**SpringSecurity**

# springSecurity 使用配置

## 设置 AuthenticationManager

@Bean

@Override

**public** AuthenticationManager **authenticationManagerBean**() **throws** Exception {

**return** **super**.authenticationManagerBean();

}

## 设置密码编译器

@Bean

**public** PasswordEncoder **passwordEncoder**() {

**return** **new** BCryptPasswordEncoder();

}

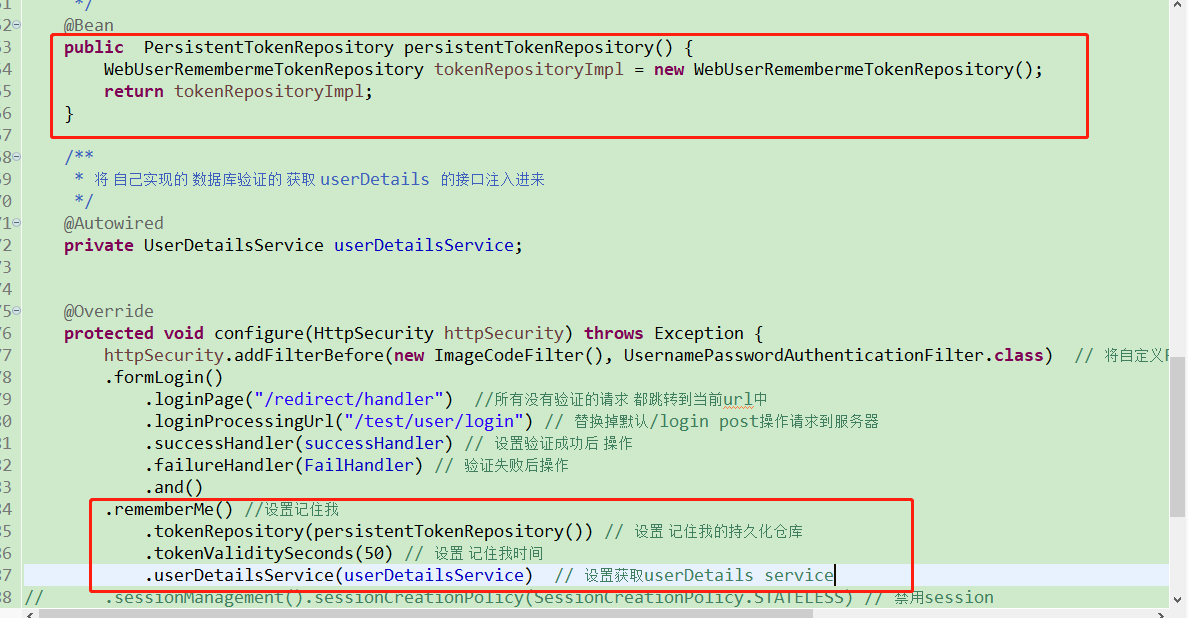
## 实现 UserDetailsService



如果有复杂操作，比如验证 密码是否过期，验证当前用户是否可用，是否被锁定 ，是否过期 可以返回值构造加上如下图UserDetails 说明

## 设置 rememberMe

设置 rememberMe 的持久化仓库（可以是数据库，可以是redis这里假想用的数据库和 默认实现JdbcTokenRepositoryImpl类似）



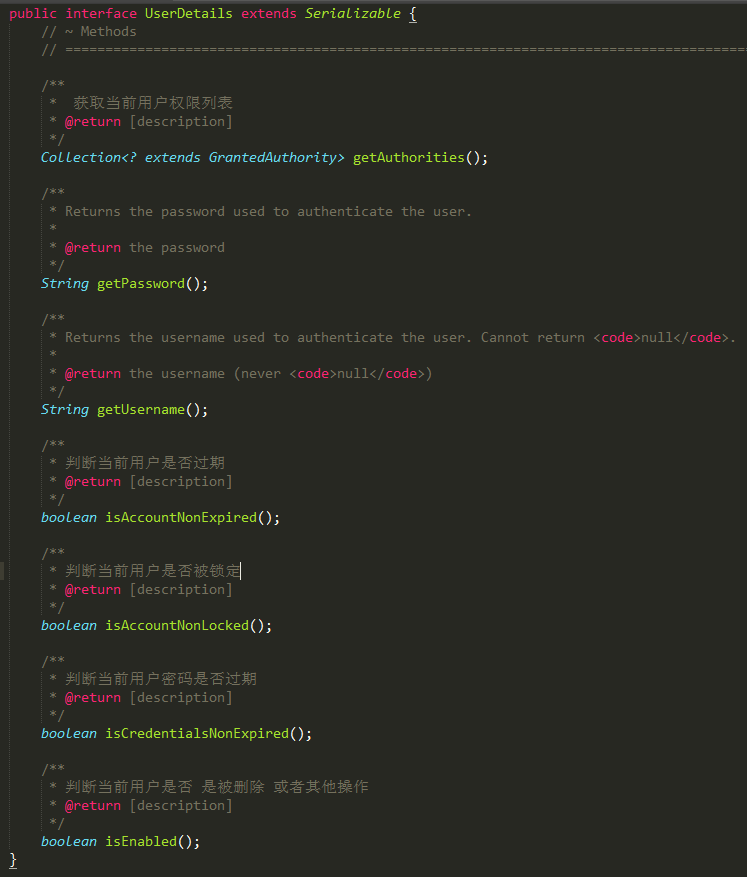
自定义 持久化仓库实现 如果有设置 内存库 比如redis 可以默认实现 当前接口将生成的 PersistentRememberMeToken存储在redis 中 可以参考 其中的默认实现方式



## UserDetailsService loadUserByUsername 说明

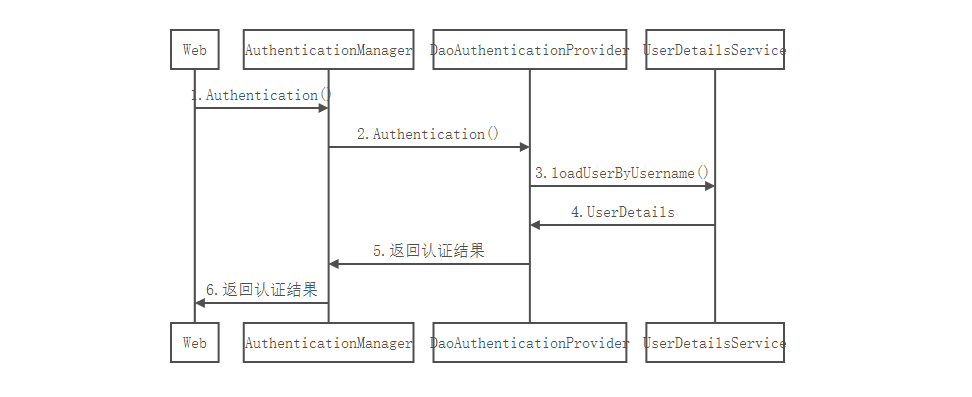
### UserDetails

返回值 UserDetails 接口实现，UserDetails 代码详解

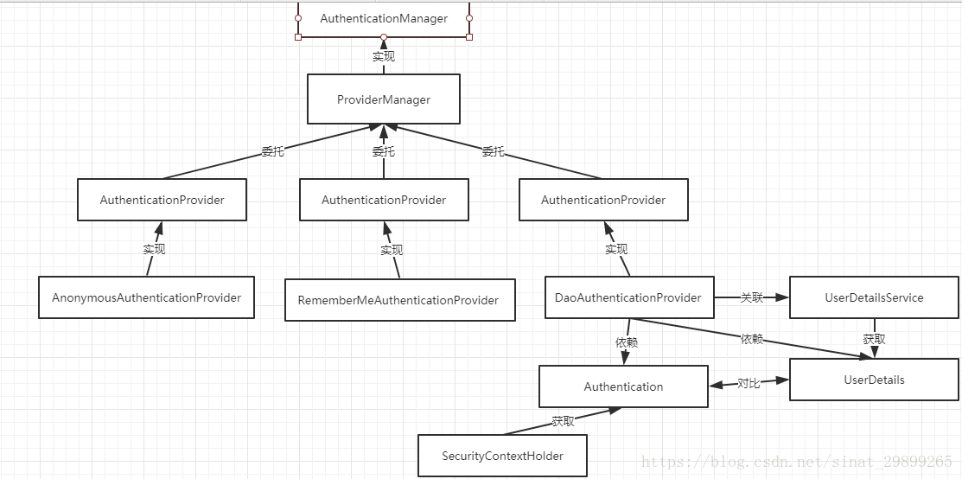


# SpringSecurity 认证源码

认证图

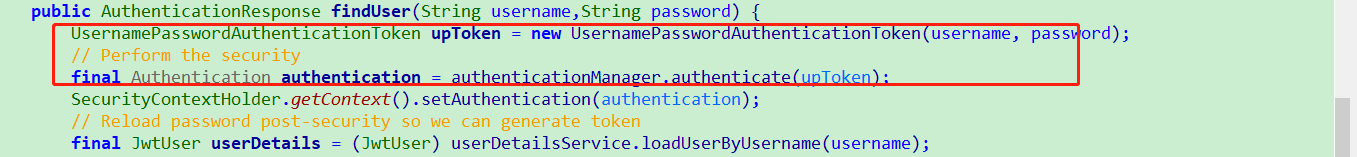


类图



## SpringSecurity 获取 Authentication 对象

Web login service代码截图



将login 页面中传来的 username password 封装成 UsernamePasswordAuthenticationToken

然后调用通过上文配置中说到的 AuthenticationManager 中的 authenticate 方法 根据类图表明为 ProviderManager 类中的authenticate 方法 然后调用 AbstractUserDetailsAuthenticationProvider 抽象类中的 authenticate 方法

**public** Authentication **authenticate**(Authentication authentication)

**throws** AuthenticationException {

// **获取当前用户信息**

String **username** = (authentication.getPrincipal() == **null**) ? "NONE\_PROVIDED"

: authentication.getName();

**boolean** **cacheWasUsed** = **true**;

// **从当前使用的内存中获取用户信息**

UserDetails **user** = **this**.userCache.getUserFromCache(username);

// **如果当前缓存中没有当前用户信息 据从数据库中进行认证**

**if** (user == **null**) {

cacheWasUsed = **false**;

**try** {

// **从数据库中获取当前用户信息 详细方法如下**

user = retrieveUser(username,

(UsernamePasswordAuthenticationToken) authentication);

}

**catch** (UsernameNotFoundException **notFound**) {

logger.debug("User '" + username + "' not found");

**if** (hideUserNotFoundExceptions) {

**throw** **new** BadCredentialsException(messages.getMessage(

"AbstractUserDetailsAuthenticationProvider.badCredentials",

"Bad credentials"));

}**else** {

**throw** notFound;

}

}

}

**try** {

// **判断当前用户是否被锁定 可用 过期 调用DefaultPreAuthenticationChecks 的//check方法**

preAuthenticationChecks.check(user);

// **判断当前用户密码是否为空，是否一致** **调用DaoAuthenticationProvider的方法**

additionalAuthenticationChecks(user,

(UsernamePasswordAuthenticationToken) authentication);

}

**catch** (AuthenticationException **exception**) {

**if** (cacheWasUsed) {

// There was a problem, so try again after checking

// we're using latest data (i.e. not from the cache)

cacheWasUsed = **false**;

user = retrieveUser(username,

(UsernamePasswordAuthenticationToken) authentication);

preAuthenticationChecks.check(user);

additionalAuthenticationChecks(user,

(UsernamePasswordAuthenticationToken) authentication);

}

**else** {

**throw** exception;

}

}

//**判断当前用户密码 过期 调用DefaultPostAuthenticationChecks 的** postAuthenticationChecks.check(user);

//**判断当前是否用缓存 如果有 将验证过的用户放缓存中**

**if** (!cacheWasUsed) {

**this**.userCache.putUserInCache(user);

}

Object **principalToReturn** = user;

**if** (forcePrincipalAsString) {

principalToReturn = user.getUsername();

}

**return** createSuccessAuthentication(principalToReturn, authentication, user);

}

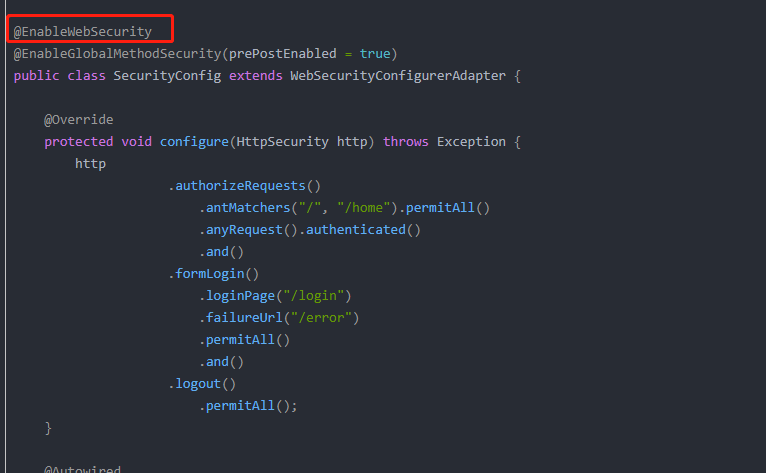
先从缓存中拿 如果没有 就调用 DaoAuthenticationProvider 中的 retrieveUser 方法

调用配置中 实现的 UserDetailsService的实现类 的 loadUserByUsername 方法获取 当前用户信息



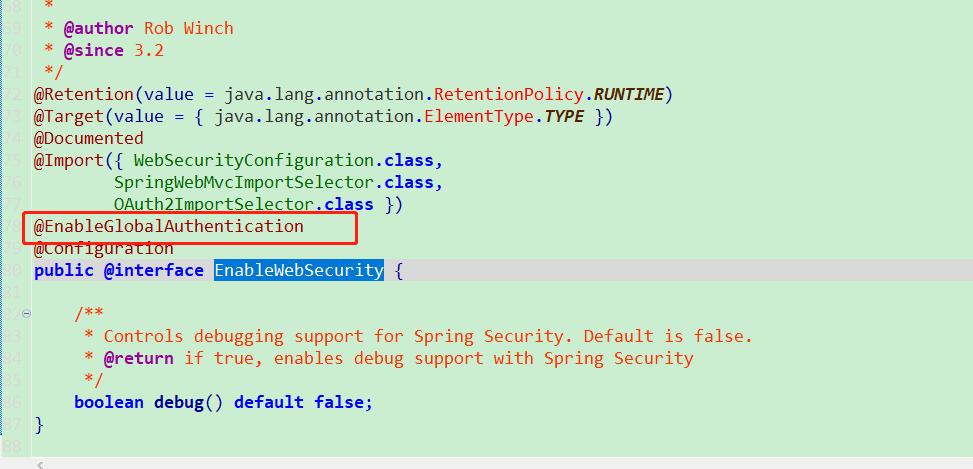
获取当前用户信息之后做检查

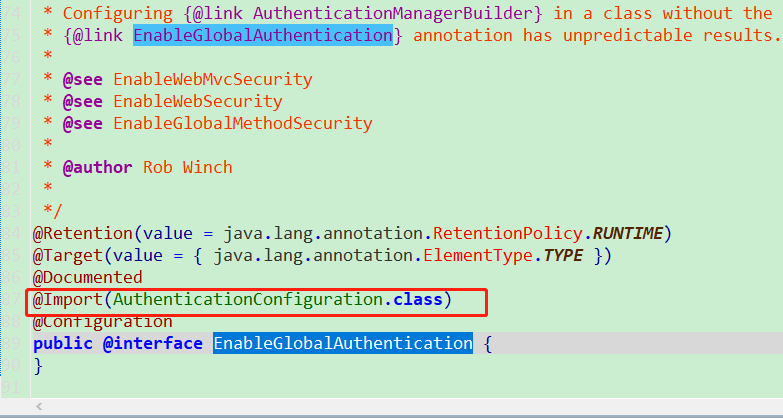
## SpringSercurity 获取AuthenticationManager 对象



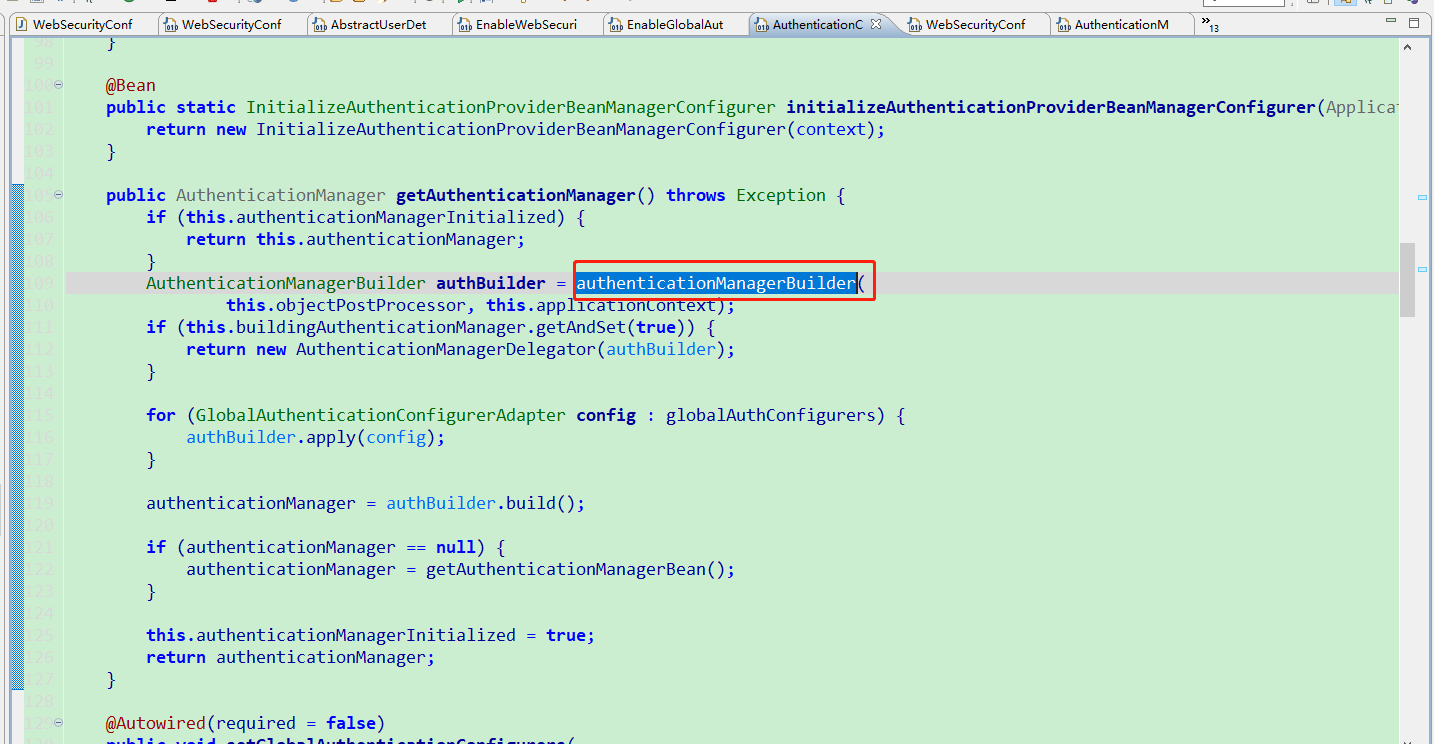
继承 WebSecurityConfigurerAdapter

@EnableWebSecurity 使用自定义配置

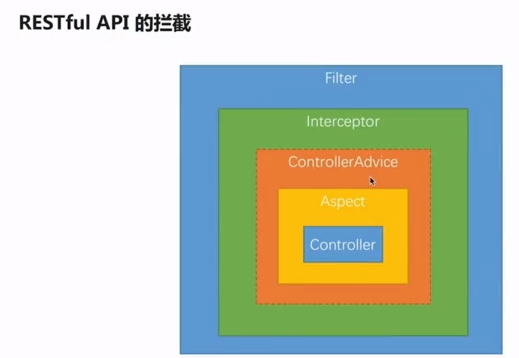








# Springboot拦截器



## Filter

Filter 中的 doFilter 是外部请求返回都会走的方法

### Springboot 加载外部filter



Springboot 自定义 Filter 在类上加注解 @component

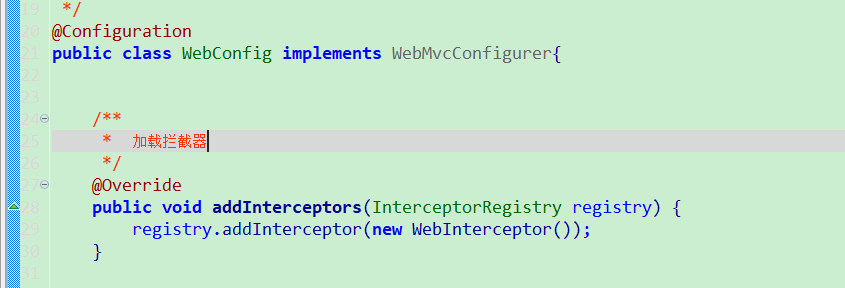
Filter 只能过滤 request ， response 从中拿到参数， 并不知道调用的哪个controller

## Springboot interceptor

### 一些解释

拦截器中 request 中有参数 但是 如果有 @RequestBody 那么 @RequestBody **修饰的变量没有值**





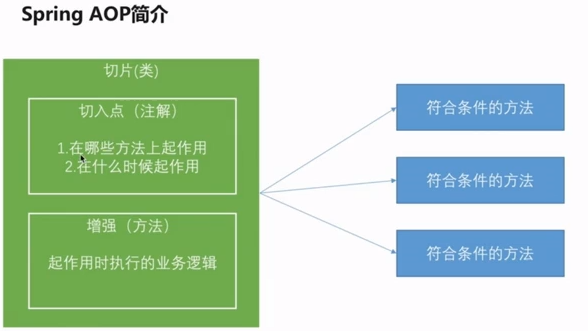
### 一些注意

如果定义全局异常处理器



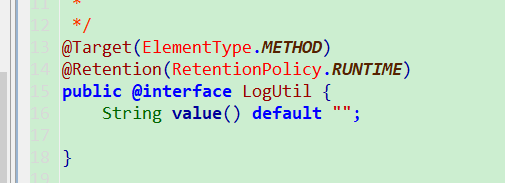
那么 拦截器中 afterCompletetion 中的异常信息为空

## 切片Aspect /aop

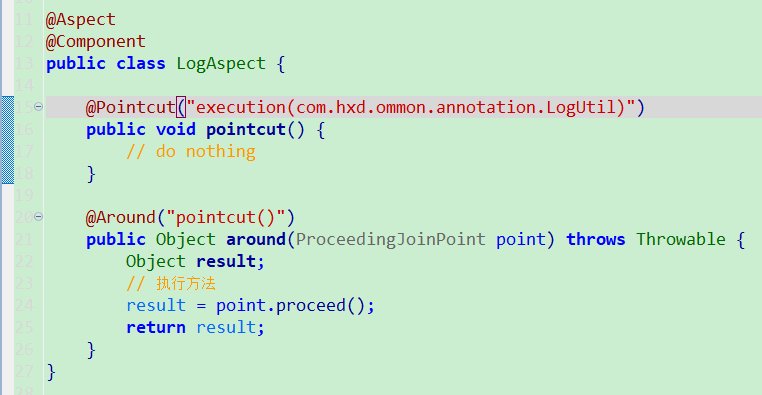


### 自定义注解切片

自定义注解



注解监视



### 监视已经有 controller

@Before 注解 方法执行前

@Around 注解 方法执行中



# Swagger

**根据代码自动生成文档**

## Swagger 引用

<!-- springfox-swagger2 -->

<dependency>

<groupId>io.springfox</groupId>

<artifactId>springfox-swagger2</artifactId>

<version>2.9.2</version>

</dependency>

<!-- springfox-swagger-ui -->

<dependency>

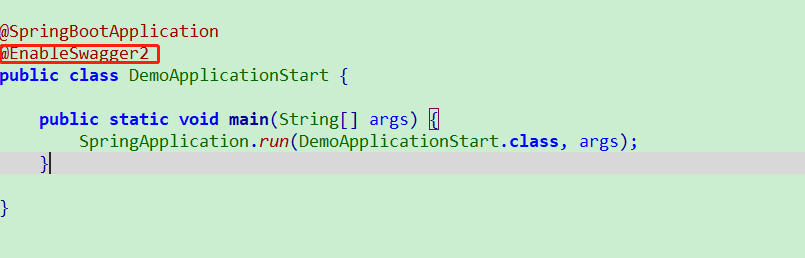
<groupId>io.springfox</groupId>

<artifactId>springfox-swagger-ui</artifactId>

<version>2.9.2</version>

</dependency>

启动注解上加：



## Swagger 常用注解



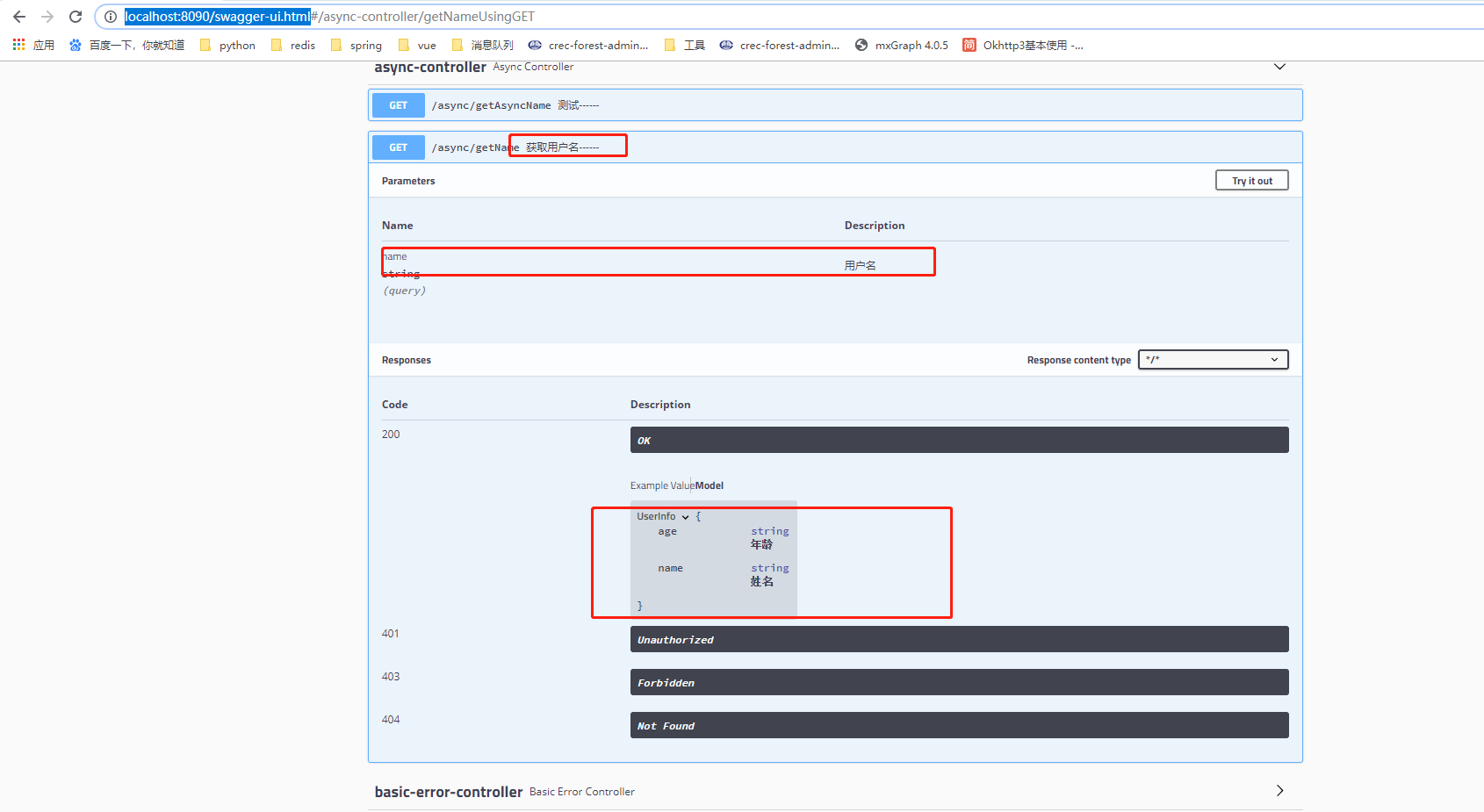
@ApiOperation("获取用户名------") 方法上 描述方法

@ApiParam("用户名") 参数上 描述参数用途

@ApiModelProperty("姓名") 放在 返回实体类属性上描述属性是什么

## Swaager 访问路径

[**http://ip:端口号/swagger-ui.html**](http://ip:端口号/swagger-ui.html)



# Wiremock

1. 搭建wireMock单机服务

1.1 下载jar包

服务jar包下载：http://repo1.maven.org/maven2/com/github/tomakehurst/wiremock-standalone/2.14.0/wiremock-standalone-2.14.0.jar

1.2 启动jar

java -jar wiremock-standalone-2.14.0.jar --port 9000

我在这里用9000端口启动

好了，看到上面的图案说明服务就搭建好了。

2. 向服务里注册Rest服务

2.1 导入依赖

<dependency>

<groupId>com.github.tomakehurst</groupId>

<artifactId>wiremock</artifactId>

</dependency>

2.2 写一个简单的模拟Rest

/\*\*

\* Created by Fant.J.

\*/

public class MockServer {

public static void main(String[] args) {

//通过端口连接服务

WireMock.configureFor(9000);

//清空之前的配置

WireMock.removeAllMappings();

//get请求

WireMock.stubFor(WireMock.get(WireMock.urlPathEqualTo("/user/1"))

.willReturn(WireMock.aResponse()

//body里面写 json

.withBody("{\"username\":FantJ}")

//返回状态码

.withStatus(200)));

}

}

运行这个main方法。

然后访问 <http://127.0.0.1:9000/user/1>

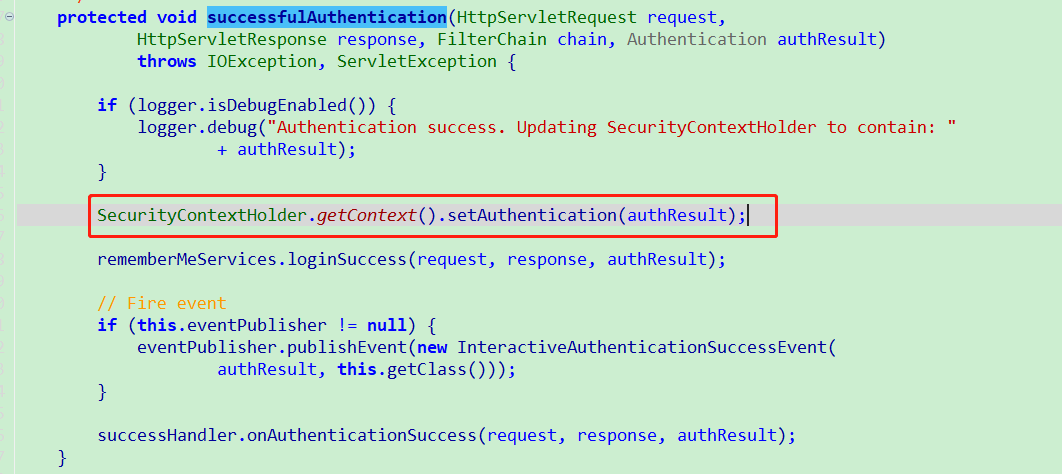
# SpringSecurity

## 认证 （你是谁 、 登陆）

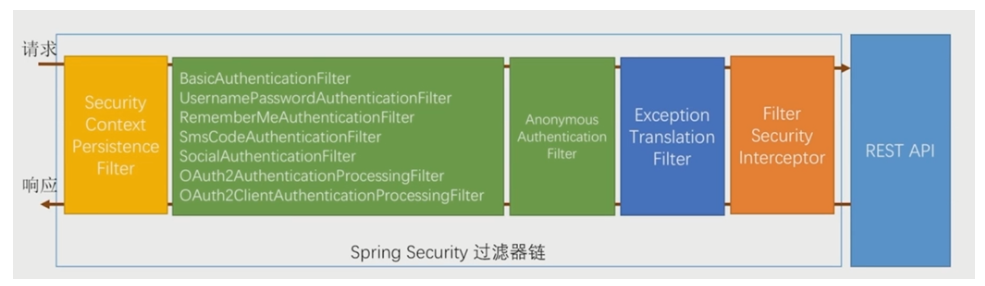
**SpringSecurity 的认证过程就是一个获取 Authentication 的过程；**

**用户发起登陆请求（一个不需要拦截认证的请求，带有登录名密码的请求）到服务器，服务器接收到请求后将用户名密码封装成一个UsernamePasswordAuthenticationToken对象，然后通过UsernamePasswordAuthenticationToken对象， 获取用户名，然后根据用户名通过自己实现的UserDetailsService 接口的loadUserByUsername 方法获取当前用户详细信息，密码、是否被锁定、是否存在、是否过期等信息； 然后将密码拿出来，对比之前封装的UsernamePasswordAuthenticationToken对象的密码，如果查询出来的这个用户，状态正常，密码和之前传入密码匹配，那么将查询出来的用户信息封装成一个通过的Authentication对象。**

**获取通过的Authentication对象之后，会通过以下方法将之存在ThreadLocal（线程级别变量）中**



### SpringSecurity认证过滤器链



1.SecurityContextPersistenceFilter 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；

2.UsernamePasswordAuthenticationFilter extends AbstractAuthenticationProcessingFilter 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；

3.FilterSecurityInterceptor 是用于保护Http 资源的，它需要一个AccessDecisionManager和一个AuthenticationManager 的引用。它会从 SecurityContextHolder 获取 Authentication，然后通过 SecurityMetadataSource 可以得知当前请求是否在请求受保护的资源。对于请求那些受保护的资源，如果Authentication.isAuthenticated()返回false或者FilterSecurityInterceptor

的alwaysReauthenticate 属性为 true，那么将会使用其引用的 AuthenticationManager 再认证一次，认证之后再使用认证后的 Authentication 替换 SecurityContextHolder 中拥有的那个。然后就是利用 AccessDecisionManager 进行权限的检查；

4.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。--- 如果捕获到的是 AuthenticationException，那么将会使用其对应的 AuthenticationEntryPoint 的commence()处理。在处理之前，ExceptionTranslationFilter先使用 RequestCache 将当前的HttpServerletRequest的信息保存起来，以至于用户成功登录后可以跳转到之前的界面；

--- 如果捕获到的是 AccessDeniedException，那么将视当前访问的用户是否已经登录认证做不同的处理，如果未登录，则会使用关联的 AuthenticationEntryPoint 的 commence()方法进行处理，否则将使用关联的 AccessDeniedHandler 的handle()方

### 多请求之间 认证共享

#### AbstractAuthenticationProcessingFilter类分析

验证当前路径是否需要进行验证 下图

**if** (!requiresAuthentication(request, response)) {

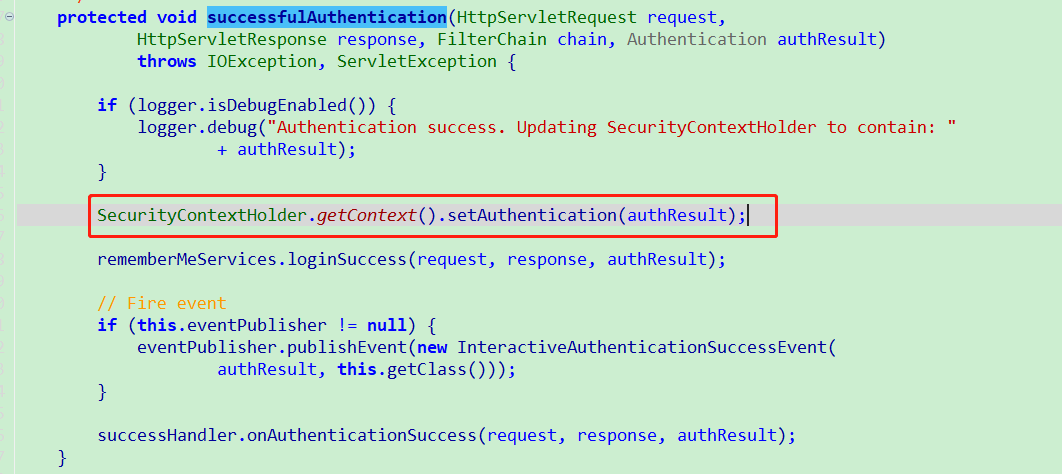
chain.doFilter(request, response);

**return**;

}



SuccessfunAuthentication 方法是当前用户认证完成以后调用的方法



其中 successHandler 用的是SavedRequestAwareAuthenticationSuccessHandler 类， 如果要自定义 成功操作 可以继承该类进行处理；

当验证成功后 调用的；

rememberMeServices.loginSuccess(request, response, authResult);

默认会调用一个空实现

**new** NullRememberMeServices();

**但是如果有在config文件中 配置HttpSecurity参数配置 rememberMe 功能的话会set**

PersistentTokenBasedRememberMeServices

代替默认实现



### 设置记住我机制

RememberMeAuthenticationFilter 记住我机制Filter



**判断当前 threadLocal 中是否存在已经验证过的 Authentication 如果不存在 从cookie 中获取**

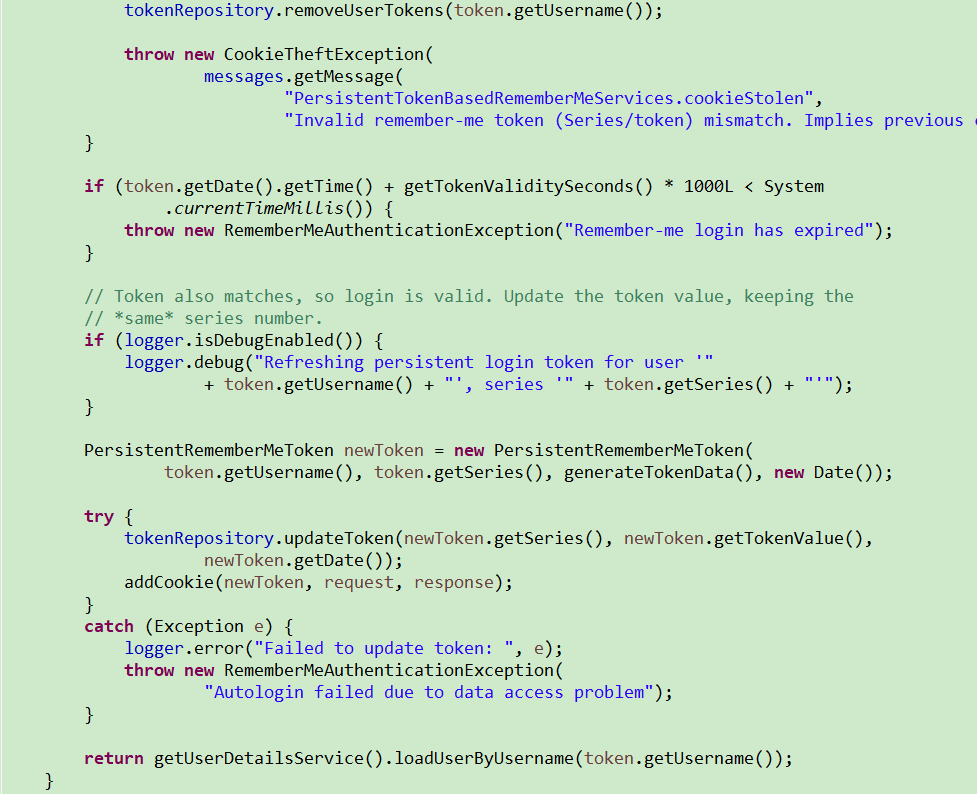


**如果 cookie 不为空 并且长度大于2 那就从配置的** tokenRepository **获取值**

**然后根据取回数据进行 比较，验证时间等等、、、**



**验证成功之后 将数据库中 记录信息更新，在前台新生成cookie 然后 根据cookie 中取出的userName 在调用自己实现的 UserDetailsService 的**loadUserByUsername 方**法 获取 UserDetails**



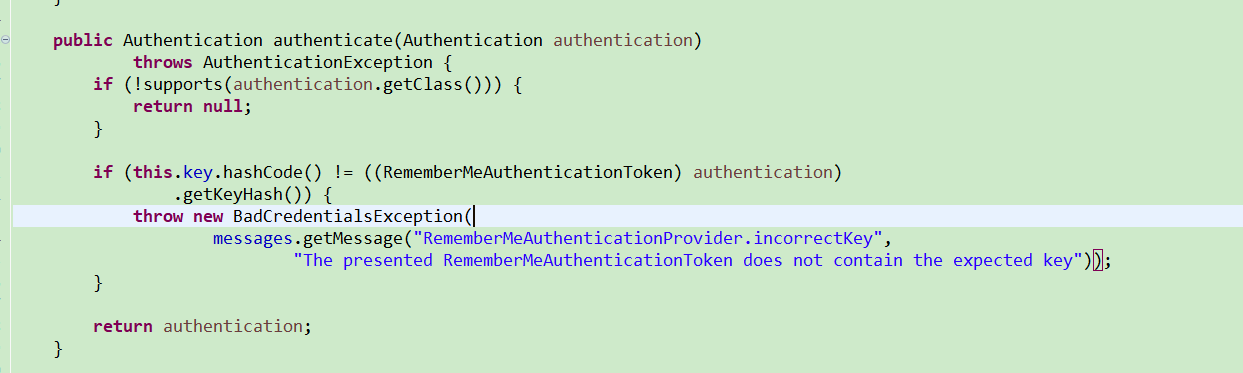
**然后将取出来的UserDetail 信息进行check 封装成 Authentication 对象**



**将取回来的 Authentication 对象在进行验证 放到 threadLocal 中 已便线程验证**



**RememberMe 的authenticate方法**



**请求时候name为remember-me 源代码解析**



### 自定义短信验证登陆

自定义手机短信 、邮箱验证 很类似于表单认证，实现的一些思路是

1. 根据手机号生成短信验证码，将验证码根据用户唯一标识存放在数据库或者内存中；**验证码持久化**
2. 填写验证码，发送请求； 服务器端拦截带有验证码的URL，将带有手机号或者邮箱号并且填写过的验证码进行拦截封装； **Filter**
3. 验证，将用户名验证码跟持久化的验证码进行对比，如果 对比成功，组装成一个验证过的Authentication对象， 如果验证失败抛出异常重新验证
4. 将验证过的Authentication对象放在LocalThread 中以便多请求之间的共享

#### 验证码生成持久化

实现方式 内存存储验证码信息；

1. 自定义内存接口规范。
2. 实现接口规范 在这用的是线程安全的ConCurrentHashMap 实现
3. 将手机号或者邮箱号 当做key 验证码当做value 放在内存中

#### 自定义拦截器

实现目标 将url 中的 手机号 跟验证码 拦截 封装成自定义的token对象，然后验证

**代码实现：**

**public** **class** PeripheralAuthenticationFilter **extends** AbstractAuthenticationProcessingFilter {

// ~ Static fields/initializers

// =====================================================================================

**public** **static** **final** String ***SPRING\_SECURITY\_FORM\_USERNAME\_KEY*** = "peripheralName";

**public** **static** **final** String ***SPRING\_SECURITY\_FORM\_MSGCODE\_KEY*** = "msgcode";

**private** String peripheralNameParameter = ***SPRING\_SECURITY\_FORM\_USERNAME\_KEY***;

**private** String peripheralMsgCodeParameter = ***SPRING\_SECURITY\_FORM\_MSGCODE\_KEY***;

**private** **boolean** postOnly = **true**;

// ~ Constructors

// ===================================================================================================

/\*\*

\* 设置默认登陆 路径

\*/

**public** PeripheralAuthenticationFilter() {

**super**(**new** AntPathRequestMatcher("/peripheral/login", "POST"));

}

// ~ Methods

// ========================================================================================================

/\*\*

\* 从request 中获取参数信息 然后再去验证

\*/

**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 peripheralName = obtainParameter(request, peripheralNameParameter);

String msgCode = obtainParameter(request, peripheralMsgCodeParameter);

**if** (peripheralName == **null**) {

peripheralName = "";

}

**if**(msgCode == **null**) {

msgCode = "";

}

PeripheralAuthenticationToken authRequest = **new** PeripheralAuthenticationToken(

peripheralName, msgCode);

// Allow subclasses to set the "details" property

setDetails(request, authRequest);

**return** **this**.getAuthenticationManager().authenticate(authRequest);

}

/\*\*

\* 从request 中获取 当前登陆设备名

\* **@param** request

\* **@return**

\*/

**protected** String obtainParameter(HttpServletRequest request, String parameter) {

**return** request.getParameter(parameter);

}

/\*\*

\* 将当前信息封装到对象中

\*/

**protected** **void** setDetails(HttpServletRequest request,

PeripheralAuthenticationToken authRequest) {

authRequest.setDetails(authenticationDetailsSource.buildDetails(request));

}

/\*\*

\* 设置设备名称

\*/

**public** **void** setUsernameParameter(String peripheralNameParameter) {

Assert.*hasText*(peripheralNameParameter, "Username parameter must not be empty or null");

**this**.peripheralNameParameter = peripheralNameParameter;

}

/\*\*

\* 设置是否只有 post 请求能通过

\*/

**public** **void** setPostOnly(**boolean** postOnly) {

**this**.postOnly = postOnly;

}

**public** **final** String getPeripheralNameParameter() {

**return** peripheralNameParameter;

}

}

#### 自定义实现Token对象

实现目标 实现Authentication接口将自定义信息存放；主要是为了区分验证流程

根据ProviderManager 的authenticate 方法的 support 支持判断取用哪个provider是根据传入的 Authentication 实现对象进行区分；

/\*\*

\*

\*/

**package** com.hxd.security.core.authentication;

**import** java.util.Collection;

**import** org.springframework.security.authentication.AbstractAuthenticationToken;

**import** org.springframework.security.core.GrantedAuthority;

**import** org.springframework.security.core.SpringSecurityCoreVersion;

/\*\*

\* **@author** hxd

\*

\* date 2019年11月12日

\*

\*/

**public** **class** PeripheralAuthenticationToken **extends** AbstractAuthenticationToken {

**private** **static** **final** **long** ***serialVersionUID*** = SpringSecurityCoreVersion.***SERIAL\_VERSION\_UID***;

// 当前登陆设备信息

**private** **final** Object peripheralName;

// 当前登录验证码

**private** **final** String msg;

// ~ Constructors

// ===================================================================================================

/\*\*

\* 设置当前登陆 设备信息

\* **@param** principal

\*/

**public** PeripheralAuthenticationToken(Object principal, String msg) {

**super**(**null**);

**this**.peripheralName = principal;

**this**.msg = msg;

setAuthenticated(**false**);

}

/\*\*

\*

\* **@param** principal 当前外设名称 手机 邮件 等等

\* **@param** authorities 当前登陆人获得权限

\* 设置当前Authentication 已经认证通过

\*/

**public** PeripheralAuthenticationToken(Object principal, String msg,

Collection<? **extends** GrantedAuthority> authorities) {

**super**(authorities);

**this**.msg = msg;

**this**.peripheralName = principal;

**super**.setAuthenticated(**true**); // must use super, as we override

}

**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** Object getCredentials() {

**return** **this**.msg;

}

@Override

**public** Object getPrincipal() {

**return** **this**.peripheralName;

}

}

#### 自定义验证器

主要验证当前filter 拦截封装的对象 是否能通过验证 验证通过之后将封装成一个验证通过的Authentication对象



#### 自定义加载



#### 自定义内存

可以参考spring中内存写法

## 授权（你能干什么）

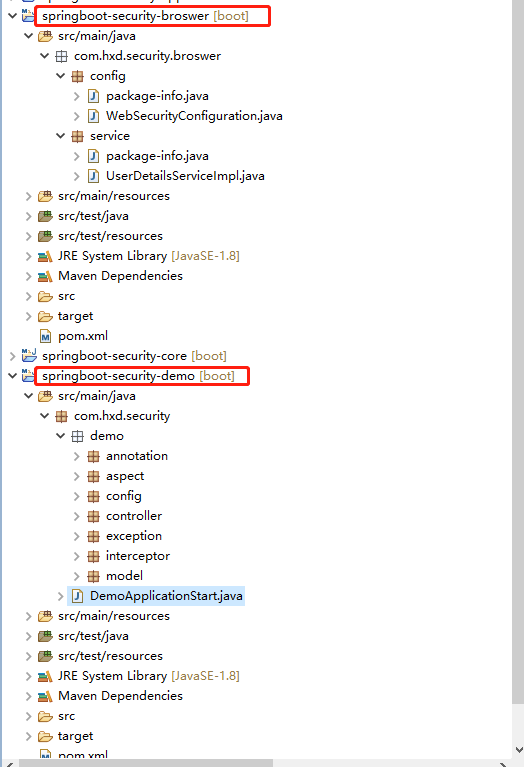
## 攻击防护（防止伪造身份）

## 单点登录

# SpringBoot 一些使用心得

## 工程之间引用

假如有两个工程 一个工程是web工程 demo， 一个是负责管理一些共通文件以及jar包的工程的工程browser， demo 工程引用 browser工程 图如下；



如果说demo工程，想将browser工程中的配置生效，那么demo工程的启动项要放在com.hxd.security下，这样browser工程下的com.hxd.security package的工程也就加载了；

或者说，如果DemoApplicationStart 启动项在com.hxd.security.demo package 下，那么就需要如图所示，将别的包扫描到当前工程



## 一些注意事项

在写实体类的时候，即使是boolean对象的也不能有is 或者has 修饰 要不spring 注入不进去

## 工程间共通抽出

将共通提出工程共通引用提出共通

## Spring 接口发现机制

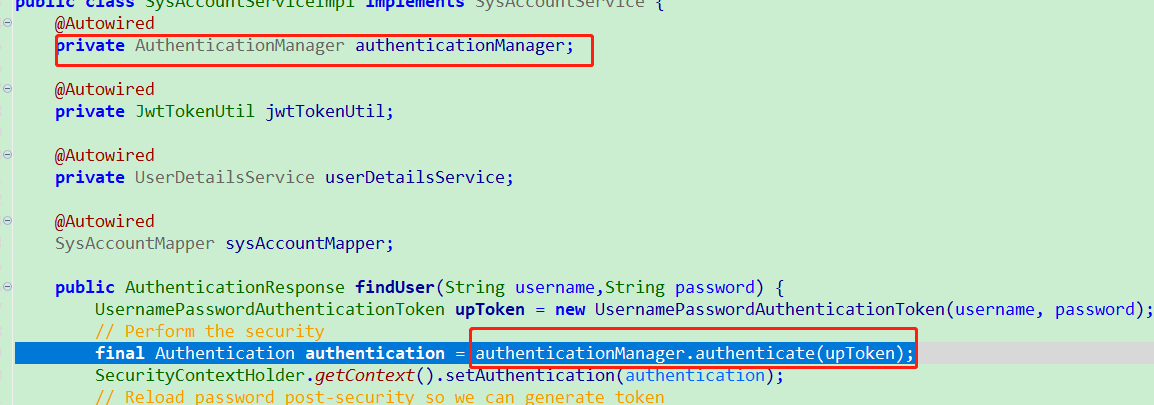
@Autowired

**private** Map<String,Interface> mapInterfaceimpls;

用Autowired 修饰后的 map 会将当前结构的所有实现类进行查找，以类名为key 实现为value

## 种树项目 认证过程刨析

给”/user/login”一个无需认证权限，然后发送一个带有 password，username 的请求 ，获取当前请求参数 封装为UsernamePasswordAuthenticationToken 对象；然后通过 AuthenticationManager 的authenticate 方法进行验证；



如果没有验证通过，通过抛出异常的方式进行抛出，验证通过后 将用户信息编码为一个String串 返回前台；

然后再次请求的时候 请求会将这个String串截取，将它反编译为对象 将用户名密码取出来封装为UsernamePasswordAuthenticationToken对象放入ThreadLocal中进行验证