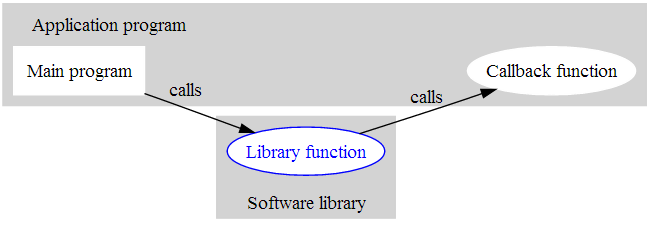
最近看到nodejs，因为有一个处理里面有好几个异步操作，调入回调大坑，不禁觉得很恶心，真的很讨厌发明这种写法的人，简直反社会！！！遂转载一篇解坑的文章，原文地址：<http://www.infoq.com/cn/articles/nodejs-callback-hell/>。

[**Node.js**](http://lib.csdn.net/base/nodejs)需要按顺序执行异步逻辑时一般采用后续传递风格，也就是将后续逻辑封装在回调函数中作为起始函数的参数，逐层嵌套。这种风格虽然可以提高CPU利用率，降低等待时间，但当后续逻辑步骤较多时会影响代码的可读性，结果代码的修改维护变得很困难。根据这种代码的样子，一般称其为”callback hell”或”pyramid of doom”，本文称之为回调大坑，嵌套越多，大坑越深。   
坑的起源

后续传递风格

为什么会有坑？这要从后续传递风格（continuation-passing style–CPS)说起。这种编程风格最开始是由Gerald Jay Sussman和Guy L. Steele, Jr. 在AI Memo 349上提出来的，那一年是1975年，Schema语言的第一次亮相。既然**[JavaScript](http://lib.csdn.net/base/javascript" \o "JavaScript知识库" \t "_blank)**的函数式编程设计原则主要源自Schema，这种风格自然也被带到了Javascript中。

这种风格的函数要有额外的参数：“后续逻辑体”，**比如带一个参数的函数。CPS函数计算出结果值后并不是直接返回，而是调用那个后续逻辑函数，并把这个结果作为它的参数。从而实现计算结果在逻辑步骤之间的传递，以及逻辑的延续。也就是说如果要调用CPS函数，调用方函数要提供一个后续逻辑函数来接收CPS函数的“返回”值。   
回调**

在JavaScript中，这个“后续逻辑体”就是我们常说的回调(callback)。这种作为参数的函数之所以被称为回调，是因为它一般在主程序中定义，由主程序交给库函数，并由它在需要时回来调用。而将回调函数作为参数的，一般是一个会占用较长时间的异步函数，要交给另一个线程执行，以便不影响主程序的后续操作。如下图所示：   
   
下面一个例子说明回调样例的恶心之处：

module.exports = function (param, cb) {

asyncFun1(param, function (er, data) {

if (er) return cb(er);

asyncFun2(data,function (er,data) {

if (er) return cb(er);

asyncFun3(data, function (er, data) {

if (er) return cb(er);

cb(data);

})

})

})

}

像function(er,data)这种回调函数签名很常见，几乎所有的Node.js核心库及第三方库中的CPS函数都接收这样的函数参数，它的第一个参数是错误，其余参数是CPS函数要传递的结果。比如Node.js中负责文件处理的fs模块，我们再看一个实际工作中可能会遇到的例子。要找出一个目录中最大的文件，处理步骤应该是：

1. 用fs.readdir获取目录中的文件列表；
2. 循环遍历文件，获取文件的stat；
3. 找出最大文件；
4. 以最大文件的文件名为参数调用回调。   
   这些都是异步操作，但需要顺序执行，后续传递风格的代码应该是下面这样的：

var fs = require('fs')

var path = require('path')

module.exports = function (dir, cb) {

fs.readdir(dir, function (er, files) { // [1]

if (er) return cb(er)

var counter = files.length

var errored = false

var stats = []

files.forEach(function (file, index) {

fs.stat(path.join(dir,file), function (er, stat) { // [2]

if (errored) return

if (er) {

errored = true

return cb(er)

}

stats[index] = stat // [3]

if (--counter == 0) { // [4]

var largest = stats

.filter(function (stat) { return stat.isFile() }) // [5]

.reduce(function (prev, next) { // [6]

if (prev.size > next.size) return prev

return next

})

cb(null, files[stats.indexOf(largest)]) // [7]

}

})

})

})

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30

对这个模块的用户来说，只需要提供一个回调函数function(er,filename)，用两个参数分别接收错误或文件名：

var findLargest = require('./findLargest')

findLargest('./path/to/dir', function (er, filename) {

if (er) return console.error(er)

console.log('largest file was:', filename)

})

* 5介绍完CPS和回调，我们接下来看看如何平坑。

解套平坑

编写正确的并发程序归根结底是要让尽可能多的操作同步进行，但各操作的先后顺序仍能正确无误。服务端的代码一般逻辑比较复杂，步骤多，此时用嵌套实现异步函数的顺序执行会比较痛苦，所以应该尽量避免嵌套，或者降低嵌套代码的复杂性，少用匿名函数。这一般有几种途径：

最简单的是把匿名函数拿出来定义成单独的函数，然后或者像原来一样用嵌套方式调用，或者借助流程控制模块放在数组里逐一调用；   
用Promis；   
如果你的Node版本>=0.11.2，可以用generator。   
我们先介绍最容易理解的流程控制模块。   
流程控制模块

Nimble是一个轻量、可移植的函数式流程控制模块。经过最小化和压缩后只有837字节，可以运行在Node.js中，也可以用在各种浏览器中。它整合了underscore和async一些最实用的功能，并且API更简单。

nimble有两个流程控制函数，*.parallel和*.series。顾名思义，我们要用的是第二个，可以让一组函数串行执行的\_.series。下面这个命令是用来安装Nimble的：

npm install nimble

* 1
* 1

如果用.series调度执行上面那个解方程的函数，代码应该是这样的：

var flow = require('nimble');

(function calculate(i) {

if(i === l-1) {

variables[i] = res[i];

process.exit();

}else {

flow.series([

function (callback) {

calculateTail(res[i],res[i+1],function(tail) {

variables[i] = tail;

callback();

});

},

function (callback) {

calculateHead(res[i],res[i+1],function(head) {

res[i+1] = head;

callback();

});

},

function(callback){

calculate(i+1);

}]);

}

})(0);

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24

.series数组参数中的函数会挨个执行，只是我们的calculateTail和calculateHead都被包在了另一个函数中。尽管这个用流程控制实现的版本代码更多，但通常可读性和可维护性要强一些。接下来我们介绍Promise。   
Promise

什么是Promise呢？在纸牌屋的第一季第一集中，当琳达告诉安德伍德不能让他做国务卿后，他说：“所谓Promise，就是说它不会受不断变化的情况影响。”

Promise不仅去掉了嵌套，它连回调都去掉了。因为按照Promise的观点，回调一点也不符合函数式编程的精神。回调函数什么都不返回，没有返回值的函数，执行它仅仅是因为它的副作用。所以用回调函数编程天生就是指令式的，是以副作用为主的过程的执行顺序，而不是像函数那样把输入映射到输出，可以组装到一起。   
这里用的Promis框架是著名的Q，可以用npm install q安装。虽然可用的Promis框架有很多，但在它们用法上都大同小异。我们在这里会用到其中的三个方法。

第一个负责将Node.js的CPS函数变成Promise。Node.js核心库和第三方库中有非常多的CPS函数，我们的程序肯定要用到这些函数，要解决回调大坑，就要从这些函数开始。这些函数的回调函数参数大多遵循一个相同的模式，即函数签名为function(err, result)。对于这种函数，可以用简单直接的Q.nfcall和Q.nfapply调用这种Node.js风格的函数返回一个Promise：

return Q.nfcall(FS.readFile, "foo.txt", "utf-8");

return Q.nfapply(FS.readFile, ["foo.txt", "utf-8"]);

* 1
* 2
* 1
* 2

也可以用Q.denodeify或Q.nbind创建一个可重用的包装函数，比如：

var readFile = Q.denodeify(FS.readFile);

return readFile("foo.txt", "utf-8");

var redisClientGet = Q.nbind(redisClient.get, redisClient);

return redisClientGet("user:1:id");

* 1
* 2
* 3
* 4
* 5
* 1
* 2
* 3
* 4
* 5

第二个是then方法，这个方法是Promise对象的核心部件。我们可以用这个方法从异步操作中得到返回值（履约值），或抛出的异常（拒绝的理由）。then方法有两个可选的参数，分别对应Promis对象的两种执行结果。成功时调用的onFulfilled函数，错误时调用onRejected函数：

var promise = asyncFun()

promise.then(onFulfilled, onRejected)

* 1
* 2
* 1
* 2

Promise被解决时（异步处理已经完成）会调用onFulfilled 或onRejected 。因为只会有一种可能，所以这两个函数中仅有一个会被触发。尽管then方法的名字让人觉得它跟某种顺序化操作有关，并且那确实是它所承担的职责的副产品，但你真的可以把它当作unwrap来看待。Promise对象是一个存放未知值的容器，而then的任务就是把这个值从Promise中提取出来，把它交给另一个函数。

var promise = readFile()

var promise2 = promise.then(readAnotherFile, console.error)

* 1
* 2
* 1
* 2

这个promise表示 onFulfilled 或 onRejected 的返回结果。既然结果只能是其中之一，所以不管是什么结果，Promise都会转发调用:

var promise = readFile()

var promise2 = promise.then(function (data) {

return readAnotherFile() // if readFile was successful, let's readAnotherFile

}, function (err) {

console.error(err) // if readFile was unsuccessful, let's log it but still readAnotherFile

return readAnotherFile()

})

promise2.then(console.log, console.error) // the result of readAnotherFile

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8

因为then 返回的是Promise，所以promise可以形成调用链，从而避免出现回调大坑：

readFile()

.then(readAnotherFile)

.then(doSomethingElse)

.then(...)

* 1
* 2
* 3
* 4
* 1
* 2
* 3
* 4

再来看一下那个找最大文件的例子用Promise实现的样子：

var fs = require('fs')

var path = require('path')

var Q = require('q')

var fs\_readdir = Q.denodeify(fs.readdir) // [1]

var fs\_stat = Q.denodeify(fs.stat)

module.exports = function (dir) {

return fs\_readdir(dir)

.then(function (files) {

var promises = files.map(function (file) {

return fs\_stat(path.join(dir,file))

})

return Q.all(promises).then(function (stats) { // [2]

return [files, stats] // [3]

})

})

.then(function (data) { // [4]

var files = data[0]

var stats = data[1]

var largest = stats

.filter(function (stat) { return stat.isFile() })

.reduce(function (prev, next) {

if (prev.size > next.size) return prev

return next

})

return files[stats.indexOf(largest)]

})

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27

这时这个模块的用法变成了：

var findLargest = require('./findLargest')

findLargest('./path/to/dir')

.then(function (er, filename) {

console.log('largest file was:', filename)

})

.fail(function(err){

console.error(err);

});

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8

因为模块返回的是Promise，所以客户端也变成了Promise风格的，调用链中的所有异常都可以在这里捕获到。不过Q也有可以支持回调风格函数的nodeify方法。   
generators

generator科普   
在计算机科学中，generator实际上是一种迭代器。它很像一个可以返回数组的函数，有参数，可以调用，并且会生成一系列的值。然而generator不是把数组中的值都准备好然后一次性返回，而是一次yield一个，所以它所需的资源更少，并且调用者可以马上开始处理开头的几个值。简言之，generator看起来像函数，但行为表现像迭代器。

Generator也被称为半协程，是协程的一种特例（别协程弱），它总是把控制权交回给调用者（同时返回一个结果值），而不是像协程一样跳转到指定的目的地。如果要说得具体一点儿，就是虽然它们两个都可以yield多次，暂停执行并允许多次进入，但协程可以指定yield之后的去向，而generator不行，它只能把控制权交回给调用者。因为generator主要是为了降低编写迭代器的难度的，所以generator中的yield语句不是用来指明程序要跳到哪里去的，而是用来把值传回给父程序的。

2008年7月，Eich宣布开始ECMAScript Harmony（即ECMAScript 6）项目，从2011年7月开始，ECMAScript Harmony草案开始定期公布，预计到2014年12月正式发布。generator就是在这一过程中出现在ECMAScript中的，随后不久就被引入了V8引擎中。

Node.js对generator的支持是从v0.11.2开始的，但因为Harmony还没正式发布，所以需要指明–harmony或–harmony-generator参数启用它。我们用node –harmony进入REPL，创建一个generator：

function\* values() {

for (var i = 0; i < arguments.length; i++) {

yield arguments[i];

}

}

* 1
* 2
* 3
* 4
* 5
* 1
* 2
* 3
* 4
* 5

注意generator的定义，用的是像函数可又不是函数的function\*，循环时每次遇到yield，程序就会暂停执行。那么暂停后，generator何时会再次执行呢？在REPL中执行o.next()：

>var o = values(1, 2, 3);

> o.next();

{ value: 1, done: false }

> o.next();

{ value: 2, done: false }

> o.next();

{ value: 3, done: false }

> o.next();

{ value: undefined, done: true }

>

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10

第一次执行o.next()，返回了一个对象{ value: 1, done: false }，执行到第四次时，value变成了undefined，状态done变成了true。表现得像迭代器一样。next()除了得到generator的下一个值并让它继续执行外，还可以把值传给generator。有些文章提到了send()，不过那是老黄历了，也许你看这篇文章的时候，本文中也有很多内容已经过时了，IT技术发展得就是这样快。我们再看一个例子，还是在REPL中：

function\* foo(x) {

yield x + 1;

var y = yield null;

return x + y;

}

* 1
* 2
* 3
* 4
* 5
* 1
* 2
* 3
* 4
* 5

再次执行next()：

>var f = foo(2);

> f.next();

{ value: 3, done: false }

> f.next();

{ value: null, done: false }

> f.next(5);

{ value: 7, done: true }

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 1
* 2
* 3
* 4
* 5
* 6
* 7

注意最后的f.next(5)，value变成了7，因为最后一个next将5压进了这个generator的栈中，y变成了5。如果要总结一下，那么generator就是：

yield可以出现在任何表达式中，所以可以随时暂停执行，比如foo(yield x, yield y)或在循环中。   
调用generator只是看起来像函数，但实际上是创建了一个generator对象。只有调用next才会再次启动generator。next还可以向generator对象中传值。   
generator返回的不是原始值，而是有两个属性的对象：value和done。当generator结束时，done会变成true，之前则一直是false。   
可是generator和回调大坑有什么关系呢？因为yield可以暂停程序，next可以让程序再次执行，所以只需稍加控制，就能让异步回调代码顺序执行。

用generator平坑   
Node.js社区中有很多借助generator实现异步回调顺序化的库，比如suspend、co等，不过我们重点介绍的还是Q。它提供了一个spawn方法。这个方法可以立即运行一个generator，并将其中未捕获的错误发给Q.onerror。