## Babel 转码器

[Babel](https://babeljs.io/) 是一个广泛使用的 ES6 转码器，可以将 ES6 代码转为 ES5 代码，从而在老版本的浏览器执行。这意味着，你可以用 ES6 的方式编写程序，又不用担心现有环境是否支持。下面是一个例子。

// 转码前input.map(item => item + 1);

// 转码后input.map(function (item) {

return item + 1;});

### 配置文件.babelrc

Babel 的配置文件是.babelrc，存放在项目的根目录下。使用 Babel 的第一步，就是配置这个文件。

该文件用来设置转码规则和插件，基本格式如下。

{

"presets": [],

"plugins": []}

presets字段设定转码规则，官方提供以下的规则集，你可以根据需要安装。

# 最新转码规则$ npm install --save-dev @babel/preset-env

# react 转码规则$ npm install --save-dev @babel/preset-react

然后，将这些规则加入.babelrc。

{

"presets": [

"@babel/env",

"@babel/preset-react"

],

"plugins": []

}

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。

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。你可能会问，如果每一轮循环的变量i都是重新声明的，那它怎么知道上一轮循环的值，从而计算出本轮循环的值？这是因为 JavaScript 引擎内部会记住上一轮循环的值，初始化本轮的变量i时，就在上一轮循环的基础上进行计算。

for循环还有一个特别之处，就是设置循环变量的那部分是一个父作用域，而循环体内部是一个单独的子作用域。

for (let i = 0; i < 3; i++) {

let i = 'abc';

console.log(i);}

// abc// abc// abc

上面代码正确运行，输出了 3 次abc。这表明函数内部的变量i与循环变量i不在同一个作用域，有各自单独的作用域（同一个作用域不可使用 let 重复声明同一个变量）。

### 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。这种现象多多少少是有些奇怪的，按照一般的逻辑，变量应该在声明语句之后才可以使用。

为了纠正这种现象，let命令改变了语法行为，它所声明的变量一定要在声明后使用，否则报错。

// var 的情况console.log(foo); // 输出undefinedvar foo = 2;

// let 的情况console.log(bar); // 报错ReferenceErrorlet bar = 2;

上面代码中，变量foo用var命令声明，会发生变量提升，即脚本开始运行时，变量foo已经存在了，但是没有值，所以会输出undefined。变量bar用let命令声明，不会发生变量提升。这表示在声明它之前，变量bar是不存在的，这时如果用到它，就会抛出一个错误。

“暂时性死区”也意味着typeof不再是一个百分之百安全的操作。

typeof x; // ReferenceError

let x;

没有let ，typeof返回undefined

### 不允许重复声明

let不允许在相同作用域内，重复声明同一个变量。

### 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。

ES6 允许块级作用域的任意嵌套。

{{{{

{let insane = 'Hello World'}

console.log(insane); // 报错}}}};

上面代码使用了一个五层的块级作用域，每一层都是一个单独的作用域。第四层作用域无法读取第五层作用域的内部变量。

内层作用域可以定义外层作用域的同名变量。

{{{{

let insane = 'Hello World';

{let insane = 'Hello World'}}}}};

### 块级作用域与函数声明

函数能不能在块级作用域之中声明？这是一个相当令人混淆的问题。

ES5 规定，函数只能在顶层作用域和函数作用域之中声明，不能在块级作用域声明。

// 情况一if (true) {

function f() {}}

// 情况二try {

function f() {}} catch(e) {

// ...}

上面两种函数声明，根据 ES5 的规定都是非法的。

但是，浏览器没有遵守这个规定，为了兼容以前的旧代码，还是支持在块级作用域之中声明函数，因此上面两种情况实际都能运行，不会报错。

ES6 引入了块级作用域，明确允许在块级作用域之中声明函数。ES6 规定，块级作用域之中，函数声明语句的行为类似于let，在块级作用域之外不可引用。

function f() { console.log('I am outside!'); }

(function () {

if (false) {

// 重复声明一次函数f function f() { console.log('I am inside!'); }

}

f();}());

上面代码在 ES5 中运行，会得到“I am inside!”，因为在if内声明的函数f会被提升到函数头部，实际运行的代码如下。

// ES5 环境function f() { console.log('I am outside!'); }

(function () {

function f() { console.log('I am inside!'); }

if (false) {

}

f();}());

ES6 就完全不一样了，理论上会得到“I am outside!”。因为块级作用域内声明的函数类似于let，对作用域之外没有影响。但是，如果你真的在 ES6 浏览器中运行一下上面的代码，是会报错的，这是为什么呢？

// 浏览器的 ES6 环境function f() { console.log('I am outside!'); }

(function () {

if (false) {

// 重复声明一次函数f function f() { console.log('I am inside!'); }

}

f();}());

// Uncaught TypeError: f is not a function

上面的代码在 ES6 浏览器中，都会报错。

原来，如果改变了块级作用域内声明的函数的处理规则，显然会对老代码产生很大影响。为了减轻因此产生的不兼容问题，ES6 在[附录 B](https://www.ecma-international.org/ecma-262/6.0/index.html" \l "sec-block-level-function-declarations-web-legacy-compatibility-semantics)里面规定，浏览器的实现可以不遵守上面的规定，有自己的[行为方式](https://stackoverflow.com/questions/31419897/what-are-the-precise-semantics-of-block-level-functions-in-es6)。

* 允许在块级作用域内声明函数。
* 函数声明类似于var，即会提升到全局作用域或函数作用域的头部。
* 同时，函数声明还会提升到所在的块级作用域的头部。

注意，上面三条规则只对 ES6 的浏览器实现有效，其他环境的实现不用遵守，还是将块级作用域的函数声明当作let处理。

根据这三条规则，浏览器的 ES6 环境中，块级作用域内声明的函数，行为类似于var声明的变量。上面的例子实际运行的代码如下。

// 浏览器的 ES6 环境function f() { console.log('I am outside!'); }(function () {

var f = undefined;

if (false) {

function f() { console.log('I am inside!'); }

}

f();}());

// Uncaught TypeError: f is not a function

考虑到环境导致的行为差异太大，应该避免在块级作用域内声明函数。如果确实需要，也应该写成函数表达式，而不是函数声明语句。

// 块级作用域内部的函数声明语句，建议不要使用{

let a = 'secret';

function f() {

return a;

}}

// 块级作用域内部，优先使用函数表达式{

let a = 'secret';

let f = function () {

return a;

};}

const声明一个只读的常量。一旦声明，常量的值就不能改变。

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。

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。

const实际上保证的，并不是变量的值不得改动，而是变量指向的那个内存地址所保存的数据不得改动。对于简单类型的数据（数值、字符串、布尔值），值就保存在变量指向的那个内存地址，因此等同于常量。但对于复合类型的数据（主要是对象和数组），变量指向的内存地址，保存的只是一个指向实际数据的指针，const只能保证这个指针是固定的（即总是指向另一个固定的地址），至于它指向的数据结构是不是可变的，就完全不能控制了。因此，将一个对象声明为常量必须非常小心。

const foo = {};

// 为 foo 添加一个属性，可以成功foo.prop = 123;

foo.prop // 123

// 将 foo 指向另一个对象，就会报错foo = {}; // TypeError: "foo" is read-only

如果真的想将对象冻结，应该使用Object.freeze方法。

const foo = Object.freeze({});

// 常规模式时，下面一行不起作用；// 严格模式时，该行会报错foo.prop = 123;

除了将对象本身冻结，对象的属性也应该冻结。下面是一个将对象彻底冻结的函数。

var constantize = (obj) => {

Object.freeze(obj);

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

if ( typeof obj[key] === 'object' ) {

constantize( obj[key] );

}

});};

顶层对象，在浏览器环境指的是window对象，在 Node 指的是global对象。ES5 之中，顶层对象的属性与全局变量是等价的。

window.a = 1;

a // 1

a = 2;

window.a // 2

顶层对象，在浏览器环境指的是window对象，在 Node 指的是global对象。ES5 之中，顶层对象的属性与全局变量是等价的。

window.a = 1;

a // 1

a = 2;

window.a // 2

上面代码中，顶层对象的属性赋值与全局变量的赋值，是同一件事。

顶层对象的属性与全局变量挂钩，被认为是 JavaScript 语言最大的设计败笔之一。这样的设计带来了几个很大的问题，首先是没法在编译时就报出变量未声明的错误，只有运行时才能知道（因为全局变量可能是顶层对象的属性创造的，而属性的创造是动态的）；其次，程序员很容易不知不觉地就创建了全局变量（比如打字出错）；最后，顶层对象的属性是到处可以读写的，这非常不利于模块化编程。另一方面，window对象有实体含义，指的是浏览器的窗口对象，顶层对象是一个有实体含义的对象，也是不合适的。

ES6 为了改变这一点，一方面规定，为了保持兼容性，var命令和function命令声明的全局变量，依旧是顶层对象的属性；另一方面规定，let命令、const命令、class命令声明的全局变量，不属于顶层对象的属性。也就是说，从 ES6 开始，全局变量将逐步与顶层对象的属性脱钩。

var a = 1;

// 如果在 Node 的 REPL 环境，可以写成 global.a// 或者采用通用方法，写成 this.awindow.a // 1let b = 1;

window.b // undefined

上面代码中，全局变量a由var命令声明，所以它是顶层对象的属性；全局变量b由let命令声明，所以它不是顶层对象的属性，返回undefined。

JavaScript 语言存在一个顶层对象，它提供全局环境（即全局作用域），所有代码都是在这个环境中运行。但是，顶层对象在各种实现里面是不统一的。

* 浏览器里面，顶层对象是window，但 Node 和 Web Worker 没有window。
* 浏览器和 Web Worker 里面，self也指向顶层对象，但是 Node 没有self。
* Node 里面，顶层对象是global，但其他环境都不支持。

同一段代码为了能够在各种环境，都能取到顶层对象，现在一般是使用this关键字，但是有局限性。

* 全局环境中，this会返回顶层对象。但是，Node.js 模块中this返回的是当前模块，ES6 模块中this返回的是undefined。
* 函数里面的this，如果函数不是作为对象的方法运行，而是单纯作为函数运行，this会指向顶层对象。但是，严格模式下，这时this会返回undefined。
* 不管是严格模式，还是普通模式，new Function('return this')()，总是会返回全局对象。但是，如果浏览器用了 CSP（Content Security Policy，内容安全策略），那么eval、new Function这些方法都可能无法使用。

综上所述，很难找到一种方法，可以在所有情况下，都取到顶层对象。下面是两种勉强可以使用的方法。

// 方法一(typeof window !== 'undefined'

? window

: (typeof process === 'object' &&

typeof require === 'function' &&

typeof global === 'object')

? global

: this);

// 方法二var getGlobal = function () {

if (typeof self !== 'undefined') { return self; }

if (typeof window !== 'undefined') { return window; }

if (typeof global !== 'undefined') { return global; }

throw new Error('unable to locate global object');};

[ES2020](https://github.com/tc39/proposal-global) 在语言标准的层面，引入globalThis作为顶层对象。也就是说，任何环境下，globalThis都是存在的，都可以从它拿到顶层对象，指向全局环境下的this。

垫片库[global-this](https://github.com/ungap/global-this)模拟了这个提案，可以在所有环境拿到globalTh

S6 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）。

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

third // "baz"let [x, , y] = [1, 2, 3];

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

head // 1tail // [2, 3, 4]let [x, y, ...z] = ['a'];

x // "a"y // undefinedz // []

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。

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

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

a // 1b // 2d // 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，参见《Iterator》一章），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] = {};

上面的语句都会报错，因为等号右边的值，要么转为对象以后不具备 Iterator 接口（前五个表达式），要么本身就不具备 Iterator 接口（最后一个表达式）。

对于 Set 结构，也可以使用数组的解构赋值。

let [x, y, z] = new Set(['a', 'b', 'c']);

x // "a"

### 默认值

解构赋值允许指定默认值。

let [foo = true] = [];

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

注意，ES6 内部使用严格相等运算符（===），判断一个位置是否有值。所以，只有当一个数组成员严格等于undefined，默认值才会生效。

let [x = 1] = [undefined];

x // 1let [x = 1] = [null];

x // null

上面代码中，如果一个数组成员是null，默认值就不会生效，因为null不严格等于undefined。

如果默认值是一个表达式，那么这个表达式是惰性求值的，即只有在用到的时候，才会求值。

function f() {

console.log('aaa');}

let [x = f()] = [1];

上面代码中，因为x能取到值，所以函数f根本不会执行。上面的代码其实等价于下面的代码。

let x;if ([1][0] === undefined) {

x = f();} else {

x = [1][0];}

默认值可以引用解构赋值的其他变量，但该变量必须已经声明。

let [x = 1, y = x] = []; // x=1; y=1let [x = 1, y = x] = [2]; // x=2; y=2let [x = 1, y = x] = [1, 2]; // x=1; y=2let [x = y, y = 1] = []; // ReferenceError: y is not defined

上面最后一个表达式之所以会报错，是因为x用y做默认值时，y还没有声明。

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（参见《对象的扩展》一章）。

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。

与数组一样，解构也可以用于嵌套结构的对象。

let obj = {

p: [

'Hello',

{ y: 'World' }

]};

let { p: [x, { y }] } = obj;

x // "Hello"y // "World"

注意，这时p是模式，不是变量，因此不会被赋值。如果p也要作为变量赋值，可以写成下面这样。

let obj = {

p: [

'Hello',

{ y: 'World' }

]};

let { p, p: [x, { y }] } = obj;

x // "Hello"y // "World"p // ["Hello", {y: "World"}]

下面是另一个例子。

const node = {

loc: {

start: {

line: 1,

column: 5

}

}};

let { loc, loc: { start }, loc: { start: { line }} } = node;

line // 1loc // Object {start: Object}start // Object {line: 1, column: 5}

上面代码有三次解构赋值，分别是对loc、start、line三个属性的解构赋值。注意，最后一次对line属性的解构赋值之中，只有line是变量，loc和start都是模式，不是变量。

下面是嵌套赋值的例子。

let obj = {};let arr = [];

({ foo: obj.prop, bar: arr[0] } = { foo: 123, bar: true });

obj // {prop:123}arr // [true]

如果解构模式是嵌套的对象，而且子对象所在的父属性不存在，那么将会报错。

// 报错let {foo: {bar}} = {baz: 'baz'};

上面代码中，等号左边对象的foo属性，对应一个子对象。该子对象的bar属性，解构时会报错。原因很简单，因为foo这时等于undefined，再取子属性就会报错。

注意，对象的解构赋值可以取到继承的属性。

const obj1 = {};

const obj2 = { foo: 'bar' };

Object.setPrototypeOf(obj1, obj2);

const { foo } = obj1;

foo // "bar"

上面代码中，对象obj1的原型对象是obj2。foo属性不是obj1自身的属性，而是继承自obj2的属性，解构赋值可以取到这个属性。

### 默认值

对象的解构也可以指定默认值。

var {x = 3} = {};

x // 3var {x, y = 5} = {x: 1};

x // 1y // 5var {x: y = 3} = {};

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

y // 5var { 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 // 3var {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。

### 注意点

（1）如果要将一个已经声明的变量用于解构赋值，必须非常小心。

// 错误的写法let x;{x} = {x: 1};

// SyntaxError: syntax error

上面代码的写法会报错，因为 JavaScript 引擎会将{x}理解成一个代码块，从而发生语法错误。只有不将大括号写在行首，避免 JavaScript 将其解释为代码块，才能解决这个问题。

// 正确的写法let x;({x} = {x: 1});

上面代码将整个解构赋值语句，放在一个圆括号里面，就可以正确执行。关于圆括号与解构赋值的关系，参见下文。

（2）解构赋值允许等号左边的模式之中，不放置任何变量名。因此，可以写出非常古怪的赋值表达式。

({} = [true, false]);({} = 'abc');({} = []);

上面的表达式虽然毫无意义，但是语法是合法的，可以执行。

（3）由于数组本质是特殊的对象，因此可以对数组进行对象属性的解构。

let arr = [1, 2, 3];let {0 : first, [arr.length - 1] : last} = arr;

first // 1last // 3

上面代码对数组进行对象解构。数组arr的0键对应的值是1，[arr.length - 1]就是2键，对应的值是3。方括号这种写法，属于“属性名表达式”（参见《对象的扩展》一章）。

## 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

## 数值和布尔值的解构赋值

解构赋值时，如果等号右边是数值和布尔值，则会先转为对象。

let {toString: s} = 123;

s === Number.prototype.toString // truelet {toString: s} = true;

s === Boolean.prototype.toString // true

上面代码中，数值和布尔值的包装对象都有toString属性，因此变量s都能取到值。

解构赋值的规则是，只要等号右边的值不是对象或数组，就先将其转为对象。由于undefined和null无法转为对象，所以对它们进行解构赋值，都会报错。

let { prop: x } = undefined; // TypeErrorlet { prop: y } = null; // TypeError

## 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。

注意，下面的写法会得到不一样的结果。

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

return [x, y];}

move({x: 3, y: 8}); // [3, 8]move({x: 3}); // [3, undefined]move({}); // [undefined, undefined]move(); // [0, 0]

上面代码是为函数move的参数指定默认值，而不是为变量x和y指定默认值，所以会得到与前一种写法不同的结果。

undefined就会触发函数参数的默认值。

[1, undefined, 3].map((x = 'yes') => x);

// [ 1, 'yes', 3 ]

### 不能使用圆括号的情况

以下三种解构赋值不得使用圆括号。

（1）变量声明语句

// 全部报错let [(a)] = [1];

let {x: (c)} = {};let ({x: c}) = {};let {(x: c)} = {};let {(x): c} = {};

let { o: ({ p: p }) } = { o: { p: 2 } };

上面 6 个语句都会报错，因为它们都是变量声明语句，模式不能使用圆括号。

（2）函数参数

函数参数也属于变量声明，因此不能带有圆括号。

// 报错function f([(z)]) { return z; }

// 报错function f([z,(x)]) { return x; }

（3）赋值语句的模式

// 全部报错({ p: a }) = { p: 42 };([a]) = [5];

上面代码将整个模式放在圆括号之中，导致报错。

// 报错[({ p: a }), { x: c }] = [{}, {}];

上面代码将一部分模式放在圆括号之中，导致报错。

### 可以使用圆括号的情况

可以使用圆括号的情况只有一种：赋值语句的非模式部分，可以使用圆括号。

[(b)] = [3]; // 正确({ p: (d) } = {}); // 正确[(parseInt.prop)] = [3]; // 正确

上面三行语句都可以正确执行，因为首先它们都是赋值语句，而不是声明语句；其次它们的圆括号都不属于模式的一部分。第一行语句中，模式是取数组的第一个成员，跟圆括号无关；第二行语句中，模式是p，而不是d；第三行语句与第一行语句的性质一致。

## 用途

变量的解构赋值用途很多。

****（1）交换变量的值****

let x = 1;let y = 2;

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

上面代码交换变量x和y的值，这样的写法不仅简洁，而且易读，语义非常清晰。

****（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）遍历 Map 结构****

任何部署了 Iterator 接口的对象，都可以用for...of循环遍历。Map 结构原生支持 Iterator 接口，配合变量的解构赋值，获取键名和键值就非常方便。

const map = new Map();

map.set('first', 'hello');

map.set('second', 'world');

for (let [key, value] of map) {

console.log(key + " is " + value);}

// first is hello// second is world

如果只想获取键名，或者只想获取键值，可以写成下面这样。

// 获取键名for (let [key] of map) {

// ...}

// 获取键值for (let [,value] of map) {

// ...}

****（7）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");

## 模板字符串 [§](https://es6.ruanyifeng.com/" \l "docs/string#%E6%A8%A1%E6%9D%BF%E5%AD%97%E7%AC%A6%E4%B8%B2) [⇧](https://es6.ruanyifeng.com/" \l "docs/string)

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。

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。

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"

模板字符串甚至还能嵌套。

const tmpl = addrs => `

<table>

${addrs.map(addr => `

<tr><td>${addr.first}</td></tr>

<tr><td>${addr.last}</td></tr>

`).join('')}

</table>

`;

## 实例：模板编译

下面，我们来看一个通过模板字符串，生成正式模板的实例。

let template = `<ul>

<% for(let i=0; i < data.supplies.length; i++) { %>

<li><%= data.supplies[i] %></li>

<% } %></ul>

`;

上面代码在模板字符串之中，放置了一个常规模板。该模板使用<%...%>放置 JavaScript 代码，使用<%= ... %>输出 JavaScript 表达式。

后面有些复杂Unicode 看不懂

# 字符串的新增方法

一些Unicode的 看不懂

* ****includes()****：返回布尔值，表示是否找到了参数字符串。
* ****startsWith()****：返回布尔值，表示参数字符串是否在原字符串的头部。
* ****endsWith()****：返回布尔值，表示参数字符串是否在原字符串的尾部。
* repeat（n）方法返回一个新字符串，表示将原字符串重复n次

参数如果是小数，会被取整。

'na'.repeat(2.9) // "nana"

如果repeat的参数是负数或者Infinity，会报错。

'na'.repeat(Infinity)

// RangeError'na'.repeat(-1)

// RangeError

但是，如果参数是 0 到-1 之间的小数，则等同于 0，这是因为会先进行取整运算。0 到-1 之间的小数，取整以后等于-0，repeat视同为 0。

'na'.repeat(-0.9) // ""

参数NaN等同于 0。

'na'.repeat(NaN) // ""

如果repeat的参数是字符串，则会先转换成数字。

'na'.repeat('na') // ""'na'.repeat('3') // "nanana"

## 实例方法：padStart()，padEnd()

ES2017 引入了字符串补全长度的功能。如果某个字符串不够指定长度，会在头部或尾部补全。padStart()用于头部补全，padEnd()用于尾部补全。

'x'.padStart(5, 'ab') // 'ababx''x'.padStart(4, 'ab') // 'abax''x'.padEnd(5, 'ab') // 'xabab''x'.padEnd(4, 'ab') // 'xaba'

上面代码中，padStart()和padEnd()一共接受两个参数，第一个参数是字符串补全生效的最大长度，第二个参数是用来补全的字符串。

如果原字符串的长度，等于或大于最大长度，则字符串补全不生效，返回原字符串。

'xxx'.padStart(2, 'ab') // 'xxx''xxx'.padEnd(2, 'ab') // 'xxx'

如果用来补全的字符串与原字符串，两者的长度之和超过了最大长度，则会截去超出位数的补全字符串。

'abc'.padStart(10, '0123456789')

// '0123456abc'

如果省略第二个参数，默认使用空格补全长度。

'x'.padStart(4) // ' x''x'.padEnd(4) // 'x '

padStart()的常见用途是为数值补全指定位数。下面代码生成 10 位的数值字符串。

'1'.padStart(10, '0') // "0000000001"'12'.padStart(10, '0') // "0000000012"'123456'.padStart(10, '0') // "0000123456"

另一个用途是提示字符串格式。

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

## 实例方法：trimStart()，trimEnd()

[ES2019](https://github.com/tc39/proposal-string-left-right-trim) 对字符串实例新增了trimStart()和trimEnd()这两个方法。它们的行为与trim()一致，trimStart()消除字符串头部的空格，trimEnd()消除尾部的空格。它们返回的都是新字符串，不会修改原始字符串。

const s = ' abc ';

s.trim() // "abc"s.trimStart() // "abc "s.trimEnd() // " abc"

上面代码中，trimStart()只消除头部的空格，保留尾部的空格。trimEnd()也是类似行为。

除了空格键，这两个方法对字符串头部（或尾部）的 tab 键、换行符等不可见的空白符号也有效。

浏览器还部署了额外的两个方法，trimLeft()是trimStart()的别名，trimRight()是trimEnd()的别名。

## 实例方法：replaceAll()

历史上，字符串的实例方法replace()只能替换第一个匹配。

'aabbcc'.replace('b', '\_')

// 'aa\_bcc'

上面例子中，replace()只将第一个b替换成了下划线。

## 实例方法：replaceAll()

历史上，字符串的实例方法replace()只能替换第一个匹配。

'aabbcc'.replace('b', '\_')

// 'aa\_bcc'

上面例子中，replace()只将第一个b替换成了下划线。

如果要替换所有的匹配，不得不使用正则表达式的g修饰符。

'aabbcc'.replace(/b/g, '\_')

// 'aa\_\_cc'

正则表达式毕竟不是那么方便和直观，[ES2021](https://github.com/tc39/proposal-string-replaceall) 引入了replaceAll()方法，可以一次性替换所有匹配。

'aabbcc'.replaceAll('b', '\_')

// 'aa\_\_cc'

它的用法与replace()相同，返回一个新字符串，不会改变原字符串。

String.prototype.replaceAll(searchValue, replacement)

上面代码中，searchValue是搜索模式，可以是一个字符串，也可以是一个全局的正则表达式（带有g修饰符）。

如果searchValue是一个不带有g修饰符的正则表达式，replaceAll()会报错。这一点跟replace()不同。

// 不报错'aabbcc'.replace(/b/, '\_')

// 报错'aabbcc'.replaceAll(/b/, '\_')

上面例子中，/b/不带有g修饰符，会导致replaceAll()报错。

replaceAll()的第二个参数replacement是一个字符串，表示替换的文本，其中可以使用一些特殊字符串。

* $&：匹配的字符串。
* $` ：匹配结果前面的文本。
* $'：匹配结果后面的文本。
* $n：匹配成功的第n组内容，n是从1开始的自然数。这个参数生效的前提是，第一个参数必须是正则表达式。
* $$：指代美元符号$。

下面是一些例子。

// $& 表示匹配的字符串，即`b`本身// 所以返回结果与原字符串一致'abbc'.replaceAll('b', '$&')

// 'abbc'

// $` 表示匹配结果之前的字符串// 对于第一个`b`，$` 指代`a`// 对于第二个`b`，$` 指代`ab`'abbc'.replaceAll('b', '$`')

// 'aaabc'

// $' 表示匹配结果之后的字符串// 对于第一个`b`，$' 指代`bc`// 对于第二个`b`，$' 指代`c`'abbc'.replaceAll('b', `$'`)

// 'abccc'

// $1 表示正则表达式的第一个组匹配，指代`ab`// $2 表示正则表达式的第二个组匹配，指代`bc`'abbc'.replaceAll(/(ab)(bc)/g, '$2$1')

// 'bcab'

// $$ 指代 $'abc'.replaceAll('b', '$$')

// 'a$c'

replaceAll()的第二个参数replacement除了为字符串，也可以是一个函数，该函数的返回值将替换掉第一个参数searchValue匹配的文本。

'aabbcc'.replaceAll('b', () => '\_')

// 'aa\_\_cc'

上面例子中，replaceAll()的第二个参数是一个函数，该函数的返回值会替换掉所有b的匹配。

这个替换函数可以接受多个参数。第一个参数是捕捉到的匹配内容，第二个参数捕捉到是组匹配（有多少个组匹配，就有多少个对应的参数）。此外，最后还可以添加两个参数，倒数第二个参数是捕捉到的内容在整个字符串中的位置，最后一个参数是原字符串。

const str = '123abc456';

const regex = /(\d+)([a-z]+)(\d+)/g;

function replacer(match, p1, p2, p3, offset, string) {

return [p1, p2, p3].join(' - ');}

str.replaceAll(regex, replacer)

// 123 - abc - 456

上面例子中，正则表达式有三个组匹配，所以replacer()函数的第一个参数match是捕捉到的匹配内容（即字符串123abc456），后面三个参数p1、p2、p3则依次为三个组匹配。

## at() [§](https://es6.ruanyifeng.com/" \l "docs/string-methods#%E5%AE%9E%E4%BE%8B%E6%96%B9%E6%B3%95%EF%BC%9Aat) [⇧](https://es6.ruanyifeng.com/" \l "docs/string-methods)

at()方法接受一个整数作为参数，返回参数指定位置的字符，支持负索引（即倒数的位置）。

const str = 'hello';

str.at(1) // "e"str.at(-1) // "o"

如果参数位置超出了字符串范围，at()返回undefined。

## RegExp 构造函数

在 ES5 中，RegExp构造函数的参数有两种情况。

第一种情况是，参数是字符串，这时第二个参数表示正则表达式的修饰符（flag）。

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

// 等价于var regex = /xyz/i;

第二种情况是，参数是一个正则表示式，这时会返回一个原有正则表达式的拷贝。

var regex = new RegExp(/xyz/i);

// 等价于var regex = /xyz/i;

但是，ES5 不允许此时使用第二个参数添加修饰符，否则会报错。

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

// Uncaught TypeError: Cannot supply flags when constructing one RegExp from another

ES6 改变了这种行为。如果RegExp构造函数第一个参数是一个正则对象，那么可以使用第二个参数指定修饰符。而且，返回的正则表达式会忽略原有的正则表达式的修饰符，只使用新指定的修饰符。

new RegExp(/abc/ig, 'i').flags

// "i"

上面代码中，原有正则对象的修饰符是ig，它会被第二个参数i覆盖。

## 字符串的正则方法

ES6 出现之前，字符串对象共有 4 个方法，可以使用正则表达式：match()、replace()、search()和split()。

ES6 将这 4 个方法，在语言内部全部调用RegExp的实例方法，从而做到所有与正则相关的方法，全都定义在RegExp对象上。

* String.prototype.match 调用 RegExp.prototype[Symbol.match]
* String.prototype.replace 调用 RegExp.prototype[Symbol.replace]
* String.prototype.search 调用 RegExp.prototype[Symbol.search]
* String.prototype.split 调用 RegExp.prototype[Symbol.split]

正则好复杂，跳过叭

## RegExp.prototype.sticky 属性 [§](https://es6.ruanyifeng.com/" \l "docs/regex#RegExp-prototype-sticky-%E5%B1%9E%E6%80%A7) [⇧](https://es6.ruanyifeng.com/" \l "docs/regex)

与y修饰符相匹配，ES6 的正则实例对象多了sticky属性，表示是否设置了y修饰符。

var r = /hello\d/y;

r.sticky // true

## RegExp.prototype.flags 属性

ES6 为正则表达式新增了flags属性，会返回正则表达式的修饰符。

// ES5 的 source 属性// 返回正则表达式的正文/abc/ig.source

// "abc"

// ES6 的 flags 属性// 返回正则表达式的修饰符/abc/ig.flags

// 'gi'

太复杂了 受不了了

## 二进制和八进制表示法

ES6 提供了二进制和八进制数值的新的写法，分别用前缀0b（或0B）和0o（或0O）表示。

0b111110111 === 503 // true0o767 === 503 // true

从 ES5 开始，在严格模式之中，八进制就不再允许使用前缀0表示，ES6 进一步明确，要使用前缀0o表示。

// 非严格模式(function(){

console.log(0o11 === 011);})() // true

// 严格模式(function(){

'use strict';

console.log(0o11 === 011);})() // Uncaught SyntaxError: Octal literals are not allowed in strict mode.

如果要将0b和0o前缀的字符串数值转为十进制，要使用Number方法。

Number('0b111') // 7Number('0o10') // 8

## 数值分隔符

欧美语言中，较长的数值允许每三位添加一个分隔符（通常是一个逗号），增加数值的可读性。比如，1000可以写作1,000。

[ES2021](https://github.com/tc39/proposal-numeric-separator)，允许 JavaScript 的数值使用下划线（\_）作为分隔符。

let budget = 1\_000\_000\_000\_000;

budget === 10 \*\* 12 // true

这个数值分隔符没有指定间隔的位数，也就是说，可以每三位添加一个分隔符，也可以每一位、每两位、每四位添加一个。

123\_00 === 12\_300 // true

12345\_00 === 123\_4500 // true12345\_00 === 1\_234\_500 // true

小数和科学计数法也可以使用数值分隔符。

// 小数0.000\_001

// 科学计数法1e10\_000

数值分隔符有几个使用注意点。

* 不能放在数值的最前面（leading）或最后面（trailing）。
* 不能两个或两个以上的分隔符连在一起。
* 小数点的前后不能有分隔符。
* 科学计数法里面，表示指数的e或E前后不能有分隔符。

下面的写法都会报错。

// 全部报错3\_.1413.\_141

1\_e12

1e\_12

123\_\_456

\_1464301

1464301\_

除了十进制，其他进制的数值也可以使用分隔符。

// 二进制0b1010\_0001\_1000\_0101

// 十六进制0xA0\_B0\_C0

可以看到，数值分隔符可以按字节顺序分隔数值，这在操作二进制位时，非常有用。

注意，分隔符不能紧跟着进制的前缀0b、0B、0o、0O、0x、0X。

// 报错0\_b111111000

0b\_111111000

数值分隔符只是一种书写便利，对于 JavaScript 内部数值的存储和输出，并没有影响。

let num = 12\_345;

num // 12345num.toString() // 12345

上面示例中，变量num的值为12\_345，但是内部存储和输出的时候，都不会有数值分隔符。

下面三个将字符串转成数值的函数，不支持数值分隔符。主要原因是语言的设计者认为，数值分隔符主要是为了编码时书写数值的方便，而不是为了处理外部输入的数据。

* Number()
* parseInt()
* parseFloat()

Number('123\_456') // NaNparseInt('123\_456') // 123

## Number.isFinite(), Number.isNaN()

ES6 在Number对象上，新提供了Number.isFinite()和Number.isNaN()两个方法。

Number.isFinite()用来检查一个数值是否为有限的（finite），即不是Infinity。

Number.isFinite(15); // true

Number.isFinite(0.8); // true

Number.isFinite(NaN); // false

Number.isFinite(Infinity); // false

Number.isFinite(-Infinity); // false

Number.isFinite('foo'); // false

Number.isFinite('15'); // false

Number.isFinite(true); // false

注意，如果参数类型不是数值，Number.isFinite一律返回false。

Number.isNaN()用来检查一个值是否为NaN。

Number.isNaN(NaN) // true

Number.isNaN(15) // false

Number.isNaN('15') // false

Number.isNaN(true) // false

Number.isNaN(9/NaN) // true

Number.isNaN('true' / 0) // true

Number.isNaN('true' / 'true') // true

如果参数类型不是NaN，Number.isNaN一律返回false。

它们与传统的全局方法isFinite()和isNaN()的区别在于，传统方法先调用Number()将非数值的值转为数值，再进行判断，而这两个新方法只对数值有效，Number.isFinite()对于非数值一律返回false, Number.isNaN()只有对于NaN才返回true，非NaN一律返回false。

isFinite(25) // true

isFinite("25") // true

Number.isFinite(25) // true

Number.isFinite("25") // false

isNaN(NaN) // true

isNaN("NaN") // true

Number.isNaN(NaN) // true

Number.isNaN("NaN") // false

Number.isNaN(1) // false

## Number.parseInt(), Number.parseFloat()

ES6 将全局方法parseInt()和parseFloat()，移植到Number对象上面，行为完全保持不变。

// ES5的写法parseInt('12.34') // 12parseFloat('123.45#') // 123.45

// ES6的写法Number.parseInt('12.34') // 12Number.parseFloat('123.45#') // 123.45

这样做的目的，是逐步减少全局性方法，使得语言逐步模块化。

Number.parseInt === parseInt // trueNumber.parseFloat === parseFloat // true

## Number.isInteger()

Number.isInteger()用来判断一个数值是否为整数。

Number.isInteger(25) // trueNumber.isInteger(25.1) // false

JavaScript 内部，整数和浮点数采用的是同样的储存方法，所以 25 和 25.0 被视为同一个值。

Number.isInteger(25) // trueNumber.isInteger(25.0) // true

如果参数不是数值，Number.isInteger返回false。

Number.isInteger() // falseNumber.isInteger(null) // falseNumber.isInteger('15') // falseNumber.isInteger(true) // false

注意，由于 JavaScript 采用 IEEE 754 标准，数值存储为64位双精度格式，数值精度最多可以达到 53 个二进制位（1 个隐藏位与 52 个有效位）。如果数值的精度超过这个限度，第54位及后面的位就会被丢弃，这种情况下，Number.isInteger可能会误判。

Number.isInteger(3.0000000000000002) // true

上面代码中，Number.isInteger的参数明明不是整数，但是会返回true。原因就是这个小数的精度达到了小数点后16个十进制位，转成二进制位超过了53个二进制位，导致最后的那个2被丢弃了。

类似的情况还有，如果一个数值的绝对值小于Number.MIN\_VALUE（5E-324），即小于 JavaScript 能够分辨的最小值，会被自动转为 0。这时，Number.isInteger也会误判。

Number.isInteger(5E-324) // falseNumber.isInteger(5E-325) // true

上面代码中，5E-325由于值太小，会被自动转为0，因此返回true。

总之，如果对数据精度的要求较高，不建议使用Number.isInteger()判断一个数值是否为整数。

## Math 对象的扩展 [§](https://es6.ruanyifeng.com/" \l "docs/number#Math-%E5%AF%B9%E8%B1%A1%E7%9A%84%E6%89%A9%E5%B1%95) [⇧](https://es6.ruanyifeng.com/" \l "docs/number)

### Math.trunc()

Math.trunc方法用于去除一个数的小数部分，返回整数部分。

Math.trunc(4.1) // 4

Math.trunc(4.9) // 4

Math.trunc(-4.1) // -4

Math.trunc(-4.9) // -4

Math.trunc(-0.1234) // -0

对于非数值，Math.trunc内部使用Number方法将其先转为数值。

Math.trunc('123.456') // 123

Math.trunc(true) //1

Math.trunc(false) // 0

Math.trunc(null) // 0

对于空值和无法截取整数的值，返回NaN。

Math.trunc(NaN); // NaN

Math.trunc('foo'); // NaN

Math.trunc(); // NaN

Math.trunc(undefined) // NaN

对于没有部署这个方法的环境，可以用下面的代码模拟。

Math.trunc = Math.trunc || function(x) {

return x < 0 ? Math.ceil(x) : Math.floor(x);};

### Math.sign()

Math.sign方法用来判断一个数到底是正数、负数、还是零。对于非数值，会先将其转换为数值。

它会返回五种值。

* 参数为正数，返回+1；
* 参数为负数，返回-1；
* 参数为 0，返回0；
* 参数为-0，返回-0;
* 其他值，返回NaN。

Math.sign(-5) // -1Math.sign(5) // +1Math.sign(0) // +0Math.sign(-0) // -0Math.sign(NaN) // NaN

如果参数是非数值，会自动转为数值。对于那些无法转为数值的值，会返回NaN。

Math.sign('') // 0Math.sign(true) // +1Math.sign(false) // 0Math.sign(null) // 0Math.sign('9') // +1Math.sign('foo') // NaNMath.sign() // NaNMath.sign(undefined) // NaN

对于没有部署这个方法的环境，可以用下面的代码模拟。

Math.sign = Math.sign || function(x) {

x = +x; // convert to a number if (x === 0 || isNaN(x)) {

return x;

}

return x > 0 ? 1 : -1;};

### Math.cbrt()

Math.cbrt()方法用于计算一个数的立方根。

Math.cbrt(-1) // -1

Math.cbrt(0) // 0

Math.cbrt(1) // 1

Math.cbrt(2) // 1.2599210498948732

对于非数值，Math.cbrt()方法内部也是先使用Number()方法将其转为数值。

Math.cbrt('8') // 2

Math.cbrt('hello') // NaN

对于没有部署这个方法的环境，可以用下面的代码模拟。

Math.cbrt = Math.cbrt || function(x) {

var y = Math.pow(Math.abs(x), 1/3);

return x < 0 ? -y : y;};

对数 立方根 等等太复杂了

## 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 Worldlog('Hello', 'China') // Hello Chinalog('Hello', '') // Hello World

使用参数默认值时，函数不能有同名参数。

// 不报错function foo(x, x, y) {

// ...}

// 报错function foo(x, x, y = 1) {

// ...}

一个容易忽略的地方是，参数默认值不是传值的，而是每次都重新计算默认值表达式的值。也就是说，参数默认值是惰性求值的。

let x = 99;function foo(p = x + 1) {

console.log(p);}

foo() // 100

x = 100;foo() // 101

上面代码中，参数p的默认值是x + 1。这时，每次调用函数foo()，都会重新计算x + 1，而不是默认p等于 100。

### 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 5foo({x: 1}) // 1 5foo({x: 1, y: 2}) // 1 2foo() // 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。

下面是另一个解构赋值默认值的例子。

function fetch(url, { body = '', method = 'GET', headers = {} }) {

console.log(method);}

fetch('http://example.com', {})

// "GET"fetch('http://example.com')

// 报错

上面代码中，如果函数fetch()的第二个参数是一个对象，就可以为它的三个属性设置默认值。这种写法不能省略第二个参数，如果结合函数参数的默认值，就可以省略第二个参数。这时，就出现了双重默认值。

function fetch(url, { body = '', method = 'GET', headers = {} } = {}) {

console.log(method);}

fetch('http://example.com')

// "GET"

上面代码中，函数fetch没有第二个参数时，函数参数的默认值就会生效，然后才是解构赋值的默认值生效，变量method才会取到默认值GET。

注意，函数参数的默认值生效以后，参数解构赋值依然会进行。

function f({ a, b = 'world' } = { a: 'hello' }) {

console.log(b);}

f() // world

上面示例中，函数f()调用时没有参数，所以参数默认值{ a: 'hello' }生效，然后再对这个默认值进行解构赋值，从而触发参数变量b的默认值生效。

### 参数默认值的位置

通常情况下，定义了默认值的参数，应该是函数的尾参数。因为这样比较容易看出来，到底省略了哪些参数。如果非尾部的参数设置默认值，实际上这个参数是没法省略的。

// 例一function f(x = 1, y) {

return [x, y];}

f() // [1, undefined]f(2) // [2, undefined]f(, 1) // 报错f(undefined, 1) // [1, 1]

// 例二function f(x, y = 5, z) {

return [x, y, z];}

f() // [undefined, 5, undefined]f(1) // [1, 5, undefined]f(1, ,2) // 报错f(1, undefined, 2) // [1, 5, 2]

上面代码中，有默认值的参数都不是尾参数。这时，无法只省略该参数，而不省略它后面的参数，除非显式输入undefined。

如果传入undefined，将触发该参数等于默认值，null则没有这个效果。

function foo(x = 5, y = 6) {

console.log(x, y);}

foo(undefined, null)

// 5 null

上面代码中，x参数对应undefined，结果触发了默认值，y参数等于null，就没有触发默认值。

### 函数的 length 属性

指定了默认值以后，函数的length属性，将返回没有指定默认值的参数个数。也就是说，指定了默认值后，length属性将失真。

(function (a) {}).length // 1(function (a = 5) {}).length // 0(function (a, b, c = 5) {}).length // 2

上面代码中，length属性的返回值，等于函数的参数个数减去指定了默认值的参数个数。比如，上面最后一个函数，定义了 3 个参数，其中有一个参数c指定了默认值，因此length属性等于3减去1，最后得到2。

这是因为length属性的含义是，该函数预期传入的参数个数。某个参数指定默认值以后，预期传入的参数个数就不包括这个参数了。同理，后文的 rest 参数也不会计入length属性。

(function(...args) {}).length // 0

如果设置了默认值的参数不是尾参数，那么length属性也不再计入后面的参数了。

(function (a = 0, b, c) {}).length // 0(function (a, b = 1, c) {}).length // 1

### 作用域

一旦设置了参数的默认值，函数进行声明初始化时，参数会形成一个单独的作用域（context）。等到初始化结束，这个作用域就会消失。这种语法行为，在不设置参数默认值时，是不会出现的。

var x = 1;

function f(x, y = x) {

console.log(y);}

f(2) // 2

上面代码中，参数y的默认值等于变量x。调用函数f时，参数形成一个单独的作用域。在这个作用域里面，默认值变量x指向第一个参数x，而不是全局变量x，所以输出是2。

再看下面的例子。

let x = 1;

function f(y = x) {

let x = 2;

console.log(y);}

f() // 1

上面代码中，函数f调用时，参数y = x形成一个单独的作用域。这个作用域里面，变量x本身没有定义，所以指向外层的全局变量x。函数调用时，函数体内部的局部变量x影响不到默认值变量x。

如果此时，全局变量x不存在，就会报错。

function f(y = x) {

let x = 2;

console.log(y);}

f() // ReferenceError: x is not defined

下面这样写，也会报错。

var x = 1;

function foo(x = x) {

// ...}

foo() // ReferenceError: Cannot access 'x' before initialization

上面代码中，参数x = x形成一个单独作用域。实际执行的是let x = x，由于暂时性死区的原因，这行代码会报错。

如果参数的默认值是一个函数，该函数的作用域也遵守这个规则。请看下面的例子。

let foo = 'outer';

function bar(func = () => foo) {

let foo = 'inner';

console.log(func());}

bar(); // outer

上面代码中，函数bar的参数func的默认值是一个匿名函数，返回值为变量foo。函数参数形成的单独作用域里面，并没有定义变量foo，所以foo指向外层的全局变量foo，因此输出outer。

如果写成下面这样，就会报错。

function bar(func = () => foo) {

let foo = 'inner';

console.log(func());}

bar() // ReferenceError: foo is not defined

上面代码中，匿名函数里面的foo指向函数外层，但是函数外层并没有声明变量foo，所以就报错了。

下面是一个更复杂的例子。

var x = 1;function foo(x, y = function() { x = 2; }) {

var x = 3;

y();

console.log(x);}

foo() // 3x // 1

上面代码中，函数foo的参数形成一个单独作用域。这个作用域里面，首先声明了变量x，然后声明了变量y，y的默认值是一个匿名函数。这个匿名函数内部的变量x，指向同一个作用域的第一个参数x。函数foo内部又声明了一个内部变量x，该变量与第一个参数x由于不是同一个作用域，所以不是同一个变量，因此执行y后，内部变量x和外部全局变量x的值都没变。

如果将var x = 3的var去除，函数foo的内部变量x就指向第一个参数x，与匿名函数内部的x是一致的，所以最后输出的就是2，而外层的全局变量x依然不受影响。

var x = 1;function foo(x, y = function() { x = 2; }) {

x = 3;

y();

console.log(x);}

foo() // 2x // 1

### 应用

利用参数默认值，可以指定某一个参数不得省略，如果省略就抛出一个错误。

function throwIfMissing() {

throw new Error('Missing parameter');}

function foo(mustBeProvided = throwIfMissing()) {

return mustBeProvided;}

foo()

// Error: Missing parameter

上面代码的foo函数，如果调用的时候没有参数，就会调用默认值throwIfMissing函数，从而抛出一个错误。

从上面代码还可以看到，参数mustBeProvided的默认值等于throwIfMissing函数的运行结果（注意函数名throwIfMissing之后有一对圆括号），这表明参数的默认值不是在定义时执行，而是在运行时执行。如果参数已经赋值，默认值中的函数就不会运行。

另外，可以将参数默认值设为undefined，表明这个参数是可以省略的。

function foo(optional = undefined) { ··· }

## 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.from(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.from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) {

// ...}

函数的length属性，不包括 rest 参数。

(function(a) {}).length // 1(function(...a) {}).length // 0(function(a, ...b) {}).length // 1

## 严格模式

从 ES5 开始，函数内部可以设定为严格模式。

function doSomething(a, b) {

'use strict';

// code}

ES2016 做了一点修改，规定只要函数参数使用了默认值、解构赋值、或者扩展运算符，那么函数内部就不能显式设定为严格模式，否则会报错。

// 报错function doSomething(a, b = a) {

'use strict';

// code}

// 报错const doSomething = function ({a, b}) {

'use strict';

// code};

// 报错const doSomething = (...a) => {

'use strict';

// code};

const obj = {

// 报错 doSomething({a, b}) {

'use strict';

// code }};

这样规定的原因是，函数内部的严格模式，同时适用于函数体和函数参数。但是，函数执行的时候，先执行函数参数，然后再执行函数体。这样就有一个不合理的地方，只有从函数体之中，才能知道参数是否应该以严格模式执行，但是参数却应该先于函数体执行。

// 报错function doSomething(value = 070) {

'use strict';

return value;}

上面代码中，参数value的默认值是八进制数070，但是严格模式下不能用前缀0表示八进制，所以应该报错。但是实际上，JavaScript 引擎会先成功执行value = 070，然后进入函数体内部，发现需要用严格模式执行，这时才会报错。

虽然可以先解析函数体代码，再执行参数代码，但是这样无疑就增加了复杂性。因此，标准索性禁止了这种用法，只要参数使用了默认值、解构赋值、或者扩展运算符，就不能显式指定严格模式。

两种方法可以规避这种限制。第一种是设定全局性的严格模式，这是合法的。

'use strict';

function doSomething(a, b = a) {

// code}

第二种是把函数包在一个无参数的立即执行函数里面。

const doSomething = (function () {

'use strict';

return function(value = 42) {

return value;

};}());

## name 属性

函数的name属性，返回该函数的函数名。

function foo() {}

foo.name // "foo"

这个属性早就被浏览器广泛支持，但是直到 ES6，才将其写入了标准。

需要注意的是，ES6 对这个属性的行为做出了一些修改。如果将一个匿名函数赋值给一个变量，ES5 的name属性，会返回空字符串，而 ES6 的name属性会返回实际的函数名。

var f = function () {};

// ES5f.name // ""

// ES6f.name // "f"

上面代码中，变量f等于一个匿名函数，ES5 和 ES6 的name属性返回的值不一样。

如果将一个具名函数赋值给一个变量，则 ES5 和 ES6 的name属性都返回这个具名函数原本的名字。

const bar = function baz() {};

// ES5bar.name // "baz"

// ES6bar.name // "baz"

Function构造函数返回的函数实例，name属性的值为anonymous。

(new Function).name // "anonymous"

bind返回的函数，name属性值会加上bound前缀。

function foo() {};

foo.bind({}).name // "bound foo"(function(){}).bind({}).name // "bound "

## 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。

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" });

下面是一种特殊情况，虽然可以运行，但会得到错误的结果。

let foo = () => { a: 1 };foo() // undefined

上面代码中，原始意图是返回一个对象{ a: 1 }，但是由于引擎认为大括号是代码块，所以执行了一行语句a: 1。这时，a可以被解释为语句的标签，因此实际执行的语句是1;，然后函数就结束了，没有返回值。

如果箭头函数只有一行语句，且不需要返回值，可以采用下面的写法，就不用写大括号了。

let fn = () => void doesNotReturn();

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;}

箭头函数使得表达更加简洁。

const isEven = n => n % 2 === 0;

const square = n => n \* n;

上面代码只用了两行，就定义了两个简单的工具函数。如果不用箭头函数，可能就要占用多行，而且还不如现在这样写醒目。

箭头函数的一个用处是简化回调函数。

// 普通函数写法[1,2,3].map(function (x) {

return x \* x;});

// 箭头函数写法[1,2,3].map(x => x \* x);

另一个例子是

// 普通函数写法var result = values.sort(function (a, b) {

return a - b;});

// 箭头函数写法var result = values.sort((a, b) => a - b);

下面是 rest 参数与箭头函数结合的例子。

const numbers = (...nums) => nums;

numbers(1, 2, 3, 4, 5)

// [1,2,3,4,5]

const headAndTail = (head, ...tail) => [head, tail];

headAndTail(1, 2, 3, 4, 5)

// [1,[2,3,4,5]]