# 一 Node与图片处理

<http://www.graphicsmagick.org/>

GraphicsMagick is the swiss army knife of image processing. 瑞士军刀。

只要服务器需要处理图片，那么这个服务器就要安装graphicsmagick软件。免费的。

装完之后，可视化工具一点用都没有，从桌面上删除。我们要把安装目录设置为环境变量。

控制台CMD命令：

|  |
| --- |
| 1. //格式转换 2. gm convert a.bmp a.jpg |

|  |
| --- |
| 1. //更改当前目录下\*.jpg的尺寸大小，并保存于目录.thumb里面 2. gm mogrify -resize 320x200 danny.jpg |

nodejs要使用graphicsmagick，需要npm装一个gm的包。

node.js命令

node.js缩略图的制作：

|  |
| --- |
| 1. var fs = require('fs'); 2. var gm = require('gm'); 3. gm('./danny.jpg') 4. .resize(50, 50,"!") 5. .write('./danny2.jpg', function (err) { 6. if (err) { 7. console.log(err); 8. } 9. }); |

node.js头像裁切：

|  |
| --- |
| 1. gm("./danny.jpg")**.crop(141,96,152,181)**.write("./2.jpg",function(err){ 2. **//141 96 是宽高 。 152 181是坐标** 3. }); |

# 二关于异步回调

回调通常用来处理一次性响应逻辑，比如数据库查询，可以指定一个回调函数来确定如何处理查询结果。事件监听器本质上也是一个回调，不同的是它和一个事件有关，如：

server.on(‘request’,functuon(){});

//自定义一个事件监听器：

## 1 异步回调机制

function Person() {  
 this.think = function (callback) {  
 setTimeout(function () {  
 console.log('Do thinking!');  
 callback();  
 },3000)  
 },  
 this.answer = function () {  
 console.log('Do answering!');  
 }  
}  
  
let p = new Person();  
p.think(function () {  
 console.log('After thinking 3s,get the answer');  
});  
p.answer();

这里的执行结果将会先展示 answer的结果，没有思考就开始回答问题！

Do answering!

Do thinking!

After thinking 3s,get the answer

因为think调用了异步函数setTimeout，因此think具备了异步非阻塞特性。

回调并非是异步调用，回调是一种解决异步函数执行结果的处理方法。所以并不是说所有的代码逻辑都是异步执行的，这取决于代码中是否应用了异步函数。

例如下列的回调函数并没有异步效果：

function waitThree(name,printName) {  
 let pus = 0;  
 let currentDate = new Date();  
 while (pus < 3000){  
 let now = new Date();  
 pus = now - currentDate;  
 }  
 printName(name); *//执行回调函数*}  
  
function getName(name) { *//定义回调函数* console.log(name);  
}  
  
waitThree('lisi',getName);  
console.log('over');

这里仅仅使用了while循环，并没有调用异步代码，所以仍然是阻塞式的调用。

## 2 异步函数的调用

思考问题：系统越过了异步函数直接执行后面的任务，那么如何获取异步函数的返回值？

示例代码：

function select(where,data) {  
 connector.collection(where,function (err,collection) {  
 collection.find(data,function (err,value) {  
 value.toArray(function (err,arr) {  
 return arr;  
 });  
 });  
 });  
 return false;  
}

这是一个和数据库有关的函数，使用collection函数获得集合，再使用find函数对集合结果转换，转换成功 返回 转换结果arr。

我们使用select函数来得到结果，要么是arr，要么是false。

但是由于collection find都是异步函数，这个select函数永远返回的是false。

解决方案一：失败

如果在回调函数中新增一个return的结果，仍然出现外部调用了select函数，异步函数刚执行，没得到结果，就调出函数了。

解决方案二：失败

在select内部创建一个变量存储异步的值，return这个变量。这里这个变量的值永远是null。

解决方案三：成功

首先，对于一个异步函数，我们必须传递一个回调函数来执行回调后的程序，上面的代码中function (err,value) 这个函数就是在异步代码执行结束后执行的回调函数程序。

我们的目的是调用select函数获取最终的成功与否的值，而select调用了异步函数，select本身也就成为了一个异步函数，获取异步调用返回结果的方法就是添加一个回调函数变量。

*//添加回调函数处理异步函数结果*function select(where,data,callback) {  
 connector.collection(where,function (err,collection) {  
 collection.find(data,function (err,value) {  
 value.toArray(function (err,arr) {  
 /\*  
 异步执行结果后，调用select传递的回调函数  
 将需要的数据作为callback的参数传递  
 \*/  
 callback(arr);  
 });  
 });  
 });  
 return false;  
}  
  
function getSelectArr(arr) {  
 console.log(arr);  
}  
select(where,data,getSelectArr);

通过getSelectArr就可以处理select返回的结果。

## 3 回调函数公式

假设A为异步函数，A有参数a和内部的回调函数function(b){} b就是A内部回调函数的返回结果。

函数C是一个封装函数，内部调用了A。

这时候我们如何在外部调用封装函数C的时候，获取到b值呢？

解决方案就是为C添加一个回调函数cb

function fnC(arg1,cb){

fnA(a,function(b){

cb(b);

}

}

function cb(arg){

console.log(arg);

}

执行获得结果：

fnc(arg,cb);

## 4 一个常见异步案例

异步执行for循环的异常

for(var i = 0; i < 10; i ++){  
 var m = i;  
 setTimeout(function () {  
 console.log(m);  
 },200);  
}

打印的结果全是9 ，而不是从0-9依次打印！

因为JS没有块级作用域！在异步函数中执行的最终结果将会出现上述代码中的bug。这是JS设计的极大败笔。

解决方案：同本笔记Node01中介绍的异步回调方式一样，我们需要得到异步函数结果，那么就用一个回调函数返回这个结果。

for(var i = 0; i < 10; i ++){  
 var m = i;  
 exec(m,function (m) {  
 console.log(m);  
 });  
}  
function exec(arg,callback) {  
 setTimeout(function () {  
 callback(arg)  
 },200);  
}

当然这在ES6中有更好的方式，let 带来了块级作用域：

for(let i = 0; i < 10; i ++){  
 let m = i;  
 setTimeout(function () {  
 console.log(m);  
 },200);  
}

## 5 回调地狱的解决

由上我们也可以看出，如果我们需要的结果来自一层一层的异步函数调用，最终会出现噩梦一样的回调地狱。

方案一：我们可以对深度回调内的每个异步函数进行封装，转化为多个函数之间的调用，但是这样仍然存在深度问题。

方案二：使用第三方库wind.js

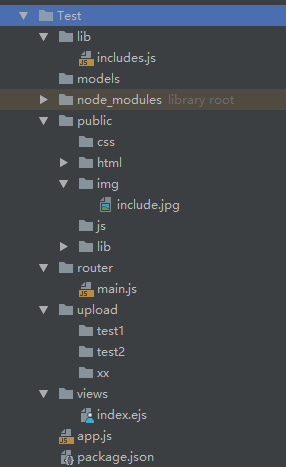
方案三：使用EventProxy

方案四：使用Step

最终方案，使用ES6规范下的 async await 但是只有Node7以上版本支持。

## 6 express异步回调实战

文件目录如下：



include.js

const fs = require('fs');  
exports.getArray = function (callback) {  
  
 fs.readdir(global.rootpath + '/upload',function (err,files) {  
 if(err){  
 callback('没有找到upload文件夹',null);  
 return;  
 } else {  
 callback(null,files);  
 return;  
 }  
 });  
};

main.js

let includes = require('../lib/includes');  
  
exports.showIndex = function (req,res,next) {  
 includes.getArray(function (err,arr) {  
 if(err){  
 console.log(err);  
 return;  
 } else {  
 res.render('index',{  
 imagesArray:arr  
 });  
 }  
 });  
};

app.js

const express = require('express');  
const main = require('./router/main');  
global.rootpath = \_\_dirname;  
  
let app = express();  
  
app.set('view engine','ejs');  
  
app.use('/test',express.static('public'));  
  
app.get('/',main.showIndex);  
  
app.listen(3000);

尤其注意：这里读取文件使用的是 绝对路径。

## 7 Node IO原理

Node的异步调用，其背后机制是Linux的Epoll，在windows下，则采用的是完成端口IOCP。这两种方式的共同点是没有启动任何其他线程。

Linux下，Node启动时，会维护一个线程池，Node使用这个线程池的线程，同步读取文件，在windows下利用IOCP通知机制，把代码交给操作系统的线程池运行。

# 三 单元测试

单元测试是指软件中的最小可测试单元进行检查和验证，又称为模块测试，用来保证程序中最小可用单元的可用性。在Node中，单元测试往往针对某个函数或者API进行正确性验证。

单元测试有许多风格，常见风格有行为驱动开发（BDD）和测试驱动开发（TDD）：

BDD：一种敏捷开发技术，鼓励软件项目中的开发者，QA和非技术人员或商业参与者之间的写作。也就是说：行为驱动开发关注的是整个系统的最终实现是否与用户期望一致。

TDD：一种软件开发过程中的应用方法，基本思想是先写测试程序，然后编码实现功能。测试驱动开发的目的是取得快速反馈，使所有功能都是可用的。

有两种常见的自动化测试：单元测试和验收测试。单元测试直接测试代码逻辑，通常是在函数层面，适用于所有类型的程序。验收测试用脚本控制浏览器，并试图用它触发web程序的功能，

## 1 Mocha基础使用

Mocha是最流行的Node单元测试框架，Mocha不仅可以运行在Node环境中，还可以运行在浏览器中。

使用前需要先安装：npm install mocha

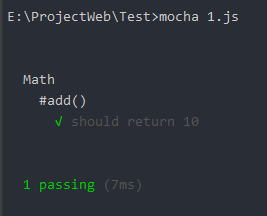
现在有一个普通的模块需要测试：

function add() {  
 if(arguments.length > 0){  
 return [].slice.call(arguments).reduce(function (a,b) {  
 return a + b;  
 });  
 } else {  
 return 0;  
 }  
}  
module.exports = {  
 add : add  
};

书写测试模块：

const lib = require('./2.js');  
  
*//测试描述*describe('Math',function () {  
 describe('#add()',function () {  
 it('should return 10', function () {  
 lib.add(5,5);  
 });  
 });  
});

在控制台使用mocha命令测试：



describe(); 该方法用来描述将要测试的模块，上述代码就是要测试Math模块下的add方法，这个方法可以嵌套。

it测试语句放在回调函数中，info是正确输出时的简单语句描述，一个it对应一个可能的情况。

此时只是使用了Mocha，并没有对结果进行检查，Node自带断言库assert，对add函数进行断言测试可以使用assert.equal()的方法：

const lib = require('./2.js');  
const assert = require('assert');  
*//测试描述*describe('Math',function () {  
 describe('#add()',function () {  
 it('should return 10', function () {  
 *// lib.add(5,5);* assert.equal('10',lib.add(5,5));  
 });  
 it('should return 0', function () {  
 *// lib.add();* assert.equal('0',lib.add());  
 });  
 });  
});

## 2 测试异步

const lib = require('./2.js');  
describe('Async',function () {  
 describe('#setTimeout()',function () {  
 it('should wait 100ms', function (done) {  
 lib.test(done);  
 });  
 });  
});

运行 mocha \*\*.js -c

## 3 路由测试

Mocha并没有对路由接口的测试支持，需要借助supertest库，

## 4 测试覆盖率

覆盖率一般包含行覆盖率、函数覆盖率、分支覆盖率、语句覆盖率四个维度。在Node中，可以使用Istanbul这个工具作为代码覆盖率工具。（需要npm安装）。

测试命令：is

tanbul cover 1.js

结合mocha测试覆盖率：

istanbul cover \_mocha

这条命令生成了一个coverage文件夹。数据报告存储在该文件夹中。

# 四 Node与多核CPU

Node采用了单线程模型，但是并不能说明Node只能运行在单核CPU下。Node原生已经支持集群特征。

制作一个简单HelloWorld示例，我们将应用程序部署在单台服务器的多核CPU上，从而搭建集群环境。也就是说：每个CPU内核都会运行一个Node进程，这样的集群只能在单台服务器上运行。

const http = require('http');  
const cluster = require('cluster');  
const os = require('os');  
  
let PORT = 8000;  
let CPUS = os.cpus().length; *//获取cpu内核书*if(cluster.isMaster){ *//当前进程为主进程* for(let i = 0; i < CPUS; i++){  
 cluster.fork();  
 }  
} else { *//当前进程为子进程:只有在子进程才执行相关代码* let app = http.createServer(function (req,res) {  
 res.writeHead(200,{'Content-Type':'text/plain'});  
 res.end('hello world');  
 }).listen(PORT,function () {  
 console.log('server is running at' + PORT);  
 });  
}

实际上，程序内部是通过主进程去调度各个子进程的。即：cpu核心数越多，单台服务器上可创建的子进程越多，支持的并发越大，从而整个应用程序的吞吐率就越高。

# 五 微服务网关

## 1 简介

微服务网关将底层复杂的细节进行评比，对外提供简单统一的调用方式，比如HTTP方式。对客户端而言，可以是PC端网页，也可以使移动端设备，客户端通过HTTP方式调用微服务网关。

微服务网关也称为服务网关（Service Gateway）或者API网关（API Gateway）。

微服务网关是微服务架构中的核心组件，是客户端请求的门户，也是调用具体服务端的桥梁。

服务网关的路由过程，我们称之为反向代理，说白了就是请求不会直接发送到目的地，而是通过一个中间件（Nginx或Apache）来转发。

使用反向代理技术的业务场景：

1 静态资源与动态资源分离

2 实现Ajax跨域

3 搭建统一的服务网关接口

## 2 Node搭建反向代理服务器

需要安装第三方模块：http-proxy

let http = require('http');  
let httpproxy = require('http-proxy');  
  
let PORT = 8000;  
  
*//创建代理服务器对象并监听错误事件*let proxyServer = httpproxy.createProxyServer();  
proxyServer.on('error',function (err,req,res) {  
 res.end(); *//输出空白格响应数据*});  
  
let app = http.createServer(function (req,res) {  
 *//执行反向代理* proxyServer.web(req,res,{  
 target: 'http://localhost:8080' *//目标地址* });  
}).listen(PORT,function () {  
 console.log('server is running at ' + PORT);  
});

这时候我们就可以通过

http://localhost:8000

来访问 http://localhost:8080

我们可以通过Apache Bench来测试性能。

Node的性能绝不亚于Nginx，且扩展性远高于Nginx，通过Node可以动态置顶被代理的目标地址，而在Nginx中目标地址是静态的。这对于微服务中实现 服务发现 功能是极其重要的。因为我们需要送网关中获取需要代理的微服务信息（如IP和端口），并执行反向代理操作，调用响应的微服务REST API。

服务网关并非仅提供反向代理和服务发现特性，还具备安全认证、性能监控、数据缓存、请求分片、静态响应等众多特性，可根据业务需求进行扩展。

## 3 Node连接ZooKeeper

需要安装第三方模块：node-zookeeper-client

let zookeeper = require('node-zookeeper-client');  
let CONNECTION\_STRING = 'localhost:8000';  
let OPTIONS = {  
 sessionTimeout:5000  
};  
  
let zk = zookeeper.createClient(CONNECTION\_STRING,OPTIONS);  
zk.on('connected',function () {  
 console.log(zk);  
 zk.close();  
});  
zk.connect();

当触发connected事件时，说明客户端已经成功连接ZooKeeper服务器，若zk可以正常输出，则说明成功连接，下面所有的操作都在该会话中进行，只需要调用zk对象相关的API即可。

## 4 Node实现服务发现

服务发现组件在微服务架构中由服务网关提供支持，前端发送的HTTP请求首先会进入服务网关，此时服务网关将从服务注册表中获取当前可用服务所对应的具体服务配置。随后通过反向代理技术调用具体的服务，像这样获取可用服务配置的过程称为服务发现。

const express = require('express');  
const swig = require('swig');  
const bodyParse = require('body-parser');  
const zookeeper = require('node-zookeeper-client');  
const httpProxy = require('http-proxy');  
  
let CONNECTION\_STRING = '127.0.0.1:2181';  
let REGISTRY\_ROOT = '/api';  
let PORT = 8000;  
let OPTIONS = {  
 sessionTimeout:5000  
};  
  
*//连接ZK*let zk = zookeeper.createClient(CONNECTION\_STRING,OPTIONS);  
zk.connect();  
  
*//创建代理服务器并监听错误事件*let proxy = httpProxy.createProxyServer();  
proxy.on('error',function (err,req,res) {  
 res.end(); *//输出空白响应数据，防止浏览器假死*});  
  
*//启动app*let app = express();  
app.engine('html',swig.renderFile);  
app.set('view engine','html');  
swig.setDefaults({cache:false});  
*//静态资源管理*app.use('/public',express.static(\_\_dirname + '/public'));  
*//POST解析*app.use(bodyParse.urlencoded({extended: true}));  
  
  
app.all('\*',function (req,res) {  
  
 *//处理图标* if(req.path == '/favicon.ico'){  
 res.end();  
 return;  
 }  
  
 *//获取服务名称* let serviceName = req.get('Service-Name');  
 console.log('servicename is ' + serviceName);  
 if(!serviceName){  
 console.log('Service-Name request header is not exist');  
 res.end();  
 return;  
 }  
  
 *//获取服务路径* let servicePath = REGISTRY\_ROOT + '/' + serviceName;  
 zk.getChildren(servicePath,function (error,addressNodes) {  
 if(error){  
 console.log(error.stack);  
 res.end();  
 return;  
 }  
 let size = addressNodes.length;  
 if(size == 0){  
 console.log('address node is not exist');  
 res.end();  
 return;  
 }  
 *//生成地址路径* let addressPath = servicePath + '/';  
 if(size == 1){ *//只有一个地址，则获取该地址* addressPath = addressNodes[0];  
 } else { *//多个地址，则随机取一个地址* addressPath += addressNodes[parseInt(Math.random()) \* size];  
 }  
  
 *//获取服务地址* zk.getDate(addressPath,function (error,serviceAddress) {  
 if(error){  
 console.log(error.stack);  
 res.end();  
 return;  
 }  
 if(!serviceAddress){  
 console.log('service address is not exist');  
 res.end();  
 return;  
 }  
  
 *//执行反向代理* proxy.web(req,res,{  
 target: 'http://' + serviceAddress *//目标地址* });  
  
 });  
 });  
  
});  
  
  
app.use('/api',require('./routers/api/testapi'));  
app.use('/',require('./routers/main'));  
  
app.listen(PORT,function () {  
 console.log('server is running on ' + PORT);  
});