[待解决问题 3](#_Toc54253260)

[API 8](#_Toc54253261)

[浏览器进程 10](#_Toc54253262)

[Virtual dom diff算法？ 11](#_Toc54253263)

[箭头函数和普通函数的区别 13](#_Toc54253264)

[原生JS实现str.indexOf方法 17](#_Toc54253265)

[基本类型包装成对象 18](#_Toc54253266)

[JS中的垃圾回收 19](#_Toc54253267)

[Null和undefined区别 19](#_Toc54253268)

[获取异步函数的返回值 19](#_Toc54253269)

[对象的键名都要转化为字符串 20](#_Toc54253270)

[js Symbol数据类型 21](#_Toc54253271)

[常见题目 23](#_Toc54253272)

[编写原生代码实现图片懒加载 28](#_Toc54253273)

[实现一个$attr(name,value)函数返回具有属性为name，值为value的元素的集合， 28](#_Toc54253274)

[将数组扁平化，并去除其中重复部分 30](#_Toc54253275)

[对象 自己编写一个\_new程序 33](#_Toc54253276)

[递归 34](#_Toc54253277)

[var 和 let的区别 39](#_Toc54253278)

[排序 42](#_Toc54253279)

[函数柯理化 45](#_Toc54253280)

[原生实现call apply 46](#_Toc54253281)

[跨域 47](#_Toc54253282)

[react&&Vue 区别 53](#_Toc54253283)

[CMD、AMD、common.js、ES6规范 54](#_Toc54253284)

[深浅克隆 55](#_Toc54253285)

[Cookie && Session 57](#_Toc54253286)

[LocalStorage与Cookie的异同 62](#_Toc54253287)

[http协议 63](#_Toc54253288)

[http2新特性 65](#_Toc54253289)

[冒泡与捕获 67](#_Toc54253290)

[页面性能优化的方法 68](#_Toc54253291)

[浏览器缓存 73](#_Toc54253292)

[http状态码 75](#_Toc54253293)

[原生JS实现ajax 77](#_Toc54253294)

[HTTPS 保证安全性 79](#_Toc54253295)

[哈希算法 81](#_Toc54253296)

[Token登录 81](#_Toc54253297)

[浏览器渲染过程 82](#_Toc54253298)

[什么是 Web Worker？ 85](#_Toc54253299)

[Fetch 86](#_Toc54253300)

[HTML标签语义化 87](#_Toc54253301)

[Doctype作用 87](#_Toc54253302)

[Cookie如何防止XSS攻击 88](#_Toc54253303)

[CRSF 跨站请求伪造 88](#_Toc54253304)

[什么是RESTful 89](#_Toc54253305)

[强缓存和协商缓存 89](#_Toc54253306)

[GET POST区别 92](#_Toc54253307)

[Mouseover 和 mouseenter 区别 93](#_Toc54253308)

[ajax解决浏览器缓存问题 93](#_Toc54253309)

[PWA 94](#_Toc54253310)

[颁发证书的机构 96](#_Toc54253311)

[Promise 96](#_Toc54253312)

[async / await 98](#_Toc54253313)

[Html5新特性 98](#_Toc54253314)

[Http头部 99](#_Toc54253315)

[函数防抖、节流 100](#_Toc54253316)

[CSS 102](#_Toc54253317)

[generate 102](#_Toc54253318)

[最大连续子序列 DP 103](#_Toc54253319)

[最大回文子串 104](#_Toc54253320)

[React 104](#_Toc54253321)

[DTD是什么 104](#_Toc54253322)

[项目页面优化的实践 105](#_Toc54253323)

[自我介绍 105](#_Toc54253324)

[事件代理 106](#_Toc54253325)

[严格模式 106](#_Toc54253326)

[敏捷开发 106](#_Toc54253327)

[Rgba&&opicity 107](#_Toc54253328)

[Dom && css 107](#_Toc54253329)

[盒模型 108](#_Toc54253330)

[Typeof 原理 109](#_Toc54253331)

[Instanceof 109](#_Toc54253332)

[js如何完整的显示较长的数字 109](#_Toc54253333)

[Css选择器 110](#_Toc54253334)

[Padding margin设置百分比基于什么设置 111](#_Toc54253335)

[移动端兼容方案 111](#_Toc54253336)

[BFC 112](#_Toc54253337)

[Get&post区别 113](#_Toc54253338)

[Viewport 114](#_Toc54253339)

[Inline-block 和 block的区别 115](#_Toc54253340)

[设计模式 115](#_Toc54253341)

[CSS3 115](#_Toc54253342)

[字符串方法 117](#_Toc54253343)

[看代码说结果 118](#_Toc54253344)

# 待解决问题

用React实现一个文件夹组件,而且并不知道有多少级子组件,可以用递归

 很重要!!!!!!!!!

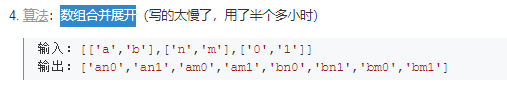


 done

 done

规范

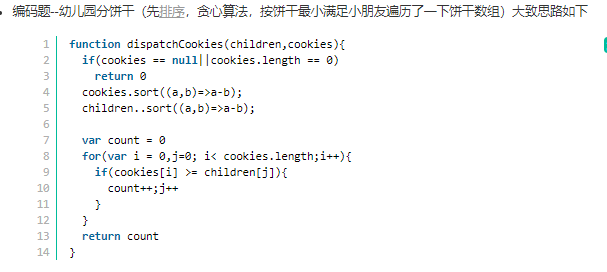
Es6

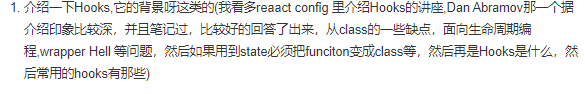


 done 用split



数组第k大的元素







~~轮播~~ 图片懒加载 ~~函数节流~~

~~Js看代码说结果题目~~

~~Obj.hasOwnPropty~~ 只能查找自己身上的属性，不能原型链查找

Promise实现并发控制 ，最多两个；

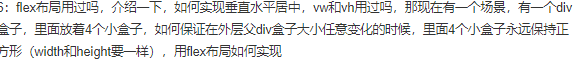
~~大数相加~~ 递归

~~https详细过程~~ 为什么安全

到这里 全部完成=>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

~~Float position的区别~~  float 浮动(脱离文档流，不脱离文本流) position:脱离文档流

~~Flex~~



~~Ajax 状态~~

~~Css3~~

~~Rem,em原理 怎么使用~~

~~发布订阅~~

~~Router原理~~

~~函数柯理化~~

力扣862

~~Bfc详细~~

~~手写jsonP~~

~~Osi 七层 五层协议~~

~~Tcp udp区别，~~tcp如何保证安全：1、停止等待协议 2、连续ARQ协议

~~Dns查询过程使用的协议 tcp udp~~

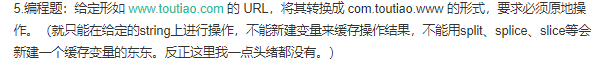
~~移动端适配方案~~

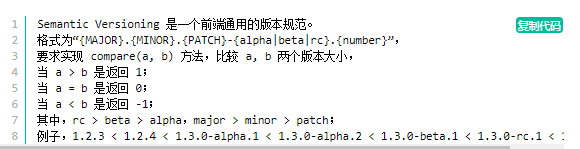
~~二叉树路径和为n的路径~~

~~手写Js render函数~~

~~进程如何通信~~

~~Get post的区别~~





~~Event.preventDefault 阻止默认行为~~

~~两个队列实现栈~~

Xss crsf

~~从根节点到叶子节点所有路径~~

~~React fiber原理~~  类似于时分多路复用

~~React hooks原理； 自定义hook~~

~~Ts js区别~~

~~Html header里面包含了哪些~~  viewport charset http-equive

Webpack sourceMap

~~http 和 websoket的联系~~

Redux 和 react-redux 看todo代码

~~两个同域页面之间的通信~~ => webstorage cookie

~~transition和animation的区别~~

~~react diff算法 生命周期 实例化~~

~~redux~~

~~Webpack打包过程，打包优化，什么时候打包成多个文件或者一个文件~~

**Webpack打包工具有哪些 怎么配置**

~~预加载和懒加载~~；

~~BFC（block formatting context）~~

最大连续子序列 动态规划

~~Object.create 第二个参数，Object.create(null) 会怎样。~~

水平居中和垂直居中

position有什么属性 **sticky定位**

~~fetch~~



# API

**~监听浏览器刷新事件：**

window.onbeforeunload

1. valueOf()会把数据类型转换成原始类型
2. toString()会把数据类型转换成string类型

需要注意的是，这两个方法在不同使用场景会有不同的优先级：

正常情况下，优先调用toString()，有运算操作符的情况下valueOf()的优先级高于toString()，当调用valueOf()方法无法运算后还是会再调用toString()方法



# 浏览器进程

浏览器多进程：

1. browser主进程
2. 第三方插件
3. GPU进程
4. 渲染进程 => 浏览器内核

GUI渲染线程

JS引擎线程（永远只有一个）

事件触发线程 属于bom

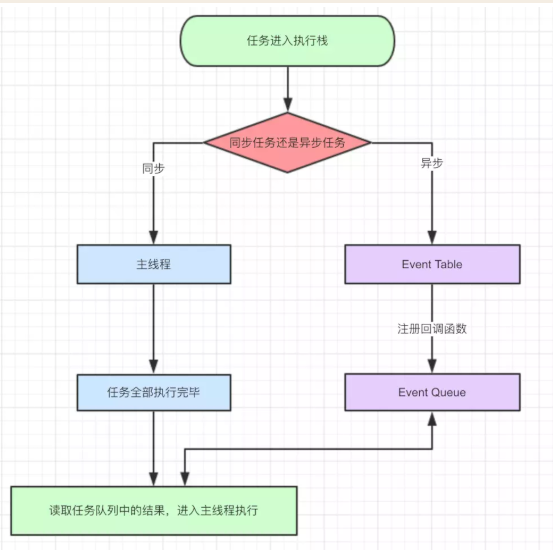
定时器线程

异步http请求线程

--浏览器内核，内部是多线程的

如果DOM一变化，界面就立刻重新渲染，效率必然很低，所以浏览器的机制规定界面渲染线程和主线程是互斥的，主线程执行任务时，浏览器渲染线程处于挂起状态。

同步任务进入执行栈 异步任务进入执行队列



# Virtual dom diff算法？

**但是在前端当中，你很少会跨越层级地移动DOM元素。所以 Virtual DOM 只会对同一个层级的元素进行对比：**

Virtual DOM是React中的一个很重要的概念，在日常开发中，前端工程师们需要将后台的数据呈现到界面中，同时要能对用户的操作提供反馈，作用到UI上…… 这些都离不开DOM操作。但是我们知道，**频繁的DOM操作会造成极大的资源浪费，也通常是性能瓶颈的原因。于是React 引入了Virtual DOM。Virtual DOM的核心就是计算比较改变前后的DOM区别，然后用最少的DOM操作语句对DOM进行操作**。

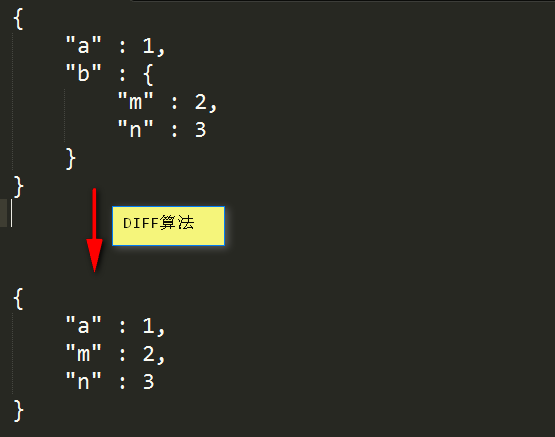
比如说要进行这样的DOM转换：



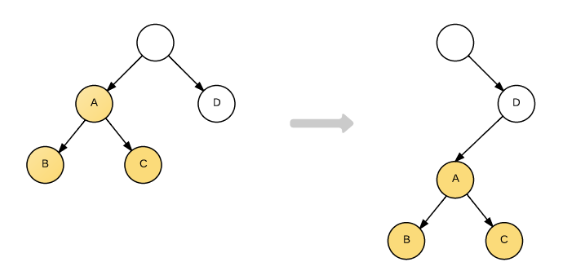
不用Virtual DOM技术：删除4个li，插入5个li。

使用Virtual DOM技术：通过计算，我们只需要改变4个li的innerHTML，并且插入一个li即可。

Virtual DOM技术使用了DIFF算法，DIFF算法是一个比较计算层次结构区别的算法，现在主要用来计算DOM之间的差异。

 用DIFF算法可以计算两个JSON结构之间的差别。

现在DIFF算法为Virtual DOM技术服务，用来比较DOM结构的差异：



如果不使用Virtual DOM技术，此时要删除3个节点，插入3个节点。使用Virtual DOM技术仅仅需要移动节点即可。

# 箭头函数和普通函数的区别

1. 箭头函数更加简洁

a = x=>y=>x+y;  
*function* a(x){  
 *return function* (y) {  
 *return* x+y  
 }  
}  
console.log(a(1)(2))//3

1. 箭头函数有自己的this,它的this继承与函数所处上下文中的this（使用，apply,call等改变this指向的函数没用，this依旧是定义函数时继承的this）

document.body.onclick = ()=>{  
 console.log(*this*)*//window*};  
  
document.body.onclick = *function* () {  
 *//this => body* arr.sort(*function* (a,b) {  
 *// => window 因为这不是sort函数 而是sort里的回调函数（回调函数this一般都是window）   
 return* a-b;  
 })  
};

3、箭头函数没有arguments(类数组)参数，只能基于...arg获取传递参数的集合

*function* a(){  
 console.log(arguments)  
}  
a(1,2,3);*//[Arguments] { '0': 1, '1': 2, '2': 3 }*b = ()=>{  
 console.log(arguments)  
};  
b(1,2,3);*//error*

sum = (...args)=>{  
 *let* sum=0;  
 *for*(*let* i=0; i<args.length; i++){  
 sum+=args[i];  
 }  
 console.log(sum)  
};  
sum(1,2,3)*//6*

1. 箭头函数实现的构造函数不能被new执行，因为

~~箭头函数this不能被修改，及new的时候this不能指向实例

~~·箭头函数没有prototype

## 用回调函数实现foreach

*function* each(callback) {  
 *for*(*let* i=0; i<*this*.length; i++){  
 *let* flag = callback.call(*this*[i],i);  
 *//另外还添加了 接收回调函数返回的结果 如果是false 就结束循环  
 if* (flag ===*false*){  
 *break*;  
 }  
 }  
}  
Array.prototype.each = each;  
  
[1,2,3].each(*function* (item,index) {  
 console.log(item)  
});*////1 0 2 1 3 2*[1,2,3].each(*function* (item,index) {  
 *if* (index >1){  
 *return false* }   
 *else* {  
 console.log(item)  
 }  
});*//1,2*

## == 和 ===的区别

**==先转换类型再比较，===先判断类型，如果不是同一类型直接为false。**

**alert(1 == “1”); // true**

**alert(1 === “1”); // false**

**1、{}=={}，两个对象比较，比较的是堆内存的地址**

**2、null==undefined 相等； null===undefined不相等**

**要比较相等性之前，不能将 null 和 undefined 转换成其他任何值**，并且规定null 和 undefined 是相等的。

null 和 undefined都代表着无效的值。

**3、NaN和谁都不相等**  
**4、如果一个是字符串，一个是数值，把字符串转换成数值再进行比较。**

**5**、**如果任一值是 true，把它转换成 1 再比较；如果任一值是 false，把它转换成 0 再比较。**

**6、如果一个是对象，另一个是数值或字符串，把对象转换成基础类型的值再比较。对象转换成基础类型**，利用它的toString或者valueOf方法。js核 心内置类，会尝试valueOf先于toString；例外的是Date，Date利用的是toString转换。  
任何其他组合，都[不相等]。

## 如何把一个字符串大小写取反（大写变小写，小写变大写），例如‘AbC’变‘aBc’

# 原生JS实现str.indexOf方法

*function* myIndexOf(testStr) {

//不能用/testStr/ 因为/t是转义字符  
 *let* reg = *new* RegExp(testStr),  
 res = reg.exec(*this*);  
 *return* res === *null*? -1:res.index;  
}  
  
String.prototype.myIndexOf = myIndexOf;

# 基本类型包装成对象

number bollean string null undefined

**以number类型为例**

**var** a = 12.3;

console.log(a.toFixed(3)); *// 输出"12.300"*

a.foo = 'bar';

console.log(a.foo); *// 输出undefined*

JavaScript引擎内部在处理对某个基本类型 a进行形如a.sth的操作时，会在内部临时创建一个对应的包装类型(对数字类型来说就是Number类型)的临时对象，并把对基本类型的操作代理到对这个临时对象身上，使得对基本类型的属性访问看起来像对象一样。但是在操作完成后，临时对象就扔掉了，下次再访问时，会重新建立临时对象，当然对之前的临时对象的修改都不会有效了。

# JS中的垃圾回收

# Null和undefined区别

1. **含义不同**

undefined：表示使用var声明变量但未对其加以初始化时，这个变量的值就是undefined

null：是一个空对象指针，表示准备用来保存对象，还没有真正保存对象的值，如果定义的变量准备在将来用于保存对象，应该将该变量初始化为null。

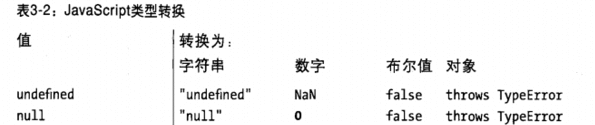
1. **类型不同**

未初始化定义的值用typeof检测出来是"undefined"(字符串)，而null值用typeof检测出来是"object"（字符串）

console.log(typeOf undefined);//undefined

console.log(typeOfnull);//object

**3、转化为值时不一样**



# 获取异步函数的返回值

1、用回调函数

*function* getsomething(callback) {  
 *var* i = 0;  
 setTimeout(*function* () {  
 i = 2;  
 callback(2)  
 },100)  
}  
*function* test(i) {  
 console.log(i\*i) ;  
}  
getsomething(test);*//4*

# 对象的键名都要转化为字符串

*var* obj={100:"aaa"};  
console.log(obj[100]);*//aaa*console.log(obj['100']);*//aaa*

*var* obj={100:"aaa",'100':'bbb'};  
console.log(obj[100]);*//bbb*console.log(obj['100']);*//bbb*

# js Symbol数据类型

ES6引入的一种特殊的数据类型

Symbol 本质上是一种唯一标识符，可用作对象的唯一属性名，这样其他人就不会改写或覆盖你设置的属性值。

**声明方法**：

let id = Symbol("id“);

**Symbol 数据类型的特点是唯一性，即使是用同一个变量生成的值也不相等。**

let id1 = Symbol('id');

let id2 = Symbol('id');

console.log(id1 == id2); //false

**Symbol 数据类型的另一特点是隐藏性，for···in，object.keys() 不能访问**

*let* id = Symbol("id");  
*let* obj = {  
 [id]:'symbol'  
};  
*for*(*let* option *in* obj){  
 console.log(obj[option]); *//空*}

**但是也有能够访问的方法：Object.getOwnPropertySymbols**

**Object.getOwnPropertySymbols 方法会返回一个数组，成员是当前对象的所有用作属性名的 Symbol 值。**

*let* id = Symbol("id");  
*let* obj = {  
 [id]:'symbol'  
};  
*let* array = Object.getOwnPropertySymbols(obj);  
console.log(array); *//[Symbol(id)]*console.log(obj[array[0]]); *//'symbol'*

**虽然这样保证了Symbol的唯一性，但我们不排除希望能够多次使用同一个symbol值的情况。**

**为此，官方提供了全局注册并登记的方法：Symbol.for()**

*let* name1 = Symbol.for('name'); *//检测到未创建后新建  
let* name2 = Symbol.for('name'); *//检测到已创建后返回*console.log(name1 === name2); *// true*

**通过这种方法就可以通过参数值获取到全局的symbol对象了，反之，能不能通过symbol对象获取到参数值呢？**

**是可以的 ，通过Symbol.keyFor()**

*let* name1 = Symbol.for('name');  
*let* name2 = Symbol.for('name');  
console.log(Symbol.keyFor(name1)); *// 'name'*console.log(Symbol.keyFor(name2)); *// 'name'*

# 常见题目

1. NaN == NaN //false
2. 基本的数据类型: Number 、Boolean、Undefined、String、Null
3. typeof 的返回结果 undefined Boolean string number object function symble
4. =>>>>>>>>>>>>>>>>>>>>>>>>

*var* arr = [11,22];  
*var* brr = {11:22};  
*var* obj = {}  
obj[arr] = 'a';  
obj[brr] = 'b'  
console.log(obj)*//{ '11,22': 'a', '[object Object]': 'b' }*

1. 因为对象的属性名最终都会转化为字符串类型即**obj[1]和obj[‘1’]是一样的**，遇数组和对象都会调用tostring方法，

而数组的tostring方法在Array.prototype上，作用是直接转化为字符串，

而Object的tostring方法在Object.prototype上，作用是检测数据类型默认情况下，toString() 方法被每个 Object 对象继承。如果此方法在自定义对象中未被覆盖，toString() 返回 "[object *type*]"，其中 type 是对象的类型。以下代码说明了这一点：

var o = new Object();

o.toString(); // returns [object Object]

1. **=>>>>>>>>>>>>>>>>>>>>>>>>**

*function* Foo() {  
 Foo.a = *function* () {  
 console.log(1);  
 }  
   
 *this*.a = *function* () {  
 console.log(2)  
 }  
}  
//把Foo当做类，在原型上设置公有的属性方法Foo.prototype.a = *function* () {  
 console.log(3)  
};  
//把Foo当做普通类，设置自己的a属性方法Foo.a = *function* () {  
 console.log(4)  
};  
  
  
Foo.a(); //4 *let* obj = *new* Foo(); //此时执行Foo()函数， Foo自己的私有属性a被改写 Foo.a=>1obj.a();*//2* 私有属性中有a 所以不会在prototype上查找Foo.a();*//1*

1. **=>>>>>>>>>>>>>>>>>>>>>>>>**

*var* a = 10;  
(*function* a() {  
 a = 20;  
 console.log(a)  
})();*// => [Function: a]*//匿名函数给名字执行时 是在该函数的作用域声明变量a指向一个函数，该名字不能再被修改, 但若重新var a=10; a最终为10console.log(a); *// => 10*//直接在全局作用域查找 a=10;

// 要使第一个a输出为20 这个a必须为私有变量 *var* a = 10;  
(*function* a(a) {  
 a = 20;*//存在于函数局部作用域* console.log(a)  
})();*// => 20*//使a 为私有变量的方法有 1、在函数中var一个,var a=20; 2、如上，传入形参，形参也为私有变量console.log(a); *// => 10*

**6、=>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>.**

obj = {  
 2:3,  
 3:4,  
 length:2,  
 push:Array.prototype.push  
};  
obj.push(1);  
obj.push(2);  
console.log(obj)

这个题目，把数组的push方法挂载到了对象上。对于这种题目，我们首先得知道push方法是怎么实现的，如下：

Array.prototype.push = *function* (value) {  
 *this*[*this*.length] = value;  
 *// 注意 这里有一个隐式的this.length++的步骤  
 // 具体源码不知道 但肯定是this身上的length属性+1；*};

所以，结果如下

obj.push(1); *//  
//this现在指向obj 即obj[obj.length]=1 => obj[2]=1 => this.length=3*obj.push(2);  
*//obj[obj.length]=2 => obj[3]=2 => this.length=4*console.log(obj)  
*//{ '2': 1, '3': 2, length: 4, push: [Function: push] }*

1. **=========================================**🡺

**题目：将{1: 22, 2: 2, 5: 6,}变成 [22, 2, *null*, *null*, 6, *null*, *null*, *null*, *null*, *null*, *null*, *null*]**

**注：**obj.keys 可以得到一个key的数组

*function* monthHandler(obj) {  
 *//创建一个长度为12 内容全空的数组  
 let* arr = *new* Array(12).fill(*null*);  
 *for* (key *in* obj) {  
 arr[key - 1] = obj[key]  
 }  
  
 *return* arr  
}

# 编写原生代码实现图片懒加载

1. 前端性能优化的重要方案，加快渲染速度，

只有滑动到某个区域，我们才加载真实的图片，节省流量

1. **处理方案**

**把所有需要延时加载的图片用一个盒子包起来，设置宽高默认值来占位置。**

**开始让所有img的src为空，把真实图片地址放到img自定义属性上，让img隐藏**

**等到所有其他资源加载完成后在加载其他图片**

**对于很多图片，需要当页面滚动时，当前图片区域完全显示出来后再加载真实图片**

**CSDN 博客 jquery 原生js**

# 实现一个$attr(name,value)函数返回具有属性为name，值为value的元素的集合，

function findAttr(property, value) {

//获得所有元素  
 let elements = document.getElementsByTagName('\*'),  
 arr = [];  
  
 // 类数组的迭代 不能直接使用forEach  
  
 //方法一  
 // [].forEach.call(elements,(item)=>console.log(item))  
  
 //方法二！！  
 elements = Array.from(elements); //把类数组转化为数组  
 elements.forEach((item, index) => {  
 //存储当前元素property对应的属性值  
 let itemValue = item.getAttribute(property);  
// className 可以有多个名字 所以要特殊判断   
 // 正则\b表示 字符边界 \bvalue\b 表示value要是单独的一个单词 不能是 itemvalue这样包含在其他单词里面  
 if (property === 'class') {  
 if (new RegExp('\\b' + value + '\\b').test(itemValue)  
 ) {  
 arr.push(item)  
 }  
 } else {  
 if (itemValue === value) {  
 //获取的值的传递的值相等  
 arr.push(item);  
 }  
 }  
 });  
  
 //方法三 => for循环遍历  
 return arr;  
 }

# 将数组扁平化，并去除其中重复部分

*let* arr = [[1,2,3,4],[2,3,5],[2,5,6],[5,6,4,[4,5,[10,12]]],18,12];  
 *//flat将数组扁平化 参数为1表示一层 Infinity表示到最深层  
 var* newarr = arr.flat(Infinity)  
  
 *//es6方法Set 数组去重 返回一个Set类型 所以还得转化为数组* newarr = *new* Set(newarr);  
  
 *//方法一  
 // newarr = Array.from(newarr)  
 //方法二* newarr = [...newarr];  
  
*// 然后进行排序 sort* newarr.sort((a,b)=>a-b);*//a-b为升序* console.log(newarr)

#### **数组扁平化**

1**、tostring split map**

*let* arr = [[1,2,3,4],[2,3,5],[2,5,6],[5,6,4,[4,5,[10,12]]],18,12];  
arr = arr.toString().split(',').map(*function* (item) {  
 *return* Number(item)  
})

结果=> [1, 2, 3, 4, 2, 3, 5, 2, 5, 6, 5, 6, 4, 4, 5, 10, 12, 18, 12]

2、**some isArray**

CSDN

1. **递归**

*function* myflat() {  
 *var* aarr = [];  
 *var* fn = (arr)=>{  
 *for* (*let* i = 0; i <arr.length ; i++) {  
 *if* (Array.isArray(arr[i])){  
 fn(arr[i]);  
 *continue* }  
 aarr.push(arr[i])  
 }  
 }  
 fn(*this*);  
 *return* aarr;  
}  
Array.prototype.myflat = myflat;

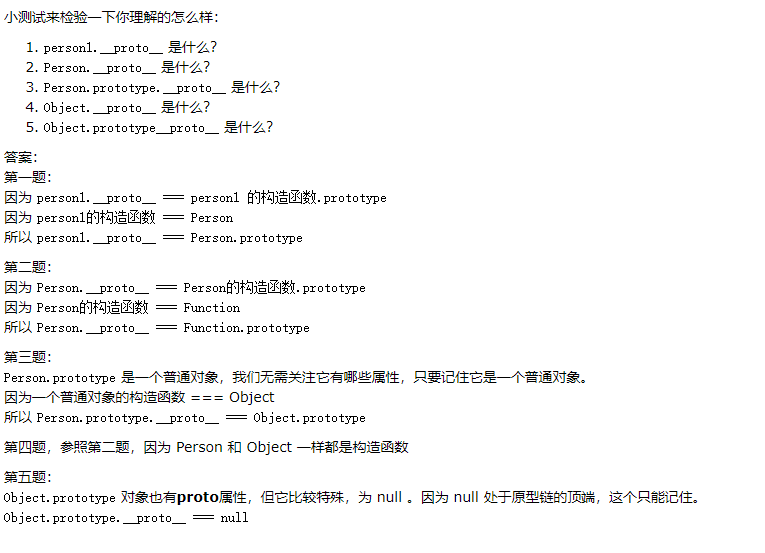
#### ****数组去重有几种方法 ：****

1. **最基础的两重for循环**
2. **Foreach+indexOf**
3. **new Set()**

# 对象 自己编写一个\_new程序

function定义的对象与使用new操作符生成的对象之间有一个重要的区别，这个区别就是function定义的对象有一个prototype属性，使用new生成的对象就没有这个prototype属性，存在\_\_proto\_\_

null 处于原型链的顶端



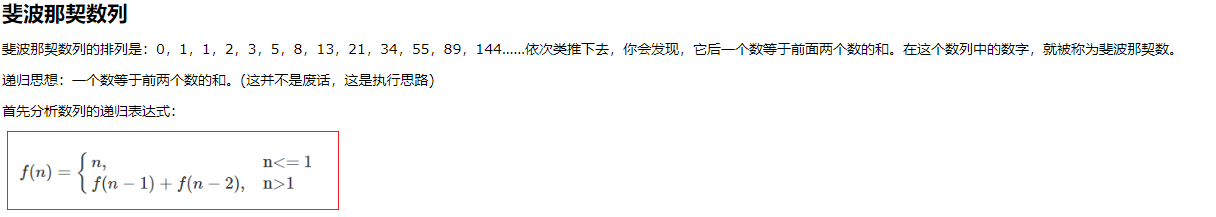
new一个对象的时候，会发生四件事：

1. 创建一个空对象，并将其挂载到原型链上
2. 让this指向构造函数
3. 执行构造函数 语句
4. 返回这个对象

function \_new(fn,...args) {  
 // var obj = {};  
 // obj.\_\_proto\_\_ = fn.prototype;  
 //以上两句话作用是： 创建一个空对象，让他的原型链\_\_proto\_\_指向fn.prototype(作为fn的一个实例)  
 //可用以下语句代替  
 var obj = Object.create(fn.prototype);//Object.create(a)作用是：创建一个空对象，并让空对象的\_\_proto\_\_指向a;  
   
 fn.call(obj,...args);  
 return obj;  
}

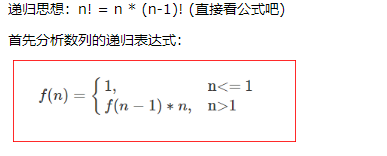
# 递归

**斐波那契数列**



*function* fn(n) {  
 *if* (n == 0) {  
 *return* 0  
 } *else if* (n == 1) {  
 *return* 1  
 } *else* {  
 *return* fn(n - 1) + fn(n - 2)  
  
 }}

**阶乘**

****

*function* fn(n) {  
 *if* (n<=1){  
 *return* 1  
 }  
 *else* {  
 *return* fn(n-1)\*n  
 }  
}

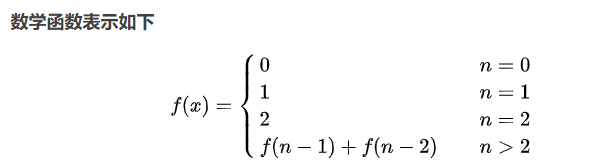
**青蛙跳台阶的问题**

##### 题目：一只青蛙一次可以跳上1级台阶，也可以跳上2级台阶。求该青蛙跳上一个n级的台阶总共有多少种跳法？

 如果只有1级台阶，那显然只有一种跳法

 如果有2级台阶，那么就有2种跳法，一种是分2次跳。每次跳1级，另一种就是一次跳2级

 如果台阶级数大于2，设为n的话，这时我们把n级台阶时的跳法看成n的函数，记为f(n),第一次跳的时候有2种不同的选择：**一是第一次跳一级，此时跳法的数目等于后面剩下的n-1级台阶的跳法数目，即为f(n-1),二是第一次跳二级，此时跳法的数目等于后面剩下的n-2级台阶的跳法数目，即为f(n-2),**因此n级台阶的不同跳法的总数为f(n-1)+ f(n-2)，不难看出就是斐波那契数列



*function* step(n) {  
 *if* (n==0){  
 *return* 1  
 }  
 *if* (n==1){  
 *return* 1  
 } *if* (n==2){  
 *return* 2  
 }  
 *return* step(n-1)+step(n-2)  
}

**矩形覆盖问题**

**问题：我们可以用2\*1的小矩形横着或者竖着去覆盖更大的矩形。请问用2\*1的小矩形无重叠地覆盖一个2\*n的大矩形，总共有多少种方法？这也是斐波那契数列问题**

若最左边竖着放，占了一列,还剩n-1列，剩下为F(n-1)种，

若横着放，占了两列，还剩n-2列，剩下为F(n-2)种。仍然是斐波那契数列

*function* step(n) {  
 *if* (n==1){  
 *return* 1  
 }  
 *if* (n==2){  
 *return* 2  
 }  
 *return* step(n-1)+step(n-2)  
}

**旋转数组**

**问题：**比如[1,2,3,4]旋转一次输出[2,3,4,1]; 旋转两次输出[3,4,1,2]

*function* rotate(arr,k) {  
 *if* (k<=0||k%arr.length==0){  
 *return* arr  
 }  
 *if* (k>arr.length){  
 k=k%arr.length  
 }  
 *var* newarr = arr.slice(1);  
 newarr.push(arr[0]);  
 k--;  
 *return* rotate(newarr,k)  
}

# **var 和 let的区别**

#### var声明的变量会挂载在window上，而let和const声明的变量不会：

#### let和const声明形成块作用域

#### 同一作用域下let和const不能声明同名变量，而var可以

#### 暂存死区

[复制代码](javascript:void(0);)

var a = 100;  
  
if(1){  
 a = 10;  
 //在当前块作用域中存在a使用let/const声明的情况下，给a赋值10时，只会在当前作用域找变量a，  
 // 而这时，还未到声明时候，所以控制台Error:a is not defined  
 let a = 1;  
}

var 有变量提升 ，若在自定义函数里面，会提升到自定义函数最顶部。若在全局函数，会提升到全局最顶部，两次定义相同的变量，后面的会把前面的替代

*function* f1() {  
 *var* n = 5;  
 *if* (*true*) {  
 *var* n = 10;  
 }  
 console.log(n); *// 10*}

用let声明的变量，不存在变量提升。而且要求必须 等let声明语句执行完之后，变量才能使用，不然会报Uncaught ReferenceError错误。

function f1() {

let n = 5;

if (true) {

let n = 10;

}

console.log(n); // 5

}

**let配合for的独特应用**

****

**如果不把var 转为let 还可以：**

1. **闭包**

*for*(*var* i=0; i<10; i++){  
~*function*(num){  
 setTimeout(*function* () {  
 console.log(num)  
 },1000)  
 }(i);  
}// => 0,1,2,3,4,5,6,7,8,9

*for*(*let* i=0; i<10; i++){  
 setTimeout((*function* (num) {  
 *return function* () {  
 console.log(num)  
 }  
 })(i),1000)  
}

1. bind预处理机制：在循环的时候就把每次函数需要输出的结果预先传给函数

*function* fn(i) {  
 console.log(i)  
}  
  
*for*(*let* i=0; i<10; i++){  
 setTimeout(fn.bind(*null*,i),1000)

# 排序

**冒泡排序**

**让数组中的当前项和后一项比较，如果当前项比后一项大，则交换**

*function* buuble(arr){  
 *for* (*let* i = 0; i <arr.length-1 ; i++) {  
 *var* t=0;  
 *for*(*let* j=0; j<arr.length-i-1; j++){  
 *if* (arr[j]>arr[j+1]){  
 t = arr[j+1];  
 arr[j+1] = arr[j];  
 arr[j] = t;  
 }  
 }  
 }  
 *return* arr;  
}

**插入排序**

**构建一个新数组，从arr中一项一项的与新数组中的每一项比较**

*function* insert(arr) {  
 *var* newarr = [];  
 newarr.push(arr[0]);  
 *//定义一共比多少轮  
 for*(*let* i=1; i<arr.length; i++){  
 *//和新数组的每一项都要比  
 for*(*let* j=0; j<newarr.length; j++){  
 *if* (arr[i]<newarr[j]) {  
 newarr.splice(j, 0, arr[i]);  
 *break*;  
 }  
 *if* (j === newarr.length-1){  
 newarr.push(arr[i]);  
 }  
 }  
 }  
 *return* newarr;  
}

**快速排序**

**平均复杂度：O(nlog2n)**

**最坏 ： O（n^2）**

*function* swift(arr) {  
 *//结束递归  
 if* (arr.length<=1){  
 *return* arr;  
 }  
   
 *var* middleNum = arr[parseInt(arr.length/2)];  
 *var* beginArr = [];  
 *var* endArr = [];  
 *for*(*let* i=0; i<arr.length; i++){  
 *if* (i==parseInt(arr.length/2)) {  
 *continue* }  
 arr[i]<middleNum? beginArr.push(arr[i]):endArr.push(arr[i]);  
 }  
 *return* swift(beginArr).concat(middleNum,swift(endArr))  
}

# 函数柯理化

一种预先处理思想，利用闭包机制

最简单的柯理化函数

Fn第一次执行的时候，返回一个匿名函数，但该匿名函数未执行，所以作用域没有销毁，x保存起来，当fn第二次执行的时候，向上层函数环境中查找x，即形成了函数的预处理

*function* fn(x) {  
 *return function* (y) {  
 *return* x+y  
 }  
}

**实现bind**

!*function* () {  
 *//obj表示要改变的this指向(默认为window)  
 //arg其余要传入的参数  
 function* mybind(obj=window,...outerArgs) {  
 *let* self = *this*;  
 *return function* (...innerArgs) {  
 self.call(obj,...outerArgs.concat(innerArgs));  
 }  
 }  
 Function.prototype.mybind = mybind;  
}();

# 原生实现call apply

实现call

*function* mycall(obj=window) {  
 *let* arr = []  
 *for*(*let* i=1; i<arguments.length; i++){  
 arr.push(arguments[i]);  
 }  
 *if* (obj === *null*){  
 *this* (...arr);  
 *return* }  
 obj.fn = *this*;  
  
 obj.fn(...arr);  
 *delete* obj.fn;  
}  
  
Function.prototype.mycall = mycall;  
*function* fn(a,b) {  
 console.log(a+b)  
};

实现apply

将参数列表变为参数数组即可

# 跨域

浏览器同源策略：协议名、端口号、域名必须一致；

当页面在执行一个脚本时,会检查访问的资源是否同源,如果不是,就会报错。

　注意：对于像 img、iframe、script 等标签的 src 属性是特例，它们是可以访问非同源网站的资源的。

JSONP跨域

只支持GET请求

*// 第一步创建script标签  
var* script = document.createElement('script');  
*//第二补 将src变为要访问的网站*script.src = 'testJS.js'  
*// 第三步 将script添加入页面*document.body.appendChild(script);  
*// 定义函数操作获取的数据  
function* response(data) {  
 console.log(data)  
}

**test.js**

response(  
 {  
 a:1  
 }  
)

script标签执行所访问文件的内容，比如test.js。当把script加入页面后，会执行response({a:1})函数

JSONP的缺点：只支持GET请求；因为script标签只支持get请求

代理跨域

由服务器出面去获取数据

CORS跨域

<https://blog.csdn.net/badmoonc/article/details/82706246>

跨域资源共享 Cross-Origin Resource Sharing

**优点 可以携带cookie**

实现CORS的关键在于服务器，只要服务器实现CORS接口，就可以实现跨域通信。

CORS分为简单请求 和 非简单请求(需预检)两类

**符合以下条件的，为简单请求**

只要同时满足以下两大条件，就属于简单请求。

（1) 请求方法是以下三种方法之一：

* HEAD
* GET
* POST

（2）HTTP的头信息不超出以下几种字段：

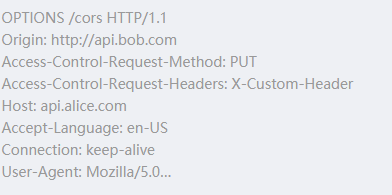
* Accept
* Accept-Language
* Content-Language
* Last-Event-ID
* Content-Type：只限于三个值application/x-www-form-urlencoded、multipart/form-data、text/plain

=>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

非简单请求(预检请求)

非简单请求是那种对服务器有特殊要求的请求

下面是一个非预检请求的http头部



"预检"请求用的请求方法是OPTIONS，表示这个请求是用来询问的。头信息里面，关键字段是Origin，表示请求来自哪个源。

除了Origin字段，"预检"请求的头信息包括两个特殊字段。

**（1）Access-Control-Request-Method**

该字段是必须的，用来列出浏览器的CORS请求会用到哪些HTTP方法，上例是PUT。

**（2）Access-Control-Request-Headers**

该字段是一个逗号分隔的字符串，指定浏览器CORS请求会额外发送的头信息字段，上例是X-Custom-Header。

=>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

当发生符合非简单请求（预检请求）的条件时，浏览器会自动先发送一个options请求，如果发现服务器支持该请求，则会将真正的请求发送到后端，反之，如果浏览器发现服务端并不支持该请求，则会在控制台抛出错误，如下：

https://img-blog.csdn.net/20180914183824949?watermark/2/text/aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L2JhZG1vb25j/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70

=>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

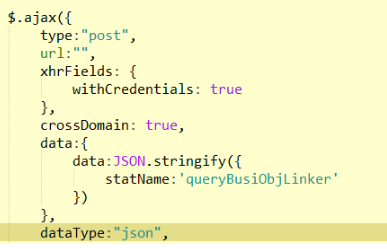
一旦服务器通过了"预检"请求，以后每次浏览器正常的CORS请求，就都跟简单请求一样，会有一个Origin头信息字段。服务器的回应，也都会有一个Access-Control-Allow-Origin头信息字段。



=>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

配置cookie

总的来说，使用CORS简单请求，非常容易，对于前端来说无需做任何配置，与发送普通ajax请求无异。唯一需要注意的是，需要携带cookie信息时，需要将withCredentials设置为true即可。



Access-Control-Allow-Origin: <origin> | \* // 授权的访问源

Access-Control-Max-Age: <delta-seconds> // 预检授权的有效期，单位：秒

Access-Control-Allow-Credentials: true | false // 是否允许携带Cookie Access-Control-Allow-Methods: <method>[, <method>]\* // 允许的请求动词

Access-Control-Allow-Headers: <field-name>[, <field-name>]\* // 额外允许携带的请求头

Access-Control-Expose-Headers: <field-name>[, <field-name>]\* // 额外允许访问的响应头

[复制代码](javascript:void(0);)

我们看到，**Access-Control-Allow-Credentials** 响应头会使浏览器允许在 Ajax 访问时携带 Cookie。其对应了ASP.NET Core的如下中间件设置：

app.UseCors(builder => builder.AllowAnyOrigin().AllowAnyHeader().AllowAnyMethod().AllowCredentials());

此外在客户端浏览器中，我们仍然需要对 XMLHttpRequest 设置其 **withCredentials** 参数，才能实现携带 Cookie 的目标。示例代码如下：

var xhr = new XMLHttpRequest();

xhr.withCredentials = true;

# react&&Vue 区别

1. **数据绑定: vue实现了数据的双向绑定,react数据流动是单向的**
2. **state对象在react应用中不可变的,需要使用setState方法更新状态;**

**在vue中,state对象不是必须的,数据由data属性在vue对象中管理；**

1. react的思路是all in js，通过js来生成html，所以设计了jsx

vue是把html，css，js组合到一起，用各自的处理方式，vue有单文件组件，可以把html、css、js写到一个文件中，html提供了模板引擎来处理。

1. react是类式的写法，api很少

vue是声明式的写法，通过传入各种options，api和参数都很多

# CMD、AMD、common.js、ES6规范

**AMD**：专门用于浏览器端，模块的加载是异步的；

**依赖前置、异步定义**



**Common.js:同步加载**

浏览器端实现：browserify，他的用途是将前端用到的众多资源（css,img,js,...) 打包成一个js文件的技术。另一个跟browserify比较类似的是webpack

虽然也适用于浏览器端，但是由于是同步的，而浏览器的script等加载天生异步，所以性能不好。

服务器端实现：node.js

module.exports 和 exports 导出模块 require 引入模块

运行时加载 是值的拷贝

**ES6**

在语言标准的层面上，实现了模块功能，而且实现得相当简单，完全可以取代 CommonJS 和 AMD 规范，成为浏览器和服务器通用的模块解决方案。

**export/import对模块进行导出导入的**

**编译时加载模块（静态加载）**

**模块输出的是值的引用，输出接口会动态绑定**

**CMD: common module definition**

同步加载

# 深浅克隆

**浅度克隆：原始类型为值传递，对象类型仍为引用传递。**

**深度克隆：所有元素或属性均完全复制，与原对象完全脱离，也就是说所有对于新对象的修改都不会反映到原对象中。**

*function* deepclone(o) {  
 *let* type = *typeof* o;  
 *if* (type=='number'||type=='bollean'||type=='undefined'||type=='srting'){  
 *return* o;  
 }*else if* (Array.isArray(o)){  
 *let* arr = [];  
 *for*(*let* i=0; i<o.length; i++){  
 *//注意递归 因为数组里面的元素还可能为引用对象，比如[1,2,3,[4,5]]* arr.push(deepclone(o[i]))  
 }  
 *return* arr;  
 } *else if* (type =='object'){  
 *let* obj = {};  
 *for*(*let* key *in* o){  
 obj[key] = deepclone(o[key]);  
 }  
 *return* o  
 }  
}

# Cookie && Session

HTTP协议本身是**无状态**的。什么是无状态呢，**即服务器无法判断用户身份。**Cookie实际上是一小段的文本信息（key-value格式）。客户端向服务器发起请求，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。

## cookie属性项

| **属性项** | **属性项介绍** |
| --- | --- |
| NAME=VALUE | 键值对，可以设置要保存的 Key/Value，注意这里的 NAME 不能和其他属性项的名字一样 |
| Expires | 过期时间，在设置的某个时间点后该 Cookie 就会失效 |
| Domain | 生成该 Cookie 的域名，如 domain="[www.baidu.com](http://www.baidu.com)" |
| Path | 该 Cookie 是在当前的哪个路径下生成的，如 path=/wp-admin/ |
| Secure | 如果设置了这个属性，那么只会在 SSH 连接时才会回传该 Cookie |

当用户第一次访问并登陆一个网站的时候，cookie的设置以及发送会经历以下4个步骤：

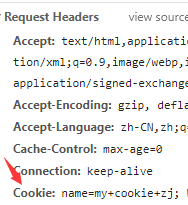
**1 客户端发送一个请求到服务器** --》

**2 服务器发送一个HttpResponse响应到客户端，其中包含Set-Cookie的头部**

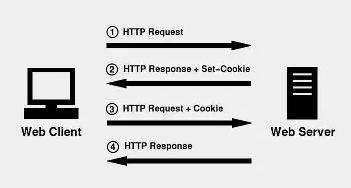
**(key = value)** --》



3 **客户端保存cookie，之后向服务器发送请求时，HttpRequest请求中会包含一个Cookie的头部** --》



**4 服务器返回响应数据**



**但是，cookie是明文传输，这种登录不安全，黑客可以修改本地cookie；**

**所以cookie一般用来本地存储，**

但是有个缺点，本来是前端本地存储的东西，现在需要不断通过http请求在server和client之间传递，效率极低，而且cookie内存很小

**使用 JavaScript 操作cookie** <https://www.runoob.com/js/js-cookies.html>

**所以引入了localstorage**

在HTML5中新加入,这个特性主要是用来作为本地存储来使用的，解决了cookie存储空间不足的问题(**cookie中每条cookie的存储空间为4k**)，localStorage中一般浏览器支持的是5M大小，这个在不同的浏览器中localStorage会有所不同。

1. localStorage 用于长久保存整个网站的数据，保存的数据没有过期时间，直到手动去删除。
2. localStorage 与 sessionStorage 的唯一一点区别就是 localStorage 属于永久性存储，而 sessionStorage 属于当会话结束的时候，sessionStorage 中的键值对会被清空。

Web Storage的API使用方法：<https://www.runoob.com/jsref/prop-win-localstorage.html>

1.添加数据项

localStorage.setItem(key,value)

2.查询已有的数据项

localStorage.getItem(key，value)

3.删除特定的数据项

localStorage.removeItem(key,value)

4.完全删除localStorage目录中的内容

localStorage.clear()

## session

Session与cookie功能效果相同。Session与Cookie的区别在于Session是记录在服务端的，而Cookie是记录在客户端的。

解释session：当访问服务器否个网页的时候，会在服务器端的内存里开辟一块内存，这块内存就叫做session，而这个内存是跟浏览器关联在一起的。这个浏览器指的是浏览器窗口，或者是浏览器的子窗口，意思就是，只允许当前这个session对应的浏览器访问，就算是在同一个机器上新启的浏览器也是无法访问的。而另外一个浏览器也需要记录session的话，就会再启一个属于自己的session

原理：HTTP协议是非连接性的，取完当前浏览器的内容，然后关闭浏览器后，链接就断开了，而没有任何机制去记录取出后的信息。而当需要访问同一个网站的另外一个页面时(就好比如在第一个页面选择购买的商品后，跳转到第二个页面去进行付款)这个时候取出来的信息，就读不出来了。所以必须要有一种机制让页面知道原理页面的session内容。

问题：如何知道浏览器和这个服务器中的session是一一对应的呢？又如何保证不会去访问其它的session呢？

原理解答：就是当访问一个页面的时候给浏览器创建一个独一无二的号码，也给同时创建的session赋予同样的号码。这样就可以在打开同一个网站的第二个页面时获取到第一个页面中session保留下来的对应信息（理解：当访问第二个页面时将号码同时传递到第二个页面。找到对应的session。）。这个号码也叫sessionID，session的ID号码，session的独一无二号码

# LocalStorage与Cookie的异同

**1>相同之处：**

1.二者都可能会被禁用；

2.存储的数据量都有最大限制:,Cookie内存为4kb；Web Storage是5MB或者更少一些；

3.用户随时可以删除或创建/修改存储目录中的内容；

**差别:**

1.通过Web Storage API保存的数据只能在客户端查询，不能被服务器端查询；

2.存储的内容不会随着每个请求“来来回回”传递；

3.除了sessionStorage（它会在会话结束时删除自己保存的数据）以外，不能明确指定数据的过期时间；

# http协议

一个基于tcp/ip协议来传输数据的协议；非常的简便、快速；并现在已发展出http2，https;一般浏览器作为http客户端用过url向服务器发送请求，服务器收到请求后做出相应

1. HTTP允许传输任意类型的数据对象.正在传输的类型由Content-Type加以标记.
2. 请求方法常用的有GET、HEAD、POST。每种方法规定了客户与服务器联系的类型不同。
3. **无连接**；无连接的含义是**限制每次连接只处理一个请求**.服务器处理完客户端的请求,然后响应,并收到应答之后,就断开连接.这种方式可以节省传输时间.
4. **无状态；**HTTP协议是无状态协议.无状态是指协议 对于事务处理没有记忆能力. 为了解决HTTP的无状态特性,出现了Cookie和Session技术.



## 方法

**1、GET方法：**对这个资源的查操作。

**2、DELETE方法：**对这个资源的删操作。但要注意：客户端无法保证删除操作一定会被执行，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。

1. **HEAD方法：**与GET方法的行为很类似，但服务器在响应中只返回实体的主体部分。这就允许客户端在未获取实际资源的情况下，对资源的首部进行检查，使用HEAD，我们可以更高效的完成以下工作：

在不获取资源的情况下，了解资源的一些信息，比如资源类型；

通过查看响应中的状态码，可以确定资源是否存在；

通过查看首部，测试资源是否被修改；

**4、PUT请求：**如果两个请求相同，后一个请求会把第一个请求覆盖掉。（所以PUT用来改资源）

PUT和POS都有更改指定URI的语义.但PUT被定义为idempotent的方法，POST则不是.idempotent的方法:如果一个方法重复执行多次，产生的效果是一样的，那就是idempotent的

1. **OPTIONS方法：**用于获取当前服务器所支持的方法。若请求成功，则它会在HTTP头中包含一个名为“Allow”的头，值是所支持的方法，如“GET, POST”。
2. **TRACE方法：**会在目的服务器端发起一个**“回环”诊断**，我们都知道，客户端在发起一个请求时，这个请求可能要穿过防火墙、代理、网关、或者其它的一些应用程序。这中间的每个节点都可能会修改原始的HTTP请求，TRACE方法允许客户端在最终将请求发送服务器时，它变成了什么样子。由于有一个“回环”诊断，在请求最终到达服务器时，服务器会弹回一条TRACE响应，并在响应主体中携带它收到的原始请求报文的最终模样。这样客户端就可以查看HTTP请求报文在发送的途中，是否被修改过了。

# http2新特性

1. 使用了[HPACK](https://link.jianshu.com/?t=https%3A%2F%2Fhttpwg.org%2Fspecs%2Frfc7541.html)算法进行压缩**头部域，来减小头部的体积,**
2. 二进制协议

H**TTP/2之前的协议都是基于ASCII码**，好处是可读性好，容易上手。其缺点是可选的空格以及多变的终止符给识别帧造成了一些困难。**采用二进制协议可以使得帧的识别更简单，并且传输信息更高效**。其缺点是不便于调试，这就需要我们使用相应的工具来理解二进制的内容。

HTTP/2完全采用二进制协议，头信息和数据体都是二进制的，统称为**帧**(frame)。

1. 通过多路复用实现keep-alive;

在HTTP/1.1中，如果在一个TCP连接内同时发起多个请求，每个消息可以被拆成互不依赖的帧并且各帧之间交错发送，然后在另一端重新把帧组装起来。这个特性就叫做多路复用。

4、HTTP/2新增的另一个强大的功能是允许服务器除了可以响应客户端请求，还可以向客户端推送额外的资源。

通常当我们请求一个网页时，客户端解析HTML源码，发现有js或css等其他静态资源，然后再发起请求下载静态资源。而实际上，当客户端请求网页后，服务器完全可以预判客户端接下来要请求相关的静态资源，那为什么不让服务器提前推送这些资源，从而减少额外的延迟时间呢？HTTP/2为此提出了服务器推送机制

# 冒泡与捕获

在内联模式中，一个元素绑定多个事件时，只会执行第一个事件，而在外联模式中，一个元素绑定多个事件时，后面的事件会覆盖前面的事件

事件流描述的是从页面中接受事件的顺序，但有意思的是，微软（IE）和网景（Netscape）开发团队居然提出了两个截然相反的事件流概念，**IE的事件流是事件冒泡流(event bubbling)，而Netscape的事件流是事件捕获流(event capturing)。**

**事件冒泡**

    　　　IE提出的事件流叫做事件冒泡，即事件开始时由**最具体的元素接收，然后逐级向上传播到body节点**。

**事件捕获**

       网景公司提出的事件流叫事件捕获流。

       事件捕获流的思想是body更早接收到事件，而最具体的节点应该最后接收到事件，针对上面同样的例子，点击按钮，那么此时click事件会按照这样传播：（下面我们就借用addEventListener的第三个参数来模拟事件捕获流）

如果对一个元素设置多个onclick，将只会执行最后一个，前面的被覆盖。

addEventListener 设置几个执行几个

### DOM2

DOM2 将事件分为三个阶段；

当一个DOM事件触发时，它不是在触发的对象上只触发一次的，而是经历三个阶段。分别为

1：一开始从文档的根节点流向目标对象（**捕获阶段**）

2：然后在目标对向上被触发（**目标阶段**）

3：之后再回溯到文档的根节点（**冒泡阶段**）.

**第三个参数默认为false，**事件冒泡可以兼容IE早版本的浏览器。

addEventListener:false =>冒泡（默认值）

addEventListener:true=>捕获

DOM0级添加事件时，后面的事件会覆盖前面的事件！而DOM2级则不会，多个事件都会执行！

Dom2添加事件和移除事件的方式只有一种：addEventListener()和removeEventListener()。

# 页面性能优化的方法

## 资源压缩与合并

**主要是html 压缩、css 压缩、js 的压缩和混乱和文件合并**。

## 二、非核心代码延迟加载的方式

**异步加载的三种方式—— async 和 defer、动态脚本创建**

**DOMContentLoaded 和 load事件：**

当初始的 **HTML**文档被完全加载和解析完成之后，**DOMContentLoaded**事件被触发，而无需等待样式表、图像和子框架的完成加载。

[load](https://developer.mozilla.org/en-US/docs/Mozilla_event_reference/load) 是在一个页面完全加载是触发 包括HTML js css img等等全部

1. **async 方式**

* async 属性是 HTML5 新增属性，
* async 属性规定浏览器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，加载完成立即执行。但可能会在DOMContentLoaded事件触发之前或之后执行。所以可能会导致DOM操作失败
* async 属性仅适用于外部脚本
* 如果是多个脚本，该方法不能保证脚本按顺序执行

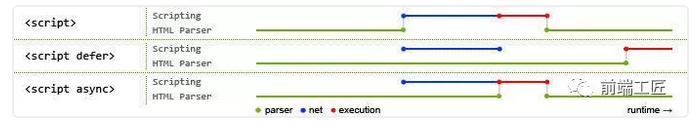
<script type="text/javascript" src="xxx.js" async="async"></script>

1. **defer 方式**

* defer 属性规定脚本会立即加载；但是 script.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。
* 如果是多个脚本，该方法可以确保所有设置了 defer 属性的脚本按顺序执行
* 如果脚本不会改变文档的内容，可将 defer 属性加入到 script 标签中，以便加快处理文档的速度

1. **动态创建 script 标签**

在还没定义 defer 和 async 前，异步加载的方式是动态创建 script，通过 window.onload 方法确保页面加载完毕再将 script 标签插入到 DOM 中,具体代码

三者区别：

其中蓝色线代表网络读取，红色线代表执行时间，这俩都是针对脚本的；绿色线代表 HTML 解析。

* 1. **defer 是在HTML解析完之后才会执行，如果是多个，按照加载的顺序依次执行**
  2. **async 是在加载完之后立即执行，如果是多个，执行顺序和加载顺序无关**

## 三 利用浏览器缓存

## 四、使用 CDN

**通过将静态资源(例如 javascript，css，图片等等）缓存到离用户很近的相同网络运营商的CDN（content delivery node）节点上，不但能提升用户的访问速度，还能节省服务器的带宽消耗，降低负载。**

**CDN网络中包含的功能实体包括内容缓存设备、内容交换机、内容路由器、CDN内容管理系统等组成。**

~内容缓存为CDN网络节点，位于用户接入点，是面向最终用户的内容提供设备，可缓存静态Web内容和流媒

~内容交换机处于用户接入集中点，可以均衡单点多个内容缓存设备的负载，并对内容进行缓存负载平衡及访问控制。

~内容路由器负责将用户的请求调度到适当的设备上。内容路由通常通过负载均衡系统来实现，动态均衡各个内容缓存站点的载荷分配，为用户的请求选择最佳的访问站点，同时提高网站的可用性。

~内容管理系统负责整个CDN的管理

CDN的基本原理是广泛采用各种缓存服务器，将这些缓存服务器分布到用户访问相对集中的地区或网络中，在用户访问网站时，利用全局负载技术将用户的访问指向距离最近的工作正常的缓存服务器上，由缓存服务器直接响应用户请求。

当请求达到CDN节点后，节点会判断自己的内容缓存是否有效，如果有效，则立即响应缓存内容给用户，从而加快响应速度。

## 五、预解析 DNS

**递归**：靠别人。客户端只发一次请求，向本地查询，然后本地向根查询，然后是顶级，然后是权限，要求对方给出最终结果，再依次返回。（较少使用）

**迭代**：靠自己。本地域名服务器作为客户向根发出请求，它就会返回一个能解答这个查询的其它名称服务器列表， 客户会再向返回的列表中发出请求，直到找到最终负责所查域名的名称服务器，从它得到最终结果。

递归和迭代相结合：客户端向本地域名服务器是迭代查询，本地服务器向外查询是递归查询。

**资源预加载是另一个性能优化技术，我们可以使用该技术来预先告知浏览器某些资源可能在将来会被使用到。**通过 DNS 预解析来告诉浏览器未来我们可能从某个特定的 URL 获取资源，当浏览器真正使用到该域中的某个资源时就可以尽快地完成 DNS 解析。例如，我们在页面中需要从 example.com 获取图片或音频资源，那么可以在文档顶部的标签中加入以下内容：

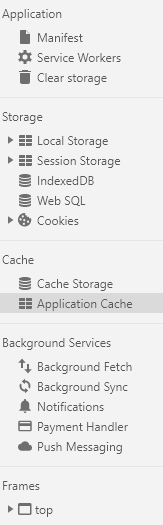
<link rel="dns-prefetch" href="//example.com">

当我们从该 URL 请求一个资源时，就不再需要等待 DNS 的解析过程。该技术对使用第三方资源特别有用。

 另外需要注意的是，**浏览器会对 a 标签的 href 自动启用 DNS Prefetching，所以 a 标签里包含的域名不需要在 head 中手动设置 link。但是在 HTTPS 下不起作用，需要 meta 来强制开启功能**。这个限制的原因是防止窃听者根据 DNS Prefetching 推断显示在 HTTPS 页面中超链接的主机名。下面这句话作用是强制打开 a 标签域名解析

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

# 浏览器缓存



<http://www.solves.com.cn/it/cxkf/bk/2019-08-01/2608.html>

**(1). http缓存**

http缓存是基于http协议的浏览器文件缓存机制，当重复请求某一个文件时，浏览器可以根据协议的**头文件**判断是从服务器端请求文件还是从本地读取文件。

上图中的Frames展示的就是http缓存

**(2). indexDB 客户端本地数据库（大型数据库）**

indexDB是浏览器中的一种数据库，用来保存结构化数据。

indexDB[设计](http://www.solves.com.cn/it/rj/ps/)的操作完全是异步进行的。

indexDB最大的特色是使用对象保存数据。

**(3). Cookies**

**(4). storage类型**

localStorage、sessionStorage

**(6). application cache 离线存储**

application cache**是HTML5的新特性**，允许浏览器在本地存储页面所需要的资源，使得页面离线也可以访问。

在用户没有与因特网进行连接时，可以正常访问站点或应用，在用户与因特网连接时，更新用户机器上的缓存文件。

application cache是从浏览器的缓存中分出来的一块缓存区。

**如何判断设备是否离线？**

if(navigator.onLine) {

//设备在线时的操作

} else {

//设备离线时的操作

}

# http状态码

|  |  |
| --- | --- |
| 1\*\* | 信息，服务器收到请求，需要请求者继续执行操作 |
| 2\*\* | 成功，操作被成功接收并处理 |
| 3\*\* | 重定向，需要进一步的操作以完成请求 |
| 4\*\* | 客户端错误，请求包含语法错误或无法完成请求 |
| 5\*\* | 服务器错误，服务器在处理请求的过程中发生了错误 |

下面是常见的HTTP状态码：

* 200 - 请求成功
* 204 No Content 意思等同于请求执行成功，但是没有数据，所以对于一些提交到服务器处理的数据只需要返回是否成功的情况下，可以考虑使用状态码204来作为返回信息，从而省掉多余的数据传输
* 301 - 资源（网页等）被永久转移到其它URL
* 400当其他4XX响应代码不适用时，就采用400。此响应代码通常用于“服务器收到客户端通过PUT或者POST请求提交的表示，表示的格式正确，但服务器不懂它什么意思”的情况。
* 401 （未授权） 请求要求身份验证。 对于需要登录的网页，服务器可能返回此响应。
* 404 - 请求的资源（网页等）不存在
* 405 不允许此方法  对于请求所标识的资源，不允许使用请求行中所指定的方法。
* 400 Bad Request：客户端请求有语法错误，服务器无法理解。
* 401 Unauthorized：请求未经授权，这个状态代码必须和WWW-Authenticate报头域一起使用。
* 403 Forbidden：服务器收到请求，但是拒绝提供服务
* 404 Not Found：请求资源不存在。比如，输入了错误的url
* 415 Unsupported media type：不支持的媒体类型
* 500 - 内部服务器错误

|  |  |  |
| --- | --- | --- |
| 304 | 未修改 | 自从上次请求后，请求的网页未修改过,告知客户端直接从缓存中提取数据 |

 **301永久性重定向**：新网址完全继承旧网址，旧网址的排名等完全清零

* 301重定向是网页更改地址后对搜索引擎友好的最好方法，只要不是暂时搬移的情况，都建议使用301来做转址。常用来使域名跳转

 **302临时性重定向**：对旧网址没有影响，但新网址不会有排名，比如：未登录的用户访问用户中心重定向到登录界面

。

# 原生JS实现ajax

//第1步创建一个xhr对象，使用new关键来调用一个内置构造函数 *var* xhr = *new* XMLHttpRequest();  
  
//第2步指定接收回来的内容，怎么处理。监听xhr对象的onreadystatechange事件，这个事件在xhr对象的“就绪状态”改变的时候触发。我们只关心就绪状态为4的时候的事情。xhr.onreadystatechange = *function*(){  
 *if*(xhr.readyState == 4){  
 //接收完文件要做的事情,返回的数据为responseTextcontent.innerHTML = xhr.responseText;  
 }  
};  
//第3步创建一个请求，第一个参数是请求的类型get或者post，第二个参数就是请求的路径，第三个参数叫做是否使用异步机制xhr.open("get","a.txt",*true*);  
//第4步发送请求，圆括号里面是请求头内容，get请求没有报文头写nullxhr.send(*null*);

请求方式为post时

xhr.open("POST", options.url, true);

23 //设置表单提交时的内容类型

24 xhr.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");

25 xhr.send(params);

0 - (未初始化)还没有调用send()方法  
1 - (载入)已调用send()方法，正在发送请求  
2 - (载入完成)send()方法执行完成，  
3 - (交互)正在解析响应内容  
4 - (完成)响应内容解析完成，可以在客户端调用了

# HTTPS 保证安全性

：Hyper Text Transfer Protocol over SecureSocket Layer

HTTPS 在HTTP 的基础下加入[SSL](https://baike.baidu.com/item/SSL/320778) 层，HTTPS 的安全基础是 SSL，因此加密的详细内容就需要 SSL。

SSL Secure Sockets Layer协议位于[TCP/IP协议](https://baike.baidu.com/item/TCP%2FIP%E5%8D%8F%E8%AE%AE)与各种[应用层](https://baike.baidu.com/item/%E5%BA%94%E7%94%A8%E5%B1%82)协议之间，为[数据通讯](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E9%80%9A%E8%AE%AF)提供安全支持。SSL协议可分为两层： SSL记录协议（SSL Record Protocol）； SSL[握手协议](https://baike.baidu.com/item/%E6%8F%A1%E6%89%8B%E5%8D%8F%E8%AE%AE)（SSL Handshake Protocol）：它建立在SSL记录协议之上，用于在实际的数据传输开始前，通讯双方进行[身份认证](https://baike.baidu.com/item/%E8%BA%AB%E4%BB%BD%E8%AE%A4%E8%AF%81)、协商[加密算法](https://baike.baidu.com/item/%E5%8A%A0%E5%AF%86%E7%AE%97%E6%B3%95)、交换加密[密钥](https://baike.baidu.com/item/%E5%AF%86%E9%92%A5)等。

**对称加密**：服务端只有一个**密钥（key）**，用这个密钥把response加密，同时客户端也对request用这个密钥加密，；黑客本身也是一个客户端，所以会知道这个密钥，然后拦截其他response就能破译

**非对称加密：**所有用户共享一个**公钥（pk）**，同时服务器有自己的**私钥(sk)**，两者可以互相解密；第一步，客户端向服务器拿到公钥；然后将request加密后发给服务器；第二步，服务器用自己的私钥进行解密；然后将response加密后发回，第三步，客户端用公钥解密response；问题来了，黑客也可以获取公钥然后解密啊

两者结合：**前期使用非对称加密**，第一步，客户端想服务器索要公钥；第二步，客户端随机生成一个number1用公钥加密，然后服务端用私钥对number1进行number1解密；然后number1就成了**对称加密**里的密钥了。

这时候黑客也能破解；就是在第一步的时候就拦截公钥，然后充当中间人的角色将；如下图



然后为了解决这个中间人问题，成立了一个机构CA，它负责将pk用cpk加密然后得到一个证书（license），CA机构的[**数字签名**](https://baike.baidu.com/item/%E6%95%B0%E5%AD%97%E7%AD%BE%E5%90%8D)使得攻击者不能伪造和篡改证书;从此客户不再请求pk而是请求license；而最终客户端与服务器请求需要得到pk，所以需要向CA请求CPK；但拿cpk的这一步又可能被黑客窃取，所以CA机构的很多cpk直接写死在OS里；

SSL Secure Sockets Layer协议位于[TCP/IP协议](https://baike.baidu.com/item/TCP%2FIP%E5%8D%8F%E8%AE%AE)与各种[应用层](https://baike.baidu.com/item/%E5%BA%94%E7%94%A8%E5%B1%82)协议之间，为[数据通讯](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E9%80%9A%E8%AE%AF)提供安全支持。SSL协议可分为两层： SSL记录协议（SSL Record Protocol）； SSL[握手协议](https://baike.baidu.com/item/%E6%8F%A1%E6%89%8B%E5%8D%8F%E8%AE%AE)（SSL Handshake Protocol）：它建立在SSL记录协议之上，用于在实际的数据传输开始前，通讯双方进行[身份认证](https://baike.baidu.com/item/%E8%BA%AB%E4%BB%BD%E8%AE%A4%E8%AF%81)、协商[加密算法](https://baike.baidu.com/item/%E5%8A%A0%E5%AF%86%E7%AE%97%E6%B3%95)、交换加密[密钥](https://baike.baidu.com/item/%E5%AF%86%E9%92%A5)等。

#### 访问百度的过程：

* 1. C=>S 支持SSL版本、非对称加密算法，随机数num1
  2. S=>C 确定就用C发来的SSL版本 对称算法 随机数num2 证书
  3. C从CA获得cpk
  4. C=>S 随机数num3，hash（第一步第二步的所有数值）=xx
  5. S xx==hash(第一步第二步的所有数值)？ 相等将随机数num1,2,3一起生成一个key **不能将key传回去，因为传输过程会泄密**
  6. S=>C hach(第一步第二步第四步)==zz
  7. C hach(第一步第二步第四步)==zz? 相等将随机数num1,2,3一起生成一个key

# 哈希算法

　Hash，一般翻译做“散列”，也有直接音译为“哈希”的，就是把任意长度的输入（又叫做预映射， pre-image），通过散列算法，变换成固定长度的输出，该输出就是散列值。这种转换是一种压缩映射，也就是，散列值的空间通常远小于输入的空间，不同的输入可能会散列成相同的输出，而不可能从散列值来唯一的确定输入值。简单的说就是一种将任意长度的消息压缩到某一固定长度的消息摘要的函数。

**它是一种单向密码体制，即它是一个从明文到密文的不可逆的映射，只有加密过程，没有解密过程。同时，哈希函数可以将任意长度的输入经过变化以后得到固定长度的输出。**

# Token登录

 1、用户发出登录请求，带着用户名和密码到服务器经行验证，服务器验证成功就在后台生成一个token返回给客户端

2、 客户端将token存储到cookie中，服务端将token存储到redis中，可以设置存储token的有效期。

 3、后续客户端的每次请求资源都必须携带token，这里放在请求头中，服务端接收到请求首先校验是否携带token，以及token是否和redis中的匹配，若不存在或不匹配直接拦截返回错误信息（如未认证）。

## JST

JSON Token

第一次访问后 服务器生成一个JSON 文件 来进行身份验证

# 浏览器渲染过程

浏览器是异步的，即边加载边解析。

**Html解析**

是异步的，每一个response包大概是64KB，返回一点解析一点。遇到其他需要请求的资源比如img,link会发送请求。

**Css的解析**

**Style标签**是**html解析器**（另开一个html解析器）来解析的，可能会出现**闪屏**的情况，是**异步**的，即解析一点返回一点。所以如果有几万行设置颜色的样式，会一直变化，知道最终颜色。

* 不阻塞渲染
* 不阻塞dom的解析

就是因为不阻塞，所以会产生闪屏

**Link**引入的CSS，**由css 解析器来解析**。是**同步**的，即解析css的时候，其他的都不能进行（**阻塞渲染，利用这个可以避免闪屏**）。所以如果有几万行设置颜色的样式，所以只会显示最终的颜色。

* 阻塞后面js的执行。由于不阻塞dom的解析，所以可以先请求script标签，然后请求回来之后不会执行。
* 不阻塞dom的解析（css 解析器解析的时候，html解析器也在运行，解析完之后就得等待，然后dom和cssom合并渲染）
* css加载不会阻塞DOM树的解析
* css加载会阻塞DOM树的渲染
* css加载会阻塞后面js语句的执行、

**Js**

* 阻塞后续dom的解析
* 阻塞页面渲染
* 阻塞后续js执行

**总结**

* Css解析和js的执行是互斥的
* Css阻塞，js阻塞都不会阻塞外部资源加载
* 现代浏览器，在执行js时，浏览器其他线程会预解析文档的其他部分，找出需要下载的其他资源。

**图层：**

1、拥有3D变化的属性

2、有canvas节点

3、CSS3动画的节点

4、will-change

**页面再次变化时：**

Layout => repaint 重绘，以图层为单位,如果某个元素变了，整个图层都要重绘

所以，为了性能，**要为变化的东西开一个图层**。比如：加一个will-change

Print => reflow 重排/回流，也是以图层为单位

Composite layer 组合图层

**常见触发重排的操作：**

重排的成本比重绘高很多，因为它往往带来其他节点的改变。

1. Dom位置的改变
2. 修改Css样式
3. Resize窗口
4. Display：none=>重排；visibility：hidden=>重绘
5. 修改网页默认字体
6. 获取某些属性:width,height也会重排。因为会把document刷新一遍

**针对重排重绘的优化：**

重新渲染的精准过程：recalculate=>layout=>paint=>composite layers

所以就要尽量减少以上四步；

1. 元素位置发生改变时，尽量用css3的transform来代替top,left等操作
2. 用opacity代替visibility
   1. 使用visibility不触发重排，但是依然触发重绘
   2. 使用opacity重排重绘都触发
   3. Opacity配合图层使用（加will-change），重排重绘都不发生；

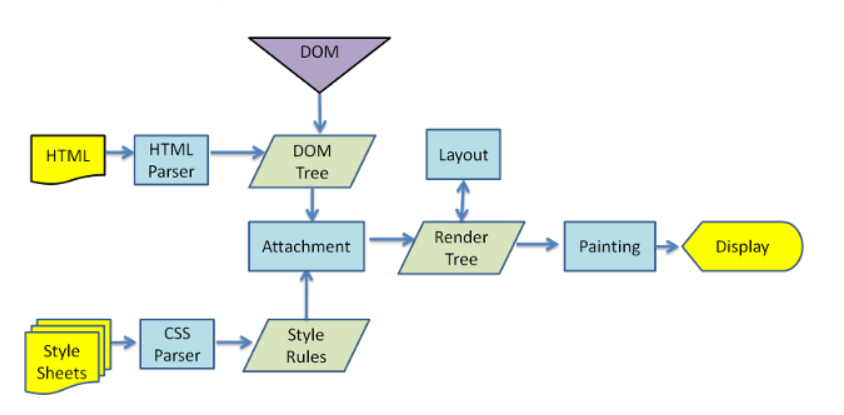
3．将多次改变属性合并成一次，比如把样式放在class里面，然后改变className

4. 先将dom离线化，比如display：none，再修改样式，这样就只用两次重排

5. 不要把width,height属性放在循环里面

6. 编写动画时，用requestAnimationFrame代替定时器动画。requestAnimationFrame在下一次重绘之前调用函数，递归调用，就可以实现动画。不用指定频率，和显示器保持一致。

用cancelAnimationFrame（id）可以停止动画。



CSS阻塞渲染意味着，在CSSOM完备前，页面将一直处理白屏状态，这就是为什么样式放在head中，仅仅是为了更快的解析CSS，保证更快的首次渲染。

JS可以操作DOM来修改DOM结构，可以操作CSSOM来修改节点样式，这就导致了浏览器在解析HTML时，一旦碰到script，就会立即停止HTML的解析，也阻塞了其后的CSS解析，整个解析进程必须等待JS的执行完成才能够继续。从性能角度上讲，将script放在页面底部，也就合情合理了

# 什么是 Web Worker？

当在 HTML 页面中执行脚本时，页面的状态是不可响应的，直到脚本已完成。

让JS有了多线程的能力

Worker工作在一个专用的作用域DedicatedWorkerGlobalScope,在这个作用域中，不能直接操作DOM节点，不能使用Window对象的默认方法和属性。不过对于网络的访问是完全没有问题的。

web worker 是运行在后台的 JavaScript，独立于其他脚本，不会影响页面的性能。您可以继续做任何愿意做的事情：点击、选取内容等等，而此时 web worker 在后台运行。

WebSocket让web端与服务端维持一个有效的长连接，实现服务端主动推送数据。将二者一结合，业务系统信息流转通知功能完全就可以剥离出来。

# Fetch

fetch是一种HTTP数据请求的方式，是XMLHttpRequest的一种替代方案。fetch不是ajax的进一步封装，而是原生js。Fetch函数就是原生js，没有使用XMLHttpRequest对象。

fetch发送post请求的时候发送两次，第一次状态码是204，第二才成功；

第一次是发了一个options请求，询问服务器是否支持修改的请求头，如果支持，则在第二次中发送真正的请求

优点：

链式写法，没有回调地狱

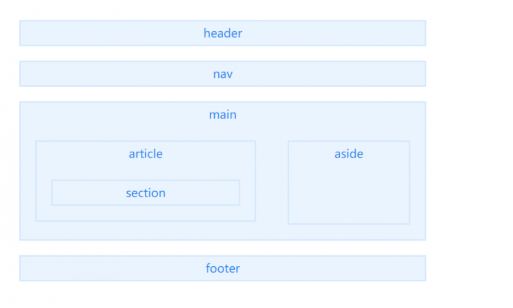
缺点：

* 1）fetchtch只对网络请求报错，对400，500都当做成功的请求，需要封装去处理
* 2）fetch默认不会带cookie，需要添加配置项
* 3）fetch没有办法原生监测请求的进度，而XHR可以

# HTML标签语义化

是指标签包含了正确的内容，结构良好，便于阅读，比如

<header></header>  
<footer></footer>  
<nav></nav>



# Doctype作用

告诉浏览器用什么方式来渲染页面；有两种模式

**严格模式**：以浏览器最高标准来运行

**混杂模式**：向下兼容

# Cookie如何防止XSS攻击

**XSS 是指攻击者在返回的HTML中嵌入js脚本，其原理是攻击者向有XSS漏洞的网站中输入(传入)恶意的js代码，当其它用户浏览该网站时，这段js代码会自动执行，从而达到攻击的目的。**

为减轻这些攻击,需要在http头部配上：

**Httponly:** 通过js脚本将无法读取到cookie信息，这样能有效的防止XSS攻击

**Secure:** 告诉浏览器，只在请求为https的时候发送cookie

# CRSF 跨站请求伪造

在没有关闭相关网页的情况下，点击其他人发来的CSRF链接，利用客户端的cookie直接向服务器发送请求

1. 检查是否包含HTTP\_REFERER（http请求头的Referer参数的值，表示来源地址），是否包含SERVER\_NAME（http包头的Host参数，表示要访问的主机名），通过这种机制抵御CSRF攻击。
2. 用户每次访问改密页面时，服务器会返回一个随机的token，向服务器发起请求时，需要提交token参数，而服务器在收到请求时，会优先检查token，只有token正确，才会处理客户端的请求。
3. 在头部自定义属性来验证

# 什么是RESTful

一种架构

用url定位资源，用http描述操作是什么

# 强缓存和协商缓存

浏览器缓存包含两种类型，即强缓存（也叫本地缓存）和协商缓存，浏览器在第一次请求发生后，再次请求时：

1. 浏览器在请求某一资源时，会先获取该资源缓存的header信息，判断是否命中强缓存（cache-control和expires信息），若命中直接从缓存中获取资源信息，包括缓存header信息；本次请求根本就不会与服务器进行通信；
2. 如果没有命中强缓存，浏览器会发送请求到服务器，请求会携带第一次请求返回的有关缓存的header字段信息（Last-Modified/If-Modified-Since和Etag/If-None-Match），由服务器根据请求中的相关header信息来比对结果是否协商缓存命中；若命中，则服务器返回新的响应header信息更新缓存中的对应header信息，但是并不返回资源内容，它会告知浏览器可以直接从缓存获取；否则返回最新的资源内容

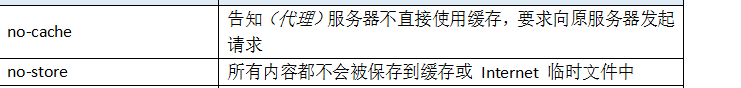
Expires

        Expires是http1.0的规范，它的值是一个绝对时间的GMT格式的时间字符串，这个时间代表这这个资源的失效时间，只要发送请求时间是在Expires之前，那么本地缓存始终有效，则在缓存中读取数据。失效的时间是一个绝对时间，当服务器与客户端时间偏差较大时，会导致缓存混乱。如果同时出现Cache-Control:max-age和Expires，那么**max-age优先级更高**。

Cache-Control

        Cache-Control是在http1.1中出现的，主要是利用该字段的max-age值来进行判断，它是一个相对时间，例如Cache-Control:max-age=3600，代表着资源的有效期是3600秒。cache-control除了该字段外，还有下面几个比较常用的设置值：

no-cache：不使用本地缓存。需要使用缓存协商，先与服务器确认返回的响应是否被更改，如果之前的响应中存在ETag，那么请求的时候会与服务端验证，如果资源未被更改，则可以避免重新下载。



**Cache-Control请求字段被各个浏览器支持的较好，其优先级也比较高，当和别的字段（如Expires）一起用时，会覆盖其他字段。**

**Cache-Control: no-cache：**这个很容易让人产生误解，使人误以为是响应不被缓存。实际上Cache-Control: no-cache是会被缓存的，只不过每次在向客户端（浏览器）提供响应数据时，缓存都要向服务器评估缓存响应的有效性。

**Cache-Control: no-store：**这**个才是响应不被缓存**的意思。

Etag

是服务端的一个资源的标识

比如，浏览器第一次请求一个资源的时候，服务端给予返回，并且返回了ETag: "50b1c1d4f775c61:df3" 这样的字样给浏览器，当浏览器再次请求这个资源的时候，浏览器会将If-None-Match: W/"50b1c1d4f775c61:df3" 传输给服务端，服务端拿到该ETAG，对比资源是否发生变化，如果资源未发生改变，则返回304HTTP状态码，不返回具体的资源。

If-Modified-Since

浏览器缓存记录的该文件的最后服务器修改时间。

浏览器发送HTTP请求时，把If-Modified-Since一起发到服务器去，服务器会把这个时间与服务器上实际文件的最后修改时间进行比较。

* + 如果时间一致，那么返回HTTP状态码304（Not Modified），客户端接到之后，直接把本地缓存文件显示到浏览器中。
  + 如果时间不一致，就返回HTTP状态码200和新的文件内容，客户端接到之后，会丢弃旧文件，把新文件缓存起来，并显示到浏览器中。

浏览器第一次访问服务器的文件时。不会携带If-Modified-Since，因为未有本地缓存。

服务器收到请求，开始处理。响应20

0，http头有标签：Last-Modified（date），即服务器本地记录的文件最后修改时间。

浏览器收到响应，缓存到本地。

浏览器第二次请求时，携带If-Modified-Since（date）,

服务器比较If-Modified-Since（date）,是否与服务器本地记录相同。决定返回状态200还是304.

# GET POST区别

1. get通过URL参数传递，post放在request body中
2. get请求参数有长度限制，post没有
3. post更安全
4. get请求参数会保存在浏览器历史记录里面，post不会

# Mouseover 和 mouseenter 区别

**mouseover**当鼠标进入元素或者其子元素时都会触发（会冒泡）；对应移除事件是**mouseout**

**mouseenter**：当鼠标进入元素本身（不包含子元素），才触发事件，对应移除事件是**mouseleave**

# ajax解决浏览器缓存问题

1. 发送请求前加上setRequestHeader(cachecontrol,’no-cache’)
2. 在URL后面加上时间戳或者随机数

# PWA

Progressive Web App，渐进式增强WEB应用

是谷歌主导的一种新时代网页（应用）

一个PWA应用首先是一个网页，可以通过web技术编写出一个网页应用。随后添加上APP manifast和service worker来实现PWA的安装和离线等功能





# 颁发证书的机构

Comodo 科摩多 最大证书认证机构

# Promise

它让异步函数不用写回调函数，链式调用

函调函数callback有一个 callback hell (回调黑洞)

比如：下面每隔1秒执行一个加法函数，必须得回调套回调

*function* sum(a,b,callback) {  
 setTimeout(*function* () {  
 callback(a+b)  
 },1000)  
}  
sum(1,2,*function* (m) {  
 console.log(m)  
 sum(3,2,*function* (m) {  
 console.log(m)  
 })  
})

### Promise本身是一个语法糖

上面代码可以写成下面这样

*function* sum(a,b) {  
 *return new* Promise((resolve,reject)=>{  
 setTimeout(*function* () {  
 resolve(a+b)  
 },1000)  
 })  
  
}  
  
sum(1,2).then(*function* (m) {  
 console.log(m);  
 *return* sum(5,5)  
}).then(*function* (m) {  
 console.log(m)  
  
})

**then是Promise实例的方法**

Resolve

**有三个状态 pending**

reject

Promise对象的错误具有“冒泡”性质，会一直向后传递，直到被捕获为止。也就是说，错误总是会被下一个catch语句捕获。

## 原生实现Promise

# async / await

必须要有一个promise函数，await 后面必须是一个promise的实例

# Html5新特性

         （一）  语义标签

         （二）增强型表单

* email
* number
* range
* color

         （三）视频和音频

         （四）Canvas绘图

         （五）SVG绘图

         （六）地理定位

         （七）拖放API

         （八） WebWorker

         （九） WebStorage

         （十）WebSocket

# Http头部

Cache-Control

Connection:keep-alive,

Connect-length:

.Content-Type

Referer

Date

Expires

Server

cookie

User-agent

# 函数防抖、节流

防抖

防抖函数 就是指触发事件后在 n 秒内函数只能执行一次，如果在 n 秒内又触发了事件，则会重新计算函数执行时间。

*function* debounce(fn,delay){  
 *let* time = *null*;  
 *// console.log(time)  
 return function* (...args) {  
 *// console.log(time)  
 //每一次调用前都要清除掉前一次触发的定时器* clearTimeout(time)  
 time = setTimeout(*function* () {  
 fn.call(*this*,...args)  
 },delay)  
 }  
 };

节流

节流函数 就是指连续触发事件但是在 n 秒中只执行一次函数。节流会稀释函数的执行频率。

*function* throttle(fn,delay) {  
*let* flag = *true*;  
 *return function* (...args) {  
 *if* (!flag){  
 *return* }*else*{  
 flag = *false*;  
 setTimeout(*function* () {  
 *// 绑定fn自己的this 和 参数，是为了让 throttle 函数最终返回的函数 this 指向不变以及依旧能接受到 e 参数。* fn.call(*this*,...args);  
 flag = *true*;  
 },delay);  
 }  
 }  
 }

# CSS

子元素会继承父盒子的宽，但是没高

# generate

<https://www.liaoxuefeng.com/wiki/1022910821149312/1023024381818112>

generator和函数不同的是，generator由function\*定义（注意多出的\*号），并且，除了return语句，还可以用yield返回多次。

**function**\* foo(x) {

**yield** x + 1;

**yield** x + 2;

**return** x + 3;

}

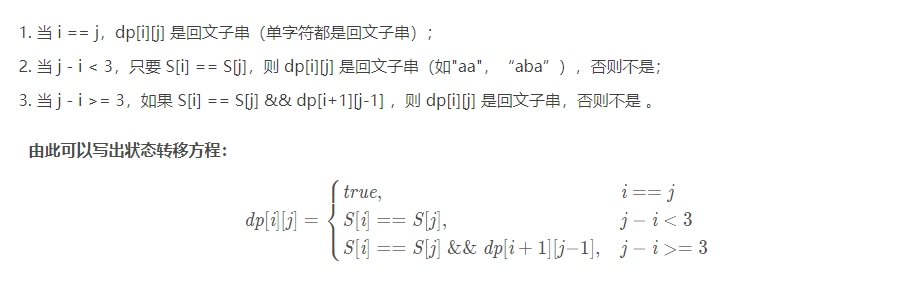
直接调用一个generator和调用函数不一样，foo()仅仅是创建了一个generator对象，还没有去执行它。

调用generator对象有两个方法，一是不断地调用generator对象的next()方法：

# 最大连续子序列 DP

*function* maxSublist(arr) {  
 *var* dp = [];  
 dp[0] = arr[0];  
 *for*(*let* i=1; i<arr.length; i++){  
 dp[i] = Math.max(arr[i],(dp[i-1]+arr[i]))  
 }  
 *let* max = dp[0]  
 *for*(*let* i=1; i<dp.length; i++){  
 *if* (dp[i]>max){  
 max = dp[i]  
 }  
 }  
 console.log(max)  
  
}

# 最大回文子串



# React

setState的“异步”并不是说内部由异步代码实现，其实本身执行的过程和代码都是同步的，只是合成事件和钩子函数的调用顺序在更新之前，导致在合成事件和钩子函数中没法立马拿到更新后的值，形式了所谓的“异步”，当然可以通过第二个参数 setState(partialState, callback) 中的callback拿到更新后的结果。

componentWillReceiveProps() 在生命周期的第一次render后不会被调用，但是会在之后的每次render中被调用 = 当父组件再次传送props。  在这个函数中调用this.setState()不会触发额外的渲染。

# DTD是什么

所谓的DTD就是**DocType Definition** 文档类型定义, 也有人翻译为**D**oc**T**ype **D**eclartion 文档类型声明

HTML文件的第一行，都是这个DTD，干什么用的？这一行语句，**告诉浏览器我是什么版本的HTML文件**。

HTML、CSS这两个语言，官网维护者就是总部位于美国的**W3C组织**

World Wide Web Consortium

HTML从诞生到现在，最新的版本HTML5。不是所有的浏览器都兼容哦，现在还有一部分人使用的浏览器过旧，所以现在很多公司还在使用HTML的上一个版本，HTML4.01。

HTML4.01又有三个小版本：

**HTML4.01 Strict** 严格版：不能使用font、b、u、i等等的废弃标签，不能使用框架集，结构和样式分离。

**HTML4.01 Transitional** 过渡版（通用版）：没有那么多限制，可以使用font等废弃标签，不能使用框架集

**HTML4.01 Frameset**：框架集版：可以使用框架集

**在XHTML1.0中，严格规定了标签必须是小写，所有的属性都必须用双引号封闭啊，必须有结尾反斜杠……**

# 项目页面优化的实践

公司主要用的vue 我们是分开做的 我是自己写的登录注册页面 商城我的 页面 他主要写商城的 就只学了点语法 而我只要还是react学的多一点

React是facebook出的一个框架 生态比较好

* 1. 减少http请求
  2. 第三方的库延迟加载 defer
  3. 合并文件
  4. 精灵图
  5. 缓存

# 自我介绍

您好 我来自2021届西北大学软件工程，我叫章景。因为对前端很兴趣，课余时间自学前端；学习路线主要

html+css=>js学了很长一段时间=>es6 =>react(webpack和gulp有了些了解；)之前一直在学校里做一些项目，自己写过一些图片懒加载，函数节流等功能；之后9月份开始实习，做的一个以售后为主的商城网站，我主要是完成登录注册表单等页面然后对网页的性能优化做出了一些尝试；

# 事件代理

在标准浏览器下我们一般使用event.target就能解决，然而低版本IE浏览器总是会出些幺蛾子，这时候就得使用event.srcElement。

# 严格模式

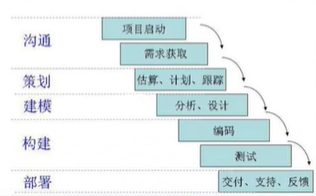
不允许使用未声明的变量：

不允许删除变量或对象。

不允许使用转义字符:

# 敏捷开发

**瀑布流开发：**



缺点很多，因为实际中需求经常改，需要改的太多了，在需求不明并且在项目进行过程中可能变化的情况下基本是不可行的。

敏捷开发强调的是适应性，在初期会被切分成各个小项目，对每个部分进行测试；

# Rgba&&opicity

opacity作用于元素本身以及元素内的所有内容，而rgba只作用于元素本身，子元素不会继承透明效果。

# Dom && css

DOM 解析和 CSS 解析是两个并行的进程,所以 CSS 加载不会阻塞 DOM 的解析。

然而,由于 Render Tree 是依赖于 DOM Tree 和 CSSOM Tree 的,

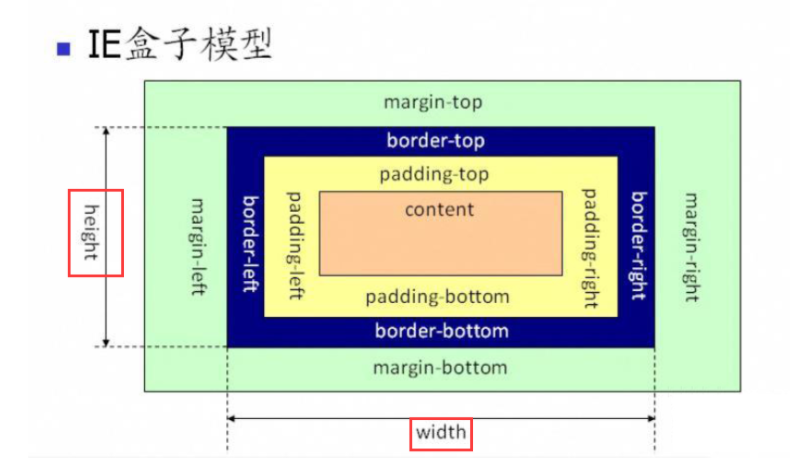
所以他必须等待到 CSSOM Tree 构建完成,也就是 CSS 资源加载完成(或者 CSS 资源加载失败)后,才能开始渲染。

因此,CSS 加载会阻塞 Dom 的渲染。

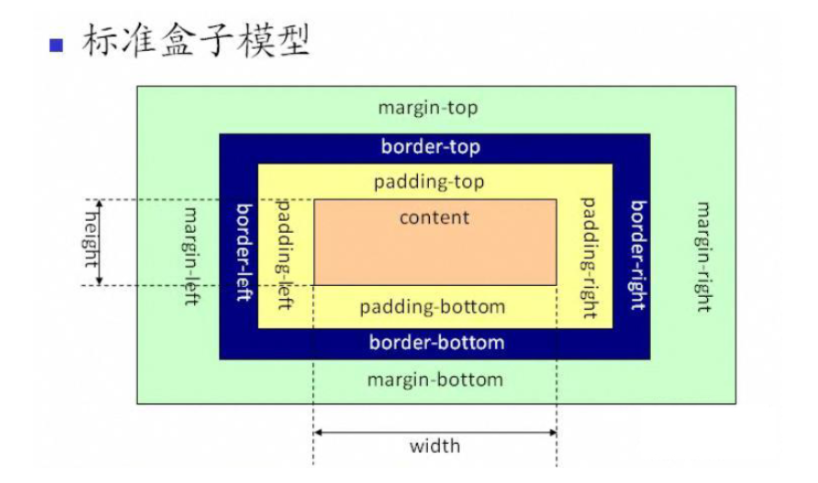
# 盒模型

box-sizing: border-box/ content-box（默认值）

IE盒模型 border-box



W3C标准盒子模型 content-box



# Typeof 原理

原理是这样的， 不同的对象在底层都表示为二进制， 在 JavaScript 中二进制前三位都为 0 的话会被判断为 object 类型， null 的二进制表示是全 0， 自然前三位也是 0， 所以执行 typeof 时会返回“object”。

# Instanceof

### instanceof的普通的用法，obj instanceof Object 检测Object.prototype是否存在于参数obj的原型链上。

# js如何完整的显示较长的数字

Math.pow(10, 99).toLocaleString().split(',').join('')

 toLocaleString([character]) 方法会将其对象转换成制定语言下的显示格式。默认为当前计算机语言。

# Css选择器

关系选择器

> 儿子，亲儿子，不是后代，必须是儿子

+ next sibling，下一个兄弟

~ next all siblings ，下面所有兄弟

属性选择器

选择器img标签中所有src属性为images/2.png的：

|  |
| --- |
| 1. **img[src="images/2.png"]**{ 2. border:10px solid red; 3. } |

**方括号就是属性选择器，可以把任何HTML标签的属性当做选择器**。

|  |
| --- |
| 1. img[kaola="你"]{ 2. transform:rotate(45deg); 3. } |

哲学上讲：

|  |
| --- |
| 1. #box |

等价于

|  |
| --- |
| 1. [id="box"] |

引申了几个选择器：

以val开头的，IE7兼容

|  |
| --- |
| 1. p[class^="para"]{ 2. color:red; 3. } |

以val结尾的，IE7兼容

|  |
| --- |
| 1. p[class$="para"]{ 2. color:red; 3. } |

任何一个部分出现了val，IE7兼容

|  |
| --- |
| 1. p[class\*="para"]{ 2. color:red; 3. } |

# Padding margin设置百分比基于什么设置

都是基于**父元素的宽度**的。

# 移动端兼容方案

1. Viewport

每个移动设备浏览器中都有一个理想的宽度，这个理想的宽度是指css中的宽度，跟设备的物理宽度没有关系，在css中，这个宽度就相当于100%的所代表的那个宽度。我们可以用meta标签把viewport的宽度设为那个理想的宽度，如果不知道这个设备的理想宽度是多少，那么用device-width这个特殊值就行了，同时initial-scale=1也有把viewport的宽度设为理想宽度的作用。所以，我们可以使用

为什么需要有理想的viewport呢？比如一个分辨率为320x480的手机理想viewport的宽度是320px，而另一个屏幕尺寸相同但分辨率为640x960的手机的理想viewport宽度也是为320px，那为什么分辨率大的这个手机的理想宽度要跟分辨率小的那个手机的理想宽度一样呢？这是因为，只有这样才能保证同样的网站在不同分辨率的设备上看起来都是一样或差不多的。实际上，现在市面上虽然有那么多不同种类不同品牌不同分辨率的手机，但它们的理想viewport宽度归纳起来无非也就 320、360、384、400等几种，都是非常接近的，理想宽度的相近也就意味着我们针对某个设备的理想viewport而做出的网站，在其他设备上的表现也不会相差非常多甚至是表现一样的。

1. vw vh

百分比单位

依赖于视口单位（layout viewport）的大小，失去了最大最小的限制，可以结合rem

1. rem em

# BFC

### 如何创建BFC

* 1、float的值不是none。
* 2、position的值**不是static或者relative。**
* 3、display的值是**inline-block**、table-cell、**flex**、table-caption或者inline-flex
* 4、overflow的值不是visible

### BFC的布局规则

* 内部的Box会在垂直方向，一个接一个地放置。
* **Box垂直方向的距离由margin决定。属于同一个BFC的两个相邻Box的margin会发生重叠。**
* 每个盒子（块盒与行盒）的margin box的左边，与包含块border box的左边相接触(对于从左往右的格式化，否则相反)。即使存在浮动也是如此。
* **BFC的区域不会与float box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。**

**应用：**

1. 自适应两栏布局
2. 利用BFC避免margin重叠。
3. 清除浮动

# Get&post区别

<https://blog.csdn.net/ever_siyan/article/details/87935455>

GET和POST是什么？HTTP协议中的两种发送请求的方法。

HTTP的底层是TCP/IP。所以GET和POST的底层也是TCP/IP，也就是说，GET/POST都是TCP链接。GET和POST能做的事情是一样一样的。你要给GET加上request body，给POST带上url参数，技术上是完全行的通的。

Http Get方法提交的数据大小长度并没有限制，HTTP协议规范没有对URL长度进行限制。这个限制是特定的**浏览器及服务器**对它的限制。

多数浏览器对于POST采用两阶段发送数据的，先发送请求头，再发送请求体，即使参数再少再短，也会被分成两个步骤来发送（相对于GET）,也就是第一步发送header数据，第二部再发送body部分。火狐只发一次

。Http是应用层的协议，而再传输层有些情况TCP会出现两次连结的过程，http协议本身不保存状态信息，一次请求一次响应。对于TCP而言，通信次数越多反而可靠性越低，能在一次连结中传输完需要的信息是最可靠的，所以尽量使用GET请求来减少网络耗时。如果通信时间增加，这段时间客户端于服务器端一直保持连接状态，在服务器侧负载可能会增加，可靠性会下降。

# Viewport

window对象有一个**devicePixelRatio**属性，devicePixelRatio = 物理像素 / 独立像素。在Retina屏的iphone上，devicePixelRatio的值为2，也就是说1个css像素相当于2个物理像素。

**layout viewport** document.documentElement.clientWidth, 定默认情况下把viewport设为一个较宽的值，比如980px，这样的话即使是那些为桌面设计的网站也能在移动浏览器上正常显示

**visual viewport**代表 浏览器可视区域的大小window.innerWidth返回窗口的文档显示区的宽度。

**ideal viewport，**完美适配移动设备的viewport。所谓的完美适配指的是，首先不需要用户缩放和横向滚动条就能正常的查看网站的所有内容；第二，显示的文字的大小是合适，比如一段14px大小的文字，不会因为在一个高密度像素的屏幕里显示得太小而无法看清，理想的情况是这段14px的文字无论是在何种密度屏幕，何种分辨率下，显示出来的大小都是差不多的。

，在iphone中，css中的320px就代表iphone屏幕的宽度。安卓设备就比较复杂了，有320px的，有360px的，有384px的等等

<meta name="viewport" content="width=device-width, initial-scale=1.0, maximum-scale=1.0, user-scalable=0">

# Inline-block 和 block的区别

**Inline**

inline元素的margin和padding属性，水平方向的padding-left, padding-right, margin-left, margin-right都产生边距效果；但竖直方向的padding-top, padding-bottom, margin-top, margin-bottom不会产生边距效果。

对inline元素设置宽度不起作用，它的宽度是随着内容的长度变化的

**Inline-block**

就是将对象呈现为inline对象，让block元素不再独占一行，多个block元素可以同排一行，且元素具有block的属性，可设置宽高，是block和inline元素的综合体。

# 设计模式

观察者模式

发布订阅模式

状态模式

MVC

MVVM

# CSS3

**box-shadow**

**向右的偏移值、向下的偏移值、模糊半径、延展宽度、颜色。**

|  |
| --- |
| box-shadow: 2px 2px 2px 2px black; |

**transition过渡的基本语法**

下面就是一个完整的过渡语法：

|  |
| --- |
| transition:all 1s ease 0s; |

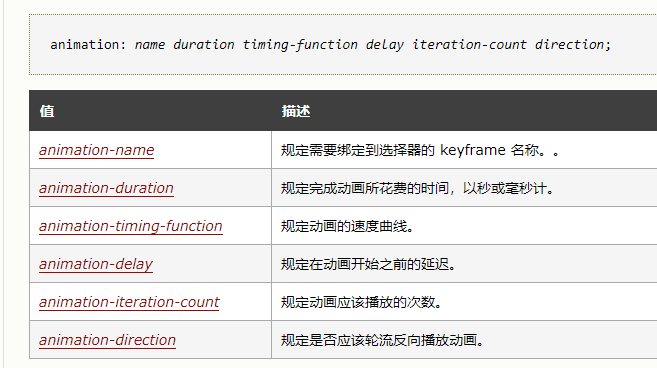
第1个参数： 就是想让什么属性参与过渡，如果想让所有属性参与过渡，写all

第2个参数： 就是过渡动画的时间，s表示秒，一定要写s，不要不写单位，也不要写别的；

第3个参数： 缓冲描述词，ease表示不匀速，匀速就是linear。

第4个参数： 延迟时间，0s就是不延迟。

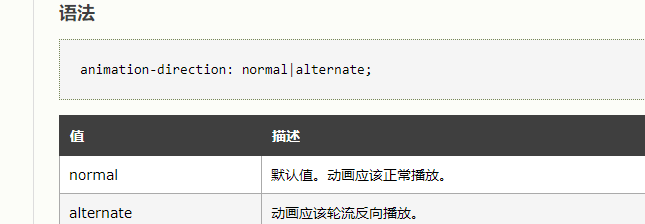
**animation**



animation-fill-mode: forwards;可以让动画停在最后一帧

无限循环运动

animation: move 1s ease-in 0s infinite alternate;

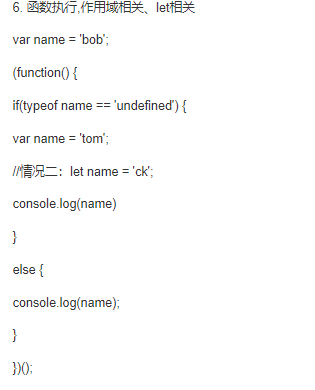


# 字符串方法

padStart padEnd

trim(),trimStart(),trimEnd()

# 看代码说结果



*let* obj = {  
 a() {  
 console.log(*this*)  
 },  
 name: 'jack',  
 showName: *this*.name  
  
}  
obj.a() //obj  
console.log(obj.showName) //undefined  
  
*let* a = obj.a;  
a(); //window

*function* Foo() {  
 getName = *function* () { console.log(1); };  
 console.log(*this*)  
 *return this*;  
}  
Foo.getName = *function* () { console.log(2);};  
Foo.prototype.getName = *function* () { console.log(3);};  
*var* getName = *function* () { console.log(4);};  
*function* getName() { console.log(5);}  
  
*//请写出以下输出结果：*Foo.getName(); *//2*getName(); *//4*Foo().getName(); *//1*getName(); *//1  
new* Foo().getName(); *//3*

*var* name = '123';  
 *var* obj = {  
 name: '456',  
 getName: *function* () {  
 console.log(*this*.name); *//456  
 function* printName () {  
 console.log(*this*.name); *//123* }  
 printName();  
 }  
 }  
obj.getName() *//123*