我从来不理解JavaScript闭包，直到有人这样向我解释它

译者：前端小智

> 原文：https://medium.com/dailyjs/i-never-understood-javascript-closures-9663703368e8

> **点赞再看**，微信搜索 **【大迁世界】** 关注这个没有大厂背景，但有着一股向上积极心态人。本文 GitHub https://github.com/qq449245884/xiaozhi 上已经收录，文章的已分类，也整理了很多我的文档，和教程资料。

大家都说简历没项目写，我就帮大家找了一个项目，还附赠【搭建教程】。

正如标题所述，JavaScript闭包对我来说一直有点神秘,看过很多闭包的文章，在工作使用过闭包，有时甚至在项目中使用闭包，但我确实是这是在使用闭包的知识。

最近看到的一些文章，终于，有人用于一种让我明白方式对闭包进行了解释，我将在本文中尝试使用这种方法来解释闭包。

准备

在理解闭包之前，有个重要的概念需要先了解一下，就是 js 执行上下文。

这篇文章是执行上下文 很不错的入门教程，文章中提到：

> 当代码在JavaScript中运行时，执行代码的环境非常重要，并将概括为以下几点：
>
> **全局作用域**——第一次执行代码的默认环境。
>
> **函数作用域**——当执行流进入函数体时。
>
> (...) —— 我们当作 执行上下文 是当前代码执行的一个环境与作

用域。

换句话说，当我们启动程序时，我们从全局执行上下文中开始。一些变量是在全局执行上下文中声明的。我们称之为全局变量。当程序调用一个函数时，会发生什么?

以下几个步骤:

1. JavaScript创建一个新的执行上下文，我们叫作本地执行上下文。
2. 这个本地执行上下文将有它自己的一组变量，这些变量将是这个执行上下文的本地变量。
3. 新的执行上下文被推到到执行堆栈中。可以将执行堆栈看作是一种保存程序在其执行中的位置的容器。

函数什么时候结束?当它遇到一个return语句或一个结束括号}。

当一个函数结束时，会发生以下情况:

1. 这个本地执行上下文从执行堆栈中弹出。

2. 函数将返回值返回调用上下文。调用上下文是调用这个本地的执行上下文，它可以是全局执行上下文，也可以是另外一个本地的执行上下文。这取决于调用执行上下文来处理此时的返回值，返回的值可以是一个对象、一个数组、一个函数、一个布尔值等等，如果函数没有return语句，则返回undefined。

3. 这个本地执行上下文被销毁，销毁是很重要，这个本地执行上下文中声明的所有变量都将被删除，不在有变量，这个就是为什么 称为本地执行上下文中自有的变量。

基础的例子

在讨论闭包之前，让我们看一下下面的代码:

```
1: let a = 3
2: function addTwo(x) {
```

```
3:   let ret = x + 2
4:   return ret
5: }
6: let b = addTwo(a)
7: console.log(b)
```

为了理解JavaScript引擎是如何工作的，让我们详细分析一下：

1. 在第1行，我们在全局执行上下文中声明了一个新变量a，并将赋值为3。

2. 接下来就变得棘手了，第2行到第5行实际上是在一起的。这里发生了什么? 我们在全局执行上下文中声明了一个名为addTwo的新变量,我们给它分配了什么?一个函数定义。两个括号{}之间的任何内容都被分配给addTwo，函数内部的代码没有被求值，没有被执行，只是存储在一个变量中以备将来使用。

3. 现在我们在第6行。它看起来很简单，但是这里有很多东西需要拆开分析。首先，我们在全局执行上下文中声明一个新变量，并将其标记为b，变量一经声明，其值即为undefined。

4. 接下来，仍然在第6行，我们看到一个赋值操作符。我们准备给变量b赋一个新值，接下来我们看到一个函数被调用。当看到一个变量后面跟着一个圆括号(...)时，这就是调用函数的信号，接着，每个函数都返回一些东西(值、对象或 undefined)，无论从函数返回什么，都将赋值给变量b。

5. 但是首先我们需要调用addTwo的函数。JavaScript将在其全局执行上下文内存中查找名为addTwo的变量。噢，它找到了一个，它是在步骤2(或第2 - 5行)中定义的。变量add2包含一个函数定义。注意，变量a作为参数传递给函数。JavaScript在全局执行上下文内存中搜索变量a，找到它，发现它的值是3，并将数字3作为参数传递给函数，准备好执行函数。

6. 现在执行上下文将切换,创建了一个新的本地执行上下文，我们将其命名为“addTwo执行上下文”，执行上下文被推送到调用堆栈

上。在addTwo执行上下文中，我们要做的第一件事是什么?

7. 你可能会说，“在addTwo执行上下文中声明了一个新的变量ret”，这是不对的。正确的答案是，我们需要先看函数的参数。在addTwo执行上下文中声明一个新的变量`x```，因为值3是作为参数传递的，所以变量x`被赋值为3。

8. 下一步是:在addTwo执行上下文中声明一个新的变量ret。它的值被设置为 undefined(第三行)。

9. 仍然是第3行，需要执行一个相加操作。首先我们需要x的值，JavaScript会寻找一个变量x，它会首先在addTwo执行上下文中寻找，找到了一个值为3。第二个操作数是数字2。两个相加结果为5就被分配给变量ret。

0. 第4行，我们返回变量ret的内容，在addTwo执行上下文中查找，找到值为5，返回，函数结束。

1. 第4-5行，函数结束。addTwo执行上下文被销毁，变量x和ret被释放，它们已经不存在了。addTwo 执行上下文从调用堆栈中弹出，返回值返回给调用上下文，在这种情况下，调用上下文是全局执行上下文，因为函数addTwo是从全局执行上下文调用的。

2. 现在我们继续第4步的内容，返回值5被分配给变量b，程序仍然在第6行。

3. 在第7行，b的值 5 被打印到控制台了。

对于一个非常简单的程序，这是一个非常冗长的解释，我们甚至还没有涉及闭包。但肯定会涉及的，不过首先我们得绕一两个弯。

词法作用域（Lexical scope）

我们需要理解词法作用域的一些知识。请看下面的例子：

```
1: let val1 = 2
```

```
2: function multiplyThis(n) {
3:   let ret = n * val1
4:   return ret
5: }
6: let multiplied = multiplyThis(6)
7: console.log('example of scope:', multiplied)
```

这里想说明，我们在函数执行上下文中有变量，在全局执行上下文中有变量。JavaScript的一个复杂之处在于它如何查找变量，如果在函数执行上下文中找不到变量，它将在调用上下文中寻找它，如果在它的调用上下文中没有找到，就一直往上一级，直到它在全局执行上下文中查找为止。(如果最后找不到，它就是 undefined)。

下面列出向个步骤来解释一下（如果你已经熟悉了，请跳过）：

1. 在全局执行上下文中声明一个新的变量val1，并将其赋值为2。

2. 第2-5行，声明一个新的变量 multiplyThis，并给它分配一个函数定义。

3. 第6行，声明一个在全局执行上下文 multiplied 新变量。

4. 从全局执行上下文内存中查找变量multiplyThis，并将其作为函数执行，传递数字 6 作为参数。

5. 新函数调用(创建新执行上下文），创建一个新的 multiplyThis 函数执行上下文。

6. 在 multiplyThis 执行上下文中，声明一个变量n并将其赋值为6。

7. 第 3 行。在multiplyThis执行上下文中，声明一个变量ret。

8. 继续第 3 行。对两个操作数 n 和 val1 进行乘法运算.
   在multiplyThis执行上下文中查找变量 n。我们在步骤6中声明了它,它的内容是数字6。在multiplyThis执行上下文中查找变量val1。multiplyThis执行上下文没有一个标记为 val1 的变量。我们向调用上下文查找，调用上下文是全局执行上下文，在全局执行

上下文中寻找 val1。哦，是的、在那儿，它在步骤1中定义，数值是2。

9. 继续第 3 行。将两个操作数相乘并将其赋值给ret变量，6 * 2 = 12，ret 现在值为 12。

0. 返回ret变量，销毁multiplyThis执行上下文及其变量 ret 和 n 。变量 val1 没有被销毁，因为它是全局执行上下文的一部分。

1. 回到第6行。在调用上下文中，数字 12 赋值给 multiplied 的变量。

2. 最后在第7行，我们在控制台中打印 multiplied 变量的值

在这个例子中，我们需要记住一个函数可以访问在它的调用上下文中定义的变量，这个就是**词法作用域（Lexical scope）**。

返回函数的函数

在第一个例子中，函数addTwo返回一个数字。请记住，函数可以返回任何东西。让我们看一个返回函数的函数示例，因为这对于理解闭包非常重要。看栗子：

```
1: let val = 7
2: function createAdder() {
3:   function addNumbers(a, b) {
4:     let ret = a + b
5:     return ret
6:   }
7:   return addNumbers
8: }
9: let adder = createAdder()
10: let sum = adder(val, 8)
11: console.log('example of function returning a function: ', sum)
```

让我们回到分步分解：

1. 第1行。我们在全局执行上下文中声明一个变量val并赋值为 7。

2. 第 2-8 行。我们在全局执行上下文中声明了一个名为 createAdder 的变量，并为其分配了一个函数定义。第3-7行描述了上述函数定义，和以前一样，在这一点上，我们没有直接讨论这个函数。我们只是将函数定义存储到那个变量(createAdder)中。

3. 第9行。我们在全局执行上下文中声明了一个名为 adder 的新变量，暂时，值为 undefined。

4. 第9行。我们看到括号()，我们需要执行或调用一个函数，查找全局执行上下文的内存并查找名为createAdder 的变量，它是在步骤2中创建的。好吧，我们调用它。

5. 调用函数时，执行到第2行。创建一个新的createAdder执行上下文。我们可以在createAdder的执行上下文中创建自有变量。js 引擎将createAdder的上下文添加到调用堆栈。这个函数没有参数，让我们直接跳到它的主体部分.

6. 第 3-6 行。我们有一个新的函数声明，我们在createAdder执行上下文中创建一个变量addNumbers。这很重要，addnumber只存在于createAdder执行上下文中。我们将函数定义存储在名为 ``addNumbers``` 的自有变量中。

7. 第7行，我们返回变量addNumbers的内容。js引擎查找一个名为addNumbers的变量并找到它，这是一个函数定义。好的，函数可以返回任何东西，包括函数定义。我们返addNumbers的定义。第4行和第5行括号之间的内容构成该函数定义。

8. 返回时，createAdder执行上下文将被销毁。addNumbers 变量不再存在。但addNumbers函数定义仍然存在，因为它返回并赋值给了adder 变量。

9. 第10行。我们在全局执行上下文中定义了一个新的变量 sum，先赋值为 undefined;

0. 接下来我们需要执行一个函数。哪个函数? 是名为adder变量中定义的函数。我们在全局执行上下文中查找它，果然找到了它，这个函数有两个参数。

1. 让我们查找这两个参数，第一个是我们在步骤1中定义的变量val，它表示数字7，第二个是数字8。

2. 现在我们要执行这个函数，函数定义概述在第3-5行，因为这个函数是匿名，为了方便理解，我们暂且叫它adder吧。这时创建一个adder函数执行上下文，在adder执行上下文中创建了两个新变量 a 和 b。它们分别被赋值为 7 和 8，因为这些是我们在上一步传递给函数的参数。

3. 第 4 行。在adder执行上下文中声明了一个名为ret的新变量,

4. 第 4 行。将变量a的内容和变量b的内容相加得15并赋给ret变量。

5. ret变量从该函数返回。这个匿名函数执行上下文被销毁，从调用堆栈中删除，变量a、b和ret不再存在。

6. 返回值被分配给我们在步骤9中定义的sum变量。

7. 我们将sum的值打印到控制台。

8. 如预期，控制台将打印15。我们在这里确实经历了很多困难，我想在这里说明几点。首先，函数定义可以存储在变量中，函数定义在程序调用之前是不可见的。其次，每次调用函数时，都会(临时)创建一个本地执行上下文。当函数完成时，执行上下文将消失。函数在遇到return或右括号}时执行完成。

最后,一个闭包

看看下面的代码，并试着弄清楚会发生什么。

```
1: function createCounter() {
2:   let counter = 0
3:   const myFunction = function() {
```

```
4:     counter = counter + 1
5:     return counter
6:   }
7:   return myFunction
8: }
9: const increment = createCounter()
10: const c1 = increment()
11: const c2 = increment()
12: const c3 = increment()
13: console.log('example increment', c1, c2, c3)
```

现在，我们已经从前两个示例中掌握了它的诀窍，让我们按照预期的方式快速执行它：

1. 第 1-8 行。我们在全局执行上下文中创建了一个新的变量createCounter，并赋值了一个的函数定义。

2. 第9行。我们在全局执行上下文中声明了一个名为increment的新变量。

3. 第9行。我们需要调用createCounter函数并将其返回值赋给increment变量。

4. 第 1-8行。调用函数，创建新的本地执行上下文。

5. 第2行。在本地执行上下文中，声明一个名为counter的新变量并赋值为 0;

6. 第 3-6行。声明一个名为myFunction的新变量，变量在本地执行上下文中声明,变量的内容是为第4行和第5行所定义。

7. 第7行。返回myFunction变量的内容，删除本地执行上下文。变量myFunction和counter不再存在。此时控制权回到了调用上下文。

8. 第9行。在调用上下文(全局执行上下文)中，createCounter返回的值赋给了increment，变量increment现在包含一个函数定义内容

为createCounter返回的函数。它不再标记为myFunction````，但它的定义是相同的。在全局上下文中，它是的标记为labeledincrement```。

9. 第10行。声明一个新变量 c1。

0. 继续第10行。查找increment变量，它是一个函数并调用它。它包含前面返回的函数定义，如第4-5行所定义的。

1. 创建一个新的执行上下文。没有参数，开始执行函数。

2. 第4行。counter=counter + 1。在本地执行上下文中查找counter变量。我们只是创建了那个上下文，从来没有声明任何局部变量。让我们看看全局执行上下文。这里也没有counter变量。Javascript会将其计算为counter = undefined + 1，声明一个标记为counter的新局部变量，并将其赋值为number 1，因为undefined被当作值为 0。

3. 第5行。我们变量counter的值 1，我们销毁本地执行上下文和counter变量。

4. 回到第10行。返回值1被赋给c1。

5. 第11行。重复步骤10-14，c2也被赋值为1。

6. 第12行。重复步骤10-14，c3也被赋值为1。

7. 第13行。我们打印变量c1 c2和c3的内容。

你自己试试，看看会发生什么。你会将注意到，它并不像从我上面的解释中所期望的那样记录1,1,1。而是记录1,2,3。这个是为什么?

不知怎么滴，increment函数记住了那个cunter的值。这是怎么回事?

counter是全局执行上下文的一部分吗?尝试 console.log(counter)，得到undefined的结果,显然不是这样的。

也许，当你调用increment时，它会以某种方式返回它创建的函数(createCounter)?这怎么可能呢?变量increment包含函数定义，而不是函数的来源，显然也不是这样的。

所以一定有另一种机制。**闭包**，我们终于找到了，丢失的那块。

它是这样工作的，无论何时声明新函数并将其赋值给变量，都要存储函数定义和闭包。闭包包含在函数创建时作用域中的所有变量，它类似于背包。函数定义附带一个小背包，它的包中存储了函数定义创建时作用域中的所有变量。

所以我们上面的解释都是错的，让我们再试一次，但是这次是正确的。

```
1: function createCounter() {
2:   let counter = 0
3:   const myFunction = function() {
4:     counter = counter + 1
5:     return counter
6:   }
7:   return myFunction
8: }
9: const increment = createCounter()
10: const c1 = increment()
11: const c2 = increment()
12: const c3 = increment()
13: console.log('example increment', c1, c2, c3)
```

1. 同上，第1-8行。我们在全局执行上下文中创建了一个新的变量createCounter，它得到了指定的函数定义。

2. 同上，第9行。我们在全局执行上下文中声明了一个名为increment的新变量。

3. 同上，第9行。我们需要调用createCounter函数并将其返回值赋给increment变量。

4. 同上，第1-8行。调用函数，创建新的本地执行上下文。

5. 同上，第2行。在本地执行上下文中，声明一个名为counter的新变量并赋值为 0 。

6. 第3-6行。声明一个名为myFunction的新变量，变量在本地执行上下文中声明,变量的内容是另一个函数定义。如第4行和第5行所定义，现在我们还创建了一个闭包，并将其作为函数定义的一部分。闭包包含作用域中的变量，在本例中是变量counter(值为0)。

7. 第7行。返回myFunction变量的内容,删除本地执行上下文。myFunction和counter不再存在。控制权交给了调用上下文，我们返回函数定义和它的闭包，闭包中包含了创建它时在作用域内的变量。

8. 第9行。在调用上下文(全局执行上下文)中，createCounter返回的值被指定为increment，变量increment现在包含一个函数定义(和闭包),由createCounter返回的函数定义,它不再标记为myFunction，但它的定义是相同的,在全局上下文中，称为increment。

9. 第10行。声明一个新变量c1。

0. 继续第10行。查找变量increment，它是一个函数，调用它。它包含前面返回的函数定义,如第4-5行所定义的。(它还有一个带有变量的闭包)。

1. 创建一个新的执行上下文，没有参数，开始执行函数。

2. 第4行。counter = counter + 1，寻找变量 counter，在查找本地或全局执行上下文之前，让我们检查一下闭包，瞧，闭包包含一个名为counter的变量，其值为0。在第4行表达式之后，它的值被设置为1。它再次被储存在闭包里，闭包现在包含值为1的变量counter。

3. 第5行。我们返回counter的值，销毁本地执行上下文。

4. 回到第10行。返回值1被赋给变量c1。

5. 第11行。我们重复步骤10-14。这一次，在闭包中此时变量counter的值是1。它在第12行设置的，它的值被递增并以2的形式存储在递增函数的闭包中,c2被赋值为2。

6. 第12行。重复步骤10-14行,c3被赋值为3。

7. 第13行。我们打印变量c1 c2和c3的值。

你可能会问，是否有任何函数具有闭包，甚至是在全局范围内创建的函数?答案是肯定的。在全局作用域中创建的函数创建闭包，但是由于这些函数是在全局作用域中创建的，所以它们可以访问全局作用域中的所有变量，闭包的概念并不重要。

当函数返回函数时，闭包的概念就变得更加重要了。返回的函数可以访问不属于全局作用域的变量，但它们仅存在于其闭包中。

闭包不是那么简单

有时候闭包在你甚至没有注意到它的时候就会出现，你可能已经看到了我们称为部分应用程序的示例，如下面的代码所示：

```
let c = 4
const addX = x => n => n + x
const addThree = addX(3)
let d = addThree(c)
console.log('example partial application', d)
```

如果箭头函数让你感到困惑，下面是同样效果：

```
let c = 4
function addX(x) {
  return function(n) {
     return n + x
  }
}
```

```
const addThree = addX(3)
let d = addThree(c)
console.log('example partial application', d)
```

我们声明一个能用加法函数addX，它接受一个参数x并返回另一个函数。返回的函数还接受一个参数并将其添加到变量x中。

变量x是闭包的一部分，当变量addThree在本地上下文中声明时，它被分配一个函数定义和一个闭包，闭包包含变量x。

所以当addThree被调用并执行时，它可以从闭包中访问变量x以及为参数传递变量n并返回两者的和 7。

总结

我将永远记住闭包的方法是通过背包的类比。当一个函数被创建并传递或从另一个函数返回时，它会携带一个背包。背包中是函数声明时作用域内的所有变量。

交流

文章每周持续更新，可以微信搜索「 大迁世界 」第一时间阅读和催更（比博客早一到两篇哟），本文 GitHub https://github.com/qq449245884/xiaozhi 已经收录，整理了很多我的文档，欢迎Star和完善，大家面试可以参照考点复习，另外关注公众号，后台回复**福利**，即可看到福利，你懂的。