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。

```
// 迭代函数
function foo(x = 5) {
  return {
    next: () => {
      return {done: !x, value: x && x--};
    }
  }
}

let x = new Object;
x[Symbol.iterator] = foo; // default `x` is 5
console.log(...x);
```

事实上相当于只调用了5次return语句，可以展开如下：

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

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。

这样的一个函数就称为**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：

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

或者更通用的过程：

```
function *foo2(x=5) {
  while (x--) yield x;
}

// 测试
let x = new Object;
x[Symbol.iterator] = foo2; // default `x` is 5
console.log(...x); // 4 3 2 1 0
```

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。

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

```
function* foo3() {
  yield 10;
}
let tor = foo3();
...
```

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`yeild`。

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

这时，foo3()所声明的函数体正式执行，并直到表达式`yeild 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。

换而言之：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：

```
x = yield x
```

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：

```
function* foo4(x=5) {
  console.log(x--); // `tor = foo4()`时传入的值5
  // ...

  x = yield x; // 传出`x`的值
  console.log(x); // 传入的arg
  // ...
}

let tor = foo4(); // default `x` is 5
result = tor.next(); // 第一次调用.next()的参数将被忽略
console.log(result.value)
```

执行结果将显示：

```
5    // <- defaut `5`
4    // <- result.value `4`
```

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：

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

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“yeild*”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。

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。

```
// 迭代函数
function foo(x = 5) {
  return {
    next: () => {
      return {done: !x, value: x && x--};
    }
  }
}

let x = new Object;
x[Symbol.iterator] = foo; // default `x` is 5
console.log(...x);
```

事实上相当于只调用了5次return语句，可以展开如下：

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

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。

这样的一个函数就称为**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：

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

或者更通用的过程：

```
function *foo2(x=5) {
  while (x--) yield x;
}

// 测试
let x = new Object;
x[Symbol.iterator] = foo2; // default `x` is 5
console.log(...x); // 4 3 2 1 0
```

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。

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

```
function* foo3() {
  yield 10;
}
let tor = foo3();
...
```

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yeild。

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

这时，foo3()所声明的函数体正式执行，并直到表达式yeild 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。

换而言之：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：

```
x = yield x
```

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：

```
function* foo4(x=5) {
  console.log(x--); // `tor = foo4()`时传入的值5
  // ...

  x = yield x; // 传出`x`的值
  console.log(x); // 传入的arg
  // ...
}
```

```
let tor = foo4(); // default `x` is 5
result = tor.next(); // 第一次调用.next()的参数将被忽略
console.log(result.value)
```

执行结果将显示：

```
5    // <- defaut `5`
4    // <- result.value `4`
```

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：

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

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“yeild*”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。

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。

```
// 迭代函数
function foo(x = 5) {
  return {
    next: () => {
      return {done: !x, value: x && x--};
    }
  }
}

let x = new Object;
x[Symbol.iterator] = foo; // default `x` is 5
console.log(...x);
```

事实上相当于只调用了5次return语句，可以展开如下：

```
// 上例在形式上可以表达为如下的逻辑
console.log(
```

```
  /*return */{done: false, value: 5}.value,
  /*return */{done: false, value: 4}.value,
  /*return */{done: false, value: 3}.value,
  /*return */{done: false, value: 2}.value,
  /*return */{done: false, value: 1}.value
);
```

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。

这样的一个函数就称为**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：

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

或者更通用的过程：

```
function *foo2(x=5) {
  while (x--) yield x;
}

// 测试
let x = new Object;
x[Symbol.iterator] = foo2; // default `x` is 5
console.log(...x); // 4 3 2 1 0
```

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。

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

```
function* foo3() {
  yield 10;
}
let tor = foo3();
...
```

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yeild。

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

这时，foo3()所声明的函数体正式执行，并直到表达式`yeild 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。

换而言之：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：

```
x = yield x
```

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：

```
function* foo4(x=5) {
  console.log(x--); // `tor = foo4()`时传入的值5
  // ...

  x = yield x; // 传出`x`的值
  console.log(x); // 传入的arg
  // ...
}

let tor = foo4(); // default `x` is 5
result = tor.next(); // 第一次调用.next()的参数将被忽略
console.log(result.value)
```

执行结果将显示：

```
5    // <- defaut `5`
4    // <- result.value `4`
```

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：

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

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“yeild*”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。

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。

```
// 迭代函数
function foo(x = 5) {
  return {
    next: () => {
      return {done: !x, value: x && x--};
    }
  }
}

let x = new Object;
x[Symbol.iterator] = foo; // default `x` is 5
console.log(...x);
```

事实上相当于只调用了5次return语句，可以展开如下：

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

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。

这样的一个函数就称为**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：

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

或者更通用的过程：

```
function *foo2(x=5) {
  while (x--) yield x;
}

// 测试
let x = new Object;
x[Symbol.iterator] = foo2; // default `x` is 5
console.log(...x); // 4 3 2 1 0
```

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。

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

```
function* foo3() {
  yield 10;
}
let tor = foo3();
...
```

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`yeild`。

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

这时，foo3()所声明的函数体正式执行，并直到表达式`yeild 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。

换而言之：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：

```
x = yield x
```

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：

```
function* foo4(x=5) {
  console.log(x--); // `tor = foo4()`时传入的值5
  // ...

  x = yield x; // 传出`x`的值
  console.log(x); // 传入的arg
  // ...
}

let tor = foo4(); // default `x` is 5
result = tor.next(); // 第一次调用.next()的参数将被忽略
console.log(result.value)
```

执行结果将显示：

```
5    // <- defaut `5`
4    // <- result.value `4`
```

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：

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

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“yeild*”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。

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。

```
// 迭代函数
function foo(x = 5) {
  return {
    next: () => {
      return {done: !x, value: x && x--};
    }
  }
}

let x = new Object;
x[Symbol.iterator] = foo; // default `x` is 5
console.log(...x);
```

事实上相当于只调用了5次return语句，可以展开如下：

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

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。

这样的一个函数就称为**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：

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

或者更通用的过程：

```
function *foo2(x=5) {
  while (x--) yield x;
}

// 测试
```

```
let x = new Object;
x[Symbol.iterator] = foo2; // default `x` is 5
console.log(...x); // 4 3 2 1 0
```

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。

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

```
function* foo3() {
  yield 10;
}
let tor = foo3();
...
```

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`yeild`。

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

这时，foo3()所声明的函数体正式执行，并直到表达式`yeild 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。

换而言之：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：

```
x = yield x
```

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：

```
function* foo4(x=5) {
  console.log(x--); // `tor = foo4()`时传入的值5
  // ...

  x = yield x; // 传出`x`的值
  console.log(x); // 传入的arg
  // ...
}

let tor = foo4(); // default `x` is 5
result = tor.next(); // 第一次调用.next()的参数将被忽略
console.log(result.value)
```

执行结果将显示：

```
5    // <- defaut `5`
4    // <- result.value `4`
```

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：

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

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“yeild*”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。

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。

```
// 迭代函数
```

```
function foo(x = 5) {
  return {
    next: () => {
      return {done: !x, value: x && x--};
    }
  }
}

let x = new Object;
x[Symbol.iterator] = foo; // default `x` is 5
console.log(...x);
```

事实上相当于只调用了5次return语句，可以展开如下：

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

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。

这样的一个函数就称为**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：

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

或者更通用的过程：

```
function *foo2(x=5) {
  while (x--) yield x;
}

// 测试
let x = new Object;
x[Symbol.iterator] = foo2; // default `x` is 5
console.log(...x); // 4 3 2 1 0
```

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。

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

```
function* foo3() {
  yield 10;
}
let tor = foo3();
...
```

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yeild。

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

这时，foo3()所声明的函数体正式执行，并直到表达式yeild 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。

换而言之：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：

```
x = yield x
```

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：

```
function* foo4(x=5) {
  console.log(x--); // `tor = foo4()`时传入的值5
  // ...

  x = yield x; // 传出`x`的值
  console.log(x); // 传入的arg
  // ...
}

let tor = foo4(); // default `x` is 5
result = tor.next(); // 第一次调用.next()的参数将被忽略
console.log(result.value)
```

执行结果将显示：

```
5    // <- defaut `5`
4    // <- result.value `4`
```

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：

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

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“yeild*”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。

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。

```
// 迭代函数
function foo(x = 5) {
  return {
    next: () => {
      return {done: !x, value: x && x--};
    }
  }
}

let x = new Object;
x[Symbol.iterator] = foo; // default `x` is 5
console.log(...x);
```

事实上相当于只调用了5次return语句，可以展开如下：

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

```
);
```

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。

这样的一个函数就称为**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：

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

或者更通用的过程：

```
function *foo2(x=5) {
  while (x--) yield x;
}

// 测试
let x = new Object;
x[Symbol.iterator] = foo2; // default `x` is 5
console.log(...x); // 4 3 2 1 0
```

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。

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

```
function* foo3() {
  yield 10;
}
let tor = foo3();
...
```

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yeild。

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

这时，foo3()所声明的函数体正式执行，并直到表达式yeild 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。

换而言之：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：

```
x = yield x
```

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：

```
function* foo4(x=5) {
  console.log(x--); // `tor = foo4()`时传入的值5
  // ...

  x = yield x; // 传出`x`的值
  console.log(x); // 传入的arg
  // ...
}

let tor = foo4(); // default `x` is 5
result = tor.next(); // 第一次调用.next()的参数将被忽略
console.log(result.value)
```

执行结果将显示：

```
5    // <- defaut `5`
4    // <- result.value `4`
```

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：

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

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“yeild*”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。

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。

```
// 迭代函数
function foo(x = 5) {
  return {
    next: () => {
      return {done: !x, value: x && x--};
    }
  }
}

let x = new Object;
x[Symbol.iterator] = foo; // default `x` is 5
console.log(...x);
```

事实上相当于只调用了5次return语句，可以展开如下：

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

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。

这样的一个函数就称为**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：

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

或者更通用的过程：

```
function *foo2(x=5) {
  while (x--) yield x;
}

// 测试
let x = new Object;
x[Symbol.iterator] = foo2; // default `x` is 5
console.log(...x); // 4 3 2 1 0
```

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。

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

```
function* foo3() {
  yield 10;
}
let tor = foo3();
...
```

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`yeild`。

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

这时，foo3()所声明的函数体正式执行，并直到表达式`yeild 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。

换而言之：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：

```
x = yield x
```

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：

```
function* foo4(x=5) {
  console.log(x--); // `tor = foo4()`时传入的值5
  // ...

  x = yield x; // 传出`x`的值
  console.log(x); // 传入的arg
  // ...
}

let tor = foo4(); // default `x` is 5
result = tor.next(); // 第一次调用.next()的参数将被忽略
console.log(result.value)
```

执行结果将显示：

```
5    // <- defaut `5`
4    // <- result.value `4`
```

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：

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

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“yeild*”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。

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。

```
// 迭代函数
function foo(x = 5) {
  return {
    next: () => {
      return {done: !x, value: x && x--};
    }
  }
}

let x = new Object;
x[Symbol.iterator] = foo; // default `x` is 5
console.log(...x);
```

事实上相当于只调用了5次return语句，可以展开如下：

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

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。

这样的一个函数就称为**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：

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

或者更通用的过程：

```
function *foo2(x=5) {
  while (x--) yield x;
}

// 测试
let x = new Object;
x[Symbol.iterator] = foo2; // default `x` is 5
```

```
console.log(...x); // 4 3 2 1 0
```

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。

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

```
function* foo3() {
  yield 10;
}
let tor = foo3();
...
```

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`yeild`。

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

这时，foo3()所声明的函数体正式执行，并直到表达式`yeild 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。

换而言之：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：

```
x = yield x
```

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：

```
function* foo4(x=5) {
  console.log(x--); // `tor = foo4()`时传入的值5
  // ...

  x = yield x; // 传出`x`的值
  console.log(x); // 传入的arg
  // ...
}

let tor = foo4(); // default `x` is 5
result = tor.next(); // 第一次调用.next()的参数将被忽略
console.log(result.value)
```

执行结果将显示：

```
5    // <- defaut `5`
4    // <- result.value `4`
```

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：

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

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“yeild*”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。

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。

```
// 迭代函数
```

```
function foo(x = 5) {
  return {
    next: () => {
      return {done: !x, value: x && x--};
    }
  }
}

let x = new Object;
x[Symbol.iterator] = foo; // default `x` is 5
console.log(...x);
```

事实上相当于只调用了5次return语句，可以展开如下：

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

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。

这样的一个函数就称为**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：

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

或者更通用的过程：

```
function *foo2(x=5) {
  while (x--) yield x;
}

// 测试
let x = new Object;
x[Symbol.iterator] = foo2; // default `x` is 5
console.log(...x); // 4 3 2 1 0
```

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。

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

```
function* foo3() {
  yield 10;
}
let tor = foo3();
...
```

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yeild。

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

这时，foo3()所声明的函数体正式执行，并直到表达式yeild 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。

换而言之：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：

```
x = yield x
```

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：

```
function* foo4(x=5) {
  console.log(x--); // `tor = foo4()`时传入的值5
  // ...

  x = yield x; // 传出`x`的值
  console.log(x); // 传入的arg
  // ...
}

let tor = foo4(); // default `x` is 5
result = tor.next(); // 第一次调用.next()的参数将被忽略
console.log(result.value)
```

执行结果将显示：

```
5    // <- defaut `5`
4    // <- result.value `4`
```

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：

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

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“yeild*”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。

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。

```
// 迭代函数
function foo(x = 5) {
  return {
    next: () => {
      return {done: !x, value: x && x--};
    }
  }
}

let x = new Object;
x[Symbol.iterator] = foo; // default `x` is 5
console.log(...x);
```

事实上相当于只调用了5次return语句，可以展开如下：

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

```
);
```

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。

这样的一个函数就称为**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：

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

或者更通用的过程：

```
function *foo2(x=5) {
  while (x--) yield x;
}

// 测试
let x = new Object;
x[Symbol.iterator] = foo2; // default `x` is 5
console.log(...x); // 4 3 2 1 0
```

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。

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

```
function* foo3() {
  yield 10;
}
let tor = foo3();
...
```

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yeild。

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

这时，foo3()所声明的函数体正式执行，并直到表达式`yeild 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。

换而言之：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：

```
x = yield x
```

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：

```
function* foo4(x=5) {
  console.log(x--); // `tor = foo4()`时传入的值5
  // ...

  x = yield x; // 传出`x`的值
  console.log(x); // 传入的arg
  // ...
}

let tor = foo4(); // default `x` is 5
result = tor.next(); // 第一次调用.next()的参数将被忽略
console.log(result.value)
```

执行结果将显示：

```
5    // <- defaut `5`
4    // <- result.value `4`
```

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：

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

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“yeild*”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。

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。

```
// 迭代函数
function foo(x = 5) {
  return {
    next: () => {
      return {done: !x, value: x && x--};
    }
  }
}

let x = new Object;
x[Symbol.iterator] = foo; // default `x` is 5
console.log(...x);
```

事实上相当于只调用了5次return语句，可以展开如下：

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

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。

这样的一个函数就称为**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：

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

或者更通用的过程：

```
function *foo2(x=5) {
  while (x--) yield x;
}

// 测试
let x = new Object;
x[Symbol.iterator] = foo2; // default `x` is 5
console.log(...x); // 4 3 2 1 0
```

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。

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

```
function* foo3() {
  yield 10;
}
let tor = foo3();
...
```

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`yeild`。

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

这时，foo3()所声明的函数体正式执行，并直到表达式`yeild 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。

换而言之：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：

```
x = yield x
```

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：

```
function* foo4(x=5) {
  console.log(x--); // `tor = foo4()`时传入的值5
  // ...

  x = yield x; // 传出`x`的值
  console.log(x); // 传入的arg
  // ...
}

let tor = foo4(); // default `x` is 5
result = tor.next(); // 第一次调用.next()的参数将被忽略
console.log(result.value)
```

执行结果将显示：

```
5    // <- defaut `5`
4    // <- result.value `4`
```

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：

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

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“yeild*”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。

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。

```
// 迭代函数
function foo(x = 5) {
  return {
    next: () => {
      return {done: !x, value: x && x--};
    }
  }
}

let x = new Object;
x[Symbol.iterator] = foo; // default `x` is 5
console.log(...x);
```

事实上相当于只调用了5次return语句，可以展开如下：

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

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。

这样的一个函数就称为**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：

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

或者更通用的过程：

```
function *foo2(x=5) {
  while (x--) yield x;
}

// 测试
let x = new Object;
x[Symbol.iterator] = foo2; // default `x` is 5
```

```
console.log(...x); // 4 3 2 1 0
```

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。

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

```
function* foo3() {
  yield 10;
}
let tor = foo3();
...
```

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`yeild`。

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

这时，foo3()所声明的函数体正式执行，并直到表达式`yeild 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。

换而言之：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：

```
x = yield x
```

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：

```
function* foo4(x=5) {
  console.log(x--); // `tor = foo4()`时传入的值5
  // ...

  x = yield x; // 传出`x`的值
  console.log(x); // 传入的arg
  // ...
}

let tor = foo4(); // default `x` is 5
result = tor.next(); // 第一次调用.next()的参数将被忽略
console.log(result.value)
```

执行结果将显示：

```
5    // <- defaut `5`
4    // <- result.value `4`
```

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：

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

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“yeild*”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。

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。

```
// 迭代函数
```

```
function foo(x = 5) {
  return {
    next: () => {
      return {done: !x, value: x && x--};
    }
  }
}

let x = new Object;
x[Symbol.iterator] = foo; // default `x` is 5
console.log(...x);
```

事实上相当于只调用了5次return语句，可以展开如下：

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

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。

这样的一个函数就称为**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：

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

或者更通用的过程：

```
function *foo2(x=5) {
  while (x--) yield x;
}

// 测试
let x = new Object;
x[Symbol.iterator] = foo2; // default `x` is 5
console.log(...x); // 4 3 2 1 0
```

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。

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

```
function* foo3() {
  yield 10;
}
let tor = foo3();
...
```

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yeild。

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

这时，foo3()所声明的函数体正式执行，并直到表达式yeild 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。

换而言之：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：

```
x = yield x
```

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：

```
function* foo4(x=5) {
  console.log(x--); // `tor = foo4()`时传入的值5
  // ...

  x = yield x; // 传出`x`的值
  console.log(x); // 传入的arg
  // ...
}

let tor = foo4(); // default `x` is 5
result = tor.next(); // 第一次调用.next()的参数将被忽略
console.log(result.value)
```

执行结果将显示：

```
5    // <- defaut `5`
4    // <- result.value `4`
```

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：

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

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“yeild*”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。

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。

```
// 迭代函数
function foo(x = 5) {
  return {
    next: () => {
      return {done: !x, value: x && x--};
    }
  }
}

let x = new Object;
x[Symbol.iterator] = foo; // default `x` is 5
console.log(...x);
```

事实上相当于只调用了5次return语句，可以展开如下：

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

```
);
```

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。

这样的一个函数就称为**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：

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

或者更通用的过程：

```
function *foo2(x=5) {
  while (x--) yield x;
}

// 测试
let x = new Object;
x[Symbol.iterator] = foo2; // default `x` is 5
console.log(...x); // 4 3 2 1 0
```

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。

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

```
function* foo3() {
  yield 10;
}
let tor = foo3();
...
```

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yeild。

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

这时，foo3()所声明的函数体正式执行，并直到表达式`yeild 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。

换而言之：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：

```
x = yield x
```

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：

```
function* foo4(x=5) {
  console.log(x--); // `tor = foo4()`时传入的值5
  // ...

  x = yield x; // 传出`x`的值
  console.log(x); // 传入的arg
  // ...
}

let tor = foo4(); // default `x` is 5
result = tor.next(); // 第一次调用.next()的参数将被忽略
console.log(result.value)
```

执行结果将显示：

```
5    // <- defaut `5`
4    // <- result.value `4`
```

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：

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

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“yeild*”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。

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。

```
// 迭代函数
function foo(x = 5) {
  return {
    next: () => {
      return {done: !x, value: x && x--};
    }
  }
}

let x = new Object;
x[Symbol.iterator] = foo; // default `x` is 5
console.log(...x);
```

事实上相当于只调用了5次return语句，可以展开如下：

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

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。

这样的一个函数就称为**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：

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

或者更通用的过程：

```
function *foo2(x=5) {
  while (x--) yield x;
}

// 测试
let x = new Object;
x[Symbol.iterator] = foo2; // default `x` is 5
console.log(...x); // 4 3 2 1 0
```

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。

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

```
function* foo3() {
  yield 10;
}
let tor = foo3();
...
```

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`yeild`。

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

这时，foo3()所声明的函数体正式执行，并直到表达式`yeild 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。

换而言之：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：

```
x = yield x
```

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：

```
function* foo4(x=5) {
  console.log(x--); // `tor = foo4()`时传入的值5
  // ...

  x = yield x; // 传出`x`的值
  console.log(x); // 传入的arg
  // ...
}

let tor = foo4(); // default `x` is 5
result = tor.next(); // 第一次调用.next()的参数将被忽略
console.log(result.value)
```

执行结果将显示：

```
5    // <- defaut `5`
4    // <- result.value `4`
```

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：

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

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“yeild*”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。