火龙果·整理
uml.org.cn

Spring Security 2 安全管理方式

一、该系统的安全管理方式是结合 Spring Security 2.0 实现的。

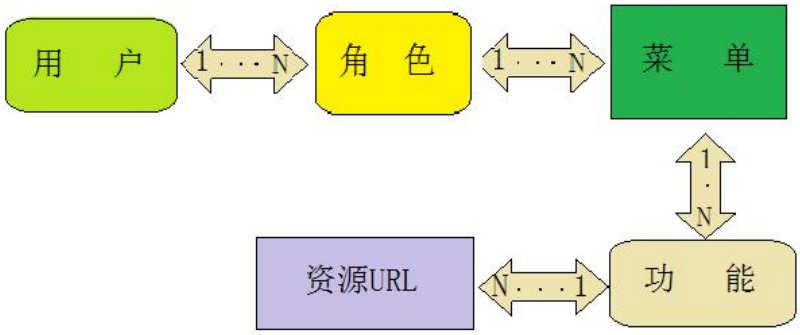
二、系统需要实现的安全管理方式有：

1. 登录时需要验证用户名和密码
2. 登录时需要加上验证码
3. 所有的数据展示及访问页面需要登录后才能访问
4. 登录后的所有系统的访问 URL 均需要授权
5. 多个用户不能使用同一个账号同时登陆系统
6. 当用户点击注销后要实现浏览器的后退按钮后退的页面失效

权限设计是采用基于角色控制的方式，用户需要访问系统的资源，首先必须要授予一个角色，而该角色具有访问系统资源的权限的能力，也可以认为是权限的集合。因此，一个用户要访问系统的某个资源（如产品列表），则首先要授予一个能够访问产品列表资源的角色（如 productAdmin）。只要任一个用户拥有了该角色，即可以访问该资源。系统的安全涉及到两个不同的概念，认证和授权。前者是验证用户是否可以进入该系统。用户进入系统的时候，首先要进行第一个操作就是进行身份认证，即 Authentication。在系统中一般表现为用户用账号跟密码登录。如果都正确了，则可以登录系统。

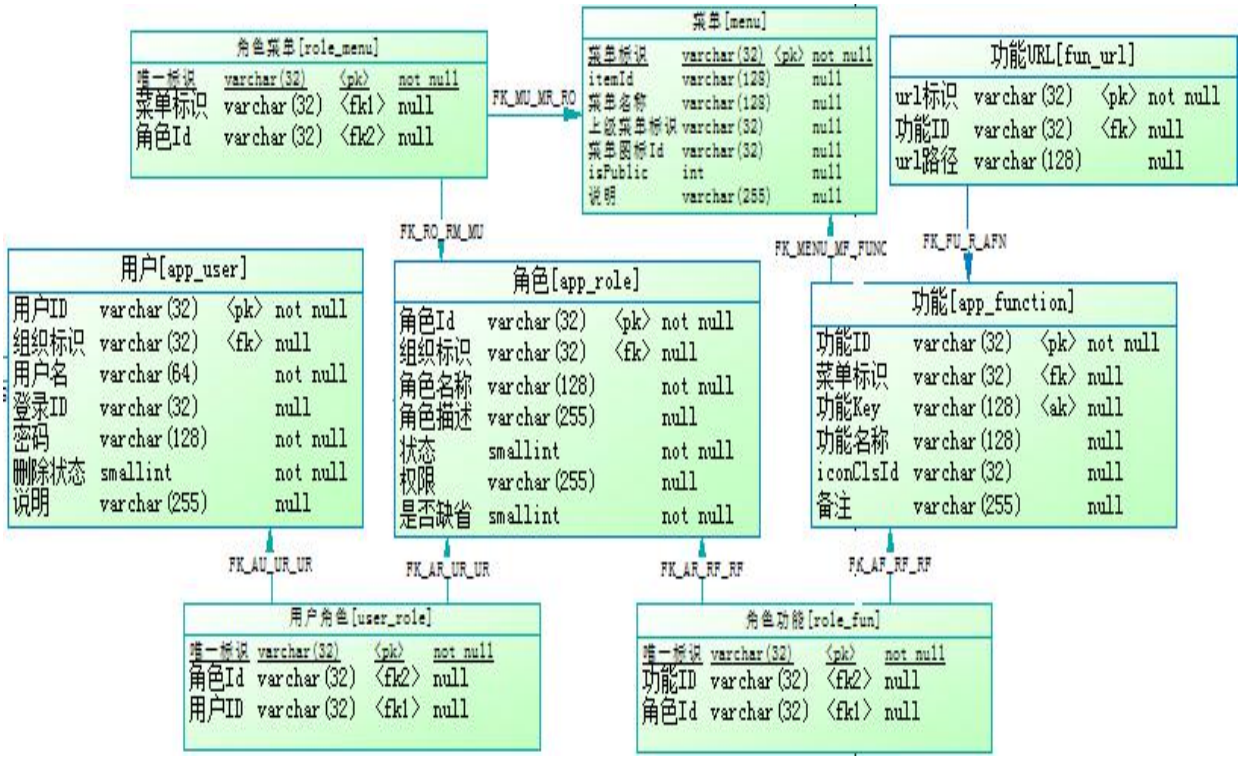
【说明】在现实中你可以这样理解，员工在进入公司之前，需要进行身份的确认。身份确认通过后，则可以进入公司。进入公司后，并不代表可以随便进入公司的每个办公室。这时就需要看当前员工具有哪些角色，即授权。授权则是关于确认用户是否允许执行一个特定的操作。如当前员工是总经理，则可以进入总经理办公室，并且可以进入普通员工的办公区域。是因为总经理已经授权可以出入这些地方。在本系统中，权限表现为功能菜单及系统访问的 URL。

因而用户、角色、权限、功能菜单之间的关系可以用如下的图描述



一个用户可以有多个角色，每个角色有多个菜单，每个菜单有多个功能，每个功能会对应多个系统访问的 URL 资源。

基于以上思想，数据库中的表设计：



app_user 系统用户表，放置系统的所有用户。

user_role 用户角色中间表，用户所拥有的角色。

app_role 角色表，放置系统的所有角色

role_menu 角色对应的菜单表，放置角色拥有的菜单

menu 菜单表，放置系统的所有菜单项

app_function 系统的功能表，放置系统菜单对应的所有功能

fun_url 系统的功能对应的权限 URL 表

role_fun 角色对应的功能表

【说明】用户表 app_user 中的密码需要保存为密文，可以防止用户的密码泄露

三、整合 Spring Security

1、为了在系统中使用 Spring Security 控制权限，首先要在 web.xml 中配置过滤器，Spring Security 是由一组的 filter 来进行统一的过滤，不同的 filter 进行相应的权限过滤功能。不过在 Security 跟 Spring 集成的过程中，其是由一个代理的类进行这些 filter 的统一管理。在 web.xml 中的配置如下：

```
<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>*.do</url-pattern>
</filter-mapping>
<filter-mapping>
     <filter-name>springSecurityFilterChain</filter-name>
     <url-pattern>/index.jsp</url-pattern>
</filter-mapping>
<filter-mapping>
     <filter-name>springSecurityFilterChain</filter-name>
     <url-pattern>/file-upload</url-pattern>
</filter-mapping>
```

所有以‘.do’结尾以及 index.jsp、文件上传的请求都会转到 DelegatingFilterProxy 类的 bean 去处理。而该 Bean 在 Spring Security 2.0 中，已经内置于安全管理的缺省的 app-security.xml 配置当中。

2、在 app-security.xml 中的配置如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<b:beans xmlns="http://www.springframework.org/schema/security"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:b="http://www.springframework.org/schema/beans"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
    http://www.springframework.org/schema/security
   http://www.springframework.org/schema/security/spring-security-2.0.4.xsd">
```

声明在 xml 中使用 Spring Security 提供的命名空间。

```
<http auto-config="true" access-denied-page="/403.jsp"
                        lowercase-comparisons="true"
                         access-decision-manager-ref="accessDecisionManager">
    <intercept-url pattern="/images/**" filters="none"/>
    <intercept-url pattern="/css/**" filters="none"/>
    <intercept-url pattern="/js/**" filters="none"/>
    <intercept-url pattern="/403*" filters="none"/>
    <intercept-url pattern="/404*" filters="none"/>
    <intercept-url pattern="/500*" filters="none"/>
    <intercept-url pattern="/ext3/**" filters="none"/>
    <intercept-url pattern="/fckeditor/**" filters="none"/>
    <intercept-url pattern="/jsonStruts**" filters="none"/>
    <form-login default-target-url="/index.jsp" login-page="/login.jsp"
                    authentication-failure-url="/login.jsp?error=true"
                                login-processing-url="/j_security_check" />
    <logout logout-url="/j_logout.do" logout-success-url="/login.jsp"/>
    <remember-me key="RememberAppUser"/>
</http>
```

① <http>标签配置如何拦截用户请求。auto-config='true'将自动配置几种常用的权限控制机制，包括 form, anonymous, rememberMe。

② access-denied-page="/403.jsp" 指自定义访问请求拒绝后显示的页面

③ lowercase-comparisons="true" 指在进行 URL 匹配前是否需要自动将 URL 中的字符都转换成小写，true 为是。

④ access-decision-manager-ref="accessDecisionManager" 指调用下面 id 为 accessDecisionManager 的访问策略管理器。

⑤ <intercept-url pattern="/images/**" filters="none"/> 标签部分是指所有的这些页面都不进行过滤处理，直接让其通过。通配符‘*’是指匹配任意多个字符，但不能跨越目录；‘**’是指可以匹配任意多个字符，可以跨越目录。

⑥ <form-login> 指 form 表单登录机制，default-target-url="/index.jsp" 设置登录成功后的页面，login-page="/login.jsp" 表示用户登陆时显示自定义的 login.jsp，authentication-failure-url="/login.jsp?error=true" 登录失败后返回的地址，当用户输入的登录名和密码不正确时，系统将再次跳转到/login.jsp，并添加一个 error=true 参数作为登陆失败的标示；
login-processing-url="/j_security_check" 提交登陆信息的 URL 地址。

⑦ <logout logout-url="/j_logout.do" logout-success-url="/login.jsp"/> 用户注销时跳转的 url，注销成功后跳转到 login.jsp 页面。

⑧ <remember-me key="RememberAppUser"/> 记住用户登录信息，当用户登录时勾选了记住我选项，默认两个星期内不需要输入用户密码，可以直接登录。

```
<b:bean     id="accessDecisionManager"
            class="org.springframework.security.vote.AffirmativeBased">
   <b:property name="allowIfAllAbstainDecisions" value="false"/>
   <b:property name="decisionVoters">
       <b:list>
           <b:bean    class="org.springframework.security.vote.RoleVoter">
                   <b:property name="rolePrefix" value="" />
           </b:bean>
           <b:bean class="org.springframework.security.vote.AuthenticatedVoter" />
       </b:list>
    </b:property>
</b:bean> ①
<authentication-manager alias="authenticationManager"/> ②
<authentication-provider user-service-ref="appUserDao">
           <password-encoder hash="sha-256" base64="true"/>
 </authentication-provider> ③
```

① 配置访问策略管理器，设置为 AffirmativeBased 策略，AffirmativeBased.java 只要有一个 voter 可以通过权限要求，就可以访问。这里是一个最小通过，只要满足其中一个条件就可以通过了。allowIfAllAbstainDecisions 属性值为 false，这样当遇到没有显式定义的被保护资源都保护起来，实际上是都投反对票，所以那些没定义的资源谁也无法访问。 <b:property name="decisionVoters"> 设置表决者，也就是哪些角色可以参与匹配，默认角色名称都要以 ROLE_开头，否则不会被计入权限控制，这里进行了修改，配置 roleVoter 的 rolePrefix 属性来改变这个前缀。

② 为认证管理器，起到认证管理的作用，对于 authenticationManager 来说，没有创建一个新实例，而是使用了“别名”（alias），因为在之前的 namespace 配置时已经自动生成了 authenticationManager 的实例，cas 只需要知道这个实例的别名就可以直接调用。它将验证的功能委托给③中的 Provider，并通过遍历 Providers, 以保证获取不同来源的身份认证，若某个 Provider 能成功确认当前用户的身份，authenticate() 方法会返回一个完整的包含用户授权信息的 Authentication 对象，否则会抛出一个 AuthenticationException。这里设置当用户登录时，会根据用户账号及密码进行身份验证，验证由 authenticationManager 来进行， 其会调用实现 UserDetailsService 接口实现类完成，设置由 appUserDaoImpl 类来实现。

appUserDaoImpl 类实现了 UserDetailsService 接口，用户 AppUser 类需要实现 UserDetails 接口，AppRole 类需实现 GranteAuthority 接口。通过重写 UserDetailsService 的 loadUserByUsername()方法返回一个存储了用户名、密码、权限等基本信息的（AppUser）UserDetails 对象。authenticationManager 通过对比用户的输入信息和 UserDetails 中的信息，来确定用户的身份和权限，完成验证工作。

<password-encoder hash="sha-256" base64="true"/> 设置密码用 sha-256 算法加密。因为在数据库中存储的密码是用 sha-256 加密的。

```
<b:bean     id="securityInterceptorFilter"
                    class="com.htsoft.core.web.filter.SecurityInterceptorFilter" >
    <custom-filter after="FILTER_SECURITY_INTERCEPTOR" />
    <b:property name="securityDataSource" ref="securityDataSource"/>
</b:bean> ①
<b:bean id="securityDataSource" class="com.htsoft.core.security.SecurityDataSource">
    <b:property name="appRoleService" ref="appRoleService"/> ②
    <b:property name="anonymousUrls">③
        <b:set>
            <b:value>/login.do</b:value>
            <b:value>/check.do</b:value>
            <b:value>/mobile/signIn.do</b:value>
        </b:set>
    </b:property>
    <b:property name="publicUrls"> ④
        <b:set>
            <b:value>/panelTreeMenu.do</b:value>
            <b:value>/file-upload</b:value>
            <b:value>/index.jsp</b:value>
            ...
        </b:set></b:property></b:bean>
```

① 配置自定义的security过滤器，在 FILTER_SECURITY_INTERCEPTOR 之后执行，处理授权认证，设置数据源为下面id为securityDataSource 的bean；其中 FILTER_SECURITY_INTERCEPTOR为用户的权限控制过滤器：

功能一：如果用户尚未登陆，则抛出 AuthenticationCredentialsNotFoundException“尚未认证异常”。

功能二：如果用户已登录，但是没有访问当前资源的权限，则抛出 AccessDeniedException“拒绝访问异常”。

功能三：如果用户已登录，也具有访问当前资源的权限，则放行。

在SecurityInterceptorFilter过滤器中通过注入securityDataSource，而SecurityDataSource又通过注入appRoleService来调用getSecurityDataSource()方法获得用户对应权限的url资源，再通过SecurityDataSource类的getDataSource()方法集合③④配置信息中可以匿名访问和公共访问的url资源，再通过SecurityInterceptorFilter类中的loadDataSource()把数据存储在roleUrlsMap对象中，最后通过doFilterInternal()方法进行验证用户访问的url是否包含在roleUrlsMap对象中，若包含就让其pass，否则不让通过。

③ 配置可以匿名访问的url，即不需要登录就可以访问的。

④ 配置只要登录就可以访问的url，即不管该用户是有无角色权限，只要成功登录就可以访问。

3、当<http>标签的属性设置为 auto-config='true'时，每次请求默认经过的过滤器表：

① HttpSessionContextIntegrationFilter

位于过滤器顶端，第一个起作用的过滤器。

用途一，在执行其他过滤器之前，率先判断用户的session中是否已经存在一个SecurityContext了。如果存在，就把SecurityContext拿出来，放到SecurityContextHolder中，供Spring Security的其他部分使用。如果不存在，就创建一个SecurityContext出来，还是放到SecurityContextHolder中，供Spring Security的其他部分使用。

用途二，在所有过滤器执行完毕后，清空SecurityContextHolder，因为SecurityContextHolder是基于ThreadLocal的，如果在操作完成后清空ThreadLocal，会受到服务器的线程池机制的影响。

② LogoutFilter

只处理注销请求，默认为/j_spring_security_logout。用途是在用户发送注销请求时，销毁用户session，清空SecurityContextHolder，然后重定向到注销成功页面。可以与rememberMe之类的机制结合，在注销的同时清空用户cookie。

③ AuthenticationProcessingFilter

处理form登陆的过滤器，与form登陆有关的所有操作都是在此进行的。默认情况下只处理/j_spring_security_check请求，这个请求是用户使用form登陆后的提交地址。此过滤器执行的基本操作时，通过用户名和密码判断用户是否有效，如果登录成功就跳转到成功页面,如果登录失败，就跳转到失败页面。

④ DefaultLoginPageGeneratingFilter

此过滤器用来生成一个默认的登录页面，默认的访问地址为/spring_security_login，这个默认的登录页面虽然支持用户输入用户名，密码，也支持rememberMe功能，不过难看了，本系统采用了自定义登录页面。

⑤ BasicProcessingFilter

此过滤器用于进行basic验证，功能与AuthenticationProcessingFilter类似，只是验证的方式不同。

⑥ SecurityContextHolderAwareRequestFilter

此过滤器用来包装客户的请求。目的是在原始请求的基础上，为后续程序提供一些额外的数据。比如getRemoteUser()时直接返回当前登陆的用户名之类的。

⑦ RememberMeProcessingFilter

此过滤器实现RememberMe功能，当用户cookie中存在rememberMe的标记，此过滤器会根据标记自动实现用户登陆，并创建SecurityContext，授予对应的权限。实际上，Spring Security中的rememberMe是依赖cookie实现的，当用户在登录时选择使用rememberMe，系统就会在登录成功后将为用户生成一个唯一标识，并将这个标识保存进cookie中，我们可以通过浏览器查看用户电脑中的cookie

⑧ AnonymousProcessingFilter

为了保证操作统一性，当用户没有登陆时，默认为用户分配匿名用户的权限。

⑨ ExceptionTranslationFilter

此过滤器的作用是处理中FilterSecurityInterceptor抛出的异常，然后将请求重定向到对应页面，或返回对应的响应错误代码。

⑩ SessionFixationProtectionFilter

防御会话伪造攻击。即在用户登录之后，销毁用户的当前session，并重新生产一个session。

⑾ FilterSecurityInterceptor

用户的权限控制都包含在这个过滤器中。功能一：如果用户尚未登陆，则抛出AuthenticationCredentialsNotFoundException“尚未认证异常”。

功能二：如果用户已登录，但是没有访问当前资源的权限，则抛出AccessDeniedException“拒绝访问异常”。

功能三：如果用户已登录，也具有访问当前资源的权限，则放行。

4、登录验证过程

Spring Security的验证过程：

Spring Security完成验证工作的主要类是AuthenticationManager,即上面介绍的过滤器<authentication-manager alias="authenticationManager"/>。这个过程从AuthenticationProcessingFilter拦截HTTP请求开始，接着，会从HTTP请求中得到用户输入的用户名和密码，并将这些输入的用户信息放到Authentication对象中，然后将这个Authentication对象传递给AuthenticationManager使用，让验证器完成用户验证功能的具体实现。在验证器的实现中，验证器会通过特有的Authentication对象，把它和服务器端取得的用户信息UserDetails对象进行对比，从而最终完成用户验证。最后将通过验证的、有效的用户信息封装在一个Authentication对象中，供以后的授权器使用。

该系统这里是让用户登录请求直接跳到action层的自定义验证方法中进行处理。实现过程：

用户登录请求会跳转到login.do地址，经过一系列的过滤器，最后到自定义过滤器*securityInterceptorFilter* 中，因为login.do是设置为允许匿名访问的，所以让其直接通过调用LoginAction.java类中login()方法对用户名、密码、验证码进行验证，并判断用户是否可用。如果验证成功就将用户信息保存到SecurityContextHolder中，然后重置session，防止会话伪造攻击；其中如果用户登录时勾选了_spring_security_remember_me选项，就会将用户信息经过加密后保存到Cookie中，保存期为两周。最后返回登录成功进入index.jsp页面。如果验证失败就返回验证失败的相关信息。

5、授权访问过程

授权的管理是通过AccessDecisionManager授权器来完成。用户访问URL 时，均需要经过Spring Security 的filterSecurityInterceptor进行授权。

首先会经过filterSecurityInterceptor（安全拦截过滤器），该过滤器首先调用认证管理器来判断用户是否已被成功验证，如果没有被验证则重定向到登录界面。否则，从Authentication获取用户的权限信息，然后从appUserDaoImpl中获取角色对应的权限信息，最后调用accessDecisionManager（访问决策管理器）来判断用户当前拥有的权限是否与当前受保护的URL资源对应的权限匹配，如果匹配就可以访问该URL资源，否则将抛出AccessDeniedException异常并返回客户端浏览器一个403错误。

系统启动时，会把所有的权限以[角色—URL 列表]的形式放置在一个全局的Map 中（在类SecurityDataSource.java中的getDataSource()方法提供实现），用户访问系统的URL 时，就会根据当前用户所拥有的角色是否包含此URL。这个全局的权限匹配源则由SecurityDataSource来提供。由于登录用户在进入系统后，都会具备一些常用的功能，所以每个用户均有一个PUBLIC_ROLE 的角色，代表可以访问系统的公共资源。配置在SecurityDataSource Bean中的publicUrls 属性中。

当上面访问均没有通过认证时， 就会进入自定义SecurityInterceptorFileter过滤器，在此类中，根据当前用户的所拥有的角色来决定是否进入下一个Filter

的处理，若没办法通过权限认证，则会抛出一个拒绝授权访问的错误，系统就会进入403 页面。403 页中写入响应的文件头“__forbidden”，EXT 展示层根据Connection 取到的文件头而弹出相应的提示信息。（实现过程在上面介绍SecurityInterceptorFileter过滤器时有详细描述）

6、多个用户不能使用同一个账号同时登陆系统

在web.xml中添加一个监听器，这个监听器会在session创建和销毁的时候通知Spring Security。

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

```
 <http  auto-config="true"  access-denied-page="/403.jsp"
         lowercase-comparisons="true"
       access-decision-manager-ref="accessDecisionManager">
       <intercept-url pattern="/images/**" filters="none"/>
       <intercept-url pattern="/css/**" filters="none"/>
       <intercept-url pattern="/js/**" filters="none"/>
       <intercept-url pattern="/403*" filters="none"/>
       <intercept-url pattern="/404*" filters="none"/>
       <intercept-url pattern="/500*" filters="none"/>
       <intercept-url pattern="/ext3/**" filters="none"/>
        <intercept-url pattern="/fckeditor/**" filters="none"/>
       <concurrent-session-control/>
       …
    </http>
```

在 Spring Security 中 concurrent-session-control 没有包含在默认生成的过滤器链中，在我们需要使用它的时候，需要自己把它添加到 http 元素中。这个 concurrent-session-control 会被放在过滤器链的最顶端，在所有过滤器使用之前起作用。concurrent-session-control 只支持两种策略：

一是允许后登陆的用户将之前登陆的用户踢出系统，先登录的用户会看到“会话已失效”的提示。这种策略实际上没办法阻止其他人在我登陆到系统之后，再使用我的账号登陆系统。最后会形成双方争先恐后的进行登录，不断将对方踢出系统的情况。

另一种是禁止用户使用已登录到系统的账户进行登录，后登陆的用户会在登录时看到“会话数量超过允许范围”的提示。这种策略需要为设置一个参数。如

下：<concurrent-session-control exception-if-maximum-exceeded="true"/> 这个参数用来控制是否在会话数目超过最大限制时抛出异常，默认值是 false，也就是不抛出异常，而是把之前的 session 都销毁掉，所以之前登陆的用户就会被踢出系统了。同时这也会导致一个比较麻烦的问题，如果用户不慎关闭了浏览器，没有通过 logout 退出系统，那么必须等到系统中用户的会话失效之后才能再次登录。如果用户不留神关了浏览器，就要再等半个小时才能再登录系统，这显然是没有道理的。

出于对实际情况的考虑，因为用户名和密码一般情况下只有用户本人知道，所以系统采用第一种策略。

Spring Security 对方法的控制：

bean method ： Spring 支持对 Service layer method 做权限判断，仅限于做角色判断。配置方式有 2 种：

1. 写在 Java 源代码里面，如： @Secured("ROLE_USER") 定义在方法前面，（该方法只有具有 USER 角色的用户能够访问，否则抛出异常）；
2. 写在配置文件里面，如：

<protect method="set*" access="ROLE_ADMIN" />

（该 bean 的所有 set 方法，只有具有 ADMIN 角色的用户能够访问，否则抛出异常）。

该系统对于权限控制到某个方法是在表现层结合Ext来处理的，因为某个方法涉及到的url如果在该用户的角色中没有分配，就算该方法按钮显示出来，用户点击也不能访问，所以只要让它不在客户端显示出来就行。下面有具体实现过程。

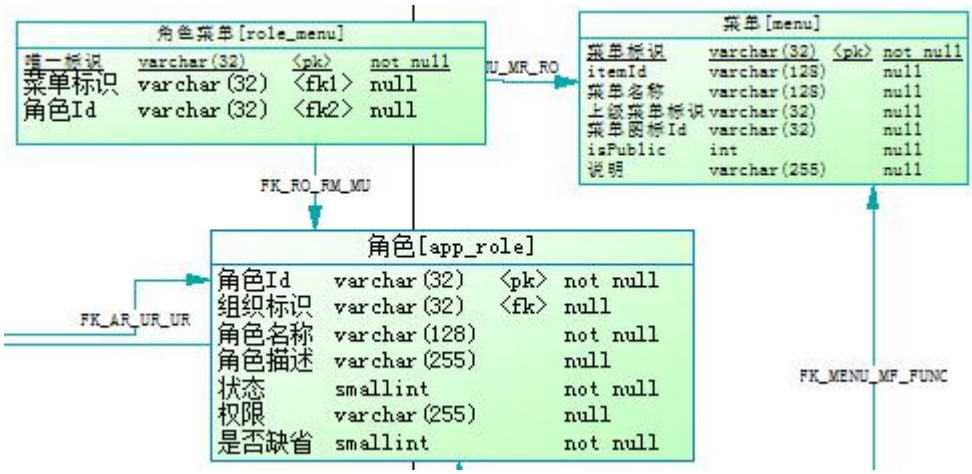
7、权限的配置及使用

以上权限的配置是基于服务端的，实际上，权限的管理并不仅限于服务端的 URL 拦截，还表现为前台功能的显示及访问。鉴于Spring Security2 的配置方式，同时能够结合EXT3 来进行整个系统的权限管理，通过使用配置文件，进行整个系统的功能集中管理，包括系统左边的导航菜单，数据列表中的功能操作权限。在传统的Web项目中，一般都是采用基于URL进行权限控制的。基于EXT 的应用也不例外，只不过我们是需要结合前台的功能菜单一起，前台显示出来的功能菜单，其后台均代表可以访问。

① 用户对应的功能菜单配置（两种方法实现）

第一种 通过在数据库建一个菜单表与角色表关联的方法实现

数据库表结构如下：



获取用户进入系统后显示的对应菜单项是通过MenuAction.java类来处理的。

在 getMenuDocument() 中 通 过 AppUser user = (AppUser)(SecurityContextHolder.getContext().getAuthentication().getPrincipal()) 获取当前用户对象，通过该用户对象获得对应的角色以及角色对应的菜单项对象；再把数据库中菜单表的所有菜单项查出来，判断其是否为公有的，是则添加到一起，利用dom4j创建一个Document对象，遍历所有菜单对象，设置对应的节点属性；这里要判断每个菜单是否有上级菜单的标识，如果有就遍历一遍Document, 将每个节点的id与子菜单的上级菜单标识进行比较，符合就添加到该节点中【说明：菜单表中添加数据要按顺序添加，即添加一个根菜单后接着添加它下面的子菜单】，然后getMenuDocument()方法将返回一个公共菜单与用户对应菜单结合的Document对象。并将该对象存储在session中，以便下次登录无需再从数据库中去取。最后通过panelTree()方法将返回的Documnet对象 转换成gson类型的字符串返回给客户端 。

测试方法如下：

下面为怎么将从数据库中取出的菜单转换成Document对象的方法（出

于对实际情况的考虑，只做了三级菜单的处理，可以扩展）

```
public static Document getMenuDocument(){
        //itemList为该用户对应的所有菜单，包括公共菜单，下面是个demo
        List<Items> itemList = new ArrayList<Items>();
        for (int i = 0; i < 5; i++) {
            Items item = new Items();
            item.setItemId("Info"+i);
            item.setItemName("信息管理");
            item.setPictureId("menu-info"+i);
            item.setIsPublic(1);
            if(i==1)item.setItemSign("Info"+0);
            if(i==3)item.setItemSign("Info"+2);
            itemList.add(item);
        }
        Document doc = DocumentHelper.createDocument();  //创建一个document对象
        doc.addElement("Menus");      //创建一个root节点
        for (Items items : itemList) {
            Element root = doc.getRootElement();
            if(items.getItemSign()!=null){
                doc = travelDoc(doc,items.getItemSign(),items);
            }else{
                Element rootElement1 = root.addElement("Items")
                .addAttribute("id", items.getItemId())
                .addAttribute("text", items.getItemName())
                .addAttribute("iconCls", items.getPictureId())
                .addAttribute("isPublic", items.getIsPublic().toString());
            }
        }
         //创建一个leftMenu.xml文件  并写入创建的属性
        try {
            FileWriter out = new FileWriter("leftMenu.xml");//定义一个输出格式
            OutputFormat format = OutputFormat.createPrettyPrint();
            XMLWriter writer = new XMLWriter(out,format);
            writer.write(doc);
            writer.flush();
            writer.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
        return doc;
    }
```

```
//添加子节点
public static Document travelDoc(Document doc,String str,Items items){
    Element root = doc.getRootElement();
      for(Iterator ite = root.elementIterator(); ite.hasNext();){
       Element eChild = (Element) ite.next();
            //判断属性id 是否与上级菜单标识相等
       if(eChild.attribute("id").getValue().equals(str)){
          eChild.setName("Items");
          eChild.addElement("Item")
          .addAttribute("id", items.getItemId())
          .addAttribute("text", items.getItemName())
          .addAttribute("iconCls", items.getPictureId())
        .addAttribute("isPublic", items.getIsPublic().toString());
        }
        for(Iterator itc = eChild.elementIterator(); itc.hasNext();){
            Element eCC = (Element) itc.next();
            if(eCC.attribute("id").getValue().equals(str)){
                eCC.setName("Items");
                eChild.addElement("Item")
                .addAttribute("id", items.getItemId())
                .addAttribute("text", items.getItemName())
                .addAttribute("iconCls", items.getPictureId())
                .addAttribute("isPublic",
                    items.getIsPublic().toString());
             }
            }
         }
         return doc;
    }
```

第二种 通过xml文件配置来实现，如下：

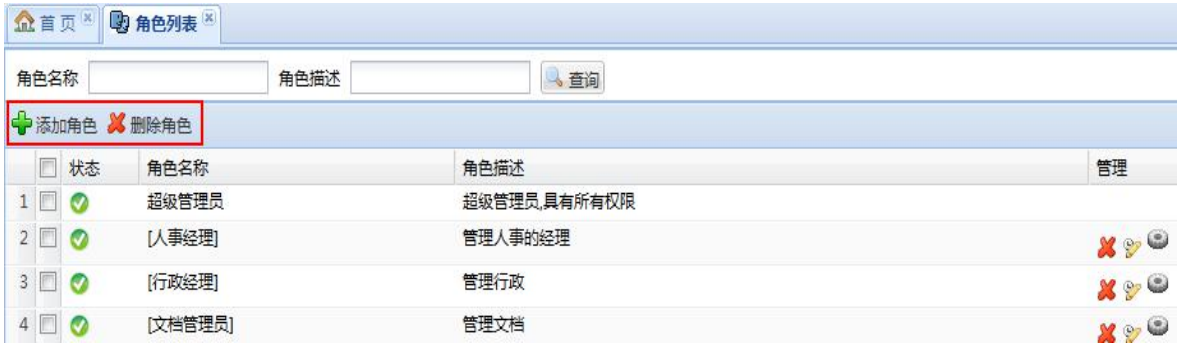
先在menu.xml 中配置系统的功能，如我们配置角色管理的权限，如下所示：

```
<Items id="SystemSetting" text="系统设置" iconCls="menu-system"
                                                    isPublic="false">
        ......
        <Item id="AppRoleView" iconCls="menu-role" text="角色设置">
            <Function id="_AppRoleList" text="查看角色" iconCls="menu-list">
                <url>/system/listAppRole.do</url>
            </Function>
            <Function id="_AppRoleAdd" text="添加角色" iconCls="menu-add">
                <url>/system/listAppRole.do</url>
                <url>/system/saveAppRole.do</url>
            </Function>
            <Function id="_AppRoleEdit" text="编辑角色" iconCls="btn-edit">
                <url>/system/listAppRole.do</url>
                <url>/system/saveAppRole.do</url>
                <url>/system/getAppRole.do</url>
            </Function>
            <Function id="_AppRoleDel" text="删除角色" iconCls="menu-del">
                <url>/system/listAppRole.do</url>
                <url>/system/mulDelAppRole.do</url>
            </Function>
            <Function id="_AppRoleGrant" text="授权角色" iconCls="btn-grant">
                <url>/system/listAppRole.do</url>
                <url>/system/grantXmlAppRole.do</url>
                <url>/system/getAppRole.do</url>
                <url>/system/grantAppRole.do</url>
            </Function>
        ......
        </Item>
    </Items>
  <!-- 其他模块 -->
```

menu.xml 为整个系统的功能列表，系统的左边菜单也是从该文件显示出来，menu.xml 是在系统启动时，由AppUtil.java 中的init()方法进行初始化，并构造一个显示左边的菜单的全局Document，同时通过MenuAction.java类中的方法构造参与授权的Document. 每个用户进入系统后，会根据当前用户所拥有的角色及角色对应的权限，进行显示授权的菜单及功能按钮。

② 界面功能划分：

当进入某项菜单中，会进入相应模块的管理界面，如角色管理：



上面的功能“添加角色”，“删除角色”以及管理一栏中的“删除”，“编辑”，“授权”等功能，则对应为menu.xml 中的Item 下的Function 标签的配置，如：

```
<Item id="AppRoleView" iconCls="menu-role" text="角色设置">
    <Function id="_AppRoleList" text="查看角色" iconCls="menu-list">
        <url>/system/listAppRole.do</url>
    </Function>
    <Function id="_AppRoleAdd" text="添加角色" iconCls="menu-add">
        <url>/system/listAppRole.do</url>
        <url>/system/saveAppRole.do</url>
    </Function>
    <Function id="_AppRoleEdit" text="编辑角色" iconCls="btn-edit">
        <url>/system/listAppRole.do</url>
        <url>/system/saveAppRole.do</url>
        <url>/system/getAppRole.do</url>
    </Function>
   <Function id="_AppRoleGrant" text="授权角色" iconCls="btn-grant">
        <url>/system/listAppRole.do</url>
        <url>/system/grantXmlAppRole.do</url>
        <url>/system/getAppRole.do</url>
        <url>/system/grantAppRole.do</url>
    </Function>
</Item>
```

添加角色对应：

```
<Function id="_AppRoleAdd" text="添加角色" iconCls="menu-add">
    <url>/system/listAppRole.do</url>
    <url>/system/saveAppRole.do</url>
</Function>
```

其下包括两个 Url：
一个是/system/listAppRole.do，另一个是/system/saveAppRole.do，表示当我们访问这个功能完成角色添加时，需要访问这两个 URL，后台会根据当前用户的角色是否授权访问这个功能而决定是否开放这两个 URL 给当前用户访问，否则，前台浏览器会弹出一个类似如下的提示，
告诉用户当前没有权限访问这个 URL。

当然，这种情况还是要避免出现，因为系统在客户端出现的功能菜单，均是有权访问的。

③ 前台的代码配置：

前台的用户登录系统后，会有一个全局的变量 curUserInfo 存储用户的所有权限（其会把所有的角色的配置抽取出来，去掉重复的权限配置），可以在客户端直接检查用户是否有权限访问某一个菜单的功能，如我们检查用户是否直接有访问“添加角色”的功能，则可以调用系统提供的方法：

isGranted(‘_AppRoleAdd’);

该函数返回为 true 则代表可以访问。

在 App.js 文件中通过 init 函数加载 curUserInfo 用户信息，包括他的权限对应的 url 资源。

如果我们要进行更高的权限粒度控制，需要需改如下：
对 AppRoleView.js 进行更改。

```
var toolbar = new Ext.Toolbar({
            id : 'AppRoleFootBar',
            height : 30,
            bodyStyle:'text-align:left',
            items : []
        });
```

```
if(isGranted('_AppRoleAdd')){
        toolbar.add(new Ext.Button({
                iconCls : 'btn-add',
                text : '添加角色',
                handler : function() {
                new AppRoleForm();
                }
        }));
    }
if (isGranted('_AppRoleDel')) {
        toolbar.add(new Ext.Button({
                iconCls : 'btn-del',
                text : '删除角色',
                handler : function() {
                        var grid = Ext.getCmp("AppRoleGrid");
                        var selectRecords = grid.getSelectionModel().getSelections();
                        if (selectRecords.length == 0) {
                        Ext.Msg.alert("信息", "请选择要删除的记录！ ");
                        return;
                }
                var ids = Array();
                for (var i = 0; i < selectRecords.length; i++) {
                       ids.push(selectRecords[i].data.roleId);
                }
                AppRoleView.remove(ids);
                }
        }));
   }
```

管理列中的栏目功能：

```
if(isGranted('_AppRoleDel'))
      str = '<button title="删除" value=" " class="btn-del"
        onclick="AppRoleView.remove('+editId + ')"></button>';
if(isGranted('_AppRoleEdit'))
      str += ' <button title="编辑" value=" " class="btn-edit"
      onclick="AppRoleView.edit('+ editId + ')"></button>';
if(isGranted('_AppRoleGrant'))
      str += ' <button title="授权" value=" " class="btn-grant"
      onclick="AppRoleView.grant('+ editId + ',\" + roleName + '\')"> </button>';
```

8、用户注销

当用户点击系统注销时，会跳到 j_logout.do 地址，

```
<logout logout-url="/j_logout.do" logout-success-url="/login.jsp"/>
```

根据上面配置经过 LogoutFilter 过滤器，调用 LogoutFilter.java 类中的相关方法销毁 session，清空 Cookie，同时返回登录页面。

补充：

对于某个用户只能看到其对应的相关信息，以及对某些信息只有读取的权限，而对另一些信息具有 CRUD 权限的实现：

（这里所说的信息资源包括用户、合同、项目、简历等各种可操作信息）
创建一个与用户关联的信息资源表

```
CREATE TABLE `user_resources` (
  `resourcesId` varchar(32) NOT NULL,
  `className` varchar(256) NOT NULL DEFAULT '' COMMENT '实体类名',
  `dataId` varchar(32) NOT NULL DEFAULT '' COMMENT '数据 id',
  `userId` varchar(32) NOT NULL COMMENT '用户 id',
  `delSign` tinyint(1) NOT NULL DEFAULT '0' COMMENT '是否可以删除的标记',
  `updateSign` tinyint(1) NOT NULL DEFAULT '0' COMMENT '是否可以修改的标记',
  PRIMARY KEY (`resourcesId`),
  CONSTRAINT `FK_user_resources` FOREIGN KEY (`userId`) REFERENCES `app_user` (`userId`) ON UPDATE NO ACTION
)
```

当我们添加了一条信息资源时，也要在资源表 user_resources 中记录这条信息的 Id，所属实体类类名，所有者 Id，删除和修改标记。即需要进行权限控制的信息实体类添加数据时，在其事务层做添加处理后还要调用添加资源表数据的方法，把这条数据对应添加到资源表中。

当我们删除用户对象时，也要删除对应的资源表中的资源信息。即删除某条数据时，在其事务层做删除操作后还要调用删除资源表中对应数据的方法。

首先当用户进入系统时就把资源表中用户对应的信息记录加载到内存中，因为用户资源表与用户表有关联，所以用户对应的资源表信息已经保存在 SecurityContextHolder 对象中。

1、当用户访问某个类型的资源时，先从内存的资源记录中找到该类型的资源记录，然后把 list 方法返回的结果集中数据 id 与资源记录的 id 不相等的过滤掉。即可实现用户只能看到其对应的资源。

2、根据用户的资源是否具有删除、修改的标记来觉定用户是否具有对这条数据的删除和修改操作权限。
相关方法的实现都可以封装到一个类中，公共调用。