Thunk 函数的含义和用法

作者： 阮一峰

日期： 2015年5月 1日

本文是《深入掌握 ECMAScript 6 异步编程》系列文章的第二篇。

一、参数的求值策略

Thunk函数早在上个世纪60年代就诞生了。

那时，编程语言刚刚起步，计算机学家还在研究，编译器怎么写比较好。**一个争论的焦点是"求值策略"，即函数的参数到底应该何时求值。**

```
var x = 1;

function f(m){
  return m * 2;
}

f(x + 5)
```

上面代码先定义函数 f，然后向它传入表达式 x + 5 。请问，这个表达式应该何时求值？

一种意见是"传值调用"（call by value），即在进入函数体之前，就计算 x + 5 的值（等于6），再将这个值传入函数 f 。C语言就采用这种策略。

```
f(x + 5)
// 传值调用时，等同于
f(6)
```

另一种意见是"传名调用"（call by name），即直接将表达式 x + 5 传入函数体，只在用到它的时候求值。Hskell语言采用这种策略。

```
f(x + 5)
// 传名调用时，等同于
(x + 5) * 2
```

传值调用和传名调用，哪一种比较好？回答是各有利弊。传值调用比较简单，但是对参数求值的时候，实际上还没用到这个参数，有可能造成性能损失。

```
function f(a, b){
  return b;
}

f(3 * x * x - 2 * x - 1, x);
```

上面代码中，函数 f 的第一个参数是一个复杂的表达式，但是函数体内根本没用到。对这个参数求值，实际上是不必要的。

因此，有一些计算机学家倾向于"传名调用"，即只在执行时求值。

二、Thunk 函数的含义

编译器的"传名调用"实现，往往是将参数放到一个临时函数之中，再将这个临时函数传入函数体。这个临时函数就叫做 Thunk 函数。

```
function f(m){
  return m * 2;
```

```
}

f(x + 5);

// 等同于

var thunk = function () {
  return x + 5;
};

function f(thunk){
  return thunk() * 2;
}
```

上面代码中，函数 f 的参数 x + 5 被一个函数替换了。凡是用到原参数的地方，对 Thunk 函数求值即可。

这就是 Thunk 函数的定义，它是"传名调用"的一种实现策略，用来替换某个表达式。

三、JavaScript 语言的 Thunk 函数

JavaScript 语言是传值调用，它的 Thunk 函数含义有所不同。**在 JavaScript 语言中，Thunk 函数替换的不是表达式，而是多参数函数，将其替换成单参数的版本，且只接受回调函数作为参数。**

```
// 正常版本的readFile（多参数版本）
fs.readFile(fileName, callback);

// Thunk版本的readFile（单参数版本）
var readFileThunk = Thunk(fileName);
readFileThunk(callback);

var Thunk = function (fileName){
  return function (callback){
    return fs.readFile(fileName, callback);
  };
};
```

上面代码中，fs 模块的 readFile 方法是一个多参数函数，两个参数分别为文件名和回调函数。经过转换器处理，它变成了一个单参数函数，只接受回调函数作为参数。这个单参数版本，就叫做 Thunk 函数。

任何函数，只要参数有回调函数，就能写成 Thunk 函数的形式。下面是一个简单的 Thunk 函数转换器。

```
var Thunk = function(fn){
  return function (){
    var args = Array.prototype.slice.call(arguments);
    return function (callback){
      args.push(callback);
      return fn.apply(this, args);
    }
  };
};
```

使用上面的转换器，生成 fs.readFile 的 Thunk 函数。

```
var readFileThunk = Thunk(fs.readFile);
readFileThunk(fileA)(callback);
```

四、Thunkify 模块

生产环境的转换器，建议使用 Thunkify 模块。

首先是安装。

```
$ npm install thunkify
```

使用方式如下。

```
var thunkify = require('thunkify');
var fs = require('fs');

var read = thunkify(fs.readFile);
read('package.json')(function(err, str){
  // ...
});
```

Thunkify 的源码与上一节那个简单的转换器非常像。

```
function thunkify(fn){
  return function(){
    var args = new Array(arguments.length);
    var ctx = this;

    for(var i = 0; i < args.length; ++i) {
      args[i] = arguments[i];
    }

    return function(done){
      var called;

      args.push(function(){
        if (called) return;
        called = true;
        done.apply(null, arguments);
      });
```

```
      try {
        fn.apply(ctx, args);
      } catch (err) {
        done(err);
      }
    }
  }
};
```

它的源码主要多了一个检查机制，变量 called 确保回调函数只运行一次。这样的设计与下文的 Generator 函数相关。请看下面的例子。

```
function f(a, b, callback){
  var sum = a + b;
  callback(sum);
  callback(sum);
}

var ft = thunkify(f);
ft(1, 2)(console.log);
// 3
```

上面代码中，由于 thunkify 只允许回调函数执行一次，所以只输出一行结果。

五、Generator 函数的流程管理

你可能会问， Thunk 函数有什么用？回答是以前确实没什么用，但是 ES6 有了 Generator 函数，Thunk 函数现在可以用于 Generator 函数的自动流程管理。

以读取文件为例。下面的 Generator 函数封装了两个异步操作。

```
var fs = require('fs');
var thunkify = require('thunkify');
var readFile = thunkify(fs.readFile);

var gen = function* (){
  var r1 = yield readFile('/etc/fstab');
  console.log(r1.toString());
  var r2 = yield readFile('/etc/shells');
  console.log(r2.toString());
};
```

上面代码中，yield 命令用于将程序的执行权移出 Generator 函数，那么就需要一种方法，将执行权再交还给 Generator 函数。

这种方法就是 Thunk 函数，因为它可以在回调函数里，将执行权交还给 Generator 函数。为了便于理解，我们先看如何手动执行上面这个 Generator 函数。

```
var g = gen();

var r1 = g.next();
r1.value(function(err, data){
  if (err) throw err;
  var r2 = g.next(data);
  r2.value(function(err, data){
    if (err) throw err;
    g.next(data);
  });
});
```

上面代码中，变量 g 是 Generator 函数的内部指针，表示目前执行到哪一步。next 方法负责将指针移动到下一步，并返回该步的信息（value 属性和 done 属性）。

仔细查看上面的代码，可以发现 Generator 函数的执行过程，其实是将同一个回调函数，反复传入 next 方法的 value 属性。这使得我们可以用递归来自动完成这个过程。

六、Thunk 函数的自动流程管理

Thunk 函数真正的威力，在于可以自动执行 Generator 函数。下面就是一个基于 Thunk 函数的 Generator 执行器。

```
function run(fn) {
  var gen = fn();

  function next(err, data) {
    var result = gen.next(data);
    if (result.done) return;
    result.value(next);
  }

  next();
}

run(gen);
```

上面代码的 run 函数，就是一个 Generator 函数的自动执行器。内部的 next 函数就是 Thunk 的回调函数。 next 函数先将指针移到 Generator 函数的下一步（gen.next 方法），然后判断 Generator 函数是否结束（result.done 属性），如果没结束，就将 next 函数再传入 Thunk 函数（result.value 属性），否则就直接退出。

有了这个执行器，执行 Generator 函数方便多了。不管有多少个异步操作，直接传入 run 函数即可。当然，前提是每一个异步操作，都要是 Thunk 函数，也就是说，跟在 yield 命令后面的必须是 Thunk 函数。

```
var gen = function* (){
  var f1 = yield readFile('fileA');
  var f2 = yield readFile('fileB');
  // ...
  var fn = yield readFile('fileN');
};

run(gen);
```

上面代码中，函数 gen 封装了 n 个异步的读取文件操作，只要执行 run 函数，这些操作就会自动完成。这样一来，异步操作不仅可以写得像同步操作，而且一行代码就可以执行。

Thunk 函数并不是 Generator 函数自动执行的唯一方案。因为自动执行的关键是，必须有一种机制，自动控制 Generator 函数的流程，接收和交还程序的执行权。回调函数可以做到这一点，Promise 对象也可以做到这一点。本系列的下一篇，将介绍基于 Promise 的自动执行器。

（完）