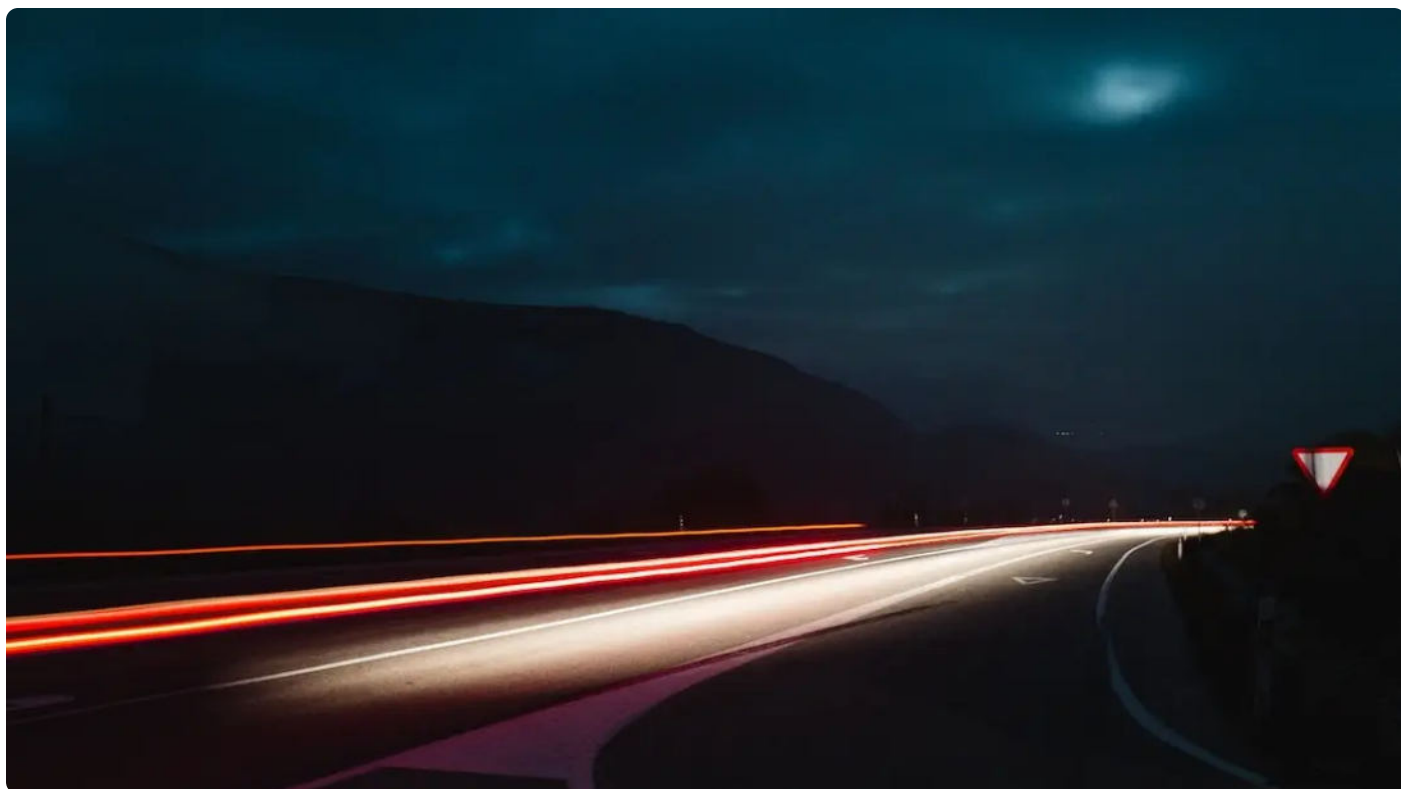


08 | $x \Rightarrow x$: 函数式语言的核心抽象：函数与表达式的同一性

2019-11-29 周爱民

《JavaScript核心原理解析》

[课程介绍 >](#)



讲述：周爱民

时长 23:19 大小 21.36M



你好，我是周爱民。

在运行期，语句执行和特殊的可执行结构都不是 JavaScript 的主角，多数情况下，它们都只充当过渡角色而不为开发人员所知。我相信，你在 JavaScript 中最熟悉的执行体一定是**全局代码**，以及**函数**。

而今天，我要为你解析的就是**函数的执行过程**。

如同在之前分析语句执行的时候与你谈到过的，语句执行是命令式范型的体现，而函数执行代表了 JavaScript 中对函数式范型的理解。厘清这样的基础概念，对于今天要讨论的内容来说，是非常重要和值得的。

很多人会从**对象**的角度来理解 JavaScript 中的函数，认为“函数就是具有[[Call]]私有槽的对象”。这并没有错，但是这却是从静态视角来观察函数的结果。



要知道函数是执行结构，那么执行过程发生了什么呢？这个问题从对象的视角是既观察不到，也得不到答案的。并且，事实上如果上面这个问题问得稍稍深入一点，例如“对象的方法是怎么执行的呢”，那么就必须要回到“函数的视角”，或者运行期的、动态的角度来解释这一切了。

函数的一体两面

用静态的视角来看函数，它就是一个函数对象（函数的实例）。如果不考虑它作为对象的那些特性，那么函数也无非就是“用三个语义组件构成的实体”。这三个语义组件是指：

1. 参数：函数总是有参数的，即使它的形式参数表为空；
2. 执行体：函数总是有它的执行过程，即使是空的函数体或空语句；
3. 结果：函数总是有它的执行的结果，即使是 `undefined`。

并且，重要的是“这三个语义组件缺一不可”。晚一点我会再来帮你分析这个观点。现在你应该关注的问题是——我为什么要在此之前强调“用静态的视角来看”。

在静态的语义分析阶段，函数的三个组件中的两个是显式的，例如在下面的声明中：

```
1 function f() {  
2     ...  
3 }
```

 复制代码

语法 `()` 指示了参数，而 `{ }` 指示了执行体，并且，我们隐式地知道该函数有一个结果。这也是 JavaScript 设计经常被批判的一处：由于没有静态类型声明，所以我们也无法知道函数返回何种结果。当我们把这三个部分构成的一个整体看作执行体的时候：

```
1 (function f() {  
2     ...  
3 })
```

 复制代码

那么它的结果是一个函数类型的“数据”。这在函数式语言中称为“函数是第一类型的”，也就是说函数既是可执行的**逻辑**，也同时是可以被逻辑处理的**数据**。



函数作为数据时，它是“原始的函数声明”的一个实例（注意这里并不强调它是对象实例）。这个实例必须包括上述三个语义组件中的两个，即**参数**与**执行体**。否则，它作为实例将是不完整的、不能准确复现原有的函数声明的。为了达到这个目的，JavaScript 为每个实例创建了一个闭包，并且作为上述“函数类型的‘数据’”的实际结果。例如：

[复制代码](#)

```
1 var arr = new Array;
2 for (var i=0; i<5; i++) arr.push(function f() {
3   // ...
4 });
```

在这个例子中，静态的函数f()有且仅有一个；而在执行后，arr[]中将存在该函数f()的 5 个实例，每一个称为该函数的一个运行期的闭包。它们各各不同，例如：

[复制代码](#)

```
1 > arr[0] === arr[1]
2 false
```

所以简而言之，任何时候只要用户代码引用一次这样的函数（的声明或字面量），那么它就会拿到该函数的一个闭包。注意，得到这个闭包的过程与是否调用它是无关的。

两个语义组件

上面说过，这样的闭包有两个语义组件：参数和执行体。在创建这个闭包时，它们也将同时被实例化。

这样做的目的，是为了保证每个实例 / 闭包都有一个自己独立的运行环境，也就是**运行期上下文**。JavaScript 中的闭包与运行环境并没有明显的语义差别，唯一不同之处，仅在于这个“运行环境”中每次都会有一套新的“参数”，且执行体的运行位置（如果有的话）被指向函数代码体的第一个指令。

然而，你或许会问：我为什么要如此细致地强调这一点，巨细无遗地还原创建这样的环境的每一步呢？



这样的小心和质疑是必要的！如果你真的这样问了，那么非常感谢，你提出了“函数”的一个关键假设：它可以是多次调用的。

这显然是废话。

但是，如果你将它与之前讨论过的 for 循环对照起来观察的话，就会发现一个事实：函数体和 for 的循环体（这些用来实现逻辑复用的执行结构）的创建技术，是完全一样的！

也就是说，命令式语句和函数式语言，是采用相同的方式来执行逻辑的。只不过前者把它叫做 `_iteratorEnv_`，是 `_loopEnv_` 的实例；后者把它叫做闭包，是函数的实例。

再往源头探究一点：导致 for 循环需要多个 `_iteratorEnv_` 实例的原因，在于循环语句试图在多个迭代中复用参数（迭代变量），而函数这样做的目的，也同时是为了处理这些参数（形式参数表）的复用而已。

所以，闭包的作用与实现方法都与“for 循环”中的迭代环境没有什么不同。同样地，对于这一讲的标题中的这行代码来说，它也（首先）代表了这样两个语义组件：

- 参数 `x`
- 执行体 `x`

在闭包创建时，参数 `x` 将作为闭包（作用域 / 环境）中的名字被初始化——这个过程中“参数 `x`”只作为名字或标识符，并且“将会在”闭包中登记一个名为“`x`”的变量；按照约定，它的值是 `undefined`。并且，还需要强调的是，这个过程是引擎为闭包初始化的，发生于用户代码得到这个闭包之前。

然而所谓“参数的登记过程”很重要吗？当然重要。

简单参数类型

完整而确切地说，这一讲标题中的函数是一个“简单参数类型的箭头函数”。而下面这个就不“简单”了：



```
1 (x = x) => x;
```

复制代码

在 ECMAScript 6 之前的函数声明中，它们的参数都是“简单参数类型”的。在 ECMAScript 6 之后，凡是在参数声明中使用了缺省参数、剩余参数和模板参数之一的，都不再是“简单的” (*non-simple parameters*)。在具体实现中，这些新的参数声明意味着它们会让函数进入一种特殊模式，由此带来三种限制：

1. 函数无法通过显式的"use strict"语句来切换到严格模式，但能接受它被包含在一个严格模式的语法块中（从而隐式地切换到严格模式）；
2. 无论是否在严格模式中，函数参数声明都将不接受“重名参数”；
3. 无论是否在严格模式中，形式参数与 arguments 之间都将解除绑定关系。

这样处理的原因在于：在使用传统的简单参数时，只需要将调用该参数时传入的实际参数与参数对象 (arguments) 绑定就可以了；而使用“非简单参数”时，需要通过“初始器赋值”来完成名字与值的绑定。同样，这也是导致“形式参数与 arguments 之间解除绑定关系”的原因。

NOTE 1: 两种绑定模式的区别在于：通常将实际参数与参数对象绑定时，只需要映射两个数组的下标即可，而“初始器赋值”需要通过名字来索引值（以实现绑定），因此一旦出现“重名参数”就无法处理了。

所以，所谓参数的登记过程，事实上还影响了它们今后如何绑定实际传入的参数。

传入参数的处理

要解释“参数的传入”的完整过程，得先解释为什么“形式参数需要两种不同的登记过程”。而在这所有一切之前，还得再解释什么是“传入的参数”。

首先，JavaScript 的函数是“非惰性求值”的，也就是说在函数界面上不会传入一个延迟计算的求值过程，而是“积极地”传入已经求值的结果。例如：

```
1 // 一般函数声明
2 function f(x) {
3   console.log(x);
4 }
5
6 // 表达式`a=100`是“非惰性求值”的
7
```

复制代码



`f(a = 100);`

在这个示例中，传入函数`f()`的将是赋值表达式`a = 100`完成计算求值之后的结果。考虑到这个“结果”总是存在“值和引用”两种表达形式，所以 JavaScript 在这里约定“传值”。于是，上述示例代码最终执行到的将是`f(100)`。

回顾这个过程，请你注意一个问题：`a = 100`这行表达式执行在哪个上下文环境中呢？

答案是：在函数外（上例中是全局环境）。

接下来才来到具体调用这个函数`f()`的步骤中。而直到这个时候，JavaScript 才需要向环境中的那些名字（例如`function f(x)`中的形式参数名 `x`）“绑定实际传入的值”。对于这个`x`来说，由于参数与函数体使用同一个块作用域，因此如果函数参数与函数内变量同名，那么它们事实上将是同一个变量。例如：

 复制代码

```
1 function f(x) {  
2   console.log(x);  
3   var x = 200;  
4 }  
5 // 由于“非惰性求值”，所以下面的代码在函数调用上完全等义于上例中`f(a = 100)`  
6 f(100);
```

在这个例子中，函数内的三个 `x` 实际将是同一个变量，因此这里的`console.log(x)`将显示变量`x`的传入参数值 100，而`var x = 200;`并不会导致“重新声明”一个变量，仅仅是覆盖了既有的 `x`。

现在我们回顾之前讨论的两个关键点：

1. 参数的登记过程发生在闭包创建的过程中；
2. 在该闭包中执行“绑定实际传入的参数”的过程。


意外

对于后面这个过程来说，如果参数是简单的，那么 JavaScript 引擎只需要简单地绑定它们的一个对照表就可以了。并且，由于所有被绑定的、传入的东西都是“值”，所以没有任何需要引



用其它数据的显式执行过程。“值”是数据，而非逻辑。

所以，对于简单参数来说，是没有“求值过程”发生于函数的调用界面上的。然而，对于下面例子中这样的“非简单参数”函数声明来说：

 复制代码

```
1 function foo(x = 100) {  
2   console.log(x);  
3 }  
4 foo();
```


在“绑定实际传入的参数”时，就需要执行一个“ $x = 100$ ”的计算过程。不同于之前的 $f(a = 100)$ ，在这里的表达式 $x = 100$ 将执行于这个新创建的闭包中。这很好理解，左侧的“参数 x ”是闭包中的一个语法组件，是初始化创建在闭包中的一个变量声明，因此只有将表达式放在这个闭包中，它才可以正确地完成计算过程。

然而这样一来，在下面这个示例中，表达式右侧的 x 也将是该闭包中的 x 。

 复制代码

```
1 f = (x = x) => x;
```

这貌似并没有什么了不起的，但真正使用它的时候，会触发一个异常：

 复制代码

```
1 > f()  
2 ReferenceError: x is not defined  
3   at f (repl:1:10)
```

这是一个意外。


无初值的绑定

这个异常提示其实并不准确，因为在这个上下文环境（闭包）中， x 显然是声明过的。事实上，这也是两种不同的登记过程（“直接 arguments 绑定”与“初始器赋值”）的主要区别之



一。尽管在本质上，这两种登记过程所初始化的变量都是相同的，称为“**可变绑定** (*Mutable Binding*) ”。

“可变”是指它们可以多次赋值，简单地说就是 let/var 变量。但显然地，上述的示例正好展示了 var/let 的两种不同性质：

 复制代码

```
1 function foo() {  
2   console.log(x); // ReferenceError: x is not defined  
3   let x = 100;  
4 }  
5 foo();
```

由于 let 变量不能在它的声明语句之前（亦即是未初始化之前）访问，因此上例触发了与之前的箭头函数 f() 完全相同的异常。也就是说，在 (x = x) => x 中的三个 x 都是指向相同的变量，并且当函数在尝试执行“初始器赋值”时会访问第 2 个 x，然而此时由于变量 x 是未赋值的，因此它就如同 let 变量一样不可访问，从而触发异常。

为什么在处理函数的参数表时要把 x 创建为一个 let 变量，而不是 var 变量呢？

事实上，这二者并没有区别，如之前我所讲过的，它们都是“可变绑定 (*Mutable Binding*) ”。并且，对于 var/let 来说，一开始的时候它们其实都是“无初值的绑定”。只不过 JavaScript 在处理 var 语句声明的变量时，将这个“绑定 (Binding)”赋了一个初值 undefined，因此你才可以在代码中自由、提前地访问那些“var 变量”。而对应的，let 语句声明的变量没有“缺省地”赋这个初值，所以才不能在第一行赋值语句之前访问它，例如：

 复制代码

```
1 console.log(x); // ReferenceError: x is not defined  
2 let x = 100;
```

处理函数参数的过程与此完全相同：参数被创建成“可变绑定”，如果它们是简单参数则被置以初值 undefined，否则它们就需要一个所谓的“初始器”来赋初值。也就是说，并非 JavaScript 要刻意在这里将它作为 var/let 变量之一来创建，而只是用户逻辑执行到这个位置的时候，所谓的“可变绑定”还没有来得及赋初值罢了。



然而，唯一在这个地方还存疑的是：**为什么不干脆就在“初始器”创建的时候，就赋一个初值 undefined 呢？**

说到这里，可能你也猜到了，因为在“缺省参数”的语法设计里面，undefined 正好是一个有意义的值，它用于表明参数表指定位置上的形式参数是否有传入，所以参数 undefined 也就不能作为初值来绑定，这就导致了使用“初始器”的参数表中，所对应那些变量是一个“无初值的绑定”。

因此如果这个“初始器”（我是指在它初始化的阶段里面）正好也要访问变量自身，那么就会导致出错了。而这个出错过程也就与如下示例的代码是一样的，并且也导致一样的错误：

 复制代码

```
1 > let x = x;  
2 ReferenceError: x is not defined
```

所以，最终的事实是 $(x = x) \Rightarrow x$ 这样的语法并不违例，而是第二个 x 导致了非法访问“无初值的绑定”。

最小化的函数式语言示例

那么现在我再来为你解析一下标题中的 $x \Rightarrow x$ 。

这行代码意味着一个最小化的函数。它包括了一个函数完整的三个语法组件：**参数**、**执行体**和**结果**，并且也包括了 JavaScript 实现这三个语法组件的全部处理过程——这些是我在这一讲中所讨论的全部内容。重要的是，它还直观地反映了“函数”的本质，就是“数据的转换”。也就是说，所有的函数与表达式求值的本质，都是将数据 x 映射成 x' 。

我们编写程序的这一行为，在本质上就是针对一个“输入 (x , input/arguments)”，通过无数次的数据转换来得到一个最终的”输出 (x' , output/return)”。所有计算的本质皆是如此，所有的可计算对象也可以通过这一过程来求解。

因此，函数在能力上也就等同于全部的计算逻辑（等同于结构化程序思想中的“单入口 \rightarrow 单出口”的顺序逻辑）。



箭头函数是匿名的，并且事实上所谓名字并不是函数在语言学中的重要特性。名字 / 标识符，是语法中的词法组件，它指代某个东西的抽象，但它本身既不是计算的过程（逻辑），也不是计算的对象（数据）。

那么，我接下来要说的是，**没有名字的函数在语言中的意义是什么呢？**

它既是**逻辑**，也是**数据**。例如：

 复制代码

```
1 let f = x => x;  
2 let zero = f.bind(null, 0);
```

现在，zero 既是一个逻辑，是可以执行的过程，它返回数值 0；也是一个数据，它包含数值 0。

NOTE 1: 箭头函数与别的函数的不同之处在于它并不绑定“this”和“arguments”。此外，由于箭头函数总是匿名的，因此它也不会环境中绑定函数名。

NOTE 2: ECMAScript 6 之后的规范中，当匿名函数或箭头函数赋给一个变量时，它将会以该变量名作为函数名。但这种情况下，该函数名并不会绑定给环境，而只是出现在它的属性中，例如“*f.name*”。

知识回顾

现在我来为这一讲的内容做个回顾。

1. 传入参数的过程执行于函数之外，例如 `f(a=100)`；绑定参数的过程执行于函数（的闭包）之内，例如 `function foo(x=100) ...`。
2. `x=>x` 在函数界面的两端都是值操作，也就是说 input/output 的都是数据的值，而不是引用。
3. 参数有两种初始化方法，它们根本的区别在于绑定初值的方式不同。
4. 闭包是函数在运行期的一个实例。





思考题

1. 表达式如何等价于上述计算过程？
2. 表达式与函数在抽象概念上的异同？
3. 试以表达式来实现标题中的箭头函数的能力。

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 5  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。

上一篇 07 | `\${}`：详解JavaScript中特殊的可执行结构

下一篇 09 | (...x)：不是表达式、语句、函数，但它却能执行

学习推荐

JVM + NIO + Spring

各大厂面试题及知识点详解

限时免费 





ssala

2019-12-21

表达式也是3部分，如果以类比函数的思维来看待表达式，我认为：

1. 表达式中用到的参数构成了表达式的“参数”
2. 表达式本身构成了其逻辑体，这是表达式的逻辑部分
3. 表达式的运算结果就是它的“返回值”

单值表达式x在逻辑上等同于标题中的箭头函数。

作者回复: 非常赞的思考！

表达式和函数并没有本质不同，它们的抽象形式是一样的。事实上，如果你理解“当运算符等义于某个函数”时，那么你其实就看到了“函数式语言”范式的根本逻辑，关于这一点，我之前也讲过，在《JavaScript语言精髓与编程实践》中也是拿这个来引入函数式语言的。

当运算符等义于函数，那么操作数就是入口参数，而运算结果就是函数返回。运算在本质上就是一个IO和一个处理。这样的理解会让你看到计算理论中最基础的那些模型。

谢谢你提出这一点，真的很赞！



10



kittyE

2019-11-30

我理解 x 这个单值表达式 是不是可以等价于 $x \Rightarrow x$; 的计算过程？毕竟直接return了 不知道理解的对不对

作者回复: 非常赞。这个答案其实比我想的还要好。^^.

共 2 条评论 >

10



海绵薇薇

2019-12-02

Hello，老师好：），也不知道想问啥，感觉还是没有理清楚，望指点，问题如下：

0.



闭包是函数表达式的值？

1.

```
let c = function (a) {}
```

c是闭包（上面匿名函数的实例），并且闭包中有标识符a（简单参数）

```
let e = c
```

e应该也是闭包，并且指向c（e === c 为true），也有标识符a

现在看来e和c闭包中标识符a是同一个，但是不应该是同一个。应该是哪里有了根本性问题，导致这样的理解。

2.

函数一体两面，逻辑和数据。那么闭包指的是逻辑？数据？还是函数的另一个名字表示的逻辑和数据？

作者回复: 先把你的第一个问题回复了，你试着再推演一下其它的。——你的第一个问题问得非常到位，是你思考上下文中的关键点。赞！

“闭包是函数表达式的值？”答案即“是”，且“非”。

首先说“是”的部分。单看ECMAScript这里：

<https://tc39.es/ecma262/#sec-function-definitions-runtime-semantics-evaluation>

你会发现，一个函数表达式执行之后，就是“Return closure”。所以，这个说法在ECMAScript层面是对的。

但是我通常在这里说，这个过程得到的其实还只是一个“函数实例”。为什么呢，比如说：

```
...
```

```
for (var arr = [], i = 0; i < 10; i++) arr.push(function() {});
```

```
...
```

这里有几个匿名函数在arr[]里面？答案是有10个。所以函数表达式每执行一次，就得到一个函数实例。你的示例`let c = function() {}`也一样，右操作数每执行一次就得到一个新的。

但是这是函数实例。函数闭包其实是在它调用的时候才创建和初始化的。——只不过，所有的、因为函数调用而产生的闭包，都是上面这个实例的一个“副本”。所以，一个实例其实也可以有多个闭包。这里的意思是说，ECMAScript在上面把它叫闭包：既没错，它是所有其它闭包的“母本”；也不太



对，因为它不是你在执行过程看到的任何一个闭包。

——一个函数实例可以有多个闭包，这很少见，但的确是存在的。比如递归，其实是一个函数的多次调用，那这个函数（实例）只有一个，闭包却是有多了。

共 2 条评论 >

👍 9



海绵薇薇

2019-12-02

Hello 老师好：)

一直以来各大博客描述的包括我理解的闭包是：函数可以访问函数作用域外标识符的能力。但是本文描述闭包的角度完全不一样。这两者有联系吗？

作者回复: "函数可以访问函数作用域外标识符的能力", 这个与闭包没关系, 而是scope-chains的事情。不过现在这个概念也改了, 在ES6之后, 这里被称为环境/作用域。所以, 如果仅仅是在概念层面讨论这个话题, 比较容易变形。

回到提出这个概念的早期, 在ES3~ES5的那个时代, 概念比较简单清晰一些。有一个东西称为“对象闭包 (object closure)”, 这个东西写在spidermonkey的代码里面, 也写在一个称为narcissus的开源项目里面, 尤其是narcissus, 它是JS之父本人Eich一直在维护的一个项目, 也就是说, 是一种正统的“概念”。——我说这一段是想说明, 我下面的说法不是心血来潮随口胡诌。

Ok。什么是对象闭包呢? 所谓对象闭包, 就是“with (x) ...”中打开的那个用`x`的属性表来形成的闭包。所以, 如果 (注意这个推论哟), 如果对象闭包是一种闭包, 那么要么是概念上属性表等义于闭包, 要么是闭包是一种属性表。

OVER。^^.



👍 6



leslee

2019-12-25

为什么箭头函数不绑定 this 跟arguments

作者回复: 他的设计便是如此。使用当前上下文中的this和arguments, 有些时候是很有用的, 对现有的normal function的设计也是补充。所以, 除了开始使用的时候不习惯之外, 在语言功能的设计上还是不错的。

另外, 箭头函数没有this, 也意味着它不支持this (和arguments) 的隐式传入, 这在函数式语言特性



上是更纯粹的，更接近lamada函数的概念。不过由于JavaScript语言自己的特殊性，它做不到无副作用罢了。



👍 5



Elmer

2020-07-07

非简单参数是以let声明的

```
function test (x = 2) {var x = 3; console.log(x)} // output 3
```

而let x = 2; var x = 3 却会报错

这个该怎么解释呢

作者回复: 终于写完了这个问题的回复，拖得太久了，不过希望回复能解您的所惑，值得一读。放在了这里：

<https://aimingoo.github.io/>



👍 4



海绵薇薇

2019-12-03

Hello，老师好：)

关于函数闭包又有了新的理解，函数形成的闭包是函数本身 + 函数当前的环境的集合。也就是说函数表达式的Result（闭包母本）中应该包含当前函数的环境（词法环境/执行逻辑的土壤）和函数体本身（对象/数据&逻辑）。

作者回复: 赞的。^^.

...

// f是函数x的一个实例

```
f = function x() {}
```

// 创建f的一个闭包、初始化该闭包并在其中执行代码

```
f()
```

...



👍 3



westfall

2021-01-19

老师给被人的回复说闭包是在函数调用的时候才创建和初始化的。但是文中又有一句话“得到这个闭包的过程与是否调用它是无关的。”这里是不是有矛盾？

作者回复: Oh...其实主要还是我说得不清楚。

对于函数表达式来说，函数表达式总是在其它表达式运算过程中被引用的，以下面的代码为例：

```
...  
arr.push(function() {  
  // ...  
})  
...
```

这个“匿名函数（表达式）”是在push()方法中被作为运算数引用的。同理，`1 + function() { ... }`这样的代码，是在+运算中被作为右操作数引用的。

函数表达式在被引用时，它自己作为运算元（运算数）会被“执行”一次，这就好象是“单值表达式”作为值也需要被“求值”一次一样。这个操作，可以理解为匿名函数表达式的实例化过程——从代码文本变成一个代码实例，而同时，这个实例的闭包就被创建了。这与“函数声明”中的函数是不一样的。

函数声明（比如说一个声明的具名函数）如下：

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

在引擎处理到这个具名函数的声明时，它也会被实例化。但是你看到它并没有执行。执行过程要等到类似下面的代码出现时：

```
...  
foo();  
  
// or  
aCallbackPromise.then(foo)  
  
// or  
setTimeout(foo, 1000)  
...
```

类似于此的时候，当这个foo被调用时——调用行为才会创建起这个闭包来。

所以，确切地说：函数与函数表达式，在它们的闭包的处理行为上是不一样的。函数表达式是当它被“求值到”的时候，就创建了一个闭包，而具名函数（例如foo）是在它调用、亦即时foo()发生时



候，才创建的闭包。

我应该是没有非常明确地阐述这二者的区别，这才带来了你的误解。——而在本文中，这里正好处理的是函数表达式。

共 2 条评论 >

👍 2



HoSalt

2020-05-14

「一个函数实例可以有多个闭包，这很少见，但的确是存在的。比如递归，其实是一个函数的多次调用，那这个函数（实例）只有一个，闭包却是有多多个了。」

老师，看了你回的回答，感觉表达了闭包是函数执行调用时才有的，而我的认知里闭包是和函数具体的执行调用无关的

```
// 1
```

```
function test () {
```

```
}
```

```
test()
```

我理解的闭包是和test相关的，和test()无关

```
// 2
```

```
test function () {
```

```
  return bar () {
```

```
  }
```

```
}
```

```
var a = test()
```

test有个闭包，test()执行后的闭包是bar的，和test()这个过程无关

```
// 3
```

```
function test(i) {
```

```
  if(i < 10 || i > 0) return
```

```
  return test(i - 1)
```

```
}
```

```
test(5)
```

这个递归过程，闭包是test，和执行递归的过程无关

上面是我对闭包的理解，还有就是函数调用是一个表达式，表达式怎么会产生闭包，即使返回了一个函数，那也是和返回函数相关的闭包，与当前执行的函数无关



我们常说闭包使用不当会导致内存泄漏，以2的函数为例，执行test()即使造成了内存泄漏也是关联到bar的闭包，而非test的闭包，这也是我认为闭包和执行过程无关的原因之一

老师，不知道我上面的理解有什么问题，感觉和你在评论区与大家讨论的不一样

作者回复: 有一个简单示例来说明“一个函数实例可以有多个闭包”。如下:

```
...  
function foo() {  
  return argument.callee;  
}  
foo()()  
...
```

在这个例子中，foo被执行了两次。并且在第二次执行的时候直接使用了arguments.callee——也就是第一次执行时的那个函数实例“本身(callee)”。

这种情况下，第一个实例没有被释放（它内部发起了一次call function，因此栈不能释放），只可能是新建一个同一实例的闭包。

最后，闭包是动态运行期的概念，是发生于函数调用行为之中的（为每次call创建一个）。与运行期无关的，是词法环境，早先的时候也称为作用域（scope）。



👍 2



子笺

2019-12-20

听完老师这一讲，我脑海中有幅构图：

函数有几个基本要素：执行的上下文【素材】，执行的过程【对素材的加工】，执行的结果【成果】

为了完成成果，必须要有特定的方式去处理整合素材，有合理素材处理流程

首先是整合素材：考虑到整合素材时，有普通的参数传递方式，还有一些特殊的（缺省、解构），就用了2种形式去整合素材。

然后是对素材的加工：结合我对编译的认知，这些素材都有各自的占位，在解析阶段，其实已经有了对各个占位的处理流水线，在执行阶段这些占位真正有了自己的实物，直接在已经构建的流水线上去做处理。

最终可以拿到成果

同理，我想表达式也差不多。

在这节课之前才看了老师关于如何学习的加餐，不管对错，先有自己的理解



作者回复: 赞！这是极难得的进展！

值得再赞一次！！



2

**Geek_8d73e3**

2020-06-21

两种绑定模式的区别在于：通常将实际参数与参数对象绑定时，只需要映射两个数组的下标即可，而“初始器赋值”需要通过名字来索引值（以实现绑定），因此一旦出现“重名参数”就无法处理了。

老师关于这一段：

```
function a(x=1,y=2){}
```

```
a(100,200);
```

就算使用非简单类型，依旧可以映射数组的下标呀，因为位置是不变的，为什么要解除绑定arguments？

很明显，我们的arguments[0]就是x，arguments[1]就是y呀

作者回复：在旧的模式中，参数和arguments下标是绑定的，所以：

```
...
```

```
function f(x) {  
  console.log(x); // input - 0  
  arguments[0] = 100;  
  console.log(x); // 100  
  console.log(arguments[0]); // 100  
}  
f(0);
```

```
...
```

但是非简单类型参数是，这两个东西是不绑定的。例如：

```
...
```

```
function f(x = 'a') {  
  console.log(x); // input - 0  
  arguments[0] = 100;  
  console.log(x); // input - 0  
  console.log(arguments[0]); // updated - 100  
}  
f(0);
```

```
...
```

这与参数的实现方法（两种绑定方式）有关，而与是不是调用时传入了缺省参数——以及其它非简单参数——是无关的。



1

老师你说 $x \Rightarrow x$ 两端都是值操作。

但是当传入函数的是引用类型的数据的时候。函数内部改变行参，实参也是会变的，这传入的不是引用么？

```
var obj = {
  name:"z",
  age:18
}
let ab = (myobj) => {myobj.age = 19}
```

```
ab(obj);
```

```
console.log(obj);
//输出
// {
//   "name": 'z',
//   "age": 19
// }
```

作者回复: 这引用，与你所说的引用，不是同一个东西啊。这从第一讲就开始讲的嘛。

```
...
```

```
x => x
```

```
...
```

左边的 x ，是指“ECMAScript规范类型”的引用，它向函数内传入（input）时，会取它的值，也就是`GetValue(ref_x)`；同理，右边的 x 向函数外传出（output）时，也是用`GetValue(ref_x)`来取值的。

要说明这一点并不难。例如“`obj.x`”，就是一个`x`的“ES规范类型的引用”。同理，又例如使用`var`在全局声明的“ x ”，实际上就是“`global.x`”的“ES规范类型的引用”。那么，在下例中，引用信息是丢掉的，传入的实际是“值”：

```
...
```

```
let foo = x => x;
let f = new Function;
let obj = { f };
```

```
// 对于如下代码：
foo(obj.f)
```



```
// 在引擎内部将处理成如下逻辑：
foo(GetValue(MakeReference(obj, "f")));

// 等义于：
foo(f);
...

```



HoSalt

2020-05-14

「箭头函数与别的函数的不同之处在于它并不绑定“this”和“arguments”」箭头函数的this和其定义的词法环境的this一样，这不也是一种绑定吗？

```
作者回复：``
function foo1(_this) {
  // access _this;
}

let _this;
function foo2() {
  // access _this
}

function func(_this) {
  let foo3 = () => /*access _this*/;
}
...

```

foo1()中的_this是绑定，因为这个_this是属于foo1()的上下文的。

而foo2()中的_this就不是绑定，因为它属于foo2()之外的环境。

类似foo2()的，在foo3()中，_this并不属于foo3()的环境，而属于func()的环境，因此“箭头函数...并不绑定`this`”。同理，也不绑定arguments。



IAmFineThanks

2019-12-11

函数闭包是函数执行时创建的，可是按照闭包的变量存储位置是在堆里的前提，函数每次执行

完后的变量不应该被删除掉，但是实际上函数每次调用完后函数内部的变量都会被GC回收掉，感觉有点不是很懂~

作者回复: 这里你是按传统的语言来理解的。并且，即使按“传统的语言”来理解，也并不太正确。

首先，并不是函数（或闭包）的变量存储在堆里。在一些语言中（例如Delphi），函数内的局部变量也是可以存放在栈上的，并且多数情况下就是如此。所以一些传统的语言概念和实现思路，并不见得总是对的。

其次，JavaScript是解释执行的，闭包（这里反过来说，或者函数）中的变量与标识符并不是使用堆/栈来管理的。所以它的回收也不是堆与栈的机制，通常这基于引用计数，但在不同的引擎中可能存在区别，是难以一概而论的。但是其中所谓“闭包”，才是回收所基于的一些基础组件，闭包代表了对对象、数据以及其它东西的引用，并且闭包自身也被作为引用对象——所以“函数每次调用完后函数内部的变量”其实在JavaScript中并不总是会被GC掉，因为闭包可能被别的地方引用了。例如简单的想法：你在函数内部return arguments.callee，那么函数（作为闭包）就被外部的环境引用了，那么显然变量之类的也不能被GC掉了。

总之，JavaScript的函数与传统语言中的函数有着很大的区别。从语言特性上来说，它的动态语言特性和函数式语言特性，都会给传统语言背景下的学习者带来很大的困扰。这也是我在开篇词里面一再强调“要放下既有知识包袱”的原因。



1



IAmFineThanks

2019-12-11

处理函数参数的过程与此完全相同：参数被创建成“可变绑定”，如果它们是简单参数则被置以初值undefined，否则它们就需要一个所谓的“初始器”来赋初值。也就是说，并非 JavaScript 要刻意在这里将它作为 var/let 变量之一来创建，而只是用户逻辑执行到这个位置的时候，所谓的“可变绑定”还没有来得及赋初值罢了。

什么意思？为什么突然就能得出结论来了

作者回复: let与var都是可变绑定，它们的不同不在于自身的数据结构（它们在引擎层面都是一样的）。

然而var是有初值的，而let是无初值的，这个是它们在创建的时候（或者在创建之后“随即”）发生的变化。进一步的，规范约定“不能访问一个没有初值的绑定”。因此，let就不能在它的赋值代码之前访问，而var则可以（因为它有初值），这是他们之间的差异的由来。

对于函数参数这个例子。简单参数是赋初值的，所以表现起来跟var一样。——对照起来，非简单参数也是一个“可变绑定”，但是没有赋初值（所以表现起来跟let一样）。文章中在这里的叙述，是想强



调“使这个参数表现得像var/let”并不是原始的目的，原始的状况是这里本来没有什么目的性，只不过由于undefined这个值（作为缺省值）被缺省参数占用了，所以没办法用作初值而已。



👍 1



许童童

2019-11-29

加油，希望以后来回顾的时候，可以回答出来上面三个思考题。



👍 1



Kids See Ghost

2022-01-08

老师新年快乐。请问您说的“由于所有被绑定的、传入的东西都是“值”，所以没有任何需要引用其它数据的显式执行过程。“值”是数据，而非逻辑。”跟很多人理解的，在给函数传入参数的时候，primitive values e.g. number, string 是传值，object 传引用（reference）有什么区别呢？

前段时间看了Dan Abramov 的Just JavaScript，里面也在强调任何时候给函数穿参数的都是传值，理解成传reference，或者传pointer是错误的。请问为什么不能这么理解？这样理解会有什么偏差呢？

感谢。

作者回复: 我们来考虑两个东西，它们都明显地“是引用”，注意这里说的都是“引用（规范类型）”。其一，是obj.x，而x是读写器属性；其二，是obj.foo，这里的foo是方法。如下：

...

```
let xxx = 0;
let obj = {
  get x() {
    return xxx++;
  }
  foo() {
    console.log(this.x);
  }
}
```

...

显然，这样访问的obj.x是渐增的值，而调用obj.foo()时将打印出obj.x，对吧？

我们接下来考虑一个一般函数参数“是传值，还是引用”的问题。例如：

...

```
function f(data, func) {
  func();
  console.log(data);
}
```



```
}  
f(obj.x, obj.foo); // 0, undefined  
````
```

下面说的内容是“假设传引用，而不是传值”的结果。

=====

你想，如果在这个示例中，向f()传入的obj.x是引用，而不是这个“从这个引用中取得的x`属性的值，那么就需要等到console.log(data)时，才调用内部过程GetValue (obj.x) 去取值。——那么，这个值还是f()界面上传入时的那个值吗？不一定，因为它可能已经在其它地方调用过了，是变化的。这意味着访问的函数界面是不确定的。

同理，在这个函数内部调用的func如果是引用，那么调用它的时候就“应该”还是一个引用，可以用func中得到this的，并且返回有效的this.x.....

=====

显然上述是不符合JavaScript内的事实上的。

现实的情况是，我们的js在这个函数界面上通过参数传递的“总是值”。这样一来，示例中的f()调用才是确定的，它传入了x=0，和func==obj.foo，前者是一般的数值，后者是一个函数（而不是关联到this引用的方法）。因此它的执行结果才是确定的，总是显示“0, undefined”。

当然，如果再执行一次，会是“1, undeifned”，但计算obj.x的过程是在界面上，在f()函数内的代码使用它之前。



1



2021-12-06

思考题：

- 1: let a = x => x 等价于上述的计算过程 // 首先确定了 参数。确定了函数体。并且有隐式的返回值
- 2: 函数表达式， 是一个匿名函数，参数则是通过 let声明 也就是说 (a = a) => a // 会报一个引用错误（这里的参数赋值是通过初始器执行。是通过名字和 下标进行赋值）。并且没有this和arguments(应为是初始器器执行， 无法绑定arguments)。函数声明， 可以是一个匿名/有名字的函数。参数则是通过arguments 和参数列表的下标对应赋值。并且在参数列表中声明多个重复的名字也只会对前一个名字进行覆盖（应为是var声明）
- 3: x => x 这一段语句首先确定 是有参数的。而却省略了「」 后边的x 可以理解为 x => (x) (x) 在这里是一个表达式同时也是一个返回值， 如果这里写 x => (x + 100) 则这里返回的运算之后的 值

老师我这样子理解对吗。 总是还感觉零零散散， 没有把东西穿起来。





作者回复: 先说第二题, 它并不是特指“函数声明 vs. 函数表达式”, 而是泛指。从抽象概念上来说, 函数与表达式都是“计算求值”, 并且, 都是“特定计算 (运算)”的抽象。——表达为函数名或运算符。就实现上来说, 表达式运行在当前环境中, 没有自己的闭包; 而函数拥有自己的闭包和运行环境。而在抽象概念上来说的不同之处在于, 函数是比表达式更高 (更泛义) 的一层抽象。

关于第1题, 我没有特别地说明“上述计算过程”指的是什么。可以理解为上述“知识回顾”中的4步, 或者本讲所说的“ $x \Rightarrow x$ ”, 回答成任一种都可以, 但如果是后者, 那可能跟第3题有点同质。

如果是说“用表达式来等价上述知识回顾中的4步”, 那么:

1. 传入参数与绑定参数, 与赋值表达式等义, 例如:  $x = 100$
2. 值操作 (传入求值, 与计算返回求值) 与下面表达式等义:  $(0, x)$   
注: 这里有两个操作, 其中连续运算符 (逗号) 表明后一个 $x$ 是计算求值, 而当使用成组运算 (一对括号) 的时候, 返回最后一个表达式的值。
3. 一般参数与带初始器的参数 (非简单参数) 与解构赋值表达式等义:  $\text{let } \{0:a, 1:b, \dots\} = \text{arguments}$
4. 表达式没有能够创建闭包的效果, 因此不能等义函数的闭包处理。

接下来说第3题 (它与第1题的第2种解法答案类似), 也就是说如何用表达式来理解标题中的“ $x \Rightarrow x$ ”。这也有三种解法, 其一是单值表达式“ $x$ ”, 也就是说标题中函数等义于直接使用该变量 (作为单值表达式), 这种解法多少有点赖皮。第二种解法是“要有运算符的表达式”, 那么它某种程度上等义于“ $(x)$ ”。——这对括号是运算求值 $x$ 并返回 $x$ 的值, 但是这个运算符有点特殊, 它返回的是`Result`, 而不是`GetValue(Result)`, 所以完全等义的表达式是第三种: “ $(0, x)$ ”。

总的来说, 这几道题有点“超出常理的难”, 但多思考总是好的。呵呵。另外, 留言中被赞到顶的ssal a的留言, 其实很能说明这一讲的主旨 (他对题3的答案也是上述的解法1)。能理解到那种程度就已经很好了, 上面几道思考题, 开拓一下思路可也。



G

2020-12-03

老师您好, 重新读这章的时候, 我产生了一些新的疑问。

1. 文章中有一个这样的例子:

```
console.log(x); // ReferenceError: x is not defined
let x = 100;
```

这里面提到的报错是 `x is not defined`, 我在测试的时候, 发现报错是 `can't access lexical declaration 'x' before initialization`

在静态语法解析阶段,  $x$  就会被声明并保存在 `lexicalNames` 中, 此时 $x$ 应该是处于已经被定义



但无初值的状态。但是文中给的报错文字是 `x is not defined`，上面的两种报错可以理解成一样的吗？

2.

```
>x //can't access lexical declaration 'x' before initialization
>let x
>x //undefined
```

在`let x`之前访问会报 `can't access lexical declaration 'x' before initialization` 的错误，原因是“无法访问一个无初值的变量”，在执行到第二行 `let x` 之后，在第三行打印`x`，会输出`undefined`。是不是意味着，在经过第二行代码之后，`x` 会从“无初值”变为“非值”呢？

作者回复: 1. 是一样的。“`x is not defined`”是稍早一些的引擎的报错，最新近的引擎很多都改过了。

2. 是的。“`let x`”这个语句隐含的有一个初始值置为`undefined`的操作。不过不是你所说的“非值”，而是一个值（值为`undefined`）。



🇧🇪 Hazard💎...

2020-04-19

老师好，请教老师下面这段：

> 因为在“缺省参数”的语法设计里面，`undefined` 正好是一个有意义的值，它用于表明参数表指定位置上 的形式参数是否有传入，所以参数 `undefined` 也就不能作为初值来绑定，

没太懂，`undefined`用于表明参数表指定位置上的形式参数是否有传入，所以有什么影响吗？就算用`undefined`作为初值了，对参数执行之后绑定了什么值有什么影响吗？

谢谢！

作者回复: 如果“用`undefined`作为（绑定的）初值了”，那么还能分得清一个参数到底是赋值为`undefined`了，还是初值为`undefined`么？

这样吧，我们换个模式来理解这个问题。仍然以文中说过的为例：

...

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

...

我们从语义上来理解上面这个代码。开发者如果想表达的是“`x`的初值是当前环境中的一个`x`值”呢，那么在下例中传入的就应该是200：



```
...
```

```
// 语义1
```

```
let x = 200
```

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

```
foo(); // <- 200
```

```
...
```

而如果想说明的是“初值为undefined”，那么声明中右侧的x应该是“函数内的x，且初始值为undefined”。例如：

```
...
```

```
// 语义2
```

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

```
foo() // <- x的初始值为undefined
```

```
...
```

显然，用户代码无法表达后面后一种意思。因为后面这种更显式的表达方式是：

```
...
```

```
// 语义3
```

```
function foo(x = undefined) ...
```

```
...
```

既然如此，`x被初始化为undefined`就会变成一个有歧义的设计了。如果使用(x = x)这种缺省参数声明，那么它在表达上似乎是“语义1”，设计上是“语义2”，而实现的却是“语义3”。

而实际上，只需要约定“x初始时是未绑定初值的”，那么上述问题就自然解决了。

