# SpringSecurity

SpringSecurity是一套针对JavaEE企业应用的应用层安全框架，强依赖spring。

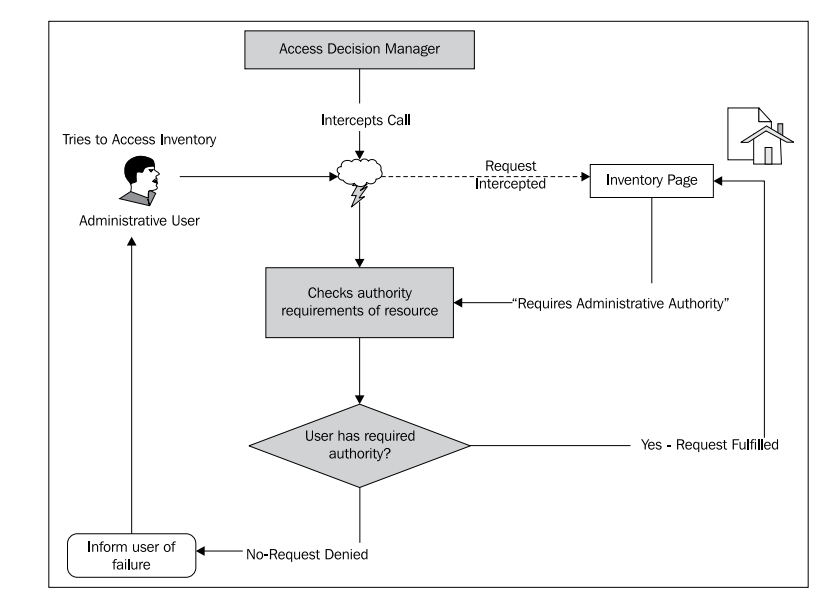
Authentication认证

支持的认证方式：

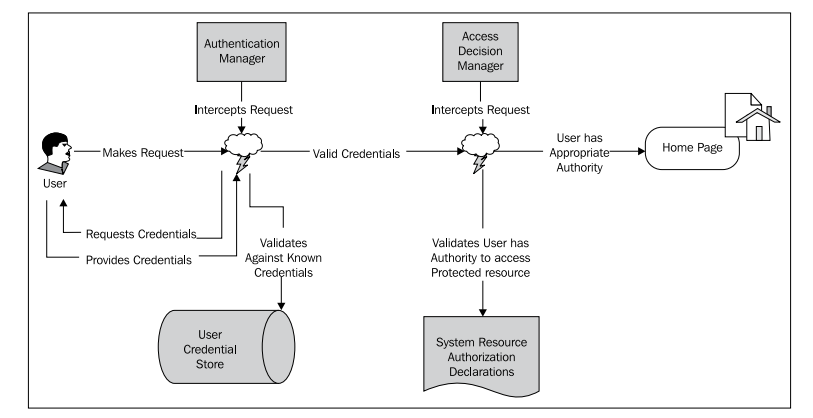
Http基本认证、http摘要认证、 X.509客户端证书交换、LDAP、表单提交、OpenID、

Authorization授权

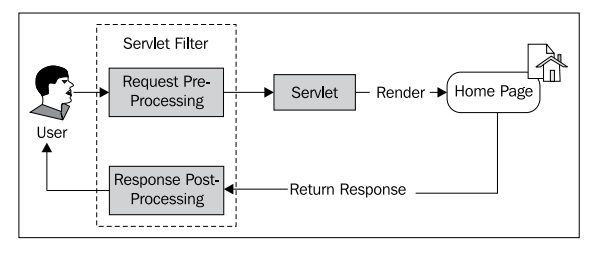
有一个名为访问决策管理器（access decision manager）的组件来负责决定一个安全实体是不是有适当的访问权限，判断基于安全实体具备的权限与被请求资源所要求资源的匹配情况。



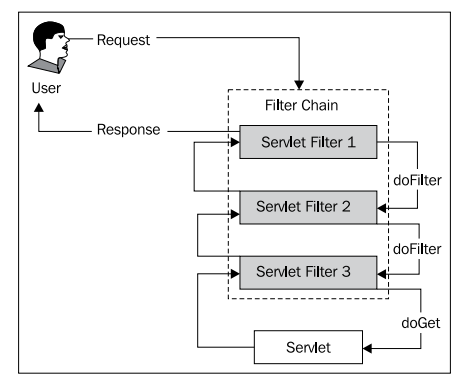
**Web支持**

****

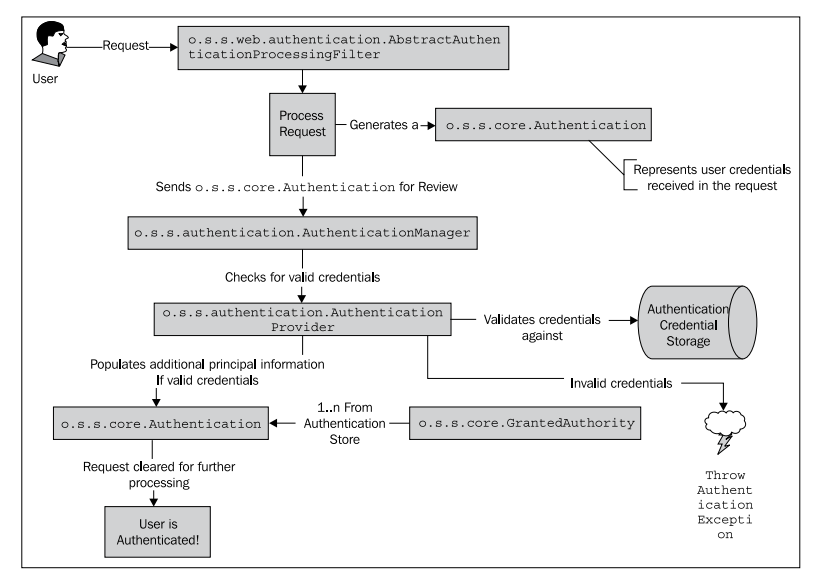
**利用javax.servlet.Filter实现请求拦截**

****

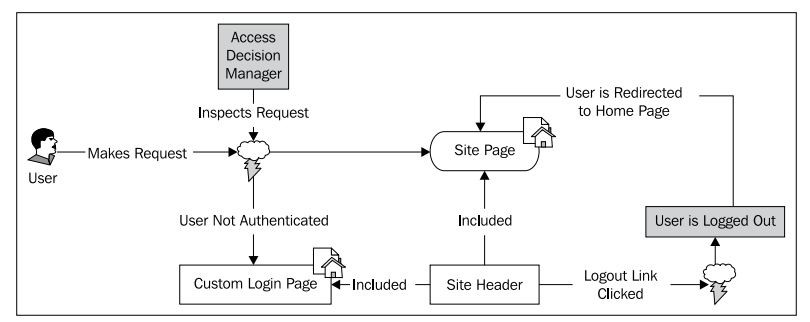
SpringSecurity在spring容器内部自动配置并建立了多个过滤器，它们通过使用Servlet规范中的javax.servlet.FilterChain顺序组合起来，请求按照一定的顺序从一个过滤器到下一个穿过整个过滤器链，最终到达目标servlet（springMVC的DispatcherServlet或者其他mvc框架的前端控制器），当servelt处理完请求并返回一个response时，过滤器链按照相反的顺序再次穿过所有的过滤器。



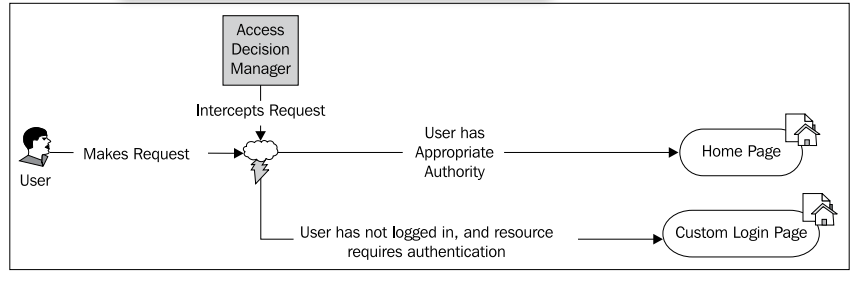
SpringSecurity涉及到认证功能的重要接口在下边的图标中有一个概览性的描述：



登陆和注销的流程



登陆流程



AbstractAuthenticationProcessingFilter

收集用户信息及凭证，创建一个部分完整的Authentication对象以在链中传递凭证信息。

AuthenticationManager

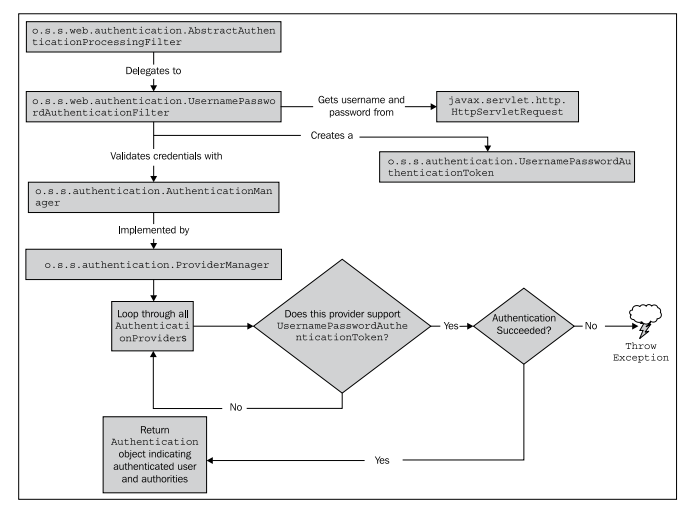
它用来校验用户的凭证信息，或者会抛出一个特定的异常（校验失败的情况）或者完整填充Authentication对象，将会包含了权限信息。

AuthenticationProvider

它为AuthenticationManager提供凭证校验。一些AuthenticationProvider的实现基于凭证信息的存储，如数据库，来判定凭证信息是否可以被认可。

Authentication存储了用户的详细信息，包括唯一标识（如用户名）、凭证信息（如密码）以及本用户被授予的一个或多个权限（GrantedAuthority）。开发人员通常会使用Authentication对象来获取用户的详细信息，或者使用自定义的认证实现以便在Authentication对象中增加应用依赖的额外信息。

基于web用户名和密码认证的请求下，这些类的处理过程：



用户名及密码的提取在UsernamePasswordAuthenticationFilter中指定

**springSecurity与spring容器的关系**

org.springframework.web.context.ContextLoaderListener

监听servlet容器的生命周期，初始化时创建spring容器并与servletContext绑定

org.springframework.web.filter.DelegatingFilterProxy

（Filter初始化在ServletContextListener之后）

初始化时从servletContext中获取spring容器，并根据filterName在spring容器中查找相应的Filter，并保持引用，等请求到来时直接代理给该Filter

在web.xml中配置DelegatingFilterProxy，并将该filter的名称叫做springSecurityFilterChain的FilterChainProxy（springSecurity框架默认在容器中创建的过滤器实现了Filter接口）

FilterChainProxy引用了所有在框架中定义的SecurityFilterChain，每个SecurityFilterChain又引用了一套Filter，FilterChainProxy根据SecurityFilterChain自身的matches判断是否将请求代理到该SecurityFilterChain引用的Filter上，默认的DefaultSecurityFilterChain聚合了多个Filter，并带有一个RequestMatcher，默认的实现是AntPathRequestMatcher，也可以用转换为正则表达式的实现

http标签的security设置为none，则不设置任何filter，同时SecurityContext为null

http标签总会创建一个SecurityFilterChain，并添加核心的Filter，其他非核心的Filter则根据配置的属性，自动添加到SecurityFilterChain，并且Filter的位置是固定的。

核心的Filter有:

SecurityContextPersistenceFilter

ExceptionTranslationFilter

FilterSecurityInterceptor.

http标签默认使用的**access-decision-manager为**AffirmativeBased，AffirmativeBased只要有同意票就通过，同时支持RoleVoter和AuthenticatedVoter.

多个http标签，使应用同时支持statefule和stateless成为可能

Session管理

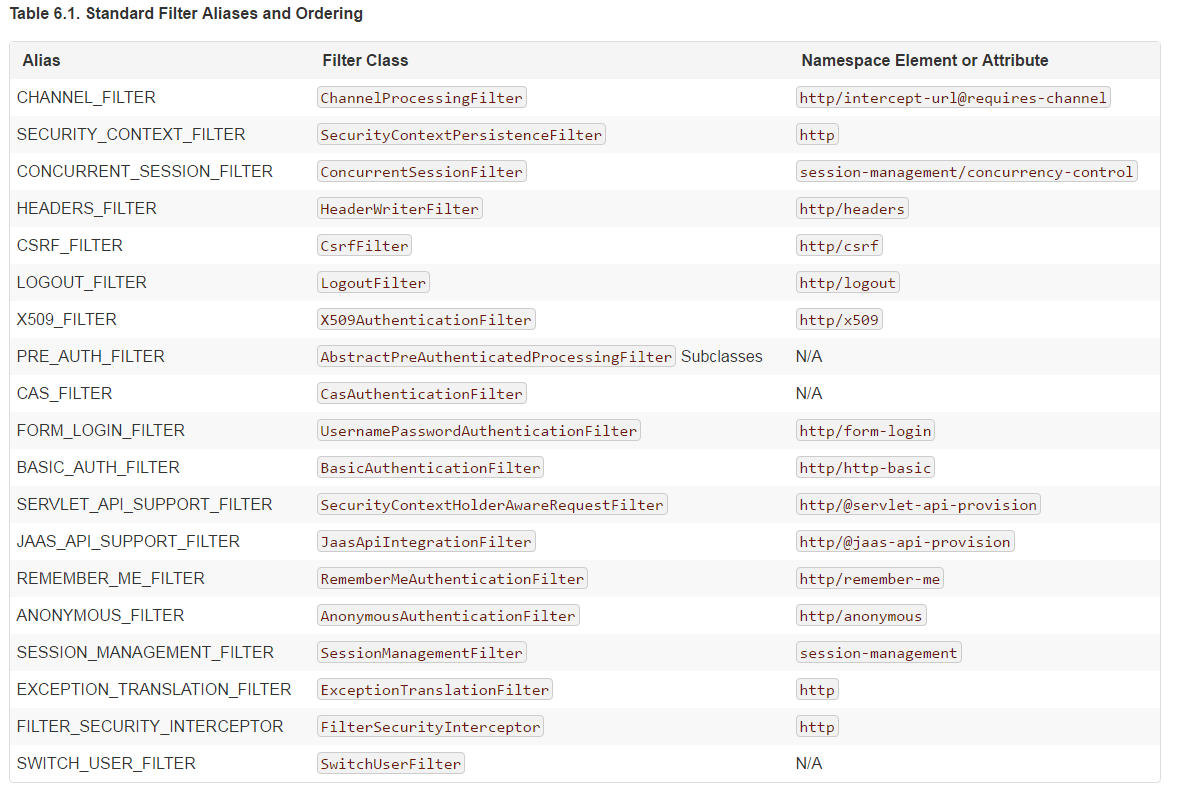
检测session过期

可以指定**invalid-session-url属性，在session过期时重定向到此页面**

并发session控制

可限制客户端同时在线的个数

Filter别名和排序



@PreAuthorize 可使用基于spel表达式的权限判断

@Secured 通过指定安全属性

ExceptionTranslationFilter

异常拦截

LoginUrlAuthenticationEntryPoint

重定向到登陆页面

UsernamePasswordAuthenticationFilter

登陆验证

FilterSecurityInterceptor

权限验证

LogoutFilter注销过滤器

可配置多个LogoutHandler和一个LogoutSuccessHandler

先执行LogoutHandler后执行LogoutSuccessHandler

LogoutHandler不能抛异常，用来做清理工作

SecurityContextLogoutHandler实现了LogoutHandler，清除SecurityContextHolder的内容，默认放到最后一个

LogoutHandler的实现有：

• PersistentTokenBasedRememberMeServices

• TokenBasedRememberMeServices

• CookieClearingLogoutHandler

• CsrfLogoutHandler

• SecurityContextLogoutHandler

LogoutSuccessHandler用来注销之后的重定向

LogoutSuccessHandler的实现有：

• SimpleUrlLogoutSuccessHandler（基于页面的重定向）

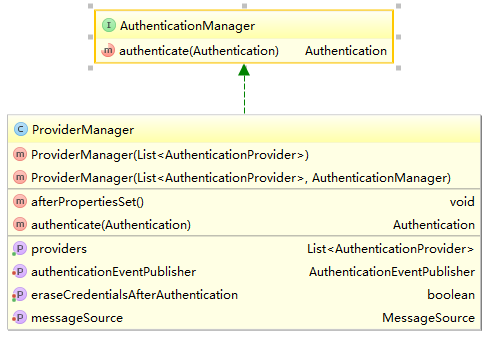
• HttpStatusReturningLogoutSuccessHandler（rest场景下使用返回状态码）

Authentication验证服务

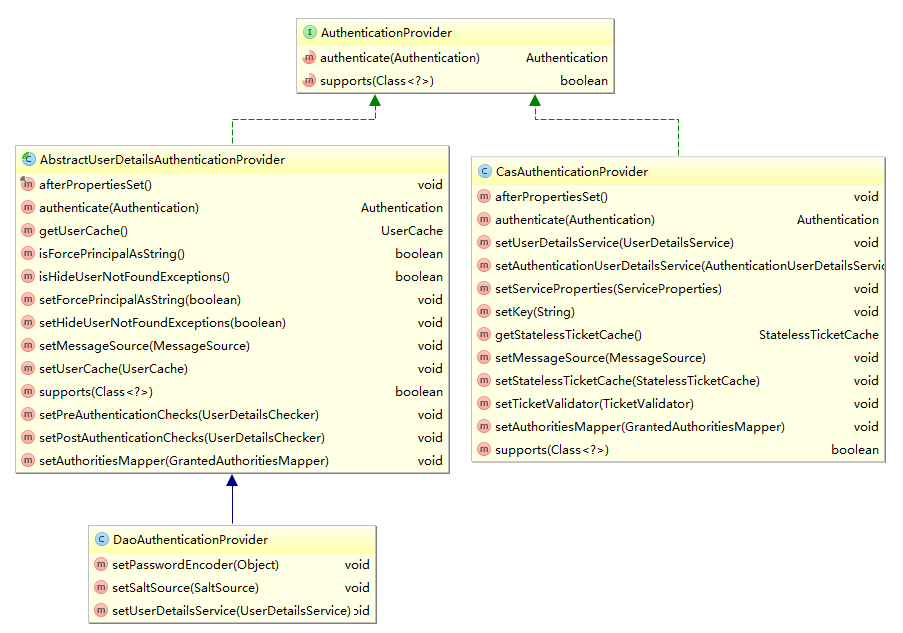
验证服务是通过接口AuthenticationManager，默认实现为ProviderManager

把验证请求代理给AuthenticationProvider，ProviderManager的作用就是一个代理，这样就可以支持多个数据源，每个数据源对应一个AuthenticationProvider

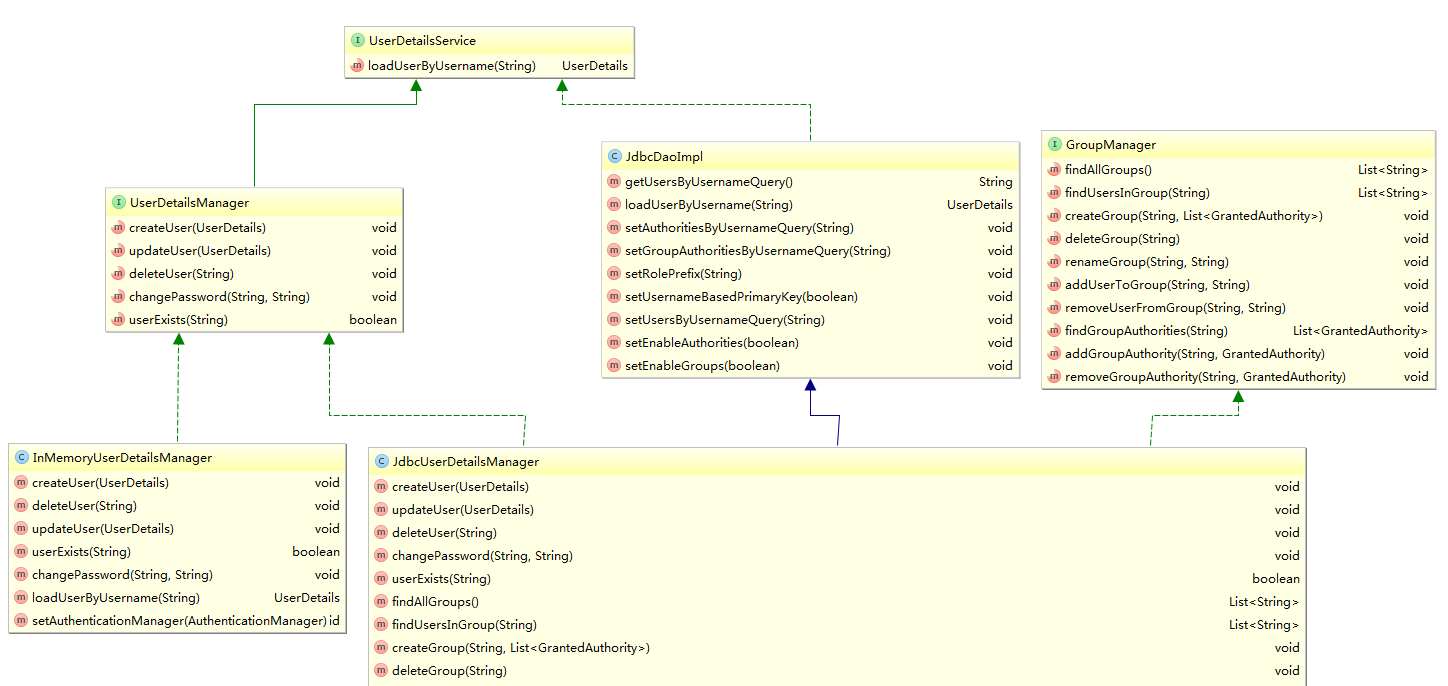
如果AuthenticationProvider未找到Authentication则应返回null



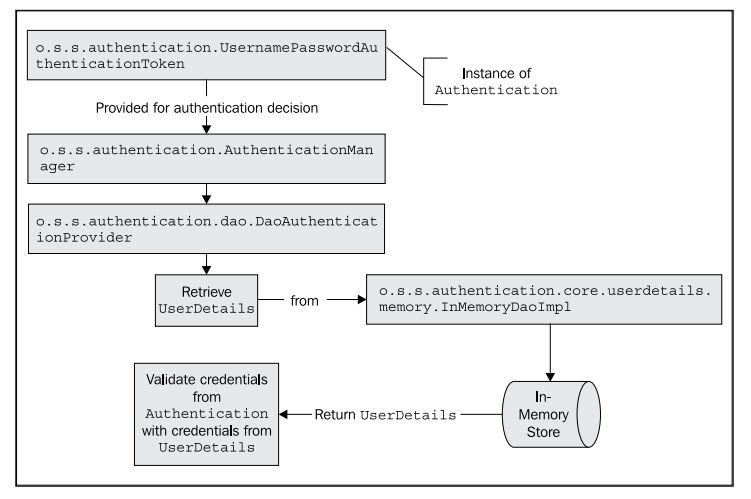
AuthenticationProvider的典型实现是DaoAuthenticationProvider，DaoAuthenticationProvider内部依赖旧版的PasswordEncoder来进行密码验证，却提供了setPasswordEncoder(Object passwordEncoder)，这样旧版和新版PasswordEncoder都能根据属性名称注入进来，setPasswordEncoder方法内部将新版代理为旧版接口



DaoAuthenticationProvider还依赖UserDetailsService来根据根据Username加载用户信息UserDetails，基于jdbc的默认实现JdbcDaoImpl，除了加载，spring还提供了维护UserDetails的接口UserDetailsManager，UserDetailsManager继承自UserDetailsService，在加载的基础上提供了crud的功能，基于jdbc的默认实现是JdbcUserDetailsManager，继承了JdbcDaoImpl

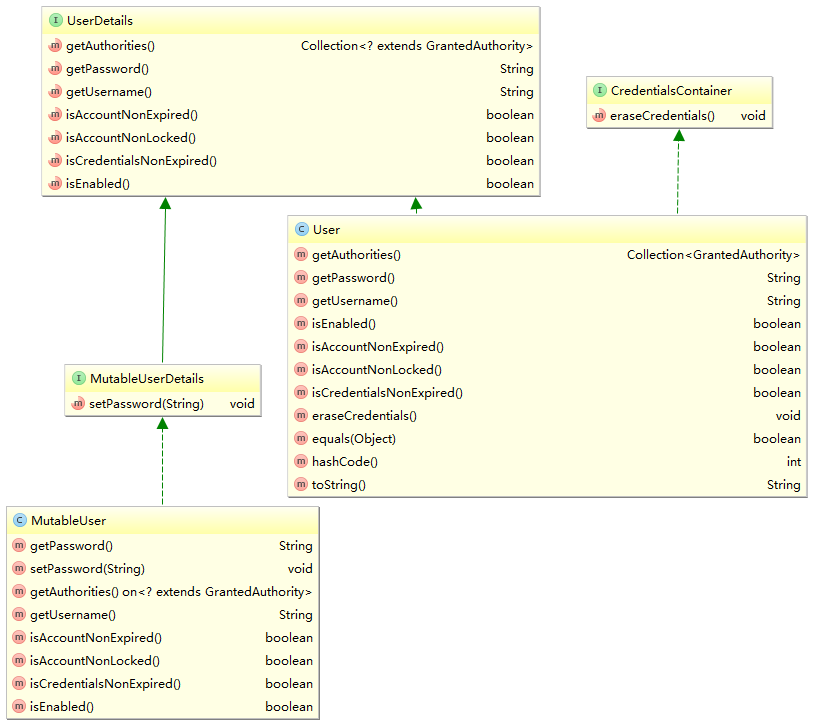


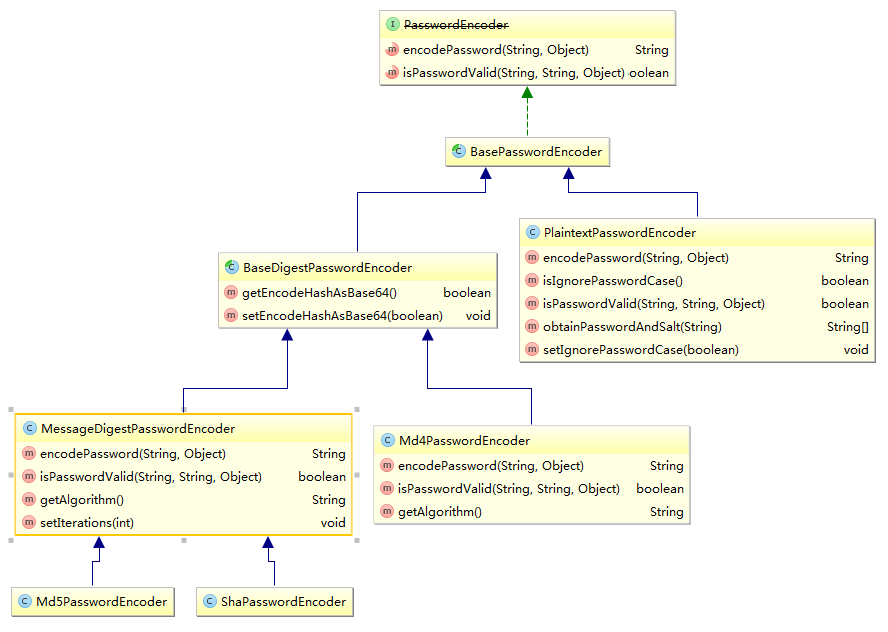
让我们更直观的看一下DaoAuthenticationProvider是如何交互的，从而AuthenticationManager提供认证支持：



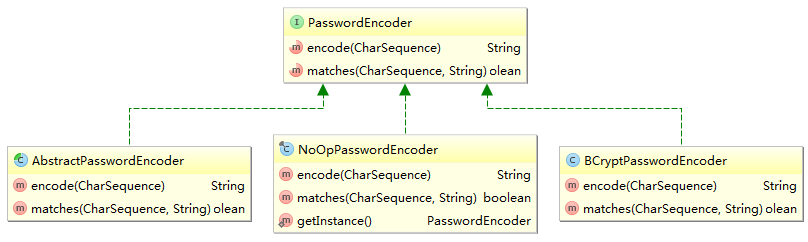
UserDetails与Authentication的功能很像，以下是它们的区别：

|  |  |
| --- | --- |
| 接口 | 目的 |
| Authentication | 它存储安全实体的标识、密码以及认证请求的上下文信息。它还包含用户认证后的信息（可能会包含一个UserDetails的实例）。通常不会被扩展，除非是为了支持某种特定类型的认证。 |
| UserDetails | 为了存储一个安全实体的概况信息，包含名字、e-mail、电话号码等。通常会被扩展以支持业务需求。 |





新版PasswordEncoder，不用提供salt参数，而是自动生成salt



Bcrypt算法可以解决大部分应用场景，如需自定义或者兼容就系统，可以继承AbstractPasswordEncoder

内置异常

所有认证相关的异常都继承自o.s.s.core.AuthenticationException基类。

AuthenticationException包含两个域：

authentication：存储关联认证请求的Authentication实例

extraInformation：根据特定的异常可以存储额外的信息。如UsernameNotFoundException在这个域上存储了用户名。

常见的异常类型：

|  |  |  |
| --- | --- | --- |
| 异常类 | 何时抛出 | extraInformation内容 |
| BadCredentialsException | 如何没有提供用户名或者密码与认证存储中用户名对应的密码不匹配 | UserDetails |
| LockedException | 如果用户的账号被发现锁定了 | UserDetails |
| UsernameNotFoundException | 如果用户名不存在或者用户没有被授予的GrantedAuthority | String（包含用户名） |

这些以及其他的异常将会传递到过滤器链上，通常将会被request请求的过滤器捕获并处理，要么将用户重定向到一个合适的界面（登录或访问拒绝），要么返回一个特殊的HTTP状态码，如HTTP 403（访问被拒绝）。

**Web模块的过滤器栈**

ChannelProcessingFilter-->

SecurityContextPersistenceFilter-->

ConcurrentSessionFilter-->

UsernamePasswordAuthenticationFilter-->

SecurityContextHolderAwareRequestFilter-->将原生的HttpServletRequest包装，以便实现HttpServletRequest相应的安全方法：比如getRemoteUser、isUserInRole、logout、getUserPrincipal

RememberMeAuthenticationFilter-->

AnonymousAuthenticationFilter-->

ExceptionTranslationFilter-->

FilterSecurityInterceptor -->target

**FilterSecurityInterceptor**

在整个过滤器链的最后，基于filter的权限验证拦截器，继承自AbstractSecurityInterceptor，它的作用是判断一个特定的请求是被允许还是被拒绝，Authentication提供了一个方法（(List<GrantedAuthority> getAuthorities()），将会返回当前安全实体的一系列权限列表。授权的过程将使用这个方法提供的信息来决定一个特定的请求是否会被允许。

**<bean id**=**"filterSecurityInterceptor" class**=**"org.springframework.security.web.access.intercept.FilterSecurityInterceptor">**

**<property name**=**"authenticationManager" ref**=**"authenticationManager"/>**

**<property name**=**"accessDecisionManager" ref**=**"accessDecisionManager"/>**

**<property name**=**"securityMetadataSource">**

**<security:filter-security-metadata-source>**

**<security:intercept-url pattern**=**"/secure/super/\*\*" access**=**"ROLE\_WE\_DONT\_HAVE"/>**

**<security:intercept-url pattern**=**"/secure/\*\*" access**=**"ROLE\_SUPERVISOR,ROLE\_TELLER"/>**

**</security:filter-security-metadata-source>**

**</property>**

**</bean>**

**ExceptionTranslationFilter 异常翻译过滤器**

在FilterSecurityInterceptor之前对验证异常拦截并转为相应的http响应

ExceptionTranslationFilter拦截到AccessDeniedException

判断是否为匿名用户，未匿名用户则利用AuthenticationEntryPoint重定向到验证入口，AuthenticationEntryPoint依赖于认证机制，具体的form实现是LoginUrlAuthenticationEntryPoint

如果不为匿名用户，则说明该用户无权限访问，则利用AccessDeniedHandler来处理没权限的情况，默认的实现AccessDeniedHandlerImpl仅仅返回一个403状态码，我们可以实现此接口来自定义这种情况下应该怎样通知用户无权限，比如：返回一个页面，或者如果是基于ajax的页面，返回一个无权限的json字符串

DelegatingAccessDeniedHandler还可以将异常与AccessDeniedHandler对应，来根据异常类型分别采取不同的响应

ExceptionTranslationFilter的另一项职责是调用AuthenticationEntryPoint之前保存请求对象，在验证通过之后继续之前的请求

RequestCache用来保存请求对象，默认实现是HttpSessionRequestCache，把请求对象保存到session中

AbstractAuthenticationProcessingFilter会在验证成功后执行AuthenticationSuccessHandler，默认实现是SavedRequestAwareAuthenticationSuccessHandler，此实现重定向到RequestCache中保存的request对象的地址，并通过RequestCacheFilter拦截重定向的请求，并将请求与RequestCache中的请求进行匹配，如果匹配成功，则将请求对象替换成RequestCache中的请求对象

**SecurityContextPersistenceFilter**

在请求之间利用SecurityContextRepository保存SecurityContext，并在请求结束时清理SecurityContextHolder，清理是为了在线程被servlet容器的线程池收回之后，被其他线程错误的复用

**SecurityContextRepository**

SecurityContext仓库，默认基于session的HttpSessionSecurityContextRepository

**UsernamePasswordAuthenticationFilter**

用户名密码登陆认证过滤器 拦截提交的登陆表单，并对将用户名和密码组成UsernamePasswordAuthenticationToken交给authenticationManager进行校验，如果认证成功返回一个Authentication，并设置到SecurityContextHolder上面。

验证成功和失败的处理流程：

验证成功之后调用AuthenticationSuccessHandler接口，默认提供了

SimpleUrlAuthenticationSuccessHandler，成功之后重定向到指定页面

SavedRequestAwareAuthenticationSuccessHandler，成功之后重定向到之前请求的页面

失败调用AuthenticationFailureHandler接口，默认提供了

SimpleUrlAuthenticationFailureHandler重定向到指定页面

ExceptionMappingAuthenticationFailureHandler根据异常的不同，重定向到不同的页面

**基本认证和摘要认证**

基本认证经常与无状态客户端一起使用，每次请求都携带凭证信息。

由于基本认证在每次请求中都携带明文的密码（只是在请求头中将username:password以base64编码），所以只在加密传输层上使用

**BasicAuthenticationFilter**

**<bean id**=**"basicAuthenticationFilter"**

**class**=**"org.springframework.security.web.authentication.www.BasicAuthenticationFilter">**

**<property name**=**"authenticationManager" ref**=**"authenticationManager"/>**

**<property name**=**"authenticationEntryPoint" ref**=**"authenticationEntryPoint"/>**

**</bean>**

**<bean id**=**"authenticationEntryPoint"**

**class**=**"org.springframework.security.web.authentication.www.BasicAuthenticationEntryPoint">**

**<property name**=**"realmName" value**=**"Name Of Your Realm"/>**

**</bean>**

从请求头Authorization中获取用户名和密码，并交由authenticationManager对用户进行验证，验证失败流程导向BasicAuthenticationEntryPoint，返回401，要求用户重新输入用户名密码

**DigestAuthenticationFilter**

未使用authenticationManager，直接用userDetailsService验证，并将Authentication设置到SecurityContextHolder

**RememberMeAuthenticationFilter**

依赖RememberMeServices和AuthenticationManager

第一次登陆请求到达AbstractAuthenticationProcessingFilter并且登陆成功之后，调用RememberMeServices的loginSuccess将token写入Cookie或者数据库

Remember-me默认参数为remember-me，设置为true、1、on、yes，即可开启rememberMe功能

Session过期之后，请求经过RememberMeAuthenticationFilter，拦截器调用RememberMeServices的autoLogin方法，从请求中提取Cookie，并对Cookie的有效性验证，成功之后根据解析出的username，从userDetailsService中获取用户信息，加RememberMeServices的私钥组装RememberMeAuthenticationToken返回

拦截器将RememberMeAuthenticationToken提交给AuthenticationManager，providers中只有RememberMeAuthenticationProvider能处理，RememberMeAuthenticationProvider将RememberMeAuthenticationToken中的私钥与RememberMeAuthenticationProvider中的私钥对比完成验证

RememberMeServices有两种实现方式

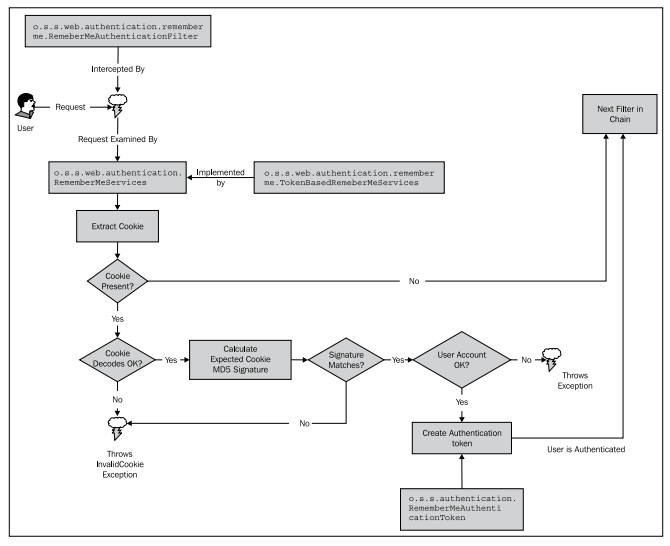
一种是将token的hash存在Cookie中，一种是将token存于数据库中

1. **TokenBasedRememberMeServices**基于hash令牌的方式

登陆成功之后，将Cookie发送给浏览器：Cookie值的算法

base64(username + ":" + expirationTime + ":" +md5Hex(username + ":" + expirationTime + ":" password + ":" + key))

Key：私钥，添加私钥是为了避免Cookie被修改



1. **PersistentTokenBasedRememberMeServices**基于持久化的方式

依赖持久化仓库PersistentTokenRepository

具体实现有两种：

基于内存的InMemoryTokenRepositoryImpl

基于数据库的JdbcTokenRepositoryImpl



Cookie的计算方式series:token:cookie

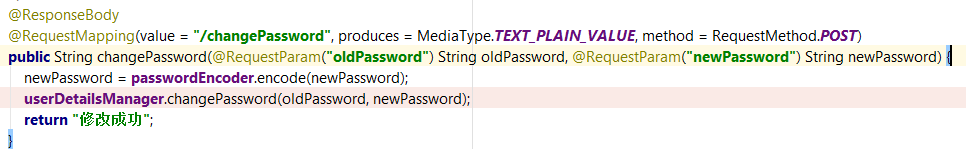
**修改密码**

UserDetailsManager中有修改密码的功能

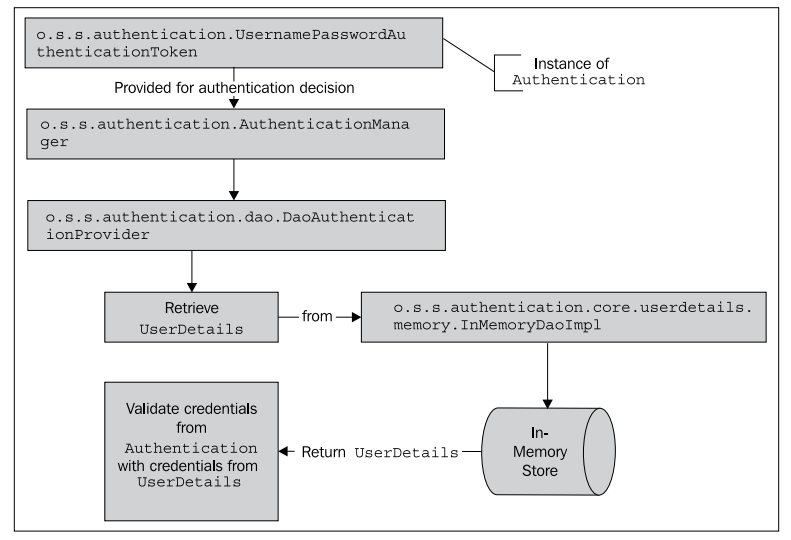
*/\*\*  
 \* Modify the current user's password. This should change the user's password in the  
 \* persistent user repository (datbase, LDAP etc).  
 \*  
 \** ***@param oldPassword*** *current password (for re-authentication if required)  
 \** ***@param newPassword*** *the password to change to  
 \*/***void** changePassword(String oldPassword, String newPassword);

其中oldPassword是客户端输入的明文密码，而newPassword需要经过PasswordEncoder加密之后的密码

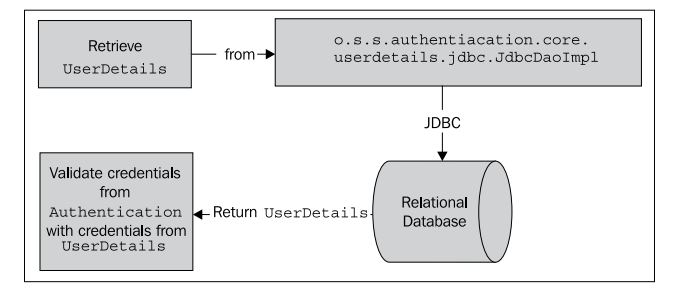
可能的Controller实现



其中UserDetailsManager依赖AuthenticationManager来验证旧密码是否正确，而AuthenticationManager又依赖UserDetailsManager提供的UserDetailsService功能，所以产生了循环引用的问题，目前还未找到合适的处理方式



向jdbc中查找用户信息



关于组和角色

SpringSecurity中将组理解为人员的聚集、而角色则视为一种粒度更大的一种权限

基于组的管理方式的简便性和扩展性带来的价值远远超过了它稍微增加的复杂性。这种将用户权限集中到组中的技术通常叫做基于组的访问控制（Group-Based Access Control ，GBAC）。

基于组的访问控制几乎在市面上任何安全的操作系统和软件包中都能看到。微软的活动目录（Active Directory，AD)是大范围使用GBAC的典型实现，它把AD的用户纳入组中并给组授权权限。通过使用GBAC，能够指数级得简化对大量基于AD组织的权限管理。

**对CSRF（跨站请求伪造）的支持**

防止非法跨域请求

**Synchronizer Token Pattern**

只在普通用户使用浏览器的情况下使用

1、查询用GET，修改用其他

2、在所有的PATCH, POST, PUT, DELETE 请求中添加**CsrfToken**参数

3、在请求参数、请求头、cookie中添加

Xml配置默认启用了csrf

**利用CORS实现跨域请求**

CORS必须在其他认证filter之前

@Bean

CorsConfigurationSource corsConfigurationSource() {

CorsConfiguration configuration = **new** CorsConfiguration();

configuration.setAllowedOrigins(Arrays.asList(***"https://example.com"***));

configuration.setAllowedMethods(Arrays.asList(***"GET"***,***"POST"***));

UrlBasedCorsConfigurationSource source = **new** UrlBasedCorsConfigurationSource();

source.registerCorsConfiguration(***"/\*\*"***, configuration);

**return** source;

}

关于cors的介绍：http://www.ruanyifeng.com/blog/2016/04/cors.html

**Http响应头**

**Cache Control**：启用cache control则不启用缓存，即使是在https连接的情况下

**Content Type Options**：禁用浏览器的内容类型推断，浏览器的内容类型推断会给攻击者提供便利，使浏览器来执行攻击者的脚本

**HTTP Strict Transport Security (HSTS)**：HSTS可以很大程度上解决SSL剥离攻击，因为只要浏览器曾经与服务器创建过一次安全连接，之后浏览器会强制使用HTTPS，即使链接被换成了HTTP。

**HTTP Public Key Pinning (HPKP) HTTP公钥钉**：

**X-Frame-Options**：是否允许页面被嵌入frame，

当响应头设置为X-Frame-Options: DENY时，该页面不允许被嵌入到frame中；

当响应头设置为X-Frame-Options: SAMEORIGIN时，允许同一个域名下的页面被嵌入frame中；

遗留的浏览器可能不会认识这个指令，这时需要frame breaking代码：

<style id="antiClickjack">body{display:none !important;}</style>

<script type="text/javascript">

if (self === top) {

var antiClickjack = document.getElementById("antiClickjack");

antiClickjack.parentNode.removeChild(antiClickjack);

} else {

top.location = self.location;

}

</script>

**X-XSS-Protection：跨站点脚本攻击**

攻击者在请求参数或者提交表单字段中存有执行脚本，当脚本内容被保存到数据库时，再次查询出来会对其他访问用户进行攻击，浏览器默认开启xss保护之后会自动替换脚本内容，但有时阻塞比替换更好，框架默认自动向响应头添加X-XSS-Protection: 1; mode=block指令来阻塞用户提交非法字符

**Content Security Policy (CSP):网页安全政策**

**Content-Security-Policy：**CSP 的实质就是白名单制度，开发者明确告诉客户端，哪些外部资源可以加载和执行，等同于提供白名单。它的实现和执行全部由浏览器完成，开发者只需提供配置。

Content-Security-Policy: script-src https://trustedscripts.example.com; report-uri /csp-report-endpoint/

**Content-Security-Policy-Report-Only：**表示不执行限制选项，只是记录违反限制的行为。它必须与report-uri选项配合使用。

Content-Security-Policy-Report-Only: script-src 'self' https://trustedscripts.example.com; report-uri /csp-report-endpoint/

参考地址：http://www.ruanyifeng.com/blog/2016/09/csp.html

**Session管理**

Session相关功能由SessionManagementFilter和SessionAuthenticationStrategy组成，主要功能有

防止会话固定攻击

检测会话过期

限制验证用户同时在线的session个数

**SessionManagementFilter**

如果SecurityContextRepository包含SecurityContext直接放过

否则，判断SecurityContextHolder是否存在SecurityContext，如果存在则被视为被AbstractAuthenticationProcessingFilter刚刚验证过，则调用SessionAuthenticationStrategy来创建一个新的session，防止固化session攻击

否则，判断session是否过期，并调用InvalidSessionStrategy

**并发控制**

**ConcurrentSessionControlAuthenticationStrategy**

默认规则是当session个数到达最大值时，使最后一个未使用的session无效

**SessionRegistry**

默认实现类SessionRegistryImpl监听SessionDestroyedEvent（需要注册HttpSessionEventPublisher）事件，session过期时从注册器中移除，此类还保存了用户的会话信息及所有的用户信息，可用来统计在线用户，及他们所有的会话信息

**ConcurrentSessionFilter**

及时更新已过期的session信息，并重定向到session过期提示页面

**防止固定会话攻击**

**SessionFixationProtectionStrategy**默认值,验证成功后创建一个新的session

**ChangeSessionIdAuthenticationStrategy**验证成功后改变sessionid

**新会话注册**

**RegisterSessionAuthenticationStrategy**

并发控制只对会话的数量进行了控制（默认为移除旧的会话），并没有注册新的会话，该策略实现了对新会话的注册功能

**组合各种策略**

**CompositeSessionAuthenticationStrategy**

可以组合以上策略来实现多种功能，且必须要以以上的顺序组合策略

**Anonymous Authentication匿名认证**

有时一个页面既可以已验证的用户来访问，也可以未验证的用户来访问，只是展示的内容不同，此时只通过url来控制访问已不能达到要求。所以springSecurity引入了一个Anonymous Authentication，与未认证的用户没有什么不同。同时在审计拦截器调用SecurityContextHolder时总会返回一个Authentication，而不会返回null

**核心接口**

**SecurityContextHolder访问SecurityContext的入口**

将SecurityContext绑定到当前的执行线程，

SecurityContextHolderStrategy当前SecurityContext的存放策略（默认为策略1）

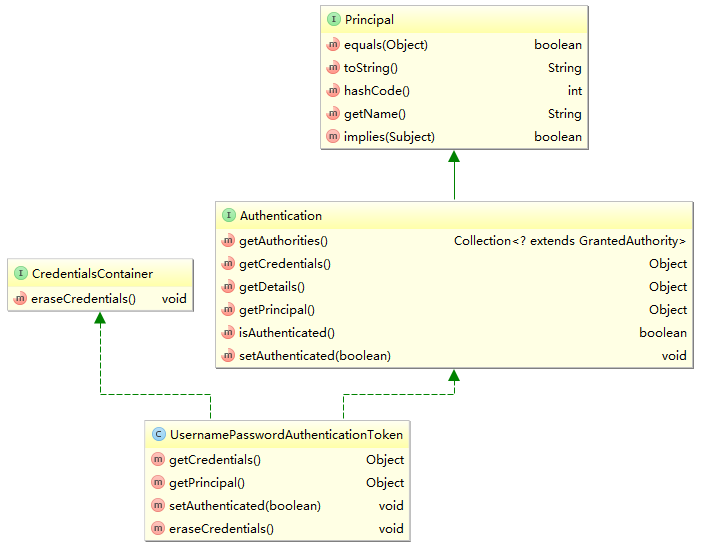
内置三种策略：

1）存放到当前线程的ThreadLocal

2）全局存放（所有线程共用一个，通常用在客户端，如swing）

3）可继承的ThreadLocal（可在子线程中获取父线程中的SecurityContext）

**SecurityContext 持有验证主体Authentication**



**Authentication验证主体详细信息的spring内置表示**

可获取的信息：权限、凭证、主体、其他信息（如登陆信息）、是否认证

**UserDetails提供了构建Authentication的必要信息**

用来表示主体的框架内置接口，可将此接口视为框架与应用内部用户实体的适配器，spring提供了此接口的内置实现User

**UserDetailsService根据Username加载用户信息UserDetails**

基于jdbc的默认实现JdbcDaoImpl，除了加载，spring还提供了维护UserDetails的接口UserDetailsManager，UserDetailsManager继承自UserDetailsService，在加载的基础上提供了crud的功能，基于jdbc的默认实现是JdbcUserDetailsManager，继承了JdbcDaoImpl

在验证成功之后，UserDetails被用来创建Authentication，并绑定到SecurityContextHolder

# 验证流程

1、根据用户提供的用户名和密码组成UsernamePasswordAuthenticationToken（实现了Authentication接口）

2、将token传递给AuthenticationManager的authenticate

3、如果验证成功，AuthenticationManager返回一个全新的Authentication

4、调用SecurityContextHolder.getContext().setAuthentication(...)将authentication设置到执行线程

如果第2步抛出AuthenticationException，则认为验证失败，抛出的AuthenticationException的子异常有：

UsernameNotFoundException 账号不存在

DisabledException 账户不可用

LockedException 账户被锁定时抛出

BadCredentialsException 凭证错误或者用户名不存在时抛出

AccountExpiredException 账户是否过期

CredentialsExpiredException 凭证是否过期

实际上我们可以在AbstractSecurityInterceptor执行权限验证之前，

直接将Authentication设置到SecurityContextHolder，这样使在集成旧项目时获得

最大的灵活性

Web应用中的认证流程

1. 用户访问受保护的资源
2. 框架判断当前请求未认证，服务器重定向到认证入口
3. 服务器判断当前的凭证是否有效，如果有效，流程继续，如果无效，服务器重定向到认证入口
4. 认证有效，服务器继续处理之前的请求（访问受保护资源）
5. 判断是否有权限，如果没有权限，返回403

涉及到的类：

**ExceptionTranslationFilter**

用来捕获安全相关的异常（比如AbstractSecurityInterceptor抛出的异常），并重定向或者返回客户端适当的http状态码

**AuthenticationEntryPoint**

表示处理导向认证入口的策略类，主要负责向用户提供认证入口，被ExceptionTranslationFilter在接收到重新认证的时候调用，在2、3步的时候使用

**在不同的请求之间存储SecurityContext**

SecurityContextPersistenceFilter：每次请求到来时从SecurityContextRepository（默认实现HttpSessionSecurityContextRepository）

中获取SecurityContext，并绑定到SecurityContextHolder，在请求结束时清除SecurityContextHolder中的SecurityContext

这样做还保证了并发访问时，多个线程绑定了同一个SecurityContext，因为都是从session中获取，还可以配置

# 授权

**GrantedAuthority 主体的应用级别权限表示**

Authentication的getAuthorities()可返回该主体所有的GrantedAuthority

一般情况下GrantedAuthority是应用范围的权限，不能指派给领域对象

如果GrantedAuthority的getAuthority()返回值为null，则此GrantedAuthority为复杂的，AccessDecisionManager需要特殊处理此类型的GrantedAuthority

**Pre-Invocation Handling调用前处理**

**AccessDecisionManager访问控制**

AccessDecisionManager的默认实现是使用AffirmativeBased和RoleVoter、AuthenticatedVoter

AbstractSecurityInterceptor通过调用AccessDecisionManager来判断权限

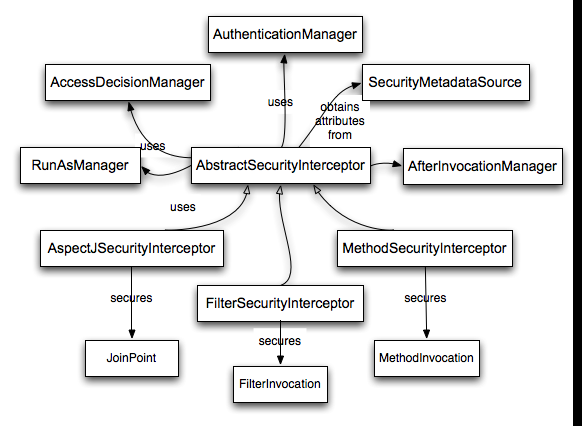
**基于投票的实现**

AffirmativeBased只要有同意票就通过

UnanimousBased只要有反对票就不通过

ConsensusBased如果同意票大于反对票，则通过

如果反对票大于同意票，则不通过

如果同意票与反对票相等，则根据配置来判断是否通过

**AccessDecisionVoter投票器**

**投票器对用户是否可以访问资源进行投票，有三种投票结果**

ACCESS\_GRANTED 投票器允许对资源的访问

ACCESS\_ABSTAIN 投票器拒绝对资源的访问

ACCESS\_DENIED 投票器对是否能够访问做了弃权处理

原因可能有：不支持这种请求类型的判断、没有确凿的判断信息

RoleVoter专门针对角色的投票器，如果目标方法的配置属性以“ROLE\_”开头，则必须投同意或者反对票，否则投弃权票

AuthenticatedVoter专门针对内置Authentication的投票器，如果目标方法的配置属性为IS\_AUTHENTICATED\_ANONYMOUSLY、IS\_AUTHENTICATED\_FULLY、IS\_AUTHENTICATED\_REMEMBERED，则必须投同意或者反对票，否则投弃权票

RoleHierarchyVoter专门针对角色继承的情况下的投票器，继承自RoleVoter，需要注入RoleHierarchy，RoleHierarchy实现角色继承关系的查找功能

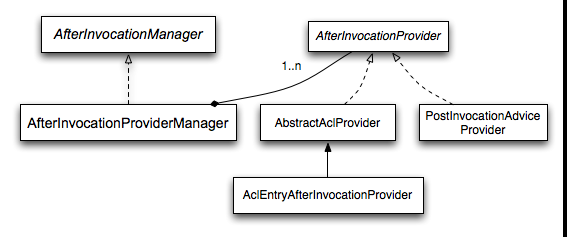
**WebExpressionVoter**实现了根据spel表达式web环境下来判断权限的逻辑

PreInvocationAuthorizationAdviceVoter实现了根据PreInvocationAuthorizationAdvice来判断权限的逻辑，默认实现了method情况下的表达式计算逻辑

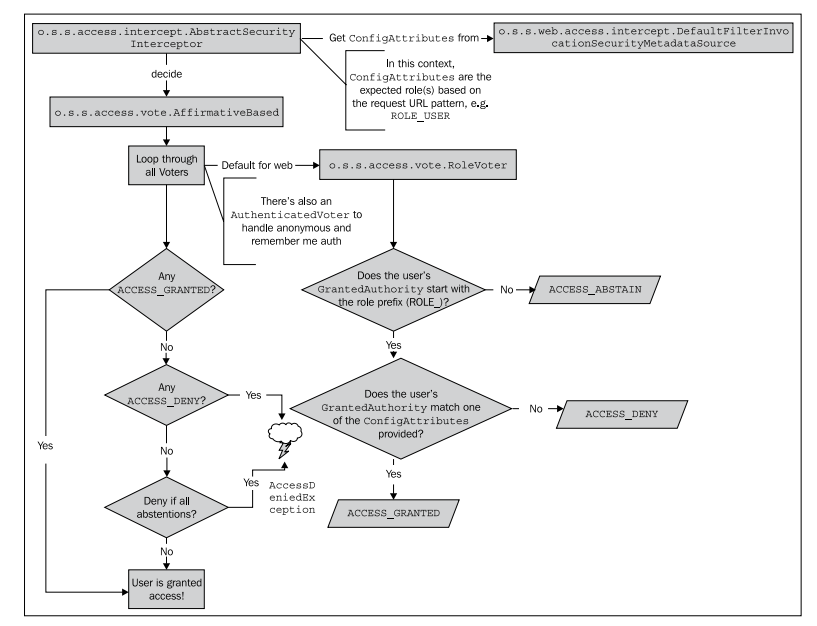
默认实现由RoleVoter、AuthenticatedVoter组合实现

**AfterInvocationManager 调用后处理**

默认实现AfterInvocationProviderManager，用来管理AfterInvocationProvider



对web请求的默认认证检查流程



**Aop联盟安全拦截器**

MethodSecurityInterceptor拦截MethodInvocation方法调用，继承自AbstractSecurityInterceptor并实现了MethodInterceptor接口来拦截方法调用

拦截方法的同时，将验证流程交给父类AbstractSecurityInterceptor（FilterSecurityInterceptor也继承于此）

MethodSecurityInterceptor依赖的MethodSecurityMetadataSource用来获取方法上的配置属性，专门针对xml配置的MapBasedMethodSecurityMetadataSource用来保存方法与配置属性的映射，其他实现则实现基于注解的MetadataSource

**AspectJ安全拦截器**

AspectJMethodSecurityInterceptor实现了基于aspectJ的方法拦截器，不同于aop联盟安全拦截器，此拦截器基于编译阶段的代码植入，主要用在领域实体的安全拦截，而MethodSecurityInterceptor主要用在Service层的安全拦截

**基于SPEL的访问控制**

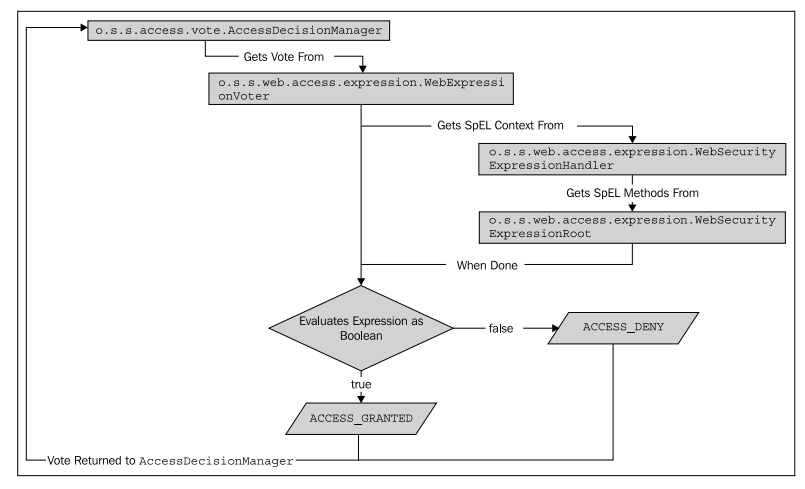
springEL根据一个跟对象来进行表达式的计算，springSecurity为web和方法安全提供了两个跟对象，WebSecurityExpressionRoot和MethodSecurityExpressionRoot，都继承自SecurityExpressionRoot，因此这两种情况下有一些公共的方法

**Web Security Expressions**

当启动配置文件中的use-expressions属性时，自动添加WebExpressionVoter，并启用web表达式，可在access属性中使用表达式，此时RoleVotor将会失效，所以必须要将access全部改为spel表达式

借助于WebSecurityExpressionHandler 来创建表达式评估上下文EvaluationContext，并将其与WebSecurityExpressionRoot关联来访问该对象的方法，

表达式的根对象就是WebSecurityExpressionRoot，并只能引用该类及父类的方法



SecurityExpressionRoot中的方法及属性

|  |  |  |
| --- | --- | --- |
| 属性 | 描述 | 例子 |
| permitAll | 任何用户均可访问 | access="permitAll" |
| denyAll | 任何用户均不可访问 | access="denyAll" |
| anonymous | 匿名用户可访问 | access="anonymous" |
| authenticated | 检查用户是否认证过 | access="authenticated" |
| rememberMe | 检查用户是否通过remember me 功能认证的 | access="rememberMe" |
| fullyAuthenticated | 检查用户是否通过提供完整的凭证信息来认证的 | access="fullyAuthenticated |

|  |  |  |
| --- | --- | --- |
| hasRole(role) | 用于匹配一个使用GrantedAuthority 的角色（类似于 RoleVoter ） | access="hasRole('ROLE  USER')" |
| hasAnyRole(role) | 用于匹配一个使用GrantedAuthority 的角色列表。用户匹配其中的任何一个均可放行。 | access="hasRole('ROLE\_  USER','ROLE\_ADMIN')" |

**Method Security Expressions**

**@Pre and @Post Annotations**

开启@PreAuthorize, @PreFilter, @PostAuthorize, @PostFilter注解支持

**<global-method-security pre-post-annotations**=**"enabled"/>**

@PreAuthorize最常使用的注解，目标方法执行前用来判断方法是否可以调用，可以引用方法参数进行判断

@PostAuthorize 较少用的，目标方法执行后进行判断，可以用returnObject引用返回对象

@PreFilter可以对指定参数为集合的元素进行过滤，表达式为false的将被剔除集合，filterTarget属性用来指定迭代的集合，表达式内置对象filterObject引用迭代的当前对象

@PostFilter可以对方法的返回值为集合或数组的元素进行过滤，表达式为false的将被剔除集合，表达式内置对象filterObject引用迭代的当前对象

表达式中可以根据名称引用被调用方法的参数，方法参数的名称是根据DefaultSecurityParameterNameDiscoverer来检测的

默认提供了org.springframework.security.access.method.P

如果springdata在类路径下，也会添加org.springframework.data.repository.query.Param

将这两个注解代理给AnnotationParameterNameDiscoverer，将java8的功能代理给DefaultParameterNameDiscoverer来进行具体的方法参数的判断

“secure object”是指任何能被实施保护的对象，最常用的有两种：方法调用和web请求

利用aop来实现对方法进行权限检查

利用filter来实现对web请求的权限检查

每一种安全对象类型都有自己的拦截器类，拦截器类继承自AbstractSecurityInterceptor，如果AbstractSecurityInterceptor被调用的时候，SecurityContextHolder必须要有效的Authentication

AbstractSecurityInterceptor的流程：

1. 检查当前与当前请求关联的配置属性
2. 将当前的Authentication和配置属性提交给AccessDecisionManager来做权限验证
3. 更改Authentication对象
4. 验证成功，允许继续执行安全对象
5. 目标安全对象成功返回则调用AfterInvocationManager，如果抛异常则不调用AfterInvocationManager

Configuration attribute 配置属性，简单的字符串，也许表示的是角色名称、或者更复杂的含义

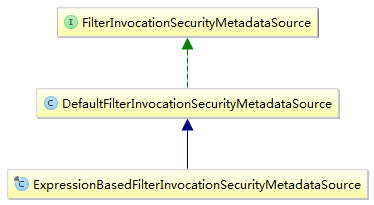
Spring框架内部用ConfigAttribute来表示，在框架配置上用注解或者access属性来表示

SecurityMetadataSource用来从安全对象中提取配置属性

SecurityMetadataSource的子接口主要分为两种：

FilterInvocationSecurityMetadataSource主要用来对web权限配置属性的提取

具体实现



DefaultFilterInvocationSecurityMetadataSource提取字符串类型的ConfigAttribute

ExpressionBasedFilterInvocationSecurityMetadataSource提取表达式类型的ConfigAttribute

MethodSecurityMetadataSource主要用来从方法上提取配置属性

注解解析步骤：(SpringSecurity不会查找跨越多层aop代理的目标对象，所以必须要将SpringSecurity的aop代理设置为最后一层代理，或者把注解放到接口上，或者自己实现查找逻辑)

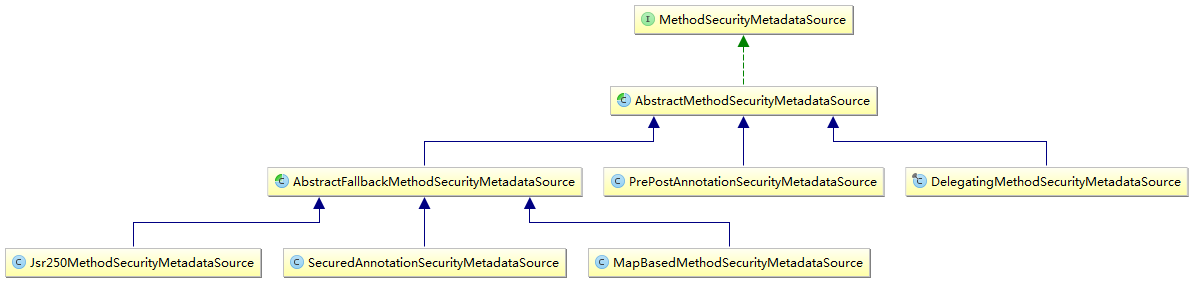
@Secured 指定配置属性

*JSR 250 规范注解*

@DenyAll 禁止任何人访问

@PermitAll 允许任何人访问

@RolesAllowed 只允许指定角色的人访问，如：@RolesAllowed("ADMIN") 自动转换为值为ROLE\_ADMIN的配置属性



PrePostAnnotationSecurityMetadataSource从*@PreFilter 、@PreAuthorize*

*@PostFilter 、@PostAuthorize*

注解中提取权限元数据，注解可放置在类和方法上，方法上的优先级更高

如果注解中没有任何前置条件则与*@PreAuthorize("permitAll")含义相同*

*只能放置spel表达式*

配置属性含义解析依赖于AccessDecisionManager的实现

例如：前缀ROLE\_表示这是一个角色，这样可以判断RoleVoter可以解析这个ConfigAttribute

**RunAsManager**

在执行受保护方法时临时更改Authentication

方法执行完，则恢复回原来的Authentication

权限表示以“RUN\_AS\_”开头，常用在系统间临时通信，需要将凭证在网络中传递的情况下

**AfterInvocationManager**

在受保护方法成功执行并返回之后执行，可以用来修改返回的结果对象

同时也可以用来在方法调用之前无法验证权限的情况下

**本地化**

框架支持权限消息的本地化

**在ApplicationContext中注入bean**

**<bean id**=**"messageSource" class**=**"org.springframework.context.support.ReloadableResourceBundleMessageSource">**

**<property name**=**"basename" value**=**"classpath:org/springframework/security/messages"/>**

**</bean>**

本地化的支持需要验证方法调用之前在LocaleContextHolder中设置合适的值

springMVC会自动设置Local，但是是在Security之后，所以需要在web.xml中配置org.springframework.web.filter.RequestContextFilter

**预验证场景的处理(已被外部系统验证过)**

针对预处理场景，springsecurity提供了一系列的抽象类，来帮助快速实现

**AbstractPreAuthenticatedProcessingFilter**

该类实现了判断SecurityContext是否为空，如果为空则从请求中提取用户信息和凭证及其他明细，并创建PreAuthenticatedAuthenticationToken提交给AuthenticationManager验证，抽象方法为

protected abstract Object getPreAuthenticatedPrincipal(HttpServletRequest request);

protected abstract Object getPreAuthenticatedCredentials(HttpServletRequest request);

这里所谓的验证其实就是加载权限信息，也是为了以后的扩展

**AuthenticationDetailsSource**

根据请求对象创建验证信息明细信息，如sessionid、原始ip、权限（实现GrantedAuthoritiesContainer接口）

**PreAuthenticatedAuthenticationProvider实现了AuthenticationProvider**

作用仅仅负责创建UserDetails对象，把请求代理给AuthenticationUserDetailsService，AuthenticationUserDetailsService根据Authentication对象创建UserDetails

默认实现类PreAuthenticatedGrantedAuthoritiesUserDetailsService仅仅是从Authentication提取用户名和权限信息，然后创建User对象

可以实现AuthenticationUserDetailsService接口来自定义提取信息的逻辑

另外还可以通过UserDetailsByNameServiceWrapper将UserDetailsService适配给AuthenticationUserDetailsService

**Http403ForbiddenEntryPoint**

AuthenticationEntryPoint通常用来重定向到验证入口，但是对于预验证的情况不用重定向，只需将其配置到ExceptionTranslationFilter即可

**请求头验证（**Siteminder**）**

Siteminder将用户名放到请求头的SM\_USER上，RequestHeaderAuthenticationFilter实现了从请求头提取用户信息的逻辑

**JSP 标签库**

SpringSecurity拥有自己的标签库来提供在jsp中访问权限信息

在页面顶部引用标签库

**<%@ taglib prefix**=**"sec" uri**=**"http://www.springframework.org/security/tags" %>**

**authorize标签**

**<sec:authorize access**=**"hasRole('supervisor')">**

This content will only be visible to users who have the "supervisor" authority in their list

of **<tt>**GrantedAuthority**</tt>**s.

**</sec:authorize>**

**<sec:authorize access**=**"hasPermission(#domain,'read') or hasPermission(#domain,'write')">**

This content will only be visible to users who have read or write permission to the Object found as a

request attribute named "domain".

**</sec:authorize>**

**authentication标签 访问当前用户信息**

<sec:authentication property="principal.username" />

**csrfInput标签**

**与JAAS集成**

**与CAS集成**

**加密模块**

**BytesEncryptor**

**对并发的支持**

**与SpringMVC集成**

AntPathRequestMatcher

MvcRequestMatcher 启用mvc匹配模式

**@AuthenticationPrincipal**

当我们自己的CustomUser 对象实现了UserDetails接口时，我们可以直接在UserDetailsService中返回我们的CustomUser 对象，同时Authentication.getPrincipal()即为我们自定义的CustomUser 对象，获取当前登陆的CustomUser 对象就需要如下代码：

Authentication authentication = SecurityContextHolder.getContext().getAuthentication();

CustomUser custom = (CustomUser) authentication == null ? null : authentication.getPrincipal();

但是这样就耦合了Controller和springsecurity

自动解析当前的principle到mvc参数

javaConfig方式注入了AuthenticationPrincipalArgumentResolver，

Xml方式需配置如下

<mvc:annotation-driven>

<mvc:argument-resolvers>

<bean class="org.springframework.security.web.method.annotation.AuthenticationPrincipalArgumentResolver"/>

</mvc:argument-resolvers>

</mvc:annotation-driven>

在springMVC中就可以如下引用当前用户

@RequestMapping("/messages/inbox")

public ModelAndView findMessagesForUser(@AuthenticationPrincipal CustomUser customUser) {

}

这样就解耦了spring security和Controller

如果我们自定义的CustomUser实现了UserDetails接口，这样也存在我们的代码强耦合springsecurity，为了使我们的领域对象解耦springsecurity，我们可以自定义一个“适配器”

**public class** CustomUserUserDetails **extends** User {

**public** CustomUser getCustomUser() {

**return** customUser;

}

}

但此时我们使用@AuthenticationPrincipal注解获取的是适配器对象，并不是我们的CustomUser，此时我们可以指定该注解的expression，来调用适配器的getCustomUser方法

@RequestMapping("/messages/inbox")

**public** ModelAndView findMessagesForUser(@AuthenticationPrincipal(expression = "customUser") CustomUser customUser) {

}

为了消除重复，还可以自定义一个注解@CurrentUser，来代替@AuthenticationPrincipal(expression = "customUser")

@Target({ElementType.PARAMETER, ElementType.TYPE})

@Retention(RetentionPolicy.RUNTIME)

@Documented

@AuthenticationPrincipal(expression = "customUser")

**public** @interface CurrentUser {}

这样就完全与springsecurity框架解耦了

**CSRF**

# 与spring-data集成

spring-data提供了实体审计的功能，实体实现Auditable接口或者在字段上添加注解@CreatedBy、@LastModifiedBy、@CreatedDate 、@LastModifiedDate

就会在新增实体和修改实体时，自动设置当前用户和时间，此时需要我们提供一个实现了AuditorAware<T>接口的实现类来提供当前登录用户，T类型参数变量为审计人的类型，可以是我们自定义的用户实体类型，可能是实现如下：

class SpringSecurityAuditorAware implements AuditorAware<User> {

public User getCurrentAuditor() {

Authentication authentication = SecurityContextHolder.getContext().getAuthentication();

if (authentication == null || !authentication.isAuthenticated()) {

return null;

}

return ((MyUserDetails) authentication.getPrincipal()).getUser();

}

}

在@Query注解中使用表达式

配置xml文件

**<bean class**=**"org.springframework.security.data.repository.query.SecurityEvaluationContextExtension"/>**

或者基于javaConfig的配置

@Bean

**public** SecurityEvaluationContextExtension securityEvaluationContextExtension() {

**return new** SecurityEvaluationContextExtension();

}

然后就可以在Query的表达式中使用以SecurityExpressionRoot为根的安全表达式

@Repository

public interface MessageRepository extends PagingAndSortingRepository<Message,Long> {

@Query("select m from Message m where m.to.id = ?#{ principal?.id }")

Page<Message> findInbox(Pageable pageable);

}

**领域对象安全（ACL访问控制列表）**

**（这块内容看的有点懵逼）**

除了访问控制和方法调用级别，安全控制还需who (Authentication),where (MethodInvocation)，what (SomeDomainObject)组成，安全控制还需要考虑被调用方法的所属对象