# ECMAScript和JavaScript的关系

一个常见的问题是，ECMAScript和JavaScript到底是什么关系？

要讲清楚这个问题，需要回顾历史。1996年11月，JavaScript的创造者Netscape公司，决定将JavaScript提交给国际标准化组织ECMA，希望这种语言能够成为国际标准。次年，ECMA发布262号标准文件（ECMA-262）的第一版，规定了浏览器脚本语言的标准，并将这种语言称为ECMAScript，这个版本就是1.0版。

该标准从一开始就是针对JavaScript语言制定的，但是之所以不叫JavaScript，有两个原因。一是商标，Java是Sun公司的商标，根据授权协议，只有Netscape公司可以合法地使用JavaScript这个名字，且JavaScript本身也已经被Netscape公司注册为商标。二是想体现这门语言的制定者是ECMA，不是Netscape，这样有利于保证这门语言的开放性和中立性。

因此，ECMAScript和JavaScript的关系是，前者是后者的规格，后者是前者的一种实现（另外的ECMAScript方言还有Jscript和ActionScript）。日常场合，这两个词是可以互换的。

ES6既是一个历史名词，也是一个泛指，含义是5.1版以后的JavaScript的下一代标准，涵盖了ES2015、ES2016、ES2017等等，而ES2015则是正式名称，特指该年发布的正式版本的语言标准。本书中提到“ES6”的地方，一般是指ES2015标准，但有时也是泛指“下一代JavaScript语言”。

# ES6核心

## 块级作用域

ES6中，我们使用let/const， 而不是var 声明变量。

{

let a = 10;

var b = 1;

}

常用:for循环的计数器，就很合适使用let命令。

经典：

/变量i是let声明的，当前的i只在本轮循环有效，

// 所以每一次循环的i其实都是一个新的变量，所以最后输出的是6。

var a = [];

for (let i = 0; i < 10; i++) {

a[i] = function () {

console.log(i);

};

}

a[6](); // 6

a // ReferenceError: a is not defined.

b // 1

let变量提升：

let不像var那样会发生“变量提升”现象。所以，变量一定要在声明后使用，否则报错。

let不允许在相同作用域内，重复声明同一个变量。

var 有什么缺点呢？

var 的问题是变量会泄露到其它代码块，比如 for 循环或是 if 块。

test(false) 应该返回 outer，但并不是，得到的值是 undefined。

为什么？

因为即使 if 块没有执行，第 4 行的表达式 var x 也被提升了。

var 提升：

var 是在函数作用域中的，即使在声明前，它在整个函数内都是可用的。

声明被提升了。所以变量在声明前就可以使用。

初始化 没有 被提升。如果使用 var 的话，一定 要在顶部声明变量。

应用提升规则后，就好理解代码是如何执行的了：

例1：

let x = 'outer';

function test(inner) {

if (inner) {

let x = 'inner';

return x;

}

return x; // gets result from line 1 as expected

}

test(false); // outer

test(true); // inner

例2：

function f1() {

let n = 5;

if (true) {

let n = 10;

}

console.log(n); // 5

}

用 let 替代 var 以便按预期的那样去执行代码。如果没有调用 if 块，变量 x 就不会提升到块外。

Let 提升 和“temporal dead zone”

在 ES6 中，let 会将变量提升到块顶部（而 非 像 ES5 是函数顶部）。

但是在变量声明前引用变量会造成 ReferenceError。

let 是块作用域的，不可以在声明前使用。

“Temporal dead zone”是块开始直到变量被声明的这段区域。

## IIFE

在介绍 IIFE 之前，我们来看个例子：

ES5

{

var private = 1;

}

console.log(private); // 1

如你所见，private 会发生泄漏。需要使用 IIFE（立即执行函数表达式）将其包起来：

ES5

(function(){

var private2 = 1;

})();

console.log(private2); // Uncaught ReferenceError

如果看过 jQuery/lodash 或是其它开源项目的代码，你会注意到它们都利用了 IIFE，以避免污染全局环境，而只在全局下定义 \_、$或是 jQuery。

ES6 更工整，块级作用域的出现，实际上使得获得广泛应用的立即执行函数表达式（IIFE）不再必要了。不再需要使用 IIFE，只要用块和 let 就可以了：

ES6

{

let private3 = 1;

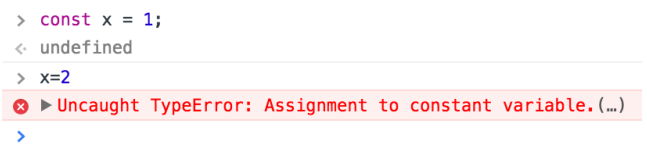
}

console.log(private3); // Uncaught ReferenceError

## Const

const声明一个只读的常量。一旦声明，常量的值就不能改变。

如果不希望变量的值再改变，可以使用 const。



总之：用 let 和 const 代替 var。

使用 const 进行引用；避免使用 var。

如果必须重新指定引用，可以用 let 代替 const。

## 文本模板

遇到文本模板时，不必再用嵌套连接了。比如：

ES5

var first = 'Adrian';

var last = 'Mejia';

console.log('Your name is ' + first + ' ' + last + '.');

现在可以用 反引号（`） 和字符串插值 ${}：

ES6

const first = 'Adrian';

const last = 'Mejia';

console.log(`Your name is ${first} ${last}.`);

反引号：键盘的左上方 1的左边的键，直接按下

## 多行字符串

不必像这样再连接 + n 字符串了：

ES5

var template = '<li \*ngFor="let todo of todos" [ngClass]="{completed: todo.isDone}">n' +

' <div class="view">n' +

' <input class="toggle" type="checkbox" [checked]="todo.isDone">n' +

' <label></label>n' +

' <button class="destroy"></button>n' +

' </div>n' +

' <input class="edit" value="">n' +

'</li>';

console.log(template);

ES6 中同样可以用反引号解决：

ES6

const template = `<li \*ngFor="let todo of todos" [ngClass]="{completed: todo.isDone}">

<div class="view">

<input class="toggle" type="checkbox" [checked]="todo.isDone">

<label></label>

<button class="destroy"></button>

</div>

<input class="edit" value="">

</li>`;

console.log(template);

两段代码会得到完全相同的结果。

## 解构赋值

ES6允许按照一定模式，从数组和对象中提取值，对变量进行赋值，这被称为解构。ES6 解构非常简明并且好用。看看下面的例子：

最简单的例子：

**多个变量赋值**：

ES5

var a = 1;

var b = 2;

var c = 3;

ES6：

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

**获取数组元素**

ES5

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

var first = array[0];

var third = array[2];

console.log(first, third); // 1 3

ES6

const array = [1, 2, 3, 4];

const [first, ,third] = array;

console.log(first, third); // 1 3

const array = [1, 2, 3, 4];

let [first, ,,third] = array;

console.log(first, third); // 1 4

**调换值**

ES5

var a = 1;

var b = 2;

var tmp = a;

a = b;

b = tmp;

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

等同于

ES6

let a = 1;

let b = 2;

[a, b] = [b, a];

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

**返回多个值的解构**

ES5

function margin() {

var left=1, right=2, top=3, bottom=4;

return { left: left, right: right, top: top, bottom: bottom };

}

var data = margin();

var left = data.left;

var bottom = data.bottom;

console.log(left, bottom); // 1 4

在第3行，也可以像这样用数组返回（并保存序列）：

return [left, right, top, bottom];

但之后调用时需要考虑返回数据的顺序。

var left = data[0];

var bottom = data[3];

ES6 中调用时只会选择需要的数据（第 6 行）：

ES6

function margin() {

const left=1, right=2, top=3, bottom=4;

return { left, right, top, bottom };

}

const { left, bottom } = margin();

console.log(left, bottom); // 1 4

注意：第3行用到了一些其它的 ES6 功能。可以将 { left: left } 简化为 { left }。看看和 ES5 的版本相比，现在多简洁啊~很酷不是吗？

注意：

1.解构要求等号两边的解构完全一致平衡，不然会发生解构失败，出现undefined

var [foo] = [];

var [bar, foo] = [1];

1. 如果等号的右边不是数组（或者严格地说，不是可遍历的结构），那么将会报错。

// 报错

let [foo] = 1;

let [foo] = false;

let [foo] = NaN;

## 参数匹配解构

ES5

var user = {firstName: 'Adrian', lastName: 'Mejia'};

function getFullName(user) {

var firstName = user.firstName;

var lastName = user.lastName;

return firstName + ' ' + lastName;

}

console.log(getFullName(user)); // Adrian Mejia

等同于（但更简洁）：

ES6

const user = {firstName: 'Adrian', lastName: 'Mejia'};

function getFullName({ firstName, lastName }) {

return `${firstName} ${lastName}`;

}

console.log(getFullName(user)); // Adrian Mejia

## 深度匹配

ES5

function settings() {

return { display: { color: 'red' }, keyboard: { layout: 'querty'} };

}

var tmp = settings();

var displayColor = tmp.display.color;

var keyboardLayout = tmp.keyboard.layout;

console.log(displayColor, keyboardLayout); // red querty

等同于（但更简洁）：

ES6

function settings() {

return { display: { color: 'red' }, keyboard: { layout: 'querty'} };

}

const { display: { color: displayColor }, keyboard: { layout: keyboardLayout }} = settings();

console.log(displayColor, keyboardLayout); // red querty

也叫对象解构。

如你所见，解构很有用，并有助于形成好的编码风格。

最佳实践：

使用数组解构获取元素或调换变量，这样就不用创建临时引用了。

对于多返回值的情况，不要用数组解构，用对象解构。

## 类和对象

ES6 用“类”替代“构造函数”。

在 JavaScript 中，每个对象都有原型对象。所有 JavaScript 对象都从原型上继承方法和属性。

ES5 以面向对象编程（OOP）的方式创建对象，是利用构造函数实现的：

ES5

function MyConstructor(name) {

this.name = name;

}

MyConstructor.prototype.speak = function speak() {

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

};

var animal = new MyConstructor('animal');

animal.speak(); // animal makes a noise.

ES6 提供了语法糖，可以用 class、constructor 等新的关键字、更少的样板代码实现相同的效果。同样可以看到相比于constructor.prototype.speak = function ()，用 speak() 定义方法更加清晰：

ES6

class Animal {

constructor(name) { //构造函数

this.name = name;

}

speak() {

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

}

}

const animal = new Animal('animal');

animal.speak(); // animal makes a noise.

可以看到两种方式（ES5/6）的结果和使用方式相同。

最佳实践：

最好使用 class 语法，避免直接操作 prototype。原因是这样代码更加简明易懂。

避免出现空的构造器。如果没有指明，类会有默认的构造器的。

## 继承

基于前面的 Animal 类，现在想要拓展 Animal，定义一个 Lion 类。

ES5 原型继承的方式有些复杂。

ES5

var Lion = (function () {

function MyConstructor(name){

Animal.call(this, name);

}

// prototypal inheritance

MyConstructor.prototype = Object.create(Animal.prototype);

MyConstructor.prototype.constructor = Animal;

MyConstructor.prototype.speak = function speak() {

Animal.prototype.speak.call(this);

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

};

return MyConstructor;

})();

var lion = new Lion('Simba');

lion.speak(); // Simba makes a noise.

// Simba roars.

在此我没有详细解读所有的代码，但是需要注意：

第 3 行，明确地用参数调用 Animal 构造器。

7-8 行，将 Lion 原型赋值为 Animal的原型。

11 行，从父类 Animal 调用了 speak 方法。

ES6 提供了新的关键字 extends 和 super。

ES6

class Lion extends Animal {

speak() {

super.speak();

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

}

}

const lion = new Lion('Simba');

lion.speak(); // Simba makes a noise.

// Simba roars.

效果相同，但是相比于 ES5，ES6 代码更易读，完胜。

最佳实践：

使用内置的 extends 实现继承。

## 原生 Promise

用 promise 替代回调地狱.

Promise有各种开源实现，在ES6中被统一规范，由浏览器直接支持。

Promise简例：

function test(resolve, reject) {

var timeOut = Math.random() \* 2;

log('set timeout to: ' + timeOut + ' seconds.');

setTimeout(function () {

if (timeOut < 1) {

Console.log('call resolve()...');

resolve('200 OK'); //对应then的参数

}

else {

console.log('call reject()...');

reject('timeout in ' + timeOut + ' seconds.'); //对应catch的参数

}

}, timeOut \* 1000);

}

var p1 = new Promise(test);

// 如果成功，执行这个函数：

var p2 = p1.then(function (result) {

console.log('成功：' + result);

});

//当test函数执行失败时，我们告诉Promise对象：

var p3 = p2.catch(function (reason) {

console.log('失败：' + reason);

});

Promise对象可以串联起来，所以上述代码可以简化为：

new Promise(test).then(function (result) {

console.log('成功：' + result);

}).catch(function (reason) {

console.log('失败：' + reason);

});

ES5

function printAfterTimeout(string, timeout, done){

setTimeout(function(){

done(string);

}, timeout);

}

printAfterTimeout('Hello ', 2e3, function(result){

console.log(result);

// nested callback

printAfterTimeout(result + 'Reader', 2e3, function(result){

console.log(result);

});

});

这个函数接收一个回调，在 done 后执行。我们想要先后执行两次，所以在回调中又一次调用了 printAfterTimeout。

如果需要 3 或 4 次回调，代码很快就一团糟了。那么用 promise 如何实现呢：

ES6

function printAfterTimeout(string, timeout){

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

setTimeout(function(){

resolve(string);

}, timeout);

});

}

printAfterTimeout('Hello ', 2e3).then((result) => {

console.log(result);

return printAfterTimeout(result + 'Reader', 2e3);

}).then((result) => {

console.log(result);

});

promise 中可以用 then 在某个函数完成后执行新的代码，而不必再嵌套函数。

## 箭头函数

ES6 没有移除函数表达式，但是新增了箭头函数。

ES5 中，this 的指向有问题：

ES5

var \_this = this; // need to hold a reference

$('.btn').click(function(event){

\_this.sendData(); // reference outer this

});

$('.input').on('change',function(event){

this.sendData(); // reference outer this

}.bind(this)); // bind to outer this

在函数内，需要用临时变量指向 this 或者使用 bind 绑定。ES6 中可以使用箭头函数。

我们先来看看箭头函数的语法：

([param] [, param]) => {

statements

}

param => expression

param 是参数，根据参数个数不同，分这几种情况：

() => { … } // 零个参数用 () 表示；

x => { … } // 一个参数可以省略 ()；

(x, y) => { … } // 多参数不能省略 ()；

多行语句需要用{}括起来，单行表达式不需要{}，并且会作为函数返回值。

例1：

// ES5

var total = values.reduce(function (a, b) {

return a + b;

}, 0);

// ES6

var total = values.reduce((a, b) => a + b, 0);

例2：

// ES5

$("#confetti-btn").click(function (event) {

playTrumpet();

fireConfettiCannon();

});

// ES6

$("#confetti-btn").click(event => {

playTrumpet();

fireConfettiCannon();

});

例3：

var func1 = (x = 1, y = 2) => x + y;

Var r = func1(); // 得到 3

Console.log(r);

var func2 = (x, ...args) => { console.log(args) };

func2(1,2,3); // 输出 [2, 3]

***箭头函数的 this 始终指向函数定义时的 this，而非执行时。***

原始：

var o = {

x : 1,

func : function() { console.log(this.x) },

test : function() {

var \_this = this;

setTimeout(function() { //this -- > window

\_this.func();

}, 100);

}

};

o.test();

使用ES6箭头函数：

var o = {

x : 1,

func : function() { console.log(this.x) },

test : function() {

setTimeout(() => { this.func() }, 100);

}

};

o.test();

***this是不会改变指向对象的，我们知道call和apply可以改变this的指向，但是在箭头函数中是无效的。***

var x = 1,

o = {

x : 10,

test : () => this.x

};

o.test(); // 1

o.test.call(o); // 依然是1

## For…of

最开始用 for ，然后使用 forEach，而现在可以用 for…of：

ES6

// this will reference the outer one

$('.btn').click((event) => this.sendData());

// implicit returns

const ids = [291, 288, 984];

const messages = ids.map(value => `ID is ${value}`);

ES6 的 for…of 也可以用来迭代。

## 默认参数

之前需要检测变量是否定义了，而现在可以指定 default parameters 的值。或许你之前像下面这样写过？

ES5

function point(x, y, isFlag){

x = x || 0;

y = y || -1;

isFlag = isFlag || true;

console.log(x,y, isFlag);

}

point(0, 0) // 0 -1 true

point(0, 0, false) // 0 -1 true

point(1) // 1 -1 true

point() // 0 -1 true

这可能是检测变量有值或指定默认值的惯用模式，但也存在一些问题：

第 8 行，我们传的值是 0, 0 但是得到的是 0, -1

第 9 行，传进去 false 但是得到的是 true。

如果默认参数是布尔值或将值设为 0，是没有用的。想知道为什么？我会在下面的 ES6 示例后说明。

有了 ES6，现在可以用更少的代码实现更好的效果了。

ES6

function point(x = 0, y = -1, isFlag = true){

console.log(x,y, isFlag);

}

point(0, 0) // 0 0 true

point(0, 0, false) // 0 0 false

point(1) // 1 -1 true

point() // 0 -1 true

注意第 5 行和第 6 行我们拿到了想要的值。ES5 的示例不好用，是因为先要检测 undefined的值，而 false、 null、 undefined 和 0 都是假的值。我们可以加些代码：

ES5

function point(x, y, isFlag){

x = x || 0;

y = typeof(y) === 'undefined' ? -1 : y;

isFlag = typeof(isFlag) === 'undefined' ? true : isFlag;

console.log(x,y, isFlag);

}

point(0, 0) // 0 0 true

point(0, 0, false) // 0 0 false

point(1) // 1 -1 true

point() // 0 -1 true

现在当检测 undefined 值时就符合我们的要求了。

## 剩余参数

之前使用 arguments，而现在可以用展开操作符。

ES5 中处理不定参数很麻烦：

ES5

function printf(format) {

var params = [].slice.call(arguments, 1); //把参数对象从1索引开始的部分切割出来

console.log('params: ', params);

console.log('format: ', format);

}

printf('%s %d %.2f', 'adrian', 321, Math.PI);

现在可以用展开操作符 ... 达到相同的目的。

ES6

function printf(format, ...params) { //把剩下部分组成一个数组

console.log('params: ', params);

console.log('format: ', format);

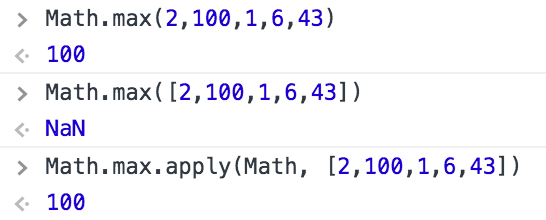
}

printf('%s %d %.2f', 'adrian', 321, Math.PI);

## 展开操作符

之前用 apply()，现在可以方便地使用展开操作符 ... 了：

提示：apply() 可以将数组转化为一系列参数。例如 Math.max() 接收一系列参数，但如果想应用于数组的话可以用 apply 帮助实现。



如上所述，apply 可以将数组当作参数序列进行传递：

ES5

Math.max.apply(Math, [2,100,1,6,43]) // 100

ES6 可以用展开操作符：

ES6

Math.max(...[2,100,1,6,43]) // 100

之前用 concat 合并数组，现在也可以用展开操作符：

ES5

var array1 = [2,100,1,6,43];

var array2 = ['a', 'b', 'c', 'd'];

var array3 = [false, true, null, undefined];

console.log(array1.concat(array2, array3));

ES6 可以用展开操作符展开嵌套的数组：

ES6

const array1 = [2,100,1,6,43];

const array2 = ['a', 'b', 'c', 'd'];

const array3 = [false, true, null, undefined];

console.log([...array1, ...array2, ...array3]);