阅读笔记10

关于阅读笔记10 这个是 对阅读笔记9的补充

以下内容绝大部分都是来自复制与极客网上的wiki

在之前的对认证信息的来源处理
除了之前的几种 还有可以

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。

这个跟之前的用数据库是一样的 但是这样非常的不方便 只是在此介绍一下

但是这种还是有特殊的地方的 这就在于用户权限与用户组权限

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>
```

另外 我们可以使用一种算法对密码进行加密

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

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。 ``` ``` #### 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。 ``` ``` 2.下面的配置将使用 UserDetails 的 username 作为 salt。 ``` ``` 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。 ``` ``` 需要注意的是 AuthenticationProvider 进行认证时所使用的

另外 我们可以自己插入自己的filter

Filter

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。

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。

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

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。

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。

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。

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。

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">
      <bean class="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。

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>
```

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>
```

匿名认证

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。

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"/>
```

```
<anonymous enabled="true" key="doesNotMatter" username="anonymous" granted-authority="ROLE_ANONYMOUS"/>
```

异常信息本地化

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。

缓存UserDeatils

另外还是有些重要的地方需要说明 其中之一 就是核心类

核心类简介

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。

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。

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。

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。

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。

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>
```

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
```

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。

来源： http://wiki.jikexueyuan.com/project/spring-security/core-classes.html

缓存UserDeatils

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() {
        returncache;
    }

    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">
         <bean class="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》。

session管理

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。

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。

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 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 指定我们的 session-authentication-error-url 不受 Spring Security 控制呢？把它换成 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 设置为不受 Spring Security 控制，而不是使用。

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>
```

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。

- 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。
- none：表示继续使用原来的 session。
- newSession：表示重新创建一个新的 session，但是不 copy 原 session 拥有的 attribute。

我将这些功能做了一个demo
这里的具体配置文件如下

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


<!--全局方法权限控制-->
  <security:global-method-security secured-annotations="enabled" jsr250-annotations="enabled" pre-post-
annotations="enabled">
        <security:protect-pointcut expression="execution(* com.test.Spitter.service.MessageServiceImpl.admin*(..))"
access="ROLE_ADMIN"/>
         </security:global-method-security>




   <!--使用bean的方式进行手动方法权限控制-->

    <bean id="service" class="com.test.Spitter.service.MessageServiceImpl">
      <!-- <security:intercept-methods>
          <security:protect method="adminMessage" access="ROLE_USER"/>
          <security:protect method="adminDate" access="ROLE_USER"/>
       </security:intercept-methods>-->
    </bean>

   <bean id="service2" class="com.test.Spitter.service.securitCeshi"/>




 <!--具体拦截以及登录页面配置的地方-->
   <security:http security="none" pattern="/Test/Test" />
   <security:http security="none" pattern="/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 auto-config="true"  use-expressions="true">
        <security:form-login  login-page="/login.do"
                        login-processing-url="/login" username-parameter="username"
                        password-parameter="password"
                authentication-success-handler-ref="authSuccess"
                authentication-failure-url="/login_failure"
            />



    <!-- Session管理-->
     <!--检测 session 超时 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-->
        <!--通常的session登录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。session-management 提供了这种限制。--
>
        <security:session-management invalid-session-url="/login.do"
                        session-authentication-error-url="/none/session_authentication_error.jsp">
            <!--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:concurrency-control  max-sessions="1"
                error-if-maximum-exceeded="true"
                        expired-url="/none/session_authentication_error.jsp"
                />
            <!-- 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
            如果你的登陆时Remember me功能登陆的  session失败的地址就是 session-authentication-error-url定义的地址了
            -->
            <!--expired-url 由于其再次登录导致 session 超时的 session 时所要跳转的页面-->
        </security:session-management>



        <!--匿名认证
        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。
         匿名认证就是ROLE_ANONYMOUS权限 -->
        <security:anonymous enabled="true" key="doesNotMatter" username="anonymous" granted-
authority="ROLE_ANONYMOUS"/>

        <!--Remember-me功能 暂时出错 -->
          <security:remember-me key="Test"   data-source-ref="dataSource"  user-service-ref="userDetailsService" />
        <!--退出登录-->
        <security:logout logout-url="/logout.do"/>


        <security:http-basic/><!--注意当我们同时定义了 http-basic 和 form-login 元素时，form-login 将具有更高的优先级。
        功能就是 即在需要认证的时候 将引导我们到登录页面，而不是弹出一个窗口。-->


        <!--具体配置页面所需要的权限-->
        <security:intercept-url pattern="/login.do" requires-channel="http"/>
        <security:intercept-url pattern="/login.do" access="hasRole('ROLE_ANONYMOUS')"/>
        <security:intercept-url pattern="/login_failure" access="hasRole('ROLE_ANONYMOUS')"  /><!--指定请求方法-->
        <security:intercept-url pattern="/admin/**" access="hasRole('ROLE_ADMIN') and hasIpAddress('192.168.1.1')"/>
        <security:intercept-url pattern="/spitter/form" requires-channel="https"/>
        <security:intercept-url pattern="/home" requires-channel="http"/>
        <security:intercept-url pattern="/**" access="hasRole('ROLE_USER')" /><!--表示请求的用户应当具有 ROLEUSER 角色-->
    </security:http>
```

```
    <!--修改默认登录后的返回地址的类 指定自己的返回地址-->
    <bean id="authSuccess" class="com.test.Spitter.securityController.AuthenticationSuccessHandlerImpl"/>

    <!--LDAP嵌入式服务器-->
    <!--<security:ldap-server id="userServer" root="dc=habuma,dc=com" ldif="users.ldif"></security:ldap-server>-->

    <!--认证信息配置管理器-->
    <security:authentication-manager>
            <security:authentication-provider user-service-ref="ceshi">
                <!--下面这个是进行密码加密的功能 出错 原因可能是认证与保存用户密码的时候
            没有采用相同的加密算法与规则 -->
                <!--  <security:password-encoder hash="md5" base64="true">
                    <security:salt-source user-property="username"/>
                </security:password-encoder>-->

            </security:authentication-provider>
            <security:authentication-provider user-service-ref="ceshi2" >
            </security:authentication-provider>
        <!--LDAP服务供应者-->
         <!--  <security:ldap-authentication-provider   server-ref="userServer"  user-search-base="ou=people" user-search-
filter="(uid={0})" group-search-base="ou=groups"
                                    group-search-filter="member=0">
              <security:password-compare hash="md5" password-attribute="password"></password-compare>
          </security:ldap-authentication-provider>-->

    </security:authentication-manager>


<!--认证信息 1-->
    <security:user-service id="ceshi">
        <!-- 这里创建一个用户，可以通过用户名密码登录 -->
        <security:user name="user" password="user" authorities="ROLE_USER"  />
    </security:user-service>


<!--认证信息JDBC 2-->
    <security:jdbc-user-service id="ceshi2" data-source-ref="dataSource"
                    users-by-username-query="
                    SELECT username,password,enabled  FROM  users WHERE username=?
    " authorities-by-username-query="
    SELECT  username,authority FROM  authorities WHERE  username =?"  role-prefix="ROLE_"/>
    <!-- role-prefix 可以用来指定角色的前缀。-->




    <!--remember-me功能使用的配置Bean-->
    <bean id="userDetailsService" class="org.springframework.security.core.userdetails.jdbc.JdbcDaoImpl">
        <property name="dataSource" ref="dataSource"/>
    </bean>



    <!--异常信息本地化-->
    <bean id="messgaeSource" class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
      <property name="basenames">
          <array>
```

```
            <value>classpath*:messages_zh_CN.properties</value>
            <value>org.springframework.security.messages</value>
        </array>
    </property>
</bean>


<!--数据库配置-->
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
    <property name="url" value="jdbc:mysql://localhost:3306/test"/>
    <property name="username" value="root"/>
    <property name="password" value="liuziye"/>
</bean>

    </beans>
```