目录

[1 JS篇 1](#_Toc105456724)

[1.1 JS语法基础 1](#_Toc105456725)

[1.1.1 JS的数据类型有哪些 1](#_Toc105456726)

[1.1.2判断数据类型的方法有哪些 1](#_Toc105456727)

[1.1.3 null和undefined的区别 2](#_Toc105456728)

[1.1.4 为什么typeof null 返回object 2](#_Toc105456729)

[1.1.5 为什么0.1+0.2 == 0.3 是false 2](#_Toc105456730)

[1.1.6 NaN 3](#_Toc105456731)

[1.1.7 isNaN 和 Number.isNaN 函数的区别 3](#_Toc105456732)

[1.1.8 == 操作符的强制类型转换规则 3](#_Toc105456733)

[1.1.9其他值到字符串的转换规则 5](#_Toc105456734)

[1.1.10 其他值到数字值的转换规则 5](#_Toc105456735)

[1.1.11 转换为Boolean 5](#_Toc105456736)

[1.1.12 || 和 && 操作符的返回值(不太确定) 6](#_Toc105456737)

[1.1.13 Object.is(a,b) 与比较操作符 `===`、`==`的区别 6](#_Toc105456738)

[1.1.14 什么是 JavaScript 中的包装类型 6](#_Toc105456739)

[1.1.15 JavaScript 中如何进行隐式类型转换 6](#_Toc105456740)

[1.1.16 JS基本和非基本数据类型之间的区别 7](#_Toc105456741)

[1.1.17 JS严格模式 7](#_Toc105456742)

[1.1.18 深拷贝和浅拷贝 7](#_Toc105456743)

[1.1.19 JS变量提升 8](#_Toc105456744)

[1.1.20 let、const、var的区别 8](#_Toc105456745)

[1.1.21 JavaScript为什么要进行变量提升，它导致了什么问题 8](#_Toc105456746)

[1.1.22 const对象的属性可以修改吗 9](#_Toc105456747)

[1.1.23 什么是 DOM 和 BOM 9](#_Toc105456748)

[1.1.24 常用的DOM操作方法 9](#_Toc105456749)

[1.2 JS数组 10](#_Toc105456750)

[1.2.1 数组有哪些原生方法 10](#_Toc105456751)

[1.2.2 数组的reduce方法 11](#_Toc105456752)

[1.2.3 数组扁平化 12](#_Toc105456753)

[1.2.4 清空数组 12](#_Toc105456754)

[1.2.5 遍历数组的方法 12](#_Toc105456755)

[1.2.6 遍历map键值对方法 13](#_Toc105456756)

[1.2.7 遍历的forEach和map方法有什么区别 13](#_Toc105456757)

[1.2.8 判断一个数据是不是数组的方法 13](#_Toc105456758)

[1.2.9 数组去重 13](#_Toc105456759)

[1.3 JS对象 13](#_Toc105456760)

[1.3.1 new操作符的过程 13](#_Toc105456761)

[1.3.2 对象创建的方式有哪些 14](#_Toc105456762)

[1.3.3 对象继承的方式有哪些 14](#_Toc105456763)

[1.3.4 JavaScript有哪些内置对象 15](#_Toc105456764)

[1.3.5 如何判断一个对象是否属于某个类 15](#_Toc105456765)

[1.3.6 Map和Object的区别 （Map是键值对） 15](#_Toc105456766)

[1.3.7 for…in和for…of的区别 16](#_Toc105456767)

[1.4 函数与作用域 16](#_Toc105456768)

[1.4.1 原型、原型链的理解 16](#_Toc105456769)

[1.4.2闭包的理解 16](#_Toc105456770)

[1.4.3 作用域 16](#_Toc105456771)

[1.4.4 作用域链 17](#_Toc105456772)

[1.4.5 console.log 和console.dir的区别 17](#_Toc105456773)

[1.4.6 尾调用 17](#_Toc105456774)

[1.5 JS事件循环与ES6新特性（重点） 17](#_Toc105456775)

[1.5.1 箭头函数与普通函数的区别 17](#_Toc105456776)

[1.5.2 扩展运算符的作用及使用场景 18](#_Toc105456777)

[1.5.3 Proxy 18](#_Toc105456778)

[1.5.4 JSON的理解 18](#_Toc105456779)

[1.5.5 延迟加载的方式有哪些 18](#_Toc105456780)

[1.5.6异步编程的实现方式 18](#_Toc105456781)

[1.5.7 对AJAX的理解 19](#_Toc105456782)

[1.5.8 promise 19](#_Toc105456783)

[1.5.9 promise链式回调 21](#_Toc105456784)

[1.5.10 promise的reject 22](#_Toc105456785)

[1.5.11 promise的catch用法 23](#_Toc105456786)

[1.5.12 promise的all 23](#_Toc105456787)

[1.5.13 promise的race 24](#_Toc105456788)

[1.5.14 js 执行机制、事件循环 24](#_Toc105456789)

[1.5.15 宏任务、微任务 25](#_Toc105456790)

[1.5.16为什么会有宏任务和微任务两种类型的异步任务 25](#_Toc105456791)

[1.5.17 对this对象的理解 25](#_Toc105456792)

[1.5.18 javascript 的垃圾回收机制 26](#_Toc105456793)

[1.5.19 内存泄漏 26](#_Toc105456794)

[1.5.20 let解决for循环 26](#_Toc105456795)

[1.5.21 函数柯里化 27](#_Toc105456796)

[1.5.22 eval 27](#_Toc105456797)

[1.6 面试原题补充（只列举不在上述知识点范围内的） 28](#_Toc105456798)

[1.6.1怎么判断一个对象是不是可迭代的 28](#_Toc105456799)

[1.6.2 如何修改一个数组的副本而不更改原数组 28](#_Toc105456800)

[1.6.3 说说宏任务和微任务的执行顺序 28](#_Toc105456801)

[1.6.4 定时器相同时间 29](#_Toc105456802)

[1.6.5 设置两个一样的定时器事件，会输出什么结果 30](#_Toc105456803)

[1.6.6 JS语法糖 30](#_Toc105456804)

[1.6.7 async和await 30](#_Toc105456805)

[1.6.8 JS为什么要有异步操作 31](#_Toc105456806)

[1.6.9 call、apply、bind的区别 32](#_Toc105456807)

[1.6.10 关于prototype和\_\_proto\_\_ （字节一面原题） 32](#_Toc105456808)

[1.6.11 js函数的节流与防抖 32](#_Toc105456809)

[1.6.12 手写new操作符 32](#_Toc105456810)

[1.6.13 require /exports 和import/export 33](#_Toc105456811)

[1.6.14 ES6新特性整理 33](#_Toc105456812)

[1.6.15 ES6新函数 33](#_Toc105456813)

[1.6.16async和promise输出顺序： 34](#_Toc105456814)

[1.6.17 js的值拷贝和引用拷贝 35](#_Toc105456815)

[1.6.18 js number的最大值 37](#_Toc105456816)

[1.6.19 js基本数据类型占多少个字节 38](#_Toc105456817)

[1.6.20 promise.then什么时候加入微任务队列 38](#_Toc105456818)

[1.6.21 js如何测网速 39](#_Toc105456819)

[1.6.22 实现repeat函数 39](#_Toc105456820)

[1.6.23 split 39](#_Toc105456821)

[1.6.24 微任务MutationObserver 39](#_Toc105456822)

[1.6.25 装饰器 39](#_Toc105456823)

[1.6.26正则表达式是否加g 40](#_Toc105456824)

[1.6.27 Array.from 40](#_Toc105456825)

[1.6.28 stopPropagation和preventDefault 40](#_Toc105456826)

[1.6.29 Math.Ceil() 40](#_Toc105456827)

[2 HTML篇 41](#_Toc105456828)

[2.1HTML语义化 41](#_Toc105456829)

[2.2浏览器乱码的原因是什么 如何解决 42](#_Toc105456830)

[2.3 HTML标准模式和混杂模式的区别 42](#_Toc105456831)

[2.4HTML标准模式和严格模式的区别 43](#_Toc105456832)

[2.5 如何实现浏览器内多个标签页之间的通信 43](#_Toc105456833)

[2.6 前端页面有哪三层构成，分别是什么 作用是什么 43](#_Toc105456834)

[2.7 什么是XML文件 43](#_Toc105456835)

[2.8 XML和JSON的区别 44](#_Toc105456836)

[2.9 移动html5开发流行的框架有哪些 44](#_Toc105456837)

[2.10 嵌入视频 45](#_Toc105456838)

[2.11 浏览器解析HTML全过程 45](#_Toc105456839)

[2.12 src和href的区别 45](#_Toc105456840)

[2.13 defer和async的区别 45](#_Toc105456841)

[2.14 常用的meta标签有哪些 46](#_Toc105456842)

[2.15 img的srcset属性的作用 46](#_Toc105456843)

[2.16 Canvas和SVG的区别 46](#_Toc105456844)

[2.17 meta viewport 是做什么用的 47](#_Toc105456845)

[2.18 **defer\async的区别** 47](#_Toc105456846)

[2.19 getElementById和getElementsByClassName 47](#_Toc105456847)

[2.20 meta viewport 原理是什么 47](#_Toc105456848)

[2.11 懒加载 48](#_Toc105456849)

[2.12预加载 48](#_Toc105456850)

[2.13 页面有大量图片加载很慢，有哪些方法优化图片的加载 49](#_Toc105456851)

[2.14 常见的web优化方式 49](#_Toc105456852)

[2.14.1Web语义化优化 49](#_Toc105456853)

[2.14.2 异步加载js 49](#_Toc105456854)

[2.14.3 CSSDOM优化 49](#_Toc105456855)

[2.15 浏览器的线程 50](#_Toc105456856)

[2.16 获取标签内的文字 50](#_Toc105456857)

# 1 JS篇

## 1.1 JS语法基础

### 1.1.1 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。

### 1.1.2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。

### 1.1.3 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

### 1.1.4 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

### 1.1.5 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。

### 1.1.6 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

### 1.1.7 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。

### 1.1.8 == 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

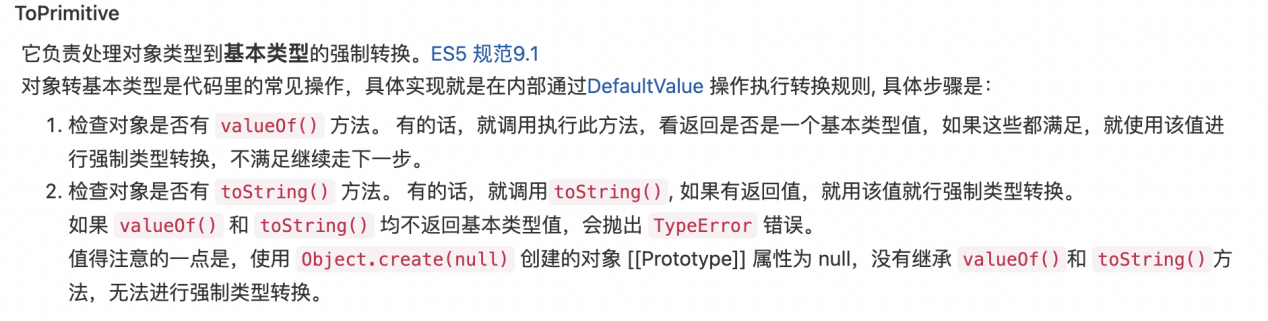
2.类型一致时比较二者的值（无论原始值或引用值），值相同则相等（例外: NaN）

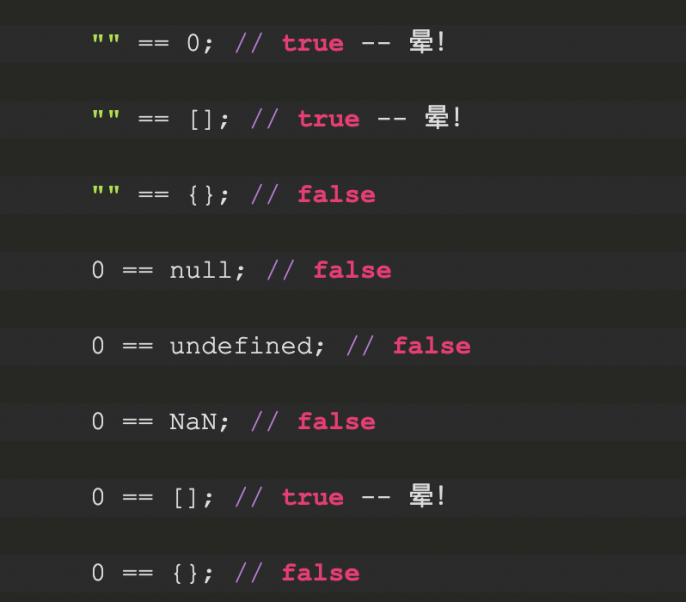
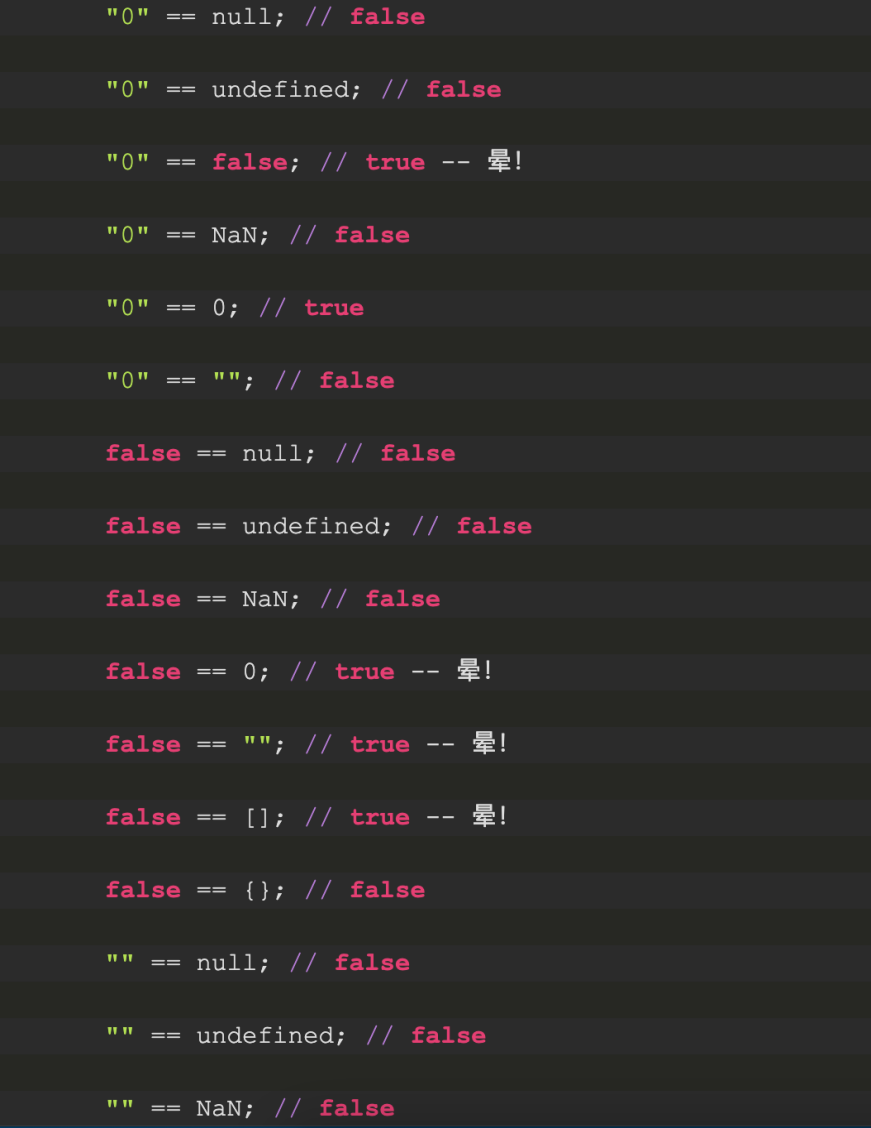
3.对象和基本类型值，会将对象转为基本数据类型

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

### 1.1.9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]

### 1.1.10 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

### 1.1.11 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

### 1.1.12 || 和 && 操作符的返回值(不太确定)

|| 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。

### 1.1.13 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。

### 1.1.14 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。

### 1.1.15 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 ！ && ||

### 1.1.16 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

### 1.1.17 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

### 1.1.18 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

### 1.1.19 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

### 1.1.20 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

### 1.1.21 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

### 1.1.22 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。

### 1.1.23 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

### 1.1.24 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()

## 1.2 JS数组

### 1.2.1 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

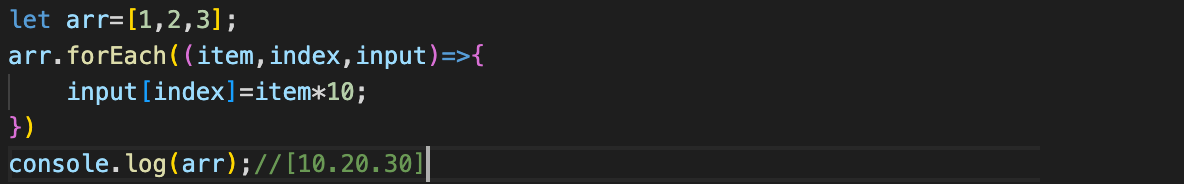
lastIndexOf()：在数组中搜索item，并返回其位置。从末尾开始搜索

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

### 1.2.2 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 16.44.441.2.3 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。

### 1.2.4 清空数组

1.arr.length=0

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

3.arr=[]

### 1.2.5 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

### 1.2.6 遍历map键值对方法

For of

### 1.2.7 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。如果数组都是常量，则无法改变，需要用到第三个参数的写法才能改变。forEach 循环无法中途跳出。

map()不改变原数组的值，返回新数组。

### 1.2.8 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

### 1.2.9 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。

## 1.3 JS对象

### 1.3.1 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

### 1.3.2 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

### 1.3.3 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

### 1.3.4 JavaScript有哪些内置对象



### 1.3.5 如何判断一个对象是否属于某个类

·第一种方式，使用 instanceof 运算符来判断构造函数的 prototype 属性是否出现在对象的原型链中的任何位置

·第二种方式，通过对象的 constructor 属性来判断，对象的 constructor 属性指向该对象的构造函数，但是这种方式不是很安全，因为 constructor 属性可以被改写。

### 1.3.6 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

### 1.3.7 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

## 1.4 函数与作用域

### 1.4.1 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

### 1.4.2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

### 1.4.3 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

### 1.4.4 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。

### 1.4.5 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。

### 1.4.6 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。

## 1.5 JS事件循环与ES6新特性（重点）

### 1.5.1 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

### 1.5.2 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.

通常用于数组的深拷贝。

### 1.5.3 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

### 1.5.4 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

### 1.5.5 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

### 1.5.6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

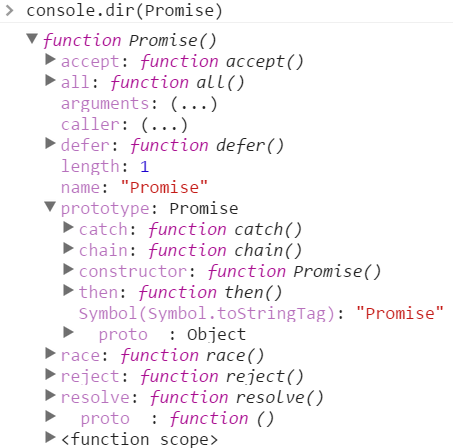
### 1.5.7 对AJAX的理解

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.

### 1.5.8 promise

复杂的概念先不讲，我们先简单粗暴地把Promise用一下，有个直观感受。那么第一个问题来了，Promise是什么玩意呢？是一个类？对象？数组？函数？

别猜了，直接打印出来看看吧，console.dir(Promise)，就这么简单粗暴。



这么一看就明白了，Promise是一个构造函数，自己身上有all、reject、resolve这几个眼熟的方法，原型上有then、catch等同样很眼熟的方法。这么说用Promise new出来的对象肯定就有then、catch方法喽，没错。

那就new一个玩玩吧。

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参数，是函数，并且传入两个参数：resolve，reject，分别表示异步操作执行成功后的回调函数和异步操作执行失败后的回调函数。其实这里用“成功”和“失败”来描述并不准确，按照标准来讲，resolve是将Promise的状态置为fullfiled，reject是将Promise的状态置为rejected。不过在我们开始阶段可以先这么理解，后面再细究概念。

在上面的代码中，我们执行了一个异步操作，也就是setTimeout，2秒后，输出“执行完成”，并且调用resolve方法。

运行代码，会在2秒后输出“执行完成”。注意！我只是new了一个对象，并没有调用它，我们传进去的函数就已经执行了，这是需要注意的一个细节。所以我们用Promise的时候一般是包在一个函数中，在需要的时候去运行这个函数，如：

function runAsync(){

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);

});

return p;

}

runAsync()

这时候你应该有两个疑问：1.包装这么一个函数有毛线用？2.resolve('随便什么数据');这是干毛的？

我们继续来讲。在我们包装好的函数最后，会return出Promise对象，也就是说，执行这个函数我们得到了一个Promise对象。还记得Promise对象上有then、catch方法吧？这就是强大之处了，看下面的代码：

runAsync().then(function(data){

console.log(data);

//后面可以用传过来的数据做些其他操作

//......

});

在runAsync()的返回上直接调用then方法，then接收一个参数，是函数，并且会拿到我们在runAsync中调用resolve时传的的参数。运行这段代码，会在2秒后输出“执行完成”，紧接着输出“随便什么数据”。

这时候你应该有所领悟了，原来then里面的函数就跟我们平时的回调函数一个意思，能够在runAsync这个异步任务执行完成之后被执行。这就是Promise的作用了，简单来讲，就是能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。

你可能会不屑一顾，那么牛逼轰轰的Promise就这点能耐？我把回调函数封装一下，给runAsync传进去不也一样吗，就像这样：

function runAsync(callback){

setTimeout(function(){

console.log('执行完成');

callback('随便什么数据');

}, 2000);

}

runAsync(function(data){

console.log(data);

});

效果也是一样的，还费劲用Promise干嘛。那么问题来了，有多层回调该怎么办？如果callback也是一个异步操作，而且执行完后也需要有相应的回调函数，该怎么办呢？总不能再定义一个callback2，然后给callback传进去吧。而Promise的优势在于，可以在then方法中继续写Promise对象并返回，然后继续调用then来进行回调操作。

### 1.5.9 promise链式回调

所以，从表面上看，Promise只是能够简化层层回调的写法，而实质上，Promise的精髓是“状态”，用维护状态、传递状态的方式来使得回调函数能够及时调用，它比传递callback函数要简单、灵活的多。所以使用Promise的正确场景是这样的：

runAsync1()

.then(function(data){

console.log(data);

return runAsync2();

})

.then(function(data){

console.log(data);

return runAsync3();

})

.then(function(data){

console.log(data);

});

这样能够按顺序，每隔两秒输出每个异步回调中的内容，在runAsync2中传给resolve的数据，能在接下来的then方法中拿到。运行结果如下：

### 1.5.10 promise的reject

到这里，你应该对“Promise是什么玩意”有了最基本的了解。那么我们接着来看看ES6的Promise还有哪些功能。我们光用了resolve，还没用reject呢，它是做什么的呢？事实上，我们前面的例子都是只有“执行成功”的回调，还没有“失败”的情况，reject的作用就是把Promise的状态置为rejected，这样我们在then中就能捕捉到，然后执行“失败”情况的回调。看下面的代码。

function getNumber(){

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

//做一些异步操作

setTimeout(function(){

var num = Math.ceil(Math.random()\*10); //生成1-10的随机数

if(num<=5){

resolve(num);

}

else{

reject('数字太大了');

}

}, 2000);

});

return p;

}

getNumber()

.then(

function(data){

console.log('resolved');

console.log(data);

},

function(reason, data){

console.log('rejected');

console.log(reason);

}

);

getNumber函数用来异步获取一个数字，2秒后执行完成，如果数字小于等于5，我们认为是“成功”了，调用resolve修改Promise的状态。否则我们认为是“失败”了，调用reject并传递一个参数，作为失败的原因。

### 1.5.11 promise的catch用法

在执行resolve的回调（也就是上面then中的第一个参数）时，如果抛出异常了（代码出错了），那么并不会报错卡死js，而是会进到这个catch方法中。

我们知道Promise对象除了then方法，还有一个catch方法，它是做什么用的呢？其实它和then的第二个参数一样，用来指定reject的回调，用法是这样：

getNumber()

.then(function(data){

console.log('resolved');

console.log(data);

})

.catch(function(reason){

console.log('rejected');

console.log(reason);

});

### 1.5.12 promise的all

Promise的all方法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。我们仍旧使用上面定义好的runAsync1、runAsync2、runAsync3这三个函数，看下面的例子。

Promise

.all([runAsync1(), runAsync2(), runAsync3()])

.then(function(results){

console.log(results);

});

用Promise.all来执行，all接收一个数组参数，里面的值最终都算返回Promise对象。这样，三个异步操作的并行执行的，等到它们都执行完后才会进到then里面。那么，三个异步操作返回的数据哪里去了呢？都在then里面呢，all会把所有异步操作的结果放进一个数组中传给then，就是上面的results。

Promise.all中传入的是数组，返回的也是是数组，并且会将进行映射，传入的promise对象返回的值是按照顺序在数组中排列的，但是注意的是他们执行的顺序并不是按照顺序的，除非可迭代对象为空

### 1.5.13 promise的race

顾名思义，Promse.race就是赛跑的意思，意思就是说，Promise.race([p1, p2, p3])里面哪个结果获得的快，就返回那个结果，不管结果本身是成功状态还是失败状态。当要做一件事，超过多长时间就不做了，可以用这个方法来解决。

### 1.5.14 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

### img1.5.15 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)

### 1.5.16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

### 1.5.17 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

### 1.5.18 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

### 1.5.19 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

### 1.5.20 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++){

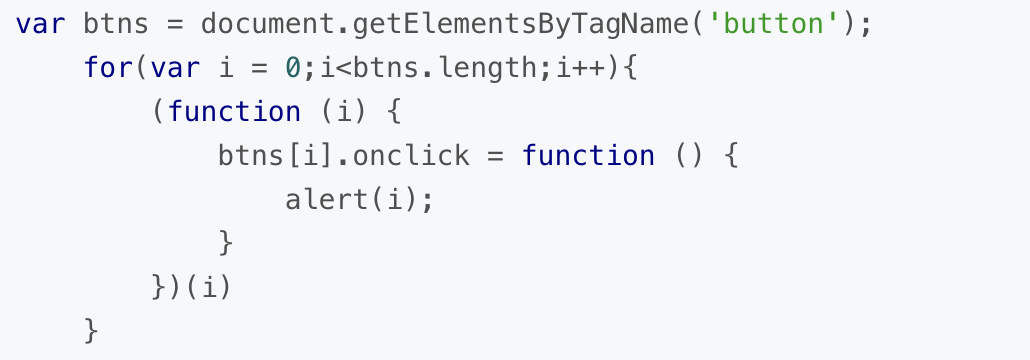
btns[i].onclick = function () {

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

### 1.5.21 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

### 1.5.22 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

## 1.6 面试原题补充（只列举不在上述知识点范围内的）

### 1.6.1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]

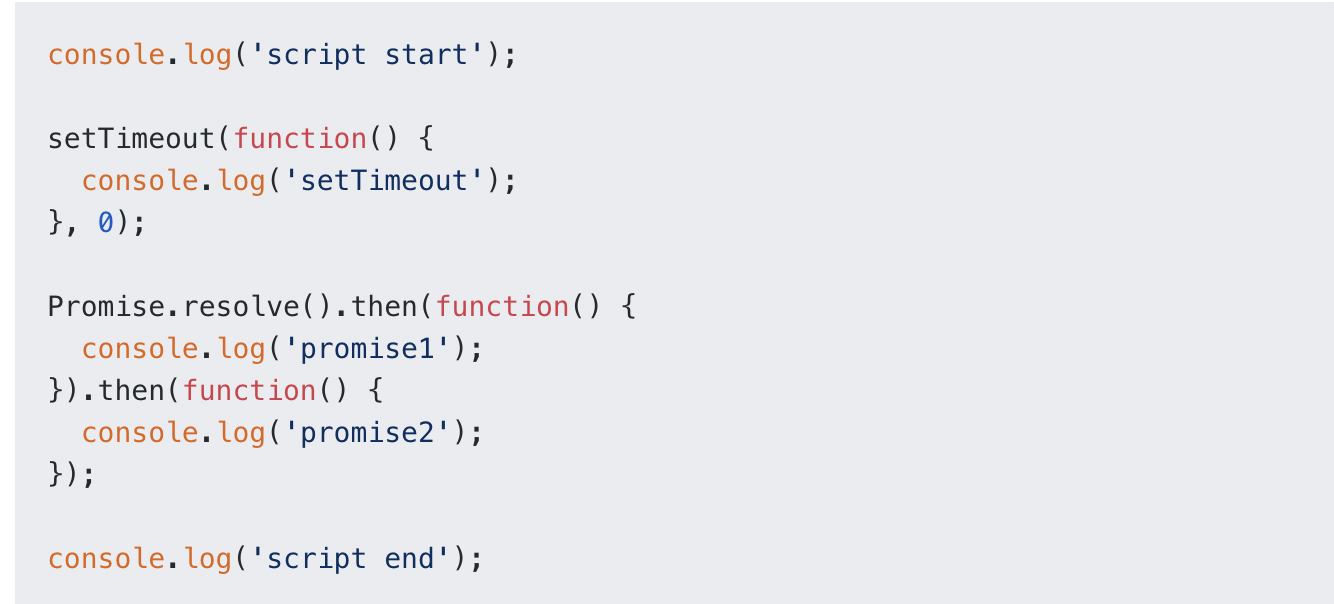
### 1.6.2 如何修改一个数组的副本而不更改原数组

slice、map、concat，filter

### 1.6.3 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。

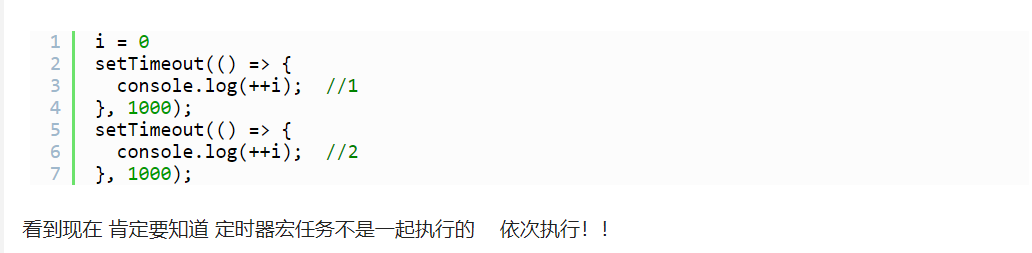
另一个例子：

### 1.6.4 定时器相同时间



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

### 1.6.5 **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

### 1.6.6 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**

### 1.6.7 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**

### 1.6.8 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

### 1.6.9 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

### 1.6.10 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

### 1.6.11 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**

### 1.6.12 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

### 1.6.13 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

### 1.6.14 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

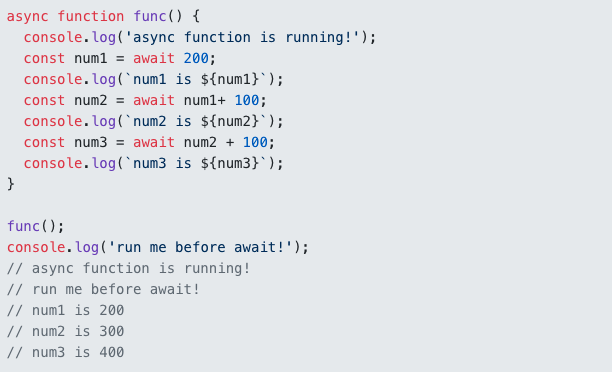
9.async/await

10.module

### 1.6.15 ES6新函数

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

### 1.6.16async和promise输出顺序：

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

输出结果：

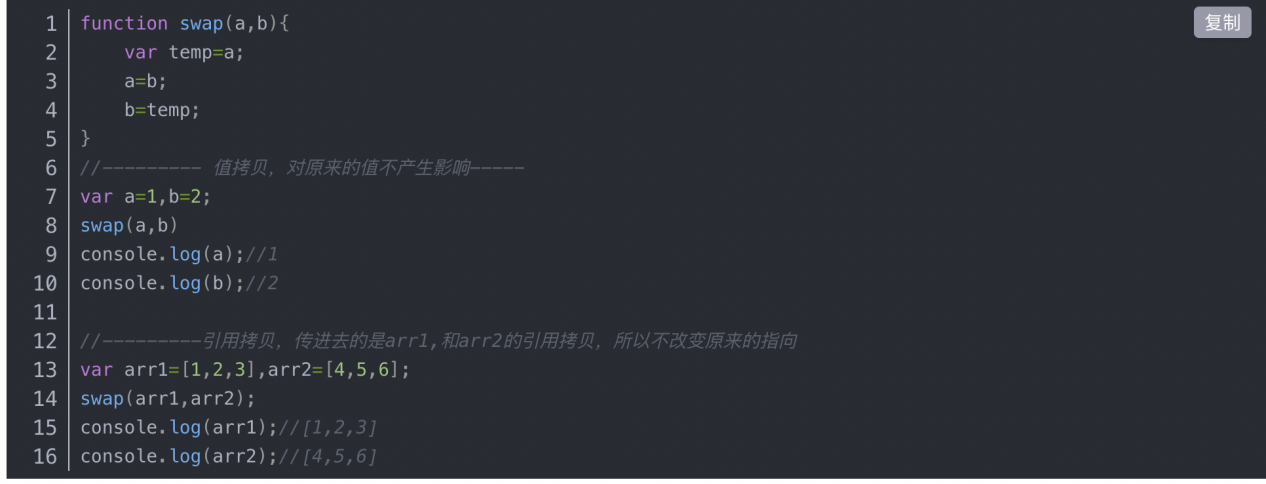


### 1.6.17 js的值拷贝和引用拷贝

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：

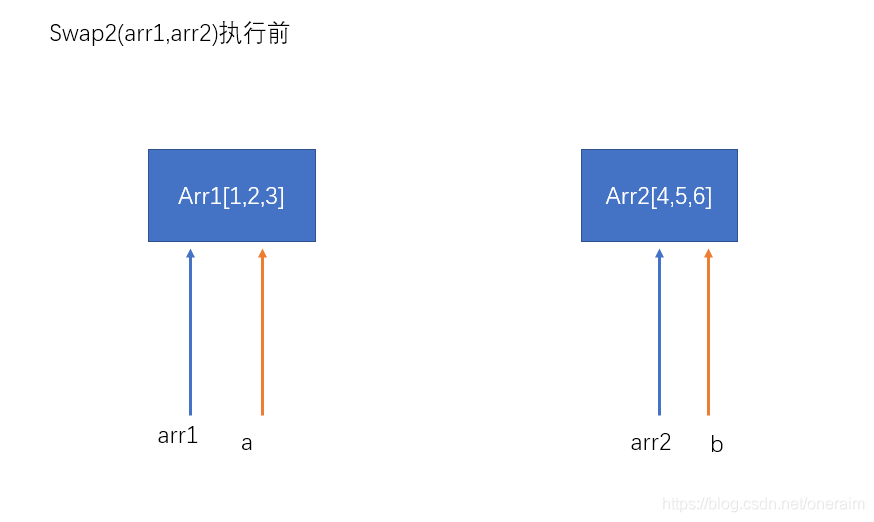
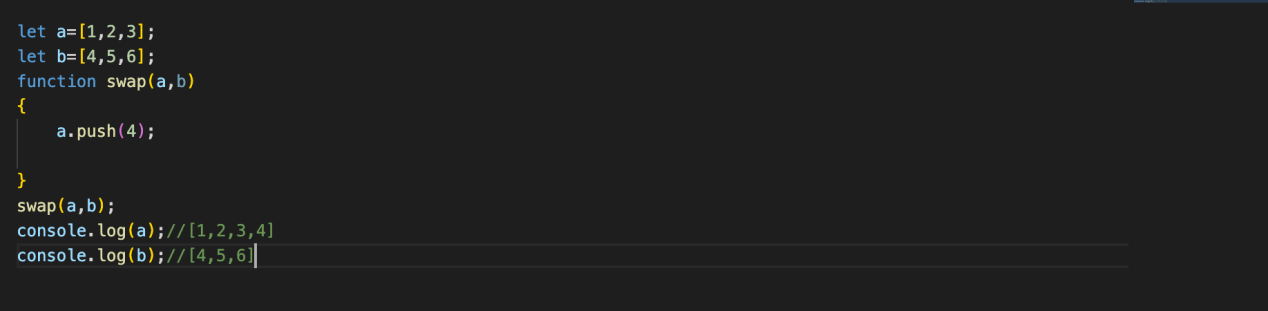
数组arr1和arr2指向内存中的[1,2,3]和[4,5,6]，然后将arr1和arr2传递给a和b，使得a和b指向内存中的[1,2,3]和[4,5,6]，交换a和b，a指向[4,5,6]，b指向[1,2,3]，但是arr1和arr2的指向仍然不变，arr1仍然指向[1,2,3]，arr2仍然指向[4,5,6]。

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

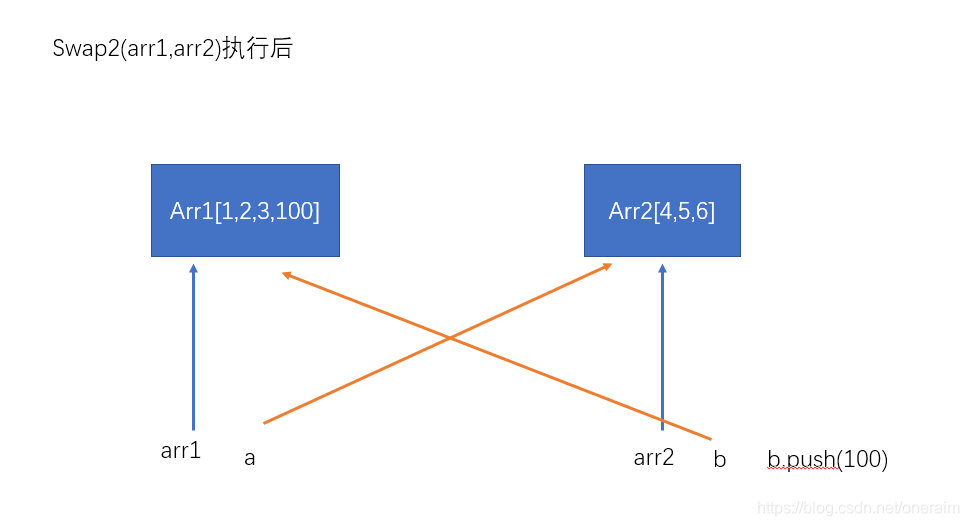
对于基本数据类型来说，a=1传递给了函数swap，实际上是把数字1传递给了swap，swap中的参数a接收到了数字1，此时参数a已经变成了另外一个变量，与外层传入的a没有了任何关系。无论对参数a进行怎么样的改变（交换a和b，令a=3），都永远不会再改变外层的a了，外层的a永远等于1。

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

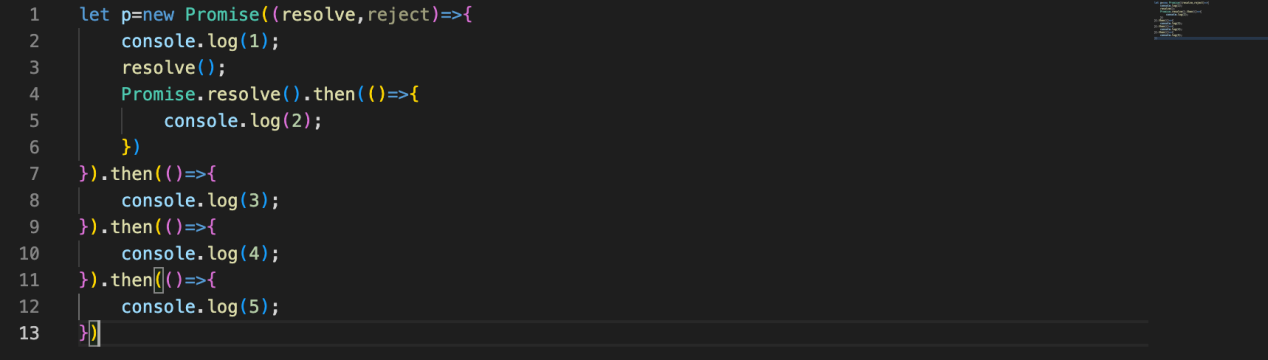
### 1.6.18 js number的最大值

2的53次方,为9007199254740992(16位)

### 1.6.19 js基本数据类型占多少个字节

8个字节

### 1.6.20 promise.then什么时候加入微任务队列



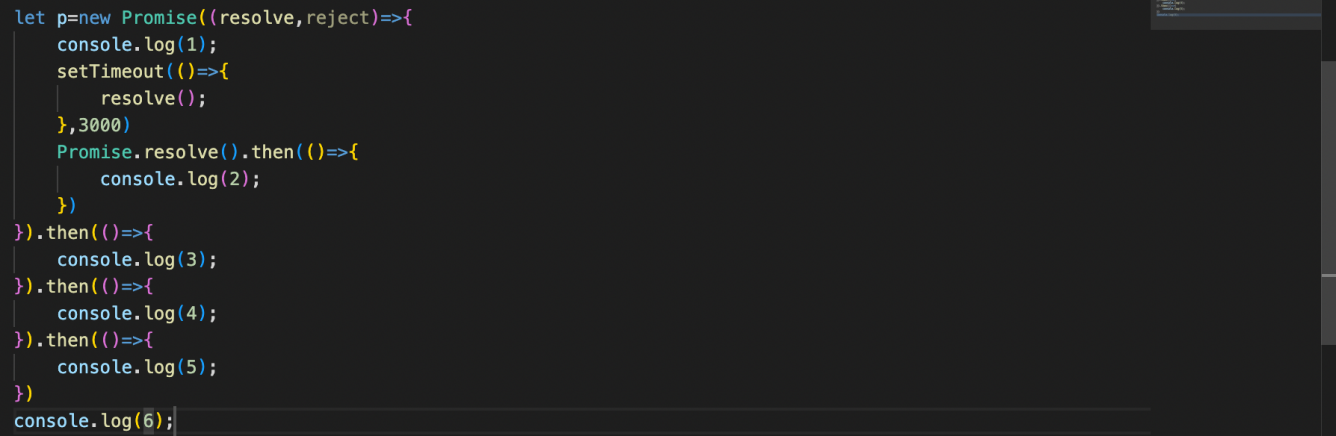
输出：1，2，3，4，5

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

### 1.6.21 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)

### 1.6.22 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

### 1.6.23 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

### 1.6.24 微任务MutationObserver

MutationObserver 可以对 DOM 树的更改进行监视，微任务

### 1.6.25 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

### 1.6.26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

### 1.6.27 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

### 1.6.28 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

### 1.6.29 Math.Ceil()

向上取整，Math.ceil(6.5)为7

# HTML篇

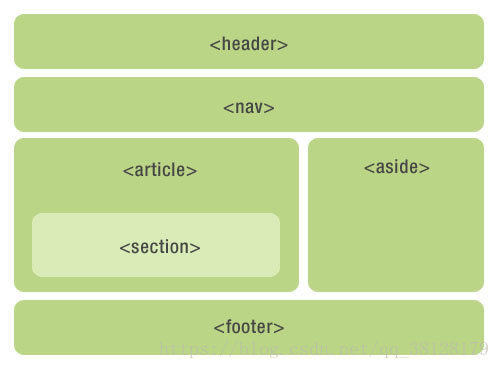
## 2.1HTML语义化

**语义化是指根据内容的结构化（内容语义化），选择合适的标签（代码语义化）**。通俗来讲就是用正确的标签做正确的事情。

语义化的优点如下：

* 对机器友好，带有语义的文字表现力丰富，更适合搜索引擎的爬虫爬取有效信息，有利于SEO。除此之外，语义类还支持读屏软件，根据文章可以自动生成目录；
* 对开发者友好，使用语义类标签增强了可读性，结构更加清晰，开发者能清晰的看出网页的结构，便于团队的开发与维护。

HTML常用的语义化标签如下图所示



**<header>元素**

用于定义页面的介绍展示区域，通常包括网站logo、主导航、全站链接以及搜索框。也适合对页面内部一组介绍性或导航性内容进行标记。

**<nav>元素**

定义页面的导航链接部分区域，不是所有的链接都需要包含在<nav>中，除了页脚再次显示顶级全局导航、或者包含招聘信息等重要链接。

**<main>元素**

定义页面的主要内容，一个页面只能使用一次。如果是web应用，则包围其主要功能。

**<article>元素**

定义页面独立的内容，它可以有自己的header、footer、sections等，专注于单个主题的博客文章，报纸文章或网页文章。article可以嵌套article，只要里面的article与外面的是部分与整体的关系。

**<aside>元素**

定义与主要内容相关的内容块。通常显示为侧边栏。

**<footer>元素**

定义文档的底部区域，通常包含文档的作者，著作权信息，链接的使用条款，联系信息等

## 2.2浏览器乱码的原因是什么 如何解决

原因：网页源代码和网页内容中的文字编码不匹配，例如网页源代码是GBK类型的而文字是utf-8类型的。

原因：网页源代码和请求后端后，从数据库返回的字符编码不匹配，例如网页源码是GBK类型的而数据库是UTF-8类型的。

解决方法：把编码置为统一的类型。

## 2.3 HTML标准模式和混杂模式的区别

**盒模型的处理差异**：标准CSS盒模型的宽度和高度等于内容区的高度和宽度，不包含内边距和边框，而IE6之前的浏览器实现的盒模型的宽高计算方式是包含内边距和边框的。因此，对于IE，怪异模式和标准模式下的盒模型宽高计算方式是不一样的；

**行内元素的垂直对齐**：很多早期的浏览器对齐图片至包含它们的盒子的下边框，虽然CSS的规范要求它们被对齐至盒内文本的基线。标准模式下，基于Gecko的浏览器将会对齐至基线，而在quirks模式下它们会对齐至底部。最直接的例子就是图片的显示。在标准模式下，图片并不是与父元素的下边框对齐的，如果仔细观察，你会发现图片与父元素下边框之间存在一点小空隙。那是因为标准模式下，图片是基线对齐的。而怪异模式下，则不存在这个问题。

## 2.4HTML标准模式和严格模式的区别

**严格模式主要有以下限制：**

* 变量必须声明后再使用
* 函数的参数不能有同名属性，否则报错
* 不能使用with语句
* 不能对只读属性赋值，否则报错
* 不能使用前缀0表示八进制数，否则报错
* 不能删除不可删除的属性，否则报错
* 禁止this指向全局对象

## 2.5 如何实现浏览器内多个标签页之间的通信

**调用cookie+setInterval()**

将要传递的信息存储在cookie中，每隔一定时间读取getCookie获取信息，即可随时获取要传递的信息。

## 2.6 前端页面有哪三层构成，分别是什么 作用是什么

分成：结构层、表示层、行为层。

结构层（structural layer）由 HTML 或 XHTML 之类的标记语言负责创建。标签，也就是那些出现在尖括号里的单词，对网页内容的语义含义做出了描述，但这些标签不包含任何关于如何显示有关内容的信息。例如，P 标签表达了这样一种语义：“这是一个文本段。”

表示层（presentation layer）

由 CSS 负责创建。 CSS 对“如何显示有关内容”的问题做出了回答

行为层（behaviorlayer）

负责回答“内容应该如何对事件做出反应”这一问题。这是 Javascript 语言和 DOM 主宰的领域。

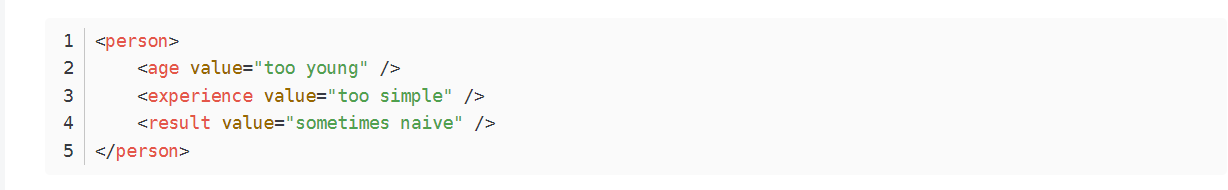
## 2.7 什么是XML文件

在XML语言中，它允许用户自定义标签。一个标签用于描述一段数据；一个标签可以分为开始标签和结束标签，在开始标签和结束标签之间，又可以使用其他标签描述其他数据，以此来实现数据关系的描述。

1.XML的出现解决了程序间数据传输的问题：   
比如QQ之间的数据传送，用XML格式来传送数据，具有良好的可读性，可维护性

2.XML可以做配置文件   
XML文件做配置文件可以说非常普遍，比如我们的Tomcat服务器的server.xml，web.xml。再比如我们的structs中的structs-config.xml文件，和hibernate的hibernate.cfg.xml等等。

3.XML可以充当小型的数据库   
XML文件可以做小型数据库，也是不错的选择，我们程序中可能用到一些经常要人工配置的数据，如果放在数据库中读取不合适（因为这会增加维护数据库的工作），则可以考虑直接用XML来做小型数据库。这种方式直接读取文件显然要比读数据库快。比如msn中保存用户聊天记录就是用XML文件。



上图就是一个典型的xm格式文件

## 2.8 XML和JSON的区别

(1).数据体积方面。

JSON相对于XML来讲，数据的体积小，传递的速度更快些。

(2).数据交互方面。

JSON与JavaScript的交互更加方便，更容易解析处理，更好的数据交互。

(3).数据描述方面。

JSON对数据的描述性比XML较差。

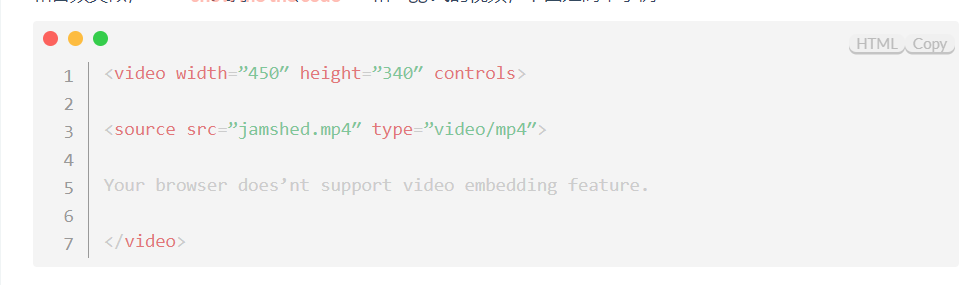
(4).传输速度方面。

JSON的速度要远远快于XML。

## 2.9 移动html5开发流行的框架有哪些

* jQueryMobile，jQueryMobile是jQuery在手机上和平板设备上的版本。jQueryMobile不仅会给主流移动平台带来jQuery核心库，而且会发布一个完整统一的jQuery移动UI框架。支持全球主流的移动平台。jQueryMobile开发团队说：能开发这个项目，非常兴奋。移动web太需要一个跨浏览器的框架，让开发人员开发出真正的移动web网站。
* jQTouch，jQTouch是一个jQuery的插件，主要用于手机上的Webkit浏览器上实现一些包括动画、列表导航、默认应用样式等各种常见UI效果的Java库。支持包括iPhone、Android等手机。

## 2.10 嵌入视频



## 2.11 浏览器解析HTML全过程

DOM文档通常加载的步骤：

1.解析HTML结构。

2.加载外部脚本和样式表文件

3.解析并执行脚本代码。

4.DOM树构建完成。//DOMContentLoaded

5.加载图片等外部资源。

6.页面加载完毕。//load

## 2.12 src和href的区别

src和href都是**用来引用外部的资源**，它们的区别如下：

**src：** 表示对资源的引用，它指向的内容会嵌入到当前标签所在的位置。src会将其指向的资源下载并应⽤到⽂档内，如请求js脚本。当浏览器解析到该元素时，会暂停其他资源的下载和处理，直到将该资源加载、编译、执⾏完毕，所以⼀般js脚本会放在页面底部。

**href：** 表示超文本引用，它指向一些网络资源，建立和当前元素或本文档的链接关系。当浏览器识别到它他指向的⽂件时，就会并行下载资源，不会停⽌对当前⽂档的处理。 常用在a、link等标签上。

## 2.13 defer和async的区别

如果没有defer或async属性，浏览器会立即加载并执行相应的脚本。它不会等待后续加载的文档元素，读取到就会开始加载和执行，这样就阻塞了后续文档的加载。**defer 和 async属性都是去异步加载外部的JS脚本文件，它们都不会阻塞页面的解析**，其区别如下：

* **执行顺序：** 多个带async属性的标签，不能保证加载的顺序；多个带defer属性的标签，按照加载顺序执行；
* **脚本是否并行执行：async属性，表示后续文档的加载和执行与js脚本的加载和执行是并行进行的**，即异步执行；defer属性，加载后续文档的过程和js脚本的加载(此时仅加载不执行)是并行进行的(异步)，js脚本需要等到文档所有元素解析完成之后才执行，DOMContentLoaded事件触发执行之前。

## 2.14 常用的meta标签有哪些

meta 标签由 name 和 content 属性定义，**用来描述网页文档的属性**，比如网页的作者，网页描述，关键词等。

charset，用来描述HTML文档的编码类型：

<meta charset="UTF-8" >

 keywords，页面关键词：

<meta name="keywords" content="关键词" />

description，页面描述：

<meta name="description" content="页面描述内容" />

## 2.15 img的srcset属性的作用

响应式页面中经常用到根据屏幕密度设置不同的图片。这时就用到了 img 标签的srcset属性。srcset属性用于设置不同屏幕密度下，img 会自动加载不同的图片。用法如下：

<img src="image-128.png" srcset="image-256.png 2x" />

## 2.16 Canvas和SVG的区别

**SVG：** SVG可缩放矢量图形（Scalable Vector Graphics）是基于可扩展标记语言XML描述的2D图形的语言，SVG基于XML就意味着SVG DOM中的每个元素都是可用的，可以为某个元素附加Javascript事件处理器。

* 不依赖分辨率，可以动态改变大小，不会失真
* 支持事件处理器
* 最适合带有大型渲染区域的应用程序（比如谷歌地图）
* 复杂度高会减慢渲染速度（任何过度使用 DOM 的应用都不快）
* 不适合游戏应用

**Canvas：** Canvas是画布，通过Javascript来绘制2D图形，是逐像素进行渲染的。其位置发生改变，就会重新进行绘制。

* 依赖分辨率，不能动态改变大小
* 不支持事件处理器
* 弱的文本渲染能力
* 能够以 .png 或 .jpg 格式保存结果图像
* 最适合图像密集型的游戏，其中的许多对象会被频繁重绘

## 2.17 meta viewport 是做什么用的

是为了在移动端不让用户缩放页面使用的

## 2.18 **defer\async的区别**

首先都是让js文件能够异步下载，不阻塞页面的渲染 区别就是defer必须等待整个文档渲染完成后才执行 而async在下载完成后，会暂停html的解析，转去执行js。

## 2.19 getElementById和getElementsByClassName

区别：

Js可以通过ById获取innerText和innerHTML，但是ByClassName不可以

## 2.20 meta viewport 原理是什么

meta viewport 标签的作用是让当前 viewport 的宽度等于设备的宽度，同时不允许用户进行手动缩放。是为了在移动端不让用户缩放页面使用的。

<meta name="viewport" content="width=device-width, initial-scale=1, maximum-scale-1, minimum-scale=1">

* with=device-width 将布局视窗（layout viewport）的宽度设置为设备屏幕分辨率的宽度
* initial-scale=1 页面初始缩放比例为屏幕分辨率的宽度
* maximum-scale=1 指定用户能够放大的最大比例
* minimum-scale=1 指定用户能够缩小的最大比例

## 2.11 懒加载

懒加载也就是延迟加载。

当访问一个页面的时候，先把img元素或是其他元素的背景图片路径替换成一张占位图的路径，这样就只需请求一次，只有当图片出现在浏览器的可视区域内时，才设置图片正真的路径，让图片显示出来。

**优点：很多页面，内容很丰富，页面很长，图片较多。比如瀑布流，图片数量多且大，如果一次性加载完毕，用户等待时间过长。 懒加载页面加载速度快、可以减轻服务器的压力、节约了流量，用户体验好。**

一张图片就是一个<img>标签，浏览器是否发起请求图片是根据<img>的src属性，所以实现懒加载的关键就是，在图片没有进入可视区域时，先不给<img>的src赋值，这样浏览器就不会发送请求了，等到图片进入可视区域再给src赋值。

## 2.12预加载

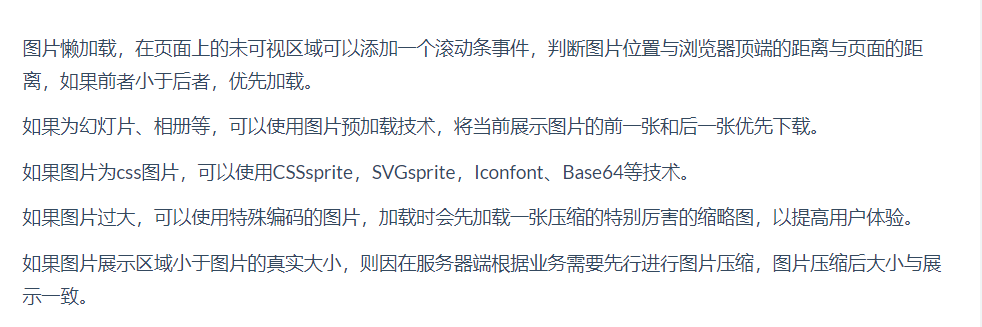
预加载也就是提前加载图片

具体表现为，当用户需要查看时可直接从本地缓存中渲染

一般用于移动端 用于图片比较多的情况 避免在项目中加载缓慢 影响用户体验 所以需要在进入项目前提前加载图片

做法是把所有图片的路径 放在一个数组中 然后使用循环遍历这个数组 每次创建一个图片对象 并把路径赋值给这个图片对象的src属性 这样就完成加载 其中使用 onload事件判断每个图片是否加载完成。

## 2.13 页面有大量图片加载很慢，有哪些方法优化图片的加载



## 2.14 常见的web优化方式

### 2.14.1Web语义化优化

特点：增加代码的可读性，方便网页的爬取，简洁化DOM的标签，减少页面渲染的时间消耗。

语义化标签是浏览器内置就能解析识别的标签，最终的实现也会dom的结构简洁话，所以就会减少时间消耗哦

### 2.14.2 异步加载js

把不需要立刻加载的js异步加载，例如在script标签上增加async标签，实现异步加载。

JS不要写在head里，而是要卸载html的结尾处，因为html解析是自上而下解析的，所以html遇到js会暂停对DOM的解析转而去执行js，会降低页面加载速度。

### 2.14.3 CSSDOM优化

内部样式尽量写在head中（或者说成写在文件尽可能的上方）因为是文档是子上而下的解析，如果写在下方，或者中间，遇到js会阻塞dom和cssom操作，所以避免这个情况，应该规范在head中。

尽量少使用内联样式，原因如上

尽可能的不使用 @import 属性，import属性会在页面加载完成之后，或者使用到该css代码块的时候采取加载对应的文件，等效于把css写到的文档的底部，所以应该避免这种情况。

## 2.15 浏览器的线程

JavaScript 单线程指的是浏览器中负责解释和执行 JavaScript 代码的只有一个线程，即为 JS引擎线程，但是浏览器的渲染进程是提供多个线程的，如下

1.渲染引擎线程：负责页面的渲染

2.JS引擎线程：负责JS的解析和执行

3.定时触发器线程：处理定时事件，比如setTimeout，setInterval

4.事件触发线程：处理DOM响应事件

5.异步http请求线程：处理http请求

三个常驻的线程：渲染线程、js引擎线程、事件触发线程

**渲染线程和js引擎线程不能同时运行，一个运行时，另一个被挂起，GUI更新会被保存在一个队列中等到JS引擎空闲时立即被执行。HTML和CSS的解析是渲染引擎线程，而js的解析则是js引擎线程，二者同一时刻只能有一个在工作**

浏览器端JavaScript是以单线程的方式执行的，也就是说JavaScript和UI渲染占用同一个主线程，那就意味着，如果JavaScript进行高负载的数据处理，UI渲染就很有可能被阻断，浏览器就会出现卡顿，降低了用户体验。

Ajax通过http请求线程得到结果后，将回调函数加入消息队列尾部，等到js引擎空闲，再去执行消息队列中的回调函数，因此，Ajax并不是真正的异步

setTimeout和setInterval也是一样

## 2.16 获取标签内的文字

let d=document.getElementById("a");

console.log(d.innerHTML);