Spring Security Reference

作者
Ben Alex，Luke Taylor，Rob Winch，Gunnar Hillert

4.0.3.RELEASE

4.Security命名空间配置

4.1 简介

命名空间的配置方法是Spring2.0之后的新特性。此特性允许你用xml schema中定义的元素节点来定义传统的Spring bean。更多信息参见Spring官方文档。命名空间的配置一方面可以简化传统的对于bean的定义，更进一步，提供了一种更好的匹配你的应用具体情况的配置方式，同时屏蔽了大量的底层细节配置。即一个简单的元素可能屏蔽掉了大量的底层bean的配置及应用上下文中的具体处理步骤。例如：在应用上下文的security命名空间中添加如下的元素将在应用中启动一个内嵌的LDAP服务以作为测试用：

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

这相对于配置等价的Apache Directory Server beans可以说是简单多了。ladp-server标签的属性能很好的支持这个场景的常见的配置需求，用户不需要考虑他们要创建什么bean以及每个bean的需要设定的属性名等。如果使用一个好的XML编辑器，那在你编辑应用上下文文件的时候还能提示有哪些元素或者属性可选。我们推荐你使用SpringSource Tool Suite，在编辑标准的Spring命名空间的时候，它有很好的特效。

想要开始编辑Security的命名空间，你首先需要在你的classpath下面添加spring-security-config jar包。然后你需要在你的应用上下文文件中添加schema的声明：

```
<beansxmlns="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>
```

在今后的许多例子中你会看到，我们经常使用“security”作为默认的命名空间而不是“beans”，这意味着我们可以忽略所有的security命名空间前缀不写，这样便于阅读。但是你也可以这样做，尤其是当你的配置文件分割在不同的文件中，但是其中某些文件又含有security的配置的时候。默认“security”时，你的security应用上下文文件应该这样开始：

```
<beans:beansxmlns="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>
```

下文我们假设使用的是这种配置。

4.1.1 命名空间的设计

命名空间的设计迎合了整个框架的最常见的使用方式，并且提供了简化的和简洁的语法来在应用中使用。这种设计基于整个框架大规模的依赖体系，并且可以分为以下几个部分：

1、Web/HTTP Security，最复杂的部分。建立过滤器及其依赖的beans来应用框架的认证机制，来保护URL，递交登陆，错误页等等。
2、Business Object（Method）Security，对于服务层可选的保护措施。
3、AuthenticationManager，处理来自框架其它部分的认证请求。
4、AccessDecisionManager，提供针对web和方法的访问决策安全。默认的实现会被注册，但是你也可以通过通用的Spring bean定义来定制一个个性化的。
5、AuthenticationProviders，针对认证管理器认证用户的机制。命名空间提供多个标准选项的支持，同样也提供使用Spring bean的标准语法来制定个性化实例的方案。
6、UserDetailsService，与认证提供器关系很近，但是通常也为其他beans提供服务。

我们将在下一章节研究他们如何配置。

4.2 开始Security命名空间配置

在这一节，我们将会看看如何建立Security命名空间的配置以应用整个框架的主要特性。我们假设你首先期望尽快的构建整个框架，并对已有的web工程建立认证支持及访问控制。然后我们看看如何调整，来使得认证能够基于数据库或者其他的安全库。最后一节我们介绍更多的命名空间配置项。

4.2.1 web.xml配置

首先你需要在的web.xml文件中声明如下过滤器：

```
<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>
```

这提供了一个连接到Spring Security web设施的钩子。DelegatingFilterProxy是一个Spring框架的类，它会将自己代理到一个应用上下文中定义的过滤器bean。

在这个场景中，名为“springSecurityFilterChain”的bean被它代理了，这个bean是一个命名空间配置创建的内部结构（类），能用来处理web系统的安全。注意你不能自己乱用这个bean名。一旦你将这个配置添加到web.xml中，相当于你已经开始编辑你的安全上下文配置了。Web安全服务配置使用<http>标签。

4.2.2 最小的<http>配置

开始web security的配置，只需要如下：

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

这代表我们期望应用中所有的url都被安全过滤，需要ROLE_USER角色访问，需要用户使用带有用户名和密码的表单登入系统，并且我们要求注册一个登出页面（url）来允许我们登出系统。<http>元素是所有web相关的安全元素（功能）的父元素。<intercept-url>元素定义了一种ant风格的url模式，将对进入的请求url进行匹配。你也可以取而代之的使用正则匹配。access属性定义了针对匹配上的url的访问需求。在默认配置下，这典型的是一个逗号分隔的角色列表，用户只有符合了列表中的角色才能发起这个请求。前缀“ROLE_”是一个标记，指示着需要与用户的权限做一个简单的比对。换句话说，需要进行一个正常的基于角色的检验。Spring Security中的访问控制并不局限于使用简单的角色（因此前缀用来区分security属性的不同类型）。

你可以使用多个<intercept-url>元素来针对不同的url集合定义不同的访问需求，但是他们是被有序的匹配，即第一个匹配的项目会被应用。所以你必须将最特殊的放在前面，你也可以在其中添加method属性来将匹配限制到具体的HTTP方法。

加入几个用户，你可以直接在命名空间中定义几个测试数据：

```
<authentication-manager><authentication-provider><user-service><username="jimi"password="jimispassword"au
thorities="ROLE_USER, ROLE_ADMIN" /><username="bob"password="bobspassword"authorities="ROLE_USER" /></use
r-service></authentication-provider></authentication-manager>
```

如果你对之前版本的命名空间配置比较熟悉，在这里你已经可以大概猜到这里发生了什么。<http>元素的责任为创建一个FilterChainProxy（Spring Security的类）及其使用的过滤器bean。这样，一些通常的问题，比如你自行把过滤器顺序放错了这样的问题就不需要考虑了，因为这些过滤器已经定义好了放好了。<authentication-provider>元素创建了一个DaoAuthenticationProvider的bean，<user-service>元素创建了一个InMemoryDaoImpl。所有的authentication-provider元素必须是authentication-manager元素的子元素，它创建了ProviderManager并且将所有其子元素的认证提供器注册进去。更多的beans创建类型可以参考附录。这块是值得仔细确认的，尤其是以后如果你想要理解框架中的类的作用及定制自己的类的时候。

上面的配置定义了系统内的两个用户（将被用来做访问控制），他们的密码及角色，也可以用user-service元素的property属性从标准的java属性文件中读取用户的信息。参加in-memory authentication章节来了解其配置文件的具体格式。使用authentication-provider元素意味着认证管理器将使用用户信息来处理认证请求。可以定义多个authentication-provider元素来定义不同的认证数据源，然后每个数据源都会被轮流询问。
在这个时候，你已经能开始你的应用了，并且你会被要求登录。

4.2.3 表单及基础登录选项

你现在可能开始琢磨登录表单在哪了，我们如何被提示需要登录呢，因为至今为止我们还没有提到任何的html文件或者jsp文件。事实上，因为我们还没显式的指定登录页面的url，Spring Security基于你打开的security功能，自动为你生成了一个使用默认值来处理登录的页面，一个在你登录后默认跳转到的页面，等等。然而，命名空间的配置方法提供了你大量的能定制这些选项的方式。例如：如果你想要支持你自己的登录页面，你可以使用：

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

同样注意我们添加了一个额外的intercept-url元素来声明任意的针对login页面的请求都对匿名用户有效，AuthenticatedVoter类含有更多的IS_AUTHENTICATED_ANONYMOUSLY的处理方法。否则请求将会被/**模式匹配，并且不能够主动访问登录页面！这是一个常见的配置错误，将会导致应用中的无限循环。Spring Security会在日志中放出一个警告，如果你的登录页面看上去也被加上权限了的话。将所有请求都通过匹配一个特殊的模式来完全略过security过滤器链也是可能的，通过这样定义：

```
<httppattern="/css/**"security="none"/><httppattern="/login.jsp*"security="none"/><httpuse-expressions="f
alse"><intercept-urlpattern="/**"access="ROLE_USER" /><form-loginlogin-page='/login.jsp'/></http>
```

从Spring Security3.1之后，可以使用多个http标签来针对不同的请求url模式定义单独的security过滤器链配置。如果pattern属性在一个http元素中不存在，则它匹配所有url请求模式。创建不需要保护的url模式是这种语法的简单例子，也就是实际上这个url模式是映射到空的过滤器链的。我们会在Security Filter Chain章节看见更多细节。
一件你需要注意的很重要的事情是：这些没有被保护的请求将会完全的被任何Spring Security Web安全相关的配置及附带的一些特性（如requires-channel）忽略掉。所以你将不能访问当前用户的信息或者调用请求的安全相关方法。使用access=“IS_AUTHENTICATED_ANONYMOUSLY”是一个替代，如果你扔需要安全过滤器链被应用的话。

如果你想要使用基础的认证代替表单登录，那么改变配置为：

```
<httpuse-expressions="false"><intercept-urlpattern="/**"access="ROLE_USER" /><http-basic /></http>
```

基础认证将会优先生效，并在用户尝试访问被保护的资源的时候用来提示登录。表单登录在这种配置下仍然有效，如果你想使用它的话。例如通过一个嵌入在另外网页中的登录表单。

设置默认的登陆后目的

如果表单登录没有提示尝试访问一些被保护的资源，default-target-url选项则登上舞台。这个url是用户成功登录以后被跳转到的目的地，默认是“/”。你也可以配置一些东西使得用户的登录结果永远结束在这个页面上（不论用户被要求登录还是自己选择登录），通过设置always-use-default-target属性为true。这个在你的应用期望用户永远从一个“主页”开始的时候特别有效，例如：

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

对于在登录后目的的进一步控制，你可以使用authentication-success-handler-ref属性作为default-target-url的替代。被引用到的bean应该是一个AuthenticationSuccessHandler的实例。你会在Core Filter章节找到它的介绍，及配置方法，也有当认证失败的时候的配置方式。

4.2.4 登出处理器

logout元素添加对登出动作的支持，通过导航到一个特定的url。默认的登出url是/logout。但是你可以使用logout-url属性设置。更多细节参加附录。

4.2.5 使用认证提供器

实践中，你或许需要一个更有扩展性的用户信息数据源，而不是几个添加在应用上下文文件中的名字。比如最有可能的是你想要在一些数据库中或者LDAP服务器中存储你的用户信息。LDAP命名空间配置在相关章节处理，所以我们这边不会涉及。如果你有Spring Security的UserDetailsService的个性化实现，例如一个在你的上下文中定义的叫做myUserDetailsService的bean，你可以这样配置：

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

如果你想用数据库，你这样用：

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

这里securityDataSource是一个DataSource bean的名称，指向一个包含标准Spring Security用户信息表的数据库。相对的，你能够配置一个Spring Security JdbcDaoImpl bean，并使用user-service-ref来指定。

```
<authentication-manager><authentication-provideruser-service-ref='myUserDetailsService'/></authentication
-manager><beans:beanid="myUserDetailsService"class="org.springframework.security.core.userdetails.jdbc.Jd
bcDaoImpl"><beans:propertyname="dataSource"ref="dataSource"/></beans:bean>
```

你也可以使用标准的AuthenticationProvider：

```
<authentication-manager><authentication-providerref='myAuthenticationProvider'/></authentication-manager>
```

这里myAuthenticationProvider是你应用上下文中的bean的名称，它实现了AuthenticationProvider接口。你尅使用多个authentication-provider元素，这种情况下每个认证提供器都会按照他们声明的顺序被询问。参见4.6节介绍的AuthenticationManager的配置介绍。

添加一个密码编码器

密码应该总是使用安全哈希算法来编码（不是标准的算法，例如SHA或者MD5），这由<password-encoder>标签支持。使用bcrypt编码的密码，原始的认证提供器配置应该如下：

```
<beans:beanname="bcryptEncoder"class="org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder"/>
<authentication-manager><authentication-provider><password-encoderref="bcryptEncoder"/><user-service><use
rname="jimi"password="d7e6351eaa13189a5a3641bab846c8e8c69ba39f"authorities="ROLE_USER, ROLE_ADMIN" /><use
rname="bob"password="4e7421b1b8765d8f9406d87e7cc6aa784c4ab97f"authorities="ROLE_USER" /></user-service></
authentication-provider></authentication-manager>
```

Bcrypt在大部分情况下是一个好选择，除非你在开发某些私密的系统要求你使用特定的算法。如果你使用简单的哈希算法或者，更糟糕的，存储密码原文，你应该考虑添加一个更安全的选项。

4.3 高级Web特性

4.3.1 记住我认证

请单独参加Remember-Me章节。

4.3.2 添加HTTP/HTTPS隧道安全

如果你的应用同时支持HTTP和HTTPS，并且你需要指定的url只能通过https访问，那么intercept-url元素中的requires-channel属性直接支持：

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

有这个配置的情况下，如果一个用户尝试使用HTTP访问任何匹配“/secure/**”模式的url，他们会首先被重定向到一个HTTPS的url上。可行的选项是“http”，“https”，“any”。使用“any”意味着http和https均可用。

如果你的应用使用非标准的HTTP或者HTTPS端口，你可以指定一个端口列表：

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

注意为了真正达到安全效果，应用不应该使用HTTP，也不应该在HTTP和HTTPS之间切换，而是应该从HTTPS开始（用户输入的url就是HTTPS的），并从始至终的使用安全连接，来避免任何的人为的从中发起的攻击。

4.3.3 Session管理

超时探测
你可以配置Spring Security来探测非法session id，并将用户跳转到一个合适的url。这通过session-management元素来实现：

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

注意如果你使用这个机制来探测session超时，在用户登出并且没有关闭浏览器然后重新登入的情况下，它或许会失误的报告一个错误。这是因为你在服务端使当前session失效时，cookie中还没有清除这个session，即使用户登出了，这个session也会被重新提交。你或许能够显式的在登出时删除JSESSIONID的cookie，例如通过使用如下的语法：

```
<http><logoutdelete-cookies="JSESSIONID" /></http>
```

不幸的是这个不能保证在每种servlet容器中都能正常工作，所以你需要在你的环境中测试。

如果你在一个代理服务器后面跑你的应用，你也可以通过配置代理服务器来达到删除有session的cookie这个效果。例如，使用Apache HTTPD的mod_headers，下面的指令会删除JSESSIONID的cookie，通过在登出请求对应的响应中将它设为超时（假设应用部署在/tutorial路径下）：

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

并发Session控制
如果你期望对于一个单独的用户向你应用登录的次数进行限制，Spring Security用如下的简单补充作为支撑。首先你需要添加如下的监听器到你的web.xml中，来保证Spring Security在session生命周期事件中都会更新：

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

然后添加如下的内容到你的上下文中：

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

这将会防止用户登入多次——第二次登录会导致第一次登录失效。通常你期望阻止第二次登录，这样你可以使用：

```
<http>
...
<session-management><concurrency-controlmax-sessions="1"error-if-maximum-exceeded="true" /></session-mana
gement></http>
```

这样第二次登录会被拒绝，这里“拒绝”，我们指用户会被转到authentication-failure-url，如果使用的是基于表单的登录的话。如果第二次登录通过一种非交互机制产生，例如：记住我，则一个未授权（401）错误会被发送到客户端。如果你想使用一个错误页，你可以再session-management元素中添加session-authentication-error-url属性。
如果你使用个性化的认证过滤器来做表单登录，那你必须显式配置并发session控制器。更多细节参加Session Management章节。

Session固定攻击防护
Session固定攻击是一种潜在的危险，这里一个恶意的攻击者通过访问一个站点来创建一个session，然后说服另外一个用户使用相同的session来登录（例如，通过发送给他们一个包含session id参数的链接），这样，攻击者就可以得到其他用户的身份。Spring Security自动保护这一情况，通过在用户登录的时候创建一个新session或者改变session id。如果你不需要这个保护，或者与你其他某些需求冲突，你可以通过<session-management>节点中的session-fixation-protection属性来控制这一行为，这有如下四种选项：
none：什么也不做，原始session保留
newSession：创建一个新的干净的session，而不去复制现存的session数据（spring security相关的属性会复制）
migrateSession：创建一个新session并且复制所有现存的session属性到新session中，这是Servlet3.0或者老式容器默认的。
changeSessionId：不要创建新的session，取而代之的，使用Servlet容器提供的session固定攻击防护。这个选项仅在Servlet3.1（Java EE 7）和更新的容器中支持，在老式容器中指定它会导致一个异常，在Servlet3.1和新容器中是默认的。

当session固定攻击发生了，它将会导致一个SessionFixationProtectionEvent发布到应用上下文中。如果你使用了changeSessionId，这个保护也会导致HttpSessionIdListener被通知，所以需要注意一下如果你的代码对这两个事件都进行处理的情况。具体参见Session Management章节。

4.3.4 OpenID支持
命名空间配置法支持OpenID登录，可以用来作为基于表单登录的补充或者直接代替之，只要稍微改一下配置文件：

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

你应该使用一个OpenID提供者注册来注册你自己，并且将用户信息添加到你的<user-service>中：

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

你应该可以使用myopenid.com站点来登录以认证。也可以指定一个UserDetailsService以使用OpenID，通过设定openid-login元素中的user-service-ref属性。参加之前的描述认证提供器的章节。注意我们在上面的用户配置中漏掉了密码项，因为这种针对用户数据的设置仅仅用来读取用户的权限。内部将随机生成一个密码，防止你偶然的在你的配置中使用这个用户数据作为其他配置项的认证数据源。

属性交换

支持OpenId的属性交换。例如：下面的配置会尝试从OpenID提供者获取email和全称，以便应用使用：

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

OpenID的属性的类型是一个uri，由一个特定的schema定义，在这里是http://axschema.org。如果一个属性在认证成功的时候必须被返回，则可以设定required属性。确切支持的schema和属性取决于你的OpenID提供商。属性值会作为认证处理的一部分返回，你可以在以后的代码中使用：

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

OpenIDAttribute包含了属性类型和返回值（或者多个值）。在Spring Security组件章节我们会看到对SecurityContextHolder类工作机制的介绍。多个属性交换的配置也支持，如果你想要使用多个id供应商的话。你可以写多个attribute-exchange元素，每个都使用identifier-matcher属性。这包含一个针对OpenID身份的正则匹配的表达式。

4.3.5 响应头

具体如何定制响应头元素，请参见17章。

4.3.6 添加你自己的过滤器

如果你之前用过Spring Security，你会知道整个框架提供服务的机制是内部维护了一个过滤器链。你或许期望在具体某个位置添加一个你自己定义的过滤器，或者使用一个Spring Security自带的，但是在当前命名空间默认配置中不包含的过滤器。或者你期望使用一个标准命名空间过滤器的个性化版本，例如UsernamePasswordAuthenticationFilter，这个过滤器被<form-login>元素创建，你期望利用一些这个bean提供的可以个性化定制的选项，来达到个性化配置的目的。这在命名空间的配置方式中怎么做呢，尤其是过滤器链并没有直接暴露出来？
当使用命名空间配置的时候，过滤器的顺序永远是严格限定的。当应用上下文创建时，过滤器bean被命名空间中相关处理代码排序，并且标准的Spring Security在命名空间中都具有一个别名和一个广泛知晓的位置。

在先前的版本，排序工作在过滤器实例创建之后进行，是应用上下文的post-processing引发。在3.0以上版本，排序在bean的元数据层就已经完成，也就是说在类实例化之前。这对于你是如何将你的过滤器添加进去是有影响的，因为整个过滤器列表必须在解析<http>元素的时候被得知，所以3.0的语法有了轻微的变化。

过滤器，别名和命名空间中创建该过滤器的元素/属性在表4.1中展示，过滤器被列出的顺序就是他们在过滤器链中出现的顺序：

表4.1 标准过滤器别名及排序

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	`HeaderWriterFilter`	`http/headers`
CSRF_FILTER	`CsrfFilter`	`http/csrf`
LOGOUT_FILTER	`LogoutFilter`	`http/logout`
X509_FILTER	`X509AuthenticationFilter`	`http/x509`
PRE_AUTH_FILTER	`AbstractPreAuthenticatedProcessingFilter` Subclasses	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`

Alias	Filter Class	Namespace Element or Attribute
FILTER_SECURITY_INTERCEPTOR	`FilterSecurityInterceptor`	`http`
SWITCH_USER_FILTER	`SwitchUserFilter`	N/A

你可以使用custom-filter元素添加你自己的过滤器，使用上面的过滤器名来指定你的过滤器出现的位置：

```
<http><custom-filterposition="FORM_LOGIN_FILTER"ref="myFilter" /></http><beans:beanid="myFilter"class="com.mycompany.MySpecialAuthenticationFilter"/>
```

也可以使用after和before属性，如果你期望你自己的过滤器是插在具体已存在的两个过滤器之间的时候。“FIRST”和“LAST”可以和position属性同时使用，来指示你的过滤器是存在于整个过滤器链的最前面或者说最后面。

如果你插入一个个性化的过滤器，这个过滤器会占据命名空间创建的某个标准过滤器的相同位置的话，重要的事情是你不要错误的引入命名空间定义的那个标准版本过滤器。也就是说，要移除所有的你期望代替的过滤器的配置元素。
注意你不能替代使用<http>元素自身创建的过滤器——SecurityContextPersistenceFilter，ExceptionTranslationFilter，FilterSecurityInterceptor。一些其他的过滤器是默认添加的，但是你不能关闭它们。AnonymousAuthenticationFilter就是默认添加的，然后除非你关闭session固定攻击防护，否则SessionManagementFilter也是默认添加的。

如果你替掉了一个需要认证入口点的命名空间过滤器（例如，一个未认证的用户来访问一个被保护的资源的时候，从哪里触发认证的过程），你会同时也需要一个认证入口点bean。

设置个性化认证入口点
如果你不使用表单登录、OpenID或者命名空间提供的基础认证，你或许期望通过使用传统的bean语法，定义一个认证过滤器和一个认证入口点，然后将他们链接到命名空间中。相应的AuthenticationEntryPoint可以在<http>元素中使用entry-point-ref来设置。
CAS示例工程是一个使用自定义bean结合命名空间的好例子，包括这种语法的使用。如果你对认证入口点不熟悉，在技术总览章节有介绍。

4.4 方法安全

从2.0版本之后，Spring Security有相当大的改进来添加你服务层方法的安全性。它支持JSR-250支持的安全标签以及框架自身的@Secured标签。从3.0以后，你也可以使用基于表达式的标签。你可以应用Spring Security到一个单独的bean，使用intercept-method来装饰bean的声明，或者你可以使用AspectJ风格的切面来保护整个服务层的多个bean。

4.4.1 <global-method-security>元素

这个元素用来打开你应用中基于注解的安全性（通过设置该元素的适当属性），也用来把你的整个应用上下文中定义的安全切面声明整合在一起。你应该仅仅声明一个<global-method-security>元素。下面的声明会打开Spring Security的@Secured标签：

```
<global-method-securitysecured-annotations="enabled" />
```

向一个方法或者一个类或者接口添加标签，将相应的现在对那些方法的访问。Spring Security的原生注解支持定义了针对方法的一系列属性，这些会被传递到AccessDecisionManager中，来做出实际的决策：

```
publicinterface 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);
}
```

对JSR-250注解的支持可以使用如下打开：

```
<global-method-securityjsr250-annotations="enabled" />
```

这是基于标准的注解，并且允许使用简单的基于角色的限制，但是没有Spring Security原生注解的能力。使用新的基于表达式的语法，你应该：

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

相对的java代码或许为：

```
publicinterface BankService {

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

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

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

基于表达式的注解是一个很好的选择，如果你需要定义在检查角色名称对应的角色权限之外的安全规则的话。

添加注解的方法仅仅当这个实例被spring容器管理的时候才会有安全验证。（存在于method-security打开的应用上下文中）如果你想要保护不是被Spring创建的实例，那你需要使用AspectJ。
你可以在相同的应用上下文中打开更多的注解类型，但是在任意接口或者类上只应该使用一种类型，因为不这样做的话其行为或许定义的不完美。如果两个注解应用到同一个方法上，那么只有一个会生效。

使用protect-pointcut添加安全切面
protect-pointcut是相当强力的，因为它允许你仅使用一个简单的声明，就能在许多bean上应用安全性。考虑下面的例子：

```
<global-method-security><protect-pointcutexpression="execution(* com.mycompany.*Service.*(..))"access="RO
LE_USER"/></global-method-security>
```

这将会保护应用上下文中所有的bean定义，只要其类存在于com.mycompany包中，并且其类名以“Service”结尾。仅有具备ROLE_USER的角色有能力调用这些方法。像url匹配一样，最特殊的匹配规则必须放在切面的首位，作为第一个匹配的规则。安全注解将会在切面中具有优先地位。

4.5 默认的AccessDecisionManager

这一章假设你具有一些Spring Security的访问控制架构的先验知识。否则你可以先跳过本章回头再看，因为这一章仅对需要使用比简单的基于角色的权限更复杂的权限配置的用户有必要。
当你使用命名空间配置，一个默认的AccessDecisionManager自动为你注册，然后基于你指定在intercept-url和protect-pointcut中的声明（及注解中的声明），对方法调用及web url的访问做访问决策。
默认策略是使用带有RoleVoter和AuthenticatedVoter的AffirmativeBased AccessDecisionManager。可以在认证章节找到更多的信息。

4.5.1 定制AccessDecisionManager

如果你需要使用更复杂的访问控制策略，对于方法和web安全都可以简单的修改。
对于方法安全，通过设置global-method-security上的access-decision-manager-ref，指向应用上下文中的合适的AccessDecisionManager bean的id即可：

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

对于web安全类似，只是在http元素上：

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

4.6 认证管理器及其命名空间

在Spring Security中提供认证服务的主接口是AuthenticationManager。这通常被一个Spring Security的ProviderManager类的实例实现，如果之前你用过这个框架，会比较熟悉。后面在技术概览章节还会介绍。这个bean的实例在命名空间中使用authentication-manager注册的。不论你是通过命名空间使用http还是方法安全，都不可以使用个性化的AuthenticationManager（读者注：怪不得通常这里会有别名，下面有介绍），但是这不是问题，因为你对于其使用的AuthenticationProvider具有完全控制权。
你或许期望向ProviderManager注册额外的AuthenticationProvider bean，你可以通过使用<authentication-provider>元素的ref属性来实现，其中属性的值是你期望添加的bean的名称，例如：

```
<authentication-manager><authentication-providerref="casAuthenticationProvider"/></authentication-manager
><beanid="casAuthenticationProvider"class="org.springframework.security.cas.authentication.CasAuthenticat
ionProvider">
...
</bean>
```

另外的常见需求是，上下文中的其它bean或许需要引用到AuthenticationManager。可以通过为AuthenticationManager注册一个别名来使用。

```
<security:authentication-manageralias="authenticationManager">
...
</security:authentication-manager><beanid="customizedFormLoginFilter"class="com.somecompany.security.web.
CustomFormLoginFilter"><propertyname="authenticationManager"ref="authenticationManager"/>
...
</bean>
```

7.技术概览

7.1 运行时环境

Spring Security3需要java5或更高的环境来运行。因为Spring Security的目标是实现自包含，也就是不需要向你的java运行环境中放入任何特殊的配置文件。特别是不需要配置一个特殊的Java Authentication and Authorization Service（JAAS）策略文件或者必须将Spring Security放在通常的classpath路径下。
相似的，如果你使用EJB容器或者Servlet容器，不需要在任何地方放置任何特殊的配置文件，或者在服务器的类加载器中包含Spring Security。所有的必须的文件都在你的应用本身包含了。
这种设计最大的特供了部署时的灵活性，因为你可以简单的复制你的目标包（jar，war，ear）在系统间移植。

7.2 核心组件

在Spring Security3，spring-security-core这个jar包的内容被剥光到最小了。它不再包含任何与web应用安全或者LDAP或者命名空间相关的代码。我们在这里看一眼其中的一些java类，你会找到一些核心模块。它们代表了整个框架的大的构建模块，所以如果你需要深入定制命名空间，理解这一块是必要的。

7.2.1 SecurityContextHolder，SecurityContext，和Authentication对象

最基础的对象是SecurityContextHolder。这是我们存储当前应用安全上下文的地方，这包含当前应用各项安全主体（principal，表示用户的标识、角色、操作等）的细节。默认的，SecurityContextHolder使用一个ThreadLocal来存储这些细节，这意味着在相同线程的执行中安全上下文永远是可用的，即使安全上下文没有显式的作为方法的参数传到这些方法中。在这种情况下使用ThreadLocal是很安全的，只需注意在当前安全主体对应的请求已经处理结束后进行一下清理。当然了，Spring Security已经为你做好了这些，你不需要担心。
一些应用不是整体上都适合使用ThreadLocal的，因为有可能它们有特殊的利用线程的方式。例如，一个Swing客户端或许期望所有的jvm中的线程都使用同一个安全上下文。SecurityContextHolder可以在启动时配置一种策略，来指定你如何存储安全上下文。对于一个单独应用你可以使用SecurityContextHolder.MODE_GLOBAL策略。其它应用或许期望安全线程大量生产的子线程也能具备相同的安全身份。这可以使

用SecurityContextHolder.MODE_INHERITABLETHREADLOCAL选项。你可以通过两种方式来修改默认的SecurityContextHolder.MODE_THREADLOCAL。第一种是设定一个系统属性，第二种是调用SecurityContextHolder的一个静态方法。大部分应用不需要改变默认配置，如果你需要，请参见SecurityContextHolder的文档。

获取当前用户的信息

在SecurityContextHolder内部，我们存储了与当前应用交互的安全主体（principal）的细节。Spring Security使用Authentication对象来代表这一信息。你通常不需要自己创建这个对象，但是查找这个对象的操作倒是很常见的。你可以使用下面的代码块，在你应用的任何地方都可以，来获取当前认证用户的名字：

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

if (principal instanceof UserDetails) {
String username = ((UserDetails)principal).getUsername();
} else {
String username = principal.toString();
}
```

getContext()方法返回的对象是一个SecurityContext接口的实例。这是保存在threadLocal中的对象。我们下面会看到，Spring Security中大多数的认证机制都会返回一个UserDetails对象作为安全主体。

7.2.2 UserDetailsService

上面的代码中另外一个需要注意的点是，你可以从Authentication对象中获取安全主体。安全主体仅是一个Object（类的实例）。大部分时间这个对象可以被强制转型为一个UserDetails对象。UserDetails是Spring Security中的一个核心接口，它代表了一个安全主体，但是表现为一种可扩展的及应用指定的方式。可以将UserDetails视为你自己的用户数据库及Spring Security的SecurityContextHolder需要的类型之间的适配器（adapter）。作为一个从你自己的用户数据库抽出的信息的代表，常常你会将你应用提供的那个原始对象转型成一个UserDetails，所以你可以继续调用业务指定的方法。

到现在为止，你或许会琢磨，什么时候需要我提供一个UserDetails对象？我怎么提供？我认为你说的这些东西看上去都是声明出来的，我不需要写任何的java代码，那么是谁写的？简短的回答是，有一个特殊的接口称为UserDetailsService。这个接口的唯一方法接受一个String参数的用户名并且返回一个UserDetails：

```
UserDetails loadUserByUsername(String username) throws UsernameNotFoundException;
```

这是在Spring Security中最常用的读取一个用户的信息的方法，你将会在整个框架中经常见到它，当需要知道用户信息的时候。

在认证成功时，UserDetails用来构建存储在SecurityContextHolder中的Authentication对象。好消息是我们提供了一系列UserDetailsService的实现，包括使用内存映射表的InMemoryDaoImpl，和使用JDBC的JdbcDaoImpl。大多数用户倾向于使用他们自己写的，这种情况下他们的实现经常是简单的处于自己应用DAO层的上层，代表了应用中的企业用户，商户等等。记住不论你的UserDetailsService返回什么，都可以通过上面的代码片段来获取。

通常对于UserDetailsService总会有一些疑惑。它纯是一个DAO来获取用户信息并将这些信息提供给整个框架的其他组件的东西，并没有其它功能。特别的，它不执行认证，这是由AuthenticationManager完成的。在许多情况下实现AuthenticationProvider是更清晰的，如果你需要一个个性化的认证过程的话。

7.2.3 GrantedAuthority

除了应用主体，另外一个Authentication提供的重要方法是getAuthorities()。这个方法提供一个GrantedAuthority对象数组。一个GrantedAuthority对象是指，授予安全主体的权力。这样的权力通常是"角色"，例如ROLE_ADMINISTRATOR或者ROLE_HR_SUPERVISOR。这些角色以后会被配置用来做web、方法、领域对象的认证。Spring Security的其它部分也能够理解这些权限。GrantedAuthority对象通常被UserDetailsService加载。

通常，GrantedAuthority对象在整个应用范围内适用。他们不是限定在具体某个领域对象上。这样，你就不会出现某个GrantedAuthority对象代表了某个具体用户的权限这种情况出现，如果真的是这样的话，如果有成百上千这样的权限，你很快就会内存溢出了（或者最少，会导致你的应用花费特别长的时间来认证一个用户）。所以，Spring Security特地的做出设计来解决这个通用的需求，但是你经常会取而代之的使用工程的领域对象安全性来达到这一目的。

7.2.4 总结

概括一下，Spring Security至今为止的主要构件模块是：

SecurityContextHolder，来提供对SecurityContext的访问

SecurityContext，持有Authentication，通常还有针对请求的安全信息

Authentication，Spring Security指定的表示安全主体的方式

GrantedAuthority，代表整个应用范围内的授予一个安全主体的权力

UserDetails，用来从你应用的DAO层或者其它安全数据源提供必要的用来构建Authentication对象的信息

UserDetailsService，接收一个基于String的用户名，创建一个UserDetails

现在你大致理解了这些天天念叨的组件，下面让我们近距离观察一下认证的流程。

7.3 Authentication

Spring Security可以参与到多种不同的认证环境中。然而我们建议使用Spring Security来实现认证，而不要与已存的由容器管理的认证混合使用，或者与你自己写的认证体系混合使用，虽然这也是支持的。

7.3.1 什么是Spring Security中的认证？

让我们考虑一个每个人都熟悉的标准认证场景：

1、一个用户被提示使用用户名和密码登录

2、系统（成功的）验证当前密码属于该用户

3、该角色的上下文信息被获取（他的角色列表等等）

4、该用户的安全上下文建立

5、用户执行一些操作，这些操作潜在的被访问控制机制所保护，依据的是检查当前操作在安全上下文中是否允许

其中前三项构成认证过程，所以我们会看一下再Spring Security中这些是如何进行的。

1、用户名及密码被获取，并写入一个UsernamePasswordAuthenticationToken（一个Authentication接口的实例）

2、该令牌被传递到AuthenticationManager中用来验证

3、AuthenticationManager在成功认证的情况下，返回一个完全赋值完毕的Authentication实例

4、通过调用SecurityContextHolder.getContext().setAuthentication(...)方法建立安全上下文，将上面生成的Authentication对象传给它

从这时候开始，用户被认为是已经得到认证了。让我们看一下下面的例子：

```
import org.springframework.security.authentication.*;
import org.springframework.security.core.*;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.core.context.SecurityContextHolder;

publicclass AuthenticationExample {
privatestatic AuthenticationManager am = new SampleAuthenticationManager();

publicstaticvoid main(String[] args) throws Exception {
 BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

 while(true) {
 System.out.println("Please enter your username:");
 String name = in.readLine();
 System.out.println("Please enter your password:");
 String password = in.readLine();
 try {
  Authentication request = new UsernamePasswordAuthenticationToken(name, password);
  Authentication result = am.authenticate(request);
  SecurityContextHolder.getContext().setAuthentication(result);
  break;
 } catch(AuthenticationException e) {
  System.out.println("Authentication failed: " + e.getMessage());
 }
 }
 System.out.println("Successfully authenticated. Security context contains: " +
   SecurityContextHolder.getContext().getAuthentication());
}
}

class SampleAuthenticationManager implements AuthenticationManager {
staticfinal List<GrantedAuthority> AUTHORITIES = new ArrayList<GrantedAuthority>();

static {
 AUTHORITIES.add(new SimpleGrantedAuthority("ROLE_USER"));
}

public Authentication authenticate(Authentication auth) throws AuthenticationException {
 if (auth.getName().equals(auth.getCredentials())) {
 returnnew UsernamePasswordAuthenticationToken(auth.getName(),
  auth.getCredentials(), AUTHORITIES);
 }
 thrownew BadCredentialsException("Bad Credentials");
}
}
```

这里我们写了一段小程序，要求用户输入用户名和密码，执行上面的片段。这里我们实现的AuthenticationManager会对任何用户名等于密码的用户通过认证。它对每个用户分配了一个单独的角色，上面的输出大概是这样：

```
Please enter your username:
bob
Please enter your password:
password
Authentication failed: Bad Credentials
Please enter your username:
bob
Please enter your password:
bob
Successfully authenticated. Security context contains: \
org.springframework.security.authentication.UsernamePasswordAuthenticationToken@441d0230: \
Principal: bob; Password: [PROTECTED]; \
Authenticated: true; Details: null; \
Granted Authorities: ROLE_USER
```

注意你通常你不需要这样写代码。这个处理通常是在内部执行，例如web认证过滤器中。这个例子让我们对Spring Security实际上如何构成认证有了一个初步的了解，如果SecurityContextHolder持有了一个赋值完全的Authentication对象，则意味着一个用户已经被认证了。

7.3.2 直接设置SecurityContextHolder的内容

事实上，Spring Security不关心你如何将Authentication对象放进SecurityContextHolder，唯一关键的需求是SecurityContextHolder能在AbstractSecurityInterceptor需要对一个用户做认证之前具备一个代表安全主体的Authentication。

你可以写自己的过滤器和controller来与不基于Spring Security提供的认证系统协同工作。例如，你或许使用容器管理的认证，这允许当前用户可以使用某个ThreadLocal或者JNDI位置。或者你或许为一家具有私有认证系统的企业工作，这个“企业标准”你基本无法控制。在这些情况下，很容易让Spring Security进行工作，提供认证能力。你所需要做的一切就是写一个过滤器（或者等价的东西）来从指定位置读取第三方用户信息，构建一个Spring Security指定的Authentication对象，然后将它放进SecurityContextHolder。在这种情况下，你也需要去考虑本来应该是内建的认证架构自动会去做的事情。例如，你或许需要先去创建一个HTTP session来在请求之间缓存上下文，这应该发生在向客户端写响应之前。（注：在响应提交以后就不可能创建session了）

如果你在琢磨AuthenticationManager是如何实现的，参加核心服务章节。

7.4 Web应用中的认证

现在让我们探索一下当你在web应用中使用Spring Security的情况（没有开启web.xml安全），用户的认证和安全上下文是如何建立的？
考虑典型的web应用中的认证流程：
1、你访问一个主页，点击一个链接
2、一个请求发到服务器，服务器看一下你申请的是不是一个受保护的资源
3、因为你当前还没被认证，服务器发回一个响应，指示你必须认证。该响应要么是一个http响应代码，要么重定向到一个特殊的网页
4、依赖认证机制，你的浏览器要么重定向到指定的网页，你来填写表单，要么浏览器用某种方式获取你的身份（通过BASIC认证对话框，cookie，X.509证书等）
5、浏览器会发回服务器一个响应。这要么是一个包含你在登录表里填写的内容的HTTP POST，要么是一个包含你的认证细节的HTTP头
6、然后服务器会判定到底当前的凭证是不是合法的。如果是合法的，执行下一步；如果是非法的，通常浏览器会被要求重试
7、你引发认证的原始请求会被重试。充满着你被认证成功，具有足够的权限来访问那个受保护资源的希望。如果你具有足够的访问权，请求会成功，否则，你会被返回403。

Spring Security对于大部分上面描述的步骤都有具体的相关责任类来负责。主要的参与者（按照它们使用的顺序）是ExceptionTranslationFilter，AuthenticationEntryPoint，和一个“认证机制”，这通过调用AuthenticationManager来实现。

7.4.1 ExceptionTranslationFilter

ExceptionTranslationFilter是一个负责探测任何Spring Security抛出的异常的过滤器。这种异常通常是被AbstractSecurityInterceptor抛出的，这是认证服务的主要提供者。我们在下一章讨论AbstractSecurityInterceptor，但是现在我们只需要知道这家伙创造java异常，并且对http或者如何认证安全主体一无所知。相对的ExceptionTranslationFilter提供这一功能，其责任为要么返回一个错误码403（如果安全主体被认证过，发现权限不足以访问），要么加载一个AuthenticationEntryPoint（如果安全主体没有认证过并且因此我们需要返回上面的步骤3）。

7.4.2 AuthenticationEntryPoint

AuthenticationEntryPoint在上面的步骤中对步骤3负责。就像你猜的那样，每个web工程都会有一个默认的认证策略（恩，这可以跟其他东西一样的在Spring Security中配置，但是我们现在仅考虑最简单的情况）。每个主要的认证系统都会有其自己的AuthenticationEntryPoint实现，这通常会支持步骤3的一种实现。

7.4.3 认证机制

一旦你的浏览器提交了你的认证证书（要么一个http post表单，要么一个http头），服务器端就需要一个东西来收集这些认证细节。现在我们在上面步骤的第6步。在Spring Security中我们有一个特殊的名称来称呼从用户端（通常是web浏览器）收集认证细节的功能，称为“认证机制”。例子是基于表单的登录或者是BASIC认证。一旦认证细节从用户端收集好了，一个Authentication“请求”对象就被创建并且提供给AuthenticationManager。
在认证机制收到赋值好的Authentication对象之后（这里的意思是AuthenticationManager认证通过，才会返回Authentication对象），它将会认为请求是合法的，将Authentication对象放进SecurityContextHolder，然后导致原始请求重试（第7步）。如果，另一种可能，AuthenticationManager拒绝请求，认证机制会让用户端重试（第2步）

7.4.4 在请求之间存储SecurityContext

针对不同类型的应用，或许需要一个策略来在用户的操作之间保存安全上下文。在典型的web应用中，一个用户登录一次，后续的操作都通过其session id识别。服务器在整个session的生命周期缓存安全主体信息。在Spring Security中，在请求之间存储SecurityContext的责任落在了SecurityContextPersistenceFilter肩上，它默认的将安全上下文作为HttpSession的一个属性保存。它针对每个请求将安全上下文恢复到SecurityContextHolder中，最关键的，当请求完成时清空SecurityContextHolder。你不需要出于安全的角度直接与HttpSession对象交互。为什么呢？因为你没什么正当理由用它，想用的话就使用SecurityContextHolder吧。
许多其它类型的应用（例如：无状态的RESTful服务）不使用http session，会在每个请求上重新认证。然而，在过滤器链中包含SecurityContextPersistenceFilter仍然是非常重要的，来保证SecurityContextHolder在每次请求后都被清除。

在那种在一个session中接收多线程请求的应用中，相同的SecurityContext实例会在线程之间共享。即使使用ThreadLocal，对每个线程都是从HttpSession中获取相同的实例。如果你期望临时的改变在某个正在运行的线程下的安全上下文，这是有影响的。如果你仅仅使用SecurityContextHolder.getContext()，然后在返回的上下文对象上调用setAuthentication(anAuthentication)，那么Authentication对象会在所有的共享同一个SecurityContext的并发线程上改变。你可以个性化定制SecurityContextPersistenceFilter的行为，来针对每个请求创建一个完整的新的SecurityContext，防止对一个线程的改变影响了另一个线程。另外一种方法是你可以在你需要临时改变上下文的地方（那个时间点）创建一个新的实例。SecurityContextHolder.createEmptyContext()方法总是返回一个新的上下文对象。

7.5 Spring Security中的访问控制（授权）

在Spring Security中对访问控制决策负责的主接口是AccessDecisionManager。它具有一个decide方法，拿一个Authentication对象代表主体请求的访问，一个“安全对象”和一个安全元数据属性列表被应用在这个对象上（例如，访问被授权所需要的角色列表）

7.5.1 安全与AOP切面

如果你对AOP熟悉，你会注意到切面有多种可能的类型：前置切面，后置切面，抛出后切面，和环绕切面。环绕切面是非常有用的，因为advisor可以自行决定是否去处理方法调用，是否去修改返回值，是否去抛出一个异常。Spring Security对方法调用（或者web请求）提供一种环绕切面。我们通过标准的Spring AOP实现对于方法调用的环绕切面，而对于web请求，我们使用标准的过滤器来达到环绕切面的效果。
对于那些不熟悉AOP的选手，你只需要记住Spring Security可以帮助你保护方法调用和web请求。多数人对在他们的服务层实现安全方法调用感兴趣。这是因为服务层是当前版本的Java EE应用中业务逻辑所在的位置。如果你仅仅想要对服务层的方法调用实现安全性，标准的Spring AOP就足够了。如果你需要直接保证领域对象的安全性，你或许会发现AspectJ值得考虑。
你可以选择使用AspectJ或者SpringAOP做方法授权，或者你可以选择使用过滤器执行web请求的授权。你可以结合使用0个，1个，2个或者3种以上的方法。主流的使用模式是执行一些web请求授权，同时结合一定的Spring AOP应用层方法调用级别的授权。

7.5.2 安全对象及AbstractSecurityInterceptor

所以，到底什么是“安全对象”？Spring Security使用这个术语来代表任何可以被应用安全性的对象（例如认证决策）。最常见的例子是方法调用和web请求。
每个被支持的安全对象类型都有它自己的拦截器类，拦截器类是AbstractSecurityInterceptor的子类。重要的是，在AbstractSecurityInterceptor对象被调用的时候，SecurityContextHolder将会包含一个合法的Authentication，如果安全主体被认证的话。
AbstractSecurityInterceptor提供一个连贯的处理安全对象请求的工作流，典型的：
1、查找与当前请求联系的“配置属性”
2、提交安全对象、当前的Authentication和配置属性到AccessDecisionManager中来做认证决策
3、有可能在调用发生的地点改变Authentication
4、允许安全对象调用被处理（假设访问被授权）
5、一旦调用返回，如果有配置AfterInvocationManager，则调用之。如果调用抛出一个异常，则AfterInvocationManager不会被调用。

什么是配置属性？

配置属性，可以被认为是一个对AbstractSecurityInterceptor使用的类具有特殊意义的String。在框架中它们被ConfigAttribute接口代表。它们可以是简单的角色名或者有更复杂的含义，依赖于AccessDecisionManager的实现有多复杂。AbstractSecurityInterceptor配置了SecurityMetadataSource，用来查找安全对象的属性。通常的这个配置对用户隐藏。配置属性会作为被保护方法上的注解或者被保护url上的访问属性被进入。例如：当我们看见命名空间中的这种声明：<intercept-url pattern='/secure/**' access='ROLE_A,ROLE_B'/>，这是说配置属性ROLE_A和ROLE_B应用到匹配给定模式的web请求。实践中，对于默认的AccessDecisionManager配置，这意味着任何一个具有能匹配上这两个属性中任意一个的GrantedAuthority的人被允许访问。尽管严格的说，他们仅仅是个属性，而且它们代表的含义依赖于AccessDecisionManager怎么理解。前缀ROLE_是一个标识符，说明这个属性是个角色，应该被Spring Security的RoleVoter使用。这是当基于投票的AccessDecisionManager被使用时唯一的关联。我们将在认证章节看看AccessDecisionManager是如何实现的。

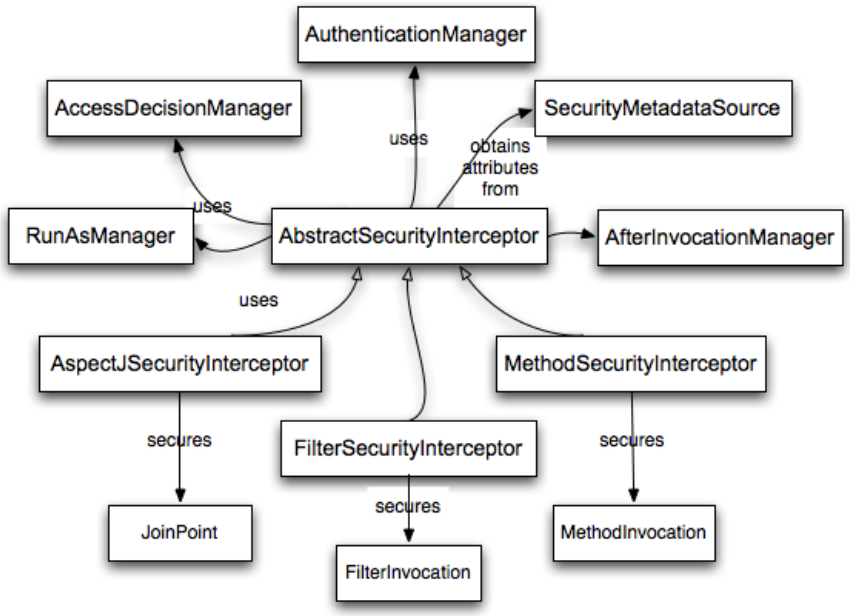
RunAsManager

假设AccessDecisionManager决定允许请求，AbstractSecurityInterceptor会正常的就去处理那个请求。已经说了，在很少的情况下，用户获取想要将SecurityContext中的Authentication替代为另一个不同的Authentication，这通过AccessDecisionManager调用一个RunAsManager来实现。这或许在一些非常不寻常但是合理的场景下是非常有用的，例如如果一个服务层方法需要调用一个远端系统并且呈现一个不同身份的话。因为Spring Security自动的从一个服务器到另一个服务器传播安全id（假设你使用了正常配置的RMI或者HttpInvoker远程协议客户端），这或许是有用的。

AfterInvocationManager

紧接着安全对象的调用处理和随后的返回（这或许意味着一个方法调用的完成或者一个过滤器链处理完成），AbstractSecurityInterceptor得到了一个最后的机会来处理调用。在这个场合，AbstractSecurityInterceptor对于修改返回对象或许会感兴趣。我们也许期望这一事件发生，因为一个认证决策不能在被保护对象调用时做出。为了做成高插拔性，AbstractSecurityInterceptor会将控制权传递给AfterInvocationManager来执行实际的对对象的修改，如果需要的话。这个类甚至可以整个替换掉对象，或者抛出一个异常，或者不修改任何东西。仅当调用成功后，调用后检查才会执行。如果异常抛出，额外的检查将跳过。
AbstractSecurityInterceptor和它相关的对象如下图展示：

Figure 7.1. Security interceptors and the "secure object" model



扩展安全对象模型

只有期望一个整个的新的拦截和认证请求方式的开发者会需要直接使用保护对象。例如，或许会构建一个新的保护对象用作消息系统的安全调用。任何需要安全的或提供拦截调用的方法的东西都能被做成受保护的对象（例如AOP环绕切面语义）。需要说明一下的是，大部分Spring应用会简单的透明的使用三种当前支持的保护对象类型（AOP联盟的MethodInvocation，AspectJ的JoinPoint和web请求的FilterInvocation）。

7.6 区域化

Spring Security支持异常消息的区域化，这样终端用户喜闻乐见。如果你的应用是为说英文的用户设计的，你不需要做任何事情，因为Spring Security的信息默认是英文的。如果你需要其他的地区信息，这一章节会描述。
所有的异常消息都会被区域化，包括认证失败的消息和访问被拒的消息（授权失败）。开发者或者系统部署者关注的异常和登录消息（包括不正确的属性，接口转型错误，使用了不正确的构造器，启动超时，调试日志等）是硬编码在Spring Security的源码中的，是不能区域化的。
移步到spring-security-core-xx.jar，你会找到一个org.springframework.security包，包含一个message.properties文件，和一些常见语言的区域化版本。这应该被你的ApplicationContext关注，因为Spring Security的类实现了Spring的MessageSourceAware接口，期待其依赖的消息解析器在应用上下文启动的时候被注入。通常的所有你需要做的是在应用上下文中注册一个bean来指向这些消息，例如：

```
<beanid="messageSource"class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
<propertyname="basename"value="classpath:org/springframework/security/messages"/></bean>
```

message.properties使用标准的资源文件格式命名，代码Spring Security消息支持的默认语言。默认文件是英语的。
如果你想去定制message.properties文件，或者支持其它语言，你应该拷贝这个文件，按照规矩重命名，然后在上面的bean定义中注册。在这个文件里没有大量的key，所以区域化本身不是个大工作。如果你在这个文件中做了区域化，请考虑通过jira任务的的形式来分享你的工作。
Spring Security依赖Spring的区域化支持，来查找合适的消息。为了这个工作，你需要确保进入的请求的区域已经在org.springframework.context.i18n.LocaleContextHolder中存储好了。Spring MVC的DispatcherServlet自动的为你的应用做了这一工作，但是因为Spring Security的过滤器在此之前调用，LocaleContextHolder需要在过滤器调用之前就设定好包含有需要的信息。你可以要么在一个你自己的过滤器中做这件事（这必须在web.xml的Spring Security定义的过滤器之前），或者你可以使用Spring的RequestContextFilter。具体请参见Spring关于区域化的描述。

8.核心服务

现在我们开始对Spring Security架构及其核心类进行更高层次的概览。让我们近距离观察一到两个核心接口和它们的实现，尤其是AuthenticationManager，UserDetailsService和AccessDecisionManager。它们会在这篇文档中随时出现，所以了解它们是如何配置和操作的是非常重要的。

8.1 AuthenticationManager，ProviderManager，AuthenticationProvider

AuthenticationManager仅仅是个接口，所以其实现可以是我们选择的任意实现，但是实践中它是怎么工作的呢？如果我们需要检查多个认证数据库或者不同的认证服务（例如数据库或者LDAP服务器）会发生什么？
Spring Security的默认实现称作ProviderManager，但是区别于自己处理认证请求，它将认证请求的处理委托给一系列配置好的AuthenticationProvider，它们中的每一个都会被询问，来看看是否可以执行认证。每个认证提供器都会要么抛出一个异常，要么返回一个灌注好内容的Authentication对象。还记得我们的好朋友，UserDetails和UserDetailsService吗？如果忘了，请回头看看之前的章节，刷新一下你的脑子。最常见的验证一个认证请求的方法是加载相关的UserDetails并且检查用户输入的密码与保存的密码是否一致。这是DaoAuthenticationProvider使用的方法。加载进来的UserDetails对象，尤其是它包含的GrantedAuthority对象，会在灌注成功认证时的Authentication对象的时候使用，并保存到SecurityContext。
如果你使用命名空间，内部会创建并维护一个ProviderManager的实例，然后你通过命名空间的authentication provider元素向其添加认证提供者。在这种情况下，你不应该在你的应用上下文中声明ProviderManager这个bean。然而，如果你没使用命名空间，你应该这样声明：

```
<beanid="authenticationManager"class="org.springframework.security.authentication.ProviderManager"><const
ructor-arg><list><reflocal="daoAuthenticationProvider"/><reflocal="anonymousAuthenticationProvider"/><ref
local="ldapAuthenticationProvider"/></list></constructor-arg></bean>
```

在上面的例子中，我们有三个提供器。它们会按照配置的顺序被尝试查询，每个提供器都会尝试认证，或者通过返回null来跳过认证。如果所有的实现都返回null，ProviderManager会抛出一个ProviderNotFoundException。如果你对此感兴趣，参见ProviderManager文档。
认证机制，例如web登录表单处理过滤器被作为一个引用注入到ProviderManager，并且被调用来处理认证请求。你需要的提供器，在某些时候是可以与认证机制交换的。例如：DaoAuthenticationProvider和LdapAuthenticationProvider是与任何提交简单的用户名/密码认证请求兼容的，所以可以与基于表单的或基于http的认证协同工作。另一方面，一些认证机制创建的认证请求对象，仅仅能被一种单独的AuthenticationProvider理解。这个的一个例子是JA-SIG CAS，它使用一种服务门票的概念，只能被CasAuthenticationProvider理解。你不需要关心这些，因为如果你忘记注册合适的提供器，当尝试认证的时候你会简单的获得ProviderNotFoundException。

8.1.1 在成功的认证上擦除凭证

默认的（从Spring Security3.1以后），ProviderManager会尝试清除任何从成功的认证请求返回的Authentication对象中的敏感的凭证信息。这可以防止密码之类的信息在不必要的时间内仍然存在。
但是如果你使用用户对象缓存的时候，这会发生一点问题。例如，为了改善无状态应用的性能。如果Authentication对象包含了在缓存中的对象的引用（例如UserDetails实例）并且其凭证被删除了，则对于缓存的值的认证就不再有效了。如果你需要使用缓存，你要考虑这一点。之前的解决方案是首先创建一个该对象的拷贝，要么在缓存的实现中要么在返回Authentication对象的AuthenticationProvider中。相对的，你可以关闭ProviderManager的eraseCredentialAfterAuthentication属性。

8.1.2 DaoAuthenticationProvider

在Spring Security中最简单的AuthenticationProvider的实现是DaoAuthenticationProvider，这也是框架最早支持的。它使用一个UserDetailsService（作为DAO）来查询用户名，密码，及GrantedAuthority。它简单的通过比对从UsernamePasswordAuthenticationToken中提交上来的密码及UserDetailsService加载的密码来实现对用户的认证，配置如下：

```
<beanid="daoAuthenticationProvider"class="org.springframework.security.authentication.dao.DaoAuthenticati
onProvider"><propertyname="userDetailsService"ref="inMemoryDaoImpl"/><propertyname="passwordEncoder"ref="
passwordEncoder"/></bean>
```

PasswordEncoder是可选的。PasswordEncoder提供对UserDetailsService返回的UserDetails对象中的密码的编码及解码。这将在以后讨论。

8.2 UserDetailsService实现

像这篇文档之前提到的，大部分认证提供器利用UserDetails和UserDetailsService接口。回忆UserDetailsService接口，它是一个单一方法：

```
UserDetails loadUserByUsername(String username) throws UsernameNotFoundException;
```

返回的UserDetails是一个接口，提供存取器保证给出非空的认证信息，例如用户名，密码，授权的权限和该账户是否激活等等。大部分认证提供器会使用一个UserDetailsService，即使用户名和密码没有实际的用来做认证决策。它们或许使用返回的UserDetails对象仅仅为了获取GrantedAuthority信息，因为一些其他的系统（例如LDAP或者X.509或者CAS等），承担了事实上如何验证凭据的方式。
UserDetailsService实现起来非常简单，它应该提供一个对用户来说很简单的一致性的策略来返回认证信息。已经说了，Spring Security已经包含了一系列有用的基础实现。

8.2.1 内存中认证

创建一个个性化的UserdetailsService实现类，从自己选的持久化引擎中抽取信是很简单的，但是许多系统不需要这么复杂。这在当你构建一个原型应用或者仅仅是刚开始整合Spring Security的时候特别有用，此时你还没真正的想花时间配置数据库或者写UserDetailsService实现。对于这种情况，一个简单的选择是使用命名空间中的user-service：

```
<user-serviceid="userDetailsService"><username="jimi"password="jimispassword"authorities="ROLE_USER, ROLE
_ADMIN" /><username="bob"password="bobspassword"authorities="ROLE_USER" /></user-service>
```

使用外部配置文件也是支持的：

```
<user-serviceid="userDetailsService"properties="users.properties"/>
```

这个配置文件应该包括如下表格这种键值对：

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

例如：

```
jimi=jimispassword,ROLE_USER,ROLE_ADMIN,enabled
```

```
jimi=jimispassword,ROLE_USER,ROLE_ADMIN,enabled
bob=bobspassword,ROLE_USER,enabled
```

8.2.2 JdbcDaoImpl

Spring Security也有能从JDBC数据源获取认证信息的UserDetailsService。内部是使用Spring JDBC，所以它规避了全特性的对象关系映射（ORM）的复杂性，仅仅存一些用户信息。如果你的应用使用了ORM工具，你或许会偏好自己写一个UserDetailsService来重用你代码中已经有的映射关系。本段讲的方法最后通过配置返回一个JdbcDaoImpl来实现，例子如下：

```
<beanid="dataSource"class="org.springframework.jdbc.datasource.DriverManagerDataSource"><propertyname="driverClassName"value="org.hsqldb.jdbcDriver"/><propertyname="url"value="jdbc:hsqldb:hsql://localhost:9001"/><propertyname="username"value="sa"/><propertyname="password"value=""/></bean><beanid="userDetailsService"class="org.springframework.security.core.userdetails.jdbc.JdbcDaoImpl"><propertyname="dataSource"ref="dataSource"/></bean>
```

你可以通过修改DriverManagerDataSource使用不同的关系数据库管理系统。你也可以使用一个基于JNDI的全局数据源等等。

权限组

默认的，JdbcDaoImpl读取单独用户的权限时，假设权限直接映射到用户本身。另外一种方法是将权限分组并且将权限组指定到用户。一些人喜欢这种管理用户权限的方式。这种做法如何实现参加参见

8.3 密码编码

Spring Security的PasswordEncoder接口用来支持密码加密存储。你永远也不应该存储密码原文。永远使用一种单向的密码哈希算法。诸如bcrypt，使用内建的盐值加密，对于每个存储的密码盐值都不一样。不要使用平平无奇的哈希算法如MD5或者SHA，或者即使是其加盐版本。bcrypt故意被设计为很慢的并且妨碍线下密码破解的，相对而言，标准的哈希算法可以算的很快并且能够通过存储在普通硬件上的成百上千的密码并行测试来破解。你或许认为这不关你事，因为你的密码数据库是安全的并且线下的攻击不足为惧，如果是这样的话，做一些科研，看看是不是大多数受瞩目的网站在这方面会妥协，以至于使自己因为不恰当的保存用户的密码而被嘲笑。最好是直接站在安全性的一边。使用org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder是对安全性的一个好选择。在其它的编程语言中这一实现也会有兼容版本，因此对于互用性来说也是一个好选择。
如果你使用一个私有系统并且已经有哈希处理的密码，那么你会需要使用一种能够匹配你当前算法的编码器，最少直到你能融合你的用户到一个更加安全的机制中（通常这个会涉及让用户设定新密码，因为哈希是不可逆的）。Spring Security有一个包包含私有的密码加密算法实现，命名为：org.springframework.security.authentication.encoding。DaoAuthenticationProvider可以用其中的任意一个注入，或者私有的PasswordEncoder类型。

8.3.1 哈希是什么？

对密码进行哈希不是Spring Security的专利，但是对于不熟悉这个概念的用户来说总是容易混淆的。哈希（或者叫做摘要（digest））算法是一种单向函数，针对输入数据（例如密码）生成一个固定长度的数据片。作为例子，MD5哈希算法针对字符串“password”的处理结果为：

```
5f4dcc3b5aa765d61d8327deb882cf99
```

哈希算法“单向”的意义是，如果给定你一个哈希值，是很难（通常是不可能）来获得原始输入数据的，或者说有可能好多种不同的输入数据都有可能得到相同的哈希值。这个特性使得哈希值在认证过程中非常有用。它们可以存储在你的用户数据库中作为密码原文的替代品，或者该值是一个中间产物，并不直接代表一个用户密码。注意这同时意味着你在对密码哈希以后无法得到其原文。

8.3.2 哈希上加盐

对密码进行哈希的一个潜在问题是，如果一个常见的单词用作密码输入，（通过比对）可以相对简单的猜到这个单向哈希的算法。人们倾向于选择相似的密码，然而这些密码被黑客们从之前的网站上黑到而组成的密码字典已经广为存在了。例如，如果你使用谷狗搜索哈希值：5f4dcc3b5aa765d61d8327deb882cf99，你会很容易得知其原始单词“password”。相似的方法，攻击者可以从标准词汇表出发构建一个哈希字典，然后使用这个字典来查找原始的密码。一种帮助你防止这种事情的方法是搞一个合适的强密码策略来防止常见的单词被用作密码。另一个方法是在计算哈希的时候使用“盐值”。这对每个用户来讲是一个已知数据的额外的串，添加到密码上，然后再进行哈希。理想的数据是越随机越好，但是实践中任何盐值通常都是比没有强。使用盐值意味着攻击者需要去针对每个盐值构建一个单独的哈希字典，使得攻击更加复杂（单不是不可能）。
Bcrypt对每个密码在编码的时候自动生成一个随机盐值，然后将它使用标准格式存到一个bcrypt串中。

处理盐值的方法是对DaoAuthenticationProvider注入一个SaltSource，它会获取一个针对每个用户的盐值，并传进PasswordEncoder。使用bcrypt意味着你不需要关注盐值处理的细节（例如值存在哪），内部都为你做好了。所以我们强烈的推荐你使用bcrypt除非你已经有了一个存储盐值的系统了。

8.3.3 哈希与认证

当认证处理提供者（例如Spring Security的DaoAuthenticationProvider）需要比对提交的认证请求中的密码与已经保存的用户密码，并且存储的密码已经以某种方式编码了，那么提交的密码必须以存储的密码相同的算法进行哈希。你需要自己来检查这些方法是不是兼容，因为Spring Security在这些值上不做什么控制。如果你在Spring Security的认证配置中添加了密码哈希，并且你的数据库包含原文密码，那么认证就不可能成功了。即使你的数据库使用的是MD5对密码进行编码你也要警惕，比如此时如果你的应用配置了Spring Security的Md5PasswordEncoder，认证也可能发生错误。因为数据库可能用的是Base64编码，但是编码器默认使用的是十六进制串。也有可能你的数据库使用的是大写但是从编码器出来的字符是小写等等。在真正投产之前，使用一个随机的密码和已知的盐值测试一下。然而使用标准的编码器例如bcrypt可以避免这些问题。
如果你想要直接在java中生成已编码的密码用来存储在数据库，你可以使用PasswordEncoder的encode方法。

11.安全过滤器链

Spring Security的web构造完全基于标准的servlet过滤器。它不使用servlets或者任何基于servlet的框架（例如Spring MVC），所以它与任何特定的web技术都没有很强的联系。它处理HttpServletRequest和HttpServletResponse，但是并不关心究竟请求是从浏览器来的还是web service客户端来的，还是HttpInvoker或者是AJAX来的。
Spring Security内部维护了一个过滤器链，其中每个过滤器都有特定的职责，过滤器可以依据服务的需要，在配置文件中添加或者删除。过滤器的顺序是很重要的，因为它们之间可能有依赖关系。如果你使用命名空间的配置，那么过滤器自动的为你设定好，你不需要显式定义任何的Spring bean，但是有时候你可能需要对整个安全过滤器链的完全控制，这或者是因为你想要使用命名空间配置不支持的功能，或者是使用你自己个性化定制的类。

11.1 DelegatingFilterProxy

当你使用servlet过滤器时，你需要在你的web.xml中配置它们，否则它们就会被servlet容器忽略掉。在Spring Security中，过滤器也是在应用上下文中定义的Spring bean，这样就能够利用Spring的丰富的依赖注入设施和生命周期接口。Spring的DelegatingFilterProxy提供了将web.xml和应用上下文连接起来的桥梁。
当使用DelegatingFilterProxy时，你可能看见如下配置：

```
<filter><filter-name>myFilter</filter-name><filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class></filter><filter-mapping><filter-name>myFilter</filter-name><url-pattern>/*</url-pattern></filter-mapping>
```

注意这个过滤器实际上是一个DelegatingFilterProxy，而不是实际实现了过滤器逻辑的类。DelegatingFilterProxy做的事情是将过滤器的方法代理到一个从Spring的应用上下文中获取的bean上。这使得这个bean可以具有Springweb应用上下文的生命周期及灵活配置的支持。这个bean必须实现javax.servlet.Filter接口，并且必须与filter-name元素中的定义具有相同的名字。（读者注，你可以简单的理解成，这个类会替换掉spring上下文中的某个过滤器bean，前提是这个bean首先得是个过滤器（废话），其次其bean名称要与filter-name一样）

11.2 FilterChainProxy

Spring Security的web架构应该只能通过代理到一个FilterChainProxy来起效。所有的安全过滤器不应该自己单独拿出来用。理论上，你可以在你的应用上下文中声明每个你自己应用需要的Spring Security的过滤器bean，然后在web.xml添加一个个的DelegatingFilterChainProxy，来向每个你自己定义的过滤器bean做映射，并保证它们的顺序是对的，但是这是很笨重的，并且如果你的过滤器很多的话，很快就会把你的web.xml文件写的七零八落。FilterChainProxy允许我们在web.xml中添加一个单独的入口点，来在配置文件中统一处理我们的web安全过滤器bean。它也需要绑定一个DelegatingFilterProxy一起使用，就像上面的例子一样，但是filter-name字段必须写成filterChainProxy。然后所需的过滤器链就可以在应用上下文中用相同的bean名称声明。下面是例子：

```
<beanid="filterChainProxy"class="org.springframework.security.web.FilterChainProxy"><constructor-arg><list><sec:filter-chainpattern="/restful/**"filters="
  securityContextPersistenceFilterWithASCFalse,
  basicAuthenticationFilter,
  exceptionTranslationFilter,
  filterSecurityInterceptor" /><sec:filter-chainpattern="/**"filters="
  securityContextPersistenceFilterWithASCTrue,
  formLoginFilter,
  exceptionTranslationFilter,
  filterSecurityInterceptor" /></list></constructor-arg></bean>
```

命名空间元素filter-chain用来在你的应用中方便的建立需要安全过滤器链。它映射一个特定的URL模式到一个通过filter元素中指定的bean名称来构建的过滤器列表，然后将他们结合到一个SecurityFilterChain类型的bean中。pattern属性使用Ant风格的Paths定义，最特殊的uri应该最先出现。在运行时，FilterChainProxy会定位第一个匹配上当前web请求的uri模式，然后其相对应的filter们会在这个请求上被应用。过滤器会按照它们定义的顺序被调用，所以你可以针对特定的url对过滤器链进行完全控制。
你或许已经注意到了，我们声明了两个SecurityContextPersistenceFilter，（ASC是allowSessionCreation的缩写，一个该过滤器的属性）。因为web service永远不会在未来的请求上展现jsessionid，所以对每个客户端都创建HttpSession是个浪费。如果你有一个需要最大扩展性的大容量的应用，我们推荐你使用上面的方式。对于小一些的系统，使用单独的SecurityContextPersistenceFilter（默认allowSessionCreation为true）可能会更有效率。
注意FilterChainProxy并不调用其上配置的过滤器的标准过滤器的生命周期方法。我们建议你使用Spring的应用上下文生命周期接口来取代，就像你操作普通的spring bean一样。
当我们在看如何通过命名空间配置来设定web安全的时候，我们使用了springSecurityFilterChain名字的DelegatingFilterProxy，你现在应该能了解这是通过命名空间创建的FilterChainProxy的名字。

11.2.1 绕过过滤器链

你可以使用属性filter="none"来配置过滤器bean列表。这将会对于匹配上的请求模式忽略掉整个过滤器链。注意任何匹配上这个路径的东东都完全会略过认证和授权服务，能被完全自由的访问。如果你想要在请求期间使用SecurityContext的东西，那它必须已经被传到安全过滤器链中。否则SecurityContextHolder不会被灌注内容，其内容都是null。

11.3 过滤器顺序

过滤器链中定义的过滤器顺序是非常重要的，不管你具体用了什么过滤器，其顺序应该是如下的：

- `ChannelProcessingFilter`,因为或许需要直接重定向到其它协议
- `SecurityContextPersistenceFilter`,所以 `SecurityContext`能在web请求开始的时候被建立到`SecurityContextHolder` ,并且当web请求结束的时候，任何对于`SecurityContext`的改变能被复制到`HttpSession`中 (准备为下次请求使用)
- `ConcurrentSessionFilter`, 因为它使用 `SecurityContextHolder`的功能，并且需要更新 `SessionRegistry` 来反映当前的请求
- 认证处理机制 - `UsernamePasswordAuthenticationFilter`,`CasAuthenticationFilter`, `BasicAuthenticationFilter` 等等- 所以`SecurityContextHolder` 可以被修改以持有 `Authentication` 请求令牌
- `SecurityContextHolderAwareRequestFilter`, 如果你使用它来装配一个 `HttpServletRequestWrapper` 到你的servlet容器的话
- `JaasApiIntegrationFilter`, 如果 `JaasAuthenticationToken` 存在于`SecurityContextHolder`，这会将 `FilterChain` 像 `Subject` 在`JaasAuthenticationToken`中一样处理
- `RememberMeAuthenticationFilter`, 所以如果没有先前的处理机制来更新 `SecurityContextHolder`, 并且请求带有一个cookie，激活了记住我服务，一个适当的记住我 `Authentication` 对象会被使用
- `AnonymousAuthenticationFilter`, 所以如果没有先前的处理机制来更新 `SecurityContextHolder`, 一个匿名 `Authentication` 对象会被使用
- `ExceptionTranslationFilter`处理任意的Spring Security异常，所以要么一个http错误响应能够被返回，要么一个适当的 `AuthenticationEntryPoint` 能够被加载
- `FilterSecurityInterceptor`,保护web uri，并且当访问拒绝的时候抛出异常

11.4 请求匹配及防火墙HttpFirewall

Spring Security有一些测试（比对）输入的请求与你定义的url模式的地方，来决定请求应该如何被处理。这在当FilterChainProxy决定将请求传递到哪个过滤器链，及FilterSecurityInterceptor决定对请求应用哪种安全限制的时候发生。重要的是理解其机制，及当比对你定义的模式的时候使用的是那个url的值。
Servlet说明书定义了一些HttpServletRequest的属性，可以通过存取器来访问，并且我们或许需要去匹配。它们是contextPath，servletPath，pathInfo和queryString。Spring Security仅对应用中的受保护路径有兴趣，所以contextPath被忽略。不幸的是，Servlet说明书没有精确的定义针对特定的uri请求，servletPath和pathInfo怎么写。例如，每个url的每一段应该含有什么，像RFC2396里写的那样。Servlet说明书也没有清楚的陈述究竟这些是否需要包含在servletPath和pathInfo中，及这些字段在不同的servlet容器中的行为是否有区别。如果一个应用部署在一个不去从这些值中剥离路径变量的容器中的时候可能会造成危险，攻击者可以将它们添加到请求url中来导致一个不可预料的匹配结果。其他的针对输入的url的写法也是有可能的，例如，可以包含路径斜线序列（如/../）或者多个斜线（//），都有可能导致模式匹配失败。一些容器在执行servlet匹配的时候会将这些url规范化，但是有的容器就不会。为了防止这些问题，FilterChainProxy使用一个HttpFirewall策略来检查和包装实际的请求。非规范化的请求会默认被拒绝，路径变量和重复的斜线会被移除。这就是为什么FilterChainProxy用来管理整个安全过滤器链是必要的。注意servletPath和pathInfo值会被容器解码，所以你的应用不应该有任何合法的路径包含分号，因为这些部分会被移除，出于匹配的原因。
如上所述，默认的策略是使用Ant风格的路径来匹配，这可能对大部分用户是一个最好的选择。这个策略在AntPathRequestMatcher中实现，它使用Spring的AntPathMatcher来实现一个大小写敏感的，针对servletPath和pathInfo的匹配，忽略queryString。
如果出于一些原因，你需要一个更强力的匹配策略，你可以使用正则表达式。这个策略被RegexRequestMatcher实现，详情参见javadoc。
在实践中，我们推荐你在服务层使用方法安全性，来控制对你的应用的访问，不要全部都依赖web应用层的安全限制。url会改变，并且照顾所有应用可能支持的url及相关的请求如何处理是很困难的。你应该尝试并限制你自己使用一些简单的容易理解的ant风格的路径。当你在最后有一个处理所有路径的通配符的时候，永远使用一个“默认拒绝”的方法，来拒绝乱七八糟的请求。
服务层定义的安全性会更强壮并且难以绕过，所以你应该总是利用Spring Security的方法安全性选项。

11.5 使用其它基于过滤器的框架

如果你使用一些其它的基于过滤器的框架，你需要确保Spring Security的过滤器先做。这打开了对SecurityContextHolder的注入，使得它可以被其它过滤器使用。例子是使用SiteMesh来装饰你的网页或者像Wicket这种使用过滤器来处理请求的web框架。

11.6 高级命名空间配置

像我们在之前的命名空间章节看见的，可以使用多个http元素来对不同的url模式定义不同的安全配置。每个元素会在内部的FilterChainProxy创建一个过滤器链并且将相应的url模式匹配到之上。这些元素会按照它们声明顺序被添加，所以最特殊的模式应该放在第一个。**这里有另一个例子：与上面的情况类似，这个应用一边支持一个无状态的RESTful API，同时支持一个正常的web应用，使用表单登录：**

```
<!-- Stateless RESTful service using Basic authentication --><httppattern="/restful/**"create-session="stateless"><intercept-urlpattern='/**'access="hasRole('REMOTE')" /><http-basic /></http><!-- Empty filter chain for the login page --><httppattern="/login.htm*"security="none"/><!-- Additional filter chain for normal users, matching all other requests --><http><intercept-urlpattern='/**'access="hasRole('USER')" /><form-loginlogin-page='/login.htm'default-target-url="/home.htm"/><logout /></http>
```

12.核心安全过滤器

在使用Spring Security的web应用中，总是有几个核心过滤器总是会用到的，所以我们看一下它们及其支持类与接口。我们不会所有特性都研究，所以最好再去看一下javadoc，如果你想有一个全面了解的话。

12.1 FilterSecurityInterceptor

我们在讨论访问控制的时候，已经简单的见过FilterSecurityInterceptor，并且我们已经在命名空间的<intercept-url>元素中使用它了。现在我们会看看如何在使用FilterChainProxy的时候显式的配置和使用它，及其相关的过滤器ExceptionTranslationFilter。一种典型的配置方式如下：

```
<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="/secure/super/**" access="ROLE_WE_DONT_HAVE"/>
 <security:intercept-url pattern="/secure/**" access="ROLE_SUPERVISOR,ROLE_TELLER"/>
 </security:filter-security-metadata-source>
</property>
</bean>
```

FilterSecurityInterceptor对处理http资源的安全性负责。它需要一个到AuthenticationManager的引用和一个到AccessDecisionManager的引用。它也会被提供配置属性，来应用到不同的http url请求。具体请回头看技术介绍章节。
FilterSecurityInterceptor可以通过两种方法被配上配置属性。第一，如上面所述，使用<filter-security-metadata-source>命名空间元素。这个与<http>元素相似，但是<intercept-url>子元素只使用pattern和access属性。逗号用来分割应用到每个http url的不同的配置属性。第二种方法是你自己写SecurityMetadataSource，但是这超出了这篇文档的范畴。不论使用哪种方法，SecurityMetadataSource有责任返回一个List<ConfigAttribute>，包含所有的与受保护url有关的配置属性。
需要注意的是，FilterSecurityInterceptor.setSecurityMetadataSource()方法实际上期待的是一个FilterInvocationSecurityMetadataSource的实例。这是一个标记接口，其父类是SecurityMetadataSource。它简单的指出SecurityMetadataSource理解FilterInvocation。处于简单的目的，我们继续将FilterInvocationSecurityMetadataSource理解为一个SecurityMetadataSource，因为对大多数用户来说其中的区别很小。
命名空间的语法创建的SecurityMetadataSource获得针对一次特定的FilterInvocation的配置属性，通过匹配请求url及配置的url模式。这一行为与命名空间配置得到的类的行为一致。默认的是将所有的表达式视为Apache Ant路径，负责的情况下正则表达式也支持。request-matcher属性用来指定使用的匹配模式。但是不可以在一个定义中混合表达式语法。例如，下面的例子展示了使用正则表达式匹配：

```
<bean id="filterInvocationInterceptor"
 class="org.springframework.security.web.access.intercept.FilterSecurityInterceptor">
<property name="authenticationManager" ref="authenticationManager"/>
<property name="accessDecisionManager" ref="accessDecisionManager"/>
```

```
<property name="runAsManager" ref="runAsManager"/>
<property name="securityMetadataSource">
 <security:filter-security-metadata-source request-matcher="regex">
 <security:intercept-url pattern="\A/secure/super/.*\Z" access="ROLE_WE_DONT_HAVE"/>
 <security:intercept-url pattern="\A/secure/.*\" access="ROLE_SUPERVISOR,ROLE_TELLER"/>
 </security:filter-security-metadata-source>
</property>
</bean>
```

模式总是按照它们定义的顺序匹配。所以重要的是最特殊的模式需要定义在相对不特殊的模式之前。这在上面的例子中有所反应，更特殊的/secure/super/出现在相对不特殊的/secure/之前。如果他俩颠倒位置，/secure/模式会匹配上，而/secure/super/模式就永远也别想了。

12.2 ExceptionTranslationFilter

在Spring Security过滤器栈中，ExceptionTranslationFilter处于FilterSecurityInterceptor之前。它自己不做任何的实际的安全限制，但是处理安全拦截器抛出的异常，并且提供合适的http响应。

```
<bean id="exceptionTranslationFilter"
class="org.springframework.security.web.access.ExceptionTranslationFilter">
<property name="authenticationEntryPoint" ref="authenticationEntryPoint"/>
<property name="accessDeniedHandler" ref="accessDeniedHandler"/>
</bean>

<bean id="authenticationEntryPoint"
class="org.springframework.security.web.authentication.LoginUrlAuthenticationEntryPoint">
<property name="loginFormUrl" value="/login.jsp"/>
</bean>

<bean id="accessDeniedHandler"
 class="org.springframework.security.web.access.AccessDeniedHandlerImpl">
<property name="errorPage" value="/accessDenied.htm"/>
</bean>
```

12.2.1 AuthenticationEntryPoint

如果用户请求一个受保护的http资源但是他们没用被认证，AuthenticationEntryPoint会被调用。一个合适的AuthenticationException或者AccessDeniedException会被一个安全拦截器抛出，沿着调用栈，触发入口点的commence方法。其工作是为用户展现一个合适的响应，这样认证可以开始。我们这里使用的是LoginUrlAuthenticationEntryPoint，这将请求重定向到一个不同的url(通常是个登陆页)。实际的实现会依赖于你应用需要的认证机制。

12.2.2 AccessDeniedHandler

如果用户已经认证了，然后他们尝试去访问一个受保护的资源，那么会发生什么？正常使用时，这不应该发生，因为应用控制流应该被限制在用户已经访问的操作。例如，一个链接到一个管理页面的html应该对没有管理员角色的用户隐藏。你不能依靠于隐藏这个链接来实现安全性，因为总是有可能一个用户就巧了手动直接输入了一个url绕过了安全限制。或者他们修改一个RESTful的url来改变一些参数值。你的应用必须有能力防止这种场景，否则确实是不安全的。典型的你可以使用一些简单的web层安全来对基础url做限制，及在服务层使用更复杂的基于方法的安全性，来真正的决定什么是允许的。
如果一个AccessDeniedException抛出了，并且用户已经被认证了，那么这意味着他们尝试的操作不具有足够的权限。在这种情况下ExceptionTranslationFilter会调用第二策略——AccessDeniedHandler。默认的，使用AccessDeniedHandlerImpl，它向客户端发回一个403。或者你可以显式配置一个实例，设定一个错误页url，它可以将请求跳转过去。这可以是一个简单的“访问拒绝”页面，例如一个jsp，或者可能是一个更复杂的处理器。当然你可以实现其接口写你自己的逻辑。
当你使用命名空间的配置的时候，当然也是可以提供一个个性化的AccessDeniedHandler的。

12.2.3 SavedRequest和RequestCache接口

ExceptionTranslationFilter的另外一个职责是在调用AuthenticationEntryPoint之前保存当前的请求。这允许请求在用户在被认证之后保存请求。一个典型的例子是用户使用表单登录，然后重定向到原始url的时候，默认通过SavedRequestAwareAuthenticationSuccessHandler。
RequestCache封装了存储和获取HttpServletRequest实例的功能。默认的，HttpSessionRequestCache被使用，它将请求存到HttpSession中。
RequestCacheFilter的职责是当用户重定向的时候从缓存中恢复保存的请求。
在常见的环境下，你不应该需要修改这些功能，但是对已经保存的请求的处理是一个“尽最大努力”的方法，因为会有一些情况下，默认的配置不能处理。这些接口的使用从Spring Security3之后都是完全可插拔的。

12.3 SecurityContextPersistenceFilter

我们已经在技术概览章节讲过这个最重要的过滤器，因此你或许想要回头看一眼。让我们先看看你应该怎么配置它，基础的配置只需要它自己：

```
<bean id="securityContextPersistenceFilter"
class="org.springframework.security.web.context.SecurityContextPersistenceFilter"/>
```

像我们之前看见的，这个过滤器有两个主要任务。它负责在http请求之间存储SecurityContext的内容，和在请求完成的时候清空SecurityContextHolder。清理context存储的ThreadLocal是必要的，否则如果某个（servlet容器线程池中的）线程被另一次请求拿到了，但是你context没清，那就成了第二次请求的用户有了第一次请求用户的身份。就会有错误的权限问题。

12.3.1 SecurityContextRepository

从Spring Security3之后，加载和存储安全上下文代理到了一个单独的策略接口：

```
public interface SecurityContextRepository {

SecurityContext loadContext(HttpRequestResponseHolder requestResponseHolder);

void saveContext(SecurityContext context, HttpServletRequest request,
  HttpServletResponse response);
}
```

HttpRequestResponseHolder是个简单的容器，面向进入的请求和响应对象，允许其包装类的使用。返回内容会被传递到过滤器链。
默认的实现是HttpSessionSecurityContextRepository，它将安全上下文作为一个HttpSession的属性存储。对这个实现最重要的配置参数是allowSessionCreation，默认是true，这允许它发现对于已认证用户需要创建一个session来存储安全上下文的时候可以创建之（如果不是认证发生，并且安全上下文改变，它不会创建）。如果你不需要创建session，则设置这个属性为false。

```
<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>
```

可选的，你可以提供一个NullSecurityContextRepository实例，它会阻止安全上下文存储，即使session已经创建。

12.4 UsernamePasswordAuthenticationFilter

我们现在已经看见了Spring Security web配置中永远出现的三个主要过滤器，他们也是在命名空间配置的<http>标签中自动创建并且不能被替换的三个。现在唯一没提到的东西是实际的认证机制，那个允许用户认证的东西。这个过滤器是最常用的认证过滤器并且也是最常被个性化定制的一个。它也通过命名空间配置的<form-login>提供默认实现。配置它需要经过三步：
1、配置一个带有登录页url的LoginUrlAuthenticationEntryPoint，将它设定到ExceptionTranslationFilter
2、实现登陆页
3、在应用上下文中配置一个UsernamePasswordAuthenticationFilter实例
4、将过滤器bean添加到你的过滤器链中，注意顺序。

登录表单简单的包含username和password字段，然后post到过滤器链监控的url（默认/login），基础配置如下：

```
<bean id="authenticationFilter" class=
"org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter">
<property name="authenticationManager" ref="authenticationManager"/>
</bean>
```

12.4.1 认证成功和失败的应用流

过滤器调用配置的AuthenticationManager来处理每个认证请求。紧跟着认证成功或者认证失败的目的地被AuthenticationSuccessHandler和AuthenticationFailureHandler策略接口控制。过滤器有能让你配置这些处理器的字段。框架提供的一些标准实现，例如：SimpleUrlAuthenticationSuccessHandler, SavedRequestAwareAuthenticationSuccessHandler, SimpleUrlAuthenticationFailureHandler和ExceptionMappingAuthenticationFailureHandler. 详情参见 Javadoc，也看看 AbstractAuthenticationProcessingFilter，体会一下他们是怎么工作的。
如果认证成功，返回的Authentication对象会被放在SecurityContextHolder，配置的AuthenticationSuccessHandler然后会被调用，要么重定向，要么跳转到何时的目的。默认的使用SavedRequestAwareAuthenticationSuccessHandler，这意味着用户会重定向到他们在被要求登录之前的目的页。

ExceptionTranslationFilter缓存用户做的原始请求。当用户认证时，请求处理器拿这个缓存的请求来读取原始url然后重定向过去，原始请求然后被重建作为一个替代。

如果认证失败，配置的AuthenticationFailureHandler被调用。

21.权限架构

21.1 权限

像我们在技术概览中见到的，所有认证实现都会保存一个GrantedAuthority对象的列表。这些代表授予安全主体的权限。GrantedAuthority对象被AuthenticationManager插入到Authentication对象，然后在做权限判断的时候被AccessDecisionManager读取。
GrantedAuthority是一个只有一个方法的接口：

```
String getAuthority();
```

这个方法允许AccessDecisionManager获取一个精确的String来代表GrantedAuthority。通过这个返回的String，GrantedAuthority可以被AccessDecisionManager简单的读取。如果一个GrantedAuthority不能被简洁的通过一个String来代表，则GrantedAuthority被认为是个复杂对象，然后getAuthority()方法必须返回null。
作为一个复杂GrantedAuthority的例子，其实现可能是保存了一个操作列表和权限阈值，应用到不同的用户账户上。将这种复杂的GrantedAuthority表示为一个String是非常困难的，所以getAuthority()方法应该返回null。这暗示着任何需要支持这种GrantedAuthority的AccessDecisionManager都需要有能力理解其内容。Spring Security包含一个具体的GrantedAuthority实现，GrantedAuthorityImpl。这允许任何用户指定的String能被转换到一个GrantedAuthority。所有的框架内部的AuthenticationProvider都使用GrantedAuthorityImpl来灌注Authentication对象。

21.2 调用前处理

像我们在技术概览中见到的，Spring Security提供控制对受保护对象（例如方法调用或web请求）进行访问的控制器。AccessDecisionManager会做出一个调用

前决策来决定这次调用是不是允许进行。

21.2.1 AccessDecisionManager

AccessDecisionManager被AbstractSecurityInterceptor调用，并且对做出最后的访问控制负责。AccessDecisionManager接口包含了三个方法：

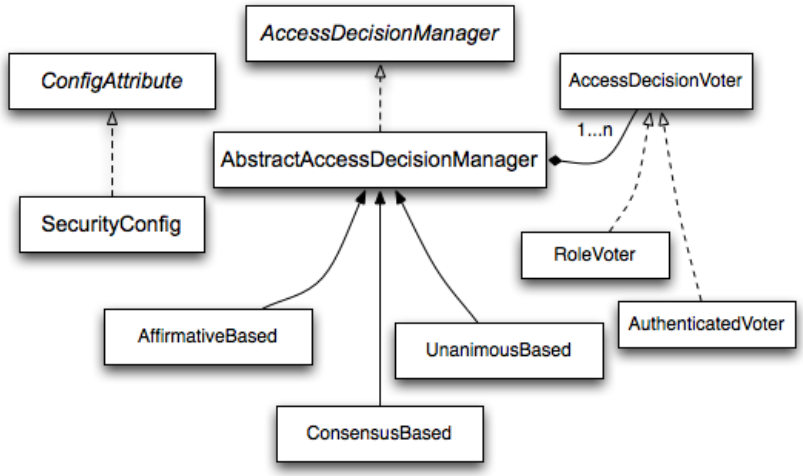
```
void decide(Authentication authentication, Object secureObject,
 Collection<ConfigAttribute> attrs) throws AccessDeniedException;

boolean supports(ConfigAttribute attribute);

boolean supports(Class clazz);
```

AccessDecisionManager的decide方法接受所有的相关信息，来做出权限的决定。特别的， 将受保护的Object传递进去，使得实际的受保护对象调用所包含的方法参数被检查。例如：我们假设受保护对象是个“方法调用”，从MethodInvocation中查询任何Customer参数是很容易的，然后在AccessDecisionManager中实现一些安全逻辑来确保安全主体允许在那个Customer上执行。如果访问被拒绝，方法期望抛出一个AccessDecisionException。
support(ConfigAttribute)方法被AbstractSecurityInterceptor在启动时调用，来决定是否AccessDecisionManager可以处理传入的ConfigAttribute。
support(Class)方法被一个安全拦截器实现调用，来确保AccessDecisionManager支持安全拦截器提供的受保护对象类型。

21.2.2 基于投票的AccessDecisionManager实现

同时用户可以实现他们自己的AccessDecisionManager来控制权限的所有方面。Spring Security包含多个基于投票的AccessDecisionManager的实现。下图展示了相关的实现类。

Figure 21.1. Voting Decision Manager



使用这个方法，一系列的AccessDecisionVoter会对一次授权决定进行投票。AccessDecisionManager然后依据唱票的结果来做出决定是不是要抛出一个AccessDecisionException。
AccessDecisionVoter接口有三个方法：

```
int vote(Authentication authentication, Object object, Collection<ConfigAttribute> attrs);

boolean supports(ConfigAttribute attribute);

boolean supports(Class clazz);
```

vote的实现需要返回一个int值，可能的取值在AccessDecisionVoter的静态域中，为ACCESS_ABSTAIN，ACCESS_DENIED，ACCESS_GRANTED。一个投票的实现会在它不知道针对一次认证决策投什么票的时候返回ACCESS_ABSTAIN。如果它知道，那么要么投ACCESS_DENIED，要么投ACCESS_GRANTED。
Spring Security提供三个固定的AccessDecisionManager的实现来记录选票。ConsensusBased的实现会基于非弃权票的统计来授权或者拒绝访问。有一些属性可以配置来处理如果同意票和反对票相同的情况，或者所有票都弃权的情况。AffirmativeBased的实现会在只要有同意票的时候就对访问进行授权（拒绝票都会被忽略，只要有同意票就授权）。同ConsensusBased一样，也有属性可以配置当所有的票都是弃权票的时候如何处理。UnanimousBased的实现期待全体票都是ACCESS_GRANTED的时候才会对访问授权，忽略弃权票，如果票中有拒绝票，就会直接拒绝访问。同样的，它也有参数控制所有票都弃权的情况。
也可以实现自己的AccessDecisionManager采用不同的唱票方法。例如从特定的AccessDecisionVoter来的投票会加上一个额外的权重，而从特定的投票者来的选票具有一票否决权等等。

RoleVoter

Spring Security提供的最常使用的AccessDecisionVoter是最简单的RoleVoter，它将配置属性看成是简单的角色名，如果用户有相关的角色它就投赞成票。
它会在任意ConfigAttribute以ROLE_开头的时候投票。它会在有一个GrantedAuthority返回的代表角色的String与一个或者多个ROLE_开头的ConfigAttribute匹配上的时候投授权票。如果没有任何ROLE_开头的ConfigAttribute精确匹配，RoleVoter会拒绝。如果压根没有ROLE_开头的ConfigAttribute，它就会投弃权票。

AuthenticatedVoter

另一个我们通常不大能看见的投票器是AuthenticatedVoter，这可以用来区别匿名用户，完全认证的用户，和记住我认证的用户。许多站点允许在记住我认证的情况下的进行一些有具体限制的访问，但是需要用户确认他们的身份才能进行完全访问。
当我们使用IS_AUTHENTICATED_ANONYMOUSLY属性来授权匿名访问时，这个属性被AuthenticatedVoter处理，详情参见javadoc。

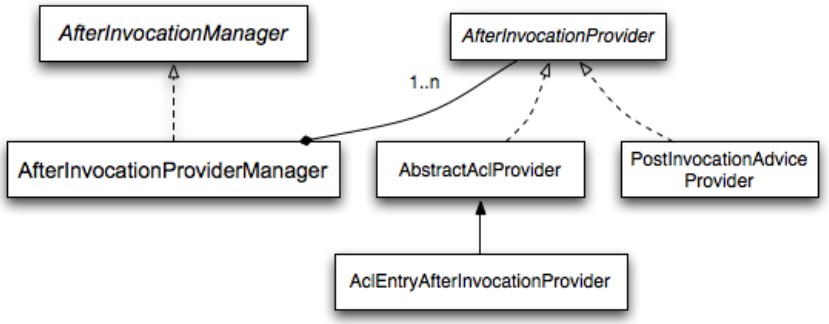
Custom Voters

明显的，你也可以实现一个个性化的AccessDecisionVoter，然后你可以将你需要的访问控制逻辑加进去。这或许是对你的应用特别定制的（关于业务逻辑）或者可能实现了一些安全管理逻辑。例如，你会在SpringSource上面找到一篇博客文章，描述了如何使用投票器来实时拒绝被挂起的用户的访问。

21.3 调用后处理

当在对受保护对象的调用处理之前，AccessDecisionManager被AbstractSecurityInterceptor调用的时候，一些应用需要一种方法来修改受保护的对象的方法调用的真实返回值。你可以简单的实现你自己的AOP来做这事，Spring Security也提供了一个便利的钩子方法，也有一些内部的实现，整合了ACL的能力。
图21.2展示了Spring Security的AfterInvocationManager及其固定的实现：

Figure 21.2. After Invocation Implementation



类似于Spring Security的许多其它部分，AfterInvocationManager有一个单独的实现，AfterInvocationProviderManager，它选举一个AfterInvocationProvider的列表。每个AfterInvocationProvider都允许修改返回对象或者抛出一个AccessDeniedException。确实多个提供器都能修改返回对象，因为之前的提供器的处理结果会传递给下一个提供器。
请警惕如果你使用AfterInvocationManager，你仍然需要配置属性来允许MethodSecurityInterceptor的AccessDecisionManager去允许一个处理。如果你使用典型的Spring Security内部的AccessDecisionManager实现，对于某个特定的受保护方法调用没有配置属性的定义，会导致每个AccessDecisionVoter弃权。相应的，如果AccessDecisionManager属性allowIfAllAbstainDecision都是false，会抛出AccessDecisionException。你可以避免这种潜在的问题(i)要么设置allowIfAllAbstainDecision为true（尽管这个大体上是不推荐的），(ii)要么简单的确保至少有一个配置属性，使得AccessDecisionVoter会投赞成票。后者（推荐的）是经常使用的，通过配置ROLE_USER或者ROLE_AUTHENTICATED配置属性。

21.4 角色等级

在某个应用中，一个特定的角色自动的“包含”另外的角色这是个常见的需求。例如，在一个具有“管理员”和“用户”角色的应用中，你或许想要一个管理员能做任何用户能做的事情。为了达到这一目的，你可以保证所有的管理员同时被分配了用户角色。或者，你可以修改每个用户角色访问限制也包括管理员角色。如果你的应用有很多角色，这事其实挺复杂的。
角色等级的使用，允许你配置某个角色（或者权限）包含其它的。一个Spring Security的RoleVoter的扩展类，RoleHierarchyVoter，配置了一个RoleHierarchy，从这里它获取所有的用户“可达权限”。一种典型配置如下：

```
<beanid="roleVoter"class="org.springframework.security.access.vote.RoleHierarchyVoter"><constructor-argre
f="roleHierarchy" /></bean><beanid="roleHierarchy"class="org.springframework.security.access.hierarchical
roles.RoleHierarchyImpl"><propertyname="hierarchy"><value>
   ROLE_ADMIN > ROLE_STAFF
   ROLE_STAFF > ROLE_USER
   ROLE_USER > ROLE_GUEST
  </value></property></bean>
```

这里我们在等级表中有四个角色，ROLE_ADMIN ⇒ ROLE_STAFF ⇒ ROLE_USER ⇒ ROLE_GUEST。被认证为ROLE_ADMIN的用户，在使用配置了上面的RoleHierarchyVoter的AccessDecisionManager进行安全限制的评估的时候，会具有所有四个角色的权限。>符号可以认为是“包括”的含义。
角色阶级提供了一种方便的方式，来简化应用中的访问配置数据，减少分配给用户的权限的数量。对于更多复杂的需求，你可以在你应用需要的指定的访问权限与分配给用户的角色之间定义一个逻辑映射，在需要使用的时候相应进行转义。

22.受保护的对象的实现

22.1 AOP联盟（MethodInvocation）安全拦截器

在Spring Security2之前，保护MethodInvocation需要一大面配置。现在对于方法安全的推荐方式是使用命名空间配置。这种方式下，方法安全相关的bean为你自动配置好，所以你不需要知道具体实现类。我们在这提供一个快速概览。
方法安全强制使用MethodSecurityInterceptor，它保护MethodInvocation。依赖配置的方法，一个拦截器被指定到一个单独的bean或者在多个bean之间共享。拦截器使用一个MethodSecurityMetadataSource实例来获取应用到具体方法调用上的配置属性。MapBasedMethodSecurityMetadataSource通过将方法名（可以支持通配符）作为键来存储配置属性，当属性在应用上下文中的<intercept-mothod>或者<protect-point>元素中定义时被内部调用。其他的实现用来处理基于注解的配置。

22.1.1 显式MethodSecurityInterceptor配置

你当然可以直接在你的应用上下文中配置MethodSecurityInterceptor，使用Spring的AOP代理机制：

```
<bean id="bankManagerSecurity" class=
 "org.springframework.security.access.intercept.aopalliance.MethodSecurityInterceptor">
<property name="authenticationManager" ref="authenticationManager"/>
<property name="accessDecisionManager" ref="accessDecisionManager"/>
<property name="afterInvocationManager" ref="afterInvocationManager"/>
<property name="securityMetadataSource">
 <sec:method-security-metadata-source>
 <sec:protect method="com.mycompany.BankManager.delete*" access="ROLE_SUPERVISOR"/>
 <sec:protect method="com.mycompany.BankManager.getBalance" access="ROLE_TELLER,ROLE_SUPERVISOR"/>
 </sec:method-security-metadata-source>
</property>
</bean>
```

22.1 AspectJ（JoinPoint）安全拦截器

AspectJ安全拦截器与AOP联盟安全拦截器非常相似，因此我们这一章只讨论区别。
AspectJ拦截器命名AspectJSecurityInterceptor，不像AOP联盟安全拦截器依赖Spring应用上下文通过代理来织入安全拦截器，AspectJSecurityInterceptor通过AspectJ编译器织入。在同一个应用中同时使用这两种类型的安全拦截器不算罕见，AspectJSecurityInterceptor用在领域对象实例的安全上，AOP联盟的MethodSecurityInterceptor用在服务层的安全上。
让我们先考虑AspectJSecurityInterceptor的配置：

```
<bean id="bankManagerSecurity" class=
 "org.springframework.security.access.intercept.aspectj.AspectJMethodSecurityInterceptor">
<property name="authenticationManager" ref="authenticationManager"/>
<property name="accessDecisionManager" ref="accessDecisionManager"/>
<property name="afterInvocationManager" ref="afterInvocationManager"/>
<property name="securityMetadataSource">
 <sec:method-security-metadata-source>
 <sec:protect method="com.mycompany.BankManager.delete*" access="ROLE_SUPERVISOR"/>
 <sec:protect method="com.mycompany.BankManager.getBalance" access="ROLE_TELLER,ROLE_SUPERVISOR"/>
 </sec:method-security-metadata-source>
</property>
</bean>
```

像你看见的，除了类名，AspectJSecurityInterceptor与AOP联盟安全拦截器完全一样。确实两者可以共享同一个securityMetadataSource，因为SecurityMetadataSource用java.lang.reflect.Method工作，而不是AOP库指定的类。当然，你的访问决策可以去调用AOP库指定的方法（例如，MethodInvocation或者JoinPoint），这样可以在做出访问决策的时候考虑额外的准则。
然后你需要定义AspectJ的aspect，例如：

```
package org.springframework.security.samples.aspectj;

import org.springframework.security.access.intercept.aspectj.AspectJSecurityInterceptor;
import org.springframework.security.access.intercept.aspectj.AspectJCallback;
import org.springframework.beans.factory.InitializingBean;

public aspect DomainObjectInstanceSecurityAspect implements InitializingBean {

 private AspectJSecurityInterceptor securityInterceptor;

 pointcut domainObjectInstanceExecution(): target(PersistableEntity)
  && execution(public * *(..)) && !within(DomainObjectInstanceSecurityAspect);

 Object around(): domainObjectInstanceExecution() {
  if (this.securityInterceptor == null) {
   return proceed();
  }

  AspectJCallback callback = new AspectJCallback() {
   public Object proceedWithObject() {
    return proceed();
   }
  };

  return this.securityInterceptor.invoke(thisJoinPoint, callback);
 }

 public AspectJSecurityInterceptor getSecurityInterceptor() {
  return securityInterceptor;
 }

 public void setSecurityInterceptor(AspectJSecurityInterceptor securityInterceptor) {
  this.securityInterceptor = securityInterceptor;
 }

 public void afterPropertiesSet() throws Exception {
  if (this.securityInterceptor == null)
   throw new IllegalArgumentException("securityInterceptor required");
  }
 }
```

```
}
```

在上面的例子，安全拦截器会应用到每个PersistableEntity实例，它是一个没有展示的抽象类（你也可以使用其它类或者pointcut表达式）。对于那些好奇的人，AspectJCallback是需要的，因为proceed()句法只有在around（）体中才有特殊含义。AspectJSecurityInterceptor调用这个匿名的AspectJCallback类，当它想要目标对象继续的时候。
你会需要配置Spring来加载方面，然后绑定到AspectJSecurityInterceptor上。一种声明方法可以实现之：

```
<bean id="domainObjectInstanceSecurityAspect"
 class="security.samples.aspectj.DomainObjectInstanceSecurityAspect"
 factory-method="aspectOf">
<property name="securityInterceptor" ref="bankManagerSecurity"/>
</bean>
```

这就对了！现在你可以在你的应用内部任何地方创建你的bean，使用你想要的任意方式（如，new Person();），他们都会被应用上安全拦截器。