[spring security 1](http://www.cnblogs.com/fenglan/p/5912714.html)

首先我们为Spring Security专门建立一个Spring的配置文件，该文件就专门用来作为Spring Security的配置。使用Spring Security我们需要引入Spring Security的NameSpace。

<beans xmlns="http://www.springframework.org/schema/beans"

  xmlns:security="http://www.springframework.org/schema/security"

  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

  xsi:schemaLocation="http://www.springframework.org/schema/beans

          http://www.springframework.org/schema/beans/spring-beans-3.1.xsd

          http://www.springframework.org/schema/security

          http://www.springframework.org/schema/security/spring-security-3.1.xsd">

</beans>

       Spring Security命名空间的引入可以简化我们的开发，它涵盖了大部分Spring Security常用的功能。它的设计是基于框架内大范围的依赖的，可以被划分为以下几块。

* Web/Http 安全：这是最复杂的部分。通过建立filter和相关的service bean来实现框架的认证机制。当访问受保护的URL时会将用户引入登录界面或者是错误提示界面。
* 业务对象或者方法的安全：控制方法访问权限的。
* AuthenticationManager：处理来自于框架其他部分的认证请求。
* AccessDecisionManager：为Web或方法的安全提供访问决策。会注册一个默认的，但是我们也可以通过普通bean注册的方式使用自定义的AccessDecisionManager。
* AuthenticationProvider：AuthenticationManager是通过它来认证用户的。
* UserDetailsService：跟AuthenticationProvider关系密切，用来获取用户信息的。

       引入了Spring Security的NameSpace之后我们就可以使用该命名空间下的元素来配置Spring Security了。首先我们来定义一个http元素，security只是我们使用命名空间的一个前缀。http元素是用于定义Web相关权限控制的。

   <security:http auto-config="true">

      <security:intercept-url pattern="/\*\*" access="ROLE\_USER"/>

   </security:http>

       如上定义中，intercept-url定义了一个权限控制的规则。pattern属性表示我们将对哪些url进行权限控制，其也可以是一个正则表达式，如上的写法表示我们将对所有的URL进行权限控制；access属性表示在请求对应的URL时需要什么权限，默认配置时它应该是一个以逗号分隔的角色列表，请求的用户只需拥有其中的一个角色就能成功访问对应的URL。这里的“ROLE\_USER”表示请求的用户应当具有ROLE\_USER角色。“ROLE\_”前缀是一个提示Spring使用基于角色的检查的标记。

       有了权限控制的规则了后，接下来我们需要定义一个AuthenticationManager用于认证。我们先来看如下定义：

   <security:authentication-manager>

      <security:authentication-provider>

         <security:user-service>

            <security:user name="user" password="user" authorities="ROLE\_USER"/>

            <security:user name="admin" password="admin" authorities="ROLE\_USER, ROLE\_ADMIN"/>

         </security:user-service>

      </security:authentication-provider>

   </security:authentication-manager>

       authentication-manager元素指定了一个AuthenticationManager，其需要一个AuthenticationProvider（对应authentication-provider元素）来进行真正的认证，默认情况下authentication-provider对应一个DaoAuthenticationProvider，其需要UserDetailsService（对应user-service元素）来获取用户信息UserDetails（对应user元素）。这里我们只是简单的使用user元素来定义用户，而实际应用中这些信息通常都是需要从数据库等地方获取的，这个将放到后续再讲。我们可以看到通过user元素我们可以指定user对应的用户名、密码和拥有的权限。user-service还支持通过properties文件来指定用户信息，如：

   <security:user-service properties="/WEB-INF/config/users.properties"/>

       其中属性文件应遵循如下格式：

username=password,grantedAuthority[,grantedAuthority][,enabled|disabled]

       所以，对应上面的配置文件，我们的users.properties文件的内容应该如下所示：

#username=password,grantedAuthority[,grantedAuthority][,enabled|disabled]

user=user,ROLE\_USER

admin=admin,ROLE\_USER,ROLE\_ADMIN

       至此，我们的Spring Security配置文件的配置就完成了。完整配置文件将如下所示。

<beans xmlns="http://www.springframework.org/schema/beans"

  xmlns:security="http://www.springframework.org/schema/security"

  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

  xsi:schemaLocation="http://www.springframework.org/schema/beans

          http://www.springframework.org/schema/beans/spring-beans-3.1.xsd

          http://www.springframework.org/schema/security

          http://www.springframework.org/schema/security/spring-security-3.1.xsd">

   <security:http auto-config="true">

      <security:intercept-url pattern="/\*\*" access="ROLE\_USER"/>

   </security:http>

   <security:authentication-manager>

      <security:authentication-provider>

         <security:user-service>

            <security:user name="user" password="user" authorities="ROLE\_USER"/>

            <security:user name="admin" password="admin" authorities="ROLE\_USER, ROLE\_ADMIN"/>

         </security:user-service>

      </security:authentication-provider>

   </security:authentication-manager>

</beans>

       之后我们告诉Spring加载这个配置文件。通常，我们可以在web.xml文件中通过context-param把它指定为Spring的初始配置文件，也可以在对应Spring的初始配置文件中引入它。这里我们采用前者。

   <context-param>

      <param-name>contextConfigLocation</param-name>

   <param-value>/WEB-INF/config/applicationContext.xml,/WEB-INF/config/spring-security.xml</param-value>

   </context-param>

   <listener>

   <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

   </listener>

       Spring的配置文件是通过对应的ContextLoaderListener来加载和初始化的，上述代码中的applicationContext.xml文件就是对应的Spring的配置文件，如果没有可以不用配置。接下来我们还需要在web.xml中定义一个filter用来拦截需要交给Spring Security处理的请求，需要注意的是该filter一定要定义在其它如SpringMVC等拦截请求之前。这里我们将拦截所有的请求，具体做法如下所示：

   <filter>

      <filter-name>springSecurityFilterChain</filter-name>

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

   </filter>

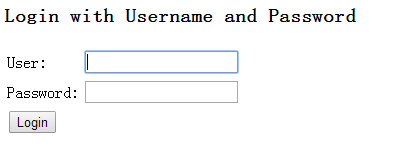
   <filter-mapping>

      <filter-name>springSecurityFilterChain</filter-name>

      <url-pattern>/\*</url-pattern>

   </filter-mapping>

       接下来可以启动我们的应用，然后在浏览器中访问我们的主页。你会看到如下页面。

       因为我们的spring-security.xml文件中配置好了所有的请求都需要“ROLE\_USER”权限，所以当我们在请求主页的时候，Spring Security发现我们还没有登录，Spring会引导我们到登录界面。使用正确的用户名和密码（如上面配置的user/user或admin/admin）登录后，如果符合对应的权限我们就可以访问主页了，否则将出现403（禁止访问）界面。

       可能你会奇怪，我们没有建立上面的登录页面，为什么Spring Security会跳到上面的登录页面呢？这是我们设置http的auto-config=”true”时Spring Security自动为我们生成的。

       当指定http元素的auto-config=”true”时，就相当于如下内容的简写。

   <security:http>

      <security:form-login/>

      <security:http-basic/>

      <security:logout/>

   </security:http>

       这些元素负责建立表单登录、基本的认证和登出处理。它们都可以通过指定对应的属性来改变它们的行为。

# [Spring Security（02）——关于登录](http://www.cnblogs.com/fenglan/p/5912751.html)

**目录**

1.1     form-login元素介绍

1.1.1    使用自定义登录页面

1.1.2    指定登录后的页面

1.1.3    指定登录失败后的页面

1.2     http-basic

## 1.1     form-login元素介绍

       http元素下的form-login元素是用来定义表单登录信息的。当我们什么属性都不指定的时候Spring Security会为我们生成一个默认的登录页面。如果不想使用默认的登录页面，我们可以指定自己的登录页面。

### 1.1.1   使用自定义登录页面

       自定义登录页面是通过login-page属性来指定的。提到login-page我们不得不提另外几个属性。

* username-parameter：表示登录时用户名使用的是哪个参数，默认是“j\_username”。
* password-parameter：表示登录时密码使用的是哪个参数，默认是“j\_password”。
* login-processing-url：表示登录时提交的地址，默认是“/j-spring-security-check”。这个只是Spring Security用来标记登录页面使用的提交地址，真正关于登录这个请求是不需要用户自己处理的。

       所以，我们可以通过如下定义使Spring Security在需要用户登录时跳转到我们自定义的登录页面。

   <security:http auto-config="true">

      <security:form-login login-page="/login.jsp"

         login-processing-url="/login.do" username-parameter="username"

         password-parameter="password" />

      <!-- 表示匿名用户可以访问 -->

      <security:intercept-url pattern="/login.jsp" access="IS\_AUTHENTICATED\_ANONYMOUSLY"/>

      <security:intercept-url pattern="/\*\*" access="ROLE\_USER" />

   </security:http>

       需要注意的是，我们之前配置的是所有的请求都需要ROLE\_USER权限，这意味着我们自定义的“/login.jsp”也需要该权限，这样就会形成一个死循环了。解决办法是我们需要给“/login.jsp”放行。通过指定“/login.jsp”的访问权限为“IS\_AUTHENTICATED\_ANONYMOUSLY”或“ROLE\_ANONYMOUS”可以达到这一效果。此外，我们也可以通过指定一个http元素的安全性为none来达到相同的效果。如：

   <security:http security="none" pattern="/login.jsp" />

   <security:http auto-config="true">

      <security:form-login login-page="/login.jsp"

         login-processing-url="/login.do" username-parameter="username"

         password-parameter="password" />

      <security:intercept-url pattern="/\*\*" access="ROLE\_USER" />

   </security:http>

       它们两者的区别是前者将进入Spring Security定义的一系列用于安全控制的filter，而后者不会。当指定一个http元素的security属性为none时，表示其对应pattern的filter链为空。从3.1开始，Spring Security允许我们定义多个http元素以满足针对不同的pattern请求使用不同的filter链。当为指定pattern属性时表示对应的http元素定义将对所有的请求发生作用。

       根据上面的配置，我们自定义的登录页面的内容应该是这样子的：

   <form action="login.do" method="post">

      <table>

         <tr>

            <td>用户名：</td>

            <td><input type="text" name="username"/></td>

         </tr>

         <tr>

            <td>密码：</td>

            <td><input type="password" name="password"/></td>

         </tr>

         <tr>

            <td colspan="2" align="center">

                <input type="submit" value="登录"/>

                <input type="reset" value="重置"/>

            </td>

         </tr>

      </table>

   </form>

### 1.1.2   指定登录后的页面

**通过default-target-url指定**

       默认情况下，我们在登录成功后会返回到原本受限制的页面。但如果用户是直接请求登录页面，登录成功后应该跳转到哪里呢？默认情况下它会跳转到当前应用的根路径，即欢迎页面。通过指定form-login元素的default-target-url属性，我们可以让用户在直接登录后跳转到指定的页面。如果想让用户不管是直接请求登录页面，还是通过Spring Security引导过来的，登录之后都跳转到指定的页面，我们可以通过指定form-login元素的always-use-default-target属性为true来达到这一效果。

**通过authentication-success-handler-ref指定**

       authentication-success-handler-ref对应一个AuthencticationSuccessHandler实现类的引用。如果指定了authentication-success-handler-ref，登录认证成功后会调用指定AuthenticationSuccessHandler的onAuthenticationSuccess方法。我们需要在该方法体内对认证成功做一个处理，然后返回对应的认证成功页面。使用了authentication-success-handler-ref之后认证成功后的处理就由指定的AuthenticationSuccessHandler来处理，之前的那些default-target-url之类的就都不起作用了。

       以下是自定义的一个AuthenticationSuccessHandler的实现类。

**publicclass** AuthenticationSuccessHandlerImpl **implements**

      AuthenticationSuccessHandler {

**publicvoid** onAuthenticationSuccess(HttpServletRequest request,

         HttpServletResponse response, Authentication authentication)

**throws** IOException, ServletException {

      response.sendRedirect(request.getContextPath());

   }

}

       其对应使用authentication-success-handler-ref属性的配置是这样的：

   <security:http auto-config="true">

      <security:form-login login-page="/login.jsp"

         login-processing-url="/login.do" username-parameter="username"

         password-parameter="password"

         authentication-success-handler-ref="authSuccess"/>

      <!-- 表示匿名用户可以访问 -->

      <security:intercept-url pattern="/login.jsp"

         access="IS\_AUTHENTICATED\_ANONYMOUSLY" />

      <security:intercept-url pattern="/\*\*" access="ROLE\_USER" />

   </security:http>

   <!-- 认证成功后的处理类 -->

   <bean id="authSuccess" class="com.xxx.AuthenticationSuccessHandlerImpl"/>

### 1.1.3   指定登录失败后的页面

       除了可以指定登录认证成功后的页面和对应的AuthenticationSuccessHandler之外，form-login同样允许我们指定认证失败后的页面和对应认证失败后的处理器AuthenticationFailureHandler。

**通过authentication-failure-url指定**

       默认情况下登录失败后会返回登录页面，我们也可以通过form-login元素的authentication-failure-url来指定登录失败后的页面。需要注意的是登录失败后的页面跟登录页面一样也是需要配置成在未登录的情况下可以访问，否则登录失败后请求失败页面时又会被Spring Security重定向到登录页面。

   <security:http auto-config="true">

      <security:form-login login-page="/login.jsp"

         login-processing-url="/login.do" username-parameter="username"

         password-parameter="password"

         authentication-failure-url="/login\_failure.jsp"

         />

      <!-- 表示匿名用户可以访问 -->

      <security:intercept-url pattern="/login\*.jsp\*"

         access="IS\_AUTHENTICATED\_ANONYMOUSLY" />

      <security:intercept-url pattern="/\*\*" access="ROLE\_USER" />

   </security:http>

**通过authentication-failure-handler-ref指定**

       类似于authentication-success-handler-ref，authentication-failure-handler-ref对应一个用于处理认证失败的AuthenticationFailureHandler实现类。指定了该属性，Spring Security在认证失败后会调用指定AuthenticationFailureHandler的onAuthenticationFailure方法对认证失败进行处理，此时authentication-failure-url属性将不再发生作用。

## 1.2     http-basic

       之前介绍的都是基于form-login的表单登录，其实Spring Security还支持弹窗进行认证。通过定义http元素下的http-basic元素可以达到这一效果。

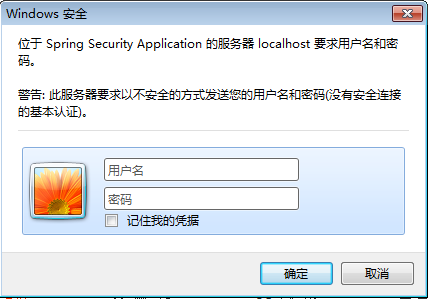
   <security:http auto-config="true">

      <security:http-basic/>

      <security:intercept-url pattern="/\*\*" access="ROLE\_USER" />

   </security:http>

       此时，如果我们访问受Spring Security保护的资源时，系统将会弹出一个窗口来要求我们进行登录认证。效果如下：

       当然此时我们的表单登录也还是可以使用的，只不过当我们访问受包含资源的时候Spring Security不会自动跳转到登录页面。这就需要我们自己去请求登录页面进行登录。

       需要注意的是当我们同时定义了http-basic和form-login元素时，form-login将具有更高的优先级。即在需要认证的时候Spring Security将引导我们到登录页面，而不是弹出一个窗口

# [Spring Security（03）——核心类简介](http://www.cnblogs.com/fenglan/p/5912820.html)

**目录**

1.1     Authentication

1.2     SecurityContextHolder

1.3     AuthenticationManager和AuthenticationProvider

1.3.1    认证成功后清除凭证

1.4     UserDetailsService

1.4.1    JdbcDaoImpl

1.4.2    InMemoryDaoImpl

1.5     GrantedAuthority

## 1.1     Authentication

       Authentication是一个接口，用来表示用户认证信息的，在用户登录认证之前相关信息会封装为一个Authentication具体实现类的对象，在登录认证成功之后又会生成一个信息更全面，包含用户权限等信息的Authentication对象，然后把它保存在SecurityContextHolder所持有的SecurityContext中，供后续的程序进行调用，如访问权限的鉴定等。

## 1.2     SecurityContextHolder

       SecurityContextHolder是用来保存SecurityContext的。SecurityContext中含有当前正在访问系统的用户的详细信息。默认情况下，SecurityContextHolder将使用ThreadLocal来保存SecurityContext，这也就意味着在处于同一线程中的方法中我们可以从ThreadLocal中获取到当前的SecurityContext。因为线程池的原因，如果我们每次在请求完成后都将ThreadLocal进行清除的话，那么我们把SecurityContext存放在ThreadLocal中还是比较安全的。这些工作Spring Security已经自动为我们做了，即在每一次request结束后都将清除当前线程的ThreadLocal。

       SecurityContextHolder中定义了一系列的静态方法，而这些静态方法内部逻辑基本上都是通过SecurityContextHolder持有的SecurityContextHolderStrategy来实现的，如getContext()、setContext()、clearContext()等。而默认使用的strategy就是基于ThreadLocal的ThreadLocalSecurityContextHolderStrategy。另外，Spring Security还提供了两种类型的strategy实现，GlobalSecurityContextHolderStrategy和InheritableThreadLocalSecurityContextHolderStrategy，前者表示全局使用同一个SecurityContext，如C/S结构的客户端；后者使用InheritableThreadLocal来存放SecurityContext，即子线程可以使用父线程中存放的变量。

       一般而言，我们使用默认的strategy就可以了，但是如果要改变默认的strategy，Spring Security为我们提供了两种方法，这两种方式都是通过改变strategyName来实现的。SecurityContextHolder中为三种不同类型的strategy分别命名为MODE\_THREADLOCAL、MODE\_INHERITABLETHREADLOCAL和MODE\_GLOBAL。第一种方式是通过SecurityContextHolder的静态方法setStrategyName()来指定需要使用的strategy；第二种方式是通过系统属性进行指定，其中属性名默认为“spring.security.strategy”，属性值为对应strategy的名称。

       Spring Security使用一个Authentication对象来描述当前用户的相关信息。SecurityContextHolder中持有的是当前用户的SecurityContext，而SecurityContext持有的是代表当前用户相关信息的Authentication的引用。这个Authentication对象不需要我们自己去创建，在与系统交互的过程中，Spring Security会自动为我们创建相应的Authentication对象，然后赋值给当前的SecurityContext。但是往往我们需要在程序中获取当前用户的相关信息，比如最常见的是获取当前登录用户的用户名。在程序的任何地方，通过如下方式我们可以获取到当前用户的用户名。

**public** String getCurrentUsername() {

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

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

**return** ((UserDetails) principal).getUsername();

      }

**if** (principal **instanceof** Principal) {

**return** ((Principal) principal).getName();

      }

**return** String.valueOf(principal);

   }

       通过Authentication.getPrincipal()可以获取到代表当前用户的信息，这个对象通常是UserDetails的实例。获取当前用户的用户名是一种比较常见的需求，关于上述代码其实Spring Security在Authentication中的实现类中已经为我们做了相关实现，所以获取当前用户的用户名最简单的方式应当如下。

**public** String getCurrentUsername() {

**return** SecurityContextHolder.getContext().getAuthentication().getName();

   }

       此外，调用SecurityContextHolder.getContext()获取SecurityContext时，如果对应的SecurityContext不存在，则Spring Security将为我们建立一个空的SecurityContext并进行返回。

## 1.3     AuthenticationManager和AuthenticationProvider

       AuthenticationManager是一个用来处理认证（Authentication）请求的接口。在其中只定义了一个方法authenticate()，该方法只接收一个代表认证请求的Authentication对象作为参数，如果认证成功，则会返回一个封装了当前用户权限等信息的Authentication对象进行返回。

    Authentication authenticate(Authentication authentication) **throws**AuthenticationException;

       在Spring Security中，AuthenticationManager的默认实现是ProviderManager，而且它不直接自己处理认证请求，而是委托给其所配置的AuthenticationProvider列表，然后会依次使用每一个AuthenticationProvider进行认证，如果有一个AuthenticationProvider认证后的结果不为null，则表示该AuthenticationProvider已经认证成功，之后的AuthenticationProvider将不再继续认证。然后直接以该AuthenticationProvider的认证结果作为ProviderManager的认证结果。如果所有的AuthenticationProvider的认证结果都为null，则表示认证失败，将抛出一个ProviderNotFoundException。校验认证请求最常用的方法是根据请求的用户名加载对应的UserDetails，然后比对UserDetails的密码与认证请求的密码是否一致，一致则表示认证通过。Spring Security内部的DaoAuthenticationProvider就是使用的这种方式。其内部使用UserDetailsService来负责加载UserDetails，UserDetailsService将在下节讲解。在认证成功以后会使用加载的UserDetails来封装要返回的Authentication对象，加载的UserDetails对象是包含用户权限等信息的。认证成功返回的Authentication对象将会保存在当前的SecurityContext中。

       当我们在使用NameSpace时， authentication-manager元素的使用会使Spring Security 在内部创建一个ProviderManager，然后可以通过authentication-provider元素往其中添加AuthenticationProvider。当定义authentication-provider元素时，如果没有通过ref属性指定关联哪个AuthenticationProvider，Spring Security默认就会使用DaoAuthenticationProvider。使用了NameSpace后我们就不要再声明ProviderManager了。

   <security:authentication-manager alias="authenticationManager">

      <security:authentication-provider

         user-service-ref="userDetailsService"/>

   </security:authentication-manager>

       如果我们没有使用NameSpace，那么我们就应该在ApplicationContext中声明一个ProviderManager。

### 1.3.1    认证成功后清除凭证

       默认情况下，在认证成功后ProviderManager将清除返回的Authentication中的凭证信息，如密码。所以如果你在无状态的应用中将返回的Authentication信息缓存起来了，那么以后你再利用缓存的信息去认证将会失败，因为它已经不存在密码这样的凭证信息了。所以在使用缓存的时候你应该考虑到这个问题。一种解决办法是设置ProviderManager的eraseCredentialsAfterAuthentication 属性为false，或者想办法在缓存时将凭证信息一起缓存。

## 1.4     UserDetailsService

       通过Authentication.getPrincipal()的返回类型是Object，但很多情况下其返回的其实是一个UserDetails的实例。UserDetails是Spring Security中一个核心的接口。其中定义了一些可以获取用户名、密码、权限等与认证相关的信息的方法。Spring Security内部使用的UserDetails实现类大都是内置的User类，我们如果要使用UserDetails时也可以直接使用该类。在Spring Security内部很多地方需要使用用户信息的时候基本上都是使用的UserDetails，比如在登录认证的时候。登录认证的时候Spring Security会通过UserDetailsService的loadUserByUsername()方法获取对应的UserDetails进行认证，认证通过后会将该UserDetails赋给认证通过的Authentication的principal，然后再把该Authentication存入到SecurityContext中。之后如果需要使用用户信息的时候就是通过SecurityContextHolder获取存放在SecurityContext中的Authentication的principal。

       通常我们需要在应用中获取当前用户的其它信息，如Email、电话等。这时存放在Authentication的principal中只包含有认证相关信息的UserDetails对象可能就不能满足我们的要求了。这时我们可以实现自己的UserDetails，在该实现类中我们可以定义一些获取用户其它信息的方法，这样将来我们就可以直接从当前SecurityContext的Authentication的principal中获取这些信息了。上文已经提到了UserDetails是通过UserDetailsService的loadUserByUsername()方法进行加载的。UserDetailsService也是一个接口，我们也需要实现自己的UserDetailsService来加载我们自定义的UserDetails信息。然后把它指定给AuthenticationProvider即可。如下是一个配置UserDetailsService的示例。

   <!-- 用于认证的AuthenticationManager -->

   <security:authentication-manager alias="authenticationManager">

      <security:authentication-provider

         user-service-ref="userDetailsService" />

   </security:authentication-manager>

   <bean id="userDetailsService"

      class="org.springframework.security.core.userdetails.jdbc.JdbcDaoImpl">

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

   </bean>

       上述代码中我们使用的JdbcDaoImpl是Spring Security为我们提供的UserDetailsService的实现，另外Spring Security还为我们提供了UserDetailsService另外一个实现，InMemoryDaoImpl。

其作用是从数据库中加载UserDetails信息。其中已经定义好了加载相关信息的默认脚本，这些脚本也可以通过JdbcDaoImpl的相关属性进行指定。关于JdbcDaoImpl使用方式会在讲解AuthenticationProvider的时候做一个相对详细一点的介绍。

### 1.4.1    JdbcDaoImpl

       JdbcDaoImpl允许我们从数据库来加载UserDetails，其底层使用的是Spring的JdbcTemplate进行操作，所以我们需要给其指定一个数据源。此外，我们需要通过usersByUsernameQuery属性指定通过username查询用户信息的SQL语句；通过authoritiesByUsernameQuery属性指定通过username查询用户所拥有的权限的SQL语句；如果我们通过设置JdbcDaoImpl的enableGroups为true启用了用户组权限的支持，则我们还需要通过groupAuthoritiesByUsernameQuery属性指定根据username查询用户组权限的SQL语句。当这些信息都没有指定时，将使用默认的SQL语句，默认的SQL语句如下所示。

**select** username, password, enabled **from** users **where** username=? --根据username查询用户信息

**select** username, authority **from** authorities **where** username=? --根据username查询用户权限信息

**select** g.id, g.group\_name, ga.authority **from** groups g, groups\_members gm, groups\_authorities ga **where** gm.username=? **and** g.id=ga.group\_id **and** g.id=gm.group\_id --根据username查询用户组权限

       使用默认的SQL语句进行查询时意味着我们对应的数据库中应该有对应的表和表结构，Spring Security为我们提供的默认表的创建脚本如下。

**create table** users(

      username varchar\_ignorecase(50) **not null primary key**,

      password varchar\_ignorecase(50) **not null**,

      enabled **boolean not null**);

**create table** authorities (

      username varchar\_ignorecase(50) **not null**,

      authority varchar\_ignorecase(50) **not null**,

**constraint** fk\_authorities\_users **foreign key**(username) **references** users(username));

**create unique index** ix\_auth\_username **on** authorities (username,authority);

**create table** groups (

  id bigint generated **by default as** identity(start **with** 0) **primary key**,

  group\_name varchar\_ignorecase(50) **notnull**);

**create table** group\_authorities (

  group\_id bigint **notnull**,

  authority **varchar**(50) **notnull**,

**constraint** fk\_group\_authorities\_group **foreign key**(group\_id) **references** groups(id));

**create table** group\_members (

  id bigint generated **by default as** identity(start **with** 0) **primary key**,

  username **varchar**(50) **notnull**,

  group\_id bigint **notnull**,

**constraint** fk\_group\_members\_group **foreign key**(group\_id) **references** groups(id));

       此外，使用jdbc-user-service元素时在底层Spring Security默认使用的就是JdbcDaoImpl。

   <security:authentication-manager alias="authenticationManager">

      <security:authentication-provider>

         <!-- 基于Jdbc的UserDetailsService实现，JdbcDaoImpl -->

         <security:jdbc-user-service data-source-ref="dataSource"/>

      </security:authentication-provider>

   </security:authentication-manager>

### 1.4.2    InMemoryDaoImpl

       InMemoryDaoImpl主要是测试用的，其只是简单的将用户信息保存在内存中。使用NameSpace时，使用user-service元素Spring Security底层使用的UserDetailsService就是InMemoryDaoImpl。此时，我们可以简单的使用user元素来定义一个UserDetails。

   <security:user-service>

      <security:user name="user" password="user" authorities="ROLE\_USER"/>

   </security:user-service>

       如上配置表示我们定义了一个用户user，其对应的密码为user，拥有ROLE\_USER的权限。此外，user-service还支持通过properties文件来指定用户信息，如：

   <security:user-service properties="/WEB-INF/config/users.properties"/>

       其中属性文件应遵循如下格式：

username=password,grantedAuthority[,grantedAuthority][,enabled|disabled]

       所以，对应上面的配置文件，我们的users.properties文件的内容应该如下所示：

#username=password,grantedAuthority[,grantedAuthority][,enabled|disabled]

user=user,ROLE\_USER

## 1.5     GrantedAuthority

       Authentication的getAuthorities()可以返回当前Authentication对象拥有的权限，即当前用户拥有的权限。其返回值是一个GrantedAuthority类型的数组，每一个GrantedAuthority对象代表赋予给当前用户的一种权限。GrantedAuthority是一个接口，其通常是通过UserDetailsService进行加载，然后赋予给UserDetails的。

       GrantedAuthority中只定义了一个getAuthority()方法，该方法返回一个字符串，表示对应权限的字符串表示，如果对应权限不能用字符串表示，则应当返回null。

       Spring Security针对GrantedAuthority有一个简单实现SimpleGrantedAuthority。该类只是简单的接收一个表示权限的字符串。Spring Security内部的所有AuthenticationProvider都是使用SimpleGrantedAuthority来封装Authentication对象。

# [Spring Security（03）——核心类简介](http://www.cnblogs.com/fenglan/p/5912820.html)

**目录**

1.1     Authentication

1.2     SecurityContextHolder

1.3     AuthenticationManager和AuthenticationProvider

1.3.1    认证成功后清除凭证

1.4     UserDetailsService

1.4.1    JdbcDaoImpl

1.4.2    InMemoryDaoImpl

1.5     GrantedAuthority

## 1.1     Authentication

       Authentication是一个接口，用来表示用户认证信息的，在用户登录认证之前相关信息会封装为一个Authentication具体实现类的对象，在登录认证成功之后又会生成一个信息更全面，包含用户权限等信息的Authentication对象，然后把它保存在SecurityContextHolder所持有的SecurityContext中，供后续的程序进行调用，如访问权限的鉴定等。

## 1.2     SecurityContextHolder

       SecurityContextHolder是用来保存SecurityContext的。SecurityContext中含有当前正在访问系统的用户的详细信息。默认情况下，SecurityContextHolder将使用ThreadLocal来保存SecurityContext，这也就意味着在处于同一线程中的方法中我们可以从ThreadLocal中获取到当前的SecurityContext。因为线程池的原因，如果我们每次在请求完成后都将ThreadLocal进行清除的话，那么我们把SecurityContext存放在ThreadLocal中还是比较安全的。这些工作Spring Security已经自动为我们做了，即在每一次request结束后都将清除当前线程的ThreadLocal。

       SecurityContextHolder中定义了一系列的静态方法，而这些静态方法内部逻辑基本上都是通过SecurityContextHolder持有的SecurityContextHolderStrategy来实现的，如getContext()、setContext()、clearContext()等。而默认使用的strategy就是基于ThreadLocal的ThreadLocalSecurityContextHolderStrategy。另外，Spring Security还提供了两种类型的strategy实现，GlobalSecurityContextHolderStrategy和InheritableThreadLocalSecurityContextHolderStrategy，前者表示全局使用同一个SecurityContext，如C/S结构的客户端；后者使用InheritableThreadLocal来存放SecurityContext，即子线程可以使用父线程中存放的变量。

       一般而言，我们使用默认的strategy就可以了，但是如果要改变默认的strategy，Spring Security为我们提供了两种方法，这两种方式都是通过改变strategyName来实现的。SecurityContextHolder中为三种不同类型的strategy分别命名为MODE\_THREADLOCAL、MODE\_INHERITABLETHREADLOCAL和MODE\_GLOBAL。第一种方式是通过SecurityContextHolder的静态方法setStrategyName()来指定需要使用的strategy；第二种方式是通过系统属性进行指定，其中属性名默认为“spring.security.strategy”，属性值为对应strategy的名称。

       Spring Security使用一个Authentication对象来描述当前用户的相关信息。SecurityContextHolder中持有的是当前用户的SecurityContext，而SecurityContext持有的是代表当前用户相关信息的Authentication的引用。这个Authentication对象不需要我们自己去创建，在与系统交互的过程中，Spring Security会自动为我们创建相应的Authentication对象，然后赋值给当前的SecurityContext。但是往往我们需要在程序中获取当前用户的相关信息，比如最常见的是获取当前登录用户的用户名。在程序的任何地方，通过如下方式我们可以获取到当前用户的用户名。

**public** String getCurrentUsername() {

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

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

**return** ((UserDetails) principal).getUsername();

      }

**if** (principal **instanceof** Principal) {

**return** ((Principal) principal).getName();

      }

**return** String.valueOf(principal);

   }

       通过Authentication.getPrincipal()可以获取到代表当前用户的信息，这个对象通常是UserDetails的实例。获取当前用户的用户名是一种比较常见的需求，关于上述代码其实Spring Security在Authentication中的实现类中已经为我们做了相关实现，所以获取当前用户的用户名最简单的方式应当如下。

**public** String getCurrentUsername() {

**return** SecurityContextHolder.getContext().getAuthentication().getName();

   }

       此外，调用SecurityContextHolder.getContext()获取SecurityContext时，如果对应的SecurityContext不存在，则Spring Security将为我们建立一个空的SecurityContext并进行返回。

## 1.3     AuthenticationManager和AuthenticationProvider

       AuthenticationManager是一个用来处理认证（Authentication）请求的接口。在其中只定义了一个方法authenticate()，该方法只接收一个代表认证请求的Authentication对象作为参数，如果认证成功，则会返回一个封装了当前用户权限等信息的Authentication对象进行返回。

    Authentication authenticate(Authentication authentication) **throws**AuthenticationException;

       在Spring Security中，AuthenticationManager的默认实现是ProviderManager，而且它不直接自己处理认证请求，而是委托给其所配置的AuthenticationProvider列表，然后会依次使用每一个AuthenticationProvider进行认证，如果有一个AuthenticationProvider认证后的结果不为null，则表示该AuthenticationProvider已经认证成功，之后的AuthenticationProvider将不再继续认证。然后直接以该AuthenticationProvider的认证结果作为ProviderManager的认证结果。如果所有的AuthenticationProvider的认证结果都为null，则表示认证失败，将抛出一个ProviderNotFoundException。校验认证请求最常用的方法是根据请求的用户名加载对应的UserDetails，然后比对UserDetails的密码与认证请求的密码是否一致，一致则表示认证通过。Spring Security内部的DaoAuthenticationProvider就是使用的这种方式。其内部使用UserDetailsService来负责加载UserDetails，UserDetailsService将在下节讲解。在认证成功以后会使用加载的UserDetails来封装要返回的Authentication对象，加载的UserDetails对象是包含用户权限等信息的。认证成功返回的Authentication对象将会保存在当前的SecurityContext中。

       当我们在使用NameSpace时， authentication-manager元素的使用会使Spring Security 在内部创建一个ProviderManager，然后可以通过authentication-provider元素往其中添加AuthenticationProvider。当定义authentication-provider元素时，如果没有通过ref属性指定关联哪个AuthenticationProvider，Spring Security默认就会使用DaoAuthenticationProvider。使用了NameSpace后我们就不要再声明ProviderManager了。

   <security:authentication-manager alias="authenticationManager">

      <security:authentication-provider

         user-service-ref="userDetailsService"/>

   </security:authentication-manager>

       如果我们没有使用NameSpace，那么我们就应该在ApplicationContext中声明一个ProviderManager。

### 1.3.1    认证成功后清除凭证

       默认情况下，在认证成功后ProviderManager将清除返回的Authentication中的凭证信息，如密码。所以如果你在无状态的应用中将返回的Authentication信息缓存起来了，那么以后你再利用缓存的信息去认证将会失败，因为它已经不存在密码这样的凭证信息了。所以在使用缓存的时候你应该考虑到这个问题。一种解决办法是设置ProviderManager的eraseCredentialsAfterAuthentication 属性为false，或者想办法在缓存时将凭证信息一起缓存。

## 1.4     UserDetailsService

       通过Authentication.getPrincipal()的返回类型是Object，但很多情况下其返回的其实是一个UserDetails的实例。UserDetails是Spring Security中一个核心的接口。其中定义了一些可以获取用户名、密码、权限等与认证相关的信息的方法。Spring Security内部使用的UserDetails实现类大都是内置的User类，我们如果要使用UserDetails时也可以直接使用该类。在Spring Security内部很多地方需要使用用户信息的时候基本上都是使用的UserDetails，比如在登录认证的时候。登录认证的时候Spring Security会通过UserDetailsService的loadUserByUsername()方法获取对应的UserDetails进行认证，认证通过后会将该UserDetails赋给认证通过的Authentication的principal，然后再把该Authentication存入到SecurityContext中。之后如果需要使用用户信息的时候就是通过SecurityContextHolder获取存放在SecurityContext中的Authentication的principal。

       通常我们需要在应用中获取当前用户的其它信息，如Email、电话等。这时存放在Authentication的principal中只包含有认证相关信息的UserDetails对象可能就不能满足我们的要求了。这时我们可以实现自己的UserDetails，在该实现类中我们可以定义一些获取用户其它信息的方法，这样将来我们就可以直接从当前SecurityContext的Authentication的principal中获取这些信息了。上文已经提到了UserDetails是通过UserDetailsService的loadUserByUsername()方法进行加载的。UserDetailsService也是一个接口，我们也需要实现自己的UserDetailsService来加载我们自定义的UserDetails信息。然后把它指定给AuthenticationProvider即可。如下是一个配置UserDetailsService的示例。

   <!-- 用于认证的AuthenticationManager -->

   <security:authentication-manager alias="authenticationManager">

      <security:authentication-provider

         user-service-ref="userDetailsService" />

   </security:authentication-manager>

   <bean id="userDetailsService"

      class="org.springframework.security.core.userdetails.jdbc.JdbcDaoImpl">

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

   </bean>

       上述代码中我们使用的JdbcDaoImpl是Spring Security为我们提供的UserDetailsService的实现，另外Spring Security还为我们提供了UserDetailsService另外一个实现，InMemoryDaoImpl。

其作用是从数据库中加载UserDetails信息。其中已经定义好了加载相关信息的默认脚本，这些脚本也可以通过JdbcDaoImpl的相关属性进行指定。关于JdbcDaoImpl使用方式会在讲解AuthenticationProvider的时候做一个相对详细一点的介绍。

### 1.4.1    JdbcDaoImpl

       JdbcDaoImpl允许我们从数据库来加载UserDetails，其底层使用的是Spring的JdbcTemplate进行操作，所以我们需要给其指定一个数据源。此外，我们需要通过usersByUsernameQuery属性指定通过username查询用户信息的SQL语句；通过authoritiesByUsernameQuery属性指定通过username查询用户所拥有的权限的SQL语句；如果我们通过设置JdbcDaoImpl的enableGroups为true启用了用户组权限的支持，则我们还需要通过groupAuthoritiesByUsernameQuery属性指定根据username查询用户组权限的SQL语句。当这些信息都没有指定时，将使用默认的SQL语句，默认的SQL语句如下所示。

**select** username, password, enabled **from** users **where** username=? --根据username查询用户信息

**select** username, authority **from** authorities **where** username=? --根据username查询用户权限信息

**select** g.id, g.group\_name, ga.authority **from** groups g, groups\_members gm, groups\_authorities ga **where** gm.username=? **and** g.id=ga.group\_id **and** g.id=gm.group\_id --根据username查询用户组权限

       使用默认的SQL语句进行查询时意味着我们对应的数据库中应该有对应的表和表结构，Spring Security为我们提供的默认表的创建脚本如下。

**create table** users(

      username varchar\_ignorecase(50) **not null primary key**,

      password varchar\_ignorecase(50) **not null**,

      enabled **boolean not null**);

**create table** authorities (

      username varchar\_ignorecase(50) **not null**,

      authority varchar\_ignorecase(50) **not null**,

**constraint** fk\_authorities\_users **foreign key**(username) **references** users(username));

**create unique index** ix\_auth\_username **on** authorities (username,authority);

**create table** groups (

  id bigint generated **by default as** identity(start **with** 0) **primary key**,

  group\_name varchar\_ignorecase(50) **notnull**);

**create table** group\_authorities (

  group\_id bigint **notnull**,

  authority **varchar**(50) **notnull**,

**constraint** fk\_group\_authorities\_group **foreign key**(group\_id) **references** groups(id));

**create table** group\_members (

  id bigint generated **by default as** identity(start **with** 0) **primary key**,

  username **varchar**(50) **notnull**,

  group\_id bigint **notnull**,

**constraint** fk\_group\_members\_group **foreign key**(group\_id) **references** groups(id));

       此外，使用jdbc-user-service元素时在底层Spring Security默认使用的就是JdbcDaoImpl。

   <security:authentication-manager alias="authenticationManager">

      <security:authentication-provider>

         <!-- 基于Jdbc的UserDetailsService实现，JdbcDaoImpl -->

         <security:jdbc-user-service data-source-ref="dataSource"/>

      </security:authentication-provider>

   </security:authentication-manager>

### 1.4.2    InMemoryDaoImpl

       InMemoryDaoImpl主要是测试用的，其只是简单的将用户信息保存在内存中。使用NameSpace时，使用user-service元素Spring Security底层使用的UserDetailsService就是InMemoryDaoImpl。此时，我们可以简单的使用user元素来定义一个UserDetails。

   <security:user-service>

      <security:user name="user" password="user" authorities="ROLE\_USER"/>

   </security:user-service>

       如上配置表示我们定义了一个用户user，其对应的密码为user，拥有ROLE\_USER的权限。此外，user-service还支持通过properties文件来指定用户信息，如：

   <security:user-service properties="/WEB-INF/config/users.properties"/>

       其中属性文件应遵循如下格式：

username=password,grantedAuthority[,grantedAuthority][,enabled|disabled]

       所以，对应上面的配置文件，我们的users.properties文件的内容应该如下所示：

#username=password,grantedAuthority[,grantedAuthority][,enabled|disabled]

user=user,ROLE\_USER

## 1.5     GrantedAuthority

       Authentication的getAuthorities()可以返回当前Authentication对象拥有的权限，即当前用户拥有的权限。其返回值是一个GrantedAuthority类型的数组，每一个GrantedAuthority对象代表赋予给当前用户的一种权限。GrantedAuthority是一个接口，其通常是通过UserDetailsService进行加载，然后赋予给UserDetails的。

       GrantedAuthority中只定义了一个getAuthority()方法，该方法返回一个字符串，表示对应权限的字符串表示，如果对应权限不能用字符串表示，则应当返回null。

       Spring Security针对GrantedAuthority有一个简单实现SimpleGrantedAuthority。该类只是简单的接收一个表示权限的字符串。Spring Security内部的所有AuthenticationProvider都是使用SimpleGrantedAuthority来封装Authentication对象。

**目录**

1.1     认证过程

1.2     Web应用的认证过程

1.2.1    ExceptionTranslationFilter

1.2.2    在request之间共享SecurityContext

## 1.1     认证过程

       1、用户使用用户名和密码进行登录。

       2、Spring Security将获取到的用户名和密码封装成一个实现了Authentication接口的UsernamePasswordAuthenticationToken。

       3、将上述产生的token对象传递给AuthenticationManager进行登录认证。

       4、AuthenticationManager认证成功后将会返回一个封装了用户权限等信息的Authentication对象。

       5、通过调用SecurityContextHolder.getContext().setAuthentication(...)将AuthenticationManager返回的Authentication对象赋予给当前的SecurityContext。

       上述介绍的就是Spring Security的认证过程。在认证成功后，用户就可以继续操作去访问其它受保护的资源了，但是在访问的时候将会使用保存在SecurityContext中的Authentication对象进行相关的权限鉴定。

## 1.2     Web应用的认证过程

       如果用户直接访问登录页面，那么认证过程跟上节描述的基本一致，只是在认证完成后将跳转到指定的成功页面，默认是应用的根路径。如果用户直接访问一个受保护的资源，那么认证过程将如下：

       1、引导用户进行登录，通常是重定向到一个基于form表单进行登录的页面，具体视配置而定。

       2、用户输入用户名和密码后请求认证，后台还是会像上节描述的那样获取用户名和密码封装成一个UsernamePasswordAuthenticationToken对象，然后把它传递给AuthenticationManager进行认证。

       3、如果认证失败将继续执行步骤1，如果认证成功则会保存返回的Authentication到SecurityContext，然后默认会将用户重定向到之前访问的页面。

       4、用户登录认证成功后再次访问之前受保护的资源时就会对用户进行权限鉴定，如不存在对应的访问权限，则会返回403错误码。

       在上述步骤中将有很多不同的类参与，但其中主要的参与者是ExceptionTranslationFilter。

### 1.2.1   ExceptionTranslationFilter

       ExceptionTranslationFilter是用来处理来自AbstractSecurityInterceptor抛出的AuthenticationException和AccessDeniedException的。AbstractSecurityInterceptor是Spring Security用于拦截请求进行权限鉴定的，其拥有两个具体的子类，拦截方法调用的MethodSecurityInterceptor和拦截URL请求的FilterSecurityInterceptor。当ExceptionTranslationFilter捕获到的是AuthenticationException时将调用AuthenticationEntryPoint引导用户进行登录；如果捕获的是AccessDeniedException，但是用户还没有通过认证，则调用AuthenticationEntryPoint引导用户进行登录认证，否则将返回一个表示不存在对应权限的403错误码。

### 1.2.2  在request之间共享SecurityContext

       可能你早就有这么一个疑问了，既然SecurityContext是存放在ThreadLocal中的，而且在每次权限鉴定的时候都是从ThreadLocal中获取SecurityContext中对应的Authentication所拥有的权限，并且不同的request是不同的线程，为什么每次都可以从ThreadLocal中获取到当前用户对应的SecurityContext呢？在Web应用中这是通过SecurityContextPersistentFilter实现的，默认情况下其会在每次请求开始的时候从session中获取SecurityContext，然后把它设置给SecurityContextHolder，在请求结束后又会将SecurityContextHolder所持有的SecurityContext保存在session中，并且清除SecurityContextHolder所持有的SecurityContext。这样当我们第一次访问系统的时候，SecurityContextHolder所持有的SecurityContext肯定是空的，待我们登录成功后，SecurityContextHolder所持有的SecurityContext就不是空的了，且包含有认证成功的Authentication对象，待请求结束后我们就会将SecurityContext存在session中，等到下次请求的时候就可以从session中获取到该SecurityContext并把它赋予给SecurityContextHolder了，由于SecurityContextHolder已经持有认证过的Authentication对象了，所以下次访问的时候也就不再需要进行登录认证了。

Spring Security支持将展现给终端用户看的异常信息本地化，这些信息包括认证失败、访问被拒绝等。而对于展现给开发者看的异常信息和日志信息（如配置错误）则是不能够进行本地化的，它们是以英文硬编码在Spring Security的代码中的。在Spring-Security-core-xxx.jar包的org.springframework.security包下拥有一个以英文异常信息为基础的messages.properties文件，以及其它一些常用语言的异常信息对应的文件，如messages\_zh\_CN.properties文件。那么对于用户而言所需要做的就是在自己的ApplicationContext中定义如下这样一个bean。

   <bean id="messageSource"

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

      <property name="basename"

         value="classpath:org/springframework/security/messages" />

   </bean>

       如果要自己定制messages.properties文件，或者需要新增本地化支持文件，则可以copy Spring Security提供的默认messages.properties文件，将其中的内容进行修改后再注入到上述bean中。比如我要定制一些中文的提示信息，那么我可以在copy一个messages.properties文件到类路径的“com/xxx”下，然后将其重命名为messages\_zh\_CN.properties，并修改其中的提示信息。然后通过basenames属性注入到上述bean中，如：

   <bean id="messageSource"

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

      <property name="basenames">

         <array>

            <!-- 将自定义的放在Spring Security内置的之前 -->

            <value>classpath:com/xxx/messages</value>

            <value>classpath:org/springframework/security/messages</value>

         </array>

      </property>

   </bean>

       有一点需要注意的是将自定义的messages.properties文件路径定义在Spring Security内置的message.properties路径定义之前。

# [Spring Security（06）——AuthenticationProvider](http://www.cnblogs.com/fenglan/p/5912979.html)

**目录**

1.1     用户信息从数据库获取

1.1.1    使用jdbc-user-service获取

1.1.2    直接使用JdbcDaoImpl

1.2     PasswordEncoder

1.2.1    使用内置的PasswordEncoder

1.2.2    使用自定义的PasswordEncoder

       认证是由AuthenticationManager来管理的，但是真正进行认证的是AuthenticationManager中定义的AuthenticationProvider。AuthenticationManager中可以定义有多个AuthenticationProvider。当我们使用authentication-provider元素来定义一个AuthenticationProvider时，如果没有指定对应关联的AuthenticationProvider对象，Spring Security默认会使用DaoAuthenticationProvider。DaoAuthenticationProvider在进行认证的时候需要一个UserDetailsService来获取用户的信息UserDetails，其中包括用户名、密码和所拥有的权限等。所以如果我们需要改变认证的方式，我们可以实现自己的AuthenticationProvider；如果需要改变认证的用户信息来源，我们可以实现UserDetailsService。

       实现了自己的AuthenticationProvider之后，我们可以在配置文件中这样配置来使用我们自己的AuthenticationProvider。其中myAuthenticationProvider就是我们自己的AuthenticationProvider实现类对应的bean。

   <security:authentication-manager>

      <security:authentication-provider ref="myAuthenticationProvider"/>

   </security:authentication-manager>

       实现了自己的UserDetailsService之后，我们可以在配置文件中这样配置来使用我们自己的UserDetailsService。其中的myUserDetailsService就是我们自己的UserDetailsService实现类对应的bean。

   <security:authentication-manager>

      <security:authentication-provider user-service-ref="myUserDetailsService"/>

   </security:authentication-manager>

## 1.1     用户信息从数据库获取

       通常我们的用户信息都不会向第一节示例中那样简单的写在配置文件中，而是从其它存储位置获取，比如数据库。根据之前的介绍我们知道用户信息是通过UserDetailsService获取的，要从数据库获取用户信息，我们就需要实现自己的UserDetailsService。幸运的是像这种常用的方式Spring Security已经为我们做了实现了。

### 1.1.1   使用jdbc-user-service获取

       在Spring Security的命名空间中在authentication-provider下定义了一个jdbc-user-service元素，通过该元素我们可以定义一个从数据库获取UserDetails的UserDetailsService。jdbc-user-service需要接收一个数据源的引用。

   <security:authentication-manager>

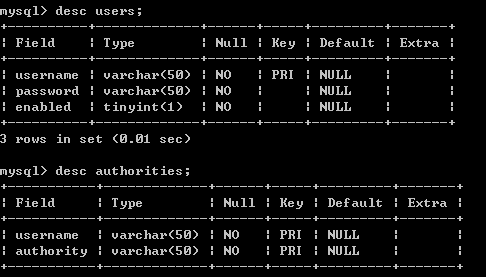
      <security:authentication-provider>

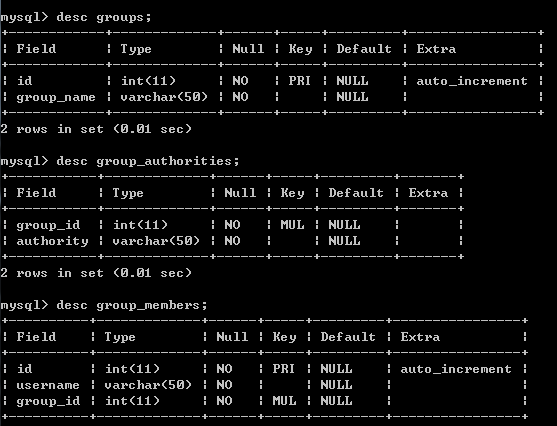
         <security:jdbc-user-service data-source-ref="dataSource"/>

      </security:authentication-provider>

   </security:authentication-manager>

       上述配置中dataSource是对应数据源配置的bean引用。使用此种方式需要我们的数据库拥有如下表和表结构。



       这是因为默认情况下jdbc-user-service将使用SQL语句“select username, password, enabled from users where username = ?”来获取用户信息；使用SQL语句“select username, authority from authorities where username = ?”来获取用户对应的权限；使用SQL语句“select g.id, g.group\_name, ga.authority from groups g, group\_members gm, group\_authorities ga where gm.username = ? and g.id = ga.group\_id and g.id = gm.group\_id”来获取用户所属组的权限。需要注意的是jdbc-user-service定义是不支持用户组权限的，所以使用jdbc-user-service时用户组相关表也是可以不定义的。如果需要使用用户组权限请使用JdbcDaoImpl，这个在后文后讲到。

       当然这只是默认配置及默认的表结构。如果我们的表名或者表结构跟Spring Security默认的不一样，我们可以通过以下几个属性来定义我们自己查询用户信息、用户权限和用户组权限的SQL。

|  |  |
| --- | --- |
| **属性名** | **说明** |
| users-by-username-query | 指定查询用户信息的SQL |
| authorities-by-username-query | 指定查询用户权限的SQL |
| group-authorities-by-username-query | 指定查询用户组权限的SQL |

       假设我们的用户表是t\_user，而不是默认的users，则我们可以通过属性users-by-username-query来指定查询用户信息的时候是从用户表t\_user查询。

   <security:authentication-manager>

      <security:authentication-provider>

         <security:jdbc-user-service

            data-source-ref="dataSource"

            users-by-username-query="select username, password, enabled from t\_user where username = ?" />

      </security:authentication-provider>

   </security:authentication-manager>

**role-prefix属性**

       jdbc-user-service还有一个属性role-prefix可以用来指定角色的前缀。这是什么意思呢？这表示我们从库里面查询出来的权限需要加上什么样的前缀。举个例子，假设我们库里面存放的权限都是“USER”，而我们指定了某个URL的访问权限access=”ROLE\_USER”，显然这是不匹配的，Spring Security不会给我们放行，通过指定jdbc-user-service的role-prefix=”ROLE\_”之后就会满足了。当role-prefix的值为“none”时表示没有前缀，当然默认也是没有的。

### 1.1.2   直接使用JdbcDaoImpl

       JdbcDaoImpl是UserDetailsService的一个实现。其用法和jdbc-user-service类似，只是我们需要把它定义为一个bean，然后通过authentication-provider的user-service-ref进行引用。

   <security:authentication-manager>

      <security:authentication-provider user-service-ref="userDetailsService"/>

   </security:authentication-manager>

   <bean id="userDetailsService"class="org.springframework.security.core.userdetails.jdbc.JdbcDaoImpl">

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

   </bean>

       如你所见，JdbcDaoImpl同样需要一个dataSource的引用。如果就是上面这样配置的话我们数据库表结构也需要是标准的表结构。当然，如果我们的表结构和标准的不一样，可以通过usersByUsernameQuery、authoritiesByUsernameQuery和groupAuthoritiesByUsernameQuery属性来指定对应的查询SQL。

**用户权限和用户组权限**

       JdbcDaoImpl使用enableAuthorities和enableGroups两个属性来控制权限的启用。默认启用的是enableAuthorities，即用户权限，而enableGroups默认是不启用的。如果需要启用用户组权限，需要指定enableGroups属性值为true。当然这两种权限是可以同时启用的。需要注意的是使用jdbc-user-service定义的UserDetailsService是不支持用户组权限的，如果需要支持用户组权限的话需要我们使用JdbcDaoImpl。

   <security:authentication-manager>

      <security:authentication-provider user-service-ref="userDetailsService"/>

   </security:authentication-manager>

   <bean id="userDetailsService"class="org.springframework.security.core.userdetails.jdbc.JdbcDaoImpl">

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

      <property name="enableGroups" value="true"/>

   </bean>

## 1.2     PasswordEncoder

### 1.2.1   使用内置的PasswordEncoder

       通常我们保存的密码都不会像之前介绍的那样，保存的明文，而是加密之后的结果。为此，我们的AuthenticationProvider在做认证时也需要将传递的明文密码使用对应的算法加密后再与保存好的密码做比较。Spring Security对这方面也有支持。通过在authentication-provider下定义一个password-encoder我们可以定义当前AuthenticationProvider需要在进行认证时需要使用的password-encoder。password-encoder是一个PasswordEncoder的实例，我们可以直接使用它，如：

   <security:authentication-manager>

      <security:authentication-provider user-service-ref="userDetailsService">

         <security:password-encoder hash="md5"/>

      </security:authentication-provider>

   </security:authentication-manager>

       其属性hash表示我们将用来进行加密的哈希算法，系统已经为我们实现的有plaintext、sha、sha-256、md4、md5、{sha}和{ssha}。它们对应的PasswordEncoder实现类如下：

|  |  |
| --- | --- |
| **加密算法** | **PasswordEncoder实现类** |
| plaintext | PlaintextPasswordEncoder |
| sha | ShaPasswordEncoder |
| sha-256 | ShaPasswordEncoder，使用时new ShaPasswordEncoder(256) |
| md4 | Md4PasswordEncoder |
| md5 | Md5PasswordEncoder |
| {sha} | LdapShaPasswordEncoder |
| {ssha} | LdapShaPasswordEncoder |

**使用BASE64编码加密后的密码**

       此外，使用password-encoder时我们还可以指定一个属性base64，表示是否需要对加密后的密码使用BASE64进行编码，默认是false。如果需要则设为true。

<security:password-encoder hash="md5" base64="true"/>

**加密时使用salt**

       加密时使用salt也是很常见的需求，Spring Security内置的password-encoder也对它有支持。通过password-encoder元素下的子元素salt-source，我们可以指定当前PasswordEncoder需要使用的salt。这个salt可以是一个常量，也可以是当前UserDetails的某一个属性，还可以通过实现SaltSource接口实现自己的获取salt的逻辑，SaltSource中只定义了如下一个方法。

**public** Object getSalt(UserDetails user);

       下面来看几个使用salt-source的示例。

       （1）下面的配置将使用常量“abc”作为salt。

   <security:authentication-manager>

      <security:authentication-provider user-service-ref="userDetailsService">

         <security:password-encoder hash="md5" base64="true">

            <security:salt-source system-wide="abc"/>

         </security:password-encoder>

      </security:authentication-provider>

   </security:authentication-manager>

       （2）下面的配置将使用UserDetails的username作为salt。

   <security:authentication-manager>

      <security:authentication-provider user-service-ref="userDetailsService">

         <security:password-encoder hash="md5" base64="true">

            <security:salt-source user-property="username"/>

         </security:password-encoder>

      </security:authentication-provider>

   </security:authentication-manager>

       （3）下面的配置将使用自己实现的SaltSource获取salt。其中mySaltSource就是SaltSource实现类对应的bean的引用。

   <security:authentication-manager>

      <security:authentication-provider user-service-ref="userDetailsService">

         <security:password-encoder hash="md5" base64="true">

            <security:salt-source ref="mySaltSource"/>

         </security:password-encoder>

      </security:authentication-provider>

   </security:authentication-manager>

       需要注意的是AuthenticationProvider进行认证时所使用的PasswordEncoder，包括它们的算法和规则都应当与我们保存用户密码时是一致的。也就是说如果AuthenticationProvider使用Md5PasswordEncoder进行认证，我们在保存用户密码时也需要使用Md5PasswordEncoder；如果AuthenticationProvider在认证时使用了username作为salt，那么我们在保存用户密码时也需要使用username作为salt。如：

   Md5PasswordEncoder encoder = **new** Md5PasswordEncoder();

   encoder.setEncodeHashAsBase64(**true**);

   System.out.println(encoder.encodePassword("user", "user"));

### 1.2.2   使用自定义的PasswordEncoder

       除了通过password-encoder使用Spring Security已经为我们实现了的PasswordEncoder之外，我们也可以实现自己的PasswordEncoder，然后通过password-encoder的ref属性关联到我们自己实现的PasswordEncoder对应的bean对象。

   <security:authentication-manager>

      <security:authentication-provider user-service-ref="userDetailsService">

         <security:password-encoder ref="passwordEncoder"/>

      </security:authentication-provider>

   </security:authentication-manager>

   <bean id="passwordEncoder" class="com.xxx.MyPasswordEncoder"/>

       在Spring Security内部定义有两种类型的PasswordEncoder，分别是org.springframework.security.authentication.encoding.PasswordEncoder和org.springframework.security.crypto.password.PasswordEncoder。直接通过password-encoder元素的hash属性指定使用内置的PasswordEncoder都是基于org.springframework.security.authentication.encoding.PasswordEncoder的实现，然而它现在已经被废弃了，Spring Security推荐我们使用org.springframework.security.crypto.password.PasswordEncoder，它的设计理念是为了使用随机生成的salt。关于后者Spring Security也已经提供了几个实现类，更多信息请查看Spring Security的API文档。我们在通过password-encoder使用自定义的PasswordEncoder时两种PasswordEncoder的实现类都是支持的。

# [Spring Security（07）——缓存UserDetails](http://www.cnblogs.com/fenglan/p/5913015.html)

 Spring Security提供了一个实现了可以缓存UserDetails的UserDetailsService实现类，CachingUserDetailsService。该类的构造接收一个用于真正加载UserDetails的UserDetailsService实现类。当需要加载UserDetails时，其首先会从缓存中获取，如果缓存中没有对应的UserDetails存在，则使用持有的UserDetailsService实现类进行加载，然后将加载后的结果存放在缓存中。UserDetails与缓存的交互是通过UserCache接口来实现的。CachingUserDetailsService默认拥有UserCache的一个空实现引用，NullUserCache。以下是CachingUserDetailsService的类定义。

**public class** CachingUserDetailsService **implements** UserDetailsService {

**private** UserCache userCache = **new** NullUserCache();

**private final** UserDetailsService delegate;

    CachingUserDetailsService(UserDetailsService delegate) {

**this**.delegate = delegate;

    }

**public** UserCache getUserCache() {

**return**userCache;

    }

**public void** setUserCache(UserCache userCache) {

**this**.userCache = userCache;

    }

**public** UserDetails loadUserByUsername(String username) {

        UserDetails user = userCache.getUserFromCache(username);

**if** (user == **null**) {

            user = delegate.loadUserByUsername(username);

        }

        Assert.notNull(user, "UserDetailsService " + delegate + " returned null for username " + username + ". " +

                "This is an interface contract violation");

        userCache.putUserInCache(user);

**return** user;

    }

}

       我们可以看到当缓存中不存在对应的UserDetails时将使用引用的UserDetailsService类型的delegate进行加载。加载后再把它存放到Cache中并进行返回。除了NullUserCache之外，Spring Security还为我们提供了一个基于Ehcache的UserCache实现类，EhCacheBasedUserCache，其源码如下所示。

**public class** EhCacheBasedUserCache **implements** UserCache, InitializingBean {

**private static final** Log logger = LogFactory.getLog(EhCacheBasedUserCache.**class**);

**private** Ehcache cache;

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

        Assert.notNull(cache, "cache mandatory");

    }

**public** Ehcache getCache() {

**return**cache;

    }

**public** UserDetails getUserFromCache(String username) {

        Element element = cache.get(username);

**if** (logger.isDebugEnabled()) {

            logger.debug("Cache hit: " + (element != **null**) + "; username: " + username);

        }

**if** (element == **null**) {

**returnnull**;

        } **else** {

**return** (UserDetails) element.getValue();

        }

    }

**public void** putUserInCache(UserDetails user) {

        Element element = **new** Element(user.getUsername(), user);

**if** (logger.isDebugEnabled()) {

            logger.debug("Cache put: " + element.getKey());

        }

        cache.put(element);

    }

**public void** removeUserFromCache(UserDetails user) {

**if** (logger.isDebugEnabled()) {

            logger.debug("Cache remove: " + user.getUsername());

        }

**this**.removeUserFromCache(user.getUsername());

    }

**public void** removeUserFromCache(String username) {

        cache.remove(username);

    }

**public void** setCache(Ehcache cache) {

**this**.cache = cache;

    }

}

       从上述源码我们可以看到EhCacheBasedUserCache所引用的Ehcache是空的，所以，当我们需要对UserDetails进行缓存时，我们只需要定义一个Ehcache实例，然后把它注入给EhCacheBasedUserCache就可以了。接下来我们来看一下定义一个支持缓存UserDetails的CachingUserDetailsService的示例。

   <security:authentication-manager alias="authenticationManager">

      <!-- 使用可以缓存UserDetails的CachingUserDetailsService -->

      <security:authentication-provider

         user-service-ref="cachingUserDetailsService" />

   </security:authentication-manager>

   <!-- 可以缓存UserDetails的UserDetailsService -->

   <bean id="cachingUserDetailsService"class="org.springframework.security.config.authentication.CachingUserDetailsService">

      <!-- 真正加载UserDetails的UserDetailsService -->

      <constructor-arg ref="userDetailsService"/>

      <!-- 缓存UserDetails的UserCache -->

      <property name="userCache">

         <beanclass="org.springframework.security.core.userdetails.cache.EhCacheBasedUserCache">

            <!-- 用于真正缓存的Ehcache对象 -->

            <property name="cache" ref="ehcache4UserDetails"/>

         </bean>

      </property>

   </bean>

   <!-- 将使用默认的CacheManager创建一个名为ehcache4UserDetails的Ehcache对象 -->

   <bean id="ehcache4UserDetails"class="org.springframework.cache.ehcache.EhCacheFactoryBean"/>

   <!-- 从数据库加载UserDetails的UserDetailsService -->

   <bean id="userDetailsService"

      class="org.springframework.security.core.userdetails.jdbc.JdbcDaoImpl">

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

   </bean>

       在上面的配置中，我们通过EhcacheFactoryBean定义的Ehcache bean对象采用的是默认配置，其将使用默认的CacheManager，即直接通过CacheManager.getInstance()获取当前已经存在的CacheManager对象，如不存在则使用默认配置自动创建一个，当然这可以通过cacheManager属性指定我们需要使用的CacheManager，CacheManager可以通过EhCacheManagerFactoryBean进行定义。此外，如果没有指定对应缓存的名称，默认将使用beanName，在上述配置中即为ehcache4UserDetails，可以通过cacheName属性进行指定。此外，缓存的配置信息也都是使用的默认的。更多关于Spring使用Ehcache的信息可以参考我的另一篇文章《[Spring使用Cache](http://elim.iteye.com/blog/2123030)》。

# [Spring Security（08）——intercept-url配置](http://www.cnblogs.com/fenglan/p/5913043.html)

**目录**

1.1     指定拦截的url

1.2     指定访问权限

1.3     指定访问协议

1.4     指定请求方法

## 1.1    指定拦截的url

       通过pattern指定当前intercept-url定义应当作用于哪些url。

<security:intercept-url pattern="/\*\*" access="ROLE\_USER"/>

## 1.2     指定访问权限

       可以通过access属性来指定intercept-url对应URL访问所应当具有的权限。access的值是一个字符串，其可以直接是一个权限的定义，也可以是一个表达式。常用的类型有简单的角色名称定义，多个名称之间用逗号分隔，如：

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

       在上述配置中就表示secure路径下的所有URL请求都应当具有ROLE\_USER或ROLE\_ADMIN权限。当access的值是以“ROLE\_”开头的则将会交由RoleVoter进行处理。

       此外，其还可以是一个表达式，上述配置如果使用表达式来表示的话则应该是如下这个样子。

   <security:http use-expressions="true">

      <security:form-login />

      <security:logout />

      <security:intercept-url pattern="/secure/\*\*"access="hasAnyRole('ROLE\_USER','ROLE\_ADMIN')"/>

   </security:http>

       或者是使用hasRole()表达式，然后中间以or连接，如：

   <security:intercept-url pattern="/secure/\*\*" access="hasRole('ROLE\_USER') or hasRole('ROLE\_ADMIN')"/>

       需要注意的是使用表达式时需要指定http元素的use-expressions=”true”。更多关于使用表达式的内容将在后文介绍。当intercept-url的access属性使用表达式时默认将使用WebExpressionVoter进行处理。

       此外，还可以指定三个比较特殊的属性值，默认情况下将使用AuthenticatedVoter来处理它们。IS\_AUTHENTICATED\_ANONYMOUSLY表示用户不需要登录就可以访问；IS\_AUTHENTICATED\_REMEMBERED表示用户需要是通过Remember-Me功能进行自动登录的才能访问；IS\_AUTHENTICATED\_FULLY表示用户的认证类型应该是除前两者以外的，也就是用户需要是通过登录入口进行登录认证的才能访问。如我们通常会将登录地址设置为IS\_AUTHENTICATED\_ANONYMOUSLY。

   <security:http>

      <security:form-login login-page="/login.jsp"/>

      <!-- 登录页面可以匿名访问 -->

      <security:intercept-url pattern="/login.jsp\*"access="IS\_AUTHENTICATED\_ANONYMOUSLY"/>

      <security:intercept-url pattern="/\*\*" access="ROLE\_USER"/>

   </security:http>

## 1.3     指定访问协议

       如果你的应用同时支持Http和Https访问，且要求某些URL只能通过Https访问，这个需求可以通过指定intercept-url的requires-channel属性来指定。requires-channel支持三个值：http、https和any。any表示http和https都可以访问。

   <security:http auto-config="true">

      <security:form-login/>

      <!-- 只能通过https访问 -->

      <security:intercept-url pattern="/admin/\*\*" access="ROLE\_ADMIN" requires-channel="https"/>

      <!-- 只能通过http访问 -->

      <security:intercept-url pattern="/\*\*" access="ROLE\_USER" requires-channel="http"/>

   </security:http>

       需要注意的是当试图使用http请求限制了只能通过https访问的资源时会自动跳转到对应的https通道重新请求。如果所使用的http或者https协议不是监听在标准的端口上（http默认是80，https默认是443），则需要我们通过port-mapping元素定义好它们的对应关系。

   <security:http auto-config="true">

      <security:form-login/>

      <!-- 只能通过https访问 -->

      <security:intercept-url pattern="/admin/\*\*" access="ROLE\_ADMIN" requires-channel="https"/>

      <!-- 只能通过http访问 -->

      <security:intercept-url pattern="/\*\*" access="ROLE\_USER" requires-channel="http"/>

      <security:port-mappings>

         <security:port-mapping http="8899" https="9988"/>

      </security:port-mappings>

   </security:http>

## 1.4     指定请求方法

       通常我们都会要求某些URL只能通过POST请求，某些URL只能通过GET请求。这些限制Spring Security也已经为我们实现了，通过指定intercept-url的method属性可以限制当前intercept-url适用的请求方式，默认为所有的方式都可以。

   <security:http auto-config="true">

      <security:form-login/>

      <!-- 只能通过POST访问 -->

      <security:intercept-url pattern="/post/\*\*" method="POST"/>

      <!-- 只能通过GET访问 -->

      <security:intercept-url pattern="/\*\*" access="ROLE\_USER" method="GET"/>

   </security:http>

       method的可选值有GET、POST、DELETE、PUT、HEAD、OPTIONS和TRACE。

# [Spring Security（09）——Filter](http://www.cnblogs.com/fenglan/p/5913248.html)

**目录**

1.1     Filter顺序

1.2     添加Filter到FilterChain

1.3     DelegatingFilterProxy

1.4     FilterChainProxy

1.5     Spring Security定义好的核心Filter

1.5.1    FilterSecurityInterceptor

1.5.2    ExceptionTranslationFilter

1.5.3    SecurityContextPersistenceFilter

1.5.4    UsernamePasswordAuthenticationFilter

       Spring Security的底层是通过一系列的Filter来管理的，每个Filter都有其自身的功能，而且各个Filter在功能上还有关联关系，所以它们的顺序也是非常重要的。

## 1.1     Filter顺序

       Spring Security已经定义了一些Filter，不管实际应用中你用到了哪些，它们应当保持如下顺序。

       （1）ChannelProcessingFilter，如果你访问的channel错了，那首先就会在channel之间进行跳转，如http变为https。

       （2）SecurityContextPersistenceFilter，这样的话在一开始进行request的时候就可以在SecurityContextHolder中建立一个SecurityContext，然后在请求结束的时候，任何对SecurityContext的改变都可以被copy到HttpSession。

       （3）ConcurrentSessionFilter，因为它需要使用SecurityContextHolder的功能，而且更新对应session的最后更新时间，以及通过SessionRegistry获取当前的SessionInformation以检查当前的session是否已经过期，过期则会调用LogoutHandler。

       （4）认证处理机制，如UsernamePasswordAuthenticationFilter，CasAuthenticationFilter，BasicAuthenticationFilter等，以至于SecurityContextHolder可以被更新为包含一个有效的Authentication请求。

       （5）SecurityContextHolderAwareRequestFilter，它将会把HttpServletRequest封装成一个继承自HttpServletRequestWrapper的SecurityContextHolderAwareRequestWrapper，同时使用SecurityContext实现了HttpServletRequest中与安全相关的方法。

       （6）JaasApiIntegrationFilter，如果SecurityContextHolder中拥有的Authentication是一个JaasAuthenticationToken，那么该Filter将使用包含在JaasAuthenticationToken中的Subject继续执行FilterChain。

       （7）RememberMeAuthenticationFilter，如果之前的认证处理机制没有更新SecurityContextHolder，并且用户请求包含了一个Remember-Me对应的cookie，那么一个对应的Authentication将会设给SecurityContextHolder。

       （8）AnonymousAuthenticationFilter，如果之前的认证机制都没有更新SecurityContextHolder拥有的Authentication，那么一个AnonymousAuthenticationToken将会设给SecurityContextHolder。

       （9）ExceptionTransactionFilter，用于处理在FilterChain范围内抛出的AccessDeniedException和AuthenticationException，并把它们转换为对应的Http错误码返回或者对应的页面。

       （10）FilterSecurityInterceptor，保护Web URI，并且在访问被拒绝时抛出异常。

## 1.2     添加Filter到FilterChain

       当我们在使用NameSpace时，Spring Security是会自动为我们建立对应的FilterChain以及其中的Filter。但有时我们可能需要添加我们自己的Filter到FilterChain，又或者是因为某些特性需要自己显示的定义Spring Security已经为我们提供好的Filter，然后再把它们添加到FilterChain。使用NameSpace时添加Filter到FilterChain是通过http元素下的custom-filter元素来定义的。定义custom-filter时需要我们通过ref属性指定其对应关联的是哪个Filter，此外还需要通过position、before或者after指定该Filter放置的位置。诚如在上一节《Filter顺序》中所提到的那样，Spring Security对FilterChain中Filter顺序是有严格的规定的。Spring Security对那些内置的Filter都指定了一个别名，同时指定了它们的位置。我们在定义custom-filter的position、before和after时使用的值就是对应着这些别名所处的位置。如position=”CAS\_FILTER”就表示将定义的Filter放在CAS\_FILTER对应的那个位置，before=”CAS\_FILTER”就表示将定义的Filter放在CAS\_FILTER之前，after=”CAS\_FILTER”就表示将定义的Filter放在CAS\_FILTER之后。此外还有两个特殊的位置可以指定，FIRST和LAST，分别对应第一个和最后一个Filter，如你想把定义好的Filter放在最后，则可以使用after=”LAST”。

       接下来我们来看一下Spring Security给我们定义好的FilterChain中Filter对应的位置顺序、它们的别名以及将触发自动添加到FilterChain的元素或属性定义。下面的定义是按顺序的。

|  |  |  |
| --- | --- | --- |
| **别名** | **Filter类** | **对应元素或属性** |
| CHANNEL\_FILTER | ChannelProcessingFilter | http/intercept-url@requires-channel |
| SECURITY\_CONTEXT\_FILTER | SecurityContextPersistenceFilter | http |
| CONCURRENT\_SESSION\_FILTER | ConcurrentSessionFilter | http/session-management/concurrency-control |
| LOGOUT\_FILTER | LogoutFilter | http/logout |
| X509\_FILTER | X509AuthenticationFilter | http/x509 |
| PRE\_AUTH\_FILTER | AstractPreAuthenticatedProcessingFilter 的子类 | 无 |
| CAS\_FILTER | CasAuthenticationFilter | 无 |
| FORM\_LOGIN\_FILTER | UsernamePasswordAuthenticationFilter | http/form-login |
| BASIC\_AUTH\_FILTER | BasicAuthenticationFilter | http/http-basic |
| SERVLET\_API\_SUPPORT\_FILTER | SecurityContextHolderAwareRequestFilter | http@servlet-api-provision |
| JAAS\_API\_SUPPORT\_FILTER | JaasApiIntegrationFilter | http@jaas-api-provision |
| REMEMBER\_ME\_FILTER | RememberMeAuthenticationFilter | http/remember-me |
| ANONYMOUS\_FILTER | AnonymousAuthenticationFilter | http/anonymous |
| SESSION\_MANAGEMENT\_FILTER | SessionManagementFilter | http/session-management |
| EXCEPTION\_TRANSLATION\_FILTER | ExceptionTranslationFilter | http |
| FILTER\_SECURITY\_INTERCEPTOR | FilterSecurityInterceptor | http |
| SWITCH\_USER\_FILTER | SwitchUserFilter | 无 |

## 1.3     DelegatingFilterProxy

       可能你会觉得奇怪，我们在web应用中使用Spring Security时只在web.xml文件中定义了如下这样一个Filter，为什么你会说是一系列的Filter呢？

   <filter>

      <filter-name>springSecurityFilterChain</filter-name>

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

   </filter>

   <filter-mapping>

      <filter-name>springSecurityFilterChain</filter-name>

      <url-pattern>/\*</url-pattern>

   </filter-mapping>

       而且如果你不在web.xml文件声明要使用的Filter，那么Servlet容器将不会发现它们，它们又怎么发生作用呢？这就是上述配置中DelegatingFilterProxy的作用了。

       DelegatingFilterProxy是Spring中定义的一个Filter实现类，其作用是代理真正的Filter实现类，也就是说在调用DelegatingFilterProxy的doFilter()方法时实际上调用的是其代理Filter的doFilter()方法。其代理Filter必须是一个Spring bean对象，所以使用DelegatingFilterProxy的好处就是其代理Filter类可以使用Spring的依赖注入机制方便自由的使用ApplicationContext中的bean。那么DelegatingFilterProxy如何知道其所代理的Filter是哪个呢？这是通过其自身的一个叫targetBeanName的属性来确定的，通过该名称，DelegatingFilterProxy可以从WebApplicationContext中获取指定的bean作为代理对象。该属性可以通过在web.xml中定义DelegatingFilterProxy时通过init-param来指定，如果未指定的话将默认取其在web.xml中声明时定义的名称。

   <filter>

      <filter-name>springSecurityFilterChain</filter-name>

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

   </filter>

   <filter-mapping>

      <filter-name>springSecurityFilterChain</filter-name>

      <url-pattern>/\*</url-pattern>

   </filter-mapping>

       在上述配置中，DelegatingFilterProxy代理的就是名为SpringSecurityFilterChain的Filter。

       需要注意的是被代理的Filter的初始化方法init()和销毁方法destroy()默认是不会被执行的。通过设置DelegatingFilterProxy的targetFilterLifecycle属性为true，可以使被代理Filter与DelegatingFilterProxy具有同样的生命周期。

## 1.4     FilterChainProxy

       Spring Security底层是通过一系列的Filter来工作的，每个Filter都有其各自的功能，而且各个Filter之间还有关联关系，所以它们的组合顺序也是非常重要的。

       使用Spring Security时，DelegatingFilterProxy代理的就是一个FilterChainProxy。一个FilterChainProxy中可以包含有多个FilterChain，但是某个请求只会对应一个FilterChain，而一个FilterChain中又可以包含有多个Filter。当我们使用基于Spring Security的NameSpace进行配置时，系统会自动为我们注册一个名为springSecurityFilterChain类型为FilterChainProxy的bean（这也是为什么我们在使用SpringSecurity时需要在web.xml中声明一个name为springSecurityFilterChain类型为DelegatingFilterProxy的Filter了。），而且每一个http元素的定义都将拥有自己的FilterChain，而FilterChain中所拥有的Filter则会根据定义的服务自动增减。所以我们不需要显示的再定义这些Filter对应的bean了，除非你想实现自己的逻辑，又或者你想定义的某个属性NameSpace没有提供对应支持等。

       Spring security允许我们在配置文件中配置多个http元素，以针对不同形式的URL使用不同的安全控制。Spring Security将会为每一个http元素创建对应的FilterChain，同时按照它们的声明顺序加入到FilterChainProxy。所以当我们同时定义多个http元素时要确保将更具有特性的URL配置在前。

   <security:http pattern="/login\*.jsp\*" security="none"/>

   <!-- http元素的pattern属性指定当前的http对应的FilterChain将匹配哪些URL，如未指定将匹配所有的请求 -->

   <security:http pattern="/admin/\*\*">

      <security:intercept-url pattern="/\*\*" access="ROLE\_ADMIN"/>

   </security:http>

   <security:http>

      <security:intercept-url pattern="/\*\*" access="ROLE\_USER"/>

   </security:http>

       需要注意的是http拥有一个匹配URL的pattern，未指定时表示匹配所有的请求，其下的子元素intercept-url也有一个匹配URL的pattern，该pattern是在http元素对应pattern基础上的，也就是说一个请求必须先满足http对应的pattern才有可能满足其下intercept-url对应的pattern。

## 1.5     Spring Security定义好的核心Filter

       通过前面的介绍我们知道Spring Security是通过Filter来工作的，为保证Spring Security的顺利运行，其内部实现了一系列的Filter。这其中有几个是在使用Spring Security的Web应用中必定会用到的。接下来我们来简要的介绍一下FilterSecurityInterceptor、ExceptionTranslationFilter、SecurityContextPersistenceFilter和UsernamePasswordAuthenticationFilter。在我们使用http元素时前三者会自动添加到对应的FilterChain中，当我们使用了form-login元素时UsernamePasswordAuthenticationFilter也会自动添加到FilterChain中。所以我们在利用custom-filter往FilterChain中添加自己定义的这些Filter时需要注意它们的位置。

### 1.5.1   FilterSecurityInterceptor

       FilterSecurityInterceptor是用于保护Http资源的，它需要一个AccessDecisionManager和一个AuthenticationManager的引用。它会从SecurityContextHolder获取Authentication，然后通过SecurityMetadataSource可以得知当前请求是否在请求受保护的资源。对于请求那些受保护的资源，如果Authentication.isAuthenticated()返回false或者FilterSecurityInterceptor的alwaysReauthenticate属性为true，那么将会使用其引用的AuthenticationManager再认证一次，认证之后再使用认证后的Authentication替换SecurityContextHolder中拥有的那个。然后就是利用AccessDecisionManager进行权限的检查。

       我们在使用基于NameSpace的配置时所配置的intercept-url就会跟FilterChain内部的FilterSecurityInterceptor绑定。如果要自己定义FilterSecurityInterceptor对应的bean，那么该bean定义大致如下所示：

   <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="/admin/\*\*" access="ROLE\_ADMIN" />

            <security:intercept-url pattern="/\*\*" access="ROLE\_USER,ROLE\_ADMIN" />

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

      </property>

   </bean>

       filter-security-metadata-source用于配置其securityMetadataSource属性。intercept-url用于配置需要拦截的URL与对应的权限关系。

### 1.5.2   ExceptionTranslationFilter

       通过前面的介绍我们知道在Spring Security的Filter链表中ExceptionTranslationFilter就放在FilterSecurityInterceptor的前面。而ExceptionTranslationFilter是捕获来自FilterChain的异常，并对这些异常做处理。ExceptionTranslationFilter能够捕获来自FilterChain所有的异常，但是它只会处理两类异常，AuthenticationException和AccessDeniedException，其它的异常它会继续抛出。如果捕获到的是AuthenticationException，那么将会使用其对应的AuthenticationEntryPoint的commence()处理。如果捕获的异常是一个AccessDeniedException，那么将视当前访问的用户是否已经登录认证做不同的处理，如果未登录，则会使用关联的AuthenticationEntryPoint的commence()方法进行处理，否则将使用关联的AccessDeniedHandler的handle()方法进行处理。

**AuthenticationEntryPoint**是在用户没有登录时用于引导用户进行登录认证的，在实际应用中应根据具体的认证机制选择对应的AuthenticationEntryPoint。

**AccessDeniedHandler**用于在用户已经登录了，但是访问了其自身没有权限的资源时做出对应的处理。ExceptionTranslationFilter拥有的AccessDeniedHandler默认是AccessDeniedHandlerImpl，其会返回一个403错误码到客户端。我们可以通过显示的配置AccessDeniedHandlerImpl，同时给其指定一个errorPage使其可以返回对应的错误页面。当然我们也可以实现自己的AccessDeniedHandler。

   <bean id="exceptionTranslationFilter"

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

      <property name="authenticationEntryPoint">

         <beanclass="org.springframework.security.web.authentication.LoginUrlAuthenticationEntryPoint">

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

         </bean>

      </property>

      <property name="accessDeniedHandler">

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

            <property name="errorPage" value="/access\_denied.jsp" />

         </bean>

      </property>

   </bean>

       在上述配置中我们指定了AccessDeniedHandler为AccessDeniedHandlerImpl，同时为其指定了errorPage，这样发生AccessDeniedException后将转到对应的errorPage上。指定了AuthenticationEntryPoint为使用表单登录的LoginUrlAuthenticationEntryPoint。此外，需要注意的是如果该filter是作为自定义filter加入到由NameSpace自动建立的FilterChain中时需把它放在内置的ExceptionTranslationFilter后面，否则异常都将被内置的ExceptionTranslationFilter所捕获。

   <security:http>

      <security:form-login login-page="/login.jsp"

         username-parameter="username" password-parameter="password"

         login-processing-url="/login.do" />

      <!-- 退出登录时删除session对应的cookie -->

      <security:logout delete-cookies="JSESSIONID" />

      <!-- 登录页面应当是不需要认证的 -->

      <security:intercept-url pattern="/login\*.jsp\*"

         access="IS\_AUTHENTICATED\_ANONYMOUSLY" />

      <security:intercept-url pattern="/\*\*" access="ROLE\_USER" />

      <security:custom-filter ref="exceptionTranslationFilter"after="EXCEPTION\_TRANSLATION\_FILTER"/>

   </security:http>

       在捕获到AuthenticationException之后，调用AuthenticationEntryPoint的commence()方法引导用户登录之前，ExceptionTranslationFilter还做了一件事，那就是使用RequestCache将当前HttpServletRequest的信息保存起来，以至于用户成功登录后需要跳转到之前的页面时可以获取到这些信息，然后继续之前的请求，比如用户可能在未登录的情况下发表评论，待用户提交评论的时候就会将包含评论信息的当前请求保存起来，同时引导用户进行登录认证，待用户成功登录后再利用原来的request包含的信息继续之前的请求，即继续提交评论，所以待用户登录成功后我们通常看到的是用户成功提交了评论之后的页面。Spring Security默认使用的RequestCache是HttpSessionRequestCache，其会将HttpServletRequest相关信息封装为一个SavedRequest保存在HttpSession中。

### 1.5.3   SecurityContextPersistenceFilter

       SecurityContextPersistenceFilter会在请求开始时从配置好的SecurityContextRepository中获取SecurityContext，然后把它设置给SecurityContextHolder。在请求完成后将SecurityContextHolder持有的SecurityContext再保存到配置好的SecurityContextRepository，同时清除SecurityContextHolder所持有的SecurityContext。在使用NameSpace时，Spring  Security默认会给SecurityContextPersistenceFilter的SecurityContextRepository设置一个HttpSessionSecurityContextRepository，其会将SecurityContext保存在HttpSession中。此外HttpSessionSecurityContextRepository有一个很重要的属性allowSessionCreation，默认为true。这样需要把SecurityContext保存在session中时，如果不存在session，可以自动创建一个。也可以把它设置为false，这样在请求结束后如果没有可用的session就不会保存SecurityContext到session了。SecurityContextRepository还有一个空实现，NullSecurityContextRepository，如果在请求完成后不想保存SecurityContext也可以使用它。

       这里再补充说明一点为什么SecurityContextPersistenceFilter在请求完成后需要清除SecurityContextHolder的SecurityContext。SecurityContextHolder在设置和保存SecurityContext都是使用的静态方法，具体操作是由其所持有的SecurityContextHolderStrategy完成的。默认使用的是基于线程变量的实现，即SecurityContext是存放在ThreadLocal里面的，这样各个独立的请求都将拥有自己的SecurityContext。在请求完成后清除SecurityContextHolder中的SucurityContext就是清除ThreadLocal，Servlet容器一般都有自己的线程池，这可以避免Servlet容器下一次分发线程时线程中还包含SecurityContext变量，从而引起不必要的错误。

       下面是一个SecurityContextPersistenceFilter的简单配置。

   <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>

### 1.5.4   UsernamePasswordAuthenticationFilter

       UsernamePasswordAuthenticationFilter用于处理来自表单提交的认证。该表单必须提供对应的用户名和密码，对应的参数名默认为j\_username和j\_password。如果不想使用默认的参数名，可以通过UsernamePasswordAuthenticationFilter的usernameParameter和passwordParameter进行指定。表单的提交路径默认是“j\_spring\_security\_check”，也可以通过UsernamePasswordAuthenticationFilter的filterProcessesUrl进行指定。通过属性postOnly可以指定只允许登录表单进行post请求，默认是true。其内部还有登录成功或失败后进行处理的AuthenticationSuccessHandler和AuthenticationFailureHandler，这些都可以根据需求做相关改变。此外，它还需要一个AuthenticationManager的引用进行认证，这个是没有默认配置的。

   <bean id="authenticationFilter"

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

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

      <property name="usernameParameter" value="username"/>

      <property name="passwordParameter" value="password"/>

      <property name="filterProcessesUrl" value="/login.do" />

   </bean>

       如果要在http元素定义中使用上述AuthenticationFilter定义，那么完整的配置应该类似于如下这样子。

<beans xmlns="http://www.springframework.org/schema/beans"

   xmlns:security="http://www.springframework.org/schema/security"

   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

   xsi:schemaLocation="http://www.springframework.org/schema/beans

          http://www.springframework.org/schema/beans/spring-beans-3.1.xsd

          http://www.springframework.org/schema/security

          http://www.springframework.org/schema/security/spring-security-3.1.xsd">

   <!-- entry-point-ref指定登录入口 -->

   <security:http entry-point-ref="authEntryPoint">

      <security:logout delete-cookies="JSESSIONID" />

      <security:intercept-url pattern="/login\*.jsp\*"

         access="IS\_AUTHENTICATED\_ANONYMOUSLY" />

      <security:intercept-url pattern="/\*\*" access="ROLE\_USER" />

      <!-- 添加自己定义的AuthenticationFilter到FilterChain的FORM\_LOGIN\_FILTER位置 -->

      <security:custom-filter ref="authenticationFilter" position="FORM\_LOGIN\_FILTER"/>

   </security:http>

   <!-- AuthenticationEntryPoint，引导用户进行登录 -->

   <bean id="authEntryPoint"class="org.springframework.security.web.authentication.LoginUrlAuthenticationEntryPoint">

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

   </bean>

   <!-- 认证过滤器 -->

   <bean id="authenticationFilter"

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

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

      <property name="usernameParameter" value="username"/>

      <property name="passwordParameter" value="password"/>

      <property name="filterProcessesUrl" value="/login.do" />

   </bean>

   <security:authentication-manager alias="authenticationManager">

      <security:authentication-provider

         user-service-ref="userDetailsService">

         <security:password-encoder hash="md5"

            base64="true">

            <security:salt-source user-property="username" />

         </security:password-encoder>

      </security:authentication-provider>

   </security:authentication-manager>

   <bean id="userDetailsService"

      class="org.springframework.security.core.userdetails.jdbc.JdbcDaoImpl">

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

   </bean>

</beans>

# [Spring Security（10）——退出登录logout](http://www.cnblogs.com/fenglan/p/5913252.html)

     要实现退出登录的功能我们需要在http元素下定义logout元素，这样Spring Security将自动为我们添加用于处理退出登录的过滤器LogoutFilter到FilterChain。当我们指定了http元素的auto-config属性为true时logout定义是会自动配置的，此时我们默认退出登录的URL为“/j\_spring\_security\_logout”，可以通过logout元素的logout-url属性来改变退出登录的默认地址。

   <security:logout logout-url="/logout.do"/>

       此外，我们还可以给logout指定如下属性：

|  |  |
| --- | --- |
| **属性名** | **作用** |
| invalidate-session | 表示是否要在退出登录后让当前session失效，默认为true。 |
| delete-cookies | 指定退出登录后需要删除的cookie名称，多个cookie之间以逗号分隔。 |
| logout-success-url | 指定成功退出登录后要重定向的URL。需要注意的是对应的URL应当是不需要登录就可以访问的。 |
| success-handler-ref | 指定用来处理成功退出登录的LogoutSuccessHandler的引用。 |

# [Spring Security（11）——匿名认证](http://www.cnblogs.com/fenglan/p/5913265.html)

**目录**

1.1     配置

1.2     AuthenticationTrustResolver

       对于匿名访问的用户，Spring Security支持为其建立一个匿名的AnonymousAuthenticationToken存放在SecurityContextHolder中，这就是所谓的匿名认证。这样在以后进行权限认证或者做其它操作时我们就不需要再判断SecurityContextHolder中持有的Authentication对象是否为null了，而直接把它当做一个正常的Authentication进行使用就OK了。

## 1.1     配置

       使用NameSpace时，http元素的使用默认就会启用对匿名认证的支持，不过我们也可以通过设置http元素下的anonymous元素的enabled属性为false停用对匿名认证的支持。以下是anonymous元素可以配置的属性，以及它们的默认值。

      <security:anonymous enabled="true" key="doesNotMatter" username="anonymousUser"granted-authority="ROLE\_ANONYMOUS"/>

       key用于指定一个在AuthenticationFilter和AuthenticationProvider之间共享的值。username用于指定匿名用户所对应的用户名，granted-authority用于指定匿名用户所具有的权限。

       与匿名认证相关的类有三个，AnonymousAuthenticationToken将作为一个Authentication的实例存放在SecurityContextHolder中；过滤器运行到AnonymousAuthenticationFilter时，如果SecurityContextHolder中持有的Authentication还是空的，则AnonymousAuthenticationFilter将创建一个AnonymousAuthenticationToken并存放在SecurityContextHolder中。最后一个相关的类是AnonymousAuthenticationProvider，其会添加到ProviderManager的AuthenticationProvider列表中，以支持对AnonymousAuthenticationToken的认证。AnonymousAuthenticationToken的认证是在AbstractSecurityInterceptor中的beforeInvocation()方法中进行的。使用http元素定义时这些bean都是会自动定义和添加的。如果需要手动定义这些bean的话，那么可以如下定义：

   <bean id="anonymousAuthFilter"

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

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

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

   </bean>

   <bean id="anonymousAuthenticationProvider"

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

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

   </bean>

       key是在AnonymousAuthenticationProvider和AnonymousAuthenticationFilter之间共享的，它们必须保持一致，AnonymousAuthenticationProvider将使用本身拥有的key与传入的AnonymousAuthenticationToken的key作比较，相同则认为可以进行认证，否则将抛出异常BadCredentialsException。userAttribute属性是以usernameInTheAuthenticationToken,grantedAuthority[,grantedAuthority]的形式进行定义的。

## 1.2     AuthenticationTrustResolver

       AuthenticationTrustResolver是一个接口，其中定义有两个方法，isAnonymous()和isRememberMe()，它们都接收一个Authentication对象作为参数。它有一个默认实现类AuthenticationTrustResolverImpl，Spring Security就是使用它来判断一个SecurityContextHolder持有的Authentication是否AnonymousAuthenticationToken或RememberMeAuthenticationToken。如当ExceptionTranslationFilter捕获到一个AccessDecisionManager后就会使用它来判断当前Authentication对象是否为一个AnonymousAuthenticationToken，如果是则交由AuthenticationEntryPoint处理，否则将返回403错误码。

# [Spring Security（12）——Remember-Me功能](http://www.cnblogs.com/fenglan/p/5913324.html)

**目录**

1.1     概述

1.2     基于简单加密token的方法

1.3     基于持久化token的方法

1.4     Remember-Me相关接口和实现类

1.4.1    TokenBasedRememberMeServices

1.4.2    PersistentTokenBasedRememberMeServices

## 1.1          概述

       Remember-Me是指网站能够在Session之间记住登录用户的身份，具体来说就是我成功认证一次之后在一定的时间内我可以不用再输入用户名和密码进行登录了，系统会自动给我登录。这通常是通过服务端发送一个cookie给客户端浏览器，下次浏览器再访问服务端时服务端能够自动检测客户端的cookie，根据cookie值触发自动登录操作。Spring Security为这些操作的发生提供必要的钩子，并且针对于Remember-Me功能有两种实现。一种是简单的使用加密来保证基于cookie的token的安全，另一种是通过数据库或其它持久化存储机制来保存生成的token。

       需要注意的是两种实现都需要一个UserDetailsService。如果你使用的AuthenticationProvider不使用UserDetailsService，那么记住我将会不起作用，除非在你的ApplicationContext中拥有一个UserDetailsService类型的bean。

## 1.2          基于简单加密token的方法

       当用户选择了记住我成功登录后，Spring Security将会生成一个cookie发送给客户端浏览器。cookie值由如下方式组成：

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

* username：登录的用户名。
* password：登录的密码。
* expirationTime：token失效的日期和时间，以毫秒表示。
* key：用来防止修改token的一个key。

       这样用来实现Remember-Me功能的token只能在指定的时间内有效，且必须保证token中所包含的username、password和key没有被改变才行。需要注意的是，这样做其实是存在安全隐患的，那就是在用户获取到实现记住我功能的token后，任何用户都可以在该token过期之前通过该token进行自动登录。如果用户发现自己的token被盗用了，那么他可以通过改变自己的登录密码来立即使其所有的记住我token失效。如果希望我们的应用能够更安全一点，可以使用接下来要介绍的持久化token方式，或者不使用Remember-Me功能，因为Remember-Me功能总是有点不安全的。

       使用这种方式时，我们只需要在http元素下定义一个remember-me元素，同时指定其key属性即可。key属性是用来标记存放token的cookie的，对应上文提到的生成token时的那个key。

   <security:http auto-config="true">

      <security:form-login/>

      <!-- 定义记住我功能 -->

      <security:remember-me key="elim"/>

      <security:intercept-url pattern="/\*\*" access="ROLE\_USER" />

   </security:http>

       这里有两个需要注意的地方。第一，如果你的登录页面是自定义的，那么需要在登录页面上新增一个名为“\_spring\_security\_remember\_me”的checkbox，这是基于NameSpace定义提供的默认名称，如果要自定义可以自己定义TokenBasedRememberMeServices或PersistentTokenBasedRememberMeServices对应的bean，然后通过其parameter属性进行指定，具体操作请参考后文关于《Remember-Me相关接口和实现类》部分内容。第二，上述功能需要一个UserDetailsService，如果在你的ApplicationContext中已经拥有一个了，那么Spring Security将自动获取；如果没有，那么当然你需要定义一个；如果拥有在ApplicationContext中拥有多个UserDetailsService定义，那么你需要通过remember-me元素的user-service-ref属性指定将要使用的那个。如：

   <security:http auto-config="true">

      <security:form-login/>

      <!-- 定义记住我功能，通过user-service-ref指定将要使用的UserDetailsService-->

      <security:remember-me key="elim" user-service-ref="userDetailsService"/>

      <security:intercept-url pattern="/\*\*" access="ROLE\_USER" />

   </security:http>

   <bean id="userDetailsService"class="org.springframework.security.core.userdetails.jdbc.JdbcDaoImpl">

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

   </bean>

## 1.3          基于持久化token的方法

       持久化token的方法跟简单加密token的方法在实现Remember-Me功能上大体相同，都是在用户选择了“记住我”成功登录后，将生成的token存入cookie中并发送到客户端浏览器，待到下次用户访问系统时，系统将直接从客户端cookie中读取token进行认证。所不同的是基于简单加密token的方法，一旦用户登录成功后，生成的token将在客户端保存一段时间，如果用户不点击退出登录，或者不修改密码，那么在cookie失效之前，他都可以使用该token进行登录，哪怕该token被别人盗用了，用户与盗用者都同样可以进行登录。而基于持久化token的方法采用这样的实现逻辑：

       （1）用户选择了“记住我”成功登录后，将会把username、随机产生的序列号、生成的token存入一个数据库表中，同时将它们的组合生成一个cookie发送给客户端浏览器。

       （2）当下一次没有登录的用户访问系统时，首先检查cookie，如果对应cookie中包含的username、序列号和token与数据库中保存的一致，则表示其通过验证，系统将重新生成一个新的token替换数据库中对应组合的旧token，序列号保持不变，同时删除旧的cookie，重新生成包含新生成的token，就的序列号和username的cookie发送给客户端。

       （3）如果检查cookie时，cookie中包含的username和序列号跟数据库中保存的匹配，但是token不匹配。这种情况极有可能是因为你的cookie被人盗用了，由于盗用者使用你原本通过认证的cookie进行登录了导致旧的token失效，而产生了新的token。这个时候Spring Security就可以发现cookie被盗用的情况，它将删除数据库中与当前用户相关的所有token记录，这样盗用者使用原有的cookie将不能再登录，同时提醒用户其帐号有被盗用的可能性。

       （4）如果对应cookie不存在，或者包含的username和序列号与数据库中保存的不一致，那么将会引导用户到登录页面。

       从以上逻辑我们可以看出持久化token的方法比简单加密token的方法更安全，因为一旦你的cookie被人盗用了，你只要再利用原有的cookie试图自动登录一次，原有的token将失效导致盗用者不能再使用原来盗用的cookie进行登录了，同时用户可以发现自己的cookie有被盗用的可能性。但因为cookie被盗用后盗用者还可以在用户下一次登录前顺利的进行登录，所以如果你的应用对安全性要求比较高就不要使用Remember-Me功能了。

    使用持久化token方法时需要我们的数据库中拥有如下表及其表结构。

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

                                    series varchar(64) primary key,

                                    token varchar(64) not null,

                                    last\_used timestamp not null)

    然后还是通过remember-me元素来使用，只是这个时候我们需要其data-source-ref属性指定对应的数据源，同时别忘了它也同样需要ApplicationContext中拥有UserDetailsService，如果拥有多个，请使用user-service-ref属性指定remember-me使用的是哪一个。

   <security:http auto-config="true">

      <security:form-login/>

      <!-- 定义记住我功能 -->

      <security:remember-me data-source-ref="dataSource"/>

      <security:intercept-url pattern="/\*\*" access="ROLE\_USER" />

   </security:http>

## 1.4     Remember-Me相关接口和实现类

在上述介绍中，我们实现Remember-Me功能是通过Spring Security为了简化Remember-Me而提供的NameSpace进行定义的。而底层实际上还是通过RememberMeServices、UsernamePasswordAuthenticationFilter和RememberMeAuthenticationFilter的协作来完成的。RememberMeServices是Spring Security为Remember-Me提供的一个服务接口，其定义如下。

**publicinterface** RememberMeServices {

    /\*\*

     \* 自动登录。在实现这个方法的时候应该判断用户提供的Remember-Me cookie是否有效，如果无效，应当直接忽略。

     \* 如果认证成功应当返回一个AuthenticationToken，推荐返回RememberMeAuthenticationToken；

     \* 如果认证不成功应当返回null。

     \*/

    Authentication autoLogin(HttpServletRequest request, HttpServletResponse response);

    /\*\*

     \* 在用户登录失败时调用。实现者应当做一些类似于删除cookie之类的处理。

     \*/

**void** loginFail(HttpServletRequest request, HttpServletResponse response);

    /\*\*

     \* 在用户成功登录后调用。实现者可以在这里判断用户是否选择了“Remember-Me”登录，然后做相应的处理。

     \*/

**void** loginSuccess(HttpServletRequest request, HttpServletResponse response,

        Authentication successfulAuthentication);

}

       UsernamePasswordAuthenticationFilter拥有一个RememberMeServices的引用，默认是一个空实现的NullRememberMeServices，而实际当我们通过remember-me定义启用Remember-Me时，它会是一个具体的实现。用户的请求会先通过UsernamePasswordAuthenticationFilter，如认证成功会调用RememberMeServices的loginSuccess()方法，否则调用RememberMeServices的loginFail()方法。UsernamePasswordAuthenticationFilter是不会调用RememberMeServices的autoLogin()方法进行自动登录的。之后运行到RememberMeAuthenticationFilter时如果检测到还没有登录，那么RememberMeAuthenticationFilter会尝试着调用所包含的RememberMeServices的autoLogin()方法进行自动登录。关于RememberMeServices Spring Security已经为我们提供了两种实现，分别对应于前文提到的基于简单加密token和基于持久化token的方法。

### 1.4.1   TokenBasedRememberMeServices

       TokenBasedRememberMeServices对应于前文介绍的使用namespace时基于简单加密token的实现。TokenBasedRememberMeServices会在用户选择了记住我成功登录后，生成一个包含token信息的cookie发送到客户端；如果用户登录失败则会删除客户端保存的实现Remember-Me的cookie。需要自动登录时，它会判断cookie中所包含的关于Remember-Me的信息是否与系统一致，一致则返回一个RememberMeAuthenticationToken供RememberMeAuthenticationProvider处理，不一致则会删除客户端的Remember-Me cookie。TokenBasedRememberMeServices还实现了Spring Security的LogoutHandler接口，所以它可以在用户退出登录时立即清除Remember-Me cookie。

       如果把使用namespace定义Remember-Me改为直接定义RememberMeServices和对应的Filter来使用的话，那么我们可以如下定义。

   <security:http>

      <security:form-login login-page="/login.jsp"/>

      <security:intercept-url pattern="/login\*.jsp\*"access="IS\_AUTHENTICATED\_ANONYMOUSLY"/>

      <security:intercept-url pattern="/\*\*" access="ROLE\_USER" />

      <!-- 把usernamePasswordAuthenticationFilter加入FilterChain -->

      <security:custom-filter ref="usernamePasswordAuthenticationFilter"before="FORM\_LOGIN\_FILTER"/>

      <security:custom-filter ref="rememberMeFilter" position="REMEMBER\_ME\_FILTER"/>

   </security:http>

   <!-- 用于认证的AuthenticationManager -->

   <security:authentication-manager alias="authenticationManager">

      <security:authentication-provider

         user-service-ref="userDetailsService"/>

      <security:authentication-provider ref="rememberMeAuthenticationProvider"/>

   </security:authentication-manager>

   <bean id="userDetailsService"

      class="org.springframework.security.core.userdetails.jdbc.JdbcDaoImpl">

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

   </bean>

   <bean id="usernamePasswordAuthenticationFilter"class="org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter">

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

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

      <!-- 指定request中包含的用户名对应的参数名 -->

      <property name="usernameParameter" value="username"/>

      <property name="passwordParameter" value="password"/>

      <!-- 指定登录的提交地址 -->

      <property name="filterProcessesUrl" value="/login.do"/>

   </bean>

   <!-- Remember-Me对应的Filter -->

   <bean id="rememberMeFilter"

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

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

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

   </bean>

   <!-- RememberMeServices的实现 -->

   <bean id="rememberMeServices"

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

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

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

      <!-- 指定request中包含的用户是否选择了记住我的参数名 -->

      <property name="parameter" value="rememberMe"/>

   </bean>

   <!-- key值需与对应的RememberMeServices保持一致 -->

   <bean id="rememberMeAuthenticationProvider"

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

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

   </bean>

       需要注意的是RememberMeAuthenticationProvider在认证RememberMeAuthenticationToken的时候是比较它们拥有的key是否相等，而RememberMeAuthenticationToken的key是TokenBasedRememberMeServices提供的，所以在使用时需要保证RememberMeAuthenticationProvider和TokenBasedRememberMeServices的key属性值保持一致。需要配置UsernamePasswordAuthenticationFilter的rememberMeServices为我们定义好的TokenBasedRememberMeServices，把RememberMeAuthenticationProvider加入AuthenticationManager的providers列表，并添加RememberMeAuthenticationFilter和UsernamePasswordAuthenticationFilter到FilterChainProxy。

### 1.4.2   PersistentTokenBasedRememberMeServices

       PersistentTokenBasedRememberMeServices是RememberMeServices基于前文提到的持久化token的方式实现的。具体实现逻辑跟前文介绍的以NameSpace的方式使用基于持久化token的Remember-Me是一样的，这里就不再赘述了。此外，如果单独使用，其使用方式和上文描述的TokenBasedRememberMeServices是一样的，这里也不再赘述了。

       需要注意的是PersistentTokenBasedRememberMeServices是需要将token进行持久化的，所以我们必须为其指定存储token的PersistentTokenRepository。Spring Security对此有两种实现，InMemoryTokenRepositoryImpl和JdbcTokenRepositoryImpl。前者是将token存放在内存中的，通常用于测试，而后者是将token存放在数据库中。PersistentTokenBasedRememberMeServices默认使用的是前者，我们可以通过其tokenRepository属性来指定使用的PersistentTokenRepository。

       使用JdbcTokenRepositoryImpl时我们可以使用在前文提到的默认表结构。如果需要使用自定义的表，那么我们可以对JdbcTokenRepositoryImpl进行重写。定义JdbcTokenRepositoryImpl时需要指定一个数据源dataSource，同时可以通过设置参数createTableOnStartup的值来控制是否要在系统启动时创建对应的存入token的表，默认创建语句为“create table persistent\_logins (username varchar(64) not null, series varchar(64) primary key, token varchar(64) not null, last\_used timestamp not null)”，但是如果自动创建时对应的表已经存在于数据库中，则会抛出异常。createTableOnStartup属性默认为false。

       直接显示地使用PersistentTokenBasedRememberMeServices和上文提到的直接显示地使用TokenBasedRememberMeServices的方式是一样的，我们只需要将上文提到的配置中RememberMeServices实现类TokenBasedRememberMeServices换成PersistentTokenBasedRememberMeServices即可。

   <!-- RememberMeServices的实现 -->

   <bean id="rememberMeServices"

  class="org.springframework.security.web.authentication.rememberme.PersistentTokenBasedRememberMeServices">

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

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

      <!-- 指定request中包含的用户是否选择了记住我的参数名 -->

      <property name="parameter" value="rememberMe"/>

      <!-- 指定PersistentTokenRepository -->

      <property name="tokenRepository">

         <beanclass="org.springframework.security.web.authentication.rememberme.JdbcTokenRepositoryImpl">

            <!-- 数据源 -->

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

            <!-- 是否在系统启动时创建持久化token的数据库表 -->

            <property name="createTableOnStartup" value="false"/>

         </bean>

      </property>

   </bean>

# [Spring Security（13）——session管理](http://www.cnblogs.com/fenglan/p/5913352.html)

1.1     检测session超时

1.2     concurrency-control

1.3     session 固定攻击保护

       Spring Security通过http元素下的子元素session-management提供了对Http Session管理的支持。

## 1.1     检测session超时

       Spring Security可以在用户使用已经超时的sessionId进行请求时将用户引导到指定的页面。这个可以通过如下配置来实现。

   <security:http>

      ...

      <!-- session管理，invalid-session-url指定使用已经超时的sessionId进行请求需要重定向的页面 -->

      <security:session-management invalid-session-url="/session\_timeout.jsp"/>

      ...

   </security:http>

       需要注意的是session超时的重定向页面应当是不需要认证的，否则再重定向到session超时页面时会直接转到用户登录页面。此外如果你使用这种方式来检测session超时，当你退出了登录，然后在没有关闭浏览器的情况下又重新进行了登录，Spring Security可能会错误的报告session已经超时。这是因为即使你已经退出登录了，但当你设置session无效时，对应保存session信息的cookie并没有被清除，等下次请求时还是会使用之前的sessionId进行请求。解决办法是显示的定义用户在退出登录时删除对应的保存session信息的cookie。

   <security:http>

      ...

      <!-- 退出登录时删除session对应的cookie -->

      <security:logout delete-cookies="JSESSIONID"/>

      ...

   </security:http>

       此外，Spring Security并不保证这对所有的Servlet容器都有效，到底在你的容器上有没有效，需要你自己进行实验。

## 1.2     concurrency-control

       通常情况下，在你的应用中你可能只希望同一用户在同时登录多次时只能有一个是成功登入你的系统的，通常对应的行为是后一次登录将使前一次登录失效，或者直接限制后一次登录。Spring Security的session-management为我们提供了这种限制。

       首先需要我们在web.xml中定义如下监听器。

   <listener>

   <listener-class>org.springframework.security.web.session.HttpSessionEventPublisher</listener-class>

   </listener>

       在session-management元素下有一个concurrency-control元素是用来限制同一用户在应用中同时允许存在的已经通过认证的session数量。这个值默认是1，可以通过concurrency-control元素的max-sessions属性来指定。

   <security:http auto-config="true">

      ...

      <security:session-management>

         <security:concurrency-control max-sessions="1"/>

      </security:session-management>

      ...

   </security:http>

       当同一用户同时存在的已经通过认证的session数量超过了max-sessions所指定的值时，Spring Security的默认策略是将先前的设为无效。如果要限制用户再次登录可以设置concurrency-control的error-if-maximum-exceeded的值为true。

   <security:http auto-config="true">

      ...

      <security:session-management>

         <security:concurrency-control max-sessions="1" error-if-maximum-exceeded="true"/>

      </security:session-management>

      ...

   </security:http>

       设置error-if-maximum-exceeded为true后如果你之前已经登录了，然后想再次登录，那么系统将会拒绝你的登录，同时将重定向到由form-login指定的authentication-failure-url。如果你的再次登录是通过Remember-Me来完成的，那么将不会转到authentication-failure-url，而是返回未授权的错误码401给客户端，如果你还是想重定向一个指定的页面，那么你可以通过session-management的session-authentication-error-url属性来指定，同时需要指定该url为不受Spring Security管理，即通过http元素设置其secure=”none”。

   <security:http security="none" pattern="/none/\*\*" />

   <security:http>

      <security:form-login/>

      <security:logout/>

      <security:intercept-url pattern="/\*\*" access="ROLE\_USER"/>

      <!-- session-authentication-error-url必须是不受Spring Security管理的 -->

      <security:session-management session-authentication-error-url="/none/session\_authentication\_error.jsp">

         <security:concurrency-control max-sessions="1" error-if-maximum-exceeded="true"/>

      </security:session-management>

      <security:remember-me data-source-ref="dataSource"/>

   </security:http>

       在上述配置中我们配置了session-authentication-error-url为“/none/session\_authentication\_error.jsp”，同时我们通过<security:http security="none" pattern="/none/\*\*" />指定了以“/none”开始的所有URL都不受Spring Security控制，这样当用户进行登录以后，再次通过Remember-Me进行自动登录时就会重定向到“/none/session\_authentication\_error.jsp”了。

       在上述配置中为什么我们需要通过<security:http security="none" pattern="/none/\*\*" />指定我们的session-authentication-error-url不受Spring Security控制呢？把它换成<security:intercept-url pattern="/none/\*\*"access="IS\_AUTHENTICATED\_ANONYMOUSLY"/>不行吗？这就涉及到之前所介绍的它们两者之间的区别了。前者表示不使用任何Spring Security过滤器，自然也就不需要通过Spring Security的认证了，而后者是会被Spring Security的FilterChain进行过滤的，只是其对应的URL可以匿名访问，即不需要登录就可访问。使用后者时，REMEMBER\_ME\_FILTER检测到用户没有登录，同时其又提供了Remember-Me的相关信息，这将使得REMEMBER\_ME\_FILTER进行自动登录，那么在自动登录时由于我们限制了同一用户同一时间只能登录一次，后来者将被拒绝登录，这个时候将重定向到session-authentication-error-url，重定向访问session-authentication-error-url时，经过REMEMBER\_ME\_FILTER时又会自动登录，这样就形成了一个死循环。所以session-authentication-error-url应当使用<security:http security="none" pattern="/none/\*\*" />设置为不受Spring Security控制，而不是使用<security:intercept-url pattern="/none/\*\*"access="IS\_AUTHENTICATED\_ANONYMOUSLY"/>。

       此外，可以通过expired-url属性指定当用户尝试使用一个由于其再次登录导致session超时的session时所要跳转的页面。同时需要注意设置该URL为不需要进行认证。

   <security:http auto-config="true">

      <security:form-login/>

      <security:logout/>

      <security:intercept-url pattern="/expired.jsp" access="IS\_AUTHENTICATED\_ANONYMOUSLY"/>

      <security:intercept-url pattern="/\*\*" access="ROLE\_USER"/>

      <security:session-management>

         <security:concurrency-control max-sessions="1" expired-url="/expired.jsp" />

      </security:session-management>

   </security:http>

## 1.3     session 固定攻击保护

       session固定是指服务器在给客户端创建session后，在该session过期之前，它们都将通过该session进行通信。session 固定攻击是指恶意攻击者先通过访问应用来创建一个session，然后再让其他用户使用相同的session进行登录（比如通过发送一个包含该sessionId参数的链接），待其他用户成功登录后，攻击者利用原来的sessionId访问系统将和原用户获得同样的权限。Spring Security默认是对session固定攻击采取了保护措施的，它会在用户登录的时候重新为其生成一个新的session。如果你的应用不需要这种保护或者该保护措施与你的某些需求相冲突，你可以通过session-management的session-fixation-protection属性来改变其保护策略。该属性的可选值有如下三个。

l  **migrateSession**：这是默认值。其表示在用户登录后将新建一个session，同时将原session中的attribute都copy到新的session中。

l  **none**：表示继续使用原来的session。

l  **newSession**：表示重新创建一个新的session，但是不copy原session拥有的attribute。

# [Spring Security（14）——权限鉴定基础](http://www.cnblogs.com/fenglan/p/5913387.html)

**目录**

1.1     Spring Security的AOP Advice思想

1.2     AbstractSecurityInterceptor

1.2.1    ConfigAttribute

1.2.2    RunAsManager

1.2.3    AfterInvocationManager

       Spring Security的权限鉴定是由AccessDecisionManager接口负责的。具体来说是由其中的decide()方法负责，其定义如下。

**void** decide(Authentication authentication, Object object, Collection<ConfigAttribute> configAttributes)

**throws** AccessDeniedException, InsufficientAuthenticationException;

       如你所见，该方法接收三个参数，第一个参数是包含当前用户信息的Authentication对象；第二个参数表示当前正在请求的受保护的对象，基本上来说是MethodInvocation（使用AOP）、JoinPoint（使用Aspectj）和FilterInvocation（Web请求）三种类型；第三个参数表示与当前正在访问的受保护对象的配置属性，如一个角色列表。

## 1.1     Spring Security的AOP Advice思想

       对于使用AOP而言，我们可以使用几种不同类型的advice：before、after、throws和around。其中around advice是非常实用的，通过它我们可以控制是否要执行方法、是否要修改方法的返回值，以及是否要抛出异常。Spring Security在对方法调用和Web请求时也是使用的around advice的思想。在方法调用时，可以使用标准的Spring AOP来达到around advice的效果，而在进行Web请求时是通过标准的Filter来达到around advice的效果。

       对于大部分人而言都比较喜欢对Service层的方法调用进行权限控制，因为我们的主要业务逻辑都是在Service层进行实现的。如果你只是想保护Service层的方法，那么使用Spring AOP就可以了。如果你需要直接保护领域对象，那么你可以考虑使用Aspectj。

       你可以选择使用Aspectj或Spring AOP对方法调用进行鉴权，或者选择使用Filter对Web请求进行鉴权。当然，你也可以选择使用这三种方式的任意组合进行鉴权。通常的做法是使用Filter对Web请求进行一个比较粗略的鉴权，辅以使用Spring AOP对Service层的方法进行较细粒度的鉴权。

## 1.2     AbstractSecurityInterceptor

       AbstractSecurityInterceptor是一个实现了对受保护对象的访问进行拦截的抽象类，其中有几个比较重要的方法。beforeInvocation()方法实现了对访问受保护对象的权限校验，内部用到了AccessDecisionManager和AuthenticationManager；finallyInvocation()方法用于实现受保护对象请求完毕后的一些清理工作，主要是如果在beforeInvocation()中改变了SecurityContext，则在finallyInvocation()中需要将其恢复为原来的SecurityContext，该方法的调用应当包含在子类请求受保护资源时的finally语句块中；afterInvocation()方法实现了对返回结果的处理，在注入了AfterInvocationManager的情况下默认会调用其decide()方法。AbstractSecurityInterceptor只是提供了这几种方法，并且包含了默认实现，具体怎么调用将由子类负责。每一种受保护对象都拥有继承自AbstractSecurityInterceptor的拦截器类， MethodSecurityInterceptor将用于调用受保护的方法，而FilterSecurityInterceptor将用于受保护的Web请求。它们在处理受保护对象的请求时都具有一致的逻辑，具体的逻辑如下。

       1、先将正在请求调用的受保护对象传递给beforeInvocation()方法进行权限鉴定。

       2、权限鉴定失败就直接抛出异常了。

       3、鉴定成功将尝试调用受保护对象，调用完成后，不管是成功调用，还是抛出异常，都将执行finallyInvocation()。

       4、如果在调用受保护对象后没有抛出异常，则调用afterInvocation()。

       以下是MethodSecurityInterceptor在进行方法调用的一段核心代码。

**public** Object invoke(MethodInvocation mi) **throws** Throwable {

        InterceptorStatusToken token = **super**.beforeInvocation(mi);

        Object result;

**try** {

            result = mi.proceed();

        } **finally** {

**super**.finallyInvocation(token);

        }

**returnsuper**.afterInvocation(token, result);

    }

### 1.2.1   ConfigAttribute

       AbstractSecurityInterceptor的beforeInvocation()方法内部在进行鉴权的时候使用的是注入的AccessDecisionManager的decide()方法进行的。如前所述，decide()方法是需要接收一个受保护对象对应的ConfigAttribute集合的。一个ConfigAttribute可能只是一个简单的角色名称，具体将视AccessDecisionManager的实现者而定。AbstractSecurityInterceptor将使用一个SecurityMetadataSource对象来获取与受保护对象关联的ConfigAttribute集合，具体SecurityMetadataSource将由子类实现提供。ConfigAttribute将通过注解的形式定义在受保护的方法上，或者通过access属性定义在受保护的URL上。例如我们常见的<intercept-url pattern=”/\*\*” access=”ROLE\_USER,ROLE\_ADMIN”/>就表示将ConfigAttribute ROLE\_USER和ROLE\_ADMIN应用在所有的URL请求上。对于默认的AccessDecisionManager的实现，上述配置意味着用户所拥有的权限中只要拥有一个GrantedAuthority与这两个ConfigAttribute中的一个进行匹配则允许进行访问。当然，严格的来说ConfigAttribute只是一个简单的配置属性而已，具体的解释将由AccessDecisionManager来决定。

### 1.2.2  RunAsManager

       在某些情况下你可能会想替换保存在SecurityContext中的Authentication。这可以通过RunAsManager来实现的。在AbstractSecurityInterceptor的beforeInvocation()方法体中，在AccessDecisionManager鉴权成功后，将通过RunAsManager在现有Authentication基础上构建一个新的Authentication，如果新的Authentication不为空则将产生一个新的SecurityContext，并把新产生的Authentication存放在其中。这样在请求受保护资源时从SecurityContext中获取到的Authentication就是新产生的Authentication。待请求完成后会在finallyInvocation()中将原来的SecurityContext重新设置给SecurityContextHolder。AbstractSecurityInterceptor默认持有的是一个对RunAsManager进行空实现的NullRunAsManager。此外，Spring Security对RunAsManager有一个还有一个非空实现类RunAsManagerImpl，其在构造新的Authentication时是这样的逻辑：如果受保护对象对应的ConfigAttribute中拥有以“RUN\_AS\_”开头的配置属性，则在该属性前加上“ROLE\_”，然后再把它作为一个GrantedAuthority赋给将要创建的Authentication（如ConfigAttribute中拥有一个“RUN\_AS\_ADMIN”的属性，则将构建一个“ROLE\_RUN\_AS\_ADMIN”的GrantedAuthority），最后再利用原Authentication的principal、权限等信息构建一个新的Authentication进行返回；如果不存在任何以“RUN\_AS\_”开头的ConfigAttribute，则直接返回null。RunAsManagerImpl构建新的Authentication的核心代码如下所示。

**public** Authentication buildRunAs(Authentication authentication, Object object, Collection<ConfigAttribute> attributes) {

        List<GrantedAuthority> newAuthorities = **new** ArrayList<GrantedAuthority>();

**for** (ConfigAttribute attribute : attributes) {

**if** (**this**.supports(attribute)) {

                GrantedAuthority extraAuthority = **new**SimpleGrantedAuthority(getRolePrefix() + attribute.getAttribute());

                newAuthorities.add(extraAuthority);

            }

        }

**if** (newAuthorities.size() == 0) {

**returnnull**;

        }

        // Add existing authorities

        newAuthorities.addAll(authentication.getAuthorities());

**returnnew** RunAsUserToken(**this**.key, authentication.getPrincipal(), authentication.getCredentials(),

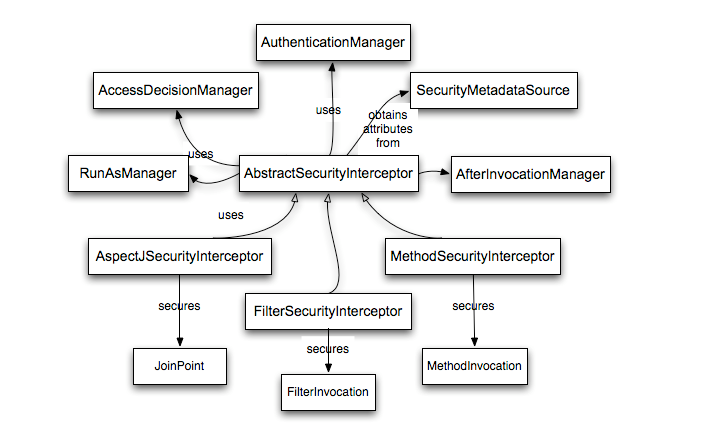
                newAuthorities, authentication.getClass());

    }

### 1.2.3  AfterInvocationManager

       在请求受保护的对象完成以后，可以通过afterInvocation()方法对返回值进行修改。AbstractSecurityInterceptor把对返回值进行修改的控制权交给其所持有的AfterInvocationManager了。AfterInvocationManager可以选择对返回值进行修改、不修改或抛出异常（如：后置权限鉴定不通过）。

       以下是Spring Security官方文档提供的一张关于AbstractSecurityInterceptor相关关系的图。

（注：本文是基于Spring Security3.1.6所写）

# [Spring Security（15）——权限鉴定结构](http://www.cnblogs.com/fenglan/p/5913432.html)

**目录**

1.1      权限

1.2      调用前的处理

1.2.1     AccessDecisionManager

1.2.2     基于投票的AccessDecisionManager实现

1.3      调用后的处理

1.4      角色的继承

## 1.1    权限

       所有的Authentication实现类都保存了一个GrantedAuthority列表，其表示用户所具有的权限。GrantedAuthority是通过AuthenticationManager设置到Authentication对象中的，然后AccessDecisionManager将从Authentication中获取用户所具有的GrantedAuthority来鉴定用户是否具有访问对应资源的权限。

       GrantedAuthority是一个接口，其中只定义了一个getAuthority()方法，其返回值为String类型。该方法允许AccessDecisionManager获取一个能够精确代表该权限的字符串。通过返回一个字符串，一个GrantedAuthority能够很轻易的被大部分AccessDecisionManager读取。如果一个GrantedAuthority不能够精确的使用一个String来表示，那么其对应的getAuthority()方法调用应当返回一个null，这表示AccessDecisionManager必须对该GrantedAuthority的实现有特定的支持，从而可以获取该GrantedAuthority所代表的权限信息。

       Spring Security内置了一个GrantedAuthority的实现，SimpleGrantedAuthority。它直接接收一个表示权限信息的字符串，然后getAuthority()方法直接返回该字符串。Spring Security内置的所有AuthenticationProvider都是使用它来封装Authentication对象的。

## 1.2     调用前的处理

       Spring Security是通过拦截器来控制受保护对象的访问的，如方法调用和Web请求。在正式访问受保护对象之前，Spring Security将使用AccessDecisionManager来鉴定当前用户是否有访问对应受保护对象的权限。

### 1.2.1AccessDecisionManager

       AccessDecisionManager是由AbstractSecurityInterceptor调用的，它负责鉴定用户是否有访问对应资源（方法或URL）的权限。AccessDecisionManager是一个接口，其中只定义了三个方法，其定义如下。

**public interface** AccessDecisionManager {

    /\*\*

     \* 通过传递的参数来决定用户是否有访问对应受保护对象的权限

     \*

     \* **@param** authentication 当前正在请求受包含对象的Authentication

     \* **@param** object 受保护对象，其可以是一个MethodInvocation、JoinPoint或FilterInvocation。

     \* **@param** configAttributes 与正在请求的受保护对象相关联的配置属性

     \*

     \*/

**void** decide(Authentication authentication, Object object, Collection<ConfigAttribute> configAttributes)

**throws** AccessDeniedException, InsufficientAuthenticationException;

    /\*\*

     \* 表示当前AccessDecisionManager是否支持对应的ConfigAttribute

     \*/

**boolean** supports(ConfigAttribute attribute);

    /\*\*

     \* 表示当前AccessDecisionManager是否支持对应的受保护对象类型

     \*/

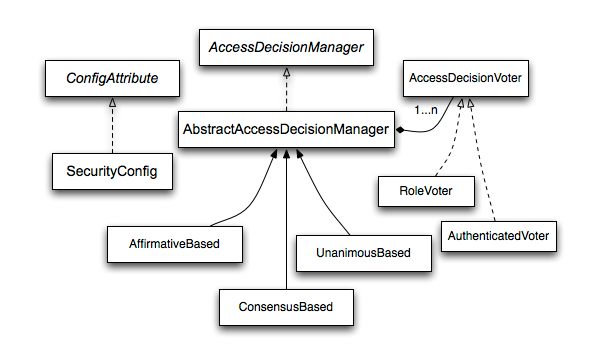
**boolean** supports(Class<?> clazz);

}

       decide()方法用于决定authentication是否符合受保护对象要求的configAttributes。supports(ConfigAttribute attribute)方法是用来判断AccessDecisionManager是否能够处理对应的ConfigAttribute的。supports(Class<?> clazz)方法用于判断配置的AccessDecisionManager是否支持对应的受保护对象类型。

### 1.2.2基于投票的AccessDecisionManager实现

       Spring Security已经内置了几个基于投票的AccessDecisionManager，当然如果需要你也可以实现自己的AccessDecisionManager。以下是Spring Security官方文档提供的一个图，其展示了与基于投票的AccessDecisionManager实现相关的类。



       使用这种方式，一系列的AccessDecisionVoter将会被AccessDecisionManager用来对Authentication是否有权访问受保护对象进行投票，然后再根据投票结果来决定是否要抛出AccessDeniedException。AccessDecisionVoter是一个接口，其中定义有三个方法，具体结构如下所示。

**public interface** AccessDecisionVoter<S> {

**int**ACCESS\_GRANTED = 1;

**int**ACCESS\_ABSTAIN = 0;

**int**ACCESS\_DENIED = -1;

**boolean** supports(ConfigAttribute attribute);

**boolean** supports(Class<?> clazz);

**int** vote(Authentication authentication, S object, Collection<ConfigAttribute> attributes);

}

       vote()方法的返回结果会是AccessDecisionVoter中定义的三个常量之一。ACCESS\_GRANTED表示同意，ACCESS\_DENIED表示返回，ACCESS\_ABSTAIN表示弃权。如果一个AccessDecisionVoter不能判定当前Authentication是否拥有访问对应受保护对象的权限，则其vote()方法的返回值应当为弃权ACCESS\_ABSTAIN。

       Spring Security内置了三个基于投票的AccessDecisionManager实现类，它们分别是AffirmativeBased、ConsensusBased和UnanimousBased。

**AffirmativeBased**的逻辑是这样的：

       （1）只要有AccessDecisionVoter的投票为ACCESS\_GRANTED则同意用户进行访问；

       （2）如果全部弃权也表示通过；

       （3）如果没有一个人投赞成票，但是有人投反对票，则将抛出AccessDeniedException。

**ConsensusBased**的逻辑是这样的：

       （1）如果赞成票多于反对票则表示通过。

       （2）反过来，如果反对票多于赞成票则将抛出AccessDeniedException。

       （3）如果赞成票与反对票相同且不等于0，并且属性allowIfEqualGrantedDeniedDecisions的值为true，则表示通过，否则将抛出异常AccessDeniedException。参数allowIfEqualGrantedDeniedDecisions的值默认为true。

       （4）如果所有的AccessDecisionVoter都弃权了，则将视参数allowIfAllAbstainDecisions的值而定，如果该值为true则表示通过，否则将抛出异常AccessDeniedException。参数allowIfAllAbstainDecisions的值默认为false。

**UnanimousBased**的逻辑与另外两种实现有点不一样，另外两种会一次性把受保护对象的配置属性全部传递给AccessDecisionVoter进行投票，而UnanimousBased会一次只传递一个ConfigAttribute给AccessDecisionVoter进行投票。这也就意味着如果我们的AccessDecisionVoter的逻辑是只要传递进来的ConfigAttribute中有一个能够匹配则投赞成票，但是放到UnanimousBased中其投票结果就不一定是赞成了。UnanimousBased的逻辑具体来说是这样的：

       （1）如果受保护对象配置的某一个ConfigAttribute被任意的AccessDecisionVoter反对了，则将抛出AccessDeniedException。

       （2）如果没有反对票，但是有赞成票，则表示通过。

       （3）如果全部弃权了，则将视参数allowIfAllAbstainDecisions的值而定，true则通过，false则抛出AccessDeniedException。

#### 1.2.2.1RoleVoter

       RoleVoter是Spring Security内置的一个AccessDecisionVoter，其会将ConfigAttribute简单的看作是一个角色名称，在投票的时如果拥有该角色即投赞成票。如果ConfigAttribute是以“ROLE\_”开头的，则将使用RoleVoter进行投票。当用户拥有的权限中有一个或多个能匹配受保护对象配置的以“ROLE\_”开头的ConfigAttribute时其将投赞成票；如果用户拥有的权限中没有一个能匹配受保护对象配置的以“ROLE\_”开头的ConfigAttribute，则RoleVoter将投反对票；如果受保护对象配置的ConfigAttribute中没有以“ROLE\_”开头的，则RoleVoter将弃权。

#### 1.2.2.2AuthenticatedVoter

       AuthenticatedVoter也是Spring Security内置的一个AccessDecisionVoter实现。其主要用来区分匿名用户、通过Remember-Me认证的用户和完全认证的用户。完全认证的用户是指由系统提供的登录入口进行成功登录认证的用户。

       AuthenticatedVoter可以处理的ConfigAttribute有IS\_AUTHENTICATED\_FULLY、IS\_AUTHENTICATED\_REMEMBERED和IS\_AUTHENTICATED\_ANONYMOUSLY。如果ConfigAttribute不在这三者范围之内，则AuthenticatedVoter将弃权。否则将视ConfigAttribute而定，如果ConfigAttribute为IS\_AUTHENTICATED\_ANONYMOUSLY，则不管用户是匿名的还是已经认证的都将投赞成票；如果是IS\_AUTHENTICATED\_REMEMBERED则仅当用户是由Remember-Me自动登录，或者是通过登录入口进行登录认证时才会投赞成票，否则将投反对票；而当ConfigAttribute为IS\_AUTHENTICATED\_FULLY时仅当用户是通过登录入口进行登录的才会投赞成票，否则将投反对票。

       AuthenticatedVoter是通过AuthenticationTrustResolver的isAnonymous()方法和isRememberMe()方法来判断SecurityContextHolder持有的Authentication是否为AnonymousAuthenticationToken或RememberMeAuthenticationToken的，即是否为IS\_AUTHENTICATED\_ANONYMOUSLY和IS\_AUTHENTICATED\_REMEMBERED。

#### 1.2.2.3自定义Voter

       当然，用户也可以通过实现AccessDecisionVoter来实现自己的投票逻辑。

## 1.3     调用后的处理

       AccessDecisionManager是用来在访问受保护的对象之前判断用户是否拥有访问该对象的权限。有的时候我们可能会希望在请求执行完成后对返回值做一些修改，当然，你可以简单的通过AOP来实现这一功能。Spring Security为我们提供了一个AfterInvocationManager接口，它允许我们在受保护对象访问完成后对返回值进行修改或者进行权限鉴定，看是否需要抛出AccessDeniedException，其将由AbstractSecurityInterceptor的子类进行调用。AfterInvocationManager接口的定义如下。

**public interface** AfterInvocationManager {

    Object decide(Authentication authentication, Object object, Collection<ConfigAttribute> attributes,

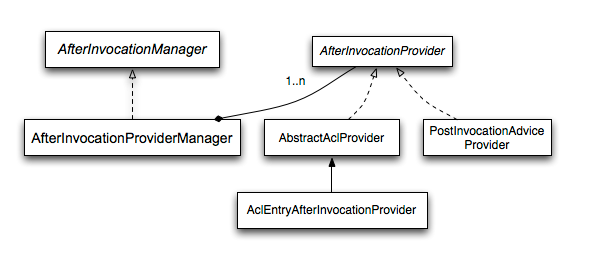
        Object returnedObject) **throws** AccessDeniedException;

**boolean** supports(ConfigAttribute attribute);

**boolean** supports(Class<?> clazz);

}

       以下是Spring Security官方文档提供的一个AfterInvocationManager构造实现的图。

       类似于AuthenticationManager，AfterInvocationManager拥有一个默认的实现类AfterInvocationProviderManager，其中拥有一个由AfterInvocationProvider组成的集合，AfterInvocationProvider与AfterInvocationManager具有相同的方法定义，在调用AfterInvocationProviderManager中的方法时实际上就是依次调用其中包含的AfterInvocationProvider对应的方法。

       需要注意的是AfterInvocationManager需要在受保护对象成功被访问后才能执行。

## 1.4     角色的继承

       对于角色继承这种需求也是经常有的，比如要求ROLE\_ADMIN将拥有所有的ROLE\_USER所具有的权限。当然我们可以给拥有ROLE\_ADMIN角色的用户同时授予ROLE\_USER角色来达到这一效果或者修改需要ROLE\_USER进行访问的资源使用ROLE\_ADMIN也可以访问。Spring Security为我们提供了一种更为简便的办法，那就是角色的继承，它允许我们的ROLE\_ADMIN直接继承ROLE\_USER，这样所有ROLE\_USER可以访问的资源ROLE\_ADMIN也可以访问。定义角色的继承我们需要在ApplicationContext中定义一个RoleHierarchy，然后再把它赋予给一个RoleHierarchyVoter，之后再把该RoleHierarchyVoter加入到我们基于Voter的AccessDecisionManager中，并指定当前使用的AccessDecisionManager为我们自己定义的那个。以下是一个定义角色继承的完整示例。

<beans xmlns="http://www.springframework.org/schema/beans"

   xmlns:security="http://www.springframework.org/schema/security"

   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

   xsi:schemaLocation="http://www.springframework.org/schema/beans

          http://www.springframework.org/schema/beans/spring-beans-3.1.xsd

          http://www.springframework.org/schema/security

          http://www.springframework.org/schema/security/spring-security-3.1.xsd">

   <!-- 通过access-decision-manager-ref指定将要使用的AccessDecisionManager -->

   <security:http access-decision-manager-ref="accessDecisionManager">

      <security:form-login/>

      <security:intercept-url pattern="/admin.jsp" access="ROLE\_ADMIN" />

      <security:intercept-url pattern="/\*\*" access="ROLE\_USER" />

   </security:http>

   <security:authentication-manager alias="authenticationManager">

      <security:authentication-provider

         user-service-ref="userDetailsService"/>

   </security:authentication-manager>

   <bean id="userDetailsService"

      class="org.springframework.security.core.userdetails.jdbc.JdbcDaoImpl">

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

   </bean>

   <!-- 自己定义AccessDecisionManager对应的bean -->

   <bean id="accessDecisionManager"class="org.springframework.security.access.vote.AffirmativeBased">

      <property name="decisionVoters">

         <list>

            <ref local="roleVoter"/>

         </list>

      </property>

   </bean>

   <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"><!-- 角色继承关系 -->

         <value>

            ROLE\_ADMIN > ROLE\_USER

         </value>

      </property>

   </bean>

</beans>

       在上述配置中我们就定义好了ROLE\_ADMIN是继承自ROLE\_USER的，这样ROLE\_ADMIN将能够访问所有ROLE\_USER可以访问的资源。通过RoleHierarchyImpl的hierarchy属性我们可以定义多个角色之间的继承关系，如：

   <bean id="roleHierarchy"

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

      <property name="hierarchy"><!-- 角色继承关系 -->

         <value>

            ROLE\_ADMIN > ROLE\_USER

            ROLE\_A > ROLE\_B

            ROLE\_B > ROLE\_C

            ROLE\_C > ROLE\_D

         </value>

      </property>

   </bean>

       在上述配置我们同时定义了ROLE\_ADMIN继承了ROLE\_USER，ROLE\_A继承了ROLE\_B，ROLE\_B又继承了ROLE\_C，ROLE\_C又继承了ROLE\_D，这样ROLE\_A将能访问ROLE\_B、ROLE\_C和ROLE\_D所能访问的所有资源。

# [Spring Security（16）——基于表达式的权限控制](http://www.cnblogs.com/fenglan/p/5913463.html)

**目录**

1.1      通过表达式控制URL权限

1.2      通过表达式控制方法权限

1.2.1     使用@PreAuthorize和@PostAuthorize进行访问控制

1.2.2     使用@PreFilter和@PostFilter进行过滤

1.3      使用hasPermission表达式

       Spring Security允许我们在定义URL访问或方法访问所应有的权限时使用Spring EL表达式，在定义所需的访问权限时如果对应的表达式返回结果为true则表示拥有对应的权限，反之则无。Spring Security可用表达式对象的基类是SecurityExpressionRoot，其为我们提供了如下在使用Spring EL表达式对URL或方法进行权限控制时通用的内置表达式。

|  |  |
| --- | --- |
| **表达式** | **描述** |
| hasRole([role]) | 当前用户是否拥有指定角色。 |
| hasAnyRole([role1,role2]) | 多个角色是一个以逗号进行分隔的字符串。如果当前用户拥有指定角色中的任意一个则返回true。 |
| hasAuthority([auth]) | 等同于hasRole |
| hasAnyAuthority([auth1,auth2]) | 等同于hasAnyRole |
| Principle | 代表当前用户的principle对象 |
| authentication | 直接从SecurityContext获取的当前Authentication对象 |
| permitAll | 总是返回true，表示允许所有的 |
| denyAll | 总是返回false，表示拒绝所有的 |
| isAnonymous() | 当前用户是否是一个匿名用户 |
| isRememberMe() | 表示当前用户是否是通过Remember-Me自动登录的 |
| isAuthenticated() | 表示当前用户是否已经登录认证成功了。 |
| isFullyAuthenticated() | 如果当前用户既不是一个匿名用户，同时又不是通过Remember-Me自动登录的，则返回true。 |

## 1.1     通过表达式控制URL权限

       URL的访问权限是通过http元素下的intercept-url元素进行定义的，其access属性用来定义访问配置属性。默认情况下该属性值只能是以字符串进行分隔的字符串列表，且每一个元素都对应着一个角色，因为默认使用的是RoleVoter。通过设置http元素的use-expressions=”true”可以启用intercept-url元素的access属性对Spring EL表达式的支持，use-expressions的值默认为false。启用access属性对Spring EL表达式的支持后每个access属性值都应该是一个返回结果为boolean类型的表达式，当表达式返回结果为true时表示当前用户拥有访问权限。此外WebExpressionVoter将加入AccessDecisionManager的AccessDecisionVoter列表，所以如果不使用NameSpace时应当手动添加WebExpressionVoter到AccessDecisionVoter。

   <security:http use-expressions="true">

      <security:form-login/>

      <security:logout/>

      <security:intercept-url pattern="/\*\*" access="hasRole('ROLE\_USER')" />

   </security:http>

       在上述配置中我们定义了只有拥有ROLE\_USER角色的用户才能访问系统。

       使用表达式控制URL权限使用的表达式对象类是继承自SecurityExpressionRoot的WebSecurityExpressionRoot类。其相比基类而言新增了一个表达式hasIpAddress。hasIpAddress可用来限制只有指定IP或指定范围内的IP才可以访问。

   <security:http use-expressions="true">

      <security:form-login/>

      <security:logout/>

      <security:intercept-url pattern="/\*\*" access="hasRole('ROLE\_USER') and hasIpAddress('10.10.10.3')" />

   </security:http>

       在上面的配置中我们限制了只有IP为”10.10.10.3”，且拥有ROLE\_USER角色的用户才能访问。hasIpAddress是通过Ip地址或子网掩码来进行匹配的。如果要设置10.10.10下所有的子网都可以使用，那么我们对应的hasIpAddress的参数应为“10.10.10.n/24”，其中n可以是合法IP内的任意值。具体规则可以参照hasIpAddress()表达式用于比较的IpAddressMatcher的matches方法源码。以下是IpAddressMatcher的源码。

**package** org.springframework.security.web.util;

**import** java.net.InetAddress;

**import** java.net.UnknownHostException;

**import** java.util.Arrays;

**import** javax.servlet.http.HttpServletRequest;

**import** org.springframework.util.StringUtils;

/\*\*

 \* Matches a request based on IP Address or subnet mask matching against the remote address.

 \* <p>

 \* Both IPv6 and IPv4 addresses are supported, but a matcher which is configured with an IPv4 address will

 \* never match a request which returns an IPv6 address, and vice-versa.

 \*

 \* **@author** Luke Taylor

 \* **@since** 3.0.2

 \*/

**public final class** IpAddressMatcher **implements** RequestMatcher {

**private final int**nMaskBits;

**private final** InetAddress requiredAddress;

    /\*\*

     \* Takes a specific IP address or a range specified using the

     \* IP/Netmask (e.g. 192.168.1.0/24 or 202.24.0.0/14).

     \*

     \* **@param** ipAddress the address or range of addresses from which the request must come.

     \*/

**public** IpAddressMatcher(String ipAddress) {

**if** (ipAddress.indexOf('/') > 0) {

            String[] addressAndMask = StringUtils.split(ipAddress, "/");

            ipAddress = addressAndMask[0];

            nMaskBits = Integer.parseInt(addressAndMask[1]);

        } **else** {

            nMaskBits = -1;

        }

        requiredAddress = parseAddress(ipAddress);

    }

**public boolean** matches(HttpServletRequest request) {

**return** matches(request.getRemoteAddr());

    }

**public boolean** matches(String address) {

        InetAddress remoteAddress = parseAddress(address);

**if** (!requiredAddress.getClass().equals(remoteAddress.getClass())) {

**return false**;

        }

**if** (nMaskBits < 0) {

**return** remoteAddress.equals(requiredAddress);

        }

**byte**[] remAddr = remoteAddress.getAddress();

**byte**[] reqAddr = requiredAddress.getAddress();

**int** oddBits = nMaskBits % 8;

**int** nMaskBytes = nMaskBits/8 + (oddBits == 0 ? 0 : 1);

**byte**[] mask = **newbyte**[nMaskBytes];

        Arrays.fill(mask, 0, oddBits == 0 ? mask.length : mask.length - 1, (**byte**)0xFF);

**if** (oddBits != 0) {

**int** finalByte = (1 << oddBits) - 1;

            finalByte <<= 8-oddBits;

            mask[mask.length - 1] = (**byte**) finalByte;

        }

 //       System.out.println("Mask is " + new sun.misc.HexDumpEncoder().encode(mask));

**for** (**int** i=0; i < mask.length; i++) {

**if** ((remAddr[i] & mask[i]) != (reqAddr[i] & mask[i])) {

**return alse**;

            }

        }

**return true**;

    }

**private** InetAddress parseAddress(String address) {

**try** {

**return** InetAddress.getByName(address);

        } **catch** (UnknownHostException e) {

**thrownew** IllegalArgumentException("Failed to parse address" + address, e);

        }

    }

}

## 1.2     通过表达式控制方法权限

       Spring Security中定义了四个支持使用表达式的注解，分别是@PreAuthorize、@PostAuthorize、@PreFilter和@PostFilter。其中前两者可以用来在方法调用前或者调用后进行权限检查，后两者可以用来对集合类型的参数或者返回值进行过滤。要使它们的定义能够对我们的方法的调用产生影响我们需要设置global-method-security元素的pre-post-annotations=”enabled”，默认为disabled。

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

### 1.2.1使用@PreAuthorize和@PostAuthorize进行访问控制

       @PreAuthorize可以用来控制一个方法是否能够被调用。

@Service

**public class** UserServiceImpl **implements** UserService {

   @PreAuthorize("hasRole('ROLE\_ADMIN')")

**public void** addUser(User user) {

      System.out.println("addUser................" + user);

   }

   @PreAuthorize("hasRole('ROLE\_USER') or hasRole('ROLE\_ADMIN')")

**public** User find(**int** id) {

      System.out.println("find user by id............." + id);

**return null**;

   }

}

       在上面的代码中我们定义了只有拥有角色ROLE\_ADMIN的用户才能访问adduser()方法，而访问find()方法需要有ROLE\_USER角色或ROLE\_ADMIN角色。使用表达式时我们还可以在表达式中使用方法参数。

**public class** UserServiceImpl **implements** UserService {

   /\*\*

    \* 限制只能查询Id小于10的用户

    \*/

   @PreAuthorize("#id<10")

**public** User find(**int** id) {

      System.out.println("find user by id........." + id);

**return null**;

   }

   /\*\*

    \* 限制只能查询自己的信息

    \*/

   @PreAuthorize("principal.username.equals(#username)")

**public** User find(String username) {

      System.out.println("find user by username......" + username);

**return null**;

   }

   /\*\*

    \* 限制只能新增用户名称为abc的用户

    \*/

   @PreAuthorize("#user.name.equals('abc')")

**public void** add(User user) {

      System.out.println("addUser............" + user);

   }

}

       在上面代码中我们定义了调用find(int id)方法时，只允许参数id小于10的调用；调用find(String username)时只允许username为当前用户的用户名；定义了调用add()方法时只有当参数user的name为abc时才可以调用。

       有时候可能你会想在方法调用完之后进行权限检查，这种情况比较少，但是如果你有的话，Spring Security也为我们提供了支持，通过@PostAuthorize可以达到这一效果。使用@PostAuthorize时我们可以使用内置的表达式returnObject表示方法的返回值。我们来看下面这一段示例代码。

   @PostAuthorize("returnObject.id%2==0")

**public** User find(**int** id) {

      User user = **new** User();

      user.setId(id);

**return** user;

   }

       上面这一段代码表示将在方法find()调用完成后进行权限检查，如果返回值的id是偶数则表示校验通过，否则表示校验失败，将抛出AccessDeniedException。       需要注意的是@PostAuthorize是在方法调用完成后进行权限检查，它不能控制方法是否能被调用，只能在方法调用完成后检查权限决定是否要抛出AccessDeniedException。

### 1.2.2使用@PreFilter和@PostFilter进行过滤

       使用@PreFilter和@PostFilter可以对集合类型的参数或返回值进行过滤。使用@PreFilter和@PostFilter时，Spring Security将移除使对应表达式的结果为false的元素。

   @PostFilter("filterObject.id%2==0")

**public** List<User> findAll() {

      List<User> userList = **new** ArrayList<User>();

      User user;

**for** (**int** i=0; i<10; i++) {

         user = **new** User();

         user.setId(i);

         userList.add(user);

      }

**return** userList;

   }

       上述代码表示将对返回结果中id不为偶数的user进行移除。filterObject是使用@PreFilter和@PostFilter时的一个内置表达式，表示集合中的当前对象。当@PreFilter标注的方法拥有多个集合类型的参数时，需要通过@PreFilter的filterTarget属性指定当前@PreFilter是针对哪个参数进行过滤的。如下面代码就通过filterTarget指定了当前@PreFilter是用来过滤参数ids的。

   @PreFilter(filterTarget="ids", value="filterObject%2==0")

**public void** delete(List<Integer> ids, List<String> usernames) {

      ...

   }

## 1.3     使用hasPermission表达式

       Spring Security为我们定义了hasPermission的两种使用方式，它们分别对应着PermissionEvaluator的两个不同的hasPermission()方法。Spring Security默认处理Web、方法的表达式处理器分别为DefaultWebSecurityExpressionHandler和DefaultMethodSecurityExpressionHandler，它们都继承自AbstractSecurityExpressionHandler，其所持有的PermissionEvaluator是DenyAllPermissionEvaluator，其对于所有的hasPermission表达式都将返回false。所以当我们要使用表达式hasPermission时，我们需要自已手动定义SecurityExpressionHandler对应的bean定义，然后指定其PermissionEvaluator为我们自己实现的PermissionEvaluator，然后通过global-method-security元素或http元素下的expression-handler元素指定使用的SecurityExpressionHandler为我们自己手动定义的那个bean。

       接下来看一个自己实现PermissionEvaluator使用hasPermission()表达式的简单示例。

       首先实现自己的PermissionEvaluator，如下所示：

**public class** MyPermissionEvaluator **implements** PermissionEvaluator {

**public boolean** hasPermission(Authentication authentication,

         Object targetDomainObject, Object permission) {

**if** ("user".equals(targetDomainObject)) {

**return this**.hasPermission(authentication, permission);

      }

**return false**;

   }

   /\*\*

    \* 总是认为有权限

    \*/

**public boolean** hasPermission(Authentication authentication,

         Serializable targetId, String targetType, Object permission) {

**return true**;

   }

   /\*\*

    \* 简单的字符串比较，相同则认为有权限

    \*/

**private boolean** hasPermission(Authentication authentication, Object permission) {

      Collection<? **extends** GrantedAuthority> authorities = authentication.getAuthorities();

**for** (GrantedAuthority authority : authorities) {

**if** (authority.getAuthority().equals(permission)) {

**returntrue**;

         }

      }

**return false**;

   }

}

       接下来在ApplicationContext中显示的配置一个将使用PermissionEvaluator的SecurityExpressionHandler实现类，然后指定其所使用的PermissionEvaluator为我们自己实现的那个。这里我们选择配置一个针对于方法调用使用的表达式处理器，DefaultMethodSecurityExpressionHandler，具体如下所示。

   <bean id="expressionHandler"

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

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

   </bean>

   <!-- 自定义的PermissionEvaluator实现 -->

   <bean id="myPermissionEvaluator" class="com.xxx.MyPermissionEvaluator"/>

       有了SecurityExpressionHandler之后，我们还要告诉Spring Security，在使用SecurityExpressionHandler时应该使用我们显示配置的那个，这样我们自定义的PermissionEvaluator才能起作用。因为我们上面定义的是针对于方法的SecurityExpressionHandler，所以我们要指定在进行方法权限控制时应该使用它来进行处理，同时注意设置pre-post-annotations=”true”以启用对支持使用表达式的@PreAuthorize等注解的支持。

   <security:global-method-security

      pre-post-annotations="enabled">

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

   </security:global-method-security>

       之后我们就可以在需要进行权限控制的方法上使用@PreAuthorize以及hasPermission()表达式进行权限控制了。

@Service

**public class** UserServiceImpl **implements** UserService {

   /\*\*

    \* 将使用方法hasPermission(Authentication authentication,

         Object targetDomainObject, Object permission)进行验证。

    \*/

   @PreAuthorize("hasPermission('user', 'ROLE\_USER')")

**public** User find(**int** id) {

**return null**;

   }

   /\*\*

    \* 将使用PermissionEvaluator的第二个方法，即hasPermission(Authentication authentication,

         Serializable targetId, String targetType, Object permission)进行验证。

    \*/

   @PreAuthorize("hasPermission('targetId','targetType','permission')")

**public** User find(String username) {

**return null**;

   }

   @PreAuthorize("hasPermission('user', 'ROLE\_ADMIN')")

**public void** add(User user) {

   }

}

       在上面的配置中，find(int id)和add()方法将使用PermissionEvaluator中接收三个参数的hasPermission()方法进行验证，而find(String username)方法将使用四个参数的hasPermission()方法进行验证。因为hasPermission()表达式与PermissionEvaluator中hasPermission()方法的对应关系就是在hasPermission()表达式使用的参数基础上加上当前Authentication对象调用对应的hasPermission()方法进行验证。

       其实Spring Security已经针对于ACL实现了一个AclPermissionEvaluator。关于ACL的内容将在后文进行介绍。

**目录**

1.1     intercept-methods定义方法权限控制

1.2     使用pointcut定义方法权限控制

1.3     使用注解定义方法权限控制

1.3.1    JSR-250注解

1.3.2    @Secured注解

1.3.3    支持表达式的注解

1.4     方法权限控制的拦截器

1.4.1    MethodSecurityInterceptor

1.4.2    AspectJMethodSecurityInterceptor

       之前介绍的都是基于URL的权限控制，Spring Security同样支持对于方法的权限控制。可以通过intercept-methods对某个bean下面的方法进行权限控制，也可以通过pointcut对整个Service层的方法进行统一的权限控制，还可以通过注解定义对单独的某一个方法进行权限控制。

## 1.1     intercept-methods定义方法权限控制

       intercept-methods是需要定义在bean元素下的，通过它可以定义对当前的bean的某些方法进行权限控制，具体方法是使用其下的子元素protect进行定义的。protect元素需要指定两个属性，access和method，method表示需要拦截的方法名称，可以使用通配符，access表示执行对应的方法需要拥有的权限，多个权限之间可以使用逗号分隔。

   <bean id="userService" class="com.xxx.service.impl.UserServiceImpl">

      <security:intercept-methods>

         <security:protect access="ROLE\_USER" method="find\*"/>

         <security:protect access="ROLE\_ADMIN" method="add\*"/>

         <security:protect access="ROLE\_ADMIN" method="update\*"/>

         <security:protect access="ROLE\_ADMIN" method="delete\*"/>

      </security:intercept-methods>

   </bean>

       在上面的配置中表示在执行UserServiceImpl的方法名以find开始的方法时需要当前用户拥有ROLE\_USER的权限，在执行方法名以add、update或delete开始的方法时需要拥有ROLE\_ADMIN的权限。当访问被拒绝时还是交由ExceptionTranslationFilter处理，这也就意味着如果用户未登录则会引导用户进行登录，否则默认将返回403错误码到客户端。

## 1.2     使用pointcut定义方法权限控制

       基于pointcut的方法权限控制是通过global-method-security下的protect-pointcut来定义的。可以在global-method-security元素下定义多个protect-pointcut以对不同的pointcut使用不同的权限控制。

   <security:global-method-security>

      <security:protect-pointcut access="ROLE\_READ" expression="execution(\* com.elim.\*..\*Service.find\*(..))"/>

      <security:protect-pointcut access="ROLE\_WRITE" expression="execution(\* com.elim.\*..\*Service.\*(..))"/>

   </security:global-method-security>

       上面的定义表示我们在执行com.elim包或其子包下任意以Service结尾的类，其方法名以find开始的所有方法时都需要用户拥有ROLE\_READ的权限，对于com.elim包或其子包下任意以Service结尾的类的其它方法在执行时都需要ROLE\_WRITE的权限。需要注意的是对应的类需要是定义在ApplicationContext中的bean才行。此外同对于URL的权限控制一样，当定义多个protect-pointcut时更具有特性的应当先定义，因为在pointcut匹配的时候是按照声明顺序进行匹配的，一旦匹配上了后续的将不再进行匹配了。

## 1.3     使用注解定义方法权限控制

       基于注解的方法权限控制也是需要通过global-method-security元素定义来进行启用的。Spring Security在方法的权限控制上支持三种类型的注解，JSR-250注解、@Secured注解和支持表达式的注解。这三种注解默认都是没有启用的，需要单独通过global-method-security元素的对应属性进行启用。

### 1.3.1   JSR-250注解

       要使用JSR-250注解，首先我们需要通过设置global-method-security元素的jsr250-annotation=”enabled”来启用基于JSR-250注解的支持，默认为disabled。

       <security:global-method-security jsr250-annotations="enabled"/>

       此外，还需要确保添加了jsr250-api到我们的类路径下。之后就可以在我们的Service方法上使用JSR-250注解进行权限控制了。

@Service

@RolesAllowed("ROLE\_ADMIN")

**public class** UserServiceImpl **implements** UserService {

**public void** addUser(User user) {

      System.out.println("addUser................" + user);

   }

**public void** updateUser(User user) {

      System.out.println("updateUser.............." + user);

   }

   @RolesAllowed({"ROLE\_USER", "ROLE\_ADMIN"})

**public** User find(**int** id) {

      System.out.println("find user by id............." + id);

**return null**;

   }

**public void** delete(**int** id) {

      System.out.println("delete user by id................");

   }

   @RolesAllowed("ROLE\_USER")

**public** List<User> findAll() {

      System.out.println("find all user...............");

**return null**;

   }

}

       上面的代码表示执行UserServiceImpl里面所有的方法都需要角色ROLE\_ADMIN，其中findAll()方法的执行需要ROLE\_USER角色，而find()方法的执行对于ROLE\_USER或者ROLE\_ADMIN角色都可以。

       顺便介绍一下JSR-250中对权限支持的注解。

**RolesAllowed**表示访问对应方法时所应该具有的角色。其可以标注在类上，也可以标注在方法上，当标注在类上时表示其中所有方法的执行都需要对应的角色，当标注在方法上表示执行该方法时所需要的角色，当方法和类上都使用了@RolesAllowed进行标注，则方法上的@RolesAllowed将覆盖类上的@RolesAllowed，即方法上的@RolesAllowed将对当前方法起作用。@RolesAllowed的值是由角色名称组成的数组。

**PermitAll**表示允许所有的角色进行访问，也就是说不进行权限控制。@PermitAll可以标注在方法上也可以标注在类上，当标注在方法上时则只对对应方法不进行权限控制，而标注在类上时表示对类里面所有的方法都不进行权限控制。（1）当@PermitAll标注在类上，而@RolesAllowed标注在方法上时则按照@RolesAllowed将覆盖@PermitAll，即需要@RolesAllowed对应的角色才能访问。（2）当@RolesAllowed标注在类上，而@PermitAll标注在方法上时则对应的方法也是不进行权限控制的。（3）当在方法上同时使用了@PermitAll和@RolesAllowed时先定义的将发生作用，而都定义在类上时则是反过来的，即后定义的将发生作用（这个没多大的实际意义，实际应用中不会有这样的定义）。

**DenyAll**是和PermitAll相反的，表示无论什么角色都不能访问。@DenyAll只能定义在方法上。你可能会有疑问使用@DenyAll标注的方法无论拥有什么权限都不能访问，那还定义它干啥呢？使用@DenyAll定义的方法只是在我们的权限控制中不能访问，脱离了权限控制还是可以访问的。

### 1.3.2  @Secured注解

       @Secured是由Spring Security定义的用来支持方法权限控制的注解。它的使用也是需要启用对应的支持才会生效的。通过设置global-method-security元素的secured-annotations=”enabled”可以启用Spring Security对使用@Secured注解标注的方法进行权限控制的支持，其值默认为disabled。

   <security:global-method-security secured-annotations="enabled"/>

@Service

**public class** UserServiceImpl **implements** UserService {

   @Secured("ROLE\_ADMIN")

**public void** addUser(User user) {

      System.out.println("addUser................" + user);

   }

   @Secured("ROLE\_USER")

**public** List<User> findAll() {

      System.out.println("find all user...............");

**return null**;

   }

}

       在上面的代码中我们使用@Secured定义了只有拥有ROLE\_ADMIN角色的用户才能调用方法addUser()，只有拥有ROLE\_USER角色的用户才能调用方法findAll()。

### 1.3.3   支持表达式的注解

       Spring Security中定义了四个支持使用表达式的注解，分别是@PreAuthorize、@PostAuthorize、@PreFilter和@PostFilter。其中前两者可以用来在方法调用前或者调用后进行权限检查，后两者可以用来对集合类型的参数或者返回值进行过滤。要使它们的定义能够对我们的方法的调用产生影响我们需要设置global-method-security元素的pre-post-annotations=”enabled”，默认为disabled。

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

**使用@PreAuthorize和@PostAuthorize进行访问控制**

       @PreAuthorize可以用来控制一个方法是否能够被调用。

@Service

**public class** UserServiceImpl **implements** UserService {

   @PreAuthorize("hasRole('ROLE\_ADMIN')")

**public void** addUser(User user) {

      System.out.println("addUser................" + user);

   }

   @PreAuthorize("hasRole('ROLE\_USER') or hasRole('ROLE\_ADMIN')")

**public** User find(**int** id) {

      System.out.println("find user by id............." + id);

**return null**;

   }

}

       在上面的代码中我们定义了只有拥有角色ROLE\_ADMIN的用户才能访问adduser()方法，而访问find()方法需要有ROLE\_USER角色或ROLE\_ADMIN角色。使用表达式时我们还可以在表达式中使用方法参数。

**public class** UserServiceImpl **implements** UserService {

   /\*\*

    \* 限制只能查询Id小于10的用户

    \*/

   @PreAuthorize("#id<10")

**public** User find(**int** id) {

      System.out.println("find user by id........." + id);

**return null**;

   }

   /\*\*

    \* 限制只能查询自己的信息

    \*/

   @PreAuthorize("principal.username.equals(#username)")

**public** User find(String username) {

      System.out.println("find user by username......" + username);

**return null**;

   }

   /\*\*

    \* 限制只能新增用户名称为abc的用户

    \*/

   @PreAuthorize("#user.name.equals('abc')")

**public void** add(User user) {

      System.out.println("addUser............" + user);

   }

}

       在上面代码中我们定义了调用find(int id)方法时，只允许参数id小于10的调用；调用find(String username)时只允许username为当前用户的用户名；定义了调用add()方法时只有当参数user的name为abc时才可以调用。

       有时候可能你会想在方法调用完之后进行权限检查，这种情况比较少，但是如果你有的话，Spring Security也为我们提供了支持，通过@PostAuthorize可以达到这一效果。使用@PostAuthorize时我们可以使用内置的表达式returnObject表示方法的返回值。我们来看下面这一段示例代码。

   @PostAuthorize("returnObject.id%2==0")

**public** User find(**int** id) {

      User user = **new** User();

      user.setId(id);

**return** user;

   }

       上面这一段代码表示将在方法find()调用完成后进行权限检查，如果返回值的id是偶数则表示校验通过，否则表示校验失败，将抛出AccessDeniedException。       需要注意的是@PostAuthorize是在方法调用完成后进行权限检查，它不能控制方法是否能被调用，只能在方法调用完成后检查权限决定是否要抛出AccessDeniedException。

**使用@PreFilter和@PostFilter进行过滤**

       使用@PreFilter和@PostFilter可以对集合类型的参数或返回值进行过滤。使用@PreFilter和@PostFilter时，Spring Security将移除使对应表达式的结果为false的元素。

   @PostFilter("filterObject.id%2==0")

**public** List<User> findAll() {

      List<User> userList = **new** ArrayList<User>();

      User user;

**for** (**int** i=0; i<10; i++) {

         user = **new** User();

         user.setId(i);

         userList.add(user);

      }

**return** userList;

   }

       上述代码表示将对返回结果中id不为偶数的user进行移除。filterObject是使用@PreFilter和@PostFilter时的一个内置表达式，表示集合中的当前对象。当@PreFilter标注的方法拥有多个集合类型的参数时，需要通过@PreFilter的filterTarget属性指定当前@PreFilter是针对哪个参数进行过滤的。如下面代码就通过filterTarget指定了当前@PreFilter是用来过滤参数ids的。

   @PreFilter(filterTarget="ids", value="filterObject%2==0")

**public void** delete(List<Integer> ids, List<String> usernames) {

      ...

   }

## 1.4     方法权限控制的拦截器

       关于方法权限控制，Spring Security提供了两类AbstractSecurityInterceptor，基于AOP Alliance的MethodSecurityInterceptor，和基于Aspectj继承自MethodSecurityInterceptor的AspectJMethodSecurityInterceptor。

### 1.4.1   MethodSecurityInterceptor

       当我们在使用基于NameSpace进行方法保护的配置时，Spring Security默认配置的就是MethodSecurityInterceptor。根据配置的不同，一个拦截器可能只是针对于一个bean，也可能是针对于多个bean的。MethodSecurityInterceptor使用一个MethodSecurityMetadataSource的实例来获取特定方法调用配置的ConfigAttribute。当我们在ApplicationContext配置文件中使用intercept-methods元素或protect-point元素定义需要保护的方法调用时，Spring Security内部默认会使用一个MapBasedMethodSecurityMetadataSource来保存在这些元素上定义的配置信息，保存的key是对应的方法名（可以是含有通配符的）。类似的使用JSR-250注解时将使用Jsr250MethodSecurityMetadataSource解析配置属性；使用@Secured注解时将使用SecuredAnnotationSecurityMetadataSource解析配置属性；使用pre-post-annotations时将使用PrePostAnnotationSecurityMetadataSource解析配置属性。

       MethodSecurityInterceptor是实现了MethodInterceptor接口的，所以我们在使用Spring Aop时，可以自己配置一个MethodSecurityInterceptor的bean。

   <!-- 自定义MethodSecurityInterceptor -->

   <bean id="methodSecurityInterceptor"

  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">

         <security:method-security-metadata-source>

            <!-- 指定需要受保护的方法和需要的权限 -->

            <security:protect method="com.xxx.service.UserService.find\*"

                access="ROLE\_USER" />

            <security:protect method="com.xxx.service.UserService.delete\*"

                access="ROLE\_ADMIN" />

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

      </property>

   </bean>

       定义了MethodSecurityInterceptor以后，我们需要类似AOP配置那样，配置哪些该MethodInterceptor需要拦截哪些方法的执行。这种可选配置是很多种的，因为我们这里只是拦截UserService中的具体方法，所以就采用基于bean name的自动代理。

   <!-- 基于bean的拦截 -->

   <bean class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">

      <property name="interceptorNames">

         <list>

            <value>methodSecurityInterceptor</value>

         </list>

      </property>

      <property name="beanNames">

         <list>

            <value>userService</value>

         </list>

      </property>

   </bean>

       按照上面的配置，我们在访问UserService的find方法时就需要ROLE\_USER的权限，而访问delete方法时则需要ROLE\_ADMIN权限。

### 1.4.2   AspectJMethodSecurityInterceptor

       AspectJMethodSecurityInterceptor是继承自MethodSecurityInterceptor的，不同的是AspectJMethodSecurityInterceptor是用来支持AspectJ的JointPoint的，但在底层还是会把它封装成一个MethodInvocation进行调用。

# [Spring Security（17）——基于方法的权限控制](http://www.cnblogs.com/fenglan/p/5913481.html)

**目录**

1.1     intercept-methods定义方法权限控制

1.2     使用pointcut定义方法权限控制

1.3     使用注解定义方法权限控制

1.3.1    JSR-250注解

1.3.2    @Secured注解

1.3.3    支持表达式的注解

1.4     方法权限控制的拦截器

1.4.1    MethodSecurityInterceptor

1.4.2    AspectJMethodSecurityInterceptor

       之前介绍的都是基于URL的权限控制，Spring Security同样支持对于方法的权限控制。可以通过intercept-methods对某个bean下面的方法进行权限控制，也可以通过pointcut对整个Service层的方法进行统一的权限控制，还可以通过注解定义对单独的某一个方法进行权限控制。

## 1.1     intercept-methods定义方法权限控制

       intercept-methods是需要定义在bean元素下的，通过它可以定义对当前的bean的某些方法进行权限控制，具体方法是使用其下的子元素protect进行定义的。protect元素需要指定两个属性，access和method，method表示需要拦截的方法名称，可以使用通配符，access表示执行对应的方法需要拥有的权限，多个权限之间可以使用逗号分隔。

   <bean id="userService" class="com.xxx.service.impl.UserServiceImpl">

      <security:intercept-methods>

         <security:protect access="ROLE\_USER" method="find\*"/>

         <security:protect access="ROLE\_ADMIN" method="add\*"/>

         <security:protect access="ROLE\_ADMIN" method="update\*"/>

         <security:protect access="ROLE\_ADMIN" method="delete\*"/>

      </security:intercept-methods>

   </bean>

       在上面的配置中表示在执行UserServiceImpl的方法名以find开始的方法时需要当前用户拥有ROLE\_USER的权限，在执行方法名以add、update或delete开始的方法时需要拥有ROLE\_ADMIN的权限。当访问被拒绝时还是交由ExceptionTranslationFilter处理，这也就意味着如果用户未登录则会引导用户进行登录，否则默认将返回403错误码到客户端。

## 1.2     使用pointcut定义方法权限控制

       基于pointcut的方法权限控制是通过global-method-security下的protect-pointcut来定义的。可以在global-method-security元素下定义多个protect-pointcut以对不同的pointcut使用不同的权限控制。

   <security:global-method-security>

      <security:protect-pointcut access="ROLE\_READ" expression="execution(\* com.elim.\*..\*Service.find\*(..))"/>

      <security:protect-pointcut access="ROLE\_WRITE" expression="execution(\* com.elim.\*..\*Service.\*(..))"/>

   </security:global-method-security>

       上面的定义表示我们在执行com.elim包或其子包下任意以Service结尾的类，其方法名以find开始的所有方法时都需要用户拥有ROLE\_READ的权限，对于com.elim包或其子包下任意以Service结尾的类的其它方法在执行时都需要ROLE\_WRITE的权限。需要注意的是对应的类需要是定义在ApplicationContext中的bean才行。此外同对于URL的权限控制一样，当定义多个protect-pointcut时更具有特性的应当先定义，因为在pointcut匹配的时候是按照声明顺序进行匹配的，一旦匹配上了后续的将不再进行匹配了。

## 1.3     使用注解定义方法权限控制

       基于注解的方法权限控制也是需要通过global-method-security元素定义来进行启用的。Spring Security在方法的权限控制上支持三种类型的注解，JSR-250注解、@Secured注解和支持表达式的注解。这三种注解默认都是没有启用的，需要单独通过global-method-security元素的对应属性进行启用。

### 1.3.1   JSR-250注解

       要使用JSR-250注解，首先我们需要通过设置global-method-security元素的jsr250-annotation=”enabled”来启用基于JSR-250注解的支持，默认为disabled。

       <security:global-method-security jsr250-annotations="enabled"/>

       此外，还需要确保添加了jsr250-api到我们的类路径下。之后就可以在我们的Service方法上使用JSR-250注解进行权限控制了。

@Service

@RolesAllowed("ROLE\_ADMIN")

**public class** UserServiceImpl **implements** UserService {

**public void** addUser(User user) {

      System.out.println("addUser................" + user);

   }

**public void** updateUser(User user) {

      System.out.println("updateUser.............." + user);

   }

   @RolesAllowed({"ROLE\_USER", "ROLE\_ADMIN"})

**public** User find(**int** id) {

      System.out.println("find user by id............." + id);

**return null**;

   }

**public void** delete(**int** id) {

      System.out.println("delete user by id................");

   }

   @RolesAllowed("ROLE\_USER")

**public** List<User> findAll() {

      System.out.println("find all user...............");

**return null**;

   }

}

       上面的代码表示执行UserServiceImpl里面所有的方法都需要角色ROLE\_ADMIN，其中findAll()方法的执行需要ROLE\_USER角色，而find()方法的执行对于ROLE\_USER或者ROLE\_ADMIN角色都可以。

       顺便介绍一下JSR-250中对权限支持的注解。

**RolesAllowed**表示访问对应方法时所应该具有的角色。其可以标注在类上，也可以标注在方法上，当标注在类上时表示其中所有方法的执行都需要对应的角色，当标注在方法上表示执行该方法时所需要的角色，当方法和类上都使用了@RolesAllowed进行标注，则方法上的@RolesAllowed将覆盖类上的@RolesAllowed，即方法上的@RolesAllowed将对当前方法起作用。@RolesAllowed的值是由角色名称组成的数组。

**PermitAll**表示允许所有的角色进行访问，也就是说不进行权限控制。@PermitAll可以标注在方法上也可以标注在类上，当标注在方法上时则只对对应方法不进行权限控制，而标注在类上时表示对类里面所有的方法都不进行权限控制。（1）当@PermitAll标注在类上，而@RolesAllowed标注在方法上时则按照@RolesAllowed将覆盖@PermitAll，即需要@RolesAllowed对应的角色才能访问。（2）当@RolesAllowed标注在类上，而@PermitAll标注在方法上时则对应的方法也是不进行权限控制的。（3）当在方法上同时使用了@PermitAll和@RolesAllowed时先定义的将发生作用，而都定义在类上时则是反过来的，即后定义的将发生作用（这个没多大的实际意义，实际应用中不会有这样的定义）。

**DenyAll**是和PermitAll相反的，表示无论什么角色都不能访问。@DenyAll只能定义在方法上。你可能会有疑问使用@DenyAll标注的方法无论拥有什么权限都不能访问，那还定义它干啥呢？使用@DenyAll定义的方法只是在我们的权限控制中不能访问，脱离了权限控制还是可以访问的。

### 1.3.2  @Secured注解

       @Secured是由Spring Security定义的用来支持方法权限控制的注解。它的使用也是需要启用对应的支持才会生效的。通过设置global-method-security元素的secured-annotations=”enabled”可以启用Spring Security对使用@Secured注解标注的方法进行权限控制的支持，其值默认为disabled。

   <security:global-method-security secured-annotations="enabled"/>

@Service

**public class** UserServiceImpl **implements** UserService {

   @Secured("ROLE\_ADMIN")

**public void** addUser(User user) {

      System.out.println("addUser................" + user);

   }

   @Secured("ROLE\_USER")

**public** List<User> findAll() {

      System.out.println("find all user...............");

**return null**;

   }

}

       在上面的代码中我们使用@Secured定义了只有拥有ROLE\_ADMIN角色的用户才能调用方法addUser()，只有拥有ROLE\_USER角色的用户才能调用方法findAll()。

### 1.3.3   支持表达式的注解

       Spring Security中定义了四个支持使用表达式的注解，分别是@PreAuthorize、@PostAuthorize、@PreFilter和@PostFilter。其中前两者可以用来在方法调用前或者调用后进行权限检查，后两者可以用来对集合类型的参数或者返回值进行过滤。要使它们的定义能够对我们的方法的调用产生影响我们需要设置global-method-security元素的pre-post-annotations=”enabled”，默认为disabled。

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

**使用@PreAuthorize和@PostAuthorize进行访问控制**

       @PreAuthorize可以用来控制一个方法是否能够被调用。

@Service

**public class** UserServiceImpl **implements** UserService {

   @PreAuthorize("hasRole('ROLE\_ADMIN')")

**public void** addUser(User user) {

      System.out.println("addUser................" + user);

   }

   @PreAuthorize("hasRole('ROLE\_USER') or hasRole('ROLE\_ADMIN')")

**public** User find(**int** id) {

      System.out.println("find user by id............." + id);

**return null**;

   }

}

       在上面的代码中我们定义了只有拥有角色ROLE\_ADMIN的用户才能访问adduser()方法，而访问find()方法需要有ROLE\_USER角色或ROLE\_ADMIN角色。使用表达式时我们还可以在表达式中使用方法参数。

**public class** UserServiceImpl **implements** UserService {

   /\*\*

    \* 限制只能查询Id小于10的用户

    \*/

   @PreAuthorize("#id<10")

**public** User find(**int** id) {

      System.out.println("find user by id........." + id);

**return null**;

   }

   /\*\*

    \* 限制只能查询自己的信息

    \*/

   @PreAuthorize("principal.username.equals(#username)")

**public** User find(String username) {

      System.out.println("find user by username......" + username);

**return null**;

   }

   /\*\*

    \* 限制只能新增用户名称为abc的用户

    \*/

   @PreAuthorize("#user.name.equals('abc')")

**public void** add(User user) {

      System.out.println("addUser............" + user);

   }

}

       在上面代码中我们定义了调用find(int id)方法时，只允许参数id小于10的调用；调用find(String username)时只允许username为当前用户的用户名；定义了调用add()方法时只有当参数user的name为abc时才可以调用。

       有时候可能你会想在方法调用完之后进行权限检查，这种情况比较少，但是如果你有的话，Spring Security也为我们提供了支持，通过@PostAuthorize可以达到这一效果。使用@PostAuthorize时我们可以使用内置的表达式returnObject表示方法的返回值。我们来看下面这一段示例代码。

   @PostAuthorize("returnObject.id%2==0")

**public** User find(**int** id) {

      User user = **new** User();

      user.setId(id);

**return** user;

   }

       上面这一段代码表示将在方法find()调用完成后进行权限检查，如果返回值的id是偶数则表示校验通过，否则表示校验失败，将抛出AccessDeniedException。       需要注意的是@PostAuthorize是在方法调用完成后进行权限检查，它不能控制方法是否能被调用，只能在方法调用完成后检查权限决定是否要抛出AccessDeniedException。

**使用@PreFilter和@PostFilter进行过滤**

       使用@PreFilter和@PostFilter可以对集合类型的参数或返回值进行过滤。使用@PreFilter和@PostFilter时，Spring Security将移除使对应表达式的结果为false的元素。

   @PostFilter("filterObject.id%2==0")

**public** List<User> findAll() {

      List<User> userList = **new** ArrayList<User>();

      User user;

**for** (**int** i=0; i<10; i++) {

         user = **new** User();

         user.setId(i);

         userList.add(user);

      }

**return** userList;

   }

       上述代码表示将对返回结果中id不为偶数的user进行移除。filterObject是使用@PreFilter和@PostFilter时的一个内置表达式，表示集合中的当前对象。当@PreFilter标注的方法拥有多个集合类型的参数时，需要通过@PreFilter的filterTarget属性指定当前@PreFilter是针对哪个参数进行过滤的。如下面代码就通过filterTarget指定了当前@PreFilter是用来过滤参数ids的。

   @PreFilter(filterTarget="ids", value="filterObject%2==0")

**public void** delete(List<Integer> ids, List<String> usernames) {

      ...

   }

## 1.4     方法权限控制的拦截器

       关于方法权限控制，Spring Security提供了两类AbstractSecurityInterceptor，基于AOP Alliance的MethodSecurityInterceptor，和基于Aspectj继承自MethodSecurityInterceptor的AspectJMethodSecurityInterceptor。

### 1.4.1   MethodSecurityInterceptor

       当我们在使用基于NameSpace进行方法保护的配置时，Spring Security默认配置的就是MethodSecurityInterceptor。根据配置的不同，一个拦截器可能只是针对于一个bean，也可能是针对于多个bean的。MethodSecurityInterceptor使用一个MethodSecurityMetadataSource的实例来获取特定方法调用配置的ConfigAttribute。当我们在ApplicationContext配置文件中使用intercept-methods元素或protect-point元素定义需要保护的方法调用时，Spring Security内部默认会使用一个MapBasedMethodSecurityMetadataSource来保存在这些元素上定义的配置信息，保存的key是对应的方法名（可以是含有通配符的）。类似的使用JSR-250注解时将使用Jsr250MethodSecurityMetadataSource解析配置属性；使用@Secured注解时将使用SecuredAnnotationSecurityMetadataSource解析配置属性；使用pre-post-annotations时将使用PrePostAnnotationSecurityMetadataSource解析配置属性。

       MethodSecurityInterceptor是实现了MethodInterceptor接口的，所以我们在使用Spring Aop时，可以自己配置一个MethodSecurityInterceptor的bean。

   <!-- 自定义MethodSecurityInterceptor -->

   <bean id="methodSecurityInterceptor"

  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">

         <security:method-security-metadata-source>

            <!-- 指定需要受保护的方法和需要的权限 -->

            <security:protect method="com.xxx.service.UserService.find\*"

                access="ROLE\_USER" />

            <security:protect method="com.xxx.service.UserService.delete\*"

                access="ROLE\_ADMIN" />

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

      </property>

   </bean>

       定义了MethodSecurityInterceptor以后，我们需要类似AOP配置那样，配置哪些该MethodInterceptor需要拦截哪些方法的执行。这种可选配置是很多种的，因为我们这里只是拦截UserService中的具体方法，所以就采用基于bean name的自动代理。

   <!-- 基于bean的拦截 -->

   <bean class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">

      <property name="interceptorNames">

         <list>

            <value>methodSecurityInterceptor</value>

         </list>

      </property>

      <property name="beanNames">

         <list>

            <value>userService</value>

         </list>

      </property>

   </bean>

       按照上面的配置，我们在访问UserService的find方法时就需要ROLE\_USER的权限，而访问delete方法时则需要ROLE\_ADMIN权限。

### 1.4.2   AspectJMethodSecurityInterceptor

       AspectJMethodSecurityInterceptor是继承自MethodSecurityInterceptor的，不同的是AspectJMethodSecurityInterceptor是用来支持AspectJ的JointPoint的，但在底层还是会把它封装成一个MethodInvocation进行调用。

# [Spring Security（18）——Jsp标签](http://www.cnblogs.com/fenglan/p/5913502.html)

**目录**

1.1     authorize

1.2     authentication

1.3     accesscontrollist

       Spring Security也有对Jsp标签的支持的标签库。其中一共定义了三个标签：authorize、authentication和accesscontrollist。其中authentication标签是用来代表当前Authentication对象的，我们可以利用它来展示当前Authentication对象的相关信息。另外两个标签是用于权限控制的，可以利用它们来包裹需要保护的内容，通常是超链接和按钮。

       如果需要使用Spring Security的标签库，那么首先我们应当将对应的jar包spring-security-taglibs-xxx.jar放入WEB-INF/lib下；其次我们需要在页面上引入Spring Security的标签库。

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

       接下来就可以在页面上自由的使用Spring Security的标签库提供的标签了。

## 1.1     authorize

       authorize是用来判断普通权限的，通过判断用户是否具有对应的权限而控制其所包含内容的显示，其可以指定如下属性。

       1、**access**

       access属性需要使用表达式来判断权限，当表达式的返回结果为true时表示拥有对应的权限。

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

      <a href="admin.jsp">admin page</a>

   </sec:authorize>

       需要注意的是因为access属性是使用表达式的，所以我们必须确保ApplicationContext中存在一个WebSecurityExpressionHandler，最简单的办法就是直接使用NameSpace，通过设置http元素的use-expressions="true"让NameSpace自动为我们创建一个WebSecurityExpressionHandler。

       2、**ifAllGranted、ifAnyGranted和ifNotGranted**

       这三个属性的用法类似，它们都接收以逗号分隔的权限列表，且不能使用表达式。ifAllGranted表示需要包含所有的权限，ifAnyGranted表示只需要包含其中的任意一个即可，ifNotGranted表示不能包含指定的任意一个权限。

   <!-- 需要拥有所有的权限 -->

   <sec:authorize ifAllGranted="ROLE\_ADMIN">

      <a href="admin.jsp">admin</a>

   </sec:authorize>

   <!-- 只需拥有其中任意一个权限 -->

   <sec:authorize ifAnyGranted="ROLE\_USER,ROLE\_ADMIN">hello</sec:authorize>

   <!-- 不允许拥有指定的任意权限 -->

   <sec:authorize ifNotGranted="ROLE\_ADMIN">

      <a href="user.jsp">user</a>

   </sec:authorize>

       3、**url**

       url表示如果用户拥有访问指定url的权限即表示可以显示authorize标签包含的内容。

   <!-- 拥有访问指定url的权限才显示其中包含的内容 -->

   <sec:authorize url="/admin.jsp">

      <a href="admin.jsp">admin</a>

   </sec:authorize>

       4、**method**

       method属性是配合url属性一起使用的，表示用户应当具有指定url指定method访问的权限，method的默认值为GET，可选值为http请求的7种方法。

   <!-- 拥有访问指定url的权限才显示其中包含的内容 -->

   <sec:authorize url="/admin.jsp">

      <a href="admin.jsp">admin</a>

   </sec:authorize>

       限制访问方法是通过http元素下的intercept-url元素的method属性来指定的，如：

   <security:intercept-url pattern="/admin.jsp" access="ROLE\_ADMIN" method="POST"/>

       5、**var**

       用于指定将权限鉴定的结果存放在pageContext的哪个属性中。该属性的主要作用是对于在同一页面的多个地方具有相同权限鉴定时，我们只需要定义一次，然后将鉴定结果以var指定的属性名存放在pageContext中，其它地方可以直接使用之前的鉴定结果。

   <sec:authorize access="isFullyAuthenticated()" var="isFullyAuthenticated">

      只有通过登录界面进行登录的用户才能看到1。<br/>

   </sec:authorize>

   上述权限的鉴定结果是：${isFullyAuthenticated }<br/>

   <%**if**((Boolean)pageContext.getAttribute("isFullyAuthenticated")) {%>

      只有通过登录界面进行登录的用户才能看到2。

   <%}%>

**各属性对应的优先级**

       既然我们可以通过属性access、url、ifAllGranted、ifAnyGranted等来指定应当具有的权限，那么当同时指定多个属性时，它们的作用效果是什么样的呢？authorize标签进行权限鉴定的属性根据优先级的不同可以分为三类，access为一类；url为一类；ifAllGranted、ifAnyGranted和ifNotGranted为一类。这三类将同时只有一类产生效果。它们的优先级如下：

       1、access具有最高的优先级，如果指定了access属性，那么将以access属性指定的表达式来鉴定当前用户是否有权限。不管结果如何，此时其它属性都将被忽略。

       2、如果没有指定access属性，那么url属性将具有最高优先级，此时将直接通过url属性和method属性（默认为GET）来鉴定当前用户是否有权限。不管结果如何，此时都将忽略ifAllGranted、ifAnyGranted和ifNotGranted属性。

       3、如果access和url都没有指定，那么将使用第三类属性来鉴定当前用户的权限。当第三类里面同时指定了多个属性时，它们将都发生效果，即必须指定的三类权限都满足才认为是有对应的权限。如ifAllGranted要求有ROLE\_USER的权限，同时ifNotGranted要求不能有ROLE\_ADMIN的权限，则结果是它们的并集，即只有拥有ROLE\_USER权限，同时不拥有ROLE\_ADMIN权限的用户才被允许获取指定的内容。

## 1.2     authentication

       authentication标签用来代表当前Authentication对象，主要用于获取当前Authentication的相关信息。authentication标签的主要属性是property属性，我们可以通过它来获取当前Authentication对象的相关信息。如通常我们的Authentication对象中存放的principle是一个UserDetails对象，所以我们可以通过如下的方式来获取当前用户的用户名。

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

       当然，我们也可以直接通过Authentication的name属性来获取其用户名。

   <sec:authentication property="name"/>

     property属性只允许指定Authentication所拥有的属性，可以进行属性的级联获取，如“principle.username”，不允许直接通过方法进行调用。

       除了property属性之外，authentication还可以指定的属性有：var、scope和htmlScape。

**var属性**

       var属性用于指定一个属性名，这样当获取到了authentication的相关信息后会将其以var指定的属性名进行存放，默认是存放在pageConext中。可以通过scope属性进行指定。此外，当指定了var属性后，authentication标签不会将获取到的信息在页面上进行展示，如需展示用户应该通过var指定的属性进行展示，或去掉var属性。

   <!-- 将获取到的用户名以属性名username存放在session中 -->

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

   ${username }

**scope属性**

       与var属性一起使用，用于指定存放获取的结果的属性名的作用范围，默认我pageContext。Jsp中拥有的作用范围都进行进行指定。

**htmlScape属性**

       表示是否需要将html进行转义。默认为true。

## 1.3     accesscontrollist

       accesscontrollist标签是用于鉴定ACL权限的。其一共定义了三个属性：hasPermission、domainObject和var，其中前两个是必须指定的。hasPermission属性用于指定以逗号分隔的权限列表；domainObject用于指定对应的域对象；而var则是用以将鉴定的结果以指定的属性名存入pageContext中，以供同一页面的其它地方使用。需要注意的是使用accesscontrollist标签时ApplicationContext中必须存在一个PermissionEvaluator bean，因为accesscontrollist标签就是通过PermissionEvaluator来鉴定对应的权限的。如果我们正在使用Spring Security的ACL模块，那么PermissionEvaluator通常就对应着AclPermissionEvaluator。此外，如果domainObject属性指定的domainObject为null则默认认为是有权限的，否则如果当前Authentication对象为null则默认认为是没有权限的。

   <sec:accesscontrollist hasPermission="1,2" domainObject="${someTargetDomainObject }" >

      如果当前Authentication对指定的domainObject拥有指定的hasPermission则将可以看到这部分内容。

   </sec:accesscontrollist>

# [Spring Security（19）——对Acl的支持](http://www.cnblogs.com/fenglan/p/5913540.html)

**目录**

1.1           准备工作

1.2           表功能介绍

1.2.1     表acl\_sid

1.2.2     表acl\_class

1.2.3     表acl\_object\_identity

1.2.4     表acl\_entry

1.3           Acl主要接口

1.4           配置AclService

1.4.1     配置DataSource

1.4.2     配置LookupStrategy

1.4.3     配置AclAuthorizationStrategy

1.4.4     配置grantingStrategy

1.4.5     配置AclCache

1.5           使用AclService

1.5.1     创建Acl

1.5.2     查找Acl

1.5.3     更新Acl

1.5.4     删除Acl

1.6           注入到AclPermissionEvaluator

       Acl的全称是Access Control List，俗称访问控制列表，是用以控制对象的访问权限的。其主要思想是将某个对象的某种权限授予给某个用户，或某种GrantedAuthority（可以简单的理解为某种角色），它们之间的关系都是多对多。如果某一个对象的某一操作是受保护的，那么在对该对象进行某种操作时就需要有对应的权限。

## 1.1     准备工作

       使用Spring Security的Acl功能需要引入Acl相关的jar包。如果我们的应用是使用Maven构建的，则可以在应用的pom.xml文件中加入如下依赖。

   <dependency>

      <groupId>org.springframework.security</groupId>

      <artifactId>spring-security-acl</artifactId>

      <version>${spring.security.version}</version>

   </dependency>

       此外，使用Spring Security的Acl时需要在数据库中建立四张表。在其官方文档中给出了一个基于数据库HSQLDB的建表语句。其脚本如下：

create table acl\_sid (

  id bigint generated by default as identity(start with 100) not null primary key,

  principal boolean not null,

  sid varchar\_ignorecase(100) not null,

  constraint unique\_uk\_1 unique(sid,principal) );

create table acl\_class (

  id bigint generated by default as identity(start with 100) not null primary key,

  class varchar\_ignorecase(100) not null,

  constraint unique\_uk\_2 unique(class) );

create table acl\_object\_identity (

  id bigint generated by default as identity(start with 100) not null primary key,

  object\_id\_class bigint not null,

  object\_id\_identity bigint not null,

  parent\_object bigint,

  owner\_sid bigint not null,

  entries\_inheriting boolean not null,

  constraint unique\_uk\_3 unique(object\_id\_class,object\_id\_identity),

  constraint foreign\_fk\_1 foreign key(parent\_object)references acl\_object\_identity(id),

  constraint foreign\_fk\_2 foreign key(object\_id\_class)references acl\_class(id),

  constraint foreign\_fk\_3 foreign key(owner\_sid)references acl\_sid(id) );

create table acl\_entry (

  id bigint generated by default as identity(start with 100) not null primary key,

  acl\_object\_identity bigint not null,ace\_order int not null,sid bigint not null,

  mask integer not null,granting boolean not null,audit\_success boolean not null,

  audit\_failure boolean not null,

  constraint unique\_uk\_4 unique(acl\_object\_identity,ace\_order),

  constraint foreign\_fk\_4 foreign key(acl\_object\_identity)

      references acl\_object\_identity(id),

  constraint foreign\_fk\_5 foreign key(sid) references acl\_sid(id) );

       笔者使用的是Oracle数据库，其中没有boolean和主键自增功能，对于boolean类型都使用一位number表示。具体建表语句如下所示：

create table acl\_sid (

  id number not null primary key,

  principal number(1) not null,

  sid varchar(100) not null,

  constraint unique\_uk\_1 unique(sid,principal) );

create table acl\_class (

  id number not null primary key,

  class varchar(100) not null,

  constraint unique\_uk\_2 unique(class) );

create table acl\_object\_identity (

  id number not null primary key,

  object\_id\_class number not null,

  object\_id\_identity number not null,

  parent\_object number,

  owner\_sid number not null,

  entries\_inheriting number(1) not null,

  constraint unique\_uk\_3 unique(object\_id\_class,object\_id\_identity),

  constraint foreign\_fk\_1 foreign key(parent\_object)references acl\_object\_identity(id),

  constraint foreign\_fk\_2 foreign key(object\_id\_class)references acl\_class(id),

  constraint foreign\_fk\_3 foreign key(owner\_sid)references acl\_sid(id) );

create table acl\_entry (

  id number not null primary key,

  acl\_object\_identity number not null,ace\_order int not null,sid number not null,

  mask number(3) not null,granting number(1) not null,audit\_success number(1) not null,

  audit\_failure number(1) not null,

  constraint unique\_uk\_4 unique(acl\_object\_identity,ace\_order),

  constraint foreign\_fk\_4 foreign key(acl\_object\_identity)

      references acl\_object\_identity(id),

  constraint foreign\_fk\_5 foreign key(sid) references acl\_sid(id) );

       新增记录时用于生成主键的sequence定义为：

  create or replace sequence seq\_acl\_sid start with 1 increment by 1;

  create or replace sequence seq\_acl\_class start with 1 increment by 1;

  create or replace sequence seq\_acl\_object\_identity start with 1 increment by 1;

  create or replace sequence seq\_acl\_entry start with 1 increment by 1;

## 1.2     表功能介绍

       如上所示，Spring Security的Acl功能需要使用到四张数据库表，分别为acl\_sid、acl\_class、acl\_object\_identity和acl\_entry。

### 1.2.1表acl\_sid

       表acl\_sid的结构如下所示：

|  |  |  |
| --- | --- | --- |
| **字段名** | **类型** | **说明** |
| id | number | 主键 |
| sid | varchar | 字符串类型的sid |
| principal | boolean | 是否用户 |

       表acl\_sid是用来保存Sid的。对于Acl而言，有两种类型的Sid，一种是基于用户的Sid，叫PrincipalSid；另一种是基于GrantedAuthority的Sid，叫GrantedAuthoritySid。acl\_sid表的sid字段存放的是用户名或者是GrantedAuthority的字符串表示。prinpal是用来区分对应的Sid是用户还是GrantedAuthority的。正如在前文所描述的那样，Acl中对象的权限是用来授予给Sid的，Sid有用户和GrantedAuthority之分，所以我们的对象权限是可以用来授予给用户或GrantedAuthority的。

### 1.2.2表acl\_class

       表acl\_class的结构如下所示：

|  |  |  |
| --- | --- | --- |
| **字段名** | **类型** | **说明** |
| id | number | 主键 |
| class | varchar | 对象类型的全限定名 |

       表acl\_class是用来保存对象类型的，字段class中保存的是对应对象的全限定名。Acl需要使用它来区分不同的对象类型。

### 1.2.3表acl\_object\_identity

       表acl\_object\_identity的结构如下：

|  |  |  |
| --- | --- | --- |
| **字段名** | **类型** | **描述** |
| id | number | 主键 |
| object\_id\_class | number | 关联acl\_class，表示对象类型 |
| object\_id\_identity | number | 对象的主键，对于相同的class而言，其需要是唯一的。对象的主键默认需要是Long型，或者可以转换为Long型的对象，如Integer、Short等。 |
| parent\_object | number | 父对象的id，关联acl\_object\_identity |
| owner\_sid | number | 拥有者的sid，关联acl\_sid |
| entries\_inheriting | boolean | 是否继承父对象的权限。打个比方，删除对象childObj需要有delete权限，用户A他没有childObj的delete权限，但是他有childObj的父对象parentObj的delete权限，当entries\_inheriting为true时，用户A同样可以删除childObj。 |

       表acl\_object\_identity是用来存放需要进行访问控制的对象的信息的。其保存的信息有对象的拥有者、对象的类型、对象的主键、对象的父对象和是否继承父对象的权限。

### 1.2.4表acl\_entry

       表acl\_entry的结构如下：

|  |  |  |
| --- | --- | --- |
| **字段名** | **类型** | **说明** |
| id | number | 主键 |
| acl\_object\_identity | number | 对应acl\_object\_identity的id |
| ace\_order | number | 所属Acl的权限顺序 |
| sid | number | 对应acl\_sid的id |
| mask | number | 权限对应的掩码 |
| granting | boolean | 是否授权 |
| audit\_success | boolean | 暂未发现其作用，Acl中有一个更新其值的方法，但未见被调用。 |
| audit\_failure | boolean |  |

       表acl\_entry是用于存放具体的权限信息的，从表结构我们也可以看出来，其描述的就是某个主体（Sid）对某个对象（acl\_object\_identity）是否（granting）拥有某种权限（mask）。当同一对象acl\_object\_identity在acl\_entry表中拥有多条记录时，就会使用ace\_order来标记对应的顺序，其对应于往Acl中插入AccessControlEntry时的位置，在进行权限判断时也是依靠ace\_order的顺序来进行的，ace\_order越小的越先进行判断。ace是Access Control Entry的简称。

## 1.3     Acl主要接口

       对于Acl而言，有两块比较核心的功能，一块是往对应的数据库表里面插数据，另一块是从数据库表里面取出对应的数据进行权限鉴定。要了解这些功能我们先来了解Acl中用到的主要接口。

l  **Sid**：可以用来表示一个principal，或者是一个GrantedAuthority。其对应的实现类有表示principal的PrincipalSid和表示GrantedAuthority的GrantedAuthoritySid。其信息会保存在acl\_sid表中。

l  **ObjectIdentity**：ObjectIdentity表示Spring Security Acl中一个域对象，其默认实现类是ObjectIdentityImpl。ObjectIdentity并不是直接与acl\_object\_identity表相对应的，真正与acl\_object\_identity表直接相对应的是Acl。

l  **Acl**：每一个领域对象都会对应一个Acl，而且只会对应一个Acl。Acl是将Spring Security Acl中使用到的四个表串联起来的一个接口，其中会包含对象信息ObjectIdentity、对象的拥有者Sid和对象的访问控制信息AccessControlEntry。在Spring Security Acl中直接与acl\_object\_identity表相关联的是Acl接口，因为acl\_object\_identity表中的数据是通过保存Acl来进行的。一个Acl对应于一个ObjectIdentity，但是会包含有多个Sid和多个AccessControlEntry，即一个Acl表示所有Sid对一个ObjectIdentity的所有AccessControlEntry。Acl的默认实现类是AclImpl，该类实现Acl接口、MutableAcl接口、AuditableAcl接口和OwnershipAcl接口。

l  **AccessControlEntry**：一个AccessControlEntry表示一条访问控制信息，一个Acl中可以拥有多个AccessControlEntry。在Spring Security Acl中很多地方会使用ACE来简单的表示AccessControlEntry这个概念，比如insertAce其实表示的就是insert AccessControlEntry。每一个AccessControlEntry表示对应的Sid对于对应的对象ObjectIdentity是否被授权某一项权限Permission，是否被授权将使用granting进行区分。AccessControlEntry对应表acl\_entry。

l  **Permission**：在Acl中使用一个bit掩码来表示一个Permission。Spring Security的Acl中默认使用的是BasePermission，其中已经定义了0-4五个bit掩码，分别对应于1、2、4、8、16，代表五种不同的Permission，分别是read (bit 0)、write (bit 1)、create (bit 2)、delete (bit 3)和administer (bit 4)。如果已经定义好的这五个bit掩码不能满足需求，我们可以对BasePermission进行扩展，也可以实现自己的Permission。Spring Security Acl默认的实现最多可以支持32个不同的掩码。

l  **AclService**：AclService是用来通过ObjectIdentity解析Acl的，其默认实现类是JdbcAclService。JdbcAclService底层操作是通过LookupStrategy来进行的，LookupStrategy的默认实现是BasicLookupStrategy。

l  **MutableAclService**：MutableAclService是用来对Acl进行持久化的，其默认实现类是JdbcMutableAclService。JdbcMutableAclService是继承自JdbcAclService的，所以我们可以同时通过JdbcMutableAclService对Acl进行读取和保存。如果我们希望自己来实现Acl信息的保存的话，我们也可以不使用该接口。

## 1.4     配置AclService

       AclService是使用Spring Security Acl功能的主入口。这里选择一个既可以从数据库读取Acl信息，又可以保存Acl信息到数据库的JdbcMutableAclService做示例。

       JdbcMutableAclService只有一个构造方法，它接收三个参数，DataSource、LookupStrategy和AclCache。其对应配置信息如下所示：

   <bean id="aclService"

      class="org.springframework.security.acls.jdbc.JdbcMutableAclService">

      <constructor-arg ref="dataSource" />

      <constructor-arg ref="lookupStrategy" />

      <constructor-arg ref="aclCache" />

   </bean>

       配置JdbcMutableAclService有一点需要注意的地方，那就是其在与数据库进行交互的时候所基于的脚本是本文开始部分我们提到的那些脚本。其对应的数据库表的主键是自增的，所以在保存Acl时所给出的脚本中没有新增主键id。比如在新增sid时默认使用的脚本是“insert into acl\_sid (principal, sid) values (?, ?)”，显然对于使用Oracle数据库作为示例的我们来说这条SQL是有问题的，因为新增的时候主键不能为空，所以如果我们需要使用JdbcMutableAclService来创建Acl的话我们得给JdbcMutableAclService指定新增记录时使用的脚本。这里我们将在新增的时候从之前建立好的Sequence获取值作为主键，示例如下：

   <bean id="aclService"

      class="org.springframework.security.acls.jdbc.JdbcMutableAclService">

      <constructor-arg ref="dataSource" />

      <constructor-arg ref="lookupStrategy" />

      <constructor-arg ref="aclCache" />

      <!-- 指定新增acl\_sid的脚本 -->

      <property name="insertSidSql"

         value="insert into acl\_sid(id, principal, sid) values (seq\_acl\_sid.nextval, ?, ?)" />

      <!-- 指定新增acl\_class的脚本 -->

      <property name="insertClassSql"

         value="insert into acl\_class(id, class) values (seq\_acl\_class.nextval, ?)" />

      <!-- 指定新增acl\_object\_identity的脚本 -->

      <property name="insertObjectIdentitySql"

         value="insert into acl\_object\_identity(id, object\_id\_class, object\_id\_identity, owner\_sid, entries\_inheriting) values(seq\_acl\_object\_identity.nextval, ?, ?, ?, ?)" />

      <!-- 指定新增acl\_entry的脚本 -->

      <property name="insertEntrySql"

         value="insert into acl\_entry(id, acl\_object\_identity, ace\_order, sid, mask, granting, audit\_success, audit\_failure) values (seq\_acl\_entry.nextval, ?, ?, ?, ?, ?, ?, ?)" />

   </bean>

       除了上述四个SQL之外，我们还需要指定两个属性对应的查询SQL，sidIdentityQuery和classIdentityQuery。因为JdbcMutableAclService在创建Acl时，如果当前用户在acl\_sid表中不存在或当前对象类型在acl\_class表中不存在，其会先将对应的信息存入acl\_sid表和acl\_class表，然后需要取出刚刚新增的acl\_sid的主键和acl\_class的主键以往acl\_object\_identity表中插入数据，对应acl\_object\_identity表中的owner\_sid和object\_id\_class字段。这两个属性的默认值是“call identity()”，显然对于Oracle数据库来说这是行不通的，所以我们需要自己指定它们。这里我们通过对应Sequence的当前值来获取刚刚新增的记录的主键。如：

   <bean id="aclService"

      class="org.springframework.security.acls.jdbc.JdbcMutableAclService">

      <constructor-arg ref="dataSource" />

      <constructor-arg ref="lookupStrategy" />

      <constructor-arg ref="aclCache" />

      <!-- 指定新增acl\_sid的脚本 -->

      <property name="insertSidSql"

         value="insert into acl\_sid(id, principal, sid) values (seq\_acl\_sid.nextval, ?, ?)" />

      <!-- 指定新增acl\_class的脚本 -->

      <property name="insertClassSql"

         value="insert into acl\_class(id, class) values (seq\_acl\_class.nextval, ?)" />

      <!-- 指定新增acl\_object\_identity的脚本 -->

      <property name="insertObjectIdentitySql"

         value="insert into acl\_object\_identity(id, object\_id\_class, object\_id\_identity, owner\_sid, entries\_inheriting) values(seq\_acl\_object\_identity.nextval, ?, ?, ?, ?)" />

      <!-- 指定新增acl\_entry的脚本 -->

      <property name="insertEntrySql"

         value="insert into acl\_entry(id, acl\_object\_identity, ace\_order, sid, mask, granting, audit\_success, audit\_failure) values (seq\_acl\_entry.nextval, ?, ?, ?, ?, ?, ?, ?)" />

      <!-- 查询刚刚新增的acl\_sid的主键的SQL -->

      <property name="sidIdentityQuery" value="select seq\_acl\_sid.currval from dual" />

      <!-- 查询刚刚新增的acl\_class的主键的SQL -->

      <property name="classIdentityQuery" value="select seq\_acl\_class.currval from dual"/>

   </bean>

### 1.4.1配置DataSource

       配置数据源这个没什么好说的，大家都见惯了，为保持文章的完整性，我这里还是把它列一下。直接上代码：

   <context:property-placeholder location="/WEB-INF/config/jdbc.properties" />

   <bean id="dataSource" class="org.apache.commons.dbcp2.BasicDataSource"

      destroy-method="close">

      <property name="driverClassName" value="${jdbc.driverClassName}" />

      <property name="url" value="${jdbc.url}" />

      <property name="username" value="${jdbc.username}" />

      <property name="password" value="${jdbc.password}" />

   </bean>

### 1.4.2配置LookupStrategy

       LookupStrategy是用来通过ObjectIdentity解析为对应的Acl的。Spring Security Acl中的默认实现类是BasicLookupStrategy，其的构造需要接收四个参数。

   <bean id="lookupStrategy"

      class="org.springframework.security.acls.jdbc.BasicLookupStrategy">

      <constructor-arg ref="dataSource" />

      <constructor-arg ref="aclCache" />

      <constructor-arg ref="aclAuthorizationStrategy" />

      <constructor-arg ref="grantingStrategy" />

   </bean>

### 1.4.3配置AclAuthorizationStrategy

       aclAuthorizationStrategy是在构造Acl的实现类AclImpl时必须给定的一个参数，其会用来在对Acl进行某些操作时检查当前用户是否具有对应的权限。AclAuthorizationStrategy的默认实现类是AclAuthorizationStrategyImpl，其构造需要接收一个或三个GrantedAuthority参数，用来对Acl进行相关操作时所需要的权限，包括更改Acl对应对象的所有者需要的权限、更改Acl中包含的某个AccessControlEntry的audit信息（对应acl\_entry表中的is\_audit\_success和is\_audit\_failure字段）需要的权限以及其它如增、删、改Acl中所包含的AccessControlEntry等需要的权限。这些权限的鉴定是我们在操作Acl时由Spring Security Acl内部进行判断的，我们只需要在这里定义就好。当Acl对应的所有者对Acl进行操作时，不管其是否拥有指定需要的权限，除了改变audit信息之外的所有操作默认都是被允许的。当只有一个参数时表示三者共用一个GrantedAuthority。

   <bean id="aclAuthorizationStrategy"

      class="org.springframework.security.acls.domain.AclAuthorizationStrategyImpl">

      <constructor-arg>

         <list>

            <beanclass="org.springframework.security.core.authority.SimpleGrantedAuthority">

                <constructor-arg value="ROLE\_ADMIN" /><!-- 改变所有权需要的权限 -->

            </bean>

            <beanclass="org.springframework.security.core.authority.SimpleGrantedAuthority">

                <constructor-arg value="gaModifyAuditing" /><!-- 改变授权需要的权限 -->

            </bean>

            <beanclass="org.springframework.security.core.authority.SimpleGrantedAuthority">

                <constructor-arg value="gaGeneralChanges" /><!-- 改变其它信息所需要的权限 -->

            </bean>

         </list>

      </constructor-arg>

   </bean>

### 1.4.4配置grantingStrategy

       grantingStrategy对应类型为PermissionGrantingStrategy接口，其中只定义了一个isGranted方法，用于判断基于指定的Permission列表和Sid列表指定的Acl是否被授予了访问权限。其默认实现类是DefaultPermissionGrantingStrategy。DefaultPermissionGrantingStrategy对于isGranted的实现逻辑是依次遍历Permission列表、Sid列表和Acl中包含的AccessControlEntry列表，找到第一个三者能够匹配的AccessControlEntry的isGranting（对应acl\_entry表的granting字段）作为isGranted的返回结果。如果在当前Acl中没有找到匹配的AccessControlEntry，同时Acl对应的entriesInheriting为true时将继续使用父级的Acl进行匹配，并依次进行，如果都没有匹配到，则将抛出异常。

   <bean id="grantingStrategy"

   class="org.springframework.security.acls.domain.DefaultPermissionGrantingStrategy">

      <constructor-arg>

         <bean class="org.springframework.security.acls.domain.ConsoleAuditLogger" />

      </constructor-arg>

   </bean>

### 1.4.5配置AclCache

       AclCache是用来缓存Acl信息的，Spring Security Acl中对于AclCache的默认实现是基于Ehcache的实现类EhCacheBasedAclCache。

   <bean id="aclCache"

      class="org.springframework.security.acls.domain.EhCacheBasedAclCache">

      <constructor-arg ref="cache" /><!-- 对应于Ehcache -->

      <constructor-arg ref="grantingStrategy" />

      <constructor-arg ref="aclAuthorizationStrategy" />

   </bean>

   <!-- 定义一个Ehcache -->

   <bean id="cache" class="org.springframework.cache.ehcache.EhCacheFactoryBean">

      <property name="cacheName" value="aclCache" />

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

   </bean>

   <!-- 定义CacheManager -->

   <bean id="aclCacheManager"

      class="org.springframework.cache.ehcache.EhCacheManagerFactoryBean">

      <!-- 指定配置文件的位置 -->

      <property name="configLocation" value="/WEB-INF/config/ehcache.xml" />

      <!-- 指定新建的CacheManager的名称 -->

      <property name="cacheManagerName" value="aclCacheManager" />

   </bean>

       为保持本文的完整性，这里贴出上述使用到的配置文件ehcache.xml的内容。具体如下所示：

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

   xsi:noNamespaceSchemaLocation="http://ehcache.org/ehcache.xsd"

   maxBytesLocalHeap="100M">

   <diskStore path="d:\\ehcache" />

   <defaultCache maxEntriesLocalHeap="200" />

   <cache name="aclCache" maxBytesLocalHeap="50M" maxBytesLocalDisk="5G"

      timeToIdleSeconds="120" timeToLiveSeconds="600" />

</ehcache>

       关于Ehcache的更多内容不在本文讨论范围之内，有需要的读者可以参考官方文档，也可以参考我的另一个关于Ehcache的系列文章。

       至此，关于AclService配置的内容就讲完了，AclService配置的完整内容如下：

   <bean id="aclService"

      class="org.springframework.security.acls.jdbc.JdbcMutableAclService">

      <constructor-arg ref="dataSource" />

      <constructor-arg ref="lookupStrategy" />

      <constructor-arg ref="aclCache" />

      <!-- 指定新增acl\_sid的脚本 -->

      <property name="insertSidSql"

         value="insert into acl\_sid(id, principal, sid) values (seq\_acl\_sid.nextval, ?, ?)" />

      <!-- 指定新增acl\_class的脚本 -->

      <property name="insertClassSql"

         value="insert into acl\_class(id, class) values (seq\_acl\_class.nextval, ?)" />

      <!-- 指定新增acl\_object\_identity的脚本 -->

      <property name="insertObjectIdentitySql"

         value="insert into acl\_object\_identity(id, object\_id\_class, object\_id\_identity, owner\_sid, entries\_inheriting) values(seq\_acl\_object\_identity.nextval, ?, ?, ?, ?)" />

      <!-- 指定新增acl\_entry的脚本 -->

      <property name="insertEntrySql"

         value="insert into acl\_entry(id, acl\_object\_identity, ace\_order, sid, mask, granting, audit\_success, audit\_failure) values (seq\_acl\_entry.nextval, ?, ?, ?, ?, ?, ?, ?)" />

      <!-- 查询刚刚新增的acl\_sid的主键的SQL -->

      <property name="sidIdentityQuery" value="select seq\_acl\_sid.currval from dual" />

      <!-- 查询刚刚新增的acl\_class的主键的SQL -->

      <property name="classIdentityQuery" value="select seq\_acl\_class.currval from dual"/>

   </bean>

   <bean id="lookupStrategy"

      class="org.springframework.security.acls.jdbc.BasicLookupStrategy">

      <constructor-arg ref="dataSource" />

      <constructor-arg ref="aclCache" />

      <constructor-arg ref="aclAuthorizationStrategy" />

      <constructor-arg ref="grantingStrategy" />

   </bean>

   <bean id="aclCache"

      class="org.springframework.security.acls.domain.EhCacheBasedAclCache">

      <constructor-arg ref="cache" /><!-- 对应于Ehcache -->

      <constructor-arg ref="grantingStrategy" />

      <constructor-arg ref="aclAuthorizationStrategy" />

   </bean>

   <!-- 定义一个Ehcache -->

   <bean id="cache" class="org.springframework.cache.ehcache.EhCacheFactoryBean">

      <property name="cacheName" value="aclCache" />

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

   </bean>

   <!-- 定义CacheManager -->

   <bean id="aclCacheManager"

      class="org.springframework.cache.ehcache.EhCacheManagerFactoryBean">

      <!-- 指定配置文件的位置 -->

      <property name="configLocation" value="/WEB-INF/config/ehcache.xml" />

      <!-- 指定新建的CacheManager的名称 -->

      <property name="cacheManagerName" value="aclCacheManager" />

   </bean>

   <bean id="aclAuthorizationStrategy"

      class="org.springframework.security.acls.domain.AclAuthorizationStrategyImpl">

      <constructor-arg>

         <list>

            <beanclass="org.springframework.security.core.authority.SimpleGrantedAuthority">

                <constructor-arg value="ROLE\_ADMIN" /><!-- 改变所有权需要的权限 -->

            </bean>

            <beanclass="org.springframework.security.core.authority.SimpleGrantedAuthority">

                <constructor-arg value="gaModifyAuditing" /><!-- 改变授权需要的权限 -->

            </bean>

            <beanclass="org.springframework.security.core.authority.SimpleGrantedAuthority">

                <constructor-arg value="gaGeneralChanges" /><!-- 改变其它信息所需要的权限 -->

            </bean>

         </list>

      </constructor-arg>

   </bean>

   <bean id="grantingStrategy"

   class="org.springframework.security.acls.domain.DefaultPermissionGrantingStrategy">

      <constructor-arg>

         <bean class="org.springframework.security.acls.domain.ConsoleAuditLogger" />

      </constructor-arg>

   </bean>

## 1.5     使用AclService

       配置好AclService之后我们就可以使用该JdbcMutableAclService来创建、更新和查找Acl了。在Spring Security Acl中Acl接口的默认实现类是AclImpl，该类实现Acl接口、MutableAcl接口、AuditableAcl接口和OwnershipAcl接口，当有必要在这几种接口之间切换时可以任意切换。

### 1.5.1创建Acl

       可以通过调用JdbcMutableAclService的createAcl()方法来创建一个Acl，其对应返回的是一个MutableAcl，该方法接收一个ObjectIdentity作为参数。在创建的时候如果ObjectIdentity对应的类型在acl\_class表中不存在，则会把ObjectIdentity对应的类型添加到acl\_class表中；如果当前用户对应的Sid在acl\_sid表中不存在则会将其添加到acl\_sid中。最后会将ObjectIdentity保存到acl\_object\_identity表中。正如在本文开始部分所描述的那样，一个Acl对应于一个ObjectIdentity，创建Acl就是创建ObjectIdentity的过程。在这三部分都完成之后，会重新从数据库查询出一个Acl，只是此时该Acl对应的AccessControlEntry列表为空。通常如果我们的对象是需要利用Acl进行访问控制的话，那么我们可以在创建该对象的时候一并创建该对象对应的Acl。

   @Autowired

**private** MutableAclService aclService;

**public void** addUser(User user) {

      ...

      //1、构建一个ObjectIdentity

      ObjectIdentity oi = **new** ObjectIdentityImpl(User.**class**, user.getId());

      //2、创建一个Acl、此时会如果对应的信息不存在会依次创建，如当前用户对应的Sid、ObjectIdentity对应于acl\_class表中的类型

      //最后是往acl\_object\_identity中插入对应的数据

      MutableAcl acl = aclService.createAcl(oi);

      ...

   }

       ObjectIdentityImpl拥有多个构造方法，具体可以参考Spring Security的API文档。

### 1.5.2查找Acl

       通过AclService的系列readAclById()方法可以通过给定的ObjectIdentity查找对应的Acl。此外通过findChildren()方法可以查找指定ObjectIdentity的子ObjectIdentity。关于这些方法的具体信息可以参考Spring Security的API文档。以下是一个简单的示例。

   ObjectIdentity oi = **new** ObjectIdentityImpl(User.**class**, user.getId());

   //获取ObjectIdentity对应的Acl

   MutableAcl acl = (MutableAcl) aclService.readAclById(oi);

### 1.5.3更新Acl

       Acl的更新主要是对应AccessControlEntry的更新，即对AccessControlEntry的增、删、改；此外还包括对Acl对应的ObjectIdentity信息的变更，如更改所有者、父子关系等。

       如下是一些更新Acl的示例，需要注意的是在调用MutableAclService的updateAcl()方法将对应信息同步到数据库之前，对Acl所做的所有修改都只是在内存中的。使用updateAcl()更新Acl信息到数据库时，其底层实现会先将数据库中所有对应的AccessControlEntry都删除，然后再将内存中的AccessControlEntry列表保存到数据库中。以下是其实现代码。

**public** MutableAcl updateAcl(MutableAcl acl) **throws** NotFoundException {

        Assert.notNull(acl.getId(), "Object Identity doesn't provide an identifier");

        // Delete this ACL's ACEs in the acl\_entry table

        deleteEntries(retrieveObjectIdentityPrimaryKey(acl.getObjectIdentity()));

        // Create this ACL's ACEs in the acl\_entry table

        createEntries(acl);

        // Change the mutable columns in acl\_object\_identity

        updateObjectIdentity(acl);

        // Clear the cache, including children

        clearCacheIncludingChildren(acl.getObjectIdentity());

        // Retrieve the ACL via superclass (ensures cache registration, proper retrieval etc)

**return** (MutableAcl) **super**.readAclById(acl.getObjectIdentity());

    }

#### 1.5.3.1添加AccessControlEntry

       添加AccessControlEntry是通过MutableAcl的insertAce()方法进行的，该方法的定义如下所示：

   /\*\*

    \*

    \* **@param** atIndexLocation 添加的位置，对应于acl\_entry表中的ace\_order字段

    \* **@param** permission 对应的Permission

    \* **@param** sid 对应Sid

    \* **@param** granting 是否赋予

    \*/

**void** insertAce(**int** atIndexLocation, Permission permission, Sid sid, **boolean** granting)

       参数atIndexLocation对应的是需要插入的AccessControlEntry在Acl对应的AccessControlEntry列表（java.util.List类型）中的位置，对应于acl\_entry表中ace\_order，而一个Acl代表其对应ObjectIdentity的所有关联Sid关联的所有AccessControlEntry，这个ace\_order是在这个范围内的order。以下是一个添加AccessControlEntry的示例：

      MutableAcl acl = ...;

      //基于principal的Sid

      Sid sid = **new** PrincipalSid(SecurityContextHolder.getContext().getAuthentication());

      Permission p = BasePermission.ADMINISTRATION;//管理员权限

      //将当前Acl的管理员权限赋予给指定的Sid

      acl.insertAce(acl.getEntries().size(), p, sid, **true**);   //添加AccessControlEntry到内存

      //保存到数据库

      acl = aclService.updateAcl(acl);

       上述的Sid，也可以是一个GrantedAuthoritySid，当把一个Acl的某Permission赋予给一个GrantedAuthoritySid时表示拥有该GrantedAuthority的用户都将拥有对应的Permission。如：

      MutableAcl acl = ...;

      //基于GrantedAuthority的Sid

      Sid sid = **new** GrantedAuthoritySid("ROLE\_ADMIN");

      Permission p = BasePermission.ADMINISTRATION;//管理员权限

      //将当前Acl的管理员权限赋予给指定的Sid

      acl.insertAce(acl.getEntries().size(), p, sid, **true**);   //添加AccessControlEntry到内存

      //保存到数据库

      acl = aclService.updateAcl(acl);

#### 1.5.3.2删除AccessControlEntry

       通过调用MutableAcl的deleteAce(int aceIndex)方法可以删除Acl中指定位置的AccessControlEntry，aceIndex是从0开始的，底层是使用的List的remove(int index)方法。

   ObjectIdentity oi = **new** ObjectIdentityImpl(User.**class**, user.getId());

   //获取ObjectIdentity对应的Acl

   MutableAcl acl = (MutableAcl) aclService.readAclById(oi);

   acl.deleteAce(0);//内存中删除第一个AccessControlEntry

   aclService.updateAcl(acl);//同步到数据库

#### 1.5.3.3更新AccessControlEntry

       通过MutableAcl的updateAce()可以更新指定位置的AccessControlEntry的Permission。

   ObjectIdentity oi = **new** ObjectIdentityImpl(User.**class**, user.getId());

   //获取ObjectIdentity对应的Acl

   MutableAcl acl = (MutableAcl) aclService.readAclById(oi);

   acl.updateAce(0, BasePermission.CREATE);//内存中更新第一个AccessControlEntry对应的Permission

   aclService.updateAcl(acl);//同步到数据库

#### 1.5.3.4修改所有者

       可以通过Acl的getOwner()方法获取Acl对应ObjectIdentity的拥有者Sid。通过MutableAcl的setOwner()方法可以在内存中更新对应Acl的拥有者。

   ObjectIdentity oi = **new** ObjectIdentityImpl(User.**class**, user.getId());

   //获取ObjectIdentity对应的Acl

   MutableAcl acl = (MutableAcl) aclService.readAclById(oi);

   acl.setOwner(**new** PrincipalSid("user"));//内存中更改拥有者

   aclService.updateAcl(acl);//同步到数据库

#### 1.5.3.5修改父Acl

       通过Acl的getParentAcl()方法可以获取到Acl对应ObjectIdentity对应的父ObjectIdentity对应的Acl。通过MutableAcl的setParent()方法可以在内存中修改Acl对应的父Acl，即修改Acl对应ObjectIdentity对应的父ObjectIdentity。

   ObjectIdentity oi = **new** ObjectIdentityImpl(User.**class**, user.getId());

   //获取ObjectIdentity对应的Acl

   MutableAcl acl = (MutableAcl) aclService.readAclById(oi);

   Acl newParent = ...;//以某种方式获取到Acl

   acl.setParent(newParent);//内存中更改父Acl

   aclService.updateAcl(acl);//同步到数据库

#### 1.5.3.6修改继承策略

       通过Acl的isEntriesInheriting()可以获取到当前Acl对应ObjectIdentity的继承策略，创建Acl时该值默认为true。通过MutableAcl的setEntriesInheriting()方法可以在内存中修改该Acl对应ObjectIdentity的继承策略。

   ObjectIdentity oi = **new** ObjectIdentityImpl(User.**class**, user.getId());

   //获取ObjectIdentity对应的Acl

   MutableAcl acl = (MutableAcl) aclService.readAclById(oi);

   acl.setEntriesInheriting(**false**);//内存中修改为不从父Acl继承AccessControlEntry

   aclService.updateAcl(acl);//同步到数据库

### 1.5.4删除Acl

       当我们删除对象的时候应该连同对应的Acl也一起删除。使用MutableAclService的deleteAcl(ObjectIdentity oi, boolean deleteChildren)方法可以删除指定ObjectIdentity对应的Acl，deleteChildren表示是否连同子ObjectIdentity对应的Acl也一起删除。deleteAcl将删除对应的ObjectIdentity，以及对应的AccessControlEntry，即其会删除acl\_object\_identity表和acl\_entry表中与当前Acl对应的ObjectIdentity相关的记录。

   ObjectIdentity objectIdentity = **new** ObjectIdentityImpl(User.**class**, id);

   aclService.deleteAcl(objectIdentity, **true**);

## 1.6     注入到AclPermissionEvaluator

       AclPermissionEvaluator是PermissionEvaluator的一个实现类。在之前关于使用基于表达式的权限控制一文中有提到过，PermissionEvaluator是为表达式hasPermission提供支持的。此外，PermissionEvaluator还为标签accesscontrollist提供支持。本节将就使用AclPermissionEvaluator支持在方法上使用@PreAuthorize进行权限控制时使用表达式hasPermission做一个简单讲解。

       AclPermissionEvaluator的构造需要接收一个AclService参数，在进行权限鉴定时其需要通过AclService获取到对应对象对应的Acl，然后判断该Acl中是否具有指定的Sid和指定的Permission。

       对于方法使用hasPermission表达式进行权限鉴定时需要做两个事情，首先需要指定global-method-security的pre-post-annotations="enabled"。其次需要手工定义DefaultMethodSecurityExpressionHandler并指定其permissionEvaluator为我们定义的AclPermissionEvaluator。

   <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="aclPermissionEvaluator"/>

   </bean>

   <bean id="aclPermissionEvaluator"

      class="org.springframework.security.acls.AclPermissionEvaluator">

      <constructor-arg ref="aclService" />

   </bean>

       定义后之后我们就可以在方法上使用@PreAuthorize和hasPermission表达式了。关于PermissionEvaluator和hasPermission的更多介绍可以参考《基于表达式的权限控制》一文。hasPermission有两种用法，一种是直接传一个对象和对应需要的权限，如hasPermission(object,permission)；另一种是传对象的id、对应类型和需要的权限，如hasPermission(targetId,targetType,permission)。传入的Permission是Spring Security Acl中的Permission接口实现类，而不是Spring Security的GrantedAuthority。传入的permission参数可以是一个Permission对象或Permission对象数组，也可以是一个整数或字符串。当传入的permission是整数或字符串时将由AclPermissionEvaluator的PermissionFactory进行解析，AclPermissionEvaluator默认拥有的PermissionFactory是DefaultPermissionFactory，其会将整形或字符串类型的permission解析成对应的BasePermission。如前所述，BasePermission中定义了五个BasePermission，其对应的名称和掩码分别为：READ (1)、WRITE (2)、CREATE (4)、DELETE (8)和ADMINISTER (16)。当permission使用字符串时我们只能使用这五种字符串，不区分大小写，表示当前用户或其所拥有的GrantedAuthority必须拥有指定对象的指定Permission才允许访问。但是当使用掩码时我们可以使用1、2、4、8和16。

       接下来将简单的介绍一个使用@PreAuthorize和hasPermission表达式在方法上进行权限控制的示例。

   @PreAuthorize("hasPermission(#id, 'com.spring.security.entity.User', 1)")

**public** User find(**int** id) {

      User user = **new** User();

      user.setId(id);

**return** user;

   }

       上述配置即表示调用find方法查看指定id对应的User对象时必须拥有掩码为1对应的Permission才行，或者在继续策略为true时拥有指定id父对象掩码为1对应的Permission也行。这时哪怕你是该User对象的拥有者，或者你拥有ADMINISTER权限，如果你没有对应的READ权限，你也不能访问该方法。如果觉得这种实现不符合你的要求，你可以实现自己的PermissionGrantingStrategy，然后将实现类bean注入到BasicLookupStrategy和EhcacheBasedAclCache中，这样在判断一个用户是否具有指定Acl的指定Permission时就可以使用自己的逻辑了。

       上面的定义如果改成permission参数直接使用对象，可以这样定义：

   @PreAuthorize("hasPermission(#id, 'com.spring.security.entity.User', T(org.springframework.security.acls.domain.BasePermission).READ)")

**public** User find(**int** id) {

      User user = **new** User();

      user.setId(id);

**return** user;

   }

       当permission参数定义为一个Permission数组时，会根据顺序依次匹配当前用户在指定的Acl中是否拥有对应Permission的AccessControlEntry，如果拥有则以第一个匹配到的AccessControlEntry的granting属性作为判断结果，没有匹配到还可以根据继承策略决定是否利用父级Acl进行匹配，都没匹配到就会抛异常了。

       默认使用的BasePermission中只定义了五种Permission，如果这不能满足你的要求，那么我们可以实现自己的Permission，然后把它们注册到DefaultPermissionFactory中，并手工将该DefaultPermissionFactory注入到AclPermissionEvaluator中。前文已经说过AclPermissionEvaluator中使用的PermissionFactory默认是DefaultPermissionFactory，DefaultPermissionFactory中默认只注册了BasePermission中对应的五种Permission。以下是一个扩展Permission的简单示例。

       首先实现自己的Permission类，这里简单的定义一个自己的类，然后继承BasePermission类。

**public class** BasePermissionExt **extends** BasePermission {

   /\*\*

    \* serialVersionUID

    \*/

**private static final long**serialVersionUID = 1L;

**public** BasePermissionExt(**int** mask) {

**super**(mask);

   }

**public** BasePermissionExt(**int** mask, **char** code) {

**super**(mask, code);

    }

}

       然后在DefaultPermissionFactory中注册基于我们自己实现的Permission对象，并将该DefaultPermissionFactory注入到AclPermissionEvaluator中。

   <bean id="aclPermissionEvaluator"

      class="org.springframework.security.acls.AclPermissionEvaluator">

      <constructor-arg ref="aclService" />

      <!-- 手工注册DefaultPermissionFactory和其中的Permission -->

      <property name="permissionFactory">

         <bean class="org.springframework.security.acls.domain.DefaultPermissionFactory">

            <constructor-arg>

                <map>

                   <entry key="READ">

                      <bean class="com.xxx.spring.security.BasePermissionExt">

                         <constructor-arg value="1"/>

                      </bean>

                   </entry>

                   <entry key="WRITE">

                      <bean class="com.xxx.spring.security.BasePermissionExt">

                         <constructor-arg value="2"/>

                      </bean>

                   </entry>

                </map>

            </constructor-arg>

         </bean>

      </property>

   </bean>

       此外，还可以将我们的BasePermissionExt改成如下这样：

**public class** BasePermissionExt **extends** BasePermission {

   /\*\*

    \* serialVersionUID

    \*/

**private static final long**serialVersionUID = 1L;

**public static final** Permission READ = **new** BasePermissionExt(1 << 0, 'R'); // 1

**public static final** Permission WRITE = **new** BasePermissionExt(1 << 1, 'W'); // 2

**public static final** Permission CREATE = **new** BasePermissionExt(1 << 2, 'C'); // 4

**public static final** Permission DELETE = **new** BasePermissionExt(1 << 3, 'D'); // 8

**public static final** Permission ADMINISTRATION = **new** BasePermissionExt(1 << 4, 'A'); // 16

**public** BasePermissionExt(**int** mask) {

**super**(mask);

   }

**public** BasePermissionExt(**int** mask, **char** code) {

**super**(mask, code);

    }

}

       然后通过传入Class参数来构造DefaultPermissionFactory。这个时候会将对应Class中所有类型为Permission的字段分别以字段名和字段值Permission对应的掩码为Key，以字段值Permission为Value进行注册。

   <bean id="aclPermissionEvaluator"

      class="org.springframework.security.acls.AclPermissionEvaluator">

      <constructor-arg ref="aclService" />

      <!-- 手工注册DefaultPermissionFactory和其中的Permission -->

      <property name="permissionFactory">

         <bean class="org.springframework.security.acls.domain.DefaultPermissionFactory">

            <constructor-arg value="com.spring.security.BasePermissionExt"/>

         </bean>

      </property>

   </bean>

[使用rsync无密码传输](http://www.cnblogs.com/fenglan/p/5916170.html)

众多数据库服务器的管理过程中，在不同服务器间的文件传输是免不了的。您可以使用scp命令或FTP方法完成文件的发送和接收，这篇文章我将给大家介绍另外一种方法，这就是rsync命令。  
rsync是文件传输程序，它的功能和算法描述可以参考维基百科中的介绍。  
参考链接：http://zh.wikipedia.org/zh-cn/Rsync  
  
我来通过实验给大家展示一下rsync命令的常用方法。  
这里我们演示使用的操作系统是Linux，rsync已经预装好。如果是AIX操作系统，需要单独在IBM网站下载安装包安装。  
  
演示场景：将secDB1服务器SRC目录下的内容同步传输到secDB2服务器DEST目录下，整个过程在secDB1服务器上来完成。  
  
1.在第一台主机的根目录下创建一个名为“/usr/20160626”的目录，作为我们传输演示的源目录。  
[root@secDB1 /]# mkdir /usr/20160626  
  
2.创建文件，根据日期生成文件夹，并拷贝每日生成的文件到新的文件夹中

datestr=$(date +%Y%m%d)  
mkdir $datestr  
cp $(find /orcl/dir -name "\*$datestr.txt" ) /orcl/$datestr/

3.在第二台主机的根目录下创建一个名为“test”的目录，作为我们传输演示的目标目录  
[root@secDB2 /]# cd /usr  
[root@secDB2 /]# mkdir /test  
  
4.将secDB1服务器20160926目录下的内容同步传输到secDB2服务器test目录下  
[root@secDB1 SRC]#  rsync -av "ssh -i /usr/rsync\_id\_dsa" /orcl/20160926/ csdn@61.132.54.173:/usr/test  
root@172.17.193.211's password:  
building file list ... done  
./

pos898321045110581\_20160926.txt  
pos898321045110582\_20160926.txt  
pos898321045110583\_20160926.txt  
pos898321045110584\_20160926.txt  
pos898321045110593\_20160926.txt  
pos898321045110594\_20160926.txt  
pos898321045110595\_20160926.txt  
pos898321045110596\_20160926.txt

sent 162 bytes  received 70 bytes  42.18 bytes/sec  
total size is 0  speedup is 0.00  
  
5.验证第二台主机的test目录下是否有发送过来的文件  
[root@secDB2 /]# cd /test  
[root@secDB2 DEST]# ls -ltr  
total 0  
-rw-r--r-- 1 root root 0 Feb 21 02:37 f1  
-rw-r--r-- 1 root root 0 Feb 21 02:37 f2  
  
文件发送成功，目的达到。  
  
6.修改test目录下的f1文件内容，使之与源目录SRC中的f1目录不相同  
[root@secDB2 DEST]# echo "mod" > f1  
[root@secDB2 DEST]# cat f1  
mod  
[root@secDB2 DEST]# ls -ltr  
total 4  
-rw-r--r-- 1 root root 0 Feb 21 02:37 f2  
-rw-r--r-- 1 root root 4 Feb 21 02:59 f1  
  
7.再次将20160926目录下的内容向test目录同步  
[root@secDB1 SRC]# rsync -av /SRC/ -e ssh root@172.17.193.211:/DEST  
root@172.17.193.211's password:  
building file list ... done  
f1  
  
sent 114 bytes  received 48 bytes  46.29 bytes/sec  
total size is 0  speedup is 0.00  
  
注意，此时仅仅将有变化的内容f1同步到了test目录，这也是rsync命令的优势所在。  
  
8.在经过再次同步之后，此时DEST目录下的文件再一次与SRC目录下的内容保持一致。  
[root@secDB2 DEST]# ls -ltr  
total 0  
-rw-r--r-- 1 root root 0 Feb 21 02:37 f1  
-rw-r--r-- 1 root root 0 Feb 21 02:37 f2  
  
9.注意“/”符号带来的差别  
如果将“/SRC/”后面的“/”去掉，此时将会将SRC目录和目录的下的所有内容统统的发送到DEST目录下。  
[root@secDB1 SRC]# rsync -av /SRC -e ssh root@172.17.193.211:/DEST  
root@172.17.193.211's password:  
building file list ... done  
SRC/  
SRC/f1  
SRC/f2  
  
sent 166 bytes  received 70 bytes  67.43 bytes/sec  
total size is 0  speedup is 0.00  
  
10.使用rsync命令的联机帮助文档可以获得更多的提示信息  
[root@secDB1 SRC]# rsync -h  
……此处内容省略……  
  
11.小结  
rsync在给我们传输文件提供另外一种途径的同时，也提高了我们的工作效率。在存在少量差异需要完成文件同步的场合下是不二选择。善用之。  
  
Good luck.

 -------------------------------------------------------------------------------------------------------------------------------------

二、取消输入密码

问题描述

在linux平台上常用rsync协议来同步文件和目录，'rsync'也是各种linux发行版标配的命令。当我们基于linux平台开发应用程序时，常常需要从[Java](http://lib.csdn.net/base/17)或者[Python](http://lib.csdn.net/base/11)代码中调用rsync命令来实现文件的传输。

例如，假设我们要按照如下需求来传输文件：

* 有两台机器：localhost和remotehost
* Java程序deploy在localhost上，通过rsync将本机文件传送到remotehost，或者从remotehost pull文件到localhost
* rsync通过ssh连接remotehost
* 在remotehost有个叫csdn的帐户，专门用来rsync传输文件，只能用密钥文件登陆
* 最重要的是，在rsync连接过程中不需要输入csdn的帐户密码，否则密码输入界面会block住Java程序的运行

要实现rsync的无密码传输，基本思路是用一对公私钥来代替帐户密码，实现ssh连接时的认证。

下面我们来看看一步步如何实现。

配置rsync

1. [远端机器] 创建专用帐户csdn

**[plain]** [view plain](http://blog.csdn.net/fuguoq1984/article/details/32331941) [copy](http://blog.csdn.net/fuguoq1984/article/details/32331941)

1. # 创建用户组csdn
2. $ groupadd -g 1212 csdn
4. # 创建csdn用户，并加入csdn用户组
5. $ useradd -m -s /bin/bash -g 1212 -u 1212 csdn
7. # 为csdn创建.ssh目录
8. $ mkdir /home/csdn/.ssh
9. $ chown csdn:csdn /home/csdn/.ssh
10. 要上传的文件夹$ chown csdn:csdn/usr/test

注意：

* gid和uid均为1212，此为示例，也可以选用其它值，只要保证不和现有的用户和组冲突
* "-m"表示要为csdn用户创建home目录，即/home/csdn/

2. [本地机器] 用ssh-keygen生成密钥

**[plain]** [view plain](http://blog.csdn.net/fuguoq1984/article/details/32331941) [copy](http://blog.csdn.net/fuguoq1984/article/details/32331941)

1. [root@xxxx ~]$ ssh-keygen
2. Generating public/private rsa key pair.
3. Enter file in which to save the key (/root/.ssh/id\_rsa): /usr/rsync\_id\_dsa
4. Enter passphrase (empty for no passphrase):  //输入空
5. Enter same passphrase again:     //输入空
6. Your identification has been saved in /usr/rsync\_id\_dsa.
7. Your public key has been saved in /usr/rsync\_id\_dsa.pub.

注意：

* 在提示输入key file path时，为避免覆盖已有的root帐户的id\_rsa文件，将其存储到/usr/rsync\_id\_dsa
* 在提示输入文件密码时，直接按回车，采用空密码
* 最后生成了两个文件，分别存储私钥和公钥

3. [本地机器] 将公钥传送到远端机器csdn帐户的.ssh目录下

**[plain]** [view plain](http://blog.csdn.net/fuguoq1984/article/details/32331941) [copy](http://blog.csdn.net/fuguoq1984/article/details/32331941)

1. $ scp /usr/rsync\_id\_dsa.pub root@171.69.201.209:/home/csdn/.ssh

4. [远端机器] 将传输过来的公钥导入csdn的authorized\_keys文件

**[plain]** [view plain](http://blog.csdn.net/fuguoq1984/article/details/32331941) [copy](http://blog.csdn.net/fuguoq1984/article/details/32331941)

1. $ cd /home/csdn/.ssh
2. $ cat rsync\_id\_dsa.pub > authorized\_keys
3. $ chown csdn:csdn authorized\_keys

rsync命令

这样，我们就可以免输密码在两机器间同步文件和目录了，比如，一个最简单的命令：

**[plain]** [view plain](http://blog.csdn.net/fuguoq1984/article/details/32331941) [copy](http://blog.csdn.net/fuguoq1984/article/details/32331941)

1. $　rsync -e "ssh -i /usr/rsync\_id\_dsa" /tmp/testfile csdn@remotehost:/tmp/
2. 传递文件代码为   [root@postest ~]# rsync -av -e  "ssh -i /usr/rsync\_id\_dsa" /orcl/20160926/ csdn@61.132.54.173:/usr/test

值得注意的是，当从Java或Python代码中构建rsync命令时，有时很难将-e "ssh -i xxxx"作为一个普通的command option，这里有一个简单的替代方法，就是在启动rsync命令之前，往rsync进程中添加一个环境变量：RSYNC\_RSH="ssh -i xxxx"

上面只是一个最简单的rsync命令，其实rsync有许多有意思的选项，可以帮助我们完成很多高端大气上档次的功能。比如，下面是一个更加复杂的命令，用来在两台机器间传输整个目录并在传输完成后删除源文件：

**[plain]** [view plain](http://blog.csdn.net/fuguoq1984/article/details/32331941) [copy](http://blog.csdn.net/fuguoq1984/article/details/32331941)

1. $　rsync -e "ssh -i /usr/rsync\_id\_dsa" -v --progress -r -z --compress-level=9 -o -g -p --remove-source-files --temp-dir=/tmp/rsynctmpdir/ --partial-dir=/tmp/rsynctmpdir/ --delay-updates --itemize-changes --out-format=FILEDETAIL::%%n::%%l::%%i::%%M /tmp/testfolder/ csdn@remotehost:/tmp/testfolder/

具体参数的含义可参考rsync文档，这里仅作简单说明：

* + -v：传输过程中打印调式信息
  + --progress：传输过程中打印当前进度
  + -r：递归传输所有子目录和文件
  + -z, --compress-level=9：传输过程中对文件进行压缩，可提高传输效率
  + -o, -g, -p：传输过程中保持文件的ownership, group和permission
  + --remove-source-files：传输结束之后删除源文件。注意，在rsync 2.x中，相应的选项为--remove-sent-files
  + --temp-dir, --partial-dir, --delay-updates：将源目录下所有的文件放在一个transaction里，要么全成功，要么全失败。传输过程中将文件放在temp-dir和partial-dir中，全部完成之后再移动到目标目录。这几个选项在remotehost上有其它程序需要consume传输完成之后的文件时特别有用。
  + --itemize-changes, --out-format：按照指定格式输出每一个要传送的文件信息，这在应用程序需要捕获传输的文件列表时很有用。比如，应用程序仅指定在两个机器间要同步的目录(从localhost到remotehost，或者从remotehost到localhost)，然后通过这两个选项就可以捕获rsync的output，过滤出相应的输出行(根据--out-format中指定的"FILEDETAIL"前缀字符串)，然后就可以构建出目录下的文件列表了，比如文件名(%%n)、文件大小(%%l)等等

# [crontab 添加sh文件定时](http://www.cnblogs.com/fenglan/p/5917593.html)

（1）编写sh文件，比如/orcl/test/export.sh

      编写crond文件

      chmod 755 /orcl/test/\*      //复制所有权限

        [root@postest test]# vi crond.txt  
         19 18 \* \* \* sh  /orcl/test/cpfile.sh

（2）上传到/opt目录

       [root@postest test]# cp /orcl/test/crond.txt /opt/  
（3）运行命令   
crontab /opt/crond.txt   
crontab –l   
让配置文件生效：如果让配置文件生效，还得重新启动cron，切记，既然每个用户下的cron配置文件修改后，也要重新启动cron服务器。

[root@postest test]# service crond restart  
Stopping crond: [ OK ]  
Starting crond:

如果让crond在开机时运行，应该改变其运行级别：   
chkconfig –level 35 crond on   
service crond status查看cron服务状态，如果没有启动则service crond start启动它，cron服务是一个定时执行的服务，可以通过crontab命令添加或者编辑需要定时执行的任务

# [mybatis-generator 代码自动生成工具（maven方式）](http://www.cnblogs.com/fenglan/p/5956953.html)

由于MyBatis属于一种半自动的ORM框架，所以主要的工作将是书写Mapping映射文件，但是由于手写映射文件很容易出错，mybatis-gennerator插件帮我们自动生成mybatis所需要的dao、bean、mapper xml文件。

这里主要通过eclipse工具，来讲解实现;

1、建表语句

CREATE TABLE `user` (

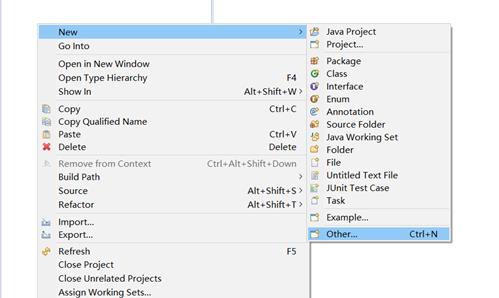
`id` int(**11) NOT NULL AUTO\_INCREMENT,**

**`name` varchar(100) DEFAULT NULL,**

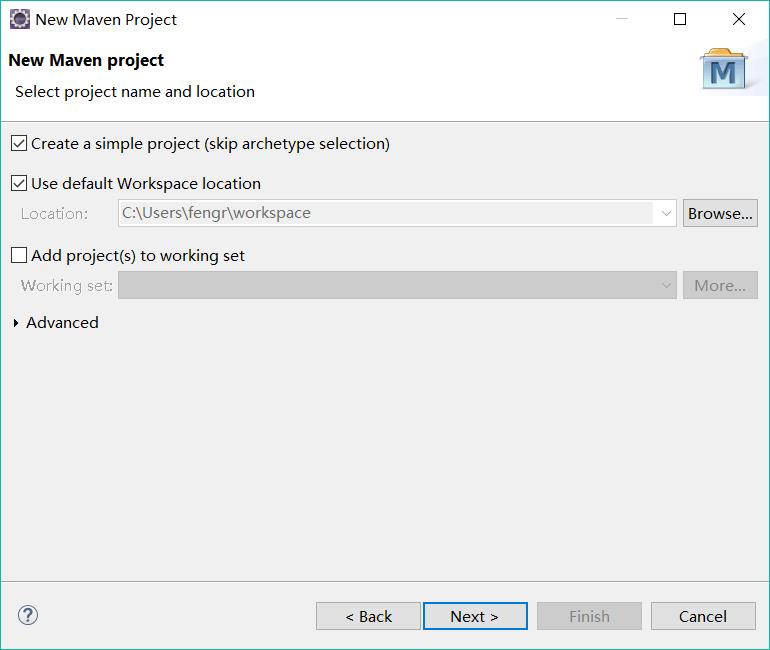
**PRIMARY KEY (`id`)**

**) ENGINE=InnoDB AUTO\_INCREMENT=3 DEFAULT CHARSET=utf8;**

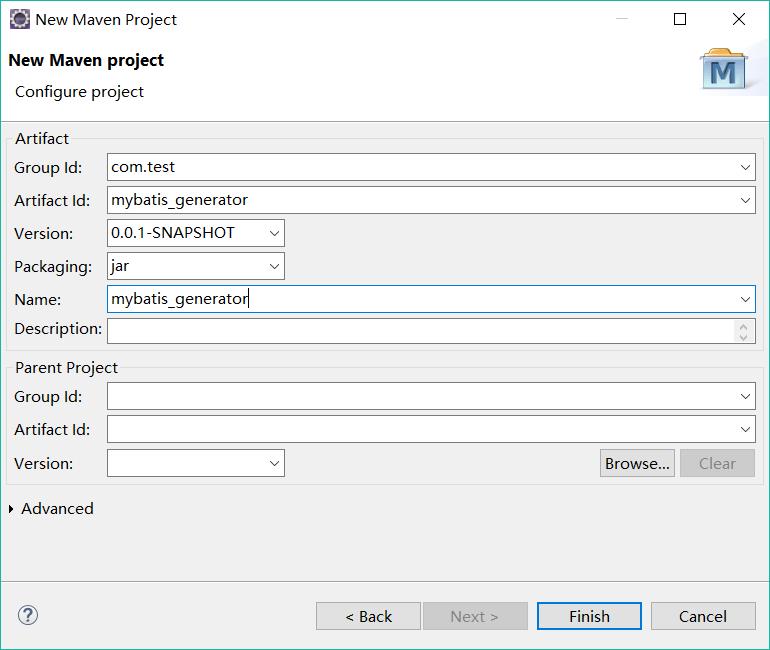
2、新建测试工程



选择maven工程



选择create a simple project就行



点击finish，测试项目就建完了

3、在maven配置文件pom.xml中添加依赖

[复制代码](javascript:void(0);)

复制代码

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.test</groupId>

<artifactId>mybatis\_generator</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>mybatis\_generator</name>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

<dependencies>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>5.1.35</version>

</dependency>

<dependency>

<groupId>org.mybatis.generator</groupId>

<artifactId>mybatis-generator-core</artifactId>

<version>1.3.2</version>

</dependency>

</dependencies>

<build>

<pluginManagement>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<configuration>

<source>1.7</source>

<target>1.7</target>

</configuration>

<version>3.3</version>

</plugin>

<plugin>

<groupId>org.mybatis.generator</groupId>

<artifactId>mybatis-generator-maven-plugin</artifactId>

<version>1.3.2</version>

<dependencies>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>5.1.35</version>

</dependency>

</dependencies>

<configuration> <!--配置文件的路径-->

<configurationFile>${basedir}/src/main/resources/generatorConfig.xml</configurationFile>

<overwrite>true</overwrite>

</configuration>

</plugin>

</plugins>

</pluginManagement>

</build>

</project>

复制代码

[复制代码](javascript:void(0);)

4、在resources下，创建一个generatorConfig.xml配置文件

[复制代码](javascript:void(0);)

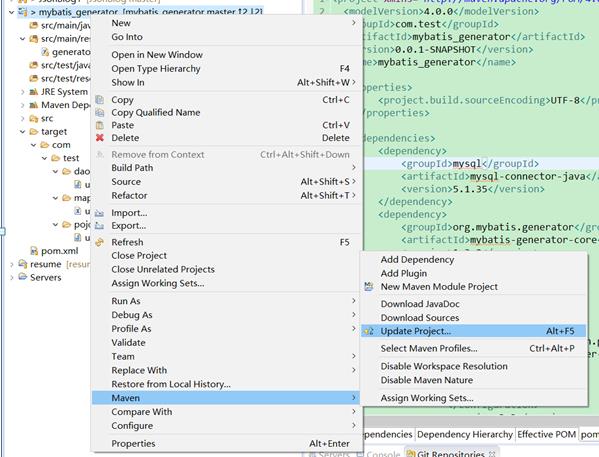
按 Ctrl+C 复制代码



按 Ctrl+C 复制代码

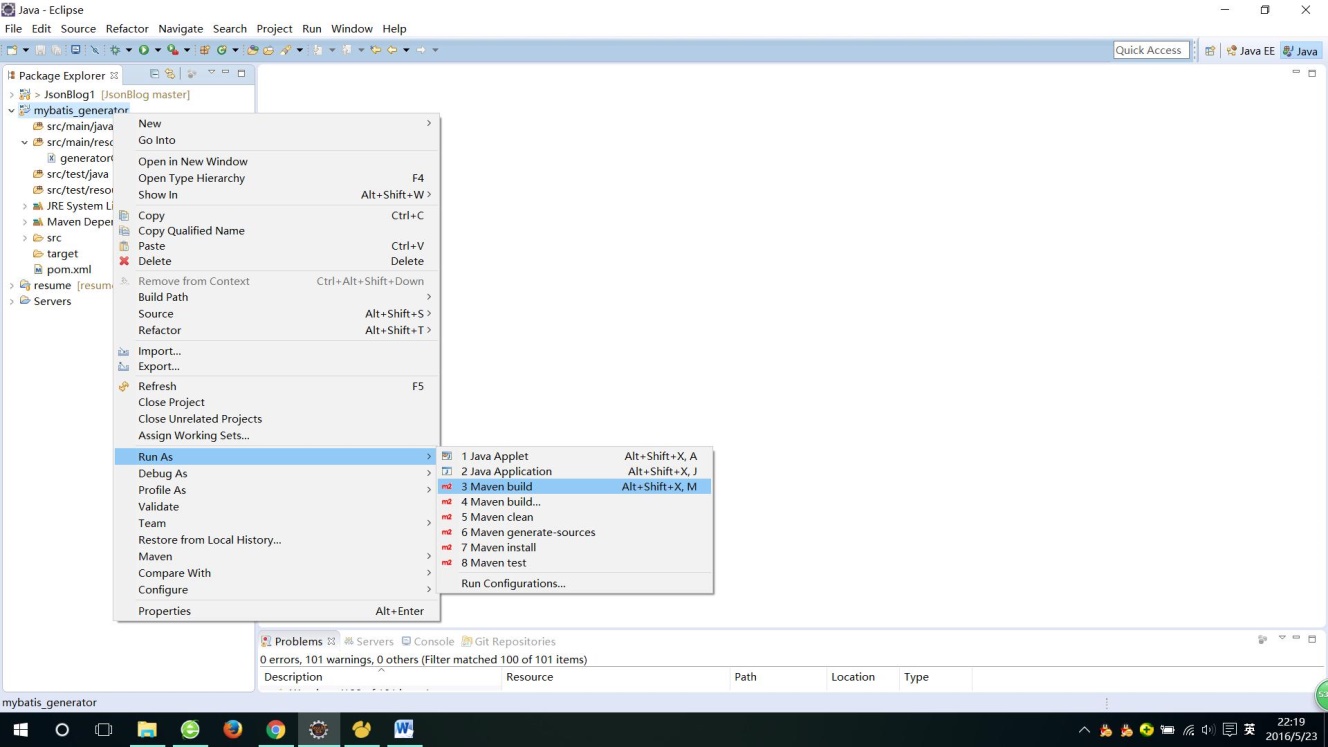
[复制代码](javascript:void(0);)

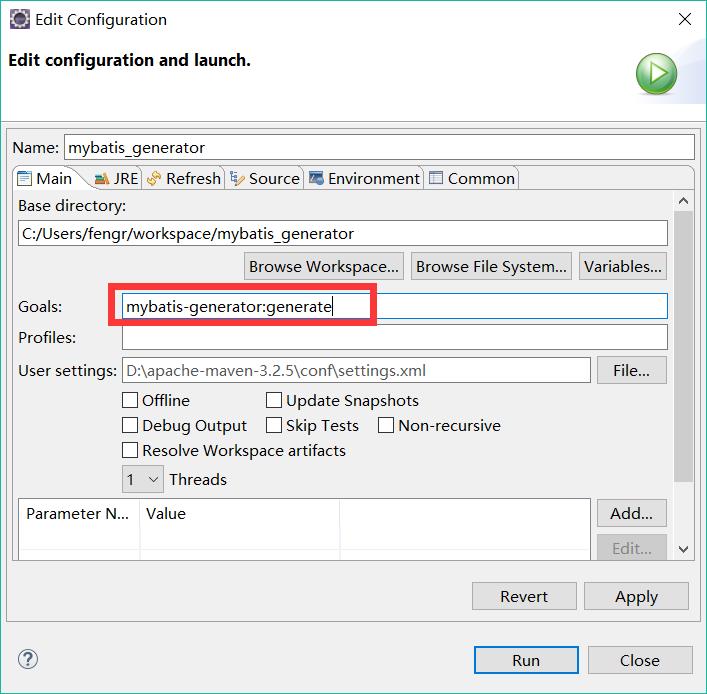
5、下载maven依赖包，update project



这个要稍微等一下，需要时间~~~

6、执行mabatis-generator:generate命令，生成文件

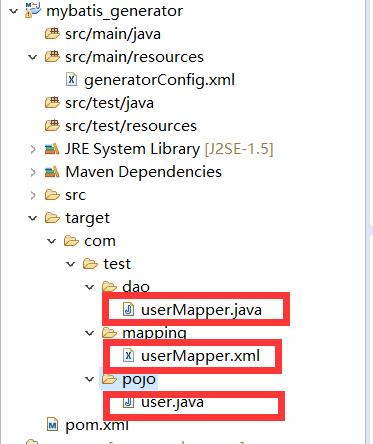




在控制台 显示 build success，说明已经成功了：



7、在项目上F5刷新，target目录下呢就会出现对应的文件



这是想要的文件自动生成了。

 github项目地址：<https://github.com/JsonShare/mybatis_generator.git>

# Security登录用户的IP和在线人数（20）

在spring security3 中获取所有登录用户的信息，包括比较难找的IP

一：如果单纯获取当前登录用户的信息，那么简单，只要认证成功了，在任何地方使用：

import org.springframework.security.core.Authentication;

import org.springframework.security.core.GrantedAuthority;

import org.springframework.security.core.context.SecurityContextHolder; import

org.springframework.security.web.authentication.WebAuthenticationDetails

import org.springframework.security.core.userdetails.User

//auth包含两个很重要的对象1:Details 2:Principal

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

//Details可转换为WebAuthenticationDetails

//wauth包含着1:remoteAddress 2:sessionId

WebAuthenticationDetails wauth = (WebAuthenticationDetails) auth.getDetails();

System.out.println("当前登录用户的ip：" + wauth.getRemoteAddress());

System.out.println("当前登录用户的sessionID：" + wauth.getSessionId());

//Principal可转换为User

//u包含登录账户和权限，扩展此对象可以让其包含更多信息，像真实姓名，所在部门，用户id...

User u = (User) auth.getPrincipal();

System.out.println("当前登录用户的账号：" + u.getUsername());

System.out.println("当前登录用户的权限：" + u.getAuthorities());

System.out.println("当前登录用户的姓名(扩展User之后才能使用)：" + u.getId());

二：如果是想获取所有登录用户的信息，也不复杂，所有的登录用户都保存在SessionRegistry 上，所有只要变量此对象就能获取所有在线用户，获取它也简单，只要用spring注入就可以了。

代码如下：

package com.baodian;

import java.text.Format;

import java.text.SimpleDateFormat;

import java.util.Date; import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.security.core.session.SessionInformation;

import org.springframework.security.core.session.SessionRegistry;

import org.springframework.security.core.userdetails.User;

import org.springframework.stereotype.Component;

@Component("getUser")

public class GetUser {

//利用spring注入后就可以取所有登录用户，不能加static属性，否则会取不到

@Autowired

private SessionRegistry sessionRegistry;

private Format ft = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

public void findlgUser() {

List<Object> slist = sessionRegistry.getAllPrincipals();

String json = "共 " + slist.size() + " 用户登录，服务器时间：" + ft.format(new Date()) + "\n";

for(int i=0; i<slist.size(); i++) {

User u = (User) slist.get(i);

json = json + "第 " + (i+1) + " 个用户 => " + "账号：" + u.getUsername() + "\n";

//第二个参数表示是否取单点登录时，超出限制而被弹出用户

List<SessionInformation> ilist = sessionRegistry.getAllSessions(slist.get(i), true);

json = json + "----" + ilist.size() + " 处登录\n";

for(int j=0; j<ilist.size(); j++) {

SessionInformation sif = ilist.get(j);

u = (User) sif.getPrincipal();

json = json + "--------" + (j+1) +

" 用户名" + u.getUsername() +

" 最后访问时间：" + ft.format(sif.getLastRequest()) + " session:" + sif.getSessionId() +

" 限制登录：" + sif.isExpired() + "\n";

//强制退出1.将其限制，2.将其移除，使用第一种比较好

//sif.expireNow();

//sessionRegistry.removeSessionInformation(sif.getSessionId());

}

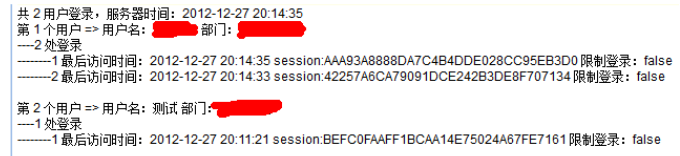
}

System.out.println(json);

}

}

运行截图：我设置的是允许多个用户同时使用一个账号



三：但是要想获取所有登录用户的ip就有点复杂了（至少我找了很久都没发现有更简单的办法）， 因为用来保存ip的WebAuthenticationDetails对象只保存在spring的上下文中，并没有在像 SessionRegistry这样好用的对象中保存一份，那怎么办，只有改写代码，把ip保存到 SessionRegistry中，这样就省事很多了。

1.首先找到把登录用户扔进SessionRegistry中的对象：

org.springframework.security.web.authentication.session.ConcurrentSessionControlStrategy

此对象的61行：onAuthentication() 和 148行onSessionChange()上都有使用 sessionRegistry.registerNewSession()把sessionId和权限两个元素保存起来，那么好办了，在这个registerNewSession方法中再加一个ip元素就行了。

关于ip另外保存的看法：一开始我想，既然spring security

自己有保存每个登录用户的ip，也就是在WebAuthenticationDetails中，如果自己另外再保存一份，不是多余了吗，直接找到保存ip的那个地方不是最好的解决方案吗，可是搜了很久都没发现，到了准备自己另外保存ip的时候，发现ConcurrentSessionControlStrategy这里，它也是这样多存一份的，因为 WebAuthenticationDetails已经有sessionID了，但它还是自己创建一个sessionID自己保存起来，虽然这两个sessionID获取有点区别，但他们如果同时存在时，他们是一样的。既然这样我自己来保存ip不是理所当然了嘛，废话少说，改代码。

改代码就是找到源码，把它复制到自己项目的src代码中，注意包名和类名都要一样。因为tomcat在找class时，是先从自己写的代码classes目录中找，找不到时才到lib包目录中找。所以自己写的代码会优先生效。另外更可靠的办法是在原包中修改代码，改完后，重新打包。

以下是改代码的部分：

2.ConcurrentSessionControlStrategy

中的67行原来： sessionRegistry.registerNewSession(request.getSession().getId(),authentication.getPrincipal());

改为：

//getRemoteAddr()在外网使用可能不可靠，可以改进下

sessionRegistry.registerNewSession(request.getSession().getId(), request.getRemoteAddr(), authentication.getPrincipal());

152行原来：

sessionRegistry.registerNewSession(newSession.getId(),auth.getPrincipal())

改为：

sessionRegistry.registerNewSession(newSession.getId(),( (WebAuthenticationDetails)auth.getDetails()).getRemoteAddress(), auth.getPrincipal());

3.org.springframework.security.core.session.SessionRegistry中的73行：

void registerNewSession(String sessionId, Object principal);

改为： void registerNewSession(String sessionId, String remoteIp, Object principal);

4.实现类Org.springframework.security.core.session.SessionRegistryImpl中的102行

改为：

public void registerNewSession(String sessionId, String remoteIp, Object principal) {

114行改为：

sessionIds.put(sessionId, new SessionInformation(principal, sessionId, remoteIp, new Date()));

5.org.springframework.security.core.session.SessionInformation

添加属性：

private final String remoteIp;

添加get方法： public String getRemoteIp() { return remoteIp; }

52行改为：

public SessionInformation(Object principal, String sessionId, String remoteIp, Date lastRequest) {

并在这个构造函数中添加：

this.remoteIp = remoteIp;

改到这里已经完成，如果不出意外就实现查看所有登录用户的ip的功能。

取的时候在本文第二页20行作用的这段代码上改为：

SessionInformation sif = ilist.get(j);

u = (User) sif.getPrincipal();

json = json + "--------" + (j+1) +

" 用户名" + u.getUsername() +

" 最后访问时间：" + ft.format(sif.getLastRequest()) +

" ip：" + sif.getRemoteIp() +

" session:" + sif.getSessionId() +

" 限制登录：" + sif.isExpired() +

"\n";

再次运行，可看到结果：

