# SpringSecurity 学习札记：

1. 核心组件
2. SecurityContextHolder：一个功能对象，存储着security的上下文（SecurityContex），主要是和用户身份相关的一些信息（principal）。默认是绑定在ThreadLocal，也就是说和身份相关的请求处理完以后线程被清理，这个holder也就没有了。

也可以更改这个绑定策略：

* SecurityContextHolder.MODE\_GLOBAL：所有线程公用同一个
* SecurityContextHolder.MODE\_THREADLOCAL:默认的。一个线程一个context
* SecurityContextHolder.MODE\_INHERITABLETHREADLOCAL：安全线程开启的子线中共享context

有两种更改策略的方式：a.设置系统参数属性；b.通过SecurityContextHolder的静态方法设置

1. SecurityContext：SS中的安全上下文，主要作用是设置/获取Authentication对象
2. Authentication：一个接口。存储着用户身份相关信息

* getAuthorities()-获取用户拥有的权限/角色，一般由 AuthenticationManager设置进去
* getPrincipal()-获取身份验证的表示，一般是principal 代表了用户是谁
* getCredentials()-获取安全凭证。一般是密码
* getDetails()-一些额外的有关用户的信息，如IP地址，证书编号
* isAuthenticated/setAuthenticated 这个用户是否验证过。

1. userDetails: 用户信息的接口。一般principal都可以转换成一个UerDetails。

他代表了principal的概念，同时根据不同应用做出了扩展。

他是实际应用中的用户类和SS中需要用户类的一个适配器。我们一般要将系统里的用户类实现这个接口。一般可以通过SecurityContextHolder获得userDetails，然后转型成应用中的实际用户对象(可以是employee,customer,user等等….)。

1. UserDetailsService：接口，用于获取userDetails

* UserDetails loadUserByUsername(String username) throws UsernameNotFoundException;

SS验证时调用这个方法读出系统的用户，和用户输入的用户信息做比较。如果比较匹配，就用userDeatils构建Authentication对象，并保存到SecurityContext。JdbcDaoImpl、InMemoryDaoImpl都是UserDetailsService的实现类

UserDetailsService他一般是在DAO层上，用来获取实际应用中的“用户”。他只用来提供信息，不进行实践的验证操作。验证由AuthenticationManager完成（应用中定制AuthenticationProvider注册给AuthenticationManager）

\*总结：userDetilas就是我们系统里的entity中的一部分，含有ss需要用的信息。一般让entity继承他。UserDetailsService就是我们系统里的DAO，让对应的entityDao继承他即可。

1. GrantedAuthority：用户拥有的一组角色/权限。他一般代表着应用级别粗粒的的角色，不要把domain级别的权限设置给他。

Object principal = SecurityContextHolder.getContext().getAuthentication().getPrincipal();

**if** (principal **instanceof** UserDetails) {

String username = ((UserDetails)principal).getUsername();

} **else** {

String username = principal.toString();

}

1. 身份验证
2. **整体流程**：（假设用户登录成功）
3. 用户通过名称/密码登录系统--系统将username和password组装成一个UsernamePasswordAuthenticationToken（他是Authentication接口的实例）。
4. 系统验证身份 -- 将上面的token实例交给AuthenticationManager验证。
5. 获取用户信息 – AuthenticationManager根据token中的信息查找系统里是否有该用户，如果验证成功（找到了给用户密码也对了），就用读出来的用户身份重新构建一个上一步中的Authentication对象，并返回。
6. 根据用户信息，建立安全上下文 – 将AuthenticationManager返回的对象 设置到SecurityContextHolder中

SecurityContextHolder.getContext().setAuthentication(…​)

1. ExceptionTranslationFilter：SS的Filter。SS的异常分发中心.他负责分发异常，不负责具体异常处理。

* 如果是AuthenticationException，他启动一个AuthenticationEntryPoint处理
* 如果是AccessDeniedException，他先判断用户有没有经过身份验证。如果是匿名用户，则启动一个AuthenticationEntryPoint。如果确实没有权限则委托AccessDeniedHandler进行下一步处理。
* 有一个属性requestCache：负责记录用户当前访问的资源，如果用户验明身份，可以通过他跳转回刚刚想访问的资源。默认实现是HttpSessionRequestCache

1. AuthenticationEntryPoint：SS认证登录点。

当用户想访问一个资源，但是还没登录，SS会抛出异常，这个异常被ExceptionTranslationFilter截获有分发给entryPoint类，他负责转发请求到登录页，供用户进行身份验证。

1. AuthenticationManager：SS身份验证的核心接口。

管理验证是否接受的核心接口 只有一个方法

Authentication authenticate(Authentication authentication) throws AuthenticationException

如果验证成功，返回完全的Authentication对象

如果失败，抛出具体的异常

1. ProviderManager：AuthenticationManager的默认实现类

* 其内部是有一组AuthenticationProvider 集合，首先他遍历这些集合，轮训每一个provider是否支持对现有用户信息的验证。如果支持，就用这个具体的authticationProvider校验身份。（只是用第一个支持的provider，之后的就不轮训了）
* 验证后，如果抛异常ProviderManager负责截获，如果返回了一个result（含有全信息的Authentication对象）。则用这个result构建全信息的authentication对象。
* 他有一个boolean属性：eraseCredentialsAfterAuthentication.即验证成功会是否擦除登录时用户输入的凭证信息。以免后续泄露造成安全问题。具体的逻辑是存放这些信息的实体类，如user类，xxxtoken类。一般会扩展CredentialsContainer接口，表明这个类含凭证信息。然后实现这个接口的eraseCredentials，置空其中的关键信息，如密码。

需要注意的是，如果你把这样一个user缓存起来，以便后续再验证时候使用，当·eraseCredentialsAfterAuthentication为true是，第一次验证成功了就把密码信息擦除了，以后就不能再用它来进行身份验证了。有个变通的方法是，可以在验证前copy一份这个对象，这样就不会清cache里的了。也可以直接把这个属性设为false

1. AuthenticationProvider 标准接口，负责具体验证。

* provider.supports(Authentication.getClass());--每一个AuthenticationProvider实现类 都有support方法，表名自己支持什么样的Authentication对象验证
* result = provider.authenticate(authentication)；每一个AuthenticationProvider实现类 都有authenticate方法用于验证

有很多他的实现类如DaoAuthenticationProvider，LdapAuthenticationProvider， CasAuthenticationProvider。他们有的可以协同工作，有的需要自己的逻辑。但是都是按照统一的流程进行。

1. DaoAuthenticationProvider他是一个重要的AuthenticationProvider 实现类

一般的应用（把用户信息存在数据库里，通过输入用户名、密码进行登录的）。都是用这个provider进行身份验证的。

他内部使用UserDetailsService当做数据访问对象（DAO）来查询实际应用中的username、password、用户拥有的权限等信息。

内部的验证逻辑就是把用户输入的信息UsernamePasswordAuthenticationToken和他用useDetailsService查到的信息做判断。

有两个属性：

1. userDetailsServer：作为他内部的一个dao。如inMemoryImpl，JdbcImpl。一般这个需要我们自己实现，以符合具体的应用内部逻辑
2. PasswordEncoder：密码编码器。用来把明文密码加密的。  
   关于Password Encoding。SS说了一大堆，就是推荐这个类：

org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder

\*关于密码补充一些基本常识：

* 首先是encode（编码）：把明文密码变成我们看不懂的一个串 一般由两种格式：Base64和HEX。
* 其次是Hash（散列算法）：他是把明文密码变成一种加密的长度相等的串。一般有MD5，SHA126,SHA256,SHA512,Bcrype.

Hash是不可逆的，而encode是可以互相转换的。

Hash虽说不可逆，但是可以通过穷举试出原始值。所以就有了salt的感念，就是在密码中加入一些干扰字符。Password = password +salt 然后对新password做hash。系统里存储好这个盐。这样每一次都是用户输入password，系统给出盐，拼合后做指定次数的hash操作，然后和系统里的已经散列化密码对比。从而增加破解难度。

存密码一般是先做对密码做几次hash 然后再编码:MD5(MD5(“pasword”)).HEX()

* 加密、解密：如对称式加密/解密算法 AES Blowfish.这个一般就不涉及了
* Bcrypt好像能随机生成salt，并攒在passowrd串里，这样就不用我们在数据库里维护盐这个字段了。反正现在这个是默认passwordEncoder用他这个就行了。

1. 访问控制
2. SecurityContextPersistenceFilter SS的filter：

想象一个常见，一个网站，用户应只需登录一次，就能访问所有他能看到的资源，而不是每一次请求都需要重新登录，这就需要网站能够在多次request之间存储用户信息。

在SS中SecurityContextPersistenceFilter负责缓存这些principal信息。他默认是把SecurityContext 的内容放到 HTTPSession中，在多次请求之间将SecurityContext还原到SecurityContextHolder，请求结束后清除SecurityContextHolder的内容。

* 有的应用把并发请求放到一个session中，这样的话他们将共享同一个SecurityContext，尽管SS用了ThreadLocal，但是SecurityContextPersistenceFilter从session中获取的是同一个实例对象，这样的话如果你在一个线程中临时性的修改了context中的authtication对象，那么这种修改会波及到他的线程。针对这种使用情况：可定制SecurityContextPersistenceFilter的行为，让他每一个请求都new一个SecurityContext。或者你可以在你想临时改动context内容的时候调用SecurityContextHolder.createEmptyContext()，返回一个新实例，从而不影响其他的请求

1. **AccessDecisionManager**：SS中访问控制的核心接口

* decide() 方法，决定是否有权方法。如果没有，则抛出异常AccessDeniedException.

需要三个参数：

1. Authentication authenticaitoon：有用户登录填入的身份信息构建的对象
2. Object secureObject：用户要访问的资源
3. Collection<ConfigAttribute> configAttributes caList：该资源需要的角色/权限信息

SS中可能同时存在多个AccessDecisionManager的实现类，AbstractSecurityInterceptor会根据不同的访问资源、权限信息选择使用哪一个AccessDecisionManager的实现类进行判断

* boolean supports(ConfigAttribute attribute)

根据资源需要的权限判断是否使用这个AccessDecisionManager

* boolean supports(Class<?> clazz)

根据被访问的资源类型决定否使用这个AccessDecisionManager

1. **AbstractSecurityInterceptor：**

在具体的应用中受保护的资源（secure object）可以是web request也可以是业务方法

SS提供Filter来控制web request安全，通过AOP来控制业务层方法访问的安全（可以使用Spring aop或者AspectJ）.不同类型的受保护资源都有其不同的interceptor类用来实现访问控制。他们共同了一个抽象类AbstractSecurityInterceptor。他提供了一个访问控制统一流程：

1. 查询资源所需的权限（"configuration attributes"）
2. 将资源、所需权限、和用户信息交给**AccessDecisionManager**来决定是否能访问
3. （可选）变更用户信息（Authentication） -RunAs的情况
4. 允许资源访问（如果有权限），就调用目标资源，执行目标方法。
5. 如果配置了AfterInvocationManager ，当方法调用正常返回时调用它，如果调用出现异常，则不执行他。
6. **ConfigAttribute**  访问受保护资源的所需的权限信息。可以理解成一个参数接口。

SS中一般是一个含有特殊意义的字符。但实际应用中他可以是任何东西。他的作用就是传递到AccessDecisionManager做判断。所以他是什么要看AcceessDecisionManager的具体实现类的判断逻辑需要什么样参数。

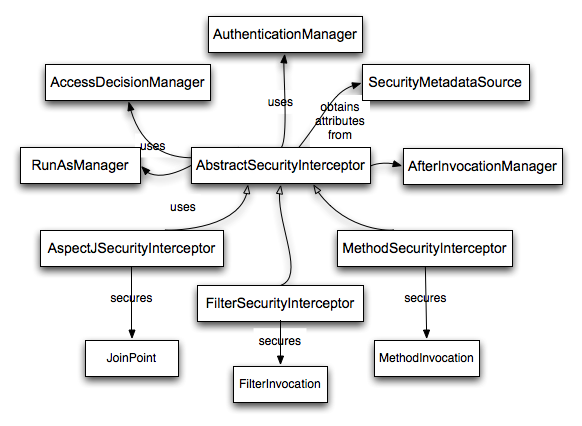
1. **SecurityMetadataSource**可以理解为存放**ConfigAttribute** 的仓库。**AbstractSecurityInterceptor**用他来查找具体资源所需要的权限。
2. **RunAsManager ：**

极少数情况，如果用户调用了一个方法，验证也成功了，但他想以另一个身份去执行这个方法，这个时候需要**AbstractSecurityInterceptor**临时将SecurityContex中Authentication的替换为另一个身份（**RunAsManager**提供的另一个Authentication），待访问执行完成后再变会原有的Authentication

1. **AfterInvocationManager：**

当一个方法执行完，程序又返回到过滤器链中的**AbstractSecurityInterceptor，**他会执行**afterInvocation**，即复原**SecurityContext**（如有必要）**，**然后执行**AfterInvocationManager**（如果配置了）。在**AfterInvocationManager**中，我们可以修改方法返回的对象（个人认为是可以做一些过滤等等）。

**AbstractSecurityInterceptor相关联的对象：**



1. Web 应用安全

SS内部有一组自己Filter，实现web访问的安全。具体介绍一些filter和用传统Spring bean的方式配置，以达到SS命名空间做不到的细粒度的掌控web安全

1. **过滤器链**

SS使用是标准的web servlet filter机制，也就是说他没有使用servlets以及基于servlets的框架，如SpringMVC。他直接处理HttpServletRequest和HpptServletResponse。

SS内部的过滤器有特定的顺序和配置，如果用SS命名空间，他隐藏了这些内部实现细节。但是也因此不能做灵活的配置。可以用Spring标准的bean形式，配置SS的内部实现，达到更彻底的程序控制

1. **DelegatingFilterProxy**

他是一个Spring的类，配置在web.xml中，作用是web filter执行代理给spring内部配置的bean。这样做，可以把具体的Filter写成Spring的bean。从而能享受到Spring提供的依赖注入、生命周期管理等好处

在web.xml中例子：

<filter>

<filter-name>myFilter</filter-name>

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

</filter>

<filter-mapping>

<filter-name>myFilter</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

Spring的bean必须和<filter-name>里配置的相同，Spring会把filter控制流代理给这个具体的bean。

1. **FilterChainProxy**

SS的Filter流程控制总入口点。也是定义过滤器链的地方。

所有SS的Filter必需交由他代理，不能单独运行。配置成Spring中的bean。名字要和web.xml中相同。

<bean id="filterChainProxy" class="org.springframework.security.web.FilterChainProxy">

<constructor-arg>

<list>

<sec:filter-chain pattern="/restful/\*\*" filters="

securityContextPersistenceFilterWithASCFalse,

basicAuthenticationFilter,

exceptionTranslationFilter,

filterSecurityInterceptor" />

<sec:filter-chain pattern="/\*\*" filters="

securityContextPersistenceFilterWithASCTrue,

formLoginFilter,

exceptionTranslationFilter,

filterSecurityInterceptor" />

</list>

</constructor-arg>

</bean>

他把一组filter按名字组装，交给一个实现了SecurityFilterChain的过滤器类。

* <filter-chain>元素：SS的xml命名空间里有filter-chain，用来方便维护这些信息.里面的pattern是这一组过滤器链适用的url。按照Ant风格或者正则表达式风格。匹配的就交给这个链执行。第一次匹配后就不继续找了。所以要把最详细pattern过滤器链先到最前面。filters指的是这个过滤器链中的过滤器，名字就是定义具体过滤器的bean名字。过滤器执行顺序就列出的循序。Filters=”none”，就意味着制定的url不会走过滤器链。也就不能享受到SS提供的验证授权这些服务。一般的资源类的url会这么写。如：res/，images/，css/等等。

过滤器功能和顺序：

1. ChannelProcessingFilter：

判断请求是https还是http。内部通过HttpServletRequest.isSecure()判断一个请求是不是安全频道。

把他放到最前是因为，他可能会redirect请求到其他协议。如访问一个资源需要https

用户访问的是<http://xxx>.html资源，他可能会重定向到https://xxx.html

1. SecurityContextPersistenceFilter,

在用户登录进行一连串的请求时，他负责保管SecurityContext内容

在这个位置存放可以保证在请求开始的时建立SecurityContext并放入SecurityContextHolder中。当请求结束时对SecurityContext进行的任何修改都会同步到session中。以供用户进一步的web reqeust

1. ConcurrentSessionFilter,

用于同步session监管的filter。如果同一个账号在多处同时上线。他可以配置一些策略，T掉一个或者是怎么样。

放到这个位置因为他需要使用SecurityContextHolder的功能并且更新SessionRegistry以反映出同一个principal不间断的请求。

1. 执行身份验证的Filter。如：UsernamePasswordAuthenticationFilter, CasAuthenticationFilter, BasicAuthenticationFilter 等等

一般这些filter会修改SecurityContextHolder中的信息。把一个用户输入的Authentication（token）对象变成一个合法的还有详细信息的Authentication（token）对象

1. SecurityContextHolderAwareRequestFilter,

用实现了ServletAPI的安全方法的reqeust wrapper 包装原始request。我估计就是在身份验证后把用户验证状态，和SS内部的一切方法包装成ServletAPI中的方法，做到和ServletAPI的集成。

1. JaasApiIntegrationFilter,

如果SecurityContextHolder中存在JaasAuthenticationToken，这个filter会使用JaasAuthenticationToken中的subject继续执行过滤器链

1. RememberMeAuthenticationFilter,

用于remeberMeF服务的Filter

如果之前的认证并没有更新SecurityContextHolder中的内容并且请求中有可供rememberMe服务使用的cookie。那么就会在SecurityContextHolder将对应的放入一个remembered Authentication对象

1. AnonymousAuthenticationFilter,

匿名登录的filter：

如果之前的认证都没有更新SecurityContextHolder中的内容（都没有成功） 匿名登录机制会更新SecurityContextHolder，放入一个匿名Authentication 对象

1. ExceptionTranslationFilter,

用来捕获SS的异常，保证之前filter抛出的异常能正确的返回或者根据异常开启一个合适的AuthenticationEntryPoint

1. FilterSecurityInterceptor,

用来保证url的访问控制，如果无权访问抛出异常

\*注意：FilterChainProxy并不会调用具体Filter里的标准生命周期方法。意思是servlet filter里一般会定义init(),doFilter(),destroy()。由servlet负责调用。但是SS的FilterChainProxy不一定会调用这些。如果你自己定制的Filter，想要让程序必须在执行时调用它的某个方法，可以考虑用Spring的生命周期支持，比方继承initLazingBean接口，写一个afterPropertiesSet。或者用@PostConstruct。

1. **HTTP请求匹配和HttpFirewall策略**SS中有几处请求匹配，即用配置的模式串和进入的请求串做匹配。如FilterChainProxy应这个机制匹配请求属于哪个过滤器练。FilterSecurityInterceptor中用他匹配请求对应的访问权限。

在Servlet规范中 HttpServletRequest 中有contextPath, servletPath, pathInfo 和queryString

这些属性都和url相关。SS值application中与安全相关的path 所以可以不管contextPath。但是servlet规范没有定义servletPath，pathInfo应该是设么值，而不同的servlet实现容器对他们的处理也不一样，用户甚至可以写一些而已的uri地址，去访问某一受保护资源。

SS中FilterChainProxy使用了HttpFirewall策略来检测和包装reqeust。一些不合理的url直接拒绝访问，将url中的多余的斜线“/”和参数部分除去方便匹配。正因为这样，所以用FilterChainProxy管理过滤器链至关重要。

因为servletPath, pathInfo具体是什么值是有servlet容器负责解析的，所以我们应用中定义的url串应该不含有分号，因为在匹配前会去掉这些部分。

关于请求匹配，SS默认使用的是Ant风格的匹配。默认的实现类AntPathRequestMatcher。此类内部使用Spring的AntPathMatcher，对url中的servletPath、pathInfo组成的串进行忽略大小写匹配。忽略了url中的queryString部分。

也可以用正则表达式匹配，他默认的实现类是RegexRequestMatcher

1. **FilterSecurityInterceptor**

他的基本概念前面访问控制已经讲了，他就等于是命名空间中的<intercept-url>元素。这里主要是介绍不适用命名空间时，如果以bean的形式配置和使用。

先看一个例子：

<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>

他内部需要authenticationManager，accessDecisionManager和securityMetadataSource。

securityMetadataSource可以用命名空间的方法配，如果要定制securityMetaDataSource，可以把它单独看成一个bean，然后这里ref=”xxx”.

securityMetaDataSource目的是获取针对某一个url所需要的权限集合

这里其实需要的是FilterInvocationSecurityMetadataSource的实现类。他是一个标记接口 extends AbstractSecurityMetadataSource。里面没有新东西，就是表示用于filterInvocation的SecurityMetadataSource。

1. **ExceptionTranslationFilter**

过滤器连中它位于FilterSecurityInterceptor之前。用来捕获由FilterSecurityInterceptor抛出的异常，上例子：

<bean id="exceptionTranslationFilter"

class="org.springframework.security.web.access.ExceptionTranslationFilter">

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

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

</bean>

<bean id="authenticationEntryPoint"

class="org.springframework.security.web.authentication.LoginUrlAuthenticationEntryPoint">

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

</bean>

<bean id="accessDeniedHandler"

class="org.springframework.security.web.access.AccessDeniedHandlerImpl">

<property name="errorPage" value="/accessDenied.htm"/>

</bean>

他需要authenticationEntryPoint和accessDeniedHandler。

具体逻辑是：security interceptor会抛出了AuthenticationException 或者 AccessDeniedException。

如果是AuthenticationException，则filter交给AuthenticationEntryPoint，负责给用户提供登录入口点。

如果是AccessDeniedException，再看是不是匿名用户 （是否经过身份验证）

（可通过authenticationTrustResolver.isAnonymous(Athentication)判断）

如果是，交给AuthenticationEntryPoint重新验证任务

如果不是，说明他确实没有权限访问改资源，交给accessDeniedHandler

authenticationEntryPoint负责对尝试访问资源但还没有进行身份验证的用户提供验证点。里面有一个commence()方法，提供合适的响应让用户验明正身。这里我们用的是LoginUrlAuthenticationEntryPoint。他直接重定向到某个url，一般是登录页。针对不同的登录机制，可选择不同的实现类

accessDeniedHandler 负责处理用户无权访问的情况。默认的现实是AccessDeniedHandlerImpl，他返回一个403 response给客户端。可以定制，就像上面的例子，加上一个返回错误页。错误页url可以是执行一个网页也可以指向一个controller，用来做更多处理。

\*ExceptionTranslationFilter的另一个作用缓存请求：在需要调用authenticationEntryPoint的地方，先缓存用户的请求，这个功能由RequestCache实现

试想一个用户想访问某一资源，但还没登录，应该让他登陆后，跳转到之前想访问的内容上。

这个功能由SavedRequestAwareAuthenticationSuccessHandler 类处理，内部也用RequestCache具体负责存储请求。

RequestCache接口--负责缓存存储和获取HttpServletRequest请求对象。默认实现是HttpSessionRequestCache，他将请求缓存到session中

RequestCacheFilter 实际负责当用户重定向到原来的url是恢复请求。

1. **SecurityContextPersistenceFilter**

具体配置：

<bean id="securityContextPersistenceFilter"

class="org.springframework.security.web.context.SecurityContextPersistenceFilter">

<property name='securityContextRepository'>

<bean class='org.springframework.security.web.context.HttpSessionSecurityContextRepository'>

<property name='allowSessionCreation' value='false' />

</bean>

</property>

</bean>

作用已经说过了：

在reqeust之间存储SecurityContext中的内容，在reqeust完成后，清空SecurityCoontextHolder。

清理存放SecurityContext的ThreadLocal很有必要。一个进程用完了回到容器的进程池中，如果下次又被使用，里面还附带着上一次的SecurityContext。这个进程再被执行的时候，里面存放的就是错误的凭证信息了。

SS 3.0后 载入恢复SecurityContext由SecurityContextRepository接口负责，默认实现是HttpSessionSecurityContextRepository。存放到HttpSession中。他有一个属性allowSessionCreation。是否允许HttpSessionSecurityContextRepository为经过验证的用户新建一个session存放security context（验证已经进行过或者security context改变才会新建）

还有一个实现NullSecurityContextRepository。即使有session也不将context存放进去，参见http://en.wikipedia.org/wiki/Null\_Object\_pattern[null object]

1. **UsernamePasswordAuthenticationFilter**

具体负责登录验证的filter：配置如下

<bean id="authenticationFilter" class=

"org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter">

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

</bean>

UsernamePasswordAuthenticationFilter继承自AbstractAuthenticationProcessingFilter 里面有AuthenticationSuccessHandler 和AuthenticationFailureHandler

这俩handler负责验证完成后针对成功和失败要进行的处理。默认的实现分别是SavedRequestAwareAuthenticationSuccessHandler，SimpleUrlAuthenticationFailureHandler。

另外 SS提供的有：  
SimpleUrlAuthenticationSuccessHandler, SavedRequestAwareAuthenticationSuccessHandler, SimpleUrlAuthenticationFailureHandler 和ExceptionMappingAuthenticationFailureHandler。

1. **BasicAuthenticationFilter**
2. **DigestAuthenticationFilter**
3. RememberMe

“记住我”服务。

底层通常是，服务器在验证成功以后会发一个cookie给客户端，用户再次访问的时候，后台检测客户端的特定cookie，然后内部执行自动登录。

SS针对rememberMe提供了两个具体实现，一个是hash的散列化token，他将用户名密码和过期时限用hash加密，然后存在客户端。一个是用持久化技术存储token。具体原理参考[http://jaspan.com/improved\_persistent\_login\_cookie\_best\_practice SS](http://jaspan.com/improved_persistent_login_cookie_best_practice%20SS)中要用这种token需要在数据库里建一张表

create table persistent\_logins (username varchar(64) not null,

series varchar(64) primary key,

token varchar(64) not null,

last\_used timestamp not null)

在SS中想使用rememberMe必须要有实现了UserDetailsService的bean。

RememberMe可以和UsernamePasswordAuthenticationFilter BasicAuthenticationFilter协同使用，具体方式是，在这俩Fillter的共同父类AbstractAuthenticationProcessingFilter 中有一个hook，他会在合适时机调用。

* RemeberMeServices接口。

**public** **interface** RememberMeServices {  
 Authentication autoLogin(HttpServletRequest request, HttpServletResponse response);

void loginFail(HttpServletRequest request, HttpServletResponse response);

void loginSuccess(HttpServletRequest request, HttpServletResponse response,

Authentication successfulAuthentication);  
}

在AbstractAuthenticationProcessingFilter 会调用它的loginFail和loginSuccess方法，而在RememberMeAuthenticationFilter中会执行autoLogin()方法，当这个filter发现SecurityContextHolder中没有Authentication的时候执行。下面具体看他两个实现类。

* TokenBasedRememberMeServices

<bean id="rememberMeFilter" class=

"org.springframework.security.web.authentication.rememberme.RememberMeAuthenticationFilter">

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

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

</bean>

<bean id="rememberMeServices" class=

"org.springframework.security.web.authentication.rememberme.TokenBasedRememberMeServices">

<property name="userDetailsService" ref="myUserDetailsService"/>

<property name="key" value="springRocks"/>

</bean>

<bean id="rememberMeAuthenticationProvider" class=

"org.springframework.security.authentication.RememberMeAuthenticationProvider">

<property name="key" value="springRocks"/>

</bean>

TokenBasedRememberMeService具体生成RemeberMeAuthenticationToken.它使用userDetailsService获得用户名和密码，用来比较一个用户生成对应含有用户权限的token。如果有需要可以提供一个logout服务，用来将生成的token作废。所以在TokenBasedRemerberMeServices里也提供了SS的LogoutHandler接口，他的作用是配合LogoutFilter使用，自动切除cookie。

RememberMeAuthenticationProvider负责处理生成的token。这两个类必须持有相同的Token key值，可看配置。

\*使用中，还要配置UssernamePasswordAuthenticationFilter的RememberMeservice为对应的shixian类bean，在AuthenticationManager的providers列表中加入RememberMeAuthenticationProvider。最后在FilterChainProxy中加入RememberMeAuthenticationFilter（这个filter要放到UPAFilter之后）

* PersistentTokenBasedRememberMeServices

他的配置和TokenBasedRememberMeServices基本相同，还有一个额外的属性 PersisitentTokenRepository，用来存储token。SS中提供了两个实现类：InMemoryTokenRepositoryImpl—测试类用的。JdbcTokenRepository—存储在数据库用。

1. Session管理

HttpSession相关的功能是由SessionManagementFilter和SessionAuthenticationStrategy这两个接口完成的。

一般划分为 session固化攻击的保护，监测session过期，限制用户多出同时登录的个数。

1. **SessionManagementFilter**

这个Filter检查SecurityContextRepository中的SecurityContextHolder，判断用户是否已经登录过了，基本上供不需要交互的authentication机制使用的（如pre-authentication和rememberMe）。

内部逻辑是：如果SecurityContextRository中含有相应的security context，这个Filter直接过；如果没有，并且本地线程中的SecurityContext中有Authentication对象（不包括匿名情况），他会调用内部配置的SessionAuthenticationStrategy。

如果用户目前没有登录，他会检测request中是否含有一个不合法的sessionID（有的情况比方说过期，或导致session不合法），然后调用InvliadSessionStrategy.一般如果出现了这种情况直接调转到一个固定的URL，SS提供了SimpleRedirectInvalidSeessionStrategy用来做这个事。

1. **SessionAuthenticationStrategy**

\*这个接口在两个地方使用：1.SessionManagementFilter 2.AbstractAuthenticationProcessingFilter(UPAFilter的父类)。记得要在这两处配置上

\* 如果使用SessionFixationProtectionStragegy会引发问题，如果你同时用了实现了Spring的HttpSessionBindingListener的bean。

1. **Session并发管理**

管理同一个用户同时在不同的位置登录。（指不同的session，同一个机器不同的浏览器也是不同的session），可以配置具体允许同时登陆的数量。

如果出现这种情况，你可以T出之前登录的人，或者阻止后登录的人，弹出一些错误信息。

注意如果使用后一种策略，之前的人要是没有通过正规渠道登录，比方说仅仅是把浏览器关了，这样程序中还认为他是登录的（除非等到session过期了），那么后面的人再尝试用这个相同账号登录就会被阻止了。

SS命名空间支持并发管理。<session-management>这个节点 相当于sessionManagementFilter

SS中这个功能的具体实现类是ConcurrentSessionControlAuthenticationStrategy,他是SessionAuthenticationStrategy接口的实现类。SS3以后的逻辑和之前略有不同，在这里就说一下SS3之后 这个类的实现逻辑：

用户第一次登陆通过AuthenticationManager时，如果成功登录， session就会创建，并且检测是否允许创建再允许其他session。

具体配置：

1. 首先在web.xml中加上：

<listener>

<listener-class>

org.springframework.security.web.session.HttpSessionEventPublisher

</listener-class>

</listener>

在web.xml中配置的HttpSessionEventPublisher会将一个ApplicationEnvent事件发布到ApplicationContext中，他监听HttpSession的开始和终结

这个配置至关重要，因为下面的配置中SessionRegistryImpl会通过这个事件，得到session结束的通知，如果没有他，当某个用户超过了设置的最多允许同步登陆的session数量的时候，他就再也不能登录了，即使另一处也用这个用户名登录的人退出或者超时。

1. 在ss配置中加上ConcurrentSessionFilter，他需要两个属性，1.sessionResigtry一般直接是SessionResigtryImpl，2.expireUrl session过期后跳转到那个页面。

贴配置，用的命名空间的过滤器链，也可以轻松改成完全配置的情况

<http>

<custom-filter position="CONCURRENT\_SESSION\_FILTER" ref="concurrencyFilter" />

<custom-filter position="FORM\_LOGIN\_FILTER" ref="myAuthFilter" />

<session-management session-authentication-strategy-ref="sas"/>

</http>

<beans:bean id="concurrencyFilter"

class="org.springframework.security.web.session.ConcurrentSessionFilter">

<beans:property name="sessionRegistry" ref="sessionRegistry" />

<beans:property name="expiredUrl" value="/session-expired.htm" />

</beans:bean>

<beans:bean id="myAuthFilter" class=

"org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter">

<beans:property name="sessionAuthenticationStrategy" ref="sas" />

<beans:property name="authenticationManager" ref="authenticationManager" />

</beans:bean>

<beans:bean id="sas" class="org.springframework.security.web.authentication.session.CompositeSessionAuthenticationStrategy">

<beans:constructor-arg>

<beans:list>

<beans:bean class="org.springframework.security.web.authentication.session.ConcurrentSessionControlAuthenticationStrategy">

<beans:constructor-arg ref="sessionRegistry"/>

<beans:property name="maximumSessions" value="1" />

<beans:property name="exceptionIfMaximumExceeded" value="true" />

</beans:bean>

<beans:bean class="org.springframework.security.web.authentication.session.SessionFixationProtectionStrategy">

</beans:bean>

<beans:bean class="org.springframework.security.web.authentication.session.RegisterSessionAuthenticationStrategy">

<beans:constructor-arg ref="sessionRegistry"/>

</beans:bean>

</beans:list>

</beans:constructor-arg>

</beans:bean>

<beans:bean id="sessionRegistry"

class="org.springframework.security.core.session.SessionRegistryImpl" />

1. **SessionRegistry**：

他负责管理一个应用中的在线session。

如果你配置了session同步管理，他会带来一个好处就是，你可以持有sessionRegistry的引用，程序中其他的bean可以通过这个sessionRegistry管理同步在线的用户。

如果使用命名空间，他内部创建的sessionRegistry，可以通过设置session-registry-alias给出一个别名，从而将引用暴露到ApplicationContext中。介绍他的几个方法

* getAllPrincipals() – 获取当前登录的用户
* getAllSessions(Object principal, boolean includeExpiredSessions)

指定一个用户的principal，获取这个账户所有登录的session。

他会返回一个SessionInformation对象列表。

SessionInformation含有客户端登录的具体信息，如最后一次访问的时间，是否过期（是否上线、下线？）。

在SessionInfomation.expireNow()可以使某个具体的用户session过期，从而得到T人的目的

1. 匿名登录

实践中，最好给应用定义“deny-by-default”的安全策略：即默认所有资源都需要登录，然后在定义那些可以不需要登录即可看到，那些需要登录。不需登录就能看的情况就是匿名登录。对于一个系统，部分资源需要访问授权，但是至少有个别的页面是不需要权限的，比方说登录页，登出页，主页这些。所以给这些页对应的URL设置匿名访问很有必要，而且在基于角色管理的系统里，可以给匿名的用户赋予一定的觉得，比方说ROLE\_Anonymous。当然也可以选择压根不走SS的过滤器，通过把这样的地址排除在过滤器链之外，可以绕开整个SS的验证体系。

具体的功能性页面最好还是放到过滤器链里面，但是给一个匿名权限。而那些资源性的url，比方说图标，css，js这些，可以排除在过滤器链之外。

如果使用了SS命名空间的<http>标签，默认自动加上了匿名登录的功能。具体的定制可以使用这个标签里的<anonymous>元素。

1. **AnonymousAuthenticationToken**

匿名身份验证实体：Authentication接口的匿名登录实现类，同时也负责存储匿名身份的权限GrantedAuthority.

1. **AnonymousAuthenticationProvider**

匿名身份验证器：用于验证AnonymousAuthenticationToken的AuthenticationProvider，需要放置到providerManager的属性里。

1. **AnonymousAuthenticationFilter**

匿名身份验证过滤器，需要放到过滤器链中。位置是普通的登录验证过滤器之后，逻辑是：如果在SecurityContextHolder中没有对应的Authentication，他会自动把一个AnonymousAuthenticationToken加进去。

1. **具体配置：**

<bean id="anonymousAuthFilter"

class="org.springframework.security.web.authentication.AnonymousAuthenticationFilter">

<property name="key" value="foobar"/>

<property name="userAttribute" value="anonymousUser,ROLE\_ANONYMOUS"/>

</bean>

<bean id="anonymousAuthenticationProvider"

class="org.springframework.security.authentication.AnonymousAuthenticationProvider">

<property name="key" value="foobar"/>

</bean>

Key要在filter和provider中相同，保证他俩协同工作。

UserAttribute是一个表达式，格式是：usernameInTheAuthenticationToken,grantedAuthority[,grantedAuthority]

1. **AuthenticationTrustResolver**

这个接口（他的实现类AuthenticationTrustResolverImpl）提供了一个isAnonymous(Authentication)方法，判断一个token是不是匿名的。

ExceptionTranslationFilter就用到了这个方法，当接收到AccessDeniedException时，可能意味着用户是通过匿名进来所以没有合适的权限。这时要做判断，这个Authentication是不是匿名，如果不是，就说明他真没有权限，如果是，则开启一个登陆点，让他登录。

1. WebSocket安全
2. 授权

访问控制那块讲了基本的控制流。复习一下就是SS能将不同系统里不同的授权控制体系融进来，然后加以统一的程序执行流程，达到框架的抽象统一目的。用户只需针对不同情况进行扩展某异功能即可

现在重点讲解AbstractSecurityInterceptor 的几个实现类，讨论如何细粒度的进行授权控制。外加自己的一点理解和扩展。

1. **权限定义实体—粒度**

\*个人理解：一个系统里授权的核心是定义权限的粒度。SS的模式是基于Role。即用户有什么角色，资源访问需要什么角色。这样的授权在应用中个人感觉有点太宽泛。因为一般后台系统里有各种各样的资源。像界面菜单，页面上的功能控制按钮，以及domain级别的数量会很庞大的各种访问条目。比方说CMS里的各种栏目，每个栏目下各种文章。这样就需要定义无数个role，SS里资源对应role又是直接放在内存里，配置也在xml中。这样内存负担太大，也不能实现后台配置。SS也说他给出的Role-base的体系不适用于domain级别的访问控制，但是他提供的ACL方案太难，而且需要按照他给的数据结构走，很死。

所以我想的是权限不要仅仅基于Role。这一点shiro就很好值得借鉴。他在Role级别下定义了permission，负责具体的资源粒度。即用户给予Role角色，Role又可以赋予不同的permission，资源直接和permission绑定。一个（或几个，尽量是单元粒度）资源对应一个permission。

这也符合我们平时对权限的认识，具体的后台系统里可以专门有一个配置role-permission-resource的页面来负责维护管理。

* SS里GrantedAuthority是用户拥有的单元实体类，里面有一个方法String getAuthority(),能返回一个String，代表了用户有什么权限。SS里默认的实现类是GrantedAuthorityImpl。用户一般是有一组这样的GrantedAuthority，授权的时候传递给AccessDecisionManager进行具体逻辑的比较。这样的好处是String读起来比较友好。但是如果觉得这个粒度不够好的话，可以具体写一个自己的实现类，然后这个访问直接返回空，这就等于考苏AccessDecisionManager针对我们提供的GA，需要特殊的授权实现类来处理。这是一个很好的扩展点，可以考虑基于byte的权限单元实体。
* SS里的ConfigAttribue表示访问某个资源所需的权限。之所以叫这个名我想是因为SS里这些定义都是配置在资源url或者注释上的。反正是**配置**进去的。真正实现当时也要配置，但最好不要把一段定义都直接读出来放到内存里（虽然SS默认是这么做的）

1. **AccessDicisionManager**

复习一下：

**void** decide(Authentication authentication, Object secureObject,

Collection<ConfigAttribute> attrs) **throws** AccessDeniedException;

**boolean** supports(ConfigAttribute attribute);

**boolean** supports(Class clazz);

* decide方法决定是否有权访问，如果没有则抛出异常。他需要的参数分别是:

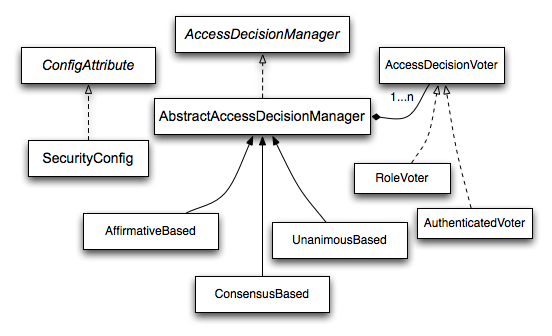
1. authentication：用户信息
2. secureObject：访问的资源对象，如方法调用里的MethodInvocation， web里的FilterInvocation。
3. CA集合：该资源上定义的访问所需权限。

* Supports(ConfigAttribute):表示某个具体实现类是否支持某类权限定义的检测
* Supports(Class secureObject):表示某一个实现类是否支持某种类型资源的检测

1. **基于投票器的AccessDicisionManager实现类**

SS给出的实现类一般都是基于投票体系的。即实现类里有一组AccessDecisionVoter。就是有很多个投票器，每个投票器可返回通过，不通过，弃权这几种结果，最后把这些结果汇总，基于某种策略（如一票否决制，或者必须全部通过才可访问）给出最终结果：

实现类结构图：



SS默认有三个实现类：

* ConsensusBased：综合否定票和通过票，那个多就按那个走。

这个类里面有一个属性定义如果票数相等，或者全部弃权时改怎么办。

* AffirmativeBased: 有一票通过即可

有一个属性决定如果都弃权怎么处理。

* UnanimousBased:一票否决制。必须都投出了通过票才表示能通过

有一个属性决定如果都弃权怎么处理。

也可以自己定制，一个想法是：定制一个voter接口实现类，能返回一个带有权重的投票结果，最后根据权重给出一个决定意见。

1. **AccessDecisionVoter**

**int** vote(Authentication authentication, Object object, Collection<ConfigAttribute> attrs);

**boolean** supports(ConfigAttribute attribute);

**boolean** supports(Class clazz);

方法意思基本和AccessDecisionManager类似。有三个值

int ACCESS\_GRANTED = 1;//允许

int ACCESS\_ABSTAIN = 0;//弃权

int ACCESS\_DENIED = -1;//不允许

1. **AccessDecisionVoter实现类：**

* RoleVoter：SS常用的投票器。他支持以ROLE\_开头的权限定义实体。

只要GA中有ROLE\_开头的权限和CA中ROLE\_开头的权限有一个对应上，他就投通过票。

如果一个也没对上就投反对票。

如果CA中没有ROLE\_开头的权限就弃权。

* AuthenticatedVoter：判断是匿名，还是完全登录，还是通过remeberMe登录。

如果权限定义上用了IS\_AUTHENTICATED\_ANONYMOUSLY .就会走这个投票器。

* RoleHierarchyVoter：就是Role加上了层级关系。比方说ROLE\_ADMIN肯定包含了ROLE\_USER,ROLE\_GUEST就可以用这个。

贴配置  
<bean id="roleVoter" class="org.springframework.security.access.vote.RoleHierarchyVoter">

<constructor-arg ref="roleHierarchy" />

</bean>

<bean id="roleHierarchy"

class="org.springframework.security.access.hierarchicalroles.RoleHierarchyImpl">

<property name="hierarchy">

<value>

ROLE\_ADMIN > ROLE\_STAFF

ROLE\_STAFF > ROLE\_USER

ROLE\_USER > ROLE\_GUEST

</value>

</property>

</bean>

里面有一个RoleHierarchyImpl，负责维护层级关系。

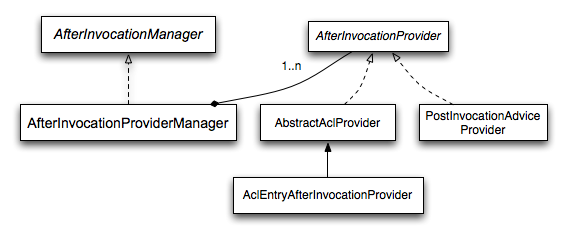
ROLE\_ADMIN > ROLE\_STAFF 就表示ADMIN里含有STAFF. >表示包含。

1. **调用后的处理器：**

AbstractSecurityInterceptor的执行流程是：在执行都某个资源的调用（访问）前，执行AccessDecisionVoter，决定是否执行该资源。响应的，在该资源调用结束后，会执行AfterInvocationManager，对该资源的返回值可以做一些操作，比方说过滤结果集。我个人觉得有点类似AOP的后处理。

个人理解：如果要根据权限过滤结果集：我个人觉得在结果集返回后筛选效率不好，不如自己实现AOP，先定义好service层的query方法，最好有一个权限参数的重载版本，把权限拼到SQL中。AOP的advice中控制根据不同的权限给出不同的查询子句，然后调用重载版本的query方法。在数据库级别就把筛选的操作完成了。省的数据库读出来了再用java筛。

类图：



结构和AuthenticationManager那块类似：

AfterInvocationManager有一个默认实现类AfterInvocationProviderManager，里面有一组AfterInvocationProvider.每一个AfterInvocationProvider实现类都能修改返回结果，或者抛出AccessDeniedException.因为是一组list，一般前provider一个修改后的结果再传给下一个Provider进行操作。

个人理解：如果我们AOP方式过滤数据，加上了权限串后返回的结果集是空，（因为有事用户没有权限访问任何一个domain）那么可以在这里判断一下，抛一个AccessDenyExcepton，省的在后续方法里判断这个结果集空不空？

1. **保护方法调用**

方法调用的保护推荐直接用命名空间比较好。SS保护方法调用（MethodInvocation），用的是MethodSecurityInterceptror。内部有一个MethodSecurityMetadataSource接口，用来获取某个方法调用的所需权限。其中有一个实现类MapBasedMethodSecurityMetadataSource，如果配置文件中出现了<intercept-methods>或者<protect-point>元素，默认内部就会用这个实现类。他把方法名当做是CA的key值（方法名可能带通配符）。如果是基于注解的配置，则会用到MethodSecurityMetadataSource其他实现类。

* 使用MethodSecurityInterceptor

他使用的是SpringAOP代理机制，在配置文件中如下下：

<bean id="bankManagerSecurity" class=

"org.springframework.security.access.intercept.aopalliance.MethodSecurityInterceptor">

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

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

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

<property name="securityMetadataSource">

<sec:method-security-metadata-source>

<sec:protect method="com.mycompany.BankManager.delete\*" access="ROLE\_SUPERVISOR"/>

<sec:protect method="com.mycompany.BankManager.getBalance" access="ROLE\_TELLER,ROLE\_SUPERVISOR"/>

</sec:method-security-metadata-source>

</property>

</bean>

* 使用AspectJ Security Interceptor

AspectJ方式编译时植入的方式实现AOP。他使用的是SS提供的AspectJSecurityInterceptror

一般的，对于domian级别的保护，用AspectJ方式，对于业务层的方法保护用SpringAOP。

<bean id="bankManagerSecurity" class=

"org.springframework.security.access.intercept.aspectj.AspectJMethodSecurityInterceptor">

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

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

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

<property name="securityMetadataSource">

<sec:method-security-metadata-source>

<sec:protect method="com.mycompany.BankManager.delete\*" access="ROLE\_SUPERVISOR"/>

<sec:protect method="com.mycompany.BankManager.getBalance" access="ROLE\_TELLER,ROLE\_SUPERVISOR"/>

</sec:method-security-metadata-source>

</property>

</bean>

两个不同的方式除了SecurityInterceptor实现类不同，其他的到一样。他们能使用同一个SecurityMetaDataSource，应为SecurityMetaDataSource内部使用的是java反射机制并非AOP特有的类。

下一步是下一个aspect，如：

**package** org.springframework.security.samples.aspectj;

**import** org.springframework.security.access.intercept.aspectj.AspectJSecurityInterceptor;

**import** org.springframework.security.access.intercept.aspectj.AspectJCallback;

**import** org.springframework.beans.factory.InitializingBean;

**public** aspect DomainObjectInstanceSecurityAspect **implements** InitializingBean {

**private** AspectJSecurityInterceptor securityInterceptor;

pointcut domainObjectInstanceExecution(): target(PersistableEntity)

&& execution(**public** \* \*(..)) && !within(DomainObjectInstanceSecurityAspect);

Object around(): domainObjectInstanceExecution() {

**if** (**this**.securityInterceptor == null) {

**return** proceed();

}

AspectJCallback callback = **new** AspectJCallback() {

**public** Object proceedWithObject() {

**return** proceed();

}

};

**return** **this**.securityInterceptor.invoke(thisJoinPoint, callback);

}

**public** AspectJSecurityInterceptor getSecurityInterceptor() {

**return** securityInterceptor;

}

**public** **void** setSecurityInterceptor(AspectJSecurityInterceptor securityInterceptor) {

**this**.securityInterceptor = securityInterceptor;

}

**public** **void** afterPropertiesSet() **throws** Exception {

**if** (**this**.securityInterceptor == null)

**throw** **new** IllegalArgumentException("securityInterceptor required");

}

}

}

针对上面的例子解释一下：

这个例子会让Security Interceptor应用到每一个PersisitableEntity上（pointCut定义的）

在around()这种advice中，要定义一个AspectJCallback，因为要执行processed()方法。把这个AspectJCallback对象连同连接点信息当做参数传递给this.securityInterceptor.invoke(thisJoinPoint, callback);

这样securityInterceptor内部判定如果有权执行，就执行callback的process().

因为要在这个aspect里注入securityInterceptor，所以要让spring管理这个aspect。

配置如下：

<bean id="domainObjectInstanceSecurityAspect"

class="security.samples.aspectj.DomainObjectInstanceSecurityAspect"

factory-method="aspectOf">

<property name="securityInterceptor" ref="bankManagerSecurity"/>

</bean>

1. **\*基于表达式的访问控制**

SS3.0以后引入了基于SpringEL的表达式，来判断有无权访问。他作为基于投票机制的一种扩展，能够允许更复杂的比较裸机，而且可以把这些逻辑融进一个表达式里。

作为表达式予以按，一般在表达式上下文中（这里叫evaluation context）都有一个根对象.SS里为web和方法调用指定了特定的跟对象，并且提过了一些内建的表达式

最基础的跟对象是SecurityExpressionRoot，他同时服务于web和method。

他也提供了内建的表达式：

|  |  |
| --- | --- |
| 表达式 | 描述 |
| hasRole([role]) | 如果用户身份中含有指定的Role，即返回true  默认的Role可以是不带“ ROLE\_”开头的字符，这个可以通过  DefaultWebSecurityExpressionHandler中的defaultRolePrefix来配置 |
| hasAnyRole([role1,role2]) | 如果用户身份中含有指定的权限集合中任意一个即可。权限集合用逗号分隔  可不带ROLE\_前缀,定制方法同上。 |
| hasAuthority([authority]) | 用户的principal里不含有authority字符。 |
| hasAnyAuthority([authority1,authority2]) | 用户principal是否含有任意一个authority字符，逗号分隔 |
| principal | 可以理解成一个内置对象，直接访问用户的principal |
| authentication | 内置对象，获取SecurityContext当前用户对应的Authentication对象。 |
| permitAll | 永远返回true，即可以访问 |
| denyAll | 永远返回false，即不允许访问 |
| isAnonymous() | 当前用户是否是匿名登录 |
| isRememberMe() | 当前用户是否是通过rememberMe进来的 |
| isAuthenticated() | 当前用户不是匿名用户（rememberMe进来或者登录进来） |
| isFullyAuthenticated() | 当前用户是经过验证进来的（不是通过rememberMe或是匿名登录进来的） |
| hasPermission(Object target, Object permission) | 当一个方法本是不是要保护的对象，但他里面可能涉及到安全对象，比方说方法参数需要一个安全对象，他要处理一个安全对象这样的。  同时指定访问的安全对象和所需要的权限。如：hasPermission(domainObject, 'read') |
| hasPermission(Object targetId, String targetType, Object permission) | 对于方法里含受保护对象的情况，同时制定对象ID和对象类型，所需要的权限，如：  hasPermission(1, 'com.example.domain.Message', 'read') |

* web环境的表达式

受保护对象是一个web环境的url制定的对象。

在命名空间里用，需要设置< http use-expressions=true> 此时SS就会期望<intercept-url>中的access指定的是一个表达式，如果表达式最终返回true就表示可以访问。

<http>

<intercept-url pattern="/admin\*"

access="hasRole('admin') and hasIpAddress('192.168.1.0/24')"/>

...

</http>

Web环境用的是WebSecurityExpressionRoot，他继承了SecurityExpressionRoot又提供了hasIpAddress()这个内建方法，并且有提供了一个内置变量 request:代表了访问的HttpServletRequest对象。

\* 这个的web表达式其实就是在AccessDecisionManager里加入了WebExpressionVoter这个投票器，如果不用命名空间，自己把对应的投票器加进去就行啦

* Method环境的表达式

受保护对象是一个方法调用时，除了简单的允许不允许规则外，SS3.0以后又加入了几个注解，支持更全面的表达式使用。

@Pre 和 @Post方式：

首先在命名空间配置：

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

然后就可以用额外的四个注解：

@PreAuthorize(expression)：方法调用前检测权限

@PostAuthorize(expression)：方法调用后检测权限

@PreFilter(expression)：方法调用前做过滤

@PostFilter(expression)：调用后过滤。

在方法上加上对应的注解，就表示如果表达式返回true才能访问，或者按照表达式的条件，过滤返回true的值。

这个expression里可以写方法参数的具体指，如

*@PreAuthorize("hasPermission(#contact, 'admin')")*

**public** **void** deletePermission(Contact contact, Sid recipient, Permission permission);

#contact就代表了contact。

这里面其实应用到了参数名自动发现这个机制，有很多方法可以将餐使命暴露到Spel中

比方说是SS特有的@P将参数暴露到了Spel中：

*@PreAuthorize("#c.name == authentication.name")*

**public** **void** doSomething(*@P("c")* Contact contact);

再比方说SpringData提供的@Param

*@PreAuthorize("#n == authentication.name")*

Contact findContactByName(*@Param("n")* String name);

他们都是扩展了AnnotationParameterNameDiscoverer

在比方说JDK8，把参数名编译到class中的选项，Spring4以后以后这么用，这样用java的反射就可以获取到用户名。（接口，类都可以这么干）

在比方说，我们主导编译的时候用debug模式，就会带参数信息了。（但不包括接口类型的参数信息？）

再看一个Filter的例子

*@PreAuthorize("hasRole('USER')")*

*@PostFilter("hasPermission(filterObject, 'read') or hasPermission(filterObject, 'admin')")*

**public** List<Contact> getAll();

getAll返回一个contact集合，PostFilter会过滤返回值，其中的filterObject内置对象，代表了返回集合放的对象，这里面就是contact对象。

@preFilter和这个差不多，如果参数里有多个集合值类型，可以通过filterTarget这个表达式内置对象指定对那个集合进行过滤。

还有一个returnValue表示返回值。

* 单讲@hasPermission()

他其实是可以涉及到domain级别的权限了，他是ACL部分和SS 方法保护的桥梁。我们可以改造这个注解，实现自己domian级别的方法保护。所以很重要。

SecurityExpressionRoot中的hasPermission()实现内部其实是委托给了PermissionEvaluator接口

public boolean hasPermission(Object target, Object permission) {

return permissionEvaluator.hasPermission(authentication, target, permission);

}

public boolean hasPermission(Object targetId, String targetType, Object permission) {

return permissionEvaluator.hasPermission(authentication, (Serializable) targetId,

targetType, permission);

}

这个接口其实是桥接SS的ACL模式用的，但我们可以写一个自己的实现类，从而控制domian级别的访问。

**boolean** hasPermission(Authentication authentication, Object targetDomainObject,

Object permission);

**boolean** hasPermission(Authentication authentication, Serializable targetId,

String targetType, Object permission);

第一个方法是用在受保护对象是哪一个实例在已经知晓（已经在方法中载入了，受到SS的控制），判断用户拥有对于要访问的targetDomainObject是否有permission权限。

第二个方式是用在受保护对象实例还没有载入，我们只知道这个对象的一个id。对于这个方法，id的类型也需要指定，以供能正确读取到permission。id的作用就是能够让我们从数据库（或者其他地方），读到这个id对应的实例需要什么访问权限（permission）。

要使用hasPermission()表达式，需要明确配置一个PermissionEvaluator。

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

<security:expression-handler ref="expressionHandler"/>

</security:global-method-security>

<bean id="expressionHandler" class=

"org.springframework.security.access.expression.method.DefaultMethodSecurityExpressionHandler">

<property name="permissionEvaluator" ref="myPermissionEvaluator"/>

</bean>

myPermissionEvaluator可以自己实现，在SS的ACL模块中，默认用的是AclPermissionEvaluator，可以参考。

1. 测试，
2. **初始化**：

*@RunWith(SpringJUnit4ClassRunner.class)*

*@ContextConfiguration*

**public** **class** WithMockUserTests {

和一般的SpringTest一模一样.SS底层使用了WithSecurityContextTestExecutionListener确保在conext载入前将SecurityContextHolder暴露到context，测试结束后，清空holder。

如果你只想测SS，不需要Spring上下文。可替换@ContextConfiguration为@SecurityExecutionListeners

1. **@WithMockUser：**模拟用户身份

使用UsernamePasswordAuthenticationToken作为Authtication实现

Principal是SS中默认的User类

1. **@WithUserDetails** 模拟用户身份

使用UsernamePasswordAuthenticationToken作为Authtication实现

Principal是SS中的一个我们自己定制的类

可以通过吧UserDetailsService当做一个bean，暴露到上下文中，从而帮助获取具体的UserDetails实现，原理同@WithUser 但是principal是用户定制的一个类

1. **@WithSecurityContext** 他是一个用于标注注解的注解。

如果要完全定制SecurityContext中的内容。可以用到它

1. 自定义一个注解，给他标注上@WithSecurityContext

@WithSecurityContext(factory = WithMockCustomUserSecurityContextFactory.class)

public @interface WithMockCustomUser

1. 实现一个WithSecurityContextFactory接口，实现接口方法：

createSecurityContext,用来填充对应的SecurityContext

自己实现的SecurityContextFactory可以当做普通的Spring bean里使用，也可以给facotory注入属性

**public** **class** WithMockCustomUserSecurityContextFactory

**implements** WithSecurityContextFactory<WithMockCustomUser> {

*@Override*

**public** SecurityContext createSecurityContext(WithMockCustomUser customUser) {

SecurityContext context = SecurityContextHolder.createEmptyContext();

CustomUserDetails principal =

**new** CustomUserDetails(customUser.name(), customUser.username());

Authentication auth =

**new** UsernamePasswordAuthenticationToken(principal, "password", principal.getAuthorities());

context.setAuthentication(auth);

**return** context;

}

}

**final** **class** WithUserDetailsSecurityContextFactory

**implements** WithSecurityContextFactory<WithUserDetails> {

**private** UserDetailsService userDetailsService;

*@Autowired*

**public** WithUserDetailsSecurityContextFactory(UserDetailsService userDetailsService) {

**this**.userDetailsService = userDetailsService;

}

**public** SecurityContext createSecurityContext(WithUserDetails withUser) {

String username = withUser.value();

Assert.hasLength(username, "value() must be non empty String");

UserDetails principal = userDetailsService.loadUserByUsername(username);

Authentication authentication = **new** UsernamePasswordAuthenticationToken(principal, principal.getPassword(), principal.getAuthorities());

SecurityContext context = SecurityContextHolder.createEmptyContext();

context.setAuthentication(authentication);

**return** context;

}

}

1. **Web测试初始化-与**SpringMVC集成测试

和SpringMVC组合测试，需要添加Ffilter：SS的FilterChainProxy  
还需添加TestSecurityContextHolderPostProcesso实现以某种身份执行测试  
可以通过调用SecurityMockMvcConfigurers.springSecurity() 来实现。这个方法会实现初始化操作

*@Before*

**public** **void** setup() {

mvc = MockMvcBuilders.webAppContextSetup(context)

.apply(SecurityMockMvcConfigurers。springSecurity())

.build();

}

1. **SecurityMockMvcRequestPostProcessors—实现在mockMVC中将身份信息设置到模拟的request中**

SpringMVC Test提供了RequestPostProcessor 接口，用来修改request

SS针对他，提供了一系列的实现。可以在Test类中静态导入SS的实现类

**import** **static** org.springframework.security.test.web.servlet.request.SecurityMockMvcRequestPostProcessors.\*;

* mvc.perform(get("/admin").with(user("admin").password("pass").roles("USER","ADMIN")))

模拟一个登陆身份，以UsernamePasswordAuthenticationToken作为Authentication的实现。Authentication里的principle存放的是SS默认的User类对象。

* mvc.perform(get("/").with(user(userDetails)))

如果定制了UserDetails用这个。他会在Authentication的里存放你定制的UserDetails

* mvc.perform(get("/").with(securityContext(securityContext)))

如果定制的securityContext用这个

\*以某一身份运行测试的最佳实践：

其实有两种方式：

1是在方法是假@WithXXX注解

2是使用SS的 SecurityMockMvcRequestPostProcessors静态类，然后在mock对象上调用with（xxx）方法。

如果一个测试类的方法都要以相同身份运行可以在@Before方法里写：

mvc = MockMvcBuilders

.webAppContextSetup(context)

.defaultRequest(get("/").with(user("user").roles("ADMIN")))

.apply(springSecurity())

.build();

这样所有的测试方法，都以某一身份运行。

如果很多方法都需要统一身份，也可以把模拟身份抽出来写一个静态方法。

public static RequestPostProcessor rob() {

return user("rob").roles("ADMIN");

}

Movk对象调用时：mvc .perform(get("/").with(rob()))

1. **SecurityMockMvcRequestBuilders**

SpringMVCTest提供了一个RequestBuilder 接口，创建MockHttpServletRequest

SS提供了一些RequestBuilder接口的实现，可在测试类静态导入“

**import** **static** org.springframework.security.test.web.servlet.request.SecurityMockMvcRequestBuilders.\*;

* formLogin(): 模拟登陆请求

mvc.perform(formLogin()

默认：POST发送请求到/login，参数username=user&password=password，并提供一个合法的CSRF Token

mvc.perform(formLogin("/auth").user("admin").password("pass")):

定制提交url和提交内容：以username=admin&password=pass提交form到auth.

mvc.perform(formLogin("/auth").user("a","admin").password("p","pass"))

可以定制url和提交参数。这里提交到/auth里的是a=admin&p=pass

* logout()

mvc.perform(logout())

模拟提交POST请求到/logout并带有一个合法的CSRF Token

mvc.perform(logout("/signout"))

定制POST参数为：/signout

1. **SecurityMockMvcResultMatchers:**

SpringMVCTest里有ResultMatcher，提供web相关的断言

SS实现类这个接口，提供了SecurityMockMvcResultMatchers，可静态导入

**import static org.springframework.security.test.web.servlet.response.SecurityMockMvcResultMatchers.\*;**

* 验证失败

mvc .perform(formLogin().password("invalid"))

.andExpect(unauthenticated());

* 验证成功

mvc.perform(formLogin()).andExpect(authenticated());

验证是否是某一用户名

Mvc .perform(formLogin().user("admin"))

.andExpect(authenticated().withUsername("admin"));

验证是有拥有角色

mvc .perform(formLogin().user("admin"))

.andExpect(authenticated().withRoles("USER","ADMIN"));

1. 本地化：

SS支持异常信息的本地化，但不支持调试信息本地化。

SS实现了Spring的MessageSourceAware接口，期待Application初始化的时候注入message resolver用来解析messages.properties.SS的core.jar中内含了一些messages.properties。你也可以自己定制，如

<bean id="messageSource"

class="org.springframework.context.support.ReloadableResourceBundleMessageSource">

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

</bean>

SS需要Spring提供的localization支持，以查找具体使用哪个properties文件。所以要保证request中的local信息存储在spring的org.springframework.context.i18n.LocaleContextHolder中，在SpringMVC里，

DispatcherServlet负责存储local信息，但是SS的Filter早于servlet执行，所以要提前将local信息设置到LocaleContextHolder中。可以自己写一个Filter干这件事，或者使用Spring提供的RequestContextFilter。

注意！在WEB.XML定义中，一定要把这个Filter放到SS的之前

1. 跨站请求伪造CSRF 保护
2. 保护HTTP请求头