# JavaScript

## 事件绑定的方式

* 嵌入dom
* <button onclick="func()">按钮</button>
* 直接绑定
* btn.onclick = function(){}
* 事件监听
* btn.addEventListener('click',function(){})

## 事件触发的流程

1. 捕获阶段：从外到内
2. 目标阶段：在目标元素上触发事件
3. 冒泡阶段：从内到外

## 事件委托

事件委托利用了事件冒泡，只指定一个事件处理程序，就可以管理某一类型的所有事件。所有用到按钮的事件（多数鼠标事件和键盘事件）都适合采用事件委托技术， 使用事件委托可以节省内存。

<ul>  
  <li>苹果</li>  
  <li>香蕉</li>  
  <li>凤梨</li>  
</ul>  
  
// good  
document.querySelector('ul').onclick = (event) => {  
 let target = event.target  
 if (target.nodeName === 'LI') {  
 console.log(target.innerHTML)  
 }  
}  
  
// bad  
document.querySelectorAll('li').forEach((e) => {  
 e.onclick = function() {  
 console.log(this.innerHTML)  
 }  
})

## 事件循环

事件循环是一个单线程循环，用于监视调用堆栈并检查是否有工作即将在任务队列中完成。如果调用堆栈为空并且任务队列中有回调函数，则将回调函数出队并推送到调用堆栈中执行。

* [浏览器事件循环](https://github.com/woai3c/Front-end-articles/blob/master/eventloop.md)
* [Node.js 事件循环，定时器和 process.nextTick()](https://nodejs.org/zh-cn/docs/guides/event-loop-timers-and-nexttick/)

## 事件模型

### DOM0

直接绑定

<input onclick="sayHi()"/>  
  
btn.onclick = function() {}  
btn.onclick = null

### DOM2

* DOM2级事件可以冒泡和捕获
* 通过addEventListener绑定
* 通过removeEventListener解绑
* // 绑定  
  btn.addEventListener('click', sayHi)  
  // 解绑  
  btn.removeEventListener('click', sayHi)

### DOM3

* DOM3具有更多事件类型
* DOM3级事件在DOM2级事件的基础上添加了更多的事件类型，全部类型如下：
* UI事件，当用户与页面上的元素交互时触发，如：load、scroll  
  焦点事件，当元素获得或失去焦点时触发，如：blur、focus  
  鼠标事件，当用户通过鼠标在页面执行操作时触发如：dbclick、mouseup  
  滚轮事件，当使用鼠标滚轮或类似设备时触发，如：mousewheel  
  文本事件，当在文档中输入文本时触发，如：textInput  
  键盘事件，当用户通过键盘在页面上执行操作时触发，如：keydown、keypress  
  合成事件，当为IME（输入法编辑器）输入字符时触发，如：compositionstart  
  变动事件，当底层DOM结构发生变化时触发，如：DOMsubtreeModified
* 参考资料：
* [dom0、dom2、dom3事件](https://www.jianshu.com/p/3acdf5f71d5b)

## 如何自定义事件

### 新模式

const div = document.createElement('div') // 不创建元素，直接用 window 对象也可以  
const event = new Event('build')  
  
div.addEventListener('build', function(e) {  
 console.log(111)  
})  
  
div.dispatchEvent(event)

### 过时的模式

1. 原生提供了3个方法实现自定义事件
2. document.createEvent('Event') 创建事件
3. initEvent 初始化事件
4. dispatchEvent 触发事件

const events = {}  
  
function registerEvent(name) {  
 const event = document.createEvent('Event')  
 event.initEvent(name, true, true) // 事件名称，是否允许冒泡，该事件的默认动作是否可以被取消  
 events[name] = event  
}  
  
function triggerEvent(name) {  
 window.dispatchEvent(events[name])  
}  
  
registerEvent('resize') // 注册 resize 事件  
triggerEvent('resize') // 触发 resize 事件

[MDN](https://developer.mozilla.org/zh-CN/docs/Web/Guide/Events/Creating_and_triggering_events)

## 事件列表

[事件参考-MDN](https://developer.mozilla.org/zh-CN/docs/Web/Events#%E6%A0%87%E5%87%86%E4%BA%8B%E4%BB%B6)

## target和currentTarget区别

* event.target 返回触发事件的元素
* event.currentTarget 返回绑定事件的元素

## prototype和\_\_proto\_\_的关系是什么 **先说结论：** 1. prototype 用于访问函数的原型对象。 2. \_\_proto\_\_ 用于访问对象实例的原型对象（或者使用 Object.getPrototypeOf()）。

function Test() {}  
const test = new Test()  
test.\_\_proto\_\_ == Test.prototype // true

也就是说，函数拥有 prototype 属性，对象实例拥有 \_\_proto\_\_ 属性，它们都是用来访问原型对象的。

函数有点特别，它不仅是个函数，还是个对象。所以它也有 \_\_proto\_\_ 属性。

**为什么会这样呢**？因为函数是内置构造函数 Function 的实例：

const test = new Function('function Test(){}')  
test.\_\_proto\_\_ == Function.prototype // true

所以函数能通过 \_\_proto\_\_ 访问它的原型对象。

由于 prototype 是一个对象，所以它也可以通过 \_\_proto\_\_ 访问它的原型对象。对象的原型对象，那自然是 Object.prototype 了。

function Test() {}  
Test.prototype.\_\_proto\_\_ == Object.prototype // true

这样看起来好像有点复杂，我们可以换个角度来看。Object 其实也是内置构造函数:

const obj = new Object() // 我们可以把这个 obj 想象成原型对象 prototype  
obj.\_\_proto\_\_ == Object.prototype // true 换个角度来看，相当于 prototype.\_\_proto\_\_ == Object.prototype

从这一点来看，是不是更好理解一点。

为了防止无休止的循环下去，所以 Object.prototype.\_\_proto\_\_ 是指向 null 的，null 是万物的终点。

Object.prototype.\_\_proto\_\_ == null // true

既然 null 是万物的终点，那使用 Object.create(null) 创建的对象是没有 \_\_proto\_\_ 属性的，也没有 prototype 属性。

## 原型链

所有的 JS 对象(JS 函数是 prototype)都有一个 \_\_proto\_\_ 属性，指向它的原型对象。当试图访问一个对象的属性时，如果没有在该对象上找到，它还会搜寻该对象的原型，以及该对象的原型的原型，依次层层向上搜索，直到找到一个名字匹配的属性或到达原型链的末尾。

## 用 ES5 和 ES6 实现继承

ES5 prototype 寄生组合式继承

function SuperType(name) {  
 this.name = name  
}  
  
SuperType.prototype.sayName = function() {  
 console.log(this.name)  
}  
  
function SubType(name, age) {  
 SuperType.call(this, name)  
 this.age = age  
}  
  
function extendPrototype(Sub, Super) {  
 Sub.prototype = Object.create(Super.prototype)  
 Sub.prototype.constructor = Sub  
}  
  
extendPrototype(SubType, SuperType)  
  
SubType.prototype.sayAge = function() {  
 console.log(this.age)  
}  
  
const sub = new SubType('tom', 18)  
sub.sayAge() // 18  
sub.sayName() // tom

ES6 class

class SuperType {  
 constructor(name) {  
 this.name = name  
 }  
  
 sayName() {  
 console.log(this.name)  
 }  
}  
  
class SubType extends SuperType {  
 constructor(name, age) {  
 super(name)  
 this.age = age  
 }  
  
 sayAge(age) {  
 console.log(this.age)  
 }  
}  
  
const sub = new SubType('tom', 18)  
sub.sayAge() // 18  
sub.sayName() // tom

从实现上来看，ES6 更加简洁。

## 闭包

闭包是指有权访问另一个函数作用域中的变量的函数。

function sayHi(name) {  
 return () => {  
 console.log(`Hi! ${name}`)  
 }  
}  
const test = sayHi('xiaoming')  
test() // Hi! xiaoming

虽然sayHi函数已经执行完毕，但是其活动对象也不会被销毁，因为test函数仍然引用着sayHi函数中的变量name，这就是闭包。

但也因为闭包引用着另一个函数的变量，导致另一个函数即使不使用了也无法销毁，所以闭包使用过多，会占用较多的内存，这也是一个副作用。

### 利用闭包实现私有属性

const test = (function () {  
 let value = 0  
 return {  
 getVal() { return value },  
 setVal(val) { value = val }  
 }   
})()

上面的代码，就实现了一个私有属性 value，它只能用过 getVal() 来获取值，通过 setVal(val) 来设置值。

## 内存回收

在 JS 中，有两种内存回收算法。第一种是引用计数垃圾收集，第二种是标记-清除算法（从2012年起，所有现代浏览器都使用了标记-清除垃圾回收算法）。 ### 引用计数垃圾收集 如果一个对象没有被其他对象引用，那它将被垃圾回收机制回收。

let o = { a: 1 }

一个对象被创建，并被 o 引用。

o = null

刚才被 o 引用的对象现在是零引用，将会被回收。 #### 循环引用 引用计数垃圾收集有一个缺点，就是循环引用会造成对象无法被回收。

function f(){  
 var o = {};  
 var o2 = {};  
 o.a = o2; // o 引用 o2  
 o2.a = o; // o2 引用 o  
  
 return "azerty";  
}  
  
f();

在 f() 执行后，函数的局部变量已经没用了，一般来说，这些局部变量都会被回收。但上述例子中，o 和 o2 形成了循环引用，导致无法被回收。

### 标记-清除算法

这个算法假定设置一个叫做根（root）的对象（在Javascript里，根是全局对象）。垃圾回收器将定期从根开始，找所有从根开始引用的对象，然后找这些对象引用的对象……从根开始，垃圾回收器将找到所有可以获得的对象和收集所有不能获得的对象。

对于刚才的例子来说，在 f() 执行后，由于 o 和 o2 从全局对象出发无法获取到，所以它们将会被回收。

### 高效使用内存

在 JS 中能形成作用域的有函数、全局作用域、with，在 es6 还有块作用域。局部变量随着函数作用域销毁而被释放，全局作用域需要进程退出才能释放或者使用 delete 和赋空值 null undefined。

在 V8 中用 delete 删除对象可能会干扰 V8 的优化，所以最好通过赋值方式解除引用。

参考资料： \* [内存管理](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Memory_Management)

## 有一个函数，参数是一个函数，返回值也是一个函数，返回的函数功能和入参的函数相似，但这个函数只能执行3次，再次执行无效，如何实现

这个题目是考察闭包的使用

function sayHi() {  
 console.log('hi')  
}  
  
function threeTimes(fn) {  
 let times = 0  
 return () => {  
 if (times++ < 3) {  
 fn()  
 }  
 }  
}  
  
const newFn = threeTimes(sayHi)  
newFn()  
newFn()  
newFn()  
newFn()  
newFn() // 后面两次执行都无任何反应

通过闭包变量 times 来控制函数的执行

## 实现add函数,让add(a)(b)和add(a,b)两种调用结果相同

### 实现1

function add(a, b) {  
 if (b === undefined) {  
 return function(x) {  
 return a + x  
 }  
 }  
  
 return a + b  
}

### 实现2——柯里化

function curry(fn, ...args1) {  
 // length 是函数对象的一个属性值，指该函数有多少个必须要传入的参数，即形参的个数。  
 if (fn.length == args1.length) {  
 return fn(...args1)  
 }  
  
 return function(...args2) {  
 return curry(fn, ...args1, ...args2)  
 }  
}  
  
function add(a, b) {  
 return a + b  
}  
  
console.log(curry(add, 1)(2)) // 3  
console.log(curry(add, 1, 2)) // 3

## Ajax

Ajax(asynchronous JavaScript and XML) 是客户端上创建异步 Web 应用的一种 Web 开发技术。

借助 Ajax，Web 应用可以异步（在后台）向服务器发送数据和从服务器检索数据，而不会干扰现有页面的显示和行为。通过将数据交换层与表示层分离，Ajax 允许网页和扩展 Web 应用程序动态更改内容，而无需重新加载整个页面。

XMLHttpRequest API 经常用于异步通信。此外还有最近流行的fetch API。

const xhr = new XMLHttpRequest()  
xhr.onload = (res) => {  
 if (xhr.status == 200) {  
 console.log(xhr.response)  
 }  
}  
  
xhr.open('get', 'https://api.github.com/')  
xhr.send()

## 使用Ajax的优缺点分别是什么

### 优点

* 交互性更好。来自服务器的新内容可以动态更改，无需重新加载整个页面。
* 减少与服务器的连接，因为脚本和样式只需要被请求一次。
* 状态可以维护在一个页面上。JavaScript 变量和 DOM 状态将得到保持，因为主容器页面未被重新加载。
* 基本上包括大部分 SPA 的优点。

### 缺点

* 动态网页很难收藏。
* 如果 JavaScript 已在浏览器中被禁用，则不起作用。
* 有些网络爬虫不执行 JavaScript，也不会看到 JavaScript 加载的内容。
* 基本上包括大部分 SPA 的缺点。

参考资料： \* [使用 Ajax 的优缺点分别是什么？](https://github.com/yangshun/front-end-interview-handbook/blob/master/contents/zh/javascript-questions.md#%E4%BD%BF%E7%94%A8ajax%E7%9A%84%E4%BC%98%E7%BC%BA%E7%82%B9%E5%88%86%E5%88%AB%E6%98%AF%E4%BB%80%E4%B9%88)

## Ajax和Fetch区别

* ajax是使用XMLHttpRequest对象发起的，但是用起来很麻烦，所以ES6新规范就有了fetch，fetch发一个请求不用像ajax那样写一大堆代码。
* 使用fetch无法取消一个请求，这是因为fetch基于Promise，而Promise无法做到这一点。
* 在默认情况下，fetch不会接受或者发送cookies
* fetch没有办法原生监测请求的进度，而XMLHttpRequest可以
* fetch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
* fetch由于是ES6规范，兼容性上比不上XMLHttpRequest

## 变量提升

var 会使变量提升，var 无论发生在何处，都在执行任何代码之前进行处理。let和const不会使变量提升，提前使用会报错。 变量提升（hoisting）是用于解释代码中变量声明行为的术语。使用var关键字声明或初始化的变量，会将声明语句“提升”到当前作用域的顶部。 但是，只有声明才会触发提升，赋值语句（如果有的话）将保持原样。

## 使用let、var和const创建变量有什么区别

用 var 声明的变量的作用域是它当前的执行上下文，它可以是嵌套的函数，也可以是声明在任何函数外的变量。let 和 const 是块级作用域，意味着它们只能在最近的一组花括号（function、if-else 代码块或 for 循环中）中访问。

var 声明的全局变量和函数都会成为 window 对象的属性和方法。使用 let 和 const 的顶级声明不会定义在全局上下文中，但在作用域链解析上效果是一样的。

function foo() {  
 // 所有变量在函数中都可访问  
 var bar = 'bar';  
 let baz = 'baz';  
 const qux = 'qux';  
  
 console.log(bar); // bar  
 console.log(baz); // baz  
 console.log(qux); // qux  
}  
  
console.log(bar); // ReferenceError: bar is not defined  
console.log(baz); // ReferenceError: baz is not defined  
console.log(qux); // ReferenceError: qux is not defined

if (true) {  
 var bar = 'bar';  
 let baz = 'baz';  
 const qux = 'qux';  
}  
  
// 用 var 声明的变量在函数作用域上都可访问  
console.log(bar); // bar  
// let 和 const 定义的变量在它们被定义的语句块之外不可访问  
console.log(baz); // ReferenceError: baz is not defined  
console.log(qux); // ReferenceError: qux is not defined

var会使变量提升，这意味着变量可以在声明之前使用。let和const不会使变量提升，提前使用会报错。

console.log(foo); // undefined  
  
var foo = 'foo';  
  
console.log(baz); // ReferenceError: can't access lexical declaration 'baz' before initialization  
  
let baz = 'baz';  
  
console.log(bar); // ReferenceError: can't access lexical declaration 'bar' before initialization  
  
const bar = 'bar';

用var重复声明不会报错，但let和const会。

var foo = 'foo';  
var foo = 'bar';  
console.log(foo); // "bar"  
  
let baz = 'baz';  
let baz = 'qux'; // Uncaught SyntaxError: Identifier 'baz' has already been declared

let和const的区别在于：let允许多次赋值，而const只允许一次。

// 这样不会报错。  
let foo = 'foo';  
foo = 'bar';  
  
// 这样会报错。  
const baz = 'baz';  
baz = 'qux';

## 对象浅拷贝和深拷贝有什么区别

在 JS 中，除了基本数据类型，还存在对象、数组这种引用类型。 基本数据类型，拷贝是直接拷贝变量的值，而引用类型拷贝的其实是变量的地址。

let o1 = {a: 1}  
let o2 = o1

在这种情况下，如果改变 o1 或 o2 其中一个值的话，另一个也会变，因为它们都指向同一个地址。

o2.a = 3  
console.log(o1.a) // 3

而浅拷贝和深拷贝就是在这个基础之上做的区分，如果在拷贝这个对象的时候，只对基本数据类型进行了拷贝，而对引用数据类型只是进行了引用的传递，而没有重新创建一个新的对象，则认为是浅拷贝。反之，在对引用数据类型进行拷贝的时候，创建了一个新的对象，并且复制其内的成员变量，则认为是深拷贝。

## 怎么实现对象深拷贝

这种方法有缺陷，详情请看[关于JSON.parse(JSON.stringify(obj))实现深拷贝应该注意的坑](https://www.jianshu.com/p/b084dfaad501)

let o1 = {a:{  
 b:1  
 }  
}  
let o2 = JSON.parse(JSON.stringify(o1))

基础版

function deepCopy(target) {  
 if (typeof target == 'object') {  
 const result = Array.isArray(target)? [] : {}  
 for (const key in target) {  
 if (typeof target[key] == 'object') {  
 result[key] = deepCopy(target[key])  
 } else {  
 result[key] = target[key]  
 }  
 }  
  
 return result  
 } else if (typeof target == 'function') {  
 return eval('(' + test.toString() + ')')  
 } else {  
 return target  
 }  
}

完整版

const mapTag = '[object Map]'  
const setTag = '[object Set]'  
const arrayTag = '[object Array]'  
const objectTag = '[object Object]'  
const symbolTag = '[object Symbol]'  
  
function deepCopy(origin, map = new WeakMap()) {  
 if (!origin || !isObject(origin)) return origin  
  
 const objType = getObjType(origin)  
 const result = createObj(origin, objType)  
  
 // 防止循环引用，不会遍历已经在 map 中的对象，因为在上一层正在遍历  
 if (map.get(origin)) {  
 return map.get(origin)  
 }  
  
 map.set(origin, result)  
  
 // set  
 if (objType == setTag) {  
 for (const value of origin) {  
 result.add(deepCopy(value, map))  
 }  
  
 return result  
 }  
  
 // map  
 if (objType == mapTag) {  
 for (const [key, value] of origin) {  
 result.set(key, deepCopy(value, map))  
 }  
  
 return result  
 }  
  
 // 对象或数组  
 if (objType == objectTag || objType == arrayTag) {  
 for (const key in origin) {  
 result[key] = deepCopy(origin[key], map)  
 }  
  
 return result  
 }  
  
 return result  
}  
  
function getObjType(obj) {  
 return Object.prototype.toString.call(obj)  
}  
  
function createObj(obj, type) {  
 if (type == objectTag) return {}  
 if (type == arrayTag) return []  
 if (type == symbolTag) return Object(Symbol.prototype.valueOf.call(obj))  
  
 return new obj.constructor(obj)  
}  
  
function isObject(origin) {  
 return typeof origin == 'object'  
}

[如何写出一个惊艳面试官的深拷贝?](https://juejin.im/post/5d6aa4f96fb9a06b112ad5b1)

## 数组去重

ES5

function unique(arr) {  
 const result = []  
 arr.forEach(function(item) {  
 if (result.indexOf(item) == -1) {  
 result.push(item)  
 }  
 })  
   
 return result  
}

ES6

function unique(arr) {  
 return Array.from(new Set(arr))  
}

空间换时间

function unique(arr) {  
 const result = []  
 const map = new Map()  
  
 for (const val of arr) {  
 if (!map.has(val)) {  
 result.push(val)  
 map.set(val, true)  
 }  
 }  
  
 return result  
}

## 数据类型

1. Undefined
2. Null
3. Boolean
4. Number
5. String
6. Object
7. Symbol
8. BigInt

## 内置函数(原生函数)

* String
* Number
* Boolean
* Object
* Function
* Array
* Date
* RegExp
* Error
* Symbol

原始值 "I am a string" 并不是一个对象，它只是一个字面量，并且是一个不可变的值。

如果要在这个字面量上执行一些操作，比如获取长度、访问其中某个字符等，那需要将其转换为 String 对象。

幸好，在必要时语言会自动把字符串字面量转换成一个 String 对象，也就是说你并不需要显式创建一个对象。

## 如何判断数组与对象

Array.isArray([]) // true  
Array.isArray({}) // false  
  
typeof [] // "object"  
typeof {} // "object"  
  
Object.prototype == [].\_\_proto\_\_ // false  
Object.prototype == {}.\_\_proto\_\_ // true  
Array.prototype == [].\_\_proto\_\_ // true  
Array.prototype == {}.\_\_proto\_\_ // false

## 自动分号

有时 JavaScript 会自动为代码行补上缺失的分号，即自动分号插入（Automatic SemicolonInsertion，ASI）。

因为如果缺失了必要的 ; ，代码将无法运行，语言的容错性也会降低。ASI 能让我们忽略那些不必要的 ; 。

请注意，ASI 只在换行符处起作用，而不会在代码行的中间插入分号。

如果 JavaScript 解析器发现代码行可能因为缺失分号而导致错误，那么它就会自动补上分号。并且，只有在代码行末尾与换行符之间除了空格和注释之外没有别的内容时，它才会这样做。

## 浮点数精度

https://www.css88.com/archives/7340

## cookie、localStorage、sessionStorage区别

|  |  |  |  |
| --- | --- | --- | --- |
| 特性 | cookie | localStorage | sessionStorage |
| 由谁初始化 | 客户端或服务器，服务器可以使用Set-Cookie请求头。 | 客户端 | 客户端 |
| 数据的生命周期 | 一般由服务器生成，可设置失效时间，如果在浏览器生成，默认是关闭浏览器之后失效 | 永久保存，可清除 | 仅在当前会话有效，关闭页面后清除 |
| 存放数据大小 | 4KB | 5MB | 5MB |
| 与服务器通信 | 每次都会携带在HTTP头中，如果使用cookie保存过多数据会带来性能问题 | 仅在客户端保存 | 仅在客户端保存 |
| 用途 | 一般由服务器生成，用于标识用户身份 | 用于浏览器缓存数据 | 用于浏览器缓存数据 |
| 访问权限 | 任意窗口 | 任意窗口 | 当前页面窗口 |

## 自执行函数?用于什么场景？好处?

#### 自执行函数:

1. 声明一个匿名函数
2. 马上调用这个匿名函数。

作用：创建一个独立的作用域。

#### 好处

* 防止变量弥散到全局，以免各种js库冲突。
* 隔离作用域避免污染，或者截断作用域链，避免闭包造成引用变量无法释放。
* 利用立即执行特性，返回需要的业务函数或对象，避免每次通过条件判断来处理。

#### 场景

一般用于框架、插件等场景

## 多个页面之间如何进行通信

有如下几个方式： \* cookie \* web worker \* localeStorage和sessionStorage

## css动画和js动画的差异

1. 代码复杂度，js 动画代码相对复杂一些
2. 动画运行时，对动画的控制程度上，js 能够让动画，暂停，取消，终止，css动画不能添加事件
3. 动画性能看，js 动画多了一个js 解析的过程，性能不如 css 动画好

https://zhuanlan.zhihu.com/p/41479807

## 如何实现文件断点续传

[字节跳动面试官：请你实现一个大文件上传和断点续传](https://juejin.cn/post/6844904046436843527)

## new一个对象经历了什么

function Test(){}  
const test = new Test()

1. 创建一个新对象：

* const obj = {}

1. 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.constructor = Test  
  obj.\_\_proto\_\_ = Test.prototype

1. 使用新对象调用函数，函数中的this被指向新实例对象

* Test.call(obj)

1. 将初始化完毕的新对象地址，保存到等号左边的变量中

## bind、call、apply的区别

call和apply其实是一样的，区别就在于传参时参数是一个一个传或者是以一个数组的方式来传。 call和apply都是在调用时生效，改变调用者的this指向。

let name = 'Jack'  
const obj = {name: 'Tom'}  
function sayHi() {console.log('Hi! ' + this.name)}  
  
sayHi() // Hi! Jack  
sayHi.call(obj) // Hi! Tom

bind也是改变this指向，不过不是在调用时生效，而是返回一个新函数。

const newFunc = sayHi.bind(obj)  
newFunc() // Hi! Tom

## 实现 bind call apply 函数

### bind

Function.prototype.bind = function(context, ...extra) {  
 const self = this  
 // 这里不能用箭头函数，防止绑定函数为构造函数  
 return function(...arg) {  
 return self.call(context, ...extra.concat(arg))  
 }  
}

### call

Function.prototype.call = function (context, ...args) {  
 if (context === null || context === undefined) {  
 context = window  
 }  
  
 context = context instanceof Object ? context : {}  
  
 let key = Math.random()  
 while (context[key]) {  
 key = Math.random()  
 }  
  
 context[key] = this  
 const result = context[key](...args)  
 delete context[key]  
  
 return result  
}

### apply

Function.prototype.apply = function (context, args = []) {  
 if (!Array.isArray(args)) throw '参数必须为数组'  
  
 if (context === null || context === undefined) {  
 context = window  
 }  
  
 context = context instanceof Object ? context : {}  
  
 let key = Math.random()  
 while (context[key]) {  
 key = Math.random()  
 }  
  
 context[key] = this  
 const result = context[key](...args)  
 delete context[key]  
  
 return result  
}

## 实现 compose 函数

compose（组合）函数特点： \* compose 的参数是函数，返回的也是一个函数 \* 因为除了第一个函数的接受参数，其他函数的接受参数都是上一个函数的返回值，所以初始函数的参数是多元的，而其他函数的接受值是一元的 \* compsoe 函数可以接受任意的参数，所有的参数都是函数，且执行方向是自右向左的，初始函数一定放到参数的最右面

function compose(...fns) {  
 let isFirst = true  
 return (...args) => {  
 return fns.reduceRight((result, fn) => {  
 if (!isFirst) return fn(result)  
 isFirst = false  
 return fn(...result)  
 }, args)  
 }  
}

测试

const greeting = (firstName, lastName) => 'hello, ' + firstName + ' ' + lastName  
const toUpper = str => str.toUpperCase()  
const fn = compose(toUpper, greeting)  
console.log(fn('jack', 'smith')) // HELLO, JACK SMITH

参考资料： \* [关于javascript函数式编程中compose的实现](https://segmentfault.com/a/1190000008394749)

## 请简述JavaScript中的this

this取值符合以下规则： 1. 在调用函数时使用new关键字，函数内的this是一个全新的对象。 2. 如果apply、call或bind方法用于调用、创建一个函数，函数内的 this 就是作为参数传入这些方法的对象。 3. 当函数作为对象里的方法被调用时，函数内的this是调用该函数的对象。比如当obj.method()被调用时，函数内的 this 将绑定到obj对象。 4. 如果调用函数不符合上述规则，那么this的值指向全局对象（global object）。浏览器环境下this的值指向window对象，但是在严格模式下('use strict')，this的值为undefined。 5. 如果符合上述多个规则，则较高的规则（1 号最高，4 号最低）将决定this的值。 6. 如果该函数是 ES2015 中的箭头函数，将忽略上面的所有规则，this被设置为它被创建时的上下文。

想获得更深入的解释，请查看[这篇文章](https://codeburst.io/the-simple-rules-to-this-in-javascript-35d97f31bde3)。

## ==和===的区别是什么

==是抽象相等运算符，而===是严格相等运算符。==运算符是在进行必要的类型转换后，再比较。===运算符不会进行类型转换，所以如果两个值不是相同的类型，会直接返回false。使用==时，可能发生一些特别的事情，例如：

1 == '1'; // true  
1 == [1]; // true  
1 == true; // true  
0 == ''; // true  
0 == '0'; // true  
0 == false; // true

如果你对==和===的概念不是特别了解，建议大多数情况下使用===

## 箭头函数和普通函数有什么区别

* 函数体内的this对象，就是定义时所在的对象，而不是使用时所在的对象，用call apply bind也不能改变this指向
* 不可以当作构造函数，也就是说，不可以使用new命令，否则会抛出一个错误。
* 不可以使用arguments对象，该对象在函数体内不存在。如果要用，可以用 rest 参数代替。
* 不可以使用yield命令，因此箭头函数不能用作 Generator 函数。
* 箭头函数没有原型对象prototype

## 白屏时间

白屏时间是指浏览器从输入网址，到浏览器开始显示内容的时间。

Performance 接口可以获取到当前页面中与性能相关的信息,该类型的对象可以通过调用只读属性 Window.performance 来获得。

performance.timing.navigationStart 是一个返回代表一个时刻的 unsigned long long 型只读属性，为紧接着在相同的浏览环境下卸载前一个文档结束之时的 Unix 毫秒时间戳。如果没有上一个文档，则它的值相当于 PerformanceTiming.fetchStart。

所以将以下脚本放在 </head> 前面就能获取白屏时间。

<script>  
 new Date() - performance.timing.navigationStart  
</script>

参考资料： \* [PerformanceTiming.navigationStart](https://developer.mozilla.org/zh-CN/docs/Web/API/PerformanceTiming/navigationStart)

## 当你在浏览器输入一个地址后发生了什么

[当···时发生了什么？](https://github.com/skyline75489/what-happens-when-zh_CN/blob/master/README.rst?utm_medium=social&utm_source=wechat_session&from=timeline&isappinstalled=0)

## 页面大量图片，如何优化加载，优化用户体验

1. 图片懒加载。在页面的未可视区域添加一个滚动事件，判断图片位置与浏览器顶端的距离与页面的距离，如果前者小于后者，优先加载。
2. 如果为幻灯片、相册等，可以使用图片预加载技术，将当前展示图片的前一张和后一张优先下载。
3. 如果图片为css图片，可以使用CSSsprite，SVGsprite等技术。
4. 如果图片过大，可以使用特殊编码的图片，加载时会先加载一张压缩的特别厉害的缩略图，以提高用户体验。
5. 如果图片展示区域小于图片的真实大小，应在服务器端根据业务需要先进行图片压缩，图片压缩后大小与展示一致。

## 防抖与节流

#### 防抖(debounce)

在函数需要频繁触发时，只有当有足够空闲的时间时，才执行一次。就好像在百度搜索时，每次输入之后都有联想词弹出，这个控制联想词的方法就不可能是输入框内容一改变就触发的，他一定是当你结束输入一段时间之后才会触发。

#### 节流(thorttle)

预定一个函数只有在大于等于执行周期时才执行，周期内调用不执行。就好像你在淘宝抢购某一件限量热卖商品时，你不断点刷新点购买，可是总有一段时间你点上是没有效果，这里就用到了节流，就是怕点的太快导致系统出现bug。

#### 区别

在发生持续触发事件时，防抖设置事件延迟并在空闲时间去触发事件，而节流则是隔一定的时间触发一次。

一个简单的防抖示例

let timer  
input.on('input', () => {  
 clearTimeout(timer)  
 // 停止输入 500 毫秒后开始搜索  
 timer = setTimeout(() => {  
 // 搜索  
 }, 500)  
})

一个简单的节流示例

let isClick = false  
button.on('click', () => {  
 if (isClick) return  
 isClick = true  
 // 其他代码。。。  
 // 每 10 秒只允许点击一次  
 setTimeout(() => {  
 isClick = false  
 }, 10000)  
})

来个困难点的，根据图片要求实现节流函数

实现

const throttle = (function(delay) {  
 let waitForCallFunc  
 let canCall = true  
 return function throttle(callback) {  
 if (!canCall) {  
 if (callback) waitForCallFunc = callback  
 return  
 }  
  
 callback()  
 canCall = false  
 setTimeout(() => {  
 canCall = true  
 if (waitForCallFunc) {  
 throttle(waitForCallFunc)  
 waitForCallFunc = null  
 }  
 }, delay)  
 }  
})(1000)  
  
throttle(() => console.log(1))  
  
setTimeout(() => {  
 throttle(() => console.log(2))  
}, 500)  
  
setTimeout(() => {  
 throttle(() => console.log(3))  
}, 700)  
  
setTimeout(() => {  
 throttle(() => console.log(4))  
}, 1200)  
  
setTimeout(() => {  
 throttle(() => console.log(5))  
}, 1400)  
  
setTimeout(() => {  
 throttle(() => console.log(6))  
}, 1600)  
  
setTimeout(() => {  
 throttle(() => console.log(7))  
}, 2500)

参考资料： \* [js网络请求性能优化之防抖与节流](https://blog.csdn.net/jacoox/article/details/80719456)

## 如何做到修改url参数页面不刷新

HTML5引入了 history.pushState() 和 history.replaceState() 方法，它们分别可以添加和修改历史记录条目。

let stateObj = {  
 foo: "bar",  
};  
  
history.pushState(stateObj, "page 2", "bar.html");

假设当前页面为 foo.html，执行上述代码后会变为 bar.html，点击浏览器后退，会变为 foo.html，但浏览器并不会刷新。 pushState() 需要三个参数: 一个状态对象, 一个标题 (目前被忽略), 和 (可选的) 一个 URL. 让我们来解释下这三个参数详细内容：

* 状态对象 — 状态对象 state 是一个 JavaScript 对象，通过 pushState () 创建新的历史记录条目。无论什么时候用户导航到新的状态，popstate 事件就会被触发，且该事件的 state 属性包含该历史记录条目状态对象的副本。 状态对象可以是能被序列化的任何东西。原因在于 Firefox 将状态对象保存在用户的磁盘上，以便在用户重启浏览器时使用，我们规定了状态对象在序列化表示后有640k的大小限制。如果你给 pushState() 方法传了一个序列化后大于 640k 的状态对象，该方法会抛出异常。如果你需要更大的空间，建议使用 sessionStorage 以及 localStorage.
* 标题 — Firefox 目前忽略这个参数，但未来可能会用到。传递一个空字符串在这里是安全的，而在将来这是不安全的。二选一的话，你可以为跳转的 state 传递一个短标题。
* URL — 该参数定义了新的历史URL记录。注意，调用 pushState() 后浏览器并不会立即加载这个 URL，但可能会在稍后某些情况下加载这个 URL，比如在用户重新打开浏览器时。新URL不必须为绝对路径。如果新URL是相对路径，那么它将被作为相对于当前 URL 处理。新 URL 必须与当前URL同源，否则 pushState() 会抛出一个异常。该参数是可选的，缺省为当前 URL。

参考资料： \* [History API](https://developer.mozilla.org/zh-CN/docs/Web/API/History_API)

## 格式化金钱，每千分位加逗号

function format(str) {  
 let s = ''  
 let count = 0  
 for (let i = str.length - 1; i >= 0; i--) {  
 s = str[i] + s  
 count++  
 if (count % 3 == 0 && i != 0) {  
 s = ',' + s  
 }  
 }  
 return s  
}

function format(str) {  
 return str.replace(/(\d)(?=(?:\d{3})+$)/g, '$1,')  
}

## 请用js去除字符串空格

### 去除所有空格

str.replace(/\s/g, '')

### 去除两边空格

str.replace(/^\s+|\s+$/g, '')  
// 原生方法  
str.trim()

## 创建对象有几种方法

* 字面量
* const obj = {a: 1}
* 构造函数 ```js function Obj(val) { this.a = val }

const obj = new Obj(1) \* Object.createjs const obj = Object.create({a: 1}) ```

## null和undefined的区别

null 表示一个对象是“没有值”的值，也就是值为“空”

undefined 表示一个变量声明了没有初始化(赋值)

undefined 和 null 在if语句中，都会被自动转为false

undefined 不是一个有效的JSON，而 null 是

undefined 的类型(typeof)是 undefined

null 的类型(typeof)是 object

Javascript将未赋值的变量默认值设为 undefined

Javascript从来不会将变量设为 null。 它是用来让程序员表明某个用var声明的变量时没有值的

## 反转数组

### 要求

**input**: I am a student **output**: student a am I 输入是数组 输出也是数组 不允许用 split splice reverse

#### 解法一

function reverseArry(arr) {  
 const result = []  
 while (arr.length) {  
 result.push(arr.pop())  
 }  
   
 return result  
}  
  
console.log(reverseArry(['I', 'am', 'a', 'student']))  
// ["student", "a", "am", "I"]

#### 解法二

function reverseArry(arry) {  
 const result = []  
 const distance = arry.length - 1  
 for (let i = distance; i >= 0; i--) {  
 result[distance - i] = arry[i]  
 }  
  
 return result  
}

## 将金额12345转成中文金额表示

### 要求

12345 => 一万两千三百四十五  
10086 => 一万零八十六  
100010001 => 一亿零一万零一  
100000000 => 一亿

单位支持到亿

### 实现

function numToString(num) {  
 if (num > 999999999) throw '超过金额上限，最大单位为亿'  
 const unitMap = ['', '十', '百', '千', '万', '十', '百', '千', '亿']  
 const stringMap = ['零', '一', '二', '三', '四', '五', '六', '七', '八', '九']  
 const n = num + ''  
 const len = n.length  
 let lastIndex = len - 1  
 let result = ''  
 for (let i = 0; i < len; i++) {  
 if (i > 0 && n[i] == '0') {  
 if (n[i - 1] != '0') result += '零'  
 } else {  
 result += stringMap[n[i]] + unitMap[lastIndex]  
 }  
  
 lastIndex--  
 }  
   
 lastIndex = result.length - 1  
 if (result[lastIndex] == '零') return result.slice(0, lastIndex)  
 return result  
}  
  
console.log(numToString(12345)) // 一万二千三百四十五  
console.log(numToString(10086)) // 一万零八十六  
console.log(numToString(100010001)) // 一亿零一万零一  
console.log(numToString(100000000)) // 一亿

## 异步求和

### 要求

提供一个异步 add 方法如下，需要实现一个 await sum(...args) 函数：

function asyncAdd(a, b, callback) {  
 setTimeout(function() {  
 callback(null, a + b)  
 }, 1000)  
}

### 实现

function sum(...args) {  
 let start = 0  
 let result = 0  
 let count = 0 // 用于计算开启了多少个 Promise  
  
 function \_sum(resolve) {  
 count++  
 new Promise((r, j) => {  
 let a = args[start++]  
 let b = args[start++]  
 a = a !== undefined? a : 0  
 b = b !== undefined? b : 0 // 如果访问的元素超出了数组范围，则转为 0  
 asyncAdd(a, b, (context, sum) => {  
 r(sum)  
 })  
  
 if (start < args.length) {  
 \_sum(resolve)  
 }  
 })  
 .then(sum => {  
 result += sum  
 count--  
 if (count == 0) resolve(result) // 所有的 Promise 执行完毕，返回结果  
 })  
 }  
  
 return new Promise((resolve, reject) => {  
 if (!args || !args.length) return resolve(0)  
 if (args.length == 1) return resolve(args[0])  
 \_sum(resolve)  
 })  
}

### 测试

sum(1,2,3,4,5,6,7,8,9,10,11).then(sum => console.log(sum)) // 66  
// or  
async function test() {  
 console.log(await sum(1,2,3,4,5,6,7,8,9,10,11))  
}  
  
test() // 66

## 异步求和升级版

### 要求

假设有一台本地机器，无法做加减乘除运算，因此无法执行 a + b、a+ = 1 这样的 JS 代码，然后我们提供一个服务器端的 HTTP API，可以传两个数字类型的参数，响应结果是这两个参数的和，这个 HTTP API 的 JS SDK（在本地机器上运行）的使用方法如下：

asyncAdd(3, 5, (err, result) => {  
 console.log(result); // 8  
});

模拟实现：

function asyncAdd(a, b, cb) {  
 setTimeout(() => {  
 cb(null, a + b);  
 }, Math.floor(Math.random()\*100))  
}

现在要求在本地机器上实现一个 sum 函数，支持以下用法：

(async () => {  
 const result1 = await sum(1, 4, 6, 9, 1, 4);  
 const result2 = await sum(3, 4, 9, 2, 5, 3, 2, 1, 7);  
 const result3 = await sum(1, 6, 0, 5);  
 console.log([result1, result2, result3]); // [25, 36, 12]  
})();

要求 sum 能在最短的时间里返回以上结果

### 实现

function asyncAdd(a, b, cb) {  
 setTimeout(() => {  
 cb(null, a + b);  
 }, Math.floor(Math.random()\*100))  
}  
  
function sum(...args) {  
 const result = []  
 function \_sum(resolve, reject) {  
 new Promise((r, j) => {  
 let a = args.pop()  
 let b = args.pop()  
 a = a !== undefined? a : 0  
 b = b !== undefined? b : 0 // 如果访问的元素超出了数组范围，则转为 0  
 asyncAdd(a, b, (err, res) => {  
 if (err) j(err)  
 r(res)  
 })  
  
 if (args.length) {  
 \_sum(resolve, reject)  
 }  
 })  
 .then(val => {  
 result.push(val)  
 setTimeout(() => {  
 if (args.length <= 0) {  
 resolve(sum(...result))  
 }  
 }, 100)  
 })  
 }  
  
 return new Promise((resolve, reject) => {  
 if (!args || !args.length) resolve(0)  
 if (args.length == 1) resolve(args[0])  
 \_sum(resolve, reject)  
 })  
}  
  
(async () => {  
 const result1 = await sum(1, 4, 6, 9, 1, 4)  
 const result2 = await sum(3, 4, 9, 2, 5, 3, 2, 1, 7)  
 const result3 = await sum(1, 6, 0, 5)  
 console.log([result1, result2, result3]) // [25, 36, 12]  
})()

## 数字集转换成字母集

### 要求

a~z 有 26个字母，按照 1~26 编码，现在给定一个数字字符串，输出所有可能的解码结果，如：输入 1234，输出 ['awd', 'abcd', 'lcd']

const map = [0,'a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p','q','r','s','t','u','v','w','x','y','z']  
function getDecodes(num) {  
 if (!num) return []  
 num += ''  
 const result = []  
 \_getDecodes(num, 0, [], result)  
 return result  
}  
  
function \_getDecodes(num, start, path, result) {  
 if (start == num.length) return result.push([...path])  
 let c = num[start++]  
 path.push(map[c])  
 \_getDecodes(num, start, path, result)  
 path.pop()  
   
 if (start == num.length) return  
 c += num[start]  
  
 if (c > 26) return  
 path.push(map[c])  
 \_getDecodes(num, start + 1, path, result)  
 path.pop()  
}

## CommonJS，ES module 是什么，有什么区别？

它们都是一种模块规范，例如 Node 使用的就是 CommonJS 规范。ES module 则是语言标准上的模块规范。

区别： 1. CommonJS 模块使用 require() 和 module.exports，ES6 模块使用 import和 export。 2. CommonJS 模块输出的是一个值的浅拷贝，ES6 模块输出的是值的引用。 3. CommonJS 模块是运行时加载，ES6 模块是编译时输出接口。 4. CommonJS 模块的 require() 是同步加载模块，ES6 模块的 import 命令是异步加载，有一个独立的模块依赖的解析阶段。 5. ES6 模块之中，顶层的 this 指向 undefined；CommonJS 模块的顶层 this 指向当前模块， 6. 对于循环加载的处理方法不同

第 3 个差异是因为 CommonJS 加载的是一个对象（即 module.exports 属性），该对象只有在脚本运行完才会生成。

而 ES6 模块不是对象，它的对外接口只是一种静态定义，在代码静态解析阶段就会生成。

参考资料： \* [Module 的加载实现](https://es6.ruanyifeng.com/#docs/module-loader)

## preload和prefetch

### preload

preload 是 <link> 标签 rel 属性的属性值，同时需要配合 as 属性使用。

as 指定将要预加载的内容的类型，使得浏览器能够： 1. 更精确地优化资源加载优先级。 2. 匹配未来的加载需求，在适当的情况下，重复利用同一资源。 3. 为资源应用正确的内容安全策略。 4. 为资源设置正确的 Accept 请求头。

看一下这个示例：

<link rel="preload" href="https://cdn.jsdelivr.net/npm/vue/dist/vue.js" as="script">

这种做法将把 <link> 标签塞入一个预加载器中。

这个预加载器在不阻塞页面 onload 事件的情况下，去加载资源。

我们可以通过以下两个示例来作一个对比：

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>Document</title>  
 <script>  
 console.time('load')  
 </script>  
 <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>  
 <script src="https://cdn.bootcdn.net/ajax/libs/echarts/2.1.10/chart/bar.js"></script>  
 <script src="https://unpkg.com/element-ui/lib/index.js"></script>  
 <link rel="stylesheet" href="https://unpkg.com/element-ui/lib/theme-chalk/index.css">  
</head>  
<body>  
<script>  
window.onload = () => {  
 console.timeEnd('load') // load: 1449.759033203125ms  
}  
</script>  
</body>  
</html>

上面这个示例从加载到触发 onload 事件需要大概 1400 ms 的时间。再看一下使用 preload 预加载的时间：

<link rel="preload" href="https://cdn.jsdelivr.net/npm/vue/dist/vue.js" as="script">  
<link rel="preload" href="https://cdn.bootcdn.net/ajax/libs/echarts/2.1.10/chart/bar.js" as="script">  
<link rel="preload" href="https://unpkg.com/element-ui/lib/index.js" as="script">  
<link rel="preload" href="https://unpkg.com/element-ui/lib/theme-chalk/index.css" as="style">  
  
window.onload = () => {  
 console.timeEnd('load') // load: 10.8818359375ms  
}

用 preload 来加载资源，只需要 10 ms 就触发了 onload 事件。

说明同样是下载文件，使用 preload 不会阻塞 onload 事件。

### prefetch

prefetch 和 preload 不同，使用 prefetch 属性指定的资源将在浏览器空闲时间下下载。

在资源的请求头如果发现有下面这个属性，就代表它是通过 prefetch 加载的：

purpose: prefetch

另外，空闲时间是如何确定、如何获取的，目前还没有相关 API。

## preload 和 defer 的区别

preload 和 defer 的相同点是异步下载。那它们的不同点是什么呢？

preload 下载的资源只有在遇到同样的 script 标签时，才会执行对应的脚本。例如下面预加载的 vue.js：

<link rel="preload" as="script" href="https://cdn.jsdelivr.net/npm/vue/dist/vue.js">

只有在遇到下面的标签时，才会执行加载的 vue.js：

<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>

defer 则是异步下载资源，在所有元素解析完成后，触发 DOMContentLoaded 事件前执行。

## window.onload 和 DOMContentLoaded 的区别

当整个页面及所有依赖资源如样式表和图片都已完成加载时，将触发load事件。

它与 DOMContentLoaded不同，当纯HTML被完全加载以及解析时，DOMContentLoaded 事件会被触发，而不必等待样式表，图片或者子框架完成加载。

* [load](https://developer.mozilla.org/zh-CN/docs/Web/Events/load)
* [DOMContentLoaded 事件](https://developer.mozilla.org/zh-CN/docs/Web/API/Document/DOMContentLoaded_event)

## websocket 鉴权、多人连接、心跳机制

* [WebSocket 的鉴权授权方案](http://www.moye.me/2017/02/10/websocket-authentication-and-authorization/)
* [WebSocket学习（一）——基于socket.io实现简单多人聊天室](https://segmentfault.com/a/1190000011538416)
* [理解WebSocket心跳及重连机制（五）](https://www.cnblogs.com/tugenhua0707/p/8648044.html)

## Object 与 Map 的区别

1. Object 只能选择字符、数值、符号作为 key，Map 则可以使用任何类型的数据作为 key。
2. Map 实例会维护键值对的插入顺序，因此可以根据插入顺序执行迭代操作。Chrome Opera 中使用 for-in 语句遍历 Object 属性时会遵循一个规律：它们会先提取所有 key 的 parseFloat 值为非负整数的属性，然后根据数字顺序对属性排序首先遍历出来，然后按照对象定义的顺序遍历余下的所有属性。其它浏览器则完全按照对象定义的顺序遍历属性。

### 选择 Object 还是 Map

对于多数Web开发任务来说，选择 Object 还是 Map 只是个人偏好问题，影响不大。不过，对于在乎内存和性能的开发者来说，对象和映射之间确实存在显著的差别。 #### 1. 内存占用 Object 和 Map 的工程级实现在不同浏览器间存在明显差异，但存储单个键/值对所占用的内存数量都会随键的数量线性增加。批量添加或删除键/值对则取决于各浏览器对该类型内存分配的工程实现。

不同浏览器的情况不同，但给定固定大小的内存， Map 大约可以比 Object 多存储50%的键/值对。 #### 2. 插入性能 向 Object 和 Map 中插入新键/值对的消耗大致相当，不过插入Map 在所有浏览器中一般会稍微快一点儿。对这两个类型来说，插入速度并不会随着键/值对数量而线性增加。

如果代码涉及大量插入操作，那么显然 Map 的性能更佳。 #### 3. 查找速度 与插入不同，从大型 Object 和 Map 中查找键/值对的性能差异极小，但如果只包含少量键/值对，则 Object 有时候速度更快。在把 Object 当成数组使用的情况下（比如使用连续整数作为属性），浏览器引擎可以进行优化，在内存中使用更高效的布局。

这对 Map 来说是不可能的。对这两个类型而言，查找速度不会随着键/值对数量增加而线性增加。如果代码涉及大量查找操作，那么某些情况下可能选择 Object 更好一些。 #### 4. 删除性能 使用 delete 删除 Object 属性的性能一直以来饱受诟病，目前在很多浏览器中仍然如此。为此，出现了一些伪删除对象属性的操作，包括把属性值设置为 undefined 或 null 。但很多时候，这都是一 种讨厌的或不适宜的折中。

而对大多数浏览器引擎来说， Map 的 delete() 操作都比插入和查找更快。如果代码涉及大量删除操作，那么毫无疑问应该选择 Map 。

参考资料： \* [JavaScript高级程序设计（第4版）](https://book.douban.com/subject/35175321/?from=tag) \* [js能够保证object属性的输出顺序吗？](http://jartto.wang/2016/10/25/does-js-guarantee-object-property-order/)

## 为什么 WeakMap 和 WeakSet 的键只能使用对象？

是为了保证只有通过键对象的引用来取得值。

const m = new WeakMap()  
m.set({}, 100) // 由于 {} 没有在其他地方引用，所以在垃圾回收时，这个值也会被回收。  
  
const a = {}  
m.set(a, 100) // 如果使用这种方式，则不会被回收。因为 {} 有 a 变量在引用它。  
  
a = null // 将 a 置为空后，m 里的值 100 在垃圾回收时将会被回收。

如果允许原始值，那就没办法区分初始化时使用的字符串字面量和初始化之后使用的一个相等的字符串了。

所以这句话的意思很明确：

const a = {} // 在创建对象时，分配了一块内存，并把这块内存的地址传给 a   
m.set(a, 100) // 执行 set 操作时，实际上是将 a 指向的内存地址和 100 关联起来  
  
const a = 'abc' // 由于基本数据类型在传递时，传递的是值，而不是引用。  
m.set(a, 100) // 所以执行 set 操作时，实际上是将新的 'abc' 和 100 关联起来，而不是原来 a 变量指向的那个。  
 // 那这样就会有问题，m 里存储的永远是没有被引用的键，随时都会被回收。

参考资料： \* [JavaScript高级程序设计（第4版）](https://book.douban.com/subject/35175321/?from=tag)

## 实现 async/await

利用 generator() 实现 async/await 主要就是用一个函数（自动执行器）来包装 generator()，从而实现自动执行 generator()。

每次执行 next() 返回的 { value, done } 中的 value 是一个 Promise，所以要等它执行完毕后再执行下一次 next()。

即在它的后面加一个 then() 函数，并且在 then() 函数中执行 next()。

function t(data) {  
 return new Promise(r => setTimeout(() => r(data), 100))  
}  
  
function \*test() {  
 const data1 = yield t(1)  
 console.log(data1)  
 const data2 = yield t(2)  
 console.log(data2)  
 return 3  
}  
  
function async(generator) {  
 return new Promise((resolve, reject) => {  
 const gen = generator()  
  
 function step(nextFun) {  
 // 每一次 next() 都是返回这样的数据 { value: xx, done: false }，结束时 done 为 true  
 let next  
 try {  
 // 如果 generator() 执行报错，需要 reject 给外面的 catch 函数  
 next = nextFun()  
 } catch(e) {  
 return reject(e)  
 }  
  
 // done: true 代表 generator() 结束了  
 if (next.done) {  
 return resolve(next.value)  
 }  
  
 Promise.resolve(next.value).then(  
 (val) => step(() => gen.next(val)), // 通过 next(val) 将 val 传给 yield 后面的变量   
 (err) => step(() => gen.trhow(err)),  
 )  
 }  
  
 step(() => gen.next())  
 })  
}  
  
// 1 2 3  
async(test).then(val => console.log(val))

## 实现发布/订阅模式

class Event {  
 constructor() {  
 this.events = {}  
 }  
  
 on(event, callback) {  
 if (!this.events[event]) {  
 this.events[event] = []  
 }  
  
 this.events[event].push(callback)  
 }  
  
 off(event, callback) {  
 if (this.events[event]) {  
 if (callback) {  
 const cbs = this.events[event]  
 let l = cbs.length  
 while (l--) {  
 if (callback == cbs[l]) {  
 cbs.splice(l, 1)  
 }  
 }  
 } else {  
 this.events[event] = []  
 }  
 }  
 }  
  
 emit(event, ...args) {  
 if (this.events[event]) {  
 for (const func of this.events[event]) {  
 func.call(this, ...args)  
 }  
 }  
 }  
  
 once(event, callback) {  
 const self = this  
  
 function wrap(...args) {  
 callback.call(self, ...args)  
 self.off(event, wrap)  
 }  
  
 this.on(event, wrap)  
 }  
}

## 5个fetch请求，请求完成后要求立即执行，但最终的输出顺序要按照要求输出ABCDE

function run(fetchs = []) {  
 return new Promise((resolve, reject) => {  
 const result = new Array(fetchs.length).fill(null)  
 fetchs.forEach((fetch, i) => {  
 fetch  
 .then(res => {  
 result[i] = res  
 if (!result.includes(null)) {  
 resolve(result)  
 }  
 })  
 .catch(err => {  
 reject(err)  
 })  
 })  
 })  
}  
  
function delay(str) {  
 return new Promise(resolve => {  
 setTimeout(() => {  
 resolve(str)  
 }, Math.random() \* 1000)  
 })  
}  
  
run([delay('A'), delay('B'), delay('C'), delay('D'), delay('E')]).then(res => {  
 console.log(res) // ["A", "B", "C", "D", "E"]  
})

## requestIdleCallback 是干什么用的

[你应该知道的requestIdleCallback](https://juejin.cn/post/6844903592831238157)

## js 对象循环引用会导致什么问题

1. 引用计数无法回收内存
2. JSON.stringify() 执行会报错。

## 如何埋点，为什么用1 \* 1像素的gif图片做上报

1. 能够完成整个 HTTP 请求+响应（尽管不需要响应内容）
2. 触发 GET 请求之后不需要获取和处理数据、服务器也不需要发送数据
3. 没有跨域问题
4. 执行过程无阻塞
5. 相比 XMLHttpRequest 对象发送 GET 请求，性能上更好
6. GIF的最低合法体积最小（最小的BMP文件需要74个字节，PNG需要67个字节，而合法的GIF，只需要43个字节）

[为什么通常在发送数据埋点请求的时候使用的是 1x1 像素的透明 gif 图片](https://github.com/Advanced-Frontend/Daily-Interview-Question/issues/87)

## Base64图片有什么问题

1. 使用base64 编码后比原图大
2. 内联到 HTML/CSS 文件，会造成文件尺寸变大，影响首屏加载。如果用外链图片的话，图片可以在页面渲染完成后继续加载，不会造成阻塞。

## 如何实现一个可设置过期时间的 localStorage

(function () {  
 const getItem = localStorage.getItem.bind(localStorage)  
 const setItem = localStorage.setItem.bind(localStorage)  
 const removeItem = localStorage.removeItem.bind(localStorage)  
  
 localStorage.getItem = function (key) {  
 const expires = getItem(key + '\_expires')  
 if (expires && new Date() > new Date(Number(expires))) {  
 removeItem(key)  
 removeItem(key + '\_expires')  
 }  
  
 return getItem(key)  
 }  
  
 localStorage.setItem = function (key, value, time) {  
 if (typeof time !== 'undefined') {  
 setItem(key + '\_expires', new Date().getTime() + Number(time))  
 }  
  
 return setItem(key, value)  
 }  
})()

## JavaScript的sort方法内部使用的什么排序

sort 使用的是插入排序和快速排序结合的排序算法。

数组长度不超过10时，使用插入排序。长度超过10使用快速排序。在数组较短时插入排序更有效率。

## 服务端渲染和预渲染的区别

[服务端渲染(SSR)和预渲染(Prerendering)有什么区别？](https://www.zhihu.com/question/273930443)

## canvas 和 svg 区别

[SVG 与 HTML5 的 canvas 各有什么优点，哪个更有前途？](https://www.zhihu.com/question/19690014)