**目录**

[1 JS基本数据类型 3](#_Toc89894551)

[1.1 JS的数据类型有哪些 3](#_Toc89894552)

[1.2判断数据类型的方法有哪些 3](#_Toc89894553)

[1.3 null和undefined的区别 4](#_Toc89894554)

[1.4 为什么typeof null 返回object 5](#_Toc89894555)

[1.5 为什么0.1+0.2 == 0.3 是false 5](#_Toc89894556)

[1.6 typeof NaN 的结果是什么 5](#_Toc89894557)

[1.7 isNaN 和 Number.isNaN 函数的区别 5](#_Toc89894558)

[1.8 == 操作符的强制类型转换规则 5](#_Toc89894559)

[1.9 其他值到字符串的转换规则 6](#_Toc89894560)

[1.10 其他值到数字值的转换规则 6](#_Toc89894561)

[1.11 || 和 && 操作符的返回值 6](#_Toc89894562)

[1.12 Object.is() 与比较操作符 `===`、`==`的区别 7](#_Toc89894563)

[1.13 什么是 JavaScript 中的包装类型 7](#_Toc89894564)

[1.14 JavaScript 中如何进行隐式类型转换 8](#_Toc89894565)

[1.15 JS基本和非基本数据类型之间的区别 9](#_Toc89894566)

[1.16 数组有哪些原生方法 9](#_Toc89894567)

[2 JS变量提升 9](#_Toc89894568)

[2.1 深拷贝和浅拷贝 9](#_Toc89894569)

[2.2 JS变量提升 9](#_Toc89894570)

[2.3 let、const、var的区别 10](#_Toc89894571)

[2.4 JavaScript为什么要进行变量提升，它导致了什么问题 10](#_Toc89894572)

[2.5 const对象的属性可以修改吗 11](#_Toc89894573)

[3 JS对象 11](#_Toc89894574)

[3.1 new操作符的过程 11](#_Toc89894575)

[3.2 对象创建的方式有哪些 12](#_Toc89894576)

[3.3 对象继承的方式有哪些 12](#_Toc89894577)

[3.4 箭头函数与普通函数的区别 13](#_Toc89894578)

[3.5 扩展运算符的作用及使用场景 13](#_Toc89894579)

[3.6 Proxy 可以实现什么功能 13](#_Toc89894580)

[3.7 Map和Object的区别 （Map是键值对） 14](#_Toc89894581)

[3.8 JavaScript有哪些内置对象 15](#_Toc89894582)

[3.9 JSON的理解 15](#_Toc89894583)

[3.10 什么是 DOM 和 BOM 16](#_Toc89894584)

[3.11 常用的DOM操作方法 16](#_Toc89894585)

[3.12 use strict 17](#_Toc89894586)

[3.13 如何判断一个对象是否属于某个类 18](#_Toc89894587)

[4 JS的异步 18](#_Toc89894588)

[4.1 延迟加载的方式有哪些 18](#_Toc89894589)

[4.2 对AJAX的理解，实现一个AJAX请求 18](#_Toc89894590)

[4.3异步编程的实现方式 19](#_Toc89894591)

[4.4 Promise 19](#_Toc89894592)

[4.5 promise链式回调 22](#_Toc89894593)

[4.6 promise的reject 22](#_Toc89894594)

[4.7 promise的catch用法 23](#_Toc89894595)

[4.8 promise的all 24](#_Toc89894596)

[4.9 promise的race 24](#_Toc89894597)

[5 函数的调用和作用域 24](#_Toc89894598)

[5.1 console.log 和console.dir的区别 24](#_Toc89894599)

[5.2 尾调用 25](#_Toc89894600)

[5.3 js 执行机制、事件循环 25](#_Toc89894601)

[5.4 宏任务、微任务 27](#_Toc89894602)

[5.5 for…in和for…of的区别 27](#_Toc89894603)

[5.6 遍历数组的方法 27](#_Toc89894604)

[5.7 遍历map键值对方法 28](#_Toc89894605)

[5.8 遍历的forEach和map方法有什么区别 28](#_Toc89894606)

[5.9 原型、原型链的理解 28](#_Toc89894607)

[5.10 闭包的理解 29](#_Toc89894608)

[5.11 作用域 30](#_Toc89894609)

[5.12 作用域链 31](#_Toc89894610)

[5.13 对this对象的理解 31](#_Toc89894611)

[5.14 call() 和 apply() 的区别 32](#_Toc89894612)

[5.15 javascript 的垃圾回收机制 32](#_Toc89894613)

[5.16 内存泄漏 33](#_Toc89894614)

[6 实战篇 34](#_Toc89894615)

[6.1 字节JS实战篇 34](#_Toc89894616)

[6.1.1怎么判断一个对象是不是可迭代的 34](#_Toc89894617)

[6.1.2 for of 和 for in的区别 34](#_Toc89894618)

[6.1.3 如何修改一个数组的副本而不更改原数组 34](#_Toc89894619)

[6.1.4 说说宏任务和微任务的执行顺序 35](#_Toc89894620)

[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](#_Toc89894621)

[6.1.6 设置两个一样的定时器事件，会输出什么结果 39](#_Toc89894622)

[6.1.7 JS的== 和=== 39](#_Toc89894623)

[6.1.8 JS语法糖 39](#_Toc89894624)

[6.1.9 async和await 41](#_Toc89894625)

[6.1.10 JS为什么要有异步操作 43](#_Toc89894626)

[6.1.11 JS闭包 43](#_Toc89894627)

[6.1.12 this指针深入理解 45](#_Toc89894628)

[6.1.13 JS数据类型及其判断方法 46](#_Toc89894629)

[6.1.14 call、apply、bind的区别 46](#_Toc89894630)

[6.1.15 判断一个数据是不是数组的方法 47](#_Toc89894631)

[6.1.16 数组去重 47](#_Toc89894632)

[6.1.17 console.log和console.dir的区别 47](#_Toc89894633)

[6.1.18 JS的原型和原型链 47](#_Toc89894634)

# 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。其中 Symbol 和 BigInt 是ES6 中新增的数据类型。

·Symbol 代表创建后独一无二且不可变的数据类型，它主要是为了解决可能出现的全局变量冲突的问题。

·BigInt 是一种数字类型的数据，它可以表示任意精度格式的整数，使用 BigInt 可以安全地存储和操作大整数，即使这个数已经超出了 Number 能够表示的安全整数范围。

## 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**可以正确判断对象的类型**，其内部运行机制是判断在其原型链中能否找到该类型的原型。**可以看到，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 运算符可以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。instanceof 运算符用于判断构造函数的 prototype 属性是否出现在对象的原型链中的任何位置。



**·constructor**



## 1.3 null和undefined的区别

首先 Undefined 和 Null 都是基本数据类型，这两个基本数据类型分别都只有一个值，就是 undefined 和 null。

·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 在 JavaScript 中不是一个保留字，这意味着可以使用 undefined 来作为一个变量名，但是这样的做法是非常危险的，它会影响对 undefined 值的判断

·当对这两种类型使用 typeof 进行判断时，Null 类型化会返回 “object”，这是一个历史遗留的问题。当使用双等号对两种类型的值进行比较时会返回 true，使用三个等号时会返回 false。

## 1.4 为什么typeof null 返回object

在 JavaScript 第一个版本中，null就是一个基本数据类型，但是它属于object的类型。

## 1.5 为什么0.1+0.2 == 0.3 是false

计算机是通过二进制的方式存储数据的，所以计算机计算0.1+0.2的时候，实际上是计算的两个数的二进制的和。0.1的二进制是0.0001100110011001100...（1100循环），0.2的二进制是：0.00110011001100...（1100循环）。

但是在 JavaScript 中只有一种数字类型：Number，它的实现遵循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。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。

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

## 1.6 typeof NaN 的结果是什么

NaN 指“不是一个数字”（not a number），NaN 是一个“警戒值”（sentinel value，有特殊用途的常规值），用于指出数字类型中的错误情况，即“执行数学运算没有成功，这是失败后返回的结果。

typeof NaN; // "number"

NaN 是一个特殊值，它和自身不相等，是唯一一个非自反（自反，reflexive，即 x= = =x 不成立）的值。

## 1.7 isNaN 和 Number.isNaN 函数的区别

·函数 isNaN 接收参数后，尝试将这个参数转换为数值，任何不能被转换为数值的的值都会返回 true，因此非数字值传入也会返回 true ，会影响 NaN 的判断。

·函数 Number.isNaN 会首先判断传入参数是否为数字，如果是数字再继续判断是否为 NaN ，不会进行数据类型的转换，这种方法对于 NaN 的判断更为准确。

## 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。

·首先会判断两者类型是否**相同，**相同的话就比较两者的大小；

·类型不相同的话，就会进行类型转换；

·会先判断是否在对比 null 和 undefined，是的话就会返回 true

·判断两者类型是否为 string 和 number，是的话就会将字符串转换为 number

## 其他值到字符串的转换规则

·Null 和 Undefined 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，但是只允许显式强制类型转换，使用隐式强制类型转换会产生错误

## 1.10 其他值到数字值的转换规则

·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 类型的值转换如同使用 Number() 函数进行转换，如果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

## 1.11 || 和 && 操作符的返回值

|| 和 && 首先会对第一个操作数执行条件判断，如果其不是布尔值就先强制转换为布尔类型，然后再执行条件判断。

·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。

var a = 3

var b = 5

var c = a&&b

console.log(c) //打印 5

var a = 3

var b = 5

var c = a||b

console.log(c) //打印3

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

·使用双等号= =进行相等判断时，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。

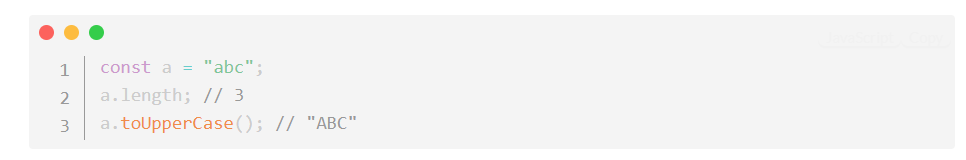
·使用三等号===（严格比较运算符）进行相等判断时，如果两边的类型不一致时，不会做强制类型准换，直接返回 false。

·使用 Object.is 来进行相等判断时，一般情况下和三等号的判断相同，它处理了一些特殊的情况，比如 -0 和 +0 不再相等，两个 NaN 是相等的。

console.log(Object.is(0,-0)) //打印false 0和-0不相同

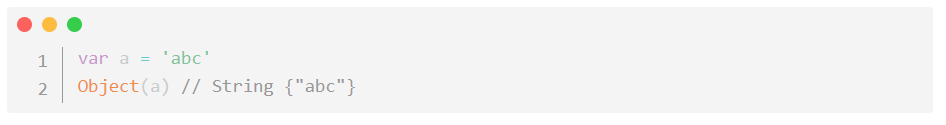
## 1.13 什么是 JavaScript 中的包装类型

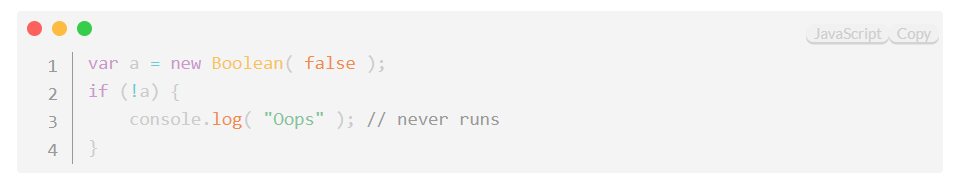
在 JavaScript 中，基本类型是没有属性和方法的，但是为了便于操作基本类型的值，在调用基本类型的属性或方法时 JavaScript 会在后台隐式地将基本类型的值转换为对象。



在访问'abc'.length时，JavaScript 将'abc'在后台转换成String('abc')，然后再访问其length属性。

JavaScript也可以使用Object函数显式地将基本类型转换为包装类型。





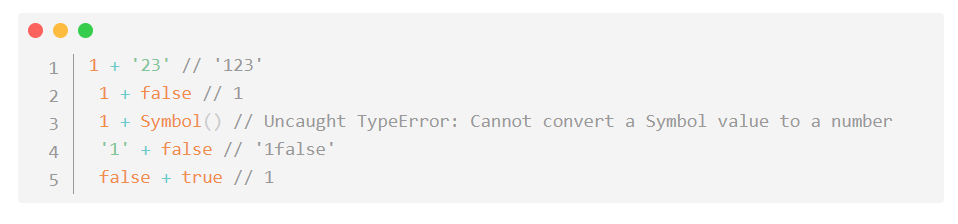
答案是什么都不会打印，因为虽然包裹的基本类型是false，但是false被包裹成包装类型后就成了对象，所以其非值为false，所以循环体中的内容不会运行。

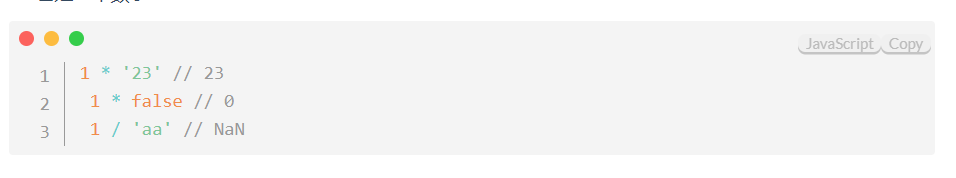
## 1.14 JavaScript 中如何进行隐式类型转换

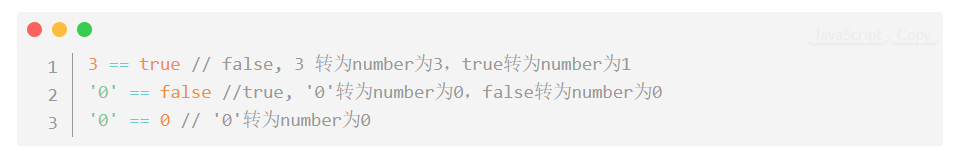
首先要介绍ToPrimitive方法，这是 JavaScript 中每个值隐含的自带的方法，用来将值 （无论是基本类型值还是对象）转换为基本类型值。如果值为基本类型，则直接返回值本身。

+ 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数字。



\* 运算符号

== 运算符号 操作数两边都转换成number

## 1.15 JS基本和非基本数据类型之间的区别

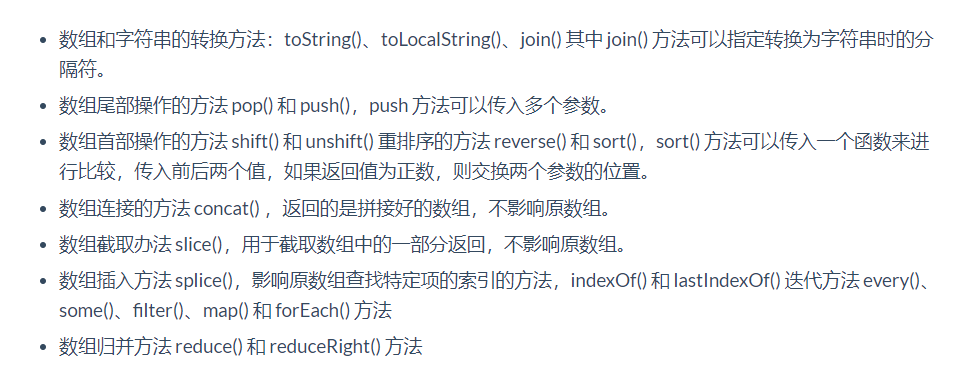
·.目前JS中有6种基本数据类型: Undefined、Null、Boolean、Number、Symbol 和 String。还有1种复杂的数据类型————Object，Object本质上是由一组无序的名值对组成的。Object、Array和Function则属于引用类型。

·基本数据类型是不可变的，而非基本数据类型是可变的。

·基本数据类型是不可变的，因为它们一旦创建就无法更改，但非基本数据类型刚可更改，意味着一旦创建了对象，就可以更改它。

·将基本数据类型与其值进行比较，这意味着如果两个值具有相同的数据类型并具有相同的值，那么它们是严格相等的。

## 1.16 数组有哪些原生方法



# 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

## 2.2 JS变量提升

提升简单来说就是把我们所写的类似于var a = 123;这样的代码，声明提升到它所在作用域的顶端去执行，到我们代码所在的位置来赋值。

function test () {

console.log(a); //输出undefined

var a = 123;

};

a = 1;

var a;

console.log(a); //输出1

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

·块级作用域： 块作用域由 { }包括，let和const具有块级作用域，var不存在块级作用域。块级作用域解决了ES5中的两个问题

· var存在变量提升，let和const不存在变量提升，即在变量只能在声明之后使用，否在会报错

· var声明变量时，可以重复声明变量，后声明的同名变量会覆盖之前声明的遍历。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

变量提升的表现是，无论在函数中何处位置声明的变量，好像都被提升到了函数的首部，可以在变量声明前访问到而不会报错。

造成变量声明提升的**本质原因**是 js 引擎在代码执行前有一个解析的过程，创建了执行上下文，初始化了一些代码执行时需要用到的对象。当访问一个变量时，会到当前执行上下文中的作用域链中去查找，而作用域链的首端指向的是当前执行上下文的变量对象，这个变量对象是执行上下文的一个属性，它包含了函数的形参、所有的函数和变量声明，这个对象的是在代码解析的时候创建的。

JS在拿到一个变量或者一个函数的时候，会有两步操作，即解析和执行。

·在解析阶段，JS会检查语法，并对函数进行预编译。解析的时候会先创建一个全局执行上下文环境，先把代码中即将执行的变量、函数声明都拿出来，变量先赋值为undefined，函数先声明好可使用。

**变量提升的原因**

**提高性能** 在JS代码执行之前，会进行语法检查和预编译，并且这一操作只进行一次。这么做就是为了提高性能，如果没有这一步，那么每次执行代码前都必须重新解析一遍该变量（函数），而这是没有必要的，因为变量（函数）的代码并不会改变，解析一遍就够了。在解析的过程中，还会为函数生成预编译代码。在预编译时，会统计声明了哪些变量、创建了哪些函数，并对函数的代码进行压缩，去除注释、不必要的空白等。

**容错性更好**

防止一些代码报错

## 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（数值、字符串、布尔值），其值就保存在变量指向的那个内存地址，因此等同于常量。

但对于引用类型的数据（主要是对象和数组）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。

# 3 JS对象

## 3.1 new操作符的过程

·创建一个对象

·将构造函数的作用域赋给新对象（也就是将对象的**proto**属性指向构造函数的prototype属性）

·指向构造函数中的代码，构造函数中的this指向该对象（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

js和一般的面向对象的语言不同，在 ES6 之前它没有类的概念。但是可以使用函数来进行模拟，从而产生出可复用的对象创建方式，常见的有以下几种

**第一种是工厂模式**。

工厂模式的主要工作原理是用函数来封装创建对象的细节，从而通过调用函数来达到复用的目的。但是它有一个很大的问题就是创建出来的对象无法和某个类型联系起来，它只是简单的封装了复用代码，而没有建立起对象和类型间的关系。

**第二种是构造函数模式**。

js 中每一个函数都可以作为构造函数，只要一个函数是通过 new 来调用的，那么就可以把它称为构造函数。执行构造函数首先会创建一个对象，然后将对象的原型指向构造函数的 prototype 属性，然后将执行上下文中的 this 指向这个对象，最后再执行整个函数，如果返回值不是对象，则返回新建的对象。因为 this 的值指向了新建的对象，因此可以使用 this 给对象赋值。构造函数模式相对于工厂模式的优点是，所创建的对象和构造函数建立起了联系，因此可以通过原型来识别对象的类型。但是构造函数存在一个缺点就是造成了不必要的函数对象的创建，因为在 js 中函数也是一个对象，因此如果对象属性中如果包含函数的话，那么每次都会新建一个函数对象，浪费了不必要的内存空间，因为函数是所有的实例都可以通用的。

**第三种模式是原型模式**

因为每一个函数都有一个 prototype 属性，这个属性是一个对象，它包含了通过构造函数创建的所有实例都能共享的属性和方法。因此可以使用原型对象来添加公用属性和方法，从而实现代码的复用。这种方式相对于构造函数模式来说，解决了函数对象的复用问题。但是这种模式也存在一些问题，一个是没有办法通过传入参数来初始化值，另一个是如果存在一个引用类型如 Array 这样的值，那么所有的实例将共享一个对象，一个实例对引用类型值的改变会影响所有的实例

## 3.3 对象继承的方式有哪些

第一种是以原型链的方式来实现继承，但是这种实现方式存在的缺点是，在包含有引用类型的数据时，会被所有的实例对象所共享，容易造成修改的混乱。还有就是在创建子类型的时候不能向超类型传递参数。

## 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**

箭头函数不会创建自己的this， 所以它没有自己的this，它只会在自己作用域的上一层继承this。所以箭头函数中this的指向在它在定义时已经确定了，之后不会改变。

·**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window对象。



对象obj的方法b是使用箭头函数定义的，这个函数中的this就永远指向它定义时所处的全局执行环境中的this，即便这个函数是作为对象obj的方法调用，this依旧指向Window对象。需要注意，定义对象的大括号{}是无法形成一个单独的执行环境的，它依旧是处于全局执行环境中。

## 3.5 扩展运算符的作用及使用场景

·**对象扩展运算符**

对象的扩展运算符(…)用于取出参数对象中的所有可遍历属性，拷贝到当前对象之中。

**let bar = { a: 1, b: 2 };**

**let baz = { ...bar }; // { a: 1, b: 2 }**

## 3.6 Proxy 可以实现什么功能

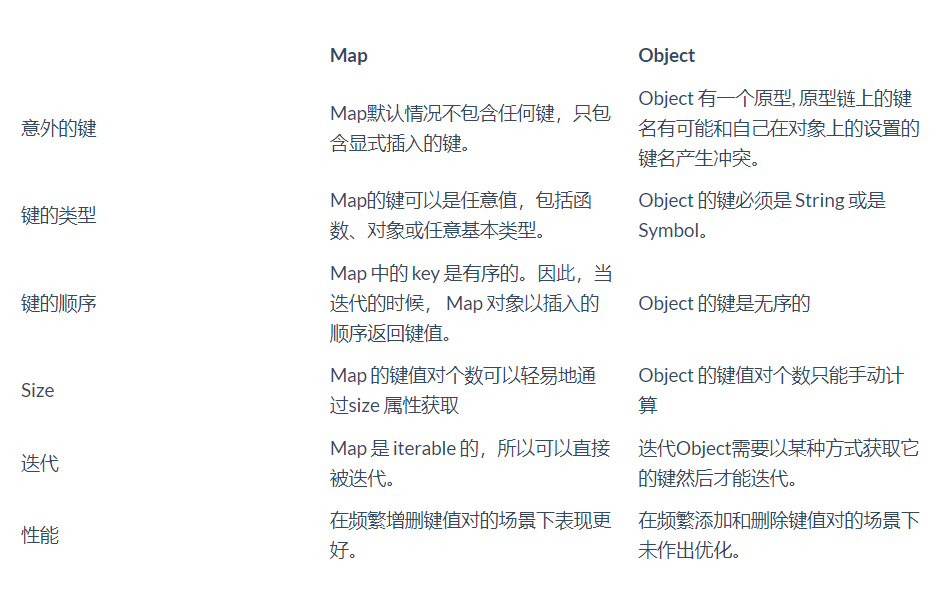
Proxy 是 ES6 中新增的功能，它可以用来自定义对象中的操作

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

target 代表需要添加代理的对象，handler 用来自定义对象中的操作，比如可以用来自定义 set 或者 get 函数。

下面来通过 Proxy 来实现一个数据响应式：

## 3.7 Map和Object的区别 （Map是键值对）

Map可直接通过 var I in map 遍历

## 3.8 JavaScript有哪些内置对象



## 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。它可以被任何的编程语言读取和作为数据格式来传递。

·在项目开发中，使用 JSON 作为前后端数据交换的方式。在前端通过将一个符合 JSON 格式的数据结构序列化为 JSON 字符串，然后将它传递到后端，后端通过 JSON 格式的字符串解析后生成对应的数据结构，以此来实现前后端数据的一个传递。

·因为 JSON 的语法是基于 js 的，因此很容易将 JSON 和 js 中的对象弄混，但是应该注意的是 JSON 和 js 中的对象不是一回事，JSON 中对象格式更加严格，比如说在 JSON 中属性值不能为函数，不能出现 NaN 这样的属性值等，因此大多数的 js 对象是不符合 JSON 对象的格式的。

·SON.stringify 函数，通过传入一个符合 JSON 格式的数据结构，将其转换为一个 JSON 字符串。如果传入的数据结构不符合 JSON 格式，那么在序列化的时候会对这些值进行对应的特殊处理，使其符合规范。在前端向后端发送数据时，可以调用这个函数将数据对象转化为 JSON 格式的字符串。

·JSON.parse() 函数，这个函数用来将 JSON 格式的字符串转换为一个 js 数据结构，如果传入的字符串不是标准的 JSON 格式的字符串的话，将会抛出错误。当从后端接收到 JSON 格式的字符串时，可以通过这个方法来将其解析为一个 js 数据结构，以此来进行数据的访问。

## 3.10 什么是 DOM 和 BOM

DOM 指的是文档对象模型，它指的是把文档当做一个对象，这个对象主要定义了处理网页内容的方法和接口。

BOM 指的是浏览器对象模型，它指的是把浏览器当做一个对象来对待，这个对象主要定义了与浏览器进行交互的法和接口。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。

window 对象含有 location 对象、navigator 对象、screen 对象等子对象，并且 DOM 的最根本的对象 document 对象也是 BOM 的 window 对象的子对象

## 3.11 常用的DOM操作方法

**DOM 节点的获取**

**DOM 节点的创建**



**DOM 节点的删除**



**修改 DOM 元素**



## 3.12 use strict

use strict 是一种 ECMAscript5 添加的（严格模式）运行模式，这种模式使得 Javascript 在更严格的条件下运行。设立严格模式的目的如下。

·消除 Javascript 语法的不合理、不严谨之处，减少怪异行为

·消除代码运行的不安全之处，保证代码运行的安全

·提高编译器效率，增加运行速度

·禁止使用 with 语句

·禁止 this 关键字指向全局对象

## 3.13 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。

# 4 JS的异步

## 4.1 延迟加载的方式有哪些

延迟加载就是等页面加载完成之后再加载 JavaScript 文件。 js 延迟加载有助于提高页面加载速度。

·**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，这样的话就能使页面的渲染不被阻塞。多个设置了 defer 属性的脚本按规范来说最后是顺序执行的，但是在一些浏览器中可能不是这样。

·**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，但是当脚本加载完成后立即执行 js 脚本，这个时候如果文档没有解析完成的话同样会阻塞。多个 async 属性的脚本的执行顺序是不可预测的，一般不会按照代码的顺序依次执行。通常和await配合使用，await只能在async函数体内使用，就是等待 async执行完，才会执行后面的东西， 等待的东西是异步的，它就会阻塞当前代码。

·**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，实现一个AJAX请求

AJAX是 Asynchronous JavaScript and XML 的缩写，指的是通过 JavaScript 的异步通信，从服务器获取 XML 文档从中提取数据，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

JavaScript中的异步机制可以分为以下几种

**回调函数** 的方式，使用回调函数的方式有一个缺点是，多个回调函数嵌套的时候会造成回调函数地狱，上下两层的回调函数间的代码耦合度太高，不利于代码的可维护。（简单点说，一个函数被作为参数传递给另一个函数就是回调函数）

**Promise** 的方式，使用 Promise 的方式可以将嵌套的回调函数作为链式调用。但是使用这种方法，有时会造成多个 then 的链式调用，可能会造成代码的语义不够明确。

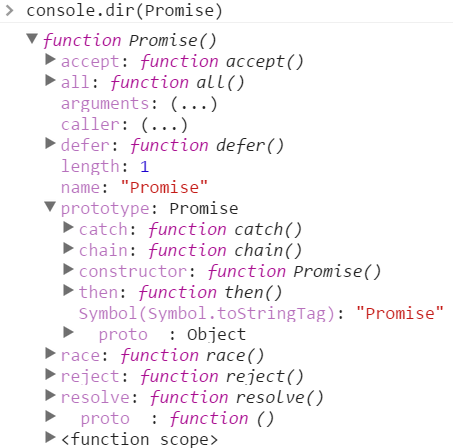
**generator** 的方式，它可以在函数的执行过程中，将函数的执行权转移出去，在函数外部还可以将执行权转移回来。当遇到异步函数执行的时候，将函数执行权转移出去，当异步函数执行完毕时再将执行权给转移回来。因此在 generator 内部对于异步操作的方式，可以以同步的顺序来书写。使用这种方式需要考虑的问题是何时将函数的控制权转移回来，因此需要有一个自动执行 generator 的机制，比如说 co 模块等方式来实现 generator 的自动执行。

**async 函数** 的方式，async 函数是 generator 和 promise 实现的一个自动执行的语法糖，它内部自带执行器，当函数内部执行到一个 await 语句的时候，如果语句返回一个 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。因此可以将异步逻辑，转化为同步的顺序来书写，并且这个函数可以自动执行。

## 4.4 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。

## 4.5 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：

## 4.6 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。

## 4.7 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);

});

## 4.8 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

## 4.9 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

# 5 函数的调用和作用域

## 5.1 console.log 和console.dir的区别

·console.log()可以取代alert()或document.write()，在网页脚本中使用console.log()时，会在浏览器控制台打印出信息。

·console.dir()可以显示一个对象所有的属性和方法。

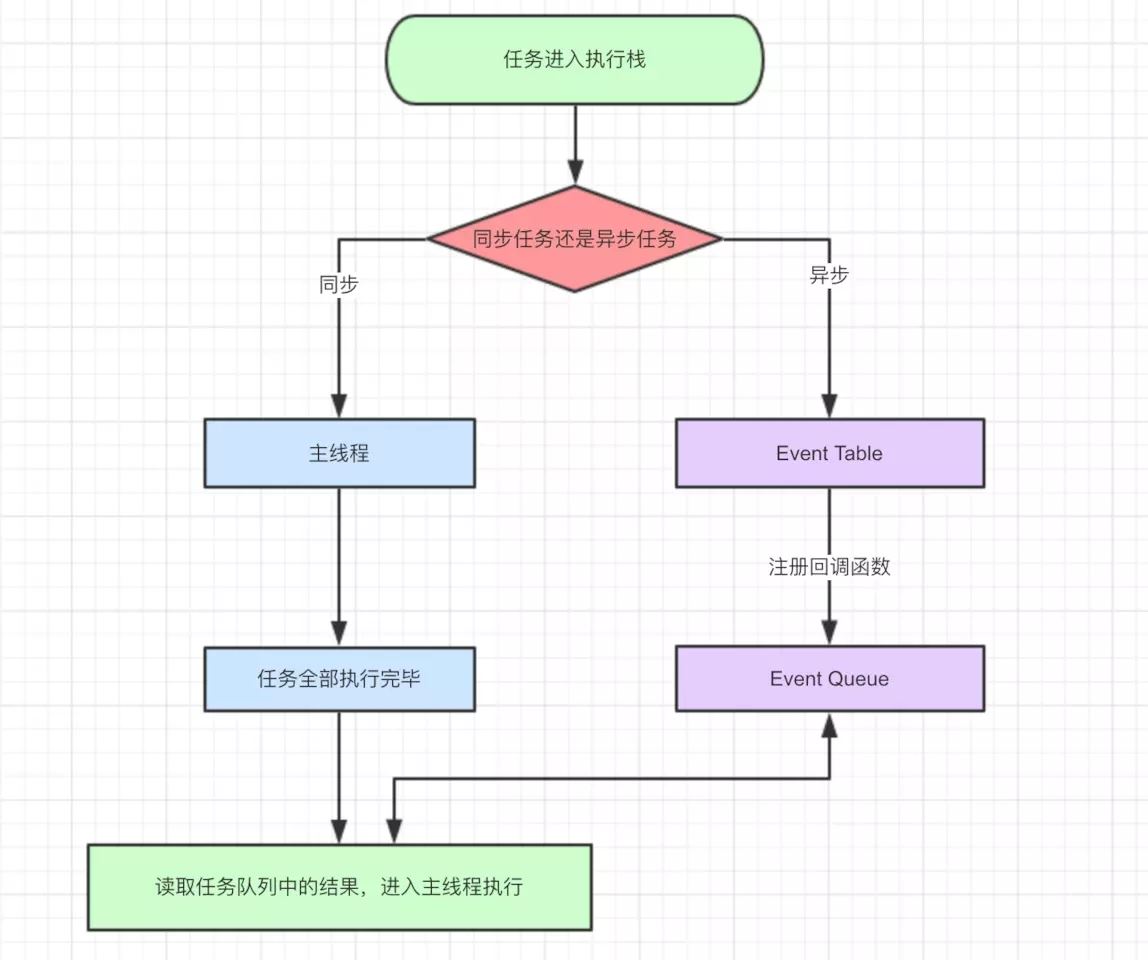
## 5.2 尾调用

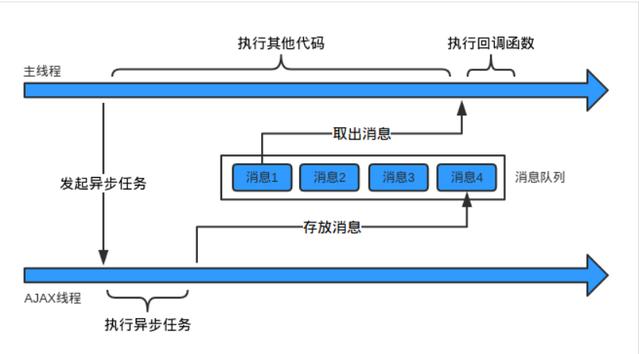
尾调用指的是函数的最后一步调用另一个函数。

代码执行是基于执行栈的，所以当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加入栈中。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，这就是尾调用优化。但是 ES6 的尾调用优化只在严格模式开启，正常模式无效。

## 5.3 js 执行机制、事件循环

JavaScript 语言的一大特点就是单线程，同一个时间只能做一件事。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。JavaScript 语言的设计者意识到这个问题，将所有任务分成两种，一种是**同步任务（synchronous），另一种是异步任务（asynchronous）**在所有同步任务执行完之前，任何的异步任务是不会执行的。



同步和异步任务分别进入不同的执行”场所”，同步的进入主线程，异步的进入 Event Table 并注册函数。当**指定的事情完成时**，Event Table 会将这个函数移入 Event Queue。主线程内的任务执行完毕为空，会去 Event Queue 读取对应的函数，进入主线程执行。上述过程会不断重复，也就是常说的 Event Loop(事件循环)。

我们不禁要问了，那怎么知道主线程执行栈为空啊？js 引擎存在 monitoring process 进程，会持续不断的检查主线程执行栈是否为空，一旦为空，就会去 Event Queue 那里检查是否有等待被调用的函数。换一张图片也许更好理解主线程的执行过程。

## 5.4 宏任务、微任务

**JS 异步有一个机制，就是遇到宏任务，先执行宏任务，将宏任务放入 Event Queue，然后再执行微任务，将微任务放入 Event Queue，但是，这两个 Queue 不是一个 Queue。当你往外拿的时候先从微任务里拿这个回调函数，然后再从宏任务的 Queue 拿宏任务的回调函数。**

而宏任务一般是：包括整体代码script，setTimeout，setInterval。

微任务：Promise.then（必须执行到resolve后才将then()放入微任务队列），process.nextTick。

## 5.5 for…in和for…of的区别

for…of 遍历获取的是对象的键值，for…in 获取的是对象的键名

for…in 循环主要是为了遍历对象而生，不适用于遍历数组；for…of 循环可以用来遍历数组、类数组对象，字符串、Set、Map 以及 Generator 对象

## 5.6 遍历数组的方法

**·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

**for(j = 0; j < arr.length; j++) { }**

**·改进版for循环**

使用临时变量，将长度缓存起来，避免重复获取数组长度，当数组较大时优化效果才会比较明显。性能最高！！！

**for(j = 0,len=arr.length; j < len; j++) { }**

**·弱化版for循环**

这种方法其实严格上也属于for循环，只不过是没有使用length判断，而使用变量本身判断

**for(j = 0; arr[j]!=null; j++) {}**

**·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

**arr.forEach(function(e){ });**

**·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(j in arr) { }**

**·map遍历**

这种方式也是用的比较广泛的，虽然用起来比较优雅，但实际效率还比不上foreach。

**arr.map(function(n){ });**

**·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

**for(let value of arr) { });**

## 5.7 遍历map键值对方法

console.log(jsonData);*//火狐控制台打印输出： Object { fileNumber="文件编号", fileName="文件名称"}*

for(var key in jsonData)

console.log("属性：" + key + ",值："+ jsonData[key]);

}

## 5.8 遍历的forEach和map方法有什么区别

这方法都是用来遍历数组的，两者区别如下

**·**forEach()方法会针对每一个元素执行提供的函数，对数据的操作会改变原数组，该方法没有返回值。forEach 循环无法中途跳出，break 命令或 return 命令都不能奏效。

**·**map()方法不会改变原数组的值，返回一个新数组，新数组中的值为原数组调用函数处理之后的值

## 5.9 原型、原型链的理解

在JavaScript中是使用构造函数来新建一个对象的，每一个构造函数的内部都有一个 prototype 属性，它的属性值是一个对象，这个对象包含了可以由该构造函数的所有实例共享的属性和方法。

当使用构造函数新建一个对象后，在这个对象的内部将包含一个指针，这个指针指向构造函数的 prototype 属性对应的值，在 ES5 中这个指针被称为对象的原型。

一般来说不应该能够获取到这个值的，但是现在浏览器中都实现了 **proto** 属性来访问这个属性，但是最好不要使用这个属性，因为它不是规范中规定的。ES5 中新增了一个 Object.getPrototypeOf() 方法，可以通过这个方法来获取对象的原型。

当访问一个对象的属性时，如果这个对象内部不存在这个属性，那么它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，于是就这样一直找下去，也就是原型链的概念。原型链的尽头一般来说都是 Object.prototype 所以这就是新建的对象为什么能够使用 toString() 等方法的原因。

## 5.10 闭包的理解

**闭包是指有权访问另一个函数作用域中变量的函数**，创建闭包的最常见的方式就是在一个函数内创建另一个函数，创建的函数可以访问到当前函数的局部变量。

闭包有两个常用的用途：

**-**闭包的第一个用途是使我们在函数外部能够访问到函数内部的变量。通过使用闭包，可以通过在外部调用闭包函数，从而在外部访问到函数内部的变量，可以使用这种方法来创建私有变量。

-闭包的另一个用途是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，因为闭包函数保留了这个变量对象的引用，所以这个变量对象不会被回收。

比如，函数 A 内部有一个函数 B，函数 B 可以访问到函数 A 中的变量，那么函数 B 就是闭包。



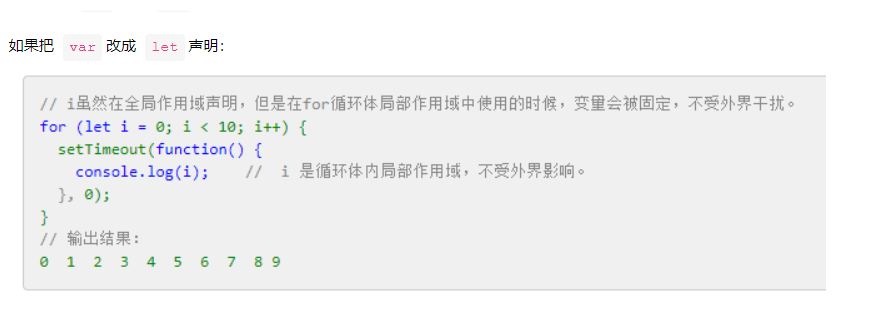
在 JS 中，闭包存在的意义就是让我们可以间接访问函数内部的变量。经典面试题：循环中使用闭包解决 var 定义函数的问题。首先因为 setTimeout 是个异步函数，所以会先把循环全部执行完毕，这时候 i 就是 6 了，所以会输出一堆 6。解决办法有三种：

第一种使用闭包的方式

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

第二种就是使用 setTimeout 的第三个参数，这个参数会被当成 timer 函数的参数传入

第三种就是使用 let 定义 i 了来解决问题了，这个也是最为推荐的方式



## 5.11 作用域

**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

·所有未定义直接赋值的变量自动声明为全局作用域

·所有window对象的属性拥有全局作用域

·全局作用域有很大的弊端，过多的全局作用域变量会污染全局命名空间，容易引起命名冲突

**函数作用域**

·函数作用域声明在函数内部的变零，一般只有固定的代码片段可以访问到

·作用域是分层的，内层作用域可以访问外层作用域，反之不行

**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）

·let和const声明的变量不会有变量提升，也不可以重复声明

·在循环中比较适合绑定块级作用域，这样就可以把声明的计数器变量限制在循环内部

## 5.12 作用域链

 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，那这个变量就是自由变量。如果在自己作用域找不到该变量就去父级作用域查找，依次向上级作用域查找，直到访问到window对象就被终止，这一层层的关系就是作用域链。

作用域链的作用是**保证对执行环境有权访问的所有变量和函数的有序访问，通过作用域链，可以访问到外层环境的变量和函数。**

作用域链的本质上是一个指向变量对象的指针列表。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。作用域链的前端始终都是当前执行上下文的变量对象。全局执行上下文的变量对象（也就是全局对象）始终是作用域链的最后一个对象。

## 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。在实际开发中，this 的指向可以通过四种调用模式来判断。

第一种是**函数调用模式**，当一个函数不是一个对象的属性时，直接作为函数来调用时，this 指向全局对象。

第二种是**方法调用模式**，如果一个函数作为一个对象的方法来调用时，this 指向这个对象。

第三种是**构造器调用模式**，如果一个函数用 new 调用时，函数执行前会新创建一个对象，this 指向这个新创建的对象。

第四种是 **apply 、 call 和 bind 调用模式**，这三个方法都可以显示的指定调用函数的 this 指向。其中 apply 方法接收两个参数：一个是 this 绑定的对象，一个是参数数组。call 方法接收的参数，第一个是 this 绑定的对象，后面的其余参数是传入函数执行的参数。也就是说，在使用 call() 方法时，传递给函数的参数必须逐个列举出来。bind 方法通过传入一个对象，返回一个 this 绑定了传入对象的新函数。这个函数的 this 指向除了使用 new 时会被改变，其他情况下都不会改变。

第五种是**内部函数模式**，嵌套函数中的this不会继承上层函数的this，如果需要，可以用一个变量保存上层函数的this。 Var that = this

## 5.14 call() 和 apply() 的区别

它们的作用一模一样，区别仅在于传入参数的形式的不同。

·apply 接受两个参数，第一个参数指定了函数体内 this 对象的指向，第二个参数为一个带下标的集合，这个集合可以为数组，也可以为类数组，apply 方法把这个集合中的元素作为参数传递给被调用的函数。

·call 传入的参数数量不固定，跟 apply 相同的是，第一个参数也是代表函数体内的 this 指向，从第二个参数开始往后，每个参数被依次传入函数。

## 5.15 javascript 的垃圾回收机制

内存泄露是每个开发者最终都要面对的问题，它是许多问题的根源：反应迟缓，崩溃，高延迟，以及其他应用问题。

垃圾：指一块被分配的内存既不能使用，又不能回收，直到浏览器进程结束。

像 C 这样的编程语言，具有低级内存管理原语，如 malloc()和 free()。开发人员使用这些原语显式地对操作系统的内存进行分配和释放。

而 JavaScript 在创建对象(对象、字符串等)时会为它们分配内存，不再使用对时会“自动”释放内存，这个过程称为垃圾收集。JavaScript 是一种垃圾回收语言。垃圾回收语言通过周期性地检查先前分配的内存是否可达，帮助开发者管理内存。换言之，垃圾回收语言减轻了“内存仍可用”及“内存仍可达”的问题。两者的区别是微妙而重要的：仅有开发者了解哪些内存在将来仍会使用，而不可达内存通过算法确定和标记，适时被操作系统回收。

**内存生命周期中的每一个阶段**

**分配内存** —  内存是由操作系统分配的，它允许您的程序使用它。在低级语言(例如 C 语言)中，这是一个开发人员需要自己处理的显式执行的操作。然而，在高级语言中，系统会自动为你分配内存。

**使用内存** — 这是程序实际使用之前分配的内存，在代码中使用分配的变量时，就会发生读和写操作。  
**释放内存** — 释放所有不再使用的内存,使之成为自由内存,并可以被重利用。与分配内存操作一样,这一操作在低级语言中也是需要显式地执行。

## 5.16 内存泄漏

·全局变量 不用 var 声明的变量，相当于挂载到 window 对象上。如：b=1; 解决：使用严格模式，即开启use strict。例如下图



·被遗忘的定时器和回调函数。



·闭包

·没有清理的 DOM 元素引

# 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

能使用for循环遍历的就是可迭代对象

### 6.1.2 for of 和 for in的区别

### 6.1.3 如何修改一个数组的副本而不更改原数组

使用slice()方法

var arr = [1, 2, 3, "up"];

var duplicate = arr;

duplicate[3] = "down";

console.log(arr[3]); //"down"

console.log(duplicate[3]); //"down"

如果直接修改，会导致原数组也被修改

var arr = [1, 2, 3, "up"];

var duplicate = arr.slice(0);

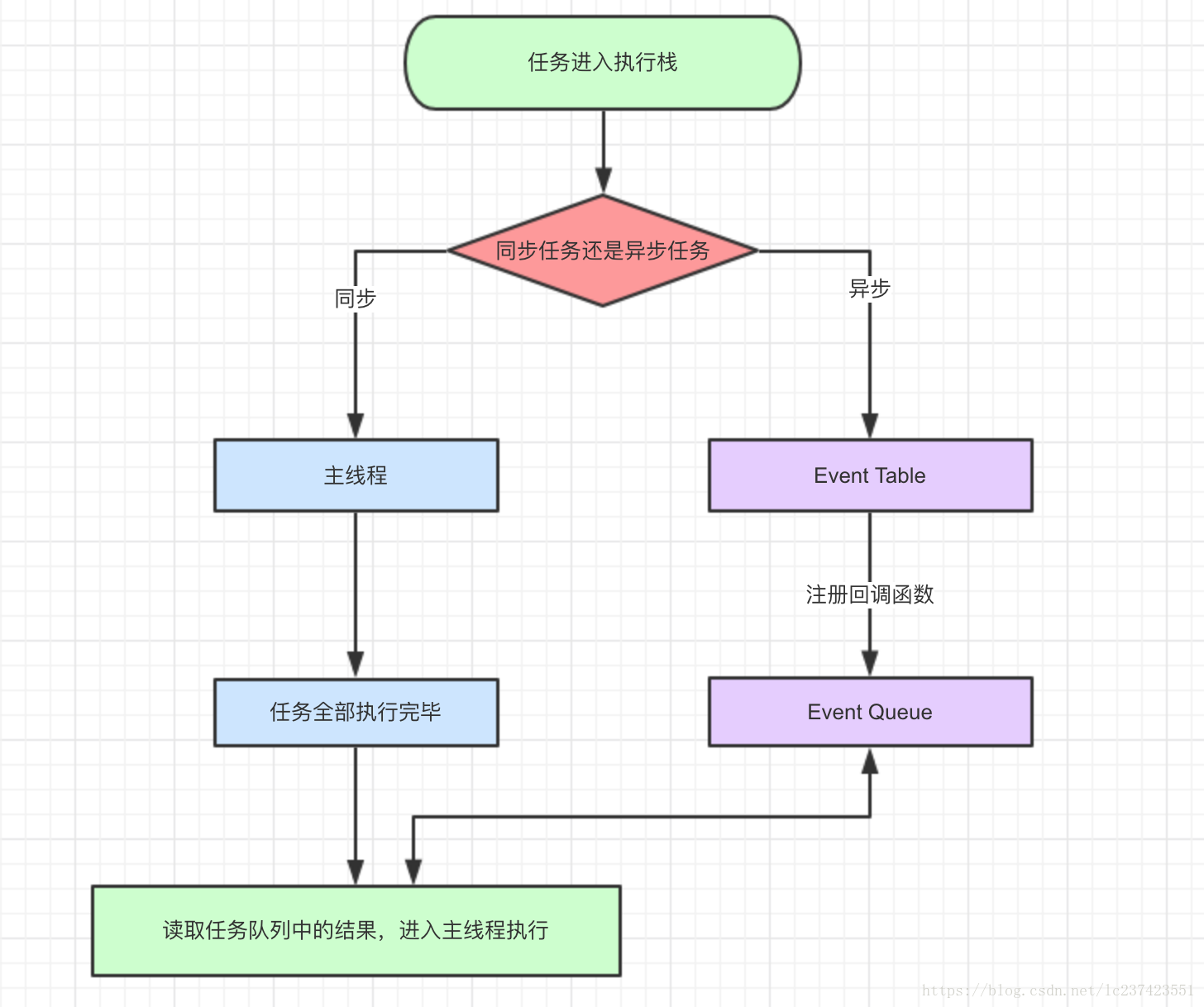
duplicate[3] = "down";

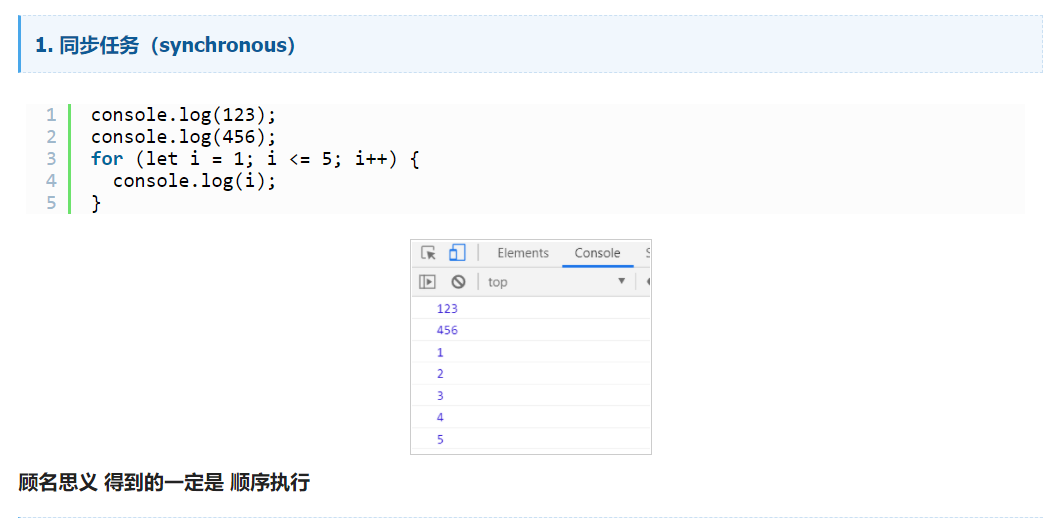
console.log(arr[3]); //"up"

console.log(duplicate[3]); //"down"

需要通过slice克隆，slice(start,end)，如果end省略则默认为数组结尾，上述的sliec(0)即代表克隆了整个数组。

### 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！



通过上面代码知道setTimeout是异步的，我们就搞清了执行顺序优先级 **同步代码>异步代码 。所以说在任务队列中分为两大类 1.同步任务   2. 异步任务**

（1）所有同步任务都在主线程上执行，形成一个执行栈。

（2）主线程之外，还存在一个"任务队列"。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，称为事件循环(Event Loop)。

**简单了解后，我们再来深入了解异步任务中的宏任务和微任务。**

**宏任务中的方法：**

1. 整个script (可以理解为外层同步代码，作为入口 )

2. setTimeout/setInterval

**微任务中的方法：**

1.Promise 2. process.nextTick（类似node.js版的"setTimeout"），但是它属于微任务，serTimeout属于宏任务。



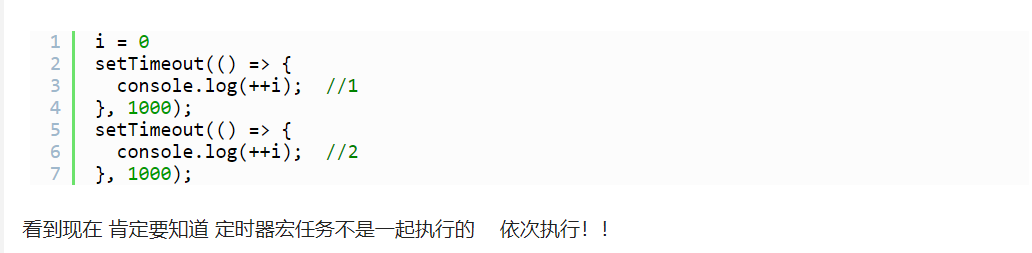
**在执行一个 Promise 对象的时候，当走完resolve();之后，就会立刻把 .then()里面的代码加入到微任务队列当中**

**这是一道经典的面试题**

### 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？**



### 6.1.6 **设置两个一样的定时器事件，会输出什么结果**



### 6.1.7 JS的== 和===

== 先转换后比较 ，===直接比较，除了不转换操作数全等和相等没有区别。

var a = ("123" == 123);

alert(a);//结果为true

var a = ("123" === 123);

alert(a)//结果为false

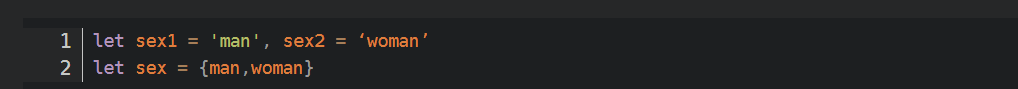
var b = ("123" !== 123);

alert(b)//结果为true

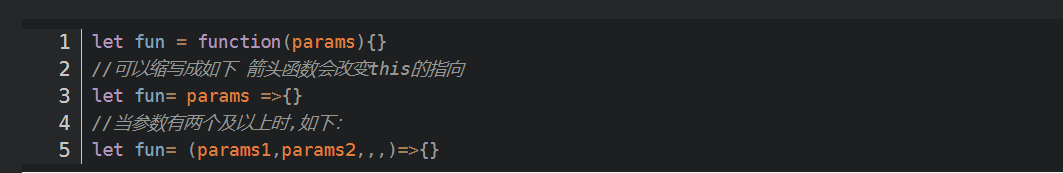
### 6.1.8 JS语法糖

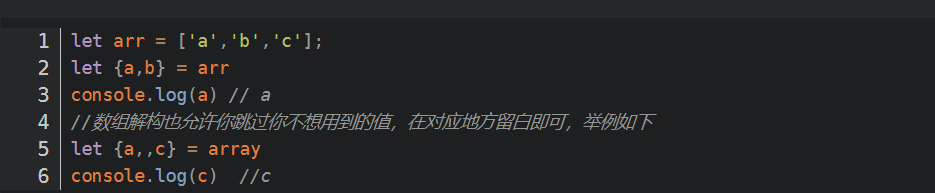
语法糖（Syntactic sugar），也译为糖衣语法。指计算机语言中添加的某种语法，这种语法对语言的功能并没有影响，但是更方便程序员使用。

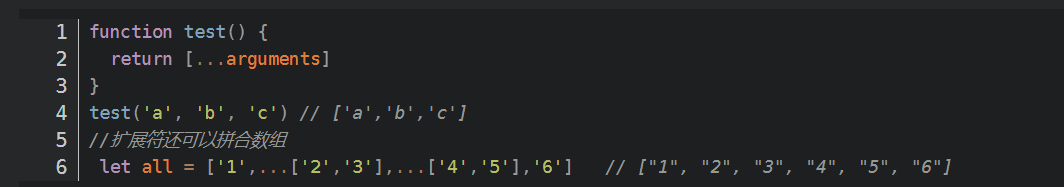
通常来说使用语法糖能够增加程序的可读性，从而减少程序代码出错的机会。

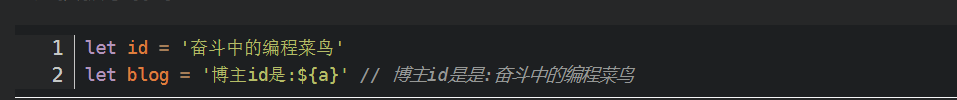
**·对象字面量**

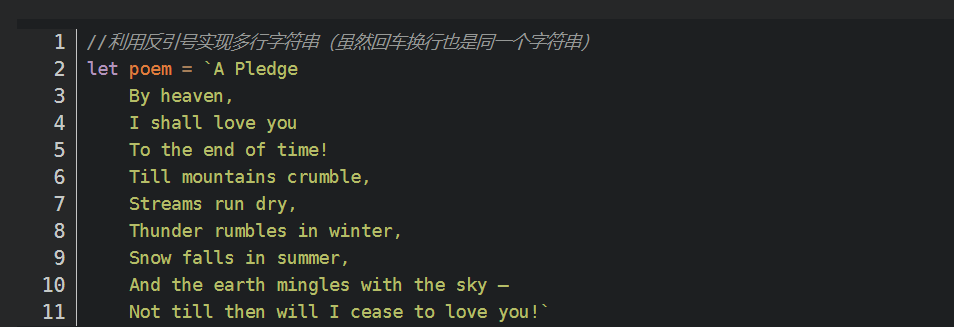
**·箭头函数**

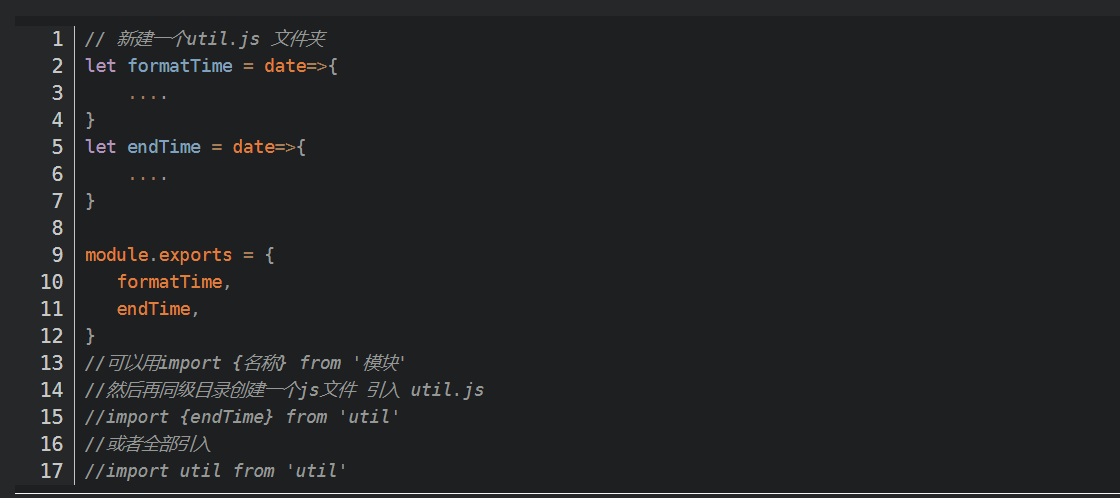


**·数组解构**

**·拓展运算符**

**·模板字符串**

**·多行文本**

**·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，在函数内部使用 await 来表示异步。

async 作为一个关键字放到函数的前面，用于表示函数是一个异步函数，该函数的执行不会阻塞后面代码的执行。

await是等待，只能放到async函数里面，在后面放一个返回promise对象的表达式。

async和await是为了解决大量复杂不易读的Promise异步的问题。

·**async 作为一个关键字放到函数的前面，用于表示函数是一个异步函数，该函数的执行不会阻塞后面代码的执行。**

**·async声明的函数的返回本质上是一个Promise，async函数内部会返回一个Promise对象，then方法回调函数的参数**

**·await的本质是可以提供等同于”同步效果“的等待异步返回能力的语法糖，用await声明的Promise异步返回，必须“等待”到有返回值的时候，代码才继续执行下去**



**async 函数内部的实现原理是resolved，如果函数内部抛出错误，则会导致返回的 Promise 对象状态变为 reject 状态，promise 对象有一个catch 方法进行捕获，被 catch 方法回调函数接收到**



**·async必须声明的是一个function，await就必须是在这个async声明的函数内部使用，必须是直系，作用域链不能隔代，在后面放一个返回promise对象的表达式**



**·函数的错误处理，当 async 函数中只要一个 await 出现 reject 状态，则后面的 await 都不会被执行,可以添加 try/catch**



使用 async/await 需要注意什么？

1）await 命令后面的Promise对象，运行结果可能是 rejected，此时等同于 async 函数返回的 Promise 对象被reject。因此需要加上错误处理，可以给每个 await 后的 Promise 增加 catch 方法；也可以将 await 的代码放在 try…catch 中。

2）多个await命令后面的异步操作，如果不存在继发关系最好让它们同时触发

### 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。

但是遇到等待 ( 网络请求、定时任务 ) 就会卡住，卡住的时候CPU就是空闲的，这种等待让人不甘心。

异步不会阻塞代码执行，页面就不会卡住，所以使用异步，异步采用回调callback的形式。

### 6.1.11 JS闭包

**「函数」和「函数内部能访问到的变量」的总和，就是一个闭包。**

闭包的本质就是在一个函数内部创建另一个函数。

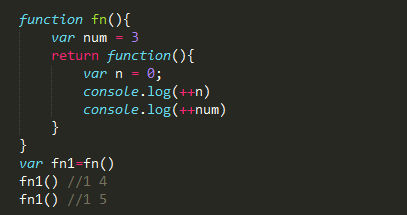
**闭包有3个特性：**

①函数嵌套函数

②函数内部可以引用函数外部的参数和变量

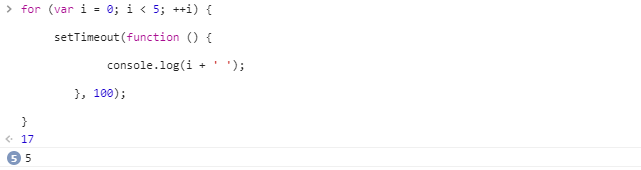
③参数和变量不会被垃圾回收机制回收

**一个闭包的经典例子：**



一般情况下，在函数fn执行完后，就应该连同它里面的变量一同被销毁，但是在这个例子中，匿名函数作为fn的返回值被赋值给了fn1，这时候相当于fn1=function(){var n = 0 ... }，并且匿名函数内部引用着fn里的变量num，所以变量num无法被销毁，而变量n是每次被调用时新创建的，所以每次fn1执行完后它就把属于自己的变量连同自己一起销毁，于是乎最后就剩下孤零零的num，于是这里就产生了内存消耗的问题。

**下一个经典例子**



按照预期它应该依次输出1 2 3 4 5，而结果它输出了五次5，这是为什么呢？原来由于js是单线程的，所以在执行for循环的时候定时器setTimeout被安排到任务队列中排队等待执行，而在等待过程中for循环就已经在执行，等到setTimeout可以执行的时候，for循环已经结束，i的值也已经编程5，所以打印出来五个5。正确的做法是引入闭包来实现



### 6.1.12 this指针深入理解

This指针在不同情况下，指向不同的位置，主要有以下几种情况。

**·方法调用模式**

当函数被保存为一个对象的属性时，它就可称为这个对象的方法。当一个方法被调用时，this被绑定到这个对象上。

var name = "window";

var obj = {

name: "obj",

getName: function(){

alert(this.name);

}

}

obj.getName(); //"obj" 输出为obj 表明this指针指向了obj对象内的name变量，而没有指向obj对象外的全局变量name

**·函数调用模式**

当一个函数并非一个对象的属性时，那么它就是被当做函数来调用的。在此种模式下，this被绑定为全局对象，在浏览器环境下就是window对象。

var name = "window";

function getName(){

alert(this.name);

}

getName(); //"window" 表明this指针指向了函数外的全局变量name

**·构造函数模式**

如果在一个函数前面加上new关键字来调用，那么就会创建一个连接到该函数的prototype成员的新对象，同时，this会被绑定到这个新对象上。这种情况下，这个函数就可以成为此对象的构造函数。

function obj(){

this.name = "obj";

}

var o = new obj();

alert(o.name); //"obj" 使用new创建对象o时，在构造函数中this指向了o，因此o的name属性被赋值为“obj”

### 6.1.13 JS数据类型及其判断方法

详见该文档的 [链接](#_JS的数据类型有哪些) （ctrl+鼠标左键访问）

代码实例见文件夹 **基本数据类型**

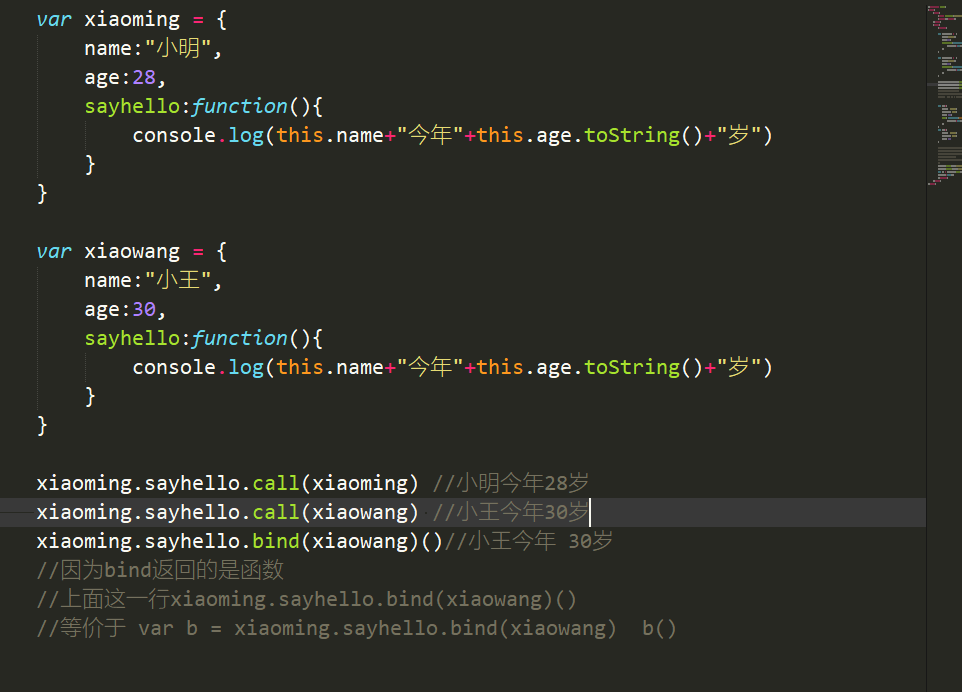
### 6.1.14 call、apply、bind的区别

Call 和 apply

·这两个方法都是函数对象的方法，需要通过函数对象来调用。·当函数调用call()和apply()时，函数都会立即执行。·都可以用来改变函数的this对象的指向。·第一个参数都是this要指向的对象（函数执行时，this将指向这个对象），后续参数用来传实参。bind

·都能改变this的指向·call()/apply()是立即调用函数·bind()是将函数返回，因此后面还需要加()才能调用。

参考如下例子：详见代码 this指针/call、apply和bind的区别.html



### 6.1.15 判断一个数据是不是数组的方法

Var a = [1,2,3,4]

·console.log(Array.isArray(a)) //true

·console.log(a instanceof Array) //true

·console.log(a.constructor == Number) //true

·console.log(b.\_\_proto\_\_ == Array.prototype) //true

Array.isArray 方法判断，instance关键字判断，constructor关键字判断

\_proto\_ 原型链判断。

**A instanceof B**

instanceof的判断规则是：沿着A的\_\_proto\_\_这条线来找，同时沿着B的prototype这条线来找，如果两条线能找到同一个引用，即同一个对象，那么就返回true。如果找到终点还未重合，则返回false。

### 6.1.16 数组去重

Array.from(new Set(a));

### 6.1.17 console.log和console.dir的区别

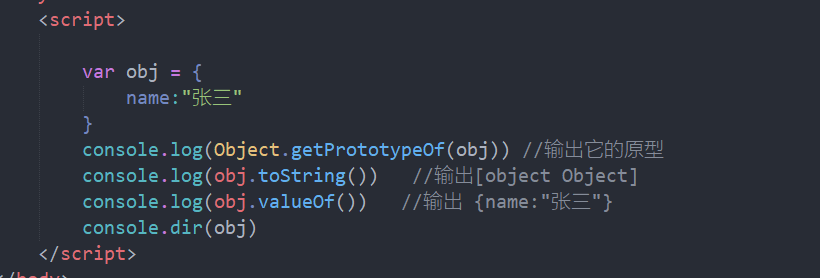
详见本文档[5.1 console.log 和console.dir的区别](#_5.1_console.log_和console.dir的区别)

Console.log是单纯的输出这个数据内容

Console,dir能够输出这个数据的对象和所具有的的方法

### 6.1.18 JS的原型和原型链

详见本文档[5.9 原型、原型链的理解](#_5.9_原型、原型链的理解)



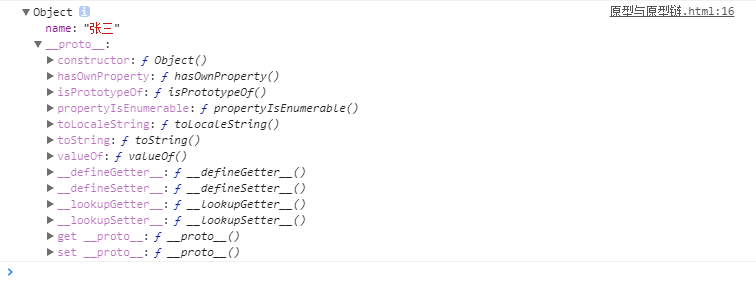
我们发现 console.dir(obj) 打出来的结果是：

1. obj 本身有一个属性 name（这是我们给它加的）

2. obj 还有一个属性叫做 \_\_proto\_\_（它是一个对象）

3. obj.\_\_proto\_\_ 有很多属性，包括 valueOf、toString、constructor 等

4. obj.\_\_proto\_\_ 其实也有一个叫做 \_\_proto\_\_ 的属性（console.log 没有显示），值为 null



这跟\_\_proto\_\_有关。

当我们「读取」obj.toString时，JS引擎会做下面的事情：

1.看看obj对象本身有没有toString属性。没有就走到下一步。

2.看看obj.\_\_proto\_\_对象有没有toString属性，发现obj.\_\_proto\_\_有toString属性，于是找到了

所以obj.toString实际上就是第2步中找到的obj.\_\_proto\_\_.toString。

可以想象，

3.如果obj.\_\_proto\_\_没有，那么浏览器会继续查看obj.\_\_proto\_\_.\_\_proto\_\_

4.如果obj.\_\_proto\_\_.\_\_proto\_\_也没有，那么浏览器会继续查看obj.\_\_proto\_\_.\_\_proto\_\_.proto\_\_

5.直到找到toString或者\_\_proto\_\_为null。

上面的过程，就是「读」属性的「搜索过程」。

而这个「搜索过程」，是连着由\_\_proto\_\_组成的链子一直走的。

**这个链子，就叫做「原型链」。**