Spring Security文档第六章翻译

6.1 介绍

从Spring框架的2.0版本开始，命名空间配置就被提供了。它允许您用传统的Spring bean应用上下文语法和来自其他XML模式的元素进行补充。你可以在Spring Reference Documentation中找到更多信息。命名空间元素给我们提供了简洁的配置一个bean的方法，更强大的是它能让我们去定义更接近于待解决问题域的语法来掩盖底层的复杂性。一个简单的元素能隐藏多个bean和处理步骤以及被添加到应用上下文中的事实。例如，使用下面的安全命名空间配置到应用上下文，那么将会在你的应用上下文中打开一个嵌入式LDAP服务器，方便测试。

```
<security:ldap-server />
```

这比连接相应的Apache Directory Server bean简单得多。最一般的可选配置需求可以通过`ldap-server`元素的属性进行配置，这样就使用户不用担心哪个bean被创建以及哪个属性的名字是啥。使用一个好的XML编辑器可以提示你每个元素有哪些可用的属性。我们建议您尝试使用Spring Tool Suite，因为它具有用于处理标准Spring命名空间的特殊功能。

为了开始在你的application context中使用安全命名空间，你需要去把`spring-security-config`的jar包放到你的classpath里。然后你需要做的就是把下面的模式声明添加到你的应用application context文件中：

```
<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-3.0.xsd
http://www.springframework.org/schema/security
http://www.springframework.org/schema/security/spring-security.xsd">
...
</beans>
```

在很多例子中，你会看到，我们将使用“security”作为默认的命名空间，而不是“beans”，这样做意味着我们能省略所有的安全命名空间元素的前缀，让内容更易读。如果你的应用程序上下文被分成多个文件，并且你的安全配置大部分在其中一个里，你也会考虑这么做。这时，你的应用程序上下文文件应该按照下面的方式开头：

```
<beans:beans xmlns="http://www.springframework.org/schema/security"
xmlns:beans="http://www.springframework.org/schema/beans"
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-3.0.xsd
        http://www.springframework.org/schema/security
        http://www.springframework.org/schema/security/spring-security.xsd">
...
</beans:beans>
```

我们将假设这章你使用的就是上述的语法。

6.1.1 命名空间设计

命名空间的设计主要关注于框架的一般应用以及提供一个简便简洁的方式去在你的程序中启动这些应用的语法。该设计基于框架内的大规模依赖关系，能被分成如下部分：

- Web/HTTP Security - 最复杂的部分。设置用于应用框架认证机制的过滤器和相关服务bean，保护URL，呈现登录和错误页面等等。
- Business Object(Method) Security - 保护服务层安全的选项。
- AuthenticationManager - 处理来自框架其他部分的验证请求。
- AccessDecisionManager - 为网络和方法安全提供访问决策。默认的一个将被注册，但是你可以选择去自定义一个，并使用标准Spring bean语法去声明。
- AuthenticationProviders - 身份验证管理员验证用户身份的机制。命名空间提供了几个标准选项并且也提供了方法去添加一个使用传

统语法声明的bean。

- UserDetailsServie - 与验证提供者密切相关，但是也被其他的组件所需要。

我们将会在下面的章节看到如何去配置这些。

6.2 开始使用Security命名空间配置

在这节，我们将会看到如何让你去通过命名空间配置使用框架的一些主要功能。让我们现在假想你想要尽可能快的启动一个已经存在的项目并且添加验证支持和访问控制，还有一些登录测试。然后我们会看到如何基于数据库或者其他安全仓库改变验证方式。在之后的章节我们将介绍更多的高级命名空间配置的选项。

6.2.1 web.xml配置

你需要做的第一件事就是把下面的过滤器声明添加到`web.xml`文件中：

```
<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>
```

这为Spring Security web基础结构提供了一个钩子。`DelegatingFilterProxy`是一个Spring Framework 类，这个类代理了一个过滤器实现，在你的应用程序上下文中被作为一个Spring bean进行定义。在这个例子中，这个bean的名字叫做"springSecurityFilterChain"，它是一个被处理web Security的命名空间创造的内部基础结构bean。注意你自己不要用这个名字作为其他bean的名字。一旦你已经添加了上述内容到你的`web.xml`，你就已经做好了去编辑你的应用上下文文件的准备。Web安全服务通过`<http>`元素进行配置。

6.2.2 一个最小的配置

我们需要的去启动web安全所有配置如下：

```
<http>
    <intercept-url pattern="/**" access="hasRole('USER')" />
    <form-login />
    <logout />
</http>
```

上述的配置对我们应用程序中所有的URL进行了保护，只有具有`ROLE_USER`角色的人才能访问，我们想要去通过使用一个包含username和password的表达去进行登录，并且我们想要一个被注册的注销URL去允许我们从应用中注销。`<http>`元素是所有和web相关的命名空间功能的父元素。`<intercept-url>`元素定义了一个`pattern`，当传入请求时，使用这个pattern采用ant风格的语法进行路径匹配。您也可以使用正则表达式匹配作为替代（有关更多详细信息，请参阅命名空间附录）。`access`属性定义了匹配给定模式的请求的访问需求（话不太好理解，其实就是当一个请求传入时，只有既匹配URL模式，又满足access属性条件，才能正常访问）。这个属性中一般情况下是个逗号分隔的列表，每个条件都要满足才能进行访问。"ROLE_"是一个前缀，用于和用户的授权进行区分。换句话说，一个基本的基于用户的检验应该被使用。在Spring Security中访问控制不应该被限制到单一角色的使用里（因此使用前缀来区分不同类型的安全属性）。

你可以使用多个`<intercept-url>`元素来为多个URL进行访问控制。但是它们将被放在一个队列，然后第一个匹配到的会被使用。所以你一定要吧最特别匹配的放在最前面。你也可以在`method`属性中对HTTP方法进行限制。

为了添加几个用户，你可以定义在命名空间中定义一些测试数据：

```
<authentication-manager>
<authentication-provider>
    <user-service>
    <!-- password是以前缀{noop}开始的，这是为了指明使用的DelegatingPasswordEncoder是NoOppasswordEncoder。
这在产品中是不安全的，但是在这里更容易阅读。正常情况下密码都应该使用BCryt进行哈希。 -->
    <user name="jimi" password="{noop}jimipassword" authorities="ROLE_USER,ROLE_ADMIN"/>
```

```
    <user name="bob" password="{noop}bobpassword" authorities="ROLE_USER,ROLE_ADMIN"/>
    </user-service>
</authentication-provider>
</authentication-manager>
```

这是一个简单的存储相同密码的简单例子。密码的前缀是{bcrypt}，用于指示DelegatingPasswordEncoder，它支持任何已配置的PasswordEncoder进行匹配，并使用BCrypt对密码进行散列。

```
<authentication-manager>
<authentication-provider>
<user-service>
<user name="jimi" password="{bcrypt}<img src="https://latex.codecogs.com/gif.latex?2a"/>10<img
src="https://latex.codecogs.com/gif.latex?
ddEWZUl8aU0GdZPPpy7wbu82dvEw&#x2F;pBpbRvDQRqA41y6mK1CoH00m&amp;quot;
authorities=&amp;quot;ROLE_USER,%20ROLE_ADMIN&amp;quot;%20&#x2F;&gt;
&lt;user%20name=&amp;quot;bob&amp;quot;%20password=&amp;quot;{bcrypt}"/>2a<img
src="https://latex.codecogs.com/gif.latex?10"/>/elFpMBnAYYig6KRR5bvOOYeZr1ie1hSogJryg9qDlhza4oCw1Qka"
                authorities="ROLE_USER" />
<user name="jimi" password="{noop}jimispassword" authorities="ROLE_USER, ROLE_ADMIN" />
<user name="bob" password="{noop}bobspassword" authorities="ROLE_USER" />
</user-service>
</authentication-provider>
</authentication-manager>
```

如果你熟悉框架的命名空间版本，你可能已经可以大概猜出这里发生了什么。`<http>`元素负责创建一个`FilterChainProxy`和它使用的过滤器bean。一般的问题，像以前的过滤器顺序问题已经不会出现了，因为过滤器的位置已经被提前定义了。`<authentication-provider>`元素创建了一个`DaoAuthenticationProvider`bean，`<user-service>`元素创建了一个`InMemoryDaoImpl`。所有的`authentication-provider`元素一定要是`authentication-manager`元素的子元素，它创建了一个`ProviderManager`并且为其注册权限提供者。你可以在命名空间附录中找到更多细节。它是很值得彻底研究的，如果你想要理解框架中的重要的类是什么以及他们是如何被使用的，尤其是你想之后自定义它们。

上面的配置定义了两个用户，他们在应用程序中的密码和角色（这些用于权限控制）。可以使用`user-service`上的`properties`属性从一个标准的properties文件中加载数据。关于文件格式的详细信息请看基于内存的验证章节。使用`<authentication-provider>`元素意味着用户信息将被验证管理员使用去处理验证请求。你有多个`<authentication-provider>`元素去定义不同的验证源并且每个都将按顺序解析。

此时您应该能够启动您的应用程序，并且您将需要登录才能继续。 尝试一下，或尝试使用该项目附带的“教程”示例应用程序。

6.2.3 表单和基本登录选项

你可能想知道当我们登录的时候登陆表单从哪里来的，因为我们并没有写任何的HTML文件或者JSP文件。事实上，因为我们没有明确为登录页面设定一个URL，Spring Security基于登录需要处理的标准数据自动生成一个，当用户登陆后发送到默认的目标URL。然而命名空间提供了许多支持去让你自定义很多选项。如果你想自定义自己的登录页，你可以使用：

```
<http>
    <intercept-url pattern="/login.jsp*" access="IS_AUTHENTICATED_ANONYMOUSLY"/>
    <intercept-url pattern="/**" access="ROLE_USER" />
    <form-login login-page='login.jsp' />
</http>
```

注意我们添加了一个额外的`intercept-url`元素，并且配置允许匿名用户访问登录页。这个AuthenticatedVoter类提供了更多信息关于`IS_AUTHENTICATED_ANONYMOUSLY`是如何处理的。否则对登录页面的请求将会匹配到“/**”，那样将不可能访问到登录页面。这是一个很正常的配置错误，并且会导致应用程序中出现一个死循环。如果你的登录页面是被保护的，那么日志中将出现一个警告。也可以让所有匹配特定模式的请求完全绕过安全过滤器链，靠为此模式定义一个单独的`http`元素：

```
<http pattern="/css/**" security="none"/>
<http pattern="/login.jsp*" security="none"/>
```

```
<http use-expressions="false">
<intercept-url pattern="/**" access="ROLE_USER" />
<form-login login-page='login.jsp'/>
</http>
```

从Spring Security 3.1开始，允许去对于不同的请求模式使用多个`http`元素去定义域分隔的安全过滤器链配置。如果`http`标签中省略`pattern`元素，那么它将匹配所有请求。创建一个不安全的模式就是这种语法的简单例子，其中模式映射到一个空过滤器链。我们将在Security Filter Chain章节中看到更多的新语法的细节。意识到这些不安全的请求将会完全忽略所有Spirng Security中与web相关的配置或者添加的属性，例如`requires-channel`,是很重要的.因此您无法在请求期间访问当前用户的信息或调用受保护的方法。如果你依旧想要安全过滤器链被使用的话，使用`access='IS_AUTHENTICATED_ANONYMOUSLY'`是一个好的选择。

如果你想用基础验证代替表单登录的话，你可以使用如下配置：

```
<http>
    <intercept-url pattern="/**" access="ROLE_USER"/>
    <http-basic />
</http>
```

然后，基本身份验证将优先，并在用户尝试访问受保护资源时用于提示登录。如果您希望使用此配置，表单登录仍然可用，例如通过嵌入其他网页的登录表单。

设置一个默认的Post-Login 目的地

如果尝试访问受保护资源时没有提示表单登录，则`default-target-url`选项会发挥作用。用户在成功登录后将被带到一个URL处，默认是"/"。您还可以配置一些东西，以便通过将always-use-default-target属性设置为"true"，使用户始终停留在此页面（无论登录是"按需"还是明确选择登录）。如果你的应用程序总是要求用户以一个"home"页面开始，那么这是很有用的，例如：

```
<http pattern="/login.htm" security="none" />
<http use-expressions="false">
<intercept-url pattern='/**' access='ROLE_USER' />
<form-login login-page='login.htm' default-target-url='/home.htm' always-use-default-target='true' />
</http>
```

对于更多关于目的地的控制，你可以使用`authentication-success-handler-ref`属性作为对`default-target-url`的另一个选择。被引用的bean是一个`AuthenticationSuccessHandler`的实例。你将会找到更多在Core Filters章节和命名空间附录中，以及如何去自定义当验证失败后的流向。

6.2.4 登出处理

`logout`元素提供了登出后导航到特定URL的支持。默认的登出URL是`/logout`，但是你可以使用`logout-url`属性将它设置为别的东西。更多可使用的信息你可以在命名空间附录中看到。

6.2.5 使用验证提供器

事实上你还需要一个记录用户信息的更可扩展的源文件，而不是将一系列名字添加到应用上下文文件中。你最可能将用户信息存储到数据库或者LDAP服务器中。LDAP命名空间配置会在LDAP 章节进行处理，我们现在不会讲解。如果你有一个Spring Security的`UserDetailsService`的自定义实现，在你的应用上下文中叫"myUserDetailsService"，那么你能使用如下配置：

```
<authentication-manager>
    <authentication-provider user-service-ref='myUserDetailsService'/>
</authentication-manager>
```

如果你想要用一个数据库，那么你可能要用如下配置：

```
<authentication-manager>
<authentication-provider>
    <jdbc-user-service data-source-ref="securityDataSource" />
</authentication-provider>
</authentication-manager>
```

"securityDataSource"指定的是应用上下文中DataSourcebean的名称，这个bean指定了包含标准Spring Security用户数据表的数据库。你也可以配置一个Spring Security JdbcDaoImplbean来指定使用的user-service-ref属性:

```
<authentication-manager>
<authentication-provider user-service-ref='myUserDetailsService' />
</authentication-manager>

<beans:bean id="myUserDetailsService"
class="org.springframework.security.core.userdetails.jdbc.JdbcDaoImpl">
<beans:property name="dataSource" ref="dataSource" />
</beans:bean>
```

你也可以按照如下方式配置标准AuthenticationProviderbean：

```
<authentication-manager>
    <authentication-provider ref='myAuthenticationProvider' />
</authentication-manager>
```

myAuthenticationProvider是你应用上下文中实现了AuthenticationProvider的bean的名字。你可以使用多个authentication-provider元素，这种情况下，验证提供其将会被按照声明顺序挨个查询。关于AuthenticationManager应该如何使用命名空间配置，请查看6.6节 "验证管理器和命名空间";

添加一个密码编码器

密码应该总是使用一个专门为了加密密码的安全哈希算法加密（不是一个标准算法像SHA或者MD5）。<password>元素为添加密码编码器提供了支持。使用bcrypt加密密码，原始的验证提供者应该像如下配置：

```
<beans:bean name="bcryptEncoder"
class="org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder" />

<authentication-manager>
<authentication-provider>
    <password-encoder ref="bcryptEncoder" />
    <user-service>
    <user name="jimi" password="<img src="https://latex.codecogs.com/gif.latex?2a"/>10<img
src="https://latex.codecogs.com/gif.latex?
ddEWZUl8aU0GdZPPpy7wbu82dvEw&#x2F;pBpbRvDQRqA41y6mK1CoH00m&quot;
authorities=&quot;ROLE_USER,%20ROLE_ADMIN&quot;%20&#x2F;&gt;
%20%20%20%20&lt;user%20name=&quot;bob&quot;%20password=&quot;"/>2a<img
src="https://latex.codecogs.com/gif.latex?10"/>/elFpMBnAYYig6KRR5bvOOYeZr1ie1hSogJryg9qDlhza4oCw1Qka"
                authorities="ROLE_USER" />
    </user-service>


</authentiaction-provider>
</authentication-manager>
```

在大多数情况下，bcrypt是一个不错的选择，除非你有一个遗留系统迫使你使用不同的算法。如果您使用简单的哈希算法，或者更糟糕的是，存储纯文本密码，那么您应该考虑迁移到更安全的选项，如bcrypt。

6.3 高级Web特点

6.3.1 Remember-Me 验证

关于更多的remeber-me命名空间的配置请查看Remember-Me 章节。

6.3.2 添加HTTP/HTTPS 通道支持

如果你的应用程序支持HTTP和HTTPS，并且你需要特定的URL能仅仅通过HTTPS协议访问，那么这里可以通过`<intercept-url>`的`requires-channel`属性获得直接支持。

```
<http>
    <intercept-url pattern="/secure/**" access="ROLE_USER" requires-channel="https" />
    <intercept-url pattern="/**" access="ROLE_USER" requires-channel="any" />
</http>
```

使用上面的配置，如果一个用户试图访问匹配"/secure/**"模式的URL时，若使用的是HTTP协议，那么会首先被重定向到一个HTTPS URL。可用的选项是"http"，"https"或"any"。当使用"any"时，意味着HTTP或者HTTPS都能被使用。

如果你的应用程序使用HTTP或HTTPS协议时使用不标准的端口你可以指定特定的端口：

```
<http>
...
<port-mappings>
    <port-mapping http="9080" https="9443"/>
</port-mappings>
</http>
```

请注意，为了确保安全，应用程序不应该使用HTTP或在HTTP和HTTPS之间切换。它应该从HTTPS开始（用户输入HTTPS URL）并始终使用安全连接以避免任何可能的中间人攻击。

6.3.3 Session管理

超时检测

您可以配置Spring Security来检测提交的无效会话ID并将用户重定向到适当的URL。这个功能通过`session-management`元素实现：

```
<http>
...
<session-management invalid-session-url="/invalidSession.htm" />
</http>
```

注意，如果你使用这个机制去预防session超时，那么如果用户在没有关闭浏览器的情况下登出后又登入，那么它可能会错误地引发一个错误。这是因为当你使这个Session无效时session cookie没有被清理，并且及时用户已经登出了，这个cookie还将被重新提交。你可以在登出时明显的删除JSESSIONIF cookie，例如靠下面的配置：

```
<http>
<logout delete-cookies="JSESSIONID" />
</http>
```

不幸的是这种操作并不是在每个servlet容器中都鼓励的，所以你将需要去在你的环境中进行测试。

如果你在一个代理后面运行你的应用程序，那么你可能要去通过配置代理服务器来移除这个session cookie。例如，使用Apache HTTPD的mod_headers，以下指令将通过在对注销请求的响应中过期而删除JSESSIONID cookie。

```
<LocationMatch "/tutorial/logout">
Header always set Set-Cookie "JSESSIONID=;Path=/tutorial;Expires=Thu, 01 Jan 1970 00:00:00 GMT"
</LocationMatch>
```

并发Session控制

如果您希望限制单个用户登录您的应用程序的能力，Spring Security通过以下简单添加支持这种开箱即用的功能。首先你要去添加下面的监听器到你的`web.xml`文件中来保证Spring Security更新Session的生命周期：

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

然后添加下面一行到你的应用上下文中：

```
<http>
...
<session-management>
    <concurrency-control max-sessions="1" />
</session-management>
</http>
```

这将防止用户进行多次登录，第二次登录将会把第一次登录注销。当然，如果你更想要防止二次登陆，你可以使用下面例子：

```
<http>
...
<session-management>
    <concurrency-control max-sessions="1" error-if-maximum-exceeded="true" />
</session-management>
</http>
```

第二次登录将会被拒绝。当被拒绝的时候，如果基于表单的登录被使用，那么用户将被定向到`authentication-failure-url`处。如果二次登录使用的是一个非交互的机制，例如“remember-me”，一个“unauthorized”（401）错误将会被送到客户端。如果你想用一个错误页面代替，你要为`session-management`添加`session-authentication-error-url`属性。

如果你将要为基于表单的登录提供自定义的验证过滤器，那么你不得不显示配置并发session控制支持。更多的细节你会在Session Management 章节看到。

会话固定攻击保护

会话固定攻击是一种潜在风险，因为恶意攻击者可能通过访问站点来创建会话，然后诱使另一个用户使用同一会话登录（例如靠发送给他们一个包含session认证参数的链接）。Spring Security自动保护这个问题，靠当用户登录时创建一个新的session或者改变session ID。 如果你不需要这个保护，或者它和其他的需求冲突，你可以使用`<session-management>`的`session-fixation-protection`属性控制这个行为，这里有4个选项：

- `none` - 不做任何事情，原始的session将被保存。
- `newSession` - 创建一个新的“干净的”session，这个session中没有复制已经存在的session数据（与Spring Security有关的属性将被复制）。
- `migrateSession` - 创建一个新的session，这个session会复制所有已经存在的属性到新的session中。在Servlet 3.0或者更老的容器中是默认的。
- `changeSessionId` - 不创建一个新的session。代替的是，请使用Servlet容器提供的会话固定保护（HttpServletRequest#changeSessionId（））。这个选项仅仅在Servlet 3.1（Java EE 7 ）以及更新的容器中才允许使用。如果在老的容器中使用它将导致一

个异常。在Servlet 3.1或者更新的容器中这是默认的。

当session固定保护触发时，应用上下文中会发布一个SessionFixationProtectEvent。如果你使用的是changeSessionId,这个保护也会导致任何的javax.servlet.http.HttpSessionIdListener被通知，如果你监听所有的事件，请小心谨慎使用。关于更多信息请查看Session Management章节。

6.3.4 OpenID 支持

命名空间除了支持基于表单登录以外还支持OpenID登录。配置会有些变化：

```
<http>
<intercept-url pattern="/**" access="ROLE_USER" />
<openid-login />
</http>
```

你应该然后给你自己注册一个OpenID 提供器，并且添加用户信息到你的内存<user-service>中。

```
<user name="http://jimi.hendrix.myopenid.com/" authorities="ROLE_USER" />
```

你应该能去使用myopenid.com网站去验证登录。在openid-login元素中设置user-service-ref属性可以让你在使用OpenID时选择一个特定的UserDetailsServicebean。对于更多的关于验证提供器的信息，请查看前面的章节。注意我们已经省略了上述用户配置的密码属性，因为这个用户信息仅仅被用来架子啊用户权限。随机密码将在内部生成，从而防止您在配置中的其他位置意外地将此用户数据用作身份验证源。

属性交换

支持OpenID属性交换,作为一个例子，下面的配置试图从OpenID提供器中获取email和全名，以供应用程序使用：

```
<openid-login>
<attribute-exchange>
    <openid-attribute name="email" type="http://axschema.org/contact/email" requited="true" />
    <openid-attribute name="name" type="http://axschema.org/namePerson"/>
</attribute-exchange>
</openid-login>
```

每个OpenID属性的"类型"是由特定模式确定的URI，在这种情况下是http://axschema.org/。如果一个属性对于成功验证一定要背获取，那么这个required属性一定要被设置。精确的模式和属性提供将依赖于你的OpenID提供器。属性值将会被作为验证处理的一部分返回并且能通过下面的代码进行访问：

```
OpenIDAuthenticationToken token =
(OpenIDAuthenticationToken)SecurityContextHolder.getContext().getAuthentication();
List<OpenIDAttribute> attributes = token.getAttributes();
```

OpenIDAttribute包含了属性类型和获取的值（如果是多值属性就是代表值们）。我们将在技术概述章节中看到Spring Security 核心组件，了解更多关于SecurityContextHolder怎么使用。如果您希望使用多个身份提供程序，则还支持多个属性交换配置。你能提供多个attrubute-exchange元素，在每个上面都提供一个identifier-matcher属性。这包含一个正则表达式，它将与用户提供的OpenID标识符相匹配。有关示例配置，请参阅代码库中的OpenID示例应用程序，为Google，Yahoo和MyOpenID提供程序提供不同的属性列表。

6.3.5 响应头

对于如何自定义响应头元素的更多信息请查看第21章 Security HTTP响应头。

6.3.6 添加你自己的过滤器

如果你在之前已经用过Spring Security，那么你将会知道为了应用其服务，该框架维护了一个过滤器链。你可能想要添加你自己的过滤器到这个栈中的特定位置或者使用目前没有名称空间配置选项的Spring Security过滤器（CAS，例如）。或者你可能想要去用一个自定义版本的标准命名空间过滤器，例如UsernamePasswordAuthenticationFilter,它被<form-login>元素创建，利用一些额外配置选项，这些选项通过显式使用bean可用。你如何通过命名空间配置做到这些呢，因为过滤器链并没有直接暴露。

当使用命名空间时，过滤器的顺序总是被严格制定。当应用上下文被创建时，过滤器bean已经被命名空间处理代码排好序了，并且标准Spring Security 过滤器在命名空间中每个都有一个别名和一个好的位置。

在以前的版本里，排序操作发生在过滤器实例建立之后，在应用上下文的后处理期间。在版本3.0+中，在bean实例化之前，在bean元数据级别完成排序。他对如何将自己的过滤器添加到堆栈中有影响，因为在解析<http>元素时必须知道整个过滤器列表，因此语法在3.0中略有变化。

过滤器、别名和创建过滤器的命名空间元素/属性都被展示在下表中，过滤器按照他们在过滤器链中的顺序进行排列：

Alias	Filter Class	Namespace Element or Attribute
CHANNEL_FILTER	ChannelProcessingFilter	http/intercept-url@requires-channel
SECURITY_CONTEXT_FILTER	SecurityContextPersistenceFilter	http
CONCURRENT_SESSION_FILTER	ConcurrentSessionFilter	session-management/concurrency-control
HEADERS_FILTER	HeadWriterFilter	http/headers
CSRF_FILTER	CsrfFilter	http/csrf
LOGOUT_FILTER	LogoutFilter	http/logout
X509_FILTER	X509AuthenticationFilter	http/x509
PRE_AUTH_FILTER	AbstractPreAuthenticatedProcessingFilter子类	N/A
CAS_FILTER	CasAuthenticationFilter	N/A
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	session-management
EXCEPTION_TRANSLATION_FILTER	ExceptionTranslationFIlter	http
FILTER_SECURITY_INTERCEPTOR	FilterSecurityInterceptor	http
SWITCH_USER_FILTER	SwitchUserFilter	N/A

你可以使用custom-filter元素和这些名字中的一个来将你自己的过滤器添加到指定位置，例如下面配置：

```
<http>
<custom-filter position="FORM_LOGIN_FILTER" ref="myFilter" />
</http>

<beans:bean id="myFilter" class="com.mycompany.MySpecialAuthenticationFilter"/>
```

如果你想让你的过滤器插入到某个过滤器之前或者之后，可以使用after或者before属性。position可以使用FIRST和LAST去指定这个过

滤器出现在整个过滤器链最前面或者最后面。

如果您插入的自定义过滤器可能与名称空间创建的标准过滤器之一占据相同的位置，请务必不要错误地包含命名空间版本。删除所有创建要替换其功能的过滤器的元素。 注意你不能替换使用`<http>`元素创建的过滤器，如`SecurityContextPersistenceFilter`，`ExceptionTranslationFilter`或者`FilterSecurityIntercept`。一些其他的过滤器被默认添加，但是你可以不使用它们。默认情况下添加AnonymousAuthenticationFilter，除非禁用了会话固定保护，否则会将SessionManagementFilter添加到过滤器链中。如果您正在替换需要身份验证入口点的命名空间过滤器（身份验证过程由未经身份验证的用户尝试访问安全资源所触发），则您还需要添加一个自定义入口点bean。

设置一个自定义验证入口点

如果你没有通过命名空间配置使用表单登录，OpenID登录或者基础验证，你可能需要使用传统的bean语法定义一个验证过滤器和一个入口点，并将他们链接到命名空间中，就像我们刚才看到的。相应的`AuthenticationEntryPoint`可以使用`<http>`元素上的`entry-point-ref`属性进行设置。

CAS简单应用是一个很好的使用命名空间自定义bean的例子，包括这些语法。如果你不熟悉验证入口点，它们将在先面对技术概论章节进行讨论。

6.4 方法安全

从版本2.0之后，Spring Security已提供在你的服务层方法上添加安全控制的支持。它提供了对于JSR-250注解和框架原始的`@Secured`注解的支持。从3.0版本开始你也可以使用新的基于语法的注解.您可以将安全性应用于单个bean，使用intercept-methods元素来装饰bean声明，或者可以使用AspectJ样式切入点在整个服务层中保护多个bean。

6.4.1 元素

此元素用于在应用程序中启用基于注释的安全性（通过在元素上设置适当的属性），并将安全性切入点声明组合在一起，这些声明将应用于整个应用程序上下文中。你应该仅仅声明一个`<global-method-security>`元素。下面的声明将会启动Spring Security的`@Secured`注解：

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

对一个方法（在一个类或者接口上）添加注解，访问时将会根据注解内容限制访问。Spring Security的本地注解支持定义了一系列方法属性。这些将会传给`AccessDecisionManager`用来判断访问条件。

public interface BankService { @Secured("IS_AUTHENTICATED_ANONYMOUSLY") public Account readAccount(Long id);

```
@Secured("IS_AUTHENTICATED_ANONYMOUSLY")
public Account[] findAccounts();

@Secured("ROLE_TELLER")
public Account post(Account account,double amount);
```

}

启动JSR-250注解使用下面的配置：

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

这些都是基于标准的，允许应用简单的基于角色的约束，但是没有Spring Security的本地注释的强大功能。使用新语法按照如下配置

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

上述Java代码换用新语法如下：

```
public interface BankService {

@PreAuthorize("isAnonymous()")
public Account readAccount(Long id);

@PreAuthorize("isAnonymous()")
public Account[] findAccounts();

@PreAuthorize("hasAuthority('ROLE_TELLER')")
public Account post(Account account, double amount);
}
```

注释的方法只能用于定义为Spring bean的实例（在同一启动了方法安全的上下文中）。如果你想保护的实例并不是靠Spring创建的（例如，用new符号）,这时你就要使用AspectJ。

你可以在同一个应用中启动多种注解，但是应该仅有对于所有的类型或者类应该只有一种形式被使用，否则这些行为不能被良好的表现。如果一个方法上存在两个注解，那么只有一个会被适用。

使用保护切点添加安全切点

protect-pointcut是十分强力的，它允许你对多个bean只使用一个声明去配置安全设置。例如：

```
<global-method-security>
<protect-pointcut expression="execution(* com.mycompany.*Service.*(...))" access="ROLE_USER" />
</global-method-security>
```

上述配置将会保护被声明在应用程序上下文中的在com.mycompany包中的所有名字以"Service"结尾的类。仅仅是ROLE_USER角色的用户才能调用这些方法。和URL匹配一样，最特别的匹配要放在切点列表的第一位，首先匹配的表达式将会被使用。安全注释优先于切入点。

6.5 默认AccessDecisionManager

这节假设你已经对Spring Security的访问控制的底层结构有一定了解。如果你没有这些知识，可以先跳过这一节，然后再回来看。因为这部分只对那些需要进行一些定制以便使用不仅仅是简单的基于角色的安全性的人真正相关。

当你使用一个命名空间配置时，一个默认的AccessDecisionManager实例将会自动创建和注册，当方法调用和web URL访问时，这个家伙被用来做访问判断，而这个判断是基于在你的intercept-url和protect-pointcut生命中指定的参数（或者如果你用的基于注解的安全方法，那么就是基于你的注解）。

默认策略是使用具有RoleVoter和AuthenticatedVoter的AffirmativeBased AccessDecisionManager。

6.5.1 自定义AccessDecisionManager

如果你想要使用一个更复杂的访问控制策略，则很容易为方法和网络安全设置替代方案。

对于方法安全，你要做的事是设置global-method-security的access-decision-manager-ref为应用上下文中适当的AccessDecisionManagerbean的id。

```
<global-method-security access-decision-manager-ref="myAccessDecisionManagerBean">
...
</global-method-security>
```

网络安全也一样，不过是要在http元素中：

```
<http access-decision-manager-ref="myAccessDecisionManagerBean">
...
</http>
```

6.6 验证管理器和命名空间

在Spring Security中提供验证服务的最主要接口就是`AuthenticationManager`。通常是一个Spring Security的`ProviderManager`类实例，如果你以前使用过这个框架你应该很熟悉它。如果没有，那么我们会在下面的技术概论章节中介绍。这个bean实例使用`authentication-manager`命名空间元素注册。如果你通过命名空间使用HTTP或者方法安全，你没有办法去使用自定义`AuthenticationManager`，但是这应该不是啥问题，因为你对使用的`AuthenticationProvider`具有完全的控制。

你可能想在`ProviderManager`上注册另外的`AuthenticationProvider`bean，你可以使用`<authentication-provider>`元素的`ref`属性去做到这个，这个属性的值就是你想要添加的provider的bean名称，例如：

```
<authentication-manager>
<authentication-provider ref="casAuthenticationProvider"/>
</authentication-manager>

<bean id="casAuthenticationProvider"
class="org.springframework.security.cas.authentication.CasAuthenticationProvider">
...
</bean>
```

另一个常见的需求时上下文中的另一个bean可能需要一个`AuthenticationManager`引用。你可以轻易的为`AuthenticationManager`注册一个别名并且在应用上下文中到处使用它。

```
<security:authentication-manager alias="authenticationManager">
...
</security:authentication-manager>

<bean id="custonizedFormLoginFilter" class="com.somecompany.security.web.CustomFormLoginFilter">
<properity name="authenticationManager" ref="authenticationManager"/>
...
</bean>
```