

# ECMAScript 6 入门

作者：阮一峰

授权：署名-非商用许可证

🔍

目录

- 0.前言
- 1.ECMAScript 6简介
- 2.let 和 const 命令
- 3.变量的解构赋值
- 4.字符串的扩展
- 5.正则的扩展
- 6.数值的扩展
- 7.函数的扩展
- 8.数组的扩展
- 9.对象的扩展
- 10.对象的新增方法
- 11.Symbol
- 12.Set 和 Map 数据结构
- 13.Proxy
- 14.Reflect
- 15.Promise 对象
- 16.Iterator 和 for...of 循环
- 17.Generator 函数的语法
- 18.Generator 函数的异步应用
- 19.async 函数
- 20.Class 的基本语法
- 21.Class 的继承
- 22.Decorator
- 23.Module 的语法
- 24.Module 的加载实现
- 25.编程风格
- 26.读懂规格
- 27.ArrayBuffer
- 28.最新提案
- 29.参考链接

其他

- 源码
- 修订历史
- 反馈意见

## 函数的扩展

- 1.函数参数的默认值
- 2.rest 参数
- 3.严格模式
- 4.name 属性
- 5.箭头函数
- 6.双冒号运算符
- 7.尾调用优化
- 8.函数参数的尾逗号

# 1. 函数参数的默认值

---

## 基本用法

ES6 之前，不能直接为函数的参数指定默认值，只能采用变通的方法。

```
function log(x, y) {
  y = y || 'World';
  console.log(x, y);
}

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

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

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

```
if (typeof y === 'undefined') {
  y = 'World';
}
```

ES6 允许为函数的参数设置默认值，即直接写在参数定义的后面。

```
function log(x, y = 'World') {
  console.log(x, y);
}

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

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

```
function Point(x = 0, y = 0) {
  this.x = x;
  this.y = y;
}

const p = new Point();
p // { x: 0, y: 0 }
```

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

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，[再声明](#)，[就会报错](#)，[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 5
foo({x: 1}) // 1 5
foo({x: 1, y: 2}) // 1 2
foo() // TypeError: Cannot read property 'x' of undefined
```

上面代码只使用了对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() // 3  
x // 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() // 2  
x // 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) { ... }
```

---

## 2. 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
```

---

## 3. 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;
  };
})();
```

---

## 4. 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 () {};

// ES5
f.name // ""

// ES6
f.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() {};

// ES5
bar.name // "baz"
```

```
// ES6  
bar.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 "
```

---

## 5. 箭头函数

---

### 基本用法

ES6 允许使用“箭头”（`=>`）定义函数。

```
var f = v => v;  
  
// 等同于  
var f = function (v) {  
  return v;  
};
```

如果箭头函数不需要参数或需要多个参数，就使用一个圆括号代表参数部分。

```
var f = () => 5;  
// 等同于  
var f = function () { return 5 };  
  
var sum = (num1, num2) => num1 + num2;  
// 等同于  
var sum = function(num1, num2) {  
  return num1 + num2;  
};
```

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

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

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

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

下面是一种特殊情况，虽然可以运行，但会得到错误的结果

[上一章](#)

[下一章](#)

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

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

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

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

箭头函数可以与变量解构结合使用。

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

// 等同于
function full(person) {
  return person.first + ' ' + person.last;
}
```

箭头函数使得表达更加简洁。

```
const isEven = n => n % 2 === 0;
const square = n => n * n;
```

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

箭头函数的一个用处是简化回调函数。

```
// 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。

```

// ES6
function foo() {
  setTimeout(() => {
    console.log('id:', this.id);
  }, 100);
}

// ES5
function 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: 1
var t2 = f().call({id: 3})(); // id: 1
var 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。

---

## 不适用场合

由于箭头函数使得 `this` 从“动态”变成“静态”，下面两个场合不应该使用箭头函数。

第一个场合是定义函数的方法，且该方法内部包括 `this`。

```
const cat = {
  lives: 9,
  jumps: () => {
    this.lives--;
  }
}
```

上面代码中，`cat.jumps()` 方法是一个箭头函数，这是错误的。调用 `cat.jumps()` 时，如果是普通函数，该方法内部的 `this` 指向 `cat`；如果写成上面那样的箭头函数，使得 `this` 指向全局对象，因此不会得到预期结果。

第二个场合是需要动态 `this` 的时候，也不应使用箭头函数。

```
var button = document.getElementById('press');
button.addEventListener('click', () => {
  this.classList.toggle('on');
});
```

上面代码运行时，点击按钮会报错，因为 `button` 的监听函数是一个箭头函数，导致里面的 `this` 就是全局对象。如果改成普通函数，`this` 就会动态指向被点击的按钮对象。

另外，如果函数体很复杂，有许多行，或者函数内部有大量的读写操作，不单纯是为了计算值，这时也不应该使用箭头函数，而是要使用普通函数，这样可以提高代码可读性。

---

## 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  $\lambda$  演算。

```
//  $\lambda$ 演算的写法
fix =  $\lambda f.(\lambda x.f(\lambda v.x(x)(v)))(\lambda x.f(\lambda v.x(x)(v)))$ 

// 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  $\lambda$  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。

## 6. 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，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);
```

如果双冒号运算符的运算结果，还是一个对象，就可以采用链式写法。

```
import { map, takeWhile, forEach } from "iterlib";

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

---

## 7. 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) // 89
Fibonacci(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) // 573147844013817200000
Fibonacci2(1000) // 7.0330367711422765e+208
Fibonacci2(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;
```

[上一章](#)

[下一章](#)

```
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。

## 8. 函数参数的尾逗号

ES2017 允许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。

留言

188 Comments

ECMAScript 6 入门

 Login ▾

 Recommend 12

 Tweet

 Share

Sort by Best ▾



Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS 

Name



**Huazz Tsai** • 3 years ago

感觉以后看别人的代码越来越困难了

33 ^ | v • Reply • Share ›



**Mickey-** ➔ Huazz Tsai • 3 years ago

哈哈，握一爪

2 ^ | v • Reply • Share ›



**p2227** • 4 years ago

[...5] 如何扩展成 [0,1,2,3,4,5] ?

47 ^ | v • Reply • Share ›



**Doujun Zhang** ➔ p2227 • 15 days ago

数组展开操作符只能作用于部署了遍历器对象的数据结构，如果非要实现这种操作，可以使用下面的实现：

```
Number.prototype[Symbol.iterator] = function*() {  
  let i = 0;  
  let num = this.valueOf();  
  while (i <= num) {  
    yield i++;  
  }  
}
```

```
console.log([...5]);
```

2 ^ | v • Reply • Share ›



**ruanyf** Mod ➔ Doujun Zhang • 15 days ago

这个例子很好，我放入教程了。

1 ^ | v • Reply • Share ›



**Doujun Zhang** ➔ ruanyf • 13 days ago

嗯呢

^ | v • Reply • Share ›



**happy wang** • a year ago

关于函数参数默认值这块，有个小问题

```
function init({list = [], placeholder = ""} = {}) {  
  console.log(list)  
  console.log(placeholder)  
}
```

上面这个函数，是可以很好的规范传递给函数的对象里面，**list**和**placeholder**的默认值。

但是我想在使用的时候，这个对象是以一个整体的形式去引用。

但是现在因为做了解构，这个对象形参就没了名字，没法直接引用。

我尝试过用**arguments[0]**的方式使用，但是这个得到的是实际传递的，而不是进过默认值修饰过的。

不知道这块有什么好的实践方案吗，还

6 ^ | v • Reply • Share ›





郭晓刚 → happy wang • a month ago

看到有推荐这种写法的  

```
function init(data= {}) {
  let {list = [], placeholder = ""} = data
  console.log(list)
  console.log(placeholder)
  console.log(data)
}
```

^ | v • Reply • Share ›



ruanyf Mod → happy wang • a year ago

这样写试试看, `function init(obj, {list = [], placeholder = ""} = {}) {}`

2017-07-25 21:21 GMT+08:00 Disqus <notifications@disqus.net>:

^ | v • Reply • Share ›



宋增 → ruanyf • a year ago

他这个的意思是既想获取到 `obj` 本身, 也想解构 `obj` 里面的值, 您的这种写法调用 `init` 函数时需要传入两次 `obj` 才能满足他的需求, `function init(obj = {}, {list = [], placeholder = ""} = obj) {}` 这个写法是否更好一点, 调用时传入一次 `obj` 就行。

2 ^ | v • Reply • Share ›



yy • 3 years ago

报个小问题, `document.addEventListener`, 如果不用箭头函数, 回调中 `this` 指向 `document`, 而不是全局对象

5 ^ | v • Reply • Share ›



UoiCano → yy • 2 years ago

``document.addEventListener('click', e => {console.log(this)});``

可我在 `chrome stable` 上得到的是 ``window`` 对象呢。

^ | v • Reply • Share ›



邓臻华 → yy • 2 years ago

在文档中全局对象为 `document`, 当回调函数调用时 `this` 已经从 `window` 更改为 `document` 了。举个例子

```
var a = 666;
function test() {
  console.log(a);
}
```

```
a = 888;
test(); // 888
```

^ | v • Reply • Share ›



Youmu • 5 years ago

箭头函数本质上就是 `lambda` 表达式嘛 = =

3 ^ | v • Reply • Share ›



J.W → Youmu • 5 years ago

照这么说 `function` 的本质也是 `lambda` 表达式。

[上一章](#)

[下一章](#)

^ | v • Reply • Share ›



**Youmu** → J.W • 5 years ago

function可以有side effect. lambda表达式ensure pure function.

2 ^ | v • Reply • Share ›



**J.W** → Youmu • 5 years ago

我就是看到有this，而且可以引用全局变量，箭头函数也是Object（可以添加自定义属性），才不觉得它是纯粹的lambda表达式.....

```
( () => this.x=2 )();  
( () => console.log(x) )();
```

^ | v • Reply • Share ›



**Liu Ning** • 4 years ago

30行的PHP理解他，顶多只需要5分钟，但30行的JS也许一天两天你都可以无法理解。未来JS可读性以后是个大问题

5 ^ | v • Reply • Share ›



**zhe-he** → Liu Ning • 3 years ago

30行php只做一件事，

30行js可以做十件事

2 ^ | v • Reply • Share ›



**Jacob** → zhe-he • 2 years ago

PHP也在更新啊，也不是php 5那个时代了，而且OO能力直接秒了JS.

^ | v • Reply • Share ›

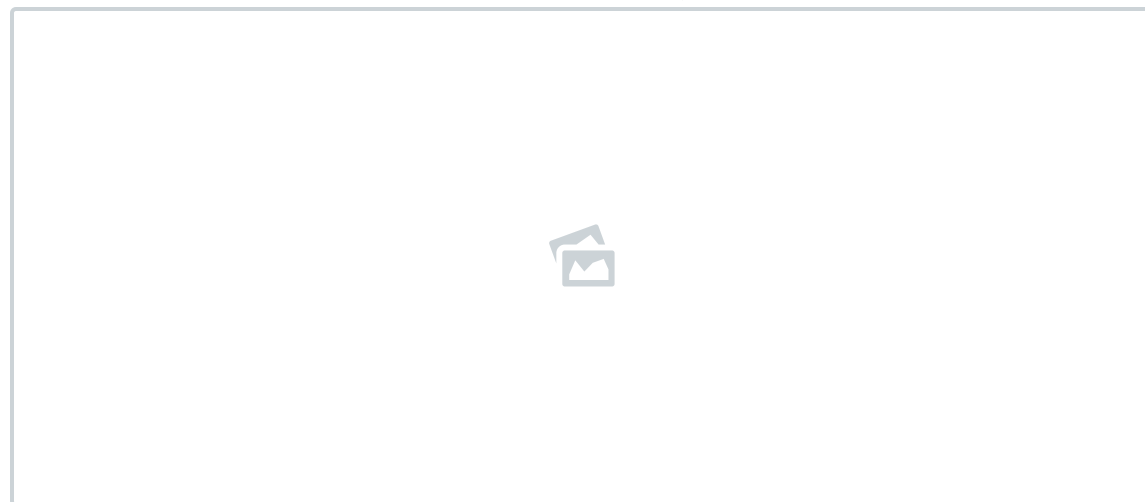


**zhe-he** → Jacob • 2 years ago

这个地方要翻墙才能回复，--!

js也在更新，比如typescript.

能写出人能看懂的代码就好的代码，多少行最后压缩都是一行。



^ | v • Reply • Share ›



**Chenghao Lan** • 3 years ago

这是.....酒糖

2 ^ | v • Reply • Share ›



**Frank Fang** • 5 years ago

三颗糖！

3 ^ | v • Reply • Share ›



**Amio** ➔ Frank Fang • 5 years ago

ES6一大半都是语法糖.....这样真的好吗

5 ^ | v • Reply • Share ›



**Leo SC** ➔ Amio • 3 years ago

每个人写的代码都不一样，有些人觉得缺乏统一性，但是实际上和语言多样性一样。同一个意思有不同的表达方式，使你的代码读起来不是那么生涩。不过确实新人很难读。团队合作的话，确保每个人自己的模块写好就可以啦。暴露出来的那部分代码统一，具体怎么实现黑盒就好。

1 ^ | v • Reply • Share ›



**旺旺** ➔ Amio • 3 years ago

不给块糖，js这么多的坑要怎么过

^ | v • Reply • Share ›



**Frank Fang** ➔ Amio • 5 years ago

加糖有什么不好？你可以选择完全不用这些糖，你的程序照样能运行。像 Python 3 一样大改才蛋疼。

^ | v • Reply • Share ›



**Amio** ➔ Frank Fang • 5 years ago

这样代码风格统一的成本会更高，对于 JS 的学习者而言负担也更重了。况且，某些语法糖的风格并不直观优雅

5 ^ | v • Reply • Share ›



**超王** ➔ Amio • 4 years ago

代码风格多样是js永远的痛，最痛的还是接手别人的代码的时候...其实不是别人的代码不好，而是风格太多变了，尤其是多个人维护的代码，ES6的语法糖，更是多的醉人。

^ | v • Reply • Share ›



**Stuart Zhang** ➔ 超王 • 3 years ago

ESLint有一个Stylistic规则集几乎能够完全解决《代码风格》统一化的问题。而且，ESLint还允许开发团队写自己的rule。

如果团队里真有那个“愣头青”写出来的代码与预定义的代码风格不一致的话，直接让他的代码build失败。Grunt-ESLint插件有一个silent的属性，把这个属性设置为false。ESLint的代码静态扫描失败直接导致整个工程的build失败。看谁还胡乱写来着。对于CSS，StyleLint也能够起到相同的作用。

找解决问题的办法 比 抱怨 更管用。

1 ^ | v • Reply • Share ›



**Hanyu Jiang** ➔ 超王 • 2 years ago

这样好啊，多活多工资：)

^ | v • Reply • 上一章 下一章



**t\_tfrozen** → 超王 • 4 years ago

现在的js还好啦，，看看人家ruby那语法，，ES6也比不了\ (￣▽￣) ㄟ

^ | v • Reply • Share ›



**Orange** • 6 months ago

阮老师，拜读您的文章有些不解的地方。麻烦您帮忙指点迷津。

“蹦床函数并不是真正的尾递归优化，下面的实现才是。”

看完这章，我有以下两点疑问：

1. 为什么蹦床函数不是真正的尾递归？
2. 严格模式下的尾递归仅存储一个累加和，而下面所谓的“真正的尾递归”使用栈结构`accumulated`存储递归参数相比严格模式下的尾递归有额外的空间开销，应该也不是真正的尾递归吧？

1 ^ | v • Reply • Share ›



**Orange** → Orange • 5 months ago

自己解答一下第二个疑问哈，相比严格模式下的尾递归是没有额外的空间开销的，因为每次都会进行`unshift`弹出参数，所以保证参数栈不会扩大（之前代码没仔细读懂，2333）

^ | v • Reply • Share ›



**杨小** • 9 months ago

哇，阮老师，我用了“尾递归优化过的 `Fibonacci`” 这里尾递归优化过的代码，还是栈溢出了。什么鬼。



1 ^ | v • Reply • Share ›



**ruanyf** Mod → 杨小 • 9 months ago

**chrome** 现在还不支持尾递归优化

Disqus <notifications@disqus.net> 于 2018年4月13日周五 13:13写道：

2 ^ | v • Reply • Share ›



**choreHu** • a year ago

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。

请问这里和函数作用域是什么关系？

1 ^ | v • Reply • Share ›





linkkingjay • 4 years ago

“函数绑定运算符是并排的两个双引号 (::)”应该是两个冒号。

1 ^ | v • Reply • Share ›



maxpeng • 3 years ago

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。

@ruanyf 我在 chrome50 和 Firefox45 上验证了一下，和描述不符。

example1: var a = function(a=0, b, c){console.log(a,b,c)}

example2: var b = function(a, b=0, c){console.log(a,b,c)}

example3: var c = function(a, b, c=0){console.log(a,b,c)}

1 ^ | v • Reply • Share ›



ruanyf Mod → maxpeng • 3 years ago

看来只要用了默认值，JavaScript引擎就不再预期后面的参数了。

^ | v • Reply • Share ›



maxpeng → ruanyf • 3 years ago

我的理解是引擎会从左到右扫描参数，若参数没有默认值则 length加1，否则短路退出。

^ | v • Reply • Share ›



于振楠 • a month ago

javascript这语言复杂度堪比C++了，已经远远脱离原本的设计定位，典型的小三强上位

^ | v • Reply • Share ›



Hank Chan • 4 months ago

目前只有 Safari 有支援 尾递归优化, Chrome 和 node 都没有~

<https://kangax.github.io/co...>

^ | v • Reply • Share ›



杨小 • 5 months ago

大佬，问个困扰我很久的问题，

```
``js
```

```
function foo() {
  setTimeout(() => {
    console.log('id:', this.id);
  }, 100);
}
```

```
var id = 21;
```

```
foo.call({ id: 42 });
```

```
...
```

箭头函数绑定外层的this,在上面的例子，我理解的是箭头函数是在setTimeout里，绑定的this是window,而不是foo里的this({ id: 42 })

还有箭头函数作为对象属性的时候

```
``js
```

```
a = {
```

```
name: 'a',
sayName: () => {
  console.log(this.name) //undefined
}
}
...
```

这里是因为js没有对象作用域吗？  
先谢谢大佬了

^ | v • Reply • Share ›



**ruanyf** Mod ➔ 杨小 • 5 months ago

函数定义时，不会执行内部代码，所以 `setTimeout` 里面的代码没有执行，所以没有绑定 `this`。

^ | v • Reply • Share ›



**Oliverance** • 7 months ago

软老师，你好，我有在GitHub上看到trekhleb/javascript-algorithms中作者用循环的方式实现了非递归模式下的阶乘运算，您能分析一下它与您写法上复杂度和stack溢出方面的区别吗？



^ | v • Reply • Share ›



任维涛 • 7 months ago



这一段就看不懂了，y这里是一个函数，foo函数内运行y，最近的x应该是foo内部定义的x，这里x没变，意思是y内的x对函数参数作用域内的x做了绑定？

^ | v • Reply • Share ›



**ruanyf** Mod ➔ 任维涛 • 7 months ago

原因是 函数参数 与 函数体 是两个作用域。函数体里面的 x 与 y 里面的 x 是不一样的。

^ | v • Reply • Share ›



杨小 • 10 months ago

```
let func = () => { a: 1 }
```

这里没有报错

但是下面这样

```
let getTempltem = id => { id: id, name: "Temp" }
```

就报错了。

这个和“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”这段描述有冲突啊。为啥，阮老师

Chrome 65.0.3325.181

^ | v • Reply • Share ›



杨小 ➔ 杨小 • 10 months ago

我知道了，js a:1 居然就输出了1，然后啥也没做什么鬼啊

^ | v • Reply • Share ›

[Load more comments](#)

ALSO ON ECMASCRIPT 6 入门

## let和const命令

222 comments • 5 years ago

Seaborn Lee —

## Iterator和for...of循环

54 comments • 5 years ago

**miusuncle** — 其实是不一样的，不通过var或let声明的话，该变量会污染全局环境的：

```
// badvoid function () {
for ...
```

## 修饰器

43 comments • 3 years ago

**bpceee** — 方法的修饰，target 是 Class 的 prototype，并不是实例。这个时候实例还未创建出来。

## Module 的加载实现

18 comments • 2 years ago

**ruanyf** — 你说的对，这里是有错，我已经把缓存这部分删掉了。import() 可以用 Babel 执行。``javascript// ...

Subscribe Add Disqus to your siteAdd DisqusAdd Disqus' Privacy PolicyPrivacy PolicyPrivacy Policy

