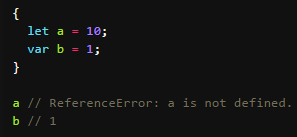
ECMAScript 6.0（以下简称ES6）是JavaScript语言的下一代标准，已经在2015年6月正式发布了。它的目标，是使得JavaScript语言可以用来编写复杂的大型应用程序，成为企业级开发语言。各大浏览器的最新版本，随着时间的推移，支持度已经越来越高了，ES6的大部分特性都实现了。那么也就意味着低版本浏览器是不支持ES6的。

## 1：let/const

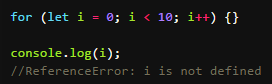
**let命令：**

ES6新增了let命令，用来声明变量。它的用法类似于var，但是所声明的变量，只在let命令所在的代码块内有效。

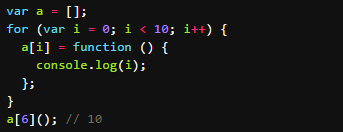


上面代码在代码块之中，分别用let和var声明了两个变量。然后在代码块之外调用这两个变量，结果let声明的变量报错，var声明的变量返回了正确的值。这表明，let声明的变量只在它所在的代码块有效。

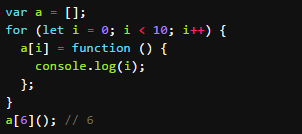
for循环的计数器，就很合适使用let命令：



上面代码中，计数器i只在for循环体内有效，在循环体外引用就会报错。下面的代码如果使用var，最后输出的是10。



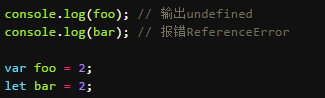
上面代码中，变量i是var声明的，在全局范围内都有效。所以每一次循环，新的i值都会覆盖旧值，导致最后输出的是最后一轮的i的值。如果使用let，声明的变量仅在块级作用域内有效，最后输出的是6。



上面代码中，变量i是let声明的，当前的i只在本轮循环有效，所以每一次循环的i其实都是一个新的变量，所以最后输出的是6。

**不存在变量提升。**

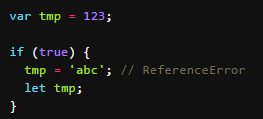
let不像var那样会发生“变量提升”现象。所以，变量一定要在声明后使用，否则报错。



上面代码中，变量foo用var命令声明，会发生变量提升，即脚本开始运行时，变量foo已经存在了，但是没有值，所以会输出undefined。变量bar用let命令声明，不会发生变量提升。这表示在声明它之前，变量bar是不存在的，这时如果用到它，就会抛出一个错误。

**暂时性死区。**

只要块级作用域内存在let命令，它所声明的变量就“绑定”（binding）这个区域，不再受外部的影响。



上面代码中，存在全局变量tmp，但是块级作用域内let又声明了一个局部变量tmp，导致后者绑定这个块级作用域，所以在let声明变量前，对tmp赋值会报错。

ES6明确规定，如果区块中存在let和const命令，这个区块对这些命令声明的变量，从一开始就形成了封闭作用域。凡是在声明之前就使用这些变量，就会报错。

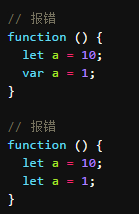
总之，在代码块内，使用let命令声明变量之前，该变量都是不可用的。这在语法上，称为“暂时性死区”（temporal dead zone，简称TDZ）。

ES6规定暂时性死区和let、const语句不出现变量提升，主要是为了减少运行时错误，防止在变量声明前就使用这个变量，从而导致意料之外的行为。这样的错误在ES5是很常见的，现在有了这种规定，避免此类错误就很容易了。

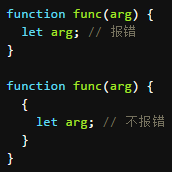
总之，暂时性死区的本质就是，只要一进入当前作用域，所要使用的变量就已经存在了，但是不可获取，只有等到声明变量的那一行代码出现，才可以获取和使用该变量。

**不允许重复声明。**

let不允许在相同作用域内，重复声明同一个变量。

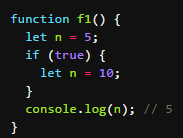


因此，不能在函数内部重复声明参数。



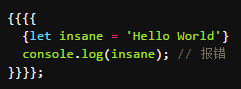
**块级作用域：**

ES5只有全局作用域和函数作用域，let的出现实际上为JavaScript新增了块级作用域。



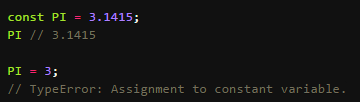
上面的函数有两个代码块，都声明了变量n，运行后输出5。这表示外层代码块不受内层代码块的影响。如果使用var定义变量n，最后输出的值就是10。

ES6允许块级作用域的任意嵌套，外层作用域无法读取内层作用域的变量。



**const命令：**

const声明一个只读的常量。一旦声明，常量的值就不能改变。

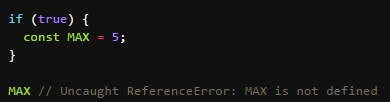


上面代码表明改变常量的值会报错。

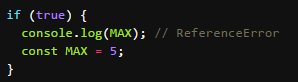
const声明的变量不得改变值，这意味着，const一旦声明变量，就必须立即初始化，不能留到以后赋值。



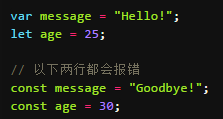
上面代码表示，对于const来说，只声明不赋值，就会报错。const的作用域与let命令相同：只在声明所在的块级作用域内有效。



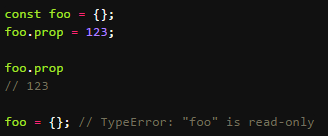
const命令声明的常量也是不提升，同样存在暂时性死区，只能在声明的位置后面使用。



上面代码在常量MAX声明之前就调用，结果报错。const声明的常量，也与let一样不可重复声明。



对于**复合类型**（？？）的变量，变量名不指向数据，而是指向数据所在的地址。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，所以将一个对象声明为常量必须非常小心。



上面代码中，常量foo储存的是一个地址，这个地址指向一个对象。不可变的只是这个地址，即不能把foo指向另一个地址，但对象本身是可变的，所以依然可以为其添加新属性。

**顶层对象属性：**

顶层对象，在浏览器环境指的是window对象，在Node指的是global对象。ES5之中，顶层对象的属性与全局变量是等价的。



上面代码中，顶层对象的属性赋值与全局变量的赋值，是同一件事。ES6为了改变这一点，一方面规定，为了保持兼容性，var命令和function命令声明的全局变量，依旧是顶层对象的属性；另一方面规定，let命令、const命令、class命令声明的全局变量，不属于顶层对象的属性。也就是说，从ES6开始，全局变量将逐步与顶层对象的属性脱钩。



上面代码中，全局变量a由var命令声明，所以它是顶层对象的属性；全局变量b由let命令声明，所以它不是顶层对象的属性，返回undefined。

## 2：变量解构赋值

ES6允许按照一定模式，从数组和对象中提取值，对变量进行赋值，这被称为解构（Destructuring）。

以前，为变量赋值，只能直接指定值。

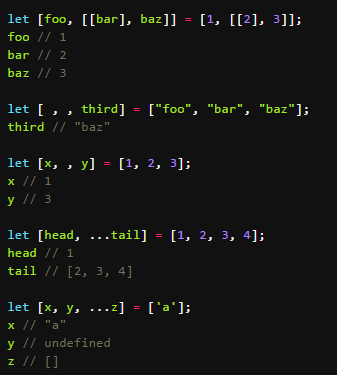


ES6允许写成下面这样。



上面代码表示，可以从数组中提取值，按照对应位置，对变量赋值。

本质上，这种写法属于“模式匹配”，只要等号两边的模式相同，左边的变量就会被赋予对应的值。下面是一些使用嵌套数组进行解构的例子。

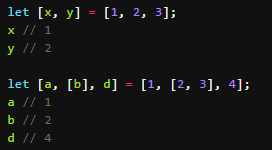


如果解构不成功，变量的值就等于undefined。



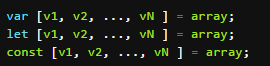
以上两种情况都属于解构不成功，foo的值都会等于undefined。

另一种情况是不完全解构，即等号左边的模式，只匹配一部分的等号右边的数组。这种情况下，解构依然可以成功。



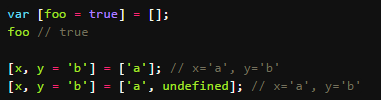
上面两个例子，都属于不完全解构，但是可以成功。

解构赋值不仅适用于var命令，也适用于let和const命令。

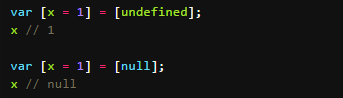


**默认值：**

解构赋值允许指定默认值。



注意，ES6内部使用严格相等运算符（===），判断一个位置是否有值。所以，如果一个数组成员不严格等于undefined，默认值是不会生效的。（？？）



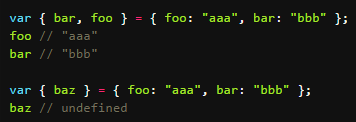
上面代码中，如果一个数组成员是null，默认值就不会生效，因为null不严格等于undefined。

**对象的解构赋值：**

解构不仅可以用于数组，还可以用于对象。

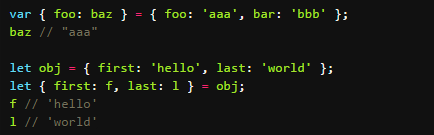


对象的解构与数组有一个重要的不同。数组的元素是按次序排列的，变量的取值由它的位置决定；而对象的属性没有次序，变量必须与属性同名，才能取到正确的值。



上面代码的第一个例子，等号左边的两个变量的次序，与等号右边两个同名属性的次序不一致，但是对取值完全没有影响。第二个例子的变量没有对应的同名属性，导致取不到值，最后等于undefined。

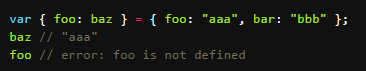
如果变量名与属性名不一致，必须写成下面这样。



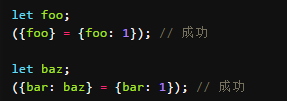
这实际上说明，对象的解构赋值是下面形式的简写。



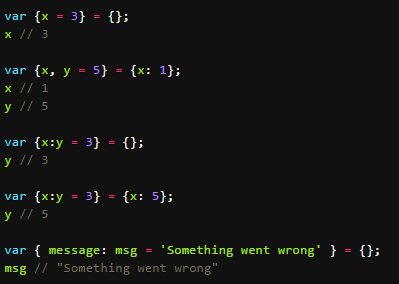
也就是说，对象的解构赋值的内部机制，是先找到同名属性，然后再赋给对应的变量。真正被赋值的是后者，而不是前者。



下面代码中，let命令下面一行的圆括号是必须的，否则会报错。因为解析器会将起首的大括号，理解成一个代码块，而不是赋值语句。

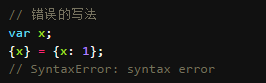


对象的解构也可以指定默认值。



默认值生效的条件是，对象的属性值严格等于undefined。

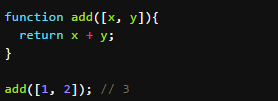
如果要将一个已经声明的变量用于解构赋值，必须非常小心。



上面代码的写法会报错，因为JavaScript引擎会将{x}理解成一个代码块，从而发生语法错误。只有不将大括号写在行首，避免JavaScript将其解释为代码块，才能解决这个问题。

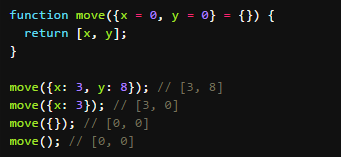


**函数参数解构赋值：**



上面代码中，函数add的参数表面上是一个数组，但在传入参数的那一刻，数组参数就被解构成变量x和y。对于函数内部的代码来说，它们能感受到的参数就是x和y。

**函数参数的解构也可以使用默认值。(重点理解)**



上面代码中，函数move的参数是一个对象，通过对这个对象进行解构，得到变量x和y的值。如果解构失败，x和y等于默认值。

**用途：**

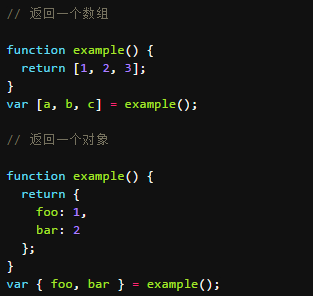
1：交换变量的值。



上面代码交换变量x和y的值，这样的写法不仅简洁，而且易读，语义非常清晰。

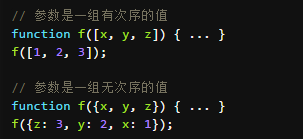
2：从函数返回多个值。

函数只能返回一个值，如果要返回多个值，只能将它们放在数组或对象里返回。有了解构赋值，取出这些值就非常方便。



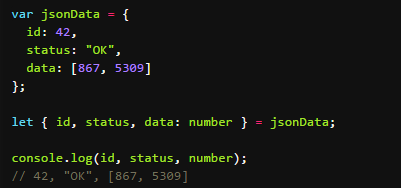
3：函数参数的定义。

解构赋值可以方便地将一组参数与变量名对应起来。



4：提取JSON数据。

解构赋值对提取JSON对象中的数据，尤其有用。



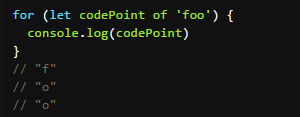
5：函数参数的默认值（？？？）。

指定参数的默认值，就避免了在函数体内部再写var foo = config.foo || 'default foo';这样的语句。

## 3：字符串扩展

**字符串的遍历器接口：**

ES6为字符串添加了遍历器接口，使得字符串可以被for...of循环遍历。与for…in的区别？



**includes()、startsWith()、endsWith()：**

传统上，JavaScript只有indexOf方法，可以用来确定一个字符串是否包含在另一个字符串中。ES6又提供了三种新方法。

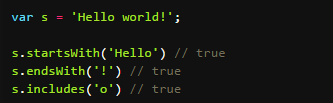
includes()：返回布尔值，表示是否找到了参数字符串。

startsWith()：返回布尔值，表示参数字符串是否在源字符串的头部。

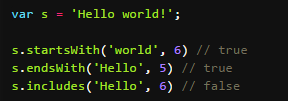
endsWith()：返回布尔值，表示参数字符串是否在源字符串的尾部。



10表示个数，前10个里面的末尾是否为dgf



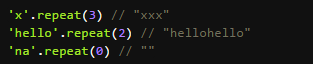
这三个方法都支持第二个参数，表示开始搜索的位置。



上面代码表示，使用第二个参数n时，endsWith的行为与其他两个方法有所不同。它针对前n个字符，而其他两个方法针对从第n个位置直到字符串结束。(不是下标,是个数)

**repeat(**n**)：**

repeat方法返回一个新字符串，表示将原字符串重复n次。



参数如果是小数，会被向下取整。



参数NaN等同于0。

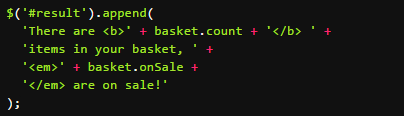


如果repeat的参数是字符串，则会先转换成数字。

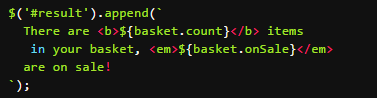


**模板字符串：**

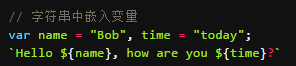
传统的JavaScript语言，输出模板通常是这样写的。



上面这种写法相当繁琐不方便，ES6引入了模板字符串解决这个问题。



模板字符串（template string）是增强版的字符串，用反引号（`）标识。它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。模板字符串中嵌入变量，需要将变量名写在${}之中。



上面代码中的模板字符串，都是用反引号表示。如果在模板字符串中需要使用反引号，则前面要用反斜杠转义。



如果使用模板字符串表示多行字符串，所有的空格和缩进都会被保留在输出之中。



## 应用：

1：掌握ES6上午学习的内容。

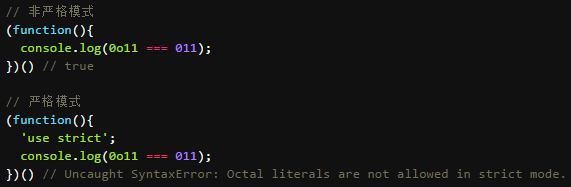
## 4：数值扩展

**二进制和八进制的表示法：**

ES6提供了二进制和八进制数值的新的写法，分别用前缀0b（或0B）和0o（或0O）表示。



从ES5开始，在严格模式之中，八进制就不再允许使用前缀0表示，ES6进一步明确，要使用前缀0o表示。

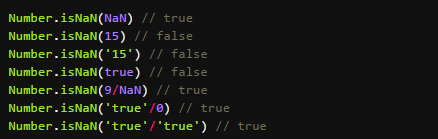


如果要将0b和0o前缀的字符串数值转为十进制，要使用Number方法。

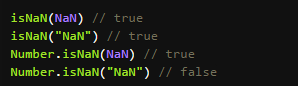


**Number.isNaN()、Number.parseInt()、Number.parseFloat()：**

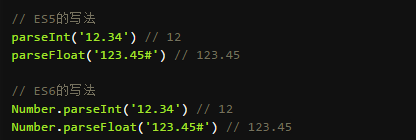
ES6在Number对象上，新提供了Number.isNaN()方法。Number.isNaN()用来检查一个值是否为NaN。



它们与传统的全局方法isNaN()的**区别**在于，传统方法先调用Number()将非数值的值转为数值，再进行判断，而**该新方法只对数值有效，非数值一律返回false。**



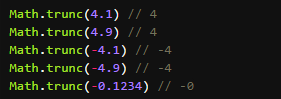
ES6将全局方法parseInt()和parseFloat()，移植到Number对象上面，行为完全保持不变。



这样做的目的，是逐步减少全局性方法，使得语言逐步模块化。

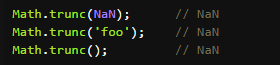
**Math对象的扩展：**

**Math.trunc()**：用于去除一个数的小数部分，返回整数部分。



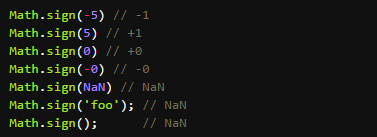
对于非数值，Math.trunc内部使用Number方法将其先转为数值。对于空值和无法截取整数的值，返回NaN。



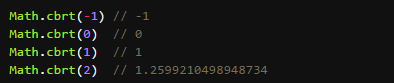


**Math.sign()**：用来判断一个数到底是正数、负数、还是零。

它会返回五种值：参数为正数，返回+1；参数为负数，返回-1；参数为0，返回0；参数为-0，返回-0;其他值，返回NaN。



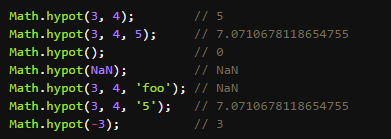
**Math.cbrt()**：用于计算一个数的立方根。



对于非数值，Math.cbrt方法内部也是先使用Number方法将其转为数值。



**Math.hypot()**：返回所有参数的平方和的平方根。

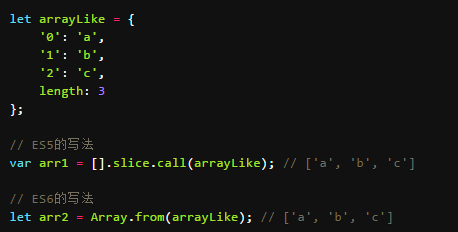


如果参数不是数值，Math.hypot方法会将其转为数值。只要有一个参数无法转为数值，就会返回NaN。

## 5：数组扩展

**Array.from()：**用于将两类对象转为真正的数组：类似数组的对象（array-like object）和可遍历（iterable）的对象。

下面是一个类似数组的对象，Array.from将它转为真正的数组。

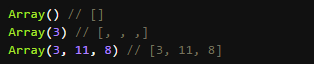


实际应用中，常见的类似数组的对象是DOM操作返回的NodeList集合，以及函数内部的arguments对象。Array.from都可以将它们转为真正的数组。

**Array.of()：**用于将一组值，转换为数组。

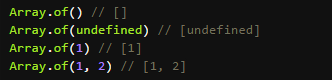


这个方法的主要目的，是弥补数组构造函数Array()的不足。因为参数个数的不同，会导致Array()的行为有差异。



上面代码中，Array方法没有参数、一个参数、三个参数时，返回结果都不一样。只有当参数个数不少于2个时，Array()才会返回由参数组成的新数组。参数个数只有一个时，实际上是指定数组的长度。

Array.of基本上可以用来替代Array()或new Array()，并且不存在由于参数不同而导致的重载。它的行为非常统一。



Array.of总是返回参数值组成的数组。如果没有参数，就返回一个空数组。

**数组实例的copyWithin()（？？？）**：在当前数组内部，将指定位置的成员复制到其他位置（会覆盖原有成员），然后返回当前数组。也就是说，使用这个方法，会修改当前数组。

它接受三个参数：target（必需）：从该位置开始替换数据。start（可选）：从该位置开始读取数据，默认为0。如果为负值，表示倒数。end（可选）：到该位置前停止读取数据，默认等于数组长度。如果为负值，表示倒数。

这三个参数都应该是数值，如果不是，会自动转为数值。



上面代码表示将从3号位直到数组结束的成员（4和5），复制到从0号位开始的位置，结果覆盖了原来的1和2。

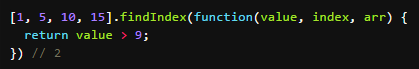
**数组实例的find()和findIndex()：**

数组实例的find方法，用于找出第一个符合条件的数组成员。它的参数是一个回调函数，所有数组成员依次执行该回调函数，直到找出第一个返回值为true的成员，然后返回该成员。如果没有符合条件的成员，则返回undefined。



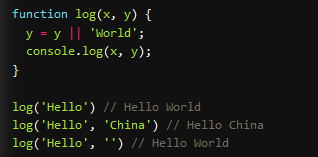
上面代码中，find方法的回调函数可以接受三个参数，依次为当前的值、当前的位置和原数组。

数组实例的findIndex方法的用法与find方法非常类似，返回第一个符合条件的数组成员的位置（索引值），如果所有成员都不符合条件，则返回-1。



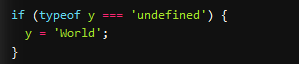
## 6：函数扩展

**函数参数默认值：**在ES6之前，不能直接为函数的参数指定默认值，只能采用变通的方法。

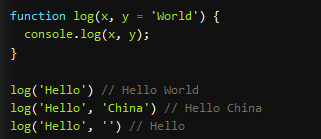


上面代码检查函数log的参数y有没有赋值，如果没有，则指定默认值为World。这种写法的缺点在于，如果参数y赋值了，但是对应的布尔值为false，则该赋值不起作用。就像上面代码的最后一行，参数y等于空字符，结果被改为默认值。

为了避免这个问题，通常需要先判断一下参数y是否被赋值，如果没有，再等于默认值。

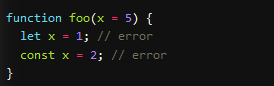


ES6允许为函数的参数设置默认值，即直接写在参数定义的后面。



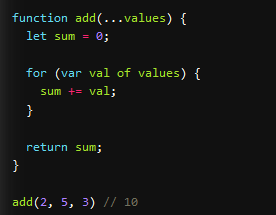
可以看到，ES6的写法比ES5简洁许多，而且非常自然。除了简洁，ES6的写法还有两个好处：首先，阅读代码的人，可以立刻意识到哪些参数是可以省略的，不用查看函数体或文档；其次，有利于将来的代码优化，即使未来的版本在对外接口中，彻底拿掉这个参数，也不会导致以前的代码无法运行。

参数变量是默认声明的，所以不能用let或const再次声明。



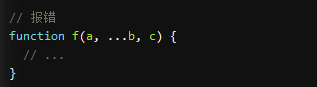
上面代码中，参数变量x是默认声明的，在函数体中，不能用let或const再次声明，否则会报错。

**rest参数：**ES6引入rest参数（形式为“...变量名”），用于获取函数的多余参数，这样就不需要使用arguments对象了。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。

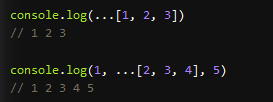


上面代码的add函数是一个求和函数，利用rest参数，可以向该函数传入任意数目的参数。

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。

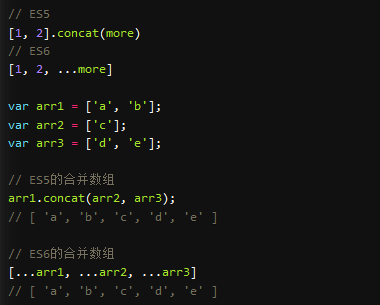


**扩展运算符：**扩展运算符（spread）是三个点（...）。它好比rest参数的逆运算，将一个数组转为用逗号分隔的参数序列。

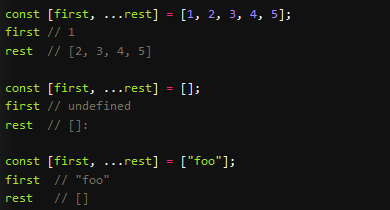


扩展运算符的应用：

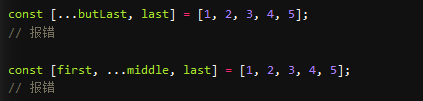
1：合并数组：扩展运算符提供了数组合并的新写法。



2：与解构赋值结合：扩展运算符可以与解构赋值结合起来，用于生成数组。



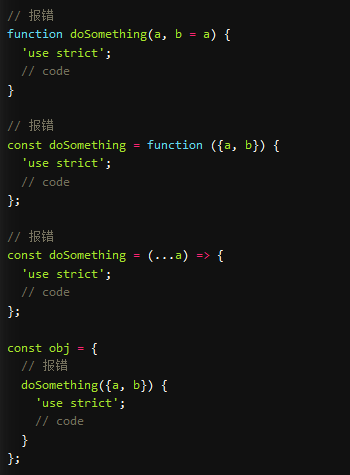
如果将扩展运算符用于数组赋值，只能放在参数的最后一位，否则会报错。



3：字符串：扩展运算符还可以将字符串转为真正的数组。



**严格模式：**从ES5开始，函数内部可以设定为严格模式。《ECMAScript 2016标准》做了一点修改，规定只要函数参数使用了默认值、解构赋值、或者扩展运算符，那么函数内部就不能显式设定为严格模式，否则会报错。



这样规定的原因是，函数内部的严格模式，同时适用于函数体代码和函数参数代码。但是，函数执行的时候，先执行函数参数代码，然后再执行函数体代码。这样就有一个不合理的地方，只有从函数体代码之中，才能知道参数代码是否应该以严格模式执行，但是参数代码却应该先于函数体代码执行。

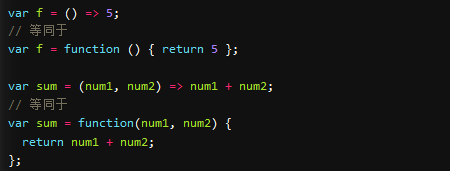
**箭头函数：**ES6允许使用“箭头”（=>）定义函数。



上面的箭头函数等同于：



如果箭头函数不需要参数或需要多个参数，就使用一个圆括号代表参数部分。



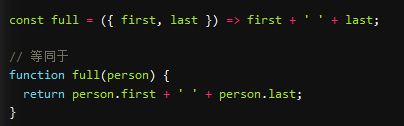
如果箭头函数的代码块部分多于一条语句，就要使用大括号将它们括起来，并且使用return语句返回。



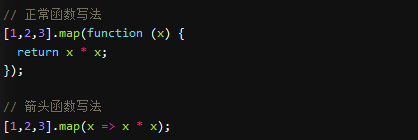
由于大括号被解释为代码块，所以如果箭头函数直接返回一个对象，必须在对象外面加上括号。



箭头函数可以与变量解构结合使用。



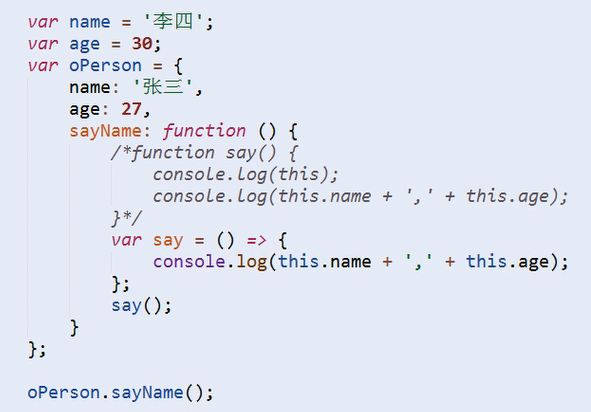
箭头函数的一个用处是简化回调函数，使得表达更加简洁。



使用注意点：

1：函数体内的this对象，就是定义时所在的对象，而不是使用时所在的对象。？

试一试》》》对象方法内的调用this指向改方法，其他的正常调用都指向Window



2：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。

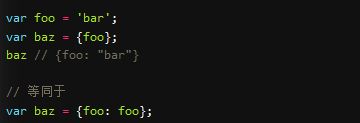
3：不可以使用arguments对象，该对象在函数体内不存在。如果要用，可以用Rest参数代替。

4：不可以使用yield命令，因此箭头函数不能用作Generator函数。

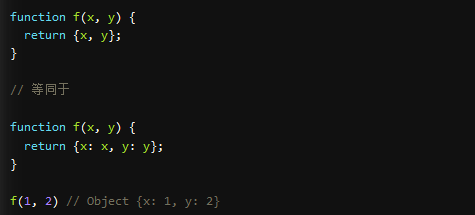
上面四点中，第一点尤其值得注意。this对象的指向是可变的，但是在箭头函数中，它是固定的。

## 7：对象扩展

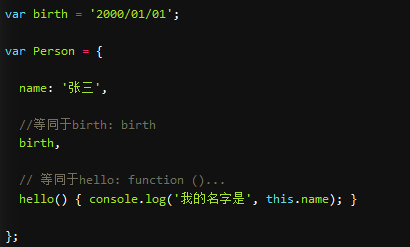
**属性的简洁表示法：**ES6允许直接写入变量和函数，作为对象的属性和方法。这样的书写更加简洁。



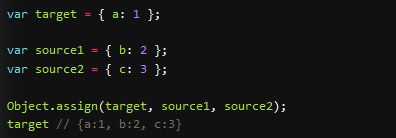
上面代码表明，ES6允许在对象之中，直接写变量。这时，属性名为变量名, 属性值为变量的值。下面是另一个例子。



除了属性简写，方法也可以简写。

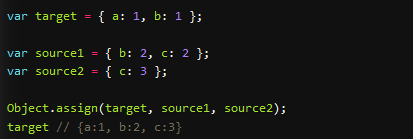


**Object.assign()：**用于对象的合并，将源对象（source）的所有属性，复制到目标对象（target）。

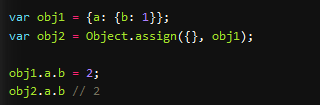


Object.assign方法的第一个参数是目标对象，后面的参数都是源对象。

注意，如果目标对象与源对象有同名属性，或多个源对象有同名属性，则后面的属性会覆盖前面的属性。



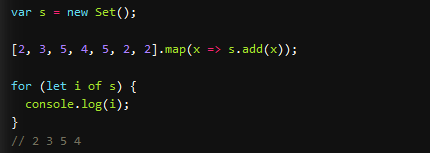
注意：Object.assign方法实行的是浅拷贝，而不是深拷贝。也就是说，如果源对象某个属性的值是对象，那么目标对象拷贝得到的是这个对象的引用。



上面代码中，源对象obj1的a属性的值是一个对象，Object.assign拷贝得到的是这个对象的引用。这个对象的任何变化，都会反映到目标对象上面。

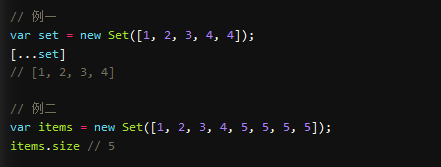
## 8：Set和Map结构（？？）

**Set：**ES6提供了新的数据结构Set。它类似于数组，但是成员的值都是唯一的，没有重复的值。Set本身是一个构造函数，用来生成Set数据结构。



上面代码通过add方法向Set结构加入成员，结果表明Set结构不会添加重复的值。

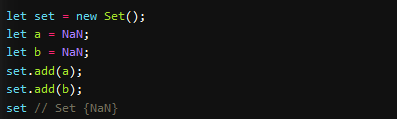
Set函数可以接受一个数组（或类似数组的对象）作为参数，用来初始化。



上面代码中，也展示了一种去除数组重复成员的方法。



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。Set内部判断两个值是否不同，使用的算法叫做“Same-value equality”，它类似于精确相等运算符（===），主要的区别是NaN等于自身，而精确相等运算符认为NaN不等于自身。



上面代码向Set实例添加了两个NaN，但是只能加入一个。这表明，在Set内部，两个NaN是相等。

**Set实例的属性和方法：**

Set结构的实例有以下属性。

constructor属性：构造函数，默认就是Set函数。

size属性：返回Set实例的成员总数。

Set实例的方法分为两大类：操作方法（用于操作数据）和遍历方法（用于遍历成员）。下面先介绍四个操作方法。

add(value)：添加某个值，返回Set结构本身。

delete(value)：删除某个值，返回一个布尔值，表示删除是否成功。

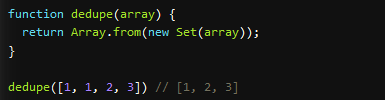
has(value)：返回一个布尔值，表示该值是否为Set的成员。

clear()：清除所有成员，没有返回值。

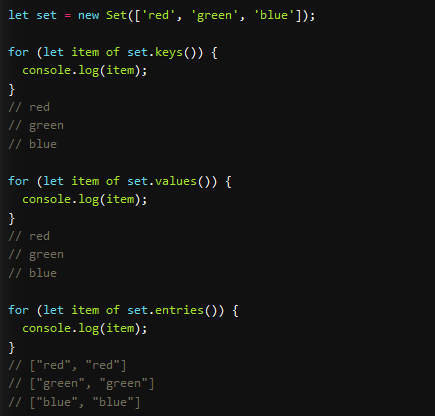
Array.from方法可以将Set结构转为数组。



这就提供了去除数组重复成员的另一种方法。

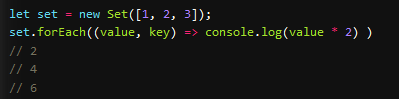


keys()，values()，entries()：keys方法、values方法、entries方法返回的都是遍历器对象。由于 Set 结构没有键名，只有键值（或者说键名和键值是同一个值），所以keys方法和values方法的行为完全一致。



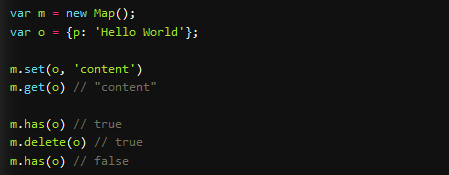
上面代码中，entries方法返回的遍历器，同时包括键名和键值，所以每次输出一个数组，它的两个成员完全相等。

forEach()：使用回调函数遍历每个成员。Set结构的实例的forEach方法，用于对每个成员执行某种操作，没有返回值。



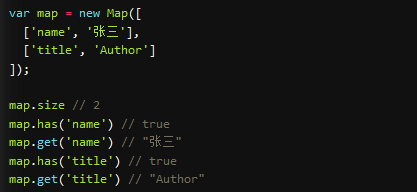
上面代码说明，forEach方法的参数就是一个处理函数。该函数的参数依次为键值、键名、集合本身（上例省略了该参数）。

**Map：**JavaScript的对象（Object），本质上是键值对的集合，但是传统上只能用字符串当作键。这给它的使用带来了很大的限制。为了解决这个问题，ES6提供了Map数据结构。它类似于对象，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。也就是说，Object结构提供了“字符串—值”的对应，Map结构提供了“值—值”的对应。如果你需要“键值对”的数据结构，Map比Object更合适。



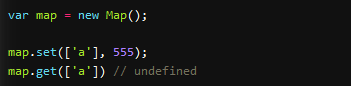
上面代码使用set方法，将对象o当作m的一个键，然后又使用get方法读取这个键，接着使用delete方法删除了这个键。

作为构造函数，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。



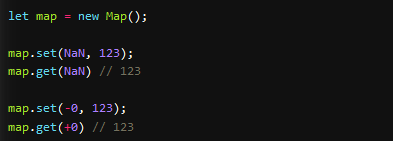
上面代码在新建Map实例时，就指定了两个键name和title。

注意，只有对同一个对象的引用，Map结构才将其视为同一个键。这一点要非常小心。



上面代码的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。

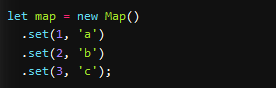
如果Map的键是一个简单类型的值（数字、字符串、布尔值），则只要两个值严格相等，Map将其视为一个键，包括0和-0。另外，虽然NaN不严格相等于自身，但Map将其视为同一个键。



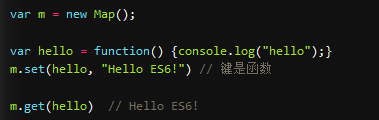
**Map实例的属性和方法：**

size属性：返回Map结构的成员总数。

set(key, value)：set方法设置key所对应的键值，然后返回整个Map结构。如果key已经有值，则键值会被更新，否则就新生成该键。set方法返回的是Map本身，因此可以采用链式写法。



get(key)：get方法读取key对应的键值，如果找不到key，返回undefined。



has(key)：返回一个布尔值，表示某个键是否在Map数据结构中。

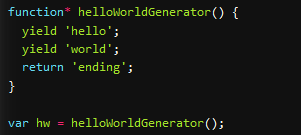
delete(key)：删除某个键，返回true。如果删除失败，返回false。

clear()：清除所有成员，没有返回值。

## 9：Generator生成器函数

Generator函数是ES6提供的一种异步编程解决方案，语法行为与传统函数完全不同。从语法上，可以把它理解成，Generator函数是一个状态机，封装了多个内部状态。形式上，Generator函数是一个普通函数，但是有两个特征。一是，function关键字与函数名之间有一个星号；二是，函数体内部使用yield语句，定义不同的内部状态（yield语句在英语里的意思就是“产出”）。

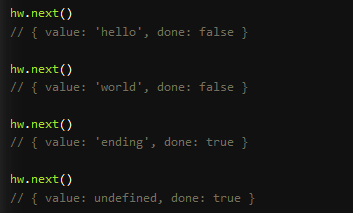
执行Generator函数会返回一个遍历器对象，也就是说，Generator函数除了状态机，还是一个遍历器对象生成函数。返回的遍历器对象，可以依次遍历Generator函数内部的每一个状态。



上面代码定义了一个Generator函数helloWorldGenerator，它内部有两个yield语句“hello”和“world”，即该函数有三个状态：hello，world和return语句（结束执行）。

然后，Generator函数的调用方法与普通函数一样，也是在函数名后面加上一对圆括号。不同的是，调用Generator函数后，该函数并不执行，返回的也不是函数运行结果，而是一个指向内部状态的指针对象，也就是刚才介绍的遍历器对象（Iterator Object）。

下一步，必须调用遍历器对象的next方法，使得指针移向下一个状态。也就是说，每次调用next方法，内部指针就从函数头部或上一次停下来的地方开始执行，直到遇到下一个yield语句（或return语句）为止。换言之，Generator函数是分段执行的，yield语句是暂停执行的标记，而next方法可以恢复执行。



**yield语句：**由于Generator函数返回的遍历器对象，只有调用next方法才会遍历下一个内部状态，所以其实提供了一种可以暂停执行的函数。yield语句就是暂停标志。

遍历器对象的next方法的运行逻辑如下：

1：遇到yield语句，就暂停执行后面的操作，并将紧跟在yield后面的那个表达式的值，作为返回的对象的value属性值。

2：下一次调用next方法时，再继续往下执行，直到遇到下一个yield语句。

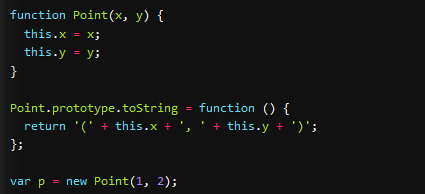
3：如果没有再遇到新的yield语句，就一直运行到函数结束，直到return语句为止，并将return语句后面的表达式的值，作为返回的对象的value属性值。

4：如果该函数没有return语句，则返回的对象的value属性值为undefined。

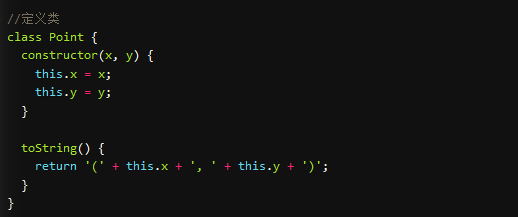
**next()方法**：会执行generator的代码，然后，每次遇到yield x;就返回一个对象{value: x, done: true/false}，然后“暂停”。返回的value就是yield的返回值，done表示这个generator是否已经执行结束了。如果done为true，则value就是return的返回值。当执行到done为true时，这个generator对象就已经全部执行完毕，不要再继续调用next()了。

## 10：class的写法

JavaScript语言的传统方法是通过构造函数，定义并生成新对象。下面是一个例子。



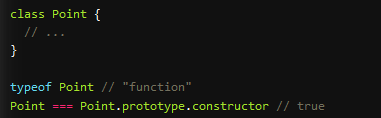
ES6提供了更接近传统语言的写法，引入了Class（类）这个概念，作为对象的模板。通过class关键字，可以定义类。基本上，ES6的class可以看作只是一个语法糖，它的绝大部分功能，ES5都可以做到，新的class写法只是让对象原型的写法更加清晰、更像面向对象编程的语法而已。上面的代码用ES6的“类”改写，就是下面这样。



上面代码定义了一个“类”，可以看到里面有一个constructor方法，这就是构造方法，而this关键字则代表实例对象。也就是说，ES5的构造函数Point，对应ES6的Point类的构造方法。

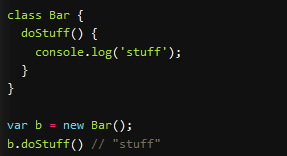
Point类除了构造方法，还定义了一个toString方法。**注意**，定义“类”的方法的时候，前面不需要加上function这个关键字，直接把函数定义放进去了就可以了。另外，方法之间不需要逗号分隔，加了会报错。

ES6的类，完全可以看作构造函数的另一种写法。



上面代码表明，类的数据类型就是函数，类本身就指向构造函数。

使用的时候，也是直接对类使用new命令，跟构造函数的用法完全一致。



注：事实上，类的所有方法都定义在类的prototype属性上面。？？？

**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。

注：Class不存在变量提升，这一点与ES5完全不同。



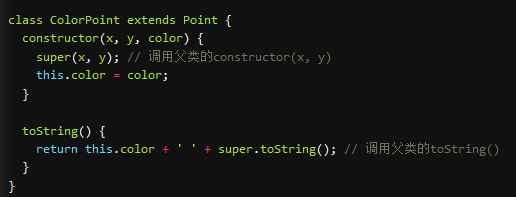
上面代码中，Foo类使用在前，定义在后，这样会报错，因为ES6不会把类的声明提升到代码头部。这种规定的原因与下文要提到的继承有关，必须保证子类在父类之后定义。

**类的继承：**

Class之间可以通过extends关键字实现继承，这比ES5的通过修改原型链实现继承，要清晰和方便很多。

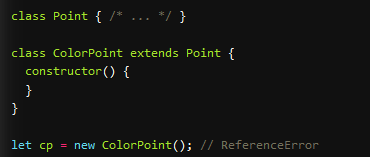
****

上面代码定义了一个ColorPoint类，该类通过extends关键字，继承了Point类的所有属性和方法。



上面代码中，constructor方法和toString方法之中，都出现了super关键字，它在这里表示父类的构造函数，用来新建父类的this对象。

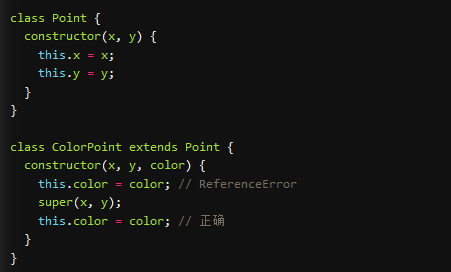
子类必须在constructor方法中调用super方法，否则新建实例时会报错。这是因为子类没有自己的this对象，而是继承父类的this对象，然后对其进行加工。如果不调用super方法，子类就得不到this对象。



如果子类没有定义constructor方法，这个方法会被默认添加，代码如下。也就是说，不管有没有显式定义，任何一个子类都有constructor方法。



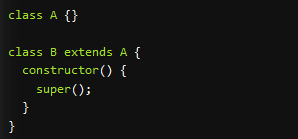
另一个需要注意的地方是，在子类的构造函数中，只有调用super之后，才可以使用this关键字，否则会报错。这是因为子类实例的构建，是基于对父类实例加工，只有super方法才能返回父类实例。



上面代码中，子类的constructor方法没有调用super之前，就使用this关键字，结果报错，而放在super方法之后就是正确的。

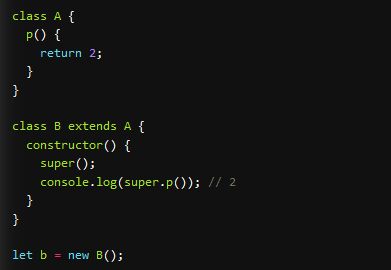
**super关键字：**既可以当作函数使用，也可以当作对象使用。在这两种情况下，它的用法完全不同。

第一种情况，super作为函数调用时，代表父类的构造函数。ES6 要求，子类的构造函数必须执行一次super函数。



**注：**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。

第二种情况，super作为对象时，指向父类的原型对象。



上面代码中，子类B当中的super.p()，就是将super当作一个对象使用。这时，super指向A.prototype（？？？），所以super.p()就相当于A.prototype.p()。

## 应用：

1：掌握class面向对象编程。

2：掌握ES6语法的灵活运用。

## 综合运用：

1：封装拖拽插件。

2：封装常见兼容。

## 周六练习：

1：掌握事件流的概念。

2：掌握兼容的解决方案。

3：掌握事件委托的应用场景。

4：鼠标跟随特效。



5：扩展案例：滑动条拖拽

