

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. 数组的扩展
- 8. 函数的扩展
- 9. 对象的扩展
- 10. Symbol
- 11. Proxy和Reflect
- 12. 二进制数组
- 13. Set和Map数据结构
- 14. Iterator和for...of循环
- 15. Generator函数
- 16. Promise对象
- 17. 异步操作和Async函数
- 18. Class
- 19. Decorator
- 20. Module
- 21. 编程风格

函数的扩展

- 1. 函数参数的默认值
- 2. rest参数
- 3. 扩展运算符
- 4. name属性
- 5. 箭头函数
- 6. 函数绑定
- 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

在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);  
}
```

21. 测试用例

22. 读懂规格

23. 参考链接

其他

- 源码
- 修订历史
- 反馈意见

```
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;
}
```

```
}  
  
var 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。

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
```

上面代码使用了对象的解构赋值默认值，而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` 才会通过解构赋值而生成。如果函数 `foo` 调用时参数不是对象，变量 `x` 和 `y` 就不会生成，从而报错。如果参数对象没有 `y` 属性，`y` 的默认值5才会生效。

下面是另一个对象的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。

上面的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, { method = 'GET' } = {}) {
  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`。

再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，`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

一个需要注意的地方是，如果参数默认值是一个变量，则该变量所处的作用域，与其他变量的作用域规则是一样的，即先是当前函数的作用域，然后才是全局作用域。

```
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 `x`。调用时，由于函数作用域内部的变量 `x` 已经生成，所以 `y` 等于参数 `x`，而不是全局变量 `x`。

如果调用时，函数作用域内部的变量 `x` 没有生成，结果就会不一样。

```
let x = 1;  
  
function f(y = x) {  
  let x = 2;  
  console.log(y);  
}  
  
f() // 1
```

上面代码中，函数调用时，`y` 的默认值变量 `x` 尚未在函数内部生成，所以 `x` 指向全局变量。

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，函数 `foo` 的参数 `x` 的默认值也是 `x`。这时，默认值 `x` 的作用域是函数作用域，而不是全局作用域。由于在函数作用域中，存在变量 `x`，但是默认值在 `x` 赋值之前先执行了，所以这时属于暂时性死区（参见《`let`和`const`命令》一章），任何对 `x` 的操作都会报错。

如果参数的默认值是一个函数，该函数的作用域是其声明时所在的作用域。请看下面的例子。

```
let foo = 'outer';

function bar(func = x => foo) {
  let foo = 'inner';
  console.log(func()); // outer
}

bar();
```

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`。这个匿名函数声明时，`bar` 函数的作用域还没有形成，所以匿名函数里面的 `foo` 指向外层作

用域的 `foo`，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 `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
```

上面代码中，函数 `foo` 的参数 `y` 的默认值是一个匿名函数。函数 `foo` 调用时，它的参数 `x` 的值为 `undefined`，所以 `y` 函数内部的 `x` 一开始是 `undefined`，后来被重新赋值 `2`。但是，函数 `foo` 内部重新声明了一个 `x`，值为 `3`，这两个 `x` 是不一样的，互相不产生影响，因此最后输出 `3`。

如果将 `var x = 3` 的 `var` 去除，两个 `x` 就是一样的，最后输出的就是 `2`。

```
var x = 1;
function foo(x, y = function() { x = 2; }) {
  x = 3;
  y();
  console.log(x);
}

foo() // 2
```

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（即函数名之后有一对圆括号），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（即如果参数已经赋值，默认值中的函数就不会运行），这与python语言不一样。

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。

`rest`参数中的变量代表一个数组，所以数组特有的方法都可以用于这个变量。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. 扩展运算符

含义

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

```
console.log(...[1, 2, 3])  
// 1 2 3  
  
console.log(1, ...[2, 3, 4], 5)  
// 1 2 3 4 5  
  
[...document.querySelectorAll('div')]  
// [<div>, <div>, <div>]
```

该运算符主要用于函数调用。

```
function push(array, ...items) {  
  array.push(...items);  
}  
  
function add(x, y) {  
  return x + y;  
}  
  
var numbers = [4, 38];  
add(...numbers) // 42
```

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

扩展运算符与正常的函数参数可以结合使用，非常灵活。

```
function f(v, w, x, y, z) { }  
var args = [0, 1];  
f(-1, ...args, 2, ...[3]);
```

替代数组的 **apply** 方法

由于扩展运算符可以展开数组，所以不再需要 `apply` 方法，将数组转为函数的参数了。

```
// ES5的写法
```



```
function f(x, y, z) {  
    // ...  
}  
var args = [0, 1, 2];  
f.apply(null, args);  
  
// ES6的写法  
function f(x, y, z) {  
    // ...  
}  
var args = [0, 1, 2];  
f(...args);
```

下面是扩展运算符取代 `apply` 方法的一个实际的例子，应用 `Math.max` 方法，简化求出一个数组最大元素的写法。

```
// ES5的写法  
Math.max.apply(null, [14, 3, 77])  
  
// ES6的写法  
Math.max(...[14, 3, 77])  
  
// 等同于  
Math.max(14, 3, 77);
```

上面代码表示，由于JavaScript不提供求数组最大元素的函数，所以只能套用 `Math.max` 函数，将数组转为一个参数序列，然后求最大值。有了扩展运算符以后，就可以直接用 `Math.max` 了。

另一个例子是通过 `push` 函数，将一个数组添加到另一个数组的尾部。

```
// ES5的写法
var arr1 = [0, 1, 2];
var arr2 = [3, 4, 5];
Array.prototype.push.apply(arr1, arr2);

// ES6的写法
var arr1 = [0, 1, 2];
var arr2 = [3, 4, 5];
arr1.push(...arr2);
```

上面代码的ES5写法中，`push`方法的参数不能是数组，所以只好通过`apply`方法变通使用`push`方法。有了扩展运算符，就可以直接将数组传入`push`方法。

下面是另外一个例子。

```
// ES5
new (Date.bind.apply(Date, [null, 2015, 1, 1]))
// ES6
new Date(...[2015, 1, 1]);
```

扩展运算符的应用

(1) 合并数组

扩展运算符提供了数组合并的新写法。

```
// ES5
[1, 2].concat(more)
```

```
// ES6
[1, 2, ...more]

var arr1 = ['a', 'b'];
var arr2 = ['c'];
var arr3 = ['d', 'e'];

// ES5的合并数组
arr1.concat(arr2, arr3);
// [ 'a', 'b', 'c', 'd', 'e' ]

// ES6的合并数组
[...arr1, ...arr2, ...arr3]
// [ 'a', 'b', 'c', 'd', 'e' ]
```

(2) 与解构赋值结合

扩展运算符可以与解构赋值结合起来，用于生成数组。

```
// ES5
a = list[0], rest = list.slice(1)
// ES6
[a, ...rest] = list
```

下面是另外一些例子。

```
const [first, ...rest] = [1, 2, 3, 4, 5];
first // 1
rest  // [2, 3, 4, 5]

const [first, ...rest] = [];
first // undefined
```

```
rest // []:  
  
const [first, ...rest] = ["foo"];  
first // "foo"  
rest // []
```

如果将扩展运算符用于数组赋值，只能放在参数的最后一位，否则会报错。

```
const [...butLast, last] = [1, 2, 3, 4, 5];  
// 报错  
  
const [first, ...middle, last] = [1, 2, 3, 4, 5];  
// 报错
```

(3) 函数的返回值

JavaScript的函数只能返回一个值，如果需要返回多个值，只能返回数组或对象。扩展运算符提供了解决这个问题的一种变通方法。

```
var dateFields = readDateFields(database);  
var d = new Date(...dateFields);
```

上面代码从数据库取出一行数据，通过扩展运算符，直接将其传入构造函数 `Date`。

(4) 字符串

扩展运算符还可以将字符串转为真正的数组。

```
[...'hello']  
// [ "h", "e", "l", "l", "o" ]
```

上面的写法，有一个重要的好处，那就是能够正确识别32位的Unicode字符。

```
'x\uD83D\uDE80y'.length // 4  
[...'x\uD83D\uDE80y'].length // 3
```

上面代码的第一种写法，JavaScript会将32位Unicode字符，识别为2个字符，采用扩展运算符就没有这个问题。因此，正确返回字符串长度的函数，可以像下面这样写。

```
function length(str) {  
  return [...str].length;  
}  
  
length('x\uD83D\uDE80y') // 3
```

凡是涉及到操作32位Unicode字符的函数，都有这个问题。因此，最好都用扩展运算符改写。

```
let str = 'x\uD83D\uDE80y';  
  
str.split('').reverse().join('')  
// 'y\uDE80\uD83Dx'  
  
[...str].reverse().join('')  
// 'y\uD83D\uDE80x'
```

上面代码中，如果不用扩展运算符，字符串的 `reverse` 操作就不正确。

(5) 实现了 **Iterator** 接口的对象

任何 **Iterator** 接口的对象，都可以用扩展运算符转为真正的数组。

```
var nodeList = document.querySelectorAll('div');
var array = [...nodeList];
```

上面代码中，`querySelectorAll`方法返回的是一个`nodeList`对象。它不是数组，而是一个类似数组的对象。这时，扩展运算符可以将其转为真正的数组，原因就在于`NodeList`对象实现了`Iterator`接口。

对于那些没有部署`Iterator`接口的类似数组的对象，扩展运算符就无法将其转为真正的数组。

```
let arrayLike = {
  '0': 'a',
  '1': 'b',
  '2': 'c',
  length: 3
};

// TypeError: Cannot spread non-iterable object.
let arr = [...arrayLike];
```

上面代码中，`arrayLike`是一个类似数组的对象，但是没有部署`Iterator`接口，扩展运算符就会报错。这时，可以改为使用`Array.from`方法将`arrayLike`转为真正的数组。

（6）Map和Set结构，Generator函数

扩展运算符内部调用的是数据结构的`Iterator`接口，因此只要具有`Iterator`接口的对象，都可以使用扩展运算符，比如Map结构。

```
let map = new Map([
```

```
[1, 'one'],  
[2, 'two'],  
[3, 'three'],  
]);  
  
let arr = [...map.keys()]; // [1, 2, 3]
```

Generator函数运行后，返回一个遍历器对象，因此也可以使用扩展运算符。

```
var go = function*() {  
  yield 1;  
  yield 2;  
  yield 3;  
};  
  
[...go()] // [1, 2, 3]
```

上面代码中，变量 `go` 是一个Generator函数，执行后返回的是一个遍历器对象，对这个遍历器对象执行扩展运算符，就会将内部遍历得到的值，转为一个数组。

如果对没有 `iterator` 接口的对象，使用扩展运算符，将会报错。

```
var obj = {a: 1, b: 2};  
let arr = [...obj]; // TypeError: Cannot spread non-iterable object
```

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 func1 = function () {};  
  
// ES5  
func1.name // ""  
  
// ES6  
func1.name // "func1"
```

上面代码中，变量 `func1` 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;
```

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。

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

箭头函数可以与变量解构结合使用。

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

// 等同于
function full(person) {
```

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

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。

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))
```

箭头函数还有一个功能，就是可以很方便地改写 λ 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 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。

6. 函数绑定

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ES7提出了“函数绑定”(`function bind`)运算符，用来取代 `call`、`apply`、`bind` 调用。虽然该语法还是ES7的一个提案，但是Babel转码器已经支持。

函数绑定运算符是并排的两个双冒号 (`::`)，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);
```

由于双冒号运算符返回的还是原对象，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");
```

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

如果是非尾递归的fibonacci 递归方法

```
function Fibonacci (n) {  
  if ( n <= 1 ) {return 1};  
  
  return Fibonacci(n - 1) + Fibonacci(n - 2);  
}
```

```
Fibonacci(10); // 89
// Fibonacci(100)
// Fibonacci(500)
// 堆栈溢出了
```

如果我们使用尾递归优化过的fibonacci 递归算法

```
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也是如此，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，在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1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。

8. 函数参数的尾逗号

ES7有一个提案，允许函数的最后一个参数有尾逗号（trailing comma）。

目前，函数定义和调用时，都不允许有参数的尾逗号。

```
function clownsEverywhere(  
    param1,  
    param2  
) { /* ... */ }  
  
clownsEverywhere(  
    param1,  
    param2,  
    // trailing comma  
)
```



```
'foo',  
'bar'  
);
```

如果以后要在函数的定义之中添加参数，就势必还要添加一个逗号。这对版本管理系统来说，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，因此新提案允许定义和调用时，尾部直接有一个逗号。

```
function clownsEverywhere(  
  param1,  
  param2,  
) { /* ... */ }  
  
clownsEverywhere(  
  'foo',  
  'bar',  
);
```

留言

上一章

下一章