

12 | 延迟解析：V8是如何实现闭包的？

2020-04-11 李兵

图解 Google V8

[进入课程 >](#)



讲述：李兵

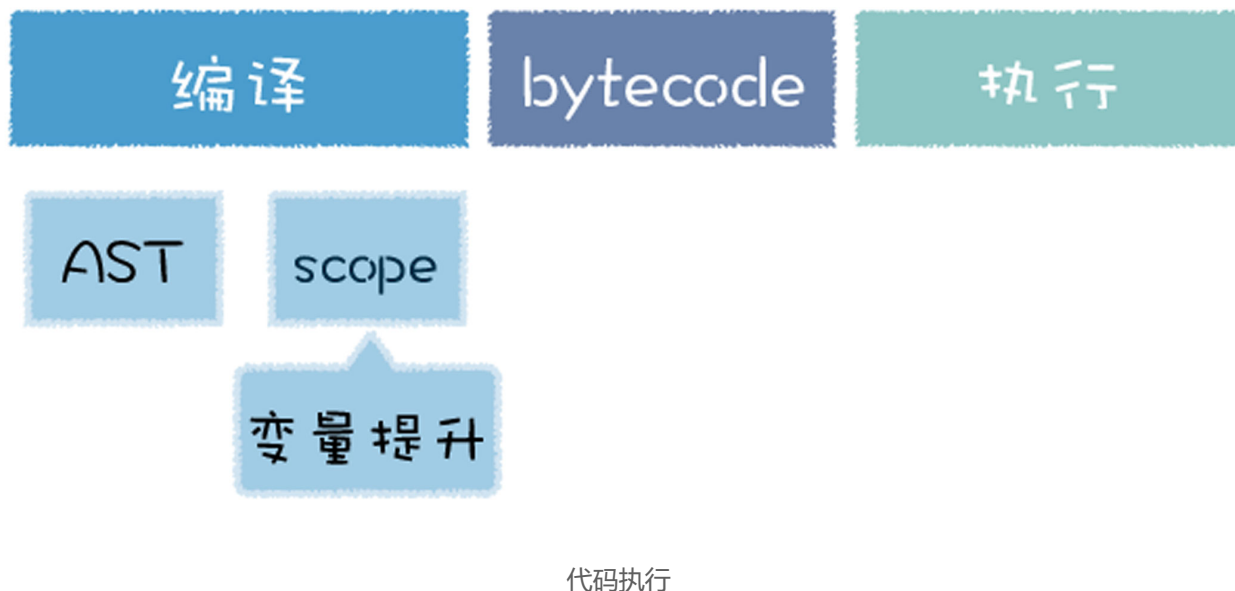
时长 13:51 大小 12.70M



你好，我是李兵。

在第一节我们介绍过 V8 执行 JavaScript 代码，需要经过**编译**和**执行**两个阶段，其中**编译过程**是指 V8 将 JavaScript 代码转换为字节码或者二进制机器代码的阶段，而执行阶段则是指解释器解释执行字节码，或者是 CPU 直接执行二进制机器代码的阶段。总的流程你可以参看下图：





在编译 JavaScript 代码的过程中，V8 并不会一次性将所有的 JavaScript 解析为中间代码，这主要是基于以下两点：

首先，如果一次解析和编译所有的 JavaScript 代码，过多的代码会增加编译时间，这会严重影响到首次执行 JavaScript 代码的速度，让用户感觉到卡顿。因为有时候一个页面的 JavaScript 代码都有 10 多兆，如果要将所有的代码一次性解析编译完成，那么会大大增加用户的等待时间；

其次，解析完成的字节码和编译之后的机器代码都会存放在内存中，如果一次性解析和编译所有 JavaScript 代码，那么这些中间代码和机器代码将会一直占用内存，特别是在手机普及的年代，内存是非常宝贵的资源。

基于以上的原因，所有主流的 JavaScript 虚拟机都实现了**惰性解析**。所谓惰性解析是指解析器在解析的过程中，如果遇到函数声明，那么会跳过函数内部的代码，并不会为其生成 AST 和字节码，而仅仅生成顶层代码的 AST 和字节码。

惰性解析的过程

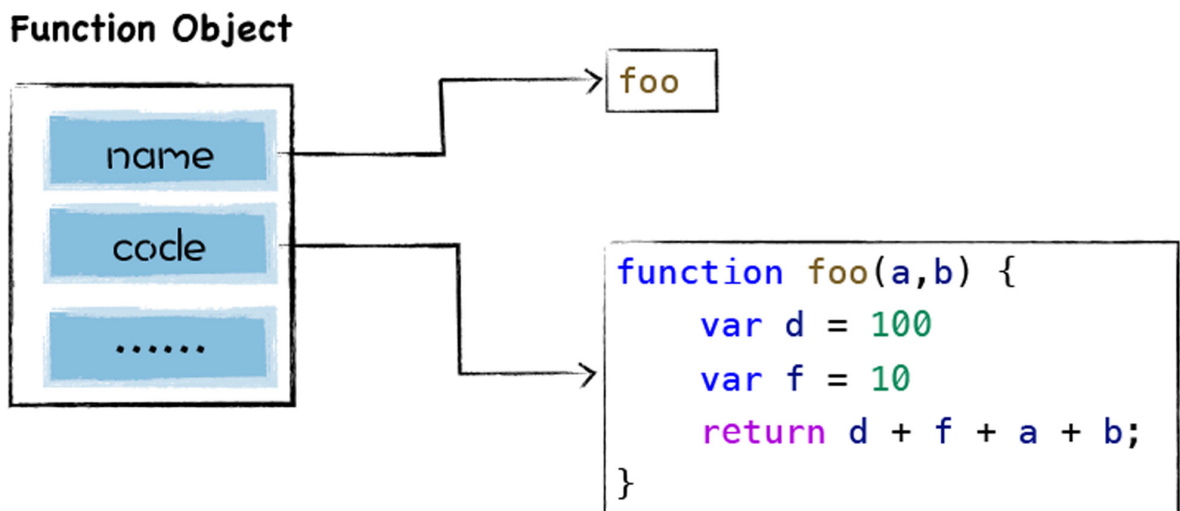
关于惰性解析，我们可以结合下面这个例子来分析下：

```
1 function foo(a,b) {  
2     var d = 100  
3     var f = 10  
4     return d + f + a + b;  
}
```

 复制代码

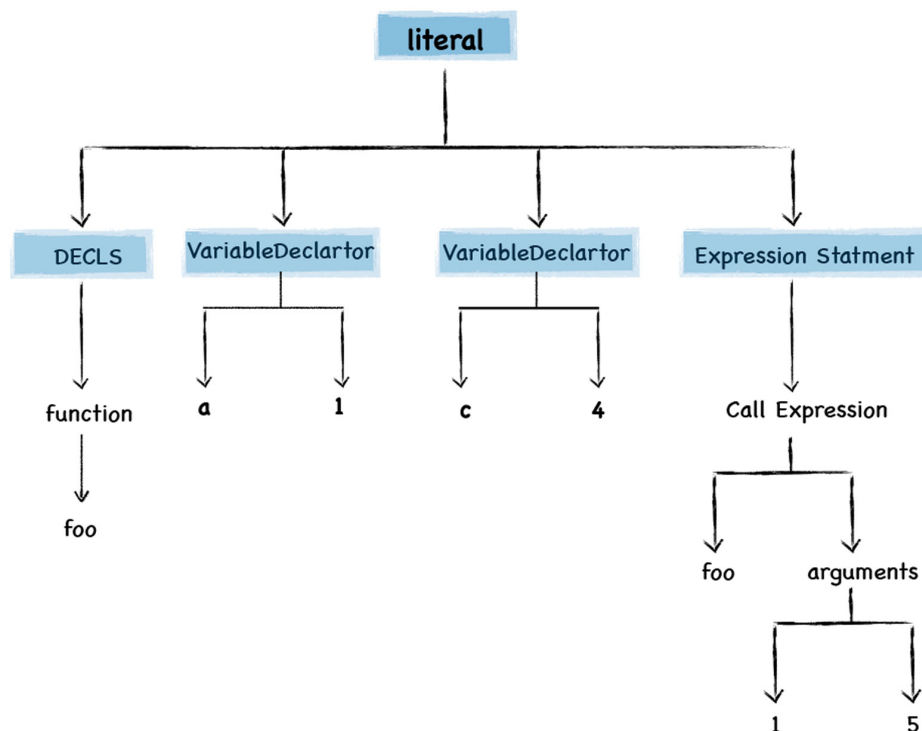
```
5 }  
6 var a = 1  
7 var c = 4  
8 foo(1, 5)
```

当把这段代码交给 V8 处理时，V8 会至上而下解析这段代码，在解析过程中首先会遇到 foo 函数，由于这只是一个函数声明语句，V8 在这个阶段只需要将该函数转换为函数对象，如下图所示：



注意，这里只是将该函数声明转换为函数对象，但是并没有解析和编译函数内部的代码，所以也不会为 foo 函数的内部代码生成抽象语法树。

然后继续往下解析，由于后续的代码都是顶层代码，所以 V8 会为它们生成抽象语法树，最终生成的结果如下所示：



生成顶层代码的抽象语法树

代码解析完成之后，V8 便会按照顺序自上而下执行代码，首先会先执行 “a=1” 和 “c=4” 这两个赋值表达式，接下来执行 foo 函数的调用，过程是从 foo 函数对象中取出函数代码，然后和编译顶层代码一样，V8 会先编译 foo 函数的代码，编译时同样需要先将其编译为抽象语法树和字节码，然后再解释执行。

好了，上面就是惰性解析的一个大致过程，看上去是不是很简单，不过在 V8 实现惰性解析的过程中，需要支持 JavaScript 中的闭包特性，这会使得 V8 的解析过程变得异常复杂。

为什么闭包会让 V8 解析代码的过程变得复杂呢？要解答这个问题，我们先来拆解闭包的特性，然后再来分析为什么闭包影响到了 V8 的解析流程。

拆解闭包——JavaScript 的三个特性

JavaScript 中的闭包有三个基础特性。

第一，JavaScript 语言允许在函数内部定义新的函数，代码如下所示：

```
1 function foo() {
2   function inner() {
3   }
4   inner()
```

 复制代码

```
5 }
```

这和其他的流行语言有点差异，在其他的大部分语言中，函数只能声明在顶层代码中，而 JavaScript 中之所以可以在函数中声明另外一个函数，主要是因为 JavaScript 中的函数即对象，你可以在函数中声明一个变量，当然你也可以在函数中声明一个函数。

第二，可以在**内部函数中访问父函数中定义的变量**，代码如下所示：

 复制代码

```
1 var d = 20
2 //inner函数的父函数，词法作用域
3 function foo() {
4     var d = 55
5     //foo的内部函数
6     function inner() {
7         return d+2
8     }
9     inner()
10 }
```

由于可以在函数中定义新的函数，所以很自然的，内部的函数可以使用外部函数中定义的变量，注意上面代码中的 inner 函数和 foo 函数，inner 是在 foo 函数内部定义的，我们就称 inner 函数是 foo 函数的子函数，foo 函数是 inner 函数的父函数。这里的父子关系是针对词法作用域而言的，因为词法作用域在函数声明时就决定了，比如 inner 函数是在 foo 函数内部声明的，所以 inner 函数可以访问 foo 函数内部的变量，比如 inner 就可以访问 foo 函数中的变量 d。

但是如果在 foo 函数外部，也定义了一个变量 d，那么当 inner 函数访问该变量时，到底是该访问哪个变量呢？

在《[06 | 作用域链：V8 是如何查找变量的？](#)》这节课，我介绍了词法作用域和词法作用域链，每个函数有自己的词法作用域，该函数中定义的变量都存在于该作用域中，然后 V8 会将这些作用域按照词法的位置，也就是代码位置关系，将这些作用域串成一个链，这就是词法作用域链，查找变量的时候会沿着词法作用域链的途径来查找。

所以，inner 函数在自己的作用域中没有查找到变量 d，就接着在 foo 函数的作用域中查找，再查找不到才会查找顶层作用域中的变量。所以 inner 函数中使用的变量 d 就是 foo 函数中的变量 d。

第三，**因为函数是一等公民，所以函数可以作为返回值**，我们可以看下面这段代码：

 复制代码

```
1 function foo() {  
2     return function inner(a, b) {  
3         const c = a + b  
4         return c  
5     }  
6 }  
7 const f = foo()
```

观察上面这段代码，我们将 inner 函数作为了 foo 函数的返回值，也就是说，当调用 foo 函数时，最终会返回 inner 函数给调用者，比如上面我们将 inner 函数返回给了全局变量 f，接下来就可以在外部像调用 inner 函数一样调用 f 了。

以上就是和 JavaScript 闭包相关的三个重要特性：

可以在 JavaScript 函数内部定义新的函数；

内部函数中访问父函数中定义的变量；

因为 JavaScript 中的函数是一等公民，所以函数可以作为另外一个函数的返回值。

这也是 JavaScript 过于灵活的一个原因，比如在 C/C++ 中，你就不可以在一个函数中定义另外一个函数，所以也就没了内部函数访问外部函数中变量的问题了。

闭包给惰性解析带来的问题

好了，了解了 JavaScript 的这三个特性之后，下面我们就来使用这三个特性组装的一段经典的闭包代码：

 复制代码

```
1 function foo() {  
2     var d = 20  
3     return function inner(a, b) {
```

```
4         const c = a + b + d
5         return c
6     }
7 }
8 const f = foo()
```

观察上面这段代码，我们在 `foo` 函数中定义了 `inner` 函数，并返回 `inner` 函数，同时在 `inner` 函数中访问了 `foo` 函数中的变量 `d`。

我们可以分析下上面这段代码的执行过程：

当调用 `foo` 函数时，`foo` 函数会将它的内部函数 `inner` 返回给全局变量 `f`；

然后 `foo` 函数执行结束，执行上下文被 V8 销毁了；

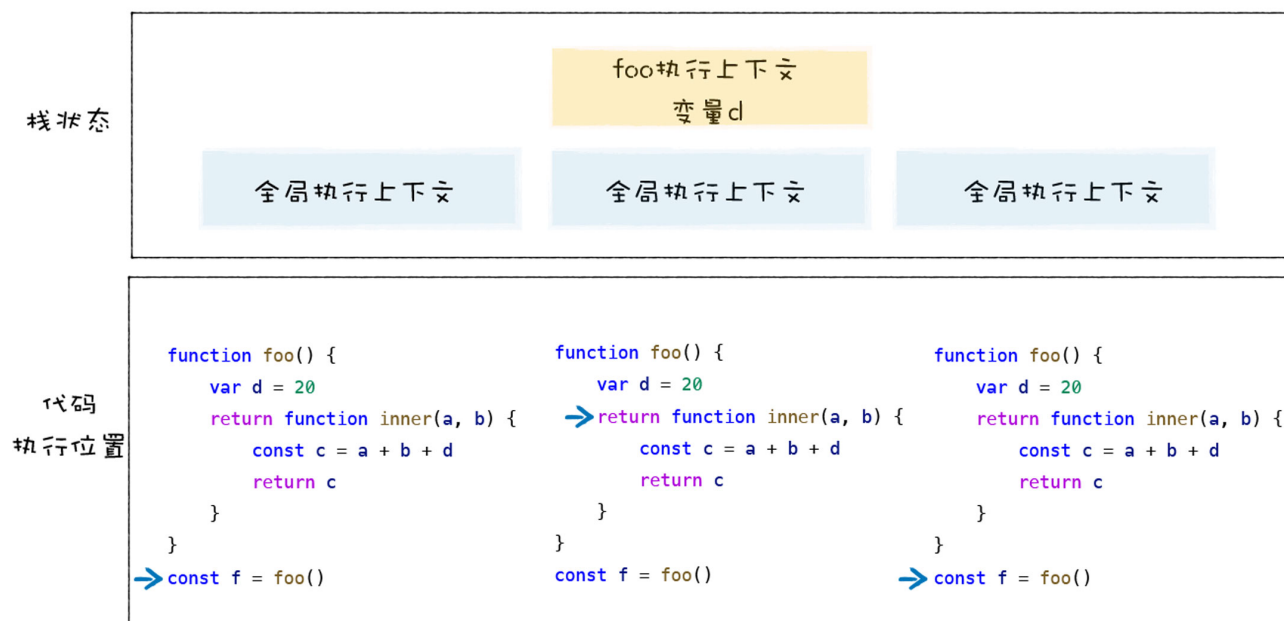
虽然 `foo` 函数的执行上下文被销毁了，但是依然存活的 `inner` 函数引用了 `foo` 函数作用域中的变量 `d`。

按照通用的做法，`d` 已经被 v8 销毁了，但是由于存活的函数 `inner` 依然引用了 `foo` 函数中的变量 `d`，这样就会带来两个问题：

当 `foo` 执行结束时，变量 `d` 该不该被销毁？如果不应该被销毁，那么应该采用什么策略？

如果采用了惰性解析，那么当执行到 `foo` 函数时，V8 只会解析 `foo` 函数，并不会解析内部的 `inner` 函数，那么这时候 V8 就不知道 `inner` 函数中是否引用了 `foo` 函数的变量 `d`。

这么讲可能有点抽象，下面我们就来看一下上面这段代码的执行流程，我们上节分析过了，JavaScript 是一门基于堆和栈的语言，当执行 `foo` 函数的时候，堆栈的变化如下图所示：



从上图可以看出，在执行全局代码时，V8 会将全局执行上下文压入到调用栈中，然后进入执行 `foo` 函数的调用过程，这时候 V8 会为 `foo` 函数创建执行上下文，执行上下文中包括了变量 `d`，然后将 `foo` 函数的执行上下文压入栈中，`foo` 函数执行结束之后，`foo` 函数执行上下文从栈中弹出，这时候 `foo` 执行上下文中的变量 `d` 也随之被销毁。

但是这时候，由于 `inner` 函数被保存到全局变量中了，所以 `inner` 函数依然存在，最关键的地方在于 `inner` 函数使用了 `foo` 函数中的变量 `d`，按照正常执行流程，变量 `d` 在 `foo` 函数执行结束之后就被销毁了。

所以正常的处理方式应该是 `foo` 函数的执行上下文虽然被销毁了，但是 `inner` 函数引用的 `foo` 函数中的变量却不能被销毁，那么 V8 就需要为这种情况做特殊处理，需要保证即便 `foo` 函数执行结束，但是 `foo` 函数中的 `d` 变量依然保持在内存中，不能随着 `foo` 函数的执行上下文被销毁掉。

那么怎么处理呢？


在执行 `foo` 函数的阶段，虽然采取了惰性解析，不会解析和执行 `foo` 函数中的 `inner` 函数，但是 V8 还是需要判断 `inner` 函数是否引用了 `foo` 函数中的变量，负责处理这个任务的模块叫着预解析器。

预解析器如何解决闭包所带来的问题？

V8 引入预解析器，比如当解析顶层代码的时候，遇到了一个函数，那么预解析器并不会直接跳过该函数，而是对该函数做一次快速的预解析，其主要目的有两个。


第一，是判断当前函数是不是存在一些语法上的错误，如下面这段代码：

```
1 function foo(a, b) {  
2   {/} //语法错误  
3 }  
4 var a = 1  
5 var c = 4  
6 foo(1, 5)
```

 复制代码

在预解析过程中，预解析器发现了语法错误，那么就会向 V8 抛出语法错误，比如上面这段代码的语法错误是这样的：

```
1 Uncaught SyntaxError: Invalid regular expression: missing /
```

 复制代码

第二，除了检查语法错误之外，预解析器另外的一个重要的功能就是检查函数内部是否引用了外部变量，如果引用了外部的变量，预解析器会将栈中的变量复制到堆中，在下次执行到该函数的时候，直接使用堆中的引用，这样就解决了闭包所带来的问题。

总结

今天我们主要介绍了 V8 的惰性解析，所谓惰性解析是指解析器在解析的过程中，如果遇到函数声明，那么会跳过函数内部的代码，并不会为其生成 AST 和字节码，而仅仅生成顶层代码的 AST 和字节码。

利用惰性解析可以加速 JavaScript 代码的启动速度，如果要将所有的代码一次性解析编译完成，那么会大大增加用户的等待时间。

由于 JavaScript 是一门天生支持闭包的语言，由于闭包会引用当前函数作用域之外的变量，所以当 V8 解析一个函数的时候，还需要判断该函数的内部函数是否引用了当前函数内

部声明的变量，如果引用了，那么需要将该变量存放到堆中，即便当前函数执行结束之后，也不会释放该变量。

思考题

观察下面两段代码：

 复制代码

```
1 function foo() {  
2     var a = 0  
3 }
```

 复制代码

```
1 function foo() {  
2     var a = 0  
3     return function inner() {  
4         return a++  
5     }  
6 }
```

请你思考下，当调用 foo 函数时，foo 函数内部的变量 a 会分别分配到栈上？还是堆上？为什么？欢迎你在留言区与我分享讨论。

感谢你的阅读，如果你觉得这篇文章对你有所启发，也欢迎把它分享给你的朋友。

图解 Google V8

一门课搞懂 JavaScript 执行逻辑

李兵

前盛大创新院高级研究员



新版升级：点击「 请朋友读」，20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 11 | 堆和栈：函数调用是如何影响到内存布局的？

精选留言 (9)

 写留言



一步

2020-04-13

第一个函数：变量a在栈上，当解析foo函数的时候，预解析没有发现内部函数，所有变量a就只会在栈上，当foo函数的执行上下文销毁的时候a变量也会销毁；

第二个函数，变量a同时在栈和堆上，当解析foo函数的时候，预解析有发现内部函数引用外部变量a，这时候就会把a复制到堆上，当父函数执行到a的赋值语句时，会同时...
展开



1



潇潇雨歇

2020-04-13

第一个栈，第二个堆

展开





I WANN BE THAT G...

2020-04-13

第一个在栈上，第二个应该栈上，但是会拷贝一份到堆里?根据你文中的描述，我也不是很确定，老师能给下答案吗? 😊

展开 ∨



bright

2020-04-12

第一个a在栈中，第二个形成了闭包，先在栈中存在a，然后在预解析器阶段在堆中复制了一个一样的a，调用foo函数使foo出栈栈中的a被销毁，只剩下堆中的a。

展开 ∨



刘大夫

2020-04-12

按照文中所讲，惰性解析（编译）和预解析是发生在编译阶段的，与代码是否执行无关，预解析器只要找到一个被内部函数引用的变量，就会把它保存到堆中，因此第一个函数中声明的变量 a 仅仅保存到调用栈，而第二个函数中声明的 a 则会保存到堆中

展开 ∨



William

2020-04-12

惰性解析 + 预解析器 组合

展开 ∨



saber

2020-04-11

应该都是分配在栈上，然后销毁foo的执行上下文的时候会有一个预解析的过程，检测到如果内部函数引用到了该作用域内变量，再将该变量放入到堆中存储。

展开 ∨



tt

2020-04-11

按照本课的讲解，第一个foo函数被预解析器扫描后，发现没有内部函数引用变量a，所以a会被分配到栈上，而与之相反，第二个foo中的a会被分配到堆上。

我忍不住脑补C++的实现，JS中的变量都是C++中的智能指针吧，执行上下文、变量环境都是一个C++中的对象，JS中的堆也是C++中的一个对象容器。...

展开 ▾



mfist

2020-04-11

函数1 foo变量挂在栈上面，编译阶段的预解析扫描foo函数内部没有发现闭包和语法错误，将其转换成一个函数对象放在堆中，执行过程中编译执行`` var a=0``

函数2 foo中内部变量a挂在预解析创建的对象上存在堆上。执行foo函数过程中先预解析inner函数，发现引用了外部变量，创建Closure变量放在inner函数本地变量中，执行inner可以直接使用。...

展开 ▾

