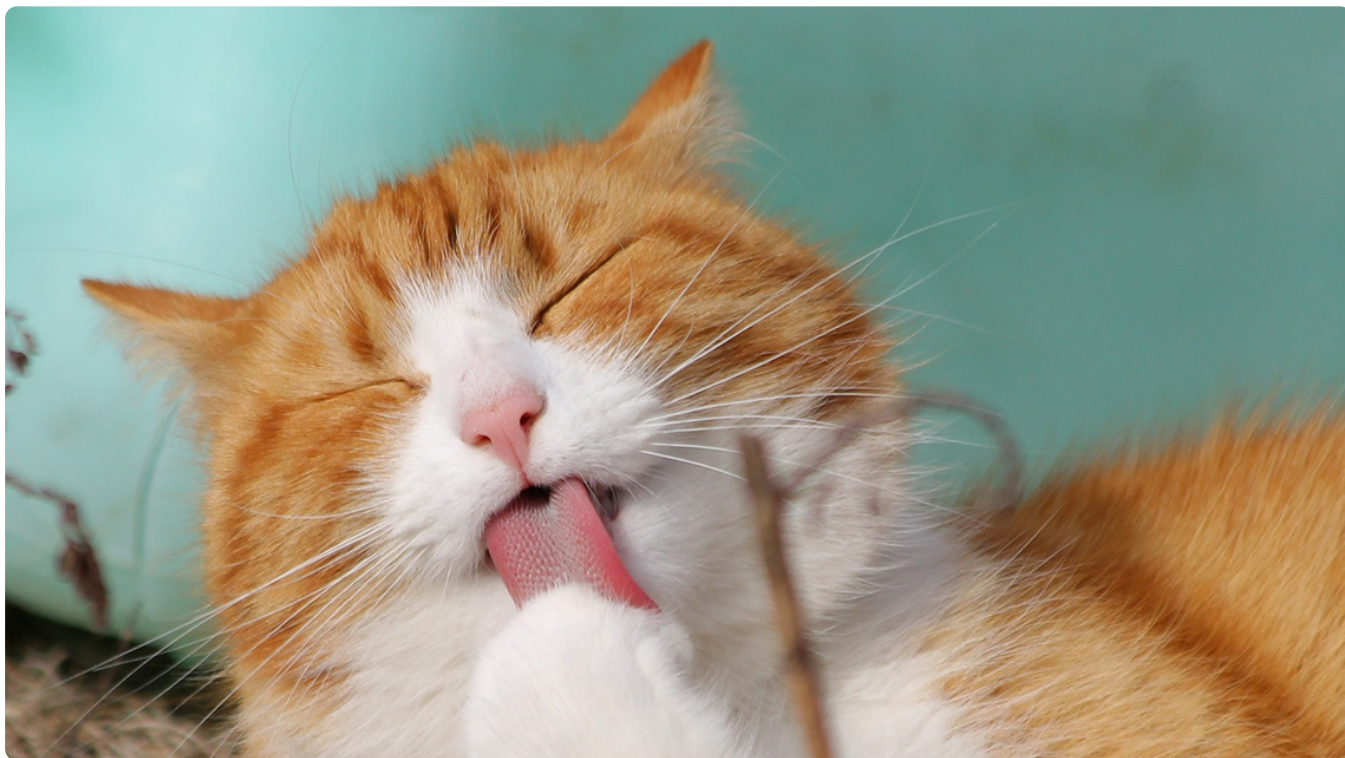


< 重学前端

首页 | 🔍

## 17 | JavaScript 执行（二）：闭包和执行上下文到底是怎么回事？

2019-02-26 winter



讲述：winter

时长 15:37 大小 14.32M



你好，我是 winter。

在上一课，我们了解了 JavaScript 执行中最粗粒度的任务：传给引擎执行的代码段。并且，我们还根据“由 JavaScript 引擎发起”还是“由宿主发起”，分成了宏观任务和微观任务，接下来我们继续去看一看更细的执行粒度。

一段 JavaScript 代码可能会包含函数调用的相关内容，从今天开始，我们就用两节课的时间来了解一下函数的执行。

我们今天要讲的知识在网上有不同的名字，比较常见的可能有：

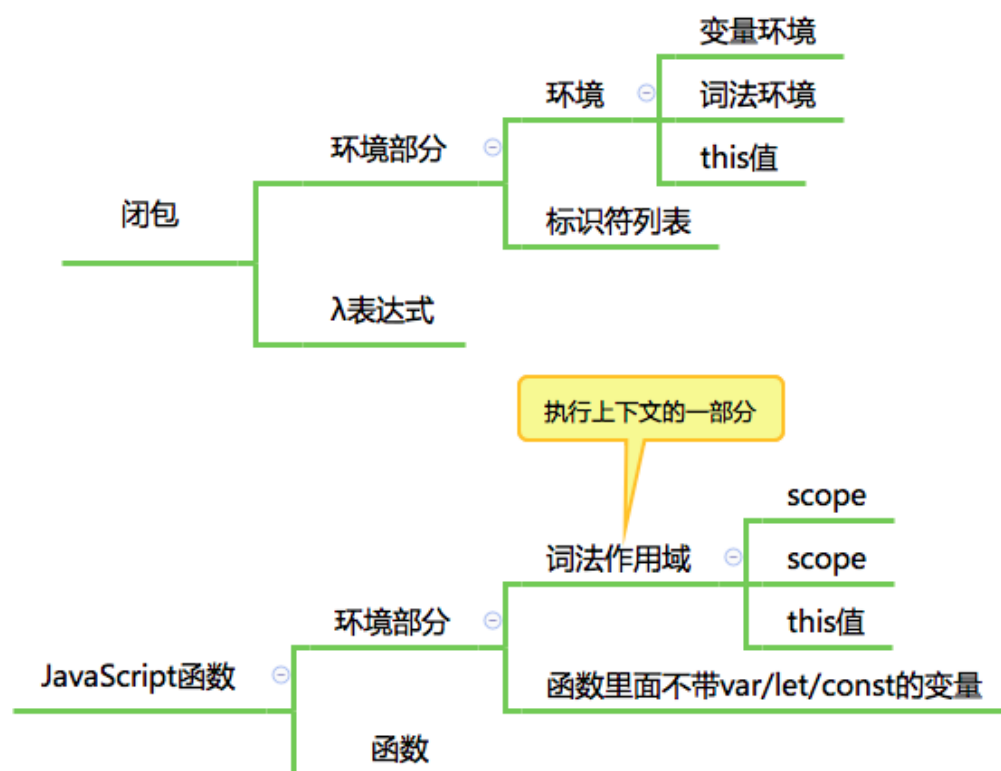
闭包；

作用域链；

执行上下文；

this 值。

实际上，尽管它们是表示不同的意思的术语，所指向的几乎是同一部分知识，那就是函数执行过程相关的知识。我们可以简单看一下图。



看着也许会有点晕，别着急，我会和你共同理一下它们之间的关系。

当然，除了让你理解函数执行过程的知识，理清这些概念也非常重要。所以我们先来讲讲这个有点复杂的概念：闭包。

## 闭包

闭包翻译自英文单词 closure，这是个不太好翻译的词，在计算机领域，它就有三个完全不同的意义：编译原理中，它是处理语法产生式的一个步骤；计算几何中，它表示包裹平面点集的凸多边形（翻译作凸包）；而在编程语言领域，它表示一种函数。

闭包这个概念第一次出现在 1964 年的《The Computer Journal》上，由 P. J. Landin 在《The mechanical evaluation of expressions》一文中提出了 applicative expression

## 和 closure 的概念。

More generally, this derived environment consists of  $E$ , modified by pairing the identifier(s) in  $bvX$  with corresponding components of the given argument  $x$  (and using the new value for preference if any variable

We shall not bother below to distinguish between  $E$  and  $E^*$ .

Also we represent the value of a  $\lambda$ -expression by a bundle of information called a “closure,” comprising

316

### Mechanical evaluation

the  $\lambda$ -expression and the environment relative to which it was evaluated. We must therefore arrange that such a bundle is correctly interpreted whenever it has to be applied to some argument. More precisely:

a *closure* has

an *environment part* which is a list whose two items are:

- (1) an environment
- (2) an identifier or list of identifiers,

and a *control part* which consists of a list whose sole item is an AE.

The value relative to  $E$  of a  $\lambda$ -expression  $X$  is represented

2. If  $C$  is not null, then  $hC$  is inspected, and:

- (2a) If  $hC$  is an identifier  $X$  (whose value relative to  $E$  occupies the position  $locationEX$  in  $E$ ), then  $S$  is replaced by

$locationEXE : S$

and  $C$  is replaced by  $\iota C$ . We describe this step as follows: “Scanning  $X$  causes  $locationEXE$  to be loaded.”

- (2b) If  $hC$  is a  $\lambda$ -expression  $X$ , scanning it causes the closure derived from  $E$  and  $X$  (as indicated above) to be loaded on to the stack.

Download

在上世纪 60 年代，主流的编程语言是基于 lambda 演算的函数式编程语言，所以这个最初的闭包定义，使用了大量的函数式术语。一个不太精确的描述是“带有一系列信息的  $\lambda$  表达式”。对函数式语言而言， $\lambda$  表达式其实就是函数。

我们可以这样简单理解一下，闭包其实只是一个绑定了执行环境的函数，这个函数并不是印在书本里的一条简单的表达式，闭包与普通函数的区别是，它携带了执行的环境，就像人在外星中需要自带吸氧的装备一样，这个函数也带有在程序中生存的环境。

这个古典的闭包定义中，闭包包含两个部分。

环境部分

环境

标识符列表

表达式部分

当我们把视角放在 JavaScript 的标准中，我们发现，标准中并没有出现过 closure 这个术语，但是，我们却不难根据古典定义，在 JavaScript 中找到对应的闭包组成部分。

## 环境部分

环境：函数的词法环境（执行上下文的一部分）

标识符列表：函数中用到的未声明的变量

表达式部分：函数体

至此，我们可以认为，JavaScript 中的函数完全符合闭包的定义。它的环境部分是函数词法环境部分组成，它的标识符列表是函数中用到的未声明变量，它的表达式部分就是函数体。

这里我们容易产生一个常见的概念误区，有些人会把 JavaScript 执行上下文，或者作用域（Scope，ES3 中规定的执行上下文的一部分）这个概念当作闭包。

实际上 JavaScript 中跟闭包对应的概念就是“函数”，可能是这个概念太过于普通，跟闭包看起来又没什么联系，所以大家才不自觉地把这个概念对应到了看起来更特别的“作用域”吧（其实我早年也是这么理解闭包，直到后来被朋友纠正，查了资料才改正过来）。

## 执行上下文：执行的基础设施

相比普通函数，JavaScript 函数的主要复杂性来自于它携带的“环境部分”。当然，发展到今天的 JavaScript，它所定义的环境部分，已经比当初经典的定义复杂了很多。

JavaScript 中与闭包“环境部分”相对应的术语是“词法环境”，但是 JavaScript 函数比  $\lambda$  函数要复杂得多，我们还要处理 this、变量声明、with 等等一系列的复杂语法， $\lambda$  函数中可没有这些东西，所以，在 JavaScript 的设计中，词法环境只是 JavaScript 执行上下文的一部分。

JavaScript 标准把一段代码（包括函数），执行所需的所有信息定义为：“执行上下文”。

因为这部分术语经历了比较多的版本和社区的演绎，所以定义比较混乱，这里我们先来理一下 JavaScript 中的概念。

**执行上下文在 ES3 中**，包含三个部分。

scope：作用域，也常常被叫做作用域链。

variable object：变量对象，用于存储变量的对象。

this value：this 值。

**在 ES5 中**，我们改进了命名方式，把执行上下文最初的三个部分改为下面这个样子。

lexical environment：词法环境，当获取变量时使用。

variable environment：变量环境，当声明变量时使用。

this value：this 值。

**在 ES2018 中**，执行上下文又变成了这个样子，this 值被归入 lexical environment，但是增加了不少内容。

lexical environment：词法环境，当获取变量或者 this 值时使用。

variable environment：变量环境，当声明变量时使用

code evaluation state：用于恢复代码执行位置。

Function：执行的任务是函数时使用，表示正在被执行的函数。

ScriptOrModule：执行的任务是脚本或者模块时使用，表示正在被执行的代码。

Realm：使用的基础库和内置对象实例。

Generator：仅生成器上下文有这个属性，表示当前生成器。

我们在这里介绍执行上下文的各个版本定义，是考虑到你可能会从各种网上的文章中接触这些概念，如果不把它们理清楚，我们就很难分辨对错。如果是我们自己使用，我建议统一使用最新的 ES2018 中规定的术语定义。

尽管我们介绍了这些定义，但我并不打算按照 JavaScript 标准的思路，从实现的角度去介绍函数的执行过程，这是不容易被理解的。

我想试着从代码实例出发，跟你一起推导函数执行过程中需要哪些信息，它们又对应着执行上下文中的哪些部分。

比如，我们看以下的这段 JavaScript 代码：

```
1 var b = {}  
2 let c = 1  
3 this.a = 2;
```

要想正确执行它，我们需要知道以下信息：

1. var 把 b 声明到哪里；
2. b 表示哪个变量；
3. b 的原型是哪个对象；
4. let 把 c 声明到哪里；
5. this 指向哪个对象。

这些信息就需要执行上下文来给出了，这段代码出现在不同的位置，甚至在每次执行中，会关联到不同的执行上下文，所以，同样的代码会产生不一样的行为。

在这两篇文章中，我会基本覆盖执行上下文的组成部分，本篇我们先讲 var 声明与赋值，let，realm 三个特性来分析上下文提供的信息，分析执行上下文中提供的信息。

## var 声明与赋值


我们来分析一段代码：

```
1 var b = 1
```

通常我们认为它声明了 b，并且为它赋值为 1，var 声明作用域函数执行的作用域。也就是说，var 会穿透 for、if 等语句。


在只有 var，没有 let 的旧 JavaScript 时代，诞生了一个技巧，叫做：立即执行的函数表达式（IIFE），通过创建一个函数，并且立即执行，来构造一个新的域，从而控制 var 的范围。

由于语法规定了 function 关键字开头是函数声明，所以要想让函数变成函数表达式，我们必须得加点东西，最常见的做法是加括号。

 复制代码


```
1 (function(){
2     var a;
3     //code
4 }());
5
6
7 (function(){
8     var a;
9     //code
10 }());
```

但是，括号有个缺点，那就是如果上一行代码不写分号，括号会被解释为上一行代码最末的函数调用，产生完全不符合预期，并且难以调试的行为，加号等运算符也有类似的问题。所以一些推荐不加分号的代码风格规范，会要求在括号前面加上分号。

 复制代码

```
1 ;(function(){
2     var a;
3     //code
4 }())
5
6
7 ;(function(){
8     var a;
9     //code
10 }())
```

我比较推荐的写法是使用 `void` 关键字。也就是下面的这种形式。


 复制代码

```
1 void function(){
2     var a;
3     //code
4 }();
```

这有效避免了语法问题，同时，语义上 `void` 运算表示忽略后面表达式的值，变成 `undefined`，我们确实不关心 IIFE 的返回值，所以语义也更为合理。



值得注意的是，有时候 var 的特性会导致声明的变量和被赋值的变量是两个 b，JavaScript 中有特例，那就是使用 with 的时候：

 复制代码

```
1 var b;
2 void function(){
3     var env = {b:1};
4     b = 1;
5     console.log("In function b:", b);
6     with(env) {
7         var b = 1;
8         console.log("In with b:", b);
9     }
10 }();
11 console.log("Global b:", b);
```

在这个例子中，我们利用立即执行的函数表达式（IIFE）构造了一个函数的执行环境，并且在里面使用了我们一开头的代码。

可以看到，在 Global function with 三个环境中，b 的值都不一样，而在 function 环境中，并没有出现 var b，这说明 with 内的 var b 作用到了 function 这个环境当中。

var b = {} 这样一句对两个域产生了作用，从语言的角度是个非常糟糕的设计，这也是一些人坚定地反对在任何场景下使用 with 的原因之一。

## let

let 是 ES6 开始引入的新的变量声明模式，比起 var 的诸多弊病，let 做了非常明确的梳理和规定。

为了实现 let，JavaScript 在运行时引入了块级作用域。也就是说，在 let 出现之前，JavaScript 的 if for 等语句皆不产生作用域。

我简单统计了下，以下语句会产生 let 使用的作用域：

for ;

if ;



```
switch ;  
  
try/catch/finally。
```

## Realm

在最新的标准（9.0）中，JavaScript 引入了一个新概念 Realm，它的中文意思是“国度”“领域”“范围”。这个英文的用法就有点比喻的意思，几个翻译都不太适合 JavaScript 语境，所以这里就不翻译啦。

我们继续来看这段代码：

 复制代码


```
1 var b = {}
```

在 ES2016 之前的版本中，标准中甚少提及{}的原型问题。但在实际的前端开发中，通过 iframe 等方式创建多 window 环境并非罕见的操作，所以，这才促成了新概念 Realm 的引入。

Realm 中包含一组完整的内置对象，而且是复制关系。

对不同 Realm 中的对象操作，会有一些需要格外注意的问题，比如 instanceof 几乎是失效的。

以下代码展示了在浏览器环境中获取来自两个 Realm 的对象，它们跟本土的 Object 做 instanceof 时会产生差异：

 复制代码

```
1 var iframe = document.createElement('iframe')  
2 document.documentElement.appendChild(iframe)  
3 iframe.src="javascript:var b = {};"  
4  
5 var b1 = iframe.contentWindow.b;  
6 var b2 = {};  
7  
8 console.log(typeof b1, typeof b2); //object object  
9  
10 console.log(b1 instanceof Object, b2 instanceof Object); //false true
```

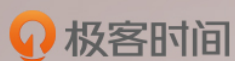
可以看到，由于 b1、 b2 由同样的代码 “ {} ” 在不同的 Realm 中执行，所以表现出了不同的行为。

## 结语

在今天的课程中，我帮你梳理了一些概念：有编程语言的概念闭包，也有各个版本中的 JavaScript 标准中的概念：执行上下文、作用域、this 值等等。

之后我们又从代码的角度，分析了一些执行上下文中所需要的信息，并从 var、let、对象字面量等语法中，推导出了词法作用域、变量作用域、Realm 的设计。

最后留给你一个问题：你喜欢使用 let 还是 var？听过今天的课程，你的想法是否有改变呢？为什么？



# 重学前端

每天 10 分钟，重构你的前端知识体系

winter 程劭非  
前手机淘宝前端负责人



新版升级：点击「👤请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得转载

上一篇 16 | JavaScript 执行（一）：Promise 里的代码为什么比 setTimeout 先执行？

下一篇 18 | JavaScript 执行（三）：你知道现在有多少种函数吗？

## 精选留言 (13)

[写留言](#)**Geek\_56013...**

2019-02-27

13

老师您的专业知识太强了，文中包含很多专业术语，在介绍某专业术语时带上了其他专业术语，而这些带上的专业术语部分在网上搜也是解释不清，导致很多地方看不懂、看起来比较费劲、只能猜测大意。比如对于「realm」的描述，只提了中文意思是“国度”“领域”“范围”和“包含一组完整的内置对象，而且是复制关系”，看完文章后，在js领域还是不清楚具体「realm」是什么含义，只能大概猜测。希望老师后续文章如果解释某...  
展开 ∨

**张祥儒**

2019-02-26

3

winter大大，我觉得应该用global object，和active object 来解释这个闭包，作用域，执行器上下文。

作者回复: 这是ES3里的解释法，现在已经解释不了很多语法了。

**乃乎**

2019-02-26

2

更喜欢 const 哈哈

展开 ∨

**麦哲伦**

2019-02-27

1

老师能解释下这个么？

```
var b = 10;
```

```
(function b(){
```

```
b = 20;
```

```
console.log(b); // [Function: b]...
```

展开 ∨

作者回复: 这个地方比较特殊, "具有名称的函数表达式"会在外层词法环境和它自己执行产生的词法环境之间产生一个词法环境, 再把自己的名称和值当作变量塞进去, 所以你这里的  $b = 20$  并没有改变外面的  $b$ , 而是试图改变一个只读的变量  $b$ 。

这块知识有点偏, 随便看看就好。



Carson

2019-02-26

👍 1

今天是自己第一次结构性整理清楚 JavaScript 的函数部分。原来它除了函数体之外, 还包括了函数所处的环境, 而其中的词法环境, 其实只是执行上下文七个部分中的一支。

个人感觉 `var` 声明在不同的执行上下文中相对 `let` 更容易出错, 同时也会增加冗余的临时变量。比如在 `for loop` 中, 会遇到需要为不同的 `loop` 声明 `i`、`j`、`k` 变量。...

展开 ∨



zhbo

2019-03-04

👍

老师, 我不懂这个地方说的:。。相比普通函数, JavaScript 函数的主要复杂性来自于它携带的“环境部分”。什么算普通函数, 什么是js函数



追梦

2019-03-01

👍

`async/await`是否可以理解为`promise`和`generator`的语法糖? 用`promise`和`generator`是可以模拟`async/await`的行为的



九

2019-02-28

👍

吸氧装备的比喻好玩

展开 ∨



追梦

2019-02-28

👍

请问老师，这个例子，是如何得出“可以看到，在 Global function with 三个环境中，b 的值都不一样”这个结论的？

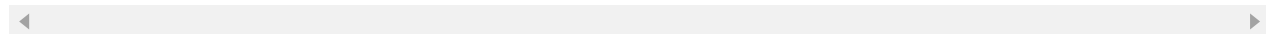
```
var b;
```

```
void function(){
```

```
    var env = {b:1};...
```

展开 ▾

作者回复: 这里写错了一个地方，我改一下



**Rushan-Ch...**

2019-02-27



var 声明与赋值，`var b = 1` 下面的 “var 声明作用域函数执行的作用域”，这句没看明白😓

展开 ▾



**无羨**

2019-02-27



> Before it is evaluated, all ECMAScript code must be associated with a realm. Conceptually, a realm consists of a set of intrinsic objects, an ECMAScript global environment, all of the ECMAScript code that is loaded within the scope of that global environment, and other associated state and resources.



**芳玥**

2019-02-26



用let更多一点

展开 ▾



**@Tc彭于晏**

2019-02-26



最后一个代码片段，中的b1 b2 o1 o2

展开 ▾

