# 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

Web支持

在web.xml中配置DelegatingFilterProxy，并将该filter的名称叫做springSecurityFilterChain

ApplicationContext内部的Bean，FilterChainProxy（根据是否使用了http来创建）实现了Filter接口，并代理到

SecurityFilterChain（http标签负责创建），默认的DefaultSecurityFilterChain聚合了Filter，并带有一个RequestMatcher，默认的实现是AntPathRequestMatcher，也可以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

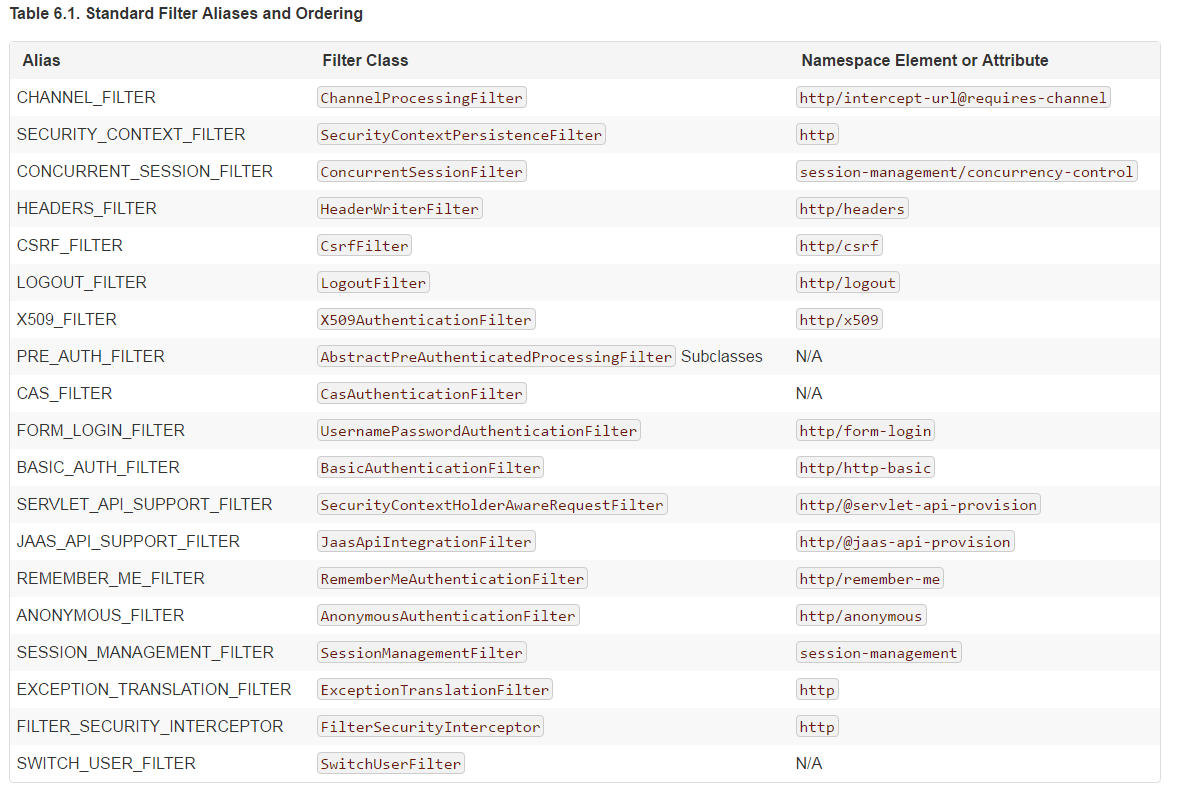
检测session过期

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）

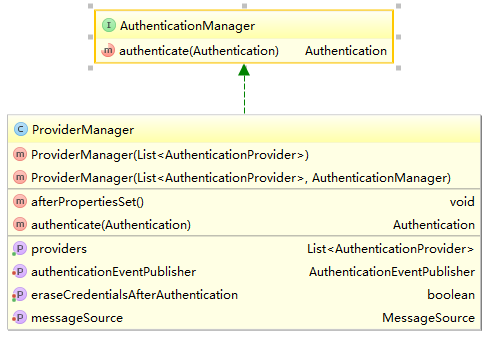
• HttpStatusReturningLogoutSuccessHandler（rest场景下使用返回状态码）

authentication验证服务

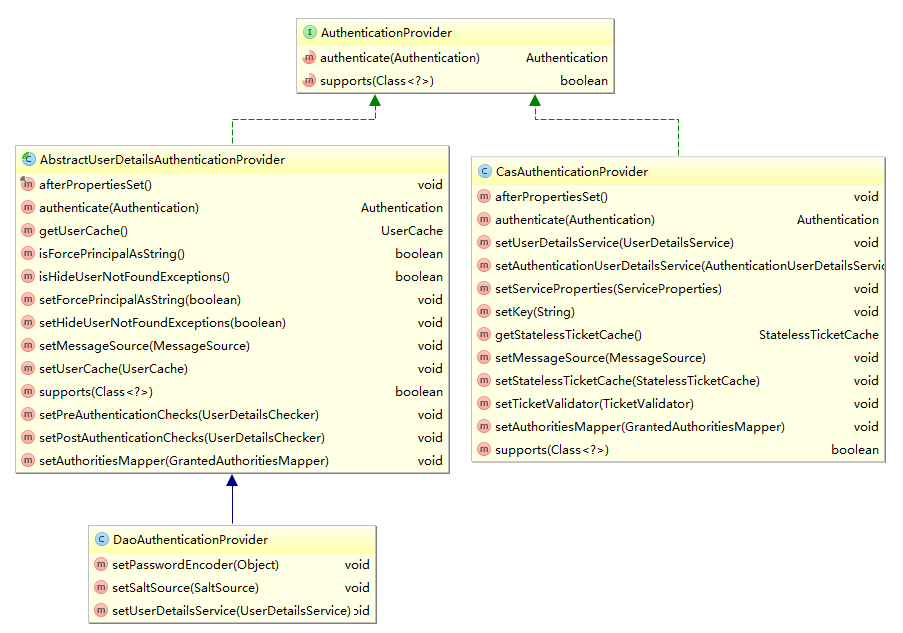
验证服务是通过接口AuthenticationManager，默认实现为ProviderManager

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

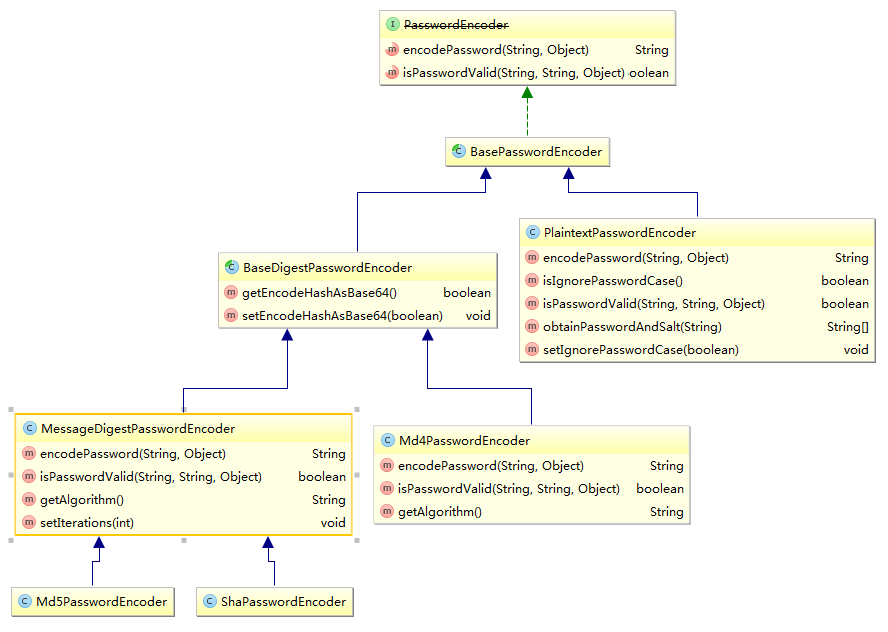


AuthenticationProvider的典型实现是DaoAuthenticationProvider，DaoAuthenticationProvider依赖PasswordEncoder来进行密码的验证

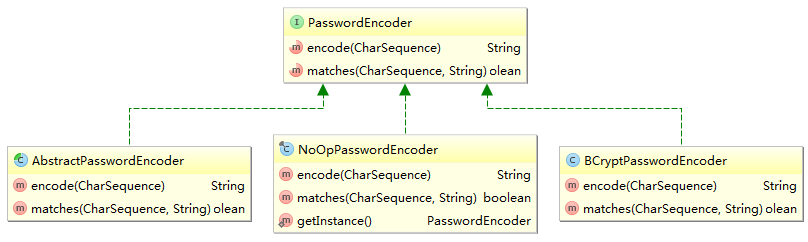


DaoAuthenticationProvider内部依赖旧版的PasswordEncoder来进行验证，却提供了setPasswordEncoder(Object passwordEncoder)，这样旧版和新版PasswordEncoder都能根据属性名称注入进来，setPasswordEncoder方法内部将新版转为旧版

旧版的PasswordEncoder，需要提供salt



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

**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**

基于filter的权限验证拦截器，继承自AbstractSecurityInterceptor并实现了filter接口

实现了http的访问权限的验证功能

**<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

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，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

**对CSRF的支持**

**利用CORS实现跨域安全认证**

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;

}

**Http响应头**

**//TODO**

**Session管理**

**Session管理的主要功能有**

**防止会话固定攻击**

**检测会话过期**

**限制验证用户只能有一个session同时在线**

SessionManagementFilter

SecurityContextRepository（持久化SecurityContext的仓库，默认实现为基于Session的HttpSessionSecurityContextRepository）

用户已经认证过的处理方式

SessionAuthenticationStrategy

**保存在线已认证的用户信息**

**SessionRegistry**

**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

GrantedAuthority的getAuthority()返回值为null，则暗示着AccessDecisionManager需要特殊处理此类型的GrantedAuthority

**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

如果反对票大于同一票，则不通过

如果同一票与反对票相等，则根据配置来判断是否通过

AccessDecisionVoter投票因素

默认实现由RoleVoter、AuthenticatedVoter

AfterInvocationManager 调用之后的处理器

RoleHierarchyVoter角色继承投票

RoleHierarchy角色继承访问器

表达式的支持

WebSecurity表达式的根类WebSecurityExpressionRoot

MethodSecurity表达式的根类MethodSecurityExpressionRoot

@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

@PreFilter可以对指定的参数为集合的元素进行过滤，表达式为false的将被剔除集合

“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

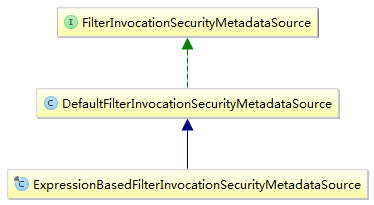
Spring框架内部用ConfigAttribute来表示，在框架配置上用注解或者access属性来表示

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)

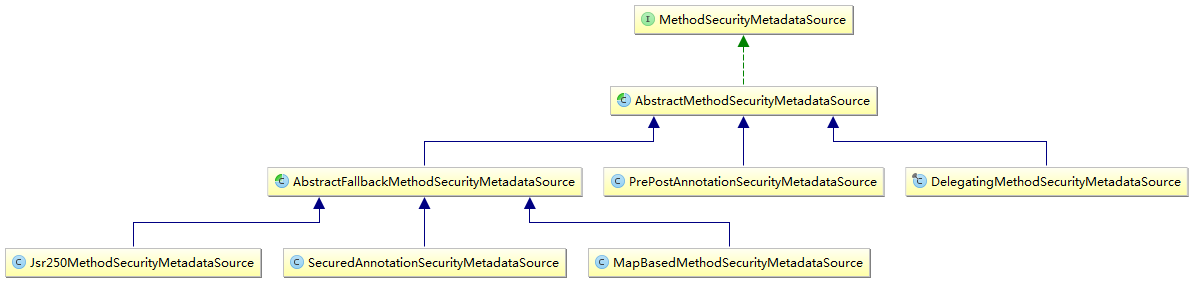
@Secured 指定配置属性

*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

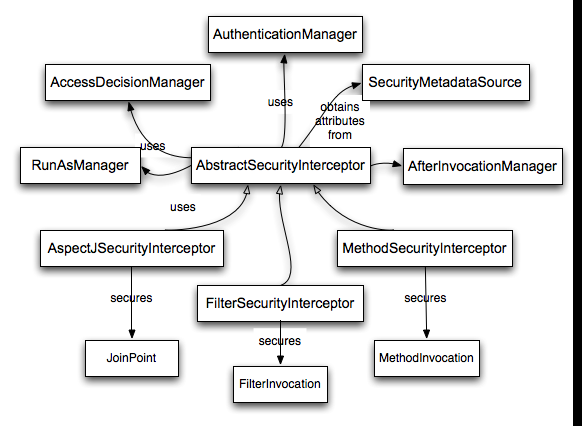
方法执行完，则恢复回原来的Authentication

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