# ECMAScript

# 一、简介

ECMAScript 6.0（以下简称 ES6）是 JavaScript 语言的下一代标准，已经在 2015 年 6 月正式发布了.

1. **ES6 与 ECMAScript 2015 的关系**

2011 年，ECMAScript 5.1 版发布后，就开始制定 6.0 版了。因此，ES6 这个词的原意，就是指 JavaScript 语言的下一个版本。

但是，因为这个版本引入的语法功能太多，而且制定过程当中，还有很多组织和个人不断提交新功能。事情很快就变得清楚了，不可能在一个版本里面包括所有将要引入的功能。常规的做法是先发布 6.0 版，过一段时间再发 6.1 版，然后是 6.2 版、6.3 版等等。

但是，标准的制定者不想这样做。他们想让标准的升级成为常规流程：任何人在任何时候，都可以向标准委员会提交新语法的提案，然后标准委员会每个月开一次会，评估这些提案是否可以接受，需要哪些改进。如果经过多次会议以后，一个提案足够成熟了，就可以正式进入标准了。这就是说，标准的版本升级成为了一个不断滚动的流程，每个月都会有变动。

标准委员会最终决定，标准在每年的 6 月份正式发布一次，作为当年的正式版本。接下来的时间，就在这个版本的基础上做改动，直到下一年的 6 月份，草案就自然变成了新一年的版本。这样一来，就不需要以前的版本号了，只要用年份标记就可以了。

ES6 的第一个版本，就这样在 2015 年 6 月发布了，正式名称就是《ECMAScript 2015 标准》（简称 ES2015）。2016 年 6 月，小幅修订的《ECMAScript 2016 标准》（简称 ES2016）如期发布，这个版本可以看作是 ES6.1 版，因为两者的差异非常小（只新增了数组实例的includes方法和指数运算符），基本上是同一个标准。根据计划，2017 年 6 月发布 ES2017 标准。

因此，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”。

# 二、let 和 const 命令

1. **let命令**

**基本用法**

ES6 新增了let命令，用来声明变量。它的用法类似于var，但是所声明的变量，只在let命令所在的代码块内有效。

{

let a = 10;

var b = 1;

}

a // ReferenceError: a is not defined.

b // 1

上面代码在代码块之中，分别用let和var声明了两个变量。然后在代码块之外调用这两个变量，结果let声明的变量报错，var声明的变量返回了正确的值。这表明，let声明的变量只在它所在的代码块有效。

for循环的计数器，就很合适使用let命令。

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

// ...

}

console.log(i);

// ReferenceError: i is not defined

上面代码中，计数器i只在for循环体内有效，在循环体外引用就会报错。

下面的代码如果使用var，最后输出的是10。

var a = [];

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

a[i] = function () {

console.log(i);

};

}

a[6](); // 10

上面代码中，变量i是var命令声明的，在全局范围内都有效，所以全局只有一个变量i。每一次循环，变量i的值都会发生改变，而循环内被赋给数组a的函数内部的console.log(i)，里面的i指向的就是全局的i。也就是说，所有数组a的成员里面的i，指向的都是同一个i，导致运行时输出的是最后一轮的i的值，也就是 10。

如果使用let，声明的变量仅在块级作用域内有效，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

上面代码中，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。

**暂时性死区**

只要块级作用域内存在let命令，它所声明的变量就“绑定”（binding）这个区域，不再受外部的影响

var tmp = 123;

if (true) {

tmp = 'abc'; // ReferenceError

let tmp;

}

上面代码中，存在全局变量tmp，但是块级作用域内let又声明了一个局部变量tmp，导致后者绑定这个块级作用域，所以在let声明变量前，对tmp赋值会报错。

总之，在代码块内，使用let命令声明变量之前，该变量都是不可用的。这在语法上，称为“暂时性死区”（temporal dead zone，简称 TDZ）。

if (true) {

// TDZ开始

tmp = 'abc'; // ReferenceError

console.log(tmp); // ReferenceError

let tmp; // TDZ结束

console.log(tmp); // undefined

tmp = 123;

console.log(tmp); // 123

}

上面代码中，在let命令声明变量tmp之前，都属于变量tmp的“死区”。

**不允许重复声明**

// 报错

function func() {

let a = 10;

var a = 1;

}

// 报错

function func() {

let a = 10;

let a = 1;

}

因此，不能在函数内部重新声明参数。

function func(arg) {

let arg;

}

func() // 报错

function func(arg) {

{

let arg;

}

}

func() // 不报错

1. **块级作用域**

**为什么需要块级作用域？**

ES5 只有全局作用域和函数作用域，没有块级作用域，这带来很多不合理的场景。

第一种场景，内层变量可能会覆盖外层变量。

var tmp = new Date();

function f() {

console.log(tmp);

if (false) {

var tmp = 'hello world';

}}

f(); // undefined

上面代码的原意是，if代码块的外部使用外层的tmp变量，内部使用内层的tmp变量。但是，函数f执行后，输出结果为undefined，原因在于变量提升，导致内层的tmp变量覆盖了外层的tmp变量。

第二种场景，用来计数的循环变量泄露为全局变量。

var s = 'hello';

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

console.log(s[i]);

}

console.log(i); // 5

上面代码中，变量i只用来控制循环，但是循环结束后，它并没有消失，泄露成了全局变量。

**ES6 的块级作用域**

let实际上为 JavaScript 新增了块级作用域。

function f1() {

let n = 5;

if (true) {

let n = 10;

}

console.log(n); // 5

}

上面的函数有两个代码块，都声明了变量n，运行后输出 5。这表示外层代码块不受内层代码块的影响。如果两次都使用var定义变量n，最后输出的值才是 10。

1. **const 命令**

**基本用法**

const声明一个只读的常量。一旦声明，常量的值就不能改变。

const PI = 3.1415;

PI // 3.1415

PI = 3;

// TypeError: Assignment to constant variable.

上面代码表明改变常量的值会报错。

const声明的变量不得改变值，这意味着，const一旦声明变量，就必须立即初始化，不能留到以后赋值。

const foo;

// SyntaxError: Missing initializer in const declaration

上面代码表示，对于const来说，只声明不赋值，就会报错。

const的作用域与let命令相同：只在声明所在的块级作用域内有效。

if (true) {

const MAX = 5;

}

MAX // Uncaught ReferenceError: MAX is not defined

const命令声明的常量也是不提升，同样存在暂时性死区，只能在声明的位置后面使用。

if (true) {

console.log(MAX); // ReferenceError

const MAX = 5;

}

上面代码在常量MAX声明之前就调用，结果报错。

const声明的常量，也与let一样不可重复声明。

var message = "Hello!";

let age = 25;

// 以下两行都会报错

const message = "Goodbye!";

const age = 30;

# 变量的解构赋值

1. **数组的解构赋值**

**基本用法**

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（Destructuring）。

以前，为变量赋值，只能直接指定值。

let a = 1;

let b = 2;

let c = 3;

ES6 允许写成下面这样。

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

上面代码表示，可以从数组中提取值，按照对应位置，对变量赋值。

本质上，这种写法属于“模式匹配”，只要等号两边的模式相同，左边的变量就会被赋予对应的值。下面是一些使用嵌套数组进行解构的例子。

let [foo, [[bar], baz]] = [1, [[2], 3]];

foo // 1

bar // 2

baz // 3

let [ , , third] = ["foo", "bar", "baz"];

third // "baz"

let [x, , y] = [1, 2, 3];

x // 1

y // 3

let [head, ...tail] = [1, 2, 3, 4];

head // 1

tail // [2, 3, 4]

let [x, y, ...z] = ['a'];

x // "a"

y // undefined

z // []

如果解构不成功，变量的值就等于undefined。

let [foo] = [];

let [bar, foo] = [1];

以上两种情况都属于解构不成功，foo的值都会等于undefined。

另一种情况是不完全解构，即等号左边的模式，只匹配一部分的等号右边的数组。这种情况下，解构依然可以成功。

let [x, y] = [1, 2, 3];

x // 1

y // 2

let [a, [b], d] = [1, [2, 3], 4];

a // 1

b // 2

d // 4

上面两个例子，都属于不完全解构，但是可以成功。

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;

let [foo] = undefined;

let [foo] = null;

let [foo] = {};

**默认值**

解构赋值允许指定默认值。

let [foo = true] = [];

foo // true

let [x, y = 'b'] = ['a']; // x='a', y='b'

let [x, y = 'b'] = ['a', undefined]; // x='a', y='b'

1. **对象的解构赋值**

**简介**

解构不仅可以用于数组，还可以用于对象。

let { foo, bar } = { foo: 'aaa', bar: 'bbb' };

foo // "aaa"

bar // "bbb"

对象的解构与数组有一个重要的不同。数组的元素是按次序排列的，变量的取值由它的位置决定；而对象的属性没有次序，变量必须与属性同名，才能取到正确的值。

let { bar, foo } = { foo: 'aaa', bar: 'bbb' };

foo // "aaa"

bar // "bbb"

let { baz } = { foo: 'aaa', bar: 'bbb' };

baz // undefined

上面代码的第一个例子，等号左边的两个变量的次序，与等号右边两个同名属性的次序不一致，但是对取值完全没有影响。第二个例子的变量没有对应的同名属性，导致取不到值，最后等于undefined。

如果解构失败，变量的值等于undefined。

let {foo} = {bar: 'baz'};

foo // undefined

上面代码中，等号右边的对象没有foo属性，所以变量foo取不到值，所以等于undefined。

对象的解构赋值，可以很方便地将现有对象的方法，赋值到某个变量。

// 例一

let { log, sin, cos } = Math;

// 例二

const { log } = console;

log('hello') // hello

上面代码的例一将Math对象的对数、正弦、余弦三个方法，赋值到对应的变量上，使用起来就会方便很多。例二将console.log赋值到log变量。

如果变量名与属性名不一致，必须写成下面这样。

let { foo: baz } = { foo: 'aaa', bar: 'bbb' };

baz // "aaa"

let obj = { first: 'hello', last: 'world' };

let { first: f, last: l } = obj;

f // 'hello'

l // 'world'

这实际上说明，对象的解构赋值是下面形式的简写。

let { foo: foo, bar: bar } = { foo: 'aaa', bar: 'bbb' };

也就是说，对象的解构赋值的内部机制，是先找到同名属性，然后再赋给对应的变量。真正被赋值的是后者，而不是前者。

let { foo: baz } = { foo: 'aaa', bar: 'bbb' };

baz // "aaa"

foo // error: foo is not defined

上面代码中，foo是匹配的模式，baz才是变量。真正被赋值的是变量baz，而不是模式foo。

**默认值**

对象的解构也可以指定默认值。

var {x = 3} = {};

x // 3

var {x, y = 5} = {x: 1};

x // 1

y // 5

var {x: y = 3} = {};

y // 3

var {x: y = 3} = {x: 5};

y // 5

var { message: msg = 'Something went wrong' } = {};

msg // "Something went wrong"

默认值生效的条件是，对象的属性值严格等于undefined。

var {x = 3} = {x: undefined};

x // 3

var {x = 3} = {x: null};

x // null

上面代码中，属性x等于null，因为null与undefined不严格相等，所以是个有效的赋值，导致默认值3不会生效。

1. **字符串的解构赋值**

字符串也可以解构赋值。这是因为此时，字符串被转换成了一个类似数组的对象。

const [a, b, c, d, e] = 'hello';

a // "h"

b // "e"

c // "l"

d // "l"

e // "o"

类似数组的对象都有一个length属性，因此还可以对这个属性解构赋值。

let {length : len} = 'hello';

len // 5

1. **函数参数的解构赋值**

函数的参数也可以使用解构赋值。

function add([x, y]){

return x + y;

}

add([1, 2]); // 3

上面代码中，函数add的参数表面上是一个数组，但在传入参数的那一刻，数组参数就被解构成变量x和y。对于函数内部的代码来说，它们能感受到的参数就是x和y。

下面是另一个例子。

[[1, 2], [3, 4]].map(([a, b]) => a + b);

// [ 3, 7 ]

函数参数的解构也可以使用默认值。

function move({x = 0, y = 0} = {}) {

return [x, y];

}

move({x: 3, y: 8}); // [3, 8]

move({x: 3}); // [3, 0]

move({}); // [0, 0]

move(); // [0, 0]

上面代码中，函数move的参数是一个对象，通过对这个对象进行解构，得到变量x和y的值。如果解构失败，x和y等于默认值。

1. **用途**

****（1）交换变量的值****

let x = 1;

let y = 2;

[x, y] = [y, x];

****（2）从函数返回多个值****

函数只能返回一个值，如果要返回多个值，只能将它们放在数组或对象里返回。有了解构赋值，取出这些值就非常方便。

// 返回一个数组

function example() {

return [1, 2, 3];

}

let [a, b, c] = example();

// 返回一个对象

function example() {

return {

foo: 1,

bar: 2

}; }

let { foo, bar } = example();

****（3）函数参数的定义****

解构赋值可以方便地将一组参数与变量名对应起来。

// 参数是一组有次序的值

function f([x, y, z]) { ... }

f([1, 2, 3]);

// 参数是一组无次序的值

function f({x, y, z}) { ... }

f({z: 3, y: 2, x: 1});

****（4）提取 JSON 数据****

解构赋值对提取 JSON 对象中的数据，尤其有用。

let jsonData = {

id: 42,

status: "OK",

data: [867, 5309]

};

let { id, status, data: number } = jsonData;

console.log(id, status, number);

// 42, "OK", [867, 5309]

上面代码可以快速提取 JSON 数据的值。

****（5）函数参数的默认值****

jQuery.ajax = function (url, {

async = true,

beforeSend = function () {},

cache = true,

complete = function () {},

crossDomain = false,

global = true,

// ... more config

} = {}) {

// ... do stuff

};

指定参数的默认值，就避免了在函数体内部再写var foo = config.foo || 'default foo';这样的语句。

****（6）输入模块的指定方法****

加载模块时，往往需要指定输入哪些方法。解构赋值使得输入语句非常清晰。

const { SourceMapConsumer, SourceNode } = require("source-map");

# 字符串的扩展

**模板字符串**

传统的 JavaScript 语言，输出模板通常是这样写的（下面使用了 jQuery 的方法）。

$('#result').append(

'There are <b>' + basket.count + '</b> ' +

'items in your basket, ' +

'<em>' + basket.onSale +

'</em> are on sale!'

);

上面这种写法相当繁琐不方便，ES6 引入了模板字符串解决这个问题。

$('#result').append(`

There are <b>${basket.count}</b> items

in your basket, <em>${basket.onSale}</em>

are on sale!

`);

模板字符串（template string）是增强版的字符串，用反引号（`）标识。它可以当作普通字符串使用，也可以用来定义多行字符串，或者在字符串中嵌入变量。

// 普通字符串

`In JavaScript '\n' is a line-feed.`

// 多行字符串

`In JavaScript this is

not legal.`

console.log(`string text line 1

string text line 2`);

// 字符串中嵌入变量

let name = "Bob", time = "today";

`Hello ${name}, how are you ${time}?`

上面代码中的模板字符串，都是用反引号表示。如果在模板字符串中需要使用反引号，则前面要用反斜杠转义。

let greeting = `\`Yo\` World!`;

如果使用模板字符串表示多行字符串，所有的空格和缩进都会被保留在输出之中。

$('#list').html(`<ul>

<li>first</li>

<li>second</li></ul>

`);

模板字符串中嵌入变量，需要将变量名写在${}之中。

function authorize(user, action) {

if (!user.hasPrivilege(action)) {

throw new Error(

// 传统写法为

// 'User '

// + user.name

// + ' is not authorized to do '

// + action

// + '.'

`User ${user.name} is not authorized to do ${action}.`);

}

}

大括号内部可以放入任意的 JavaScript 表达式，可以进行运算，以及引用对象属性。

let x = 1;

let y = 2;

`${x} + ${y} = ${x + y}`

// "1 + 2 = 3"

`${x} + ${y \* 2} = ${x + y \* 2}`

// "1 + 4 = 5"

let obj = {x: 1, y: 2};

`${obj.x + obj.y}`

// "3"

模板字符串之中还能调用函数。

function fn() {

return "Hello World";

}

`foo ${fn()} bar`

// foo Hello World bar

如果大括号中的值不是字符串，将按照一般的规则转为字符串。比如，大括号中是一个对象，将默认调用对象的toString方法。

如果模板字符串中的变量没有声明，将报错。

// 变量place没有声明

let msg = `Hello, ${place}`;

// 报错

由于模板字符串的大括号内部，就是执行 JavaScript 代码，因此如果大括号内部是一个字符串，将会原样输出。

`Hello ${'World'}`

// "Hello World"

# 函数的扩展

1. **函数参数的默认值**

**基本用法**

ES6 之前，不能直接为函数的参数指定默认值，只能采用变通的方法。

function log(x, y) {

y = y || 'World';

console.log(x, y);

}

log('Hello') // Hello World

log('Hello', 'China') // Hello China

log('Hello', '') // Hello World

上面代码检查函数log的参数y有没有赋值，如果没有，则指定默认值为World。这种写法的缺点在于，如果参数y赋值了，但是对应的布尔值为false，则该赋值不起作用。就像上面代码的最后一行，参数y等于空字符，结果被改为默认值。

为了避免这个问题，通常需要先判断一下参数y是否被赋值，如果没有，再等于默认值。

if (typeof y === 'undefined') {

y = 'World';

}

ES6 允许为函数的参数设置默认值，即直接写在参数定义的后面。

function log(x, y = 'World') {

console.log(x, y);

}

log('Hello') // Hello World

log('Hello', 'China') // Hello China

log('Hello', '') // Hello

可以看到，ES6 的写法比 ES5 简洁许多，而且非常自然。下面是另一个例子。

function Point(x = 0, y = 0) {

this.x = x;

this.y = y;

}

const p = new Point();

p // { x: 0, y: 0 }

除了简洁，ES6 的写法还有两个好处：首先，阅读代码的人，可以立刻意识到哪些参数是可以省略的，不用查看函数体或文档；其次，有利于将来的代码优化，即使未来的版本在对外接口中，彻底拿掉这个参数，也不会导致以前的代码无法运行。

**与解构赋值默认值结合使用**

参数默认值可以与解构赋值的默认值，结合起来使用。

function foo({x, y = 5}) {

console.log(x, y);

}

foo({}) // undefined 5

foo({x: 1}) // 1 5

foo({x: 1, y: 2}) // 1 2

foo() // TypeError: Cannot read property 'x' of undefined

上面代码只使用了对象的解构赋值默认值，没有使用函数参数的默认值。只有当函数foo的参数是一个对象时，变量x和y才会通过解构赋值生成。如果函数foo调用时没提供参数，变量x和y就不会生成，从而报错。通过提供函数参数的默认值，就可以避免这种情况。

function foo({x, y = 5} = {}) {

console.log(x, y);

}

foo() // undefined 5

上面代码指定，如果没有提供参数，函数foo的参数默认为一个空对象。

1. **rest 参数**

ES6 引入 rest 参数（形式为...变量名），用于获取函数的多余参数，这样就不需要使用arguments对象了。rest 参数搭配的变量是一个数组，该变量将多余的参数放入数组中。

function add(...values) {

let sum = 0;

for (var val of values) {

sum += val;

}

return sum;

}

add(2, 5, 3) // 10

上面代码的add函数是一个求和函数，利用 rest 参数，可以向该函数传入任意数目的参数。

下面是一个 rest 参数代替arguments变量的例子。

// arguments变量的写法

function sortNumbers() {

return Array.prototype.slice.call(arguments).sort();

}

// rest参数的写法

const sortNumbers = (...numbers) => numbers.sort();

上面代码的两种写法，比较后可以发现，rest 参数的写法更自然也更简洁。

arguments对象不是数组，而是一个类似数组的对象。所以为了使用数组的方法，必须使用Array.prototype.slice.call先将其转为数组。rest 参数就不存在这个问题，它就是一个真正的数组，数组特有的方法都可以使用。下面是一个利用 rest 参数改写数组push方法的例子。

function push(array, ...items) {

items.forEach(function(item) {

array.push(item);

console.log(item);

});

}

var a = [];

push(a, 1, 2, 3)

注意，rest 参数之后不能再有其他参数（即只能是最后一个参数），否则会报错。

// 报错

function f(a, ...b, c) {

// ...

}

1. **箭头函数**

**基本用法**

ES6 允许使用“箭头”（=>）定义函数。

var f = v => v;

// 等同于

var f = function (v) {

return v;

};

如果箭头函数不需要参数或需要多个参数，就使用一个圆括号代表参数部分。

var f = () => 5;

// 等同于

var f = function () { return 5 };

var sum = (num1, num2) => num1 + num2;

// 等同于

var sum = function(num1, num2) {

return num1 + num2;

};

如果箭头函数的代码块部分多于一条语句，就要使用大括号将它们括起来，并且使用return语句返回。

var sum = (num1, num2) => { return num1 + num2; }

由于大括号被解释为代码块，所以如果箭头函数直接返回一个对象，必须在对象外面加上括号，否则会报错。

// 报错

let getTempItem = id => { id: id, name: "Temp" };

// 不报错

let getTempItem = id => ({ id: id, name: "Temp" });

箭头函数可以与变量解构结合使用。

const full = ({ first, last }) => first + ' ' + last;

// 等同于

function full(person) {

return person.first + ' ' + person.last;

}

# 数组的扩展

1. **扩展运算符**

**含义**

扩展运算符（spread）是三个点（...）。它好比 rest 参数的逆运算，将一个数组转为用逗号分隔的参数序列。

console.log(...[1, 2, 3])

// 1 2 3

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

// 1 2 3 4 5

[...document.querySelectorAll('div')]

// [<div>, <div>, <div>]

该运算符主要用于函数调用。

function push(array, ...items) {

array.push(...items);

}

function add(x, y) {

return x + y;

}

const numbers = [4, 38];

add(...numbers) // 42

上面代码中，array.push(...items)和add(...numbers)这两行，都是函数的调用，它们都使用了扩展运算符。该运算符将一个数组，变为参数序列。

扩展运算符与正常的函数参数可以结合使用，非常灵活。

function f(v, w, x, y, z) { }

const args = [0, 1];

f(-1, ...args, 2, ...[3]);

如果扩展运算符后面是一个空数组，则不产生任何效果。

[...[], 1] // [1]

**扩展运算符的应用**

****（1）复制数组****

ES5 只能用变通方法来复制数组。

const a1 = [1, 2];

const a2 = a1.concat();

a2[0] = 2;

a1 // [1, 2]

上面代码中，a1会返回原数组的克隆，再修改a2就不会对a1产生影响。

扩展运算符提供了复制数组的简便写法。

const a1 = [1, 2];

// 写法一

const a2 = [...a1];

// 写法二

const [...a2] = a1;

上面的两种写法，a2都是a1的克隆。

****（2）合并数组****

扩展运算符提供了数组合并的新写法。

const arr1 = ['a', 'b'];

const arr2 = ['c'];

const arr3 = ['d', 'e'];

// ES5 的合并数组

arr1.concat(arr2, arr3);

// [ 'a', 'b', 'c', 'd', 'e' ]

// ES6 的合并数组

[...arr1, ...arr2, ...arr3]

// [ 'a', 'b', 'c', 'd', 'e' ]

****（4）字符串****

扩展运算符还可以将字符串转为真正的数组。

[...'hello']

// [ "h", "e", "l", "l", "o" ]

1. **数组实例的 entries()，keys() 和 values()**

ES6 提供三个新的方法——entries()，keys()和values()——用于遍历数组。它们都返回一个遍历器对象，可以用for...of循环进行遍历，唯一的区别是keys()是对键名的遍历、values()是对键值的遍历，entries()是对键值对的遍历。

for (let index of ['a', 'b'].keys()) {

console.log(index);

}

// 0

// 1

for (let elem of ['a', 'b'].values()) {

console.log(elem);

}

// 'a'

// 'b'

for (let [index, elem] of ['a', 'b'].entries()) {

console.log(index, elem);

}

// 0 "a"

// 1 "b"

# 对象的扩展

1. **属性的简洁表示法**

ES6 允许在大括号里面，直接写入变量和函数，作为对象的属性和方法。这样的书写更加简洁。

const foo = 'bar';

const baz = {foo};

baz // {foo: "bar"}

// 等同于

const baz = {foo: foo};

上面代码中，变量foo直接写在大括号里面。这时，属性名就是变量名, 属性值就是变量值。下面是另一个例子。

function f(x, y) {

return {x, y};

}

// 等同于

function f(x, y) {

return {x: x, y: y};

}

f(1, 2) // Object {x: 1, y: 2}

除了属性简写，方法也可以简写。

const o = {

method() {

return "Hello!";

}};

// 等同于

const o = {

method: function() {

return "Hello!";

}

};

下面是一个实际的例子。

let birth = '2000/01/01';

const Person = {

name: '张三',

//等同于birth: birth

birth,

// 等同于hello: function ()...

hello() {

console.log('我的名字是', this.name);

}

};

这种写法用于函数的返回值，将会非常方便。

function getPoint() {

const x = 1;

const y = 10;

return {x, y};

}

getPoint()

// {x:1, y:10}

CommonJS 模块输出一组变量，就非常合适使用简洁写法。

let ms = {};

function getItem (key) {

return key in ms ? ms[key] : null;

}

function setItem (key, value) {

ms[key] = value;

}

function clear () {

ms = {};

}

module.exports = { getItem, setItem, clear };

// 等同于

module.exports = {

getItem: getItem,

setItem: setItem,

clear: clear

};

1. **对象的扩展运算符**

**解构赋值**

对象的解构赋值用于从一个对象取值，相当于将目标对象自身的所有可遍历的（enumerable）、但尚未被读取的属性，分配到指定的对象上面。所有的键和它们的值，都会拷贝到新对象上面。

let { x, y, ...z } = { x: 1, y: 2, a: 3, b: 4 };

x // 1

y // 2

z // { a: 3, b: 4 }

上面代码中，变量z是解构赋值所在的对象。它获取等号右边的所有尚未读取的键（a和b），将它们连同值一起拷贝过来。

由于解构赋值要求等号右边是一个对象，所以如果等号右边是undefined或null，就会报错，因为它们无法转为对象。

let { ...z } = null; // 运行时错误

let { ...z } = undefined; // 运行时错误

解构赋值必须是最后一个参数，否则会报错。

let { ...x, y, z } = someObject; // 句法错误

let { x, ...y, ...z } = someObject; // 句法错误

上面代码中，解构赋值不是最后一个参数，所以会报错。

**扩展运算符**

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

let z = { a: 3, b: 4 };

let n = { ...z };

n // { a: 3, b: 4 }

由于数组是特殊的对象，所以对象的扩展运算符也可以用于数组。

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

foo

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

如果扩展运算符后面是一个空对象，则没有任何效果。

{...{}, a: 1}

// { a: 1 }

如果扩展运算符后面不是对象，则会自动将其转为对象。

// 等同于 {...Object(1)}

{...1} // {}

上面代码中，扩展运算符后面是整数1，会自动转为数值的包装对象Number{1}。由于该对象没有自身属性，所以返回一个空对象。

1. **Object.assign()**

**基本用法**

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

const target = { a: 1 };

const source1 = { b: 2 };

const source2 = { c: 3 };

Object.assign(target, source1, source2);

target // {a:1, b:2, c:3}

Object.assign方法的第一个参数是目标对象，后面的参数都是源对象。

注意，如果目标对象与源对象有同名属性，或多个源对象有同名属性，则后面的属性会覆盖前面的属性。

const target = { a: 1, b: 1 };

const source1 = { b: 2, c: 2 };

const source2 = { c: 3 };

Object.assign(target, source1, source2);

target // {a:1, b:2, c:3}

如果只有一个参数，Object.assign会直接返回该参数。

const obj = {a: 1};

Object.assign(obj) === obj // true

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

Object.assign(undefined) // 报错

Object.assign(null) // 报错

# Promise 对象

1. **基本用法**

ES6 规定，Promise对象是一个构造函数，用来生成Promise实例。

下面代码创造了一个Promise实例。

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

// ... some code

if (/\* 异步操作成功 \*/){

resolve(value);

} else {

reject(error);

}

});

Promise构造函数接受一个函数作为参数，该函数的两个参数分别是resolve和reject。它们是两个函数，由 JavaScript 引擎提供，不用自己部署。

resolve函数的作用是，将Promise对象的状态从“未完成”变为“成功”（即从 pending 变为 resolved），在异步操作成功时调用，并将异步操作的结果，作为参数传递出去；reject函数的作用是，将Promise对象的状态从“未完成”变为“失败”（即从 pending 变为 rejected），在异步操作失败时调用，并将异步操作报出的错误，作为参数传递出去。

Promise实例生成以后，可以用then方法分别指定resolved状态和rejected状态的回调函数。

promise.then(function(value) {

// success

}, function(error) {

// failure

});

then方法可以接受两个回调函数作为参数。第一个回调函数是Promise对象的状态变为resolved时调用，第二个回调函数是Promise对象的状态变为rejected时调用。其中，第二个函数是可选的，不一定要提供。这两个函数都接受Promise对象传出的值作为参数。

下面是一个Promise对象的简单例子。

function timeout(ms) {

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

setTimeout(resolve, ms, 'done');

});

}

timeout(100).then((value) => {

console.log(value);

});

上面代码中，timeout方法返回一个Promise实例，表示一段时间以后才会发生的结果。过了指定的时间（ms参数）以后，Promise实例的状态变为resolved，就会触发then方法绑定的回调函数。

Promise 新建后就会立即执行。

let promise = new Promise(function(resolve, reject) {

console.log('Promise');

resolve();

});

promise.then(function() {

console.log('resolved.');

});

console.log('Hi!');

// Promise// Hi!// resolved

上面代码中，Promise 新建后立即执行，所以首先输出的是Promise。然后，then方法指定的回调函数，将在当前脚本所有同步任务执行完才会执行，所以resolved最后输出。

1. **Promise.prototype.then()**

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

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

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

return json.post;}).then(function(post) {

// ...

});

上面的代码使用then方法，依次指定了两个回调函数。第一个回调函数完成以后，会将返回结果作为参数，传入第二个回调函数。

采用链式的then，可以指定一组按照次序调用的回调函数。这时，前一个回调函数，有可能返回的还是一个Promise对象（即有异步操作），这时后一个回调函数，就会等待该Promise对象的状态发生变化，才会被调用。

getJSON("/post/1.json").then(function(post) {

return getJSON(post.commentURL);}).then(function (comments) {

console.log("resolved: ", comments);}, function (err){

console.log("rejected: ", err);

});

上面代码中，第一个then方法指定的回调函数，返回的是另一个Promise对象。这时，第二个then方法指定的回调函数，就会等待这个新的Promise对象状态发生变化。如果变为resolved，就调用第一个回调函数，如果状态变为rejected，就调用第二个回调函数。

如果采用箭头函数，上面的代码可以写得更简洁。

getJSON("/post/1.json").then(

post => getJSON(post.commentURL)

).then(

comments => console.log("resolved: ", comments),

err => console.log("rejected: ", err)

);

1. **Promise.prototype.catch()**

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

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

// ...

}).catch(function(error) {

// 处理 getJSON 和 前一个回调函数运行时发生的错误

console.log('发生错误！', error);

});

上面代码中，getJSON方法返回一个 Promise 对象，如果该对象状态变为resolved，则会调用then方法指定的回调函数；如果异步操作抛出错误，状态就会变为rejected，就会调用catch方法指定的回调函数，处理这个错误。另外，then方法指定的回调函数，如果运行中抛出错误，也会被catch方法捕获。

下面是一个例子。

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

throw new Error('test');

});

promise.catch(function(error) {

console.log(error);

});

// Error: test

上面代码中，promise抛出一个错误，就被catch方法指定的回调函数捕获。注意，上面的写法与下面两种写法是等价的。

// 写法一

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

try {

throw new Error('test');

} catch(e) {

reject(e);

}});

promise.catch(function(error) {

console.log(error);

});

// 写法二

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

reject(new Error('test'));

});

promise.catch(function(error) {

console.log(error);

});

比较上面两种写法，可以发现reject方法的作用，等同于抛出错误。

1. **Promise.prototype.finally()**

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

promise

.then(result => {···})

.catch(error => {···})

.finally(() => {···});

上面代码中，不管promise最后的状态，在执行完then或catch指定的回调函数以后，都会执行finally方法指定的回调函数。

下面是一个例子，服务器使用 Promise 处理请求，然后使用finally方法关掉服务器。

server.listen(port)

.then(function () {

// ...

})

.finally(server.stop);

finally方法的回调函数不接受任何参数，这意味着没有办法知道，前面的 Promise 状态到底是fulfilled还是rejected。这表明，finally方法里面的操作，应该是与状态无关的，不依赖于 Promise 的执行结果。

1. **Promise.prototype.all()**

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

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

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

p的状态由p1、p2、p3决定，分成两种情况。

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

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

下面是一个具体的例子。

// 生成一个Promise对象的数组

const promises = [2, 3, 5, 7, 11, 13].map(function (id) {

return getJSON('/post/' + id + ".json");

});

Promise.all(promises)

.then(function (posts) {

// ...

}).catch(function(reason){

// ...

});

上面代码中，promises是包含 6 个 Promise 实例的数组，只有这 6 个实例的状态都变成fulfilled，或者其中有一个变为rejected，才会调用Promise.all方法后面的回调函数。

1. **Promise.resolve()**

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

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

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

Promise.resolve()等价于下面的写法。

Promise.resolve('foo')

// 等价于

new Promise(resolve => resolve('foo'))

Promise.resolve方法的参数分成四种情况。

****（1）参数是一个 Promise 实例****

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

****（2）参数是一个****thenable****对象****

thenable对象指的是具有then方法的对象，比如下面这个对象。

let thenable = {

then: function(resolve, reject) {

resolve(42);

}};

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

let thenable = {

then: function(resolve, reject) {

resolve(42);

}};

let p1 = Promise.resolve(thenable);

p1.then(function(value) {

console.log(value); // 42

});

上面代码中，thenable对象的then方法执行后，对象p1的状态就变为resolved，从而立即执行最后那个then方法指定的回调函数，输出 42。

****（3）参数不是具有****then****方法的对象，或根本就不是对象****

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

const p = Promise.resolve('Hello');

p.then(function (s){

console.log(s)

});

// Hello

上面代码生成一个新的 Promise 对象的实例p。由于字符串Hello不属于异步操作（判断方法是字符串对象不具有 then 方法），返回 Promise 实例的状态从一生成就是resolved，所以回调函数会立即执行。Promise.resolve方法的参数，会同时传给回调函数。

****（4）不带有任何参数****

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

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

const p = Promise.resolve();

p.then(function () {

// ...

});

上面代码的变量p就是一个 Promise 对象。

1. **Promise.reject()**

Promise.reject(reason)方法也会返回一个新的 Promise 实例，该实例的状态为rejected。

const p = Promise.reject('出错了');

// 等同于

const p = new Promise((resolve, reject) => reject('出错了'))

上面代码生成一个 Promise 对象的实例p，状态为rejected，回调函数会立即执行。

注意，Promise.reject()方法的参数，会原封不动地作为reject的理由，变成后续方法的参数。这一点与Promise.resolve方法不一致。

const thenable = {

then(resolve, reject) {

reject('出错了');

}};

Promise.reject(thenable).catch(e => {

console.log(e === thenable)

})

// true

上面代码中，Promise.reject方法的参数是一个thenable对象，执行以后，后面catch方法的参数不是reject抛出的“出错了”这个字符串，而是thenable对象。

# async 函数

1. **基本用法**

async函数返回一个 Promise 对象，可以使用then方法添加回调函数。当函数执行的时候，一旦遇到await就会先返回，等到异步操作完成，再接着执行函数体内后面的语句。 下面是一个例子。

async function getStockPriceByName(name) {

const symbol = await getStockSymbol(name);

const stockPrice = await getStockPrice(symbol);

return stockPrice;

}

getStockPriceByName('goog').then(function (result) {

console.log(result);

});

上面代码是一个获取股票报价的函数，函数前面的async关键字，表明该函数内部有异步操作。调用该函数时，会立即返回一个Promise对象。

下面是另一个例子，指定多少毫秒后输出一个值。

function timeout(ms) {

return new Promise((resolve) => {

setTimeout(resolve, ms);

});

}

async function asyncPrint(value, ms) {

await timeout(ms);

console.log(value);

}

asyncPrint('hello world', 50);

上面代码指定 50 毫秒以后，输出hello world。

由于async函数返回的是 Promise 对象，可以作为await命令的参数。所以，上面的例子也可以写成下面的形式。

async function timeout(ms) {

await new Promise((resolve) => {

setTimeout(resolve, ms);

});

}

async function asyncPrint(value, ms) {

await timeout(ms);

console.log(value);

}

asyncPrint('hello world', 50);

async 函数有多种使用形式。

// 函数声明

async function foo() {}

// 函数表达式

const foo = async function () {};

// 对象的方法

let obj = { async foo() {} };

obj.foo().then(...)

// 箭头函数

const foo = async () => {};

1. **语法**

**返回 Promise 对象**

async函数返回一个 Promise 对象。

async函数内部return语句返回的值，会成为then方法回调函数的参数。

async function f() {

return 'hello world';

}

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

// "hello world"

上面代码中，函数f内部return命令返回的值，会被then方法回调函数接收到。

async函数内部抛出错误，会导致返回的 Promise 对象变为reject状态。抛出的错误对象会被catch方法回调函数接收到。

async function f() {

throw new Error('出错了');

}

f().then(

v => console.log(v),

e => console.log(e))

// Error: 出错了

**Promise 对象的状态变化**

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

下面是一个例子。

async function getTitle(url) {

let response = await fetch(url);

let html = await response.text();

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

}

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

// "ECMAScript 2017 Language Specification"

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

**await 命令**

await命令后面的 Promise 对象如果变为reject状态，则reject的参数会被catch方法的回调函数接收到。

async function f() {

await Promise.reject('出错了');

}

f()

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

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

// 出错了

注意，上面代码中，await语句前面没有return，但是reject方法的参数依然传入了catch方法的回调函数。这里如果在await前面加上return，效果是一样的。

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

async function f() {

await Promise.reject('出错了');

await Promise.resolve('hello world'); // 不会执行

}

上面代码中，第二个await语句是不会执行的，因为第一个await语句状态变成了reject。

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

async function f() {

try {

await Promise.reject('出错了');

} catch(e) {}

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

}

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

// hello world

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

async function f() {

await Promise.reject('出错了')

.catch(e => console.log(e));

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

}

f()

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

// 出错了

// hello world

**http://kangax.github.io/compat-table/es6/**