器，一个叫主DNS服务器，一个叫辅助DNS服务器。在一个区中主DNS服务器从自己本机的数据文件中读取该区的DNS数据信息，而辅助DNS服务器则从区的主DNS服务器中读取该区的DNS数据信息。当一个辅助DNS服务器启动时，它需要与主DNS服务器通信，并加载数据信息，这就叫做区传送（zone transfer）。

辅域名服务器会定时（一般时3小时）向主域名服务器进行查询以便了解数据是否有变动。如有变动，则会执行一次区域传送，进行数据同步。****区域传送将使用TCP而不是UDP，一是因为数据同步传送的数据量比一个请求和应答的数据量要多得多；二是因为TCP是一种可靠的连接，保证了数据的准确性。****

为什么既使用TCP又使用UDP？  
　　首先了解一下TCP与UDP传送字节的长度限制：  
　　UDP报文的最大长度为512字节，而TCP则允许报文长度超过512字节。当DNS查询超过512字节时，协议的TC标志出现删除标志，这时则使用TCP发送。通常传统的UDP报文一般不会大于512字节。

链接：<https://baike.baidu.com/item/dns%E5%8D%8F%E8%AE%AE/1860066>

笔试选择题：

```

1-----------------------------------------------------------

var arr = [,,,,,] // 5个逗号，最后一项会被省略，所以arr.length是5

console.log(arr.length) // 5

2-----------------------------------------------------------

~(2+'3') // -24，~的作用是按位非，计算可当成改变符号后减1；

// 原理:如~1，1的二进制00000001->取反11111110->除符号位外取反10000001->加1得补码10000010->十进制的-2

3-----------------------------------------------------------

34e5+50 // 3400050,e5是10的5次方

4-----------------------------------------------------------

const a = [3,5];

const f = (b,...a)=>a+b;

f(2); //”2”,因为参数a=[],a+b=[]+2=””+2=”2”

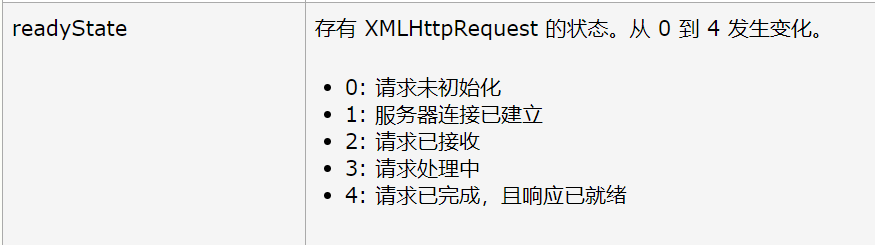
1. ----------------------------------------------------------

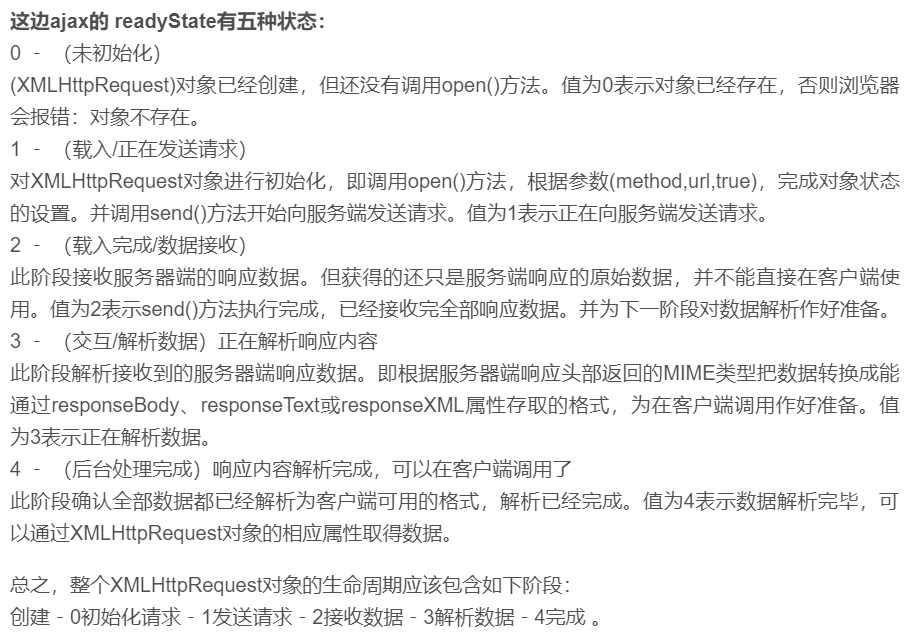
‘5’+3 // ‘53’

‘5’-3 // 2

```

AJAX-readState





链接：<http://www.w3school.com.cn/ajax/ajax_xmlhttprequest_onreadystatechange.asp>

<https://blog.csdn.net/ranchonono/article/details/76903994>

video 元素支持三种视频格式：

Ogg/MPEG 4/WebM



链接：<http://www.w3school.com.cn/html5/html_5_video.asp>

浮点数加减：

浮点数的精度问题不是JavaScript特有的，因为有些小数以**二进制表示位数是无穷的。**

**浮点数加减：使用Number.toFixed(0-20)，**把 Number 四舍五入为指定小数位数的数字。

**链接：<https://www.cnblogs.com/ppforever/p/5011660.html>**

**链接：<http://www.w3school.com.cn/jsref/jsref_tofixed.asp>**

# ****Ul插入100万个li的优化:**requestAnimationFrame+DocumentFragment**

**链接：<https://juejin.im/post/5ae59db46fb9a07a9e4cfe07>**

**js如何实现面向对象？js如何利用原型链实现继承？**

**js通过函数模拟类，通过函数的构造调用(new操作符)实例化一个对象，new操作符的作用(5点balabala)，对象拥有类定义的属性和方法，可以通过obj.attribure或obj.function的方式操作对象的属性和方法；类之间通过原型链的方式实现继承，若某个属性或方法找不到，则沿着原型链一层一层往上找，知道找到为止，若找不到则返回undefined，一般对象的原型链最顶层是Object对象；通过这种方法底层的属性/方法可以覆盖上层，实现类的多态。**

**利用原型链实现继承有2种方式：**

**1.subType.prototype = new superType();**

**2.subType.prototype = Object.create(superType.prototype)**

****new操作符：****

1. **新建一个对象**
2. **this指向这个对象**
3. **这个对象继承构造函数的prototype**
4. **执行构造函数，进行赋值**
5. **若显示return一个对象，则返回这个对象；否则隐式返回刚构造完的对象**

****prototype与\_\_proto\_\_区别****

**\_\_proto\_\_是每个对象都有的一个属性，而prototype是函数才会有的属性**

**对象具有属性\_\_proto\_\_，可称为隐式原型，一个对象的隐式原型指向构造该对象的构造函数的原型**

**prototype是通过调用构造函数而创建的那个对象实例的原型对象**

**hasOwnProperty()判断指定属性是否为自有属性；in操作符对原型属性和自有属性都返回true**

**可以调用对象的Object.getPrototypeOf()方法读取[[Prototype]]属性的值，也可以使用isPrototypeOf()方法检查某个对象是否是另一个对象的原型对象。大部分JavaScript引擎在所有对象上都支持一个名为\_\_proto\_\_的属性，该属性可以直接读写[[Prototype]]属性。**

**链接：**<https://blog.csdn.net/ligang2585116/article/details/53522741>

****客户端渲染与服务器渲染****

**链接：**<https://www.cnblogs.com/zhuzhenwei918/p/8795945.html>

****vue双向绑定****

**1.采用数据劫持结合发布者-订阅者模式。**

**2.有3个模块：发布者（Observer），编译器（Compile），订阅者（Watcher）**

**3.首先，发布者通过object.defineProperty定义了getter方法，用来添加订阅者；通过定义setter方法，用来劫持数据，若数据改变，则通知（notify）他的订阅者数据已变化；**

**4.订阅者收到发布者的数据变化信息，调用自己的update方法更新自己的数据，并触发编译器更新视图；**

**5.编译器的作用是解析模板指令，将模板中的变量替换成数据，更新视图**

****vue虚拟DOM****

**1.DOM是树型结构，一般修改DOM会使用innerHTML修改它自己以及它的所有子树，开销过大**

**2.JS可以用JS对象结构表示DOM树型结构，这就是虚拟DOM**

**3.每次文档状态变化，会新建一颗虚拟DOM树，并将新树与旧树进行对比，记录两棵树的差异**

**4.将对比后的差异应用到真实的DOM树上**

# mousedown mouseup focus click事件的触发顺序

**Mousedown->focus->mouseup->click**

# keydown keypress keyup事件的触发顺序

**Keydown->keypress->keyup**

# 文本块中首行文本的缩进:text-indent

# 字符间距:letter-spacing

****Touch事件****

最基本的touch事件包括4个事件：

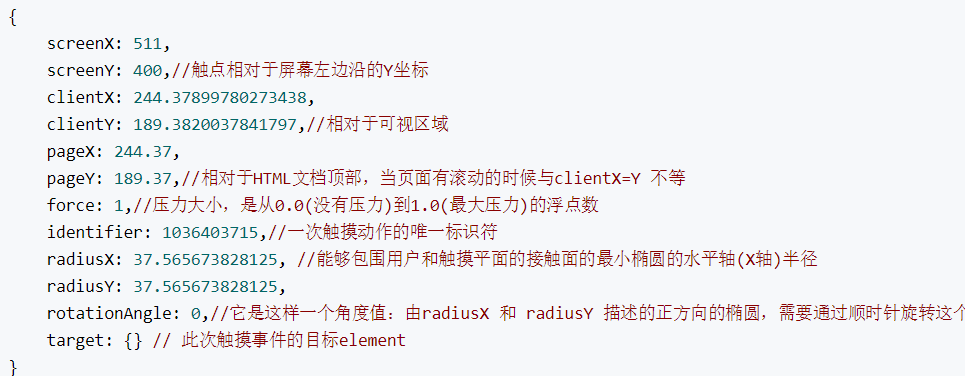
touchstart 当在屏幕上按下手指时触发

touchmove 当在屏幕上移动手指时触发

touchend 当在屏幕上抬起手指时触发

touchcancel 当一些更高级别的事件发生的时候（如电话接入或者弹出信息）会取消当前的touch操作，即触发touchcancel。一般会在touchcancel时暂停游戏、存档等操作。

属性



**Vue/React框架层面的优化**

1.路由

2.状态管理

3.服务端渲染

**面向对象编程、函数式编程、响应式编程：（待完善）**

**函数式编程：关注的重点是需要描述什么，而不是如何描述。是一个纯函数：**

符合两个条件：

1.**此函数在相同的输入值时，总是产生相同的输出**。函数的输出和当前运行环境的上下文状态无关。

2.**此函数运行过程不影响运行环境，也就是无副作用**（如触发事件、发起http请求、打印/log等）。

**map & reduce他们是最常用的函数式编程的方法**

#### **纯函数：**对于相同的输入，永远会得到相同的输出，而且没有任何可观察的副作用，也不依赖外部环境的状态的函数，叫做纯函数。

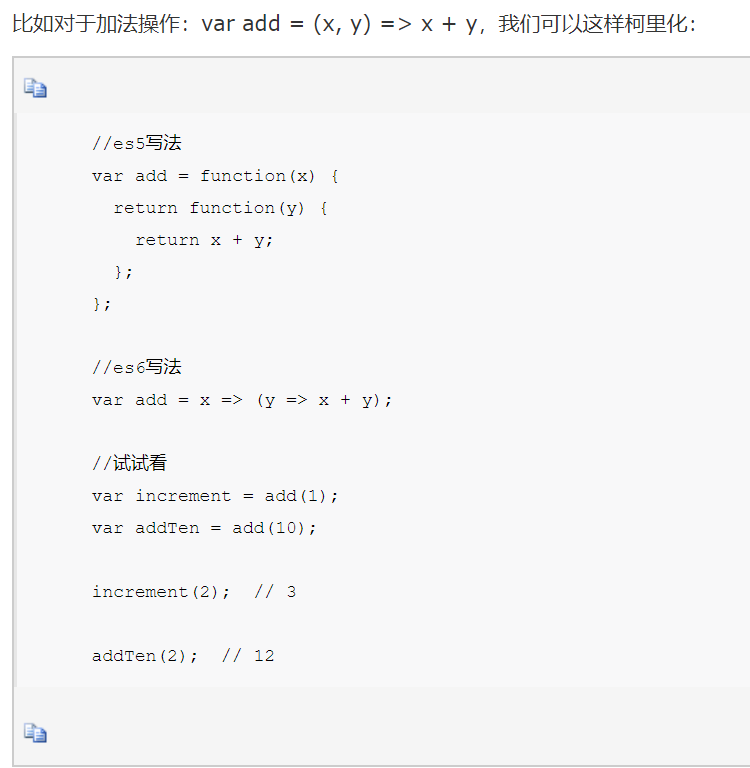
#### **函数柯里化：**传递给函数一部分参数来调用它，让它返回一个函数去处理剩下的参数

#### **函数组合：**为了解决函数嵌套过深，洋葱代码：h(g(f(x)))，我们需要用到“函数组合”，我们一起来用柯里化来改他，让多个函数像拼积木一样。

链接：<https://www.cnblogs.com/fengyuqing/p/functional_programming_1.html>

链接：<https://www.cnblogs.com/tjyoung/p/8976013.html>

****函数柯里化：****



链接：<https://www.cnblogs.com/fengyuqing/p/functional_programming_1.html>

TCP拥塞控制：

慢开始、拥塞避免、快重传、快恢复

Referrer

\* Document.referrer查看referrer

\* meta referrer

<meta name="referrer" content="no-referrer">



<https://developer.mozilla.org/zh-CN/docs/Web/HTML/Element/meta>

\* rel noreferrer

<a rel="noreferrer" href="http://www.ctrip.com.cn/">Cheap Flights</a>

<http://www.w3school.com.cn/html5/att_a_rel.asp>

\* Referrer-Policy

<https://developer.mozilla.org/zh-CN/docs/Web/HTTP/Headers/Referrer-Policy>