13 全面掌握现代异步解决方案

更新时间：2020-04-02 09:49:46

> 读书给人以快乐、给人以光彩、给人以才干。——培根

> 注：关于 Promise 面试考点的详细展开，欢迎大家跳读下一章针对 Promise 的专门讲解。在本文中，我们仅将其作为异步方案来讨论。

Promise

长久以来，我们一直期望着一种既能实现异步、又可以确保我们的代码好写又好看的解决方案出现。带着这样的目标，经过反复的探索，我们终于迎来了 Promise。

用 Promise 实现异步，我们这样做（这里我改造了一个网络请求的过程）：

```
const https = require('https');

function httpPromise(url){
  return new Promise(function(resolve,reject){
    https.get(url, (res) => {
      resolve(data);
    }).on("error", (err) => {
      reject(error);
    });
  })
}

httpPromise().then(function(data){
}).catch(function(error){})
```

可以看出，Promise 会接收一个执行器，在这个执行器里，我们需要把目标的异步任务给”填进去“。
在 Promise 实例创建后，执行器里的逻辑会立刻执行，在执行的过程中，根据异步返回的结果，决定如何使用 resolve 或 reject 来改变 Promise实例的状态。 Promise 实例有三种状态：

• pending 状态，表示进行中。这是 Promise 实例创建后的一个初始态；
• fulfilled 状态，表示成功完成。这是我们在执行器中调用 resolve 后，达成的状态；
• rejected 状态，表示操作失败、被拒绝。这是我们在执行器中调用 reject后，达成的状态。

在上面这个例子里，当我们用 resolve 切换到了成功态后，Promise 的逻辑就会走到 then 中的传入的方法里去；用 reject 切换到失败态后，Promise 的逻辑就会走到 catch 传入的方法中去。

这样的逻辑，本质上与回调函数中的成功回调和失败回调无异。但这种写法毫无疑问大大地提高了代码的质量。最直接的例子就是当我们进行大量的异步链式调用时，回调地狱不复存在了。取而代之的，是层级简单、赏心悦目的 Promise 调用链：

```
httpPromise(url1)
  .then(res => {
    console.log(res);
    return httpPromise(url2);
  })
  .then(res => {
    console.log(res);
    return httpPromise(url3);
  })
  .then(res => {
   console.log(res);
   return httpPromise(url4);
  })
  .then(res => console.log(res));。
```

Generator

除了 Promise， ES2015 还为我们提供了 Generator 这个好帮手~ 。
如果你对 Generator 是什么、以及其语法特性暂时还没有太多的了解，可以点击 这里先进行预备知识的学习。

Generator 一个有利于异步的特性是，它可以在执行中被中断、然后等待一段时间再被我们唤醒。通过这个“中断后唤醒”的机制，我们可以把 Generator看作是异步任务的容器，利用 yield 关键字，实现对异步任务的等待。

比如咱们用 Promise 链式调用这么写的例子：

```
httpPromise(url1)
  .then(res => {
    console.log(res);
    return httpPromise(url2);
  })
  .then(res => {
    console.log(res);
    return httpPromise(url3);
  })
  .then(res => {
   console.log(res);
   return httpPromise(url4);
  })
  .then(res => console.log(res));
```

其实完全可以用 yield 来这么写：

```
function* httpGenerator() {
  let res1 = yield httpPromise(url1)
  console.log(res);
  let res2 = yield httpPromise(url2)
  console.log(res);
  let res3 = yield httpPromise(url3)
  console.log(res);
  let res4 = yield httpPromise(url4)
  console.log(res);
}
```

当然啦，单纯这么改还不够，我们还需要在调用层面再完善一下才能让这个生成器如期运行起来。
但在完善之前，咱们就单纯看这种写法，是不是比 Promise 链式调用更好看、更清晰了？这时候你一眼看过去就知道这段逻辑在干嘛，而不必再对所谓的“链”作分析。干干净净、一目了然！

现在我们想办法让 httpGenerator 按照我们的预期跑起来：我们知道，Generator 要想跑起来，需要为它创建迭代器，然后去执行这个迭代器的 next 方法。在 httpGenerator 这个例子里，我们要想把整个函数体的逻辑走完，就必须让迭代器的 next 反复调用、直到返回值中的 done 为 true 为止。这个过程，我们当然不能手动调用，而要让程序来帮我们做：

```
function runGenerator(gen) {
  var it = gen(), ret;

  // 创造一个立即执行的递归函数
  (function iterate(val){
    ret = it.next(val);

    if (!ret.done) {
      // 如果能拿到一个 promise 实例
      if ("then" in ret.value) {
        // 就在它的 then 方法里递归调用 iterate
        ret.value.then( iterate );
      }
    }
  })();
}

runGenerator(httpGenerator)
```

大家一起来看下 runGenerator 这个方法，当我们把 httpGenerator 传进去后，会发生如下过程：

为传入的 Generator 创建它对应的迭代器 it。然后，我们第一次调用 iterate 函数，入参为空。

iterate 函数内部，调用 it 的 next 方法，生成器函数开始执行，执行到第一个 yield 关键字处的逻辑执行完后暂停。它会返回一个包含了 httpPromise(url1) 这个调用返回的 promise对象（我们下文称 promise1）、以及一个 done: false 的标识，用来表示当前生成器函数内部的逻辑还没执行完（大致如下）：

```
{
  value:
   Promise {
     <pending>,
     ...// 省略一系列 promise 对象关联信息
   },
  done: false
}
```

因为 done 为 false，所以我们会进一步判断当前拿到的是否是一个 promise 对象（根据它有没有 then 属性）。判断为真后，我们在 promise1 的 then 方法里传入 iterate 函数本身。

promise1 的 then 方法里的 iterate 函数调用，拿到了 promise1 的返回结果（即针对 url1 的请求结果）作为入参。it.next 被第二次调用，生成器函数被“唤醒”了。注意，被“唤醒”后的生成器函数，按照流程走，它执行的第一个语句就是:

```
let res1 = yield httpPromise(url1)
```

这一步会把 next(val) 中的 val 传给 res1，而 val，恰恰就是 promise1 的返回结果。一切正如我们所预期~~

而后，生成器函数会继续执行到第二个 yield 关键字处，执行完后暂停。

此时 next 方法返回一个包含了 httpPromise(url2) 这个调用返回的 promise 对象（我们下文称 promise2）、以及一个 done: false 的标识（用来表示当前生成器函数内部的逻辑还没执行完）。因为 done 为 false，所以我们会进一步判断当前拿到的是否是一个 promise 对象（根据它有没有 then 属性）。判断为真后，我们在 promise2 的 then 方法里传入 iterate 函数本身。

循环上述过程过程，直到生成器内部逻辑执行完为止。
通过“自动执行”生成器函数对应迭代器的 next 方法，我们把异步的写法进一步优化了。它不再需要地狱般的回调，甚至不再需要 Promise 长长的链式调用，而是可以像写同步代码一样简单、清晰地实现异步特性！

不过仔细想想，咱们这个 runGenerator 其实非常简陋，它虽然体现了自动执行的思想，却不具备通用性，无法兼容更多场景——确实，要写出一个完整周到的 runGenerator 函数，不是一件轻松的事情。但是有一个好用的 runGenerator，又确实是广大开发者的强诉求。于是我们有了一个叫 co 的库，专门来封装自执行这一层的逻辑：

```
const co = require('co');
co(httpGenerator());
```

这里的 co，大家就可以把它看作是一个加强版的 runGenerator。我们只需要在代码里引入 co 库，然后把写好的 generator 传进去，就可以轻松地实现 generator 异步了。

Async/Await

就当大家正在纷纷感慨 co 真好使，generator + promise + co 的异步方案真优雅时，更强的家伙出现了。这玩意儿甚至甩开了 co、甩开了 generator，有了它，你什么都不用操心，只需要写几个关键字，就能把异步代码处理得像同步代码一样优雅！这玩意儿就是 async/await。

它的用法非常简单。首先，我们用 async 关键字声明一个函数为“异步函数”：

```
async function httpRequest() {

}
```

然后，我们就可以在这个函数内部使用 await 关键字了：

```
async function httpRequest() {
  let res1 = await httpPromise(url1)
  console.log(res1)
}
```

这个 await 关键字很绝，它的意思就是“我要异步了，可能会花点时间，后面的语句都给我等着”。当我们给 httpPromise(url1) 这个异步任务应用了 await 关键字后，整个函数会像被“yield”了一样，暂停下来，直到异步任务的结果返回后，它才会被“唤醒”，继续执行后面的语句。

是不是觉得这个“暂停”、”唤醒“的操作，和 generator 异步非常相似？事实上，async/await 本身就是 generator 异步方案的语法糖。它的诞生主要就是为了这个单纯而美好的目的——让你写得更爽，让你写出来的代码更美。

注：async/await 和 generator 方案，相较于 Promise 而言，有一个重要的优势：Promise 的错误需要通过回调函数捕获，try catch 是行不通的。而 async/await 和 generator 允许 try/catch。这也是一个可以作为命题点细节，大家留心把握。

}

12 异步编程模型与异步解决方案

14 Promise 命题思路全解析