09-块级作用域：var缺陷以及为什么要引入let和const?

在前面《07 | 变量提升：JavaScript代码是按顺序执行的吗？》这篇文章中，我们已经讲解了JavaScript中变量提升的相关内容，**正是由于JavaScript存在变量提升这种特性，从而导致了很多与直觉不符的代码，这也是JavaScript的一个重要设计缺陷**。

虽然ECMAScript6（以下简称ES6）已经通过引入块级作用域并配合let、const关键字，来避开了这种设计缺陷，但是由于JavaScript需要保持向下兼容，所以变量提升在相当长一段时间内还会继续存在。这也加大了你理解概念的难度，因为既要理解新的机制，又要理解变量提升这套机制，关键这两套机制还是同时运行在“一套”系统中的。

但如果抛开JavaScript的底层去理解这些，那么你大概率会很难深入理解其概念。俗话说，“断病要断因，治病要治根”，所以为了便于你更好地理解和学习，今天我们这篇文章会先“**探病因**”——分析为什么在JavaScript中会存在变量提升，以及变量提升所带来的问题；然后再来“**开药方**”——介绍如何通过**块级作用域并配合let和const关键字**来修复这种缺陷。

作用域（scope）

为什么JavaScript中会存在变量提升这个特性，而其他语言似乎都没有这个特性呢？要讲清楚这个问题，我们就得先从作用域讲起。

作用域是指在程序中定义变量的区域，该位置决定了变量的生命周期。通俗地理解，作用域就是变量与函数的可访问范围，即作用域控制着变量和函数的可见性和生命周期。

在ES6之前，ES的作用域只有两种：全局作用域和函数作用域。

- **全局作用域**中的对象在代码中的任何地方都能访问，其生命周期伴随着页面的生命周期。
- **函数作用域**就是在函数内部定义的变量或者函数，并且定义的变量或者函数只能在函数内部被访问。函数执行结束之后，函数内部定义的变量会被销毁。

在ES6之前，JavaScript只支持这两种作用域，相较而言，其他语言则都普遍支持**块级作用域**。块级作用域就是使用一对大括号包裹的一段代码，比如函数、判断语句、循环语句，甚至单独的一个{}都可以被看作是一个块级作用域。

为了更好地理解块级作用域，你可以参考下面的一些示例代码：

```
//if块
if(1){}

//while块
while(1){}

//函数块
function foo(){

//for循环块
for(let i = 0; i<100; i++){}

//单独一个块
```

```
{}
```

简单来讲，如果一种语言支持块级作用域，那么其代码块内部定义的变量在代码块外部是访问不到的，并且等该代码块中的代码执行完成之后，代码块中定义的变量会被销毁。你可以看下面这段C代码：

```
char* myname = "极客时间";
void showName() {
  printf("%s \n",myname);
  if(0){
    char* myname = "极客邦";
   }
}

int main(){
   showName();
   return 0;
}
```

上面这段C代码执行后，最终打印出来的是上面全局变量myname的值，之所以这样，是因为C语言是支持块级作用域的，所以if块里面定义的变量是不能被if块外面的语句访问到的。

和Java、C/C++不同，**ES6之前是不支持块级作用域的**，因为当初设计这门语言的时候，并没有想到JavaScript会火起来，所以只是按照最简单的方式来设计。没有了块级作用域，再把作用域内部的变量统一提升无疑是最快速、最简单的设计，不过这也直接导致了函数中的变量无论是在哪里声明的，在编译阶段都会被提取到执行上下文的变量环境中，所以这些变量在整个函数体内部的任何地方都是能被访问的，这也就是JavaScript中的变量提升。

变量提升所带来的问题

由于变量提升作用，使用JavaScript来编写和其他语言相同逻辑的代码，都有可能会导致不一样的执行结果。那为什么会出现这种情况呢？主要有以下两种原因。

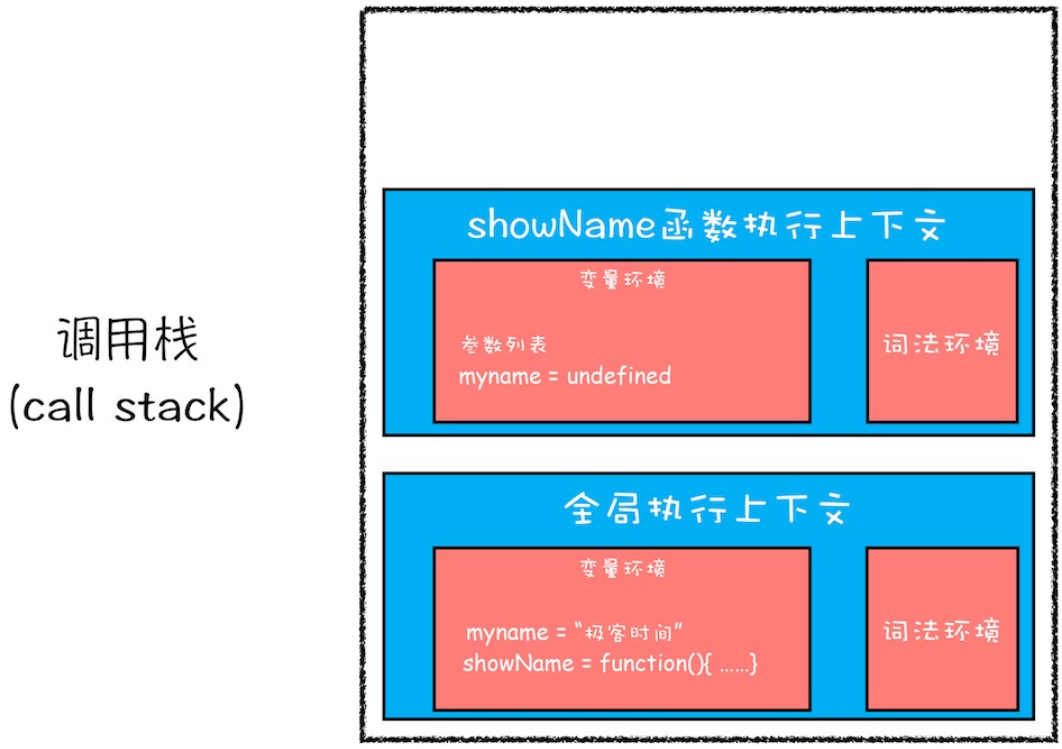
1. 变量容易在不被察觉的情况下被覆盖掉

比如我们重新使用JavaScript来实现上面那段C代码，实现后的JavaScript代码如下：

```
var myname = "极客时间"
function showName(){
  console.log(myname);
  if(0){
   var myname = "极客邦"
  }
  console.log(myname);
}
showName()
```

执行上面这段代码，打印出来的是undefined，而并没有像前面C代码那样打印出来“极客时间”的字符串。为什么输出的内容是undefined呢？我们再来分析一下。

首先当刚执行到showName函数调用时，执行上下文和调用栈的状态是怎样的？具体分析过程你可以回顾《08 | 调用栈：为什么JavaScript代码会出现栈溢出？》这篇文章的分析过程，这里我就直接展示出来了，最终的调用栈状态如下图所示：



开始执行showName函数时的调用栈

showName函数的执行上下文创建后，JavaScript引擎便开始执行showName函数内部的代码了。首先执行的是：

```
console.log(myname);
```

执行这段代码需要使用变量myname，结合上面的调用栈状态图，你可以看到这里有两个myname变量：一个在全局执行上下文中，其值是“极客时间”；另外一个在showName函数的执行上下文中，其值是undefined。那么到底该使用哪个呢？

相信做过JavaScript开发的同学都能轻松回答出来答案：“当然是**先使用函数执行上下文里面的变量**啦！”的确是这样，这是因为在函数执行过程中，JavaScript会优先从当前的执行上下文中查找变量，由于变量提升，当前的执行上下文中就包含了变量myname，而值是undefined，所以获取到的myname的值就是undefined。

这输出的结果和其他大部分支持块级作用域的语言都不一样，比如上面C语言输出的就是全局变量，所以这会很容易造成误解，特别是在你会一些其他语言的基础之上，再来学习JavaScript，你会觉得这种结果很不自然。

2. 本应销毁的变量没有被销毁

接下来我们再来看下面这段让人误解更大的代码：

```
function foo(){
  for (var i = 0; i < 7; i++) {
  }
  console.log(i);
}
foo()
```

如果你使用C语言或者其他的大部分语言实现类似代码，在for循环结束之后，i就已经被销毁了，但是在JavaScript代码中，i的值并未被销毁，所以最后打印出来的是7。

这同样也是由变量提升而导致的，在创建执行上下文阶段，变量i就已经被提升了，所以当for循环结束之后，变量i并没有被销毁。

这依旧和其他支持块级作用域的语言表现是不一致的，所以必然会给一些人造成误解。

ES6是如何解决变量提升带来的缺陷

上面我们介绍了变量提升而带来的一系列问题，为了解决这些问题，**ES6引入了let和const关键字**，从而使JavaScript也能像其他语言一样拥有了块级作用域。

关于let和const的用法，你可以参考下面代码：

```
let x = 5
const y = 6
x = 7
y = 9 //报错，const声明的变量不可以修改
```

从这段代码你可以看出来，两者之间的区别是，使用let关键字声明的变量是可以被改变的，而使用const声明的变量其值是不可以被改变的。但不管怎样，两者都可以生成块级作用域，为了简单起见，在下面的代码中，我统一使用let关键字来演示。

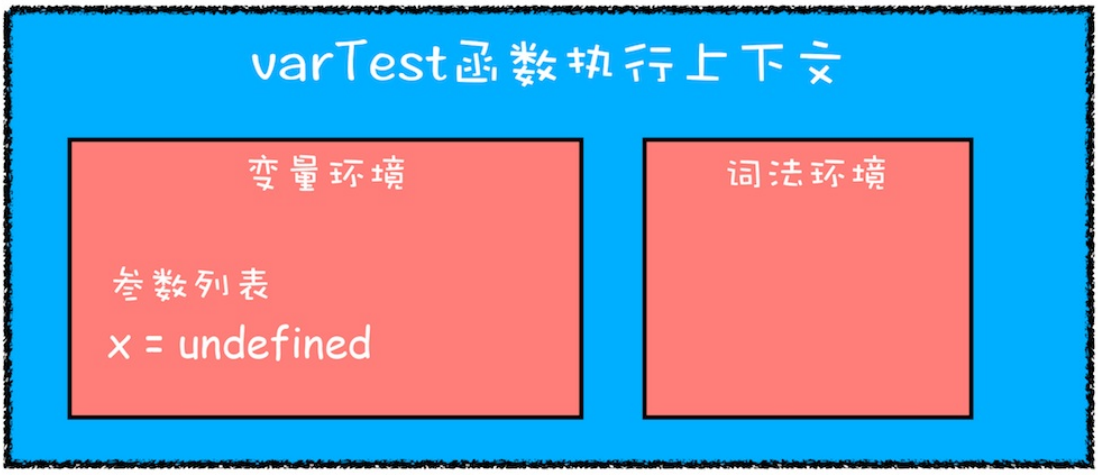
那么接下来，我们就通过实际的例子来分析下，ES6是如何通过块级作用域来解决上面的问题的？

你可以先参考下面这段存在变量提升的代码：

```
function varTest() {
  var x = 1;
  if (true) {
    var x = 2;  // 同样的变量!
    console.log(x);  // 2
  }
```

```
  console.log(x);  // 2
}
```

在这段代码中，有两个地方都定义了变量x，第一个地方在函数块的顶部，第二个地方在if块的内部，由于var的作用范围是整个函数，所以在编译阶段，会生成如下的执行上下文：



varTest函数的执行上下文

从执行上下文的变量环境中可以看出，最终只生成了一个变量x，函数体内所有对x的赋值操作都会直接改变变量环境中的x值。

所以上述代码最后通过console.log(x)输出的是2，而对于相同逻辑的代码，其他语言最后一步输出的值应该是1，因为在if块里面的声明不应该影响到块外面的变量。

既然支持块级作用域和不支持块级作用域的代码执行逻辑是不一样的，那么接下来我们就来改造上面的代码，让其支持块级作用域。

这个改造过程其实很简单，只需要把var关键字替换为let关键字，改造后的代码如下：

```
function letTest() {
  let x = 1;
  if (true) {
    let x = 2;  // 不同的变量
    console.log(x);  // 2
  }
  console.log(x);  // 1
}
```

执行这段代码，其输出结果就和我们的预期是一致的。这是因为let关键字是支持块级作用域的，所以在编译阶段，JavaScript引擎并不会把if块中通过let声明的变量存放到变量环境中，这也就意味着在if块通过let声明的关键字，并不会提升到全函数可见。所以在if块之内打印出来的值是2，跳出语块之后，打印出来的值就是1了。这种就非常**符合我们的编程习惯了：作用块内声明的变量不影响块外面的变量**。

JavaScript是如何支持块级作用域的

现在你知道了ES可以通过使用let或者const关键字来实现块级作用域，不过你是否有过这样的疑问：“在同一段代码中，ES6是如何做到既要支持变量提升的特性，又要支持块级作用域的呢？”

那么接下来，我们就要**站在执行上下文的角度**来揭开答案。

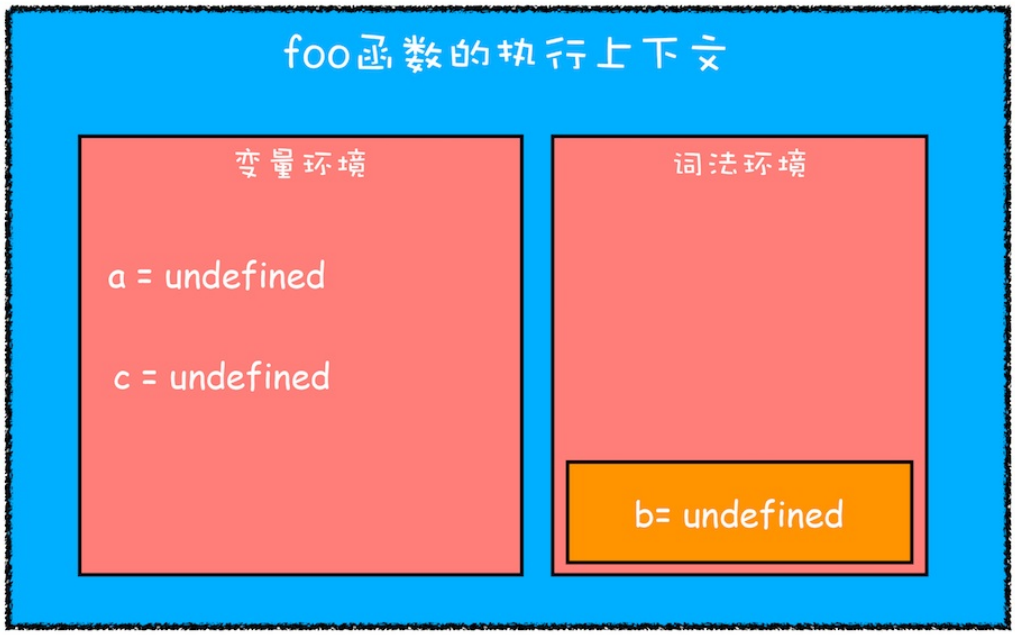
你已经知道JavaScript引擎是通过变量环境实现函数级作用域的，那么ES6又是如何在函数级作用域的基础之上，实现对块级作用域的支持呢？你可以先看下面这段代码：

```
function foo(){
    var a = 1
    let b = 2
    {
      let b = 3
      var c = 4
      let d = 5
      console.log(a)
      console.log(b)
    }
    console.log(b)
    console.log(c)
    console.log(d)
}
foo()
```

当执行上面这段代码的时候，JavaScript引擎会先对其进行编译并创建执行上下文，然后再按照顺序执行代码，关于如何创建执行上下文我们在前面的文章中已经分析过了，但是现在的情况有点不一样，我们引入了let关键字，let关键字会创建块级作用域，那么let关键字是如何影响执行上下文的呢？

接下来我们就来一步步分析上面这段代码的执行流程。

第一步是编译并创建执行上下文，下面是我画出来的执行上下文示意图，你可以参考下：

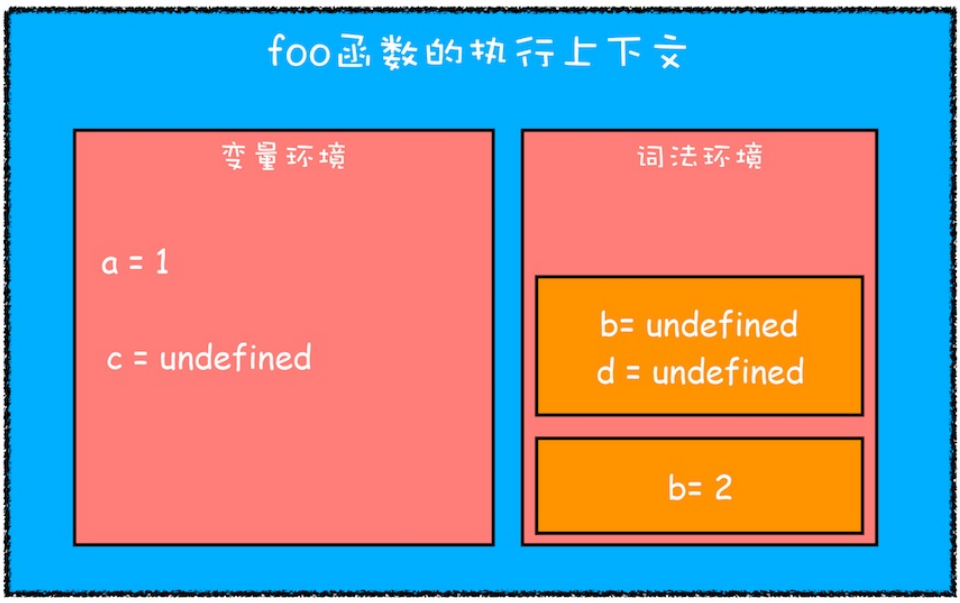


刚执行时foo函数的执行上下文

通过上图，我们可以得出以下结论：

- 函数内部通过var声明的变量，在编译阶段全都被存放到**变量环境**里面了。
- 通过let声明的变量，在编译阶段会被存放到**词法环境（Lexical Environment）**中。
- 在函数的作用域内部，通过let声明的变量并没有被存放到词法环境中。

接下来，**第二步继续执行代码**，当执行到代码块里面时，变量环境中a的值已经被设置成了1，词法环境中b的值已经被设置成了2，这时候函数的执行上下文就如下图所示：



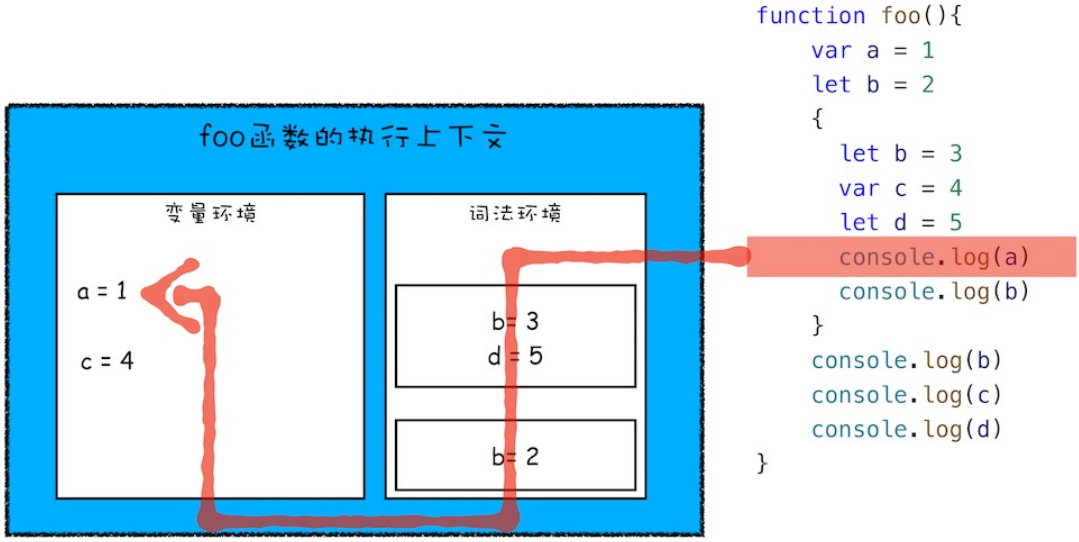
执行foo函数内部作用域块时的执行上下文

从图中可以看出，当进入函数的作用域块时，作用域块中通过let声明的变量，会被存放在词法环境的一个单独的区域中，这个区域中的变量并不影响作用域块外面的变量，比如在作用域外面声明了变量b，在该作用域块内部也声明了变量b，当执行到作用域内部时，它们都是独立的存在。

其实，在词法环境内部，维护了一个小型栈结构，栈底是函数最外层的变量，进入一个作用域块后，就会把该作用域块内部的变量压到栈顶；当作用域执行完成之后，该作用域的信息就会从栈顶弹出，这就是词法环境的结构。需要注意下，我这里所讲的变量是指通过let或者const声明的变量。

再接下来，当执行到作用域块中的`console.log(a)`这行代码时，就需要在词法环境和变量环境中查找变量a的值了，具体查找方式是：沿着词法环境的栈顶向下查询，如果在词法环境中的某个块中查找到了，就直接返回给JavaScript引擎，如果没有查找到，那么继续在变量环境中查找。

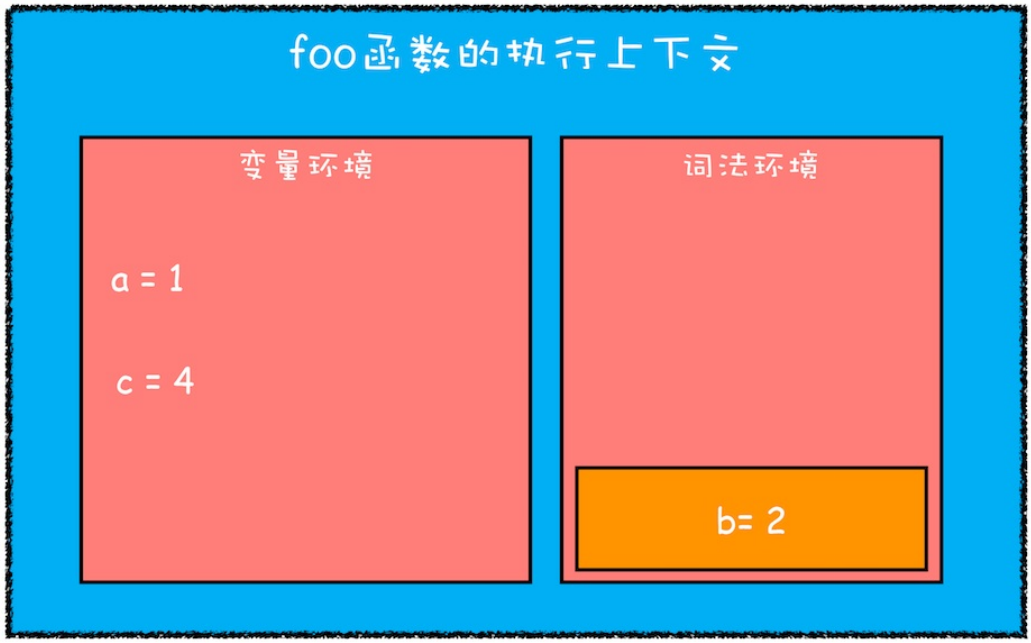
这样一个变量查找过程就完成了，你可以参考下图：



变量查找过程

从上图你可以清晰地看出变量查找流程，不过要完整理解查找变量或者查找函数的流程，就涉及到作用域链了，这个我们会在下篇文章中做详细介绍。

当作用域块执行结束之后，其内部定义的变量就会从词法环境的栈顶弹出，最终执行上下文如下图所示：



作用域执行完成示意图

通过上面的分析，想必你已经理解了词法环境的结构和工作机制，块级作用域就是通过词法环境的栈结构来实现的，而变量提升是通过变量环境来实现，通过这两者的结合，JavaScript引擎也就同时支持了变量提升和块级作用域了。

总结

好了，今天的内容就讲到这里，下面我来简单总结下今天的内容。

由于JavaScript的变量提升存在着变量覆盖、变量污染等设计缺陷，所以ES6引入了块级作用域关键字来解决这些问题。

之后我们还通过对变量环境和词法环境的介绍，分析了JavaScript引擎是如何同时支持变量提升和块级作用域的。

既然聊到了作用域，那最后我们再简单聊下编程语言吧。经常有人争论什么编程语言是世界上最好的语言，但如果站在语言本身来说，我觉得这种争论没有意义，因为语言是工具，而工具是用来创造价值的，至于能否创造价值或创造多大价值不完全由语言本身的特性决定。这么说吧，即便一门设计不那么好的语言，它也可能拥有非常好的生态，比如有完善的框架、非常多的落地应用，又或者能够给开发者带来更多的回报，这些都是评判因素。

如果站在语言层面来谈，每种语言其实都是在相互借鉴对方的优势，协同进化，比如JavaScript引进了作用域、迭代器和协程，其底层虚拟机的实现和Java、Python又是非常相似，也就是说如果你理解了JavaScript协程和JavaScript中的虚拟机，其实你也就理解了Java、Python中的协程和虚拟机的实现机制。

所以说，语言本身好坏不重要，重要的是能为开发者创造价值。

思考时间

下面给你留个思考题，看下面这样一段代码：

```
let myname= '极客时间'
{
  console.log(myname)
  let myname= '极客邦'
}
```

你能通过分析词法环境，得出来最终的打印结果吗？

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

精选留言：

- ytd 2019-08-24 09:52:55
  ```js
  let myname= '极客时间'
  {
  console.log(myname) // 报错，在chrome下会报Uncaught ReferenceError: Cannot access 'myname' before initialization
  let myname= '极客邦'
  }
  ```
  let声明的变量在编译阶段会被加入执行上下文的词法环境，而且不会被提升到作用域的顶部（这也就是块级作用域原理）
  或者可以这么说：myname直到`let myname= '极客邦'`声明语句被执行时才被初始化
  所以，在声明之前访问let声明的变量会报错。不知道这么理解合不合理？

但是浏览器的报错感觉有些让人疑惑，`Cannot access 'myname' before initialization
`，变量在初始化之前不能访问，那么既然let声明的变量未被提升，为什么不报变量未定义错误呢？期待老师的解惑。

- 高斌 2019-08-24 09:03:16
let myname= '极客时间'
{
console.log(myname)
let myname= '极客邦'
}

编译过程：
生成执行上下文压入栈
变量环境为空
词法环境中myname=undefined压入栈

执行过程：
词法环境中myname=极客时间
新开一个 myname =undefined 压入词法环境栈
查找myname并输出undefined
赋值当前栈头上myname=极客邦
pop栈头

结束

- 爱吃锅巴的沐泡 2019-08-24 08:33:39
有个疑问：
在思考题中，
1、执行到console.log(myname)这句话时，编译阶段已经完成，那么词法环境中的栈顶 是不是已经有了该作用域块了，let myname = ‘极客邦’ 是不是也已经在栈顶的作用域快中了？
2、执行到console.log(myname)这句话时，是按着从词法环境栈顶到栈底到变量环境的顺序查找，栈底已经存在了函数级的 let myname了，那为什么还是会报错呢？

- mfist 2019-08-24 07:03:53
1. 在块级作用域中，从 {开始到let myname= '极客邦' 代码之间会形成一个暂时性死区，如果中间去访问变量myname，会报初始化之前不能访问myname的错误。Uncaught ReferenceError

2. 另外上面的一个foo函数也会报d没有定义吧，d在块级作用域中声明，在外面是访问不到的
function foo(){
var a = 1
let b = 2
{
let b = 3
var c = 4
let d = 5
console.log(a)
console.log(b)
}
console.log(b)

console.log(c)
console.log(d)
}
foo()

作者回复2019-08-24 07:19:24
对的，你的分析没问题，这两行都会报错

- pyhhou 2019-08-24 04:39:04
思考题中，在 node 环境中 run 的话，console.log(myname) 这一行会报错，但是在网上的一些 JavaScript 的 editor 中 run，输出就是 undefined。一开始没有 run 代码，把词法环境当作变量环境来分析的话，我认为会输出 undefined，可能在 node 环境下，词法环境中可能还是会有逻辑去判断一个声明是在运行代码前还是后，比如
let a;
console.log(a); // undefined

如果声明在运行之后就会报错：
console.log(a); // 报错
let a;

想请教老师的一点细节方面的问题，就是平时写 JavaScript 代码每行结束后需要带上分号吗？看老师您这里写的代码有很多结束都没有带分号，但是之前看到过一篇文章说 JavaScript 里面是通过分号去判断一个语句的结束，不知道这一点在实际的开发中是否有影响？

作者回复2019-08-24 07:18:31
不写分号，代码更加清晰

另外这段代码就是会报错的，因为涉及到另外一个知识点，我在文中没有介绍，letl和const要在声明之后才能使用，很多人称这个为“暂时性死区”！

通过题目实践下会加深印象。