# 一、ES6

## let、const和var的概念与区别

### 1.var

（1）局部作用域

（2）变量提升

（3）可重复声明

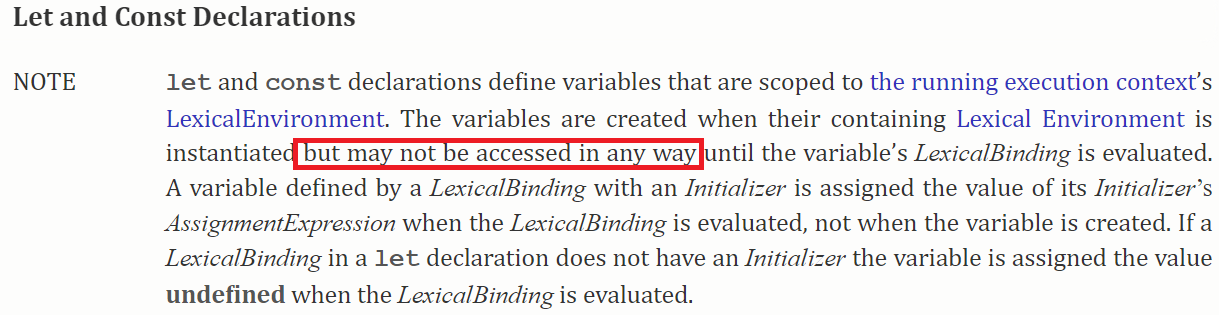
### 2.let

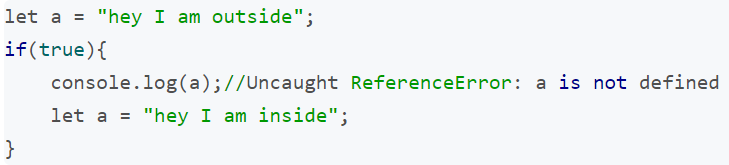
（1）块级作用域

（2）相同作用域内不可重复声明

（3）声明前不允许赋值（相当于创建全局变量，类似于重复声明）

（4）或许存在变量提升，但不允许访问





（如果块内的a没有提升，那么console.log(a)应该访问全局变量a，但我很好奇，如果编译器只是单纯地对当前作用域所有叫a的变量进行禁止访问，如果上面这个步骤在不提升或提升之前呢？这是我们无法知晓的）

**我感觉最好的解释**：

当前作用域顶部到该变量声明位置中间的部分，都是该let变量的死区，在死区中，禁止访问该变量。由此，我们给出结论，**let声明的变量存在变量提升， 但是由于死区我们无法在声明前访问这个变量（或者知晓是否存在提升）。**

**新：看了一篇新的博客https://blog.csdn.net/u013263917/article/details/106673193，居然可以通过控制台看到，函数中的var、let变量在函数开始时就存在于local作用域中了。**

（5）暂时性死区

### 3.const

（1）块级作用域

（2）相同作用域内不可重复声明

（3）声明前不允许赋值（相当于创建全局变量，类似于重复声明）

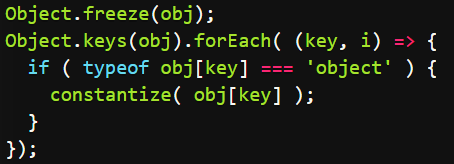
（4）不存在变量提升

（5）暂时性死区

（6）必须立刻赋值，并且不可改变

（7）const保证变量的内存地址指向不变，但是对于引用类型，只能保证指向的对象不变，不能保证对象的值不变。

若想保证完全不变，可利用Object.freeze冻结对象及其属性



## 变量提升与暂时性死区

### 1.变量提升

**定义：**在执行上下文的创建阶段，将变量、函数的声明提到当前作用域最顶端。

**提升顺序：** 先执行变量提升，再执行函数提升。但是函数声明的优先级最高，会被提升至当前作用域最顶端。

**Javascript仅提升声明：**对变量、函数表达式提升后不初始化，强行使用结果为undefined。

对函数会提升声明并赋值，在定义函数前就可调用函数

### 2.暂时性死区

**定义**：使用let、const修饰变量时，从其所在作用域顶部到其声明之前，都无法对该变量进行访问。

## 变量的解构赋值

### 1.定义

按照一定的模式从数组或者对象中取值，对变量进行赋值的过程称为解构。

### 2.用法

#### （1）数组

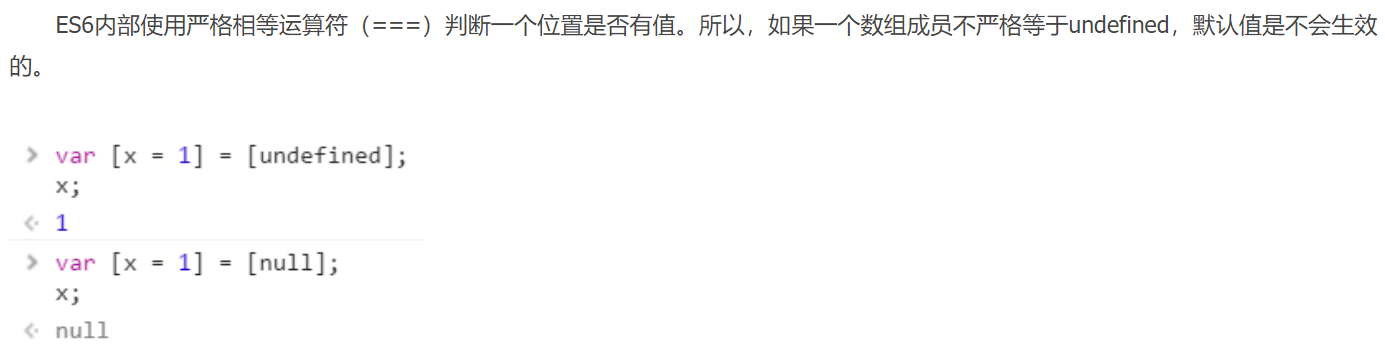
**<标准赋值>**



**<默认值>**



注：

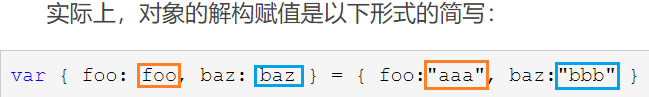


#### （2）对象

**<标准赋值>**



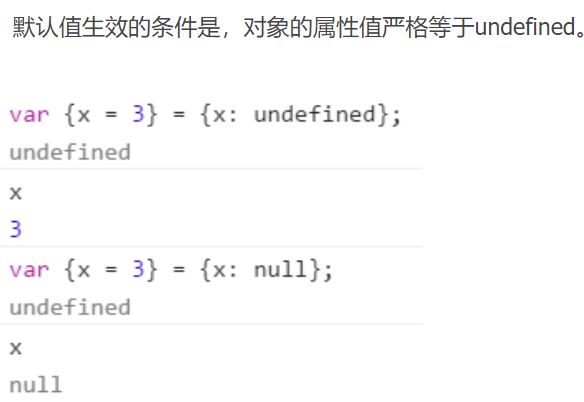
对象的属性没有次序，变量必须与属性同名，才能取到正确的值。



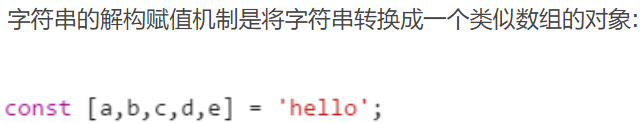
**<默认赋值>**

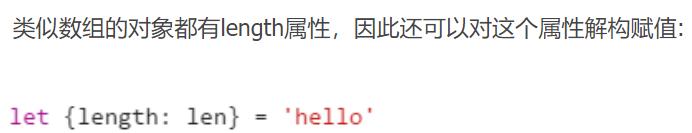


注：



#### （3）字符串的解构赋值



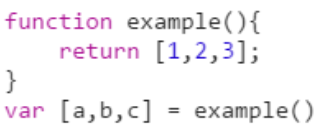


#### （4）用途

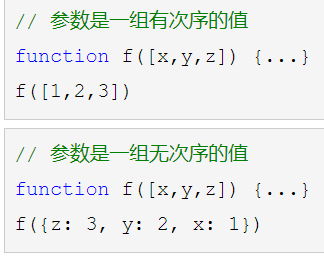
**1）交换变量的值**

;[a,b]=[b,a];

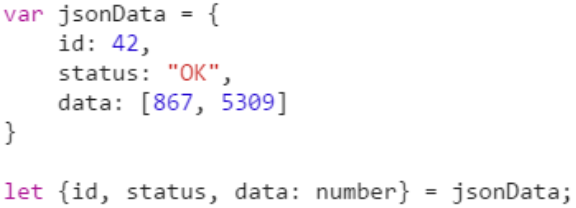
**2）从函数返回多个值**



**3）函数参数的定义**



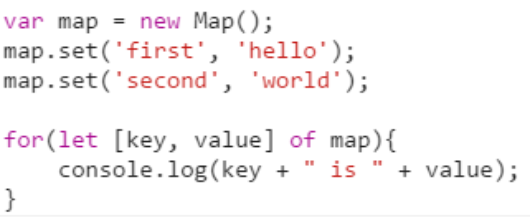
**4）提取JSON数据**



**5）函数参数的默认值**



**6）遍历Map解构**



**7）输入模块的指定方法**



## 箭头函数及其this问题

### 1.定义

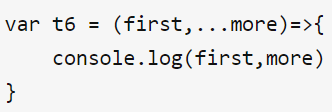
ES6可以使用“箭头”（=>）定义函数

### 2.与普通函数的区别

1.箭头函数的this是作用域链中离自己最近函数的this

2.不能作为constructor，因此不能用new调用

3.没有argument属性，可以通过...rest获取



4.不能直接使用call、apply、bind修改this

5.不能使用yield，不能用来作为generator函数

## Symbol概念及其作用

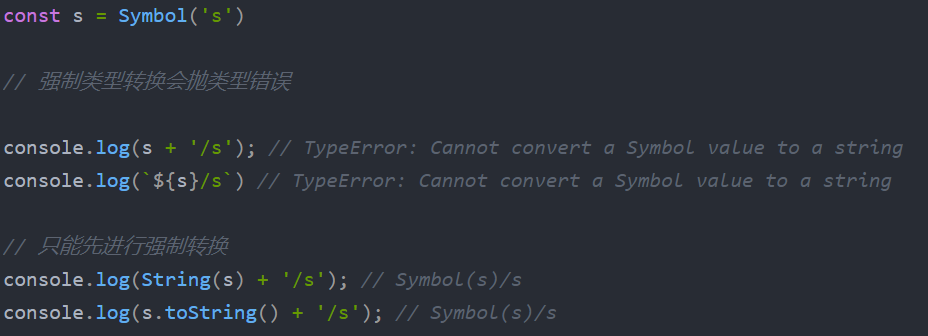
### 1.定义

Symbol 是 ES6 新推出的一种基本类型，它表示独一无二的值，它可以接受一个字符串作为参数，创建Symbol的参数只是表示Symbol值的描述而已，不代表它们相等。主要用于程序调试时的跟踪。

### 2.特征

（1）不是完整的构造函数，不能通过new创建。

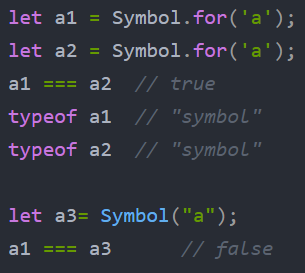
（2）不能进行隐式类型转换，不能转为数字，但是可以转为字符串、布尔值。



### 3.方法

**（1）Symbol.for()**

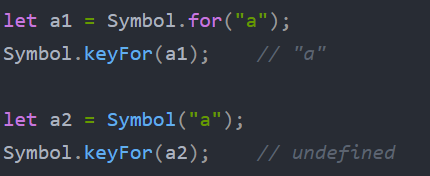
Symbol.for()是用于将描述相同的Symbol变量指向同一个Symbol值



它跟symbol()的区别是Symbol()定义的值每次都是新建，即使描述相同值也不相等，而Symbol.for()定义的值会先检查给定的描述是否已经存在，如果不存在才会新建一个值，并把这个值登记在全局环境中供搜索，Symbol.for()定义相同描述的值时会被搜索到，描述相同则他们就是同一个值

**（2）Symbol.keyFor()**

​ Symbol.keyFor()是用来检测该字符串参数作为名称的 Symbol值是否已被登记，返回一个已登记的 Symbol 类型值的key



### 4.属性

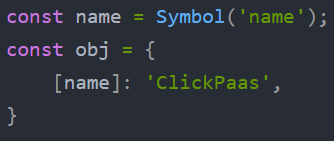
**Symbol.prototype.description**



### 5.用途

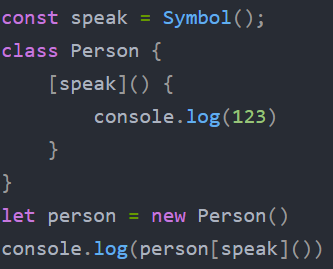
**1、作为对象属性**

当一个复杂对象中含有多个属性的时候，很容易将某个属性名覆盖掉，利用 Symbol 值作为属性名可以很好的避免这一现象



**2、模拟私有属性**

ES6 中的类是没有 private 关键字来声明类的私有方法和私有变量的，但是我们可以利用 Symbol 的唯一性来模拟



## Set和Map数据结构

### 1.Set

#### （1）定义

**Set是ES6新的数据结构，类似数组，但成员的值是唯一的，没有重复的值。**

#### （2）初始化

let s = new Set()

let s = new Set([1,1,2,2,3])

#### （3）添加

s.add(2)

#### （4）查询

s.has(2)

接受一个值，并验证当前对象是否包含指定的值。如果是这样，则此函数返回布尔值true，否则返回false。

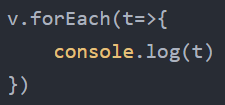
#### （5）删除

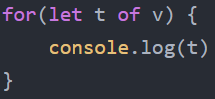
s.delete(2)

#### （6）获取个数

s.size

#### （7）遍历





#### （8）转为数组



### 2.map

Map(映射)是一种可迭代的键值对（key/value）结构。

所有的值都可以通过键来获取。

Map 中的键都是唯一的。

#### （1）定义

var m = new Map([['Michael', 95], ['Bob', 75], ['Tracy', 85]]);

#### （2）获取键对应的值

m.get('Michael'); // 95

#### （3）查询

m.has(key)

#### （4）删除

m.delete(key)

#### （5）添加

m.set(key, value)

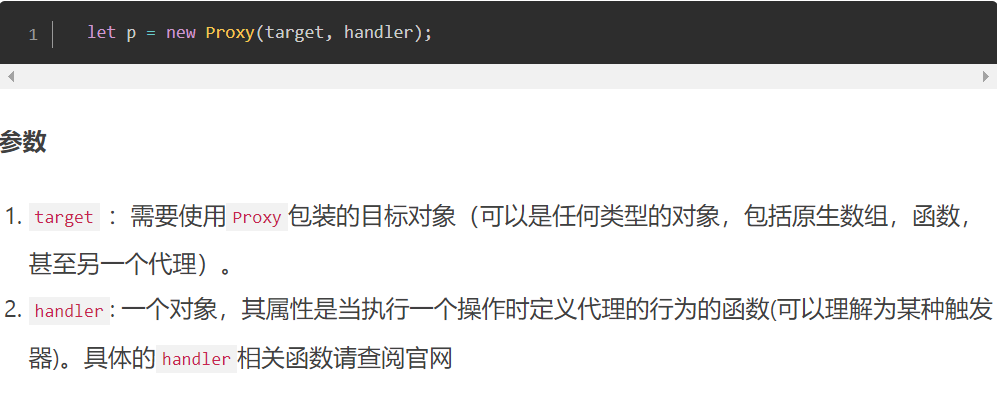
### weakSet、weakMap

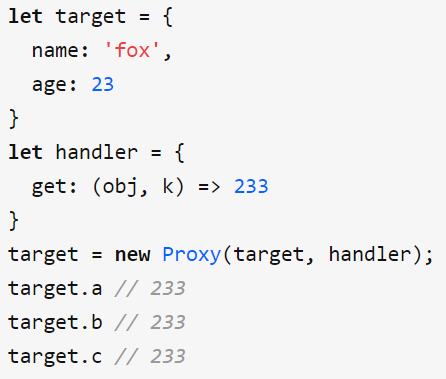
## Proxy

### 1.定义

proxy接受一个待代理目标对象和一些包含元操作的对象，为待代理目标创建一个‘屏障’，并拦截所有操作，重定向到自定义的元操作对象上。

### 2.使用





## Reflect对象

### 1.定义：

Reflect是一个内建的对象，用来提供方法去拦截JavaScript的操作。Reflect不是一个函数对象，所以它是不可构造的，也就是说它不是一个构造器，你不能通过`new`操作符去新建或者将其作为一个函数去调用Reflect对象。Reflect的所有属性和方法都是静态的。

### 2.作用

反射机制：指的是程序在运行时能够获取自身的信息

相当于说提供Reflect对象将这些能够实现反射机制的方法都归结于一个地方并且做了简化，保持JS的简单

### 3.使用

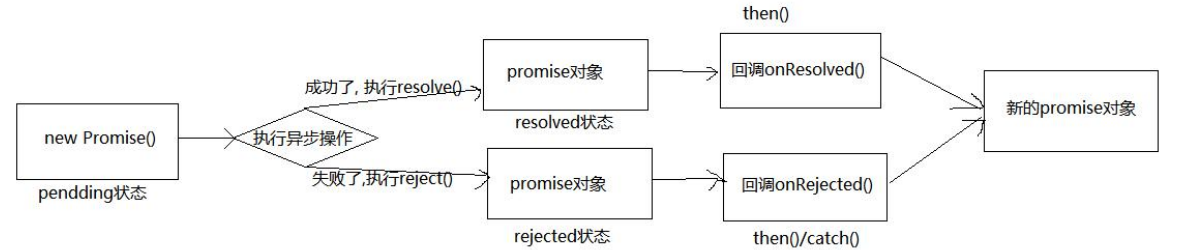




## Promise（手撕Promise A+规范、Promise.all、Promise相关API和方法）

### 1.定义

Promise 是异步编程的一种解决方案，比传统的解决方案——回调函数和事件——更合理且更强大。（之前用纯回调函数解决）



### 2.优势

#### （1）比纯回调灵活

可以在异步任务执行完成后定义回调函数

#### （2）链式调用

### 3.方法

#### （1）状态

pending: 初始状态，不是成功或失败状态。

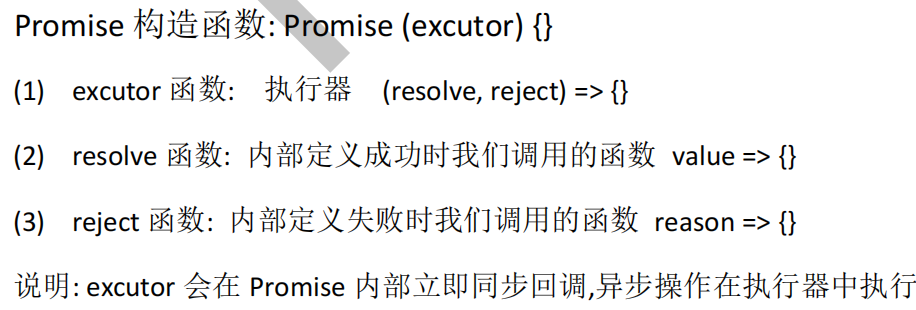
fulfilled: 意味着操作成功完成。

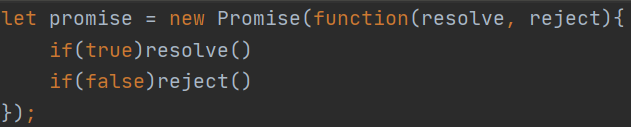
rejected: 意味着操作失败。

一旦状态改变就不会再变

可以先设置回调函数，后改变状态new promise{setTimeout}.then(value...,reject...)

#### （2）创建





如果使用了throw，也会变为reject状态，throw输入参数时，返回为reject的参数

#### （3）then

对promise的两种状态设置处理行为，并根据其所属状态执行。等待产生结果后，返回新的promise对象，结果没好时，将回调函数放入p的待执行数组中。前面的返回值为常数时，为onResolved状态，为promise时，为其最终状态。

其中输入参数对应之前的参数。onResolved的输入value对应之前resolved（value);onRejected的输入reason对应rejected(reason)



#### （4）catch

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。

处理上个promise的onRejected状态

#### （5）all

将多个Promise实例包装成一个新Promise实例，只有全部fulfilled才会成功。当所有promise为fullfilled时，返回一个promise，其：



#### （6）race

选择率先改变的Promise状态作为最终状态。



#### （7）finally

不管promise最后的状态，在执行完then或catch指定的回调函数以后，都会执行finally方法指定的回调函数。

#### （8）resolve

返回一个以给定值解析后的[Promise](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Promise) 对象。

当Promise.resolve(value)中value为promise时，返回该promise

#### （9）reject

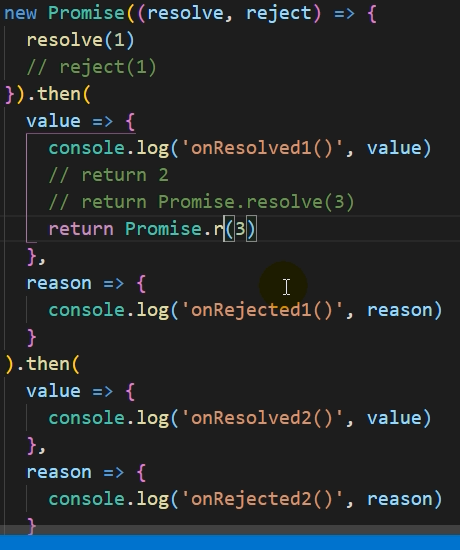
### 4.注意点

#### （1）Promise/then方法本身是同步代码，只不过其中的部分代码是异步执行。



Promise中正常的代码是同步执行，resolve、reject是异步执行，.then()中的方法都是异步执行。

#### （2）then链式调用



<1>如果之前的then没有返回值，那么下个then的value/reason就是undefined，除非定义return值

<2>then中不支持直接使用resolve，因为then方法中根本没有定义resolve方法。then是串联执行，不管内部异步与否。

<3>下一个then的结果，是根据上一个then的某个回调函数执行情况决定的（resolved、rejected都有可能，并不是resolved对应resolved）

<4>当没有定义then中对rejected的处理时，会默认构造：reason => {throw reason}，因此链式调用最后加.catch会返回调用链最前面的rejected。因此修改调用链中某个then的rejected可能会影响catch。

<5>如果在catch后增加then，then的返回值取决于catch的执行结果，catch中没有指定value，reason格式，因此默认是resolve结果，想返回reject可以直接返回Promise.reject

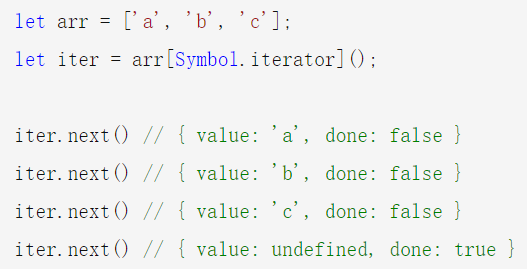
<6>new Promise(()=>{})会返回一个padding的promise

## Iterator和for...of（Iterator遍历器的实现）

### 1.Iterator遍历器

遍历器（Iterator）就是这样一种机制。它是一种接口，为各种不同的数据结构提供统一的访问机制。任何数据结构只要部署Iterator接口，就可以完成遍历操作（即依次处理该数据结构的所有成员）。

ES6规定，默认的Iterator接口部署在数据结构的Symbol.iterator属性，或者说，一个数据结构只要具有Symbol.iterator属性，就可以认为是“可遍历的”（iterable）。Symbol.iterator属性本身是一个函数，就是当前数据结构默认的遍历器生成函数。执行这个函数，就会返回一个遍历器。凡是部署了Symbol.iterator属性的数据结构，就称为部署了遍历器接口。调用这个接口，就会返回一个遍历器对象。



**调用Iterator接口的场合**

有一些场合会默认调用Iterator接口（即Symbol.iterator方法），除了下文会介绍的for...of循环，还有几个别的场合。

**（1）解构赋值**

对数组和Set结构进行解构赋值时，会默认调用Symbol.iterator方法。

**（2）扩展运算符**

扩展运算符（...）也会调用默认的iterator接口。

**（3）yield\***

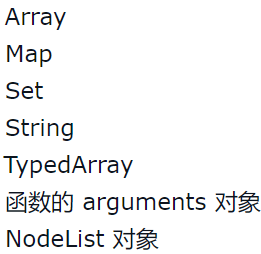
yield\*后面跟的是一个可遍历的结构，它会调用该结构的遍历器接口。

**（4）其他场合**

由于数组的遍历会调用遍历器接口，所以任何接受数组作为参数的场合，其实都调用了遍历器接口。下面是一些例子。

* for...of
* Array.from()
* Map(), Set(), WeakMap(), WeakSet()（比如new Map([['a',1],['b',2]])）
* Promise.all()
* Promise.race()

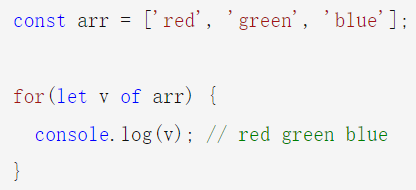
**（5）拥有接口的对象（可迭代对象）：**

****

### 2.for...of

一个数据结构只要部署了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。也就是说，for...of循环内部调用的是数据结构的Symbol.iterator方法。

for...of循环可以使用的范围包括数组、Set 和 Map 结构、某些类似数组的对象（比如arguments对象、DOM NodeList 对象）、后文的 Generator 对象，以及字符串。



**循环语法比较及使用场景（for、forEach、for...in、for...of）**

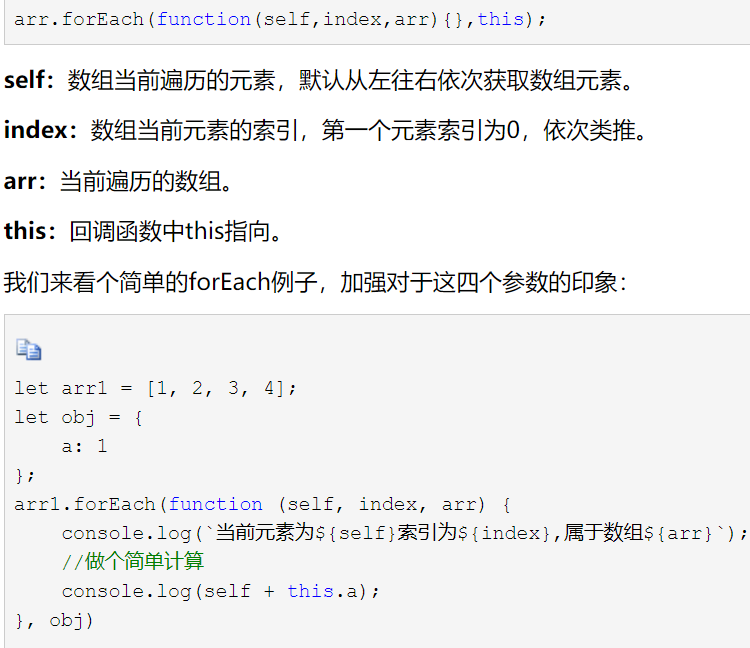
**1、for循环**

缺点是很麻烦。

**2、forEach**

回调函数参数分别是value,index,array

缺点在于无法中途跳出forEach循环，break和return命令都不行。



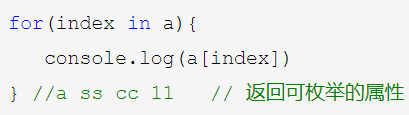
### 3.for…in

用于遍历所有的可枚举**属性**，功能类似于Object.keys()，但是遍历不到constructor、length这样的不可枚举属性。

缺点：

数组的键名为数字，但是for…in循环是以字符串作为键名"0",“1”,“2”

for…in循环主要是为遍历对象而设计的，不适用于遍历数组。



### 4、for…of

for…of是ES6新增的遍历方式，它提供了统一的遍历机制。所有实现了[Symbol.iterator]接口的对象都可以被遍历。for…of循环可以使用的范围包括数组、Set 和 Map 结构、某些类似数组的对象（比如arguments对象、DOM NodeList 对象）、Generator 对象，以及字符串

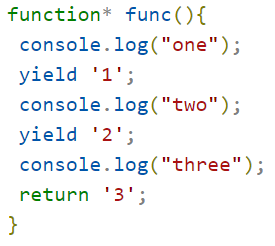
优点：1、用法简洁 2、可以和break,continue,return配合使用 3、提供了遍历所有数据结构的统一操作接口。

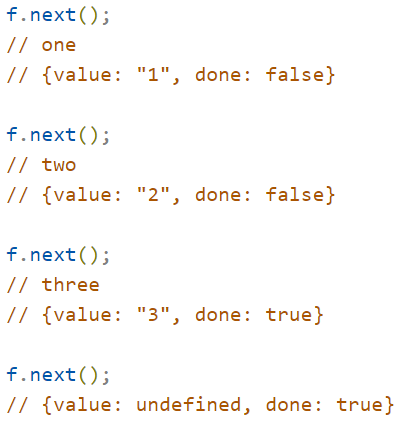
## Generator及其异步方面的应用

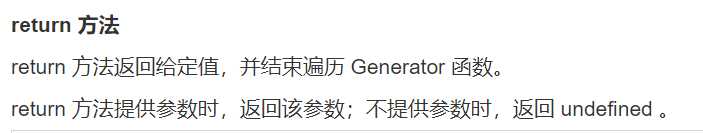
### 1.定义

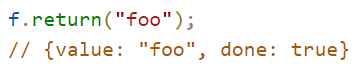
使用 function\* 语法和一个或多个 yield 表达式以创建一个函数即为生成器，当然它的返回值就是一个生成器

### 2.使用









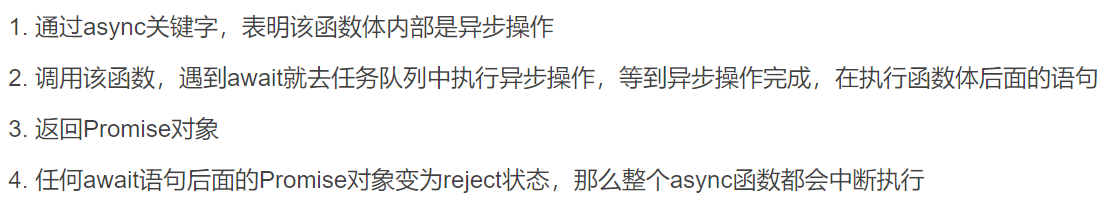
## async函数

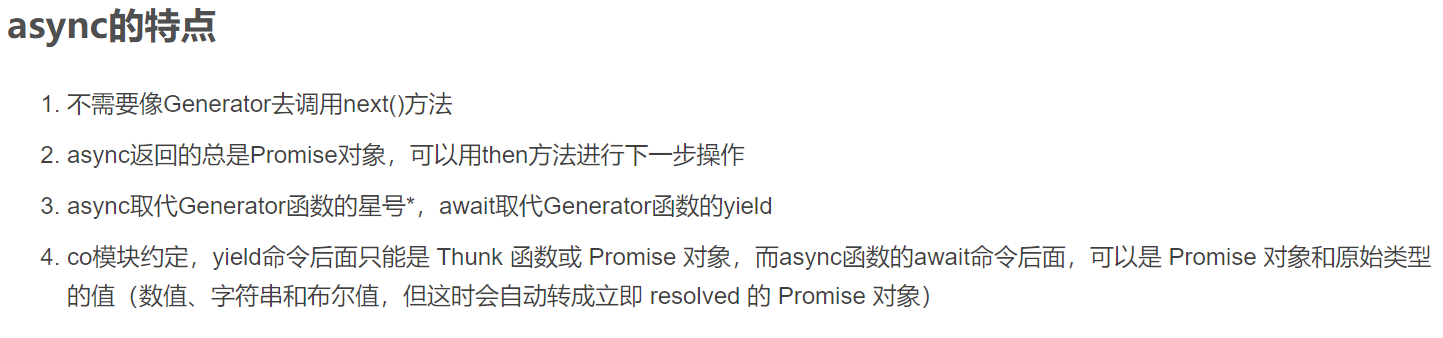
### 1.定义

async 是 ES7 才有的与异步操作有关的关键字，和 Promise ， Generator 有很大关联的。

### 2.使用

**步骤：**





### 3.await

操作符用于等待一个 Promise 对象, 它只能在异步函数 async function 内部使用。

返回 Promise 对象的处理结果。如果等待的不是 Promise 对象，则返回该值本身。

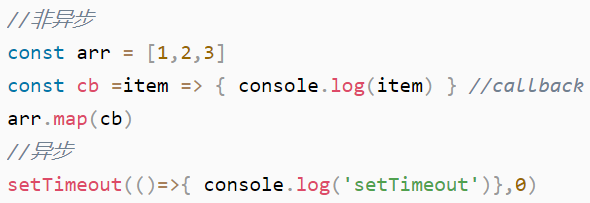
如果一个 Promise 被传递给一个 await 操作符，await 将等待 Promise 正常处理完成并返回其处理结果。

## 几种异步方式的比较（回调、setTimeout、Promise、Generator、async）

### 1.回调函数

回调函数：在函数中充当参数的函数

回调函数不一定都是异步的：



### 2.发布订阅模式

pub/sub模式是js设计模式中的一种，本身是借鉴于java的模式，但是在处理异步处理的时候非常有作用。通过一个信息中心`*EventCenter*` 来处理的监听(`*on*`)和触发(`*triggle*`)

### 3.promise

是异步编程的一种解决方案, Promise构造函数成为承诺，它分为三种状态`resolve`, `reject`, `pending` ,一旦状态pending改为其它2个状态之后，就不能修改了，就一个承诺一样。

Promise接收2个参数resolve , rejecj，分别表示成功后执行和失败后执行，可以通过实例的`then()`方法传递对于的函数。

### 4.generator

generator是一个构造器，`generator`函数执行并不会执行函数体内部部分，而是返回一个构造器对象，通过构造器对象的`next()`方法调用函数主体，并且每当遇到`yield`都会暂停执行，并返回一个对象。

### 5.async/await异步处理

ES7出现了`async/await`进行异步的处理，使得异步操作就像同步代码一样简单，方便了使用，由于async/await内部封装了generator的处理，所有就很少有人用generator来处理异步了，但是在异步的推动中`generator`起到了很大的作用。

await：后面接受一个promise实例

async: 返回一个promise对象

## class基本语法及继承

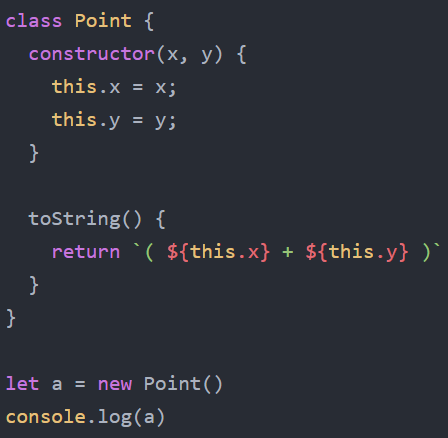
### 1.定义

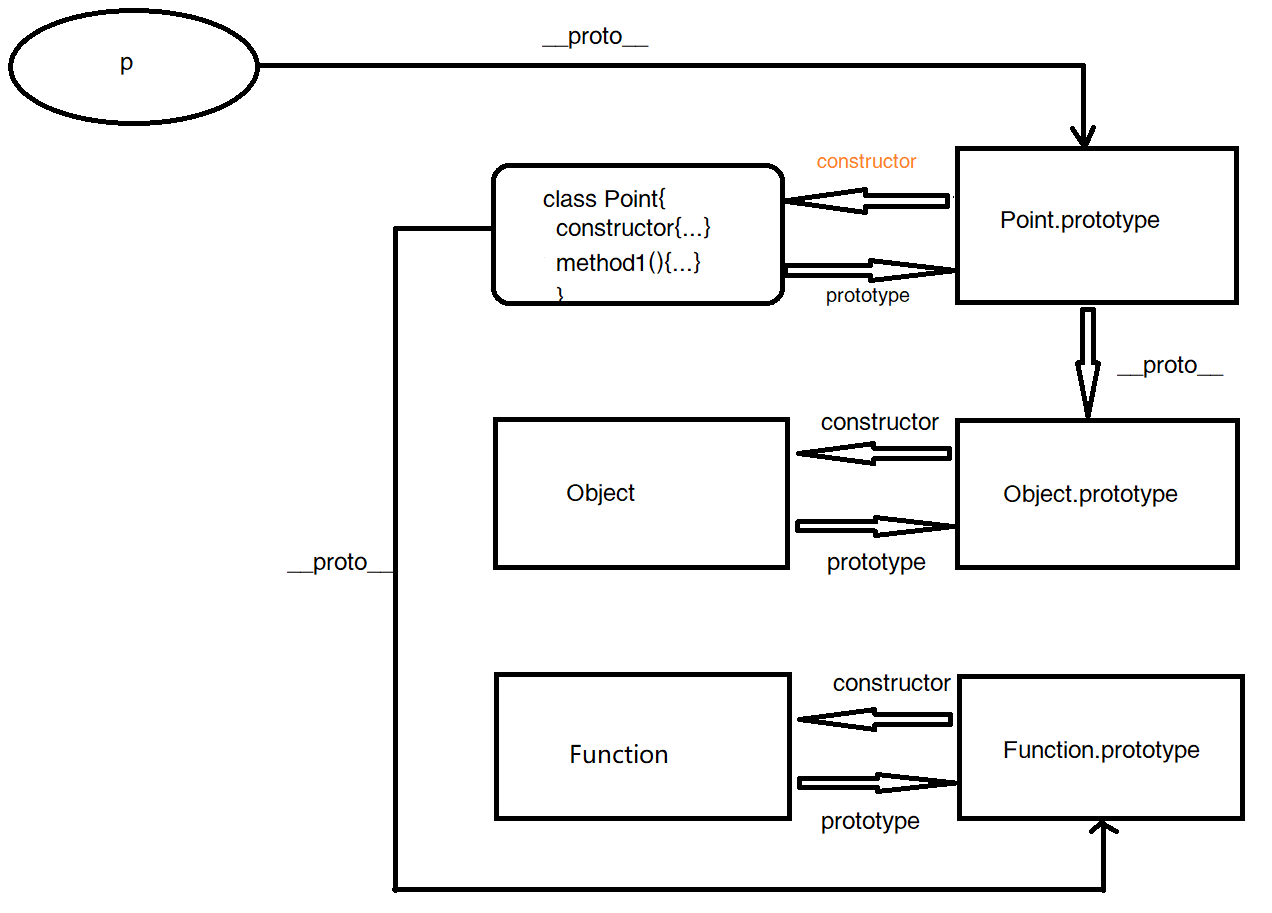
class是es6推出的语法糖，完全可以看作构造函数的另一种写法。

### 2.使用

1）构造方法constructor

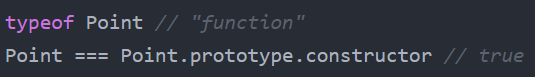
2）自定义方法（不必加function）





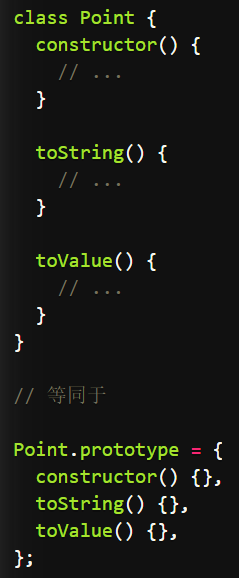
### 3.特点

1）类的数据类型是函数function，类本身就指向构造函数（就像Array.prototype指向Array构造函数，符合规范）。



2）使用new创建实例（必须加上new），同时调用class内constructor内容

3）方法放在类的原型上



4）默认遵循严格模式

5）class不存在遍历提升

### 4.方法

**（1）constructor 方法**

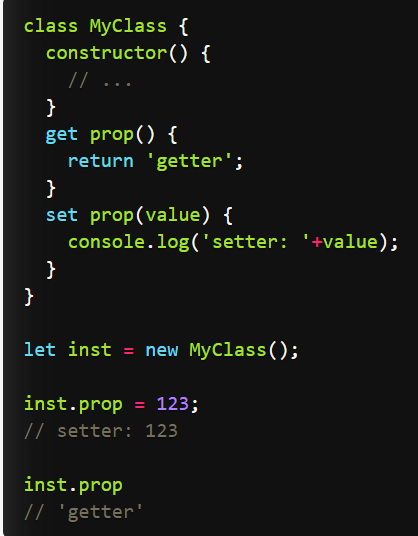
<1>constructor()方法是类的默认方法，通过new命令生成对象实例时，自动调用该方法。一个类必须有constructor()方法，如果没有显式定义，一个空的constructor()方法会被默认添加。

<2>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>类必须使用new调用，否则会报错。这是它跟普通构造函数的一个主要区别，后者不用new也可以执行。

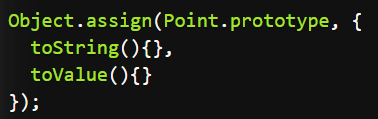
<4>属性在实例对象本身，方法在其原型上

**（2）取值函数（getter）和存值函数（setter）**



**（3）Object.assign()方法**

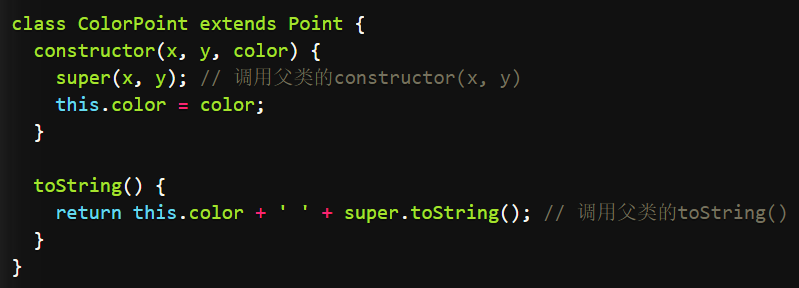
可以很方便地一次向类添加多个方法。



### 5.继承

**（1）方式**

Class 可以通过extends关键字实现继承，创建出的实例既是子类的实例，也是父类的实例



构造函数constructor中super为必需项。这是因为子类自己的this对象，必须先通过父类的构造函数完成塑造，得到与父类同样的实例属性和方法，然后再对其进行加工，加上子类自己的实例属性和方法。如果不调用super方法，子类就得不到this对象。

ES6 的继承机制实质是先将父类实例对象的属性和方法，加到this上面（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）Object.getPrototypeOf()**

可以用来从子类上获取父类。没有就返回空对象

**（3）Super**

作为函数时，super()只能用在子类的构造函数之中，用在其他地方就会报错。

super虽然代表了父类A的构造函数，但是返回的是子类B的实例，即super内部的this指的是B的实例，因此super()在这里相当于A.prototype.constructor.call(this)。

作为对象时，在普通方法中，指向父类的原型对象；在静态方法中，指向父类。

## 模块加载方案比较（CommonJS和ES6的Module）

 CommonJS模块输出的是一个值的拷贝，es6模块输出的是值的引用（commonjs的内部变化不会影响这个值）

 CommonJS模块是运行时加载（运行时才确定依赖关系），es6模块是编译时输出接口（编译时就确定依赖关系）

 CommonJS加载的是一个对象，该对象是运行时生成；es6模块不是对象，对外接口是代码静态解析就完成

 ES6 模块之中，顶层的this指向undefined；CommonJS 模块的顶层this指向当前模块

## ES6模块加载与CommonJS加载的原理

### CommonJS

CommonJS是nodejs也就是服务器端广泛使用的模块化机制。  
该规范的主要内容是，模块必须通过module.exports 导出对外的变量或接口，通过 require() 来导入其他模块的输出到当前模块作用域中。

根据这个规范，**每个文件就是一个模块**，有自己的作用域，文件中的变量、函数、类等都是对其他文件不可见的。

**对外暴露：**



**加载模块：**

****

**模块缓存：**

第一次加载某个模块时，Node会缓存该模块。以后再加载该模块，就直接从缓存取出该模块的module.exports属性。

**加载机制：**

CommonJS模块的加载机制是，输入的是被输出的值的拷贝。也就是说，一旦输出一个值，模块内部的变化就影响不到这个值。

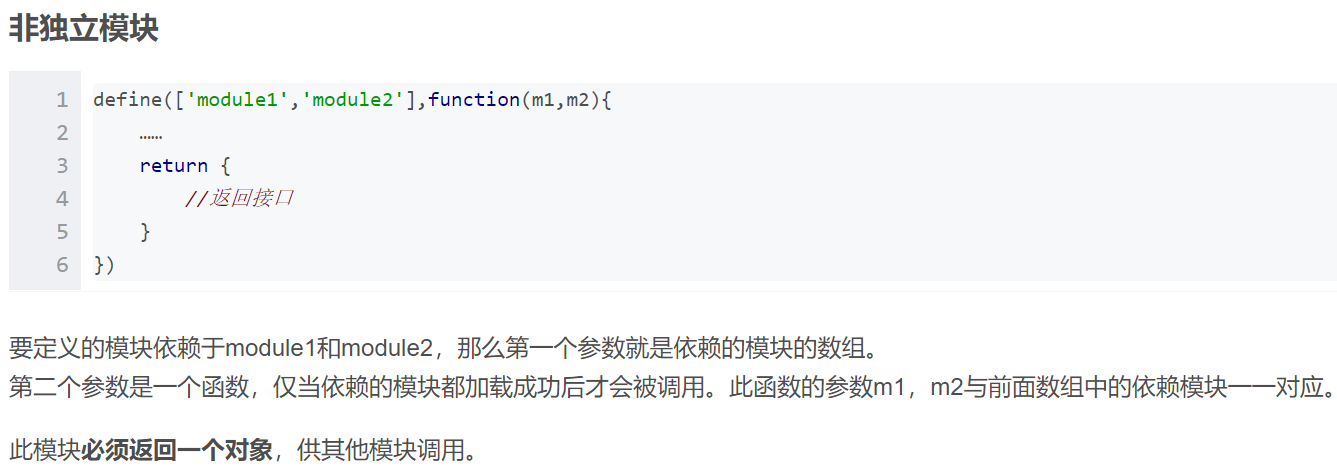
### AMD

AMD（异步模块定义）, requirejs即为遵循AMD规范的模块化工具。  
RequireJS的基本思想是，通过define方法，将代码定义为模块；通过require方法，实现代码的模块加载。

**定义模块**

define方法用于定义模块，RequireJS要求每个模块放在一个单独的文件里







可添加第三个参数，作为错误处理的回调函数

**CommonJS与AMD比较**

CommonJS规范加载模块是同步的，也就是说，只有加载完成，才能执行后面的操作。

AMD规范则是异步加载模块，允许指定回调函数，在回调函数中执行操作。

由于Node.js主要用于服务器编程，模块文件一般都已经存在于本地硬盘，所以加载起来比较快，不用考虑非同步加载的方式，所以CommonJS规范比较适用。但是，如果是浏览器环境，要从服务器端加载模块，这时就必须采用非同步模式，因此浏览器端一般采用AMD规范。

### ES6 Modules

ES6正式提出了内置的模块化语法

**特点：**

模块自动运行在严格模式下

在模块的顶级作用域创建的变量，不会被自动添加到共享的全局作用域，它们只会在模块顶级作用域的内部存在；

模块顶级作用域的 this 值为 undefined

对于需要让模块外部代码访问的内容，模块必须导出它们

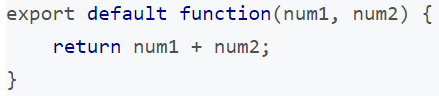
**定义模块：**

使用export关键字将任意变量、函数或者类公开给其他模块。

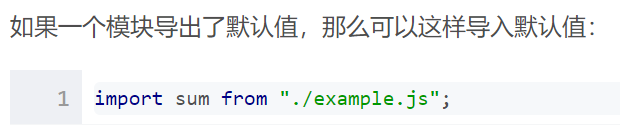


**导出默认值**

模块的默认值是使用 **default 关键字**所指定的单个变量、函数或类，而你在每个模块中**只能设置一个默认导出**

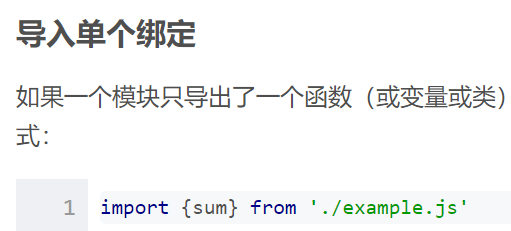


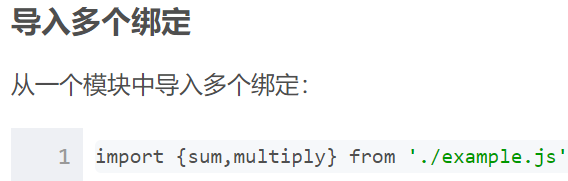
**导入默认值**

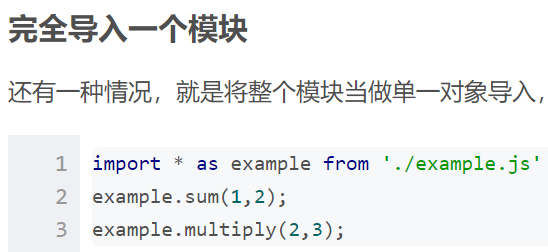


**加载模块**

在模块中使用**import关键字**来导入其他模块。  
import 语句有两个部分，一是需要导入的标识符，二是需导入的标识符的来源模块。此处是导入语句的基本形式：







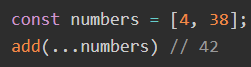
**限制**

export 与 import 都有一个重要的限制，那就是它们必须被用在其他语句或表达式的外部，而不能使用在if等代码块内部。原因之一是模块语法需要让 JS 能静态判断需要导出什么，正因为此，你只能在模块的顶级作用域使用 export与import。

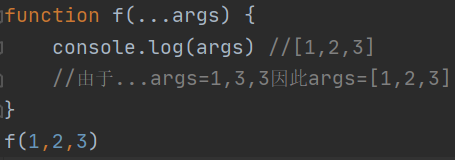
## 拓展运算符作用

将引用类型变为参数

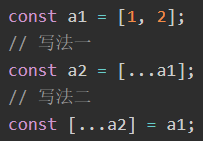
### （1）变参数



### （2）多个参数简写



### （3）复制数组、对象

（对象也可）

### （4）合并数组、对象



### （5）字符串转数组

