# Koa框架技术分享

## 写在前面：

Koa使用了ES6规范的generator和异步编程是一个更轻量级Web开发的框架，Koa 的先天优势在于 generator。之前我们的综合监控系统我们一直在用，却都没有了解它到底是怎么工作的，这次主要讲一讲Koa框架的工作流程，由于是我个人的分享交流，所以Node基础、ES6标准、Web开发基础以及Koa的"Hello World"程序都不在讨论，希望各位小伙伴提出意见和指导。

PS：Koa 内核中没有捆绑任何中间件，但不用担心，Koa 拥有极其强悍的拓展性，正文所有中间件都可以在npm官网下载安装，但国内域名安装会有一些限制，提供一个国内镜像安装方法，速度非常快，在直接npm模块失败的时候非常好用，使用npm --registry=http://registry.npmjs.org install XXXXX –XX 命令安装，只需要在install后面加上要安装的中间件名称和相应的参数即可。

## 一、使用Koa搭建Web项目流程

### 1、Koa项目创建：

个人认为不管任何框架，Web项目搭建必需的几个方面，页面、中间件、路由、会话和存储、日志、静态文件指定，以及错误的处理。当然，网站开发不止这些东西，还有许多主题，比如实时通讯，搜索引擎架构，权限控制，邮件优先队列，日志记录分析，对Web开发还刚刚入门属于菜鸟级别，这里就不做深入的讨论了。了解Express框架的小伙伴一定知道Express的部署过程，不管是通过express-generator生成还是WebStorm等编译器直接创建，它的目录结构大概是这样的：

|——app.js

|——bin

|——node\_modules

|——package.json

|——public

|——routes

|——views

\*app.js，是程序启动文件

\*bin，存放执行程序

\*node\_modules，存放项目依赖库

\*package.json，是配置和一些相关信息

\*public，存放静态文件（css，js，img）

\*routes，存放路由文件

\*views，存放前台页面文件

这些结构基本包含了上述提到的Web项目搭建的要素，但是目前类似express-generator的Koa部署工具Koa-generator（非官方）并不完善并且个人测试存在些许错误。其实Koa-generator也是仿造上述express-generator生成的目录，既然这样还不如手动创建目录来的爽快（generator-k是另一款生成器，用上去感觉还行），在根目录新建app.js作为程序的启动文件，创建三个文件夹分别命名public、routes和views，最后新建package.json文件存放你的项目的一些信息。完成这些创建之后，用npm命令安装Koa，这样的话一个基本的Koa框架就搭建好了，非常的的轻量级，它的目录结构如下：

|——app.js

|——node\_modules

|——public

| |——img

| |——css

| |——js

|

|——routes

| |——index.js

| |——user.Js

|

|——views

| |——\_layout.html

| |——index.html

|

|——package.json

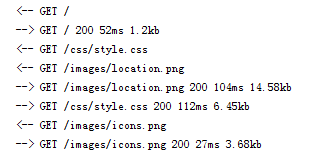
Koa项目运行：node --harmony app.js

必须加 --harmony ，这样才会支持 ES6 语法。

### 2、Koa日志：

日志是项目error调试和日常维护的基本手段，Koa有日志模块Koa-logger，npm install Koa-logger后使用app.use(logger());命令程序就会在控制台自动打印日志，当然如果你对Koa-logger的风格不满意或者想要看到更多得信息也可以自己编辑代码实现有自己风格的日志打印。

例如：



最后呢，如果有需要，要把日志进行存储。

### 3、Koa的错误处理

Koa 有 error 事件，当发生错误时，可以通过该事件，对错误进行统一的处理。

var Koa = require('Koa');

var app = Koa();

app.on('error', function(err,ctx){

console.log(err);

});

app.listen(3000);

上面这段代码在如果捕获到错误，页面会打印出 “Internal Server Error” （这是Koa对错误的默认处理）。这个错误我们在综合监控系统中也经常见到，那么我们显然无法根据这条日志得到什么信息



这些错误信息是怎么报出来的的呢，其实是Koa-onerror 中间件，它优化错误信息，根据这些错误信息就能更好的捕获到错误。

Koa-onerror使用方法：

var onerror = require('Koa-onerror');

onerror(app);

### 4、Koa静态文件指定：

Koa静态文件指定中间件Koa-static，npm install Koa-static之后就可以使用Koa-static负责托管 Koa 应用内的静态资源。映射了静态文件目录，引用的时候直接去该目录下寻找资源，会减少一些消耗。（不知道讲的准确不准确，只是个人的理解）指定public为静态文件目录的代码如下：

var staticServer = require('Koa-static');

var path = require('path');

app.use(staticServer(path.join(\_\_dirname,'public')));

### 5、ejs模板的使用：

渲染页面需要一种模板，这里选择风格接近html的ejs模板。npm install Koa-ejs后就可

以在Koa框架中使用ejs模版。

var render = require('Koa-ejs');

render(app, {

root: path.join(\_\_dirname, 'views'),

layout: '\_\_layout',

viewExt: 'html',

cache: false,

debug: true

});

app.use(function \*(){

yield this.render('index',{layout:false});

});

### 6、Koa路由设置：

Koa个极简的web框架，简单到连路由模块都没有配备。自己手写路由是这样的：

app.use(function \*(){

//我是首页

if(this.path==='/'){

}

});

使用更加强大的路由中间件，Koa中设置路由一般安装Koa-router，Koa-router支持五种方法

router.get()

router.post()

router.put()

router.del()

router.patch()

GET方法举例：

var app = require('Koa')();

var Router = require('Koa-router');

var myRouter = new Router();

myRouter.get('/', function \*(next) {

yield this.render('index',{layout:false});

});

app.use(myRouter.routes());

app.listen(3000);

Koa-router 拥有丰富的 api 细节，用好这些 api ，可以让页面代码更为优雅与可维护。

接收query参数

http://localhost:3000/?a=1(条件)

index.js

var router = require('Koa-router')();

router

.get('/',function \*(next){

console.log(this.query);

yield this.render('index',{layout:false});

})

.get('/home',function \*(ctx,next){

ctx.render('home');

});

//ctx为Koa2.0中支持

... ...

module.exports = router;

控制台打印：

<-- GET /?a=1

{ a: '1' }

{ a: '1' }

接收params参数

http://localhost:3000/users/123（参数）

router.get('/user/:id', function \*(next) {

console.log(this.params.id);

});

param() 用于封装参数处理中间件，当访问 /detail/:id 路由时，会先执行 param() 定义的 generator function 逻辑。

函数的第一个是路由参数的值，next 是中间件流程关键标识变量。

yield next;

表示执行下一个中间件。

app.param('id',function \*(id,next){

this.id = Number(id);

if ( typeof this.id != 'number') return this.status = 404;

yield next;

}).get('/detail/:id', function \*(next) {

//我是详情页面

var id = this.id; //123

this.body = id;

});

### 7、Koa中间件：

Koa的中间件很像Express的中间件，也是对HTTP请求进行处理的函数，但是必须是一个Generator函数即 function \*(){} 语法，不然会报错。可以这么说，Nodejs的Web程序中任何请求和响应都是中间件在操作。

app

.use(logger()) //日志中间件

.use(serve(\_\_dirname + '/public')) //静态文件指定中间件

.use(router.routes()) //路由中间件

.use(router.allowedMethods()); //路由中间件

app.use 加载用于处理http请求的middleware（中间件），当一个请求来的时候，会依次被这些 middlewares处理。执行的顺序是你定义的顺序。中间件的执行顺序规则是类似“栈”的结构，所有需要执行的中间件都被一个一个放入“栈”中，当没有遇到next()的时候，“栈”里边的这些中间件被逆序执行。

app.use(function \*(next){

this; // is the Context

this.request; // is a Koa Request

this.response; // is a Koa Response

});

说明：

•this是上下文

•\*代表es6里的generator

http模型里的请求和响应

•this.request

•this.response

app.use() 究竟发生了什么不可思议的化学反应呢？

其实 app.use() 就干了一件事，就是将中间件放入一个数组，真正执行逻辑的是：app.listen(3000);

Koa 的 listen() 除了指定了 http 服务的端口号外，还会启动 http server，等价于：

var http = require('http');

http.createServer(app.callback()).listen(3000);

后面这种繁琐的形式有什么用呢？

一个典型的场景是启动 https 服务，默认 app.listen(); 是启动 http 服务，启动 https 服务就需要：

var https = require('https');

https.createServer(app.callback()).listen(3000);

## 二、异步编程

### 1、异步流程控制：

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

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。

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？

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。异步编程的语法目标，就是怎样让它更像同步编程。

Koa 的先天优势在于 generator。

generator指的是

function\* xxx(){

}

是es6里的写法。

fvar r = 3;

function\* infinite\_ap(a) {

for( var i = 0; i < 3 ; i++) {

a = a + r ;

yield a;

}

}

var sum = infinite\_ap(5);

console.log(sum.next()); // returns { value : 8, done : false }

console.log(sum.next()); // returns { value : 11, done: false }

console.log(sum.next()); // returns { value : 14, done: false }

console.log(sum.next()); //return { value: undefined, done: true }

yield语句就是暂停标志，next方法遇到yield，就会暂停执行后面的操作，并将紧跟在yield后面的那个表达式的值，作为返回对象的value属性的值。当下一次调用next方法时，再继续往下执行，直到遇到下一个yield语句。如果没有再遇到新的yield语句，就一直运行到函数结束，将return语句后面的表达式的值，作为value属性的值，如果该函数没有return语句，则value属性的值为undefined。

当第一次调用 sum.next() 时 返回的a变量值是5 + 3，同理第二次调用 sum.next() ，a变量值是8 +3，知道循环执行结束，返回done:true标识。

大家有没有发现个问题，Koa 中 generator 的用法与上述 demo 演示的用法有非常大得差异，那是因为 Koa 中的 generator 使用了 co 进行了封装。

### 2、co的使用：

Ps:(这里只是简单介绍，后续可以作为一个专题来讲)

co 函数库是著名程序员 TJ Holowaychuk 于2013年6月发布的一个小工具，用于 Generator 函数的自动执行。

比如，有一个 Generator 函数，用于依次读取两个文件。

var gen = function\* (){

var f1 = yield readFile('/etc/fstab');

var f2 = yield readFile('/etc/shells');

console.log(f1.toString());

console.log(f2.toString());

};

co 函数库可以让你不用编写 Generator 函数的执行器。

var co = require('co');

co(gen);

上面代码中，Generator 函数只要传入 co 函数，就会自动执行。

co 函数返回一个 Promise 对象，因此可以用 then 方法添加回调函数。

co(gen).then(function (){

console.log('Generator 函数执行完成');

})

上面代码中，等到 Generator 函数执行结束，就会输出一行提示。

为什么 co 可以自动执行 Generator 函数？

前面文章说过，Generator 函数就是一个异步操作的容器。它的自动执行需要一种机制，当异步操作有了结果，能够自动交回执行权。

两种方法可以做到这一点。

（1）回调函数。将异步操作包装成 Thunk 函数，在回调函数里面交回执行权。

（2）Promise 对象。将异步操作包装成 Promise 对象，用 then 方法交回执行权。

co 函数库其实就是将两种自动执行器（Thunk 函数和 Promise 对象），包装成一个库。使用 co 的前提条件是，Generator 函数的 yield 命令后面，只能是 Thunk 函数或 Promise 对象。

参考：http://www.ruanyifeng.com/blog/2015/05/co.html

### 3、Koa 中间件机制实现原理：

使用 Koa 的同学一定会有如下疑问：

1. Koa 的中间件机制是如何实现？
2. 为什么中间件必须是 generator function？
3. next 实参指向是什么？为什么可以通过 yield next 可以执行下一个中间件？
4. 为什么中间件从上到下执行完后，可以从下到上执行 yield next 后的逻辑？

通过实现简单的 Koa 框架（剥离除中间件外所有的逻辑）来解答上述问题，这个框架的名字叫 SimpleKoa：

**var *co*** = require(**'co'**);  
**function** *SimpleKoa*(){  
 **this**.**middlewares** = [];  
}  
*SimpleKoa*.**prototype** = {  
 *//注入个中间件* use: **function**(gf){  
 **this**.**middlewares**.push(gf);  
 },  
 *//执行中间件* listen: **function**(){  
 **this**.\_run();  
 },  
 \_run: **function**(){  
 **var** ctx = **this**;  
 **var** middlewares = ctx.**middlewares**;  
 **return *co***(**function** \*(){  
 **var** prev = **null**;  
 **var** i = middlewares.**length**;  
 *//从最后一个中间件到第一个中间件的顺序开始遍历* **while** (i--) {  
 *//实际Koa的ctx应该指向server的上下文，这里做了简化  
 //prev 将前面一个中间件传递给当前中间件* prev = middlewares[i].call(ctx, prev);  
 }  
 *//执行第一个中间件* **yield** prev;  
 })();  
 }  
};

写个 demo 印证下中间件执行顺序：  
**var *app*** = **new** SimpleKoa();  
***app***.use(**function** \*(next){  
 **this**.**body** = **'1'**;  
 **yield** next;  
 **this**.**body** += **'5'**;  
 **console**.log(**this**.body);  
});  
***app***.use(**function** \*(next){  
 **this**.**body** += **'2'**;  
 **yield** next;  
 **this**.**body** += **'4'**;  
});  
***app***.use(**function** \*(next){  
 **this**.**body** += **'3'**;  
});  
***app***.listen();  
 执行后控制台输出：123456，对照 Koa 中间件执行顺序，完全一致！寥寥几行代码，我们就实现了 Koa 的中间件机制！这就是 co 的魔力。

## 三、Koa中涉及但本次没有讲的问题

### 1、Koa中的cookie和session（后续卢毅详细讲解）：

web应用程序都离不开cookie和session的使用，是因为Http是一种无状态性的协议。保存用户状态信息的一种方法或手段，Session 与 Cookie 的作用都是为了保持访问用户与后端服务器的交互状态。

### 2、Koa中nosql（后续技术分享会详细讲解）

mongodb是一个基于文档的非关系型数据库，所有数据是从磁盘上进行读写的，其优势在于查询功能比较强大，能存储海量数据。

redis是内存型数据库，数据保存在内存中，通过tcp直接存取，优势是速度快，并发高，缺点是数据类型有限，查询功能不强，一般用作缓存。它由C语言实现的，与 NodeJS工作原理近似，同样以单线程异步的方式工作，先读写内存再异步同步到磁盘，读写速度上比MongoDB有巨大的提升，当并发达到一定程度时，即可考虑使用Redis来缓存数据和持久化Session。

var mongoose = require('mongoose');

// 引入 mongoose 模块

mongoose.connect('mongodb://localhost/blog');

// 然后连接对应的数据库：mongodb://localhost/test

// 其中，前面那个 mongodb 是 protocol scheme 的名称；localhost 是 mongod 所在的地址；

// 端口号省略则默认连接 27017；test 是数据库的名称

// mongodb 中不需要建立数据库，当你需要连接的数据库不存在时，会自动创建一个出来。

module.exports = mongoose;

// 导出 mongoose 模块

var mongoose = require('../modules/db');

// 引入 mongoose 模块

var User = mongoose.model('User',{

name: {type: String, match: /^(\w)+(\.\w+)\*@(\w)+((\.\w+)+)$/},

password: String

});

//创建了一个名为 User 的 model

var user1 = new User({name:'196887298@qq.com'});

user1.password = 'a5201314';

user1.save(function(err){

if(err){

console.log("save successful");

}

});