# 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）

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。

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

解构赋值语法是JavaScript的一种表达式，可以方便的从数组或者对象中快速提取值赋给定义的变量。

获取数组中的值

从数组中获取值并赋值到变量中，变量的顺序与数组中对象顺序对应。

var foo = ["one", "two", "three", "four"];

var [one, two, three] = foo;console.log(one); // "one"console.log(two); // "two"console.log(three); // "three"

//如果你要忽略某些值，你可以按照下面的写法获取你想要的值var [first, , , last] = foo;console.log(first); // "one"console.log(last); // "four"

//你也可以这样写var a, b; //先声明变量

[a, b] = [1, 2];console.log(a); // 1console.log(b); // 2

如果没有从数组中的获取到值，你可以为变量设置一个默认值。

var a, b;

[a=5, b=7] = [1];console.log(a); // 1console.log(b); // 7

通过解构赋值可以方便的交换两个变量的值。

var a = 1;var b = 3;

[a, b] = [b, a];console.log(a); // 3console.log(b); // 1

获取对象中的值

const student = {

name:'Ming',

age:'18',

city:'Shanghai'

};

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

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

在之前JS是没有块级作用域的，const与let填补了这方便的空白，const与let都是块级作用域。

使用var定义的变量为函数级作用域：

{

var a = 10;

}

console.log(a); // 输出10

使用let与const定义的变量为块级作用域：

{

let a = 10;

}

console.log(a); //-1 or Error“ReferenceError: a is not defined”

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。

a \*\* b指数运算符，它与 Math.pow(a, b)相同。

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');

}

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

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. 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

1. 函数参数列表结尾允许逗号

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。

{

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

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。

/\*\*

\*

\* @param {\*} length 所创建的数组缓冲区的大小，以字节(byte)为单位。

\* @returns {SharedArrayBuffer} 一个大小指定的新 SharedArrayBuffer 对象。其内容被初始化为 0。

\*/new SharedArrayBuffer(length)

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. 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()和flatMap()本质上就是是归纳（reduce） 与 合并（concat）的操作。

Array.prototype.flat()

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()方法最基本的作用就是数组降维

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]

其次，还可以利用flat()方法的特性来去除数组的空项

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

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

Array.prototype.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 和 深度值1的 flat 几乎相同，但 flatMap 通常在合并成一种方法的效率稍微高一些。 这里我们拿map方法与flatMap方法做一个比较。

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]]

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

新增的这两个方法很好理解，分别去除字符串首尾空白字符，这里就不用例子说声明了。

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，例如

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. 不使用if-eles按条件创建不同的数组或对象

粉我粉我反而我