Contents

[var let的差异 2](#_Toc35877962)

[Number和parseInt的对比 3](#_Toc35877963)

[NaN 3](#_Toc35877964)

[防抖和节流 4](#_Toc35877965)

[防抖： 4](#_Toc35877966)

[节流： 4](#_Toc35877967)

[SetTimeout与Setinterval的用法 5](#_Toc35877968)

[Set和WeakSet 6](#_Toc35877969)

[Map和WeakMap 7](#_Toc35877970)

[判断数组的几个方法 8](#_Toc35877971)

[Object.prototype.toString.call() 8](#_Toc35877972)

[Instanceof 8](#_Toc35877973)

[Array.isArray() 8](#_Toc35877974)

[变量提升 8](#_Toc35877975)

[For in 和 for of 的区别 9](#_Toc35877976)

[For in 9](#_Toc35877977)

[For of 9](#_Toc35877978)

[Call和Apply的区别 10](#_Toc35877979)

[Bind的使用 10](#_Toc35877980)

[箭头函数 11](#_Toc35877981)

[ForEach 11](#_Toc35877982)

[Array, new Array(), []的区别与使用 11](#_Toc35877983)

[闭包 12](#_Toc35877984)

[This 13](#_Toc35877985)

[默认绑定 13](#_Toc35877986)

[隐式绑定 13](#_Toc35877987)

[显示绑定 13](#_Toc35877988)

[绑定规则优先级 13](#_Toc35877989)

[New 14](#_Toc35877990)

[Instanceof 原理 15](#_Toc35877991)

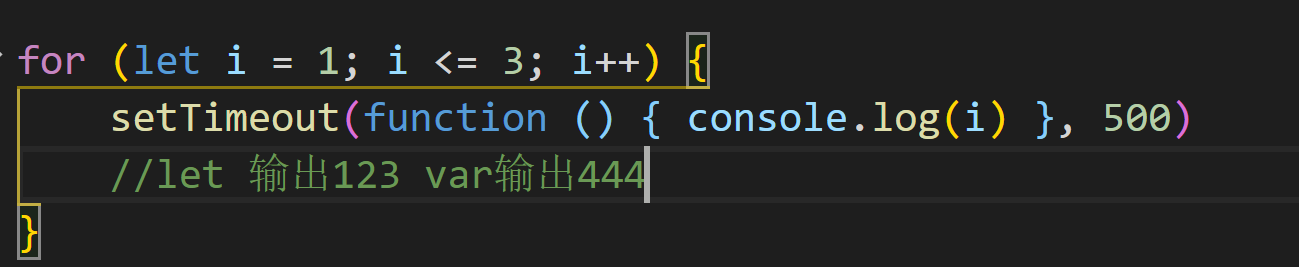
[强制构造函数通过new调用 15](#_Toc35877992)

[New.target 15](#_Toc35877993)

[原型链 16](#_Toc35877994)

# var let的差异

**js没有块级作用域，var声明的变量作用域是整个封闭函数体**



**let声明的变量拥有块级作用域**

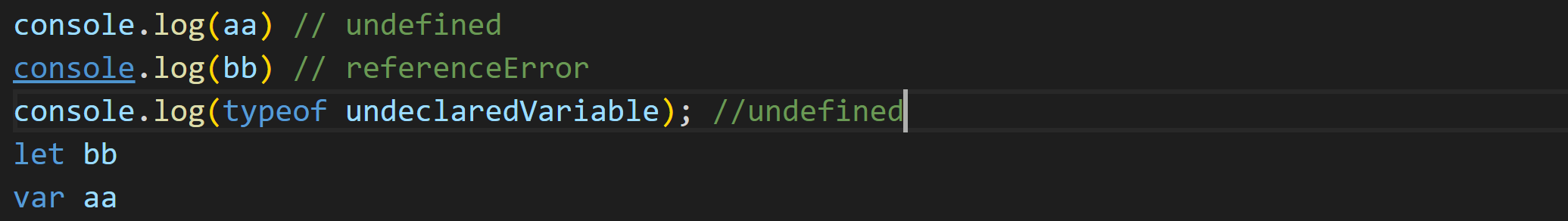
**let声明的全局变量不是全局对象的属性。**它们只存在于一个不可见的块的作用域中，这个块理论上是Web页面中运行的所有JS代码的外层块。

**用let在一个作用域内重定义变量会抛出一个语法错误（SyntaxError）而var可以**

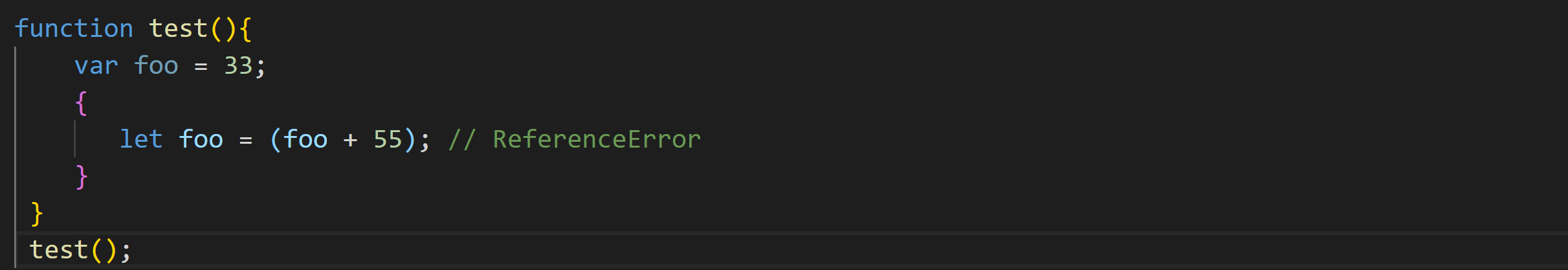
Var和let都支持解构特性

在构造函数中可以通过let声明而非闭包来创建私有成员

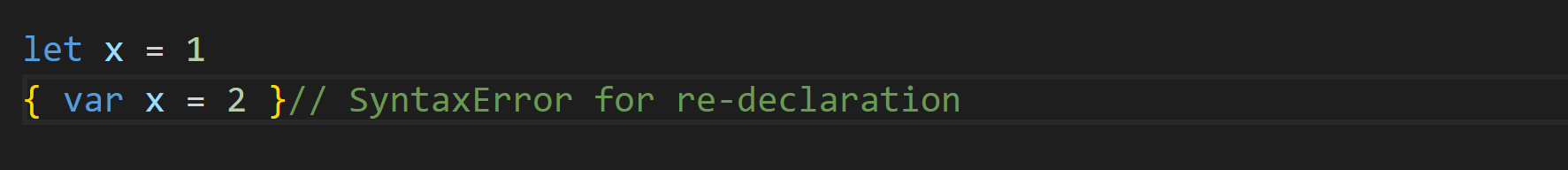
与通过var声明的有初始值undefined不同，let声明的变量直到他们的定义被执行时才初始化，在初始化前访问该变量会导致ReferenceError，该变量处在一个自快顶部到初始化处理的“暂存死区”中，如果用typeof检测在暂存死区中的变量也会抛出ReferenceError

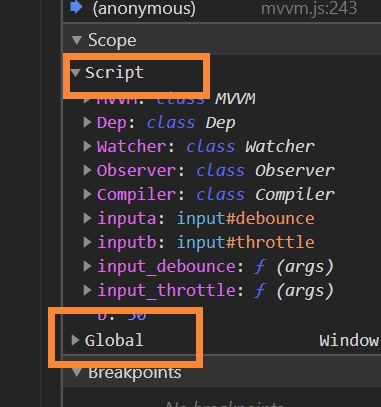


特别注意词法作用域



以及var的变量提升（var a被提升到函数体顶部，js没有块级作用域）



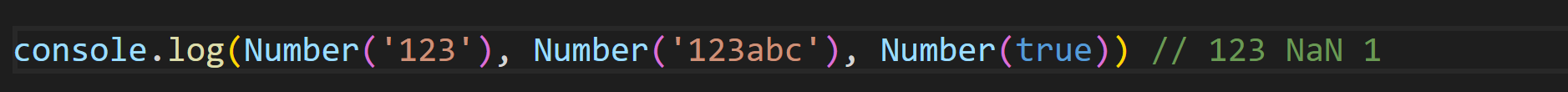


ES5中顶层属性和全局变量是相同的，因为全局的var和function在顶层属性window中

而ES6中的let、const声明的全局变量不在顶层属性中，而是在一个属于浏览器的块级作用域Script中

# Number和parseInt的对比

根本实现原理不同，前者是通过构造函数实现，支持多种数据类型，注意字符串必须是纯数字，转换为一个64bit的浮点数。转化失败返回NaN。

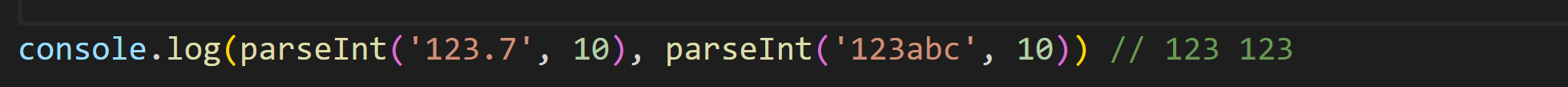


后者接受两个参数 (string,redix)

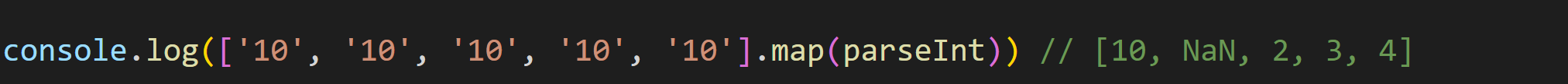
String会从头遍历到非数字为止进行转化，故属于一种“向下取整“

Redix表示进制，2-36的整数（0默认10进制），没有第二个参数时依照es版本不同而不同：String以0x开头十六进制，0开头8或10进制，其余十进制。

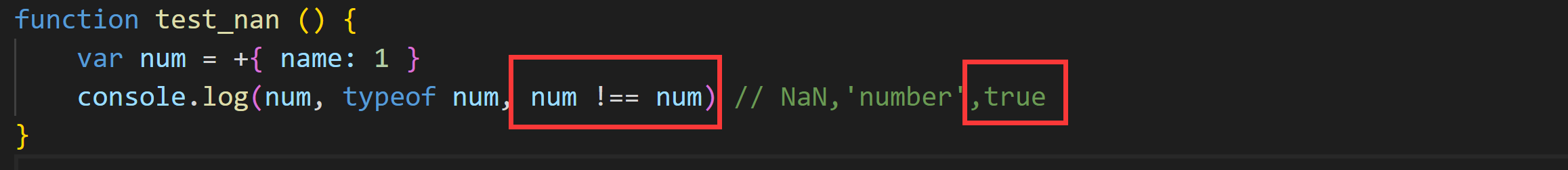
如果转化失败返回NaN，注意redix为0的情况



例题：



# NaN



# Const

Const非常类似一个不能改变的let（块级作用域，暂时性死区，不能重新声明）不能通过重新赋值来改变，但是对于const声明的对象可以修改其内部的值。

# 防抖和节流

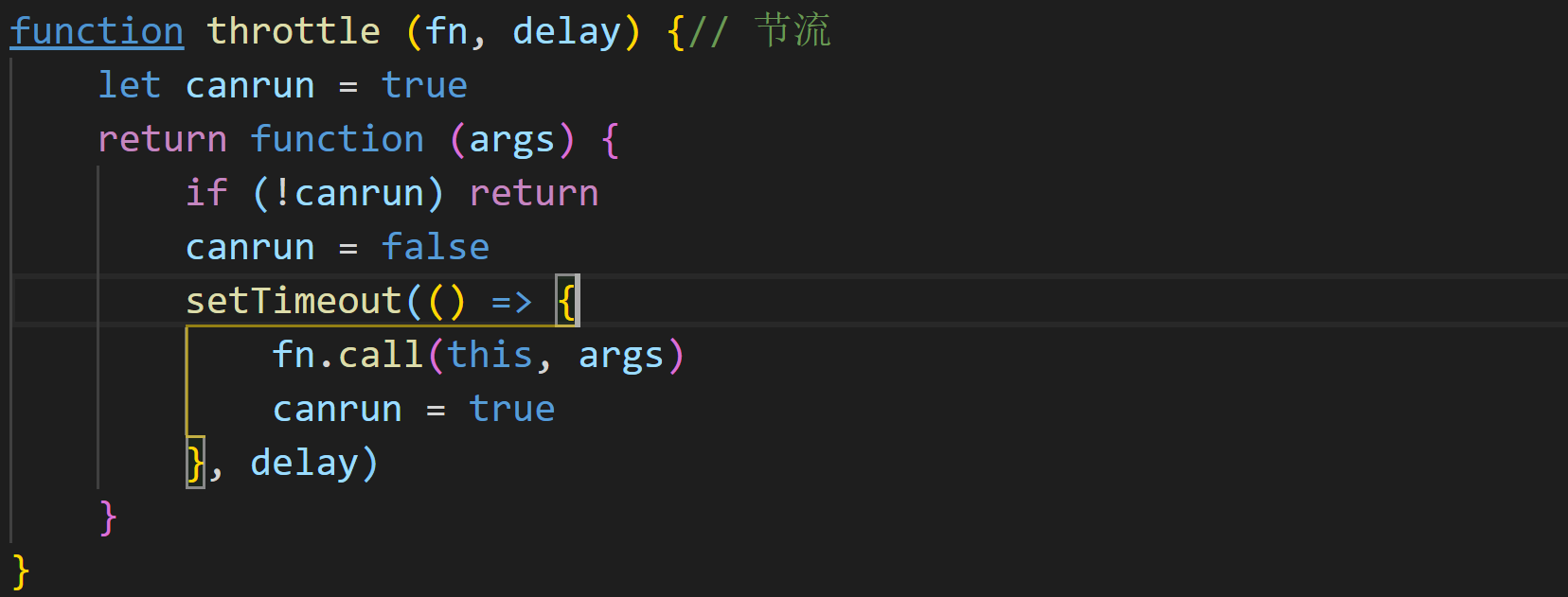
防抖：在事件被触发n秒后再回调，如果n秒内再次触发则重置计时

通俗的讲：法师读条，如果再次放技能重新吟唱

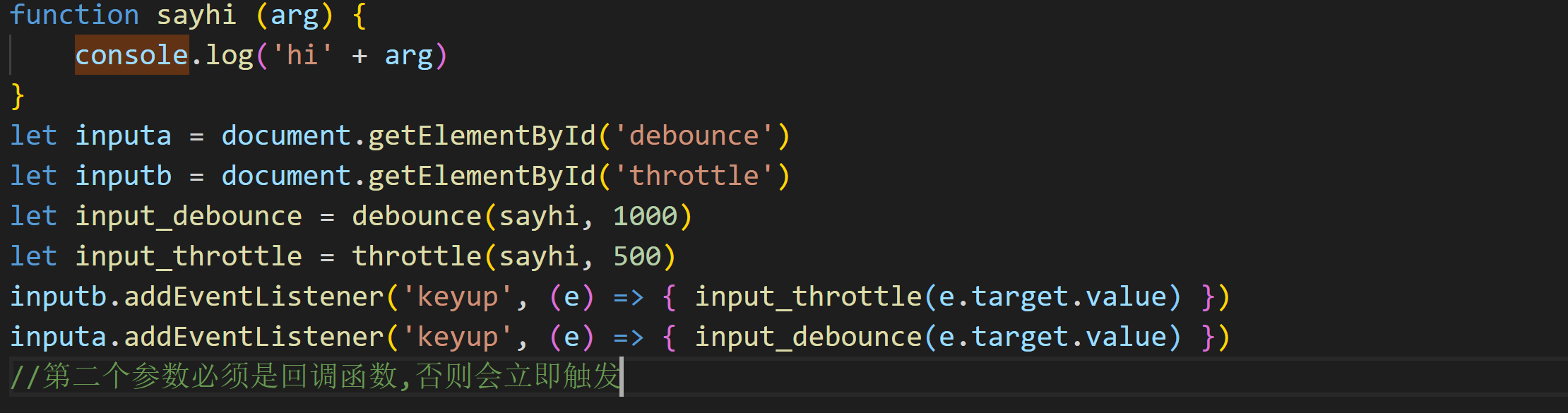


节流：规定在单位时间内只执行一次回调

通俗的讲：FPS鼠标点的再快射速也是固定的



通过getelementbyid获取元素，addeventlistener用事件触发回调



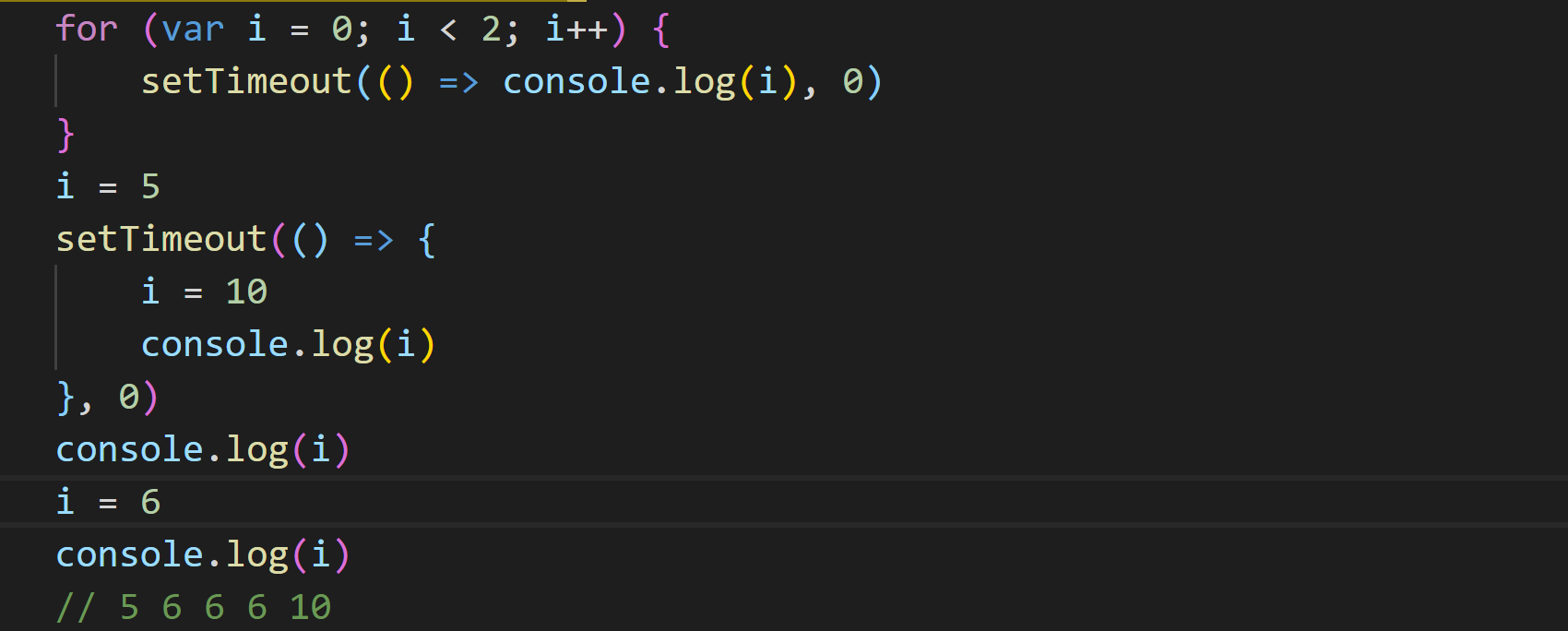
# SetTimeout与Setinterval的用法

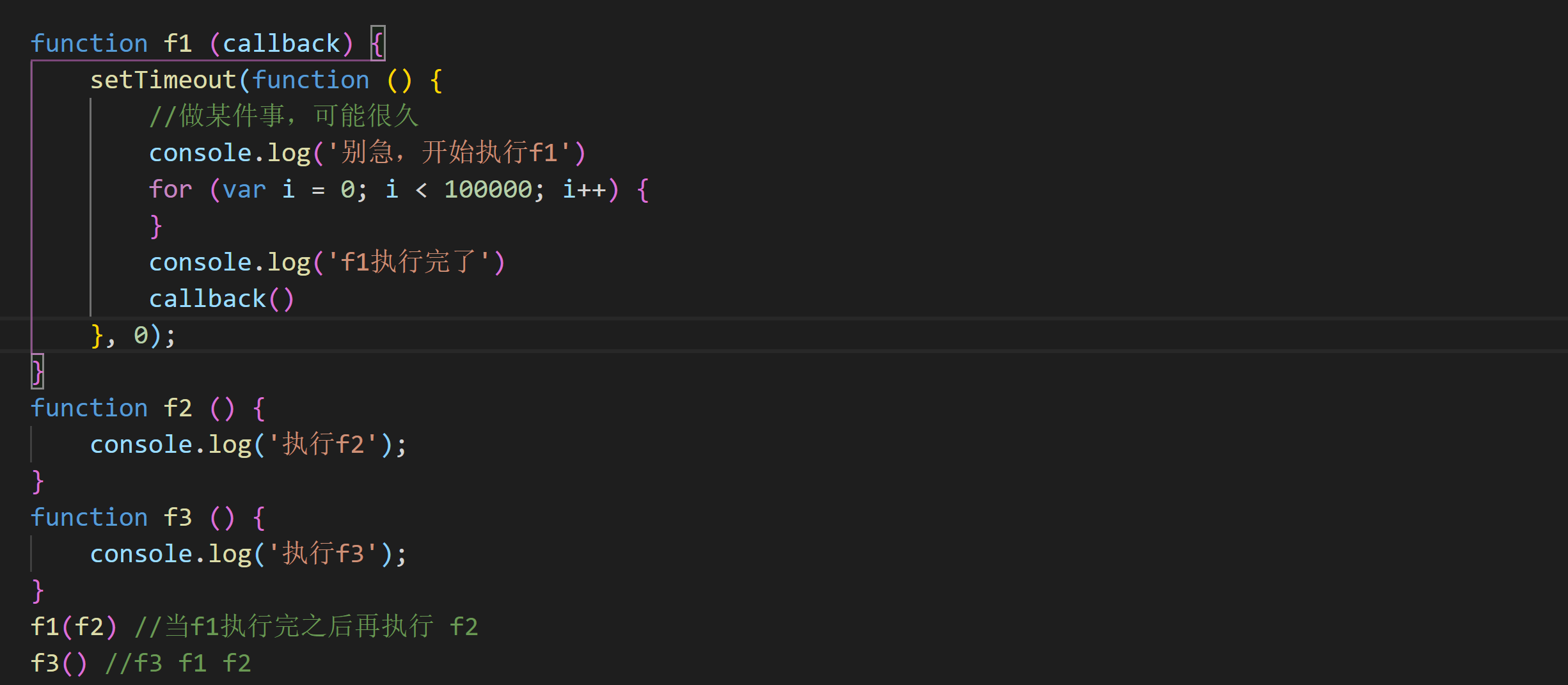
两个函数的用法一致，前者是推迟到delay时间段执行，后者是周期运行。

注意第一个参数如果是函数可以直接作为参数，如果是代码则需要以字符串形式带入

运行机制：把代码移出本次执行，等到下一轮event loop时判断计时器，如果时间到了则执行否则继续等待下一轮。

setTimeout(f,0)将第二个参数设为0，作用是让f在现有的任务（脚本的同步任务和“任务队列”中已有的事件）一结束就立刻执行。也就是说，setTimeout(f,0)的作用是，尽可能早地执行指定的任务。





# Set和WeakSet

**Set**

ES6新增数据结构，类似数组，但成员唯一且无序

向Set加入值的时候不会进行类型转换，所以5和’5’是不同的，Set 内部判断两个值是否不同，使用的算法叫做“Same-value-zero equality”，它类似于精确相等运算符（===），主要的区别是NaN等于自身，而精确相等运算符认为NaN不等于自身。

实例方法：add delete has clear

**WeakSet**

WeakSet 对象允许你将**弱引用对象**储存在一个集合中

WeakSet 与 Set 的区别：

* WeakSet 只能储存对象引用，不能存放值，而 Set 对象都可以
* WeakSet 对象中储存的对象值都是被弱引用的，即垃圾回收机制不考虑 WeakSet 对该对象的应用，如果没有其他的变量或属性引用这个对象值，则这个对象将会被垃圾回收掉（不考虑该对象还存在于 WeakSet 中），所以，WeakSet 对象里有多少个成员元素，取决于垃圾回收机制有没有运行，运行前后成员个数可能不一致，遍历结束之后，有的成员可能取不到了（被垃圾回收了），WeakSet 对象是无法被遍历的（ES6 规定 WeakSet 不可遍历），也没有办法拿到它包含的所有元素

应用实例：在一个类中，通过下文可以优雅的组织在实例化前调用成员函数



# Map和WeakMap

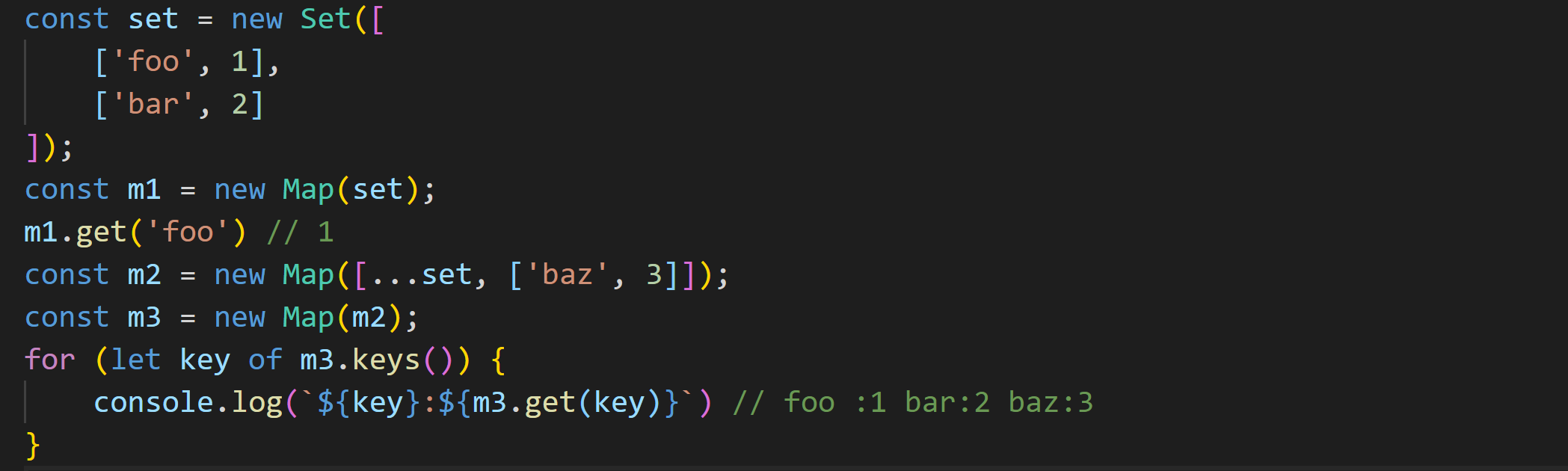
**Map和Set的区别：**

前者以[key,value]存储（类似python中的dic），后者以[value,value]存储

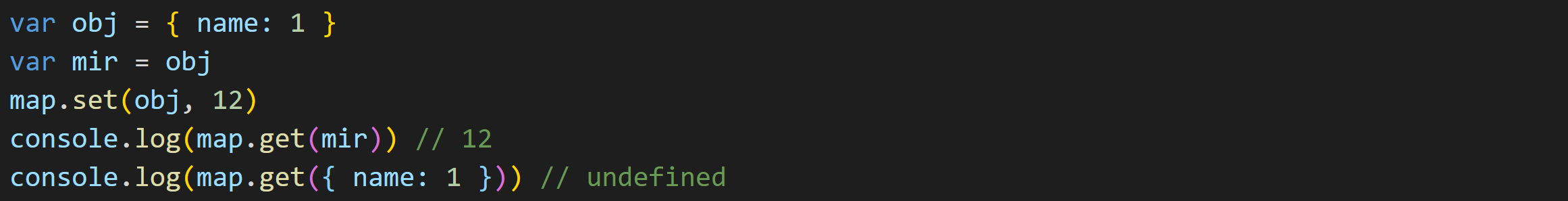
**共同点：**

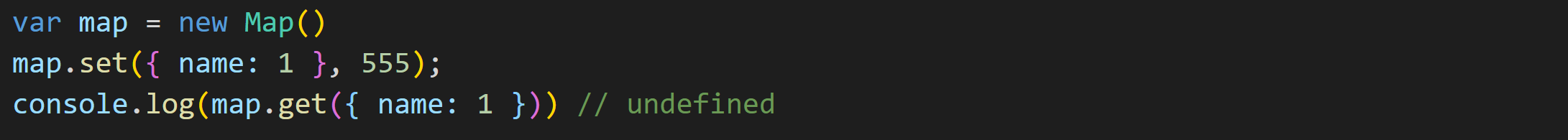
都没有重复的值

**任何具有 Iterator 接口、且每个成员都是一个双元素的数组的数据结构**都可以当作Map构造函数的参数，例如：



只有对同一个对象的引用，Map才会认为是一个键值





上图的set和get表面针对的是一个值，但是实际内存地址不同，可见map的键是和内存地址绑定的，这解决了同名clash问题。

如果Map的键是一个简单类型的值（数字，字符串，布尔值）只要两个值严格相等则视为同一个值，注意同set一样NaN虽然不严格等于自身但是视为同一个键。

**WeakMap**

WeakMap中的键也是对象的弱引用

# 判断数组的几个方法

## Object.prototype.toString.call()

每一个继承Object的对象都有toString方法，如果没有被重写会返回[Object type]。但当除了Object类型的对象外其他类型直接使用toString方法时会直接返回都是内容的字符串，所以需要使用call或apply来改变toString执行的上下文

常用于判断浏览器内置对象

## Instanceof

内部机制是通过判断对象的原型链中能不能找到类型的prototype，但只能用来判断对象类型，原始类型不可以，并且所有对象类型的 instanceof object都是true

## Array.isArray()

检测Array实例时，Array.isArray()优于instanceof因为可以检测出iframe

此方法ES5新增，不支持时可以使用Object.prototype.toString.call()

# 变量提升

在**编译阶段**把变量和函数的声明放到内存中。

函数声明在编译阶段会被添加到词法环境，当js引擎遇到函数时从此法环境中寻找并执行。

这里特殊注意let的暂存死期，个人理解let是某种意义上的变量提升，但是let声明的变量不会像var那样声明时自带undefined，let带来的暂存死区会导致在死区内任何形式的访问都会抛出referenceError（包括typeof），这代表let是存在变量提升的，否则typeof应返回undefined。

注意Class\const\let情况一致

# For in 和 for of 的区别

## For in

可以用for in来遍历数组和对象

ES5还可以用foreach，map，filter，reduce等方法遍历数组

注意for in同时也会遍历数组(含对象)和对象原型的属性和方法

* 故数组返回的是数组的下标和数组的属性和原型上的方法和属性
* 对象返回的是对象的属性和原型上的方法和属性，对象可以用hasOwnProperty来判断只显示对象包含的属性

For in遍历数组存在以下问题

* Index是字符串 需要转换才能进行几何运算
* 遍历顺序有可能不是按着实际数组的内部顺序
* 会遍历所有的可枚举属性

## For of

ES6新增迭代（iterator）接口供for…of消费，迭代器为各种不同的数据结构提供统一的访问机制，任何数据只要部署了iterator接口就可以完成遍历操作

Object.keys\values\entries (obj), Set\Map.keys\values\entries()

对于for of而言他会调用被遍历对象的[Symbol.iterator]方法

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 数据类型 | Object | Set | Array | Map |
| Symbol.iterator | undefined | values | values | entries |

因而对于数组而言，返回的是值而不是下标（可以改成for (let key of Object.keys())），

不会返回可枚举属性

# Call和Apply的区别

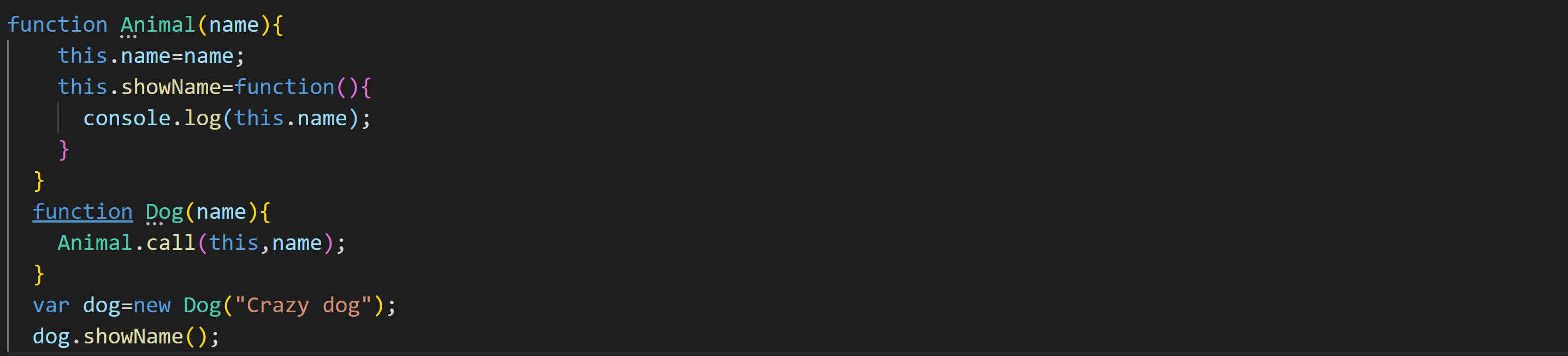
二者都接受两个参数

* 第一个参数相同，都是函数内this的指向，如果不传\null\undefined则函数中的this指向全局，如果传String\Number\Boolean则指向对应的包装对象，如果传对象则指向对象，如果传函数则指向这个函数的引用
* 第二个参数不同在于，call接受的参数不固定，都会传给函数作为参数而apply接受一个带下标的集合，数组或类数组。

实践表明call的性能比apply要好，尤其是ES6支持解构以后我们调用call更加的方便。

个人考虑可能是apply对第二个参数要进行拆解，增大了成本。

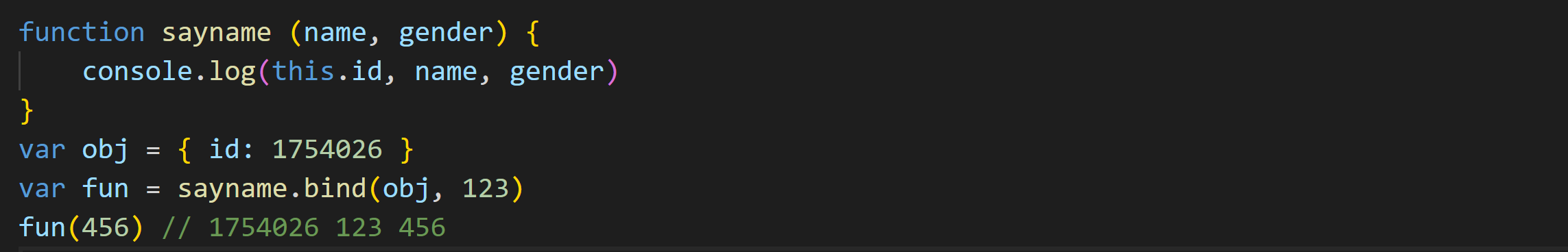
Call继承



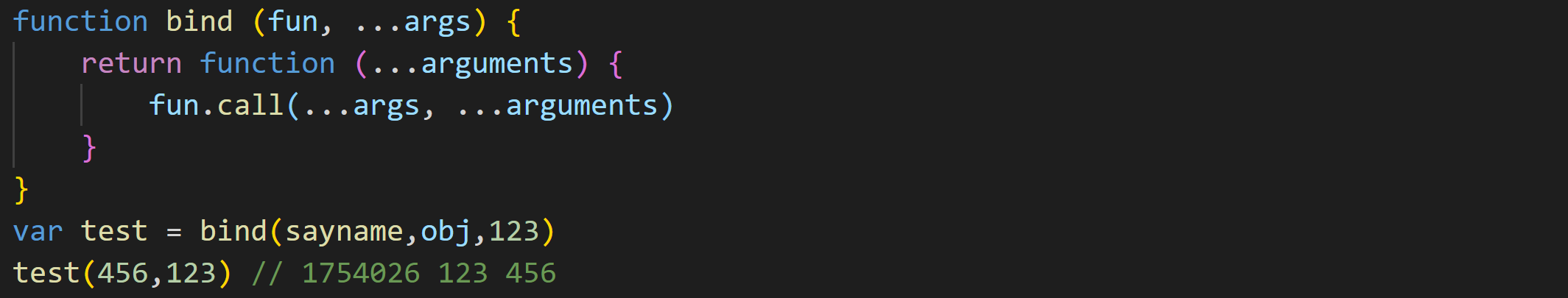
# Bind的使用

ES5方法，参数使用与call几乎相同，但是返回一个函数

Bind的参数会覆盖需要格外注意：



利用call实现bind



# 箭头函数

适用于需要匿名函数的地方，箭头函数表达式适合非方法函数

语法特点：

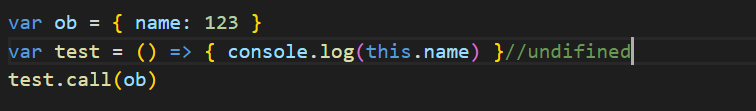
* 函数体只有一个return语句时可以省略return
* 支持剩余参数，默认参数和参数列表解构

注意：

* 不能用作构造器，和new一起用会抛出错误
* 函数体内不使用var定义的变量是全局变量，而箭头函数内是局部变量
* 箭头函数也可以使用闭包

箭头函数没有自己的this,arguments,super或new.target

* 从作用域链的上一层继承this
* 对箭头函数任何方式绑定一个this是无效的



# ForEach

回调函数参数(value,index,arr)，第二个参数表示回调函数中this的指向，不指定时为外层作用域内的this

没有break，不能return，删除自身index不会重置

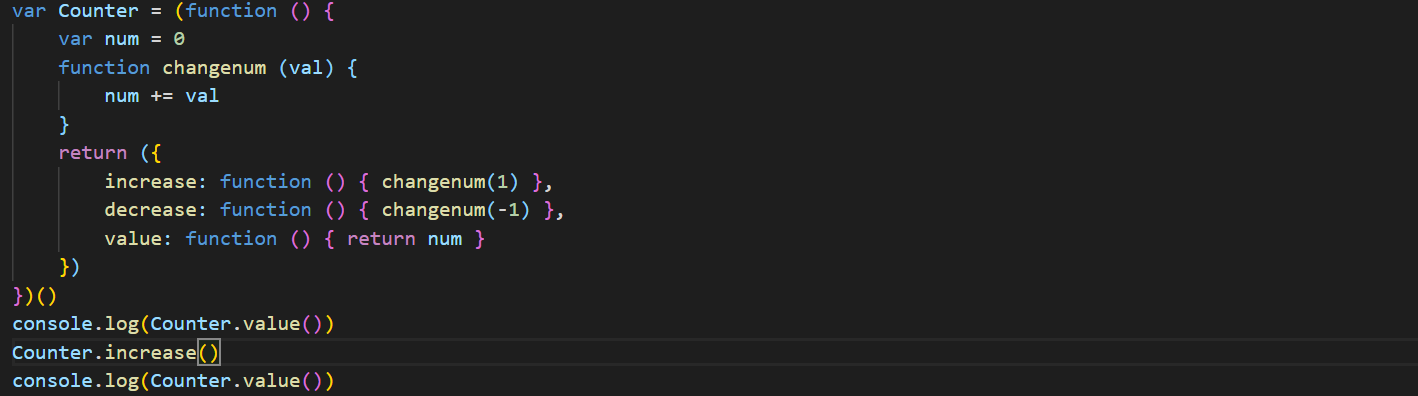
# Array, new Array(), []的区别与使用

前两者完全一致，new语法糖，new Array()创建了一个对象，新对象的\_proto\_为Array.prototype。

从结果上来讲，[]得到的结果和以上两者也一样

# 闭包

函数和对其周围状态（词法环境）的引用捆绑在一起构成闭包。闭包允许从内部函数访问外部函数作用域。在JS中，每当函数被创建，就会在生成时生成闭包。



上例中一个词法环境为三个函数所共享。这个共享环境创建于一个立即执行的匿名函数体内，环境中包含两个私有项：属性num和函数changenum，他们只能被匿名函数返回的三个公共函数访问，这就利用闭包实现了私有属性。把匿名函数赋值给变量即可创建多个计数器。

闭包提供了许多面向对象编程的好处，尤其是数据隐藏和封装。



赋值给onfocus的是闭包，由于变量提升，item也属于函数作用域，三个在循环中创建的闭包共享一个词法作用域，而循环早就在事件出发前就执行完毕，所以三个闭包用的item是一个值。可以通过使用更多（匿名）闭包，或使用let。

**性能考量**

如果不是因为特定任务需要使用闭包，在函数中创建函数是不明智的，因为闭包在处理速度和内存消耗方面对脚本性能有负面影响。例如在创建对象或类时我们一般把方法和原型关联，把属性定义到构造器中，这是因为每次构造器被调用时，构造器中的方法都会被重新赋值一次。

# This

## 默认绑定

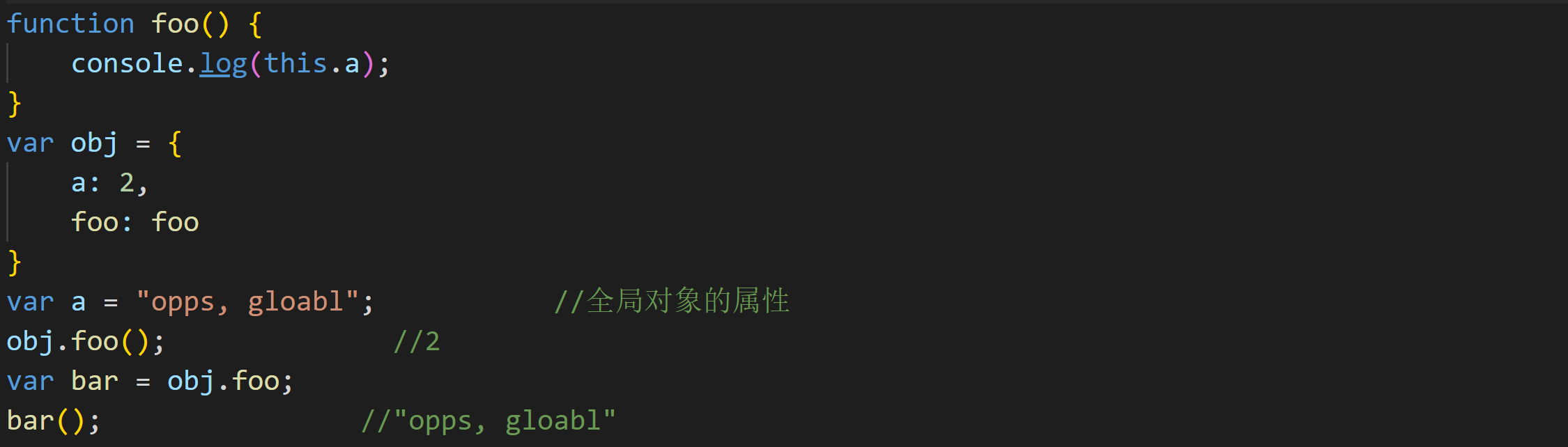
一般情况下无其他规则，则指全局对象



## 隐式绑定

有上下文对象，如把函数设定为对象的一个方法。注意隐式丢失

绑定丢失也会发生在函数作为参数传递的情况下，即传入回调函数时，因为参数传递就是一种隐式赋值

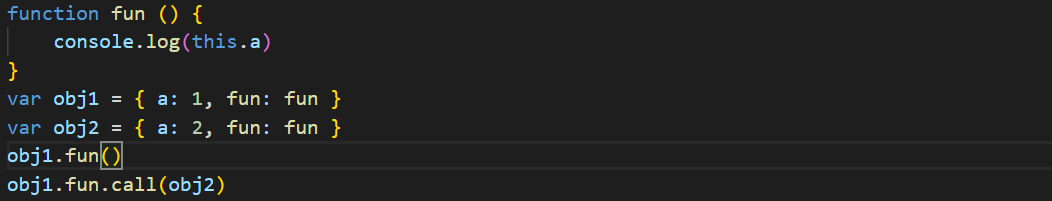


## 显示绑定

* apply\call
* bind硬绑定
* 通过API调用参数指定this如forEach
* 通过new绑定（new时创建的新对象会绑定到函数调用的this）
* 箭头函数根据外层作用域来决定，且绑定无法修改

## 绑定规则优先级

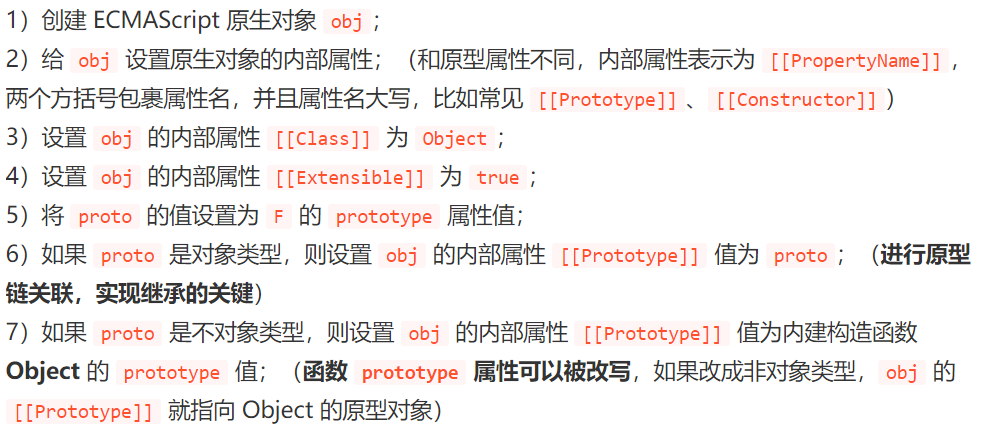
New>显示绑定>隐式绑定

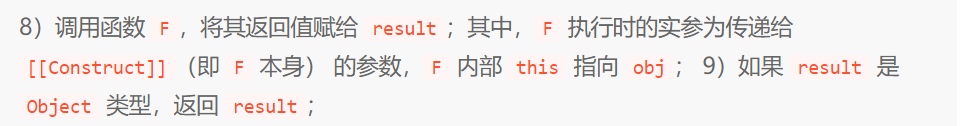


# New

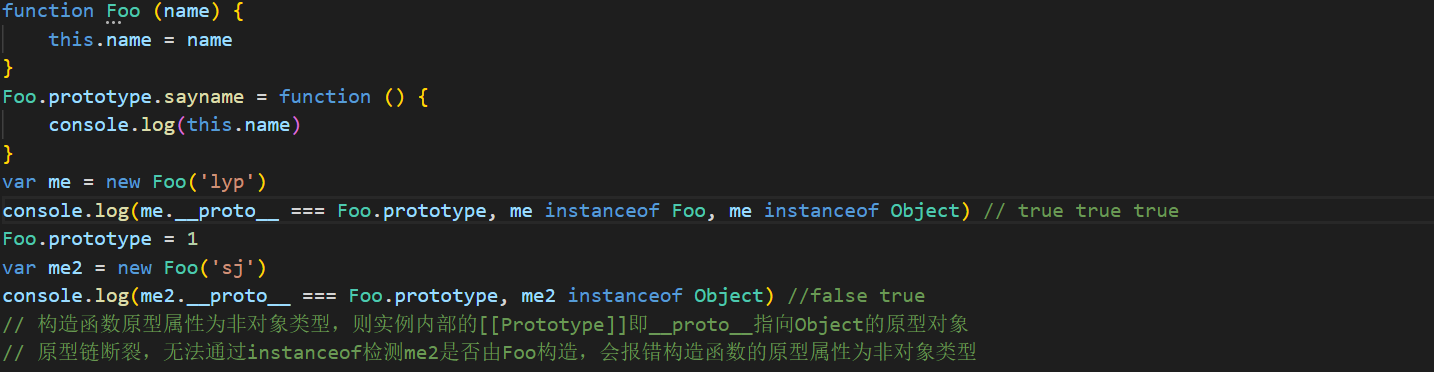
通过new操作符调用一个函数来通过构造函数创建对象

* 无return 构造函数默认情况
* Return对象 返回return的对象，此前工作白做
* Return 基本数据类型 和没有return效果相同

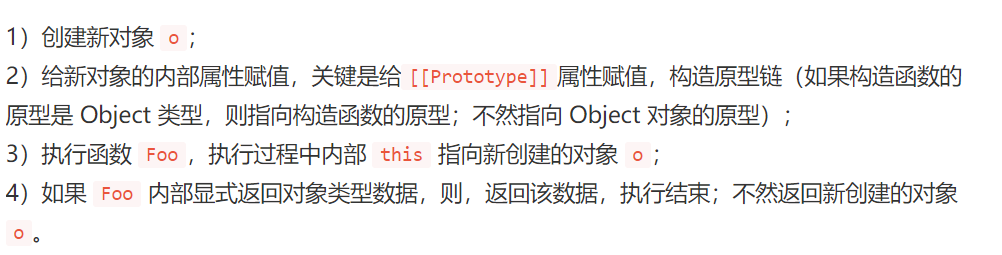








简明过程表述：

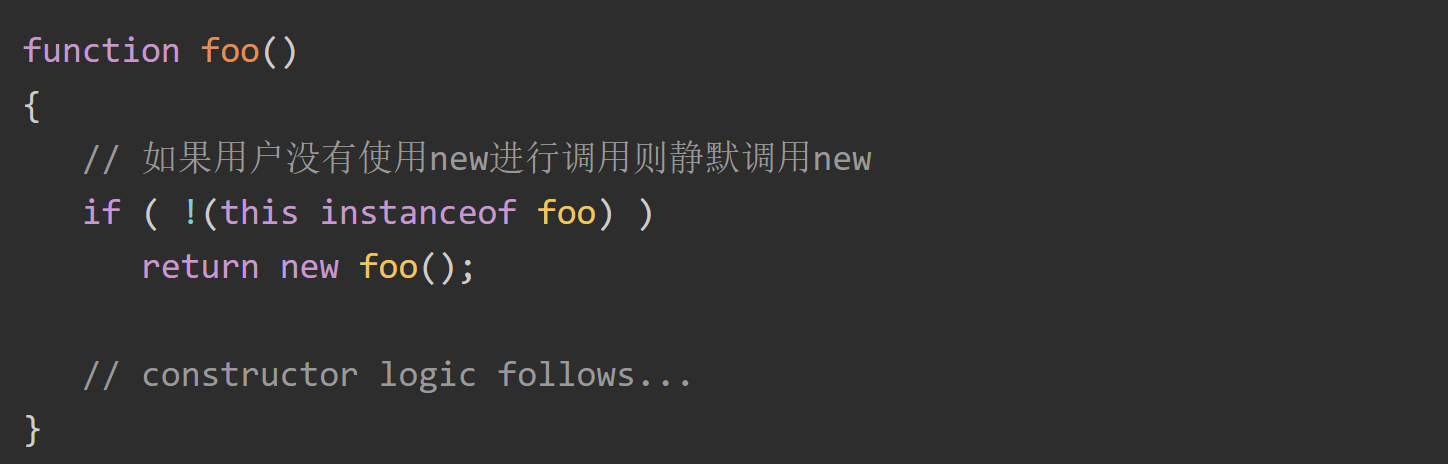




# Instanceof 原理

X instanceof Foo 如果Foo的原型（Foo.prototype）出现在x的原型链中则true。如果x通过new Foo()构造，然后修改构造函数原型，则x instance of Foo 返回false

# 强制构造函数通过new调用



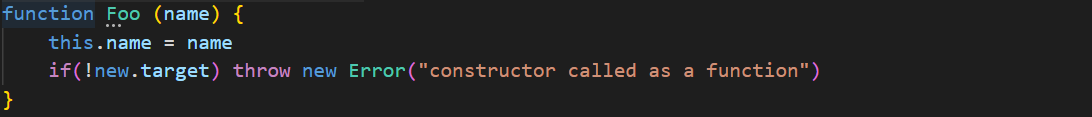
OR



在new的过程中，新建对象的[[prototype]]内部属性是Foo的prototype且foo的this在执行时被指向了对象，所以instanceof会判定为真。

# New.target

指向被new调用的构造函数，常和上一点一样用于构造函数非new调用的异常抛出



# 原型链

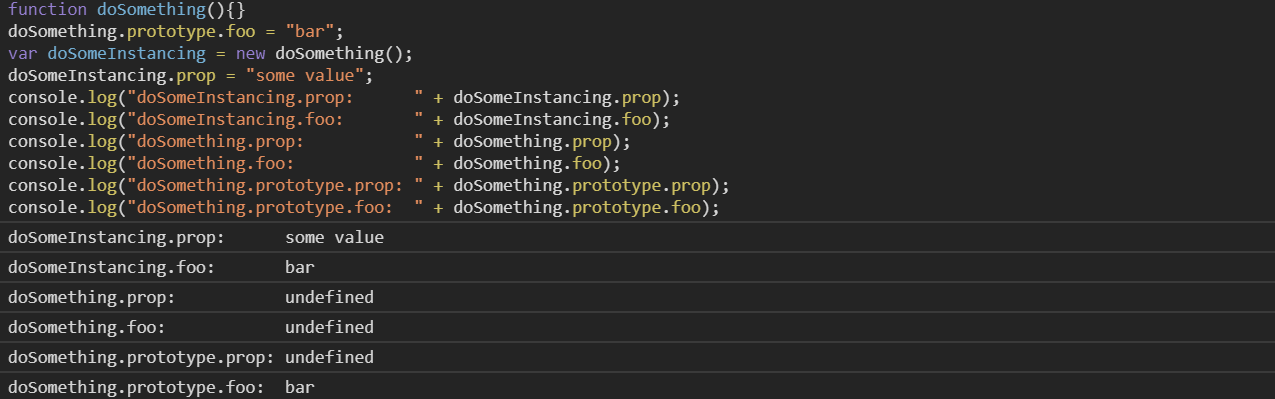
参考链接：<https://developer.mozilla.org/zhCN/docs/Web/JavaScript/Inheritance_and_the_prototype_chain>

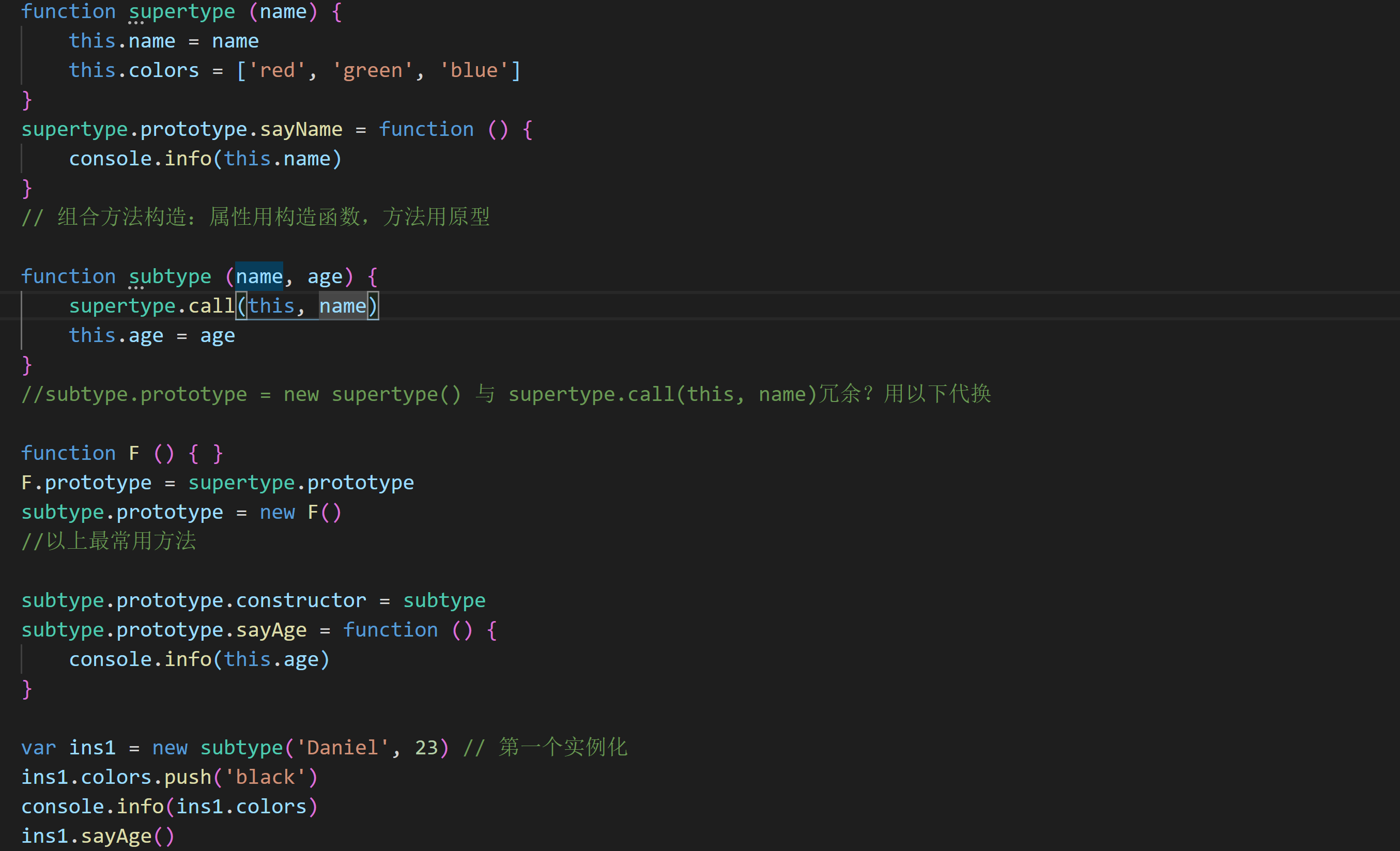
JavaScript本身不提供class实现，虽然ES6中加入了Class关键字，但这只是语法糖，根本实现仍然是基于原型的。

谈到继承时，JavaScript只有一种结构：对象。每个实例对象都有一个私有属性\_\_proto\_\_指向构造函数的原型对象(prototype)。该原型对象也有自己的\_\_proto\_\_，层层向上直到一个对象的原型对象为null，根据定义null没有原型，并作为这个原型链的最后一个环节。

几乎所有JavaScript中的对象都是位于原型链顶端的Object的实例（在往上就是null了）可以通过[[Prototype]]即\_\_proto\_\_访问原型。

Js中所有的函数都有prototype。注意遍历对象属性时，每个可枚举属性都会被美剧出来，要检查对象是否有自己的属性而不是从原型链继承的必须使用hasOwnProperty方法，以及Object.keys()方法，以上两种不会遍历原型链。





# 模块化

**模块化的进化过程**

* **全局function模式：**将不同的功能封装成不同的全局函数，污染全局命名空间，容易引起命名冲突和数据不安全，而且模块成员之间看不出直接关系
* **Namespace模式（简单对象封装）：**减少了全局变量，解决了命名冲突但是数据仍不安全（暴露所有模块成员，外部可以直接访问和修改内部属性）
* **IIFE模式（匿名函数自调用\闭包）：**属性私有，外部只能通过暴露的方法操作

引入多个script后会出现请求过多，依赖模糊，难以维护的问题，因而需要模块化规范

**模块化规范的进化过程**

无模块化🡪CommonJS规范🡪AMD规范🡪CMD规范🡪ES6规范

* **无模块化：**通过script标签引入js文件，要求依赖在前，污染全局作用域，维护成本高，依赖关系不明显
* **CommonJS规范：**最初应用于服务器端node，module.exports导出，require同步加载模块，同步不适合浏览器端，所以有了异步的AMD规范
* **AMD规范：**允许指定回调函数，AMD是requirejs的规范化产出，定义了三个API
  + define(id,[depends],callback) // 定义模块
  + require([module],callback) // 加载模块
  + require.config() // 指定引用路径

适合在浏览器中异步加载模块，并行加载多个模块

* **CMD规范：**AMD推崇依赖前置，提前执行；CMD推崇依赖就近，延迟执行。SeaJS的规范化产出
* **ES6模块化：**import和export实现导入和导出，注意一个模块只能有一个export default。动态引用（模块输出值的引用对比Commonjs输出值），CommonJS是运行时加载（载入时加载整个模块生成对象再从对象上读取方法），ES6不像CommonJS，ES6模块不是对象，通过export输出指定代码，import时可以加载某个特定输出值而不是加载整个模块，即编译时加载。

# Event Loop

遇到事件或者settimeout等会把对应的回调函数挪出当前执行放到一个事件队列，等到主程序执行完毕就依次把队列里的函数压入栈中执行，特别注意settimeout(fn,0)

# IIFE

# Try-Catch

1. 把有可能出现报错的代码放到try中，只要有一行代码出现问题，整个程序的执行流程会立即调到catch语句中执行
2. 一旦try中有一行代码发生异常，其后的try块中的代码都不会再被执行。
3. 在执行catch中代码之前，js引擎会首先根据错误类型自动创建一个错误，并通过catch后面的参数传递到catch中，不同浏览器创建的error对象不一样但是都会包含一个message属性表示错误的信息
4. Catch中的代码执行完毕之后会继续执行之后的代码，程序不会停止下来

在trycatch中引入了finally块放在catch(error)后，无论是否发生错误都会执行这段代码，注意如果添加了finally可以省略catch语句，但是如果try中发生错误，在finally执行完毕后程序会终止。

# Promise

Promise是一个对象，有两个特征

* 对象的状态不受外界影响，Promise对象代表一个异步操作有三种状态，pending，fulfilled和rejected。这个状态只受异步操作的影响。
* 一旦状态改变就不会再变，在对promise对象添加回调函数也会立即得到这个结果

Promise构造函数接受一个函数作为参数，这个函数的两个参数是resolve和reject。其中resolve的作用是把当前promise对象的状态转化为成功，调用resolve把操作结果作为参数传递出去。Reject则是把状态转化为错误并传递错误。可以通过then注册多个promise从而实现链式调用。

注意promise创建和注册的都是微任务，微任务的执行是在当前执行栈结束和下一个任务执行之前，即当前执行站-》micro-task-》taskqueue中取一个回调不断消费taskqueue

Promise并发控制的all/race，他们的返回值也是promise，all会等待所有promise决议之后再决议（与的关系）而race是只要有一个决议就会决议（或的关系），注意参数为空的时候all会立刻决议，而race会挂住

Promise可以通过catch来捕获错误