目录

[JS 4](#_Toc11255)

[● 基本类型 4](#_Toc18677)

[● Symbol 4](#_Toc20395)

[● ==和=== 4](#_Toc1129)

[● 变量 4](#_Toc9661)

[● es6 5](#_Toc29833)

[● var、变量提升 5](#_Toc23424)

[● let 、const，暂时性死区 5](#_Toc10619)

[● 0.1+0.2 为什么不等于 0.3 5](#_Toc25)

[● 正则 6](#_Toc7939)

[● 事件委托 / 事件代理 7](#_Toc27813)

[● proxy 7](#_Toc3211)

[● js反射机制/Reflect 9](#_Toc15014)

[● 内存泄漏 9](#_Toc20067)

[● 垃圾回收 10](#_Toc8878)

[● ajax 11](#_Toc31282)

[● jsonp 11](#_Toc11691)

[● js常用输入输出语句 11](#_Toc32676)

[● require和import的区别 12](#_Toc13627)

[JS this关键字 13](#_Toc32733)

[● this 13](#_Toc21488)

[● this 指向 14](#_Toc6463)

[● js严格模式下this指向 14](#_Toc11376)

[● 箭头函数 14](#_Toc4724)

[● call，apply，bind 15](#_Toc26320)

[JS 继承 16](#_Toc26352)

[● 构造函数定义和使用 16](#_Toc32165)

[● 原型链、proto 和 prototype 16](#_Toc2106)

[● 原型链继承 16](#_Toc144)

[● 构造函数继承 17](#_Toc700)

[● 组合继承 18](#_Toc24670)

[● 原型式继承 19](#_Toc27125)

[● 寄生式继承 19](#_Toc14575)

[● 寄生组合式继承 20](#_Toc20275)

[● ES6 class继承 21](#_Toc28541)

[JS 事件循环 22](#_Toc24398)

[● 事件循环 、macrotask, microtask 22](#_Toc28670)

[● 事件循环——执行栈、调用栈 23](#_Toc26355)

[● js实现异步操作原理 23](#_Toc27318)

[● 什么是异步编程 24](#_Toc14970)

[● promise 26](#_Toc15154)

[● 回调函数，promise，await，generator 区别 27](#_Toc9581)

[JS 函数 29](#_Toc11332)

[● 函数和方法 29](#_Toc26394)

[● 纯函数 30](#_Toc10424)

[● 立即执行函数 30](#_Toc11500)

[● 柯里化 31](#_Toc9946)

[● new 对象的过程 31](#_Toc16480)

[● 闭包 32](#_Toc19720)

[● 防抖节流 32](#_Toc3539)

[● 深浅拷贝 34](#_Toc20157)

[● sort 方法，根据对象字段排序 34](#_Toc29477)

[● 手写 jsonp 34](#_Toc6608)

[● 手写 ajax 34](#_Toc19222)

[JS 设计模式 34](#_Toc25764)

[● MVVM 和 MVC 34](#_Toc30795)

[● 发布订阅模式、观察者模式 34](#_Toc8976)

[● 设计模式 34](#_Toc7556)

[JS 数据结构/方法 35](#_Toc31251)

[● Map 35](#_Toc3090)

[● map 和 set 35](#_Toc31937)

[● array.sort()方法失效 35](#_Toc7582)

[● for in 和 for of 36](#_Toc28028)

[● push、shift 36](#_Toc1268)

[● forEach 和 map 36](#_Toc3592)

[● slice 和 splice 和 split 37](#_Toc5704)

[● substr 和 substring 37](#_Toc7489)

[● 数组方法 reduce，some，find 37](#_Toc19096)

[● typeof null 37](#_Toc22394)

[● typeof 和 instanceof 37](#_Toc7952)

[● 判断数组 、typeof 和 instanceof 37](#_Toc14026)

[● 自定义事件监听 37](#_Toc14492)

[● eventEmitter 38](#_Toc23253)

[● PWA 38](#_Toc13585)

[● jsBridge 38](#_Toc31702)

# JS

* 基本类型

[面试官：说说JavaScript中的数据类型？区别？ · Issue #51 · febobo/web-interview (github.com)](https://github.com/febobo/web-interview/issues/51)

|  |
| --- |
| JS 中有六种简单数据类型：undefined、null、boolean、string、number、symbol，以及引用类型：object |

* Symbol

[Symbol - JavaScript | MDN (mozilla.org)](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Symbol)

|  |
| --- |
| **symbol** 是一种基本数据类型（[primitive data type](https://developer.mozilla.org/zh-CN/docs/Glossary/Primitive)）。Symbol() 函数会返回 **symbol** 类型的值，该类型具有静态属性和静态方法。它的静态属性会暴露几个内建的成员对象；它的静态方法会暴露全局的 symbol 注册，且类似于内建对象类，但作为构造函数来说它并不完整，因为它不支持语法："new Symbol()"。  每个从 Symbol() 返回的 symbol 值都是唯一的。一个 symbol 值能作为对象属性的标识符；这是该数据类型仅有的目的。 |

* ==和===

[JavaScript - Equality (==) vs. Identity (===) Operators (rumenz.com)](https://rumenz.com/java-topic/javascript/javascript-equality-vs-identity-operators/index.html)

|  |
| --- |
| 在执行比较时， ****相等运算符（**==**）将****在继续操作之前****尝试使数据类型相同**** 。 另一方面， ****身份运算符（**===**）要求这两种数据类型必须相同，作为前提**** 。  使用了===运算符，并且由于该运算符未进行任何类型转换，因此我们看到字符串值"3"和数字3毕竟是不同的。 |
| var a = {'a':'a','aa':'aa','aaa':'aaa'} var b = {'b':'b','bb':'bb','bbb':'bbb'} console.log(a instanceof Object) console.log(b instanceof Object)  console.log(a === b)//false  var c = {'a':'a','aa':'aa','aaa':'aaa'} console.log(a === c)//false  c = a console.log(a === c)//true  c = b console.log(b === c)//true  console.log(null === undefined)//false |

* 变量

|  |
| --- |
| javascript声明变量的4种方法：   1. 使用“var”声明变量，例“var a;”；   2、使用“function”声明变量，例“function Fun (num) {}”；  3、使用“let”声明变量；  4、使用“const”声明变量。 |

* es6

[JavaScript ES6 (w3school.com.cn)](https://www.w3school.com.cn/js/js_es6.asp)

|  |
| --- |
| * JavaScript let * JavaScript const * 幂 (\*\*) * 默认参数值。ES6 允许函数参数具有默认值。 * Array.find(value,index,array) * Array.findIndex(value,index,array) * 箭头函数允许使用简短的语法来编写函数表达式。 * ES6 中，通过 class 关键字来定义类，子类可以通过 extends 继承父类。 * promise |

* var、变量提升

[浅谈 JavaScript 变量提升 - 掘金 (juejin.cn)](https://juejin.cn/post/7007224479218663455)

|  |
| --- |
| JavaScript 和其他语言一样，都要经历编译和执行阶段。在这个短暂的****编译阶段****，JS 引擎会搜集所有的变量声明，并且提前让声明生效。而剩下的语句需要等到执行阶段、等到执行到具体的某一句时才会生效。这就是变量提升背后的机制。  在 ES6 之前，作用域分为两种：   * ****全局作用域****中的对象在代码中的任何地方都可以访问，其生命周期伴随着页面的生命周期。 * ****函数作用域****是在函数内部定义的变量或者函数，并且定义的变量或者函数只能在函数内部被访问。函数执行结束之后，函数内部定义的变量会被销毁。 |

* let 、const，暂时性死区

[ES6 中 let 暂时性死区详解 - SegmentFault 思否](https://segmentfault.com/a/1190000015603779" \l ":~:text=ES6%E8%A7%84%E5%AE%9A%EF%BC%8Clet/const %E5%91%BD%E4%BB%A4%E4%BC%9A%E4%BD%BF%E5%8C%BA%E5%9D%97%E5%BD%A2%E6%88%90%E5%B0%81%E9%97%AD%E7%9A%84%E4%BD%9C%E7%94%A8%E5%9F%9F%E3%80%82 %E8%8B%A5%E5%9C%A8%E5%A3%B0%E6%98%8E%E4%B9%8B%E5%89%8D%E4%BD%BF%E7%94%A8%E5%8F%98%E9%87%8F%EF%BC%8C%E5%B0%B1%E4%BC%9A%E6%8A%A5%E9%94%99%E3%80%82 %E6%80%BB%E4%B9%8B%EF%BC%8C%E5%9C%A8%E4%BB%A3%E7%A0%81%E5%9D%97%E5%86%85%EF%BC%8C%E4%BD%BF%E7%94%A8 let %E5%91%BD%E4%BB%A4%E5%A3%B0%E6%98%8E%E5%8F%98%E9%87%8F%E4%B9%8B%E5%89%8D%EF%BC%8C%E8%AF%A5%E5%8F%98%E9%87%8F%E9%83%BD%E6%98%AF%E4%B8%8D%E5%8F%AF%E7%94%A8%E7%9A%84%E3%80%82 %E8%BF%99%E5%9C%A8%E8%AF%AD%E6%B3%95%E4%B8%8A%EF%BC%8C%E7%A7%B0%E4%B8%BA %E2%80%9C%E6%9A%82%E6%97%B6%E6%80%A7%E6%AD%BB%E5%8C%BA%E2%80%9D%EF%BC%88,%E4%B9%9F%E6%84%8F%E5%91%B3%E7%9D%80 typeof %E4%B8%8D%E5%86%8D%E6%98%AF%E4%B8%80%E4%B8%AA%E7%99%BE%E5%88%86%E4%B9%8B%E7%99%BE%E5%AE%89%E5%85%A8%E7%9A%84%E6%93%8D%E4%BD%9C%E3%80%82 %E5%9C%A8%E6%B2%A1%E6%9C%89 let %E4%B9%8B%E5%89%8D%EF%BC%8C typeof %E8%BF%90%E7%AE%97%E7%AC%A6%E6%98%AF%E7%99%BE%E5%88%86%E4%B9%8B%E7%99%BE%E5%AE%89%E5%85%A8%E7%9A%84%EF%BC%8C%E4%B8%8D%E4%BC%9A%E6%8A%A5%E9%94%99%E3%80%82)

|  |
| --- |
| ES6规定，let/const 命令会使区块形成封闭的作用域。若在声明之前使用变量，就会报错。 总之，在代码块内，使用 let 命令声明变量之前，该变量都是不可用的。 这在语法上，称为 ****“暂时性死区”****（ temporal dead zone，简称 ****TDZ****）  TDZ 也意味着 typeof 不再是一个百分之百安全的操作  typeof a // ReferenceError: a is not defined |

* 0.1+0.2 为什么不等于 0.3

[(4条消息) 面试官提问js篇3：js中 0.1 + 0.2 为什么不等于0.3？怎么才能等于0.3？\_咖啡壶子的博客-CSDN博客](https://blog.csdn.net/TroyeSivanlp/article/details/121439928)

|  |
| --- |
| js中的Number类型遵循IEEE754标准，在IEEE754标准的64位浮点数相加，因为浮点数自身小数位数的限制而截断的二进制在转化为十进制，就变成0.30000000000000004，所以在计算时会产生误差。 |
| 第一种办法：  使用Math类中的toFixed()方法，保留你需要的小数点位数。  第二种方法：  Number.EPSILON可以用来设置“能够接受的误差范围”。比如，误差范围设为 2 的-52 次方（即Number.EPSILON \* Math.pow(2, -52)），即如果两个浮点数的差小于这个值，我们就认为这两个浮点数相等。  Number.EPSILON=(function(){  //解决兼容性问题（IE10不兼容）  return Number.EPSILON?Number.EPSILON:Math.pow(2,-52);  })();    //上面是一个自调用函数，当JS文件刚加载到内存中，就会去判断并返回一个结果，相比  // if(!Number.EPSILON){  // Number.EPSILON=Math.pow(2,-52);  //}  //这种代码更节约性能，也更美观。  function numbersequal(a,b){  return Math.abs(a-b)<Number.EPSILON;  }  //接下来再判断  var a=0.1+0.2, b=0.3;  console.log(numbersequal(a,b)); //这里就为true了 |

* 正则

[JavaScript 正则表达式 | 菜鸟教程 (runoob.com)](https://www.runoob.com/js/js-regexp.html)

[js正则表达式及练习（超详细）\_小生听雨园的博客-CSDN博客\_js正则表达式练习](https://blog.csdn.net/weixin_44154094/article/details/112685412)

|  |  |
| --- | --- |
| **修饰符** | **描述** |
| i | 执行对大小写不敏感的匹配。 |
| g | 执行全局匹配（查找所有匹配而非在找到第一个匹配后停止）。 |
| m | 执行多行匹配。 |

**^ 脱字符：**匹配开头，若存在多行匹配多行的行头。

**$ 美元符：**匹配尾部，若存在多行匹配多行的尾部。

|  |  |
| --- | --- |
| **量词** | **描述** |
| n+ | 匹配任何包含至少一个 n 的字符串。 |
| n\* | 匹配任何包含零个或多个 n 的字符串。 |
| n? | 匹配任何包含零个或一个 n 的字符串。 |
| **元字符** | **描述** |
| \d | 查找数字。 |
| \s | 查找空白字符。 |
| \b | 匹配单词边界。 |

* 事件委托 / 事件代理

[JS事件委托（事件代理） (biancheng.net)](http://c.biancheng.net/view/9380.html)

[面试官：解释下什么是事件代理？应用场景？ · Issue #66 · febobo/web-interview (github.com)](https://github.com/febobo/web-interview/issues/66)

|  |  |
| --- | --- |
| 事件代理，俗地来讲，就是把一个元素响应事件（click、keydown......）的函数委托到另一个元素。  使用事件委托，在动态绑定事件的情况下是可以减少很多重复工作的 | |
| <ul id="list">  <li>item 1</li>  <li>item 2</li>  <li>item 3</li>  ......  <li>item n</li>  </ul> | // 给父层元素绑定事件document.getElementById('list').addEventListener('click', function (e) {  // 兼容性处理  var event = e || window.event;  var target = event.target || event.srcElement;  // 判断是否匹配目标元素  if (target.nodeName.toLocaleLowerCase === 'li') {  console.log('the content is: ', target.innerHTML);  }}); |

* proxy

[JavaScript之Proxy - 简书 (jianshu.com)](https://www.jianshu.com/p/f0f8ae715f9b)

[Proxy - JavaScript | MDN (mozilla.org)](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Proxy)

[面试官：你是怎么理解ES6中Proxy的？使用场景? · Issue #42 · febobo/web-interview (github.com)](https://github.com/febobo/web-interview/issues/42)

|  |
| --- |
| Proxy代理是一个共通的概念，可以起到拦截的作用。ES6里将Proxy标准化了，提供了Proxy构造函数，用来生成Proxy实例。例如var p = new Proxy(target, handler);。  构造函数有两个参数，第一个参数target是要拦截的对象，第二个参数是拦截函数对象。  var handler = {  get: function(target, name){  return name in target ? target[name] : 'No prop!';  }};  var p = new Proxy({}, handler);  p.a = 1;  p.b = 2;  console.log(p.a); //1  console.log(p.b); //2  console.log(p.c); //No prop!  handler.getPrototypeOf(target)可以拦截取对象的原型对象的行为：  参数target即想获取它原型对象的对象。返回值是返回该原型对象或null。  var proto = {};  var p = new Proxy({}, {  getPrototypeOf(target) {  return proto;  }});  console.log(Object.getPrototypeOf(p) === proto); // true  handler.setPrototypeOf(target, prototype)可以拦截变更对象的原型对象的行为：  参数target是目标对象，参数prototype是给目标对象设置的原型对象或null。返回值如果目标对象的原型对象被成功改变，返回true，否则返回false。  var handler = {  setPrototypeOf (target, prototype) {  return false;  }  };  var newProto = {};  var p = new Proxy({}, handler);  console.log(Object.setPrototypeOf(p, newProto));//TypeError: proxy setPrototypeOf handler returned false  console.log(Reflect.setPrototypeOf(p, newProto)); //false |

* js反射机制/Reflect

[JS 反射机制及 Reflect 详解 - Leophen - 博客园 (cnblogs.com)](https://www.cnblogs.com/Leophen/p/14838608.html" \l ":~:text=Reflect %E6%98%AF%E4%B8%80%E4%B8%AA%E5%86%85%E5%BB%BA%E7%9A%84%E5%AF%B9%E8%B1%A1%EF%BC%8C%E7%94%A8%E6%9D%A5%E6%8F%90%E4%BE%9B%E6%96%B9%E6%B3%95%E5%8E%BB%E6%8B%A6%E6%88%AA JavaScript %E7%9A%84%E6%93%8D%E4%BD%9C%E3%80%82 Reflect,%E4%B8%8D%E6%98%AF%E4%B8%80%E4%B8%AA%E5%87%BD%E6%95%B0%E5%AF%B9%E8%B1%A1%EF%BC%8C%E6%89%80%E4%BB%A5%E5%AE%83%E6%98%AF%E4%B8%8D%E5%8F%AF%E6%9E%84%E9%80%A0%E7%9A%84%EF%BC%8C%E4%B9%9F%E5%B0%B1%E6%98%AF%E8%AF%B4%E5%AE%83%E4%B8%8D%E6%98%AF%E4%B8%80%E4%B8%AA%E6%9E%84%E9%80%A0%E5%99%A8%EF%BC%8C%E4%B8%8D%E8%83%BD%E9%80%9A%E8%BF%87 new %E6%93%8D%E4%BD%9C%E7%AC%A6%E5%8E%BB%E6%96%B0%E5%BB%BA%E6%88%96%E8%80%85%E5%B0%86%E5%85%B6%E4%BD%9C%E4%B8%BA%E4%B8%80%E4%B8%AA%E5%87%BD%E6%95%B0%E5%8E%BB%E8%B0%83%E7%94%A8 Reflect %E5%AF%B9%E8%B1%A1%E3%80%82)

|  |
| --- |
| 反射机制是**在编译阶段不知道是哪个类被加载，而是在运行的时候才加载、执行**。 也就是说，反射机制指的是程序在运行时能够获取自身的信息。  js 中的 apply 就是反射机制。  Reflect 是一个内建的对象，用来提供方法去拦截 JavaScript 的操作。  Reflect 不是一个函数对象，所以它是不可构造的，也就是说它不是一个构造器，不能通过 new 操作符去新建或者将其作为一个函数去调用 Reflect 对象。  Reflect 的所有属性和方法都是静态的。  Reflect 内部封装了一系列对对象的底层操作  Reflect 成员方法就是 Proxy 处理对象的默认实现。  const proxy = new Proxy(obj, {  get(target, property) {  // 如果没有定义 get 方法，那么默认返回的就是 Reflect 的 get 方法  return Reflect.get(target, property)  }  }) |

* 内存泄漏

[(4条消息) JavaScript内存泄露的4种方式及如何避免\_ruanhongbiao的博客-CSDN博客\_前端内存泄漏](https://blog.csdn.net/qappleh/article/details/80337630)

|  |  |
| --- | --- |
| 内存泄漏（Memory leak）是在计算机科学中，由于疏忽或错误造成程序未能释放已经不再使用的内存  意外的全局变量  function foo(arg) {  bar = "this is a hidden global variable";  }  另一种意外的全局变量可能由 this 创建：  function foo() {  this.variable = "potential accidental global";  }  // foo 调用自己，this 指向了全局对象（window）  foo();  上述使用严格模式，可以避免意外的全局变量 | 定时器也常会造成内存泄露  var someResource = getData();  setInterval(function() {  var node = document.getElementById('Node');  if(node) {  // 处理 node 和 someResource  node.innerHTML = JSON.stringify(someResource));  }}, 1000);  如果id为Node的元素从DOM中移除，该定时器仍会存在，同时，因为回调函数中包含对someResource的引用，定时器外面的someResource也不会被释放 |
| 包括我们之前所说的闭包，维持函数内局部变量，使其得不到释放  function bindEvent() {  var obj = document.createElement('XXX');  var unused = function () {  console.log(obj, '闭包内引用obj obj不会被释放');  };  obj = null; // 解决方法} | 没有清理对DOM元素的引用同样造成内存泄露  const refA = document.getElementById('refA');  document.body.removeChild(refA); // dom删除了console.log(refA, 'refA'); // 但是还存在引用能console出，整个div 没有被回收  refA = null;  console.log(refA, 'refA'); // 解除引用  包括使用事件监听addEventListener监听的时候，在不监听的情况下使用removeEventListener取消对事件监听 |

* 垃圾回收

[js垃圾回收机制原理给你聊的明明白白 - 云+社区 - 腾讯云 (tencent.com)](https://cloud.tencent.com/developer/article/1852932)

[面试官：说说 JavaScript 中内存泄漏的几种情况？ · Issue #78 · febobo/web-interview (github.com)](https://github.com/febobo/web-interview/issues/78)

|  |  |
| --- | --- |
| （1）、在javascript中，如果一个对象不再被引用，那么这个对象就会被GC回收； （2）、如果两个对象互相引用，而不再被第3者所引用，那么这两个互相引用的对象也会被回收。 | |
| Javascript 具有自动垃圾回收机制（GC：Garbage Collecation），也就是说，执行环境会负责管理代码执行过程中使用的内存  原理：垃圾收集器会定期（周期性）找出那些不再继续使用的变量，然后释放其内存 | |
| **标记清除**  JavaScript最常用的垃圾收回机制  当变量进入执行环境是，就标记这个变量为“进入环境“。进入环境的变量所占用的内存就不能释放，当变量离开环境时，则将其标记为“离开环境“  垃圾回收程序运行的时候，会标记内存中存储的所有变量。然后，它会将所有在上下文中的变量，以及被在上下文中的变量引用的变量的标记去掉  在此之后被标记的变量就是待删除的了，原因是任何在上下文中的变量都访问不到它们了  随后垃圾回收程序做一次内存清理，销毁带标记的所有值并收回它们的内存  举个例子：  var m = 0,n = 19 // 把 m,n,add() 标记为进入环境。  add(m, n) // 把 a, b, c标记为进入环境。  console.log(n) // a,b,c标记为离开环境，等待垃圾回收。  function add(a, b) {  a++  var c = a + b  return c  } | **引用计数**  语言引擎有一张"引用表"，保存了内存里面所有的资源（通常是各种值）的引用次数。如果一个值的引用次数是0，就表示这个值不再用到了，因此可以将这块内存释放  如果一个值不再需要了，引用数却不为0，垃圾回收机制无法释放这块内存，从而导致内存泄漏  const arr = [1, 2, 3, 4];  console.log('hello world');  上面代码中，数组[1, 2, 3, 4]是一个值，会占用内存。变量arr是仅有的对这个值的引用，因此引用次数为1。尽管后面的代码没有用到arr，它还是会持续占用内存  如果需要这块内存被垃圾回收机制释放，只需要设置如下：  arr = null  通过设置arr为null，就解除了对数组[1,2,3,4]的引用，引用次数变为 0，就被垃圾回收了 |

* ajax

[AJAX 简介 (w3school.com.cn)](https://www.w3school.com.cn/js/js_ajax_intro.asp)

|  |
| --- |
| * **不刷新页面更新网页** * **在页面加载后从服务器请求数据** * **在页面加载后从服务器接收数据** * **在后台向服务器发送数据** |

* jsonp

[9种常见的前端跨域解决方案（详解） - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/81809258)

|  |
| --- |
| <script>       var script = document.createElement('script');       script.type = 'text/javascript';       script.scr = 'http://www.baidu.com/login?a=param&callback=callbackFunction';       function callbackFunction(res){           JSON.stringify(res)       }  </script> |

* js常用输入输出语句

[(2条消息) JavaScript常用输入输出语句\_Sky\_武的博客-CSDN博客\_javascript常用输入语句](https://blog.csdn.net/Sky____________/article/details/121069346)

|  |
| --- |
| **JavaScript V8 输入：**  **var line = readline();**  **var num = Number(line);**  **var arr = [];**  **for (let i = 0; i < num; i++) {**  **var kline = readline();**  **arr.push(Number(kline));**  **}**  **// 1 22 3 4**  **var line = readline();**  **var arr = line.split(' ');**  **Number(arr[0]);**  **var line = readline();**  **var num = Number(line);**  **var arr = [];**  **for(let i = 0; i< num; i++){**  **var \_line = readline();**  **arr.push(\_line.split(' '));**  **}** |
| **JavaScript node输入**  var readline = require('readline');  **const** rl = readline.createInterface({      input: process.stdin,      output: process.stdout,      terminal: **false**  });    var n,      m = 0;  var sc = [];    var cur\_line = 0;  rl.on('line', function(line) {      var tokens = line.split(' ');      cur\_line += 1;  **if** (cur\_line >= m + 2) {              n = 0;              m = 0;              sc.splice(0, sc.length);              cur\_line = 0;          }      }  }); |

* require和import的区别

[吊打面试官前端系列（一）： require和import的区别 - 腾讯云开发者社区-腾讯云 (tencent.com)](https://cloud.tencent.com/developer/article/1611197" \l ":~:text=require%E5%92%8Cimport%E7%9A%84%E5%8C%BA%E5%88%AB,1.import%E5%9C%A8%E4%BB%A3%E7%A0%81%E7%BC%96%E8%AF%91%E6%97%B6%E8%A2%AB%E5%8A%A0%E8%BD%BD%EF%BC%8C%E6%89%80%E4%BB%A5%E5%BF%85%E9%A1%BB%E6%94%BE%E5%9C%A8%E6%96%87%E4%BB%B6%E5%BC%80%E5%A4%B4%EF%BC%8Crequire%E5%9C%A8%E4%BB%A3%E7%A0%81%E8%BF%90%E8%A1%8C%E6%97%B6%E8%A2%AB%E5%8A%A0%E8%BD%BD%EF%BC%8C%E6%89%80%E4%BB%A5require%E7%90%86%E8%AE%BA%E4%B8%8A%E5%8F%AF%E4%BB%A5%E8%BF%90%E7%94%A8%E5%9C%A8%E4%BB%A3%E7%A0%81%E7%9A%84%E4%BB%BB%E4%BD%95%E5%9C%B0%E6%96%B9%EF%BC%8C%E6%89%80%E4%BB%A5import%E6%80%A7%E8%83%BD%E6%9B%B4%E5%A5%BD%E3%80%82)

[require和import的区别 - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/121770261)

|  |
| --- |
| **require和import的区别**   1. import在代码编译时被加载，所以必须放在文件开头，require在代码运行时被加载，所以require理论上可以运用在代码的任何地方，所以import性能更好。 2. import引入的对象被修改时，源对象也会被修改，相当于浅拷贝，require引入的对象被修改时，源对象不会被修改，官网称值拷贝，我们可以理解为深拷贝。   3.import有利于tree-shaking（移除JavaScript上下文中未引用的代码），require对tree-shaking不友好。  4.import会触发代码分割（把代码分离到不同的bundle中，然后可以按需加载或者并行加载这些文件），require不会触发。  5.import是es6的一个语法标准，如果要兼容浏览器的话必须转化成es5的语法，require 是 AMD规范引入方式。  目前所有的引擎都还没有实现import，import最终都会被转码为require，在webpack打包中，import和require都会变为\_webpack\_require\_。 |
| **require/exports 是运行时动态加载，import/export 是静态编译**  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。- 阮一峰 |

# JS this关键字

* this

[JavaScript this 关键字 | 菜鸟教程 (runoob.com)](https://www.runoob.com/js/js-this.html" \l ":~:text=JavaScript this %E5%85%B3%E9%94%AE%E5%AD%97 %E9%9D%A2%E5%90%91%E5%AF%B9%E8%B1%A1%E8%AF%AD%E8%A8%80%E4%B8%AD this %E8%A1%A8%E7%A4%BA%E5%BD%93%E5%89%8D%E5%AF%B9%E8%B1%A1%E7%9A%84%E4%B8%80%E4%B8%AA%E5%BC%95%E7%94%A8%E3%80%82 %E4%BD%86%E5%9C%A8 JavaScript,this %E4%B8%8D%E6%98%AF%E5%9B%BA%E5%AE%9A%E4%B8%8D%E5%8F%98%E7%9A%84%EF%BC%8C%E5%AE%83%E4%BC%9A%E9%9A%8F%E7%9D%80%E6%89%A7%E8%A1%8C%E7%8E%AF%E5%A2%83%E7%9A%84%E6%94%B9%E5%8F%98%E8%80%8C%E6%94%B9%E5%8F%98%E3%80%82 %E5%9C%A8%E6%96%B9%E6%B3%95%E4%B8%AD%EF%BC%8Cthis %E8%A1%A8%E7%A4%BA%E8%AF%A5%E6%96%B9%E6%B3%95%E6%89%80%E5%B1%9E%E7%9A%84%E5%AF%B9%E8%B1%A1%E3%80%82 %E5%A6%82%E6%9E%9C%E5%8D%95%E7%8B%AC%E4%BD%BF%E7%94%A8%EF%BC%8Cthis %E8%A1%A8%E7%A4%BA%E5%85%A8%E5%B1%80%E5%AF%B9%E8%B1%A1%E3%80%82 %E5%9C%A8%E5%87%BD%E6%95%B0%E4%B8%AD%EF%BC%8Cthis %E8%A1%A8%E7%A4%BA%E5%85%A8%E5%B1%80%E5%AF%B9%E8%B1%A1%E3%80%82)

|  |
| --- |
| 面向对象语言中 this 表示当前对象的一个引用。  但在 JavaScript 中 this 不是固定不变的，它会随着执行环境的改变而改变。  在***方法***中，this 表示该方法所属的对象。  如果单独使用，this 表示全局对象。  在***函数***中，this 表示全局对象。（意外的全局变量，内存泄漏）  在函数中，在严格模式下，this 是未定义的(undefined)。  在事件中，this 表示接收事件的元素。  类似 call() 和 apply() 方法可以将 this 引用到任何对象 |

* this 指向

[【前端工程师面试宝典】学习说明\_互联网校招面试真题面经汇总\_牛客网 (nowcoder.com)](https://www.nowcoder.com/tutorial/96/1678a0fd35cd4db486af18589e34e4d4)

|  |
| --- |
| 普通函数中的this：谁调用了函数或者方法，那么这个函数或者对象中的this就指向谁。  匿名函数中的this：匿名函数的执行具有全局性，则匿名函数中的this指向是window，而不是调用该匿名函数的对象。  return function () {  console.log(this);  }  箭头函数中的this：  箭头函数中的this是在函数定义的时候就确定下来的，而不是在函数调用的时候确定的；  箭头函数中的this指向父级作用域的执行上下文；（技巧：**因为javascript中除了全局作用域，其他作用域都是由函数创建出来的，所以如果想确定this的指向，则找到离箭头函数最近的function，与该function平级的执行上下文中的this即是箭头函数中的this**）  箭头函数无法使用apply、call和bind方法改变this指向，因为其this值在函数定义的时候就被确定下来。  **箭头函数表达式**的语法比函数表达式更简洁，并且没有自己的this，arguments，super或new.target。箭头函数表达式更适用  那些本来需要匿名函数的地方，并且它不能用作构造函数。 |

* js严格模式下this指向

[js严格模式下this指向\_wy\_Blog的博客-CSDN博客\_js严格模式下的this](https://blog.csdn.net/wy_Blog/article/details/79270607)

|  |
| --- |
| **可以看链接博客中的例子打印示例，更容易理解。**  在严格模式下，在全局作用域中，this指向window对象；  在严格模式下，全局函数中的this等于undefined；  在严格模式下，对象的函数中的this指向调用函数的对象实例；  在严格模式下，构造函数中的this指向构造函数创建的对象实例；  在严格模式下，在事件处理函数中，this指向触发事件的目标对象； |

* 箭头函数

[【前端工程师面试宝典】学习说明\_互联网校招面试真题面经汇总\_牛客网 (nowcoder.com)](https://www.nowcoder.com/tutorial/96/1678a0fd35cd4db486af18589e34e4d4)

|  |
| --- |
| **箭头函数表达式**的语法比函数表达式更简洁，并且没有自己的this，arguments，super或new.target。箭头函数表达式更适用于那些本来需要匿名函数的地方，并且它不能用作构造函数。 |
| 距离箭头函数最近的是getThis(){}，与该函数平级的执行上下文是obj中的执行上下文，箭头函数中的this就是下注释代码处的this，即obj。  let obj = {  //此处的this即是箭头函数中的this  getThis: function () {  return ()=> {  console.log(this);  }  }  }  obj.getThis()(); //obj |

* call，apply，bind

[【前端工程师面试宝典】学习说明\_互联网校招面试真题面经汇总\_牛客网 (nowcoder.com)](https://www.nowcoder.com/tutorial/96/1678a0fd35cd4db486af18589e34e4d4)

[js 手动实现bind方法，超详细思路分析！ - 听风是风 - 博客园 (cnblogs.com)](https://www.cnblogs.com/echolun/p/12178655.html)

|  |
| --- |
| 1. call 和 apply 会调用函数，并且改变函数内部this指向。 2. call 和 apply 传递的参数不一样，call 传递参数arg1,arg2...形式 apply 必须数组形式[arg] 3. bind 不会调用函数，可以改变函数内部this指向。   fun.call(thisArg, arg1, arg3, ...)  function Person(uname, age) {  this.uname = uname;  this.age = age;  }  function Son(uname, age) {  Person.call(this, uname, age);  }  var son = new Son("zhang", 12);  console.log(son);  fun.apply(thisArg, [argsArray])  bind的实现过程：涉及new 对象的过程、闭包、柯里化、slice、instanceof……  Function.prototype.myBind = function (context) {  // 判断调用对象是否为函数  if (typeof this !== "function") {  throw new TypeError("Error");  }  // 获取参数  const args = [...arguments].slice(1),  fn = this;  return function Fn() {  // 根据调用方式，传入不同绑定值  return fn.apply(this instanceof fn ? new fn(...arguments) : context, args.concat(...arguments));  }} |

# JS 继承

* 构造函数定义和使用

[js构造函数的定义 - 简书 (jianshu.com)](https://www.jianshu.com/p/7e21e23ffba9)

|  |
| --- |
| 只有当一个函数以 new 关键字来调用的时候，我们才能说它是一个构造函数。 |

* 原型链、proto 和 prototype

[详谈JavaScript原型链 - 程序猿的程 - 博客园 (cnblogs.com)](https://www.cnblogs.com/chengzp/p/prototype.html)

|  |
| --- |
| 简单理解就是原型组成的链，对象的\_\_proto\_\_它的是原型，而原型也是一个对象，也有\_\_proto\_\_属性，原型的\_\_proto\_\_又是原型的原型，就这样可以一直通过\_\_proto\_\_想上找，这就是原型链，当向上找找到Object的原型的时候，这条原型链就算到头了。  通过一个构造函数创建出来的多个实例，如果都要添加一个方法，给每个实例去添加并不是一个明智的选择。这时就该用上原型了。  在实例的原型上添加一个方法，这个原型的所有实例便都有了这个方法。  IMG_256   1. 获取\_\_proto\_\_可能的兼容问题   Object.getPrototypeOf()  Object.getPrototypeOf() 方法返回指定对象的原型（内部[[Prototype]]属性的值）。  用法：Object.getPrototypeOf(object)  [Object.prototype.\_\_proto\_\_ - JavaScript | MDN (mozilla.org)](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Object/proto) |

* 原型链继承

[【前端工程师面试宝典】学习说明\_互联网校招面试真题面经汇总\_牛客网 (nowcoder.com)](https://www.nowcoder.com/tutorial/96/1678a0fd35cd4db486af18589e34e4d4)

[JavaScript 继承的八种方式 - 云+社区 - 腾讯云 (tencent.com)](https://cloud.tencent.com/developer/article/1851143)

|  |
| --- |
| 原型链继承  实现方式：将子类的原型链指向父类的对象实例  function Parent(){  this.name = "parent";  this.list = ['a'];  }  Parent.prototype.sayHi = function(){  console.log('hi');  }  function Child(){  }  Child.prototype = new Parent();  var child = new Child();  console.log(child.name);  child.sayHi();  原理：子类实例child的\_\_proto\_\_指向Child的原型链prototype，而Child.prototype指向Parent类的对象实例，该父类对象实例的\_\_proto\_\_指向Parent.prototype,所以Child可继承Parent的构造函数属性、方法和原型链属性、方法 优点：可继承构造函数的属性，父类构造函数的属性，父类原型的属性 缺点：无法向父类构造函数传参；且所有实例共享父类实例的属性，若父类共有属性为引用类型，一个子类实例更改父类构造函数共有属性时会导致继承的共有属性发生变化；实例如下：  var a = new Child();  var b = new Child();  a.list.push('b');  console.log(b.list); // ['a','b'] |

* 构造函数继承

|  |
| --- |
| 构造函数继承  实现方式：在子类构造函数中使用call或者apply劫持父类构造函数方法，并传入参数  function Parent(name, id){  this.id = id;  this.name = name;  this.printName = function(){  console.log(this.name);  }  }  Parent.prototype.sayName = function(){  console.log(this.name);  };  function Child(name, id){  Parent.call(this, name, id);  // Parent.apply(this, arguments);  }  var child = new Child("jin", "1");  child.printName(); // jin  child.sayName() // Error  原理：使用call或者apply更改子类函数的作用域，使this执行父类构造函数，子类因此可以继承父类共有属性 优点：可解决原型链继承的缺点 缺点：不可继承父类的原型链方法，构造函数不可复用 |

* 组合继承

|  |
| --- |
| 组合继承  原理：综合使用构造函数继承和原型链继承  function Parent(name, id){  this.id = id;  this.name = name;  this.list = ['a'];  this.printName = function(){  console.log(this.name);  }  }  Parent.prototype.sayName = function(){  console.log(this.name);  };  function Child(name, id){  Parent.call(this, name, id);  // Parent.apply(this, arguments);  }  Child.prototype = new Parent();  var child = new Child("jin", "1");  child.printName(); // jin  child.sayName() // jin  var a = new Child();  var b = new Child();  a.list.push('b');  console.log(b.list); // ['a']  优点：可继承父类原型上的属性，且可传参；每个新实例引入的构造函数是私有的 缺点：会执行两次父类的构造函数，消耗较大内存，子类的构造函数会代替原型上的那个父类构造函数 |

* 原型式继承

|  |
| --- |
| 原型式继承  原理：类似Object.create，用一个函数包装一个对象，然后返回这个函数的调用，这个函数就变成了个可以随意增添属性的实例或对象，结果是将子对象（子类的实例）的\_\_proto\_\_指向父对象  var parent = {  names: ['a']  }  function copy(object) {  function F() {}  F.prototype = object;  return new F();  }  var child = copy(parent);  缺点：共享引用类型 |

* 寄生式继承

|  |
| --- |
| 寄生式继承  原理：二次封装原型式继承，并拓展  function createObject(obj) {  var o = copy(obj);  o.getNames = function() {  console.log(this.names);  return this.names;  }  return o;  }  优点：可添加新的属性和方法 |

* 寄生组合式继承

|  |
| --- |
| 寄生组合式继承  原理：改进组合继承，利用寄生式继承的思想继承原型  function inheritPrototype(subClass, superClass) {  // 复制一份父类的原型  var p = copy(superClass.prototype);  // 修正构造函数  p.constructor = subClass;  // 设置子类原型  subClass.prototype = p;  }  function Parent(name, id){  this.id = id;  this.name = name;  this.list = ['a'];  this.printName = function(){  console.log(this.name);  }  }  Parent.prototype.sayName = function(){  console.log(this.name);  };  function Child(name, id){  Parent.call(this, name, id);  // Parent.apply(this, arguments);  }  inheritPrototype(Child, Parent); |

* ES6 class继承

|  |
| --- |
| **class 继承**  ES6 中，通过 class 关键字来定义类，子类可以通过 extends 继承父类。  代码示例：  class Parent{  constructor(name) {  this.name = name;  this.hobbies = ["sing", "dance", "rap"];  }  getHobbies() {  return this.hobbies;  }  static getCurrent() {  console.log(this);  }}  class Child extends Parent {  constructor(name) {  super(name);  }}  var c1 = new Child('c1');var c2 = new Child('c2');  console.log(c1 instanceof Child); // true  console.log(c1 instanceof Parent); // true  要点：   * constructor 为构造函数，即使未定义也会自动创建。 * 在父类构造函数内 this 定义的都是实例属性和方法，其他方法包括 constructor、getHobbies 都是原型方法。 * static 关键字定义的静态方法都必须通过类名调用，其 this 指向调用者而并非实例。 * 通过 extends 可以继承父类的所有原型属性及 static 类方法，子类 constructor 调用 super 父类构造函数实现实例属性和方法的继承。   对比：   * ES5 的继承，实质是先创造子类的实例对象 this，然后再将父类的方法添加到 this 上面（Parent.apply(this)）。   ES6 的继承机制完全不同，实质是先将父类实例对象的属性和方法，加到 this 上面（所以必须先调用 super 方法），然后再用子类的构造函数修改 this。 |

# JS 事件循环

* 事件循环 、macrotask, microtask

[Js 的事件循环(Event Loop)机制以及实例讲解 - 云+社区 - 腾讯云 (tencent.com)](https://cloud.tencent.com/developer/article/1731010)

[(4条消息) promise.then,process.nextTick, setTimeout 以及 setImmediate 的执行顺序\_roamingcode的博客-CSDN博客](https://blog.csdn.net/roamingcode/article/details/81737275)

[(3条消息) 面试率 90% 的JS事件循环Event Loop，看这篇就够了！! !\_王同学要努力的博客-CSDN博客](https://blog.csdn.net/m0_46374969/article/details/119969908)

|  |
| --- |
| macro-task: script (整体代码)，setTimeout, setInterval, setImmediate, I/O, UI rendering. micro-task: process.nextTick, [Promise](https://so.csdn.net/so/search?q=Promise&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/roamingcode/article/details/_blank)(原生)，Object.observe，MutationObserver  并执行script console.log(3); resolve(); console.log(4); 此时输出3和4，虽然resolve调用了，执行了但是整体代码还没执行完，无法进入Promise.then 流程。  ****执行栈与任务队列****  因为js是单线程语言，当遇到异步任务(如ajax操作等)时，不可能一直等待异步完成，再继续往下执行，在这期间浏览器是空闲状态，显而易见这会导致巨大的资源浪费。  ****执行栈****  当执行某个函数、用户点击一次鼠标，Ajax完成，一个图片加载完成等事件发生时，只要指定过回调函数，这些事件发生时就会进入执行栈队列中，等待主线程读取,遵循先进先出原则。  ****主线程****  要明确的一点是，主线程跟执行栈是不同概念，主线程规定现在执行执行栈中的哪个事件。  主线程循环：即主线程会不停的从执行栈中读取事件，会执行完所有栈中的同步代码。  当遇到一个异步事件后，并不会一直等待异步事件返回结果，而是会将这个事件挂在与执行栈不同的队列中，我们称之为任务队列(Task Queue)。  ****当主线程将执行栈中所有的代码执行完之后，主线程将会去查看任务队列是否有任务****。如果有，那么主线程会依次执行那些任务队列中的回调函数。  ****Event Loop(事件循环)：****  Event Loop(事件循环)中，每一次循环称为 tick, 每一次tick的任务如下：   * 选择最先进入队列的宏任务(通常是script整体代码)，如果有则执行 * 检查是否存在 Microtask，如果存在则不停的执行，直至清空 microtask 队列 * 更新render(每一次事件循环，浏览器都可能会去更新渲染) * 重复以上步骤 |

* 事件循环——执行栈、调用栈

[js执行栈与事件循环简单理解 - 腾讯云开发者社区-腾讯云 (tencent.com)](https://cloud.tencent.com/developer/article/1863830)

[事件循环机制-执行栈、调用栈 - 渣渣逆天 - 博客园 (cnblogs.com)](https://www.cnblogs.com/zhazhanitian/p/13064274.html)

|  |
| --- |
| **浏览器线程**  JS解析线程 (javaScript属于单线程,每次只能去处理一件事)  GUI渲染线程 (可以理解成解析加载css tree和 dom tree,生成render tree生成页面,包括重绘都是会触发GUI线程,与此同时 ，GUI线程和JS主线程是互斥的,即不能同时存在)  网络请求线程 (它可以异步处理http请求,请求回来的数据仍在事件队列线程中,等JS线程空下来之后, 才会推到JS线程中执行, 属于微任务)  定时器线程 ( 指的是setTimeout,setInterval,JS线程没办法读秒,所以读秒的任务就是定时器线程在做, 定时器属于宏任务)  事件队列线程 ( 这个线程指的是异步回调结束之后, 暂时放在这个线程中,等待JS线程空下来后再次执行 ) |
| **执行栈**  栈，是一种数据结构，具有先进后出的原则。JS 中的执行栈就具有这样的结构，当引擎第一次遇到 JS 代码时，会产生一个全局执行上下文并压入执行栈，每遇到一个函数调用，就会往栈中压入一个新的上下文。引擎执行栈顶的函数，执行完毕，弹出当前执行上下文  function foo() {    console.log('1')    bar();    console.log('3')  }    function bar() {    console.log('2')  }    foo()  以引例来说明。  当 foo() 函数被调用，将 foo 函数的执行上下文压入执行栈，接着执行输出 ‘1’；  当 bar() 函数被调用，将 bar 函数的执行上下文压入执行栈，接着执行输出 ‘2’；  bar() 执行完毕，被弹出执行栈，foo() 函数接着执行，输出 ‘3’；  foo() 函数执行完毕，被弹出执行栈 |
| **调用栈（执行堆栈）**  调用堆栈顾名思义是一个具有LIFO(后进先出)结构的堆栈，用于存储在代码执行期间创建的所有执行上下文  JS 只有一个调用栈，因为它是一种单线程编程语言，调用堆栈具有 LIFO 结构，这意味着项目只能从堆栈顶部添加或删除  可以看出其实执行栈和调用堆栈是可以相互嵌套的 |

* js实现异步操作原理

[事件循环机制-执行栈、调用栈 - 渣渣逆天 - 博客园 (cnblogs.com)](https://www.cnblogs.com/zhazhanitian/p/13064274.html)

|  |
| --- |
| **JS是单线程的，那是如何实现异步操作**  JS的异步是通过回调函数实现的，即通过任务队列，在主线程执行完当前的任务栈（所有的同步操作），主线程空闲后轮询任务队列，并将任务队列中的任务（回调函数）取出来执行。"回调函数"（callback），就是那些会被主线程挂起来的代码。异步任务必须指定回调函数，当主线程开始执行异步任务，就是执行对应的回调函数。  虽然JS是单线程的但是浏览器的内核是多线程的，在浏览器的内核中不同的异步操作由不同的浏览器内核模块调度执行，异步操作会将相关回调添加到任务队列中。而不同的异步操作添加到任务队列的时机也不同，如 onclick, setTimeout, ajax 处理的方式都不同，这些异步操作是由浏览器内核的 webcore 来执行的，webcore 包含上图中的3种 webAPI，分别是 DOM Binding、network、timer模块   1. onclick 由浏览器内核的 DOM Binding 模块来处理，当事件触发的时候，回调函数会立即添加到任务队列中 2. setTimeout 会由浏览器内核的 timer 模块来进行延时处理，当时间到达的时候，才会将回调函数添加到任务队列中 3. ajax 则会由浏览器内核的 network 模块来处理，在网络请求完成返回之后，才将回调添加到任务队列中   总结得出异步运行机制如下：   1. 所有同步任务都在主线程上执行，形成一个执行栈（execution context stack） 2. 主线程之外，还存在一个"任务队列"（task queue），只要异步任务有了运行结果，就在"任务队列"之中放置一个事件 3. 一旦"执行栈"中的所有同步任务执行完毕，系统就会读取"任务队列"，看看里面有哪些事件。那些对应的异步任务，于是结束等待状态，进入执行栈，开始执行 4. 主线程不断重复上面的第三步   只要主线程空了，就会去读取"任务队列"，这就是JavaScript的运行机制。这个过程会不断重复。(该过程又称之为事件轮询) |

* 什么是异步编程

[Javascript异步编程的4种方法 - 阮一峰的网络日志 (ruanyifeng.com)](https://www.ruanyifeng.com/blog/2012/12/asynchronous%EF%BC%BFjavascript.html)

|  |
| --- |
| ****一、回调函数****  这是异步编程最基本的方法。  假定有两个函数f1和f2，后者等待前者的执行结果。  　　f1();  　　f2();  如果f1是一个很耗时的任务，可以考虑改写f1，把f2写成f1的回调函数。  　　function f1(callback){  　　　　setTimeout(function () {  　　　　　　// f1的任务代码  　　　　　　callback();  　　　　}, 1000);  　　}  执行代码就变成下面这样：  　　f1(f2);  采用这种方式，我们把同步操作变成了异步操作，f1不会堵塞程序运行，相当于先执行程序的主要逻辑，将耗时的操作推迟执行。  回调函数的优点是简单、容易理解和部署，缺点是不利于代码的阅读和维护，各个部分之间高度[耦合](https://en.wikipedia.org/wiki/Coupling_(computer_programming)" \t "https://www.ruanyifeng.com/blog/2012/12/_blank)（Coupling），流程会很混乱，而且每个任务只能指定一个回调函数。 |
| ****二、事件监听****  另一种思路是采用事件驱动模式。任务的执行不取决于代码的顺序，而取决于某个事件是否发生。  还是以f1和f2为例。首先，为f1绑定一个事件（这里采用的jQuery的[写法](https://api.jquery.com/on/" \t "https://www.ruanyifeng.com/blog/2012/12/_blank)）。  　　f1.on('done', f2);  上面这行代码的意思是，当f1发生done事件，就执行f2。然后，对f1进行改写：  　　function f1(){  　　　　setTimeout(function () {  　　　　　　// f1的任务代码  ****f1.trigger('done');****  　　　　}, 1000);  　　}  f1.trigger('done')表示，执行完成后，立即触发done事件，从而开始执行f2。  这种方法的优点是比较容易理解，可以绑定多个事件，每个事件可以指定多个回调函数，而且可以["去耦合"](https://en.wikipedia.org/wiki/Decoupling" \t "https://www.ruanyifeng.com/blog/2012/12/_blank)（Decoupling），有利于实现[模块化](https://www.ruanyifeng.com/blog/2012/10/javascript_module.html" \t "https://www.ruanyifeng.com/blog/2012/12/_blank)。缺点是整个程序都要变成事件驱动型，运行流程会变得很不清晰。 |
| ****三、发布/订阅****  上一节的"事件"，完全可以理解成"信号"。  我们假定，存在一个"信号中心"，某个任务执行完成，就向信号中心"发布"（publish）一个信号，其他任务可以向信号中心"订阅"（subscribe）这个信号，从而知道什么时候自己可以开始执行。这就叫做["发布/订阅模式"](https://en.wikipedia.org/wiki/Publish-subscribe_pattern" \t "https://www.ruanyifeng.com/blog/2012/12/_blank)（publish-subscribe pattern），又称["观察者模式"](https://en.wikipedia.org/wiki/Observer_pattern" \t "https://www.ruanyifeng.com/blog/2012/12/_blank)（observer pattern）。  这个模式有多种[实现](https://msdn.microsoft.com/en-us/magazine/hh201955.aspx" \t "https://www.ruanyifeng.com/blog/2012/12/_blank)，下面采用的是Ben Alman的[Tiny Pub/Sub](https://gist.github.com/661855" \t "https://www.ruanyifeng.com/blog/2012/12/_blank)，这是jQuery的一个插件。  首先，f2向"信号中心"jQuery订阅"done"信号。  　　jQuery.subscribe("done", f2);  然后，f1进行如下改写：  　　function f1(){  　　　　setTimeout(function () {  　　　　　　// f1的任务代码  ****jQuery.publish("done");****  　　　　}, 1000);  　　}  jQuery.publish("done")的意思是，f1执行完成后，向"信号中心"jQuery发布"done"信号，从而引发f2的执行。  此外，f2完成执行后，也可以取消订阅（unsubscribe）。  　　jQuery.unsubscribe("done", f2);  这种方法的性质与"事件监听"类似，但是明显优于后者。因为我们可以通过查看"消息中心"，了解存在多少信号、每个信号有多少订阅者，从而监控程序的运行。 |
| ****四、Promises对象****  Promises对象是CommonJS工作组提出的一种规范，目的是为异步编程提供[统一接口](http://wiki.commonjs.org/wiki/Promises/A" \t "https://www.ruanyifeng.com/blog/2012/12/_blank)。  简单说，它的思想是，每一个异步任务返回一个Promise对象，该对象有一个then方法，允许指定回调函数。比如，f1的回调函数f2,可以写成：  　　f1().then(f2); |

* promise

[面试官：你是怎么理解ES6中 Promise的？使用场景？ · Issue #40 · febobo/web-interview (github.com)](https://github.com/febobo/web-interview/issues/40)

|  |  |
| --- | --- |
| promise对象仅有三种状态   * pending（进行中） * fulfilled（已成功） * rejected（已失败） | Promise构建出来的实例存在以下方法   * then() * catch()   catch()方法是.then(null, rejection)或.then(undefined, rejection)的别名，用于指定发生错误时的回调函数catch()方法之中，还能再抛出错误，通过后面catch方法捕获到   * finally() |
| Promise构造函数存在以下方法：   * all() * race() * allSettled() * resolve() * reject() * try() | // ***Promise 新建后就会立即执行***。  let promise = new Promise(function(resolve, reject) {  console.log('Promise');  resolve();  });  promise.then(function() {  console.log('resolved.');  });  console.log('Hello!');  // Promise  // Hello!  // resolved |

* 回调函数，promise，await，generator 区别

[Promise、Generator、Async有什么区别？ - 掘金 (juejin.cn)](https://juejin.cn/post/7062155174436929550)

[ES6 Async/Await ,Generator 与Promise区别 - 简书 (jianshu.com)](https://www.jianshu.com/p/2a8cd4170765)

|  |
| --- |
| 回调函数：  ajax('aaa', () => {  // callback 函数体  ajax('bbb', () => {  // callback 函数体  ajax('ccc', () => {  // callback 函数体  })  })  }) |
| promise：  let promise = new Promise(function(resolve, reject) {  console.log('Promise');  resolve();  });  promise.then(function() {  console.log('resolved.');  });  console.log('Hello!');  // Promise  // Hello!  // resolved  promise缺点   * Promise的内部错误使用**try catch**捕获不到，只能只用**then**的第二个回调或**catch**来捕获 * Promise一旦新建就会立即执行，无法取消 |
| Generator：  声明  与函数声明类似，不同的是**function**关键字与函数名之间有一个星号，以及函数体内部使用**yield**表达式，定义不同的内部状态（**yield**在英语里的意思就是“产出”）。  function\* gen(x){  const y = yield x + 6;  return y;  }  // yield 如果用在另外一个表达式中,要放在()里面  // 像上面如果是在=右边就不用加()  function\* genOne(x){  const y = `这是第一个 yield 执行:${yield x + 1}`;  return y;  }  执行  const g = gen(1);  //执行 Generator 会返回一个Object,而不是像普通函数返回return 后面的值  g.next() // { value: 7, done: false }  //调用指针的 next 方法,会从函数的头部或上一次停下来的地方开始执行，直到遇到下一个 yield 表达式或return语句暂停,也就是执行yield 这一行  // 执行完成会返回一个 Object,  // value 就是执行 yield 后面的值,done 表示函数是否执行完毕  g.next() // { value: undefined, done: true }  // 因为最后一行 return y 被执行完成,所以done 为 true |
| Async/await ：  Async/await其实就是上面Generator的语法糖，async函数其实就相当于funciton \*的作用，而await就相当与yield的作用。而在async/await机制中，自动包含了我们上述封装出来的spawn自动执行函数。  async function fetch() {  await ajax('aaa')  await ajax('bbb')  await ajax('ccc')  }  // 但这是在这三个请求有相互依赖的前提下可以这么写，不然会产生性能问题，因为你每一个请求都需要等待上一次请求完成后再发起请求，如果没有相互依赖的情况下，建议让它们同时发起请求，这里可以使用Promise.all()来处理  如果await特别多怎么办  （当时没有回答出来，后来了解了一下，应该从并发上面说（promise.all与await）） |

# JS 函数

* 函数和方法

[javascript中函数和方法的区别\_web前端 zxp的博客-CSDN博客\_js方法和函数的区别](https://blog.csdn.net/qq_44163269/article/details/104627274)

|  |
| --- |
| 方法（method）: 当将****函数****和****对象****合写在一起时，函数就变成了 “方法”（method）// 当函数赋值给对象的属性，我们称为"方法"  函数和方法本质上是一样的，只不过方法是函数的特例，是将****函数****值赋给了****对象****  函数和对象合写在一起  var obj = {  name : '张三',  age : 18  fun : function(){  console.log(this.name) //这里的this指向的是obj这个对象  }//fun就成了该对象的一个方法  }  或者这样  var abc = function(){  ......  } |

* 纯函数

[js 纯函数思想 - 简书 (jianshu.com)](https://www.jianshu.com/p/a9daa1464f9d)

|  |
| --- |
| 若一个函数对相同的输入，永远会得到相同的输出，并且不会影响该函数作用域以外的环境变量，则此函数称为纯函数。 |

* 立即执行函数

[JavaScript中的立即执行函数 - SegmentFault 思否](https://segmentfault.com/a/1190000003902899)

|  |
| --- |
| 立即执行函数通常有下面两种写法：  (function(){  ...  })();  (function(){  ...  }());  在Javascript中，一对圆括号“()”是一种运算符，跟在函数名之后，表示调用该函数。 |
| 很多人刚开始理解立即执行函数的时候，觉得应该是这样的：  function (){ ... }()；  //或者  function fName(){ ... }();  然而事实却是这样：SyntaxError: Unexpected token (。 |

* 柯里化

[面试官：说说你对闭包的理解？闭包使用场景 · Issue #57 · febobo/web-interview (github.com)](https://github.com/febobo/web-interview/issues/57)

|  |
| --- |
| 柯里化的目的在于避免频繁调用具有相同参数函数的同时，又能够轻松的重用  // 假设我们有一个求长方形面积的函数  function getArea(width, height) {  return width \* height  }  // 如果我们碰到的长方形的宽老是10  const area1 = getArea(10, 20)  const area2 = getArea(10, 30)  const area3 = getArea(10, 40)  // 我们可以使用闭包柯里化这个计算面积的函数  function getArea(width) {  return height => {  return width \* height  }  }  const getTenWidthArea = getArea(10) // 之后碰到宽度为10的长方形就可以这样计算面积  const area1 = getTenWidthArea(20) //10：width；20：height  // 而且如果遇到宽度偶尔变化也可以轻松复用  const getTwentyWidthArea = getArea(20) |

* new 对象的过程

[面试官：说说new操作符具体干了什么？ · Issue #69 · febobo/web-interview (github.com)](https://github.com/febobo/web-interview/issues/69)

|  |
| --- |
| function mynew(Func, ...args) {  // 1.创建一个新对象  const obj = {}  // 2.新对象原型指向构造函数原型对象  obj.\_\_proto\_\_ = Func.prototype  // 3.将构建函数的this指向新对象  let result = Func.apply(obj, args)  // 4.根据返回值判断  return result instanceof Object ? result : obj} |
| function newFunc(Func,...args) {  //new obj  let obj = {};  //obj.\_\_proto\_\_ = Func.prototype  ***Object***.setPrototypeOf(obj,Func.prototype)  //console.log(Object.getPrototypeOf(obj) === Func.prototype)  //this  let res = Func.apply(obj,args);  //constructor  return res instanceof Func ? res : obj; } |

* 闭包

[详解js闭包 - SegmentFault 思否](https://segmentfault.com/a/1190000000652891" \l ":~:text=%E9%97%AD%E5%8C%85 %EF%BC%88 closure,%EF%BC%89%E6%98%AF Javascript %E8%AF%AD%E8%A8%80%E7%9A%84%E4%B8%80%E4%B8%AA%E9%9A%BE%E7%82%B9%EF%BC%8C%E4%B9%9F%E6%98%AF%E5%AE%83%E7%9A%84%E7%89%B9%E8%89%B2%EF%BC%8C%E5%BE%88%E5%A4%9A%E9%AB%98%E7%BA%A7%E5%BA%94%E7%94%A8%E9%83%BD%E8%A6%81%E4%BE%9D%E9%9D%A0%E9%97%AD%E5%8C%85%E5%AE%9E%E7%8E%B0%E3%80%82)

|  |  |
| --- | --- |
| 1.函数嵌套函数  2.函数内部可以引用外部的参数和变量  3.参数和变量不会被垃圾回收机制回收 | |
| <script>  function outer(){  var x=10;  return function(){  //函数嵌套函数  x++;  alert(x);  }  }  var y = outer();  //外部函数赋给变量y;  y();  //y函数调用一次，结果为11，相当于outer()()；  y();  //y函数调用第二次，结果为12，实现了累加  </script> | <script>  var aaa = (function(){  var a = 1;  function bbb(){  a++;  alert(a);  }  function ccc(){  a++;  alert(a);  }  return {  b:bbb, //json结构  c:ccc  }  })();  aaa.b(); //2  aaa.c() //3  </script> |

* 防抖节流

[JavaScript---节流与防抖 - 掘金 (juejin.cn)](https://juejin.cn/post/7030787304696315918)

[面试官：什么是防抖和节流？有什么区别？如何实现？ · Issue #83 · febobo/web-interview (github.com)](https://github.com/febobo/web-interview/issues/83)（防抖debounce、节流throttle）

|  |  |
| --- | --- |
| **节流**  使用时间戳写法，事件会立即执行，停止触发后没有办法再次执行  function throttled1(fn, delay = 500) {  let oldtime = Date.now()  return function (...args) {  let newtime = Date.now()  if (newtime - oldtime >= delay) {  fn.apply(null, args)  oldtime = Date.now()  }  }  }  使用定时器写法，delay毫秒后第一次执行，第二次事件停止触发后依然会再一次执行  function throttled2(fn, delay = 500) {  let timer = null  return function (...args) {  if (!timer) {  timer = setTimeout(() => {  fn.apply(this, args)  timer = null  }, delay);  }  }  }  可以将时间戳写法的特性与定时器写法的特性相结合，实现一个更加精确的节流。实现如下  function throttled(fn, delay) {  let timer = null  let starttime = Date.now()  return function () {  let curTime = Date.now() // 当前时间  let remaining = delay - (curTime - starttime) // 从上一次到现在，还剩下多少多余时间  let context = this  let args = arguments  clearTimeout(timer)  if (remaining <= 0) {  fn.apply(context, args)  starttime = Date.now()  } else {  timer = setTimeout(fn, remaining);  }  }  } | **防抖**  简单版本的实现  function debounce(func, wait) {  let timeout;  return function () {  let context = this; // 保存this指向  let args = arguments; // 拿到event对象  clearTimeout(timeout)  timeout = setTimeout(function(){  func.apply(context, args)  }, wait);  }  }  防抖如果需要立即执行，可加入第三个参数用于判断，实现如下：  function debounce(func, wait, immediate) {  let timeout;  return function () {  let context = this;  let args = arguments;  if (timeout) clearTimeout(timeout); // timeout 不为null  if (immediate) {  let callNow = !timeout; // 第一次会立即执行，以后只有事件执行后才会再次触发  timeout = setTimeout(function () {  timeout = null;  }, wait)  if (callNow) {  func.apply(context, args)  }  }  else {  timeout = setTimeout(function () {  func.apply(context, args)  }, wait);  }  }  } |

* 深浅拷贝

[面试官：深拷贝浅拷贝的区别？如何实现一个深拷贝？ · Issue #56 · febobo/web-interview (github.com)](https://github.com/febobo/web-interview/issues/56)

* sort 方法，根据对象字段排序
* 手写 jsonp

[9种常见的前端跨域解决方案（详解） - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/81809258)

|  |
| --- |
| <script>  var script = document.createElement('script');  script.type = 'text/javascript';  // 传参一个回调函数名给后端，方便后端返回时执行这个在前端定义的回调函数  script.src = 'http://www.domain2.com:8080/login?user=admin&callback=handleCallback';  document.head.appendChild(script);  // 回调执行函数  function handleCallback(res) {  alert(JSON.stringify(res));  }  </script> |

* 手写 ajax

[面试题：手写ajax - 掘金 (juejin.cn)](https://juejin.cn/post/6992604888165253156)

# JS 设计模式

* MVVM 和 MVC

[MVC、MVVM模式的概念与区别 - 理想三旬· - 博客园 (cnblogs.com)](https://www.cnblogs.com/wenkangIT/p/15149077.html" \l ":~:text=mvc %E5%92%8C mvvm %E5%85%B6%E5%AE%9E%E5%8C%BA%E5%88%AB%E5%B9%B6%E4%B8%8D%E5%A4%A7%E3%80%82,%E9%83%BD%E6%98%AF%E4%B8%80%E7%A7%8D%E8%AE%BE%E8%AE%A1%E6%80%9D%E6%83%B3%EF%BC%8C%E4%B8%BB%E8%A6%81%E5%8C%BA%E5%88%AB%E5%A6%82%E4%B8%8B%EF%BC%9A 2.mvvm %E9%80%9A%E8%BF%87%E6%95%B0%E6%8D%AE%E6%9D%A5%E9%A9%B1%E5%8A%A8%E8%A7%86%E5%9B%BE%E5%B1%82%E7%9A%84%E6%98%BE%E7%A4%BA%E8%80%8C%E4%B8%8D%E6%98%AF%E8%8A%82%E7%82%B9%E6%93%8D%E4%BD%9C%E3%80%82 3.mvc%E4%B8%ADModel%E5%92%8CView%E6%98%AF%E5%8F%AF%E4%BB%A5%E7%9B%B4%E6%8E%A5%E6%89%93%E4%BA%A4%E9%81%93%E7%9A%84%EF%BC%8C%E9%80%A0%E6%88%90Model%E5%B1%82%E5%92%8CView%E5%B1%82%E4%B9%8B%E9%97%B4%E7%9A%84%E8%80%A6%E5%90%88%E5%BA%A6%E9%AB%98%E3%80%82)

* 发布订阅模式、观察者模式

[面试官：说说你对发布订阅、观察者模式的理解？区别？ · Issue #295 · febobo/web-interview (github.com)](https://github.com/febobo/web-interview/issues/295)

* 设计模式

[面试官：说说对设计模式的理解？常见的设计模式有哪些？ · Issue #290 · febobo/web-interview (github.com)](https://github.com/febobo/web-interview/issues/290)

|  |
| --- |
| 观察者模式：  class subject{      constructor(){          this.observerList = [];      }      addObserver(observer){          this.observerList.push(observer);      }      removeObserver(observer){          const index = this.observerList.findIndex(o => o.name === observer.name);          this.observerList.splice(index,1);      }      notifyObserver(message){          this.observerList.forEach(observer => observer.notified(message));      }  }  class Observer{      constructor(name,subject){          this.name = name;          if(subject){              subject.addObserver(this);          }      }      notified(message){          console.log(this.name,"know the message :",message);      }  }  var s = new subject();  var o = new Observer("name",s);  var oo = new Observer("namee");  s.addObserver(oo);  s.notifyObserver("message"); |

# JS 数据结构/方法

* Map

[JavaScript Map 对象 (w3school.com.cn)](https://www.w3school.com.cn/js/js_object_maps.asp)

* map 和 set

[JS基础：Map和Set对象 - 云+社区 - 腾讯云 (tencent.com)](https://cloud.tencent.com/developer/article/1499648)

* array.sort()方法失效

[JavaScript中sort方法排序不正确\_大前端工程师的博客-CSDN博客\_js sort排序错误](https://blog.csdn.net/chengqige/article/details/119358622?utm_medium=distribute.pc_relevant.none-task-blog-2~default~baidujs_baidulandingword~default-1-119358622-blog-112616713.pc_relevant_multi_platform_whitelistv3&spm=1001.2101.3001.4242.2&utm_relevant_index=3)

|  |
| --- |
| 为什么我们省略回调函数这个参数，就不行了呢？  原因很简单，当你没有传递这个参数的时候，会触发sort的默认行为，它的默认行为就是把数组中所有的项转化为字符串，然后进行unicode编码比较。 |
| 如果你想升序排列  var arr=[12,45,7,5,23,5,54]  arr.sort(function (a,b){  return a-b  })  如果你想降序排列  var arr=[12,45,7,5,23,5,54]  arr.sort(function (a,b){  return b-a  }) |

* for in 和 for of

[Js中for in 和for of的区别 - 掘金 (juejin.cn)](https://juejin.cn/post/6844903601261772808)

|  |
| --- |
| for in遍历的是数组的索引（即键名），而for of遍历的是数组元素值。 所以for in更适合遍历对象，不要使用for in遍历数组。  for in遍历对象，for of 遍历数组值 |

* push、shift

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | [shift()](https://www.w3school.com.cn/jsref/jsref_shift.asp" \o "JavaScript Array shift() 方法) | 删除数组的第一个元素，并返回该元素。 | | [unshift()](https://www.w3school.com.cn/jsref/jsref_unshift.asp" \o "JavaScript Array unshift() 方法) | 将新元素添加到数组的开头，并返回新的长度。 | | [push()](https://www.w3school.com.cn/jsref/jsref_push.asp" \o "JavaScript Array push() 方法) | 将新元素添加到数组的末尾，并返回新的长度。 | | [pop()](https://www.w3school.com.cn/jsref/jsref_pop.asp" \o "JavaScript Array pop() 方法) | 删除数组的最后一个元素，并返回该元素。 | |

* forEach 和 map

[JavaScript中Map和ForEach的区别 - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/86442642" \l ":~:text=%E4%B9%9F%E5%B0%B1%E6%98%AF%E8%AF%B4%EF%BC%8C forEach () %E4%BC%9A%E4%BF%AE%E6%94%B9%E5%8E%9F%E6%9D%A5%E7%9A%84%E6%95%B0%E7%BB%84%E3%80%82 %E8%80%8C map,() %E6%96%B9%E6%B3%95%E4%BC%9A%E5%BE%97%E5%88%B0%E4%B8%80%E4%B8%AA%E6%96%B0%E7%9A%84%E6%95%B0%E7%BB%84%E5%B9%B6%E8%BF%94%E5%9B%9E%E3%80%82 %E4%B8%8B%E6%96%B9%E6%8F%90%E4%BE%9B%E4%BA%86%E4%B8%80%E4%B8%AA%E6%95%B0%E7%BB%84%EF%BC%8C%E5%A6%82%E6%9E%9C%E6%88%91%E4%BB%AC%E6%83%B3%E5%B0%86%E5%85%B6%E4%B8%AD%E7%9A%84%E6%AF%8F%E4%B8%80%E4%B8%AA%E5%85%83%E7%B4%A0%E7%BF%BB%E5%80%8D%EF%BC%8C%E6%88%91%E4%BB%AC%E5%8F%AF%E4%BB%A5%E4%BD%BF%E7%94%A8 map %E5%92%8C forEach %E6%9D%A5%E8%BE%BE%E5%88%B0%E7%9B%AE%E7%9A%84%E3%80%82)

[JavaScript Array map() 方法 (w3school.com.cn)](https://www.w3school.com.cn/jsref/jsref_map.asp)

|  |
| --- |
| * forEach(): 针对每一个元素执行提供的函数(executes a provided function once for each array element)。 * map(): 创建一个新的数组，其中每一个元素由调用数组中的每一个元素执行提供的函数得来(creates a new array with the results of calling a provided function on every element in the calling array)。   到底有什么区别呢？  forEach()方法不会返回执行结果，而是undefined。也就是说，forEach()会修改原来的数组。而map()方法会得到一个新的数组并返回。  arr.forEach((num, index) => {  return arr[index] = num \* 2;}  );  let doubled = arr.map(num => {  return num \* 2;  }); |

* slice 和 splice 和 split

[(2条消息) js中slice,splice和split的区别\_tsw529的博客-CSDN博客\_js splice和split](https://blog.csdn.net/tsw529/article/details/89429347)

|  |
| --- |
| split（“”）输入时用到，字符串转为字符串数组  splice（index，howmany，item1，item2.。。）索引除开始删除或添加  slice（start，end）数组切割返回新数组，不修改原数组。 |

* substr 和 substring

[js中substring()和substr()的使用及区别-阿里云开发者社区 (aliyun.com)](https://developer.aliyun.com/article/793099)

|  |
| --- |
| substring（start，end） 包头不包尾  substr（start，length）后一个参数都可以省略，直接到结尾。都返回新对象。 |

* 数组方法 reduce，some，find

[Array.prototype.reduce() - JavaScript | MDN (mozilla.org)](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Array/Reduce)

[Array.prototype.find() - JavaScript | MDN (mozilla.org)](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Array/find)

[Array.prototype.findIndex() - JavaScript | MDN (mozilla.org)](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Array/findIndex)

[Array.prototype.includes() - JavaScript | MDN (mozilla.org)](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Array/includes)

[Array.prototype.some() - JavaScript | MDN (mozilla.org)](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Array/some)

* typeof null

[JavaScript typeof, null, 和 undefined | 菜鸟教程 (runoob.com)](https://www.runoob.com/js/js-typeof.html)

* typeof 和 instanceof

[面试官：typeof 与 instanceof 区别 · Issue #65 · febobo/web-interview (github.com)](https://github.com/febobo/web-interview/issues/65)

* 判断数组 、typeof 和 instanceof

[在JavaScript中，如何判断数组是数组？ - SegmentFault 思否](https://segmentfault.com/a/1190000006150186" \l ":~:text=%E6%AF%94%E5%A6%82%E6%88%91%E4%BB%AC%E4%BB%8A%E5%A4%A9%E8%A6%81%E8%AE%A8%E8%AE%BA%E7%9A%84%EF%BC%8C%E5%9C%A8JavaScript%E5%BD%93%E4%B8%AD%E5%A6%82%E4%BD%95%E5%88%A4%E6%96%AD%E4%B8%80%E4%B8%AA%E6%95%B0%E7%BB%84%E6%98%AF%E6%95%B0%E7%BB%84%E3%80%82,JavaScript%E6%9C%89%E4%BA%94%E7%A7%8D%E6%96%B9%E6%B3%95%E5%8F%AF%E4%BB%A5%E7%A1%AE%E5%AE%9A%E4%B8%80%E4%B8%AA%E5%80%BC%E5%88%B0%E5%BA%95%E6%98%AF%E4%BB%80%E4%B9%88%E7%B1%BB%E5%9E%8B%EF%BC%8C%E5%88%86%E5%88%AB%E6%98%AFtypeof%E8%BF%90%E7%AE%97%E7%AC%A6%EF%BC%8Cconstructor%E6%B3%95%EF%BC%8Cinstanceof%E8%BF%90%E7%AE%97%E7%AC%A6%EF%BC%8CObject.prototype.toString%E6%96%B9%E6%B3%95%E4%BB%A5%E5%8F%8AArray.isArray%E6%B3%95. typeof%E6%98%AFjavascript%E5%8E%9F%E7%94%9F%E6%8F%90%E4%BE%9B%E7%9A%84%E5%88%A4%E6%96%AD%E6%95%B0%E6%8D%AE%E7%B1%BB%E5%9E%8B%E7%9A%84%E8%BF%90%E7%AE%97%E7%AC%A6%EF%BC%8C%E5%AE%83%E4%BC%9A%E8%BF%94%E5%9B%9E%E4%B8%80%E4%B8%AA%E8%A1%A8%E7%A4%BA%E5%8F%82%E6%95%B0%E7%9A%84%E6%95%B0%E6%8D%AE%E7%B1%BB%E5%9E%8B%E7%9A%84%E5%AD%97%E7%AC%A6%E4%B8%B2%EF%BC%8C%E4%BE%8B%E5%A6%82%EF%BC%9A)

* 自定义事件监听

[(2条消息) js自定义事件、事件监听、触发事件\_海边、lucky的博客-CSDN博客\_js自定义监听事件](https://blog.csdn.net/qq_37338658/article/details/103577259)

[自定义事件的触发dispatchEvent - 简书 (jianshu.com)](https://www.jianshu.com/p/5f9027722204)

|  |
| --- |
| // 创建事件  let myEvent = new CustomEvent('addCountReady', {  detail: {  result: '200',  data: {},  msg: '成功'  }  })  // 监听事件  window.addEventListener('addCountReady', function (result) {  console.log(result.detail)  })  // 触发事件  if(window.dispatchEvent) {  window.dispatchEvent(myEvent);  } else {  window.fireEvent(myEvent);  }  fireEvent是IE提供的一种方法 |

* eventEmitter

[(2条消息) JavaScript 事件派发器 EventEmitter\_杏子\_1024的博客-CSDN博客](https://blog.csdn.net/weixin_44135121/article/details/105819489)

* PWA
* jsBridge