# **[Spring Boot：整合Shiro权限框架](https://www.cnblogs.com/xifengxiaoma/p/11061142.html)**

## 地址 https://www.cnblogs.com/xifengxiaoma/p/11061142.html

## **综合概述**

Shiro是Apache旗下的一个开源项目，它是一个非常易用的安全框架，提供了包括认证、授权、加密、会话管理等功能，与Spring Security一样属基于权限的安全框架，但是与Spring Security 相比，Shiro使用了比较简单易懂易于使用的授权方式。Shiro属于轻量级框架，相对于Spring Security简单很多，并没有security那么复杂。

## **优势特点**

它是一个功能强大、灵活的、优秀的、开源的安全框架。

它可以胜任身份验证、授权、企业会话管理和加密等工作。

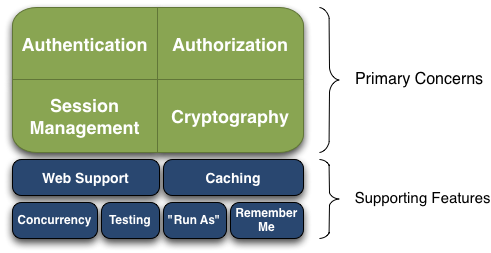
它易于使用和理解，与Spring Security相比，入门门槛低。

## **主要功能**

* 验证用户身份
* 用户访问权限控制
* 支持单点登录(SSO)功能
* 可以响应认证、访问控制，或Session事件
* 支持提供“Remember Me”服务
* .......

### **框架体系**

Shiro 的整体框架大致如下图所示（图片来自互联网）：



Authentication（认证）, Authorization（授权）, Session Management（会话管理）, Cryptography（加密）代表Shiro应用安全的四大基石。

它们分别是：

* Authentication（认证）：用户身份识别，通常被称为用户“登录”。
* Authorization（授权）：访问控制。比如某个用户是否具有某个操作的使用权限。
* Session Management（会话管理）：特定于用户的会话管理,甚至在非web 应用程序。
* Cryptography（加密）：在对数据源使用加密算法加密的同时，保证易于使用。

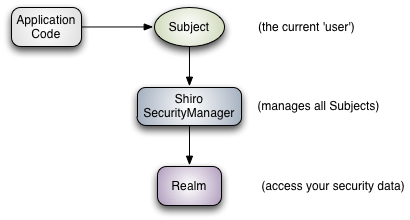
除此之外，还有其他的功能来支持和加强这些不同应用环境下安全领域的关注点。特别是对以下的功能支持：

* Web支持：Shiro 提供的 web 支持 api ，可以很轻松的保护 web 应用程序的安全。
* 缓存：缓存是 Apache Shiro 保证安全操作快速、高效的重要手段。
* 并发：Apache Shiro 支持多线程应用程序的并发特性。
* 测试：支持单元测试和集成测试，确保代码和预想的一样安全。
* “Run As”：这个功能允许用户在许可的前提下假设另一个用户的身份。
* “Remember Me”：跨 session 记录用户的身份，只有在强制需要时才需要登录。

### **主要流程**

在概念层，Shiro 架构包含三个主要的理念：Subject，SecurityManager 和 Realm。下面的图展示了这些组件如何相互作用，我们将在下面依次对其进行描述。

Shiro执行流程图（图片来自互联网）



三个主要理念：

* Subject：代表当前用户，Subject 可以是一个人，也可以是第三方服务、守护进程帐户、时钟守护任务或者其它当前和软件交互的任何事件。
* SecurityManager：管理所有Subject，SecurityManager 是 Shiro 架构的核心，配合内部安全组件共同组成安全伞。
* Realms：用于进行权限信息的验证，我们自己实现。Realm 本质上是一个特定的安全 DAO：它封装与数据源连接的细节，得到Shiro 所需的相关的数据。在配置 Shiro 的时候，你必须指定至少一个Realm 来实现认证（authentication）和/或授权（authorization）。

我们需要实现Realms的Authentication 和 Authorization。其中 Authentication 是用来验证用户身份，Authorization 是授权访问控制，用于对用户进行的操作授权，证明该用户是否允许进行当前操作，如访问某个链接，某个资源文件等。

## **实现案例**

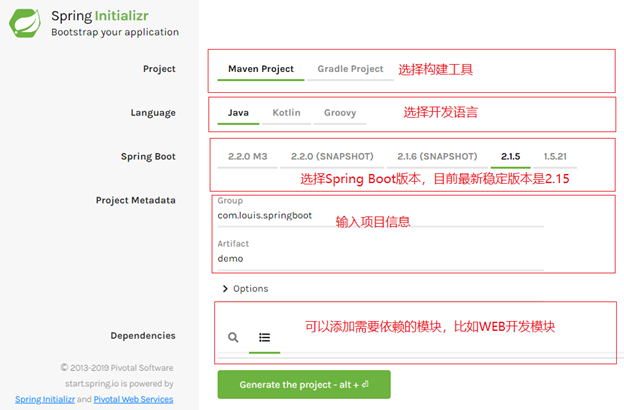
接下来，我们就通过一个具体的案例，来讲解如何进行Shiro的整合，然后借助Shiro实现登录认证和访问控制。

### **生成项目模板**

为方便我们初始化项目，Spring Boot给我们提供一个项目模板生成网站。

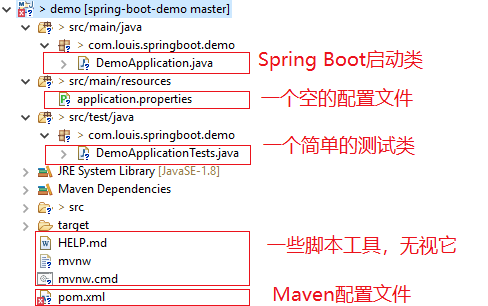
1.  打开浏览器，访问：<https://start.spring.io/>

2.  根据页面提示，选择构建工具，开发语言，项目信息等。



3.  点击 Generate the project，生成项目模板，生成之后会将压缩包下载到本地。

4.  使用IDE导入项目，我这里使用Eclipse，通过导入Maven项目的方式导入。



## **添加相关依赖**

清理掉不需要的测试类及测试依赖，添加 Maven 相关依赖，这里需要添加上WEB、Swagger、JPA和Shiro的依赖，Swagger的添加是为了方便接口测试。

pom.xml

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  <modelVersion>4.0.0</modelVersion>  <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>2.1.7.RELEASE</version>  <relativePath/> <!-- lookup parent from repository -->  </parent>  <groupId>com.louis.springboot</groupId>  <artifactId>demo</artifactId>  <version>0.0.1-SNAPSHOT</version>  <name>demo</name>  <description>Demo project for Spring Boot</description>  <properties>  <java.version>1.8</java.version>  </properties>  <dependencies>  <!-- web -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <!-- swagger -->  <dependency>  <groupId>io.springfox</groupId>  <artifactId>springfox-swagger2</artifactId>  <version>2.9.2</version>  </dependency>  <dependency>  <groupId>io.springfox</groupId>  <artifactId>springfox-swagger-ui</artifactId>  <version>2.9.2</version>  </dependency>  <!-- jpa -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-data-jpa</artifactId>  </dependency>  <!-- mysql -->  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  </dependency>  <!-- shiro -->  <dependency>  <groupId>org.apache.shiro</groupId>  <artifactId>shiro-spring</artifactId>  <version>1.4.1</version>  </dependency>  </dependencies>  <build>  <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  </plugin>  </plugins>  <!-- 打包时拷贝MyBatis的映射文件 -->  <resources>  <resource>  <directory>src/main/java</directory>  <includes>  <include>\*\*/sqlmap/\*.xml</include>  </includes>  <filtering>false</filtering>  </resource>  <resource>  <directory>src/main/resources</directory>  <includes>  <include>\*\*/\*.\*</include>  </includes>  <filtering>true</filtering>  </resource>  </resources>  </build>  </project> |

## **添加相关配置**

1.添加数据源和jpa相关配置

添加application.properties文件或者application.yml ，并在其中添加MySQL数据源连接信息。

注意：

这里需要首先创建一个MySQL数据库，并输入自己的用户名和密码。这里的数据库是springboot。

另外，如果你使用的是MySQL 5.x及以前版本，驱动配置driverClassName是com.mysql.jdbc.Driver。

application.yml

|  |
| --- |
| **server.port**: **8080  spring.datasource.url**= **jdbc:mysql://127.0.0.1:3306/springboot?useUnicode=true&characterEncoding=utf8&zeroDateTimeBehavior=convertToNull&autoReconnect=true** *# 或者是 spring.datasource.url=jdbc:mysql://localhost:3306/double\_ball?useUnicode=true&characterEncoding=utf8&zeroDateTimeBehavior=convertToNull* **spring.datasource.username**= **root spring.datasource.password**= **wjx123456 spring.datasource.driver-class-name**= **com.mysql.jdbc.Driver  spring.jpa.database**= *mysql # 默认false，在日志里显示执行的sql语句* **spring.jpa.show-sql**= **true spring.jpa.properties.hibernate.dialect**= **org.hibernate.dialect.MySQL5Dialect spring.jpa.database-platform**= **org.hibernate.dialect.MySQL5Dialect** *#指定为update，每次启动项目检测表结构有变化的时候会新增字段，表不存在时会新建，如果指定create，则每次启动项目都会清空数据并删除表，再新建*  **spring.jpa.hibernate.ddl-auto**= **update** *#指定jpa的自动表生成策略，驼峰自动映射为下划线格式* **spring.jpa.hibernate.naming.implicit-strategy**= **org.hibernate.boot.model.naming.ImplicitNamingStrategyLegacyJpaImpl spring.jpa.hibernate.naming.physical-strategy**= **org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl** |

2. 添加swagger 配置

添加一个swagger 配置类，在工程下新建 config 包并添加一个 SwaggerConfig 配置类。

|  |
| --- |
| **package** com.doubleball.config;  **import** org.springframework.context.annotation.Bean; **import** org.springframework.context.annotation.Configuration; **import** springfox.documentation.builders.ApiInfoBuilder; **import** springfox.documentation.builders.PathSelectors; **import** springfox.documentation.builders.RequestHandlerSelectors; **import** springfox.documentation.service.ApiInfo; **import** springfox.documentation.spi.DocumentationType; **import** springfox.documentation.spring.web.plugins.Docket; **import** springfox.documentation.swagger2.annotations.EnableSwagger2; @Configuration @EnableSwagger2 **public class** SwaggerConfig {   @Bean  **public** Docket createRestApi(){  **return new** Docket(DocumentationType.***SWAGGER\_2***).apiInfo(apiInfo())  .select()  .apis(RequestHandlerSelectors.*any*())  .paths(PathSelectors.*any*()).build();  }   **private** ApiInfo apiInfo(){  **return new** ApiInfoBuilder()  .title(**"SpringBoot API Doc"**)  .description(**"This is a restful api document of Spring Boot."**)  .version(**"1.0"**)  .build();  }  } |

### **编写业务代码**

添加一个用户类User，包含用户名和密码，用来进行登录认证，另外用户可以拥有角色。

|  |
| --- |
| package com.louis.springboot.demo.model;  import java.util.List;  import javax.persistence.CascadeType;  import javax.persistence.Column;  import javax.persistence.Entity;  import javax.persistence.GeneratedValue;  import javax.persistence.GenerationType;  import javax.persistence.Id;  import javax.persistence.OneToMany;  @Entity  public class User {  @Id *// 主键声明*  @GeneratedValue(strategy = GenerationType.AUTO) *// 主键生成策略，由JPA自动生成*  private Long id;  @Column(unique = true) *// unique 属性表示该字段是否为唯一标识*  private String name;  private String password;  @OneToMany(cascade = CascadeType.ALL,mappedBy = "user") *// 关联实体的类型，级联类型为所有，关联实体的属性为“user”*  private List<Role> roles;  public Long getId() {  return id;  }  public void setId(Long id) {  this.id = id;  }  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  public List<Role> getRoles() {  return roles;  }  public void setRoles(List<Role> roles) {  this.roles = roles;  }  public String getPassword() {  return password;  }  public void setPassword(String password) {  this.password = password;  }  } |

添加一个角色类Role，表示用户角色，角色拥有可操作的权限集合。

Role.java

|  |
| --- |
| package com.louis.springboot.demo.model;  import java.util.List;  import javax.persistence.CascadeType;  import javax.persistence.Entity;  import javax.persistence.FetchType;  import javax.persistence.GeneratedValue;  import javax.persistence.GenerationType;  import javax.persistence.Id;  import javax.persistence.ManyToOne;  import javax.persistence.OneToMany;  @Entity  public class Role {  @Id  @GeneratedValue(strategy = GenerationType.AUTO)  private Long id;  private String roleName;  @ManyToOne(fetch = FetchType.EAGER)  private User user;  @OneToMany(cascade = CascadeType.ALL,mappedBy = "role")  private List<Permission> permissions;  public Long getId() {  return id;  }  public void setId(Long id) {  this.id = id;  }  public String getRoleName() {  return roleName;  }  public void setRoleName(String roleName) {  this.roleName = roleName;  }  public User getUser() {  return user;  }  public void setUser(User user) {  this.user = user;  }  public List<Permission> getPermissions() {  return permissions;  }  public void setPermissions(List<Permission> permissions) {  this.permissions = permissions;  }  } |

添加一个权限类Permission，表示资源访问权限。

Permission.java

|  |
| --- |
| package com.louis.springboot.demo.model;  import javax.persistence.Entity;  import javax.persistence.FetchType;  import javax.persistence.GeneratedValue;  import javax.persistence.GenerationType;  import javax.persistence.Id;  import javax.persistence.ManyToOne;  @Entity  public class Permission {  @Id  @GeneratedValue(strategy = GenerationType.AUTO)  private Long id;  private String permission;  @ManyToOne(fetch = FetchType.EAGER)  private Role role;  public Long getId() {  return id;  }  public void setId(Long id) {  this.id = id;  }  public String getPermission() {  return permission;  }  public void setPermission(String permission) {  this.permission = permission;  }  public Role getRole() {  return role;  }  public void setRole(Role role) {  this.role = role;  }  } |

添加一个DAO基础接口，用来被其他DAO继承。

BaseDao.java

|  |
| --- |
| package com.louis.springboot.demo.dao;  import java.io.Serializable;  import org.springframework.data.jpa.repository.JpaSpecificationExecutor;  import org.springframework.data.repository.NoRepositoryBean;  import org.springframework.data.repository.PagingAndSortingRepository;  @NoRepositoryBean  public interface BaseDao<T, I extends Serializable>  extends PagingAndSortingRepository<T, I>, JpaSpecificationExecutor<T> {  } |

添加一个UserDao，用来操作用户信息。

UserDao.java

|  |
| --- |
| package com.louis.springboot.demo.dao;  import com.louis.springboot.demo.model.User;  public interface UserDao extends BaseDao<User, Long> {  User findByName(String name);  } |

添加一个RoleDao，用来操作角色信息。

RoleDao.java

|  |
| --- |
| package com.louis.springboot.demo.dao;  import com.louis.springboot.demo.model.Role;  public interface RoleDao extends BaseDao<Role, Long> {  } |

添加一个LoginService服务接口。

LoginService.java

|  |
| --- |
| package com.louis.springboot.demo.service;  import com.louis.springboot.demo.model.Role;  import com.louis.springboot.demo.model.User;  public interface LoginService {  User addUser(User user);  Role addRole(Role role);  User findByName(String name);  } |

添加一个LoginServiceImpl，实现服务功能，这里为了方便，在插入角色的时候会默认设置其权限。

LoginServiceImpl.java

|  |
| --- |
| package com.louis.springboot.demo.service.impl;  import java.util.ArrayList;  import java.util.List;  import javax.transaction.Transactional;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.stereotype.Service;  import com.louis.springboot.demo.dao.RoleDao;  import com.louis.springboot.demo.dao.UserDao;  import com.louis.springboot.demo.model.Permission;  import com.louis.springboot.demo.model.Role;  import com.louis.springboot.demo.model.User;  import com.louis.springboot.demo.service.LoginService;  @Service  @Transactional  public class LoginServiceImpl implements LoginService {  @Autowired  private UserDao userDao;  @Autowired  private RoleDao roleDao;  //添加用户 @Override  public User addUser(User user) {  userDao.save(user); *// save方法是org.springframework.data.repository.CrudRepository中的方法*  return user;  }  //添加角色 @Override  public Role addRole(Role role) {  User user = userDao.findByName(role.getUser().getName());  role.setUser(user);  Permission permission1 = new Permission();  permission1.setPermission("create");  permission1.setRole(role);  Permission permission2 = new Permission();  permission2.setPermission("update");  permission2.setRole(role);  List<Permission> permissions = new ArrayList<Permission>();  permissions.add(permission1);  permissions.add(permission2);  role.setPermissions(permissions);  roleDao.save(role);  return role;  }  //查询用户通过用户名 @Override  public User findByName(String name) {  return userDao.findByName(name);  }  } |

添加一个登录控制器，编写相关的接口。create接口添加了@RequiresPermissions("create")，用于进行权限注解测试。

LoginController.java

|  |
| --- |
| package com.louis.springboot.demo.controller;  import org.apache.shiro.SecurityUtils;  import org.apache.shiro.authc.UsernamePasswordToken;  import org.apache.shiro.authz.annotation.RequiresPermissions;  import org.apache.shiro.authz.annotation.RequiresRoles;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.web.bind.annotation.GetMapping;  import org.springframework.web.bind.annotation.PostMapping;  import org.springframework.web.bind.annotation.RequestBody;  import org.springframework.web.bind.annotation.RestController;  import com.louis.springboot.demo.model.Role;  import com.louis.springboot.demo.model.User;  import com.louis.springboot.demo.service.LoginService;  @RestControllerpublic class LoginController {  @Autowired  private LoginService loginService;  /\*\*  \* POST登录  \* @param map  \* @return  \*/  @PostMapping(value = "/login")  public String login(@RequestBody User user) {  // 添加用户认证信息  UsernamePasswordToken usernamePasswordToken = new UsernamePasswordToken(user.getName(), user.getPassword());  // 进行验证，这里可以捕获异常，然后返回对应信息 SecurityUtils.getSubject().login(usernamePasswordToken);  return "login ok!";  }  /\*\*  \* 添加用户  \* @param user  \* @return  \*/  @PostMapping(value = "/addUser")  public String addUser(@RequestBody User user) {  user = loginService.addUser(user);  return "addUser is ok! \n" + user;  }  /\*\*  \* 添加角色  \* @param role  \* @return  \*/  @PostMapping(value = "/addRole")  public String addRole(@RequestBody Role role) {  role = loginService.addRole(role);  return "addRole is ok! \n" + role;  }  /\*\*  \* 注解的使用  \* @return  \*/  @RequiresRoles("admin")  @RequiresPermissions("create")  @GetMapping(value = "/create")  public String create() {  return "Create success!";  }  @GetMapping(value = "/index")  public String index() {  return "index page!";  }  @GetMapping(value = "/error")  public String error() {  return "error page!";  }  /\*\*  \* 退出的时候是get请求，主要是用于退出  \* @return  \*/  @GetMapping(value = "/login")  public String login() {  return "login";  }  @GetMapping(value = "/logout")  public String logout() {  return "logout";  }  } |

添加一个MyShiroRealm并继承AuthorizingRealm，实现其中的两个方法。

doGetAuthenticationInfo：实现用户认证，通过服务加载用户信息并构造认证对象返回。

doGetAuthorizationInfo：实现权限认证，通过服务加载用户角色和权限信息设置进去。

MyShiroRealm.java

|  |
| --- |
| package com.louis.springboot.demo.config;  import org.apache.shiro.authc.AuthenticationException;  import org.apache.shiro.authc.AuthenticationInfo;  import org.apache.shiro.authc.AuthenticationToken;  import org.apache.shiro.authc.SimpleAuthenticationInfo;  import org.apache.shiro.authz.AuthorizationInfo;  import org.apache.shiro.authz.SimpleAuthorizationInfo;  import org.apache.shiro.realm.AuthorizingRealm;  import org.apache.shiro.subject.PrincipalCollection;  import org.springframework.beans.factory.annotation.Autowired;  import com.louis.springboot.demo.model.Permission;  import com.louis.springboot.demo.model.Role;  import com.louis.springboot.demo.model.User;  import com.louis.springboot.demo.service.LoginService;  /\*\*  \* 实现AuthorizingRealm接口用户用户认证  \* @author Louis  \* @date Jun 20, 2019  \*/public class MyShiroRealm extends AuthorizingRealm {  @Autowired  private LoginService loginService;  /\*\*  \* 用户认证  \*/  @Override  protected AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken authenticationToken)  throws AuthenticationException {  // 加这一步的目的是在Post请求的时候会先进认证，然后在到请求  if (authenticationToken.getPrincipal() == null) {  return null;  }  // 获取用户信息  String name = authenticationToken.getPrincipal().toString();  User user = loginService.findByName(name);  if (user == null) {  // 这里返回后会报出对应异常  return null;  } else {  // 这里验证authenticationToken和simpleAuthenticationInfo的信息  SimpleAuthenticationInfo simpleAuthenticationInfo = new SimpleAuthenticationInfo(name,  user.getPassword().toString(), getName());  return simpleAuthenticationInfo;  }  }  /\*\*  \* 角色权限和对应权限添加  \*/  @Override  protected AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principalCollection) {  // 获取登录用户名  String name = (String) principalCollection.getPrimaryPrincipal();  // 查询用户名称  User user = loginService.findByName(name);  // 添加角色和权限  SimpleAuthorizationInfo simpleAuthorizationInfo = new SimpleAuthorizationInfo();  for (Role role : user.getRoles()) {  // 添加角色 simpleAuthorizationInfo.addRole(role.getRoleName());  for (Permission permission : role.getPermissions()) {  // 添加权限 simpleAuthorizationInfo.addStringPermission(permission.getPermission());  }  }  return simpleAuthorizationInfo;  }  } |

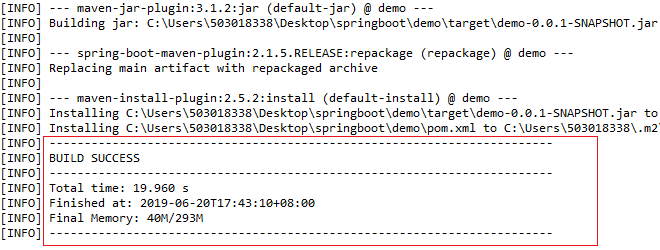
添加一个Shiro配置类，主要配置路由的访问控制，以及注入自定义的认证器MyShiroRealm。

ShiroConfig.java

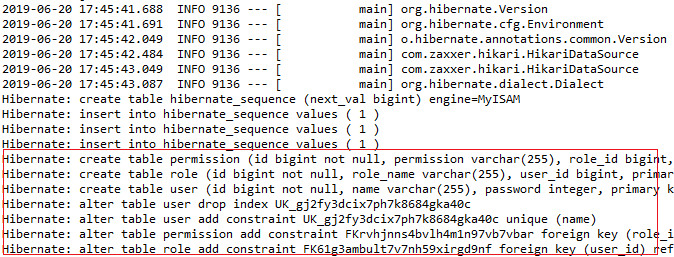
|  |
| --- |
| package com.louis.springboot.demo.config;  import java.util.HashMap;  import java.util.Map;  import org.apache.shiro.mgt.SecurityManager;  import org.apache.shiro.spring.security.interceptor.AuthorizationAttributeSourceAdvisor;  import org.apache.shiro.spring.web.ShiroFilterFactoryBean;  import org.apache.shiro.web.mgt.DefaultWebSecurityManager;  import org.springframework.context.annotation.Bean;  import org.springframework.context.annotation.Configuration;  @Configuration  public class ShiroConfig {  // 将自己的验证方式加入容器 @Bean  public MyShiroRealm myShiroRealm() {  MyShiroRealm myShiroRealm = new MyShiroRealm();  return myShiroRealm;  }  // 权限管理，配置主要是Realm的管理认证 @Bean  public SecurityManager securityManager() {  DefaultWebSecurityManager securityManager = new DefaultWebSecurityManager();  securityManager.setRealm(myShiroRealm());  return securityManager;  }  // Filter工厂，设置对应的过滤条件和跳转条件 @Bean  public ShiroFilterFactoryBean shiroFilterFactoryBean(SecurityManager securityManager) {  ShiroFilterFactoryBean shiroFilterFactoryBean = new ShiroFilterFactoryBean();  shiroFilterFactoryBean.setSecurityManager(securityManager);  Map<String, String> filterMap = new HashMap<String, String>();  // 登出  filterMap.put("/logout", "logout");  // swagger  filterMap.put("/swagger\*\*/\*\*", "anon");  filterMap.put("/webjars/\*\*", "anon");  filterMap.put("/v2/\*\*", "anon");  // 对所有用户认证  filterMap.put("/\*\*", "authc");  // 登录  shiroFilterFactoryBean.setLoginUrl("/login");  // 首页  shiroFilterFactoryBean.setSuccessUrl("/index");  // 错误页面，认证不通过跳转  shiroFilterFactoryBean.setUnauthorizedUrl("/error");  shiroFilterFactoryBean.setFilterChainDefinitionMap(filterMap);  return shiroFilterFactoryBean;  }  // 加入注解的使用，不加入这个注解不生效 @Bean  public AuthorizationAttributeSourceAdvisor authorizationAttributeSourceAdvisor(SecurityManager securityManager) {  AuthorizationAttributeSourceAdvisor authorizationAttributeSourceAdvisor = new AuthorizationAttributeSourceAdvisor();  authorizationAttributeSourceAdvisor.setSecurityManager(securityManager);  return authorizationAttributeSourceAdvisor;  }  } |

### **编译测试运行**

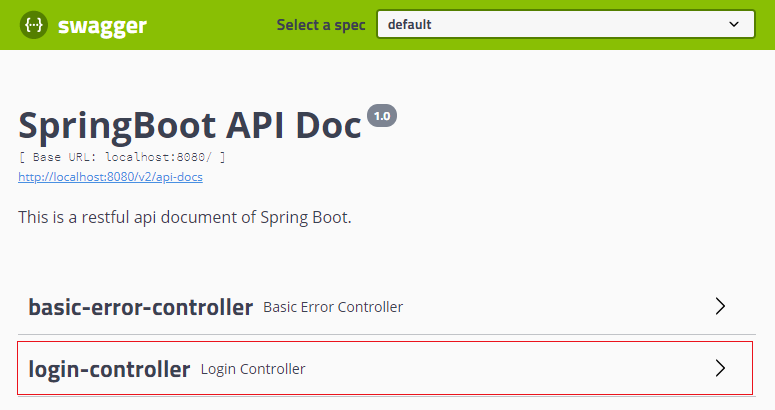
1.  右键项目 -> Run as -> Maven install，开始执行Maven构建，第一次会下载Maven依赖，可能需要点时间，如果出现如下信息，就说明项目编译打包成功了。



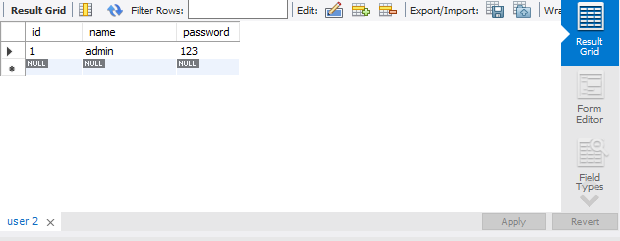
2.  右键文件 DemoApplication.java -> Run as -> Java Application，开始启动应用，如果一开始数据库没有对应的表，在应用启动时会创建，我们可以通过控制台查看到对应的SQL语句。



3.  打开浏览器，访问：<http://localhost:8080/swagger-ui.html>，进入swagger接口文档界面。



4.  首先用MySQL客户端，打开数据库，往用户表里面插入一条记录，{id=1, name="admin", password="123"}。



然后试着用Swagger调用addUser往用户表插入一条记录。

{

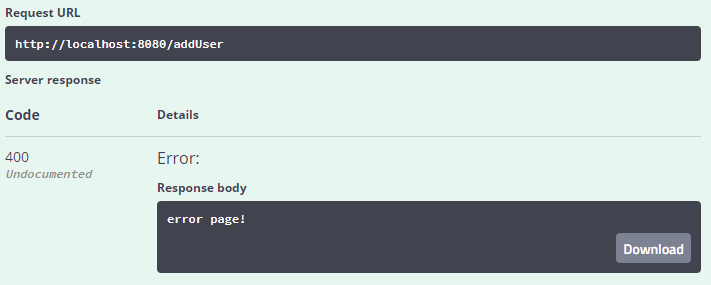
"id": 2,

"name": "xiaoming",

"password": "123"

}

结果返回"error page!"，这是因为我们没有登录，还没有操作权限。



接着调用POST的login接口，输入以下用户信息进行登录。

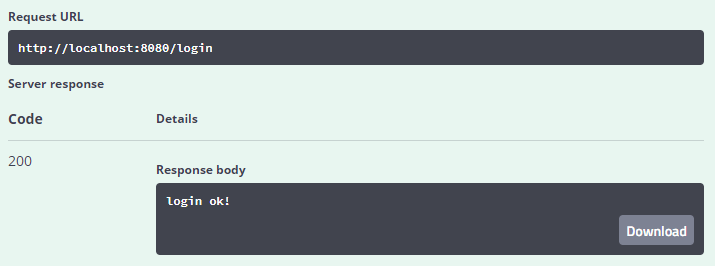
{

"name": "admin",

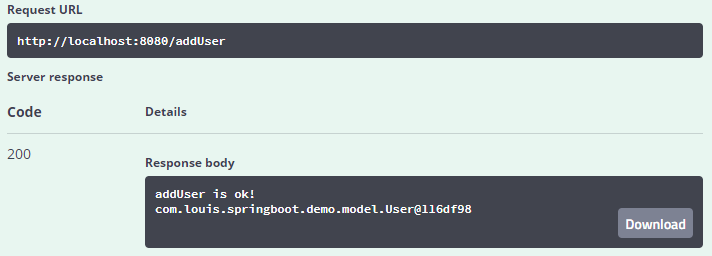
"password": "123"

}

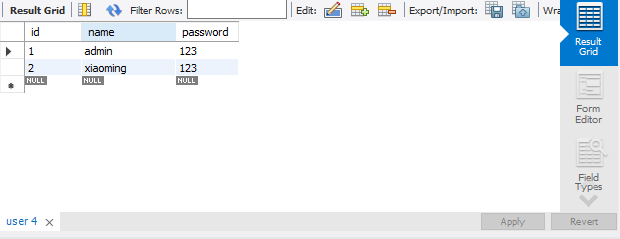
登录成功之后，返回“login ok!”信息。



再次调用addUser往用户表插入记录，发现记录已经可以成功插入了。

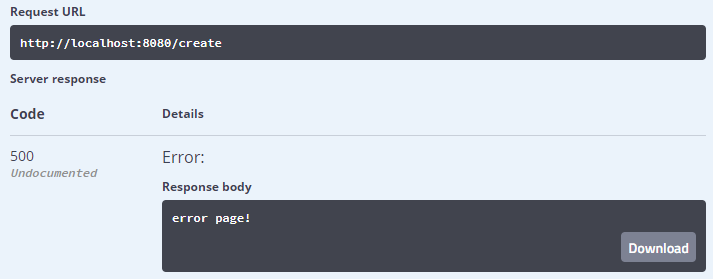


通过客户端工具我们也可以查看到记录已经插入进来了。



通过上面的测试，我们已经成功的验证了，受保护的接口需要在登录之后才允许访问。接下来我们来测试一下权限注解的效果，我们在create方法上加上了权限注解@RequiresPermissions("create")，表示用户需要拥有"create"的权限才能访问。

先尝试调用以下create接口，发现尽管我们已经登录了，依然因为没有权限返回了“error page!”。



然后我们调用addRole插入以下角色记录，这个角色关联了我们当前登录admin用户，且角色在创建时我们代码默认设置拥有了“create”权限。

IMG_297

{

"id": 1,

"roleName": "admin",

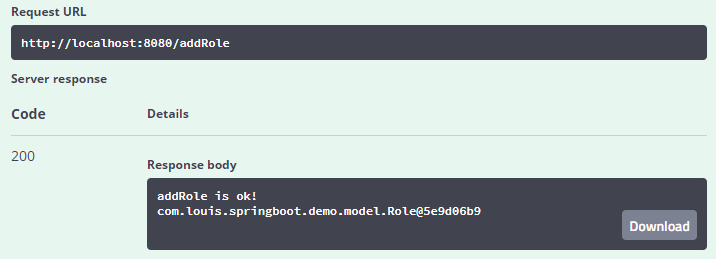
"user": {

"name": "admin"

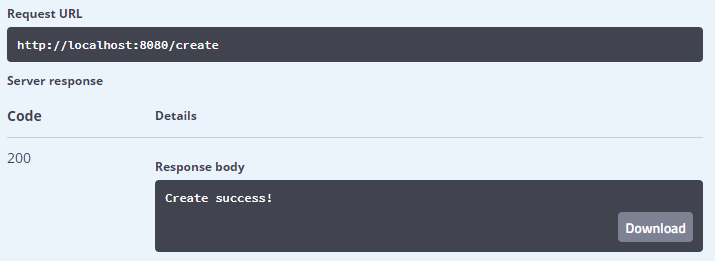
}

}

如果执行正确的话，会返回如下信息，说明角色已经成功插入了。



然后我们再一次调用create接口，因为此刻admin用户拥有admin角色，而admin角色拥有“create”权限，所以已经具有接口访问权限了。



# **[Swagger](https://www.cnblogs.com/xifengxiaoma/p/11061142.html)**

## 地址 https://www.cnblogs.com/xifengxiaoma/p/11061142.html

## **添加相关依赖**

在pom.xml添加Swagger依赖。

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  <modelVersion>4.0.0</modelVersion>  <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>2.1.7.RELEASE</version>  <relativePath/> <!-- lookup parent from repository -->  </parent>  <groupId>com.louis.springboot</groupId>  <artifactId>demo</artifactId>  <version>0.0.1-SNAPSHOT</version>  <name>demo</name>  <description>Demo project for Spring Boot</description>  <properties>  <java.version>1.8</java.version>  </properties>  <dependencies>  <!-- web -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <!-- swagger -->  <dependency>  <groupId>io.springfox</groupId>  <artifactId>springfox-swagger2</artifactId>  <version>2.9.2</version>  </dependency>  <dependency>  <groupId>io.springfox</groupId>  <artifactId>springfox-swagger-ui</artifactId>  <version>2.9.2</version>  </dependency>  </dependencies>  </project> |

## **添加相关配置**

添加一个swagger 配置类，在工程下新建 config 包并添加一个 SwaggerConfig 配置类。

|  |
| --- |
| **package** com.doubleball.config;  **import** org.springframework.context.annotation.Bean; **import** org.springframework.context.annotation.Configuration; **import** springfox.documentation.builders.ApiInfoBuilder; **import** springfox.documentation.builders.PathSelectors; **import** springfox.documentation.builders.RequestHandlerSelectors; **import** springfox.documentation.service.ApiInfo; **import** springfox.documentation.spi.DocumentationType; **import** springfox.documentation.spring.web.plugins.Docket; **import** springfox.documentation.swagger2.annotations.EnableSwagger2; @Configuration @EnableSwagger2 **public class** SwaggerConfig {   @Bean  **public** Docket createRestApi(){  **return new** Docket(DocumentationType.***SWAGGER\_2***).apiInfo(apiInfo())  .select()  .apis(RequestHandlerSelectors.*any*())  .paths(PathSelectors.*any*()).build();  }   **private** ApiInfo apiInfo(){  **return new** ApiInfoBuilder()  .title(**"SpringBoot API Doc"**)  .description(**"This is a restful api document of Spring Boot."**)  .version(**"1.0"**)  .build();  }  } |

## **编写业务代码**

添加一个控制器，编写相关的接口。

OperateBallController.java

|  |
| --- |
| **package** com.doubleball.controller;  **import** org.springframework.stereotype.Controller; **import** org.springframework.web.bind.annotation.GetMapping; **import** org.springframework.web.bind.annotation.RequestMapping; **import** org.springframework.web.bind.annotation.RequestMethod; @Controller @RequestMapping(**"/api/doubleball"**) **public class** OperateBallController {   @GetMapping(**"/initball"**)  **public void** initBall() {  System.***out***.println(**"initBall"**);  }   @RequestMapping(value = **"/openbrowser"**,method = RequestMethod.***GET***)  **public void** openBrowserWithFirxfoxAndChrome(){  System.***out***.println(**"openBrowserWithFirefoxAndChrome"**);  }   @RequestMapping(**"/index"**)  **public** String index(){  **return "template/index"**;  }  } |

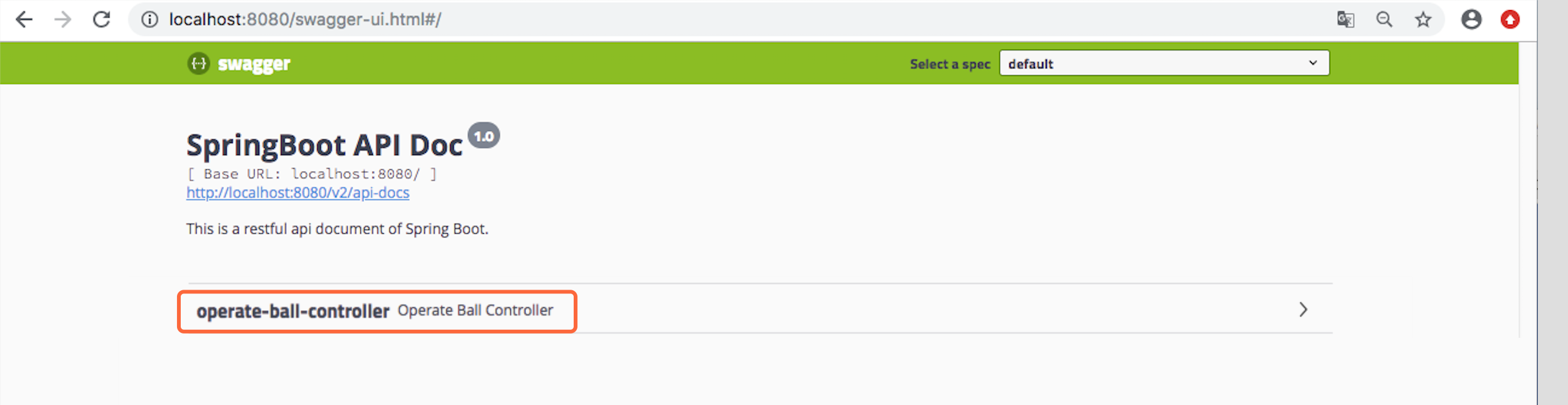
## **编写入口类**

添加一个入口类

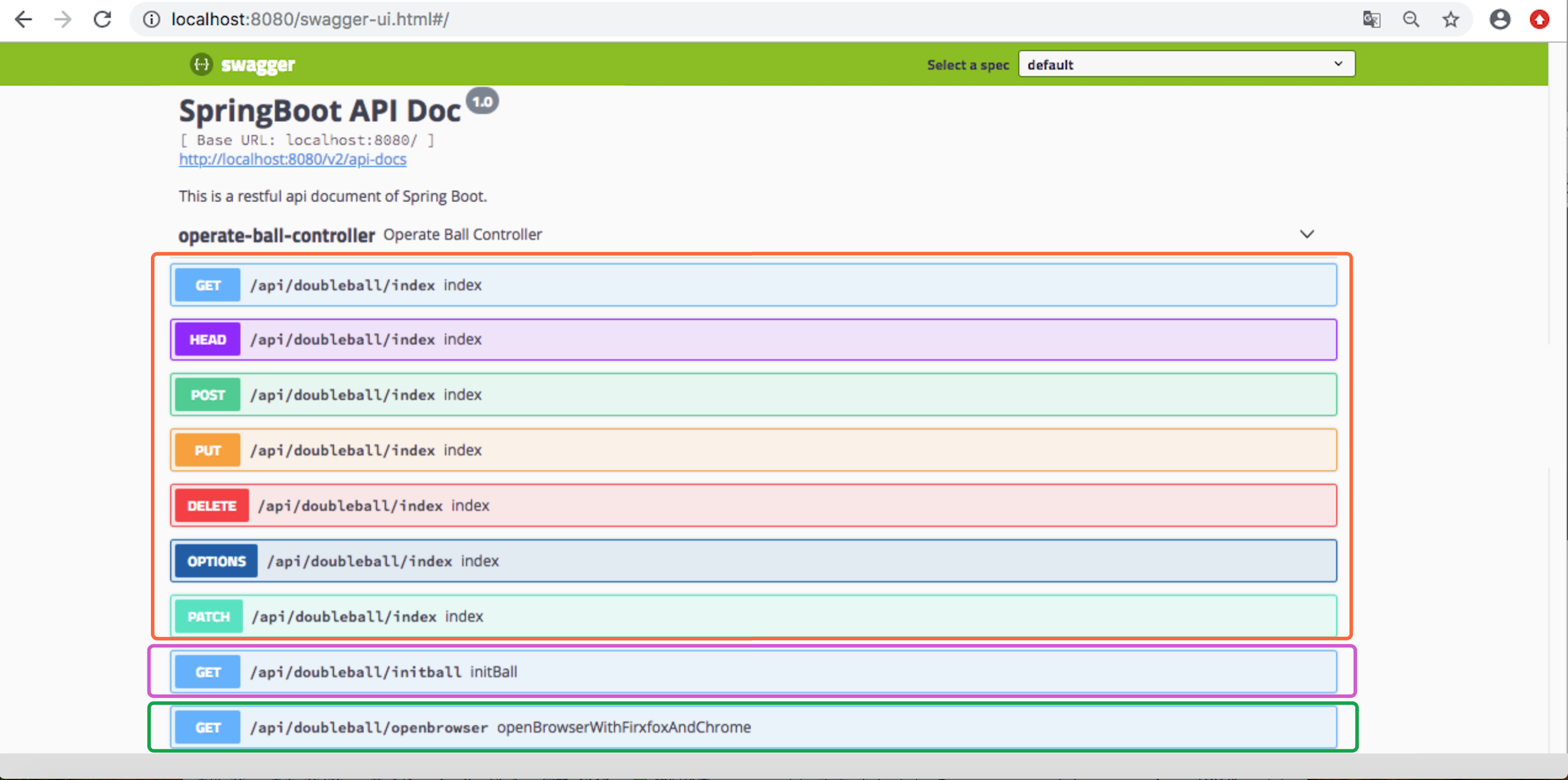
|  |
| --- |
| **package** com.doubleball;  **import** org.springframework.boot.SpringApplication; **import** org.springframework.boot.autoconfigure.SpringBootApplication; @SpringBootApplication **public class** APP {   **public static void** main(String[] args) {  SpringApplication.*run*(APP.**class**,args);  }  } |

## **查看swagger信息**

打开浏览器，访问：<http://localhost:8080/swagger-ui.html>，进入swagger接口文档界面。



点击控制类，查看里面的方法



上图中之所以会出现不同的分类，是因为控制类中方法的不同注释导致。

1、方法 “/api/doubleball/index”的注解是 @RequestMapping("/index") ，没有指明是哪种请求，所以swagger把所有请求都加在了方法上。

2、方法 “/api/doubleball/initball”的注解是 @GetMapping("/index") ，指明了是get请求，所以swagger把get请求都加在了方法上。

3、方法 “/api/doubleball/openbrowser”的注解是 @RequestMapping(value = "/openbrowser",method = RequestMethod.GET) ，指明了是get请求，所以swagger把get请求都加在了方法上。