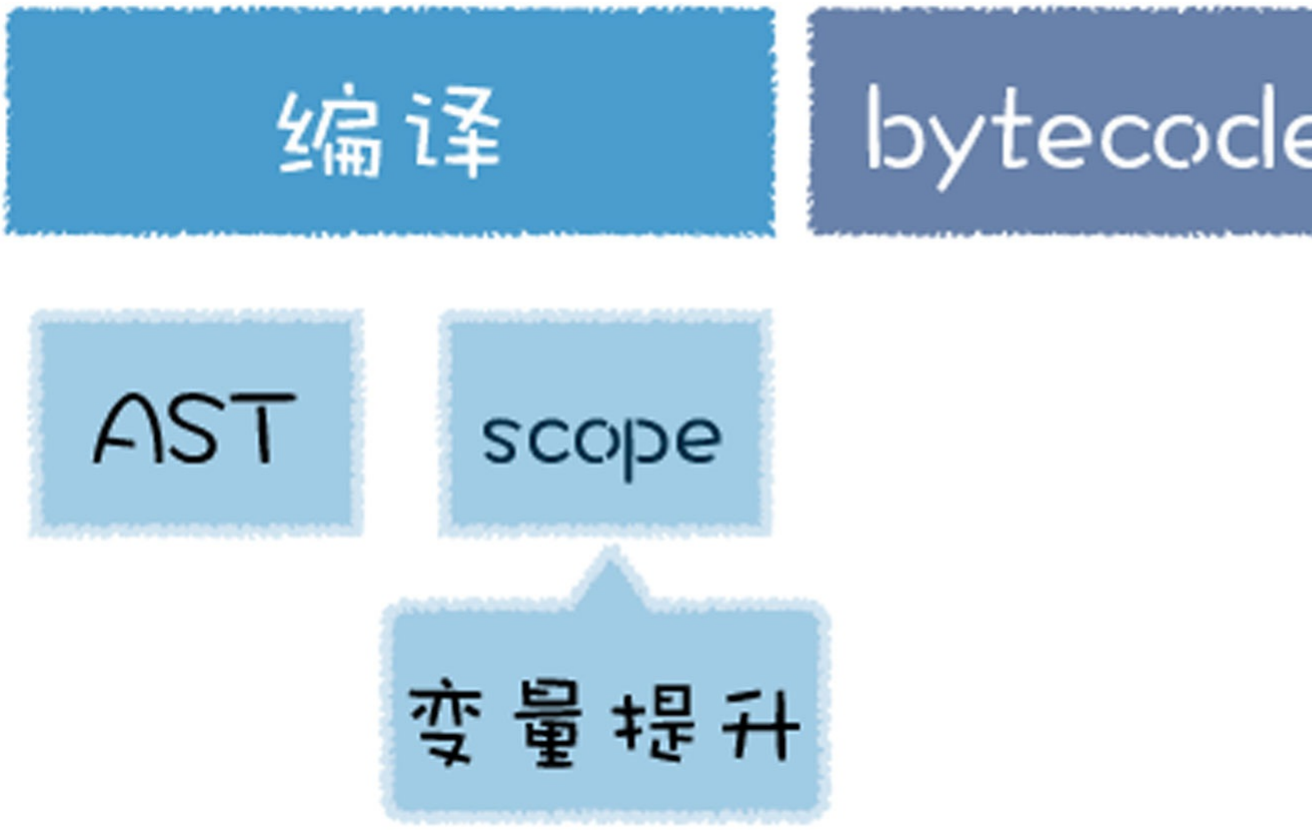
你好，我是李兵。

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。总的流程你可以参考下图：



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。

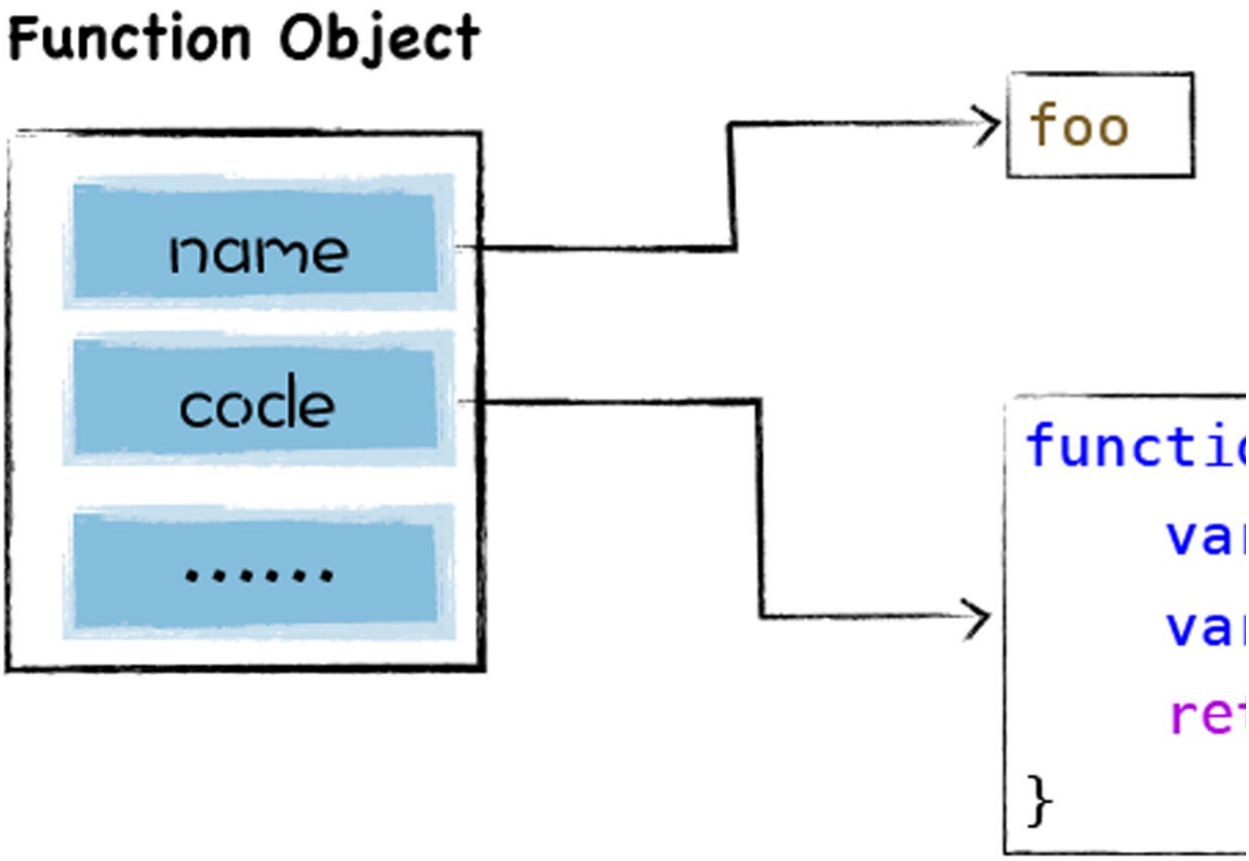
基于以上的原因，所有主流的JavaScript虚拟机都实现了**惰性解析**。所谓惰性解析是指解析器在解析的过程中，如果遇到函数声明，那么会跳过函数内部的代码，并不会为其生成AST和字节码，而仅仅生成顶层代码的AST和字节码。

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：

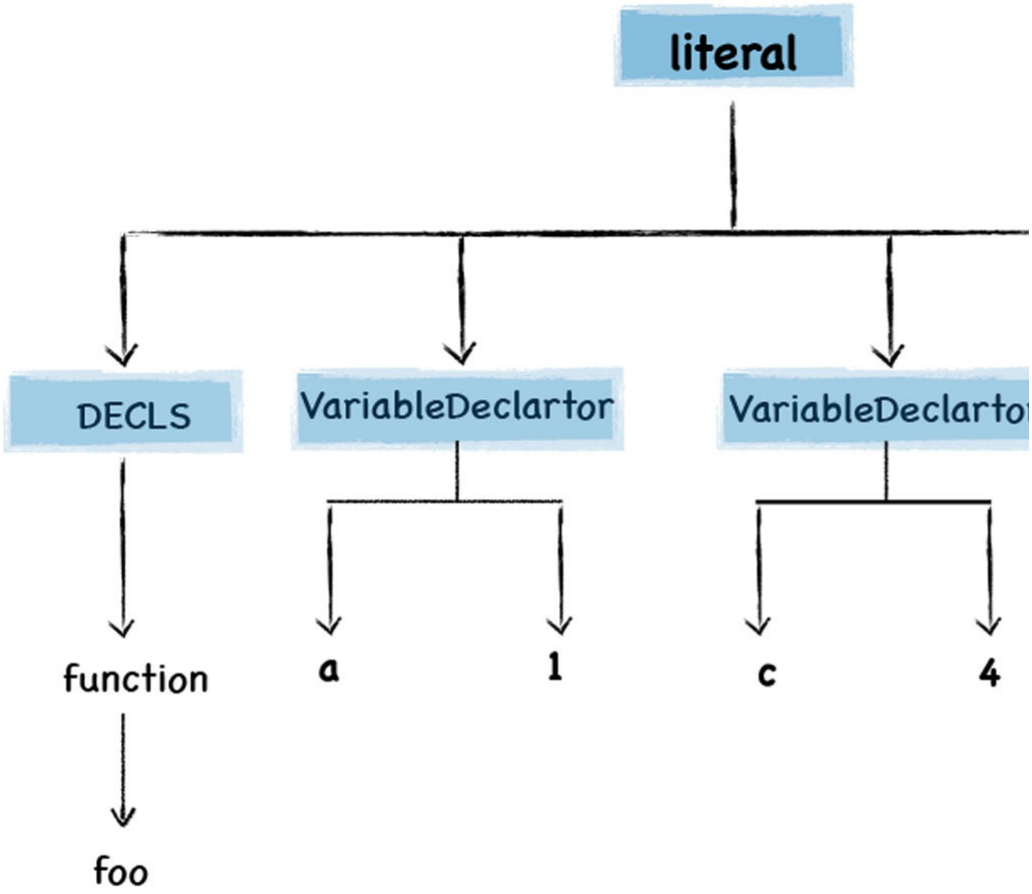
```
function foo(a,b) {
    var d = 100
    var f = 10
    return d + f + a + b;
}
var a = 1
var c = 4
foo(1, 5)
```

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：



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：

```
function foo() {
    function inner() {
    }
    inner()
}
```

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：

```
var d = 20
//inner函数的父函数，词法作用域
function foo() {
    var d = 55
    //foo的内部函数
    function inner() {
      return d+2
    }
    inner()
}
```

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丨作用域链：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：

```
function foo() {
    return function inner(a, b) {
        const c = a + b
        return c
    }
}
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：

```
function foo() {
    var d = 20
    return function inner(a, b) {
        const c = a + b + d
        return c
    }
}
const f = foo()
```

观察上面上面这段代码，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。

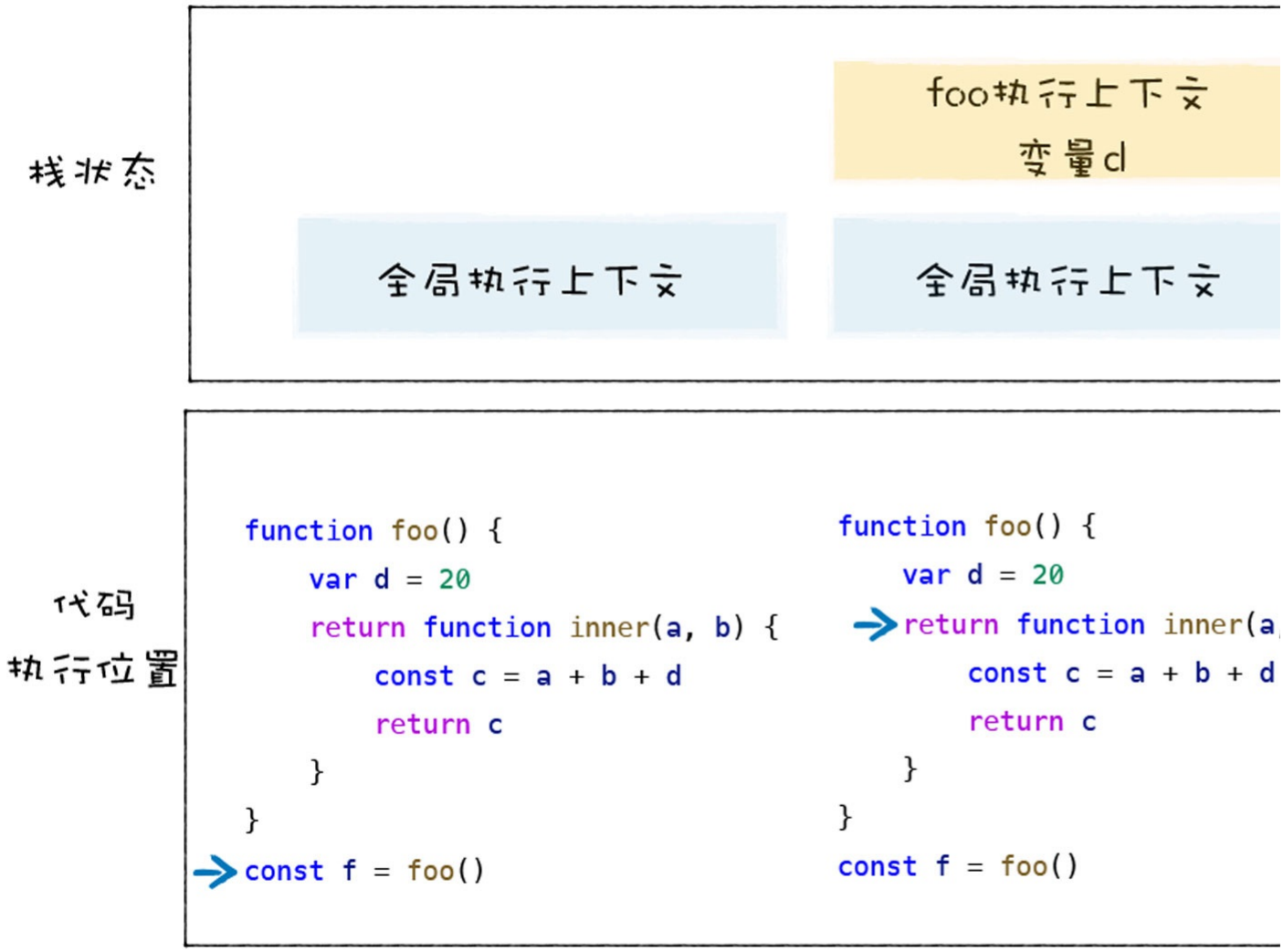
我们可以分析下上面这段代码的执行过程：

- 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；
- 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：



从上图可以看出来，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做预解析器。

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：

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

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：

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

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：

```
function foo() {
   var a = 0
}
```

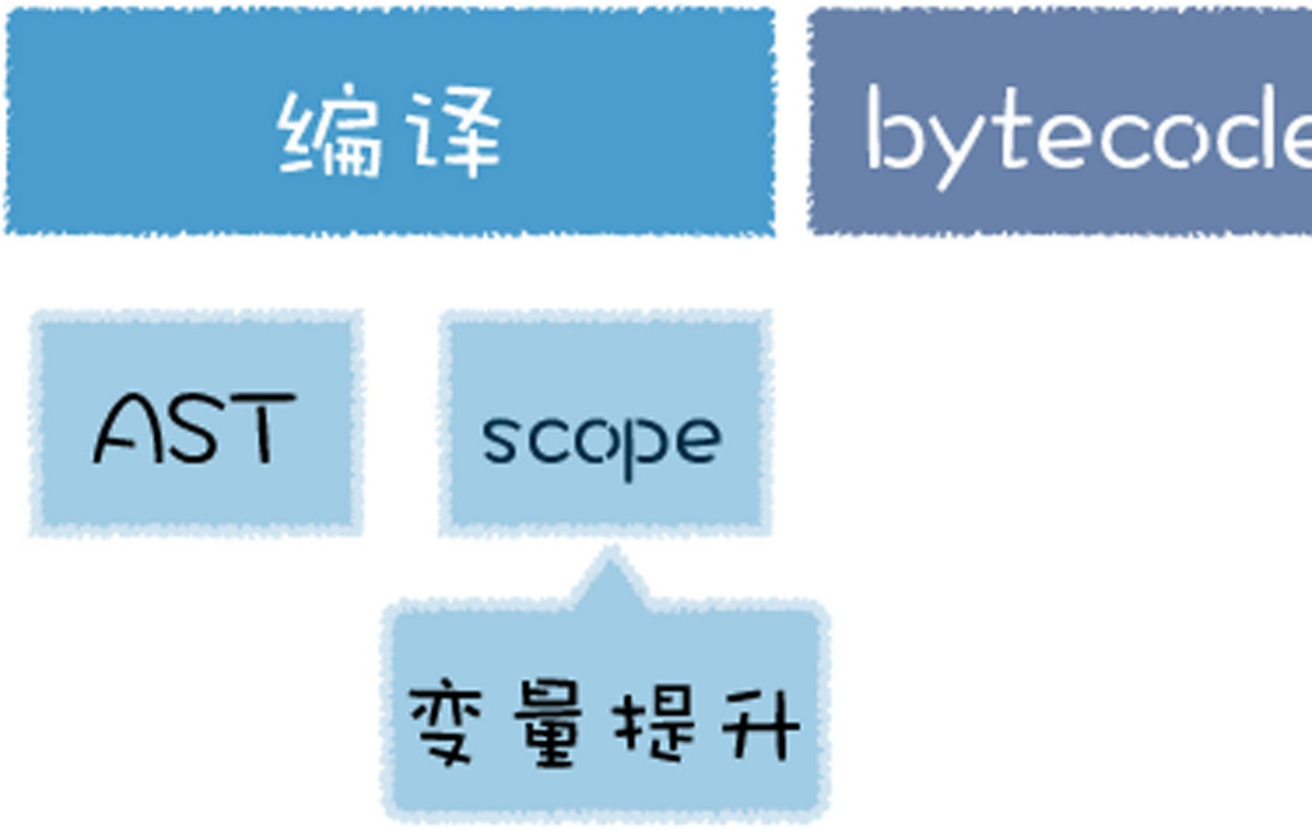
```
function foo() {
    var a = 0
    return function inner() {
        return a++
    }
}
```

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，如果你觉得这一讲的内容对你有所启发，也欢迎把它分享给你的朋友。

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。总的流程你可以参考下图：



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。

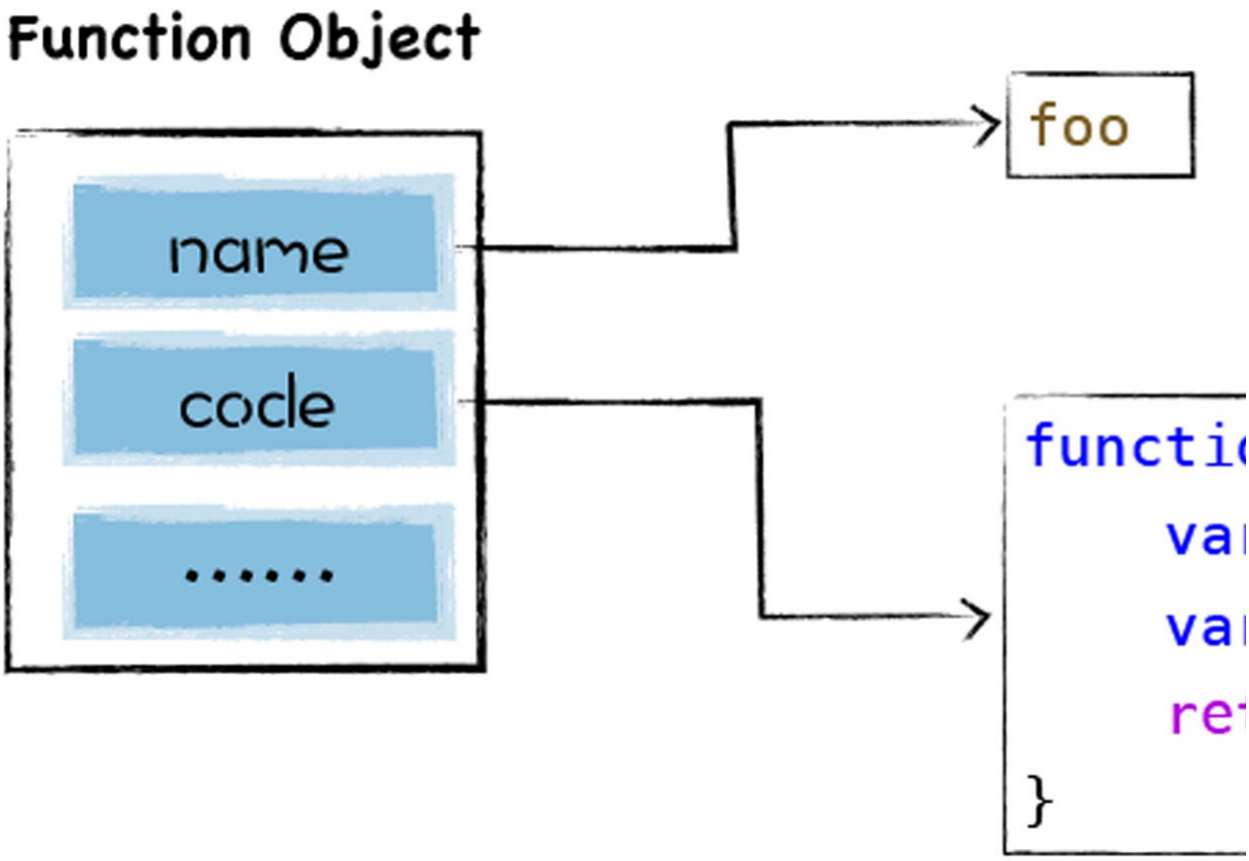
基于以上的原因，所有主流的JavaScript虚拟机都实现了**惰性解析**。所谓惰性解析是指解析器在解析的过程中，如果遇到函数声明，那么会跳过函数内部的代码，并不会为其生成AST和字节码，而仅仅生成顶层代码的AST和字节码。

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：

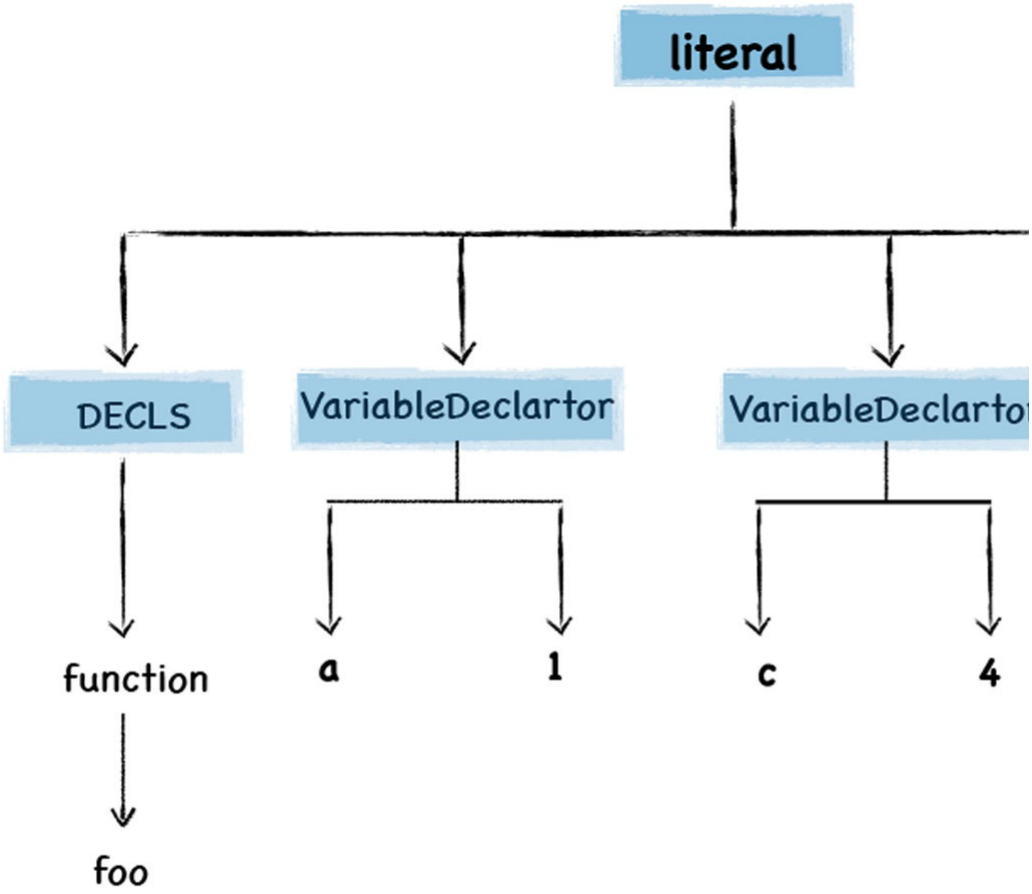
```
function foo(a,b) {
    var d = 100
    var f = 10
    return d + f + a + b;
}
var a = 1
var c = 4
foo(1, 5)
```

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：



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：

```
function foo() {
    function inner() {
    }
    inner()
}
```

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：

```
var d = 20
//inner函数的父函数，词法作用域
function foo() {
    var d = 55
    //foo的内部函数
    function inner() {
      return d+2
    }
    inner()
}
```

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丨作用域链：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：

```
function foo() {
    return function inner(a, b) {
        const c = a + b
        return c
    }
}
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：

```
function foo() {
    var d = 20
    return function inner(a, b) {
        const c = a + b + d
        return c
    }
}
const f = foo()
```

观察上面上面这段代码，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。

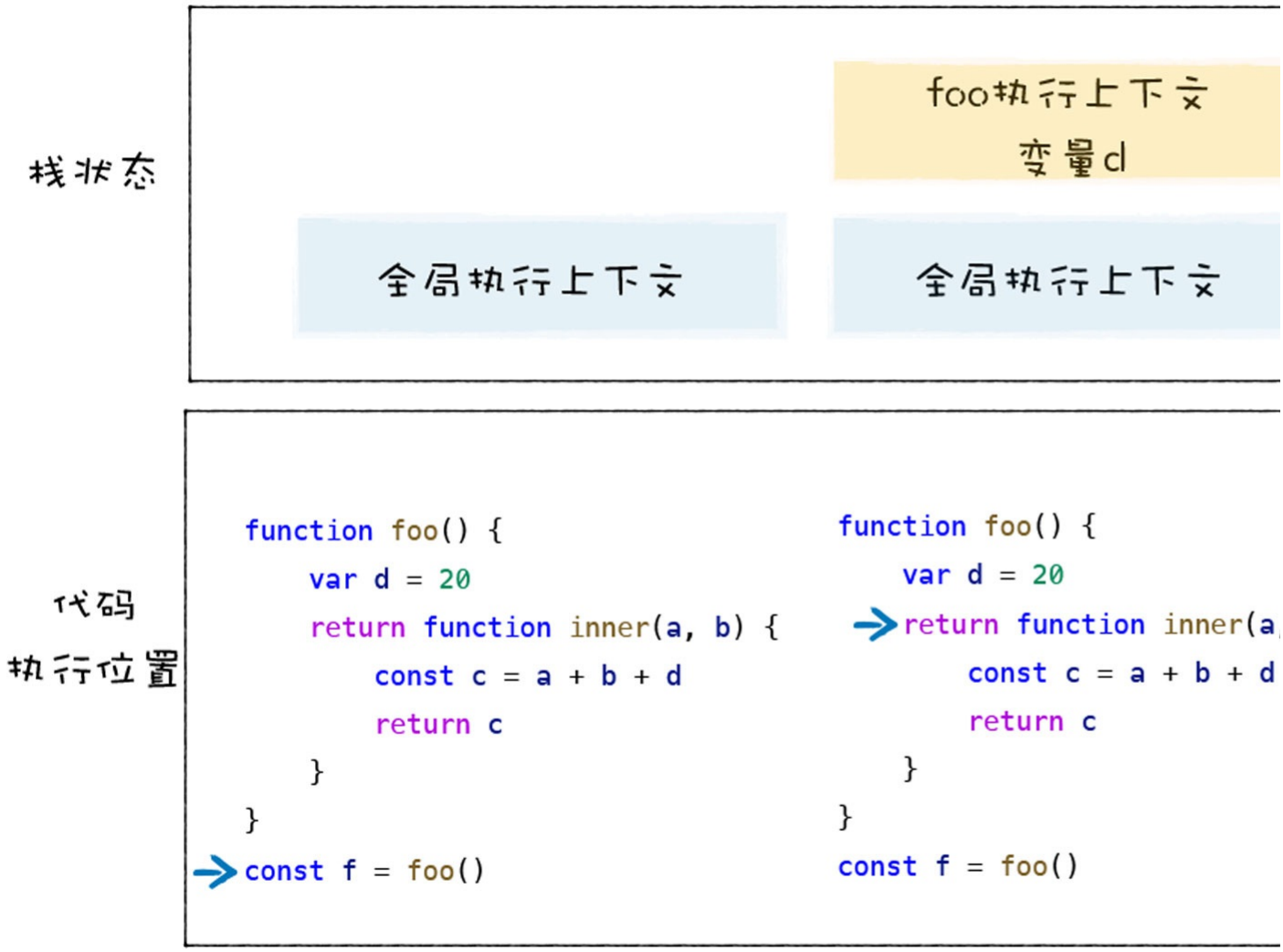
我们可以分析下上面这段代码的执行过程：

- 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；
- 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：



从上图可以看出来，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做预解析器。

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：

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

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：

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

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：

```
function foo() {
   var a = 0
}
```

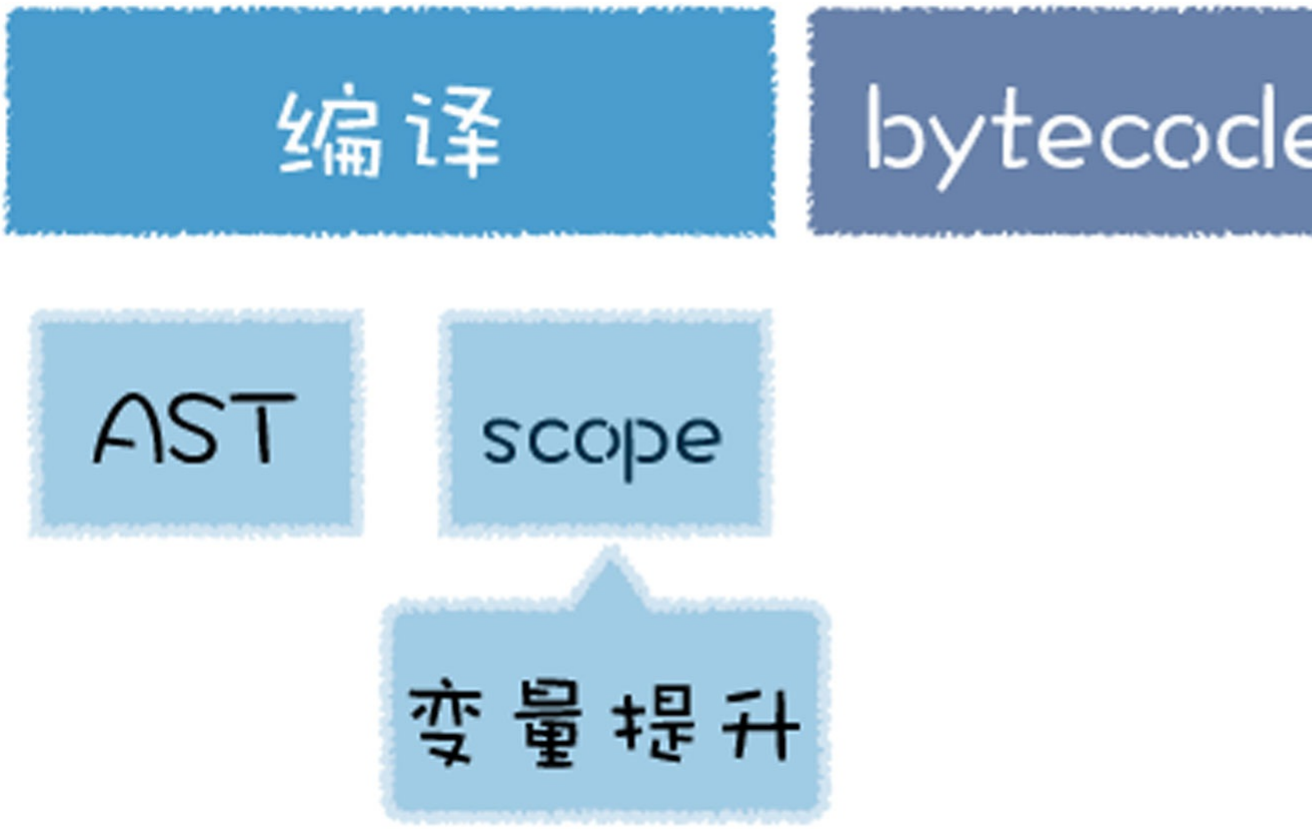
```
function foo() {
    var a = 0
    return function inner() {
        return a++
    }
}
```

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，如果你觉得这一讲的内容对你有所启发，也欢迎把它分享给你的朋友。

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。总的流程你可以参考下图：



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。

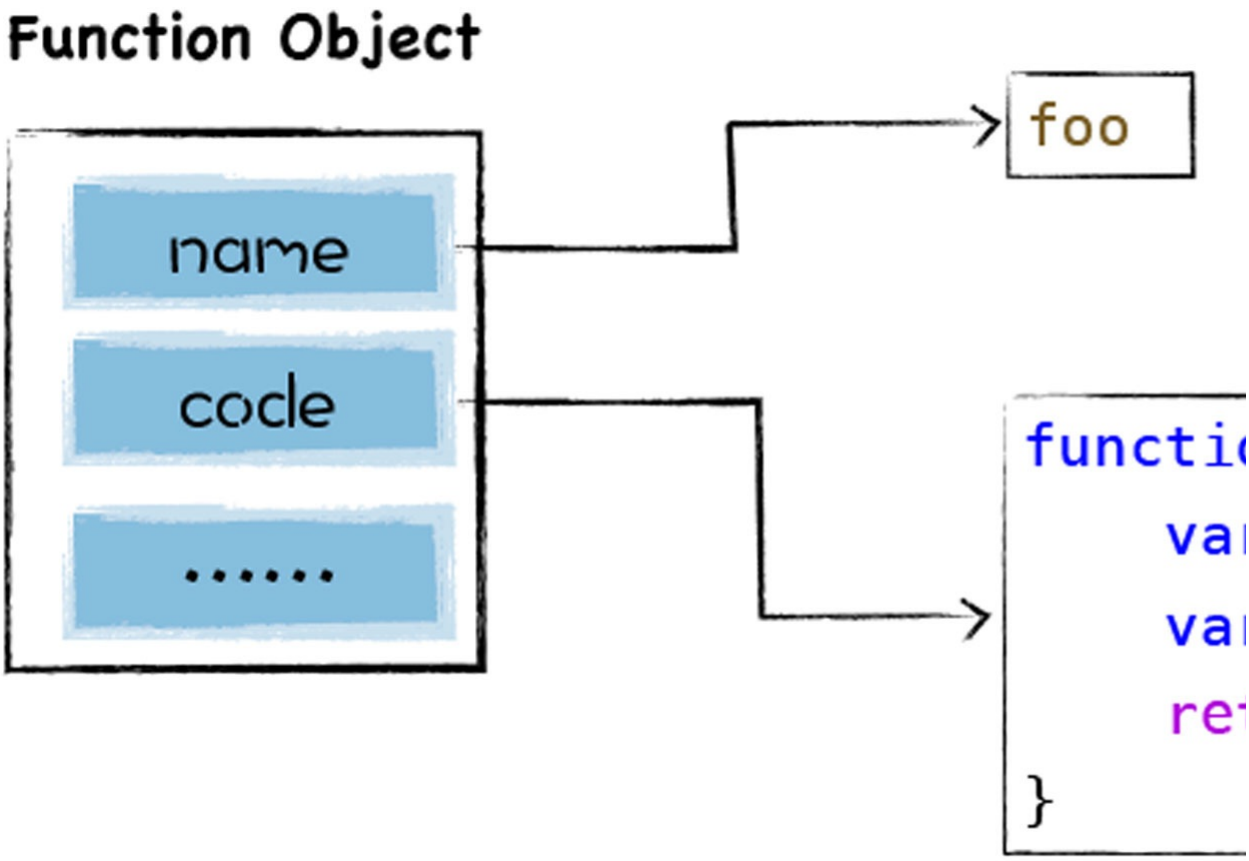
基于以上的原因，所有主流的JavaScript虚拟机都实现了**惰性解析**。所谓惰性解析是指解析器在解析的过程中，如果遇到函数声明，那么会跳过函数内部的代码，并不会为其生成AST和字节码，而仅仅生成顶层代码的AST和字节码。

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：

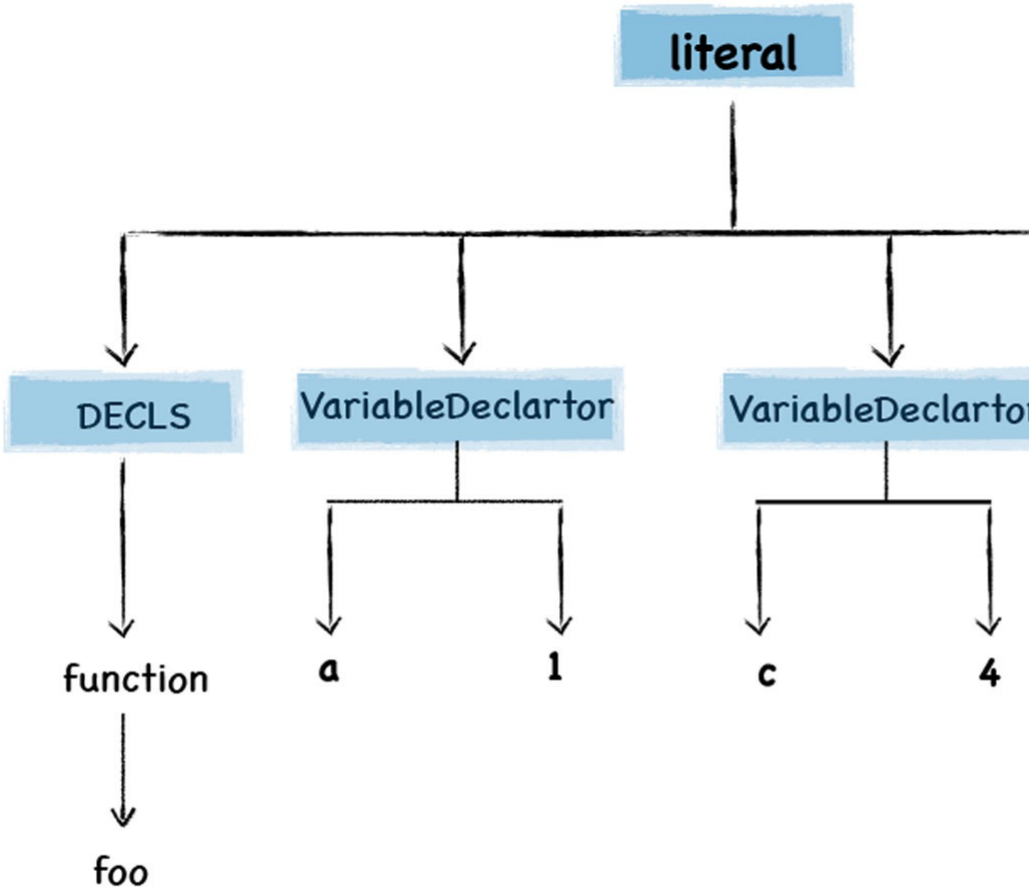
```
function foo(a,b) {
    var d = 100
    var f = 10
    return d + f + a + b;
}
var a = 1
var c = 4
foo(1, 5)
```

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：



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：

```
function foo() {
    function inner() {
    }
    inner()
}
```

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：

```
var d = 20
//inner函数的父函数，词法作用域
function foo() {
    var d = 55
    //foo的内部函数
    function inner() {
      return d+2
    }
    inner()
}
```

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丨作用域链：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：

```
function foo() {
    return function inner(a, b) {
        const c = a + b
        return c
    }
}
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：

```
function foo() {
    var d = 20
    return function inner(a, b) {
        const c = a + b + d
        return c
    }
}
const f = foo()
```

观察上面上面这段代码，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。

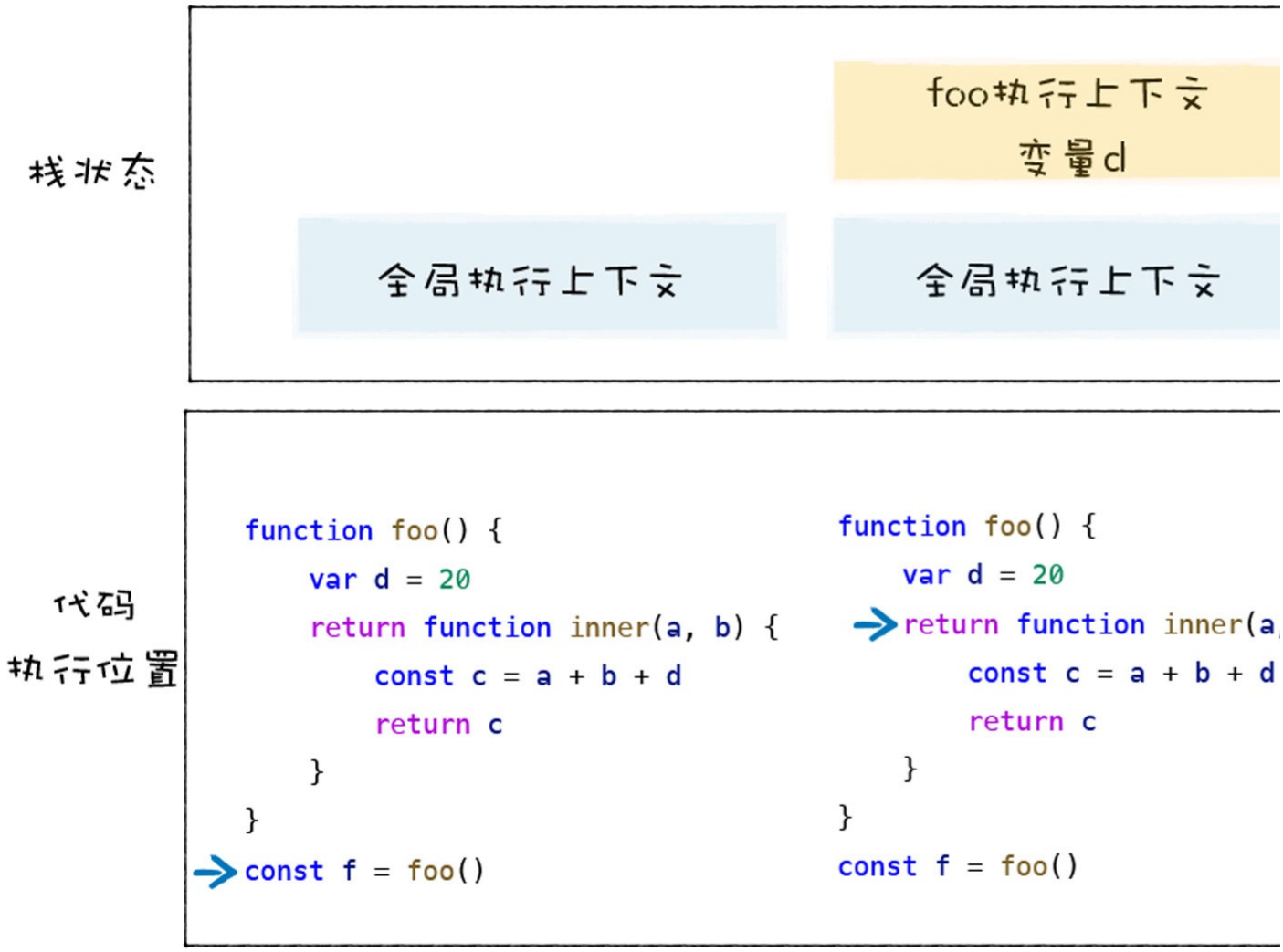
我们可以分析下上面这段代码的执行过程：

- 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；
- 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：



从上图可以看出来，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做预解析器。

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：

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

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：

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

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：

```
function foo() {
   var a = 0
}
```

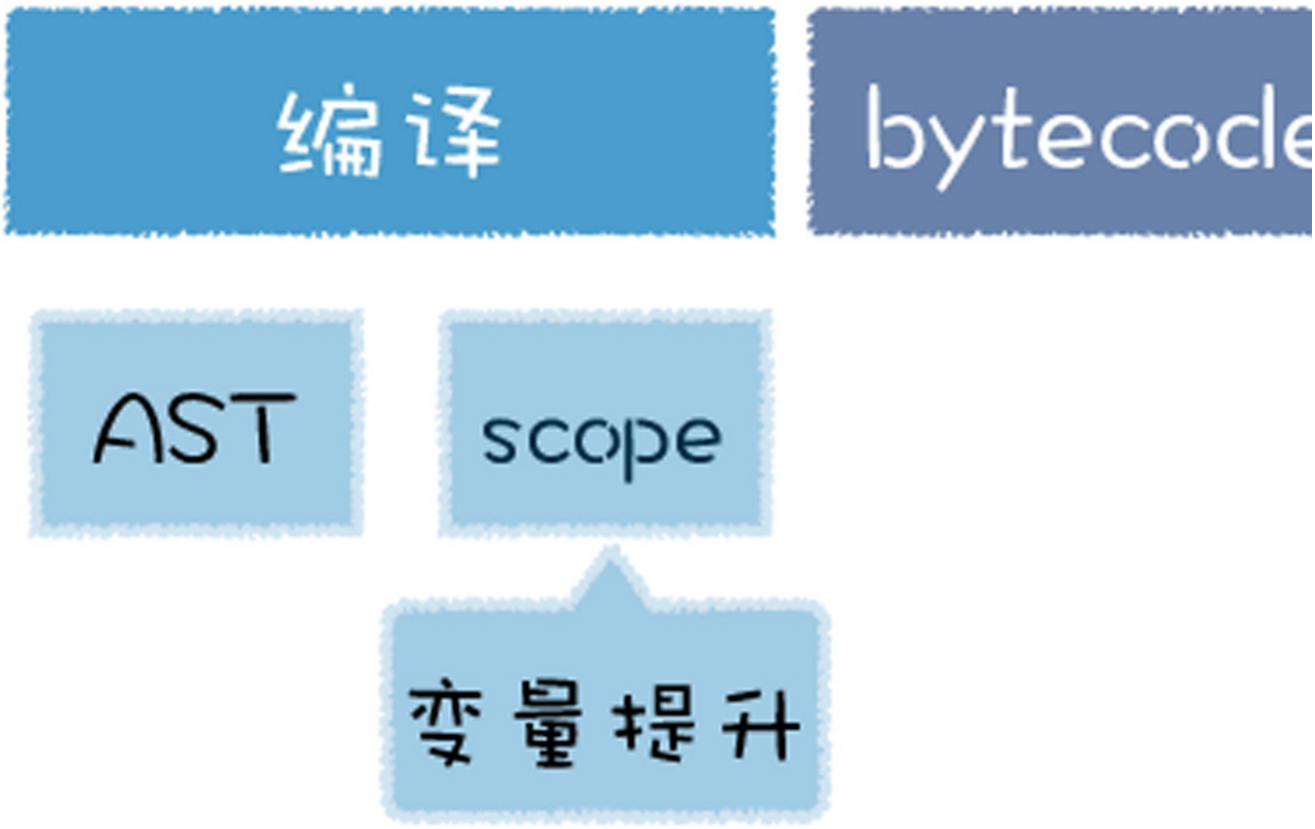
```
function foo() {
    var a = 0
    return function inner() {
        return a++
    }
}
```

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，如果你觉得这一讲的内容对你有所启发，也欢迎把它分享给你的朋友。

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。总的流程你可以参考下图：



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。

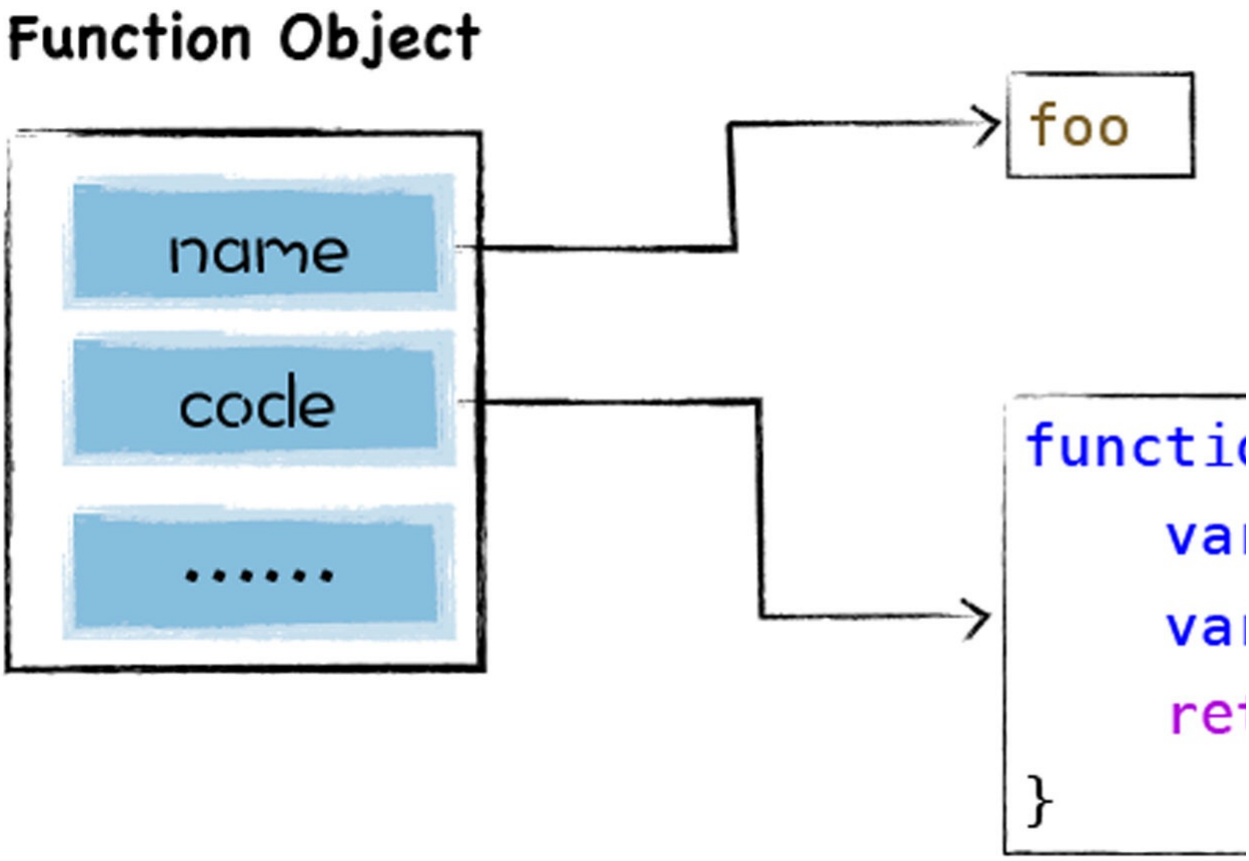
基于以上的原因，所有主流的JavaScript虚拟机都实现了**惰性解析**。所谓惰性解析是指解析器在解析的过程中，如果遇到函数声明，那么会跳过函数内部的代码，并不会为其生成AST和字节码，而仅仅生成顶层代码的AST和字节码。

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：

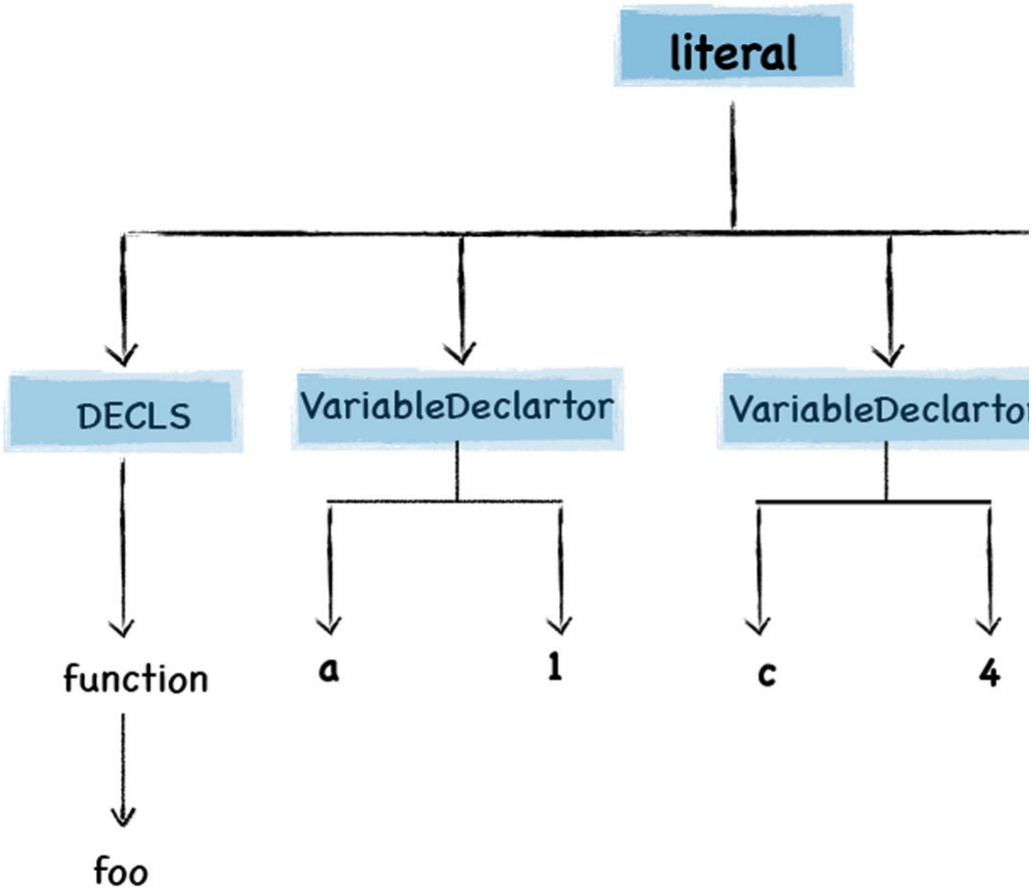
```
function foo(a,b) {
    var d = 100
    var f = 10
    return d + f + a + b;
}
var a = 1
var c = 4
foo(1, 5)
```

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：



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：

```
function foo() {
    function inner() {
    }
    inner()
}
```

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：

```
var d = 20
//inner函数的父函数，词法作用域
function foo() {
    var d = 55
    //foo的内部函数
    function inner() {
      return d+2
    }
    inner()
}
```

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丨作用域链：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：

```
function foo() {
    return function inner(a, b) {
        const c = a + b
        return c
    }
}
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：

```
function foo() {
    var d = 20
    return function inner(a, b) {
        const c = a + b + d
        return c
    }
}
const f = foo()
```

观察上面上面这段代码，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。

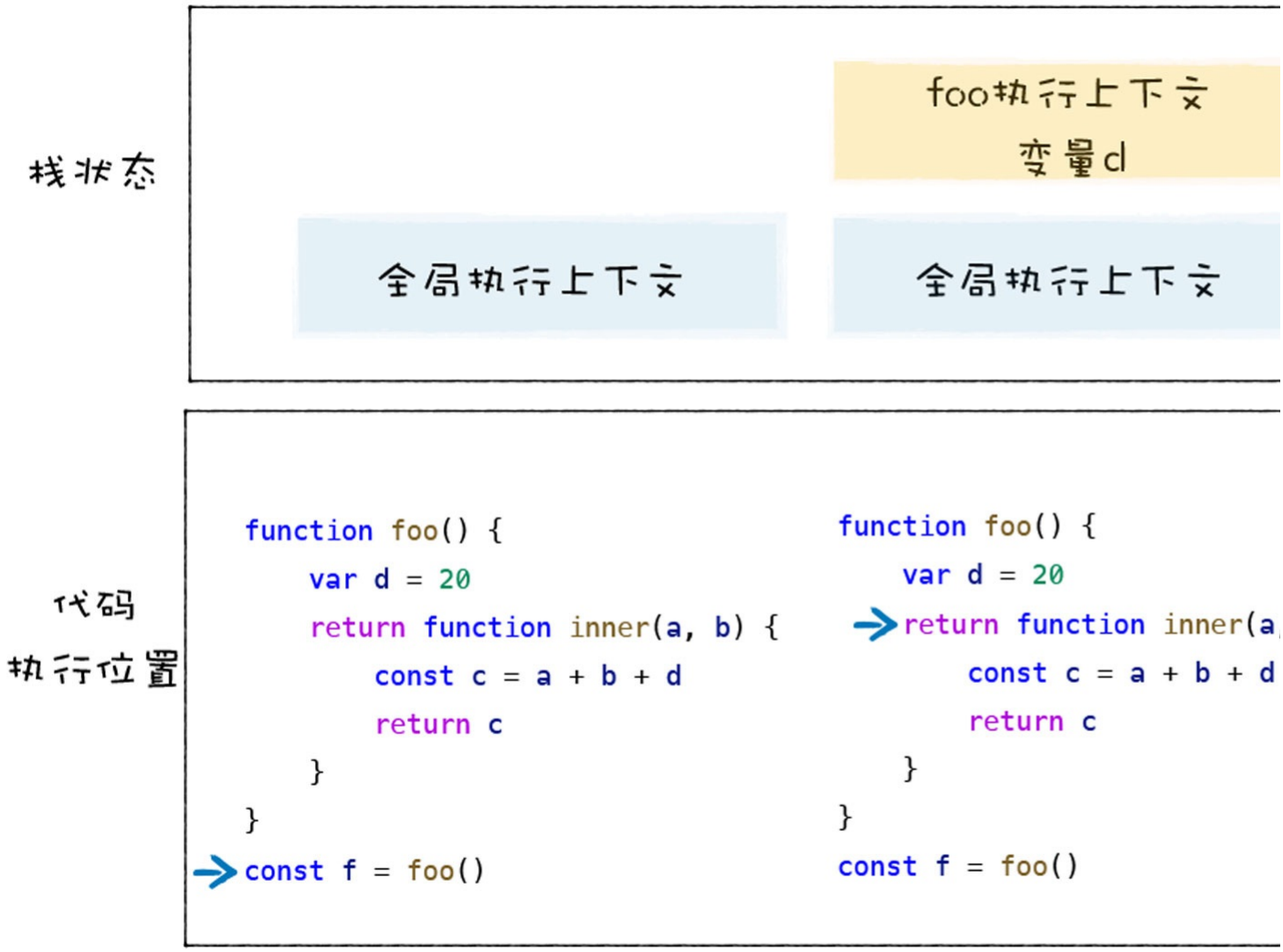
我们可以分析下上面这段代码的执行过程：

- 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；
- 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：



从上图可以看出来，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做预解析器。

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：

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

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：

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

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：

```
function foo() {
   var a = 0
}
```

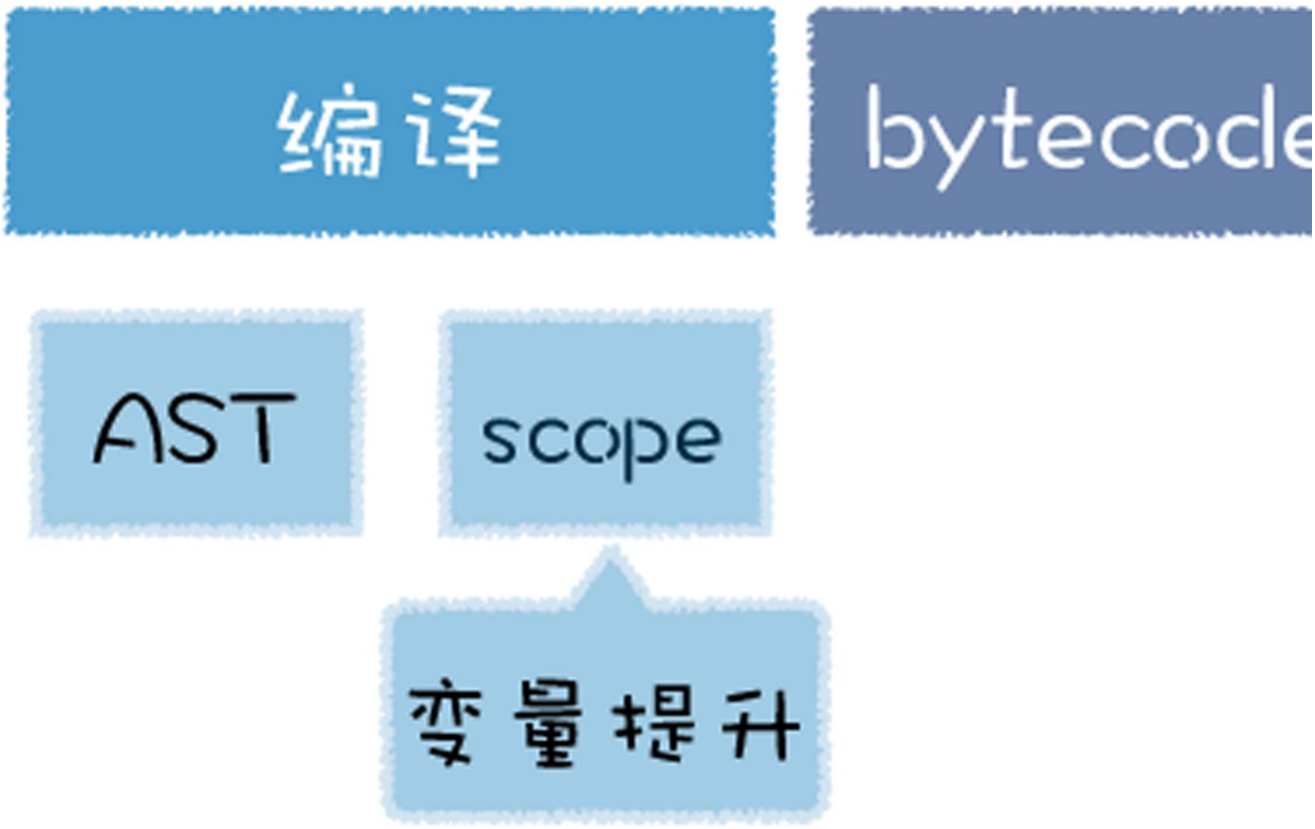
```
function foo() {
    var a = 0
    return function inner() {
        return a++
    }
}
```

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，如果你觉得这一讲的内容对你有所启发，也欢迎把它分享给你的朋友。

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。总的流程你可以参考下图：



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。

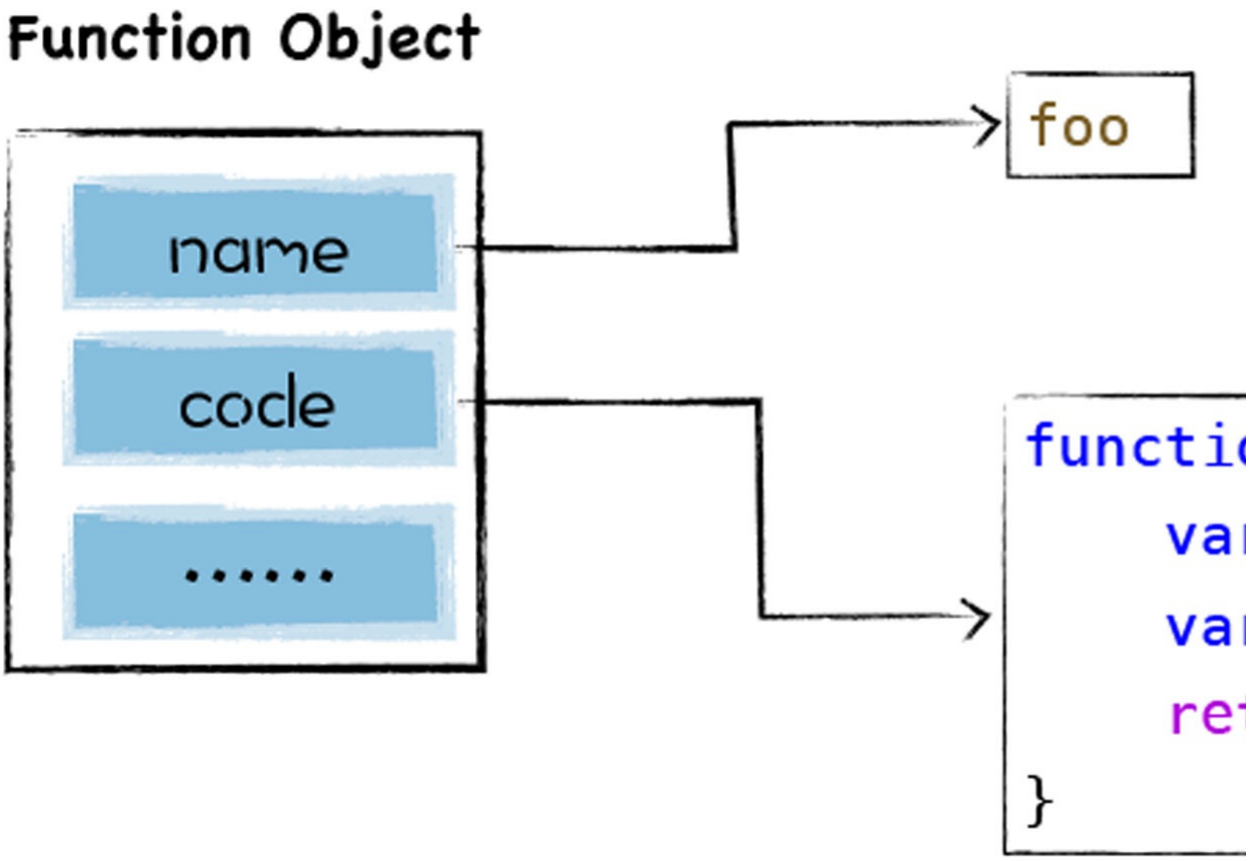
基于以上的原因，所有主流的JavaScript虚拟机都实现了**惰性解析**。所谓惰性解析是指解析器在解析的过程中，如果遇到函数声明，那么会跳过函数内部的代码，并不会为其生成AST和字节码，而仅仅生成顶层代码的AST和字节码。

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：

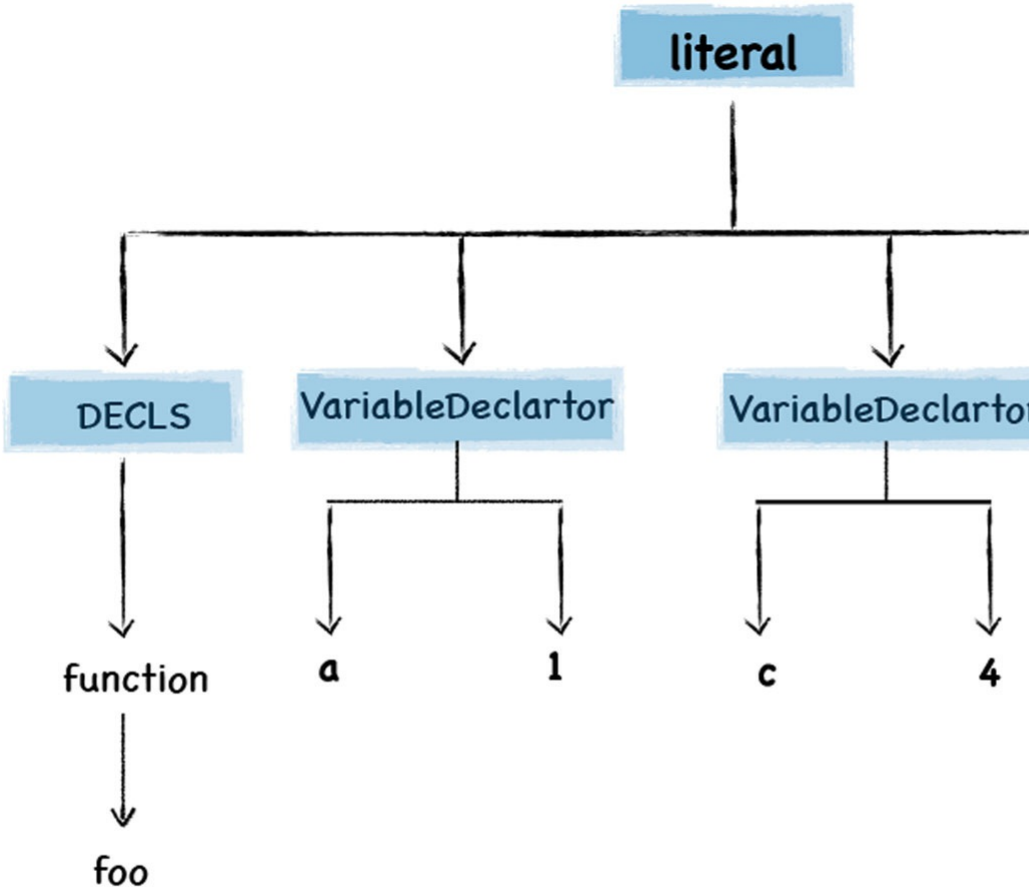
```
function foo(a,b) {
    var d = 100
    var f = 10
    return d + f + a + b;
}
var a = 1
var c = 4
foo(1, 5)
```

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：



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：

```
function foo() {
    function inner() {
    }
    inner()
}
```

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：

```
var d = 20
//inner函数的父函数，词法作用域
function foo() {
    var d = 55
    //foo的内部函数
    function inner() {
      return d+2
    }
    inner()
}
```

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丨作用域链：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：

```
function foo() {
    return function inner(a, b) {
        const c = a + b
        return c
    }
}
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：

```
function foo() {
    var d = 20
    return function inner(a, b) {
        const c = a + b + d
        return c
    }
}
const f = foo()
```

观察上面上面这段代码，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。

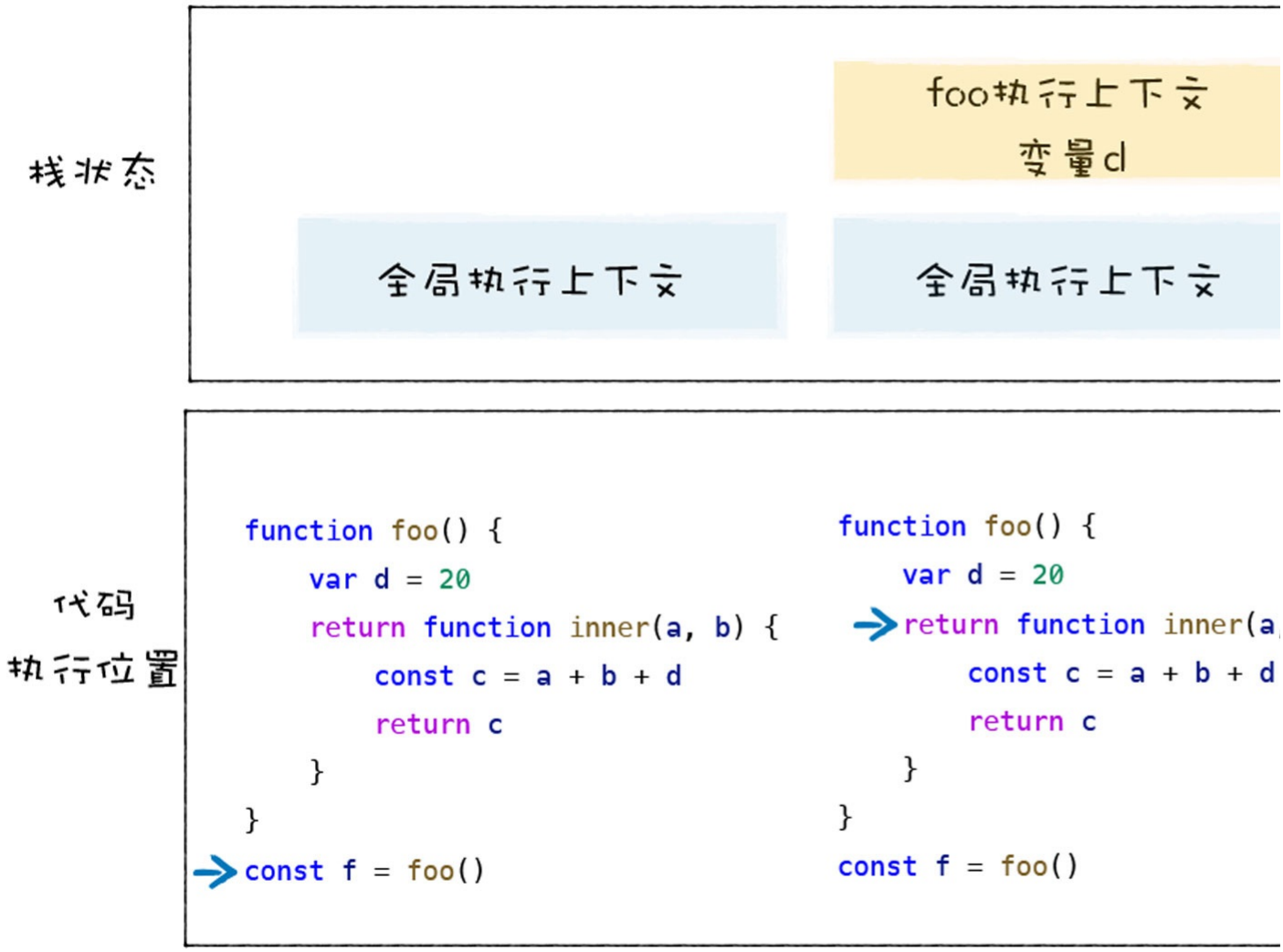
我们可以分析下上面这段代码的执行过程：

- 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；
- 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：



从上图可以看出来，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做预解析器。

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：

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

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：

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

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：

```
function foo() {
   var a = 0
}
```

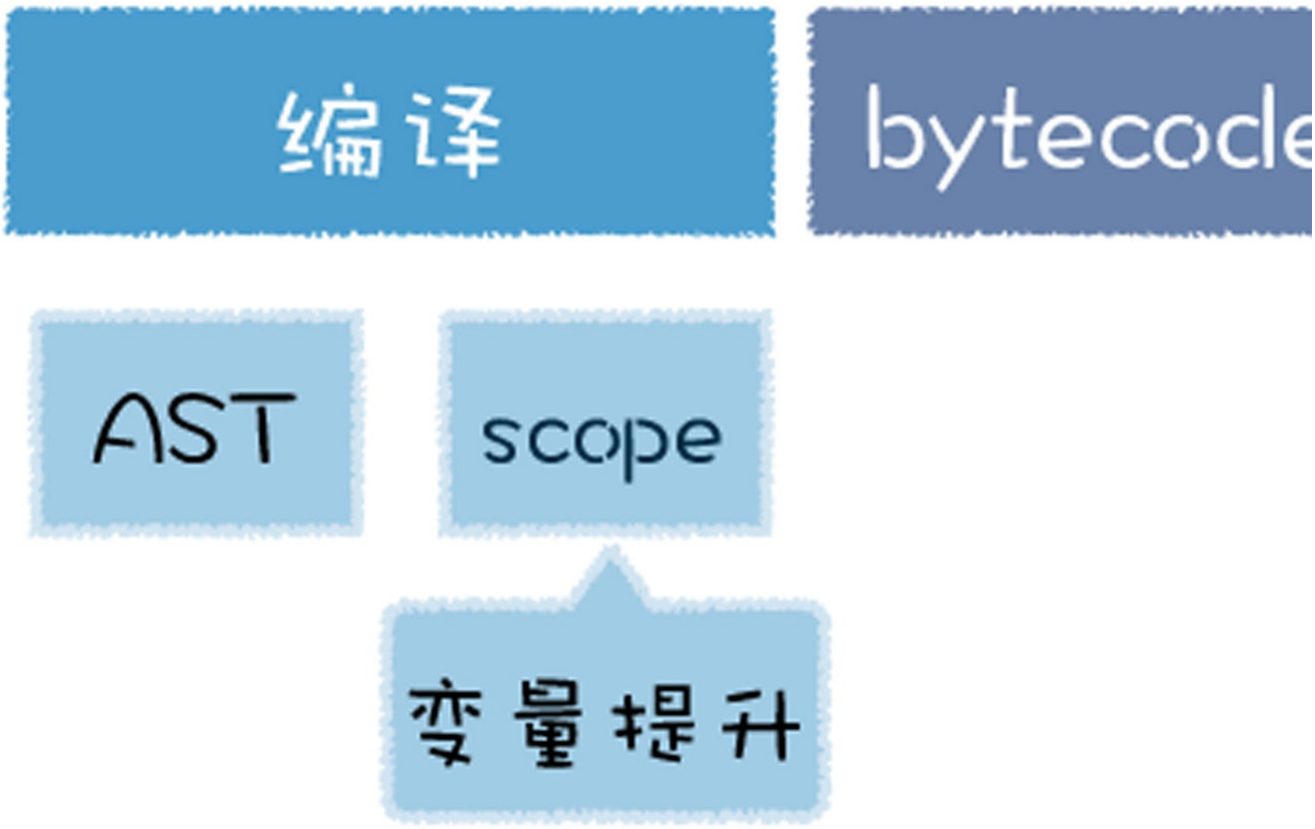
```
function foo() {
    var a = 0
    return function inner() {
        return a++
    }
}
```

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，如果你觉得这一讲的内容对你有所启发，也欢迎把它分享给你的朋友。

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。总的流程你可以参考下图：



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。

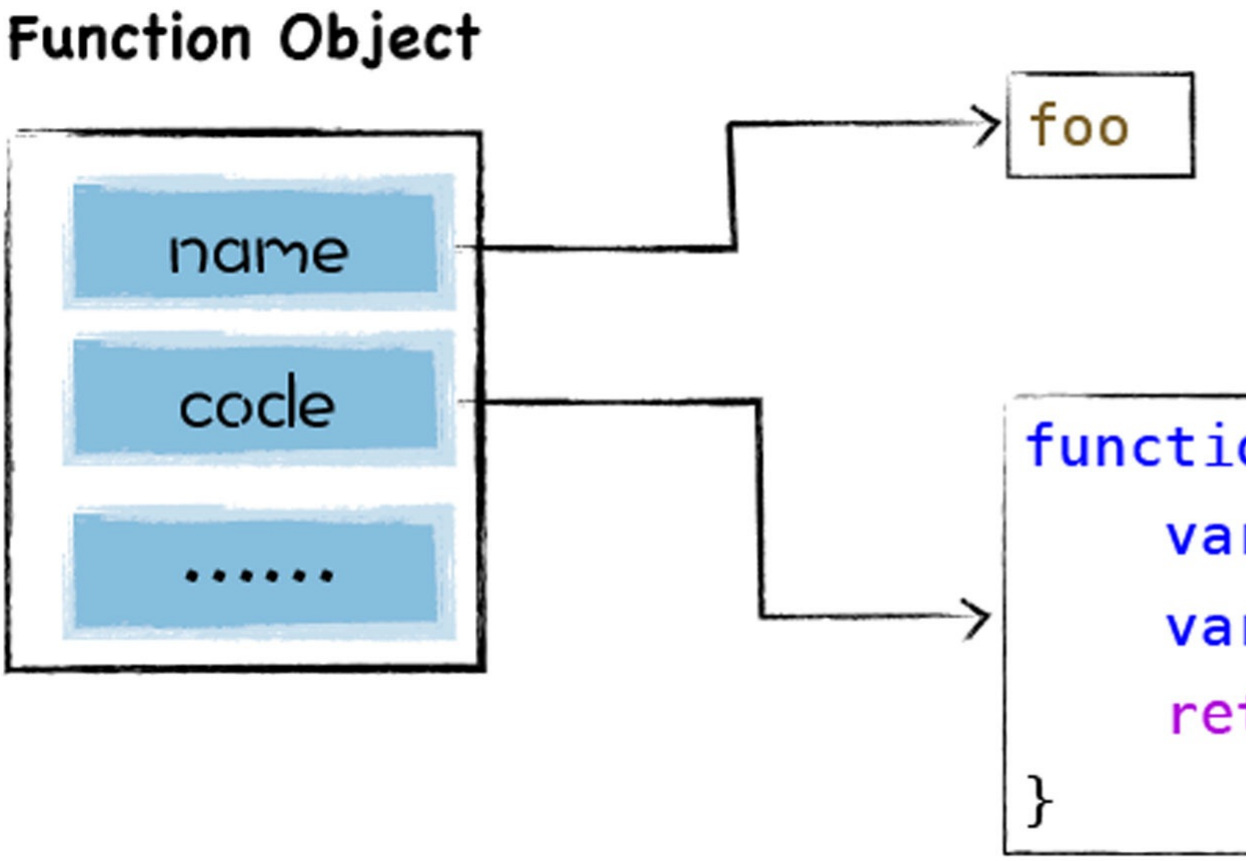
基于以上的原因，所有主流的JavaScript虚拟机都实现了**惰性解析**。所谓惰性解析是指解析器在解析的过程中，如果遇到函数声明，那么会跳过函数内部的代码，并不会为其生成AST和字节码，而仅仅生成顶层代码的AST和字节码。

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：

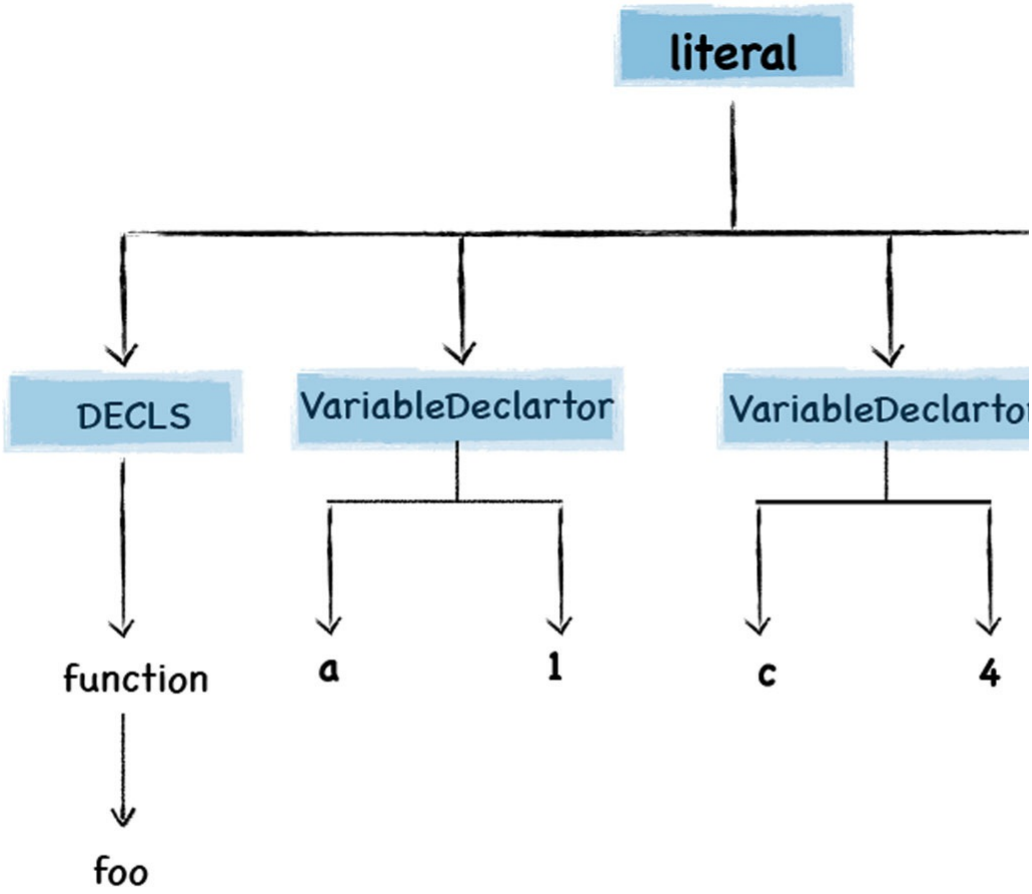
```
function foo(a,b) {
    var d = 100
    var f = 10
    return d + f + a + b;
}
var a = 1
var c = 4
foo(1, 5)
```

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：



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：

```
function foo() {
    function inner() {
    }
    inner()
}
```

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：

```
var d = 20
//inner函数的父函数，词法作用域
function foo() {
    var d = 55
    //foo的内部函数
    function inner() {
      return d+2
    }
    inner()
}
```

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丨作用域链：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：

```
function foo() {
    return function inner(a, b) {
        const c = a + b
        return c
    }
}
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：

```
function foo() {
    var d = 20
    return function inner(a, b) {
        const c = a + b + d
        return c
    }
}
const f = foo()
```

观察上面上面这段代码，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。

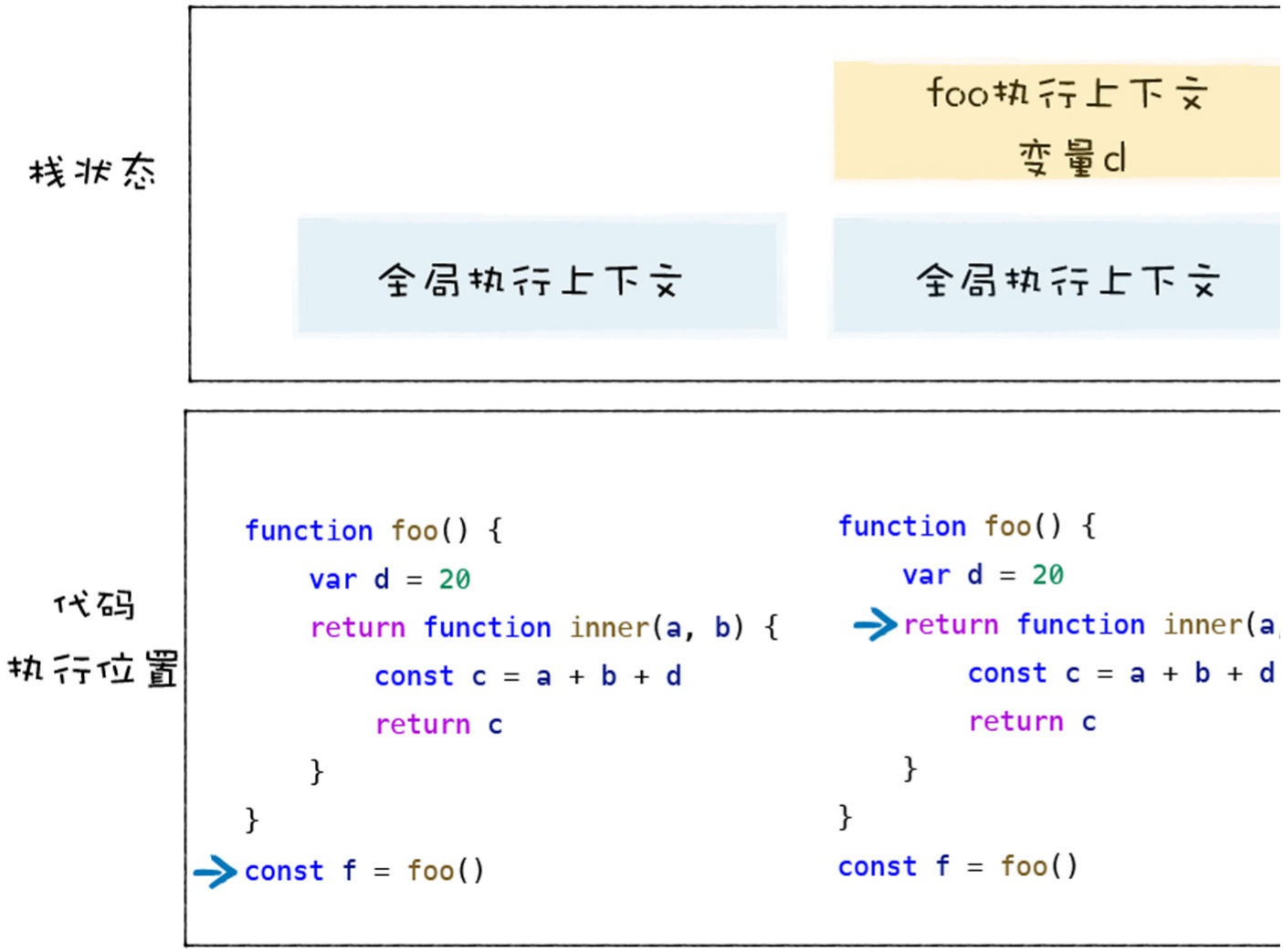
我们可以分析下上面这段代码的执行过程：

- 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；
- 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：



从上图可以看出来，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做预解析器。

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：

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

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：

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

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：

```
function foo() {
   var a = 0
}
```

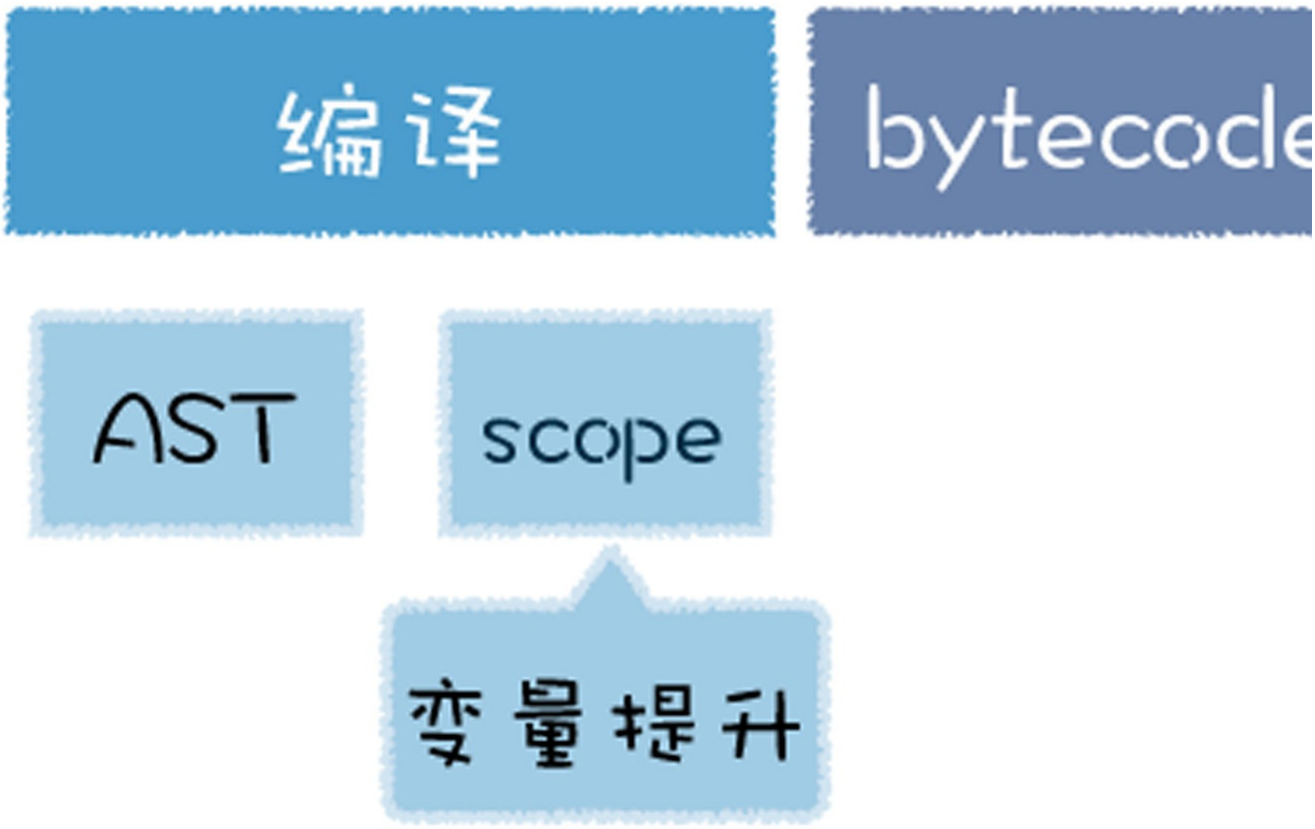
```
function foo() {
    var a = 0
    return function inner() {
        return a++
    }
}
```

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，如果你觉得这一讲的内容对你有所启发，也欢迎把它分享给你的朋友。

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。总的流程你可以参考下图：



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。

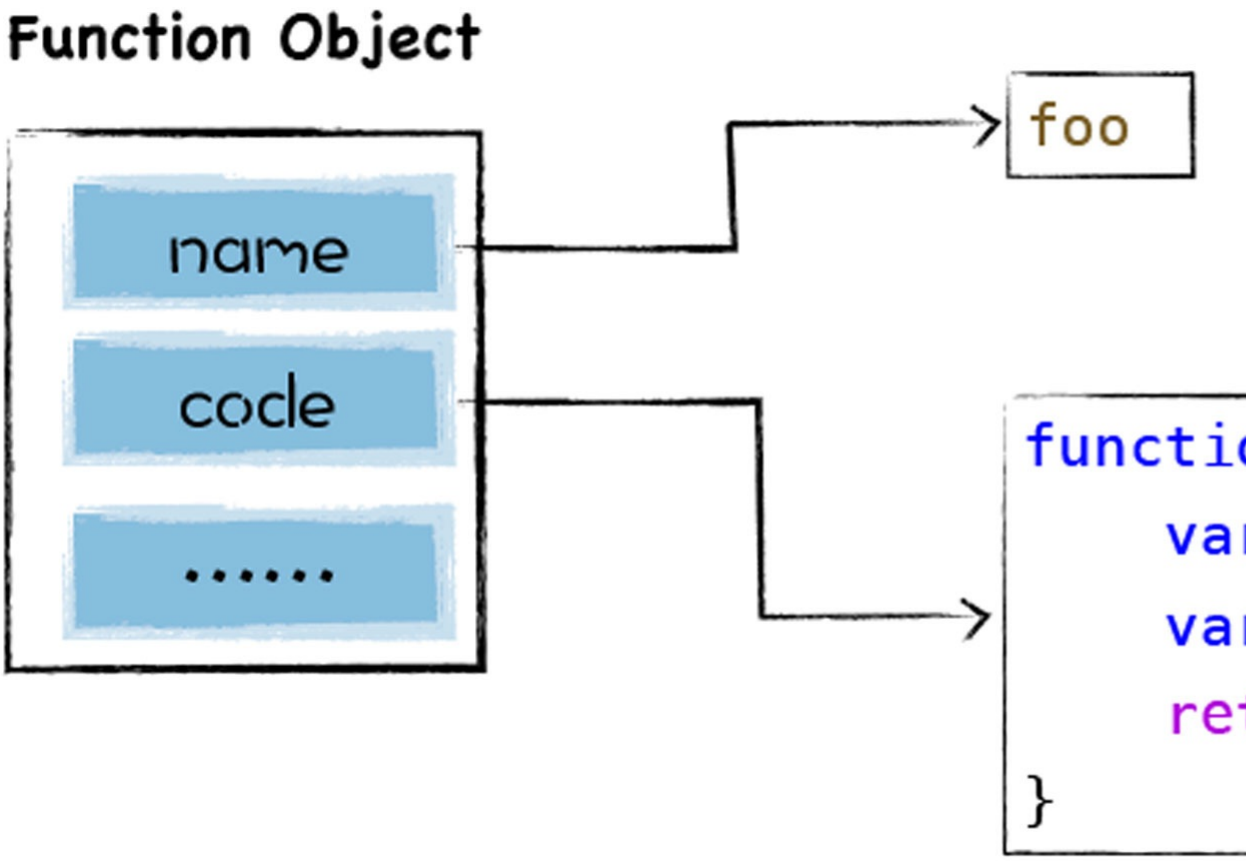
基于以上的原因，所有主流的JavaScript虚拟机都实现了**惰性解析**。所谓惰性解析是指解析器在解析的过程中，如果遇到函数声明，那么会跳过函数内部的代码，并不会为其生成AST和字节码，而仅仅生成顶层代码的AST和字节码。

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：

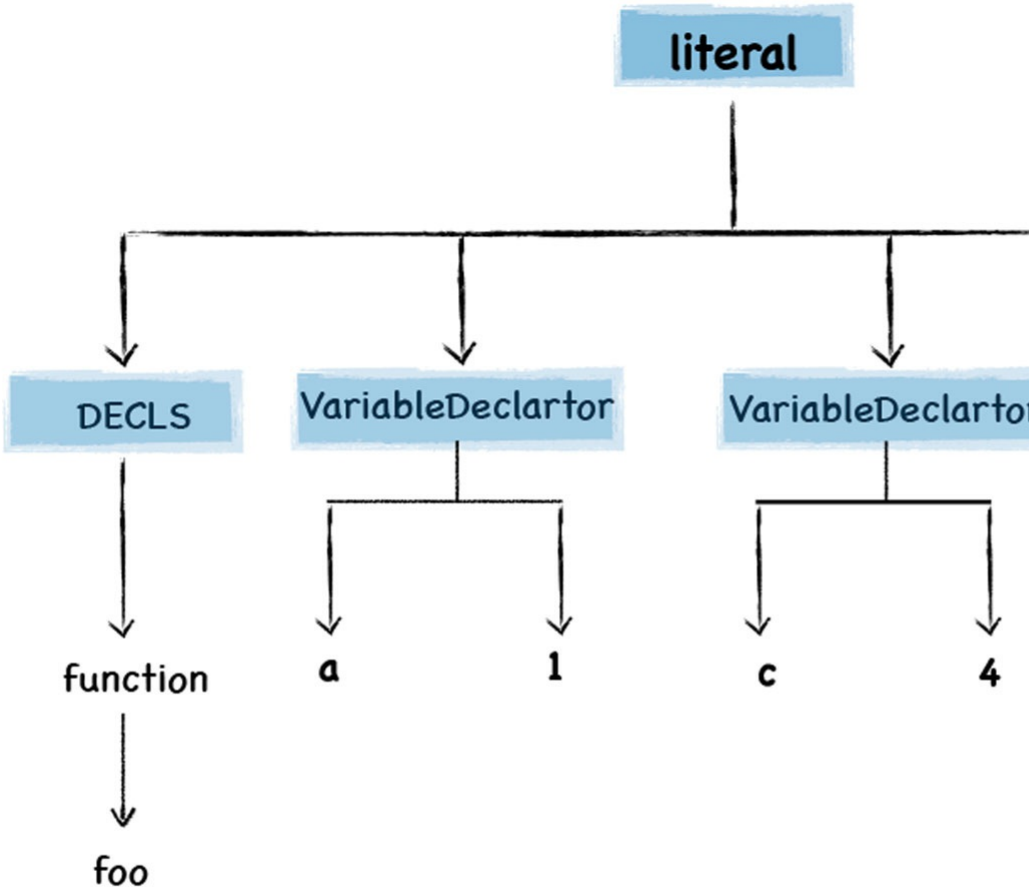
```
function foo(a,b) {
    var d = 100
    var f = 10
    return d + f + a + b;
}
var a = 1
var c = 4
foo(1, 5)
```

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：



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：

```
function foo() {
    function inner() {
    }
    inner()
}
```

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：

```
var d = 20
//inner函数的父函数，词法作用域
function foo() {
    var d = 55
    //foo的内部函数
    function inner() {
      return d+2
    }
    inner()
}
```

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丨作用域链：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：

```
function foo() {
    return function inner(a, b) {
        const c = a + b
        return c
    }
}
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：

```
function foo() {
    var d = 20
    return function inner(a, b) {
        const c = a + b + d
        return c
    }
}
const f = foo()
```

观察上面上面这段代码，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。

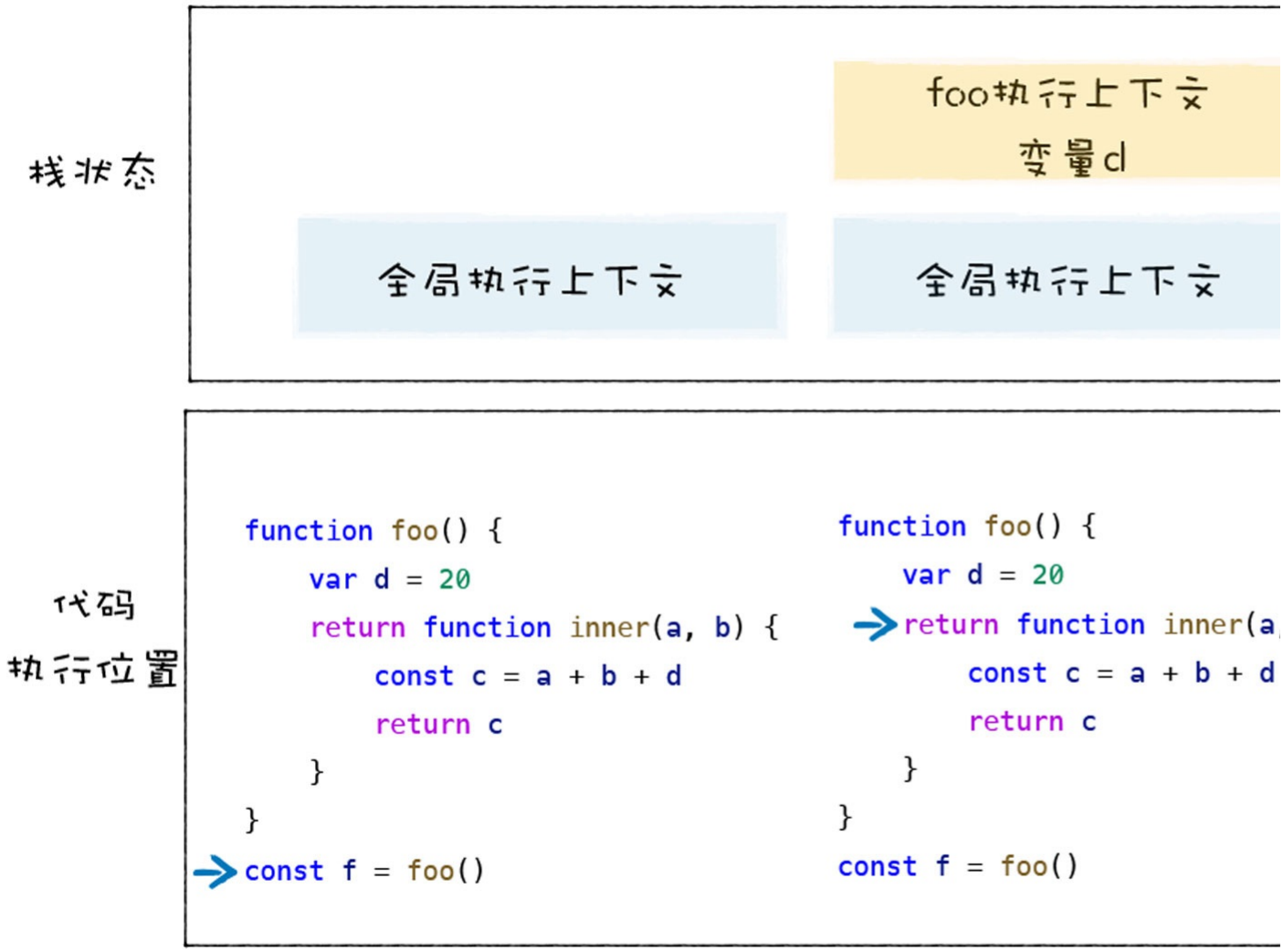
我们可以分析下上面这段代码的执行过程：

- 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；
- 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：



从上图可以看出来，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做预解析器。

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：

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

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：

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

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：

```
function foo() {
   var a = 0
}
```

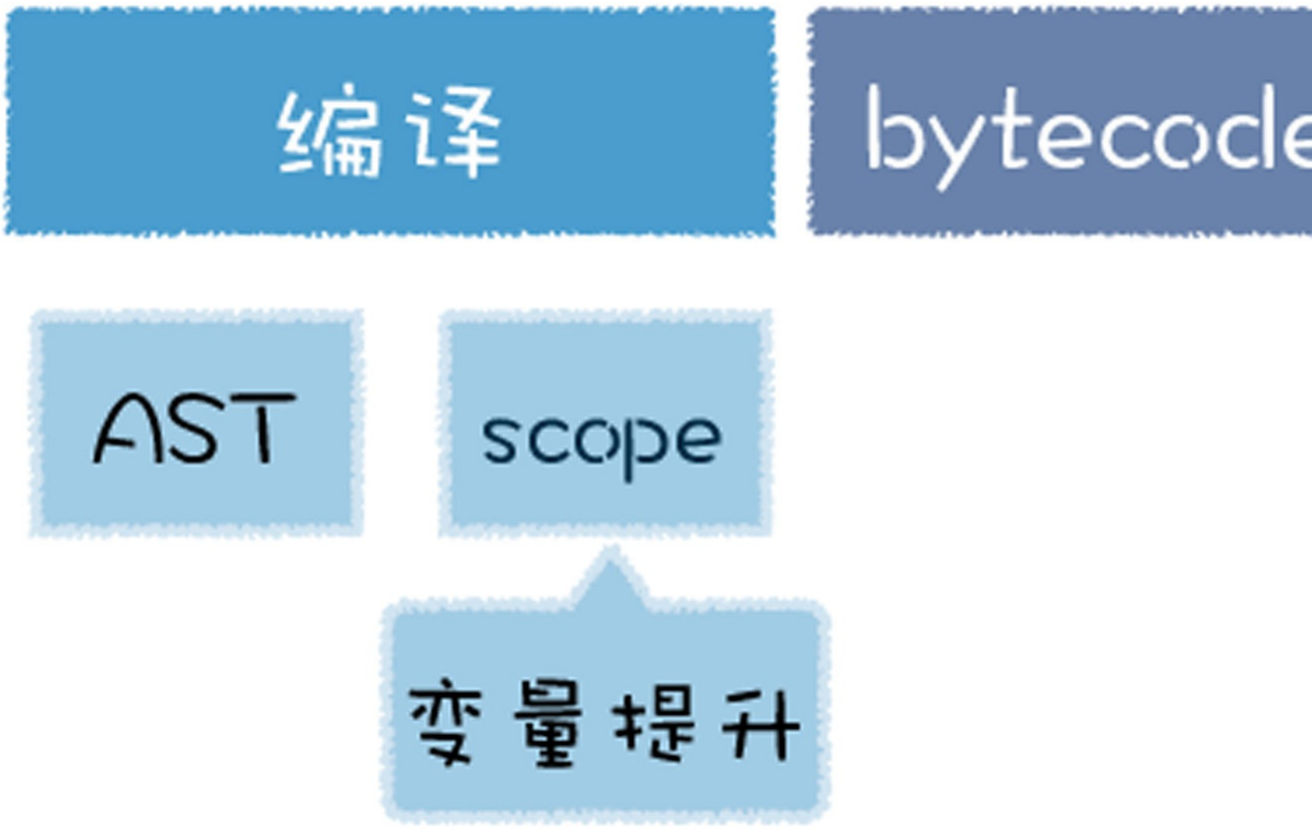
```
function foo() {
    var a = 0
    return function inner() {
        return a++
    }
}
```

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，如果你觉得这一讲的内容对你有所启发，也欢迎把它分享给你的朋友。

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。总的流程你可以参考下图：



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。

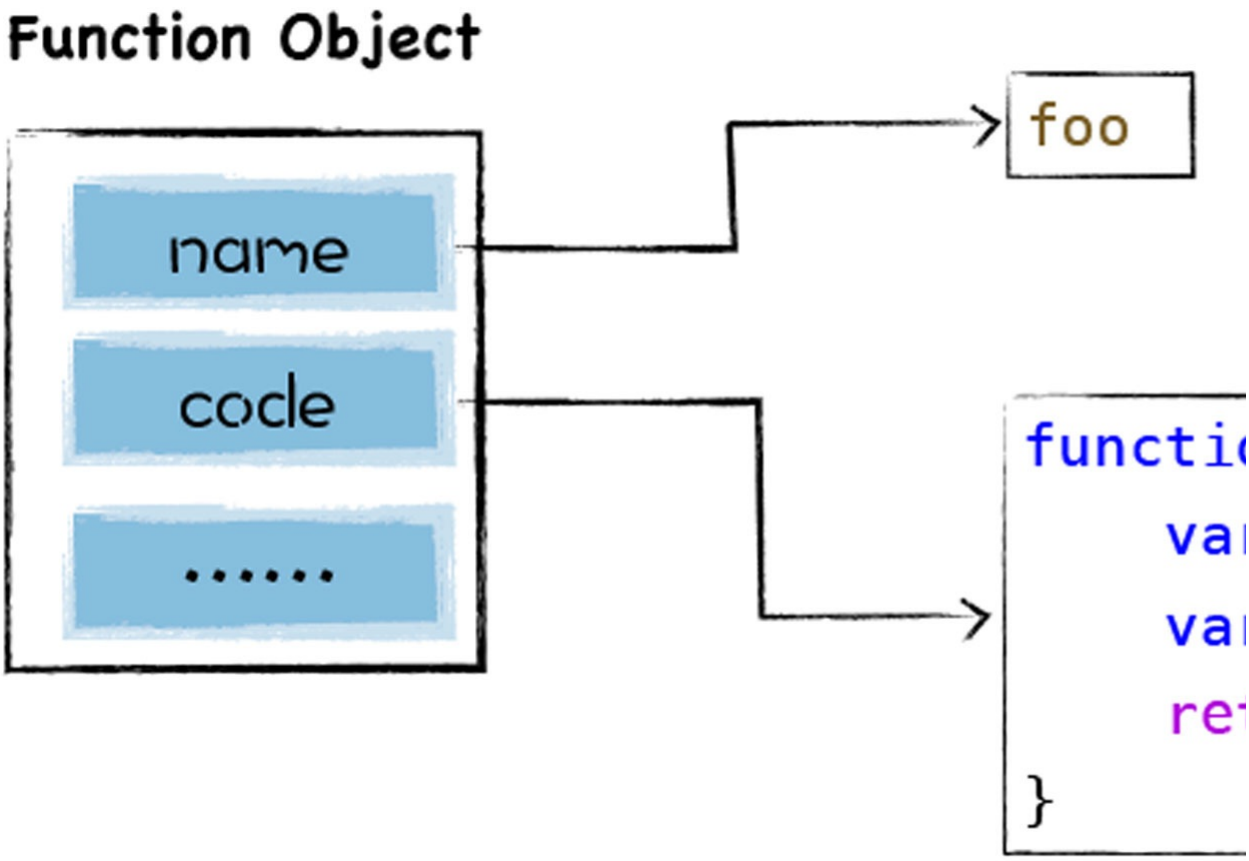
基于以上的原因，所有主流的JavaScript虚拟机都实现了**惰性解析**。所谓惰性解析是指解析器在解析的过程中，如果遇到函数声明，那么会跳过函数内部的代码，并不会为其生成AST和字节码，而仅仅生成顶层代码的AST和字节码。

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：

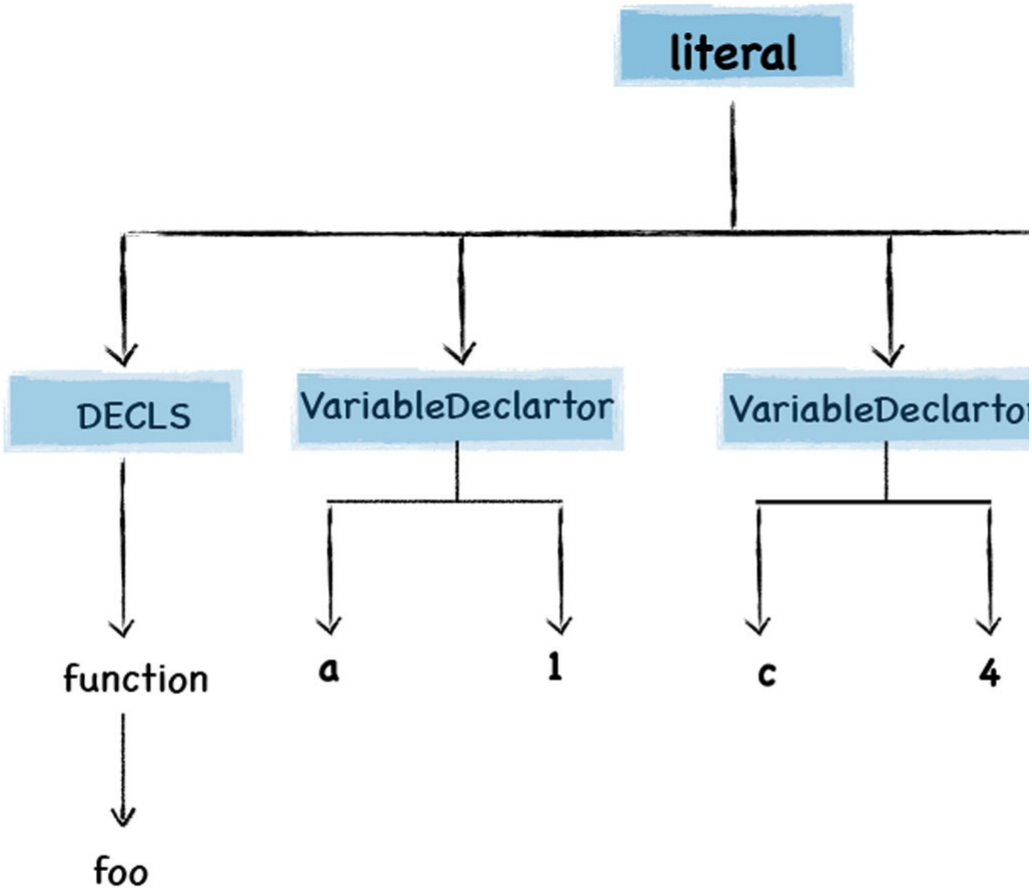
```
function foo(a,b) {
    var d = 100
    var f = 10
    return d + f + a + b;
}
var a = 1
var c = 4
foo(1, 5)
```

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：



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：

```
function foo() {
    function inner() {
    }
    inner()
}
```

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：

```
var d = 20
//inner函数的父函数，词法作用域
function foo() {
    var d = 55
    //foo的内部函数
    function inner() {
      return d+2
    }
    inner()
}
```

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丨作用域链：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：

```
function foo() {
    return function inner(a, b) {
        const c = a + b
        return c
    }
}
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：

```
function foo() {
    var d = 20
    return function inner(a, b) {
        const c = a + b + d
        return c
    }
}
const f = foo()
```

观察上面上面这段代码，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。

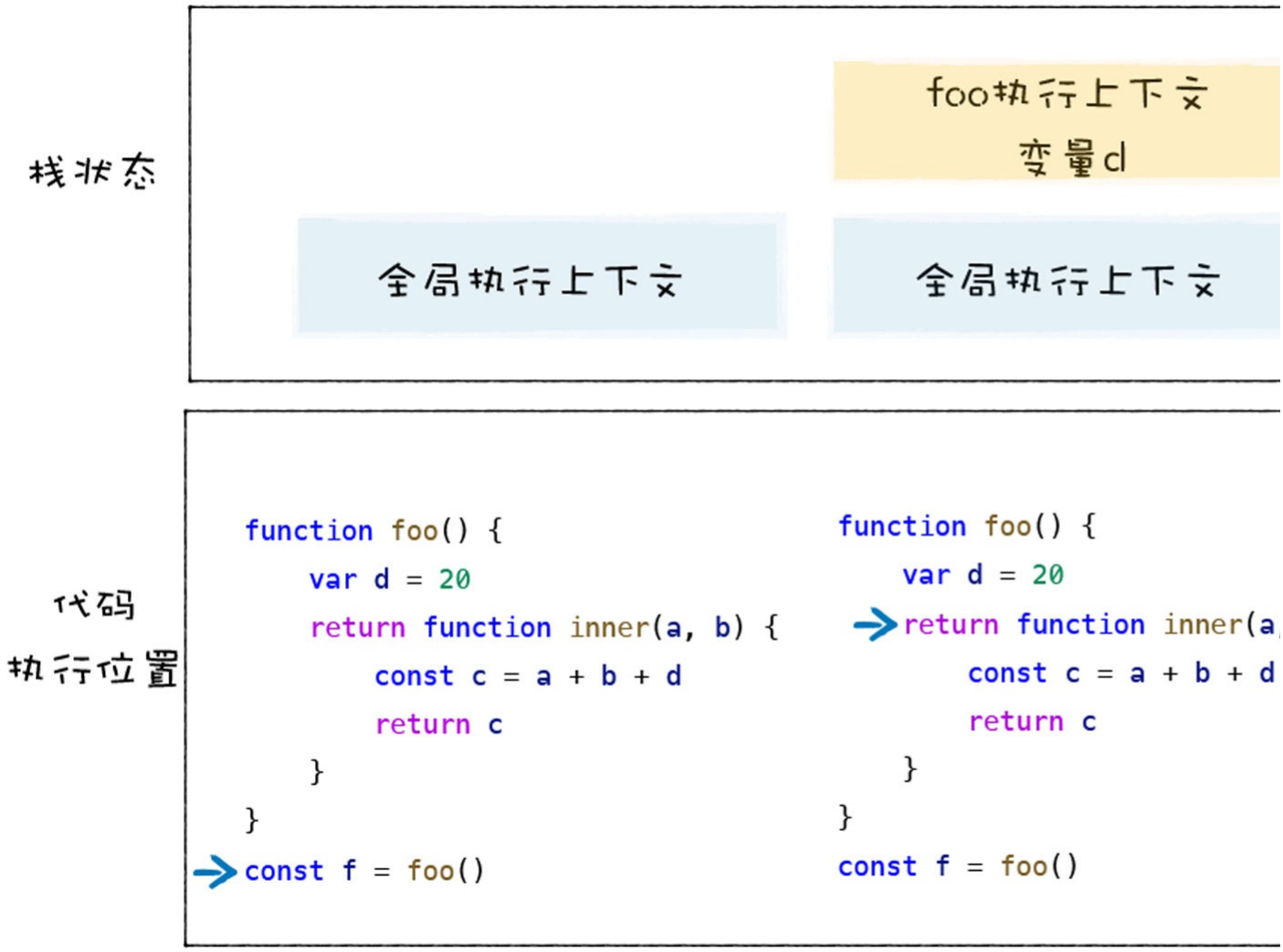
我们可以分析下上面这段代码的执行过程：

- 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；
- 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：



从上图可以看出来，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做预解析器。

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：

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

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：

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

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：

```
function foo() {
   var a = 0
}
```

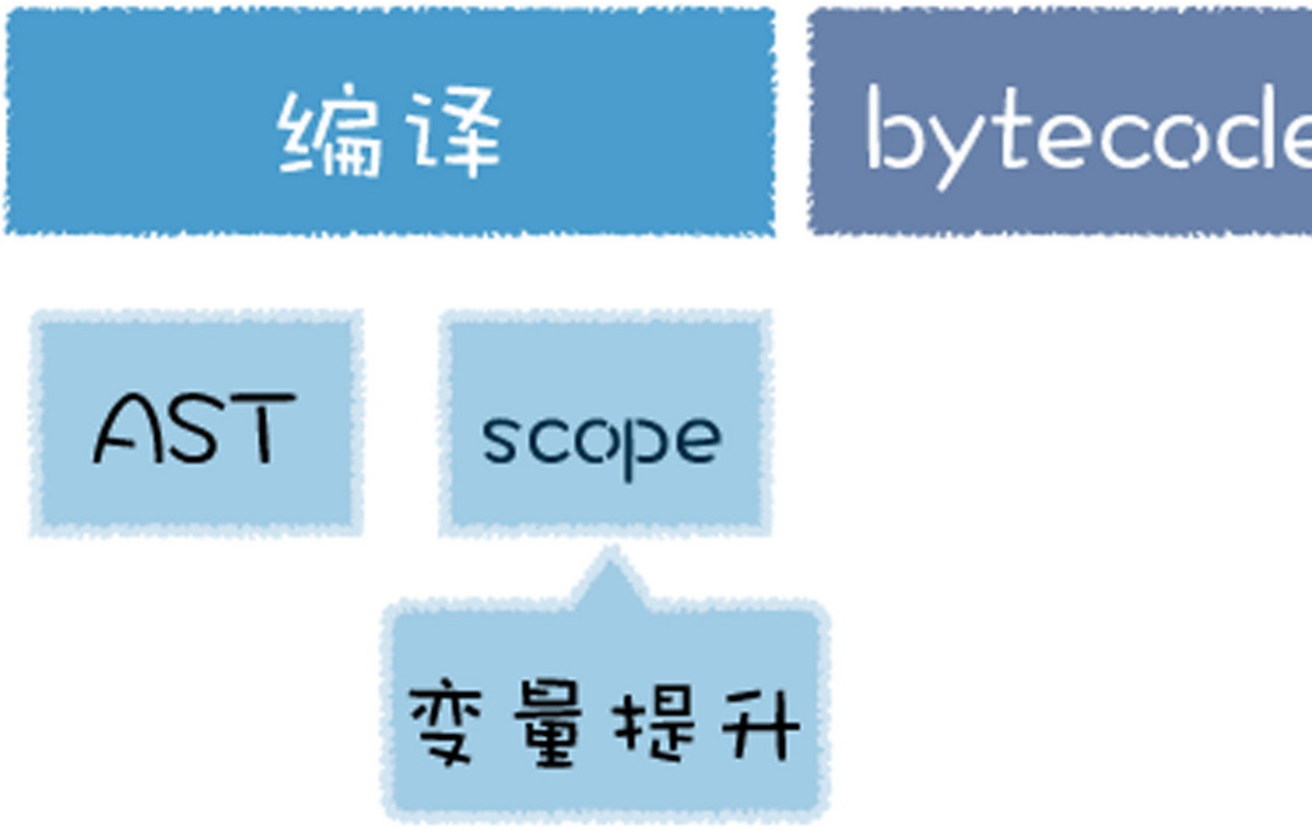
```
function foo() {
    var a = 0
    return function inner() {
        return a++
    }
}
```

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，如果你觉得这一讲的内容对你有所启发，也欢迎把它分享给你的朋友。

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。总的流程你可以参考下图：



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。

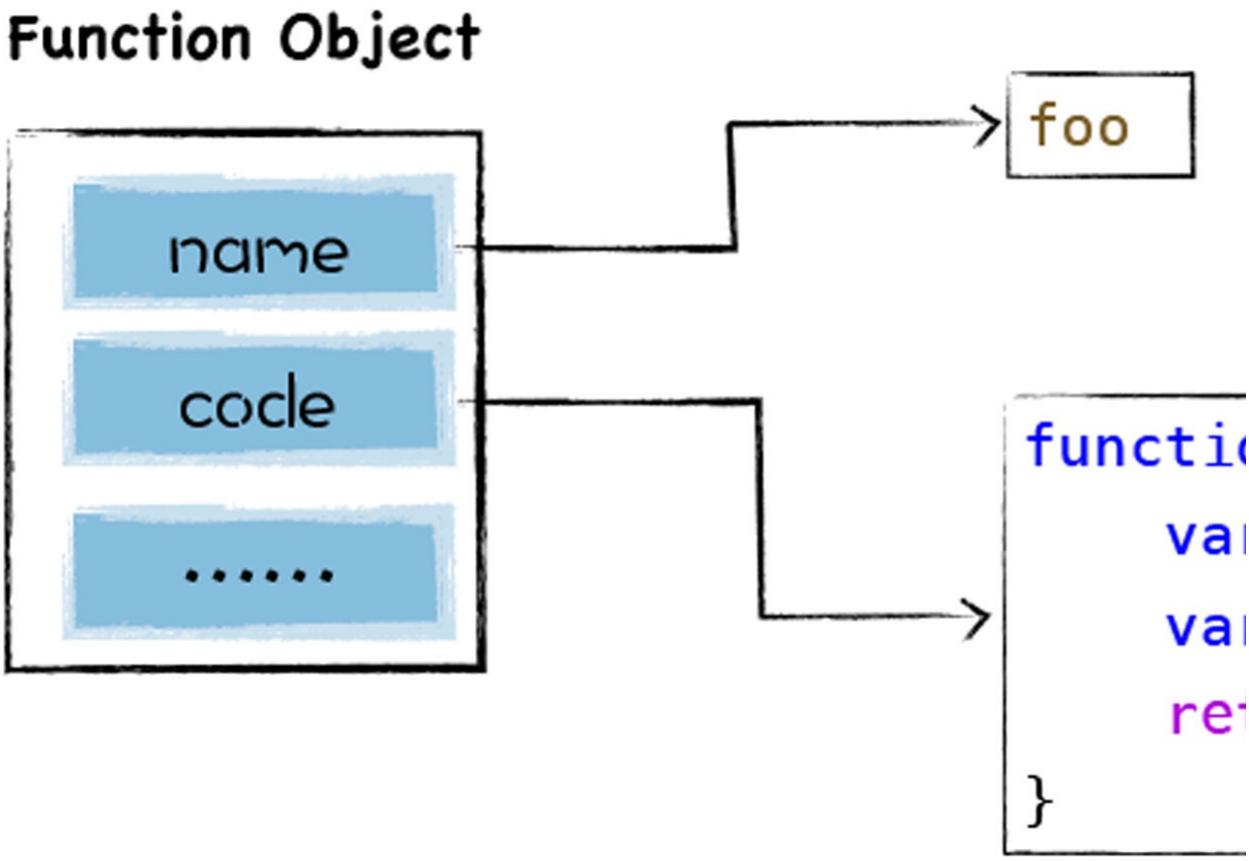
基于以上的原因，所有主流的JavaScript虚拟机都实现了**惰性解析**。所谓惰性解析是指解析器在解析的过程中，如果遇到函数声明，那么会跳过函数内部的代码，并不会为其生成AST和字节码，而仅仅生成顶层代码的AST和字节码。

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：

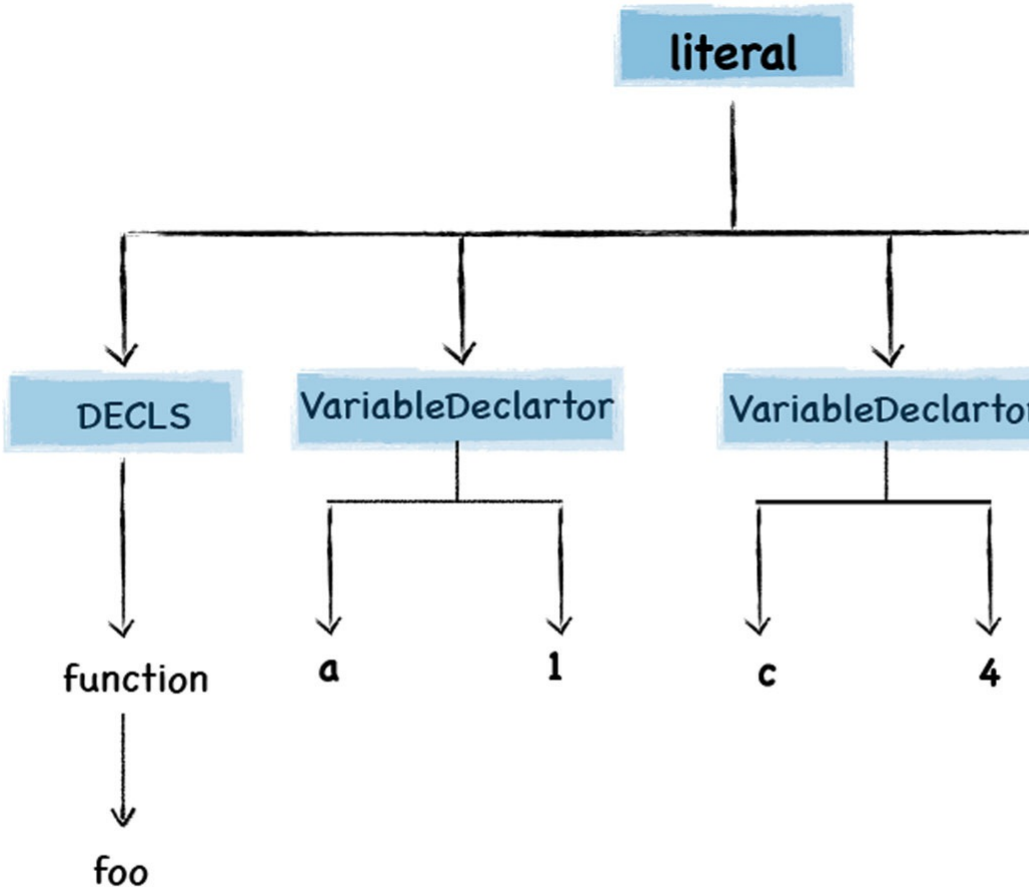
```
function foo(a,b) {
    var d = 100
    var f = 10
    return d + f + a + b;
}
var a = 1
var c = 4
foo(1, 5)
```

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：



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：

```
function foo() {
    function inner() {
    }
    inner()
}
```

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：

```
var d = 20
//inner函数的父函数，词法作用域
function foo() {
    var d = 55
    //foo的内部函数
    function inner() {
      return d+2
    }
    inner()
}
```

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丨作用域链：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：

```
function foo() {
    return function inner(a, b) {
        const c = a + b
        return c
    }
}
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：

```
function foo() {
    var d = 20
    return function inner(a, b) {
        const c = a + b + d
        return c
    }
}
const f = foo()
```

观察上面上面这段代码，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。

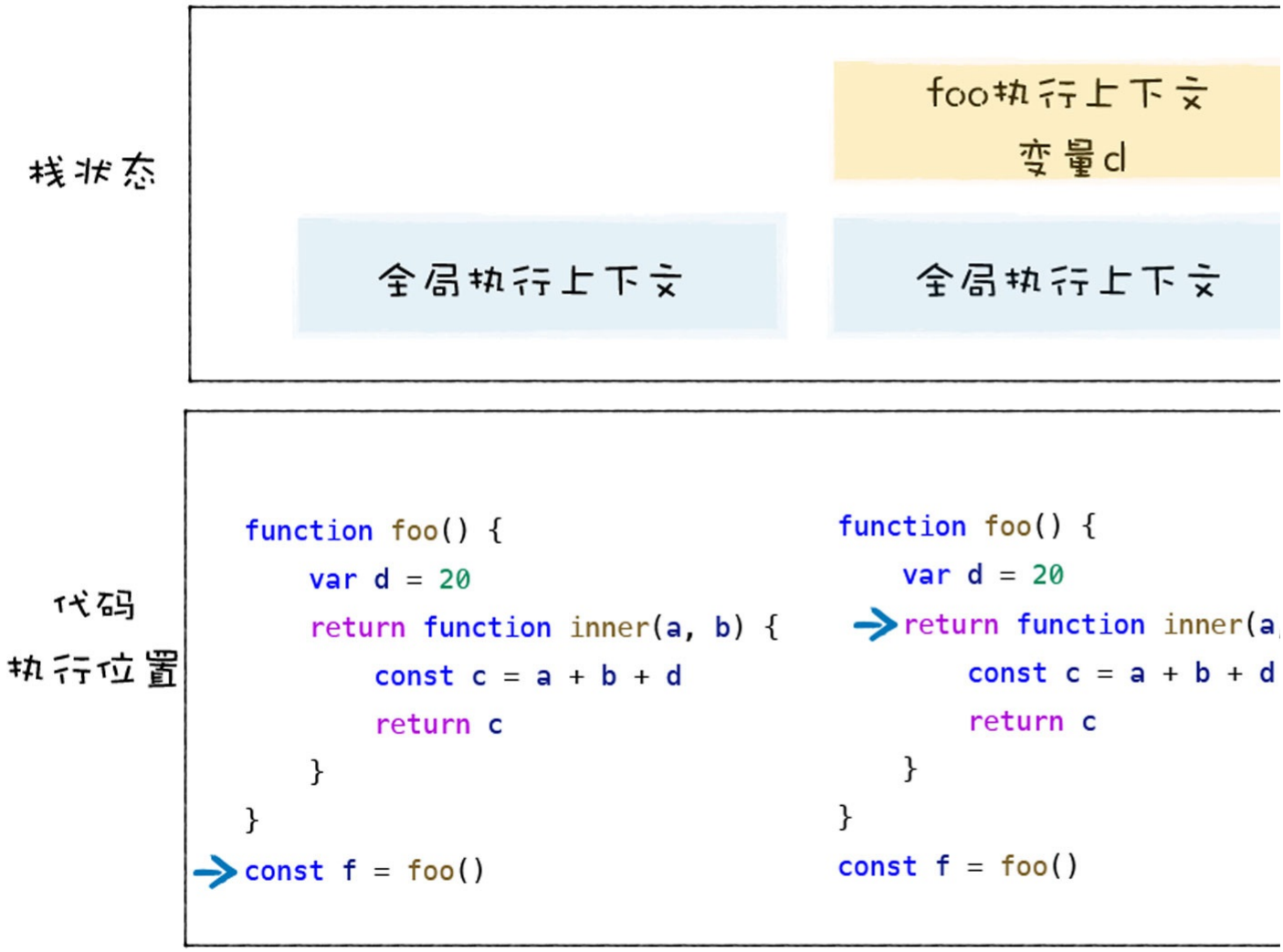
我们可以分析下上面这段代码的执行过程：

- 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；
- 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：



从上图可以看出来，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做预解析器。

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：

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

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：

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

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：

```
function foo() {
   var a = 0
}
```

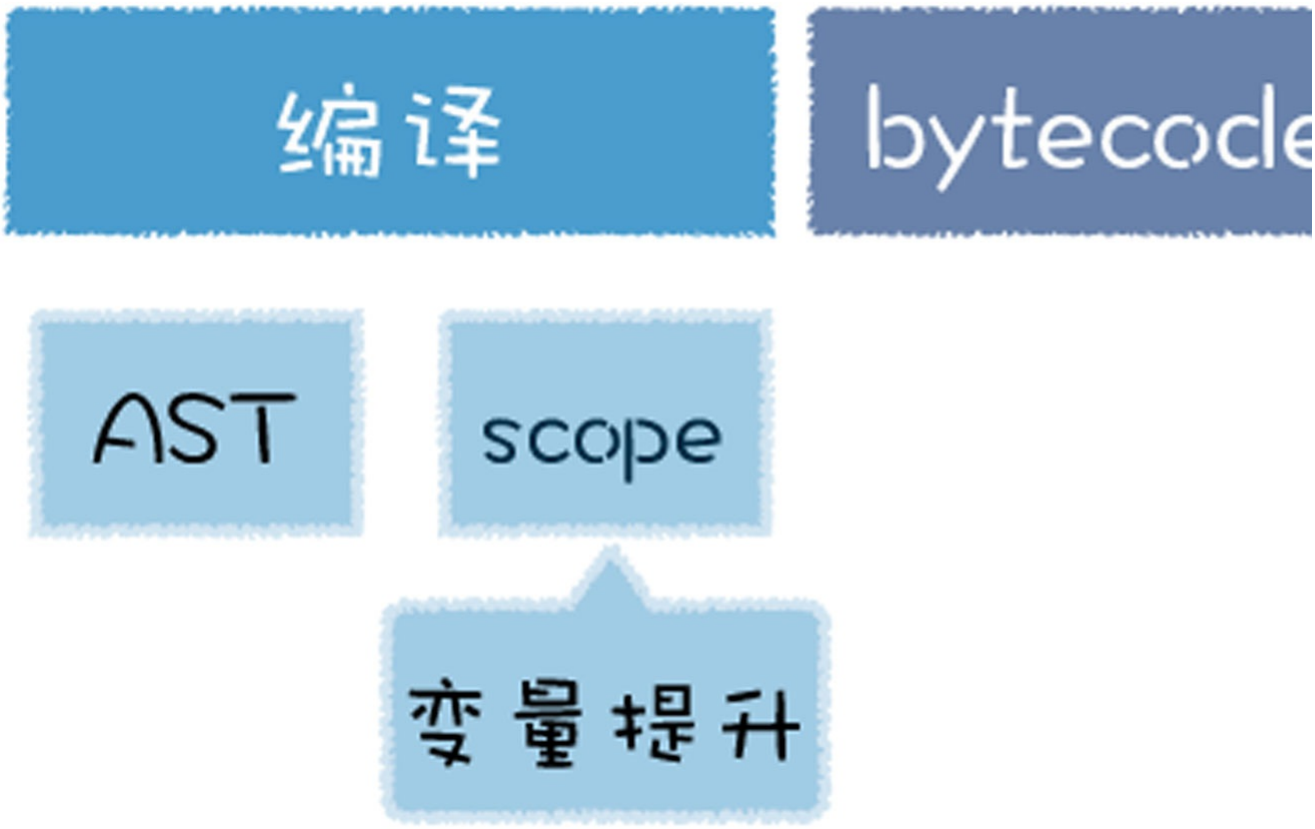
```
function foo() {
    var a = 0
    return function inner() {
        return a++
    }
}
```

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，如果你觉得这一讲的内容对你有所启发，也欢迎把它分享给你的朋友。

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。总的流程你可以参考下图：



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。

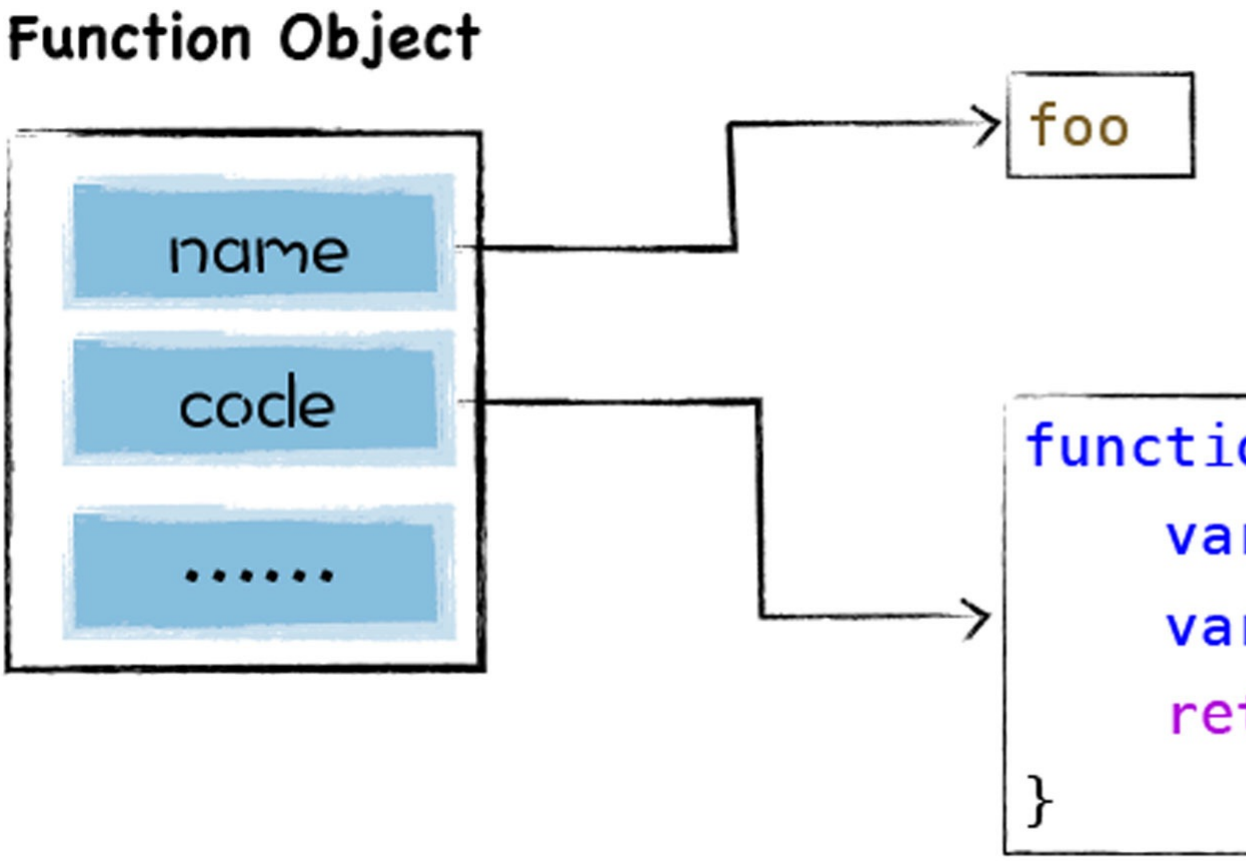
基于以上的原因，所有主流的JavaScript虚拟机都实现了**惰性解析**。所谓惰性解析是指解析器在解析的过程中，如果遇到函数声明，那么会跳过函数内部的代码，并不会为其生成AST和字节码，而仅仅生成顶层代码的AST和字节码。

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：

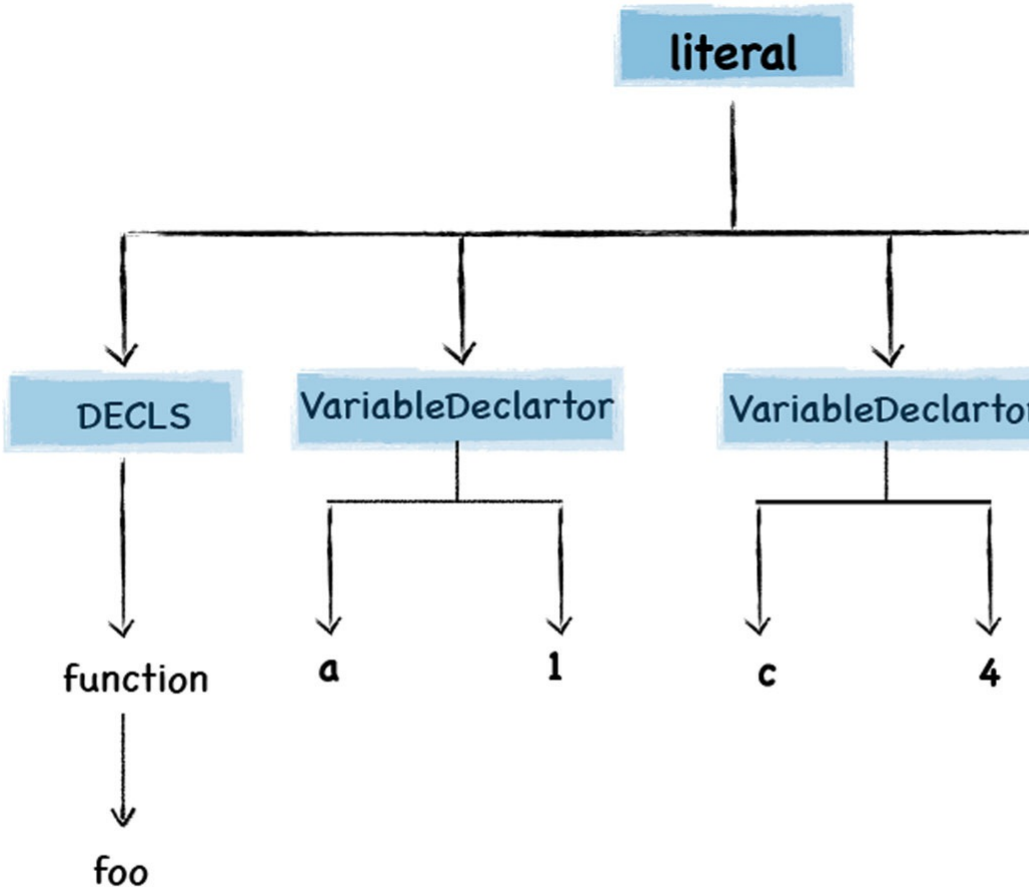
```
function foo(a,b) {
    var d = 100
    var f = 10
    return d + f + a + b;
}
var a = 1
var c = 4
foo(1, 5)
```

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：



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：

```
function foo() {
    function inner() {
    }
    inner()
}
```

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：

```
var d = 20
//inner函数的父函数，词法作用域
function foo() {
    var d = 55
    //foo的内部函数
    function inner() {
      return d+2
    }
    inner()
}
```

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丨作用域链：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：

```
function foo() {
    return function inner(a, b) {
        const c = a + b
        return c
    }
}
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：

```
function foo() {
    var d = 20
    return function inner(a, b) {
        const c = a + b + d
        return c
    }
}
const f = foo()
```

观察上面上面这段代码，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。

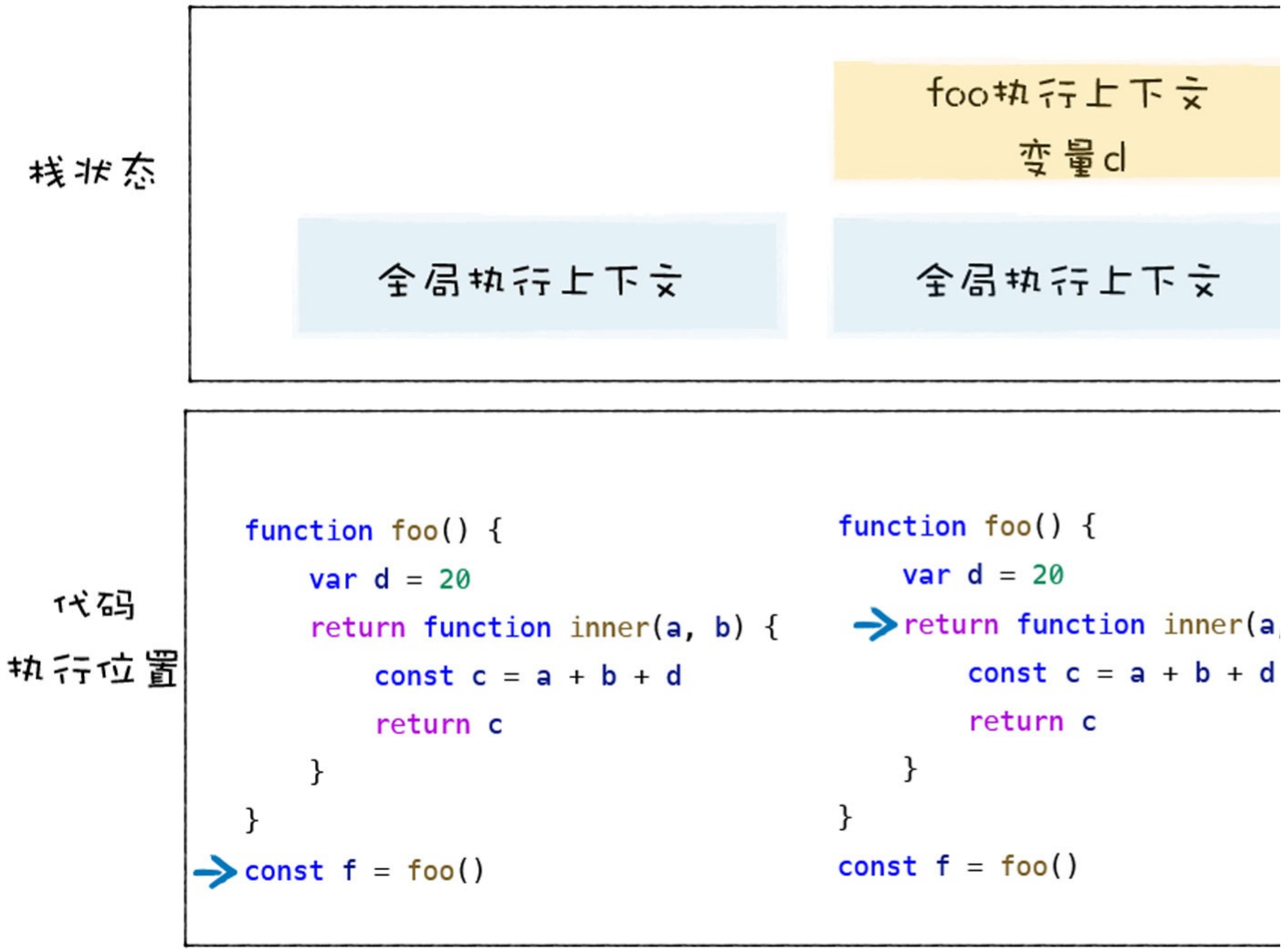
我们可以分析下上面这段代码的执行过程：

- 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；
- 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：



从上图可以看出来，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做预解析器。

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：

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

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：

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

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：

```
function foo() {
   var a = 0
}
```

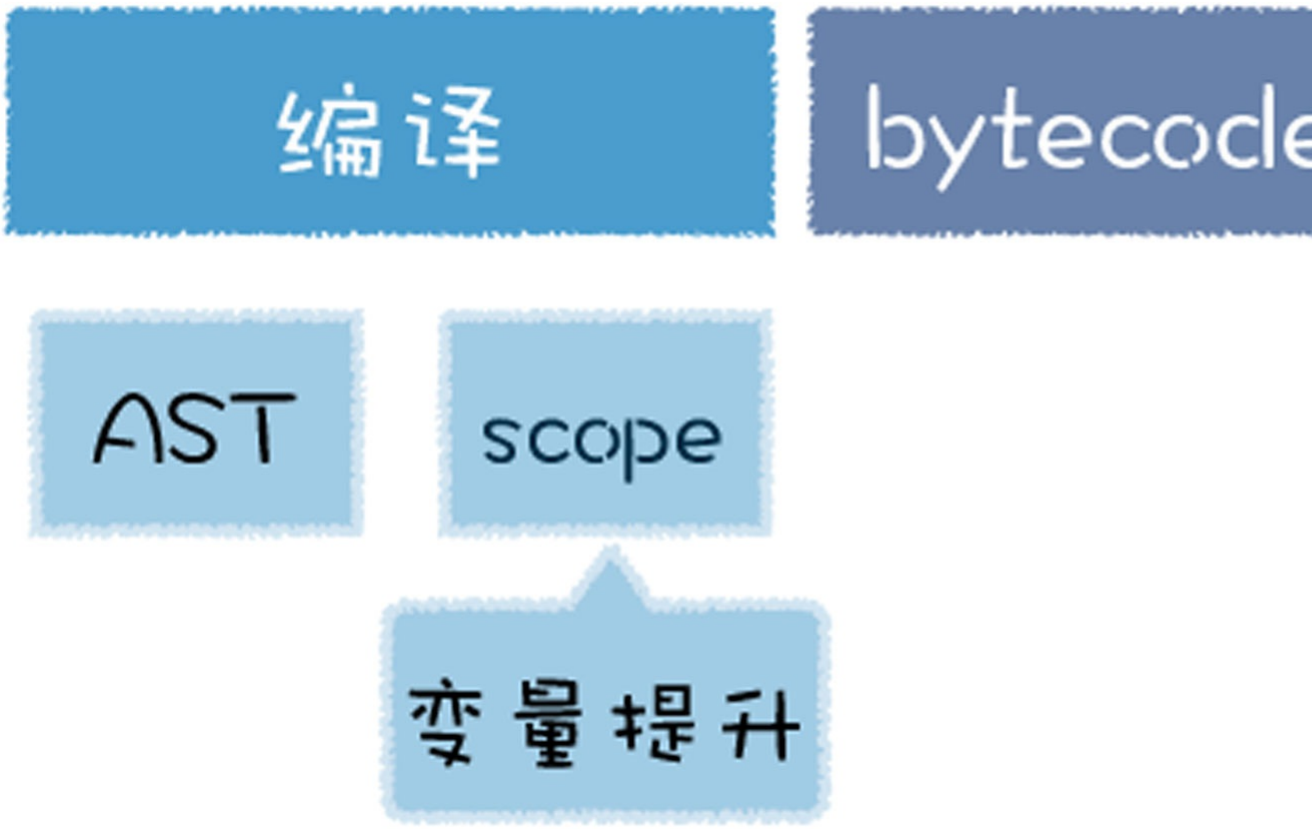
```
function foo() {
    var a = 0
    return function inner() {
        return a++
    }
}
```

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，如果你觉得这一讲的内容对你有所启发，也欢迎把它分享给你的朋友。

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。总的流程你可以参考下图：



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。

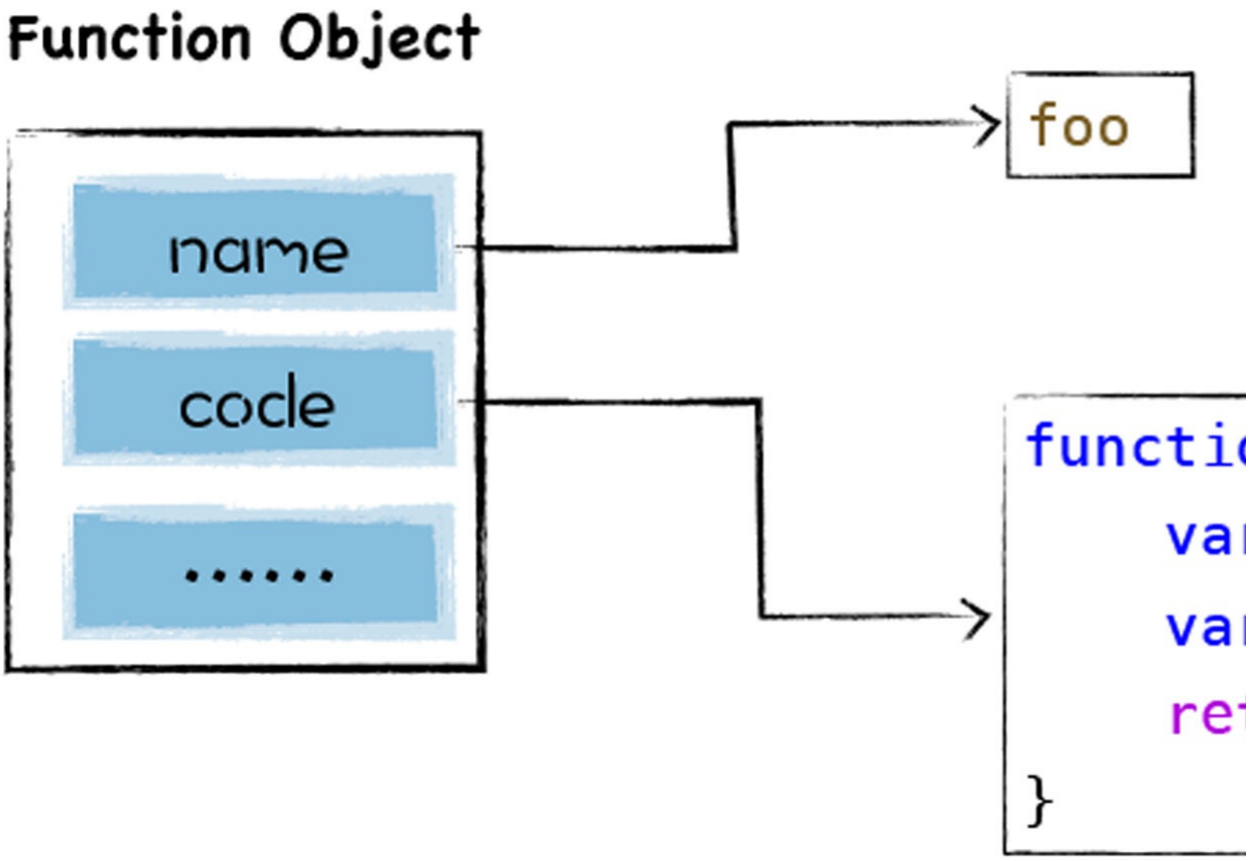
基于以上的原因，所有主流的JavaScript虚拟机都实现了**惰性解析**。所谓惰性解析是指解析器在解析的过程中，如果遇到函数声明，那么会跳过函数内部的代码，并不会为其生成AST和字节码，而仅仅生成顶层代码的AST和字节码。

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：

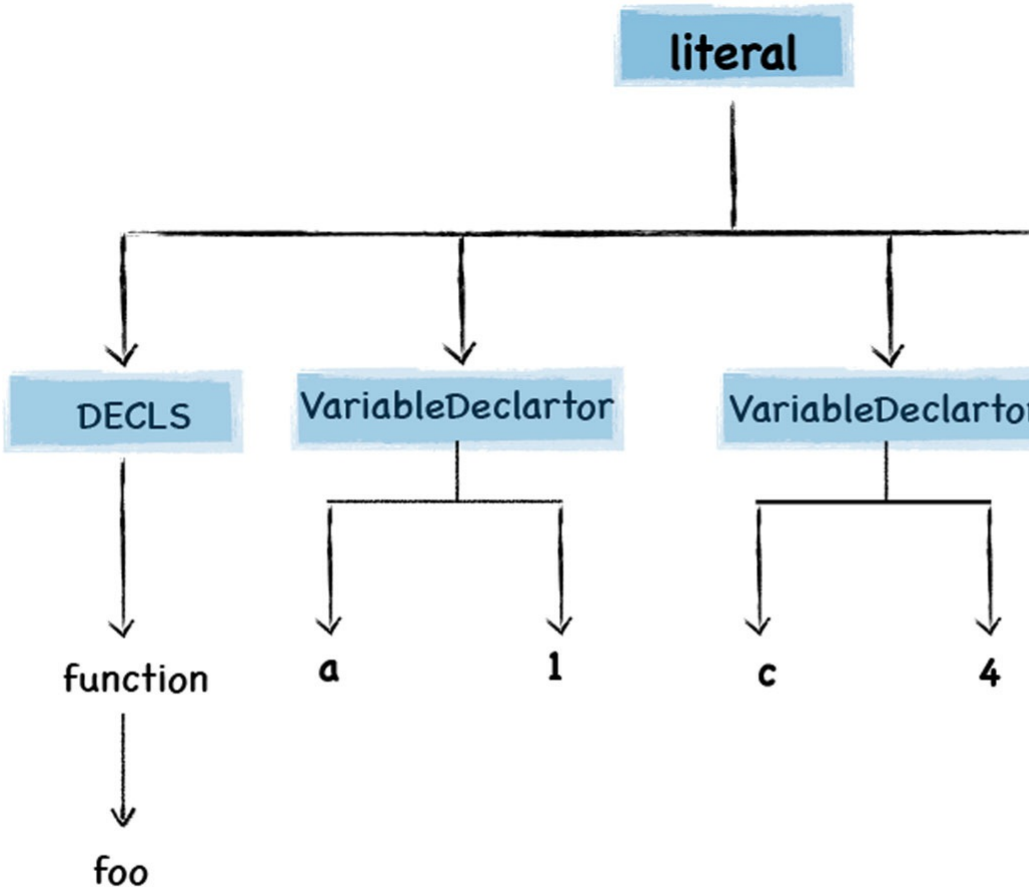
```
function foo(a,b) {
    var d = 100
    var f = 10
    return d + f + a + b;
}
var a = 1
var c = 4
foo(1, 5)
```

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：



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：

```
function foo() {
    function inner() {
    }
    inner()
}
```

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：

```
var d = 20
//inner函数的父函数，词法作用域
function foo() {
    var d = 55
    //foo的内部函数
    function inner() {
      return d+2
    }
    inner()
}
```

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丨作用域链：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：

```
function foo() {
    return function inner(a, b) {
        const c = a + b
        return c
    }
}
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：

```
function foo() {
    var d = 20
    return function inner(a, b) {
        const c = a + b + d
        return c
    }
}
const f = foo()
```

观察上面上面这段代码，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。

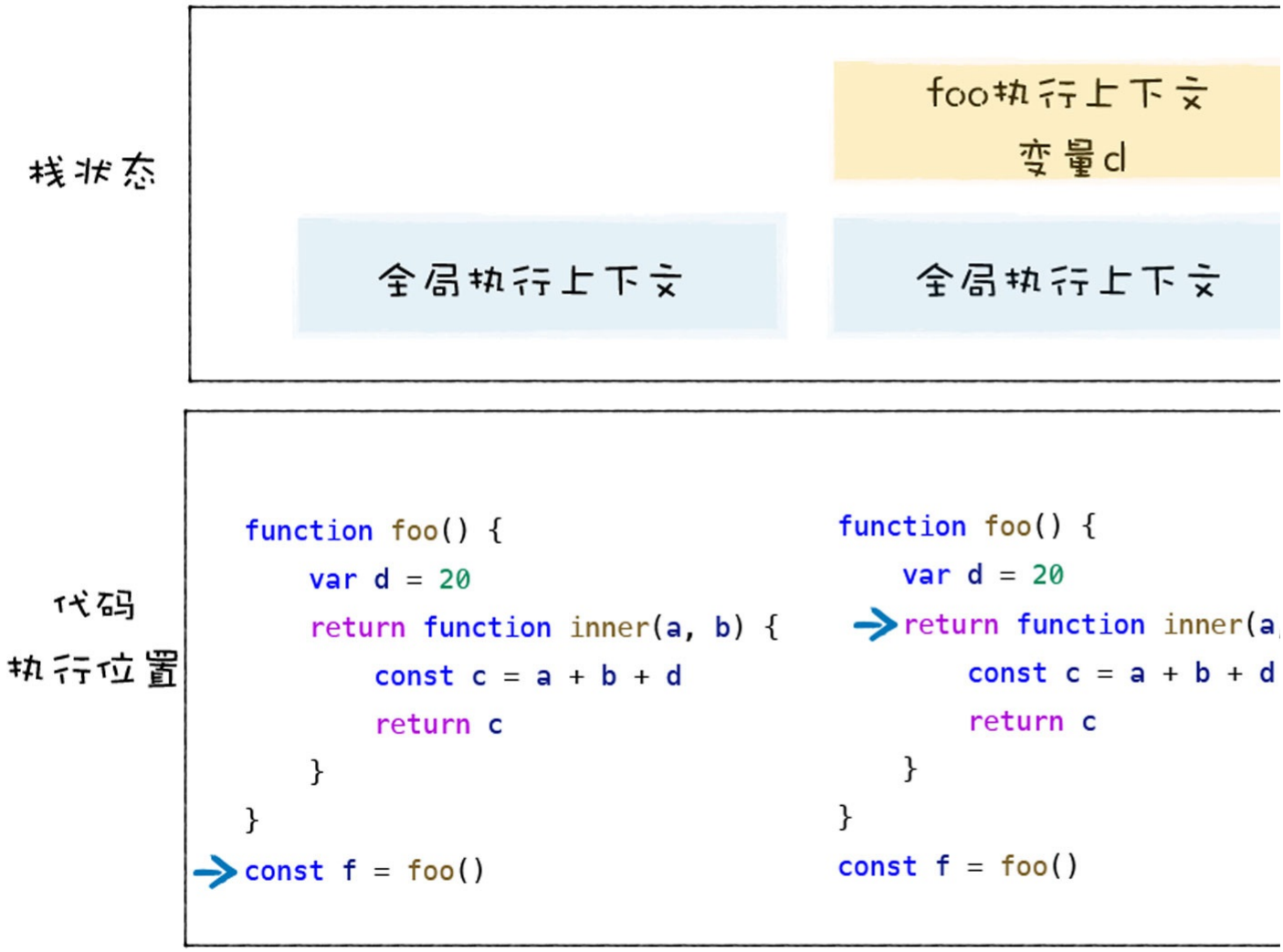
我们可以分析下上面这段代码的执行过程：

- 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；
- 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：



从上图可以看出来，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做预解析器。

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：

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

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：

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

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：

```
function foo() {
   var a = 0
}
```

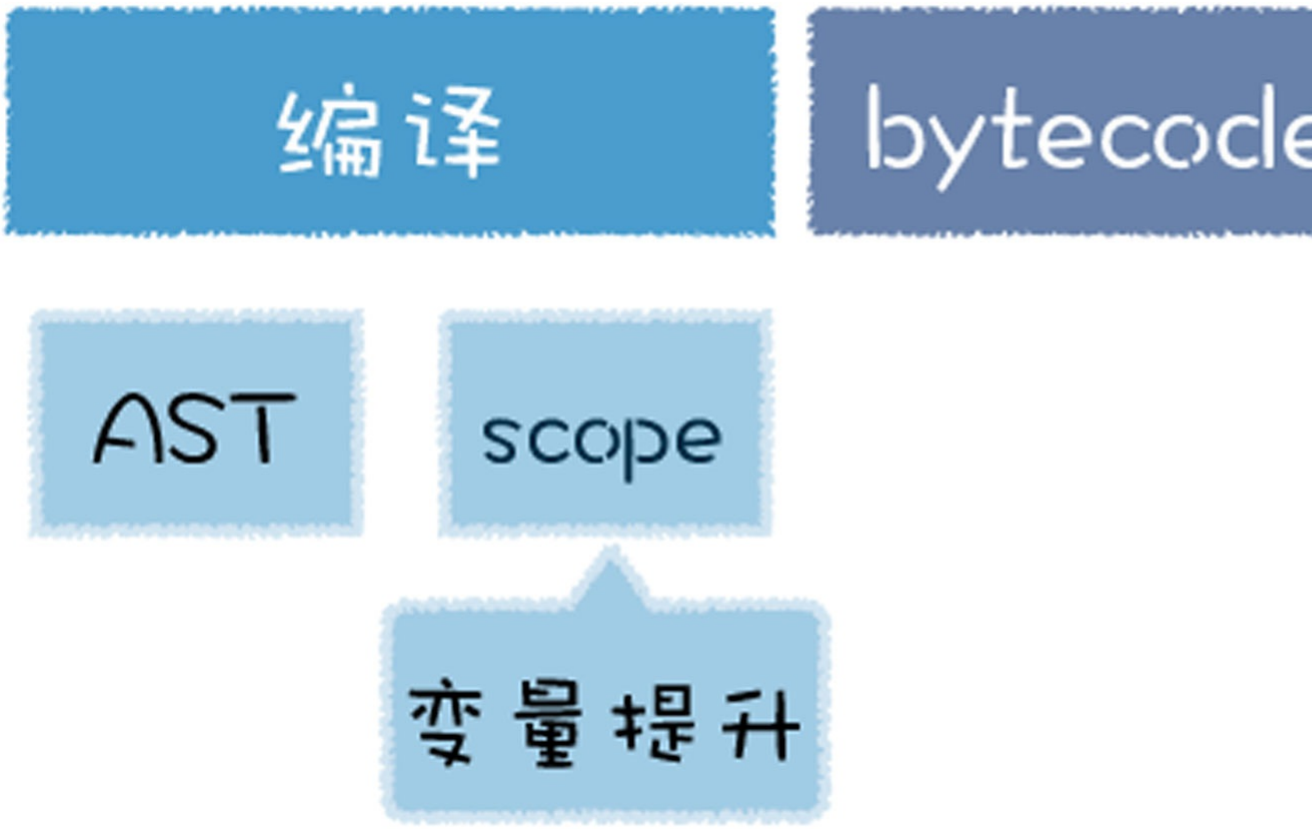
```
function foo() {
    var a = 0
    return function inner() {
        return a++
    }
}
```

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，如果你觉得这一讲的内容对你有所启发，也欢迎把它分享给你的朋友。

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。总的流程你可以参考下图：



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。

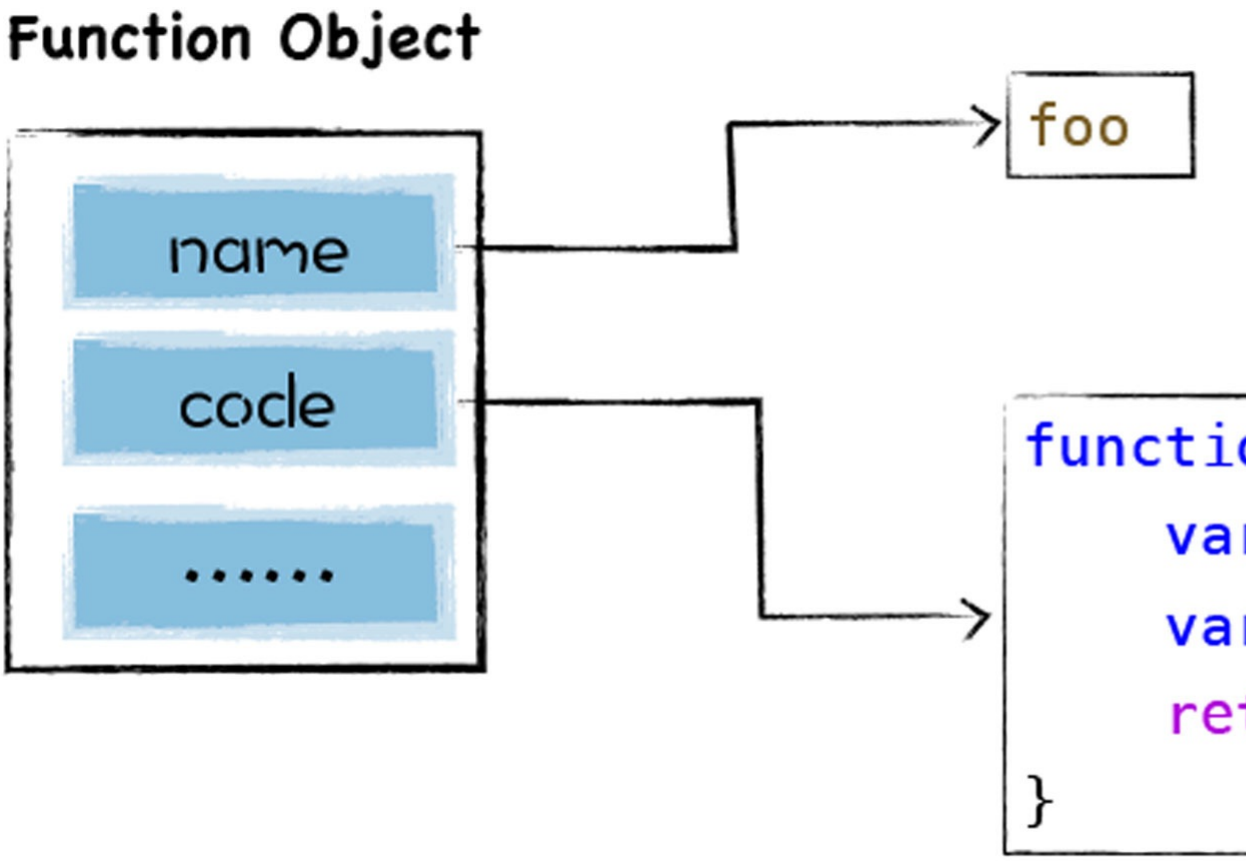
基于以上的原因，所有主流的JavaScript虚拟机都实现了**惰性解析**。所谓惰性解析是指解析器在解析的过程中，如果遇到函数声明，那么会跳过函数内部的代码，并不会为其生成AST和字节码，而仅仅生成顶层代码的AST和字节码。

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：

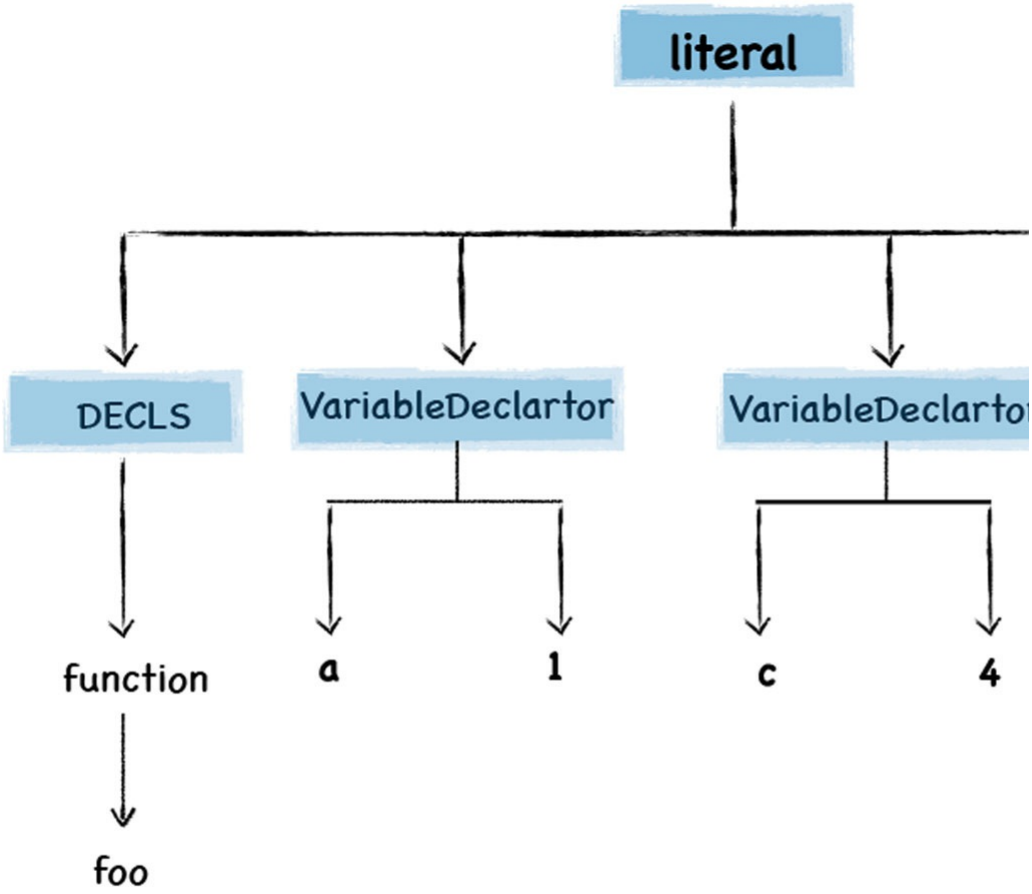
```
function foo(a,b) {
    var d = 100
    var f = 10
    return d + f + a + b;
}
var a = 1
var c = 4
foo(1, 5)
```

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：



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：

```
function foo() {
    function inner() {
    }
    inner()
}
```

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：

```
var d = 20
//inner函数的父函数，词法作用域
function foo() {
    var d = 55
    //foo的内部函数
    function inner() {
      return d+2
    }
    inner()
}
```

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丨作用域链：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：

```
function foo() {
    return function inner(a, b) {
        const c = a + b
        return c
    }
}
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：

```
function foo() {
    var d = 20
    return function inner(a, b) {
        const c = a + b + d
        return c
    }
}
const f = foo()
```

观察上面上面这段代码，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。

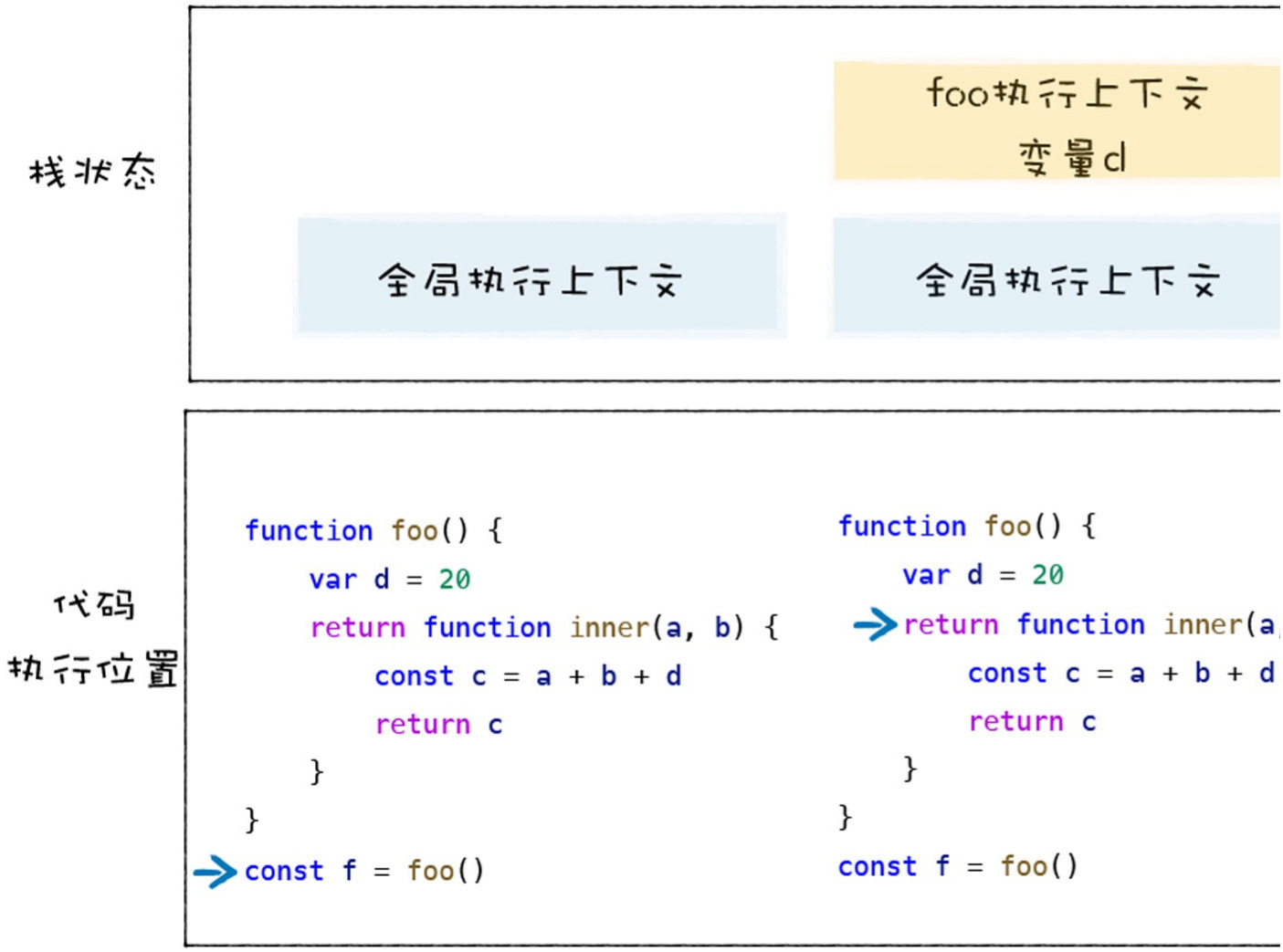
我们可以分析下上面这段代码的执行过程：

- 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；
- 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：



从上图可以看出来，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做预解析器。

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：

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

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：

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

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：

```
function foo() {
   var a = 0
}
```

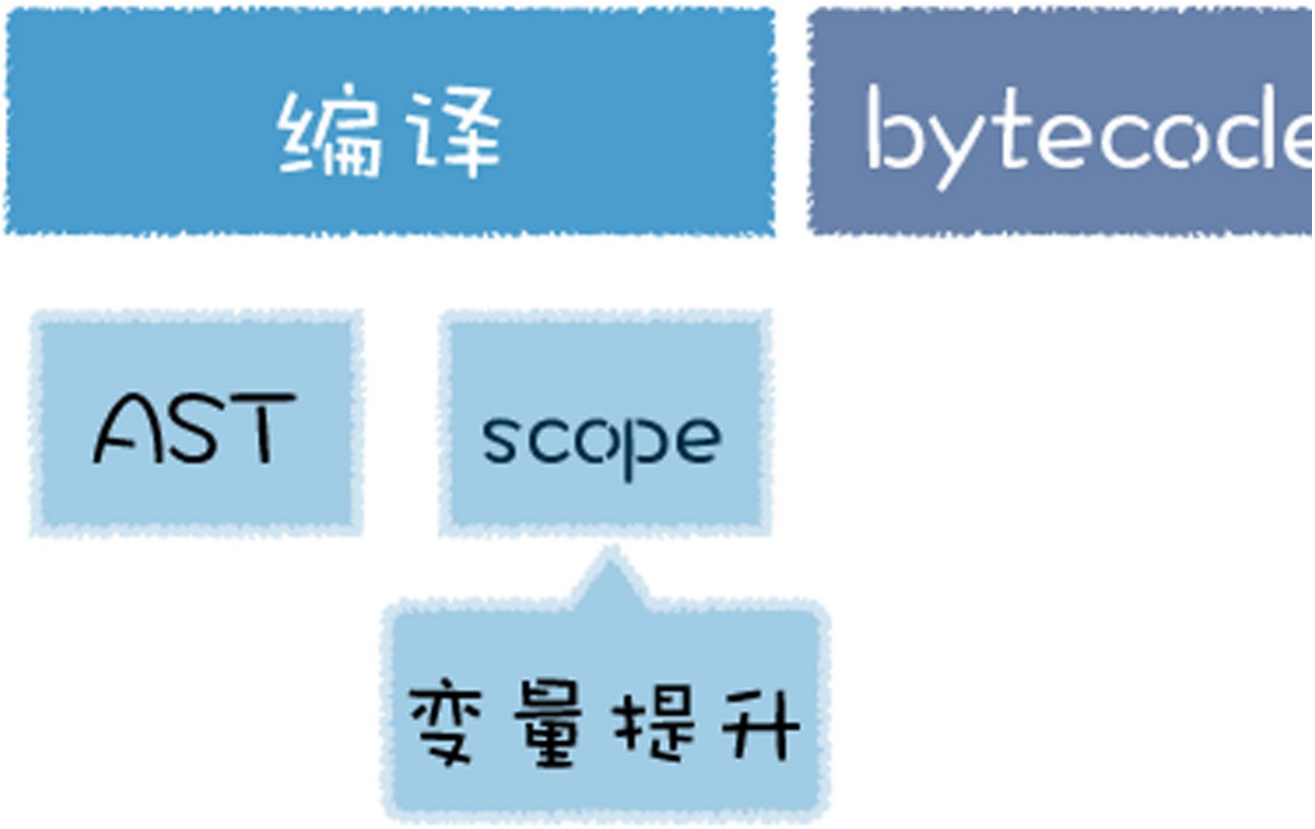
```
function foo() {
    var a = 0
    return function inner() {
        return a++
    }
}
```

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，如果你觉得这一讲的内容对你有所启发，也欢迎把它分享给你的朋友。

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。总的流程你可以参考下图：



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。

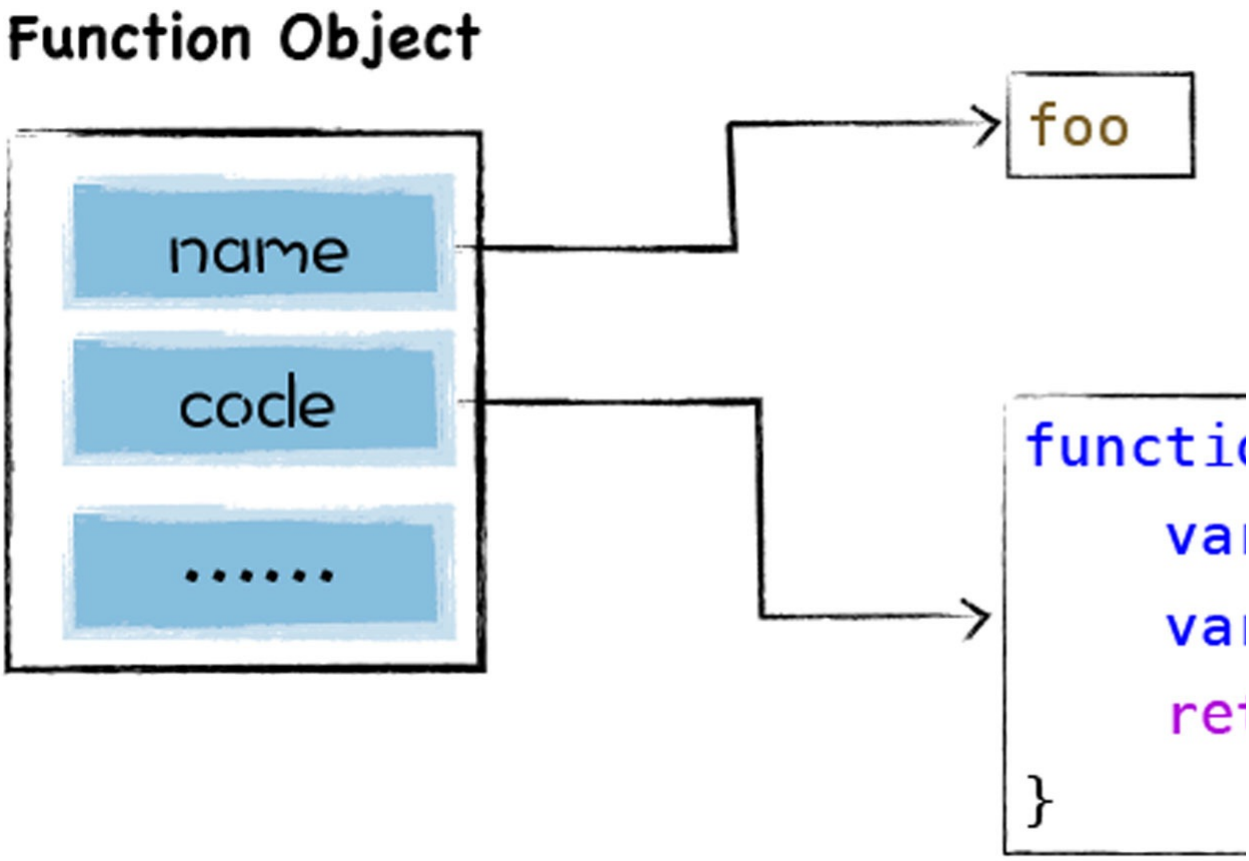
基于以上的原因，所有主流的JavaScript虚拟机都实现了**惰性解析**。所谓惰性解析是指解析器在解析的过程中，如果遇到函数声明，那么会跳过函数内部的代码，并不会为其生成AST和字节码，而仅仅生成顶层代码的AST和字节码。

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：

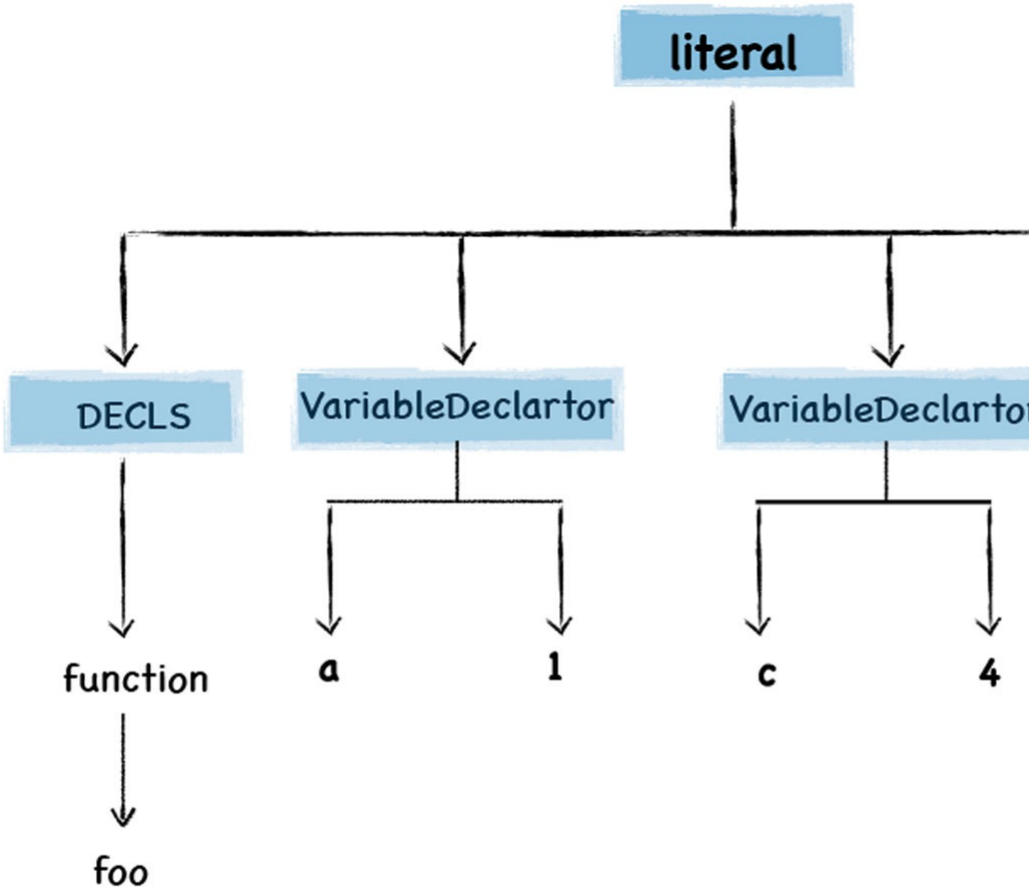
```
function foo(a,b) {
    var d = 100
    var f = 10
    return d + f + a + b;
}
var a = 1
var c = 4
foo(1, 5)
```

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：



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：

```
function foo() {
    function inner() {
    }
    inner()
}
```

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：

```
var d = 20
//inner函数的父函数，词法作用域
function foo() {
    var d = 55
    //foo的内部函数
    function inner() {
      return d+2
    }
    inner()
}
```

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丨作用域链：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：

```
function foo() {
    return function inner(a, b) {
        const c = a + b
        return c
    }
}
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：

```
function foo() {
    var d = 20
    return function inner(a, b) {
        const c = a + b + d
        return c
    }
}
const f = foo()
```

观察上面上面这段代码，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。

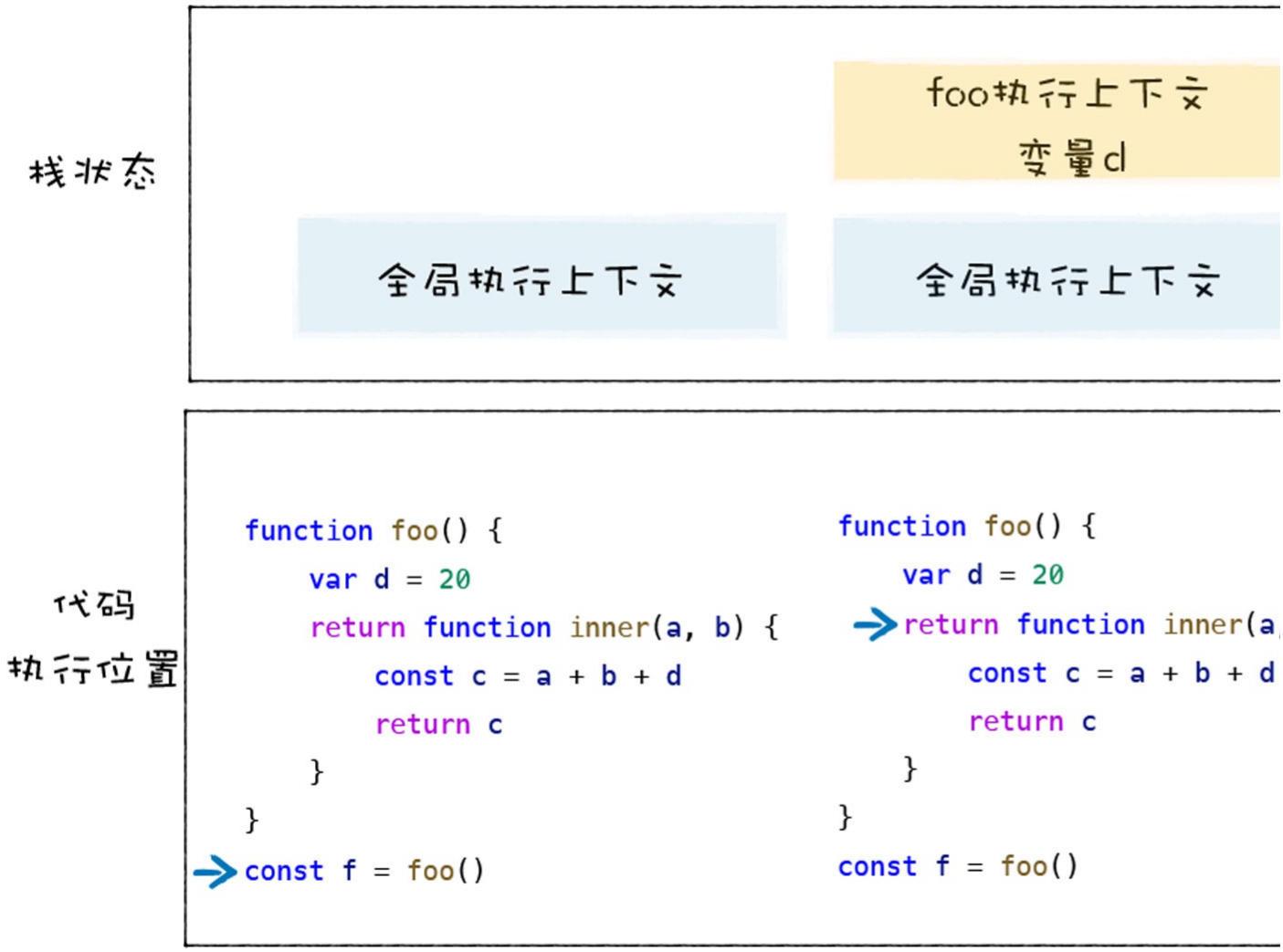
我们可以分析下上面这段代码的执行过程：

- 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；
- 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：



从上图可以看出来，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做预解析器。

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：

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

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：

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

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：

```
function foo() {
   var a = 0
}
```

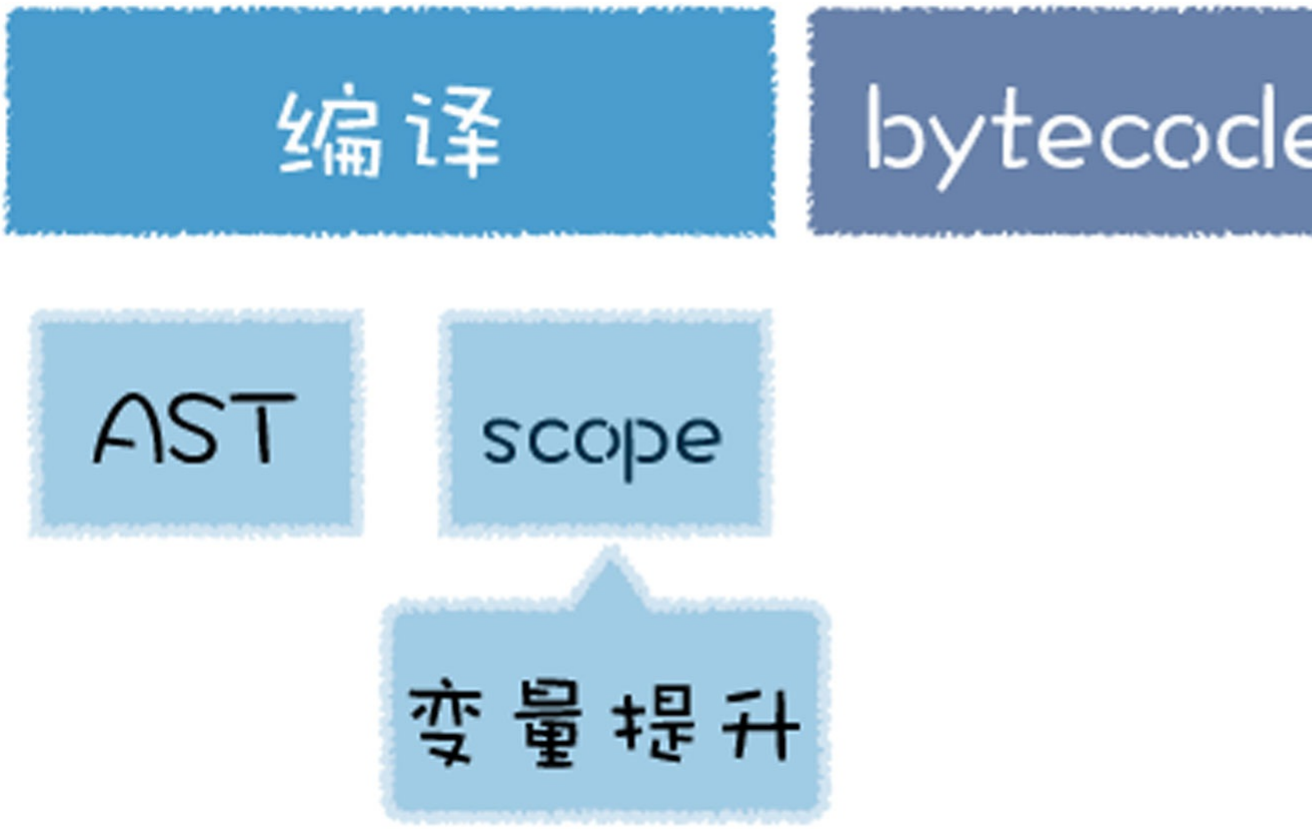
```
function foo() {
    var a = 0
    return function inner() {
        return a++
    }
}
```

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，如果你觉得这一讲的内容对你有所启发，也欢迎把它分享给你的朋友。

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。总的流程你可以参考下图：



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。

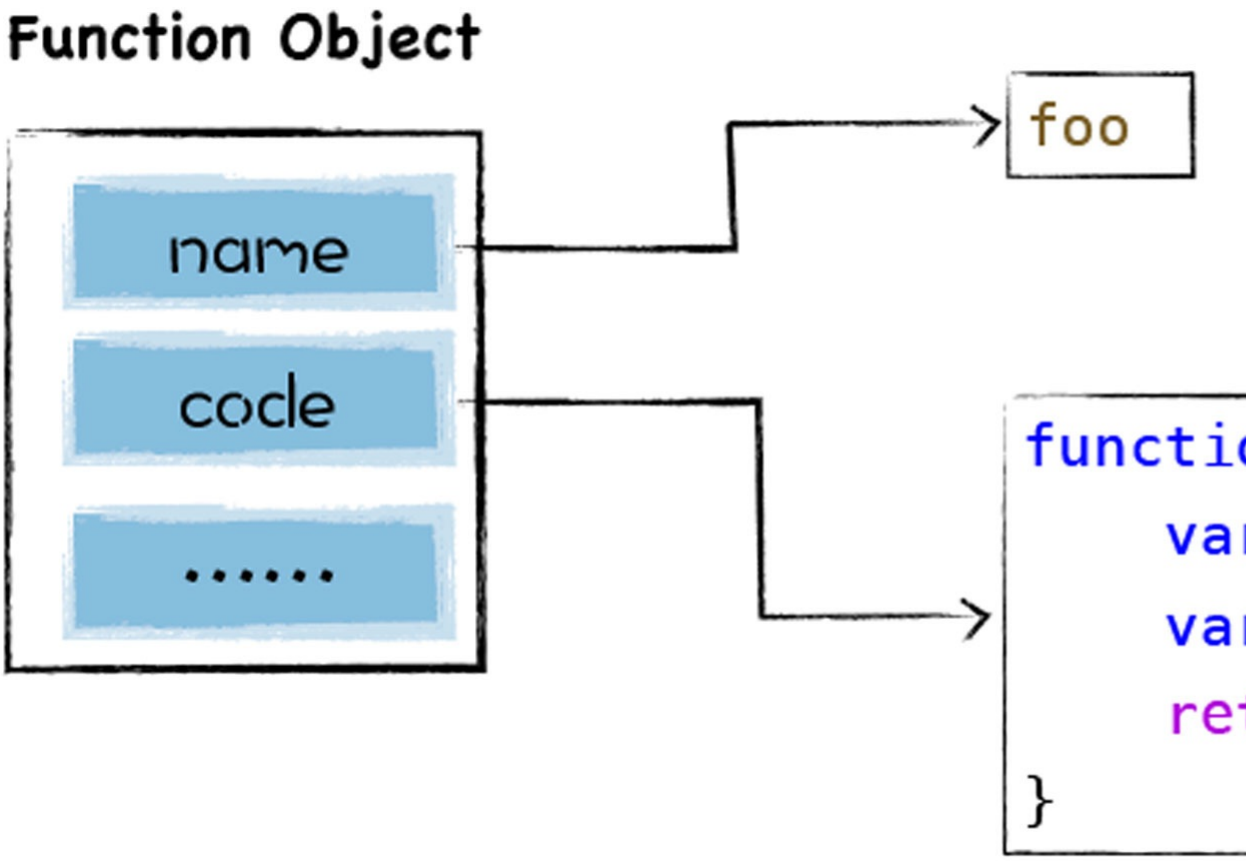
基于以上的原因，所有主流的JavaScript虚拟机都实现了**惰性解析**。所谓惰性解析是指解析器在解析的过程中，如果遇到函数声明，那么会跳过函数内部的代码，并不会为其生成AST和字节码，而仅仅生成顶层代码的AST和字节码。

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：

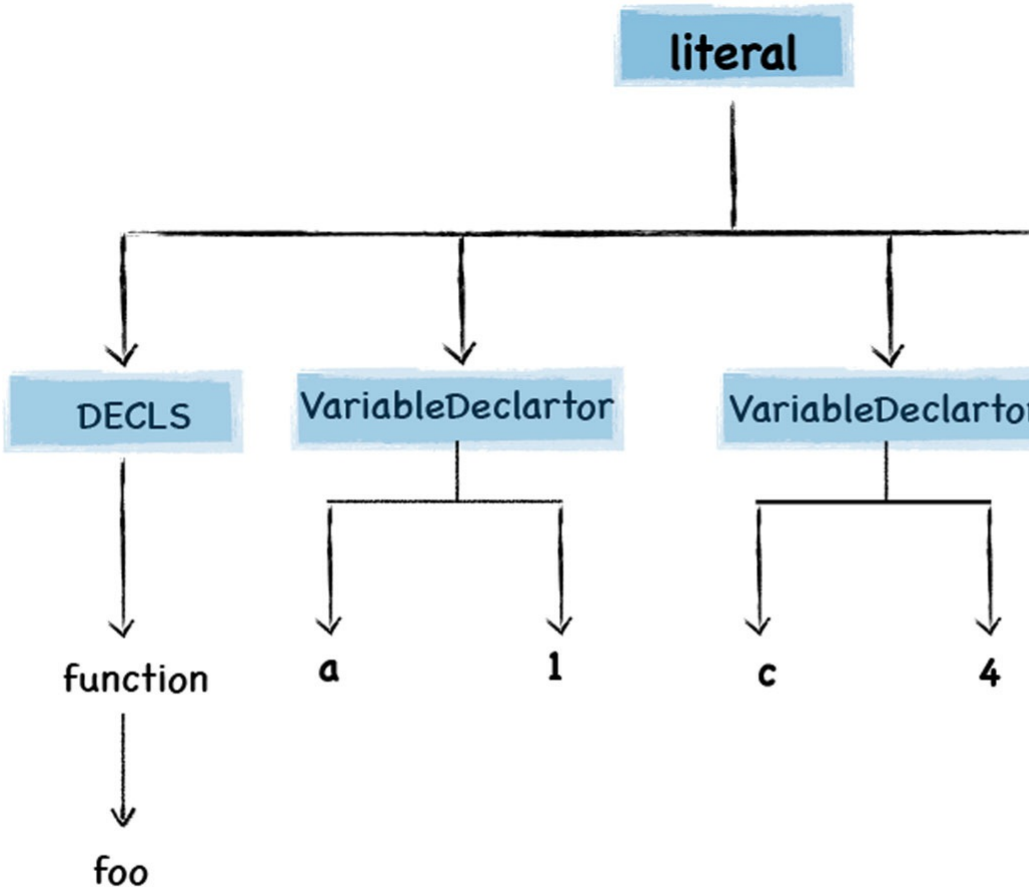
```
function foo(a,b) {
    var d = 100
    var f = 10
    return d + f + a + b;
}
var a = 1
var c = 4
foo(1, 5)
```

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：



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：

```
function foo() {
    function inner() {
    }
    inner()
}
```

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：

```
var d = 20
//inner函数的父函数，词法作用域
function foo() {
    var d = 55
    //foo的内部函数
    function inner() {
      return d+2
    }
    inner()
}
```

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丨作用域链：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：

```
function foo() {
    return function inner(a, b) {
        const c = a + b
        return c
    }
}
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：

```
function foo() {
    var d = 20
    return function inner(a, b) {
        const c = a + b + d
        return c
    }
}
const f = foo()
```

观察上面上面这段代码，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。

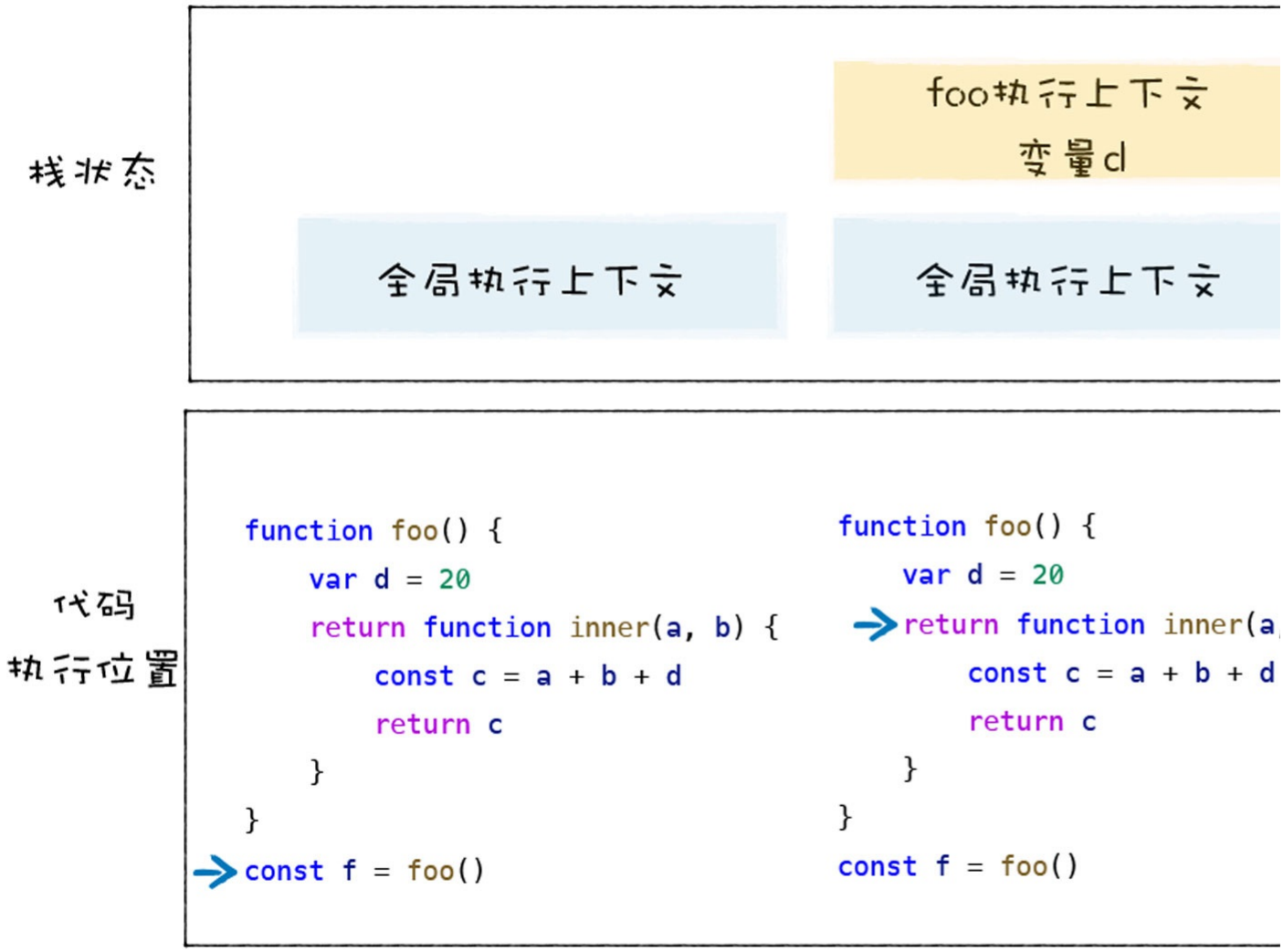
我们可以分析下上面这段代码的执行过程：

- 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；
- 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：



从上图可以看出来，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做预解析器。

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：

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

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：

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

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：

```
function foo() {
   var a = 0
}
```

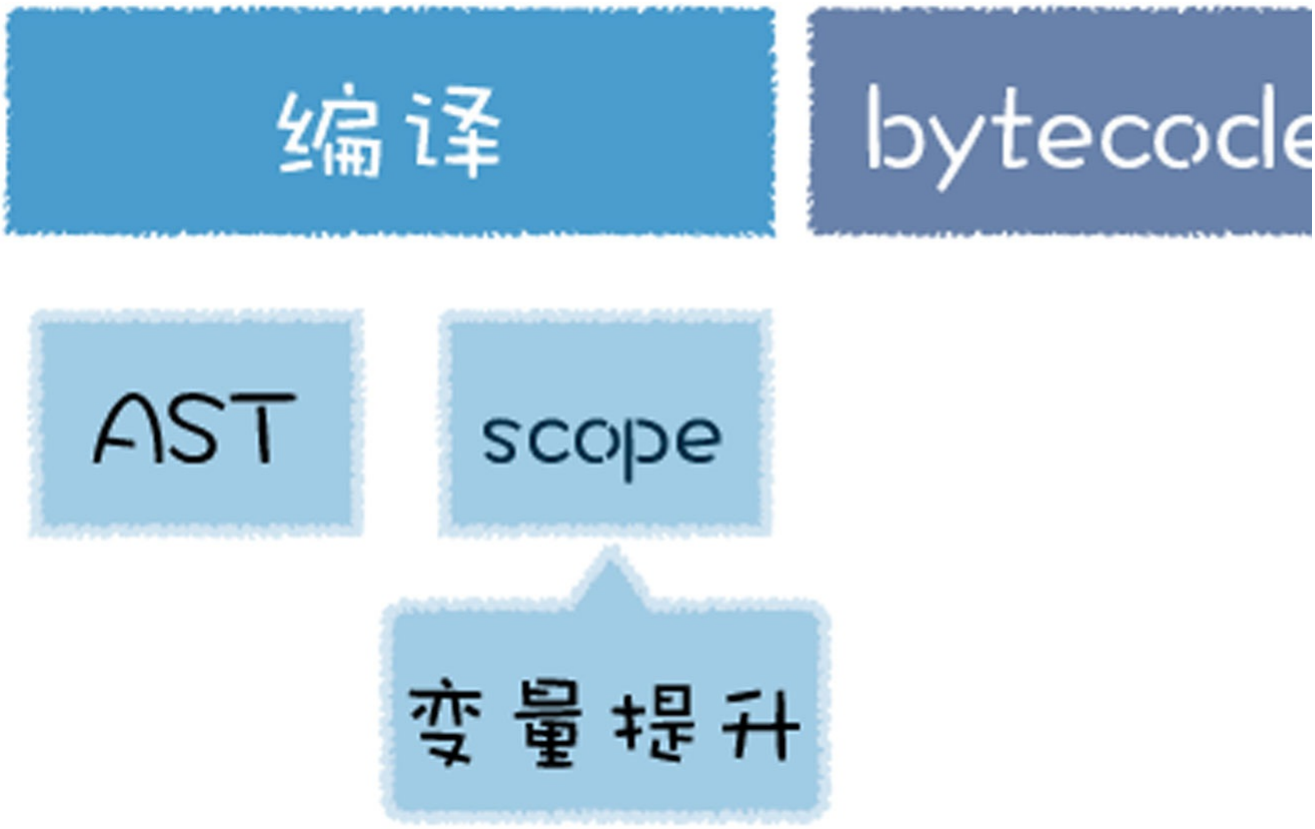
```
function foo() {
    var a = 0
    return function inner() {
        return a++
    }
}
```

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，如果你觉得这一讲的内容对你有所启发，也欢迎把它分享给你的朋友。

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。总的流程你可以参考下图：



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。

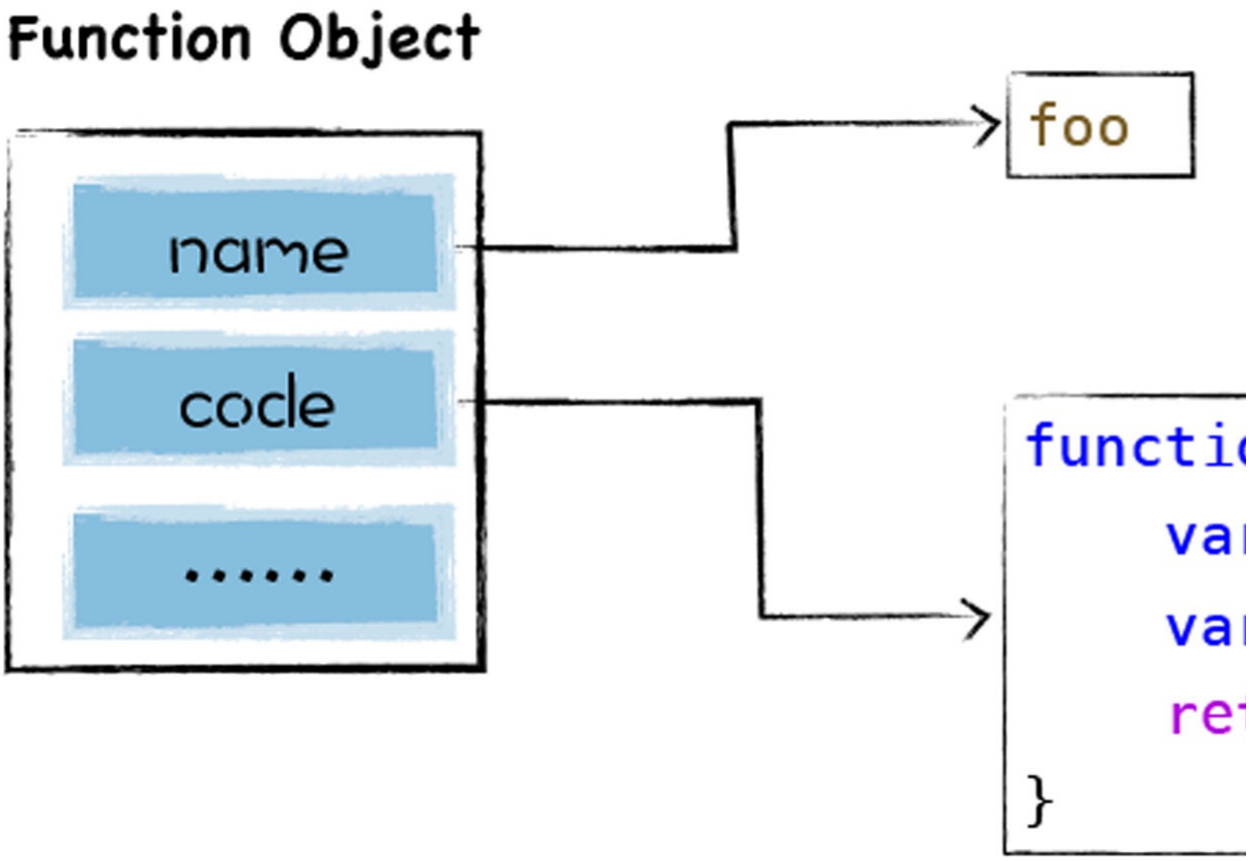
基于以上的原因，所有主流的JavaScript虚拟机都实现了**惰性解析**。所谓惰性解析是指解析器在解析的过程中，如果遇到函数声明，那么会跳过函数内部的代码，并不会为其生成AST和字节码，而仅仅生成顶层代码的AST和字节码。

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：

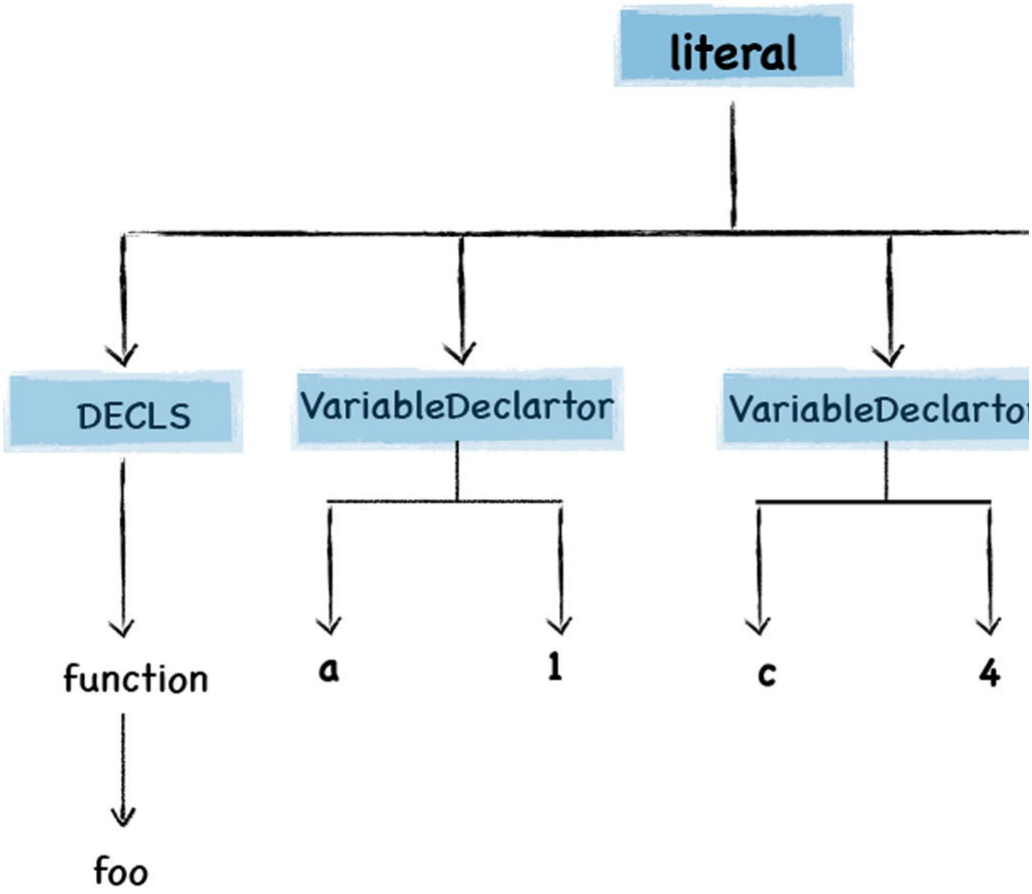
```
function foo(a,b) {
    var d = 100
    var f = 10
    return d + f + a + b;
}
var a = 1
var c = 4
foo(1, 5)
```

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：



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：

```
function foo() {
    function inner() {
    }
    inner()
}
```

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：

```
var d = 20
//inner函数的父函数，词法作用域
function foo() {
    var d = 55
    //foo的内部函数
    function inner() {
      return d+2
    }
    inner()
}
```

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丨作用域链：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：

```
function foo() {
    return function inner(a, b) {
        const c = a + b
        return c
    }
}
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：

```
function foo() {
    var d = 20
    return function inner(a, b) {
        const c = a + b + d
        return c
    }
}
const f = foo()
```

观察上面上面这段代码，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。

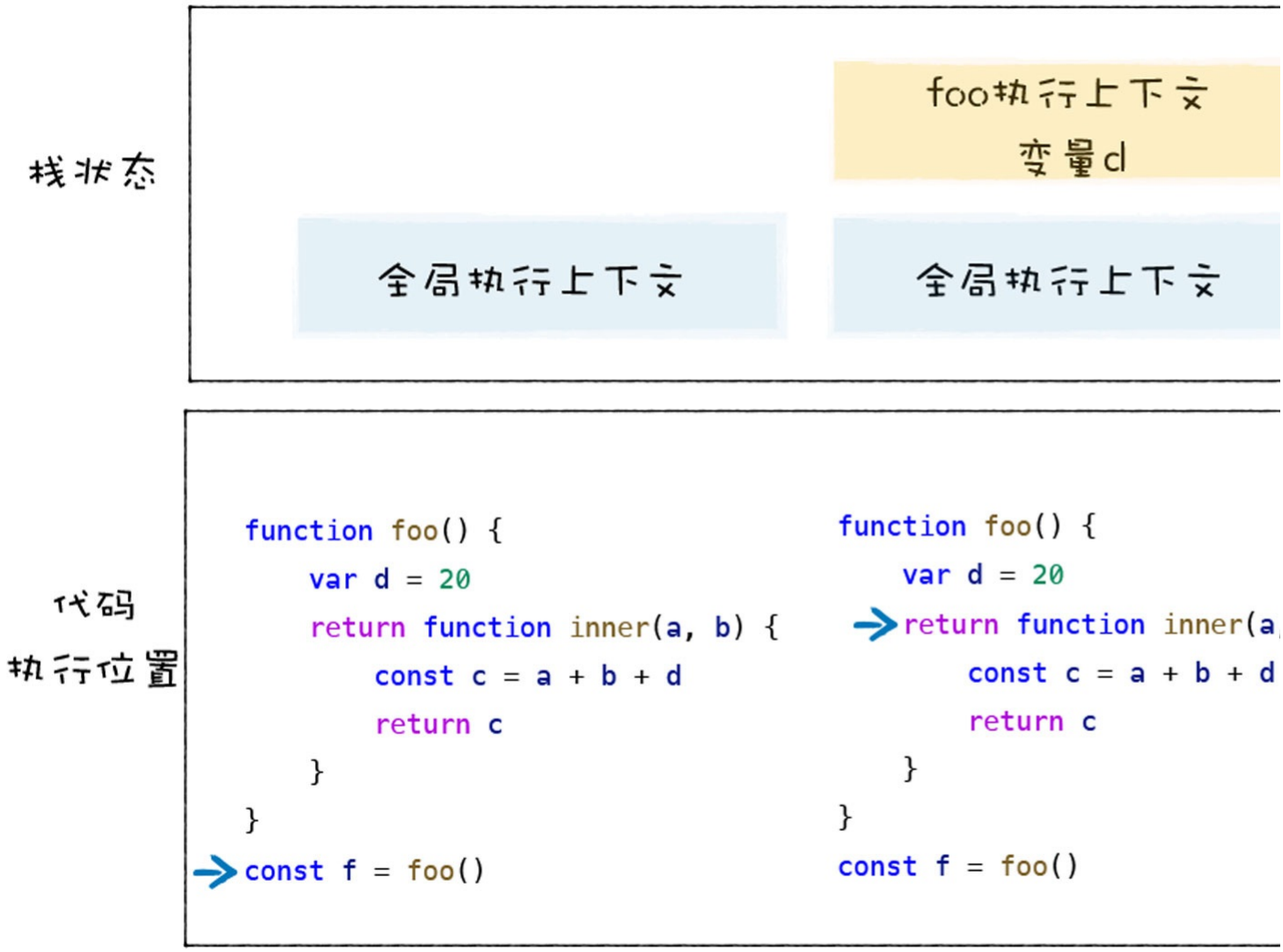
我们可以分析下上面这段代码的执行过程：

- 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；
- 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：



从上图可以看出来，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做预解析器。

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：

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

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：

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

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：

```
function foo() {
   var a = 0
}
```

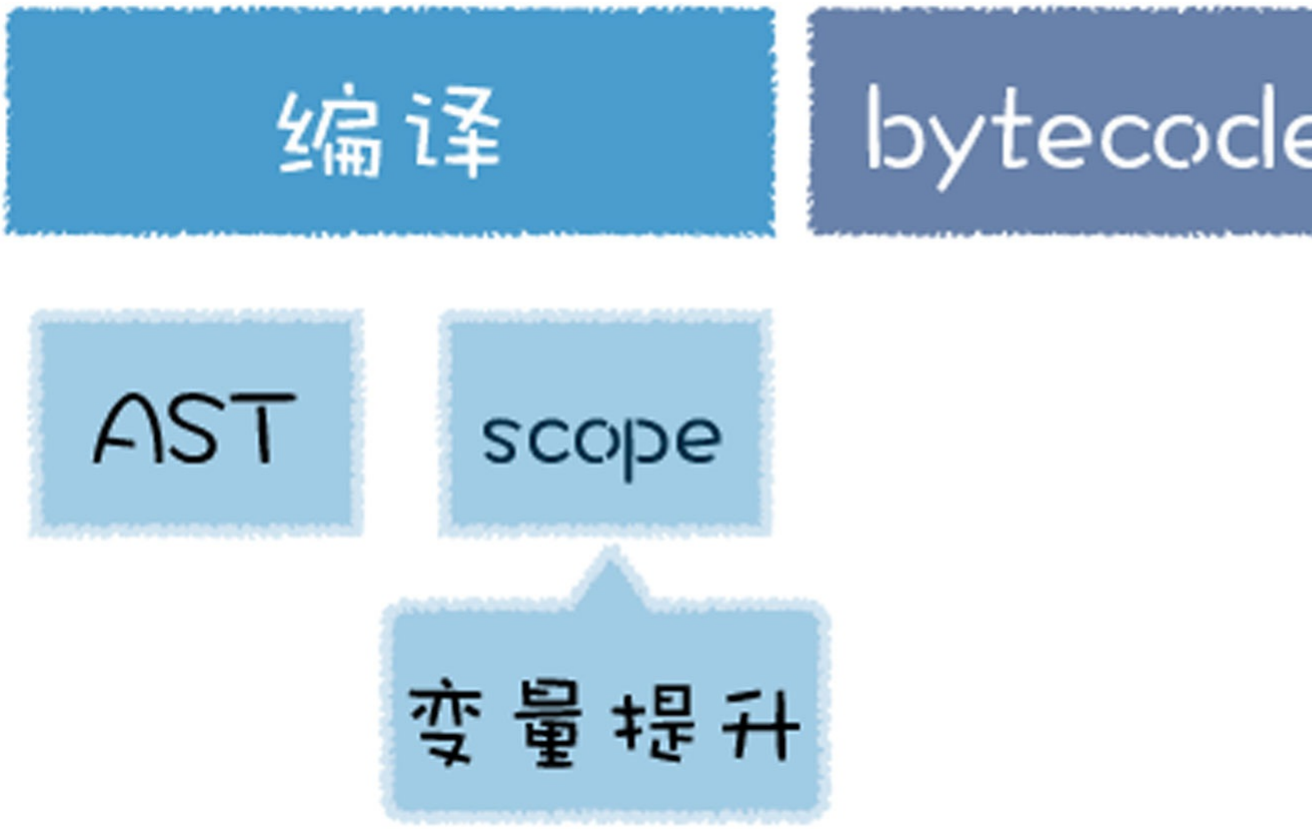
```
function foo() {
    var a = 0
    return function inner() {
        return a++
    }
}
```

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，如果你觉得这一讲的内容对你有所启发，也欢迎把它分享给你的朋友。

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。总的流程你可以参考下图：



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。

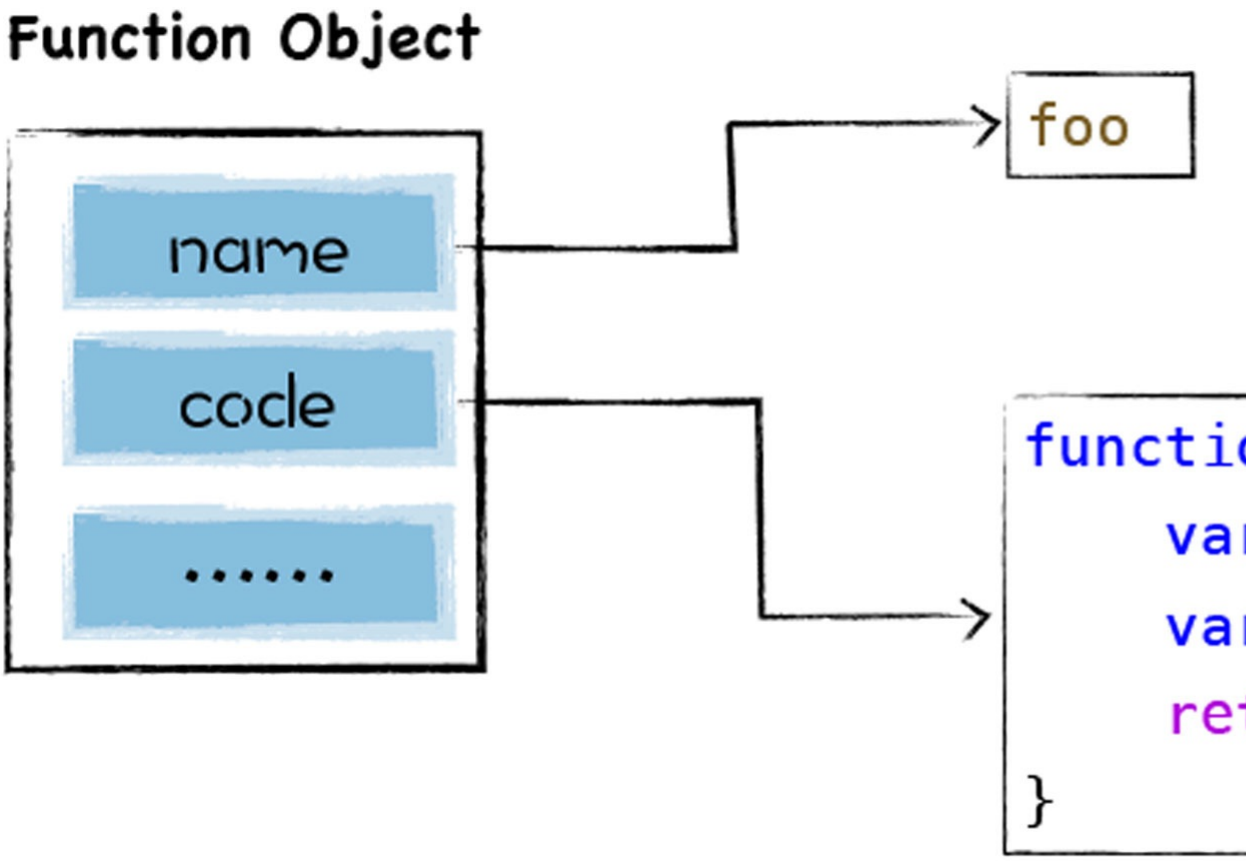
基于以上的原因，所有主流的JavaScript虚拟机都实现了**惰性解析**。所谓惰性解析是指解析器在解析的过程中，如果遇到函数声明，那么会跳过函数内部的代码，并不会为其生成AST和字节码，而仅仅生成顶层代码的AST和字节码。

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：

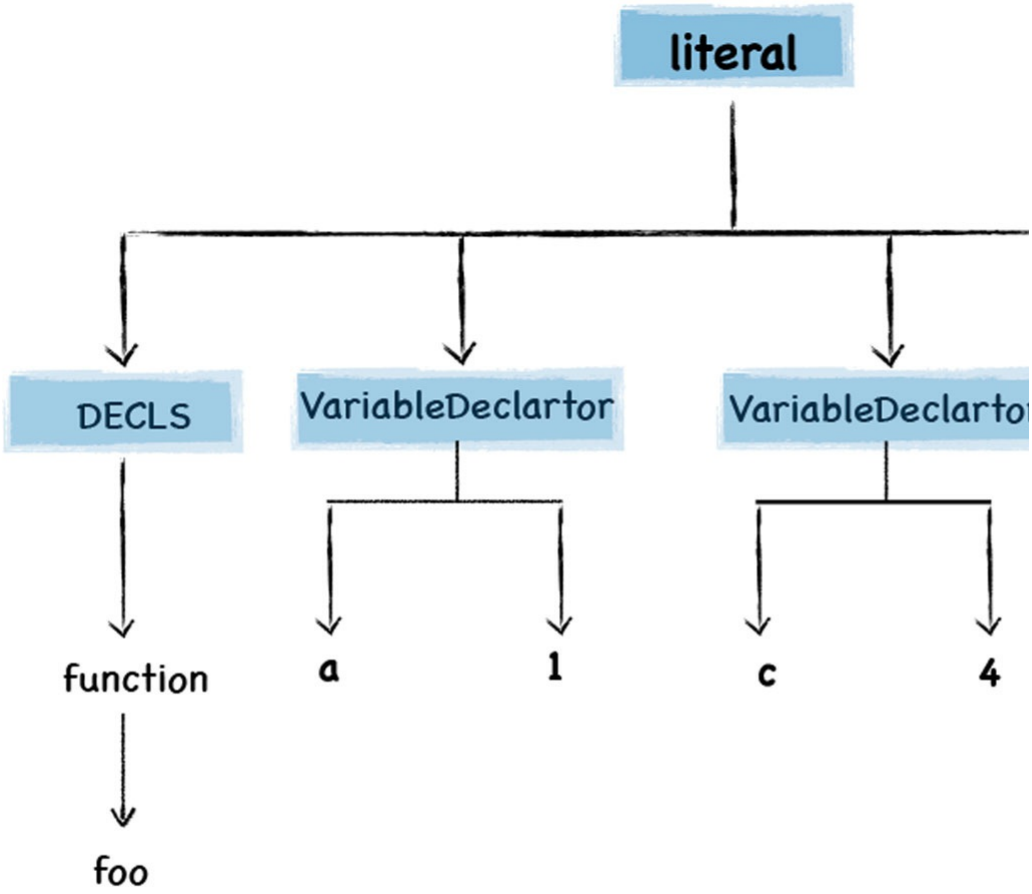
```
function foo(a,b) {
    var d = 100
    var f = 10
    return d + f + a + b;
}
var a = 1
var c = 4
foo(1, 5)
```

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：



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：

```
function foo() {
    function inner() {
    }
    inner()
}
```

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：

```
var d = 20
//inner函数的父函数，词法作用域
function foo() {
    var d = 55
    //foo的内部函数
    function inner() {
      return d+2
    }
    inner()
}
```

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丨作用域链：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：

```
function foo() {
    return function inner(a, b) {
        const c = a + b
        return c
    }
}
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：

```
function foo() {
    var d = 20
    return function inner(a, b) {
        const c = a + b + d
        return c
    }
}
const f = foo()
```

观察上面上面这段代码，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。

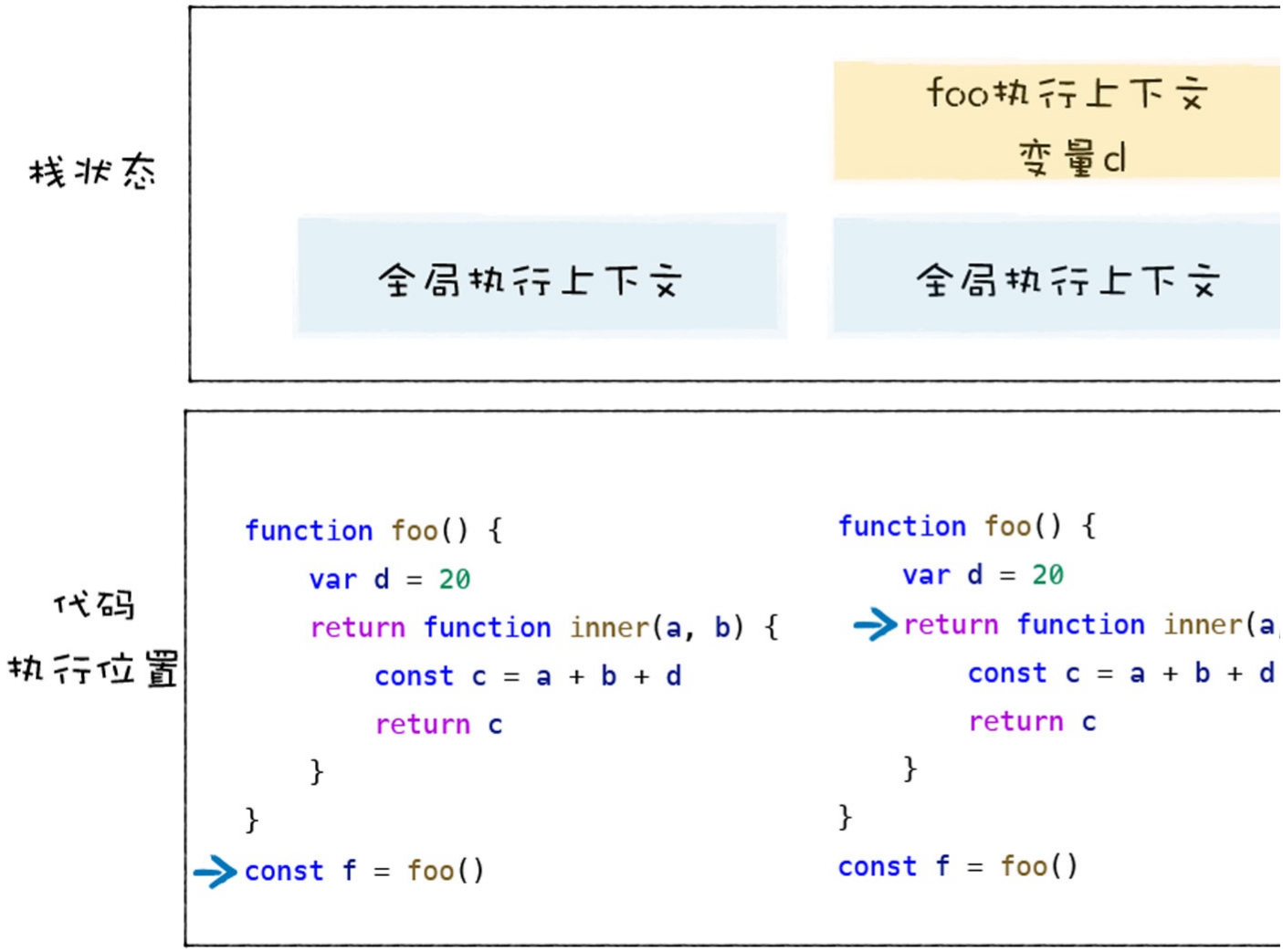
我们可以分析下上面这段代码的执行过程：

- 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；
- 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：



从上图可以看出来，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做预解析器。

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：

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

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：

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

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：

```
function foo() {
   var a = 0
}
```

```
function foo() {
    var a = 0
    return function inner() {
        return a++
    }
}
```

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，如果你觉得这一讲的内容对你有所启发，也欢迎把它分享给你的朋友。