Part I. 入门

本指南的后面部分提供对框架结构和实现类的深入讨论，了解它们，对你进行复杂的定制是十分重要的。在这部分，我们将介绍 Spring Security 3.0，简要介绍该项目的历史，然后看看如何开始在程序中使用框架。特别是，我们将看看命名控件配置提供了一个更加简单的方式，在使用传统的 spring bean 配置时，你不得不实现所有类。

我们也会看看可用的范例程序。它们值得试着运行，实验，在你阅读后面的章节之前 - 你可以在对框架有了更多连接之后再回来看这些例子。也请参考 项目网站 获得构建项目有用的信息，另外链接到网站，视频和教程。

Chapter 1. 介绍

1.1. Spring Security 是什么？

Spring Security 为基于 J2EE 企业应用软件提供了全面安全服务。 特别是使用领先的 J2EE 解决方案-spring 框架开发的企业软件项目。 如果你没有使用 Spring 开发企业软件，我们热情的推荐你仔细研究一下。 熟悉 Spring-尤其是依赖注入原理-将帮助你更快的掌握 Spring Security。

人们使用 Spring Security 有很多种原因， 不过通常吸引他们的是在 J2EE Servlet 规范或 EJB 规范中找不到典型企业应用场景的解决方案。 提到这些规范，特别要指出的是它们不能在 WAR 或 EAR 级别进行移植。这样，如果你更换服务器环境， 就要，在新的目标环境进行大量的工作，对你的应用系统进行重新配置安全。 使用 Spring Security 解决了这些问题，也为你提供了很多有用的，可定制的其他安全特性。

你可能知道，安全包括两个主要操作，“认证”和“验证”（或权限控制）。 这就是 Spring Security 面向的两个主要方向。“认证” 是为用户建立一个他所声明的主体的过程，（“主体”一般是指用户，设备或可以在你系统中执行行动的其他系统）。 “验证”指的一个用户能否在你的应用中执行某个操作。 在到达授权判断之前，身份的主体已经由身份验证过程建立了。 这些概念是通用的，不是 Spring Security 特有的。

在身份验证层面，Spring Security 广泛支持各种身份验证模式。 这些验证模型绝大多数都由第三方提供，或正在开发的有关标准机构提供的，例如 Internet Engineering Task Force。 作为补充，Spring Security 也提供了自己的一套验证功能。 Spring Security 目前支持认证一体化和如下认证技术：

- HTTP BASIC authentication headers (一个基于 IEFT RFC 的标准)
- HTTP Digest authentication headers (一个基于 IEFT RFC 的标准)
- HTTP X.509 client certificate exchange (一个基于 IEFT RFC 的标准)
- LDAP (一个非常常见的跨平台认证需要做法，特别是在大环境)
- Form-based authentication (提供简单用户接口的需求)
- OpenID authentication

- 基于预先建立的请求头进行认证 （比如 Computer Associates Siteminder）
- JA-SIG Central Authentication Service (也被称为 CAS，这是一个流行的开源单点登录系统)
- Transparent authentication context propagation for Remote Method Invocation (RMI) and HttpInvoker (一个 Spring 远程调用协议)
- Automatic "remember-me" authentication (这样你可以设置一段时间，避免在一段时间内还需要重新验证)
- Anonymous authentication (允许任何调用，自动假设一个特定的安全主体)
- Run-as authentication (这在一个会话内使用不同安全身份的时候是非常有用的)
- Java Authentication and Authorization Service (JAAS)
- JEE Container autentication (这样，你可以继续使用容器管理认证，如果想的话)
- Kerberos
- Java Open Source Single Sign On (JOSSO) *
- OpenNMS Network Management Platform *
- AppFuse *
- AndroMDA *
- Mule ESB *
- Direct Web Request (DWR) *
- Grails *
- Tapestry *
- JTrac *
- Jasypt *
- Roller *
- Elastic Plath *
- Atlassian Crowd *
- 你自己的认证系统(向下看)

(* 是指由第三方提供，查看我们的整合网页，获得最新详情的链接。)

许多独立软件供应商（ISVs, independent software vendors）采用 Spring Security，是因为它拥有丰富灵活的验证模型。 这样，无论终端用户需要什么，他们都可以快速集成到系统中，不用花很多功夫，也不用让用户改变运行环境。 如果上述的验证机制都没有满足你的需要，Spring Security 是一个开放的平台，编写自己的验证机制是十分简单的。 Spring Security 的许多企业用户需要整合不遵循任何特定安全标准的“遗留”系统，Spring Security 在这类系统上也表现的很好。

有时基本的认证是不够的。 有时你需要根据在主体和应用交互的方式来应用不同的安全措施。 比如，你可能，为了保护密码，不被窃听或受到中间人攻击，希望确保请求

只通过 HTTPS 到达。 这在防止暴力攻击保护密码恢复过程特别有帮助， 或者简单的，让人难以复制你的系统的关键字内容。 为了帮助你实现这些目标，Spring Security 完全支持自动"信道安全"， 整合 jcaptcha 一体化进行人类用户检测。

Spring Security 不仅提供认证功能，也提供了完备的授权功能。 在授权方面主要有三个领域，授权 web 请求，授权被调用方法，授权访问单个对象的实例。 为了帮你了解它们之间的区别，对照考虑授在 Servlet 规范 web 模式安全，EJB 容器管理安全，和文件系统安全方面的授权方式。 Spring Security 在所有这些重要领域都提供了完备的能力，我们将在这份参考指南的后面进行探讨。

1.2. 历史

Spring Security 开始于 2003 年年底，""spring 的 acegi 安全系统"。起因是 Spring 开发者邮件列表中的一个问题，有人提问是否考虑提供一个基于 spring 的安全实现。在当时 Spring 的社区相对较小（尤其是和今天的规模比！），其实 Spring 本身是从 2003 年初才作为一个 sourceforge 的项目出现的。 对这个问题的回应是，这的确是一个值得研究的领域，虽然限于时间问题阻碍了对它的继续研究。

有鉴于此，一个简单的安全实现建立起来了，但没有发布。 几周之后，spring 社区的其他成员询问安全问题，代码就被提供给了他们。 随后又有人请求，在 2004 年一月左右，有 20 人在使用这些代码。 另外一些人加入到这些先行者中来，并建议在 sourceforge 上建立一个项目，项目在 2004 年 3 月正式建立起来。

在早期，项目本身没有自己的认证模块。 认证过程都是依赖容器管理安全的，而 acegi 则注重授权。 这在一开始是合适的，但随着越来越多用户要求提供额外的容器支持，基于容器认证的限制就显现出来了。 还有一个有关的问题，向容器的 classpath 中添加新 jar，常常让最终用户感到困惑，又容易出现配置错误。

随后 acegi 加入了认证服务。大约一年后，acegi 成为 spring 的官方子项目。 经过了两年半在许多生产软件项目中的活跃使用和数以万计的改善和社区的贡献，1.0.0 最终版本发布于 2006 年 5 月。

acegi 在 2007 年年底，正式成为 spring 组合项目，被更名为"Spring Security"。

现在，Spring Security 成为了一个强大而又活跃的开源社区。 在 Spring Security 支持论坛上有成千上万的信息。 有一个积极的核心开发团队专职开发，一个积极的社区定期共享补丁并支持他们的同伴。

1.3. 发行版本号

了解 spring Security 发行版本号是非常有用的。它可以帮助你判断升级到新的版本是否需要花费很大的精力。 我们使用 apache 便携式运行项目版本指南，可以在以下网址查看 http://apr.apache.org/versioning.html。 为了方便大家，我们引用页面上的一段介绍：

"版本号是一个包含三个整数的组合：主要版本号.次要版本号.补丁。 基本思路是主要版本是不兼容的，大规模升级 API。 次要版本号在源代码和二进制要与老版本保持兼容，补丁则意味着向前向后的完全兼容。"

1.4. 获得 Spring Security

你可以通过多种方式获得 Spring Security。你可以下载打包好的发行包，从 Spring 的网站 下载页， 下载单独的 jar（和实例 WAR 文件）从 Maven 中央资源库（或者 SpringSource Maven 资源库，获得快照和里程碑发布）。 可选的，你可以通过源代码创建项目。参考项目网站获得更多细节。

1.4.1. 项目模块

在 Spring Security 3.0 中，项目已经分割成单独的 jar，这更清楚的按照功能进行分割模块和第三方依赖。 如果你在使用 Maven 来构建你的项目，你需要把这些模块添加到你的 pom.xml 中。 即使你没有使用 Maven，我们也推荐你参考这个 pom.xml 文件， 来了解第三方依赖和对应的版本。可选的，一个好办法是参考实例应用中包含的依赖库。

1.4.1.1. Core - spring-security-core.jar

包含了核心认证和权限控制类和接口， 运程支持和基本供应 API。使用 Spring Security 所必须的。支持单独运行的应用， 远程客户端，方法（服务层）安全和 JDBC 用户供应。包含顶级包：

- org.springframework.security.core
- org.springframework.security.access
- org.springframework.security.authentication
- org.springframework.security.provisioning
- org.springframework.security.remoting

1.4.1.2. Web - spring-security-web.jar

包含过滤器和对应的 web 安全架构代码。任何需要依赖 servlet API 的。 你将需要它，如果你需要 Spring Security Web 认证服务和基于 URL 的权限控制。 主包是 org.springframework.security.web。

1.4.1.3. Config - spring-security-config.jar

包含安全命名控制解析代码（因此我们不能直接把它用在你的应用中）。你需要它， 如果使用了 Spring Security XML 命名控制来进行配置。主包是 org.springframework.security.config。

1.4.1.4. LDAP - spring-security-ldap.jar

LDAP 认证和实现代码，如果你需要使用 LDAP 认证或管理 LDAP 用户实体就是必须的。顶级包是 org.springframework.security.ldap。

1.4.1.5. ACL - spring-security-acl.jar

处理领域对象 ACL 实现。用来提供安全给特定的领域对象实例，在你的应用中。 顶级包是 org.springframework.security.acls。

1.4.1.6. CAS - spring-security-cas-client.jar

Spring Security 的 CAs 客户端集成。如果你希望使用 Spring Security web 认证 整合一个 CAS 单点登录服务器。顶级包是 org.springframework.security.cas。

1.4.1.7. OpenID - spring-security-openid.jar

OpenID web 认证支持。用来认证用户，通过一个外部的 OpenID 服务。org.springframework.security.openid。需要 OpenID4Java。

1.4.2. 获得源代码

Spring Security 是一个开源项目，我们大力推荐你从 subversion 获得源代码。 这样你可以获得所有的示例，你可以很容易的建立目前最新的项目。 获得项目的源代码对调试也有很大的帮助。 异常堆栈不再是模糊的黑盒问题，你可以直接找到发生问题的那一行，查找发生了什么额外难题。 源代码也是项目的最终文档，常常是最简单的方法，找出这些事情是如何工作的。

要像获得项目最新的源代码，使用如下 subversion 命令：

```
svn checkout
http://acegisecurity.svn.sourceforge.net/svnroot/acegisecurity/
spring-security/trunk/
```

Security 命名空间配置

2.1. 介绍

从 Spring-2.0 开始可以使用命名空间的配置方式。 使用它呢，可以通过附加 xml 架构，为传统的 spring beans 应用环境语法做补充。 你可以在 spring 参考文档得到更多信息。 命名空间元素可以简单的配置单个 bean，或使用更强大的，定义一个备用配置语法，这可以更加紧密的匹配问题域，隐藏用户背后的复杂性。 简单元素可能隐藏事实，多种 bean 和处理步骤添加到应用环境中。 比如，把下面的 security 命名元素添加到应用环境中，将会为测试用途，在应用内部启动一个内嵌 LDAP 服务器：

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

这比配置一个 Apache 目录服务器 bean 要简单得多。 最常见的替代配置需求都可以使用 ldap-server 元素的属性进行配置，这样用户就不用担心他们需要设置什么，不用担心 bean 里的各种属性。 [1]。使用一个良好的 XML 编辑器来编辑应用环境文件，

应该提供可用的属性和元素信息。 我们推荐你尝试一下 SpringSource 工具套件 因为它具有处理 spring 组合命名空间的特殊功能。

要开始在你的应用环境里使用 security 命名空间，你所需要的就是把架构声明添加到你的应用环境文件里：

```
<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:security="http://www.springframework.org/schema/security"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xs
d
          http://www.springframework.org/schema/security

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

在许多例子里，你会看到（在示例中）应用，我们通常使用"security"作为默认的命名空间，而不是"beans"，这意味着我们可以省略所有 security 命名空间元素的前缀，使上下文更容易阅读。 如果你把应用上下文分割成单独的文件，让你的安全配置都放到其中一个文件里，这样更容易使用这种配置方法。 你的安全应用上下文应该像这样开头

```
<beans:beans

xmlns="http://www.springframework.org/schema/security"

  xmlns:beans="http://www.springframework.org/schema/beans"

  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"


xsi:schemaLocation="http://www.springframework.org/schema/beans


http://www.springframework.org/schema/beans/spring-beans-3.0.xs

d

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


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

3.0.xsd">

    ...

</beans:beans>
```

就在这一章里，我们都将假设使用这种语法。

2.1.1. 命名空间的设计

命名空间被用来设计成，处理框架内最常见的功能，提供一个简化和简洁的语法，使他们在一个应用程序里。 这种设计是基于框架内的大型依赖，可以分割成下面这些部分：

- *Web/HTTP 安全* - 最复杂的部分。设置过滤器和相关的服务 bean 来应用框架验证机制， 保护 URL，渲染登录和错误页面还有更多。
- *业务类（方法）安全* - 可选的安全服务层。
- *AuthenticationManager* - 通过框架的其它部分，处理认证请求。
- *AccessDecisionManager* - 提供访问的决定，适用于 web 以及方法的安全。一个默认的主体会被注册，但是你也可以选择自定义一个，使用正常的 spring bean 语法进行声明。

- *AuthenticationProviders* - 验证管理器验证用户的机制。 该命名空间提供几种标准选项，意味着使用传统语法添加自定义 bean。
- *UserDetailsService* - 密切相关的认证供应器，但往往也需要由其他 bean 需要。

下一章中，我们将看到如何把这些放到一起工作。

2.2. 开始使用安全命名空间配置

在本节中，我们来看看如何使用一些框架里的主要配置，建立一个命名空间配置。 我们假设你最初想要尽快的启动运行，为已有的 web 应用添加认证支持和权限控制，使用一些测试登录。然后我们看一下如何修改一下，使用数据库或其他安全信息参数。在以后的章节里我们将引入更多高级的命名空间配置选项。

2.2.1. 配置 web.xml

我们要做的第一件事是把下面的 filter 声明添加到 web.xml 文件中：

```
<filter>

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



<filter-class>org.springframework.web.filter.DelegatingFilterPr

oxy</filter-class>

</filter>



<filter-mapping>

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

  <url-pattern>/*</url-pattern>

</filter-mapping>
```

这是为 Spring Security 的 web 机制提供了一个调用钩子。DelegatingFilterProxy 是一个 Spring Framework 的类，它可以代理一个 application context 中定义的 Spring bean 所实现的 filter。 这种情况下，bean 的名字是

"springSecurityFilterChain"，这是由命名空间创建的用于处理 web 安全的一个内部的机制。 注意，你不应该自己使用这个 bean 的名字。 一旦你把这个添加到你的 web.xml 中，你就准备好开始编辑呢的 application context 文件了。 web 安全服务是使用<http>元素配置的。

2.2.2. 最小 <http>配置

只需要进行如下配置就可以实现安全配置：

```
<http auto-config='true'>
  <intercept-url pattern="/**" access="ROLE_USER" />
</http>
```

这表示，我们要保护应用程序中的所有 URL，只有拥有 ROLE_USER 角色的用户才能访问。<http>元素是 所有 web 相关的命名空间功能的上级元素。<intercept-url>元素定义了 pattern，用来匹配进入的请求 URL，使用一个 ant 路径语法。 access 属性定义了请求匹配了指定模式时的需求。使用默认的配置， 这个一般是一个逗号分隔的角色队列，一个用户中的一个必须被允许访问请求。 前缀"ROLE_"表示的是一个用户应该拥有的权限比对。换句话说， 一个普通的基于角色的约束应该被使用。Spring Security 中的访问控制不限于简单角色的应用 （因此，我们使用不同的前缀来区分不同的安全属性）.我们会在后面看到这些解释是可变的 [2]

Note

你可以使用多个<intercept-url>元素为不同 URL 的集合定义不同的访问需求，它们会被归入一个有序队列中，每次取出最先匹配的一个元素使用。 所以你必须把期望使用的匹配条件放到最上边。你也可以添加一个 method 属性 来限制匹配一个特定的 HTTP method(GET, POST, PUT 等等)。对于一个模式同时定义在定义了 method 和未定义 method 的情况， 指定 method 的匹配会无视顺序优先被使用。

要是想添加一些用户，你可以直接使用下面的命名空间直接定义一些测试数据：

```
<authentication-manager>
  <authentication-provider>
    <user-service>
      <user        name="jimi"        password="jimispassword"
authorities="ROLE_USER, ROLE_ADMIN" />
```

```
        <user          name="bob"          password="bobspassword"
authorities="ROLE_USER" />
      </user-service>
    </authentication-provider>
  </authentication-manager>
```

如果你熟悉以前的版本，你很可能已经猜到了这里是怎么回事。 <http>元素会创建一个 FilterChainProxy 和 filter 使用的 bean。 以前常常出现的，因为 filter 顺序不正确产生的问题，不会再出现了，现在这些过滤器的位置都是预定义好的。

<authentication-provider>元素创建了一个 DaoAuthenticationProvider bean，<user-service>元素创建了一个 InMemoryDaoImpl。 所有 authentication-provider 元素必须作为<authentication-manager>的子元素，它创建了一个 ProviderManager，并把 authentication provider 注册到它里面。 你可以在命名空间附录中找到关于创建这个 bean 的更新信息。 很值得去交叉检查一下这里，如果你希望开始理解框架中哪些是重要的类 以及它们是如何使用的，特别是如果你希望以后做一些自定义工作。

上面的配置定义了两个用户，他们在应用程序中的密码和角色（用在权限控制上）。 也可以从一个标准 properties 文件中读取这些信息，使用 user-service 的 properties 属性。 参考 in-memory authentication 获得更多信息。 使用<authentication-provider>元素意味着用户信息将被认证管理用作处理认证请求。你可以拥有多个<authentication-provider>元素来定义不同的认证数据，每个会被需要时使用。

现在，你可以启动程序，然后就会进入登录流程了。 试试这个，或者试试工程里的"tutorial"例子。 上述配置实际上把很多服务添加到了程序里，因为我们使用了 auto-config 属性。 比如，表单登录和"remember-me"服务自动启动了。

2.2.2.1. auto-config 包含了什么？

我们在上面用到的 auto-config 属性，其实是下面这些配置的缩写：

```
  <http>
    <form-login />
    <http-basic />
```

```
    <logout />

  </http>
```

这些元素分别与 form-login，基本认证和注销处理对应。 [3] 他们拥有各自的属性，来改变他们的具体行为。

2.2.2.2. 表单和基本登录选项

你也许想知道，在需要登录的时候，去哪里找这个登录页面，到现在为止我们都没有提到任何的 HTML 或 JSP 文件。 实际上，如果我们没有确切的指定一个页面用来登录，Spring Security 会自动生成一个，基于可用的功能，为这个 URL 使用标准的数据，处理提交的登录，然后在登陆后发送到默认的目标 URL。 然而，命名空间提供了许多支持，让你可以自定义这些选项。 比如，如果你想实现自己的登录页面，你可以使用：

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

    <intercept-url                              pattern="/login.jsp*"

access="IS_AUTHENTICATED_ANONYMOUSLY"/>

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

    <form-login login-page='/login.jsp'/>

  </http>
```

注意，你依旧可以使用 auto-config。 这个 form-login 元素会覆盖默认的设置。 也要注意我们需要添加额外的 intercept-url 元素，指定用来做登录的页面的 URL， 这些 URL 应该可以被匿名访问。 [4] 否则，这些请求会被/**部分拦截，它没法访问到登录页面。 这是一个很常见的配置错误，它会导致系统出现无限循环。 Spring Security 会在日志中发出一个警告，如果你的登录页面是被保护的。 也可能让所有的请求都匹配特定的模式，通过完全的安全过滤器链：

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

```
    <intercept-url pattern="/css/**" filters="none"/>
    <intercept-url pattern="/login.jsp*" filters="none"/>
    <intercept-url pattern="/**" access="ROLE_USER" />
    <form-login login-page='/login.jsp'/>
  </http>
```

主要的是意识到这些请求会被完全忽略,对任何 Spring Security 中 web 相关的配置，或额外的属性，比如 requires-channel， 所以你会不能访问当前用户信息，或调用被保护方法，在请求过程中。 使用 access='IS_AUTHENTICATED_ANONYMOUSLY'作为一个选择方式 如果你还想要安全过滤器链起作用。

如果你希望使用基本认证，代替表单登录，可以把配置改为：

```
  <http auto-config='true'>
    <intercept-url pattern="/**" access="ROLE_USER" />
    <http-basic />
  </http>
```

基本身份认证会被优先用到,在用户尝试访问一个受保护的资源时,用来提示用户登录。在这种配置中，表单登录依然是可用的，如果你还想用的话，比如，把一个登录表单内嵌到其他页面里。

2.2.2.2.1. 设置一个默认的提交登陆目标

如果在进行表单登陆之前，没有试图去访问一个被保护的资源，default-target-url 就会起作用。 这是用户登陆后会跳转到的 URL，默认是"/"。 你也可以把 always-use-default-target 属性配置成"true",这样用户就会一直跳转到这一页(无论登陆是"跳转过来的"还是用户特定进行登陆）。 如果你的系统一直需要用户从首页进入，就可以使用它了，比如：

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

2.2.3. 使用其他认证提供器

现实中，你会需要更大型的用户信息源，而不是写在 application context 里的几个名字。 多数情况下，你会想把用户信息保存到数据库或者是 LDAP 服务器里。 LDAP 命名空间会在LDAP 章里详细讨论,所以我们这里不会讲它。如果你自定义了一个Spring Security 的 UserDetailsService 实现，在你的 application context 中名叫 "myUserDetailsService"，然后你可以使用下面的验证。

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

如果你想用数据库，可以使用下面的方式

```
  <authentication-manager>
    <authentication-provider>
```

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

这里的"securityDataSource"就是 DataSource bean 在 application context 里的名字，它指向了包含着 Spring Security 用户信息的表。另外，你可以配置一个 Spring Security JdbcDaoImpl bean，使用 user-service-ref 属性指定：

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

  <beans:bean id="myUserDetailsService"

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

你也可以使用标准的 AuthenticationProvider 类，像下面

```
  <authentication-manager>
```

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

这里 myAuthenticationProvider 是你的 application context 中的一个 bean 的名字，它实现了 AuthenticationProvider。 查看 Section 2.6,"验证管理器和命名空间"了解更多信息， AuthenticationManager 使用命名空间在 Spring Security 中是如何配置的。

2.2.3.1. 添加一个密码编码器

你的密码数据通常要使用一种散列算法进行编码。 使用<password-encoder>元素支持这个功能。 使用 SHA 加密密码，原始的认证供应器配置，看起来就像这样：

```
<authentication-manager>
  <authentication-provider>
    <password-encoder hash="sha" />
    <user-service>
      <user name="jimi"
password="d7e6351eaa13189a5a3641bab846c8e8c69ba39f"
            authorities="ROLE_USER, ROLE_ADMIN" />
      <user name="bob"
password="4e7421b1b8765d8f9406d87e7cc6aa784c4ab97f"
            authorities="ROLE_USER" />
    </user-service>
  </authentication-provider>
</authentication-manager>
```

```
```

在使用散列密码时，用盐值防止字典攻击是个好主意，Spring Security 也支持这个功能。 理想情况下，你可能想为每个用户随机生成一个盐值，不过，你可以使用从 UserDetailsService 读取出来的 UserDetails 对象中的属性。比如，使用 username 属性，你可以这样用：

```
<password-encoder hash="sha">
  <salt-source user-property="username"/>
</password-encoder>
```

你可以通过 password-encoder 的 ref 属性，指定一个自定义的密码编码器 bean。这应该包含 application context 中一个 bean 的名字，它应该是 Spring Security 的 PasswordEncoder 接口的一个实例。

2.3. 高级 web 特性

2.3.1. Remember-Me 认证

参考 Remember-Me 章获得 remember-me 命名空间配置的详细信息。

2.3.2. 添加 HTTP/HTTPS 信道安全

如果你的同时支持 HTTP 和 HTTPS 协议，然后你要求特定的 URL 只能使用 HTTPS，这时可以直接使用<intercept-url>的 requires-channel 属性：

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

使用了这个配置以后，如果用户通过 HTTP 尝试访问"/secure/**"匹配的网址，他们会先被重定向到 HTTPS 网址下。 可用的选项有"http", "https" 或 "any"。 使用"any"意味着使用 HTTP 或 HTTPS 都可以。

如果你的程序使用的不是 HTTP 或 HTTPS 的标准端口，你可以用下面的方式指定端口对应关系：

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

你可以在???找到更详细的讨论。

2.3.3. 会话管理

2.3.3.1. 检测超时

你可以配置 Spring Security 检测失效的 session ID， 并把用户转发到对应的 URL。这可以通过 session-management 元素配置： <http> ... <session-management invalid-session-url="/sessionTimeout.htm" /> </http>

2.3.3.2. 同步会话控制

如果你希望限制单个用户只能登录到你的程序一次，Spring Security 通过添加下面简单的部分支持这个功能。 首先，你需要把下面的监听器添加到你的 web.xml 文件里，让 Spring Security 获得 session 生存周期事件： <listener> <listener-class> org.springframework.security.web.session.HttpSessionEventPublisher </listener-class> </listener> 然后，在你的 application context 加入如下部分：

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

```
  </http>
```

这将防止一个用户重复登录好几次-第二次登录会让第一次登录失效。 通常我们更想防止第二次登录，这时候我们可以使用

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

第二次登录将被阻止，通过"注入"，我们的意思是用户会被转发到 authentication-failure-url，如果使用了 form-based 登录。 如果第二次验证使用了其他非内置的机制，比如"remember-me"， 一个"未认证"(402)错误就会发送给客户端。 如果你希望使用一个错误页面替代，你可以在 session-management 中添加 session-authentication-error-url 属性。

如果你为 form-based 登录使用了自定义认证， 你就必须特别配置同步会话控制。更多的细节可以在 会话管理章节找到。

2.3.3.3. 防止 Session 固定攻击

Session 固定攻击是一个潜在危险，当一个恶意攻击者可以创建一个 session 访问一个网站的时候，然后说服另一个用户登录到同一个会话上（比如，发送给他们一个包含了 session 标识参数的链接）。 Spring Security 通过在用户登录时，创建一个新 session 来防止这个问题。 如果你不需要保护，或者它与其他一些需求冲突，你可以通过使用<http>中的 session-fixation-protection 属性来配置它的行为，它有三个选项

- migrateSession - 创建一个新 session，把原来 session 中所有属性复制到新 session 中。这是默认值。
- none - 什么也不做，继续使用原来的 session。
- newSession - 创建一个新的"干净的"session，不会复制 session 中的数据。

2.3.4. 对 OpenID 的支持

命名空间支持 OpenID 登录，替代普通的表单登录，或作为一种附加功能，只需要进行简单的修改：

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

你应该注册一个 OpenID 供应器（比如 myopenid.com），然后把用户信息添加到你的内存<user-service>中：

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

你应该可以使用 myopenid.com 网站登录来进行验证了。 也可能选择一个特定的 UserDetailsService bean 来使用 OpenID，通过设置元素。 查看上一节认证提供器获得更多信息。 请注意，上面用户配置中我们省略了密码属性，因为这里的用户数据只用来为数据读取数据。 内部会生成一个随机密码，放置我们使用用户数据时出现问题， 无论在你的配置的地方使用认证信息。

2.3.4.1. 属性交换

支持 OpenID 的 属性交换。 作为一个例子，下面的配置会尝试从 OpenID 提供器中获得 email 和全名， 这些会被应用程序使用到：

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

```
    </attribute-exchange>
  </openid-login>
```

每个 OpenID 的"type"属性是一个 URI，这是由特定的 schema 决定的， 在这个例子中是 http://axschema.org/。 如果一个属性必须为了成功认证而获取，可以设置 required。 确切的 schema 和对属性的支持会依赖于你使用的 OpenID 提供器。属性值作为认证过程的一部分返回， 可以使用下面的代码在后面的过程中获得：

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

OpenIDAttribute 包含的属性类型和获取的值 （或者在多属性情况下是多个值）。我们将看到更多关于 SecurityContextHolder 如何使用的信息， 只要我们在技术概述章节浏览核心 Spring Security 组件。

2.3.5. 添加你自己的 filter

如果你以前使用过 Spring Security，你就应该知道这个框架里维护了一个过滤器链，来提供服务。 你也许想把你自己的过滤器添加到链条的特定位置，或者使用一个 Spring Security 的过滤器，这个过滤器现在还没有在命名空间配置中进行支持（比如 CAS）。 或者你想要使用一个特定版本的标准命名空间过滤器，比如<form-login>创建的 UsernamePasswordAuthenticationFilter，从而获得一些额外的配置选项的优势，这些可以通过直接配置 bean 获得。你如何在命名空间配置里实现这些功能呢？过滤器链现在已经不能直接看到了。

过滤器顺序在使用命名空间的时候是被严格执行的。 当 application context 创建时，过滤器 bean 通过 namespace 的处理代码进行排序， 标准的 spring security 过滤器都有自己的假名和一个容易记忆的位置。

Note

在之前的版本中，排序是在过滤器实例创建后执行的，在 application context 的执行后的过程中。在 3.0+版本中，执行会在 bean 元元素级别被执行，在 bean 实例化之前。 这会影响到你如何添加自己的过滤器，实体过滤器列表 必须在解析<http>元素的过程中了解这些， 所以 3.0 中的语法变化的很明显。

有关创建过滤器的过滤器，假名和命名空间元素，属性可以在 Table 2.1,"标准过滤器假名和顺序"中找到。 过滤器按照次序排列在过滤器链中。

Table 2.1. 标准过滤器假名和顺序

假名	过滤器累	命名空间元素或属性
CHANNEL_FILTER	ChannelProcessingFilter	http/intercept-url@requires-channel
CONCURRENT_SESSION_FILTER	ConcurrentSessionFilter	session-management/concurrency-control
SECURITY_CONTEXT_FILTER	SecurityContextPersistenceFilter	http
LOGOUT_FILTER	LogoutFilter	http/logout
X509_FILTER	X509AuthenticationFilter	http/x509
PRE_AUTH_FILTER	AstractPreAuthenticatedProcessingFilter Subclasses	N/A
CAS_FILTER	CasAuthenticationFilter	N/A
FORM_LOGIN_FILTER	UsernamePasswordAuthenticationFilter	http/form-login
BASIC_AUTH_FILTER	BasicAuthenticationFilter	http/http-basic
SERVLET_API_SUPPO	SecurityContextHolderAwa	http/@servlet-api-provi

假名	过滤器累	命名空间元素或属性
RT_FILTER	reFilter	sion
REMEMBER_ME_FILTER	RememberMeAuthenticationFilter	http/remember-me
ANONYMOUS_FILTER	SessionManagementFilter	http/anonymous
SESSION_MANAGEMENT_FILTER	AnonymousAuthenticationFilter	session-management
EXCEPTION_TRANSLATION_FILTER	ExceptionTranslationFilter	http
FILTER_SECURITY_INTERCEPTOR	FilterSecurityInterceptor	http
SWITCH_USER_FILTER	SwitchUserAuthenticationFilter	N/A

你可以把你自己的过滤器添加到队列中，使用 custom-filter 元素，使用这些名字中的一个，来指定你的过滤器应该出现的位置：

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

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

你还可以使用 after 或 before 属性，如果你想把你的过滤器添加到队列中另一个过滤器的前面或后面。 可以分别在 position 属性使用"FIRST" 或 "LAST"来指定你想让你的过滤器出现在队列元素的前面或后面。

避免过滤器位置发生冲突

如果你插入了一个自定义的过滤器，而这个过滤器可能与命名空间自己创建的标准过滤器放在同一个位置上，这样首要的是你不要错误包含命名空间的版本信息。 避免使用 auto-config 属性，然后删除所有会创建你希望替换的过滤器的元素。

注意，你不能替换那些<http>元素自己使用而创建出的过滤器，比如 SecurityContextPersistenceFilter, ExceptionTranslationFilter 或 FilterSecurityInterceptor。

如果你替换了一个命名空间的过滤器，而这个过滤器需要一个验证入口点（比如，认证过程是通过一个未通过验证的用户访问受保护资源的尝试来触发的），你将也需要添加一个自定义的入口点 bean。

2.3.5.1. 设置自定义 AuthenticationEntryPoint

如果你没有通过命名空间，使用表单登陆，OpenID 或基本认证，你可能想定义一个认证过滤器，并使用传统 bean 语法定义一个入口点然后把它链接到命名空间里，就像我们已经看到的那样。 对应的 AuthenticationEntryPoint 可以使用<http>元素中的 entry-point-ref 属性来进行设置。

CAS 示例程序是一个在命名空间中使用自定义 bean 的好例子，包含这种语法。 如果你对认证入口点并不熟悉，可以在技术纵览章中找到关于它们的讨论。

2.4. 保护方法

从版本 2.0 开始 Spring Security 大幅改善了对你的服务层方法添加安全。 它提供对 JSR-250 安全注解的支持，这与框架提供的@secured 注解相似。 从 3.0 开始，你也可以使用新的基于表达式的注解。 你可以提供安全给单个 bean，使用 intercept-methods 来装饰 bean 的声明， 或者你可以控制多个 bean，通过实体服务层，使用 AspectJ 演示的切点功能。

2.4.1. <global-method-security>元素

这个元素用来在你的应用程序中启用基于安全的注解（通过在这个元素中设置响应的属性），也可以用来声明将要应用在你的实体 application context 中的安全切点组。 你

应该只定义一个<global-method-security>元素。 下面的声明同时启用 Spring Security 的@Secured 和 JSR-250 注解：

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

向一个方法（或一个类或一个接口）添加注解，会限制对这个方法的访问。 Spring Security 原生注解支持为方法定义一系列属性。 这些属性将传递给 AccessDecisionManager，进行决策：

```
  public interface BankService {


    @Secured("IS_AUTHENTICATED_ANONYMOUSLY")

    public Account readAccount(Long id);


    @Secured("IS_AUTHENTICATED_ANONYMOUSLY")

    public Account[] findAccounts();


    @Secured("ROLE_TELLER")

    public Account post(Account account, double amount);

  }
```

为了使用新的基于表达式的预付，你可以好似用

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

对应的代码将会是这样

```
   public interface BankService {
```

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

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

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

2.4.1.1. 使用 protect-pointcut 添加安全切点

protect-pointcut 是非常强大的，它让你可以用简单的声明对多个 bean 的进行安全声明。 参考下面的例子：

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

这样会保护 application context 中的符合条件的 bean 的所有方法，这些 bean 要在 com.mycompany 包下，类名以"Service"结尾。 ROLE_USER 的角色才能调用这些方法。

就像 URL 匹配一样，指定的匹配要放在切点队列的最前面，第一个匹配的表达式才会被用到。

2.5. 默认的 AccessDecisionManager

这章假设你有一些 Spring Security 权限控制有关的架构知识。 如果没有，你可以跳过这段，以后再来看，因为这章只是为了自定义的用户设置的，需要在简单基于角色安全的基础上加一些客户化的东西。

当你使用命名空间配置时，默认的 AccessDecisionManager 实例会自动注册，然后用来为方法调用和 web URL 访问做验证，这些都是基于你设置的 intercept-url 和 protect-pointcut 权限属性内容（和注解中的内容，如果你使用注解控制方法的权限）。

默认的策略是使用一个 AffirmativeBased AccessDecisionManager ，以及 RoleVoter 和 AuthenticatedVoter。 可以在 authorization 中获得更多信息。

2.5.1. 自定义 AccessDecisionManager

如果你需要使用一个更复杂的访问控制策略，把它设置给方法和 web 安全是很简单的。

对于方法安全，你可以设置 global-security 里的 access-decision-manager-ref 属性，用对应 AccessDecisionManager bean 在 application context 里的 id：

```
  <global-method-security

access-decision-manager-ref="myAccessDecisionManagerBean">

    ...

  </global-method-security>
```

web 安全安全的语法也是一样，但是放在 http 元素里：

```
  <http

access-decision-manager-ref="myAccessDecisionManagerBean">

    ...

  </http>
```

2.6. 验证管理器和命名空间

主要接口提供了验证服务在 Spring Security 中， 是 AuthenticationManager。 通常是 Spring Security 中 ProviderManager 类的一个实例， 如果你以前使用过框架，你可能已经很熟悉了。 如果没有，它会在稍后被提及，在#tech-intro-authentication。 bean 实例被使用 authentication-manager 命名空间元素注册。 你不能好似用一个自定义的 AuthenticationManager 如果你使用 HTTp 或方法安全，在命名空间中，但是它不应该是一个问题， 因为你完全控制了使用的 AuthenticationProvider。

你可能注册额外的 AuthenticationProviderbean， 在 ProviderManager 中，你可以使用 <authentication-provider>做这些事情，使用 ref 属性， 这个属性的值，是你希望添加的 provider 的 bean 的名字，比如：

```
  <authentication-manager>

    <authentication-provider ref="casAuthenticationProvider"/>

  </authentication-manager>

  <bean id="casAuthenticationProvider"



class="org.springframework.security.cas.authentication.CasAuthe

nticationProvider">

    <security:custom-authentication-provider />

    ...

  </bean>
```

另一个常见的需求是，上下文中的另一个 bean 可能需要引用 AuthenticationManager。 你可以为 AuthenticationManager 注册一个别名，然后在 application context 的其他地方使用这个名字。

```
  <security:authentication-manager

alias="authenticationManager">

     ...

  </security:authentication-manager>
```

```
  <bean id="customizedFormLoginFilter"

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

示例程序

项目中包含了很多 web 实例程序。 为了不让下载包变得太大，我们只把"tutorial"和"contacts"两个例子放到了 zip 发布包里。你可以自己编译部署它们,也可以从 Maven 中央资源库里获得单独的 war 文件。 我们建议你使用前一种方法。 你可以按照简介里的介绍获得源代码，使用 maven 编译它们也很简单。 如果你需要的话，可以在 http://www.springsource.org/security/ 网站上找到更多信息。

3.1. Tutorial 示例

这个 tutorial 示例是带你入门的很好的一个基础例子。 它完全使用了简单命名空间配置。 编译好的应用就放在 zip 发布包中，已经准备好发布到你的 web 容器中（spring-security-samples-tutorial-2.0.x.war）。 使用了 form-based 验证机制，与常用的 remember-me 验证提供器相结合，自动使用 cookie 记录登录信息。

我们推荐你从 tutorial 例子开始，因为 XML 非常小，也很容易看懂。 更重要的是，你很容易就可以把这个 XML 文件（和它对应的 web.xml 入口）添加到你的程序中。 只有做基本集成成功的时候，我们建议你试着添加方法验证和领域对象安全。

3.2. Contacts

Contacts 例子，是一个很高级的例子，它在基本程序安全上附加了领域对象的访问控制列表，演示了更多强大的功能。

要发布它，先把 Spring Security 发布中的 war 文件按复制到你的容器的 webapps 目录下。这个 war 文件应该叫做 spring-security-samples-contacts-2.0.0.war(后边的版本号，很大程度上依赖于你使用的发布版本）。

在启动你的容器之后，检测一下程序是不是加载了，访问 http://localhost:8080/contacts（或是其他你把 war 发布后，对应于你 web 容器的 URL）。

下一步，点击"Debug"。 你将看到需要登录的提示，这页上会有一些测试用的用户名和密码。 随便使用其中的一个通过认证，就会看到结果页面。 它应该会包含下面这样的一段成功信息：

Security Debug Information

Authentication object is of type:
org.springframework.security.authentication.UsernamePasswordAuthenticationToken

Authentication object as a String:

org.springframework.security.authentication.UsernamePasswordAuthenticationToken@1f127853:
Principal: org.springframework.security.core.userdetails.User@b07ed00:
Username: rod; \
Password: [PROTECTED]; Enabled: true; AccountNonExpired: true;
credentialsNonExpired: true; AccountNonLocked: true; \
Granted Authorities: ROLE_SUPERVISOR, ROLE_USER; \
Password: [PROTECTED]; Authenticated: true; \
Details: org.springframework.security.web.authentication.WebAuthenticationDetails@0: \
RemoteIpAddress: 127.0.0.1; SessionId: 8fkp8t83ohar; \
Granted Authorities: ROLE_SUPERVISOR, ROLE_USER

Authentication object holds the following granted authorities:

ROLE_SUPERVISOR (getAuthority(): ROLE_SUPERVISOR)
ROLE_USER (getAuthority(): ROLE_USER)

Success! Your web filters appear to be properly configured!

一旦你成功的看到了上面的信息，就可以返回例子程序的主页，点击"Manage"了。 然后你就可以尝试这个程序了。 注意，只有当前登录的用户对应的联系信息会显示出来，而且只有 ROLE_SUPERVISOR 权限的用户可以授权删除他们的联系信息。 在这场景后面，MethodSecurityInterceptor 保护着业务对象。

陈程序允许你修改访问控制列表，分配不同的联系方式。 确保自己好好试用过，看看程序里的上下文 XML 文件，搞明白它是如何运行的。

3.3. LDAP 例子

LDAP 例子程序提供了一个基础配置，同时使用命名空间配置和使用传统方式 bean 的配置方式，这两种配置方式都写在 application context 文件里。 这意味着，在这个程序里，其实是配置了两个定义验证提供器。

3.4. CAS 例子

CAS 示例要求你同时运行 CAS 服务器和 CAS 客户端。 它没有包含在发布包里，你应该使用简介中的介绍来获得源代码。你可以在 sample/cas 目录下找到对应的文件。这里还有一个 Readme.txt 文件，解释如何从源代码树中直接运行服务器和客户端，提供完全的 SSL 支持。 你应该下载 CAS 服务器 web 程序（一个 war 文件）从 CAS 网站，把它放到 samples/cas/server 目录下。

3.5. Pre-Authentication 例子

这个例子演示了如何从 pre-authentication 框架绑定 bean，从 J2EE 容器中获得有用的登录信息。 用户名和角色是由容器设置的。

代码在 samples/preauth 目录下。

Spring Security 社区

4.1. 任务跟踪

Spring Security 使用 JIRA 管理 bug 报告和扩充请求。 如果你发现一个 bug，请使用 JIRA 提交一个报告。 不要把它放到支持论坛上，邮件列表里，或者直接发邮件给项目开发者。 这样做法是特设的，我们更愿意使用更正式的方式管理 bug。

如果有可能，最好为你的任务报告提供一个 Junit 单元测试，演示每一种不正确的行为。或者，更好的是，提供一个不定来修正这个问题。 一般来说，扩充也也可以提交到任务跟踪系统里，虽然我们只接受提供了对应的单元测试的扩充请求。 因为保证项目的测试覆盖率是非常必要的，它需要适当的进行维护。

你可以访问任务跟踪的网址 http://jira.springsource.org/browse/SEC。

4.2. 成为参与者

我们欢迎你加入到 Spring Security 项目中来。 这里有很多贡献的方式，包括在论坛上阅读别人的帖子发表回复，写新代码，提升旧代码，帮助写文档，开发例子或指南，或简单的提供建议。

4.3. 更多信息

欢迎大家为 Spring Security 提出问题或评论。 你可以使用 Spring 社区论坛网址 http://forum.springsource.org 同框架的其他用户讨论 Spring Security。 记得使用 JIRA 提交 bug，这部分在上面提到过了。

Part II. 结构和实现

当你熟悉了如何设置和使用基于命名空间配置的应用， 你也许希望了解更多关于框架是如何在命名空间的外观下实际运行的。 和大多数软件一样，Spring Security 有一系列的中央接口， 类和抽象概念，贯穿整个框架。 在指南的这个部分，我们会观察它们是如何在一起工作的， 同 Spring Security 支持验证和权限控制。

技术概述

5.1. 运行环境

Spring Security 3.0 需要运行在 Java 5.0 或更高版本环境上。 因为 Spring Security 的目标是自己容器内管理， 所以不需要为你的 Java 运行环境进行什么特别的配置。 特别是，不需要特别配置一个 Java Authentication and Authorization Service (JAAS)政策文件，也不需要把 Spring Security 放到 server 的 classLoader 下。

相同的，如果你使用了一个 EJB 容器或者是 Servlet 容器，都不需要把任何 特定的配置文件放到什么地方，也不需要把 Spring Security 放到 server 的 classloader 下。所有必须的文件都可以配置在你的应用中。

这些设计确保了发布时的最大轻便性， 你可以简单把你的目标文件（JAR 或 WAR 或 EAR）从一个系统复制到另一个系统， 它会立即正常工作。

5.2. 核心组件

在 Spring Security 3.0 中，spring-security-corejar 的内容 已经被缩减到最小。它不再包含任何与 web 应用安全， LDAP 或命名空间相关的代码。我们会看一下这里，看看你在核心模块中找到的 Java 类型。它们展示了框架的构建基础， 所以如果你需要超越简单的命名空间配置， 那么理解它们就是很重要的，即便你不需要直接 操作他们。

5.2.1. SecurityContextHolder, SecurityContext 和 Authentication 对象

最基础的对象就是 SecurityContextHolder。我们把当前应用程序的当前安全环境的细节存储到它里边了， 它也包含了应用当前使用的主体细节。 默认情况下，SecurityContextHolder 使用 ThreadLocal 存储这些信息， 这意味着，安全环境在同一个线程执行的方法一直是有效的， 即使这个安全环境没有作为一个方法参数传递到那些方法里。 这种情况下使用 ThreadLocal 是非常安全的， 只要记得在处理完当前主体的请求以后，把这个线程清除就行了。 当然，Spring Security 自动帮你管理这一切了， 你就不用担心什么了。

有些程序并不适合使用 ThreadLocal， 因为它们处理线程的特殊方法。比如，swing 客户端也许希望 JVM 里的所有线程都使用同一个安全环境。SecurityContextHolder 可以使用一个策略进行配置 在启动时，指定你想让上下文怎样被保存。对于一个单独的应用系统，你可以使用 SecurityContextHolder.MODE_GLOBAL 策略。 其他程序可能想让一个线程创建的线

程也使用相同的安全主体。 这时可以使用 SecurityContextHolder.MODE_INHERITABLETHREADLOCAL。 想要修改默认的 SecurityContextHolder.MODE_THREADLOCAL 模式，可以使用两种方法。 第一个是设置系统属性。另一个是调用 SecurityContextHolder 的静态方法。大多数程序不需要修改默认值， 但是如果你需要做修改，先看一下 SecurityContextHolder 的 JavaDoc 中的详细信息。

5.2.1.1. 获得当前用户的信息

我们把安全主体和系统交互的信息都保存在 SecurityContextHolder 中了。 Spring Security 使用一个 Authentication 对应来表现这些信息。 虽然你通常不需要自己创建一个 Authentication 对象，但是常见的情况是，用户查询 Authentication 对象。你可以使用下面的代码 - 在你程序中的任何位置 - 来获得已认证用户的名字， 比如：

```
Object principal =
SecurityContextHolder.getContext().getAuthentication().getPrinc
ipal();

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

调用 getContext()返回的对象是一个 SecurityContext 接口的实例。这个对象是保存在 thread-local 中的。如我们下面看到的，大多数 Spring Security 的验证机制 都返回一个 UserDetails 的实例 作为主体。

5.2.2. UserDetailsService

从上面的代码片段中还可以看出另一件事，就是你可以从 Authentication 对象中获得安全主体。这个安全主体就是一个对象。大多数情况下，可以强制转换成 UserDetails 对象。 UserDetails 是一个 Spring Security 的核心接口。 它代表一个主体，是扩展的，而且是为特定程序服务的。 想一下 UserDetails 章节，在你自己的用户数据库和如何把 Spring Security 需要的数据放到 SecurityContextHolder 里。 为了让你自己的用户数据库起作用，我们常常把 UserDetails 转换成你系统提供的类，这样你就可以直接调用业务相关的方法了（比如 getEmail(),getEmployeeNumber()等等）。

现在，你可能想知道，我应该什么时候提供这个 UserDetails 对象呢？ 我怎么做呢？我想你说这个东西是声明式的，我不需要写任何代码，怎么办？ 简单的回答是，这里

有一个特殊的接口，叫 UserDetailsService。 这个接口里的唯一一个方法，接收 String 类型的用户名参数，返回 UserDetails：

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

这是获得从 Spring Security 中获得用户信息的最常用方法，你会看到它在框架中一直被用到。 当需要获得一个用户的信息的时候。

当成功通过验证时，UserDetails 会被用来 建立 Authentication 对象，保存在 SecurityContextHolder 里 。 (更多的信息可以参考下面的 #tech-intro-authentication-mgr) 。 好消息是我们提供了好几个 UserDetailsService 实现，其中一个使用了 内存中的 map(InMemoryDaoImpl)另一个而是用了 JDBC (JdbcDaoImpl)。 虽然，大多数用户倾向于写自己的，使用这些实现常常放到已有的数据访问对象（DAO）上，表示它们的雇员，客户或其他企业应用中的用户。 记住这个优势，无论你用什么 UserDetailsService 返回的数据都可以通过 SecurityContextHolder 获得，就像上面的代码片段讲的一样。

5.2.3. GrantedAuthority

除了主体，另一个 Authentication 提供的重要方法是 getAuthorities()。 这个方法提供了 GrantedAuthority 对象数组。 毫无疑问，GrantedAuthority 是赋予到主体的权限。 这些权限通常使用角色表示，比如 ROLE_ADMINISTRATOR 或 ROLE_HR_SUPERVISOR。 这些角色会在后面，对 web 验证，方法验证和领域对象验证进行配置。 Spring Security 的其他部分用来拦截这些权限，期望他们被表现出现。 GrantedAuthority 对象通常使用 UserDetailsService 读取的。

通常情况下，GrantedAuthority 对象是应用程序范围下的授权。 它们不会特意分配给一个特定的领域对象。 因此，你不能设置一个 GrantedAuthority，让它有权限展示编号 54 的 Employee 对象，因为如果有成千上网的这种授权，你会很快用光内存（或者，至少，导致程序花费大量时间去验证一个用户）。 当然，Spring Security 被明确设计成处理常见的需求，但是你最好别因为这个目的使用项目领域模型安全功能。

5.2.4. 小结

简单回顾一下，Spring Security 主要是由一下几部分组成的：

- SecurityContextHolder，提供几种访问 SecurityContext 的方式。
- SecurityContext，保存 Authentication 信息，和请求对应的安全信息。
- HttpSessionContextIntegrationFilter，为了在不同请求使用，把 SecurityContext 保存到 HttpSession 里。
- Authentication，展示 Spring Security 特定的主体。
- GrantedAuthority，反应，在应用程序范围你，赋予主体的权限。
- UserDetails，通过你的应用 DAO，提供必要的信息，构建 Authentication 对象。

- UserDetailsService，创建一个 UserDetails，传递一个 String 类型的用户名（或者证书 ID 或其他）。

现在，你应该对这种重复使用的组件有一些了解了。 让我们贴近看一下验证的过程。

5.3. 验证

Spring Security 可以用在多种不同的验证环境下。 我们推荐人们使用 Spring Security 进行验证，而不是与现存的容器管理验证相结合， 然而这种方式也是被支持的 - 作为与你自己的 验证系统相整合的一种方式。

5.3.1. 什么是 Spring Security 的验证呢？

让我们考虑一种标准的验证场景，每个人都很熟悉的那种。

- 一个用户想使用一个账号和密码进行登陆。
- 系统（成功的）验证了密码对于这个用户名 是正确的。
- 这个用户对应的信息呗获取 （他们的角色列表以及等等）。
- 为用户建立一个安全环境。
- 用户会执行一些操作，这些都是潜在被 权限控制机制所保护的，通过对操作的授权， 使用当前的安全环境信息。

前三个项目执行了验证过程，所以我们可以看一下 Spring Security 的作用。

- 用户名和密码被获得，并进行比对， 在一个 UsernamePasswordAuthenticationToken 的实例中（它是 Authentication 接口的一个实例， 我们在之前已经见过了）。
- 这个标志被发送给一个 AuthenticationManager 的实例进行校验。
- AuthenticationManager 返回一个完全的 Authentication 实例， 在成功校验后。
- 安全环境被建立，通过调用 SecurityContextHolder.getContext().setAuthentication(...)， 传递到返回的验证对象中。

从这一点开始，用户已经通过校验了。让我们 看一些代码作为例子。

```
import org.springframework.security.authentication.*;

import org.springframework.security.core.*;

import

org.springframework.security.core.authority.GrantedAuthorityImp

l;
```

```
import
org.springframework.security.core.context.SecurityContextHolder
;


public class AuthenticationExample {
  private    static    AuthenticationManager    am    =    new
SampleAuthenticationManager();


  public static void 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 {
  static final List<GrantedAuthority> AUTHORITIES = new
ArrayList<GrantedAuthority>();

  static {
    AUTHORITIES.add(new GrantedAuthorityImpl("ROLE_USER"));
```

```
  }

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

这里 我们写了一些程序，询问用户输入一个用户名和密码， 然后执行上面的顺序。我们实现的 AuthenticationManager 会验证所有用户名和密码一样的用户。 它为每个永固分配一个单独的角色。上面输出的信息 将会像这样：

```
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.UsernamePasswordAut

henticationToken@441d0230: \

 Principal: bob; Password: [PROTECTED]; \

 Authenticated: true; Details: null; \

 Granted Authorities: ROLE_USER
```

注意，你没必要写这些代码。这些处理都是发生在内部的， 比如在一个 web 验证过滤器中。我们只是使用了这些代码， 来演示真实情况下的问题，Spring Security 提供了一个简单的答案。 一个用户被验证，当 SecurityContextHolder 包含了完整的 Authentiation 对象。

5.3.2. 直接设置 SecurityContextHolder 的内容

实际上， Spring Security 不知道你怎么把 Authentication 对象放到 SecurityContextHolder 里。唯一关键的要求是 SecurityContextHolder 包含了一个 Authentication 表示了一个主体， 在 AbstractSecurityInterceptor 之前（我们以后会看到更多） 需要验证一个用户操作。

你可以（许多人都这样做）写自己的过滤器，或 MVC 控制器来提供验证系统， 不基于 Spring Security。比如 你可能使用容器管理的验证，让当前用户有效 在 TheadLocal 或 JNDI 位置。或者你可能为一个公司工作，你没有什么控制力。 这种情况下，使用 Spring Security 很简单， 还是提供验证功能。 你需要做的是些一个过滤器（或什么设备） 从一个地方读取第三方用户信息， 构建一个 Spring Security 特定的 Authentication 对象， 把它放到 SecurityContextHolder 里。

如果你想知道 AuthenticationManager 是如何实现的， 我们会在***看到。

5.4. 在 web 应用中验证

现在让我们研究一下情景，当我们在 web 应用中使用 Spring Security （不使用 web.xml 安全）。一个用户 如何验证，安全环境如何创建？

考虑一个典型的 web 应用的验证过程：

- 你访问主页，点击链接。
- 一个请求发送给服务器，服务器决定你是否在 请求一个被保护的资源。

- 如果你还没有授权，服务器发回一个相应，提示你必须登录。 响应会是一个 HTTP 响应代码， 或重定向到特定的 web 页面。
- 基于验证机制，你的浏览器会重定向到特殊的 web 页面， 所以你可以填写表单，或者浏览器会验证你的身份 （通过一个 BASIC 验证对话框，一个 cookie，一个 X.509 验证，等等）。
- 浏览器会发送回一个响应到服务器。这会是一个 HTTP POST 包含你填写的表单中的内容，或者一个 HTTP 头部 包含你的验证细节。
- 下一步，服务器决定，当前证书是否有效。 如果它们有效，下一步会执行。如果它们无效，通常你的浏览器会 被询问再试一次（所以 initial 返回上两步）。
- 你的原始请求会引发验证过程。 希望你验证了获得了授予的权限来访问被保护的资源。 如果你完允许访问，请求会成功。 否则，你会收到一个 HTTP 错误代码 403，意思是"拒绝访问"。

Spring Security 拥有不同的泪，对应很多常用的上面所说的步骤。 主要的部分（使用的次序）是 ExceptionTranslationFilter， 一个 AuthenticationEntryPoint 和一个 "验证机制"， 对应着 AuthenticationManager 的调用 我们在上一章见过。

5.4.1. ExceptionTranslationFilter

ExceptionTranslationFilter 是一个 Spring Security 过滤器 负责检测任何一个 Spring Security 抛出的异常。这些异常会被 AbstractSecurityInterceptor 抛出，这是一个验证服务的主要提供器。我们会在下一章讨论 AbstractSecurityInterceptor， 而现在我们需要知道它产生 Java 异常， 不知道 HTTP，也不知道如何验证一个主体。 对应的 ExceptionTranslationFilter 负责这个服务， 特别负责返回错误代码 403（如果主体已经通过授权， 但是权限不足 - 像上面的第七步）， 或者启动一个 AuthenticationEntryPoint（如果 主体还没有授权，因此我们会进入上面的第三步）。

5.4.2. AuthenticationEntryPoint

AuthenticationEntryPoint 负责上面的步骤三。 像你想的那样，每个 web 应用会有一个默认的验证策略 （好，这可能像其他东西一样在 Spring Security 里配置， 但现在让我们保持简单）。每个主要的验证系统会有他们自己的 AuthenticationEntryPoint 实现， 典型的执行一个动作，描述在第三步。

5.4.3. 验证机制

一旦你的浏览器提交了你的验证证书（像 HTTP 表单 POST 或者 HTTP 头） 这些需要一些服务器的东西保存这些权限信息。 但是现在我们进入上面的第六步。 在 Spring Security 中我们有一个特定的名称，为了收集验证信息的操作。从一个用户代码中（通常是浏览器），引用它作为一个"验证机制"。 例子是基于表单的登录和 BASIC 验证。一旦验证细节被从用户代理处收集到， 一个 Authentication 请求对象就会被建立，然后放到 AuthenticationManager。

在验证机制获得完全的 Authentication 后，它会认为请求合法，把 Authentication 放到 SecurityContextHolder 里，然后让原始请求重试 （上面第七步）。如果，其他可能， AuthenticationManager 拒绝了请求， 请求机制会让用户代理重试（上面第二步）。

5.4.4. 在请求之间保存 SecurityContext。

依照应用类型，这里需要一个策略，在用户操作之间保存安全环境。在一个典型的 web 应用中，一个用户日志，一次或顺序被它的 session id。服务器缓存 主体信息在 session 整个过程中，在 Spring Security 中，保存 SecurityContext，从请求失败 SecurityContextPersistenceFilter，默认保存到 HttpSession 里的一个属性，在 HTTP 请求之间。它重新保存环境到 SecurityContextHolder，为每个请求。然后为每个请求清空 SecurityContextHolder。你不应该为了安全目的，直接操作 HttpSession。这里有简单的方法实现 - 一直使用 SecurityContextHolder 代替。

很多其他类型的应用（比如，一个无状态的 REST web 服务）不会使用 HTTP 会话，会在每次请求时，重新验证。然而，这对 SecurityContextPersistenceFilter 也很重要，确保包含在 SecurityContextHolder 中，在每次请求后清空。

Note

在一个单一会话接收同步请求的应用里，相同的 SecurityContext 实例会在线程之间共享。即使使用一个 ThreadLocal，也是使用了来自 HttpSession 的相同实例。如果你希望暂时改变一个线程的上下文 就会造成影响。如果你只是使用 SecurityContextHolder.getContext().setAuthentication(anAuthentication)，然后 Authentication 对象会反应到 *所有*并发线程，共享相同的 SecurityContext 实例。你可以自定义 SecurityContextPersistenceFilter 的行为来创建完全新的一个线程 避免影响其他的。还可以选择的是，你可以创建一个新实例，只在你暂时修改上下文的时候。 这个方法 SecurityContextHolder.createEmptyContext()总会返回一个新的上下文实例。

5.5. Spring Security 中的访问控制（验证）

主要接口，负责访问控制的决定，在 Spring Security 中是 AccessDecisionMananger。它有一个 decide 方法，可以获得一个 Authentication 对象。展示主体的请求权限，一个"secure object"（看下边）和一个安全元数据属性队列，为对象提供了（比如一个角色列表，为访问被授予的请求）。

5.5.1. 安全和 AOP 建议

如果你熟悉 AOP 的话，就会知道有几种不同的拦截方式：之前，之后，抛异常和环绕。其中环绕是非常有用的，因为 advisor 可以决定是否执行这个方法，是否修改返回的结果，是否抛出异常。 Spring Security 为方法调用提供了一个环绕 advice，就像 web 请求一样。我们使用 Spring 的标准 AOP 支持制作了一个处理方法调用的环绕 advice，我们使用标准 filter 建立了对 web 请求的环绕 advice。

对那些不熟悉 AOP 的人，需要理解的关键问题是 Spring Security 可以帮助你保护方法的调用，就像保护 web 请求一样。大多数人对保护服务层里的安全方法非常按兴趣。这是因为在目前这一代 J2EE 程序里，服务器放了更多业务相关的逻辑（需要澄清，作者不建议这种设计方法，作为替代的，而是应该使用 DTO，集会，门面和透明持久模式压缩领域对象，但是使用贫血领域对象是当前的主流思路，所以我们还是会在这里讨论它）。 如果你只是需要保护服务层的方法调用，Spring 标准 AOP 平台（一般被称作 AOP 联盟）就够了。 如果你想直接保护领域对象，你会发现 AspectJ 非常值得考虑。

可以选择使用 AspectJ 还是 Spring AOP 处理方法验证，或者你可以选择使用 filter 处理 web 请求验证。 你可以不选，选择其中一个，选择两个，或者三个都选。 主流的应用是处理一些 web 请求验证，再结合一些在服务层里的 Spring AOP 方法调用验证。

5.5.2. 安全对象和 AbstractSecurityInterceptor

所以，什么是"secure object"？ Spring Security 使用应用任何对象，可以被安全控制（比如一个验证决定）提供到它上面。 最常见的例子是方法调用和 web 请求。

Spring Security 支持的每个安全对象类型都有它自己的类型，它们都是 AbstractSecurityInterceptor 的子类。 很重要的是，如果主体是已经通过了验证，在 AbstractSecurityInterceptor 被调用的时候，SecurityContextHolder 将会包含一个有效的 Authentication。

AbstractSecurityInterceptor 提供了一套一致的工作流程，来处理对安全对象的请求，通常是：

- 查找当前请求里分配的"配置属性"。
- 把安全对象，当前的 Authentication 和配置属性，提交给 AccessDecisionManager，来进行以此认证决定。
- 有可能在调用的过程中，对 Authentication 进行修改。
- 允许安全对象进行处理（假设访问被允许了）。
- 在调用返回的时候执行配置的 AfterInvocationManager。

5.5.2.1. 配置属性是什么？

一个"配置属性"可以看做是一个字符串，它对于 AbstractSecurityInterceptor 使用的类是有特殊含义的。 它们通过框架中的 ConfigAttribute 接口表现。 它们可能是简单的角色名称或拥有更复杂的含义，这就与 AccessDecisionManager 实现的先进程度有关了。 AbstractSecurityInterceptor 和配置在一起的 SecurityMetadataSource 用来为一个安全对象搜索属性。 通常这个属性对用户是不可见的。 配置属性将以注解的方式设置在受保护方法上，或者作为受保护 URL 的访问属性。 比如，当我们查看一些像 <intercept-url pattern='/secure/**' access='ROLE_A,ROLE_B'/> 在命名空间介绍里，这就是在说这些配置属性 ROLE_A 和 ROLE_B 应用到 web 请求匹配到指定的模式中。实际上，使用默认的 AccessDecisionManager 配置，这意味着任何人 拥有 GrantedAuthority 匹配任何这些两个属性中的一个 会被允许访问。严格意义上，他们只是树形，解释是基于 AccessDecisionManager 实现的。 前缀 ROLE_的使用标记了这些属性是角色，会被 Spring Security 的 RoleVoter 处理。它只与基于角色的 AccessDecisionManager 有关。我们会在 验证章节看到 AccessDecisionManager 是如何实现的。

5.5.2.2. RunAsManager

假设 AccessDecisionManager 决定允许执行这个请求，AbstractSecurityInterceptor 会正常执行这个请求。 话虽如此，罕见情况下，用户可能需要把 SecurityContext 的 Authentication 换成另一个 Authentication，通过访问 RunAsManager。 这也许在，有原因，不常见的情况下有用，比如，服务层方法需要调用远程系统，表现不同的身份。 因为 Spring Security 自动传播安全身份，

从一个服务器到另一个（假设你使用了配置好的 RMI 或者 HttpInvoker 远程调用协议客户端），就可以用到它了。

5.5.2.3. AfterInvocationManager

按照下面安全对象执行和返回的方式-可能意味着完全的方法调用或过滤器链的执行。这种状态下 AbstractSecurityInterceptor 对有可能修改返回对象感兴趣。 你可能想让它发生，因为验证决定不能"关于如何在"一个安全对象调用。 高可插拔性，AbstractSecurityInterceptor 通过控制 AfterInvocationManager,实际上在需要的时候，修改对象。 这里类实际上可能替换对象，或者抛出异常，或者什么也不做。

AbstractSecurityInterceptor 和相关对象展示在 Figure 5.1, "关键"secure object"模型"中。

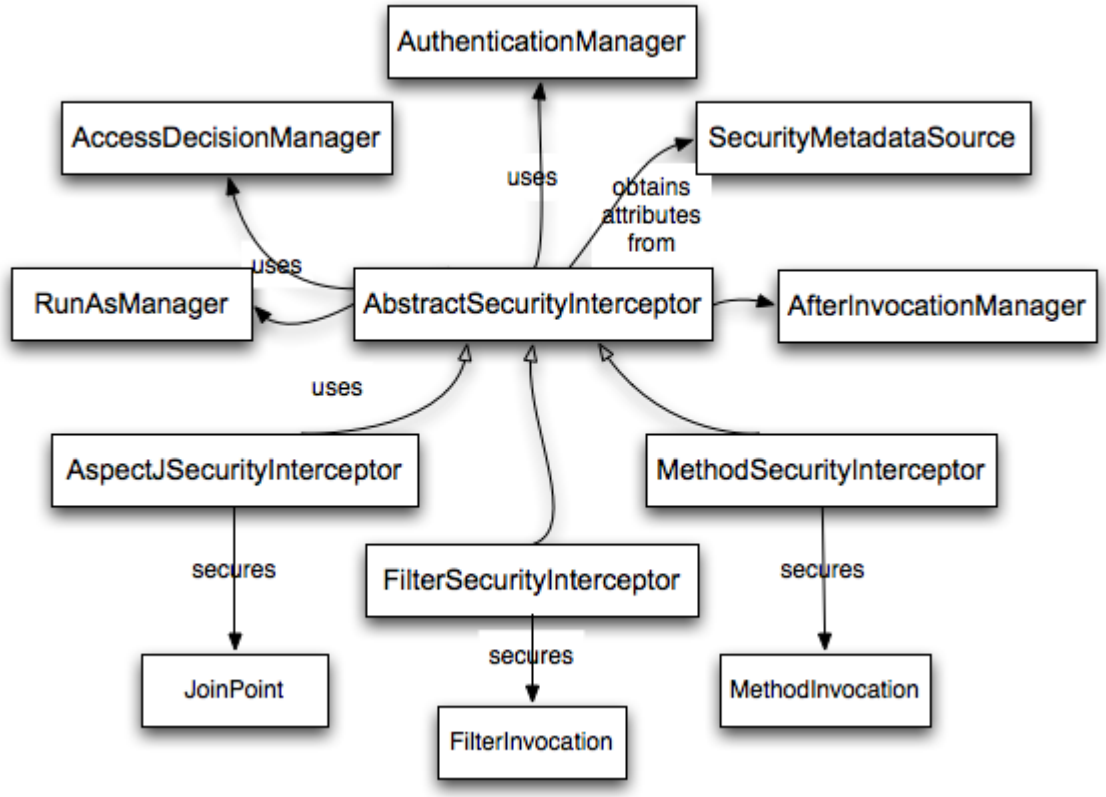

Figure 5.1. 关键"secure object"模型

5.5.2.4. 扩展安全对象模型

只有开发者才会关心使用全心的方法，进行拦截和验证请求，将直接使用安全方法。 比如，可能新建一个安全方法，控制对消息系统的权限。 安全需要的任何事情，也可以提供一种拦截的方法（好像 AOP 的环绕 advice 语法那样）有可能在安全对象里处理。这样说的话，大多数 Spring 应用简单拥有三种当前支持的安全类型（AOP 联盟的 MethodInvocation，AspectJ JoinPoint 和 web 请求 FilterInterceptor）完全透明的。

5.6. 国际化

Spring Security 支持异常信息的国际化，最终用户希望看到这些信息。 如果你的应用被设计成给讲英语的用户的，你不需要做任何事情， 因为默认情况下 Spring Security 的信息都是引用的。如果你需要支持其他语言。 你所需要做的事情都包含在这一章节中的。

所有的异常信息都支持国际化，包括验证失败和访问被拒绝的相关信息（授权失败）。应该被开发者和系统开发者关注（包括不正确的属性，接口契约，使用非法构造方法，开始时间校验，调试级日志等等）的异常和日志没有被国际化，而是使用英语硬编码到 Spring Security 的代码中。

从 spring-security-core-xx.jar 中 ， 你 可 以 找 到 org.springframework.security 包下，包含了一些 messages.properties 文件，这应该引用到你的 ApplicationContext 中，因为 Spring Security 的类 都实现了 spring 的 MessageSourceAware 接口， 期待的信息处理器会在 application context 启动的时候注入进来。 通常所有你需要做的就是在你的 application context 中注册一个 bean 来引用这些信息。 下面是一个例子：

```
<bean id="messageSource"



class="org.springframework.context.support.ReloadableResourceBu

ndleMessageSource">

  <property                                          name="basename"

value="org/springframework/security/messages"/>

</bean>
```

messages.properties 是按照标准资源束命名的，里边包括了 Spring security 所使用的默认语言的信息。 默认的文件是英文的。如果你没有注册一个信息源，Spring Security 也会正常工作， 并使用硬编码的英文版本的信息。

如果你想自定义 messages.properties 文件，或者支持其他语言， 你需要复制这个文件，正确的把它重新命名，再把它注册到 bean 定义中。 这个文件中并没有太多的信息。所以国际化应该不是很繁重的工作。 如果你国际化了这个文件，请考虑一下把你的工作和社区分享，通过记录一个 JIRA 任务 把你翻译的 messages.properties 版本作为一个附件发送上去。

围 绕 国 际 化 的 讨 论 ， spring 的 ThreadLocal 是 org.springframework.context.i18n.LocaleContextHolder 。 你 应 该 把 LocaleContextHolder 设置成为每个用户对应的 Locale。 Spring Security 会尝试从信息源中寻找信息，根据 ThreadLocal 中获得 的 Locale。请参考 Spring 的文档，来获得更多使用 LocaleContextHolder 的信息。

Chapter 6. 核心服务

现在，我们对 Spring Security 的架构和核心类有了高层次的了解了， 让我们近距离看看这些核心接口和他们的实现， 特别是 AuthenticationManager，UserDetailsService 和 AccessDecisionManager。 它们的信息都在这个文档的后面，所以重要的是我们要知道如何配置，如何操作。

6.1. The AuthenticationManager, ProviderManager 和 AuthenticationProviders

AuthenticationManager 只是一个接口，所以呢，它的实现 可以让我们随便选择，但是实际上它是如何工作的呢？ 如果我们需要检查多个授权数据库或者将不同的授权服务结合起来，比如数据库和 lDAP 服务器？

在 Spring Security 中的默认实现是 ProviderManager 不只是处理授权请求自己，它委派了一系列配置好的 AuthenticationProvider， 每个按照顺序查看它是否可以执行验证。每个供应器会跑出一个异常，或者返回一个完整的 Authentication 对象。要记得我们的好朋友，UserDetails 和、 UserDetailsService。 如果不记得了，返回到前面的章节刷新一下你的记忆。最常用的方式是验证一个授权请求读取 对应的 UserDetails，并检查用户录入的密码。 这是通过 DaoAuthenticationProvider 实现的(见下面)，加载的 UserDetails 对象 - 特别是包含的 GrantedAuthority - 会在建立 Authentication 时使用，这回返回一个成功验证，保存到 SecurityContext 中。

如果你使用了命名空间,一个 ProviderMananger 的实例会被创建 并在内部进行维护，你可以使用命名空间验证元素，或给一个 bean 添加一个<custom-authentication-provider>元素。 （参考命名空间章节）。 在这里，你不应该在你的 application context 中声明一个 ProviderManager bean。然而，如果你没有使用命名空间，你应该像下面 这样进行声明：

```
<bean id="authenticationManager"



class="org.springframework.security.authentication.ProviderMana

ger">

  <property name="providers">

    <list>
```

```
      <ref local="daoAuthenticationProvider"/>

      <ref local="anonymousAuthenticationProvider"/>

      <ref local="ldapAuthenticationProvider"/>

    </list>

  </property>

</bean>
```

在上面的例子中，我们有三个供应器。它们按照顺序显示（使用 List 实现）， 每个供应器能够尝试进行授权，或通过返回 null 跳过授权。 如果所有的实现都返回 null。ProviderManager 会跑出一个 ProviderNotFoundException 异常。 如果你对链状供应器感兴趣，请参考 ProviderManager 的 javadoc。

验证机制，比如表单登陆处理过滤器被注入一个 ProviderManager， 会被用来处理它们的认证请求。你需要的供应器有时需要被认证机制内部改变的，当在其他时候， 他们会以来一个特定的认证机制，比如 DaoAuthenticationProvider 和 LdapAuthenticationProvider 可疑对应任何一个提交简单 username/password 的认证请求，所以可以和基于表单登陆和 HTTP 基本认证一起工作。其他时候，一些认证机制创建了 一个认证请求对象，只可以被单个类型的 AuthenticationProvider 拦截。 一个例子就是 JA-SIG CAS，它使用一个提醒的服务票据，所以只可以被 CasAuthenticationProvider 认证。你不需要很了解这些， 因为如果你忘记了注册合适的供应器，你会得到一个 ProviderNotFoundException 当这个验证尝试起作用的时候。

6.1.1. DaoAuthenticationProvider

spring security 中最简单的 AuthenticationProvider 实现 是 DaoAuthenticationProvider，这也是框架中最早支持的功能之一。 它是 UserDetailsService 的杠杆（作为 DAO）， 为了获得 username, password 和 GrantedAuthority 。 它认证用户， 通过简单比较密码， 在 UsernamePasswordAuthenticationToken 中，和 UserDetailsService 中加载的信息。 配置供应器十分简单：

```
<bean id="daoAuthenticationProvider"



class="org.springframework.security.authentication.dao.DaoAuthe

nticationProvider">
```

```
  <property name="userDetailsService" ref="inMemoryDaoImpl"/>

  <property name="saltSource" ref bean="saltSource"/>

  <property name="passwordEncoder" ref="passwordEncoder"/>

</bean>
```

PasswordEncoder 和 SaltSource 都是可选的，一个 PasswordEncoder 提供了编码和解码密码， 在 UserDetails 对象中，被返回自配置好的 UserDetailsService。一个 SaltSource 可以让密码使用"盐值"生成，这可以提高授权仓库中密码的安全性。更多的细节会在 下面进行讨论。

6.2. UserDetailsService 实现

像在前面提及的一样，大多数认证供应器都是用了 UserDetails 和 UserDetailsService 接口。 调用 UserDetailsService 中的单独的方法：

```
  UserDetails loadUserByUsername(String username) throws

UsernameNotFoundException;
```

返回的 UserDetails 是一个接口，它提供了获得保证 非空的认证信息，比如用户名，密码，授予的权限和用户账号是可用还是禁用。 大多数认证供应器会使用 UserDetailsService，即使 username 和 password 没有实际用在这个认证决策中。它们可以使用返回的 UserDetails 对象，获得它的 GrantedAuthority 信息，因为一些其他系统（比如 LDAP 或者 X.509 或 CAS 等等） 了解真实验证证书的作用。

这里的 UserDetailsService 也很简单实现， 它应该为用户简单的获得认证信息，使用它们选择的持久化策略。 这样说，Spring Security 包含了很多有用的基本实现，下面我们会看到。

6.2.1. 内存认证

创建一个自定义的 UserDetailsService 的实现是很容易的， 可以从选择的持久化引擎中获得信息，但是许多应用没有那么复杂。尤其是如果你建立一个原型应用 或只是开始集成 Spring Security 的时候，当我们不是真的需要耗费时间配置数据库或者写 UserDetailsService 实现。为了这些情况， 一个简单的选择是使用安全命名空间中的 user-service 元素：

```
  <user-service id="userDetailsService">
```

```
    <user          name="jimi"          password="jimispassword"
authorities="ROLE_USER, ROLE_ADMIN" />
    <user           name="bob"           password="bobspassword"
authorities="ROLE_USER" />
  </user-service>
```

也支持使用外部的属性文件：

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

属性文件需要包含下面格式的内容

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

比如

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

6.2.2. JdbcDaoImpl

Spring Security 也包含了一个 UserDetailsService， 它包含从一个 JDBC 数据源中获得认证信息。内部使用了 Spring JDBC，所以它避免了负责的 功能完全的对象关系映射（ORM）只用来保存用户细节。如果你的应用使用了一个 ORM 工具， 你应该写一个自己的 UserDetailsService 重用你已经创建了的映射文件。返回到 JdbcDaoImpl， 一个配置的例子如下所示：

```
<bean                                                id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSou
rce">
  <property                                    name="driverClassName"
value="org.hsqldb.jdbcDriver"/>
  <property                                                name="url"
value="jdbc:hsqldb:hsql://localhost:9001"/>
  <property name="username" value="sa"/>
  <property name="password" value=""/>
</bean>


<bean                                        id="userDetailsService"
class="org.springframework.security.core.userdetails.jdbc.JdbcD
aoImpl">
  <property name="dataSource" ref="dataSource"/>
</bean>
```

你可以使用不同的关系数据库管理系统，通过修改上面的 DriverManagerDataSource。你也可以使用通过 JNDI 获得的全局数据源， 使用其他的 Spring 配置。

6.2.2.1. 权限分组

默认情况下，JdbcDaoImpl 会假设用户的权限都保存在 authorities 表中。 （参考数据库结构附录）. 还有一种选择是把权限分组，然后让用户加入这些用户组。一些人更喜欢使用这种方法来管理用户的权限。 参考 JdbcDaoImpl 的 Javadoc 以获得更多的信息，了 ijeruhe 启用权限分组。 用户组使用的数据库结构也包含在附录中。

6.3. 密码加密

Spring Security 的 PasswordEncoder 接口用来支持 对密码通过一些方式进行加密，并保存到媒介中。 这通常意味着密码被"散列加密"，使用一个加密算法，比如 MD5 或者 SHA。

6.3.1. 什么是散列加密？

密码加密不是 Spring Security 唯一的，但这对一个不了解这个概念的用户来说 还是一个很容易搞混的来源。一个散列（或摘要）算法是一个单向方法提供了一小段固定长度 的输出数据（散列）从一些输入数据中，比如一个密码。作为一个例子， 字符串"password"的 MD5 散列（16 进制）是

```
5f4dcc3b5aa765d61d8327deb882cf99
```

散列是"单向的"在这种情况下，很难（基本上不可能）根据给出的散列值获得原始输入，或是找出任何可能的输入将生成散列值。这个特点让散列值对权限方面很有用。它们可以保存在你的用户数据库中作为原始明文密码的替换，假设这些值被泄露了 也无法立即盗取登录的密码。注意这也意味着你没有办法把编码后的密码还原。

6.3.2. 为散列加点儿盐

使用密码加密的一个潜在的问题是，因为散列是单向的，如果输入是一个常用的单词的话 找到输入值就相对容易很多了。比如，如果我们查找散列值 5f4dcc3b5aa765d61d8327deb882cf99 通过 google。我们会很快找到 原始词是"password"。简单的方法，一个攻击者可以建立一个散列值的字典 把标准单词排列，使用它来查找原始密码。一个方法来帮助防止这种问题是使用高强度的密码 策略来防止使用常用单词。另一个是在计算散列时使用"盐值"。 这是一个对每个用户都知道的附加字符串，它会结合到密码中，在计算散列之前。 注意这个数值应该是尽可能的随机数，但是实际中任何盐值通常都是不可取的。 Spring Security 有一个 SaltSource 接口， 可以被验证供应器用来为特定的用户生成一个盐值。 使用盐值，意味着攻击者必须创建单独的散列字典，为不同的盐值， 这让攻击更难了（但不是不可能）。

6.3.3. 散列和认证

当一个认证供应器（比如 Spring Security 的 DaoAuthenticationProvider） 需要检验密码，在提交认证请求中，与用户知道的数据进行比较，保存的密码通过一些方式进行了加密， 然后提交的数据必须也使用相同的算法进行加密。这要求你去检查兼容性，因为 Spring Security 对持久化的没有任何控制。如果你在 Spring Security 的认证配置中添加了密码散列功能， 你的数据库包含原始明文密码，那么认证就绝对不可能成功。 如果你在数据库中使用 MD5 对密码加密，比如，你的应用配置为使用 Spring Security 的 Md5PasswordEncoder，这也有其他可能的问题。 数据库可能用 Base 64 进行了加密，比如当加密器使用 16 进制的字符串（默认） [5]。 可以选择，你的数据库可能使用了大写，当编码器输出的是小写。 确定你编写了一个测试来检测从你的密码编码器的输出，使用一个知道的密码和盐值结合 检测它是否与数据库值匹配，在更深入之前，尝试通过你的系统认证。 要想获得更多信息，在默认的方法从结合盐值和密码，查看 BasePasswordEncoder 的 Javadoc。如果你希望直接通过 java 生成密码， 为你的用户数据库保存，然后你可以使用 PasswordEncoder 的 encodePassword 方法。

Part III. web 应用安全

大多数 Spring Security 的用户使用框架是基于 HTTP 和 Servlet API 的。 在这一章和以后的章节中，我们会看一下如何使用 Spring Security 提供验证和权限控制特性，用于应用的 web 层。我们会介绍命名空间外观的背后，看看哪些类和接口实际上用于提供 web 层的安全。 在一些情况下，必须使用以前的 bean 配置来提供对配置的完全孔子和，所以我们 也会看到如何直接配置这些类，不使用命名空间。

安全过滤器链

Spring Security 的 web 架构是完全基于标准的 servlet 过滤器的。 它没有在内部使用 servlet 或任何其他基于 servlet 的框架（比如 spring mvc）， 所以它没有与任何特定的 web 技术强行关联。 它只管处理 HttpServletRequest 和 HttpServletResponse，不关心请求时来自浏览器，web 服务客户端，HttpInvoker 还是一个 AJAX 应用。

Spring Security 维护了一个过滤器链，每个过滤器拥有特定的功能，过滤器需要服务也会对应添加和删除。 过滤器的次序是非常重要的，它们之间都有依赖关系。 如果你已经使用了命名空间配置，过滤器会自动帮你配置， 你不需要定义任何 Spring Bean，但是有时候你需要完全控制 Spring 过滤器链， 因为你使用了命名空间没有提供的特性，或者你需要使用你自己自定义的类。

7.1. DelegatingFilterProxy

当使用 servlet 过滤器时，你很需要在你的 web.xml 中声明它们， 它们可能被 servlet 容器忽略。在 Spring Security，过滤器类也是定义在 xml 中的 spring bean， 因此可以获得 Spring 的依赖注入机制和生命周期接口。 spring 的 DelegatingFilterProxy 提供了在 web.xml 和 application context 之间的联系。

当使用 DelegatingFilterProxy，你会看到像 web.xml 文件中的这样内容：

```
  <filter>

    <filter-name>myFilter</filter-name>



<filter-class>org.springframework.web.filter.DelegatingFilterPr

oxy</filter-class>

  </filter>



  <filter-mapping>
```

```
    <filter-name>myFilter</filter-name>

    <url-pattern>/*</url-pattern>

  </filter-mapping>
```

注意这个过滤器其实是一个 DelegatingFilterProxy，这个过滤器里没有实现过滤器的任何逻辑。 DelegatingFilterProxy 做的事情是代理 Filter 的方法，从 application context 里获得 bean。 这让 bean 可以获得 spring web application context 的生命周期支持，使配置较为轻便。 bean 必须实现 javax.servlet.Filter 接口，它必须和 filter-name 里定义的名称是一样的。查看 DelegatingFilterProxy 的 javadoc 获得更多信息。

7.2. FilterChainProxy

现在应该清楚了，你可以声明每个 Spring Security 过滤器 bean，你在 application context 中需要的。 把一个 DelegatingFilterProxy 入口添加到 web.xml， 确认它们的次序是正确的。 这是一种繁琐的方式，会让 web.xml 显得十分杂乱，如果我们配置了太多过滤器的话。 我们最好添加一个单独的入口，在 web.xml 中，然后在 application context 中处理实体， 管理我们的 web 安全 bean。 这就是 FilterChainProxy 所做的事情。它使用 DelegatingFilterProxy（就像上面例子中那样），但是对应的 class 是 org.springframework.security.web.FilterChainProxy 。 过滤器链是在 application context 中声明的。这里有一个例子：

```
<bean                                          id="filterChainProxy"

class="org.springframework.security.web.FilterChainProxy">

  <sec:filter-chain-map path-type="ant">

    <sec:filter-chain pattern="/webServices/**" filters="

          securityContextPersistenceFilterWithASCFalse,

          basicAuthenticationFilter,

          exceptionTranslationFilter,

          filterSecurityInterceptor" />

    <sec:filter-chain pattern="/**" filters="
```

```
            securityContextPersistenceFilterWithASCFalse,
            formLoginFilter,
            exceptionTranslationFilter,
            filterSecurityInterceptor" />
  </sec:filter-chain-map>
</bean>
```

你可能注意到 FilterSecurityInterceptor 声明的不同方式。 命名空间元素 filter-chain-map 被用来设置安全过滤器链。 它映射一个特定的 URL 模式，到过滤器链中，从 bean 名称来定义的 filters 元素。 它同时支持正则表达式和 ant 路径，并且只使用第一个出现的匹配 URI。在运行阶段 FilterChainProxy 会定位当前 web 请求匹配的第一个 URI 模式，由 filters 属性指定的过滤器 bean 列表将开始处理请求。 过滤器会按照定义的顺序依次执行，所以你可以对处理特定 URL 的过滤器链进行完全的控制。

你可能注意到了，我们在过滤器链里声明了两个 SecurityContextPersistenceFilter（ASC 是 allowSessionCreation 的简写，是 SecurityContextPersistenceFilter 的一个属性）。 因为 web 服务从来不会在请求里带上 jsessionid,为每个用户代理都创建一个 HttpSession 完全是一种浪费。如果你需要构建一个高等级最高可扩展性的系统，我们推荐你使用上面的配置方法。 对于小一点儿的项目，使用一个 HttpSessionContextIntegrationFilter（让它的 allowSessionCreation 默认为 true）就足够了。

在有关声明周期的问题上，如果这些方法被 FilterChainProxy 自己调用，FilterChainProxy 会始终根据下一层的 Filter 代理 init(FilterConfig)和 destroy()方法。 这时，FilterChainProxy 会保证初始化和销毁操作只会在 Filter 上调用一次， 而不管它在过滤器链中被声明了多少次）。你控制着所有的抉择，比如这些方法是否被调用 或 targetFilterLifecycle 初始化参数 DelegatingFilterProxy。 默认情况下，这个参数是 false，servlet 容器生命周期调用不会传播到 DelegatingFilterProxy。

当我们了解如何使用命名控制配置构建 web 安全。 我们使用一个 DelegatingFilterProxy，它的名字是“springSecurityFilterChain”。 你应该现在可以看到 FilterChainProxy 的名字，它是由命名空间创建的。

7.2.1. 绕过过滤器链

通过命名空间，你可以使用 filters = "none"，来提供一个过滤器 bean 列表。 这会朝向请求模式，使用安全过滤器链整体。注意任何匹配这个模式的路径不会有任何授权或校验的服务 起作用，它们是可以自由访问的。

7.3. 过滤器顺序

定义在 web.xml 里的过滤器的顺序是非常重要的。 不论你实际使用的是哪个过滤器，<filter-mapping>的顺序应该像下面这样：

- ChannelProcessingFilter，因为它可能需要重定向到其他协议。
- ConcurrentSessionFilter，因为它不使用 SecurityContextHolder 功能，但是需要更新 SessionRegistry 来从主体中放映正在进行的请求。
- SecurityContextPersistenceFilter，这样 SecurityContext 可以在 web 请求的开始阶段通过 SecurityContextHolder 建立，然后 SecurityContext 的任何修改都会在 web 请求结束的时候（为下一个 web 请求做准备）复制到 HttpSession 中。
- 验证执行机制 - UsernamePasswordAuthenticationFilter, CasAuthenticationFilter, BasicAuthenticationFilter 等等 - 这样 SecurityContextHolder 可以被修改，并包含一个合法的 Authentication 请求标志。
- SecurityContextHolderAwareRequestFilter，如果，你使用它，把一个 Spring Security 提醒 HttpServletRequestWrapper 安装到你的 servlet 容器里。
- RememberMeAuthenticationFilter，这样如果之前的验证执行机制没有更新 SecurityContextHolder，这个请求提供了一个可以使用的 remember-me 服务的 cookie，一个对应的已保存的 Authentication 对象会被创建出来。
- AnonymousAuthenticationFilter，这样如果之前的验证执行机制没有更新 SecurityContextHolder，会创建一个匿名 Authentication 对象。
- ExceptionTranslationFilter，用来捕捉 Spring Security 异常，这样，可能返回一个 HTTP 错误响应，或者执行一个对应的 AuthenticationEntryPoint。
- FilterSecurityInterceptor，保护 web URI。

7.4. 使用其他过滤器 —— 基于框架

如果你在使用 SiteMesh，确认 Spring Security 过滤器在 SiteMesh 过滤器之前调用。这可以保证 SecurityContextHolder 为每个 SiteMesh 渲染器及时创建。

Chapter 8. 核心安全过滤器

这儿有几个在 web 应用中一直会用到的 Spring Security 关键过滤器， 所以我们回来看看它们，和支持的类和接口。 我们不会覆盖所有功能，所以确认参考它们的 javadoc，如果你想要获得完全的信息。

8.1. FilterSecurityInterceptor

我们已经简要了解了 FilterSecurityInterceptor，在简要讨论访问控制的时候（见 #tech-intro-access-control ），我们也已经在命名空间中使用过它，<intercept-url>元素的结合在内部对它进行了配置。 现在我们会看如何精确的配置它，使用 FilterChainProxy， 通过结合 ExceptionTranslationFilter 过滤器。一个典型的配置例子如下：

```
<bean id="filterSecurityInterceptor"

class="org.springframework.security.intercept.web.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 负责处理 HTTP 资源的安全。 它需要一个 AuthenticationManager 和 AccessDecisionManager 的引用。它也需要不同 HTTP

URL 请求的配置属性。 引用回#tech-intro-config-attributes 这里可以看到原始信息。

FilterSecurityInterceptor 可是通过两种方式定义配置属性。 第一种，向上面演示的，使用<filter-security-metadata-source> 命名空间元素。这和用来配置 FilterChainProxy 的<filter-chain-map>一样，但是使用的是 <intercept-url> 子元素，只使用 pattern 和 access 属性。逗号用来分隔不同的配置属性，对于每个 HTTP URL。 第二个选择是编写你自己的 SecurityMetadataSource， 但是这超越了我们的文档的范围。根据使用的方式， SecurityMetadataSource 负责返回一个包含了所有配置属性 的 List<ConfigAttribute>，它分配给一个单独的安全 HTTP URL。

应该注意的是 FilterSecurityInterceptor.setSecurityMetadataSource()方法其实需要一个 FilterSecurityMetadataSource 实例。 这是一个标记接口，它是 SecurityMetadataSource 的子类。 它只标记了 SecurityMetadataSource 需要 FilterInvocation 。 对 于 相 似 感 兴 趣 ， 我 们 会 继 续 引 用 FilterInvocationDefinitionSource 作为一个 SecurityMetadataSource,作为区别大多数用户的微笑相关不同。

SecurityMetadataSource 通过命名空间获得配置属性， 为一个特定的 FilterInvocation，通过匹配请求 URL，对于 配置好的 pattern 属性。这个行为和它在命名空间中配置的一样。 默认使用 apache ant path 的方式来处理所有表达式，也支持正则表达式来支持更复杂的请求。 这个 path-type 属性用来指定模式使用的类型。 它不可能在同一个定义中使用多个表达式语法的复合结构。作为一个例子，上面的配置使用正则表达式 替换了 ant path，可以写成下面这样：

```
<bean id="filterInvocationInterceptor"



class="org.springframework.security.intercept.web.FilterSecurit

yInterceptor">

  <property                              name="authenticationManager"

ref="authenticationManager"/>

  <property                              name="accessDecisionManager"

ref="accessDecisionManager"/>

  <property name="runAsManager" ref="runAsManager"/>

  <property name="securityMetadataSource">

    <security:filter-security-metadata-source path-type="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>
```

模式总是根据他们定义的顺序进行执行。因此很重要的是，把更确定的模式定义到列表的上面。 这会反映在你上面的例子中，更确定的/secure/super/模式放在，没那么确定的 /secure/模式的上面。如果它们被反转了。/secure/会一直 被匹配，/secure/super/就永远也不会执行。

8.2. ExceptionTranslationFilter

ExceptionTranslationFilter 处在 FilterSecurityInterceptor 的上面。 它不执行任何真正的安全控制，但是处理安全监听器抛出的一场， 提供对应的 HTTP 响应。

```
<bean id="exceptionTranslationFilter"

class="org.springframework.security.web.access.ExceptionTransla
tionFilter">
  <property                    name="authenticationEntryPoint"
ref="authenticationEntryPoint"/>
  <property                          name="accessDeniedHandler"
ref="accessDeniedHandler"/>
</bean>
```

```
<bean id="authenticationEntryPoint"

class="org.springframework.security.web.authentication.LoginUrl
AuthenticationEntryPoint">
  <property name="loginFormUrl" value="/login.jsp"/>
</bean>

<bean id="accessDeniedHandler"

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

8.2.1. AuthenticationEntryPoint

AuthenticationEntryPoint 会被调用， 在用户请求一个安全 HTTP 资源，但是他们还没有被认证。一个对应的 AuthenticationException 或 AccessDeniedException 会被抛出， 由一个安全监听器在下面的调用栈中，触发入口点的 commence 方法。这执行展示对应的响应给用户， 这样认证可以开始。我们这里使用的是 LoginUrlAuthenticationEntryPoint，它会把请求 重定向到另一个不同的 URL（一般是一个登陆页面）。实际的使用使用会依赖 你希望使用到的认证机制。

8.2.2. AccessDeniedHandler

如果一个用户已经认证了，他再去访问一个被保护的资源时会怎么样呢？ 正常情况下，这不会发生，因为应用工作流应该限制哪些资源用户可以访问。 比如一个 HTML 连接到管理员页面，可能对没有管理权限的用户隐藏。 你不能对一个隐藏的链接点击，因为安全的原因，这总有可能用户直接输入了 URL 尝试越过限制。 或者他们可能修改了 RESTful URL 来改变一些参数的值。你的应用必须保护这些场景， 或者它们会变的不安全。你将使用简单的 web 层安全在你的服务层接口上， 来确认哪些是被允许的。

如果一个 AccessDeniedException 被抛出了， 一个用户已经被认证，然后这意味着操作已经被尝试了，而他们没有足够的权限， 这种情况下，ExceptionTranslationFilter 会调用第二个策略， AccessDeniedHandler。默认下，一个 AccessDeniedHandlerImpl 会被使用，它只发送一个 403（拒绝访问） 响应到客户端。可选的，你可以确切配置一个实例（像上面的例子）然后设置一个错误页 URL 它会把请求跳转到错误页。 [6]。 这可能是一个简单的"access denied"页面，比如一个 JSP，或者它可能是一个 更加复杂的处理器，比如一个 MVC 控制器。当然，我们可以实现自己的接口， 使用你自己的实现。

也可能提供一个自定义的 AccessDeniedHandler 然后使用命名空间配置你的应用，参考#nsa-access-denied-handler.

8.3. SecurityContextPersistenceFilter

我们介绍了所有重要的过滤器，在技术概述一章， 所以你可能希望重新阅读以下这些章节。让我们首先看一下如何使用 FilterChainProxy 配置它们。 一个基本的配置需要 bean 自己

```
<bean id="securityContextPersistenceFilter"



class="org.springframework.security.web.context.SecurityContext

PersistenceFilter"/>
```

像我们之前看到的，这个过滤器有两个主要的任务，它负责保存在不同的 HTTP 请求之间 SecurityContext，负责清理在请求完成时 SecurityContextHolder。 清理上下文中保存的 ThreadLocal 是很基本的，因为它可能 是 servlet 容器线程池中的一个替换的 thread，这样 spring security 对应一个特定用户 可能出现冲突。这个线程可能被下一个阶段使用，执行操作的时候就会使用错误的证书。

8.3.1. SecurityContextRepository

对于 Spring Security 3.0，读取和保存安全上下文的任务被委托给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 是一个简单的容器，为了进入的请求和相应对象， 允许使用封装类替换它们。返回的内容会被发送给过滤器链。

默认的实现是 HttpSessionSecurityContextRepository， 它会把安全上下文保存为一个 HttpSession 的属性。 [7]. 这个实现中最重要的配置参数是 allowSessionCreation 属性， 默认是 true，因此允许类创建会话，如果它需要保存 spring context 为一个认证用户（它不会创建一个除非认证已经执行，security context 的内容发生改变）。如果你不想要创建会话，你可以把这个属性设置为 false：

```
<bean id="securityContextPersistenceFilter"

class="org.springframework.security.web.context.SecurityContext
PersistenceFilter">
  <property name='securityContextRepository'>
    <bean
class='org.springframework.security.web.context.HttpSessionSecu
rityContextRepository'>
      <property name='allowSessionCreation' value='false' />
    </bean>
  </property>
</bean>
```

可选的是，你可以提供一个 SecurityContextRepository 接口的 null 实现，这就可以防止安全上下文被保存，即使一个 session 已经在请求期间被创建了。

8.4. UsernamePasswordAuthenticationFilter

我现在看到了三个主要的过滤器，它们会一直在 spring security 的 web 配置中起作用。 它们也是由命名空间<http>元素自动创建的三个，它们是无法修改的。 唯一丢失的是一个真正的认证机制，它们会执行一个用户的认证。这个过滤器是最常用的认证过滤器， 这个过滤器也通常需要自定义。 [8]. 它也提供了使用<form-login>元素的实现，来自命名空间。 有三个阶段需要配置它。

- 配置一个 LoginUrlAuthenticationEntryPoint 使用一个登陆页 URL， 就像我们上面那样，把它配置到 ExceptionTranslationFilter 中。
- 实现一个登陆页面（使用 JSP 或 MVC 控制器）。
- 配置一个 UsernamePasswordAuthenticationFilter 的实例，放在 application context 中。
- 把过滤器 bean 添加到你的过滤器链代理中 （确认你注意了顺序）。

登陆表单简单包含了 j_username 和 j_password 输入框，提交由过滤器管理的 URL，（默认为：/j_spring_security_check）. 基本的过滤器配置看起来像这样：

```
<bean id="authenticationFilter" class=

"org.springframework.security.web.authentication.UsernamePasswo

rdAuthenticationFilter">

  <property                          name="authenticationManager"

ref="authenticationManager"/>

  <property                             name="filterProcessesUrl"

value="/j_spring_security_check"/>

</bean>
```

8.4.1. 认证成功和失败的应用流程

过滤器调用了配置的 AuthenticationManager 处理每个认证请求。 认证成功或失败的目的地是由 AuthenticationSuccessHandler 和 AuthenticationFailureHandler 策略接口各自控制的。 过滤器的属性允许我们设置这些，这样你可以随心所欲的自定义他们的行为。 [9]. 我们提供了一些标准实现，比如 SimpleUrlAuthenticationSuccessHandler ，

SavedRequestAwareAuthenticationSuccessHandler，SimpleUrlAuthenticationFailureHandler 和 ExceptionMappingAuthenticationFailureHandler。 参考这些类的 javadoc，了解他们是如何工作的。

如果认证成功，结果 Authentication 对象会被放到 SecurityContextHolder 中。配置的 AuthenticationSuccessHandler 会被调用， 进行重定向会把用户跳转到合适的目的地。默认情况，会使用 SavedRequestAwareAuthenticationSuccessHandler。这意味着，用户会被重定向到原始的目标，他们在登录之前请求的页面。

Note

ExceptionTranslationFilter 缓存了一个用户的原始请求。 当用户认证时，请求处理器从这个缓存的请求中获得原始的 URL，并重定向到它。 原始请求然后重新构造，作为一个可选项使用。

如果认证失败，配置好的 AuthenticationFailureHandler 会被调用。

---

[6] 我们使用 forward，这样 SecurityContextHolder 会包含主体的信息， 这可能对现实用户信息很有帮助。在老版本的 Spring Security 中，我们让 servlet 容器处理一个 403 错误信息，这可能丢失了有用的上下文信息。

[7] 在 spring security 2.0 以及以前，这个过滤器叫做 HttpSessionContextIntegrationFilter，它会执行保存上下文 的所有工作。如果你对这个类很熟悉，然后大多数的配置属性现在都可以在 HttpSessionSecurityContextRepository 中找到。

[8] 因为一些历史原因，在 Spring Security 3.0 之前，这个过滤器被称为 AuthenticationProcessingFilter，入口点被称为 AuthenticationProcessingFilterEntryPoint。 因为这个框架现在支持了很多不同的认证表单，它们都需要在 3.0 中给与更确切的名字。

[9] 在版本 3.0 之前，这一点的应用流程被当做一个状态，通过这个类的一系列属性 和策略插件进行处理。这个决定让 3.0 重构了代码，让两个策略完全负责。

Basic（基本）和 Digest（摘要）验证

Basic（基本）和 Digest（摘要）验证都是 web 应用中很受欢迎的可选机制。 Basic 验证一般用来处理无状态的客户端，它们在每次请求都附带它们的证书。 很常见的用法是把它和基于表单的验证一起使用，这里的应用会同时使用基于浏览器的用户接口和 web 服务。 然而，basic 验证使用原文传送密码，所以应该只通过加密的传输途径发送，比如 HTTPS。

9.1. BasicAuthenticationFilter

BasicAuthenticationFilter 负责处理通过 HTTP 头部发送来的 basic 验证证书。它可以用来像对待普通用户代理一样（比如 IE 和 Navigator）认证由 Spring 远程协议

的调用（比如 Hessian 和 Burlap）。 HTTP 基本认证的执行标准定义在 RFC 1945，11 章，BasicAuthenticationFilter 符合这个 RFC。 基本认证是一个极具吸引力的认证方法，因为它在用户代理发布很广泛， 实现也特别简单（只需要对 username:password 进行 Base64 编码，再放到 HTTP 头部里）。

9.1.1. 配置

要实现 HTTP 基本认证，要先在过滤器链里定义 BasicAuthenticationFilter。 还要在 application context 里定义 BasicAuthenticationFilter 和协作的类：

```
<bean id="basicAuthenticationFilter"

class="org.springframework.security.web.authentication.www.BasicAuthenticationFilter">
  <property name="authenticationManager" ref="authenticationManager"/>
  <property name="authenticationEntryPoint" ref="authenticationEntryPoint"/>
</bean>

<bean id="authenticationEntryPoint"

class="org.springframework.security.web.authentication.www.BasicAuthenticationEntryPoint">
  <property name="realmName" value="Name Of Your Realm"/>
</bean>
```

配置好的 AuthenticationManager 会处理每个认证请求。 如果认证失败，配置好的 AuthenticationEntryPoint 会用来重试认证过程。 通常你会使用 BasicAuthenticationEntryPoint，它会返回一个 401 响应，使用对应的头部重试 HTTP 基本验证。 如果验证成功，就把得到的 Authentication 对象放到 SecurityContextHolder 里。

如果认证事件成功，或者因为 HTTP 头部没有包含支持的认证请求所以没有进行认证，过滤器链会像通常一样继续下去。 唯一打断过滤器的情况是在认证失败并调用 AuthenticationEntryPoint 的时候，向上面段落里讨论的那样。

9.2. DigestAuthenticationFilter

Spring Security 提供了一个 DigestAuthenticationFilter，它可以处理 HTTP 头部中的摘要认证证书。 摘要认证在尝试着解决许多基本认证的缺陷，特别是保证不会通过纯文本发送证书。 许多用户支持摘要式认证，包括 FireFox 和 IE。 HTTP 摘要式认证的执行标准定义在 RFC 2617，它是对 RFC 2069 这个早期摘要式认证标准的更新。 Spring Security DigestAuthenticationFilter 会保证"auth"的安全质量（qop），它订明在 RFC 2617 中，并与 RFC 2069 提供了兼容。 如果你需要使用没有加密的 HTTP（比如没有 TLS/HTTP），还希望认证达到最大的安全性的时候，摘要式认证便具有很高吸引力。 事实上，摘要式认证是 WebDAV 协议的强制性要求，写在 RFC 2518 的 17.1 章，所以我们应该期望看到更多的代替基本认证。

摘要式认证，是表单认证，基本认证和摘要式认证中最安全的选择，不过更安全也意味着更复杂的用户代理实现。 摘要式认证的中心是一个“nonce”。 这是由服务器生成的一个值。 Spring Security 的 nonce 采用下面的格式：

```
base64(expirationTime + ":" + md5Hex(expirationTime + ":" + key))

expirationTime: The date and time when the nonce expires, expressed in milliseconds
key: A private key to prevent modification of the nonce token
```

这个 DigestAuthenticationEntryPoint 有一个属性，通过指定一个 key 来生成 nonce 标志，通过 nonceValiditySeconds 属性来决定过期时间（默认 300，等于 5 分钟）。 只要 nonce 是有效的，摘要就会通过串联字符串计算出来，包括用户名，密码，nonce，请求的 URI，一个客户端生成 nonce（仅仅是一个随机值，用户代理每

个请求生成一个），realm 名称等等，然后执行一次 MD5 散列。 服务器和用户代理都要执行这个摘要计算，如果他们包含的值不同（比如密码），就会生成不同的散列码。在 Spring Security 的实现中，如果服务器生成的 nonce 已经过期（但是摘要还是有效），DigestAuthenticationEntryPoint 会发送一个"stale=true"头信息。 这告诉用户代理，这里不再需要打扰用户（像是密码和用户其他都是正确的），只是简单尝试使用一个新 nonce。

DigestAuthenticationEntryPoint 的 nonceValiditySeconds 参数，会作为一个适当的值依附在你的程序上。 对安全要求很高的用户应该注意，一个被拦截的认证头部可以用来假冒主体，直到 nonce 达到 expirationTime。 在选择合适的配置的时候，这是一个必须考虑到的关键性条件，但是在对安全性要求很高的程序里，第一次请求都会首先运行在 TLS/HTTPS 之上。

因为摘要式认证需要更复杂的实现，这里常常有用户代理的问题。 比如，IE 不能在同一个会话的请求进程里阻止"透明"标志。 因此 Spring Security 把所有状态信息都概括到"nonce"标记里。 在我们的测试中，Spring Security 在 FireFox 和 IE 里都可以工作，正确的处理 nonce 超时等等。

9.2.1. Configuration

现在我们重新看一下理论，让我们看看如何使用它。 为了实现 HTTP 摘要认证，必须在过滤器链里定义 DigestAuthenticationFilter。 application context 还需要定义 DigestAuthenticationFilter 和它需要的合作伙伴：

```
<bean id="digestnFilter" class=

"org.springframework.security.web.authentication.www.DigestAuth

enticationFilter">

  <property name="userDetailsService" ref="jdbcDaoImpl"/>

  <property                        name="authenticationEntryPoint"

ref="digestEntryPoint"/>

  <property name="userCache" ref="userCache"/>

</bean>

<bean id="digestEntryPoint" class=
```

```
"org.springframework.security.web.authentication.www.DigestAuth
enticationEntryPoint">
  <property name="realmName" value="Contacts Realm via Digest
Authentication"/>
  <property name="key" value="acegi"/>
  <property name="nonceValiditySeconds" value="10"/>
</bean>
```

需要配置一个 UserDetailsService，因为 DigestAuthenticationFilter 必须直接访问用户的纯文本密码。如果你在 DAO 中使用编码过的密码，摘要式认证就没法工作。DAO 合作者，与 UserCache 一起，通常使用 DaoAuthenticationProvider 直接共享。这个 AuthenticationEntryPoint 属性必须是 DigestAuthenticationEntryPoint，这样 DigestAuthenticationFilter 可以在进行摘要计算时获得正确的 realmName 和 key。

像 BasicAuthenticationFilter 一样，如果认证成功，会把 Authentication 请求标记放到 SecurityContextHolder 中。 如果认证事件成功，或者认证不需要执行，因为 HTTP 头部没有包含摘要认证请求，过滤器链会正常继续。 过滤器链中断的唯一情况是，如果认证失败，就会像上面讨论的那样调用 AuthenticationEntryPoint。

摘要式认证的 RFC 要求附加功能范围，来更好的提升安全性。 比如，nonce 可以在每次请求的时候变换。 但是，Spring Security 的设计思路是最小复杂性的实现（毫无疑问，用户代理会出现不兼容），也避免保存服务器端的状态。 如果你想研究这些功能的更多细节，我们推荐你看一下 RFC 2617。 像我们知道的那样，Spring Security 实现类遵守了 RFC 的最低标准。

Remember-Me 认证

10.1. 概述

记住我（remember-me）或持久登录（persistent-login）认证，指的是网站可以在不同会话之间记忆验证的身份。 通常情况是发送一个 cookie 给浏览器，在以后的 session 里检测 cookie，进行自动登录。 Spring Security 为 remember-me 实现提供了必要的调用钩子，并提供了两个 remember-me 的具体实现。 其中一个使用散

列来保护基于 cookie 标记的安全性，另一个使用了数据库或其他持久化存储机制来保存生成的标记。

注意，两个实现方式，都需要 UserDetailsService。 如果你使用了认证提供器，没有使用 UserDetailsService（比如 LDAP 供应器），那它就没法工作，除非你在 application context 里设置了一个 UserDetailsService。

10.2. 简单基于散列标记的方法

这种方法使用散列来完成 remember-me 策略。 本质上，在成功进行认证的之后，把一个 cookie 发送给浏览器，使用的 cookie 组成结构如下：

```
    base64(username + ":" + expirationTime + ":" +

        md5Hex(username + ":" + expirationTime + ":" password + ":"

+ key))



    username:         As identifiable to the UserDetailsService

    password:             That matches the one in the retrieved

UserDetails

    expirationTime:  The date and time when the remember-me token

expires,

                      expressed in milliseconds

    key:              A private key to prevent modification of the

remember-me token
```

这个 remember-me 标记只适用于指定范围，提供用户名，密码和关键字都不会改变。值得注意，这里有一个潜在的安全问题，来自任何一个用户代理的 remember-me 标记，直到标记过期都是可用的。 这个问题和摘要式认证相同。 如果一个用户发现标记已经设置了，他们可以轻易修改他们的密码，并且立即注销所有的 remember-me 标记。 如果需要更好的安全性，你应该使用下一章描述的方法。 或者不应该使用 remember-me 服务。

如果你还记得在命名空间配置中讨论的主题，你只要添加<remember-me>元素就可以使用 remember-me 认证：

```
<http>

  ...

  <remember-me key="myAppKey"/>

</http>
```

这个 UserDetailsService 会自动选上。 如果你在 application context 中配置了多个，你需要使用 user-service-ref 属性指定应该使用哪一个，这里的值要放上你的 UserDetailsService bean 的名字。

10.3. 持久化标记方法

这个方法是基于这篇文章 http://jaspan.com/improved_persistent_login_cookie_best_practice 进行了一些小修改 [10]。 要用在命名空间配置里使用这个方法，你应该提供一个 datasource 引用：

```
<http>

  ...

  <remember-me data-source-ref="someDataSource"/>

</http>
```

数据应该包含一个 persistent_logins 表，可以使用下面的 SQl 创建（或等价物）：

```
    create table persistent_logins (username varchar(64) not null,
series varchar(64) primary key, token varchar(64) not null,
last_used timestamp not null)
```

10.4. Remember-Me 接口和实现

Remember-me 认证不能和基本认证一起使用，因为基本认证往往不使用 HttpSession。Remember-me 使用在 UsernamePasswordAuthenticationFilter 中,通过在它的超类 AbstractAuthenticationProcessingFilter 里实现的一个调用钩子。 这个钩子会在合适的时候调用一个具体的 RememberMeServices。 这个接口看起来像这样：

```
  Authentication      autoLogin(HttpServletRequest      request,
HttpServletResponse response);
  void loginFail(HttpServletRequest request, HttpServletResponse
response);
  void          loginSuccess(HttpServletRequest          request,
HttpServletResponse response,
  Authentication successfulAuthentication);
```

请参考 JavaDocs 获得有关这些方法的完整讨论，不过注意在这里，AbstractAuthenticationProcessingFilter 只调用 loginFail() 和 loginSuccess()方法。 当 SecurityContextHolder 没有包含 Authentication 的时候，RememberMeProcessingFilter 才去调用 autoLogin()。 因此，这个接口通过使用完整的认证相关事件的提醒提供了下面 remember-me 实现，然后在可能包含一个 cookie 希望被记得的申请 web 请求中调用这个实现。 这个设计允许任何数目的 remember-me 实现策略。我们在下面看看上面介绍过的两个 Spring Security 提供的实现。

10.4.1. TokenBasedRememberMeServices

这个实现支持在 Section 10.2, "简单基于散列标记的方法"里描述的简单方法。TokenBasedRememberMeServices 被 RememberMeAuthenticationProvider 执行的时候生成一个 RememberMeAuthenticationToken 。 认证提供器和

TokenBasedRememberMeServices 之间共享一个 key。另外 TokenBasedRememberMeServices 需要一个 UserDetailsService，用它来获得用户名和密码，进行比较，然后生成 RememberMeAuthenticationToken 来包含正确的 GrantedAuthority[]。如果用户请求注销，让 cookie 失效，就应该使用系统提供的一系列注销命令。TokenBasedRememberMeServices 也实现 Spring Security 的 LogoutHandler 接口，这样可以使用 LogoutFilter 自动清除 cookie。

这些 bean 要求在 application context 里启用 remember-me 服务，像下面一样：

```
<bean id="rememberMeFilter"

class="org.springframework.security.web.authentication.remember
me.RememberMeAuthenticationFilter">
  <property name="rememberMeServices" ref="rememberMeServices"/>
  <property name="authenticationManager"
ref="theAuthenticationManager" />
</bean>

<bean id="rememberMeServices" class=

"org.springframework.security.web.authentication.rememberme.Tok
enBasedRememberMeServices">
  <property name="userDetailsService"
ref="myUserDetailsService"/>
  <property name="key" value="springRocks"/>
</bean>
```

```
<bean id="rememberMeAuthenticationProvider"

class="org.springframework.security.web.authentication.remember
me.RememberMeAuthenticationProvider">
  <property name="key" value="springRocks"/>
</bean>
```

不要忘记把你的 RememberMeServices 实现添加到 UsernamePasswordAuthenticationFilter.setRememberMeServices()属性中，包括把 RememberMeAuthenticationProvider 添加到你的 UsernamePasswordAuthenticationFilter.setProviders() 队列中，把 RememberMeProcessingFilter 添加到你的 FilterChainProxy 中（要放到 AuthenticationProcessingFilter 后面）。

10.4.2. PersistentTokenBasedRememberMeServices

这个类可以像 TokenBasedRememberMeServices 一样使用，但是它还需要配置一个 PersistentTokenRepository 来保存标记。 这里有两个标准实现。

- InMemoryTokenRepositoryImpl 最好是只用来测试。
- JdbcTokenRepositoryImpl 把标记保存到数据库里。

数据库表结构在 Section 10.3, “持久化标记方法”.

---

[10] 基本上，为了防止暴露有效登录名，用户名没有包含在 cookie 里。在这个文章的评论里有一个相关的讨论

会话管理

HTTP 会话相关的功能是由 SessionManagementFilter 和 SessionAuthenticationStrategy 接口联合处理的， 过滤器会代理它们。 典型的应用包括会话伪造攻击预防，检测会话超时， 限制已登录用户可以同时打开多少会话。

11.1. SessionManagementFilter

SessionManagementFilter 会检测 SecurityContextRepository 的内容，比较当前 SecurityContextHolder，决定用户在当前请求中是否已经登录， 通常被一个非内部认证机制，比如预验证或 remember-me[11] 如果资源库中包含一个安全上下文，过滤器什么也不会做。如果没有包含，thread-local 中 SecurityContext 包含一个（非匿名） Authentication 对象，过滤器就会假设他们已经在 过滤器栈中的前一个过滤器中被认证过了。它会调用配置好的 SessionAuthenticationStrategy。

如果用户当前没有认证，过滤器会检测是否有无效的 session ID 被请求了（比如，因为超时） 并会重定向到配置好的 invalidSessionUrl，如果设置了。 最简单的配置方法是通过命名空间，如前面描述的。

11.2. SessionAuthenticationStrategy

SessionAuthenticationStrategy 被 SessionManagementFilter 和 AbstractAuthenticationProcessingFilter 都是用了， 所以如果你使用了一个自定义的 formlogin 类，比如，你需要把它注入到两者中。 比如，在典型配种，结合命名空间和自定义 bean 看起来像这样：

```
  <http>
    <custom-filter                    position="FORM_LOGIN_FILTER"
ref="myAuthFilter" />
    <session-management
session-authentication-strategy-ref="sas"/>
  </http>

  <beans:bean id="myAuthFilter"

class="org.springframework.security.web.authentication.Username
PasswordAuthenticationFilter">
    <beans:property          name="sessionAuthenticationStrategy"
ref="sas" />
```

```
    <beans:property                    name="authenticationManager"
ref="authenticationManager" />
  </beans:bean>

  <beans:bean id="sas"

class="org.springframework.security.web.authentication.session.
ConcurrentSessionControlStrategy">
    <beans:constructor-arg                 name="sessionRegistry"
ref="sessionRegistry" />
    <beans:property name="maximumSessions" value="1" />
  </beans:bean>
```

11.3. 同步会话

Spring Security 可以防止一个主体同时被一个相同的应用授权多于一个特定的次数。许多 ISV 可以利用这一点来加强协议，网络管理员就很喜欢这点功能，因为它可以防止人们共享登陆账号。 你可以，比如，停止在两个不同的会话中的登陆 web 应用的用户"Batman"。 你可以选择让上一次登录过期，或者当他们尝试重复登录时报告一个错误，防止第二次登录。 注意，如果你使用第二种方式，一个用户如果没有使用注销（而是仅仅关闭了他们的浏览器，比如） 就不能再次登陆了，直到他们之前的会话失效之前。

这个功能在命名空间中已经支持了，所以请参考前面介绍命名空间的章节， 来了解简便的配置方式。有时你需要做些一次自定义的工作。

实现使用了特定版本的 SessionAuthenticationStrategy， 称作 ConcurrentSessionControlStrategy。

Note

之前的验证检测是通过 ProviderManager 实现的， 可能是被注入了一个 ConcurrentSessionController,它可以检测 用户是否视图超过最大会话限制的允许数量。然而，这种方式要求预先创建一个 HTTP 会话， 这是不合理的。在 Spring Security 3 中，用户首先被 AuthenticationManager 验证 一旦认证成功，就会创建一个会话，并进行检测，是否允许另一个会话打开。

如果要使用 concurernt session 支持，你需要向 web.xml 添加如下内容：

```
  <listener>
    <listener-class>

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

另外，你需要把 org.springframework.security.web.authentication.concurrent.ConcurrentSessionFilter 添加到你的 FilterChainProxy 中。ConcurrentSessionFilter 需要两个属性， sessionRegistry，通常是一个 SessionRegistryImpl 的实例。 和 expiredUrl，这指向一个页面，当会话过期的时候就会显示它。 一个配置在命名空间中会创建 FilterChainProxy 和其他 bean 可能看起来就像这样：

```
  <http>
    <custom-filter          position="CONCURRENT_SESSION_FILTER"
ref="concurrencyFilter" />
    <custom-filter   position="AUTHENTICATION_PROCESSING_FILTER"
ref="myAuthFilter" />
```

```
    <session-management
session-authentication-strategy-ref="sas"/>
  </http>

  <beans:bean id="concurrencyFilter"

class="org.springframework.security.web.authentication.concurre
nt.ConcurrentSessionFilter">
    <beans:property name="sessionRegistry" ref="sessionRegistry"
/>
    <beans:property                              name="expiredUrl"
value="/session-expired.htm" />
  </beans:bean>

  <beans:bean id="myAuthFilter"

class="org.springframework.security.web.authentcation.UsernameP
asswordAuthenticationFilter">
    <beans:property         name="sessionAuthenticationStrategy"
ref="sas" />
    ...
  </beans:bean>
```

```
  <beans:bean id="sas"

class="org.springframework.security.web.session.ConcurrentSessi
onControlStrategy">
    <beans:property name="sessionRegistry" ref="sessionRegistry"
/>
    <beans:property name="maximumSessions" value="1" />
  </beans:bean>

  <beans:bean                                    id="sessionRegistry"
class="org.springframework.security.authentication.concurrent.S
essionRegistryImpl" />
```

在 web.xml 添加的监听器，会触发一个 ApplicationEvent 发布到 Spring ApplicationContext 中，每当一个 HttpSession 生成或销毁时。这是非常重要，它允许 SessionRegistryImpl 在会话结束时被提醒。 没有它的话，用户就永远不能再登陆到系统中了，一旦他们超过了他们允许的绘画最大量，即使他们 注销了另外的会话，或会话超时了。

---

[11] 在那些会执行重定向的 认证机制中（比如 form-based），无法使用 SessionManagementFilter 检测， 因为过滤器不会在验证请求过程中被调用。 会话管理功能必须在这些情况下单独处理。

匿名认证

12.1. 概述

通常考虑的一个好的安全事件是采取“deny-by-default”（默认拒绝所有请求的方式）这种情况下，你可以明确指定允许哪些，然后拒绝其他所有操作。 定义未认证用户可以做些什么，也是一种简单情况，特别是对于 web 应用。 多数网站要求用户必须通过一些途径进行认证，不仅仅是一些 URL（比如，首页和登陆页面）。 在这种情况下，更简单的定义访问配置属性，为这些特殊的 URL，而不是把每个资源都当做被保护的资源。 不同的是，有时它最好被成为 ROLE_SOMETHING，默认是这样要求的， 对于这种规定，只有几个例外，比如，登陆，注销，应用的首页。 你也可以从过滤器链中完全忽略他们，这样就可以通过安全控制的检测， 但是，对于其他原因这些不好的，特别是，这些页面的行为与已认证用户不同。

这就是为什么我们使用匿名认证的原因，注意这里没有一个真正的理论来区分“匿名认证”的用户和未认证用户。spring security 匿名认证只是给你一个 更方便的方式来配置你的权限控制属性，调用 servlet API，比如 getCallerPrincipal 比如，也会返回 null，即使这里已经有一个匿名认证对象在 SecurityContextHolder 里的。

这里有一些其他匿名认证发挥作用的请抗，比如当一个审计拦截器查询 SecurityContextHolder 来验证哪些主体用来处理给定的操作。 类可以更好的工作，如果它们知道 SecurityContextHolder 里总是包含一个 Authentication 对象， 永远不会是 null。

12.2. 配置

当使用 spring security 3.0 的 HTTP 配置时，就自动提供了对匿名认证的支持。 可以使用<anonymous>元素进行自定义（或禁用）。 你不需要在这里配置 bean，除非使用了以前的 bean 配置方式。

Spring Security 提供三个类来一起提供匿名认证功能。AnonymousAuthenticationToken 实现了 Authentication，保存着 GrantedAuthority[],用来处理匿名主体。有一个对应的需要链入 ProviderManager 的 AnonymousAuthenticationProvider，可以从中获得 AnonymousAuthenticationTokens。 最后是 AnonymousAuthenticationFilter，需要串链到普通认证机制后面，如果还没有存在的 Authentication 的话，它会自动向 SecurityContextHolder 添加一个 AnonymousAuthenticationToken。 过滤器和认证提供器的配置如下：

```
<bean id="anonymousAuthFilter"
```

```
class="org.springframework.security.web.authentication.AnonymousAuthenticationFilter">
  <property name="key" value="foobar"/>
  <property name="userAttribute" value="anonymousUser,ROLE_ANONYMOUS"/>
</bean>

<bean id="anonymousAuthenticationProvider"

class="org.springframework.security.authentication.AnonymousAuthenticationProvider">
  <property name="key" value="foobar"/>
</bean>
```

这个 key 会在过滤器和认证提供器之间共享， 这样创建的标记可以在以后用到。[12] userAttribute 表达式的格式是 usernameInTheAuthenticationToken,grantedAuthority[,grantedAuthority]。这和 InMemoryDaoImpl 中 userMap 属性的语法一样。

如上面所讲的，匿名认证的好处是，可以对所有的 URL 模式都进行安全配置。 比如：

```
<bean id="filterSecurityInterceptor"
```

```
class="org.springframework.security.intercept.web.access.Filter
SecurityInterceptor">
  <property                          name="authenticationManager"
ref="authenticationManager"/>
  <property                          name="accessDecisionManager"
ref="httpRequestAccessDecisionManager"/>
  <property name="securityMetadata">
    <security:filter-security-metadata-source>
      <security:intercept-url              pattern='/index.jsp'
access='ROLE_ANONYMOUS,ROLE_USER'/>
      <security:intercept-url              pattern='/hello.htm'
access='ROLE_ANONYMOUS,ROLE_USER'/>
      <security:intercept-url             pattern='/logoff.jsp'
access='ROLE_ANONYMOUS,ROLE_USER'/>
      <security:intercept-url              pattern='/login.jsp'
access='ROLE_ANONYMOUS,ROLE_USER'/>
      <security:intercept-url pattern='/**' access='ROLE_USER'/>
    </security:filter-security-metadata-source>
  </property>
</bean>
```

12.3. AuthenticationTrustResolver

简略对匿名认证的讨论，就是 AuthenticationTrustResolver 接口，它对应着 AuthenticationTrustResolverImpl 实现。 这个接口提供了一个 isAnonymous(Authentication)方法，允许感兴趣的类评估认证的特殊状态类型。 在处理 AccessDeniedException 异常的时候，ExceptionTranslationFilter 使用这个接口。 如果抛出了一个 AccessDeniedException 异常，而且认证是匿名类型，那么不会抛出 403（禁止）响应，这个过滤器会展开 AuthenticationEntryPoint，这样主体可以正确验证。 这是一个必要的区别，否则主体会一直被认为"需要"认证""，没有机会通过表单，摘要，或其他普通的认证机制登录。

你会经常看到 ROLE_ANONYMOUS 属性，在上面的拦截器配置中， 被替换成 IS_AUTHENTICATED_ANONYMOUSLY， 这在定义权限控制时是完全相同的。 这时一个使用 AuthenticatedVoter 的例子，可以参考验证章节。 它使用一个 AuthenticationTrustResolver 来处理这个特殊的配置属性，并给匿名用户授权。 AuthenticatedVoter 的方式更强大，因为它允许你区别 匿名，rememberMe 和完全认证用户。如果你不需要这些功能，你可以直接使用 ROLE_ANONYMOUS，这会被 Spring Security 的标准 RoleVoter 处理。

---

[12] key 参数应该没有提供任何真实的安全。 它仅仅用来做一个标志。如果你共享了一个 ProviderManager 包含了一个 AnonymousAuthenticationProvider，在一个场景中。 可能对于一个认证客户端，创建 Authentication 对象 （比如通过 RMI 调用），然后一个恶意的可能提交一个 AnonymousAuthenticationToken，它会创建自己 （选择用户名和权限队列）。如果 key 可以被猜出来，或可以被找到， 这个 token 就可能被匿名提供者获得。这不是一个通常使用中的问题，但是如果你使用 RMI， 你最好使用一个自定义的 ProviderManager，这可以避免匿名供应器 共享你使用的 HTTP 验证机制。

Part IV. 授权

Spring Security 的高级授权能力，为它的受欢迎成都提供了一个更可信服的原因。 无论你选择如何认证-是否使用 Spring Security 提供的机制和供应器，或整合容器或非 Spring Security 认证权限-你可以找到授权服务，可以在你的程序用通过统一和简单的方式使用。

在这部分，我们将探索不同的 AbstractSecurityInterceptor 实现，他们在第一部分都介绍过了。 我们再继续研究如何使用授权，通过使用领域访问对象列表。

验证架构

13.1. 验证

在验证部分简略提过了，所有的 Authentication 实现需要保存在一个 GrantedAuthority 对象数组中。 这就是赋予给主体的权限。 GrantedAuthority 对象通过 AuthenticationManager 保存到 Authentication 对象里，然后从 AccessDecisionManager 读出来，进行授权判断。

GrantedAuthority 是一个只有一个方法接口：

```
String getAuthority();
```

这个方法允许 AccessDecisionManager 获得一个精确的 String 来表示 GrantedAuthority。 通过返回的 String，一个 GrantedAuthority 可以简单的用大多数 AccessDecisionManager"读取"。 如果 GrantedAuthority 不能表示为一个 String，GrantedAuthority 会被看作是"复杂的"，然后返回 null。

一个"复杂的" GrantedAuthority 的例子会是保存了操作列表和授权信息并应用在不同的客户帐号数值的一个实现。 如果要显示这种复杂的 GrantedAuthority，把转换成 String 是非常复杂的，getAuthority()方法的结果应该返回 null。 这会展示给任何 AccessDecisionManager，它需要特别支持 GrantedAuthority 的实现，来了解它的内容。

Spring Security 包含一个具体的 GrantedAuthority 实现，GrantedAuthorityImpl 。 这允许任何用户指定的 String 转换成 GrantedAuthority。 所有 AuthenticationProvider 包含安全结构，它使用 GrantedAuthorityImpl 组装 Authentication 对象。

13.2. 处理预调用

像我们在技术概述一章看到的那样，Spring Security 提供了一些拦截器，来控制对安全对象的访问权限，例如方法调用或 web 请求。一个是否允许执行调用的预调用决定，是由 AccessDecisionManager 实现的。

13.2.1. AccessDecisionManager

这个 AccessDecisionManager 被 AbstractSecurityInterceptor 调用，它用来作最终访问控制的决定。 这个 AccessDecisionManager 接口包含三个方法：

```
void decide(Authentication authentication, Object secureObject,
   List<ConfigAttributeDefinition> config) throws
AccessDeniedException;

boolean supports(ConfigAttribute attribute);

boolean supports(Class clazz);
```

从第一个方法可以看出来，AccessDecisionManager 使用方法参数传递所有信息，这好像在认证评估时进行决定。 特别是，在真实的安全方法期望调用的时候，传递安全 Object 启用那些参数。 比如，让我们假设安全对象是一个 MethodInvocation。 很容易为任何 Customer 参数查询 MethodInvocation，，然后在 AccessDecisionManager 里实现一些有序的安全逻辑，来确认主体是否允许在那个客户上操作。 如果访问被拒绝，实现将抛出一个 AccessDeniedException 异常。

这个 supports(ConfigAttribute) 方法在启动的时候被 AbstractSecurityInterceptor 调用，来决定 AccessDecisionManager 是否可以执行传递 ConfigAttribute。 supports(Class)方法被安全拦截器实现调用，包含安全拦截器将显示的 AccessDecisionManager 支持安全对象的类型。

13.2.2. 基于投票的 AccessDecisionManager 实现

虽然用户可以实现它自己的 AccessDecisionManager 来控制所有授权的方面，Spring Security 包括很多基于投票的 AccessDecisionManager 实现。 Figure 13.1, "投票决议管理器"显示有关的类。

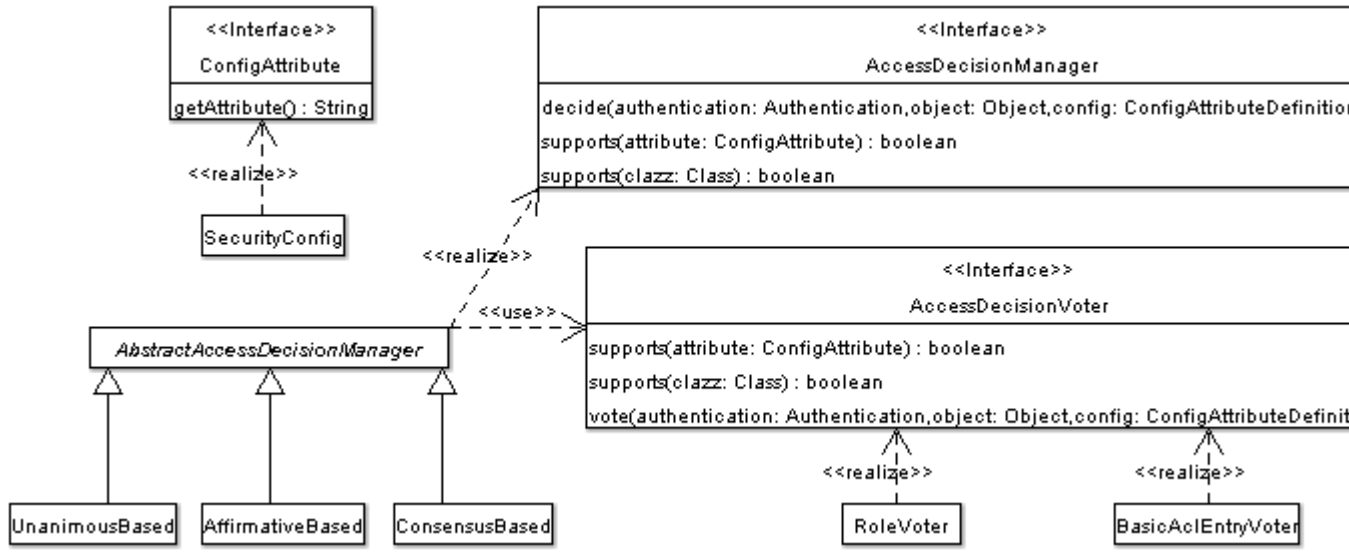

Figure 13.1. 投票决议管理器

使用这种方法，一系列的 AccessDecisionVoter 实现为授权做决定。 这个 AccessDecisionManager 会决定是否基于它的投票评估抛出 AccessDeniedException 异常。

AccessDecisionVoter 接口有三个方法：

```
int vote(Authentication authentication, Object object,
List<ConfigAttributeDefinition> config);
```

```
boolean supports(ConfigAttribute attribute);

boolean supports(Class clazz);
```

具体实现返回一个 int，使用可能的反映的 AccessDecisionVoter 静态属性 ACCESS_ABSTAIN，ACCESS_DENIED 和 ACCESS_GRANTED。 如果一个投票实现没有选择授权决定，会返回 ACCESS_ABSTAIN。 如果它进行过抉择，它必须返回 ACCESS_DENIED 或 ACCESS_GRANTED。

这儿有三个由 Spring Security 提供的具体 AccessDecisionManager，可以进行投票。 ConsensusBased 实现会授权，或拒绝访问，基于没有放弃的那些投票的共识。 那些属性在平等投票时间上被提供来控制行为，或如果所有投票都是弃权了。 AffirmativeBased 实现会授予访问权限，如果收到一个或多个 ACCESS_GRANTED 投票（比如，一个反对投票会被忽略，如果这里至少有一个赞成票）。 像 ConsensusBased 实现一样，这里有一个参数控制如果所有投票都弃权的行为。 UnanimousBased 提供器希望一致的 ACCESS_GRANTED，来允许访问，忽略弃权。 如果这里有任何一个 ACCESS_DENIED 投票，它会拒绝访问。 像其他实现一样，有一个参数控制如果所有投票都弃权的行为。

有可能实现一个自定义的 AccessDecisionManager 进行不同的投票统计。 比如，投票一个特定的 AccessDecisionVoter 可能获得更多的权重，这样一个拒绝票对特定的投票者可能有否决权的效果。

13.2.2.1. RoleVoter

Spring Security 中最常用到的 AccessDecisionVoter 实现是简单的 RoleVoter，它把简单的角色名称作为配置属性，如果用户分配了某个角色就被允许访问。

如果有任何一个配置属性是以 ROLE_ 开头的，就可以进行投票。 如果 GrantedAuthority 返回的 String 内容（通过 getAuthority()方法），与一个或多个以 ROLE_开头的 ConfigAttributes 完全相同的话，就表示允许访问。 如果没有匹配任何一个以 ROLE_开头的 ConfigAttribute，RoleVoter 就会拒绝访问。 如果没有以 ROLE_开头的 ConfigAttribute，投票者就会弃权。 RoleVoter 在匹配的时候是大小写敏感的，这也包括对 ROLE_这个前缀。

13.2.2.2. AuthenticatedVoter

另一个表决器我们不直接看到的是 AuthenticatedVoter， 它可以被用在不同的场景下，比如匿名，完全授权和 remember-me 认证用户。很多网站允许在 rememberMe 认证情况下访问有限的资源，但是需要用户验证 自己的身份通过登录来获得完全访问的权限。

什么时候我们使用 IS_AUTHENTICATED_ANONYMOUSLY 来授权匿名访问呢， 这个属性被 AuthenticatedVoter 处理，参考这个类的 javadoc 获得更多信息。

13.2.2.3. Custom Voters

也可能实现一个自定义的 AccessDecisionVoter。 Spring Security 的单元测试提供了很多例子，包括 ContactSecurityVoter 和 DenyVoter。 如果 CONTACT_OWNED_BY_CURRENT_USER 的 ConfigAttribute 没有找到，ContactSecurityVoter 在投票决定的时候就会弃权。 如果投票，它通过 MethodInvocation 来确认 Contact 对象的主体,这是方法调用的主体。如果 Contact 主体匹配 Authentication 对象中表现的主体，它就会投赞成票。 它可能对 Contact 主体有一个简单的比较，使用一些 Authentication 表现的 GrantedAuthority。 所有这些对应不多几行代码，演示授权模型的灵活性。

13.3. 处理后决定

虽然 AccessDecisionManager 被 AbstractSecurityInterceptor 在执行安全方法调用之前调用，一些程序需要一个方法来修改安全对象调用返回的对象。 虽然你可以简单实现自己的 AOP 涉及实现，Spring Security 提供有许多具体实现方面调用，集成它的 ACL 功能。

Figure 13.2,"后决定实现" 展示 Spring Security 的 AfterInvocationManager 和它的具体实现。

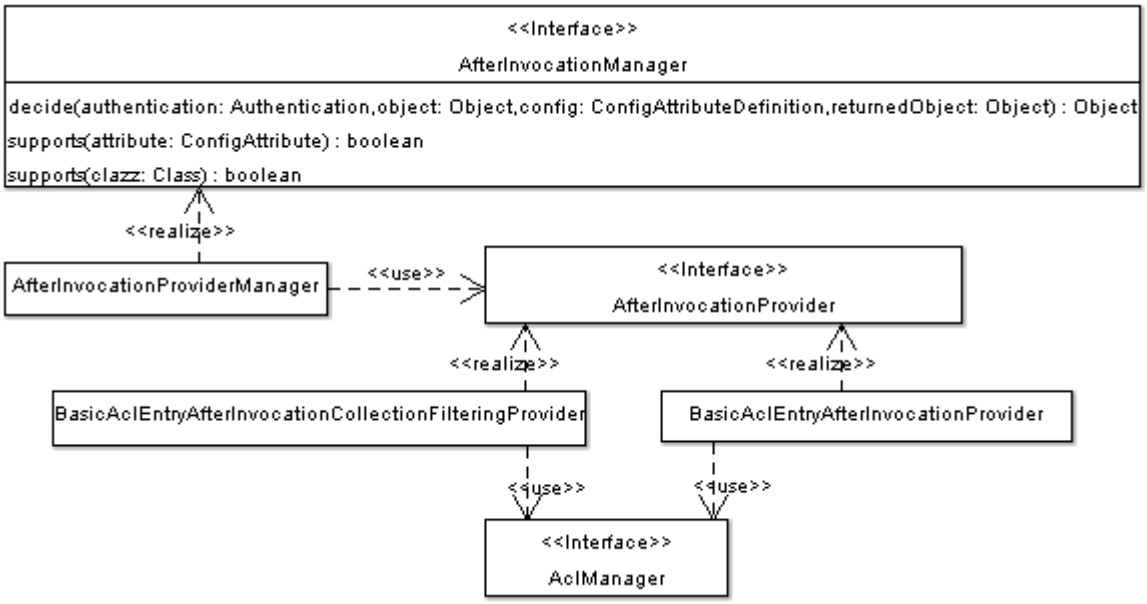

Figure 13.2. 后决定实现

就像 Spring Security 的其他很多部分一样，AfterInvocationManager 有一个单独的具体实现，AfterInvocationProviderManager 有 AfterInvocationProvider 列表。 每个 AfterInvocationProvider 被允许修改返回对象，或抛出 AccessDeniedException 异常。 确实多个提供器可以修改对象，作为之前提供器的结果传递给队列的下一个。 让我们现在考虑我们的 AfterInvocationProvider 的 ACL 提醒实现。

请注意，如果你使用 AfterInvocationManager，你将依然需要配置属性，让 MethodSecurityInterceptor 的 AccessDecisionManager 允许一个操作。 如果你使用典型的 Spring Security 包含 AccessDecisionManager 实现，没有在特定的安全

方法调用上配置属性定义，会导致 AccessDecisionVoter 弃权。 这里，如果 AccessDecisionManager 的"allowIfAllAbstainDecisions"属性是 false，会抛出一个 AccessDeniedException 异常。 你可以避免这个潜在的问题，使用(i)把 "allowIfAllAbstainDecisions" 设置为 true（虽然通常不建议这么做），或者(ii)直接确保这里至少有一个配置属性，这样 AccessDecisionVoter 会投赞成票。 后者（推荐）方法通常使用 ROLE_USER 或 ROLE_AUTHENTICATED 配置属性。

安全对象实现

14.1. AOP 联盟（MethodInvocation）安全拦截器

在 Spring Security 2.0 之前，安全 MethodInvocation 需要进行很多厚重的配置。现在为方法安全推荐的方式，是使用命名空间配置. 使用这个方式，会自动为你配置好方法安全基础 bean，，你不需要了解那些实现类。 我们只需要对这些类提供一个在这里提到的快速概述。

方法安全强制使用 MethodSecurityInterceptor，它会保障 MethodInvocation。依靠配置方法，一个拦截器可能作用于一个单独的 bean 或者在多个 bean 之间共享。拦截器使用 MethodSecurityMetadataSource 实例获得配置属性，应用特定的方法调用。 MapBasedMethodSecurityMetadataSource 通过方法名（也可以是通配符）保存配置属性关键字， 当使用<intercept-methods> 或<protect-point>元素把属性定义在 application context 里，将在内部使用。 其他实现会用来处理基于注解的配置。

14.1.1. 精确的 MethodSecurityIterceptor 配置

你可以使用一个 Spring AOP 代理机制，直接在你的 application context 里配置一个 MethodSecurityIterceptor ：

```
<bean id="bankManagerSecurity"



class="org.springframework.security.intercept.aopalliance.Metho

dSecurityInterceptor">

  <property                          name="authenticationManager"

ref="authenticationManager"/>

  <property                          name="accessDecisionManager"

ref="accessDecisionManager"/>
```

```
  <property                          name="afterInvocationManager"
ref="afterInvocationManager"/>
  <property name="securityMetadataSource">
    <value>
      com.mycompany.BankManager.delete*=ROLE_SUPERVISOR


com.mycompany.BankManager.getBalance=ROLE_TELLER,ROLE_SUPERVISO
R
    </value>
  </property>
</bean>
```

14.2. AspectJ (JoinPoint) 安全拦截器

AspectJ 安全拦截器相对于上面讨论的 AOP 联盟安全拦截器，就非常简单了。 事实上，我们这节只讨论不同的部分。

AspectJ 拦截器的名字是 AspectJSecurityInterceptor。与 AOP 联盟安全拦截器不同，在 Spring 的 application context 中的安全拦截器通过代理织入，AspectJSecurityInterceptor 是通过 AspectJ 编译器织入。在一个系统里使用两种类型的安全拦截器也是常见的，使用 AspectJSecurityInterceptor 处理领域对象实例的安全，AOP 联盟 MethodSecurityInterceptor 用来处理服务层安全。

让我们首先考虑如何把 AspectJSecurityInterceptor 配置到 spring 的 application context 里：

```
<bean id="bankManagerSecurity"


class="org.springframework.security.intercept.aspectj.AspectJSe
curityInterceptor">
```

```
  <property                              name="authenticationManager"
ref="authenticationManager"/>
  <property                              name="accessDecisionManager"
ref="accessDecisionManager"/>
  <property                             name="afterInvocationManager"
ref="afterInvocationManager"/>
  <property name="securityMetadataSource">
    <value>
        com.mycompany.BankManager.delete*=ROLE_SUPERVISOR


com.mycompany.BankManager.getBalance=ROLE_TELLER,ROLE_SUPERVISO
R
    </value>
</property>
</bean>
```

像你看到的，除了类名的部分，AspectJSecurityInterceptor 其实与 AOP 联盟安全拦截器一样。 实际上，两个拦截器共享同样的 securityMetadataSource，securityMetadataSource运行的时候使用 java.lang.reflect.Method而不是AOP库特定的类。 当然，你的访问表决，已经获得了 AOP 库具体的引用（比如 MethodInvocation 或 JoinPoint），也可以考虑在使用防伪决议时进行更精确的判断（比如利用方法参数）。

下一步，你需要定义一个 AspectJ 切面。比如：

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

```
import
org.springframework.security.intercept.aspectj.AspectJSecurityI
nterceptor;
import
org.springframework.security.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");
  }
}
```

在上面例子里，安全拦截器会作用在每一个 PersistableEntity 实例上，这是没提到过的一个抽象类（你可以使用任何其他的类或你喜欢的切点）。 对于那些情况 AspectJCallback 是必须的，因为 proceed();语句只有在 around()内部才有意义。 AspectJSecurityInterceptor 在想继续执行目标对象时，调用这个匿名 AspectJCallback 类。

你会需要配置 Spring 读取切面，织入到 AspectJSecurityInterceptor 中。 下面的声明会处理这个：

```
<bean id="domainObjectInstanceSecurityAspect"

class="org.springframework.security.samples.aspectj.DomainObjec
tInstanceSecurityAspect"
        factory-method="aspectOf">
<property                          name="securityInterceptor"><ref
bean="aspectJSecurityInterceptor"/></property>
</bean>
```

就是这个了！现在你可以在你的系统里任何地方创建 bean 了，无论用你想到的什么方式（比如 new Person();），他们都会被安全拦截器处理。

Chapter 15. 基于表达式的权限控制

Spring Security 3.0 介绍了使用 Spring EL 表达式的能力，作为一种验证机制 添加简单的配置属性的使用和访问决策投票，就像以前一样。 基于表达式的安全控制是建立在相同架构下的，但是允许使用复杂的布尔逻辑 包含在单独的表达式中。

15.1. 概述

Spring Security 使用 Spring El 来支持表达式，你应该看一下如何工作的 如果你对深入了解这个话题感兴趣的话。表达式是执行在一个"根对象" 上的，作为一部分执行上下文。Spring Security 使用特定的类对应 web 和方法安全，作为根对象， 为了提供内建的表达式，访问值，比如当前的认证主体。

15.1.1. 常用内建表达式

表达式根对象的基本类是 SecurityExpressionRoot。这个提供了一些常用的表达式，都可以在 web 和 method 权限控制中使用。

Table 15.1. 常用内建表达式

表达式	说明
hasRole([role])	返回 true 如果当前主体拥有特定角色。
hasAnyRole([role1,role2])	返回 true 如果当前主体拥有任何一个提供的角色（使用逗号分隔的字符串队列）
principal	允许直接访问主体对象，表示当前用户
authentication	允许直接访问当前 Authentication 对象 从 SecurityContext 中获得
permitAll	一直返回 true
denyAll	一直返回 false
isAnonymous()	如果用户是一个匿名登录的用户 就会返回 true

表达式	说明
isRememberMe()	如果用户是通过 remember-me 登录的用户 就会返回 true
isAuthenticated()	如果用户不是匿名用户就会返回 true
isFullyAuthenticated()	如果用户不是通过匿名也不是通过 remember-me 登录的用户时， 就会返回 true。

15.2. Web 安全表达式

为了保护单独的 URL，你需要首先把<http>的 use-expressions 属性设置为 true。Spring Security 会把 <intercept-url>的 access 当做包含了 Spring EL 表达式。表达式应该返回 boolean，定义访问是否应该被允许，比如：

```
  <http use-expressions="true">
    <intercept-url pattern="/admin*"
        access="hasRole('admin')                                    and
hasIpAddress('192.168.1.0/24')"/>
    ...
  </http>
```

这里我们已经定义了应用的"admin"范围（通过 URL 模式定义的） 应该只对拥有"admin"权限的用户有效，并且用户要在本地子网的 IP 地址下。 我们已经看到了内建的 hasRole 表达式，在上一章节。 表达式 hasIpAddress 是一个附加的内建表达式，是由 web 安全提供的。 它是由 WebSecurityExpressionRoot 定义的，它的实例作为表达式根对象 当执行 web 权限表达式时。这个对象也直接暴露了 HttpServletRequest 对象 使用 request 这个名字，所以你可以直接调用 request 在一个表达式里。

如果使用了表达式,一个 WebExpressionVoter 会被添加到 AccessDecisionManager 中，这是被命名空间创建的。 所以如果你没有使用命名空间还希望使用表达式，你必须把这些添加到你的配置中。

15.3. 方法安全表达式

方法安全有一点儿复杂，比单独允许和拒绝规则来说。Spring Security 3.0 介绍了一些新注解， 为了对复杂的表达式进行支持。

15.3.1. @Pre 和 @Post 注解

这里有四个注解，支持表达式属性允许进行前置和后置调用验证检测，也支持过滤提交的 集合参数或返回值。它们是@PreAuthorize, @PreFilter, @PostAuthorize 和 @PostFilter。它们可以通过 global-method-security 命名空间元素启用：

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

15.3.1.1. 访问控制使用 @PreAuthorize 和 @PostAuthorize

使用最广泛的注解是@PreAuthorize，它可以决定一个方法是否可以被调用。 比如（来自"Contacts"实例应用）

```
  @PreAuthorize("hasRole('ROLE_USER')")

  public void create(Contact contact);
```

这意味着只允许拥有"ROLE_USER"角色的用户访问。 很明显，相同的事情可以简单实用传统的配置方法，简单的配置属性来要求角色。 但是如果是这样呢：

```
  @PreAuthorize("hasPermission(#contact, 'admin')")

  public void deletePermission(Contact contact, Sid recipient,

Permission permission);
```

这里我们其实使用了方法参数作为表达式的一部分来决定是否当前的用户拥有 "admin" 表达式对给定的 contract。内建的 hasPermission() 表达式链接到 Spring Security ACL 模块，通过 applicaton context。你可以访问任何方法参数，通过名称就像表达式的变量，在编译的时候使用 debug 信息。任何 Spring EL 功能都可以在表达式里使用， 所以你可以访问参数的属性。比如，如果你希望特定的方法只允许访问一个用户名和 contract 匹配， 你可以写成

```
  @PreAuthorize("#contact.name == principal.name)")
```

```
  public void doSomething(Contact contact);
```

这里我们使用另一个内建表达式，它是当前 Spring Security 从 security context 中获得的 Authentication 的 principal。你也可以直接访问 Authentication 对象自己， 使用表达式名 authentication。

不太常见的是，你可能希望之星一个访问控制检测，在方法调用之后。这个可以使用 @PostAuthorize 注解。为了访问一个方法的返回值， 使用表达式中的内建名称 returnObject。

15.3.1.2. 过滤使用 @PreFilter 和 @PostFilter

你可能已经知道了，Spring Security 支持对集合和数据的过滤， 现在也可以使用表达式实现了。通常是用在一个方法的返回值中。比如：

```
  @PreAuthorize("hasRole('ROLE_USER')")

  @PostFilter("hasPermission(filterObject,       'read')       or

hasPermission(filterObject, 'admin')")

  public List<Contact> getAll();
```

当使用@PostFilter 注解时，Spring Security 遍历返回的集合 删除任何表达式返回 false 的元素。名字 filterObject 引用 集合中的当前对象。你也可以在方法调用之前进行过滤，使用@PreFilter 虽然很少需要这样做。这个语法是一样的，但是如果这里有更多参数，都是集合类型 然后你必须使用注解的 filterTarget 属性选择其中一个。

注意过滤显然不是一个让你读取数据查询的解决方法。如果你过滤大型集合，删除很多实体， 然后这就会是非常没有效率的。

Part V. 高级话题

在这部分，我们会介绍一些这些框架的更高级特性，它们需要前面章节的一些知识，这些功能并不常用到。

领域对象安全(ACLs)

16.1. 概述

复杂程序常常需要定义访问权限，不是简单的 web 请求或方法调用级别。而是，安全决议需要包括谁（认证），哪里（MethodInvocation）和什么（一些领域对象）。 换而言之，验证决议也需要考虑真实的领域对象实例，方法调用的主体。

想像我们为宠物店设计一个程序。 在你的基于 Spring 程序里有两个主要的用户组：宠物商店的工作人员和宠物商店的顾客。 工作人员可以访问所有数据，而你的顾客只能看到他自己的数据。 让它更有趣一点儿，你的客户可以允许其他用户看他自己的数据，比如他们"学龄前小狗"教练，或他们本地"小马俱乐部"的负责人。 以 Spring Security 为基础，我们可以使用很多方法：

- 编写你的业务方法来提升安全。 你可以使用一个集合，包含 Customer 领域对象实例，来决定哪个用户可以访问。 通过 SecurityContextHolder.getContext().getAuthentication()， 你可以得到 Authentication 对象。
- 编写一个 AccessDecisionVoter 提升安全性，通过保存在 Authentication 对象里的 GrantedAuthority[]。 这意味着你的 AuthenticationManager 需要使用自定义 GrantedAuthority[]组装这个 Authentication，处理每个主体访问的 Customer 领域对象实例。
- 编写一个 AccessDecisionVoter 提升安全性，直接打开目标 Customer 领域对象。 这意味着你的投票者需要访问一个 DAO，允许它重审 Customer 对象。 它会访问用户提交的 Customer 对象的集合，然后执行合适的决议。

每个方法都是完全可用的。 然而，你的第一种认证会涉及你的业务代码。 它的主要问题是单元测试困难，也很难在其他地方重用 Customer 的授权逻辑。 从 Authentication 获得 GrantedAuthority[]也还好，但是不适合大规模数量的 Customer。 如果用户可以访问五万个 Customer（不是在这个例子里，但是想像一下，如果它是一个大型的小马俱乐部），这么大的内存消耗，和时间消耗，建造 Authentication 是不可取的。 最后一个方法，直接从外部代码打开 Customer，可能是三个中最好的了。 它分离了概念，没有滥用内存或 CPU 周期，但它还是没什么效率，在 AccessDecisionVoter 和最终业务方法里，它自己会执行一个 DAO 响应，来重申 Customer 对象。 每个方法调用，都要评估两次，非常不可取。 另外，每个方法列出了你需要，从头写自己访问控制列表（ACL）持久化和业务逻辑。

幸运的是，这里有另一个选择，我们在下面讨论。

16.2. 关键概念

Spring Security 的 ACL 服务放在 spring-security-acl-xxx.jar 中。 你需要把这个 JAR 添加到你的 classpath 下，来使用 Spring Security 的领域对象实例安全能力。

Spring Security 的领域对象实例安全能力其实是一个访问控制列表（ACL）的概念。在你的系统中每个领域对象实例都有它自己的 ACL，然后这个 ACL 数据信息，谁可以，谁不可以和领域对象工作。 在这种思想下，Spring Security 在你的系统提供三个主要的 ACL 相关能力：

- 一个有效的方法，为所有你的领域对象（修改那些 ACL）检索 ACL 条目。
- 一个方法，在方法调用之前确认给定的主体有权限同你的对象工作。
- 一个方法，在方法调用之后确认给定的主体有权限同你的对象工作（或它返回的什么东西）。

像第一点所示，Spring Security 的 ACL 模块的一个主要能力，是提供高性能的检索 ACL。 这个 ACL 资源能力特别重要，因为在你的系统中每个领域对象实例，可能有多

个访问控制条目,每个 ACL 可能继承其他 ACL,像一个树形结构(这是 Spring Security 支持的,非常常用)。 Spring Security 的 ACL 能力仔细定义来支持高性能检索 ACL,可插拔缓存,最小死锁数据库更新,不依赖 ORM(我们直接使用的 JDBC),适当封装,数据库透明更新。

给定的数据库是 ACL 模块操作的中心,让我们来看看默认实现使用的四个主要表。 下面介绍的这些表,为了 Spring Security ACL 的部署,使用的表在最后列出:

- ACL_SID 让我们定义系统中唯一主体或授权("SID"意思是"安全标识")。 它包含的列有 ID,一个文本类型的 SID,一个标志,用来表示是否使用文本显示引用的主体名或一个 GrantedAuthority。 因此,对每个唯一的主体或 GrantedAuthority 都有单独一行。 在使用获得授权的环境下,一个 SID 通常叫做"recipient"授予者。
- ACL_CLASS 让我们在系统中确定唯一的领域对象类。包含的列有 ID 和 java 类名。 因此,对每个我们希望保存 ACL 权限的类都有单独一行。
- ACL_OBJECT_IDENTITY 为系统中每个唯一的领域对象实例保存信息。列包括 ID,指向 ACL_CLASS 的外键,唯一标识,所以我们知道为哪个 ACL_CLASS 实例提供信息,parent,一个外键指向 ACL_SID 表,展示领域对象实例的拥有者,我们是否允许 ACL 条目从任何父亲 ACL 继承。 我们对每个领域对象实例有一个单独的行,来保存 ACL 权限。
- 最后,ACL_ENTRY 保存分配给每个授予者单独的权限。 列包括一个 ACL_OBJECT_IDENTITY 的外键,recipient(比如一个 ACL_SID 外键),我们是否通过审核,和一个整数位掩码,表示真实的权限被授权或被拒绝。 我们对于每个授予者都有单独一行,与领域对象工作获得一个权限。

像上一段提到的,ACL 系统使用整数位掩码。 不要担心,你不需要知道使用 ACL 系统位转换的好处,但我们有充足的 32 位可以转换。 每个位表示一个权限,默认授权是可读(位 0),写(位 1),创建(位 2),删除(位 3)和管理(位 4)。 如果你希望使用其他权限,很容易实现自己的 Permission 实例,其他的 ACL 框架部分不了解你的扩展,依然可以运行。

了解你的系统中领域对象的数量很重要,完全用不害怕我们选择使用整数位掩码的事实。 虽然我们有 32 位可用来作权限,你可能有几亿领域对象实例(意味着在 ACL_OBJECT_IDENTITY 表中有几亿行,ACL_ENTRY 也很可能是这样)。我们说出这点,因为我们有时发现人们犯错误,决定他们为每个潜在的领域对象提供一位,情况并非如此。

现在我们提供了 ACL 系统可以做的基本概述,它看起来像一个表结构,现在让我们探讨关键接口。 关键接口是:

- Acl: 每个领域对象有一个,并只有一个 Acl 对象,它的内部保存着 AccessControlEntry,记住这是 Acl 的所有者。 一个 Acl 不直接引用领域对象,但是作为替代的是使用一个 ObjectIdentity。 这个 Acl 保存在 ACL_OBJECT_IDENTITY 表里。
- AccessControlEntry: 一个 Acl 里有多个 AccessControlEntry,在框架里常常略写成 ACE。 每个 ACE 引用特别的 Permission,Sid 和 Acl。 一个 ACE 可以授权或不授权,包含审核设置。 ACE 保存在 ACL_ENTRY 表里。

- Permission: 一个 permission 表示特殊不变的位掩码，为位掩码和输出信息提供方便的功能。 上面的基本权限（位 0 到 4）保存在 BasePermission 类里。
- Sid: 这个 ACL 模块需要引用主体和 GrantedAuthority[]。 间接的等级由 Sid 接口提供，简写成"安全标识"。 通常类包含 PrincipalSid（表示主体在 Authentication 里）和 GrantedAuthoritySid。 安全标识信息保存在 ACL_SID 表里。
- ObjectIdentity: 每个领域对象放在 ACl 模型的内部，使用 ObjectIdentity。默认实现叫做 ObjectIdentityImpl。
- AclService: 重审 Acl 对应的 ObjectIdentity。 包含的实现（JdbcAclService），重审操作代理 LookupStrategy。 这个 LookupStrategy 为检索 ACL 信息提供高优化策略，使用批量检索（BasicLookupStrategy）然后支持自定义实现，和杠杆物化视图，继承查询和类似的表现中心，非 ANSI 的 SQL 能力。
- MutableAclService: 允许修改了的 Acl 放到持久化中。 如果你不愿意，可以不使用这个接口。

请注意，我们的 AclService 和对应的数据库类都使用 ANSI SQL。 这应该可以在所有的主流数据库上工作。 在写作的时候，系统成功在 Hypersonic SQL, PostgreSQL, Microsoft SQL Server 和 Oracle 上测试通过。

Spring Security 的两个实例演示了 ACL 模块。 第一个是 Contacts 实例，另一个是文档管理系统（DMS）实例。 我们建议大家看一看这些例子。

16.3. 开始

为了开始使用 Spring Security 的 ACL 功能，你会需要在一些地方保存你的 ACL 信息。有必要使用 Spring 的 DataSource 实例。 DataSource 注入到 JdbcMutableAclService 和 BasicLookupStrategy 实例中。 后一个提供了高性能 ACL 检索能力，前一个提供变异能力。 参考例子之一，使用 Spring Security，为一个例子配置。 你也需要使用四个 ACL 指定的表建立数据库，这写在最后一章（参考 ACL 实例，查看对应的 SQL 语句）。

一旦你创建了需要的结构，和 JdbcMutableAclService 的实例，你下一个需求是确认你的领域模型支持 Spring Security ACL 包的互操作。 希望的 ObjectIdentityImpl 会证明足够，它提供可以使用的大量方法。 大部分人会使用领域对象，包含 public Serializable getId()方法。 如果返回类型是 long 或与 long 兼容（比如 int），你会发现你不需要为 ObjectIdentity 进行更多考虑。 ACL 模块的许多部分对应 long 标识符。 如果你没有使用 long（或 int, byte 等等），你需要重新实现很多类。 我们不倾向在 Spring Security ACL 模块中支持非 long 标识符，因为所有数据库序列都支持，最常用的数据类型标识，也可以容纳所有常用场景的足够长度。

下面的代码片段，显示如何创建一个 Acl，或修改一个存在的 Acl：

```
// Prepare the information we'd like in our access control entry

(ACE)
```

```
ObjectIdentity oi = new ObjectIdentityImpl(Foo.class, new
Long(44));
Sid sid = new PrincipalSid("Samantha");
Permission p = BasePermission.ADMINISTRATION;


// Create or update the relevant ACL
MutableAcl acl = null;
try {
  acl = (MutableAcl) aclService.readAclById(oi);
} catch (NotFoundException nfe) {
  acl = aclService.createAcl(oi);
}


// Now grant some permissions via an access control entry (ACE)
acl.insertAce(acl.getEntries().length, p, sid, true);
aclService.updateAcl(acl);
```

在上面的例子里，我们检索 ACl，分配给"Foo"领域对象，使用数字 44 作标识。 我们添加一个 ACE，这样名叫"Samantha"的主体可以"管理"这个对象。 代码片段是自解释的，除了 insertAce 方法。 insertAce 方法的第一个参数是 Acl 里新条目被插入的决定位置。 在上面的的例子里，我们只把新 ACE 放到以存在的 ACE 的尾部。 最后一个参数是一个布尔值，显示是否 ACE 授权或拒绝。 大多数时间，是授权（true），如果它是拒绝（false），权限就会被冻结。

Spring Security 没有提供任何特定整合，自动创建，更新，或删除 ACL，作为你的 DAO 的一部分或资源操作。 作为替代的，你会需要像上面一样为你的单独领域对象写代码。 值得考虑在你的服务层使用 AOP，来自动继承 ACL 信息，使用你的服务层操作。 我们发现以前这是一个非常有效的方式。

一旦，你使用上面的技术，在数据库里保存一些 ACL 信息，下一步是使用 ACL 信息，作为授权决议逻辑的一部分。 这里你有一大堆选择。 你可以写你自己的 AccessDecisionVoter 或 AfterInvocationProvider，期待在方法调用之前或之后触发。 这些类使用 AclService 来检索对应的 ACL，然后调用 Acl.isGranted(Permission[] permission, Sid[] sids, boolean administrativeMode)，决定权限是授予还是拒绝。 可选的，你可能使用我们的 AclEntryVoter，AclEntryAfterInvocationProvider 或 AclEntryAfterInvocationCollectionFilteringProvider 类。 所有这些类提供一个基于声明的方法，在运行阶段来执行 ACL 信息，释放你从需要写任何代码。 请参考例子程序，学习更多如何使用这些类。

预认证场景

有的情况下，你需要使用 Spring Security 进行认证，但是用户已经在访问系统之前，在一些外部系统中认证过了。 我们把这种情况叫做"预认证"场景。 例子包括 X.509，Siteminder 和应用所在的 J2EE 容器进行认证。 在使用预认证的使用，Spring Security 必须

- 定义使用请求的用户
- 从用户里获得权限

细节信息要依靠外部认证机制。 一个用户可能，在 X.509 的情况下由认证信息确定，或在 Siteminder 的情况下使用 HTTP 请求头。 对于容器认证，需要调用获得 HTTP 请求的 getUserPrincipal()方法来确认用户。 一些情况下，外部机制可能为用户提供角色/权限信息，其他情况就需要通过单独信息源获得，比如 UserDetailsService。

17.1. 预认证框架类

因为大多数预认证机制都遵循相同的模式，所以 Spring Security 提供了一系列的类，它们作为内部框架实现预认证认证提供器。 这样就避免了重复实现，让新实现很容易添加到结构中，不需要一切从脚手架开始写起。 你不需要知道这些类，如果你想使用一些东西，比如 X.509 认证，因为它已经是命名空间配置里的一个选项了，可以很简单的使用，启动它。 如果你需要使用精确的 bean 配置，或计划编写你自己的实现，这时了解这些提供的实现是如何工作就很有用了。 你会在 org.springframework.security.web.authentication.preauth 包下找到这些类。我们这里只提供一个纲要，你应该从对应的 Javadoc 和源代码里获得更多信息。

17.1.1. AbstractPreAuthenticatedProcessingFilter

这个类会检测安全环境的当前内容，如果是空的，它会从 HTTP 请求里获得信息，提交给 AuthenticationManager。 子类重写了以下方法来获得信息：

```
  protected                    abstract                    Object

getPreAuthenticatedPrincipal(HttpServletRequest request);
```

```
  protected                    abstract                    Object
getPreAuthenticatedCredentials(HttpServletRequest request);
```

在调用之后，过滤器会创建一个包含了返回数据的PreAuthenticatedAuthenticationToken，然后提交它进行认证。 通过这里的"authentication"，我们其实只是可能进行读取用户的权限，不过下面就是标准的Spring Security认证结构了。

17.1.2. AbstractPreAuthenticatedAuthenticationDetailsSource

就像其他的 Spring Security 认证过滤器一样，预认证过滤器有一个authenticationDetailsSource属性,默认会创建一个WebAuthenticationDetails对象来保存额外的信息，比如在Authentication对象的details属性里的会话标识，原始 IP 地址。 用户角色信息可以从预认证机制中获得，数据也保存在这个属性里。AbstractPreAuthenticatedAuthenticationDetailsSource 的子类，使用实现了GrantedAuthoritiesContainer接口的扩展信息，因此可以使用认证提供其器来读取权限，明确定位用户。 下面我们看一个具体的例子。

17.1.2.1. J2eeBasedPreAuthenticatedWebAuthenticationDetailsSource

如果过滤器配置了 authenticationDetailsSource 的实例，通过调用isUserInRole(String role)方法为每个预先决定的"可映射角色"集合获得认证信息。 这个类从MappableAttributesRetriever里获得这些信息。 可能的实现方法，包含了在application context中进行硬编码，或者从web.xml的<security-role>中读取角色信息。 预认证例子程序使用了后一种方式。

这儿有一个额外的步骤，使用一个Attributes2GrantedAuthoritiesMapper把角色（或属性）映射到Spring Security的GrantedAuthority。 它默认只会为名称添加一个ROLE_前缀，但是你可以对这些行为进行完全控制。

17.1.3. PreAuthenticatedAuthenticationProvider

预认证提供器除了从用户中读取UserDetails以外，还要一些其他事情。 它通过调用一个 AuthenticationUserDetailsService 来做这些事情。 后者就是一个标准的UserDetailsService,但要需要的参数是一个Authentication对象,而不是用户名:

```
  public interface AuthenticationUserDetailsService {
    UserDetails  loadUserDetails(Authentication  token)  throws
UsernameNotFoundException;
  }
```

这个接口可能也含有其他用户，但是预认证允许访问权限，打包在 Authentication 对象里，像上一节所见的。这个 PreAuthenticatedGrantedAuthoritiesUserDetailsService 就是用来作这个的。或者，它可能调用标准 UserDetailsService，使用 UserDetailsByNameServiceWrapper 这个实现。

17.1.4. Http403ForbiddenEntryPoint

这个 AuthenticationEntryPoint 在 技术概述 那章讨论过。 通常它用来为未认证用户（当他们想访问被保护资源的时候）启动认证过程，但是在预认证情况下这不会发生。 如果你不使用预认证结合其他认证机制的话，你只要配置 ExceptionTranslationFilter 的一个实例。如果用户的访问被拒绝了，它就会调用，AbstractPreAuthenticatedProcessingFilter 结果返回的一个空的认证。 调用的时候，它总会返回一个 403 禁用响应代码。

17.2. 具体实现

X.509 认证写在它自己的章里。 这里，我们看一些支持其他预认证的场景。

17.2.1. 请求头认证（Siteminder）

一个外部认证系统可以通过在 HTTP 请求里设置特殊的头信息，给应用提供信息。 一个众所周知的例子就是 Siteminder，它在头部传递用户名，叫做 SM_USER。 这个机制被 RequestHeaderPreAuthenticatedProcessingFilter 支持，直接从头部得到用户名。 默认使用 SM_USER 作为头部名。 看一下 Javadoc 获得更多信息。

Tip

注意使用这种系统时，框架不需要作任何认证检测，*极端*重要的是，要把外部系统配置好，保护系统的所有访问。 如果攻击者可以从原始请求中获得请求头，不通过检测，他们可能潜在修改任何他们想要的用户名。

17.2.1.1. Siteminder 示例配置

使用这个过滤器的典型配置应该像这样：

```
<security:http>
  <!-- Additional http configuration omitted -->
  <security:custom-filter ref="siteminderFilter" />
</security:http>

  <bean id="siteminderFilter"
```

```
class="org.springframework.security.web.authentication.preauth.
header.RequestHeaderAuthenticationFilter">
    <property name="principalRequestHeader" value="SM_USER"/>
    <property                          name="authenticationManager"
ref="authenticationManager" />
  </bean>

  <bean id="preauthAuthProvider"

class="org.springframework.security.web.authentication.preauth.
PreAuthenticatedAuthenticationProvider">
    <property name="preAuthenticatedUserDetailsService">
      <bean id="userDetailsServiceWrapper"

class="org.springframework.security.core.userdetails.UserDetail
sByNameServiceWrapper">
        <property                          name="userDetailsService"
ref="userDetailsService"/>
      </bean>
    </property>
    </bean>
```

```
        <security:authentication-manager
alias="authenticationManager">
          <security:authentication-provider
ref="preauthAuthProvider" />
        </security-authentication-manager>
```

我们假设使用安全命名空间的配置方式，custom-filter，使用了 authentication-manager 和 custom-authentication-provider 三个元素（你可以从命名空间章节里了解它们的更多信息）。 你应该走出传统的配置方式。 我们也假设了你在配置里添加了一个 UserDetailsService（名叫"userDetailsService"），来读取用户的角色信息。

17.2.2. J2EE 容器认证

这个 J2eePreAuthenticatedProcessingFilter 类会从 HttpServletRequest 的 userPrincipal 属性里获得准确的用户名。 这个过滤器的用法常常结合使用 J2EE 角色，像上面描述的 Section 17.1.2.1, "J2eeBasedPreAuthenticatedWebAuthenticationDetailsSource"。

在根目录下有一个使用这个功能的实例应用，可以从 svn 里找到， 如果你感兴趣的话，可以看一下 application context 文件。 代码在 samples/preauth 目录下。

LDAP 认证

18.1. 综述

LDAP 通常被公司用作用户信息的中心资源库，同时也被当作一种认证服务。 它也可以为应用用户储存角色信息。

这里有很多如何对 LDAP 服务器进行配置的场景，所以 Spring Security 的 LDAP 提供器也是完全可配置的。 它使用为验证和角色检测提供了单独的策略接口，并提供了默认的实现，这些都是可配置成处理绝大多数情况。

你还是应该熟悉一下 LDAP，在你在 Spring Security 使用它之前。 下面的链接提供了很好的概念介绍，也是一个使用免费的 LDAP 服务器建立一个目录 http://www.zytrax.com/books/ldap/的指南。我们也应该熟悉一下通过 JNDI API 使用 java 访问 LDAP。我们没有在 LDAP 提供器里使用任何第三方 LDAP 库（Mozilla,

JLDAP 等等），但是还是用到了 Spring LDAP，所以如果你希望自己进行自定义，对这个工程熟悉一下也是有好处的。

18.2. 在 Spring Security 里使用 LDAP

Spring Security 的 LDAP 认证可以粗略分成以下几部分。

- 从登录名中获得唯一的"辨别名称"或 DN。 这就意味着要对目录执行搜索，除非预先知道了用户名和 DN 之前的明确映射关系。
- 验证这个用户，进行绑定用户，或调用远程"比较"操作，比对用户的密码和 DN 在目录入口中的密码属性。
- 为这个用户读取权限队列。

例外情况是，当 LDAP 目录只是用来检索用户信息和进行本地验证的时候，这也许不可能的，因为目录的属性，比如对用户密码属性，常常被设置成只读权限。

让我们看看下面的一些配置场景。 要是想得到所有可用的配置选项，请参考安全命名空间结构（使用你的 XML 编辑器应该就可以看到所有有效信息）。

18.3. 配置 LDAP 服务器

你需要做的第一件事是配置服务器，它里面应该存放着认证信息。 这可以使用安全命名空间里的<ldap-server>元素实现。 使用 url 属性指向一个外部 LDAP 服务器：

```
  <ldap-server

url="ldap://springframework.org:389/dc=springframework,dc=org"

/>
```

18.3.1. 使用嵌入测试服务器

这个<ldap-server>元素也可以用来创建一个嵌入服务器，这在测试和演示的时候特别有用。 在这种情况，你不需要使用 url 属性：

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

这里我们指定目录的根 DIT 应该是"dc=springframework,dc=org"，这是默认的。使用这种方式，命名空间解析器会建立一个嵌入 Apache 目录服务器，然后检索

classpath 下的 LDIF 文件，尝试从它里边把数据加载到服务器里。 你可以通过 ldif 属性自定义这些行为，这样可以定义具体要加载哪个 LDIF 资源：

```
<ldap-server ldif="classpath:users.ldif" />
```

这就让启动和运行 LDAP 变得更轻松了，因为使用一个外部服务器还是不大方便。 它也避免链接到 Apache 目录服务器的复杂 bean 配置。 如果使用普通 Spring bean 配置方法会变的更加混乱。你必须把必要的 Apache 目录依赖的 jar 放到你的程序中。这些都可以从 LDAP 示例程序中获得。

18.3.2. 使用绑定认证

这是一个非常常见的 LDAP 认证场景。

```
<ldap-authentication-provider

user-dn-pattern="uid={0},ou=people"/>
```

这个很简单的例子可以根据用户登录名提供的模式为用户获得 DN，然后尝试和用户的登录密码进行绑定。 如果所有用户都保存到一个目录的单独节点下就没有问题。 如果你想配置一个 LDAP 搜索过滤器来定位用户，你可以使用如下配置：

```
<ldap-authentication-provider user-search-filter="(uid={0})"

  user-search-base="ou=people"/>
```

如果使用了上面的服务器定义，它会在 DNou=people,dc=springframework,dc=org 下执行搜索，使用 user-search-filter 里的值作为过滤条件。 然后把用户登录名作为过滤名称的一个参数。 如果没有提供 user-search-base，搜索将从根开始。

18.3.3. 读取授权

如果从 LDAP 目录的组里读取权限信息呢，这是通过下面的属性控制的。

- group-search-base。定义目录树部分，哪个组应该执行搜索。
- group-role-attribute。这个属性包含了组入口中定义的权限名称。默认是 cn
- group-search-filter 。这个过滤器用来搜索组的关系。 默认是 uniqueMember={0}，对应于 groupOfUniqueMembersLDAP 类。 在这情况下，取代参数是用户的辨别名称。 如果你想对登录名搜索，可以使用{1}这个参数。

因此，如果我们使用下面进行配置

```
  <ldap-authentication-provider

user-dn-pattern="uid={0},ou=people"

    group-search-base="ou=groups" />
```

并以用户"ben"的身份通过认证，在读取权限信息的子流程里，要在目录入口 ou=groups,dc=springframework,dc=org 下执行搜索，查找包含 uniqueMember 属性值为 ou=groups,dc=springframework,dc=org 的入口。默认,权限名都要以 ROLE_ 作为前缀。 你可以使用 role-prefix 属性修改它。 如果你不想使用任何前缀，可以使用 role-prefix="none"。 要想得到更多读取权限的信息，可以查看 DefaultLdapAuthoritiesPopulator 类的 Javadoc。

18.4. 实现类

我们上面使用到的命名空间选项很容易使用，也比使用 spring bean 更准确。 也有可能你需要知道如何配置在你的 application context 里配置 Spring Security LDAP 目录。 比如，你可能想自定义一些类的行为。 如果你想使用命名空间配置，你可以跳过这节，直接进入下一段。

最主要的 LDAP 提供器类是 LdapAuthenticationProvider。 它自己没做什么事情，而是代理了其他两个 bean 的工作，一个是 LdapAuthenticator，一个是 LdapAuthoritiesPopulator，用来处理用户认证和检索用户的 GrantedAuthority 属性集合。

18.4.1. LdapAuthenticator 实现

验证者还负责检索所有需要的用户属性。 这是因为对于属性的授权可能依赖于使用的验证类型 比如，如果对某个用户进行绑定，它也许必须通过用户自己的授权才能进行读取。

当前 Spring Security 提供两种验证策略：

- 直接去 LDAP 服务器验证（"绑定"验证）。
- 比较密码，将用户提供的密码与资源库中保存的进行比较。 这可以通过检索密码属性的值并在本地检测，或者执行 LDAP"比较"操作，提供用来比较的密码是从服务器获得的，绝对不会检索真实密码的值。

18.4.1.1. 常用功能

在认证一个用户之前（使用任何一个策略），辨别名称（DN）必须从系统提供的登录名中获得。 这可以通过，简单的模式匹配（设置 setUserDnPatterns 数组属性）或者设置 userSearch 属性。 为了实现 DN 模式匹配方法，一个标准的 java 模式格式被用到了，登录名将被参数 *{0}* 替代。 这个模式应该和 DN 有关系，并绑定到配置好的 SpringSecurityContextSource（看看链接到 LDAP 服务器那节，获得更多信息）。比如，如果你使用了 LDAP 服务的 URL 是

ldap://monkeymachine.co.uk/dc=springframework,dc=org，并有一个模式uid={0},ou=greatapes，然后登录名"gorilla"会映射到DNuid=gorilla,ou=greatapes,dc=springframework,dc=org。每个配置好的DN模式将尝试进行定位，直到有一个匹配上。使用搜索获得信息，看看下面的安全对象那节。两种方式也可以结合在一起使用 - 模式会先被检测一下，然后如果没有找到匹配的DN，就会使用搜索。

18.4.1.2. BindAuthenticator

这个类BindAuthenticator在这个包里org.springframework.security.ldap.authentication实现了绑定认证策略。它只是尝试对用户进行绑定。

18.4.1.3. PasswordComparisonAuthenticator

这个类PasswordComparisonAuthenticator实现了密码比较认证策略。

18.4.1.4. 活动目录认证

除了标准LDAP认证以外（绑定到一个DN），活动目录对于用户认证提供了自己的非标准语法。

18.4.2. 链接到LDAP服务器

上面讨论的bean必须连接到服务器。它们都必须使用ContextSource，这个是Spring LDAP的一个扩展。除非你有特定的需求，你通常只需要配置一个DefaultSpringSecurityContextSource bean，这个可以使用你的LDAP服务器的URL进行配置，可选项还有管理员用户的用户名和密码，这将默认用在绑定服务器的时候(而不是匿名绑定)。参考Spring LDAP的AbstractContextSource类的Javadoc获得更多信息。

18.4.3. LDAP搜索对象

通常，比简单DN匹配越来越复杂的策略需要在目录里定位一个用户入口。这可以使用LdapUserSearch的一个示例，它可以提供认证者实现，比如让他们定位一个用户。提供的实现是FilterBasedLdapUserSearch。

18.4.3.1. FilterBasedLdapUserSearch

这个bean使用一个LDAP过滤器，来匹配目录里的用户对象。这个过程在javadoc里进行过解释，在对应的搜索方法，JDK DirContext class。就如那里解释的，搜索过滤条件可以通过方法指定。对于这个类，唯一合法的参数是*{0}*，它会代替用户的登录名。

18.4.4. LdapAuthoritiesPopulator

在成功认证用户之后，LdapAuthenticationProvider会调用配置好的LdapAuthoritiesPopulator bean，尝试读取用户的授权集合。这个DefaultLdapAuthoritiesPopulator是一个实现类，它将通过搜索目录读取授权，查找用户成员所在的组(典型的这会是目录中的groupOfNames或groupOfUniqueNames入口）。查看这个类的Javadoc获得它如何工作的更多信息。

如果你只想在验证时使用LDAP，而是从另外的地方读取认证信息（比如数据库）你可以提供你的自己的接口实现，然后使用注入作为替代。

18.4.5. Spring Bean 配置

典型的配置方法，使用到像我们这在里讨论的这些 bean，就像这样：

```
<bean id="contextSource"


class="org.springframework.security.ldap.DefaultSpringSecurityC
ontextSource">
  <constructor-arg
value="ldap://monkeymachine:389/dc=springframework,dc=org"/>
  <property                                          name="userDn"
value="cn=manager,dc=springframework,dc=org"/>
  <property name="password" value="password"/>
</bean>


<bean id="ldapAuthProvider"


class="org.springframework.security.ldap.authentication.LdapAut
henticationProvider">
 <constructor-arg>
   <bean
class="org.springframework.security.ldap.authentication.BindAut
henticator">
     <constructor-arg ref="contextSource"/>
     <property name="userDnPatterns">
```

```
      <list><value>uid={0},ou=people</value></list>
    </property>
  </bean>
 </constructor-arg>
 <constructor-arg>
  <bean


class="org.springframework.security.ldap.userdetails.DefaultLda
pAuthoritiesPopulator">
    <constructor-arg ref="contextSource"/>
    <constructor-arg value="ou=groups"/>
    <property name="groupRoleAttribute" value="ou"/>
  </bean>
 </constructor-arg>
</bean>
```

这里建立了一个提供器，访问 LDAP 服务，URL 是 ldap://monkeymachine:389/dc=springframework,dc=org。 认证会被执行，尝试绑定这个 DN uid=<user-login-name>,ou=people,dc=springframework,dc=org。在成功认证之后，会通过查找下面的 DN ou=groups,dc=springframework,dc=org 使用默认的过滤条件 (member=<user's-DN>)，将角色分配给用户。 角色名会通过每个匹配的"ou"属性获得。

要配置用户的搜索对象，使用过滤条件 (uid=<user-login-name>) 替代 DN 匹配（或附加到它上面），你需要配置下面的 bean

```
<bean id="userSearch"
```

```
class="org.springframework.security.ldap.search.FilterBasedLdap
UserSearch">
  <constructor-arg index="0" value=""/>
  <constructor-arg index="1" value="(uid={0})"/>
  <constructor-arg index="2" ref="contextSource" />
</bean>
```

并使用它，设置认证者的 userSearch 属性。 这个认证者会调用搜索对象，在尝试绑定到用户之前获得正确的用户 DN。

18.4.6. LDAP 属性和自定义 UserDetails

使用 LdapAuthenticationProvider 进行认证的结果，和使用普通 Spring Security 认证一样，都要使用标准 UserDetailsService 接口。 它会创建一个 UserDetails 对象，并保存到返回的 Authentication 对象里。 在使用 UserDetailsService 时，常见的需求是可以自定义这个实现，添加额外的属性。 在使用 LDAP 的时候，这些基本都来自用户入口的属性。 UserDetails 对象的创建结果被提供者的 UserDetailsContextMapper 策略控制，它负责在用户对象和 LDAP 环境数据之间进行映射：

```
public interface UserDetailsContextMapper {
  UserDetails mapUserFromContext(DirContextOperations ctx, String
username,
          Collection<GrantedAuthority> authority);

  void mapUserToContext(UserDetails user, DirContextAdapter ctx);
}
```

只有第一个方法与认证有关。 如果你提供这个接口的实现，你可以精确控制如何创建 UserDetails 对象。 第一个参数是 Spring LDAP 的 DirContextOperations 实例，他给你访问加载的 LDAP 属性的通道。 username 参数是用来认证的名字，最后一个参数是从用户加载的授权列表。]

环境数据加载的方式不同，视乎你采用的认证方法。 使用 BindAuthenticator，从绑定操作返回的环境会用来读取属性，否则数据会通过标准的环境，从配置好的 ContextSource 获得（当测试配置好定位用户，这会从搜索对象中获得数据）。

Chapter 19. JSP 标签库

Spring Security 有它自己的 Taglib，提供了 JSP 中访问权限信息和提供安全约束的功能。

19.1. 声明 Taglib

要想使用这些标签，你必须在你的 JSP 中声明安全 taglib：

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

19.2. authorize 标签

这个标签用来决定它的内容是否会被执行。在 Spring Security 3.0 中， 它可以被用在两种方式中[13]。第一个方式是使用 web-security 表达式， 指定标签中的 access 属性。 表达式执行会被 WebSecurityExpressionHandlder 代理， 这个类定义在 application context 中（你应该在<http>命名空间配置中 启用了 web 表达式，并确认这个服务可以使用）。所以，比如，你可能使用

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


这段内容只能被拥有在他们的<tt>GrantedAuthority</tt>列表中

含有"supervisor"权限的用户才能看到。


</sec:authorize>
```

一个常见的需求是指显示一个特定的链接，如果用户允许点击它。 怎么让我们进一步决定是否一些事情可以允许呢？ 标签也可以用一种可选的模式操作，允许你定义一个

特定的 URL 作为属性。 如果用户允许调用这个 URL，标签内容就会被执行， 否则它会被略过。所以，你可能会使用一些像

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

这些内容之会被有权限发送请求到"/admin" URL 的用户才可以看到。

</sec:authorize>
```

为了使用这个标签，必须在你的 application context 中拥有一个 WebInvocationPrivilegeEvaluator 实例。 如果你使用了命名空间，会自动注册一个。这是一个 DefaultWebInvocationPrivilegeEvaluator 的实例， 它会创建一个默认 web 请求对提供的 URL，调用安全拦截器来查看请求是成功还是失败的。 这允许你代理到访问控制设置，你使用 intercept-url 声明在 <http>命名空间中的配置，保存信息（比如必须的角色）在你的 JSP 中。 这种方式也可以结合 method 属性，提供 HTTP method， 为了更详细的匹配。

19.3. authentication 标签

这个标签允许访问当前的 Authentication 对象， 保存在安全上下文中。它直接渲染一个对象的属性在 JSP 中。 所以，比如，如果 Authentication 的 principal 属性是 Spring Security 的 UserDetails 对象的一个实例，就要使用 <sec:authentication property="principal.username" /> 来渲染当前用户的名称。

当然,它不必使用 JSP 标签来实现这些功能,一些人更愿意在视图中保持逻辑越少越好。你可以在你的 MVC 控制器中访问 Authentication 对象 （通过调用 SecurityContextHolder.getContext().getAuthentication()） 然后直接在模型中添加数据，来渲染视图。

19.4. accesscontrollist 标签

这个标签纸在使用 Spring Security ACL 模块时才可以使用。它检测一个用逗号分隔的 特定领域对象的需要权限列表。如果当前用户拥有这些权限的任何一个，标签内容就会被执行。 否则，就会被略过。一个例子可能像

```
<sec:accesscontrollist                          hasPermission="1,2"

domainObject="someObject">
```

```
 这些将被显示，如果用户拥有指定对象的权限显示为"1"或"2"。



</sec:accesscontrollist>
```

权限会被传递到 PermissionFactory 定义在 application context 中，把它们转换为 ACL 的 Permission 实例，所以他们可以使用 工厂支持的任何格式 - 不是必须使用整数，它们可以是字符串，像 READ 或者 WRITE。如果没有找到 PermissionFactory，一个 DefaultPermissionFactory 实例会被使用。 application context 中的 AclService 会被用来加载 对应的对象的 Acl 实例。 Acl 会被调用，使用需要的权限来检测， 如果它们中的任何一个被授权了。

---

[13] Spring Security 2.0 遗留的方式也是支持的， 但是不推荐使用。

Java 认证和授权服务（JAAS）供应器

20.1. 概述

Spring Security 提供一个包，可以代理 Java 认证和授权服务（JAAS）的认证请求。这个包的细节在下面讨论。

JAAS 的核心是登录配置文件。 想要了解更多 JAAS 登录配置文件的信息，可以查询 Sun 公司的 JAAS 参考文档。 我们希望你对 JAAS 有一个基本了解，也了解它的登录配置语法，这才能更好的理解这章的内容。

20.2. 配置

这个 JaasAuthenticationProvider 通过 JAAS 认证用户的主体和证书。

让我们假设我们有一个 JAAS 登录配置文件，/WEB-INF/login.conf，里边的内容如下：

```
JAASTest {

    sample.SampleLoginModule required;

};
```

就像所有的 Spring Security bean 一样，这个 JaasAuthenticationProvider 要配置在 application context 里。 下面的定义是与上面的 JAAS 登录配置文件对应的：

```
<bean id="jaasAuthenticationProvider"
```

```
class="org.springframework.security.authentication.jaas.JaasAut
henticationProvider">
 <property name="loginConfig" value="/WEB-INF/login.conf"/>
 <property name="loginContextName" value="JAASTest"/>
 <property name="callbackHandlers">
  <list>
   <bean

class="org.springframework.security.authentication.jaas.JaasNam
eCallbackHandler"/>
   <bean

class="org.springframework.security.authentication.jaas.JaasPas
swordCallbackHandler"/>
  </list>
  </property>
  <property name="authorityGranters">
    <list>
      <bean
class="org.springframework.security.authentication.jaas.TestAut
horityGranter"/>
```

```
    </list>
  </property>
</bean>
```

这个 CallbackHandler 和 AuthorityGranter 会在下面进行讨论。

20.2.1. JAAS CallbackHandler

大多数 JAAS 的登录模块需要设置一系列的回调方法。 这些回调方法通常用来获得用户的用户名和密码。

在 Spring Security 发布的时候，Spring Security 负责用户交互（通过认证机制）。因此，现在认证请求使用 JAAS 代理，Spring Security 的认证机制将组装一个 Authentication 对象，它包含了所有 JAASLoginModule 需要的信息。

因此，Spring Security 的 JAAS 包提供两个默认的回调处理器，JaasNameCallbackHandler 和 JaasPasswordCallbackHandler。 他们两个都实现了 JaasAuthenticationCallbackHandler。 大多数情况下，这些回调函数可以直接使用，不用了解它们的内部机制。

为了需要完全控制回调行为，内部 JaasAuthenticationProvider 使用一个 InternalCallbackHandler 封装这个 JaasAuthenticationCallbackHandler。 这个 InternalCallbackHandler 才是实际实现了 JAAS 通常的 CallbackHandler 接口。任何时候 JAASLoginModule 被使用的时候，它传递一个 application context 里配置的 InternalCallbackHandler 列表。 如果这个 LoginModule 需要回调 InternalCallbackHandler，回调会传递封装好的 JaasAuthenticationCallbackHandler。

20.2.2. JAAS AuthorityGranter

JAAS 工作在主体上。 任何"角色"在 JAAS 里都是作为主体表现的。 另一方面 Spring Security 使用 Authentication 对象。每个 Authentication 对象包含单独的主体和多个 GrantedAuthority[]。 为了方便映射不同的概念，Spring Security 的 JAAS 包包含了 AuthorityGranter 接口。

一个 AuthorityGranter 负责检查 JAAS 主体，返回一个 String 的集合，用来表示分配给这个主体的权限。 对于每一个返回的权限字符串，JaasAuthenticationProvider 会创建一个 JaasGrantedAuthority（它实现了 Spring Security 的 GrantedAuthority 接口），包含了 AuthorityGranter 返回的字符串和 AuthorityGranter 传递的 JAAS 主体。 JaasAuthenticationProvider 获得 JAAS 主体，通过首先成功认证用户的证书，使用 JAAS 的 LoginModule，然后调用 LoginContext.getSubject().getPrincipals()，使用返回的每个主体，传递到每个 AuthorityGranter 里，最后定义在 JaasAuthenticationProvider.setAuthorityGranters(List)属性里。

Spring Security 没有包含任何产品型的 AuthorityGranter，因为每个 JAAS 主体都有特殊实现的意义。 但是，这里的单元测试里有一个 TestAuthorityGranter，演示了一个简单的 AuthorityGranter 实现。

Chapter 21. CAS 认证

21.1. 概述

JA-SIG 开发了一个企业级的单点登录系统，叫做 CAS。 与其他项目不同，JA-SIG 的中心认证服务是开源的，广泛使用的，简单理解的，不依赖平台的，而且支持代理能力。Spring Security 完全支持 CAS，提供一个简单的整合方式，把使用 Spring Security 的单应用发布，转换成使用企业级 CAS 服务器的多应用发布安全

你可以从 http://www.ja-sig.org/cas/找到 CAS 的更多信息。 你还需要访问这个网站下载 CAS 服务器文件。

21.2. CAS 是如何工作的

虽然 CAS 网站包含了 CAS 的架构文档，我们这里还是说一下使用 Spring Security 环境的一般性概述，。 Spring Security 3.0 支持 CAS 3。 在写文档的时候，CAS 服务器的版本是 3.3。

你要在公司内部安装 CAS 服务器。 CAS 服务器就是一个 WAR 文件，所以安装服务器没有什么难的。 在 WAR 文件里，你需要自定义登录和其他单点登录展示给用户的页面。

发布 CAS 3.3 的时候，你也需要指定一个 CAS 的 deployerConfigContext.xml 里包含的 AuthenticationHandler。 AuthenticationHandler 有一个简单的方法，返回布尔值，判断给出的证书集合是否有效。 你的 AuthenticationHandler 实现会需要链接到后台认证资源类型里，像是 LDAP 服务器或数据库。 CAS 自己也包含非常多 AuthenticationHandler 帮助实现这些。 在你下载发布服务器 war 文件的时候，它会把用户名和密码匹配的用户成功验证，这对测试很有用。

除了 CAS 服务器，其他关键角色当然是你企业发布的其他安全 web 应用。 这些 web 应用被叫做"services"。 这儿有两种服务：标准服务和代理服务。 代理服务可以代表用户，从其他服务中请求资源。 后面会进行更详细的介绍。

21.3. 配置 CAS 客户端

CAS 的 web 应用端通过 Spring Security 使用起来很简单。 我们假设你已经知道 Spring Security 的基本用法，所以下面都没有涉及这些。 我们会假设使用基于命名空间配置的方法，并且添加了 CAS 需要的 bean。

你需要添加一个 ServiceProperties bean，到你的 application context 里。 这表现你的 CAS 服务：

```
  <bean id="serviceProperties"



class="org.springframework.security.cas.ServiceProperties">
```

```
    <property name="service"

value="https://localhost:8443/cas-sample/j_spring_cas_security_check"/>
    <property name="sendRenew" value="false"/>
  </bean>
```

这里的 service 必须是一个由 CasAuthenticationFilter 监控的 URL。 这个 sendRenew 默认是 false，但如果你的程序特别敏感就应该设置成 true。 这个参数作用是，告诉 CAS 登录服务，一个单点登录没有到达。 否则，用户需要重新输入他们的用户名和密码，来获得访问服务的权限。

下面配置的 bean 就是展开 CAS 认证的过程（假设你使用了命名空间配置）：

```
<security:http entry-point-ref="casEntryPoint">
   ...
   <custom-filter position="FORM_LOGIN_FILTER" ref="myFilter" />
</security:http>

<bean id="casFilter"

class="org.springframework.security.cas.web.CasAuthenticationFilter">
  <property name="authenticationManager"
ref="authenticationManager"/>
</bean>
```

```
<bean id="casEntryPoint"

class="org.springframework.security.cas.web.CasAuthenticationEn
tryPoint">
  <property                                          name="loginUrl"
value="https://localhost:9443/cas/login"/>
  <property name="serviceProperties" ref="serviceProperties"/>
</bean>
```

应该使用 entry-point-ref 选择驱动认证的 CasAuthenticationEntryPoint 类。

CasAuthenticationFilter 的属性与 UsernamePasswordAuthenticationFilter 非常相似（在基于表单登录时用到）。

为了 CAS 的操作，ExceptionTranslationFilter 必须有它的 AuthenticationEntryPoint，这里设置成 CasAuthenticationEntryPoint bean。

CasAuthenticationEntryPoint 必须指向 ServiceProperties bean （上面讨论过了），它为企业 CAS 登录服务器提供 URL。 这是用户浏览器将被重定向的位置。

下一步，你需要添加一个 CasAuthenticationProvider 和它的合作伙伴：

```
  <security:authentication-manager
alias="authenticationManager">
    <security:authentication-provider
ref="casAuthenticationProvider" />
  </security:authentication-manager>
```

```
  <bean id="casAuthenticationProvider"

class="org.springframework.security.cas.authentication.CasAuthe
nticationProvider">
    <property name="userDetailsService" ref="userService"/>
    <property name="serviceProperties" ref="serviceProperties" />
    <property name="ticketValidator">
      <bean
class="org.jasig.cas.client.validation.Cas20ServiceTicketValida
tor">
        <constructor-arg                                    index="0"
value="https://localhost:9443/cas" />
        </bean>
    </property>
    <property                                              name="key"
value="an_id_for_this_auth_provider_only"/>
  </bean>

  <security:user-service id="userService">
    <security:user            name="joe"            password="joe"
authorities="ROLE_USER" />
    ...
```

```
  </security:user-service>
```

一旦通过了 CAS 的认证，CasAuthenticationProvider 使用一个 UserDetailsService 实例，来加载用户的认证信息。 这里我们展示一个简单的基于内存的设置。

如果你翻回头看一下"How CAS Works"那节，这些 beans 都是从名字就可以看懂的。

X.509 认证

22.1. 概述

X.509 证书认证最常见的使用方法是使用 SSL 验证服务器的身份，通常情况是在浏览器使用 SSL。 浏览器使用一个它维护的可信任的证书权限列表，自动检测服务器发出的证书（比如数字签名）。

你也可以使用 SSL 进行"mutual authentication"相互认证；服务器会从客户端请求一个合法的证书，作为 SSL 握手协议的一部分。 服务器将验证客户端，通过检测它被签在一个可接受的权限里的认证。 如果已经提供了一个有效的证书，就可以从程序的 servlet API 里获得。 Spring Security X.509 模块使用过滤器确认证书。 它将证书映射为应用程序的用户，并使用标准的 Spring Security 基础设施读取用户的已授予权限集合。

你应该很熟悉使用证书，在使用 Spring Security 之前为你的 servlet 容器启动客户端认证。 大多数工作都是创建和安装合适的证书和密匙。 比如，如果你使用 tomcat，可以阅读这里的教程 http://tomcat.apache.org/tomcat-6.0-doc/ssl-howto.html。 在你把它用在 Spring Security 里之前，先知道它是怎么工作，是很重要的。

22.2. 把 X.509 认证添加到你的 web 系统中

启用 X.509 客户端认证非常直观。 只需要把<x509/>元素添加到你的 http 安全命名空间配置里。

```
<http>

 ...

    <x509                    subject-principal-regex="CN=(.*?),"

user-service-ref="userService"/>

 ...
```

```
</http>
```

这个元素有两个可选属性：

- subject-principal-regex。这是一个正则表达式，用来从证书主体名称里获得用户名。默认值已经写在上面了。这个用户名会传递给 UserDetailsService 来获得用户的认证信息。
- user-service-ref。这是 X.509 需要用到的一个 UserDetailsService 的 bean 的 id。如果你的 application context 里只定义了一个 bean，就不需要使用它。

subject-principal-regex 应该包含一个单独的组。比如默认的表达式"CN=(.*?)," 匹配普通的名字字段。 所以，如果证书主题名是"CN=Jimi Hendrix, OU=..."，就会得到一个名叫"Jimi Hendrix"的用户。 这个匹配是大小写不敏感的。 所以"emailAddress=(.?),"也会匹配"EMAILADDRESS=jimi@hendrix.org,CN=..."，得到一个"jimi@hendrix.org"用户名。 如果客户端给出一个证书，并成功获得了一个合法用户名，然后在安全环境里应该有一个有效的 Authentication 对象。 如果没有找到证书，或没有找到对应的用户，安全环境会保持为空。 这说明你可以很简单的和其他选项一起使用 X.509 认证，比如基于表单登录。

22.3. 为 tomcat 配置 SSL

在 Spring Security 项目的 samples/certificate 目录下,有几个已经生成好的证书。如果你不想自己去生成，就可以使用们启用 SSL 做测试，。 server.jks 文件包含了服务器证书，私匙和签发证书颁发机构证书。 这里还有一些客户端证书文件，提供给例子程序的用户。 你可以把他们安装到你的浏览器，启动 SSL 客户端认证。

要运行支持 SSL 的 tomcat，把 server.jks 文件放到 tomcat 的 conf 目录下，然后把下面的连接器添加到 server.xml 文件中

```
<Connector  port="8443"  protocol="HTTP/1.1"  SSLEnabled="true"
scheme="https" secure="true"
           clientAuth="true" sslProtocol="TLS"
           keystoreFile="${catalina.home}/conf/server.jks"
           keystoreType="JKS" keystorePass="password"
           truststoreFile="${catalina.home}/conf/server.jks"
           truststoreType="JKS" truststorePass="password"
```

```
/>
```

clientAuth 也可以设置成 *want*，如果你希望客户端没有提供证书的时候 SSL 链接也能成功。 客户端不提供证书的话，就不能访问 Spring Security 的任何安全对象，除非你使用了非 X.509 认证机制，比如表单认证。

替换验证身份

23.1. 概述

AbstractSecurityInterceptor 可以在安全对象回调期间，暂时替换 SecurityContext 和 SecurityContextHolder 里的 Authentication 对象。只有在原始 Authentication 对象被 AuthenticationManager 和 AccessDecisionManager 成功处理之后，才有可能发生这种情况。 如果有需要，RunAsManager 会显示替换的 Authentication 对象，这应该通过 SecurityInterceptorCallback 调用。

通过在安全对象回调过程中临时替换 Authentication 对象，安全调用可以调用其他需要不同认证授权证书的对象。 这也可以为特定的 GrantedAuthority 对象执行内部安全检验。 因为 Spring Security 提供不少帮助类，能够基于 SecurityContextHolder 的内容自动配置远程协议，这些运行身份替换在远程 web 服务调用的时候特别有用。

23.2. 配置

一个 Spring Security 提供的 RunAsManager 接口：：

```
  Authentication buildRunAs(Authentication authentication, Object

object,

    List<ConfigAttributeDefinition> config);

  boolean supports(ConfigAttribute attribute);

  boolean supports(Class clazz);
```

第一个方法返回 Authentication 对象，在方法的调用期间替换以前的 Authentication 对象。 如果方法返回 null，意味着不需要进行替换。 第二个方法用在 AbstractSecurityInterceptor 中，作为它启动时校验配置属性的一部分。

supports(Class)方法会被安全拦截器的实现调用,确保配置的RunAsManager支持安全拦截器即将执行的安全对象类型。

Spring Security 提供了一个 RunAsManager 的具体实现。 如果任何一个 ConfigAttribute 是以 RUN_AS_开头的，RunAsManagerImpl 类返回一个替换的 RunAsUserToken 。 如果找到了任何这样的 ConfigAttribute，替换的 RunAsUserToken 会通过一个新的 GrantedAuthorityImpl，为每一个 RUN_AS_ ConfigAttribute 包含同样的主体，证书，赋予的权限，就像原来的 Authentication 对象一样。 每个新 GrantedAuthorityImpl 会以 ROLE_开头，对应 RUN_AS ConfigAttribute。 比如，一个替代 RunAsUserToken，对于 RUN_AS_SERVER 的结果是包含一个 ROLE_RUN_AS_SERVER 赋予的权限。

替代的 RunAsUserToken 就像其他 Authentication 对象一样。 它可能需要通过代理合适的 AuthenticationProvider 被 AuthenticationManager 验证。 这个 RunAsImplAuthenticationProvider 执行这样的认证，它直接获得任何一个有效的 RunAsUserToken。

为了保证恶意代码不会创建一个 RunAsUserToken，由 RunAsImplAuthenticationProvider 保障获得一个 key 的散列值被保存在所有生成的标记里。 RunAsManagerImpl 和 RunAsImplAuthenticationProvider 在 bean 上下文里，创建使用同样的 key：

```
<bean id="runAsManager"

class="org.springframework.security.access.intercept.RunAsManag
erImpl">
  <property name="key" value="my_run_as_password"/>
</bean>

<bean id="runAsAuthenticationProvider"

class="org.springframework.security.access.intercept.RunAsImplA
uthenticationProvider">
```

```
  <property name="key" value="my_run_as_password"/>

</bean>
```

通过使用相同的 key，每个 RunAsUserToken 可以被它验证，并使用对应的 RunAsManagerImpl 创建。 出于安全原因，这个 RunAsUserToken 创建后就不能改变。

安全数据库表结构

可以为框架采用不同的数据库结构，这个附录为所有功能提供了一种参考形式。 你只要为需要的功能部分提供对应的表结构。

这些 DDL 语句都是对应于 HSQLDB 数据库的。 你可以把它们当作一个指南，参照它，在你使用的数据库中定义表结构。

A.1. User 表

UserDetailsService 的标准 JDBC 实现(JdbcDaoImpl)，需要从这些表里读取用户的密码，帐号信息（可用或禁用）和权限（角色）列表。

```
  create table users(

      username varchar_ignorecase(50) not null primary key,

      password varchar_ignorecase(50) not null,

      enabled boolean not null);


  create table authorities (

      username varchar_ignorecase(50) not null,

      authority varchar_ignorecase(50) not null,

      constraint fk_authorities_users foreign key(username)
references users(username));

      create unique index ix_auth_username on authorities
(username,authority);
```

A.1.1. 组权限

Spring Security 2.0 在 JdbcDaoImpl 中支持了权限分组。如果启用了权限分组功能，对应的数据库结果如下所示：

```
create table groups (
  id bigint generated by default as identity(start with 0) primary
key,
  group_name varchar_ignorecase(50) not null);

create table group_authorities (
  group_id bigint not null,
  authority varchar(50) not null,
  constraint  fk_group_authorities_group  foreign  key(group_id)
references groups(id));

create table group_members (
  id bigint generated by default as identity(start with 0) primary
key,
  username varchar(50) not null,
  group_id bigint not null,
  constraint   fk_group_members_group   foreign   key(group_id)
references groups(id));
```

A.2. 持久登陆（Remember-Me）表

这个表用来保存安全性更高的持久登陆 remember-me 实现所需要的数据。如果你直接或通过命名空间使用了 JdbcTokenRepositoryImpl，你就会需要这些表结构。

```
create table persistent_logins (

  username varchar(64) not null,

  series varchar(64) primary key,

  token varchar(64) not null,

  last_used timestamp not null);
```

A.3. ACL 表

这里有四个表被 Spring Security 用来实现 ACL。

- acl_sid 保存被 ACL 系统分配的安全标示符。 它们可能是唯一的实体或可能分配给多个实体的权限。
- acl_class 定义 ACL 可以处理的实体类型。class 列保存了对象的 Java 类名。
- acl_object_identity 保存值得那个领域对象昂的对象标示定义。
- acl_entry 保存 ACL 权限，分配给一个特定的对象标示和安全标示。

假设数据库会自动生成主键作为每个标示。JdbcMutableAclService 必须可以获得这些，当创建了一个新的 acl_sid 或 acl_class 表中的数据。 它有两个属性可以定义需要的 SQL 来获得这些数据， classIdentityQuery 和 sidIdentityQuery。 这两个属性的默认值是 call identity()。

A.3.1. Hypersonic SQL

默认的表结构可以工作在内嵌的 HSQLDB 中，可以在框架内用作单元测试。

```
create table acl_sid (

  id bigint generated by default as identity(start with 100) not

null primary key,

  principal boolean not null,

  sid varchar_ignorecase(100) not null,

  constraint unique_uk_1 unique(sid,principal) );
```

```
create table acl_class (
  id bigint generated by default as identity(start with 100) not
null primary key,
  class varchar_ignorecase(100) not null,
  constraint unique_uk_2 unique(class) );


create table acl_object_identity (
  id bigint generated by default as identity(start with 100) not
null primary key,
  object_id_class bigint not null,
  object_id_identity bigint not null,
  parent_object bigint,
  owner_sid bigint not null,
  entries_inheriting boolean not null,
  constraint                                          unique_uk_3
unique(object_id_class,object_id_identity),
  constraint foreign_fk_1 foreign key(parent_object) references
acl_object_identity(id),
  constraint foreign_fk_2 foreign key(object_id_class) references
acl_class(id),
  constraint  foreign_fk_3  foreign  key(owner_sid)  references
acl_sid(id) );
```

```
create table acl_entry (
  id bigint generated by default as identity(start with 100) not
null primary key,
  acl_object_identity bigint not null,ace_order int not null,sid
bigint not null,
  mask integer not null,granting boolean not null,audit_success
boolean not null,
  audit_failure boolean not null,
  constraint unique_uk_4 unique(acl_object_identity,ace_order),
  constraint foreign_fk_4 foreign key(acl_object_identity)
      references acl_object_identity(id),
  constraint    foreign_fk_5    foreign    key(sid)    references
acl_sid(id) );
```

A.3.1.1. PostgreSQL

```
create table acl_sid(
  id bigserial not null primary key,
  principal boolean not null,
  sid varchar(100) not null,
  constraint unique_uk_1 unique(sid,principal));
```

```
create table acl_class(
  id bigserial not null primary key,
  class varchar(100) not null,
  constraint unique_uk_2 unique(class));


create table acl_object_identity(
  id bigserial primary key,
  object_id_class bigint not null,
  object_id_identity bigint not null,
  parent_object bigint,
  owner_sid bigint,
  entries_inheriting boolean not null,
  constraint                                    unique_uk_3
unique(object_id_class,object_id_identity),
  constraint foreign_fk_1 foreign key(parent_object) references
acl_object_identity(id),
  constraint foreign_fk_2 foreign key(object_id_class) references
acl_class(id),
  constraint  foreign_fk_3  foreign  key(owner_sid)  references
acl_sid(id));


create table acl_entry(
```

```
  id bigserial primary key,

  acl_object_identity bigint not null,

  ace_order int not null,

  sid bigint not null,

  mask integer not null,

  granting boolean not null,

  audit_success boolean not null,

  audit_failure boolean not null,

  constraint unique_uk_4 unique(acl_object_identity,ace_order),

  constraint foreign_fk_4 foreign key(acl_object_identity)

      references acl_object_identity(id),

  constraint    foreign_fk_5    foreign    key(sid)    references

acl_sid(id));
```

你需要把 classIdentityQuery 和 sidIdentityQuery 两个 JdbcMutableAclService 的属性设置成下面的值：

- `select currval(pg_get_serial_sequence('acl_class', 'id'))`
- `select currval(pg_get_serial_sequence('acl_sid', 'id'))`

安全命名空间

这个附录提供了对安全命名空间中可用元素的参考信息，也介绍了它们创建的 bean 的信息（对单独类的知识和它们是如何在一起工作的 - 你可以在工程的 Javadoc 和这个文档的其他部分找到更多信息）。 如果你以前没有使用过命名空间，请阅读命名空间配置部分的介绍章节，这部分的信息是作为那些章节的一个补充资料的。 我们推荐你在编辑基于 schema 的配置时使用一个高质量的 XML 编辑器，这会为你提供上下文环境相关的信息，哪些元素和属性是可用的，注释会解释它们的用途。 命名空间是使用 RELAX NG 兼容格式编写的，随后转换为 XSD Schema。 如果你对这种格式很熟悉，很可能希望直接查看 schema 文件。

B.1. Web 应用安全 - <http>元素

<http>元素为你的应用程序的 web 层封装了安全性配置。 它创建了一个名为"springSecurityFilterChain"的 FilterChainProxy bean，这个 bean 维护了一系列的建立了 web 安全配置的安全过滤器。[14]。 一些核心的过滤器总是要被创建的，其他的将根据子元素的配置添加到过滤器队列中。 标准过滤器的位置都是固定的（参考命名空间配置中的过滤器顺序表格），这避免了之前版本中的一个常见问题，那时候用户必须自己在 FilterChainProxy bean 中配置过滤器链。当然如果你需要对配置进行完全控制，依然可以这样做。

所有需要引用 AuthenticationManager 的过滤器，都会自动注入命名空间配置创建的内部实例（查看介绍章节获得 AuthenticationManager 的更多信息）。

<http>命名空间块会创建一个 HttpSessionContextIntegrationFilter，一个 ExceptionTranslationFilter 和一个 FilterSecurityInterceptor。 它们是固定的，不能使用其他可选方式替换。

B.1.1. <http>属性

<http>元素的属性控制核心过滤器的一些属性。

B.1.1.1. servlet-api-provision

支持一些版本的 HttpServletRequest 提供的安全方法，必须 isUserInRole()和 getPrincipal()，通过向堆栈中添加一个 SecurityContextHolderAwareRequestFilter bean 来实现。 默认是"true"。

B.1.1.2. path-type

控制拦截 URL 的时候，使用 ant 路径（默认）或是使用正则表达式。 实际中，它向 FilterChainProxy 中设置了特定的 UrlMatcher。

B.1.1.3. lowercase-comparisons

是否在对 URL 进行匹配前，先将 URL 转换成小写。 如果没有定义，默认是"true"。

B.1.1.4. realm

为基础认证设置 realm 名称（如果启用）。 对应 BasicAuthenticationEntryPoint 中的 realmName 属性。

B.1.1.5. entry-point-ref

正常情况下 AuthenticationEntryPoint 将根据配置的认证机制进行设置。这个属性让这个行为使用自定义的 AuthenticationEntryPoint bean 进行覆盖，它会启动认证流程。

B.1.1.6. access-decision-manager-ref

可选的属性，指定 AccessDecisionManager 实现的 ID，这应该被认证的 HTTP 请求使用。 默认情况下一个 AffirmativeBased 实现会被 RoleVoter 和 AuthenticatedVoter 使用。

B.1.1.7. access-denied-page

这个属性已经被 access-denied-handler 子元素取代了。

B.1.1.8. once-per-request

对应 FilterSecurityInterceptor 的 observeOncePerRequest 属性。 默认是 "true"。

B.1.1.9. create-session

控制创建一个 HTTP 会话的紧急程度。 如果不设置，默认是"ifRequired"。 其他选项是 "always" 和 "never" 。 这个属性的设置影响 HttpSessionContextIntegrationFilter 的 allowSessionCreation 和 forceEagerSessionCreation 属性。 除非把属性设置为 "never"allowSessionCreation 会一直为 "true" 。 除非把属性设置为 "always"forceEagerSessionCreation 会一直为"false"。 所以默认的配置允许会话的创建，但不会强制。 如果启用同步会话控制，当 forceEagerSessionCreation 被设置为 "true"，不管这里设置的什么都会抛出异常。 使用 "never" 会在 HttpSessionContextIntegrationFilter 初始化的过程中导致异常。

B.1.2. <access-denied-handler>

这个元素允许你为默认的 AccessDeniedHandler 设置 errorPage 属性， 它会被 ExceptionTranslationFilter 用到，（使用 error-page 属性，或通过 ref 属性 提供你自己的实现。参考 ExceptionTranslateFilter 获得更多信息。）

B.1.3. <intercept-url>元素

这个元素用来定义 URL 模式集合，应用对什么感兴趣并配置它们应该如何处理。 它用来构建被 FilterSecurityInterceptor 使用的 FilterInvocationDefinitionSource，也可以从过滤器链中排除特定的模式（通过使用 filters="none"属性）。 它也负责配置 ChannelAuthenticationFilter，如果特定的 URL 需要通过 HTTPS 访问，比如。 当匹配时指定的模式对应了进入的请求，匹配过程就会完成，按照声明的元素顺序。 所以最希望被匹配的模式应该放在上面，最常用的模式应该放在最后。

B.1.3.1. pattern

这个模式定义了 URL 路径。 内容依赖于 http 元素中的 path-type 属性，它的默认值是 ant 路径语法。

B.1.3.2. method

HTTP Method 会被用来结合模式来匹配进入的请求。 如果忽略，所有的 Method 都会匹配。 如果一个相同的模式指定了，使用 method 和没有使用 method 两种方式，指定了 method 的匹配将被优先使用。

B.1.3.3. access

列出会被存储在 FilterInvocationDefinitionSource 中的访问属性，为定义的模式/Method 结合的形式。 这应该是由分号分隔的安全配置属性队列（比如角色名称）。

B.1.3.4. requires-channel

可以是 http 或 https，这是根据一个特定的 URL 模式是否应该通过 HTTP 或 HTTPS 访问。 如果没有偏好还可以选择 any。 如果这个属性已经出现在任何一个<intercept-url>上，一个 ChannelAuthenticationFilter 会添加到过滤器堆栈里，它的附加依赖也会添加到 application context 中。 查看信道安全获得使用传统 bean 的例子配置。

如果添加了一个<port-mappings>配置，它会被 SecureChannelProcessor 和 InsecureChannelProcessor 使用来决定在重定向到 HTTP/HTTPS 的时候使用什么端口。

B.1.3.5. filters

可以只使用"none"作为属性值。 它会导致任何匹配的请求完全被 Spring Security 忽略。所有<http> 中的其他配置，影响在请求上，在这个过程中都无法访问安全上下文。在这个请求过程中访问 被保护的方法都会失败。

B.1.4. <port-mappings>元素

默认情况下，PortMapperImpl 的实例会添加到配置中，在重定向到安全和不安全的 URL 时使用到。 这个元素可以选择用来覆盖类定义的默认映射。 每个子<port-mapping>元素都定义了一对 HTTP:HTTPS 端口。 默认的映射是 80:443 和 8080:8443。 一个覆盖这些的例子可以在命名空间介绍中看到。

B.1.5. <form-login>元素

用来把一个 UsernamePasswordAuthenticationFilter 添加到过滤器堆栈中，把一个 LoginUrlAuthenticationEntryPoint 添加到 application context 中来提供需要的认证。 这将永远凌驾于其他命名空间创建的切入点。 如果没有提供属性，一个登录页面会自动创建在"/spring-security-login"这个 URL 下[15]。这个行为可以使用下面的属性自定义。

B.1.5.1. login-page

这个 URL 应该用来生成登录页面。 对应 LoginUrlAuthenticationEntryPoint 的 loginFormUrl 属性。 默认是"/spring-security-login"。

B.1.5.2. login-processing-url

对应 UsernamePasswordAuthenticationFilter 的 filterProcessesUrl 属性。 默认是"/j_spring_security_check"

B.1.5.3. default-target-url

对应 UsernamePasswordAuthenticationFilter 的 defaultTargetUrl 属性。 如果没有设置，默认值是"/"（应用的根路径）。 一个用户会在登录之后到达这个 URL，在他们没有在登录之前尝试访问一个安全资源，否则他们就会被转向到原来请求的 URL。

B.1.5.4. always-use-default-target

如果设置成"true"，用户会一直转发到 default-target-url 指定的位置，无论他们在登录页面之前访问的什么位置。 对应 UsernamePasswordAuthenticationFilter 的 alwaysUseDefaultTargetUrl 属性。

B.1.5.5. authentication-failure-url

对应 UsernamePasswordAuthenticationFilter 的 authenticationFailureUrl 属性。 定义了在登录失败时浏览器会重定向的 URL。 默认是"/spring_security_login?login_error"，它会自动被登陆页面生成器处理，并使用一个错误信息重新渲染登录页面。

B.1.5.6. authentication-success-handler-ref

这可以用来替换 default-target-url 和 always-use-default-target，你可以完全控制 成功认证之后的导航流向。这个值应该是 application context 中的 AuthenticationSuccessHandlerbean 的名称。

B.1.5.7. authentication-failure-handler-ref

可以用来替换 authentication-failure-url，你可以完全控制认证失败之后的导航流向。这个值应该是 application context 中的 AuthenticationFailureHandlerbean 的名称。

B.1.6. <http-basic>元素

向配置中添加一个 BasicAuthenticationFilter 和 BasicAuthenticationEntryPoint。 后一个只有在基于表单登录没有启用的时候才会被用作配置入口。

B.1.7. <remember-me>元素

向堆栈中添加 RememberMeAuthenticationFilter。 这会在配置了一个 TokenBasedRememberMeServices，或一个 PersistentTokenBasedRememberMeServices，或一个用户自定义的实现了 RememberMeServices 的配置设置后启用。

B.1.7.1. data-source-ref

如果设置了这个，PersistentTokenBasedRememberMeServices 会被使用到，并配置上一个 JdbcTokenRepositoryImpl 实例。

B.1.7.2. token-repository-ref

配置一个 PersistentTokenBasedRememberMeServices 但是允许使用一个自定义的 PersistentTokenRepository bean。

B.1.7.3. services-ref

允许对将要用在过滤器里的 RememberMeServices 的实现提供完全控制。 这个值将是 application context 里的一个实现了这个接口的 bean 的 id。

B.1.7.4. token-repository-ref

配置一个 PersistentTokenBasedRememberMeServices 但是允许使用一个自定义的 PersistentTokenRepository bean。

B.1.7.5. key 属性

对应 AbstractRememberMeServices 的"key"属性。 应该设置一个唯一的值来确定 remember-me 的 cookies 只对唯一的应用有效。[16]。

B.1.7.6. token-validity-seconds

对应 AbstractRememberMeServices 的 tokenValiditySeconds 属性。 指定 remember-me cookie 生效的秒数周期。 默认它会在 14 日内生效。

B.1.7.7. user-service-ref

remember-me 服务实现要求可以访问 UserDetailsService，所以在 application context 中必须有一个定义。 如果只定义了一个，它会被选中，并被命名空间配置自动使用。 如果这里有多个实例，你可以使用这个树形指定一个 bean 的 id。

B.1.8. <session-management> 元素

会话管理相关的功能由额外的 过滤器栈中的 SessionManagementFilter 实现。

B.1.8.1. session-fixation-protection

分析一个已存在的绘画是否应该被销毁，当一个用户认证通过，并启动了一个新会话。如果设置为"none"，则不会出现任何改变。"newSession"会创建一个新的空会话。"migrateSession"会创建一个新会话，并把之前会话中的属性都复制到新会话中。 默认是"migrateSession"。

如果启用了会话伪造防御， SessionManagementFilter 会使用一个 匹配的 DefaultSessionAuthenticationStrategy。 参考这个类的 javadoc 获得更多细节。

B.1.9. <concurrent-control>元素

添加对同步会话控制的支持，允许限制一个用户可以拥有的活动会话的数量。 会创建一个 ConcurrentSessionFilter ， 连同一个 ConcurrentSessionControllerStrategy 和 SessionManagementFilter 的实例。 如果已经声明了 form-login 元素，策略对象也会注入到创建的 验证过滤器中。一个 SessionRegistry 的实例（ SessionRegistryImpl 的实例，除非用户希望使用自定义 bean） 会被创建，交给策略使用。

B.1.9.1. max-sessions 属性

对应 ConcurrentSessionControllerImpl 的 maximumSessions 属性。

B.1.9.2. expired-url 属性

如果一个用户尝试使用一个已经过期"expired"的会话，同步会话控制器会重定性到的 URL。因为用户超过了允许的会话数量，但是又在其他地方登录了系统。 除非设置 exception-if-maximum-exceeded，其他时候都应该设置这个属性。 如果没有设置值，一个过期信息会直接写到响应中。

B.1.9.3. error-if-maximum-exceeded 属性

如果设置成"true"，一个 SessionAuthenticationException 会被抛出， 当一个用户尝试超过最大会话允许数量。 默认行为是让原始会话过期。

B.1.9.4. session-registry-alias 和 session-registry-ref 属性

用户可以提供他们自己的 SessionRegistry 实现,使用 session-registry-ref 属性。其他同步会话控制 bean 就可以使用它。

它也可以用来使用内部会话注册的引用，用在你自己的 bean 或一个管理接口里。 你可以使用 session-registry-alias 属性暴露内部 bean,给它一个名字你可以在你的配置的任意地方都使用它。

B.1.10. <anonymous>元素

添加一个 AnonymousAuthenticationFilter 和 AnonymousAuthenticationProvider 到堆栈里。 如果你使用 IS_AUTHENTICATED_ANONYMOUSLY 属性，就是必要的。

B.1.11. <x509>元素

添加 X.509 认证的支持。 一个 X509AuthenticationFilter 会被添加到堆栈中，会创建一个 PreAuthenticatedProcessingFilterEntryPoint。 后一个只有的其他认证机制都没有使用的情况下才会用到(它唯一的功能是返回一个 HTTP 403 错误代号)。

一个 PreAuthenticatedAuthenticationProvider 也会被创建,并代理用户权限读取到一个 UserDetailsService 里。

B.1.11.1. subject-principal-regex 属性

定义一个正则表达式，会从证书中取出用户名（与 UserDetailsService 一起使用）。

B.1.11.2. user-service-ref 属性

允许一个特定的 UserDetailsService，与 X.509 一起使用，当多个实例被配置的时候。 如果没有设置，会尝试自动定位一个合适的实例并使用它。

B.1.12. <openid-login>元素

与<form-login>类似，拥有相同的属性。 login-processing-url 的默认值是"/j_spring_openid_security_check"。 一个 OpenIDAuthenticationFilter 和 OpenIDAuthenticationProvider 会被注册上。后者需要一个 UserDetailsService 的引用。 它也可以使用 id 指定，使用 user-service-ref 属性，或者在 application context 中自动定位。

B.1.13. <logout>元素

添加一个 LogoutFilter 到过滤器堆栈中。 它和 SecurityContextLogoutHandler 一起配置。

B.1.13.1. logout-url 属性

这个 URL 会触发注销操作（比如，会被过滤器处理）。 默认是"/j_spring_security_logout"。

B.1.13.2. logout-success-url 属性

用户在注销后转向的 URL。 默认是"/"。

B.1.13.3. invalidate-session 属性

对应 SecurityContextLogoutHandler 的 invalidateHttpSession 属性。 默认是"true"，这样注销的时候会销毁会话。

B.1.14. <custom-filter>元素

这个元素用来将一个过滤器添加到过滤器链中。它不会创建额外的 bean， 但是它用来选择选择一个 javax.servlet.Filter 类型的 bean， 这个 bean 已经定义在 application context 中，把它添加到 Spring Security 维护的过滤器链的特定位置。全部信息可以在命名空间章节找到。

B.2. 认证服务

在 Spring Security 3.0 之前，一个 AuthenticationManager 会自动注册， 现在你必须使用<authentication-manager>元素注册一个 bean。 这个 bean 是 Spring Security 的 ProviderManager 类的一个实例，它需要配置一个或多个 AuthenticationProvider 的实例。 这里可以使用命名空间支持的语法元素，也可以使用标准的 bean 定义，使用 custom-authentication-provider 元素来添加列表。

B.2.1. <authentication-manager>元素

每个 Spring Security 应用，只要使用了命名空间，就必须在什么地方包含对应的元素。它负责注册 AuthenticationManager，为应用各提供验证服务。 它也允许你定义一个别名，为内部实例，在你的配置中来使用。 这些都写在命名空间介绍中。 所有元素，创建了 AuthenticationProvider 实例， 应该是这个元素的子元素。

B.2.1.1. <authentication-provider>元素

这个元素基本是配置 DaoAuthenticationProvider 的简化形式。DaoAuthenticationProvider 读取用户信息，从 UserDetailsService 中，比较用户名/密码，来进行用户登录。 UserDetailsService 实例可以被定义，无论是命名空间中的 (jdbc-user-service 或使用 user-service-ref 属性来引用一个 bean，定义在 application context 中)。 你可以在命名空间介绍中找到。。

B.2.1.2. 使用 <authentication-provider> 来引用一个 AuthenticationProvider Bean

如果你已经创建了自己的 AuthenticationProvider 实现，（或希望配置 Spring Security 提供的一个实现，因为什么原因使用传统配置方式， 你可以使用下面的语法，来把它添加到内部 ProviderManager 列表中：）

```
  <security:authentication-manager>

    <security:authentication-provider

ref="myAuthenticationProvider" />

  </security:authentication-manager>

  <bean                                id="myAuthenticationProvider"

class="com.something.MyAuthenticationProvider"/>
```

B.3. 方法安全

B.3.1. <global-method-security>元素

这个元素是为 Spring Security 中的 bean 提供安全方法支持的最基本元素。 方法可以通过使用注解来保护（在接口或类级别进行定义）或者作为子元素的切点集合，使用 AspectJ 语法。

方法安全使用与 web 安全相同的 AccessDecisionManager 配置，但是可以使用 Section B.1.1.6, "access-decision-manager-ref"中的解释进行覆盖，使用相同的属性。

B.3.1.1. secured-annotations 和 jsr250-annotations 属性

把这些设置为"true"会分别启用对 Spring Security 自己的@Secured 注解和 JSR-250 注解的支持， 默认情况下它们两个都是禁用的。 JSR-250 注解的应用还需要向 AccessDecisionManager 添加一个 Jsr250Voter，这样你需要确定你需要做这个，如果你使用一个自定义的实现，然后想要使用这些注解。

B.3.1.2. 安全方法使用<protect-pointcut>

除了在单独的方法或类的基础上使用@Secured 定义安全属性，你可以定义交叉安全实体，覆盖你服务层中所有的方法和接口，使用<protect-pointcut>元素。 它有两个属性：

- expression - 切点表达式
- access - 提供的安全属性

你可以在命名空间介绍中找到一个例子。

B.3.1.3. <after-invocation-provider> 元素

这个元素可以用来装饰一个 AfterInvocationProvider 来使用安全拦截器， 通过<global-method-security>命名空间。 你可以定义 0，或者多个类，在 global-method-security 元素中， 每个都使用一个 ref 属性引用到一个 AfterInvocationProvider 实例， 在你的 application context 中

B.3.2. LDAP 命名空间选项

LDAP 已经在它自己的章节中讨论过一些细节了。 我们将在这里进行一些扩展，解释命名空间中的选项如何对应 Spring 的 bean。 LDAP 实现使用 Spring LDAP 扩展，所以最好熟悉一下工程的 API。

B.3.2.1. 使用<ldap-server>元素定义 LDAP 服务器

这个元素使用其他 LDAP bean 来建立一个 Spring LDAP ContextSource，定义 LDAP 服务器的位置和其他信息（比如用户名和密码，如果它不允许匿名访问）来连接它。 它也可以用来创建一个测试用的嵌入式服务器。 所有选项的语法细节信息都在 LDAP 章节中。 实际上的 ContextSource 实现是 DefaultSpringSecurityContextSource，它扩展了 Spring LDAP 的 LdapContextSource 类。 manager-dn 和 manager-password 属性分别对应后者的 userDn 和 password 属性。

如果你只在你的 application context 中定义了一个服务器，其他 LDAP 命名空间定义的 bean 会自动使用它。 否则，你可以为这个元素定义一个"id"属性，然后在其他命

名空间 bean 中使用 server-ref 属性引用它。这其实是 ContextSource 实例的 bean 的 id，如你你想要在其他传统 Spring bean 中使用它。

B.3.2.2. <ldap-provider>元素

这个元素是为了创建 LdapAuthenticationProvider 实例的简写。默认情况下它会和 BindAuthenticator 实例和一个 DefaultAuthoritiesPopulator 一起配置。

B.3.2.2.1. user-dn-pattern 属性

如果你的用户在目录中的一个固定的位置（比如，你可以不需要进行目录查询，就从用户名直接找到一个 DN），你可以使用这个属性来直接映射 DN。 它直接对应 AbstractLdapAuthenticator 的 userDnPatterns 属性。

B.3.2.2.2. user-search-base 和 user-search-filter 属性

如果你需要执行查询，来定义目录中的用户，然后你可以设置这些属性来控制查询。BindAuthenticator 会被配置在 FilterBasedLdapUserSearch 中，属性值直接对应 bean 构造方法的前两个参数。 如果这些属性没有设置，也没有提供可选的 user-dn-pattern，就会使用默认的查询值 user-search-filter="(uid={0})"和 user-search-base=""。

B.3.2.2.3. group-search-filter ， group-search-base, group-role-attribute 和 role-prefix 属性

group-search-base 的值对应 DefaultAuthoritiesPopulator 的构造方法参数 groupSearchBase ， 默 认 是 "ou=groups" 。 默 认 过 滤 器 值 是 "(uniqueMember={0})" ， 这 假 设 入 口 是 "groupOfUniqueNames" 类 型 。 group-role-attribute 对应 groupRoleAttribute 属性,默认是"cn"。role-prefix 对应于 rolePrefix，默认为"ROLE_"。

B.3.2.2.4. <password-compare>元素

它是作为<ldap-provider>的子元素，切换认证策略从 BindAuthenticator 到 PasswordComparisonAuthenticator。 这可以选择性提供 hash 属性或者使用 <password-encoder>子元素来对密码进行编码，在提交到目录进行比较之前。

B.3.2.3. <ldap-user-service>元素

这 个 元 素 配 置 一 个 LDAP UserDetailsService 。 这 个 类 使 用 LdapUserDetailsService ， 这 是 FilterBasedLdapUserSearch 和

名空间 bean 中使用 server-ref 属性引用它。这其实是 ContextSource 实例的 bean 的 id，如你你想要在其他传统 Spring bean 中使用它。

B.3.2.2. <ldap-provider>元素

这个元素是为了创建 LdapAuthenticationProvider 实例的简写。默认情况下它会和 BindAuthenticator 实例和一个 DefaultAuthoritiesPopulator 一起配置。

B.3.2.2.1. user-dn-pattern 属性

如果你的用户在目录中的一个固定的位置（比如，你可以不需要进行目录查询，就从用户名直接找到一个 DN），你可以使用这个属性来直接映射 DN。 它直接对应 AbstractLdapAuthenticator 的 userDnPatterns 属性。

B.3.2.2.2. user-search-base 和 user-search-filter 属性

如果你需要执行查询，来定义目录中的用户，然后你可以设置这些属性来控制查询。BindAuthenticator 会被配置在 FilterBasedLdapUserSearch 中，属性值直接对应 bean 构造方法的前两个参数。 如果这些属性没有设置，也没有提供可选的 user-dn-pattern，就会使用默认的查询值 user-search-filter="(uid={0})"和 user-search-base=""。

B.3.2.2.3. group-search-filter ， group-search-base, group-role-attribute 和 role-prefix 属性

group-search-base 的值对应 DefaultAuthoritiesPopulator 的构造方法参数 groupSearchBase ， 默 认 是 "ou=groups" 。 默 认 过 滤 器 值 是 "(uniqueMember={0})" ， 这 假 设 入 口 是 "groupOfUniqueNames" 类 型 。 group-role-attribute 对应 groupRoleAttribute 属性,默认是"cn"。role-prefix 对应于 rolePrefix，默认为"ROLE_"。

B.3.2.2.4. <password-compare>元素

它是作为<ldap-provider>的子元素，切换认证策略从 BindAuthenticator 到 PasswordComparisonAuthenticator。 这可以选择性提供 hash 属性或者使用 <password-encoder>子元素来对密码进行编码，在提交到目录进行比较之前。

B.3.2.3. <ldap-user-service>元素

这 个 元 素 配 置 一 个 LDAP UserDetailsService 。 这 个 类 使 用 LdapUserDetailsService ， 这 是 FilterBasedLdapUserSearch 和