# Shiro简介

## 知识大纲

shiro学习知识点：！！！！！！

-1）自定义Realm实现身份认证（Authentication）、授权（Authorization）

-2）使用ehcache将权限菜单缓存（10分钟过期）

-3）密码加密实现，并实现登录次数限制。（通过继承HashedCredentialsMatcher类，并使用EhCache记录登录重试次数和超时时间）

-4）使用自定义filter：

验证码过滤器实现（继承AccessControlFilter）

登录过滤器实现（继承FormAuthenticationFiltershiro）

设置强制修改密码（继承FormAuthenticationFiltershiro）

-5）集群时session共享：

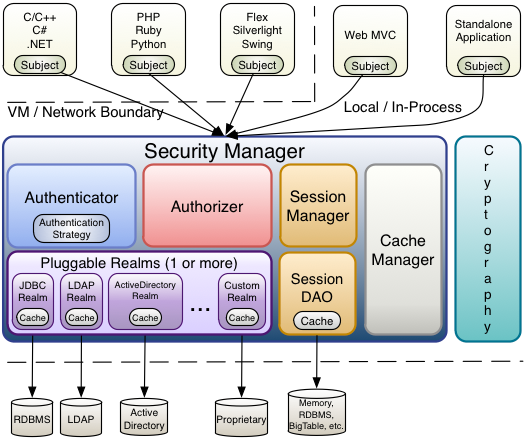
redis+shiro实现，重写shiro的SessionManager和SessionDao

使用Spring Session进行session共享

## 概述

Shiro是Apache下的一个开源项目，我们称之为Apache Shiro。它是一个很易用于Java项目的的安全框架，提供了身份认证（Authentication）、授权（Authorization）、加密、会话管理等功能，与 Spring Security 一样都是做一个权限的安全框架，但是与Spring Security 相比，在于 Shiro 使用了比较简单易懂易于使用的授权方式。

完整架构图：



基本使用：

参见spring-boot-test的spring-boot-shiro项目

## Shiro核心组件

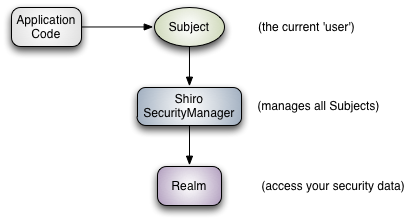
三大核心组件：

-1）Subject（主体）：所有Subject都绑定到SecurityManager，与Subject的所有交互都会委托给SecurityManager。

-2）SecurityManager（安全管理器）：Shiro的核心，管理所有的Subject，所有与安全相关的操作都与它有关，主要进行“认证-验证用户身份”和“授权-用户对应角色权限”两大操作。类似SpringMVC的DispatcherServlet类。属性：realm、cacheManager等

-3）Realm（域）：用于身份验证（Authentication）和授权（Authorization），从这里获取安全数据（如用户、角色、权限等）。

Shiro的核心通过Filter实现，跟SpringMVC中的DispatcherServlet一样。Shiro通过URL规则进行过滤和权限校验，所以我们需要定义一系列关于URL的规则和访问权限。



## Realm

### 身份认证和授权方法

UserRealm的父类AuthorizingRealm将获取Subject相关信息分两步：

-1）获取身份验证信息方法：doGetAuthenticationInfo(token)

-2）获取授权信息方法：doGetAuthorizationInfo(principals)

doGetAuthenticationInfo方法描述：

-1）根据传入的用户名获取User信息

-2）user为空，抛出账号异常UnknownAccountException

-3）账号已被锁，抛出锁定异常LockedAccountException

-3）生成AuthenticationInfo信息，交给间接父类AuthenticatingRealm使用CredentialsMatcher进行判断密码是否匹配，若不匹配抛出密码错误异常IncorrectCredentialsException

-4）另外如果密码重试此处太多将抛出超出重试次数异常ExcessiveAttemptsException

附：在组装SimpleAuthenticationInfo信息时，需要传入：身份信息（用户名）、凭据（密文密码）、盐（username+salt），CredentialsMatcher使用盐加密传入的明文密码和此处的密文密码进行匹配。

@Override

public AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken authcToken) throws AuthenticationException {

UsernamePasswordToken token = (UsernamePasswordToken) authcToken;

User user = userService.getByUsername(token.getUsername());

if (user == null) {

throw new UnknownAccountException();

}

if (!user.isEnabled()) {

throw new DisabledAccountException();

}

// 盐值

byte[] salt = Encodes.decodeHex(user.getSalt());

return new SimpleAuthenticationInfo(user, user.getPassword(), ByteSource.Util.bytes(salt), getName());

}

doGetAuthorizationInfo方法描述：

String username = (String)token.getPrincipal();

//获取菜单、角色及对应权限

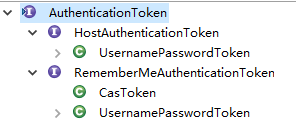
### AuthenticationToken介绍

public interface AuthenticationToken extends Serializable {

Object getPrincipal();

Object getCredentials();

}

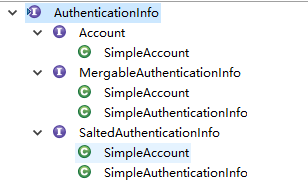


### AuthenticationInfo介绍

AuthenticationInfo有两个作用：

-1）若Realm是AuthenticatingRealm子类，则提供给AuthenticatingRealm内部使用CredentialsMatcher进行凭据验证，若不是则需要在自己的Realm中实现验证。

-2）提供给SecurityManager来创建Subject。



MergableAuthenticationInfo用于提供在多Realm时合并AuthenticationInfo的功能，主要合并Principal、如果是其他的如credentialsSalt，会用后边的信息覆盖前边的。

## 登录方式

### 身份认证

登录Servlet描述：

-1）doGet请求时展现登录页面

-2）doPost请求时进行登录。若登录成功则返回成功页面（附：应该返回访问登录页面之前的那个页面，或没有上一个页面时返回主页）。

当前实现存在的问题：

永远返回到一个成功页面（比如首页），在实际项目中会不一样。比如支付时没有登录将跳转到登录页面，登录成功后在跳转到支付页面。

解决：将当前请求链接保存，在登录成功后后再重定向会该请求上即可。

protected void doGet(HttpServletRequest request, HttpServletResponse response) {

request.getRequestDispatcher("/WEB-INF/jsp/login.jsp").forward(request, response);

}

protected String doPost(HttpServletRequest request, HttpServletResponse response) {

String username = request.getParameter("username");

String password = request.getParameter("password");

Subject subject = SecurityUtils.getSubject();

UsernamePasswordToken token = new UsernamePasswordToken(username, password);

try {

subject.login(token);

} catch (UnknownAccountException e) {

error = "用户名/密码错误";

} catch (IncorrectCredentialsException e) {

error = "用户名/密码错误";

} catch (AuthenticationException e) {

//其他错误，比如锁定，如果想单独处理请单独catch处理

error = "其他错误：" + e.getMessage();

}

if(error != null) {

request.setAttribute("error", error);

request.getRequestDispatcher("/WEB-INF/jsp/login.jsp").forward(req, resp);

}

request.getRequestDispatcher("/WEB-INF/jsp/loginSuccess.jsp").forward(request, response);

}

### 身份认证（拦截器实现）！！！

实现：自定义Filter类，继承FormAuthenticationFilter类！

登录Servlet：

在登录Servlet中通过shiroLoginFailure得到authc登录失败时的异常类名，然后根据异常类名来决定显示什么错误信息。

@WebServlet(name = "formFilterLoginServlet", urlPatterns = "/formfilterlogin")

public class FormFilterLoginServlet extends HttpServlet {

@Override

protected void doGet(HttpServletRequest req, HttpServletResponse resp)

throws ServletException, IOException {

doPost(req, resp);

}

@Override

protected void doPost(HttpServletRequest req, HttpServletResponse resp)

throws ServletException, IOException {

String errorClassName = (String)req.getAttribute("shiroLoginFailure");

if(UnknownAccountException.class.getName().equals(errorClassName)) {

req.setAttribute("error", "用户名/密码错误");

} else if(IncorrectCredentialsException.class.getName().equals(errorClassName)) {

req.setAttribute("error", "用户名/密码错误");

} else if(errorClassName != null) {

req.setAttribute("error", "未知错误：" + errorClassName);

}

req.getRequestDispatcher("/WEB-INF/jsp/formfilterlogin.jsp").forward(req, resp);

}

}

# Shiro登录

专有名词：

身份验证(Authentication)

授权(Authorization)

Subject

SecurityManager

认证器Authenticator

Realm

## Shiro登录源码流程

-1）Controller层

SecurityUtils.getSubject().login(token);

或继承了FormAuthenticationFilter类，其onAccessDenied方法执行executeLogin(request, response)时会执行上面的login(token)

-2）DelegatingSubject类（Subject）：

login(AuthenticationToken token) {

Subject subject = securityManager.login(this, token);

}

-3）DefaultSecurityManager类（安全管理器）：

login(Subject subject, AuthenticationToken token) {

AuthenticationInfo info = authenticate(token);

Subject loggedIn = createSubject(token, info, subject);

onSuccessfulLogin(token, info, loggedIn);

}

-4）AuthenticatingSecurityManager类（DefaultSecurityManager的父类）：

authenticate(AuthenticationToken token) {

return this.authenticator.authenticate(token); //该方法在AbstractAuthenticator类中，实际调用自定义realm的doGetAuthenticationInfo(token)方法

}

-5）AbstractAuthenticator类（认证器）：

authenticate(AuthenticationToken token) {

AuthenticationInfo info = doAuthenticate(token);

}

-6）ModularRealmAuthenticator类（AbstractAuthenticator的子类）：

doAuthenticate(AuthenticationToken authenticationToken) {

return doSingleRealmAuthentication(realms.iterator().next(), authenticationToken);

}

doSingleRealmAuthentication(Realm realm, AuthenticationToken token) {

AuthenticationInfo info = realm.getAuthenticationInfo(token); //realm即自定义Realm

return info;

}

-7）AuthenticatingRealm（Realm，自定义Realm类的父类）：

getAuthenticationInfo(AuthenticationToken token) {

AuthenticationInfo info = getCachedAuthenticationInfo(token);

if (info == null) {

info = doGetAuthenticationInfo(token); //自定义Realm实现的方法

cacheAuthenticationInfoIfPossible(token, info); //存入cache

}

assertCredentialsMatch(token, info); //密码匹配

}

assertCredentialsMatch(AuthenticationToken token, AuthenticationInfo info) {

CredentialsMatcher cm = getCredentialsMatcher();

if (!cm.doCredentialsMatch(token, info)) { //密码匹配，匹配密码前可以进行账号登录次数限制功能，doCredentialsMatch需要自己实现

throw new IncorrectCredentialsException(msg);

}

}

-8）自定义密码匹配器，继承HashedCredentialsMatcher类

public boolean doCredentialsMatch(AuthenticationToken token, AuthenticationInfo info) {

String username = ((UsernamePasswordToken) token).getUsername();

AtomicInteger retryCount = passwordRetryCache.get(username);

if (retryCount == null) {

retryCount = new AtomicInteger(0);

passwordRetryCache.put(username, retryCount);

}

if (retryCount.incrementAndGet() > 3) {

logger.warn("username: " + username + " tried to login more than 3 times in period");

throw new ExcessiveAttemptsException(

"username: " + username + " tried to login more than 3 times in period");

}

boolean matches = super.doCredentialsMatch(token, info);

if (matches) {

passwordRetryCache.remove(username);

}

return matches;

}

## Shiro密码加密

自定义shiro密码加密，需继承HashedCredentialsMatcher类，并重写doCredentialsMatch即可。

shiro加密算法如下：

ByteSource saltByteSource = ByteSource.Util.bytes(toBytes(salt));

ByteSource sourceByteSource = ByteSource.Util.bytes(toBytes(source));

protected byte[] hash(byte[] bytes, byte[] salt, int hashIterations) throws UnknownAlgorithmException {

MessageDigest digest = getDigest(getAlgorithmName()); //常用算法MD2、MD5、SHA-1、SHA-256、SHA-384SHA-512

if (salt != null) {

digest.reset();

digest.update(salt);

}

byte[] hashed = digest.digest(bytes);

int iterations = hashIterations - DEFAULT\_ITERATIONS; //already hashed once above

//iterate remaining number:

for (int i = 0; i < iterations; i++) {

digest.reset();

hashed = digest.digest(hashed);

}

return hashed;

}

# 缓存

相关链接：

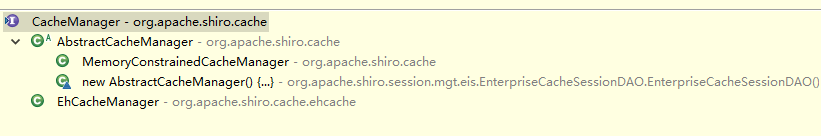
https://www.cnblogs.com/nuccch/p/8044226.html

https://www.jianshu.com/p/cc92ed4138ce --\*\*\*\*

## 概述

Shiro提供了类似Spring的cache抽象，即shiro本身不实现cache，Shiro官方说法是实现一套完整缓存机制会使得shiro偏离核心功能（认证和授权）。

Shiro提供一个CacheManager接口，用于提供具体缓存实现（如ehcache、memcache、redis）。并提供了ehcache缓存的实现，引入了shiro-ehcache.jar包。



Shiro支持在2个地方定义缓存管理器：

-1）SecurityManager（CachingSecurityManager声明CacheManager类型变量）

-2）Realm（CachingRealm声明CacheManager类型变量）。

通常我们都会自定义Realm实现，将权限数据存放到缓存，因此通过在Realm中定义缓存管理器。

Shiro内部相应的组件（DefaultSecurityManager）会自动检测相应的对象（如Realm）是否实现CacheManagerAware并自动注入相应的CacheManager。

详情：源码RealmSecurityManage的applyCacheManagerToRealms方法中。

其实SecurityManager中设置的CacheManager组中都会给Realm使用，即：真正使用CacheManager的组件是Realm。

## Shiro使用EhCache缓存

相关链接：

<https://blog.csdn.net/tanleijin/article/details/81118963>

### xml配置

-1）配置ehcache-shiro.xml

<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="http://ehcache.org/ehcache.xsd"

updateCheck="false" name="shiroCache">

<!-- 用户权限/菜单缓存，10分钟过期 -->

<defaultCache maxElementsInMemory="10000" eternal="false" timeToIdleSeconds="600" timeToLiveSeconds="600"

overflowToDisk="false" diskPersistent="false" diskExpiryThreadIntervalSeconds="600" />

<!-- 登录记录缓存 锁定10分钟 -->

<cache name="passwordRetryCache" maxEntriesLocalHeap="20000" eternal="false" timeToIdleSeconds="600"

timeToLiveSeconds="0" overflowToDisk="false" statistics="false">

</cache>

</ehcache>

-2）配置缓存

<bean id="shiroEhcacheManager" class="org.apache.shiro.cache.ehcache.EhCacheManager">

<property name="cacheManagerConfigFile" value="classpath:ehcache-shiro.xml" />

</bean>

-3）配置安全管理器

<bean id="securityManager" class="org.apache.shiro.web.mgt.DefaultWebSecurityManager">

<property name="realm" ref="shiroDbRealm" />

<property name="cacheManager" ref="shiroEhcacheManager" />

</bean>

2、Bean模式

-1）创建安全管理器的Bean

@Bean(name = "securityManager")

public DefaultWebSecurityManager securityManager() {

DefaultWebSecurityManager securityManager = new DefaultWebSecurityManager();

securityManager.setRealm(shiroRealm());

// securityManager.setCacheManager(ehCacheManager()); //在realm中注入

return securityManager;

}

-2）创建EhCache的Bean

@Bean (name = "ehCacheManager")

public EhCacheManager ehCacheManager() {

EhCacheManager em = new EhCacheManager();

// 需把EhCache注入CacheManager！！

em.setCacheManager(ehCacheManager());

em.setCacheManagerConfigFile("classpath:ehcache.xml");

return em;

}

-3）创建Realm的Bean

@Bean(name = "shiroRealm")

public ShiroRealm shiroRealm() {

ShiroRealm realm = new ShiroRealm();

// 注入cacheManager依赖

realm.setCacheManager(ehCacheManager());

// 启用缓存，默认关闭

realm.setCachingEnabled(true);

// 启用身份验证缓存，即缓存AuthenticationInfo，默认false

realm.setAuthenticationCachingEnabled(true);

// 启用授权缓存，即缓存AuthorizationInfo的信息，默认为false

realm.setAuthorizationCachingEnabled(true);

return realm;

}

## 缓存分类

缓存分2类：

-1）集中式缓存（如redis、memcached等存储中间件）

-2）本地缓存

使用集中式缓存,页面的每次访问都会向缓存发起一次网络请求。如果大量使用shiro的jsp标签，则会一个页面访问将出现N个到集中缓存的网络请求。因此如果大量使用shiro的jsp标签库，那么采用本地缓存才是最佳选择。

### 集中式缓存

使用场景：没有使用shiro的jsp标签。比如前后端完全分离的项目。

集中式缓存组件有：EhCache、redis、Mencached等。

案例：

基于Ehcache集群模式的存放方案：http://www.ehcache.org/

基于Redis的集中式缓存方案：https://github.com/alexxiyang/shiro-redis

基于Memcached的集中式缓存方案：https://github.com/mythfish/shiro-memcached

### 本地缓存

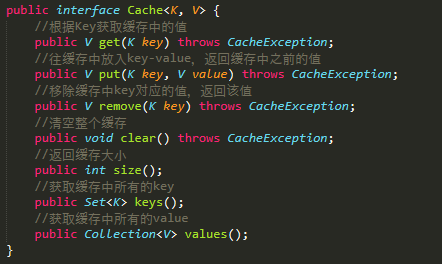
本地缓存的实现方式：

-1）直接存放到JVM堆内存

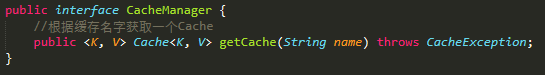
-2）使用NIO存在在堆外内存，自定义实现或借助于第三方缓存组件。

## Shiro提供的缓存接口

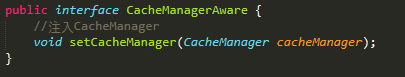
（1）shiro提供的Cache接口



（2）shiro提供的CacheManager接口



（3）shiro提供了CacheManagerAware用于注入CacheManager



# 限制登录次数

相关链接：https://blog.csdn.net/zsg88/article/details/73929237

## 概述

功能描述：自定义验证器，继承HashedCredentialsMatcher，并加入缓存。每次验证用户密码之前验证用户名尝试数，若超过5次抛出异常，否则验证用户名密码，验证成功把尝试次数清零，不成功则直接退出。

Shiro将用户名密码的验证交给CredentialsMatcher类。自定义Realm继承AuthorizingRealm，在其父类AuthenticatingRealm的的getAuthenticationInfo方法中会调用CredentialsMatcher的 doCredentialsMatch 来验证用户输入用户名密码是否匹配。

（1）编写验证器，增加登录次数校验功能

public class RetryLimitCredentialsMatcher extends HashedCredentialsMatcher {

private static final Logger log = LoggerFactory.getLogger(RetryLimitCredentialsMatcher.class);

//集群中可能会导致出现验证多过5次的现象

private Cache<String, AtomicInteger> lgoinRetryCache;

private String lgoinRetryCacheName;

public RetryLimitCredentialsMatcher(CacheManager cacheManager, String lgoinRetryCacheName) {

this.lgoinRetryCacheName = lgoinRetryCacheName;

lgoinRetryCache = cacheManager.getCache(lgoinRetryCacheName);

}

@Override

public boolean doCredentialsMatch(AuthenticationToken token, AuthenticationInfo info) {

String username = (String) token.getPrincipal();

//retry count + 1

AtomicInteger retryCount = lgoinRetryCache.get(username);

if (null == retryCount) {

retryCount = new AtomicInteger(0);

lgoinRetryCache.put(username, retryCount);

}

if (retryCount.incrementAndGet() > 5) {

log.warn("username: " + username + " tried to login more than 5 times in period");

throw new ExcessiveAttemptsException("username: " + username + " tried to login more than 5 times in period");

}

boolean matches = super.doCredentialsMatch(token, info);

if (matches) {

//clear retry data

lgoinRetryCache.remove(username);

}

return matches;

}

}

（2）spring shiro配置

<!-- Realm实现 -->

<bean id="userRealm" class="org.jstudioframework.freeway.shiro.UserRealm">

<!-- 将凭证匹配器设置到realm中，realm按照凭证匹配器的要求进行散列 -->

<property name="credentialsMatcher" ref="credentialsMatcher"/>

</bean>

<!-- 凭证匹配器 -->

<bean id="credentialsMatcher"

class="org.jstudioframework.freeway.shiro.authc.credential.RetryLimitCredentialsMatcher">

<constructor-arg index="0" ref="cacheManager"/>

<constructor-arg index="1" value="5"/>

<property name="hashAlgorithmName" value="md5"/>

<property name="hashIterations" value="1"/>

</bean>

<!-- 缓存管理器 -->

<bean id="cacheManager" class="org.apache.shiro.cache.ehcache.EhCacheManager">

<property name="cacheManagerConfigFile" value="classpath:cache/shiro-ehcache.xml"/>

</bean>

# 拦截器机制

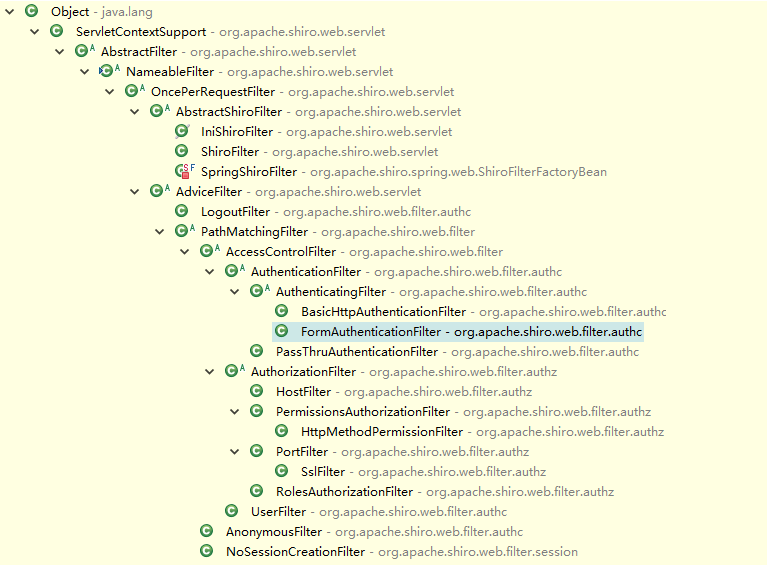
## 拦截器介绍

相关链接：

<https://www.cnblogs.com/yoohot/p/6085830.html>

Shiro自带的filter：NameableFilter-》OncePerRequestFilter-》AdviceFilter-》PathMatchingFilter-》AuthenticationFilter & AuthenticatingFilter -》FromAuthenticationFilter

1.2.5版本filter关系：

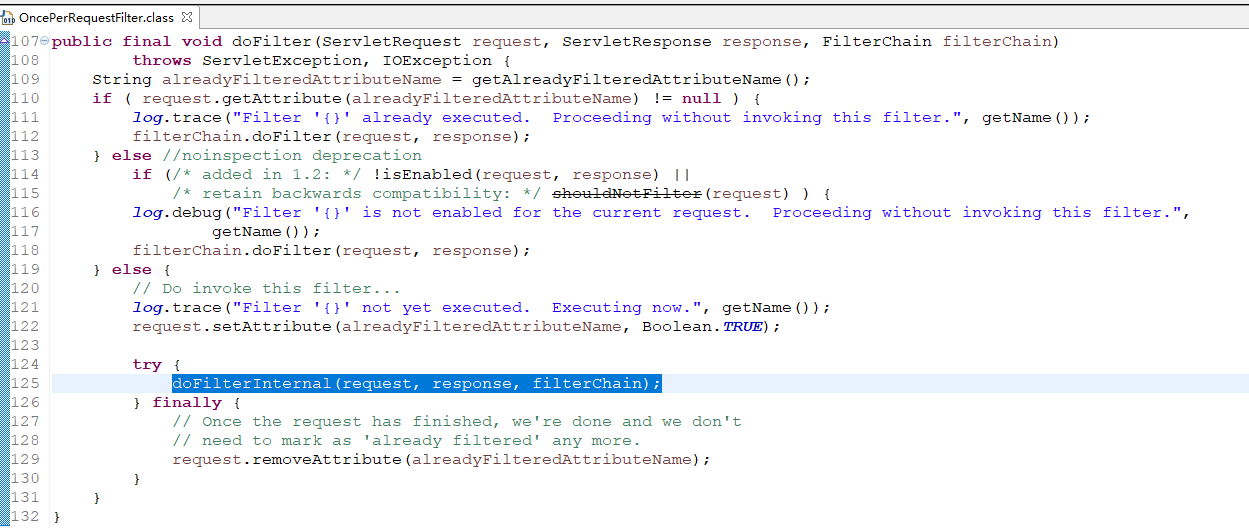


### NameableFilter

仅有一个name属性，定义每一个filter的名字。在FileChainManager中会调用配置文件中的配置属性名字来为每一个filter命名以及为默认的filter命名（如authc）。

### OncePerRequestFilter

保证客户端请求后该filter的doFilter只会执行一次。



可以看出doFilter的实质内容是在doFilterInternal方法中完成的。所以实质上是保证每一个filter的doFilterInternal只会被执行一次，例如[/user/\*\*=authc,authc]则只会执行authc中的doFilterInternal一次。

doFilterInternal为抽象方法，在shiro整个filter体系中的核心方法及实质入口。

另：有一个enabled属性，表示是否启用该filter。true-默认，启用；false-不启用，会跳过该filter的doFilterInternal方法而去执行filter链中其它filter。

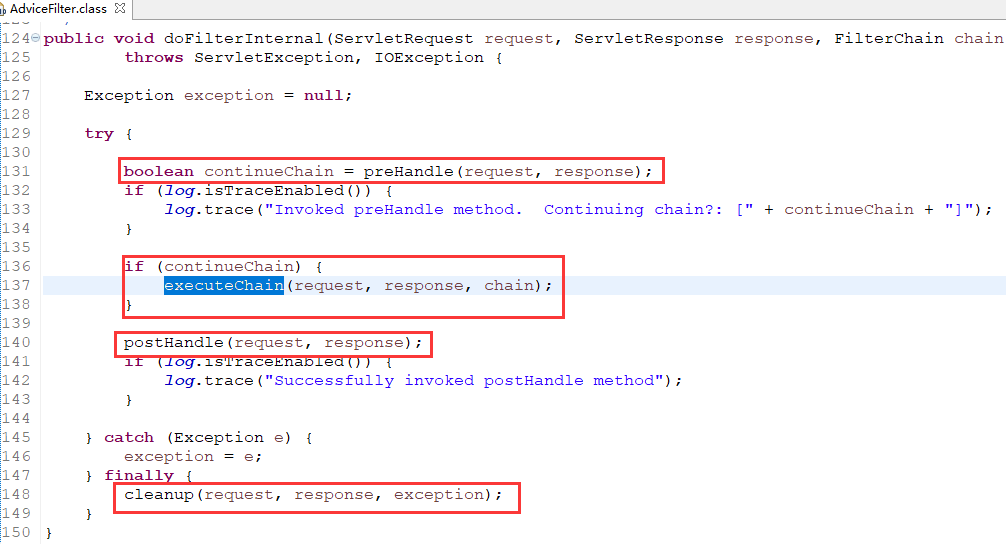
另：shiro是通过在request中设置一个该filter特定的属性值来保证该filter只会执行一次的。

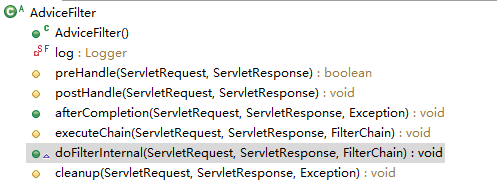
### ShiroFilter

ShiroFilter是整个Shiro的入口点，用于拦截器需要安全控制的请求进行处理。

### AdviceFilter

AdviceFilter总体是对OncePerRequestFilter中的doFilterInternal进一步细化控制。





类似springmvc中的Interceptor。先调用preHandle做一些前置判断，如果返回false则filter链不继续往下执行。postHandler在目标方法（即客户端请求的接口）正常（未抛异常）执行后完成一些操作，默认不做任何操作。在finally的cleanup方法中会调用afterCompletion方法。

类描述如下：

public abstract class AdviceFilter extends OncePerRequestFilter {

// 类似于AOP中的前置增强，在拦截器链执行之前执行，ture-继续执行拦截器链，false-中断后续的拦截器链的执行并直接返回

// 用途：进行预处理（比如基于表单的身份验证、授权）

boolean preHandle(ServletRequest request, ServletResponse response) throws Exception

// 类似于AOP中的后置返回增强，在拦截器链执行完成后执行

// 用途：进行后处理（如记录执行时间之类的）

void postHandle(ServletRequest request, ServletResponse response) throws Exception

// 类似于AOP中的后置最终增强，即不管有没有异常都会执行

// 用途：进行清理资源（如接触Subject与线程的绑定之类的）

void afterCompletion(ServletRequest request, ServletResponse response, Exception exception) throws Exception

void executeChain(ServletRequest request, ServletResponse response, FilterChain chain) throws Exception

}

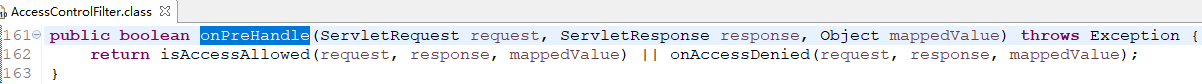
### PathMatchingFilter

抽象类，对AdviceFilter中的preHandle方法做进一步细化。

preHandle中，若请求的路径非该filter配置中配置的拦截路径，则直接返回true进行下一个filter。若包含在此filter路径中，则会在isFilterChainContinued方法中进一步控制，该方法中会调用onPreHandle方法，所以子类可以在onPreHandle中编写filter控制流程代码。

### AccessControlFilter

对PathMatchingFilter的onPreHandle方法做了进一步细化。



isAccessAllowed方法和onAccessDenied方法达到控制效果。这两个方法都是抽象方法，由子类去实现。

说明：isAccessAllowed和onAccessDenied方法影响到onPreHandle方法；onPreHandle方法影响到preHandle方法；preHandle方法会达到控制filter链是否执行下去的效果。

所以如果正在执行的filter中isAccessAllowed和onAccessDenied都返回false，则整个filter控制链都将结束，不会到达目标方法（客户端请求的接口），而是直接跳转到某个页面（由filter定义的，将会在authc中看到）。

isAccessAllowed方法描述：

\* 表示是否允许访问；如果isAccessAllowed返回true则不会执行onAccessDenied方法。

\* @param object 表示写在拦截器中括号里面的字符串mappedValue就是[urls]配置中拦截器参数部分

public boolean isAccessAllowed(ServletRequest servletRequest, ServletResponse servletResponse, Object object) throws Exception

附：该方法类似白名单被放行的。若有些不需要登录可直接访问的接口，可以在这里配置。

onAccessDenied方法描述：

\* 表示当访问拒绝时是否已经处理了；返回true表示需要继续处理，false表示该拦截器实例已经处理，将直接返回即可。

\* onAccessDenied的执行取决于isAccessAllowed的值，若返回true则onAccessDenied不会执行，若返回false则执行onAccessDenied

\* 若onAccessDenied也返回false，则直接返回，不会进入请求的方法（只有isAccessAllowed和onAccessDenied的情况下）

public boolean onAccessDenied(ServletRequest request, ServletResponse response) throws Exception

附：onAccessDenied是我主要逻辑功能处理的模块。在这里生成token对象后使用subject进行登录，以便后面方法中权限/角色的验证。

想进行访问访问的控制就可以继承AccessControlFilter；如果我们要添加一些通用数据我们可以直接继承PathMatchingFilter。

### AuthenticationFilter和AuthenticatingFilter

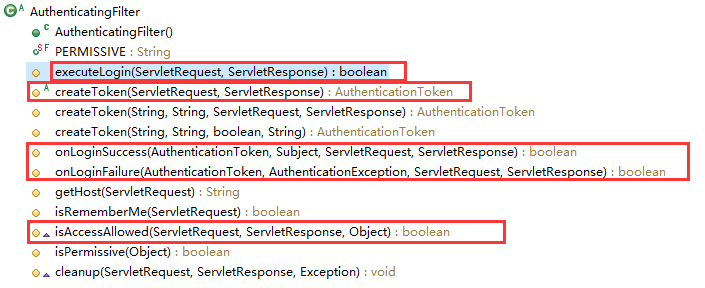
AuthenticationFilter描述：

对当前用户进行身份验证的基类。

该类封装了检查用户是否已经登录的逻辑，而子类需要对未经身份验证的请求实现特定逻辑。

AuthenticatingFilter描述：

AuthenticationFilter能根据传入的请求自动执行身份验证尝试。



executeLogin方法：实现用户登录（FormAuthenticationFiltershiro细化）。

isAccessAllowed方法：用来判断用户是否已经登录。若未登录再判断是否请求的是登录地址，是登录地址则放行，否则返回false终止filter链。

onLoginSuccess和onLoginFailure方法：登录成功或失败时做一些操作。

### FormAuthenticationFiltershiro

提供登录实现的Filter。

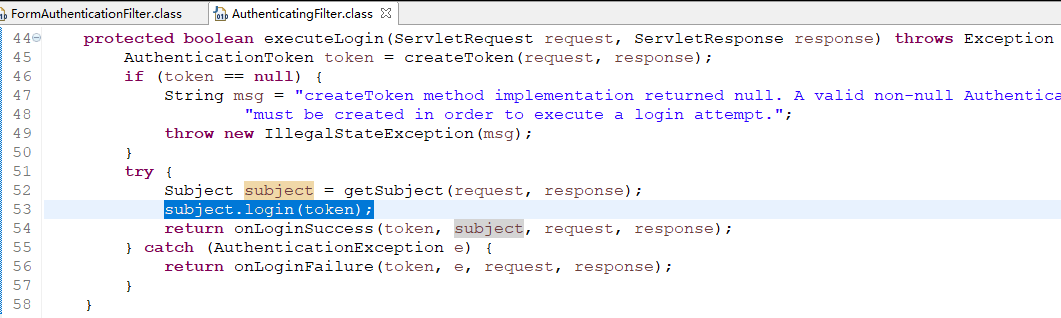
FormAuthenticationFiltershiro的onAccessDenied方法描述：

若用户未登录（AuthenticatingFilter中的isAccessAllowed判断了用户未登录），则会调用onAccessDenied方法做用户登录操作。



saveRequestAndRedirectToLogin保存当前request，并重定向到登录页面，登陆成功后可从WebUtils.getSavedRequest(req);中取出。

AuthenticatingFilter中的executeLogin：



若登录成功返回false，表示终止filter链，直接重定向到成功页面。

若登录失败直接返回true，表示继续执行filter链并最终跳转到登录页面，该方法还会设置一些登录失败提示信息shiroLoginFailure。

上面的onLoginSuccess和onLoginFailure方法属于的FormAuthenticationFiltershiro类。

附： FormAuthenticationFiltershiro类中的onLoginSuccess和onLoginFailure方法的返回值与AuthenticatingFilter类中的方法返回值不一同！

## 拦截器链

Shiro对Servlet容器的FilterChain进行了代理，即Shiro通过ProxiedFilterChain对Servlet容器的FilterChain进行了代理。

Shiro的ProxiedFilterChain执行流程：1）先执行Shiro自己的Filter链；2）再执行Servlet容器的Filter链（原始Filter）。

## 自定义拦截器

分类：

-1）继承OncePerRequestFilter

只需实现doFilterInternal方法即可，在这里面实现filter的功能。

切记在该方法中最后调用filterChain.doFilter(request, response)，允许filter链继续执行下去。

可以在这个自定义filter中覆盖isEnable达到控制该filter是否需要被执行（实质是doFilterInternal方法）以达到动态控制的效果，一般不建议直接继承这个类;

-2）继承AdviceFilter

提供了三个方法preHandle、postHandle、afterCompletion

若需要在目标方法执行前后都做一些判断的话应该继承这个类覆盖preHandle 和postHandle

-3）继承PathMatchingFilter

preHandle实质会判断onPreHandle来决定是否继续往下执行。所以只需覆盖onPreHandle方法即可。

-4）继承AccessControlFilter

最常用的，该filter中onPreHandle调用isAccessAllowed和onAccessDenied决定是否继续执行。

一般继承该filter，isAccessAllowed决定是否继续执行。onAccessDenied做后续的操作，如重定向到另外一个地址、添加一些信息到request域等等。

-5）自定义登录filter

若要自定义登录filter，一般是由于前端传过来的需求所定义的token与shiro默认提供token的不符，可以继承AuthenticatingFilter ，在这里面实现createToken来创建自定义token。

另外需要自定义凭证匹配器credentialsMatcher。重写public boolean doCredentialsMatch(AuthenticationToken token, AuthenticationInfo info)即可。realm也需要自定义以返回自定义的token。

Shiro登录：

Shiro提供的FormAuthenticationFilter认证过滤器，继承了FormAuthenticationFilter，若已登录则isAccessAllowed直接通过，否则在 onAccessDenied中判断是否是登录请求，若是请求登录页面，直接通过，若是post提交登录信息则会进行登录操作。否则直接跳转到登录页面。登录是由shiro的securityManager完成的，securityManager从Realm获取用户的真实身份，从FormAuthenticationFilter的createToken获取用户提交的token，credentialsMatcher完成是否匹配成功操作。

### 基于表单登录拦截器

onPreHandle主要流程：

-1）首先判断是否已登录过了，如果已经登录了则继续拦截器链即可；

-2）如果没有登录，看看是否是登录请求，如果是get方法的登录页面请求，则继续拦截器链（到请求页面），否则如果是get方法的其他页面请求则保存当前请求并重定向到登录页面；

-3）如果是post方法的登录页面表单提交请求，则收集用户名/密码登录即可，如果失败了保存错误消息到“shiroLoginFailure”并返回到登录页面；

-4）如果登录成功了，且之前有保存的请求，则重定向到之前的这个请求，否则到默认的成功页面。

代码如下：

public class FormLoginFilter extends PathMatchingFilter {

private String loginUrl = "/login.jsp";

private String successUrl = "/";

@Override

protected boolean onPreHandle(ServletRequest request, ServletResponse response, Object mappedValue) throws Exception {

if(SecurityUtils.getSubject().isAuthenticated()) {

return true;//已经登录过

}

HttpServletRequest req = (HttpServletRequest) request;

HttpServletResponse resp = (HttpServletResponse) response;

if(isLoginRequest(req)) {

if("post".equalsIgnoreCase(req.getMethod())) {//form表单提交

boolean loginSuccess = login(req); //登录

if(loginSuccess) {

req.getRequestDispatcher(successUrl).forward(req, resp);

return false;

}

}

return true;//继续过滤器链

} else {//保存当前地址并重定向到登录界面

req.getRequestDispatcher(loginUrl).forward(req, resp);

return false;

}

}

private boolean login(HttpServletRequest req) {

String username = req.getParameter("username");

String password = req.getParameter("password");

try {

SecurityUtils.getSubject().login(new UsernamePasswordToken(username, password));

} catch (Exception e) {

req.setAttribute("shiroLoginFailure", e.getClass());

return false;

}

return true;

}

private boolean isLoginRequest(HttpServletRequest req) {

return pathsMatch(loginUrl, WebUtils.getPathWithinApplication(req));

}

}

配置文件：

[filters]

formLogin=com.github.zhangkaitao.shiro.chapter8.web.filter.FormLoginFilter

### 任意角色授权拦截器(未完待续…)

Shiro提供roles拦截器，其验证用户拥有所有角色，没有提供验证用户拥有任意角色的拦截器。

流程：

-1）首先判断用户有没有任意角色，如果没有返回false，将到onAccessDenied进行处理；

-2）如果用户没有角色，接着判断用户有没有登录，如果没有登录先重定向到登录；

-3）如果用户没有角色且设置了未授权页面（unauthorizedUrl），那么重定向到未授权页面；否则直接返回401未授权错误码。

代码如下：

<https://jinnianshilongnian.iteye.com/blog/2025656> --未完待续...

# 工具

## 工具

### 密码生成类

shiro密码生成类：

public class PasswordHelper {

private RandomNumberGenerator randomNumberGenerator = new SecureRandomNumberGenerator();

private String algorithmName = "md5";

private final int hashIterations = 2;

public void encryptPassword(User user) {

user.setSalt(randomNumberGenerator.nextBytes().toHex());

String newPassword = new SimpleHash(

algorithmName,

user.getPassword(),

ByteSource.Util.bytes(user.getCredentialsSalt()),

hashIterations).toHex();

user.setPassword(newPassword);

}

}

### 登录次数限制

通过继承HashedCredentialsMatcher，且使用Ehcache记录重试次数和超时时间。

public boolean doCredentialsMatch(AuthenticationToken token, AuthenticationInfo info) {

String username = (String)token.getPrincipal();

//retry count + 1

AtomicInteger retryCount = passwordRetryCache.get(username);

if (retryCount == null) {

retryCount = new AtomicInteger(0);

passwordRetryCache.put(username, retryCount);

}

if (retryCount.incrementAndGet() > 3) { //retry more 3 times

throw new ExcessiveAttemptsException();

}

boolean matches = super.doCredentialsMatch(token, info);

if(matches) {

//clear retry count

passwordRetryCache.remove(username);

}

return matches;

}

# Shiro-filter执行流程

## 配置DelegatingFilterProxy代理

在xml中配置：

<filter>

<filter-name>shiroFilter</filter-name>

<filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>

<async-supported>true</async-supported>

<init-param>

<param-name>targetFilterLifecycle</param-name>

<param-value>true</param-value>

</init-param>

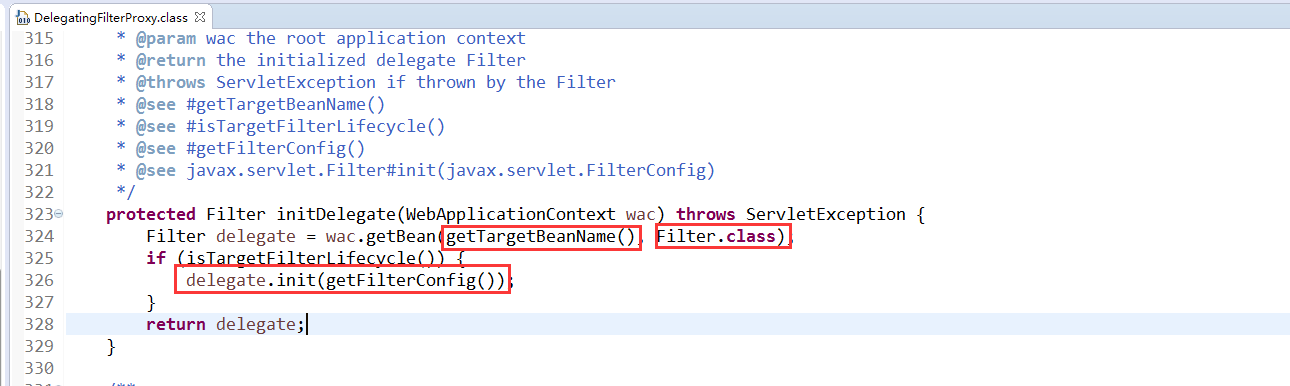
</filter>

在web.xml中配置shiroFilter代理，以后每当request请求时都会被该代理拦截，然后代理中调用真正的被代理filter执行处理。

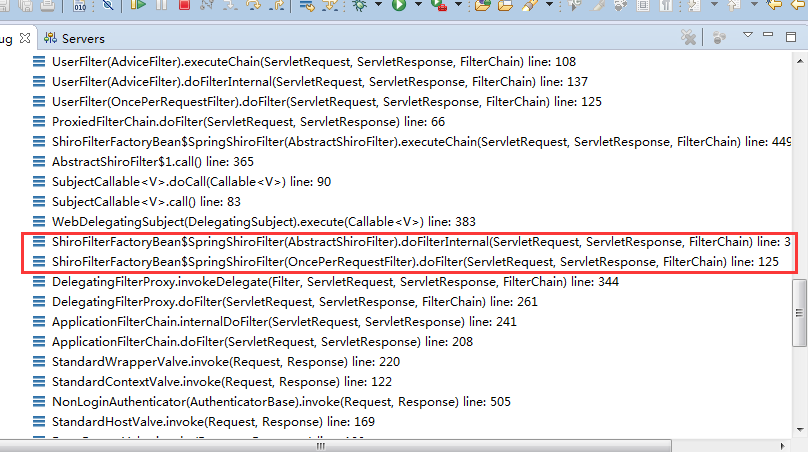
DelegatingFilterProxy中初始化delegate的过程

根据web.xml中<filter-name>值，在spring容器中getBean("shiroFilter", Filter.class)最终得到org.apache.shiro.spring.web.ShiroFilterFactoryBean 的一个单实例，可以看到ShiroFilterFactoryBean没有实现任何Filter接口。

Filter delegate = wac.getBean(getTargetBeanName(), Filter.class); //生成该Filter



被代理的对象org.apache.shiro.spring.web.ShiroFilterFactoryBean



Spring生成的是ShiroFilterFactoryBean代理对象，实际上是个Filter

## 在spring.xml配置

<!-- Shiro的Web过滤器 -->

<bean id="shiroFilter" class="org.apache.shiro.spring.web.ShiroFilterFactoryBean">

<property name="securityManager" ref="securityManager"/>

<property name="loginUrl" value="/login.jsp"/>

<property name="unauthorizedUrl" value="/unauthorized.jsp"/>

<property name="filters">

<util:map>

<entry key="authc" value-ref="formAuthenticationFilter"/>

</util:map>

</property>

<property name="filterChainDefinitions">

<value>

/index.jsp = anon

/unauthorized.jsp = anon

/login.jsp = authc

/logout = logout

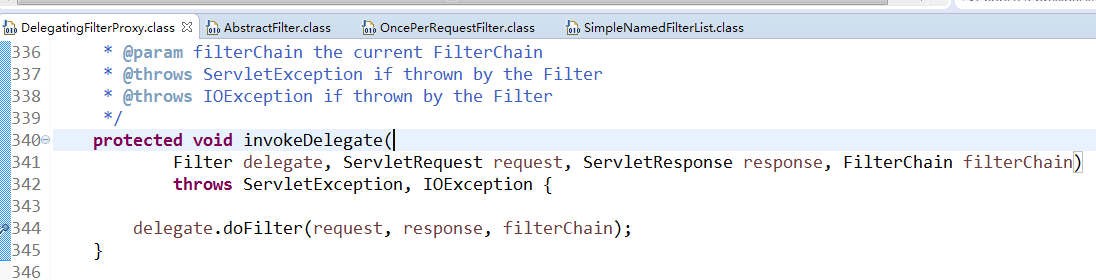
/\*\* = user

</value>

</property>

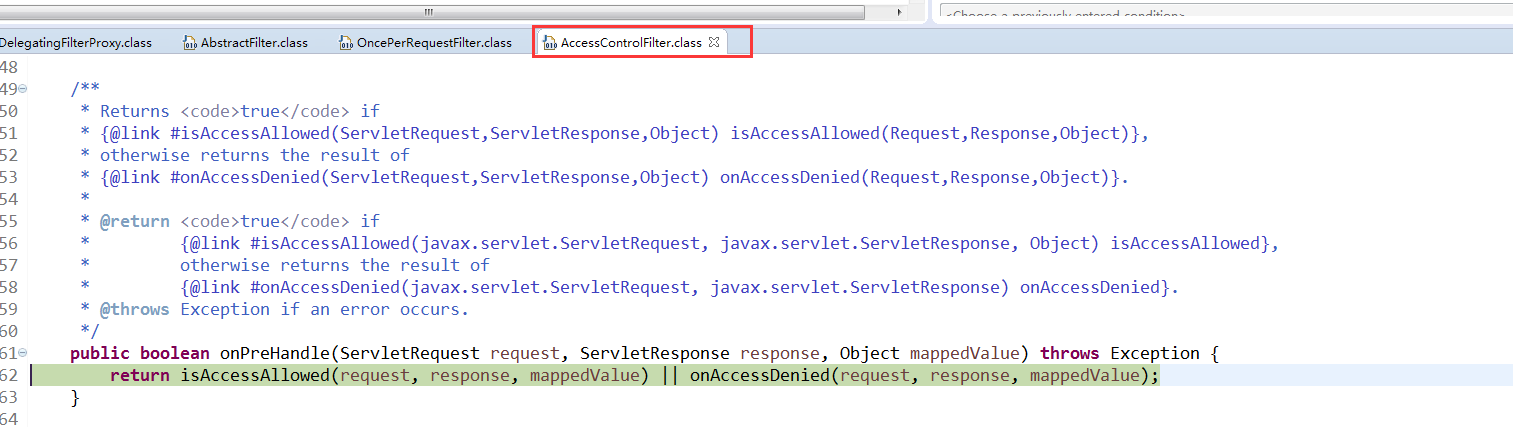
</bean>

后面所有的处理都是DelegatingFilterProxy中交给delegate去执行的，实际上就是执行一个过滤器链。

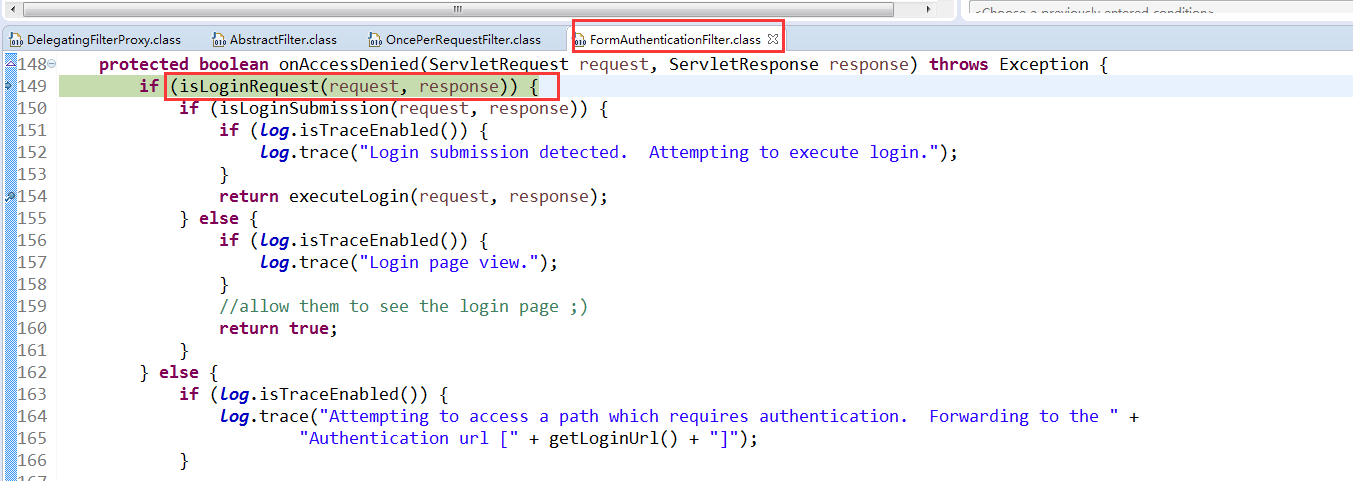


auth

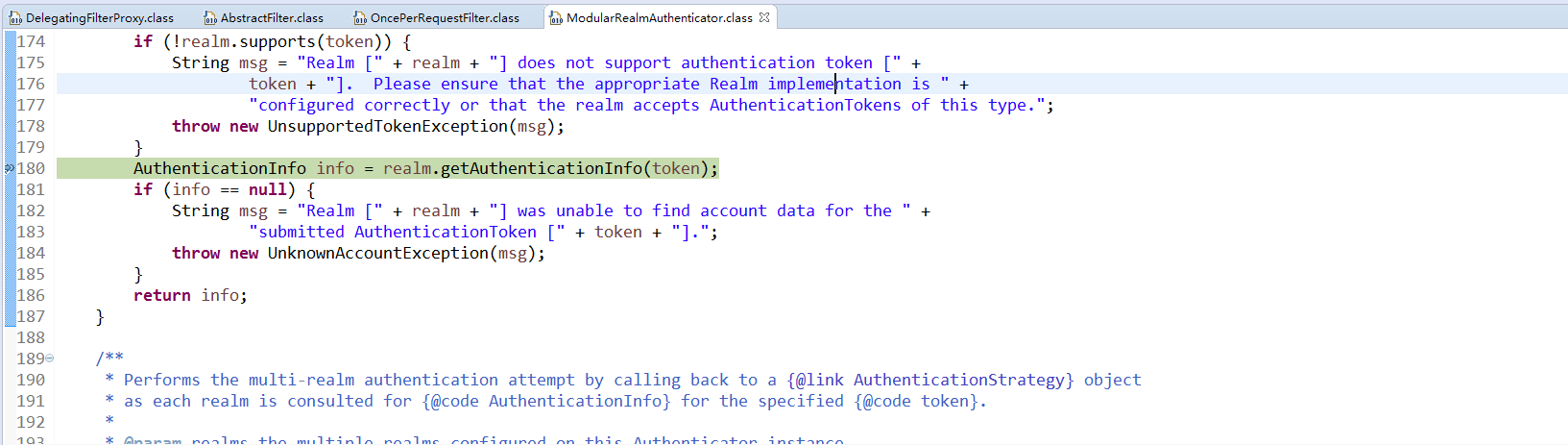




表单登录

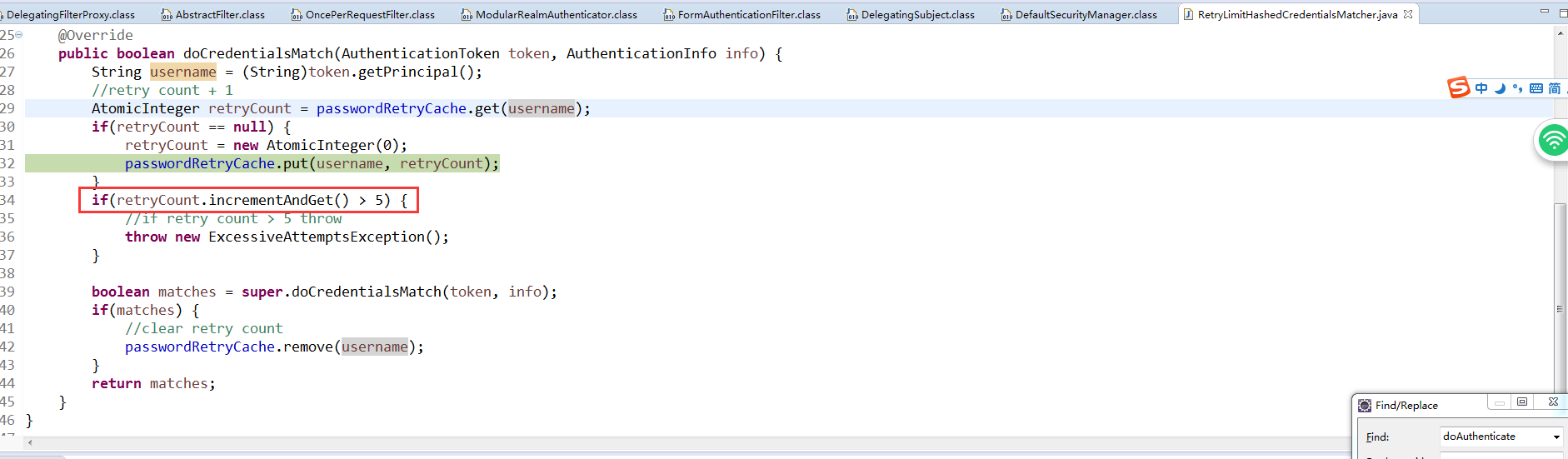


真正调用Realm auth的地方

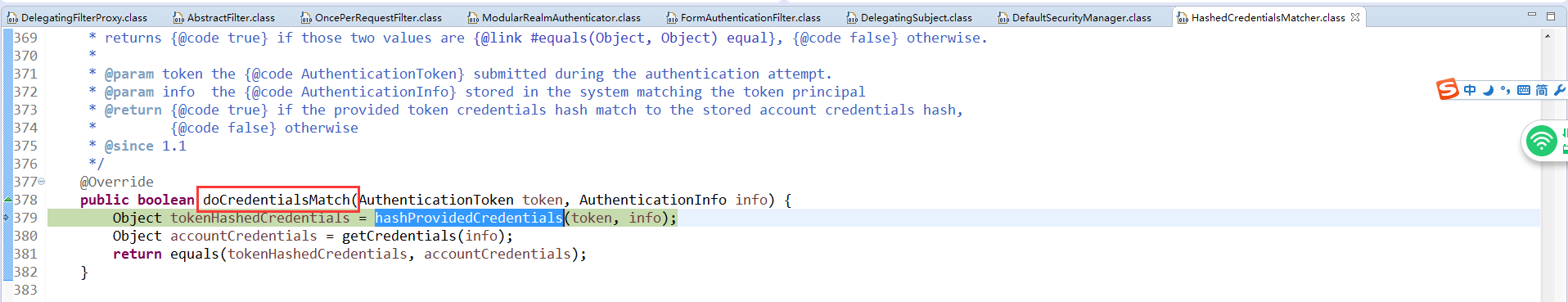




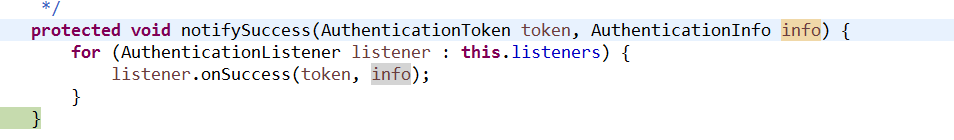
retryCount判断



密码匹配



登录成功



# 代码分析

## 认证和授权流程

认证过程：

构造SecurityManager环境🡪Subject.login()提交认证🡪SecurityManager.login()执行认证🡪Authenticator执行认证🡪Realm根据身份获取验证信息

授权流程：

构造SecurityManager环境🡪Subject.isPermitted()授权🡪SecurityManager.isPermitted()执行授权🡪Authorizer执行授权

### 认证流程

认证执行流程：

-1）通过ini配置文件创建securityManager

-2）调用subject.login()方法主体提交认证，提交的token

-3）securityManager进行认证，securityManager最终由ModularRealmAuthenticator进行认证。

-4）ModularRealmAuthenticator调用IniRealm(给realm传入token) 去ini配置文件中查询用户信息

-5）IniRealm根据输入的token（UsernamePasswordToken）从 shiro.ini查询用户信息，根据账号查询用户信息（账号和密码）

如果查询到用户信息，就给ModularRealmAuthenticator返回用户信息（账号和密码）

如果查询不到，就给ModularRealmAuthenticator返回null

-6）ModularRealmAuthenticator接收IniRealm返回Authentication认证信息

如果返回的认证信息是null，ModularRealmAuthenticator抛出异常（org.apache.shiro.authc.UnknownAccountException）

如果返回的认证信息不是null（说明inirealm找到了用户），对IniRealm返回用户密码 （在ini文件中存在）和token中的密码 进行对比，如果不一致抛出异常（org.apache.shiro.authc.IncorrectCredentialsException）

### 授权流程

授权执行流程：

-1）对subject进行授权，调用方法isPermitted（"permission串"）

-2）SecurityManager执行授权，通过ModularRealmAuthorizer执行授权

-3）ModularRealmAuthorizer执行realm（自定义的Realm）从数据库查询权限数据调用realm的授权方法：doGetAuthorizationInfo

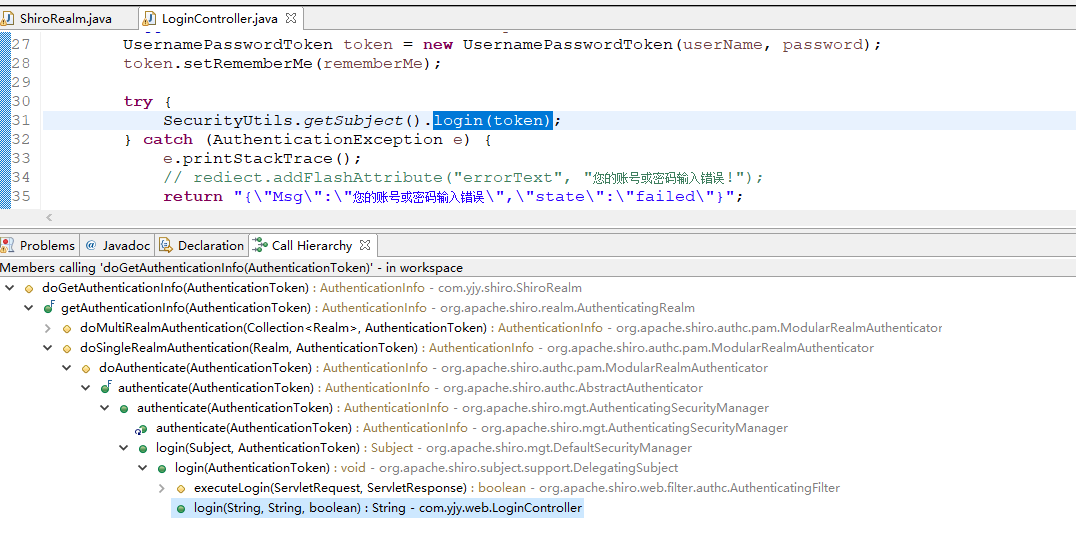
-4）realm从数据库查询权限数据，返回ModularRealmAuthorizer

-5）ModularRealmAuthorizer调用PermissionResolver进行权限串比对

-6）如果比对后，isPermitted中"permission串"在realm查询到权限数据中，说明用户访问permission串有权限，否则没有权限，抛出异常。

## 登录认证源码分析

登录流程各方法调用关系如下：



登录Controller：

try {

UsernamePasswordToken token = token = new UsernamePasswordToken(username, password);

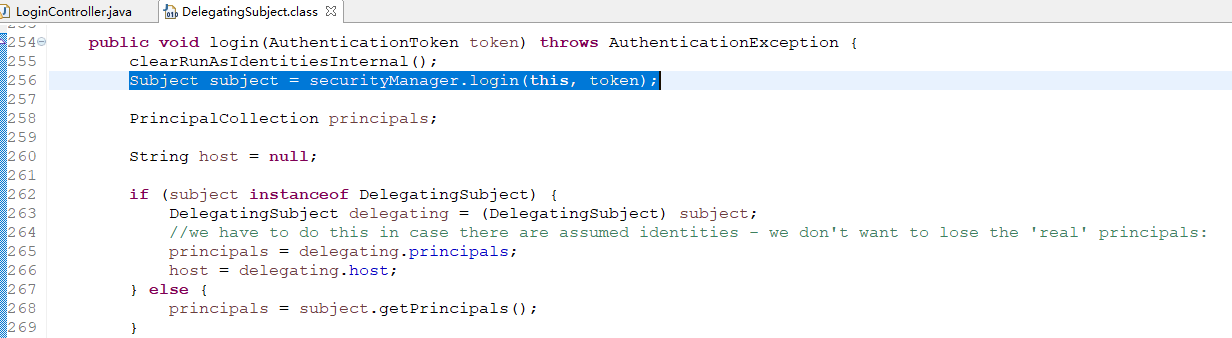
SecurityUtils.getSubject().login(token);

} catch(UnknownAccountException | IncorrectCredentialsException e) {

msg = "账号密码有误";

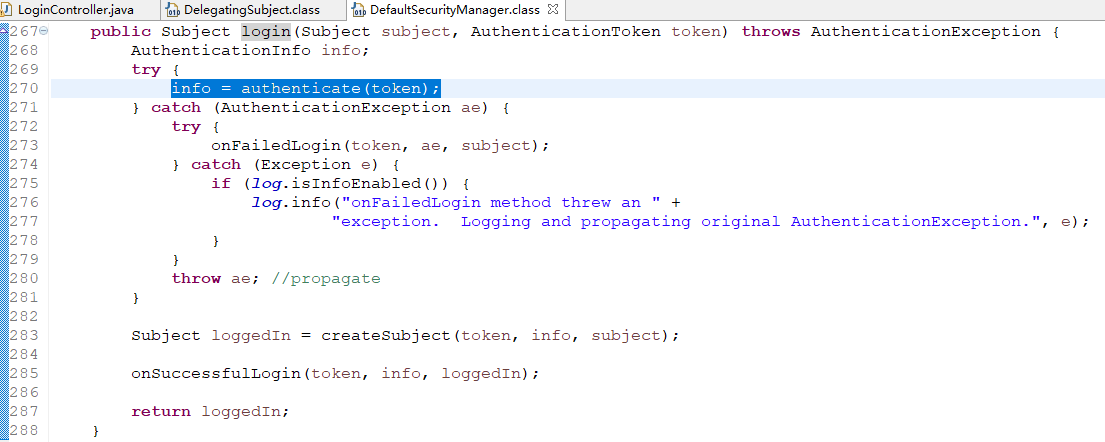
}

查看Subject接口的实现类的login()方法如下：



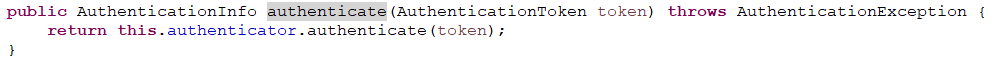
主要使用了securityManager的login()方法。

查看securityManager接口实现类的login()方法如下：



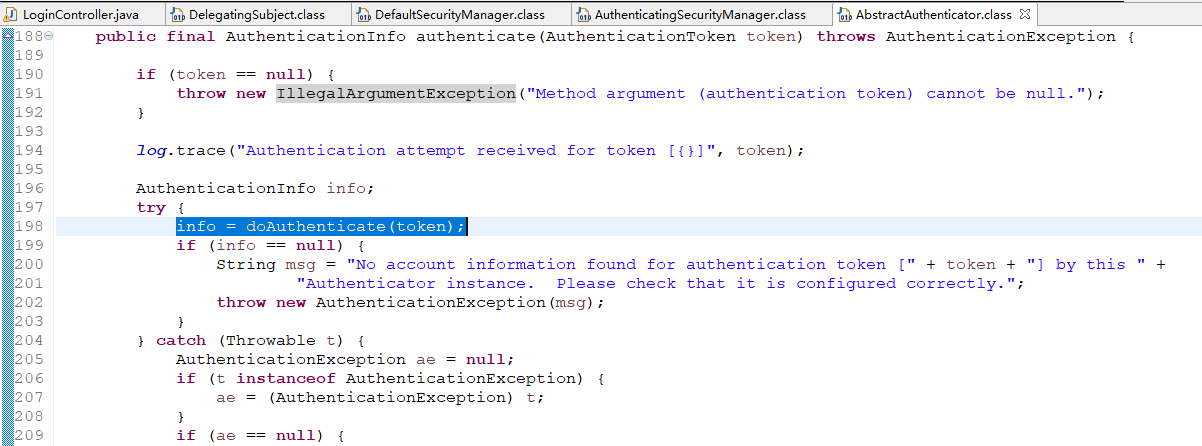
在这个方法中定义了AuthenticationInfo对象来接收从Realm传来的信息。

进入authenticate()方法中：

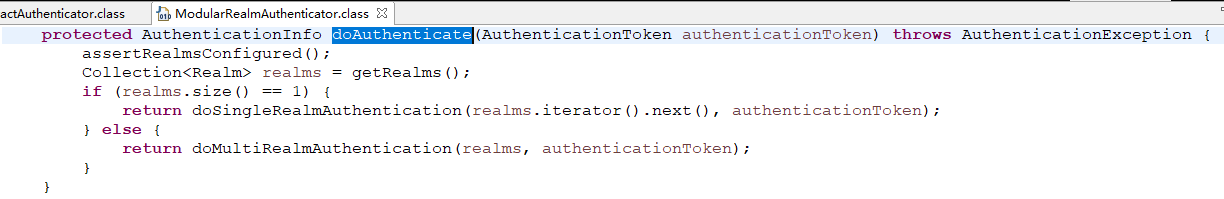


发现调用的是authenticator的authenticate这个方法。

进入this.authenticator.authenticate(token)这个方法中：

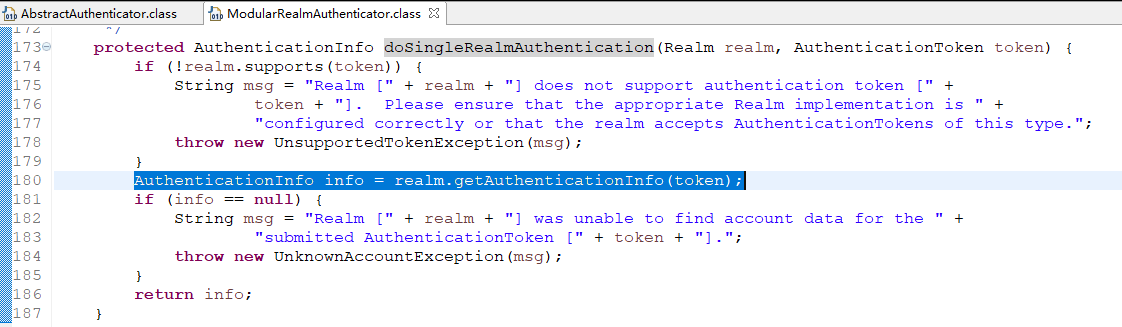


查看它的实现：



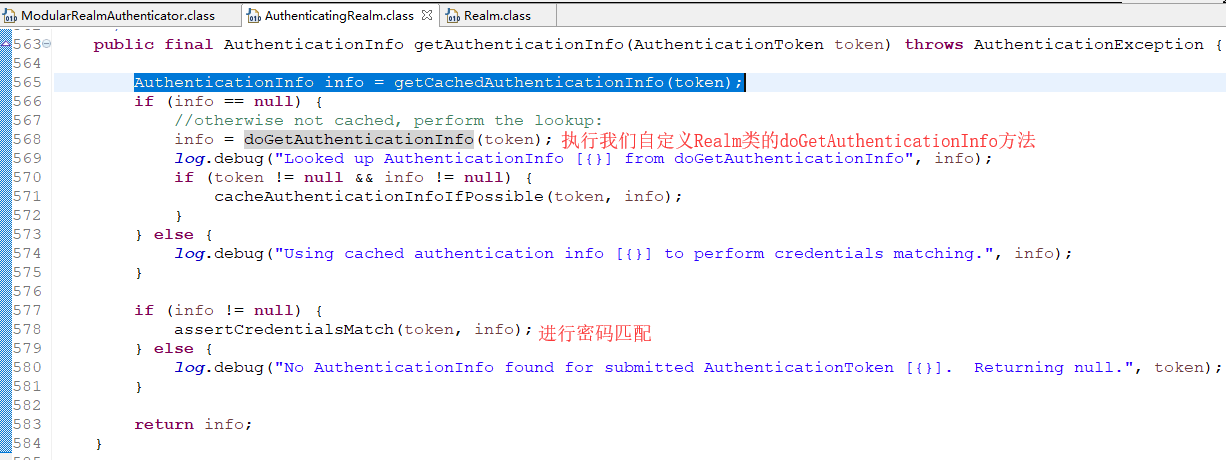
由于仅配置一个Realm，所以会执行doSingleRealmAuthentication(realms.iterator().next(), authenticationToken); 方法，并且将realm和token作为参数传入。这里的Realm就是我们自定义的MyShiroRealm类。

查看doSingleRealmAuthentication方法：



查看getAuthenticationInfo方法：

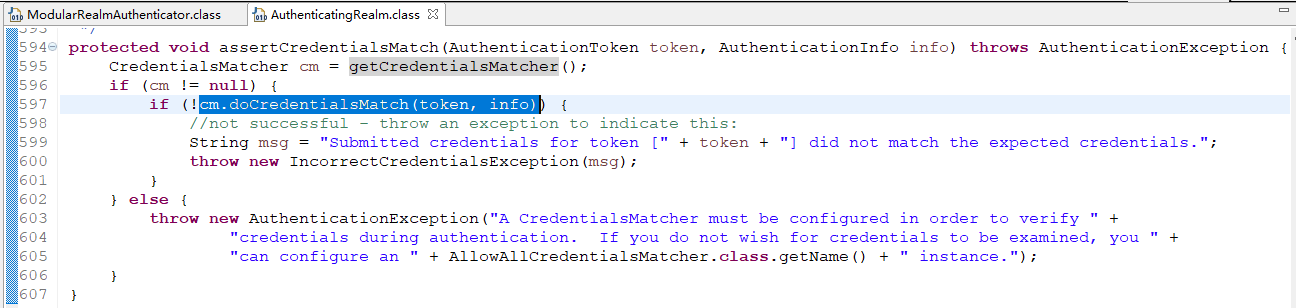
--该方法从realm中获取AuthenticationInfo信息，并对其进行密码匹配。



该方法先从shiro缓存中读取用户信息，如没有则从realm中获取。如果是第一次登录，缓存肯定没有认证信息，所以会执行doGetAuthenticationInfo(token)方法。

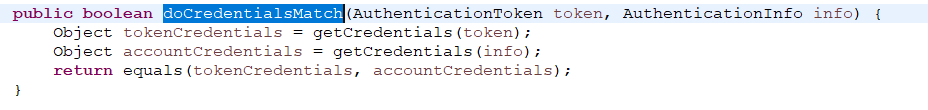
doGetAuthenticationInfo()方法就是我们自定义MyshiroRealm的中重写的方法。到此成功获取到AuthenticationInfo信息。

查看assertCredentialsMatch方法：



首先获取一个CredentialsMatcher对象，译为凭证匹配器。

再看一下cm.doCredentialsMatch(token,info)：



这里用到equals方法对token中加密的密码和从数据库中取出来的info中的密码进行对比。认证成功则返回true；失败则返回false。