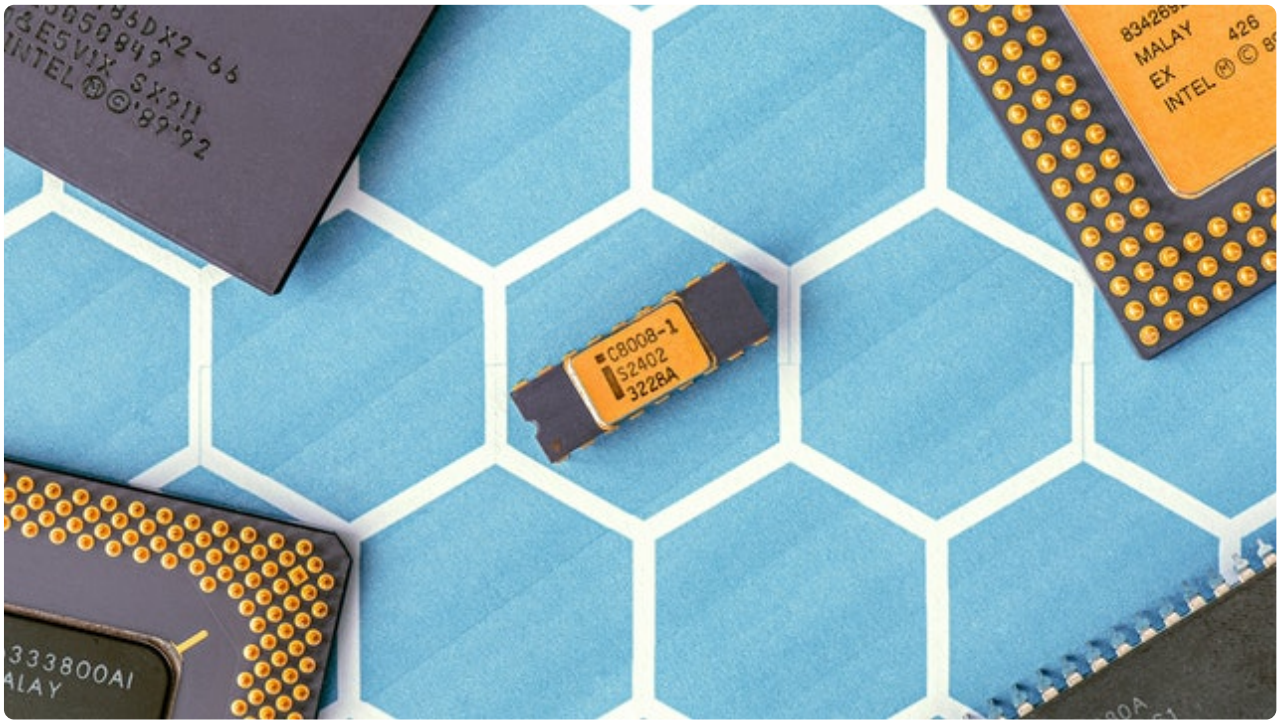
02 闭包—从编译原理的角度理解作用域

更新时间：2020-05-13 09:55:49



> 当你做成功一件事，千万不要等待着享受荣誉，应该再做那些需要的事。 —— 巴斯德

面试官视角的“闭包”

闭包问题不好答，原因往往不在于闭包本身有多么晦涩神秘，而在于闭包的背后有太多太多的故事可以深挖。面试经历稍微丰富一些的同学会发现，大多数面试场景下，面试官不会直接问你“闭包是什么”，而是会直接甩出来一套代码片段给你，问“这段代码的运行结果是什么？”。遇到这种情况，大家真要在心里松一口气了，毕竟直接脑内跑代码。是闭包最温和、痛苦程度最小的一种考察方式。针对这种考察方式，我们本节也会有习题给到大家来做。但除此之外，我希望大家能对另一种提问方式引起注意 ——“你如何理解 JavaScript 中的闭包”？

这个问题的区分度非常高。它的特点就是人人都能答上那么一两句，但只有很少的人可以答好。而且它往往不作为一个孤立的问题出现，而是作为一些更加深入的问题的“引子”。

当面试官问你“如何理解 ”的时候，他大概率并不是想听你背诵“闭包是一种 xxxxxx 的函数”，而是想跟你聊聊**作用域、作用域链**等触及 JS 语言核心的一些知识点，聪明的面试官，还会借机引出**变量提升、暂时性死区、执行上下文**等附加话题，甚至想问问你 **JS 中的不同异常之间本质的区别在哪里？词法作用域模型又是啥？**

所以说，想答好闭包问题，能看懂闭包代码只是登堂入室的第一步。问题的关键，在于闭包背后的这些故事。接下来，我们就抽丝剥茧，一点一点把这些看似高深的问题肢解掉，帮助各位从根儿上掌握闭包所牵扯出来的这一整块的知识脉络。

理解作用域的实现机制

大家知道，几乎每一种编程语言，它最基本的能力都是**能够存储变量当中的值、并且允许我们对这个变量的值进行访问和修改**。那么有了变量之后，应该把它放在那里、程序如何找到它们？这是不是需要我们提前约定好一套存储变量、访问变量的规则？这套规则，就是我们常说的**作用域**。更多时候，我们提到作用域的时候，指的是这个规则约束下的一个变量、函数、标识符可以被访问的区域（这时它就更具体了）。

要想理解作用域的实现机制，我们需要结合 JS 的编译原理一起来看（别跑，这块不难）。

我们来看一个简单的声明语句：

```
var name = 'xiuyan'
```

大家觉得 JS 会怎么理解这句“话”呢？

在我们看来，这只是一个声明语句。但是在 JS 引擎眼里，它却包含了两个声明：

- var name （编译时处理）
- name = ‘xiuyan’（运行时处理）

何为编译时、何为运行时？难道 JS 不是不存在编译阶段的“动态语言”吗？

事实上，JS 也是有编译阶段的，它和传统语言的区别在于，JS 不会早早地把编译工作做完，而是一边编译一边执行。简单来说，所有的 JS 代码片段在执行之前都会被编译，只是这个编译的过程非常短暂（可能就只有几微妙、或者更短的时间），紧接着这段代码就会被执行。

回到我们这个语句上来，我们来看看编译阶段和执行阶段阶段都发生了什么事情：

- 编译阶段： 这时登场的是一个叫 **编译器** 的家伙。编译器会找遍当前作用域，看看是不是已经有一个叫 name 的家伙了。如果有，那么就忽略 var name 这个声明，继续编译下去；如果没有，则在当前作用域里新增一个 name。然后，编译器会为引擎生成运行时所需要的代码，程序就进入了执行阶段
- 执行阶段： 这时登场的就是大家常常听到的 **JS 引擎** 了。JS 引擎在执行代码的时候，仍然会找遍当前作用域，看看是不是有一个叫 name 的家伙。如果能找到，那么万事大吉，我来给你赋值。如果找不到，它也不会灰心，它会从当前作用域里“探出头去”，看看“外面”有没有，或者“外面的外面”有没有。如果最终仍然找不到 name 变量，引擎就会抛出一个异常。

这里出现了一个有趣的东西，就是我们引擎的**查找过程** —— 何谓**探出头去**？何谓“**外面**”呢？这就引出了我们 JS 作用域里一个非常重要的概念 —— 作用域链。

作用域套作用域，就有了作用域链

现在我们已经知道，作用域本质上就是程序**存储和访问变量的规则**。上个小节，我们聊过了作用域在 JS 这门语言中的实现机制。现在，我们来看看，这套规则的内容具体是怎么回事儿。

在 JS 世界中，目前已经有了三种作用域：

- 全局作用域
- 函数作用域
- 块作用域

我们先来通过例子唤醒一下大家大脑里关于这三种作用域的记忆：

全局作用域

声明在任何函数之外的顶层作用域的变量就是全局变量，这样的变量拥有全局作用域：

```
var name = 'xiuyan'; // 全局作用域内的变量

// 函数作用域
function showName() {
    console.log(name);
}

// 块作用域
{
  name = 'BigBear'
}

showName(); // 输出 'BigBear'
```

上面这个例子我们可以看出，全局变量在全局作用域、函数作用域和块作用域里都可以获取到～

函数作用域

在函数内部定义的变量，拥有函数作用域。

```
var name = 'xiuyan'; // name 是全局变量
function showName(myName) {
  // myName 是传入 showName 的局部变量
  console.log(myName);
}
function sayHello() {
  // hello 被定义成局部作用域变量
  var helloString = 'hello everyone';
  console.log(helloString);
}


showName(name); // 输出 'xiuyan'
sayHello(); // 输出 'hello everyone'
console.log(myName); // 抛出错误：myName 在全局作用域未定义
console.log(helloString); // 抛出错误：hello 在全局作用域未定义

{
  console.log(helloString, myName) // 抛出错误
}
```

在这个例子里，myName 和 hello 都是在函数内部定义的变量，它们就被“画地为牢”，作用域仅局限于函数内部。全局作用域和块作用域里都访问不到它们。

块作用域

ES6 开始，我们迎来了了两个用于声明变量的新关键词: `let` 和 `const`。这两个关键字定义的变量，如果被一个大括号 { } 这样括住了，那么这个大括号就是一个代码块，大括号括住的这些变量就形成了一个块作用域：

```
{
  let a = 1;
  console(a);
}

 console(a); // 报错

function showA() {
  console.log(a) // 报错
}
```

在这个例子里我们可以看出，块作用域内的变量只要出了自己被定义的那个代码块，那么就无法访问了。这点和函数作用域比较相似 —— 它们都只在“自己的地盘”上生效，所以它们也统称为” 局部作用域 “。

作用域链

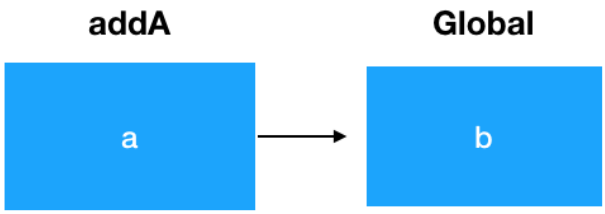
OK，现在我们完成了作用域类型的面面观，话题回到作用域链上。在我们实际开发中，通常不止用到一种作用域。当一个块或者一个函数嵌套在另一个块或者函数中时，就发生了作用域的嵌套。比如这样：

```
function addA(a) {
  console.log(a + b)
  console.log(c) // 报错
}

var b = 1

addA(2) //3
```

在这个例子中，有两个作用域：addA 的函数作用域和全局作用域。它们的关系示意如下：

我们试图在 addA 这个函数里访问变量 b 的时候，考虑到函数作用域内并没有对 b、c 这两个变量作定义，所以一开始肯定是找不到的。要想找到 b、c ，该怎么做？就是我们上文提到的“探出头去”，对吧？探出头去，去上层作用域（全局作用域找），找到了 b ，那么就可以直接拿来用了；没找到 c，并且全局作用域已经没有上层作用域了（头探不出去了），那就歇菜，报错！这就是上文“执行阶段 ”里我们描述的那个过程。

在这个查找过程中，层层递进的作用域，就形成了一条作用域链。上面这个例子里，作用域链比较短：



理解闭包

我们再来看一个例子：

```
function addABC(){
  var a = 1,b = 2;

  function add(){
    return a+b+c;
  }
  return add;
}

var c = 3

var globalAdd = addABC()

console.log(globalAdd()) // 6
```

在这个例子里，作用域嵌套的情况展示如下：

作用域链关系展示如下：

其中 add 这个函数，它嵌套在函数 addABC 的内部，想要查找 a、b、c 三个变量，它得去上层的 addABC 作用域里找，对吧？像 a、b、c 这样**在函数中被使用，但它既不是函数参数、也不是函数的局部变量，而是一个不属于当前作用域的变量，此时它相对于当前作用域来说，就是一个自由变量**。而像 add 这样**引用了自由变量的函数，就叫闭包。**

对于闭包，大家如果能理解到这个程度，面试时已经不会在这上面吃亏了。可以说上面这个版本的闭包定义，大家面试的时候答出来，就是 100 分。但是如果你想要进大厂、想要进好团队，那么 100 分可能还不太够，需要 120 分。120 的答案，我们会在下一节用整整一节的时间给大家展开讲。

加餐：LHS、RHS—— 面试官到底在问啥？

在面试过程中，一些对技术深度期望比较高的面试官，为了试大家的“底细”（也不排除一部分就是为了装 x：））），可能会在作用域、变量访问相关问题上冷不丁抛出 LHS、 RHS 这样听上去比较“高深”的名词。为了避免大家吃亏，我们这里把这俩哥拉出来遛遛：

LHS、RHS，是引擎在执行代码的时候，查询变量的两种方式。其中的 L、R，分别意味着 Left、Right。这个“左”和“右”，是相对于赋值操作来说的。当变量出现在赋值操作的左侧时，执行的就是 LHS 操作，右侧则执行 RHS 操作：

```
name = 'xiuyan';
```

在这个例子里，name 变量出现在赋值操作的左侧，它就属于 LHS。LHS 意味着 **变量赋值或写入内存**，

它强调的是一个写入的动作，所以 LHS 查询查的是这个变量的“家”（对应的内存空间）在哪。

```
var myName = name
console.log(name)
```

在这个例子里，第一行有赋值操作，但是 name 在操作的右侧，所以是 RHS；第二行没有赋值操作，name 就可以理解为没有出现在赋值操作的左侧，这种情况下我们也认为 name 的查询是 RHS。RHS 意味着 **变量查找或从内存中读取**，它强调的是读这个动作，查询的是变量的内容。

对于 LHS、RHS 这两个概念，大家若能理解上面两个示例，能说出它们各自是怎么回事儿，就非常足够了。接下来，我们就进入词法作用域环节的学习～

}

01 开篇词：前端面试的知识体系与世界观

03 闭包—探索词法作用域模型