函数的扩展

目录

- 函数参数的默认值
  - 基本用法
  - 与解构赋值默认值结合使用
  - 参数默认值的位置
  - 函数的 length 属性
  - 作用域
  - 应用
- rest 参数
- 严格模式
- name 属性
- 箭头函数
  - 基本用法
  - 使用注意点
  - 不适用场合
  - 嵌套的箭头函数
- 尾调用优化
  - 什么是尾调用？
  - 尾调用优化
  - 尾递归
  - 递归函数的改写
  - 严格模式
  - 尾递归优化的实现
- 函数参数的尾逗号
- Function.prototype.toString()
- catch 命令的参数省略

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，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 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象的解构赋值默认值，没有使用函数参数的默认值。只有当函数 `foo()` 的参数是一个对象时，变量 `x` 和 `y` 才会通过解构赋值生成。如果函数 `foo()` 调用时没提供参数，变量 `x` 和 `y` 就不会生成，从而报错。通过提供函数参数的默认值，就可以避免这种情况。

```
function foo({x, y = 5} = {}) {
  console.log(x, y);
}

foo() // undefined 5
```

上面代码指定，如果没有提供参数，函数 `foo` 的参数默认为一个空对象。

下面是另一个解构赋值默认值的例子。

```
function fetch(url, { body = '', method = 'GET', headers = {} }) {
  console.log(method);
}

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

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

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

```
function fetch(url, { body = '', method = 'GET', headers = {} } = {}) {
  console.log(method);
}

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

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

注意，函数参数的默认值生效以后，参数解构赋值依然会进行。

```
function f({ a, b = 'world' } = { a: 'hello' }) {
  console.log(b);
}

f() // world
```

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

作为练习，大家可以思考一下，下面两种函数写法有什么差别？

```
// 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];
}

// 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) // [illegible]
f(1, undefined, 2) // [1, 5, 2]
```

[illegible] `undefined` [illegible]

[illegible] `undefined` [illegible] `null` [illegible]

```
function foo(x = 5, y = 6) {
  console.log(x, y);
}

foo(undefined, null)
// 5 null
```

[illegible] `x` [illegible] `undefined` [illegible] `y` [illegible] `null` [illegible]

[illegible] length [illegible]

[illegible] `length` [illegible] `length` [illegible]

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

[illegible] `length` [illegible] 3 [illegible] `c` [illegible] `length` [illegible] `3` [illegible] `1` [illegible] `2` [illegible]

[illegible] `length` [illegible] rest [illegible] `length` [illegible]

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

[illegible] `length` [illegible]

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

[illegible]

[illegible] context [illegible]

```
var x = 1;


function f(x, y = x) {
  console.log(y);
}


f(2) // 2
```

[illegible] `y` [illegible] `x` [illegible] `f` [illegible] `x` [illegible] `x` [illegible] `x` [illegible] `2` [illegible]

[illegible]

```
let x = 1;


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


f() // 1
```

[illegible] `f` [illegible] `y = x` [illegible] `x` [illegible] `x` [illegible] `x` [illegible] `x` [illegible]

[illegible] `x` [illegible]

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


f() // ReferenceError: x is not defined
```

[illegible]

```
var x = 1;

function foo(x = x) {
  // ...
}

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

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

如果参数的默认值是一个函数，该函数的作用域也遵守这个规则。请看下面的例子。

```
let foo = 'outer';

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

bar(); // outer
```

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

如果写成下面这样，就会报错。

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

bar() // ReferenceError: foo is not defined
```

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

下面是一个更复杂的例子。

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

```
  y();
  console.log(x);
}

foo() // 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) { ··· }
```

rest 参数

ES6 引入 rest 参数（形式为 `...变量名`），用于获取函数的多余参数，这样就不需要使用 `arguments` 对象了。rest 参数搭配的变量是一个数组，该变量将多余的参数放入数组中。

```
function add(...values) {
  let sum = 0;

  for (var val of values) {
    sum += val;
  }

  return sum;
}

add(2, 5, 3) // 10
```

上面代码的 `add` 函数是一个求和函数，利用 rest 参数，可以向该函数传入任意数目的参数。

下面是一个 rest 参数代替 `arguments` 变量的例子。

```
// arguments变量的写法
function sortNumbers() {
  return Array.from(arguments).sort();
}

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

上面代码的两种写法，比较后可以发现，rest 参数的写法更自然也更简洁。

`arguments` Array.from
rest
rest
`push`

```
function push(array, ...items) {
  items.forEach(function(item) {
    array.push(item);
    console.log(item);
  });
}

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

rest

```
// 
function f(a, ...b, c) {
  // ...
}
```

`length` rest

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


ES5

```
function doSomething(a, b) {
  'use strict';
  // code
}
```

ES2016

```
// 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。

[illegible]
[illegible]

[illegible]

```
'use strict';

function doSomething(a, b = a) {
  // code
}
```

[illegible]

```
const doSomething = (function () {
  'use strict';
  return function(value = 42) {
    return value;
  };
}());
```

name [illegible]

[illegible] `name` [illegible]

```
function foo() {}
foo.name // "foo"
```

[illegible] ES6[illegible]

[illegible] ES6 [illegible] ES5 [illegible] `name` [illegible] ES6 [illegible] `name` [illegible]

```
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 "
```

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;
// [illegible]
var f = function () { return 5 };

var sum = (num1, num2) => num1 + num2;
// [illegible]
var sum = function(num1, num2) {
  return num1 + num2;
};
```

[illegible] `return` [illegible]

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

[illegible]

```
// [illegible]
let getTempItem = id => { id: id, name: "Temp" };

// [illegible]
let getTempItem = id => ({ id: id, name: "Temp" });
```

[illegible]

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

[illegible] `{ a: 1 }` [illegible] `a: 1` [illegible] `a` [illegible] `1;` [illegible]

[illegible]

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

[illegible]

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

// [illegible]
function full(person) {
  return person.first + ' ' + person.last;
}
```

箭头函数使得表达更加简洁。

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

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

箭头函数的一个用处是简化回调函数。

```
// 普通函数写法
[1,2,3].map(function (x) {
  return x * x;
});

// 箭头函数写法
[1,2,3].map(x => x * x);
```

另一个例子是

```
// 普通函数写法
var result = values.sort(function (a, b) {
  return a - b;
});

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

下面是 rest 参数与箭头函数结合的例子。

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

numbers(1, 2, 3, 4, 5)
// [1,2,3,4,5]

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

headAndTail(1, 2, 3, 4, 5)
// [1,[2,3,4,5]]
```

使用注意点

箭头函数有几个使用注意点。

（1）箭头函数没有自己的 `this` 对象（详见下文）。

（2）不可以当作构造函数，也就是说，不可以对箭头函数使用 `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，最重要的是第一点。对于普通函数来说，内部的 `this` 指向函数运行时所在的对象，但是这一点对箭头函数不成立。它没有自己的 `this` 对象，内部的 `this` 就是定义时上层作用域中的 `this`。也就是说，箭头函数内部的 `this` 指向是固定的，相比之下，普通函数的 `this` 指向是可变的。

```
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}`），所以打印出来的是 `42`。

下面例子是回调函数分别为箭头函数和普通函数，对比它们内部的 `this` 指向。

```
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。

箭头函数实际上可以让`this`指向固定化，绑定`this`使得它不再可变，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。如果回调函数是普通函数，那么运行`this.doSomething()`这一行会报错，因为此时`this`指向`document`对象。

总之，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。

下面是 Babel 转箭头函数产生的 ES5 代码，就能清楚地说明`this`的指向。

```
// 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，`this`的指向有几个？

```
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
```

答案是`this`的指向只有一个，就是函数`foo`的`this`，这是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`。所以不管怎么嵌套，`t1`、`t2`、`t3`都输出同样的结果。如果这个例子的所有内层函数都写成普通函数，那么每个函数的`this`都指向运行时所在的不同对象。

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。

第一个场合是定义对象的方法，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。这是因为对象不构成单独的作用域，导致 `jumps`

箭头函数定义时的作用域就是全局作用域。

再看一个例子。

```
globalThis.s = 21;

const obj = {
  s: 42,
  m: () => console.log(this.s)
};

obj.m() // 21
```

上面例子中，`obj.m()`使用箭头函数定义。JavaScript引擎的处理方法是，先在全局空间生成这个箭头函数，然后赋值给`obj.m`，这导致箭头函数内部的`this`指向全局对象，所以`obj.m()`输出的是全局空间的`21`，而不是对象内部的`42`。上面的代码实际上等同于下面的代码。

```
globalThis.s = 21;
globalThis.m = () => console.log(this.s);

const obj = {
  s: 42,
  m: globalThis.m
};

obj.m() // 21
```

由于上面这个原因，对象的属性建议使用传统的写法定义，不要用箭头函数定义。

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
```

[illegible] λ [illegible]

```
// λ[illegible]
fix = λf.(λx.f(λv.x(x)(v)))(λx.f(λv.x(x)(v)))

// ES6[illegible]
var fix = f => (x => f(v => x(x)(v)))
               (x => f(v => x(x)(v)));
```

[illegible] λ [illegible] ES6 [illegible]

[illegible]

[illegible]

[illegible] Tail Call [illegible]

```
function f(x){
  return g(x);
}
```

[illegible] `f` [illegible] `g` [illegible]

[illegible]

```
// [illegible]
function f(x){
  let y = g(x);
  return y;
}

// [illegible]
```

```
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`。

注意，目前只有 Safari 浏览器支持尾调用优化，Chrome 和 Firefox 都不支持。

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) // 超时
Fibonacci(500) // 超时
```

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 亦是如此，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（或者层层递归造成的超时），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',
);
```

[illegible]

Function.prototype.toString()

ES2019 [illegible] `toString()` [illegible]

`toString()` [illegible]

```
function /* foo comment */ foo () {}

foo.toString()
// function foo() {}
```

[illegible] `foo` [illegible] `foo` [illegible] `toString()` [illegible]

[illegible] `toString()` [illegible]

```
function /* foo comment */ foo () {}

foo.toString()
// "function /* foo comment */ foo () {}"
```

catch [illegible]

JavaScript [illegible] `try...catch` [illegible] `catch` [illegible] `try` [illegible]

```
try {
  // ...
} catch (err) {
  // [illegible]
}
```

上面代码中，`catch`命令后面带有参数`err`。

很多时候，`catch`代码块可能用不到这个参数。但是，为了保证语法正确，还是必须写。[ES2019](#)做出了改变，允许`catch`语句省略参数。

```
try {
  // ...
} catch {
  // ...
}
```

版本号 #2
由 壳壳虫 创建于 2020-04-16 10:29:52 UTC
由 壳壳虫 更新于 2022-04-09 20:01:36 UTC