SpringSecurity第一天

上海千锋 文威

**课程目标:**

* 初识认证和授权
* SpringSecurity案例入门
* 理解SpringSecurity过滤器
* 使用自定义认证页面
* 使用数据库完成认证

# 1 初识权限管理

## 1.1 权限管理的概念

权限管理，一般指根据系统设置的安全规则或者安全策略，用户可以访问而且只能访问自己被授权的资源。权限管理几乎出现在任何系统里面，前提是需要有用户和密码认证的系统。

在权限管理的概念中，有两个非常重要的名词：

认证：通过用户名和密码成功登陆系统后，让系统得到当前用户的角色身份。

如何理解认证?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。

授权：系统根据当前用户的角色，给其授予对应可以操作的权限资源。

如何理解授权?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。其中有一个功能：微信运动。只有你授权运行微信运动功能的运行，你每天才可以看见你的步数。如果不授权你是看不到的。

## 1.2 权限管理的三个对象

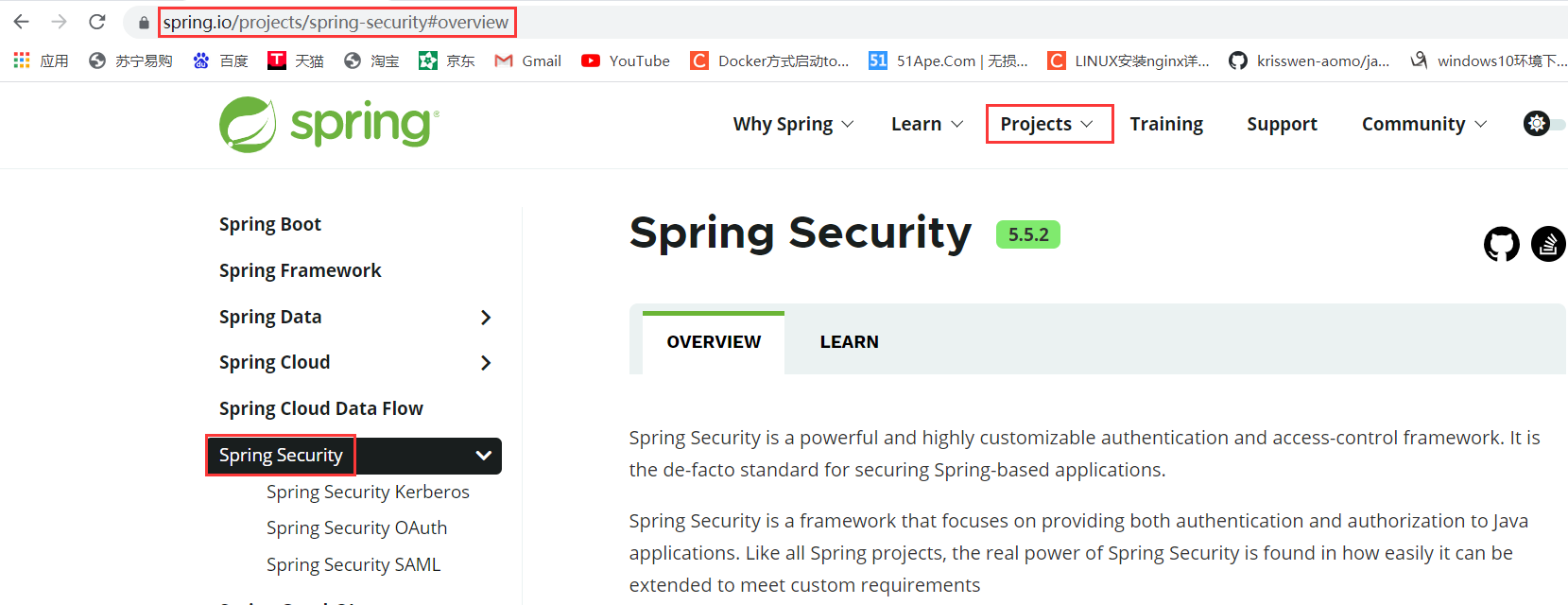
用户：主要包含用户名，密码和当前用户的角色信息，可实现认证操作。

角色：主要包含角色名称，角色描述和当前角色拥有的权限信息，可实现授权操作。

权限：权限也可以称为菜单，主要包含当前权限名称，url地址等信息，可实现动态展示菜单。

注：这三个对象中，用户与角色是多对多的关系，角色与权限是多对多的关系，用户与权限没有直接关系，二者是通过角色来建立关联关系的。

## 1.3 什么是SpringSecurity



SpringSecurity是spring采用AOP思想，基于servlet过滤器实现的安全框架。它提供了完善的认证机制和方法级的授权功能。是一款非常优秀的权限管理框架。

# 2 SpringSecurity第一个入门程序

## 2.1 SpringSecurity需要的依赖

SpringSecurity主要jar包功能介绍:

spring-security-core.jar核心包，任何SpringSecurity功能都需要此包。

spring-security-web.jarweb工程必备，包含过滤器和相关的Web安全基础结构代码。spring-security-config.jar用于解析xml配置文件，用到SpringSecurity的xml配置文件的就要用到此包。

spring-security-taglibs.jarSpringSecurity提供的动态标签库，jsp页面可以用。

|  |
| --- |
| <**dependency**> <**groupId**>org.springframework.security</**groupId**> <**artifactId**>spring-security-taglibs</**artifactId**> <**version**>${spring.security.version}</**version**> </**dependency**> <**dependency**> <**groupId**>org.springframework.security</**groupId**> <**artifactId**>spring-security-config</**artifactId**> <**version**>${spring.security.version}</**version**> </**dependency**> |

由于和web工程集成，我们在项目中可以不用引入core和web的依赖。因为spring工程有spring-core包，springmvc有spring-web包。可以依赖传递过来。

## 2.2 创建web工程

### 2.2.1 使用maven构建web项目

构建过程省略，然后引入相关的依赖。

|  |
| --- |
| <**dependencies**> <**dependency**> <**groupId**>org.springframework</**groupId**> <**artifactId**>spring-context</**artifactId**> <**version**>${spring.version}</**version**> </**dependency**> <**dependency**> <**groupId**>org.springframework</**groupId**> <**artifactId**>spring-web</**artifactId**> <**version**>${spring.version}</**version**> </**dependency**> <**dependency**> <**groupId**>org.springframework</**groupId**> <**artifactId**>spring-webmvc</**artifactId**> <**version**>${spring.version}</**version**> </**dependency**> <**dependency**> <**groupId**>javax.servlet</**groupId**> <**artifactId**>javax.servlet-api</**artifactId**> <**version**>3.1.0</**version**> <**scope**>provided</**scope**> </**dependency**> <**dependency**> <**groupId**>javax.servlet</**groupId**> <**artifactId**>jsp-api</**artifactId**> <**version**>2.0</**version**> <**scope**>provided</**scope**> </**dependency**>  <**dependency**> <**groupId**>org.slf4j</**groupId**> <**artifactId**>slf4j-log4j12</**artifactId**> <**version**>1.7.26</**version**> </**dependency**>  <**dependency**> <**groupId**>org.springframework.security</**groupId**> <**artifactId**>spring-security-taglibs</**artifactId**> <**version**>${spring.security.version}</**version**> </**dependency**> <**dependency**> <**groupId**>org.springframework.security</**groupId**> <**artifactId**>spring-security-config</**artifactId**> <**version**>${spring.security.version}</**version**> </**dependency**> </**dependencies**> |

### 2.2.2 配置web.xml

|  |
| --- |
| *<?***xmlversion="1.0"encoding="UTF-8"***?>* <**web-appxmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://xmlns.jcp.org/xml/ns/javaee" xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app\_3\_1.xsd" version="3.1"**> <**filter**> <**filter-name**>springSecurityFilterChain</**filter-name**> <**filter-class**>org.springframework.web.filter.DelegatingFilterProxy</**filter-class**> </**filter**> <**filter-mapping**> <**filter-name**>springSecurityFilterChain</**filter-name**> <**url-pattern**>/\*</**url-pattern**> </**filter-mapping**> </**web-app**> |

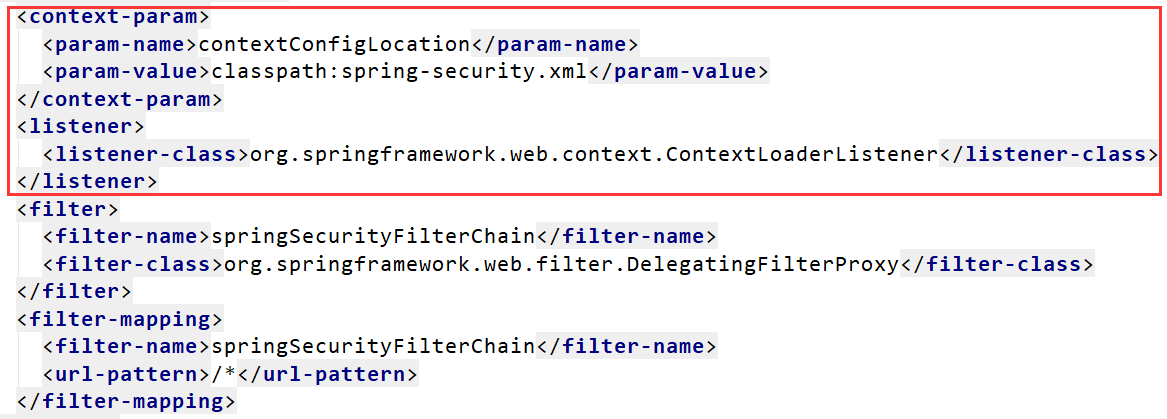
### 2.2.3 创建springSecurity.xml

|  |
| --- |
| *<?***xmlversion="1.0"encoding="UTF-8"***?>* <**beansxmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:security="http://www.springframework.org/schema/security" xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd http://www.springframework.org/schema/security http://www.springframework.org/schema/security/spring-security.xsd"**> *<!-- auto-config:true开启spring-security的自动配置 use-expressions="true"允许使用spring表达式 -->* <**security:httpauto-config="true"use-expressions="true"**> *<!-- intercept-url:拦截的请求资源/\*\*所有的请求全部拦截 access:访问资源所需要的角色 -->* <**security:intercept-urlpattern="/\*\*"access="hasAnyRole('ROLE\_ADMIN','ROLE\_USER')"**></**security:intercept-url**> </**security:http**> *<!--配置认证管理器-->* <**security:authentication-manager**> *<!--配置认证提供者-->* <**security:authentication-provider**> *<!--配置认证服务-->* <**security:user-service**> *<!-- 配置认证者 name用户名 password密码 authorities权限资源 -->* <**security:username="user"password="{noop}user"authorities="ROLE\_USER"**></**security:user**> <**security:username="admin"password="{noop}admin"authorities="ROLE\_ADMIN"**></**security:user**> </**security:user-service**> </**security:authentication-provider**> </**security:authentication-manager**> </**beans**> |

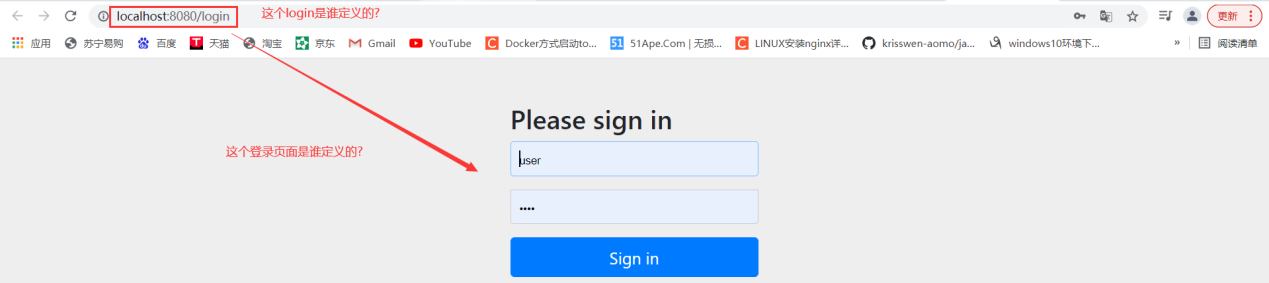
### 2.2.4 加载springSecurity.xm配置文件

|  |
| --- |
| <**context-param**> <**param-name**>contextConfigLocation</**param-name**> <**param-value**>classpath:spring-security.xml</**param-value**> </**context-param**> <**listener**> <**listener-class**>org.springframework.web.context.ContextLoaderListener</**listener-class**> </**listener**> |

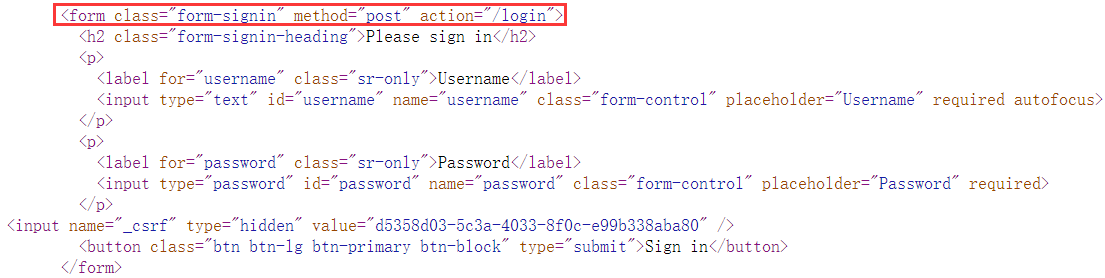
整个web.xml文件最后目录如下:



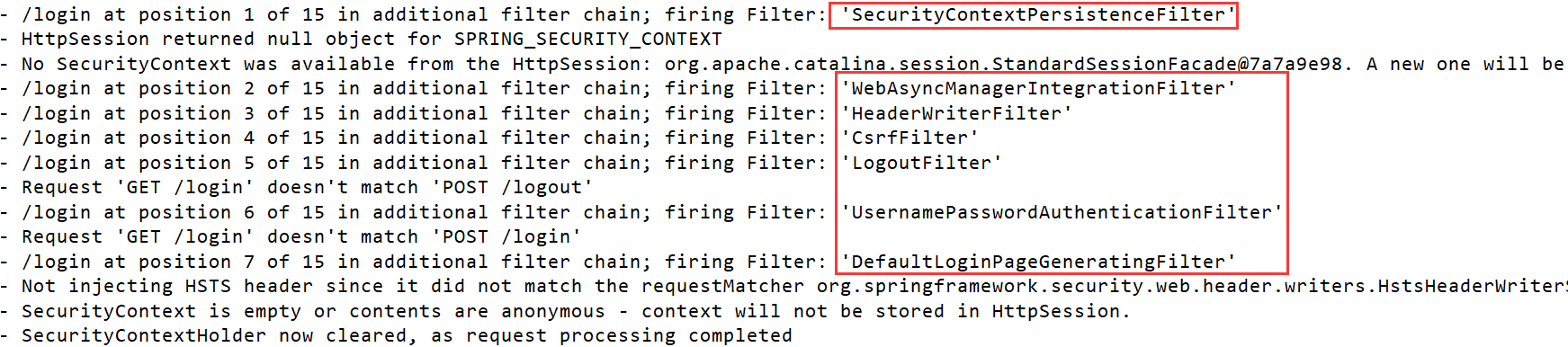
到这里，我们第一个springSecurity案例搭建完毕。启动tomcat，查看效果:



这个登录页面是谁写的?我们查看网页源代码看看:



我们再去看看控制台发生了什么：



我们发现加载了很多过滤器。

最后，我们在这个登录页面上输入用户名user，密码user，点击Signin，好了，总算再次看到首页了！



# 3 SpringSecurity过滤器链

## 3.1 SpringSecurity常用过滤器介绍

过滤器是一种典型的AOP思想，关于什么是过滤器，就不赘述了，谁还不知道凡是web工程都能用过滤器？接下来咱们就一起看看SpringSecurity中这些过滤器都是干啥用的，源码我就不贴出来了，有名字，大家可以自己在idea中DoubleShift去。我也会在后续的学习过程中穿插详细解释。

1.org.springframework.security.web.context.SecurityContextPersistenceFilter

首当其冲的一个过滤器，作用之重要，自不必多言。

SecurityContextPersistenceFilter主要是使用SecurityContextRepository在session中保存或更新一个SecurityContext，并将SecurityContext给以后的过滤器使用，来为后续filter建立所需的上下文。SecurityContext中存储了当前用户的认证以及权限信息。

简单的说就是初始化一个类似于springIOC的容器。

2.org.springframework.security.web.context.request.async.WebAsyncManagerIntegrationFilter

此过滤器用于集成SecurityContext到Spring异步执行机制中的WebAsyncManager

如果springSecurity要整合到spring工程里面去，必须使用这个过滤器。

3.org.springframework.security.web.header.HeaderWriterFilter

向请求的Header中添加相应的信息,可在http标签内部使用security:headers来控制

4.org.springframework.security.web.csrf.CsrfFilter

csrf又称跨域请求伪造，SpringSecurity会对所有post请求验证是否包含系统生成的csrf的token信息，如果不包含，则报错。起到防止csrf攻击的效果。

5.org.springframework.security.web.authentication.logout.LogoutFilter

匹配URL为/logout的请求，实现用户退出,清除认证信息。

6.org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter

认证操作全靠这个过滤器，默认匹配URL为/login且必须为POST请求。

7.org.springframework.security.web.authentication.ui.DefaultLoginPageGeneratingFilter

如果没有在配置文件中指定认证页面，则由该过滤器生成一个默认认证页面。

8.org.springframework.security.web.authentication.ui.DefaultLogoutPageGeneratingFilter

由此过滤器可以生产一个默认的退出登录页面

9.org.springframework.security.web.authentication.www.BasicAuthenticationFilter

此过滤器会自动解析HTTP请求中头部名字为Authentication，且以Basic开头的头信息。

简而言之就是解析Http请求头信息。

10.org.springframework.security.web.savedrequest.RequestCacheAwareFilter

通过HttpSessionRequestCache内部维护了一个RequestCache，用于缓存HttpServletRequest

11.org.springframework.security.web.servletapi.SecurityContextHolderAwareRequestFilter

针对ServletRequest进行了一次包装，使得request具有更加丰富的API

12.org.springframework.security.web.authentication.AnonymousAuthenticationFilter

当SecurityContextHolder中认证信息为空,则会创建一个匿名用户存入到SecurityContextHolder中。springsecurity为了兼容未登录的访问，也走了一套认证流程，只不过是一个匿名的身份。

13.org.springframework.security.web.session.SessionManagementFilterSecurityContextRepository限制同一用户开启多个会话的数量

14.org.springframework.security.web.access.ExceptionTranslationFilter

异常转换过滤器位于整个springSecurityFilterChain的后方，用来转换整个链路中出现的异常。简单说就是进行springSecurity中的异常处理。

15.org.springframework.security.web.access.intercept.FilterSecurityInterceptor获取所配置资源访问的授权信息，根据SecurityContextHolder中存储的用户信息来决定其是否有权限。简单的说就是授权。

好了！这一堆排山倒海的过滤器介绍完了。那么，是不是springsecurity一共就这么多过滤器呢？

答案是否定的！随着spring-security.xml配置的添加，还会出现新的过滤器。

那么，是不是springsecurity每次都会加载这些过滤器呢？

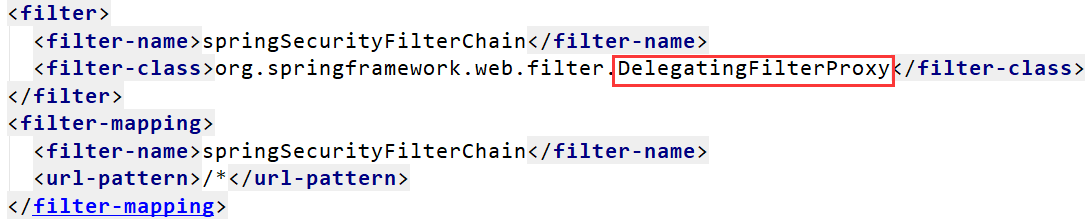
答案也是否定的！随着spring-security.xml配置的修改，有些过滤器可能会被去掉。

## 3.2 Springsecurity过滤器链加载原理

通过前面十五个过滤器功能的介绍，对于SpringSecurity简单入门中的疑惑是不是在心中已经有了答案了呀？但新的问题来了！我们并没有在web.xml中配置这些过滤器啊？它们都是怎么被加载出来的？

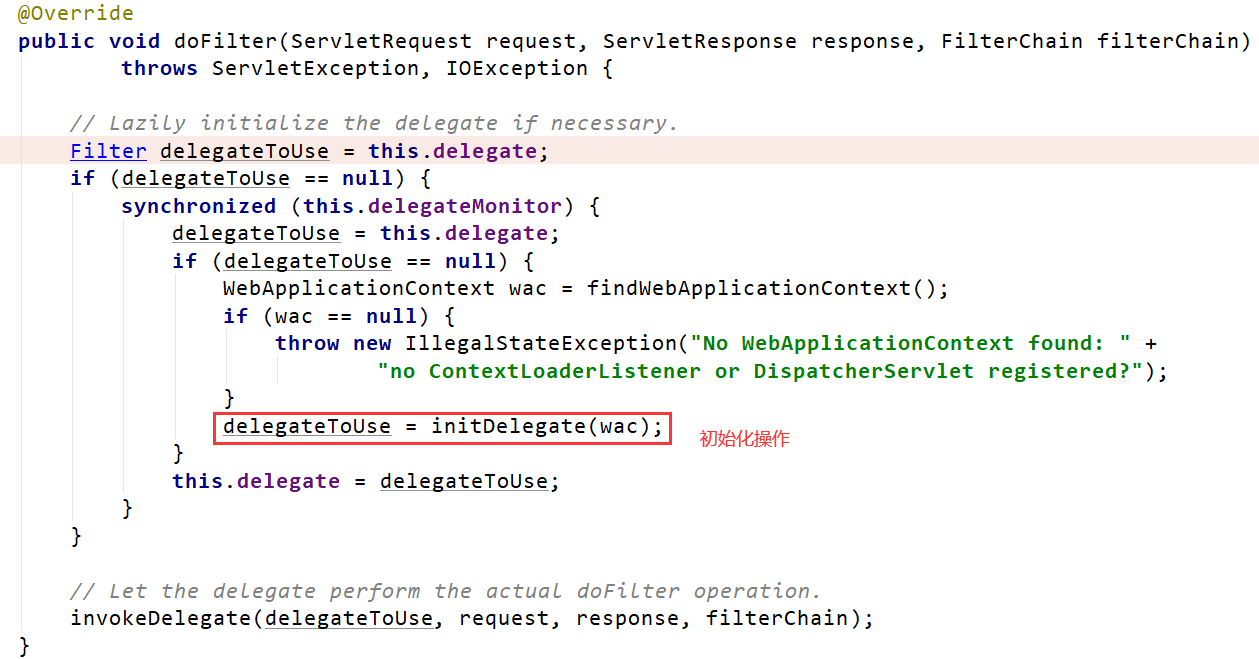
### 3.2.1 DelegatingFilterProxy

我们在web.xml中配置了一个名称为springSecurityFilterChain的过滤器DelegatingFilterProxy，接下我直接对DelegatingFilterProxy源码里重要代码进行说明。

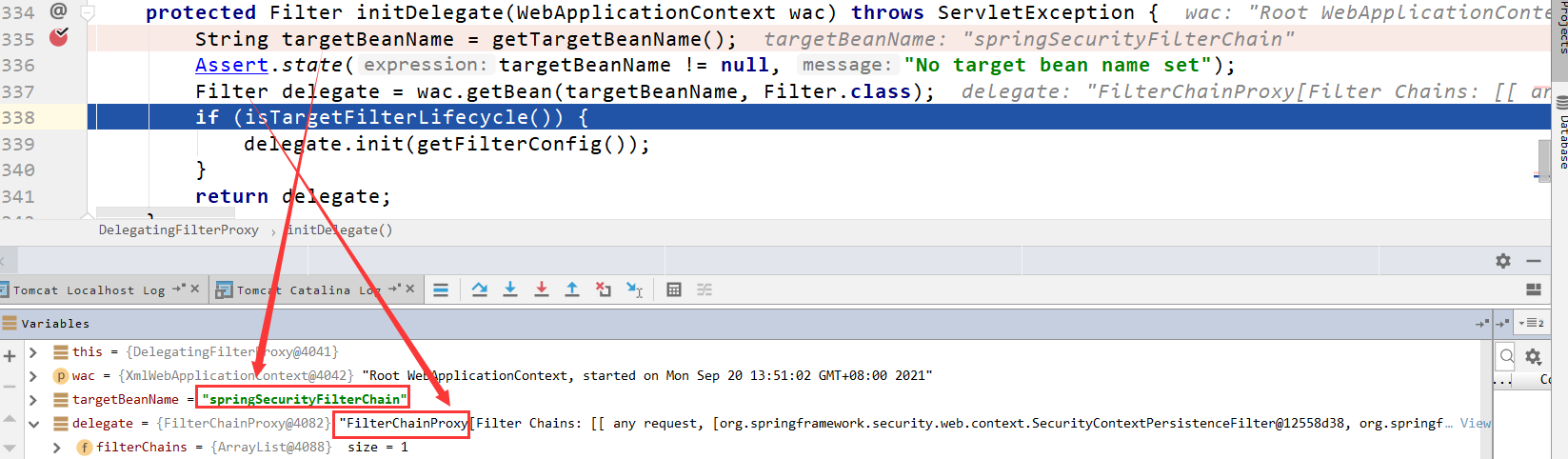


我们点击进入DelegatingFilterProxy的源码：

执行过滤器，肯定要看这个类的doFiler源码:



然后我们再进入initDelegate方法里面。观察它的具体的初始化流程。我们可以打断点调试：



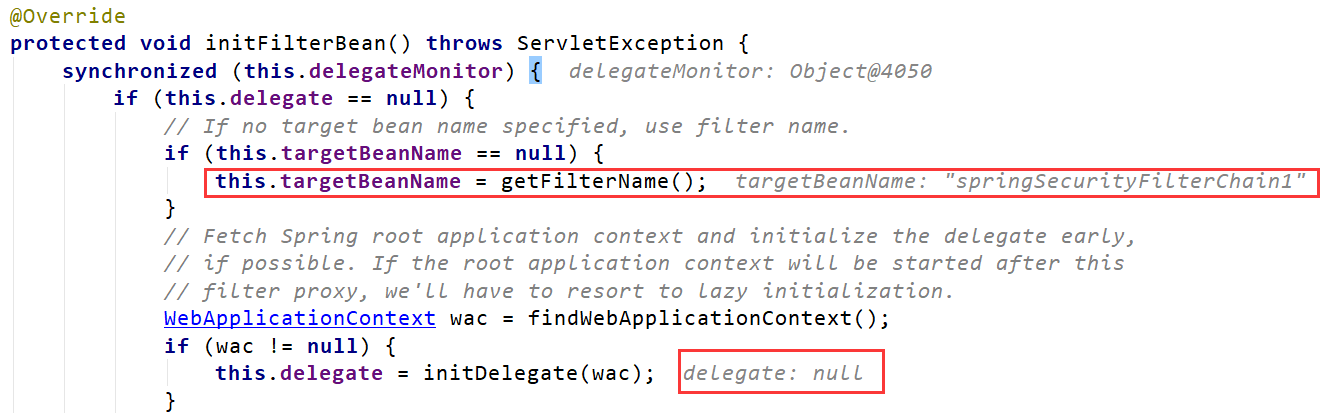
总结两点:

1.这里我们在web.xml里面配置的filter-name必须是这个名字:

springSecurityFilterChain。否则最后初始化的delegate为null。

2.DelegatingFilterProxy通过springSecurityFilterChain这个名称，得到了一个FilterChainProxy过滤器。

如果filter-name不是这个名字会出现什么情况?

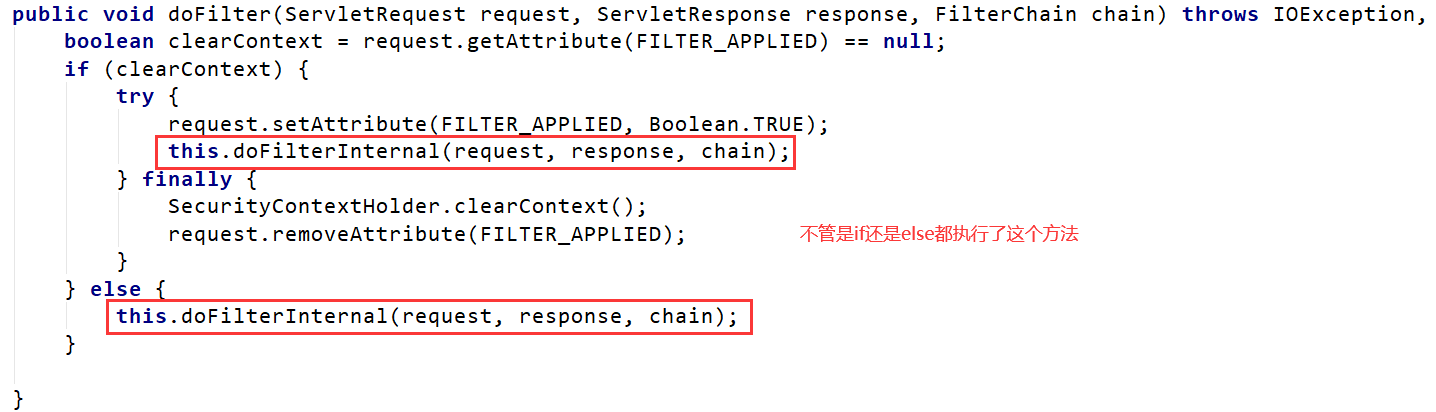


**所以再说一遍！！！！filter-name名称必须是springSecurityFilterChain！！！**

好了，那现在疑问又来了，什么是FilterChainProxy?

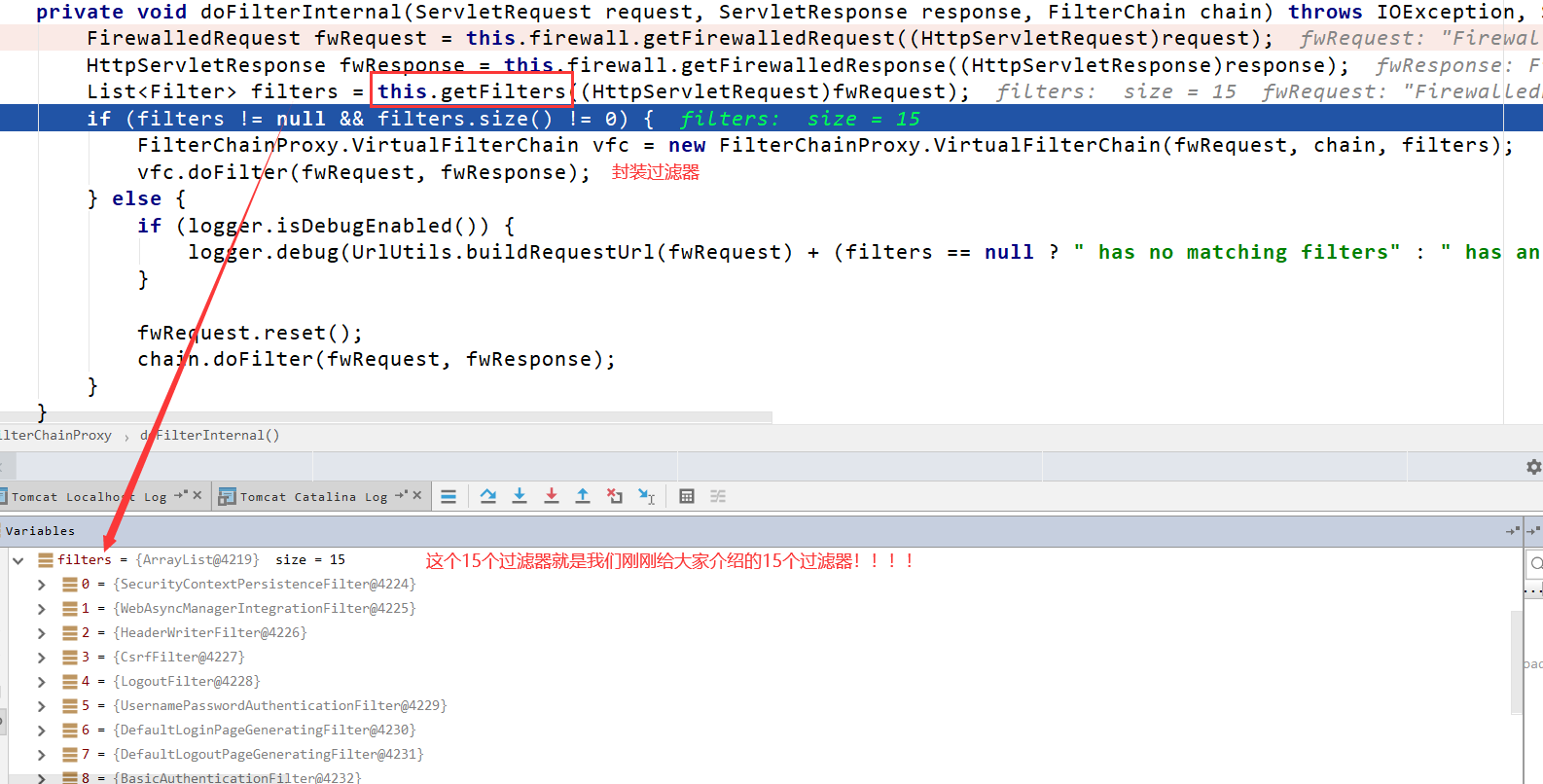
### 3.2.2 FilterChainProxy

我们进入FilterChainProxy源码就知道了。还是找到doFilter方法。



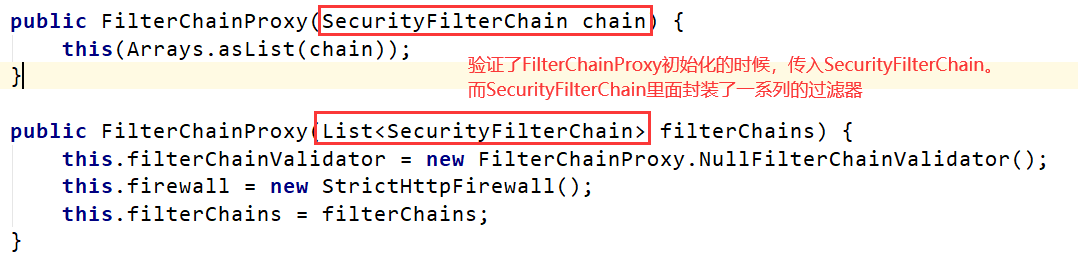
那我们就进入这个方法看看。我们在这个方法里面打断点测试看看。

我们重点观察它是如何封装过滤器的。



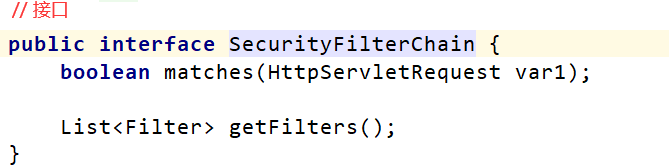
再看看getFilters方法怎么执行的?



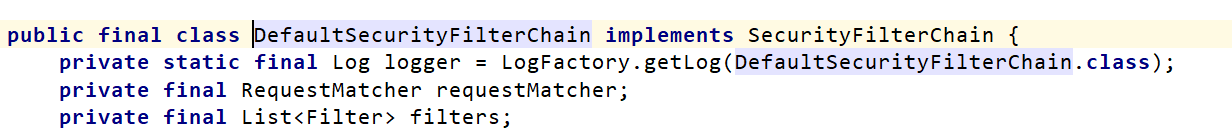


### 3.2.3 SecurityFilterChain

它是一个接口



其实现类是



# 4 SpringSecurity使用自定义认证页面

## 4.1 在SpringSecurity主配置文件中指定认证页面配置信息

### 4.1.1 配置springSecurity.xml配置文件

|  |
| --- |
| *<?***xmlversion="1.0"encoding="UTF-8"***?>* <**beansxmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:security="http://www.springframework.org/schema/security" xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd http://www.springframework.org/schema/security http://www.springframework.org/schema/security/spring-security.xsd"**>  *<!--资源不过滤-->* <**security:httpsecurity="none"pattern="/login.html"**></**security:http**> <**security:httpsecurity="none"pattern="/failer.html"**></**security:http**> *<!-- auto-config:true开启spring-security的自动配置 use-expressions="false"开启使用spring表达式的功能 -->* <**security:httpauto-config="true"use-expressions="true"**> *<!--指定login.jsp页面可以被匿名访问-->* <**security:intercept-urlpattern="/login.jsp"access="permitAll()"**/> *<!-- intercept-url:拦截的请求资源/\*\*所有的请求全部拦截 access:访问资源所需要的权限 -->* <**security:intercept-urlpattern="/\*\*"access="hasAnyRole('ROLE\_USER','ROLE\_ADMIN')"**></**security:intercept-url**> *<!-- 配置自定义表单 login-page：自定义的登录页面 login-processing-url:表单提交之后，交给那么action去处理springSecurity默认提供了一个叫/login的逻辑处理 username-parameter:值必须和登录表单里面的控件名称保持一致。 password-parameter值必须和登录表单里面的控件名称保持一致。 authentication-failure-forward-url:认证失败之后，需要跳转的页面 default-target-url:认证成功之后，需要跳转的页面 -->* <**security:form-loginlogin-page="/login.jsp"login-processing-url="/login" username-parameter="username"password-parameter="password" authentication-failure-forward-url="/failer.html" default-target-url="/index.jsp"** /> *<!--指定退出登录后跳转的页面-->* <**security:logoutlogout-url="/logout" logout-success-url="/login.jsp"**/></**security:http**> *<!--配置认证管理器-->* <**security:authentication-manager**> *<!--配置认证提供者-->* <**security:authentication-provider**> *<!--配置认证服务-->* <**security:user-service**> *<!-- 配置认证者 name用户名 password密码 authorities权限资源 -->* <**security:username="user"password="{noop}user"authorities="ROLE\_USER"**></**security:user**> <**security:username="admin"password="{noop}admin"authorities="ROLE\_ADMIN"**></**security:user**> </**security:user-service**> </**security:authentication-provider**> </**security:authentication-manager**> </**beans**> |

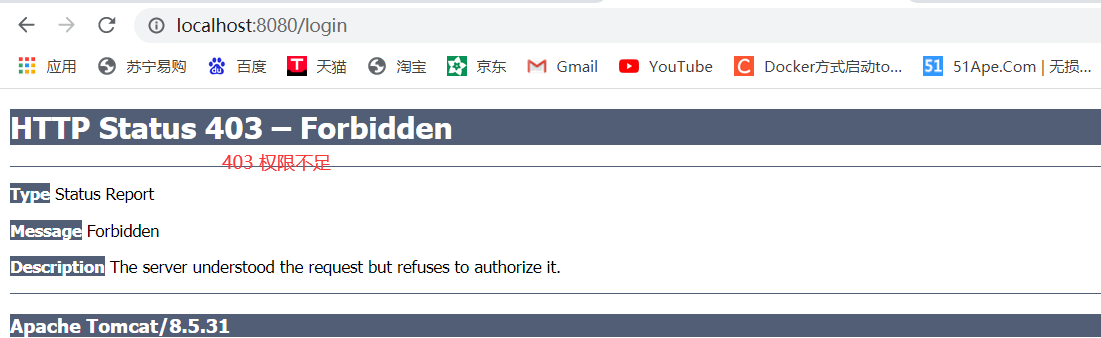
### 4.1.2 定义login.jsp

|  |
| --- |
| <%@**pagecontentType**="**text/html;charset=UTF-8**"**language**="**java**"%> <**html**> <**head**> <**title**>Title</**title**> </**head**> <**body**> <**formaction="/login"method="post"**> <**security:csrfInput**/> *<%--<security:csrfMetaTags/>--%><%--ajax提交用这个--%>* 用户名:<**inputtype="text"name="username"**><**br**> 密码:<**inputtype="password"name="password"**><**br**> <**inputtype="submit"value="提交"**> </**form**> </**body**> </**html**> |

再次启动项目后就可以看到自定义的酷炫认证页面了:



然后你开开心心的输入了用户名user，密码user，就出现了如下的界面：



403什么异常？这是SpringSecurity中的权限不足！这个异常怎么来的？还记得上面SpringSecurity内置认证页面源码中的那个\_csrf隐藏input吗？问题就在这了！



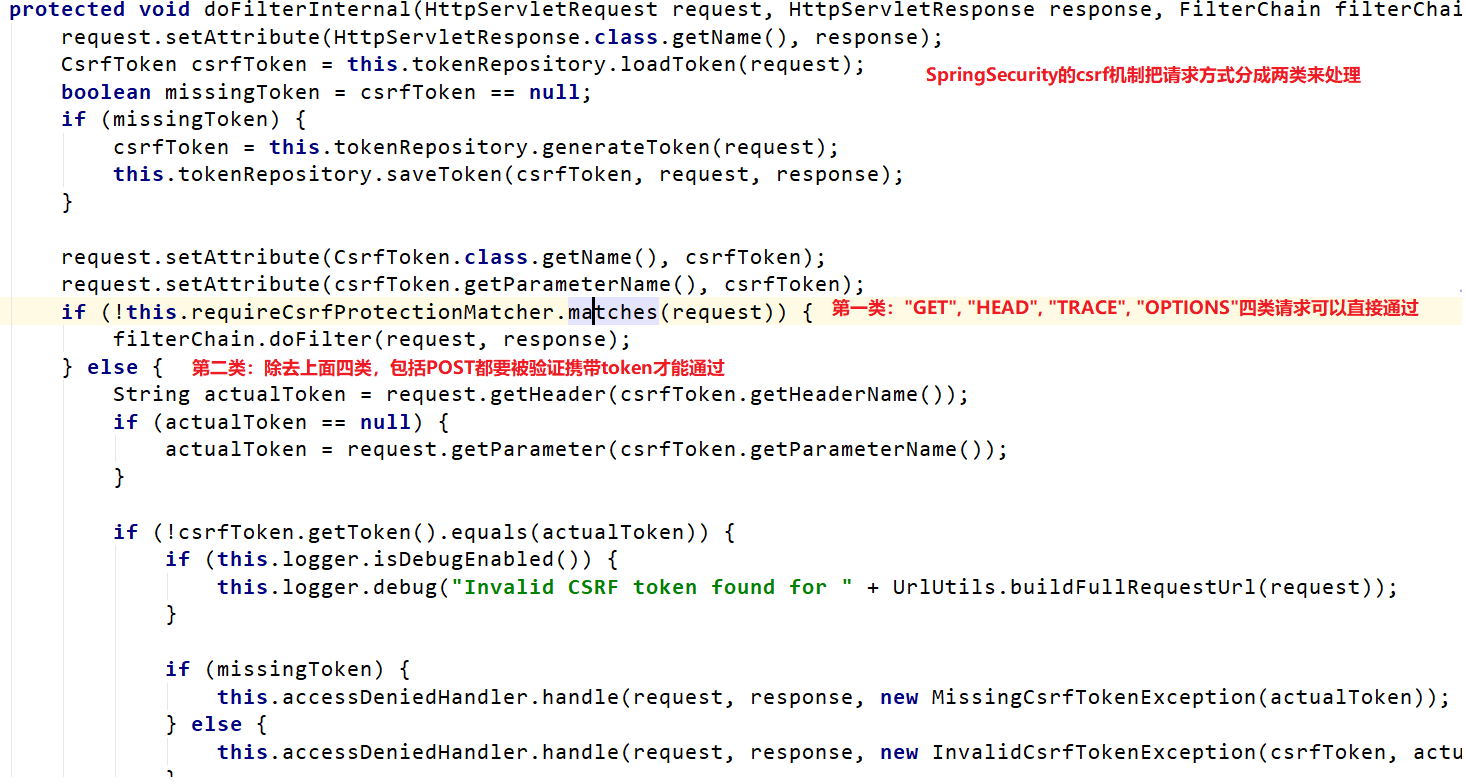
## 4.2 SpringSecurity的csrf防护机制

CSRF（Cross-siterequestforgery）跨站请求伪造，是一种难以防范的网络攻击方式。

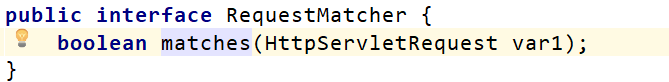


### 4.2.1 SpringSecurity中CsrfFilter过滤器说明

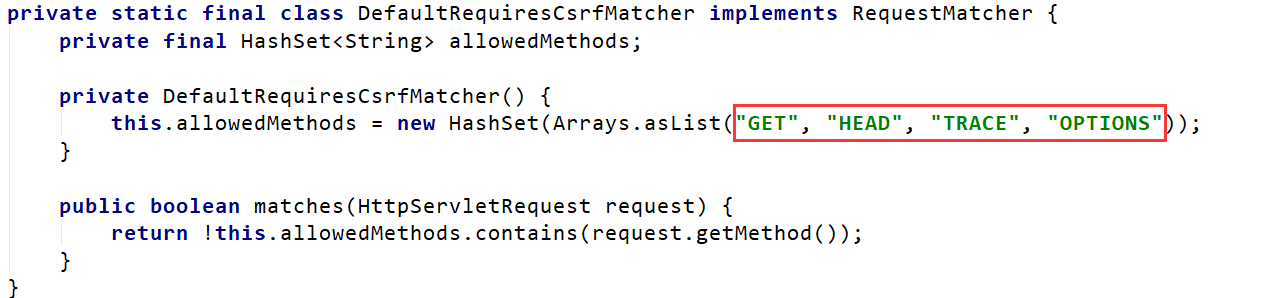
打开CsrfFilter过滤器源码：



如何查看GET HEAD ERTRACE OPTIONS方法直接放行?点击if语句里面的matches方法。



查看其实现类DefaultRequiresCsrfMatcher



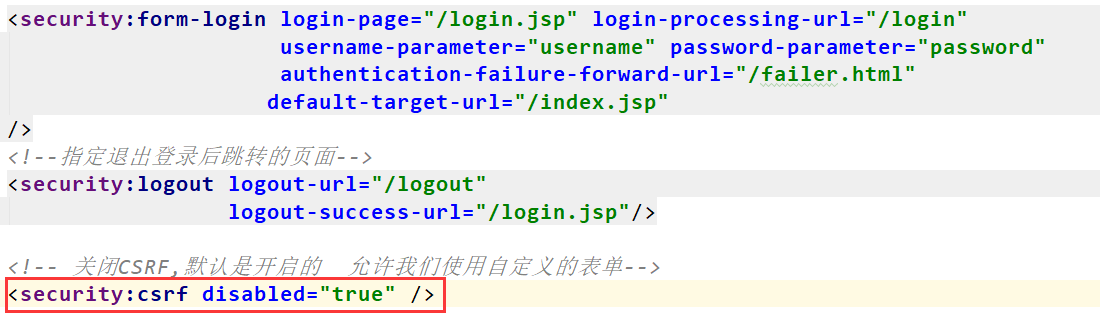
通过源码分析，我们明白了，自己的认证页面，请求方式为POST，但却没有携带token，所以才出现了403权限不足的异常。那么如何处理这个问题呢？

方式一：直接禁用csrf，不推荐。

方式二：在认证页面携带token请求。

### 4.2.2 禁用csrf防护机制

在SpringSecurity主配置文件中添加禁用crsf防护的配置。



但是这种方法一般不使用，禁用csrf过滤器，风险很大。

### 4.2.3 在认证页面携带token请求

先把禁用csrf过滤器的配置去掉。



再次请求，我们发现认证成功

### 4.2.4 退出认证

如何在主页上显示当前认证成功的用户呢?

在index.jsp页面上添加:

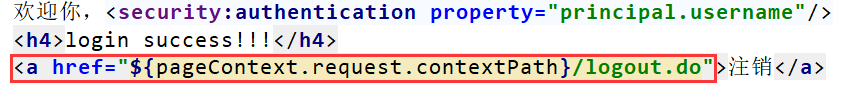
|  |
| --- |
| 欢迎你，<**security:authenticationproperty="principal.username"**/> |

#### 4.2.4.1 禁用csrf过滤器

|  |
| --- |
| *<!--关闭CSRF,默认是开启的允许我们使用自定义的表单-->* <**security:csrfdisabled="true"**/> |

|  |
| --- |
| <**ahref="${**pageContext.request.contextPath**}/logout.do"**>注销</**a**> |

最后定义如下:

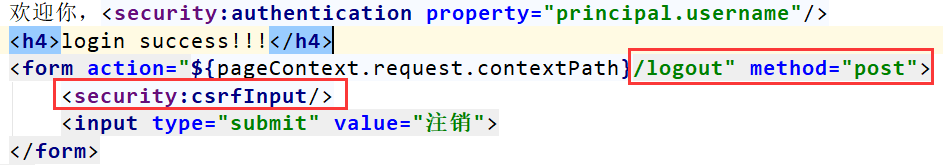


#### 4.2.4.2 没有禁用csrf过滤器

如果没有禁用csrf过滤器，那么上面这种方法就不好使了。我们需要使用表单的post方法进行退出。

|  |
| --- |
| <**formaction="${**pageContext.request.contextPath**}/logout"method="post"**> <**security:csrfInput**/> <**inputtype="submit"value="注销"**> </**form**> |

最后定义如下：



# 5 SpringSecurity使用数据库数据完成认证

## 5.1 认证流程分析

### 5.1.1 UsernamePasswordAuthenticationFilter

还记得我们在前面将的15个过滤器吧，这个过滤器就是其中之一。

先看主要负责认证的过滤器UsernamePasswordAuthenticationFilter，看重点，注意注释。

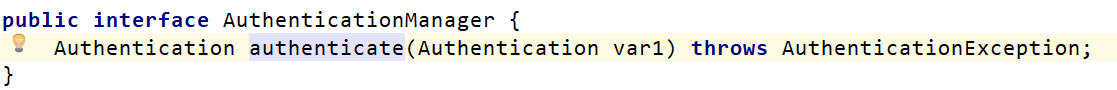
我们首先Ctrl+N在对话框里面输入这个过滤器的名字。



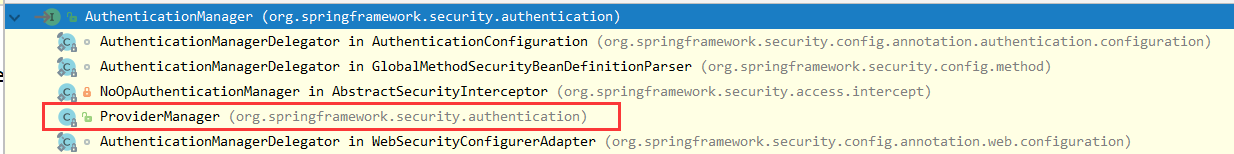
我们点击最后一行代码的authenticate方法。看看具体的认证流程是什么样子的。

### 5.1.2 AuthenticationManager

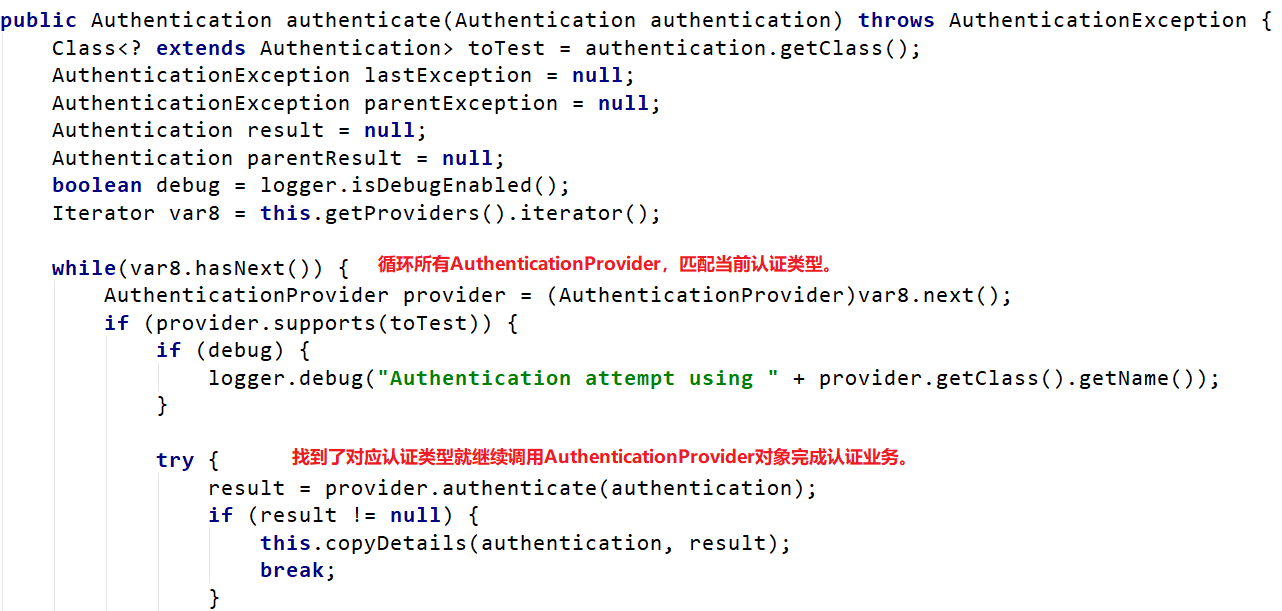
我们进入这个方法之后，发现这个方法是一个接口提供的。



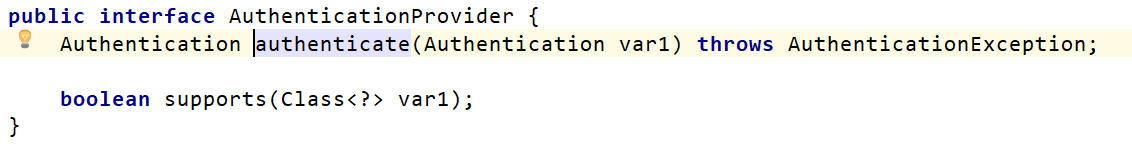
由上面源码得知，真正认证操作在AuthenticationManager里面！然后看AuthenticationManager的实现类ProviderManager：



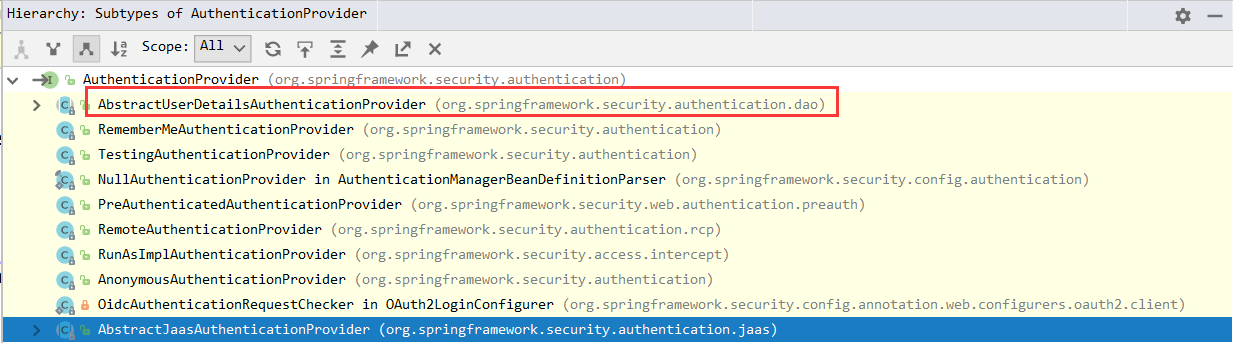




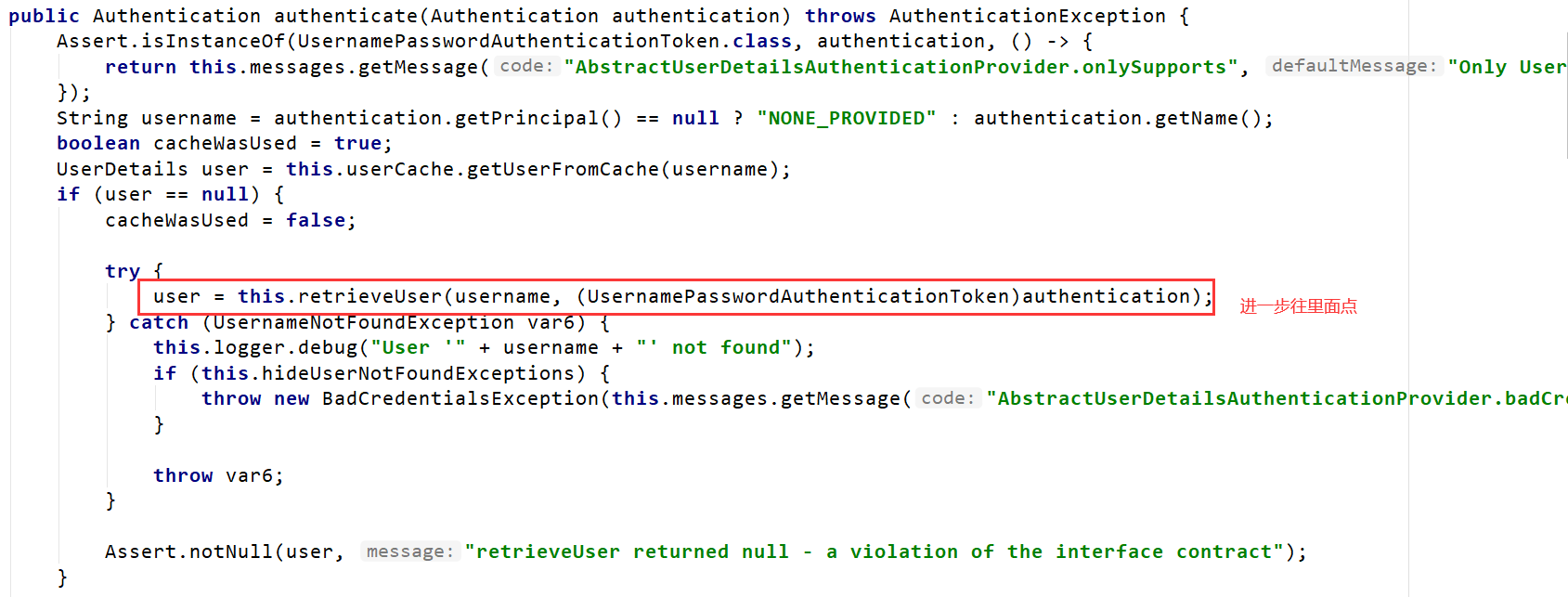
点击进入authenticate方法，进一步观察认证流程：



我们发现这个方法也是一个接口提供的方法。我们查看其实现类。



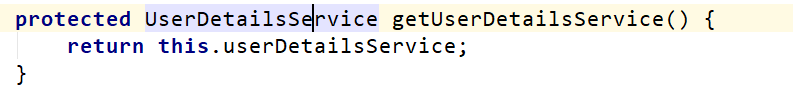
### 5.1.3 AbstractUserDetailsAuthenticationProvider



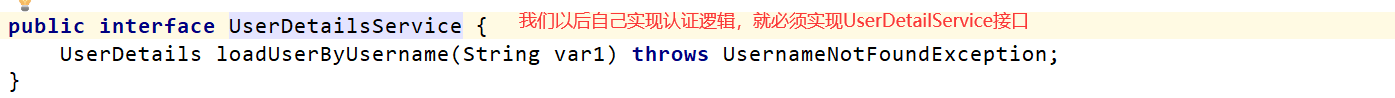
在DaoAuthenticationProvider这个类里面：



我们查看this.getUsrDetailService()方法。



这个方法的返回值是UserDetailService。点击UserDetailService查看源码：

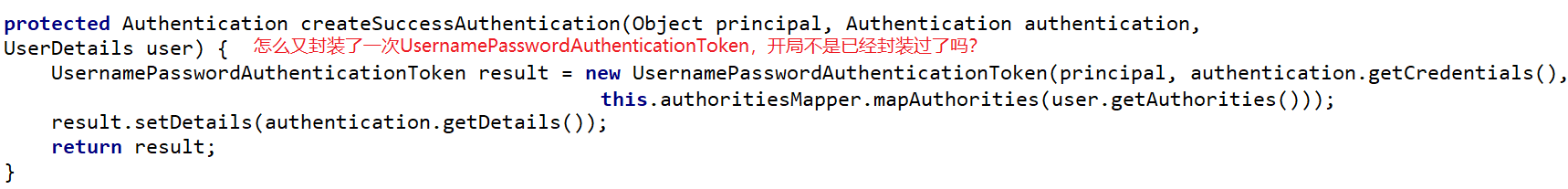


### 5.1.4 AbstractUserDetailsAuthenticationProvider中authenticate返回值

按理说到此已经知道自定义认证方法的怎么写了，但咱们把返回的流程也大概走一遍，上面不是说到返回了一个UserDetails对象对象吗？跟着它，就又回到了AbstractUserDetailsAuthenticationProvider对象中authenticate方法的最后一行了。



点进去看看：



来到UsernamePasswordAuthenticationToken对象发现里面有两个构造方法。

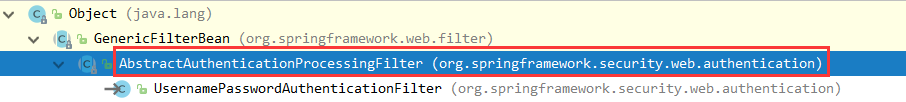
注意：这里传递的是三个参数，开局传递的是两个参数。



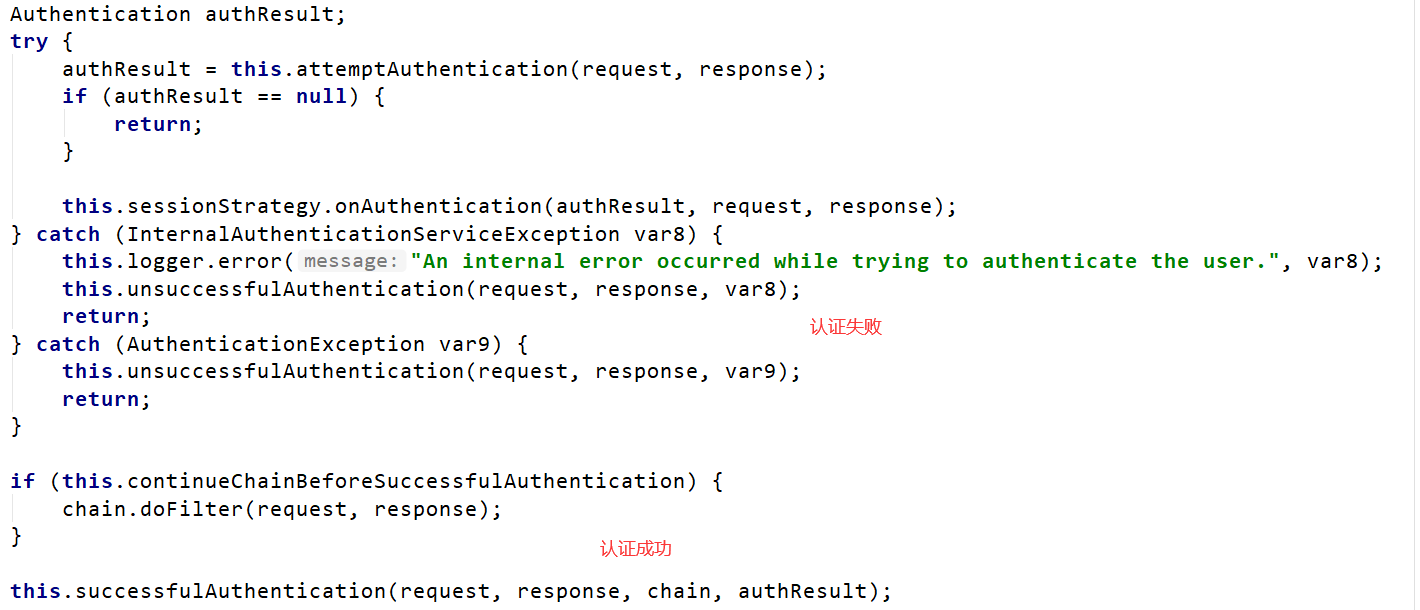
我们看看这个类的父类做了什么?点击super



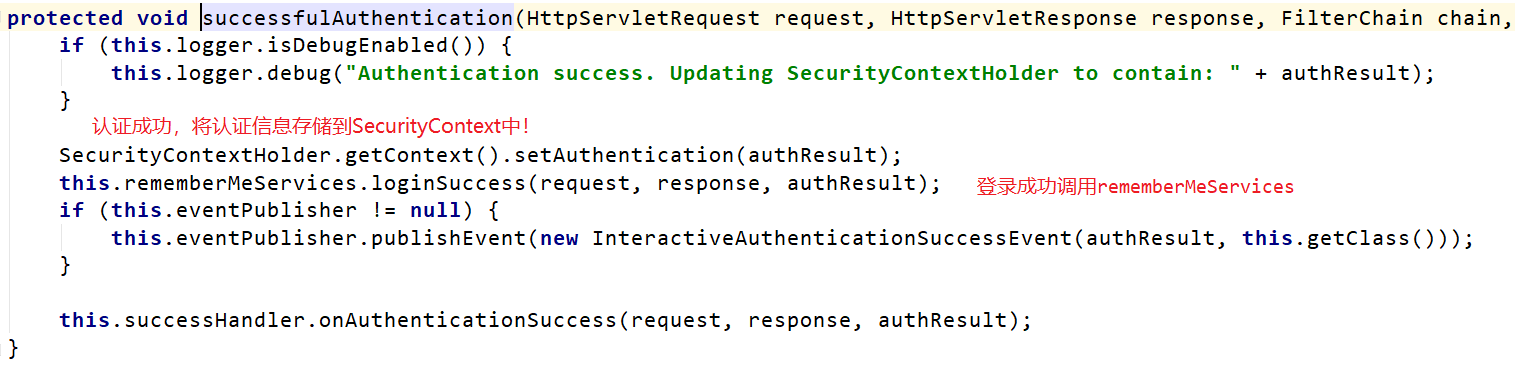
由此，咱们需要牢记自定义认证业务逻辑返回的UserDetails对象中一定要放置权限信息啊！可是这里面也没有doFilter呀？那就从父类找！



### 5.1.5 AbstractAuthenticationProcessingFilter



点击successfulAuthentication方法。

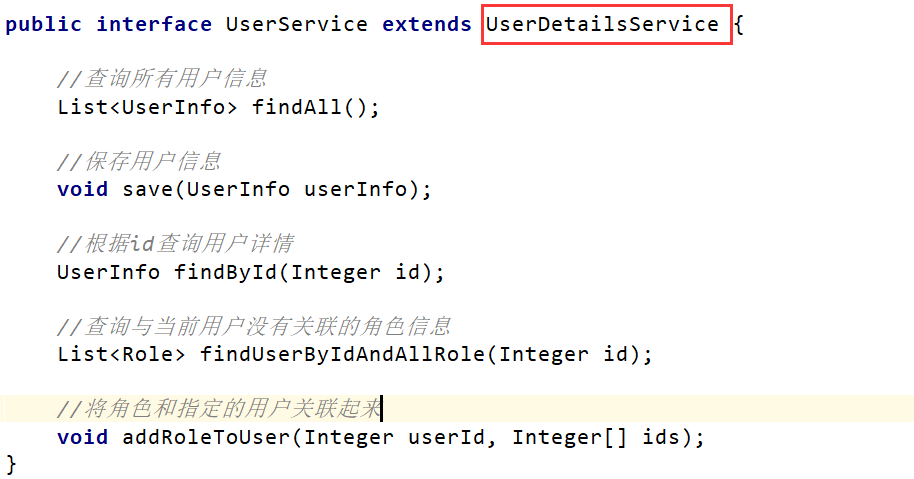


可见AbstractAuthenticationProcessingFilter这个过滤器对于认证成功与否，做了两个分支，成功执行successfulAuthentication，失败执行unsuccessfulAuthentication。在successfulAuthentication内部，将认证信息存储到了SecurityContext中。并调用了loginSuccess方法，这就是常见的“记住我”功能！此功能具体应用，咱们后续再研究！

## 5.2 初步实现认证功能

### 5.2.1 编写UserService接口

让我们自己的UserService接口继承UserDetailsService。



### 5.2.2 实现UserService接口

定义UserServiceImpl类，实现这个接口，编写loadUserByUsername业务。

|  |
| --- |
| @Service(**"userService"**) **public class** UserServiceImpl **implements** UserService {   @Autowired  UserDao **userDao**;   @Autowired  BCryptPasswordEncoder **bCryptPasswordEncoder**;   */\*\*  \* 根据用户名查询用户信息  \** ***@param username*** *\** ***@return*** *\** ***@throws*** *UsernameNotFoundException  \*/* **public** UserDetails loadUserByUsername(String username) **throws** UsernameNotFoundException {  UserInfo userInfo = **userDao**.findUserByName(username);  List<Role> roles = userInfo.getRoles();  User user = **new** User(userInfo.getUsername(),”{noop}”+userInfo.getPassword(),getAuthority(roles));  System.***out***.println(user);  **return** user;  }  *//封装角色信息，也就是认证信息* **public** Collection getAuthority(List<Role> roles){  List<SimpleGrantedAuthority> list = **new** ArrayList<SimpleGrantedAuthority>();  **for**(Role role : roles){  String roleName = role.getRoleName();  list.add(**new** SimpleGrantedAuthority(**"ROLE\_"** + roleName));  }  **return** list;  }  } |

### 5.2.3 在SpringSecurity主配置文件中指定认证使用的业务对象

|  |
| --- |
| *<!--配置认证管理器-->* <**security:authentication-manager**>  <**security:authentication-provider user-service-ref="userService"**>  </**security:authentication-provider**> </**security:authentication-manager**> |

## 5.3 加密认证

### 5.3.1 在IOC容器中提供加密对象

在springSecurity.xml里面配置加密对象。

|  |
| --- |
| *<!--配置加密工具类-->* <**bean id="bCryptPasswordEncoder" class="org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder"**></**bean**> |

### 5.3.2 修改认证方法

去掉{noop}。

|  |
| --- |
| @Service(**"userService"**) **public class** UserServiceImpl **implements** UserService {   @Autowired  UserDao **userDao**;   @Autowired  BCryptPasswordEncoder **bCryptPasswordEncoder**;   */\*\*  \* 根据用户名查询用户信息  \** ***@param username*** *\** ***@return*** *\** ***@throws*** *UsernameNotFoundException  \*/* **public** UserDetails loadUserByUsername(String username) **throws** UsernameNotFoundException {  UserInfo userInfo = **userDao**.findUserByName(username);  List<Role> roles = userInfo.getRoles();  User user = **new** User(userInfo.getUsername(),userInfo.getPassword(),userInfo.getStatus() == 0 ? **false** : **true**, **true**, **true**, **true**,getAuthority(roles));  System.***out***.println(user);  **return** user;  }  *//封装角色信息，也就是认证信息* **public** Collection getAuthority(List<Role> roles){  List<SimpleGrantedAuthority> list = **new** ArrayList<SimpleGrantedAuthority>();  **for**(Role role : roles){  String roleName = role.getRoleName();  list.add(**new** SimpleGrantedAuthority(**"ROLE\_"** + roleName));  }  **return** list;  } |

### 5.3.3 修改添加用户的操作

|  |
| --- |
| *//新增用户信息* **public void** save(UserInfo userInfo) {  *//加密之后的密码信息* String encode = **bCryptPasswordEncoder**.encode(userInfo.getPassword());  userInfo.setPassword(encode);  **userDao**.save(userInfo); } |

执行新增操作:

