# JavaScript 问题答案

下面所有的代码都是个人作答，可能存在理解错误或者答案错误的地方，欢迎大家在[项目下方](https://github.com/springHyc/InterviewLibrary)或者[项目的 issues](https://github.com/springHyc/InterviewLibrary/issues)中留言指正。

## 0. this 指向性问题总结。下面代码中分别输出什么？

由于在很多地方都遇到了不同形式的this指向性问题，所以干脆在这里把我遇到的题目都总结一下，也是为了更加集中的复习。

// code 1  
var length = 10;  
function fn() {  
 alert(this.length);  
}  
var obj = {  
 length: 5,  
 method: function() {  
 fn();  
 }  
};  
obj.method(); // 10 隐式绑定的函数会丢失绑定对象，会应用默认绑定。`obj.method()`它实际上引用的是函数fn本身。  
  
// code 2  
var num = 100;  
var obj = {  
 num: 200,  
 inner: {  
 num: 300,  
 print: function() {  
 console.log(this.num);  
 }  
 }  
};  
  
obj.inner.print(); //300  
  
var func = obj.inner.print;  
func(); //100 默认绑定，绑定的是全局对象  
  
obj.inner.print(); //300 隐式绑定，当函数引用有上下文对象时，隐式绑定规则会把函数调用的`this`绑定到这个上下文对象。  
  
(obj.inner.print = obj.inner.print)(); //100 赋值对象返回的是一个函数，默认绑定，绑定的是全局对象  
  
// code 3  
function foo() {  
 console.log(this.a);  
}  
var obj2 = { a: 42, foo: foo };  
var obj1 = { a: 2, obj2: obj2 };  
obj1.obj2.foo(); // 42 对象属性引用链中只有上一层或者说最后一层在调用位置中起作用。  
  
var obj3 = { a: 2 };  
foo.call(obj3); // 2  
  
var bar = function() {  
 foo.call(obj3);  
};  
bar(); // 2  
setTimeout(bar, 100); // 2  
bar.call(window); // 2 硬绑定的bar不可能在修改它的this  
  
var obj4 = { a: 3, foo: foo };  
obj2.foo(); // 42  
obj4.foo(); // 3  
obj2.foo.call(obj4); // 3  
obj4.foo.call(obj2); // 42 显示绑定比隐式绑定优先级高  
  
// code 4  
function foo() {  
 console.log(this.a);  
}  
var obj = {  
 a: 2,  
 foo: foo  
};  
var a = "oops, global"; // a是全局对象的属性  
setTimeout(obj.foo, 100); // "oops, global"  
obj.foo(); // 2  
  
// code 5 (new绑定)  
function foo(a) {  
 this.a = a;  
}  
var bar = new foo(2);  
console.log(bar.a); // 2 new绑定  
  
var obj1 = { foo: foo };  
var obj2 = {};  
  
obj1.foo(2);  
console.log(obj1.a); // 2  
  
obj1.foo.call(obj2, 3);  
console.log(obj2.a); // 3  
  
var bar = new obj1.foo(4);  
console.log(obj1.a); // 2  
console.log(bar.a); // 4 new绑定比隐式绑定优先级高  
  
// code 6  
  
function foo() {  
 console.log(this.a);  
}  
  
var a = 2;  
  
// 如果你把null或者undefined作为this的绑定对象传入call\apply\bind，这些值在调用的时候会被忽略，实际应用的是默认绑定规则。  
foo.call(null); // 2  
var bar = foo.bind(null);  
bar(); // 2  
foo.apply(undefined); // 2  
  
// code 7 箭头函数  
  
function foo() {  
 return a => console.log(this.a);  
}  
  
var obj1 = { a: 2 };  
var obj2 = { a: 3 };  
var bar = foo.call(obj1);  
bar.call(obj2); // 2 箭头函数是根据外层（函数或者全局）作用域来决定this。并且绑定后无法修改。  
  
// code 8 独立函数调用  
  
var a = 1;  
function main() {  
 alert(a); // undefined  
 var a = 2;  
 alert(this.a); // 1 这个地方的this一定指的全局对象，使用的是默认绑定规则  
 alert(a); // 2  
}  
main();  
  
//----------  
function foo() {  
 console.log(this.a);  
}  
var a = 2;  
foo(); // 2  
  
// code 9 网上改编自360的题目  
  
window.val = 1;  
var obj = {  
 val: 2,  
 dbl: function() {  
 this.val \*= 2;  
 val \*= 2;  
 console.log(val);  
 console.log(this.val);  
 }  
};  
// 说出下面的输出结果  
obj.dbl(); // 2 4 单独调用obj.dbl(); 其中this.val= 2； 而val 则是window.val => 第一个打印的结果是：1\*2 ,第二个打印的结果是： 2\*2  
var func = obj.dbl;  
func(); // 函数是引用，其中this.val === val === window.val === 2  
  
输出的结果是： 2 4 8 8  
  
// ----箭头函数中的this----  
// code 10  
var obj = {  
 say: function () {  
 setTimeout(() => {  
 console.log(this)  
 });  
 }  
 }  
 obj.say(); // obj  
 > 此时的 this继承自obj, 指的是定义它的对象obj, 而不是 window!  
  
// code 11  
var obj = {  
say: function () {  
 var f1 = () => {  
 console.log(this); // obj  
 setTimeout(() => {  
 console.log(this); // obj  
 })  
 }  
 f1();  
 }  
}  
obj.say()  
> 因为f1定义时所处的函数 中的 this是指的 obj, setTimeout中的箭头函数this继承自f1, 所以不管有多层嵌套,都是 obj  
  
// code 12  
  
var obj = {  
say: function () {  
 var f1 = function () {  
 console.log(this); // window, f1调用时,没有宿主对象,默认是window  
 setTimeout(() => {  
 console.log(this); // window  
 })  
 };  
 f1();  
 }  
}  
obj.say()  
  
// code 13  
var obj = {  
say: function () {  
 'use strict';  
 var f1 = function () {  
 console.log(this); // undefined  
 setTimeout(() => {  
 console.log(this); // undefined  
 })  
 };  
 f1();  
 }  
}  
obj.say()  
  
> 严格模式下,没有宿主调用的函数中的this是undefined!!!所以箭头函数中的也是undefined!  
// ----箭头函数中的this----

## 1. 字符串实现倒序

个人答案

使用最简单的 JS 操作就可以完成。下面是简单的实现 code：

function reverseString(str) {  
 var strArray = str.split(""); // 使用空字符串来分割成字符数组  
 return strArray.reverse().join(""); // 反转并连接  
}  
reverseString("hello");

## 2. 下面这段代码的执行结果是什么？

for (var i = 1; i <= 5; i++) {  
 setTimeout(function () {  
 console.log(i);  
 }, i \* 1000);  
}

继续问，怎么实现期望一共返回 1-5，5 个值，并且一秒返回一个值？

这段代码,我们预期的结果是：分别输出数字 1~5，每秒一次，每次一个。但是实际结果却是：这段代码在运行时会以每秒一次的频率输出五次 6。

这是为什么？

首先解释 6 是从哪里来的。这个循环的终止条件是 i 不再 <=5。条件首次成立时 i 的值是 6。因此，输出显示的是循环结束时 i 的最终值。仔细想一下，这好像又是显而易见的，延迟函数的回调会在循环结束时才执行。事实上， 当定时器运行时即使每个迭代中执行的是 setTimeout(.., 0)，所有的回调函数依然是在循 环结束后才会被执行，因此会每次输出一个 6 出来。

因为外层作用域中只有一个i，这个i被封闭进去，而不是每个迭代的函数会封闭一个新的i.

---以上解释来自《你不知道的 JavaScript（上卷）》第五章

这样就可以实现期望一共返回 1-5，5 个值，并且一秒返回一个值

for (let i = 1; i <= 5; i++) {  
 setTimeout(function timer() {  
 console.log(i);  
 }, i \* 1000);  
}  
  
// or  
  
for (var i = 1; i <= 5; i++) {  
 (function (j) {  
 setTimeout(function () {  
 console.log(j);  
 }, j \* 1000);  
 })(i);  
}  
  
// or  
  
for (var i = 1; i <= 5; i++) {  
 let j = i;  
 setTimeout(function timer() {  
 console.log(j);  
 }, j \* 1000);  
}

let 声明只属于作用域块。for循环头部的let i不只是为 for 循环本身声明了一个i，而是为循环的每一次迭代都重新声明了一个新的i。这意味着 loop 迭代内部创建的闭包封闭的是每次迭代中的变量，就像期望的那样。

或者

for (var i = 1; i <= 5; i++) {  
 (function (j) {  
 setTimeout(function timer() {  
 console.log(j);  
 }, j \* 1000);  
 })(i);  
}

在迭代内使用 IIFE 会为每个迭代都生成一个新的作用域，使得延迟函数的回调可以将新的 作用域封闭在每个迭代内部，每个迭代中都会含有一个具有正确值的变量供我们访问。

## 3. JavaScript 的基本数据类型有哪些？如何准确的检测数据类型？

string number Boolean null undefined object 还有 es6 新增的 symbol，主要特征是唯一性

继续问：typeof 的返回值都有哪些？ string number boolean object undefined object(和上面是一一对应的) function 还有 array 也会返回 object typeof Symbol()返回的是"symbol" typeof(null) // "object",检测null的方式是my\_value === null

继续问：为什么 es6 推出了 Symbol?它的作用是什么？（搜狗） 是全新的，第七种原始数据类型。被创建后不能改变，并且是唯一的。 Q: 为什么推出？ A: Symbol 的目的就是为了实现一个唯一不重复不可变的值，任何一个 Symbol 都是唯一的，不会和其他任何 Symbol 相等。很多时候我们需要用到唯一不重复的值，但是每次创建的值很大可能会被重复 => 创建了 Symbol 类型。 Q:它的作用是什么？ A: 提供一个唯一不重复不可变的值。

备注： typeof 是一个一元运算，放在一个运算数之前，运算数可以是任意类型。它返回值是一个字符串，该字符串说明运算数的类型。

## 4. 以下代码执行结果分别是什么？

* 3 + "3" // '33'
* "23" > "3" // false
* var b = true && 2; // 2
* "abc123".slice(2, -1) // 'c12' // 如果 start是负数（绝对值小于字符串长度），则 start+ 字符串长度。// 不会交换 start 和 stop
* "abc123".substring(2, -1) // 'ab' //如果 start or stop 是负数或 NaN，会把它当成 0 对待;如果 start > stop,则会交换这两个参数 // 交换 start 和 stop

解释：

**String.slice() 和 String.substring(),String.substr()的区别**

Syntax: string.slice(start, stop); Syntax: string.substring(start, stop); 返回一个字符串，左包含，右不包含

Syntax:String.substr(start, num); 返回字符串,包含 start 开始,num 为字符数

slice VS substring

相同：

1. 如果 start == stop,return 一个空字符串
2. stop 如果被省略，则直接扫至字符串尾
3. 如果 start 或 stop 大于了字符串长度，则会被替换成字符串长度

不同： substing():

* 1.如果 start > stop,则会交换这两个参数
* 2.如果 start or stop 是负数或 NaN，会把它当成 0 对待

slice():

* 1.如果 start > stop,不会交换这两个参数，返回空字符串””
* 2.如果 start,or stop 是负数，且绝对值小于字符串长度，则开头start+字符串长度
* 3.如果 start or stop 是负数，且绝对值大于字符串长度，则当作 0 处理

## 5. 以下代码执行结果是什么？

* 1

js var foo = 1, bar = 2, j, test; test = function (j) { j = 5; var bar = 5; console.log(bar); // 5 foo = 5; }; test(10); console.log(foo); // 5 改变的全局变量 console.log(bar); // 2 由于函数作用域对全局作用域的隐藏，所以只有在test函数内部，bar=5,并不能影响到全局中的bar console.log(j); // undefined test(10)函数调用的时候，是函数内部的参数j接收到了10，但是它也是函数作用域内的变量，并不会改变全局作用域中的j。

这个题目还有一个类似的题目：这个考察的是，数组和对象都是引用复制

js var j = [1, 2, 3]; test = function (j) { j.push(4); }; test(j); console.log(j); // [1,2,3,4],因为test(j)中的j是对[1,2,3]的引用复制给function(j)中的j,而在test函数内部，通过引用改变的是[1,2,3]这是数组本身，所以console.log(j); 为 [1,2,3,4]

* 2

js for (var i = 0; i < 5; i++) { window.setTimeout(function () { console.log(i); // }, 1000); } console.log("i=" + i); // 先输出“i=5”,然后再隔1s后输出一个5个5

只有

js for (var i = 0; i < 5; i++) { window.setTimeout(function () { console.log(i); }, i \* 1000); // 只有i\*1000才会每隔1S输出一个 }

"i=5" 5 5 5 5 5

setTimeout 语法: var timeoutID = scope.setTimeout(function[, delay, param1, param2,...]); var timeoutID = scope.setTimeout(code[, delay]);

* function 是你想要在 delay 毫秒之后执行的函数。
* delay (可选) 延迟的毫秒数 (一秒等于 1000 毫秒)，函数的调用会在该延迟之后发生。如果省略该参数，delay 取默认值 0。实际的延迟时间可能会比 delay 值长，原因请查看 Reasons for delays longer than specified。
* param1, ..., paramN (可选) 附加参数，一旦定时器到期，它们会作为参数传递给 function 或 执行字符串（setTimeout 参数中的 code）

for (var i = 0; i < 5; i++) {  
 window.setTimeout(  
 function (j) {  
 console.log(i + j); // 每隔100ms输出一个13  
 },  
 100,  
 8  
 );  
}  
console.log("i=" + i); // 5

* 3

js var length = 10; function fn() { alert(this.length); } var obj = { length: 5, method: function () { fn(); }, }; obj.method(); // 10 , 对fn间接引用，调用这个函数会应用默认的绑定规则

这个考察的是this的指向问题。

还有一个类似的问题：

var num = 100;  
  
var obj = {  
 num: 200,  
 inner: {  
 num: 300,  
 print: function () {  
 console.log(this.num);  
 },  
 },  
};  
  
obj.inner.print(); //300  
  
var func = obj.inner.print;  
func(); //100  
  
obj.inner.print(); //300  
  
(obj.inner.print = obj.inner.print)(); //100  
  
// 问题：第一个和第三个有什么区别？第三个和第四个有什么区别？

1.第一个和第三个没有区别，运行的都是 obj.inner.print()，里面的 this 指向 obj.inner.num

2.第四个，首先你要知道一点，复制操作，会返回所赋的值。

var a;  
console.log((a = 5)); //5

所以(obj.inner.print = obj.inner.print)的结果就是一个函数，内容是

function () {  
 console.log(this.num);  
}

在全局下运行这个函数，里面的 this 指向的就是 window，所以(obj.inner.print = obj.inner.print)();的结果就是

var num = 100;  
function () {  
 console.log(this.num);  
}()  
// 100

1. 第二个

赋值操作，fun 完全就是一个函数引用，这个引用丢失了函数原本所在的上下文信息，所以最终是在全局上下文中运行

function() {  
 console.log(this.num);  
}

所以这个时候 num 是全局的 num,也就是 100

* 4

js function Foo() { this.value = 42; } Foo.prototype = { method: function () { return true; }, }; function Bar() { var value = 1; return { method: function () { return value; }, }; } Foo.prototype = new Bar(); console.log(Foo.prototype.constructor); // ƒ Object() { [native code] } 指向内置的Object()方法 console.log(Foo.prototype instanceof Bar); // false var test = new Foo(); console.log(test instanceof Foo); // true console.log(test instanceof Bar); // false console.log(test.method()); // 1

1: Foo.prototype 的.constructor 属性只是 Foo 函数在声明时的默认属性。如果你创建了一个新对象并替换了函数默认的.prototype 对象引用，**那么新对象并不会自动获得.constructor 属性**。（这也是原型继承时，需要重新赋值的原因。） Foo.prototype 没有.constructor 属性，所以他会委托[[Prototype]]链上的委托连顶端的 Object.prototype。这个对象有.constructor 属性，只想内置的 Object(...)函数。

* 5

js if (!("sina" in window)) { var sina = 1; } console.log("sina:", sina); // undefined

由于 JavaScript 在编译阶段会对声明进行提升，所以上述代码会做如下处理：

js var sina; if (!("sina" in window)) { sina = 1; } console.log("sina:", sina);

声明被提升后，window.sina的值就是 undefined，但是!("sina" in window)这段代码的运行结果是true，所以sina = 1;就不会被执行，所以 本题目的输出结果是undefined。

* 6

js var t1 = new Date().getTime(); var timer1 = setTimeout(function () { clearTimeout(timer1); console.info("实际执行延迟时间：", new Date().getTime() - t1, "ms"); // 500+ms }, 500);

需要查看setTimeout的运行机制。

阮一峰老师有篇不错的文章（[JavaScript 运行机制详解：再谈 Event Loop](http://www.ruanyifeng.com/blog/2014/10/event-loop.html)），我就不再重复造轮子了；如果觉得太长不看的话，楼主简短地大白话描述下。

JavaScript 都是单线程的，单线程就意味着，所有任务需要排队，前一个任务结束，才会执行后一个任务。如果前一个任务耗时很长，后一个任务就不得不一直等着。如果排队是因为计算量大，CPU 忙不过来，倒也算了，但是很多时候 CPU 是闲着的，因为 IO 设备（输入输出设备）很慢（比如 Ajax 操作从网络读取数据），不得不等着结果出来，再往下执行。 JavaScript 语言的设计者意识到，这时主线程完全可以不管 IO 设备，挂起处于等待中的任务，先运行排在后面的任务。等到 IO 设备返回了结果，再回过头，把挂起的任务继续执行下去。于是，所有任务可以分成两种，一种是同步任务（synchronous），另一种是异步任务（asynchronous）。同步任务指的是，在主线程上排队执行的任务，只有前一个任务执行完毕，才能执行后一个任务；异步任务指的是，不进入主线程、而进入"任务队列"（task queue）的任务，只有"任务队列"通知主线程，某个异步任务可以执行了，该任务才会进入主线程执行。具体来说，异步执行的运行机制如下。（同步执行也是如此，因为它可以被视为没有异步任务的异步执行。） （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。

一段 js 代码（里面可能包含一些 setTimeout、鼠标点击、ajax 等事件），从上到下开始执行，遇到 setTimeout、鼠标点击等事件，异步执行它们，此时并不会影响代码主体继续往下执行(当线程中没有执行任何同步代码的前提下才会执行异步代码)，一旦异步事件执行完，回调函数返回，将它们按次序加到执行队列中,加入到队列中，只是在确定额 的时间候调用，但是并不一定立马执行。

综上所述， 500ms 后异步任务执行完毕，然后就在“任务队列”之中防止一个事件。但是，需要主线程的 “执行栈”中所有的同步任务执行完毕后，“任务队列”中的时间才会开始执行。所以，500+ms 后才真正的执行输出。

* 7 js function SINA() { return 1; } var SINA; console.log(typeof SINA); // function

重复声明被忽略掉了，所以var SINA并没有起到作用，而是被忽略掉了。

* 8

var sinaNews = {  
 name: "sinNewsName",  
 test: function () {  
 console.log("this.name:", this.name, "//");  
 },  
};  
setTimeout(sinaNews.test, 500); // "this.name: //"

回调函数丢失 this 绑定

## 6. 如何对数组进行排序？如：[2, [1,2], 3, "2", "a", "b", "a", [1, 2]]，重排序后[2, [1, 2], 3, "2", "a", "b"]

个人答案

function deleteCopy(arr) {  
 var map = {};  
 var newarr = [];  
 arr.forEach((item) => {  
 if (!map[JSON.stringify(item)]) {  
 newarr.push(item);  
 map[JSON.stringify(item)] = 1;  
 }  
 });  
 return newarr;  
}

## 7. 要给羡慕所有的 li 元素保定 click 时间，在鼠标点击每个 li 的时候 alert 该 li 里面的内容；且在鼠标离开外部 ul 元素范围的时候弹出一个 alert 提示、（实现时请注意代码执行小路及浏览器兼容性，不要使用现成的框架库，用原生 js 编写完成）

<ul id="ulItem">  
 <li>内容1</li>  
 ......此处省略1000+个li对象(注：不要使用循环绑定，如果使用循环绑定1000+的绑定事件会很慢)......  
 <li>内容n</li>  
</ul>

### 8. 下面代码打印出什么？

const a = 2;  
console.log(a); // 2  
a = 3; // TypeError: Assignment to constant variable.

常量的值在设定之后就不能再更改。

顺便记录一下TypeError与ReferenceError的区别： ReferenceError： 如果 RHS 查询（取值查询）在所有前台的作用域中找寻不到所需的变量，引擎就会抛出ReferenceError异常。 ReferenceError同作用域判别失败相关，而TypeError则代表作用域判别成功了，但是对结果的操作是非法或者不合理的。比如视图对一个非函数类型的值进行函数调用，或者引用null或undefined类型的值中的属性，那么引擎会抛出类型异常，TypeError.

### 9. 下面代码分别输出什么？

function foo() {  
 "use strict";  
 console.log(this.a);  
}  
  
function bar() {  
 console.log(this.a);  
}  
  
var a = "this is a 'a'";  
  
bar(); // "this is a 'a'"  
foo(); // "TypeError: Cannot read property 'a' of undefined

接下来我们可以看到当调用 foo() 时，this.a 被解析成了全局变量 a。为什么?因为在本 例中，函数调用时应用了 this 的默认绑定，因此 this 指向全局对象。

bar() 调用使用严格模式(strict mode)，那么全局对象将无法使用默认绑定，因此 this 会绑定 到 undefined

### 10. 根据以下代码，写出结果

// 第一组  
alert(a);  
a();  
var a = 3;  
function a() {  
 alert(10);  
}  
alert(a);  
a = 6;  
a();  
  
//------------分割线------------------  
// 第二组  
alert(a);  
a();  
var a = 3;  
var a = function () {  
 // 函数表达式  
 alert(10);  
};  
alert(a);  
a = 6;  
a();

考点其实就两个，第一变量声明提前，第二函数声明优先于变量声明！但是函数表达式并不会被提升！！

第一部分的代码片段会被引擎理解为如下形式：

function a() {  
 // 变量声明、函数声明会被提升，函数声明有先有变量申明  
 alert(10);  
}  
var a;  
alert(a);  
a();  
a = 3;  
alert(a);  
a = 6;  
a();

第一部分运行结果：

1.函数声明优先于变量声明，所以，刚开始，a 就是 function a(){alert(10)} ，就会看到这个函数。

2.a()，执行函数，就是出现 alert(10)。

3.执行了 var a=3; 所以 alert(a)就是显示 3 。

4.由于 a 不是一个函数了，所以往下在执行到 a()的时候， 报错。

第二部的代码片段会被引擎理解为：

var a;  
alert(a);  
a();  
a = 3;  
a = function () {  
 // 函数表达式  
 alert(10);  
};  
alert(a);  
a = 6;  
a();

第二部分运行结果： 1.underfind

2.报错在之前说过，预解析是把带有 var 和 function 关键字的事先声明，但不会赋值。所以一开始是 underfind，然后报错是因为执行到 a()的时候，a 并不是一个函数。因为报错了，错误会冒泡上浮，知道 catch 或者是最上层，所以不会继续执行下去。

备注

//函数表达式，和变量声明同等  
var a = function () {  
 alert(10);  
};  
//函数声明，优于变量声明  
function a() {  
 alert(10);  
}

## 11. 给出一个字符串，找到里面重复最多的字符？

个人答案, 如果有重复数相同的，那么不会记录后面的字符

function findMax(str) {  
 var map = {},  
 max = { num: 0 };  
  
 for (var index in str) {  
 if (map[str[index]]) {  
 map[str[index]]++;  
 } else {  
 map[str[index]] = 1;  
 }  
 if (map[str[index]] > max.num) {  
 max.num = map[str[index]];  
 max.key = str[index];  
 }  
 }  
 console.log(`max num is ${max.num}, the key is ${max.key}`);  
}

## 12.写一个函数实现n!

实现阶乘，最开始我只想到了递归的方式。还一开始写的有问题，后来修改了才没问题的。

### 递归方式

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

### while 循环方式

function factorial(n) {  
 var result = 1;  
  
 while (n > 1) {  
 result = result \* n;  
 n--;  
 }  
 return result;  
}

## 13. 下面写出答案

a = 1;  
b = 1;  
a == b; // true  
name1 = { a: 1 };  
name2 = { a: 1 };  
name1 == name2; //false

## 14. 给一个<div>添加点击事件的方法？

* 第一种，可以直接添加onclick事件 <div onclick="alert('成功')">点击</div>
* 第二种，DOM 级事件处理

<div id="div">  
<script type="text/javascript">  
 var divM = document.getElementById("div");  
 divM.onclick = function () {  
 alert("成功");  
 }  
 </script>

DOM 级事件处理 2

<script type="text/javascript">  
 var oDiv = document.getElementById("div");  
 oDiv.addEventListener("click", function(){  
 alert("成功");  
 });  
 </script>

* 第三种， jQuery 的方式来添加

$("#div").on("click", function () {  
 alert("成功");  
});

## 15. ajax 中有 3 个请求，如何顺序实现这 3 个请求？

## 16. localStorage、sessionStorage 的区别？还要去了解一下 Cookie

在 HTML5 之前，主要是使用 cookies 存储，cookies 的缺点有：需要再请求头上带着数据，存储大小不过在 4k 之内。

HTML5 web 存储，是一个比 cookie 更好的本地存储方式。

使用 HTML5 可以在本地存储用户的浏览数据。

客户端存储数据的两个对象为：

* localStorage - 没有时间限制的数据存储
* sessionStorage - 针对一个 session 的数据存储

他们拥有相同的 API：

* 保存数据:localStorage.setItem(key,value)
* 读取数据：localStorage.getItem(key)
* 删除单个数据：localStorage.removeItem(key)
* 得到某个索引的 key: localStorage.key(index)

localStorage、sessionStorage、Cookie 共同点：都是保存在浏览器端，且同源的。

### localStorage

localStorage 对象存储的数据没有时间限制。**第二天、第二周或者下一年之后，数据依然可用。**

localStorage 生命周期是永久的，这意味着除非用户显示在浏览器提供的 UI 上清除 localStorage 的信息，苟泽这些信息将永远存在。存放数据大小为一般为 5MB，而且它仅在客户端（即浏览器）中保存，不参与和服务器的通信。

### sessionStorage

sessionStorage 方法针对一个 session 进行数据存储。**当用户关闭浏览器窗口后，数据就会被删除。**

存放数据大小为一般为 5MB，而且它仅在客户端（即浏览器）中保存，不参与和服务器的通信。

### localStorage 和 sessionStorage 作用域不同

* 不同浏览器无法共享 localStorage 和 sessionStorage 中的信息。
* 相同浏览器的不同页面间可以共享相同的 localStorage（页面属于相同域名和端口）；但是不同页面或标签页（仅指顶级窗口）间无法共享 sessionStorage 的信息。

### Cookie

声明周期为只在设置的 cookie 过期之前一直有效，及时窗口或浏览器关闭。存储大小为 4k 左右，再长了会被截断。有个数限制（各个浏览器不同），一般不能超过 20 个。与服务器端通信：每次都会携带在 http 头中，**如果使用 cookie 保存过多数据就会带来性能问题。**

## 17. 数组的哪些操作会改变数组？

* pop
* push
* shift
* unshift
* sort
* reverse
* splice(删除)

## 18. 事件代理的原理，请使用原生 js 实现一个

## 19. 请说明下列方法功能：

|  |  |  |  |
| --- | --- | --- | --- |
| 方法 | 功能 | 是否改变原数组 | 备注 |
| push | 添加元素到数组的末尾 | √ |  |
| pop | 删除数组末尾的元素 | √ |  |
| shift | 删除数组头部的元素 | √ | 第一次写错啦 |
| unshift | 添加元素到数组的头部 | √ | 第一次写错啦 |
| splice(pos,n) | 通过索引,从 pos 位置开始删除 n 个元素 | √ | 第一次写错啦 |
| slice(start,end) | 返回一个新的数组，包含从start到end(不包括该元素）的数组中的元素。 | × | 和 splice 记混啦 |
| sort | 数组排序 | √ |  |
| reverse | 数组倒序 | √ | 是改变原数组的 |
| slice() | 复制整个数组 | × |  |
| indexOf | 找出某个元素在数组中的索引 | × |  |

## 20. apply 和 call 的作用和区别?

apply(thisObj[, argArray]); call(thisObj[, arg1[, arg2[, [,...argN]]]])

都可以用来代替另一个对象调用一个方法，将一个函数的对象上下文从初始的上下文改变为由 thisObj 指定的新对象。

不同之处：

* apply 最多有两个参数，新 this 对象和一个数组 argArray(参数)。
* call 可以接受多个参数，第一个参数与 apply 一样，后面则是一串参数列表。

第一个参数与 apply 一样，后面则是一串参数列表

## 21. 写一个函数递归调用的例子

我写的是!n

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

## 22. 写出下列代码的运行结果：

var a = 1;  
function main() {  
 alert(a); // undefined  
 var a = 2;  
 alert(this.a); // 1 // 这个地方我写错了，为什么呢  
 alert(a); // 2  
}  
main();

## 23. 参数传递中引用和复制的区别？

JavaScript 的值类型和引用类型 值类型：

* Number
* Boolean
* null
* undefined

引用类型：

* 对象
* 数组
* 函数

值传参针对的是基本类型。值传参可以理解为复制变量值。基本类型复制后俩个变量完全独立，之后任何一方改变都不会影响另一方。

引用传参针对的是引用类型，引用类型复制的是引用（即指针），之后的任何一方改变都会映射到另一方。

一个面试题目

function main() {  
 var re = [];  
 test(re);  
 console.log(re); // ?  
}  
  
function test(re) {  
 re = re.concat([1, 2, 3]);  
}  
  
main();

这个应该输出[]才对。 ECMAScript 中所有函数的参数都是按值传递的。也就是说，把函数外部的值复制给函数内部的参数，就和把值从一个变量复制到另一个变量一样。**但是如果传递的是对象，即使这个变量是按值传递的， obj 也会按引用来访问同一个对象。** 对象变量它里面的值是这个对象在堆内存中的内存地址，这一点你要时刻铭记在心！因此它传递的值也就是这个内存地址，这也就是为什么函数内部对这个参数的修改会体现在外部的原因了，因为它们都指向同一个对象呀。 **保存对象的变量，它里面装的值是这个对象在堆内存中的地址。so => 在函数内部的对象改变了，函数外部的对象也会跟随着一起变**

## 24. es6 的特性有什么？

* 箭头函数
* 类的支持
* 解构
* 不定参数
* 默认参数
* 块级作用域 let
* 模板字符串
* 多行字符串
* 拆包表达式
* promise
* 改进的对象表达式
* 模块化
* 延展操作符
* 更新的数据结构（map 和 set）
* generator

### 默认参数

function fn(name="world") {...}

### 模板字符串

var url = `http://login?username=${userName}&password=${password}`;

### 多行字符串

一个比较好的语法糖。

var roadPoem = `Then took the other, as just as fair,  
 And having perhaps the better claim  
 Because it was grassy and wanted wear,  
 Though as for that the passing there  
 Had worn them really about the same,`;

### 解构赋值

ES6 允许按照一定模式，从数组和对象中提取值，对变量进行赋值，这被称为解构（Destructuring）。

var [first, second, third] = someArray;  
  
//------  
let [a, b, c] = [1, 2, 3];  
//等同于  
let a = 1;  
let b = 2;  
let c = 3;  
  
// 对象的解构赋值：获取对象的多个属性并且使用一条语句将它们赋给多个变量。  
var { StyleSheet, Text, View } = React;

### 箭头函数

箭头最神奇的地方在于他会让你写正确的代码。比如，this 在上下文和函数中的值应当是相同的，它不会变化，通常变化的原因都是因为你创建了闭包。

使用箭头函数可以让我们不再用 that = this 或者 self = this 或者\_this = this 或者.bind(this)这样的代码。

### 延展操作符

* 通过它可以将数组作为参数直接传入函数

var people = ["Wayou", "John", "Sherlock"];  
function sayHello(people1, people2, people3) {  
 console.log(`Hello ${people1},${people2},${people3}`);  
}  
//改写为  
sayHello(...people); //输出：Hello Wayou,John,Sherlock

* 在函数定义时可以通过…rest 获取定义参数外的所有参数

function foo(a, b, ...rest) {}

### Promise

现在在 ES6 中终于有一个标准的 Promise 实现。

### 模块化

es6 有 import 和 export 运算符来实现了.

你在模块中的所有声明相对于模块而言都是寄存在本地的。如果你希望公开在模块中声明的内容，并让其它模块加以使用，你一定要导出这些功能。想要导出模块的功能有很多方法，其中最简单的方式是添加 export 关键字。你可以导出所有的最外层函数、类以及 var、let 或 const 声明的变量。

模块与脚本还是有两点区别

* “在 ES6 模块中，无论你是否加入“use strict;”语句，默认情况下模块都是在严格模式下运行。” 摘录来自: InfoQ 中文站. “深入浅出 ES6。” iBooks.
* “在模块中你可以使用 import 和 export 关键字。”

## 25. 箭头函数相比于普通函数的优点？

* 简洁
* this function 传统定义的函数，this 指向随着调用环境的改变而改变，而箭头 函数中的指向则是固定不变，一直指向定义环境的。 > 箭头函数在定义之后，this 就不会发生改变了，无论用什么样子的方式调用它，this 都不会改变 > 箭头函数没有它自己的 this 值，箭头函数内的 this 值继承自外围作用域。
* 构造函数 箭头函数固然好用，但是不能用于构造函数，即不能被 new 一下
* 变量提升 由于 js 的内存机制，function 的级别最高，而用箭头函数定义函数的时候，需要 var(let const 定义的时候更不必说)关键词，而 var 所定义的变量不能得到变量提升，故箭头函数一定要定义于调用之前！

## 26. for-of 的原理？

for-of 通过方法调用(遍历器方法)来实现集合的遍历。数组、Maps、Sets 以及其他我们讨论过的对象之间有个共同点：有迭代器方法。

另外一种解释 for-of 工作原理：迭代器(iterator)有一个 next 方法，for 循环会不断调用这个 iterator.next 方法来获取下一个值,直到返回值中的 done 属性为 true 的时候结束循环。

另外一种解释 JavaScript 原有的表示“ 集合” 的数据结构， 主要是数组（ Array） 和对象（ Object）， ES6 又添加了 Map 和 Set。 这样就有了四种数据集合， 用户还可以组合使用它们， 定义自己的数据结构， 比如数组的成员是 Map， Map 的成员是对象。 这样就需要一种统一的接口机制， 来处理所有不同的数据结构。 遍历器（ Iterator） 就是这样一种机制。 它是一种接口， 为各种不同的数据结构提供统一的访问机制。 任何数据结构只要部署 Iterator 接口， 就可以完成遍历操作（ 即依次处理该数据结构的所有成员）。 Iterator 的作用有三个： 一是为各种数据结构， 提供一个统一的、 简便的访问接口。 二是使得数据结构的成员能够按某种次序排列； 三是 ES6 创造了一种新的遍历命令 for...of 循环， Iterator 接口主要供 for...of 消费。 Iterator 的遍历过程是这样的。 （ 1） 创建一个指针对象， 指向当前数据结构的起始位置。 也就是说， 遍历器对象本质上， 就是一个指针对象。 （ 2） 第一次调用指针对象的 next 方法， 可以将指针指向数据结构的第一个成员。 （ 3） 第二次调用指针对象的 next 方法， 指针就指向数据结构的第二个成员。 （ 4） 不断调用指针对象的 next 方法， 直到它指向数据结构的结束位置。 每一次调用 next 方法， 都会返回数据结构的当前成员的信息。 具体来说， 就是返回一个包含 value 和 done 两个属性的对象。 其中， value 属性是当前成员的值， done 属性是一个布尔值， 表示遍历是否结束。

## 27. 迭代器是什么？

生成器返回的是迭代器。

## 28. for in 和 for of 的区别是什么？

for-in 用于遍历对象更好 for-of 用于遍历数组更好

使用 for in 会遍历数组所有的可枚举属性，包括原型。所以**for-in 用于遍历对象更好**。

记住，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。

for-in 遍历数组的坏处？

* for-in赋给 index 的值不是实际的数字，而是字符串“0”、“1”、“2”，此时很可能在无意之间进行字符串算数计算，例如：“2” + 1 == “21”，这给编码过程带来极大的不便。
* 作用于数组的 for-in 循环体除了遍历数组元素外，还会遍历自定义属性。举个例子，如果你的数组中有一个可枚举属性 myArray.name，循环将额外执行一次，遍历到名为“name”的索引。就连数组原型链上的属性都能被访问到。
* 在某些情况下，这段代码可能按照随机顺序遍历数组元素

var list = [  
 { name: "hehe", age: 12 },  
 { name: "zz", age: 34 },  
];  
for (var item of list) {  
 console.log(`${item.name} is ${item.age} years old.`);  
}  
  
for (var index in list) {  
 console.log(`${list[index].name} is ${list[index].age} years old.`);  
}

## 29. 简述 arguments 的作用，在 es6 中更好的替代方案是什么?

不定参数和默认参数

## 30. 实现一个动物类，动物有吃饭，吼叫的方法，有眼睛、鼻子属性。动物类有子类：猫和狗。猫的叫声是喵喵，猫的眼睛是蓝色的，狗得见叫声是汪汪，狗的眼睛是棕色的。请用代码实现上述描述。

class Animal {  
 constructor(eye, nose) {  
 this.eye = eye;  
 this.nose = nose;  
 }  
 eat() {  
 return "吃东西";  
 }  
  
 howl() {  
 return "吼叫";  
 }  
}  
  
class Dog extends Animal {  
 constructor() {  
 super("棕色");  
 }  
  
 howl() {  
 return "wangwang";  
 }  
}  
  
class Cat extends Animal {  
 constructor() {  
 super("blue");  
 }  
  
 hol = () => "喵喵喵";  
}  
  
var dog = new Dog();  
console.log(dog.eye, dog.howl());  
  
var cat = new Cat();  
console.log(cat.eye, cat.howl());

## 31. 下面的执行结果是什么？

function Promise1() {  
 return new Promise(function (resolve, reject) {  
 for (let i = 0; i < 2; i++) {  
 console.log("111");  
 }  
 resolve(true);  
 });  
}  
function Promise2() {  
 return new Promise(function (resolve, reject) {  
 for (let i = 0; i < 2; i++) {  
 console.log("222");  
 }  
 resolve(true);  
 });  
}  
  
setTimeout(function () {  
 console.log("333");  
}, 0); // 这是是会执行的。考察的是异步执行，js的任务队列  
  
Promise.all([Promise1(), Promise2()]).then(function () {  
 console.log("All Done!");  
});

结果是：

"111";  
"111";  
"222";  
"222";  
"All Done!";  
"333";

## 32. 下面代码运行的结果是什么？

let obj = {  
 fun1: () => {  
 console.log("111");  
 },  
 fun2: () => {  
 this.fun1();  
 },  
};  
  
obj.fun2(); // 报错 this不是指向obj的，而是指向undefined ，箭头函数的this是指向外部作用域的

## 33. 下面代码会输出什么？

let arr = [1, 2, 3, 4];  
let it1 = arr[Symbol.iterator](); // 遍历器接口  
let res = it1.next();  
console.log(res);

结果是：

{  
 done: false,  
 value: 1  
}

## 34. 下面代码输出什么？ ？

### 题目 1

Promise.resolve(1)  
 .then((x) => x + 1)  
 .then((x) => {  
 throw new Error("my error");  
 })  
 .catch(() => 1)  
 .then((x) => x + 1)  
 .then((x) => console.log(x))  
 .catch(console.error);

结果是：

2

### 题目 2

setTimeout(() => {  
 console.log(1);  
}, 0);  
  
new Promise((resolve) => {  
 console.log(2);  
 resolve();  
 console.log(3);  
 4;  
}).then(() => {  
 console.log(4);  
});  
  
console.log(5);

结果是： 2 3 5 4 1

### 题目 3

var p1 = new Promise(function(resolve, reject) {  
 setTimeout(() =>reject(new Error('p1 中failure')) , 3000);  
})  
  
var p2 = new Promise(function(resolve, reject){  
 setTimeout(() => resolve(p1), 1000);  
});  
var p3 = new Promise(function(resolve, reject) {  
 resolve(2);  
});  
var p4 = new Promise(function(resolve, reject) {  
 reject(new Error('error in p4'));  
});  
  
1. p3.then(re => console.log(re));  
2. p4.catch(error => console.log(error));  
  
3. p2.then(null,re => console.log(re));  
4. p2.catch(re => console.log(re));

打印的顺序是：2， "error in p4 "这是立即打印出来的。

而 3S 后会打印出两个'p1 中 failure'。

如果 3 直接写成p2.then(re => console.log(re));是会报错，说没有捕捉到错误。

### 题目 4

var p1 = Promise.resolve(1);  
var p2 = new Promise((resolve) => {  
 setTimeout(() => resolve(2), 100);  
});  
var v3 = 3;  
var p4 = new Promise((resolve, reject) => {  
 setTimeout(() => reject("oops"), 10);  
});  
  
var p5 = new Promise((resolve) => {  
 setTimeout(() => resolve(5), 0);  
});  
var p1 = Promise.resolve(1);  
Promise.race([v3, p1, p2, p4, p5]).then((val) => console.log(val)); // 3  
Promise.race([p1, v3, p2, p4, p5]).then((val) => console.log(val)); // 1  
Promise.race([p1, p2, p4, p5]).then((val) => console.log(val)); // 1  
Promise.race([p2, p4, p5]).then((val) => console.log(val)); //5  
  
// 这些的打印顺序是什么？  
Promise.race([v3, p1, p2, p4, p5]).then((val) => console.log(val));  
Promise.race([p1, v3, p2, p4, p5]).then((val) => console.log(val));  
Promise.race([p1, p2, p4, p5]).then((val) => console.log(val));  
Promise.race([p2, p4, p5]).then((val) => console.log(val));  
  
Promise.resolve(6).then((re) => console.log(re)); // 6

打印顺序是：6 3 1 1 5

## 35. 说一下对 Promise 的理解

Promise 是一种封装和组合未来值的易于复用的机制。 Promise 采用的是异步任务队列的方式来处理异步。

* Promise 是一个构造函数，对回调函数的一种封装，对异步编程的一种改进。
* new 出来的 promise 对象。它有三种状态：pending（进行中），resolved（已完成），rejected（已失败），状态一旦发生，就不能改变。执行 new 的时候状态就开始变化。promise 对象身上有两个方法：then()，和 catch()。

promise 的特点：

对象的状态不受外界影响。

promise 的对待一个异步操作有 3 种状态：Pending（进行中）、Resolved（已完成，又称 Fulfilled）和 Rejected（失败）。

一旦状态改变，就不会再变，任何时候都可以得到这个结果。

promise 的好处：

* 代码结构更加扁平化，易读易理解，更加清晰明了。
* 能解决回调地狱的问题
* 可以将数据请求和业务逻辑分离开来。
* 便于维护管理
* 可以更好的捕捉错误
* es6 新增了 promise 来弥补回调的主要缺陷之一：缺少对可预测行为方式的保证。 i promise 的方法

|  |  |  |
| --- | --- | --- |
| 方法 | 作用 | 例子 |
| Promise.all | Promise.all 方法用于将多个 Promise 实例，包装成一个新的 Promise 实例:只有 p1、p2、p3 的状态都变成 fulfilled，p 的状态才会变成 fulfilled，此时 p1、p2、p3 的返回值组成一个数组，传递给 p 的回调函数。 | var p = Promise.all([p1,p2,p3]) |
| Promise.race() | Promise.race 方法同样是将多个 Promise 实例，包装城一个新的 Promise 实例: 上面代码中，只要 p1、p2、p3 之中有一个实例率先改变状态，p 的状态就跟着改变。那个率先改变的 Promise 实例的返回值，就传递给 p 的回调函数。 | var p = Promise.race([p1,p2,p3]) |
| Promise.resolve() | 有时需要将现有对象转为 Promise 对象，Promise.resolve 方法就起到这个作用 | Promise.resolve('foo')等价于 new Promise(resolve => resolve('foo')) |
| Promise.reject() | Promise.reject(reason)方法也会返回一个新的 Promise 实例，该实例的状态为 rejected。 |  |
| done() | Promise 对象的回调链，不管以 then 方法或 catch 方法结尾，要是最后一个方法抛出错误，都有可能无法捕捉到（因为 Promise 内部的错误不会冒泡到全局）。因此，我们可以提供一个 done 方法，总是处于回调链的尾端，保证抛出任何可能出现的错误。 | asyncFunc().then(f1).catch(r1).then(f2).done(); |
| finally() | finally 方法用于指定不管 Promise 对象最后状态如何，都会执行的操作。它与 done 方法的最大区别，它接受一个普通的回调函数作为参数，该函数不管怎样都必须执行。服务器使用 Promise 处理请求，然后使用 finally 方法关掉服务器。 | server.listen(0).then(function () {// run test}).finally(server.stop); |

## 36. 回调地狱的缺点？

fs.readdir(source, function (err, files) {  
 if (err) {  
 console.log("Error finding files: " + err);  
 } else {  
 files.forEach(function (filename, fileIndex) {  
 console.log(filename);  
 gm(source + filename).size(function (err, values) {  
 if (err) {  
 console.log("Error identifying file size: " + err);  
 } else {  
 console.log(filename + " : " + values);  
 aspect = values.width / values.height;  
 widths.forEach(  
 function (width, widthIndex) {  
 height = Math.round(width / aspect);  
 console.log(  
 "resizing " + filename + "to " + height + "x" + height  
 );  
 this.resize(width, height).write(  
 dest + "w" + width + "\_" + filename,  
 function (err) {  
 if (err) console.log("Error writing file: " + err);  
 }  
 );  
 }.bind(this)  
 );  
 }  
 });  
 });  
 }  
});

这个一堆以})结尾的金字塔，我们很亲切地称它为——“回调地狱”。

* 代码嵌套太多
* 代码太复杂

## 37. 一个列表中给每项添加点击事件，如何添加？当列表有一万项的时候怎么添加？（事件委托是什么）

假如有个列表，如下：

<ul id="contaniner">  
 <li id="li1">1</li>  
 <li id="li2">2</li>  
 <li id="li3">3</li>  
</ul>

可以单个的给每个表单项添加点击事件，如<li id="li1" onclick="alert(1)">1</li>,可以看[##14]题目中的答案。

但是当列表数据过多时，则不能采用这种形式来添加点击事件。

想象以上示例中，<li>标签的数量很大时，循环为每个子元素添加事件，绝非好方法。下面给出一种优雅的方法，采用事件委托。

document.getElementById("contaniner").addEventListener(  
 "click",  
 function (e) {  
 var target = e.target;  
 if (target.tagName == "LI") {  
 alert(target.innerText);  
 }  
 },  
 false  
);

这段代码里，使用事件委托只为<ul>元素添加一个 onclick 事件处理程序。因为有事件冒泡机制，单击每个<li>标签时，都会被这个函数处理。

React 中是怎么实现的？

React 并不会真正的绑定事件到每一个具体的元素上，而是采用事件代理的模式：在根节点 document 上为每种事件添加唯一的 Listener，然后通过事件的 target 找到真实的触发元素。这样从触发元素到顶层节点之间的所有节点如果有绑定这个事件，React 都会触发对应的事件处理函数。这就是所谓的 React 模拟事件系统。

尽管整个事件系统由 React 管理，但是其 API 和使用方法与原生事件一致。这种机制确保了跨浏览器的一致性：在所有浏览器（IE8 及以上）都可以使用符合 W3C 标准的 API，包括 stopPropagation()，preventDefault()等等。对于事件的冒泡（bubble）和捕获（capture）模式也都完全支持。

## 38. 原型链是什么？

### 原型是什么

JavaScript 中对象有一个特殊的[[Prototype]]内置属性，其实就是对其他对象的引用。

### 原型链是什么

如果要访问对象中并不存在的一个属性，[[Get]]操作就会查找对象内部的[[Prototype]]关联的对象。这个关联关系实际上定了一条“原型链”，在属性查找时会对它进行遍历。

## 39. 继承有几种方式？写一下

// "动物"对象的构造函数  
function Animal() {  
 this.species = "动物";  
}  
// "猫"对象的构造函数。  
function Cat(name, color) {  
 this.name = name;  
 this.color = color;  
}

怎样才能使"猫"继承"动物"呢？

* 构造函数绑定

第一种方法也是最简单的方法，使用 call 或 apply 方法，将父对象的构造函数绑定在子对象上，即在子对象构造函数中加一行：

function Cat(name, color) {  
 Animal.apply(this, arguments);  
 this.name = name;  
 this.color = color;  
}  
var cat1 = new Cat("大毛", "黄色");  
alert(cat1.species); // 动物

* prototype 模式

如果"猫"的 prototype 对象，指向一个 Animal 的实例，那么所有"猫"的实例，就能继承 Animal 了。

// 将Cat的prototype对象指向一个Animal的实例.它相当于完全删除了prototype 对象原先的值，然后赋予一个新值。  
Cat.prototype = new Animal();  
// 原来，任何一个prototype对象都有一个constructor属性，指向它的构造函数。  
// 如果没有"Cat.prototype = new Animal();"这一行，Cat.prototype.constructor是指向Cat的；  
// 加了这一行以后，Cat.prototype.constructor指向Animal。  
Cat.prototype.constructor = Cat; //？是否需要？  
var cat1 = new Cat("大毛", "黄色");  
alert(cat1.species); // 动物

## 40. promise 能够一直.then 下去的原因？

Promise 决议后的值仍然是 promise 对象。

如何将一个立即值封装为 promise 对象? var p1 = Promise.resolve(42);

## 41. 有一个场景，我们有很多以前的代码，都是回调函数的形式写的。如何将 callback 形式的回调函数转化为 promise 的调用方式？

## 42. let const var 的区别？

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 声明方式 | 变量提升 | 作用域 | 初始值 | 重复定义 |
| var | 是 | 函数级 | 不必须 | 允许 |
| let | 否 | 块级 | 不必须 | 不允许 |
| const | 否 | 块级 | 必须 | 复合类型的变量可以，其他并不可以 |

const对于复合类型的变量，变量名不指向数据，而是指向数据所在的地址。

Q：为什么 const 一旦声明常量，就必须立即初始化，不能留到以后再赋值？

Q： const obj = {a: 1};那么 a 还能赋值为其他值吗？为什么？

A: obj.a = 2;这是完全正确的。因为const对于复合类型的变量，变量名不指向数据，而是指向数据所在的地址。 常量 obj 存储的是一个地址，指向一个对象。不可变的只是这个地址，即不能把 obj 指向另一个地址，但对象本身是可变的，所以依然可以为其添加新属性。

Q: 有什么办法可以让const obj = {a: 1};的值不可变？

A: 如果真的想讲对象冻结，应该使用 Object.freeze 方法。 const foo = Object.freeze({}); foo.prop= 123; // 不起作用

## 43. 写一个函数，实现对象的深度拷贝。

### 浅拷贝

浅拷贝只复制指向某个对象的指针，而不复制对象本身，新旧对象还是共享同一块内存。

shallow-copy

shallow-copy

* for 循环方式

function shallowCopy(p, c) {  
 var c = c || {};  
 for (var i = 0; i < p.length; i++) {  
 c[i] = p[i];  
 }  
 return c;  
}

* es6

Object.assign(); //也能实现对象的浅拷贝

### 深拷贝

这类题目的发问方式又很多，除了“模拟 JAVA 中的克隆接口”、“JavaScript 实现原型模式”以外，它更常见、更友好的发问形式是“请实现 JS 中的深拷贝”。

深拷贝会另外创造一个一模一样的对象，新对象跟原对象不共享内存，修改新对象不会改到原对象。

deep-copy

deep-copy

* for 循环方式

function deepCopy(p, c) {  
 var c = c || {};  
  
 for (var i in p) {  
 // 这会有问题，如果是数组的话，遍历会有问题。所以要确定这个对象不是数组，数组的拷贝有很多种方式。  
 // 另外还可能出现相互引用的对象导致死循环的情况  
  
 var prop = p[i]; // p.a = p的情况  
 if (prop === c) {  
 continue;  
 }  
 if (typeof p[i] === "object") {  
 c[i] = p[i].constructor == Array ? [] : {};  
 deepCopy(p[i], c[i]);  
 } else {  
 c[i] = p[i];  
 }  
 }  
 return c;  
}

* 对于对象是 json 安全的，可以使用 json 序列化的方式

// 对于对象是json安全的，可以使用  
var newObj = JSON.parse(JSON.stringify(someobj));

* 很热门的函数库 lodash，也有提供\_.cloneDeep 用来做 Deep Copy

实现数组的拷贝 const a1=[1,2];

1. const a2=a1.concat();
2. const a3 = a1.slice();

## 44. 给一个字符串，找到里面重复最多的字符？

function findMax(str) {  
 var map = {},  
 max = { num: 0 };  
  
 for (var i in str) {  
 if (map[str[i]]) {  
 map[str[i]]++;  
 } else {  
 map[str[i]] = 1;  
 }  
  
 if (map[str[i]] > max.num) {  
 max.num = map[str[i]];  
 max.key = str[i];  
 }  
 }  
 console.log(`max num is ${max.num}, and the key is ${max.key}`);  
 // 打印出来所有值的重复次数  
 for (var key in map) {  
 console.log(`${key} copied is ${map[key]}`);  
 }  
}

## 45. 一个列表很长，如何自己实现一个滚动条？

* css 方式

<div style="width:260px;height:120px; overflow:scroll; border:1px solid;">  
 这里是你要显示的内容,当内容很多时，会出现滚动条。这个属性定义溢出元素内容区的内容会如何处理。如果值为  
 scroll，不论是否需要，用户代理都会提供一种滚动机制。因此，有可能即使元素框中可以放下所有内容也会出现滚动条。  
</div>

* js 方式

通过鼠标的按下事件（onmousedown）和 JS 滚轮事件(mousewheel-非火狐/DOMMouseScroll-火狐浏览器)来改变 top 即可。

下面是一个具体的实现:

<!DOCTYPE html>  
<html>  
 <head>  
 <meta charset="utf-8" />  
 <title>标题</title>  
 <meta name="keywords" content="" />  
 <meta name="description" content="" />  
 <style>  
 \* {  
 margin: 0;  
 padding: 0;  
 list-style: none;  
 }  
  
 body {  
 height: 2000px;  
 }  
  
 .box1 {  
 width: 320px;  
 height: 400px;  
 background: #ccc;  
 overflow: hidden;  
 overflow-y: scroll;  
 margin: 20px 0 0 100px;  
 }  
  
 .con1 {  
 font-size: 18px;  
 }  
  
 #out {  
 width: 320px;  
 height: 400px;  
 background: url(bg1.png) 0 0 repeat-x;  
 position: absolute;  
 left: 500px;  
 top: 20px;  
 overflow: hidden;  
 }  
  
 #con {  
 width: 280px;  
 padding: 5px;  
 font-size: 18px;  
 position: absolute;  
 left: 0px;  
 top: 0px;  
 }  
  
 #box {  
 width: 30px;  
 height: 400px;  
 position: absolute;  
 right: 0;  
 top: 0;  
 }  
  
 #drag {  
 width: 30px;  
 height: 53px;  
 /\* background: url(icon2.png) 0 0 no-repeat; \*/  
 background-color: slategray;  
 position: absolute;  
 left: 0px;  
 top: 0px;  
 }  
 </style>  
 </head>  
  
 <body>  
 <div class="box1">  
 <p class="con1">  
 据新华社电中航工业、国机集团等8家央企6日在京签约，在重要项目、科技转化等相关领域开展重组合作整合，这将成为央企间产业联合协作的新典范，也将开启央企间产业重组合作整合的新阶段。  
 相关合作内容包括：中航工业和国机集团共同发展八万吨模锻压机项目；中航工业将房地产业务全部划转保利集团，中核建设和中国一重在高温气冷堆主设备制造领域合作，兵器工业、兵器装备、中国国新对北方公司进行股权重组。  
 国务院国资委主任肖亚庆在6日参加中央企业产业合作座谈会上表示，下一步，国资委政策将加码推动央企产业重组步伐，通过业务整合、资产重组、股权合作、资产置换、无偿划转、协议转让、战略联盟、联合开发等多途径，加快央企间产业重组合作整合。  
 肖亚庆同时清晰地勾勒出国企改革兼并重组“路线图”：通过产业重组，在突破关键技术、掌握核心资源，打造知名品牌等方面，实现产业重组合作“一加一大于二”的效果。肖亚庆同时透露，目前部分央企在牵头技术创新战略联盟、设立创新投资基金、构建创新孵化平台等方面，取得了重大突破。据统计，央企牵头国家及地方技术创新联盟141个，50多家中央企业共发起和参与基金179只，构建面向社会的创新孵化平台57个，创业创新平台27个。  
 “此次集中签约，意味着央企重组的重心开始向资本、项目、产业板块等内部要素转移。”中国企业研究院首席研究员李锦说，未来，央企间产业重组合作整合将被更快地推进，央企内部的各种要素将被再次优化。  
 </p>  
 </div>  
  
 <div id="out">  
 <div id="con">  
 据新华社电中航工业、国机集团等8家央企6日在京签约，在重要项目、科技转化等相关领域开展重组合作整合，这将成为央企间产业联合协作的新典范，也将开启央企间产业重组合作整合的新阶段。  
 相关合作内容包括：中航工业和国机集团共同发展八万吨模锻压机项目；中航工业将房地产业务全部划转保利集团，中核建设和中国一重在高温气冷堆主设备制造领域合作，兵器工业、兵器装备、中国国新对北方公司进行股权重组。  
 国务院国资委主任肖亚庆在6日参加中央企业产业合作座谈会上表示，下一步，国资委政策将加码推动央企产业重组步伐，通过业务整合、资产重组、股权合作、资产置换、无偿划转、协议转让、战略联盟、联合开发等多途径，加快央企间产业重组合作整合。  
 肖亚庆同时清晰地勾勒出国企改革兼并重组“路线图”：通过产业重组，在突破关键技术、掌握核心资源，打造知名品牌等方面，实现产业重组合作“一加一大于二”的效果。肖亚庆同时透露，目前部分央企在牵头技术创新战略联盟、设立创新投资基金、构建创新孵化平台等方面，取得了重大突破。据统计，央企牵头国家及地方技术创新联盟141个，50多家中央企业共发起和参与基金179只，构建面向社会的创新孵化平台57个，创业创新平台27个。  
 “此次集中签约，意味着央企重组的重心开始向资本、项目、产业板块等内部要素转移。”中国企业研究院首席研究员李锦说，未来，央企间产业重组合作整合将被更快地推进，央企内部的各种要素将被再次优化。  
 </div>  
 <div id="box">  
 <p id="drag"></p>  
 </div>  
 </div>  
 <script>  
 var out = document.getElementById("out");  
 var con = document.getElementById("con");  
 var box = document.getElementById("box");  
 var drag = document.getElementById("drag");  
 drag.onmousedown = function (ev) {  
 var e = ev || window.event;  
 if (e.preventDefault) {  
 e.preventDefault();  
 } else {  
 e.returnValue = false;  
 }  
 var d\_bkt = e.clientY - drag.offsetTop;  
 document.onmousemove = function (ev) {  
 var e = ev || window.event;  
 var top = e.clientY - d\_bkt;  
 if (top <= 0) {  
 top = 0;  
 }  
 if (top >= box.clientHeight - drag.clientHeight) {  
 top = box.clientHeight - drag.clientHeight;  
 }  
 var scale = top / (box.clientHeight - drag.clientHeight);  
 var cony = scale \* (con.clientHeight - out.clientHeight);  
 drag.style.top = top + "px";  
 con.style.top = -cony + "px";  
 console.log(top);  
 };  
 document.onmouseup = function () {  
 document.onmousemove = null;  
 };  
 };  
 var str = window.navigator.userAgent.toLowerCase();  
 if (str.indexOf("firefox") != -1) {  
 //火狐浏览器  
 out.addEventListener(  
 "DOMMouseScroll",  
 function (e) {  
 e.preventDefault(); //阻止窗口默认的滚动事件  
 if (e.detail < 0) {  
 var t = con.offsetTop + 20;  
 if (t >= 0) {  
 t = 0;  
 }  
 if (t <= -(con.clientHeight - out.clientHeight)) {  
 t = -(con.clientHeight - out.clientHeight);  
 }  
 var scale = t / (con.clientHeight - out.clientHeight);  
 var top = scale \* (box.clientHeight - drag.clientHeight);  
 con.style.top = t + "px";  
 drag.style.top = -top + "px";  
 }  
 if (e.detail > 0) {  
 var t = con.offsetTop - 20;  
 if (t >= 0) {  
 t = 0;  
 }  
 if (t <= -(con.clientHeight - out.clientHeight)) {  
 t = -(con.clientHeight - out.clientHeight);  
 }  
 var scale = t / (con.clientHeight - out.clientHeight);  
 var top = scale \* (box.clientHeight - drag.clientHeight);  
 con.style.top = t + "px";  
 drag.style.top = -top + "px";  
 }  
 },  
 false  
 );  
 } else {  
 //非火狐浏览器  
 out.onmousewheel = function (ev) {  
 var e = ev || window.event;  
 if (e.preventDefault) {  
 e.preventDefault();  
 } else {  
 e.returnValue = false;  
 }  
 if (e.wheelDelta > 0) {  
 var t = con.offsetTop + 20;  
 if (t >= 0) {  
 t = 0;  
 }  
 if (t <= -(con.clientHeight - out.clientHeight)) {  
 t = -(con.clientHeight - out.clientHeight);  
 }  
 var scale = t / (con.clientHeight - out.clientHeight);  
 var top = scale \* (box.clientHeight - drag.clientHeight);  
 con.style.top = t + "px";  
 drag.style.top = -top + "px";  
 }  
 if (e.wheelDelta < 0) {  
 var t = con.offsetTop - 20;  
 if (t >= 0) {  
 t = 0;  
 }  
 if (t <= -(con.clientHeight - out.clientHeight)) {  
 t = -(con.clientHeight - out.clientHeight);  
 }  
 var scale = t / (con.clientHeight - out.clientHeight);  
 var top = scale \* (box.clientHeight - drag.clientHeight);  
 con.style.top = t + "px";  
 drag.style.top = -top + "px";  
 }  
 };  
 }  
 </script>  
 </body>  
</html>

## 46. 简单点的，一个div方块如何移动？css3 方式，js 方式呢？用 js 如何实现 1s 动一下的效果?

* CSS3 的动画效果:animation

animation-name 规定需要绑定到选择器的 keyframe 名称 animation-duration 规定完成动画所花费的时间，以秒或毫秒计 animation-timing-function 规定动画的速度曲线 animation-delay 规定在动画开始之前的延迟 animation-iteration-count 规定动画应该播放的次数 animation-direction 规定是否应该轮流反向播放动画。

* js 方式实现

<html>  
 <head>  
 <style>  
 .container {  
 border: 1px solid red;  
 width: 100px;  
 height: 100px;  
 position: relative;  
 }  
 </style>  
 <script>  
 function init() {  
 var div = document.getElementsByClassName("container")[0];  
 for (var i = 0; i < 11; i++) {  
 (function (j) {  
 setTimeout(function () {  
 div.style.top = j \* 10 + "px";  
 div.style.left = j \* 10 + "px";  
 }, j \* 1000);  
 })(i);  
 }  
 }  
 </script>  
 </head>  
 <body>  
 <div class="container">  
 一个可以移动的方框  
 </div>  
 </body>  
</html>

## 47. 页面上一个可以触摸的方块，如何让其跟着这手指的移动而移动？手机触屏事件了解多少？

### 手机触屏事件

|  |  |
| --- | --- |
| 事件 | 动作含义 |
| touchstart | 当手指触摸屏幕时候触发，即使已经有一个手指放在屏幕上也会触发。 |
| touchmove | 当手指在屏幕上滑动的时候连续地触发。在这个事件发生期间，调用 preventDefault()事件可以阻止滚动。 |
| touchend | 当手指从屏幕上离开的时候触发 |
| touchcancel | 当系统停止跟踪触摸的时候触发 |

### 如何跟随手指移动？

利用这些事件，event.touches[0].clientX，有触摸点的位置，就可以进行位置的移动。

## 48. 定时有几种方式？

* setTimeout

设定一个定时器，在指定时间后执行一段代码。 > let timeoutId = setTimeout(func,delay,param1,param2,……);

* timeoutId：为此定时器的 id，此 id 可以传入 clearTimeout（）来取消这次定时器
* func：为指定时间后执行的函数
* delay：是延迟的毫秒数。如果省略的话，默认取值 0。
* param：向延迟函数传递而外的参数，IE9 以上支持
* setInterval

以给定的时间间隔重复执行一个函数。var intervalID = window.setInterval(func, delay[, param1, param2, ...]); 其参数和 setTimeout 相同

* setImmediate > 该方法用来把一些需要长时间运行的操作放在一个回调函数里,在浏览器完成后面的其他语句后,就立刻执行这个回调函数。类似 setTimeout(func, 0)

## 49. 如何用原生来实现 promise.all()?

function PromiseM() {  
 this.status = "pending";  
 this.msg = "";  
 var that = this;  
 var process = arguments[0];  
 process(  
 function () {  
 that.status = "resolve";  
 that.msg = arguments[0];  
 },  
 function () {  
 that.status = "reject";  
 that.msg = arguments[0];  
 }  
 );  
 return this;  
}  
PromiseM.prototype.then = function () {  
 if (this.status == "resolve") {  
 arguments[0](this.msg);  
 }  
 if (this.status == "reject" && arguments[1]) {  
 arguments[1](this.msg);  
 }  
};  
  
// 测试用例  
  
var p = new PromiseM(function (resolve, reject) {  
 resolve("123");  
});  
  
p.then(  
 function (success) {  
 console.log(success);  
 console.log("success");  
 },  
 function () {  
 console.log("fail！");  
 }  
);

## 50. 如何实现一下 Object.create()?

Object.create(parent)主要完成了三件事情：

* 创建一个对象
* 继承指定父对象
* 为新对象扩展新属性

何时使用 create: 希望在创建对象时就提前指定继承的父对象，并同时扩展新属性时。

Object.mycreate = function(parent,props) {  
 var obj = new Object();  
 Object.setPrototypeOf(obj,parent);  
 Object.defineProperties(obj,props);  
 return obj;  
}  
  
// 用法  
  
var father={bal:1000,car:"=b="}  
var hmm=Object.mycreate(father,{//create调用的也是defineproperty  
 phone:{//默认为false  
 value:"肾6s",  
 writable:true,  
 enumertable:true,  
 configurable:true.  
 },  
});

## 51. js 的路由是如何实现的？

location 是 javascript 里边管理地址栏的内置对象，比如 location.href 就管理页面的 url，用 location.href=url 就可以直接将页面重定向 url。而 location.hash 则可以用来获取或设置页面的标签值。

# 后的字符

* 在第一个#后面出现的任何字符，都会被浏览器解读为位置标识符。这意味着，这些字符都不会被发送到服务器端。
* window.hash hash 属性是一个可读可写的字符串，该字符串是 URL 的锚部分（从 # 号开始的部分）。

## 52. 对 websocket 的理解？

WebSocket 协议在 2008 年诞生，2011 年成为国际标准。所有浏览器都已经支持了。

WebSocket 是一个持久化的协议，是基于 http 协议来完成一部分握手。

Websocket 握手中多了: Upgrade: websocket Connection: Upgrade 这就是 WebSocket 的核心，告诉服务器是一个 WebSocket 协议。

ajax 轮询和 long poll 都是不断的建立 http 连接，等待服务端处理，可以体现 http 协议的一个特点：被动性（服务器端不能主动联系客户端，只能客户端发起）。

* ajax 轮询

ajax轮询的原理特别简单。让浏览器隔个几秒就发送一次请求，询问服务器是否有新信息。

* long poll

long poll 其实原理跟 ajax轮询 差不多，都是采用轮询的方式，不过采取的是阻塞模型（一直打电话，没收到就不挂电话），也就是说，客户端发起连接后，如果没消息，就一直不返回Response给客户端。直到有消息才返回，返回完之后，客户端再次建立连接，周而复始。

### 特点

* 服务器可以主动向客户端推送信息，客户端也可以主动向服务器发送信息，是真正的双向平等对话，属于服务器推送技术的一种。
* 建立在 TCP 协议之上，服务器端的实现比较容易。
* 与 HTTP 协议有着良好的兼容性。默认端口也是 80 和 443，并且握手阶段采用 HTTP 协议，因此握手时不容易屏蔽，能通过各种 HTTP 代理服务器。
* 数据格式比较轻量，性能开销小，通信高效。
* 可以发送文本，也可以发送二进制数据。
* 没有同源限制，客户端可以与任意服务器通信。
* 协议标识符是 ws（如果加密，则为 wss），服务器网址就是 URL。

ws和wss

ws和wss

### 使用

主要是的是webSocket.onmessage属性，用于指定接收服务器数据后的回调函数。webSocket.send()方法，可用于向服务器发送数据。

### 参考

* [websocket 原理](https://www.cnblogs.com/fuqiang88/p/5956363.html)

## 53. 对 Pure functions（纯函数）的理解？

纯函数是满足如下条件的函数：

* 相同输入总是会返回相同的输出。
* 不产生副作用。
* 不依赖于外部状态。

## 54. js 单线程的理解？

JS 的单线程是指一个浏览器进程中只有一个 JS 的执行线程，同一时刻内只会有一段代码在执行。

作为浏览器脚本语言，JavaScript 的主要用途是与用户互动，以及操作 DOM。这决定了它只能是单线程，否则会带来很复杂的同步问题。

## 55. 如何用 promise 和 setTimeout 实现一个 delay 函数，.then 里面可以正常完成 promise 的一些后续操作？

// delay函数是这样的  
delay(1000).then();

当时没有想好，但是知道肯定要返回一个 promise，不然没有办法.then下去。 其实也是主要没有理解好 promise。

实现方法如下：

function delay(delayTime) {  
 return new Promise(function (resolve, reject) {  
 setTimeout(resolve, delayTime);  
 });  
}  
delay(1000).then(function () {  
 console.log("0：执行成功！");  
});  
  
delay(2000)  
 .then(function () {  
 console.log("1：执行成功！");  
 return delay(1000);  
 })  
 .then(function () {  
 console.log("2：执行失败！");  
 });

## 56. array、null、object 数据如何判断？

typeof 方法判断array、null、object类型的数据，都会返回object。

设变量为i ,那么i== null为 true,则 i 为 null; i.constructor.name 如果是Array则为数组，如果是Object则是简单的 object 类型数据。

还可以b.constructor == Array来进行是否是数组的判断； 还可以b.constructor == Object来进行是否是对象的判断；

## 58. 写出下面的值？

var a = {};  
var b = {};  
var c = {a: 1};  
var d[a] = 1;  
d[b] = 1;  
d[c] // 1

原因： **在对象中属性名永远都是字符串。**如果使用 string 以外的其他值作为属性名，那么它首先会被转化为一个字符串。

var obj = {};  
obj[true] = "foo";  
obj[3] = "bar";  
  
//等价于  
obj["true"] = "foo";  
obj["3"] = "bar";  
obj["[object Object]"] = "baz";

obj.toString() // "[object Object]",所有对象的toString()的值都是"[object Object]"

备注： .a - 属性访问值 ["a"] - 键访问 二者是等价的。

## 58. es6 中的 Map 结构

### 1. es6 为什么推出 Map 结构？

JavaScript 的对象（object）本质上就是键值对的集合（Hash 结构），但是只能用字符串作为键。这就给它的使用带来了很大的限制。

## 59. 说一说 es5 中 object 是如何存储的？

var name = "hehe";  
var age = 27;  
var job;  
var arr = [1, 2, 3, 4];  
var obj = { name: "hehe", age: 27 };

这些都是如何存储的？

值的存储

值的存储

## 60. 判断是否是回文

如果给定的字符串是回文，返回 true，反之，返回 false。 如果一个字符串忽略标点符号、大小写和空格，正着读和反着读一模一样，那么这个字符串就是 palindrome(回文)。

注意你需要去掉字符串多余的标点符号和空格，然后把字符串转化成小写来验证此字符串是否为回文。

function palindrome(str) {  
 // Good luck!  
 str = str.toLowerCase();  
 var array = str.split("");  
  
 var array1 = array.filter(function(ch) {  
 return 'A'<=ch && ch <= 'z' || 1<=ch && ch <=9;  
 });  
  
 var re = true;  
 for(var i=0, len = array1.length; i<=Math.floor(len/2); i++){  
 if(array1[i] != array1[len-i-1]){  
 re = false;  
 }  
 }  
 return re;  
}  
  
palindrome("eye");

## 62. JavaScript 如何判断一个对象是空的？

* 第一种： const isEmptyObject = obj => Object.getOwnPropertyNames(obj).length === 0
* 第二种： Object.keys(obj).length == 0

## 63. js 的作用域是什么？作用域是什么时候确定的？

## 64. setTimeout 和 promise 的区别？宏任务和微任务是什么？有什么区别？

宏任务队列可以有多个，微任务队列只有一个。

宏任务：script（全局任务）, setTimeout, setInterval, setImmediate, I/O, UI rendering. 微任务：process.nextTick, Promise, Object.observer, MutationObserver.

取一个宏任务来执行。执行完毕后，下一步。 取一个微任务来执行，执行完毕后，再取一个微任务来执行。直到微任务队列为空，执行下一步。 更新 UI 渲染。

写出下面的执行结果：

console.log("1");  
  
setTimeout(function () {  
 console.log("2");  
 new Promise(function (resolve) {  
 console.log("3");  
 resolve();  
 }).then(function () {  
 console.log("4");  
 });  
});  
  
new Promise(function (resolve) {  
 console.log("5");  
 resolve();  
}).then(function () {  
 console.log("6");  
});  
  
setTimeout(function () {  
 console.log("7");  
  
 new Promise(function (resolve) {  
 console.log("8");  
 resolve();  
 }).then(function () {  
 console.log("9");  
 });  
});

结果是： 1 5 6 2 3 4 7 8 9

## 65. 构造函数是什么？new 的时候都去做了什么

## 68. js 的顶级对象是什么

## 69. instanceof 是做什么的? Array instanceof Object 是什么？Array 是谁 new 出来的

instanceof运算符是用来测试一个对象在其原型链中是否存在一个构造函数的prototype属性。 语法：

// object: 要检测的对象 // constructor: 某个构造函数 object instanceod constructor

var a=new Array();  
alert(a instanceof Array); // true，  
同时 alert(a instanceof Object) //也会返回 true;  
这是因为 Array 是 object 的子类。

另外，更重的一点是 instanceof 可以在继承关系中用来判断一个实例是否属于它的父类型。

function Foo() {}  
Foo.prototype = new Aoo(); //JavaScript 原型继承  
var foo = new Foo();  
console.log(foo instanceof Foo); //true  
console.log(foo instanceof Aoo); //true

Q: function 的 arguments instanceof Array // false

## 70. 高阶函数是什么

把函数作为参数传入，这样的函数称为高阶函数，函数式编程就是指这种高度抽象的编程范式。

es6 的迭代器、filter、forEach、map 等都是高阶函数。