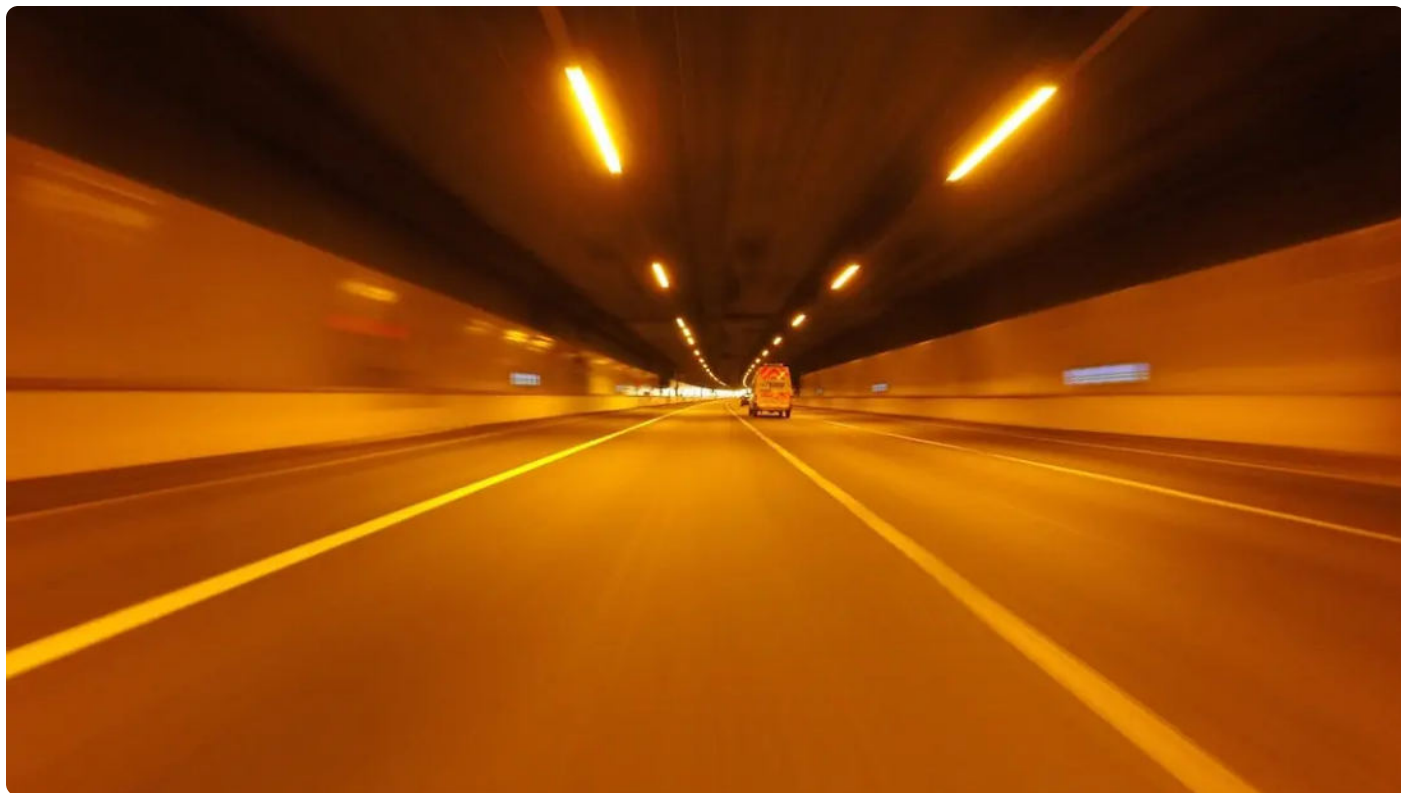


# 10 | `x = yield x`: 迭代过程的“函数式化”

2019-12-04 周爱民

《JavaScript核心原理解析》

[课程介绍 >](#)



讲述：周爱民

时长 18:34 大小 17.01M



你好，我是周爱民。欢迎回到我的专栏。

相信上一讲的迭代过程已经在许多人心中留下了巨大的阴影，所以很多人一看今天的标题，第一个反应是：“又来！”

其实我经常习惯用**同一个例子**，或者**同类型示例的细微不同**去分辨与反映语言特性上的核心与本质的不同。如同在 [第 2 讲](#) 和 [第 3 讲](#) 中都在讲的连续赋值，看起来形似，却根本上不同。

同样，我想你可能也已经注意到了，在 [第 5 讲](#) (`for (let x of [1,2,3]) ...`) 和 [第 9 讲](#) (`((...x))`) 中所讲述的内容是有一些相关性的。它们都是在讲循环。但第 5 讲主要讨论的是语句对循环的抽象和如何在循环中处理块。而第 9 讲则侧重于如何通过函数执行把（类似第 5 讲的）语句执行重新来实现一遍。事实上，仅仅是一个“循环过程”，在 JavaScript 中就实现了好几次。这些我将来都会具体地来为你分析。



至于今天，我还是回到函数的三个语义组件，也就是“参数、执行体和结果”来讨论。上一讲本质上讨论的是对“执行体”这个组件的重造，今天，则讨论对“参数和结果”的重构。


## 将迭代过程展开

通过上一讲，你应该知道迭代器是可以表达为一组函数的连续执行的。那么，如果我们要把这一组函数展开来看的话，其实它们之间的相似性是极强的。例如上一讲中提到的迭代函数 `foo()`，当你把它作为对象 `x` 的迭代器符号名属性，并通过对象 `x` 来调用它的迭代展开，事实上也就相当于只调用了多次的 `return` 语句。

 复制代码

```
1 // 迭代函数
2 function foo(x = 5) {
3   return {
4     next: () => {
5       return {done: !x, value: x && x--};
6     }
7   }
8 }
9
10 let x = new Object;
11 x[Symbol.iterator] = foo; // default `x` is 5
12 console.log(...x);
```

事实上相当于只调用了 5 次 `return` 语句，可以展开如下：

 复制代码

```
1 // 上例在形式上可以表达为如下的逻辑
2 console.log(
3   /*return *//{done: false, value: 5}.value,
4   /*return *//{done: false, value: 4}.value,
5   /*return *//{done: false, value: 3}.value,
6   /*return *//{done: false, value: 2}.value,
7   /*return *//{done: false, value: 1}.value
8 );
```

在形式上，类似上面这样的例子也可以展开来，表现它作为“多个值”的输出过程。



事实上连续的 `tor.next()` 调用最终仅是为了获取它们的值 (`result.value`)，那么如果封装这些值的生成过程，就可以用一个新的函数来替代一批函数。

这样的函数就称为**生成器函数**。

但是，由于函数只有一个出口（RETURN），所以用“函数的退出”是无法映射“**函数包含一个多次生成值的过程**”这样的概念的。如果要实现这一点，就必须让函数可以多次进入和退出。而这，就是今天这一讲的标题上的这个yield 运算符的作用。这些作用有两个方面：

1. 逻辑上：它产生一次函数的退出，并接受下一次 `tor.next()` 调用所需要的进入；
2. 数据上：它在退出时传出指定的值（结果），并在进入时携带传入的数据（参数）。

所以，yield实际上就是在生成器函数中用较少的代价来实现一个完整“函数执行”过程所需的“参数和结果”。而至于“执行体”这个组件，如果你听过上一讲的话，相信你已经知道了：执行体就是 `tor.next()` 所推动的那个迭代逻辑。

例如，上面的例子用生成器来实现就是：

```
1 function *foo() {
2   yield 5;
3   yield 4;
4   yield 3;
5   yield 2;
6   yield 1;
7 }
```

 复制代码

或者更通用的过程：

```
1 function *foo2(x=5) {
2   while (x--) yield x;
3 }
4
5 // 测试
6 let x = new Object;
7 x[Symbol.iterator] = foo2; // default `x` is 5
8 console.log(...x); // 4 3 2 1 0
```

 复制代码



我们又看到了循环，尽管它被所谓的生成器函数封装了一次。

## 逻辑的重现

我想你已经注意到了，生成器的关键在于如何产生yield运算所需要的两个逻辑：（函数的）退出和进入。

事实上生成器内部是顺序的 5 行代码，还是一个循环逻辑，所以对于外部的使用者来说它是不可知的。生成器通过一个迭代器接口的界面与外部交互，只要for...of或...x以及其他任何语法、语句或表达式识别该迭代器接口，那么它们就可以用 tor.next() 以及 result.done 状态来组织外部的业务逻辑，而不必界面后面的（例如数据传入传出的）细节了。

然而，对于生成器来说，“（函数的）退出和进入”是如何实现的呢？

在🔗第 6 讲 (x: break x;) 中提到过“**执行现场**”这个东西，事实上它包括三个层面的概念：

1. 块级作用域以及其他的作用域本质上就是一帧数据，交由所谓“环境”来管理；
2. 函数是通过 CALL/RETURN 来模拟上述“数据帧”在栈上的入栈与出栈过程，也称为调用栈；
3. 执行现场是上述环境和调用栈的一个瞬时快照（包括栈上数据的状态和执行的“位置”）。

其中的“位置”是一个典型的与“（逻辑的）执行过程”相关的东西，第六讲中的“break”就主要在讲这个“位置”的控制——包括静态的标签，以及标签在执行过程中所映射到的位置。

函数的进入（CALL）意味着数据帧的建立以及该数据帧压入调用栈，而退出（RETURN）意味着它弹出栈和数据帧的销毁。从这个角度上来说，yield运算必然不能使该函数退出（或者说必须不能让数据帧从栈上移除和销毁）。因为yield之后还有其他代码，而一旦数据帧销毁了，那么其他代码就无法执行了。

所以，yield是为数不多的能“挂起”当前函数的运算。但这并不是yield主要的、标志性的行为。yield操作最大的特点是**它在挂起当前函数时，还将函数所在栈上的执行现场移出了调用栈**。由于yield可以存在于生成器函数内的第 n 层作用域中。



复制代码

```
1 function foo3() { // 块作用域1
2   if (true) { // 块作用域2
3     while (true) { // 块作用域3
4       yield 100
```

所以，一个在多级的块作用域深处的`yield`运算发生时，需要向这个数据帧（作用域链）外层检索到第一个函数帧（即函数环境，`FunctionEnvironment`），挂起它以及它内部的全部环境。而执行位置，将会通过函数的调用关系，一次性地返回到上一次 `tor.next()` 的下一行代码。也就是说相当于在 `tor.next()` 内部执行了一次`return`。

为了简化所谓“向外层检索”这一行为，JavaScript 通常是使用所谓“执行上下文”来管理这些数据帧（环境）与执行位置的。执行上下文与函数或代码块的词法上下文不同，因为执行上下文只与“可执行体”相关，是 JavaScript 引擎内部的数据结构，它总是被关联（且仅只关联）到一个函数入口。

由于 JavaScript 引擎将 JavaScript 代码理解为函数，因此事实上这个“执行上下文”能关联所有的用户代码文本。

“所有的代码文本”意味着“.js 文件”的全局入口也会被封装成一个函数，且全部的模块顶层代码也会做相同的封装。这样一来，所有通过文件装载的代码文本都会只存在于同一个函数中。由于在 Node.js 或其他一些具体实现的引擎中，无法同时使用标准的 ECMAScript 模块装载和.js 文件装载，因此事实上来说，这些引擎在运行 JavaScript 代码时（通常地）也就只有一个入口的函数。

而所有的代码其实也就只运行在该函数的、唯一的一个“执行上下文”中。

如果用户代码——通过任意的手段——试图挂起这唯一的执行上下文，那么也就意味着整个的 JavaScript 都停止了执行。因此，“挂起”这个上下文的操作是受限制的，被一系列特定的操作规范管理。这些规范我在这一讲的稍晚部分内容中会详细讲述，但这里，我们先关注一个关键问题：**到底有多少个执行上下文？**

如果模块与文件装载机制分开，那么模块入口和文件入口就是二选一的。当然在不同的引擎中这也不尽相同，只是在这里分开讨论会略为清晰一些。

**模块入口**是所有模块的顶层代码的顺序组合，它们被封装为一个称为“顶层模块执行（`TopLevelModule Evaluation Job`）”的函数，作为模块加载的第一个执行上下文创建。类



似的是，一般的.js 文件装载也会创建一个称为“脚本执行（Script EvaluationJob）”的函数。后者，也是文件加载中所有全局代码块称为“Script 块”的原因。

除了这两种执行上下文之外，eval() 总是会开启一个执行上下文的。

JavaScript 为 eval() 所分配的这个执行上下文，与调用 eval() 时的函数上下文享有同一个环境（包括词法环境和变量环境等等），并在退出 eval() 时释放它的引用，以确保同一个环境中“同时”只有一个逻辑在执行。

接下来，如果一个一般函数被调用，那么它也将形成一个对应的执行上下文，但是由于这个上下文是“被”调用而产生的，所以它会创建一个“调用者（caller）”函数的上下文的关联，并创建在 caller 之后。由于栈是后入先出的结构，因此总是立即执行这个“被调用者（callee）”函数的上下文。

这也是调用栈入栈“等义于”调用函数的原因。

但这个过程也就意味着这个“当前的（活动的）”调用栈是由一系列执行上下文以及它们所包含的数据帧所构成的。而且，就目前来说，这个调用栈的底部，要么是模块全局

（\_TopLevelModuleEvaluationJob\_ 任务），要么就是脚本全局（\_ScriptEvaluationJob\_ 任务）。

一旦你了解了这些，那么你就很容易理解生成器的特殊之处了：

所有其他上下文都在执行栈上，而生成器的上下文（多数时间是）在栈的外面。

## 有趣的.next() 方法

如果有一行yield代码出现在生成器函数中，那么当这个生成器函数执行到yield表达式时会发生什么呢？

这个问题貌似不好回答，但是如果问：是什么让这个生成器函数执行到“yeild表达式”所在位置的呢？这个问题就好回答了：是 tor.next() 方法。如下例：



复制代码

```
1 function* foo3() {  
2   yield 10;  
}
```

```
3 }  
4 let tor = foo3();  
5 ...
```

我们可以简单地写一个生成器函数`foo3()`，它的内部只有一行`yield`代码。在这样的一个示例中，调用 `foo3()` 函数之后，你就已经获得了来自 `foo3()` 的一个迭代器对象，在习惯上的，我称它为 `tor`。并且，在语法形式上，貌似 `foo3()` 函数已经执行了一次。

但是，事实上 `foo3()` 所声明的函数体并没有执行（因为它是生成器函数），而是直到用户代码调用`tor.next()`的时候，`foo3()` 所声明的函数体才正式执行并直到那唯一的一行代码：表达式`yield`。

 复制代码

```
1 # 调用迭代器方法  
2 > tor.next()  
3 { value: 10, done: false }
```

这时，`foo3()` 所声明的函数体正式执行，并直到表达式`yield 10`，生成器函数才返回了第一个值`10`。

如同上一讲中所说到的，这表明在代码`tor = foo3()`中，函数调用“`foo3()`”的实际执行效果是：**生成一个迭代过程，并将该过程交给了 `tor` 对象。**

换言之：`tor` 是 `foo3()` 生成器（内部的）迭代过程的一个句柄。从引擎内的实现过程来说，`tor` 其实包括状态（`state`）和执行上下文（`context`）两个信息，它是 `GeneratorFunction.prototype` 的一个实例。这个 `tor` 所代表的生成器在创建出来的时候将立即被挂起，因此状态值（`state`）初始化置为“启动时挂起（`suspendedStart`）”，而当在调用 `tor.next()` 因`yield`运算而导致的挂起称为“Yield 时挂起（`suspendedYield`）”。

另一个信息，即 `context`，就指向 `tor` 被创建时的上下文。上面已经说过了，所谓上下文一定指的是一个外部的、内部的或由全局 / 模块入口映射成的函数。

接下来，当 `tor.next()` 执行时，`tor` 所包括的 `context` 信息被压到栈顶执行；当 `tor.next()` 退出时，这个 `context` 就被从栈上移除。这个过程与调用 `eval()` 是类似的，总是能保证指定栈是全局唯一活动的一个栈。



如果活动栈唯一，那么系统就是同步的。

因为只需要一个执行线程。

## 对传入参数的改造

生成器对“函数执行”的执行体加以改造，使之变成由 `tor.next()` 管理的多个片断。用来映射多次函数调用的“每个 body”。除此之外，它还对传入参数加以改造，使执行“每个 body”时可以接受不同的参数。这些参数是通过 `tor.next()` 来传入，并作为 `yield` 运算的结果而使用的。

这里 JavaScript 偷偷地更换了概念。也就是说，在：

```
1 x = yield x
```

 复制代码

这行表达式中，从语法上看是表达式 `yield x` 求值，实际的执行效果是：

- `yield` 向函数外发送计算表达式 `x` 的值；

而 `x = ...` 的赋值语义变成了：

- `yield` 接受外部传入的参数并作为结果赋给 `x`。

将 `tor.next()` 联合起来看，由于 `tor` 所对应的上下文在创建后总是挂起的，因此第一个 `tor.next()` 调用总是将执行过程“推进”到第一行 `yield` 并挂起。例如：

```
1 function* foo4(x=5) {
2   console.log(x--); // `tor = foo4()` 时传入的值5
3   // ...
4
5   x = yield x; // 传出 `x` 的值
6   console.log(x); // 传入的arg
7   // ...
8 }
9
10 let tor = foo4(); // default `x` is 5
11 result = tor.next(); // 第一次调用.next()的参数将被忽略
12 console.log(result.value)
```

 复制代码





执行结果将显示：

```
1 5    // <- default `5`  
2 4    // <- result.value `4`
```

 复制代码

而 `foo4()` 函数在 `yield` 表达式执行后将挂起。而当在下一次调用 `tor.next(arg)` 时，一个已经被 `yield` 挂起的生成器将恢复（resume），这时传入的参数 `arg` 就将作为 `yield` 表达式（在它的上下文中）的结果。也就是上例中第二个 `console.log(x)` 中的 `x` 值。例如：

```
1 # 传入100，将作为foo4()内的yield表达式求值结果赋给`x = ...`  
2 > tor.next(100)  
3 100
```

 复制代码

## 知识回顾

今天这一讲，谈的是将迭代过程展开并重新组织它的语义，然后变成生成器与 `yield` 运算的全过程。

在这个过程中，你需要关注的是 JavaScript 对“迭代过程”展开之后的代码体和参数处理。

事实上，这包含了对函数的全部三个组件的重新定义：代码体、参数传入、值传出。只不过，在 `yield` 中尤其展现了对传入传出的处理而已。

## 思考题

今天的这一讲不安排什么特别的课后思考，我希望你能补充一下一个小知识点的内容：由于今天的内容中没有讲“委托的 `yield`”这个话题，因此你可以安排一些时间查阅资料，对这个运算符——也就是“`yield*`”的实现过程和特点做一些深入探索。

欢迎你在进行深入思考后，与其他同学分享自己的想法，也让我有机会能听听你的收获。



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 1 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 09 | (...x): 不是表达式、语句、函数，但它却能执行

下一篇 11 | throw 1; 它在“最简单语法榜”上排名第三

## 学习推荐

# JVM + NIO + Spring

各大厂面试题及知识点详解

限时免费



## 精选留言 (10)

写留言



授人以摸鱼

2020-03-26

我忽然想明白为啥会有迭代器的next需要支持传入参数这样的功能了.....以前一直没想明白来着.....

其实就是作为生成器函数的一个应用实例，co模块需要这个功能，需要把yield返回的promise like对象的then方法传回的值从next给生成器函数传回去，这个需求抽象一下，就成了“外部执行环境会根据yield传出的结果进行变换后用next传入”这样的通用需求了。



所以我同时也就理解了，为啥说async await是生成器函数的语法糖了，而且这糖真甜www  
w

作者回复: : )

赞的！又是大有所得~ ^^.



👍 8



许童童

2019-12-04

如果遇到 yield\* 就将当前的yield执行权交到 yield\* 里面，yield\* 里面return的值，将返回给外层的x = yield\* xxx 中的x

作者回复: Yes. 赞的！ +2



👍 3



红白十万一只

2020-03-08

老师，最近遇到个问题

```
if (true) {  
  a = 5  
  function a() {}  
  a = 1  
  console.log(a)  
}  
console.log(a)  
1  
5
```

外部的a变成了5，内部的a变成了1

我查了资料说：

function a() {}的函数提升只提升到了if的代码块顶部，但也有一个var声明被悬挂到了全局中值为undefined

之后在a=5时if内部的变量a从函数被修改为了5

function a() {}这步计算函数声明时，函数对象被分配给函数作用域变量。因为被修改为5，所以此时外部的a也被改变成了5

之后在a=1时if内部的变量a从5被修改为了1

最后输出1 5

我的理解是

函数提升只提升到代码块顶部，但是为了符合函数作用域的规则在最近的函数作用域创建了一



个同名变量且值为undefined。

之后再执行到声明语句时将这个函数对象转换为值赋值给这个函数作用域同名变量。

老师有更详细的解释么？我找了ES规范没有发现对这里解释。

作者回复: 这个东西说起来，就有点历史了。呵呵

早期的JScript认为，只要在同一个作用域（函数/全局）内声明函数，那么同名函数是覆盖的，也就是后一个声明覆盖前一个。所以按照这个逻辑，对“同一个名字的”连续两次声明其实只有后一个有效，并且即使是在前一个之前访问它，也是一样的（因为有变量提升，所以这个域中能访问到的只有第二个声明）。

这个特性直到较晚一些的IE中都是如此，包括IE8或者IE9，但是IE10之后我就知道了。然而Firefox/SpiderMonkey却不这么认为，在早期还没有块级作用域的时候，他们就搞了一个称为“条件声明”的东西出来，也就是在if的两个分支中如果声明了函数，只有执行到指定分支时，该分支中的声明才有效。——因为那时还没有“块级作用域”这么个概念，所以这个就算成了动态的、“执行期的”逻辑。

而你的这个例子，就是这个特性在起作用了。——在执行到function a()这个声明之前，会因为变量泄露而在全局创建标识符`a`；而在执行到function a()的时候，就在当前块中创建了函数a()。

这个特性呢，在标准的ECMAScript（的正文部分）中是没有的。因为对于Firefox来说，这是一个特殊的、自有的特性，不属于ECMAScript规范，但由于它一直存在于SpiderMonkey系的引擎中，所以在MDN中也一直有这一部分的解释。

后来在ECMAScript中，因为这是一个“通常应于浏览器环境”的特性，所以把它归在了“Web Legacy Compatibility”中，是作为扩展特性来介绍的，不在ECMAScript的正文中。这与两个主题有关：

> <https://tc39.es/ecma262/#sec-block-level-function-declarations-web-legacy-compatibility-semantics>

> <https://tc39.es/ecma262/#sec-functiondeclarations-in-ifstatement-statement-clauses>

你仔细读读就好了。至于MDN，现在也有关于这个内容的条目，并且也指出在不同的浏览器环境中并不一致（因为是ECMAScript扩展规范，并不强制引擎实现）。在这里：

> [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function#Conditionally\\_created\\_functions](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function#Conditionally_created_functions)

共 3 条评论 >

👍 3



老师，从返回结果比较，“迭代器函数”和“生成器函数”的作用是一样的，都是通过迭代器或生成器的.next()方法，一次调用获取一个值。

作者回复: 严格来说，并没有所谓的“迭代器函数”。所谓“迭代”是一组接口规范，它约定的是一种行为。——在《JavaScript语言精髓与编程实践》中称“（可迭代对象是）一种可循环处理的‘可计算对象’”，并称“处理这种对象的‘行为’为迭代”。

而“生成器（函数）”是一个实现。



1



HoSalt

2020-05-19

「如果用户代码——通过任意的手段——试图挂起这惟一的执行上下文，那么也就意味着整个的 JavaScript 都停止了执行」

老师，这是什么意思？唯一的执行上下文指全局上下文？

作者回复: 不完全正确。

js的执行上下文是放在一个执行栈里面，而所谓的全局上下文只不过是在这个栈的栈底。上下文之间是通过类似arguments.caller的方式关联起来的（对的，在严格模式中沒有，但内部机制上还是有的）。

所以“挂起这惟一的执行上下文”更确切的说法是“活动的(active)上下文”。活动上下文是通过栈顶的移入移出操作来切换的，如果用户代码“通过别的方法”使得这个上下文挂起了，那么整个上下文的栈就挂起了，没有办法来激活它。

js里面所谓的“单线程”，本质上就是“所有的上下文中有且仅有一个活动的（active）上下文”。如果引擎中存在多个这样的活动上下文，那么要么是多线程的，要么是多引擎的，要么是多虚拟机（vm）的。总之这些活动上下文会被隔离在不同的“全局”里面，做不到真并行。

在js的单一线程中，所谓的“（一般意义上的）全局上下文”是惟一的，但它不一定是“活动的（active）”，当然活动的上下文可能在一个函数中（当前函数），或者一个被launch起来的promise的reactions（就是.then里的回调）中，等等。但无论是哪种情况，它们“所处的调用栈”的栈底，总会是一个全局环境（称为GlobalEnv，是用全局对象来作为词法环境的），但不一定是被称为“全局上下文”的那一个。——也就是说，所谓“一般意义上的全局上下文”，并不一定处于活动上下文（栈）的底部。



1



.Alter

2019-12-09

老师好，我想问一下生成器这个挂起和调用栈移动的机制是协程吗？

作者回复: 不是。

不过实现上，可以是。OS真实线程的切换成本高，在实现上用协程来做是可以的。但这与具体引擎的选择有关。另外按照ECMAScript的约定，这里自己实现一个上下文的管理器也是可以的，与线程什么的，并没有关系。

共 2 条评论 >

👍 2



阿鑫

2019-12-04

我的理解就是 `tor` 这个句柄其实就是包含了这个迭代器的一切，包括上下文 `context` 和执行函数。每次执行 `tor.next()` 就是把 `context` 压入栈顶，然后执行执行函数？

作者回复: 确实，ECMAScript就是这样做的。

如果`tor`是一个生成器对象，那么它就会有`[[GeneratorContext]]`这个私有槽，而`tor.next()`方法就是从这个私有槽中取出上下文给塞回到栈上。



👍 1



行问

2019-12-04

`x = yield x`

首先，`yield` 是向函数外发送 `x` 的值

其次，`yield` 接收外部传入的参数并赋值给 `x`

解惑了之前理解 `yield` 是一个“等待”的过程，没有往“挂起”去构思

时不时会用到 `async await` 来写并行的 `Promise`，但 `yield` 只知其知识点和应用，还没有开发中实际使用过

作者回复: `Promise`比`await`要难用一点，但其实深刻理解`promise`对整个的语言学习提升很大很大，因为它提供了一种新的理解程序执行逻辑的模式。不过这些内容，会放到20讲之后才讨论，这一次的课程中是不包括的。:)





静坐常思己过，闲谈莫...

2020-09-28

如果我能搞懂这些就不是初级前端了，学路漫漫感谢爱民老师

作者回复:        
谢谢      



潇潇雨歇

2019-12-04

先看看yield再来

