链滴

深入浅出ES6（三）：生成器 Generators

作者：Vanessa
原文链接：https://ld246.com/article/1436407882048
来源网站：链滴

今天的这篇文章令我感到非常兴奋，我们将一起领略ES6中最具魔力的特性。
为什么说是“最具魔力的”？对于初学者来说，此特性与JS之前已有的特性截然不同，可能会觉有点晦涩难懂。但是，从某种意义上来说，它使语言内部的常态行为变得更加强大，如果这都不算有力，我不知道还有什么能算。
不仅如此，此特性可以极大地简化代码，它甚至可以帮助你逃离“回调地狱”。
既然新特性如此神奇，那么就一起深入了解它的魔力吧！
<h2 id="toc_h2_0">ES6生成器（Generators）简介</h2>
什么是生成器？
我们从一个示例开始：
<pre class=" language-javascript"><code class=" language-javascript">function</spn>* quips(name) {
  yield "你好 " + name + <san>"!";
  yield "希望你能喜欢这篇介绍ES6的译文";
  if (name.startsWith(</spa>"X")) {
    yield "你的名字 " + name +</spa> " 首字母是X，这很酷！";
  }
  yield "我们下次再见！";
}</code></pre>
这是一只<a href="https://ld246.com/forward?goto=http%3A%2F%2Fpeople.mozilla.org2F%7Ejorendorff%2Fdemos%2Fmeow.html" target="_blank" rel="nofollow ugc">会说话的猫<a>，这段代码很可能代表着当今互联网上最重要的一类应用。（试着点击这个链接，与这只猫互动一下，如果你感到有些惑，回到这里继续阅读）。
这段代码看起来很像一个函数，我们称之为生成器函数，它与普通函数有很多共同点，但是二者如下区别：

普通函数使用function声明，而生成器函数使用function*声明。
在生成器函数内部，有一种类似return的语法：关键字yield。二者的区别是，普通函数只可以retrn一次，而生成器函数可以yield多次（当然也可以只yield一次）。在生成器的执行过程中，遇到yiel表达式立即暂停，后续可恢复执行状态。

这就是普通函数和生成器函数之间最大的区别，普通函数不能自暂停，生成器函数可以。
<h2 id="toc_h2_1">生成器做了什么？</h2>
当你调用quips()生成器函数时发生了什么？
<pre><code class="language-nolanguage">> var iter = quips("jorendorff");
  [object Generator]
> iter.next()
  { value: "你好 jorendorff!", done: false }
> iter.next()
  { value: "希望你能喜欢这篇介绍ES6的译文", done: false }
> iter.next()
  { value: "我们下次再见！", done: false }
> iter.next()
  { value: undefined, done: true }</code></pre>
你大概已经习惯了普通函数的使用方式，当你调用它们时，它们立即开始运行，直到遇到return抛出异常时才退出执行，作为JS程序员你一定深谙此道。
生成器调用看起来非常类似：quips("jorendorff")。但是，当你调用一个生成器时，它并非立即行，而是返回一个已暂停的生成器对象（上述实例代码中的iter）。你可将这个生成器对象视为一次数调用，只不过立即冻结了，它恰好在生成器函数的最顶端的第一行代码之前冻结了。

每当你调用生成器对象的.next()方法时，函数调用将其自身解冻并一直运行到下一个yield表达式再次暂停。
这也是在上述代码中我们每次都调用iter.next()的原因，我们获得了quips()函数体中yield表达式成的不同的字符串值。
调用最后一个iter.next()时，我们最终抵达生成器函数的末尾，所以返回结果中done的值为true抵达函数的末尾意味着没有返回值，所以返回结果中value的值为undefined。
现在回到会说话的猫demo页面，尝试在循环中加入一个yield，会发生什么？
如果用专业术语描述，每当生成器执行yields语句，生成器的堆栈结构（本地变量、参数、临时、生成器内部当前的执行位置）被移出堆栈。然而，生成器对象保留了对这个堆栈结构的引用（备份，所以稍后调用.next()可以重新激活堆栈结构并且继续执行。
值得特别一提的是，生成器不是线程，在支持线程的语言中，多段代码可同时运行，通通常导致竞态条件和非确定性，不过同时也带来不错的性能。生成器则完全不同。当生器运行时，它和调用者处于同一线程中，拥有确定的连续执行顺序，永不并发。与系统线程不同的是生成器只有在其函数体内标记为yield的点才会暂停。
现在，我们了解了生成器的原理，领略过生成器的运行、暂停恢复运行的不同状态。那么，这些怪的功能究竟有何用处？
<h2 id="toc_h2_2">生成器是迭代器！</h2>
上周，我们学习了ES6的迭代器，它是ES6中独立的内建类，同时也是语言的一个扩展点，通过现[Symbol.iterator]()和.next()两个方法你就可以创建自定义迭代器。
实现一个接口不是一桩小事，我们一起实现一个迭代器。举个例子，我们创建一个简单的range代器，它可以简单地将两个数字之间的所有数相加。首先是传统C的for(;;)循环：
<pre class=" language-javascript"><code class=" language-javascript">// 应该弹出次 "ding"
for (var value of range<spa>(0, 3)<san>) {
  alert("Ding! at floor #" + vlue);
}</code></pre>
使用ES6的类的解决方案（如果不清楚语法细节，无须担心，我们将在接下来的文章中为你讲解：
<pre class=" language-javascript"><code class=" language-javascript">class RangeIterator {
  constructor(start, stop) <san>{
    this.value = start;
    this.stop = stop;
  }
[Symbol.iterator](<spa>) { return this; <san>}
next() {
var value = this.value<spa>;
if (value < this.</pan>stop) {
this.value++;
return {done: false,/span> value: value};
} else {
return {done: true, value: undefined};

}
}
}
</code> <code class=" language-javascript">// 返回一个新的迭代器，可以从 start 到 stop 计数。
function range(start,</spa> stop) {
return new RangeIterator(start, stop);
}</code> </pre>

查看代码运行情况。
这里的实现类似Java或Swift中的迭代器，不是很糟糕，但也不是完全没有问题。我们很难说清这段代码中是否有bu，这段代码看起来完全不像我们试图模仿的传统for (;;)循环，迭代器协议迫使我们拆解掉循环部分。
此时此刻你对迭代器可能尚无感觉，他们用起来很酷，但看起来有些难以实现。
你大概不会为了使迭代器更易于构建从而建议我们为JS语言引入一个离奇古怪又野蛮的新型控制结构，但是既然我们有生成器，是否可以在这里应用它们呢？一起尝试一下：

<pre class=" language-javascript"><code class=" language-javascript">function</spn>* range(start, stop<spa>) {
  for (var i = start; i < stop; i++)
    yield i;
}</code></pre>

查看代码运行情况。
以上4行代码实现的生成器完全可以替代之前引入了一整个RangeIterator类的23行代码的实现可行的原因是：生成器是迭代器。所有的生成器都有内建.next()和[Symbol.iterator]()方法的实现。你只须编写循环部分的行为。
我们都非常讨厌被迫用被动语态写一封很长的邮件，不借助生成器实现迭代器的过程与之类似，人痛苦不堪。当你的语言不再简练，说出的话就会变得难以理解。RangeIterator的实现代码很长并非常奇怪，因为你需要在不借助循环语法的前提下为它添加循环功能的描述。所以生成器是最好的解方案！
我们如何发挥作为迭代器的生成器所产生的最大效力？
l 使任意对象可迭代。编写生成器函数遍历这个对象，运行时yield每一个值。然后将这个生成器数作为这个对象的[Symbol.iterator]方法。
l 简化数组构建函数。假设你有一个函数，每次调用的时候返回一个数组结果，就像这样：

<pre class=" language-javascript"><code class=" language-javascript">// 拆分一维组icons
// 根据长度rowLength
function splitIntoRows(icons, rowLength) {
  var rows = [];
  for (var i = 0</spn>; i < icons.length; i <san>+= rowLength) {
    rows.push(icons.sice(i, i + rowLength)</spa

```
><span>)</span><span>;</span>
  <span>}</span>
  <span>return</span> rows<span>;</span>
<span>}</span></code></pre>
<p>使用生成器创建的代码相对较短：</p>
<pre class=" language-javascript"><code class=" language-javascript"><span>function</sp
n><span>*</span> <span>splitIntoRows<span>(</span></span>icons<span>,</span> r
wLength<span>)</span> <span>{</span>
  <span>for</span> <span>(</span><span>var</span> i <span>=</span> <span>0</sp
n><span>;</span> i <span>&lt;</span> icons<span>.</span>length<span>;</span> i <s
an>+</span><span>=</span> rowLength<span>)</span> <span>{</span>
    <span>yield</span> icons<span>.</span><span>slice<span>(</span></span>i<span>
</span> i <span>+</span> rowLength<span>)</span><span>;</span>
  <span>}</span>
<span>}</span></code></pre>
<p>行为上唯一的不同是，传统写法立即计算所有结果并返回一个数组类型的结果，使用生成器则返
一个迭代器，每次根据需要逐一地计算结果。</p>
<ul>
<li>获取异常尺寸的结果。你无法构建一个无限大的数组，但是你可以返回一个可以生成一个永无止
的序列的生成器，每次调用可以从中取任意数量的值。</li>
<li>重构复杂循环。你是否写过又丑又大的函数？你是否愿意将其拆分为两个更简单的部分？现在，
的重构工具箱里有了新的利刃——生成器。当你面对一个复杂的循环时，你可以拆分出生成数据的代
，将其转换为独立的生成器函数，然后使用for (var data of myNewGenerator(args))遍历我们所需
数据。</li>
<li>构建与迭代相关的工具。ES6不提供用来过滤、映射以及针对任意可迭代数据集进行特殊操作的
展库。借助生成器，我们只须写几行代码就可以实现类似的工具。</li>
</ul>
<p>举个例子，假设你需要一个等效于Array.prototype.filter并且支持DOM NodeLists的方法，可
这样写：</p>
<pre class=" language-javascript"><code class=" language-javascript"><span>function</sp
n><span>*</span> <span>filter<span>(</span></span>test<span>,</span> iterable<spa
>)</span> <span>{</span>
  <span>for</span> <span>(</span><span>var</span> item of iterable<span>)</span> <
pan>{</span>
    <span>if</span> <span>(</span><span>test<span>(</span></span>item<span>)</sp
n><span>)</span>
      <span>yield</span> item<span>;</span>
  <span>}</span>
<span>}</span></code></pre>
<p>你看，生成器魔力四射！借助它们的力量可以非常轻松地实现自定义迭代器，记住，迭代器贯穿E
6的始终，它是数据和循环的新标准。</p>
<p>以上只是生成器的冰山一角，最重要的功能请继续观看！</p>
<h2 id="toc_h2_3">生成器和异步代码</h2>
<p>这是我在一段时间以前写的一些JS代码</p>
<pre><code class="language-nolanguage">        };
        })
      });
    });
  });
});</code></pre>
<p>可能你已经在自己的代码中见过类似的片段，<a href="https://ld246.com/forward?goto=htt
%3A%2F%2Fwww.html5rocks.com%2Fen%2Ftutorials%2Fasync%2Fdeferred%2F" target="_bl
nk" rel="nofollow ugc">异步API</a>通常需要一个回调函数，这意味着你需要为每一次任务执行
写额外的异步函数。所以如果你有一段代码需要完成三个任务，你将看到类似的三层级缩进的代码，
```

非简单的三行代码。
后来我就这样写了：
<pre class=" language-javascript"><code class=" language-javascript">}<spn>).on('close'<spa>, function () {
done(undefined, undefined)</spa>;
}).on(<spa>'error', function (error)</pan> {
done(error);
});</code></pre>
异步API拥有错误处理规则，不支持异常处理。不同的API有不同的规则，大多数的错误规则是默的；在有些API里，甚至连成功提示都是默认的。
这些是到目前为止我们为异步编程所付出的代价，我们正慢慢开始接受异步代码不如等效同步代美观又简洁的这个事实。
生成器为你提供了避免以上问题的新思路。
实验性的Q.async()尝试结合promises使用生成器产生异步代码的等效同步代码。举个例子：
<pre class=" language-javascript"><code class=" language-javascript">// 制造一些音的同步代码。
function makeNoise()</spn> {
shake();
rattle();
roll();
}
</code> <code class=" language-javascript">// 制造一些噪音的异步代码。
// 返回一个 Promise 对象
// 当我们制造完噪音的时候会变为 resolved
function makeNoise_async(
 {
return Q.async(function* () {
yield shake_async(

yield rattle_async(/span>
yield roll_async(pan>
});
}</code> </pre>
二者主要的区别是，异步版本必须在每次调用异步函数的地方添加yield关键字。
在Q.async版本中添加一个类似if语句的判断或try/catch块，如同向同步版本中添加类似功能一简单。与其它异步代码编写方法相比，这种方法更自然，不像是学一门新语言一样辛苦。
如果你已经看到这里，你可以试着阅读来自James Long的更深入地讲解生成器的文章。
生成器为我们提供了一个新的异步编程模型思路，这种方法更适合人类的大脑。相关工作正在不展开。此外，更好的语法或许会有帮助，ES7中有一有关异步函数的提案，它基于pr

mises和生成器构建，并从C#相似的特性中汲取了大量灵感。
<h2 id="toc_h2_4">如何应用这些疯狂的新特性？</h2>
在服务器端，现在你可以在io.js中使用ES6（在Node中你需要使用--harmony这个命令行选项。
在浏览器端，到目前为止只有Firefox 27+和Chrome 39+支持了ES6生成器。如果要在web端使生成器，你需要使用Babel或Traceur来将你的ES6代码转译为Web友好的ES5。
起初，JS中的生成器由Brendan Eich实现，他的设计参考了Python生成器，而此Python生成器则受到Icon的启发。他们早在2006年就在Firefox 2.0中移植了相关代码。但是，准化的道路崎岖不平，相关语法和行为都在原先的基础上有所改动。Firefox和Chrome中的ES6生成都是由编译器hacker Andy Wingo实现的。这项工作由<a href "https://ld246.com/forward?goto=http%3A%2F%2Fwww.bloomberg.com%2F" target="_blank rel="nofollow ugc">彭博赞助支持（没听错，就是大名鼎鼎的那个彭博！）。
<h2 id="toc_h2_5">yield;</h2>
生成器还有更多未提及的特性，例如：.throw()和.return()方法、可选参数.next()、yield*表达式法。由于行文过长，估计观众老爷们已然疲乏，我们应该学习一下生成器，暂时yield在这里，剩下的货择机为大家献上。
下一次，我们变换一下风格，由于我们接连搬了两座大山：迭代器和生成器，下次就一起研究下会改变你编程风格的ES6特性好不？就是一些简单又实用的东西，你一定会喜笑颜开哒！你还别说，什么都要“微”一下的今天，ES6当然要有微改进了！
下回预告：ES6模板字符串深度解析，每天都会写的代码！观众老爷们记得回来哦！我会想你们！