

## 04 | export default function() {}: 你无法导出一个匿名函数表达式

2019-11-18 周爱民

《JavaScript核心原理解析》

课程介绍 >



讲述：周爱民

时长 24:09 大小 22.12M



你好，我是周爱民，欢迎回到我的专栏。

今天我要讲述的内容是从 ECMAScript 6 开始在 JavaScript 中出现的**模块技术**，这对许多 JavaScript 开发者来说都是比较陌生的。

一方面在于它出现得较晚，另一方面，则是因为在普遍使用的 Node.js 环境带有自己内置的模块加载技术。因此，ECMAScript 6 模块需要通过特定的命令行参数才能开启，它的应用一直以来也就不够广泛。

导致这种现象的根本原因在于 **ECMAScript 6 模块是静态装配的**，而传统的 Node.js 模块却是动态加载的。因而两种模块的实现效果与处理逻辑都大相径庭，Node.js 无法在短期内提供有效的手段帮助开发者将既有代码迁移到新的模块规范下。



总结起来，确实是这些更为现实的原因阻碍了 ECMAScript 6 模块技术的推广，而非是 ECMAScript 6 模块是否成熟，或者设计得好与不好。

不过即使如此，ECMAScript 6 模块仍然在 JavaScript 的一些大型应用库、包，或者对新规范更友好的项目中得到了不错的运用和不俗的反响，尤其是在使用转译器（例如 Babel）的项目中，开发者通常是首选 ECMAScript 6 模块语法的。

因此 ECMAScript 6 模块也有着非常好的应用环境与前景。

## 导出的内容

上一讲我提到过有且仅有六种声明语法，而本质上 `export` 也就只能导出这六种声明语法所声明的标识符，并且在导出时将它们统一称为“名字”。

在语言设计中，所谓“标识符”与“名字”是有语义差别的，`export` 将之称为名字，就意味着这是一个标识符的子集。类似的其它子集也是存在的，例如“保留字是标识符名，但不能用作标识符（A reserved word is an IdentifierName that cannot be used as an Identifier）”。

在 JavaScript 语言的设计上，除了那些预设的标点符号（例如大括号、运算符之类），以及部分的保留字和关键字之外，事实上用户代码可以书写的只有三种东西。这包括：

- 标识符：（通常是）一个**名字**；
- 字面量：表明由它的字面含义所决定的一个**值**；
- 模板：一个可计算结果的字符串**值**。

所以，如果在这个层面上解构一份你所书写的 JavaScript 代码，那么你能书写 / 声明的，就一定只有“名字和值”。

这个结论是非常非常关键的。为什么呢？因为 `export` 事实上就只能导出“名字和值”。然而一旦它能导出“名字和值”，也就意味着它能导出一个模块中的“全部内容”，因为如上所面所讲的：

“名字和值”正是你所书写的代码的全部。

我的代码去哪儿了呢？



你是不是一刹那之间觉得自己的代码都白写了。:)

确实是的，真的是白写了。不过，我在前面讲的都是纯粹的“语言设计”，在语言设计层面上来讲，代码就是文本，是没有应用逻辑的。而你所写的代码绝大多数都是应用逻辑，当去除掉这些应用逻辑之后，那些剩下的死气沉沉的、纯粹的符号，才是语言层面的所谓“代码文本”。

去掉了执行逻辑所表达的那些行为、动作、结果和用户操作的代码，就是静态代码了。而事实上，ECMAScript 6 中的模块就是用来理解你的程序中的那些静态代码的，也就是那些没有任何生气的字符和符号。因此它也就只能理解上面所谓的 6 种声明，以及它们声明出来的那些“名字和值”。

再无其它。

## 解析 export

所以，将所有 export 语法分类，其实也就只有两个大类。如下：

 复制代码

```
1 // 导出“（声明的）名字”
2 export <let/const/var> x ...;
3 export function x() ...
4 export class x ...
5 export {x, y, z, ...};
6
7
8 // 导出“（重命名的）名字”
9 export { x as y, ...};
10 export { x as default, ... };
11
12
13 // 导出“（其它模块的）名字”
14 export ... from ...;
15
16
17 // 导出“值”
18 export default <expression>
```


关于导出声明的、重命名的和其它模块的名字这三种情况，其实都比较容易理解，就是形成一个名字表，让外部模块能够查看就可以了。



但是对于最后这种形式，也就是“（导出）值”的形式，事实上是非常特殊的。因为如同我在上面所讲过的，要导出一个模块的全部内容就必须导出“（全部的）名字和值”，然而纯粹的值没有名字，于是也就没法访问了，所以这就与“导出点什么东西”的概念矛盾了。

因为这个东西要是没名字，也就连“自己是什么”都说不清楚，也就什么也不是了。

所以 ECMAScript 6 模块约定了一个称为"default"的名字，用于来导出当前模块中的一个“值”。显然的，由于所谓“值”是表达式的运算结果，所以这里的语法形式就是：

 复制代码

```
1 export default <expression>;
```

其中的“\_expression\_”就是用于求值的，以便得到一个结果（Result）并导出成为缺省的名字“default”。这里有两个便利的情况，一个是在 JavaScript 中，一般的字面量也是值、也是单值表达式，因此导出这样一个字面量也是合法的：

 复制代码

```
1 export default 2; // as state of the module, etc.
2 export default "some messages"; // data or information
3 ...
```

第二个便利的情况，是因为 JavaScript 中对象也是字面量、也是值、也是单值表达式。而对对象成员可以组合其它任何数据，所以通过上述的语法几乎可以导出当前模块中全部的“值”（亦即任何可以导出的数据）。例如：


 复制代码

```
1 var varName = 100;
2 export default {
3   varName, // 直接导出名字
4   propName: 123, // 导出值
5   funcName: function() { }, // 导出函数
6   foo() { // 或导出与主对象相关联的方法
7     // method
8   }
9 }
```



所以，事实上`export default ...`虽然简单，却是对“导出名字”的非常必要的补充。这样一来，用户既可以导出那些有名字的数据，也可以导出那些没有名字的数据，即一个模块中所有的数据都可以被导出了。

那么接下来，就要讲到标题中的这个语法了：

 复制代码

```
1 export default function() {}
```

你知道在这个语法中 `export` 到底导出了什么吗？是名字？还是值？

## 导出语句的处理逻辑

在讨论这个问题之前，你得先思考一个更关键的问题：“`export` 如何导出名字”。这个问题的关键之处在于，如果只是导出一个名字，那么它其实在“某个名字表”中做一个登记项就可以了。并且 JavaScript 中也的确是这样处理的。但是实际使用的时候，这个名字还是要绑定一个具体的值才是可以使用的。因此，一个 `export` 也必须理解为这样两个步骤：

1. 导出一个名字
2. 为上述名字绑定一个值

这两个步骤其实与使用“`var x = 100`”来声明一个变量的过程是一致的。因此以如下代码为例（注意六种声明在名字处理上是类似的），

 复制代码

```
1 export var x = 100;
```

在导出的时候，其实是先在“某个名字表”中登记一个“名字 `x`”就可以了。这个过程也就是 JavaScript 在模块装载之前对 `export` 所做的全部工作。不过如果是从另一端（亦即是 `import` 语句）的角度看过来，那么就会多出来一个步骤。`import` 语句会（例如 `import {x} from ...`）：



1. （与 `export` 类似）按照语法在当前模块中声明名字，例如上面的 `x`；

## 2. 添加一个当前模块对目标模块的依赖项。

有了上述的第二步操作，JavaScript 就可以依据所有它能在静态文本中发现的 `import` 语句来形成模块依赖树，最后就可以找到这个模块依赖树最顶端的根模块，并尝试加载之。

所以关键的是，在“模块 `export/import`”语法中，JavaScript 是依赖 `import` 来形成依赖树的，与 `export` 无关。但是直到目前为止（我的意思是直到找到所有导入和导出的名字，并完成所有模块的装配的现在为止），没有任何一行用户的 JavaScript 代码是被执行过的。至于原因，从本讲的最开始我就讲过了：这个 `export/import` 过程中，源代码只被理解为静态的、没有逻辑的“代码文本”。那么既然“没有逻辑”，又怎么可能执行类似于：

 复制代码

```
1 export default <expression>;
```

中的“*expression*”呢？要知道所谓表达式，就是程序的计算逻辑啊。

所以，这里先得出了第一个关键结论：

在处理 `export/import` 语句的全程，没有表达式被执行！

## 导出名字与导出值的差异

现在，假如：

 复制代码

```
1 export default <expression>;
```

中的“*expression*”在导入导出中完全不起作用（不执行），那么这行语句又能做什么呢？事实上，这行语句与直接“导出一个名字”并没有任何区别。它与这样的语法相同：

 复制代码

```
1 export var x = 100;
```



它们都只是导出一个名字，只是前者导出的是“default”这个特殊名字，而后者导出的是一个变量名“x”。它们都是确定的、符合语法规则的标识符，也可以表示为一个字符串的字面文本。它们的作用也完全一致：就是在前面所说的“某个名字表”中添加“一个登记项”而已。

所以，导出名字与导出值本质上并没有差异，在静态装配的阶段，它们都只是表达为一个名字而已。

然后，也正是如同`var x = 100`；在执行阶段需要有一个将“值 100”绑定给“变量 x（的引用）”的过程一样，这个`export default ...`；语句也需要有完全相同的一个过程来将它后面的表达式（*expression*）的结果绑定给“default”这个名字。如果不这么做，那么“*export default*”在语义上的就无法实现导出名字“*default*”了——在静态装配阶段，名字“default”只是被初始化为一个“单次绑定的、未初始化的标识符”。

所以现在你就可以在语义上模拟这样一个过程，即：

 复制代码

```
1 export default function() {}
2
3 // 类似于如下代码
4 // （但并不在当前模块中声明名字"default"）
5 export var default = function() {}
6
```

你可以进一步地模拟 JavaScript 后续的装配过程。这个过程其实非常简单：

- 找到并遍历模块依赖树的所有模块（这个树是排序的），然后
- 执行这些模块最顶层的代码（*Top Level Module Evaluation*）。

在执行到上述`var default ....`（或类似对应的`export default ...`）语句时，执行后面的表达式，并将执行结果（Result）绑定给左侧的那个变量就可以了。如此，直到所有模块的顶层代码都执行完毕，那么所有的导出名字和它们的值也都必然是绑定完成了的。

同样，由于 import 的名字与 export 的名字只是一个映射关系，所以 import 的名字（所对应的值）也就初始化完成了。



再确切地说（这是第二个关键结论）：

所谓模块的装配过程，就是执行一次顶层代码而已。

## 匿名函数表达式的执行结果

接下来讨论语句中的... `function() {}`这个匿名函数表达式。

按照 JavaScript 的约定，匿名函数表达式可以理解为一个函数的“字面量（值）”。理解“字面量值”这个说法是很有意义的，因为它意味着它没有名字。你可不要在心中暗骂哦，这绝不是废话。

“字面量（值）没有名字”就意味着执行这个“单值表达式”不会在当前作用域中产生一个名字，即使这个函数是具名的，也必然是如此。所以，这才带来了 JavaScript 中的经典示例，即：具名函数作为表达式时，名字在块级作用域中无意义。例如：

```
1 // 具名函数作为表达式
2 var x1 = function x2() {
3     ...
4 }
5
6 // 具名函数（声明）
7 function x3() {
8     ...
9 }
```

 复制代码

上面的例子中，x1~3 都是具有不同的语义的。其中，x2 是不会在当前作用域（示例中是全局）中登记为名字的。而现在，就这一讲的主题来说，在使用下面的语法：

```
1 export default function() { }
2 export default function x() { }
```

 复制代码

导出一个匿名函数，或者一个具名的函数的时候，这两种情况下是不同的。但无论它是否具名，它们都是不可能在当前作用域中绑定给default这个名字，作为这个名字对应的值的。





这段处理逻辑被添加在语法：

```
ExportDeclaration: export default AnonymousFunctionDefinition;
```

NOTE: ECMAScript 是将这里导出的对象称为 `_Expression_/AssignmentExpression`，这里所谓 `_AnonymousFunctionDefinition_` 则是其中 `_AssignmentExpression_` 的一个具体实例。

的执行 (*Evaluation*) 处理过程中。也就是说当执行这行声明时，如果后面的表达式是匿名函数声明，那么它将强制在当前作用域中登记为“**default**”这样一个特殊的名字，并且在执行时绑定该匿名函数。所以，尽管语义上我们需要将它登记为类似 `var default ...` 所声明的名字“**default**”，但事实上它被处理成了一个不可访问的中间名字，然后影射给该模块的“某个名字表”。

不过需要注意的是，这是一个**匿名函数定义** (*AnonymousFunctionDefinition*)，而不是一个匿名函数表达式 (*Anonymous FunctionExpression*)。一般函数的语句则被称为声明（或更严谨地称为宣告，*Function Declarations*）。而所谓**匿名函数定义**，其本身是表述为：

```
aName = FunctionExpression
```

或类似于此的语法风格的。它可以用在一般的赋值表达式、变量声明的右操作数，以及对象声明的成员初始值等等位置。在这些位置上，该函数表达式总是被关联给一个名字。一方面，这种关联不是严格意义上的“名字 → 值”的绑定语义；另一方面，当该函数关联给名字

(*aName*) 时，JavaScript 又会反向地处理该函数（作为对象 *f*）的属性 *f.name*，使该名字指向 *aName*。

所以，在本讲中的“`export default function() {}`”，在严格意义上来说（这是第三个关键结论）：

它并不是导出了一个匿名函数表达式，而是导出了一个匿名函数定义 (*Anonymous Function Definition*)。

因此，该匿名函数初始化时才会绑定给它左侧的名字“**default**”，这会导致 `import f from ...` 之后访问 *f.name* 值会得到“**default**”这个名字。



类似的，你使用下面的代码也会得到这个“*default*”：

 复制代码

```
1 var obj = {  
2   "default": function() {}  
3 };  
4 console.log(obj.default.name); // "default"
```

## 知识补充

关于 `export`，还可以有一些补充的知识点。

- `export ...` 语句通常是按它的词法声明来创建的标识符的，例如 `export var x = ...` 就意味着在当前模块环境中创建的是一个变量，并可以修改等等。但是当它被导入时，在 `import` 语句所在的模块中却是一个常量，因此总是不可写的。
- 由于 `export default ...` 没有显式地约定名字“`default`（或 *default*）”应该按 `let/const/var` 的哪一种来创建，因此 JavaScript 缺省将它创建成一个普通的变量（`var`），但即使是在当前模块环境中，它事实上也是不可写的，因为你无法访问一个命名为“*default*”的变量——它不是一个合法的标识符。
- 所谓匿名函数，仅仅是当它直接作为操作数（而不是具有上述“匿名函数定义”的语法结构）时，才是真正匿名的，例如：

 复制代码

```
1 console.log((function(){}).name); // ""
```

- 由于类表达式（包括匿名类表达式）在本质上就是函数，因此它作为 `default` 导出时的性质与上面所讨论的是一致的。
- 导出项（的名字）总是作为词法声明被声明在当前模块作用域中的，这意味着它不可删除，且不可重复导出。亦即是说即使是用 `var x...` 来声明，这个 `x` 也是在 `_lexicalNames_` 中，而不是在 `_varNames_` 中。
- 所谓“某个名字表”，对于 `export` 来说是模块的导出表，对于 `import` 来说就是名字空间（名字空间是用户代码可以操作的组件，它映射自内部的模块导入名字表）。不过，如果用户代码不使用“`import * as ...`”的语法来创建这个名字空间，那么该名字表就只存在于



JavaScript 的词法分析过程中，而不会（或并不必要）创建它在运行期的实例。这也是我一直用“某个名字表”来称呼它的原因，它并不总是以实体形式存在的。

- 上述名字表简化了 ECMAScript 中对导入导出记录 (*ImportEntry/ExportEntry Record Fields*) 的理解。因此如果你试图了解更多，建议你阅读 ECMAScript 的具体章节。
- 没有模块会导出（传统意义上的）`main()`，因为 ECMAScript 为了维护模块的静态语义，而把执行过程及其入口的定义丢回给了引擎或宿主本身。

## 思考题


本讲的内容中，你需要重点复习三个关键结论的得出过程。这对于之前几讲中所讨论的内容会是很好的回顾。

除此之外，建议你思考如下问题：

- 为什么在 `import` 语句中会出现“变量提升”的效果？

如果你并不了解什么是“变量提升”，不用担心，下一讲中我会再次提到它。

分享给需要的人，Ta购买本课程，你将得 20 元

 生成海报并分享

 赞 8  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 03 | `a.x = a = {n:2}`：一道被无数人无数次地解释过的经典面试题

下一篇 05 | `for (let x of [1,2,3]) ...`：for循环并不比使用函数递归节省开销



# JVM + NIO + Spring

各大厂面试题及知识点详解

限时免费



## 精选留言 (37)

写留言



weineel

2019-11-18

ESModule 根据 import 构建依赖树，所以在代码运行前名字就是已经存在于上下文，然后在运行模块最顶层代码，给名字绑定值，就出现了‘变量提升’的效果。

作者回复: Yes! 满分答案👍



42



海绵薇薇

2019-11-22

hello 老师好，感谢老师之前的回答，有醍醐灌顶之效。

下面是读完这篇文章和下面评论之后的观点，不知是否有误，望指正，一如既往的感谢：)

1.

function a() {} // 函数声明，在六种声明内

function () {} // 报错，以function 开头应该是声明，但是没有名字



(function() {})// 函数表达式 (这是一个正真的匿名函数(function() {})).name 为 “”) , 即使是具名函数 (function a() {}), 当前作用域也找不到a, 因为这不是声明

var a = function() {} // 函数定义, 这里的function() {} 也是表达式, 只是赋给了变量a, 所以有了区别, 也有了名字a.name为a, 称作函数定义

var b = function c() {} // 函数定义, 函数function c() {} 也是表达式, 只是赋值给了变量b, 但是b.name却为c, 和上面存在的区别, 但也是函数定义

2.

导出的是"名字", 我理解为名字就像一个绳子, 后面拴的牛是会变的。这就是为什么import {a} from './a.js' 这个a会变, 虽然当前模块不能赋值给a。

作者回复: 对哒! 赞+2



11



万籁无声

2019-11-18

感觉没有抓住主题思想在表达什么, 可能是我层次太低了

作者回复: 正好, 刚写完“Y”同学的留言, 你不妨看看, 应该正好能回答你的疑问。

(万恶的极客时间没有提供分留言链接的功能, 产品同学要打手板心5次 🙄)



11



许童童

2019-11-18

为什么在 import 语句中会出现“变量提升”的效果?

如老师所说, 在代码真正被执行前, 会先进行模块的装配过程, 也就是执行一次顶层代码。所以如果import了一个模块, 就会先执行模块内部的顶层代码, 看起来的现象就是“变量提升”了。

作者回复: 😊👍



10



Y

2019-11-18

老师，关于这边文章的中心，我能总结成这个意思吗。

`export default function(){}.`这个语法本身没有任何的问题。但是他看似导出一个匿名函数表达式。其实他真正导出的是一个具有名字的函数，名字的`default`。

作者回复: 是的。不过，这算是题解。中心还是模块装载执行和标识符绑定全过程来着😄

标识符和值绑定是“声明”语法处理的核心，而六种声明是js静态语法的核心。而静态语法，也就是这一整篇“语言如何构建”的核心了🤔



👍 10



🇧🇪 Hazard🔗...

2020-03-23

老师，有一句话不太明白。

" `import` 的名字与 `export` 的名字只是一个映射关系 "。

`export` 一个变量，比如 `count`，如果设一个定时器执行，每次`count`都加 1；

`import { count }`，这个`count`也会每次都改变。这就是所说的映射关系吗？

这个映射关系是怎么做到的？

作者回复: 验证这个映射关系很简单。

B模块中`export`一个`let`变量，然后在A模块中`import`它为`x`。然后你尝试在A模块中`x++`，你会发现提示为常量不可写。

所以A、B两个模块中的名字其实并不是同一个变量，它们名字相同（或者不同），但A模块中只是通过一个（类似于别名的）映射来指向B模块中的名字。

映射是通过创建一个专用的数据结构来实现的，访问该结构就跳到目标数据，但每个操作都有特定的限制（例如上面的只读）。——这整体上有些类似于属性中的`get/setter`的机制，但并不是用属性描述符来做到的。



👍 6



leslee

2019-11-19

第三个结论推导过程的中间语法定义的引用那里(markdown '>' 符号表示的引用)读得不是很通顺, 有点迷....

作者回复: 这是因为类似于：



```
obj = {  
  f: function() {  
  },  
  ...  
}
```

这样位置中的匿名函数，在ECMAScript中都是称为“匿名函数定义”，而不是“匿名函数表达式”。所有在语法上记为“x = functionExpression”的，在处理上都与一般表达式有不同，这是一个非常非常小的细节，但在引擎层面，加入了好大一段逻辑呢。

真正的匿名函数表达式，是下面这样的：

```
> (1 + function() {})
```

就是：把它直接用在表达式计算过程中，而不是把它用来赋值（或绑定，或引用）给另一个东西。这种情况下，它才是按匿名函数表达式来处理的。

这几讲都是讲JavaScript的静态语言特性的，所以“词法分析以及对应的引擎处理”是要点，在词法分析阶段，关键在于“不能为它（函数、函数表达式、函数定义等等）创建闭包”。因为在静态处理阶段，还没有“闭包”这个概念，所以好多东西处理起来跟我们平常的理解不同，这就是根由了。



👍 5



leslee

2019-12-14

是否可以理解为，一个具有了名字的函数表达式就可以称为函数定义

作者回复: Yes. 这样理解没错。



👍 4



穿秋裤的男孩

2019-11-29

所谓模块的装配过程，就是执行一次顶层代码而已。

这边的顶层代码是指什么呢？模块装配不是在静态解析期进行的吗？为什么还会执行代码？还是这边指的执行并不是一般意义上的执行呢？

作者回复: ``

```
// t.mjs
```

```
console.log("here =>", typeof f);
```

```
import f from './f.mjs';
```



```
// f.mjs
export default function() {}
console.log('NOW');

// test
> node --experimental-modules t.mjs
NOW
here => function
````
```

想想，

1. 为什么`here`为什么是function呢？import语句还没有到呢。
2. 为什么`NOW`在`here`之前？这是哪个时候的执行过程？

共 2 条评论 >

👍 3



**Marvin**

2019-11-20

export default v=>v 这种，箭头函数是特例吗？

作者回复: 有点特殊，但就处理逻辑（以及目的）上来说，也并不算是特例。

其实“函数定义（Function Definition）”这个概念出现得比较奇怪。

仔细分析一下就明白了，你想，“函数声明（Function Declaration）”是静态语义的，它在执行期的结果是empty，所以它必须是具名的才能导出，因为“声明（6种）”的目的都是具名，而export原则上只能“导出一个名字”。所以，由于“函数定义（Function Definition）”没有名字，所以它不能按函数声明来处理。

然后，由于“函数表达式（Function Expression）”是动态语义的，有执行语义（也就是执行结果返回不是empty），得到一个运行期概念上的“闭包”。但这并不是最关键处，最关键的地方在于函数表达式没名字——即使是具名的函数表达式，它的名字也只能闭包内有影响。由于它没有名字一个可供导出的名字，所以也不能直接直接用作export的对象。

那么到底在概念上该怎么说这个东西呢？ECMAScript在这里就加了这么一层概念，叫“函数声明（Function Declaration）”，一方面它是有静态语义的，它声明了某个东西；另一方面，它的名字又是迟绑定的，需要到了执行期根据“name = FunctionExpression”中的`name`来确认。

在这种情况下，其实“函数定义（Function Definition）”就是“函数表达式”的一层概念封装：它又有在外层（或被关联的对象）中的名字，它又是表达式；它的执行结果又是闭包，又是实例。





所以箭头函数看起来是特例，但用在导出语法的“这个位置”时，概念上却仍然是“封装了一层的‘箭头函数表达式’”，仍然还是“函数定义”。

共 4 条评论 >

👍 3



七月有风

2020-02-22

ECMAScript 6 模块是静态装配的，而传统的 Node.js 模块却是动态加载的。是不是说node是在执行阶段才会执行模块的顶层代码。

作者回复: nodejs中，是在require()函数执行过程中来执行模块的顶层代码的。

nodejs模块被封装在一个函数中（亦即是作为一个函数的函数体），由require()在加载完指定模块的文本代码之后，用普通的调用函数的方法调用，从而实现模块装载的。



👍 3



晓小东

2019-12-19

老师，我又来了，怕您看不到我的问题，接上一个问题，函数声明标识符不应该放入词法环境中，本来我想函数声明标识符放入词法环境，来验证函数声明提升优先级高于var，因为标识符的查找先从词法环境中查找，再到变量环境，再到上级作用域，从而实现声明的优先级。老师对于函数声明的优先级，你怎么看。

作者回复: 关于这个问题，其实还挺好玩儿的，因为它涉及到`execute\_context.VariableEnvironment`这个东西怎么用的问题。

首先，其实词法环境(LexicalEnvironment)与变量环境(VariableEnvironment)并没有一个所谓优先级的问题。在实现上，它们之间是一个使用env.outer来衔接起来的链，所以所谓查找顺序，本质上就是二者谁在链的外层的问题。——然而，从实际实现的角度上，二者并不需要强调谁在外层，这种关系不是必须的（它们只需要衔接在一起就可以了）。

除了在函数或全局初始化需要一个表来指示“哪些东西是var和函数名”之外，事实上区分var/let/const之间的必要性是不大的。并且即使是在这种情况下，引擎也并不需要VariableEnvironment这个东东的参与，因为在它们初始化时，引擎是可以访问来自源代码的ParserNode的。也就是说，它可以直接访问原始的信息，而不必依赖VariableEnvironment这个列表。

VariableEnvironment这个东西，以及LexicalEnvironment，它们都是给运行期的上下文用的，也只在运行期才有意义。——更进一步的，只有对全局和函数，在它们的执行期才有意义（对函数来说，是它被调用的时候）。



为什么呢？就目前而言，VariableEnvironment其实只在一种情况下被用到。——就是当全局或函数内出现eval('var x...')这样的代码的时候。因为只有在这种情况下，在相应的变量环境中，才会需要执行上下文去访问变量环境列表，并动态地向中间插入一个新的名字。由于事实上var变量只能全局和函数有用，所以四种执行上下文（Global/Function/Module/Eval）中，虽然都有这两个成员，但其实Module.VariableEnvironment是没有用的，而Eval.VariableEnvironment受限于是否是在严格模式（当处在非严格模式时，它指向外层的——例如函数的VariableEnvironment；当处在严格模式时，它将自己创建一个，以隔离开对外部环境的影响）。

所以，本质上你来看VariableEnvironment这个东西的时候，不是要去“检查”它有什么样的优先级，而是直接看到“它有什么用，它怎么用”。再一次强调，对于单向链表访问来说，所谓“优先级”就是谁在链尾的问题；但即使如此，它对VariableEnvironment的使用来说也没有什么意义，因为VariableEnvironment归ExecuteContext使用，而ExecuteContext根本不care这个顺序。



👍 3



**穿秋裤的男孩**

2019-11-29

可以这样理解吗？

静态解析期：export只导出名字到某个名字表，import从名字表获取映射关系。

执行期：执行代码，为名字赋值。

作者回复: 是的。这个“执行期”在用户代码之前。



👍 3



**ZXCv**

2021-04-01

在语言设计中，所谓“标识符”与“名字”是有语义差别的，export 将之称为名字，就意味着这是一个标识符的子集。类似的其它子集也是存在的，例如“保留字是标识符名，但不能用作标识符（A reserved word is an IdentifierName that cannot be used as an Identifier）”。

看不懂哇～

作者回复: 这部分在ECMAScript里面就是这么讲述的，那段E文是取自ECMAScript的原文。关于这些概念，你可以看ECMAScript的相关内容，有些中文的可以看，在这里：

<https://www.w3.org/html/ig/zh/wiki/ES5>

有些内容是在后面会介绍到，比如这些概念为什么要这么讲。第5讲后面的加餐，以及第11讲后的加餐都很重要。不过，不用着急，慢慢学到那些章节，就明白了。



👍 2



```
"use strict";  
(function a() {  
  const a = 2;  
  console.log(a);  
})();
```

老师您好,这个函数名a 不是已经作为函数内部的标识符了吗,为什么还可以重新声明呢?

作者回复: 关于这个问题, 在《JavaScript语言精髓与编程实践》的第“5.5.2.4 函数表达式的特殊性”中专门有讲过, 主要是因为“函数名作为标识符所声明的位置”所导致的。

具体来说, 如果是函数声明, 那么函数名是声明在它“所在”上下文的, 因此它是否能“重新声明”取决于它所在的（外部的）上下文的严格模式状态。例如:

```
...  
  
function foo() {  
  function f() {  
    "use strict";  
    f = 1; // 可重写, 因为`f`声明在foo()中  
  }  
  f()  
  console.log(typeof f); // number  
}  
  
foo()  
...
```

为了在函数表达式中达成类似的效果（语言的一致性），所以函数表达式中这个函数名，也不是声明在函数体（以及由函数体所决定的闭包）中的。它采用了“双层作用域”的特殊构造，也就是函数名声明在外部作用域中（outerScope），而闭包的parent再指向这个outerScope。——函数的"use strict"只影响到函数自己的闭包。

所以回到你的例子，

```
...  
  
"use strict";  
(function a() {  
  const a = 2;  
  console.log(a);  
})();  
...
```



由于是函数表达式，所以`a()`作为名字其实是声明在一个outerScope中的——没错，这个scope也是strict模式的。接下来函数body中声明了`count a`，这个名字所在的作用域（闭包）中并没有`a`这个名字，所以无论其外部，或者内部是否是严格模式，这个名字`a`都是可以创建的。

与此不同的是，函数参数是声明在闭包中的，所以它表现得跟函数名不同：如果函数参数中有名字a，那么上例中的`const a`就无法声明了。这同样也证明了函数名`a()`需要一个outerScope的重要性，因为历史中下面这样的代码“总是”合法的（无论是函数声明还是函数表达式）：

```
...  
function a(a) {  
  ...  
}
```

共 2 条评论 >



w

李李

2020-08-14

我认为知识点讲解是要深入浅出，好难接受这种风格。 所以的知识点都事无巨细看似很全但是没有重点。 看着难受....



孜孜

2020-06-29

是不是因为“import语句所在的模块中却是一个常量”，这样才能保证无论多少个import，它们始终都是指向的是export那个变量？

作者回复: 是的。

另外，在设计上这符合“谁创建谁负责”的原则。对于export出来的x，所有的importer只能读；如果source发布x的写行为，那么应该另外export一个写方法出来。这类似于在对象中使用get/setter来访问属性，其设计原则和风格是一致的。



Gamehu

2020-02-21

所以当都是export default...，以default为名字，但是import xx from ...，其实xx是import 重命名了default是么？不然就没法使用了



作者回复: 你写得有点不通顺, 我只能试着答复你了。

export default会导致当前模块在命名空间中有一个特殊的名字, 这个名字被记为`\*default\*`。由于它不符合命名规则, 所以用户代码中既不可能使用, 也不能声明出来。这样处理, 是当“缺省导出”的名字有唯一性。

在使用`import xx from ...`的语法时, `xx`与模块的命名空间中的`\*default\*`其实创建了一个映射。这个映射在引擎内部也就是一次访问的跳转, 这个是基于“引用 (规范类型)”来实现的。

你说它是“重命名了default”, 宽泛地说, 是对的。因为基本上这与“重命名”的效果很接近。但, 本质上来说, “名字访问”是一次跳转, 而“(对名字空间中的) 映射访问”其实是二次跳转。



2



海绵薇薇

2019-11-28

Hello 老师好: )

函数定义

```
var a = function foo() {  
  
    console.log(foo)  
  
}
```

当前上下文没有标识符foo, 但是foo函数内却可以拿到该标识符, 所以foo这个标识符应该是声明了, 但是不在当前作用域, 那么可以简单理解为

```
var a = eval(`\  
    let foo;\  
    foo = function () {\  
        console.log(foo)\  
    }\  
`)
```

可以这么理解吗?



作者回复: 是的。这样没问题。除了缺一咪咪的严谨之外, 你的理解是对的。^^.



2

**周星星**

2021-03-30

因此，该匿名函数初始化时才会绑定给它左侧的名字“default”，这会导致import f from ...之后访问f.name值会得到“default”这个名字。

在导出匿名函数定义，我没有在导入的js中打印出来 'default'，而是空字符串，老师，这里是为什么啊，有其他同学遇到过吗

作者回复: 这个是课程中排版错误导致的误会。匿名函数定义在内部被声明的名字是`\*default\*`，注意前后有个`\*`号，这个被极客时间课程排版给吞掉了，所以.....

由于这个名字是不合JS的命名规范的，因此它也不会被显示出来。当使用import来导入这个函数时，也会在它作为一个导入变量时被赋予一个函数名（function.name），因此不必过分在意这个名字的使用。

这一段主要是说明匿名函数定义会用一个“特殊的（不合命名规范的）”名字被登记——它需要一个登记名字，而又不与其它被导出的名字潜在冲突。

