1. 语法
   1. 数据类型
      1. 简单类型和引用类型

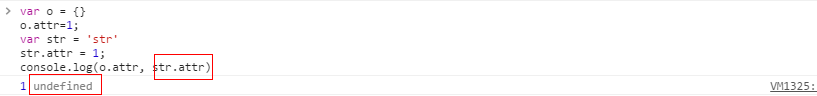
**1.概要**

（1）js中的数据类型可以分为两种：简单类型和引用类型。简单类型又有5种，因此一共6种数据类型（es6还新增了1种简单类型：Symbol类型，为简单类型。Set、Map等类型仍为对象类型）。

（2）5种简单数据类型为：undefined、null、boolean、number、string，保存的是简单数据段，无法被修改（修改的过程其实是先销毁，再赋新值）。复制时创建新值，分配新内存地址。

（3）引用类型又叫复杂数据类型，即object，保存的是引用/指针。复制时复制的是指针/地址。

（4）对于引用类型，可以随时增加属性。对于简单类型，无法添加属性。



（5）简单类型向函数传参，简单类型会复制新值给形参，而引用类型仅传递地址给形参。

（6）简单类型采用值比较，而引用类型采用引用比较，即使对象属性等一样也不相等。



**2.变量**

（1）销毁1个变量，无论是简单类型，还是引用类型，还是引用类型的1个属性，都有delete方法。

（2）在if中使用var声明变量没法限定其作用域，但可以使用let限制其作用域。

* + 1. 类型转换

**1.转为boolean**

（1）无论是显示转换还是隐式转换，转为boolean值都是调用Boolean()函数显示转换。只有""、0、NaN、null、undefined五个值可以转化为false，其余包括[]、{}都会转换为true。

（2）任何值使用两次!!转为boolean值。

**2.转为number**

（1）parseInt/parseFloat（函数）：专门处理字符串转数字，只转换前面部分为数字的字符串，其余均转为NaN。



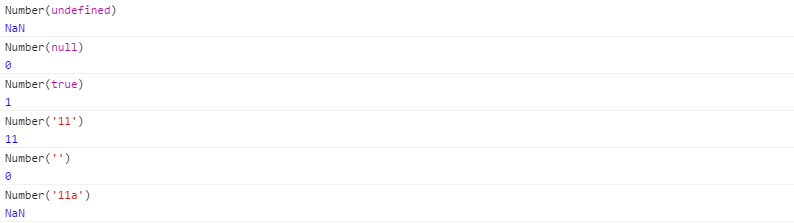


（2）隐式转换，其中，对于+，即，除非两边都是数字或boolean，均按字符串连接处理。



对于除+以外的算数运算符，将调用Number方法隐式转换，规则如下：

* undefined：NaN；null：0。
* boolean：true：1，false：0。
* 字符串：（1）空字符串：0。（2）只包含数字：数字。（3）其他如123a等均转为NaN。

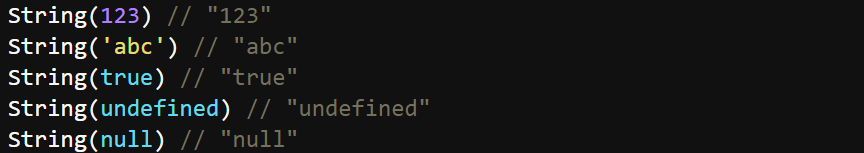


**3.转为字符串**

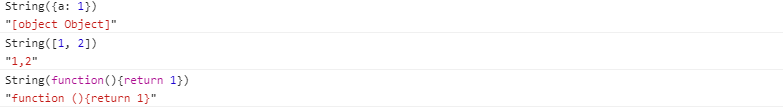
可以使用String()强制转换。如果是隐式转换，也是调用String()方法。有如下规则。

（1）简单类型

直接转为对应的字面量字符串形式。



（2）如果是对象，有如下形式（记住即可）：



* + 1. 类型判断

（1）typeof

返回一个值的数据类型（字符串）。其中，函数返回function，但无论对象还是数组都返回object，null也返回object。

typeof可以用来检查一个没有声明的变量，而不报错。

（2）instanceof

可以使用instanceof判断具体的对象类型，返回1个布尔值。



只能用于对象，不适用原始类型的值。



（3）判断数组

单框架（网页中没有iframe，如果有，因不同的window有不同的Array原型导致无法使用）。



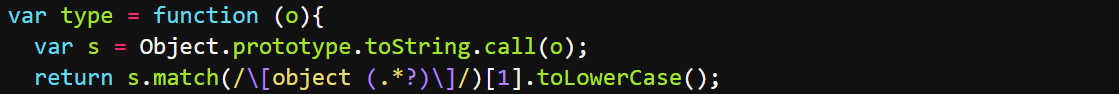
多框架，使用EC5的数组方法isArray判断。



（4）使用Object.prototype.toString方法判断一个值的类型（内置对象和包装对象）。









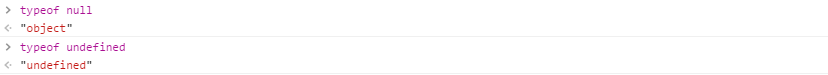
* + 1. 简单类型

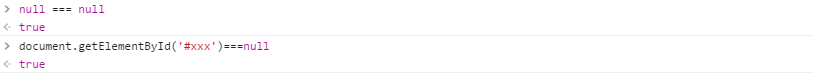
**1.undefined和null**

（1）undefined和null都表示没有。

（2）undefined出现情形：变量声明没有赋值、获取对象不存在的属性、函数形参没有传值其值就为undefined、无返回值的函数的执行结果等。

（3）逻辑角度看，null值表示一个空对象指针。





**2.boolean**

只有true/false两个值。js大小写敏感，不是大写。

**3.number**

（1）0开头视作八进制，0x开头视作十六进制（貌似严格模式下报错）。

（2）有小数点就视作浮点数。

（2）使用isNaN（函数）检测是否为NaN，使用isFinite（函数）检测是否在最大值和最小值之间。



**4.string**

'和"完全相同。

* + 1. 对象概述

**1.概要**

（1）js中的对象是1组无序属性的集合，即名值对。

（2）创建方式：字面量（程序编写时看到的就是字面量）、构造函数、其他方法。

（3）访问对象属性可以用点表示法或方括号表示法。

（4）in运算符可用于检查对象是否包含某个属性（注意，检查的是键名，不是键值），如果包含就返回true，否则返回false。



（5）全局对象

任何全局属性方法都是Global对象的属性。window对象也是Global对象的一部分，Global对象无法直接访问，可以直接通过window对象访问Global对象的属性。

可以用关键字this引用全局对象，但通常不必用这种方式引用全局对象，因为通过原型链最终都会从全局对象中获取全局属性方法。

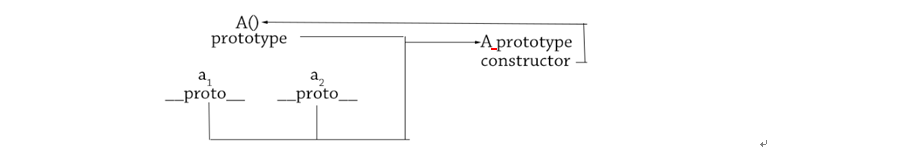
全局属性、方法有：decodeURI()、decodeURIComponent()、isFinite()、parseInt()等。

（6）内置对象

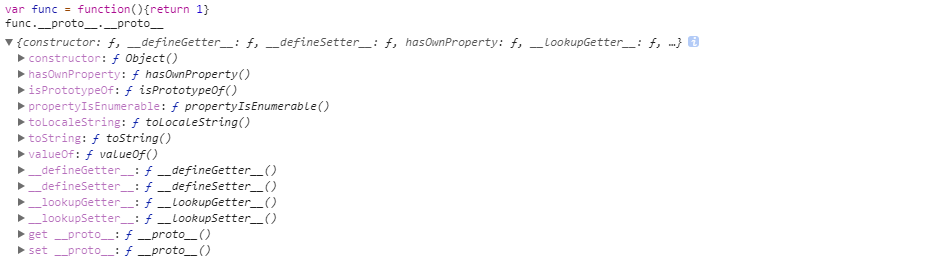
不依赖宿主环境，程序执行前即存在的对象，如Object、Math、String等都是内置对象。

**2.Function对象理解**

（1）每新建1个函数，就会自动为该函数创建1个prototype属性，该属性指向该函数的原型对象。且初始情况下，该原型对象有constuctor（构造函数）属性，指向这个新建的函数（具体见后面的面向对象）。



（2）所谓对象，就是名值对，既然函数能够添加属性，那么它就是对象。



（3）函数名就是指向函数对象的指针。不带()的函数名就是指针。

（4）1个函数对象可以有多个函数名。这些函数名都是指向同一对象的指针。

（5）可以将函数名作为参数传递。

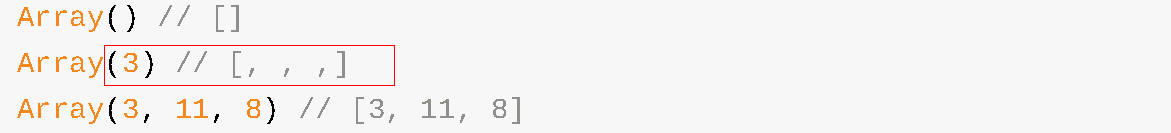
（6）为什么没有重载？因为函数名是指向函数对象的指针，两次声明会导致第2次声明的函数名指向1个新的函数对象，从而导致覆盖而非重载。

* + 1. 数组

**1.概述**

（1）array是一种特殊的对象。凡是对象有的性质，数组都有，另外，数组还有自己独特的性质。

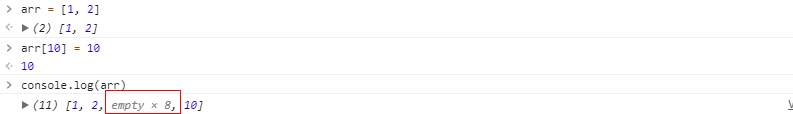
（2）有2种数组创建方法：Array()构造函数或Array()函数，均存在数字参数不一致问题；以及[]数组字面量方式。



（3）使用数组字面量方式创建数组，最后面不要有“,”，如['a', 'b',]，浏览器会存在差异，有的会是2个元素，有的会是3个元素。

（4）使用[]读取/设置，如果数字索引大于现有长度将会自动增加长度（只是length变成了11，但元素个数仍然为3个，即key为0/1/10）。





**2.length属性和稀疏数组**

（1）稀疏数组就是索引不连续的数组。

（2）数组的length属性返回的值是其数组的最大下标值+1，而不是返回数组中的元素个数，因此并不一定能真实反映数组中的元素个数。



（3）delete操作符无法改变length属性的值。



（4）只有使用Array对象自带的添加或移除元素的方法才能够改变length值，比如push/pop/unshift/shift/splice等方法。



（5）length属性可以人为更改，大于length-1下标的元素将会丢弃。



**3.类数组**

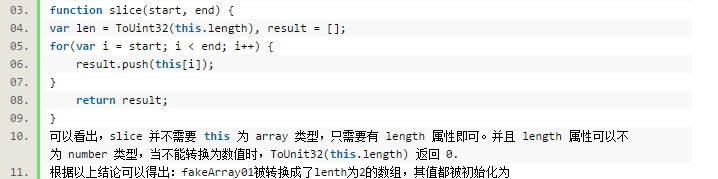
类数组对象索引必须从0顺序开始，且有length属性。类数组可以使用slice方法将其转换为数组。



类数组对象下标必须从0开始，否则无法获得正确结果。



之所以能调用slice方法，原因是array的原型中slice定义方法如下：



* 1. 语句和表达式
     1. 概述

（1）语句是为了完成某种任务而进行的操作。而表达式是为了得到返回值的计算式。区别在于，前者主要为了进行某种操作，一般情况下不需要返回值；后者则是为了得到返回值，一定会返回一个值。



上面就是1个赋值语句，其中1+3就是表达式。

（2）块语句又叫复合语句，javascript将多条语句联合在一起，形成一条复合语句。

（3）js中没有foreach语句（php才有），有for、for-in。

（4）js中没有break/continue N，只能使用标签声明语句跳出。

* + 1. 表达式

（1）原始表达式是表达式的最小单位，有字面量、关键字、变量、分组表达式等。如分组表达式实际上就是括号，用于重写运算符的优先级。

（2）复杂表达式由原始表达式和操作符组合而成。

* + 1. 运算符

**1.逻辑运算符**

|  |  |  |
| --- | --- | --- |
| **运算** | **Js** | **Php** |
| 1 && 2 | 2 | 1 |
| 0 && 2 | 0 | 0 |
| '' && 'a' | '' | '' |
| 'hello' && 'a' | 'a' | 1 |

即，js如果均为true会返回第2个值，php只返回0、1、''。

|  |  |  |
| --- | --- | --- |
| **运算** | **Js** | **Php** |
| 1 || 2 | 1 | 1 |
| 0 || 2 | 2 | 1 |
| '' || 0 | 0 | 0 |
| '' || 'a' | 'a' | 1 |
| 'a' || 'b' | 'a' | 1 |

即，js中第1个为false时，返回第2个，否则就返回第1个。

**2.关系运算符**

（1）尽量避免类型转换。

（2）<、<=、>、>=只适用于数字、字符串比较。

（3）对于对象，使用===必须是两个值引用同一对象时才相等。



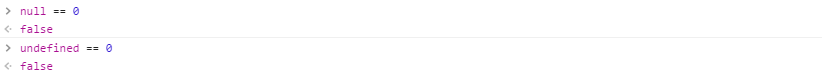
（4）对于NaN（这有它不等）、null、undefined，有：



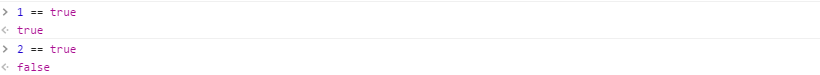
判断相等最一致的方法是Object.is()方法。

（5）尽管null、undefined在if等判断语句中可以隐式转换为false，但不会在关系运算符隐式转换为false，而是调用toString转为字符串。





（6）如果==一边有数字，true会转换为1。



**3.其他**

（1）逗号运算符中，首先计算左操作数，然后计算右操作数，最后返回右操作数的值。用逗号运算符可以在一条语句中执行多个运算。



* 1. 函数
     1. 概述

**1.函数属性**

（1）name属性：返回函数的名字。



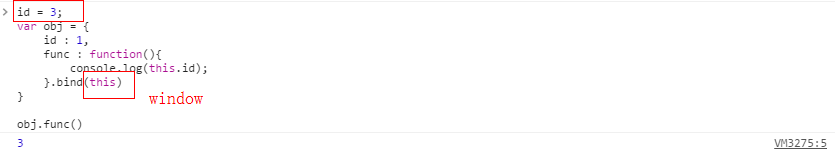
（2）length属性：希望接受的参数个数（形参个数）。

**2.函数方法**

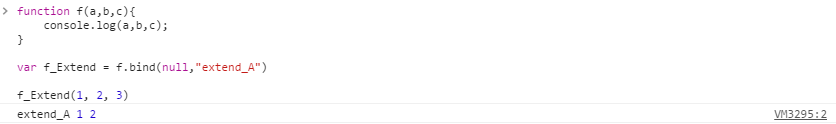
（1）静态方法：函数也是对象，直接给函数添加属性/方法就是静态属性/方法。

（2）apply、call、bind。bind使用如下：

跟call类型，都是改变函数执行时this指向，但区别在于一是bind返回函数，call就立即执行了。



另一个区别在于bind传参也会绑定。



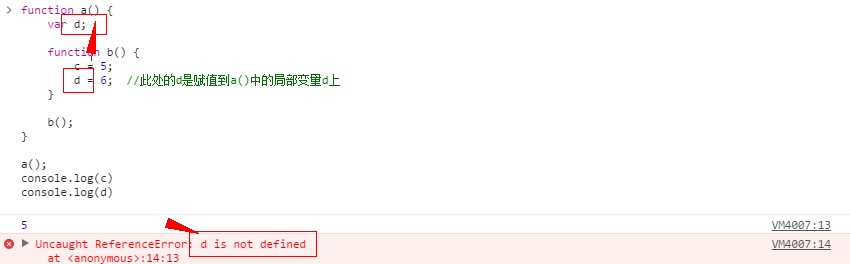
即，等同于



* + 1. 作用域链

（1）js中函数嵌套比较常见。对于1个变量，js是由内向外寻找，直到window。

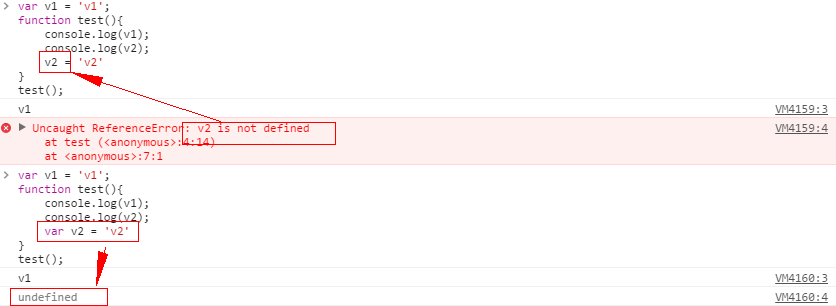
（2）变量声明时如果没有加var，则仅作为1个赋值操作。赋值时仍然向上寻找变量声明。



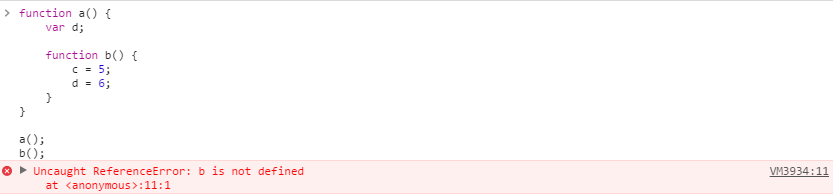
执行a函数时，c、d都会向上查看是否有声明，其中d被a拦截，c则一直查找到window，作为window的属性。此时输入window.d只是undefined但不会报错。



（3）var声明变量会在词法分析时提升。



（4）在函数内部嵌套声明的函数，外部无法访问，不会将声明提升外部，即使外部函数执行也不行。



* + 1. 词法分析

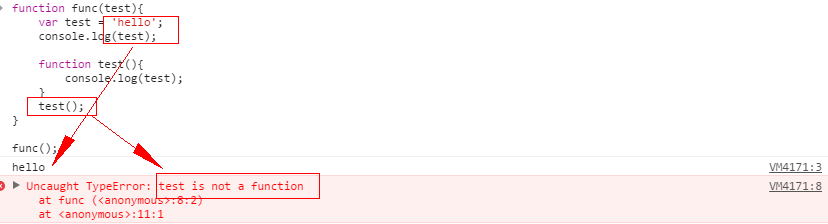
分为3步：首先分析参数，再分析变量声明，在分析函数声明。具体步骤为：

（1）函数运行的一瞬间生成活动对象（active object）。

（2）将形参作为活动对象的属性名，将形参值作为对应属性值。

（3）分析函数体内变量声明（var），如果AO已有该属性（形参），不做任何处理。如果没有，添加到AO上，但不赋值（undefined）。

（4）分析函数声明，将函数名添加到AO上，如果之前已存在，覆盖。



具体过程为：

（1）首先创建AO={}。

（2）然后添加形参属性AO={test:undefined}。

（3）分析变量声明，因为已经有test属性，不影响。

（4）分析函数声明，更改为AO={test:function(){}}。

但是，在执行过程中，有：

（1）对test赋值，更改AO为{test:hello}。

（2）test为字符串，因此报错。

* + 1. 函数声明和函数表达式

（1）js是函数式语言，函数可以作为参数传递。

（2）函数声明和函数表达式区别在于声明会在词法分析时提升到顶部。

* + 1. 函数的内部属性

函数内部属性有arguments和this，使用时不需要在前面指定调用方。

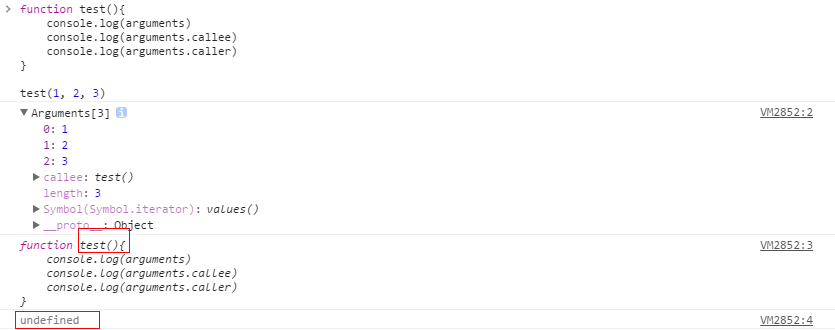
**1.arguments**

（1）arguments是1个类数组对象，保存函数运行时的实参列表，索引为0、1、2...，与实参逐个对应（不是关联索引）。

（2）arguments.length为收到的实参个数。

（3）实参与arguments元素对应，修改1个能改变另外1个。

（4）有callee属性，指向拥有该arguments的函数即函数自身，有caller函数指向调用当前函数的函数。



**2.this绑定**

有4种绑定方式：

（1）普通函数调用，此时this指向window。



（2）作为对象的方法调用，此时this指向方法的调用者即obj对象。



又如yy调用test方法是显示为yy。即，不管声明在何处，只管调用时被谁调用。



（3）作为构造函数

根据此时创建对象的过程，this会绑定到新创建的那个对象上。

（4）函数使用apply、call调用

注意：this也不能沿着AO链往上查找。

* + 1. 闭包

**1.特性**

（1）函数嵌套函数。

（2）内部函数可以引用外部函数环境。

（3）内部函数参数和变量不会被垃圾回收机制回收。

**2.优缺点**

（1）优点是可用于设计私有的方法和变量。

（2）缺点是常驻内存，会增大内存使用量。

**3.形式**

（1）形式1：直接返回1个函数。



在大多数程序语言中，func1执行时申请内存，并将局部变量push栈中。func1指向完毕，内部局部变量随函数销毁而销毁。而在js中，func1执行时生成了func2，func2又能在作用域链上访问到age。func1指向完毕后，age并没有消失，而是随func2保存到func2的环境包中，这个环境包只有func2能访问到，因此又叫闭包。此时func2不是1个孤立的函数，而是伴随着1个环境。

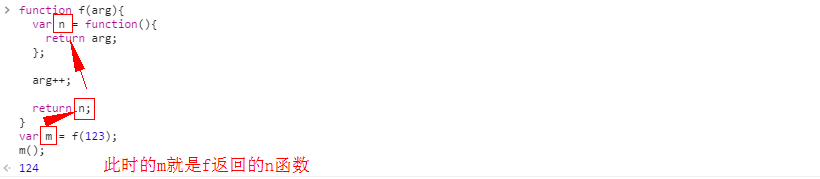
即：函数的作用域链不取决于函数调用时，而取决于函数声明时。函数可以“记忆”它被创建时候的环境。

（2）形式2：将函数赋给1个变量，然后返回这个变量。



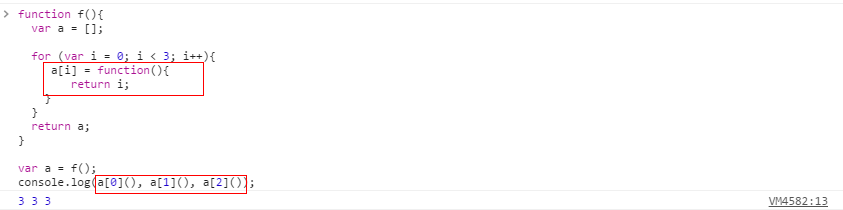
因在f内部只是将闭包赋值给全局n，但仍然可以访问闭包环境。

（3）形式3：仍然是返回1个变量。只是又将这个返回的函数变量又赋值给另外1个变量。



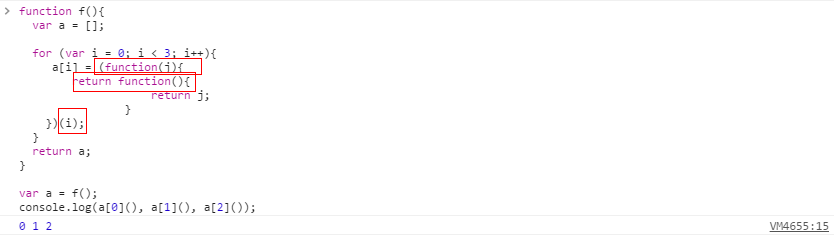
n声明和赋值的时候只是给了个地址，return时arg已计算了值。记住，闭包绑定的是整个环境。

（4）形式4：将多个函数保存到1个数组中，返回返回这个数组。

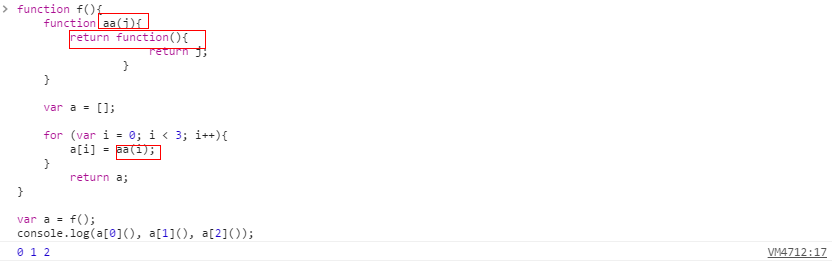


a数组元素分别被赋值了3个闭包，但3个闭包都指向了同一个局部变量i。**但闭包不会记录变量值，而只是1个引用。**i最终的值为3，因此都返回3。

解决方法1：利用立即执行函数，将i赋值给局部变量j，立即指向函数应该在返回1个函数。

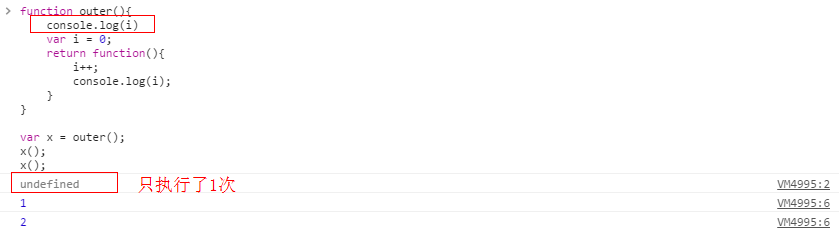


解决方法2：不使用立即执行函数，但仍然是通过执行1个函数，将i赋值给执行函数的局部变量。

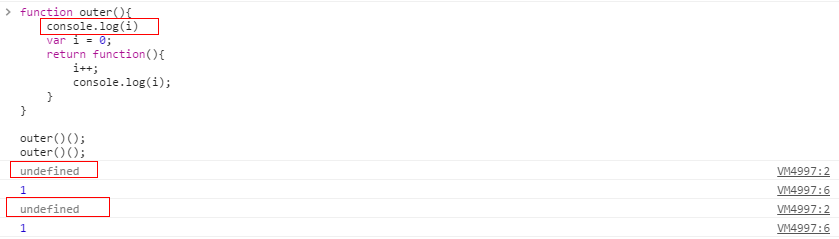


**4.应用**

（1）局部变量计算器



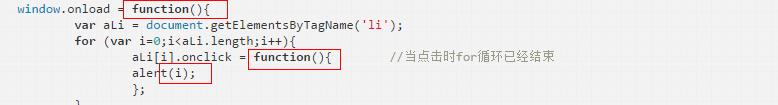
如果像下面这样使用，结果又不一样。原因是上面只生成了并且调用的是同1个闭包，下面生成了3个闭包。



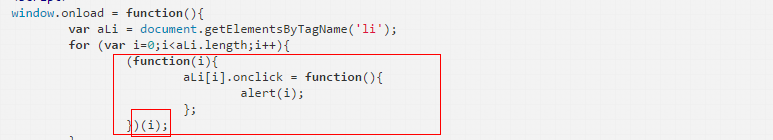
（2）模拟私有变量



（3）绑定dom



上面只会绑定aLi.length-1这个节点事件。



**5.销毁闭包**

将闭包设为null。

* 1. 面向对象
     1. 创建对象模式

**1.字面量/Object构造函数**

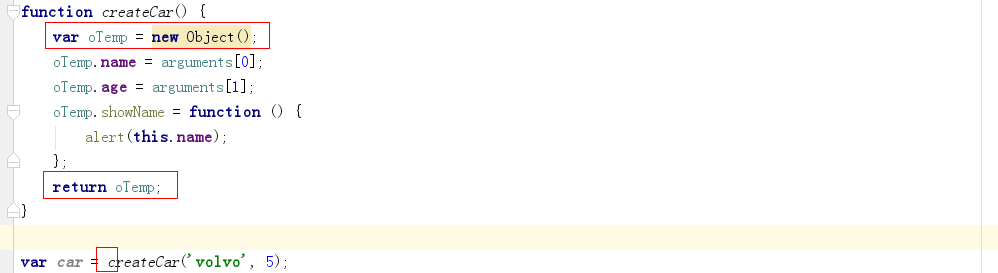




（1）Object后面的()可以省略。

（2）每次创建对象，都要写属性名/值，不同对象不能复用，导致重复代码太多。

**2.工厂模式**

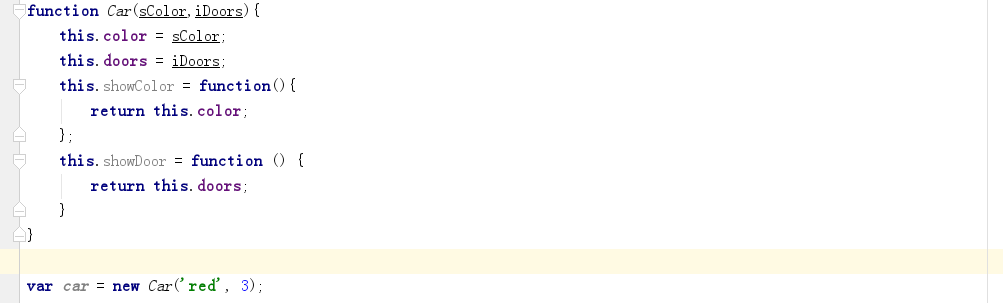


（1）将创建对象的过程包装在1个函数中，通过入参接口返回不同的对象。

（2）优点：减少代码量。

（3）缺点：1）无法识别对象类别（创建对象不用new）。2）每个方法都是不同实例。

**3.构造函数模式**



创建过程：（1）new出1个新的空对象。（2）将构造函数的作用域指向这个对象。（3）执行构造函数。（4）返回新对象。

优点：通过new构造函数创建，因此能够识别对象类型。

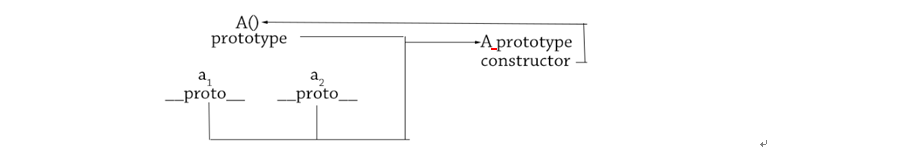


缺点：不同实例的方法都不同。解决方法：将方法从构造函数中移到外面，但这样1是破坏语义，2是方法成了全局的了。

**4.原型模式**

（1）原理

* 每新建1个函数，就会自动为该函数创建1个prototype属性，该属性指向该函数的原型对象。
* 初始情况下，该原型对象除继承Object属性外，还会为该原型对象生成constuctor（构造函数）属性，指向新建的函数。
* 用new这个函数创建出来的实例，都会有\_\_proto\_\_属性，指向原型对象（不是指向构造函数，实例和构造函数没有直接联系）。



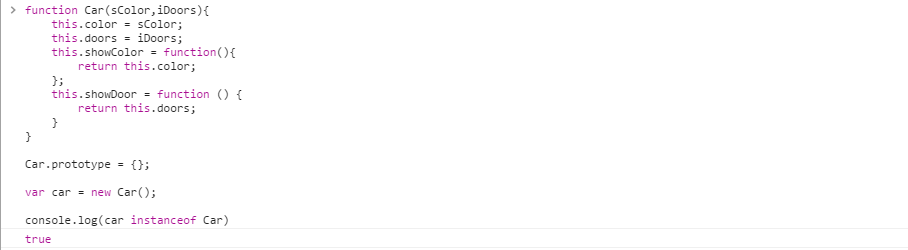
* 对象是new创建的，并非函数创建的。实例通过\_\_proto\_\_指向原型建立间接关系，因此，给函数添加静态方法实例无法访问。
* 可以通过A.prototype.func=?创建方法，实例能够访问。
* 可以通过hasOwnProperty()检测是自身还是原型对象属性。



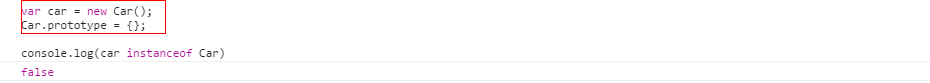
* 可以通过in/for-in判断/获取对象可访问属性（无论是自身还是原型中）。



* 如果重写整个原型，将导致原型的constructor不再指向构造函数，而是指向Object的构造函数。



* 如果实例化后再重写原型，将会导致实例无法访问原型属性/方法。



上面2种情况原型无法通过其constructor属性获取构造函数，但前者instanceof没问题，说明不是用的constructor来判断。

（2）创建对象

将所有属性定义在原型对象上，导致所有实例共用同一个属性。

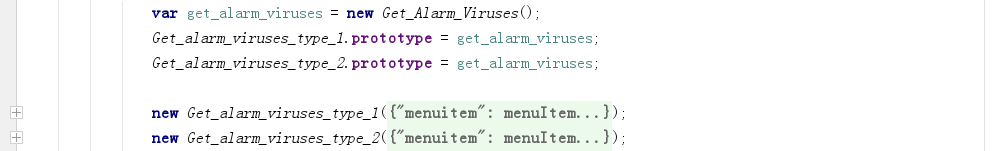
（3）构造函数+原型对象

属性定义在构造函数中，方法定义在原型中。

* + 1. 继承

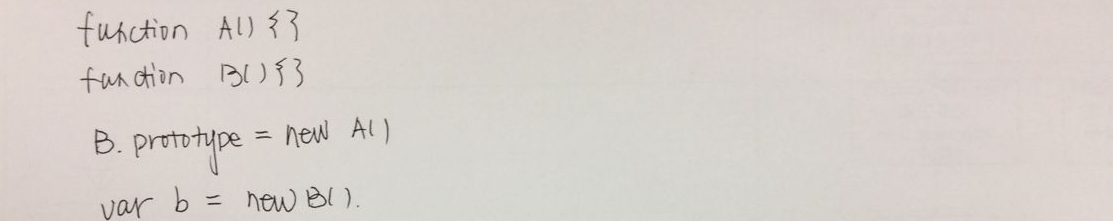
（1）原型链继承

重写1个构造函数的原型对象，使其成为另1个构造函数的实例，使得实例的\_\_proto\_\_指向另1个构造函数实例，进而能够访问这个实例的属性、方法。

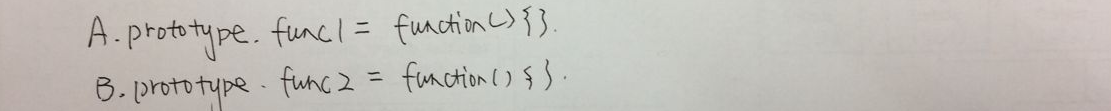




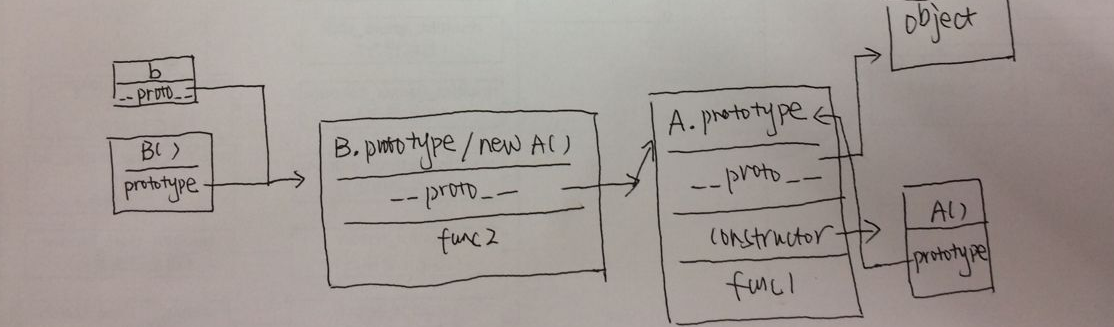
（2）完整原型链



B默认的原型为object，现在重写为A的实例。



给B的原型即A的实例添加方法，是在对象上的方法。



* 实例b的\_\_proto\_\_和构造方法B的prototype都为A的1个实例。
* 因为重写B的prototype，因此它没有constructor。
* A的prototype（是1个对象）的\_\_proto\_\_指向object。



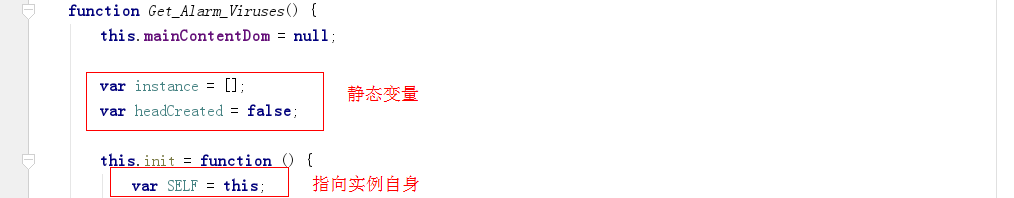
（3）借用构造函数继承（伪对象继承）



因为方法都是在构造函数中定义，因此仍然无法复用。

* + 1. 总结

（1）在原型内的this为实例自身。但变量可以视为静态变量。



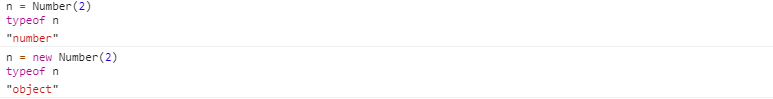
（2）原型中的属性为所有子类所共享，但是，通过1个方法返回这些属性值，可以实现每个子类共用一段代码但同时属性值又不相同的效果。



1. 标准库
   1. 包装对象
      1. 概述

**1.包装对象**

（1）包装对象，就是数值、字符串、布尔值对应的Number、String、Boolean三个原生对象。



（2）js中设计包装对象的目的，首先是使js中的对象涵盖所有的值。其次，可以使简单类型的值可以方便地调用特定方法。

**2.临时包装对象**

当数值、字符串、布尔值在它们调用方法时，也可以转换为对象，即简单类型的临时包装对象。如下：



当时有a.substring(0)时，会经过如下过程：

（1）创建String类型的1个实例：var a = new String('a')

（2）执行方法a.substring(0)。

（3）销毁该实例a=null。

引用类型和基本类型包装对象区别在于引用类型一直存在于内存中。基本包装类型只存在于代码执行过程的一瞬间。

* + 1. Boolean对象

（1）Boolean对象是js的三个包装对象之一。作为构造函数，它主要用于生成布尔值的包装对象的实例。



（2）Boolean还可以单独使用，作为1个工具方法，将任意值转为布尔值。





注意：使用双重的否运算符（!）也可以将任意值转为对应的布尔值。

* + 1. Number对象

（1）toFixed方法用于将一个数转为指定位数的小数，返回这个小数对应的字符串。



（2）ES6新增Number.isFinite()和Number.isNaN()方法，检查是否为Infinite和NaN。

（3）ES6将全局方法parseInt()和parseFloat()，移植到Number对象上面，即Number.parseInt()和Number.parseFloat()。

（4）ES6新增Number.isInteger()判断一个值是否为整数。注意，如同3和3.0在js中被视为同一个值。

（5）ES6新增Number.isSafeInteger()判断一个整数是否落在这个范围之内。

* + 1. String对象

**1.说明**

字符串的包装对象是一个类似数组的对象。



**2.属性**

length属性：返回字符串的长度。

**3.方法**

（1）获取特定字符方法

可以使用CharAt，也可以使用[]。



（2）查找子字符串位置

indexOf/lastIndexOf：查找方向不一样，但值都是从前往后的顺序。如果没有找到返回-1。



search(reExp)：传入正则表达式，从前往后匹配，返回首次出现位置。如果没有找到返回-1。



（3）拼接字符串

可以使用concat和+。



（4）截取子字符串

slice(start[,end])：如果没有end就为字符串长度。如果传入负值，使用length+负值（start/end都是）。不包含end。不建议使用substring。

substr(start[,length])：如果传入负值，start为length+负值，length视同0。

（5）正则匹配方法

match()基本同正则的exec()方法，只是调换了下位置。如果不设置全局标志，match()方法和exec()方法结果相同。设置全局标志后，exec()方法依然返回单次的匹配结果，而match()方法会返回一个字符串数组，其中包括各次成功匹配的文本。



（6）替换

replace(regexp/substr,replacement)，可以使用正则。正则如果设置全局标志g，替换所有匹配的子字符串。



（7）其他

split(separator,howmany)：分割为数组。拼接数组为字符串为join。



trim：去除字符串两端的空格（各种不可见字符），返回一个新字符串。

repeat方法：返回一个新字符串，表示将原字符串重复n次。



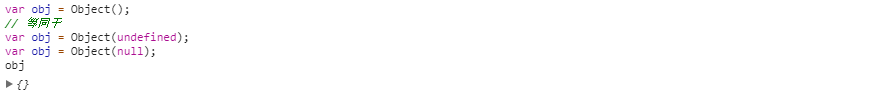
toLowerCase()/toUpperCase()：大小写转换。

* 1. Object对象
     1. 概述

**1.作为函数**

（1）Object本身是一个函数，可以当作工具方法使用，将任意值（入参）转为对象。

1）如果入参为空、undefined、null，则返回1个空对象。



2）如果入参是Number、String、Boolean这3种简单类型，将返回对应的包装对象实例。



3）如果入参就是对象，如Array、Function等，直接返回。



**2.作为构造函数**

可以结合new作为构造函数，入参用法和作为工具方法函数一样。



如上，还是同一个对象。

**3.定义在Object上的静态方法**

即，定义在Object函数自身上的方法，如Object.keys()、Object.getOwnPropertyNames()等。

**4.Object的实例方法**

即，定义在Object.prototype对象上的方法。



（1）构造函数

指向Object()方法。

（2）hasOwnProperty()

判断某个属性是否为当前对象自身的属性，还是继承自原型对象的属性。

（3）isPrototypeOf()

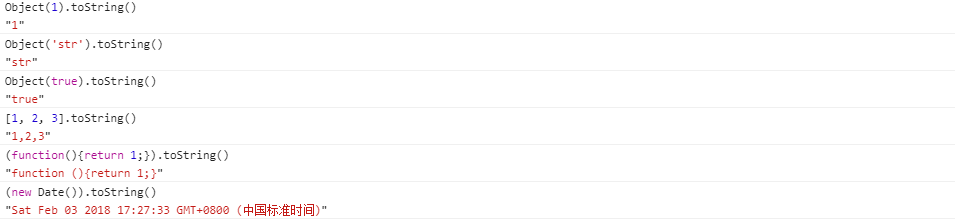
判断当前对象是否为另一个对象的原型。

（4）propertyIsEnumerable()

判断某个属性是否可枚举。

（5）toString()

1）对象实例调用（包括临时包装对象）返回一个对象的字符串形式。



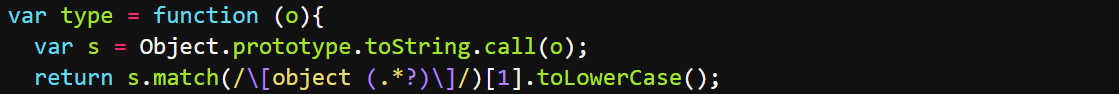
2）可以通过自定义toString方法，可以让对象在自动类型转换时，得到想要的字符串形式。



3）使用Object.prototype.toString方法判断一个值的类型。



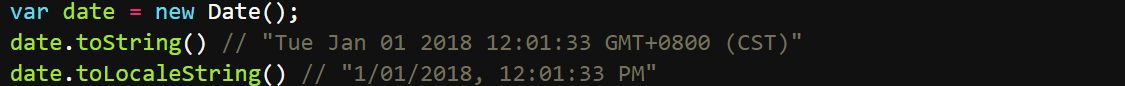






（6）toLocaleString()

与toString的返回结果相同，主要作用是留出一个接口，让各种不同的对象实现自己版本的toLocaleString，用来返回针对某些地域的特定的值。日期的实例对象的toString和toLocaleString返回值就不一样。



（7）valueOf()

返回对象本身。



主要用途是，js自动类型转换时会默认调用这个方法。



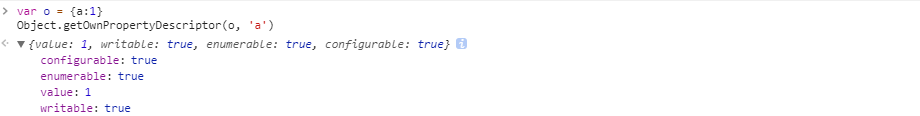
如果自定义了obj对象的valueOf方法，如下：

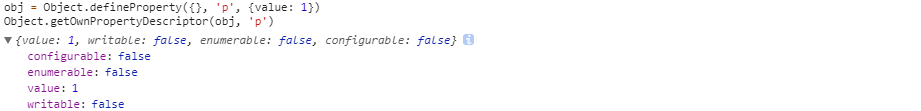


* + 1. 属性描述对象

**1.4个元属性**

js内提供了1个描述对象属性特性的数据结构，即属性描述对象。注意，通过字面量定义时，3个布尔值都为true，而通过defineProperty则都为false。





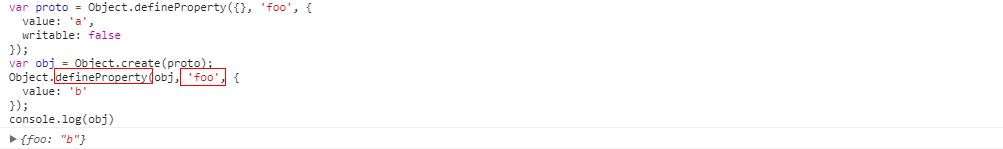
（1）value

该属性的属性值，默认为undefined。

（2）writable

表示属性值是否可改变（即是否可写）。当writable为false，正常模式下，修改属性不会报错，只会默默失败。但是，严格模式下会报错。

如果父对象某个属性的writable为false，那么子对象也无法对这个属性赋值，但可以通过在子对象新定义这个及属性描述对象，覆盖父元素属性。

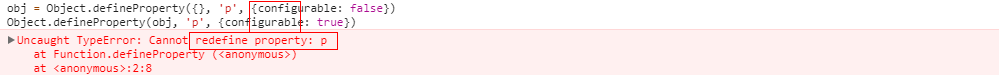


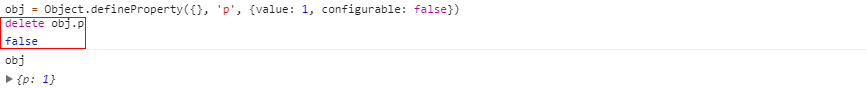
（3）enumerable

表示该属性是否可遍历。如果设为false，for..in循环、Object.keys方法、JSON.stringify方法这3个操作无法获取该属性。

（4）configurable

用于控制属性描述对象的可写性。如果设为false时，value、writable、enumerable和configurable都不能被修改（包括configurable），而且，属性也不能被删除。





但有如下例外：

1）writable从false改为true会报错，从true改为false允许。

2）只要writable和configurable有一个为true，value就可以修改。





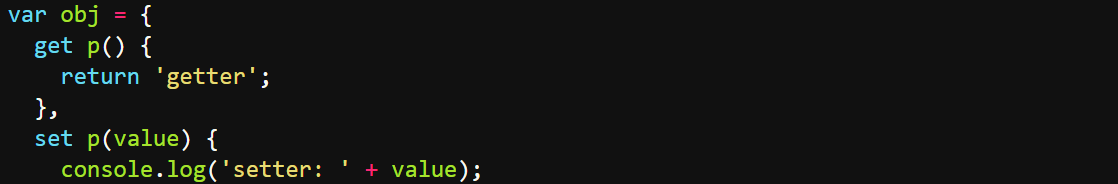
**2.存取器**

（1）除了直接定义属性外，属性还可以用存取器定义。其中，存值函数称为setter，使用属性描述对象的set属性；取值函数称为getter，使用属性描述对象的get属性。





（2）还可以使用如下写法：



**3.获取属性描述对象**

使用Object.getOwnPropertyDescriptor方法可以获取属性描述对象。注意，只能用于对象自身的属性，不能用于继承的属性。



* + 1. 定义对象



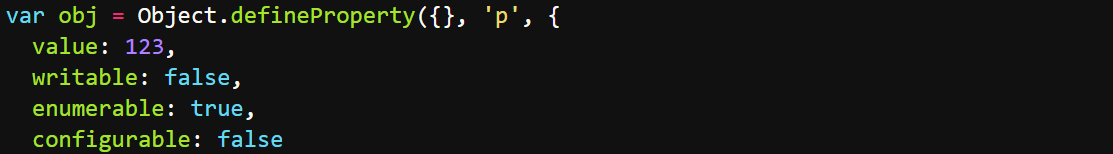
propertiesObject为属性描述对象。

* + 1. 属性相关

**1.增加属性（也可以用于定义对象）**

（1）要想定义/修改1个对象，且能定义该对象的属性描述对象，可以通过Object.defineProperty()和Object.defineProperties()方法。

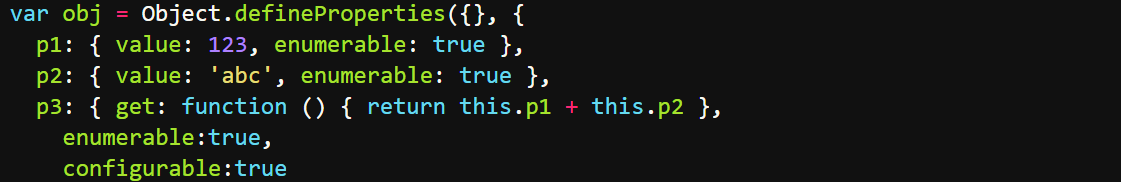
（2）Object.defineProperty方法接受三个参数：对象（定义或修改）、属性名、属性描述对象。



注意，这里的对象为{}，这是新建1个对象的写法。因设置writable为false，因此不能更改value。



（3）可以使用Object.defineProperties方法一次性定义或修改多个属性。



**2.获取属性名**

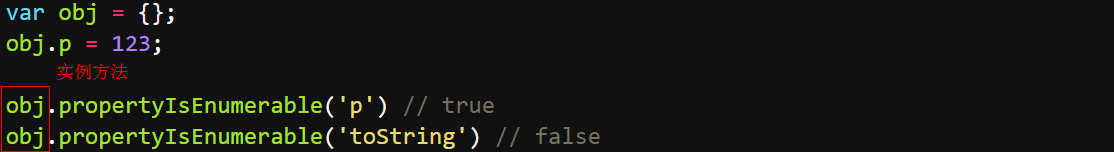
（1）for...in：循环遍历对象自身的和继承的可枚举属性。

（2）Object.getOwnPropertyNames：返回一个数组，不包括继承属性，但包括不可枚举属性。

（3）Object.keys()：对象自身的（不含继承的）、可遍历（enumerable）的属性键名

**3.判断属性是否可遍历**

对象实例方法propertyIsEnumerable()返回一个布尔值，用来判断某个属性是否可遍历。



* + 1. 遍历对象

**1.Object.keys()、Object.values()、Object.entries()**

返回对象自身的（不含继承的）、可遍历（enumerable）的属性键名/键值/键名值对数组。



**2.其他获取属性名方法**

（1）for...in：循环遍历对象自身的和继承的可枚举属性。

（2）Object.getOwnPropertyNames：返回一个数组，不包括继承属性，但包括不可枚举属性。

* + 1. 对象拷贝（合并）

**1.原始方法**

（1）初步方法如下：



有2个问题：

1）如果from是多层引用结构，尽管from所有层级的属性都拷贝到to中，但存在from和to引用的是同1个对象问题。

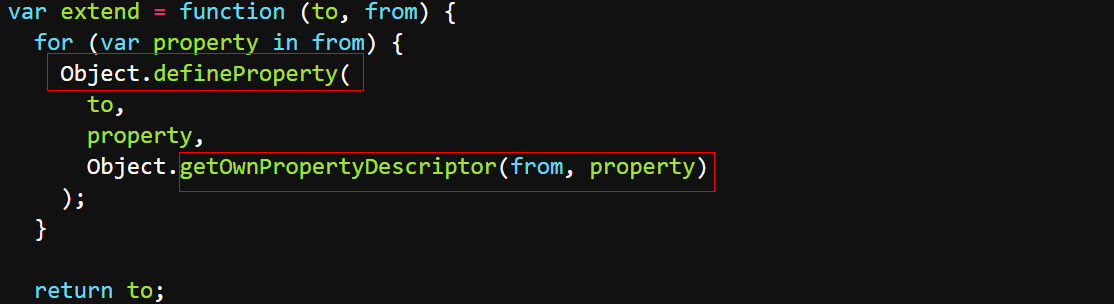


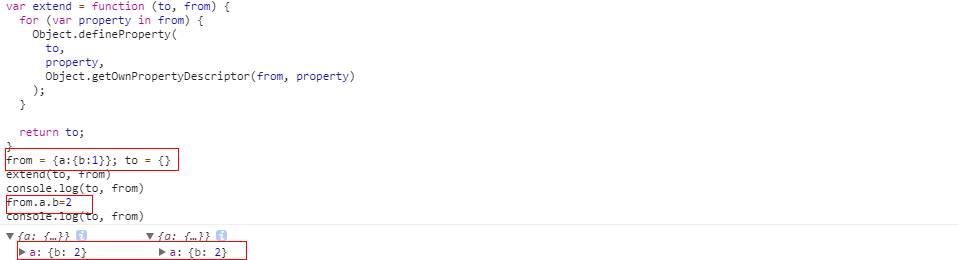


2）如果遇到存取器定义的属性，会只拷贝值。



（2）可以通过Object.defineProperty方法来拷贝属性，但还是存在浅拷贝问题。





（3）浅拷贝问题需要通过递归拷贝解决。



**2.Object.assign()**



（1）Object.assign是浅拷贝，而不是深拷贝。

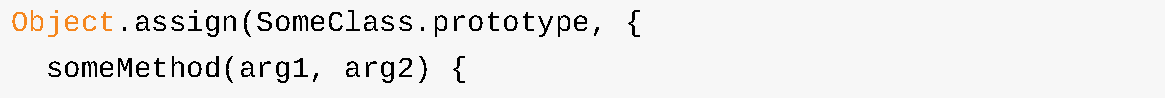
（2）只拷贝对象自身属性，不拷贝继承属性，也不拷贝不可枚举的属性。

（3）Object.assign可以用来处理数组，索引值相同的会覆盖。

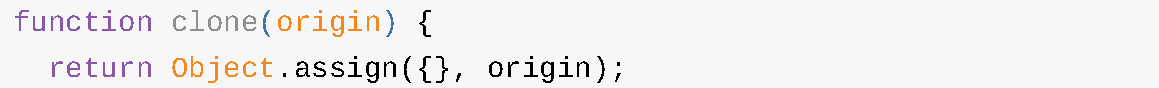
常见用途有：

（1）添加属性、方法。





（2）克隆对象。



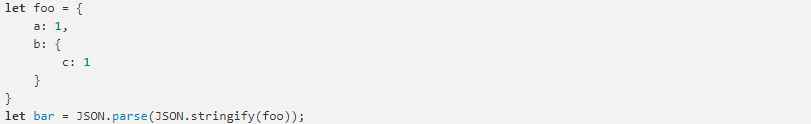
（3）合并多个对象。

（4）为属性指定默认值。



**3.最简单粗暴的方法**

使用JSON对象的stringify()和parse()，但会破坏原型链，且无法拷贝属性值为function的属性。前提是支持JSON对象。



* + 1. 判断相等

（1）尽管使用===可以避免触发强制类型转换的行为。但也不完全准确。



（2）ES6引入了Object.is()方法来弥补全等运算符的不准确运算。







* + 1. 原型相关

（1）\_\_proto\_\_

标准要求尽量不要使用这个属性，而是使用下面两个。

（2）Object.getPrototypeOf()

读取一个对象的原型对象。

（3）Object.setPrototypeOf()

可以任意指定对象的原型。接受两个参数：被改变原型的对象及替代第一个参数原型的对象。





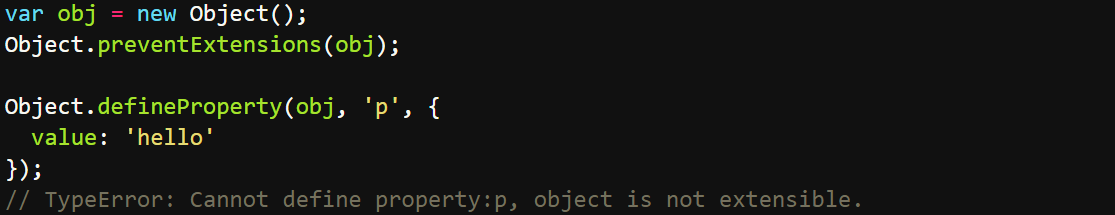
* + 1. 控制对象状态

有时需要冻结对象的读写状态，防止对象被改变。js提供了3种冻结方法，最弱的一种是Object.preventExtensions，其次是Object.seal，最强的是Object.freeze。

**1.操作**

（1）Object.preventExtensions()

对象无法再添加新的属性。



Object.isExtensible方法用于检查一个对象是否使用了Object.preventExtensions方法。也就是说，检查是否可以为一个对象添加属性。

（2）Object.seal()

使得一个对象既无法添加新属性，也无法删除旧属性。



Object.isSealed方法用于检查一个对象是否使用了Object.seal方法。如果Object.isSealed方法返回true，Object.isExtensible方法也返回false。

（3）Object.freeze()

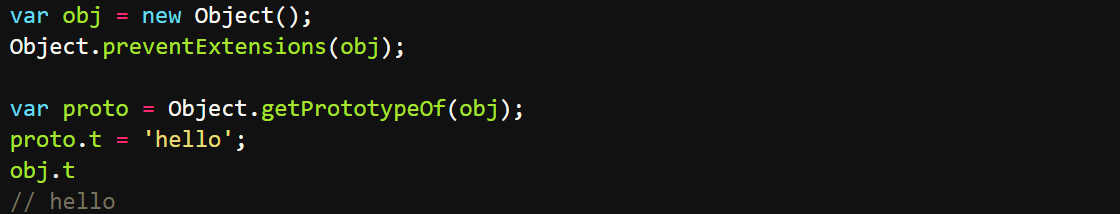
使得一个对象无法添加新属性、无法删除旧属性、也无法改变属性的值，使得这个对象实际上变成了常量。



Object.isFrozen方法用于检查一个对象是否使用了Object.freeze方法。同样也会影响Object.isSealed方法和Object.isExtensible方法。

**2.局限性**

（1）3个方法都能使对象无法新增属性，但是，可以通过在对象的原型上增加属性来规避。



解决办法是将原型也冻结。

（2）通过上述方法限制对象无法修改也只是限制属性所指的对象，如果对象再引用其他对象则无能为力。

* 1. Array对象
     1. 构造函数

Array可以用作构造函数，生成1个新的数组。但因其不同参数生成数组的行为不一致，有重大缺陷，不建议使用。

* + 1. 静态方法

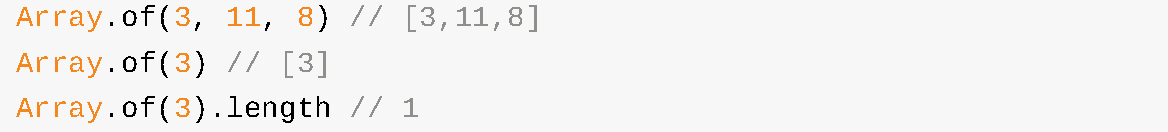
**1.Array.isArray()**

返回一个布尔值，表示参数是否为数组。它可以弥补typeof运算符的不足。

**2.Array.of**

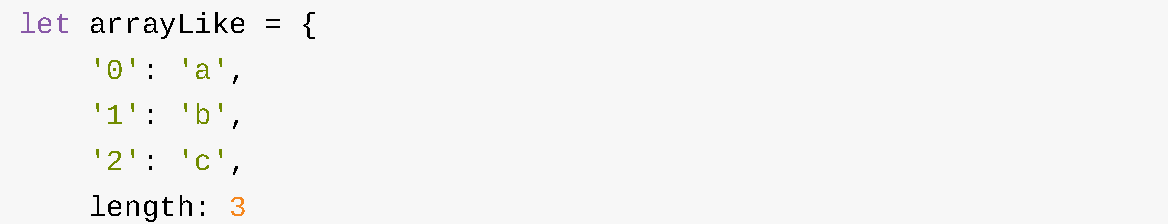
（1）主要用来规避通过Array构造函数创建数组时的怪异行为。

（2）传入一组数值，可以按照一致行为生成数组。

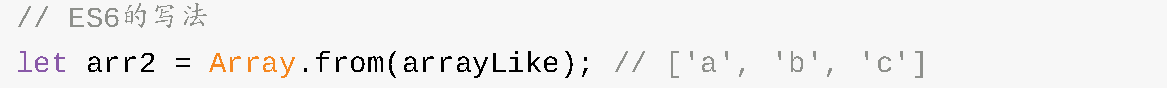


**3.Array.from**

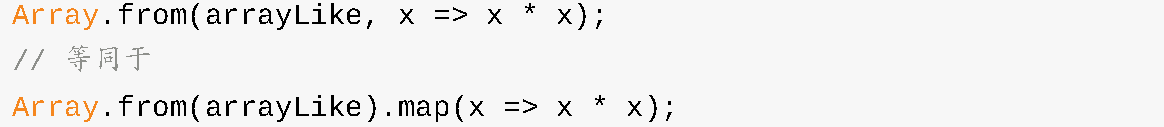
（1）可将类数组对象和可遍历对象转为真正的数组。







（2）映射转换：Array.from还可以接受第二个参数，作用类似于数组的map方法，用来对每个元素进行处理，将处理后的值放入返回的数组。



* + 1. 实例方法

**1.生成、转换相关**

（1）转为字符串

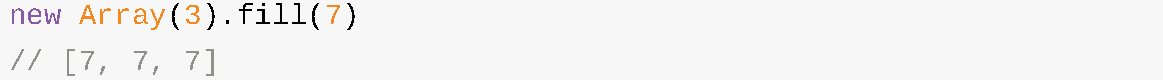
使用toString转为,分割的字符串。使用join转为自定义分隔符的字符串。



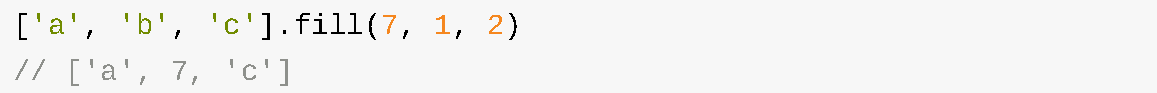


（2）fill

用于填充一个数组，或重新数组一部分或全部。



可以接受第2个和第3个参数，用于指定填充的起始位置和结束位置。



（3）copyWithin

复制自身部分元素，填充到自身部分地方。需要传入两个参数：一个是该方法开始填充值的索引位置，另一个是开始复制值的索引位置。默认会一直复制直到数组末尾的值，可以提供可选的第三个参数来限制被重写元素的数量。



**2.查找相关**

（1）indexOf/lastIndexOf

可以使用indexOf和lastIndexOf获取元素在数组中的位置。没有找到返回-1（同字符串）。



注意，indexOf/lastIndexOf不能判断NaN。

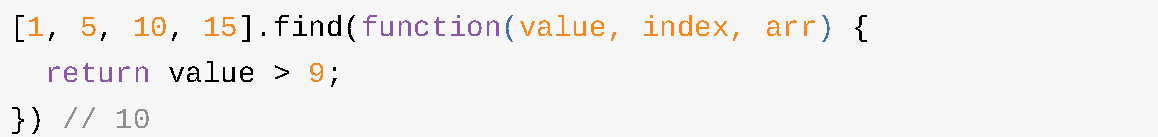
（2）includes

返回一个布尔值，判断某个数组是否包含给定的值（同字符串）。



（3）find/findIndex

接受1个回调函数，入参分别为value、index、arr。如果回调函数返回true时，停止编译后面的部分。其中find返回此时的value，findIndex返回此时的index。





原来的indexOf方法无法识别数组的NaN成员，findIndex方法可以借助Object.is方法做到。



**3.栈/队列方法**

push/pop/unshift/shift。

**4.拼接、截取、替换、删除**

（1）数组拼接

使用concat，返回1个新数组。参数是一个或多个数组。如果不传值，相当于复制了当前数组。



（2）截取

使用slice截取，同字符串。原数组不变，返回截取元素组成的数组。

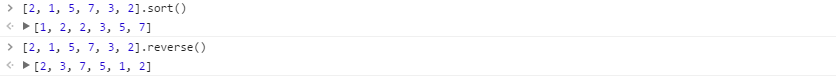
（3）删除/替换

splice，接受2个/3个以上参数，分别为起始位置，要删除的个数，第3个开始为插入项。返回的是删除的元素组成的数组。



**5.排序**

使用reverse/sort方法。但sort方法并非按数字排序，而是按ascii码排序，如1，10，2排序。



**6.遍历**

（1）map

将函数结果返回一个新数组。注意，不改变原数组。



如果数组有空位（不会跳过undefined和null，只会跳过空位），map方法的回调函数在这个位置不会执行，会跳过数组的空位。

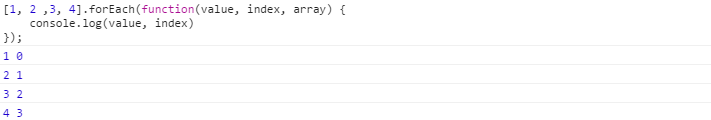


map还可作用于字符串，但不要使用call方法间接使用。



（2）forEach

用来执行操作数组成员，没有返回值，其他同map。



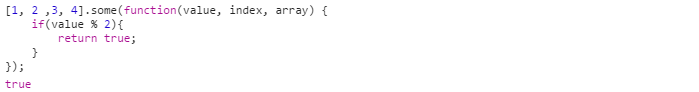
（3）filter

返回新数组，不会改变原数组。



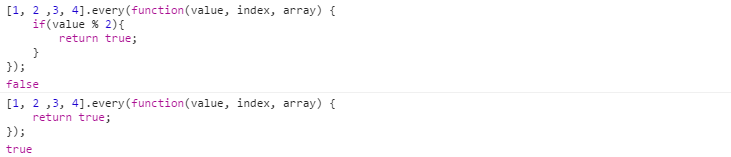
（4）some

用于断言，只要有1个满足条件返回true，即不再遍历。



（5）every

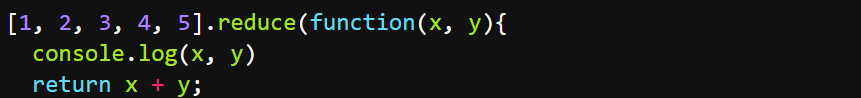
用于断言，所有的都要满足条件才会返回true。



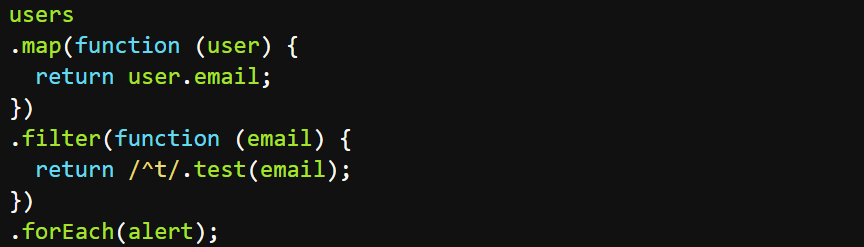
（6）reduce/reduceRight

依次处理数组的每个成员，最终累计为一个值返回，不影响原数组。reduce是从左到右处理（从第一个成员到最后一个成员），reduceRight则是从右到左（从最后一个成员到第一个成员），其他完全一样。

参数都是一个函数，接受4个参数：累积变量、当前变量、当前位置、原数组。



另外，因map、filter返回1个数组，因此可以链式调用。



（7）entries、keys和values方法

均返回1个迭代器。



【总结】

（1）栈/队列方法、删除/替换方法，既影响原数组，又有新东西，所以返回的不是原数组。没有新东西直接作用原数组，就返回原数组。有新东西但不影响原数组，就返回新东西。记住栈/队列方法、删除/替换方法的返回即可。

（2）除栈/队列方法、排序方法以外，其他（包括遍历方法）均不影响原数组。

（3）和字符串方法比较，均有indexOf、includes、concat、slice、split方法。

（4）数组乱序

没有原生方法，可以使用如下方法实现：



* 1. Date对象

**1.作为普通函数调用**

Date对象可以作为普通函数直接调用，返回一个代表当前时间的字符串。



即使有参数，返回的也是当前时间。



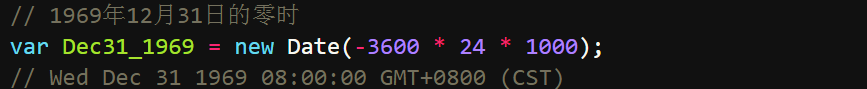
**2.作为构造函数使用**

（1）不传参为当前时间。



（2）传入与UTC标准时间间隔毫秒数。





可以使用Date.parse（使用见后）转换与UTC标准时间间隔毫秒数。

（3）传入日期时间字符串，凡是能被Date.parse()方法解析的日期字符串，都可以当作Date对象的参数。



（4）new Date(yyyy,mth,dd,hh,mm,ss)，至少传入2个参数，否则当作据标准时间间隔。mth从0开始，其他从1开始。后面没有传参的当作0。



**3.静态方法**

（1）Date.now()

返回当前距离1970年1月1日00:00:00的UTC的毫秒数（Unix时间戳乘以1000）。



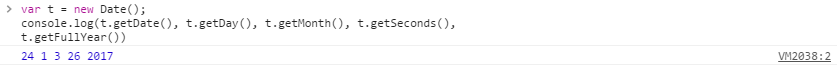
（2）Date.parse()

传入日期或时间字符串，格式为YYYY-MM-DDTHH:mm:ss.sssZ，其中Z表示时区。返回时间戳。解析失败，返回NaN。



**4.实例方法**

（1）get系列方法（其中getDate获取日期；getDay获取星期；getMonth从0开始，如4月显示为3，getTime为据标准时间的毫秒数）。



（2）set系列方法（缺省时间为当前值）。



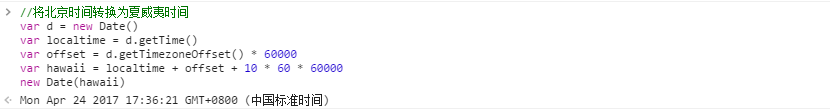
**5.时区问题**

（1）默认为客户端时区。

（2）可用getTimezoneOffset获取当地时间和GMT时间（格林威治时间）的差值，返回值以分钟为单位。



（3）如果想显示某个时区时间，需要先获取当地时间与GMT时间差值，然后再转到设定时区。



* 1. RegExp对象

**1.创建**

js中有2种正则写法：

（1）字面量写法，为perl语法形式，不存在双重转义。



（2）RegExp构造函数方式，因为partten为字符串（字符串转义后就没\了），需要双重转义。

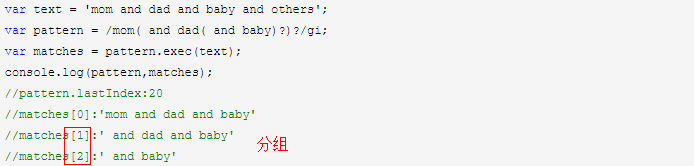




匹配模式支持下列3个标志：g/global：查找所有而非第一个匹配，ignoreCase/i，multiLine/m。

**2.实例方法**

（1）exec，永远只返回第1个匹配项，无论是否有全局修饰符。



与str.match关系：



即，str.match在全局匹配下返回只是匹配项的数组。如果不是，则跟exec返回相同。

（2）test：检索是否存在指定值，true或false。

**3.字符串方法中与正则相关方法**

match、search、replace、split。

**4.总结**

（1）原子：最小的匹配单位，所有未指定为元字符的打印和非打印字符。在1个正则表达式中，至少有1个原子，写在定界符中。

（2）转义符可以让有意义的元字符转义为没有意义的原子，也可以将没有意义的原子转义为有意义的元字符如\w。

（3）元字符：不能单独使用，用来扩展原子功能的，修饰或限制原子，写在定界符中。

（4）模式修正符：对整个正则表达式进行修饰，写在定界符外面。其中，i大小写；m多行，如果没有m，即使模式中有换行符也不管用，如.就可以匹配换行符了。

（5）原子表：即[]中的原子列表或范围，如[123]或[1-3]，[^1-3] 。

（6）注意，出现次数，有{m}、{n,m}、{n,}，但没有{,m}。因为不可能为负数，需要写作{0,m}。

（7）|：或的关系，至于两边匹配多远，涉及优先级，|的优先级最低，因此匹配两边全部原子，如mysql|oracle，要想匹配l或o，则要用mysq(1|o)racle才行，（注意，匹配的是mysqlracle或mysqoracle）。

（8）^和\A代表以什么开始。

（9）$和\Z代表以什么结束。

（10）\b，匹配边界，如this is island，要匹配中间的is，则要用'/\bis\b/'。

（11）\B相反。

（12）()的作用：1、改变优先级；2、将小原子变成大原子（如'/fas{2}/'出现2次只是对s有效，如果想对a也有效，需要加'/f(as){2}/'）；3、子组；4、反向引用。

（13）反向引用：如'/a(-|\_)b.{0,5}\1/'此处因为是单引号，反向引用可以只要1个\，如果是双引号，则要用2个（注意，反向引用，如果在模式中只能用\，在preg\_replace中替代表达式中则还可以有${1}等代替。

（14）如果想去掉()的3,4项作用，可以在括号里加(?:)，则只有12作用，不代表分组。

（15）惰性模式：匹配量词后加?，如+?/\*?/??/{n}?/{n,m}?/{n,}?/{,m}?。

分组内使用惰性模式，分组后加上其他限定，可以匹配很长的部分。



* 1. JSON对象

**1.语法规则**

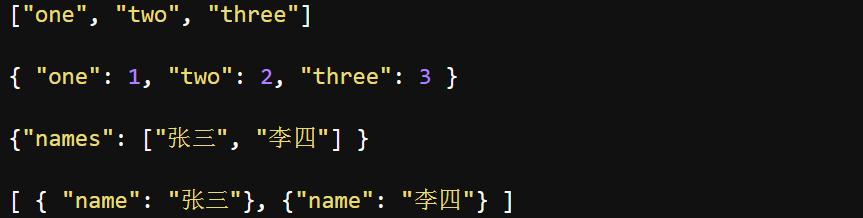
JSON的语法可以表示以下三种类型的值：

（1）简单值：可以在JSON中表示字符串、数值、布尔值和null，但不能使用undefined，且字符串必须使用双引号表示，数值必须以十进制表示，且不能使用NaN和Infinity。

（2）对象：对象的键名必须放在双引号里面。

（3）数组：数组或对象最后一个成员的后面，不能加逗号。

以下是合格的JSON值。



**2.方法**

（1）JSON.parse(text)。

（2）JSON.stringify(obj)。

* 1. Math对象

常用方法如floor/ceil/round/min/max/random等。其中，Math.random()返回0到1之间的一个伪随机数，可能等于0，但是一定小于1。

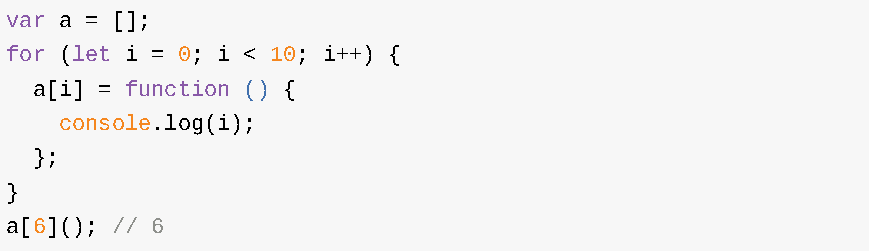
任意范围的随机数生成函数如下：



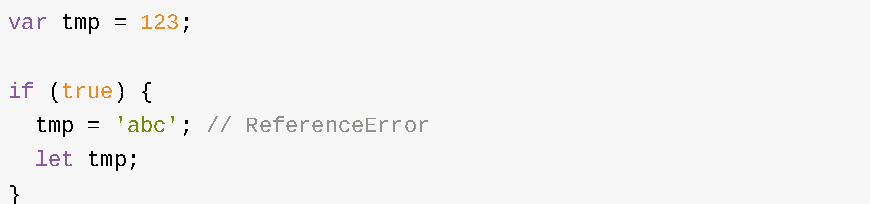
1. ES6
   1. let/const

**1.let**

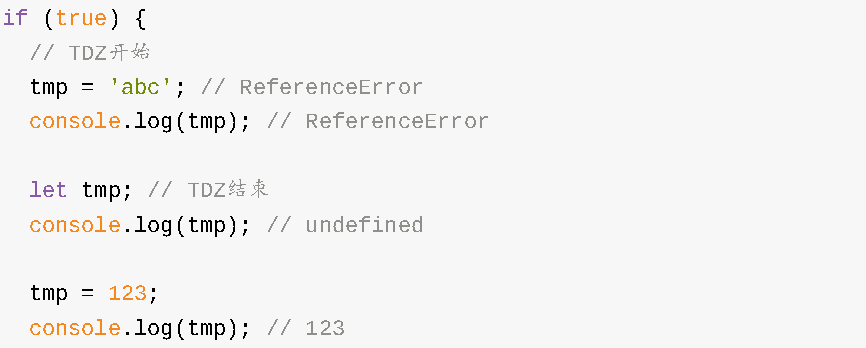
（1）let命令用来声明变量，用法类似于var。但是所声明的变量，只在所在的代码块内有效。



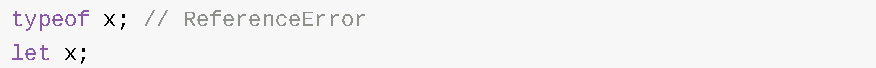
（2）不存在变量提升。



（3）只要代码块内使用了let，在let之前包括赋值在内的使用变量方式都会报错，即出现“暂时性死区”。



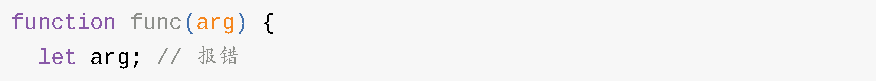
从而导致使用let后，typeof也不再是1个安全的操作。

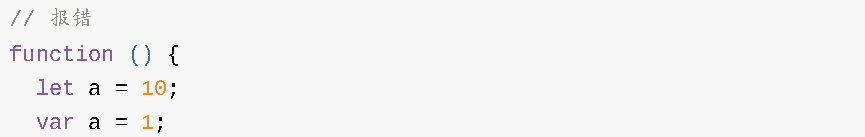


如果变量没有声明，反而不会报错。



（4）不允许在相同作用域内，重复声明。





**2.const**

（1）const声明一个只读的常量。一旦声明，常量的值就不能改变。

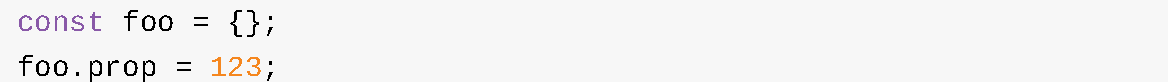


因此，const声明和初始化变量应一同完成，只声明不赋值，就会报错。



（2）同let一样，只在声明所在的块级作用域内有效；也不存在变量提升；同样存在暂时性死区，只能在声明的位置后面使用；不可重复声明。

（3）因复合类型变量名不指向数据，而是指向数据所在的地址。使用const命令只是保证变量名指向的地址不变，并不保证该地址的数据不变，尽量避用。

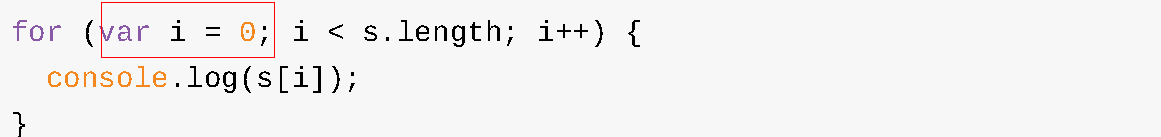


**3.块级作用域**

（1）let/const的出现为js增添了块级作用域，使得内层的变量不会覆盖外层的变量。



（2）也不会出现循环计数变量泄露为全局变量。



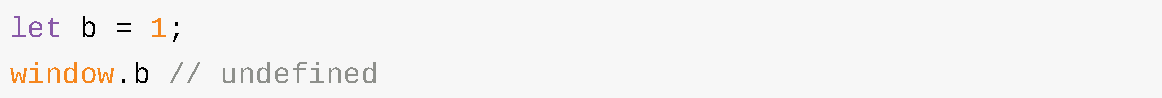
（3）ES5规定，函数只能在顶层作用域和函数作用域之中声明，不能在块级作用域声明。ES6引入了块级作用域，明确允许在块级作用域之中声明函数。块级作用域之中，函数声明语句的行为类似于let，在块级作用域之外不可引用。

**4.全局对象的属性**

（1）全局对象是最顶层的对象，在浏览器环境是window对象，在Node.js指的是global对象。ES5，全局对象的属性与全局变量是等价的。浏览器环境和nodejs的REPL环境下，直接声明全局变量即可成为global对象的属性，模块环境下，全局变量必须显式声明。

（2）为了保持兼容性，ES6规定，var命令和function命令声明的全局变量，依旧是全局对象的属性。let命令、const命令、class命令声明的全局变量，不属于全局对象的属性。





* 1. 变量的解构赋值

即，按照一定模式，从数组和对象中提取值，对变量进行赋值。

* + 1. 数组的解构赋值

**1.用法**

（1）基本用法，等同于声明了a/b/c三个变量。



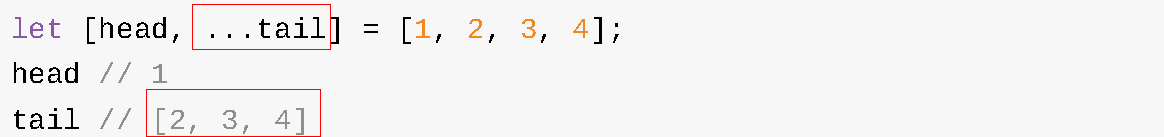
（2）嵌套解构。



（3）忽略部分变量。



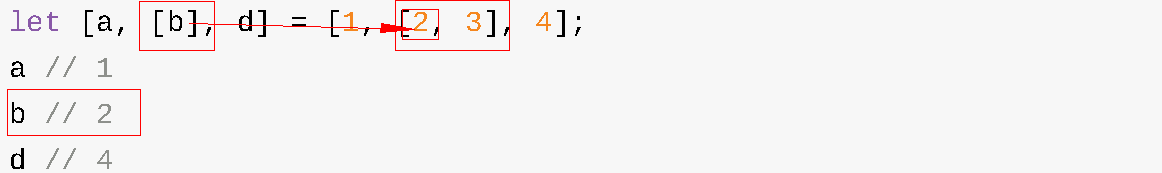
（4）解构到数组中。



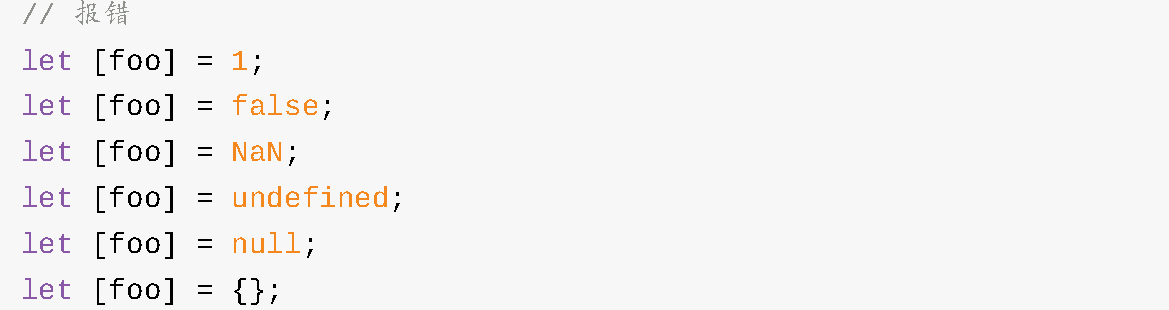
（5）解构不成功，即左边模式下的变量名多于右边数组值，为undefined。



（6）不完全解构，即左边模式下的变量名少于右边数组值，依然能够解构成功。

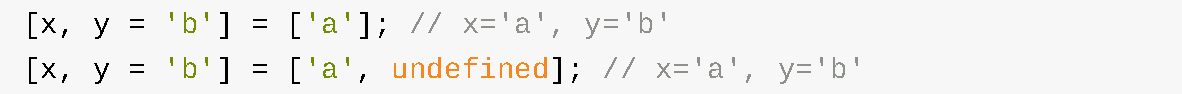


（7）当右边不是数组（可遍历结构）时，报错。

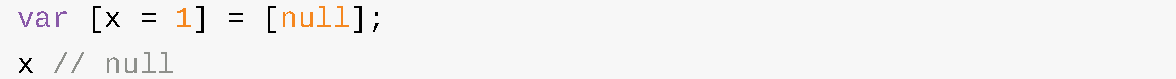


**2.默认值**

（1）解构赋值允许指定默认值。



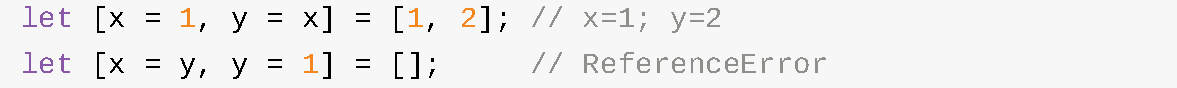
ES6内部使用===判断一个位置是否有值。如果不严格等于undefined，默认值就不会生效。



（2）如果默认值是1个表达式，会惰性求值，即在使用时才求值。



（3）默认值可以引用解构赋值模式中的其他变量，但该变量必须已经声明（即，只能引用前面的）。



* + 1. 对象的解构赋值

**1.对象解构赋值特点**

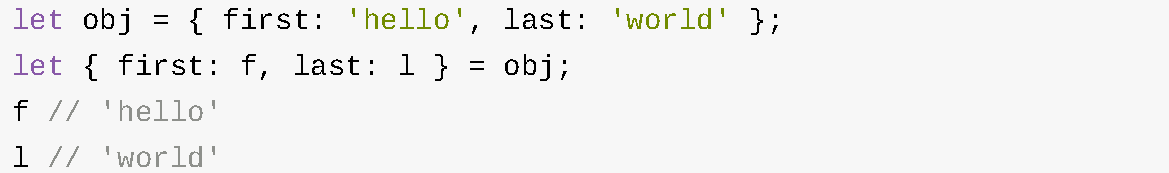
数组的元素是按次序排列的，解构赋值时变量取值由它的位置决定；而对象的属性没有次序，变量必须与属性同名，才能取到正确的值。如下面bar和foo的顺序不重要。



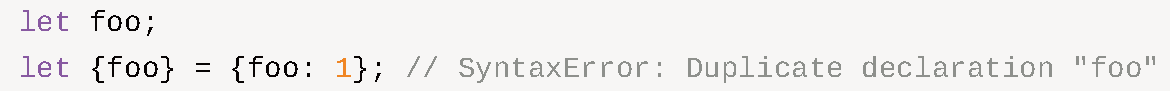
实际上，对象解构赋值是下面形式的简写。

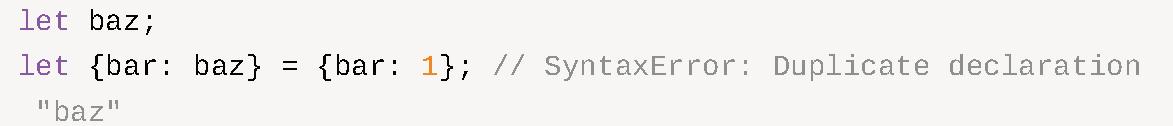


真正被赋值的是后者，而不是前者。对象解构赋值内部机制是先找到同名属性，然后再赋给对应的变量。因此，变量名可以和属性名不一致，但必须写成下面这样：



因为对象解构赋值变量声明和赋值一同进行，因此对于let和const来说，变量之前不能声明过。



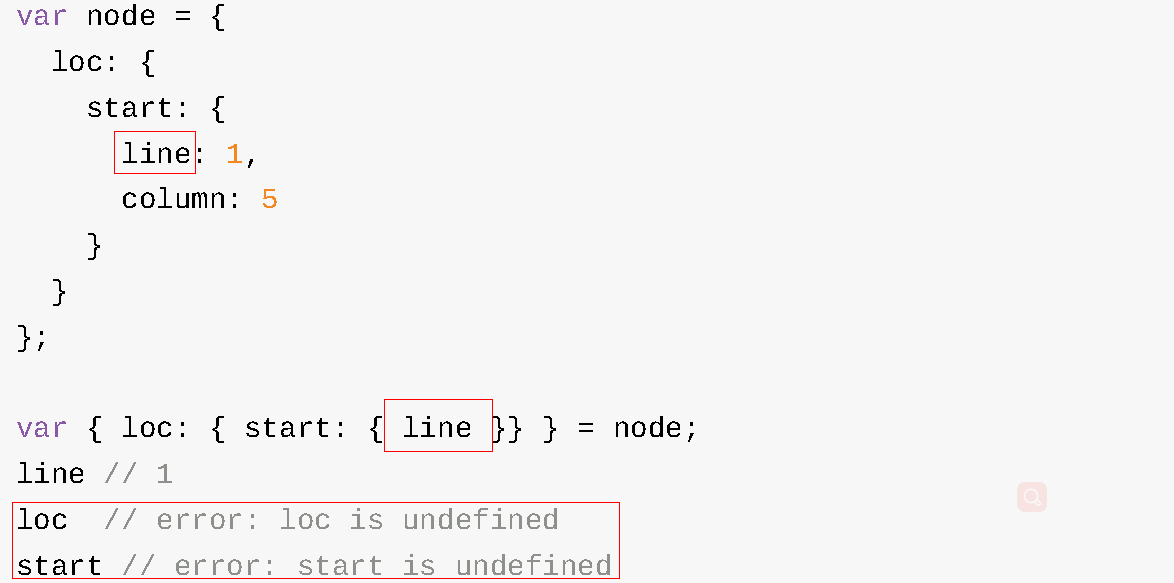


此时后面的解构赋值语句中，用()代替let，可以避免报错（原因是否是因为内部机制会判断，如果以前声明过就直接赋值？）。

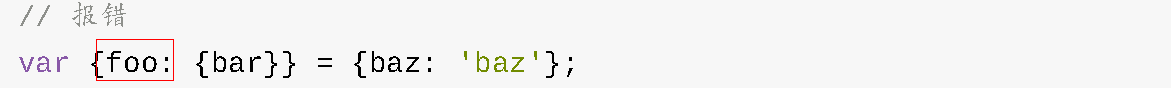


**2.对象的嵌套解构赋值**

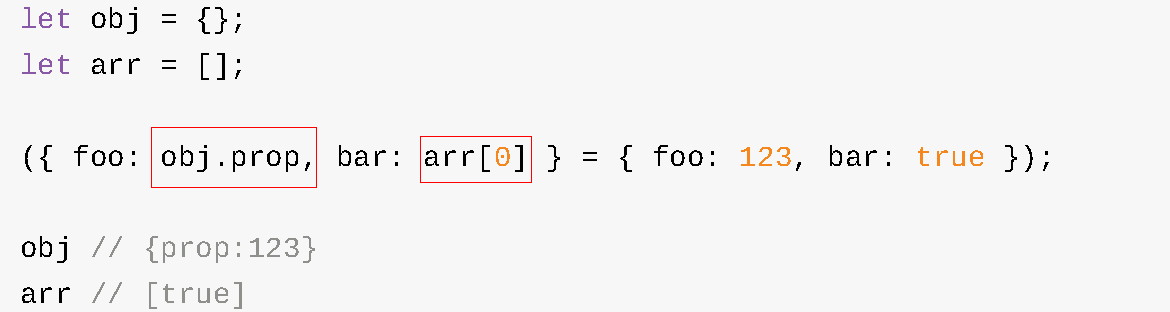
注意一点，解构赋值的目的是给简单变量赋值，而不是声明赋值1个对象。使用对象嵌套解构赋值时，要通过模式指定，最终只能给变量赋1个简单值。如下，只有line是变量，loc和start都是模式，不会被赋值。



如果子对象所在的父属性不存在，那么将会报错。

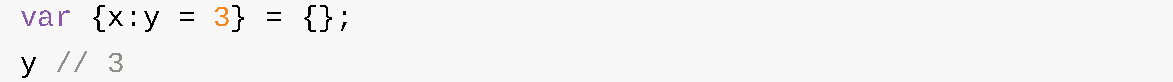


可以通过如下方式对复杂对象赋值。



**3.默认值**

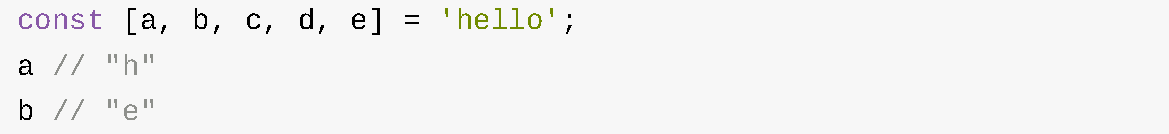
同数组类似，对象的属性值严格等于undefined。

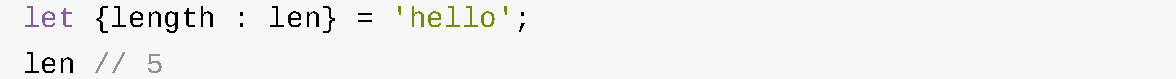


**4.其他**

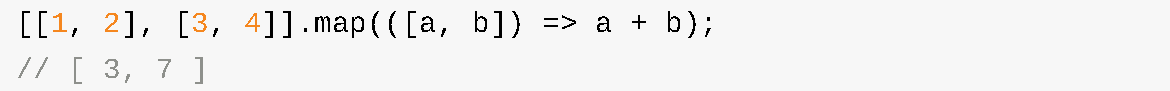
同数组，解构失败，变量的值等于undefined；也可以不完全解构；也不能重复声明，可以用()代替。

* + 1. 字符串的解构赋值





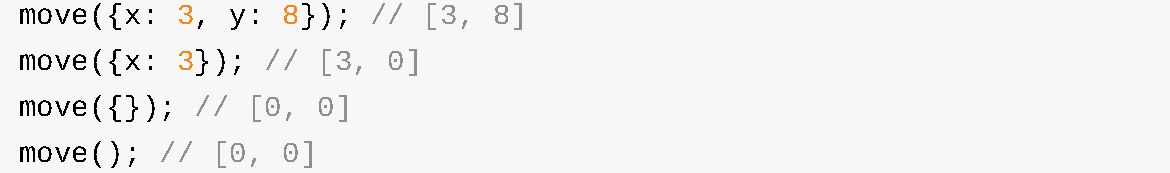
* + 1. 函数参数的解构赋值



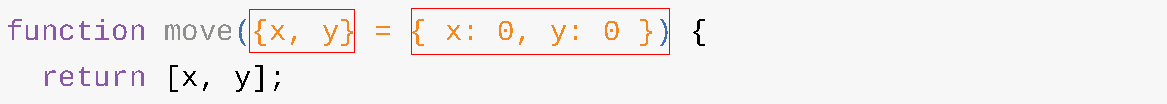
传入1个数组，但传入的那一刻被解构成2个变量。

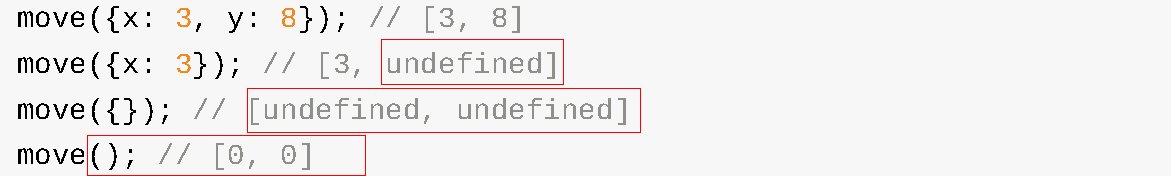
函数参数的解构也可以使用默认值（涉及函数的默认值）。





下面的写法会得到不一样的结果（并没有用到解构赋值，只用到函数默认值）。





* + 1. 圆括号问题

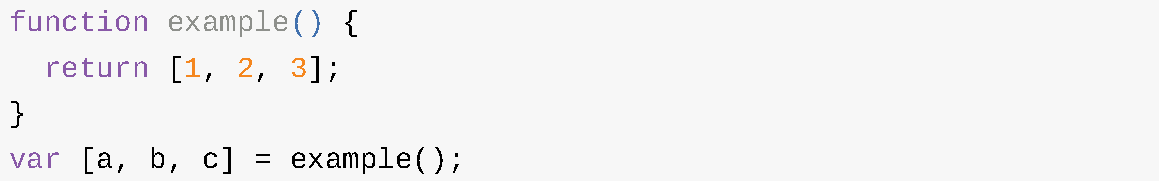
尽量不要在解构模式中使用()。

* + 1. 解构赋值用途

（1）交换变量值。



（2）从函数返回多个值。



（3）函数参数定义。



（4）提取JSON数据。

（5）输入模块的指定方法。



* 1. 数字扩展

（1）新增Number.isFinite()和Number.isNaN()方法，检查是否为Infinite和NaN。

（2）将全局方法parseInt()和parseFloat()，移植到Number对象上面，即Number.parseInt()和Number.parseFloat()。

（3）新增Number.isInteger()判断一个值是否为整数。注意，如同3和3.0在js中被视为同一个值。

（4）新增Number.isSafeInteger()判断一个整数是否落在这个范围之内。

* 1. 字符串扩展

**1.方法扩展**

新增match()、replace()、search()、split()、repeat()方法。

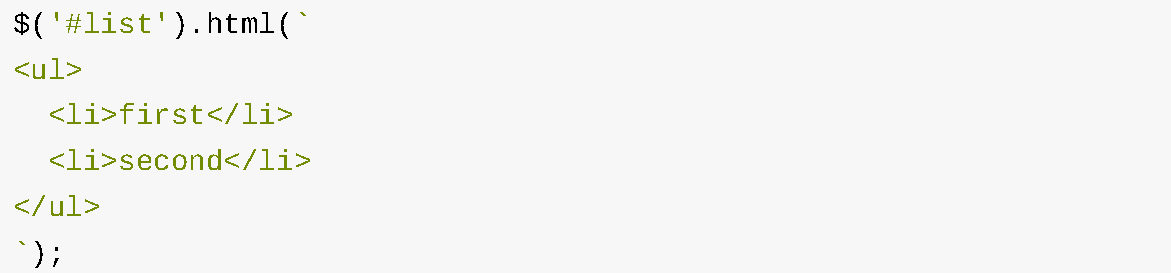
**2.模板字符串**

用反引号`标识模板，可以当作普通字符串使用，也可以用来定义多行字符串，或者在字符串中嵌入变量。

（1）如果在模板字符串中需要使用`，则前面要用反斜杠转义。

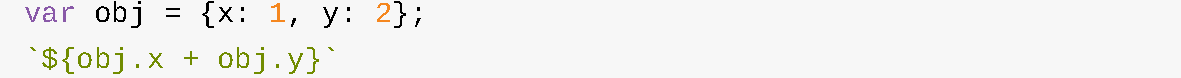


（2）如果表示多行字符串，所有的空格和缩进都会被保留在输出之中。



（3）嵌入变量使用${}。可以在其中进行运算，以及引用对象属性，调用函数。







如果${}中的值不是字符串，将按通常规则转为字符串。

* 1. 数组扩展

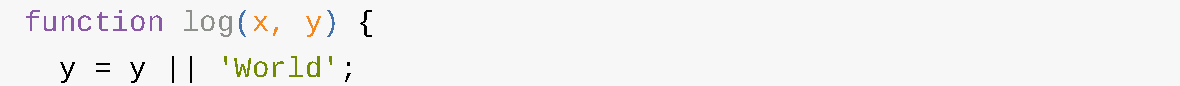
Array.of、Array.from、find、findIndex、fill及数组实例的entries、keys和values方法。

* 1. 函数扩展
     1. 参数扩展

**1.默认值参数**

（1）基本用法

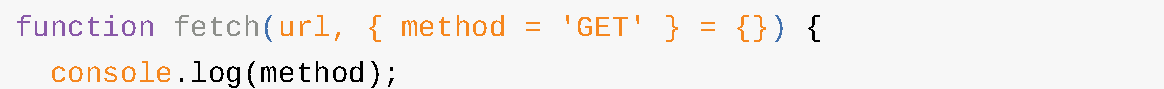
ES6之前，通常使用如下方法来设置默认值：

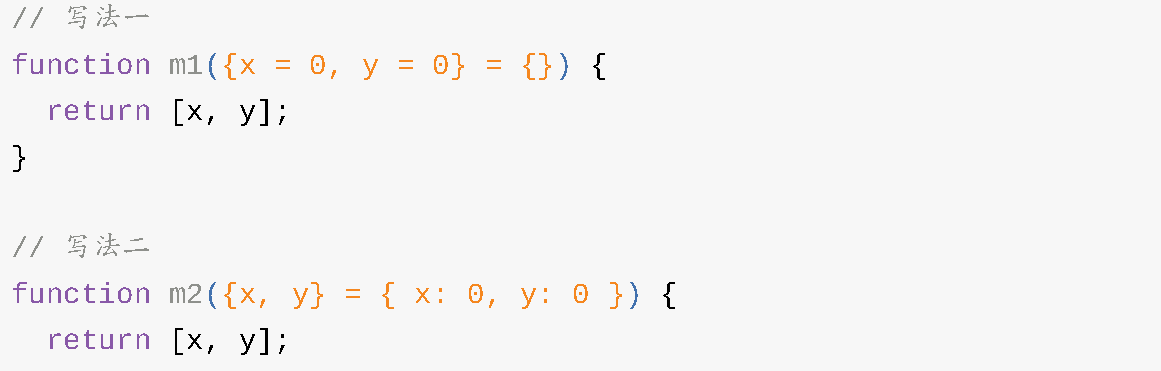


但如果y实参为false，则不起作用。ES6允许定义默认参数：



（2）与解构赋值结合使用





（3）参数默认值应该定义在参数末尾。

**2.rest参数**

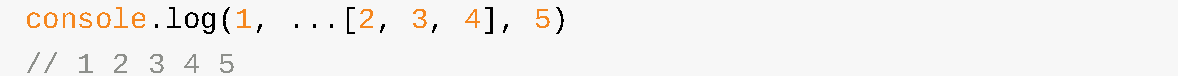
形式为“...变量名”，用于获取函数的多余参数，可以代替arguments对象。rest参数搭配的变量是一个数组，该变量将多余的参数放入数组中。注意，rest参数之后不能再有其他参数（即只能是最后一个参数），否则会报错。



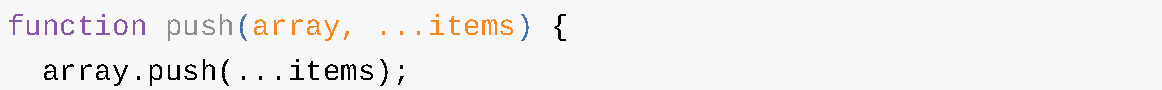
* + 1. 扩展运算符（使用上的）

**1.概念**

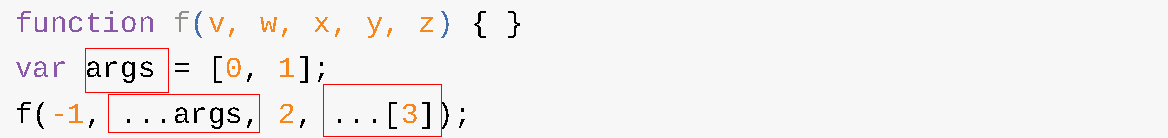
形式为三个点（...），为比rest参数的逆运算，可以将一个数组转为用逗号分隔的参数序列。



该运算符主要用于函数调用。

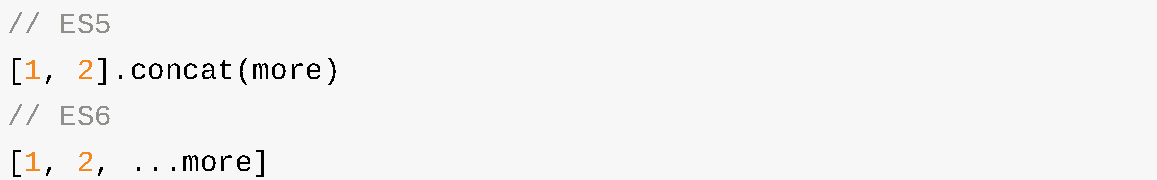


扩展运算符可以与正常的函数参数结合使用（注意rest参数是函数定义时，这里是调用函数时）。

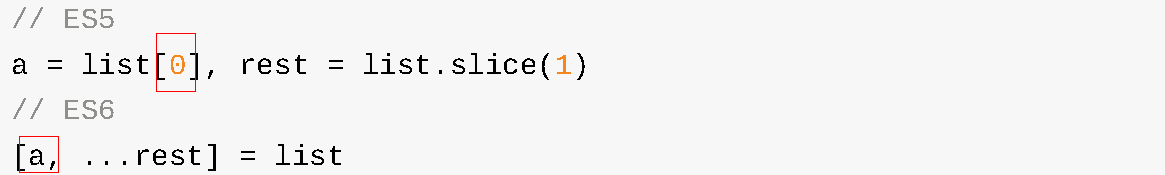


**2.应用**

（1）合并数组。



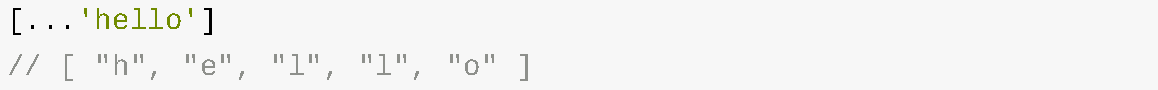
（2）与解构赋值结合，用于生成新数组和变量赋值。



但扩展运算符必须放在最后，否则报错。



（3）将字符串转为数组。



* + 1. 属性扩展

**1.length属性**

返回没有指定默认值的参数个数，rest参数也不会计入length属性。

**2.name属性**

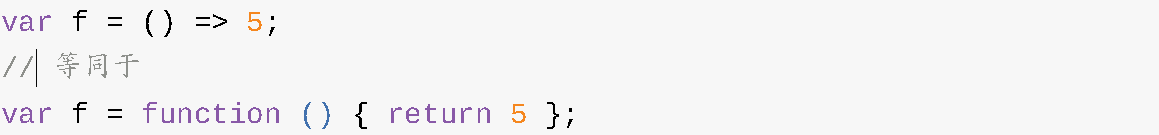
返回该函数的函数名。

* + 1. 箭头函数

**1.基本用法**



（1）如果不需要参数或需要多个参数，需要使用一个圆括号代表参数部分。



（2）如果箭头函数的代码块部分多于一条语句，就要使用大括号括起来。



（3）箭头函数也可以与变量解构结合使用。



**2.箭头函数使用注意事项**

（1）箭头函数内的this对象，就是定义时所在的对象，而不是使用时所在的对象（箭头函数不能使用bind/call/apply）。

（2）箭头函数不可以当作构造函数，也就是说，不可以使用new命令，否则会抛出一个错误。

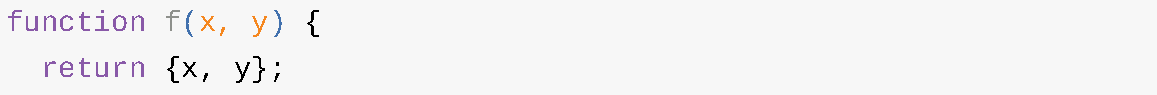
（3）箭头函数的函数体内不存在arguments对象。

（4）不可以使用yield命令，因此箭头函数不能用作Generator函数。

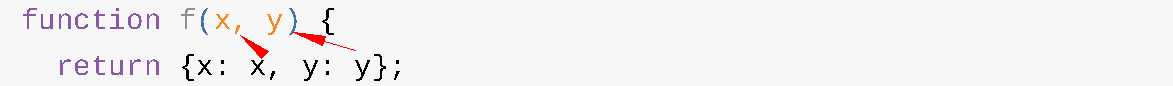
* 1. 对象扩展

**1.属性的简洁表示法**

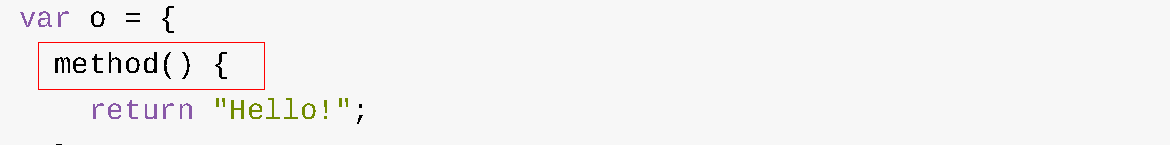
只写属性名，不写属性值。这时，属性值等于属性名所代表的变量。



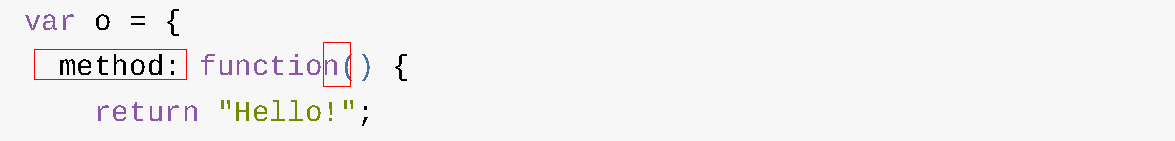
等价于：



方法也可以简写，函数名就方法名。



等价于：

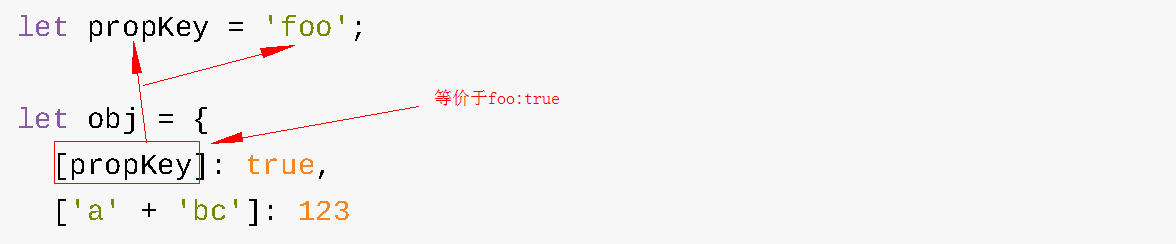


CommonJS模块输出变量，就非常合适使用简洁写法。



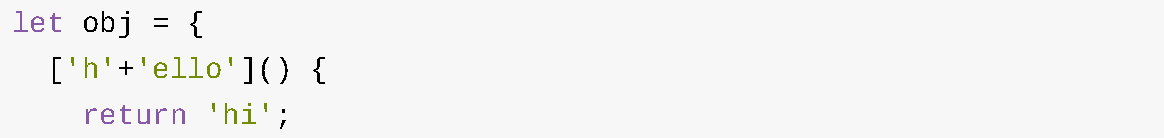
**2.属性名表达式**

ES6允许字面量定义对象时，可以将表达式放在方括号内作为属性名。



上面访问的时候，即可以用obj[propkey]也可以用obj['foo']访问到同一个属性。

表达式还可以用于定义方法名。



属性名表达式与简洁表示法，不能同时使用，否则报错（方法貌似可以）。

**3.方法扩展**

Object.setPrototypeOf/Object.getPrototypeOf、Object.is、Object.assign。

* 1. Symbol

**1.使用场景**

当使用了一个第3方提供的对象，但又想为这个对象添加新的方法（mixin模式），新方法的名字就有可能与现有方法产生冲突。引入Symbol的原因就是为了保证每个属性的名字都是独一无二。对于一个对象由多个模块构成的情况非常有用，能防止某一个键被不小心改写或覆盖。

**2.创建**

（1）使用Symbol函数生成。

（2）Symbol函数前不能使用new命令，否则会报错，因为Symbol值不是对象。

（3）Symbol函数可以接受一个字符串作为参数，表示对Symbol实例的描述（只是描述），主要是用于在控制台显示，或者转为字符串时，便于区分。



**3.类型**

（1）由此，对象的属性名可以有两种类型，一种是原来就有的字符串，另一种就是新增的Symbol类型。凡是属性名属于Symbol类型，就都是独一无二的，可以保证不会与其他属性名产生冲突。



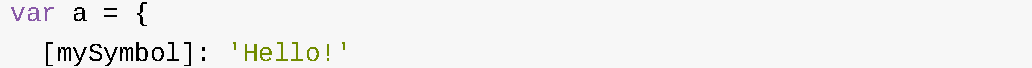
（2）Symbol值不能与其他类型的值进行运算（不能隐式转换，可以显示转换），会报错。

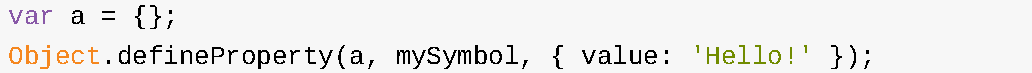


**4.基本使用**

和字符串属性名类型，但只能使用[]读取。









**5.遍历**

（1）Symbol属性名不会出现在for...in、for...of及Object.keys()、Object.getOwnPropertyNames()中。可以通过Object.getOwnPropertySymbols()方法获取，返回一个数组，为对象所有Symbol属性名。

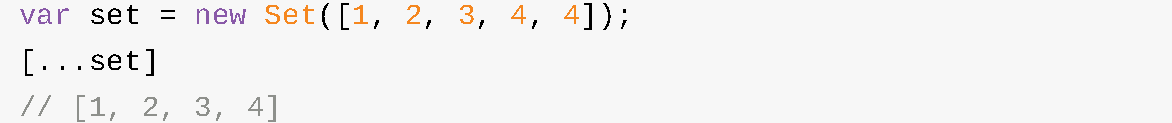
* 1. Set和Map数据结构
     1. Set

**1.基本用法**

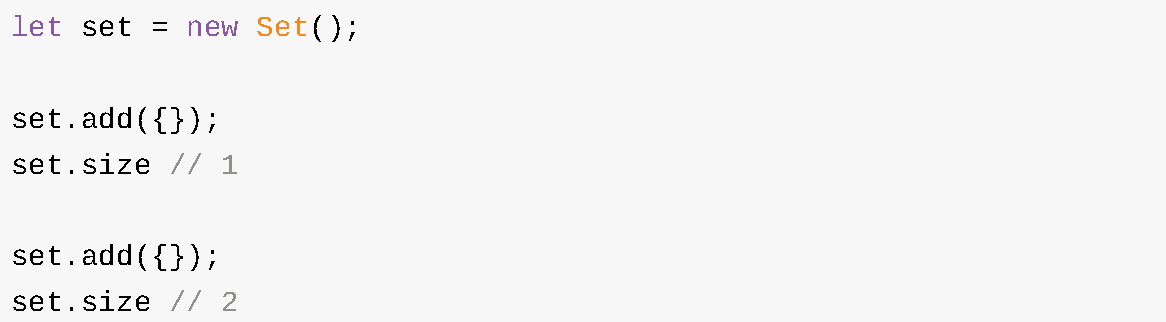
（1）Set集合，类似于数组，但成员的值都是唯一的。仍然是对象。

（2）Set本身是一个构造函数，用来生成Set数据结构。

（3）Set函数可以接受一个数组（或类数组对象）作为参数初始化。



（4）向Set加入值的时候，不会发生类型转换，所以5和"5"是两个不同的值。在判断NaN时，NaN等于自身。另外，两个对象总是不相等的。



**2.实例属性**

（1）Set.prototype.constructor：构造函数，默认就是Set函数。

（2）Set.prototype.size：返回Set实例的成员总数。

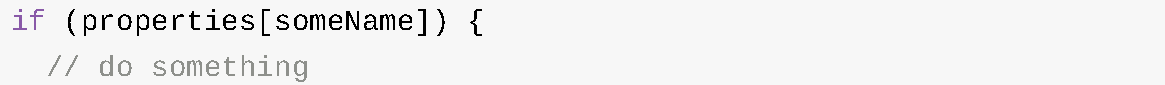
**3.实例方法**

（1）add(value)：添加某个值，返回Set结构本身。

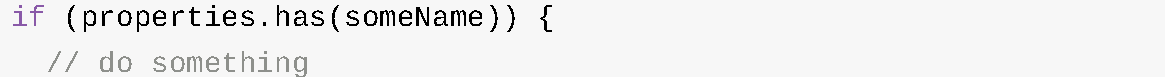
（2）delete(value)：删除某个值，返回一个布尔值，表示删除是否成功。

（3）has(value)：返回一个布尔值，表示该值是否为Set的成员。

对象判断是否拥有某个属性，用如下方法：



set实例用如下方法：



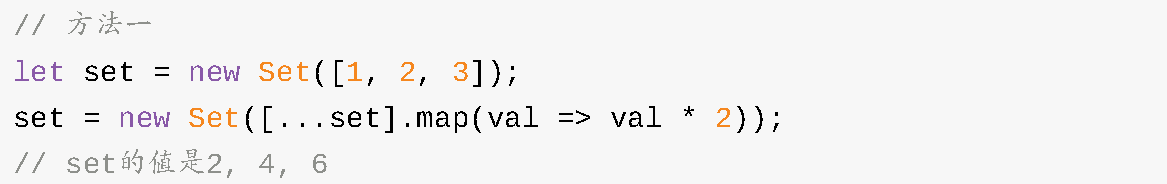
（4）clear()：清除所有成员，没有返回值。

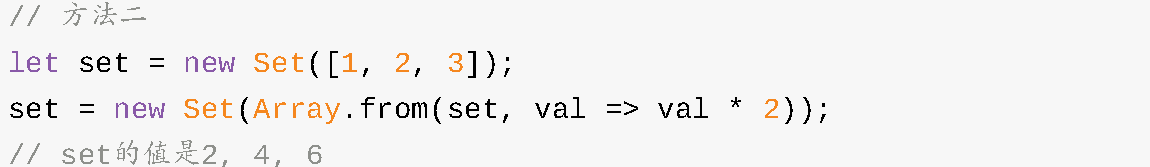
（5）keys()/values()/entries()：返回对应的迭代器。

（6）forEach()：使用回调函数遍历每个成员，直接对每个成员执行某种操作，没有返回值。



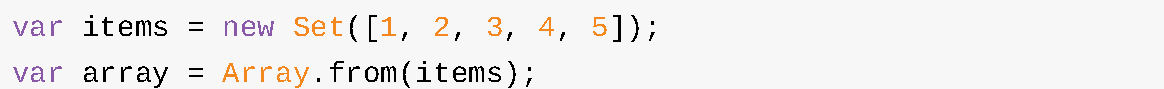
用forEach无法直接改变原set结构，可以使用如下方法（转为数组）实现：



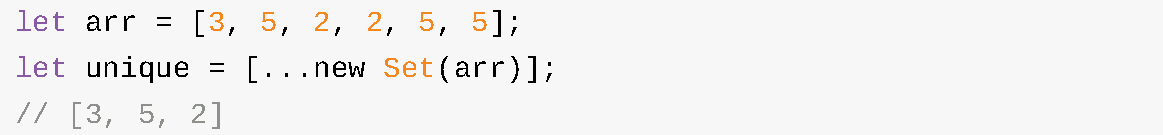


**4.用处**

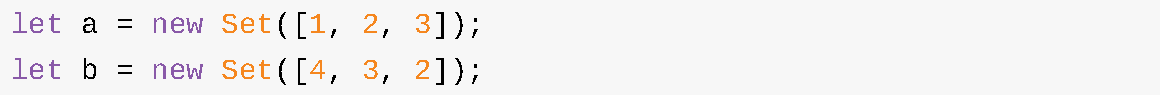
（1）Array.from方法可以将Set结构转为数组。从而可以对数组去重。

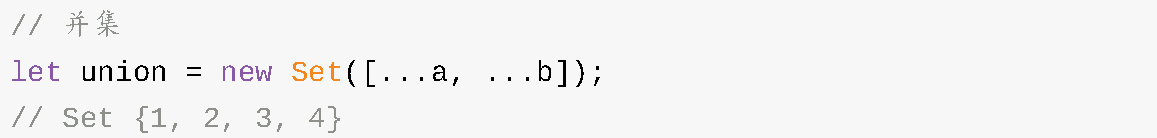


更方便的，使用扩展运算符和Set结构相结合，就可以去除数组的重复成员。



（2）使用Set可以很容易地实现并集（Union）、交集（Intersect）和差集（Difference）。









* + 1. Map

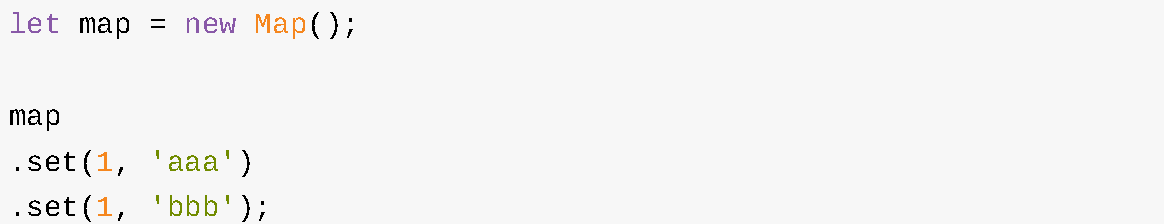
**1.概要**

（1）Map数据结构类似于对象，也是键值对的集合，但是键的范围不限于字符串，各种类型的值（包括对象）都可以当作键。

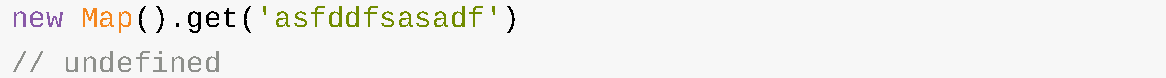
（2）Map也可以接受一个数组作为参数。该数组的成员是一个个表示键值对的数组。



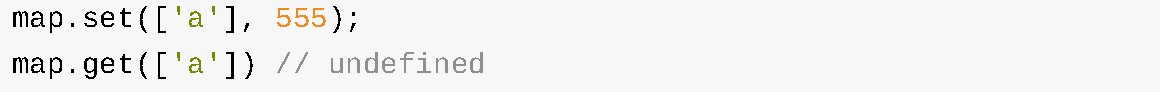
（3）如果对同一个键多次赋值，后面的值将覆盖前面的值。



（4）如果读取一个未知的键，则返回undefined。



（5）只有对同一个对象的引用，Map结构才将其视为同一个键。



上面的set和get方法，表面是针对同一个键，但实际上这是两个值，内存地址是不一样的，因此get方法无法读取该键，返回undefined。

**1.实例属性**

size：属性返回Map结构的成员总数。

**2.实例方法**

（1）set：设置对应的键值。

（2）get：获取键对应的值。

（3）has：检测是否存在。

（4）delete：删除。

（5）clear：清除所有。

（6）keys/values/entries：返回对应的迭代器。

（7）forEach：遍历Map的所有成员，遍历顺序就是插入顺序。

可以使用扩展运算符将map转为数组。



* 1. Iterator和Generator
     1. Iterator

**1.概要**

（1）迭代器是一种接口，用程序化的方式返回被操作对象的每一个元素，并为各种不同的数据结构提供统一的访问机制。新的数组方法和集合类型都依赖迭代器的实现，for-of循环、展开运算符(...)，异步编程也使用迭代器。

（2）所有的迭代器对象都有一个next()方法，每次调用都返回一个结果对象。结果对象有两个属性：一个是value，表示下一个将要返回的值；另一个是done，它是一个布尔类型的值，当没有更多可返回数据时返回true。迭代器还会保存一个内部指针，用来指向当前集合中值的位置，每调用一次next()方法，都会返回下一个可用的值。

（3）如果在最后一个值返回后再调用next()方法，那么返回的对象中属性done的值为true，属性value则包含迭代器最终返回的值，这个返回值不是数据集的一部分，它与函数的返回值类似，是函数调用过程中最后一次给调用者传递信息的方法，如果没有相关数据则返回undefined。

（4）迭代器的遍历过程为：

1）创建一个指针对象，指向当前数据结构的起始位置。即，迭代器对象本质上就是一个指针对象。

2）第1次调用指针对象的next方法，可以将指针指向数据结构的第1个成员（{value:}）。

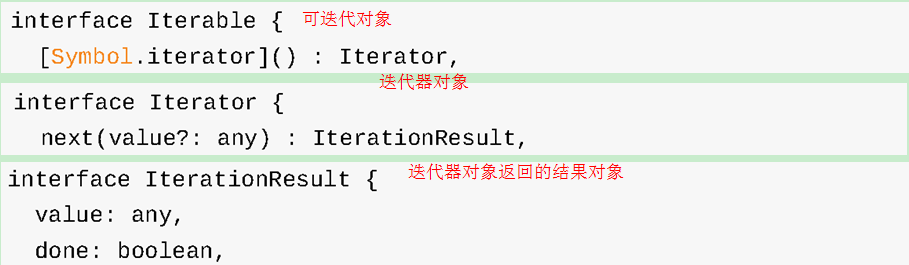
3）第2次调用指针对象的next方法，指针就指向数据结构的第2个成员。

4）不断调用指针对象的next方法，直到它指向数据结构的结束位置{done:true}。

**2.默认****Iterator接口**

（1）Iterator接口的目的，就是为所有数据结构，提供了一种统一的访问机制，即for...of循环。当使用for...of循环遍历某种数据结构时，该循环会自动去寻找Iterator接口。

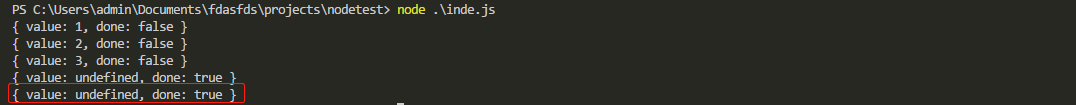
（2）ES6规定，默认的Iterator接口部署在Symbol.iterator属性上，即，当1个数据结构具有Symbol.iterator属性，就可以认为是可遍历的。调用Symbol.iterator方法，就会得到当前数据结构默认的迭代器生成函数。



（3）在ES6中，有3类数据结构原生具备Iterator接口：数组、某些类似数组的对象（arguments对象、DOM NodeList对象可以，自己写的类数组对象没有部署）、Set和Map结构。

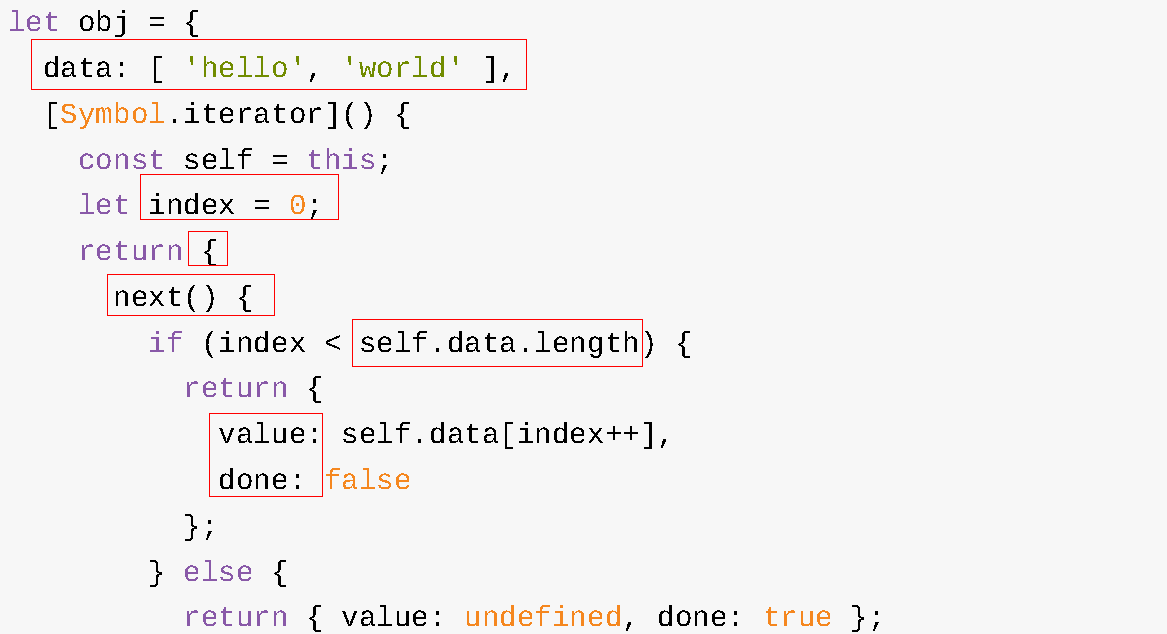
即，不用任何处理，就可以被for...of循环遍历。对象之所以没有默认部署Iterator接口，是因为对象的哪个属性先遍历，哪个属性后遍历是不确定的，需要开发者手动指定。





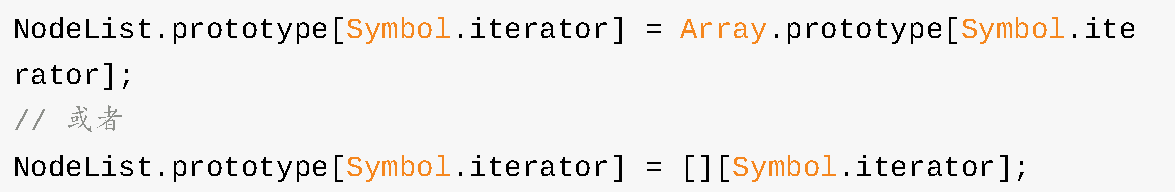
**3.部署Iterator接口**

除3类原生数据结构外，都需要自己在Symbol.iterator属性上面部署Iterator接口。



即，关键是（1）有Symbol.iterator属性，（2）属性为1个函数，函数返回1个对象，对象有next方法，（3）next方法返回1个对象，有value和done属性。

对于类数组对象（普通对象不行），简便方法就是Symbol.iterator方法直接引用数组的Iterator接口。或者，使用Array.from方法将其转为数组。



**4.调用Iterator接口的场合**

（1）解构赋值、扩展运算符、任何接受数组作为参数的函数时，会默认调用Symbol.iterator方法。

（2）yield\*后面跟的是一个可遍历的结构，也会调用该结构的迭代器接口。

（3）字符串是一个类似数组的对象，也原生具有Iterator接口。

**5.迭代器对象的return和throw方法**

（1）对于自己部署的迭代器，next方法必须部署，return方法和throw方法是否部署是可选的。

（2）return方法作用是，如果for...of循环提前退出（出错，或者有break语句或continue语句），就会调用return方法。

（3）throw方法主要是配合Generator函数使用，通常用不到。

* + 1. Generator

**1.概念**

（1）生成器是一种返回迭代器的函数。可以理解为1个状态机，返回1个迭代器对象（自动会将yield的值帮着包装）。可通过该迭代器对象依次遍历Generator函数内部的每一个状态。





（2）形式上，与普通函数相比有2个不同特征：function关键字与函数名之间有一个星号，函数体内部使用yield语句，定义不同的内部状态。

因没有严格规定\*和函数名、function关键字位置，所以下列4种形式写法都是正确的：









（3）Generator函数不能使用new实例化，且总是返回一个迭代器。内部给this添加属性，实例无法获取。所以，不要在其中使用this。

（4）执行Generator返回的迭代器本身又有1个Symbol.iterator属性，执行后返回自身。



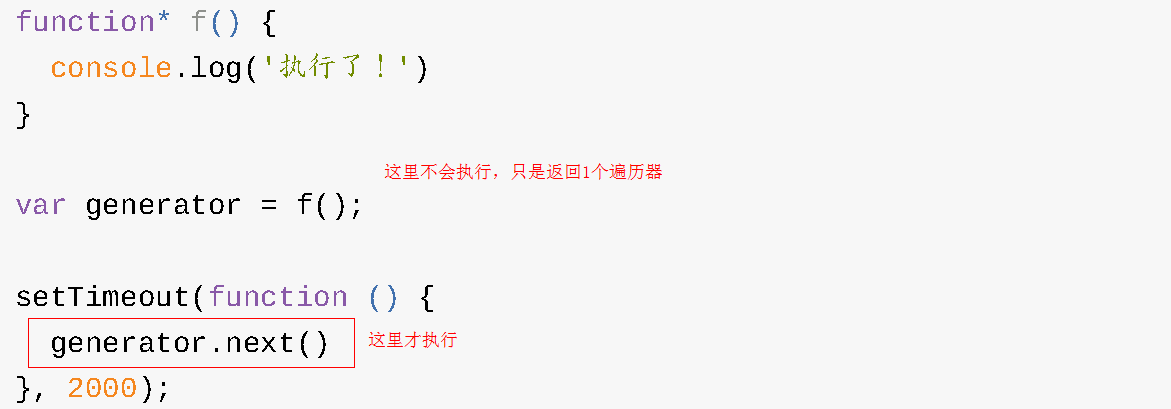
**2.yield语句**

（1）迭代器对象的next方法的运行逻辑为：遇到yield语句就暂停执行后面的语句。再次调用next方法从上次暂停的位置继续执行，直到下一个yield语句。如果没有遇到yield语句就一直执行到return为止。

（2）yield表达式后面的表达式，只有当调用next方法、内部指针指向该语句时才会执行，即相当于提供了惰性求值。

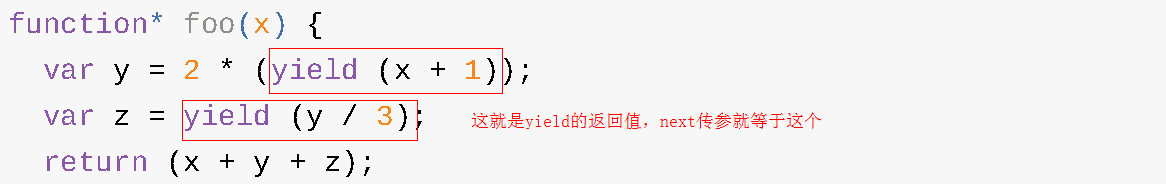
（3）yield语句不能用在普通函数中，否则报错。

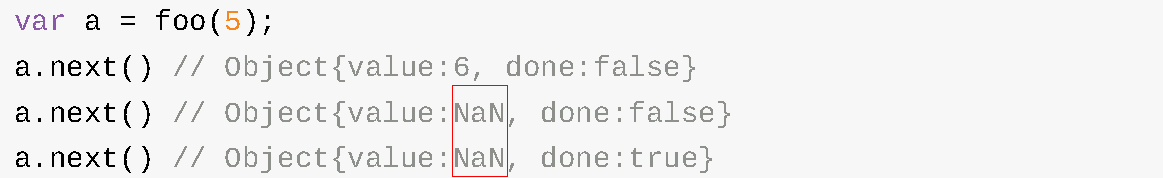
（4）Generator函数可以不用yield语句，此时就变成了一个单纯的暂缓执行函数。



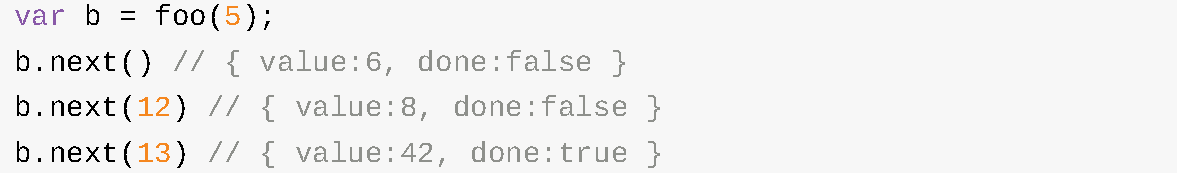
**3.next方法的参数**

上1个yield不会返回给下个yield所需的值，但可以通过传给next方法1个参数，作为这个返回值。





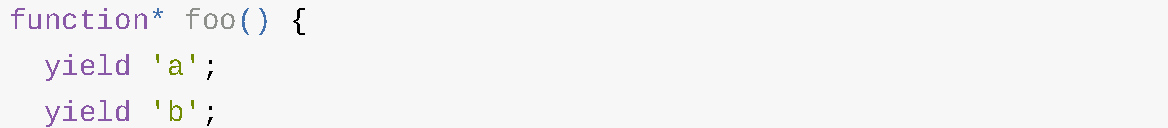
之所以后两次next都返回NaN，因为没有没法获取上次yield的返回值。

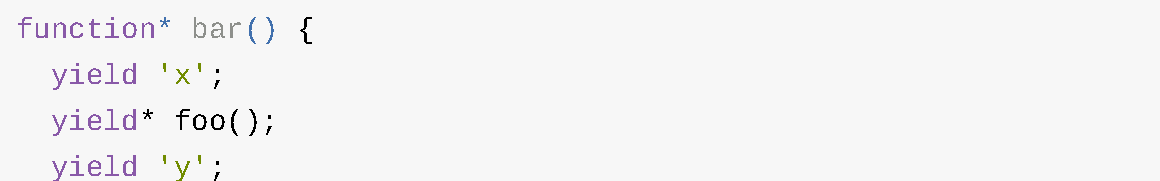


第2次，y=12×2（第2次传进去12赋值给了yield，就是x+1那个整体，然后再乘以2，所以y=24），第3次，z=13，即整个yield语句就是next的返回值（会保持住），因此，第3次x=5，y=24，z=13，返回42。

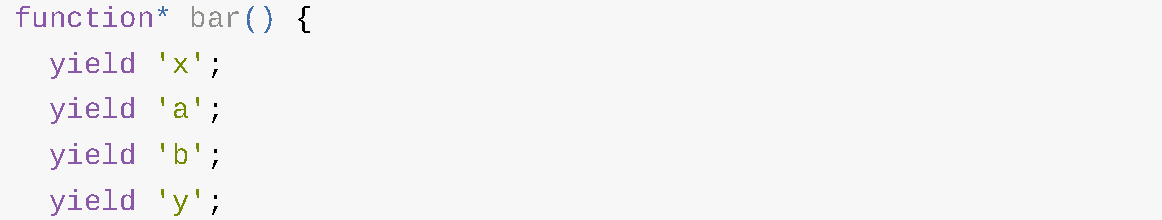
**4.yield\*语句**

（1）如果在Generater函数内部，调用另一个Generator函数，等同于合并被调用的迭代器。





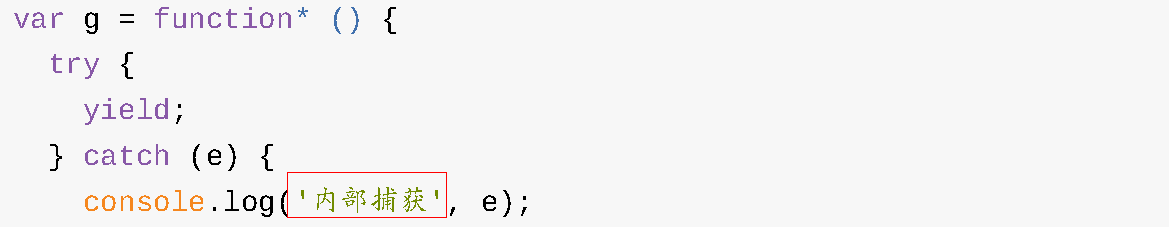
等同于：

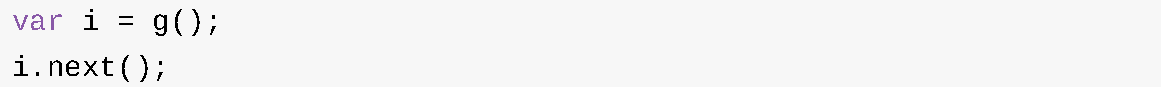


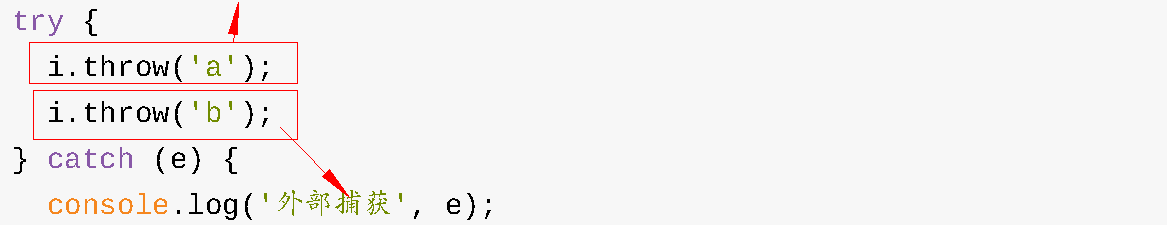
（2）如果yield后面没有\*，此时的next将返回foo这个函数执行返回的迭代器（不再是合并内部的状态）。

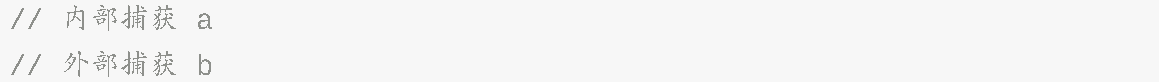
**5.Generator.prototype.throw**

（1）Generator函数返回的迭代器对象，都有一个throw方法，可以在函数体外抛出错误，然后在Generator函数体内捕获。









如上，迭代器对象i连续抛出2个错误，第1个错误被Generator函数体内的catch语句捕获。i第2次抛出错误，由于Generator函数内部的catch语句已经执行过了，不会再捕捉到这个错误了，所以这个错误就被抛出了Generator函数体，被函数体外的catch语句捕获。

（2）throw方法最好抛出Error对象的实例。



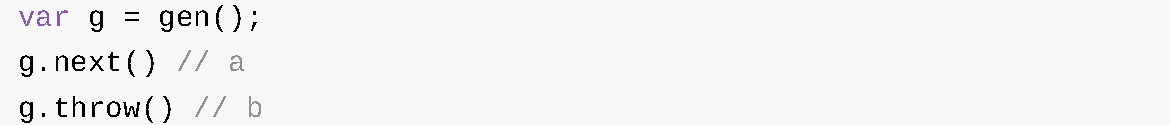
（3）不要混淆迭代器对象的throw方法和全局的throw命令。后者只能被函数体外的catch语句捕获。

（4）如果Generator函数内部没有部署try...catch代码块，那么throw方法抛出的错误，将被外部try...catch代码块捕获。

（5）如果Generator函数内部和外部都没有部署try...catch代码块，那么程序将报错，直接中断执行。

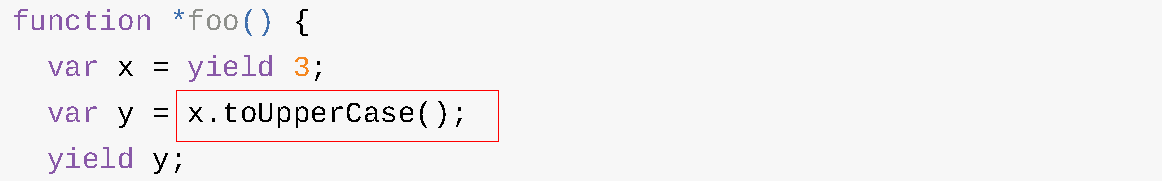
（6）throw方法被捕获以后，会附带执行下一条yield语句。

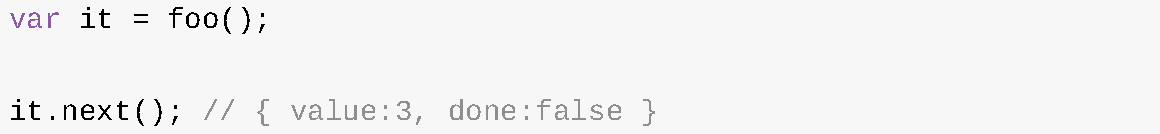


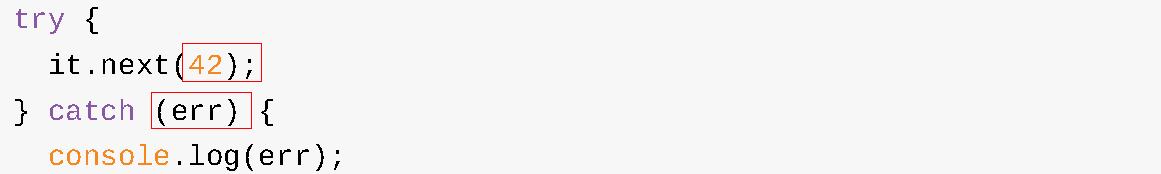


g.throw方法被捕获以后，自动执行了一次next方法，所以会打印b。

（7）Generator函数体内抛出的错误，也可以被函数体外的catch捕获。





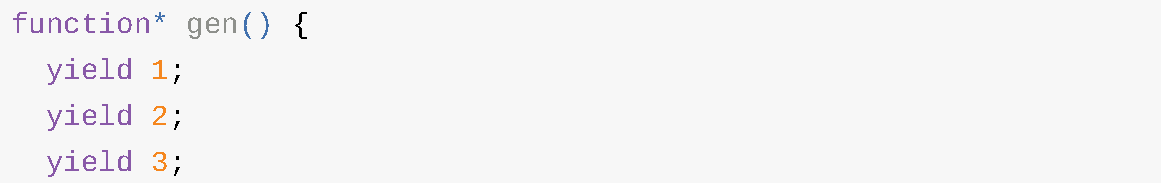


第2次next传入42，使用toUpperCase方法会抛出TypeError错误，被函数体外的catch捕获。

（8）如果Generator执行过程中抛出错误，且没有被内部捕获，就不会再执行下去了。此后如果再调用next方法，JavaScript引擎认为这个Generator已经运行结束了。

**6.Generator.prototype.return**

返回给定的值，并且终结遍历Generator函数。



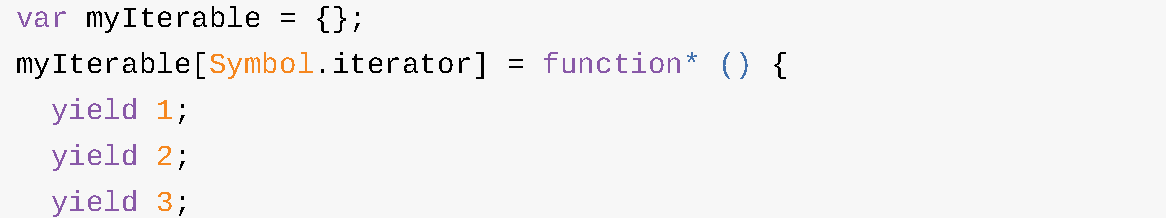




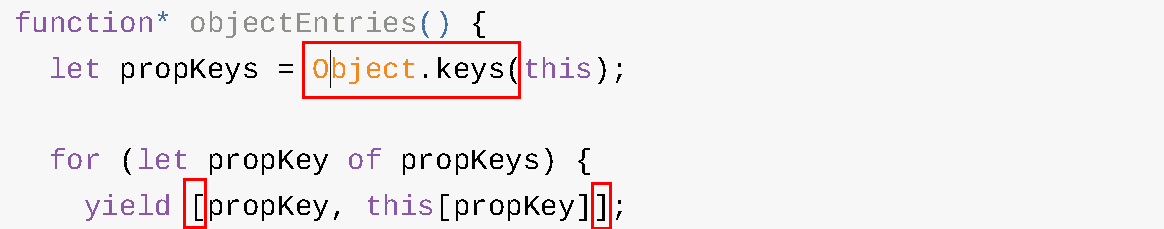
迭代器对象g调用return方法后，返回值的value属性就是return方法的参数foo。Generator函数的遍历就终止了。如果return方法调用时，不提供参数，则返回值的value属性为undefined。

* + 1. 使用Generator给对象部署迭代器

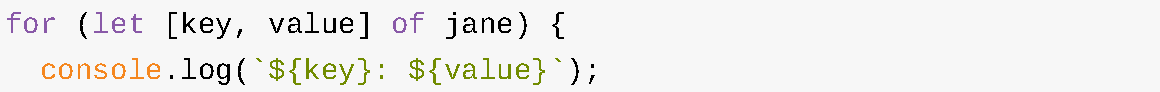
（1）因Generator就是迭代器函数（返回迭代器），可以把Generator赋值给对象的Symbol.iterator属性，从而使得该对象具有Iterator接口。



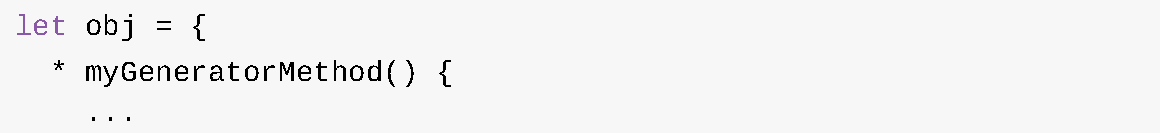
（2）原生对象不具备Iterator接口，通过Generator函数为它加上迭代器接口。



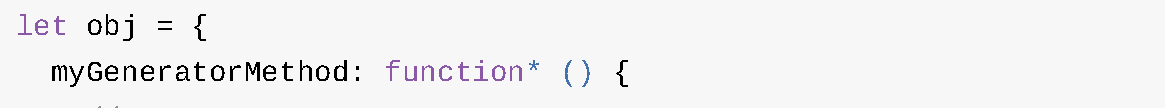




（3）如果一个对象的属性是Generator函数，可以简写成下面的形式。



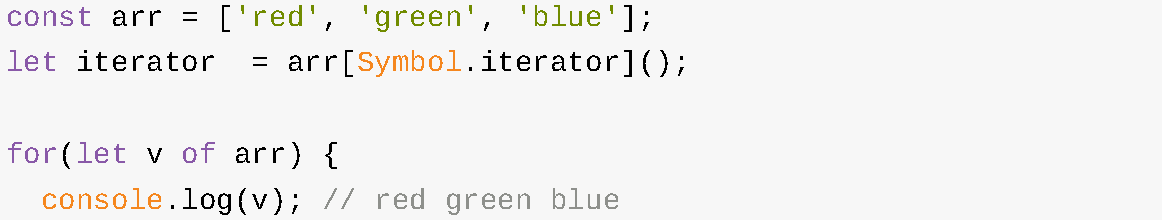
等价于：



* + 1. for...of循环

只要一个数据结构部署了Symbol.iterator属性，就被视为具有iterator接口，就可以用for...of循环遍历它的成员。

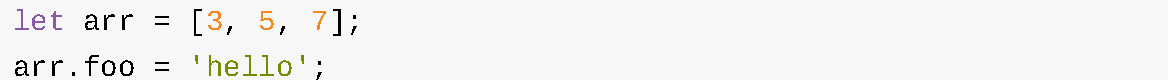
**1.数组**





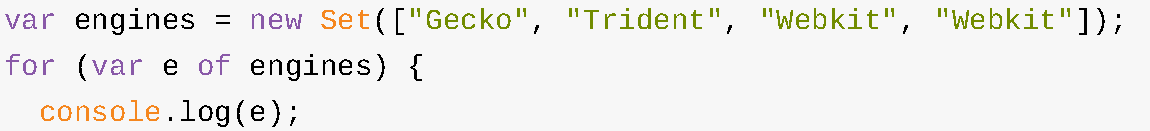


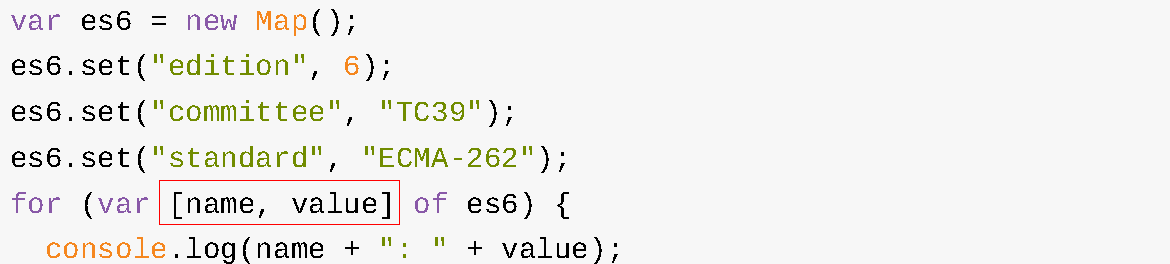
与for-in循环区别：for-in循环只能获得对象的键名，不能直接获取键值。for-of循环可以获得键值，但不能获得键名。另外，for-of只返回具有数字索引的属性，for-in可以返回任何索引。





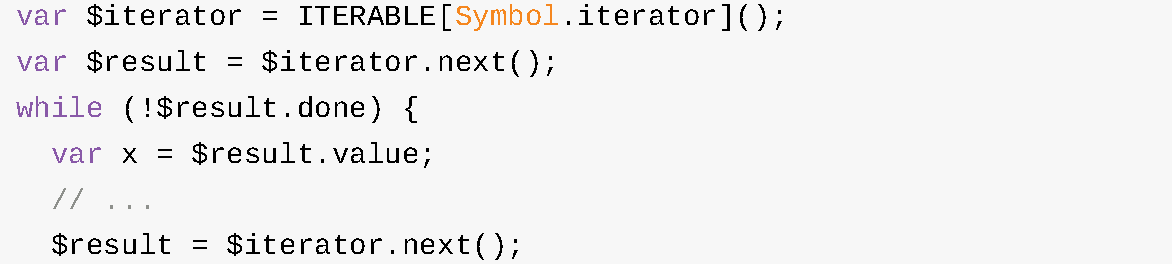
**2.Set和Map结构**





首先，遍历的顺序是按照各个成员被添加进数据结构的顺序。其次，Set结构遍历时，返回的是一个值，而Map结构遍历时，返回的是一个数组，该数组的两个成员分别为当前Map成员的键名和键值。

* ES6的数组、Set、Map都部署了keys/values/entries，调用后都返回迭代器对象。
* 对象不能直接使用for-of，但可以使用for-in来获取属性名。
* 除用for...of循环遍历，也可以使用while循环遍历。

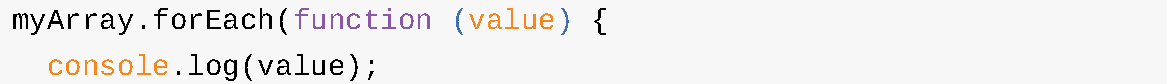


**3.与其他遍历语法的比较**

（1）for循环



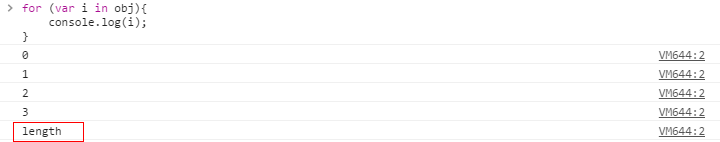
（2）forEach



问题在于，无法中途跳出forEach循环，即使break命令或return命令都不能奏效。

（3）for-in

缺点有：1）数组的键名是数字，但是for-in循环是以字符串作为键名，如“0”、“1”、“2”等。2）for-in循环不仅遍历数字键名，还会遍历手动添加的其他键，甚至包括原型链上的键。3）某些情况下，for-in循环会以任意顺序遍历键名。



（4）for-of

优点是简洁，可以与break、continue和return配合使用，并为不同数据结构提供了1个统一操作接口。

* 1. 异步编程
     1. Promise对象

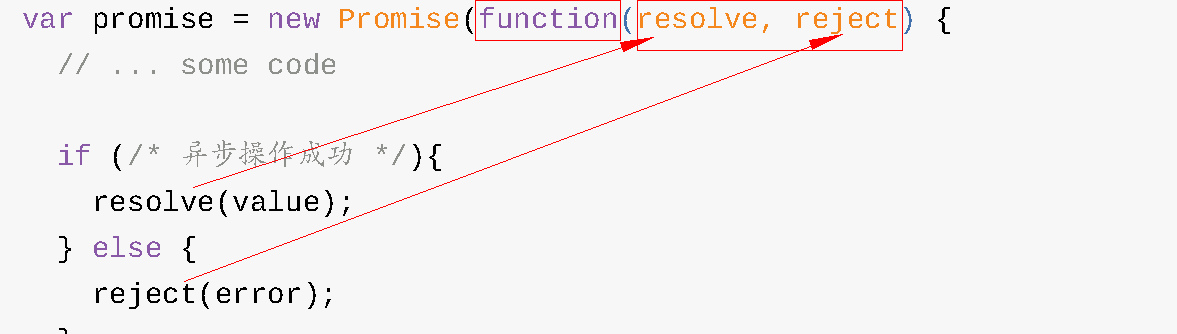
**1.概述**

（1）Promise是异步编程的一种解决方案。简单而言，Promise就是一个容器，保存着某个未来才会结束的事件的结果。

（2）Promise特点有：1）对象的状态不受外界影响。对象状态一共有3种：Pending（进行中）、Resolved（已完成，又称Fulfilled）和Rejected（已失败）。只有异步操作的结果，可以决定当前是哪一种状态，任何其他操作都无法改变状态。2）一旦状态改变，就不会再变，任何时候都可以得到这个结果。这与事件完全不同，事件的特点是，如果错过了它，再去监听，无法得到结果。

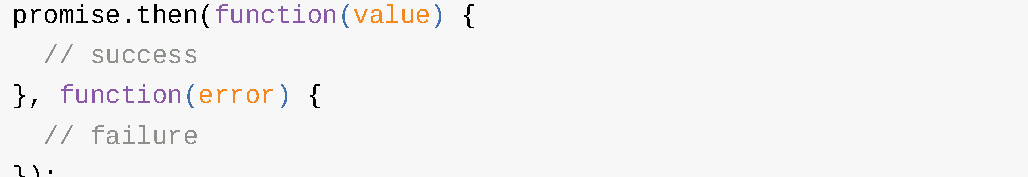
（3）Promise缺点有：1）无法取消Promise，一旦新建它就会立即执行，无法中途取消。2）如果不设置回调函数，Promise内部抛出的错误，不会反应到外部。

（4）Promise对象是一个构造函数，用来生成Promise实例。



**2.Promise.prototype.then**

（1）then方法分别指定Resolved状态和Reject状态的回调函数。



（2）then方法返回的是一个新的Promise实例。注意：1）返回的不是原来那个Promise实例；2）无论是否在then中手动构造promise，then都会对其中的返回结果进行promise包装。

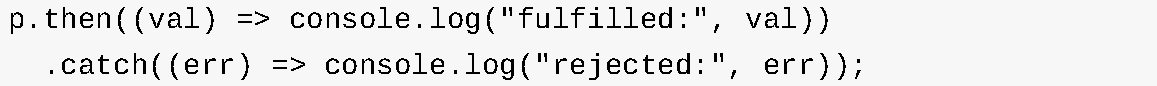




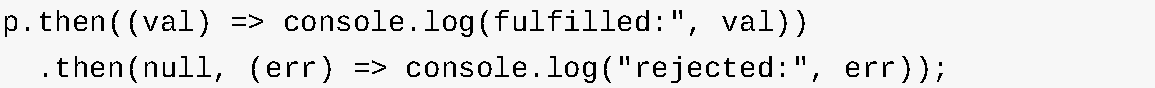
因以上特点，then可以使用链式写法，前面的then的返回值（虽然包装了，但仍然只传递返回值）会传递给后面的then。

**3.Promise.prototype.catch**

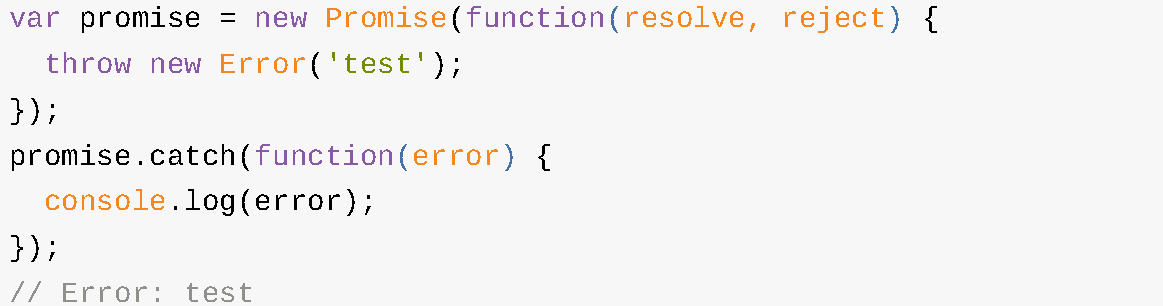
（1）Promise.prototype.catch方法是.then(null,rejection)的别名，用于指定发生错误时的回调函数。



等价于：



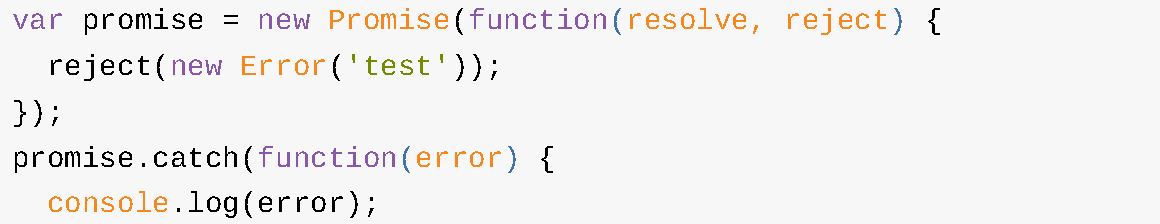
（2）使用throw抛出错误，等同于reject获取/抛出错误。



等价于：



或



（3）如果Promise内语法错误（程序错误，非人为设定错误），then不会执行，也不会收到报错，程序可能终止。

（4）catch方法返回的还是一个Promise对象，因此后面还可以接着调用then方法。要是后面的then方法里面报错，就与前面的catch无关。catch方法之中，还能再抛出错误。可以继续在后面用catch捕获。

**4.Promise.prototype.finally**

无论Promise 对象最后状态如何，都会执行的操作。

**5.Promise.all**

（1）Promise.all方法用于将多个Promise实例，包装成一个新的Promise实例。



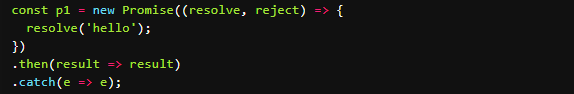
（2）Promise.all方法的参数可以不是数组，但必须具有Iterator接口，且返回的每个成员都是Promise实例。

（3）p的状态由p1、p2、p3决定，分成2种情况：

1）只有p1、p2、p3的状态都变成fulfilled，p的状态才会变成fulfilled，此时p1、p2、p3的返回值组成一个数组，传递给p的回调函数。

2）只要p1、p2、p3之中有一个被rejected，p的状态就变成rejected，此时第一个被reject的实例的返回值，会传递给p的回调函数。

（4）如果1个promise绑定了catch，则它一旦被rejected，并不会触发Promise.all()的catch方法。



**6.Promise.race**



只要p1、p2、p3之中有一个实例率先改变状态，p的状态就跟着改变。

**7.Promise.resolve和Promise.reject**

要将现有对象转为Promise对象。参数分成四种情况：

（1）参数是一个Promise实例，将不做任何修改、原封不动地返回这个实例。

（2）参数是一个thenable对象（是具有then方法的对象），将这个对象转为Promise对象，然后就立即执行thenable对象的then方法。



（3）参数不是具有then方法的对象，或根本就不是对象，返回一个新的Promise对象，状态为Resolved。

（4）不带有任何参数，直接返回一个Resolved状态的Promise对象。

* + 1. async函数

**1.概念**

（1）async函数就是Generator函数的语法糖。直接在普通函数前面加上async，表示这是一个异步函数，在要异步执行的语句前面加上await，表示后面的表达式需要等待。

（2）async函数就是将Generator函数的星号（\*）替换成async（返回1个Promise），将yield替换成await，仅此而已。async函数完全可以看作多个异步操作，包装成的一个Promise对象，而await命令就是内部then命令的语法糖。如下：



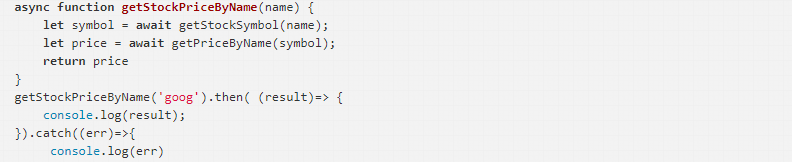
async函数特点有：

（1）有内置执行器，不用调用next。

（2）语义化更明确，相比较于Generator的\*和yield，async和await更明确。

（3）await后面可以跟promise或者任意类型的值。

（4）返回一个promise对象，可以调用then。



**2.语法**

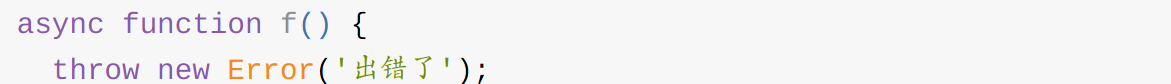
（1）async函数返回一个Promise对象，可以使用then/catch获取数据传出。

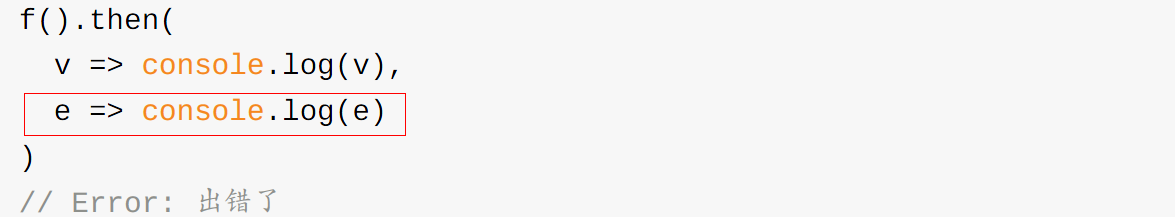
（2）正常情况下，await命令后面是一个Promise对象。如果不是，会被转成一个立即resolve的Promise对象。

（3）async函数返回的Promise对象，必须等到内部所有await命令的Promise对象执行完，才会发生状态改变。

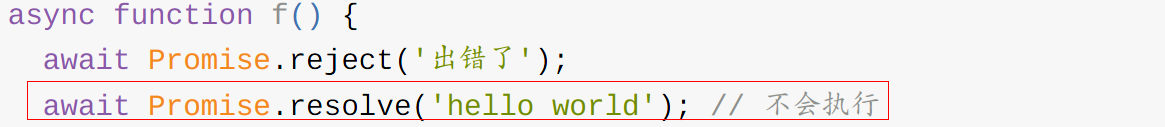


（4）async函数内部抛出错误，会导致返回的Promise对象变为reject状态。





（5）只要一个await语句后面的Promise变为reject，那么整个async函数都会中断执行。为了避免这个问题，可以将第一个await放在try...catch结构里面，这样第二个await就会执行。如果有多个await命令，可以统一放在try...catch结构中。

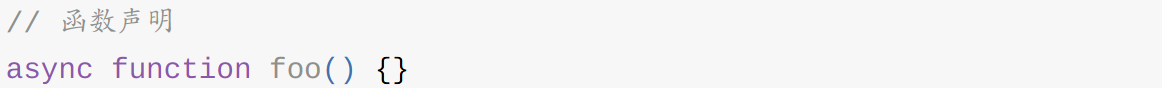


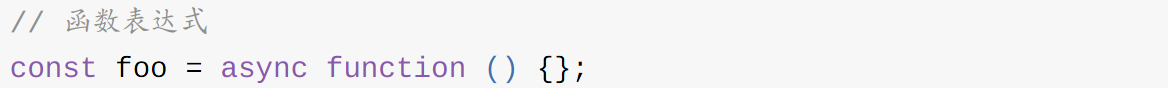


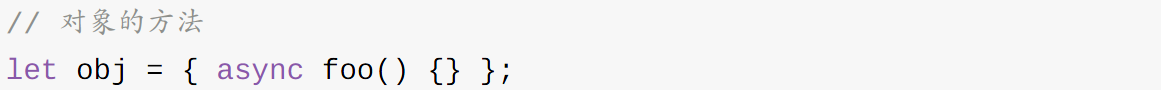
（6）如果await后面的异步操作出错，那么等同于async函数返回的Promise对象被reject。

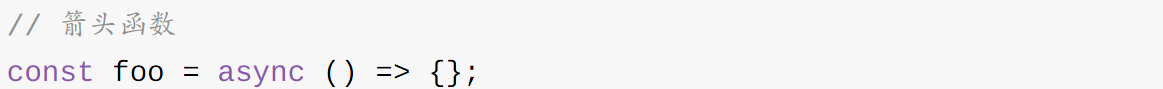
**3.使用**

Async函数有多种使用形式。

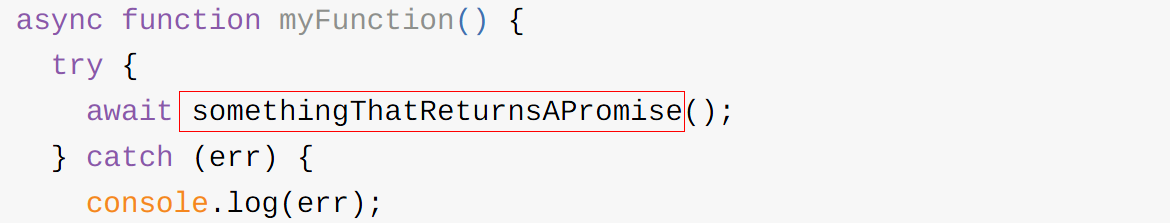


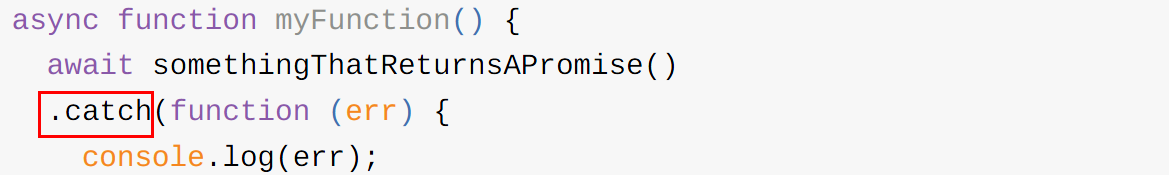




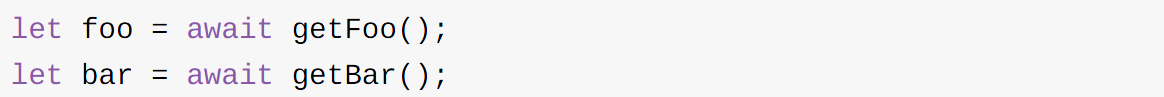


await命令后面的Promise对象，运行结果可能是rejected，所以最好把await命令放在try...catch代码块中。

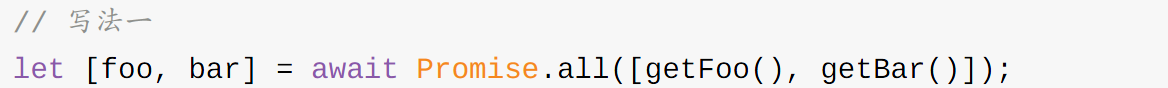


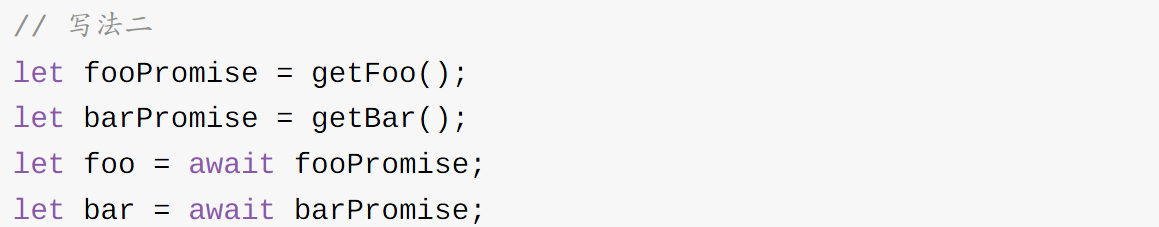


多个await命令后面的异步操作，如果不存在继发关系，最好让它们同时触发。



如果getFoo和getBar是两个独立的异步操作（互不依赖），应该改写为同步方式，否则比较耗时。





await命令只能用在async函数之中，如果用在普通函数，就会报错。

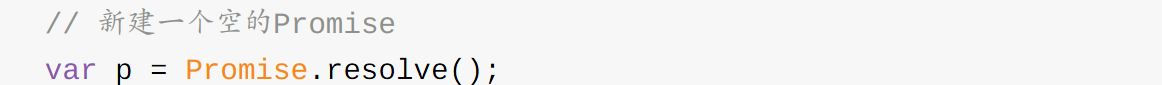
**4.与Promise、Generator的比较**

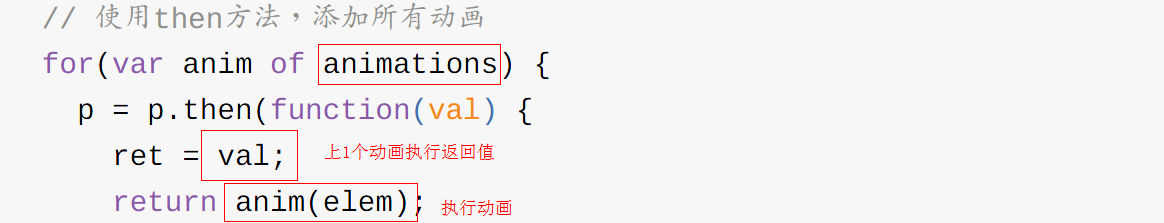
假定某个DOM元素上面，部署了一系列的动画，前一个动画结束，才能开始后一个。如果当中有一个动画出错，就不再往下执行，返回上一个成功执行的动画的返回值。

（1）Promise的写法











（2）Generator函数的写法





需要手动编写执行器。即循环yield。

（3）Async函数的写法

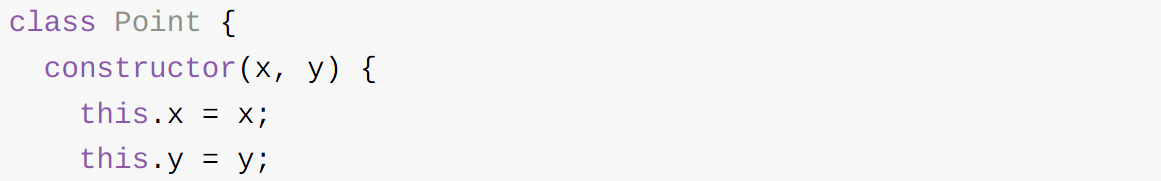


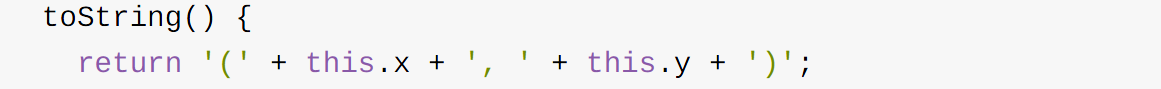
Async函数的实现最简洁。

* 1. Class
     1. 概述

**1.概念**

可以看作只是一个语法糖。





（1）定义方法的时候，前面不需要加上function这个关键字，直接把函数定义放进去了就可以了。

（2）方法之间不需要逗号分隔，加了会报错。

（3）构造函数的prototype属性，在ES6的类上面继续存在。

（4）类内部定义的方法不可枚举（与ES5的行为不一致）。

（5）ES6不提供私有方法，通常只是在命名上加以区别。

（6）可以通过Object.assign方法一次向类添加多个方法。

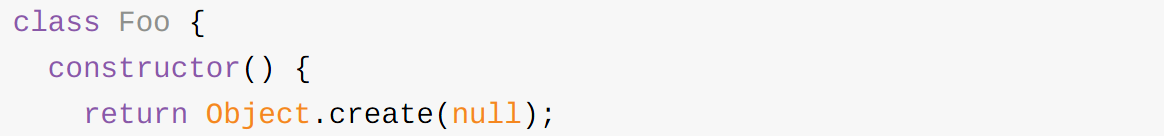


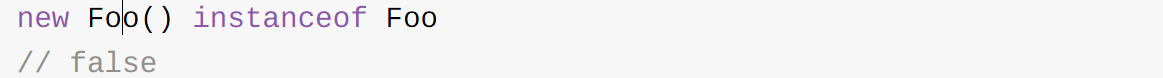
**2.constructor方法**

（1）通过new命令生成对象实例时，自动调用该方法。

（2）一个类必须有constructor方法，如果没有显式定义，一个空的constructor方法会被默认添加。

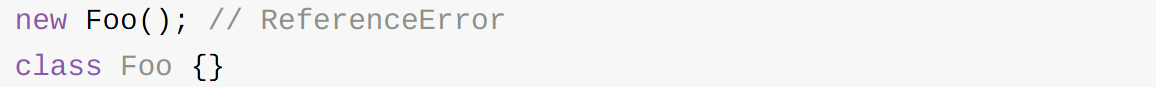
（3）constructor方法默认返回实例对象（即this），完全可以指定返回另外一个对象。





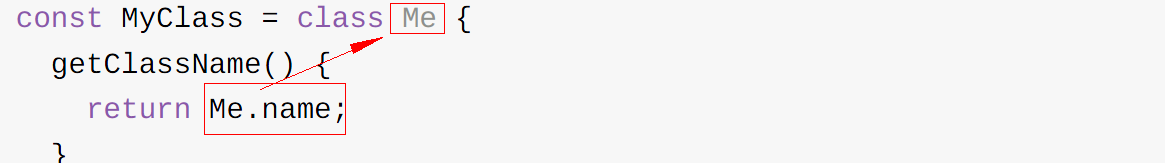
**3.不存在变量提升**

如下，如果Foo类使用在前，定义在后，会报错。



**4.可以使用Class表达式**

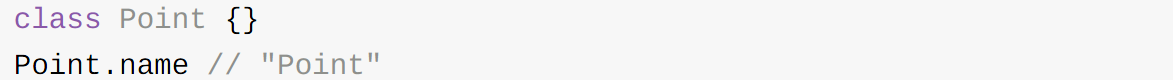
（1）如下，可以将1个Class赋给1个常量。其中，Me只在Class的内部代码可用，指代当前类。如果Class内部没用到的话，可以省略Me。



（2）使用Class表达式还可以写出立即执行的Class。



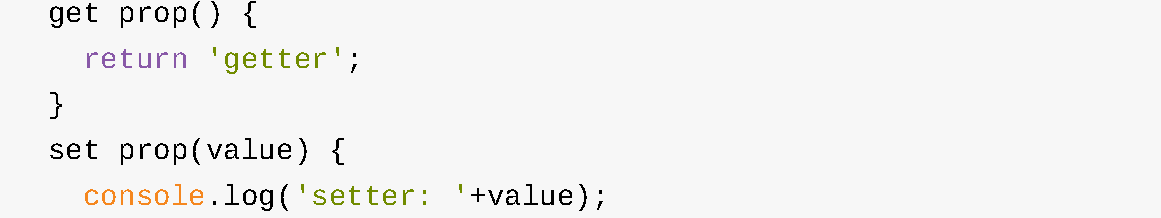
**5.name属性**



**6.getter和setter**

同ES5，在Class内部可以使用get和set关键字，对某个属性设置存值函数和取值函数，拦截该属性的存取行为。





存值函数和取值函数是设置在属性的descriptor对象上的。

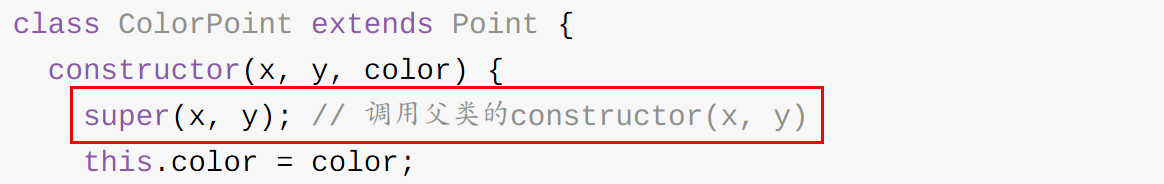
**7.Class的Generator方法**

如果某个方法之前加上星号（\*），就表示该方法是一个Generator函数。

* + 1. 继承

**1.概要**

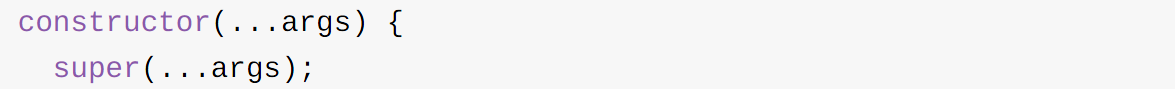




（1）子类必须在constructor方法中调用super方法，否则新建实例时会报错。这是因为子类没有自己的this对象，而是继承父类的this对象，然后对其进行加工。如果不调用super方法，子类就得不到this对象。

ES6的继承机制完全不同，实质是先创造父类的实例对象this（所以必须先调用super方法），然后再用子类的构造函数修改this。

（2）如果子类没有定义constructor方法，这个方法会被默认添加



**2.类的prototype属性和\_\_proto\_\_属性**

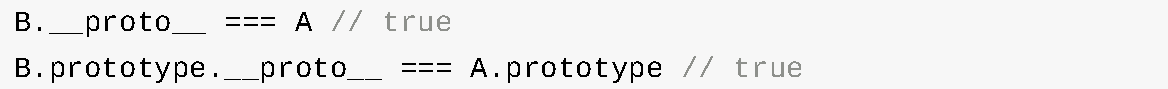
ES5中，每一个对象都有\_\_proto\_\_属性，指向对应的构造函数的prototype属性（原型）。Class作为构造函数的语法糖，同时有prototype属性和\_\_proto\_\_属性，因此同时存在两条继承链：

（1）子类的\_\_proto\_\_属性，表示构造函数的继承，总是指向父类。

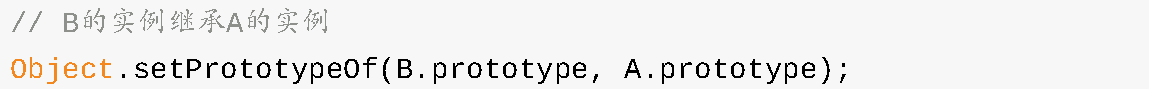
（2）子类prototype属性的\_\_proto\_\_属性，表示方法的继承，总是指向父类的prototype属性。

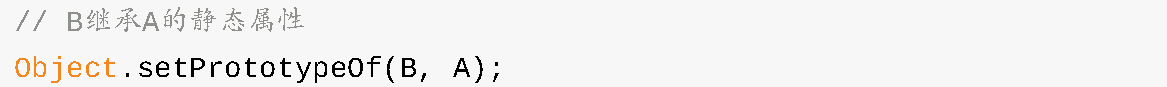




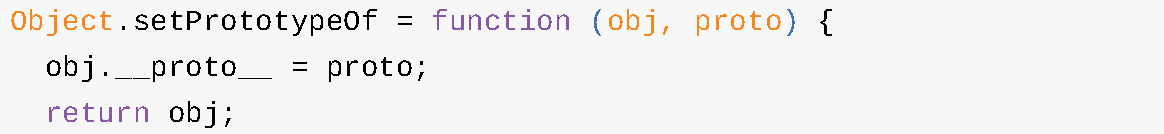


之所以如此，是因为类的继承是按照如下模式实现：





setPrototypeOf实现如下：



可以理解为：作为一个对象，子类（B）的原型（\_\_proto\_\_属性）是父类（A）；作为一个构造函数，子类（B）的原型（prototype属性）是父类的实例。

**3.实例的\_\_proto\_\_属性**

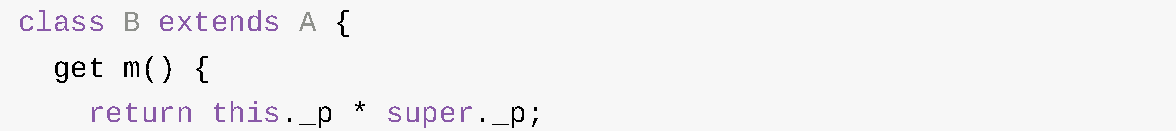
子类实例的\_\_proto\_\_属性的\_\_proto\_\_属性，指向父类实例的\_\_proto\_\_属性。即，子类的原型的原型，是父类的原型。

**4.super关键字**

super关键字有2种含义：

（1）作为函数调用时（即super(...args)），super代表父类的构造函数。

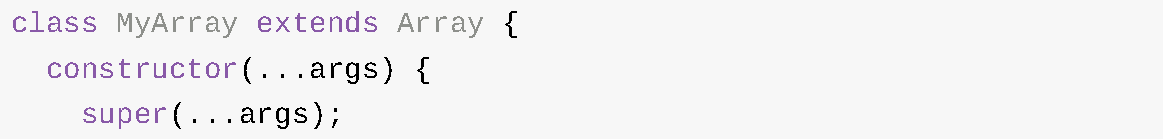
（2）作为对象调用时（即super.prop或super.method()），super代表父类。注意，此时super即可以引用父类实例的属性和方法，也可以引用父类的静态方法。



**5.原生构造函数的继承**

原生构造函数是指语言内置的构造函数，通常用来生成数据结构。主要有：Boolean、Number、String、Array、Date、Function、RegExp、Error、Object。

以前，这些原生构造函数是无法继承的。ES6允许继承原生构造函数定义子类，因为ES6是先新建父类的实例对象this，然后再用子类的构造函数修饰this，使得父类的所有行为都可以继承。



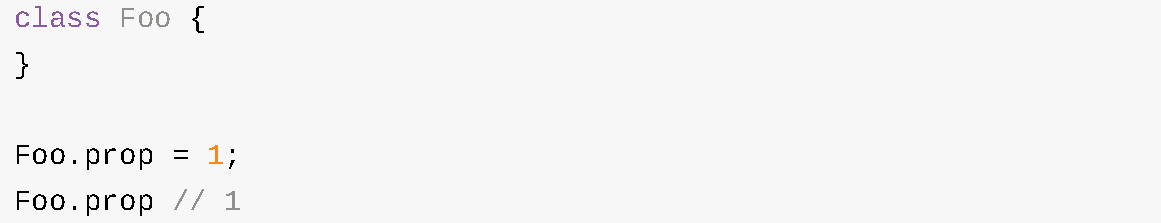
* + 1. 静态属性和方法

（1）如果在方法前加上static，则该方法不能在在实例中访问，而是直接通过类来调用，即静态方法。

（2）父类的静态方法，可以被子类继承。

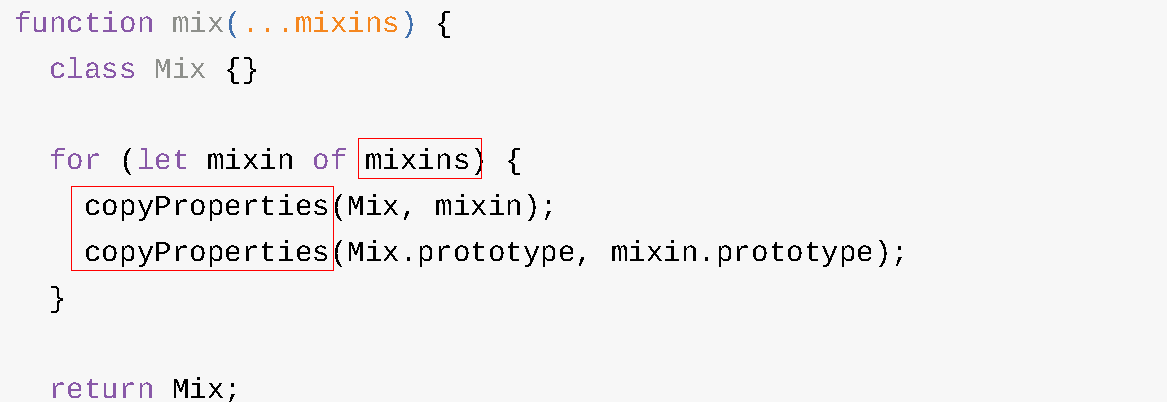
（3）静态方法也是可以从super对象上调用的，也可以在子类中重写。

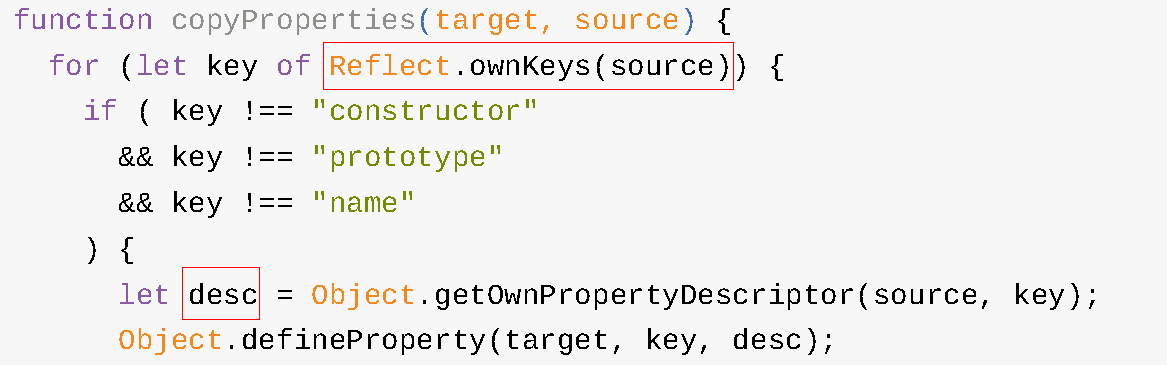
（4）静态属性指的是Class本身的属性，即Class.propname。只有如下方法可以定义静态属性。



* + 1. Mixin模式的实现

将多个类的接口混入（mix in）另一个类。ES6的实现如下。





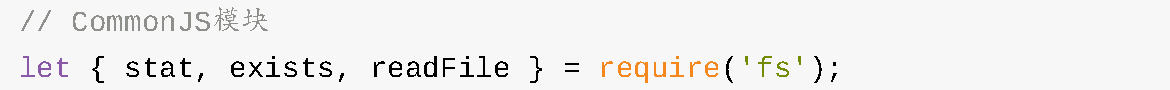
上面的mix函数就可以将多个对象合成为一个类。



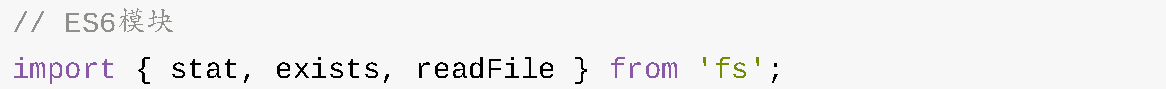
* 1. Module
     1. 概要

**1.比较**

CommonJS和AMD模块，都只能在运行时确定模块的依赖关系。ES6模块的设计思想，是尽量的静态化。



上面需要先整体加载fs模块（即加载fs的所有方法），生成一个对象，然后再从这个对象上面读取3个方法。这种加载称为“运行时加载”，因为只有运行时才能得到这个对象，导致完全没办法在编译时做“静态优化”。



上面代码的实质是从fs模块加载3个方法，其他方法不加载。这种加载称为“编译时加载”。

由于ES6模块是编译时加载，使得静态分析成为可能，可以进一步拓宽JavaScript的语法，比如引入宏（macro）和类型检验（type system）这些只能靠静态分析实现的功能。

**2.export命令**

模块功能主要由两个命令构成：export和import。export命令用于规定模块的对外接口，import命令用于输入其他模块提供的功能。

export有2种写法，优先使用后者。

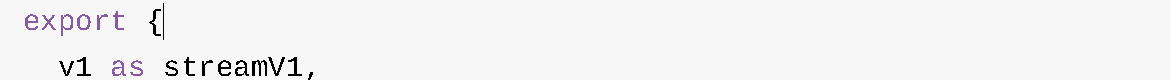




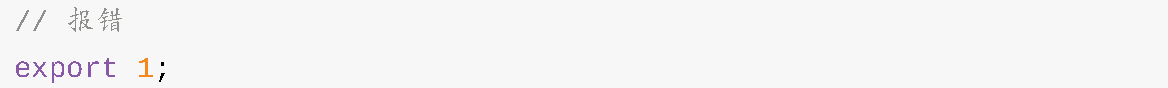
也可以输出函数。

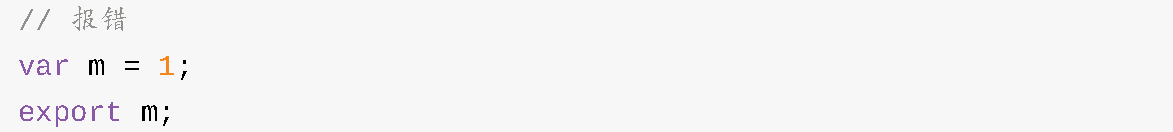


export输出的变量就是本来的名字，但是可以使用as关键字重命名。



如下方式会报错：





与CommonJS规范完全不同，export语句输出的接口，与其对应的值是动态绑定关系，即通过该接口，可以取到模块内部实时的值。

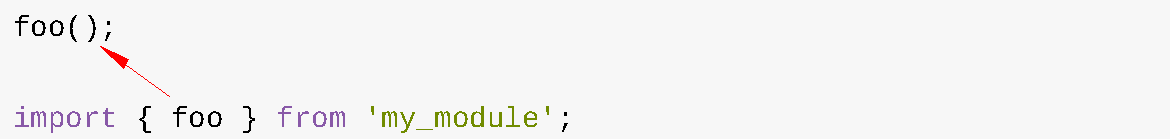
**3.import命令**



（1）重命名

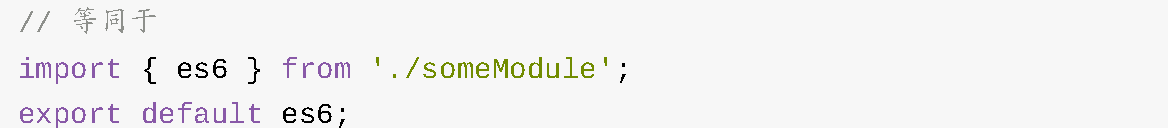


（2）import命令具有提升效果，会提升到整个模块的头部，首先执行。



（3）如果在一个模块之中，先输入后输出同一个模块，import语句可以与export语句写在一起。但不建议写在1行。





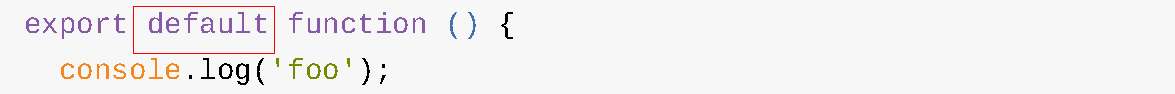
**4.模块的整体加载**

使用星号（\*）指定一个对象，所有输出值都加载在这个对象上面。

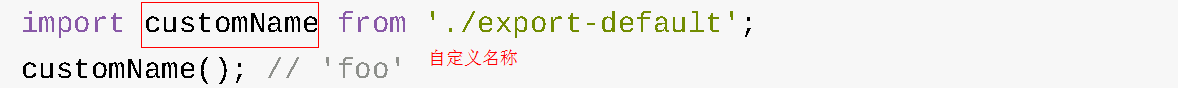


**5.export default命令**

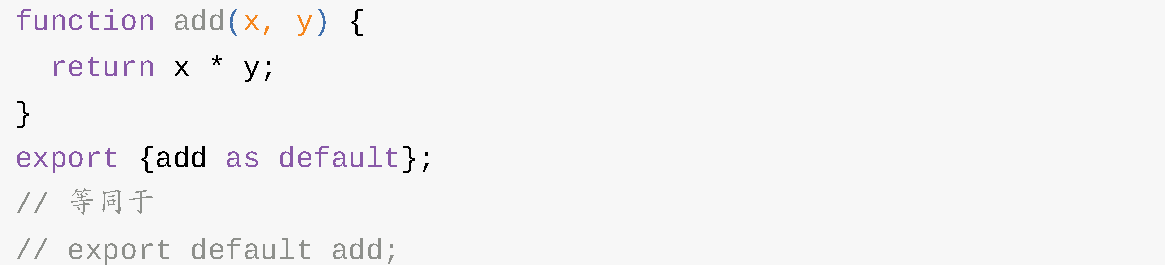
（1）为不用阅读文档就能加载模块，可使用export default命令，为模块指定默认输出。

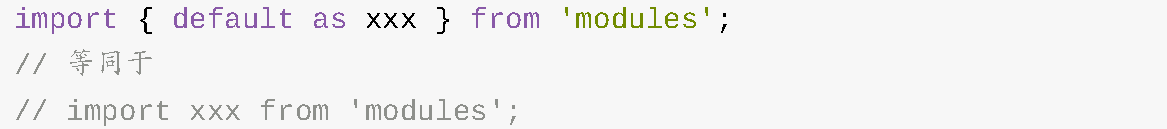






（2）export deault命令只能使用一次，因此import命令后面不使用大括号。本质上，export default就是输出一个叫做default的变量或方法，然后系统允许你为它取任意名字。





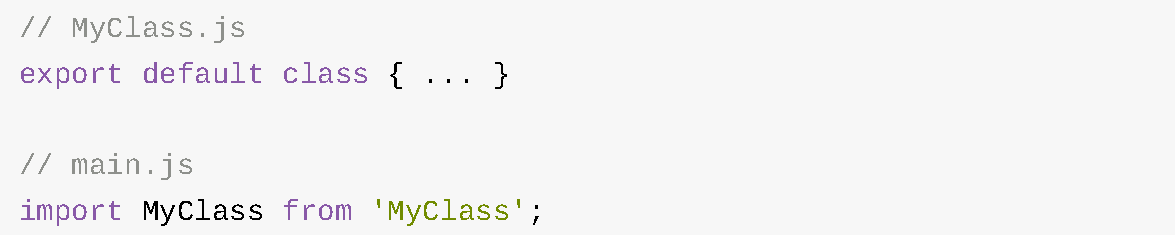
（3）如果想在一条import语句中，同时输入默认方法和其他变量，可以写成下面这样。



（4）如果要输出默认的值，只需将值跟在export default之后即可。

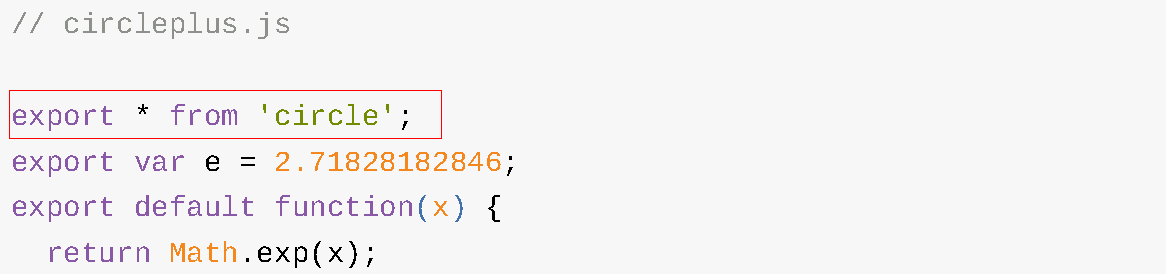


（5）export default也可以用来输出类。



* + 1. 模块的继承

假设有一个circleplus模块，继承了circle模块。



（1）export\*，表示再输出circle模块的所有属性和方法。

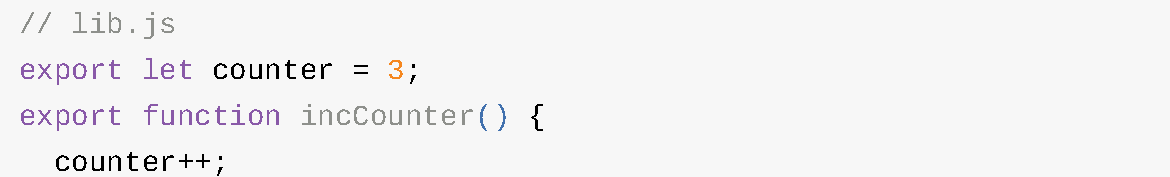
（2）export\*命令会忽略circle模块的default方法。

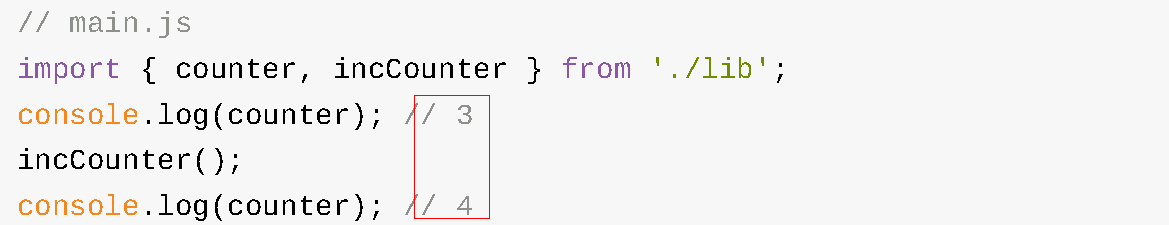
（3）可以将circle的属性或方法，改名后再输出。



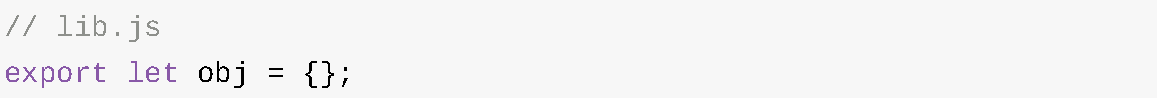
* + 1. ES6模块加载的实质

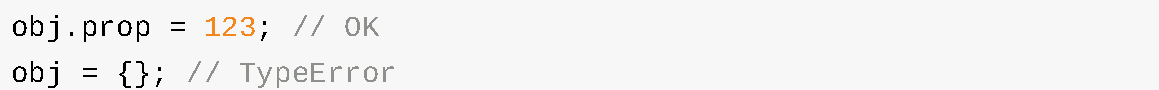
与CommonJS模块完全不同。CommonJS模块输出的是一个值的拷贝，而ES6模块输出的是值的引用。





由于ES6输入的模块变量，只是一个“符号连接”，所以这个变量是只读的，对它进行重新赋值会报错。



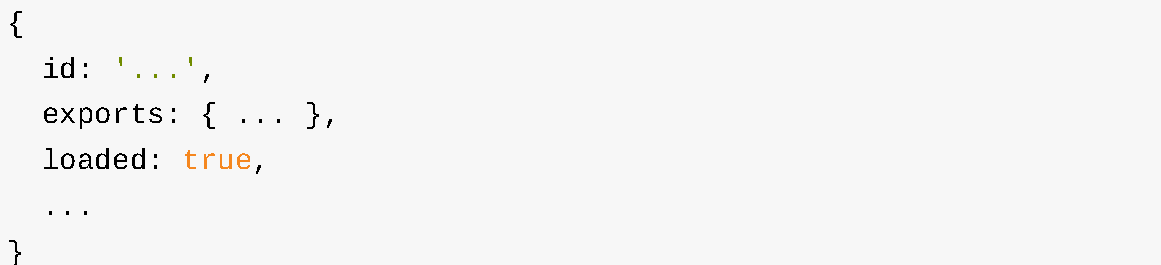


* + 1. 循环加载

如，a脚本的执行依赖b脚本，而b脚本的执行又依赖a脚本。

**1.CommonJS模块的加载原理**

CommonJS在执行require命令时，第1次加载/执行该脚本，在内存中生成1个对象：

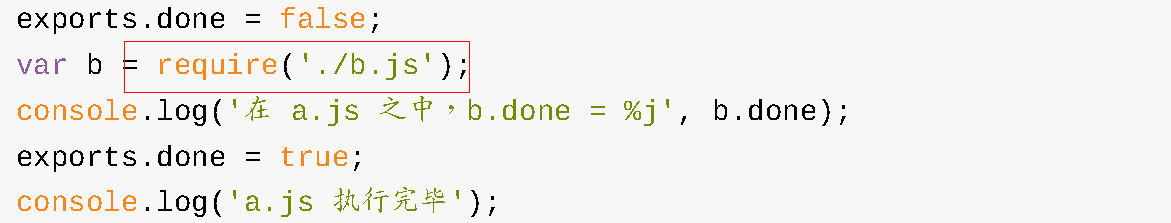


id为模块名，exports为模块输出的各个接口，loaded为布尔值，表示该模块的脚本是否执行完毕。即使以后再次执行require命令，也不会再次执行该模块，而是到缓存之中取值。即，无论CommonJS模块加载多少次，都只会在第1次加载时运行1次，以后再加载，就返回第1次运行的结果，除非手动清除系统缓存。

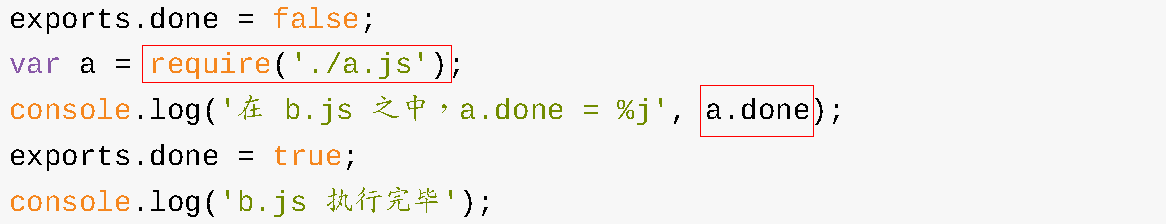
**2.CommonJS模块的循环加载**

CommonJS模块的重要特性是加载时执行。

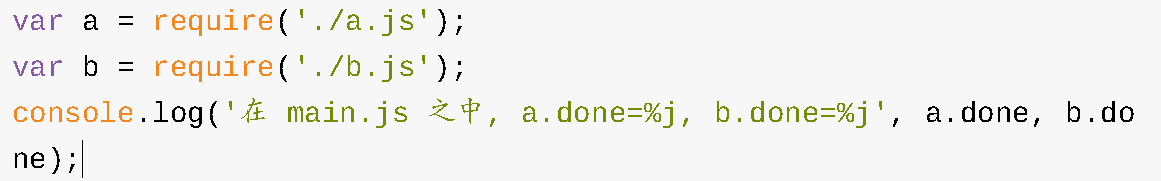
a.js



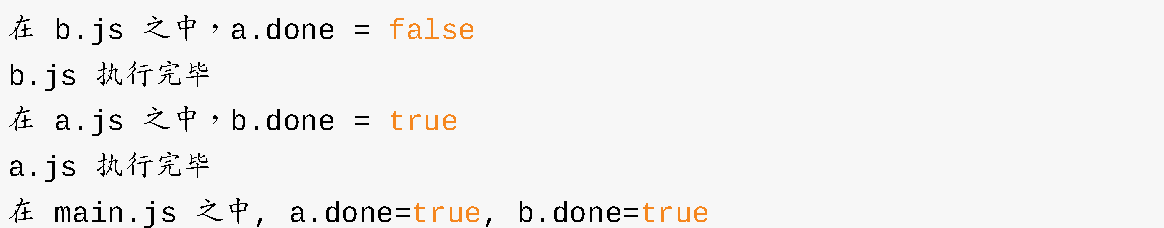
b.js



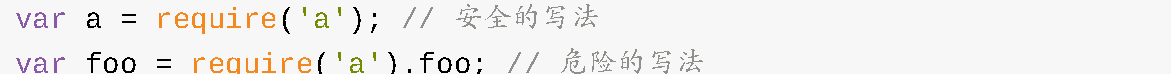
main.js



执行a.js时，第2行就去执行b.js，等待b.js执行完毕。在b.js中，第2行循环加载，因a.js还没有执行完，此时就只返回已执行完的部分，而不是最终的值，a.js已经执行的部分，只有一行，即exports.done=false。之后，b.js继续执行，完成后再将执行权交给a.js。

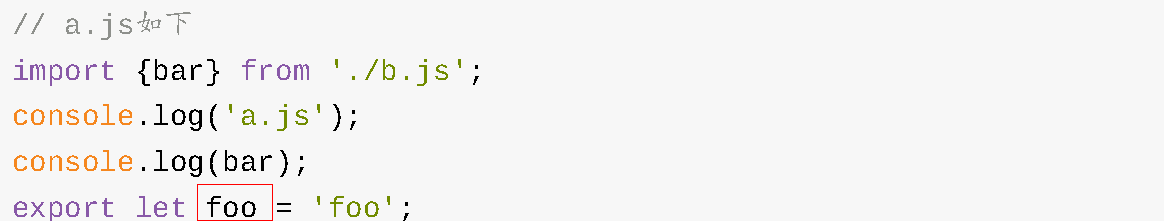


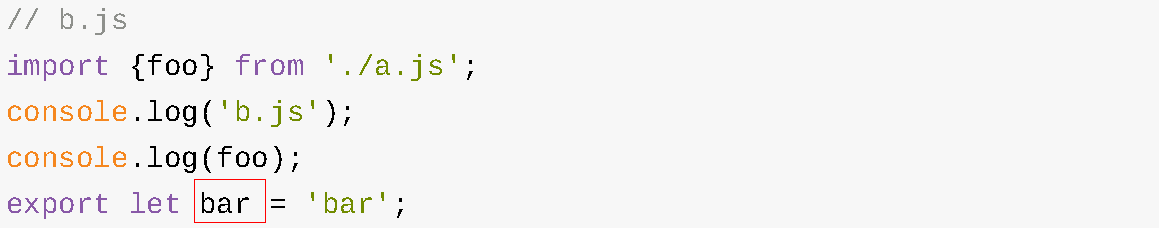
即，CommonJS模块遇到循环加载时，返回的是当前已经执行的部分的值，而不是代码全部执行后的值。使用时，在输入变量的时候，尽量使用全部值。



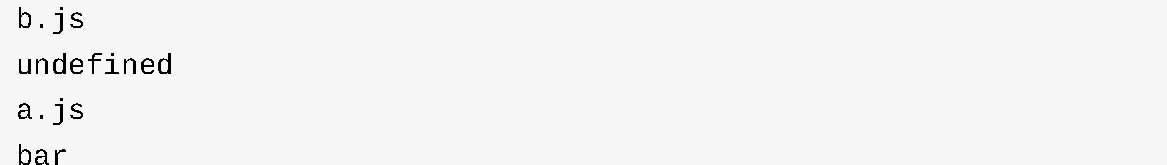
**3.ES6模块的循环加载**

ES6模块是动态引用，需要开发者自己保证，真正取值的时候能够取到值。





执行a.js，结果为：



1. 其他
   1. ES5扩展说明

**1.严格模式**

新增严格模式。

**2.JSON**

新增JSON对象。

**3.String扩展**

新增String.prototpye.trim。

**4.Object扩展**

（1）create、defineProperty、getOwnPropertyDescriptor。

（2）遍历属性

Object.getOwnPropertyNames、Object.keys

（3）限制对象

Object.preventExtensions、Object.isExtensible、seal、Object.isSealed、freeze、isFrozen。

（4）新增get和set方法。

**5.Array扩展**

（1）indexOf/lastIndexOf。

（2）map、forEach、filter、some、every、reduce/reduceRight。

**6.Function扩展**

新增bind方法。

**7.Date**

新增Date.now。

* 1. 严格模式

主要有以下限制：

（1）变量必须声明后再使用。

（2）函数的参数不能有同名属性，否则报错。

（3）不能对只读属性赋值，否则报错。

（4）不能删除不可删除的属性，否则报错。

（5）不能删除变量delete prop，会报错，只能删除属性delete global[prop]。

（6）禁止this指向全局对象。

（7）arguments不会自动反映函数参数的变化。不能使用arguments.callee。不能使用arguments.caller。

（8）eval不会在它的外层作用域引入变量。eval和arguments不能被重新赋值。

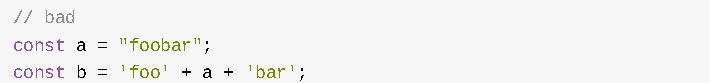
（9）不能使用fn.caller和fn.arguments获取函数调用的堆栈。

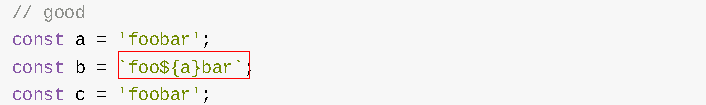
（10）不能使用前缀0表示八进制数，否则报错。

（11）不能使用with语句。

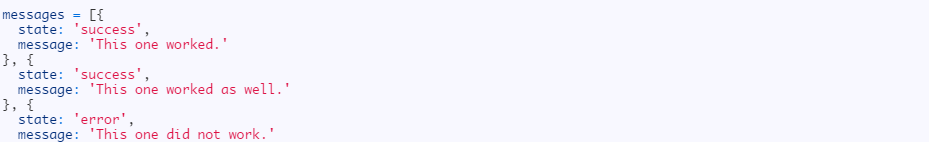
（12）增加了保留字（比如protected、static和interface）。

* 1. 编程风格
     1. 字符串
* 静态字符串一律使用单引号或反引号，不使用双引号。动态字符串使用反引号。

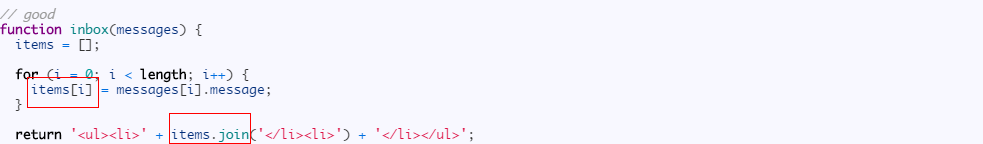




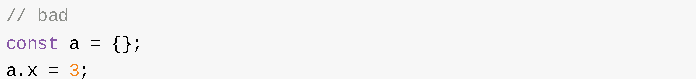
* 大于80个元素的字符串需要通过分隔符进行多行操作（用+连起来）。
* 通过编程的方式创建字符串，应该使用数组的join方法，而不是字符串链接方法。特别是对于IE而言。

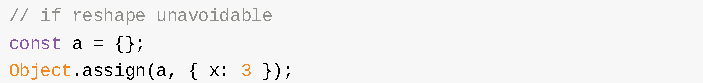


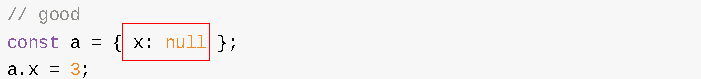




* + 1. 对象
* 使用字面量语法来创建对象。
* 使用点符号.来访问属性，除非表达式。
* json对象属性名加双引号。
* 对象尽量静态化，一旦定义，就不得随意添加新的属性。如果添加属性不可避免，要使用Object.assign方法。







* 如果对象的属性名是表达式（需要计算求值），使用属性表达式定义。





* 对象的属性和方法，尽量采用简洁表达法。
* 注意区分Object和Map，只有模拟现实世界的实体对象时，才使用Object。如果只是需要 key: value 的数据结构，使用Map结构。
  + 1. 数组
* 使用字面量语法来创建数组。
* 如果你不知道数组长度，数组添加成员使用push方法。
* 使用扩展运算符（...）或slice拷贝数组。

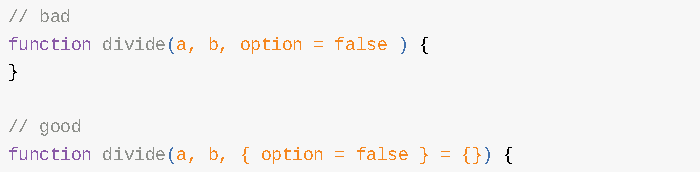




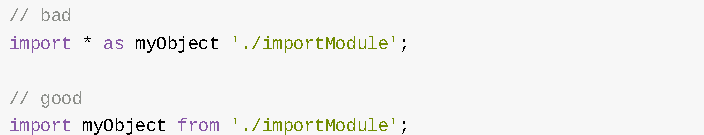
* 使用Array.from方法，将类似数组的对象转为数组。



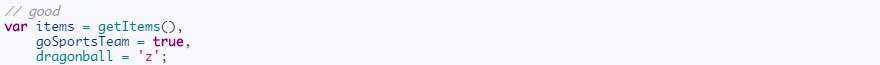
* + 1. 函数
* 需要使用函数表达式的场合，尽量用箭头函数代替。
* 箭头函数取代Function.prototype.bind，不应再用self/\_this/that绑定this。
* 简单的、单行的、不会复用的函数，建议采用箭头函数。如果函数体较为复杂，行数较多，还是应该采用传统的函数写法。
* 所有配置项都应该集中在一个对象，放在最后一个参数，布尔值不可以直接作为参数。



* 不要在函数体内使用arguments变量，使用rest运算符（...）代替。
* 使用默认值语法设置函数参数的默认值。
  + 1. Class
* 总是用Class，取代需要prototype的操作。
  + 1. 模块
* 使用import取代require。
* 使用export取代module.exports。
* 如果模块只有一个输出值，就使用export default，如果模块有多个输出值，就不使用export default，不要export default与普通的export同时使用。
* 不要在模块输入中使用通配符。因为这样可以确保你的模块之中，有一个默认输出（export default）。

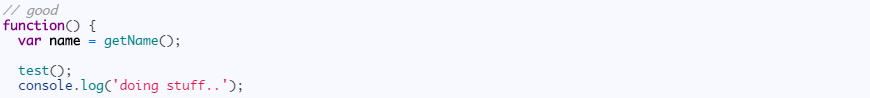


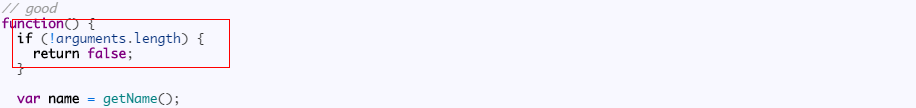
* + 1. 块级作用域
* 使用let取代var。
* 在let和const之间，优先使用const，尤其是在全局环境，不应该设置变量，只应设置常量。这符合函数式编程思想，有利于将来的分布式运算。
* 所有的函数都应该设置为常量（const）。
* 多个变量时只使用一个var/let声明，每个变量占一新行。



* 在范围内将变量赋值置顶，这有助于避免变量声明和赋值提升相关的问题





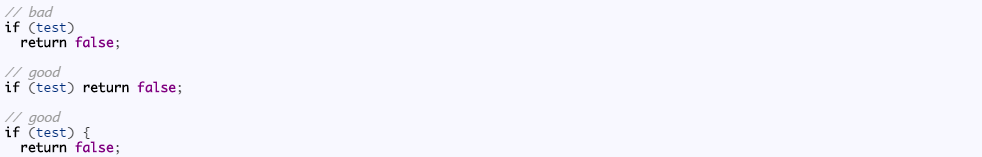


* + 1. 条件表达式和相等
* 使用简易方式。





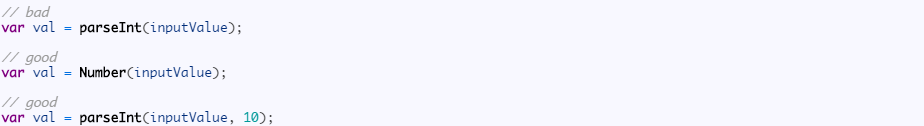
* 在多行块中使用大括号。



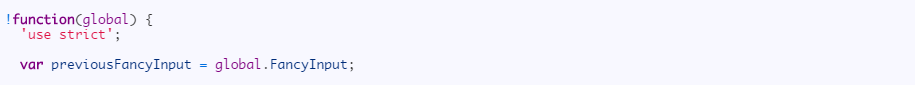
* 多行注释使用/\*\* ... \*/，包括描述，指定类型、所有参数的值和返回值。单行注释使用//,在注释的内容前另起一行进行单行注释，并在注释前留一空行。
  + 1. 转型&强制
* Strings



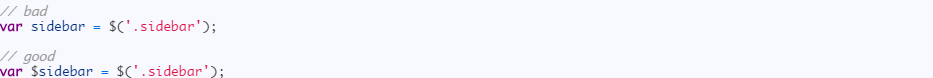
* 使用parseInt对Numbers型进行强制转型。



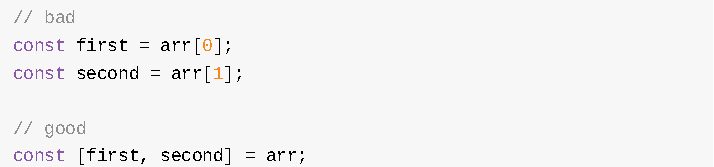
* 储存this的引用使用\_this。
* 模块应该以！开始，以确保当模块忘记包含最后一个分号时，在脚本连接时不会报错。



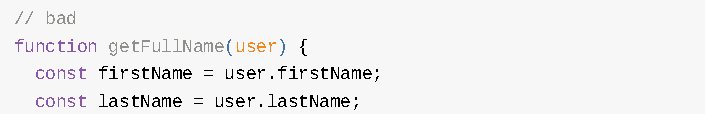
* JQuery对象变量前缀使用$。



* + 1. 解构赋值
* 使用数组成员对变量赋值时，优先使用解构赋值。



* 函数的参数如果是对象的成员，优先使用解构赋值。





* 如果函数返回多个值，优先使用对象的解构赋值，而不是数组的解构赋值。

