**介绍**

ECMAScript是一种由Ecma国际（前身为欧洲计算机制造商协会）在标准ECMA-262中定义的脚本语言规范。这种语言在万维网上应用广泛，它往往被称为JavaScript或JScript，但实际上后两者是ECMA-262标准的实现和扩展。

**历史版本**

**至发稿日为止有九个ECMA-262版本发表。其历史版本如下：**

1. 1997年6月：第一版
2. 1998年6月：修改格式，使其与ISO/IEC16262国际标准一样
3. 1999年12月：强大的正则表达式，更好的词法作用域链处理，新的控制指令，异常处理，错误定义更加明确，数据输出的格式化及其它改变
4. 2009年12月：添加严格模式("use strict")。修改了前面版本模糊不清的概念。增加了getters，setters，JSON以及在对象属性上更完整的反射。
5. 2011年6月：ECMAScript标5.1版形式上完全一致于国际标准ISO/IEC 16262:2011。
6. 2015年6月：ECMAScript 2015（ES2015），第 6 版，最早被称作是 ECMAScript 6（ES6），添加了类和模块的语法，其他特性包括迭代器，Python风格的生成器和生成器表达式，箭头函数，二进制数据，静态类型数组，集合（maps，sets 和 weak maps），promise，reflection 和 proxies。作为最早的 ECMAScript Harmony 版本，也被叫做ES6 Harmony。
7. 2016年6月：ECMAScript 2016（ES2016），第 7 版，多个新的概念和语言特性。
8. 2017年6月：ECMAScript 2017（ES2017），第 8 版，多个新的概念和语言特性。
9. 2018年6月：ECMAScript 2018 （ES2018），第 9 版，包含了异步循环，生成器，新的正则表达式特性和 rest/spread 语法。
10. 2019年6月：ECMAScript 2019 （ES2019），第 10 版。

**发展标准**

TC39（Technical Committee 39）是一个推动JavaScript发展的委员会，它的成语来自各个主流浏览器的代表成语。会议实行多数决，每一项决策只有大部分人同意且没有强烈反对才能去实现。

TC39成员制定着ECMAScript的未来。

每一项新特性最终要进入到ECMAScript规范里，需要经历5个阶段，这5个阶段如下：

* **Stage 0**: Strawperson  
  只要是TC39成员或者贡献者，都可以提交想法
* **Stage 1**: Proposal  
  这个阶段确定一个正式的提案
* **Stage 2**: draft  
  规范的第一个版本，进入此阶段的提案大概率会成为标准
* **Stage 3**: Candidate  
  进一步完善提案细则
* **Stage 4**: Finished  
  表示已准备好将其添加到正式的ECMAScript标准中

**由于ES6以前的属性诞生年底久远，我们使用也比较普遍，遂不进行说明，ES6之后的语言风格跟ES5以前的差异比较大，所以单独拎出来做个记录。**

ES6(ES2015)

ES6是一次重大的革新，比起过去的版本，改动比较大，本文仅对常用的API以及语法糖进行讲解。

Let 和 Const

在ES6以前，JS只有var一种声明方式，但是在ES6之后，就多了let跟const这两种方式。用var定义的变量没有块级作用域的概念，而let跟const则会有，因为这三个关键字创建是不一样的。

区别如下：

{

var a = 10

let b = 20

const c = 30

}

a // 10

b // Uncaught ReferenceError: b is not defined

c // c is not defined

let d = 40

const e = 50

d = 60

d // 60

e = 70 // VM231:1 Uncaught TypeError: Assignment to constant variable.



**类（Class）**

在ES6之前，如果我们要生成一个实例对象，传统的方法就是写一个构造函数，例子如下：

function Person(name, age) {

this.name = name

this.age = age

}

Person.prototype.information = function () {

return 'My name is ' + this.name + ', I am ' + this.age

}

但是在ES6之后，我们只需要写成以下形式：

class Person {

constructor(name, age) {

this.name = name

this.age = age

}

information() {

return 'My name is ' + this.name + ', I am ' + this.age

}

}

**箭头函数（Arrow function）**

箭头函数表达式的语法比函数表达式更简洁，并且没有自己的this，arguments，super或 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。

在ES6以前，我们写函数一般是：

var list = [1, 2, 3, 4, 5, 6, 7]

var newList = list.map(function (item) {

return item \* item

})

但是在ES6里，我们可以：

const list = [1, 2, 3, 4, 5, 6, 7]

const newList = list.map(item => item \* item)

看，是不是简洁了不少

**函数参数默认值（Function parameter defaults）**

在ES6之前，如果我们写函数需要定义初始值的时候，需要这么写：

function config (data) {

var data = data || 'data is empty'

}

这样看起来也没有问题，但是如果参数的布尔值为**falsy**时就会出问题，例如我们这样调用config：

config(0)

config('')

那么结果就永远是后面的值

如果我们用函数参数默认值就没有这个问题，写法如下：

const config = (data = 'data is empty') => {}

**模板字符串（Template string）**

在ES6之前，如果我们要拼接字符串，则需要像这样：

var name = 'kris'

var age = 24

var info = 'My name is ' + this.name + ', I am ' + this.age

但是在ES6之后，我们只需要写成以下形式：

const name = 'kris'

const age = 24

const info = `My name is ${name}, I am ${age}`

**解构赋值（Destructuring assignment）**

我们通过解构赋值, 可以将属性/值从对象/数组中取出,赋值给其他变量。

比如我们需要交换两个变量的值，在ES6之前我们可能需要：

var a = 10

var b = 20

var temp = a

a = b

b = temp

但是在ES6里，我们有：

let a = 10

let b = 20

[a, b] = [b, a]

是不是方便很多

**模块化（Module）**

在ES6之前，JS并没有模块化的概念，有的也只是社区定制的类似CommonJS和AMD之类的规则。例如基于CommonJS的NodeJS：

// circle.js

// 输出

const { PI } = Math

exports.area = (r) => PI \* r \*\* 2

exports.circumference = (r) => 2 \* PI \* r

// index.js

// 输入

const circle = require('./circle.js')

console.log(`半径为 4 的圆的面积是 ${circle.area(4)}`)

在ES6之后我们则可以写成以下形式：

// circle.js

// 输出

const { PI } = Math

export const area = (r) => PI \* r \*\* 2

export const circumference = (r) => 2 \* PI \* r

// index.js

// 输入

import {

area

} = './circle.js'

console.log(`半径为 4 的圆的面积是: ${area(4)}`)

**扩展操作符（Spread operator）**

扩展操作符可以在函数调用/数组构造时, 将数组表达式或者string在语法层面展开；还可以在构造字面量对象时, 将对象表达式按key-value的方式展开。

比如在ES5的时候，我们要对一个数组的元素进行相加，在不使用reduce或者reduceRight的场合，我们需要：

function sum(x, y, z) {

return x + y + z;

}

var list = [5, 6, 7]

var total = sum.apply(null, list)

但是如果我们使用扩展操作符，只需要如下：

const sum = (x, y, z) => x + y + z

const list = [5, 6, 7]

const total = sum(...list)

非常的简单，但是要注意的是扩展操作符只能用于可迭代对象

如果是下面的情况，是会报错的：

var obj = {'key1': 'value1'}

var array = [...obj] // TypeError: obj is not iterable

对象属性简写（Object attribute shorthand）

在ES6之前，如果我们要将某个变量赋值为同样名称的对象元素，则需要：

var cat = 'Miaow'

var dog = 'Woof'

var bird = 'Peet peet'

var someObject = {

cat: cat,

dog: dog,

bird: bird

}

但是在ES6里我们就方便很多：

let cat = 'Miaow'

let dog = 'Woof'

let bird = 'Peet peet'

let someObject = {

cat,

dog,

bird

}

console.log(someObject)

//{

// cat: "Miaow",

// dog: "Woof",

// bird: "Peet peet"

//}

非常方便

**Promise**

Promise 是ES6提供的一种异步解决方案，比回调函数更加清晰明了。

Promise 翻译过来就是承诺的意思，这个承诺会在未来有一个确切的答复，并且该承诺有三种状态，分别是：

1. 等待中（pending）
2. 完成了 （resolved）
3. 拒绝了（rejected）

这个承诺一旦从等待状态变成为其他状态就永远不能更改状态了，也就是说一旦状态变为 resolved 后，就不能再次改变

new Promise((resolve, reject) => {

resolve('success')

// 无效

reject('reject')

})

当我们在构造Promise的时候，构造函数内部的代码是立即执行的

new Promise((resolve, reject) => {

console.log('new Promise')

resolve('success')

})

console.log('finifsh')

// new Promise -> finifsh

Promise实现了链式调用，也就是说每次调用then之后返回的都是一个Promise，并且是一个全新的Promise，原因也是因为状态不可变。如果你在then中 使用了return，那么return的值会被Promise.resolve()包装

Promise.resolve(1)

.then(res => {

console.log(res) // => 1

return 2 // 包装成 Promise.resolve(2)

})

.then(res => {

console.log(res) // => 2

})

当然了，Promise也很好地解决了回调地狱的问题，例如：

ajax(url, () => {

// 处理逻辑

ajax(url1, () => {

// 处理逻辑

ajax(url2, () => {

// 处理逻辑

})

})

})

可以改写成：

ajax(url)

.then(res => {

console.log(res)

return ajax(url1)

}).then(res => {

console.log(res)

return ajax(url2)

}).then(res => console.log(res))

**for…of**

for...of语句在可迭代对象（包括 Array，Map，Set，String，TypedArray，arguments 对象等等）上创建一个迭代循环，调用自定义迭代钩子，并为每个不同属性的值执行语句。

例子如下：

const array1 = ['a', 'b', 'c'];

for (const element of array1) {

console.log(element)

}

// "a"

// "b"

// "c"

**Symbol**

**symbol** 是一种基本数据类型，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。

例子如下：

const symbol1 = Symbol();

const symbol2 = Symbol(42);

const symbol3 = Symbol('foo');

console.log(typeof symbol1); // "symbol"

console.log(symbol3.toString()); // "Symbol(foo)"

console.log(Symbol('foo') === Symbol('foo')); // false

**迭代器（Iterator）/ 生成器（Generator）**

迭代器（Iterator）是一种迭代的机制，为各种不同的数据结构提供统一的访问机制。任何数据结构只要内部有 Iterator 接口，就可以完成依次迭代操作。

一旦创建，迭代器对象可以通过重复调用next()显式地迭代，从而获取该对象每一级的值，直到迭代完，返回{ value: undefined, done: true }

虽然自定义的迭代器是一个有用的工具，但由于需要显式地维护其内部状态，因此需要谨慎地创建。生成器函数提供了一个强大的选择：它允许你定义一个包含自有迭代算法的函数， 同时它可以自动维护自己的状态。 生成器函数使用 [function\*](https://link.zhihu.com/?target=https%3A//developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Statements/function%2A)语法编写。 最初调用时，生成器函数不执行任何代码，而是返回一种称为Generator的迭代器。 通过调用生成器的下一个方法消耗值时，Generator函数将执行，直到遇到yield关键字。

可以根据需要多次调用该函数，并且每次都返回一个新的Generator，但每个Generator只能迭代一次。

所以我们可以有以下例子：

function\* makeRangeIterator(start = 0, end = Infinity, step = 1) {

for (let i = start; i < end; i += step) {

yield i;

}

}

var a = makeRangeIterator(1,10,2)

a.next() // {value: 1, done: false}

a.next() // {value: 3, done: false}

a.next() // {value: 5, done: false}

a.next() // {value: 7, done: false}

a.next() // {value: 9, done: false}

a.next() // {value: undefined, done: true}

**Set/WeakSet**

Set 对象允许你存储任何类型的唯一值，无论是原始值或者是对象引用。

所以我们可以通过Set实现数组去重

const numbers = [2,3,4,4,2,3,3,4,4,5,5,6,6,7,5,32,3,4,5]

console.log([...new Set(numbers)])

// [2, 3, 4, 5, 6, 7, 32]

WeakSet 结构与 Set 类似，但区别有以下两点：

* WeakSet 对象中只能存放对象引用, 不能存放值, 而 Set 对象都可以。
* WeakSet 对象中存储的对象值都是被弱引用的, 如果没有其他的变量或属性引用这个对象值, 则这个对象值会被当成垃圾回收掉. 正因为这样, WeakSet 对象是无法被枚举的, 没有办法拿到它包含的所有元素。

所以代码如下：

var ws = new WeakSet()

var obj = {}

var foo = {}

ws.add(window)

ws.add(obj)

ws.has(window) // true

ws.has(foo) // false, 对象 foo 并没有被添加进 ws 中

ws.delete(window) // 从集合中删除 window 对象

ws.has(window) // false, window 对象已经被删除了

ws.clear() // 清空整个 WeakSet 对象

**Map/WeakMap**

Map 对象保存键值对。任何值(对象或者原始值) 都可以作为一个键或一个值。

例子如下，我们甚至可以使用NaN来作为键值：

var myMap = new Map();

myMap.set(NaN, "not a number");

myMap.get(NaN); // "not a number"

var otherNaN = Number("foo");

myMap.get(otherNaN); // "not a number"

WeakMap 对象是一组键/值对的集合，其中的键是弱引用的。其键必须是对象，而值可以是任意的。

跟Map的区别与Set跟WeakSet的区别相似，具体代码如下：

var wm1 = new WeakMap(),

wm2 = new WeakMap(),

wm3 = new WeakMap();

var o1 = {},

o2 = function(){},

o3 = window;

wm1.set(o1, 37);

wm1.set(o2, "azerty");

wm2.set(o1, o2); // value可以是任意值,包括一个对象

wm2.set(o3, undefined);

wm2.set(wm1, wm2); // 键和值可以是任意对象,甚至另外一个WeakMap对象

wm1.get(o2); // "azerty"

wm2.get(o2); // undefined,wm2中没有o2这个键

wm2.get(o3); // undefined,值就是undefined

wm1.has(o2); // true

wm2.has(o2); // false

wm2.has(o3); // true (即使值是undefined)

wm3.set(o1, 37);

wm3.get(o1); // 37

wm3.clear();

wm3.get(o1); // undefined,wm3已被清空

wm1.has(o1); // true

wm1.delete(o1);

wm1.has(o1); // false

**Proxy/Reflect**

Proxy 对象用于定义基本操作的自定义行为（如属性查找，赋值，枚举，函数调用等）。

Reflect 是一个内置的对象，它提供拦截 JavaScript 操作的方法。这些方法与 Proxy 的方法相同。Reflect不是一个函数对象，因此它是不可构造的。

Proxy跟Reflect是非常完美的配合，例子如下：

const observe = (data, callback) => {

return new Proxy(data, {

get(target, key) {

return Reflect.get(target, key)

},

set(target, key, value, proxy) {

callback(key, value);

target[key] = value;

return Reflect.set(target, key, value, proxy)

}

})

}

const FooBar = { open: false };

const FooBarObserver = observe(FooBar, (property, value) => {

property === 'open' && value

? console.log('FooBar is open!!!')

: console.log('keep waiting');

});

console.log(FooBarObserver.open) // false

FooBarObserver.open = true // FooBar is open!!!

当然也不是什么都可以被代理的，如果对象带有configurable: false 跟writable: false 属性，则代理失效。

**Regex对象的扩展**

正则新增符号

i修饰符

// i 修饰符

/[a-z]/i.test('\u212A') // false

/[a-z]/iu.test('\u212A') // true

y修饰符

// y修饰符

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

String.prototype.flags

// 查看RegExp构造函数的修饰符

var regex = new RegExp('xyz', 'i')

regex.flags // 'i'

unicode模式

var s = ' '

/^.$/.test(s) // false

/^.$/u.test(s) // true

u转义

// u转义

/\,/ // /\,/

/\,/u // 报错 没有u修饰符时，逗号前面的反斜杠是无效的，加了u修饰符就报错。

引用

const RE\_TWICE = /^(?<word>[a-z]+)!\k<word>$/;

RE\_TWICE.test('abc!abc') // true

RE\_TWICE.test('abc!ab') // false

const RE\_TWICE = /^(?<word>[a-z]+)!\1$/;

RE\_TWICE.test('abc!abc') // true

RE\_TWICE.test('abc!ab') // false

字符串方法的实现改为调用RegExp方法

* 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]

**正则新增属性**

* RegExp.prototype.sticky 表示是否有y修饰符

/hello\d/y.sticky // true

RegExp.prototype.flags获取修饰符

/abc/ig.flags // 'gi'

**Math对象的扩展**

* 二进制表示法 : 0b或0B开头表示二进制(0bXX或0BXX)
* 二进制表示法 : 0b或0B开头表示二进制(0bXX或0BXX)
* 八进制表示法 : 0o或0O开头表示二进制(0oXX或0OXX)
* Number.EPSILON : 数值最小精度
* Number.MIN\_SAFE\_INTEGER : 最小安全数值(-2^53)
* Number.MAX\_SAFE\_INTEGER : 最大安全数值(2^53)
* Number.parseInt() : 返回转换值的整数部分
* Number.parseFloat() : 返回转换值的浮点数部分
* Number.isFinite() : 是否为有限数值
* Number.isNaN() : 是否为NaN
* Number.isInteger() : 是否为整数
* Number.isSafeInteger() : 是否在数值安全范围内
* Math.trunc() : 返回数值整数部分
* Math.sign() : 返回数值类型(正数1、负数-1、零0)
* Math.cbrt() : 返回数值立方根
* Math.clz32() : 返回数值的32位无符号整数形式
* Math.imul() : 返回两个数值相乘
* Math.fround() : 返回数值的32位单精度浮点数形式
* Math.hypot() : 返回所有数值平方和的平方根
* Math.expm1() : 返回e^n - 1
* Math.log1p() : 返回1 + n的自然对数(Math.log(1 + n))
* Math.log10() : 返回以10为底的n的对数
* Math.log2() : 返回以2为底的n的对数
* Math.sinh() : 返回n的双曲正弦
* Math.cosh() : 返回n的双曲余弦
* Math.tanh() : 返回n的双曲正切
* Math.asinh() : 返回n的反双曲正弦
* Math.acosh() : 返回n的反双曲余弦
* Math.atanh() : 返回n的反双曲正切

**Array对象的扩展**

* Array.prototype.from：转换具有Iterator接口的数据结构为真正数组，返回新数组。

console.log(Array.from('foo')) // ["f", "o", "o"]

console.log(Array.from([1, 2, 3], x => x + x)) // [2, 4, 6]

Array.prototype.of()：转换一组值为真正数组，返回新数组。

Array.of(7) // [7]

Array.of(1, 2, 3) // [1, 2, 3]

Array(7) // [empty, empty, empty, empty, empty, empty]

Array(1, 2, 3) // [1, 2, 3]

Array.prototype.copyWithin()：把指定位置的成员复制到其他位置，返回原数组

const array1 = ['a', 'b', 'c', 'd', 'e']

console.log(array1.copyWithin(0, 3, 4)) // ["d", "b", "c", "d", "e"]

console.log(array1.copyWithin(1, 3)) // ["d", "d", "e", "d", "e"]

Array.prototype.find()：返回第一个符合条件的成员

const array1 = [5, 12, 8, 130, 44]

const found = array1.find(element => element > 10)

console.log(found) // 12

Array.prototype.findIndex()：返回第一个符合条件的成员索引值

const array1 = [5, 12, 8, 130, 44]

const isLargeNumber = (element) => element > 13

console.log(array1.findIndex(isLargeNumber)) // 3

Array.prototype.fill()：根据指定值填充整个数组，返回原数组

const array1 = [1, 2, 3, 4]

console.log(array1.fill(0, 2, 4)) // [1, 2, 0, 0]

console.log(array1.fill(5, 1)) // [1, 5, 5, 5]

console.log(array1.fill(6)) // [6, 6, 6, 6]

Array.prototype.keys()：返回以索引值为遍历器的对象

const array1 = ['a', 'b', 'c']

const iterator = array1.keys()

for (const key of iterator) {

console.log(key)

}

// 0

// 1

// 2

Array.prototype.values()：返回以属性值为遍历器的对象

const array1 = ['a', 'b', 'c']

const iterator = array1.values()

for (const key of iterator) {

console.log(key)

}

// a

// b

// c

Array.prototype.entries()：返回以索引值和属性值为遍历器的对象

const array1 = ['a', 'b', 'c']

const iterator = array1.entries()

console.log(iterator.next().value) // [0, "a"]

console.log(iterator.next().value) // [1, "b"]

数组空位：ES6明确将数组空位转为undefined或者empty

Array.from(['a',,'b']) // [ "a", undefined, "b" ]

[...['a',,'b']] // [ "a", undefined, "b" ]

Array(3) // [empty × 3]

[,'a'] // [empty, "a"]

**ES7(ES2016)**

**Array.prototype.includes()**

includes() 方法用来判断一个数组是否包含一个指定的值，根据情况，如果包含则返回 true，否则返回false。

代码如下：

const array1 = [1, 2, 3]

console.log(array1.includes(2)) // true

const pets = ['cat', 'dog', 'bat']

console.log(pets.includes('cat')) // true

console.log(pets.includes('at')) // false

**幂运算符\*\***

幂运算符\*\*，具有与Math.pow()一样的功能，代码如下：

console.log(2\*\*10) // 1024

console.log(Math.pow(2, 10)) // 1024

模板字符串（Template string）

自ES7起，带标签的模版字面量遵守以下转义序列的规则：

* Unicode字符以"\u"开头，例如\u00A9
* Unicode码位用"\u{}"表示，例如\u{2F804}
* 十六进制以"\x"开头，例如\xA9
* 八进制以""和数字开头，例如\251

这表示类似下面这种带标签的模版是有问题的，因为对于每一个ECMAScript语法，解析器都会去查找有效的转义序列，但是只能得到这是一个形式错误的语法：

latex`\unicode`

// 在较老的ECMAScript版本中报错（ES2016及更早）

// SyntaxError: malformed Unicode character escape sequence

**ES8(ES2017)**

**async/await**

虽然Promise可以解决回调地狱的问题，但是链式调用太多，则会变成另一种形式的回调地狱 —— 面条地狱，所以在ES8里则出现了Promise的语法糖async/await，专门解决这个问题。

我们先看一下下面的Promise代码：

fetch('coffee.jpg')

.then(response => response.blob())

.then(myBlob => {

let objectURL = URL.createObjectURL(myBlob)

let image = document.createElement('img')

image.src = objectURL

document.body.appendChild(image)

})

.catch(e => {

console.log('There has been a problem with your fetch operation: ' + e.message)

})

然后再看看async/await版的，这样看起来是不是更清晰了。

async function myFetch() {

let response = await fetch('coffee.jpg')

let myBlob = await response.blob()

let objectURL = URL.createObjectURL(myBlob)

let image = document.createElement('img')

image.src = objectURL

document.body.appendChild(image)

}

myFetch()

当然，如果你喜欢，你甚至可以两者混用

async function myFetch() {

let response = await fetch('coffee.jpg')

return await response.blob()

}

myFetch().then((blob) => {

let objectURL = URL.createObjectURL(blob)

let image = document.createElement('img')

image.src = objectURL

document.body.appendChild(image)

})

**Object.values()**

Object.values()方法返回一个给定对象自身的所有可枚举属性值的数组，值的顺序与使用for…in循环的顺序相同 ( 区别在于 for-in 循环枚举原型链中的属性 )。

代码如下：

const object1 = {

a: 'somestring',

b: 42,

c: false

}

console.log(Object.values(object1)) // ["somestring", 42, false]

**Object.entries()**

Object.entries()方法返回一个给定对象自身可枚举属性的键值对数组，其排列与使用 for…in 循环遍历该对象时返回的顺序一致（区别在于 for-in 循环还会枚举原型链中的属性）。

代码如下：

const object1 = {

a: 'somestring',

b: 42

}

for (let [key, value] of Object.entries(object1)) {

console.log(`${key}: ${value}`)

}

// "a: somestring"

// "b: 42"

**padStart()**

padStart() 方法用另一个字符串填充当前字符串(重复，如果需要的话)，以便产生的字符串达到给定的长度。填充从当前字符串的开始(左侧)应用的。

代码如下：

const str1 = '5'

console.log(str1.padStart(2, '0')) // "05"

const fullNumber = '2034399002125581'

const last4Digits = fullNumber.slice(-4)

const maskedNumber = last4Digits.padStart(fullNumber.length, '\*')

console.log(maskedNumber) // "\*\*\*\*\*\*\*\*\*\*\*\*5581"

**padEnd()**

padEnd() 方法会用一个字符串填充当前字符串（如果需要的话则重复填充），返回填充后达到指定长度的字符串。从当前字符串的末尾（右侧）开始填充。

const str1 = 'Breaded Mushrooms'

console.log(str1.padEnd(25, '.')) // "Breaded Mushrooms........"

const str2 = '200'

console.log(str2.padEnd(5)) // "200 "

###函数参数结尾逗号（Function parameter lists and calls trailing commas）

在ES5里就添加了对象的尾逗号，不过并不支持函数参数，但是在ES8之后，便开始支持这一特性，代码如下：

// 参数定义

function f(p) {}

function f(p,) {}

(p) => {}

(p,) => {}

class C {

one(a,) {},

two(a, b,) {},

}

var obj = {

one(a,) {},

two(a, b,) {},

};

// 函数调用

f(p)

f(p,)

Math.max(10, 20)

Math.max(10, 20,)

但是以下的方式是不合法的：

仅仅包含逗号的函数参数定义或者函数调用会抛出 SyntaxError。 而且，当使用剩余参数的时候，并不支持尾后逗号，例子如下：

function f(,) {} // SyntaxError: missing formal parameter

(,) => {} // SyntaxError: expected expression, got ','

f(,) // SyntaxError: expected expression, got ','

function f(...p,) {} // SyntaxError: parameter after rest parameter

(...p,) => {} // SyntaxError: expected closing parenthesis, got ','

在解构里也可以使用，代码如下：

// 带有尾后逗号的数组解构

[a, b,] = [1, 2]

// 带有尾后逗号的对象解构

var o = {

p: 42,

q: true,

}

var {p, q,} = o

同样地，在使用剩余参数时，会抛出 SyntaxError，代码如下：

var [a, ...b,] = [1, 2, 3] // SyntaxError: rest element may not have a trailing comma

**ShareArrayBuffer（因安全问题，暂时在Chrome跟FireFox中被禁用）**

SharedArrayBuffer 对象用来表示一个通用的，固定长度的原始二进制数据缓冲区，类似于 ArrayBuffer 对象，它们都可以用来在共享内存（shared memory）上创建视图。与 ArrayBuffer 不同的是，SharedArrayBuffer 不能被分离。

代码如下：

let sab = new SharedArrayBuffer(1024) // 必须实例化

worker.postMessage(sab)

Atomics对象

**Atomics对象** 提供了一组静态方法用来对 [SharedArrayBuffer](https://link.zhihu.com/?target=https%3A//developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/SharedArrayBuffer) 对象进行原子操作。

方法如下：

* **Atomics.add()** ：将指定位置上的数组元素与给定的值相加，并返回相加前该元素的值。
* **Atomics.and()**：将指定位置上的数组元素与给定的值相与，并返回与操作前该元素的值。
* **Atomics.compareExchange()**：如果数组中指定的元素与给定的值相等，则将其更新为新的值，并返回该元素原先的值。
* **Atomics.exchange()**：将数组中指定的元素更新为给定的值，并返回该元素更新前的值。
* **Atomics.load()**：返回数组中指定元素的值。
* **Atomics.or()**：将指定位置上的数组元素与给定的值相或，并返回或操作前该元素的值。
* **Atomics.store()**：将数组中指定的元素设置为给定的值，并返回该值。
* **Atomics.sub()**：将指定位置上的数组元素与给定的值相减，并返回相减前该元素的值。
* **Atomics.xor()**：将指定位置上的数组元素与给定的值相异或，并返回异或操作前该元素的值。
* **Atomics.wait()**：检测数组中某个指定位置上的值是否仍然是给定值，是则保持挂起直到被唤醒或超时。返回值为 “ok”、“not-equal” 或 “time-out”。调用时，如果当前线程不允许阻塞，则会抛出异常（大多数浏览器都不允许在主线程中调用 wait()）。
* **Atomics.wake()**：唤醒等待队列中正在数组指定位置的元素上等待的线程。返回值为成功唤醒的线程数量。
* **Atomics.isLockFree(size)**：可以用来检测当前系统是否支持硬件级的原子操作。对于指定大小的数组，如果当前系统支持硬件级的原子操作，则返回 true；否则就意味着对于该数组，Atomics 对象中的各原子操作都只能用锁来实现。此函数面向的是技术专家。

**Object.getOwnPropertyDescriptors()**

Object.getOwnPropertyDescriptors() 方法用来获取一个对象的所有自身属性的描述符。代码如下：

const object1 = {

property1: 42

}

const descriptors1 = Object.getOwnPropertyDescriptors(object1)

console.log(descriptors1.property1.writable) // true

console.log(descriptors1.property1.value) // 42

// 浅拷贝一个对象

Object.create(

Object.getPrototypeOf(obj),

Object.getOwnPropertyDescriptors(obj)

)

// 创建子类

function superclass() {}

superclass.prototype = {

// 在这里定义方法和属性

}

function subclass() {}

subclass.prototype = Object.create(superclass.prototype, Object.getOwnPropertyDescriptors({

// 在这里定义方法和属性

}))

**ES9(ES2018)**

**for await…of**

for await...of 语句会在异步或者同步可迭代对象上创建一个迭代循环，包括 String，Array，Array-like 对象（比如arguments 或者NodeList)，TypedArray，Map， Set和自定义的异步或者同步可迭代对象。其会调用自定义迭代钩子，并为每个不同属性的值执行语句。

配合迭代异步生成器，例子如下：

async function\* asyncGenerator() {

var i = 0

while (i < 3) {

yield i++

}

}

(async function() {

for await (num of asyncGenerator()) {

console.log(num)

}

})()

// 0

// 1

// 2

**模板字符串（Template string）**

ES9开始，模板字符串允许嵌套支持常见转义序列，移除对ECMAScript在带标签的模版字符串中转义序列的语法限制。

不过，非法转义序列在"cooked"当中仍然会体现出来。它们将以undefined元素的形式存在于"cooked"之中，代码如下：

function latex(str) {

return { "cooked": str[0], "raw": str.raw[0] }

}

latex`\unicode` // { cooked: undefined, raw: "\\unicode" }

正则表达式反向(lookbehind)断言

首先我们得先知道什么是**断言(Assertion)**。

\*\*断言(Assertion)\*\*是一个对当前匹配位置之前或之后的字符的测试， 它不会实际消耗任何字符，所以断言也被称为“非消耗性匹配”或“非获取匹配”。

正则表达式的断言一共有 4 种形式：

* (?=pattern) 零宽正向肯定断言(zero-width positive lookahead assertion)
* (?!pattern) 零宽正向否定断言(zero-width negative lookahead assertion)
* (?<=pattern) 零宽反向肯定断言(zero-width positive lookbehind assertion)
* (?<!pattern) 零宽反向否定断言(zero-width negative lookbehind assertion)

在ES9之前，JavaScript 正则表达式，只支持正向断言。正向断言的意思是：当前位置后面的字符串应该满足断言，但是并不捕获。例子如下：

'fishHeadfishTail'.match(/fish(?=Head)/g) // ["fish"]

反向断言和正向断言的行为一样，只是方向相反。例子如下：

'abc123'.match(/(?<=(\d+)(\d+))$/) // ["", "1", "23", index: 6, input: "abc123", groups: undefined]

**正则表达式 Unicode 转义**

正则表达式中的Unicode转义符允许根据Unicode字符属性匹配Unicode字符。 它允许区分字符类型，例如大写和小写字母，数学符号和标点符号。

部分例子代码如下：

// 匹配所有数字

const regex = /^\p{Number}+$/u;

regex.test('²³¹¼½¾') // true

regex.test('㉛㉜㉝') // true

regex.test('ⅠⅡⅢⅣⅤⅥⅦⅧⅨⅩⅪⅫ') // true

// 匹配所有空格

\p{White\_Space}

// 匹配各种文字的所有字母，等同于 Unicode 版的 \w

[\p{Alphabetic}\p{Mark}\p{Decimal\_Number}\p{Connector\_Punctuation}\p{Join\_Control}]

// 匹配各种文字的所有非字母的字符，等同于 Unicode 版的 \W

[^\p{Alphabetic}\p{Mark}\p{Decimal\_Number}\p{Connector\_Punctuation}\p{Join\_Control}]

// 匹配 Emoji

/\p{Emoji\_Modifier\_Base}\p{Emoji\_Modifier}?|\p{Emoji\_Presentation}|\p{Emoji}\uFE0F/gu

// 匹配所有的箭头字符

const regexArrows = /^\p{Block=Arrows}+$/u;

regexArrows.test('←↑→↓⇏⇐⇑⇒⇓⇔⇕⇖⇗⇘⇙⇧⇩') // true

具体的属性列表可查看：[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular\_Expressions/Unicode\_Property\_Escapes](https://link.zhihu.com/?target=https%3A//developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions/Unicode_Property_Escapes)

**正则表达式 s/dotAll 模式**

在以往的版本里，JS的正则的.只能匹配emoji跟行终结符以外的所有文本，例如：

let regex = /./;

regex.test('\n'); // false

regex.test('\r'); // false

regex.test('\u{2028}'); // false

regex.test('\u{2029}'); // false

regex.test('\v'); // true

regex.test('\f'); // true

regex.test('\u{0085}'); // true

/foo.bar/.test('foo\nbar'); // false

/foo[^]bar/.test('foo\nbar'); // true

/foo.bar/.test('foo\nbar'); // false

/foo[\s]bar/.test('foo\nbar'); // true

但是在ES9之后，JS正则增加了一个新的标志s用来表示 dotAll，这可以匹配任意字符。代码如下：

/foo.bar/s.test('foo\nbar'); // true

const re = /foo.bar/s; // 等价于 const re = new RegExp('foo.bar', 's');

re.test('foo\nbar'); // true

re.dotAll; // true

re.flags; // "s"

**正则表达式命名捕获组**

在以往的版本里，JS的正则分组是无法命名的，所以容易混淆。例如下面获取年月日的例子，很容易让人搞不清哪个是月份，哪个是年份:

const matched = /(\d{4})-(\d{2})-(\d{2})/.exec('2019-01-01')

console.log(matched[0]); // 2019-01-01

console.log(matched[1]); // 2019

console.log(matched[2]); // 01

console.log(matched[3]); // 01

ES9引入了命名捕获组，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。代码如下：

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

const RE\_OPT\_A = /^(?<as>a+)?$/;

const matchObj = RE\_OPT\_A.exec('');

matchObj.groups.as // undefined

'as' in matchObj.groups // true

**对象扩展操作符**

ES6中添加了数组的扩展操作符，让我们在操作数组时更加简便，美中不足的是并不支持对象扩展操作符，但是在ES9开始，这一功能也得到了支持，例如：

var obj1 = { foo: 'bar', x: 42 };

var obj2 = { foo: 'baz', y: 13 };

var clonedObj = { ...obj1 };

// 克隆后的对象: { foo: "bar", x: 42 }

var mergedObj = { ...obj1, ...obj2 };

// 合并后的对象: { foo: "baz", x: 42, y: 13 }

上面便是一个简便的浅拷贝。这里有一点小提示，就是Object.assign() 函数会触发 setters，而展开语法则不会。所以不能替换也不能模拟Object.assign() 。

如果存在相同的属性名，只有最后一个会生效。

**Promise.prototype.finally()**

finally()方法会返回一个Promise，当promise的状态变更，不管是变成rejected或者fulfilled，最终都会执行finally()的回调。

例子如下：

fetch(url)

.then((res) => {

console.log(res)

})

.catch((error) => {

console.log(error)

})

.finally(() => {

console.log('结束')

})

**ES10(ES2019)**

Array.prototype.flat() / flatMap()

flat() 方法会按照一个可指定的深度递归遍历数组，并将所有元素与遍历到的子数组中的元素合并为一个新数组返回。

flatMap()与 map() 方法和深度为1的 flat() 几乎相同.，不过它会首先使用映射函数映射每个元素，然后将结果压缩成一个新数组，这样效率会更高。

例子如下：

var arr1 = [1, 2, 3, 4]

arr1.map(x => [x \* 2]) // [[2], [4], [6], [8]]

arr1.flatMap(x => [x \* 2]) // [2, 4, 6, 8]

// 深度为1

arr1.flatMap(x => [[x \* 2]]) // [[2], [4], [6], [8]]

flatMap()可以代替reduce()与concat()，例子如下：

var arr = [1, 2, 3, 4]

arr.flatMap(x => [x, x \* 2]) // [1, 2, 2, 4, 3, 6, 4, 8]

// 等价于

arr.reduce((acc, x) => acc.concat([x, x \* 2]), []) // [1, 2, 2, 4, 3, 6, 4, 8]

但这是非常低效的，在每次迭代中，它创建一个必须被垃圾收集的新临时数组，并且它将元素从当前的累加器数组复制到一个新的数组中，而不是将新的元素添加到现有的数组中。

String.prototype.trimStart() / trimLeft() / trimEnd() / trimRight()

在ES5中，我们可以通过trim()来去掉字符首尾的空格，但是却无法只去掉单边的，但是在ES10之后，我们可以实现这个功能。

如果我们要去掉开头的空格，可以使用trimStart()或者它的别名trimLeft()，

同样的，如果我们要去掉结尾的空格，我们可以使用trimEnd()或者它的别名trimRight()。

例子如下：

const Str = ' Hello world! '

console.log(Str) // ' Hello world! '

console.log(Str.trimStart()) // 'Hello world! '

console.log(Str.trimLeft()) // 'Hello world! '

console.log(Str.trimEnd()) // ' Hello world!'

console.log(Str.trimRight()) // ' Hello world!'

不过这里有一点要注意的是，trimStart()跟trimEnd()才是标准方法，trimLeft()跟trimRight()只是别名。

在某些引擎里（例如Chrome），有以下的等式：

String.prototype.trimLeft.name === "trimStart"

String.prototype.trimRight.name === "trimEnd"

**Object.fromEntries()**

Object.fromEntries() 方法把键值对列表转换为一个对象，它是Object.entries()的反函数。

例子如下：

const entries = new Map([

['foo', 'bar'],

['baz', 42]

])

const obj = Object.fromEntries(entries)

console.log(obj) // Object { foo: "bar", baz: 42 }

**Symbol.prototype.description**

description 是一个只读属性，它会返回Symbol对象的可选描述的字符串。与 Symbol.prototype.toString() 不同的是它不会包含Symbol()的字符串。例子如下：

Symbol('desc').toString(); // "Symbol(desc)"

Symbol('desc').description; // "desc"

Symbol('').description; // ""

Symbol().description; // undefined

// 具名 symbols

Symbol.iterator.toString(); // "Symbol(Symbol.iterator)"

Symbol.iterator.description; // "Symbol.iterator"

//全局 symbols

Symbol.for('foo').toString(); // "Symbol(foo)"

Symbol.for('foo').description; // "foo"

**String.prototype.matchAll**

matchAll() 方法返回一个包含所有匹配正则表达式的结果及分组捕获组的迭代器。并且返回一个不可重启的迭代器。例子如下：

var regexp = /t(e)(st(\d?))/g

var str = 'test1test2'

str.match(regexp) // ['test1', 'test2']

str.matchAll(regexp) // RegExpStringIterator {}

[...str.matchAll(regexp)] // [['test1', 'e', 'st1', '1', index: 0, input: 'test1test2', length: 4], ['test2', 'e', 'st2', '2', index: 5, input: 'test1test2', length: 4]]

**Function.prototype.toString() 返回注释与空格**

在以往的版本中，Function.prototype.toString()得到的字符串是去掉空白符号的，但是从ES10开始会保留这些空格，如果是原生函数则返回你控制台看到的效果，例子如下：

function sum(a, b) {

return a + b;

}

console.log(sum.toString())

// "function sum(a, b) {

// return a + b;

// }"

console.log(Math.abs.toString()) // "function abs() { [native code] }"

**try-catch**

在以往的版本中，try-catch里catch后面必须带异常参数，例如：

// ES10之前

try {

// tryCode

} catch (err) {

// catchCode

}

但是在ES10之后，这个参数却不是必须的，如果用不到，我们可以不用传，例如：

try {

console.log('Foobar')

} catch {

console.error('Bar')

}

BigInt

**BigInt** 是一种内置对象，它提供了一种方法来表示大于 253 - 1 的整数。这原本是 Javascript中可以用 Number 表示的最大数字。**BigInt** 可以表示任意大的整数。

可以用在一个整数字面量后面加 n 的方式定义一个 BigInt ，如：10n，或者调用函数BigInt()。

在以往的版本中，我们有以下的弊端：

// 大于2的53次方的整数，无法保持精度

2 \*\* 53 === (2 \*\* 53 + 1)

// 超过2的1024次方的数值，无法表示

2 \*\* 1024 // Infinity

但是在ES10引入BigInt之后，这个问题便得到了解决。

以下操作符可以和 BigInt 一起使用： +、\*、-、\*\*、% 。除 >>> （无符号右移）之外的位操作也可以支持。因为 BigInt 都是有符号的， >>> （无符号右移）不能用于 BigInt。BigInt 不支持单目 (+) 运算符。

/ 操作符对于整数的运算也没问题。可是因为这些变量是 BigInt 而不是 BigDecimal ，该操作符结果会向零取整，也就是说不会返回小数部分。

BigInt 和 Number不是严格相等的，但是宽松相等的。

所以在BigInt出来以后，JS的原始类型便增加到了7个，如下：

* Boolean
* Null
* Undefined
* Number
* String
* Symbol (ES6)
* BigInt (ES10)

**globalThis**

globalThis属性包含类似于全局对象 this值。所以在全局环境下，我们有：

globalThis === this // true

**import()**

静态的import 语句用于导入由另一个模块导出的绑定。无论是否声明了 严格模式，导入的模块都运行在严格模式下。在浏览器中，import 语句只能在声明了 type="module" 的 script 的标签中使用。

但是在ES10之后，我们有动态 import()，它不需要依赖 type="module" 的script标签。

所以我们有以下例子：

const main = document.querySelector("main")

for (const link of document.querySelectorAll("nav > a")) {

link.addEventListener("click", e => {

e.preventDefault()

import('/modules/my-module.js')

.then(module => {

module.loadPageInto(main);

})

.catch(err => {

main.textContent = err.message;

})

})

}

**私有元素与方法**

在ES10之前，如果我们要实现一个简单的计数器组件，我们可能会这么写：

// web component 写法

class Counter extends HTMLElement {

get x() {

return this.xValue

}

set x(value) {

this.xValue = value

window.requestAnimationFrame(this.render.bind(this))

}

clicked() {

this.x++

}

constructor() {

super()

this.onclick = this.clicked.bind(this)

this.xValue = 0

}

connectedCallback() {

this.render()

}

render() {

this.textContent = this.x.toString()

}

}

window.customElements.define('num-counter', Counter)

但是在ES10之后我们可以使用私有变量进行组件封装，如下：

class Counter extends HTMLElement {

#xValue = 0

get #x() {

return #xValue

}

set #x(value) {

this.#xValue = value

window.requestAnimationFrame(this.#render.bind(this))

}

#clicked() {

this.#x++

}

constructor() {

super();

this.onclick = this.#clicked.bind(this)

}

connectedCallback() {

this.#render()

}

#render() {

this.textContent = this.#x.toString()

}

}

window.customElements.define('num-counter', Counter)