初识Spring Security

极客学院出版

前言

Spring Security，这是一种基于 Spring AOP 和 Servlet 过滤器的安全框架。它提供全面的安全性解决方案，同时在 Web 请求级和方法调用级处理身份确认和授权。本教程对 Spring Security 的使用进行一个比较全面的简要介绍。

适用人群

帮助 Java 工程师了解企业级安全认证框架。

学习前提

学习本教程前，你需要对 AOP 、Servlet 等概念有所了解，如果你熟悉 Java 语言，那么学起来会很容易。

鸣谢：http://www.iteye.com/blogs/subjects/spring_security

目录

1 初体验

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。

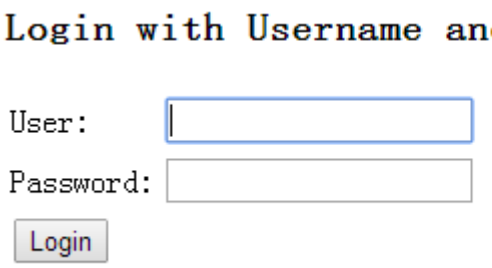
```
  <context-param>
    <param-name>contextConfigLocation</param-name>
  <param-value>/WEB-INF/config/applicationContext.xml,/WEB-INF/config/spring-security.xml</param-value>
  </context-param>

  <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。

2

关于登录

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。

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 请求使用不能的 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>
```

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 AuthenctictionSuccessHandler 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"/>
```

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

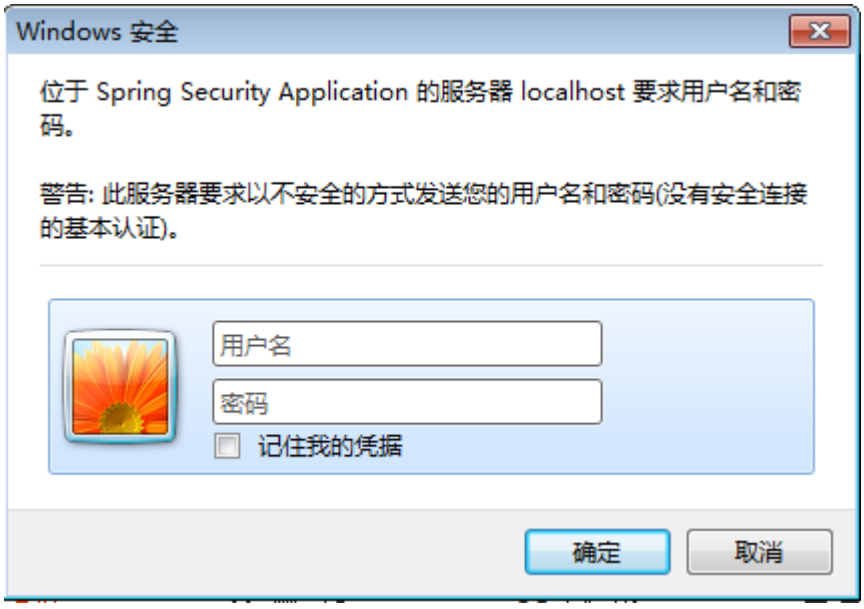
类似于 authentication-success-handler-ref，authentication-failure-handler-ref 对应一个用于处理认证失败的 AuthenticationFailureHandler 实现类。指定了该属性，Spring Security 在认证失败后会调用指定 AuthenticationFailureHandler 的 onAuthenticationFailure 方法对认证失败进行处理，此时 authentication-failure-url 属性将不再发生作用。

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。

3

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
```

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。

4

认证简介

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。

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。

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。

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。

5

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。

6

AuthenticationProvider

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>
```

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。

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。

```
mysql> desc users;
+----------+-------------+------+-----+---------+-------+
| Field    | Type        | Null | Key | Default | Extra |
+----------+-------------+------+-----+---------+-------+
| username | varchar(50) | NO   | PRI | NULL    |       |
| password | varchar(50) | NO   |     | NULL    |       |
| enabled  | tinyint(1)  | NO   |     | NULL    |       |
+----------+-------------+------+-----+---------+-------+
3 rows in set (0.01 sec)

mysql> desc authorities;
+-----------+-------------+------+-----+---------+-------+
| Field     | Type        | Null | Key | Default | Extra |
+-----------+-------------+------+-----+---------+-------+
| username  | varchar(50) | NO   | PRI | NULL    |       |
| authority | varchar(50) | NO   | PRI | NULL    |       |
+-----------+-------------+------+-----+---------+-------+
```

```
mysql> desc groups;
+------------+-------------+------+-----+---------+----------------+
| Field      | Type        | Null | Key | Default | Extra          |
+------------+-------------+------+-----+---------+----------------+
| id         | int(11)     | NO   | PRI | NULL    | auto_increment |
| group_name | varchar(50) | NO   |     | NULL    |                |
+------------+-------------+------+-----+---------+----------------+
2 rows in set (0.01 sec)

mysql> desc group_authorities;
+-----------+-------------+------+-----+---------+-------+
| Field     | Type        | Null | Key | Default | Extra |
+-----------+-------------+------+-----+---------+-------+
| group_id  | int(11)     | NO   | MUL | NULL    |       |
| authority | varchar(50) | NO   |     | NULL    |       |
+-----------+-------------+------+-----+---------+-------+
2 rows in set (0.01 sec)

mysql> desc group_members;
+----------+-------------+------+-----+---------+----------------+
| Field    | Type        | Null | Key | Default | Extra          |
+----------+-------------+------+-----+---------+----------------+
| id       | int(11)     | NO   | PRI | NULL    | auto_increment |
| username | varchar(50) | NO   |     | NULL    |                |
| group_id | int(11)     | NO   | MUL | NULL    |                |
+----------+-------------+------+-----+---------+----------------+
```

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。

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>
```

PasswordEncoder

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：

<

table> **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

<

table>

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"));
```

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。

7

缓存 UserDetails

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.CachingUserDetailsS
  <!-- 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》。

8

intercept-url配置

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"/>
```

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>
```

指定访问协议

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>
```

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。

9

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>
```

10

退出登录 logout

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。

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_
```

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。

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。

12

Remember-Me 功能

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。

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>
```

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>
```

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。

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.Username
  <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 TokenBasedRemember MeServices 提供的，所以在使用时需要保证 RememberMeAuthenticationProvider 和 TokenBasedReme mberMeServices 的 key 属性值保持一致。需要配置 UsernamePasswordAuthenticationFilter 的 rememb erMeServices 为我们定义好的 TokenBasedRememberMeServices，把 RememberMeAuthenticationPr ovider 加入 AuthenticationManager 的 providers 列表，并添加 RememberMeAuthenticationFilter 和 Us ernamePasswordAuthenticationFilter 到 FilterChainProxy。

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 TokenBasedRememberMeS ervices 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，InMemoryTokenR epositoryImpl 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">
    <bean class="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>
```

13

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-err

or-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。

14

权限鉴定基础

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。

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。

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);
}
```

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 ConfigAttr

ibute 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 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。

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());
}
```

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。