**目录**

[1 JS基本数据类型 3](#_Toc91005864)

[1.1 JS的数据类型有哪些 3](#_Toc91005865)

[1.2判断数据类型的方法有哪些 4](#_Toc91005866)

[1.3 null和undefined的区别 4](#_Toc91005867)

[1.4 为什么typeof null 返回object 5](#_Toc91005868)

[1.5 为什么0.1+0.2 == 0.3 是false 5](#_Toc91005869)

[1.6 typeof NaN 的结果是什么 5](#_Toc91005870)

[1.7 isNaN 和 Number.isNaN 函数的区别 5](#_Toc91005871)

[1.8 == 操作符的强制类型转换规则 6](#_Toc91005872)

[1.9 其他值到字符串的转换规则 6](#_Toc91005873)

[1.10 其他值到数字值的转换规则 6](#_Toc91005874)

[1.11 || 和 && 操作符的返回值 6](#_Toc91005875)

[1.12 Object.is() 与比较操作符 `===`、`==`的区别 7](#_Toc91005876)

[1.13 什么是 JavaScript 中的包装类型 7](#_Toc91005877)

[1.14 JavaScript 中如何进行隐式类型转换 8](#_Toc91005878)

[1.15 JS基本和非基本数据类型之间的区别 9](#_Toc91005879)

[1.16 数组有哪些原生方法 9](#_Toc91005880)

[2 JS变量提升 9](#_Toc91005881)

[2.1 深拷贝和浅拷贝 9](#_Toc91005882)

[2.2 JS变量提升 9](#_Toc91005883)

[2.3 let、const、var的区别 10](#_Toc91005884)

[2.4 JavaScript为什么要进行变量提升，它导致了什么问题 10](#_Toc91005885)

[2.5 const对象的属性可以修改吗 11](#_Toc91005886)

[3 JS对象 11](#_Toc91005887)

[3.1 new操作符的过程 11](#_Toc91005888)

[3.2 对象创建的方式有哪些 12](#_Toc91005889)

[3.3 对象继承的方式有哪些 12](#_Toc91005890)

[3.4 箭头函数与普通函数的区别 13](#_Toc91005891)

[3.5 扩展运算符的作用及使用场景 13](#_Toc91005892)

[3.6 Proxy 可以实现什么功能 14](#_Toc91005893)

[3.7 Map和Object的区别 （Map是键值对） 14](#_Toc91005894)

[3.8 JavaScript有哪些内置对象 15](#_Toc91005895)

[3.9 JSON的理解 15](#_Toc91005896)

[3.10 什么是 DOM 和 BOM 16](#_Toc91005897)

[3.11 常用的DOM操作方法 16](#_Toc91005898)

[3.12 use strict 17](#_Toc91005899)

[3.13 如何判断一个对象是否属于某个类 18](#_Toc91005900)

[4 JS的异步 18](#_Toc91005901)

[4.1 延迟加载的方式有哪些 18](#_Toc91005902)

[4.2 对AJAX的理解，实现一个AJAX请求 18](#_Toc91005903)

[4.3异步编程的实现方式 19](#_Toc91005904)

[4.4 Promise 19](#_Toc91005905)

[4.5 promise链式回调 22](#_Toc91005906)

[4.6 promise的reject 22](#_Toc91005907)

[4.7 promise的catch用法 23](#_Toc91005908)

[4.8 promise的all 24](#_Toc91005909)

[4.9 promise的race 24](#_Toc91005910)

[5 函数的调用和作用域 24](#_Toc91005911)

[5.1 console.log 和console.dir的区别 24](#_Toc91005912)

[5.2 尾调用 24](#_Toc91005913)

[5.3 js 执行机制、事件循环 25](#_Toc91005914)

[5.4 宏任务、微任务 26](#_Toc91005915)

[5.5 for…in和for…of的区别 26](#_Toc91005916)

[5.6 遍历数组的方法 27](#_Toc91005917)

[5.7 遍历map键值对方法 28](#_Toc91005918)

[5.8 遍历的forEach和map方法有什么区别 28](#_Toc91005919)

[5.9 原型、原型链的理解 28](#_Toc91005920)

[5.10 闭包的理解 28](#_Toc91005921)

[5.11 作用域 30](#_Toc91005922)

[5.12 作用域链 30](#_Toc91005923)

[5.13 对this对象的理解 31](#_Toc91005924)

[5.14 call() 和 apply() 的区别 31](#_Toc91005925)

[5.15 javascript 的垃圾回收机制 31](#_Toc91005926)

[5.16 内存泄漏 32](#_Toc91005927)

[6 实战篇 33](#_Toc91005928)

[6.1 字节JS实战篇 33](#_Toc91005929)

[6.1.1怎么判断一个对象是不是可迭代的 33](#_Toc91005930)

[6.1.2 for of 和 for in的区别 34](#_Toc91005931)

[6.1.3 如何修改一个数组的副本而不更改原数组 34](#_Toc91005932)

[6.1.4 说说宏任务和微任务的执行顺序 35](#_Toc91005933)

[6.1.5 两个定时器 上面的定时器先执行 在执行下面的定时器吗？ 38](#_Toc91005934)

[6.1.6 设置两个一样的定时器事件，会输出什么结果 39](#_Toc91005935)

[6.1.7 JS的== 和=== 39](#_Toc91005936)

[6.1.8 JS语法糖 39](#_Toc91005937)

[6.1.9 async和await 41](#_Toc91005938)

[6.1.10 JS为什么要有异步操作 43](#_Toc91005939)

[6.1.11 JS闭包 43](#_Toc91005940)

[6.1.12 this指针深入理解 45](#_Toc91005941)

[6.1.13 JS数据类型及其判断方法 46](#_Toc91005942)

[6.1.14 call、apply、bind的区别 46](#_Toc91005943)

[6.1.15 判断一个数据是不是数组的方法 47](#_Toc91005944)

[6.1.16 数组去重 47](#_Toc91005945)

[6.1.17 console.log和console.dir的区别 47](#_Toc91005946)

[6.1.18 JS的原型和原型链 47](#_Toc91005947)

[6.1.19 ==的判断比较 非常经典 49](#_Toc91005948)

[6.2 腾讯 50](#_Toc91005949)

[6.2.1 手写new操作符 50](#_Toc91005950)

[6.2.2 ES6新特性整理 50](#_Toc91005951)

# 1 JS基本数据类型

## 1.1 JS的数据类型有哪些

JavaScript共有八种数据类型，分别是 Undefined、Null、Boolean、Number、String、Object、Symbol、BigInt。

·Symbol 代表创建独一无二且不可变的数据类型，为了解决变量冲突的问题

·BigInt 可以表示任意精度格式的整数

OUNNSSBB

## 1.2判断数据类型的方法有哪些

**·typeof，**其中数组、对象、null都会被判断为object，其他判断都正确

**·**instanceof**只能正确判断引用数据类型**，而不能判断基本数据类型。instanceof 运算符用来判断一个对象的原型链中是否存在某个构造函数的 prototype 属性

**·constructor**



实例对象上调用.constructor，由于实例对象上并不存在.constructor，因此会沿着原型链去原型对象上查找.constructor，而原型对象上的.constructor属性指向构造函数，因此instance.constructor也指向构造函数，可以用来判断实例对象的类型。

## 1.3 null和undefined的区别

Undefined 和 Null 都是基本数据类型

·undefined 代表的含义是**未定义**，null 代表的含义是**空对象。**

·一般变量声明了但还没有定义的时候会返回 undefined，null主要用于赋值给一些可能会返回对象的变量，作为初始化。

·undefined可以看做是一种不严谨的、不合规范的、预料之外的错误，而null则是在意料之中，特意安排的正常现象。

·undefined 可以做变量名，null不可以

·typeof null 为object，type of undefined 为undefined

·null==undefined 为true null===undefined 为false

## 1.4 为什么typeof null 返回object

javascript 中变量的二进制前三位代表数据类型，object类型为000，而null的二进制表示全都是0，自然前三位也是0，所以执行typeof时会返回 ‘object’。

null用来给将来可能指向对象的变量赋值，本身有空对象的意思，因此返回object也在情理之中

## 1.5 为什么0.1+0.2 == 0.3 是false

js遵循IEEE 754标准，使用64位固定长度来表示，也就是标准的double双精度浮点数。IEEE-754所定义的双精度浮点数格式共64位。其中包含1位符号位S，11位指数位E 和52位尾数位M 。

根据这个原则，0.1和0.2的二进制数相加，再转化为十进制数就是：0.30000000000000004。

想要实现0.1+0.2=0.3 可以使用Number.EPSILON属性，设置一个误差范围

Number.EPSILON表示 1 与大于 1 的最小浮点数之间的差。

## 1.6 NaN

NaN 指“不是一个数字”（not a number），是执行数学失败后返回的结果

typeof NaN; // "number"

NaN 是一个特殊值，它和自身不相等，即NaN==NaN返回false

## 1.7 isNaN 和 Number.isNaN 函数的区别

·函数 isNaN通过Number()方法尝试将参数转换成Number类型

如果成功返回false，如果失败返回true。会进行数据类型的转换

·函数 Number.isNaN 不会进行数据类型的转换

Number.isNaN 会首先判断传入参数是否为数字，不是数字的话直接返回false。如果是数字再继续判断是否为 NaN 。常用于检测两个数字的运算结果。

## 1.8 == 操作符的强制类型转换规则

对于 = = 来说，如果对比双方的类型不一样，就会进行类型转换。假如对比 x 和 y 是否相同，就会进行如下判断流程。

1.null 仅与 undefined 互等

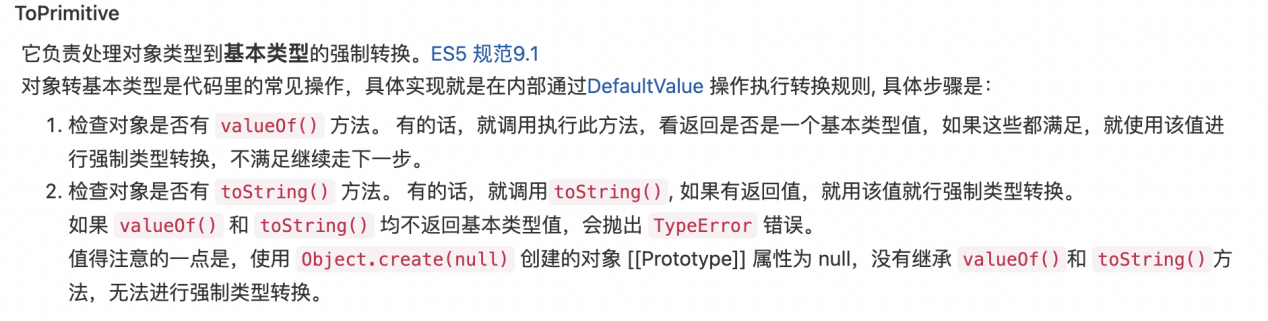
2.类型一致时比较二者的值（无论原始值或引用值），值相同则相等（例外: NaN）

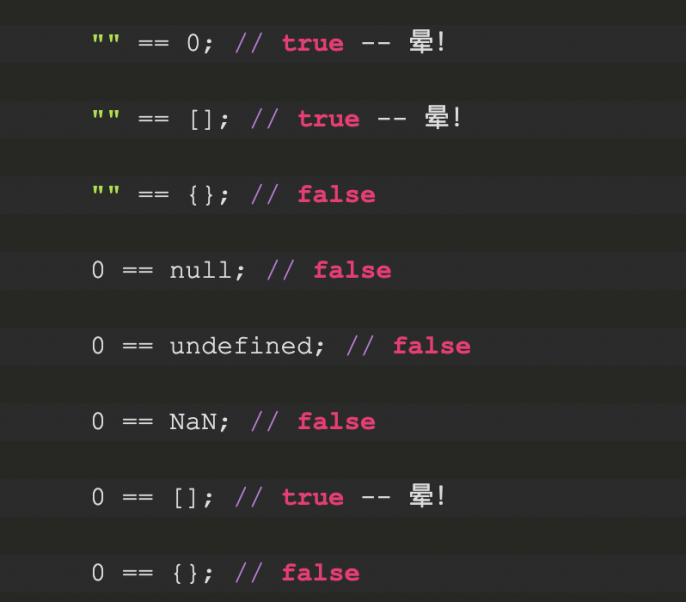
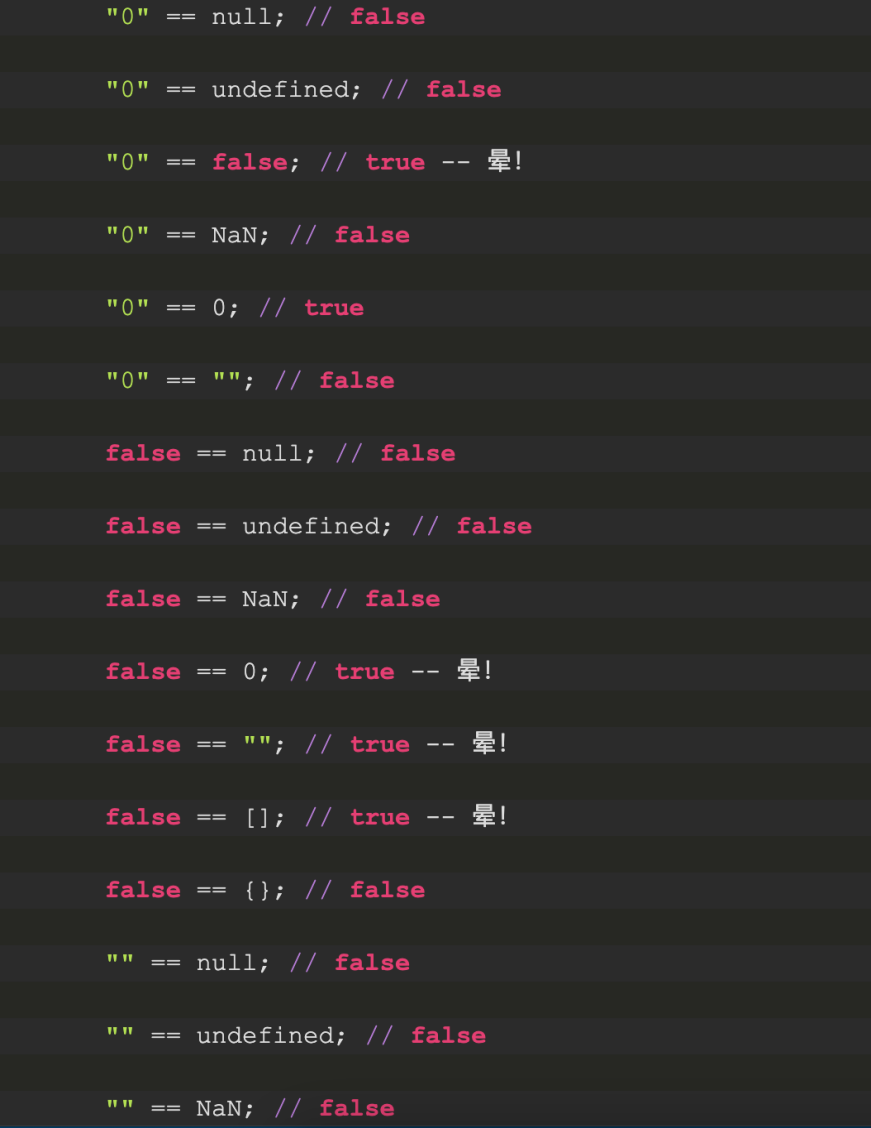
3.对象和基本类型值，会将对象转为基本数据类型

4.string和number比较，将string转为number

5.boolean和其他类型，会先将boolean转为number，然后比较

对象转为基本数据类型规则：先调用valueof，再调用toString。其中，valueof返回对象自身，非基本类型值。tostring返回"[object Object]"，为字符串类型。

对象的toprimitive方法：



NaN和任何类型进行==比较，都会返回false。

[] == ![] // true

首先，右边的!会将[]转为boolean，由于[]转为boolean为true（对象都为true），因此![]为false，而boolean在==两边，会自动转为number，因此，false转为0。至此，==右边为0，==左边的[]也会转换为数字，[]转换的数字为0，因此，0==0，故返回true

## 其他值到字符串的转换规则

基本数据类型

·null 转换为 “null”，undefined 转换为 “undefined”

·Boolean 类型，true 转换为 “true”，false 转换为 “false”

·Number 类型的值直接转换，不过那些极小和极大的数字会使用指数形式

·Symbol 类型的值直接转换(即“Symbol(“123”)”)

引用数据类型

·数组类型，空数组转为空字符串，其他数组拼接各个元素形成字符串

·函数类型，输出函数表达式

·Object转为[object Object]

## 1.10 其他值到数字值的转换规则

基本数据类型

·Undefined 类型的值转换为 NaN

·Null 类型的值转换为 0

·Boolean 类型的值，true 转换为 1，false 转换为 0

·String 类型包含非数字值则转换为 NaN，空字符串为 0

·Symbol 类型的值不能转换为数字，会报错

引用数据类型

·Object转换为NaN，日期对象返回对应的时间戳（日期对象重写了valueof方法）

·函数类型转为NaN

·空数组转为0，单个数字数组元素转为对应的数字，其他数组转为NaN

1.11 转换为Boolean

基本数据类型

undefined、null、' '、0、NaN 转为false， 其他转为true，

引用数据类型

全部转为true

## 1.12 || 和 && 操作符的返回值

|| 和 && 将第一个操作数强制转换为布尔类型，然后执行条件判断。

·对于 || 来说，如果条件判断结果为 true 就返回第一个操作数的值，如果为 false 就返回第二个操作数的值。

·&& 则相反，如果条件判断结果为 true 就返回第二个操作数的值，如果为 false 就返回第一个操作数的值。

## 1.13 Object.is(a,b) 与比较操作符 `===`、`==`的区别

·双等号==，如果两边的类型不一致，则会进行强制类型转化后再进行比较。

·三等号===，如果两边的类型不一致时，不会做强制类型准换，直接返回 false。

·Object.is 和三等号的判断相同，它处理了一些特殊情况，比如 -0 和 +0 不再相等，两个 NaN 是相等的。

## 1.13 什么是 JavaScript 中的包装类型

在 JavaScript 中，基本类型是没有属性和方法的，但是为了便于操作，在调用基本类型的属性或方法时，会在后台隐式地将基本类型的值转换为对象。

也可以使用Object函数显式地将基本类型转换为包装类型。

## 1.14 JavaScript 中如何进行隐式类型转换

1. ToPrimitive(primitive：原始的)方法，这是 JavaScript 中每个值隐含的自带的方法，用来将值 （无论是基本类型值还是对象）转换为基本类型值。如果值为基本类型，则直接返回值本身。

对象（obj，array，function）先调用对象的valueOf()，如果返回的不是基本数据类型值，再调用toString()。

2.+ 操作符号

+ 操作符的两边有至少一个string类型变量时，两边的变量都会被隐式转换为字符串；其他情况下两边的变量都会被转换为number。

3.== 运算符号 操作数两边都转换成number

4.布尔操作符 ！ && ||

## 1.15 JS基本和非基本数据类型之间的区别

基本数据类型的变量存放的是值，而非基本数据类型的变量存放的是内存地址，指向他们在内存中的位置。

基本数据类型保存在栈内存中，而非基本数据类型保存在堆内存中。

基本数据类型的复制是深拷贝，而引用数据类型的复制是浅拷贝

## 1.16 数组有哪些原生方法

1. 数组和字符串的转换：

tostring()：使用数组名称调用它，返回包含以逗号分隔的数组元素的字符串。

join(string s)：返回包含以指定符号分隔的数组元素的字符串。

1. 数组的栈操作:

push()：数组尾部插入一个元素

pop()：删除数组尾部的元素并返回该元素

1. 数组的队列操作:

shift()：把数组的第一个元素从其中删除，并返回第一个元素的值。

unshift()：unshift()向数组头部添加元素

1. 数组的分割操作:

slice(start, end) ：返回一个新的数组，包含从 start（包括该元素） 到 end （不包括该元素）的元素。不影响原数组

1. 数组的插入操作:

splice(index, howmany, item1, ....., itemX) ：index:起始位置，howmany:要删除的项目数，item1...itemX:要增添的新项目。影响原数组

splice的返回值为删除的元素

1. 数组的连接操作:

concat(array2, array3, ..., arrayX)：用于连接两个或多个数组，不影响原数组，返回拼接好的新数组

1. 查找特性索引:

indexOf(item, start)：在数组中搜索item，并返回其位置。可以通过start指定搜索起始位置

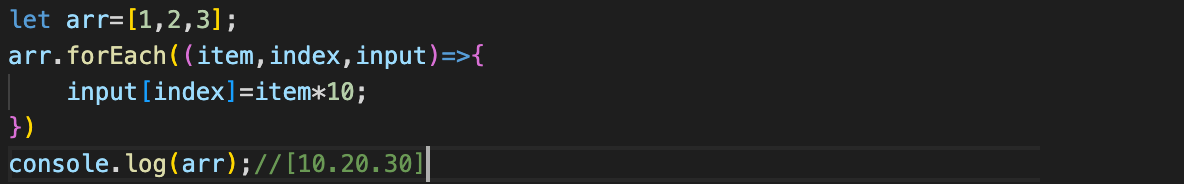
lastIndexOf()：在数组中搜索item，并返回其位置。从末尾开始搜索

1. 迭代方法:

every()：检查数组中的所有元素是否都通过了测试，所有元素通过返回true

some()：检查数组中的任何元素是否通过测试，有一个通过返回true

forEach()：按顺序为数组中的每个元素调用一次函数，修改原数组。但是要获取第三个参数，即原始数组：

通过input[index]=item\*10可以修改原数组，只通过简单的item=item\*10是不行的。但这种方法还是通过修改数组下标来实现的。

本质上还是深拷贝和浅拷贝的问题，数组中的项，如果是基本数据类型，foreach进行了深拷贝，修改不影响原数组。如果数组中的项为对象，那么就是浅拷贝，修改数组中的项会影响到原数组

map()：根据为每个数组元素调用函数的结果，来创建新数组。

filter()：创建新数组，其中是所有通过测试的数组元素

1. 归并方法:

reduce()：为数组的每个值（从左到右）执行提供的函数，函数的返回值存储在累加器中（结果/总计）

reduceRight()：为数组的每个值（从右到左）执行提供的函数，函数的返回值存储在累加器中（结果/总计）。

1. 数组排序：

reverse()：反转数组中元素的顺序。

sort()：对数组的项目进行排序

11.包含方法：

arr.include(target,index):从index开始搜索，判断arr中是否包含target。返回true/false

## 1.17 数组的reduce方法

对数组中的每个元素执行reduce函数(升序执行)，将其结果汇总为单个返回值

## 截屏2022-05-08 16.44.441.18 数组扁平化

数组扁平化就是将一个多维数组转换为一个一维数组

## 1.19 清空数组

1.arr.length=0

2.arr.splice(0,arr.length)

3.arr=[]

## 1.20 JS严格模式

严格模式的意义：

去除JS中的一些不合理、不严谨之处，减少一些怪异行为;

- 保证代码运行的安全；

- 提高运行速度；

- 为未来新版本的Javascript做好铺垫。

严格模式主要有以下限制：

1.变量必须声明后再使用

2.函数不能有同名参数

3.禁止使用with语句

4.禁止对只读属性赋值

5.禁止this指向window

# 2 JS变量提升

## 2.1 深拷贝和浅拷贝

·浅拷贝：就是假设B复制了A，当修改A时，看B是否会发生变化，如果B也跟着变了，说明这是浅拷贝。

·深拷贝：修改A的时候B不改变

深拷贝方法：

1.assign,复制一个对象。本质上是基本数据类型深拷贝，引用数据类型浅拷贝

2.先json.stringify ，再json.parse。function和symbol会丢失

## 2.2 JS变量提升

把变量声明提升到它所在作用域的顶端去执行，使得变量在声明之前也可使用

1. console.log(v1);
2. var v1 = 100;
3. function foo() {
4. console.log(v1);
5. var v1 = 200;
6. console.log(v1);
7. }
8. foo();
9. console.log(v1);

输出 undefined undefined 200 100

## 2.3 let、const、var的区别

1.块级作用域： let和const具有块级作用域，var不存在块级作用域

2.初始值：const必须有初始值，let和var不是

3.是否被修改：const不能被修改，let和var可以

4.暂时性死区。let和const有暂时性死区，var没有

5.全局属性：var会添加全局属性，let和const不会

6.重复声明变量：var可以重复声明变量，const和let不允许重复声明变量

## 2.4 JavaScript为什么要进行变量提升，它导致了什么问题

JavaScript 运行时分为两步：预解析和代码执行。

js变量提升的原因是预解析。预解析分为变量预解析和函数预解析。变量预解析也叫变量提升。就是把所有的变量声明提升到当前作用域的最前面，但不进行赋值操作。函数预解析也叫函数提升，就是把所有的函数声明提升到当前作用域的最前面，但不调用函数

**变量提升的原因**

**提高性能**

预先为变量分配内存空间

**容错性更好**

使一些不规范的代码也可以正常执行

## 2.5 const对象的属性可以修改吗

const保证的并不是变量的值不能改动，而是变量指向的那个内存地址不能改动。对于基本类型的数据，const变量不能改变

对于引用类型的数据来说，变量指向数据的内存地址，保存的只是一个指针，const只能保证这个指针是固定不变的，至于它指向的数据结构是不是可变的，就完全不能控制了。

Ps：js中，不可以直接访问堆内存空间中的位置和操作堆内存空间。只能操作对象在栈内存中的引用地址。

# 3 JS对象

## 3.1 new操作符的过程

·创建一个对象

·为对象添加原型链。（将对象的\_proto\_属性指向构造函数的prototype属性）

·绑定this

·也就是为这个对象添加属性和方法

·返回新的对象

## 3.2 对象创建的方式有哪些

**第一种是工厂模式**。

使用new Object创建对象，没有建立起对象和类型间的关系。

**第二种是构造函数模式**。

构造函数模式所创建的对象和构造函数建立起了联系，可以通过原型来识别对象的类型。缺点是每个实例对象上都有构造函数的方法，造成浪费

**第三种模式是原型模式**

使用原型对象来添加公用属性和方法，解决了公有函数的复用问题。缺点：某个实例对象修改了原型上的引用数据类型，会影响到其他实例对象

## 3.3 对象继承的方式有哪些

1. **以原型链的方式来实现继承**

其基本思想是使子类型的原型对象指向父类型的实例对象

缺点：

引用数据类型被所有的实例对象所共享

还有就是在创建子类型的时候不能向父类型传递参数。

1. **借用构造函数实现继承**

将父类构造函数引入子类构造函数，来增强子类实例

缺点：

只能继承父类的实例属性和方法，不能继承原型属性/方法；

无法实现构造函数的复用，每个子类都有父类实例函数的副本

1. **组合继承**

使用原型链继承原型上的属性和方法，使用构造函数继承实例的属性

缺点：

父类中的实例属性和方法既存在于子类的实例中，又存在于子类的原型中

## 3.4 箭头函数与普通函数的区别

·箭头函数比普通函数更加简洁

没有参数就只写一个括号就行，如果只有一个参数，可以省去参数的括号。

·箭头函数没有自己的this

箭头函数没有自己的this，它继承上一层作用域的this。

·箭头函数继承来的this指向永远不会改变

箭头函数的this永远指向最近一层作用域的this指向

## 3.5 扩展运算符的作用及使用场景

扩展运算符(…)取出对象中的所有可遍历属性，赋值给新对象

## 3.6 Proxy

Proxy （代理）是 ES6 中新增的功能，可以在访问对象前添加一层拦截

**let p = new Proxy(target, handler)**

target 代表需要添加代理的对象，handler 用来定义拦截操作

## 3.7 Map和Object的区别 （Map是键值对）

1.map的key可以为任意数据类型，object必须为string或者symbol

2.map有序，object无序

3.map对频繁的插入删除有优化，object没有

4.map通过for of遍历，object通过for in遍历

5.map.size直接获取长度，object必须手动计算

Object.getOwnPropertyNames(map).length

## 3.8 JavaScript有哪些内置对象

Math、Data、Array、Promise...

## 3.9 JSON的理解

JSON是一种基于文本的轻量级的数据交换格式，用作前后端数据交换的方式。

JSON.stringify 函数，将js数据结构，转换为 JSON 字符串。

JSON.parse() 函数，将 JSON 格式的字符串转换为 js 数据结构

## 3.10 什么是 DOM 和 BOM

DOM 指的是文档对象模型，定义了访问页面的接口和方法。

BOM 指的是浏览器对象模型，定义了访问浏览器的接口和方法。BOM的核心是 window，而 window 对象具有双重角色，它既是通过 js 访问浏览器窗口的一个接口，又是一个 Global（全局）对象。这意味着在网页中定义的任何对象，变量和函数，都作为全局对象的一个属性或者方法存在。

document对象是window 对象的一个属性，除此之外，还有 location 、navigator

## 3.11 常用的DOM操作方法

**DOM 节点的获取**

Document.getElementById()

Document.getElementByTagName()

Document.getElementByClassName()

**DOM 节点的创建**

document.createElement()

xxx.appendChild()

**DOM 节点的删除**

xxx.removeChild()

## 3.13 如何判断一个对象是否属于某个类

instanceof、constructor

# 4 JS的异步

## 4.1 延迟加载的方式有哪些

·**defer 属性：** 给 js 脚本添加 defer 属性，这个属性会让脚本的加载与文档的解析同步解析，然后在文档解析完成后再执行这个脚本文件

·**async 属性：** 给 js 脚本添加 async 属性，这个属性会使脚本异步加载，不会阻塞页面的解析过程，但是加载完立即执行，可能会阻止页面解析

·**使用 setTimeout** 延迟方法 延时函数

·**让 JS 最后加载** 把js部分放在页面的最底部，最后加载

## 4.2 对AJAX的理解

AJAX（异步javascript和xml），指的是通过 JavaScript 的异步通信，从服务器获取数据，再更新当前网页的对应部分，而不用刷新整个网页.

## 4.3异步编程的实现方式

**回调函数** ：多个回调函数嵌套的时候会造成回调函数地狱

**Promise** ：将嵌套的回调函数作为链式调用

**generator：**当遇到异步函数执行的时候，将函数执行权转移出去，当异步函数执行完毕时再将执行权给转移回来

**async ：**async 函数是 generator 和 promise 的语法糖。函数内部执行到 await 语句的时候，如果语句返回 promise 对象，那么函数将会等待 promise 对象的状态变为 resolve 后再继续向下执行

## 4.4 Promise

Promise是一个构造函数，自身上有all、reject、resolve方法，原型上有then、catch等方法。

new一个promise

var p = new Promise(function(resolve, reject){

//做一些异步操作

setTimeout(function(){

console.log('执行完成');

resolve('随便什么数据');

}, 2000);

});

Promise的构造函数接收一个函数作为参数，并且传入两个参数：resolve，reject，resolve是将Promise的状态置为fullfiled，reject是将Promise的状态置为rejected。

Promise把原来的回调写法分离出来，在异步操作执行完后，用链式调用的方式执行回调函数。

## 4.5 promise链式回调

Promise通过维护状态、传递状态的方式来使得回调函数能够及时调用

## 4.6 promise的reject

reject把Promise的状态置为rejected，在then中捕捉，执行“失败”情况的回调

## 4.7 promise的catch用法

Promise.catch是 then(undefined, () => {}) 的语法糖

在执行resolve的回调时，如果抛出异常，会进到这个catch方法中。

1.同步代码发生错误，会被catch捕获，包括new promise和then中的错误

2.异步代码发生错误(throw error)，不会被catch捕获，catch是微任务，先于setTimeout的宏任务执行，但是如果是返回reject的promise，那么会被then中的失败回调捕捉，因为reject会触发已经注册过的失败回调函数

3.promise调用resolve之后，再抛出错误不会被捕捉

4.new promise产生的错误，如果被then中的失败函数捕捉，则不会再触发catch，还是因为catch是then的语法糖

## 4.8 promise的all

提供了并行执行异步操作的能力，并且在所有异步操作执行完后才执行回调。

用Promise.all来执行，all会等到数组中的所有Promise对象都执行完毕后，才会返回promise对象，进入then。数组中的所有Promise对象全部为resolve，promise.all才返回一个resolve的promise。数组中的所有Promise对象一旦有一个为rejected，那么all就返回reject的promise

## 4.9 promise的race

Promise.race([p1, p2, p3])返回执行结果最快的那个promise，不管结果本身是成功状态还是失败状态。

## 4.10 promise的finally

不管 Promise 对象最后状态如何，都会执行的操作

## 4.11 promise的then

promise里面的then函数仅仅是注册了后续需要执行的代码，真正的执行是在resolve方法里面执行的

## 4.12 Promise的any

接收一个Promise可迭代对象，只要其中的一个 promise 成功，就返回那个已经成功的 promise 。如果可迭代对象中没有一个 promise 成功（即所有的 promises 都失败/拒绝），就返回一个失败的 promise

# 5 函数的调用和作用域

## 5.1 console.log 和console.dir的区别

console对象是JavaScript的原生对象,log和dir都可以在控制台输出信息。

·console.log()会在浏览器控制台打印出信息。但是显示的信息有些不全。

·console.dir()可以显示一个对象所有的属性和方法。主要用于对对象进行检查。

## 5.2 尾调用

尾调用指的是函数的最后一步调用另一个函数。只能在严格模式下开启

在一个函数里调用另一个函数时，会保留当前的执行上下文，然后再新建另外一个执行上下文。使用尾调用的话，因为已经是函数的最后一步，所以这时可以不必再保留当前的执行上下文，从而节省了内存。

## 5.3 js 执行机制、事件循环

JavaScript 是单线程的，它将所有任务分成两种，一种是同步任务，另一种是异步任务。同步任务进入主线程排队执行，形成执行栈。异步的任务进会交给浏览器的其他线程进行处理。当异步任务完成时，将异步任务的回调函数加入任务队列。主线程内的任务执行完毕为空，会去任务队列选取回调函数，进入主线程执行。这个过程不断重复，构成了js的事件循环

## img5.4 宏任务、微任务

在事件循环中，任务一般都是由宏任务开始执行的（script标签）。宏任务执行过程中会产生新的宏任务和微任务。微任务进入微任务队列，宏任务进入宏任务队列。当前宏任务执行完毕后，先检查微任务队列，如果微任务队列不为空，则执行微任务，直到微任务队列清空。然后把宏任务队列中下一个宏任务加入执行栈并执行，执行后清空执行栈，再去检查微任务。再把下一个宏任务加入执行栈，以此循环往复。

宏任务：包括整体代码script，setTimeout，setInterval，I/O。

微任务：Promise.then（必须执行到resolve后才将then()放入微任务队列）和catch、finally，process.nextTick。async/await(await是promise.then的语法糖)

### 5.4.1 js事件循环机制，为什么会有宏任务和微任务两种类型的异步任务

宿主环境不同，宏任务是浏览器发起的，微任务是js引擎发起的

## 5.5 for…in和for…of的区别

for...in遍历一个对象的除Symbol以外的可枚举属性

for...of在可迭代对象（包括 Array，Map，Set，String，TypedArray，arguments 对象等等）上创建一个迭代循环，可以获取可迭代对象的值

for…in 获取的是对象的key，for…of 遍历获取的是对象的value，

for…in 为了遍历对象而生，for...of不能遍历对象

## 5.6 遍历数组的方法

**1.普通for循环**

最简单的一种，也是使用频率最高的一种，虽然性能不弱，但仍有优化空间

**2.forEach循环**

数组自带的foreach循环，使用频率较高，实际上性能比普通for循环弱

**3.for in 循环**

经分析测试，在众多的循环遍历方式中，**它的效率是最低的**

for in会遍历原型链上所有的可枚举属性。常用来遍历对象

**4.map遍历**

这种方式也是用的比较广泛的，实际效率还比不上foreach。

**5.for of循环**

这种方式是es6里面用到的，性能要好于for in，但仍然比不上普通for循环

## 5.7 遍历map键值对方法

For of

## 5.8 遍历的forEach和map方法有什么区别

forEach()会改变原数组，该方法没有返回值。forEach 循环无法中途跳出

map()不改变原数组的值，返回新数组

## 5.9 原型、原型链的理解

每个构造函数都有 prototype 属性，它是一个对象，包含了实例共享的属性和方法

当使用构造函数新建一个实例对象后，可以通过\_\_**proto\_\_** 属性来访问原型对象

访问对象属性时，如果对象不存在这个属性，就会去它的原型对象里找这个属性，这个原型对象又会有自己的原型，也就是原型链的概念。原型链的尽头是 Object.prototype

## 5.10闭包的理解

闭包是在嵌套函数中，内部函数可以访问外部函数的局部变量。「内部函数」和「内部函数内部能访问到的变量」的总和，就是一个闭包。

闭包的用途：使已经运行结束的函数上下文中的变量对象继续留在内存中

## 5.11 作用域

作用域为可访问变量，对象，函数的集合。

**全局作用域**

·最外层函数和最外层函数外面定义的变量拥有全局作用域

·所有window对象的属性拥有全局作用域

**函数作用域**

·函数所包裹形成的作用域

**块级作用域**

·使用ES6中新增的let和const指令可以声明块级作用域，块级作用域可以在函数中创建也可以在一个代码块中的创建（由{ }包裹的代码片段）

·在循环中比较适合绑定块级作用域，可以把声明的计数器变量限制在循环内部

## 5.12 作用域链

通过作用域链，可以访问到外层环境的变量和函数

作用域链连接的是执行上下文中的变量对象。变量对象是一个包含了执行环境中所有变量和函数的对象。作用域链的前端是当前执行上下文的变量对象。作用域链的最后一个对象是全局执行上下文的变量对象

在当前作用域中查找所需变量，但是该作用域没有这个变量，就沿着作用域链去上一级作用域中查找

## 5.13 对this对象的理解

this 是执行上下文中的一个属性，它指向最后一次调用这个方法的对象。

函数调用模式：this 指向全局对象。

方法调用模式：this 指向调用这个方法的对象。

构造器调用模式：this 指向这个新创建的对象。

apply、call 和 bind 调用模式：指定调用函数的 this 指向。

内部函数模式：嵌套函数中的this不会继承上层函数的this

## 5.15 javascript 的垃圾回收机制

**1. 标记清理（mark-and-sweep）**

标记清理是js最常用的垃圾回收策略。先标记内存中储存的所有变量，然后将执行上下文中的变量以及被引用的变量去除标记，最后清除所有带标记的变量

**2. 引用计数**

变量被引用一次，引用值+1，释放内存时清除掉所有引用值为0的变量。但是会存在循环引用的问题

## 5.16 内存泄漏

定义：不再用到的内存，没有及时释放

·全局变量

·定时器

·闭包

·DOM 元素引用

## 5.17 let解决for循环

关于经典for循环的问题：

var btns = document.getElementsByTagName('button');

for(var i = 0;i<btns.length;i++){

btns[i].onclick = function () {

alert(i);

}

}

点击三个button时，都会输出3（假定总共三个按钮）

为什么？

1. var没有块级作用域
2. 事件响应函数通常在非回调函数解析完后执行

(3)变量的访问是从当前作用域开始，顺着作用域链向上查找的。

事件响应函数与全局变量i构成闭包

**1.使用let解决**

var btns = document.getElementsByTagName('button');

for(let i = 0;i<btns.length;i++){

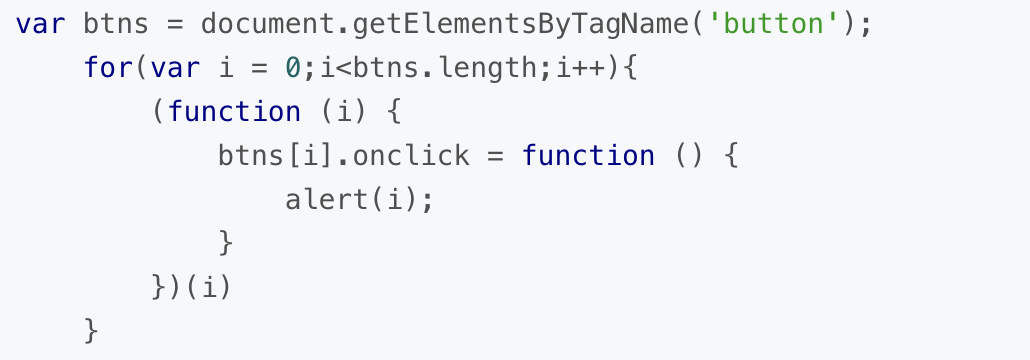
btns[i].onclick = function () {

alert(i);

}

}

事件响应函数与其块作用域中的变量i形成闭包

1. **使用闭包解决**

事件响应函数与匿名函数的参数i形成了闭包，维持着对参数i的引用

## 5.18 函数柯里化

柯里化（Currying）,把接受多个参数的函数转换成接受一个参数的函数

## 5.19 eval

eval() 函数会将传入的字符串当做 JavaScript 代码进行执行，如果传入的字符串是表达式则返回表达式求值结果，否则返回 undefined

表达式：由运算元和运算符(可选)构成，并产生运算结果的语法结构

简单来说，就是可以在左边用变量和=符号接收的运算结果

# 实战篇

## 字节JS实战篇

### 6.1.1怎么判断一个对象是不是可迭代的

1.能使用for in遍历的就是可迭代对象

2.检测对象上的Symbol.itreator是否存在 obj[Symbol.iterator]

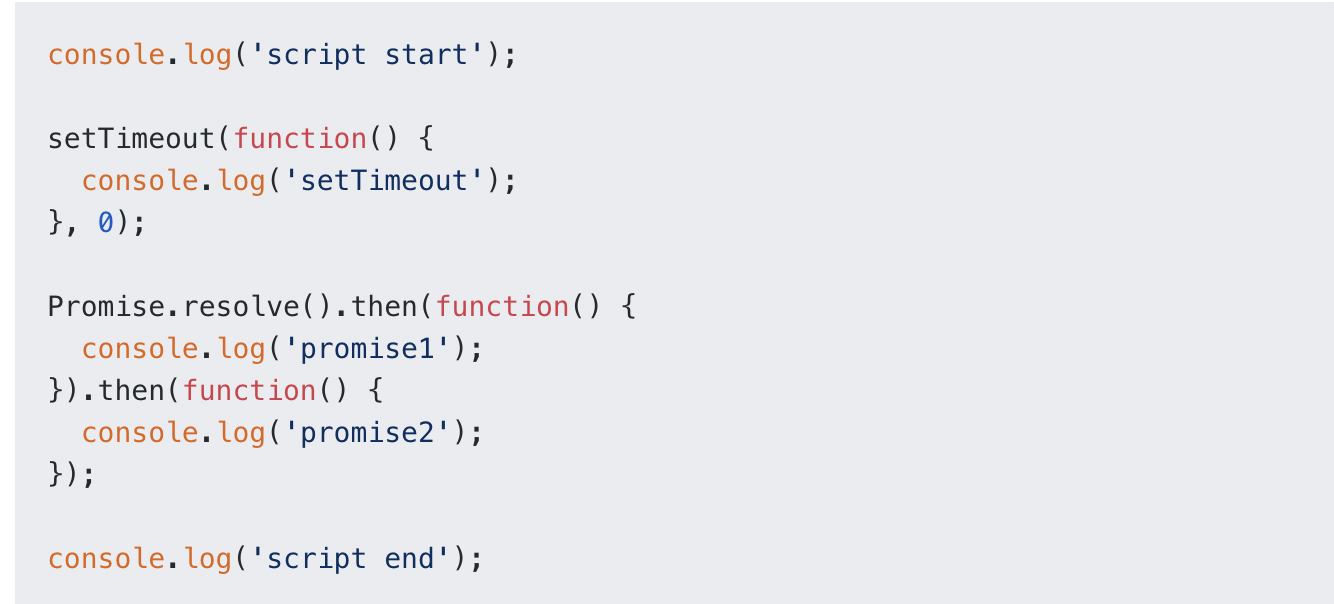
### 6.1.2 如何修改一个数组的副本而不更改原数组

slice、map、concat，filter

### 6.1.4 说说宏任务和微任务的执行顺序

<https://www.jb51.net/article/215137.htm> 这篇帖子讲的非常清晰透彻！

**在执行一个 Promise 对象的时候，当走完resolve()之后，就会立刻把 .then()里面的代码加入到微任务队列当中**

**这是一道经典的面试题**

正确答案是

script start, script end, promise1, promise2, setTimeout

宏任务：setTimeout为它的回调产生一个新的宏任务。所以‘setTimeout’在‘script end’之后。

微任务：如果在微任务执行期间微任务队列加入了新的微任务，会将新的微任务加入队列尾部，之后也会被执行。‘promise1’,'promise2'会打印在‘setTimeout’之前是因为微任务队列中的所有微任务总会在下一个宏任务之前全部执行完毕。

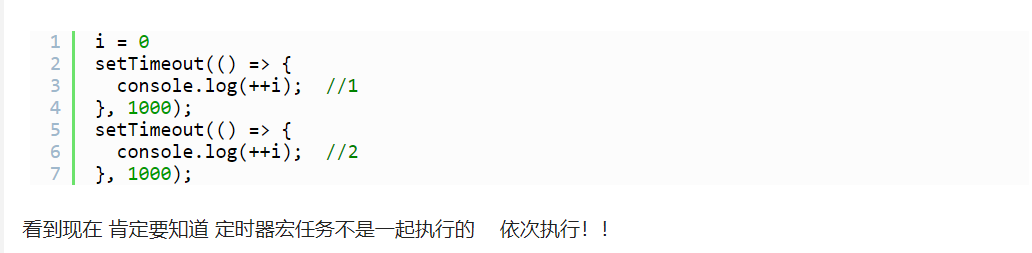
另一个例子：

### 6.1.5 **两个定时器 上面的定时器先执行 再执行下面的定时器吗？**



二者同时在浏览器的定时器模块运行，第二个先到达时间，先加入宏任务队列

### 6.1.6 **设置两个一样的定时器事件，会输出什么结果**

第一个定时器的回调函数先加入宏任务队列，第二个定时器的回调函数后加入宏任务队列。因此先输出1，再输出2

### 6.1.8 JS语法糖

为了方便程序员操作而添加的某种语法，对功能并没有影响。

能够增加程序的可读性，从而减少程序代码出错的机会。

**1.async/await**

**2.对象字面量**

**3.箭头函数**

**4.数组解构**

**5.拓展运算符**

**6.模块化开发**

### 6.1.9 async和await

定义：async 函数是 Generator 函数的语法糖，使用关键字 async 来表示异步函数，在函数内部使用 await 来表示异步。将异步逻辑转换为同步顺序执行

**async**：表示函数是一个异步函数，调用之后立即执行，它不会造成阻塞。

1.async内部返回一个promise对象，可以调用then方法。如果返回的不是promise对象，而是普通值，则会将普通值resolve为promise

2.async如果没有返回值，则会返回一个resolve的promise，值为undefined

**await**：表明异步等待，只能放到async函数里面，会引起async函数内部阻塞

1.后面跟一个返回promise对象的表达式。等到Promise有返回值的时候，代码才继续执行下去。await后面如果是正常值，会被resolve为一个promise

2.必须等到所有await后的promise都有了返回值，async才会发生状态改变

3.await是promise.then的语法糖，await之后的语句，相当于调用了promise.then，会被当作微任务，放到本次宏任务执行完毕之后执行

4.一个await返回了reject的promise，则后续的await都不会执行，需要使用try catch对await进行监听(一个try catch包含若干个await，那么某个await后的promise变为reject，后续await不会执行，直接进入错误捕获阶段。如果是每个await都对应一个try catch，那么某个await后的promise变为reject，后续await仍然可以执行)

5.使用await promise.all同时进行两个独立的异步操作，不会形成继发关系

因为await可能会返回reject的promise，会导致后续的所有await都不执行，因此需要进行错误捕获，一般使用try catch对async函数内部的所有await进行捕获。在async外部，使用promise.catch进行捕获

触发catch的几种情况：

1. **函数内部抛出错误，返回的 Promise 对象变为 reject ，触发promise.catch**
2. **async函数返回一个拒绝的期约，会触发async的promise.catch**
3. **async中出现了拒绝的期约，不会触发catch**
4. **await后面跟一个拒绝的期约，会触发async的promise.catch和try catch**

**5.await之后跟一个抛出的错误，会触发async的promise.catch**

### 6.1.10 JS为什么要有异步操作

因为js单线程，同一时间只能做一件事。遇到需要长时间等待的任务会卡住，造成cpu浪费。而使用异步处理这些长时间等待的任务，可以使得cpu继续去做其他的事情

### 6.1.12 call、apply、bind的区别

call 和 apply

1.第一个参数都是this指向

2.call传入若干个参数作为函数参数，apply传入参数数组

3.函数会立即执行bind

1.改变this的指向2.bind()返回一个绑定了this的新函数，不会立即调用

### 6.1.13 判断一个数据是不是数组的方法

**Array.isArray** ：Array.isArray(a)

**\_proto\_** ：a.\_\_proto\_\_ === Array.prototype

**instanceof ：**a instanceof Array

**constructor：**a.constructor === Array

### 6.1.14 数组去重

1.双重for循环

2.Array.from(new Set(a));

new Set(a)：通过a数组创建set，set的特性之一是不能有重复元素，因此，a转换为set之后，就去除了重复元素

Array.from()：from() 方法用于通过拥有 length 属性的对象或可迭代的对象来返回一个数组。即from再把刚才转换的set再次转换回数组。

**6.1.17 关于prototype和\_\_proto\_\_ （字节一面原题）**

prototype是构造函数的属性，指向该构造函数对应的原型对象。而\_\_proto\_\_则是实例对象所拥有的属性，指向创建它的构造函数所对应的原型对象

### 6.1.18 js函数的节流与防抖

节流和防抖，都是优化js执行的一种手段

**函数节流是**触发函数事件后，短时间内无法再次触发。只有等到一定的时间间隔后才会再次触发。**例子：登录按钮防止连续点击提交。**  
**函数防抖**是多次触发事件后，只执行一次事件处理函数。**例子：输入框输入**

**总结：即节流是不允许连续触发；防抖是允许连续触发，但只响应一次**

## 6.2 腾讯

### 6.2.1 手写new操作符

要手动实现一个 new 操作符，首先要知道 new 操作符都做了什么事，即构造函数的内部原理：

1.创建一个新对象；

2.链接到原型（将构造函数的 prototype 赋值给新对象的 \_\_proto\_\_）；

3.绑定this（构造函数中的this指向新对象并且调用构造函数）

4.为新对象添加实例属性和方法

5.返回新对象

## 6.2.2.require /exports 和import/export

require 是CommonJS规范引入方式。import是ES6的一个语法标准

require是运行时调用。import是编译时调用

require是赋值过程，将require的结果赋值给某个变量。import是解构过程

require和exports配合使用。import和export配合使用

### 6.2.3 ES6新特性整理

1.let、const

1. 箭头函数
2. 数组解构

4.扩展运算符

5.symbol、bigint

6.promise

7.class

8.for of

9.async/await

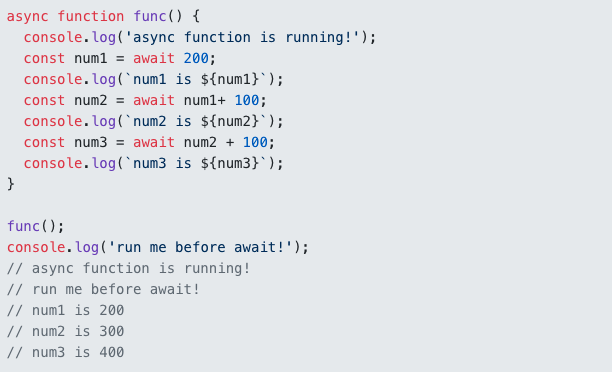
10.module

## 6.3 ES6新函数

Map、forEach、filter、every、some、reduce、include、find

## 6.4 async和promise输出顺序：

1.

分析：async立即执行，输出async function is running，遇到await，阻塞。async函数让出处理器，执行run me before await。本轮同步代码执行完毕之后，会继续将处理器交换async，继续执行await后面的语句，陆续输出num1 is 200、num2 is 300、num3 is 400

2.

输出结果：

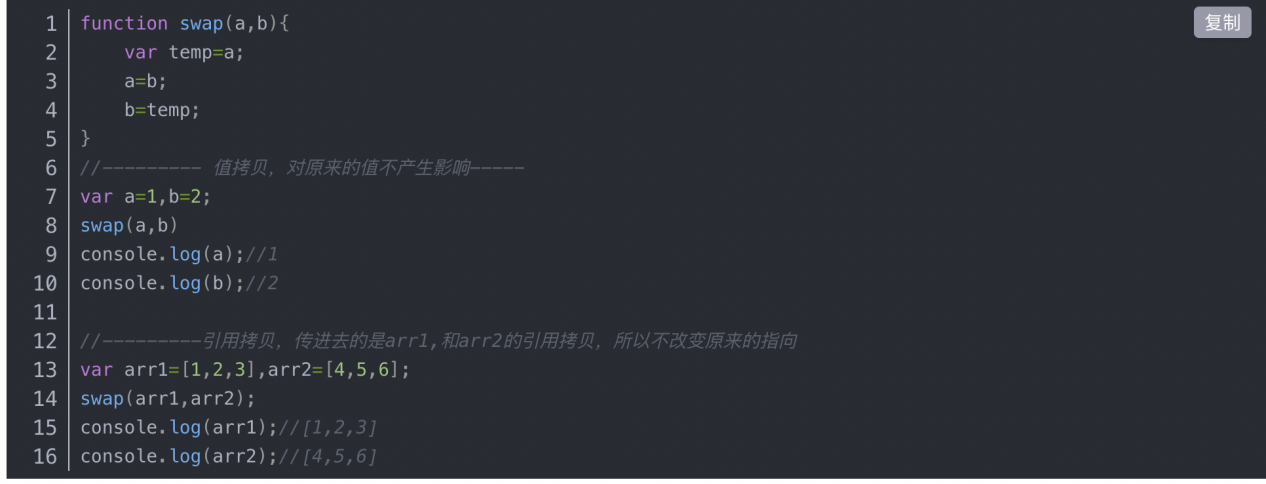


## 6.5 js的值拷贝和引用拷贝

简单值（基本类型:undefined,null,string,number,boolean,sympol)总是通过值拷贝的方式来赋值/传递。

复合值——对象（数组和封装对象）和函数总是通过引用拷贝的方式来赋值

例子：

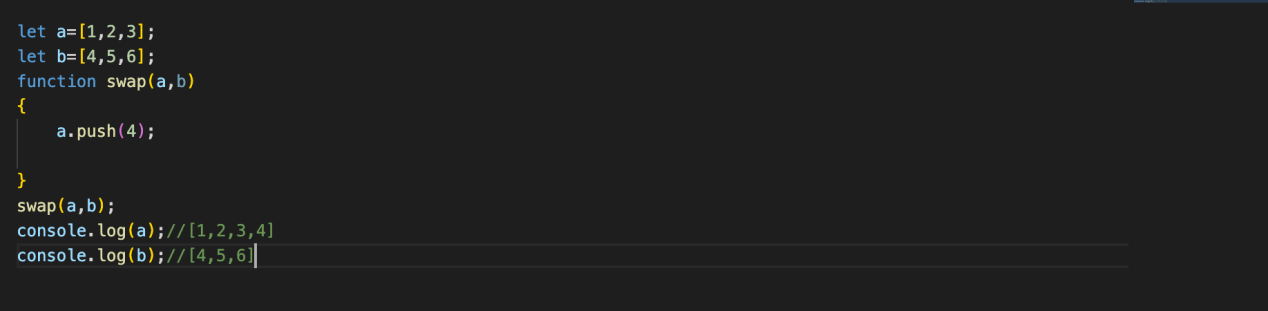
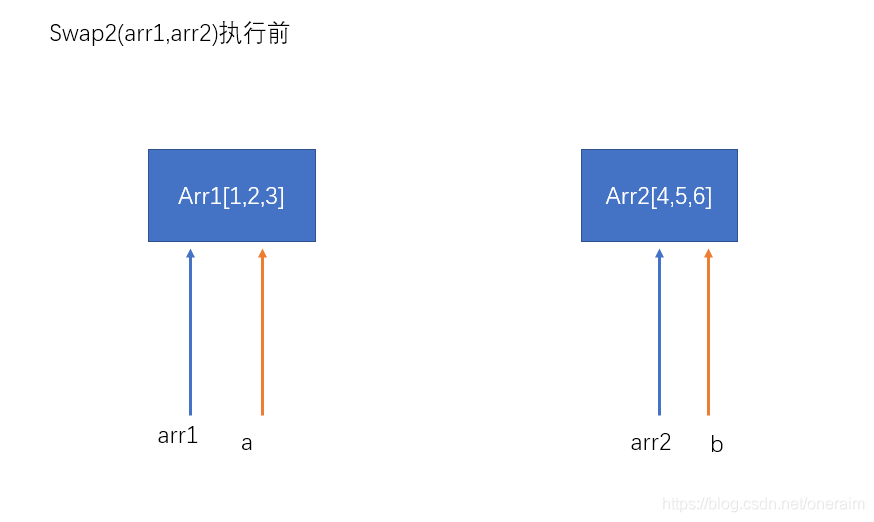
数组arr1和arr2指向内存中的[1,2,3]和[4,5,6]，然后将arr1和arr2传递给a和b，使得a和b指向内存中的[1,2,3]和[4,5,6]，交换a和b，a指向[4,5,6]，b指向[1,2,3]，但是arr1和arr2的指向仍然不变，arr1仍然指向[1,2,3]，arr2仍然指向[4,5,6]。

JavaScript没有指针，JavaScript的变量不可能成为指向另一个变量的引用。JavaScript引用的是值，如果一个值有10个引用，这些引用指向的都是同一个值，他们之间没有引用/指向关系

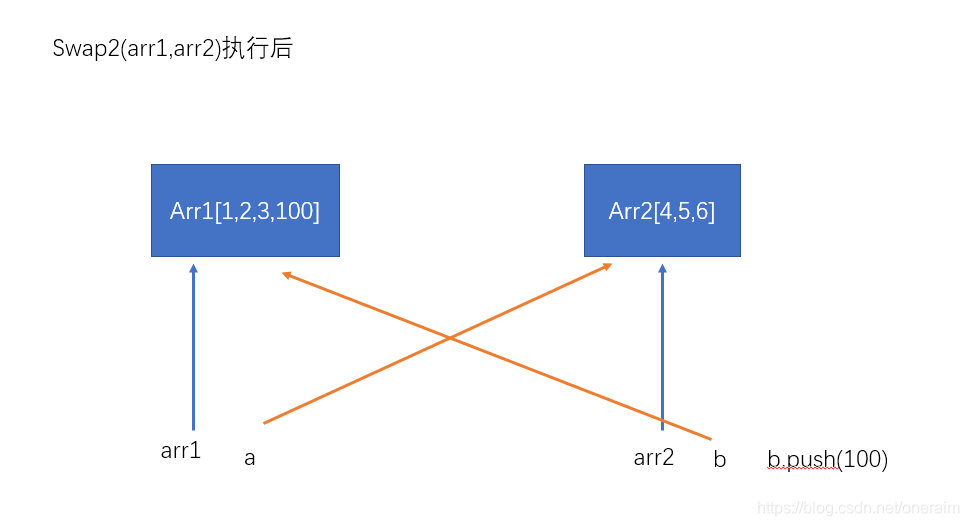
对于基本数据类型来说，a=1传递给了函数swap，实际上是把数字1传递给了swap，swap中的参数a接收到了数字1，此时参数a已经变成了另外一个变量，与外层传入的a没有了任何关系。无论对参数a进行怎么样的改变（交换a和b，令a=3），都永远不会再改变外层的a了，外层的a永远等于1。

对于引用数据类型来说，arr1=[1,2,3]传递给了swap，实际上是把[1,2,3]的内存地址给了swap，swap的参数a接收到了[1,2,3]的内存地址，因此，a也指向了[1,2,3]。换句话来说，[1,2,3]内存中只有一份，有两个变量可以访问到它，一个是arr1，一个是参数a。因此，假如令参数a=[4,5,6]，那么就相当于参数a中保存的内存地址，被更改为数组[4,5,6]的内存地址，数组[1,2,3]并没有改变，只是参数a访问不到它了，但是arr1仍然能访问到[1,2,3]。但是如果我们进行a.push(4)，那么无论是a还是arr1，都会变为[1,2,3,4]，因为通过参数a，数组[1,2,3]变为了[1,2,3,4]，那么所有之前指向[1,2,3]的变量，都会变为[1,2,3,4]，因为数组[1,2,3,4]在内存中只有一份

a.push(4)之后

arr1、a、arr2、b在swap（swap中交换了a和b）之前的指向：

arr1、a、arr2、b在swap（swap中交换了a和b）之后的指向：

可以看出，arr1和arr2并不受a和b的影响

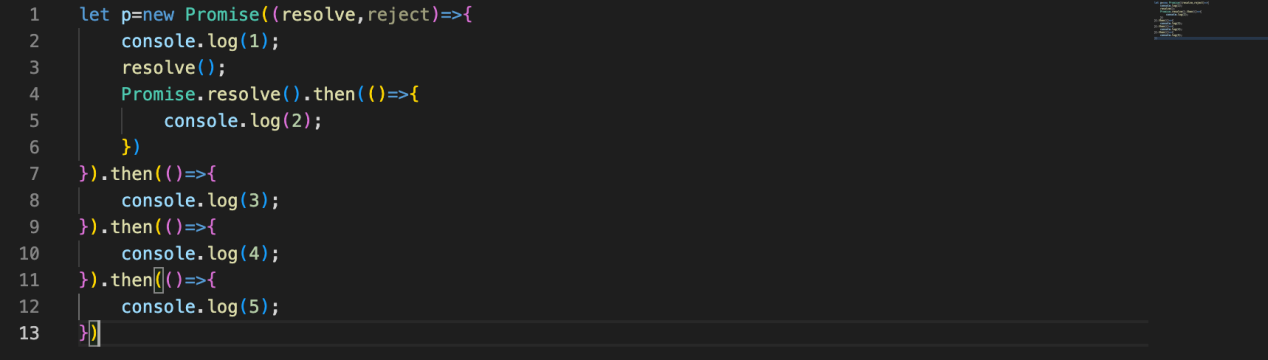
## 6.6 js number的最大值

2的53次方,为9007199254740992(16位)

## 6.7 js基本数据类型占多少个字节

8个字节

## 6.8 promise.then什么时候加入微任务队列



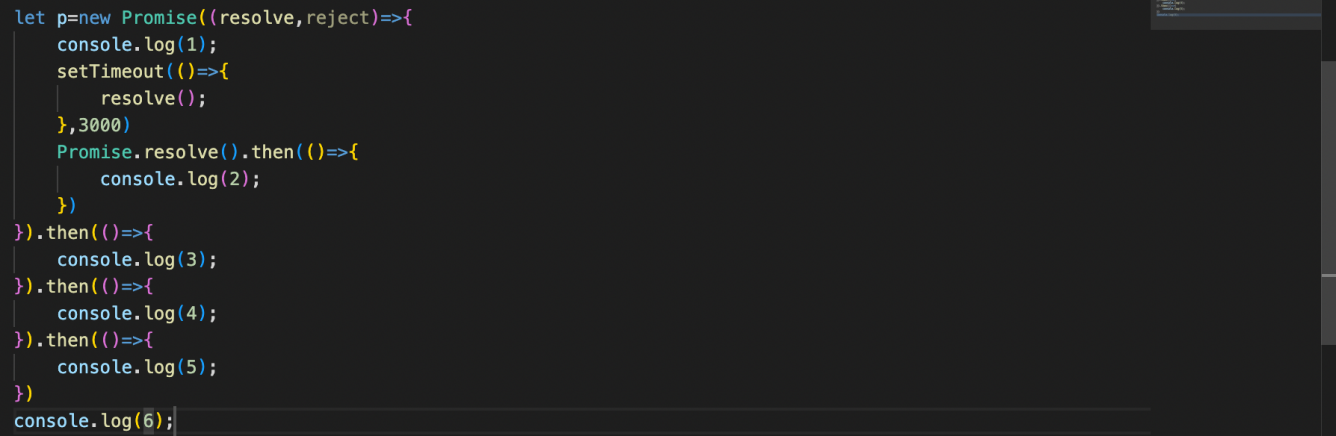
输出：1，2，3，4，5

1首先输出没有任何疑问，2和345谁先输出呢？调用resolve之后，立刻调用then吗？还是先把resolve后面的代码执行完毕呢？

结论：调用 then 方法时 Promise 状态为 fulfilled，回调函数 onFulfilled 直接加入 microtask 队列；调用 then 方法时 Promise 状态为 pending，首先将 onFulfilled 加入一个链表，待 Promise 状态变为 fulfilled 后，将链表中的每一项加入 microtask 队列。

因此，输出1后，执行resolve，promise变为fulfilled，但是对应的then还没有执行到，不会输出语句。resolve之后接着往后执行，遇到promise.resolve，把他的then方法加入微任务队列，随后遇到了外层promise的then方法，由于此时promise已经变为fulfilled，因此会把then放入微任务队列中。本轮宏任务执行完毕。开始执行微任务队列，首先取出2执行，输出2，然后取出3执行，输出3，3执行完了之后，又遇到了3的then，接着把then放入微任务队列，执行4，4之后的then也放入微任务队列，执行5

对比参考一下下面的输出：



输出结果：1，2，6，3，4，5

结果：

1.resolve之后并不会立即把then加入微任务队列中，需要执行到then时才会加入微任务队列。

2.执行到then时，并不会立即把then加入微任务队列中。此时then的回调函数在一个链表中存储，等到resolve之后，才会把then的回调加入到微任务队列中

## 6.9 js如何测网速

通过加载图片来测试，加载图片前，记录下系统的时间s1，图片加载完毕后触发onload事件，在该事件中，记录下系统时间s2，再结合图片的大小size，可以计算speed=size/(s2-s1)

# 6.10 实现repeat函数

var str = "Runoob";

str.repeat(2);

//str变为RunoobRunoob

思路：在String.prototype上添加repeat方法，来将this（即调用这个方法的字符串str）赋值给tempstr，然后重复n次即可

## 6.11 split

split() 方法使用指定的分隔符字符串将一个String对象分割成子字符串数组，以一个指定的分割字串来决定每个拆分的位置

## 6.12 微任务MutationObserver

MutationObserver 可以对 DOM 树的更改进行监视，微任务

## 6.13 装饰器

用于注释或者修改类和类的方法，在代码编译阶段执行

比如为类添加一些静态属性

## 6.14 正则表达式是否加g

let reg=/\d/g

let str=”12345”;

g表示全局匹配，区别如下：

加上了g，那么第一次匹配到1，下次匹配从1开始

如果不加g，那么第一次匹配到1，第二次还匹配到1

## 6.15 Array.from

Array.from(arrayLike[, mapFn[, thisArg]])

arrayLike

想要转换成数组的伪数组对象或可迭代对象。

mapFn 可选

如果指定了该参数，新数组中的每个元素会执行该回调函数。

(value，index)=>{ return ...} 注意第一个参数为值，第二个参数为index

thisArg 可选

可选参数，执行回调函数 mapFn 时 this 对象。

返回值

新数组

## 6.16 stopPropagation和preventDefault

stopPropagation：阻止捕获阶段和冒泡阶段中当前事件的进一步传播。但是，它不能防止任何默认行为的发生

preventDefault：阻止事件触发后默认动作的发生。但是不能阻止事件的传播

6.17 Math.Ceil()

向上取整，Math.ceil(6.5)为7