Generator 函数的含义与用法

作者： 阮一峰

日期： 2015年4月24日

本文是《深入掌握 ECMAScript 6 异步编程》系列文章的第一篇。

异步编程对 JavaScript 语言太重要。JavaScript 只有一根线程，如果没有异步编程，根本没法用，非卡死不可。

以前，异步编程的方法，大概有下面四种。

- 回调函数
- 事件监听
- 发布/订阅
- `Promise` 对象

ECMAScript 6（简称 ES6）作为下一代 JavaScript 语言，将 JavaScript 异步编程带入了一个全新的阶段。**这组系列文章的主题，就是介绍更强大、更完善的 ES6 异步编程方法。**

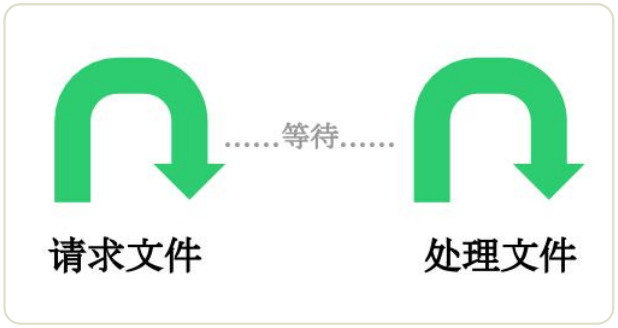
新方法比较抽象，初学时，我常常感到费解，直到很久以后才想通，**异步编程的语法目标，就是怎样让它更像同步编程。**这组系列文章，将帮助你深入理解 JavaScript 异步编程的本质。所有将要讲到的内容，都已经实现了。也就是说，马上就能用，套用一句广告语，就是"未来已来"。

一、什么是异步？

所谓"异步"，简单说就是一个任务分成两段，先执行第一段，然后转而执行其他任务，等做好了准备，再回过头执行第二段。比如，有一个任务是读取文件进行处理，异步的执行过程就是下面这样。

上图中，任务的第一段是向操作系统发出请求，要求读取文件。然后，程序执行其他任务，等到操作系统返回文件，再接着执行任务的第二段（处理文件）。

这种不连续的执行，就叫做异步。相应地，连续的执行，就叫做同步。



上图就是同步的执行方式。由于是连续执行，不能插入其他任务，所以操作系统从硬盘读取文件的这段时间，程序只能干等着。

二、回调函数的概念

JavaScript 语言对异步编程的实现，就是回调函数。**所谓回调函数，就是把任务的第二段单独写在一个函数里面，等到重新执行这个任务的时候，就直接调用这个函数。**它的英语名字 callback，直译过来就是"重新调用"。

读取文件进行处理，是这样写的。

```
fs.readFile('/etc/passwd', function (err, data) {
  if (err) throw err;
  console.log(data);
});
```

上面代码中，readFile 函数的第二个参数，就是回调函数，也就是任务的第二段。等到操作系统返回了 /etc/passwd 这个文件以后，回调函数才会执行。

一个有趣的问题是，为什么 Node.js 约定，回调函数的第一个参数，必须是错误对象err（如果没有错误，该参数就是 null）？原因是执行分成两段，在这两段之间抛出的错误，程序无法捕捉，只能当作参数，传入第二段。

三、Promise

回调函数本身并没有问题，它的问题出现在多个回调函数嵌套。假定读取A文件之后，再读取B文件，代码如下。

```
fs.readFile(fileA, function (err, data) {
  fs.readFile(fileB, function (err, data) {
    // ...
  });
});
```

不难想象，如果依次读取多个文件，就会出现多重嵌套。代码不是纵向发展，而是横向发展，很快就会乱成一团，无法管理。这种情况就称为"回调函数噩梦"（callback hell）。

Promise就是为了解决这个问题而提出的。它不是新的语法功能，而是一种新的写法，允许将回调函数的横向加载，改成纵向加载。采用Promise，连续读取多个文件，写法如下。

```
var readFile = require('fs-readfile-promise');

readFile(fileA)
.then(function(data){
  console.log(data.toString());
})
.then(function(){
  return readFile(fileB);
})
.then(function(data){
  console.log(data.toString());
})
.catch(function(err) {
  console.log(err);
});
```

上面代码中，我使用了 fs-readfile-promise 模块，它的作用就是返回一个 Promise 版本的 readFile 函数。Promise 提供 then 方法加载回调函数，catch方法捕捉执行过程中抛出的错误。

可以看到，Promise 的写法只是回调函数的改进，使用then方法以后，异步任务的两段执行看得更清楚了，除此以外，并无新意。

Promise 的最大问题是代码冗余，原来的任务被Promise 包装了一下，不管什么操作，一眼看去都是一堆 then，原来的语义变得很不清楚。

那么，有没有更好的写法呢？

四、协程

传统的编程语言，早有异步编程的解决方案（其实是多任务的解决方案）。其中有一种叫做"协程"（coroutine），意思是多个线程互相协作，完成异步任务。

协程有点像函数，又有点像线程。它的运行流程大致如下。

> 第一步，协程A开始执行。
>
> 第二步，协程A执行到一半，进入暂停，执行权转移到协程B。
>
> 第三步，（一段时间后）协程B交还执行权。
>
> 第四步，协程A恢复执行。

上面流程的协程A，就是异步任务，因为它分成两段（或多段）执行。

举例来说，读取文件的协程写法如下。

```
function asnycJob() {
  // ...其他代码
  var f = yield readFile(fileA);
  // ...其他代码
}
```

上面代码的函数 asyncJob 是一个协程，它的奥妙就在其中的 yield 命令。它表示执行到此处，执行权将交给其他协程。也就是说，yield命令是异步两个阶段的分界线。

协程遇到 yield 命令就暂停，等到执行权返回，再从暂停的地方继续往后执行。它的最大优点，就是代码的写法非常像同步操作，如果去除yield命令，简直一模一样。

五、Generator函数的概念

Generator 函数是协程在 ES6 的实现，最大特点就是可以交出函数的执行权（即暂停执行）。

```
function* gen(x){
  var y = yield x + 2;
  return y;
}
```

上面代码就是一个 Generator 函数。它不同于普通函数，是可以暂停执行的，所以函数名之前要加星号，以示区别。

整个 Generator 函数就是一个封装的异步任务，或者说是异步任务的容器。异步操作需要暂停的地方，都用 yield 语句注明。Generator 函数的执行方法如下。

```
var g = gen(1);
g.next() // { value: 3, done: false }
g.next() // { value: undefined, done: true }
```

上面代码中，调用 Generator 函数，会返回一个内部指针（即遍历器 ）g 。这是 Generator 函数不同于普通函数的另一个地方，即执行它不会返回结果，返回的是指针对象。调用指针 g 的 next 方法，会移动内部指针（即执行异步任务的第一段），指向第一个遇到的 yield 语句，上例是执行到 x + 2 为止。

换言之，next 方法的作用是分阶段执行 Generator 函数。每次调用 next 方法，会返回一个对象，表示当前阶段的信息（ value 属性和 done 属性）。value 属性是 yield 语句后面表达式的值，表示当前阶段的值；done 属性是一个布尔值，表示 Generator 函数是否执行完毕，即是否还有下一个阶段。

六、Generator 函数的数据交换和错误处理

Generator 函数可以暂停执行和恢复执行，这是它能封装异步任务的根本原因。除此之外，它还有两个特性，使它可以作为异步编程的完整解决方案：函数体内外的数据交换和错误处理机制。

next 方法返回值的 value 属性，是 Generator 函数向外输出数据；next 方法还可以接受参数，这是向 Generator 函数体内输入数据。

```
function* gen(x){
  var y = yield x + 2;
  return y;
}

var g = gen(1);
g.next() // { value: 3, done: false }
g.next(2) // { value: 2, done: true }
```

上面代码中，第一个 next 方法的 value 属性，返回表达式 x + 2 的值（3）。第二个 next 方法带有参数2，这个参数可以传入 Generator 函数，作为上个阶段异步任务的返回结果，被函数体内的变量 y 接收。因此，这一步的 value 属性，返回的就是2（变量 y 的值）。

Generator 函数内部还可以部署错误处理代码，捕获函数体外抛出的错误。

```
function* gen(x){
  try {
    var y = yield x + 2;
  } catch (e){
    console.log(e);
  }
  return y;
}
```

```
var g = gen(1);
g.next();
g.throw('出错了');
// 出错了
```

上面代码的最后一行，Generator 函数体外，使用指针对象的 throw 方法抛出的错误，可以被函数体内的 try ... catch 代码块捕获。这意味着，出错的代码与处理错误的代码，实现了时间和空间上的分离，这对于异步编程无疑是很重要的。

七、Generator 函数的用法

下面看看如何使用 Generator 函数，执行一个真实的异步任务。

```
var fetch = require('node-fetch');

function* gen(){
  var url = 'https://api.github.com/users/github';
  var result = yield fetch(url);
  console.log(result.bio);
}
```

上面代码中，Generator 函数封装了一个异步操作，该操作先读取一个远程接口，然后从 JSON 格式的数据解析信息。就像前面说过的，这段代码非常像同步操作，除了加上了 yield 命令。

执行这段代码的方法如下。

```
var g = gen();
var result = g.next();

result.value.then(function(data){
  return data.json();
}).then(function(data){
  g.next(data);
});
```

上面代码中，首先执行 Generator 函数，获取遍历器对象，然后使用 next 方法（第二行），执行异步任务的第一阶段。由于 Fetch 模块返回的是一个 Promise 对象，因此要用 then 方法调用下一个next 方法。

可以看到，虽然 Generator 函数将异步操作表示得很简洁，但是流程管理却不方便（即何时执行第一阶段、何时执行第二阶段）。本系列的后面部分，就将介绍如何自动化异步任务的流程管理。另外，本文对 Generator 函数的介绍很简单，详尽的教程请阅读我写的《ECMAScript 6入门》。

（完）