**SpringSecurity（入门到精通）**

web项目的认证与授权

认证：确认是否是当前用户

判断当前的这个用户是否有权限

**一、快速入门**

**1.1搭建环境**

pom文件

<?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>

<groupId>org.example</groupId>

<artifactId>SpringSecurity\_test</artifactId>

<version>1.0-SNAPSHOT</version>

<parent>

<groupId>org.springframework.boot</groupId>

<version>2.6.0</version>

<artifactId>spring-boot-starter-parent</artifactId>

</parent>

<properties>

<maven.compiler.source>8</maven.compiler.source>

<maven.compiler.target>8</maven.compiler.target>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

<version>2.6.0</version>

</dependency>

<dependency>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

<version>1.18.24</version>

<optional>true</optional>

</dependency>

</dependencies>

</project>

搭建启动类

package zou;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class SecurityApplication {

public static void main(String[] args) {

SpringApplication.run(SecurityApplication.class,args);

}

}

**1.2 引入SpringSecurity**

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

**二、认证**

**2.1认证的流程**

依赖的是token，看用户是否携带token

前端携带用户名密码

服务器验证用户名密码是否正确（数据库中检验）

如果正确生成jwt，并将jwt返回前端

登录后的其他请求需要在请求头中携带token

服务器获取token并解析，看用户是否拥有相关的权限，如果有则进行下一步的操作

服务器给前端响应信息

**2.2 原理初探**

本质是一个过滤器链：由多个过滤器组成的过滤器链

**三大重要的过滤器**

**FilterSecurityInterceptor**

是一个方法权限的过滤器，基本位于过滤器的最低不

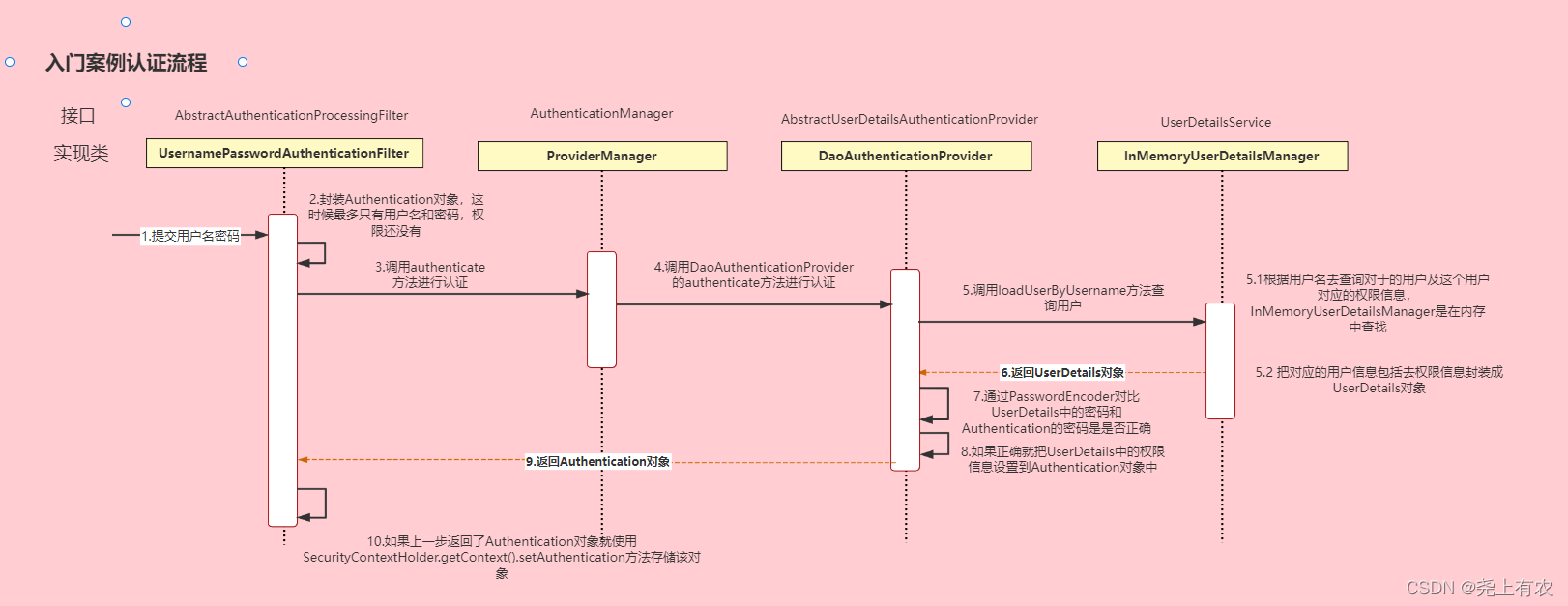
**ExceptionTranslationFilter**

**是一个异常处理器，处理在认证的过程中的异常**

**UsenamepasswordauthenticationFilter**

用户密码的过滤器

**流程图**



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。

**修改的流程图**

[Security的登录认证的修改图](note://WEB868c8f7d1076df17404afa5b03fb058c)

**解决问题的思路**

**登录**

**自定义登录接口**

调用ProviderManage的方法进行认证 如果认证通过则生成JWT

把用户存入redis中

**自定义UserDatailService**

在这个实现列中去查询数据

**校验**

JWT的认证过滤器

1.获取token

2.解析token

3.获取userid

4.封装Authentication并存入SecurityContexHolder对象

**2.3 准备工作**

**添加依赖**

<!--redis依赖-->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-redis</artifactId>

</dependency>

<!--fastjson依赖-->

<dependency>

<groupId>com.alibaba</groupId>

<artifactId>fastjson</artifactId>

<version>1.2.33</version>

</dependency>

<!--jwt依赖-->

<dependency>

<groupId>io.jsonwebtoken</groupId>

<artifactId>jjwt</artifactId>

<version>0.9.0</version>

</dependency>

**配置**

server:

port: 8091

spring:

datasource:

url: jdbc:mysql://localhost:3306/springsecurity?characterEncoding=utf-8&serverTimezone=UTC

username: root

password: root

driver-class-name: com.mysql.cj.jdbc.Driver

**sql语句建表**

CREATE TABLE `sys\_user` (

`id` BIGINT(20) NOT NULL AUTO\_INCREMENT COMMENT '主键',

`user\_name` VARCHAR(64) NOT NULL DEFAULT 'NULL' COMMENT '用户名',

`nick\_name` VARCHAR(64) NOT NULL DEFAULT 'NULL' COMMENT '昵称',

`password` VARCHAR(64) NOT NULL DEFAULT 'NULL' COMMENT '密码',

`status` CHAR(1) DEFAULT '0' COMMENT '账号状态（0正常 1停用）',

`email` VARCHAR(64) DEFAULT NULL COMMENT '邮箱',

`phonenumber` VARCHAR(32) DEFAULT NULL COMMENT '手机号',

`sex` CHAR(1) DEFAULT NULL COMMENT '用户性别（0男，1女，2未知）',

`avatar` VARCHAR(128) DEFAULT NULL COMMENT '头像',

`user\_type` CHAR(1) NOT NULL DEFAULT '1' COMMENT '用户类型（0管理员，1普通用户）',

`create\_by` BIGINT(20) DEFAULT NULL COMMENT '创建人的用户id',

`create\_time` DATETIME DEFAULT NULL COMMENT '创建时间',

`update\_by` BIGINT(20) DEFAULT NULL COMMENT '更新人',

`update\_time` DATETIME DEFAULT NULL COMMENT '更新时间',

`del\_flag` INT(11) DEFAULT '0' COMMENT '删除标志（0代表未删除，1代表已删除）',

PRIMARY KEY (`id`)

) ENGINE=INNODB AUTO\_INCREMENT=2 DEFAULT CHARSET=utf8mb4 COMMENT='用户表'

**启动类**

public class WebUtils

{

/\*\*

\* 将字符串渲染到客户端

\*

\* @param response 渲染对象

\* @param string 待渲染的字符串

\* @return null

\*/

public static String renderString(HttpServletResponse response, String string) {

try

{

response.setStatus(200);

response.setContentType("application/json");

response.setCharacterEncoding("utf-8");

response.getWriter().print(string);

}

catch (IOException e)

{

e.printStackTrace();

}

return null;

}

}

**添加Redis相关配置**

@SuppressWarnings(value = { "unchecked", "rawtypes" })

@Component

public class RedisCache

{

@Autowired

public RedisTemplate redisTemplate;

/\*\*

\* 缓存基本的对象，Integer、String、实体类等

\*

\* @param key 缓存的键值

\* @param value 缓存的值

\*/

public <T> void setCacheObject(final String key, final T value)

{

redisTemplate.opsForValue().set(key, value);

}

/\*\*

\* 缓存基本的对象，Integer、String、实体类等

\*

\* @param key 缓存的键值

\* @param value 缓存的值

\* @param timeout 时间

\* @param timeUnit 时间颗粒度

\*/

public <T> void setCacheObject(final String key, final T value, final Integer timeout, final TimeUnit timeUnit)

{

redisTemplate.opsForValue().set(key, value, timeout, timeUnit);

}

/\*\*

\* 设置有效时间

\*

\* @param key Redis键

\* @param timeout 超时时间

\* @return true=设置成功；false=设置失败

\*/

public boolean expire(final String key, final long timeout)

{

return expire(key, timeout, TimeUnit.SECONDS);

}

/\*\*

\* 设置有效时间

\*

\* @param key Redis键

\* @param timeout 超时时间

\* @param unit 时间单位

\* @return true=设置成功；false=设置失败

\*/

public boolean expire(final String key, final long timeout, final TimeUnit unit)

{

return redisTemplate.expire(key, timeout, unit);

}

/\*\*

\* 获得缓存的基本对象。

\*

\* @param key 缓存键值

\* @return 缓存键值对应的数据

\*/

public <T> T getCacheObject(final String key)

{

ValueOperations<String, T> operation = redisTemplate.opsForValue();

return operation.get(key);

}

/\*\*

\* 删除单个对象

\*

\* @param key

\*/

public boolean deleteObject(final String key)

{

return redisTemplate.delete(key);

}

/\*\*

\* 删除集合对象

\*

\* @param collection 多个对象

\* @return

\*/

public long deleteObject(final Collection collection)

{

return redisTemplate.delete(collection);

}

/\*\*

\* 缓存List数据

\*

\* @param key 缓存的键值

\* @param dataList 待缓存的List数据

\* @return 缓存的对象

\*/

public <T> long setCacheList(final String key, final List<T> dataList)

{

Long count = redisTemplate.opsForList().rightPushAll(key, dataList);

return count == null ? 0 : count;

}

/\*\*

\* 获得缓存的list对象

\*

\* @param key 缓存的键值

\* @return 缓存键值对应的数据

\*/

public <T> List<T> getCacheList(final String key)

{

return redisTemplate.opsForList().range(key, 0, -1);

}

/\*\*

\* 缓存Set

\*

\* @param key 缓存键值

\* @param dataSet 缓存的数据

\* @return 缓存数据的对象

\*/

public <T> BoundSetOperations<String, T> setCacheSet(final String key, final Set<T> dataSet)

{

BoundSetOperations<String, T> setOperation = redisTemplate.boundSetOps(key);

Iterator<T> it = dataSet.iterator();

while (it.hasNext())

{

setOperation.add(it.next());

}

return setOperation;

}

/\*\*

\* 获得缓存的set

\*

\* @param key

\* @return

\*/

public <T> Set<T> getCacheSet(final String key)

{

return redisTemplate.opsForSet().members(key);

}

/\*\*

\* 缓存Map

\*

\* @param key

\* @param dataMap

\*/

public <T> void setCacheMap(final String key, final Map<String, T> dataMap)

{

if (dataMap != null) {

redisTemplate.opsForHash().putAll(key, dataMap);

}

}

/\*\*

\* 获得缓存的Map

\*

\* @param key

\* @return

\*/

public <T> Map<String, T> getCacheMap(final String key)

{

return redisTemplate.opsForHash().entries(key);

}

/\*\*

\* 往Hash中存入数据

\*

\* @param key Redis键

\* @param hKey Hash键

\* @param value 值

\*/

public <T> void setCacheMapValue(final String key, final String hKey, final T value)

{

redisTemplate.opsForHash().put(key, hKey, value);

}

/\*\*

\* 获取Hash中的数据

\*

\* @param key Redis键

\* @param hKey Hash键

\* @return Hash中的对象

\*/

public <T> T getCacheMapValue(final String key, final String hKey)

{

HashOperations<String, String, T> opsForHash = redisTemplate.opsForHash();

return opsForHash.get(key, hKey);

}

/\*\*

\* 删除Hash中的数据

\*

\* @param key

\* @param hkey

\*/

public void delCacheMapValue(final String key, final String hkey)

{

HashOperations hashOperations = redisTemplate.opsForHash();

hashOperations.delete(key, hkey);

}

/\*\*

\* 获取多个Hash中的数据

\*

\* @param key Redis键

\* @param hKeys Hash键集合

\* @return Hash对象集合

\*/

public <T> List<T> getMultiCacheMapValue(final String key, final Collection<Object> hKeys)

{

return redisTemplate.opsForHash().multiGet(key, hKeys);

}

/\*\*

\* 获得缓存的基本对象列表

\*

\* @param pattern 字符串前缀

\* @return 对象列表

\*/

public Collection<String> keys(final String pattern)

{

return redisTemplate.keys(pattern);

}

}

**JWT的工具类**

package zou.utils;

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();

}

public static void main(String[] args) throws Exception {

String token = "eyJhbGciOiJIUzI1NiJ9.eyJqdGkiOiJjYWM2ZDVhZi1mNjVlLTQ0MDAtYjcxMi0zYWEwOGIyOTIwYjQiLCJzdWIiOiJzZyIsImlzcyI6InNnIiwiaWF0IjoxNjM4MTA2NzEyLCJleHAiOjE2MzgxMTAzMTJ9.JVsSbkP94wuczb4QryQbAke3ysBDIL5ou8fWsbt\_ebg";

Claims claims = parseJWT(token);

System.out.println(claims);

}

/\*\*

\* 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 class FastJsonRedisSerializer<T> implements RedisSerializer<T> {

public static final Charset DEFAULT\_CHARSET = Charset.forName("UTF-8");

public Class<T> clazz;

static {

ParserConfig.getGlobalInstance().setAutoTypeSupport(true);

}

public FastJsonRedisSerializer(Class<T> clazz) {

super();

this.clazz = clazz;

}

@Override

public byte[] serialize(Object o) throws SerializationException {

if (o == null) {

return null;

}

return JSON.toJSONString(o, SerializerFeature.WriteClassName).getBytes(DEFAULT\_CHARSET);

}

@Override

public T deserialize(byte[] bytes) throws SerializationException {

if (bytes == null || bytes.length <= 0) {

return null;

}

String str = new String(bytes, DEFAULT\_CHARSET);

return JSON.parseObject(str, clazz);

}

public JavaType getJavaType(Class<?> clazz) {

return TypeFactory.defaultInstance().constructType(clazz);

}

}

**WebUtils**

public class WebUtils

{

/\*\*

\* 将字符串渲染到客户端

\*

\* @param response 渲染对象

\* @param string 待渲染的字符串

\* @return null

\*/

public static String renderString(HttpServletResponse response, String string) {

try

{

response.setStatus(200);

response.setContentType("application/json");

response.setCharacterEncoding("utf-8");

response.getWriter().print(string);

}

catch (IOException e)

{

e.printStackTrace();

}

return null;

}

}

**2.4 搭建SpringSecurity的测试代码**

**controller层**

@RestController

public class HelloController {

@RequestMapping("/hello")

public String hello(){

return "hello";

}

}

**service层**

在service层实现UserdetailsService接口实现用户的数据库认证

@Service

public class UserDetailsServiceImpl implements UserDetailsService {

@Autowired

private UserMapper userMapper;

@Override

public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

//查询用户信息

LambdaQueryWrapper<User> lambdaQueryWrapper =new LambdaQueryWrapper<>();

lambdaQueryWrapper.eq(User::getUserName,username);

User user = userMapper.selectOne(lambdaQueryWrapper);

//如果没有查询到用户就抛出异常

if(Objects.isNull(user)){

throw new RuntimeException("没有查询到用户");

}

// TODO 查询相应的权限信息

//封装成UserDetail封装

return new LoginUser(user);

}

}

**mapper接口**

@Mapper

public interface UserMapper extends BaseMapper<User> {

}

**bean层**

实现UserDetail实体类并UserDetailService返回给框架验证

/\*\*

\* @Author zouzilu

\*/

@Data

@NoArgsConstructor

@AllArgsConstructor

public class LoginUser implements UserDetails {

private User user;

@Override

public Collection<? extends GrantedAuthority> getAuthorities() {

return null;

}

@Override

public String getPassword() {

return user.getPassword();

}

@Override

public String getUsername() {

return user.getUserName();

}

@Override

public boolean isAccountNonExpired() {

return true;

}

@Override

public boolean isAccountNonLocked() {

return true;

}

@Override

public boolean isCredentialsNonExpired() {

return true;

}

@Override

public boolean isEnabled() {

return true;

}

}

返回的结果响应的resultBean

/\*\*

\* @Author zou

\*/

@JsonInclude(JsonInclude.Include.NON\_NULL)

public class ResponseResult<T> {

/\*\*

\* 状态码

\*/

private Integer code;

/\*\*

\* 提示信息，如果有错误时，前端可以获取该字段进行提示

\*/

private String msg;

/\*\*

\* 查询到的结果数据，

\*/

private T data;

public ResponseResult(Integer code, String msg) {

this.code = code;

this.msg = msg;

}

public ResponseResult(Integer code, T data) {

this.code = code;

this.data = data;

}

public Integer getCode() {

return code;

}

public void setCode(Integer code) {

this.code = code;

}

public String getMsg() {

return msg;

}

public void setMsg(String msg) {

this.msg = msg;

}

public T getData() {

return data;

}

public void setData(T data) {

this.data = data;

}

public ResponseResult(Integer code, String msg, T data) {

this.code = code;

this.msg = msg;

this.data = data;

}

}

User实体类

/\*\*

\* 用户表(User)实体类

\*

\* @author zou

\*/

@Data

@AllArgsConstructor

@NoArgsConstructor

@TableName("sys\_user")

public class User implements Serializable {

private static final long serialVersionUID = -40356785423868312L;

/\*\*

\* 主键

\*/

@TableId

private Long id;

/\*\*

\* 用户名

\*/

private String userName;

/\*\*

\* 昵称

\*/

private String nickName;

/\*\*

\* 密码

\*/

private String password;

/\*\*

\* 账号状态（0正常 1停用）

\*/

private String status;

/\*\*

\* 邮箱

\*/

private String email;

/\*\*

\* 手机号

\*/

private String phonenumber;

/\*\*

\* 用户性别（0男，1女，2未知）

\*/

private String sex;

/\*\*

\* 头像

\*/

private String avatar;

/\*\*

\* 用户类型（0管理员，1普通用户）

\*/

private String userType;

/\*\*

\* 创建人的用户id

\*/

private Long createBy;

/\*\*

\* 创建时间

\*/

private Date createTime;

/\*\*

\* 更新人

\*/

private Long updateBy;

/\*\*

\* 更新时间

\*/

private Date updateTime;

/\*\*

\* 删除标志（0代表未删除，1代表已删除）

\*/

private Integer delFlag;

}

**2.5密码加密**

密码在SpringSecurity中一般是加密在数据库中存储的，如果你需要明文处理需要在数据库中加上{noop}+密码的方式存储

package zou.config;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;

import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

import org.springframework.security.crypto.password.PasswordEncoder;

/\*\*

\* @Author zou

\*/

@Configuration

public class SecurityConfig extends WebSecurityConfigurerAdapter {

@Bean

public PasswordEncoder passwordEncoder(){

return new BCryptPasswordEncoder();

}

}

**2.6 JWT的加密与jwtUtils的使用**

/

\*

\* 加密与解密

\*/

public static void main(String[] args) throws Exception {

String jwt = createJWT("1234");

System.out.println(jwt);

Claims claims=parseJWT(jwt);

String subject = claims.getSubject();

System.out.println(subject);

}

**2.7 登录接口的放行**

**登录接口的创建**

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.web.bind.annotation.PostMapping;

import org.springframework.web.bind.annotation.RequestBody;

import org.springframework.web.bind.annotation.RestController;

import zou.bean.ResponseResult;

import zou.bean.User;

import zou.service.LoginService;

@RestController

public class LoginController {

@Autowired

private LoginService loginService;

@PostMapping("/user/login")

public ResponseResult login(@RequestBody User user){

System.out.println("进入登录");

//登录

ResponseResult result = loginService.login(user);

return result;

}

}

**登录接口的service层实现**

注意在SpringSecurity的配置类中将AuthenticationManager加入容器中并暴露

/\*\*

\* @Author zou

\* 密码加密存储

\*/

@Configuration

public class SecurityConfig extends WebSecurityConfigurerAdapter {

/\*\*

\* 默认加密方式

\* @return

\*/

@Bean

public PasswordEncoder passwordEncoder(){

return new BCryptPasswordEncoder();

}

/\*\*

\* 相关的配置

\* @param http

\* @throws Exception

\*/

@Override

protected void configure(HttpSecurity http) throws Exception {

http

//关闭csrf

.csrf().disable()

//不通过Session获取SecurityContext

.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS)

.and()

.authorizeRequests()

// 对于登录接口 允许匿名访问

.antMatchers("/user/login").anonymous()

// 除上面外的所有请求全部需要鉴权认证

.anyRequest().authenticated();

}

/\*\*

\*

\* @return

\* @throws Exception

\*/

@Bean

@Override

public AuthenticationManager authenticationManagerBean() throws Exception {

return super.authenticationManagerBean();

}

}

public interface LoginService {

ResponseResult login(User user);

}

@Service

public class LoginServiceImpl implements LoginService {

@Autowired

private AuthenticationManager authenticationManager;

@Autowired

private RedisCache redisCache;

@Override

public ResponseResult login(User user) {

//AuthenticationManager中进行用户认证

UsernamePasswordAuthenticationToken authenticationToken =new UsernamePasswordAuthenticationToken(user.getUserName(),user.getPassword());

Authentication authenticate = authenticationManager.authenticate(authenticationToken);

//如果认证没有通过，使用userid生成一个jwt

if(Objects.isNull(authenticate)){

throw new RuntimeException("登录失败");

}

//如果认证通过了将用户信息存入redis把userid作为key

LoginUser loginUser = (LoginUser)authenticate.getPrincipal();

String user\_id = loginUser.getUser().getId().toString();

String jwt = JwtUtil.createJWT(user\_id);

Map<String,String> map =new HashMap<>();

map.put("token",jwt);

//保存到redis中

redisCache.setCacheObject("login"+user\_id,loginUser);

return new ResponseResult(200,"登陆成功",map);

}

}

**2.8 Token的认证过滤器**

**2.8.1 JwtAuthenticationTokenFilter的编写**

获取token

解析token获取其中的userId

从redis中获取用户信息

存入SecurityContextHolder中

/\*\*

\* jwt的过滤器

\* 并且将其注入容器

\*/

@Component

public class JwtAuthenticationTokenFilter extends OncePerRequestFilter {

@Autowired

private RedisCache redisCache;

@Override

protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain) throws ServletException, IOException {

//获取token

String token = request.getHeader("token");

String userId;

if(!StringUtils.hasText(token)){

//如果不存在token放行

filterChain.doFilter(request,response);

return;

}

//解析token

try {

Claims claims = JwtUtil.parseJWT(token);

userId = claims.getSubject();

} catch (Exception e) {

e.printStackTrace();

throw new RuntimeException("token非法");

}

//从redis中获取用户信息

String redisKey = "login:"+userId;

LoginUser loginUser = redisCache.getCacheObject(redisKey);

//判断loginUser是否存在

if(Objects.isNull(loginUser)){

throw new RuntimeException("用户未登录！");

}

//存入SecurityContextHolder

// TODO 获取权限信息封装到AuthenticationToken中

UsernamePasswordAuthenticationToken AuthenticationToken=new UsernamePasswordAuthenticationToken(loginUser,null,null);

SecurityContextHolder.getContext().setAuthentication(AuthenticationToken);

}

}

**2.8.2配置认证过滤器**

在SecurityConfig.java中配置编写

package zou.config;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.security.authentication.AuthenticationManager;

import org.springframework.security.config.annotation.web.builders.HttpSecurity;

import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;

import org.springframework.security.config.http.SessionCreationPolicy;

import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

import org.springframework.security.crypto.password.PasswordEncoder;

import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;

import zou.filter.JwtAuthenticationTokenFilter;

/\*\*

\* @Author zou

\* 密码加密存储

\*/

@Configuration

public class SecurityConfig extends WebSecurityConfigurerAdapter {

@Autowired

private JwtAuthenticationTokenFilter jwtAuthenticationTokenFilter;

/\*\*

\* 默认加密方式

\*

\* @return

\*/

@Bean

public PasswordEncoder passwordEncoder() {

return new BCryptPasswordEncoder();

}

/\*\*

\* 相关的配置

\*

\* @param http

\* @throws Exception

\*/

@Override

protected void configure(HttpSecurity http) throws Exception {

http

//关闭csrf

.csrf().disable()

//不通过Session获取SecurityContext

.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS)

.and()

.authorizeRequests()

// 对于登录接口 允许匿名访问

.antMatchers("/user/login").anonymous()

// 除上面外的所有请求全部需要鉴权认证

.anyRequest().authenticated();

// 将某个过滤器添加到某个过滤器之前

http.addFilterBefore(jwtAuthenticationTokenFilter, UsernamePasswordAuthenticationFilter.class);

}

/\*\*

\* @return

\* @throws Exception

\*/

@Bean

@Override

public AuthenticationManager authenticationManagerBean() throws Exception {

return super.authenticationManagerBean();

}

}

**2.9 退出登录的状态**

我们需要定义一个接口，然后获取SecurityContextHolder中的认证信息，删除redis中对应的数据即可。

**3. 授权**

**3.1 授权的基本流程**

把登录的权限信息存入到Authentication中

我们设置我们的资源所需要的权限控制方案

**3.2 权限实现**

**3.2.1 基于注解的实现方案**

// 开启相关的配置

@EnableClobalMethodSecurity(prePostEnabled = true)