目录

[2.let与const命令 3](#_Toc499824262)

[2.1 let作用域 3](#_Toc499824263)

[2.2 暂时性死区 5](#_Toc499824264)

[2.3 do表达式 6](#_Toc499824265)

[2.4 const命令 6](#_Toc499824266)

[2.5 顶层对象 7](#_Toc499824267)

[3.变量的解构赋值 7](#_Toc499824268)

[3.1数组的解构赋值 7](#_Toc499824269)

[3.2 对象的解构赋值 9](#_Toc499824270)

[3.3 字符串的解构赋值 11](#_Toc499824271)

[3.4 数组和布尔值的解构赋值 12](#_Toc499824272)

[3.5函数参数的解构赋值 12](#_Toc499824273)

[3.6圆括号问题 13](#_Toc499824274)

[3.7用途 14](#_Toc499824275)

[4.字符串的扩展 17](#_Toc499824276)

[4.1 字符的unicode表示法 17](#_Toc499824277)

[4.2 codePointAt( ) 17](#_Toc499824278)

[4.3 String.fromCodePoint( ) 18](#_Toc499824279)

[4.4 字符串遍历器接口 18](#_Toc499824280)

[4.5 At( )方法 19](#_Toc499824281)

[4.6 normalize( ) 19](#_Toc499824282)

[4.7 includes(),startsWith(),endsWith() 20](#_Toc499824283)

[4.8 repeat()方法 20](#_Toc499824284)

[4.9 padStart(),padEnd() 21](#_Toc499824285)

[4.10 模板字符串 22](#_Toc499824286)

[4.11 string.raw() 22](#_Toc499824287)

[5.正则的扩展 23](#_Toc499824288)

[5.1RegExp构造函数 23](#_Toc499824289)

[5.2 字符串的正则方法 24](#_Toc499824290)

[5.3 u修饰符 24](#_Toc499824291)

[5.4 y修饰符 25](#_Toc499824292)

[5.5 sticky属性 27](#_Toc499824293)

[5.6 flags属性 27](#_Toc499824294)

[5.7 doAll模式 28](#_Toc499824295)

[5.8 后行断言 28](#_Toc499824296)

[5.9 unicode属性类 29](#_Toc499824297)

[5.10 具名组匹配 29](#_Toc499824298)

[6 .数值的扩展 30](#_Toc499824299)

[6.1 二进制与八进制表示法 30](#_Toc499824300)

[6.2 Number方法 31](#_Toc499824301)

[6.3 Math对象的扩展 33](#_Toc499824302)

[6.4 指数运算符 36](#_Toc499824303)

[6.5 Integer数据类型 37](#_Toc499824304)

[8．数组的扩展 38](#_Toc499824305)

[8.1 扩展运算符 38](#_Toc499824306)

[8.2 应用 39](#_Toc499824307)

[8.3 Array.from() 42](#_Toc499824308)

[8.4 array.of() 42](#_Toc499824309)

[8.5数组实例的copyWithin() 43](#_Toc499824310)

[8.6 数组实例的find()和findIndex() 44](#_Toc499824311)

[8.7 数组实例的fill() 44](#_Toc499824312)

[8.8 数组实例的entities(),keys(),values()方法 45](#_Toc499824313)

[8.9 数组实例的includes() 46](#_Toc499824314)

[9．对象的扩展 47](#_Toc499824315)

[9.1属性的简洁表示法 47](#_Toc499824316)

[9.2 属性名表达式 48](#_Toc499824317)

[9.3方法的name属性 49](#_Toc499824318)

[9.4 Object.is() 49](#_Toc499824319)

[9.5 object.assign() 50](#_Toc499824320)

[9.6 属性的可枚举性和遍历 53](#_Toc499824321)

[9.7 Object.getOwnPropertyDescriptors() 54](#_Toc499824322)

[9.8 Object.setPrototypeOf(),Object.getPrototypeOf() 55](#_Toc499824323)

[9.9 super关键字 57](#_Toc499824324)

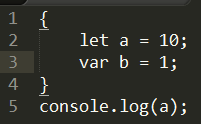
[9.10 Object.keys(),Object.values(),Object.entries() 59](#_Toc499824325)

2.let与const命令

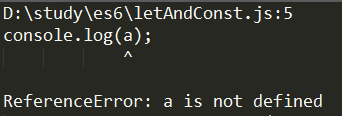
2.1 let作用域

let命令与var类似，用来申明变量，不同的是let的作用域只在let命令所在的代码块有效。

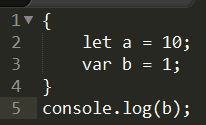
Eg2.1:



报错：



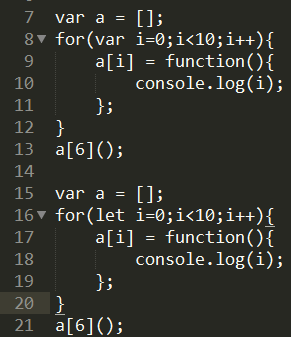
而



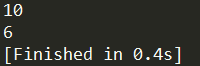
输出



Eg2.2



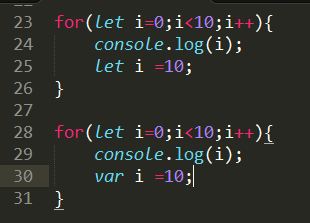
输出



使用var声明的i，是个全局的i，数组中的i指向同一个i(类似于指向同一个地址)，for循环退出时i为10，所以数组a中无论是多少输出的都是10。

而使用let申明的i只在本轮循环有效，每次循环都指向新的地址一样，所以数组a中的i是指向不同地址的i，因此数组a中的输出都不一样。

Eg2.3



分别输出：

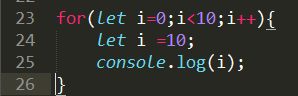




分析：

这里涉及到作用域的问题。循环体变量申明部分是父作用域，循环体内是子作用域。

上面第一个for循环由于在子作用域中已经申明了i，因此不会在找父作用域的i，但是由于输出在申明的前面，所以会报i找不到的错误。如果改成



则会输出10个10，

第二个for循环由于实际的执行顺序是这样的，先var申明，再走循环，循环中使用了var申明的i，所以报i已经被申明的错误。

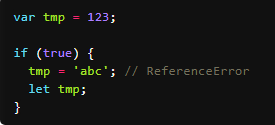
Eg2.4



分析: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。

2.2 暂时性死区

Eg:2.5

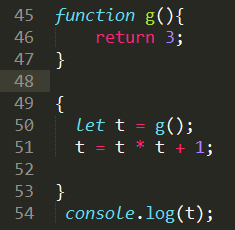


ES6明确规定，如果区块中存在let和const命令，这个**区块**对这些命令声明的变量，从一开始就形成了**封闭作用域**。凡是在声明之前就使用这些变量，就会报错。

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）。

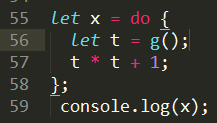
2.3 do表达式

Eg:2.6



块级作用域将两个语句封装在一起。但是，在块级作用域以外，没有办法得到t的值，因为块级作用域不返回值，除非t是全局变量。

有个提案解决方法:在块级作用域之前加上do，使它变为do表达式，然后就会返回内部最后执行的表达式的值。



2.4 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。且申明时必须赋值。

Eg:2.7



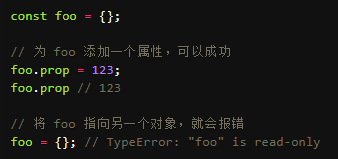
输出：



Const本质：

const实际上保证的，并不是变量的值不得改动，而是变量指向的那个内存地址不得改动。对于简单类型的数据（数值、字符串、布尔值），值就保存在变量指向的那个内存地址，因此等同于常量。但对于复合类型的数据（主要是对象和数组），变量指向的内存地址，保存的只是一个指针，const只能保证这个指针是固定的，至于它指向的数据结构是不是可变的，就完全不能控制了。

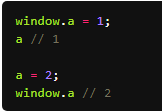
Eg:2.8



2.5 顶层对象

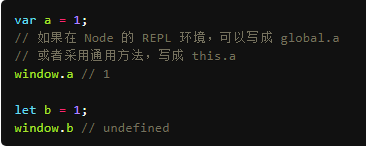
顶层对象，在浏览器环境指的是window对象，在 Node 指的是global对象。ES5 之中，顶层对象的属性与全局变量是等价的。顶层对象的属性赋值与全局变量的赋值，是同一件事。

Eg:2.9



Es6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。

Eg:2.10



3.变量的解构赋值

3.1数组的解构赋值

Eg:3.1

let [a, b, c] = [1, 2, 3];

上式，a=1,b=2,c=3。

如果解构不成功，变量的值就等于**undefined**。

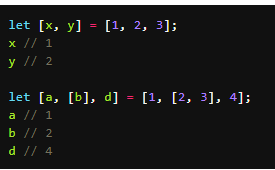
Eg:3.2



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。

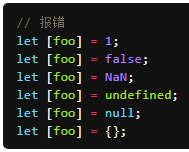
Eg:3.3



解构报错情况：

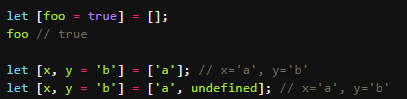
当等号右边不是数组时候，因为等号右边的值，要么转为对象以后不具备 Iterator 接口。

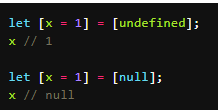
Eg:3.4



默认值情形

Eg:3.5



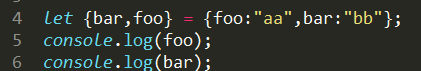


上面代码中，如果一个数组成员是**null**，默认值就不会生效，因为**null**不严格等于**undefined**。

3.2 对象的解构赋值

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。

Eg:3.6

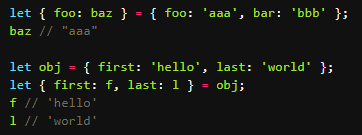


输出：



如果变量名与属性名不一致，必须写成下面这样。类似于取个别名。

Eg:3.7



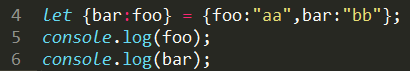
这实际上说明，对象的解构赋值是下面形式的简写



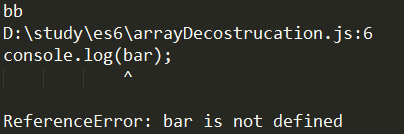
**也就是说，对象的解构赋值的内部机制，是先找到同名属性，然后再赋给对应的变量。真正被赋值的是后者，而不是前者。**

如下面这个例子

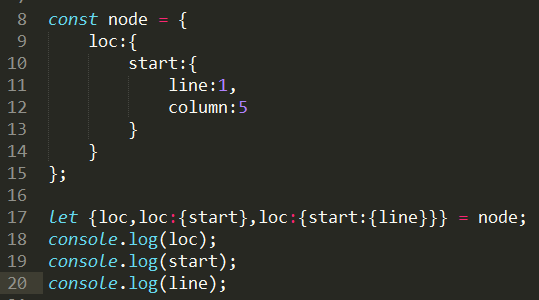
Eg:3.8



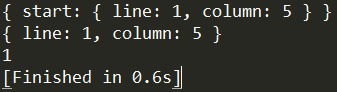
输出：



Eg:3.9



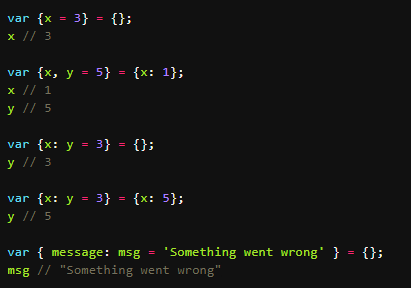
输出：



上面代码有三次解构赋值，分别是对loc、start、line三个属性的解构赋值。注意，最后一次对line属性的解构赋值之中，只有line是变量，loc和start都是**模式**，不是变量。

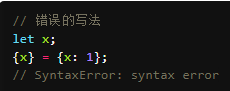
Eg:3.10

对象解构的默认值，默认值生效的条件是，对象的属性值严格等于undefined。

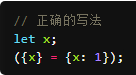


如果要将一个已经声明的变量用于解构赋值，必须非常小心。

Eg:3.11



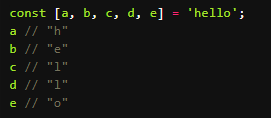
上面代码的写法会报错，因为 JavaScript 引擎会将{x}理解成一个代码块，从而发生语法错误。只有不将大括号写在行首，避免 JavaScript 将其解释为代码块，才能解决这个问题。



3.3 字符串的解构赋值

Eg3.12

此时，字符串被转换成了一个类似数组的对象



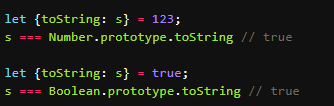
类似数组的对象都有一个length属性，因此还可以对这个属性解构赋值。



3.4 数组和布尔值的解构赋值

Eg:3.13

如果等号右边是数值和布尔值，则会先转为对象。



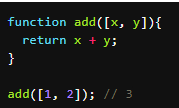
上面代码中，数值和布尔值的包装对象都有toString属性，因此变量s都能取到值。

解构赋值的规则是，只要等号右边的值不是对象或数组，就先将其转为对象。由于undefined和null无法转为对象，所以对它们进行解构赋值，都会报错。



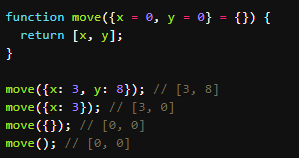
3.5函数参数的解构赋值

Eg:3.14

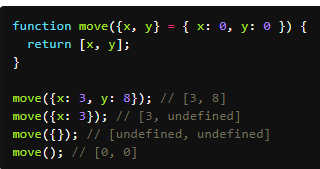


Eg:3.15

函数传参的默认值。注意比较下面两种情况,



(a)



(b)

分析：

1. 情形函数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。
2. 情形是为函数move的参数指定默认值，而不是为变量x和y指定默认值，所以会得到与前一种写法不同的结果。

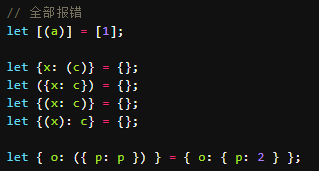
3.6圆括号问题

建议只要有可能，就不要在模式中放置圆括号。

不能使用圆括号情形

1. 变量声明语句

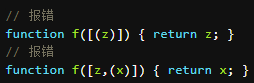
Eg：3.16



（2）函数参数

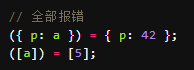
函数参数也属于变量声明，因此不能带有圆括号。

Eg：3.17



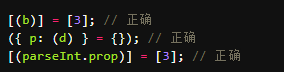
（3）赋值语句的模式

Eg：3.18



可以使用圆括号情形

Eg：3.19

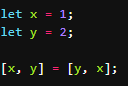


上面三行语句都可以正确执行，因为首先它们都是赋值语句，而不是声明语句；其次它们的圆括号都不属于模式的一部分。第一行语句中，模式是取数组的第一个成员，跟圆括号无关；第二行语句中，模式是p，而不是d；第三行语句与第一行语句的性质一致。

3.7用途

（1）交换变量的值

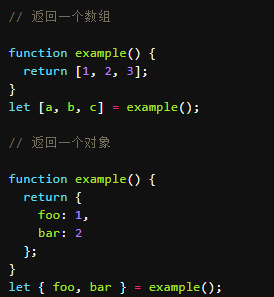
Eg：3.20



（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。

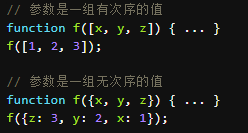
Eg：3.21



（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。

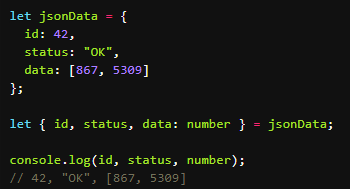
Eg：3.22



（4）提取 JSON 数据

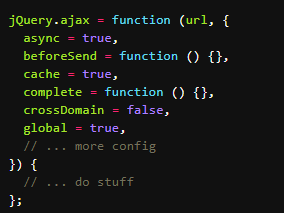
解构赋值对提取 JSON 对象中的数据，尤其有用。

Eg：3.23



（5）函数参数的默认值

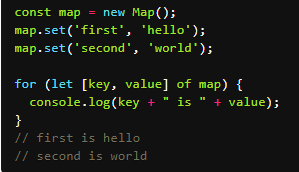
Eg：3.24



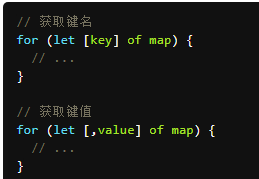
(6）遍历 Map 结构

任何部署了 Iterator 接口的对象，都可以用for...of循环遍历。Map 结构原生支持 Iterator 接口，配合变量的解构赋值，获取键名和键值就非常方便。

Eg：3.25



如果只想获取键名，或者只想获取键值，可以写成下面这样。



（7）输入模块的指定方法

加载模块时，往往需要指定输入哪些方法。解构赋值使得输入语句非常清晰。



4.字符串的扩展

4.1 字符的unicode表示法

javasSript允许采用\uxxxx形式表示一个字符，xxxx是16进制表示的，ES5中只能表示\u0000~\uFFFF之间的字符，超过这个范围，必须使用2个双字节的形式表示。

“\uD8421\uDFB7”

ES6中，只要将码点放入大括号，就能正确解读该字符。

Eg:4.1



输出：

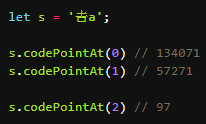


4.2 codePointAt( )

JavaScript内部，字符以UTF-16的格式存储，每个字符固定为2个字节。因此对于超过2个字节的字符其length属性为2。

ES6 提供了codePointAt方法，能够正确处理 4 个字节储存的字符，返回一个字符的码点。

Eg4.2



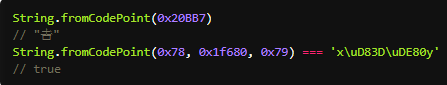
codePointAt方法返回的是码点的十进制值，如果想要十六进制的值，可以使用toString方法转换一下。

4.3 String.fromCodePoint( )

ES5 提供String.fromCharCode方法，用于从码点返回对应字符，但是这个方法不能识别 32 位的 UTF-16 字符（Unicode 编号大于0xFFFF）。

ES6 提供了String.fromCodePoint方法，可以识别大于0xFFFF的字符，弥补了String.fromCharCode方法的不足。在作用上，正好与codePointAt方法相反。

Eg:4.3



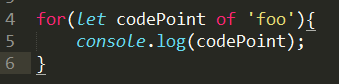
上面代码中，如果String.fromCodePoint方法有多个参数，则它们会被合并成一个字符串返回。

注意，fromCodePoint方法定义在String对象上，而codePointAt方法定义在字符串的实例对象上。

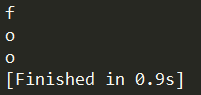
4.4 字符串遍历器接口

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循环区别是其可以识别大于0xFFFF的码点。

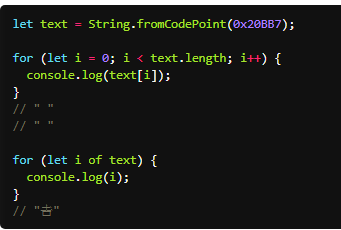
Eg:4.4



输出：



Eg4.5 与for循环对比



4.5 At( )方法

ES5 对字符串对象提供charAt方法，返回字符串给定位置的字符。该方法不能识别码点大于0xFFFF的字符。

目前，有一个提案，提出字符串实例的at方法，可以识别 Unicode 编号大于0xFFFF的字符，返回正确的字符。

Eg4.6



4.6 normalize( )

ES6 提供字符串实例的normalize()方法，用来将字符的不同表示方法统一为同样的形式，这称为 Unicode 正规化。

Eg4.7



normalize方法可以接受一个参数来指定normalize的方式，参数的四个可选值如下。

* NFC，默认参数，表示“标准等价合成”（Normalization Form Canonical Composition），返回多个简单字符的合成字符。所谓“标准等价”指的是视觉和语义上的等价。
* NFD，表示“标准等价分解”（Normalization Form Canonical Decomposition），即在标准等价的前提下，返回合成字符分解的多个简单字符。
* NFKC，表示“兼容等价合成”（Normalization Form Compatibility Composition），返回合成字符。所谓“兼容等价”指的是语义上存在等价，但视觉上不等价，比如“囍”和“喜喜”。（这只是用来举例，normalize方法不能识别中文。）
* NFKD，表示“兼容等价分解”（Normalization Form Compatibility Decomposition），即在兼容等价的前提下，返回合成字符分解的多个简单字符。

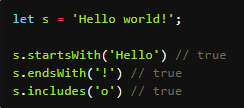
不过，normalize方法目前不能识别三个或三个以上字符的合成。

4.7 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原字符串的头部。
* **endsWith()**：返回布尔值，表示参数字符串是否在原字符串的尾部。

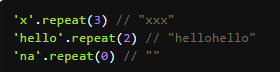
Eg:4.6



4.8 repeat()方法

repeat方法返回一个新字符串，表示将原字符串重复n次。

Eg:4.7

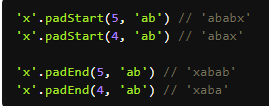


如果参数是小数，会向下取整。如果参数是0到-1，则重复0次，如果参数小于-1或者Infinity，则报错。参数NaN等同于 0。如果repeat的参数是字符串，则会先转换成数字。

4.9 padStart(),padEnd()

ES2017 引入了字符串补全长度的功能。如果某个字符串不够指定长度，会在头部或尾部补全。padStart()用于头部补全，padEnd()用于尾部补全。

Eg:4.8



上面代码中，padStart和padEnd一共接受两个参数，第一个参数用来指定字符串的最小长度，第二个参数是用来补全的字符串。

如果原字符串的长度，等于或大于指定的最小长度，则返回原字符串。

Eg:4.9



如果用来补全的字符串与原字符串，两者的长度之和超过了指定的最小长度，则会截去超出位数的补全字符串。

Eg:4.10



如果省略第二个参数，默认使用空格补全长度。

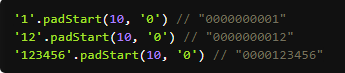
Eg:4.11



padStart()常见用途

1. 为数值补全指定位数。下面代码生成 10 位的数值字符串。

Eg:4.12



1. 提示字符串格式。

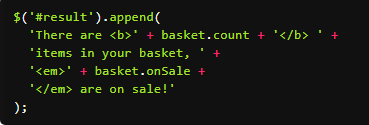
Eg:4.13



4.10 模板字符串

传统的 JavaScript 语言，输出模板通常是这样写的。

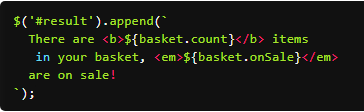
Eg:4.14



ES6模板字符串语法：

增加反引号，去除了加号，变量名的获取通过${变量名}获取，字符串不用单引号引入。

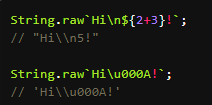
Eg:4.15



4.11 string.raw()

String.raw方法，往往用来充当模板字符串的处理函数，返回一个斜杠都被转义（即斜杠前面再加一个斜杠）的字符串，对应于替换变量后的模板字符串。

Eg:4.16



如果原字符串的斜杠已经转义，那么String.raw不会做任何处理。

5.正则的扩展

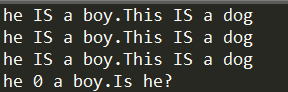
5.1RegExp构造函数

在 ES5 中，RegExp构造函数的参数有两种情况。第一种情况是，参数是字符串，这时第二个参数表示正则表达式的修饰符（flag）。第二种情况是，参数是一个正则表示式，这时会返回一个原有正则表达式的拷贝。但是，ES5 不允许此时使用第二个参数添加修饰符，否则会报错

Eg:5.1



输出：



ES6 中如果RegExp构造函数第一个参数是一个正则对象，那么可以使用第二个参数指定修饰符。而且，返回的正则表达式会忽略原有的正则表达式的修饰符，只使用新指定的修饰符。

5.2 字符串的正则方法

字符串对象共有 4 个方法，可以使用正则表达式：match()、replace()、search()和split()。

ES6 将这 4 个方法，在语言内部全部调用RegExp的实例方法，从而做到所有与正则相关的方法，全都定义在RegExp对象上。

String.prototype.match 调用 RegExp.prototype[Symbol.match]

String.prototype.replace 调用 RegExp.prototype[Symbol.replace]

String.prototype.search 调用 RegExp.prototype[Symbol.search]

String.prototype.split 调用 RegExp.prototype[Symbol.split]

5.3 u修饰符

ES6 对正则表达式添加了u修饰符，含义为“Unicode 模式”，用来正确处理大于\uFFFF的 Unicode 字符。也就是说，会正确处理四个字节的 UTF-16 编码。

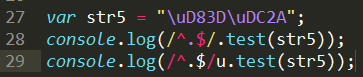
Eg:5.2

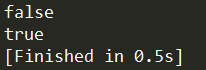


上面代码中，\uD83D\uDC2A是一个四个字节的 UTF-16 编码，代表一个字符。但是，ES5 不支持四个字节的 UTF-16 编码，会将其识别为两个字符，导致第二行代码结果为true。加了u修饰符以后，ES6 就会识别其为一个字符，所以第一行代码结果为false。

1. 点字符

Eg:5.3

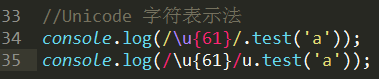


输出：  


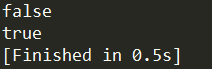
分析：对于码点大于0xFFFF的 Unicode 字符，点字符不能识别，必须加上u修饰符。，如果不添加u修饰符，正则表达式就会认为字符串为两个字符，从而匹配失败。

1. Unicode 字符表示法

Eg:5.4



输出：

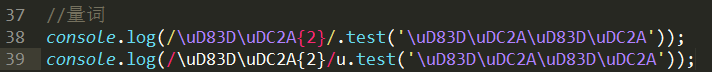


分析：

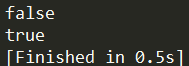
ES6 新增了使用大括号表示 Unicode 字符，这种表示法在正则表达式中必须加上u修饰符，才能识别当中的大括号，否则会被解读为量词。

1. 量词

Eg5.5:



输出：



分析：使用u修饰符后，所有量词都会正确识别码点大于0xFFFF的 Unicode 字符。

1. 预定义模式

Eg:5.6



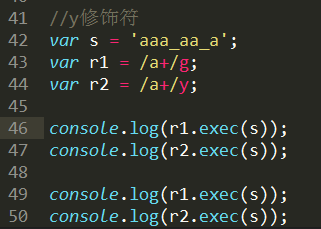
上面代码的\S是预定义模式，匹配所有不是空格的字符。只有加了u修饰符，它才能正确匹配码点大于0xFFFF的 Unicode 字符。

5.4 y修饰符

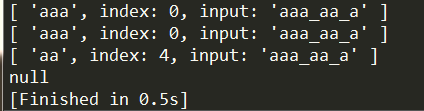
叫做“粘连”（sticky）修饰符。

y修饰符的作用与g修饰符类似，也是全局匹配，后一次匹配都从上一次匹配成功的下一个位置开始。不同之处在于，g修饰符只要剩余位置中存在匹配就可，而y修饰符确保匹配必须从剩余的第一个位置开始，这也就是“粘连”的涵义。

Eg:5.7



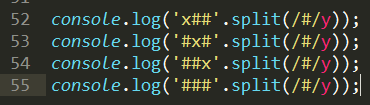
输出：



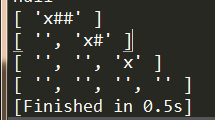
**实际上，y修饰符号隐含了头部匹配的标志^。**

在split方法中使用y修饰符，原字符串必须以分隔符开头。这也意味着，只要匹配成功，数组的第一个成员肯定是空字符串。

Eg:5.8



输出：



**y修饰符确保了匹配之间不会有漏掉的字符。**

5.5 sticky属性

与y修饰符相匹配，ES6 的正则对象多了sticky属性，表示是否设置了y修饰符。

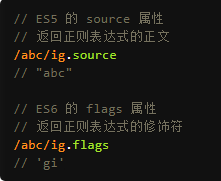
Eg:5.9



5.6 flags属性

ES6 为正则表达式新增了flags属性，会返回正则表达式的修饰符。

Eg:5.10



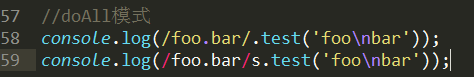
5.7 doAll模式

正则表达式中，点（.）是一个特殊字符，代表任意的单个字符，但是行终止符（line terminator character）除外。

以下四个字符属于”行终止符“。

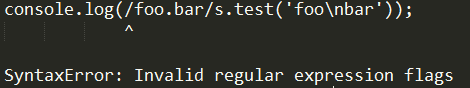
* U+000A 换行符（\n）
* U+000D 回车符（\r）
* U+2028 行分隔符（line separator）
* U+2029 段分隔符（paragraph separator）

eg:5.11



输出：

false



有一个[提案](https://github.com/mathiasbynens/es-regexp-dotall-flag)，引入/s修饰符，使得.可以匹配任意单个字符。

5.8 后行断言

有一个[提案](https://github.com/goyakin/es-regexp-lookbehind)，引入后行断言，V8 引擎 4.9 版已经支持。

”先行断言“指的是，x只有在y前面才匹配，必须写成/x(?=y)/。比如，只匹配百分号之前的数字，要写成/\d+(?=%)/。”先行否定断言“指的是，x只有不在y前面才匹配，必须写成/x(?!y)/。比如，只匹配不在百分号之前的数字，要写成/\d+(?!%)/。

Eg:5.12



输出：



“后行断言”正好与“先行断言”相反，x只有在y后面才匹配，必须写成/(?<=y)x/。比如，只匹配美元符号之后的数字，要写成/(?<=\$)\d+/。”后行否定断言“则与”先行否定断言“相反，x只有不在y后面才匹配，必须写成/(?<!y)x/。比如，只匹配不在美元符号后面的数字，要写成/(?<!\$)\d+/。

Eg:5.13



5.9 unicode属性类

目前，有一个[提案](https://github.com/mathiasbynens/es-regexp-unicode-property-escapes)，引入了一种新的类的写法\p{...}和\P{...}，允许正则表达式匹配符合 Unicode 某种属性的所有字符。

Eg:5.14



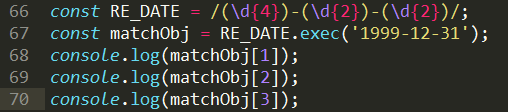
\p{Script=Greek}指定匹配一个希腊文字母，所以匹配π成功

\P{…}是\p{…}的反向匹配，即匹配不满足条件的字符。

两种类只对 Unicode 有效，所以使用的时候一定要加上u修饰符

5.10 具名组匹配

Eg:5.15



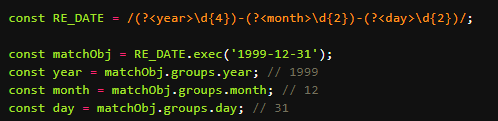
输出：



组匹配的一个问题是，每一组的匹配含义不容易看出来，而且只能用数字序号引用，要是组的顺序变了，引用的时候就必须修改序号。

现在有一个“具名组匹配”（Named Capture Groups）的[提案](https://github.com/tc39/proposal-regexp-named-groups)，允许为每一个组匹配指定一个名字，既便于阅读代码，又便于引用。

Eg:5.16



上面代码中，“具名组匹配”在圆括号内部，模式的头部添加“问号 + 尖括号 + 组名”（?<year>），然后就可以在exec方法返回结果的groups属性上引用该组名。同时，数字序号（matchObj[1]）依然有效。

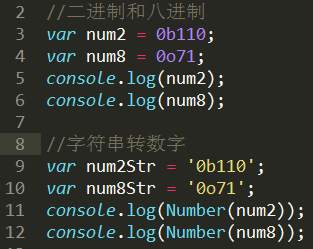
6 .数值的扩展

6.1 二进制与八进制表示法

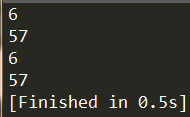
二进制：前缀0b(或0B)

八进制：前缀0o(0O)

Eg:6.1



输出：

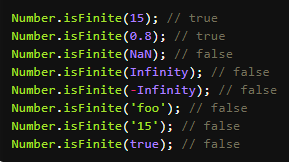


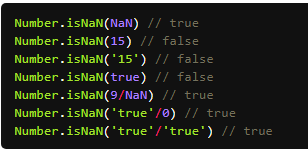
6.2 Number方法

**Number.isFinite()**用来检查一个数值是否为有限的（finite）。

**Number.isNaN()**用来检查一个值是否为NaN。

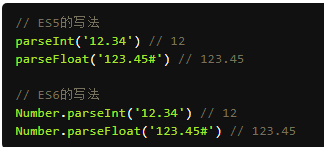
Eg6.2





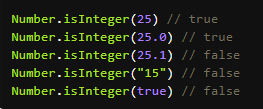
**Number.parseInt()**，**Number.parseFloat()**方法与ES5中的parseInt()和parseFloat()一样。

Eg:6.3



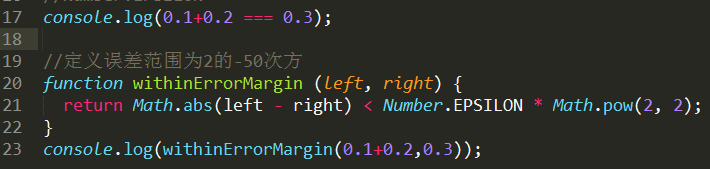
**Number.isInteger()**判断一个值是否为整数。注意：在 JavaScript 内部，整数和浮点数是同样的储存方法，所以 3 和 3.0 被视为同一个值。

Eg:6.4

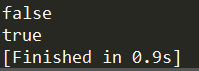


**Number.EPSILON** 它表示 1 与大于 1 的最小浮点数之间的差，为等于 2 的-52 次方。它实际上是 JavaScript 能够表示的最小精度。误差如果小于这个值，就可以认为已经没有意义了，即不存在误差了。

Eg:6.5



**结果：**



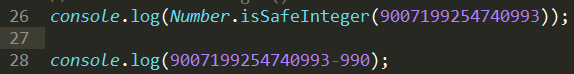
**完全整数和Number.isSafeInteger()**

JavaScript的整数范围为：(-2^53,2^53)。

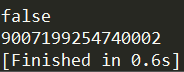
ES6引入Number.MAX\_SAFE\_INTEGER和Number.MIN\_SAFE\_INTEGER两个常量，分别表示2^53-1和-2^53+1。**Number.isSafeInteger()**方法用来判断一个整数是否落在这个范围之内。

注意：如果在**Number.isSafeInteger()**内进行计算，则用于计算的每个值都要在这个范围之内,否则计算结果会出错。

Eg:6.6



结果：

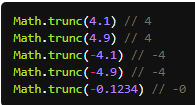


6.3 Math对象的扩展

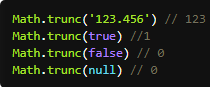
**Math.trunc()**

Math.trunc方法用于去除一个数的小数部分，返回整数部分。

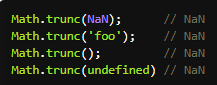
Eg：6.7



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。



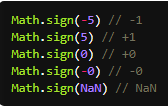
**Math.sign()**

Math.sign方法用来判断一个数到底是正数、负数、还是零。对于非数值，会先将其转换为数值。

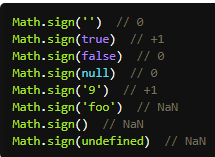
它会返回五种值。

* 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。

Eg：6.8



如果参数是非数值，会自动转为数值。对于那些无法转为数值的值，会返回NaN。



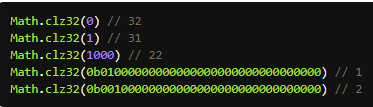
**Math.cbrt()**

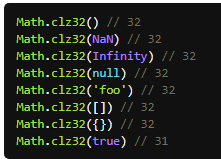
Math.cbrt方法用于计算一个数的立方根。与前面一样，对于非数值，也是先使用Number方法使其转为数值。

**Math.clz32()**

JavaScript 的整数使用 32 位二进制形式表示，Math.clz32方法返回一个数的 32 位无符号整数形式有多少个前导 0。对于小数，Math.clz32方法只考虑整数部分。对于空值或其他类型的值，Math.clz32方法会将它们先转为数值，然后再计算。

Eg：6.9

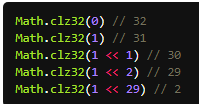




**注意：**转成数值为NaN的，经过clz32函数后值是32.

左移运算符（<<）与Math.clz32方法直接相关。

Eg:6.10



分析：左移后的结果为原来的值减去移的位数值。

**Math.imul()**

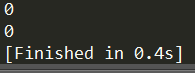
Math.imul方法返回两个数以 32 位带符号整数形式相乘的结果，返回的也是一个 32 位的带符号整数。

这个方法是因为JavaScript 有精度限制，超过 2 的 53 次方的值无法精确表示。这就是说，对于那些很大的数的乘法，低位数值往往都是不精确的，Math.imul方法可以返回正确的低位数值。

Eg:6.10



Eg：6.11

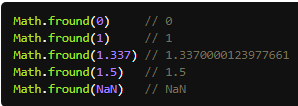


这个结果与书上不同。。。

**Math.fround()**

Math.fround 方法返回一个数的单精度浮点数形式。

Eg：6.12

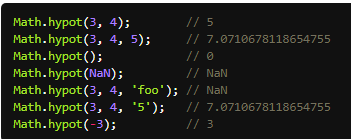


对于整数来说，Math.fround方法返回结果不会有任何不同，区别主要是那些无法用 64 个二进制位精确表示的小数。这时，Math.fround方法会返回最接近这个小数的单精度浮点数。

**Math.hypot()**

Math.hypot方法返回所有参数的平方和的平方根。

Eg:6.13



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。

**Math.expml()**

Math.expm1(x)返回 ex - 1，即Math.exp(x) - 1。

**Math.log1p()**

Math.log1p(x)方法返回1 + x的自然对数，即Math.log(1 + x)。如果x小于-1，返回NaN。

**Math.log10()**

Math.log10(x)返回以 10 为底的x的对数。如果x小于 0，则返回 NaN。

**Math.log2()**

Math.log2(x)返回以 2 为底的x的对数。如果x小于 0，则返回 NaN。

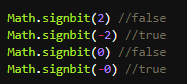
新增6个双曲函数方法

* **Math.sinh(x)** 返回x的双曲正弦（hyperbolic sine）
* **Math.cosh(x)**返回x的双曲余弦（hyperbolic cosine）
* **Math.tanh(x)** 返回x的双曲正切（hyperbolic tangent）
* **Math.asinh(x)**返回x的反双曲正弦（inverse hyperbolic sine）
* **Math.acosh(x)**返回x的反双曲余弦（inverse hyperbolic cosine）
* **Math.atanh(x)** 返回x的反双曲正切（inverse hyperbolic tangent）

**Math.signbit()**

有一个[提案](http://jfbastien.github.io/papers/Math.signbit.html)，引入了Math.signbit()方法判断一个数的符号位是否设置了。

Eg:6.14



该方法的算法如下。

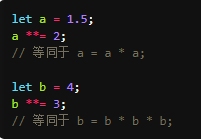
* 如果参数是NaN，返回false
* 如果参数是-0，返回true
* 如果参数是负值，返回true
* 其他情况返回false

6.4 指数运算符

ES2016 新增了一个指数运算符（\*\*）。

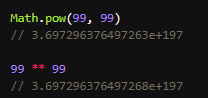
Eg:6.15





注意：在 V8 引擎中，指数运算符与Math.pow的实现不相同，对于特别大的运算结果，两者会有细微的差异。

Eg:6.16



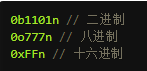
6.5 Integer数据类型

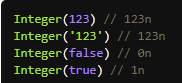
现在有一个[提案](https://github.com/tc39/proposal-bigint)，引入了新的数据类型 Integer（整数），来解决这个问题。整数类型的数据只用来表示整数，没有位数的限制，任何位数的整数都可以精确表示。

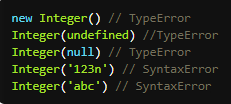
为了与 Number 类型区别，Integer 类型的数据必须使用后缀n表示。

Eg:6.17









Integer 类型的+、-、\*和\*\*这四个二元运算符，与 Number 类型的行为一致。除法运算/会舍去小数部分，返回一个整数。

几乎所有的 Number 运算符都可以用在 Integer，但是有两个除外：不带符号的右移位运算符>>>和一元的求正运算符+，使用时会报错。前者是因为>>>要求最高位补 0，但是 Integer 类型没有最高位，导致这个运算符无意义。后者是因为一元运算符+在 asm.js 里面总是返回 Number 类型或者报错。

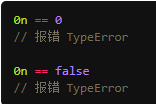
Integer 类型不能与 Number 类型进行混合运算。

Eg:6.18



相等运算符（==）会改变数据类型，也是不允许混合使用。

Eg:6.19



精确相等运算符（===）不会改变数据类型，因此可以混合使用。

Eg:6.20

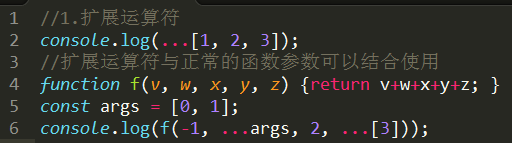


8．数组的扩展

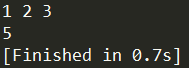
8.1 扩展运算符

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。

Eg:8.1



输出：

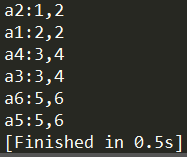


8.2 应用

Eg:8.2

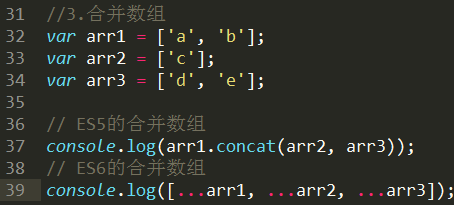


输出:

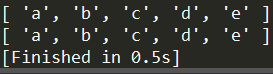


分析：a2并不是a1的克隆，而是指向同一份数据的另一个指针。修改a2，会直接导致a1的变化。而a4和a6分别是a3与a5的克隆，修改它们并不会影响原先的值。

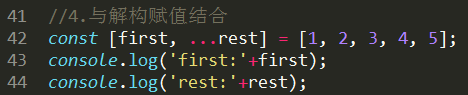
Eg8.3:



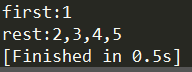
输出:



Eg:8.4



输出:

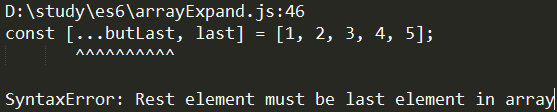


**注意:如果将扩展运算符用于数组赋值，只能放在参数的最后一位，否则会报错。**

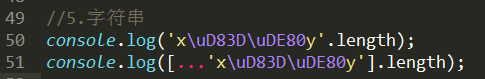
Eg：8.5



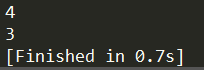
输出:



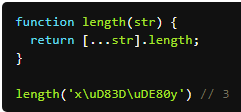
Eg:8.6



输出:



上面代码的第一种写法，JavaScript 会将四个字节的 Unicode 字符，识别为 2 个字符，采用扩展运算符就没有这个问题。因此，正确返回字符串长度的函数，可以像下面这样写。



实现 Iterator 接口的对象

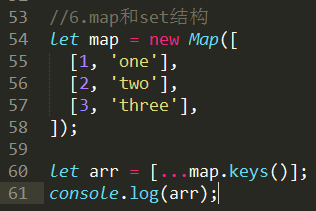
任何 Iterator 接口的对象（参阅 Iterator 一章），都可以用扩展运算符转为真正的数组。

Eg:8.7



上面代码中，querySelectorAll方法返回的是一个nodeList对象。它不是数组，而是一个类似数组的对象。这时，扩展运算符可以将其转为真正的数组，原因就在于NodeList对象实现了 Iterator 。

Eg:8.8



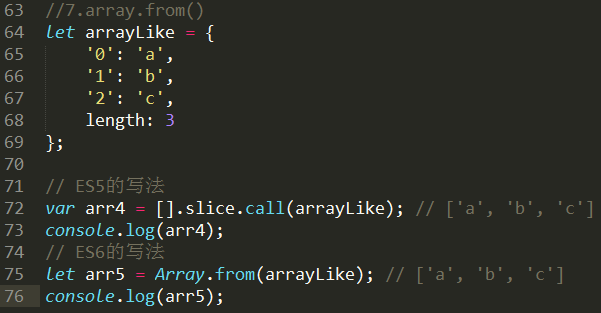
输出:



8.3 Array.from()

Array.from方法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（包括 ES6 新增的数据结构 Set 和 Map）。

Eg:8.9



扩展运算符（...）也可以将某些数据结构转为数组。扩展运算符背后调用的是遍历器接口（Symbol.iterator），如果一个对象没有部署这个接口，就无法转换。Array.from方法还支持类似数组的对象。所谓类似数组的对象，本质特征只有一点，即必须有length属性。因此，任何有length属性的对象，都可以通过Array.from方法转为数组，而此时扩展运算符就无法转换。

Eg:8.10

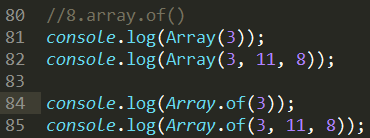


输出:

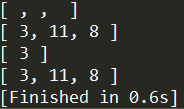


8.4 array.of()

Eg:8.11



输出:



分析:

Array方法一个参数、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。

8.5数组实例的copyWithin()

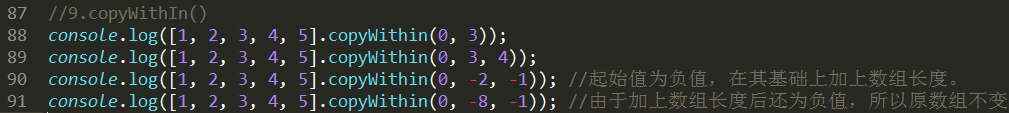
数组实例的copyWithin方法，在当前数组内部，将指定位置的成员复制到其他位置（会覆盖原有成员），然后返回当前数组。也就是说，使用这个方法，会修改当前数组。

Array.prototype.copyWithin(target, start = 0, end = this.length)

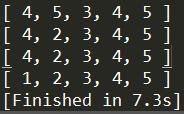
它接受三个参数。

* 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。

eg:8.12



输出:

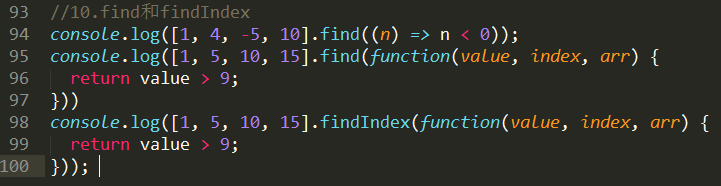


8.6 数组实例的find()和findIndex()

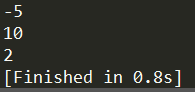
数组实例的find方法，用于找出第一个符合条件的数组成员。它的参数是一个回调函数，所有数组成员依次执行该回调函数，直到找出第一个返回值为true的成员，然后返回该成员。如果没有符合条件的成员，则返回undefined。

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，如果所有成员都不符合条件，则返回-1。

Eg:8.13



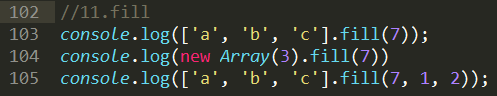
输出:



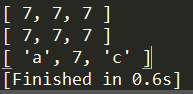
8.7 数组实例的fill()

fill方法使用给定值，填充一个数组。

Eg:8.14



输出:



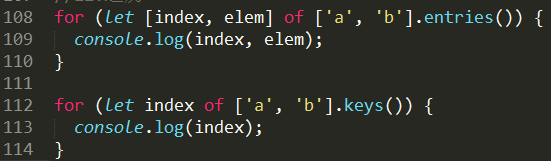
分析；

fill方法用于空数组的初始化非常方便。数组中已有的元素，会被全部抹去。fill方法还可以接受第二个和第三个参数，用于指定填充的起始位置和结束位置。

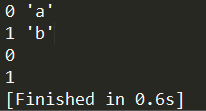
8.8 数组实例的entities(),keys(),values()方法

keys()是对键名的遍历、values()是对键值的遍历，entries()是对键值对的遍历。

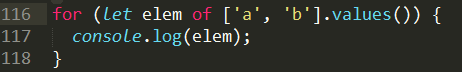
Eg:8.15



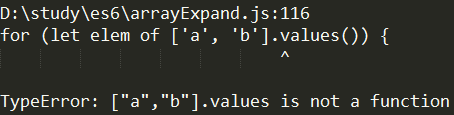
输出；



Eg：8.16



输出:

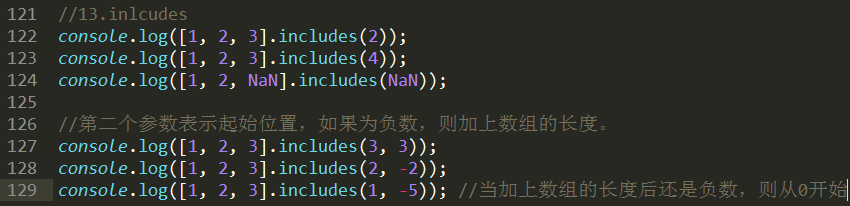


8.9 数组实例的includes()

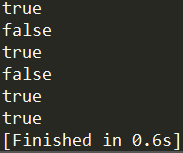
Array.prototype.includes方法返回一个布尔值，表示某个数组是否包含给定的值，与字符串的includes方法类似。ES2016 引入了该方法。

该方法的第二个参数表示搜索的起始位置，默认为0。如果第二个参数为负数，则表示倒数的位置，如果这时它大于数组长度（比如第二个参数为-4，但数组长度为3），则会重置为从0开始。

Eg:8.17



输出；



没有该方法之前，我们通常使用数组的indexOf方法，检查是否包含某个值。

if (arr.indexOf(el) !== -1) {

// ...

}

indexOf方法有两个缺点，一是不够语义化，它的含义是找到参数值的第一个出现位置，所以要去比较是否不等于-1，表达起来不够直观。二是，它内部使用严格相等运算符（===）进行判断，这会导致对NaN的误判。

Eg:8.18



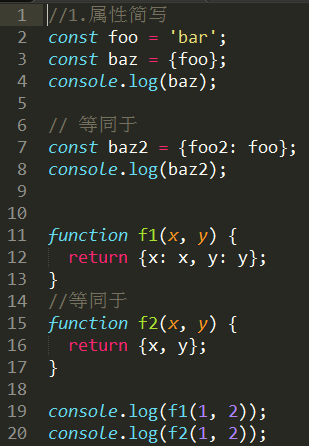
输出:



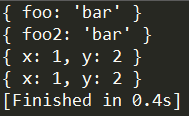
9．对象的扩展

9.1属性的简洁表示法

Eg:9.1



输出:



分析:

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。

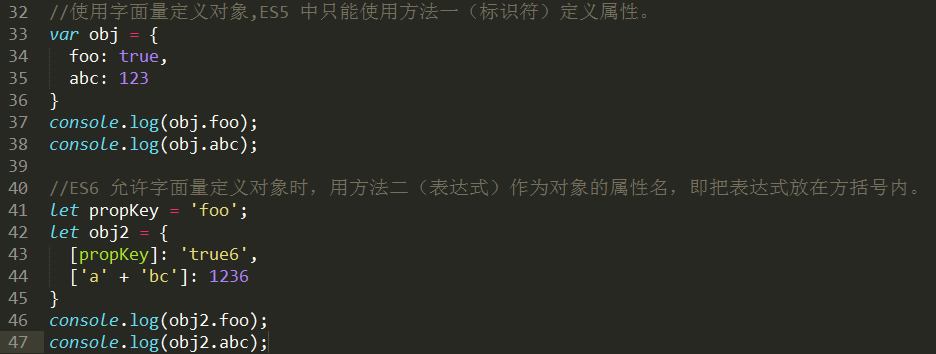
Eg:9.2

方法简写：

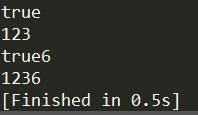


9.2 属性名表达式

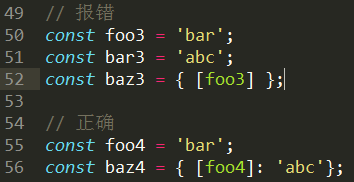
Eg:9.3



输出；



**注意，属性名表达式与简洁表示法，不能同时使用，会报错。**

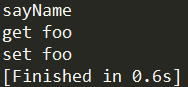


9.3方法的name属性

Eg:9.4



输出:

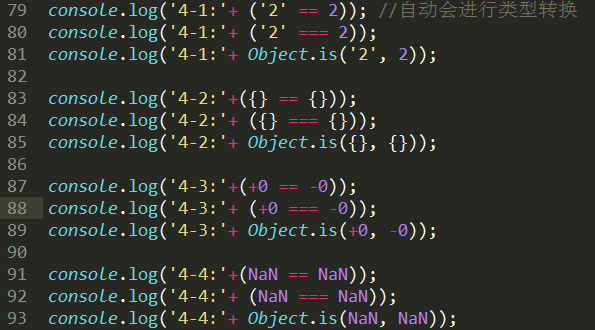


如果对象的方法使用了取值函数（getter）和存值函数（setter），则name属性不是在该方法上面，而是该方法的属性的描述对象的get和set属性上面，返回值是方法名前加上get和set。

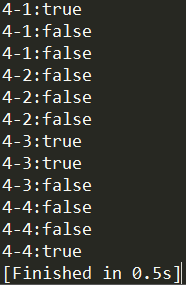
9.4 Object.is()

JavaScript 缺乏一种运算，在所有环境中，只要两个值是一样的，它们就应该相等。ES6 提出“Same-value equality”（同值相等）算法，用来解决这个问题。Object.is就是部署这个算法的新方法。它用来比较两个值是否严格相等，与严格比较运算符（===）的行为基本一致。不同之处只有两个：一是+0不等于-0，二是NaN等于自身。

Eg9.5

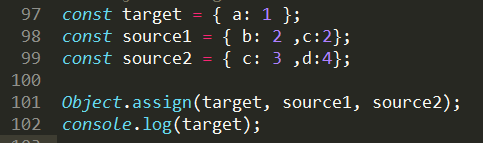


输出:



9.5 object.assign()

Eg 9.6:



输出:

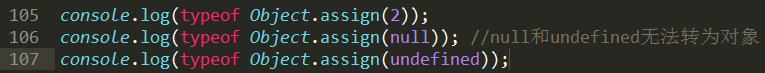


Object.assign方法的第一个参数是目标对象，后面的参数都是源对象。

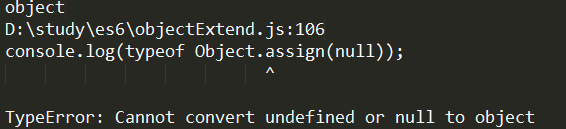
注意，如果目标对象与源对象有同名属性，或多个源对象有同名属性，则后面的属性会覆盖前面的属性。

当参数只有一个时，如果该参数不是对象，则会先其转为对象。但是null和undefined不能转为对象。

Eg 9.7:

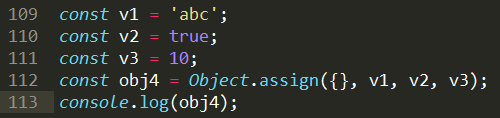


输出:



其他类型的值（即数值、字符串和布尔值）不在首参数，也不会报错。但是，除了字符串会以数组形式，拷贝入目标对象，其他值都不会产生效果。

Eg:9.8



输出:



分析:

上面代码中，v1、v2、v3分别是字符串、布尔值和数值，结果只有字符串合入目标对象（以字符数组的形式），数值和布尔值都会被忽略。这是因为只有字符串的包装对象，会产生可枚举属性。

注意点:

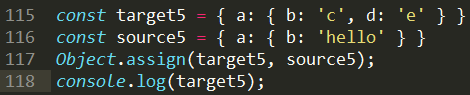
**（1）浅拷贝**

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。

**（2）同名属性的替换**

对于这种嵌套的对象，一旦遇到同名属性，Object.assign的处理方法是替换，而不是添加。

Eg:9.9



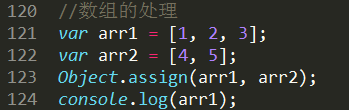
输出:



**（3）数组的处理**

Object.assign可以用来处理数组，但是会把数组视为对象。

Eg:9.10



输出:

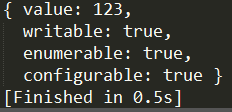


9.6 属性的可枚举性和遍历

Eg:9.11



输出；



描述对象的enumerable属性，称为”可枚举性“，如果该属性为false，就表示某些操作会忽略当前属性。

目前，有四个操作会忽略enumerable为false的属性。

* for...in循环：只遍历对象自身的和继承的可枚举的属性。
* Object.keys()：返回对象自身的所有可枚举的属性的键名。
* JSON.stringify()：只串行化对象自身的可枚举的属性。
* Object.assign()： 忽略enumerable为false的属性，只拷贝对象自身的可枚举的属性。

属性的遍历

ES6 一共有 5 种方法可以遍历对象的属性。

**（1）for...in**

for...in循环遍历对象自身的和继承的可枚举属性（不含 Symbol 属性）。

**（2）Object.keys(obj)**

Object.keys返回一个数组，包括对象自身的（不含继承的）所有可枚举属性（不含 Symbol 属性）的键名。

**（3）Object.getOwnPropertyNames(obj)**

Object.getOwnPropertyNames返回一个数组，包含对象自身的所有属性（不含 Symbol 属性，但是包括不可枚举属性）的键名。

**（4）Object.getOwnPropertySymbols(obj)**

Object.getOwnPropertySymbols返回一个数组，包含对象自身的所有 Symbol 属性的键名。

**（5）Reflect.ownKeys(obj)**

Reflect.ownKeys返回一个数组，包含对象自身的所有键名，不管键名是 Symbol 或字符串，也不管是否可枚举。

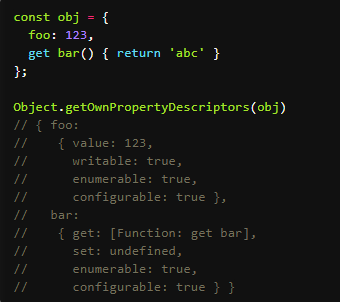
以上的 5 种方法遍历对象的键名，都遵守同样的属性遍历的次序规则。

* 首先遍历所有数值键，按照数值升序排列。
* 其次遍历所有字符串键，按照加入时间升序排列。
* 最后遍历所有 Symbol 键，按照加入时间升序排列。

9.7 Object.getOwnPropertyDescriptors()

前面说过，Object.getOwnPropertyDescriptor方法会返回某个对象属性的描述对象（descriptor）。ES2017 引入了Object.getOwnPropertyDescriptors方法，返回指定对象所有自身属性（非继承属性）的描述对象。

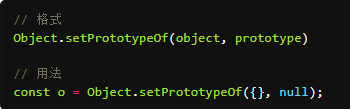
Eg:9.12



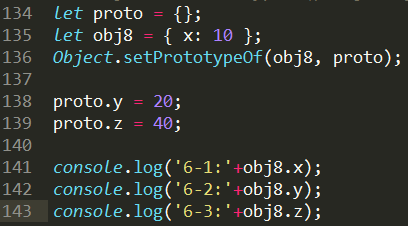
上面代码中，Object.getOwnPropertyDescriptors方法返回一个对象，所有原对象的属性名都是该对象的属性名，对应的属性值就是该属性的描述对象。

9.8 Object.setPrototypeOf(),Object.getPrototypeOf()

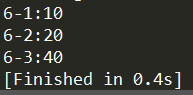
Object.setPrototypeOf方法的作用与\_\_proto\_\_相同，用来设置一个对象的prototype对象，返回参数对象本身。它是 ES6 正式推荐的设置原型对象的方法。



Eg:9.13



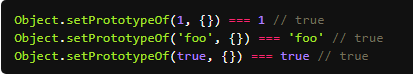
输出；



分析: 上面代码将proto对象设为obj8对象的原型，所以从obj8对象可以读取proto对象的属性。

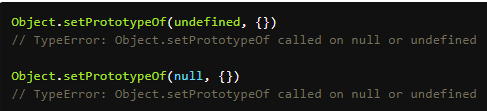
如果第一个参数不是对象，会自动转为对象。但是由于返回的还是第一个参数，所以这个操作不会产生任何效果。

Eg:9.14



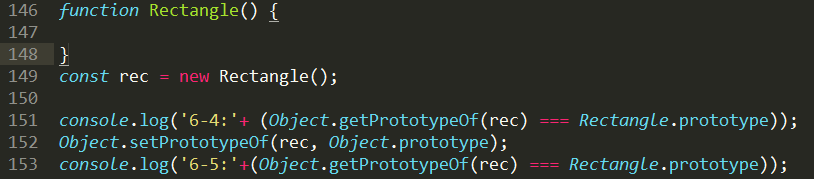
由于undefined和null无法转为对象，所以如果第一个参数是undefined或null，就会报错。

Eg:9.15

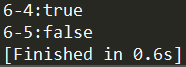


Object.getPrototypeOf()用于读取一个对象的原型

Eg:9.16

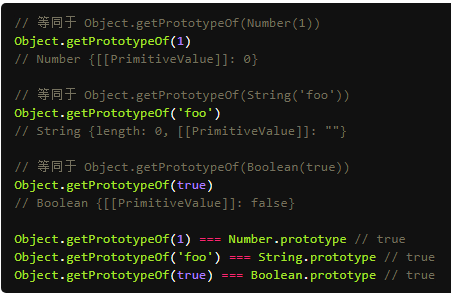


输出:



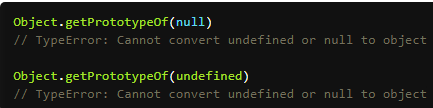
如果参数不是对象，会被自动转为对象。

Eg:9.17



如果参数是undefined或null，它们无法转为对象，所以会报错。

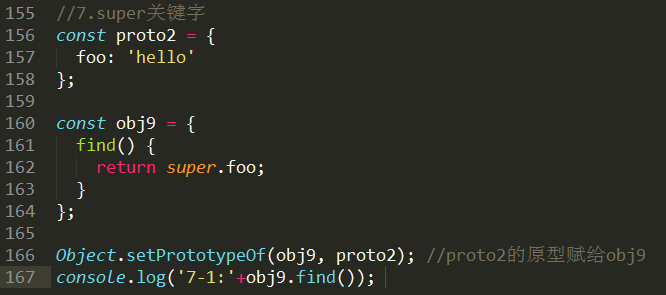
Eg:9.18



9.9 super关键字

我们知道，this关键字总是指向函数所在的当前对象，ES6 又新增了另一个类似的关键字super，指向当前对象的原型对象。

Eg:9.19

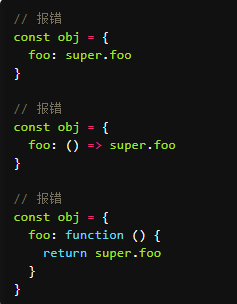


输出:

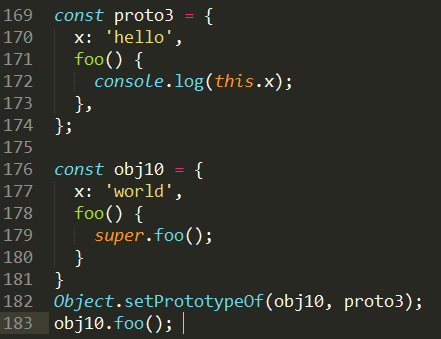


注意，super关键字表示原型对象时，只能用在对象的方法之中，用在其他地方都会报错。

Eg:9.20



Eg:9.21



输出:



分析：

Super.foo指向原型对象proto3的foo方法，但是绑定的this却还是当前对象obj10，因此输出的就是world。

9.10 Object.keys(),Object.values(),Object.entries()

ES2017 引入了跟Object.keys配套的Object.values和Object.entries，作为遍历一个对象的补充手段，供for...of循环使用。

目前这两个新加的函数运行报错。

Eg:9.22

