# 【熟悉】认证授权的基础概念

## 什么是认证

    进入移动互联网时代，大家每天都在刷手机，常用的软件有微信、支付宝、头条等，下边拿微信来举例子说明认证相关的基本概念，在初次使用微信前需要注册成为微信用户，然后输入账号和密码即可登录微信，输入账号和密码登录微信的过程就是认证。

系统为什么要认证？

    认证是为了保护系统的隐私数据与资源，用户的身份合法方可访问该系统的资源。

    认证 ：用户认证就是判断一个用户的身份是否合法的过程，用户去访问系统资源时系统要求验证用户的身份信息，身份合法方可继续访问，不合法则拒绝访问。常见的用户身份认证方式有：用户名密码登录，二维码登录，手机短信登录，指纹认证等方式。

## 什么是会话

    用户认证通过后，为了避免用户的每次操作都进行认证可将用户的信息保证在会话中。会话就是系统为了保持当前用户的登录状态所提供的机制，常见的有基于session方式、基于token方式等。

### 基于session的认证方式

    它的交互流程是，用户认证成功后，在服务端生成用户相关的数据保存在session(当前会话)中，发给客户端的sesssion\_id 存放到 cookie 中，这样用户客户端请求时带上 session\_id 就可以验证服务器端是否存在 session 数据，以此完成用户的合法校验，当用户退出系统或session过期销毁时,客户端的session\_id也就无效了。

### 基于token方式认证方式

    它的交互流程是，用户认证成功后，服务端生成一个token发给客户端，客户端可以放到 cookie 或 localStorage等存储中，每次请求时带上 token，服务端收到token通过验证后即可确认用户身份。

    基于session的认证方式由Servlet规范定制，服务端要存储session信息需要占用内存资源，客户端需要支持cookie；基于token的方式则一般不需要服务端存储token，并且不限制客户端的存储方式。如今移动互联网时代更多类型的客户端需要接入系统，系统多是采用前后端分离的架构进行实现，所以基于token的方式更适合。

## 什么是授权

还拿微信来举例子，微信登录成功后用户即可使用微信的功能，比如，发红包、发朋友圈、添加好友等，没有绑定银行卡的用户是无法发送红包的，绑定银行卡的用户才可以发红包，发红包功能、发朋友圈功能都是微信的资源即功能资源，用户拥有发红包功能的权限才可以正常使用发送红包功能，拥有发朋友圈功能的权限才可以使用发朋友圈功能，这个根据用户的权限来控制用户使用资源的过程就是授权。

### 为什么要授权？

认证是为了保证用户身份的合法性，授权则是为了更细粒度的对隐私数据进行划分，授权是在认证通过后发生的，控制不同的用户能够访问不同的资源。

授权： 授权是用户认证通过根据用户的权限来控制用户访问资源的过程，拥有资源的访问权限则正常访问，没有权限则拒绝访问。

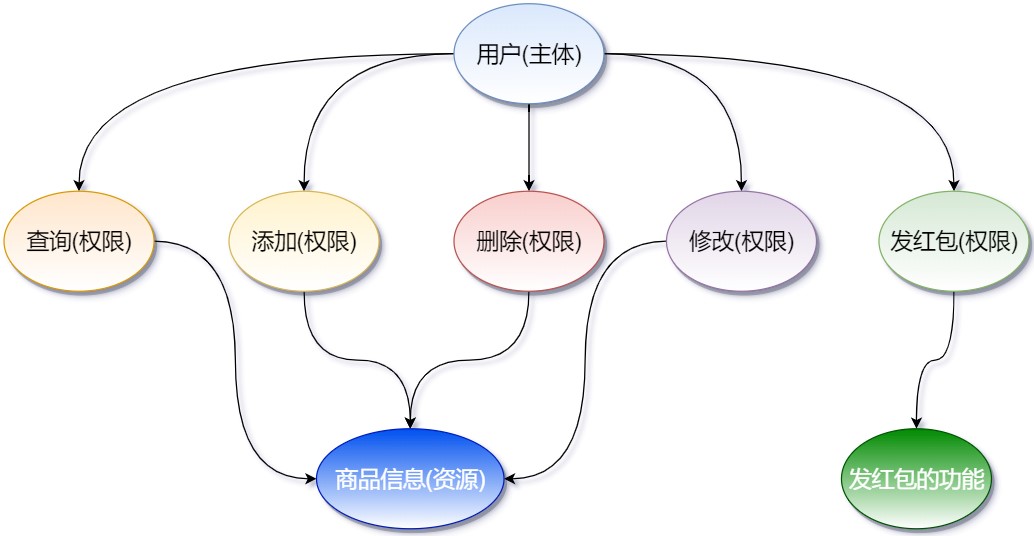
## 授权的数据模型

如何进行授权即如何对用户访问资源进行控制，首先需要学习授权相关的数据模型。

授权可简单理解为Who对What(which)进行How操作，包括如下：

Who，即主体（Subject），主体一般是指用户，也可以是程序，需要访问系统中的资源。 What，即资源（Resource），如系统菜单、页面、按钮、代码方法、系统商品信息、系统订单信息等。系统菜单、页面、按钮、代码方法都属于系统功能资源，对于web系统每个功能资源通常对应一个URL；系统商品信息、系统订单信息都属于实体资源（数据资源），实体资源由资源类型和资源实例组成，比如商品信息为资源类型，商品编号 为001的商品为资源实例。 How，权限/许可（Permission），规定了用户对资源的操作许可，权限离开资源没有意义，如用户查询权限、用户添加权限、某个代码方法的调用权限、编号为001的用户的修改权限等，通过权限可知用户对哪些资源都有哪些操作可。

主体、资源、权限关系如下图：



主体、资源、权限相关的数据模型如下：

主体（用户id、账号、密码、...）

资源（资源id、资源名称、访问地址、...）

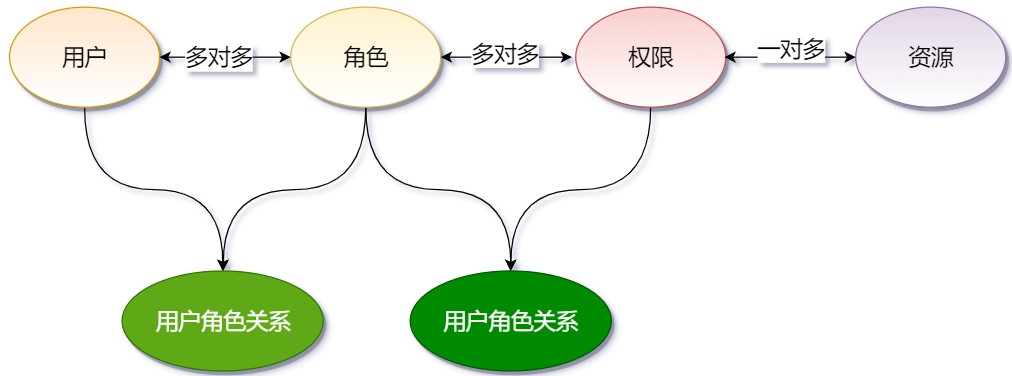
权限（权限id、权限标识、权限名称、资源id、...）

角色（角色id、角色名称、...）

角色和权限关系（角色 id、权限id、...）

主体（用户）和角色关系（用户id、角色id、...）

主体（用户）、资源、权限关系如下图：



通常企业开发中将资源和权限表合并为一张权限表，如下：

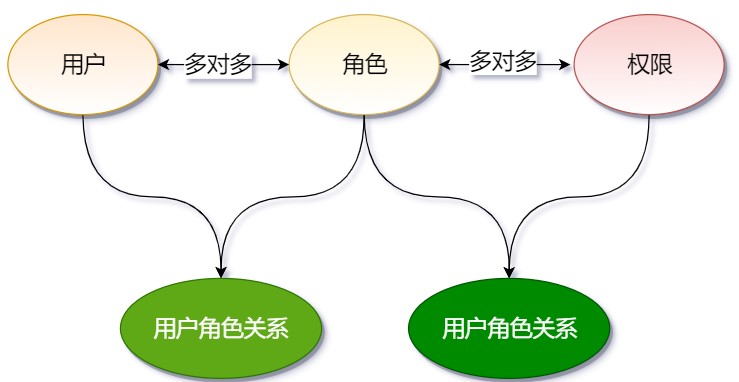
资源（资源id、资源名称、访问地址、...）

权限（权限id、权限标识、权限名称、资源id、...）

合并为：

权限（权限id、权限标识、权限名称、资源名称、资源访问地址、...）

修改后数据模型之间的关系如下图：



## RBAC

如何实现授权？业界通常基于RBAC实现授权。

### 基于角色的访问控制

RBAC基于角色的访问控制（Role-Based Access Control）是按角色进行授权，比如：主体的角色为总经理可以查询企业运营报表，查询员工工资信息等

根据上图中的判断逻辑，授权代码可表示如下：

**if(主体.hasRole("总经理角色id")){**

**查询工资**

**}**

如果上图中查询工资所需要的角色变化为总经理和部门经理，此时就需要修改判断逻辑为“判断用户的角色是否是总经理或部门经理”，修改代码如下：

**if(主体.hasRole("总经理角色id") ||  主体.hasRole("部门经理角色id")){**

**查询工资**

**}**

根据上边的例子发现，当需要修改角色的权限时就需要修改授权的相关代码，系统可扩展性差。

### 基于资源的访问控制

RBAC基于资源的访问控制（Resource-Based Access Control）是按资源（或权限）进行授权，比如：用户必须具有查询工资权限才可以查询员工工资信息等

如下的判断

**if(主体.hasPermission("查询工资") ){**

**查询工资**

**}**

优点：系统设计时定义好查询工资的权限标识，即使查询工资所需要的角色变化为总经理和部门经理也不需要修改授权代码，系统可扩展性强。

# 【熟悉】Spring Security 简介

## SpringSecurity 是什么

Spring Security是一个能够为基于Spring的企业应用系统提供声明式的安全访问控制解决方案的安全框架。它提供了一组可以在Spring应用上下文中配置的Bean，充分利用了Spring IoC，DI（控制反转Inversion of Control ,DI:Dependency Injection 依赖注入）和AOP（面向切面编程）功能，为应用系统提供声明式的安全访问控制功能，减少了为企业系统安全控制编写大量重复代码的工作。

以上解释来源于百度百科。可以一句话来概括，SpringSecurity 是一个安全框架。

## SpringSecurity 能做什么

SpringSecurity 是一个安全框架，那它提供了哪些安全措施呢？主要是三个方面，

身份认证（你是谁？）

权限校验（你能做什么？允许操作的范围）

攻击防护（防止伪造身份）

## SpringSecurity 官方文档

<https://www.springcloud.cc/spring-security.html>

<https://spring.io/projects/spring-security>

<https://docs.spring.io/spring-security/site/docs/5.3.2.RELEASE/reference/html5/>

## 同类产品

同类产品中的佼佼者就是 Shiro 了，那这两者有什么区别的吗？

### 共同点

认证功能

授权功能

加密功能

会话功能

remeberMe功能

……

就功能上来说基本是相似。

### 不同点

优点：

Spring Security基于Spring开发，项目中如果使用Spring作为基础，配合Spring Security做权限更加方便。而Shiro需要和Spring进行整合。

Spring Security功能比Shiro更加丰富，例如安全防护方面。

Spring Security社区资源相对比Shiro更加丰富。

如果使用的是Spring Boot，Spring Cloud的话，三者可以无缝集成。

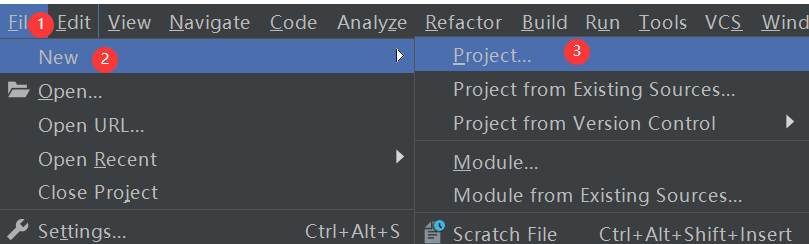
缺点：

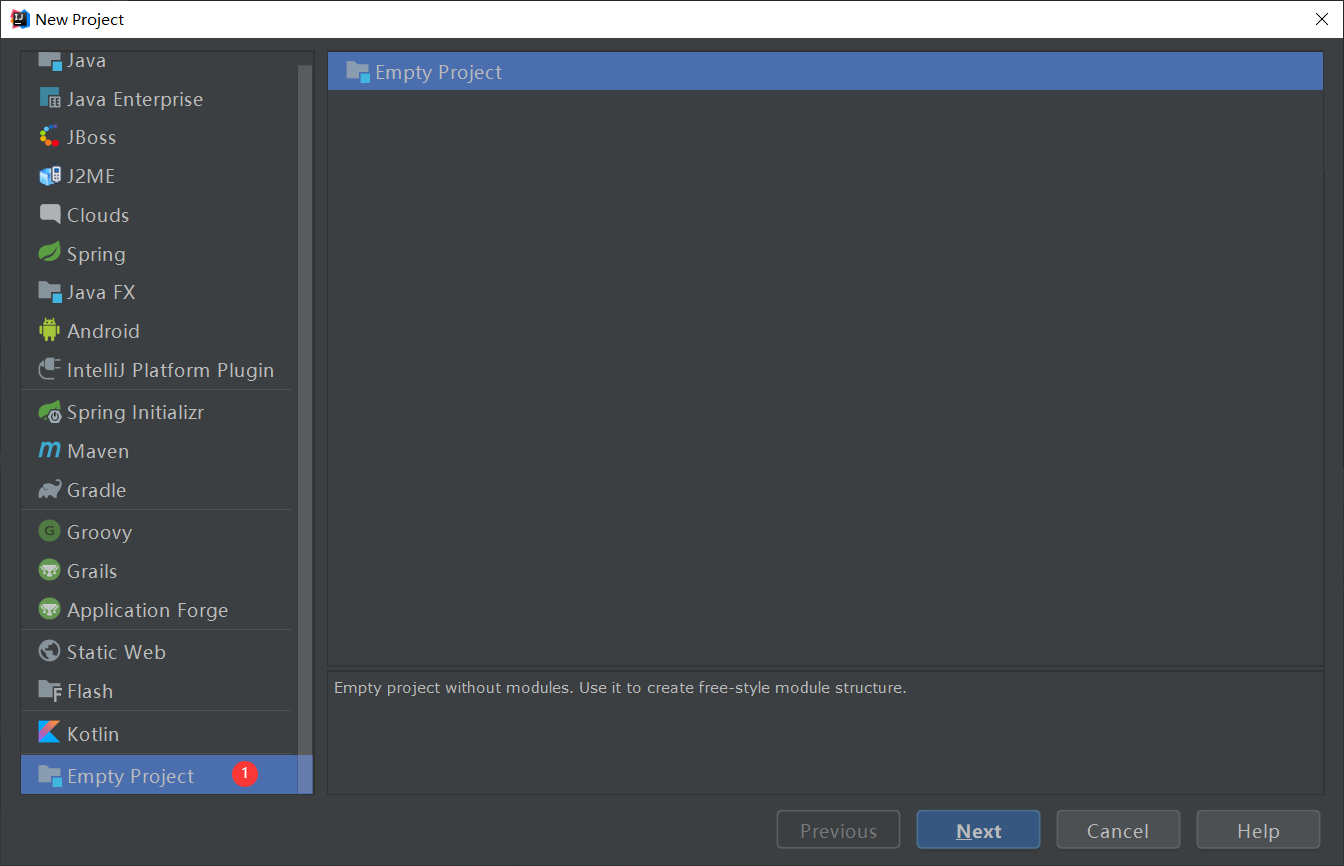
Shiro的配置和使用比较简单，Spring Security上手复杂些。

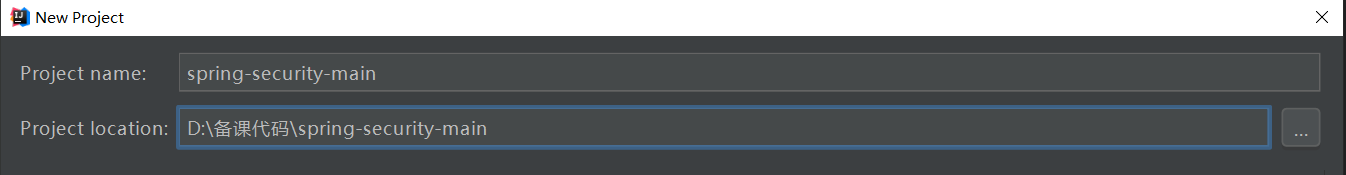
Shiro依赖性低，不需要任何框架和容器，可以独立运行，而Spring Security依赖Spring容器。

# 【掌握】Spring Security初体验

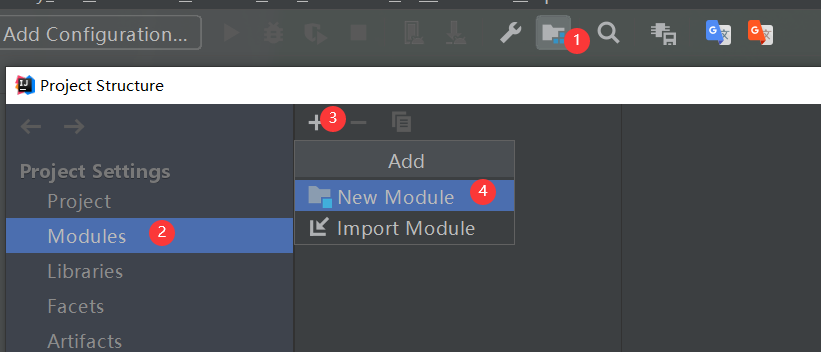
## 创建空项目spring-security-main

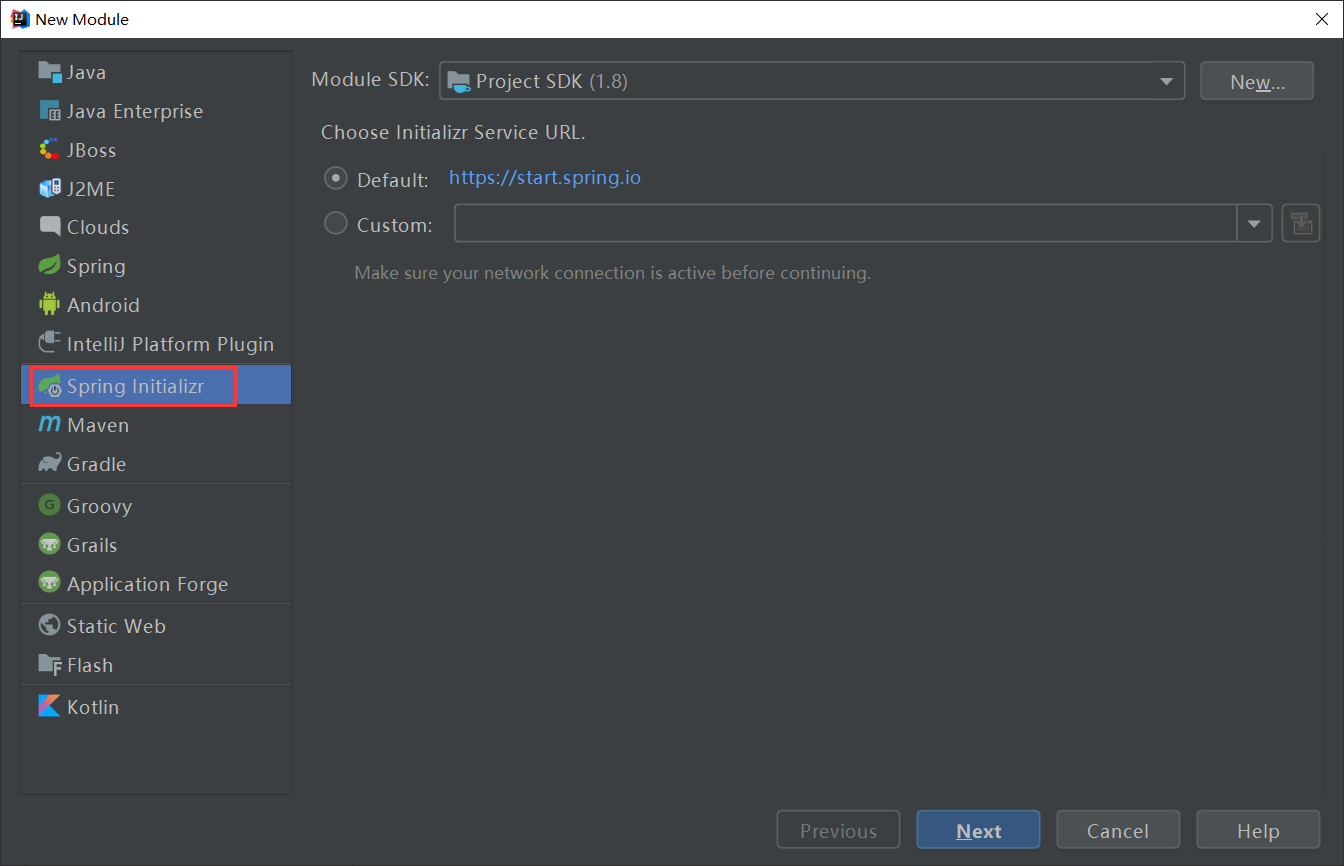


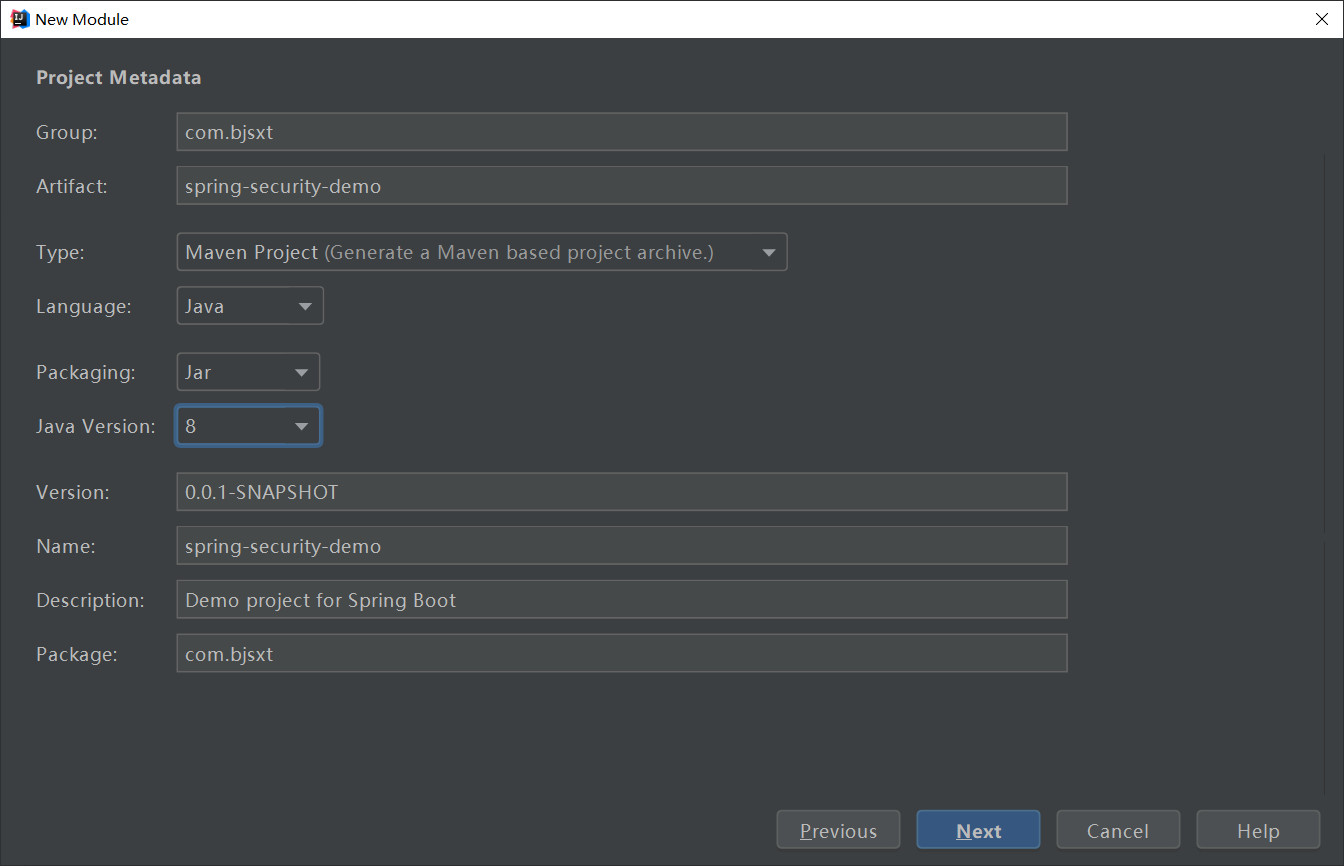


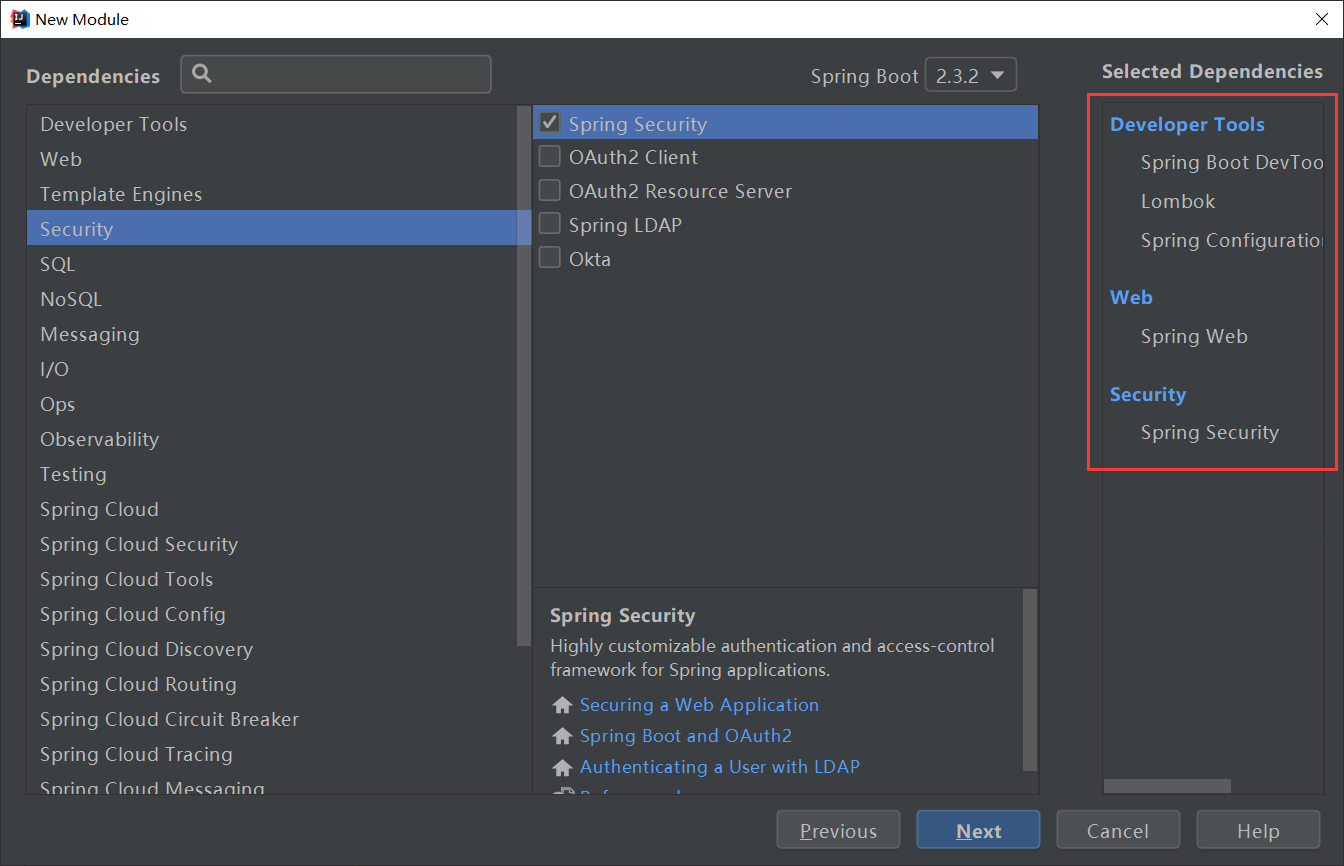


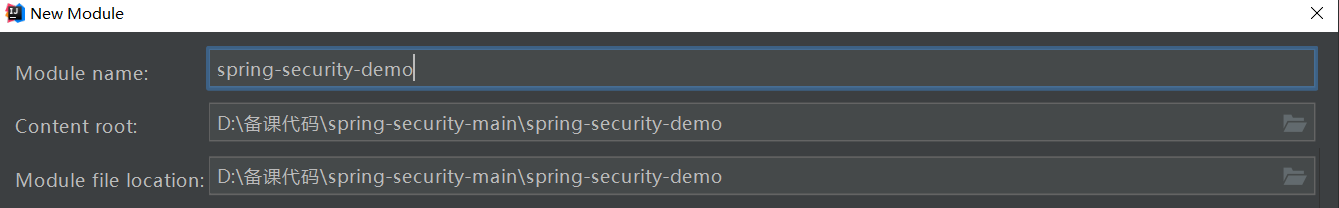
## 创建模块spring-security-demo

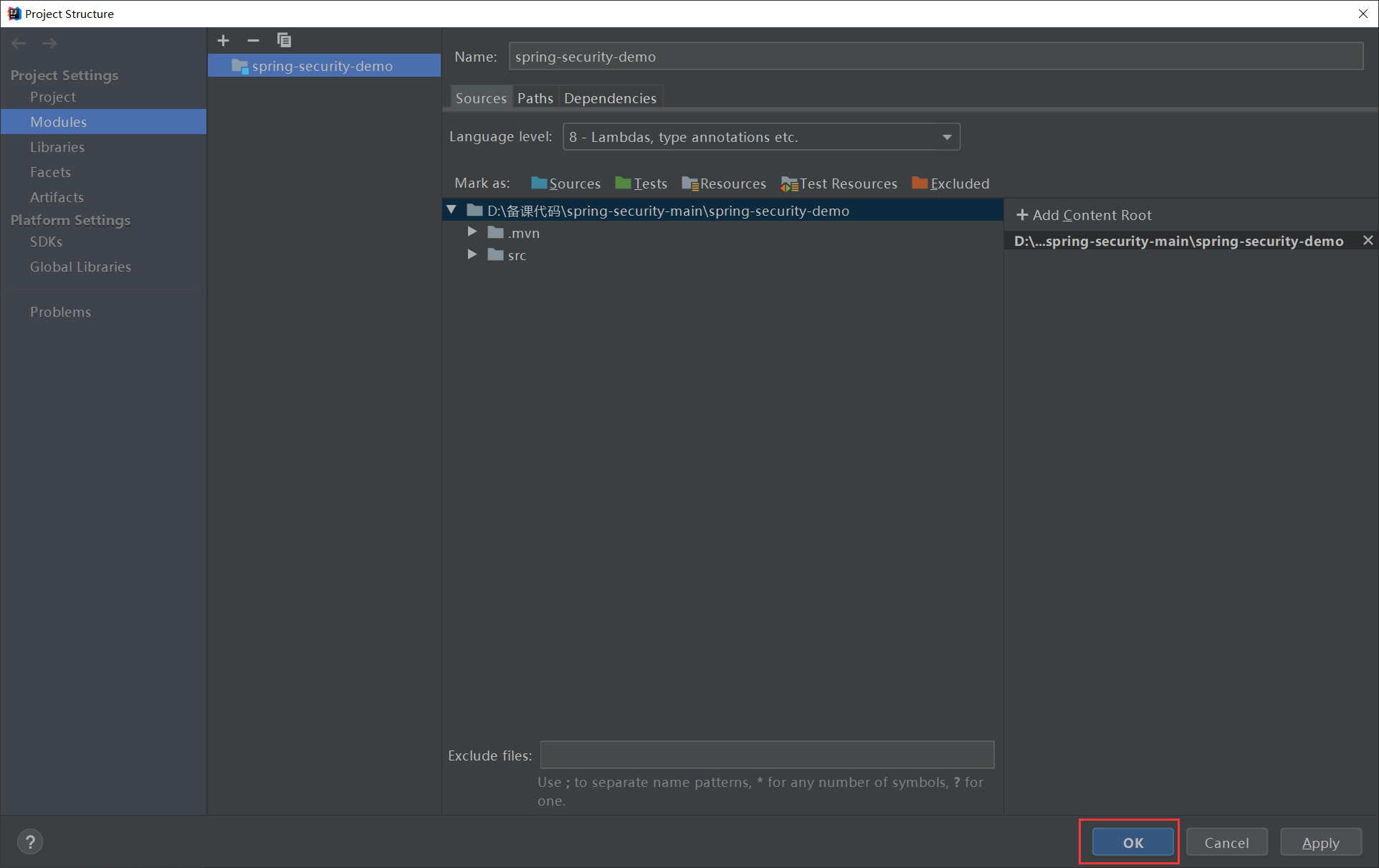












## pom.xml说明

<?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 https://maven.apache.org/xsd/maven-4.0.0.xsd">  
 <modelVersion>4.0.0</modelVersion>  
 <parent>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-parent</artifactId>  
 <version>2.3.2.RELEASE</version>  
 <relativePath/> <!-- lookup parent from repository -->  
 </parent>  
 <groupId>com.bjsxt</groupId>  
 <artifactId>spring-security-demo</artifactId>  
 <version>0.0.1-SNAPSHOT</version>  
 <name>spring-security-demo</name>  
 <description>Demo project for Spring Boot</description>  
  
 <properties>  
 <java.version>1.8</java.version>  
 </properties>  
  
 <dependencies>  
 <!--security的启动器-->  
 <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>org.springframework.boot</groupId>  
 <artifactId>spring-boot-configuration-processor</artifactId>  
 <optional>true</optional>  
 </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>  
 <exclusions>  
 <exclusion>  
 <groupId>org.junit.vintage</groupId>  
 <artifactId>junit-vintage-engine</artifactId>  
 </exclusion>  
 </exclusions>  
 </dependency>  
 <!--security测试包，可以使用Mock 不推荐使用-->  
 <dependency>  
 <groupId>org.springframework.security</groupId>  
 <artifactId>spring-security-test</artifactId>  
 <scope>test</scope>  
 </dependency>  
 </dependencies>  
 <build>  
 <plugins>  
 <plugin>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-maven-plugin</artifactId>  
 </plugin>  
 </plugins>  
 </build>  
</project>

## 修改启动类

@SpringBootApplication  
@EnableWebSecurity //启用security 在5.X版本之后可以不用加，默认就是开启的  
public class SpringSecurityDemoApplication {  
  
 public static void main(String[] args) {  
 SpringApplication.*run*(SpringSecurityDemoApplication.class, args);  
 }  
}

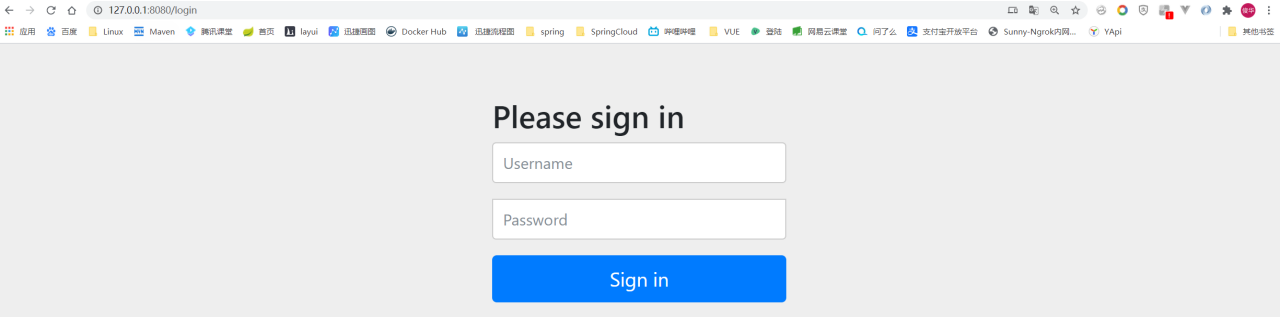
## 创建控制器HelloController

@RestController  
public class HelloController {  
  
 @RequestMapping("hello")  
 public String hello(){  
 return "你好 spring security";  
 }  
}

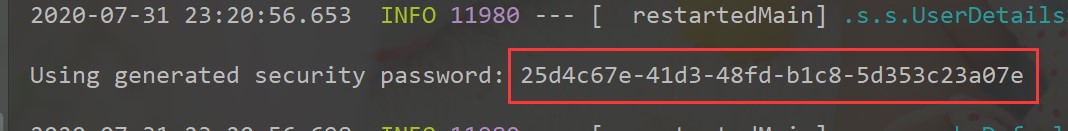
## 启动项目访问及存在的问题

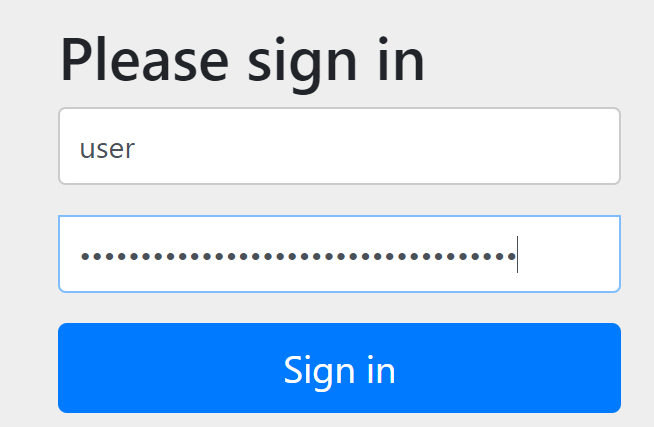
[http://127.0.0.1:8080/hello](http://127.0.0.1:8080/login)

发现我们无法访问hello这个请求，这是因为spring Security默认拦截了所有请求

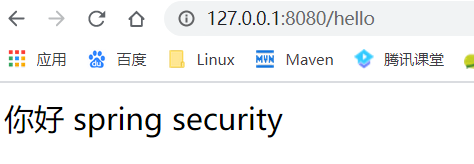


使用user+启动日志里面的密码登陆



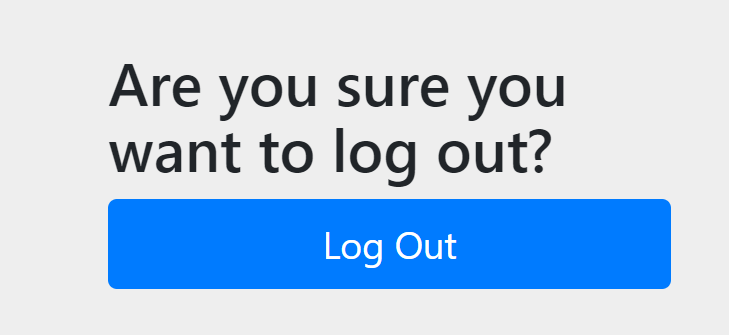


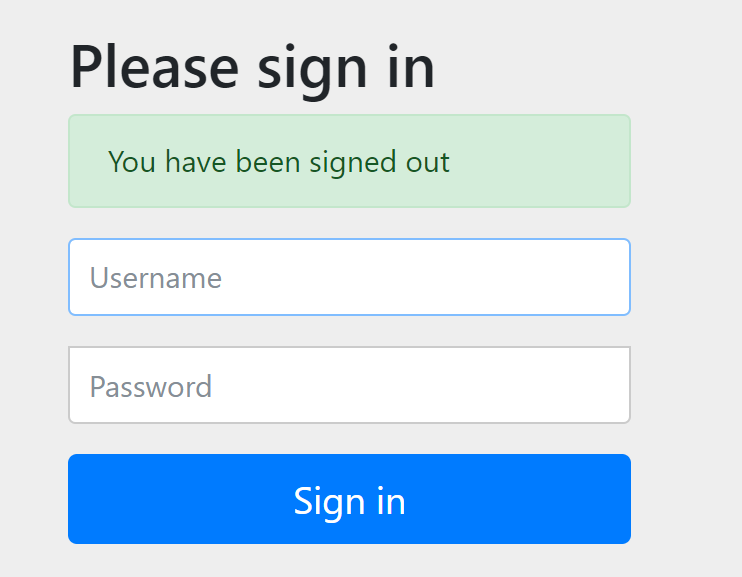
登陆成功之后会自动跳转到我们之前请求的hello



## 退出访问

<http://127.0.0.1:8080/logout>





## 配置密码

从上面的演示中，我们发现密码在控制台是日志形式输出的，那么能不能配置用户名和密码呢，可以的

### 创建application.yaml

spring:  
 security:  
 user:  
 name: system *#配置默认的用户名* password: 123456 *#配置默认的密码* roles: *#配置admin用户的角色* - ADMIN  
 - SYSTEM

### 再次登陆测试

使用system/123456再登陆测试

## 总结

从上面的体验来说，是不是感觉很简单，但是别急。后面的东西还是有点难度的

如下：

* 如何读取数据库的用户名和密码
* 如何对密码加密
* 如何使用数据的角色和权限
* 如何配置方法级别的权限访问
* 如何自定义登陆页面
* 如何集成redis把登陆信息放到Redis
* ……………………

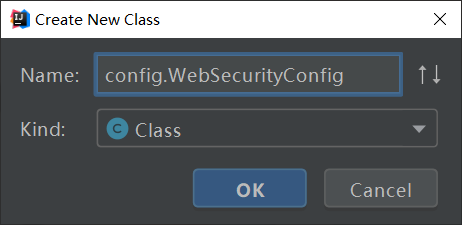
来吧，跟着雷哥一步一步走，别慌

# 【掌握】Spring Security配置多用户认证

## 概述

认证就是登陆，我们现在没有连接数据库，那么我们可以模拟下用户名和密码

## 创建认证的配置类

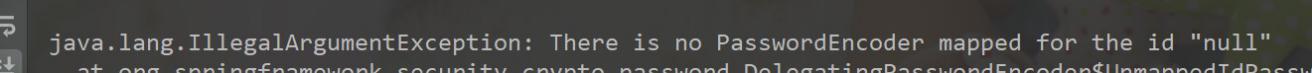


|  |
| --- |
| @Configuration public class WebSecurityConfig extends WebSecurityConfigurerAdapter {   */\*\*  \* 配置用户信息 模拟内存数据  \*/* @Override  protected void configure(AuthenticationManagerBuilder auth) throws Exception {  auth.inMemoryAuthentication()  .withUser(User.*withUsername*("admin").password("123456").roles("admin").build())  .withUser(User.*withUsername*("zhangsan").password("123456").roles("user").build())  .withUser(User.*withUsername*("lisi").password("123456").roles("admin", "user").build());  }  } |

## 重启应用使用admin/123456配置

这里注意，只要我们配置上面的配置类之后，之前在yaml里面配置的用户名和密码就无效了

## 登陆之后出现异常



这个是因为spring Sercurity强制要使用密码加密，当然我们也可以不加密，但是官方要求是不管你是否加密，都必须配置一个类似Shiro的凭证匹配器

### 配置不加密的凭证【修改WebSecurityConfig】

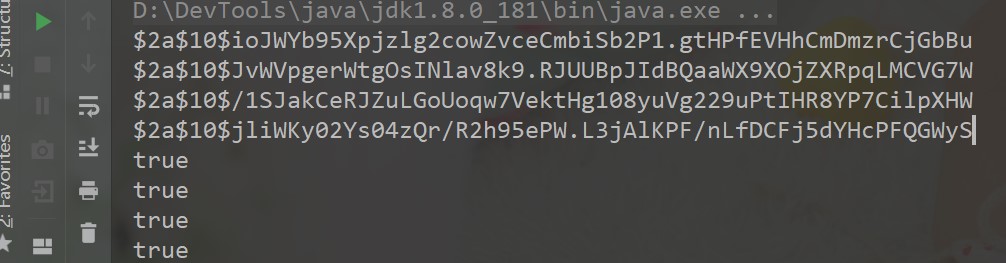
*/\*\*  
 \* 从 Spring5 开始，强制要求密码要加密， 如果非不想加密，可 以使用一个过期的 PasswordEncoder 的实例 NoOpPasswordEncoder，  
 \* 但是不建议这么做，毕竟不安全。  
 \*/*@Bean  
public PasswordEncoder passwordEncoder() {  
 return NoOpPasswordEncoder.*getInstance*();  
}

### 配置加密的凭证【修改WebSecurityConfig】

|  |
| --- |
| */\*\*  \* 配置用户信息 模拟内存数据  \*/* @Override  protected void configure(AuthenticationManagerBuilder auth) throws Exception {  auth.inMemoryAuthentication() // .withUser(User.withUsername("admin").password("123456").roles("admin").build()) // .withUser(User.withUsername("zhangsan").password("123456").roles("user").build()) // .withUser(User.withUsername("lisi").password("123456").roles("admin", "user").build());  .withUser(User.*withUsername*("admin").password(passwordEncoder.encode("123456")).roles("admin").build())  .withUser(User.*withUsername*("zhangsan").password(passwordEncoder.encode("123456")).roles("user").build())  .withUser(User.*withUsername*("lisi").password(passwordEncoder.encode("123456")).roles("admin", "user").build());  }   @Autowired  private PasswordEncoder passwordEncoder;   */\*\*  \* 从 Spring5 开始，强制要求密码要加密， 如果非不想加密，可 以使用一个过期的 PasswordEncoder 的实例 NoOpPasswordEncoder，  \* 但是不建议这么做，毕竟不安全。  \*/* @Bean  public PasswordEncoder passwordEncoder() { // return NoOpPasswordEncoder.getInstance();  return new BCryptPasswordEncoder();  } |

### 加密的凭证的测试

package com.bjsxt.controller;  
  
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;  
  
*/\*\*  
 \* Created with IntelliJ IDEA.  
 \*  
 \* @Auther: 雷哥  
 \* @Date: 2020/07/31/23:56  
 \* @Description:  
 \*/*public class TestPasswordEncoder {  
  
 public static void main(String[] args) {  
 BCryptPasswordEncoder encoder = new BCryptPasswordEncoder();  
 String encode1 = encoder.encode("123456");  
 String encode2 = encoder.encode("123456");  
 String encode3 = encoder.encode("123456");  
 String encode4 = encoder.encode("123456");  
 System.*out*.println(encode1);  
 System.*out*.println(encode2);  
 System.*out*.println(encode3);  
 System.*out*.println(encode4);  
  
 System.*out*.println(encoder.matches("123456",encode1));  
 System.*out*.println(encoder.matches("123456",encode2));  
 System.*out*.println(encoder.matches("123456",encode3));  
 System.*out*.println(encoder.matches("123456",encode4));  
 //发现特点是相同的字符串加密之后的结果都不一样，但是比较的时候是一样的，这个算法比shiro的MD5好用，不用自己在数据库去存盐了  
 }  
}



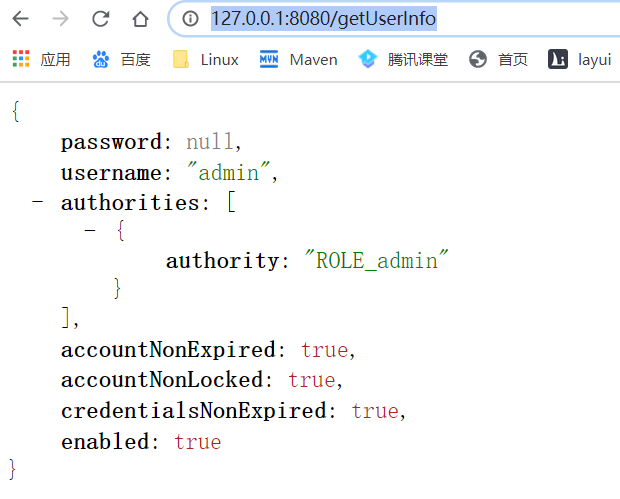
## 或取用户登陆信息

### 修改HelloController

@RequestMapping("getUserInfo")  
public Object getUserInfo() {  
 */\*\*  
 SecurityContextHolder.getContext()获取安全上下文对象，就是那个保存在 ThreadLocal 里面的安全上下文对象  
 总是不为null(如果不存在，则创建一个authentication属性为null的empty安全上下文对象)  
 获取当前认证了的 principal(当事人),或者 request token (令牌)  
 如果没有认证，会是 null,该例子是认证之后的情况  
 \*/* Authentication authentication = SecurityContextHolder.*getContext*().getAuthentication();  
 //有登陆用户就返回登录用户，没有就返回null  
 if (authentication != null) {  
 if (authentication instanceof AnonymousAuthenticationToken) {  
 return null;  
 }  
 if (authentication instanceof UsernamePasswordAuthenticationToken) {  
 return authentication.getPrincipal();  
 }  
 }  
 return null;  
}

### 再次登陆之后访问getUserInfo

<http://127.0.0.1:8080/getUserInfo>



# 【掌握】Spring Security用户角色、权限、授权拦截配置

## 角色和权限配置

*/\*\*  
 \* 配置用户信息 模拟内存数据  
 \*/*@Override  
protected void configure(AuthenticationManagerBuilder auth) throws Exception {  
 */\*\*  
 \* 重点说明  
 \* 在开发中，我们一般只针对权限，很少去使用角色，后面的讲解中我们以权限为主也就是authorities这里面的东西  
 \*/* auth.inMemoryAuthentication()  
 .withUser(User.*withUsername*("admin").password(passwordEncoder.encode("123456"))  
 .roles("admin").build())//admin用户只有admin的角色  
 .withUser(User.*withUsername*("zhangsan").password(passwordEncoder.encode("123456"))  
 .authorities("hello:query", "hello:add","hello:update","hello:delete").build())  
 .withUser(User.*withUsername*("lisi").password(passwordEncoder.encode("123456"))  
 .authorities("hello:query", "hello:add").build())  
 .withUser(User.*withUsername*("wangwu").password(passwordEncoder.encode("123456"))  
 .authorities("hello:query", "hello:add", "hello:update").build())  
 .withUser(User.*withUsername*("zhaoliu").password(passwordEncoder.encode("123456"))  
 .authorities("hello:query", "hello:delete").build());  
}

## 授权拦截配置

@Override  
 protected void configure(HttpSecurity http) throws Exception {  
// super.configure(http);//这行代码注释掉  
 http.formLogin()//表单登陆  
 .successForwardUrl("/welcome");//登陆成功之后跳转的页面路径  
 http.authorizeRequests()  
 .antMatchers("/hello/query").hasAuthority("hello:query")// /hello/query路径必须有hello:query权限  
 .antMatchers("/hello/add").hasAuthority("hello:add")// /hello/add路径必须有hello:add权限  
 .antMatchers("/hello/update").hasAuthority("hello:update")// /hello/update路径必须有hello:update权限  
 .antMatchers("/hello/delete").hasAuthority("hello:delete")// /hello/delete路径必须有hello:delete权限  
 .antMatchers("/admin/\*\*").hasRole("admin")// /admin/\*\*路径必须有admin的角色  
 .anyRequest().authenticated();//其它所有路径只要认证过了就可以访问  
 }

## 修改HelloController

@RequestMapping("admin/hello")  
public String admin(){  
 return "spring security -admin-hello";  
}  
@RequestMapping("hello/query")  
public String query(){  
 return "spring security -query";  
}  
@RequestMapping("hello/add")  
public String add(){  
 return "spring security -add";  
}  
@RequestMapping("hello/update")  
public String update(){  
 return "spring security -update";  
}  
@RequestMapping("hello/delete")  
public String delete(){  
 return "spring security -delete";  
}  
  
@RequestMapping("welcome")  
public String welcome(){  
 return "spring security -欢迎您";  
}

## 启动测试及结果【hello和getUserInfo路径除外】

### 使用admin/12345登陆访问结果

只有admin/hello路径可以访问，其它路径全部403

### 使用zhangsan/123456登陆访问结果

hello/query  hello/add hello/update  hello/delete可以访问 其它全部403

### 使用lisi/123456登陆访问结果

hello/query  hello/add 可以访问 其它全部403

### 使用wangwu/123456登陆访问结果

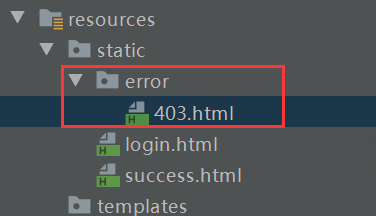
hello/query  hello/add hello/update 可以访问 其它全部403

### 使用zhouliu/123456登陆访问结果

hello/query hello/delete可以访问 其它全部403

## 403页面的配置

### 在resources/static/error下面创建403.html



<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <title>403页面</title>  
</head>  
<body>  
<h2>403:你没有权限访问此页面</h2>  
</body>  
</html>

### 使用lisi登陆，访问hello/update的效果



## 403返回JSON的配置

在上面的例子中，我们返回的是403页面，但是在开发中，如RestAPI风格的数据，是不能返回一个页面，而应该是给一个json如何配置呢

### 创建RestAuthenticationAccessDeniedHandler 类

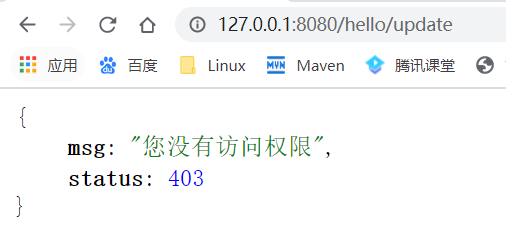
package com.bjsxt.config;  
  
import com.fasterxml.jackson.databind.ObjectMapper;  
import java.io.IOException;  
import java.io.PrintWriter;  
import java.util.HashMap;  
import java.util.Map;  
import javax.servlet.ServletException;  
import javax.servlet.http.HttpServletRequest;  
import javax.servlet.http.HttpServletResponse;  
import org.springframework.security.access.AccessDeniedException;  
import org.springframework.security.web.access.AccessDeniedHandler;  
  
*/\*\*  
 \* Created with IntelliJ IDEA.  
 \*  
 \* @Auther: 雷哥  
 \* @Date: 2020/08/01/17:27  
 \* @Description:  
 \*/*public class RestAuthenticationAccessDeniedHandler implements AccessDeniedHandler {  
  
 @Override  
 public void handle(HttpServletRequest request, HttpServletResponse response,  
 AccessDeniedException accessDeniedException) throws IOException, ServletException {  
 response.setContentType("application/json;charset=utf-8");  
 PrintWriter pw = response.getWriter();  
 Map<String, Object> map = new HashMap<>();  
 map.put("status", 403);  
 map.put("msg", "您没有访问权限");  
 pw.write(new ObjectMapper().writeValueAsString(map));  
 pw.flush();  
 pw.close();  
 }  
}

### 修改WebSecurityConfig类注册和配置RestAuthenticationAccessDeniedHandler

@Bean  
public AccessDeniedHandler getAccessDeniedHandler() {  
 return new RestAuthenticationAccessDeniedHandler();  
}



### 使用lisi登陆并访问 hello/update测试效果

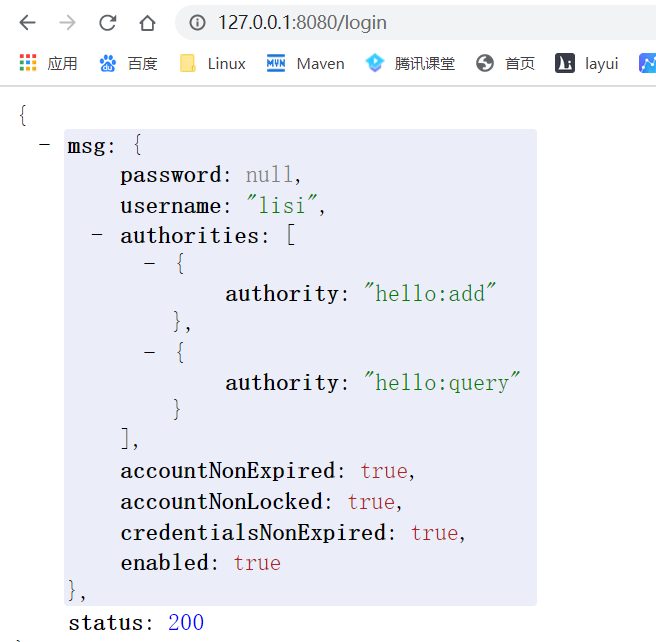


## 成功和失败返回json的配置

### 修改WebSecurityConfig类

@Override  
 protected void configure(HttpSecurity http) throws Exception {  
// super.configure(http);//这行代码注释掉  
 http.formLogin()//表单登陆  
// .successForwardUrl("/welcome")//登陆成功之后跳转的页面路径  
// .failureForwardUrl("/login/error");//自定义失败之后页面  
 .successHandler((request, response, authentication) -> { //使用这种配置在注释掉上面的  
 response.setContentType("application/json;charset=utf-8");  
 PrintWriter pw = response.getWriter();  
 Map<String, Object> map = new HashMap<>();  
 map.put("status", 200);  
 map.put("msg", authentication.getPrincipal());  
 pw.write(new ObjectMapper().writeValueAsString(map));  
 pw.flush();  
 pw.close();  
 })  
 .failureHandler((request, response, exception) -> {  
 response.setContentType("application/json;charset=utf-8");  
 PrintWriter pw = response.getWriter();  
 Map<String, Object> map = new HashMap<>();  
 map.put("status", 401);  
 if (exception instanceof LockedException) {  
 map.put("msg", "账户被锁定，登陆失败！");  
 } else if (exception instanceof BadCredentialsException) {  
 map.put("msg", "账户或者密码错误，登陆失败！");  
 } else if (exception instanceof DisabledException) {  
 map.put("msg", "账户被禁用，登陆失败！");  
 } else if (exception instanceof AccountExpiredException) {  
 map.put("msg", "账户已过期，登陆失败！");  
 } else if (exception instanceof CredentialsExpiredException) {  
 map.put("msg", "密码已过期，登陆失败！");  
 } else {  
 map.put("msg", "登陆失败！");  
 }  
 System.*out*.println(exception.getClass().getSimpleName());  
 pw.write(new ObjectMapper().writeValueAsString(map));  
 pw.flush();  
 pw.close();  
 });  
 http.authorizeRequests()  
 .antMatchers("/hello/query").hasAuthority("hello:query")// /hello/query路径必须有hello:query权限  
 .antMatchers("/hello/add").hasAuthority("hello:add")// /hello/add路径必须有hello:add权限  
 .antMatchers("/hello/update").hasAuthority("hello:update")// /hello/update路径必须有hello:update权限  
 .antMatchers("/hello/delete").hasAuthority("hello:delete")// /hello/delete路径必须有hello:delete权限  
 .antMatchers("/admin/\*\*").hasRole("admin")// /admin/\*\*路径必须有admin的角色  
 .anyRequest().authenticated();//其它所有路径只要认证过了就可以访问  
 }

### 使用lisi登陆效果



# 【掌握】Spring Security方法授权

## 相关注解说明

@PreAuthorize  在方法调用前进行权限检查

@PostAuthorize 在方法调用后进行权限检查

@Secured

上面的三个注解如果要使用的话必须加上

@EnableGlobalMethodSecurity(securedEnabled = true,prePostEnabled = true)

如果只使用PreAuthorize 就只用开启prePostEnabled = true

如果只使用@Secured 就只用开启securedEnabled = true 这种方式不推荐，有坑

        坑在这里 @Secured，而@Secured对应的角色必须要有ROLE\_前缀

## PreAuthorize和PostAuthorize测试

### 修改WebSecurityConfig

*/\*\*  
 \* @Auther: 雷哥  
 \* @Date: 2020/07/29/22:36  
 \* @Description:  
 \*/*@Configuration  
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {  
  
 */\*\*  
 \* 配置用户信息 模拟内存数据  
 \*/* @Override  
 protected void configure(AuthenticationManagerBuilder auth) throws Exception {  
 */\*\*  
 \* 重点说明  
 \* 在开发中，我们一般只针对权限，很少去使用角色  
 \*/* auth.inMemoryAuthentication()  
 .withUser(User.*withUsername*("admin").password(passwordEncoder.encode("123456"))  
 .roles("admin").build())//admin用户只有admin的角色  
 .withUser(User.*withUsername*("zhangsan").password(passwordEncoder.encode("123456"))  
 .authorities("hello:query", "hello:add","hello:update","hello:delete").build())  
 .withUser(User.*withUsername*("lisi").password(passwordEncoder.encode("123456"))  
 .authorities("hello:query", "hello:add").build())  
 .withUser(User.*withUsername*("wangwu").password(passwordEncoder.encode("123456"))  
 .authorities("hello:query", "hello:add", "hello:update").build())  
 .withUser(User.*withUsername*("zhaoliu").password(passwordEncoder.encode("123456"))  
 .authorities("hello:query", "hello:delete").build());  
 }  
  
 @Autowired  
 private PasswordEncoder passwordEncoder;  
  
 */\*\*  
 \* 从 Spring5 开始，强制要求密码要加密， 如果非不想加密，可 以使用一个过期的 PasswordEncoder 的实例 NoOpPasswordEncoder，  
 \* 但是不建议这么做，毕竟不安全。  
 \*/* @Bean  
 public PasswordEncoder passwordEncoder() {  
// return NoOpPasswordEncoder.getInstance();  
 return new BCryptPasswordEncoder();  
 }  
  
 @Bean  
 public AccessDeniedHandler getAccessDeniedHandler() {  
 return new RestAuthenticationAccessDeniedHandler();  
 }  
 @Override  
 protected void configure(HttpSecurity http) throws Exception {  
// super.configure(http);//这行代码注释掉  
 http.exceptionHandling().accessDeniedHandler(getAccessDeniedHandler());//使用处理器  
 http.formLogin()//表单登陆  
// .successForwardUrl("/welcome")//登陆成功之后跳转的页面路径  
// .failureForwardUrl("/login/error");//自定义失败之后页面  
 .successHandler((request, response, authentication) -> { //使用这种配置在注释掉上面的  
 response.setContentType("application/json;charset=utf-8");  
 PrintWriter pw = response.getWriter();  
 Map<String, Object> map = new HashMap<>();  
 map.put("status", 200);  
 map.put("msg", authentication.getPrincipal());  
 pw.write(new ObjectMapper().writeValueAsString(map));  
 pw.flush();  
 pw.close();  
 })  
 .failureHandler((request, response, exception) -> {  
 response.setContentType("application/json;charset=utf-8");  
 PrintWriter pw = response.getWriter();  
 Map<String, Object> map = new HashMap<>();  
 map.put("status", 401);  
 if (exception instanceof LockedException) {  
 map.put("msg", "账户被锁定，登陆失败！");  
 } else if (exception instanceof BadCredentialsException) {  
 map.put("msg", "账户或者密码错误，登陆失败！");  
 } else if (exception instanceof DisabledException) {  
 map.put("msg", "账户被禁用，登陆失败！");  
 } else if (exception instanceof AccountExpiredException) {  
 map.put("msg", "账户已过期，登陆失败！");  
 } else if (exception instanceof CredentialsExpiredException) {  
 map.put("msg", "密码已过期，登陆失败！");  
 } else {  
 map.put("msg", "登陆失败！");  
 }  
 System.*out*.println(exception.getClass().getSimpleName());  
 pw.write(new ObjectMapper().writeValueAsString(map));  
 pw.flush();  
 pw.close();  
 });  
 http.authorizeRequests()  
// .antMatchers("/hello/query").hasAuthority("hello:query")// /hello/query路径必须有hello:query权限  
// .antMatchers("/hello/add").hasAuthority("hello:add")// /hello/add路径必须有hello:add权限  
// .antMatchers("/hello/update").hasAuthority("hello:update")// /hello/update路径必须有hello:update权限  
// .antMatchers("/hello/delete").hasAuthority("hello:delete")// /hello/delete路径必须有hello:delete权限  
// .antMatchers("/admin/\*\*").hasRole("admin")// /admin/\*\*路径必须有admin的角色  
 .anyRequest().authenticated();//其它所有路径只要认证过了就可以访问  
 }  
}

### 修改启动类

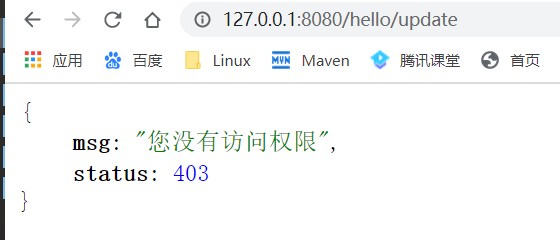
@SpringBootApplication  
//@EnableWebSecurity //启用security  
@EnableGlobalMethodSecurity(securedEnabled = true,prePostEnabled = true)  
public class SpringSecurityDemoApplication {  
  
 public static void main(String[] args) {  
 SpringApplication.*run*(SpringSecurityDemoApplication.class, args);  
 }  
}

### 修改HelloController

@PreAuthorize("hasAnyRole('admin')")  
@RequestMapping("admin/hello")  
public String admin(){  
 return "spring security -admin-hello";  
}  
@PreAuthorize("hasAuthority('hello:query')")  
@RequestMapping("hello/query")  
public String query(){  
 return "spring security -query";  
}  
@PreAuthorize("hasAuthority('hello:add')")  
@RequestMapping("hello/add")  
public String add(){  
 return "spring security -add";  
}  
@PreAuthorize("hasAuthority('hello:update')")  
@RequestMapping("hello/update")  
public String update(){  
 System.*out*.println("ok");  
 return "spring security -update";  
}  
@PreAuthorize("hasAuthority('hello:delete')")  
@RequestMapping("hello/delete")  
public String delete(){  
 return "spring security -delete";  
}

### 测试

使用lisi/123456登陆访问hello/update



## Secured测试【了解，基本不用这东西】

### 修改HelloController

// @Secured("admin")对函数访问进行控制，有一个坑(不加ROLE\_)，这个时候，  
// AccessDecissionManager会选择RoleVoter进行vote，但是RoleVoter默认的rolePrefix是“ROLE\_”。  
 @Secured("ROLE\_admin") //正确的配置如左，后台我们给的是admin但是判断这里要写ROLE\_admin  
 @RequestMapping("admin/hello")  
 public String admin(){  
 return "spring security -admin-hello";  
 }

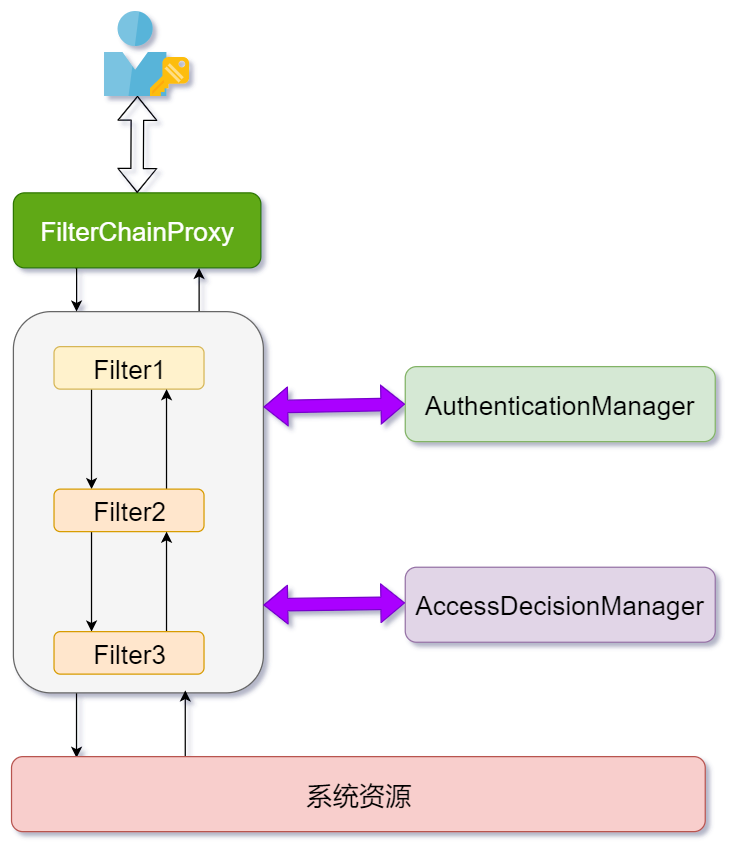
### 测试

使用admin/123456登陆 正常判断，如果配置的是@Secured("admin")那么会报没有权限的问题

# 【源码分析】Spring Security认证授权总揽

## 结构总揽

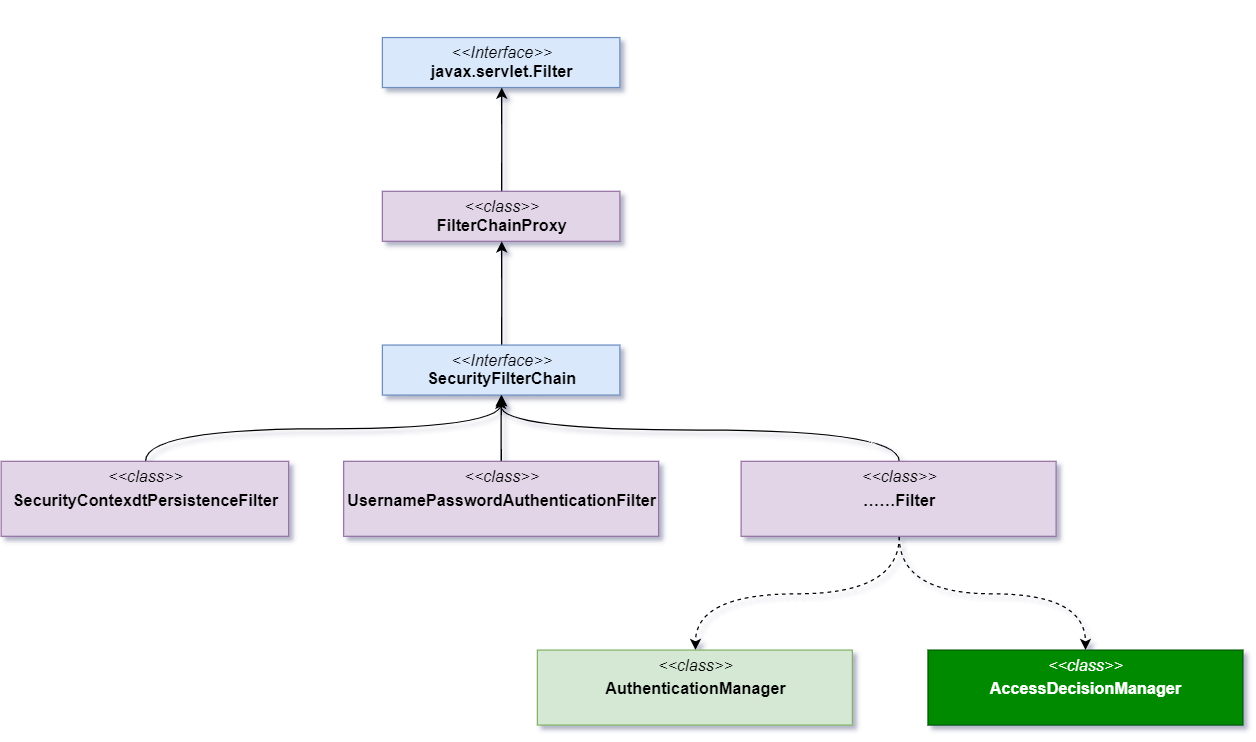
    Spring Security所解决的问题就是安全访问控制，而安全访问控制功能其实就是对所有进入系统的请求进行拦截，校验每个请求是否能够访问它所期望的资源。根据前边知识的学习，可以通过Filter或AOP等技术来实现，SpringSecurity对Web资源的保护是靠Filter实现的，所以从这个Filter来入手，逐步深入Spring Security原理。当初始化Spring Security时，会创建一个名为 SpringSecurityFilterChain 的Servlet过滤器，类型为org.springframework.security.web.FilterChainProxy，它实现了javax.servlet.Filter，因此外部的请求会经过此类，下图是Spring Security过虑器链结构图：



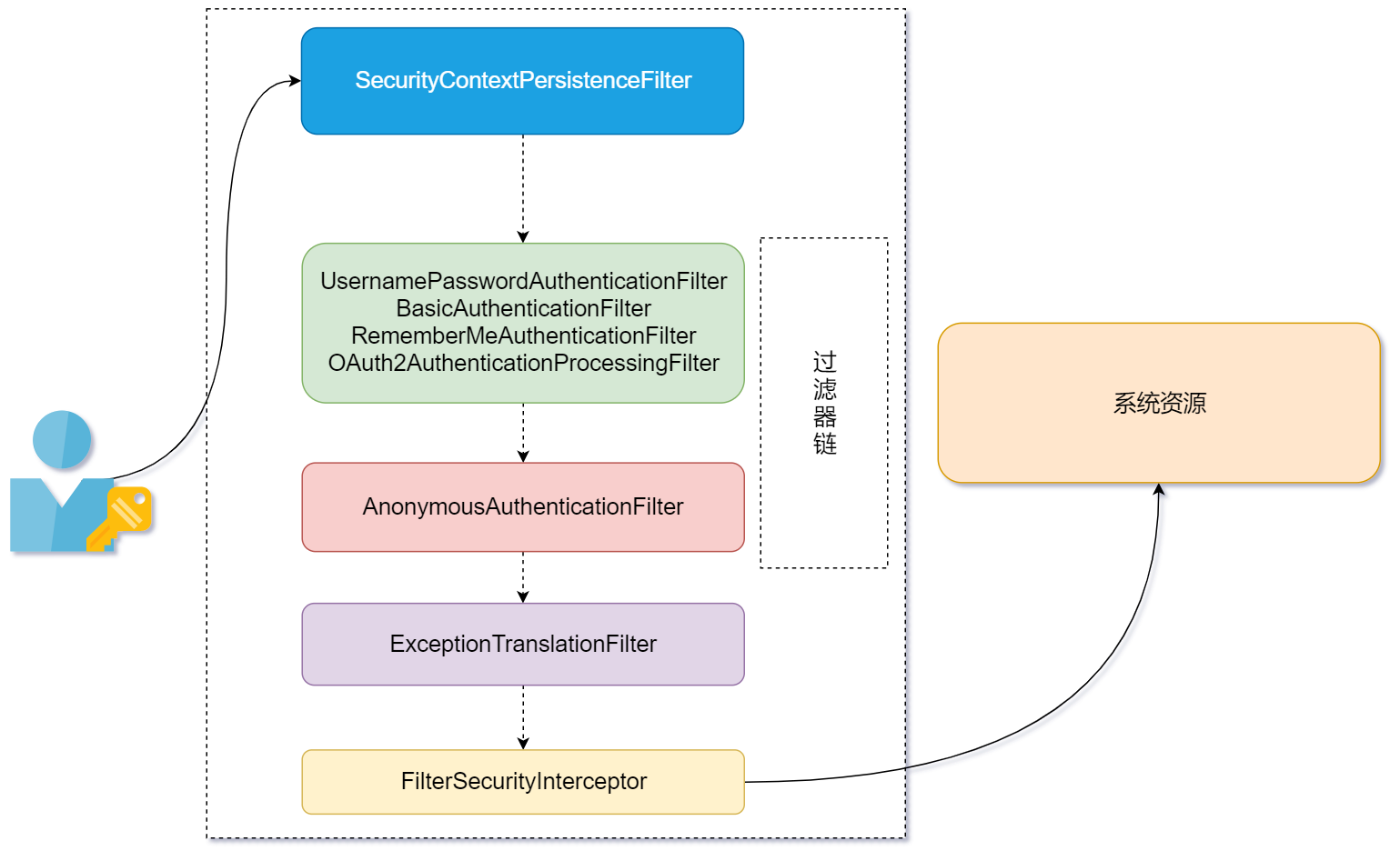
## 上图说明

FilterChainProxy 是一个代理，真正起作用的是FilterChainProxy中SecurityFilterChain所包含的各个Filter，同时这些Filter作为Bean被Spring管理，它们是Spring Security核心，各有各的职责，但他们并不直接处理用户的认证，也不直接处理用户的授权，而是把它们交给了认证管理器（AuthenticationManager）和决策管理器（AccessDecisionManager）进行处理

下图是FilterChainProxy相关类的UML图示



spring Security功能的实现主要是由一系列过滤器链相互配合完成。



## 过滤器链中主要的几个过滤器及其作用

### SecurityContextPersistenceFilter

    这个Filter是整个拦截过程的入口和出口（也就是第一个和最后一个拦截器），会在请求开始时从配置好的 SecurityContextRepository 中获取 SecurityContext，然后把它设置给SecurityContextHolder。在请求完成后将 SecurityContextHolder 持有的 SecurityContext 再保存到配置好的 SecurityContextRepository，同时清除 securityContextHolder 所持有的 SecurityContext；

### UsernamePasswordAuthenticationFilter

    用于处理来自表单提交的认证。该表单必须提供对应的用户名和密码，其内部还有登录成功或失败后进行处理的 AuthenticationSuccessHandler 和AuthenticationFailureHandler，这些都可以根据需求做相关改变；

### FilterSecurityInterceptor

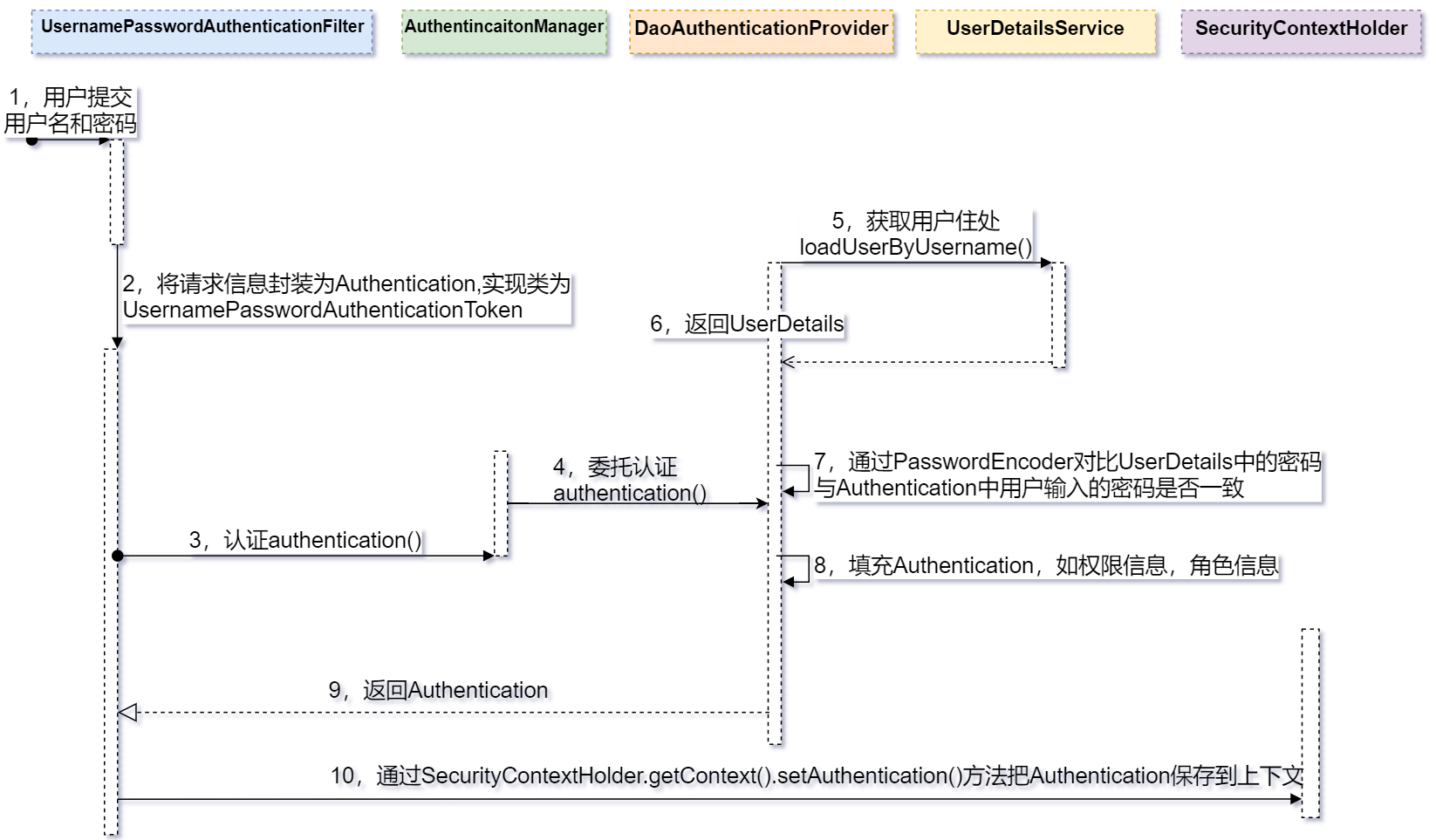
    是用于保护web资源的，使用AccessDecisionManager对当前用户进行授权访问；

### ExceptionTranslationFilter

    能够捕获来自 FilterChain 所有的异常，并进行处理。但是它只会处理两类异常：AuthenticationException 和 AccessDeniedException，其它的异常它会继续抛出。

# 【源码分析】Spring Security认证工作流程

## 认证流程图



## 流程图分析

1. 用户提交用户名、密码被SecurityFilterChain中的 UsernamePasswordAuthenticationFilter 过滤器获取到，封装为请求Authentication，通常情况下是UsernamePasswordAuthenticationToken这个实现类。
2. 然后过滤器将Authentication提交至认证管理器（AuthenticationManager）进行认证
3. 认证成功后， AuthenticationManager 身份管理器返回一个被填充满了信息的（包括上面提到的权限信息，身份信息，细节信息，但密码通常会被移除） Authentication 实例。
4. SecurityContextHolder 安全上下文容器将第3步填充了信息的 Authentication ，通过SecurityContextHolder.getContext().setAuthentication(…)方法，设置到其中。可以看出AuthenticationManager接口（认证管理器）是认证相关的核心接口，也是发起认证的出发点，它的实现类为ProviderManager。而Spring Security支持多种认证方式，因此ProviderManager维护着一个List<AuthenticationProvider> 列表，存放多种认证方式，最终实际的认证工作是由AuthenticationProvider完成的。咱们知道web表单的对应的AuthenticationProvider实现类为DaoAuthenticationProvider，它的内部又维护着一个UserDetailsService负责UserDetails的获取。最终AuthenticationProvider将UserDetails填充至Authentication

## 断点调试及源码分析

看上图打断点调试

## 结果总结

### AuthenticationProvider

通过前面的Spring Security认证流程我们得知，认证管理器（AuthenticationManager）委托AuthenticationProvider完成认证工作。

AuthenticationProvider是一个接口，定义如下：

public interface AuthenticationProvider {  
 */\*\*  
 \* 认证  
 \*/* Authentication authenticate(Authentication authentication)  
 throws AuthenticationException;  
  
 */\*\*  
 \* 判断认证方式  
 \*/* boolean supports(Class<?> authentication);  
}

    authenticate()方法定义了认证的实现过程，它的参数是一个Authentication，里面包含了登录用户所提交的用户、密码等。而返回值也是一个Authentication，这个Authentication则是在认证成功后，将用户的权限及其他信息重新组装后生成。

    Spring Security中维护着一个 List<AuthenticationProvider> 列表，存放多种认证方式，不同的认证方式使用不同的AuthenticationProvider。如使用用户名密码登录时，使用AuthenticationProvider1，短信登录时使用AuthenticationProvider2等等这样的例子很多。

    每个AuthenticationProvider需要实现supports（）方法来表明自己支持的认证方式，如我们使用表单方式认证，在提交请求时Spring Security会生成UsernamePasswordAuthenticationToken，它是一个Authentication，里面封装着用户提交的用户名、密码信息。而对应的，哪个AuthenticationProvider来处理它？

我们在DaoAuthenticationProvider的基类AbstractUserDetailsAuthenticationProvider发现以下代码：

**public boolean supports(Class<?> authentication) {**

**return UsernamePasswordAuthenticationToken.class.isAssignableFrom(authentication);**

**}**

也就是说当web表单提交用户名密码时，Spring Security由DaoAuthenticationProvider处理。

最后，我们来看一下 Authentication(认证信息)的结构，它是一个接口，我们之前提到的

UsernamePasswordAuthenticationToken就是它的实现之一：

**public interface Authentication extends Principal, Serializable {**

**Collection<? extends GrantedAuthority> getAuthorities();**

**Object getCredentials();**

**Object getDetails();**

**Object getPrincipal();**

**boolean isAuthenticated();**

**void setAuthenticated(boolean var1) throws IllegalArgumentException;**

**}**

（1）Authentication是spring security包中的接口，直接继承自Principal类，而Principal是位于 java.security

包中的。它是表示着一个抽象主体身份，任何主体都有一个名称，因此包含一个getName()方法。

（2）getAuthorities()，权限信息列表，默认是GrantedAuthority接口的一些实现类，通常是代表权限信息的一系

列字符串。

（3）getCredentials()，凭证信息，用户输入的密码字符串，在认证过后通常会被移除，用于保障安全。

（4）getDetails()，细节信息，web应用中的实现接口通常为 WebAuthenticationDetails，它记录了访问者的ip地

址和sessionId的值。

（5）getPrincipal()，身份信息，大部分情况下返回的是UserDetails接口的实现类，UserDetails代表用户的详细

信息，那从Authentication中取出来的UserDetails就是当前登录用户信息，它也是框架中的常用接口之一。

### UserDetailsService

#### 认识UserDetailsService

现在咱们现在知道DaoAuthenticationProvider处理了web表单的认证逻辑，认证成功后既得到一个Authentication(UsernamePasswordAuthenticationToken实现)，里面包含了身份信息（Principal）。这个身份信息就是一个 Object ，大多数情况下它可以被强转为UserDetails对象。

DaoAuthenticationProvider中包含了一个UserDetailsService实例，它负责根据用户名提取用户信息UserDetails(包含密码)，而后DaoAuthenticationProvider会去对比UserDetailsService提取的用户密码与用户提交的密码是否匹配作为认证成功的关键依据，因此可以通过将自定义的 UserDetailsService 公开为spring bean来定

义自定义身份验证。

**public interface UserDetailsService {**

**UserDetails loadUserByUsername(String username) throws UsernameNotFoundException;**

**}**

很多人把 DaoAuthenticationProvider和UserDetailsService的职责搞混淆，其实UserDetailsService只负责从特定的地方（通常是数据库）加载用户信息，仅此而已。而DaoAuthenticationProvider的职责更大，它完成完整的认证流程，同时会把UserDetails填充至Authentication。

上面一直提到UserDetails是用户信息，咱们看一下它的真面目：

**public interface UserDetails extends Serializable {**

**Collection<? extends GrantedAuthority> getAuthorities();**

**String getPassword();**

**String getUsername();**

**boolean isAccountNonExpired();**

**boolean isAccountNonLocked();**

**boolean isCredentialsNonExpired();**

**boolean isEnabled();**

**}**

它和Authentication接口很类似，比如它们都拥有username，authorities。Authentication的getCredentials()与UserDetails中的getPassword()需要被区分对待，前者是用户提交的密码凭证，后者是用户实际存储的密码，认证其实就是对这两者的比对。Authentication中的getAuthorities()实际是由UserDetails的getAuthorities()传递而形成的。还记得Authentication接口中的getDetails()方法吗？其中的UserDetails用户详细信息便是经过了AuthenticationProvider认证之后被填充的。

通过实现UserDetailsService和UserDetails，我们可以完成对用户信息获取方式以及用户信息字段的扩展。

Spring Security提供的InMemoryUserDetailsManager(内存认证)，JdbcUserDetailsManager(jdbc认证)就是

UserDetailsService的实现类，主要区别无非就是从内存还是从数据库加载用户。

#### 测试

自定义UserDetailsService

**@Service**

**public class SpringDataUserDetailsService implements UserDetailsService {**

**@Override**

**public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {**

**//登录账号**

**System.out.println("username="+username);**

**//根据账号去数据库查询...**

**//这里暂时使用静态数据**

**UserDetails userDetails =**

**User.withUsername(username).password("123456").authorities("hello:query").build();**

**return userDetails;**

**}**

**}**

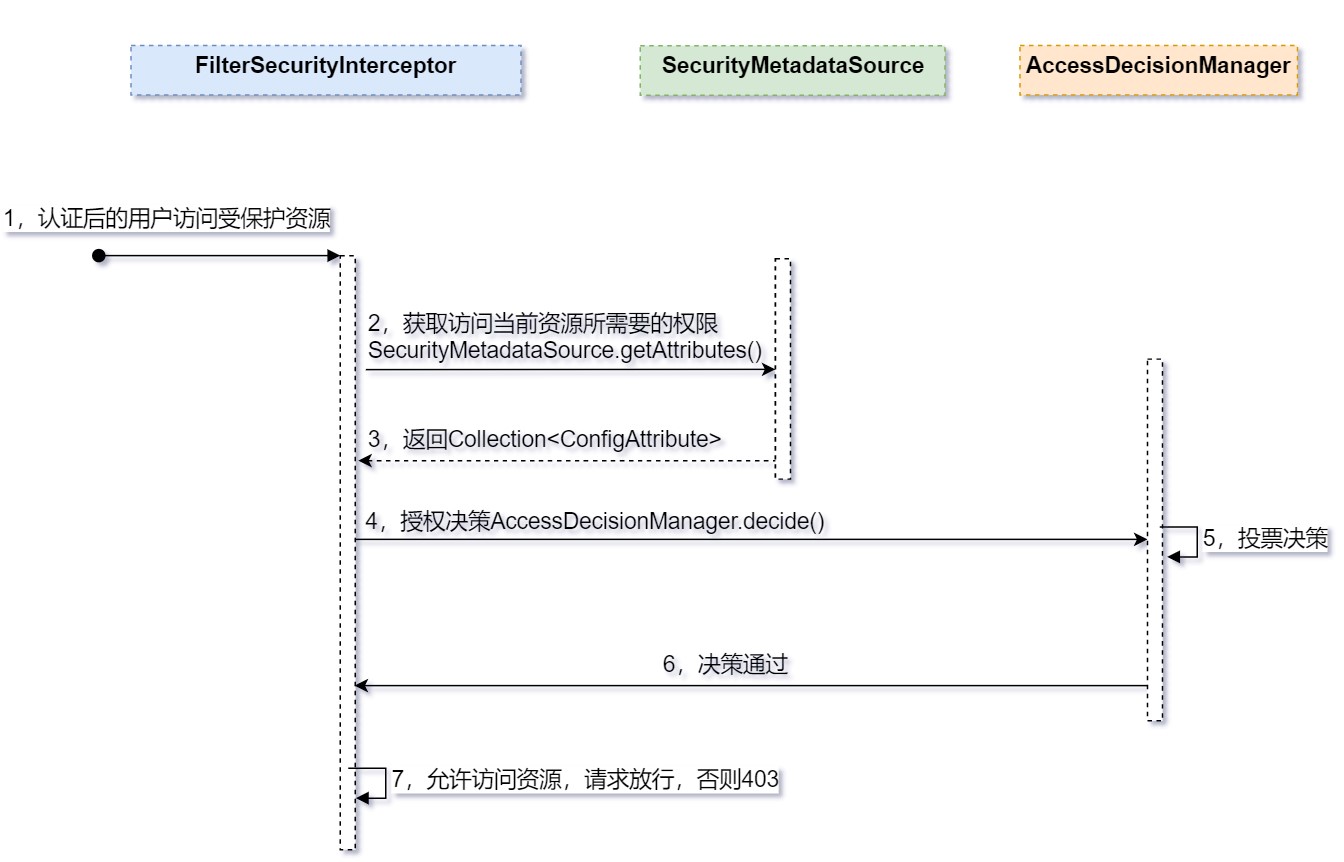
重启工程，请求认证，SpringDataUserDetailsService的loadUserByUsername方法被调用 ，查询用户信息。

# 【源码分析】Spring Security授权工作流程

## 授权流程图

通过快速上手我们知道，Spring Security可以通过 http.authorizeRequests() 对web请求进行授权保护。SpringSecurity使用标准Filter建立了对web请求的拦截，最终实现对资源的授权访问。

Spring Security的授权流程如下：



## 授权流程分析

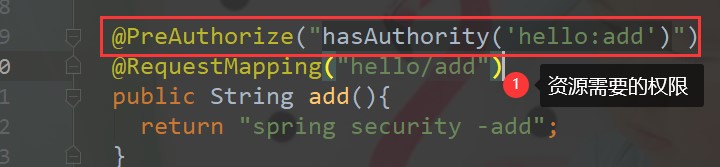
### 拦截请求

已认证用户访问受保护的web资源将被SecurityFilterChain中的 FilterSecurityInterceptor 的子类拦截。

### 获取资源访问策略

FilterSecurityInterceptor会从 SecurityMetadataSource 的子类DefaultFilterInvocationSecurityMetadataSource 获取要访问当前资源所需要的权限Collection<ConfigAttribute> 。

SecurityMetadataSource其实就是读取访问策略的抽象，而读取的内容，其实就是我们配置的访问规则， 读取访问策略如：



### 最后

    FilterSecurityInterceptor会调用 AccessDecisionManager 进行授权决策，若决策通过，则允许访问资源，否则将禁止访问。

AccessDecisionManager（访问决策管理器）的核心接口如下:

public interface AccessDecisionManager {  
 // ~ Methods  
 // ========================================================================================================  
  
 */\*\*  
 \* 通过传递的参数来决定用户是否有访问对应受保护资源的权限  
 \*  
 \* @param authentication the caller invoking the method (not null)  
 \* @param object the secured object being called  
 \* @param configAttributes the configuration attributes associated with the secured  
 \* object being invoked  
 \*/* void decide(Authentication authentication, Object object,  
 Collection<ConfigAttribute> configAttributes) throws AccessDeniedException,  
 InsufficientAuthenticationException;  
}

这里着重说明一下decide的参数：

authentication：要访问资源的访问者的身份

object：要访问的受保护资源，web请求对应FilterInvocation

configAttributes：是受保护资源的访问策略，通过SecurityMetadataSource获取。

decide接口就是用来鉴定当前用户是否有访问对应受保护资源的权限。

## 授权决策分析

AccessDecisionManager采用投票的方式来确定是否能够访问受保护资源。

AccessDecisionManager中包含的一系列AccessDecisionVoter将会被用来对Authentication是否有权访问受保护对象进行投票，AccessDecisionManager根据投票结果，做出最终决策。

AccessDecisionVoter是一个接口，其中定义有三个方法，具体结构如下所示。

**public interface AccessDecisionVoter<S> {**

**int ACCESS\_GRANTED = 1;**

**int ACCESS\_ABSTAIN = 0;**

**int ACCESS\_DENIED = ‐1;**

**boolean supports(ConfigAttribute var1);**

**boolean supports(Class<?> var1);**

**int vote(Authentication var1, S var2, Collection<ConfigAttribute> var3);**

**}**

vote()方法的返回结果会是AccessDecisionVoter中定义的三个常量之一。ACCESS\_GRANTED表示同意，ACCESS\_DENIED表示拒绝，ACCESS\_ABSTAIN表示弃权。如果一个AccessDecisionVoter不能判定当前Authentication是否拥有访问对应受保护对象的权限，则其vote()方法的返回值应当为弃权ACCESS\_ABSTAIN。

      Spring Security内置了三个基于投票的AccessDecisionManager实现类如下，它们分别是AffirmativeBased、ConsensusBased和UnanimousBased，。

### AffirmativeBased的逻辑是：

 （1）只要有AccessDecisionVoter的投票为ACCESS\_GRANTED则同意用户进行访问；

 （2）如果全部弃权也表示通过；

 （3）如果没有一个人投赞成票，但是有人投反对票，则将抛出AccessDeniedException。

**Spring security默认使用的是AffirmativeBased。**

### ConsensusBased的逻辑是：

  （1）如果赞成票多于反对票则表示通过。

  （2）反过来，如果反对票多于赞成票则将抛出AccessDeniedException。

  （3）如果赞成票与反对票相同且不等于0，并且属性allowIfEqualGrantedDeniedDecisions的值为true，则表示通过，否则将抛出异常AccessDeniedException。参数allowIfEqualGrantedDeniedDecisions的值默认为true。

  （4）如果所有的AccessDecisionVoter都弃权了，则将视参数allowIfAllAbstainDecisions的值而定，如果该值为true则表示通过，否则将抛出异常AccessDeniedException。参数allowIfAllAbstainDecisions的值默认为false。

### UnanimousBased的逻辑具体是：

UnanimousBased的逻辑与另外两种实现有点不一样，另外两种会一次性把受保护对象的配置属性全部传递给AccessDecisionVoter进行投票，而UnanimousBased会一次只传递一个ConfigAttribute给AccessDecisionVoter进行投票。这也就意味着如果我们的AccessDecisionVoter的逻辑是只要传递进来的

ConfigAttribute中有一个能够匹配则投赞成票，但是放到UnanimousBased中其投票结果就不一定是赞成了。

**UnanimousBased的逻辑具体来说是这样的：**

  （1）如果受保护对象配置的某一个ConfigAttribute被任意的AccessDecisionVoter反对了，则将抛出AccessDeniedException。

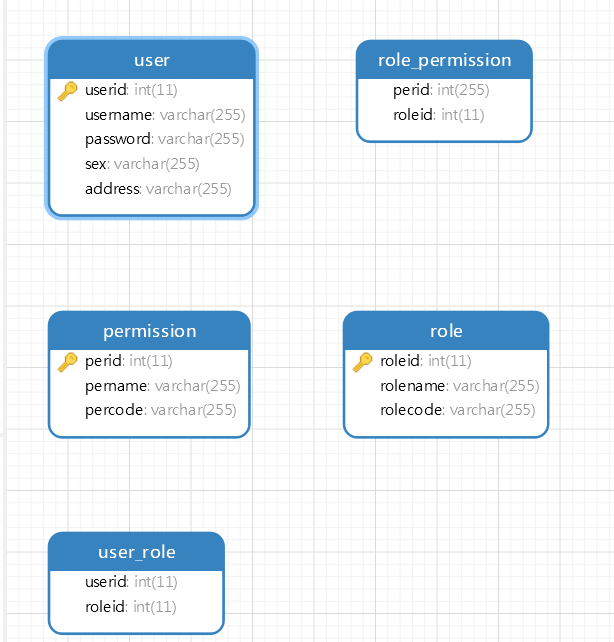
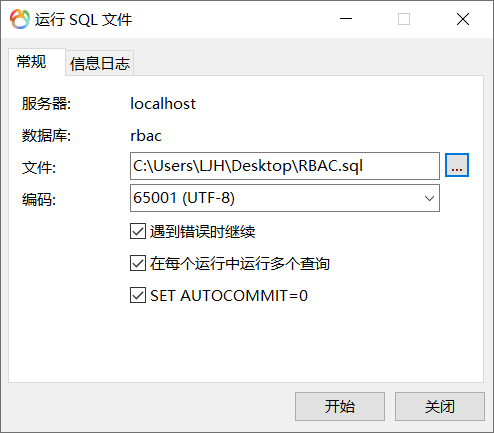
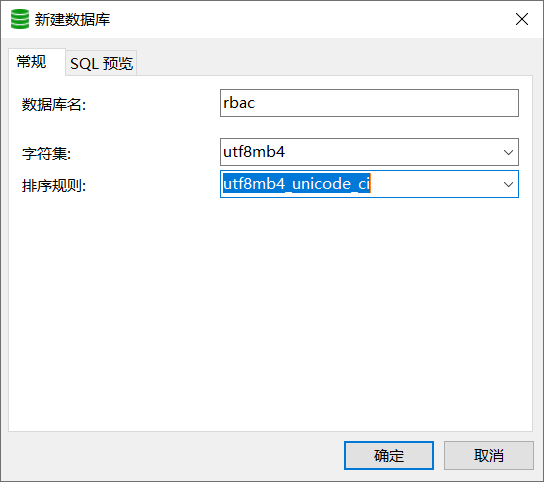
  （2）如果没有反对票，但是有赞成票，则表示通过。

  （3）如果全部弃权了，则将视参数allowIfAllAbstainDecisions的值而定，true则通过，false则抛出AccessDeniedException。

Spring Security也内置一些投票者实现类如RoleVoter、AuthenticatedVoter和WebExpressionVoter等，可以自行查阅资料进行学习。

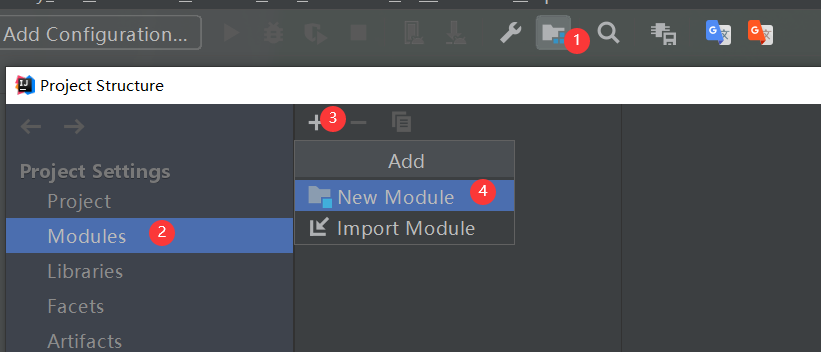
# 【实战】Spring Security集成Thymeleaf详解【1】

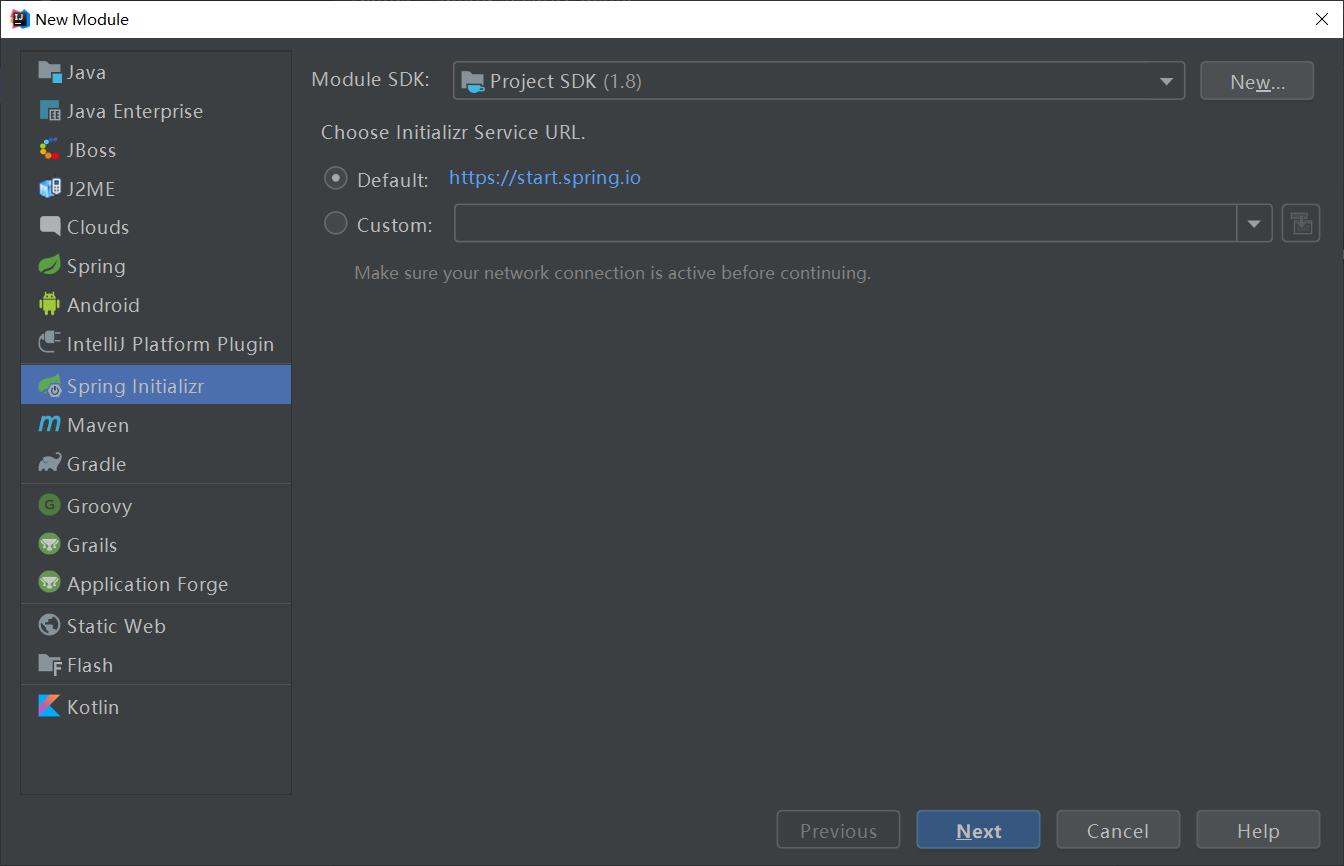
## 准备数据库

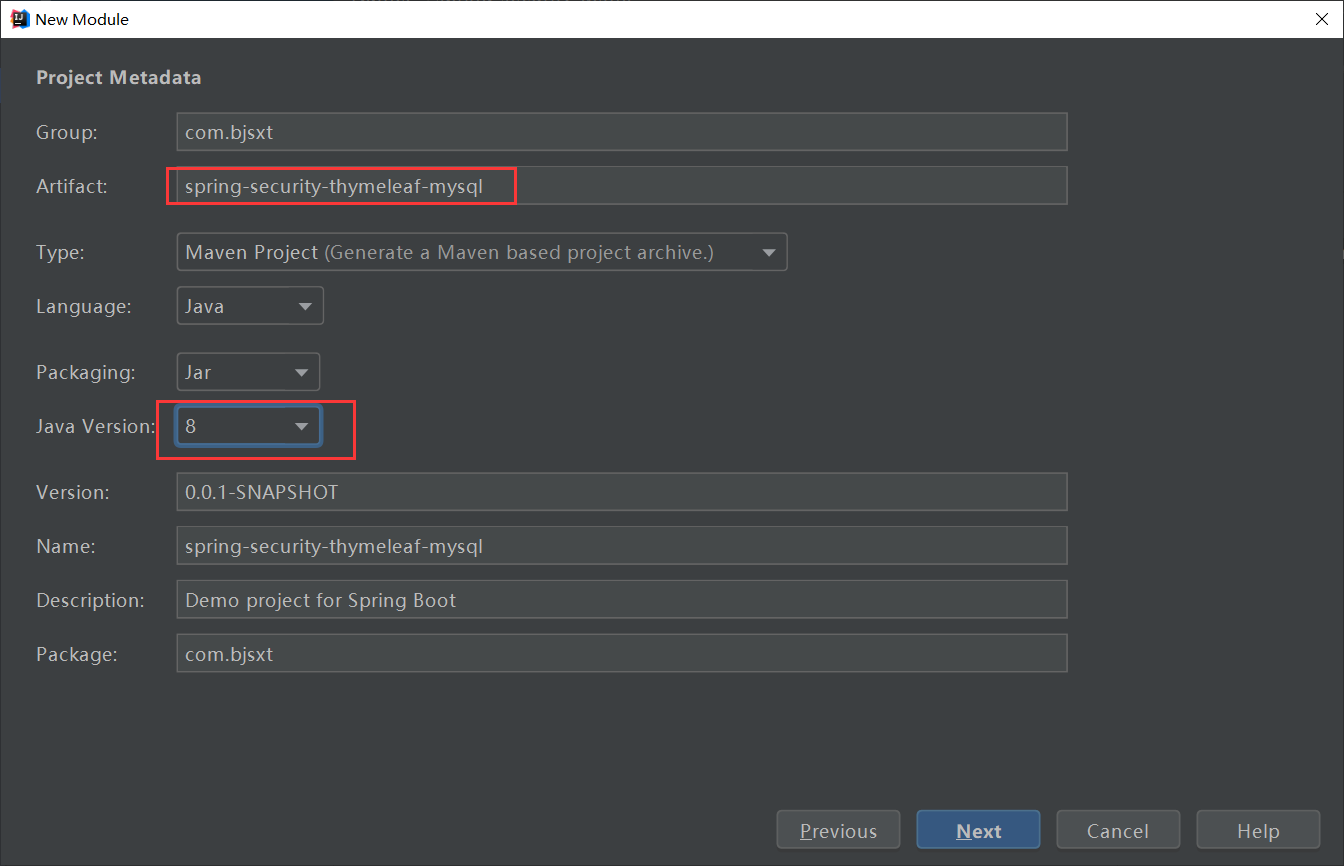


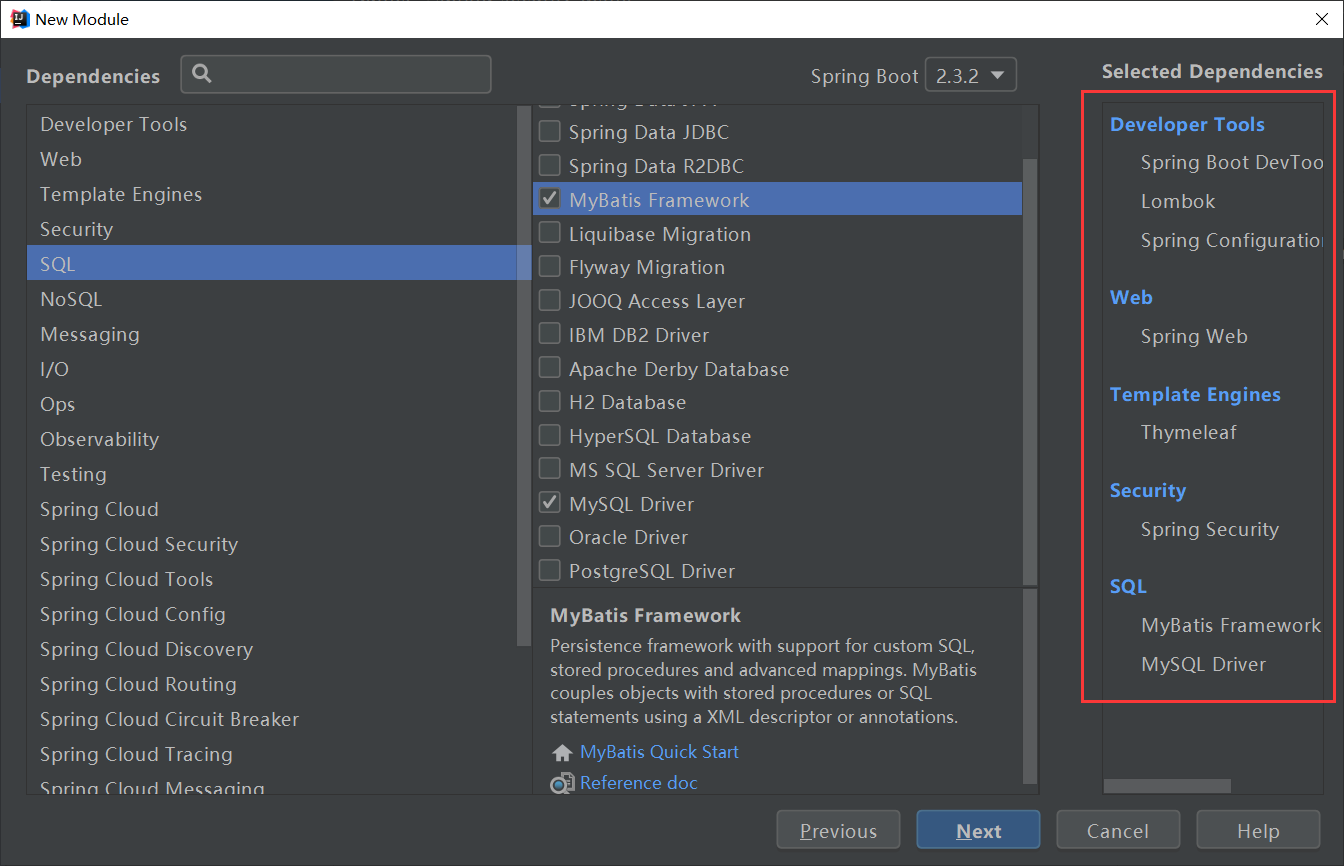
## 创建项目spring-security-thymeleaf-mysql

### 创建项目









### 修改pom.xml

<?xml version="1.0" encoding="UTF-8"?>  
<project xmlns="http://maven.apache.org/POM/4.0.0"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">  
 <modelVersion>4.0.0</modelVersion>  
 <parent>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-parent</artifactId>  
 <version>2.3.2.RELEASE</version>  
 <relativePath/> <!-- lookup parent from repository -->  
 </parent>  
 <groupId>com.bjsxt</groupId>  
 <artifactId>spring-security-thymeleaf-mysql</artifactId>  
 <version>0.0.1-SNAPSHOT</version>  
 <name>spring-security-thymeleaf-mysql</name>  
 <description>Demo project for Spring Boot</description>  
  
 <properties>  
 <java.version>1.8</java.version>  
 </properties>  
  
 <dependencies>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-security</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-thymeleaf</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-web</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>org.mybatis.spring.boot</groupId>  
 <artifactId>mybatis-spring-boot-starter</artifactId>  
 <version>2.1.3</version>  
 </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.springframework.boot</groupId>  
 <artifactId>spring-boot-configuration-processor</artifactId>  
 <optional>true</optional>  
 </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>  
 <exclusions>  
 <exclusion>  
 <groupId>org.junit.vintage</groupId>  
 <artifactId>junit-vintage-engine</artifactId>  
 </exclusion>  
 </exclusions>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.security</groupId>  
 <artifactId>spring-security-test</artifactId>  
 <scope>test</scope>  
 </dependency>  
 </dependencies>  
  
 <build>  
 <plugins>  
 <plugin>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-maven-plugin</artifactId>  
 </plugin>  
 </plugins>  
 </build>  
</project>

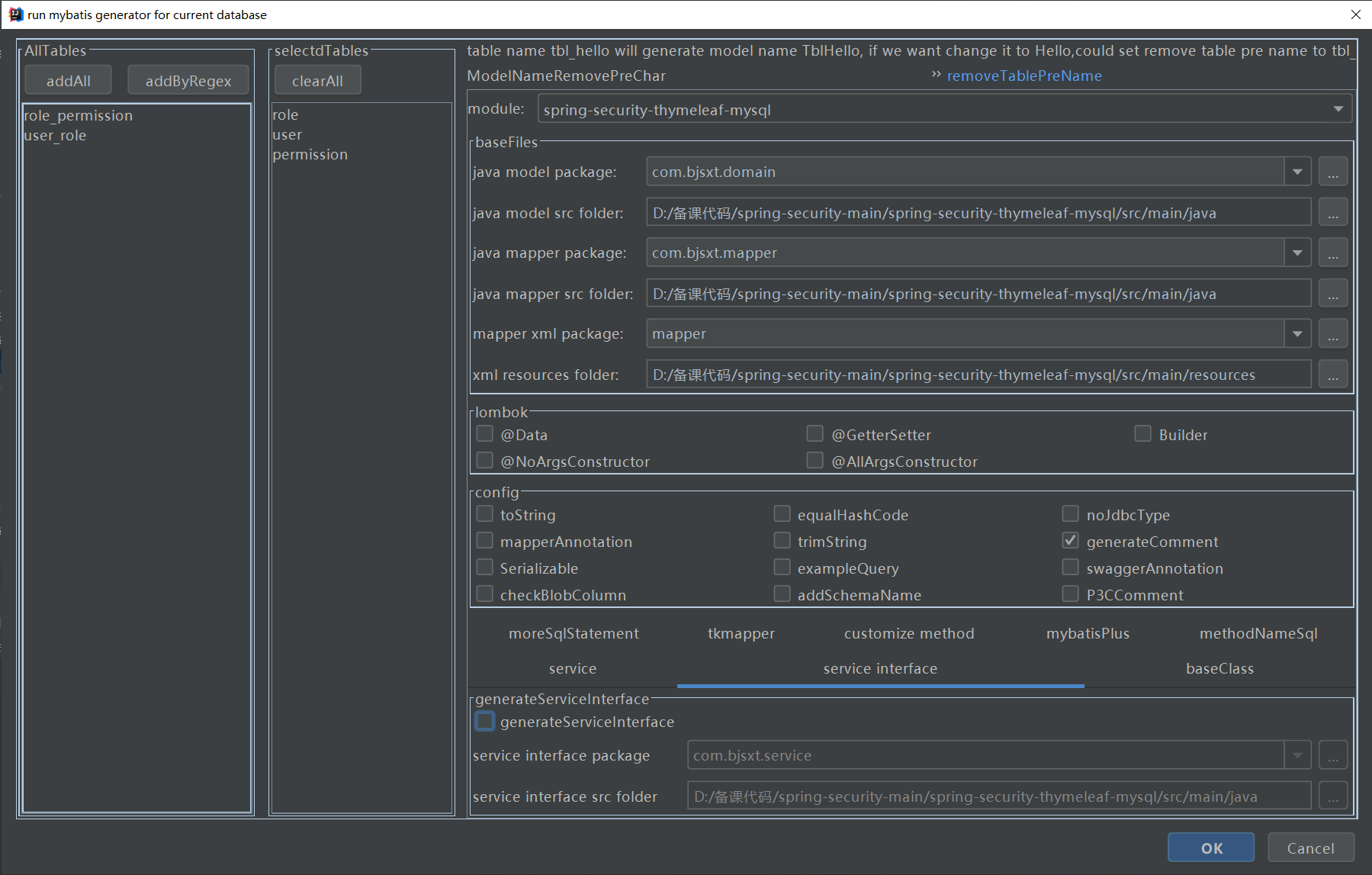
### 修改application.yaml

*#端口*server:  
 port: 8080  
*#数据源的配置*spring:  
 datasource:  
 driver-class-name: com.mysql.cj.jdbc.Driver  
 url: jdbc:mysql://127.0.0.1:3306/RBAC?useUnicode=true&characterEncoding=UTF-8&serverTimezone=UTC  
 username: root  
 password: 123456  
 type: org.springframework.jdbc.datasource.DriverManagerDataSource  
mybatis:  
 configuration:  
 log-impl: org.apache.ibatis.logging.stdout.StdOutImpl  
 mapper-locations: classpath:mapper/\*.xml

### 生成User UserMapper UserMapper.xml

### 生成Role RoleMapper RoleMapper.xml

### 生成Permission PermissionMapper PermissionMapper.xml



## 创建登陆页面templates/login.html

<!DOCTYPE html>  
<html>  
<head>  
 <meta charset="UTF-8">  
 <title>用户登陆</title>  
</head>  
<body>  
<h2>登录页面</h2>  
<form action="/login/doLogin" method="post">  
 <table>  
 <tr>  
 <td>用户名:</td>  
 <td><input type="text" name="username" value="zhangsan"></td>  
 </tr>  
 <tr>  
 <td>密码:</td>  
 <td><input type="password" name="password"></td>  
 </tr>  
 <tr>  
 <td colspan="2"><button type="submit">登录</button></td>  
 </tr>  
 </table>  
</form>  
</body>

## 建登陆页面templates/index.html

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <title>系统首页</title>  
</head>  
<body>  
 <h1 align="center">系统首页</h1>  
 <a href="/user/query">查询用户</a>  
 <br>  
 <a href="/user/add">添加用户</a>  
 <br>  
 <a href="/user/update">更新用户</a>  
 <br>  
 <a href="/user/delete">删除用户</a>  
 <br>  
 <a href="/user/export">导出用户</a>  
 <br>  
 <br><br><br>  
 <h2><a href="/login/logout">退出</a></h2>  
 <br>  
</body>  
</html>

## 建403页面static/error/403.html

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <title>403页面</title>  
</head>  
<body>  
<h2>403:你没有权限访问此页面</h2>  
<a href="/index">返回</a>  
</body>  
</html>

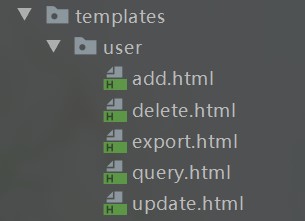
## 建500页面static/error/404.html

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <title>404页面</title>  
</head>  
<body>  
<h2>404:你访问的页面不存在</h2>  
<a href="/index">返回</a>  
</body>  
</html>

## 建500页面static/error/500.html

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <title>500页面</title>  
</head>  
<body>  
<h2>500:服务器内部出现异常，请稍后再试</h2>  
<a href="/index">返回</a>  
</body>  
</html>

## 创建templates/query.html、add.html、update.html、delete.html、export.html



### query.html

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <title>系统首页-用户管理-查询</title>  
</head>  
<body>  
<h1 align="center">系统首页-用户管理-查询</h1>  
<a href="/index">返回</a>  
<br>  
</body>  
</html>

### add.html

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <title>系统首页-用户管理-添加</title>  
</head>  
<body>  
<h1 align="center">系统首页-用户管理-添加</h1>  
<a href="/index">返回</a>  
<br>  
</body>  
</html>

### update.html

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <title>系统首页-用户管理-更新</title>  
</head>  
<body>  
<h1 align="center">系统首页-用户管理-更新</h1>  
<a href="/index">返回</a>  
<br>  
</body>  
</html>

### delete.html

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <title>系统首页-用户管理-删除</title>  
</head>  
<body>  
<h1 align="center">系统首页-用户管理-删除</h1>  
<a href="/index">返回</a>  
<br>  
</body>  
</html>

### export.html

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <title>系统首页-用户管理-导出</title>  
</head>  
<body>  
<h1 align="center">系统首页-用户管理-导出</h1>  
<a href="/index">返回</a>  
<br>  
</body>  
</html>

## 创建WebConfig

package com.bjsxt.config;  
  
import org.springframework.context.annotation.Configuration;  
import org.springframework.web.servlet.config.annotation.ViewControllerRegistry;  
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;  
  
*/\*\*  
 \* Created with IntelliJ IDEA.  
 \*  
 \* @Auther: 雷哥  
 \* @Date: 2020/08/03/22:46  
 \* @Description:  
 \*/*@Configuration  
public class WebConfig implements WebMvcConfigurer {  
 @Override  
 public void addViewControllers(ViewControllerRegistry registry) {  
 registry.addViewController("/").setViewName("/login"); //当访问/跳转到登陆页面  
 registry.addViewController("/login/toLogin").setViewName("/login");//当访问/login跳转到登陆页面  
 }  
}

## 创建WebSecurityConfig

package com.bjsxt.config;  
  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
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.builders.WebSecurity;  
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;  
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;  
import org.springframework.security.crypto.password.PasswordEncoder;  
  
*/\*\*  
 \* Created with IntelliJ IDEA.  
 \*  
 \* @Auther: 雷哥  
 \* @Date: 2020/07/29/22:36  
 \* @Description:  
 \*/*@Configuration  
@EnableGlobalMethodSecurity(securedEnabled = true,prePostEnabled = true)  
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {  
 */\*\*  
 \* 从 Spring5 开始，强制要求密码要加密  
 \*/* @Bean  
 public PasswordEncoder passwordEncoder() {  
 return new BCryptPasswordEncoder();  
 }  
  
 */\*\*  
 \* 核心配置  
 \* @param http  
 \* @throws Exception  
 \*/* @Override  
 protected void configure(HttpSecurity http) throws Exception {  
 //表单登陆配置  
 http.formLogin()  
 .loginPage("/login/toLogin") //登陆地址 跳转到登陆页面的URL  
 .loginProcessingUrl("/login/doLogin")//登陆时提交表单的URL  
 .usernameParameter("username")//表单里面的用户名 field name  
 .passwordParameter("password")//表单里面的密码 field name  
 .successForwardUrl("/index")//登陆成功之后跳转的页面路径  
 .permitAll();//方法允许任意用户访问基于表单登录的所有的URL。  
 //登陆出配置  
 http.logout()  
 .logoutUrl("/login/logout") //退出的URL  
 .logoutSuccessUrl("/login/toLogin")//退出成功后跳转到的URL  
 .permitAll();//方法允许任意用户访问基于表单退出的所有的URL。  
 //权限配置 --这里我们使用注解的方法，这里要求所有URL需要认证  
 http.authorizeRequests()  
 .anyRequest().authenticated();//其它所有路径只要认证过了就可以访问  
 http.csrf().disable();//禁用csrf  
 }  
  
 @Override  
 public void configure(WebSecurity web) throws Exception {  
 */\*\*  
 \* 解决静态资源被拦截的问题  
 \*/* web.ignoring().antMatchers("/resources/\*\*");  
 }  
}

## 创建UserServiceImpl

package com.bjsxt.service.impl;  
  
import com.bjsxt.domain.User;  
import com.bjsxt.mapper.RoleMapper;  
import com.bjsxt.mapper.UserMapper;  
import java.util.List;  
import org.springframework.beans.factory.annotation.Autowired;  
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;  
  
*/\*\*  
 \* Created with IntelliJ IDEA.  
 \*  
 \* @Auther: 雷哥  
 \* @Date: 2020/08/03/22:11  
 \* @Description:  
 \*/*@Service  
public class UserServiceImpl implements UserDetailsService {  
  
 @Autowired  
 private UserMapper userMapper;  
  
 @Autowired  
 private RoleMapper roleMapper;  
  
 @Override  
 public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {  
 User user=userMapper.queryUserByUsername(username);  
 if(null!=user){  
 //根据用户ID查询角色code  
 List<String> roles=this.roleMapper.queryRoleCodeByUserId(user.getUserid());  
 System.*out*.println(roles);  
 //根据用户ID查询权限codes  
 List<String> permissions=this.roleMapper.queryPermssionCodeByUserId(user.getUserid());  
 UserDetails userDetails = org.springframework.security.core.userdetails.User  
 .*withUsername*(user.getUsername())  
 .roles(roles.toArray(new String[roles.size()]))  
 .authorities(permissions.toArray(new String[permissions.size()]))  
 .password(user.getPassword())  
 .build();  
 return userDetails;  
 }  
 return null;  
 }  
}

## 修改UserMapper

User queryUserByUsername(@Param("username") String username);

## 修改UserMapper.xml

<select id="queryUserByUsername" resultMap="BaseResultMap">  
 select \* from user where username=#{username}  
</select>

## 修改RoleMapper

*/\*\*  
 \* 根据用户ID查询角色code  
 \*/*List<String> queryRoleCodeByUserId(@Param("userid") Integer userid);  
  
*/\*\*  
 \* 根据用户ID查询权限code  
 \*/*List<String> queryPermssionCodeByUserId(@Param("userid") Integer userid);

## 修改RoleMapper.xml

<!--根据用户ID查询角色code-->  
<select id="queryRoleCodeByUserId" resultType="java.lang.String">  
 select rolecode from role  
 where roleid in(select roleid from user\_role where userid=#{userid})  
</select>  
<!--根据用户ID查询权限code-->  
<select id="queryPermssionCodeByUserId" resultType="java.lang.String">  
 select distinct percode from permission t1 inner join  
 role\_permission t2 inner join user\_role t3  
 on (t1.perid=t2.perid and t2.roleid=t3.roleid)  
 where userid=#{userid}  
</select>

## 创建IndexController

package com.bjsxt.controller;  
import org.springframework.stereotype.Controller;  
import org.springframework.web.bind.annotation.RequestMapping;  
*/\*\*  
 \* Created with IntelliJ IDEA.  
 \* @Auther: 雷哥  
 \* @Date: 2020/08/03/23:25  
 \* @Description:  
 \*/*@Controller  
public class IndexController {  
  
 @RequestMapping("/index")  
 public String index(){  
 return "index";  
 }  
}

## 创建UserController

package com.bjsxt.controller;  
  
import org.springframework.security.access.prepost.PreAuthorize;  
import org.springframework.security.authentication.AnonymousAuthenticationToken;  
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;  
import org.springframework.security.core.Authentication;  
import org.springframework.security.core.context.SecurityContextHolder;  
import org.springframework.stereotype.Controller;  
import org.springframework.web.bind.annotation.RequestMapping;  
import org.springframework.web.bind.annotation.ResponseBody;  
  
*/\*\*  
 \* Created with IntelliJ IDEA.  
 \*  
 \* @Auther: 雷哥  
 \* @Date: 2020/07/29/23:14  
 \* @Description:  
 \*/*@Controller  
@RequestMapping("user")  
public class UserController {  
  
 @PreAuthorize("hasAuthority('user:query')")  
 @RequestMapping("query")  
 public String query(){  
 return "/user/query";  
 }  
  
 @PreAuthorize("hasAuthority('user:add')")  
 @RequestMapping("add")  
 public String add(){  
 return "/user/add";  
 }  
  
 @PreAuthorize("hasAuthority('user:update')")  
 @RequestMapping("update")  
 public String update(){  
 return "/user/update";  
 }  
 @PreAuthorize("hasAuthority('user:delete')")  
 @RequestMapping("delete")  
 public String delete(){  
 return "/user/delete";  
 }  
  
 @PreAuthorize("hasAuthority('user:export')")  
 @RequestMapping("export")  
 public String export(){  
 return "/user/export";  
 }  
  
 @RequestMapping("getUserInfo")  
 @ResponseBody  
 public Object getUserInfo() {  
 */\*\*  
 SecurityContextHolder.getContext()获取安全上下文对象，就是那个保存在 ThreadLocal 里面的安全上下文对象  
 总是不为null(如果不存在，则创建一个authentication属性为null的empty安全上下文对象)  
 获取当前认证了的 principal(当事人),或者 request token (令牌)  
 如果没有认证，会是 null,该例子是认证之后的情况  
 \*/* Authentication authentication = SecurityContextHolder.*getContext*().getAuthentication();  
 //有登陆用户就返回登录用户，没有就返回null  
 if (authentication != null) {  
 if (authentication instanceof AnonymousAuthenticationToken) {  
 return null;  
 }  
 if (authentication instanceof UsernamePasswordAuthenticationToken) {  
 return authentication.getPrincipal();  
 }  
 }  
 return null;  
 }  
}

## 测试

启动项目，分别经数据库里面zhangsan lisi  wangwu登陆测试

# 【实战】Spring Security集成Thymeleaf详解【2】

## 概述

上一讲里面我们创建的项目里面是当用户点击页面上的链接请求到后台之后没有权限会跳转到403，那么能不能像shiro一样，不让index.html页面没有权限的链接不显示呢，是可以的

我们接着上一个项目来改造

## 修改pom.xml引入security集成thymeleaf的依赖

<!-- 如果在thymeleaf页面相用spring security的标签必须要引入的依赖-->  
<dependency>  
 <groupId>org.thymeleaf.extras</groupId>  
 <artifactId>thymeleaf-extras-springsecurity5</artifactId>  
 <version>3.0.4.RELEASE</version>  
</dependency>

## 修改templates/index.html

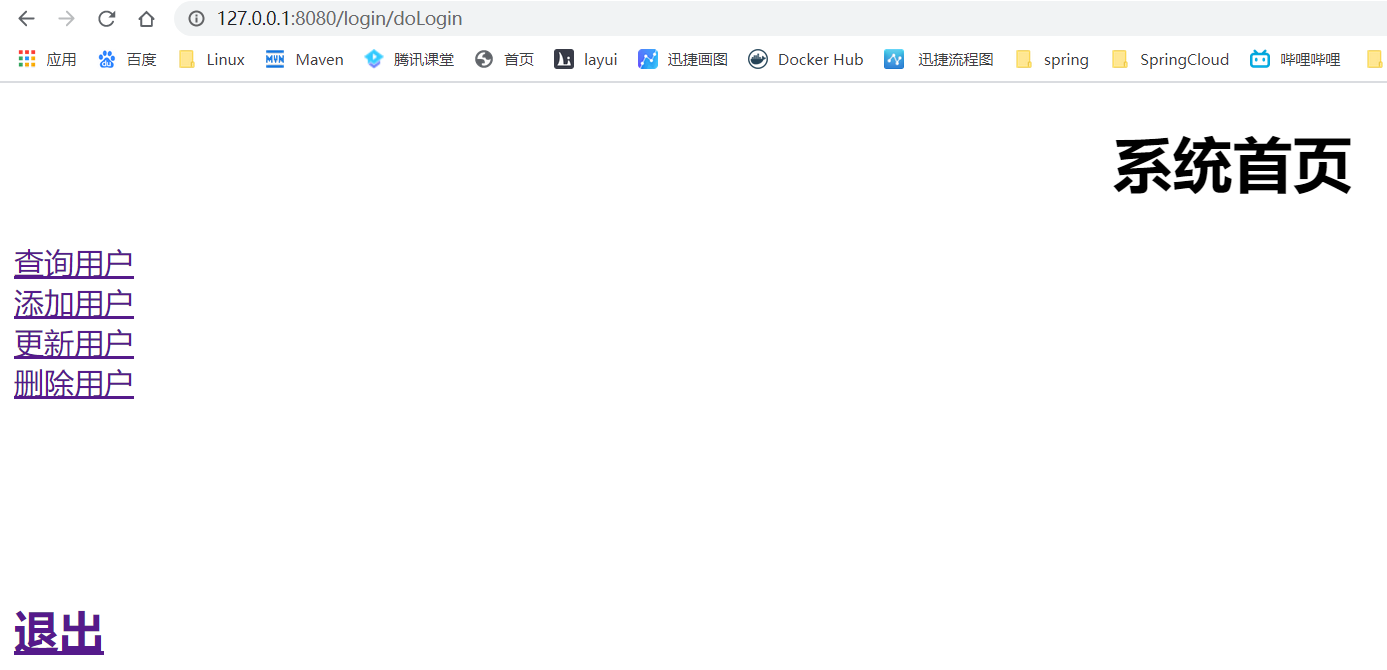
<!DOCTYPE html>  
<html lang="en" xmlns:th="http://www.thymeleaf.org"  
 xmlns:sec="http://www.thymeleaf.org/extras/spring-security">  
<head>  
 <meta charset="UTF-8">  
 <title>系统首页</title>  
</head>  
<body>  
 <h1 align="center">系统首页</h1>  
 <a href="/user/query" sec:authorize="hasAuthority('user:query')">查询用户</a>  
 <br>  
 <a href="/user/add" sec:authorize="hasAuthority('user:add')">添加用户</a>  
 <br>  
 <a href="/user/update" sec:authorize="hasAuthority('user:update')">更新用户</a>  
 <br>  
 <a href="/user/delete" sec:authorize="hasAuthority('user:delete')">删除用户</a>  
 <br>  
 <a href="/user/export" sec:authorize="hasAuthority('user:export')">导出用户</a>  
 <br>  
 <br><br><br>  
 <h2><a href="/login/logout">退出</a></h2>  
 <br>  
</body>  
</html>

## 修改templates/login.html

<!DOCTYPE html>  
<html lang="en" xmlns:th="http://www.thymeleaf.org"  
 xmlns:sec="http://www.thymeleaf.org/extras/spring-security">  
<head>  
 <meta charset="UTF-8">  
 <title>用户登陆</title>  
</head>  
<body>  
<h2>登录页面</h2>  
<h4 th:if="${param.error}" style="color: #FF0000;">帐号或密码错误,请重新输入</h4>  
<form action="/login/doLogin" method="post">  
 <table>  
 <tr>  
 <td>用户名:</td>  
 <td><input type="text" name="username" value="zhangsan"></td>  
 </tr>  
 <tr>  
 <td>密码:</td>  
 <td><input type="password" name="password"></td>  
 </tr>  
 <tr>  
 <td colspan="2"><button type="submit">登录</button></td>  
 </tr>  
 </table>  
</form>  
</body>

## 启动测试

以张三登陆，发现没有导出的链接了



# 【实战】Spring Security集成Thymeleaf图形验证码功能

## 概述

上一讲里面我们集成了Thymeleaf实现在页面链接的动态判断是否显示，那么在实际开发中，我们会遇到有验证码的功能，那么如何处理呢？

注意。在Spring Security的验证码处理方式和Shiro的完全不一样，难度也不是一个量级的

我们接着上一个项目来改造

## 原理、存在问题、解决思路

### 原理

我们知道Spring Security是通过过滤器链来完成了，但是登陆逻辑不像shiro我们可以自己定义，所以它的解决方案是创建一个过滤器放到Security的过滤器链中，在自定义的过滤器中抛出自定义的验证码异常，再使用springmvc的全局异常来捕捉。

### 存在问题

SpringBoot下全局异常处理的几种方式：

BasicExceptionController——SpringBoot默认处理异常方式，用于异常跳转到/error，可实现自定义错误页面请求。

@ExceptionHandle注解——只能在控制器中定义异常处理方法。

@ControllerAdvice+@ExceptionHandler——增强控制前Controller实现异常拦截。

SimpleMappingExceptionResolver——拦截异常跳转到error页面。

HandlerExceptionResolver——实现HandlerExceptionResolver拦截异常。

**上面几种方式只能拦截到控制层的异常，**而Filter在Controller之前，Controller层的异常捕获，是无法捕获到还没有请求到Controller时发生的异常的。

当我们自定义过滤器之后，那么如果验证码不正确，抛出来的异常Spring的全局异常是无法捕获的【是不是有点蛋疼】，别慌，下面给出大家解决方案

### 解决思路

从上面的几种异常处理方式可以发现**，如果要捕获Filter异常，只能通过控制器层定义的全局异常处理来捕获；那么也就只能想办法让Filter中的异常发送到Controller，再由Controller抛出异常，最后由全局异常捕获。**

有了上面的思路，第一个要解决的问题就是怎么让过滤器中的异常在Filter中被捕获到再发送出去。

Filter的实现方式是责任链，第一个Filter处理之后，调用第二个Filter，依次往后，直到Filter全部处理完成后结束；当某一个Filter处理中断，则依次返回结果经过前一个Filter,直到经过第一个Filter过滤后结束过滤责任链。

根据Filter过滤的特点，只需要在业务过滤器之前加上用于处理其他过滤器异常捕获的Filter就可以实现对过滤器异常的处理。

## 修改pom.xml引入hutool用它来生成验证码

<!--引入hutool-->  
<dependency>  
 <groupId>cn.hutool</groupId>  
 <artifactId>hutool-all</artifactId>  
 <version>5.1.5</version>  
</dependency>

## 创建ValidateCodeException

package com.bjsxt.config.exceptions;  
  
*/\*\*  
 \* @Auther: 雷哥  
 \* @Description:验证码异常处理  
 \*/*public class ValidateCodeException extends RuntimeException {  
 public ValidateCodeException(String msg) {  
 super(msg);  
 }  
}

## 创建ValidateCodeFilter

package com.bjsxt.config.filter;  
  
import cn.hutool.captcha.CircleCaptcha;  
import com.bjsxt.config.exceptions.ValidateCodeException;  
import com.bjsxt.controller.IndexController;  
import java.io.IOException;  
import javax.servlet.FilterChain;  
import javax.servlet.ServletException;  
import javax.servlet.http.HttpServletRequest;  
import javax.servlet.http.HttpServletResponse;  
import org.springframework.util.StringUtils;  
import org.springframework.web.filter.OncePerRequestFilter;  
  
*/\*\*  
 \* @Auther: 雷哥  
 \* @Description: 集成spring security的验证码拦击过滤器  
 \*/*public class ValidateCodeFilter extends OncePerRequestFilter {  
  
 @Override  
 protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response,  
 FilterChain filterChain) throws ServletException, IOException, ValidateCodeException {  
 String requestURI = request.getRequestURI();  
 String method = request.getMethod();  
 if (requestURI.equals("/login/doLogin") && method.equalsIgnoreCase("post")) {  
 CircleCaptcha circleCaptcha = (CircleCaptcha) request.getSession()  
 .getAttribute(IndexController.*SESSION\_KEY*);  
 String codeInRequest = request.getParameter("code");  
 if (!StringUtils.*hasText*(codeInRequest)) {  
 request.setAttribute("filter.error", new ValidateCodeException("验证码的值不能为空！"));  
 request.getRequestDispatcher("/error/validateError").forward(request, response);  
 return;  
 }  
 if (circleCaptcha == null) {  
 request.setAttribute("filter.error", new ValidateCodeException("验证码不存在！"));  
 request.getRequestDispatcher("/error/validateError").forward(request, response);  
 return;  
 }  
 if (!circleCaptcha.verify(codeInRequest)) {  
 request.setAttribute("filter.error", new ValidateCodeException("验证码输入不正确！"));  
 request.getRequestDispatcher("/error/validateError").forward(request, response);  
 return;  
 }  
 request.getSession().removeAttribute(IndexController.*SESSION\_KEY*);//移除SESSION里面的验证码对象  
 }  
 filterChain.doFilter(request, response); //如果不是登录请求，直接调用后面的过滤器链  
 }  
}

## 创建ErrorController

package com.bjsxt.controller;  
  
import com.bjsxt.config.exceptions.ValidateCodeException;  
import javax.servlet.http.HttpServletRequest;  
import org.springframework.stereotype.Controller;  
import org.springframework.web.bind.annotation.RequestMapping;  
  
*/\*\*  
 \* @Auther: 雷哥  
 \* @Description: 处理过滤器中抛出的异常问题  
 \*/*@Controller  
public class ErrorController {  
 */\*\*  
 \* 重新抛出异常  
 \*/* @RequestMapping("/error/validateError")  
 public void rethrow(HttpServletRequest request) {  
 throw ((ValidateCodeException) request.getAttribute("filter.error"));  
 }  
}

## 创建GlobalExceptionHandler全局异常处理类

package com.bjsxt.config;  
import com.bjsxt.config.exceptions.ValidateCodeException;  
import org.springframework.ui.Model;  
import org.springframework.web.bind.annotation.ControllerAdvice;  
import org.springframework.web.bind.annotation.ExceptionHandler;  
*/\*\*  
 \* @Auther: 雷哥  
 \* @Description: spring的全局异常处理  
 \*/*@ControllerAdvice  
public class GlobalExceptionHandler {  
 @ExceptionHandler(value = ValidateCodeException.class)  
 public String validateCodeExceptionHandler(Model model, ValidateCodeException e) {  
 model.addAttribute("error", e.getMessage());  
 return "login"; //转发到登陆页面  
 }  
}

## 创建IndexController提供验证码的接口

package com.bjsxt.controller;  
  
import cn.hutool.captcha.CaptchaUtil;  
import cn.hutool.captcha.CircleCaptcha;  
import com.bjsxt.config.GlobalExceptionHandler;  
import javax.imageio.ImageIO;  
import javax.servlet.http.HttpServletRequest;  
import javax.servlet.http.HttpServletResponse;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Controller;  
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.RequestMapping;  
  
*/\*\*  
 \* @Auther: 雷哥  
 \* @Description:  
 \*/*@Controller  
public class IndexController {  
  
 @RequestMapping("/index")  
 public String index() {  
 return "index";  
 }  
  
 public static final String *SESSION\_KEY* = "SESSION\_KEY\_IMAGE\_CODE";  
  
 */\*\*  
 \* 创建验证码，根据验证码类型不同  
 \*/* @GetMapping("/code/image")  
 public void createCode(HttpServletRequest request, HttpServletResponse response)  
 throws Exception {  
 //定义图形验证码的长、宽、验证码字符数、干扰元素个数  
 CircleCaptcha captcha = CaptchaUtil.*createCircleCaptcha*(200, 100, 4, 20);  
 //图形验证码写出，可以写出到文件，也可以写出到流  
 //验证图形验证码的有效性，返回boolean值  
 captcha.verify("1234");  
 request.getSession().setAttribute(*SESSION\_KEY*, captcha);  
 ImageIO.*write*(captcha.getImage(), "JPEG", response.getOutputStream());  
 }  
}

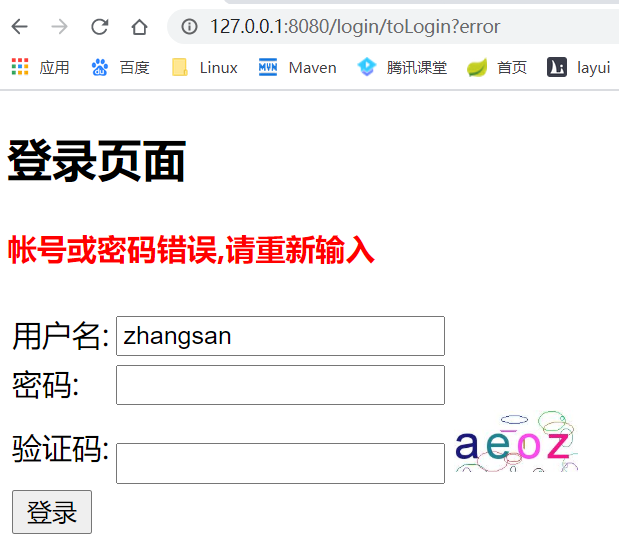
## 修改WebSecurityConfig

package com.bjsxt.config;  
  
import com.bjsxt.config.filter.ValidateCodeFilter;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
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.builders.WebSecurity;  
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;  
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;  
import org.springframework.security.crypto.password.PasswordEncoder;  
import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;  
  
*/\*\*  
 \* @Auther: 雷哥  
 \* @Description:  
 \*/*@Configuration  
@EnableGlobalMethodSecurity(securedEnabled = true,prePostEnabled = true)  
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {  
 */\*\*  
 \* 从 Spring5 开始，强制要求密码要加密  
 \*/* @Bean  
 public PasswordEncoder passwordEncoder() {  
 return new BCryptPasswordEncoder();  
 }  
  
 */\*\*  
 \* 核心配置  
 \* @param http  
 \* @throws Exception  
 \*/* @Override  
 protected void configure(HttpSecurity http) throws Exception {  
 ValidateCodeFilter validateCodeFilter = new ValidateCodeFilter();  
 //添加自定义过滤器  
 http.addFilterBefore(validateCodeFilter, UsernamePasswordAuthenticationFilter.class);  
 //表单登陆配置  
 http.formLogin()  
 .loginPage("/login/toLogin") //登陆地址 跳转到登陆页面的URL  
 .loginProcessingUrl("/login/doLogin")//登陆时提交表单的URL  
 .usernameParameter("username")//表单里面的用户名 field name  
 .passwordParameter("password")//表单里面的密码 field name  
 .successForwardUrl("/index")//登陆成功之后跳转的页面路径  
 .permitAll();//方法允许任意用户访问基于表单登录的所有的URL。  
 //登陆出配置  
 http.logout()  
 .logoutUrl("/login/logout") //退出的URL  
 .logoutSuccessUrl("/login/toLogin")//退出成功后跳转到的URL  
 .permitAll();//方法允许任意用户访问基于表单退出的所有的URL。  
 //权限配置 --这里我们使用注解的方法，这里要求所有URL需要认证  
 http.authorizeRequests()  
 //放行验证码和验证码错误的处理路径  
  **.antMatchers("/code/image","/error/validateError").permitAll()**  
 .anyRequest().authenticated();//其它所有路径只要认证过了就可以访问  
 http.csrf().disable();//禁用csrf  
 }  
  
 @Override  
 public void configure(WebSecurity web) throws Exception {  
 */\*\*  
 \* 解决静态资源被拦截的问题  
 \*/* web.ignoring().antMatchers("/resources/\*\*");  
 }  
}

## 修改页面templates/login.html

<!DOCTYPE html>  
<html lang="en" xmlns:th="http://www.thymeleaf.org"  
 xmlns:sec="http://www.thymeleaf.org/extras/spring-security">  
<head>  
 <meta charset="UTF-8">  
 <title>用户登陆</title>  
</head>  
<body>  
<h2>登录页面</h2>  
<!--${param.error}这个如果有值，就显示帐号或密码错误-->  
**<h4 th:if="${param.error}" style="color: #FF0000;">帐号或密码错误,请重新输入</h4>**  
<form action="/login/doLogin" method="post">  
 <table>  
 <tr>  
 <td>用户名:</td>  
 <td><input type="text" name="username" value="zhangsan"></td>  
 </tr>  
 <tr>  
 <td>密码:</td>  
 <td><input type="password" name="password"></td>  
 </tr>  
 <tr>  
 <td>验证码:</td>  
 <td>**<input type="text" name="code"> <img src="/code/image" style="height:33px;cursor:pointer;" onclick="this.src=this.src">  
 <span th:text="${error}"></span>**  
 </td>  
 </tr>  
 <tr>  
 <td colspan="2"><button type="submit">登录</button></td>  
 </tr>  
 </table>  
</form>  
</body>

## 启动项目测试



# 【实战】Spring Security集成Thymeleaf短信验证码功能

## 概述

上一讲里面我们集成了Thymeleaf+验证码的登陆，那么在实际开发中，我们会遇到有短信登陆的功能，那么如何处理呢？

注意。在Spring Security的验证码处理方式和Shiro的完全不一样，难度也不是一个量级的

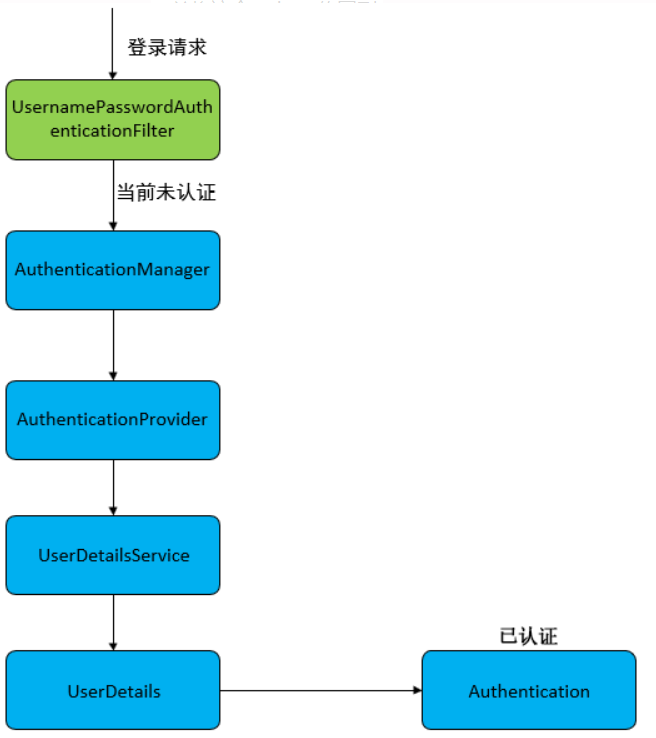
我们接着上一个项目来改造

注意：本案例只是一个模拟，并没有去发短信，【因为发短信要钱，雷哥穷B一个】

## 原理回顾和分析

### 用户名密码登录逻辑

废话不多说，前面几讲中，我们都是以标准的用户名密码登录为例，讲解了整个认证流程。大致流程如下：



* 先进入 UsernamePasswordAuthenticationFilter 中，根据输入的用户名和密码信息，构造出一个暂时没有鉴权的 UsernamePasswordAuthenticationToken，并将 UsernamePasswordAuthenticationToken 交给 AuthenticationManager 处理。
* AuthenticationManager 本身并不做验证处理，他通过 for-each 遍历找到符合当前登录方式的一个 AuthenticationProvider，并交给它进行验证处理，对于用户名密码登录方式，这个 Provider 就是 DaoAuthenticationProvider。
* 在这个 Provider 中进行一系列的验证处理，如果验证通过，就会重新构造一个添加了鉴权的 UsernamePasswordAuthenticationToken，并将这个 token 传回到 UsernamePasswordAuthenticationFilter 中。
* 在该 Filter 的父类 AbstractAuthenticationProcessingFilter 中，会根据上一步验证的结果，跳转到 successHandler 或者是 failureHandler。

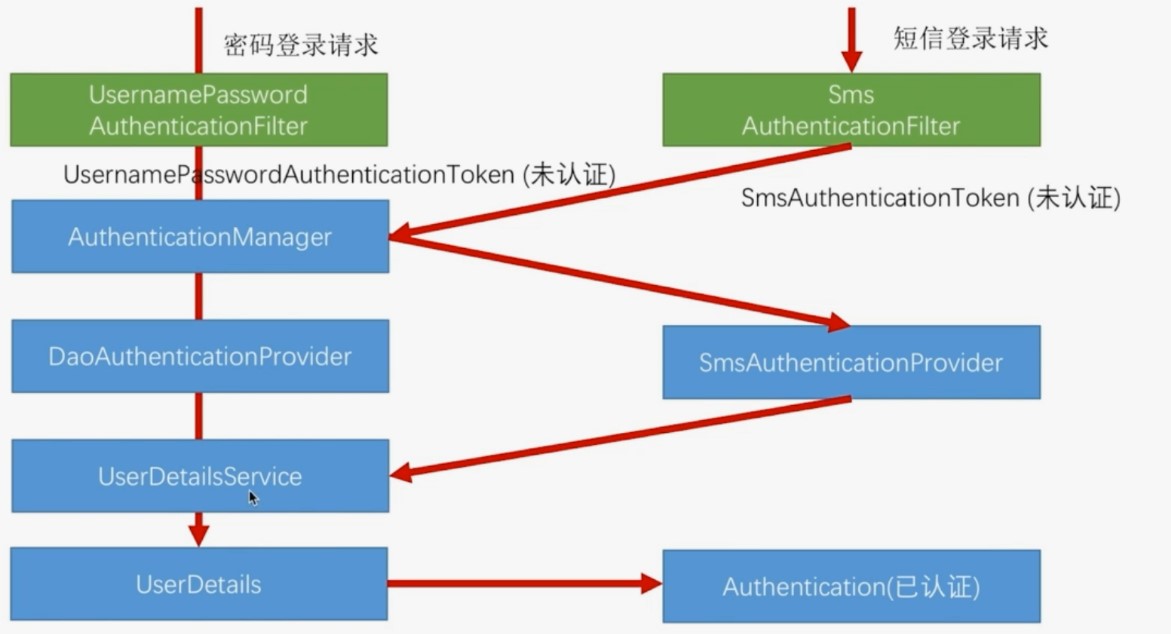
### 短信验证码登录逻辑

我们可以仿照用户名密码登录的逻辑，来实现短信验证码的登陆逻辑。

用户名密码登录有个 UsernamePasswordAuthenticationFilter ，我们搞一个 SmsAuthenticationFilter，代码粘过来改一改。

用户名密码登录需要 UsernamePasswordAuthenticationToken，我们搞一个 SmsAuthenticationToken，代码粘过来改一改。

用户名密码登录需要 DaoAuthenticationProvider，我们模仿它也 implenments AuthenticationProvider，叫做 SmsAuthenticationProvider。



我们自己搞了上面三个类以后，想要实现的效果如上图所示。当我们使用短信验证码登录的时候：

* 先经过 SmsAuthenticationFilter，构造一个没有鉴权的 SmsAuthenticationToken，然后交给 AuthenticationManager 处理。
* AuthenticationManager 通过 for-each 挑选出一个合适的 provider 进行处理，当然我们希望这个 provider 要是 SmsAuthenticationProvider。
* 验证通过后，重新构造一个有鉴权的 SmsAuthenticationToken，并返回给 SmsAuthenticationFilter。
* filter 根据上一步的验证结果，跳转到成功或者失败的处理逻辑。

## 话不多说，上代码

### 创建SmsCodeException异常处理类

*/\*\*  
 \* @Auther: 雷哥  
 \* @Description:验证码异常处理  
 \*/*public class SmsCodeException extends AuthenticationException {  
 public SmsCodeException(String msg) {  
 super(msg);  
 }  
}

### 修改IndexController模拟发送短信接口

@RequestMapping("/sms/code")  
@ResponseBody  
public void sms(String mobile, HttpSession session) {  
 int code = (int) Math.*ceil*(Math.*random*() \* 9000 + 1000);  
 Map<String, Object> map = new HashMap<>(16);  
 map.put("mobile", mobile);  
 map.put("code", code);  
 session.setAttribute("smsCode", map);  
 *log*.info("{}：为 {} 设置短信验证码：{}", session.getId(), mobile, code);  
}

### 修改ErrorController处理短信验证异常

*/\*\*  
 \* 重新抛出异常  
 \*/*@RequestMapping("/error/smsError")  
public void smsError(HttpServletRequest request) {  
 throw ((SmsCodeException) request.getAttribute("filter.error"));  
}

### 创建SmsAuthenticationToken

首先我们编写 SmsAuthenticationToken，这里直接参考 UsernamePasswordAuthenticationToken 源码，直接粘过来，改一改。

步骤：

* principal 原本代表用户名，这里保留，只是代表了手机号码。
* credentials 原本代码密码，短信登录用不到，直接删掉。
* SmsCodeAuthenticationToken() 两个构造方法一个是构造没有鉴权的，一个是构造有鉴权的。
* 剩下的几个方法去除无用属性即可。

package com.bjsxt.config.sms;  
  
import java.util.Collection;  
import org.springframework.security.authentication.AbstractAuthenticationToken;  
import org.springframework.security.core.GrantedAuthority;  
import org.springframework.security.core.SpringSecurityCoreVersion;  
  
*/\*\*  
 \* @Auther: 雷哥  
 \* @Description:  
 \*/*public class SmsCodeAuthenticationToken extends AbstractAuthenticationToken {  
  
 private static final long *serialVersionUID* = SpringSecurityCoreVersion.*SERIAL\_VERSION\_UID*;  
  
 */\*\*  
 \* 在 UsernamePasswordAuthenticationToken 中该字段代表登录的用户名，  
 \* 在这里就代表登录的手机号码  
 \*/* private final Object principal;  
  
 */\*\*  
 \* 构建一个没有鉴权的 SmsCodeAuthenticationToken  
 \*/* public SmsCodeAuthenticationToken(Object principal) {  
 super(null);  
 this.principal = principal;  
 setAuthenticated(false);  
 }  
  
 */\*\*  
 \* 构建拥有鉴权的 SmsCodeAuthenticationToken  
 \*/* public SmsCodeAuthenticationToken(Object principal, Collection<? extends GrantedAuthority> authorities) {  
 super(authorities);  
 this.principal = principal;  
 // must use super, as we override  
 super.setAuthenticated(true);  
 }  
  
 public Object getCredentials() {  
 return null;  
 }  
  
 public Object getPrincipal() {  
 return this.principal;  
 }  
  
 public void setAuthenticated(boolean isAuthenticated) throws IllegalArgumentException {  
 if (isAuthenticated) {  
 throw new IllegalArgumentException(  
 "Cannot set this token to trusted - use constructor which takes a GrantedAuthority list instead");  
 }  
  
 super.setAuthenticated(false);  
 }  
  
 @Override  
 public void eraseCredentials() {  
 super.eraseCredentials();  
 }  
}

### 创建SmsCodeAuthenticationFilter

然后编写 SmsCodeAuthenticationFilter，参考 UsernamePasswordAuthenticationFilter 的源码，直接粘过来，改一改。

步骤：

* 原本的静态字段有 username 和 password，都干掉，换成我们的手机号字段。
* SmsCodeAuthenticationFilter() 中指定了这个 filter 的拦截 Url，我指定为 post 方式的 /sms/login。
* 剩下来的方法把无效的删删改改就好了。

package com.bjsxt.config.sms;  
  
import javax.servlet.http.HttpServletRequest;  
import javax.servlet.http.HttpServletResponse;  
import org.springframework.security.authentication.AuthenticationServiceException;  
import org.springframework.security.core.Authentication;  
import org.springframework.security.core.AuthenticationException;  
import org.springframework.security.web.authentication.AbstractAuthenticationProcessingFilter;  
import org.springframework.security.web.util.matcher.AntPathRequestMatcher;  
import org.springframework.util.Assert;  
  
*/\*\*  
 \* @Auther: 雷哥  
 \* @Description:  
 \*/*public class SmsCodeAuthenticationFilter extends AbstractAuthenticationProcessingFilter {  
  
 */\*\*  
 \* form表单中手机号码的字段name  
 \*/* public static final String *SPRING\_SECURITY\_FORM\_MOBILE\_KEY* = "mobile";  
  
 private String mobileParameter = *SPRING\_SECURITY\_FORM\_MOBILE\_KEY*;  
 */\*\*  
 \* 是否仅 POST 方式  
 \*/* private boolean postOnly = true;  
  
 public SmsCodeAuthenticationFilter() {  
 // 短信登录的请求 post 方式的 /sms/login  
 super(new AntPathRequestMatcher("/sms/login", "POST"));  
 }  
  
 @Override  
 public Authentication attemptAuthentication(HttpServletRequest request, HttpServletResponse response) throws AuthenticationException {  
 if (postOnly && !request.getMethod().equals("POST")) {  
 throw new AuthenticationServiceException(  
 "Authentication method not supported: " + request.getMethod());  
 }  
  
 String mobile = obtainMobile(request);  
  
 if (mobile == null) {  
 mobile = "";  
 }  
  
 mobile = mobile.trim();  
  
 SmsCodeAuthenticationToken authRequest = new SmsCodeAuthenticationToken(mobile);  
  
 // Allow subclasses to set the "details" property  
 setDetails(request, authRequest);  
  
 return this.getAuthenticationManager().authenticate(authRequest);  
 }  
  
 protected String obtainMobile(HttpServletRequest request) {  
 return request.getParameter(mobileParameter);  
 }  
  
 protected void setDetails(HttpServletRequest request, SmsCodeAuthenticationToken authRequest) {  
 authRequest.setDetails(authenticationDetailsSource.buildDetails(request));  
 }  
  
 public String getMobileParameter() {  
 return mobileParameter;  
 }  
  
 public void setMobileParameter(String mobileParameter) {  
 Assert.*hasText*(mobileParameter, "Mobile parameter must not be empty or null");  
 this.mobileParameter = mobileParameter;  
 }  
  
 public void setPostOnly(boolean postOnly) {  
 this.postOnly = postOnly;  
 }  
}

### 创建SmsCodeAuthenticationProvider

这个方法比较重要，这个方法首先能够在使用短信验证码登陆时候被 AuthenticationManager 挑中，其次要在这个类中处理验证逻辑。

步骤：

* 实现 AuthenticationProvider 接口，实现 authenticate() 和 supports() 方法。
* supports() 方法决定了这个 Provider 要怎么被 AuthenticationManager 挑中，我这里通过 return SmsCodeAuthenticationToken.class.isAssignableFrom(authentication)，处理所有 SmsCodeAuthenticationToken 及其子类或子接口。
* authenticate() 方法处理验证逻辑。
* 首先将 authentication 强转为 SmsCodeAuthenticationToken。
* 从中取出登录的 principal，也就是手机号。
* 调用自己写的 checkSmsCode() 方法，进行验证码校验，如果不合法，抛出 AuthenticationException 异常。
* 如果此时仍然没有异常，通过调用 loadUserByUsername(mobile) 读取出数据库中的用户信息。
* 如果仍然能够成功读取，没有异常，这里验证就完成了。
* 重新构造鉴权后的 SmsCodeAuthenticationToken，并返回给 SmsCodeAuthenticationFilter 。
* SmsCodeAuthenticationFilter 的父类在 doFilter() 方法中处理是否有异常，是否成功，根据处理结果跳转到登录成功/失败逻辑。

package com.bjsxt.config.sms;  
  
import com.bjsxt.config.exceptions.SmsCodeException;  
import java.util.Map;  
import javax.servlet.http.HttpServletRequest;  
import javax.servlet.http.HttpServletResponse;  
import lombok.Data;  
import org.springframework.security.authentication.AuthenticationProvider;  
import org.springframework.security.core.Authentication;  
import org.springframework.security.core.AuthenticationException;  
import org.springframework.security.core.userdetails.UserDetails;  
import org.springframework.security.core.userdetails.UserDetailsService;  
import org.springframework.web.context.request.RequestContextHolder;  
import org.springframework.web.context.request.ServletRequestAttributes;  
  
*/\*\*  
 \* Created with IntelliJ IDEA.  
 \*  
 \* @Auther: 雷哥  
 \* @Date: 2020/08/11/0:43  
 \* @Description:  
 \*/*@Data  
public class SmsCodeAuthenticationProvider implements AuthenticationProvider {  
  
 private UserDetailsService userDetailsService;  
  
 @Override  
 public Authentication authenticate(Authentication authentication) throws AuthenticationException {  
 SmsCodeAuthenticationToken authenticationToken = (SmsCodeAuthenticationToken) authentication;  
  
 String mobile = (String) authenticationToken.getPrincipal();  
  
 HttpServletRequest request = ((ServletRequestAttributes) RequestContextHolder.*getRequestAttributes*()).getRequest();  
 HttpServletResponse response = ((ServletRequestAttributes) RequestContextHolder  
 .*getRequestAttributes*()).getResponse();  
 String inputCode = request.getParameter("smsCode");  
  
 Map<String, Object> smsCode = (Map<String, Object>) request.getSession().getAttribute("smsCode");  
 if(smsCode == null) {  
 throw new SmsCodeException("未检测到申请验证码");  
 }  
  
 String applyMobile = (String) smsCode.get("mobile");  
 int code = (int) smsCode.get("code");  
  
 if(!applyMobile.equals(mobile)) {  
 throw new SmsCodeException("申请的手机号码与登录手机号码不一致");  
 }  
 if(code != Integer.*parseInt*(inputCode)) {  
 throw new SmsCodeException("验证码错误");  
 }  
 UserDetails userDetails = userDetailsService.loadUserByUsername(mobile);  
 // 此时鉴权成功后，应当重新 new 一个拥有鉴权的 authenticationResult 返回  
 SmsCodeAuthenticationToken authenticationResult = new SmsCodeAuthenticationToken(userDetails, userDetails.getAuthorities());  
 authenticationResult.setDetails(authenticationToken.getDetails());  
 return authenticationResult;  
 }  
  
 @Override  
 public boolean supports(Class<?> authentication) {  
 // 判断 authentication 是不是 SmsCodeAuthenticationToken 的子类或子接口  
 return SmsCodeAuthenticationToken.class.isAssignableFrom(authentication);  
 }  
}

### 创建SmsCodeAuthenticationSecurityConfig

下面我们需要把我们自己写的这么多类添加进 Spring Security 框架中，在以往，我们都是直接往 WebSecurityConfig 中加，但是这样会导致 WebSecurityConfig 内容太多，难以维护。

因此我们可以为每种登录方式都建议一个专属于它的配置文件，再把这个配置文件加入到 WebSecurityConfig 中，进行解耦。

因此建立短信验证码登录的配置文件 SmsCodeAuthenticationSecurityConfig:

package com.bjsxt.config;  
  
import com.bjsxt.config.sms.SmsCodeAuthenticationFilter;  
import com.bjsxt.config.sms.SmsCodeAuthenticationProvider;  
import com.bjsxt.service.impl.UserServiceImpl;  
import java.io.IOException;  
import javax.servlet.ServletException;  
import javax.servlet.http.HttpServletRequest;  
import javax.servlet.http.HttpServletResponse;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.security.authentication.AuthenticationManager;  
import org.springframework.security.config.annotation.SecurityConfigurerAdapter;  
import org.springframework.security.config.annotation.web.builders.HttpSecurity;  
import org.springframework.security.core.Authentication;  
import org.springframework.security.core.AuthenticationException;  
import org.springframework.security.web.DefaultSecurityFilterChain;  
import org.springframework.security.web.authentication.AuthenticationFailureHandler;  
import org.springframework.security.web.authentication.AuthenticationSuccessHandler;  
import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;  
import org.springframework.stereotype.Component;  
  
*/\*\*  
 \* @Auther: 雷哥  
 \*/*@Component  
public class SmsCodeAuthenticationSecurityConfig extends  
 SecurityConfigurerAdapter<DefaultSecurityFilterChain, HttpSecurity> {  
 @Autowired  
 private UserServiceImpl userDetailsService;  
  
 @Override  
 public void configure(HttpSecurity http) {  
 SmsCodeAuthenticationFilter smsCodeAuthenticationFilter = new SmsCodeAuthenticationFilter();  
 smsCodeAuthenticationFilter.setAuthenticationManager(http.getSharedObject(AuthenticationManager.class));  
 smsCodeAuthenticationFilter.setAuthenticationSuccessHandler(new AuthenticationSuccessHandler() {  
 @Override  
 public void onAuthenticationSuccess(HttpServletRequest request, HttpServletResponse response,  
 Authentication authentication) throws IOException, ServletException {  
 System.*out*.println("验证成功");  
 response.sendRedirect("/index");  
 }  
 });  
 smsCodeAuthenticationFilter.setAuthenticationFailureHandler(new AuthenticationFailureHandler() {  
 @Override  
 public void onAuthenticationFailure(HttpServletRequest request, HttpServletResponse response,  
 AuthenticationException exception) throws IOException, ServletException {  
 System.*out*.println("验证失败");  
 System.*out*.println(exception.getClass());  
 request.setAttribute("filter.error",exception);  
 request.getRequestDispatcher("/error/smsError").forward(request,response);  
 }  
 });  
  
 // 获取验证码提供者  
 SmsCodeAuthenticationProvider smsCodeAuthenticationProvider = new SmsCodeAuthenticationProvider();  
 smsCodeAuthenticationProvider.setUserDetailsService(userDetailsService);  
  
 // 将短信验证码校验器注册到 HttpSecurity， 并将短信验证码过滤器添加在 UsernamePasswordAuthenticationFilter 之前  
 http.authenticationProvider(smsCodeAuthenticationProvider).addFilterAfter(smsCodeAuthenticationFilter, UsernamePasswordAuthenticationFilter.class);  
  
 }  
}

在这个配置文件中，首先给 SmsCodeAuthenticationFilter 指定了：

AuthenticationManager：不指定这个上面的流程图就断掉了。

指定登录成功/失败处理逻辑，方便其父类调用。

然后指定了 SmsCodeAuthenticationProvider，并指定了 UserDetailsService ，方便在验证处理时候通过 loadUserByUsername() 读取出数据库中的用户信息。

最后将 filter 和 provider 都加入 HttpSecurity 配置中。

### 修改WebSecurityConfig

package com.bjsxt.config;  
  
import com.bjsxt.config.filter.ValidateCodeFilter;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
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.builders.WebSecurity;  
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;  
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;  
import org.springframework.security.crypto.password.PasswordEncoder;  
import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;  
  
*/\*\*  
 \* Created with IntelliJ IDEA.  
 \*  
 \* @Auther: 雷哥  
 \* @Date: 2020/07/29/22:36  
 \* @Description:  
 \*/*@Configuration  
@EnableGlobalMethodSecurity(securedEnabled = true,prePostEnabled = true)  
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {  
  
 **@Autowired  
 private SmsCodeAuthenticationSecurityConfig smsCodeAuthenticationSecurityConfig;**  
 */\*\*  
 \* 从 Spring5 开始，强制要求密码要加密  
 \*/* @Bean  
 public PasswordEncoder passwordEncoder() {  
 return new BCryptPasswordEncoder();  
 }  
  
 */\*\*  
 \* 核心配置  
 \* @param http  
 \* @throws Exception  
 \*/* @Override  
 protected void configure(HttpSecurity http) throws Exception {  
 **http.apply(smsCodeAuthenticationSecurityConfig);**  
 ValidateCodeFilter validateCodeFilter = new ValidateCodeFilter();  
 //添加自定义过滤器  
 http.addFilterBefore(validateCodeFilter, UsernamePasswordAuthenticationFilter.class);  
 //表单登陆配置  
 http.formLogin()  
 .loginPage("/login/toLogin") //登陆地址 跳转到登陆页面的URL  
 .loginProcessingUrl("/login/doLogin")//登陆时提交表单的URL  
 .usernameParameter("username")//表单里面的用户名 field name  
 .passwordParameter("password")//表单里面的密码 field name  
 .successForwardUrl("/index")//登陆成功之后跳转的页面路径  
 .permitAll();//方法允许任意用户访问基于表单登录的所有的URL。  
 //登陆出配置  
 http.logout()  
 .logoutUrl("/login/logout") //退出的URL  
 .logoutSuccessUrl("/login/toLogin")//退出成功后跳转到的URL  
 .permitAll();//方法允许任意用户访问基于表单退出的所有的URL。  
 //权限配置 --这里我们使用注解的方法，这里要求所有URL需要认证  
 http.authorizeRequests()  
 //放行验证码和验证码错误的处理路径  
 .antMatchers("/code/image","/error/validateError","/sms/\*\*").permitAll()  
 .anyRequest().authenticated();//其它所有路径只要认证过了就可以访问  
 http.csrf().disable();//禁用csrf  
 }  
  
 @Override  
 public void configure(WebSecurity web) throws Exception {  
 */\*\*  
 \* 解决静态资源被拦截的问题  
 \*/* web.ignoring().antMatchers("/resources/\*\*");  
 }  
  
}

### 修改templates/login.html

<!DOCTYPE html>  
<html lang="en" xmlns:th="http://www.thymeleaf.org"  
 xmlns:sec="http://www.thymeleaf.org/extras/spring-security">  
<head>  
 <meta charset="UTF-8">  
 <title>用户登陆</title>  
</head>  
<body>  
<h2>登录页面</h2>  
<!--${param.error}这个如果有值，就显示帐号或密码错误-->  
<h4 th:if="${param.error}" style="color: #FF0000;">帐号或密码错误,请重新输入</h4>  
<form action="/login/doLogin" method="post">  
 <table>  
 <tr>  
 <td>用户名:</td>  
 <td><input type="text" name="username" value="zhangsan"></td>  
 </tr>  
 <tr>  
 <td>密码:</td>  
 <td><input type="password" name="password"></td>  
 </tr>  
 <tr>  
 <td>验证码:</td>  
 <td><input type="text" name="code"> <img src="/code/image" style="height:33px;cursor:pointer;" onclick="this.src=this.src">  
 <span th:text="${error}"></span>  
 </td>  
 </tr>  
 <tr>  
 <td colspan="2"><button type="submit">登录</button></td>  
 </tr>  
 </table>  
  
  
</form>  
**<!--短信登陆-->  
<form action="/sms/login" method="post">  
 <table>  
 <tr>  
 <td>手机号:</td>  
 <td><input type="text" id="mobile" name="mobile" value="zhangsan"></td>  
 </tr>  
 <tr>  
 <td>验证码:</td>  
 <td><input type="text" name="smsCode">  
 <a href="javascript:void(0);" onclick="sendSms()">获取验证码</a>  
 <span th:text="${smsError}"></span>  
 </td>  
 </tr>  
 <tr>  
 <td colspan="2"><button type="submit">登录</button></td>  
 </tr>  
 </table>  
</form>**  
</body>  
<script>  
  **function sendSms() {  
 window.location.href = '/sms/code?mobile=' + document.getElementById("mobile").value;  
 }**  
</script>

### 启动项目测试，发现使用验证码和短信都可以认证了



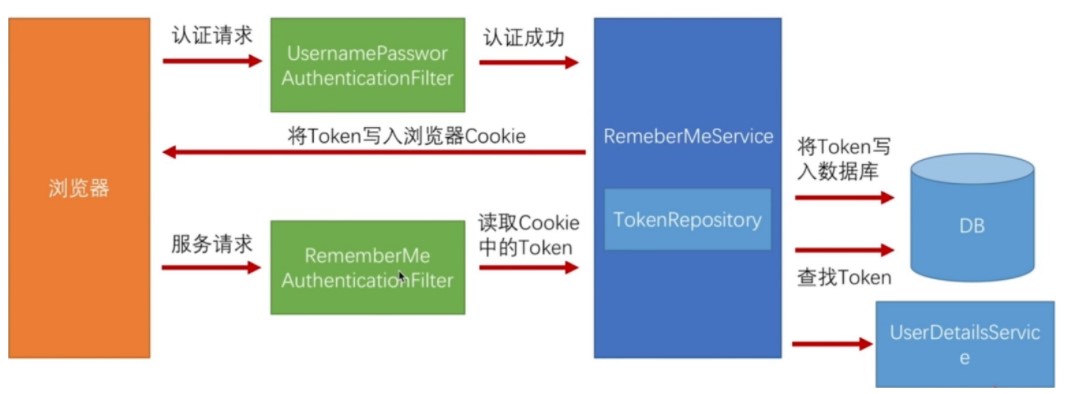
# 【实战】Spring Security集成Thymeleaf记住我功能

## 概述

当用户登录发起认证请求时，会通过UsernamePasswordAuthenticationFilter进行用户认证，认证成功之后，SpringSecurity 调用前期配置好的记住我功能，实际是调用了RememberMeService接口，其接口的实现类会将用户的信息生成Token并将它写入 response 的Cookie中，在写入的同时，内部的TokenRepositoryTokenRepository会将这份Token再存入数据库一份。

当用户再次访问服务器资源的时候，首先会经过RememberMeAuthenticationFiler过滤器，在这个过滤器里面会读取当前请求中携带的 Cookie，这里存着上次服务器保存 的Token，然后去数据库中查找是否有相应的 Token，如果有，则再通过UserDetailsService获取用户的信息

原理图



## 话不多说，搞事情

### 说明

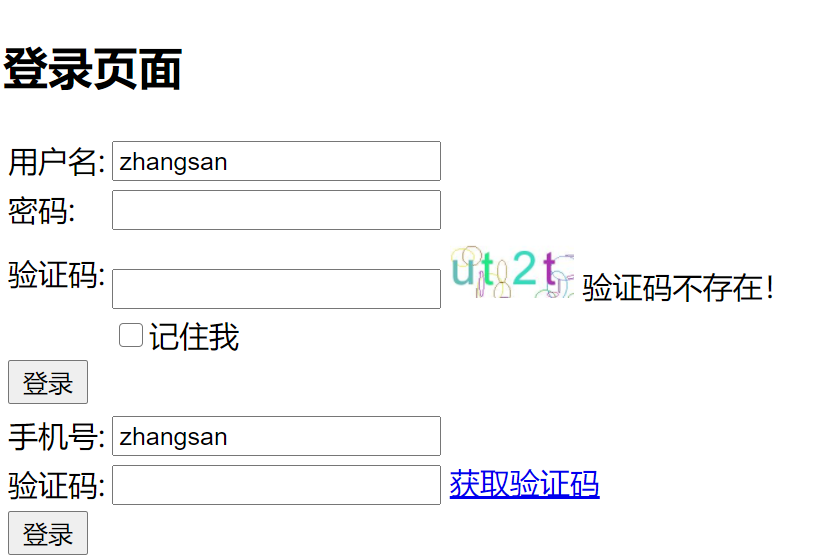
我们还是接着上一个项目来改

### 修改WebSecurityConfig

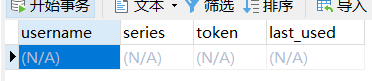
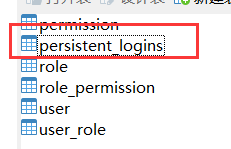
package com.bjsxt.config;  
  
import com.bjsxt.config.filter.ValidateCodeFilter;  
import com.bjsxt.service.impl.UserServiceImpl;  
import javax.sql.DataSource;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
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.builders.WebSecurity;  
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;  
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;  
import org.springframework.security.crypto.password.PasswordEncoder;  
import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;  
import org.springframework.security.web.authentication.rememberme.JdbcTokenRepositoryImpl;  
import org.springframework.security.web.authentication.rememberme.PersistentTokenRepository;  
  
*/\*\*  
 \* Created with IntelliJ IDEA.  
 \*  
 \* @Auther: 雷哥  
 \* @Date: 2020/07/29/22:36  
 \* @Description:  
 \*/*@Configuration  
@EnableGlobalMethodSecurity(securedEnabled = true,prePostEnabled = true)  
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {  
  
 @Autowired  
 private UserServiceImpl userService;  
  
 @Autowired  
 private SmsCodeAuthenticationSecurityConfig smsCodeAuthenticationSecurityConfig;  
 */\*\*  
 \* 从 Spring5 开始，强制要求密码要加密  
 \*/* @Bean  
 public PasswordEncoder passwordEncoder() {  
 return new BCryptPasswordEncoder();  
 }  
  
 */\*\*  
 \* 核心配置  
 \* @param http  
 \* @throws Exception  
 \*/* @Override  
 protected void configure(HttpSecurity http) throws Exception {  
 http.apply(smsCodeAuthenticationSecurityConfig);  
 ValidateCodeFilter validateCodeFilter = new ValidateCodeFilter();  
 //添加自定义过滤器  
 System.*out*.println(http.getSharedObjects().getClass().getSimpleName());  
 http.addFilterBefore(validateCodeFilter, UsernamePasswordAuthenticationFilter.class);  
 //表单登陆配置  
 http.formLogin()  
 .loginPage("/login/toLogin") //登陆地址 跳转到登陆页面的URL  
 .loginProcessingUrl("/login/doLogin")//登陆时提交表单的URL  
 .usernameParameter("username")//表单里面的用户名 field name  
 .passwordParameter("password")//表单里面的密码 field name  
 .successForwardUrl("/index")//登陆成功之后跳转的页面路径  
 .permitAll();//方法允许任意用户访问基于表单登录的所有的URL。  
 //登陆出配置  
 http.logout()  
 .logoutUrl("/login/logout") //退出的URL  
 .logoutSuccessUrl("/login/toLogin")//退出成功后跳转到的URL  
 .permitAll();//方法允许任意用户访问基于表单退出的所有的URL。  
 //权限配置 --这里我们使用注解的方法，这里要求所有URL需要认证  
 http.authorizeRequests()  
 //放行验证码和验证码错误的处理路径  
 .antMatchers("/code/image","/error/validateError","/sms/\*\*").permitAll()  
 .anyRequest().authenticated();//其它所有路径只要认证过了就可以访问  
 http.rememberMe().tokenValiditySeconds(30)//记住的时长单位秒  
 .rememberMeParameter("remember-me")//默认为remember-m  
 .tokenRepository(persistentTokenRepository()) //挺久化的方式  
 .userDetailsService(userService);//记住我  
 http.csrf().disable();//禁用csrf  
 }  
  
 @Override  
 public void configure(WebSecurity web) throws Exception {  
 */\*\*  
 \* 解决静态资源被拦截的问题  
 \*/* web.ignoring().antMatchers("/resources/\*\*");  
 }  
  
 ***/\*\*  
 \* 从配置文件注入  
 \*/* @Autowired  
 private DataSource dataSource;  
  
 */\*\*  
 \* token持久化对象  
 \* @return  
 \*/* @Bean  
 public PersistentTokenRepository persistentTokenRepository(){  
 JdbcTokenRepositoryImpl jdbcTokenRepositoryImpl = new JdbcTokenRepositoryImpl();  
 jdbcTokenRepositoryImpl.setDataSource(dataSource);  
 jdbcTokenRepositoryImpl.setCreateTableOnStartup(false);  
 return jdbcTokenRepositoryImpl;  
 }**  
}

### 修改templates/login.html

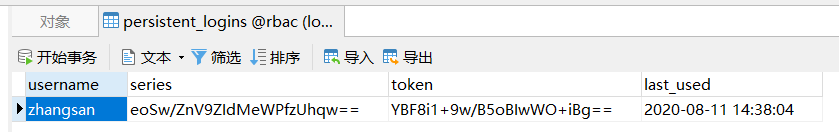
<tr>  
 <td></td>  
 <td><input type="checkbox" name="remember-me">记住我  
 </td>  
</tr>



### 启动项目发现数据库多了一张表



### 登陆一次之后发现有数据。



# 【掌握】JWT概述

## 概述

### 什么是JWT

Json web token (JWT), 是为了在网络应用环境间传递声明而执行的一种基于JSON的开放标准（(RFC 7519).该token被设计为紧凑且安全的，特别适用于分布式站点的单点登录（SSO）场景。JWT的声明一般被用来在身份提供者和服务提供者间传递被认证的用户身份信息，以便于从资源服务器获取资源，也可以增加一些额外的其它业务逻辑所必须的声明信息，该token也可直接被用于认证，也可被加密。

<https://jwt.io/introduction/>

### 跨域认证问题

互联网服务离不开用户认证。一般流程是下面这样。

* 1、用户向服务器发送用户名和密码。
* 2、服务器验证通过后，在当前对话（session）里面保存相关数据，比如用户角色、登录时间等等。
* 3、服务器向用户返回一个 session\_id，写入用户的 Cookie。
* 4、用户随后的每一次请求，都会通过 Cookie，将 session\_id 传回服务器。
* 5、服务器收到 session\_id，找到前期保存的数据，由此得知用户的身份。

这种模式的问题在于，扩展性（scaling）不好。单机当然没有问题，如果是服务器集群，或者是跨域的服务导向架构，就要求 session 数据共享，每台服务器都能够读取 session。

举例来说，A 网站和 B 网站是同一家公司的关联服务。现在要求，用户只要在其中一个网站登录，再访问另一个网站就会自动登录，请问怎么实现？

一种解决方案是 session 数据持久化，写入数据库或别的持久层。各种服务收到请求后，都向持久层请求数据。这种方案的优点是架构清晰，缺点是工程量比较大。另外，持久层万一挂了，就会单点失败。

另一种方案是服务器索性不保存 session 数据了，所有数据都保存在客户端，每次请求都发回服务器。**JWT 就是这种方案的一个代表。**

## JWT 的原理

JWT 的原理是，服务器认证以后，生成一个 JSON 对象，发回给用户，就像下面这样。

{

"姓名": "张三",

"角色": "管理员",

"到期时间": "2018年7月1日0点0分"

}

以后，用户与服务端通信的时候，都要发回这个 JSON 对象。服务器完全只靠这个对象认定用户身份。为了防止用户篡改数据，服务器在生成这个对象的时候，会加上签名（详见后文）。

服务器就不保存任何 session 数据了，也就是说，服务器变成无状态了，从而比较容易实现扩展。

## JWT 的数据结构

实际的 JWT 大概就像下面这样。



它是一个很长的字符串，中间用点（.）分隔成三个部分。注意，JWT 内部是没有换行的，这里只是为了便于展示，将它写成了几行。

JWT 的三个部分依次如下。

* Header（头部）
* Payload（负载）
* Signature（签名）

写成一行，就是下面的样子。

**Header.Payload.Signature**



下面依次介绍这三个部分。

### Header

Header 部分是一个 JSON 对象，描述 JWT 的元数据，通常是下面的样子。

{

"alg": "HS256",

"typ": "JWT"

}

上面代码中，alg属性表示签名的算法（algorithm），默认是 HMAC SHA256（写成 HS256）；typ属性表示这个令牌（token）的类型（type），JWT 令牌统一写为JWT。

最后，将上面的 JSON 对象使用 Base64URL 算法转成字符串。

### Payload

Payload 部分也是一个 JSON 对象，用来存放实际需要传递的数据。JWT 规定了7个官方字段，供选用。

iss (issuer)：签发人

exp (expiration time)：过期时间

sub (subject)：主题

aud (audience)：受众

nbf (Not Before)：生效时间

iat (Issued At)：签发时间

jti (JWT ID)：编号

除了官方字段，你还可以在这个部分定义私有字段，下面就是一个例子。

{

"sub": "1234567890",

"name": "John Doe",

"admin": true

}

注意，JWT 默认是不加密的，任何人都可以读到，所以不要把秘密信息放在这个部分。

这个 JSON 对象也要使用 Base64URL 算法转成字符串。

### Signature

Signature 部分是对前两部分的签名，防止数据篡改。

首先，需要指定一个密钥（secret）。这个密钥只有服务器才知道，不能泄露给用户。然后，使用 Header 里面指定的签名算法（默认是 HMAC SHA256），按照下面的公式产生签名。

 HMACSHA256(

base64UrlEncode(header) + "." +

base64UrlEncode(payload),

secret)

算出签名以后，把 Header、Payload、Signature 三个部分拼成一个字符串，每个部分之间用"点"（.）分隔，就可以返回给用户。

### Base64URL

前面提到，Header 和 Payload 串型化的算法是 Base64URL。这个算法跟 Base64 算法基本类似，但有一些小的不同。

JWT 作为一个令牌（token），有些场合可能会放到 URL（比如 api.example.com/?token=xxx）。Base64 有三个字符+、/和=，在 URL 里面有特殊含义，所以要被替换掉：=被省略、+替换成-，/替换成\_ 。这就是 Base64URL 算法。

## JWT 的使用方式

客户端收到服务器返回的 JWT，可以储存在 Cookie 里面，也可以储存在 localStorage。

此后，客户端每次与服务器通信，都要带上这个 JWT。你可以把它放在 Cookie 里面自动发送，但是这样不能跨域，所以更好的做法是放在 HTTP 请求的头信息Authorization字段里面。

Authorization: Bearer <token>

另一种做法是，跨域的时候，JWT 就放在 POST 请求的数据体里面。

## JWT 的几个特点

（1）JWT 默认是不加密，但也是可以加密的。生成原始 Token 以后，可以用密钥再加密一次。

（2）JWT 不加密的情况下，不能将秘密数据写入 JWT。

（3）JWT 不仅可以用于认证，也可以用于交换信息。有效使用 JWT，可以降低服务器查询数据库的次数。

（4）JWT 的最大缺点是，由于服务器不保存 session 状态，因此无法在使用过程中废止某个 token，或者更改 token 的权限。也就是说，一旦 JWT 签发了，在到期之前就会始终有效，除非服务器部署额外的逻辑。

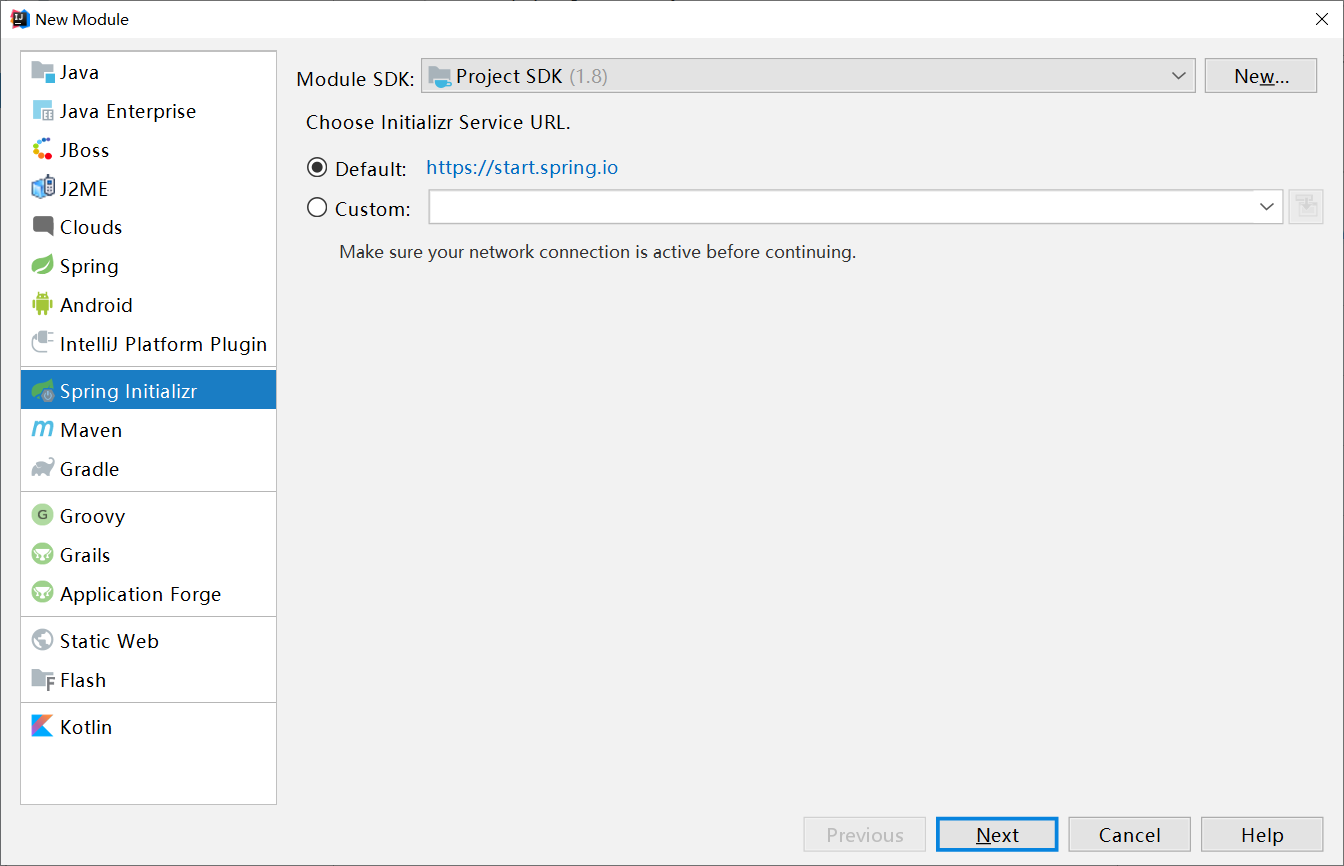
（5）JWT 本身包含了认证信息，一旦泄露，任何人都可以获得该令牌的所有权限。为了减少盗用，JWT 的有效期应该设置得比较短。对于一些比较重要的权限，使用时应该再次对用户进行认证。

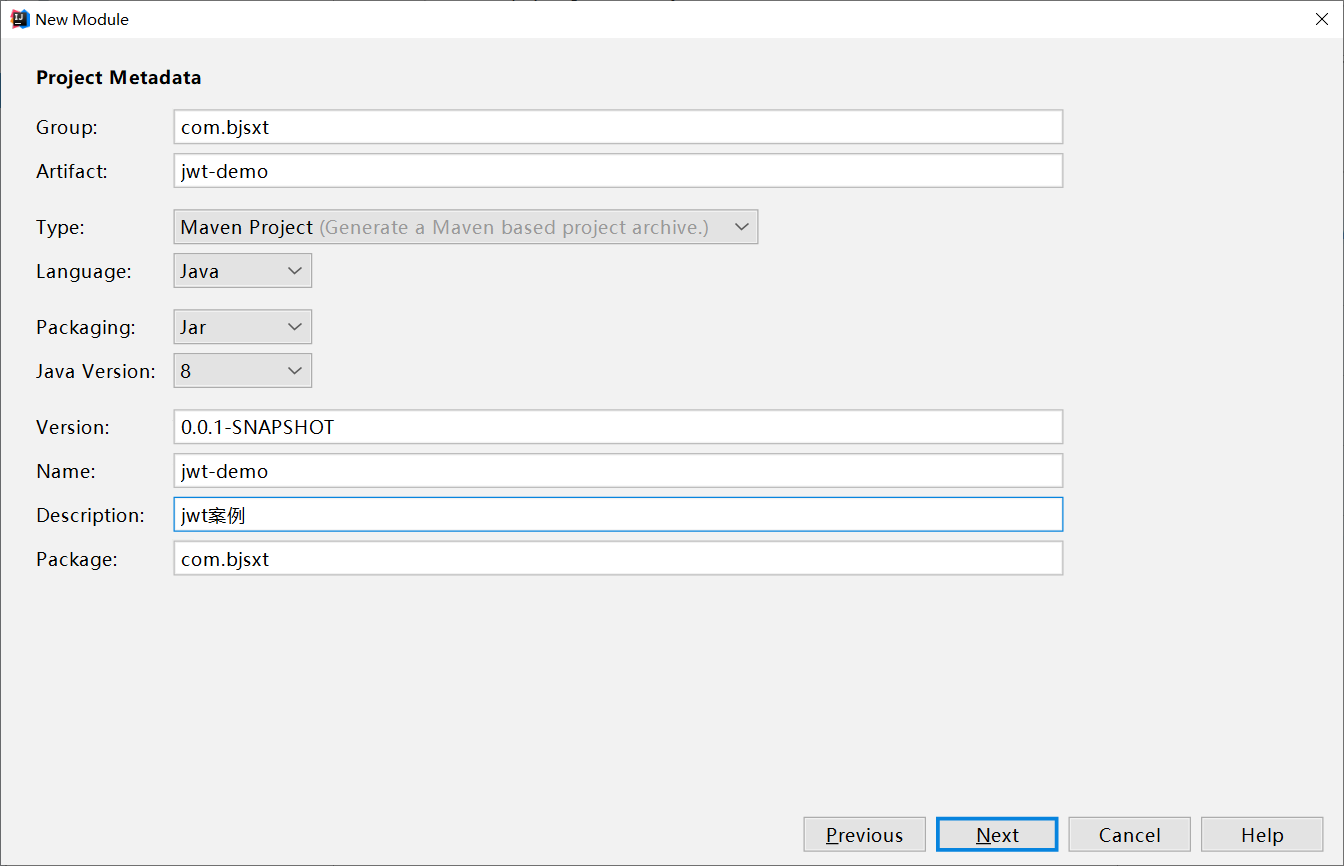
（6）为了减少盗用，JWT 不应该使用 HTTP 协议明码传输，要使用 HTTPS 协议传输。

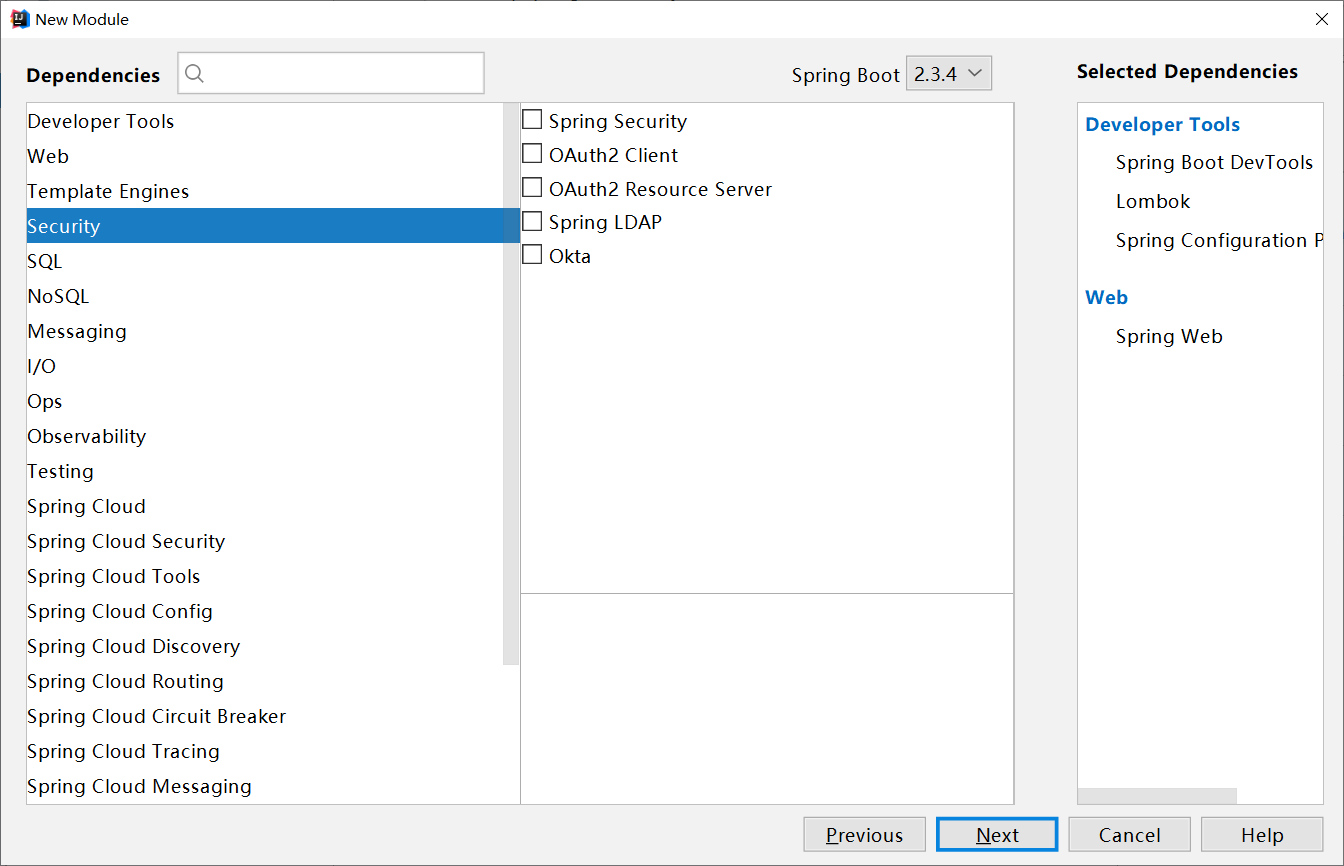
# 【掌握】JWT的使用

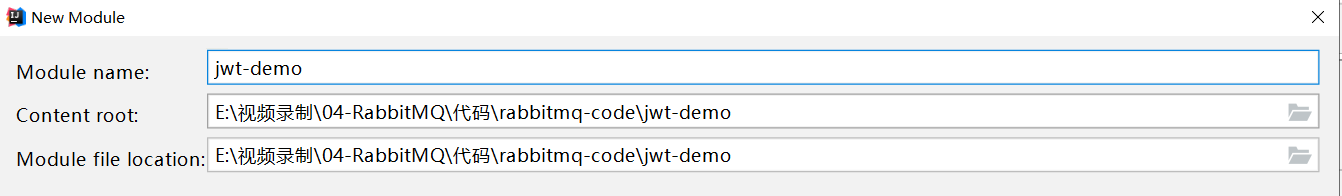
## 准备工作

### 创建项目









### 加入依赖

<!-- https://mvnrepository.com/artifact/com.auth0/java-jwt -->

<dependency>

<groupId>com.auth0</groupId>

<artifactId>java-jwt</artifactId>

<version>3.11.0</version>

</dependency>

## JWT的使用

|  |
| --- |
| package com.bjsxt;  import com.auth0.jwt.JWT; import com.auth0.jwt.algorithms.Algorithm; import com.auth0.jwt.interfaces.Claim; import com.auth0.jwt.interfaces.DecodedJWT; import com.auth0.jwt.interfaces.JWTVerifier; import java.util.Calendar; import java.util.Date; import java.util.HashMap; import java.util.Map; import org.springframework.util.StringUtils;  */\*\*  \* Created with IntelliJ IDEA.  \*  \* @Auther: 雷哥  \* @Date: 2020/10/15/23:06  \* @Description:  \*/* public class JWTTest { */\*\*  \* APP登录Token的生成和解析  \*  \*/   /\*\* token秘钥，请勿泄露，请勿随便修改 backups:JKKLJOoasdlfj \*/* public static final String *SECRET* = "JKKLJOoasdlfj";  */\*\* token 过期时间: 10天 \*/* public static final int *calendarField* = Calendar.*DATE*;  public static final int *calendarInterval* = 10;   */\*\*  \* JWT生成Token.<br/>  \*  \* JWT构成: header, payload, signature  \*  \* @param user\_id  \* 登录成功后用户user\_id, 参数user\_id不可传空  \*/* public static String createToken(Long user\_id) throws Exception {  Date iatDate = new Date();  *// expire time* Calendar nowTime = Calendar.*getInstance*();  nowTime.add(*calendarField*, *calendarInterval*);  Date expiresDate = nowTime.getTime();  *// header Map* Map<String, Object> map = new HashMap<>();  map.put("alg", "HS256");  map.put("typ", "JWT");  *// build token  // param backups {iss:Service, aud:APP}* String token = JWT.*create*().withHeader(map) *// header* .withClaim("iss", "Service") *// payload* .withClaim("aud", "APP").withClaim("user\_id", null == user\_id ? null : user\_id.toString())  .withIssuedAt(iatDate) *// sign time* .withExpiresAt(expiresDate) *// expire time* .sign(Algorithm.*HMAC256*(*SECRET*)); *// signature* return token;  }   */\*\*  \* 解密Token  \*  \* @param token  \* @return  \* @throws Exception  \*/* public static Map<String, Claim> verifyToken(String token) {  DecodedJWT jwt = null;  try {  JWTVerifier verifier = JWT.*require*(Algorithm.*HMAC256*(*SECRET*)).build();  jwt = verifier.verify(token);  } catch (Exception e) {  e.printStackTrace();  *// token 校验失败, 抛出Token验证非法异常* }  return jwt.getClaims();  }   */\*\*  \* 根据Token获取user\_id  \*  \* @param token  \* @return user\_id  \*/* public static Long getAppUID(String token) {  Map<String, Claim> claims = *verifyToken*(token);  Claim user\_id\_claim = claims.get("user\_id");  if (null == user\_id\_claim || StringUtils.*isEmpty*(user\_id\_claim.asString())) {  *// token 校验失败, 抛出Token验证非法异常* }  return Long.*valueOf*(user\_id\_claim.asString());  }   public static void main(String[] args) throws Exception {  String token = *createToken*(110L);  System.*out*.println(token);  } } |

## JJWT的使用

### 新建JwtConstants 类，用于token的chaims保存有效信息字段名

public class JwtConstants {

    public static final String JWT\_KEY\_USER\_ID = "uid";

}

### 新建JwtInfo 类，用于token的chaims保存有效信息

public class JwtInfo {

    private String uid;

    public JwtInfo(String uid) {

        this.uid = uid;

    }

    public String getUid() {

        return uid;

    }

    public void setUid(String uid) {

        this.uid = uid;

    }

}

### 新建JwtTokenUtils 工具类，用于token的生成和解析

package com.bjsxt.user.utils;

import io.jsonwebtoken.Claims;

import io.jsonwebtoken.Jws;

import io.jsonwebtoken.Jwts;

import io.jsonwebtoken.SignatureAlgorithm;

import org.joda.time.DateTime;

import javax.crypto.spec.SecretKeySpec;

import javax.xml.bind.DatatypeConverter;

import java.security.Key;

public class JwtTokenUtils {

    private static Key getKeyInstance(){

        SignatureAlgorithm signatureAlgorithm = SignatureAlgorithm.HS256;

        byte[] bytes = DatatypeConverter.parseBase64Binary("mall-user");

        return new SecretKeySpec(bytes,signatureAlgorithm.getJcaName());

    }

    /\*\*

     \* 生成token的方法

     \* @param jwtInfo

     \* @param expire

     \* @return

     \*/

    public static String generatorToken(JwtInfo jwtInfo,int expire){

        return Jwts.builder().claim(JwtConstants.JWT\_KEY\_USER\_ID,jwtInfo.getUid())

                .setExpiration(DateTime.now().plusSeconds(expire).toDate())

                .signWith(SignatureAlgorithm.HS256,getKeyInstance()).compact();

    }

    /\*\*

     \* 根据token获取token中的信息

     \* @param token

     \* @return

     \*/

    public static JwtInfo getTokenInfo(String token){

        Jws<Claims> claimsJws = Jwts.parser().setSigningKey(getKeyInstance()).parseClaimsJws(token);

        Claims claims = claimsJws.getBody();

        return new JwtInfo(claims.get(JwtConstants.JWT\_KEY\_USER\_ID).toString());

    }

}

### 新建JwtTokenService服务类，调用其方法即可进行认证和授权使用。

@Component

public class JwtTokenService {

    /\*\*

     \* token过期时间

     \*/

    private int expire = 6000;

    public String generatorToken(JwtInfo jwtInfo){

        return JwtTokenUtils.generatorToken(jwtInfo,expire);

    }

    public JwtInfo stringInfoFromToken(String token){

        return JwtTokenUtils.getTokenInfo(token);

    }

}

至此JWT的具体使用详细介绍完毕，在项目中建议使用JJWT，因为其API友好。

上面将token中的载荷放在chaims中，其实chaims是JWT内部维持的一个存放有效信息的地方，不论使用任何API，最终都使用chaims保存信息。

setClaims有2个重载

JwtBuilder setClaims(Claims claims);

JwtBuilder setClaims(Map<String, Object> claims);

不能就是说，我们也可以直接传入map值对象。