

## 19 | 打开潘多拉盒子：JavaScript异步编程

2019-10-23 四火

全栈工程师修炼指南

[进入课程 >](#)



讲述：四火

时长 22:46 大小 15.64M



你好，我是四火。

我们在本章伊始的 [\[第 14 讲\]](#) 中初步学习了 JavaScript 的事件驱动模型，体会到了思维模式的转变，也建立起了异步编程的初步概念。在本章最后一讲，我们将深入异步编程，继续探讨其中的关键技术。

异步编程就像是一个神秘的宝盒，看起来晶莹剔透，可一旦使用不当，就会是带来灾难的潘多拉盒子，状态混乱，难以维护。希望在这一讲之后，你可以了解更多的关于 JavaScript 在异步编程方面的高级特性，从而习惯并写出可靠的异步代码。

### 1. 用 Promise 优化嵌套回调

假如我们需要写这样一段代码，来模拟一只小狗向前奔跑，它一共跑了 3 次，奔跑的距离分别为 1、2、3，每次奔跑都要花费 1 秒钟时间：

 复制代码

```
1  setTimeout(  
2    () => {  
3      console.log(1);  
4      setTimeout(  
5        () => {  
6          console.log(2);  
7          setTimeout(  
8            () => {  
9              console.log(3);  
10             },  
11             1000  
12           );  
13         },  
14         1000  
15       );  
16     },  
17     1000  
18   );
```

你看，我们用了 3 次 `setTimeout`，每次都接受两个参数，第一个参数是一个函数，用以打印当前跑的距离，以及递归调用奔跑逻辑，第二个参数用于模拟奔跑耗时 1000 毫秒。这个问题其实代表了实际编程中一类很常见的 JavaScript 异步编程问题。例如，使用 Ajax 方式异步获取一个请求，在得到返回的结果后，再执行另一个 Ajax 操作。

现在，请你打开 Chrome 开发者工具中的控制台，运行一下：

 复制代码

```
1  3693  
2  1  
3  2  
4  3
```

第一行是 `setTimeout` 返回的句柄，由于控制台运行的关系，系统会把最后一行执行的返回值打印出来，因此它可以忽略。除此之外，结果恰如预期，每一行的打印都间隔了一秒，模拟了奔跑的效果。

但是，这个代码似乎不太“好看”啊，繁琐而且冗长，易理解性和可维护性显然不过关，代码的状态量在这种情况下很难预测和维护。就如同同步编程世界中常见的“[🍝 面条代码](#)（Spaghetti Code）”一样，这样“坏味道”的代码在异步编程的世界中其实也很常见，且也有个专有称呼——“金字塔厄运”（Pyramid of Doom，嵌套结构就像金字塔一样）。

到这里，不知你会不会想，能不能把重复的逻辑抽取出来呢？具体说，就是这个 `setTimeout` 方法相关的代码。于是，我们可以抽取公用逻辑，定义一个 `run` 方法，接受两个参数，一个是当前跑动距离，第二个是回调方法，用于当前跑完以后，触发下一次跑动的行为：

 复制代码


```
1 var run = (steps, callback) => {
2   setTimeout(
3     () => {
4       console.log(steps);
5       callback();
6     },
7     1000
8   );
9 };
10
11 run(1, () => {
12   run(2, () => {
13     run(3, () => {});
14   });
15 });
```

嗯，代码确实清爽多了。可是，看着这嵌套的三个 `run`，我觉得这并没有从本质上解决问题，只是代码简短了些，嵌套调用依然存在。

每当我们开始写这样反复嵌套回调的代码时，我们就应该警醒，我们是否在创造一个代码维护上的坑。那能不能使用某一种优雅的方式来解决这个问题呢？

有！它就是 `Promise`，并且从 ES6 开始，JavaScript 原生支持，不再需要第三方的库或者自己实现的工具类了。

**Promise**，就如同字面意思“承诺”一样，定义在当前，但行为发生于未来。它的构造方法中接受一个函数（如果你对这种将函数作为参数的方式传入还不习惯，请回看 [🔗\[第 15 讲\]](#) 对函数成为一等公民的介绍），并且这个函数接受 `resolve` 和 `reject` 两个参数，前者在未来的执行成功时会被调用，后者在未来的执行失败时会被调用。

 复制代码

```
1 var run = steps =>
2   () =>
3     new Promise((resolve, reject) => {
4       setTimeout(
5         () => {
6           console.log(steps);
7           resolve(); // 一秒后的未来执行成功，需要调用
8         },
9         1000
10      );
11    });
12
13 Promise.resolve()
14   .then(run(1))
15   .then(run(2))
16   .then(run(3));
```

正如代码所示，这一次我们让 `run()` 方法返回一个函数，这个函数执行的时候会返回一个 `Promise` 对象。这样，这个 `Promise` 对象，并不是在程序一开始就初始化的，而是在未来的某一时刻，前一步操作完成之后才会得到执行，这一点非常关键，并且这是一种**通过给原始代码添加函数包装的方式实现了这里的“定义、传递、但不执行”的要求**。

这样做就是把实际的执行逻辑使用一个临时函数包装起来再传递出去，以达到延迟对该逻辑求值的目的，这种方式有一个专有名字 [🔗Thunk](#)，它是一种在 JavaScript 异步编程的世界中很常见的手段（JavaScript 中有时 `Thunk` 特指用这种技术来将多参数函数包装成单参数函数，这种情况我们在此不讨论）。换言之，上面代码例子中的第二行，绝不可省略，一些刚开始学写异步编程的程序员朋友，就很容易犯这个错误。

另外，这里我还使用了两个小技巧来简化代码：


一个是 `() => { return xxx; }` 可以被简化为 `() => xxx;`

另一个是使用 `Promise.resolve()` 返回一个已经执行成功的空操作，从而将所有后续执行的 `run` 方法都可以以统一的形式放到调用链里面去。

现在，使用 `run()` 方法的代码调用已经格外地简单而清晰了。在 `Promise` 的帮助下，通过这种方式，用了几个 `then` 方法，实现了逻辑在前一步成功后的依次执行。于是，**嵌套的金字塔厄运消失了，变成了直观的链式调用**，这是异步编程中一个非常常见的优化。

如果我们乘胜追击，进一步考虑，上面那个 `run()` 方法明显不够直观，能不能以某种方式优化调整一下？

能！代码看起来复杂的原因是引入了 `setTimeout`，而我们使用 `setTimeout` 只是为了“等一下”，来模拟小狗奔跑的过程。这个“等一下”的行为，实际是有普遍意义的。在 JavaScript 这样的非阻塞代码中，不可能通过代码的方式让代码实际执行的时候真的“等一下”，但是，我们却可以使用异步的方式让代码看起来像是执行了一个“等一下”的操作。我们定义：

 复制代码

```
1 var wait = ms =>
2   new Promise(resolve => setTimeout(resolve, ms));
```

有了 `wait` 的铺垫，我们把原本奔跑的 `setTimeout` 使用更为直观的 `wait` 函数来替换，一下子就让 `run` 的实现清晰了很多：

 复制代码

```
1 var run = steps =>
2   () => wait(1000).then(() => { console.log(steps); });
```

你看，这个例子，再加上前面的 `then` 调用链的例子，你是否看出，**利用 `Promise`，我们似乎神奇地把“异步”代码变成“同步”的了**。其实，代码执行并没有真正地变成同步，但是代码却“看起来”像是同步代码，而同步和顺序执行的逻辑对于人的大脑更为友好。

经过这样的重构以后，再次执行刚才的 3 次奔跑调用，我们得到了一样的结果。



[复制代码](#)

```
1 Promise.resolve()  
2   .then(run(1))  
3   .then(run(2))  
4   .then(run(3));
```

嗯，看起来我们已经做到极致了，代码也已经很清楚了，大概没有办法再改写和优化了吧？不！其实我们还有继续操作的办法。也许我应该说，居然还有。

在 ES7 中，`async/await` 的语法糖被引入。通过它，我们可以进一步优化代码的写法，让异步编程越来越像同步编程，也越来越接近人大脑自然的思维。

`async` 用于标记当前的函数为异步函数；

`await` 用于表示它的后面要返回一个 `Promise` 对象，在这个 `Promise` 对象得到异步结果以后，再继续往下执行。

考虑一下上面的 `run` 方法，现在我们可以把它改写成 `async/await` 的形式：

[复制代码](#)

```
1 var run = async steps => {  
2   await wait(1000);  
3   console.log(steps);  
4 }
```

你看，代码看起来就和同步的没有本质区别了，等待 1000 毫秒以后，打印 `steps`。

接着，如果我们执行下面的代码（如果你不是在 Chrome 的控制台执行，你可以把下面三行代码放到任意一个 `async` 函数中去执行，效果是一样的）：

[复制代码](#)

```
1 await run(1);  
2 await run(2);  
3 await run(3);
```

我们得到了一样的结果。这段代码看起来也和顺序、同步执行的代码没有区别了，虽然，实际的运行依然是前面你看到的异步调用，这里的效果只是 `async/await` 语法糖为程序员创造的一个美好的假象。

纵观这个小狗奔跑的问题，我们一步一步把晦涩难懂的嵌套回调代码，优化成了易读、易理解的“假同步”代码。聪明的程序员总在努力地创造各种工具，去**改善代码异步调用的表达能力，但是越是深入，就越能发现，最自然的表达，似乎来自于纯粹的同步代码。**

## 2. 用生成器来实现协程

**协程，Coroutine，简单说就是一种通用的协作式多任务的子程序，它通过任务执行的挂起与恢复，来实现任务之间的切换。**

这里提到的“协作式”，是一种多任务处理的模式，它和“抢占式”相对。如果是协作式，每个任务处理的逻辑必须主动放弃执行权（挂起），将继续执行的资源让出来给别的任务，直到重新获得继续执行的机会（恢复）；而抢占式则完全将任务调度交由第三方，比如操作系统，它可以直接剥夺当前执行任务的资源，分配给其它任务。

我们知道，创建线程的开销比进程小，而协程通常完全是在同一个线程内完成的，连线程切换的代价都免去了，因此它在资源开销方面更有优势。


JavaScript 的协程是通过生成器来实现的，执行的主流程在生成器中可以以 `yield` 为界，进行协作式的挂起和恢复操作，从而在外部函数和生成器内部逻辑之间跳转，而 JavaScript 引擎会负责管理上下文的切换。

首先我们来认识一下 JavaScript 和迭代有关的两个协议，它们是我们后面学习生成器的基础：

第一个是可迭代协议，它允许定义对象自己的迭代行为，比如哪些属性方法是可以被 `for` 循环遍历到的；

第二个是迭代器协议，它定义了一种标准的方法来依次产生序列的下一个值（`next()` 方法），如果序列是有限长的，并且在所有的值都产生后，将有一个默认的返回值。


接着我就可以介绍生成器（Generator）了。在 JavaScript 中，生成器对象是由生成器函数 `function*` 返回，且符合“可迭代协议”和“迭代器协议”两者。`function*` 和 `yield` 关键字通常一起使用，`yield` 用来在生成器的 `next()` 方法执行时，标识生成器执行中断的位置，并将 `yield` 右侧表达式的值返回。见下面这个简单的例子：

 复制代码

```
1 function* IdGenerator() {  
2   let index = 1;  
3   while (true)  
4     yield index++;  
5 }  
6  
7 var idGenerator = IdGenerator();  
8  
9 console.log(idGenerator.next());  
10 console.log(idGenerator.next());
```

这是一个 id 顺序生成的生成器，初始 `index` 为 1，每次调用 `next()` 来获取序列的下一个数值，并且 `index` 会自增 1。从代码中我们可以看到，这是一个无限的序列。

执行上述代码，我们将得到：

 复制代码

```
1 {value: 1, done: false}  
2 {value: 2, done: false}
```

每次返回的对象里面，`value` 的值就是生成的 id，而 `done` 的值表示这个序列是否结束。

你看，以往我们说起遍历的时候，脑海里总会第一时间想起某个容器，某个数据集合，但是，有了生成器以后，我们就可以对更为复杂的逻辑进行迭代。

生成器可不是只能往外返回，还能往里传值。具体说，`yield` 右侧的表达式会返回，但是在调用 `next()` 方法时，入参会被替代掉 `yield` 及右侧的表达式而参与代码运算。我们将上面的例子小小地改动一下：

 复制代码


```
1 function* IdGenerator() {
```



```
2   let index = 1, factor = 1;
3   while (true) {
4       factor = yield index; // 位置①
5       index = yield factor * index; // 位置②
6   }
7 }
```

好，这是生成器的定义，其调用代码如下：

```
1 var calculate = (idGenerator) => {
2   console.log(idGenerator.next());
3   console.log(idGenerator.next(1));
4   console.log(idGenerator.next(2));
5   console.log(idGenerator.next(3));
6 };
7
8 calculate(IdGenerator());
```

 复制代码

在往下阅读以前，你能不能先想一想，这个 calculate 方法的调用，会产生怎样的输出？

好，我来解释一下整个过程。现在这个 id 生成器每个循环节可以通过 yield 返回两次，我把上述执行步骤解释一下（为了便于说明代码位置，在生成器代码中我标记了“位置①”和“位置②”，请对应起来查看）：

调用 next(), 位置①的 yield 右侧的 index 返回，因此值为 1；

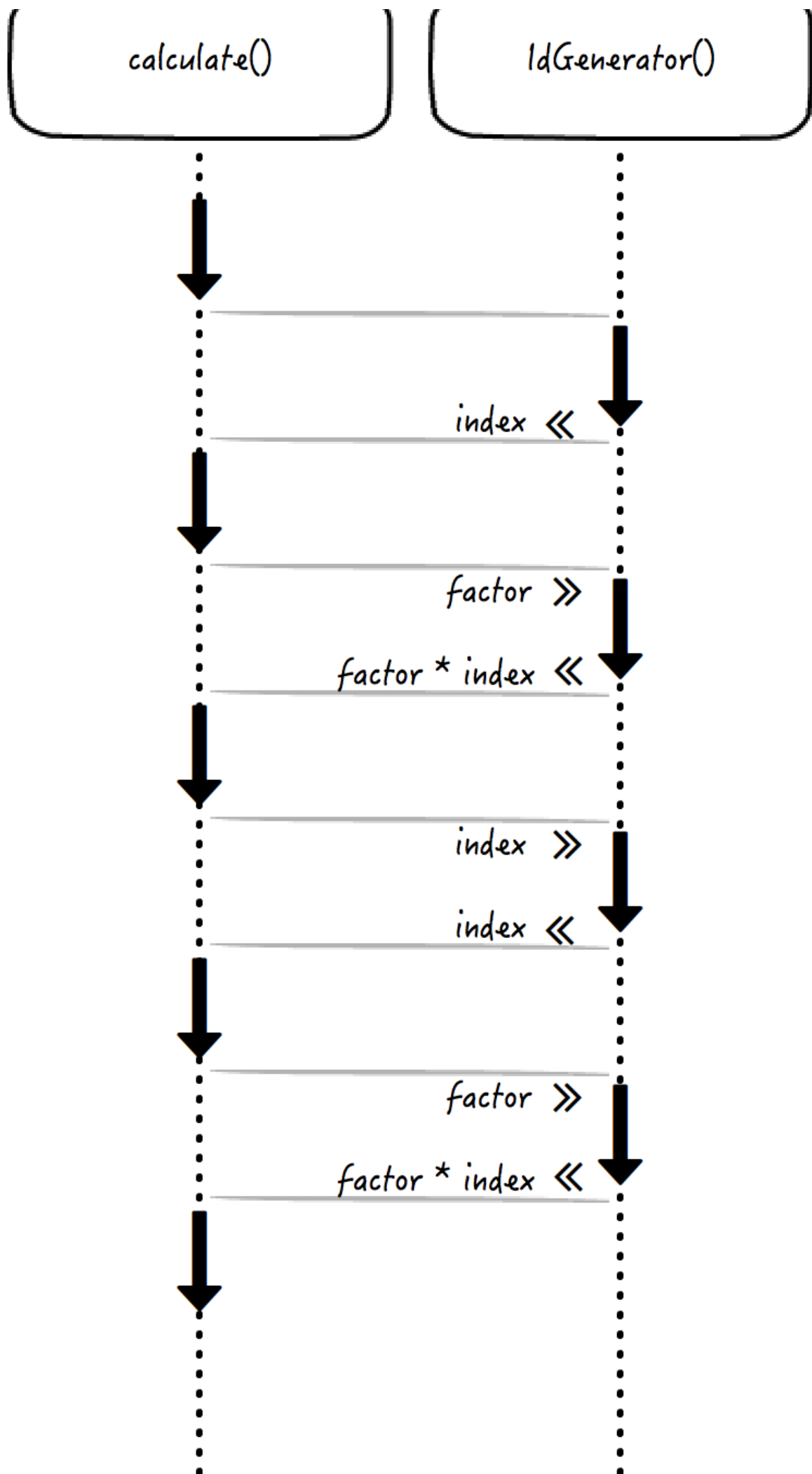
调用 next(1), 实参为 1，它被赋值给位置①的 factor，参与位置②的 yield 右侧的表达式计算，得到 1；

调用 next(2), 实参为 2，它被赋值给位置②的 index，由于 while 循环的关系，位置①的 yield 右侧的 index 返回，因此得到 2；

调用 next(3), 实参为 3，它被赋值给位置①的 factor，参与位置②的 yield 右侧的表达式计算， $3 * 2$  得到 6。

使用图来表示，就是这样子：





⋮

⋮

从图中你应该可以理解，通过生成器来实现 JavaScript 协程的原理了。本质上来说，**生成器将一个完整的方法执行通过 yield 拆分成了多个部分，并且每个部分都可以有输入输出，整个过程就是一个简单的状态机。**它和其它函数一起，以反复挂起和恢复的方式一段一段地将任务完成。


最后，结果输出如下：

 复制代码

```
1 {value: 1, done: false}
2 {value: 1, done: false}
3 {value: 2, done: false}
4 {value: 6, done: false}
```


### 3. 异步错误处理

错误处理是所有编程范型都必须要考虑的问题，在使用 JavaScript 进行异步编程时，也不例外。你可能会有这样一个疑问，如果我们不做特殊处理，会怎样呢？且看下面的代码，我先定义一个必定会失败的方法：

 复制代码

```
1 var fail = () => {
2   setTimeout(() => {
3     throw new Error("fail");
4   }, 1000);
5 };
```


然后调用一下：

 复制代码

```
1 console.log(1);
2 try {
```

```
3   fail();
4 } catch (e) {
5   console.log("captured");
6 }
7 console.log(2);
```

在 Chrome 开发者工具的控制台中执行一下，我们将看到 1 和 2 的输出，并在 1 秒钟之后，获得一个 “Uncaught Error” 的错误打印，注意观察这个错误的堆栈：

 复制代码

```
1 Uncaught Error: fail
2   at <anonymous>:3:11
3   at e (lizard-service-vendor.2b011077.js:1)
4   (anonymous) @ VM261:3
5   e @ lizard-service-vendor.2b011077.js:1
6   setTimeout (async)
7   (anonymous) @ lizard-service-vendor.2b011077.js:1
8   fail @ VM261:2
9   (anonymous) @ VM296:3
```

我们看到了其中的 `setTimeout (async)` 这样的字样，表示着这是一个异步调用抛出的堆栈，但是，“captured” 这样的字样也并未打印，因为方法 `fail()` 本身的原始顺序执行并没有失败，这个异常的抛出是在回调行为里发生的。

从上面的例子可以看出，对于异步编程来说，我们需要使用一种更好的机制来捕获并处理可能发生的异常。


## Promise 的异常处理

还记得上面介绍的 Promise 吗？它除了支持 `resolve` 回调以外，还支持 `reject` 回调，前者用于表示异步调用顺利结束，而后者则表示有异常发生，中断调用链并将异常抛出：

 复制代码


```
1 var exe = (flag) =>
2   () => new Promise((resolve, reject) => {
3     console.log(flag);
4     setTimeout(() => { flag ? resolve("yes") : reject("no"); }, 1000);
5   });
```

上面的代码中，flag 参数用来控制流程是顺利执行还是发生错误。在错误发生的时候，no 字符串会被传递给 reject 函数，进一步传递给调用链：

 复制代码

```
1 Promise.resolve()  
2   .then(exe(false))  
3   .then(exe(true));
```

你看，上面的调用链，在执行的时候，第二行就传入了参数 false，它就已经失败了，异常抛出了，因此第三行的 exe 实际没有得到执行，你会看到这样的执行结果：

 复制代码

```
1 false  
2 Uncaught (in promise) no
```


这就说明，通过这种方式，调用链被中断了，下一个正常逻辑 exe(true) 没有被执行。

但是，有时候我们需要捕获错误，而继续执行后面的逻辑，该怎样做？这种情况下我们就要在调用链中使用 catch 了：

 复制代码

```
1 Promise.resolve()  
2   .then(exe(false))  
3   .catch((info) => { console.log(info); })  
4   .then(exe(true));
```

这种方式下，异常信息被捕获并打印，而调用链的下一步，也就是第四行的 exe(true) 可以继续被执行。我们将看到这样的输出：

 复制代码

```
1 false  
2 no  
3 true
```

## async/await 下的异常处理

利用 `async/await` 的语法糖，我们可以像处理同步代码的异常一样，来处理异步代码：

 复制代码

```
1 var run = async () => {
2   try {
3     await exe(false)();
4     await exe(true)();
5   } catch (e) {
6     console.log(e);
7   }
8 }
9
10 run();
```

简单说明一下，定义一个异步方法 `run`，由于 `await` 后面需要直接跟 `Promise` 对象，因此我们通过额外的一个方法调用符号 `()` 把原有的 `exe` 方法内部的 `Thunk` 包装拆掉，即执行 `exe(false)()` 或 `exe(true)()` 返回的就是 `Promise` 对象。在 `try` 块之后，我们使用 `catch` 来捕捉。运行代码，我们得到了这样的输出：

 复制代码

```
1 false
2 no
```

这个 `false` 就是 `exe` 方法对入参的输出，而这个 `no` 就是 `setTimeout` 方法 `reject` 的回调返回，它通过异常捕获并最终在 `catch` 块中输出。就像我们所认识的同步代码一样，第四行的 `exe(true)` 并未得到执行。

## 总结思考

今天我们结合实例学习了 JavaScript 异步编程的一些方法，包括使用 `Promise` 或 `async/await` 来改善异步代码，使用生成器来实现协程，以及怎样进行异步错误处理等等。其中，`Promise` 相关的使用是需要重点理解的内容，因为它的应用性非常普遍。

现在，我来提两个问题：

在你的项目中，是否使用过 JavaScript 异步编程，都使用了和异步编程有关的哪些技术呢？



ES6 和 ES7 引入了很多 JavaScript 的高级特性和语法糖，包括这一讲提到的部分。有程序员朋友认为，这些在项目中的应用，反而给编程人员的阅读和理解造成了困扰，增加了学习曲线，还不如不用它们，写“简单”的 JavaScript 语法。对此，你怎么看？

在本章我们学习了基于 Web 的全栈技术中，前端相关的部分，希望这些内容能够帮到你，在前端这块土地上成长为更好的工程师。同时，在这一章我们学到了很多套路和方法，请回想一下，并在未来的工作中慢慢应用和体会，它们都是可以应用到软件其它领域的设计和编码上的。在第四章，我们会将目光往后移，去了解了解持久化的世界，希望现在的你依然充满干劲！

## 扩展阅读

对于今天学习的 Promise，你可以在 MDN 的 [使用 Promise](#) 一文中读到更为详尽的介绍；第二个是生成器，生成器实际上功能很强大，它甚至可以嵌套使用，你也可以参见 [MDN 的示例教程](#)。

如果你想快速浏览 ES6 新带来的 JavaScript 高级特性，我推荐你浏览 [ECMAScript 6 入门](#)，从中挑选你感兴趣的内容阅读。

[Async-Await ≈ Generators + Promises](#) 这篇文章介绍了生成器、Promise 和 async/await 之间的关系，读完你就能明白“为什么我们说 async/await 是生成器和 Promise 的语法糖”，感兴趣的朋友可以阅读，想阅读中文版的可以参见 [这个翻译](#)。

# 全栈工程师修炼指南

从全栈入门到技能实战

熊焱

Oracle 首席软件工程师



新版升级：点击「👤 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

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

上一篇 18 | 千言万语不及一幅画：谈谈数据可视化

下一篇 20 | 特别放送：全栈团队的角色构成

## 精选留言 (3)

写留言



零维

2019-10-28

老师，如何使用 promise 和 generator 模拟出 await 的返回值呢？

如果用 async/await:

```
const res = await ajax();
```

变成 yield 是:

```
const res = yield ajaxWrap();...
```

展开 ∨

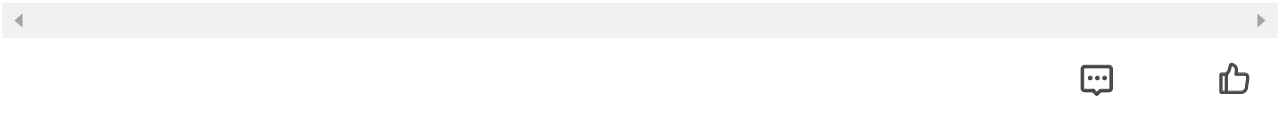


靠人品去赢

2019-10-25

这个promise “异步” 变 “同步”，就是让异步代码看起来像同步一样，刚看第一遍没看明白，还有就是箭头函数，箭头多了，我就不能第一时间看明白，看来还是用得少。

作者回复: 用得少是一个方面, 本来 JavaScript 如果书写的时候不注意结构和组织的话, 确实容易写出难懂的代码



tt

2019-10-23

老师, JavaScript和Python确实太像了, 尤其是异步函数、生成器以及协程的部分。

那么, 对于小规模团队, JavaScript+Python或者JavaScript+node.js的组合是不是比JavaScript+java在开发速度上更有优势呢?

展开 ▾

作者回复: 这个问题不太好回答, 我也没有足够的统计数据。据我个人的经历, Python 开发效率确实是要比 Java 高出不少

