目录

[一、 快速入门 2](#_Toc68553813)

[1. 使用脚手架快速创建项目 2](#_Toc68553814)

[2. 启动项目 2](#_Toc68553815)

[3. 服务器基本信息配置 2](#_Toc68553816)

[4. 静态资源托管 2](#_Toc68553817)

[二、 基础功能 3](#_Toc68553818)

[1. 框架约定目录规则 3](#_Toc68553819)

[2. 内置对象 3](#_Toc68553820)

[3. 路由Router 4](#_Toc68553821)

[4. 控制器Controller 4](#_Toc68553822)

[5. 获取提交的数据 5](#_Toc68553823)

[6. Post提交报csrf错误 6](#_Toc68553824)

[7. 获取上传的文件 6](#_Toc68553825)

[8. Cookie 7](#_Toc68553826)

[9. Session 8](#_Toc68553827)

[10. 调用service 8](#_Toc68553828)

[11. 发送HTTP响应 9](#_Toc68553829)

[三、 MySQL数据库操作 10](#_Toc68553830)

[四、 Egg跨域 11](#_Toc68553831)

[五、 验证码 12](#_Toc68553832)

**Egg服务器架构文档**

Egg为企业级框架和应用而生，奉行【约定高于配置】，按照一套统一的约定进行开发，减少团队沟通交流成本，简直太棒了。

## 快速入门

### 使用脚手架快速创建项目

$ mkdir egg-example && cd egg-example  
$ npm init egg --type=simple  
$ cnpm i

### 启动项目

$ npm run dev  
$ open http://localhost:7001

### 服务器基本信息配置

*// config/config.default.js*  
*// 配置服务器基本信息*

  config.cluster = {

    listen: {

      path: '',

      port: 8000,

      hostname: 'admin.jianmian.com',//默认localhost和ip地址,上线时用0.0.0.0

    }

  };

**Hostname是域名的相关配置**

这里是模拟域名，如果你注册了真实的域名并进行了域名解析就不用配置了。

*// c:\windows\system32\drivers\etc\hosts 增加*

192.168.3.114 admin.jianmian.com  
##ip domain

这样你就可以使用 http://admin.jianmian.com:8000 来访问了。

### 静态资源托管

Egg内置的static插件默认把访问路径/public/\* 映射到 app/public/\*目录，我们只需要把html、css、js、图片、多媒体等静态资源放到app/public/目录即可；

app/public

├──my.html  
├── css  
│ └── news.css  
└── js  
 ├── lib.js  
 └── news.js

至此，一个简单的网站就创建好了。

## 基础功能

### 框架约定目录规则

* 1. app/router.js：用于配置URL路由规则；
  2. app/controller/\*\* ：用于解析用户的输入，处理后返回相应的结果；
  3. app/service/\*\*： 用于编写业务逻辑层；
  4. app/public/\*\*： 用于放置静态资源；
  5. config/config.{env}.js： 用于编写配置文件；
  6. config/plugin.js 用于配置需要加载的插件；

### 内置对象

* 1. Application：全局应用对象，在一个应用中，只会实例化一个对象；

在继承于 Controller, Service 基类的实例中，可以通过 this.app 访问到 Application 对象。

* 1. Request & Response：可以在 Context 的实例上获取到当前请求的 Request(ctx.request) 和 Response(ctx.response) 实例；
  2. Controller：推荐所有的 [Controller](https://eggjs.org/zh-cn/basics/controller.html) 都继承于该基类实现。该基类属性有：

ctx - 当前请求的 [Context](https://eggjs.org/zh-cn/basics/objects.html#context) 实例。

app - 应用的 [Application](https://eggjs.org/zh-cn/basics/objects.html#application) 实例。

service - 应用所有的 [service](https://eggjs.org/zh-cn/basics/service.html)。

* 1. Service：推荐所有的Service都继承该基类。

Service基类属性和 Controller 基类属性一致。

### 路由Router

路由是描述请求URL和具体承担执行动作的Controller的对应。说的直白点，就是用户访问不同的路径时应该有不同的Controller去响应不同的内容。

  router.get('/getdata', controller.form.get);

  router.post('/pdata', controller.form.post);

  router.get('/user/:userid', controller.user.info);

### 控制器Controller

* 1. 控制器的定义以及和路由的关联

Controller负责解析用户的输入，处理后返回响应的结果。所有的Controller 文件都必须放在 app/controller目录下，支持多级目录，访问时可以通过目录名级联访问。如将Controller代码放到 app/controller/sub/post.js 中，则可以在 router 中这样使用：

*// app/router.js*  
module.exports = app => {  
 app.router.post('createPost', '/api/posts', app.controller.sub.post.create);  
}

同时，我们也可以自定义基类给控制器继承，官方案例如下：

*// app/core/base\_controller.js*  
**const** { Controller } = require('egg');  
**class** **BaseController** **extends** **Controller** {  
 **get** user() {  
 **return** **this**.ctx.session.user;  
 }  
  
 success(data) {  
 **this**.ctx.body = {  
 success: true,  
 data,  
 };  
 }  
  
 notFound(msg) {  
 msg = msg || 'not found';  
 **this**.ctx.throw(404, msg);  
 }  
}  
module.exports = BaseController;

定义控制器继承他：

*//app/controller/post.js*  
**const** Controller = require('../core/base\_controller');  
**class** **PostController** **extends** **Controller** {  
 **async** list() {  
 **const** posts = **await** **this**.service.listByUser(**this**.user);  
 **this**.success(posts);  
 }  
}

### 获取提交的数据

接收GET请求的数据：ctx.request.query 和 ctx.query.id；

接收POST请求的数据：ctx.request.body，**而不是** ctx.body；

Post数据默认大小是100kb，如需调整可在 config/config.default.js 中覆盖框架的默认值：

module.exports = {  
 bodyParser: {  
 jsonLimit: '1mb',  
 formLimit: '1mb',  
 },  
};

接收路由参数：

*// app.get('/projects/:projectId/app/:appId', 'app.listApp');*  
*// GET /projects/1/app/2*  
**class** **AppController** **extends** **Controller** {  
 **async** listApp() {  
 assert.equal(**this**.ctx.params.projectId, '1');  
 assert.equal(**this**.ctx.params.appId, '2');  
 }  
}

### Post提交报csrf错误

post请求时，会有安全验证问题，简单的处理方式是关闭安全验证：

*// config/config.default.js*

*// 配置安全验证*

config.security = {

csrf: {

enable: false,

ignoreJSON: true,

}

}

### 获取上传的文件

记住了，你要先在 config 文件中启用 file 模式：

*// config/config.default.js*  
exports.multipart = {  
 mode: 'file',  
};

然后，你就可以参考下面的代码了：

<form action="/upload" method="post" enctype="multipart/form-data">

        <input type="file" name="avatar" id="avatar" multiple>

        <input type="file" name="avatar1" id="avatar">

        <input type="file" name="avatar2" id="avatar" multiple>

        <input type="submit" value="上传">

    </form>

这里的参考代码只处理一张图片，多张图片循环处理就可以了：

class UploadController extends Controller {

    async file() {

        const { ctx } = this;

        const dest = '/public/upload/';

        const file = ctx.request.files[0];

        console.log(ctx.request.files);

        let to = path.dirname(\_\_dirname) + dest + path.basename(file.filepath);

*// 处理文件，比如上传到云端  或 放到指定的目录*

        await fs.copyFileSync(file.filepath, to);

        fs.unlinkSync(file.filepath);

        console.log(dest);

*// 返回图片路径*

        let cluster = this.app.config.cluster.listen;

        ctx.body = `http://${cluster.hostname}:${cluster.port}${dest}${path.basename(file.filepath)}`;

    }

}

### Cookie

通过 ctx.cookies可以在 Controller 中便捷、安全的设置和读取 Cookie。

**class** **CookieController** **extends** **Controller** {  
 **async** add() {  
 **const** ctx = **this**.ctx;  
 **let** count = ctx.cookies.get('count');  
 count = count ? Number(count) : 0;  
 ctx.cookies.set('count', ++count);  
 ctx.body = count;  
 }  
  
 **async** remove() {  
 **const** ctx = **this**.ctx;  
 **const** count = ctx.cookies.set('count', null);  
 ctx.status = 204;  
 }  
}

需要注意的是，cookie默认不支持中文，可以尝试转码，如encodeURI('中文egg')，然后再转回来decodeURI(ctx.cookies.get('username'))；也可以通过加密的方式处理。

清除cookie把值设置为null即可。

在设置cookie时有个对象类型的可选参数，可以对cookie进行相关设置：

maxAge： 设置cookie的有效期，单位毫秒，默认浏览器关闭消失；

httpOnly：设置cookie是否允许js访问，默认true，不允许；

overwrite：如果设置为true，相同的键值对会被覆盖，否则发送两个；

signed：如果为true表示对cookie进行签名，不是加密，只是防止被篡改，注意在获取的时候也要提供该设置进行匹配；

encrypt：是否加密，true加密后客户端看不到明文，只能在服务器端获取，注意在获取的时候也要提供该设置进行匹配；

### Session

Session 的使用方法非常直观，直接读取或者修改就可以，如果要删除它，直接将它赋值为 null：

**class** **SessionController** **extends** **Controller** {  
 **async** deleteSession() {  
 **this**.ctx.session = null;  
 }  
};

**注意**：设置 session 属性时不要以 \_ 开头，不能为 isNew

Session默认配置如下：

exports.session = {  
 key: 'EGG\_SESS',  
 maxAge: 24 \* 3600 \* 1000, *// 1 天*  
 httpOnly: true,  
 encrypt: true,  
};

也可以针对性设置有效期：

*// 如果用户勾选了 `记住我`，设置 30 天的过期时间*  
**if** (rememberMe) ctx.session.maxAge = ms('30d');

重置session的有效期：当用户 Session 的有效期仅剩下最大有效期一半的时候

*// config/config.default.js*  
module.exports = {  
 session: {  
 renew: true,  
 },  
};

### 调用service

在 Controller 中可以调用任何一个 Service 上的任何方法，同时 Service 是懒加载的，只有当访问到它的时候框架才会去实例化它。

*// 调用 service 进行业务处理*  
**const** res = **await** ctx.service.post.create(req);

### 发送HTTP响应

* + 1. 设置status

*// 设置状态码为 201*  
**this**.ctx.status = 201;

* + 1. 设置body

**ctx.body 是 ctx.response.body 的简写，不要和 ctx.request.body 混淆了；**

*// 响应内容*  
**this**.ctx.body = '<html><h1>Hello</h1></html>';

* + 1. JSONP

app.jsonp() 提供的中间件来让一个 controller 支持响应 JSONP 格式的数据。在路由中，我们给需要支持 jsonp 的路由加上这个中间件：

*// app/router.js*  
module.exports = app => {  
 **const** jsonp = app.jsonp();  
 app.router.get('/api/posts/:id', jsonp, app.controller.posts.show);  
 app.router.get('/api/posts', jsonp, app.controller.posts.list);  
};

在 Controller 中，只需要正常编写即可。用户请求对应的 URL 访问到这个 controller 的时候，如果 query 中有 \_callback=fn 参数，将会返回 JSONP 格式的数据，否则返回JSON格式的数据。

**可配置：**

*// config/config.default.js*  
exports.jsonp = {  
 callback: 'callback', *// 识别 query 中的 `callback` 参数*  
 limit: 100, *// 函数名最长为 100 个字符*  
};

* + 1. 重定向
* ctx.redirect(url) 如果不在配置的白名单域名内，则禁止跳转。
* ctx.unsafeRedirect(url) 不判断域名，直接跳转，一般不建议使用，明确了解可能带来的风险后使用。

用户如果使用ctx.redirect方法，需要在应用的配置文件中做如下配置：

*// config/config.default.js*  
exports.security = {  
 domainWhiteList:['.domain.com'], *// 安全白名单，以 . 开头*  
};

若用户没有配置 domainWhiteList 或者 domainWhiteList数组内为空，则默认会对所有跳转请求放行，即等同于ctx.unsafeRedirect(url)

## MySQL数据库操作

**Egg-mysql**

框架提供了 [egg-mysql](https://github.com/eggjs/egg-mysql) 插件来访问 MySQL 数据库。

**安装与配置**

$ npm i --save egg-mysql

开启插件：*// config/plugin.js*

|  |
| --- |
| 配置数据库：*// config/config.default.js* |



**具体操作：**<https://eggjs.org/zh-cn/tutorials/mysql.html>

## Egg跨域

* 1. **egg-cors**

框架提供了 [egg-](https://github.com/eggjs/egg-mysql)cors 插件来实现cors跨域请求。

* 1. **egg-cors的安装和配置**

$ cnpm i --save egg-cors

开启插件：*// config/plugin.js*

*//跨域处理*

  cors:{

    enable: true,

    package: 'egg-cors',

  }

配置cors：*// config/config.default.js*

*// 跨域的配置*

    config.cors = {

        origin: '\*',

        allowMethods: 'GET,HEAD,PUT,POST,DELETE,PATCH'

    };

默认origin只支持一个域名或者\*表示全部，如果想支持具体的多个指定域名可以如下设置：

    config.cors = {

*// origin: ['http://localhost'],*

        origin:function(*ctx*) { *//设置允许来自指定域名请求*

            console.log(ctx);

            const whiteList = ['http://lulaoshi','http://127.0.0.1'];

            let url = ctx.request.header.origin;

            if(whiteList.includes(url)){

                return url;

            }

            return 'http://localhost' *//默认允许本地请求3000端口可跨域*

        },

        allowMethods: 'GET,HEAD,PUT,POST,DELETE,PATCH'

    };

有些时候需要携带凭证，这里还是要设置的：

credentials: true

需要注意的是，客户端也要配置：withCredentials: true

## 验证码

实现验证码使用svg-captcha,兼容性比较好，不需要插件支持。

$ cnpm i --save svg-captcha

创建一个service，在里面实现验证码图片：

'use strict';

const Service = require('egg').Service;

const svgCaptcha = require('svg-captcha');

class ToolsService extends Service {

*// 产生验证码*

    async captcha() {

        const captcha = svgCaptcha.create({

            size: 4,

            fontSize: 50,

            width: 100,

            height: 40,

            bacground: '#cc9966'

        });

        this.ctx.session.code = captcha.text;

        return captcha;

    }

}

module.exports = ToolsService;

在控制器里面调用这个service方法并输出为图片：

    async coder() {

        const {ctx} = this;

        let captcha = await this.service.tools.captcha(); *// 服务里面的方法*

        ctx.response.type = 'image/svg+xml';  *// 返回的类型*

        ctx.body = captcha.data; *// 返回一张图片*

    }

跨域的话，需要传递凭证：

xhr.withCredentials = true;

然后检查session里面存储的验证码和用户输入的验证码是否相等即可。