SpringSecurity

目录

[一、简介 2](#_Toc104276817)

[二、认证流程介绍 2](#_Toc104276818)

[1.登陆校验流程 2](#_Toc104276819)

[2.认证流程详解 3](#_Toc104276820)

[三、登录模块的代码实现 3](#_Toc104276821)

[1.依赖 3](#_Toc104276822)

[2.数据库 5](#_Toc104276823)

[3.实体类 5](#_Toc104276824)

[4.Mapper接口 6](#_Toc104276825)

[5.工具类 6](#_Toc104276826)

[6.核心类 9](#_Toc104276827)

[（1）UserDetailsServiceImpl 9](#_Toc104276828)

[（2）UserService 10](#_Toc104276829)

[（3）LoginController 10](#_Toc104276830)

[（4）LoginService 11](#_Toc104276831)

[（5）SecurityConfig 12](#_Toc104276832)

[7.测试 13](#_Toc104276833)

[四、认证模块的代码实现 14](#_Toc104276834)

[五、退出模块的代码实现 18](#_Toc104276835)

[六、权限管理 20](#_Toc104276836)

[七、自定义失败处理 23](#_Toc104276837)

推荐学习：

<https://blog.csdn.net/HXBest/article/details/106433454>

<https://zhuanlan.zhihu.com/p/144536764>

# 一、简介

​ Spring Security 是 Spring 家族中的一个安全管理框架。相比与另外一个安全框架Shiro，它提供了更丰富的功能，社区资源也比Shiro丰富。

​ 一般来说中大型的项目都是使用SpringSecurity 来做安全框架。小项目有Shiro的比较多，因为相比与SpringSecurity，Shiro的上手更加的简单。

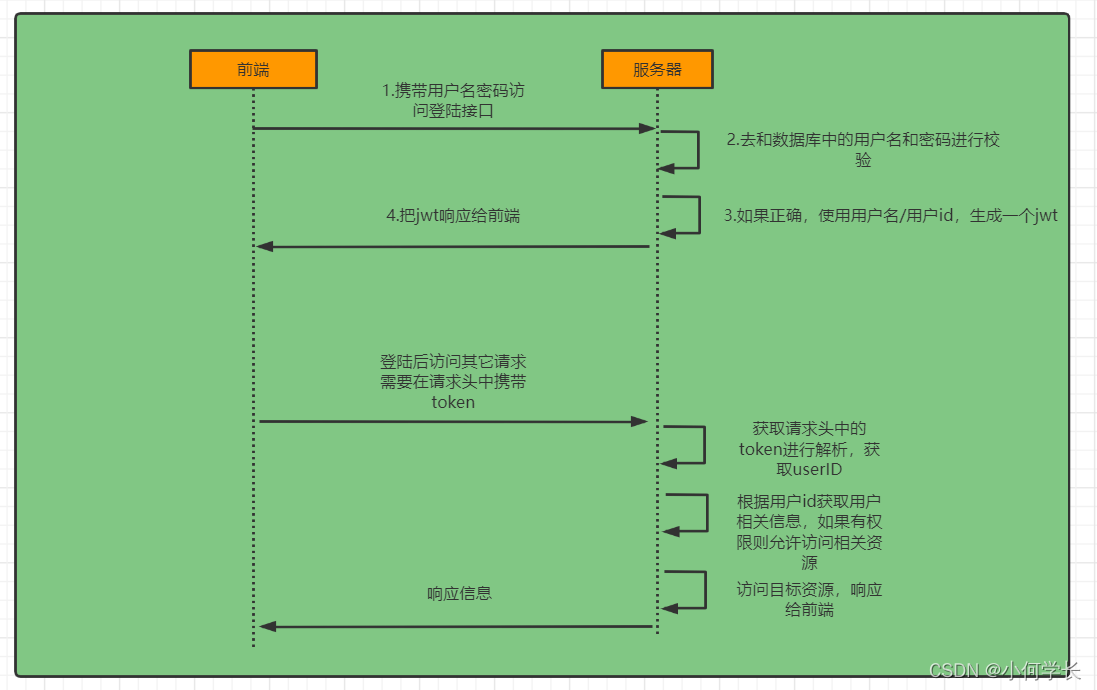
​ 一般Web应用的需要进行认证和授权。

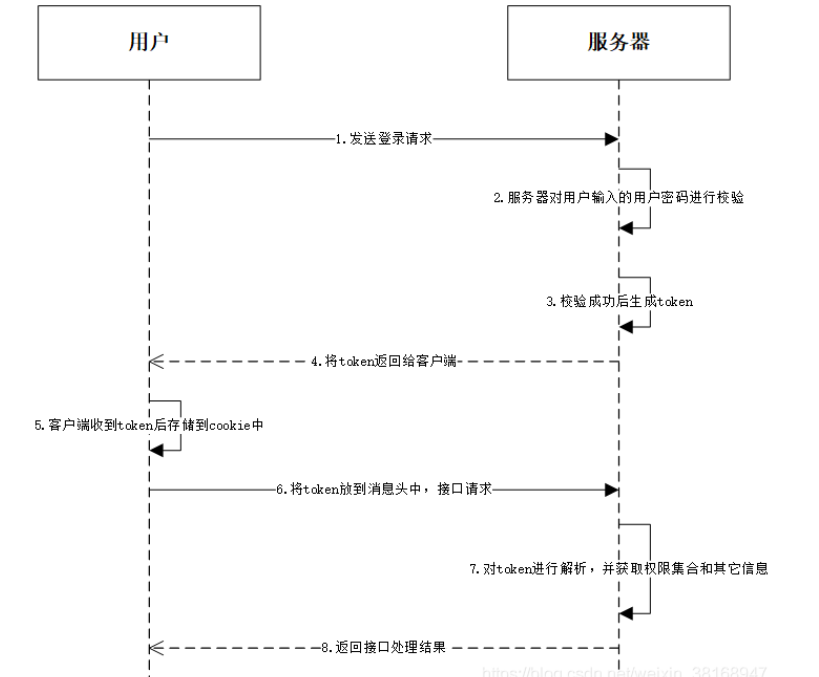
​ **认证：**验证当前访问系统的是不是本系统的用户，并且要确认具体是哪个用户

​ **授权：**经过认证后判断当前用户是否有权限进行某个操作。

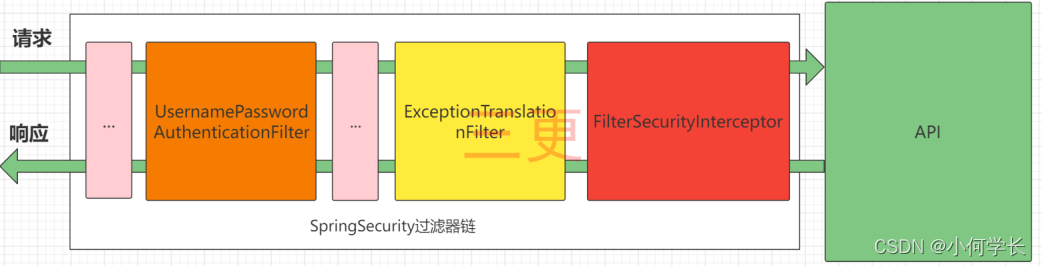
# 二、认证流程介绍

## 1.登陆校验流程





SpringSecurity的原理其实就是一个过滤器链，内部包含了提供各种功能的过滤器。这里我们可以看看入门案例中的过滤器。

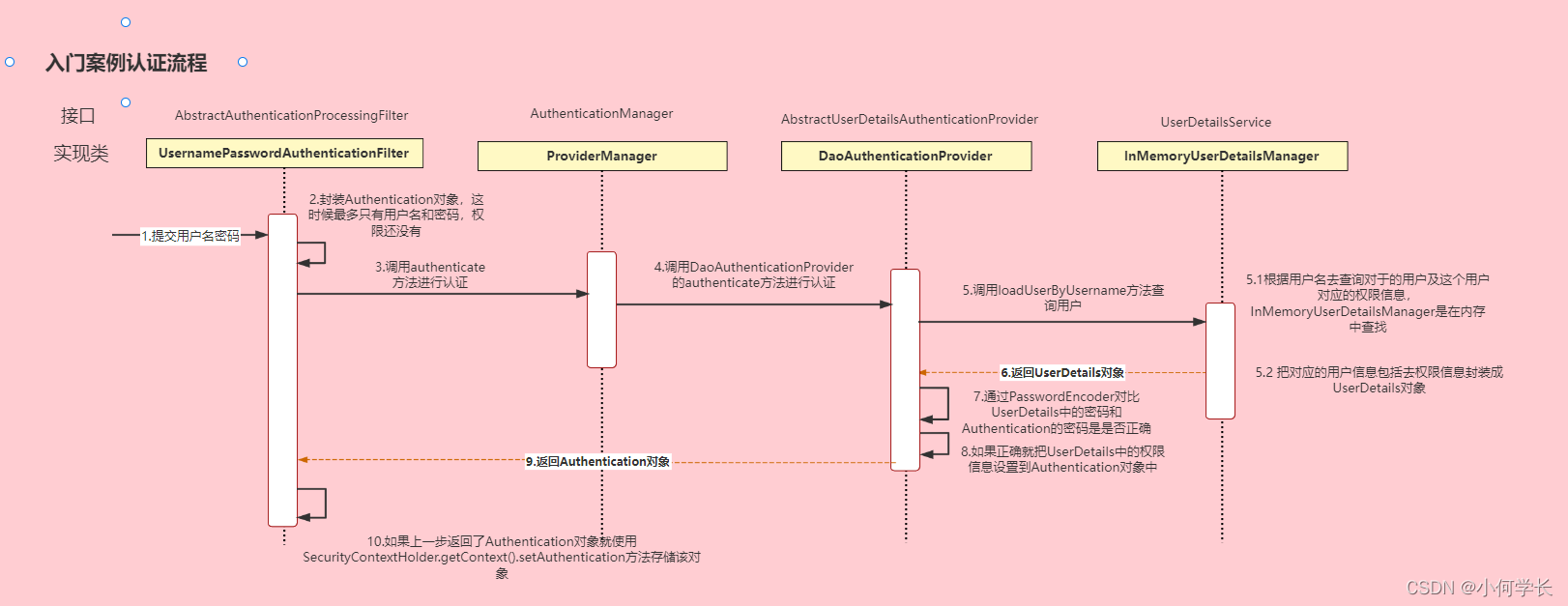
  
​ 图中只展示了核心过滤器，其它的非核心过滤器并没有在图中展示。

UsernamePasswordAuthenticationFilter:负责处理我们在登陆页面填写了用户名密码后的登陆请求。入门案例的认证工作主要有它负责。

ExceptionTranslationFilter： 处理过滤器链中抛出的任何AccessDeniedException和AuthenticationException 。

FilterSecurityInterceptor： 负责权限校验的过滤器。

## 2.认证流程详解



概念速查:

Authentication接口: 它的实现类，表示当前访问系统的用户，封装了用户相关信息。

AuthenticationManager接口：定义了认证Authentication的方法

UserDetailsService接口：加载用户特定数据的核心接口。里面定义了一个根据用户名查询用户信息的方法。

UserDetails接口：提供核心用户信息。通过UserDetailsService根据用户名获取处理的用户信息要封装成UserDetails对象返回。然后将这些信息封装到Authentication对象中。

# 三、登录模块的代码实现

## 1.依赖

<dependencies>  
 <dependency>  
 <groupId>com.alibaba</groupId>  
 <artifactId>fastjson</artifactId>  
 <version>1.2.62</version>  
 </dependency>  
 <dependency>  
 <groupId>io.jsonwebtoken</groupId>  
 <artifactId>jjwt</artifactId>  
 <version>0.9.1</version>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-data-redis</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>com.baomidou</groupId>  
 <artifactId>mybatis-plus-boot-starter</artifactId>  
 <version>3.5.1</version>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-security</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-web</artifactId>  
 </dependency>  
  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-devtools</artifactId>  
 <scope>runtime</scope>  
 <optional>true</optional>  
 </dependency>  
 <dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
 <scope>runtime</scope>  
 </dependency>  
 <dependency>  
 <groupId>org.projectlombok</groupId>  
 <artifactId>lombok</artifactId>  
 <optional>true</optional>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-test</artifactId>  
 <scope>test</scope>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.security</groupId>  
 <artifactId>spring-security-test</artifactId>  
 <scope>test</scope>  
 </dependency>  
</dependencies>

唯一需要注意的是，如果你使用的 Java 版本是 11，那么你还需要添加以下依赖，使用 Java8 则不需要。

<dependency>

<groupId>javax.xml.bind</groupId>

<artifactId>jaxb-api</artifactId>

<version>2.3.0</version>

</dependency>

## 2.数据库

CREATE TABLE `t\_user` (

`name` varchar(20) NOT NULL,

`pwd` varchar(200) DEFAULT NULL,

`state` tinyint(1) NOT NULL DEFAULT '0',

`authority` varchar(20) DEFAULT NULL,

PRIMARY KEY (`name`)

) ENGINE=InnoDB AUTO\_INCREMENT=4 DEFAULT CHARSET=utf8;

insert into t\_user values('Tom','$2a$10$5ooQI8dir8jv0/gCa1Six.GpzAdIPf6pMqdminZ/3ijYzivCyPlfK',0,'ROLE\_common');

insert into t\_user values('Mike','$2a$10$5ooQI8dir8jv0/gCa1Six.GpzAdIPf6pMqdminZ/3ijYzivCyPlfK',0,'ROLE\_vip');

insert into t\_user values('Mary','$2a$10$5ooQI8dir8jv0/gCa1Six.GpzAdIPf6pMqdminZ/3ijYzivCyPlfK',1,'ROLE\_common');

insert into t\_user values('Alice','$2a$10$5ooQI8dir8jv0/gCa1Six.GpzAdIPf6pMqdminZ/3ijYzivCyPlfK',1,'ROLE\_vip');

上面的密码是“123456”通过BCryptPasswordEncoder类加密后的密码。

## 3.全局配置文件

spring.datasource.url=jdbc:mysql://localhost:3306/bookdb?severTimezone=UTC  
spring.datasource.username=root  
spring.datasource.password=123456  
  
mybatis.configuration.map-underscore-to-camel-case=true  
  
server.port=8081

## 4.实体类

与数据表t\_user对应的实体类，因为在UserDetailsServiceImpl类中要用到org.springframework.security.core.userdetails.User，所以这里最好不要用User作为类名。

import com.baomidou.mybatisplus.annotation.TableId;  
import com.baomidou.mybatisplus.annotation.TableName;  
import lombok.AllArgsConstructor;  
import lombok.Data;  
import lombok.NoArgsConstructor;  
import java.io.Serializable;  
  
//因为在UserDetailsServiceImpl类中要用到org.springframework.security.core.userdetails.User，所以这里最好不要用User作为类名，以免混淆  
@Data  
@AllArgsConstructor  
@NoArgsConstructor  
@TableName("t\_user")  
public class SystemUser implements Serializable {  
 @TableId  
 private String name;  
 private String pwd;  
 private int state;  
 private String authority;  
}

## 5.Mapper接口

import cn.gdsdxy.springboot25security\_vue.pojo.SystemUser;  
import com.baomidou.mybatisplus.core.mapper.BaseMapper;  
import org.apache.ibatis.annotations.Mapper;  
import org.springframework.stereotype.Repository;  
  
@Mapper  
@Repository  
public interface UserMapper extends BaseMapper<SystemUser> {  
}

## 6.工具类

JwtUtil

import io.jsonwebtoken.Claims;  
import io.jsonwebtoken.JwtBuilder;  
import io.jsonwebtoken.Jwts;  
import io.jsonwebtoken.SignatureAlgorithm;  
import javax.crypto.SecretKey;  
import javax.crypto.spec.SecretKeySpec;  
import java.util.Base64;  
import java.util.Date;  
import java.util.UUID;

*/\*\*  
 \* JWT工具类  
 \*/*public class JwtUtil {  
  
 //有效期为  
 public static final Long *JWT\_TTL* = 60 \* 60 \*1000L;// 60 \* 60 \*1000 一个小时  
 //设置秘钥明文  
 public static final String *JWT\_KEY* = "sangeng";  
  
 public static String getUUID(){  
 String token = UUID.*randomUUID*().toString().replaceAll("-", "");  
 return token;  
 }  
  
 */\*\*  
 \* 生成jtw  
 \** ***@param*** *subject token中要存放的数据（json格式）  
 \** ***@return*** *\*/* public static String createJWT(String subject) {  
 JwtBuilder builder = *getJwtBuilder*(subject, null, *getUUID*());// 设置过期时间  
 return builder.compact();  
 }  
  
 */\*\*  
 \* 生成jtw  
 \** ***@param*** *subject token中要存放的数据（json格式）  
 \** ***@param*** *ttlMillis token超时时间  
 \** ***@return*** *\*/* public static String createJWT(String subject, Long ttlMillis) {  
 JwtBuilder builder = *getJwtBuilder*(subject, ttlMillis, *getUUID*());// 设置过期时间  
 return builder.compact();  
 }  
  
 private static JwtBuilder getJwtBuilder(String subject, Long ttlMillis, String uuid) {  
 SignatureAlgorithm signatureAlgorithm = SignatureAlgorithm.*HS256*;  
 SecretKey secretKey = *generalKey*();  
 long nowMillis = System.*currentTimeMillis*();  
 Date now = new Date(nowMillis);  
 if(ttlMillis==null){  
 ttlMillis= JwtUtil.*JWT\_TTL*;  
 }  
 long expMillis = nowMillis + ttlMillis;  
 Date expDate = new Date(expMillis);  
 return Jwts.*builder*()  
 .setId(uuid) //唯一的ID  
 .setSubject(subject) // 主题 可以是JSON数据  
 .setIssuer("sg") // 签发者  
 .setIssuedAt(now) // 签发时间  
 .signWith(signatureAlgorithm, secretKey) //使用HS256对称加密算法签名, 第二个参数为秘钥  
 .setExpiration(expDate);  
 }  
  
 */\*\*  
 \* 创建token  
 \** ***@param*** *id  
 \** ***@param*** *subject  
 \** ***@param*** *ttlMillis  
 \** ***@return*** *\*/* public static String createJWT(String id, String subject, Long ttlMillis) {  
 JwtBuilder builder = *getJwtBuilder*(subject, ttlMillis, id);// 设置过期时间  
 return builder.compact();  
 }  
  
  
  
 */\*\*  
 \* 生成加密后的秘钥 secretKey  
 \** ***@return*** *\*/* public static SecretKey generalKey() {  
 byte[] encodedKey = Base64.*getDecoder*().decode(JwtUtil.*JWT\_KEY*);  
 SecretKey key = new SecretKeySpec(encodedKey, 0, encodedKey.length, "AES");  
 return key;  
 }  
  
 */\*\*  
 \* 解析  
 \*  
 \** ***@param*** *jwt  
 \** ***@return*** *\** ***@throws*** *Exception  
 \*/* public static Claims parseJWT(String jwt) throws Exception {  
 SecretKey secretKey = *generalKey*();  
 return Jwts.*parser*()  
 .setSigningKey(secretKey)  
 .parseClaimsJws(jwt)  
 .getBody();  
 }  
  
 public static void main(String[] args) throws Exception {  
 String jwt = *createJWT*("Alice");//jwt=eyJhbGciOiJIUzI1NiJ9.eyJqdGkiOiI0ZWQzMzViZDUwMDA0ODcxYTYwMGExZWNkOWY0ZWRhNiIsInN1YiI6IkFsaWNlIiwiaXNzIjoic2ciLCJpYXQiOjE2NTI2MjEwNDUsImV4cCI6MTY1MjYyNDY0NX0.wiN0A8S98XldU0hbQTuFc0KruG0AC59ZRNOPcArL5u4  
 System.*out*.println(jwt);  
 Claims claims = *parseJWT*(jwt);//chaims={jti=4ed335bd50004871a600a1ecd9f4eda6, sub=Alice, iss=sg, iat=1652621045, exp=1652624645}  
 System.*out*.println(claims);  
 String subject = claims.getSubject();//subject=Alice  
 System.*out*.println(subject);  
  
 }  
}

## 7.核心类

### （1）UserDetailsServiceImpl

创建一个类实现UserDetailsService接口，重写loadUserByUsername()方法。loadUserByUsername()方法的作用是查询数据库中用户为name的用户信息，将数据库查询到的用户名、用户密码和用户权限封装在UserDetails中,并返回UserDetails对象。

import cn.gdsdxy.springboot25security\_vue.pojo.SystemUser;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.security.core.authority.SimpleGrantedAuthority;  
import org.springframework.security.core.userdetails.User;  
import org.springframework.security.core.userdetails.UserDetails;  
import org.springframework.security.core.userdetails.UserDetailsService;  
import org.springframework.security.core.userdetails.UsernameNotFoundException;  
import org.springframework.stereotype.Service;  
import java.util.ArrayList;  
import java.util.List;  
  
//UserDetailsService是spring Security提供的用于封装认证用户信息的接口  
@Service  
public class UserDetailsServiceImpl implements UserDetailsService {  
 @Autowired  
 private UserService userService;  
  
 @Override  
 */\*\*  
 \* 该方法的作用是查询数据库中用户为name的用户信息，将数据库查询到的用户名、用户密码和用户权限封装在UserDetails中。  
 \*/* public UserDetails loadUserByUsername(String name) throws UsernameNotFoundException {  
 // 通过自定义的方法获取用户及权限信息  
 SystemUser user=userService.getUser(name);  
 // 对用户权限进行封装，将SystemUser对象中的authorities字段转换成List<SimpleGrantedAuthority类型  
 List<SimpleGrantedAuthority> list = new ArrayList<SimpleGrantedAuthority>();  
 list.add(new SimpleGrantedAuthority(user.getAuthority()));  
 if (user != null) {  
 //User类是框架提供的类，是UserDetails接口的实现类，用来存放用户名、用户密码、用户权限等信息。  
 UserDetails userDetails = new User(user.getName(), user.getPwd(), list);  
 return userDetails; // 返回封装的UserDetails用户详情类  
 } else {  
 // 如果查询的用户不存在（用户名不存在），必须抛出此异常  
 throw new UsernameNotFoundException("当前用户不存在！");  
 }  
 }

### （2）UserService

在上面的UserDetailsServiceImpl 类中需要用到UserService，该类的主要功能是获取用户为name的用户信息。

import cn.gdsdxy.springboot25security\_vue.pojo.SystemUser;  
import cn.gdsdxy.springboot25security\_vue.repository.UserMapper;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.data.redis.core.RedisTemplate;  
import org.springframework.stereotype.Service;

//在UserDetailsServiceImpl类中需要用到UserService，UserService是自定义的类，该类的主要功能是获取用户为name的用户对象和用户权限，并把获取到的信息存放在缓存中。  
@Service  
public class UserService {  
 @Autowired  
 UserMapper userMapper;  
 @Autowired  
 RedisTemplate redisTemplate;  
 //根据name获取对应的SystemUser对象，该SystemUser对象包含了用户名、密码和权限。如果一个用户有多个权限，最好在本类中再定义一个独立的方法获取用户权限。由于本例中一个用户只对应一个权限，所以只用一个方法获取用户名、密码和权限  
 public SystemUser getUser(String name){  
 SystemUser user = null;  
 //判断缓存中是否存在键为“user\_XXX"的缓存对象  
 Object o= redisTemplate.opsForValue().get("user\_"+name);  
 if(o!=null){  
 user=(SystemUser)o;  
 }else {  
 user = userMapper.selectById(name);  
 if(user!=null){  
 //将数据库查询到的SystemUser对象存放在缓存中，键名为"user\_XXX"  
 redisTemplate.opsForValue().set("user\_"+name,user);  
 }  
 }  
 return user;  
 }  
// //根据name获取该用户的权限，如果一个用户有多个权限，最好保留该方法。由于本例中一个用户只对应一个权限，所以可以删掉该方法  
// public List getAuthority(String name) {   
// return null;  
// }  
}

### （3）LoginController

import cn.gdsdxy.springboot25security\_vue.pojo.SystemUser;  
import cn.gdsdxy.springboot25security\_vue.service.LoginService;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.http.ResponseEntity;  
import org.springframework.stereotype.Controller;  
import org.springframework.web.bind.annotation.\*;

@Controller  
public class LoginController {  
 @Autowired  
 LoginService loginService;  
 @PostMapping("/userLogin")  
 @ResponseBody  
 public ResponseEntity<String> login(@RequestBody SystemUser user) {  
 ResponseEntity<String> responseEntity = loginService.login(user);  
 return responseEntity;  
 }  
}

### （4）LoginService

import cn.gdsdxy.springboot25security\_vue.util.JwtUtil;  
import cn.gdsdxy.springboot25security\_vue.pojo.SystemUser;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.data.redis.core.RedisTemplate;  
import org.springframework.http.HttpStatus;  
import org.springframework.http.ResponseEntity;  
import org.springframework.security.authentication.AuthenticationManager;  
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;  
import org.springframework.security.core.Authentication;  
import org.springframework.security.core.context.SecurityContextHolder;  
import org.springframework.security.core.userdetails.UserDetails;  
import org.springframework.stereotype.Service;  
import org.springframework.util.LinkedMultiValueMap;  
import org.springframework.util.MultiValueMap;  
import java.util.Objects;

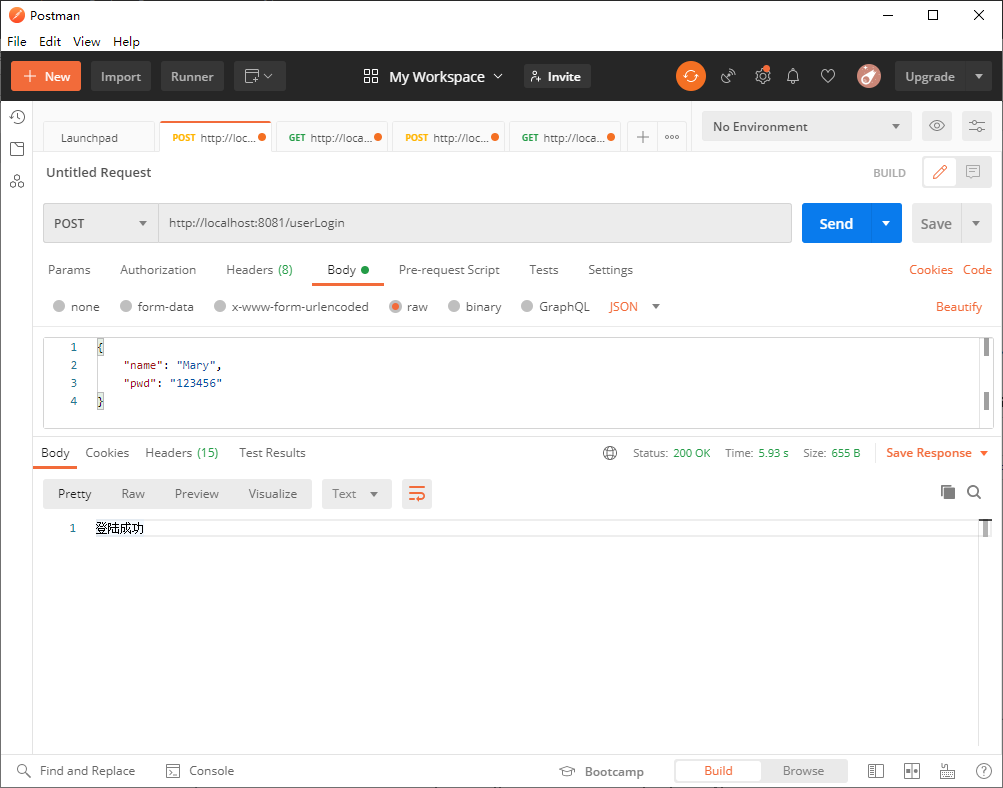
/\*  
Authentication接口: 它的实现类封装了当前访问的用户的相关信息。  
AuthenticationManager接口：定义了认证Authentication的方法  
UserDetailsService接口（自定义的实现类为UserDetailsServiceImpl）：接口中的loadUserByUsername()用来根据用户名查询用户的详细信息。  
UserDetails接口：封装了用户的详细信息。通过UserDetailsService接口中的loadUserByUsername()方法将数据库查询到的用户名、密码、权限封装成UserDetails对象返回，然后将这些信息封装到Authentication对象中。  
\*/  
@Service  
public class LoginService {  
 @Autowired  
 UserService userService;  
 @Autowired  
 private AuthenticationManager authenticationManager;//在WebSecurityConfigurerAdapter的子类SecurityConfig类（SecurityConfig是自定义的类）中，通过重载authenticationManagerBean()方法并在方法上方加上@Bean让框架的AuthenticationManager对象可以被外部程序使用  
 @Autowired  
 private RedisTemplate redisTemplate;  
 public ResponseEntity<String> login(SystemUser user) {  
 //调用AuthenticationManager对象的authenticate方法进行用户认证  
 UsernamePasswordAuthenticationToken authenticationToken = new UsernamePasswordAuthenticationToken(user.getName(),user.getPwd());  
 Authentication authenticate = authenticationManager.authenticate(authenticationToken);//默认调用UserDetailsServiceImpl类中的loadUserByUsername获取用户信息并进行认证，把用户信息封装在authenticate对象中  
 if(Objects.*isNull*(authenticate)){  
 throw new RuntimeException("用户名或密码错误");  
 }  
 //在本例中使用SystemUser.name生成token  
 UserDetails loginUser = (UserDetails) authenticate.getPrincipal();//获取authenticate对象中的UserDetails对象  
 String username=loginUser.getUsername();  
 String jwt = JwtUtil.*createJWT*(username);//利用username生成token  
 //authenticate存入redis  
 redisTemplate.opsForValue().set("login\_"+username,loginUser);  
 //把token响应给前端  
 MultiValueMap<String,String> header=new LinkedMultiValueMap<>();  
 header.add("token",jwt);  
 System.*out*.println(jwt);  
 return new ResponseEntity("登录成功",header, HttpStatus.*OK*);  
 }  
}

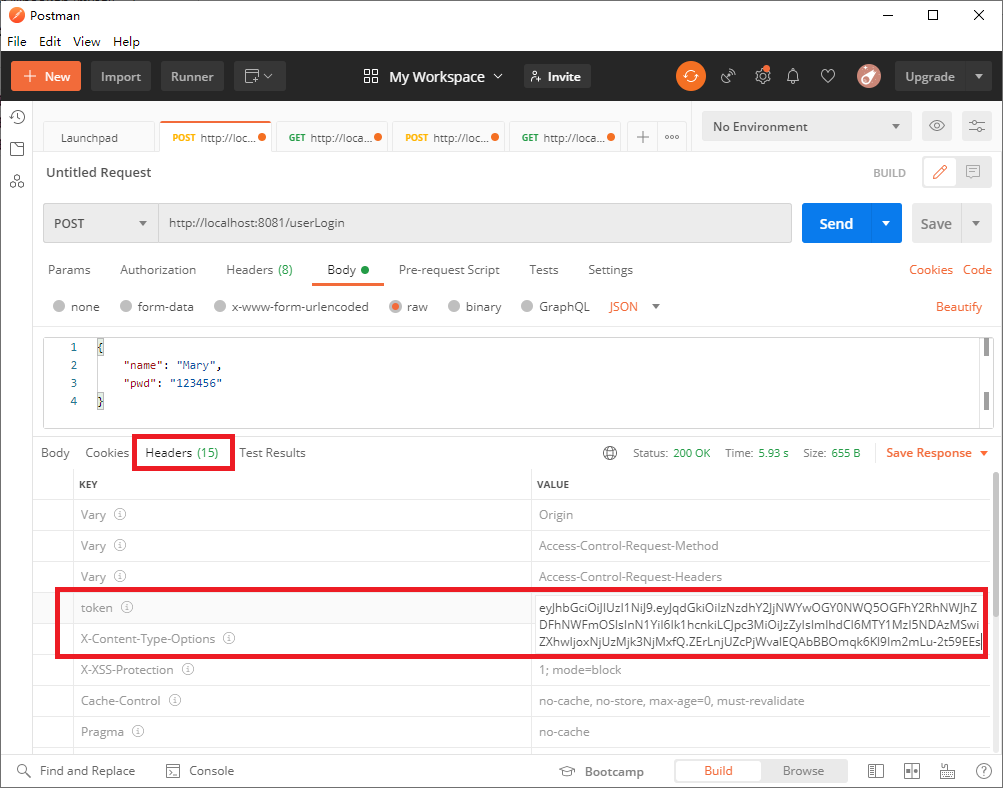
### （5）SecurityConfig

import cn.gdsdxy.springboot25security\_vue.service.UserDetailsServiceImpl;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.context.annotation.Bean;  
import org.springframework.security.authentication.AuthenticationManager;  
import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;  
import org.springframework.security.config.annotation.method.configuration.EnableGlobalMethodSecurity;  
import org.springframework.security.config.annotation.web.builders.HttpSecurity;  
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;  
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;  
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;  
import org.springframework.security.web.AuthenticationEntryPoint;  
import org.springframework.security.web.access.AccessDeniedHandler;  
import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;  
import org.springframework.web.cors.CorsConfiguration;  
import org.springframework.web.cors.CorsConfigurationSource;  
import org.springframework.web.cors.UrlBasedCorsConfigurationSource;  
import java.util.Arrays;

@EnableWebSecurity // 开启security安全支持  
public class SecurityConfig extends WebSecurityConfigurerAdapter {  
 @Autowired  
 private UserDetailsServiceImpl userDetailsService;  
 */\*\*  
 \* 用户身份认证自定义配置  
 \*/* @Override  
 protected void configure(AuthenticationManagerBuilder auth) throws Exception {  
 // 密码需要设置编码器  
 BCryptPasswordEncoder encoder = new BCryptPasswordEncoder();  
 // 使用UserDetailsService进行身份认证  
 auth.userDetailsService(userDetailsService).passwordEncoder(encoder);  
 }  
  
 */\*\*  
 \* 在LoginServiceImpl需要用到AuthenticationManager对象进行用户认证  
 \** ***@return*** *\** ***@throws*** *Exception  
 \*/* @Override  
 @Bean  
 public AuthenticationManager authenticationManagerBean() throws Exception {  
 return super.authenticationManagerBean();  
 }  
  
 @Override  
 protected void configure(HttpSecurity http) throws Exception {  
 http.cors();//允许跨域  
 http.csrf().disable();//跨域要加上该语句，否则只能获取的get请求，post请求将被拦截（即使加了 .antMatchers("/userLogin").permitAll()语句）  
 http.authorizeRequests()  
 .antMatchers("/userLogin").permitAll()  
 .anyRequest().authenticated();  
 http.formLogin();  
 }  
 //SecurityConfig中配置开启CORS默认会找name为corsConfigurationSource的bean  
 @Bean  
 CorsConfigurationSource corsConfigurationSource() {  
 // 提供CorsConfiguration实例，并配置跨域信息  
 CorsConfiguration corsConfiguration = new CorsConfiguration();  
 corsConfiguration.setAllowedHeaders(Arrays.*asList*("token"));  
 corsConfiguration.setAllowedMethods(Arrays.*asList*("\*"));  
 corsConfiguration.setAllowedOrigins(Arrays.*asList*("http://127.0.0.1"));  
 corsConfiguration.setMaxAge(3600L);  
 UrlBasedCorsConfigurationSource source;  
 source = new UrlBasedCorsConfigurationSource();  
 source.registerCorsConfiguration("/\*\*", corsConfiguration);  
 return source;  
 }  
}

## 8.测试





# 四、认证模块的代码实现

自定义一个过滤器，该过滤器用来对所有请求进行验证，判断请求的header中是否携带token，如果携带token则验证token的合法性。通过解析token获取用户名username，再从缓存中获取该username的用户信息，将用户信息存到SecurityContextHolder中。

import cn.gdsdxy.springboot25security\_vue.util.JwtUtil;

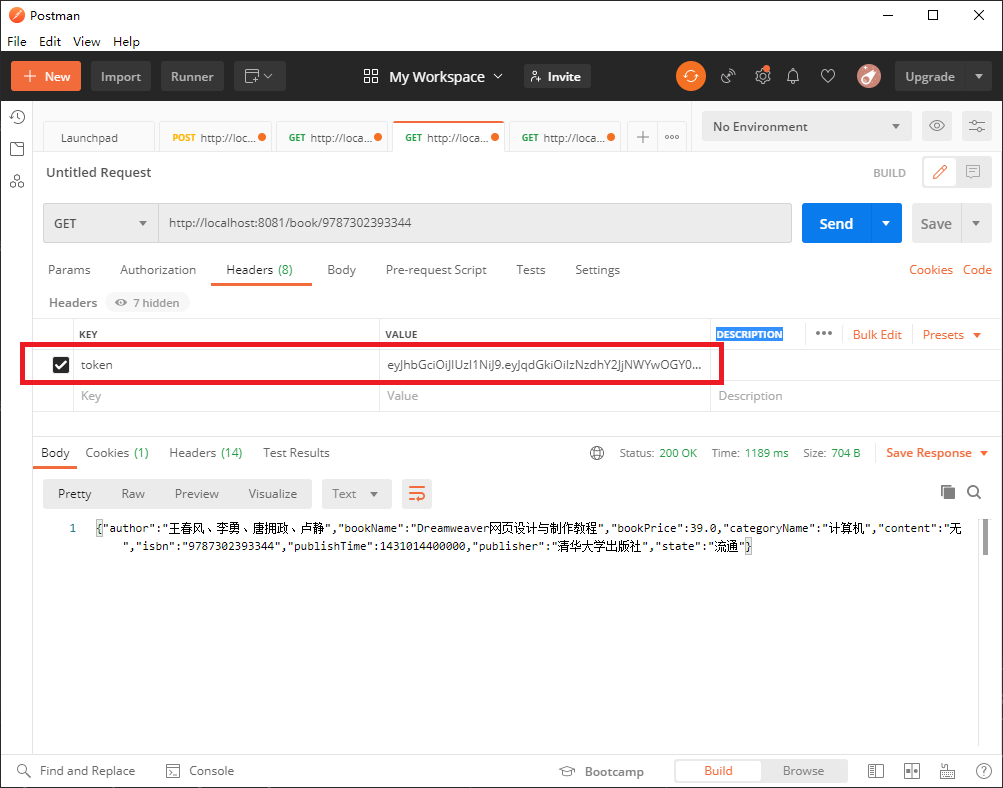
import io.jsonwebtoken.Claims;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.data.redis.core.RedisTemplate;  
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;  
import org.springframework.security.core.context.SecurityContextHolder;  
import org.springframework.security.core.userdetails.User;  
import org.springframework.stereotype.Component;  
import org.springframework.util.StringUtils;  
import org.springframework.web.filter.OncePerRequestFilter;  
import javax.servlet.FilterChain;  
import javax.servlet.ServletException;  
import javax.servlet.http.HttpServletRequest;  
import javax.servlet.http.HttpServletResponse;  
import java.io.IOException;  
import java.util.Objects;

*/\*\*  
 \* 自定义一个过滤器，该过滤器用来对所有请求进行验证，判断请求的header中是否携带token，如果携带token则验证token的合法性。通过解析token获取用户名username，再从缓存中获取该username的用户信息，将用户信息存到SecurityContextHolder中。  
 \*/*  
@Component  
public class JwtAuthenticationTokenFilter extends OncePerRequestFilter {  
  
 @Autowired  
 private RedisTemplate redisTemplate;  
  
 @Override  
 protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain) throws ServletException, IOException {  
 //获取token  
 String token = request.getHeader("token");  
 if (!StringUtils.*hasText*(token)) {  
 //如果请求头中不包含token就直接放行  
 filterChain.doFilter(request, response);  
 return;  
 }  
 //解析token  
 String username;  
 try {  
 Claims claims = JwtUtil.*parseJWT*(token);//解析token  
 username = claims.getSubject();//从token中获取用户名  
 } catch (Exception e) {  
 e.printStackTrace();  
 throw new RuntimeException("token非法");  
 }  
 //从redis中获取用户信息  
 System.*out*.println(username);  
 String redisKey = "login\_" + username;  
 User loginUser =(User) redisTemplate.opsForValue().get(redisKey);//获取缓存中的用户信息  
 System.*out*.println(loginUser);  
 if(Objects.*isNull*(loginUser)){//如果缓存没有数据，说明用户未登录  
 throw new RuntimeException("用户未登录");  
 }  
 //将获取到的用户信息存入SecurityContextHolder

//UsernamePasswordAuthenticationToken构造方法的三个参数： principal一般指的是用户名，credentials一般指的是密码，authorities指权限  
 UsernamePasswordAuthenticationToken authenticationToken =new UsernamePasswordAuthenticationToken(loginUser.getUsername(),loginUser.getPassword(),loginUser.getAuthorities());  
 SecurityContextHolder.*getContext*().setAuthentication(authenticationToken);  
 //放行  
 filterChain.doFilter(request, response);  
 }  
}

修改之前创建的SecurityConfig类，将上面创建的过滤器添加到安全框架的过滤器链中。

@EnableWebSecurity // 开启MVC security安全支持  
//@Order(Ordered.HIGHEST\_PRECEDENCE)  
public class SecurityConfig extends WebSecurityConfigurerAdapter {  
 @Autowired  
 private UserDetailsServiceImpl userDetailsService;  
 @Autowired  
 private JwtAuthenticationTokenFilter jwtAuthenticationTokenFilter;  
 */\*\*  
 \* 用户身份认证自定义配置  
 \*/* @Override  
 protected void configure(AuthenticationManagerBuilder auth) throws Exception {  
 // 密码需要设置编码器  
 BCryptPasswordEncoder encoder = new BCryptPasswordEncoder();  
 // 使用UserDetailsService进行身份认证  
 auth.userDetailsService(userDetailsService).passwordEncoder(encoder);  
 }  
  
 */\*\*  
 \* 在LoginServiceImpl需要用到AuthenticationManager对象进行用户认证  
 \** ***@return*** *\** ***@throws*** *Exception  
 \*/* @Override  
 @Bean  
 public AuthenticationManager authenticationManagerBean() throws Exception {  
 return super.authenticationManagerBean();  
 }  
  
 @Override  
 protected void configure(HttpSecurity http) throws Exception {  
 http.cors();//允许跨域  
 http.csrf().disable();//跨域要加上该语句，否则只能获取的get请求，post请求将被拦截（即使加了 .antMatchers("/userLogin").permitAll()语句）  
 http.authorizeRequests()  
 .antMatchers("/userLogin").permitAll()  
 .anyRequest().authenticated();  
 http.formLogin();  
  
 //把token校验过滤器添加到过滤器链中  
 http.addFilterBefore(jwtAuthenticationTokenFilter, UsernamePasswordAuthenticationFilter.class);  
 }  
  
 //SecurityConfig中配置开启CORS默认会找name为corsConfigurationSource的bean  
 @Bean  
 CorsConfigurationSource corsConfigurationSource() {  
 // 提供CorsConfiguration实例，并配置跨域信息  
 CorsConfiguration corsConfiguration = new CorsConfiguration();  
 corsConfiguration.setAllowedHeaders(Arrays.*asList*("token"));  
 corsConfiguration.setAllowedMethods(Arrays.*asList*("\*"));  
 corsConfiguration.setAllowedOrigins(Arrays.*asList*("http://127.0.0.1"));  
 corsConfiguration.setMaxAge(3600L);  
 UrlBasedCorsConfigurationSource source;  
 source = new UrlBasedCorsConfigurationSource();  
 source.registerCorsConfiguration("/\*\*", corsConfiguration);  
 return source;  
 }  
}



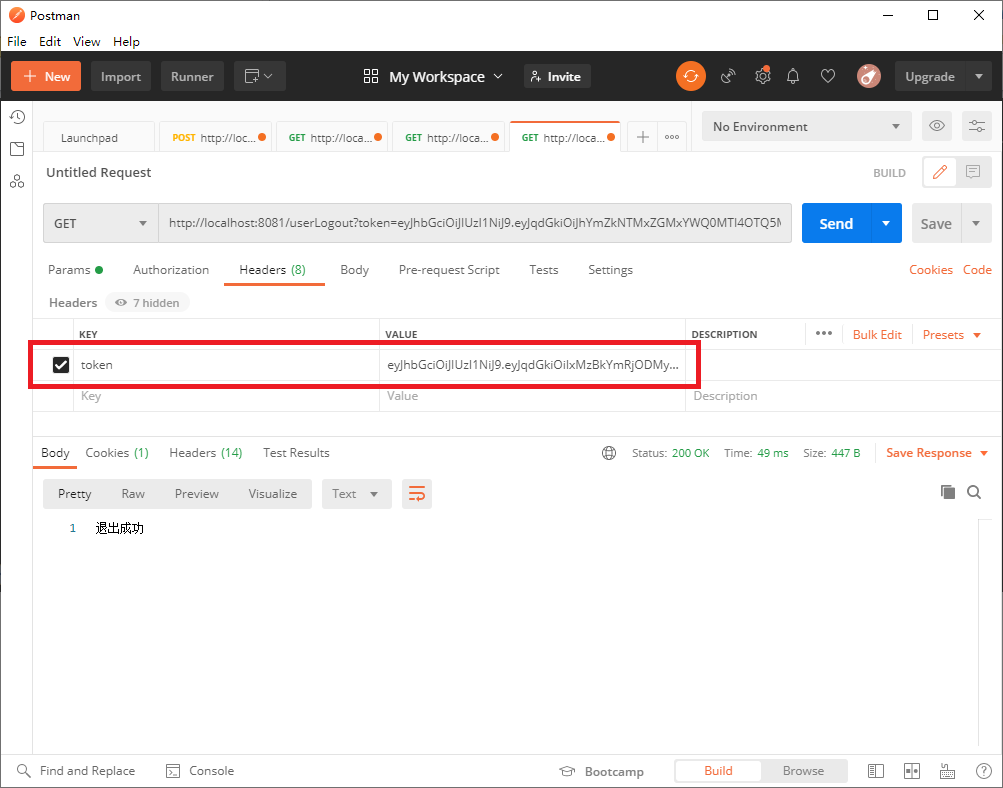
# 五、退出模块的代码实现

在之前自定义的LoginService中增加logout()方法实现用户注销，当注销用户时，删除缓存中的用户信息。

/\*  
Authentication接口: 它的实现类封装了当前访问的用户的相关信息。  
AuthenticationManager接口：定义了认证Authentication的方法  
UserDetailsService接口（自定义的实现类为UserDetailsServiceImpl）：接口中的loadUserByUsername()用来根据用户名查询用户的详细信息。  
UserDetails接口：封装了用户的详细信息。通过UserDetailsService接口中的loadUserByUsername()方法将数据库查询到的用户名、密码、权限封装成UserDetails对象返回，然后将这些信息封装到Authentication对象中。  
\*/  
@Service  
public class LoginService {  
 @Autowired  
 UserService userService;  
 @Autowired  
 private AuthenticationManager authenticationManager;//在WebSecurityConfigurerAdapter的子类SecurityConfig类中，  
 // 通过重载authenticationManagerBean()方法并在方法上方加上@Bean让框架的AuthenticationManager对象可以被外部程序使用  
 @Autowired  
 private RedisTemplate redisTemplate;  
 public ResponseEntity<String> login(SystemUser user) {  
 //调用authenticationManager对象的authenticate方法进行用户认证  
 UsernamePasswordAuthenticationToken authenticationToken = new UsernamePasswordAuthenticationToken(user.getName(),user.getPwd());  
 Authentication authenticate = authenticationManager.authenticate(authenticationToken);//默认调用UserDetailsServiceImpl类中的loadUserByUsername获取用户信息并进行认证，把用户信息封装在authenticate对象中  
 if(Objects.*isNull*(authenticate)){  
 throw new RuntimeException("用户名或密码错误");  
 }  
 //在本例中使用SystemUser.name生成token  
 UserDetails loginUser = (UserDetails) authenticate.getPrincipal();//获取authenticate对象中的UserDetails对象  
 String username=loginUser.getUsername();  
 String jwt = JwtUtil.*createJWT*(username);//利用username生成token  
// //authenticate存入redis  
 redisTemplate.opsForValue().set("login\_"+username,loginUser);  
 //把token响应给前端  
 MultiValueMap<String,String> header=new LinkedMultiValueMap<>();  
 header.add("token",jwt);  
 System.*out*.println(jwt);  
 return new ResponseEntity("登陆成功",header, HttpStatus.*OK*);  
 }  
 public ResponseEntity logout() {  
 //在自定义的过滤器JwtAuthenticationTokenFilter中，只要输入的token合法，就通过SecurityContextHolder.getContext().setAuthentication(authenticationToken)语句将对应的用户信息写入就通过SecurityContextHolder中  
 Authentication authentication = SecurityContextHolder.*getContext*().getAuthentication();//获取SecurityContextHolder中存放的用户信息  
 String username = (String) authentication.getPrincipal();//获取authenticate对象中的UserDetails对象  
 redisTemplate.delete("login\_"+username);  
 return new ResponseEntity("退出成功",HttpStatus.*OK*);  
 }  
}

在控制器类的添加用户退出操作的请求。

@Controller  
public class LoginController {  
 @Autowired  
 LoginService loginService;  
  
 @PostMapping("/userLogin")  
 @ResponseBody  
 public ResponseEntity<String> login(@RequestBody SystemUser user) {  
 ResponseEntity<String> responseEntity = loginService.login(user);  
 return responseEntity;  
 }  
  
 //执行该请求必须携带token，否则运行产生异常  
 @GetMapping("/userLogout")  
 @ResponseBody  
 public ResponseEntity<String> logout() {  
 ResponseEntity<String> responseEntity = loginService.logout();  
 return responseEntity;  
 }  
}



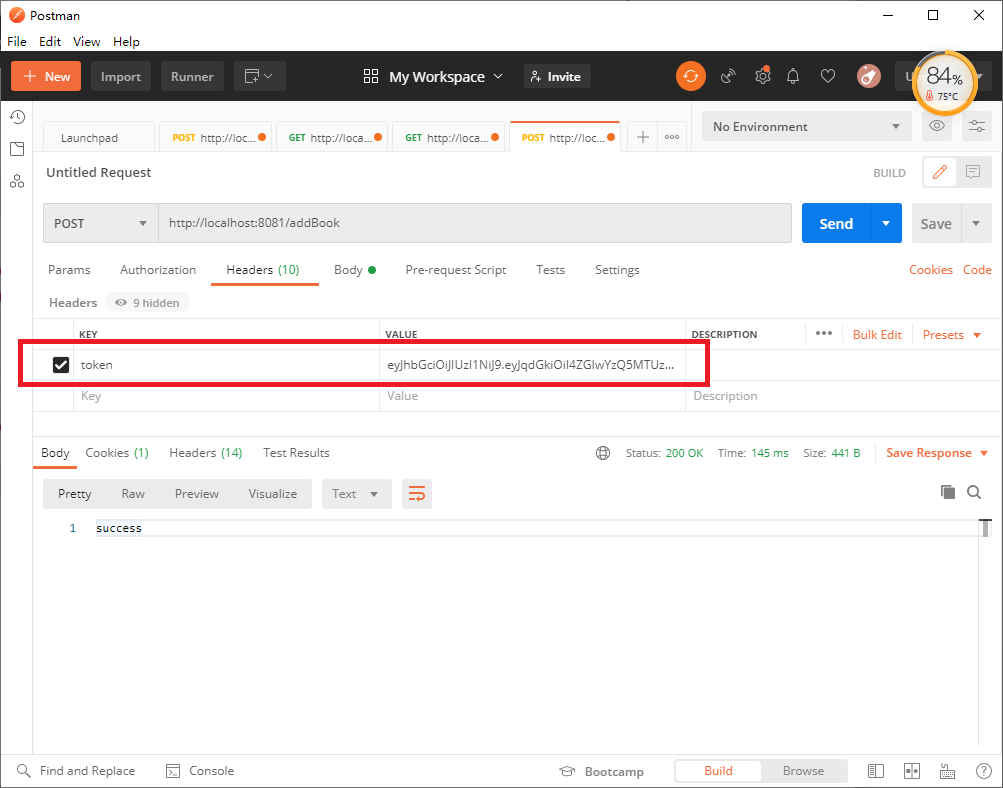
# 六、权限管理

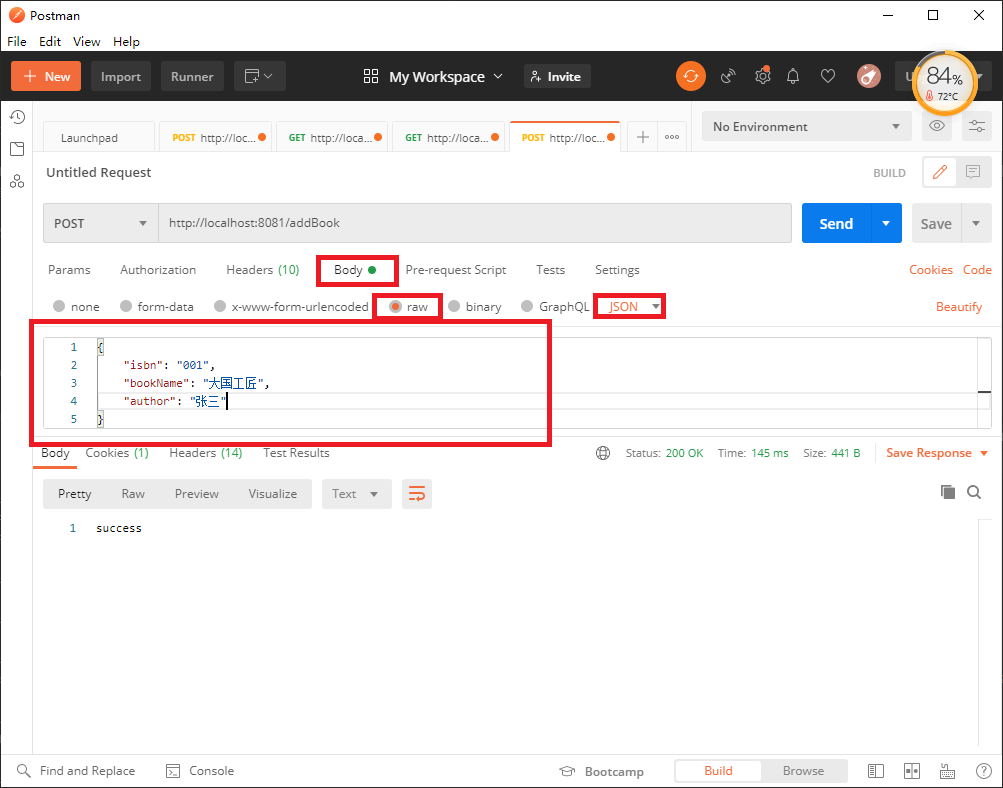
在自定义的SecurityConfig 类（WebSecurityConfigurerAdapter的实现类）上方加上@EnableGlobalMethodSecurity(prePostEnabled = true)，该注解用来开启基于注解的权限管理，加上该注解后就可以在controller类中通过注解设置各个请求对应的权限。

@EnableWebSecurity // 开启MVC security安全支持  
@EnableGlobalMethodSecurity(prePostEnabled = true)//开启基于注解的权限管理，加上该注解后就可以在controller类中通过注解设置各个请求对应的权限  
public class SecurityConfig extends WebSecurityConfigurerAdapter {  
 //省略中间的代码  
}

在controller类中设置每个请求的权限

@RestController  
public class BookController {  
 @Autowired  
 BookService bookService;  
 @GetMapping("/books")  
 @PreAuthorize("hasAuthority('ROLE\_common')")//表示权限为ROLE\_common的用户可以执行该请求  
 public String getAllBook(){  
 List<Book> books=bookService.getAllBook();  
 String jsonString = JSONObject.*toJSONString*(books);  
 System.*out*.println("books"+jsonString);  
 return jsonString;  
 }  
 @DeleteMapping("/book/{isbn}")  
 @PreAuthorize("hasAuthority('ROLE\_vip')")  
 public String deleteBook(@PathVariable("isbn") String isbn){  
 int i=bookService.delete(isbn);  
 if(i>0)  
 return "删除成功";  
 else  
 return "删除失败";  
 }  
 @GetMapping("/book/{isbn}")  
 @PreAuthorize("hasAuthority('ROLE\_common')")  
 public String getBookbyId(@PathVariable("isbn") String isbn){  
 Book book=bookService.getBookByIsbn(isbn);  
 String jsonBook = JSONObject.*toJSONString*(book);  
 return jsonBook;  
 }  
 @PostMapping("/addBook")  
 @PreAuthorize("hasAuthority('ROLE\_vip')")  
 public String addBook(@RequestBody Book book){  
 if(bookService.insert(book)>0)  
 return "success";  
 else  
 return "fail";  
 }  
 @PostMapping("/updateBook")  
 @PreAuthorize("hasAuthority('ROLE\_vip')")  
 public String updateBook(@RequestBody Book book){  
 if(bookService.update(book)>0)  
 return "success";  
 else  
 return "fail";  
 }  
}





# 七、自定义失败处理

​ 我们还希望在认证失败或者是授权失败的情况下也能和我们的接口一样返回相同结构的json，这样可以让前端能对响应进行统一的处理。要实现这个功能我们需要知道SpringSecurity的异常处理机制。

​ 在SpringSecurity中，如果我们在认证或者授权的过程中出现了异常会被ExceptionTranslationFilter捕获到。在ExceptionTranslationFilter中会去判断是认证失败还是授权失败出现的异常。

​ 如果是认证过程中出现的异常会被封装成AuthenticationException，然后调用AuthenticationEntryPoint 对象的方法去进行异常处理。

​ 如果是授权过程中出现的异常会被封装成AccessDeniedException，然后调用AccessDeniedHandler对象的方法去进行异常处理。

​ 所以如果我们需要自定义异常处理，我们只需要自定义AuthenticationEntryPoint和AccessDeniedHandler，然后在WebSecurityConfigurerAdapter的子类中配置给SpringSecurity即可。

import com.alibaba.fastjson.JSON;  
import org.springframework.http.HttpStatus;  
import org.springframework.http.ResponseEntity;  
import org.springframework.security.core.AuthenticationException;  
import org.springframework.security.web.AuthenticationEntryPoint;  
import org.springframework.stereotype.Component;  
import javax.servlet.ServletException;  
import javax.servlet.http.HttpServletRequest;  
import javax.servlet.http.HttpServletResponse;  
import java.io.IOException;

@Component  
public class AuthenticationEntryPointImpl implements AuthenticationEntryPoint {  
 @Override  
 public void commence(HttpServletRequest request, HttpServletResponse response, AuthenticationException authException) throws IOException, ServletException {  
 ResponseEntity<String> result = new ResponseEntity<String>( "认证失败请重新登录", HttpStatus.*UNAUTHORIZED*);//该错误提示可以通过前端代码获取：.catch(error => {this.msg=error.response.data.body;}  
 String json = JSON.*toJSONString*(result);  
 response.setStatus(401);  
 response.setContentType("application/json");  
 response.setCharacterEncoding("utf-8");  
 response.getWriter().print(json);  
 }  
}

import com.alibaba.fastjson.JSON;  
import org.springframework.http.HttpStatus;  
import org.springframework.http.ResponseEntity;  
import org.springframework.security.access.AccessDeniedException;  
import org.springframework.security.web.access.AccessDeniedHandler;  
import org.springframework.stereotype.Component;  
import javax.servlet.ServletException;  
import javax.servlet.http.HttpServletRequest;  
import javax.servlet.http.HttpServletResponse;  
import java.io.IOException;

@Component  
public class AccessDeniedHandlerImpl implements AccessDeniedHandler {  
 @Override  
 public void handle(HttpServletRequest request, HttpServletResponse response, AccessDeniedException accessDeniedException) throws IOException, ServletException {  
 ResponseEntity<String> result = new ResponseEntity("权限不足", HttpStatus.*FORBIDDEN*);//该错误提示可以通过前端代码获取：.catch(error => {this.msg=error.response.data.body;}  
 String json = JSON.*toJSONString*(result);  
 response.setStatus(403);  
 response.setContentType("application/json");  
 response.setCharacterEncoding("utf-8");  
 response.getWriter().print(json);  
 }  
}

@EnableWebSecurity // 开启MVC security安全支持  
@EnableGlobalMethodSecurity(prePostEnabled = true)//开启基于注解的权限管理，加上该注解后就可以在controller类中通过注解设置各个请求对应的权限  
//@Order(Ordered.HIGHEST\_PRECEDENCE)  
public class SecurityConfig extends WebSecurityConfigurerAdapter {  
 @Autowired  
 private UserDetailsServiceImpl userDetailsService;  
 @Autowired  
 private JwtAuthenticationTokenFilter jwtAuthenticationTokenFilter;  
 @Autowired  
 private AuthenticationEntryPoint authenticationEntryPoint;  
 @Autowired  
 private AccessDeniedHandler accessDeniedHandler;  
 */\*\*  
 \* 用户身份认证自定义配置  
 \*/* @Override  
 protected void configure(AuthenticationManagerBuilder auth) throws Exception {  
 // 密码需要设置编码器  
 BCryptPasswordEncoder encoder = new BCryptPasswordEncoder();  
 // 使用UserDetailsService进行身份认证  
 auth.userDetailsService(userDetailsService).passwordEncoder(encoder);  
 }  
  
 */\*\*  
 \* 在LoginServiceImpl需要用到AuthenticationManager对象进行用户认证  
 \** ***@return*** *\** ***@throws*** *Exception  
 \*/* @Override  
 @Bean  
 public AuthenticationManager authenticationManagerBean() throws Exception {  
 return super.authenticationManagerBean();  
 }  
  
 @Override  
 protected void configure(HttpSecurity http) throws Exception {  
 http.cors();//允许跨域  
 http.csrf().disable();//跨域要加上该语句，否则只能获取的get请求，post请求将被拦截（即使加了 .antMatchers("/userLogin").permitAll()语句）  
 http.authorizeRequests()  
 .antMatchers("/userLogin").permitAll()  
 .anyRequest().authenticated();  
 http.formLogin();  
// .loginPage("/login").permitAll();  
  
 //把token校验过滤器添加到过滤器链中  
 http.addFilterBefore(jwtAuthenticationTokenFilter, UsernamePasswordAuthenticationFilter.class);  
 //自定义异常处理类  
 http.exceptionHandling().  
 authenticationEntryPoint(authenticationEntryPoint).  
 accessDeniedHandler(accessDeniedHandler);  
 }  
  
 //SecurityConfig中配置开启CORS默认会找name为corsConfigurationSource的bean  
 @Bean  
 CorsConfigurationSource corsConfigurationSource() {  
 // 提供CorsConfiguration实例，并配置跨域信息  
 CorsConfiguration corsConfiguration = new CorsConfiguration();  
 corsConfiguration.setAllowedHeaders(Arrays.*asList*("token"));  
 corsConfiguration.setAllowedMethods(Arrays.*asList*("\*"));  
 corsConfiguration.setAllowedOrigins(Arrays.*asList*("http://127.0.0.1"));  
 corsConfiguration.setMaxAge(3600L);  
 UrlBasedCorsConfigurationSource source;  
 source = new UrlBasedCorsConfigurationSource();  
 source.registerCorsConfiguration("/\*\*", corsConfiguration);  
 return source;  
 }  
}

# 八、前端部分代码

## 1.Vue中跨域的设置1

注意：不同的版本跨域设置不太一样。

（1）第一种方法

在config->index.js文件中添加以下内容：

module.exports = {

  dev: {

    // Paths

    assetsSubDirectory: 'static',

    assetsPublicPath: '/',

    proxyTable: {

      '/api': {

        target:'http://localhost:8081/',//这里的地址是后端数据接口的地址

        changeorigin: true, //是否跨域

        pathRewrite: {

          '^/api': ''}//这里设置为''表示跨域后的网址是不包含/api的

      }

},

……

（2）第二种方法

在vue.config.js中添加以下代码

module.exports = {

  configureWebpack: {

    devtool: 'source-map'

  },

  devServer: {

    proxy: {

      // /api是后台数据接口的上下文路径

      '/api': {

          //这里的地址是后端数据接口的地址

          target: 'http://localhost:8081/',

         //允许跨域

          changOrigin: true,

          pathRewrite: {

          '^/api': ''}//这里设置为''表示跨域后的网址是不包含/api的

      }

    }

  },

  productionSourceMap: process.env.NODE\_ENV === 'production' ? false : true

}

## 2.Vue中跨域的设置2

在main.js文件中添加该语句axios.defaults.baseURL = "/api"。

// The Vue build version to load with the `import` command

// (runtime-only or standalone) has been set in webpack.base.conf with an alias.

import Vue from 'vue'

import App from './App'

import router from './router'

import axios from 'axios'

import VueAxios from 'vue-axios'

import store from './store'

Vue.use(VueAxios, axios) //安装插件

Vue.config.productionTip = false

axios.defaults.baseURL = "/api"

/\* eslint-disable no-new \*/

new Vue({

  el: '#app',

  router,

  store:store,

  components: { App },

  template: '<App/>'

})

## 3.登录页面

<template>

  <div>

      <p>登录</p>

      <form>

          用户名：<input type="text" v-model="username" />

          密码：<input type="password" v-model="password" />

          <input type="submit" @click.prevent="login" value="登录" />

      </form>

      <p>{{msg}}</p>

  </div>

</template>

<script>

export default {

  name: 'login',

  data() {

    return {

      username: "",

      password: "",

      msg: ""

    };

  },

  methods: {

    login() {

      this.axios.post("/userLogin",{name: this.username, pwd: this.password})

        .then(response => {

          // console.log(response);//可以先通过该语句查看传回什么内容

          this.msg=response.data;

          sessionStorage.setItem('token',response.headers.token);//将后端传回的headers中的token存放在sessionStorage中

        })

        .catch(error => {

          alert(error.message)

        })

    }

  }

}

</script>

## 4.查看图书页面

<template>

  <div>

      <p>所有图书</p>

      <div v-for="book in books" :key="book.isbn">

            {{book.isbn}},{{book.bookname}},{{book.author}}

      </div>

      <p>{{msg}}</p>

  </div>

</template>

<script>

export default {

  name: 'book',

  data() {

    return {

      msg: '',

      books: []

    };

  },

  created() {

      this.axios.get("/books",{

        headers: {

            "token": sessionStorage.getItem('token')

          }

      })

      .then(response => {

           this.books=response.data;

        })

      .catch(error => {

          // console.log(error);

          if(error.response.status==500)

               this.msg=error.response.data.message;

          else

               this.msg=error.response.data.body;

        })

    },

  methods: {

  }

}

</script>

## 4.退出登录

<template>

   <div>

       {{msg}}

   </div>

</template>

<script>

export default {

    data(){

        return{

            msg:""

        }

    },

    created(){

             this.axios.get("/userLogout",{

                   headers: {

                    "token": sessionStorage.getItem('token')

                    }

             })

          .then(response => {

            if (response.status == 200) {

                 this.msg="退出成功";

              }else{

                 this.msg="退出失败";

              }

            }

          )

          .catch(error => {

            this.msg=error.response.data.message;

          })

    },

    methods:{

    }

}

</script>