# 介绍

ECMAScript 6，也称为 ECMAScript 2015，是 ECMAScript 标准的最新版本。ES6 是该语言的一次重大更新，也是自 2009 年 ES5 标准化以来对该语言的第一次更新。这些功能在主要 JavaScript 引擎中是实现。

# 箭头

箭头是使用=>语法的函数简写。它们在语法上类似于 C#、Java 8 和 CoffeeScript 中的相关功能。它们支持语句块体以及返回表达式值的表达式体。this与函数不同，箭头与其周围的代码共享相同的词法。

// Expression bodies

var odds = evens.map(v => v + 1);

var nums = evens.map((v, i) => v + i);

var pairs = evens.map(v => ({even: v, odd: v + 1}));

// Statement bodies

nums.forEach(v => {

if (v % 5 === 0)

fives.push(v);

});

// Lexical this

var bob = {

\_name: "Bob",

\_friends: [],

printFriends() {

this.\_friends.forEach(f =>

console.log(this.\_name + " knows " + f));

}

}

箭头函数表达式的语法比函数表达式更简洁，并且没有自己的this，arguments，super或new.target。箭头函数表达式更适用于那些本来需要匿名函数的地方，并且它不能用作构造函数。

## 语法

### 基础语法

(param1, param2, …, paramN) => { statements }

(param1, param2, …, paramN) => expression

//相当于：(param1, param2, …, paramN) =>{ return expression; }

// 当只有一个参数时，圆括号是可选的：

(singleParam) => { statements }

singleParam => { statements }

// 没有参数的函数应该写成一对圆括号。

() => { statements }

### 高级语法

//加括号的函数体返回对象字面量表达式：

params => ({foo: bar})

//支持剩余参数和默认参数

(param1, param2, ...rest) => { statements }

(param1 = defaultValue1, param2, …, paramN = defaultValueN) => {

statements }

//同样支持参数列表解构

let f = ([a, b] = [1, 2], {x: c} = {x: a + b}) => a + b + c;

f(); // 6

## 描述

引入箭头函数有两个方面的作用：更简短的函数并且不绑定this。

### 更短的函数

var elements = [

'Hydrogen',

'Helium',

'Lithium',

'Beryllium'

];

elements.map(function(element) {

return element.length;

}); // 返回数组：[8, 6, 7, 9]

// 上面的普通函数可以改写成如下的箭头函数

elements.map((element) => {

return element.length;

}); // [8, 6, 7, 9]

// 当箭头函数只有一个参数时，可以省略参数的圆括号

elements.map(element => {

return element.length;

}); // [8, 6, 7, 9]

// 当箭头函数的函数体只有一个 `return` 语句时，可以省略 `return` 关键字和方法体的花括号

elements.map(element => element.length); // [8, 6, 7, 9]

// 在这个例子中，因为我们只需要 `length` 属性，所以可以使用参数解构

// 需要注意的是字符串 `"length"` 是我们想要获得的属性的名称，而 `lengthFooBArX` 则只是个变量名，

// 可以替换成任意合法的变量名

elements.map(({ "length": lengthFooBArX }) => lengthFooBArX); // [8, 6, 7, 9]

### 没有单独的this

在箭头函数出现之前，每一个新函数根据它是被如何调用的来定义这个函数的 this 值：

* 如果该函数是一个构造函数，this 指针指向一个新的对象
* 在严格模式下的函数调用下，this 指向undefined
* 如果该函数是一个对象的方法，则它的 this 指针指向这个对象

This被证明是令人厌烦的面向对象风格的编程。

function Person() {

// Person() 构造函数定义 `this`作为它自己的实例。

this.age = 0;

setInterval(function growUp() {

// 在非严格模式，growUp() 函数定义 `this`作为全局对象，

// 与在 Person() 构造函数中定义的 `this`并不相同。

this.age++;

}, 1000);

}

var p = new Person();

在ECMAScript 3/5 中，通过将this值分配给封闭的变量，可以解决this问题。

function Person() {

var that = this;

that.age = 0;

setInterval(function growUp() {

// 回调引用的是`that`变量，其值是预期的对象。

that.age++;

}, 1000);

}

可以创建绑定函数，以便将预先分配的this值传递到绑定的目标函数（上述示例中的growUp()函数）。

箭头函数不会创建自己的this，它只会从自己的作用域链的上一层继承 this。因此，在下面的代码中，传递给setInterval的函数内的this与封闭函数中的this值相同：

function Person(){

this.age = 0;

setInterval(() => {

this.age++; // |this| 正确地指向 p 实例

}, 1000);

}

var p = new Person();

### 与严格模式的关系

鉴于 this 是词法层面上的，严格模式中与 this 相关的规则都将被忽略。

var f = () => { 'use strict'; return this; };

f() === window; // 或者 global

严格模式的其他规则依然不变。

### 调用

由于箭头函数没有自己的 this 指针，通过call()或apply()方法调用一个函数时，只能传递参数（不能绑定this---译者注），他们的第一个参数会被忽略。（这种现象对于bind方法同样成立）

var adder = {

base : 1,

add : function(a) {

var f = v => v + this.base;

return f(a);

},

addThruCall: function(a) {

var f = v => v + this.base;

var b = {

base : 2

};

return f.call(b, a);

}

};

console.log(adder.add(1)); // 输出 2

console.log(adder.addThruCall(1)); // 仍然输出 2

### 不绑定arguments

箭头函数不绑定Arguments 对象。因此，在本示例中，arguments只是引用了封闭作用域内的 arguments：

var arguments = [1, 2, 3];

var arr = () => arguments[0];

arr(); // 1

function foo(n) {

var f = () => arguments[0] + n; // 隐式绑定 foo 函数的 arguments 对象. arguments[0] 是 n，即传给 foo 函数的第一个参数

return f();

}

foo(1); // 2

foo(2); // 4

foo(3); // 6

foo(3,2);//6

在大多数情况下，使用剩余参数是相较使用arguments对象的更好选择。

function foo(arg) {

var f = (...args) => args[0];

return f(arg);

}

foo(1); // 1

function foo(arg1,arg2) {

var f = (...args) => args[1];

return f(arg1,arg2);

}

foo(1,2); //2

### 箭头函数作为方法

如上所述，箭头函数表达式对非方法函数是最合适的。但是也可以作为方法来使用的。

'use strict';

var obj = {

i: 10,

b: () => console.log(this.i, this),

c: function() {

console.log( this.i, this)

}

}

obj.b();

// undefined, Window{...}

obj.c();

// 10, Object {...}

箭头函数没有定义 this 绑定。

'use strict';

var obj = {

a: 10

};

Object.defineProperty(obj, "b", {

get: () => {

console.log(this.a, typeof this.a, this);

return this.a+10;

// 代表全局对象 'Window', 因此 'this.a' 返回 'undefined'

}

});

obj.b; // undefined "undefined" Window {postMessage: ƒ, blur: ƒ, focus: ƒ, close: ƒ, frames: Window, …}

### 使用 new 操作符

箭头函数不能用作构造器，和 new一起用会抛出错误。

var Foo = () => {};

var foo = new Foo(); // TypeError: Foo is not a constructor

Copy to Clipboard

### 使用prototype属性

箭头函数没有prototype属性。

var Foo = () => {};

console.log(Foo.prototype); // undefined

Copy to Clipboard

### 使用 yield 关键字

yield 关键字通常不能在箭头函数中使用（除非是嵌套在允许使用的函数内）。因此，箭头函数不能用作函数生成器。

## 函数体

箭头函数可以有一个“简写体”或常见的“块体”。在一个简写体中，只需要一个表达式，并附加一个隐式的返回值。在块体中，必须使用明确的return语句。

var func = x => x \* x;

// 简写函数 省略 return

var func = (x, y) => { return x + y; };

//常规编写 明确的返回值

## 返回对象字面量

用params => {object:literal}这种简单的语法返回对象字面量是行不通的。

var func = () => { foo: 1 };

// Calling func() returns undefined!

var func = () => { foo: function() {} };

// SyntaxError: function statement requires a name Copy to Clipboard

这是因为花括号（{} ）里面的代码被解析为一系列语句（即 foo 被认为是一个标签，而非对象字面量的组成部分）。所以，得用圆括号把对象字面量包起来：

var func = () => ({foo: 1});

## 换行

箭头函数在参数和箭头之间不能换行。

var func = ()

=> 1;

// SyntaxError: expected expression, got '=>'

Copy to Clipboard

但是，可以通过在 ‘=>’ 之后换行，或者用 ‘( )’、'{ }'来实现换行，如下：

var func = (a, b, c) =>

1;

var func = (a, b, c) => (

1

);

var func = (a, b, c) => {

return 1

};

var func = (

a,

b,

c

) => 1;

// 不会有语法错误

## 解析顺序

虽然箭头函数中的箭头不是运算符，但箭头函数具有与常规函数不同的特殊运算符优先级解析规则。

# 类

类是用于创建对象的模板。他们用代码封装数据以处理该数据。 JS 中的类建立在原型上，但也具有某些语法和语义未与 ES5 类相似语义共享。

## 定义类

实际上，类是“特殊的函数”，就像你能够定义的函数表达式和函数声明一样，类语法有两个组成部分：类表达式和类声明。

### 类声明

定义类的一种方法是使用类声明。要声明一个类，你可以使用带有class关键字的类名

class Rectangle {

constructor(height, width) {

this.height = height;

this.width = width;

}

}

函数声明和类声明之间的一个重要区别在于，函数声明会提升，类声明不会。你首先需要声明你的类，然后再访问它，否则类似以下的代码将抛出ReferenceError：

let p = new Rectangle(); // ReferenceError

class Rectangle {}

### 类表达式

类表达式是定义类的另一种方法。类表达式可以命名或不命名。命名类表达式的名称是该类体的局部名称。

// 未命名/匿名类

let Rectangle = class {

constructor(height, width) {

this.height = height;

this.width = width;

}

};

console.log(Rectangle.name);

// output: "Rectangle"

// 命名类

let Rectangle = class Rectangle2 {

constructor(height, width) {

this.height = height;

this.width = width;

}

};

console.log(Rectangle.name);

// 输出: "Rectangle2"

## 类体和方法定义

一个类的类体是一对花括号/大括号 {} 中的部分。这是你定义类成员的位置，如方法或构造函数。

### 严格模式

类声明和类表达式的主体都执行在严格模式下。比如，构造函数，静态方法，原型方法，getter 和 setter 都在严格模式下执行。

### 构造函数

constructor方法是一个特殊的方法，这种方法用于创建和初始化一个由class创建的对象。一个类只能拥有一个名为 “constructor”的特殊方法。如果类包含多个constructor的方法，则将抛出 一个SyntaxError 。一个构造函数可以使用 super 关键字来调用一个父类的构造函数。

### 原型方法

class Rectangle {

// constructor

constructor(height, width) {

this.height = height;

this.width = width;

}

// Getter

get area() {

return this.calcArea()

}

// Method

calcArea() {

return this.height \* this.width;

}

}

const square = new Rectangle(10, 10);

console.log(square.area);

// 100

### 静态方法

static 关键字用来定义一个类的一个静态方法。调用静态方法不需要实例化该类，但不能通过一个类实例调用静态方法。静态方法通常用于为一个应用程序创建工具函数。

class Point {

constructor(x, y) {

this.x = x;

this.y = y;

}

static displayName = "Point";

static distance(a, b) {

const dx = a.x - b.x;

const dy = a.y - b.y;

return Math.hypot(dx, dy);

}

}

const p1 = new Point(5, 5);

const p2 = new Point(10,10);

p1.displayName;

// undefined

p1.distance;

// undefined

console.log(Point.displayName);

// "Point"

console.log(Point.distance(p1, p2));

// 7.0710678118654755

### 绑定 this

当调用静态或原型方法时没有指定 this 的值，那么方法内的 this 值将被置为 undefined。即使你未设置 "use strict" ，因为 class 体内部的代码总是在严格模式下执行。

class Animal {

speak() {

return this;

}

static eat() {

return this;

}

}

let obj = new Animal();

obj.speak(); // Animal {}

let speak = obj.speak;

speak(); // undefined

Animal.eat() // class Animal

let eat = Animal.eat;

eat(); // undefined

上述代码通过传统的基于函数的语法来实现，那么依据初始的 this 值，在非严格模式下方法调用会发生自动装箱。若初始值是 undefined，this 值会被设为全局对象。严格模式下不会发生自动装箱，this 值将保留传入状态。

function Animal() { }

Animal.prototype.speak = function() {

return this;

}

Animal.eat = function() {

return this;

}

let obj = new Animal();

let speak = obj.speak;

speak(); // global object

let eat = Animal.eat;

eat(); // global object

### 实例属性

实例的属性必须定义在类的方法里：

class Rectangle {

constructor(height, width) {

this.height = height;

this.width = width;

}

}

静态的或原型的数据属性必须定义在类定义的外面。

Rectangle.staticWidth = 20;

Rectangle.prototype.prototypeWidth = 25;

### 字段声明

公共和私有字段声明是 JavaScript 标准委员会TC39提出的实验性功能（第 3 阶段）。浏览器中的支持是有限的，但是可以通过Babel等系统构建后使用此功能。

使用 JavaScript 字段声明语法声明共有字段，上面的示例可以写成：

class Rectangle {

height = 0;

width;

constructor(height, width) {

this.height = height;

this.width = width;

}

}

通过预先声明字段，类定义变得更加自我记录，并且字段始终存在。正如上面看到的，这个字段可以用也可以不用默认值来声明。

使用私有字段，可以按以下方式细化定义。

class Rectangle {

#height = 0;

#width;

constructor(height, width) {

this.#height = height;

this.#width = width;

}

}

从类外部引用私有字段是错误的。它们只能在类里面中读取或写入。通过定义在类外部不可见的内容，可以确保类的用户不会依赖于内部，因为内部可能在不同版本之间发生变化。私有字段仅能在字段声明中预先定义。

## extends 扩展子类

extends 关键字在 类声明 或 类表达式 中用于创建一个类作为另一个类的一个子类。

class Animal {

constructor(name) {

this.name = name;

}

speak() {

console.log(`${this.name} makes a noise.`);

}

}

class Dog extends Animal {

constructor(name) {

super(name); // 调用超类构造函数并传入 name 参数

}

speak() {

console.log(`${this.name} barks.`);

}

}

var d = new Dog('Mitzie');

d.speak();// 'Mitzie barks.'

如果子类中定义了构造函数，那么它必须先调用 super() 才能使用 this 。也可以继承传统的基于函数的“类”：

function Animal (name) {

this.name = name;

}

Animal.prototype.speak = function () {

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

}

class Dog extends Animal {

speak() {

super.speak();

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

}

}

var d = new Dog('Mitzie');

d.speak();//Mitzie makes a noise. Mitzie barks.

Copy to Clipboard

请注意，类不能继承常规对象（不可构造的）。如果要继承常规对象，可以改用Object.setPrototypeOf()：

var Animal = {

speak() {

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

}

};

class Dog {

constructor(name) {

this.name = name;

}

}

Object.setPrototypeOf(Dog.prototype, Animal);// 如果不这样做，在调用 speak 时会返回 TypeError

var d = new Dog('Mitzie');

d.speak(); // Mitzie makes a noise.

## Species

你可能希望在派生数组类 MyArray 中返回 Array对象。这种 species 方式允许你覆盖默认的构造函数。例如，当使用像map()返回默认构造函数的方法时，您希望这些方法返回一个父Array对象，而不是MyArray对象。Symbol.species 符号可以让你这样做：

class MyArray extends Array {

// Overwrite species to the parent Array constructor

static get [Symbol.species]() { return Array; }

}

var a = new MyArray(1,2,3);

var mapped = a.map(x => x \* x);

console.log(mapped instanceof MyArray);

// false

console.log(mapped instanceof Array);

// true

## super 调用超类

super 关键字用于调用对象的父对象上的函数。

class Cat {

constructor(name) {

this.name = name;

}

speak() {

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

}

}

class Lion extends Cat {

speak() {

super.speak();

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

}

}

## Mix-ins / 混入

抽象子类或者 mix-ins 是类的模板。 一个 ECMAScript 类只能有一个单超类，所以想要从工具类来多重继承的行为是不可能的。子类继承的只能是父类提供的功能性。因此，例如，从工具类的多重继承是不可能的。该功能必须由超类提供。

一个以超类作为输入的函数和一个继承该超类的子类作为输出可以用于在 ECMAScript 中实现混合：

var calculatorMixin = Base => class extends Base {

calc() { }

};

var randomizerMixin = Base => class extends Base {

randomize() { }

};

Copy to Clipboard

使用 mix-ins 的类可以像下面这样写：

class Foo { }

class Bar extends calculatorMixin(randomizerMixin(Foo)) { }

# 增强的对象文字

对象文字被扩展为支持在构造时设置原型、foo: foo赋值的简写、定义方法、进行超级调用以及使用表达式计算属性名称。同时，它们还使对象字面量和类声明更紧密地结合在一起，并让基于对象的设计受益于一些相同的便利。

var obj = {

// \_\_proto\_\_

\_\_proto\_\_: theProtoObj,

// Shorthand for ‘handler: handler’

handler,

// Methods

toString() {

// Super calls

return "d " + super.toString();

},

// Computed (dynamic) property names

[ 'prop\_' + (() => 42)() ]: 42

};

# 模板字符串

模板字面量 是允许嵌入表达式的字符串字面量。你可以使用多行字符串和字符串插值功能。它们在 ES2015 规范的先前版本中被称为“模板字符串”。

## 语法

`string text`

`string text line 1

string text line 2`

`string text ${expression} string text`

tag `string text ${expression} string text`

## 描述

模板字符串使用反引号 (` `) 来代替普通字符串中的用双引号和单引号。模板字符串可以包含特定语法（${expression}）的占位符。占位符中的表达式和周围的文本会一起传递给一个默认函数，该函数负责将所有的部分连接起来，如果一个模板字符串由表达式开头，则该字符串被称为带标签的模板字符串，该表达式通常是一个函数，它会在模板字符串处理后被调用，在输出最终结果前，你都可以通过该函数来对模板字符串进行操作处理。在模版字符串内使用反引号（`）时，需要在它前面加转义符（\）。

`\`` === "`" // --> true

### 多行字符串

在新行中插入的任何字符都是模板字符串中的一部分，使用普通字符串，你可以通过以下的方式获得多行字符串：

console.log('string text line 1\n' + 'string text line 2');

// "string text line 1

// string text line 2"

要获得同样效果的多行字符串，只需使用如下代码：

console.log(`string text line 1

string text line 2`);

// "string text line 1

// string text line 2"

### 插入表达式

在普通字符串中嵌入表达式，必须使用如下语法：

var a = 5;

var b = 10;

console.log('Fifteen is ' + (a + b) + ' and\nnot ' + (2 \* a + b) + '.');

// "Fifteen is 15 and

// not 20."

Copy to Clipboard

现在通过模板字符串，我们可以使用一种更优雅的方式来表示：

var a = 5;

var b = 10;

console.log(`Fifteen is ${a + b} and

not ${2 \* a + b}.`);

// "Fifteen is 15 and

// not 20."

### 嵌套模板

在某些时候，嵌套模板是具有可配置字符串的最简单也是更可读的方法。 在模板中，只需在模板内的占位符 ${ } 内使用它们，就可以轻松地使用内部反引号。 例如，如果条件 a 是真的，那么返回这个模板化的文字。

ES5:

var classes = 'header'

classes += (isLargeScreen() ?

'' : item.isCollapsed ?

' icon-expander' : ' icon-collapser');

Copy to Clipboard

在 ES2015 中使用模板文字而没有嵌套：

const classes = `header ${ isLargeScreen() ? '' :

(item.isCollapsed ? 'icon-expander' : 'icon-collapser') }`;

Copy to Clipboard

在 ES2015 的嵌套模板字面量中：

const classes = `header ${ isLargeScreen() ? '' :

`icon-${item.isCollapsed ? 'expander' : 'collapser'}` }`;

### 带标签的模板字符串

更高级的形式的模板字符串是带标签的模板字符串。标签使您可以用函数解析模板字符串。标签函数的第一个参数包含一个字符串值的数组。其余的参数与表达式相关。最后，你的函数可以返回处理好的的字符串（或者它可以返回完全不同的东西 , 如下个例子所述）。用于该标签的函数的名称可以被命名为任何名字。

let person = 'Mike';

let age = 28;

function myTag(strings, personExp, ageExp) {

let str0 = strings[0]; // "That "

let str1 = strings[1]; // " is a "

let str2 = strings[2]; // "."

let ageStr;

if (ageExp > 99){

ageStr = 'centenarian';

} else {

ageStr = 'youngster';

}

// We can even return a string built using a template literal

return `${str0}${personExp}${str1}${ageStr}${str2}`;

}

let output = myTag`That ${ person } is a ${ age }.`;

console.log(output);

// That Mike is a youngster.

正如下面例子所展示的，标签函数并不一定需要返回一个字符串。

function template(strings, ...keys) {

return (function(...values) {

var dict = values[values.length - 1] || {};

var result = [strings[0]];

keys.forEach(function(key, i) {

var value = Number.isInteger(key) ? values[key] : dict[key];

result.push(value, strings[i + 1]);

});

return result.join('');

});

}

var t1Closure = template`${0}${1}${0}!`;

t1Closure('Y', 'A'); // "YAY!"

var t2Closure = template`${0} ${'foo'}!`;

t2Closure('Hello', {foo: 'World'}); // "Hello World!"

### 原始字符串

在标签函数的第一个参数中，存在一个特殊的属性raw ，我们可以通过它来访问模板字符串的原始字符串，而不经过特殊字符的替换。

function tag(strings) {

console.log(strings.raw[0]);

}

tag`string text line 1 \n string text line 2`;

// logs "string text line 1 \n string text line 2" ,

// including the two characters '\' and 'n'

使用String.raw() 方法创建原始字符串和使用默认模板函数和字符串连接创建是一样的。

var str = String.raw`Hi\n${2+3}!`;var`Hi\n${2+3}

// "Hi\\n5!"

str.length;.

// 6

str.split('').join(',');.

// "H,i,\\,n,5,!"

### 带标签的模版字面量及转义序列

自 ES2016 起，带标签的模版字面量遵守以下转义序列的规则：

* Unicode 字符以"\u"开头，例如\u00A9
* Unicode 码位用"\u{}"表示，例如\u{2F804}
* 十六进制以"\x"开头，例如\xA9
* 八进制以"\"和数字开头，例如\251

这表示类似下面这种带标签的模版是有问题的，因为对于每一个 ECMAScript 语法，解析器都会去查找有效的转义序列，但是只能得到这是一个形式错误的语法：

latex`\unicode`

// 在较老的 ECMAScript 版本中报错（ES2016 及更早）

// SyntaxError: malformed Unicode character escape sequence

# 解构赋值

解构赋值语法是一种 Javascript 表达式。通过解构赋值，可以将属性/值从对象/数组中取出，赋值给其他变量。

let a, b, rest;

[a, b] = [10, 20];

console.log(a);

// expected output: 10

console.log(b);

// expected output: 20

[a, b, ...rest] = [10, 20, 30, 40, 50];

console.log(rest);

// expected output: Array [30,40,50]

## 语法

var a, b, rest;

[a, b] = [10, 20];

console.log(a); // 10

console.log(b); // 20

[a, b, ...rest] = [10, 20, 30, 40, 50];

console.log(a); // 10

console.log(b); // 20

console.log(rest); // [30, 40, 50]

({ a, b } = { a: 10, b: 20 });

console.log(a); // 10

console.log(b); // 20

// Stage 4（已完成）提案中的特性

({a, b, ...rest} = {a: 10, b: 20, c: 30, d: 40});

console.log(a); // 10

console.log(b); // 20

console.log(rest); // {c: 30, d: 40}

## 描述

对象和数组逐个对应表达式，或称对象字面量和数组字面量，提供了一种简单的定义一个特定的数据组的方法。

var x = [1, 2, 3, 4, 5];

Copy to Clipboard

解构赋值使用了相同的语法，不同的是在表达式左边定义了要从原变量中取出什么变量。

var x = [1, 2, 3, 4, 5];

var [y, z] = x;

console.log(y); // 1

console.log(z); // 2

JavaScript 中，解构赋值的作用类似于 Perl 和 Python 语言中的相似特性。

## 解构数组

### 变量声明并赋值时的解构

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

var [one, two, three] = foo;

console.log(one); // "one"

console.log(two); // "two"

console.log(three); // "three"

### 变量先声明后赋值时的解构

通过解构分离变量的声明，可以为一个变量赋值。

var a, b;

[a, b] = [1, 2];

console.log(a); // 1

console.log(b); // 2

### 默认值

为了防止从数组中取出一个值为undefined的对象，可以在表达式左边的数组中为任意对象预设默认值。

var a, b;

[a=5, b=7] = [1];

console.log(a); // 1

console.log(b); // 7

### 交换变量

在一个解构表达式中可以交换两个变量的值。没有解构赋值的情况下，交换两个变量需要一个临时变量。

var a = 1;

var b = 3;

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

console.log(a); // 3

console.log(b); // 1

### 函数返回的数组

从一个函数返回一个数组是十分常见的情况。解构使得处理返回值为数组时更加方便。在下面例子中，要让 [1, 2] 成为函数的 f() 的输出值，可以使用解构在一行内完成解析。

function f() {

return [1, 2];

}

var a, b;

[a, b] = f();

console.log(a); // 1

console.log(b); // 2

### 忽略某些返回值

你也可以忽略你不感兴趣的返回值：

function f() {

return [1, 2, 3];

}

var [a, , b] = f();

console.log(a); // 1

console.log(b); // 3

你也可以忽略全部返回值：

[,,] = f();

### 将剩余数组赋值给一个变量

当解构一个数组时，可以使用剩余模式，将数组剩余部分赋值给一个变量。

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

console.log(a); // 1

console.log(b); // [2, 3]

Copy to Clipboard

注意：如果剩余元素右侧有逗号，会抛出 SyntaxError，因为剩余元素必须是数组的最后一个元素。

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

// SyntaxError: rest element may not have a trailing comma

### 用正则表达式匹配提取值

用正则表达式的 exec() 方法匹配字符串会返回一个数组，该数组第一个值是完全匹配正则表达式的字符串，然后的值是匹配正则表达式括号内内容部分。解构赋值允许你轻易地提取出需要的部分，忽略完全匹配的字符串——如果不需要的话。

function parseProtocol(url) {

var parsedURL = /^(\w+)\:\/\/([^\/]+)\/(.\*)$/.exec(url);

if (!parsedURL) {

return false;

}

console.log(parsedURL); // ["https://developer.mozilla.org/en-US/Web/JavaScript", "https", "developer.mozilla.org", "en-US/Web/JavaScript"]

var [, protocol, fullhost, fullpath] = parsedURL;

return protocol;

}

console.log(parseProtocol('https://developer.mozilla.org/en-US/Web/JavaScript')); // "https"

## 解构对象

### 基本赋值

var o = {p: 42, q: true};

var {p, q} = o;

console.log(p); // 42

console.log(q); // true

### 无声明赋值

一个变量可以独立于其声明进行解构赋值。

var a, b;

({a, b} = {a: 1, b: 2});

赋值语句周围的圆括号 ( ... ) 在使用对象字面量无声明解构赋值时是必须的。{a, b} = {a: 1, b: 2} 不是有效的独立语法，因为左边的 {a, b} 被认为是一个块而不是对象字面量。然而，({a, b} = {a: 1, b: 2}) 是有效的，正如 var {a, b} = {a: 1, b: 2}。你的 ( ... ) 表达式之前需要有一个分号，否则它可能会被当成上一行中的函数执行。

### 给新的变量名赋值

可以从一个对象中提取变量并赋值给和对象属性名不同的新的变量名。

var o = {p: 42, q: true};

var {p: foo, q: bar} = o;

console.log(foo); // 42

console.log(bar); // true

### 默认值

变量可以先赋予默认值。当要提取的对象对应属性解析为 undefined，变量就被赋予默认值。

var {a = 10, b = 5} = {a: 3};

console.log(a); // 3

console.log(b); // 5

### 给新的变量命名并提供默认值

一个属性可以同时：

* 从一个对象解构，并分配给一个不同名称的变量；
* 分配一个默认值，以防未解构的值是 undefined。

var {a:aa = 10, b:bb = 5} = {a: 3};

console.log(aa); // 3

console.log(bb); // 5

### 函数参数默认值

ES5 版本：

function drawES5Chart(options) {

options = options === undefined ? {} : options;

var size = options.size === undefined ? 'big' : options.size;

var cords = options.cords === undefined ? { x: 0, y: 0 } : options.cords;

var radius = options.radius === undefined ? 25 : options.radius;

console.log(size, cords, radius);

// now finally do some chart drawing

}

drawES5Chart({

cords: { x: 18, y: 30 },

radius: 30

});

Copy to Clipboard

ES2015 版本：

function drawES2015Chart({size = 'big', cords = { x: 0, y: 0 }, radius = 25} = {})

{

console.log(size, cords, radius);

// do some chart drawing

}

drawES2015Chart({

cords: { x: 18, y: 30 },

radius: 30

});

在上面的 drawES2015Chart 的函数签名中，解构的左手边被分配给右手边的空对象字面值：{size = 'big', cords = {x: 0, y: 0}, radius = 25} = {}。你也可以在没有右侧分配的情况下编写函数。但是，如果你忽略了右边的赋值，那么函数会在被调用的时候查找至少一个被提供的参数，而在当前的形式下，你可以直接调用 drawES2015Chart() 而不提供任何参数。

### 解构嵌套对象和数组

const metadata = {

title: 'Scratchpad',

translations: [

{

locale: 'de',

localization\_tags: [],

last\_edit: '2014-04-14T08:43:37',

url: '/de/docs/Tools/Scratchpad',

title: 'JavaScript-Umgebung'

}

],

url: '/en-US/docs/Tools/Scratchpad'

};

let {

title: englishTitle, // rename

translations: [

{

title: localeTitle, // rename

},

],

} = metadata;

console.log(englishTitle); // "Scratchpad"

console.log(localeTitle); // "JavaScript-Umgebung"

### For of 迭代和解构

var people = [

{

name: 'Mike Smith',

family: {

mother: 'Jane Smith',

father: 'Harry Smith',

sister: 'Samantha Smith'

},

age: 35

},

{

name: 'Tom Jones',

family: {

mother: 'Norah Jones',

father: 'Richard Jones',

brother: 'Howard Jones'

},

age: 25

}

];

for (var {name: n, family: {father: f}} of people) {

console.log('Name: ' + n + ', Father: ' + f);

}

// "Name: Mike Smith, Father: Harry Smith"

// "Name: Tom Jones, Father: Richard Jones"

### 从函数实参的对象中提取数据

function userId({id}) {

return id;

}

function whois({displayName: displayName, fullName: {firstName: name}}){

console.log(displayName + " is " + name);

}

var user = {

id: 42,

displayName: "jdoe",

fullName: {

firstName: "John",

lastName: "Doe"

}

};

console.log("userId: " + userId(user)); // "userId: 42"

whois(user); // "jdoe is John"

这段代码从 user 对象中提取并输出id、displayName 和 firstName。

### 对象属性计算名和解构

计算属性名，如 object literals，可以被解构。

let key = "z";

let { [key]: foo } = { z: "bar" };

console.log(foo); // "bar"

### 解构对象时会查找原型链

解构对象时会查找原型链（如果属性不在对象自身，将从原型链中查找）

// 声明对象 和 自身 self 属性

var obj = {self: '123'};

// 在原型链中定义一个属性 prot

obj.\_\_proto\_\_.prot = '456';

// test

const {self, prot} = obj;

// self "123"

// prot "456"（访问到了原型链）

# 默认参数值

函数默认参数允许在没有值或undefined被传入时使用默认形参。

function multiply(a, b = 1) {

return a \* b;

}

console.log(multiply(5, 2));

// expected output: 10

console.log(multiply(5));

// expected output: 5

## 语法

function [name]([param1[ = defaultValue1 ][, ..., paramN[ = defaultValueN ]]]) {

statements

}

## 描述

JavaScript 中函数的参数默认是undefined。然而，在某些情况下可能需要设置一个不同的默认值。这是默认参数可以帮助的地方。以前，一般设置默认参数的方法是在函数体测试参数是否为undefined，如果是的话就设置为默认的值。下面的例子中，如果在调用multiply时，参数b的值没有提供，那么它的值就为undefined。如果直接执行a \* b，函数会返回 NaN。

function multiply(a, b) {

return a \* b;

}

multiply(5, 2); // 10

multiply(5); // NaN !

为了防止这种情况，第二行代码解决了这个问题，其中如果只使用一个参数调用multiply，则b设置为1：

function multiply(a, b) {

b = (typeof b !== 'undefined') ? b : 1;

return a \* b;

}

multiply(5, 2); // 10

multiply(5); // 5

Copy to Clipboard

有了默认参数，我们不需要再在函数体内做不必要的检查。现在你可以在函数头将b的默认值置为1：

function multiply(a, b = 1) {

return a \* b;

}

multiply(5, 2); // 10

multiply(5); // 5

## 示例

### 传入 undefined vs 其他假值

在第二次调用中，即使第一个参数在调用时显式设置为undefined（虽然不是null或其他falsy值），但是num参数的值是默认值。

function test(num = 1) {

console.log(typeof num);

}

test(); // 'number' (num is set to 1)

test(undefined); // 'number' (num is set to 1 too)

// test with other falsy values:

test(''); // 'string' (num is set to '')

test(null); // 'object' (num is set to null)

### 调用时解析

在函数被调用时，参数默认值会被解析，所以不像 Python 中的例子，每次函数调用时都会创建一个新的参数对象。

function append(value, array = []) {

array.push(value);

return array;

}

append(1); //[1]

append(2); //[2], not [1, 2]

这个规则对于函数和变量也是适用的。

function callSomething(thing = something()) {

return thing;

}

let numberOfTimesCalled = 0;

function something() {

numberOfTimesCalled += 1;

return numberOfTimesCalled;

}

callSomething(); // 1

callSomething(); // 2

### 默认参数可用于后面的默认参数

已经遇到的参数可用于以后的默认参数：

function greet(name, greeting, message = greeting + ' ' + name) {

return [name, greeting, message];

}

greet('David', 'Hi'); // ["David", "Hi", "Hi David"]

greet('David', 'Hi', 'Happy Birthday!'); // ["David", "Hi", "Happy Birthday!"]

### 函数嵌套定义

在函数体内的函数声明不能引用内部的默认参数，并且会在 SpiderMonkey 抛出一个ReferenceError（现在是 TypeError）。默认参数总是会被首先执行，而在函数体内部的函数声明会在之后生效。

// Doesn't work! Throws ReferenceError.

function f(a = go()) {

function go() { return ':P'; }

}

### 默认参数之后非默认参数

以下代码会造成SyntaxError错误。这已经修复，并在以后的版本中按预期方式工作。参数仍然设置为从左到右，覆盖默认参数，即使后面的参数没有默认值。

function f(x = 1, y) {

return [x, y];

}

f(); // [1, undefined]

f(2); // [2, undefined]

### 有默认值的解构参数

通过解构赋值为参数赋值：

function f([x, y] = [1, 2], {z: z} = {z: 3}) {

return x + y + z;

}

f(); // 6

# 剩余参数

剩余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。

function sum(...theArgs) {

return theArgs.reduce((previous, current) => {

return previous + current;

});

}

console.log(sum(1, 2, 3));

// expected output: 6

console.log(sum(1, 2, 3, 4));

// expected output: 10

## 语法

function(a, b, ...theArgs) {

// ...

}

## 描述

如果函数的最后一个命名参数以...为前缀，则它将成为一个由剩余参数组成的真数组，其中从0（包括）到theArgs.length（排除）的元素由传递给函数的实际参数提供。在上面的例子中，theArgs将收集该函数的第三个参数（因为第一个参数被映射到a，而第二个参数映射到b）和所有后续参数。

剩余参数和 arguments对象的区别，剩余参数和 arguments对象之间的区别主要有三个：

* 剩余参数只包含那些没有对应形参的实参，而 arguments 对象包含了传给函数的所有实参。
* arguments对象不是一个真正的数组，而剩余参数是真正的 Array实例，也就是说你能够在它上面直接使用所有的数组方法，比如 sort，map，forEach或pop。
* arguments对象还有一些附加的属性 （如callee属性）。

引入了剩余参数来减少由参数引起的样板代码。

// Before rest parameters, "arguments" could be converted to a normal array using:

function f(a, b) {

var normalArray = Array.prototype.slice.call(arguments);

// -- or --

var normalArray = [].slice.call(arguments);

// -- or --

var normalArray = Array.from(arguments);

var first = normalArray.shift(); // OK, gives the first argument

var first = arguments.shift(); // ERROR (arguments is not a normal array)

}

// Now we can easily gain access to a normal array using a rest parameter

function f(...args) {

var normalArray = args;

var first = normalArray.shift(); // OK, gives the first argument

}

解构剩余参数，剩余参数可以被解构，这意味着他们的数据可以被解包到不同的变量中。

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

return a + b + c;

}

f(1) // NaN (b and c are undefined)

f(1, 2, 3) // 6

f(1, 2, 3, 4) // 6 (the fourth parameter is not destructured)

## 示例

因为theArgs是个数组，所以你可以使用length属性得到剩余参数的个数：

function fun1(...theArgs) {

alert(theArgs.length);

}

fun1(); // 弹出 "0", 因为 theArgs 没有元素

fun1(5); // 弹出 "1", 因为 theArgs 只有一个元素

fun1(5, 6, 7); // 弹出 "3", 因为 theArgs 有三个元素

下例中，剩余参数包含了从第二个到最后的所有实参，然后用第一个实参依次乘以它们：

function multiply(multiplier, ...theArgs) {

return theArgs.map(function (element) {

return multiplier \* element;

});

}

var arr = multiply(2, 1, 2, 3);

console.log(arr); // [2, 4, 6]

下例中，剩余参数包含了从第二个到最后的所有实参，然后用第一个实参依次乘以它们：

function multiply(multiplier, ...theArgs) {

return theArgs.map(function (element) {

return multiplier \* element;

});

}

var arr = multiply(2, 1, 2, 3);

console.log(arr); // [2, 4, 6]

下例演示了你可以在剩余参数上使用任意的数组方法，而arguments对象不可以：

function sortRestArgs(...theArgs) {

var sortedArgs = theArgs.sort();

return sortedArgs;

}

alert(sortRestArgs(5,3,7,1)); // 弹出 1,3,5,7

function sortArguments() {

var sortedArgs = arguments.sort();

return sortedArgs; // 不会执行到这里

}

alert(sortArguments(5,3,7,1)); // 抛出 TypeError 异常:arguments.sort is not a function

为了在arguments对象上使用Array方法，它必须首先被转换为一个真正的数组。

function sortArguments() {

var args = Array.prototype.slice.call(arguments);

var sortedArgs = args.sort();

return sortedArgs;

}

console.log(sortArguments(5, 3, 7, 1)); // shows 1, 3, 5, 7

# 展开语法

展开语法 (Spread syntax), 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；还可以在构造字面量对象时，将对象表达式按 key-value 的方式展开。(字面量一般指 [1, 2, 3] 或者 {name: "mdn"} 这种简洁的构造方式)

function sum(x, y, z) {

return x + y + z;

}

const numbers = [1, 2, 3];

console.log(sum(...numbers));

// expected output: 6

console.log(sum.apply(null, numbers));

// expected output: 6

## 语法

函数调用：

myFunction(...iterableObj);

字面量数组构造或字符串：

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

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

let objClone = { ...obj };

## 示例

### 在函数调用时使用展开语法

等价于apply的方式，如果想将数组元素迭代为函数参数，一般使用Function.prototype.apply 的方式进行调用。

function myFunction(x, y, z) { }

var args = [0, 1, 2];

myFunction.apply(null, args);

有了展开语法，可以这样写：

function myFunction(x, y, z) { }

var args = [0, 1, 2];

myFunction(...args);

### 在new表达式中应用

使用 new 关键字来调用构造函数时，不能直接使用数组 + apply 的方式（apply 执行的是调用 [[Call]] , 而不是构造 [[Construct]]）。当然，有了展开语法，将数组展开为构造函数的参数就很简单了：

var dateFields = [1970, 0, 1]; // 1970 年 1 月 1 日

var d = new Date(...dateFields);

如果不使用展开语法，想将数组元素传给构造函数，实现方式可能是这样的：

function applyAndNew(constructor, args) {

function partial () {

return constructor.apply(this, args);

};

if (typeof constructor.prototype === "object") {

partial.prototype = Object.create(constructor.prototype);

}

return partial;

}

function myConstructor () {

console.log("arguments.length: " + arguments.length);

console.log(arguments);

this.prop1="val1";

this.prop2="val2";

};

var myArguments = ["hi", "how", "are", "you", "mr", null];

var myConstructorWithArguments = applyAndNew(myConstructor, myArguments);

console.log(new myConstructorWithArguments);

// (myConstructor 构造函数中): arguments.length: 6

// (myConstructor 构造函数中): ["hi", "how", "are", "you", "mr", null]

// ("new myConstructorWithArguments"中): {prop1: "val1", prop2: "val2"}

### 构造字面量数

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，通过字面量方式，构造新数组会变得更简单、更优雅：

var parts = ['shoulders', 'knees'];

var lyrics = ['head', ...parts, 'and', 'toes'];

// ["head", "shoulders", "knees", "and", "toes"]

Copy to Clipboard

和参数列表的展开类似， ... 在构造字面量数组时，可以在任意位置多次使用。

### 数组拷贝 (copy)

var arr = [1, 2, 3];

var arr2 = [...arr]; // like arr.slice()

arr2.push(4);

// arr2 此时变成 [1, 2, 3, 4]

// arr 不受影响

实际上，展开语法和 Object.assign() 行为一致，执行的都是浅拷贝 (只遍历一层)。如果想对多维数组进行深拷贝，下面的示例就有些问题了。

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

var b = [...a];

b.shift().shift(); // 1

// Now array a is affected as well: [[2], [3]]

### 连接多个数组

Array.concat 函数常用于将一个数组连接到另一个数组的后面。如果不使用展开语法，代码可能是下面这样的：

var arr1 = [0, 1, 2];

var arr2 = [3, 4, 5];

// 将 arr2 中所有元素附加到 arr1 后面并返回

var arr3 = arr1.concat(arr2);

使用展开语法：

var arr1 = [0, 1, 2];

var arr2 = [3, 4, 5];

var arr3 = [...arr1, ...arr2];

Array.unshift 方法常用于在数组的开头插入新元素/数组. 不使用展开语法，示例如下：

var arr1 = [0, 1, 2];

var arr2 = [3, 4, 5];

// 将 arr2 中的元素插入到 arr1 的开头

Array.prototype.unshift.apply(arr1, arr2) // arr1 现在是 [3, 4, 5, 0, 1, 2]

如果使用展开语法，代码如下: [请注意，这里使用展开语法创建了一个新的 arr1 数组， Array.unshift 方法则是修改了原本存在的 arr1 数组]:

var arr1 = [0, 1, 2];

var arr2 = [3, 4, 5];

arr1 = [...arr2, ...arr1]; // arr1 现在为 [3, 4, 5, 0, 1, 2]

### 构造字面量对象时使用展开语法

对字面量对象增加了展开特性。其行为是，将已有对象的所有可枚举 (enumerable) 属性拷贝到新构造的对象中。浅拷贝 (Shallow-cloning，不包含 prototype) 和对象合并，可以使用更简短的展开语法。而不必再使用 Object.assign() 方式。

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 }

Object.assign() 函数会触发 setters，而展开语法则不会。

不能替换或者模拟 Object.assign() 函数：

var obj1 = { foo: 'bar', x: 42 };

var obj2 = { foo: 'baz', y: 13 };

const merge = ( ...objects ) => ( { ...objects } );

var mergedObj = merge ( obj1, obj2);

// Object { 0: { foo: 'bar', x: 42 }, 1: { foo: 'baz', y: 13 } }

var mergedObj = merge ( {}, obj1, obj2);

// Object { 0: {}, 1: { foo: 'bar', x: 42 }, 2: { foo: 'baz', y: 13 } }

### 只能用于可迭代对象

在数组或函数参数中使用展开语法时，该语法只能用于 可迭代对象：

var obj = {'key1': 'value1'};

var array = [...obj]; // TypeError: obj is not iterable

在函数调用时使用展开语法，请注意不能超过 JavaScript 引擎限制的最大参数个数。

## 剩余语法（剩余参数）

剩余语法 (Rest syntax) 看起来和展开语法完全相同，不同点在于，剩余参数用于解构数组和对象。从某种意义上说，剩余语法与展开语法是相反的：展开语法将数组展开为其中的各个元素，而剩余语法则是将多个元素收集起来并“凝聚”为单个元素。剩余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。

function sum(...theArgs) {

let total = 0;

for (const arg of theArgs) {

total += arg;

}

return total;

}

console.log(sum(1, 2, 3));

// expected output: 6

console.log(sum(1, 2, 3, 4));

// expected output: 10

如果函数的最后一个命名参数以...为前缀，则它将成为一个由剩余参数组成的真数组，其中从0（包括）到theArgs.length（排除）的元素由传递给函数的实际参数提供。在上面的例子中，theArgs将收集该函数的第三个参数（因为第一个参数被映射到a，而第二个参数映射到b）和所有后续参数。剩余参数和 arguments对象之间的区别主要有三个：

* 剩余参数只包含那些没有对应形参的实参，而 arguments 对象包含了传给函数的所有实参。
* arguments对象不是一个真正的数组，而剩余参数是真正的 Array实例，也就是说你能够在它上面直接使用所有的数组方法，比如 sort，map，forEach或pop。
* arguments对象还有一些附加的属性 （如callee属性）。

# let 语句

let 语句声明一个块级作用域的局部变量，并可以初始化为一个值（可选）。

let x = 1;

if (x === 1) {

let x = 2;

console.log(x);

// expected output: 2

}

console.log(x);

// expected output: 1

## 语法

let name1 [= value1] [, name2 [= value2]] [, ..., nameN [= valueN];

* nameN：变量名，必须是合法的标识符。
* valueN：变量的初始值，可以是任意合法的表达式。
* 解构赋值语法来声明变量。

## 描述

let 允许你声明一个作用域被限制在块作用域中的变量、语句或者表达式。与 var 关键字不同的是，var 声明的变量作用域是全局或者整个函数块的。 var 和 let 的另一个重要区别，let 声明的变量不会在作用域中被提升，它是在编译时才初始化。

## 示例

### 作用域规则

let 声明的变量作用域只在其声明的块或子块内部，这一点，与 var 相似。二者之间最主要的区别在于 var 声明的变量的作用域是整个封闭函数。

function varTest() {

var x = 1;

{

var x = 2; // same variable!

console.log(x); // 2

}

console.log(x); // 2

}

function letTest() {

let x = 1;

{

let x = 2; // different variable

console.log(x); // 2

}

console.log(x); // 1

}

在全局作用域中，let 和 var 不一样，它不会在顶层对象上创建属性。例如：

var x = 'global';

let y = 'global';

console.log(this.x); // "global"

console.log(this.y); // undefined

### 模仿私有成员

在处理构造函数的时候，可以通过let声明而不是闭包来创建一个或多个私有成员。

var Thing;

{

let privateScope = new WeakMap();

let counter = 0;

Thing = function() {

this.someProperty = 'foo';

privateScope.set(this, {

hidden: ++counter,

});

};

Thing.prototype.showPublic = function() {

return this.someProperty;

};

Thing.prototype.showPrivate = function() {

return privateScope.get(this).hidden;

};

}

console.log(typeof privateScope); // "undefined"

var thing = new Thing();

console.log(thing); // Thing {someProperty: "foo"}

thing.showPublic(); // "foo"

thing.showPrivate(); // 1

### 重复声明

在同一个函数或块作用域中重复声明同一个变量会抛出 SyntaxError。

if (x) {

let foo;

let foo; // SyntaxError thrown.

}

在 switch 语句中也会触发这个错误，因为它是同一个块作用域。

let x = 1;

switch(x) {

case 0:

let foo;

break;

case 1:

let foo; // SyntaxError for redeclaration.

break;

}

然而，需要特别指出的是，一个嵌套在 case 子句中的块会创建一个新的块作用域的词法环境，就不会产生上诉重复声明的错误。

let x = 1;

switch(x) {

case 0: {

let foo;

break;

}

case 1: {

let foo;

break;

}

}

### 暂时性死区

let 变量在声明之前，不能够读写。如果声明中未指定初始值，则变量将使用 undefined 值初始化，在声明之前访问变量会导致 ReferenceError。与 var 变量不同，如果在声明前使用 var，变量将会被初始化为undefined。从块作用域的顶部一直到变量声明完成之前，这个变量处在暂时性死区（TDZ，temporal dead zone）。

{ // TDZ starts at beginning of scope

console.log(bar); // undefined

console.log(foo); // ReferenceError

var bar = 1;

let foo = 2; // End of TDZ (for foo)

}

使用术语 “temporal” 是因为区域取决于执行顺序（时间），而不是编写代码的顺序（位置）。例如，下面的代码会生效，是因为即使使用 let 变量的函数出现在变量声明之前，但函数的执行是在 TDZ 的外面。

{

// TDZ starts at beginning of scope

const func = () => console.log(letVar); // OK

// Within the TDZ letVar access throws `ReferenceError`

let letVar = 3; // End of TDZ (for letVar)

func(); // Called outside TDZ!

}

### 暂时性死区与 typeof

如果使用 typeof 检测在暂时性死区中的变量，会抛出 ReferenceError 异常：

// results in a 'ReferenceError'

console.log(typeof i);

let i = 10;

这与使用 typeof 检测值为 undefined 的未声明变量不同：

// prints out 'undefined'

console.log(typeof undeclaredVariable);

### 暂时性死区和词法作用域

以下代码会导致 ReferenceError：

function test() {

var foo = 33;

if(foo) {

let foo = (foo + 55); // ReferenceError

}

}

test();

由于外部变量 foo 有值，因此会执行 if 语句块，但是由于词法作用域，该值在块内不可用：if 快内的标识符 foo 是 let foo。表达式 (foo + 55) 会抛出异常，是因为 let foo 还没完成初始化，它仍然在暂时性死区里。

在以下情况下，这种现象可能会使您感到困惑。let n of n.a 已经在 for 循环块的私有范围内，因此，标识符 n.a 被解析为位于指令本身（let n）中的“ n ”对象的属性“ a ”。在没有执行到它的初始化语句之前，它仍旧存在于暂时性死区中。

function go(n) {

// n here is defined!

console.log(n); // Object {a: [1,2,3]}

for (let n of n.a) { // ReferenceError

console.log(n);

}

}

go({a: [1, 2, 3]});

### 其他情况

用在块级作用域中， let 将变量的作用域限制在块内， 而 var 声明的变量的作用域是在函数内。

var a = 1;

var b = 2;

if (a === 1) {

var a = 11; // the scope is global

let b = 22; // the scope is inside the if-block

console.log(a); // 11

console.log(b); // 22

}

console.log(a); // 11

console.log(b); // 2

然而，var 与 let 合并的声明方式会抛出 SyntaxError 错误，因为 var 会将变量提升至块的顶部，这会导致隐式地重复声明变量。

let x = 1;

{

var x = 2; // SyntaxError for re-declaration

}

# const语句

常量是块级范围的，非常类似用 let 语句定义的变量。但常量的值是无法（通过重新赋值）改变的，也不能被重新声明。

const number = 42;

try {

number = 99;

} catch (err) {

console.log(err);

// expected output: TypeError: invalid assignment to const `number'

// Note - error messages will vary depending on browser

}

console.log(number);

// expected output: 42

## 语法

const name1 = value1 [, name2 = value2 [, ... [, nameN = valueN]]];

* nameN：常量名称，可以是任意合法的标识符。
* valueN：常量值，可以是任意合法的表达式。

## 描述

此声明创建一个常量，其作用域可以是全局或本地声明的块。 与var变量不同，全局常量不会变为 window 对象的属性。需要一个常数的初始化器；也就是说，您必须在声明的同一语句中指定它的值（这是有道理的，因为以后不能更改）。

const声明创建一个值的只读引用。但这并不意味着它所持有的值是不可变的，只是变量标识符不能重新分配。例如，在引用内容是对象的情况下，这意味着可以改变对象的内容（例如，其参数）。关于“暂存死区”的所有讨论都适用于let和const。

一个常量不能和它所在作用域内的其他变量或函数拥有相同的名称。

## 示例

### const 基本用法

常量在声明的时候可以使用大小写，但通常情况下全部用大写字母。

// 定义常量 MY\_FAV 并赋值 7

const MY\_FAV = 7;

// 报错 - Uncaught TypeError: Assignment to constant variable.

MY\_FAV = 20;

// MY\_FAV is 7

console.log('my favorite number is: ' + MY\_FAV);

// 尝试重新声明会报错

// Uncaught SyntaxError: Identifier 'MY\_FAV' has already been declared

const MY\_FAV = 20;

// MY\_FAV 保留给上面的常量，这个操作会失败

var MY\_FAV = 20;

// 也会报错

let MY\_FAV = 20;

### 块作用域

if (MY\_FAV === 7) {

// 没问题，并且创建了一个块作用域变量 MY\_FAV

// (works equally well with let to declare a block scoped non const variable)

let MY\_FAV = 20;

// MY\_FAV 现在为 20

console.log('my favorite number is ' + MY\_FAV);

// 这被提升到全局上下文并引发错误

var MY\_FAV = 20;

}

// MY\_FAV 依旧为 7

console.log('my favorite number is ' + MY\_FAV);

### 常量要求一个初始值

常量可以定义成对象和数组：

const MY\_OBJECT = {'key': 'value'};

// 重写对象和上面一样会失败

// Uncaught TypeError: Assignment to constant variable.

MY\_OBJECT = {'OTHER\_KEY': 'value'};

// 对象属性并不在保护的范围内

// 下面这个声明会成功执行

MY\_OBJECT.key = 'otherValue'; // Use Object.freeze() to make object immutable

// 也可以用来定义数组

const MY\_ARRAY = [];

// 可以向数组填充数据

MY\_ARRAY.push('A'); // ["A"]

// 但是，将一个新数组赋给变量会引发错误

// Uncaught TypeError: Assignment to constant variable.

MY\_ARRAY = ['B'];

# for...of

for...of语句在可迭代对象（包括 Array，Map，Set，String，TypedArray，arguments 对象等等）上创建一个迭代循环，调用自定义迭代钩子，并为每个不同属性的值执行语句

const array1 = ['a', 'b', 'c'];

for (const element of array1) {

console.log(element);

}

// expected output: "a"

// expected output: "b"

// expected output: "c"

## 语法

for (variable of iterable) {

//statements

}

* variable：在每次迭代中，将不同属性的值分配给变量。
* iterable：被迭代枚举其属性的对象。

## 示例

### 迭代Array

let iterable = [10, 20, 30];

for (let value of iterable) {

value += 1;

console.log(value);

}

// 11

// 21

// 31

Copy to Clipboard

如果你不想修改语句块中的变量 , 也可以使用const代替let。

let iterable = [10, 20, 30];

for (const value of iterable) {

console.log(value);

}

// 10

// 20

// 30

### 迭代String

let iterable = "boo";

for (let value of iterable) {

console.log(value);

}

// "b"

// "o"

// "o"

### 迭代 TypedArray

let iterable = new Uint8Array([0x00, 0xff]);

for (let value of iterable) {

console.log(value);

}

// 0

// 255

### 迭代Map

let iterable = new Map([["a", 1], ["b", 2], ["c", 3]]);

for (let entry of iterable) {

console.log(entry);

}

// ["a", 1]

// ["b", 2]

// ["c", 3]

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

console.log(value);

}

// 1

// 2

// 3

### 迭代 Set

let iterable = new Set([1, 1, 2, 2, 3, 3]);

for (let value of iterable) {

console.log(value);

}

// 1

// 2

// 3

Copy to Clipboard

迭代 arguments 对象

(function() {

for (let argument of arguments) {

console.log(argument);

}

})(1, 2, 3);

// 1

// 2

// 3

### 迭代 DOM 集合

迭代 DOM 元素集合，比如一个NodeList对象：下面的例子演示给每一个 article 标签内的 p 标签添加一个 "read" 类。

//注意：这只能在实现了 NodeList.prototype[Symbol.iterator] 的平台上运行

let articleParagraphs = document.querySelectorAll("article > p");

for (let paragraph of articleParagraphs) {

paragraph.classList.add("read");

}

### 关闭迭代器

对于for...of的循环，可以由 break, throw 或 return 终止。在这些情况下，迭代器关闭。

function\* foo(){

yield 1;

yield 2;

yield 3;

};

for (let o of foo()) {

console.log(o);

break; // closes iterator, triggers return

}

### 迭代生成器

你还可以迭代一个生成器：

function\* fibonacci() { // 一个生成器函数

let [prev, curr] = [0, 1];

for (;;) { // while (true) {

[prev, curr] = [curr, prev + curr];

yield curr;

}

}

for (let n of fibonacci()) {

console.log(n);

// 当 n 大于 1000 时跳出循环

if (n >= 1000)

break;

}

### 不要重用生成器

生成器不应该重用，即使for...of循环的提前终止，例如通过break关键字。在退出循环后，生成器关闭，并尝试再次迭代，不会产生任何进一步的结果。

var gen = (function \*(){

yield 1;

yield 2;

yield 3;

})();

for (let o of gen) {

console.log(o);

break;//关闭生成器

}

//生成器不应该重用，以下没有意义！

for (let o of gen) {

console.log(o);

}

### 迭代其他可迭代对象

你还可以迭代显式实现可迭代协议的对象：

var iterable = {

[Symbol.iterator]() {

return {

i: 0,

next() {

if (this.i < 3) {

return { value: this.i++, done: false };

}

return { value: undefined, done: true };

}

};

}

};

for (var value of iterable) {

console.log(value);

}

// 0

// 1

// 2

### for...of与for...in的区别

无论是for...in还是for...of语句都是迭代一些东西。它们之间的主要区别在于它们的迭代方式。

* for...in 语句以任意顺序迭代对象的可枚举属性。
* for...of 语句遍历可迭代对象定义要迭代的数据。

以下示例显示了与Array一起使用时，for...of循环和for...in循环之间的区别。

Object.prototype.objCustom = function() {};

Array.prototype.arrCustom = function() {};

let iterable = [3, 5, 7];

iterable.foo = 'hello';

每个对象将继承objCustom属性，并且作为Array的每个对象将继承arrCustom属性，因为将这些属性添加到Object.prototype (en-US)和Array.prototype。由于继承和原型链，对象iterable继承属性objCustom和arrCustom。

for (let i in iterable) {

console.log(i); // logs 0, 1, 2, "foo", "arrCustom", "objCustom"

}

此循环仅以原始插入顺序记录iterable 对象的可枚举属性。它不记录数组元素3, 5, 7 或hello，因为这些不是枚举属性。但是它记录了数组索引以及arrCustom和objCustom。

for (let i in iterable) {

if (iterable.hasOwnProperty(i)) {

console.log(i); // logs 0, 1, 2, "foo"

}

}

这个循环类似于第一个，但是它使用hasOwnProperty() 来检查，如果找到的枚举属性是对象自己的（不是继承的）。如果是，该属性被记录。记录的属性是0, 1, 2和foo，因为它们是自身的属性（不是继承的）。属性arrCustom和objCustom不会被记录，因为它们是继承的。

for (let i of iterable) {

console.log(i); // logs 3, 5, 7

}

该循环迭代并记录iterable作为可迭代对象定义的迭代值，这些是数组元素 3, 5, 7，而不是任何对象的属性。

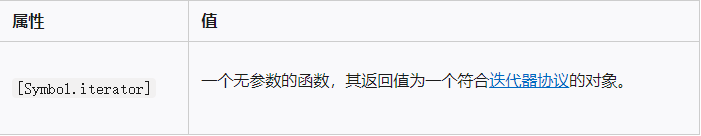
# 迭代协议

作为 ECMAScript 2015 的一组补充规范，迭代协议并不是新的内置实现或语法，而是协议。这些协议可以被任何遵循某些约定的对象来实现。迭代协议具体分为两个协议：可迭代协议和迭代器协议。

## 可迭代协议

**可迭代协议**允许 JavaScript 对象定义或定制它们的迭代行为，例如，在一个 for..of 结构中，哪些值可以被遍历到。一些内置类型同时是内置可迭代对象，并且有默认的迭代行为，比如 Array 或者 Map，而其他内置类型则不是（比如 Object)）。

要成为可迭代对象， 一个对象必须实现 @@iterator 方法。这意味着对象（或者它原型链上的某个对象）必须有一个键为 @@iterator 的属性，可通过常量 Symbol.iterator 访问该属性：



当一个对象需要被迭代的时候（比如被置入一个 for...of 循环时），首先，会不带参数调用它的 @@iterator 方法，然后使用此方法返回的迭代器获得要迭代的值。

值得注意的是调用此零个参数函数时，它将作为对可迭代对象的方法进行调用。 因此，在函数内部，this关键字可用于访问可迭代对象的属性，以决定在迭代过程中提供什么。此函数可以是普通函数，也可以是生成器函数，以便在调用时返回迭代器对象。 在此生成器函数的内部，可以使用yield提供每个条目。

## 迭代器协议

**迭代器协议**定义了产生一系列值（无论是有限个还是无限个）的标准方式。当值为有限个时，所有的值都被迭代完毕后，则会返回一个默认返回值。只有实现了一个拥有以下语义（semantic）的 next() 方法，一个对象才能成为迭代器：



## 使用迭代协议的例子

String 是一个内置的可迭代对象：

let someString = "hi";

typeof someString[Symbol.iterator]; // "function"

String 的默认迭代器会依次返回该字符串的各码点（code point）：

let iterator = someString[Symbol.iterator]();

iterator + ""; // "[object String Iterator]"

iterator.next(); // { value: "h", done: false }

iterator.next(); // { value: "i", done: false }

iterator.next(); // { value: undefined, done: true }

一些内置的语法结构——比如展开语法——其内部实现也使用了同样的迭代协议：

[...someString] // ["h", "i"]

我们可以通过提供自己的 @@iterator 方法，重新定义迭代行为：

// 必须构造 String 对象以避免字符串字面量 auto-boxing

var someString = new String("hi");

someString[Symbol.iterator] = function() {

return { // 只返回一次元素，字符串 "bye"，的迭代器对象

next: function() {

if (this.\_first) {

this.\_first = false;

return { value: "bye", done: false };

} else {

return { done: true };

}

},

\_first: true

};

};

注意重新定义的 @@iterator 方法是如何影响内置语法结构的行为的：

[...someString]; // ["bye"]

someString + ""; // "hi"

## 可迭代对象示例

### 内置可迭代对象

目前所有的内置可迭代对象如下：String、Array、TypedArray、Map 和 Set，它们的原型对象都实现了 @@iterator 方法。

### 自定义可迭代对象

我们可以实现一个自己的可迭代对象，就像这样：

var myIterable = {};

myIterable[Symbol.iterator] = function\* () {

yield 1;

yield 2;

yield 3;

};

[...myIterable]; // [1, 2, 3]

### 接受可迭代对象的内置 API

很多 API 接受可迭代对象作为参数，例如：

* new Map([iterable])
* new WeakMap([iterable])
* new Set([iterable])
* new WeakSet([iterable])
* Promise.all(iterable)
* Promise.race(iterable)
* Array.from(iterable)

new Map([[1, 'a'], [2, 'b'], [3, 'c']]).get(2); // "b"

let myObj = {};

new WeakMap([

[{}, 'a'],

[myObj, 'b'],

[{}, 'c']

]).get(myObj); // "b"

new Set([1, 2, 3]).has(3); // true

new Set('123').has('2'); // true

new WeakSet(function\* () {

yield {}

yield myObj

yield {}

}()).has(myObj); // true

### 需要可迭代对象的语法

一些语句和表达式需要可迭代对象，比如 for...of 循环、展开语法、yield\*，和解构赋值。

for(let value of ["a", "b", "c"]){

console.log(value);

}

// "a"

// "b"

// "c"

[..."abc"]; // ["a", "b", "c"]

function\* gen() {

yield\* ["a", "b", "c"];

}

gen().next(); // { value: "a", done: false }

[a, b, c] = new Set(["a", "b", "c"]);

a // "a"

### 格式不佳的可迭代对象

如果一个可迭代对象的 @@iterator 方法不能返回迭代器对象，那么可以认为它是一个格式不佳的（Non-well-formed）可迭代对象 。使用这样的可迭代对象很可能会导致如下的运行时（runtime）异常，或者不可预料的表现：

var nonWellFormedIterable = {}

nonWellFormedIterable[Symbol.iterator] = () => 1

[...nonWellFormedIterable] // TypeError: [] is not a function

## 迭代器示例

### 简单迭代器

function makeIterator(array) {

let nextIndex = 0;

return {

next: function () {

return nextIndex < array.length ? {

value: array[nextIndex++],

done: false

} : {

done: true

};

}

};

}

let it = makeIterator(['哟', '呀']);

console.log(it.next().value); // '哟'

console.log(it.next().value); // '呀'

console.log(it.next().done); // true

### 无穷迭代器

function idMaker() {

let index = 0;

return {

next: function() {

return {

value: index++,

done: false

};

}

};

}

let it = idMaker();

console.log(it.next().value); // 0

console.log(it.next().value); // 1

console.log(it.next().value); // 2

// ...

### 使用生成器

function\* makeSimpleGenerator(array) {

let nextIndex = 0;

while(nextIndex < array.length) {

yield array[nextIndex++];

}

}

let gen = makeSimpleGenerator(['哟', '呀']);

console.log(gen.next().value); // '哟'

console.log(gen.next().value); // '呀'

console.log(gen.next().done); // true

function\* idMaker() {

let index = 0;

while (true) {

yield index++;

}

}

let gen = idMaker();

console.log(gen.next().value); // 0

console.log(gen.next().value); // 1

console.log(gen.next().value); // 2

// ...

### ES2015 类 class 中的迭代器

class SimpleClass {

constructor(data) {

this.data = data

}

[Symbol.iterator]() {

// Use a new index for each iterator. This makes multiple

// iterations over the iterable safe for non-trivial cases,

// such as use of break or nested looping over the same iterable.

let index = 0;

return {

next: () => {

if (index < this.data.length) {

return {value: this.data[index++], done: false}

} else {

return {done: true}

}

}

}

}

}

const simple = new SimpleClass([1,2,3,4,5])

for (const val of simple) {

console.log(val) // 1 2 3 4 5

}

## 生成器对象

生成器对象既是迭代器，也是可迭代对象：

let aGeneratorObject = function\* (){

yield 1;

yield 2;

yield 3;

}();

typeof aGeneratorObject.next;

// 返回"function", 因为有一个 next 方法，所以这是一个迭代器

typeof aGeneratorObject[Symbol.iterator];

// 返回"function", 因为有一个 @@iterator 方法，所以这是一个可迭代对象

aGeneratorObject[Symbol.iterator]() === aGeneratorObject;

// 返回 true，因为 @@iterator 方法返回自身（即迭代器），所以这是一个格式良好的可迭代对象

[...aGeneratorObject];

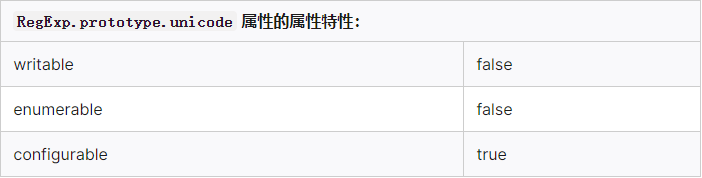
// 返回 [1, 2, 3]

console.log(Symbol.iterator in aGeneratorObject)

// 返回 true，因为 @@iterator 方法是 aGeneratorObject 的一个属性

# RegExp.prototype.unicode

unicode 属性表明正则表达式带有"u" 标志。 unicode 是正则表达式独立实例的只读属性。



unicode 的值是 Boolean，并且如果使用了 "u" 标志则为 true；否则为 false。"u" 标志开启了多种 Unicode 相关的特性。使用 "u" 标志，任何 Unicode 代码点的转义都会被解释。你不能直接修改这个属性，它是只读的。

使用 unicode 属性：

var regex = new RegExp('\u{61}', 'u');

console.log(regex.unicode); // true

# import

静态的 import 语句用于导入由另一个模块导出的绑定。无论是否声明了 strict mode，导入的模块都运行在严格模式下。在浏览器中，import 语句只能在声明了 type="module" 的 script 的标签中使用。此外，还有一个类似函数的动态 import()，它不需要依赖 type="module" 的 script 标签。

在 script 标签中使用 nomodule 属性，可以确保向后兼容。在您希望按照一定的条件或者按需加载模块的时候，动态 import() 是非常有用的。而静态型的 import 是初始化加载依赖项的最优选择，使用静态 import 更容易从代码静态分析工具和 tree shaking 中受益。

import defaultExport from "module-name";

import \* as name from "module-name";

import { export } from "module-name";

import { export as alias } from "module-name";

import { export1 , export2 } from "module-name";

import { foo , bar } from "module-name/path/to/specific/un-exported/file";

import { export1 , export2 as alias2 , [...] } from "module-name";

import defaultExport, { export [ , [...] ] } from "module-name";

import defaultExport, \* as name from "module-name";

import "module-name";

var promise = import("module-name");//这是一个处于第三阶段的提案。

* defaultExport：导入模块的默认导出接口的引用名。
* module-name：要导入的模块。通常是包含目标模块的 .js 文件的相对或绝对路径名，可以不包括 .js 扩展名。某些特定的打包工具可能允许或需要使用扩展或依赖文件，它会检查比对你的运行环境。只允许单引号和双引号的字符串。
* name：导入模块对象整体的别名，在引用导入模块时，它将作为一个命名空间来使用。
* export, exportN：被导入模块的导出接口的名称。
* alias, aliasN：将引用指定的导入的名称。

## 描述

name 参数是“导入模块对象”的名称，它将用一种名称空间来引用导入模块的接口。export 参数指定单个的命名导出，而 import \* as name 语法导入所有导出接口，即导入模块整体。以下示例阐明该语法。

### 导入整个模块的内容

这将myModule插入当前作用域，其中包含来自位于/modules/my-module.js文件中导出的所有接口。

import \* as myModule from '/modules/my-module.js';

在这里，访问导出接口意味着使用模块名称（在本例为“myModule”）作为命名空间。例如，如果上面导入的模块包含一个接口 doAllTheAmazingThings()，你可以这样调用：

myModule.doAllTheAmazingThings();

### 导入单个接口

给定一个名为 myExport 的对象或值，它已经从模块 my-module 导出（因为整个模块被导出）或显式地导出（使用 export 语句），将 myExport 插入当前作用域。

import {myExport} from '/modules/my-module.js';

### 导入多个接口

这将 foo 和 bar 插入当前作用域。

import {foo, bar} from '/modules/my-module.js';

### 导入带有别名的接口

你可以在导入时重命名接口。例如，将 shortName 插入当前作用域。

import {reallyReallyLongModuleExportName as shortName}

from '/modules/my-module.js';

### 导入时重命名多个接口

使用别名导入模块的多个接口。

import {

reallyReallyLongModuleMemberName as shortName,

anotherLongModuleName as short

} from '/modules/my-module.js';

### 仅为副作用而导入一个模块

整个模块仅为副作用（中性词，无贬义含义）而导入，而不导入模块中的任何内容（接口）。 这将运行模块中的全局代码，但实际上不导入任何值。

import '/modules/my-module.js';

### 导入默认值

引入模块可能有一个 default export（无论它是对象，函数，类等）可用。然后可以使用 import 语句来导入这样的默认接口。最简单的用法是直接导入默认值：

import myDefault from '/modules/my-module.js';

也可以同时将 default 语法与上述用法（命名空间导入或命名导入）一起使用。在这种情况下，default 导入必须首先声明。 例如：

import myDefault, \* as myModule from '/modules/my-module.js';

// myModule used as a namespace

Copy to Clipboard

或者

import myDefault, {foo, bar} from '/modules/my-module.js';

// specific, named imports

当用动态导入的方式导入默认导出时，其工作方式有所不同。你需要从返回的对象中解构并重命名 "default" 键。

(async () => {

if (somethingIsTrue) {

const { default: myDefault, foo, bar } = await import('/modules/my-module.js');

}

})();

### 动态 import

标准用法的 import 导入的模块是静态的，会使所有被导入的模块，在加载时就被编译（无法做到按需编译，降低首页加载速度）。有些场景中，你可能希望根据条件导入模块或者按需导入模块，这时你可以使用动态导入代替静态导入。下面的是你可能会需要动态导入的场景：

* 当静态导入的模块很明显的降低了代码的加载速度且被使用的可能性很低，或者并不需要马上使用它。
* 当静态导入的模块很明显的占用了大量系统内存且被使用的可能性很低。
* 当被导入的模块，在加载时并不存在，需要异步获取。
* 当导入模块的说明符，需要动态构建。（静态导入只能使用静态说明符）
* 当被导入的模块有副作用（这里说的副作用，可以理解为模块中会直接运行的代码），这些副作用只有在触发了某些条件才被需要时。（原则上来说，模块不能有副作用，但是很多时候，你无法控制你所依赖的模块的内容）

关键字 import 可以像调用函数一样来动态的导入模块。以这种方式调用，将返回一个 promise。

import('/modules/my-module.js')

.then((module) => {

// Do something with the module.

});

这种使用方式也支持 await 关键字。

let module = await import('/modules/my-module.js');

## 示例

下面的代码将会演示如何从辅助模块导入以协助处理 AJAX JSON 请求。

模块：file.js

function getJSON(url, callback) {

let xhr = new XMLHttpRequest();

xhr.onload = function () {

callback(this.responseText)

};

xhr.open('GET', url, true);

xhr.send();

}

export function getUsefulContents(url, callback) {

getJSON(url, data => callback(JSON.parse(data)));

}

主程序：main.js

import { getUsefulContents } from '/modules/file.js';

getUsefulContents('http://www.example.com',

data => { doSomethingUseful(data); });

Copy to Clipboard

动态导入：此示例展示了如何基于用户操作去加载功能模块到页面上，在例子中通过点击按钮，然后会调用模块内的函数。当然这不是能实现这个功能的唯一方式，import() 函数也可以支持 await。

const main = document.querySelector("main");

for (const link of document.querySelectorAll("nav > a")) {

link.addEventListener("click", e => {

e.preventDefault();

import('/modules/my-module.js')

.then(module => {

module.loadPageInto(main);

})

.catch(err => {

main.textContent = err.message;

});

});

}

# export

在创建 JavaScript 模块时，export 语句用于从模块中导出实时绑定的函数、对象或原始值，以便其他程序可以通过 import 语句使用它们。被导出的绑定值依然可以在本地进行修改。在使用 import 进行导入时，这些绑定值只能被导入模块所读取，但在 export 导出模块中对这些绑定值进行修改，所修改的值也会实时地更新。无论您是否声明，导出的模块都处于严格模式。 export 语句不能用在嵌入式脚本中。

存在两种 exports 导出方式：

* 命名导出（每个模块包含任意数量）
* 默认导出（每个模块包含一个）

// 导出单个特性

export let name1, name2, …, nameN; // also var, const

export let name1 = …, name2 = …, …, nameN; // also var, const

export function FunctionName(){...}

export class ClassName {...}

// 导出列表

export { name1, name2, …, nameN };

// 重命名导出

export { variable1 as name1, variable2 as name2, …, nameN };

// 解构导出并重命名

export const { name1, name2: bar } = o;

// 默认导出

export default expression;

export default function (…) { … } // also class, function\*

export default function name1(…) { … } // also class, function\*

export { name1 as default, … };

// 导出模块合集

export \* from …; // does not set the default export

export \* as name1 from …; // Draft ECMAScript® 2O21

export { name1, name2, …, nameN } from …;

export { import1 as name1, import2 as name2, …, nameN } from …;

export { default } from …;

Copy to Clipboard

* nameN：要导出的标识符（以便其他脚本通过 import 语句进行导入）.

## 描述

有两种不同的导出方式，命名导出和默认导出。你能够在每一个模块中定义多个命名导出，但是只允许有一个默认导出。每种方式对应于上述的一种语法：

命名导出：

// 导出事先定义的特性

export { myFunction, myVariable };

// 导出单个特性（可以导出 var，let，

//const,function,class）

export let myVariable = Math.sqrt(2);

export function myFunction() { ... };

Copy to Clipboard

默认导出：

// 导出事先定义的特性作为默认值

export { myFunction as default };

// 导出单个特性作为默认值

export default function () { ... }

export default class { .. }

// 每个导出都覆盖前一个导出

在导出多个值时，命名导出非常有用。在导入期间，必须使用相应对象的相同名称。但是，可以使用任何名称导入默认导出，例如：

// 文件 test.js

let k; export default k = 12;

Copy to Clipboard

// 另一个文件

import m from './test'; // 由于 k 是默认导出，所以可以自由使用 import m 替代 import k

console.log(m); // 输出为 12

你也可以重命名命名导出以避免命名冲突：

export { myFunction as function1, myVariable as variable };

为了使模块导入变得可用，在一个父模块中“导入/导出”这些不同模块也是可行的。也就是说，你可以创建单个模块，集中多个模块的多个导出。这个可以使用“export from”语法实现：

export { default as function1,

function2 } from 'bar.js';

Copy to Clipboard

与之形成对比的是联合使用导入和导出：

import { default as function1,

function2 } from 'bar.js';

export { function1, function2 };

但这里的 function1 和 function2 在当前模块中变得不可用。

## 示例

### 使用命名导出

在模块 my-module.js 中，可能包含以下代码：

// module "my-module.js"

function cube(x) {

return x \* x \* x;

}

const foo = Math.PI + Math.SQRT2;

var graph = {

options: {

color:'white',

thickness:'2px'

},

draw: function() {

console.log('From graph draw function');

}

}

export { cube, foo, graph };

### 使用默认导出

如果我们要导出一个值或得到模块中的返回值，就可以使用默认导出：

// module "my-module.js"

export default function cube(x) {

return x \* x \* x;

}

然后，在另一个脚本中，可以直接导入默认导出：

import cube from './my-module.js';

console.log(cube(3)); // 27

### 模块重定向

举个例子，假如我们有如下层次结构：

* childModule1.js: 导出 myFunction 和 myVariable
* childModule2.js: 导出 myClass
* parentModule.js: 作为聚合器（不做其他事情）
* 顶层模块：调用 parentModule.js 的导出项

你的代码看起来应该像这样：

// childModule1.js 中

let myFunction = ...; // assign something useful to myFunction

let myVariable = ...; // assign something useful to myVariable

export {myFunction, myVariable};

// childModule2.js 中

let myClass = ...; // assign something useful to myClass

export myClass;

// parentModule.js 中

// 仅仅聚合 childModule1 和 childModule2 中的导出

// 以重新导出他们

export { myFunction, myVariable } from 'childModule1.js';

export { myClass } from 'childModule2.js';

// 顶层模块中

// 我们可以从单个模块调用所有导出，因为 parentModule 事先

// 已经将他们“收集”/“打包”到一起

import { myFunction, myVariable, myClass } from 'parentModule.js'

# Proxy

Proxy 对象用于创建一个对象的代理，从而实现基本操作的拦截和自定义（如属性查找、赋值、枚举、函数调用等）。

* handler：包含捕捉器（trap）的占位符对象，可译为处理器对象。
* traps：提供属性访问的方法。这类似于操作系统中捕获器的概念。
* target：被 Proxy 代理虚拟化的对象。它常被作为代理的存储后端。根据目标验证关于对象不可扩展性或不可配置属性的不变量（保持不变的语义）。

## 语法

const p = new Proxy(target, handler)

* target：要使用 Proxy 包装的目标对象（可以是任何类型的对象，包括原生数组，函数，甚至另一个代理）。
* handler：一个通常以函数作为属性的对象，各属性中的函数分别定义了在执行各种操作时代理 p 的行为。

## 方法

创建一个可撤销的Proxy对象。

Proxy.revocable()

handler 对象是一个容纳一批特定属性的占位符对象。它包含有 Proxy 的各个捕获器（trap）。所有的捕捉器是可选的。如果没有定义某个捕捉器，那么就会保留源对象的默认行为。

* handler.getPrototypeOf() Object.getPrototypeOf 方法的捕捉器。
* handler.setPrototypeOf() Object.setPrototypeOf 方法的捕捉器。
* handler.isExtensible() Object.isExtensible 方法的捕捉器。
* handler.preventExtensions() Object.preventExtensions 方法的捕捉器。
* handler.getOwnPropertyDescriptor() Object.getOwnPropertyDescriptor 方法的捕捉器。
* handler.defineProperty() Object.defineProperty 方法的捕捉器。
* handler.has() in 操作符的捕捉器。
* handler.get() 属性读取操作的捕捉器。
* handler.set() 属性设置操作的捕捉器。
* handler.deleteProperty() delete操作符的捕捉器。
* handler.ownKeys() Object.getOwnPropertyNames 方法和 Object.getOwnPropertySymbols 方法的捕捉器。
* handler.apply() 函数调用操作的捕捉器。
* handler.construct() new操作符的捕捉器。

## 示例

### 基础示例

在以下简单的例子中，当对象中不存在属性名时，默认返回值为 37。下面的代码以此展示了 get handler 的使用场景。

const handler = {

get: function(obj, prop) {

return prop in obj ? obj[prop] : 37;

}

};

const p = new Proxy({}, handler);

p.a = 1;

p.b = undefined;

console.log(p.a, p.b); // 1, undefined

console.log('c' in p, p.c); // false, 37

### 无操作转发代理

在以下例子中，我们使用了一个原生 JavaScript 对象，代理会将所有应用到它的操作转发到这个对象上。

let target = {};

let p = new Proxy(target, {});

p.a = 37; // 操作转发到目标

console.log(target.a); // 37. 操作已经被正确地转发

### 验证

通过代理，你可以轻松地验证向一个对象的传值。下面的代码借此展示了 set handler 的作用。

let validator = {

set: function(obj, prop, value) {

if (prop === 'age') {

if (!Number.isInteger(value)) {

throw new TypeError('The age is not an integer');

}

if (value > 200) {

throw new RangeError('The age seems invalid');

}

}

// The default behavior to store the value

obj[prop] = value;

// 表示成功

return true;

}

};

let person = new Proxy({}, validator);

person.age = 100;

console.log(person.age);

// 100

person.age = 'young';

// 抛出异常: Uncaught TypeError: The age is not an integer

person.age = 300;

// 抛出异常: Uncaught RangeError: The age seems invalid

### 扩展构造函数

方法代理可以轻松地通过一个新构造函数来扩展一个已有的构造函数。这个例子使用了construct和apply。

function extend(sup, base) {

var descriptor = Object.getOwnPropertyDescriptor(

base.prototype, "constructor"

);

base.prototype = Object.create(sup.prototype);

var handler = {

construct: function(target, args) {

var obj = Object.create(base.prototype);

this.apply(target, obj, args);

return obj;

},

apply: function(target, that, args) {

sup.apply(that, args);

base.apply(that, args);

}

};

var proxy = new Proxy(base, handler);

descriptor.value = proxy;

Object.defineProperty(base.prototype, "constructor", descriptor);

return proxy;

}

var Person = function (name) {

this.name = name

};

var Boy = extend(Person, function (name, age) {

this.age = age;

});

Boy.prototype.sex = "M";

var Peter = new Boy("Peter", 13);

console.log(Peter.sex); // "M"

console.log(Peter.name); // "Peter"

console.log(Peter.age); // 13

### 操作 DOM 节点

需要互换两个不同的元素的属性或类名。下面的代码以此为目标，展示了 set handler 的使用场景。

let view = new Proxy({

selected: null

}, {

set: function(obj, prop, newval) {

let oldval = obj[prop];

if (prop === 'selected') {

if (oldval) {

oldval.setAttribute('aria-selected', 'false');

}

if (newval) {

newval.setAttribute('aria-selected', 'true');

}

}

// 默认行为是存储被传入 setter 函数的属性值

obj[prop] = newval;

// 表示操作成功

return true;

}

});

let i1 = view.selected = document.getElementById('item-1');

console.log(i1.getAttribute('aria-selected')); // 'true'

let i2 = view.selected = document.getElementById('item-2');

console.log(i1.getAttribute('aria-selected')); // 'false'

console.log(i2.getAttribute('aria-selected')); // 'true'

### 值修正及附加属性

以下products代理会计算传值并根据需要转换为数组。这个代理对象同时支持一个叫做 latestBrowser的附加属性，这个属性可以同时作为 getter 和 setter。

let products = new Proxy({

browsers: ['Internet Explorer', 'Netscape']

}, {

get: function(obj, prop) {

// 附加一个属性

if (prop === 'latestBrowser') {

return obj.browsers[obj.browsers.length - 1];

}

// 默认行为是返回属性值

return obj[prop];

},

set: function(obj, prop, value) {

// 附加属性

if (prop === 'latestBrowser') {

obj.browsers.push(value);

return;

}

// 如果不是数组，则进行转换

if (typeof value === 'string') {

value = [value];

}

// 默认行为是保存属性值

obj[prop] = value;

// 表示成功

return true;

}

});

console.log(products.browsers); // ['Internet Explorer', 'Netscape']

products.browsers = 'Firefox'; // 如果不小心传入了一个字符串

console.log(products.browsers); // ['Firefox'] <- 也没问题，得到的依旧是一个数组

products.latestBrowser = 'Chrome';

console.log(products.browsers); // ['Firefox', 'Chrome']

console.log(products.latestBrowser); // 'Chrome'

### 通过属性查找数组中的特定对象

以下代理为数组扩展了一些实用工具。如你所见，通过 Proxy，我们可以灵活地“定义”属性，而不需要使用 Object.defineProperties 方法。以下例子可以用于通过单元格来查找表格中的一行。在这种情况下，target 是 table.rows。

let products = new Proxy([

{ name: 'Firefox' , type: 'browser' },

{ name: 'SeaMonkey' , type: 'browser' },

{ name: 'Thunderbird', type: 'mailer' }

], {

get: function(obj, prop) {

// 默认行为是返回属性值， prop ?通常是一个整数

if (prop in obj) {

return obj[prop];

}

// 获取 products 的 number; 它是 products.length 的别名

if (prop === 'number') {

return obj.length;

}

let result, types = {};

for (let product of obj) {

if (product.name === prop) {

result = product;

}

if (types[product.type]) {

types[product.type].push(product);

} else {

types[product.type] = [product];

}

}

// 通过 name 获取 product

if (result) {

return result;

}

// 通过 type 获取 products

if (prop in types) {

return types[prop];

}

// 获取 product type

if (prop === 'types') {

return Object.keys(types);

}

return undefined;

}

});

console.log(products[0]); // { name: 'Firefox', type: 'browser' }

console.log(products['Firefox']); // { name: 'Firefox', type: 'browser' }

console.log(products['Chrome']); // undefined

console.log(products.browser); // [{ name: 'Firefox', type: 'browser' }, { name: 'SeaMonkey', type: 'browser' }]

console.log(products.types); // ['browser', 'mailer']

console.log(products.number); // 3

### 一个完整的 traps 列表示例

为了创建一个完整的 traps 列表示例，我们将尝试代理化一个非原生对象，这特别适用于这类操作：由发布在document.cookie页面上的“小型框架”创建的docCookies全局对象。

/\*

var docCookies = ... get the "docCookies" object here:

https://developer.mozilla.org/zh-CN/docs/DOM/document.cookie#A\_little\_framework.3A\_a\_complete\_cookies\_reader.2Fwriter\_with\_full\_unicode\_support

\*/

var docCookies = new Proxy(docCookies, {

"get": function (oTarget, sKey) {

return oTarget[sKey] || oTarget.getItem(sKey) || undefined;

},

"set": function (oTarget, sKey, vValue) {

if (sKey in oTarget) { return false; }

return oTarget.setItem(sKey, vValue);

},

"deleteProperty": function (oTarget, sKey) {

if (sKey in oTarget) { return false; }

return oTarget.removeItem(sKey);

},

"enumerate": function (oTarget, sKey) {

return oTarget.keys();

},

"ownKeys": function (oTarget, sKey) {

return oTarget.keys();

},

"has": function (oTarget, sKey) {

return sKey in oTarget || oTarget.hasItem(sKey);

},

"defineProperty": function (oTarget, sKey, oDesc) {

if (oDesc && "value" in oDesc) { oTarget.setItem(sKey, oDesc.value); }

return oTarget;

},

"getOwnPropertyDescriptor": function (oTarget, sKey) {

var vValue = oTarget.getItem(sKey);

return vValue ? {

"value": vValue,

"writable": true,

"enumerable": true,

"configurable": false

} : undefined;

},

});

/\* Cookies 测试 \*/

alert(docCookies.my\_cookie1 = "First value");

alert(docCookies.getItem("my\_cookie1"));

docCookies.setItem("my\_cookie1", "Changed value");

alert(docCookies.my\_cookie1);

# Symbol

symbol 是一种基本数据类型（primitive data type）。Symbol() 函数会返回 symbol 类型的值，该类型具有静态属性和静态方法。它的静态属性会暴露几个内建的成员对象；它的静态方法会暴露全局的 symbol 注册，且类似于内建对象类，但作为构造函数来说它并不完整，因为它不支持语法："new Symbol()"。每个从 Symbol() 返回的 symbol 值都是唯一的。一个 symbol 值能作为对象属性的标识符；这是该数据类型仅有的目的。

## 描述

Symbol([description])

* description ：可选的，字符串类型。对 symbol 的描述，可用于调试但不是访问 symbol 本身。

直接使用Symbol()创建新的 symbol 类型，并用一个可选的字符串作为其描述。

var sym1 = Symbol();

var sym2 = Symbol('foo');

var sym3 = Symbol('foo');

上面的代码创建了三个新的 symbol 类型。 注意，Symbol("foo") 不会强制将字符串 “foo” 转换成 symbol 类型。它每次都会创建一个新的 symbol 类型：

Symbol("foo") === Symbol("foo"); // false

如果你真的想创建一个 Symbol 包装器对象 (Symbol wrapper object)，你可以使用 Object() 函数：

var sym = Symbol("foo");

typeof sym; // "symbol"

var symObj = Object(sym);

typeof symObj; // "object"

### 全局共享的 Symbol

上面使用 Symbol() 函数的语法，不会在你的整个代码库中创建一个可用的全局的 symbol 类型。要创建跨文件可用的 symbol，甚至跨域（每个都有它自己的全局作用域），使用 Symbol.for() 方法和 Symbol.keyFor() 方法从全局的 symbol 注册表设置和取得 symbol。

### 在对象中查找 Symbol 属性

Object.getOwnPropertySymbols() 方法让你在查找一个给定对象的符号属性时返回一个 symbol 类型的数组。注意，每个初始化的对象都是没有自己的 symbol 属性的，因此这个数组可能为空，除非你已经在对象上设置了 symbol 属性。

## 属性

* Symbol.length：长度属性，值为 0。
* Symbol.prototype：symbol 构造函数的原型。

除了自己创建的 symbol，JavaScript 还内建了一些在 ECMAScript 5 之前没有暴露给开发者的 symbol，它们代表了内部语言行为。它们可以使用以下属性访问：

迭代 symbols Symbol.iterator：一个返回一个对象默认迭代器的方法。被 for...of 使用。

Symbol.asyncIterator Experimental：一个返回对象默认的异步迭代器的方法。被 for await of 使用。

正则表达式 symbols Symbol.match：一个用于对字符串进行匹配的方法，也用于确定一个对象是否可以作为正则表达式使用。被 String.prototype.match() 使用。

Symbol.replace：一个替换匹配字符串的子串的方法。被 String.prototype.replace() 使用。

Symbol.search：一个返回一个字符串中与正则表达式相匹配的索引的方法。被 String.prototype.search() 使用。

Symbol.split：一个在匹配正则表达式的索引处拆分一个字符串的方法.。被 String.prototype.split() 使用。

其他 symbols Symbol.hasInstance：一个确定一个构造器对象识别的对象是否为它的实例的方法。被 instanceof 使用。

Symbol.isConcatSpreadable：一个布尔值，表明一个对象是否应该 flattened 为它的数组元素。被 Array.prototype.concat() 使用。

Symbol.unscopables：拥有和继承属性名的一个对象的值被排除在与环境绑定的相关对象外。

Symbol.species：一个用于创建派生对象的构造器函数。

Symbol.toPrimitive：一个将对象转化为基本数据类型的方法。

Symbol.toStringTag：用于对象的默认描述的字符串值。被 Object.prototype.toString() 使用。

## 方法

* Symbol.for(key)：使用给定的 key 搜索现有的 symbol，如果找到则返回该 symbol。否则将使用给定的 key 在全局 symbol 注册表中创建一个新的 symbol。
* Symbol.keyFor(sym)：从全局 symbol 注册表中，为给定的 symbol 检索一个共享的？symbol key。

## Symbol 原型

所有 Symbols 继承自 Symbol.prototype 。

* Symbol.prototype.description：一个只读的字符串，意为对该 Symbol 对象的描述
* Symbol.prototype.toSource：返回该 Symbol 对象的源代码。该方法重写了 Object.prototype.toSource 方法
* Symbol.prototype.toString：返回一个包含着该 Symbol 对象描述的字符串。该方法重写了 Object.prototype.toString 方法
* Symbol.prototype.valueOf：返回该 Symbol 对象。该方法重写了 Object.prototype.valueOf 方法
* Symbol.prototype[@@toPrimitive]：返回该 Symbol 对象。

## 示例

### 对 symbol 使用 typeof 运算符

typeof运算符能帮助你识别 symbol 类型

typeof Symbol() === 'symbol'

typeof Symbol('foo') === 'symbol'

typeof Symbol.iterator === 'symbol'

### Symbol 类型转换

当使用 symbol 值进行类型转换时需要注意一些事情：

* 尝试将一个 symbol 值转换为一个 number 值时，会抛出一个 TypeError 错误 (e.g. +sym or sym | 0).
* 使用宽松相等时，Object(sym) == sym returns true.
* 这会阻止你从一个 symbol 值隐式地创建一个新的 string 类型的属性名。例如，Symbol("foo") + "bar" 将抛出一个 TypeError (can't convert symbol to string).
* "safer" String(sym) conversion 的作用会像 symbol 类型调用 Symbol.prototype.toString() 一样，但是注意 new String(sym) 将抛出异常。

### Symbols 与 for...in 迭代

Symbols 在 for...in 迭代中不可枚举。另外，Object.getOwnPropertyNames() 不会返回 symbol 对象的属性，但是你能使用 Object.getOwnPropertySymbols() 得到它们。

var obj = {};

obj[Symbol("a")] = "a";

obj[Symbol.for("b")] = "b";

obj["c"] = "c";

obj.d = "d";

for (var i in obj) {

console.log(i); // logs "c" and "d"

}

### Symbols 与 JSON.stringify()

当使用 JSON.stringify() 时，以 symbol 值作为键的属性会被完全忽略：

JSON.stringify({[Symbol("foo")]: "foo"});

// '{}'

### Symbol 包装器对象作为属性的键

当一个 Symbol 包装器对象作为一个属性的键时，这个对象将被强制转换为它包装过的 symbol 值：

var sym = Symbol("foo");

var obj = {[sym]: 1};

obj[sym]; // 1

obj[Object(sym)]; // still 1

# Promise

Promise 对象用于表示一个异步操作的最终完成（或失败）及其结果值。

## 描述

一个 Promise 对象代表一个在这个 promise 被创建出来时不一定已知值的代理。它让你能够把异步操作最终的成功返回值或者失败原因和相应的处理程序关联起来。这样使得异步方法可以像同步方法那样返回值：异步方法并不会立即返回最终的值，而是会返回一个 promise，以便在未来某个时候把值交给使用者。一个 Promise 必然处于以下几种状态之一：

* 待定（pending）：初始状态，既没有被兑现，也没有被拒绝。
* 已兑现（fulfilled）：意味着操作成功完成。
* 已拒绝（rejected）：意味着操作失败。

待定状态的 Promise 对象要么会通过一个值被兑现，要么会通过一个原因（错误）被拒绝。当这些情况之一发生时，我们用 promise 的 then 方法排列起来的相关处理程序就会被调用。如果 promise 在一个相应的处理程序被绑定时就已经被兑现或被拒绝了，那么这个处理程序也同样会被调用，因此在完成异步操作和绑定处理方法之间不存在竞态条件。

因为 Promise.prototype.then 和 Promise.prototype.catch 方法返回的是 promise，所以它们可以被链式调用。

[promise链式调用](promise的链式调用.png)

有一些语言中有惰性求值和延迟计算的特性，它们也被称为“promise”，例如 Scheme。JavaScript 中的 promise 代表的是已经在发生的进程，而且可以通过回调函数实现链式调用。如果你想对一个表达式进行惰性求值，就考虑一下使用无参数的箭头函数，如 f = () => expression 来创建惰性求值的表达式，然后使用 f() 进行求值。

如果一个 promise 已经被兑现或被拒绝，那么我们也可以说它处于 已敲定（settled） 状态。你还会听到一个经常跟 promise 一起使用的术语：已决议（resolved），它表示 promise 已经处于已敲定状态，或者为了匹配另一个 promise 的状态被“锁定”了。

### Promise 的链式调用

我们可以用Promise.prototype.then()、Promise.prototype.catch()和 Promise.prototype.finally() 这些方法将进一步的操作与一个变为已敲定状态的 promise 关联起来。

例如 .then() 方法需要两个参数，第一个参数作为处理已兑现状态的回调函数，而第二个参数则作为处理已拒绝状态的回调函数。每一个 .then() 方法还会返回一个新生成的 promise 对象，这个对象可被用作链式调用，就像这样：

const myPromise = new Promise((resolve, reject) => {

setTimeout(() => {

resolve('foo');

}, 300);

});

myPromise

.then(handleResolvedA, handleRejectedA)

.then(handleResolvedB, handleRejectedB)

.then(handleResolvedC, handleRejectedC);

当 .then() 中缺少能够返回 promise 对象的函数时，链式调用就直接继续进行下一环操作。因此，链式调用可以在最后一个 .catch() 之前把所有的处理已拒绝状态的回调函数都省略掉。

过早地处理变为已拒绝状态的 promise 会对之后 promise 的链式调用造成影响。不过有时候我们因为需要马上处理一个错误也只能这样做。例如，外面必须抛出某种类型的错误以在链式调用中传递错误状态。另一方面，在没有迫切需要的情况下，可以在最后一个 .catch() 语句时再进行错误处理，这种做法更加简单。.catch() 其实只是没有给处理已兑现状态的回调函数预留参数位置的 .then() 而已。

myPromise

.then(handleResolvedA)

.then(handleResolvedB)

.then(handleResolvedC)

.catch(handleRejectedAny);

使用箭头函数表达式作为 promise 回调函数的示例如下：

myPromise

.then(value => { return value + ' and bar'; })

.then(value => { return value + ' and bar again'; })

.then(value => { return value + ' and again'; })

.then(value => { return value + ' and again'; })

.then(value => { console.log(value) })

.catch(err => { console.log(err) });

这些函数的终止状态决定着链式调用中下一个 promise 的“已敲定”状态是什么。“已决议”状态意味着 promise 已经成功完成，而“已拒绝”则表示 promise 未成功完成。“已决议”状态的返回值会逐级传递到下一个 .then() 中，而“已拒绝”的理由则会被传递到链中的下一个已拒绝状态的处理函数。

链式调用中的 promise 们就像俄罗斯套娃一样，是嵌套起来的，但又像是一个栈，每个都必须从顶端被弹出。链式调用中的第一个 promise 是嵌套最深的一个，也将是第一个被弹出的。

(promise D, (promise C, (promise B, (promise A) ) ) )

一个 promise 可能会参与不止一次的嵌套。对于下面的代码，promiseA 向“已敲定”状态的过渡会导致两个实例的 .then 都被调用。

const promiseA = new Promise(myExecutorFunc);

const promiseB = promiseA.then(handleFulfilled1, handleRejected1);

const promiseC = promiseA.then(handleFulfilled2, handleRejected2);

一个已经处于“已敲定”状态的 promise 也可以接收操作。在那种情况下，（如果没有问题的话）这个操作会被作为第一个异步操作被执行。注意，所有的 promise 都一定是异步的。因此，一个已经处于“已敲定”状态的 promise 中的操作只有 promise 链式调用的栈被清空且一个时间片段过去之后才会被执行。这种效果跟 setTimeout(action, 10) 特别相似。

const promiseA = new Promise( (resolutionFunc,rejectionFunc) => {

resolutionFunc(777);

});

// At this point, "promiseA" is already settled.

promiseA.then( (val) => console.log("asynchronous logging has val:",val) );

console.log("immediate logging");

// produces output in this order:

// immediate logging

// asynchronous logging has val: 777

### 追踪现有设置对象

设置对象（settings object）是 JavaScript 代码运行时用于提供附加信息的环境。它包含了领域（realm）和模块映射（module map），以及 HTML 的特定信息，如来源（origin）等。对现有设置对象的追踪保证了浏览器知道用户给定的哪些代码片段需要使用。

我们可以粗略地认为领域是一个全局对象。其独特之处在于，它拥有运行 JavaScript 代码所需的所有信息。这包括像 Array 和 Error 这样的对象。每一个设置对象都有自己的“副本”，而且它们与副本之间是不共享的。这可能会导致一些与 promise 相关的意外行为。为了解决这个问题，我们需要追踪现有设置对象（incumbent settings object）。它表示负责用户某个函数调用工作的特定信息。

我们可以尝试在文档中嵌入 <iframe>，并让其与父级上下文通信。由于所有的 web API 都有现有设置对象，下面的代码能够在所有的浏览器中运行：

<!DOCTYPE html>

<iframe></iframe> <!-- we have a realm here -->

<script> // we have a realm here as well

const bound = frames[0].postMessage.bind(

frames[0], "some data", "\*");

// bound is a built-in function -- there is no user

// code on the stack, so which realm do we use?

window.setTimeout(bound);

// this still works, because we use the youngest

// realm (the incumbent) on the stack

</script>

同样的概念也适用与 promise。如果我们稍加修改上面的示例，我们就能得到这个：

<!DOCTYPE html>

<iframe></iframe> <!-- we have a realm here -->

<script> // we have a realm here as well

const bound = frames[0].postMessage.bind(

frames[0], "some data", "\*");

// bound is a built in function -- there is no user

// code on the stack -- which realm do we use?

Promise.resolve(undefined).then(bound);

// this still works, because we use the youngest

// realm (the incumbent) on the stack

</script>

如果我们修改代码，使用文档中的 <iframe> 来监听发送的消息，我们可以观察到现有设置对象的影响：

<!-- y.html -->

<!DOCTYPE html>

<iframe src="x.html"></iframe>

<script>

const bound = frames[0].postMessage.bind(frames[0], "some data", "\*");

Promise.resolve(undefined).then(bound);

</script>

Copy to Clipboard

<!-- x.html -->

<!DOCTYPE html>

<script>

window.addEventListener("message", (event) => {

document.querySelector("#text").textContent = "hello";

// 这一部分代码仅在追踪现有设置对象的浏览器中会被运行

console.log(event);

}, false);

</script>

在上面的示例中，<iframe> 仅在现有设置对象被追踪时才会被更新。这是因为在不追踪的情况下，我们可能会使用错误的环境发送消息。

## 方法解析

### 构造函数

* Promise()：创建一个新的 Promise 对象。该构造函数主要用于包装还没有添加 promise 支持的函数。

### 静态方法

* Promise.all(iterable)
  + 这个方法返回一个新的 promise 对象，等到所有的 promise 对象都成功或有任意一个 promise 失败。
  + 如果所有的 promise 都成功了，它会把一个包含 iterable 里所有 promise 返回值的数组作为成功回调的返回值。顺序跟 iterable 的顺序保持一致。
  + 一旦有任意一个 iterable 里面的 promise 对象失败则立即以该 promise 对象失败的理由来拒绝这个新的 promise。
* Promise.allSettled(iterable)
  + 等到所有 promise 都已敲定（每个 promise 都已兑现或已拒绝）。
  + 返回一个 promise，该 promise 在所有 promise 都敲定后完成，并兑现一个对象数组，其中的对象对应每个 promise 的结果。
* Promise.any(iterable)
  + 接收一个 promise 对象的集合，当其中的任意一个 promise 成功，就返回那个成功的 promise 的值。
* Promise.race(iterable)
  + 等到任意一个 promise 的状态变为已敲定。
  + 当 iterable 参数里的任意一个子 promise 成功或失败后，父 promise 马上也会用子 promise 的成功返回值或失败详情作为参数调用父 promise 绑定的相应处理函数，并返回该 promise 对象。
  + Promise.reject(reason)
  + 返回一个状态为已拒绝的 Promise 对象，并将给定的失败信息传递给对应的处理函数。
* Promise.resolve(value)
  + 返回一个状态由给定 value 决定的 Promise 对象。如果该值是 thenable（即，带有 then 方法的对象），返回的Promise 对象的最终状态由 then 方法执行结果决定；否则，返回的 Promise 对象状态为已兑现，并且将该 value 传递给对应的 then 方法。
  + 通常而言，如果你不知道一个值是否是 promise 对象，使用 Promise.resolve(value) 来返回一个 Promise 对象，这样就能将该 value 以 promise 对象形式使用。

### 实例方法

* Promise.prototype.catch()：为 promise 添加一个被拒绝状态的回调函数，并返回一个新的 promise，若回调函数被调用，则兑现其返回值，否则兑现原来的 promise 兑现的值。
* Promise.prototype.then()：为 promise 添加被兑现和被拒绝状态的回调函数，其以回调函数的返回值兑现 promise。若不处理已兑现或者已拒绝状态（例如，onFulfilled 或 onRejected 不是一个函数），则返回 promise 被敲定时的值。
* Promise.prototype.finally()：为 promise 添加一个回调函数，并返回一个新的 promise。这个新的 promise 将在原 promise 被兑现时兑现。而传入的回调函数将在原 promise 被敲定（无论被兑现还是被拒绝）时被调用。

## 示例

let myFirstPromise = new Promise((resolve,reject) => {

setTimeout(function(){

resolve("Success!")

}, 250)

});

myFirstPromise.then((msg) => {

console.log(msg);

})

# Reflect

Reflect 是一个内置的对象，它提供拦截 JavaScript 操作的方法。这些方法与proxy handlers的方法相同。Reflect不是一个函数对象，因此它是不可构造的。与大多数全局对象不同Reflect并非一个构造函数，所以不能通过new 运算符对其进行调用，或者将Reflect对象作为一个函数来调用。Reflect的所有属性和方法都是静态的（就像Math对象）。

## 静态方法

* Reflect.apply(target, thisArgument, argumentsList)：对一个函数进行调用操作，同时可以传入一个数组作为调用参数。和 Function.prototype.apply() 功能类似。
* Reflect.construct(target, argumentsList[, newTarget])：对构造函数进行 new 操作，相当于执行 new target(...args)。
* Reflect.defineProperty(target, propertyKey, attributes)：和 Object.defineProperty() 类似。如果设置成功就会返回 true
* Reflect.deleteProperty(target, propertyKey)：作为函数的delete操作符，相当于执行 delete target[name]。
* Reflect.get(target, propertyKey[, receiver])：获取对象身上某个属性的值，类似于 target[name]。
* Reflect.getOwnPropertyDescriptor(target, propertyKey)：类似于 Object.getOwnPropertyDescriptor()。如果对象中存在该属性，则返回对应的属性描述符，否则返回 undefined。
* Reflect.getPrototypeOf(target)：类似于 Object.getPrototypeOf()。
* Reflect.has(target, propertyKey)：判断一个对象是否存在某个属性，和 in 运算符 的功能完全相同。
* Reflect.isExtensible(target)：类似于 Object.isExtensible().
* Reflect.ownKeys(target)：返回一个包含所有自身属性（不包含继承属性）的数组。(类似于 Object.keys(), 但不会受enumerable 影响).
* Reflect.preventExtensions(target)：类似于 Object.preventExtensions()。返回一个Boolean。
* Reflect.set(target, propertyKey, value[, receiver])：将值分配给属性的函数。返回一个Boolean，如果更新成功，则返回true。
* Reflect.setPrototypeOf(target, prototype)：设置对象原型的函数。返回一个 Boolean， 如果更新成功，则返回true。

## 示例

检测一个对象是否存在特定属性：

const duck = {

name: 'Maurice',

color: 'white',

greeting: function() {

console.log(`Quaaaack! My name is ${this.name}`);

}

}

Reflect.has(duck, 'color');

// true

Reflect.has(duck, 'haircut');

// false

返回这个对象自身的属性：

Reflect.ownKeys(duck);

// [ "name", "color", "greeting" ]

为这个对象添加一个新的属性

Reflect.set(duck, 'eyes', 'black');

// returns "true" if successful

// "duck" now contains the property "eyes: 'black'"