typeof null ----object

Typeof [] -----object

Typeof {} -----object

Typeof math----object

Var b = [5,67,8]

b instanceof Array -- true

B instanceof Object -- true

<a href="javascript:void(document.form.submit())">

数据类型

最新的 ECMAScript 标准定义了7种数据类型：

* 六种基本数据类型:
  + 布尔值（Boolean），有2个值分别是：true 和 false.
  + null ， 一个表明 null 值的特殊关键字。 JavaScript 是大小写敏感的，因此 null 与 Null、NULL或变体完全不同。
  + undefined ，和 null 一样是一个特殊的关键字，undefined 表示变量未定义时的属性。
  + 数字（Number），整数或浮点数，例如： 42 或者 3.14159。
  + 字符串（String），字符串是一串表示文本值的字符序列，例如："Howdy" 。
  + 代表（Symbol） ( 在 ECMAScript 6 中新添加的类型).。一种实例是唯一且不可改变的数据类型。
* 以及对象（Object）。

Symbol()函数会返回**symbol**类型的值，该类型具有静态属性和静态方法。它的静态属性会暴露几个内建的成员对象；它的静态方法会暴露全局的symbol注册，且类似于内建对象类，但作为构造函数来说它并不完整，因为它不支持语法："new Symbol()"。

每个从Symbol()返回的symbol值都是唯一的。一个symbol值能作为对象属性的标识符；这是该数据类型仅有的目的。更进一步的解析见—— [glossary entry for Symbol](https://developer.mozilla.org/en-US/docs/Glossary/Symbol)。

**symbol** 是一种基本数据类型 （[primitive data type](https://developer.mozilla.org/en-US/docs/Glossary/Primitive)）。

**语法**

Symbol(*[description]*)

参数

**description**可选

可选的字符串。symbol的描述，可用于调试但不能访问symbol本身。

**描述**

直接使用Symbol()创建新的symbol类型，并用一个可选的字符串作为其描述。

var sym1 = Symbol();

var sym2 = Symbol('foo');

var sym3 = Symbol('foo');

上面的代码创建了三个新的symbol类型。 注意，Symbol("foo") 不会强制字符串 “foo” 成为一个 symbol类型。它每次都会创建一个新的 symbol类型：

Symbol("foo") === Symbol("foo"); // false

下面使用 [new](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/new) 运算符的语法将会抛出一个 [TypeError](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/TypeError) 错误：

var sym = new Symbol(); // TypeError

**箭头函数表达式**的语法比[函数表达式](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/function)更简洁，并且没有自己的[this](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/this)，[arguments](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Functions/arguments)，[super](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/super)或 [new.target](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/new.target)。这些函数表达式更适用于那些本来需要匿名函数的地方，并且它们不能用作构造函数。

基础语法

(参数1, 参数2, …, 参数N) => { 函数声明 }

//相当于：(参数1, 参数2, …, 参数N) =>{ return 表达式; }

(参数1, 参数2, …, 参数N) => 表达式（单一）

// 当只有一个参数时，圆括号是可选的：

(单一参数) => {函数声明}

单一参数 => {函数声明}

// 没有参数的函数应该写成一对圆括号。

() => {函数声明}

高级语法

//加括号的函数体返回对象字面表达式：

参数=> ({foo: bar})

//支持剩余参数和默认参数

(参数1, 参数2, ...rest) => {函数声明}

(参数1 = 默认值1,参数2, …, 参数N = 默认值N) => {函数声明}

//同样支持参数列表解构

let f = ([a, b] = [1, 2], {x: c} = {x: a + b}) => a + b + c;

f(); // 6

箭头函数没有自己的this指针，通过 call() 或 apply() 方法调用一个函数时，只能传递参数（不能绑定this---译者注）他们的第一个参数会被忽略。（这种现象对于bind方法同样成立---译者注）

不绑定arguments

箭头函数不绑定[Arguments 对象](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Functions/arguments)。因此，在本示例中，arguments只是引用了封闭作用域内的arguments，在大多数情况下，使用[剩余参数](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Functions/Rest_parameters)是相较使用arguments对象的更好选择。

全局共享的 Symbol

上面使用Symbol() 函数的语法，不会在你的整个代码库中创建一个可用的全局?symbol类型。 要创建跨文件可用的symbol，甚至跨域（每个都有它自己的全局作用域） , 使用 [Symbol.for()](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Symbol/for) 方法和  [Symbol.keyFor()](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Symbol/keyFor) 方法从全局的symbol注册?表设置和取得symbol。

https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global\_Objects/Symbol/for

https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global\_Objects/Symbol/keyFor

在对象中查找 Symbol 属性

[Object.getOwnPropertySymbols()](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Object/getOwnPropertySymbols) 方法让你在查找一个给定对象的符号属性时返回一个?symbol类型的数组。注意，每个初始化的对象都是没有自己的symbol属性的，因此这个数组可能为空，除非你已经在对象上设置了symbol属性。

Symbol 类型转换

当使用 symbol 值进行类型转换时需要注意一些事情：

* 尝试将一个 symbol 值转换为一个 number 值时，会抛出一个 [TypeError](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/TypeError) 错误  (e.g. +sym or sym | 0).
* 使用宽松相等时， Object(sym) == sym returns true.
* 这会阻止你从一个 symbol 值隐式地创建一个新的 string 类型的属性名。例如，Symbol("foo") + "bar" 将抛出一个 [TypeError](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/TypeError) (can't convert symbol to string).
* ["safer" String(sym) conversion](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String#String_conversion) 的作用会像symbol类型调用 [Symbol.prototype.toString()](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Symbol/toString) 一样，但是注意 new String(sym) 将抛出异常。

Symbols 与 for...in 迭代

Symbols 在 [for...in](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...in) 迭代中不可枚举。另外，[Object.getOwnPropertyNames()](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Object/getOwnPropertyNames) 不会返回 symbol 对象的属性，但是你能使用 [Object.getOwnPropertySymbols()](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Object/getOwnPropertySymbols) 得到它们。ymbols 与 JSON.stringify()

当使用 JSON.strIngify() 时以 symbol 值作为键的属性会被完全忽略：

数字

在 JavaScript 里面，数字均为双精度浮点类型double-precision 64-bit binary format IEEE 754 （也就是说一个数字的范围只能在 -(253 -1) 和 253 -1之间）。整型数据也不例外。除了能够表示浮点数，数字类型也还能表示三种符号值: +Infinity（正无穷）、-Infinity（负无穷）和 NaN (not-a-number非数字)。

Date时间表示的形式：

var Xmas95 = new Date("December 25, 1995 13:30:00")

var Xmas95 = new Date(1995, 11, 25)

var Xmas95 = new Date(1995, 11, 25, 9, 30, 0)

String api

|  |  |
| --- | --- |
| **方法** | **描述** |
| [charAt](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/String/charAt), [charCodeAt](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/String/charCodeAt), [codePointAt](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/String/codePointAt) | 返回字符串指定位置的字符或  者字符编码。 |
| [indexOf](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/String/indexOf), [lastIndexOf](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/String/lastIndexOf) | 分别返回字符串中指定子串的  位置或最后位置。 |
| [startsWith](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/String/startsWith), [endsWith](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/String/endsWith), [includes](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/String/includes) | 返回字符串是否以指定字符串开始、  结束或包含指定字符串。 |
| [concat](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/String/concat) | 连接两个字符串并返回新的字符串。 |
| [fromCharCode](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/String/fromCharCode), [fromCodePoint](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/String/fromCodePoint) | 从指定的Unicode值序列构造一个  字符串。这是一个String类方法，  不是实例方法。 |
| [split](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/String/split) | 通过将字符串分离成一个个子串来把一个String对象分裂到一个字符串数组中。 |
| [slice](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/String/slice) | 从一个字符串提取片段并作为  新字符串返回。 |
| [substring](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/String/substring), [substr](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/String/substr) | 分别通过指定起始和结束位置，  起始位置和长度来返回字符串的指定  子集。 |
| [match](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/String/match), [replace](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/String/replace), [search](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/String/search) | 通过正则表达式来工作. |
| [toLowerCase](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/String/toLowerCase), [toUpperCase](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/String/toUpperCase) | 分别返回字符串的小写表示和  大写表示。 |
| [normalize](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/String/normalize) | 按照指定的一种 Unicode  正规形式将当前字符串正规化。 |
| [repeat](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/String/repeat) | 将字符串内容重复指定次数后返回。 |
| [trim](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/String/trim) | 去掉字符串开头和结尾的空白字符。 |

**国际化**

[Intl](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Intl) 对象是ECMAScript国际化API的命名空间, 它提供了语言敏感的字符串比较，数字格式化和日期时间格式化功能.  [Collator](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Collator), [NumberFormat](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/NumberFormat), 和 [DateTimeFormat](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/DateTimeFormat) 对象的构造函数是Intl对象的属性.

|  |  |
| --- | --- |
| [(?:x)](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions#special-non-capturing-parentheses) | 匹配 'x' 但是不记住匹配项。这种叫作非捕获括号，使得你能够定义为与  正则表达式运算符一起使用的子表达式。来看示例表达式 /(?:foo){1,2}/。  如果表达式是 /foo{1,2}/，{1,2}将只对 ‘foo’ 的最后一个字符 ’o‘ 生效。  如果使用非捕获括号，则{1,2}会匹配整个 ‘foo’ 单词。 |
| [x(?=y)](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions#special-lookahead) | 匹配'x'仅仅当'x'后面跟着'y'.这种叫做正向肯定查找。  例如，/Jack(?=Sprat)/会匹配到'Jack'仅仅当它后面跟着'Sprat'。  /Jack(?=Sprat|Frost)/匹配‘Jack’仅仅当它后面跟着'Sprat'或者是‘Frost’。  但是‘Sprat’和‘Frost’都不是匹配结果的一部分。 |

正则表达式：



|  |  |
| --- | --- |
| [x(?!y)](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions#special-negated-look-ahead) | 匹配'x'仅仅当'x'后面不跟着'y',这个叫做正向否定查找。  例如，/\d+(?!\.)/匹配一个数字仅仅当这个数字后面没有跟小数点的时候。  正则表达式/\d+(?!\.)/.exec("3.141")匹配‘141’而不是‘3.141’ |



|  |  |
| --- | --- |
| [[\b]](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions#special-backspace) | 匹配一个退格(U+0008)。（不要和\b混淆了。） |
| [\b](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions#special-word-boundary) | 匹配一个词的边界。一个词的边界就是一个词不被另外一个“字”字符  跟随的位置或者没有其他“字”字符在其前面的位置。注意，  一个匹配的词的边界并不包含在匹配的内容中。换句话说，  一个匹配的词的边界的内容的长度是0。（不要和[\b]混淆了）  例子：  /\bm/匹配“moon”中的‘m’；  /oo\b/并不匹配"moon"中的'oo'，因为'oo'被一个“字”字符'n'紧跟着。  /oon\b/匹配"moon"中的'oon'，因为'oon'是这个字符串的结束部分。  这样他没有被一个“字”字符紧跟着。  /\w\b\w/将不能匹配任何字符串，因为在一个单词中间的字符  永远也不可能同时满足没有“字”字符跟随和有“字”字符跟随两种情况。  **注意:** JavaScript的正则表达式引擎将[特定的字符集](http://www.ecma-international.org/ecma-262/5.1/#sec-15.10.2.6)定义为“字”字符。  不在该集合中的任何字符都被认为是一个断词。这组字符相当有限：  它只包括大写和小写的罗马字母，十进制数字和下划线字符。  不幸的是，重要的字符，例如“é”或“ü”，被视为断词。 |



\b在前面和在后面的区别

|  |  |
| --- | --- |
| [\n](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions#special-line-feed) | 匹配一个换行符 (U+000A)。 |
| [\r](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions#special-carriage-return) | 匹配一个回车符 (U+000D)。 |

**表 4.2 使用正则表达式的方法**

|  |  |
| --- | --- |
| **方法** | **描述** |
| [exec](https://developer.mozilla.org/zh-CN/docs/JavaScript/Reference/Global_Objects/RegExp/exec) | 一个在字符串中执行查找匹配的RegExp方法，  它返回一个数组（未匹配到则返回null）。 |
| [test](https://developer.mozilla.org/zh-CN/docs/JavaScript/Reference/Global_Objects/RegExp/test) | 一个在字符串中测试是否匹配的RegExp方法，它返回true或false。 |
| [match](https://developer.mozilla.org/zh-CN/docs/JavaScript/Reference/Global_Objects/String/match) | 一个在字符串中执行查找匹配的String方法，  它返回一个数组或者在未匹配到时返回null。 |
| [search](https://developer.mozilla.org/zh-CN/docs/JavaScript/Reference/Global_Objects/String/search) | 一个在字符串中测试匹配的String方法，  它返回匹配到的位置索引，或者在失败时返回-1。 |
| [replace](https://developer.mozilla.org/zh-CN/docs/JavaScript/Reference/Global_Objects/String/replace) | 一个在字符串中执行查找匹配的String方法，  并且使用替换字符串替换掉匹配到的子字符串。 |
| [split](https://developer.mozilla.org/zh-CN/docs/JavaScript/Reference/Global_Objects/String/split) | 一个使用正则表达式或者一个固定字符串分隔一个字符串，  并将分隔后的子字符串存储到数组中的String方法。 |

**表 4.3 正则表达式执行返回信息**

|  |  |  |  |
| --- | --- | --- | --- |
| 对象 | 属性或索引 | 描述 | 在例子中对应的值 |
| myArray |  | 匹配到的字符串和所有被记住的子字符串。 | ["dbbd", "bb"] |
| index | 在输入的字符串中匹配到的以0开始的索引值。 | 1 |
| input | 初始字符串。 | "cdbbdbsbz" |
| [0] | 匹配到的所有字符串（并不是匹配后记住的字符串）。  注：原文"The last matched characters."，应该是原版错误。  匹配到的最终字符。 | "dbbd" |
| myRe | lastIndex | 下一个匹配的索引值。（这个属性只有在使用  g参数时可用在 [通过参数进行高级搜索](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Regular_Expressions#.E9.80.9A.E8.BF.87.E5.8F.82.E6.95.B0.E8.BF.9B.E8.A1.8C.E9.AB.98.E7.BA.A7.E6.90.9C.E7.B4.A2) 一节有详细的描述.) | 5 |
| source | 模式文本。在正则表达式创建时更新，不执行。 | "d(b+)d" |

**正则表达式标志**

|  |  |
| --- | --- |
| **标志** | **描述** |
| g | 全局搜索。 |
| i | 不区分大小写搜索。 |
| m | 多行搜索。 |
| y | 执行“粘性”搜索,匹配从目标字符串的当前位置开始，可以使用y标志。 |

**y 修饰符**

除了u修饰符，ES6 还为正则表达式添加了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。

var s = 'aaa\_aa\_a';

var r1 = /a+/g;

var r2 = /a+/y;

r1.exec(s) // ["aaa"]

r2.exec(s) // ["aaa"]

r1.exec(s) // ["aa"]

r2.exec(s) // null

上面代码有两个正则表达式，一个使用g修饰符，另一个使用y修饰符。这两个正则表达式各执行了两次，第一次执行的时候，两者行为相同，剩余字符串都是\_aa\_a。由于g修饰没有位置要求，所以第二次执行会返回结果，而y修饰符要求匹配必须从头部开始，所以返回null。

如果改一下正则表达式，保证每次都能头部匹配，y修饰符就会返回结果了。

var s = 'aaa\_aa\_a';

var r = /a+\_/y;

r.exec(s) // ["aaa\_"]

r.exec(s) // ["aa\_"]

上面代码每次匹配，都是从剩余字符串的头部开始。

使用lastIndex属性，可以更好地说明y修饰符。

const REGEX = /a/g;

// 指定从2号位置（y）开始匹配

REGEX.lastIndex = 2;

// 匹配成功

const match = REGEX.exec('xaya');

// 在3号位置匹配成功

match.index // 3

// 下一次匹配从4号位开始

REGEX.lastIndex // 4

// 4号位开始匹配失败

REGEX.exec('xaya') // null

上面代码中，lastIndex属性指定每次搜索的开始位置，g修饰符从这个位置开始向后搜索，直到发现匹配为止。

y修饰符同样遵守lastIndex属性，但是要求必须在lastIndex指定的位置发现匹配。

const REGEX = /a/y;

// 指定从2号位置开始匹配

REGEX.lastIndex = 2;

// 不是粘连，匹配失败

REGEX.exec('xaya') // null

// 指定从3号位置开始匹配

REGEX.lastIndex = 3;

// 3号位置是粘连，匹配成功

const match = REGEX.exec('xaya');

match.index // 3

REGEX.lastIndex // 4

实际上，y修饰符号隐含了头部匹配的标志^。

/b/y.exec('aba')

// null

上面代码由于不能保证头部匹配，所以返回null。y修饰符的设计本意，就是让头部匹配的标志^在全局匹配中都有效。

下面是字符串对象的replace方法的例子。

const REGEX = /a/gy;

'aaxa'.replace(REGEX, '-') // '--xa'

上面代码中，最后一个a因为不是出现在下一次匹配的头部，所以不会被替换。

单单一个y修饰符对match方法，只能返回第一个匹配，必须与g修饰符联用，才能返回所有匹配。

'a1a2a3'.match(/a\d/y) // ["a1"]

'a1a2a3'.match(/a\d/gy) // ["a1", "a2", "a3"]

y修饰符的一个应用，是从字符串提取 token（词元），y修饰符确保了匹配之间不会有漏掉的字符。

const TOKEN\_Y = /\s\*(\+|[0-9]+)\s\*/y;

const TOKEN\_G = /\s\*(\+|[0-9]+)\s\*/g;

tokenize(TOKEN\_Y, '3 + 4')

// [ '3', '+', '4' ]

tokenize(TOKEN\_G, '3 + 4')

// [ '3', '+', '4' ]

function tokenize(TOKEN\_REGEX, str) {

let result = [];

let match;

while (match = TOKEN\_REGEX.exec(str)) {

result.push(match[1]);

}

return result;

}

上面代码中，如果字符串里面没有非法字符，y修饰符与g修饰符的提取结果是一样的。但是，一旦出现非法字符，两者的行为就不一样了。

tokenize(TOKEN\_Y, '3x + 4')

// [ '3' ]

tokenize(TOKEN\_G, '3x + 4')

// [ '3', '+', '4' ]

上面代码中，g修饰符会忽略非法字符，而y修饰符不会，这样就很容易发现错误。

一旦 JavaScript 元素被保存为标准的对象属性，通过[for...in](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...in) 循环来迭代数组将变得不明智，因为正常元素和所有可枚举的属性都会被列出。

数组的方法(array methods)

[in操作符](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/in)，如果所指定的**属性**确实存在于所指定的对象中，则会返回true，语法如下：

对于数组来说，in 操作符取的是元素的下标，不是元素本身。

[concat()](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Array/concat) 连接两个数组并返回一个新的数组。

var myArray = new Array("1", "2", "3");

myArray = myArray.concat("a", "b", "c");

// myArray is now ["1", "2", "3", "a", "b", "c"]

[join(deliminator = ',')](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Array/join) 将数组的所有元素连接成一个字符串。

var myArray = new Array("Wind", "Rain", "Fire");

var list = myArray.join(" - "); // list is "Wind - Rain - Fire"

[slice(start\_index, upto\_index)](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Array/slice) 从数组提取一个片段，并作为一个新数组返回。

var myArray = new Array ("a", "b", "c", "d", "e");

myArray = myArray.slice(1, 4); // starts at index 1 and extracts all elements

// until index 3, returning [ "b", "c", "d"]

含头不含尾

[splice(index, count\_to\_remove, addElement1, addElement2, ...)](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Array/splice)从数组移出一些元素，（可选）并替换它们。

var myArray = new Array ("1", "2", "3", "4", "5");

myArray.splice(1, 3, "a", "b", "c", "d");

// myArray is now ["1", "a", "b", "c", "d", "5"]

3代表移除个数 如果没有后面的abcd就代表是移除，如果有就是替换,如果3那个位置为0意味着添加元素

[indexOf(searchElement[, fromIndex])](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Array/indexOf) 在数组中搜索searchElement 并返回第一个匹配的索引。

[lastIndexOf(searchElement[, fromIndex])](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Array/lastIndexOf) 和 indexOf 差不多，但这是从结尾开始，并且是反向搜索。

[forEach(callback[, thisObject])](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach) 在数组每个元素项上执行callback。

参数

**callback**

为数组中每个元素执行的函数，该函数接收三个参数：

**currentValue**

数组中正在处理的当前元素。

**index**可选

数组中正在处理的当前元素的索引。

**array**可选

forEach() 方法正在操作的数组。

**thisArg**可选

可选参数。当执行回调函数时用作 this 的值(参考对象)。

forEach 遍历的范围在第一次调用 callback 前就会确定。调用 forEach 后添加到数组中的项不会被 callback 访问到。如果已经存在的值被改变，则传递给 callback 的值是 forEach 遍历到他们那一刻的值。已删除的项不会被遍历到。如果已访问的元素在迭代时被删除了（例如使用 [shift()](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Array/shift)），之后的元素将被跳过 - 参见下面的示例。

forEach() 为每个数组元素执行callback函数；不像 [map()](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Array/map) 或者 [reduce()](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Array/reduce)，它总是返回 [undefined](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/undefined) 值，并且不可链式调用。典型用例是在一个链的最后执行副作用。

**注意：** 没有办法中止或者跳出 forEach() 循环，除了抛出一个异常。如果你需要这样，使用 forEach() 方法是错误的。

如果数组在迭代时被修改了，则其他元素会被跳过。

下面的例子会输出"one", "two", "four"。当到达包含值"two"的项时，整个数组的第一个项被移除了，这导致所有剩下的项上移一个位置。因为元素 "four"现在在数组更前的位置，"three"会被跳过。 forEach()不会在迭代之前创建数组的副本。

var words = ['one', 'two', 'three', 'four'];

words.forEach(function(word) {

console.log(word);

if (word === 'two') {

words.shift();

}

});

// one

// two

// four

## 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。如果为负值，表示倒数。
* 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。

[1, 2, 3, 4, 5].copyWithin(0, 3)

// [4, 5, 3, 4, 5]

## 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。

[1, 4, -5, 10].find((n) => n < 0)

// -5

上面代码找出数组中第一个小于 0 的成员。

[1, 5, 10, 15].find(function(value, index, arr) {

return value > 9;

}) // 10

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，如果所有成员都不符合条件，则返回-1。

[1, 5, 10, 15].findIndex(function(value, index, arr) {

return value > 9;

}) // 2

这两个方法都可以接受第二个参数，用来绑定回调函数的this对象。

function f(v){

return v > this.age;

}

let person = {name: 'John', age: 20};

[10, 12, 26, 15].find(f, person); // 26

上面的代码中，find函数接收了第二个参数person对象，回调函数中的this对象指向person对象。

另外，这两个方法都可以发现NaN，弥补了数组的indexOf方法的不足。

[NaN].indexOf(NaN)

// -1

[NaN].findIndex(y => Object.is(NaN, y))

// 0

上面代码中，indexOf方法无法识别数组的NaN成员，但是findIndex方法可以借助Object.is方法做到。

## 数组实例的 fill() [§](http://es6.ruanyifeng.com/#docs/array#%E6%95%B0%E7%BB%84%E5%AE%9E%E4%BE%8B%E7%9A%84-fill) [⇧](http://es6.ruanyifeng.com/#docs/array)

fill方法使用给定值，填充一个数组。

['a', 'b', 'c'].fill(7)

// [7, 7, 7]

new Array(3).fill(7)

// [7, 7, 7]

上面代码表明，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。

['a', 'b', 'c'].fill(7, 1, 2)

// ['a', 7, 'c']

上面代码表示，fill方法从 1 号位开始，向原数组填充 7，到 2 号位之前结束。

注意，如果填充的类型为对象，那么被赋值的是同一个内存地址的对象，而不是深拷贝对象。

let arr = new Array(3).fill({name: "Mike"});

arr[0].name = "Ben";

arr

//

## 数组实例的 entries()，keys() 和 values()

它们都返回一个遍历器对象（详见《Iterator》一章），可以用for...of循环进行遍历

## 数组实例的 flat()，flatMap()

数组的成员有时还是数组，Array.prototype.flat()用于将嵌套的数组“拉平”，变成一维的数组。该方法返回一个新数组，对原数据没有影响。

[1, 2, [3, 4]].flat()

// [1, 2, 3, 4]

如果不管有多少层嵌套，都要转成一维数组，可以用Infinity关键字作为参数。

[1, [2, [3]]].flat(Infinity)

// [1, 2, 3]

如果原数组有空位，flat()方法会跳过空位。

[1, 2, , 4, 5].flat()

flatMap()方法对原数组的每个成员执行一个函数（相当于执行Array.prototype.map()），然后对返回值组成的数组执行flat()方法。该方法返回一个新数组，不改变原数组。

// 相当于 [[2, 4], [3, 6], [4, 8]].flat()

[2, 3, 4].flatMap((x) => [x, x \* 2])

// [2, 4, 3, 6, 4, 8]

flatMap()只能展开一层数组。

// 相当于 [[[2]], [[4]], [[6]], [[8]]].flat()

[1, 2, 3, 4].flatMap(x => [[x \* 2]])

// [[2], [4], [6], [8]]

上面代码中，遍历函数返回的是一个双层的数组，但是默认只能展开一层，因此flatMap()返回的还是一个嵌套数组。

flatMap()方法的参数是一个遍历函数，该函数可以接受三个参数，分别是当前数组成员、当前数组成员的位置（从零开始）、原数组。

arr.flatMap(function callback(currentValue[, index[, array]]) {

// ...

}[, thisArg])

flatMap()方法还可以有第二个参数，用来绑定遍历函数里面的this。

map 不修改调用它的原数组本身（当然可以在 callback 执行时改变原数组）。

使用 map 方法处理数组时，数组元素的范围是在 callback 方法第一次调用之前就已经确定了。在 map 方法执行的过程中：原数组中新增加的元素将不会被 callback 访问到；若已经存在的元素被改变或删除了，则它们的传递到 callback 的值是 map 方法遍历到它们的那一时刻的值；而被删除的元素将不会被访问到。

求数组中每个元素的平方根

下面的代码创建了一个新数组，值为原数组中对应数字的平方根。

var numbers = [1, 4, 9];

var roots = numbers.map(Math.sqrt);

// roots的值为[1, 2, 3], numbers的值仍为[1, 4, 9]

[filter(callback[, thisObject])](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Array/filter) 返回一个包含所有在回调函数上返回为true的元素的新数组（译者注：callback在这里担任的是过滤器的角色，当元素符合条件，过滤器就返回true，而filter则会返回所有符合过滤条件的元素）。

filter 为数组中的每个元素调用一次 callback 函数，并利用所有使得 callback 返回 true 或 [等价于 true 的值](https://developer.mozilla.org/en-US/docs/Glossary/Truthy) 的元素创建一个新数组。callback 只会在已经赋值的索引上被调用，对于那些已经被删除或者从未被赋值的索引不会被调用。那些没有通过 callback 测试的元素会被跳过，不会被包含在新数组中。

使用for循环时，与map,foreach等的区别，

var arr = [5,6,7]

// for(var i = 0; i<arr.length;i++){

// arr.push(9)

// console.log('rr')

// if(arr.length==100){

// break

// }

// }

arr.forEach(function(v){

arr.push(9)

console.log('rr')

if(arr.length==100){

return

}

})

通过for循环处理的，arr的长度会不断变化，并且被for循环更新继续使用，而通过map，foreach遍历的是固定的

[reduce(callback[, initialValue])](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Array/reduce) 使用回调函数 callback(firstValue, secondValue) 把数组列表计算成一个单一值（译者注：他数组元素两两递归处理的方式把数组计算成一个值

[reduceRight(callback[, initalvalue])](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Array/reduceRight) 和 reduce()相似，但这从最后一个元素开始的。

reduce 和 reduceRight 是迭代数组方法中最不被人熟知的两个函数.。他们应该使用在那些需要把数组的元素两两递归处理，并最终计算成一个单一结果的算法。

reduce为数组中的每一个元素依次执行callback函数，不包括数组中被删除或从未被赋值的元素，接受四个参数：

* accumulator 累计器
* currentValue 当前值
* currentIndex 当前索引
* array 数组

回调函数第一次执行时，accumulator 和currentValue的取值有两种情况：如果调用reduce()时提供了initialValue，accumulator取值为initialValue，currentValue取数组中的第一个值；如果没有提供 initialValue，那么accumulator取数组中的第一个值，currentValue取数组中的第二个值。

将二维数组转化为一维

var flattened = [[0, 1], [2, 3], [4, 5]].reduce(

function(a, b) {

return a.concat(b);

},

[]

);

// flattened is [0, 1, 2, 3, 4, 5]

计算数组中每个元素出现的次数

var names = ['Alice', 'Bob', 'Tiff', 'Bruce', 'Alice'];

var countedNames = names.reduce(function (allNames, name) {

if (name in allNames) {

allNames[name]++;

}

else {

allNames[name] = 1;

}

return allNames;

}, {});

// countedNames is:

// { 'Alice': 2, 'Bob': 1, 'Tiff': 1, 'Bruce': 1 }

按属性对object分类

var people = [

{ name: 'Alice', age: 21 },

{ name: 'Max', age: 20 },

{ name: 'Jane', age: 20 }

];

function groupBy(objectArray, property) {

return objectArray.reduce(function (acc, obj) {

var key = obj[property];

if (!acc[key]) {

acc[key] = [];

}

acc[key].push(obj);

return acc;

}, {});

}

var groupedPeople = groupBy(people, 'age');

// groupedPeople is:

// {

// 20: [

// { name: 'Max', age: 20 },

// { name: 'Jane', age: 20 }

// ],

// 21: [{ name: 'Alice', age: 21 }]

// }

使用扩展运算符和initialValue绑定包含在对象数组中的数组

// friends - 对象数组

// where object field "books" - list of favorite books

var friends = [{

name: 'Anna',

books: ['Bible', 'Harry Potter'],

age: 21

}, {

name: 'Bob',

books: ['War and peace', 'Romeo and Juliet'],

age: 26

}, {

name: 'Alice',

books: ['The Lord of the Rings', 'The Shining'],

age: 18

}];

// allbooks - list which will contain all friends' books +

// additional list contained in initialValue

var allbooks = friends.reduce(function(prev, curr) {

return [...prev, ...curr.books];

}, ['Alphabet']);

// allbooks = [

// 'Alphabet', 'Bible', 'Harry Potter', 'War and peace',

// 'Romeo and Juliet', 'The Lord of the Rings',

// 'The Shining'

// ]

数组去重

let arr = [1,2,1,2,3,5,4,5,3,4,4,4,4];

let result = arr.sort().reduce((init, current)=>{

if(init.length===0 || init[init.length-1]!==current){

init.push(current);

}

return init;

}, []);

console.log(result); //[1,2,3,4,5]

按顺序运行Promise

/\*\*

\* Runs promises from array of functions that can return promises

\* in chained manner

\*

\* @param {array} arr - promise arr

\* @return {Object} promise object

\*/

function runPromiseInSequence(arr, input) {

return arr.reduce(

(promiseChain, currentFunction) => promiseChain.then(currentFunction),

Promise.resolve(input)

);

}

// promise function 1

function p1(a) {

return new Promise((resolve, reject) => {

resolve(a \* 5);

});

}

// promise function 2

function p2(a) {

return new Promise((resolve, reject) => {

resolve(a \* 2);

});

}

// function 3 - will be wrapped in a resolved promise by .then()

function f3(a) {

return a \* 3;

}

// promise function 4

function p4(a) {

return new Promise((resolve, reject) => {

resolve(a \* 4);

});

}

const promiseArr = [p1, p2, f3, p4];

runPromiseInSequence(promiseArr, 10)

.then(console.log); // 1200

缓冲区和视图：类型化的数组结构

为了实现最大的灵活性和效率，JavaScript类型数组被分解为缓冲(Buffer)和视图(views)。缓冲(由[ArrayBuffer](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/ArrayBuffer) 实现)是代表数据块的对象，它没有格式可言，并没有提供任何机制来访问其内容。为了访问包含在缓冲区中的内存，您需要使用视图。视图提供了一个上下文，即数据类型、起始偏移量和元素数，这些元素将数据转换为实际类型数组。

ArrayBuffer

[ArrayBuffer](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/ArrayBuffer)是一种数据类型，用于表示一个通用的、固定长度的二进制数据缓冲区。你不能直接操纵一个ArrayBuffer中的内容；你需要创建一个数组类型视图或[DataView](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/DataView)来代表特定格式的缓冲区，并从而实现读写缓冲区的内容。

**DataView** 视图是一个可以从 [ArrayBuffer](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/ArrayBuffer) 对象中读写多种数值类型的底层接口，在读写时不用考虑平台字节序问题。

类型数组视图(Typed array views)

类型数组视图具有自描述性的名字，并且提供数据类型信息，例如Int8, Uint32, Float64等等。如一个特定类型数组视图Uint8ClampedArray. 它意味着数据元素只包含0到255的整数值。它通常用于[Canvas数据处理](https://developer.mozilla.org/en-US/docs/Web/API/ImageData),例如.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Type** | **Value Range** | **Size in bytes** | **Description** | **Web**  **IDL**  **type** | **Equivalent C type** |
| [Int8Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Int8Array) | -128 to 127 | 1 | 8-bit two's complement signed integer | byte | int8\_t |
| [Uint8Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Uint8Array) | 0 to 255 | 1 | 8-bit unsigned integer | Octet | uint8\_t |
| [Uint8ClampedArray](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Uint8ClampedArray) | 0 to 255 | 1 | 8-bit unsigned integer (clamped) | octet | uint8\_t |
| [Int16Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Int16Array) | -32768 to 32767 | 2 | 16-bit two's complement signed integer | short | int16\_t |
| [Uint16Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Uint16Array) | 0 to 65535 | 2 | 16-bit unsigned integer | Unsigned short | uint16\_t |
| [Int32Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Int32Array) | -2147483648 to 2147483647 | 4 | 32-bit two's complement signed integer | long | int32\_t |
| [Uint32Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Uint32Array) | 0 to 4294967295 | 4 | 32-bit unsigned integer | unsigned long | uint32\_t |
| [Float32Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Float32Array) | 1.2x10-38 to 3.4x1038 | 4 | 32-bit IEEE floating point number ( 7 significant digits e.g. 1.1234567) | unrestricted float | float |
| [Float64Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Float64Array) | 5.0x10-324 to 1.8x10308 | 8 | 64-bit IEEE floating point number (16 significant digits e.g. 1.123...15) | unrestricted double | double |

var buffer = new ArrayBuffer(16);

var int32View = new Int32Array(buffer);

for (var i = 0; i < int32View.length; i++) {

int32View[i] = i \* 2;

}

object和Map的比较

一般地，[objects](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Object)会被用于将字符串类型映射到数值。Object允许设置键值对、根据键获取值、删除键、检测某个键是否存在。而Map具有更多的优势。

* Object的键均为Strings类型，在Map里键可以是任意类型。
* 一个Object的键只能是[字符串](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/String)或者 [Symbols](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Symbol)，但一个 Map 的键可以是**任意值**，包括函数、对象、基本类型。
* 必须手动计算Object的尺寸，但是可以很容易地获取使用Map的尺寸。
* Map的遍历遵循元素的插入顺序。
* Object有原型，所以映射中有一些缺省的键。（可以理解为map = Object.create(null)）。

这三条提示可以帮你决定用Map还是Object：

* 如果键在运行时才能知道，或者所有的键类型相同，所有的值类型相同，那就使用Map。
* 如果需要将原始值存储为键，则使用Map，因为Object将每个键视为字符串，不管它是一个数字值、布尔值还是任何其他原始值。
* 如果需要对个别元素进行操作，使用Object。

[WeakMap](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/WeakMap)对象也是键值对的集合。它的**键必须是对象类型**，值可以是任意类型。它的键被弱保持，也就是说，当其键所指对象没有其他地方引用的时候，它会被GC回收掉。WeakMap提供的接口与Map相同。

与Map对象不同的是，WeakMap的键是不可枚举的。不提供列出其键的方法。列表是否存在取决于垃圾回收器的状态，是不可预知的。

可以在"Why WeakMap?"[WeakMap](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/WeakMap)查看更多信息和示例。

WeakMap对象的一个用例是存储一个对象的私有数据或隐藏实施细节。Nick Fitzgerald 的博文["Hiding Implementation Details with ECMAScript 6 WeakMaps"](http://fitzgeraldnick.com/weblog/53/)提供了一个例子。对象内部的私有数据和方法被存储在WeakMap类型的privates变量中。所有暴露出的原型和情况都是公开的，而其他内容都是外界不可访问的，因为模块并未导出privates对象。

new WeakMap([iterable])

参数

**iterable**

Iterable 是一个数组（2元数组）或者其他可迭代的且其元素是键值对的对象。每个键值对会被加到新的 WeakMap 里。null 会被当做 undefined。

在 JavaScript 里，map API 可以通过在其四个 API 方法中共用两个数组(一个存放键,一个存放值)来实现。给该 map 设置值时会同时将键和值添加到这两个数组的末尾。从而使得键和值的索引在两个数组中相对应。当从该 map 取值的时候，需要遍历所有的键，然后使用此索引从存储值的数组中检索出相应的值。

但这样的实现会有两个很大的缺点，首先 set 和 has 方法都是 O(n) 的时间复杂度( n 是键值对的个数)，因为这两个操作都需要遍历全部整个数组来进行匹配。另外一个缺点是可能会导致内存泄漏：数组持续引用着每个键和值。这种引用使得垃圾回收算法不能回收处理他们，即使没有其他任何引用存在了。

相比之下，原生的WeakMap，持有的是每个键或值对象的“弱引用”，这意味着在没有其他引用存在时垃圾回收能正确进行。原生 WeakMap 的结构是特殊且有效的，其用于映射的 key 只有在其没有被回收时才是有效的。

正由于这样的弱引用，WeakMap 的 key 是不可枚举的 (没有方法能给出所有的 key)。如果key 是可枚举的话，其列表将会受垃圾回收机制的影响，从而得到不确定的结果. 因此,如果你想要这种类型对象的 key 值的列表，你应该使用 [Map](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Map)。

**属性**

**WeakMap.length**

length  属性的值为 0。

[**WeakMap.prototype**](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/WeakMap/prototype)

WeakMap 构造器的原型。 允许添加属性到所有的 WeakMap 对象。

[**WeakMap.prototype.delete(key)**](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/WeakMap/delete)

移除key的关联对象。执行后 WeakMap.prototype.has(key)返回false。

[**WeakMap.prototype.get(key)**](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/WeakMap/get)

返回key关联对象, 或者 undefined(没有key关联对象时)。

[**WeakMap.prototype.has(key)**](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/WeakMap/has)

根据是否有key关联对象返回一个Boolean值。

[**WeakMap.prototype.set(key, value)**](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/WeakMap/set)

在WeakMap中设置一组key关联对象，返回这个 WeakMap对象。

Set对象

[Set](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Set)对象是一组值的集合，这些值是不重复的，可以按照添加顺序来遍历。

数组和集合的转换

可以使用[Array.from](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Array/from)或[展开操作符](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/Spread_operator)来完成集合到数组的转换。同样，Set的构造器接受数组作为参数，可以完成从Array到Set的转换。需要重申的是，Set对象中的值不重复，所以数组转换为集合时，所有重复值将会被删除。

Array.from(mySet);

[...mySet2];

mySet2 = new Set([1,2,3,4]);

Array和Set的对比

一般情况下，在JavaScript中使用数组来存储一组元素，而新的集合对象有这些优势：

* 数组中用于判断元素是否存在的[indexOf](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Array/indexOf) 函数效率低下。
* Set对象允许根据值删除元素，而数组中必须使用基于下标的 splice 方法。
* 数组的indexOf方法无法找到[NaN](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/NaN)值。
* Set对象存储不重复的值，所以不需要手动处理包含重复值的情况。

WeakSet对象

[WeakSet](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/WeakSet)对象是一组对象的集合。WeakSet中的对象不重复且不可枚举。

与[Set](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Set)对象的主要区别有：

* WeakSets中的值必须是对象类型，不可以是别的类型
* WeakSet的“weak”指的是，对集合中的对象，如果不存在其他引用，那么该对象将可被垃圾回收。于是不存在一个当前可用对象组成的列表，所以WeakSets不可枚举

WeakSet的用例很有限，比如使用DOM元素作为键来追踪它们而不必担心内存泄漏。

**Map的键和Set的值的等值判断**

Map的键和Set的值的等值判断都基于[same-value-zero algorithm](https://people.mozilla.org/~jorendorff/es6-draft.html#sec-samevaluezero)：

* 判断使用与===相似的规则。
* -0和+0相等。
* [NaN](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/NaN)与自身相等（与===有所不同）。

**枚举一个对象的所有属性**

从 [ECMAScript 5](https://developer.mozilla.org/zh-CN/docs/JavaScript/ECMAScript_5_support_in_Mozilla) 开始，有三种原生的方法用于列出或枚举对象的属性：

* [for...in](https://developer.mozilla.org/zh-CN/docs/JavaScript/Reference/Statements/for...in) 循环 该方法依次访问一个对象及其原型链中所有可枚举的属性。
* [Object.keys(o)](https://developer.mozilla.org/zh-CN/docs/JavaScript/Reference/Global_Objects/Object/keys) 该方法返回一个对象 o 自身包含（不包括原型中）的所有属性的名称的数组。
* [Object.getOwnPropertyNames(o)](https://developer.mozilla.org/zh-CN/docs/JavaScript/Reference/Global_Objects/Object/getOwnPropertyNames) 该方法返回一个数组，它包含了对象 o 所有拥有的属性（无论是否可枚举）的名称。



使用 Object.create 方法

对象也可以用 [Object.create()](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Object/create) 方法创建。该方法非常有用，因为它允许你为创建的对象选择其原型对象，而不用定义一个构造函数。

**Object.create()**方法创建一个新对象，使用现有的对象来提供新创建的对象的\_\_proto\_\_。

Object.create(proto, [propertiesObject])

**例子**

用 Object.create实现类式继承

下面的例子演示了如何使用Object.create()来实现类式继承。这是一个所有版本JavaScript都支持的单继承。

// Shape - 父类(superclass)

function Shape() {

this.x = 0;

this.y = 0;

}

// 父类的方法

Shape.prototype.move = function(x, y) {

this.x += x;

this.y += y;

console.info('Shape moved.');

};

// Rectangle - 子类(subclass)

function Rectangle() {

Shape.call(this); // call super constructor.可写可不写

}

// 子类续承父类

Rectangle.prototype = Object.create(Shape.prototype);

Rectangle.prototype.constructor = Rectangle;//这句话就算不写也不影响使用，只不过这样让他指向自己好一点

var rect = new Rectangle();

console.log('Is rect an instance of Rectangle?',

rect instanceof Rectangle); // true

console.log('Is rect an instance of Shape?',

rect instanceof Shape); // true

rect.move(1, 1); // Outputs, 'Shape moved.'

如果你希望能继承到多个对象，则可以使用混入的方式。

function MyClass() {

SuperClass.call(this);

OtherSuperClass.call(this);

}

// 继承一个类

MyClass.prototype = Object.create(SuperClass.prototype);

// 混合其它

Object.assign(MyClass.prototype, OtherSuperClass.prototype);

// 重新指定constructor

MyClass.prototype.constructor = MyClass;

MyClass.prototype.myMethod = function() {

// do a thing

};

[Object.assign](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Object/assign) 会把  OtherSuperClass原型上的函数拷贝到 MyClass原型上，使 MyClass 的所有实例都可用 OtherSuperClass 的方法。Object.assign 是在 ES2015 引入的，且可用[polyfilled](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Object/assign#Polyfill)。要支持旧浏览器的话，可用使用 [jQuery.extend()](https://api.jquery.com/jQuery.extend/) 或者 [\_.assign()](https://lodash.com/docs/#assign)。

**Object.assign()** 方法用于将所有可枚举属性的值从一个或多个源对象复制到目标对象。它将返回目标对象。

[String](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/String)类型和 [Symbol](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Symbol) 类型的属性都会被拷贝。注意，Object.assign 不会跳过那些值为 [null](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/null) 或 [undefined](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/undefined) 的源对象。

深拷贝问题

针对深拷贝，需要使用其他办法，因为 Object.assign()拷贝的是属性值。假如源对象的属性值是一个对象的引用，那么它也只指向那个引用

let obj1 = { a: 0 , b: { c: 0}};

let obj2 = Object.assign({}, obj1);

console.log(JSON.stringify(obj2)); // { a: 0, b: { c: 0}}

obj2.a = 2;

obj1.a = 1;

obj2.b.c = 3;

console.log(JSON.stringify(obj1)); // { a: 1, b: { c: 3}}

console.log(JSON.stringify(obj2)); // { a: 2, b: { c: 3}}

// Deep Clone

obj1 = { a: 0 , b: { c: 0}};

let obj3 = JSON.parse(JSON.stringify(obj1));

obj1.a = 4;

obj1.b.c = 4;

console.log(JSON.stringify(obj3)); // { a: 0, b: { c: 0}}

合并对象

const o1 = { a: 1 };

const o2 = { b: 2 };

const o3 = { c: 3 };

const obj = Object.assign(o1, o2, o3);

console.log(obj); // { a: 1, b: 2, c: 3 }

console.log(o1); // { a: 1, b: 2, c: 3 }, 注意目标对象自身也会改变。

拷贝 symbol 类型的属性

const o1 = { a: 1 };

const o2 = { [Symbol('foo')]: 2 };

const obj = Object.assign({}, o1, o2);

console.log(obj); // { a : 1, [Symbol("foo")]: 2 } (cf. bug 1207182 on Firefox)

Object.getOwnPropertySymbols(obj); // [Symbol(foo)]

继承属性和不可枚举属性是不能拷贝的

const obj = Object.create({foo: 1}, { // foo 是个继承属性。

bar: {

value: 2 // bar 是个不可枚举属性。

},

baz: {

value: 3,

enumerable: true // baz 是个自身可枚举属性。

}

});

const copy = Object.assign({}, obj);

console.log(copy); // { baz: 3 }

使用 Object.create 的 propertyObject参数

var o;

// 创建一个原型为null的空对象

o = Object.create(null);

o = {};

// 以字面量方式创建的空对象就相当于:

o = Object.create(Object.prototype);

o = Object.create(Object.prototype, {

// foo会成为所创建对象的数据属性

foo: {

writable:true,

configurable:true,

value: "hello"

},

// bar会成为所创建对象的访问器属性

bar: {

configurable: false,

get: function() { return 10 },

set: function(value) {

console.log("Setting `o.bar` to", value);

}

}

});

function Constructor(){}

o = new Constructor();

// 上面的一句就相当于:

o = Object.create(Constructor.prototype);

// 当然,如果在Constructor函数中有一些初始化代码,Object.create不能执行那些代码

// 创建一个以另一个空对象为原型,且拥有一个属性p的对象

o = Object.create({}, { p: { value: 42 } })

// 省略了的属性特性默认为false,所以属性p是不可写,不可枚举,不可配置的:

o.p = 24

o.p

//42

o.q = 12

for (var prop in o) {

console.log(prop) 从这方面看propertyObject有些鸡肋，因为还要设置属性特性

}

//"q"

delete o.p

//false

//创建一个可写的,可枚举的,可配置的属性p

o2 = Object.create({}, {

p: {

value: 42,

writable: true,

enumerable: true,

configurable: true

}

});

**继承**

所有的 JavaScript 对象继承于至少一个对象。被继承的对象被称作原型，并且继承的属性可通过构造函数的 prototype 对象找到。查看更多详细 [Inheritance and the prototype chain](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Inheritance_and_the_prototype_chain)

**继承与原型链**

https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Inheritance\_and\_the\_prototype\_chain

当谈到继承时，JavaScript 只有一种结构：对象。每个实例对象（object ）都有一个私有属性（称之为\_\_proto\_\_）指向它的原型对象（**prototype**）。该原型对象也有一个自己的原型对象(\_\_proto\_\_) ，层层向上直到一个对象的原型对象为 null。根据定义，null 没有原型，并作为这个**原型链**中的最后一个环节。

几乎所有 JavaScript 中的对象都是位于原型链顶端的[Object](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Object)的实例

遵循ECMAScript标准，someObject.[[Prototype]] 符号是用于指向 someObject的原型。从 ECMAScript 6 开始，[[Prototype]] 可以通过[Object.getPrototypeOf()](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Object/getPrototypeOf)和[Object.setPrototypeOf()](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Object/setPrototypeOf)访问器来访问。这个等同于 JavaScript 的非标准但许多浏览器实现的属性 \_\_proto\_\_。

但它不应该与构造函数 func 的 prototype 属性相混淆。被构造函数创建的实例对象的 [[prototype]] 指向 func 的 prototype 属性。**Object.prototype**属性表示[Object](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Object)的原型对象。

var obj = {}

obj.\_\_proto\_\_=== obj.constructor.prototype obj.constructor返回obj的构造函数

**示例**

B 继承自 A：

function A(a){

this.varA = a;

}

// 以上函数 A 的定义中，既然 A.prototype.varA 总是会被 this.varA 遮蔽，

// 那么将 varA 加入到原型（prototype）中的目的是什么？

A.prototype = {

varA : null,

/\*

既然它没有任何作用，干嘛不将 varA 从原型（prototype）去掉 ?

也许作为一种在隐藏类中优化分配空间的考虑 ?

https://developers.google.com/speed/articles/optimizing-javascript

如果varA并不是在每个实例中都被初始化，那这样做将是有效果的。

\*/

doSomething : function(){

// ...

}

}

function B(a, b){

A.call(this, a);

this.varB = b;

}

B.prototype = Object.create(A.prototype, {

varB : {

value: null,

enumerable: true,

configurable: true,

writable: true

},

doSomething : {

value: function(){ // override

A.prototype.doSomething.apply(this, arguments);

  // call super

// ...

},

enumerable: true,

configurable: true,

writable: true

}

});

B.prototype.constructor = B;

var b = new B();

b.doSomething();

最重要的部分是：

* 类型被定义在 .prototype 中
* 用 Object.create() 来继承

**prototype和Object.getPrototypeOf**

对于从 Java 或 C++ 转过来的开发人员来说 JavaScript 会有点让人困惑，因为它全部都是动态的，都是运行时，而且不存在类（classes）。所有的都是实例（对象）。即使我们模拟出的 “类（classes）”，也只是一个函数对象。

你可能已经注意到我们的 function A 有一个叫做 prototype 的特殊属性。该特殊属性可与 JavaScript 的 new 操作符一起使用。对原型对象的引用被复制到新实例的内部 [[Prototype]] 属性。例如，当执行var a1 = new A()时，JavaScript（在内存中创建对象之后，和在运行函数 A() 把 this 指向对象之前）设置a1.[[Prototype]] = A.prototype。然后当您访问实例的属性时，JavaScript首先会检查它们是否直接存在于该对象上，如果不存在，则会[[Prototype]]中查找。这意味着你在prototype中定义的所有内容都可以由所有实例有效共享，你甚至可以稍后更改部分prototype，并在所有现有实例中显示更改（如果需要）。

像上面的例子中，如果你执行var a1 = new A(); var a2 = new A(); 那么 a1.doSomething事实上会指向Object.getPrototypeOf(a1).doSomething，它就是你在 A.prototype.doSomething 中定义的内容。也就是说：Object.getPrototypeOf(a1).doSomething == Object.getPrototypeOf(a2).doSomething == A.prototype.doSomething。【补充：实际上当执行a1.doSomething()相当于执行Object.getPrototypeOf(a1).doSomething.call(a1)==A.prototype.doSomething.call(a1)】

简而言之， prototype是用于类的，而 Object.getPrototypeOf() 是用于实例的（instances），两者功能一致。

[**创建层级结构**](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Details_of_the_Object_Model#%E5%88%9B%E5%BB%BA%E5%B1%82%E7%BA%A7%E7%BB%93%E6%9E%84)

function Employee () {

this.name = "";

this.dept = "general";

}

function Manager() {

Employee.call(this);

this.reports = [];

}

Manager.prototype = Object.create(Employee.prototype);

//这样写也行，只不过Manager.prototype.\_\_proto\_\_不指向Employee.prototype了，但是也一样可以使用

Manager.prototype = Employee.prototype;

function WorkerBee() {

Employee.call(this);

this.projects = [];

}

WorkerBee.prototype = Object.create(Employee.prototype);

**Object.defineProperty()** 方法会直接在一个对象上定义一个新属性，或者修改一个对象的现有属性， 并返回这个对象。

**数据描述符和存取描述符均具有**以下可选键值：

**configurable**

当且仅当该属性的 configurable 为 true 时，该属性描述符才能够被改变，同时该属性也能从对应的对象上被删除。**默认为 false**。

**enumerable**

当且仅当该属性的enumerable为true时，该属性才能够出现在对象的枚举属性中。**默认为 false**。

**数据描述符同时具有以下可选键值**：

**value**

该属性对应的值。可以是任何有效的 JavaScript 值（数值，对象，函数等）。**默认为**[**undefined**](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/undefined)。

**writable**

当且仅当该属性的writable为true时，value才能被[赋值运算符](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/Assignment_Operators)改变。**默认为 false**。

**存取描述符同时具有以下可选键值**：

**get**

一个给属性提供 getter 的方法，如果没有 getter 则为 undefined。当访问该属性时，该方法会被执行，方法执行时没有参数传入，但是会传入this对象（由于继承关系，这里的this并不一定是定义该属性的对象）。

**默认为**[**undefined**](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/undefined)。

**set**

一个给属性提供 setter 的方法，如果没有 setter 则为 undefined。当属性值修改时，触发执行该方法。该方法将接受唯一参数，即该属性新的参数值。

**默认为**[**undefined**](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/undefined)。

创建属性

如果对象中不存在指定的属性，Object.defineProperty()就创建这个属性。当描述符中省略某些字段时，这些字段将使用它们的默认值。拥有布尔值的字段的默认值都是false。value，get和set字段的默认值为undefined。一个没有get/set/value/writable定义的属性被称为“通用的”，并被“键入”为一个数据描述符。

var o = {}; // 创建一个新对象

// 在对象中添加一个属性与数据描述符的示例

Object.defineProperty(o, "a", {

value : 37,

writable : true,

enumerable : true,

configurable : true

});

// 对象o拥有了属性a，值为37

// 在对象中添加一个属性与存取描述符的示例

var bValue;

Object.defineProperty(o, "b", {

get : function(){

return bValue;

},

set : function(newValue){

bValue = newValue;

},

enumerable : true,

configurable : true

});

继承属性

如果访问者的属性是被继承的，它的 get 和set 方法会在子对象的属性被访问或者修改时被调用。如果这些方法用一个变量存值，该值会被所有对象共享。

function myclass() {

}

var value;

Object.defineProperty(myclass.prototype, "x", {

get() {

return value;

},

set(x) {

value = x;

}

});

var a = new myclass();

var b = new myclass();

a.x = 1;

console.log(b.x); // 1

这可以通过将值存储在另一个属性中解决。在 get 和 set 方法中，this 指向某个被访问和修改属性的对象。

function myclass() {

}

Object.defineProperty(myclass.prototype, "x", {

get() {

return this.stored\_x;

},

set(x) {

this.stored\_x = x;

}

});

var a = new myclass();

var b = new myclass();

a.x = 1;

console.log(b.x); // undefined

不像访问者属性，值属性始终在对象自身上设置，而不是一个原型。然而，如果一个不可写的属性被继承，它仍然可以防止修改对象的属性

**定义类**

类实际上是个“特殊的[函数](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Functions)”，就像你能够定义的[函数表达式](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/function)和[函数声明](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Statements/function)一样，类语法有两个组成部分：[类表达式](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/class)和[类声明](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Statements/class)。

静态方法

[static](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Classes/static) 关键字用来定义一个类的一个静态方法。调用静态方法不需要[实例化](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Introduction_to_Object-Oriented_JavaScript#The_object_(class_instance))该类，但不能通过一个类实例调用静态方法。静态方法通常用于为一个应用程序创建工具函数。

用原型和静态方法包装

当一个对象调用静态或原型方法时，如果该对象没有“this”值（或“this”作为布尔，字符串，数字，未定义或null) ，那么“this”值在被调用的函数内部将为 **undefined**。不会发生自动包装。即使我们以非严格模式编写代码，它的行为也是一样的，因为所有的函数、方法、构造函数、getters或setters都在严格模式下执行。因此如果我们没有指定this的值，this值将为**undefined**。

class Animal {

speak() {

return this;

}

static eat() {

return this;

}

}

let obj = new Animal();

obj.speak(); // Animal {}

let speak = obj.speak;

speak(); // undefined

Animal.eat() // class Animal

let eat = Animal.eat;

eat(); // undefined

如果我们使用传统的基于函数的类来编写上述代码，那么基于调用该函数的“this”值将发生自动装箱。

function Animal() { }

Animal.prototype.speak = function() {

return this;

}

Animal.eat = function() {

return this;

}

let obj = new Animal();

let speak = obj.speak;

speak(); // global object

let eat = Animal.eat;

eat(); // global object

请注意，类不能继承常规（非可构造）对象。如果要继承常规对象，可以改用[Object.setPrototypeOf()](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Object/setPrototypeOf)：

**Object.setPrototypeOf()**方法设置一个指定的对象的原型 ( 即, 内部[[Prototype]]属性）到另一个对象或  [null](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/null)。

**警告:** 由于现代 JavaScript 引擎优化属性访问所带来的特性的关系，更改对象的 [[Prototype]]在**各个**浏览器和 JavaScript 引擎上都是一个很慢的操作。其在更改继承的性能上的影响是微妙而又广泛的，这不仅仅限于 obj.\_\_proto\_\_ = ... 语句上的时间花费，而且可能会延伸到**任何**代码，那些可以访问**任何**[[Prototype]]已被更改的对象的代码。如果你关心性能，你应该避免设置一个对象的 [[Prototype]]。相反，你应该使用 [Object.create()](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Object/create)来创建带有你想要的[[Prototype]]的新对象。

**语法**

Object.setPrototypeOf(*obj, prototype*)

参数

**obj**

要设置其原型的对象。.

**prototype**

该对象的新原型(一个对象 或 [null](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/null)).

var Animal = {

speak() {

console.log(this.name + ' makes a noise.');

}

};

class Dog {

constructor(name) {

this.name = name;

}

}

Object.setPrototypeOf(Dog.prototype, Animal);// If you do not do this you will get a TypeError when you invoke speak

var d = new Dog('Mitzie');

d.speak(); // Mitzie makes a noise.

**使用 super 调用超类**

[super](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/super) 关键字用于调用对象的父对象上的函数。

class Cat {

constructor(name) {

this.name = name;

}

speak() {

console.log(this.name + ' makes a noise.');

}

}

class Lion extends Cat {

speak() {

super.speak();

console.log(this.name + ' roars.');

}

}

严格模式：

fun.caller和fun.arguments都是不可删除的属性而且在存值、取值时都会报错：

arguments.callee 是一个不可删除属性，而且赋值和读取时都会抛出异常：

**内存生命周期**

不管什么程序语言，内存生命周期基本是一致的：

1. 分配你所需要的内存
2. 使用分配到的内存（读、写）
3. 不需要时将其释放\归还

标记-清除算法

这个算法把“对象是否不再需要”简化定义为“对象是否可以获得”。

这个算法假定设置一个叫做根（root）的对象（在Javascript里，根是全局对象）。垃圾回收器将定期从根开始，找所有从根开始引用的对象，然后找这些对象引用的对象……从根开始，垃圾回收器将找到所有可以获得的对象和收集所有不能获得的对象。

这个算法比前一个要好，因为“有零引用的对象”总是不可获得的，但是相反却不一定，参考“循环引用”。

https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/EventLoop

函数 [setTimeout](https://developer.mozilla.org/zh-CN/docs/Web/API/WindowOrWorkerGlobalScope/setTimeout) 接受两个参数：待加入队列的消息和一个延迟（可选，默认为 0）。这个延迟代表了消息被实际加入到队列的最小延迟时间。如果队列中没有其它消息，在这段延迟时间过去之后，消息会被马上处理。但是，如果有其它消息，setTimeout 消息必须等待其它消息处理完。因此第二个参数仅仅表示最少延迟时间，而非确切的等待时间。

零延迟并不意味着回调会立即执行。以 0 为第二参数调用 setTimeout 并不表示在 0 毫秒后就立即调用回调函数。

EC6:

JavaScript 内部，字符以 UTF-16 的格式储存，每个字符固定为2个字节。对于那些需要4个字节储存的字符（Unicode 码点大于0xFFFF的字符），JavaScript 会认为它们是两个字符。

**codePointAt()**[**§**](http://es6.ruanyifeng.com/#docs/string#codePointAt)[**⇧**](http://es6.ruanyifeng.com/#docs/string)

JavaScript 内部，字符以 UTF-16 的格式储存，每个字符固定为2个字节。对于那些需要4个字节储存的字符（Unicode 码点大于0xFFFF的字符），JavaScript 会认为它们是两个字符。

var s = "𠮷";

s.length // 2

s.charAt(0) // ''

s.charAt(1) // ''

s.charCodeAt(0) // 55362

s.charCodeAt(1) // 57271

上面代码中，汉字“𠮷”（注意，这个字不是“吉祥”的“吉”）的码点是0x20BB7，UTF-16 编码为0xD842 0xDFB7（十进制为55362 57271），需要4个字节储存。对于这种4个字节的字符，JavaScript 不能正确处理，字符串长度会误判为2，而且charAt方法无法读取整个字符，charCodeAt方法只能分别返回前两个字节和后两个字节的值。

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

let s = '𠮷a';

s.length= 3也是鸡肋，需要借助for—of

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。

'x'.padStart(5, 'ab') // 'ababx'

'x'.padStart(4, 'ab') // 'abax'

'x'.padEnd(5, 'ab') // 'xabab'

'x'.padEnd(4, 'ab') // 'xaba'

如果原字符串的长度，等于或大于最大长度，则字符串补全不生效，返回原字符串。

'xxx'.padStart(2, 'ab') // 'xxx'

'xxx'.padEnd(2, 'ab') // 'xxx'

## 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。

/𠮷{2}/.test('𠮷𠮷') // false

/𠮷{2}/u.test('𠮷𠮷') // true

**s 修饰符：dotAll 模式**

ES2018 引入了[具名组匹配](https://github.com/tc39/proposal-regexp-named-groups)（Named Capture 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。

const RE\_DATE = /(?<year>\d{4})-(?<month>\d{2})-(?<day>\d{2})/;

const matchObj = RE\_DATE.exec('1999-12-31');

const year = matchObj.groups.year; // 1999

const month = matchObj.groups.month; // 12

const day = matchObj.groups.day; // 31

如果具名组没有匹配，那么对应的groups对象属性会是undefined。

let {groups: {one, two}} = /^(?<one>.\*):(?<two>.\*)$/u.exec('foo:bar');

one // foo

two // bar

**Math.trunc()**

Math.trunc方法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。

Math.trunc('123.456') // 123

Math.trunc(true) //1

Math.trunc(false) // 0

Math.trunc(null) // 0

**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。

### Math.cbrt()

Math.cbrt方法用于计算一个数的立方根。

Math.hypot() [§](http://es6.ruanyifeng.com/#docs/number#Math-hypot) [⇧](http://es6.ruanyifeng.com/#docs/number)

Math.hypot方法返回所有参数的平方和的平方根。

## 指数运算符

ES2016 新增了一个指数运算符（\*\*）。

2 \*\* 2 // 4

2 \*\* 3 // 8

## 方法的 name 属性

const obj = {

get foo() {},

set foo(x) {}

};

obj.foo.name

// TypeError: Cannot read property 'name' of undefined

const descriptor = Object.getOwnPropertyDescriptor(obj, 'foo');

descriptor.get.name // "get foo"

descriptor.set.name // "set foo"

(new Function()).name // "anonymous"

var doSomething = function() {

// ...

};

doSomething.bind().name // "bound doSomething"

const key1 = Symbol('description');

const key2 = Symbol();

let obj = {

[key1]() {},

[key2]() {},

};

obj[key1].name // "[description]"

obj[key2].name // ""

### 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。

## 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。

const proto = {

foo: 'hello'

};

const obj = {

foo: 'world',

find() {

return super.foo;

}

};

Object.setPrototypeOf(obj, proto);

obj.find() // "hello"

上面代码中，对象obj.find()方法之中，通过super.foo引用了原型对象proto的foo属性。

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。

JavaScript 引擎内部，super.foo等同于Object.getPrototypeOf(this).foo（属性）或Object.getPrototypeOf(this).foo.call(this)（方法）。

解构赋值的拷贝是浅拷贝，即如果一个键的值是复合类型的值（数组、对象、函数）、那么解构赋值拷贝的是这个值的引用，而不是这个值的副本。

扩展运算符的解构赋值，不能复制继承自原型对象的属性。

### 扩展运算符

对象的扩展运算符（...）用于取出参数对象的所有可遍历属性，拷贝到当前对象之中。

由于数组是特殊的对象，所以对象的扩展运算符也可以用于数组。

let foo = { ...['a', 'b', 'c'] };

foo

// {0: "a", 1: "b", 2: "c"}

如果扩展运算符后面不是对象，则会自动将其转为对象。

// 等同于 {...Object(1)}

{...1} // {}

但是，如果扩展运算符后面是字符串，它会自动转成一个类似数组的对象，因此返回的不是空对象。

{...'hello'}

// {0: "h", 1: "e", 2: "l", 3: "l", 4: "o"}

如果想完整克隆一个对象，还拷贝对象原型的属性，可以采用下面的写法。

// 写法一

const clone1 = {

\_\_proto\_\_: Object.getPrototypeOf(obj),

...obj

};

// 写法二

const clone2 = Object.assign(

Object.create(Object.getPrototypeOf(obj)),

obj

);

// 写法三

const clone3 = Object.create(

Object.getPrototypeOf(obj),

Object.getOwnPropertyDescriptors(obj)

)

这些都是深克隆

## Object.is()

不同之处只有两个：一是+0不等于-0，二是NaN等于自身。

+0===-0 true

NaN===NaN true

1. [**Object.is()**](http://es6.ruanyifeng.com/#docs/object-methods#Object.is())
2. [**Object.assign()**](http://es6.ruanyifeng.com/#docs/object-methods#Object.assign())
3. [**Object.getOwnPropertyDescriptors()**](http://es6.ruanyifeng.com/#docs/object-methods#Object.getOwnPropertyDescriptors())
4. [**\_\_proto\_\_属性，Object.setPrototypeOf()，Object.getPrototypeOf()**](http://es6.ruanyifeng.com/#docs/object-methods#__proto__%E5%B1%9E%E6%80%A7%EF%BC%8CObject.setPrototypeOf()%EF%BC%8CObject.getPrototypeOf())
5. [**Object.keys()，Object.values()，Object.entries()**](http://es6.ruanyifeng.com/#docs/object-methods#Object.keys()%EF%BC%8CObject.values()%EF%BC%8CObject.entries())
6. [**Object.fromEntries()**](http://es6.ruanyifeng.com/#docs/object-methods#Object.fromEntries())

## Map

const map = new Map([

['name', '张三'],

['title', 'Author']

]);

注意这种格式

事实上，不仅仅是数组，任何具有 Iterator 接口、且每个成员都是一个双元素的数组的数据结构（详见《Iterator》一章）都可以当作Map构造函数的参数

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。

const map = new Map();

map.set(['a'], 555);

map.get(['a']) // undefined

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。

由上可知，Map 的键实际上是跟内存地址绑定的，只要内存地址不一样，就视为两个键。这就解决了同名属性碰撞（clash）的问题，我们扩展别人的库的时候，如果使用对象作为键名，就不用担心自己的属性与原作者的属性同名。

Set的遍历顺序就是插入顺序。这个特性有时非常有用，比如使用 Set 保存一个回调函数列表，调用时就能保证按照添加顺序调用。

**（1）**keys()**，**values()**，**entries()

keys方法、values方法、entries方法返回的都是遍历器对象（详见《Iterator 对象》一章）。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。

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

一种数据结构只要部署了 Iterator 接口，我们就称这种数据结构是“可遍历的”（iterable）

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。至于属性名Symbol.iterator，它是一个表达式，返回Symbol对象的iterator属性，这是一个预定义好的、类型为 Symbol 的特殊值，所以要放在方括号内（参见《Symbol》一章）。

ES6 的有些数据结构原生具备 Iterator 接口（比如数组），即不用任何处理，就可以被for...of循环遍历。原因在于，这些数据结构原生部署了Symbol.iterator属性（详见下文），另外一些数据结构没有（比如对象）。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。

原生具备 Iterator 接口的数据结构如下。

* Array
* Map
* Set
* String
* TypedArray
* 函数的 arguments 对象
* NodeList 对象

具有Symbol.iterator这个属性的对象都具有可遍历性，它可以手动添加或者原生生成，或者用generator函数生成。

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。

总结一下，调用 Generator 函数，返回一个遍历器对象，代表 Generator 函数的内部指针。以后，每次调用遍历器对象的next方法，就会返回一个有着value和done两个属性的对象。value属性表示当前的内部状态的值，是yield表达式后面那个表达式的值；done属性是一个布尔值，表示是否遍历结束。

### yield 表达式 [§](http://es6.ruanyifeng.com/#docs/generator#yield-%E8%A1%A8%E8%BE%BE%E5%BC%8F) [⇧](http://es6.ruanyifeng.com/#docs/generator)

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表达式就是暂停标志。

遍历器对象的next方法的运行逻辑如下。

（1）遇到yield表达式，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表达式。

（3）如果没有再遇到新的yield表达式，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。

需要注意的是，yield表达式后面的表达式，只有当调用next方法、内部指针指向该语句时才会执行，因此等于为 JavaScript 提供了手动的“惰性求值”（Lazy Evaluation）的语法功能。

Generator 函数可以不用yield表达式，这时就变成了一个单纯的暂缓执行函数。

### 与 Iterator 接口的关系 [§](http://es6.ruanyifeng.com/#docs/generator#%E4%B8%8E-Iterator-%E6%8E%A5%E5%8F%A3%E7%9A%84%E5%85%B3%E7%B3%BB) [⇧](http://es6.ruanyifeng.com/#docs/generator)

上一章说过，任意一个对象的Symbol.iterator方法，等于该对象的遍历器生成函数，调用该函数会返回该对象的一个遍历器对象。

由于 Generator 函数就是遍历器生成函数，因此可以把 Generator 赋值给对象的Symbol.iterator属性，从而使得该对象具有 Iterator 接口。

var myIterable = {};

myIterable[Symbol.iterator] = function\* () {

yield 1;

yield 2;

yield 3;

};

[...myIterable] // [1, 2, 3]

上面代码中，Generator 函数赋值给Symbol.iterator属性，从而使得myIterable对象具有了 Iterator 接口，可以被...运算符遍历了。

Generator 函数执行后，返回一个遍历器对象。该对象本身也具有Symbol.iterator属性，执行后返回自身。

function\* gen(){

// some code

}

var g = gen();

g[Symbol.iterator]() === g

// true

上面代码中，gen是一个 Generator 函数，调用它会生成一个遍历器对象g。它的Symbol.iterator属性，也是一个遍历器对象生成函数，执行后返回它自己。

## next 方法的参数

yield表达式本身没有返回值，或者说总是返回undefined。next方法可以带一个参数，该参数就会被当作上一个yield表达式的返回值。

这个功能有很重要的语法意义。Generator 函数从暂停状态到恢复运行，它的上下文状态（context）是不变的。通过next方法的参数，就有办法在 Generator 函数开始运行之后，继续向函数体内部注入值。也就是说，可以在 Generator 函数运行的不同阶段，从外部向内部注入不同的值，从而调整函数行为。

## yield\* 表达式

如果在 Generator 函数内部，调用另一个 Generator 函数，默认情况下是没有效果的。

## Generator 函数的this

Generator 函数总是返回一个遍历器，ES6 规定这个遍历器是 Generator 函数的实例，也继承了 Generator 函数的prototype对象上的方法。

function\* g() {}

g.prototype.hello = function () {

return 'hi!';

};

let obj = g();

obj instanceof g // true

obj.hello() // 'hi!'

如果把g当作普通的构造函数，并不会生效，因为g返回的总是遍历器对象，而不是this对象。

function\* g() {

this.a = 11;

}

let obj = g();

obj.next();

obj.a // undefined

### 部署 Iterator 接口 [§](http://es6.ruanyifeng.com/#docs/generator#%EF%BC%883%EF%BC%89%E9%83%A8%E7%BD%B2-Iterator-%E6%8E%A5%E5%8F%A3) [⇧](http://es6.ruanyifeng.com/#docs/generator)

利用 Generator 函数，可以在任意对象上部署 Iterator 接口。

function\* iterEntries(obj) {

let keys = Object.keys(obj);

for (let i=0; i < keys.length; i++) {

let key = keys[i];

yield [key, obj[key]];

}

}

let myObj = { foo: 3, bar: 7 };

for (let [key, value] of iterEntries(myObj)) {

console.log(key, value);

}

// foo 3

// bar 7

上述代码中，myObj是一个普通对象，通过iterEntries函数，就有了 Iterator 接口。也就是说，可以在任意对象上部署next方法。

# Generator 函数的异步应用

Promise 的写法只是回调函数的改进，使用then方法以后，异步任务的两段执行看得更清楚了，除此以外，并无新意。

Promise 的最大问题是代码冗余，原来的任务被 Promise 包装了一下，不管什么操作，一眼看去都是一堆then，原来的语义变得很不清楚。

也是鸡肋，需要借助thunk 和 co模块配合使用，异步还是用promise吧

const p1 = new Promise(function (resolve, reject) {

setTimeout(() => reject(new Error('fail')), 3000)

})

const p2 = new Promise(function (resolve, reject) {

setTimeout(() => resolve(p1), 1000)

})

p2

.then(result => console.log(result))

.catch(error => console.log(error))

// Error: fail

上面代码中，p1是一个 Promise，3 秒之后变为rejected。p2的状态在 1 秒之后改变，resolve方法返回的是p1。由于p2返回的是另一个 Promise，导致p2自己的状态无效了，由p1的状态决定p2的状态。所以，后面的then语句都变成针对后者（p1）。又过了 2 秒，p1变为rejected，导致触发catch方法指定的回调函数。

## Promise.prototype.then() [§](http://es6.ruanyifeng.com/#docs/promise#Promise-prototype-then) [⇧](http://es6.ruanyifeng.com/#docs/promise)

Promise 实例具有then方法，也就是说，then方法是定义在原型对象Promise.prototype上的。它的作用是为 Promise 实例添加状态改变时的回调函数。前面说过，then方法的第一个参数是resolved状态的回调函数，第二个参数（可选）是rejected状态的回调函数。

then方法返回的是一个新的Promise实例（注意，不是原来那个Promise实例）。因此可以采用链式写法，即then方法后面再调用另一个then方法。

## Promise.prototype.catch()

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

getJSON('/posts.json').then(function(posts) {

// ...

}).catch(function(error) {

// 处理 getJSON 和 前一个回调函数运行时发生的错误

console.log('发生错误！', error);

});

## Promise.prototype.finally()

finally方法用于指定不管 Promise 对象最后状态如何，都会执行的操作。该方法是 ES2018 引入标准的。

promise

.then(result => {···})

.catch(error => {···})

.finally(() => {···});

上面代码中，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。

## Promise.all() [§](http://es6.ruanyifeng.com/#docs/promise#Promise-all) [⇧](http://es6.ruanyifeng.com/#docs/promise)

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

const p = Promise.all([p1, p2, p3]);

上面代码中，Promise.all方法接受一个数组作为参数，p1、p2、p3都是 Promise 实例，如果不是，就会先调用下面讲到的Promise.resolve方法，将参数转为 Promise 实例，再进一步处理。（Promise.all方法的参数可以不是数组，但必须具有 Iterator 接口，且返回的每个成员都是 Promise 实例。）

p的状态由p1、p2、p3决定，分成两种情况。

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

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

## Promise.race()

Promise.race方法同样是将多个 Promise 实例，包装成一个新的 Promise 实例。

const p = Promise.race([p1, p2, p3]);

上面代码中，只要p1、p2、p3之中有一个实例率先改变状态，p的状态就跟着改变。那个率先改变的 Promise 实例的返回值，就传递给p的回调函数。

Promise.race方法的参数与Promise.all方法一样，如果不是 Promise 实例，就会先调用下面讲到的Promise.resolve方法，将参数转为 Promise 实例，再进一步处理。

## Promise.resolve()

有时需要将现有对象转为 Promise 对象，Promise.resolve方法就起到这个作用。

const jsPromise = Promise.resolve($.ajax('/whatever.json'));

上面代码将 jQuery 生成的deferred对象，转为一个新的 Promise 对象

Promise.resolve方法的参数分成四种情况。

**（1）参数是一个 Promise 实例**

如果参数是 Promise 实例，那么Promise.resolve将不做任何修改、原封不动地返回这个实例。

**（2）参数是一个**thenable**对象**

thenable对象指的是具有then方法的对象，比如下面这个对象。

Promise.resolve方法会将这个对象转为 Promise 对象，然后就立即执行thenable对象的then方法。

**（3）参数不是具有**then**方法的对象，或根本就不是对象**

如果参数是一个原始值，或者是一个不具有then方法的对象，则Promise.resolve方法返回一个新的 Promise 对象，状态为resolved。

**（4）不带有任何参数**

Promise.resolve方法允许调用时不带参数，直接返回一个resolved状态的 Promise 对象。

所以，如果希望得到一个 Promise 对象，比较方便的方法就是直接调用Promise.resolve方法。

## Promise.try() [§](http://es6.ruanyifeng.com/#docs/promise#Promise-try) [⇧](http://es6.ruanyifeng.com/#docs/promise)

实际开发中，经常遇到一种情况：不知道或者不想区分，函数f是同步函数还是异步操作，但是想用 Promise 来处理它。因为这样就可以不管f是否包含异步操作，都用then方法指定下一步流程，用catch方法处理f抛出的错误。一般就会采用下面的写法。

Promise.try(() => database.users.get({id: userId}))

.then(...)

.catch(...)

# async 函数

async 函数是什么？一句话，它就是 Generator 函数的语法糖。

async函数对 Generator 函数的改进，体现在以下四点。

（1）内置执行器。

Generator 函数的执行必须靠执行器，所以才有了co模块，而async函数自带执行器。也就是说，async函数的执行，与普通函数一模一样，只要一行。

（4）返回值是 Promise。

async函数的返回值是 Promise 对象，这比 Generator 函数的返回值是 Iterator 对象方便多了。你可以用then方法指定下一步的操作。

进一步说，async函数完全可以看作多个异步操作，包装成的一个 Promise 对象，而await命令就是内部then命令的语法糖。

**Promise 对象的状态变化**

async函数返回的 Promise 对象，必须等到内部所有await命令后面的 Promise 对象执行完，才会发生状态改变，除非遇到return语句或者抛出错误。也就是说，只有async函数内部的异步操作执行完，才会执行then方法指定的回调函数。

下面是一个例子。

async function getTitle(url) {

let response = await fetch(url);

let html = await response.text();

return html.match(/<title>([\s\S]+)<\/title>/i)[1];

}

getTitle('https://tc39.github.io/ecma262/').then(console.log)

// "ECMAScript 2017 Language Specification"

上面代码中，函数getTitle内部有三个操作：抓取网页、取出文本、匹配页面标题。只有这三个操作全部完成，才会执行then方法里面的console.log。

### await 命令 [§](http://es6.ruanyifeng.com/#docs/async#await-%E5%91%BD%E4%BB%A4) [⇧](http://es6.ruanyifeng.com/#docs/async)

正常情况下，await命令后面是一个 Promise 对象，返回该对象的结果。如果不是 Promise 对象，就直接返回对应的值。

任何一个await语句后面的 Promise 对象变为reject状态，那么整个async函数都会中断执行。

有时，我们希望即使前一个异步操作失败，也不要中断后面的异步操作。这时可以将第一个await放在try...catch结构里面，这样不管这个异步操作是否成功，第二个await都会执行。

async function f() {

try {

await Promise.reject('出错了');

} catch(e) {

}

return await Promise.resolve('hello world');

}

f()

.then(v => console.log(v))

// hello world

另一种方法是await后面的 Promise 对象再跟一个catch方法，处理前面可能出现的错误。

第二点，多个await命令后面的异步操作，如果不存在继发关系，最好让它们同时触发。

let foo = await getFoo();

let bar = await getBar();

上面代码中，getFoo和getBar是两个独立的异步操作（即互不依赖），被写成继发关系。这样比较耗时，因为只有getFoo完成以后，才会执行getBar，完全可以让它们同时触发。

// 写法一

let [foo, bar] = await Promise.all([getFoo(), getBar()]);

// 写法二

let fooPromise = getFoo();

let barPromise = getBar();

let foo = await fooPromise;

let bar = await barPromise;

上面两种写法，getFoo和getBar都是同时触发，这样就会缩短程序的执行时间

async function logInOrder(urls) {

for (const url of urls) {

const response = await fetch(url);

console.log(await response.text());

}

}

上面代码确实大大简化，问题是所有远程操作都是继发。只有前一个 URL 返回结果，才会去读取下一个 URL，这样做效率很差，非常浪费时间。我们需要的是并发发出远程请求。

async function logInOrder(urls) {

// 并发读取远程URL

const textPromises = urls.map(async url => {

const response = await fetch(url);

return response.text();

});

// 按次序输出

for (const textPromise of textPromises) {

console.log(await textPromise);

}

}

上面代码中，虽然map方法的参数是async函数，但它是并发执行的，因为只有async函数内部是继发执行，外部不受影响。后面的for..of循环内部使用了await，因此实现了按顺序输出。

类的所有方法都定义在类的prototype属性上面。

类的内部所有定义的方法，都是不可枚举的, 这一点与 ES5 的行为不一致。

### constructor 方法

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。

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。

实例的属性除非显式定义在其本身（即定义在this对象上），否则都是定义在原型上（即定义在class上）。

采用 Class 表达式，可以写出立即执行的 Class。

let person = new class {

constructor(name) {

this.name = name;

}

sayName() {

console.log(this.name);

}

}('张三');

person.sayName(); // "张三"

## 静态方法 [§](http://es6.ruanyifeng.com/#docs/class#%E9%9D%99%E6%80%81%E6%96%B9%E6%B3%95) [⇧](http://es6.ruanyifeng.com/#docs/class)

类相当于实例的原型，所有在类中定义的方法，都会被实例继承。如果在一个方法前，加上static关键字，就表示该方法不会被实例继承，而是直接通过类来调用，这就称为“静态方法”。

父类的静态方法，可以被子类继承。类的静态方法不会被实例继承

class foo {

bar = 'hello';

baz = 'world';

constructor() {

// ...

}

}

上面的代码，一眼就能看出，foo类有两个实例属性，一目了然。另外，写起来也比较简洁。写在constructor里面要加上this

## 静态属性

静态属性指的是 Class 本身的属性，即Class.propName，而不是定义在实例对象（this）上的属性。

class Foo {

}

Foo.prop = 1;

Foo.prop // 1

上面的写法为Foo类定义了一个静态属性prop。

目前，只有这种写法可行，因为 ES6 明确规定，Class 内部只有静态方法，没有静态属性。现在有一个[提案](https://github.com/tc39/proposal-class-fields)提供了类的静态属性，写法是在实例属性法的前面，加上static关键字。

## new.target 属性

new是从构造函数生成实例对象的命令。ES6 为new命令引入了一个new.target属性，该属性一般用在构造函数之中，返回new命令作用于的那个构造函数。如果构造函数不是通过new命令或Reflect.construct()调用的，new.target会返回undefined，因此这个属性可以用来确定构造函数是怎么调用的。

利用这个特点，可以写出不能独立使用、必须继承后才能使用的类。

## Object.getPrototypeOf() [§](http://es6.ruanyifeng.com/#docs/class-extends#Object-getPrototypeOf) [⇧](http://es6.ruanyifeng.com/#docs/class-extends)

Object.getPrototypeOf方法可以用来从子类上获取父类。

Object.getPrototypeOf(ColorPoint) === Point

// true

因此，可以使用这个方法判断，一个类是否继承了另一个类。

new.target指向当前正在执行的函数

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。

class A {

p() {

return 2;

}

}

class B extends A {

constructor() {

super();

console.log(super.p()); // 2

}

}

let b = new B();

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在普通方法之中，指向A.prototype，所以super.p()就相当于A.prototype.p()。

由于this指向子类实例，所以如果通过super对某个属性赋值，这时super就是this，赋值的属性会变成子类实例的属性。super.x = 3相当于this.x = 3

在子类的静态方法中通过super调用父类的方法时，方法内部的this指向当前的子类，而不是子类的实例

只要是静态的，都是类的，跟实例没有关系。

下面是一个更完备的实现，将多个类的接口“混入”（mix in）另一个类。

function mix(...mixins) {

class Mix {

constructor() {

for (let mixin of mixins) {

copyProperties(this, new mixin()); // 拷贝实例属性

}

}

}

for (let mixin of mixins) {

copyProperties(Mix, mixin); // 拷贝静态属性

copyProperties(Mix.prototype, mixin.prototype); // 拷贝原型属性

}

return Mix;

}

function copyProperties(target, source) {

for (let key of Reflect.ownKeys(source)) {

if ( key !== 'constructor'

&& key !== 'prototype'

&& key !== 'name'

) {

let desc = Object.getOwnPropertyDescriptor(source, key);

Object.defineProperty(target, key, desc);

}

}

}

上面代码的mix函数，可以将多个对象合成为一个类。使用的时候，只要继承这个类即可。

class DistributedEdit extends mix(Loggable, Serializable) {

// ...

}

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

export var foo = 'bar';

setTimeout(() => foo = 'baz', 500);

上面代码输出变量foo，值为bar，500 毫秒之后变成baz。

import命令具有提升效果，会提升到整个模块的头部，首先执行。

foo();

import { foo } from 'my\_module';

上面的代码不会报错，因为import的执行早于foo的调用。这种行为的本质是，import命令是编译阶段执行的，在代码运行之前。

目前阶段，通过 Babel 转码，CommonJS 模块的require命令和 ES6 模块的import命令，可以写在同一个模块里面，但是最好不要这样做。因为import在静态解析阶段执行，所以它是一个模块之中最早执行的。下面的代码可能不会得到预期结果。

为了给用户提供方便，让他们不用阅读文档就能加载模块，就要用到export default命令，为模块指定默认输出。这时import可以用任意名称

不使用大括号。

有了export default命令，输入模块时就非常直观了，以输入 lodash 模块为例。

import \_ from 'lodash';

export { es6 as default } from './someModule';

// 等同于

import { es6 } from './someModule';

export default es6;

defer与async的区别是：defer要等到整个页面在内存中正常渲染结束（DOM 结构完全生成，以及其他脚本执行完成），才会执行；async一旦下载完，渲染引擎就会中断渲染，执行这个脚本以后，再继续渲染。一句话，defer是“渲染完再执行”，async是“下载完就执行”。另外，如果有多个defer脚本，会按照它们在页面出现的顺序加载，而多个async脚本是不能保证加载顺序的。

利用顶层的this等于undefined这个语法点，可以侦测当前代码是否在 ES6 模块之中。

const isNotModuleScript = this !== undefined;

import加载的值也会跟着变。因此，ES6 模块是动态引用，并且不会缓存值，模块里面的变量绑定其所在的模块。

还是举上面的例子。

// lib.js

export let counter = 3;

export function incCounter() {

counter++;

}

// main.js

import { counter, incCounter } from './lib';

console.log(counter); // 3

incCounter();

console.log(counter); // 4

上面代码说明，ES6 模块输入的变量counter是活的，完全反应其所在模块lib.js内部的变化。

由于 ES6 模块是编译时确定输出接口，CommonJS 模块是运行时确定输出接口，所以采用import命令加载 CommonJS 模块时，不允许采用下面的写法。

// 不正确

import { readFile } from 'fs';

上面的写法不正确，因为fs是 CommonJS 格式，只有在运行时才能确定readFile接口，而import命令要求编译时就确定这个接口。解决方法就是改为整体输入。

// 正确的写法一

import \* as express from 'express';

const app = express.default();

// 正确的写法二

import express from 'express';

const app = express();

### CommonJS 模块加载 ES6 模块

CommonJS 模块加载 ES6 模块，不能使用require命令，而要使用import()函数。ES6 模块的所有输出接口，会成为输入对象的属性。

### SystemJS [§](http://es6.ruanyifeng.com/#docs/module-loader#SystemJS) [⇧](http://es6.ruanyifeng.com/#docs/module-loader)

另一种解决方法是使用 [SystemJS](https://github.com/systemjs/systemjs)。它是一个垫片库（polyfill），可以在浏览器内加载 ES6 模块、AMD 模块和 CommonJS 模块，将其转为 ES5 格式。它在后台调用的是 Google 的 Traceur 转码器。

## 方法的修饰

修饰器不仅可以修饰类，还可以修饰类的属性。

class Person {

@readonly

name() { return `${this.first} ${this.last}` }

}

上面代码中，修饰器readonly用来修饰“类”的name方法。

修饰器函数readonly一共可以接受三个参数。

function readonly(target, name, descriptor){

// descriptor对象原来的值如下

// {

// value: specifiedFunction,

// enumerable: false,

// configurable: true,

// writable: true

// };

descriptor.writable = false;

return descriptor;

}

readonly(Person.prototype, 'name', descriptor);

// 类似于

Object.defineProperty(Person.prototype, 'name', descriptor);

修饰器第一个参数是类的原型对象，上例是Person.prototype，修饰器的本意是要“修饰”类的实例，但是这个时候实例还没生成，所以只能去修饰原型（这不同于类的修饰，那种情况时target参数指的是类本身）；第二个参数是所要修饰的属性名，第三个参数是该属性的描述对象。

另外，上面代码说明，修饰器（readonly）会修改属性的描述对象（descriptor），然后被修改的描述对象再用来定义属性。

如果同一个方法有多个修饰器，会像剥洋葱一样，先从外到内进入，然后由内向外执行。

function dec(id){

console.log('evaluated', id);

return (target, property, descriptor) => console.log('executed', id);

}

class Example {

@dec(1)

@dec(2)

method(){}

}

// evaluated 1

// evaluated 2

// executed 2

// executed 1

上面代码中，外层修饰器@dec(1)先进入，但是内层修饰器@dec(2)先执行。

## Mixin [§](http://es6.ruanyifeng.com/#docs/decorator#Mixin) [⇧](http://es6.ruanyifeng.com/#docs/decorator)

在修饰器的基础上，可以实现Mixin模式。所谓Mixin模式，就是对象继承的一种替代方案，中文译为“混入”（mix in），意为在一个对象之中混入另外一个对象的方法。

对象扩展：

Object.assign方法用于对象的合并，将源对象（source）的所有可枚举属性，复制到目标对象（target）。

由于undefined和null无法转成对象，所以如果它们作为参数，就会报错。

Object.assign(undefined) // 报错

Object.assign(null) // 报错

如果非对象参数出现在源对象的位置（即非首参数），那么处理规则有所不同。首先，这些参数都会转成对象，如果无法转成对象，就会跳过。这意味着，如果undefined和null不在首参数，就不会报错。

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。

const v1 = 'abc';

const v2 = true;

const v3 = 10;

const obj = Object.assign({}, v1, v2, v3);

console.log(obj); // { "0": "a", "1": "b", "2": "c" }

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。

Object.assign拷贝的属性是有限制的，只拷贝源对象的自身属性（不拷贝继承属性），也不拷贝不可枚举的属性（enumerable: false）。

属性名为 Symbol 值的属性，也会被Object.assign拷贝。

**注意点**

**（1）浅拷贝**

**（2）同名属性的替换**

const target = { a: { b: 'c', d: 'e' } }

const source = { a: { b: 'hello' } }

Object.assign(target, source)

// { a: { b: 'hello' } }

**（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。

Object.assign([1, 2, 3], [4, 5])

// [4, 5, 3]

**4）取值函数的处理**

Object.assign只能进行值的复制，如果要复制的值是一个取值函数，那么将求值后再复制。

const source = {

get foo() { return 1 }

};

const target = {};

Object.assign(target, source)

// { foo: 1 }

**Object.getOwnPropertyDescriptors()**

该方法的引入目的，主要是为了解决Object.assign()无法正确拷贝get属性和set属性的问题。

const source = {

set foo(value) {

console.log(value);

}

};

const target2 = {};

Object.defineProperties(target2, Object.getOwnPropertyDescriptors(source));

Object.getOwnPropertyDescriptor(target2, 'foo')

上面代码中，两个对象合并的逻辑可以写成一个函数。

const shallowMerge = (target, source) => Object.defineProperties(

target,

Object.getOwnPropertyDescriptors(source)

);

Object.getOwnPropertyDescriptors()方法的另一个用处，是配合Object.create()方法，将对象属性克隆到一个新对象。这属于浅拷贝。

const clone = Object.create(Object.getPrototypeOf(obj),

Object.getOwnPropertyDescriptors(obj));

// 或者

const shallowClone = (obj) => Object.create(

Object.getPrototypeOf(obj),

Object.getOwnPropertyDescriptors(obj)

);

上面代码会克隆对象obj。

如果想完整克隆一个对象，还拷贝对象原型的属性，可以采用下面的写法。

// 写法一

const clone1 = {

\_\_proto\_\_: Object.getPrototypeOf(obj),

...obj

};

// 写法二

const clone2 = Object.assign(

Object.create(Object.getPrototypeOf(obj)),

obj

);

// 写法三

const clone3 = Object.create(

Object.getPrototypeOf(obj),

Object.getOwnPropertyDescriptors(obj)

)

扩展运算符…

Object.create()

Object.setPrototype(tar, obj)