阅读笔记9

第九章 保护Spring应用

首先要说 这本书讲这个Spring security的是很不全面的 靠这本书是完全没有办法搭建起来的
http://wiki.jikexueyuan.com/project/spring-security/ 这里有俩个比较详细的教程
http://blog.csdn.net/u012367513/article/details/38866465一个相对深入一点
作为软件开发人员 我们必须采取措施来保护应用程序中的信息 安全性是绝大部分应用系统中的一个重要切面

安全性是超越应用程序功能的一个关注点 应用程序绝大部分不应该参与到自己相关的安全性处理中 尽管可以直接在应用程序中编写安全性功能相关的代码 但是最好是安全性的关注点与应用程序分离

Spring security
在刚开始第一个版本Spring security被称为Acegi Security
要配置相当多的东西 相当的麻烦
现在 第二个版本改名为Spring security 大大了改进了XML的配置
而且还融入了Sqel
Spring security从俩个角度解决安全性问题 使用Servlet过滤器保护web请求并限制URL级别的范围 也可以使用Spring AOP保护方法调用 借助于对象代理和使用通知 能够确保只有具有适当权限的用户才能访问安全保护的方法

Spring 模块添加 Spring security一共有8个模块
在应用程序类下至上要保护核心和配置这俩个模块 Spring security经常被用于保护Web应用

模块	描 述
ACL	支持通过访问控制列表为域对象提供安全性
CAS 客户端	提供与 JA-SIG 的中心认证服务（CAS，Central Authentication Service）进行集成的功能
配置	包含了对 Spring Security XML 命名空间的支持
核心	提供了 Spring Security 基本库
LDAP	支持基于轻量目录访问协议（Lightweight Directory Access Protocol）进行认证
OpenID	支持分散式 OpenID 标准
Tag Library	包含了一组 JSP 标签来实现视图级别的安全性
Web	提供了 Spring Security 基于过滤器的 Web 安全性支持

使用Spring security配置命名空间
在以前Spring security还是Acegi security的时候 所有的安全性元素都需要在Spring应用上下文用<Bean>来声明
在Spring security中大大的简化了
下面是Spring security的命名空间

```
<?xml version="1.0" encoding="UTF-8"?>

<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.xsd

        http://www.springframework.org/schema/security

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

保护WEB请求
从JAVA web应用所做的任何事情都是从HttpServletRequest开始的
如果说请求是web应用的入口的话 那这也是web应用的安全性起始的位置

对于请求级别的安全性来说 最基本的形式涉及声明一个或多个URL模式 并要求具备一定级别权限的用户才能进行访问 并阻止无这些权限的用户反问这些URL背后的内容 进一步来讲 还有可能要求只能通过HTTPS访问特定的URL

在限制只有具备一定权限的用户进行访问之前 我们必须要有一种方式来判断是谁在使用应用程序 所以 应用程序需要对用户进行认证 提醒用户进行登录并要求进行身份验证

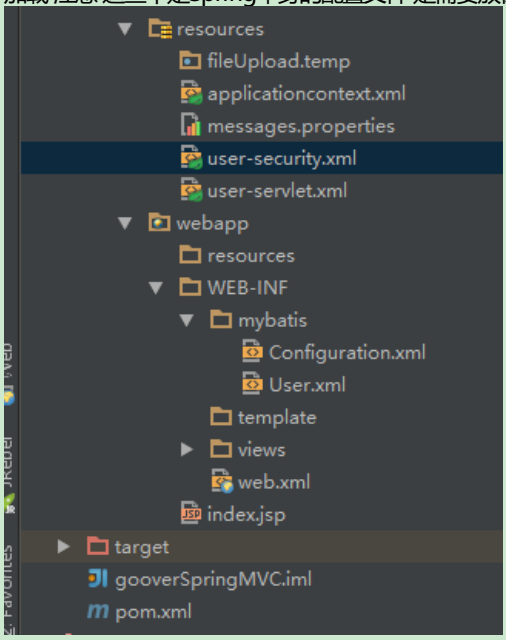
Spring security支持以上这些安全性以及其他多种方式的请求级别安全性
但是在这之前 首先要构建提供各种安全特性的Servlet的过滤器

代理Servlet过滤器
Spring security借助一系列Servlet过滤器来提供各种安全性功能
不需要配置很多的过滤器 只需要配置一个特殊的过滤器来完成各种功能

这里我插一句 关于与其他的框架进行集成的时候 需要注意 必须只能配置一个配置文件 也就是security的配置文件
至于其他的配置 如果不是Spring本身的 则通过spring的的上下文配置文件加载 而这个上下文配置文件则通过

```
<beans:import resource="applicationcontext.xml"/>
```

加载 注意 这些不是Spring本身的配置文件 是需要放在web应用程序下的 然后进行加载 比如mybatis 比如我这个




```
<filter>

    <filter-name>springSecurityFilterChain</filter-name>

    <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>

</filter>
```

DelegatingFilterProxy是一个特殊的Servlet过滤器
这个过滤器只是将工作委托给一个javax.servlet.Filter实现类 这个实现类作为一个<bean >注册在Spring应用的上下文

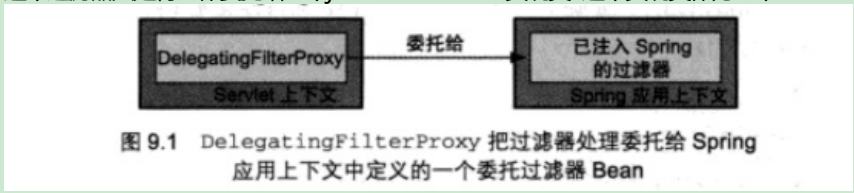


图 9.1 DelegatingFilterProxy 把过滤器处理委托给 Spring 应用上下文中定义的一个委托过滤器 Bean

为了完成各自的工作 Spring security的过滤器必须注入一些其他的Bean
我们无法对注册在web.xml中的servlet过滤器进行Bean注入 但是 通过使用DelegatingFileterProxy 我们可以在Spring中配置实际的过滤器 从而能够充分使用Spring对依赖注入的支持

DelegatingFilterProxy的<filter-name>值是有意义的 这个名字用于在Spring应用上下文中查找过滤器Bean Spring security将自动创建一个ID为springSecurityFilterChain的过滤器Bean 这就是我们在web.xml中为DelegatingFilterProxy所设置的值

springSecurityFilterChain 本身是另一个特殊的过滤器，它也被称为 FilterChainProxy。它可以链接任意一个或多个其他的过滤器。Spring Security 依赖一系列 Servlet 过滤器来提供不同的安全特性。但是，你几乎不需要知道这些细节，因为你不需要显式声明 springSecurityFilterChain 以及它所链接在一起的其他过滤器。当配置 <http> 元素时，Spring Security 将会为我们自动创建这些 Bean，接下来我们将会介绍这部分内容。

配置最小化的web安全性

```
<http auto-config="true">

    <intercept-url pattern="/*" access="ROLE_SPITTER"/>

</http>
```

这3行简单的XML就可以配置Spring Security拦截所有URL请求（通过使用Ant风格的路径来声明<intercept-url>的pattern属性），并限制只有拥有ROLE_SPITTER角色的认证用户才能访问 <http>元素将会创建一个FilterChainProxy(它会委托给配置在web.xml中的DelegatingFilter-Proxy)以及链中的所有过滤器Bean

除了这些过滤器 Bean，通过将 auto-config 属性配置为 true，我们还可以得到一些其他的“免费赠品”。自动配置会为我们的应用提供一个额外的登录页、HTTP 基本认证和退出功能。实际上，将 auto-config 属性配置为 true 等价于下面这样显式配置的特性：

```
<http>
    <form-login />
    <http-basic />
    <logout />
  <intercept-url pattern="/**" access="ROLE_SPITTER" />
</http>
```

让我们深入了解这些功能都为我们带来了什么以及如何使用它们。

通过表单进行登录

将 auto-config 属性配置为 true 的一个好处就是，Spring Security 将会自动为你生成登录页面。程序清单 9.3 给出了这个表单的 HTML 代码。

程序清单 9.3 Spring Security 能够为你自动生成一个登录表单

```
<html>
  <head><title>Login Page</title></head>
  <body onload='document.f.j_username.focus();'>
    <h3>Login with Username and Password</h3>
    <form name='f' method='POST'
          action='/Spitter/j_spring_security_check'>          <- 认证过滤器的路径
      <table>
        <tr><td>User:</td><td>
            <input type='text' name='j_username' value=''>    <- 用户名输入域
        </td></tr>
        <tr><td>Password:</td><td>
            <input type='password' name='j_password'/>        <- 密码输入域
        </td></tr>
        <tr><td colspan='2'><input name="submit" type="submit"/></td></tr>
        <tr><td colspan='2'><input name="reset" type="reset"/></td></tr>
      </table>
    </form>
  </body>
</html>
```

可以通过相对于应用文上下文URL的/spring_security_login路径来访问这个自动生成的表单
可以设置自己的登录页 我们需要配置<from-login>元素来取代默认的行为

```
<http auto-config="true" use-expressions="false">

    <form-login login-processing-url="/static/j_spring_security_check"

login-page="/login" authentication-failure-url="/login?login_error=t"/>
```

```
</http>
```

login-page属性为登录页声明了一个新的且相对于上下文的URL 这个示例中 我们登录页声明为/login **这个最终是由一个Spring MVC控制器进行处理** 同样 如果认证失败 通过设置**authentication-failyre-url**属性 就会把用户重定向到相同的登录页
需要注意的是： 我们将 login-processing-url属性设置为/static/j_spring_scurity_check 这是登录表单提交回来进行用户认证的URL （ Spring security是在/static/j_spring_scurity_check路径下处理登录请求 而且很显然 用户名和密码需要在请求中使用名为j_username和j_password的输入域进行提交 ）

关于这个/login 在spring MVC中我们进行对这个/login的匹配 对应一个新的JSP

```
<?xml version="1.0" encoding="UTF-8"?>

<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.xsd

        http://www.springframework.org/schema/security

        http://www.springframework.org/schema/security/spring-security.xsd">


<!--    <http use-expressions="false">

              &lt;!&ndash; 访问所有页面都需要有USER权限 &ndash;&gt;

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

              &lt;!&ndash; 登录功能 &ndash;&gt;

              <form-login />

              &lt;!&ndash; 登出功能 &ndash;&gt;

              <logout />

       </http>

-->

<authentication-manager>

              <authentication-provider>

                     <user-service>

<!-- 这里创建两个用户，可以通过用户名密码登录 -->

<user name="jimi" password="jimispassword" authorities="ROLE_USER, ROLE_ADMIN" />

                           <user name="bob" password="bobspassword" authorities="ROLE_USER"

/>

                     </user-service>

              </authentication-provider>

       </authentication-manager>



      <http auto-config="true" use-expressions="false">
```

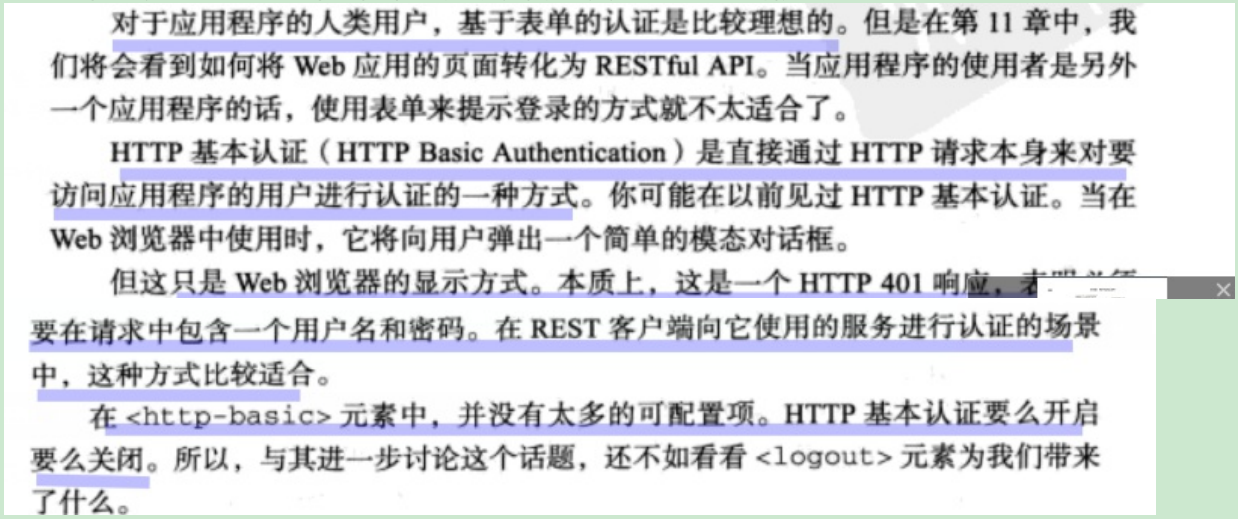
```
        <form-login login-processing-url="/static/j_spring_security_check"
login-page="/login" authentication-failure-url="/login?login_error=t"/>
      </http>


</beans:beans>
```

这就是我们设置的JSP
尽管登录页与Spring Security提供的内置页面有所不同 但是关键都是通过表单来提交包含用户凭证信息的j_username，j_password

处理基本认证
对于应用程序的用户 **基于表单的认证是比较理想的** 其实还有另外一种方式 这就是将WEB转换为RESTful API
当应用程序的使用者是另外一个应用程序的话 使用表单来提示登录的方式就不太合适了

HTTP基本认证是通过HTTP请求本身来对要求访问应用程序的用户进行认证的一种方式



对于应用程序的人类用户，基于表单的认证是比较理想的。但是在第 11 章中，我们将会看到如何将 Web 应用的页面转化为 RESTful API。当应用程序的使用者是另外一个应用程序的话，使用表单来提示登录的方式就不太适合了。

HTTP 基本认证（HTTP Basic Authentication）是直接通过 HTTP 请求本身来对要访问应用程序的用户进行认证的一种方式。你可能在以前见过 HTTP 基本认证。当在 Web 浏览器中使用时，它将向用户弹出一个简单的模态对话框。

但这只是 Web 浏览器的显示方式。本质上，这是一个 HTTP 401 响应，[illegible] 要在请求中包含一个用户名和密码。在 REST 客户端向它使用的服务进行认证的场景中，这种方式比较适合。

在 <http-basic> 元素中，并没有太多的可配置项。HTTP 基本认证要么开启要么关闭。所以，与其进一步讨论这个话题，还不如看看 <logout> 元素为我们带来了什么。

退出 springsecurity
<logout>元素会构建一个Spring Security过滤器 **这个过滤器用于使用户的会话失效** 在使用
<logout> 构建起来的过滤器将匹配 /j_spring_security_logout地址 但这与我们已经构建的DispatcherServlet并不冲突 我们需要像登录表单那样重写这个过滤器的URL 为了做到这一点 需要设置logout-url特性 <logout logout-url="/static/j_spring_security_logout"/>
关于退出登录在开头的网页链接中有说

拦截请求
<interecpt-url>元素是实现请求级别安全游戏的第一道防线 它的pattern属性定义了对传入请求要进行匹配的URL模式 如果请求匹配这个模式的话
<intercept-url>的安全规则就会启用

```
<intercept-url pattern="/**" access="ROLR_SPITTER"/>
```

pattern属性是默认的使用Ant风格的路径 但是如果你愿意的话 可以将<http>元素的path-type 属性设置为regex,pattern属性就可以使用正则表达式了
在这个例子中 我们将pattern属性设置为/** 意味着所有的请求不管其URL是什么 都需要具备ROLE_SPITTER角色才能进行访问 /**是一个很宽泛的配置 但可以更具体化
假设Spitter应用程序中的一些特定区域 只有管理用户才能访问 为了实现这一点 我们可以在已有的那条记录以下插入以下的 <intercept-url>

```
<intercept-url pattern="/admin/**" access="ROLE_ADMIN"/>
```

第一条是保证应用程序的大部分内容需要ROLE_SPITTER权限才能访问 这条<interecpt-url>限制这个站点的/admin分支只能由具有ROLE_ADMIN
可以使用任意数量的<interceot-url>条目来保护web应用程序中的各种路径 但是**比较重要的一点是<intercept-url>的规则是从上往下使用的** 所以这个新的<intercept-url>应该放在原有目录之前 否则它会因为前面更宽泛的/**路径范围失去作用

使用Spring表达式进行安全保护
列出的权限很简单 但这却显得有些功能单一 如果表达的不仅仅是权限的安全性通知
我们是可以使用SpEL表达式的 只**需要将<http>use-expressions属性设置为true**

```
<http auto-config="true" use-expressions="true">
```

这样就可以了 下面是使用SpEL表达式来声明这个需要ROLE_ADMIN权限才能访问的

```
<intercept-url pattern="/admin/**" access="hasRole('ROLE_ADMIN')"/>
```

如果用户得到了ROLE_ADMIN的权限 这个hasRole方法就返回一个true
下面是spring security支持的SpEL表达式

安全表达式	计算结果
authentication	用户的认证对象
denyAll	结果始终为 false
hasAnyRole(list of roles)	如果用户被授予了指定的任意权限，结果为 true
hasRole(role)	如果用户被授予了指定的权限，结果为 true
hasIpAddress(IP Address)	用户的 IP 地址（只能用在 Web 安全性中）
isAnonymous()	如果当前用户为匿名用户，结果为 true
isAuthenticated()	如果当前用户不是匿名用户，结果为 true
isFullyAuthenticated()	如果当前用户不是匿名用户也不是 remember-me 认证的，结果为 true
isRememberMe()	如果当前用户是通过 remember-me 自动认证的，结果为 true
permitAll	结果始终为 true
principal	用户的主要信息对象

还可以这样写 指定IP地址

```
<intercept-url  pattern="/admin/**" access="hasRole('ROLE_ADMIN') and hasIpAddress('192.168.1
.1')"/>
```

强制请求使用HTTPS

强制通道的安全性
使用Https很容易因为太多就忘记了添加这个关键的S 导致出错
我们可以将这个任务转移给<intercept-url>元素 就不会出错了

```
<intercept-url pattern="/spitter/form" requires-channel="https"/>
```

```
<intercept-url pattern="/home" requires-channel="http"/>
```

不管何时 只有对/spitter/form的请求 Spring security都视为HTTPS通道并自动将请求重定向到HTTPS
对首页的请求只需要http就好了
这样就可以看到如果在请求时保护web应用程序的安全了 **安全性的基本假设一直避免用户访问他没有权限访问的URL** 隐藏用户不能访问的链接也是个很好的办法

保护视图级别的元素

这里其实是缺一些东西的 比如开启匿名登录 然后才能使用匿名登录

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

比如退出

```
<logout  logout-url="/test/logout.do"/>
```

为了支持视图级别的安全性 Spring security提供了一个JSP标签库

表 9.3 Spring Security 通过 JSP 标签库在视图层支持安全性

JSP 标签	作用
<security:accesscontrollist>	如果当前认证用户对特定的域对象具备某一指定的权限，则渲染标签主体中的内容
<security:authentication>	访问当前用户认证对象的属性
<security:authorize>	如果特定的安全性限制满足的话，则渲染标签主体中的内容

要使用JSP标签库 (注意 首先是要添加一个包 关于这个包的maven支持如下 <!--

https://mvnrepository.com/artifact/org.springframework.security/spring-security-taglibs -->
<dependency>
<groupId>org.springframework.security</groupId>
<artifactId>spring-security-taglibs</artifactId>
<version>3.1.4.RELEASE</version>
</dependency>
)
需要在对应的JSP中声明它

```
<%@taglib prefix="security" uri="http://www.springframework.org/security/tags" %>
```

访问认证信息的细节

Spring securityJSP标签库所能做的最简单的一件事情就是**便利地访问用户的认证信息**
列如

```
Hello <sec:authentication property="username"/>
```

property属性用来标示用户认证对象的一个属性 可用的属性取决于用户认证的方式 但是可以依赖几个通用的属性 在不同的认证方式下它们都是可用的

表 9.4 你可以使用 <security:authentication> JSP 标签来访问用户的认证详细信息

认证属性	描 述
authorities	一组用于表示用户所授予权限的 GrantedAuthority 对象
credentials	用于核实用户的凭据（通常是用户的密码）
details	认证的附加信息（IP 地址、证件序列号、会话 ID 等）
principal	用户的主要信息对象

在这个例子中 实际渲染的是username属性
<sec:authentication>将在视图中渲染属性的值 但是如果你愿意将其赋值给一个变量 那只需要在var属性中指明变量的名字

```
Hello <sec:authentication property="principal.username" />
```

同时我们也是可以指定作用域 默认的作用域是page 当前的页作用域
只需要增加scope这个属性就可以

```
Hello <sec:authentication property="principal.username" var="loginId" scope="request"/>
```

根据权限渲染

有时候视图上的一部分内容需要更具用户被授予了什么权限来确定是否渲染
Spring Security的<sec:authorize>JSP 标签能够更具用户被授予的权限有条件的渲染页面的部分内容

比如 对于没有ROLE_ADMIN 不显示这个<sec:authorize>

```
<sec:authorize access="hasRole('ROLE_ADMIN')"/>
```

access属性被赋值为一个SpEL表达式 这个表达式的值将确定<security:authorize>标签的主题是否被渲染 当你设置access属性的时候 可以任意发挥SpEL的强大威力

比如：

```
<sec:authorize access="hasRole('ROLE_ADMIN') and principal.username ='admin'   ">
```

关于这个<sec:authorize> 要使用的同时还需要注意 假如

```
<sec:authorize access="hasRole('ROLE_ADMIN')">

<a href="/admin">admin</a>

</sec:authorize>
```

通过这样来防止URL暴露的话是不够的 还需要在配置文件中也进行如下说明 这样才不会导致这个人在浏览器地址栏直接输入地址而被进入的问题出现

```
<intercept-url pattern="/admin/**" access="hasRole('ROLE_ADMIN') and hasIpAddress('192.168.1.
1')"/>
```

<sec:authorize> 还有一个属性 这就是URL

不像access属性那样明确的声明安全性限制 url属性对一个给定的URL模式间接引用其安全性限制

```
<sec:authorize url="/admin**">

<spring:url value="/admin" var="admin_url"/>

<br /><a href="${admin_url}">Admin</a>

</sec:authorize>
```

因为具备ROLE_ADMIN权限的认证用户 而且来自特定IP地址的请求才能访问/admin URL 只有满足以上条件 <sec：aythorize>标签的主体才会被渲染

认证用户

每个应用程序都会有些差异 每一个应用程序是如何存储用户信息的就是一个显然易见不同之处

Spring security非常灵活 Spring security涵盖了许多常用的认证场景 包含如下的用户认证策略

1 内存
2 基于JDBC的用户存储库
3 基于LDAP的用户存储库
4 OpenID分散式的用户身份识别系统
5 中心认证服务(CAS)
6 X.509证书
7 基于JAAS的提供者

第一种方式是配置内存用户存储库

如下所示 就是像下面一样的配置方式

```
<authentication-manager>

          <authentication-provider user-service-ref="ceshi">


          </authentication-provider>

   </authentication-manager>


<user-service id="ceshi">

<!-- 这里创建两个用户，可以通过用户名密码登录 -->

<user name="user" password="user" authorities="ROLE_USER" />

    <user name="admin" password="admin" authorities="ROLE_USER,ROLE_ADMIN" />

</user-service>
```

用户服务实际上是一个数据访问对象 它在给定用户登录ID时查找用户详细信息 在使用<user-service>的场景下 用户的详细信息声明在<user-servce>之中 每个能登录应用程序的都会有一个<user>元素 属性name和password分别指定了登录名和密码 同时 authorities属性用于设置逗号分隔的权限列表 ---即指定用户做的事情
然后需要将这个<user-service>装配到<authentication-manager>和<authentication-provider>之中 也就是认证管理器

<authentication-manager>会注册一个认证管理器 更确切的讲 他将注册一个ProviderManager的实例 认证管理器把认证的任务委托给一个或多个认证提供者
还有一种方式 如下所示：

```
<authentication-manager>

          <authentication-provider>

                <user-service>

<!-- 这里创建两个用户，可以通过用户名密码登录 -->
```

```
<user name="user" password="user" authorities="ROLE_USER" />

                        <user name="admin" password="admin" authorities="ROLE_USER,ROLE_ADMIN
" />

                </user-service>

            </authentication-provider>

    </authentication-manager>
```

在进行测试或刚刚将安全性引入进来的时候 在Spring的上下文中定义用户详情信息是很便利的 但是在生成型的应用程序中 这种管理用户的方式并不现实 将用户详细存储在数据库或目录服务器中是更常见的作法

基于数据库进行认证

许多应用程序包括用户名和密码的用户信息存储在关系型数据库中
可以通过Spring security提供的<jdbc:user-service>来达到使用目的
这个<jdbc:user-service>使用方式与<user-service>类似 这包括装配到<authentication-provider>的user-service-ref属性中或嵌入到<authentication-provider> 中声明

```
<jdbc-user-service id="userService"
    data-source-ref="dataSource" />
```

<jdbc-user-service> 元素使用了一个 JDBC 数据源——通过它的 data-source-ref 属性来进行装配——来查询数据库并获取用户详细信息。如果没有其他的配置，用户服务将会使用如下的 SQL 语句来查询用户信息：

```
select username,password,enabled
  from users
 where username = ?
```

尽管我们现在讨论的是用户认证，但是一部分认证会涉及查找用户被授予的权限。默认情况下，基本的 <jdbc-user-service> 配置将使用如下 SQL 语句查询指定用户名的权限：

```
select username,authority
  from authorities
 where username = ?
```

在应用程序中，如果保存用户详细信息和授权信息的数据库表正好与这些查询相匹配，那么这是相当不错的。但是我敢打赌，对于大多数应用程序来讲，情况并不是

很多时候不太可能是像上面一样完美的契合的 所以我们可以自己指定我们需要搜索的SQL语句

表 9.5 <jdbc-user-service> 的属性能够改变查询用户详细信息的 SQL 语句

特　性	作　用
users-by-username-query	根据用户名查询用户的用户名、密码以及是否可用的状态
authorities-by-username-query	根据用户名查询用户被授予的权限
group-authorities-by-username-query	根据用户名查询用户的组权限

另外需要注明的是 我们还需要考虑一个用户可用与禁用的区别 所以在数据库中表的enable属性中 必须均都是true才能进行使用
下面是代码示例

```
<jdbc-user-service id="ceshi2" data-source-ref="dataSource"

users-by-username-query="

                SELECT username,password,enabled  FROM  users WHERE username=?

" authorities-by-username-query="
```

```
SELECT  username,authority FROM  authorities WHERE  username =?"/>



<beans:bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSourc
e">

    <beans:property name="driverClassName" value="com.mysql.jdbc.Driver"/>

    <beans:property name="url" value="jdbc:mysql://localhost:3306/test"/>

    <beans:property name="username" value="root"/>

    <beans:property name="password" value="liuziye"/>

</beans:bean>
```

```
<authentication-manager>

          <authentication-provider user-service-ref="ceshi"/>

          <authentication-provider user-service-ref="ceshi2" />

   </authentication-manager>
```

还有一种可能更常用的方式 这就是基于LDAP的进行认证

关系型数据库是非常有用 但是不能很好的表示层级数据

LDAP目录恰好擅长存储层级数据 基于这个原因 公司的组织机构在LDAP目录中进行展现也是很常见的 另外 你会发现公司的安全性限制往往对应目录中的一个条目
为了使用基于LDAP的认证 我们首先要使用spring security的LDAP模块

当配置LDAP认证时 有俩种选择（关于LDAP 不会用。。）
1 使用面向LDAP认证提供者
2 使用面向LDAP的用户服务

声明LDAP验证提供者
对于内存和基于数据库的用户服务 声明<authentication-provider>并装配用户服务 对于面向LDAP同样可以这么做 但是 我们有一个特殊的方法
就是直接声明
如下：

```
<ldap-user-service id="userService" user-search-base="ou=people" user-search-filter="(uid={0}
)"

group-search-base="ou=groups"

group-search-filter="member=0"

></ldap-user-service>
```

属性user-search-filter和group-search-filter用于为基础LDAP查询提供过滤条件 它们分别用于搜索用户和组 默认情况下 对于用户和组的基础查询都是空的 也就表示搜索会从LDAP层次结构的根开始 但是我们可以通过指定查询基础来改变这个默认想行为
这个属性就是<user-search-base>与<group-search-base>

配置密码比对

基于LDAP进行认证的默认策略是进行绑定操作 直接通过LDAP服务器认证用户 另一种可选的方式是进行比对操作 这涉及将输入的密码发送到LDAP目录上
并要求服务器将这个密码和用户的密码进行比对
如果你希望通过密码比对进行认证 则可以通过声明<password-compare>元素实现

```
<ldap-authentication-provider  server-ref="userService">

     <password-compare></password-compare>
```

```
</ldap-authentication-provider>
```

在登录表单中提交的密码将会与用户的LDAP条目中的userpassword进行比对 如果密码保存在不同名字的属性中 可以通过指定

```
<password-comparepassword-attribute="password"></password-compare>
```

另外在进行密码匹配的时候 实际的密码是保持私密的 但是在尝试的密码还是需要加密的
有几种加密的模式 这里采用了一种

- {sha}
- {ssha}
- md4
- md5
- plaintext
- sha
- sha-256

```
<password-compare  hash="md5" password-attribute="password"></password-compare>
```

引用远程的LDAP服务器

在默认情况下 Spring security的LDAP服务器监听本机的33389端口 但是 如果你的LDAP服务器在另一台机器上 那么可以使用<ldap-servr>元素来配置这个元素

```
<ldap-server url="ldap://ceshi.com:389 /dc=ceshi,dc=com"/><!--并没有这个 仅为示例-->
```

配置嵌入式的LDAP 这个是经常用来进行测试之类的用处
如果没有现成的LDAP服务器进行认证 那<ldap-server>还可以依赖嵌入式的LDAP服务 只需要去掉url参数 如下所示：

```
<ldap-serverroot="dc=springframework,dc=org"/>
```

root 属性是可选的。但它的默认值是 “dc=springframework,dc=org”，我想这不会是你的 LDAP 服务器想用的根。

当 LDAP 服务器启动时，它会尝试在类路径下查找 LDIF 文件来加载数据。LDIF（LDAP 数据交换格式）是以文本文件显示 LDAP 数据的标准方式。每条记录可以有一行或多行，每项包含一个名值对。记录之间通过空行进行分割 *。

如果你想更明确指定加载哪个 LDIF 文件，可以使用 ldif 属性：

```
<ldap-server root="dc=habuma,dc=com"
             ldif="classpath:users.ldif" />
```

启用remember-me功能（这个功能有问题）
这个remember-me就是将登陆的信息放到客户端浏览器的cookie中 到时候如果客户端需要再次登陆需要的页面的时候 就不需要再次输入账号密码进行登录了 直接登录 首先 我们只需要在<http>元素中增加一个<remember-me>元素

```
<remember-me key="Test" user-service-ref="userDetailServer"  />
```

其次 我们还需要在具体的JSP页面中说明 告诉服务器需要记住我 注意这个name值是默认的 如果要改的话需要增加以外的配置

```
<td> <input type="checkbox" name="_spring_security_remember_me"/></td>
```

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，那么记住将会不起作用，除非在你的 ApplicationContext 中拥有一个 UserDetailsService 类型的 bean。

```
<!--remember-me功能使用的配置Bean-->

<beans:bean id="userDetailServer" class="org.springframework.security.core.userdetails.jdbc.JdbcDaoImpl">

    <beans:property name="dataSource" ref="dataSource"/>

</beans:bean>
```

同时 我们还可以指定这个<remeber-me>属性的令牌（token） 默认时间是最多俩个星期有效 我们可以人为指定为4个星期

```
<remember-me key="Test" user-service-ref="userDetailServer"  token-validity-seconds="2419200"
 />
```

这种是通过简单的加密来保证基于cookie的令牌（token）的安全
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。

存储在cookie中的令牌包含用户名，密码，过期时间和一个私钥--在写入cookie之前都进行了MD5哈希 默认情况下 私钥名为SpringSecured 但可以将它设置为XXXXKey来使其专门用于XXXX应用中
另外还有一种从数据库保存令牌的方法 这种相对安全很多

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。

这种就直接粘贴代码了
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>
```

保护方法调用

这本书讲的不全 还有其他的方法可以进行保护 而且需要注意 **我们要拦截的方法 首先是需要在xml中进行bean的声明的 然后从spring中自带的方法获取才行 不能自己new一个然后使用 这种是没有用的 这在教程中都没有讲到 需要注意**

其他的几种方法只粘贴代码
顺便复制学习这几种方法的博客地址http://www.mossle.com/docs/auth/html/ch201-method.html

```
<beans:bean id="service" class="com.test.Spitter.service.MessageServiceImpl">

    <intercept-methods>

        <protect method="adminMessage" access="ROLE_USER"/>

        <protect method="adminDate" access="ROLE_USER"/>

    </intercept-methods>

</beans:bean>
```

注意这个是不能与

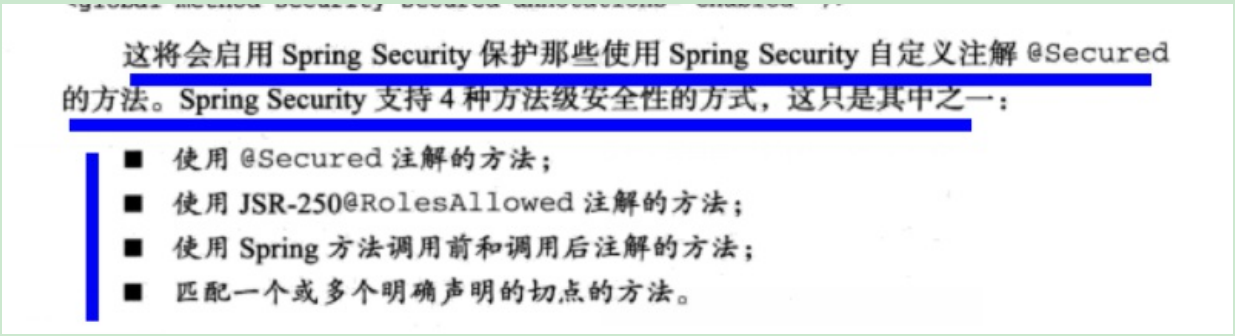
```
<global-method-security secured-annotations="enabled"/>
```

一起用的

另外是本书介绍的内容
本书都是用的是注解的方式
安全是一个面向切面的概念 Spring AOP是Spring security中方法级安全性的基础 但是在大多数情况下 你没有必要直接处理Spring security的切面
保护方法调用中所有使用的AOP打包进了一个元素
<global-method-security>

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

这是主要的使用方式



这将会启用 Spring Security 保护那些使用 Spring Security 自定义注解 @Secured 的方法。Spring Security 支持 4 种方法级安全性的方式，这只是其中之一：

- 使用 @Secured 注解的方法；
- 使用 JSR-250@RolesAllowed 注解的方法；
- 使用 Spring 方法调用前和调用后注解的方法；
- 匹配一个或多个明确声明的切点的方法。

1 使用Secured注解方式

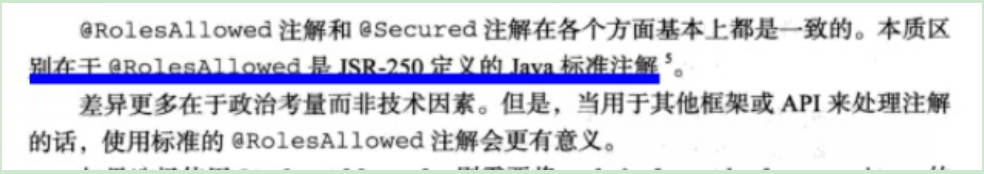
```
@Secured("ROLE_USER")
public  void ceshi1() {
    System.out.println("ceshi1");
}
```

可以声明为2个权限

```
@Secured({"ROLE_USER","ROLE_ADMIN"})
public  void ceshi1() {
    System.out.println("ceshi1");
}
```

这个就是说 认证的用户必须最少满足这俩个权限之一的才能被访问
如果没有 就抛出一个Spring security异常（AuthenticationException或Access-DeniedException的子类）这个异常必须要被捕获 如果被保护的方法 在web请求中调用 这个异常会被Spring security的过滤器自动处理
@Secured注解的不足之处在于它是Spring的注解
如果更倾向使用标准注解的话 就使用下面这个

2 使用JSR-250的RolesAllowed注解



@RolesAllowed 注解和 @Secured 注解在各个方面基本上都是一致的。本质区别在于 @RolesAllowed 是 JSR-250 定义的 Java 标准注解[5]。

差异更多在于政治考量而非技术因素。但是，当用于其他框架或 API 来处理注解的话，使用标准的 @RolesAllowed 注解会更有意义。

注意需要打开识别JSP的注解的功能

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

然后进行声明

```
@RolesAllowed("ROLE_ADMIN")
```

```
@RolesAllowed("ROLE_ADMIN")
public  void ceshi1() {
     System.out.println("ceshi1");
}
```

使用SpEL来保证调用前后的安全性

这俩种只能够满足基本的安全认证
但是 有时候
安全限制很有意思 不仅仅涉及用户是否拥有权限
在Spring security3中引入了几个新注解 它们使用SpEl能够在方法调用上实现更有意思的安全性限制 这些新的注解在表9.6进行了描述

表 9.6 Spring Security 3.0 提供了 4 个新的注解，可以使用 SpEL 表达式来保护方法调用

注解	描述
@PreAuthorize	在方法调用之前，基于表达式的计算结果来限制对方法的访问
@PostAuthorize	允许方法调用，但是如果表达式计算结果为 false，则会抛出一个安全性异常
@PostFilter	允许方法调用，但必须按照表达式来过滤方法的结果
@PreFilter	允许方法调用，但必须在进入方法之前过滤输入值

注意 使用这个注解是需要把pre-post-annotations功能开启

```
<global-method-securitysecured-annotations="enabled" jsr250-annotations="enabled" pre-post-an
notations="enabled"/>
```

1 首先是
在方法调用前验证权限
使用的是@PreAuthorize
这种注解内部的String参数是一个SpEL表达式

```
@PreAuthorize("(hasRole('ROLE_USER') and #s.length() <=10) " +
"or hasRole('ROLE_ANONYMOUS')")
public  void ceshi1(String s) {
    System.out.println("ceshi1 有参"+s);
}
```

注意这里的参数中的

```
 and #s.length() <=10)
```

s直接引用了方法中的同名参数 使得这个Spring security可以检查传入方法的参数 并将这些参数用于认证政策的制定

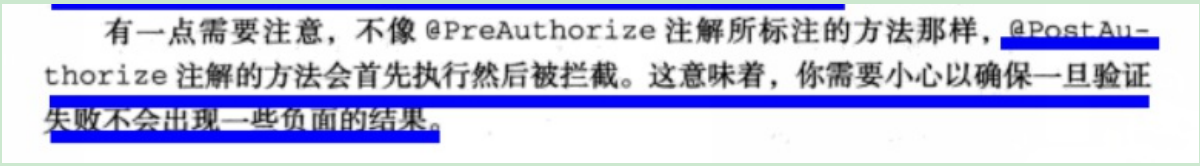
在方法调用之后进行验证权限
除了验证的时机外 @PostAuthorize与@PreAuthorize差不多
事后验证一般用于基于安全保护方法的返回值来进行安全性决策的场景中

注意这里与书上描写的是不一样的
书上描写的写法并不正确
正确的写法如下

```
@PostAuthorize("(returnObject == principal.username) or hasRole('ROLE_ADMIN')")
public  String ceshi1(String s) {
System.out.println("ceshi1 有参"+s);
return s;
}
```

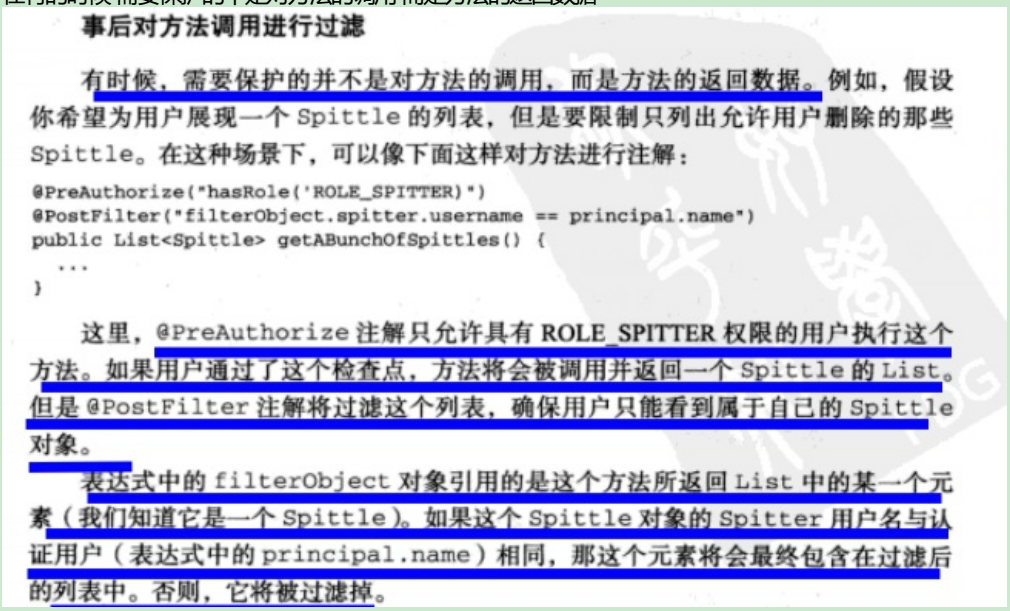
```
@PostAuthorize("returnObject.id==10")
public user ceshi1(int d) {
System.out.println(d);
user u = new user();
u.setId(d);
return u;
}
```

如果在上面的id不等于10的情况下 就会导致出错 会抛出一个AccessDenigedException异常 而调用者也不会得到user对象



有一点需要注意，不像 @PreAuthorize 注解所标注的方法那样，@PostAuthorize 注解的方法会首先执行然后被拦截。这意味着，你需要小心以确保一旦验证失败不会出现一些负面的结果。

事后对方法进行过滤
在有的时候 需要保护的不是对方法的调用 而是方法的返回数据



事后对方法调用进行过滤

有时候，需要保护的并不是对方法的调用，而是方法的返回数据。例如，假设你希望为用户展现一个 Spittle 的列表，但是要限制只列出允许用户删除的那些 Spittle。在这种场景下，可以像下面这样对方法进行注解：

```
@PreAuthorize("hasRole('ROLE_SPITTER')")
@PostFilter("filterObject.spitter.username == principal.name")
public List<Spittle> getABunchOfSpittles() {
  ...
}
```

这里，@PreAuthorize 注解只允许具有 ROLE_SPITTER 权限的用户执行这个方法。如果用户通过了这个检查点，方法将会被调用并返回一个 Spittle 的 List。但是 @PostFilter 注解将过滤这个列表，确保用户只能看到属于自己的 Spittle 对象。

表达式中的 filterObject 对象引用的是这个方法所返回 List 中的某一个元素（我们知道它是一个 Spittle）。如果这个 Spittle 对象的 Spitter 用户名与认证用户（表达式中的 principal.name）相同，那这个元素将会最终包含在过滤后的列表中。否则，它将被过滤掉。

```
@PreAuthorize("hasRole('ROLE_USER')")
@PostFilter("filterObject.id == 20")
public List<user> ceshi2(int d) {
List<user> list = new ArrayList<user>();
System.out.println(d);
user u = new user();
user u2 = new user();
u.setId(d);
u2.setId(20);
list.add(u);
```

```
list.add(u2);

return list;

}
```

```
isecuritCeshi securitCeshi =  (isecuritCeshi) context.getBean("service2");

//        System.out.println(securitCeshi.ceshi1(10).toString());

System.out.println(securitCeshi.ceshi2(10).toString());
```

```
[user{id=20}]
```

这是我自己手写的一个列子

大概意思差不多
在这里进行了这个注解的过滤器的过滤

最后返回的是经过过滤之后匹配的那个user
然后再进行使用
我们还可以进行一些其他的操作

为了让事情变得更有趣，假设用户除了能够删除自己的 Spittle，还能删除任何包含侮辱性语句的 Spittle。为了做到这一点，我们需要重写 @PostFilter 表达式，如下所示：

```
@PreAuthorize("hasRole('ROLE_SPITTER')")
@PostFilter("hasPermission(filterObject, 'delete')")
public List<Spittle> getSpittlesToDelete() {
  ...
}
```

按照这种使用方式，如果用户对 filterObject 表示的 Spittle 对象有删除的权限，hasPermission() 操作应该返回 true。在这种情况下，我是说它应该计算得到 true 的结果，但实际上 hasPermission() 默认一直返回 false。

如果 hasPermission() 默认一直返回 false，那它的用处是什么呢？好在默认行为是可以重写的。重写 hasPermission() 的默认行为涉及创建和注册一个许可计算器。这是 SpittlePermissionEvaluator 所做的事情，如程序清单 9.7 所示。

可以自己重新修改具体的方法实现

```
package com.habuma.spitter.security;
import java.io.Serializable;
import org.springframework.security.access.PermissionEvaluator;
import org.springframework.security.core.Authentication;
import com.habuma.spitter.domain.Spittle;

public class SpittlePermissionEvaluator implements PermissionEvaluator {
    public boolean hasPermission(Authentication authentication,
            Object target, Object permission) {
        if (target instanceof Spittle) {
            Spittle spittle = (Spittle) target;
            if ("delete".equals(permission)) {
                return spittle.getSpitter().getUsername().equals(
                    authentication.getName()) || hasProfanity(spittle);
            }
        }
        throw new UnsupportedOperationException(
                "hasPermission not supported for object <" + target
                        + "> and permission <" + permission + ">");
    }

    public boolean hasPermission(Authentication authentication,
            Serializable targetId, String targetType, Object permission) {
        throw new UnsupportedOperationException();
    }
    private boolean hasProfanity(Spittle spittle) {
        ...
        return false;
    }
}
```

这是个示例

valuator 接口，它需要实现两个不同的 hasPermission() 方法。其中的一个 hasPermission() 方法把要评估的对象作为第二个参数。第二个 hasPermission() 方法在只有目标对象的 ID 可以得到的时候才有用，并将 ID 作为 Serializable 传入第二个参数。

为了满足需求，假设使用 Spittle 对象来评估权限，所以第二个方法只是简单地抛出 UnsupportedOperationException。

对于另一个 hasPermission() 方法，检查要评估的对象是否为一个 Spittle 对象，并检查是否为删除权限。如果是这样，它将对比 Spitter 的用户名与认证用户的名字。它也会将 Spittle 传递给 hasProfanity() 方法来检查其是否包含侮辱性的语句[6]。

许可计算器已经准备就绪，你需要将其注册到 Spring Security 中，以便在使用 @PostFilter 时支持 hasPermission() 操作。要做到这一点需要创建一个表达式处理器并注册到 <global-method-security> 中。

对于表达式处理器，你需要创建 DefaultMethodSecurityExpressionHandler 类型的 Bean，并将 SpittlePermissionEvaluator 的实例作为它的 permissionEvaluator 属性注入进去：

```
<beans:bean id="expressionHandler" class=
    "org.springframework.security.access.expression.method.
                                ➥DefaultMethodSecurityExpressionHandler">
  <beans:property name="permissionEvaluator">
    <beans:bean class=
                "com.habuma.spitter.security.SpittlePermissionEvaluator" />
  </beans:property>
</beans:bean>
```

接下来，我们就可以在 <global-method-security> 中配置 expression-Handler，如下所示：

```
<global-method-security pre-post-annotations="enabled">
    <expression-handler ref="expressionHandler"/>
</global-method-security>
```

以前，在配置 <global-method-security> 时，我们没有指定表达式处理器。但是在这里，配置了帮我们计算的表达式处理器，用于替换默认的表达式处理器。

我在这里不想实现了

声明方法级别的安全性节点

有的时候 我们可以给几个方法设置同样的权限 这个时候我们就可以利用以下的办法了 声明一个切点 expression属性被设置Aspectj切面表达式 access 属性标示认证用户需要什么样的权限才能访问expression属性指定的方法

```
<global-method-security secured-annotations="enabled" jsr250-annotations="enabled" pre-post-a
nnotations="enabled">

       <protect-pointcut expression="execution(* com.test.Spitter.service.MessageServiceImpl.
admin*(..))" access="ROLE_ADMIN"/>

        </global-method-security>
```