# JavaScript新标准ES7-ES10调研文档

1. ES的概念，与JavaScript的关系，当前的生态与应用，当前以及未来在项目中的应用情况
2. 什么是ES

ECMAScript是一种由Ecma国际（前身为欧洲计算机制造商协会，European Computer Manufacturers Association）通过ECMA-262标准化的脚本程序设计语言。这种语言在万维网上应用广泛，它往往被称为JavaScript或JScript，所以它可以理解为是JavaScript的一个标准,但实际上后两者是ECMA-262标准的实现和扩展。

javascript于1996年发布了第一版，当时的名字其实不叫javascipt，而是叫livescript，不过发布之后一直不温不火，正好这个时候sun公司推出来了一个面向对象的语言—Java，Java拥有很好的跨平台特性，一处编写处处执行，很快得到了市场的认可。 为了蹭Java的热度，livescript的母公司Netscape，也就是王景公司，就将livescript改为javascript，事实证明，这次名字改的非常明智。 javascript的火爆引起了微软公司的注意，他们很快也推出了Jscript脚本语言，以此来抗衡Netscape公司，并且当时还有一门浏览器脚本语言，叫ScriptEase，这样市场上就有了三种脚本语言，并且各个语言各有特色，互不兼容。 1997年，Netscape给欧洲计算机制造商协会(European Computer ManufacturersAssociation，简称ECMA)提交了一个草案，要求根据javascript来制定一个浏览器脚本语言标准。ECMA邀请了Netscape、Sun、微软、Borland和其他一些对脚本编程感兴趣的公司的程序员组成了一个技术委员会—第 39 技术委员会(TC39)，共同商讨制定了一个脚本语言的规范：ECMAScript。各个脚本语言必须按照规范来实现和约束自己的功能。 当然这个规范也是不断发展的，第一版规范叫做：ECMA-262。

另外，虽然ES标准的实现叫做JavaScript，但是与Java并没有任何关系。JavaScript 与 Java 是两种完全不同的语言，无论在概念上还是设计上。 Java（由 Sun 发明）是更复杂的编程语言。 ECMA-262 是 JavaScript 标准的官方名称。 JavaScript 由 Brendan Eich 发明。它于 1995 年出现在 Netscape 中（该浏览器已停止更新），并于 1997 年被 ECMA（一个标准协会）采纳。

1. ES系列当前的生态与应用

由于浏览器类型和版本多种多样，并不是每一种都能支持到比较高的ES版本的实现，目前主流的最低支持度是IE9-11的ES5，就是普遍的JavaScript实现，曾经ES6的JavaScript实现大家都叫做JavaScript5，但实际上并没有这个语言。

当前比较高版本的webkit核心浏览器（例如chrome，safari），firefox等都有自己对应ES6+的浏览器实现。那么当前前端主流使用的ES6如何兼容并不支持的浏览器呢？现在饿作法是通过编译插件讲ES6+的代码转译为ES5的实现（对应的示例代码即使用这种方式）。现在常用的方法是通过nodejs执行babel程序，来进行转译。

Babel 是一个工具链，主要用于将 ECMAScript 2015+ 版本的代码转换为向后兼容的 JavaScript 语法，以便能够运行在当前和旧版本的浏览器或其他环境中。(具体学习使用可以查看官网https://www.babeljs.cn/，在此就不赘述)

简单来说把 JavaScript 中 es2015/2016/2017/2046 的新语法转化为 es5，让低端运行环境(如浏览器和 node )能够认识并执行。严格来说，babel 也可以转化为更低的规范。但以目前情况来说，es5 规范已经足以覆盖绝大部分浏览器，因此常规来说转到 es5 是一个安全且流行的做法。

目前使用ES6+越来越成为当前前端的主流，主要的前端框架（vue，angular，react等）现在都推荐或者强制使用ES6+来开发。

1. 当前以及未来在项目中的应用

除了极少数的angular1以及更之前的项目，当前事业部内几乎所有前端项目皆是使用ES6开发。因为目前ES标准还在不停发展，不断出现更多更好的特性和优化，对于事业部内前端开发效率，项目的健壮性，代码的合理性都有很大的提升。

1. 新增的实现/特性，旧内容的变更

除了极少数的angular1以及更之前的项目，当前事业部内几乎所有前端项目皆是使用ES6开发。因为目前ES标准还在不停发展，不断出现更多更好的特性和优化，对于事业部内前端开发效率，项目的健壮性，代码的合理性都有很大的提升。

1. ES6新特性（2015）

为了介绍新的特性，先在这里把ES已有的部分特性总结一下，方便比较。

ES6的特性比较多，在 ES5 发布近 6 年（2009-11 至 2015-6）之后才将其标准化。两个发布版本之间时间跨度很大，所以ES6中的特性比较多。 在这里列举几个常用的：

1. 类（class）

对熟悉Java，object-c，c#等纯面向对象语言的开发者来说，都会对class有一种特殊的情怀。ES6 引入了class（类），让JavaScript的面向对象编程变得更加简单和易于理解。

class Animal {

// 构造函数，实例化的时候将会被调用，如果不指定，那么会有一个不带参数的默认构造函数.

constructor(name,color) {

this.name = name;

this.color = color;

}

// toString 是原型对象上的属性

toString() {

console.log('name:' + this.name + ',color:' + this.color);

}

}

var animal = new Animal('dog','white');//实例化Animal

animal.toString();

console.log(animal.hasOwnProperty('name')); //true

console.log(animal.hasOwnProperty('toString')); // false

console.log(animal.\_\_proto\_\_.hasOwnProperty('toString')); // true

class Cat extends Animal {

constructor(action) {

// 子类必须要在constructor中指定super 函数，否则在新建实例的时候会报错.

// 如果没有置顶consructor,默认带super函数的constructor将会被添加、

super('cat','white');

this.action = action;

}

toString() {

console.log(super.toString());

}

}

var cat = new Cat('catch')

cat.toString();

// 实例cat 是 Cat 和 Animal 的实例，和Es5完全一致。

console.log(cat instanceof Cat); // true

console.log(cat instanceof Animal); // true

1. 模块化(Module)

ES5不支持原生的模块化，在ES6中模块作为重要的组成部分被添加进来。模块的功能主要由 export 和 import 组成。每一个模块都有自己单独的作用域，模块之间的相互调用关系是通过 export 来规定模块对外暴露的接口，通过import来引用其它模块提供的接口。同时还为模块创造了命名空间，防止函数的命名冲突。

导出(export)

ES6允许在一个模块中使用export来导出多个变量或函数。

导出变量

//test.jsexport var name = 'Rainbow'

心得：ES6不仅支持变量的导出，也支持常量的导出。 export const sqrt = Math.sqrt;//导出常量

ES6将一个文件视为一个模块，上面的模块通过 export 向外输出了一个变量。一个模块也可以同时往外面输出多个变量。

var name = 'Rainbow';

var age = '24';

export {name, age};

导出函数

myModule.jsexport function myModule(someArg) {

return someArg;

}

导入(import)

定义好模块的输出以后就可以在另外一个模块通过import引用。

import {myModule} from 'myModule';// main.jsimport {name,age} from 'test';// test.js

心得:一条import 语句可以同时导入默认函数和其它变量。

import defaultMethod, { otherMethod } from 'xxx.js';

1. 箭头（Arrow）函数

这是ES6中最令人激动的特性之一。=>不只是关键字function的简写，它还带来了其它好处。箭头函数与包围它的代码共享同一个this,能帮你很好的解决this的指向问题。有经验的JavaScript开发者都熟悉诸如var self = this;或var that = this这种引用外围this的模式。但借助=>，就不需要这种模式了。

当函数没有参数时，（）不能省略

当函数只有一个参数，且函数体是一句代码，且是返回语句  
参数的（）可省略、函数体 {} 可省略、return 可省略、  
中间使用 => 连接

若函数体只有一句，且不是return 语句， 不能省略 {}

若函数体有多条语句，不能省略 {}

若函数有多个参数，不能省略()

若函数的返回值为对象，此时不能省略return

使用箭头函数注意

箭头函数不适用于声明函数

箭头函数不适用于DOM事件

箭头函数不能作为构造函数（迭代器）

箭头函数内不能使用arguments

不能使用yield命令

箭头函数没有this，this是父级的

定义时候绑定，就是this是继承自父执行上下文！！中的this

ES5中，this指调用者，ES6中，this指定义时候绑定

箭头函数的箭头=>之前是一个空括号、单个的参数名、或用括号括起的多个参数名，而箭头之后可以是一个表达式（作为函数的返回值），或者是用花括号括起的函数体（需要自行通过return来返回值，否则返回的是undefined）。

// 箭头函数的例子

()=>1

v=>v+1

(a,b)=>a+b

()=>{

alert("foo");

}

e=>{

if (e == 0){

return 0;

}

return 1000/e;

}

心得：不论是箭头函数还是bind，每次被执行都返回的是一个新的函数引用，因此如果你还需要函数的引用去做一些别的事情（譬如卸载监听器），那么你必须自己保存这个引用。

卸载监听器时的陷阱

错误的做法

class PauseMenu extends React.Component{

componentWillMount(){

AppStateIOS.addEventListener('change', this.onAppPaused.bind(this));

}

componentWillUnmount(){

AppStateIOS.removeEventListener('change', this.onAppPaused.bind(this));

}

onAppPaused(event){

}

}

正确的做法

class PauseMenu extends React.Component{

constructor(props){

super(props);

this.\_onAppPaused = this.onAppPaused.bind(this);

}

componentWillMount(){

AppStateIOS.addEventListener('change', this.\_onAppPaused);

}

componentWillUnmount(){

AppStateIOS.removeEventListener('change', this.\_onAppPaused);

}

onAppPaused(event){

}

}

除上述的做法外，我们还可以这样做：

class PauseMenu extends React.Component{

componentWillMount(){

AppStateIOS.addEventListener('change', this.onAppPaused);

}

componentWillUnmount(){

AppStateIOS.removeEventListener('change', this.onAppPaused);

}

onAppPaused = (event) => {

//把函数直接作为一个arrow function的属性来定义，初始化的时候就绑定好了this指针

}

}

需要注意的是：不论是bind还是箭头函数，每次被执行都返回的是一个新的函数引用，因此如果你还需要函数的引用去做一些别的事情（譬如卸载监听器），那么你必须自己保存这个引用。

1. 函数参数默认值

ES6支持在定义函数的时候为其设置默认值：

function foo(height = 50, color = 'red'){

// ...

}

不使用默认值：

function foo(height, color){

var height = height || 50;

var color = color || 'red';

//...

}

这样写一般没问题，但当参数的布尔值为false时，就会有问题了。比如，我们这样调用foo函数：

foo(0, "")

因为0的布尔值为false，这样height的取值将是50。同理color的取值为‘red’。所以说，函数参数默认值不仅能是代码变得更加简洁而且能规避一些问题。

1. 模板字符串

ES6支持模板字符串，使得字符串的拼接更加的简洁、直观。

不使用模板字符串：

var name = 'Your name is ' + first + ' ' + last + '.'

使用模板字符串：

var name = `Your name is ${first} ${last}.`

在ES6中通过${}就可以完成字符串的拼接，只需要将变量放在大括号之中。

1. 解构赋值

本质：模式匹配

完全解构：模式完全匹配

不完全解构：模式不完全匹配

- 为解构变量设置默认值，不会出现 undefined 的现象

- 解构成功时，解构变量的默认值会被覆盖

- 解构不成功时，解构变量的值为默认值

对象的解构注意：

对象在解构时，变量名要与属性名一致

对象解构的解构变量不考虑顺序

对象在解构时，为对象属性重命名，可以方便程序的编写。注：重命名不会更改对象的属性

和解构数组一样，解析对象时可以设置默认值 outLookURL:url = 111

数组解构

让我们一起先来看数组解构的基本用法：

let [a, b, c] = [1, 2, 3] // a=1, b=2, c=3let [d, [e], f] = [1, [2], 3] // 嵌套数组解构 d=1, e=2, f=3let [g, ...h] = [1, 2, 3] // 数组拆分 g=1, h=[2, 3]let [i,,j] = [1, 2, 3] // 不连续解构 i=1, j=3let [k,l] = [1, 2, 3] // 不完全解构 k=1, l=2

上面的例子包含了数组解构时常用的基本用法

对象解构

接下来再让我们一起看看对象解构的基本用法：

let {a, b} = {a: 'aaaa', b: 'bbbb'} // a='aaaa' b='bbbb'let obj = {d: 'aaaa', e: {f: 'bbbb'}}let {d, e:{f}} = obj // 嵌套解构 d='aaaa' f='bbbb'let g;

(g = {g: 'aaaa'}) // 以声明变量解构 g='aaaa'let [h, i, j, k] = 'nice' // 字符串解构 h='n' i='i' j='c' k='e'

使用场景

变量赋值

我们先来看最基本的使用场景：变量赋值，先来看我们在平时开发中是怎么使用es5对变量赋值的：

var data = {userName: 'aaaa', password: 123456}var userName = data.userNamevar password = data.passwordconsole.log(userName)console.log(password)var data1 = ['aaaa', 123456]var userName1 = data1[0]var password1 = data1[1]console.log(userName1)console.log(password1)

上面两个例子是最简单的例子，用传统es5变量赋值，然后调用，这么写的问题就是显得代码啰嗦，明明一行可以搞定的事情非要用三行代码，来看看解构赋值是怎么干的：

const {userName, password} = {userName: 'aaaa', password: 123456}console.log(userName)console.log(password)const [userName1, password1] = ['aaaa', 123456]console.log(userName1)console.log(password1)

相对于es5的语法是不是更加简单明了，在数据量越大用解构赋值的优势越明显

函数参数的定义

一般我们在定义函数的时候，如果函数有多个参数时，在es5语法中函数调用时参数必须一一对应，否则就会出现赋值错误的情况，来看一个例子：

function personInfo(name, age, address, gender) {

console.log(name, age, address, gender)

}

personInfo('william', 18, 'changsha', 'man')

上面这个例子在对用户信息的时候需要传递四个参数，且需要一一对应，这样就会极易出现参数顺序传错的情况，从而导致bug，接下来来看es6解构赋值是怎么解决这个问题的：

function personInfo({name, age, address, gender}) {

console.log(name, age, address, gender)

}

personInfo({gender: 'man', address: 'changsha', name: 'william', age: 18})

这么写我们只需要知道要传什么参数就行来，不需要知道参数的顺序也没问题

交换变量的值

在es5中我们需要交换两个变量的值需要借助临时变量的帮助，来看一个例子：

var a=1, b=2, c

c = a

a = b

b = cconsole.log(a, b)

来看es6怎么实现：

let a=1, b=2;

[b, a] = [a, b]console.log(a, b)

是不是比es5的写法更加方便呢

函数的默认参数

在日常开发中，经常会有这种情况：函数的参数需要默认值，如果没有默认值在使用的时候就会报错，来看es5中是怎么做的：

function saveInfo(name, age, address, gender) {

name = name || 'william'

age = age || 18

address = address || 'changsha'

gender = gender || 'man'

console.log(name, age, address, gender)

}

saveInfo()

在函数离 main先对参数做一个默认值赋值，然后再使用避免使用的过程中报错，再来看es6中的使用的方法：

function saveInfo({name= 'william', age= 18, address= 'changsha', gender= 'man'} = {}) {

console.log(name, age, address, gender)

}

saveInfo()

在函数定义的时候就定义了默认参数，这样就免了后面给参数赋值默认值的过程，是不是看起来简单多了

1. 延展操作符(Spread operator)

延展操作符...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；还可以在构造对象时, 将对象表达式按key-value的方式展开。

语法

函数调用：

myFunction(...iterableObj);

数组构造或字符串：

[...iterableObj, '4', ...'hello', 6];

构造对象时,进行克隆或者属性拷贝（ECMAScript 2018规范新增特性）：

let objClone = { ...obj };

应用场景

在函数调用时使用延展操作符

function sum(x, y, z) {

return x + y + z;

}const numbers = [1, 2, 3];

//不使用延展操作符console.log(sum.apply(null, numbers));

//使用延展操作符console.log(sum(...numbers));// 6

构造数组

没有展开语法的时候，只能组合使用 push，splice，concat 等方法，来将已有数组元素变成新数组的一部分。有了展开语法, 构造新数组会变得更简单、更优雅：

const stuendts = ['Jine','Tom']; const persons = ['Tony',... stuendts,'Aaron','Anna'];

conslog.log(persions)// ["Tony", "Jine", "Tom", "Aaron", "Anna"]

和参数列表的展开类似, ... 在构造字数组时, 可以在任意位置多次使用。

数组拷贝

var arr = [1, 2, 3];var arr2 = [...arr]; // 等同于 arr.slice()

arr2.push(4); console.log(arr2)//[1, 2, 3, 4]

展开语法和 Object.assign() 行为一致, 执行的都是浅拷贝(只遍历一层)。

连接多个数组

var arr1 = [0, 1, 2];var arr2 = [3, 4, 5];var arr3 = [...arr1, ...arr2];// 将 arr2 中所有元素附加到 arr1 后面并返回//等同于var arr4 = arr1.concat(arr2);

在ECMAScript 2018中延展操作符增加了对对象的支持

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 }

在React中的应用

通常我们在封装一个组件时，会对外公开一些 props 用于实现功能。大部分情况下在外部使用都应显示的传递 props 。但是当传递大量的props时，会非常繁琐，这时我们可以使用 ...(延展操作符,用于取出参数对象的所有可遍历属性) 来进行传递。

一般情况下我们应该这样写

<CustomComponent name ='Jine' age ={21} />

使用 ... ，等同于上面的写法

const params = {

name: 'Jine',

age: 21

}

<CustomComponent {...params} />

配合解构赋值避免传入一些不需要的参数

var params = {

name: '123',

title: '456',

type: 'aaa'

}

var { type, ...other } = params;

<CustomComponent type='normal' number={2} {...other} />

//等同于

<CustomComponent type='normal' number={2} name='123' title='456' />

1. 对象属性简写

在ES6中允许我们在设置一个对象的属性的时候不指定属性名。

不使用ES6

const name='Ming',age='18',city='Shanghai';

const student = {

name:name,

age:age,

city:city

};console.log(student);//{name: "Ming", age: "18", city: "Shanghai"}

对象中必须包含属性和值，显得非常冗余。

使用ES6

const name='Ming',age='18',city='Shanghai';

const student = {

name,

age,

city

};console.log(student);//{name: "Ming", age: "18", city: "Shanghai"}

对象中直接写变量，非常简洁。

1. Promise

Promise 是异步编程的一种解决方案，比传统的解决方案callback更加的优雅。它最早由社区提出和实现的，ES6 将其写进了语言标准，统一了用法，原生提供了Promise对象。

不使用ES6

嵌套两个setTimeout回调函数：

setTimeout(function(){

console.log('Hello'); // 1秒后输出"Hello"

setTimeout(function()

{

console.log('Hi'); // 2秒后输出"Hi"

}, 1000);

}, 1000);

使用ES6

var waitSecond = new Promise(function(resolve, reject){

setTimeout(resolve, 1000);

});

waitSecond

.then(function()

{

console.log("Hello"); // 1秒后输出"Hello"

return waitSecond;

})

.then(function()

{

console.log("Hi"); // 2秒后输出"Hi"

});

上面的的代码使用两个then来进行异步编程串行化，避免了回调地狱：

1. 支持let与const

ES5中声明变量的方式：

//1.通过var声明var num;

//2.函数方式声明function fn(num){ return num; }fn(10);

ES6中声明变量的方式：

//1.使用let声明let a = 10;

//2.使用const声明const name = "小红";

var ，let ， const 的区别：

不存在变量提升

var 命令会发生变量提升现象，即变量可以在声明之前使用，值为undefined。

let 和 const 则没有变量声明提升的功能，必须要先声明才能使用

不允许重复声明

var命令能重复声明，后者覆盖前者

let 和 const不允许在相同作用域内，重复声明同一个变量

作用域

var 的作用域是以函数为界限

let 和 const 的作用域是块作用域，块级作用域指 { } 内的范围

var 可以定义全局变量和局部变量，let 和 const 只能定义局部变量

const 的声明的常量不能被修改，但对于引用类型来说，堆内存中的值是可以被改变的。

变量作为全局属性

定义的变量会作为window对象的属性，let不会

常量定义的引用类型可以修改，如：

//1.使用常量定义数组

const arr = [100, 200, 300];

console.log(arr);

arr[0] = "hello";

console.log(arr); //['hello', 200, 300]

//2.使用常量来定义对象

const obj = {

name: "Jack",

age: 22,

no: "001"

}

console.log(obj);

obj.age = 100;

console.log(obj); //{name: "Jack", age: 100, no: "001"}

1. 字符串遍历

let str = "hello";

//1.for遍历

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

console.log(i, str[i]); //i 索引 数值类型

}

//2.数组->for->for in

let arr = [1, 2, 3];

for (let i in arr) {

console.log(i, arr[i]); //i 索引 字符串类型

}

//3.for... of

for(let i of str){

console.log(i); //数据

}

//4.解构

let [a, b, c, d ,e] = str;

console.log(a, b, c, d ,e);

1. Symbol

ES5 的对象属性名都是字符串，这容易造成属性名的冲突。比如，你使用了一个他人提供的对象，但又想为这个对象添加新的方法（mixin 模式），新方法的名字就有可能与现有方法产生冲突。如果有一种机制，保证每个属性的名字都是独一无二的就好了，这样就从根本上防止属性名的冲突。这就是 ES6 引入Symbol的原因。

ES6 引入了一种新的原始数据类型Symbol，表示独一无二的值。它是 JavaScript 语言的第七种数据类型，前六种是：undefined、null、布尔值（Boolean）、字符串（String）、数值（Number）、对象（Object）。

Symbol 值通过Symbol函数生成。这就是说，对象的属性名现在可以有两种类型，一种是原来就有的字符串，另一种就是新增的 Symbol 类型。凡是属性名属于 Symbol 类型，就都是独一无二的，可以保证不会与其他属性名产生冲突。

let s = Symbol();

typeof s

// "symbol"

上面代码中，变量s就是一个独一无二的值。typeof运算符的结果，表明变量s是 Symbol 数据类型，而不是字符串之类的其他类型。

注意，Symbol函数前不能使用new命令，否则会报错。这是因为生成的 Symbol 是一个原始类型的值，不是对象。也就是说，由于 Symbol 值不是对象，所以不能添加属性。基本上，它是一种类似于字符串的数据类型。

Symbol函数可以接受一个字符串作为参数，表示对 Symbol 实例的描述，主要是为了在控制台显示，或者转为字符串时，比较容易区分。

let s1 = Symbol('foo');let s2 = Symbol('bar');

s1 // Symbol(foo)s2 // Symbol(bar)

s1.toString() // "Symbol(foo)"s2.toString() // "Symbol(bar)"

上面代码中，s1和s2是两个 Symbol 值。如果不加参数，它们在控制台的输出都是Symbol()，不利于区分。有了参数以后，就等于为它们加上了描述，输出的时候就能够分清，到底是哪一个值。

如果 Symbol 的参数是一个对象，就会调用该对象的toString方法，将其转为字符串，然后才生成一个 Symbol 值。

const obj = {

toString() {

return 'abc';

}};

const sym = Symbol(obj);

sym // Symbol(abc)

注意，Symbol函数的参数只是表示对当前 Symbol 值的描述，因此相同参数的Symbol函数的返回值是不相等的。

// 没有参数的情况let s1 = Symbol();let s2 = Symbol();

s1 === s2 // false

// 有参数的情况let s1 = Symbol('foo');let s2 = Symbol('foo');

s1 === s2 // false

上面代码中，s1和s2都是Symbol函数的返回值，而且参数相同，但是它们是不相等的。

Symbol 值不能与其他类型的值进行运算，会报错。

let sym = Symbol('My symbol');

"your symbol is " + sym

// TypeError: can't convert symbol to string`your symbol is ${sym}`

// TypeError: can't convert symbol to string

但是，Symbol 值可以显式转为字符串。

let sym = Symbol('My symbol');

String(sym) // 'Symbol(My symbol)'sym.toString() // 'Symbol(My symbol)'

另外，Symbol 值也可以转为布尔值，但是不能转为数值。

let sym = Symbol();Boolean(sym) // true!sym // falseif (sym) {

// ...}

Number(sym) // TypeErrorsym + 2 // TypeError

1. Generator

Generator 函数是 ES6 提供的一种异步编程解决方案，语法行为与传统函数完全不同。本章详细介绍 Generator 函数的语法和 API，它的异步编程应用请看《Generator 函数的异步应用》一章。

Generator 函数有多种理解角度。语法上，首先可以把它理解成，Generator 函数是一个状态机，封装了多个内部状态。

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。

形式上，Generator 函数是一个普通函数，但是有两个特征。一是，function关键字与函数名之间有一个星号；二是，函数体内部使用yield表达式，定义不同的内部状态（yield在英语里的意思就是“产出”）。

function\* helloWorldGenerator() {

yield 'hello';

yield 'world';

return 'ending';}

var hw = helloWorldGenerator();

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

然后，Generator 函数的调用方法与普通函数一样，也是在函数名后面加上一对圆括号。不同的是，调用 Generator 函数后，该函数并不执行，返回的也不是函数运行结果，而是一个指向内部状态的指针对象，也就是上一章介绍的遍历器对象（Iterator Object）。

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

hw.next()

// { value: 'hello', done: false }

hw.next()

// { value: 'world', done: false }

hw.next()

// { value: 'ending', done: true }

hw.next()

// { value: undefined, done: true }

上面代码一共调用了四次next方法。

第一次调用，Generator 函数开始执行，直到遇到第一个yield表达式为止。next方法返回一个对象，它的value属性就是当前yield表达式的值hello，done属性的值false，表示遍历还没有结束。

第二次调用，Generator 函数从上次yield表达式停下的地方，一直执行到下一个yield表达式。next方法返回的对象的value属性就是当前yield表达式的值world，done属性的值false，表示遍历还没有结束。

第三次调用，Generator 函数从上次yield表达式停下的地方，一直执行到return语句（如果没有return语句，就执行到函数结束）。next方法返回的对象的value属性，就是紧跟在return语句后面的表达式的值（如果没有return语句，则value属性的值为undefined），done属性的值true，表示遍历已经结束。

第四次调用，此时 Generator 函数已经运行完毕，next方法返回对象的value属性为undefined，done属性为true。以后再调用next方法，返回的都是这个值。

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。

ES6 没有规定，function关键字与函数名之间的星号，写在哪个位置。这导致下面的写法都能通过。

function \* foo(x, y) { ··· }function \*foo(x, y) { ··· }function\* foo(x, y) { ··· }function\*foo(x, y) { ··· }

由于 Generator 函数仍然是普通函数，所以一般的写法是上面的第三种，即星号紧跟在function关键字后面。本书也采用这种写法。

1. Set 和 Map 数据结构

ES5 的对象属性名都是字符串，这容易造成属性名的冲突。比如，你使用了一个他人提供的对象，但又想为这个对象添加新的方法（mixin 模式），新方法的名字就有可能与现有方法产生冲突。如果有一种机制，保证每个属性的名字都是独一无二的就好了，这样就从根本上防止属性名的冲突。这就是 ES6 引入Symbol的原因。ES6 提供了新的数据结构 Set。它类似于数组，但是成员的值都是唯一的，没有重复的值。

Set本身是一个构造函数，用来生成 Set 数据结构。

const s = new Set();

[2, 3, 5, 4, 5, 2, 2].forEach(x => s.add(x));

for (let i of s) {

console.log(i);}

// 2 3 5 4

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

Set函数可以接受一个数组（或者具有 iterable 接口的其他数据结构）作为参数，用来初始化。

// 例一const set = new Set([1, 2, 3, 4, 4]);[...set]

// [1, 2, 3, 4]

// 例二const items = new Set([1, 2, 3, 4, 5, 5, 5, 5]);

items.size // 5

// 例三const set = new Set(document.querySelectorAll('div'));set.size // 56

// 类似于const set = new Set();

document

.querySelectorAll('div')

.forEach(div => set.add(div));set.size // 56

上面代码中，例一和例二都是Set函数接受数组作为参数，例三是接受类似数组的对象作为参数。

上面代码也展示了一种去除数组重复成员的方法。

// 去除数组的重复成员[...new Set(array)]

上面的方法也可以用于，去除字符串里面的重复字符。

[...new Set('ababbc')].join('')

// "abc"

JavaScript 的对象（Object），本质上是键值对的集合（Hash 结构），但是传统上只能用字符串当作键。这给它的使用带来了很大的限制。

const data = {};

const element = document.getElementById('myDiv');

data[element] = 'metadata';

data['[object HTMLDivElement]'] // "metadata"

上面代码原意是将一个 DOM 节点作为对象data的键，但是由于对象只接受字符串作为键名，所以element被自动转为字符串[object HTMLDivElement]。

为了解决这个问题，ES6 提供了 Map 数据结构。它类似于对象，也是键值对的集合，但是“键”的范围不限于字符串，各种类型的值（包括对象）都可以当作键。也就是说，Object 结构提供了“字符串—值”的对应，Map 结构提供了“值—值”的对应，是一种更完善的 Hash 结构实现。如果你需要“键值对”的数据结构，Map 比 Object 更合适。

const m = new Map();

const o = {p: 'Hello World'};

m.set(o, 'content')

m.get(o) // "content"

m.has(o) // truem.delete(o) // truem.has(o) // false

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

上面的例子展示了如何向 Map 添加成员。作为构造函数，Map 也可以接受一个数组作为参数。该数组的成员是一个个表示键值对的数组。

1. ES7新特性（2016）

ES2016添加了两个小的特性来说明标准化过程：

1. 数组includes方法

数组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。

Array.prototype.includes()

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

includes 函数与 indexOf 函数很相似，下面两个表达式是等价的：

arr.includes(x)

arr.indexOf(x) >= 0

接下来我们来判断数字中是否包含某个元素：

在ES7之前的做法

使用indexOf()验证数组中是否存在某个元素，这时需要根据返回值是否为-1来判断：

let arr = ['react', 'angular', 'vue'];

if (arr.indexOf('react') !== -1)

{

console.log('react存在');

}

注意：

使用的都是 === 操作符来做值的比较。但是includes()方法有一点不同，两个NaN被认为是相等的，即使在NaN === NaN结果是false的情况下，indexOf()严格使用===判断。

let demo = [1, NaN, 2, 3]

demo.indexOf(NaN) //-1

demo.includes(NaN) //true

includes()还有一个怪异的点需要指出，在判断 +0 与 -0 时，被认为是相同的。在这一点上，indexOf()与includes()的处理结果是一样的，同样会返回 +0 的索引值。

[1, +0, 3, 4].includes(-0) //true[1, +0, 3, 4].indexOf(-0) //1

使用ES7的includes()

使用includes()验证数组中是否存在某个元素，这样更加直观简单：

let arr = ['react', 'angular', 'vue'];

if (arr.includes('react'))

{

console.log('react存在');

}

1. 指数操作符

在ES7中引入了指数运算符\*\*，\*\*具有与Math.pow(..)等效的计算结果。

不使用指数操作符

使用自定义的递归函数calculateExponent或者Math.pow()进行指数运算：

function calculateExponent(base, exponent){

if (exponent === 1)

{

return base;

}

else

{

return base \* calculateExponent(base, exponent - 1);

}

}

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

使用指数操作符

使用指数运算符\*\*，就像+、-等操作符一样：

console.log(2\*\*10);// 输出1024

1. ES8新特性（2017）
2. async/await

谷歌中默认启用异步函数，我们可以利用 async/await 像编写同步代码一样的编写基于Promise的代码，当您使用await某个Promise 时，函数暂停执行，直到Promise执行返回结果，这种暂停不会阻塞主线程，如果Promise执行，则返回值，如果拒绝，则返回错误值。

async/await 会提高代码的可读行，去掉层层回调。

function logFetch(url) {

return fetch(url)

.then(response => response.text())

.then(text => {

console.log(text);

}).catch(err => {

console.error('fetch failed', err);

});}

使用async/await改写：

async function logFetch(url) {

try {

const response = await fetch(url);

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

}

catch (err) {

console.log('fetch failed', err);

}}

但是需要注意通过babel转译后仍旧本质是Promise，await使用的目标也必须返回一个Promise，如果要自己写异步函数，那么还是要使用Promise的。

1. Object.values()

Object.values()是一个与Object.keys()类似的新函数，但返回的是Object自身属性的所有值，不包括继承的值。

假设我们要遍历如下对象obj的所有值：

const obj = {a: 1, b: 2, c: 3};

不使用Object.values() :ES7

const vals=Object.keys(obj).map(key=>obj[key]);console.log(vals);//[1, 2, 3]

使用Object.values() :ES8

const values=Object.values(obj1);console.log(values);//[1, 2, 3]

从上述代码中可以看出Object.values()为我们省去了遍历key，并根据这些key获取value的步骤。

1. Object.entries()

Object.entries()函数返回一个给定对象自身可枚举属性的键值对的数组。

接下来我们来遍历上文中的obj对象的所有属性的key和value：

不使用Object.entries() :ES7

Object.keys(obj).forEach(key=>{

console.log('key:'+key+' value:'+obj[key]);

})//key:a value:1//key:b value:2//key:c value:3

使用Object.entries() :ES8

for(let [key,value] of Object.entries(obj1)){

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

}//key:a value:1//key:b value:2//key:c value:3

1. String padding

在ES8中String新增了两个实例函数String.prototype.padStart和String.prototype.padEnd，允许将空字符串或其他字符串添加到原始字符串的开头或结尾。

String.padStart(targetLength,[padString])

targetLength:当前字符串需要填充到的目标长度。如果这个数值小于当前字符串的长度，则返回当前字符串本身。

padString:(可选)填充字符串。如果字符串太长，使填充后的字符串长度超过了目标长度，则只保留最左侧的部分，其他部分会被截断，此参数的缺省值为 " "。

console.log('0.0'.padStart(4,'10')) //10.0console.log('0.0'.padStart(20))// 0.00

String.padEnd(targetLength,padString])

targetLength:当前字符串需要填充到的目标长度。如果这个数值小于当前字符串的长度，则返回当前字符串本身。

padString:(可选) 填充字符串。如果字符串太长，使填充后的字符串长度超过了目标长度，则只保留最左侧的部分，其他部分会被截断，此参数的缺省值为 " "；

console.log('0.0'.padEnd(4,'0')) //0.00 console.log('0.0'.padEnd(10,'0'))//0.00000000

'es8'.padStart(2); // 'es8'

'es8'.padStart(5); // ' es8'

'es8'.padStart(6, 'woof'); // 'wooes8'

'es8'.padStart(14, 'wow'); // 'wowwowwowwoes8'

'es8'.padStart(7, '0'); // '0000es8'

'es8'.padEnd(2); // 'es8'

'es8'.padEnd(5); // 'es8 '

'es8'.padEnd(6, 'woof'); // 'es8woo'

'es8'.padEnd(14, 'wow'); // 'es8wowwowwowwo'

'es8'.padEnd(7, '6'); // 'es86666'

使用 padStart 进行时间格式化。

'8:00'.padStart(5, '0'); // '08:00'

'18:00'.padStart(5, '0'); // '18:00'

'12'.padStart(10, 'YYYY-MM-DD') // "YYYY-MM-12"

'09-12'.padStart(10, 'YYYY-MM-DD') // "YYYY-09-12"

使用 padStart 给命令行输出信息对齐。

Commands:

run Start a front service

start Start a background service

stop Stop current background service

restart Restart current background service

help Display help information

1. 结尾允许逗号

主要作用是方便使用git进行多人协作开发时修改同一个函数减少不必要的行变更。或者数组或对象每行一个元素写法时更方便开发者阅读。

function es8(var1, var2, var3,) {

console.log(arguments.length); // 3

}

es8(10, 20, 30,);

不止函数，数组及对象也可以在尾部添加逗号，并不会因此产生空元素或属性。

{

a:1,

b:2,

c:3,

};

[

1,

2,

3,

4,

5,

];

1. Object.getOwnPropertyDescriptors()

Object.getOwnPropertyDescriptors()函数用来获取一个对象的所有自身属性的描述符,如果没有任何自身属性，则返回空对象。

函数原型：

Object.getOwnPropertyDescriptors(obj)

返回obj对象的所有自身属性的描述符，如果没有任何自身属性，则返回空对象。

const obj2 = {

name: 'Jine',

get age() { return '18' }

};

Object.getOwnPropertyDescriptors(obj2)

// {// age: {// configurable: true,// enumerable: true,// get: function age(){}, //the getter function// set: undefined// },// name: {// configurable: true,// enumerable: true,// value:"Jine",// writable:true// }// }

1. SharedArrayBuffer对象

在了解SharedArrayBuffer之前我们首先要了解一下内存之类的知识。

假如我们要将一个数字放进内存中，我们会首先将它转化为32位或者64位字大小，如果要放入一个非数字，JS Engine会通过编码器运行该值，然后通过编码方式，例如UTF-8,获取该值得二进制表示。JS引擎会在内存中找到可以存放此二进制的位置，进行分配内存，之后js引擎会持续跟踪该变量是否仍可从程序中的任何位置访问。如果无法再访问该变量，则将回收内存，以便JS引擎可以在其中放置新值。如果无法在访问到它们，则进行清除，此过程称为垃圾回收。JS类语言称为内存管理语言，它并不会直接操作内存，自动管理内存会使管理人员更加轻松，但是会产生一定的性能开销。

但是，例如C等手动管理内存语言，C没有JavaScript在内存上做的那个抽象层。相反，你直接在内存上运行。您可以从内存加载内容，并可以将内容存储到内存中。假如React使用C写出来，那么它可以借助WebAssembly来进行内存管理，关于什么是WebAssembly？

WebAssembly是一种新的编码方式，可以在现代的网络浏览器中运行 － 它是一种低级的类汇编语言，具有紧凑的二进制格式，可以接近原生的性能运行，并为诸如C / C ++等语言提供一个编译目标，以便它们可以在Web上运行。它也被设计为可以与JavaScript共存，允许两者一起工作。对于网络平台而言，WebAssembly具有巨大的意义——它提供了一条途径，以使得以各种语言编写的代码都可以以接近原生的速度在Web中运行。在这种情况下，以前无法以此方式运行的客户端软件都将可以运行在Web中。

WebAssembly被设计为可以和JavaScript一起协同工作——通过使用WebAssembly的JavaScript API，你可以把WebAssembly模块加载到一个JavaScript应用中并且在两者之间共享功能。这允许你在同一个应用中利用WebAssembly的性能和威力以及JavaScript的表达力和灵活性，即使你可能并不知道如何编写WebAssembly代码。

那么，我们为什么需要ArrayBuffers？

ArrayBuffer 对象用来表示通用的、固定长度的原始二进制数据缓冲区。ArrayBuffer 不能直接操作，而是要通过类型数组对象或[DataView](https://link.zhihu.com/?target=https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/DataView" \t "/Users/lichddd/Documents\\x/_blank)对象来操作，它们会将缓冲区中的数据表示为特定的格式，并通过这些格式来读写缓冲区的内容。

即使你在JS中也可以通过ArrayBuffer对内存数据进行处理，你为什么需要进行处理？正如前面所说的自动内存管理语言对于内存处理都有一种权衡，增加一些开销，某种程度会导致性能问题。JS对于新创建的变量，因为引擎会对其进行分析，所以可能需要要为它进行预留大约2倍以上内存空间，这可能导致内存大量的浪费。

var buffer = new ArrayBuffer(8);

console.log(buffer.byteLength);// expected output: 8

除了使用ArrayBuffer时，您不能将任何JavaScript类型放入其中，如对象或字符串。您可以添加的唯一内容是字节，实际上并没有将这个字节直接添加到ArrayBuffer中。就其本身而言，这个ArrayBuffer不知道该字节应该有多大，或者不应该将不同类型的数字转换为字节。ArrayBuffer本身只是一堆零和一行。ArrayBuffer不知道该数组中第一个元素和第二个元素之间的除法位置。

为什么我们需要SharedArrayBuffer？

ArrayBuffers可以减少主线程必须完成的工作量。更加高效的做法是分开进行工作，在其他语言中使用线程来进行分解工作，在JS中使用Web Worker，但是他们不像其他语言一样，它们是不共享内存的，也就是你干你的，我干我的。如果我们想要共享某些数据就必须进行复制，通过函数[postMessage](https://link.zhihu.com/?target=https://developer.mozilla.org/en-US/docs/Web/API/Worker/postMessage" \t "/Users/lichddd/Documents\\x/_blank)来进行。postMessage接受放入的其他对象，对其进行序列化，将其发送给其他Web工作者，然后将其反序列化并放入内存中。对于某些类型的数据，如ArrayBuffers，您可以执行所谓的传输内存。这意味着移动特定的内存块，以便其他Web工作者可以访问它。但是第一个Web工作者再也无法访问它了。

使用SharedArrayBuffer，两个Web工作者（两个线程）都可以写入数据并从同一块内存中读取数据。这意味着他们没有使用postMessage的通信开销和延迟。两个Web工作人员都可以立即访问数据。但是，同时从两个线程立即访问存在一些危险。它可以导致所谓的竞争条件。

1. Atomics对象

Atomics 对象提供了一组静态方法用来对 SharedArrayBuffer 对象进行原子操作。

这些原子操作属于 Atomics 模块。与一般的全局对象不同，Atomics 不是构造函数，因此不能使用 new 操作符调用，也不能将其当作函数直接调用。Atomics 的所有属性和方法都是静态的（与 Math 对象一样）。

多个共享内存的线程能够同时读写同一位置上的数据。原子操作会确保正在读或写的数据的值是符合预期的，即下一个原子操作一定会在上一个原子操作结束后才会开始，其操作过程不会中断。

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。

wait() 和 wake() 方法采用的是 Linux 上的 futexes 模型（fast user-space mutex，快速用户空间互斥量），可以让进程一直等待直到某个特定的条件为真，主要用于实现阻塞。

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。

1. 类的静态变量

虽然在ES6就有类的实现，但是仍旧与传统的面向对象语言的类差别很大，特别是没有静态变量。

在ES8的规范中出现了静态变量的设计，通过这个特性可以方便实现一些以前无法方便实现的功能。

但是此新特性并未在很多网上的资料中显示，需要特别注意

class Animal {

static name=’animal’;

constructor() {

}

}

var animal = new Animal();//实例化Animal

animal.name;//这里会报错，无此属性

Animal.name;//animal

Animal.name=’dog’;

Animal.name;//dog

1. ES9新特性（2018）
2. 异步迭代

在async/await的某些时刻，你可能尝试在同步循环中调用异步函数。例如：

async function process(array) {

for (let i of array) {

await doSomething(i);

}

}

这段代码不会正常运行，下面这段同样也不会：

async function process(array) {

array.forEach(async i => {

await doSomething(i);

});

}

这段代码中，循环本身依旧保持同步，并在在内部异步函数之前全部调用完成。

ES2018引入异步迭代器（asynchronous iterators），这就像常规迭代器，除了next()方法返回一个Promise。因此await可以和for...of循环一起使用，以串行的方式运行异步操作。例如：

async function process(array) {

for await (let i of array) {

doSomething(i);

}

}

1. 2.Promise.finally()

一个Promise调用链要么成功到达最后一个.then()，要么失败触发.catch()。在某些情况下，你想要在无论Promise运行成功还是失败，运行相同的代码，例如清除，删除对话，关闭数据库连接等。

.finally()允许你指定最终的逻辑：

function doSomething() {

doSomething1()

.then(doSomething2)

.then(doSomething3)

.catch(err => {

console.log(err);

})

.finally(() => {

// finish here!

});

}

1. 3.Rest/Spread 属性

ES2015引入了Rest参数和扩展运算符。三个点（...）仅用于数组。Rest参数语法允许我们将一个不定数量的参数表示为一个数组。

restParam(1, 2, 3, 4, 5);

function restParam(p1, p2, ...p3) {

// p1 = 1

// p2 = 2

// p3 = [3, 4, 5]

}

展开操作符以相反的方式工作，将数组转换成可传递给函数的单独参数。例如Math.max()返回给定数字中的最大值：

const values = [99, 100, -1, 48, 16];

console.log( Math.max(...values) ); // 100

ES2018为对象解构提供了和数组一样的Rest参数（）和展开操作符，一个简单的例子：

const myObject = {

a: 1,

b: 2,

c: 3

};

const { a, ...x } = myObject;// a = 1// x = { b: 2, c: 3 }

或者你可以使用它给函数传递参数：

restParam({

a: 1,

b: 2,

c: 3

});

function restParam({ a, ...x }) {

// a = 1

// x = { b: 2, c: 3 }

}

跟数组一样，Rest参数只能在声明的结尾处使用。此外，它只适用于每个对象的顶层，如果对象中嵌套对象则无法适用。

扩展运算符可以在其他对象内使用，例如：

const obj1 = { a: 1, b: 2, c: 3 };const obj2 = { ...obj1, z: 26 };// obj2 is { a: 1, b: 2, c: 3, z: 26 }

可以使用扩展运算符拷贝一个对象，像是这样obj2 = {...obj1}，但是 这只是一个对象的浅拷贝。另外，如果一个对象A的属性是对象B，那么在克隆后的对象cloneB中，该属性指向对象B。

1. 4.正则表达式命名捕获组

JavaScript正则表达式可以返回一个匹配的对象——一个包含匹配字符串的类数组，例如：以YYYY-MM-DD的格式解析日期：

const

reDate = /([0-9]{4})-([0-9]{2})-([0-9]{2})/,

match = reDate.exec('2018-04-30'),

year = match[1], // 2018

month = match[2], // 04

day = match[3]; // 30

这样的代码很难读懂，并且改变正则表达式的结构有可能改变匹配对象的索引。

ES2018允许命名捕获组使用符号?<name>，在打开捕获括号(后立即命名，示例如下：

const

reDate = /(?<year>[0-9]{4})-(?<month>[0-9]{2})-(?<day>[0-9]{2})/,

match = reDate.exec('2018-04-30'),

year = match.groups.year, // 2018

month = match.groups.month, // 04

day = match.groups.day; // 30

任何匹配失败的命名组都将返回undefined。

命名捕获也可以使用在replace()方法中。例如将日期转换为美国的 MM-DD-YYYY 格式：

const

reDate = /(?<year>[0-9]{4})-(?<month>[0-9]{2})-(?<day>[0-9]{2})/,

d = '2018-04-30',

usDate = d.replace(reDate, '$<month>-$<day>-$<year>');

1. 5.正则表达式反向断言

目前JavaScript在正则表达式中支持先行断言（lookahead）。这意味着匹配会发生，但不会有任何捕获，并且断言没有包含在整个匹配字段中。例如从价格中捕获货币符号：

const

reLookahead = /\D(?=\d+)/,

match = reLookahead.exec('$123.89');

console.log( match[0] ); // $

ES2018引入以相同方式工作但是匹配前面的反向断言（lookbehind），这样我就可以忽略货币符号，单纯的捕获价格的数字：

const

reLookbehind = /(?<=\D)\d+/,

match = reLookbehind.exec('$123.89');

console.log( match[0] ); // 123.89

以上是 肯定反向断言，非数字\D必须存在。同样的，还存在 否定反向断言，表示一个值必须不存在，例如：

const

reLookbehindNeg = /(?<!\D)\d+/,

match = reLookbehind.exec('$123.89');

console.log( match[0] ); // null

1. 6.正则表达式dotAll模式

正则表达式中点.匹配除回车外的任何单字符，标记s改变这种行为，允许行终止符的出现，例如：

/hello.world/.test('hello\nworld'); // false

/hello.world/s.test('hello\nworld'); // true

1. 7.正则表达式 Unicode 转义

到目前为止，在正则表达式中本地访问 Unicode 字符属性是不被允许的。ES2018添加了 Unicode 属性转义——形式为\p{...}和\P{...}，在正则表达式中使用标记 u (unicode) 设置，在\p块儿内，可以以键值对的方式设置需要匹配的属性而非具体内容。例如：

const reGreekSymbol = /\p{Script=Greek}/u;

reGreekSymbol.test('π'); // true

此特性可以避免使用特定 Unicode 区间来进行内容类型判断，提升可读性和可维护性。

1. 8.非转义序列的模板字符串

之前，\u开始一个 unicode 转义，\x开始一个十六进制转义，\后跟一个数字开始一个八进制转义。这使得创建特定的字符串变得不可能，例如Windows文件路径 C:\uuu\xxx\111。更多细节参考模板字符串。

1. ES10新特性（2019）
2. 1.行分隔符（U + 2028）和段分隔符（U + 2029）符号现在允许在字符串文字中，与JSON匹配

以前，这些符号在字符串文字中被视为行终止符，因此使用它们会导致SyntaxError异常。

1. 2.更加友好的 JSON.stringify

如果输入 Unicode 格式但是超出范围的字符，在原先JSON.stringify返回格式错误的Unicode字符串。现在实现了一个改变JSON.stringify的第3阶段提案，因此它为其输出转义序列，使其成为有效Unicode（并以UTF-8表示）

1. 3.新增了Array的flat()方法和flatMap()方法

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。flat()方法主要作用有两个方面：

扁平化数组，并且可以指定扁平化的层级depth

移除数组中的空项

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

arr1.flat(); // [1, 2, 3, 4]

var arr2 = [1, 2, [3, 4, [5, 6]]];

arr2.flat();// [1, 2, 3, 4, [5, 6]]

var arr3 = [1, 2, [3, 4, [5, 6]]];

arr3.flat(2);// [1, 2, 3, 4, 5, 6]//使用 Infinity 作为深度，展开任意深度的嵌套数组arr3.flat(Infinity); // [1, 2, 3, 4, 5, 6]// 移除空项

var arr4 = [1, 2, , 4, 5];

arr4.flat();// [1, 2, 4, 5]

这里涉及到一个前端面试题，如何扁平化数组？在没有api支持之前，我们是这样做的：

var arr1 = [1, 2, [3, 4]];arr1.flat();方法一：// 反嵌套一层数组

arr1.reduce((acc, val) => acc.concat(val), []);// [1, 2, 3, 4]

//方法二:或使用

...const flatSingle = arr => [].concat(...arr);

//方法三：递归

function flatten(arr) {

var result = [];

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

if (Array.isArray(arr[i])) {

result = result.concat(flatten(arr[i]))

}

else {

result.push(arr[i])

}

}

return result;}方法四：toString()function flatten(arr) {

return arr.toString().split(',').map(function(item){

return +item

})}

flatMap()

flatMap() 方法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。它与[map](https://link.zhihu.com/?target=https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map" \t "/Users/lichddd/Documents\\x/_blank) 和 深度值1的 [flat](https://link.zhihu.com/?target=https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/flat" \t "/Users/lichddd/Documents\\x/_blank) 几乎相同，但 flatMap通常在合并成一种方法的效率稍微高一些。 返回一个新的数组，其中每个元素都是回调函数的结果，并且结构深度depth值为1。

注意：只会将 flatMap 中的函数返回的数组 “压平” 一层

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

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

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

// 只会将 flatMap 中的函数返回的数组 “压平” 一层

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

let arr = ["今天天气不错", "", "早上好"]

arr.map(s => s.split(""))// [["今", "天", "天", "气", "不", "错"],[""],["早", "上", "好"]]

arr.flatMap(s => s.split(''));// ["今", "天", "天", "气", "不", "错", "", "早", "上", "好"]

1. 4.新增了String的trimStart()方法和trimEnd()方法

String.trimStart()可用于从字符串的开头去掉空白，trimLeft()是此方法的别名。注意：trimStart()方法返回的是一个新的字符串，并不改变原来的字符串。并且新字符串的长度会改变。

var text = ' Hello world! ';

console.log(text); //" Hello world! "

console.log(text.trimStart()); // "Hello world! "

console.log(text) //" Hello world! "

// Length

var str = ' foo ';

console.log(str.length); // 8

str = str.trimStart();

console.log(str.length); // 5

console.log(str); // 'foo '

String.trimEnd()可用于从字符串的尾部去掉空白。trimRight()是此方法的别名。注意：返回一个新字符串，表示从其（右）端剥去空白的调用字符串。同样，新字符串长度会改变。

var text = ' Hello world! ';

console.log(text); // " Hello world! "

console.log(text.trimEnd()); // " Hello world!"

var str = ' foo ';

console.log(str.length); // 8

str = str.trimEnd();

console.log(str.length); // 6

console.log(str); // ' foo'

1. 5.Object.fromEntries()

Object.entries()方法的作用是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 循环也枚举原型链中的属性）。

而Object.fromEntries() 则是 Object.entries() 的反转。

Object.fromEntries() 函数传入一个键值对的列表，并返回一个带有这些键值对的新对象。这个迭代参数应该是一个能够实现@iterator方法的的对象，返回一个迭代器对象。它生成一个具有两个元素的类似数组的对象，第一个元素是将用作属性键的值，第二个元素是与该属性键关联的值。

通过 Object.fromEntries， 可以将 Map 转化为 Object:

const map = new Map([ ['foo', 'bar'], ['baz', 42] ]);const obj = Object.fromEntries(map);console.log(obj); // { foo: "bar", baz: 42 }

通过 Object.fromEntries， 可以将 Array 转化为 Object:

const arr = [ ['0', 'a'], ['1', 'b'], ['2', 'c'] ];const obj = Object.fromEntries(arr);console.log(obj); // { 0: "a", 1: "b", 2: "c" }

1. 6.Symbol.prototype.description

通过工厂函数Symbol（）创建符号时，您可以选择通过参数提供字符串作为描述：

const sym = Symbol('The description');

以前，访问描述的唯一方法是将符号转换为字符串：

assert.equal(String(sym), 'Symbol(The description)');

现在引入了getter Symbol.prototype.description以直接访问描述：

assert.equal(sym.description, 'The description');

1. 7.String.prototype.matchAll

matchAll() 方法返回一个包含所有匹配正则表达式及分组捕获结果的迭代器。 在 matchAll 出现之前，通过在循环中调用regexp.exec来获取所有匹配项信息（regexp需使用/g标志：

const regexp = RegExp('foo\*','g');const str = 'table football, foosball';

while ((matches = regexp.exec(str)) !== null) {

console.log(`Found ${matches[0]}. Next starts at ${regexp.lastIndex}.`);

// expected output: "Found foo. Next starts at 9."

// expected output: "Found foo. Next starts at 19."

}

如果使用matchAll ，就可以不必使用while循环加exec方式（且正则表达式需使用／g标志）。使用matchAll 会得到一个迭代器的返回值，配合 for...of, array spread, or Array.from() 可以更方便实现功能：

const regexp = RegExp('foo\*','g'); const str = 'table football, foosball';let matches = str.matchAll(regexp);

for (const match of matches) {

console.log(match);

}// Array [ "foo" ]// Array [ "foo" ]

// matches iterator is exhausted after the for..of iteration// Call matchAll again to create a new iterator

matches = str.matchAll(regexp);

Array.from(matches, m => m[0]);// Array [ "foo", "foo" ]

matchAll可以更好的用于分组

var regexp = /t(e)(st(\d?))/g;var str = 'test1test2';

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

let array = [...str.matchAll(regexp)];

array[0];// ['test1', 'e', 'st1', '1', index: 0, input: 'test1test2', length: 4]

array[1];// ['test2', 'e', 'st2', '2', index: 5, input: 'test1test2', length: 4]

1. 8.Function.prototype.toString()现在返回精确字符，包括空格和注释

function /\* comment \*/ foo /\* another comment \*/() {}

// 之前不会打印注释部分console.log(foo.toString()); // function foo(){}

// ES2019 会把注释一同打印console.log(foo.toString()); // function /\* comment \*/ foo /\* another comment \*/ (){}

// 箭头函数const bar /\* comment \*/ = /\* another comment \*/ () => {};

console.log(bar.toString()); // () => {}

1. 9.修改 catch 绑定

在 ES10 之前，我们必须通过语法为 catch 子句绑定异常变量，无论是否有必要。很多时候 catch 块是多余的。 ES10 提案使我们能够简单的把变量省略掉。

不算大的改动。

之前是

try {} catch(e) {}

现在是

try {} catch {}

1. 10.新的基本数据类型BigInt

现在的基本数据类型（值类型）不止5种（ES6之后是六种）了哦！加上BigInt一共有七种基本数据类型，分别是： String、Number、Boolean、Null、Undefined、Symbol、BigInt

1. 部分已有实现与ES6不同的对比

除了以上新增特性介绍处部分提到的对比外，还有以下一些异同

1. 可应用部分

除非问我

1. 实际可应用的部分与不推荐使用的部分
2. 可应用部分

除了不推荐使用的特性外皆可使用，以下是重点推荐的一些特性和说明

1. 数组includes方法

除非问我

1. 不推荐使用部分

未列出的大部分特性都不推荐使用，以下是另外一些不推荐使用，或者不推荐在项目里使用的特性和说明

1. Symbol

平常编程中该特性没有特别需要使用的意义，网上的各种代码库中也很难看到它的身影，只有在极少数内核架构框架层的核心代码中出现。Generator

1. Generator

该特性用起来比较繁琐并且容易出错，在新的await/async 出现后并没有使用的意义了。

1. 新特性的特异用法，新特性的组合用法，新特性用法的最佳实践
2. 特异用法

特异用法是指使用新特性进行一些特殊的代码写法或流程，实现一些以前无法实现或者比较麻烦的功能，但是写法可能有悖常理或者有违代码书写规范。

1. 不使用if-eles按条件创建不同的数组

粉我粉我反而我

1. 不使用if-eles按条件创建不同的数组

粉我粉我反而我

1. 组合用法

组合用法是指使用多种新特性进行组合编写，实现一些以前无法实现或者比较麻烦的功能

1. Symbol
2. 最佳实践

最佳实践是指使用新特性实现功能的一些比较好的示例

1. symbol