# 函数的扩展

1. **[函数参数的默认值](http://es6.ruanyifeng.com/" \l "docs/function#函数参数的默认值)**
2. **[rest 参数](http://es6.ruanyifeng.com/" \l "docs/function#rest 参数)**
3. **[严格模式](http://es6.ruanyifeng.com/" \l "docs/function#严格模式)**
4. **[name 属性](http://es6.ruanyifeng.com/" \l "docs/function#name 属性)**
5. **[箭头函数](http://es6.ruanyifeng.com/" \l "docs/function#箭头函数)**
6. **[双冒号运算符](http://es6.ruanyifeng.com/" \l "docs/function#双冒号运算符)**
7. **[尾调用优化](http://es6.ruanyifeng.com/" \l "docs/function#尾调用优化)**
8. **[函数参数的尾逗号](http://es6.ruanyifeng.com/" \l "docs/function#函数参数的尾逗号)**
9. **[catch 语句的参数](http://es6.ruanyifeng.com/" \l "docs/function#catch 语句的参数)**

## 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

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

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

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

y = 'World';}

ES6 允许为函数的参数设置默认值，即直接写在参数定义的后面。

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

console.log(x, y);}

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

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

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

this.x = x;

this.y = y;}

const p = new Point();

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

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

参数变量是默认声明的，所以不能用let或const再次声明。

function foo(x = 5) {

let x = 1; // error const x = 2; // error}

上面代码中，参数变量x是默认声明的，在函数体中，不能用let或const再次声明，否则会报错。

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) {

// ...}

// SyntaxError: Duplicate parameter name not allowed in this context

另外，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。

作为练习，请问下面两种写法有什么差别？

// 写法一function m1({x = 0, y = 0} = {}) {

return [x, y];}

// 写法二function m2({x, y} = { x: 0, y: 0 }) {

return [x, y];}

上面两种写法都对函数的参数设定了默认值，区别是写法一函数参数的默认值是空对象，但是设置了对象解构赋值的默认值；写法二函数参数的默认值是一个有具体属性的对象，但是没有设置对象解构赋值的默认值。

// 函数没有参数的情况m1() // [0, 0]m2() // [0, 0]

// x 和 y 都有值的情况m1({x: 3, y: 8}) // [3, 8]m2({x: 3, y: 8}) // [3, 8]

// x 有值，y 无值的情况m1({x: 3}) // [3, 0]m2({x: 3}) // [3, undefined]

// x 和 y 都无值的情况m1({}) // [0, 0];m2({}) // [undefined, undefined]m1({z: 3}) // [0, 0]m2({z: 3}) // [undefined, undefined]

### 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: x is not defined

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”x 未定义“。

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.prototype.slice.call(arguments).sort();}

// rest参数的写法const sortNumbers = (...numbers) => numbers.sort();

上面代码的两种写法，比较后可以发现，rest 参数的写法更自然也更简洁。

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

function push(array, ...items) {

items.forEach(function(item) {

array.push(item);

console.log(item);

});}

var a = [];push(a, 1, 2, 3)

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

// 报错function f(a, ...b, c) {

// ...}

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;

上面的箭头函数等同于：

var f = function(v) {

return v;};

如果箭头函数不需要参数或需要多个参数，就使用一个圆括号代表参数部分。

var f = () => 5;

// 等同于var f = function () { return 5 };

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

// 等同于var sum = function(num1, num2) {

return num1 + num2;};

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

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

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

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

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

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。

// 正常函数写法[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

// 正常函数写法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]]

### 使用注意点

箭头函数有几个使用注意点。

（1）函数体内的this对象，就是定义时所在的对象，而不是使用时所在的对象。

（2）不可以当作构造函数，也就是说，不可以使用new命令，否则会抛出一个错误。

（3）不可以使用arguments对象，该对象在函数体内不存在。如果要用，可以用 rest 参数代替。

（4）不可以使用yield命令，因此箭头函数不能用作 Generator 函数。

上面四点中，第一点尤其值得注意。this对象的指向是可变的，但是在箭头函数中，它是固定的。

function foo() {

setTimeout(() => {

console.log('id:', this.id);

}, 100);}

var id = 21;

foo.call({ id: 42 });

// id: 42

上面代码中，setTimeout的参数是一个箭头函数，这个箭头函数的定义生效是在foo函数生成时，而它的真正执行要等到100毫秒后。如果是普通函数，执行时this应该指向全局对象window，这时应该输出21。但是，箭头函数导致this总是指向函数定义生效时所在的对象（本例是{id: 42}），所以输出的是42。

箭头函数可以让setTimeout里面的this，绑定定义时所在的作用域，而不是指向运行时所在的作用域。下面是另一个例子。

function Timer() {

this.s1 = 0;

this.s2 = 0;

// 箭头函数 setInterval(() => this.s1++, 1000);

// 普通函数 setInterval(function () {

this.s2++;

}, 1000);}

var timer = new Timer();

setTimeout(() => console.log('s1: ', timer.s1), 3100);setTimeout(() => console.log('s2: ', timer.s2), 3100);

// s1: 3// s2: 0

上面代码中，Timer函数内部设置了两个定时器，分别使用了箭头函数和普通函数。前者的this绑定定义时所在的作用域（即Timer函数），后者的this指向运行时所在的作用域（即全局对象）。所以，3100毫秒之后，timer.s1被更新了3次，而timer.s2一次都没更新。

箭头函数可以让this指向固定化，这种特性很有利于封装回调函数。下面是一个例子，DOM 事件的回调函数封装在一个对象里面。

var handler = {

id: '123456',

init: function() {

document.addEventListener('click',

event => this.doSomething(event.type), false);

},

doSomething: function(type) {

console.log('Handling ' + type + ' for ' + this.id);

}};

上面代码的init方法中，使用了箭头函数，这导致这个箭头函数里面的this，总是指向handler对象。否则，回调函数运行时，this.doSomething这一行会报错，因为此时this指向document对象。

this指向的固定化，并不是因为箭头函数内部有绑定this的机制，实际原因是箭头函数根本没有自己的this，导致内部的this就是外层代码块的this。正是因为它没有this，所以也就不能用作构造函数。

所以，箭头函数转成 ES5 的代码如下。

// ES6function foo() {

setTimeout(() => {

console.log('id:', this.id);

}, 100);}

// ES5function foo() {

var \_this = this;

setTimeout(function () {

console.log('id:', \_this.id);

}, 100);}

上面代码中，转换后的ES5版本清楚地说明了，箭头函数里面根本没有自己的this，而是引用外层的this。

请问下面的代码之中有几个this？

function foo() {

return () => {

return () => {

return () => {

console.log('id:', this.id);

};

};

};}

var f = foo.call({id: 1});

var t1 = f.call({id: 2})()(); // id: 1var t2 = f().call({id: 3})(); // id: 1var t3 = f()().call({id: 4}); // id: 1

上面代码之中，只有一个this，就是函数foo的this，所以t1、t2、t3都输出同样的结果。因为所有的内层函数都是箭头函数，都没有自己的this，它们的this其实都是最外层foo函数的this。

除了this，以下三个变量在箭头函数之中也是不存在的，指向外层函数的对应变量：arguments、super、new.target。

function foo() {

setTimeout(() => {

console.log('args:', arguments);

}, 100);}

foo(2, 4, 6, 8)

// args: [2, 4, 6, 8]

上面代码中，箭头函数内部的变量arguments，其实是函数foo的arguments变量。

另外，由于箭头函数没有自己的this，所以当然也就不能用call()、apply()、bind()这些方法去改变this的指向。

(function() {

return [

(() => this.x).bind({ x: 'inner' })()

];}).call({ x: 'outer' });

// ['outer']

上面代码中，箭头函数没有自己的this，所以bind方法无效，内部的this指向外部的this。

长期以来，JavaScript 语言的this对象一直是一个令人头痛的问题，在对象方法中使用this，必须非常小心。箭头函数”绑定”this，很大程度上解决了这个困扰。

### 嵌套的箭头函数

箭头函数内部，还可以再使用箭头函数。下面是一个 ES5 语法的多重嵌套函数。

function insert(value) {

return {into: function (array) {

return {after: function (afterValue) {

array.splice(array.indexOf(afterValue) + 1, 0, value);

return array;

}};

}};}

insert(2).into([1, 3]).after(1); //[1, 2, 3]

上面这个函数，可以使用箭头函数改写。

let insert = (value) => ({into: (array) => ({after: (afterValue) => {

array.splice(array.indexOf(afterValue) + 1, 0, value);

return array;}})});

insert(2).into([1, 3]).after(1); //[1, 2, 3]

下面是一个部署管道机制（pipeline）的例子，即前一个函数的输出是后一个函数的输入。

const pipeline = (...funcs) =>

val => funcs.reduce((a, b) => b(a), val);

const plus1 = a => a + 1;

const mult2 = a => a \* 2;

const addThenMult = pipeline(plus1, mult2);

addThenMult(5)

// 12

如果觉得上面的写法可读性比较差，也可以采用下面的写法。

const plus1 = a => a + 1;

const mult2 = a => a \* 2;

mult2(plus1(5))

// 12

箭头函数还有一个功能，就是可以很方便地改写λ演算。

// λ演算的写法fix = λf.(λx.f(λv.x(x)(v)))(λx.f(λv.x(x)(v)))

// ES6的写法var fix = f => (x => f(v => x(x)(v)))

(x => f(v => x(x)(v)));

上面两种写法，几乎是一一对应的。由于λ演算对于计算机科学非常重要，这使得我们可以用ES6作为替代工具，探索计算机科学。

## 双冒号运算符

箭头函数可以绑定this对象，大大减少了显式绑定this对象的写法（call、apply、bind）。但是，箭头函数并不适用于所有场合，所以现在有一个[提案](https://github.com/zenparsing/es-function-bind)，提出了“函数绑定”（function bind）运算符，用来取代call、apply、bind调用。

函数绑定运算符是并排的两个冒号（::），双冒号左边是一个对象，右边是一个函数。该运算符会自动将左边的对象，作为上下文环境（即this对象），绑定到右边的函数上面。

foo::bar;

// 等同于bar.bind(foo);

foo::bar(...arguments);

// 等同于bar.apply(foo, arguments);

const hasOwnProperty = Object.prototype.hasOwnProperty;function hasOwn(obj, key) {

return obj::hasOwnProperty(key);}

如果双冒号左边为空，右边是一个对象的方法，则等于将该方法绑定在该对象上面。

var method = obj::obj.foo;

// 等同于var method = ::obj.foo;

let log = ::console.log;

// 等同于var log = console.log.bind(console);

双冒号运算符的运算结果，还是一个对象，因此可以采用链式写法。

// 例一import { map, takeWhile, forEach } from "iterlib";

getPlayers()::map(x => x.character())::takeWhile(x => x.strength > 100)::forEach(x => console.log(x));

// 例二let { find, html } = jake;

document.querySelectorAll("div.myClass")::find("p")::html("hahaha");

## 尾调用优化

### 什么是尾调用？

尾调用（Tail Call）是函数式编程的一个重要概念，本身非常简单，一句话就能说清楚，就是指某个函数的最后一步是调用另一个函数。

function f(x){

return g(x);}

上面代码中，函数f的最后一步是调用函数g，这就叫尾调用。

以下三种情况，都不属于尾调用。

// 情况一function f(x){

let y = g(x);

return y;}

// 情况二function f(x){

return g(x) + 1;}

// 情况三function f(x){

g(x);}

上面代码中，情况一是调用函数g之后，还有赋值操作，所以不属于尾调用，即使语义完全一样。情况二也属于调用后还有操作，即使写在一行内。情况三等同于下面的代码。

function f(x){

g(x);

return undefined;}

尾调用不一定出现在函数尾部，只要是最后一步操作即可。

function f(x) {

if (x > 0) {

return m(x)

}

return n(x);}

上面代码中，函数m和n都属于尾调用，因为它们都是函数f的最后一步操作。

### 尾调用优化

尾调用之所以与其他调用不同，就在于它的特殊的调用位置。

我们知道，函数调用会在内存形成一个“调用记录”，又称“调用帧”（call frame），保存调用位置和内部变量等信息。如果在函数A的内部调用函数B，那么在A的调用帧上方，还会形成一个B的调用帧。等到B运行结束，将结果返回到A，B的调用帧才会消失。如果函数B内部还调用函数C，那就还有一个C的调用帧，以此类推。所有的调用帧，就形成一个“调用栈”（call stack）。

尾调用由于是函数的最后一步操作，所以不需要保留外层函数的调用帧，因为调用位置、内部变量等信息都不会再用到了，只要直接用内层函数的调用帧，取代外层函数的调用帧就可以了。

function f() {

let m = 1;

let n = 2;

return g(m + n);}f();

// 等同于function f() {

return g(3);}f();

// 等同于g(3);

上面代码中，如果函数g不是尾调用，函数f就需要保存内部变量m和n的值、g的调用位置等信息。但由于调用g之后，函数f就结束了，所以执行到最后一步，完全可以删除f(x)的调用帧，只保留g(3)的调用帧。

这就叫做“尾调用优化”（Tail call optimization），即只保留内层函数的调用帧。如果所有函数都是尾调用，那么完全可以做到每次执行时，调用帧只有一项，这将大大节省内存。这就是“尾调用优化”的意义。

注意，只有不再用到外层函数的内部变量，内层函数的调用帧才会取代外层函数的调用帧，否则就无法进行“尾调用优化”。

function addOne(a){

var one = 1;

function inner(b){

return b + one;

}

return inner(a);}

上面的函数不会进行尾调用优化，因为内层函数inner用到了外层函数addOne的内部变量one。

### 尾递归

函数调用自身，称为递归。如果尾调用自身，就称为尾递归。

递归非常耗费内存，因为需要同时保存成千上百个调用帧，很容易发生“栈溢出”错误（stack overflow）。但对于尾递归来说，由于只存在一个调用帧，所以永远不会发生“栈溢出”错误。

function factorial(n) {

if (n === 1) return 1;

return n \* factorial(n - 1);}

factorial(5) // 120

上面代码是一个阶乘函数，计算n的阶乘，最多需要保存n个调用记录，复杂度 O(n) 。

如果改写成尾递归，只保留一个调用记录，复杂度 O(1) 。

function factorial(n, total) {

if (n === 1) return total;

return factorial(n - 1, n \* total);}

factorial(5, 1) // 120

还有一个比较著名的例子，就是计算 Fibonacci 数列，也能充分说明尾递归优化的重要性。

非尾递归的 Fibonacci 数列实现如下。

function Fibonacci (n) {

if ( n <= 1 ) {return 1};

return Fibonacci(n - 1) + Fibonacci(n - 2);}

Fibonacci(10) // 89Fibonacci(100) // 堆栈溢出Fibonacci(500) // 堆栈溢出

尾递归优化过的 Fibonacci 数列实现如下。

function Fibonacci2 (n , ac1 = 1 , ac2 = 1) {

if( n <= 1 ) {return ac2};

return Fibonacci2 (n - 1, ac2, ac1 + ac2);}

Fibonacci2(100) // 573147844013817200000Fibonacci2(1000) // 7.0330367711422765e+208Fibonacci2(10000) // Infinity

由此可见，“尾调用优化”对递归操作意义重大，所以一些函数式编程语言将其写入了语言规格。ES6 是如此，第一次明确规定，所有 ECMAScript 的实现，都必须部署“尾调用优化”。这就是说，ES6 中只要使用尾递归，就不会发生栈溢出，相对节省内存。

### 递归函数的改写

尾递归的实现，往往需要改写递归函数，确保最后一步只调用自身。做到这一点的方法，就是把所有用到的内部变量改写成函数的参数。比如上面的例子，阶乘函数 factorial 需要用到一个中间变量total，那就把这个中间变量改写成函数的参数。这样做的缺点就是不太直观，第一眼很难看出来，为什么计算5的阶乘，需要传入两个参数5和1？

两个方法可以解决这个问题。方法一是在尾递归函数之外，再提供一个正常形式的函数。

function tailFactorial(n, total) {

if (n === 1) return total;

return tailFactorial(n - 1, n \* total);}

function factorial(n) {

return tailFactorial(n, 1);}

factorial(5) // 120

上面代码通过一个正常形式的阶乘函数factorial，调用尾递归函数tailFactorial，看起来就正常多了。

函数式编程有一个概念，叫做柯里化（currying），意思是将多参数的函数转换成单参数的形式。这里也可以使用柯里化。

function currying(fn, n) {

return function (m) {

return fn.call(this, m, n);

};}

function tailFactorial(n, total) {

if (n === 1) return total;

return tailFactorial(n - 1, n \* total);}

const factorial = currying(tailFactorial, 1);

factorial(5) // 120

上面代码通过柯里化，将尾递归函数tailFactorial变为只接受一个参数的factorial。

第二种方法就简单多了，就是采用 ES6 的函数默认值。

function factorial(n, total = 1) {

if (n === 1) return total;

return factorial(n - 1, n \* total);}

factorial(5) // 120

上面代码中，参数total有默认值1，所以调用时不用提供这个值。

总结一下，递归本质上是一种循环操作。纯粹的函数式编程语言没有循环操作命令，所有的循环都用递归实现，这就是为什么尾递归对这些语言极其重要。对于其他支持“尾调用优化”的语言（比如Lua，ES6），只需要知道循环可以用递归代替，而一旦使用递归，就最好使用尾递归。

### 严格模式

ES6 的尾调用优化只在严格模式下开启，正常模式是无效的。

这是因为在正常模式下，函数内部有两个变量，可以跟踪函数的调用栈。

* func.arguments：返回调用时函数的参数。
* func.caller：返回调用当前函数的那个函数。

尾调用优化发生时，函数的调用栈会改写，因此上面两个变量就会失真。严格模式禁用这两个变量，所以尾调用模式仅在严格模式下生效。

function restricted() {

'use strict';

restricted.caller; // 报错 restricted.arguments; // 报错}restricted();

### 尾递归优化的实现

尾递归优化只在严格模式下生效，那么正常模式下，或者那些不支持该功能的环境中，有没有办法也使用尾递归优化呢？回答是可以的，就是自己实现尾递归优化。

它的原理非常简单。尾递归之所以需要优化，原因是调用栈太多，造成溢出，那么只要减少调用栈，就不会溢出。怎么做可以减少调用栈呢？就是采用“循环”换掉“递归”。

下面是一个正常的递归函数。

function sum(x, y) {

if (y > 0) {

return sum(x + 1, y - 1);

} else {

return x;

}}

sum(1, 100000)

// Uncaught RangeError: Maximum call stack size exceeded(…)

上面代码中，sum是一个递归函数，参数x是需要累加的值，参数y控制递归次数。一旦指定sum递归100000次，就会报错，提示超出调用栈的最大次数。

蹦床函数（trampoline）可以将递归执行转为循环执行。

function trampoline(f) {

while (f && f instanceof Function) {

f = f();

}

return f;}

上面就是蹦床函数的一个实现，它接受一个函数f作为参数。只要f执行后返回一个函数，就继续执行。注意，这里是返回一个函数，然后执行该函数，而不是函数里面调用函数，这样就避免了递归执行，从而就消除了调用栈过大的问题。

然后，要做的就是将原来的递归函数，改写为每一步返回另一个函数。

function sum(x, y) {

if (y > 0) {

return sum.bind(null, x + 1, y - 1);

} else {

return x;

}}

上面代码中，sum函数的每次执行，都会返回自身的另一个版本。

现在，使用蹦床函数执行sum，就不会发生调用栈溢出。

trampoline(sum(1, 100000))

// 100001

蹦床函数并不是真正的尾递归优化，下面的实现才是。

function tco(f) {

var value;

var active = false;

var accumulated = [];

return function accumulator() {

accumulated.push(arguments);

if (!active) {

active = true;

while (accumulated.length) {

value = f.apply(this, accumulated.shift());

}

active = false;

return value;

}

};}

var sum = tco(function(x, y) {

if (y > 0) {

return sum(x + 1, y - 1)

}

else {

return x

}});

sum(1, 100000)

// 100001

上面代码中，tco函数是尾递归优化的实现，它的奥妙就在于状态变量active。默认情况下，这个变量是不激活的。一旦进入尾递归优化的过程，这个变量就激活了。然后，每一轮递归sum返回的都是undefined，所以就避免了递归执行；而accumulated数组存放每一轮sum执行的参数，总是有值的，这就保证了accumulator函数内部的while循环总是会执行。这样就很巧妙地将“递归”改成了“循环”，而后一轮的参数会取代前一轮的参数，保证了调用栈只有一层。

## 函数参数的尾逗号

ES2017 [允许](https://github.com/jeffmo/es-trailing-function-commas)函数的最后一个参数有尾逗号（trailing comma）。

此前，函数定义和调用时，都不允许最后一个参数后面出现逗号。

function clownsEverywhere(

param1,

param2) { /\* ... \*/ }

clownsEverywhere(

'foo',

'bar');

上面代码中，如果在param2或bar后面加一个逗号，就会报错。

如果像上面这样，将参数写成多行（即每个参数占据一行），以后修改代码的时候，想为函数clownsEverywhere添加第三个参数，或者调整参数的次序，就势必要在原来最后一个参数后面添加一个逗号。这对于版本管理系统来说，就会显示添加逗号的那一行也发生了变动。这看上去有点冗余，因此新的语法允许定义和调用时，尾部直接有一个逗号。

function clownsEverywhere(

param1,

param2,) { /\* ... \*/ }

clownsEverywhere(

'foo',

'bar',);

这样的规定也使得，函数参数与数组和对象的尾逗号规则，保持一致了。

## catch 语句的参数

目前，有一个[提案](https://github.com/tc39/proposal-optional-catch-binding)，允许try...catch结构中的catch语句调用时不带有参数。这个提案跟参数有关，也放在这一章介绍。

传统的写法是catch语句必须带有参数，用来接收try代码块抛出的错误。

try {

// ···} catch (error) {

// ···}

新的写法允许省略catch后面的参数，而不报错。

try {

// ···} catch {

// ···}

新写法只在不需要错误实例的情况下有用，因此不及传统写法的用途广。

let jsonData;try {

jsonData = JSON.parse(str);} catch {

jsonData = DEFAULT\_DATA;}

上面代码中，JSON.parse报错只有一种可能：解析失败。因此，可以不需要抛出的错误实例。