

# 一、概述

---



**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 Spring Security 来做安全框架。
- 小项目用 Shiro 的比较多，因为相比与 Spring Security，Shiro 的上手更加的简单。

大部分 Web 应用都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

而认证和授权也是 Spring Security 作为安全框架的核心功能。

## 二、快速入门

---

### 2.1 准备工作

---

1、搭建一个简单的 Spring Boot 工程，设置父工程、添加依赖

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.7.18</version>
</parent>

<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>

  <dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <optional>>true</optional>
  </dependency>
</dependencies>
```

2、创建启动类

```
@SpringBootApplication
public class SpringBootSecurityApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringBootSecurityApplication.class, args);
    }
}
```

### 3、创建Controller

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

@RestController
public class HelloController {

    @RequestMapping("/hello")
    public String hello() {
        return "hello";
    }
}
```


## 2.2 引入SpringSecurity

在SpringBoot项目中使用SpringSecurity我们只需要引入依赖即可实现入门案例。

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

引入依赖后我们在尝试去访问之前的接口就会自动跳转到一个SpringSecurity的默认登录页面，默认用户名是user，密码会输出在控制台。

必须登录之后才能对接口进行访问。

A screenshot of the default Spring Security login page. It has a light gray background and the title "Please sign in" in bold black text. Below the title, there are two input fields: "Username" and "Password". The "Username" field has "Username" as a placeholder, and the "Password" field has "Password" as a placeholder. Below these fields is a "Sign in" button.

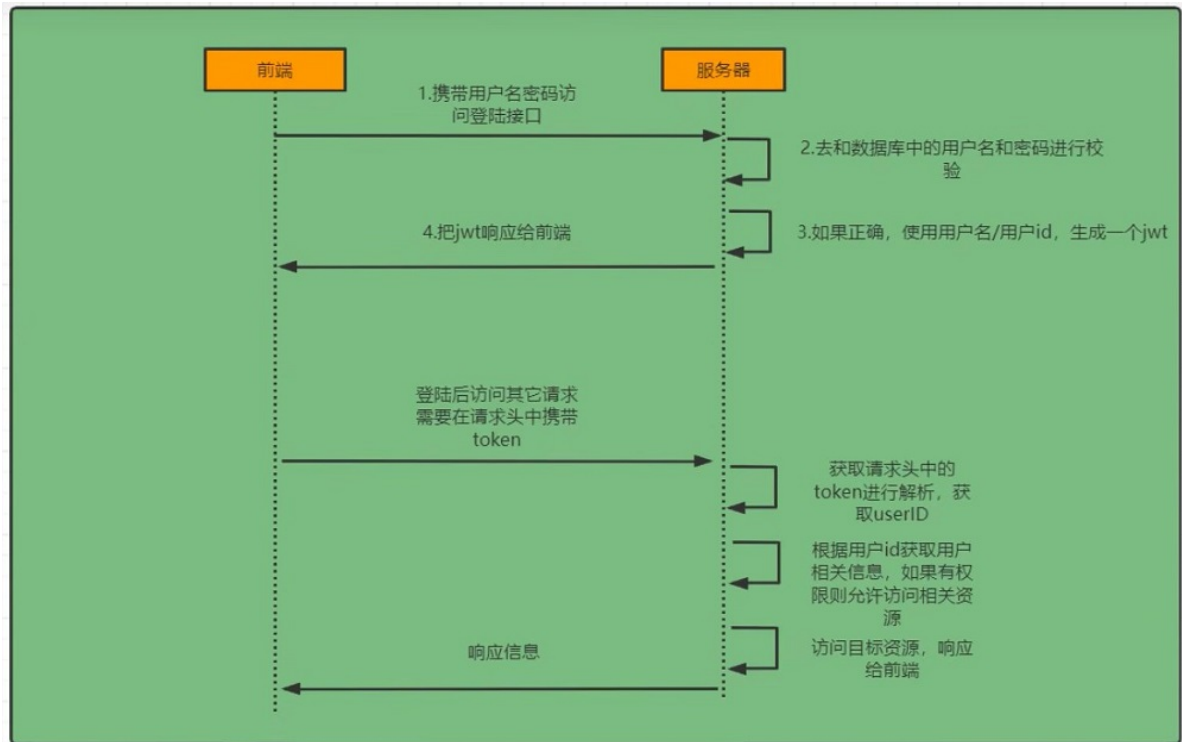
## 三、认证

### 3.1 登录校验流程

前后端分离场景下，登录校验流程的核心思路：**token**。

- Token是服务端生成的一串字符串，以作客户端进行请求的一个令牌，当第一次登录后，服务器生成一个Token便将此Token返回给客户端，以后客户端只需带上这个Token前来请求数据即可，无需再次带上用户名和密码。

- 使用token机制的身份验证方法，在服务器端不需要存储用户的登录记录。
- JWT: <https://www.jianshu.com/p/d1644e281250>

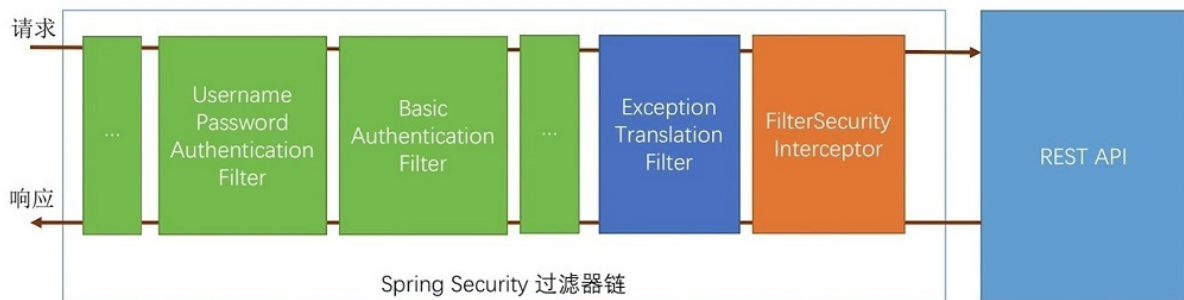


## 3.2 原理初探

想要知道如何实现自己的登录流程就必须先知道入门案例中SpringSecurity的流程。

### SpringSecurity完整流程

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`: 负责处理我们在登录页面填写了用户名密码后的登录请求。入门案例的认证工作主要由它负责。
- `ExceptionTranslationFilter`: 处理过滤器链中抛出的任何`AccessDeniedException`和`AuthenticationException`。
  - `AccessDeniedException`: 拒绝访问异常，没有权限；
  - `AuthenticationException`: 认证异常，用户名或密码错误；
- `FilterSecurityInterceptor`: 负责权限校验的过滤器。

我们可以通过Debug查看当前系统中SpringSecurity过滤器链中有哪些过滤器及它们的顺序。

```

SpringBootApplication.java x
17 ConfigurableApplicationContext run = SpringApplication.run(SpringBootApplication.class, args);
18 // 获取默认的Security过滤器链
19 DefaultSecurityFilterChain chain = run.getBean(DefaultSecurityFilterChain.class); run: "org.springframework
20 System.out.println(chain); chain: "DefaultSecurityFilterChain [RequestMatcher=any request, Filters=[c
21 }
22 }

```

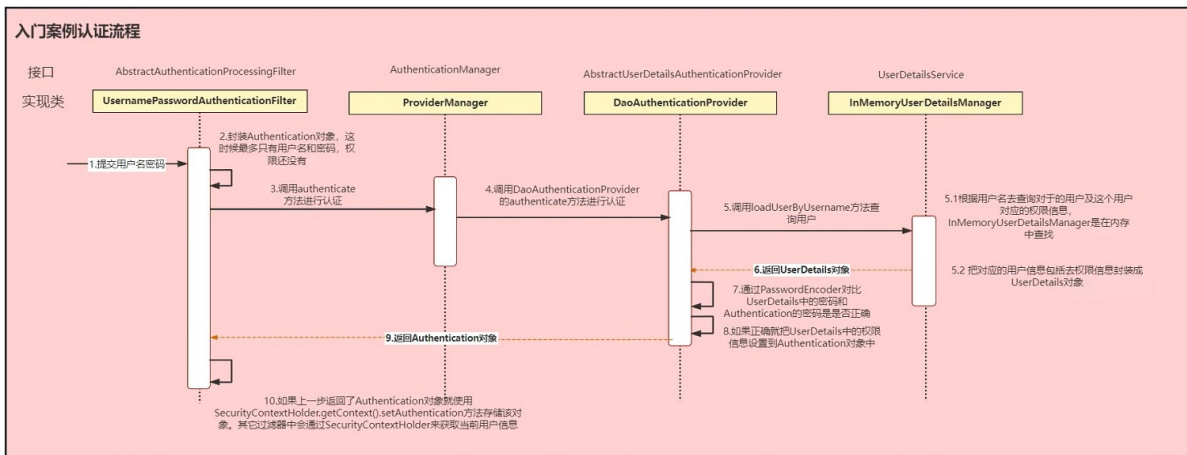
Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)

```

> f requestMatcher = (AnyRequestMatcher@6856) "any request"
v f filters = (ArrayList@6857) size = 16
> 0 = (DisableEncodeUrlFilter@6860)
> 1 = (WebAsyncManagerIntegrationFilter@6861)
> 2 = (SecurityContextPersistenceFilter@6862)
> 3 = (HeaderWriterFilter@6863)
> 4 = (CsrfFilter@6864)
> 5 = (LogoutFilter@6865)
> 6 = (UsernamePasswordAuthenticationFilter@6866)
> 7 = (DefaultLoginPageGeneratingFilter@6867)
> 8 = (DefaultLogoutPageGeneratingFilter@6868)
> 9 = (BasicAuthenticationFilter@6869)
> 10 = (RequestCacheAwareFilter@6870)
> 11 = (SecurityContextHolderAwareRequestFilter@6871)
> 12 = (AnonymousAuthenticationFilter@6872)
> 13 = (SessionManagementFilter@6873)
> 14 = (ExceptionTranslationFilter@6874)
> 15 = (FilterSecurityInterceptor@6875)

```

## 入门案例认证流程详解

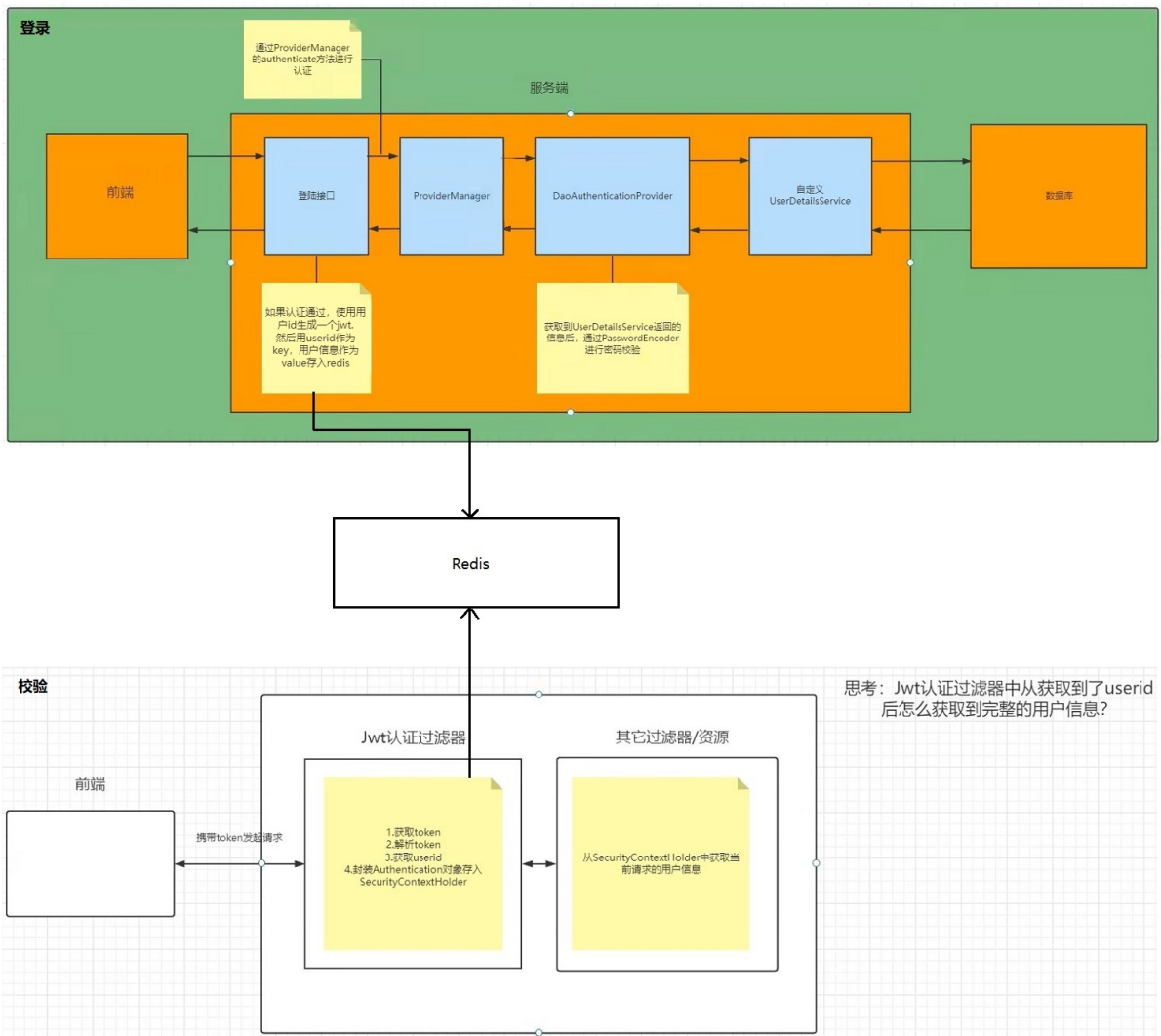


### 核心类:

- **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根据用户名获取用户信息封装成UserDetails对象返回。然后将这些信息封装到Authentication对象中。

## 3.3 解决问题

# 思路分析



## 登录

### 1. 自定义登录接口

- 调用ProviderManager的方法进行认证，如果认证通过生成jwt
- 把用户信息（登录用户信息和权限信息）存入redis中

### 2. 自定义类实现UserDetailsService接口

- 重写loadUserByUsername(), 在这个方法中去查询数据库，获取用户信息和用户权限信息

## 校验

### 定义Jwt认证过滤器

- 获取token
- 解析token获取其中的userid
- 从redis中获取用户信息
- 存入SecurityContextHolder

## 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>
```

## 添加Redis相关配置

```
import com.alibaba.fastjson.JSON;
import com.alibaba.fastjson.serializer.SerializerFeature;
import com.fasterxml.jackson.databind.JavaType;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.type.TypeFactory;
import org.springframework.data.redis.serializer.RedisSerializer;
import org.springframework.data.redis.serializer.SerializationException;
import com.alibaba.fastjson.parser.ParserConfig;
import org.springframework.util.Assert;

import java.nio.charset.Charset;

/**
 * Redis使用FastJson序列化
 */
public class FastJsonRedisSerializer<T> implements RedisSerializer<T> {

    public static final Charset DEFAULT_CHARSET = Charset.forName("UTF-8");

    private Class<T> clazz;

    static {
        ParserConfig.getGlobalInstance().setAutoTypeSupport(true);
    }

    public FastJsonRedisSerializer(Class<T> clazz) {
        super();
        this.clazz = clazz;
    }
}
```

```

@Override
public byte[] serialize(T t) throws SerializationException {
    if (t == null) {
        return new byte[0];
    }
    return JSON.toJSONString(t,
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);
}

protected JavaType getJavaType(Class<?> clazz) {
    return TypeFactory.defaultInstance().constructType(clazz);
}
}

```

```

import com.hl.boot.utils.FastJsonRedisSerializer;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.redis.connection.RedisConnectionFactory;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.data.redis.serializer.StringRedisSerializer;

@Configuration
public class RedisConfig {

    @Bean
    @SuppressWarnings(value = {"unchecked", "rawtypes"})
    public RedisTemplate<Object, Object> redisTemplate(RedisConnectionFactory
connectionFactory) {
        RedisTemplate<Object, Object> template = new RedisTemplate<>();
        template.setConnectionFactory(connectionFactory);

        FastJsonRedisSerializer serializer = new
FastJsonRedisSerializer(Object.class);

        // 使用StringRedisSerializer来序列化和反序列化redis的key值
        template.setKeySerializer(new StringRedisSerializer());
        template.setValueSerializer(serializer);

        // Hash的key也采用StringRedisSerializer的序列化方式
        template.setHashKeySerializer(new StringRedisSerializer());
        template.setHashValueSerializer(serializer);

        template.afterPropertiesSet();
        return template;
    }
}

```

```
}  
}
```

## 响应类

```
import com.fasterxml.jackson.annotation.JsonInclude;  
  
/**  
 * 响应结果类  
 */  
@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;
}
}

```

## 工具类

### JWT工具类:

```

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 = "ceshikey";

    public static String getUUID() {
        String token = UUID.randomUUID().toString().replaceAll("-", "");
        return token;
    }

    /**
     * 生成jwt
     *
     * @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());
        return builder.compact();
    }

    /**
     * 生成jwt
     *

```

```

    * @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 {
        // 创建token
        String jwt = createJWT("1001");
        System.out.println(jwt);

        // 解析token
        Claims claims = parseJWT(jwt);
        System.out.println(claims.getSubject());
    }

/**

```

```

    * 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();
    }
}

```

## RedisCache:

```

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.redis.core.BoundSetOperations;
import org.springframework.data.redis.core.HashOperations;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.data.redis.core.ValueOperations;
import org.springframework.stereotype.Component;

import java.util.*;
import java.util.concurrent.TimeUnit;

@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);
}
}

```

## WebUtils:

```

import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

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;
}
}
```

## 实体类

```
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import java.io.Serializable;
import java.util.Date;

/**
 * 用户表(User)实体类
 */
@Data
@AllArgsConstructor
@NoArgsConstructor
public class User implements Serializable {
    private static final long serialVersionUID = -40356785423868312L;

    /**
     * 主键
     */
    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;
}

```

## 认证实现

### 数据库校验用户

从之前的分析我们可以知道，我们可以自定义一个UserDetailsService的实现类，让SpringSecurity使用我们的UserDetailsService。我们自己的UserDetailsService可以从数据库中查询用户名和密码。

#### 准备工作

我们先创建一个用户表，建表语句如下：

```

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'
```

引入MybatisPuls和mysql驱动的依赖:

```
<dependency>
  <groupId>com.baomidou</groupId>
  <artifactId>mybatis-plus-boot-starter</artifactId>
  <version>3.5.2</version>
</dependency>

<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>8.0.20</version>
</dependency>
```

配置数据库信息:

```
spring:
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://localhost:3306/security?characterEncoding=utf-8&serverTimezone=UTC
    username: root
    password: root
```

定义Mapper接口:

```
public interface UserMapper extends BaseMapper<User> {
}
```

修改User实体类:

```
类名上加@TableName(value = "sys_user"), id字段上加 @TableId
```

配置Mapper扫描:

```

@SpringBootApplication
@MapperScan("com.gs.mapper")
public class SpringBootSecurityApplication {

    public static void main(String[] args) {
        ConfigurableApplicationContext run =
SpringApplication.run(SpringBootSecurityApplication.class, args);
        // 获取默认的Security过滤器链
        DefaultSecurityFilterChain chain =
run.getBean(DefaultSecurityFilterChain.class);
        //System.out.println(chain);
    }
}

```

添加junit依赖:

```

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
</dependency>

```

测试MP是否能正常使用:

```

@SpringBootTest
public class MapperTest {

    @Autowired
    private UserMapper userMapper;

    @Test
    public void testUserMapper(){
        List<User> users = userMapper.selectList(null);
        System.out.println(users);
    }
}

```

## 核心代码实现

创建一个类实现UserDetailsService接口，重写其中的方法。通过用户名从数据库中查询用户信息

```

@Service
public class UserDetailsServiceImpl implements UserDetailsService {

    @Autowired
    private UserMapper userMapper;

    @Override
    public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
        //根据用户名查询用户信息：注意使用我们自己定义的用户实体类，别导错包
        LambdaQueryWrapper<User> wrapper = new LambdaQueryWrapper<>();
        wrapper.eq(User::getUserName, username);
        User user = userMapper.selectOne(wrapper);
    }
}

```

```

//如果查询不到数据就通过抛出异常来给出提示
if (Objects.isNull(user)) {
    throw new RuntimeException("用户名错误");
}

//TODO 根据用户查询权限信息 添加到LoginUser中

//封装成UserDetails对象返回
return new LoginUser(user);
}
}

```

因为UserDetailsService方法的返回值是UserDetails类型，所以需要定义一个类，实现该接口，把用户信息封装在其中。

```

@Data
@NoArgsConstructor
@AllArgsConstructor
public class LoginUser implements UserDetails {

    // 用以封装用户信息
    private User user;

    // 用户的权限集
    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return null;
    }

    // 用户的加密后的密码，不加密会使用{noop}前缀
    @Override
    public String getPassword() {
        return user.getPassword();
    }

    // 用户名
    @Override
    public String getUsername() {
        return user.getUserName();
    }

    // 帐户未过期
    @Override
    public boolean isAccountNonExpired() {
        return true;
    }

    // 帐户未锁定
    @Override
    public boolean isAccountNonLocked() {
        return true;
    }

    // 凭证是否过期
    @Override

```

```

public boolean isCredentialsNonExpired() {
    return true;
}

// 用户是否可用
@Override
public boolean isEnabled() {
    return true;
}
}

```

注意：如果要测试，需要往用户表中写入用户数据，并且如果你想让用户的密码是明文存储，需要在密码前加{noop}。例如

id	user_name	nick_name	password	status	email	phonenumber	sex	avatar	user_type	create_by	create_time
1	admin	管理员	{noop}1234	0	admin@qq.110		0	(Null)	1		(Null) (Null)

这样登录的时候就可以用admin作为用户名，1234作为密码来登录了。

## 密码加密存储

实际项目中我们不会把密码明文存储在数据库中。

- 默认使用的PasswordEncoder要求数据库中的密码格式为：{id}password。它会根据id去判断密码的加密方式。但是我们一般不会采用这种方式。所以需要替换PasswordEncoder。
- 我们一般使用SpringSecurity为我们提供的BCryptPasswordEncoder。
- 我们只需要使用把BCryptPasswordEncoder对象注入Spring容器中，SpringSecurity就会使用该PasswordEncoder来进行密码校验。
- 我们可以定义一个SpringSecurity的配置类，SpringSecurity要求这个配置类要继承WebSecurityConfigurerAdapter。

```

@Configuration
public class SecurityConfig extends webSecurityConfigurerAdapter {

    // 创建BCryptPasswordEncoder，并注入容器
    @Bean
    public PasswordEncoder passwordEncoder(){
        return new BCryptPasswordEncoder();
    }

}

```

测试：

```

@Autowired
private PasswordEncoder passwordEncoder;

@Test
public void testBCryptPasswordEncoder(){
    // passwordEncoder.encode(String password) 对密码进行加密
    String encode = passwordEncoder.encode("1234");
    System.out.println(encode);

    // passwordEncoder.matches(String password, String encodedPassword) 使用明文密码和加密后的密码进行比对
    boolean b = passwordEncoder.matches("1234", encode);
    System.out.println(b);
}

```

数据库存储的是加密后的密码;

## 登录接口

接下来我们需要自定义登录接口，然后让SpringSecurity对这个接口放行，让用户访问这个接口的时候不用登录也能访问。

在接口中我们通过AuthenticationManager的authenticate方法来进行用户认证，所以需要在SecurityConfig中配置把AuthenticationManager注入容器。

认证成功的话要生成一个jwt，放入响应中返回。并且为了让用户下回请求时能通过jwt识别出具体的哪个用户，我们需要把用户信息存入redis，可以把用户id作为key。

```

@RestController
public class LoginController {

    @Autowired
    private LoginService loginService;

    @PostMapping("/user/login")
    public ResponseEntity login(@RequestBody User user){
        return loginService.login(user);
    }
}

```

```

@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Bean
    public PasswordEncoder passwordEncoder(){
        return new BCryptPasswordEncoder();
    }

    @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();
}

@Bean
@Override
public AuthenticationManager authenticationManagerBean() throws Exception {
    return super.authenticationManagerBean();
}
}

```

```

@Service
public class LoginServiceImpl implements LoginService {

    @Autowired
    private AuthenticationManager authenticationManager;

    @Autowired
    private RedisCache redisCache;

    @Override
    public ResponseResult login(User user) {
        // 调用AuthenticationManager的authenticate进行身份认证
        UsernamePasswordAuthenticationToken authenticationToken = new
        UsernamePasswordAuthenticationToken(user.getUserName(),user.getPassword());
        Authentication authenticate =
        authenticationManager.authenticate(authenticationToken);

        // 如果认证没有通过，抛出异常
        if(Objects.isNull(authenticate)){
            throw new RuntimeException("用户名或密码错误");
        }

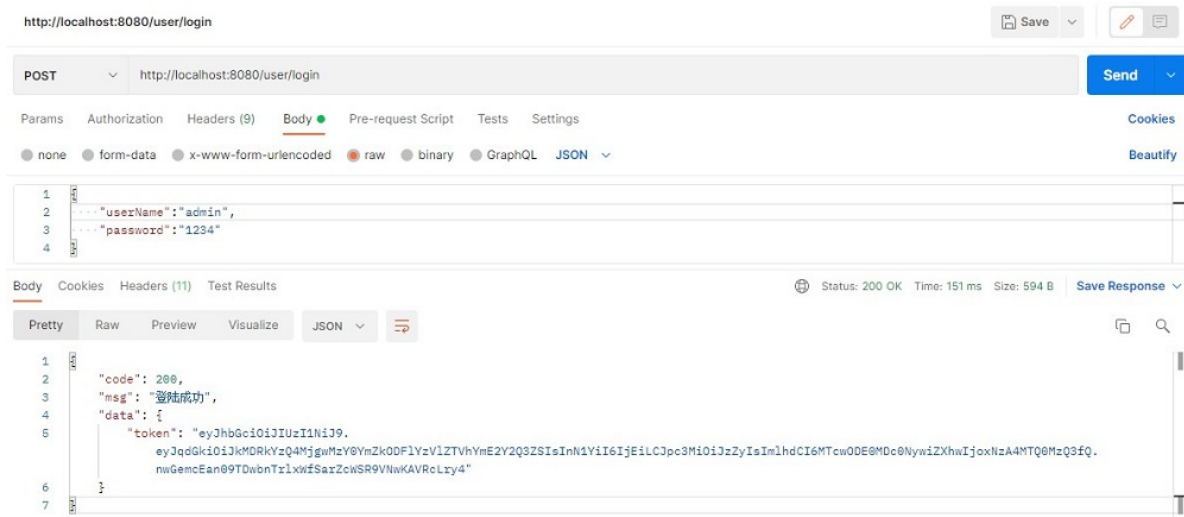
        // 认证通过，使用userid生成token
        LoginUser loginUser = (LoginUser) authenticate.getPrincipal();
        String userId = loginUser.getUser().getId().toString();
        String jwt = JwtUtil.createJWT(userId);

        // 把完整的用户信息存入redis，userid作为key【为避免重复，可以添加一些前缀】
        redisCache.setCacheObject("login:"+userId,loginUser);

        //把token响应给前端
        HashMap<String,String> map = new HashMap<>();
        map.put("token",jwt);
        return new ResponseResult(200,"登录成功",map);
    }
}
}

```

测试登录:



**注意：记得开启Redis**

## 认证过滤器

我们需要自定义一个过滤器，这个过滤器会去获取请求头中的token，对token进行解析取出其中的userid。

使用userid去redis中获取对应的LoginUser对象。

然后封装Authentication对象存入SecurityContextHolder。

### 定义Jwt认证过滤器：

```
@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");
        if (!StringUtils.hasText(token)) {
            // 放行
            filterChain.doFilter(request, response);
            return; // 没有token，无需后续的token解析
        }

        // 解析token
        String userid;
        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);
if (Objects.isNull(loginUser)) {
    throw new RuntimeException("登录过期,请重新登录");
}

//存入SecurityContextHolder
UsernamePasswordAuthenticationToken authenticationToken =
    new UsernamePasswordAuthenticationToken(loginUser, null,
loginUser.getAuthorities());

SecurityContextHolder.getContext().setAuthentication(authenticationToken);
//放行
filterChain.doFilter(request, response);
}
}

```

### 配置Jwt认证过滤器:

```

@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Bean
    public PasswordEncoder passwordEncoder(){
        return new BCryptPasswordEncoder();
    }

    @Autowired
    JwtAuthenticationTokenFilter jwtAuthenticationTokenFilter;

    @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();

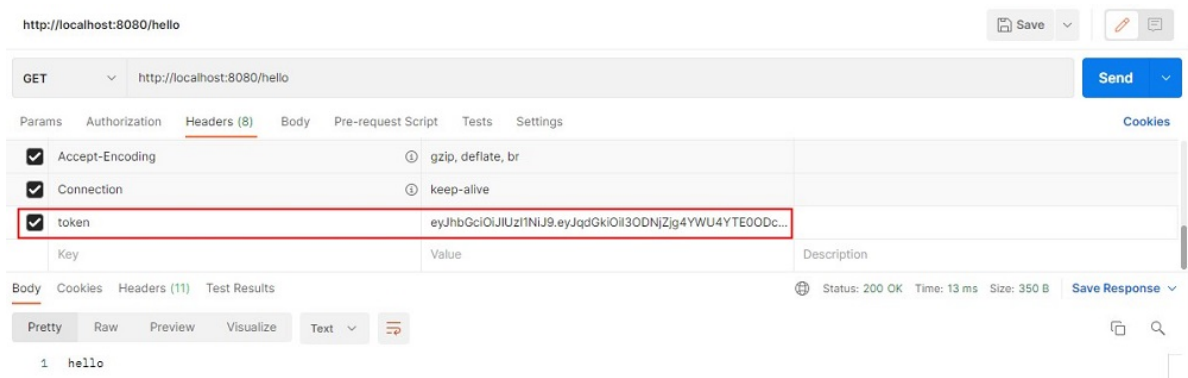
        //把token校验过滤器添加到过滤器链中
        http.addFilterBefore(jwtAuthenticationTokenFilter,
UsernamePasswordAuthenticationFilter.class);
    }

    @Bean
    @Override
    public AuthenticationManager authenticationManagerBean() throws Exception {
        return super.authenticationManagerBean();
    }
}

```

```
}
```

## 拷贝token访问资源:



http://localhost:8080/hello

GET http://localhost:8080/hello

Params Authorization Headers (8) Body Pre-request Script Tests Settings Cookies

Key	Value	Description
Accept-Encoding	gzip, deflate, br	
Connection	keep-alive	
token	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpzZW50L3NpdjE6Im54YU4YTE0ODc...	

Body Cookies Headers (11) Test Results Status: 200 OK Time: 13 ms Size: 350 B Save Response

Pretty Raw Preview Visualize Text

```
1 hello
```

## 退出登录

我们只需要定义一个退出登录接口，然后获取SecurityContextHolder中的认证信息，删除redis中对应的数据即可。

```
@RestController
public class LoginController {

    @Autowired
    private LoginService loginService;

    @PostMapping("/user/login")
    public ResponseEntity login(@RequestBody User user){
        return loginService.login(user);
    }

    // 退出登录
    @RequestMapping("/user/logout")
    public ResponseEntity logout(){
        return loginService.logout();
    }
}
```

```
@Service
public class LoginServiceImpl implements LoginService {

    @Autowired
    private AuthenticationManager authenticationManager;
    @Autowired
    private RedisCache redisCache;

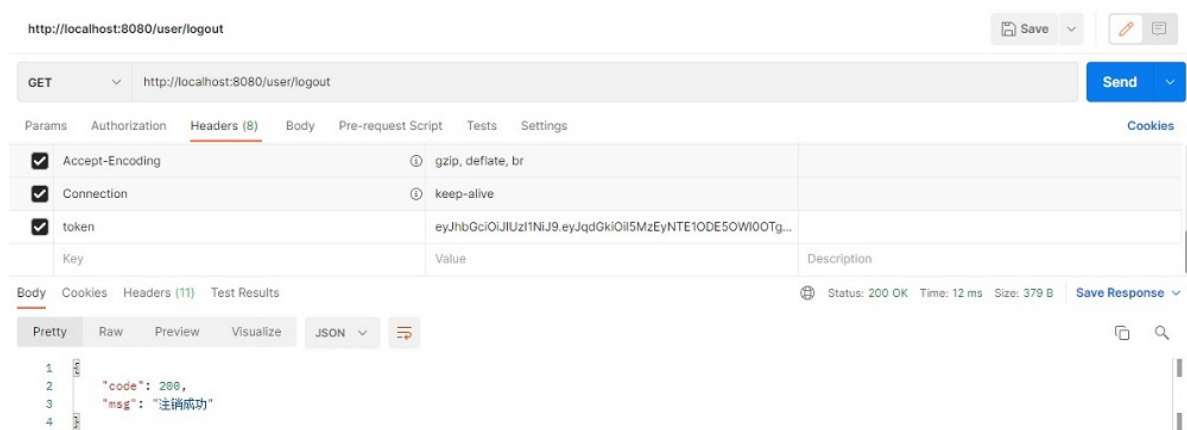
    @Override
    public ResponseEntity login(User user) {
        // 略
    }

    @Override
    public ResponseEntity logout() {
        // 获取SecurityContextHolder中的用户信息
    }
}
```

```
Authentication authentication =
SecurityContextHolder.getContext().getAuthentication();
LoginUser loginUser = (LoginUser) authentication.getPrincipal();
Long userid = loginUser.getUser().getId();

// 删除redis里面的值
redisCache.deleteObject("login:" + userid);
return new ResponseResult(200, "注销成功");
}
}
```

## 成功退出登录:



The screenshot shows a GET request to `http://localhost:8080/user/logout`. The response status is 200 OK. The response body is a JSON object: `{\"code\": 200, \"msg\": \"注销成功\"}`.

# 四、授权

## 4.1 权限系统的作用

例如一个学校图书馆的管理系统，如果是普通学生登录就能看到借书还书相关的功能，不可能让他看到并且去使用添加书籍信息，删除书籍信息等功能。但是如果是一个图书馆管理员的账号登录了，应该就能看到并使用添加书籍信息，删除书籍信息等功能。

总结起来就是**不同的用户可以使用不同的功能**。这就是权限系统要去实现的效果。

我们不能只依赖前端去判断用户的权限来选择显示哪些菜单哪些按钮。因为如果只是这样，如果有人知道了对应功能的接口地址就可以不通过前端，直接去发送请求来实现相关功能操作。

所以我们还需要在后台进行用户权限的判断，判断当前用户是否有相应的权限，必须具有所需权限才能进行相应的操作。

## 4.2 授权基本流程

在SpringSecurity中，会使用默认的FilterSecurityInterceptor来进行权限校验。在FilterSecurityInterceptor中会从SecurityContextHolder获取其中的Authentication，然后获取其中的权限信息。当前用户是否拥有访问当前资源所需的权限。

所以我们在项目中只需要把当前登录用户的权限信息也存入Authentication。

然后设置我们的资源所需要的权限即可。

## 4.3 授权实现

## 限制访问资源所需权限

SpringSecurity为我们提供了基于注解的权限控制方案，这也是我们项目中主要采用的方式。我们可以使用注解去指定访问对应的资源所需的权限。

但是要使用它我们需要先开启相关配置springSecurity里面加。

```
@Configuration
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    //...
}
```

然后就可以使用对应的注解：@PreAuthorize

- @PreAuthorize 是Spring Security框架中的一个注解，用于在方法调用之前对用户的权限进行验证。
- 允许在方法级别定义访问控制规则，确保只有满足指定条件的用户才能调用该方法。
- @PreAuthorize 注解的参数是一个 SpEL (Spring Expression Language) 表达式，用于定义权限规则 (SpEL支持在表达式中使用各种功能，包括方法调用、条件判断等)。
- 表达式的结果应该是布尔值，如果为 true，则允许方法调用，否则抛出权限异常。

```
@RestController
public class HelloController {

    @RequestMapping("/hello")
    @PreAuthorize("hasAuthority('test')") // 用户必须具备test权限才可以访问该方法
    public String hello(){
        return "hello";
    }
}
```

## 封装权限信息

我们前面在写UserDetailsServiceImpl的时候说过，在查询出用户后还要获取对应的权限信息，封装到UserDetails中返回。

我们先直接把权限信息写死封装到UserDetails中进行测试。

我们之前定义了UserDetails的实现类LoginUser，想要让其能封装权限信息就要对其进行修改。

@JSONField: <https://blog.csdn.net/u011291072/article/details/109692603>

```
@Data
@NoArgsConstructor
public class LoginUser implements UserDetails {

    // 用以封装用户信息
    private User user;

    // 存储权限信息
    private List<String> permissions;

    public LoginUser(User user, List<String> permissions) {
```

```

        this.user = user;
        this.permissions = permissions;
    }

    //存储SpringSecurity所需要的权限信息的集合
    @JSONField(serialize = false) // 不序列化该字段
    private List<GrantedAuthority> authorities;

    // 用户的权限集
    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        // 优化: 第一次转换, 后续调用直接返回
        if (authorities != null) {
            return authorities;
        }

        //把permissions中字符串类型的权限信息转换成GrantedAuthority对象存入authorities
        /*List<GrantedAuthority> authorityList = new ArrayList<>();
        for (String permission : permissions) {
            SimpleGrantedAuthority authority = new SimpleGrantedAuthority();
            authorityList.add(authority);
        }
        return authorityList;*/

        authorities = permissions.stream().
            map(SimpleGrantedAuthority::new)
            .collect(Collectors.toList());
        return authorities;
    }

    // 用户的加密后的密码, 不加密会使用{noop}前缀
    @Override
    public String getPassword() {
        return user.getPassword();
    }

    // 用户名
    @Override
    public String getUsername() {
        return user.getUserName();
    }

    // 帐户未过期
    @Override
    public boolean isAccountNonExpired() {
        return true;
    }

    // 帐户未锁定
    @Override
    public boolean isAccountNonLocked() {
        return true;
    }

```

中

```

// 凭证是否过期
@Override
public boolean isCredentialsNonExpired() {
    return true;
}

// 用户是否可用
@Override
public boolean isEnabled() {
    return true;
}
}

```

LoginUser修改完后我们就可以在UserDetailsServiceImpl中去把权限信息封装到LoginUser中了。我们写死权限进行测试，后面我们再从数据库中查询权限信息。

```

@Service
public class UserDetailsServiceImpl implements UserDetailsService {

    @Autowired
    private UserMapper userMapper;

    @Override
    public UserDetails loadUserByUsername(String username) throws
    UsernameNotFoundException {
        //根据用户名查询用户信息：注意使用我们自己定义的User实体类，别导错包
        LambdaQueryWrapper<User> wrapper = new LambdaQueryWrapper<>();
        wrapper.eq(User::getUserName, username);
        User user = userMapper.selectOne(wrapper);

        //如果查询不到数据就通过抛出异常来给出提示
        if (Objects.isNull(user)) {
            throw new RuntimeException("用户名或密码错误");
        }

        //TODO 根据用户查询权限信息 添加到LoginUser中
        List<String> list = new ArrayList<>(Arrays.asList("test", "admin"));

        //封装成UserDetails对象返回
        return new LoginUser(user, list);
    }
}

```

JwtAuthenticationTokenFilter:

```

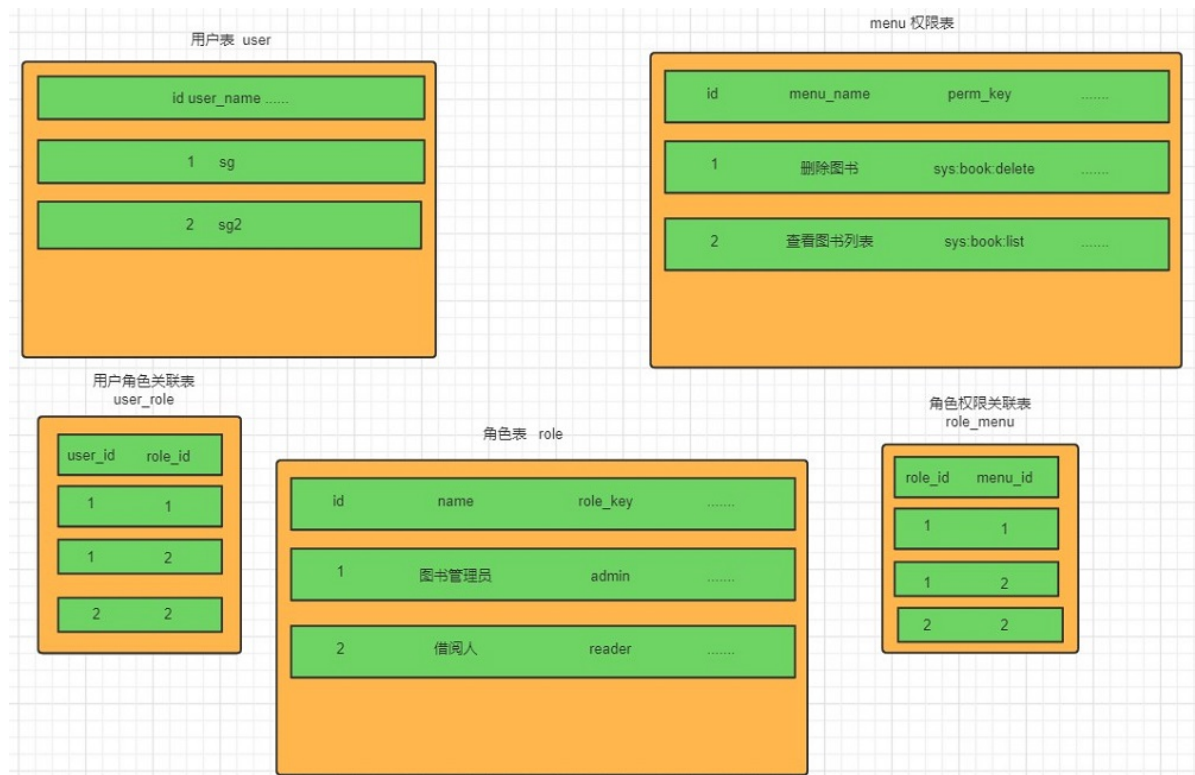
//TODO 获取权限信息封装到Authentication中
UsernamePasswordAuthenticationToken authenticationToken =
    new UsernamePasswordAuthenticationToken(loginUser, null,
    loginUser.getAuthorities());

```

# 从数据库查询权限信息

## RBAC权限模型

RBAC权限模型 (Role-Based Access Control) 即：基于角色的权限控制。这是目前最常被开发者使用也是相对易用、通用权限模型。



## 准备工作

```
/*Table structure for table `sys_menu` */  
  
DROP TABLE IF EXISTS `sys_menu`;  
  
CREATE TABLE `sys_menu` (  
  `id` bigint(20) NOT NULL AUTO_INCREMENT,  
  `menu_name` varchar(64) NOT NULL DEFAULT 'NULL' COMMENT '菜单名',  
  `path` varchar(200) DEFAULT NULL COMMENT '路由地址',  
  `component` varchar(255) DEFAULT NULL COMMENT '组件路径',  
  `visible` char(1) DEFAULT '0' COMMENT '菜单状态（0显示 1隐藏）',  
  `status` char(1) DEFAULT '0' COMMENT '菜单状态（0正常 1停用）',  
  `perms` varchar(100) DEFAULT NULL COMMENT '权限标识',  
  `icon` varchar(100) DEFAULT '#' COMMENT '菜单图标',  
  `create_by` bigint(20) DEFAULT NULL,  
  `create_time` datetime DEFAULT NULL,  
  `update_by` bigint(20) DEFAULT NULL,  
  `update_time` datetime DEFAULT NULL,  
  `del_flag` int(11) DEFAULT '0' COMMENT '是否删除（0未删除 1已删除）',  
  `remark` varchar(500) DEFAULT NULL COMMENT '备注',  
  PRIMARY KEY (`id`)  
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8mb4 COMMENT='菜单表';  
  
insert into sys_menu (id, menu_name, path, component, perms) values(1, '部门管理',  
'dept', 'system/dept/index', 'system:dept:list');
```

```

insert into sys_menu (id, menu_name, path, component, perms) values(2, '测试',
'test', 'system/test/index', 'system:test:list');

/*Table structure for table `sys_role` */

DROP TABLE IF EXISTS `sys_role`;

CREATE TABLE `sys_role` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `name` varchar(128) DEFAULT NULL,
  `role_key` varchar(100) DEFAULT NULL COMMENT '角色权限字符串',
  `status` char(1) DEFAULT '0' COMMENT '角色状态（0正常 1停用）',
  `del_flag` int(1) DEFAULT '0' COMMENT 'del_flag',
  `create_by` bigint(20) DEFAULT NULL,
  `create_time` datetime DEFAULT NULL,
  `update_by` bigint(20) DEFAULT NULL,
  `update_time` datetime DEFAULT NULL,
  `remark` varchar(500) DEFAULT NULL COMMENT '备注',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=3 DEFAULT CHARSET=utf8mb4 COMMENT='角色表';

insert into sys_role (id, name, role_key) values(1, 'CEO', 'ceo');
insert into sys_role (id, name, role_key) values(2, 'Coder', 'coder');

/*Table structure for table `sys_role_menu` */

DROP TABLE IF EXISTS `sys_role_menu`;

CREATE TABLE `sys_role_menu` (
  `role_id` bigint(200) NOT NULL AUTO_INCREMENT COMMENT '角色ID',
  `menu_id` bigint(200) NOT NULL DEFAULT '0' COMMENT '菜单id',
  PRIMARY KEY (`role_id`,`menu_id`)
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8mb4;

insert into sys_role_menu values(1, 1);
insert into sys_role_menu values(1, 2);

/*Table structure for table `sys_user` */

DROP TABLE IF EXISTS `sys_user`;

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=3 DEFAULT CHARSET=utf8mb4 COMMENT='用户表';

/*Table structure for table `sys_user_role` */

DROP TABLE IF EXISTS `sys_user_role`;

CREATE TABLE `sys_user_role` (
  `user_id` bigint(200) NOT NULL AUTO_INCREMENT COMMENT '用户id',
  `role_id` bigint(200) NOT NULL DEFAULT '0' COMMENT '角色id',
  PRIMARY KEY (`user_id`,`role_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

insert into sys_user_role values(1, 1);

```

### 通过userid查询用户权限:

```

SELECT
  DISTINCT m.`perms`
FROM
  sys_user_role ur
  LEFT JOIN `sys_role` r ON ur.`role_id` = r.`id`
  LEFT JOIN `sys_role_menu` rm ON ur.`role_id` = rm.`role_id`
  LEFT JOIN `sys_menu` m ON m.`id` = rm.`menu_id`
WHERE
  user_id = 1
  AND r.`status` = 0
  AND m.`status` = 0

```

### 菜单实体类:

@JsonInclude: [https://blog.csdn.net/Mr\\_Dracy/article/details/117950385](https://blog.csdn.net/Mr_Dracy/article/details/117950385)

```

/**
 * 菜单表(Menu)实体类
 *
 */
@TableName(value="sys_menu")
@Data
@AllArgsConstructor
@NoArgsConstructor
@JsonInclude(JsonInclude.Include.NON_NULL)
public class Menu implements Serializable {
    private static final long serialVersionUID = -54979041104113736L;

    @TableId
    private Long id;
    /**
     * 菜单名
     */
    private String menuName;
    /**
     * 路由地址

```

```

*/
private String path;
/**
 * 组件路径
 */
private String component;
/**
 * 菜单状态（0显示 1隐藏）
 */
private String visible;
/**
 * 菜单状态（0正常 1停用）
 */
private String status;
/**
 * 权限标识
 */
private String perms;
/**
 * 菜单图标
 */
private String icon;

private Long createdBy;

private Date createTime;

private Long updateBy;

private Date updateTime;
/**
 * 是否删除（0未删除 1已删除）
 */
private Integer delFlag;
/**
 * 备注
 */
private String remark;
}

```

## 代码实现

我们只需要根据用户id去查询到其所对应的权限信息即可。

所以我们可以先定义个mapper，其中提供一个方法可以根据userid查询权限信息。

```

public interface MenuMapper extends BaseMapper<Menu> {
    List<String> selectPermsByUserId(Long id);
}

```

尤其是自定义方法，所以需要创建对应的mapper文件，定义对应的sql语句：

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd" >

```

```

<mapper namespace="com.gs.mapper.MenuMapper">
  <select id="selectPermsByUserId" resultType="java.lang.String">
    SELECT
      DISTINCT m.`perms`
    FROM
      sys_user_role ur
      LEFT JOIN `sys_role` r ON ur.`role_id` = r.`id`
      LEFT JOIN `sys_role_menu` rm ON ur.`role_id` = rm.`role_id`
      LEFT JOIN `sys_menu` m ON m.`id` = rm.`menu_id`
    WHERE
      user_id = #{userid}
      AND r.`status` = 0
      AND m.`status` = 0
  </select>
</mapper>

```

在application.yaml中配置mapperXML文件的位置:

```

spring:
  datasource:
    url: jdbc:mysql://localhost:3306/security?characterEncoding=utf-8&serverTimezone=UTC
    username: root
    password: root
    driver-class-name: com.mysql.cj.jdbc.Driver
  redis:
    host: localhost
    port: 6379
  mybatis-plus:
    mapper-locations: classpath*/mapper/**/*.xml

```

然后我们可以在UserDetailsServiceImpl中去调用该mapper的方法查询权限信息封装到LoginUser对象中即可。

```

@Service
public class UserDetailsServiceImpl implements UserDetailsService {

    @Autowired
    private UserMapper userMapper;

    @Autowired
    private MenuMapper menuMapper;

    @Override
    public UserDetails loadUserByUsername(String username) throws
    UsernameNotFoundException {
        //根据用户名查询用户信息: 注意使用我们自己定义的用户实体类, 别导错包
        LambdaQueryWrapper<User> wrapper = new LambdaQueryWrapper<>();
        wrapper.eq(User::getUserName, username);
        User user = userMapper.selectOne(wrapper);

        //如果查询不到数据就通过抛出异常来给出提示
        if (Objects.isNull(user)) {
            throw new RuntimeException("用户名或密码错误");
        }
    }
}

```

```
    }

    // 测试写法
    // List<String> list = new ArrayList<>(Arrays.asList("test", "admin"));

    // 从数据库查询权限信息
    List<String> list = menuMapper.selectPermsByUserId(user.getId());

    //封装成UserDetails对象返回
    return new LoginUser(user, list);
}
}
```

```
@RestController
public class HelloController {

    @RequestMapping("/hello")
    @PreAuthorize("hasAuthority('system:test:list')")
    public String hello() {
        return "hello";
    }
}
```

## 五、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，然后配置给SpringSecurity即可。

### 5.1 自定义实现类

```

@Component
public class AuthenticationEntryPointImpl implements AuthenticationEntryPoint {

    // 处理认证过程中出现的异常
    @Override
    public void commence(HttpServletRequest request, HttpServletResponse
response, AuthenticationException authException) throws IOException,
ServletException {
        ResponseResult result = new
ResponseResult(HttpStatus.UNAUTHORIZED.value(), "认证失败,请重新登录");
        String json = JSON.toJSONString(result);
        webUtils.renderString(response,json);
    }
}

```

```

@Component
public class AccessDeniedHandlerImpl implements AccessDeniedHandler {

    // 处理授权过程中出现的异常
    @Override
    public void handle(HttpServletRequest request, HttpServletResponse response,
AccessDeniedException accessDeniedException) throws IOException,
ServletException {
        ResponseResult result = new ResponseResult(HttpStatus.FORBIDDEN.value(),
"权限不足");
        String json = JSON.toJSONString(result);
        webUtils.renderString(response,json);
    }
}

```

## 5.2 配置给SpringSecurity

先注入对应的配置类:

```

@Autowired
private AuthenticationEntryPoint authenticationEntryPoint;

@Autowired
private AccessDeniedHandler accessDeniedHandler;

```

然后我们可以使用HttpSecurity对象的方法去配置:

```

// 配置异常处理器
http.exceptionHandling().authenticationEntryPoint(authenticationEntryPoint).
accessDeniedHandler(accessDeniedHandler);

```

## 5.3 测试

The image displays two screenshots of a REST client interface. The first screenshot shows a POST request to `http://localhost:8080/user/login` with a JSON body: `{ "userName": "admin", "password": "123456" }`. The response is a JSON object: `{ "code": 401, "msg": "认证失败,请重新登录" }`. The second screenshot shows a GET request to `http://localhost:8080/hello` with headers: `Accept-Encoding: gzip, deflate, br`, `Connection: keep-alive`, and `token: eyJhbGciOiJIUzI1NiJ9.eyJqdGkiOiI0YzAxNzMwYTM5Y2E0OTI...`. The response is a JSON object: `{ "code": 403, "msg": "权限不足" }`.

## 六、跨域

浏览器出于安全的考虑，使用 XMLHttpRequest 对象发起 HTTP 请求时必须遵守同源策略，否则就是跨域的 HTTP 请求，默认情况下是被禁止的。同源策略要求源相同才能正常进行通信，即协议、域名、端口号都完全一致。

前后端分离项目，前端项目和后端项目一般都不是同源的，所以肯定会存在跨域请求的问题。

所以我们就要处理一下，让前端能进行跨域请求。

### 1、先对SpringBoot配置，运行跨域请求

```
@Configuration
public class CorsConfig implements WebMvcConfigurer {

    @Override
    public void addCorsMappings(CorsRegistry registry) {
        // 设置允许跨域的路径
        registry.addMapping("/**")
            // 设置允许跨域请求的域名
            .allowedOriginPatterns("*")
            // 是否允许cookie
            .allowCredentials(true)
            // 设置允许的请求方式
            .allowedMethods("GET", "POST", "DELETE", "PUT")
            // 设置允许的header属性
            .allowedHeaders("*")
            // 跨域允许时间
            .maxAge(3600);
    }
}
```

## 2、开启SpringSecurity的跨域访问

由于我们的资源都会受到SpringSecurity的保护，所以想要跨域访问还要让SpringSecurity运行跨域访问。

```
@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();

        //添加过滤器
        http.addFilterBefore(jwtAuthenticationTokenFilter,
            UsernamePasswordAuthenticationFilter.class);

        //配置异常处理器
        http.exceptionHandling()
            //配置认证失败处理器
            .authenticationEntryPoint(authenticationEntryPoint)
            .accessDeniedHandler(accessDeniedHandler);

        //允许跨域
        http.cors();
}
```

# 七、遗留小问题

## 7.1 其它权限校验方法

我们前面都是使用@PreAuthorize注解，然后在其中使用的是hasAuthority方法进行校验。SpringSecurity还为我们提供了其它方法，例如：hasAnyAuthority，hasRole，hasAnyRole等。

这里我们先不急着去介绍这些方法，我们先去理解hasAuthority的原理，然后再去学习其他方法你就更容易理解，而不是死记硬背区别。并且我们也可以选择定义校验方法，实现我们自己的校验逻辑。

hasAuthority方法实际是执行到了SecurityExpressionRoot的hasAuthority，大家只要断点调试既可知道它内部的校验原理。

它内部其实是调用authentication的getAuthorities方法获取用户的权限列表。然后判断我们存入的方法参数数据在权限列表中。

hasAnyAuthority方法可以传入多个权限，只有用户有其中任意一个权限都可以访问对应资源。

```
@PreAuthorize("hasAnyAuthority('admin','test','system:dept:list')")
public String hello(){
    return "hello";
}
```

hasRole要求有对应的角色才可以访问，但是它内部会把我们传入的参数拼接上 ROLE\_ 后再去比较。所以这种情况下要用用户对应的权限也要有 ROLE\_ 这个前缀才可以。

```
@PreAuthorize("hasRole('system:dept:list')")
public String hello(){
    return "hello";
}
```

hasAnyRole 有任意的角色就可以访问。它内部也会把我们传入的参数拼接上 ROLE\_ 后再去比较。所以这种情况下要用用户对应的权限也要有 ROLE\_ 这个前缀才可以。

```
@PreAuthorize("hasAnyRole('admin','system:dept:list')")
public String hello(){
    return "hello";
}
```

## 7.2 自定义权限校验方法

我们也可以定义自己的权限校验方法，在@PreAuthorize注解中使用我们的方法。

```
package com.gs.expression;

//...

@Component("ex")
public class MyExpressionRoot {

    public boolean hasAuthority(String authority){
        //获取当前用户的权限
        Authentication authentication =
        SecurityContextHolder.getContext().getAuthentication();
        LoginUser loginUser = (LoginUser) authentication.getPrincipal();
        List<String> permissions = loginUser.getPermissions();
        //判断用户权限集合中是否存在authority
        return permissions.contains(authority);
    }
}
```

在SPeL表达式中使用 @ex相当于获取容器中bean的名字为ex的对象。然后再调用这个对象的 hasAuthority方法。

```
@RequestMapping("/hello")
@PreAuthorize("@ex.hasAuthority('system:dept:list')")
public String hello(){
    return "hello";
}
```



## 7.3 基于配置的权限控制

我们也可以在配置类中使用配置的方式对资源进行权限控制。

注：该方式更多用于静态资源放行；

```
@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()
        .antMatchers("/testCors").hasAuthority("system:dept:list")
        // 除上面外的所有请求全部需要鉴权认证
        .anyRequest().authenticated();

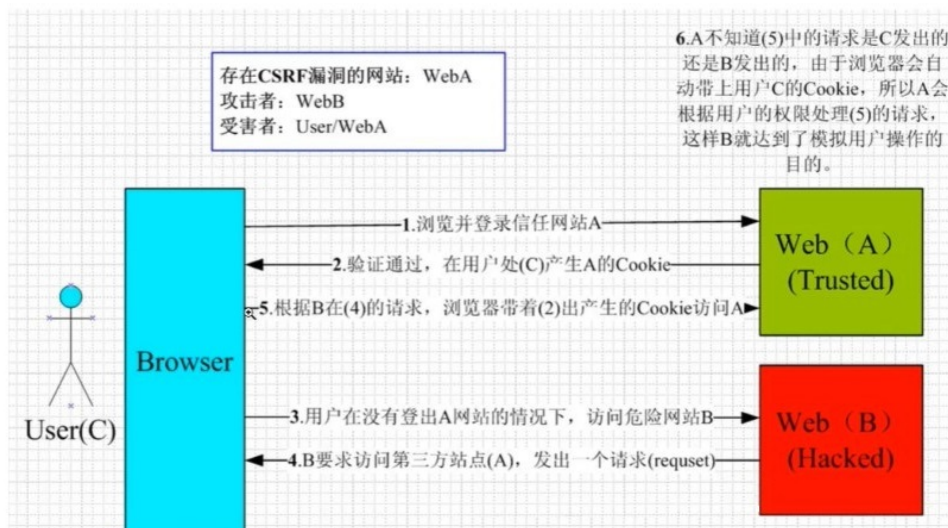
    //添加过滤器
    http.addFilterBefore(jwtAuthenticationTokenFilter,
        UsernamePasswordAuthenticationFilter.class);

    //配置异常处理器
    http.exceptionHandling()
        //配置认证失败处理器
        .authenticationEntryPoint(authenticationEntryPoint)
        .accessDeniedHandler(accessDeniedHandler);

    //允许跨域
    http.cors();
}
```

## 7.4 CSRF

CSRF是指跨站请求伪造（Cross-site request forgery），是web常见的攻击之一。



SpringSecurity去防止CSRF攻击的方式就是通过csrf\_token。后端会生成一个csrf\_token，前端发起请求的时候需要携带这个csrf\_token，后端会有过滤器进行校验，如果没有携带或者是伪造的就不允许访问。

我们可以发现CSRF攻击依靠的是cookie中所携带的认证信息。但是在前后端分离的项目中我们的认证信息其实是token，而token并不是存储中cookie中，并且需要前端代码去把token设置到请求头中才可以，所以CSRF攻击也就不担心了。

More: <https://blog.csdn.net/freeking101/article/details/86537087>

## 7.5 认证成功处理器

实际上在UsernamePasswordAuthenticationFilter进行登录认证的时候，如果登录成功了是会调用AuthenticationSuccessHandler的方法进行认证成功后的处理的。AuthenticationSuccessHandler就是认证成功处理器。

我们也可以自己去自定义成功处理器进行成功后的相应处理。

```
@Component
public class MySuccessHandler implements AuthenticationSuccessHandler {

    @Override
    public void onAuthenticationSuccess(HttpServletRequest request,
        HttpServletResponse response, Authentication authentication) throws IOException,
        ServletException {
        System.out.println("认证成功");
    }
}
```

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    private AuthenticationSuccessHandler successHandler;

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.formLogin().successHandler(successHandler);

        http.authorizeRequests().anyRequest().authenticated();
    }
}
```

## 7.6 认证失败处理器

实际上在UsernamePasswordAuthenticationFilter进行登录认证的时候，如果认证失败了是会调用AuthenticationFailureHandler的方法进行认证失败后的处理的。AuthenticationFailureHandler就是登录失败处理器。

我们也可以自己去自定义失败处理器进行失败后的相应处理。

```

@Component
public class MyFailureHandler implements AuthenticationFailureHandler {
    @Override
    public void onAuthenticationFailure(HttpServletRequest request,
    HttpServletResponse response, AuthenticationException exception) throws
    IOException, ServletException {
        System.out.println("认证失败");
    }
}

```

```

@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    private AuthenticationSuccessHandler successHandler;

    @Autowired
    private AuthenticationFailureHandler failureHandler;

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.formLogin()
            // 配置认证成功处理器
            .successHandler(successHandler)
            // 配置认证失败处理器
            .failureHandler(failureHandler);

        http.authorizeRequests().anyRequest().authenticated();
    }
}

```

## 7.7 登出成功处理器

```

@Component
public class MyLogoutSuccessHandler implements LogoutSuccessHandler {
    @Override
    public void onLogoutSuccess(HttpServletRequest request, HttpServletResponse
    response, Authentication authentication) throws IOException, ServletException {
        System.out.println("注销成功");
    }
}

```

```

@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    private AuthenticationSuccessHandler successHandler;

    @Autowired
    private AuthenticationFailureHandler failureHandler;

    @Autowired
    private LogoutSuccessHandler logoutSuccessHandler;
}

```

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.formLogin()
        // 配置认证成功处理器
        .successHandler(successHandler)
        // 配置认证失败处理器
        .failureHandler(failureHandler);

    http.logout()
        //配置注销成功处理器
        .logoutSuccessHandler(logoutSuccessHandler);

    http.authorizeRequests().anyRequest().authenticated();
}
}
```