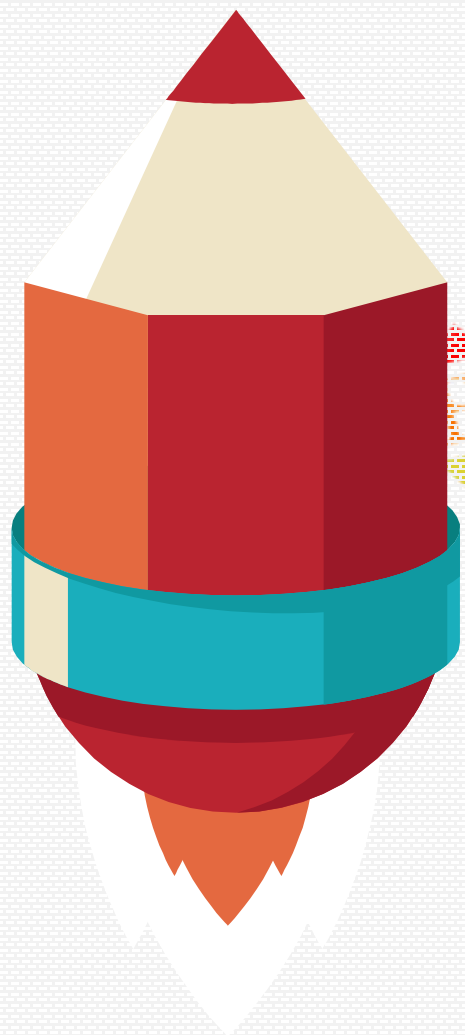




# 第29课：ES6进阶之

## 函数类型拓展及数组拓展

主讲老师：万章



## 四知

什么是函数的尾调用

函数的尾调用优化

数组的扩展运算符



# 什么是函数的尾调用

## 函数的尾调用优化

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返回调用另一个函数的执行结果。

```
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。

```
// 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);  
}
```

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



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。

情况三等同于右面的代码

## 函数的尾调用优化

```
function f(x) {  
  if (x > 0) {  
    return m(x)  
  }  
  return n(x);  
}
```

尾调用不一定出现在函数尾部，只要是最后一步操作即可。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。



# 函数的尾调用优化

## 函数尾调用的内存优化

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）。

### 全局作用域(变量区)

函数作用域(c)



函数作用域(b)



函数作用域(a)

## 函数尾调用的内存优化

```
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);
```

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。

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。

**简单来说**就是变量数据的保存周期，如果是尾调用的话，需要哪些数据不需要哪些已经是很明白的了，那么不需要用的那些就可以直接把它清除掉，以减少内存的消耗



## 函数尾调用的内存优化

```
function addOne(a){  
  var one = 1;  
  function inner(b){  
    return b + one;  
  }  
  return inner(a);  
}
```

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”。

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。

## 函数尾调用的复杂度优化

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（需要保存上一次函数的参数，相当于 $n$ 个闭包），复杂度  $O(n)$

```
function factorial(n, total) {  
  if (n === 1) return total;  
  return factorial(n - 1, n * total);  
}
```

```
factorial(5, 1) // 120
```

如果改写成尾递归，只保留一个调用记录，复杂度  $O(1)$ 。

## 函数尾调用的复杂度优化

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**

```
function factorial(n, total) {  
  if (n === 1) return total;  
  return factorial(n - 1, n * total);  
}  
  
factorial(5, 1) // 120
```

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？

## 函数尾调用的复杂度优化

```
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
```

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 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
```

或是使用柯里化进行函数改造

方法一无论用啥感觉都特别麻烦(╯\_╰;) )

## 函数尾调用的复杂度优化

```
function factorial(n, total = 1) {  
  if (n === 1) return total;  
  return factorial(n - 1, n * total);  
}  
  
factorial(5) // 120
```

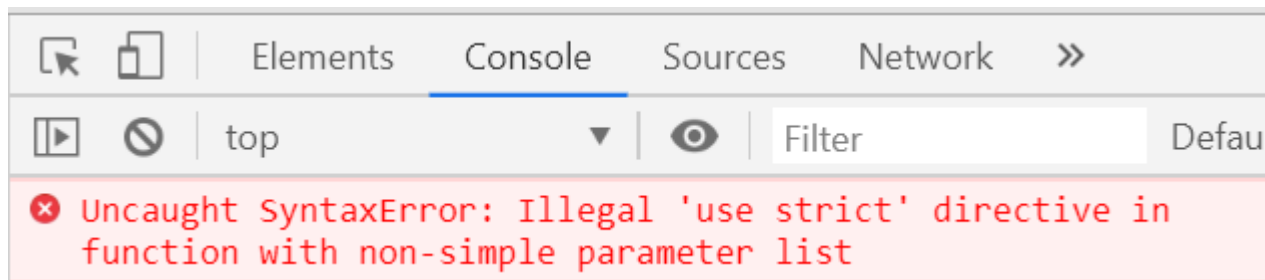
第二种方法就简单多了，就是采用 ES6 的函数默认值，瞬间搞定

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。

## 函数尾调用的开启条件

```
> function factorial(n, total = 1) {  
  console.log(factorial.caller);  
  if (n === 1) return total;  
  return factorial(n - 1, n * total);  
}  
  
factorial(5) // 120  
  
null  
  
f factorial(n, total = 1) {  
  console.log(factorial.caller);  
  if (n === 1) return total;  
  return factorial(n - 1, n * total);  
}  
  
f factorial(n, total = 1) {  
  console.log(factorial.caller);  
  if (n === 1) return total;  
  return factorial(n - 1, n * total);  
}  
  
f factorial(n, total = 1) {  
  console.log(factorial.caller);  
  if (n === 1) return total;  
  return factorial(n - 1, n * total);  
}  
  
f factorial(n, total = 1) {  
  console.log(factorial.caller);  
  if (n === 1) return total;  
  return factorial(n - 1, n * total);  
}  
  
< 120
```

```
function factorial(n, total = 1) {  
  'use strict';  
  console.log(factorial.caller);  
  if (n === 1) return total;  
  return factorial(n - 1, n * total);  
}
```



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

1. `func.arguments`: 返回调用时函数的参数。
2. `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。

## 函数尾调用的开启条件模拟

```
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(...)
```

sum递归 100000 次，报错，提示超出调用栈的最大次数

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。

## 函数尾调用的开启条件模拟

```
function trampoline(f) {  
  while (f && f instanceof Function) {  
    f = f();  
  }  
  return f;  
}
```

蹦床函数 (trampoline) 可以将递归执行转为循环执行。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。

```
trampoline(sum(1, 100000))  
// 100001
```

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。就是采用“循环”换掉“递归”。

```
function sum(x, y) {  
  if (y > 0) {  
    return sum.bind(null, x + 1, y - 1);  
  } else {  
    return x;  
  }  
}
```

然后将原来的递归函数，改写为每一步返回另一个函数。

`sum` 函数的每次执行，都会返回自身的另一个版本。



## 函数尾调用的终极优化版

右侧代码中，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。

```
function tco(f) {
  var value;
  var active = false;
  var accumulated = []; // 闭包存储每次传输进来的数据

  return function accumulator() {
    accumulated.push(arguments); // 把每次传进来的参数数组预存在这个数组里面
    // console.log(accumulated.length);
    if (!active) {
      active = true;
      console.log(2);
      while (accumulated.length) {
        value = f.apply(this, accumulated.shift()); // accumulated.shift() === arguments
        console.log(value);
      }
      active = false;
      return value;
    }
  };
}

var sum = tco(function(x, y) {
  if (y > 0) {
    return sum(x + 1, y - 1);
  }
  else {
    return x;
  }
});

sum(1, 10); // 11
```



# 数组的拓展运算符

## 数组的拓展之扩展运算符

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>]
```

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

该运算符主要用于函数调用。

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。

## 数组的拓展之扩展运算符

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

扩展运算符与正常的函数参数可以结合使用，非常灵活。

```
[...[], 1]  
// [1]
```

如果扩展运算符后面是一个空数组，则不产生任何效果。

```
const arr = [  
  ...(x > 0 ? ['a'] : []),  
  'b',  
];
```

扩展运算符后面还可以放置表达式。

```
(...[1, 2])  
// Uncaught SyntaxError: Unexpected number  
  
console.log(...[1, 2])  
// Uncaught SyntaxError: Unexpected number  
  
console.log(...[1, 2])  
// 1 2
```

注意，只有函数调用时，扩展运算符才可以放在圆括号中，否则会报错。

## 数组的拓展之扩展运算符的应用场景

```
// 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) {
  // ...
}
let args = [0, 1, 2];
f(...args);
```

### (1)数组传参

```
// 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);
```

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。

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

通过push方法，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
let arr1 = [0, 1, 2];
let arr2 = [3, 4, 5];
arr1.push(...arr2);
```

## 数组的拓展之扩展运算符的应用场景

### (2)复制数组

```
const a1 = [1, 2];  
const a2 = a1;  
  
a2[0] = 2;  
a1 // [2, 2]
```

a2并不是a1的克隆，而是指向同一份数据的另一个指针。修改a2，会直接导致a1的变化。

```
const a1 = [1, 2];  
const a2 = a1.concat();  
  
a2[0] = 2;  
a1 // [1, 2]
```

上面代码中，a1会返回原数组的克隆，再修改a2就不会对a1产生影响。

```
const a1 = [1, 2];  
  
const a2 = [...a1];
```

扩展运算符提供了复制数组的简便写法。

数组是引用类型的数据类型，直接复制的话，只是复制了指向底层数据结构的指针，而不是克隆一个全新的数组。

## 数组的拓展之扩展运算符

### (3)合并数组

```
const arr1 = ['a', 'b'];
const arr2 = ['c'];
const 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' ]
```

```
const a1 = [{ foo: 1 }];
const a2 = [{ bar: 2 }];

const a3 = a1.concat(a2);
const a4 = [...a1, ...a2];
```

```
> console.log(a3);
▼ (2) [{...}, {...}] ⓘ
  ▶ 0: {foo: 1}
  ▶ 1: {bar: 2}
  length: 2
  ▶ __proto__: Array(0)
< undefined
>
  console.log(a4);
▼ (2) [{...}, {...}] ⓘ
  ▶ 0: {foo: 1}
  ▶ 1: {bar: 2}
  length: 2
  ▶ __proto__: Array(0)
< undefined
> a3[0].foo=3
< 3
> a3
< ▼ (2) [{...}, {...}] ⓘ
  ▶ 0: {foo: 3}
  ▶ 1: {bar: 2}
  length: 2
  ▶ __proto__: Array(0)
> a4
< ▼ (2) [{...}, {...}] ⓘ
  ▶ 0: {foo: 3}
  ▶ 1: {bar: 2}
  length: 2
  ▶ __proto__: Array(0)
>
```

扩展运算符提供了数组合并的新写法

注意一点：如果数组里面的项目有对象的，那么这个对象的复制是属于浅复制，也就是说，只会复制对象的地

## 数组的拓展之扩展运算符

### (4)结构赋值

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

```
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   // []
```

扩展运算符可以与解构赋值结合起来，用于生成数组。

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



## 数组的拓展之扩展运算符

### (5) 字符串转数组

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

### (6) 类数组转真数组

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

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(后面会讲的)

对于那些没有部署Iterator接口的类似数组的对象，扩展运算符

```
> let a=document.querySelectorAll("div");  
<> undefined  
  
> a  
<> NodeList(11) [div#sidebar, div#content, div#disqus_thread, div#back_to_top, div#edit, div#loading, div#error, div#flip, div#pageup, div#pagedown, div.progress-indicator-2]   
  ▶ 0: div#sidebar  
  ▶ 1: div#content  
  ▶ 2: div#disqus_thread  
  ▶ 3: div#back_to_top  
  ▶ 4: div#edit  
  ▶ 5: div#loading  
  ▶ 6: div#error  
  ▶ 7: div#flip  
  ▶ 8: div#pageup  
  ▶ 9: div#pagedown  
  ▶ 10: div.progress-indicator-2  
  length: 11  
  __proto__: NodeList  
    ▶ entries: f entries()  
    ▶ forEach: f forEach()  
    ▶ item: f item()  
    ▶ keys: f keys()  
    length: (...)  
    ▶ values: f values()  
    ▶ constructor: f NodeList()  
    ▶ Symbol(Symbol.iterator): f values()  
    Symbol(Symbol.toStringTag): "NodeList"  
    ▶ get length: f length()  
    ▶ __proto__: Object
```

