

## 09 | (...x): 不是表达式、语句、函数，但它却能执行

2019-12-02 周爱民

《JavaScript核心原理解析》

课程介绍 >



讲述：周爱民

时长 20:26 大小 18.72M



你好，我是周爱民，欢迎回到我的专栏。

从之前的课程中，你应该已经对语句执行和函数执行有了基本的了解。事实上，这两种执行其实都是对**顺序**、**分支**与**循环**三种逻辑在语义上的表达。

也就是说，不论一门语言的语法有什么特异之处，它对“执行逻辑”都可以归纳到这三种语义的表达方式上来。这种说法事实上也并不特别严谨，因为这三种基本逻辑还存在进一步抽象的空间（这些也会是我将来会讨论到的内容，今天暂且不论）。

今天这一讲中，主要讨论的是第二种执行的一些细节，也就是对“函数执行”的进一步补充。

在上一讲中，我有意将函数分成三个语义组件来讲述。我相信在绝大多数的情况下，或者在绝大多数的语言教学中，都是不必要这样做的。这三个语义组件分别是指参数、执行体和结果，将它们分开来讨论，最主要的价值就在于：**通过改造这三个语义组件的不同部分，我们可以得**



到不同的“函数式的”执行特征与效果。换言之，可以通过更显式的、特指的或与应用概念更贴合的语法来表达新的语义。与所谓“特殊可执行结构”一样，这些语义也用于映射某种固定的、确定的逻辑。

语言的设计，本质就是为确定的语义赋以恰当的语法表达。

## 递归与迭代

如果循环是一种确定的语义，那么如何在函数执行中为它设计合适的语法表达呢？

递归绝对是一个好的、经典的求解思路。递归将循环的次数直接映射成函数“执行体”的重复次数，将循环条件放在函数的参数界面中，并通过函数调用过程中的值运算来传递循环次数之间的数值变化。递归作为语义概念简单而自然，唯一与函数执行存在（潜在的）冲突的只是所谓栈的回收问题，亦即是尾递归的处理技巧等，但这些都是实现层面的要求，而与语言设计无关。

由于递归并不改变函数的三个语义组件中的任何一个，因此它与函数执行过程完全没有冲突，也没有任何新的需求与设计。这句话的潜在意思是说，函数的三个语义组件都不需要为此作出任何的设计修改，例如：

```
1  const f = x => x && f(--x);
```

 复制代码

在这段代码中，是没有出现任何特殊的语法和运算 / 操作符的，它只是对函数、（变量或常量的）声明、表达式以及函数调用等等的简单组合。

然而迭代不是。迭代也是循环语义的一种实现，它说明循环是“函数体”的重复执行，而不是“递归”所理解的“函数调用自己”的语义。这是一种可受用户代码控制的循环体。你可以尝试创建这样一个简单的迭代函数：

```
1  // 迭代函数
2  function foo(x = 5) {
3    return {
4      next: () => {
5        return {done: !x, value: x && x--};
6      }
}
```

 复制代码



```
7   }  
8 }
```

然而请仔细观察这样的两个实现，你需要注意在这个迭代函数中有“值 (value) 和状态 (done)”两个控制变量，并且它的实际执行代码与上面的函数 `f()` 是一样的：

[复制代码](#)

```
1 // in 函数f()  
2 x && f(--x)  
3  
4 // in 迭代foo()  
5 x && x--
```

也就是说，递归函数“`f()`”和迭代函数“`foo()`”其实是在实现相同的过程。只是由于“递归完成与循环过程的结束”在这里是相同的语义，因此函数“`f()`”中不需要像迭代函数那样来处理“状态 (done)”的传出。递归函数“`f()`”，要么结束，要么无穷递归。

## 迭代对执行过程的重造和使用

在 JavaScript 中，是通过一个中间对象来使用迭代过程 `_foo()` 的。该中间对象称为迭代器，`foo()` 称为迭代器函数，用于返回该迭代器。例如：

[复制代码](#)

```
1 var tor = foo(); // default `x` is 5  
2 ...
```

迭代器具有 `.next()` 方法用于一次（或每次）迭代调用。由于没有约定迭代调用的方式，因此可以用任何过程来调用它。例如：

[复制代码](#)

```
1 // 在循环语句中处理迭代调用  
2 var tor = foo(5), result = tor.next();  
3 while (!result.done) {  
4   console.log(result.value);  
5   result = tor.next();  
6 }
```



除了一些简单的、概念名词上的置换外，这些与你所见过的绝大多数有关“迭代器与生成器”的介绍并没有什么不同。并且你也应当理解，正是这个“`.next()`”调用的界面维护了迭代过程的上下文，以及值之间的相关性（例如一个值序列的连续性）。

根据约定，如果有一个对象“包含”这样一个迭代器函数（以返回一个迭代器），那么这个对象就是可迭代的。基于 JavaScript 中“对象是属性集（所以所有包含的东西都必然是属性）”的概念，这个迭代函数被设计为称为“`Symbol.iterator`”的符号属性。例如：

[复制代码](#)

```
1 let x = new Object;
2 x[Symbol.iterator] = foo; // default `x` is 5
```

现在，你可以使用这个可迭代对象了：

[复制代码](#)

```
1 > console.log(...x);
2 5 4 3 2 1
```

现在，你看到了这一讲标题中的代码：

[复制代码](#)

```
1 (...x)
```

不过，不同的是，标题中的代码是不能执行的。

## 展开语法

问题的关键点在于：`...x`是什么？

在形式上，“`...`”看起来像是一个运算符，而`x`是它的操作数。但是，如果稍微深入地问一下这个问题，就会令人生疑了。例如：如果它是运算符，那么运算的返回值是什么？



答案是，它既不返回值，也不返回引用。

那么如果它不是运算符，或者说...x也并不是表达式，或许它们可以被理解为“语句”吗？即使如此，与上面相同的问题也会存在。例如：如果它是语句，那么该语句的返回值是什么？

答案是，既不是空（Empty），也不是其它结果（Result）。因此它也不是语句（并且，因为 `console.log()` 是表达式，而表达式显然也“不可能包含语句”）。

所以，...x既不是表达式，也不是语句。它不是我们之前讲过的任何一种概念，而仅仅只是“语法”。作为语法，ECMAScript 在这里规定它只是对一个确定的语义的封装。

在语义上，它用于“展开一个可迭代对象”。

## 如何做到呢？

为什么我要绕这么大个圈子来介绍这个“简单的”展开语法呢？又或者说，ECMAScript 为什么要弄出这么一个“新”概念呢？

这与函数的第三个语义组件——“值”是有关的。在 JavaScript 中（也包括在绝大多数支持函数的语言中），函数只能返回一个值。然而，如果迭代器表达的是一个重复执行的执行体，并且每次执行都返回一个值，那么又怎么可能用“返回一个值”的函数来返回呢？

与此类似，语句也只有一个这样的单值返回，所以批语句执行也仍然只是返回最后一行的结果。并且，一旦...x被理解为语句，那么它就不能用作操作数，成为一个表达式的部分。这在概念上是不容许的。所以，当在“函数”这个级别表达多次调用时，尽管它可以通过“对象（迭代对象）”来做形式上的封装，却无法有效地表达“多次调用的多个结果”。这才是展开语法被设计出来的原因。

如果可迭代对象表达的是“多个值”，那么可以作用于它的操作或运算通常应该是那些面向“值的集合（Collections）”的。更确切地说，它是可以面向“索引集合（Indexed Collections）”和“键值集合（Keyed Collections）”设计的语法概念。因此在现在的，以及将来的 ECMAScript 规范中，你将会看到它的操作，例如通常包括的合并、映射、筛选等等，将在包括对象、数组、集（Set）、图（Map）等等数据的处理中大放异彩。

而现在，其实我想问的问题是，在函数中是如何做到迭代处理的呢？

## 内部迭代过程



迭代的本质是多次函数调用，在 JavaScript 内部实现这一机制，本质上就是管理这些多次调用之间的关系。这显然包括一个循环过程，和至少一个循环控制变量。

这个迭代有一个开启过程，简单的如展开语法（“...”），复杂的如 for...of 语句。这些语法 / 语法结构通过类似如下两个步骤来完成迭代的开启：

 复制代码

```
1 var tor = foo(5), result = tor.next();
2 while (!result.done) ...
```

但是如同我在之前的课程，以及上面的讨论中一再强调的这是“一个执行过程”，既然是过程，那么就存在过程被中断的可能。简单的示例如下：

 复制代码

```
1 while (!result.done) {
2     break;
3 }
```

是的，这个过程什么也不会发生。如果是在经典的 while 循环里面，那么它的 result 和 tor，以及 foo() 调用所开启的那个函数闭包都被当前上下文管理或回收。然而，如果在一个展开过程，或者 for...of 循环中，相应的“语法”管理上述这些组件的时候又需要怎样的处理呢？例如：

 复制代码

```
1 function touch(x) {
2     if (x==2) throw new Error("hard break");
3 }
4
5 // 迭代函数
6 function foo2(x = 5) {
7     return {
8         next: () => {
9             touch(x); // some process methods
10            return {done: !x, value: x && x--};
11        }
12    }
13 }
14
15 // 示例
16 let x = new Object;
```



```
17 x[Symbol.iterator] = foo2; // default `x` is
```

测试如下：

 复制代码

```
1 > console.log(...x);  
2 Error: hard break
```

这个示例是一个简单异常，但如果这个异常发生于 for...of 中：

 复制代码

```
1 > for (let i of x) console.log(i);  
2 5  
3 4  
4 3  
5 Error: hard break
```

在这两种示例中，异常都是发生于 foo2() 这个函数调用的一个外部处理过程中，而等到用户代码有机会操作时，已经处于 console.log() 调用或 for...of 循环中了，如果用户在这里设计异常处理过程，那么 foo2() 中的 touch(x) 管理和涉及的资源都无法处理。因此，ECMAScript 设计了另外两个方法来确保 foo2() 中的代码在“多次调用”中仍然是受控的。这包括两个回调方法：

tor.return(), 当迭代正常过程退出时回调

tor.throw(), 当迭代过程异常退出时回调

这并不难于证实：

 复制代码

```
1 > Object.getOwnPropertyNames(tor.constructor.prototype)  
2 [ 'constructor', 'next', 'return', 'throw' ]
```

现在如果给 tor 的 return 属性加一个回调函数，会发生什么呢？

 复制代码




```

1 // 迭代函数
2 function foo2(x = 5) {
3   return {
4     // 每次.next()都不会返回done状态，因此可列举无穷次
5     "next": () => new Object, // result instance, etc.
6     "return": () => console.log("RETURN!")
7   }
8 }
9 let x = new Object;
10 x[Symbol.iterator] = foo2; // default `x` is 5

```

测试一下：

 复制代码

```

1 # 列举x，第一次迭代后即执行break;
2 > for (let i of x) break;
3 RETURN!

```

结果是RETURN!?

什么鬼？！

## 异常处理

并且如果你试图在 `for.throw` 中去响应 `foo()` 迭代中的异常，却什么也得不到。例如：

 复制代码

```

1 // 迭代函数
2 function foo3(x = 5) {
3   return {
4     // 第一个.next()执行时即发生异常
5     "next": () => { throw new Error },
6     "throw": () => console.log("THROW!")
7   }
8 }
9 let x = new Object;
10 x[Symbol.iterator] = foo3;

```



在测试中，异常直接被抛给了全局：



```
1 > console.log(...x);
2 Error
3     at Object.next (repl:4:27)
```

[复制代码](#)

继续！显然可以把这个例子跟最开始使用的 `foo()` 组合起来，`foo()` 迭代可以正确地得到 5 4 3 2 1，而上面的 `return/throw` 可以捕获过程的退出或异常。例如：

```
1 // 迭代函数
2 function foo4(x = 5) {
3     return {
4         // foo()中的next
5         next: () => {
6             return {done: !x, value: x && x--};
7         },
8
9
10        // foo2()和foo3()中的return和throw
11        "return": () => console.log("RETURN!"),
12        "throw": () => console.log("THROW!")
13    }
14 }
15
16
17 let x = new Object;
18 x[Symbol.iterator] = foo4
```

[复制代码](#)

测试：

```
1 >>console.log(...x);
2 5 4 3 2 1
```

[复制代码](#)

Ok，成功是成功了！但是，“RETURN/THROW”呢？

这里简直就是迭代的地狱！

## 是谁的退出与异常？



回顾之前的内容，迭代过程并不是一个语法执行的过程，而是应该理解为一组函数执行的过程；对于这一批函数执行过程中的结束行为，也应该理解为函数内的异常或退出。因此，尽管在 `for...of` 的表面上看，是 `break` 发生了语句中的中止，而在迭代处理的内部发生的，却是“一个迭代过程的退出”。与此同样复杂的是，在这一批函数的多个执行上下文中，不论是在哪儿发生了异常，其实只有外层的第一个能捕获异常的环境能响应这个异常。

简单地说：“退出 (RETURN)”是执行过程的，“异常 (THROW)”是外部的。


JavaScript 中，迭代被处理为两个实现用的组件，一个是（循环的）迭代过程，另一个是（循环的）迭代控制变量。表现在 `for` 这个迭代对象上来看，就是（对于循环来说，）“如果谁使用迭代变量 `for`，那么就是谁管理迭代过程”。

这个“管理循环过程”意味着：

- 如果迭代结束（不论它因为什么结束），那么触发 `for.return` 事件；
- 如果发现异常（只要是当前环境能捕获到的异常），那么触发 `for.throw` 事件。

这两个过程总是发生在“管理循环过程”的行为框架中。例如在下面这个过程中：

```
1 for (let i of x) {  
2   if (i == 2) break;  
3 }
```

 复制代码

由于 `for ... of` 语句将获得 `x` 对象的迭代变量 `for`，那么它也将管理 `x` 对象的迭代过程。因此，在 `for` 语句 `break` 之后（在 `for` 语句将会退出自己的作用之前），它也就必须去“通知”`x` 对象迭代过程也结束了，于是这个语句触发了 `for.return` 事件。


同样，如果是一个数组展开过程：

```
1 console.log(...x);
```

 复制代码



那么将是...x这个“展开语法”来负责上述的迭代过程的管理和“通知”，这个语法在它所在的位置上是无法响应异常的。该语法所在位置是一个表达式，不可能在它内部使用try..catch语句。

 复制代码

```
1 function touch(x) {
2   if (x==2) throw new Error("hard break");
3 }
4
5 // 迭代函数
6 function foo5(x = 5) {
7   return {
8     // foo2()中的next
9     next: () => {
10      touch(x); // some process methods
11      return {done: !x, value: x && x--};
12    },
13
14    // foo3()中的return和throw
15    "return": () => console.log("RETURN!"),
16    "throw": () => console.log("THROW!")
17  }
18 }
19
20 let x = new Object;
21 x[Symbol.iterator] = foo5;
22
23 try {
24   console.log(...x);
25 }
26 catch(e) {} // m
```

这段示例代码将 mute 掉一切：既没有 console.log() 输出，也没有异常信息，tor 的 return/throw 一个也没有发生。

对于 x 这个可迭代对象，以及 foo5() 这个迭代器函数来说，世界是安静的：它既不知道自己发生了什么，也不知道它的外部世界发生了什么。因为...x这个语法既没有管理迭代过程（因此不理解 tor 的退出 /return 行为），也没有在异常发生时向内“通知”tor.throw 事件的能力。



## 知识回顾

标题中的示例是不能执行的，因为其中的括号并不是表达式中分组运算符，也不是语句中的函数调用，也不是声明中的形式参数表。声明中的...x被定义为“展开语法”，是逻辑的映射（它返回的是处理逻辑），而不是“值”或“引用”。它在不同的位置被 JavaScript 解释成不同的语义，包括对象展开和数组展开，并通过一组特定的代码来实现上述的语义。

在...x被理解为数组展开时，本质上是将x视为一个可迭代对象，并通过一个迭代变量（例如tor）来管理它的迭代过程。在 JavaScript 中的迭代对象 x 的生存周期是交由使用它的表达式、语句或语法来管理的，包括在必要的时候通过 tor 来向内通知 return/throw 事件。


在本讲的示例中，展开语法“...x”是没有向内通知的能力的，而“for ... of”可以隐式地向内通知。对于后者，for...of 中的 break 和 continue，以及循环的正常退出都能够通知 return 事件，但它并没有内向通知 throw 的能力，因为 for...of 语句本身并不捕获和处理 throw。

## 思考题

- 既然上面的过程完全不使用 tor.throw，那么它被设计出来做什么？
- ...x为什么称为“展开语法”，为什么 ECMAScript 不提供一个表达式 / 语句之外的概念来指代它？
- continue 在那种情况下触发 tor.return？
- yield\* x 将导致什么？

欢迎你在进行深入思考后，与其他同学分享自己的想法，也让我有机会能听听你的收获。

分享给需要的人，Ta购买本课程，你将得 20 元

 生成海报并分享

 赞 3

 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。



## 学习推荐

# JVM + NIO + Spring

## 各大厂面试题及知识点详解

限时免费 



### 精选留言 (11)

 写留言



leslee

2019-12-26

1. 他可以手动调用..., 在you dont know js 里面说过这个 .throw 可以委托异常.
2. 这是由他的语义决定的, 如果他是一个语句, 那他就不能跟表达式连用, 如果他是一个表达式, 那他的返回值又显得有点多余.
- 3.

```js

```
function foo4(x = 5) {  
  return {  
    next: () => {  
      return { done: !x, value: x && x-- };  
    },  
    "return": () => {  
      return { done: true, value: '恭喜发财' }  
    },  
    "throw": () => {  
      console.log("THROW!")  
    }  
  }  
}
```



```

let x = new Object;
x[Symbol.iterator] = foo4
aaa: for (let item of x) {
  console.log(item, 'w')
  for (let item of x) {
    console.log(item, 'i')
    continue aaa;
  }
}
// echo 'return' x2
// 当return 函数返回undefined 的时候会报这个错 Iterator result undefined is not an object
...

```

被我试出来了, 当return函数返回undefined, 且嵌套循环且continue带有外层循环语句的标签的时候, 他会触发两次return, 缺一个条件都不行. 当return函数返回一个正常的迭代器对象`{done:true, value: 'xxx'}`, 他会输出5个return, 这个return应该由内层的forof 发出, 因为内层的循环直接被打断了, 继续下去的是外层循环, 单层循环不行是因为 continue 的语义并不是打断/结束, 是这样理解么老师, 这里面还有其他的豆子么, 老师.

4. 把x展开, 返回迭代值, 如果 没有 \* 返回的将是迭代器函数, .

作者回复: 关于1和3, 老实说, 这个东西确实灰常灰常麻烦, 你得自己去体会tor.return和tor.throw的含义与用法。

我只能提示说: 谁创建了tor, 那么就该是谁去调用tor.return和tor.throw。这两个方法不是事件触发, 而是“tor的持有者”调用。

关于2, 你的答案没有对或不对的问题, 但我觉得还有深思的余地。不妨多思考一下, 不着急有结论。另外, 这个问题没有标准答案。

关于4, 你“可能”是对的, 但似乎表达得不太清楚。^^.



Smallfly

2020-01-16



// 迭代函数

```

function foo(x = 5) {
  return {

```

```

    next: () => {
      return {done: !x, value: x && x--};
    }
  }
}
}

```

```

const tor = foo();
const names = Object.getOwnPropertyNames(tor.constructor.prototype);

```

```

console.log(names);
/*
[ 'constructor',
  '__defineGetter__',
  '__defineSetter__',
  'hasOwnProperty',
  '__lookupGetter__',
  '__lookupSetter__',
  'isPrototypeOf',
  'propertyIsEnumerable',
  'toString',
  'valueOf',
  '__proto__',
  'toLocaleString' ]
*/

```

请问老师我用 node 执行上面的代码，为什么跟文中的以下输出不一致呢。

```

> Object.getOwnPropertyNames(tor.constructor.prototype)
[ 'constructor', 'next', 'return', 'throw' ]

```

作者回复: 这里的foo()是一个模拟生成器函数的界面的，所以它当然得不到“真的”生成器。

你需要的示例是这样：

```

...

```

```

function* foo() {}

```

```

tor = foo();

```

```

names = Object.getOwnPropertyNames(tor.constructor.prototype);

```

```

...

```



Sam

2021-08-12

根据周老师的指点，谁创建谁调用。以下是我想到一个办法调用throw方法(不知理解是否合理):

```
function* foo() {
  yield 1
  yield 2
  yield 3
}
foo.prototype.throw = function(e) {
  console.log("Test Error " + e)
}
let tor = foo()
for(let item of tor) {
  console.log(item)
  tor.throw(new Error('迭代器循环体内触发'))
}
```

作者回复: 看起来这样是能用的，并且事实上通常也需要这样来使用。但是，总之，在这里throw()的原则与应用都是令人困惑的。——我的意思是js语言在这里的设计原本就很令人困惑。由于篇幅，这一讲并没有关于这个部分的更多讨论，而仅仅是开了个头。如果你有兴趣了解更深的话，建议参阅《javascript语言精髓与编程实践》的5.4.4.3和5.4.5的内容。

共 2 条评论 >



吉法师

2021-04-06

迭代感觉用不着啊.....



油菜

2020-11-13

"用户在这里设计异常处理过程，那么 foo2() 中的 touch(x) 管理和涉及的资源都无法处理。" 是指在“console.log() 调用或 for...of 循环中”处理异常么？ touch(x)和涉及的资源无法处理，是什么意思呢？ 是指touch(x)内部抛出异常，但catch不到异常么？

```
function touch(x) {
  try{
    if (x==2) throw new Error("hard break");
  }catch(c){
    console.log('c:'+ c);
  }
}
```





```

    }
}

// 迭代函数
function foo2(x = 5) {
  return {
    next: () => {
      touch(x); // some process methods
      return {done: !x, value: x && x--};
    },
    "return": () => console.log("RETURN!"),
    "throw": () => console.log("THROW!")
  }
}

// 示例
var x = new Object;
x[Symbol.iterator] = foo2; // default `x` is

try{
  console.log(...x);
}catch(d){
  console.log('d:' + d);
}

```

结果：touch(x)可以处理异常  
 c:Error: hard break  
 5 4 3 2 1

作者回复: 以for..of语句为例，在那个例子中发生异常时，在`console.log(i)`位置是无法捕获异常的；在for..of外层可以加一层异常，但是无法处理touch()过程所分配的（或者所“碰(touch)”过的）资源。例如，如果touch()过程打开了一个文件，那么当异常发生时，在哪里去关闭文件呢？



油菜

2020-11-13



- 1 语言设计者有考虑到异常处理，但功能上还不够完善。类似写了个todo
- 2 使用者或设计者需要一种语法，能够接收所有参数。例如剩余参数 function foo(...x){console.log(x)}，数组x可以接收所有入参。“为什么 ECMAScript 不提供一个表达式 / 语句之外的概念来指代它”（老师可能是想说，之内的概念，文章标题已指出(...x)既不是表达式，也不是

语句)，可能是和现有的编译逻辑冲突。例如我们写程序，如果出现分支情况，最简单的做法是写个if判断，区分分支。表达式，语句，语法，可能是不同分支处理。

3 本人简单测试，以下几种都会触发tor.return。

```
for(var i of x ) {console.log(i);break};  
for(var i of x ) {console.log(i); throw new Error('test')}  
for(var i of x ) {console.log(i);continue};
```

4 只知其然，不知其所以然

// 迭代函数

```
function foo6(x = 5) {  
  return {  
    // foo2()中的next  
    next: () => {  
      return {done: !x, value: x && x--};  
    },  
    // foo3()中的return和throw  
    "return": () => console.log("RETURN!"),  
    "throw": () => console.log("THROW!")  
  }  
}  
  
var b = foo6();  
b.next();// { done: false, value: 5 }  
b.next();// { done: false, value: 4 }
```

//yield用法

```
var x = new Object;  
x[Symbol.iterator] = foo6;  
function* g1() { yield* x};  
var a = g1();  
a.next(); // { done: false, value: 5 }  
a.next(); // { done: false, value: 4 }
```



Elmer

2020-07-07

这个贴近语义的语法和解析为可执行结构的关系是什么呢



作者回复: 这个这个，你的这句话我读不太明白意思。

如果你的意思是“语法与可执行结构”的关系，那么可以简单地说：在语法解析之后，会在相应的位置上插入一段硬代码，以支持对应的可执行结构的实际语义。

例如如果是“展开...x作为参数”，那么对应的位置上的代码，其实就是把数组x添加到参数列表的指定位置。这是一段确定的逻辑。



HoSalt

2020-05-18

老师，为什么fn(...args)怎么变成了一个的参数，而在{...args}怎么变成了对象属性的一部分，这是谁控制的？

作者回复: ...x 这个称为“展开语法”，是在语法分析阶段就被替换成了硬代码的。所以简单的说，在执行期的时候就是一个循环/迭代处理，并且根据上下文的不同，来决定是展开成参数，还是属性。

类似于用字符串替换重写了代码，硬写入的。^^.



亦枫、

2020-04-03

看了好久，对于第二个思考题恍然大悟，我想可以用文章开头的

\*\*“换言之，可以通过更显式的、特指的或与应用概念更贴合的语法来表达新的语义。”\*\*

这句话来解释吧，它只是“展开”逻辑的一种实现，叫它展开语法更为显式，与应用概念更贴合吧。



antimony

2020-02-09

老师，请问一下这个迭代和递归的关系您是从sicp中了解到的吗？

作者回复: 不是。^^.

循环与递归的关系可能源出于SICP。但我是在读SICP之前听一个朋友谈及的，后来意识到根本问题是抽象模型内在的一致性。所以才有了《程序原本》中说“本质上相同的抽象系统，其解集的抽象本质上也是相同的”。基于此，再观察论述迭代与递归的关系，就是显而易见的事情了。



《程序原本》：

<https://github.com/aimingoo/my-ebooks>



**weineel** 

2019-12-02

四个思考题，都没能找到很好的答案。。。。

共 1 条评论 >

