Spring Security

参考文档

Ben Alex, Luke Taylor

2.0.x

@author Leo.Nature

序言

Spring Security 为基于 J2EE 的企业应用软件提供了一套全面的安全解决方案。正如你在本手册中看到的那样，我们尝试为您提供一套好用，高可配置的安全系统。

安全问题是一个不断变化的目标，更重要的是寻求一种全面的，系统化的解决方案。 在安全领域我们建议你采取“分层安全”，这样让每一层确保本身尽可能的安全，并为其他层提供额外的安全保障。 每层自身越是“紧密”，你的程序就会越鲁棒越安全。 在底层，你需要处理传输安全和系统认证，减少“中间人攻击”（man-in-the-middle attacks）。 接下来，我们通常会使用防火墙，结合 VPN 或 IP 安全来确保只有获得授权的系统才能尝试连接。 在企业环境中，你可能会部署一个 DMZ（demilitarized zone，隔离区），将面向公众的服务器与后端数据库，应用服务器隔离开。 在以非授权用户运行进程和文件系统安全最大化上，你的操作系统也将扮演一个关键的角色。 操作系统通常配置了自己的防火墙。 然后你要防止针对系统的拒绝服务和暴力攻击。 入侵检测系统在检测和应对攻击的时候尤其有用。这些系统可以实时屏蔽恶意 TCP/IP 地址。 在更高层上，你需要配置 Java 虚拟机，将授予不同 java 类型权限最小化，然后，你的应用程序要添加针对自身特定问题域的安全配。Spring Security 使后者 - 应用程序安全变得更容易。

当然，你需要妥善处理上面提到的每个安全层，以及包含于每个层的管理因素。 这些管理因素具体包括：安全公告检测，补丁，人工诊断，审计，变更管理，工程管理系统，数据备份，灾难回复，性能评测，负载检测，集中日志，应急反应程序等等。

Spring Security 关注的重点是在企业应用安全层为您提供服务，你将发现业务问题领域存在着各式各样的需求。银行系统跟电子商务应用就有很大的不同。电子商务系统与企业销售自动化工具又有很大不同。这些客户化需求让应用安全显得有趣，富有挑战性而且物有所值。

请阅读 Part I, “入门”部分，以它作为开始。它向你介绍了整个框架和以命名空间为基础系统配置方式，让你可以很快启动并运行系统。要是想更多的了解 Spring Security 是如何工作和一些你可能需要用到的类，你应该阅读 Part II, “总体结构”部分。本指南的其余部分使用了较传统的参考文档方式，请按照自己的需要选择阅读的部分。 我们也推荐你阅读尽可能多的在应用安全中可能出现的一般问题。 Spring Security 也不是万能的，它不可能解决所有问题。 重要的一点，应用程序应该从一开始就为安全做好设计。 企图改造它也不是一个好主意。 特别的，如果你在制作一个 web 应用，你应该知道许多潜在的脆弱性，比如跨域脚本，伪造请求和会话劫持，这些都是你在一开始就应该考虑到的。 OWASP 网站（http://www.owasp.org/）维护了一个 web 应用脆弱性前十名的名单，还有很多有用的参考信息。

我们希望你觉得这是一篇很有用的参考指南，并欢迎您提供反馈意见和建议。

最后，欢迎加入 Spring Security 社区。

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 2.0，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。

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，完全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。 第一个被称为“认证”，是为用户建立一个他所声明的主体。 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。“授权”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
- 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)
- Container integration with JBoss, Jetty, Resin and Tomcat (这样，你可以继续使用容器管理认证，如果想的话)
- 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。 或者，你希望确保发起请求的是一个真正的人，而不是机器人或其他自动化程序。 这对保护找回密码不被暴力攻击特别有帮助，或者让别人更难复制你程序中的关键内容。 为了帮助你实现这些目标，Spring Security 支持自动“通道安全”，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，那时候叫“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/
```

你可以获得特定版本的源代码 http://acegisecurity.svn.sourceforge.net/svnroot/acegisecurity/spring-security/tags/.

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-2.0.xsd
                    http://www.springframework.org/schema/security
http://www.springframework.org/schema/security/spring-security-2.0.2.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-2.0.xsd
                        http://www.springframework.org/schema/security
http://www.springframework.org/schema/security/spring-security-2.0.2.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。一个默认的实例会通过命名空间注册到内部。
- *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残酷。 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.DelegatingFilterProxy</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。

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。

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-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>
```

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 DaoAuthenticationProviderbean，<user-service>元素创建了一个 InMemoryDaoImpl。 一个 ProviderManager bean 通常是由命名空间过程系统创建的，DaoAuthenticationProvider自动注册到它上面。 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。

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。

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>
      <intercept-url pattern="/**" access="ROLE_USER" />
      <form-login />
      <anonymous />
      <http-basic />
      <logout />
      <remember-me />
    </http>
```

这些元素分别与 form-login，匿名认证，基本认证，注销处理和 remember-me 对应。 他们拥有各自的属性，来改变他们的具体行为。

auto-config需要一个 UserDetailsService

使用 auto-config 的时候如果没配置 UserDetailsService就会出现错误（比如，如果你使用了 LDAP 认证）。 这是因为 remember-me 服务在 auto-config="true"的时候启动了，它的认证机制需要 UserDetailsService 来实现（参考 Remember-me 章获得更多信息）。 如果你遇到了一个因为没定义 UserDetailsService造成的问题，那就试着去掉 auto-config配置（或者是其他你配置上的 remember-me）。

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*" filters="none"/>
  <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 不应该被安全 filter 处理。 否则，这些请求会被 /*部分拦截，它没法访问到登录页面。 如果你想使用基本认证而不是表单登录，可以把配置修改成如下所示：

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

如果你想用数据库，可以使用下面的方式

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

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-provider user-service-ref='myUserDetailsService'/>


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

你也可以通过在 bean 定义中添加 <custom-authentication-provider>元素来使用标准 AuthenticationProvider bean。 查看 Section 2.6, “默认验证管理器”了解更多信息。

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-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>
```

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 Chapter 7，*信道安全*找到更详细的讨论。

2.3.3. 同步 Session 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.ui.session.HttpSessionEventPublisher<
/listener-class>
  </listener>
```

然后，在你的 application context 加入如下部分：

```
    <http>
      ...
      <concurrent-session-control max-sessions="1" />
    </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>
      ...
      <concurrent-session-control                    max-sessions="1"
exception-if-maximum-exceeded="true"/>
```

```
    </http>
```

第二次登录将被阻止。

2.3.4. OpenID 登录

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/" password="notused"
authorities="ROLE_USER" />
```

你应该可以使用 myopenid.com网站登录来进行验证了。

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 AuthenticationProcessingFilter，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。 每个 Spring Security 过滤器都实现了 Spring 的 Ordered 接口，这些通过命名空间穿件的过滤器在初始化的时候就预先被排好序了。 标准的过滤器在命名空间里都有自己的假名，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。

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

假名	过滤器累	命名空间元素或属性
CONCURRENT_SESSION_FILTER	ConcurrentSessionFilter	http/concurrent-session-control
SESSION_CONTEXT_INTEGRATION_FILTER	HttpSessionContextIntegrationFilter	http
LOGOUT_FILTER	LogoutFilter	http/logout
X509_FILTER	X509PreAuthenticatedProcessigFilter	http/x509
PRE_AUTH_FILTER	AstractPreAuthenticatedProcessingFilter Subclasses	N/A
CAS_PROCESSING_FILTER	CasProcessingFilter	N/A
AUTHENTICATION_PROCESSING_FILTER	AuthenticationProcessingFilter	http/form-login
BASIC_PROCESSING_FILTER	BasicProcessingFilter	http/http-basic
SERVLET_API_SUPPORT_FILTER	SecurityContextHolderAwareRequestFilter	http/@servlet-api-provision
REMEMBER_ME_FILTER	RememberMeProcessingFilter	http/remember-me
ANONYMOUS_FILTER	AnonymousProcessingFilter	http/anonymous

假名	过滤器累	命名空间元素或属性
EXCEPTION_TRANSLATION_FILTER	ExceptionTranslationFilter	http
NTLM_FILTER	NtlmProcessingFilter	N/A
FILTER_SECURITY_INTERCEPTOR	FilterSecurityInterceptor	http
SWITCH_USER_FILTER	SwitchUserProcessingFilter	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：

```
<beans:bean id="myFilter"
class="com.mycompany.MySpecialAuthenticationFilter">
  <custom-filter position="AUTHENTICATION_PROCESSING_FILTER"/>
</beans:bean>
```

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 HttpSessionContextIntegrationFilter, 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。

2.3.6. 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。

2.4. 保护方法

Spring Security 2.0 大幅改善了对你的服务层方法添加安全。 如果你使用 Java 5 或更高版本，还支持 JSR-250 的安全注解，这与框架提供的@secured 注解相似。 你可以为单个 bean 提供安全控制，通过使用 intercept-methods 元素装饰 bean 声明，或者你可以使用 AspectJ 方式的切点来控制实体服务层里的多个 bean。

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。 这个例子来自 tutorial sample，如果你想在程序中使用方法安全，那么它是一个很好的开始：

```
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);
}
```

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。

2.4.2. intercept-methodsBean 渲染器

可选语法，让你可以指定安全，为一个特定的 bean，使用元素在 bean 里。

```
<bean:bean id="target" class="com.mycompany.myapp.MyBean">
    <intercept-methods>
        <protect method="set*" access="ROLE_ADMIN" />
        <protect method="get*" access="ROLE_ADMIN,ROLE_USER" />
        <protect method="doSomething" access="ROLE_USER" />
    </intercept-methods>
```

```
</bean:bean>
```

这让你配置安全属性，为单独的方法，在 bean 或简单通配符模式。

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。

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. 默认验证管理器

我们大概知道命名空间配置会自动为我们注册一个验证管理器 bean。 这是一个 Spring Security 的 ProviderManager类，如果你以前使用过框架，应该对它很熟悉了。 如果你在命名空间中使用了 HTTP 或方法安全，你不能使用自定义的 AuthenticationProvider，但你已经可以对使用的 AuthenticationProvider进行完全的控制了，所以这不会是一个问题。

你也许想为 ProviderManager 注册另外的 AuthenticationProvider bean ，你可以使用 <custom-authentication-provider>元素实现。比如：

```
<bean id="casAuthenticationProvider"

class="org.springframework.security.providers.cas.CasAuthenticationProvider">
    <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。 这里有一个特殊的元素，可以让你为 AuthenticationManager注册一个别名，然后你可以 application context 的其他地方使用这个名字。

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

<bean id="customizedFormLoginFilter"
class="org.springframework.security.ui.webapp.AuthenticationProcessingFilter">
    <security:custom-filter position="AUTHENTICATION_PROCESSING_FILTER"/>
    <property name="authenticationManager" ref="authenticationManager"/>
    ...
</bean>
```

[1] 你可以在 LDAP 的章节里，找到更多有关使用的 ldap-server的元素。

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.springframework.org/spring-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：

```
Authentication object is of type: org.springframework.security.providers.UsernamePasswordAuthenticationToken

Authentication object as a String:
org.springframework.security.providers.UsernamePasswordAuthenticationToken@1f127853:
Principal: org.springframework.security.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.ui.WebAuthenticationDetails@0:
RemoteIpAddress: 127.0.0.1; SessionId: k5qypsawgpwb;
```

```
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 简介中的描述获得源代码。 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.springframework.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.springframework.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。 我们也欢迎大家加入 Acegisecurity-developer 邮件列表，特别是对设计部分的讨论。交通量是非常轻的。

Part II. 总体结构

和大多数软件一样，Spring Security 有一个一定的中央接口，类和抽象概念，贯穿整个框架。 在指南的这个部分，我们将介绍 Spring Security，在解释这些中央元素之前，有必要进行成功的计划，执行 Spring Security 整合。

技术概述

5.1. 运行环境

Spring Security 可以运行在标准的 Java 1.4 运行环境下。 它也支持 Java 5.0，不过这部分代码单独打包起来，放到发布的，文件名是"tiger"前缀的 JAR 文件里。 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。

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. 共享组件

让我们展示一些 Spring Security 中很重要的共享组件。 被成为"shared"的组件，是指它在框架中占有很重要的位置，框架离开它们就没法运行。 这些 java 类表达了维持系统的构建代码块，所以理解他们的位置是非常重要的，即使你不需要直接跟他们打交道。

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。 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 usoyou 线程都使用同一个安全环境。 为了这种情况，我们而已使用 SecurityContextHolder.MODE_GLOBAL。 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。

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，很常见的，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 obj =
SecurityContextHolder.getContext().getAuthentication().getPrincipal();


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

上面的代码介绍了一定数量的，有趣的，几个关键对象之间的相互关系。 首先，你会注意到SecurityContextHolder 和 Authentication 之间的中间对象。 这个SecurityContextHolder.getContext()方法会直接返回 SecurityContext。

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。 大多数验证提供器使用 Spring Security 代理 UserDetailsService，作为验证过程的一部分。 这个 UserDetailsService用来建立Authentication对象，保存在 SecurityContextHolder里。好消息是我们提供了好几个 UserDetailsService 实现，其中一个使用内存里的 map，另一个使用 JDBC。 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。

最后，但不是最不重要的，优势你需要在 HTTP 请求之间共享 SecurityContext 其他时候，主体会在每个请求里重新验证，虽然大多数情况下它可以存储。 HttpSessionContextIntegrationFilter就是实现在 HTTP

请求之间存储 SecurityContext的。 就像它的类名一样， HttpSession被用来保存这些信息。 你不应该因为安全的问题，直接与 HttpSession打交道。 根本不存在这样做的理由-一直使用 SecurityContextHolder作替代方式。

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

就像这篇指南开头提到的那样，Spring Security 可在很多不同的验证环境下使用。 虽然我们推荐人们使用 Spring Security，不与已存在的容器管理认证系统整合，但它也是支持的-使用你自己的属性验证系统进行整合。让我们先看看 Spring Security 完全依靠自己，管理 web 安全，这里会演示最复杂和最常见的情况。

讨论一个典型的 web 应用验证过程：

- 你访问首页，点击一个链接。
- 向服务器发送一个请求，服务器判断你是否在访问一个受保护的资源。
- 如果你还没有进行过认证，服务器发回一个响应，提示你必须进行认证。 响应可能是 HTTP 响应代码，或者是重新定向到一个特定的 web 页面。
- 依据验证机制，你的浏览器将重定向到特定的 web 页面，这样你可以添加表单， 或者浏览器使用其他方式校验你的身份（比如，一个基本校验对话框，cookie，或者 X509 证书，或者其他）。
- 浏览器会发回一个响应给服务器。 这将是 HTTP POST 包含你填写的表单内容，或者是 HTTP 头部，包含你的验证信息。
- 下一步，服务器会判断当前的证书是否是有效的， 如果他们是有效的，下一步会执行。 如果他们是非法的，通常你的浏览器会再尝试一次（所以你返回的步骤二）。
- 你发送的原始请求，会导致重新尝试验证过程。 有希望的是，你会通过验证，得到猪狗的授权，访问被保护的资源。 如果你有足够的权限，请求会成功。否则，你会收到一个 HTTP 错误代码 403，意思是访问被拒绝。

Spring Security 使用鲜明的类负责上面提到的每个步骤。 主要的部分是（为了使用他们）是 ExceptionTranslationFilter， 一个 AuthenticationEntryPoint， 一个验证机制，一个 AuthenticationProvider。

5.3.1. ExceptionTranslationFilter

ExceptionTranslationFilter是一个 Spring Security 过滤器，用来检测是否抛出了 Spring Security 异常。 这些异常会被 AbstractSecurityInterceptor抛出，它主要用来提供验证服务。 我们会在下一节讨论 AbstractSecurityInterceptor，但是现在，我们只需要知道，它是用来生成 Java 异常，和知道跟 HTTP 没啥关系，或者如何验证一个主体。 而 ExceptionTranslationFilter提供这些服务，使用特点那个的响应，返

回错误代码 403（如果主题被验证了，但是权限不足-在上边的步骤七），或者启动一个 AuthenticationEntryPoint（如果主体没有认证，然后我们需要进入步骤三）。

5.3.2. AuthenticationEntryPoint

AuthenticationEntryPoint 对应中上面列表中的步骤三。 如你所想的，每个 web 应用程序都有默认的验证策略（好的，这可以在 Spring Security 里配置一切，但是让我们现在保持简单）。 每个主要验证系统会有它自己的 AuthenticationEntryPoint实现， 会执行动作，如同步骤三里的描述一样。

在你的浏览器决定提交你的认证证书之后（使用 HTTP 表单发送或者是 HTTP 头），服务器部分需要有一些东西来“收集”这些验证信息。 现在我们到了上述的第六步。 在 Spring Security 里，我们需要一个特定的名字，来描述从用户代码（通常是浏览器）收集验证信息的功能，这个名字就是“验证机制”。 在从用户代码哪里收集了验证细节之后，一个" Authentication 请求"对象会被 AuthenticationProvider建立。

5.3.3. AuthenticationProvider

Spring Security 认证过程的最后一个角色是 AuthenticationProvider。 非常简单，这是跟获得 Authentication请求对象相关的，决定它是否是有效的。 这个供应器或者抛出一个异常，或者返回一个完整的 Authentication对象。 还记得我们的好朋友 UserDetails和 UserDetailsService吗？ 如果不记得，回头看看前面的章节，刷新你的记忆。 大多数 AuthenticationProvider都会要求 UserDetailsService提供一个 UserDetails对象。 像上面提到的那样，大多数程序会提供他们自己的 UserDetailsService虽然一些可以使用 Spring Security 提供的 JDBC 和内存实现。 由此产生的 UserDetails对象-特别是 UserDetails中包含的 GrantedAuthority[]-将被用来组装 Authentication对象。

在验证机制重新获得了组装好的 Authentication对象以后，他会认为请求有效，把 Authentication放到 SecurityContextHolder里，然后导致原始请求重审（第七步）。 另一方面，如果 AuthenticationProvider 驳回了请求，验证机制会让用户代码重试（第二步）。

5.3.4. 直接设置 SecurityContextHolder 的内容

虽然这表述了一个典型的验证流程，但是好消息是 Spring Security 不在意你如何把一个 Authentication 放到 SecurityContextHolder里的。 唯一关键的需求是 SecurityContextHolder包含 Authentication，用来表现一个主体，在 AbstractSecurityInterceptor之前验证请求的。

你可以（很多用户都这样做）写一个自己的过滤器或 MVC 控制器来提供验证系统的交互，这些都不是基于 Spring Security 的。 比如，你也许使用容器管理验证，从 ThreadLocal 或 JNDI 里获得当前用户信息。 或者，你的公司可能有一个遗留系统，它是一个企业标准，你不能控制它。 这种情况下，很容易让 Spring Security 工作，也能提供验证能力。 你所需要的就是写一个过滤器（或等价物）从指定位置读取第三方用户信息，把它放到 SecurityContextHolder 里。 实现这些很简单，这种整合是完全被支持的方法。

5.4. 安全对象

Spring Security 使用词组"secure object"来指代任何需要保护的对象（就像一个认证决定）。 最常见例子就是方法调用和 web 请求。

5.4.1. 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。

5.4.2. AbstractSecurityInterceptor

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。

5.4.2.1. 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 ObjectDefinitionSource 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（使用命名空间里的 <intercept-url>语法）。

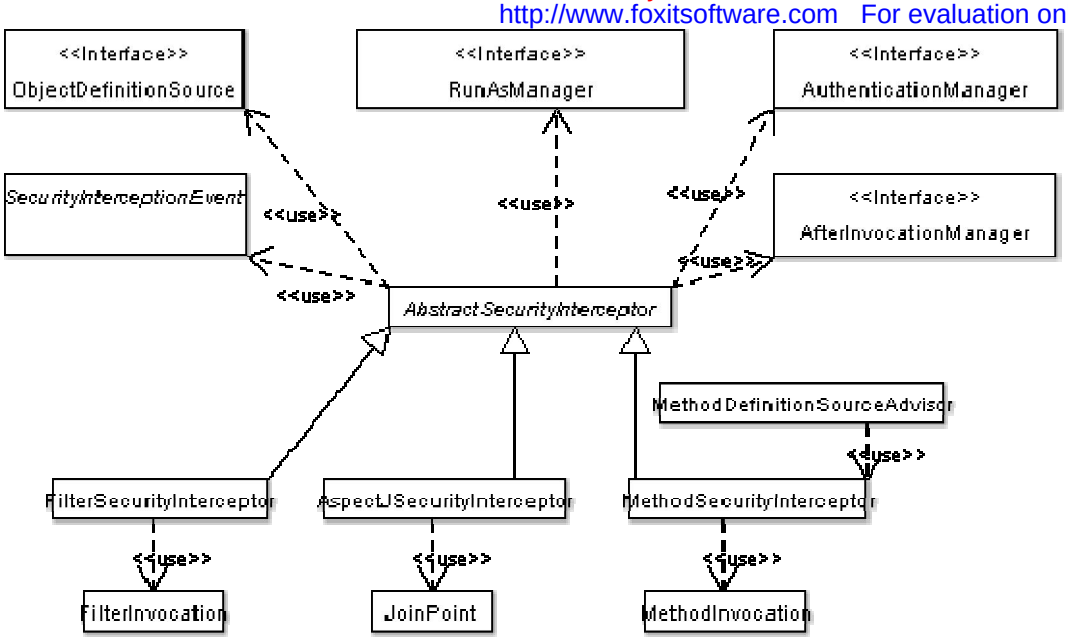
5.4.2.2. RunAsManager

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。

5.4.2.3. 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。



关键—“secure object”模型

Figure 5.1. 关键"secure object"模型

5.4.2.4. 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。

支持的基础设施

这章里讨论一些 Spring Security 中用到的基础设施。 如果这个功能跟安全没有直接关系，但是还是包含在 Spring Security 中了，我们就在这章里讨论它。

6.1. 国际化

Spring Security 支持异常信息的国际化，最终用户会很喜欢这点。 如果你的程序是为英语用户设计的，你不需要做任何事，因为默认情况下所有的 Spring Security 消息都是英文的。 如果你需要支持其他语言，你所需要做的就是看看这节内容。

所有的异常信息都是可以本地化的，包括验证失败和访问被拒绝（验证失败）信息。 开发者和系统发布者关注的异常和日志信息（包括不正确的属性，接口联系分隔，使用错误的构造器，开始验证册书，日志调试等级）等等，不能进行本地化，而是使用英文直接硬编码到了 Spring Security 代码里。

打开 spring-security-core-xx.jar，你可以找到 org.springframework.security 包，里面包含了一个 messages.properties文件。 你的 ApplicationContext应用应该引用这个文件，因为 Spring Security 实

现了 Spring 的 MessageSourceAware接口,希望这些消息会在 application context 启动的时候注入到程序中。通常，你需要做的就是，在你的 application context 里注册一个 bean，来引用这些信息，比如像下面这样：

```
<bean id="messageSource"
class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
  <property name="basename" value="org/springframework/security/messages"/>
</bean>
```

这个 messages.properties使用了标准的资源束命名方式，为 Spring Security 提供了默认的语言支持。 默认的语言是英文。 如果你没有注册消息源，Spring Security 依然可以正常工作，但是使用的是硬编码的英文信息版本。

如果你想自定义 messages.properties文件，或支持其他语言，你应该复制这个文件，改成对应的文件名，并把它注册到上面的 bean 定义中。 文件里的关键信息并不多，本地化应该不是一个很大的任务。 如果你对这个文件进行了本地化，请考虑把你的工作与社区分享，记录一个 JIRA 任务，把你的本地化版本 messages.properties以附件形式发送上来。

围绕本地化进行讨论，就要提到 Spring 里的 ThreadLocal，org.springframework.context.i18n.LocaleContextHolder。 你应该通过 LocaleContextHolder, 来表现每个用户想要的 Locale Spring Security 尝试从信息源中定位信息，使用从 ThreadLocal 里中获得的 Locale。 请参考 Spring 文档获得使用 LocaleContextHolder的更多信息，可以使用一些帮助类进行自动设置（比如 AcceptHeaderLocaleResolver, CookieLocaleResolver, FixedLocaleResolver, SessionLocaleResolver等等）。

6.2. 过滤器

Spring Security 用到了很多过滤器，参考指南的后续部分会一一提到。 如果你使用了命名空间配置，你就不用经常去明确指定过滤器 bean。 有几种可能情况，你希望对安全过滤器链进行完全控制，或许因为你使用的功能没法使用命名空间进行支持，或者你使用了自己自定义版本的类。

这种情况下，你可以选择向你的 web 应用成立里添加哪些过滤器，这里你可以使用 Spring 的 DelegatingFilterProxy或 FilterChainProxy。我们会在下面介绍它们两个。

在使用 DelegatingFilterProxy的时候，你会看到 web.xml 里这样的内容：

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

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

        <filter-mapping>
          <filter-name>myFilter</filter-name>
          <url-pattern>/*</url-pattern>
        </filter-mapping>
```

注意这个过滤器其实是一个 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。

在生命周期的问题上，要考虑在 IoC 容器里而不是在 servlet 容器里管理 Filter。 具体来说，到底是哪个容器应该调用 Filter的“启动”与“关闭”方法。 需要指出的是 Filter的初始化和销毁很容易受 servlet 容器的影响，如果一个 Filter依赖于较早初始化 Filter的配置，那么可能会引发一些问题。 Spring IoC 容器，从另一方面讲，拥有更强大的生命周期/IoC 接口（比如 InitializingBean, DisposableBean, BeanNameAware, ApplicationContextAware和很多其他的），拥有更容易理解的接口协议，可预见的方法调用顺序，支持自动绑定，更可以选择不用实现 Spring 的接口（比如通过 Spring XML 中的 destroy-method属性）。 介于这些原因，只要有可能的话，我们推荐使用 Spring 生命周期服务，代替 servlet 容器的生命周期。 可以参考 DelegatingFilterProxy的 Javadoc 获得更多信息。

最好别用 DelegatingFilterProxy，我们强烈推荐你使用 FilterChainProxy 代替它。 虽然 DelegatingFilterProxy是一个非常有用的类，问题是在需要使用几个过滤器的时候，需要在 web.xml中定义 <filter>和 <filter-mapping>入口的代码数量太多了。 为了解决这个问题，Spring Security 提供了一个 FilterChainProxy 类。 它绑定了一个 DelegatingFilterProxy(好像上面的例子那样)，但是使用的类是 org.springframework.security.util.FilterChainProxy。 过滤器链要声明在 application context 里，使用下面的代码：

```
    <bean                                                    id="filterChainProxy"
class="org.springframework.security.util.FilterChainProxy">
        <sec:filter-chain-map path-type="ant">
            <sec:filter-chain pattern="/webServices/**"

filters="httpSessionContextIntegrationFilterWithASCFalse,basicProcessingFilter,exc
eptionTranslationFilter,filterSecurityInterceptor"/>
            <sec:filter-chain pattern="/**"

filters="httpSessionContextIntegrationFilterWithASCTrue,authenticationProcessingFi
lter,exceptionTranslationFilter,filterSecurityInterceptor"/>
        </sec:filter-chain-map>
    </bean>
```

你可能注意到 FilterSecurityInterceptor 声明的不同方式。 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 HttpSessionContextIntegrationFilter（ASC 是 allowSessionCreation的简写，是 HttpSessionContextIntegrationFilter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，而不管它们被 FilterInvocationDefinitionSource声明了多少次。 你可以完全控制是否调用这些方法,通过代理 DelegatingFilterProxy的 targetFilterLifecycle初始化参数,。像上面讨论的那样，默认的 servlet 容器生命周期调用不会被 DelegatingFilterProxy代理。

在使用命名空间配置的时候，你可以通过同样的方式使用 filters = "none"属性，来建立 FilterChainProxy。 这将从安全过滤器链中完全忽略请求的模式。 注意，任何匹配这个路径的请求，不会要求认证，不会使用验证服务，可以自由的访问。

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。
- HttpSessionContextIntegrationFilter,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 - AuthenticationProcessingFilter, CasProcessingFilter, BasicProcessingFilter, HttpRequestIntegrationFilter, JbossIntegrationFilter 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。
- RememberMeProcessingFilter，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。
- AnonymousProcessingFilter，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。

上面所有的过滤器，都使用了 DelegatingFilterProxy 或 FilterChainProxy。推荐使用单独的 DelegatingFilterProxy 为每个程序代理一个单独的 FilterChainProxy，通过这个 FilterChainProxy 定义 Spring Security 的所有过滤器。

如果你使用了 SiteMesh，一定要确保 Spring Security 过滤器在 SiteMesh 的过滤器之前被调用。 这才可以保证为 SiteMesh 渲染器使用的 SecurityContextHolder 先被组装起来。

6.3. 标签库

Spring Security 捆绑了很多 JSP 标签库，提供了各种不同的服务。

6.3.1. 配置

所有的 taglib 类都包含在核心 spring-security-xx.jar 文件里， security.tld 文件放在 JAR 的 META-INF目录下。 这意味着，对于 JSP 1.2+的 web 容器，你可以直接把 JAR 放到 WAR 的 WEB-INF/lib目录下，然后就可以用了。 如果你使用的是 JSP 1.1 容器，你需要在你的 web.xml文件里声明 JSP taglib，还要把 security.tld放到 WEB-INF/lib目录下。 把下面的片段添加到 web.xml里：

```
<taglib>

  <taglib-uri>http://www.springframework.org/security/tags</taglib-uri>

  <taglib-location>/WEB-INF/security.tld</taglib-location>

</taglib>
```

6.3.2. 使用

现在你已经配置好了标签库，可以转到单独的参考指南章节，阅读如何使用他们的详细信息。 注意当使用标签时，你应该在你的 JSP 里包含 taglib 引用：

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

信道安全

7.1. 总述

为了在你的程序中协调认证与验证的要求，Spring Security 也可以让未认证的 web 请求携带一些属性。这些属性可能包括特定的传输类型，拥有特定的 HttpSession 属性设置，或其他。 最常见的需求是，让你的 web 请求使用特定的传输协议发送，比如 HTTPS。

威胁传输安全的一个重要因素是会话劫持。你的 web 容器通过用户代理发送 cookie 或 URL 重写，使用 jsessionid 来管理 HttpSession。 如果 jsessionid使用 HTTP 发送，会话标识符很可能在用户通过验证过后，被其他人拦截，再去模拟用户访问。 这是因为大多数 web 容器，为给定的用户维护相同的 session 标识符，即使他们从 HTTP 转换到 HTTPS 页面也不会改变。

如果会话劫持在你的特定程序里被认为是一个特别值得关注的威胁，唯一的解决方式就是在每个请求里都使用 HTTPS。 这意味着 jsessionid 再也不会通过不安全的通道。 你会需要确认你的 web.xml 定义的 <welcome-file>指向 HTTPS 位置。 Spring Security 在后面提供一个方法来帮助实现这些功能。

7.2. 配置

信道安全已在安全命名空间获得支持， 使用 <intercept-url>元素里的 requires-channel属性，就能很简单实现（也推荐这样做）。

为了明确配置信道安全你需要在你的 application context 中定义下面的过滤器：

```
    <bean                                                id="channelProcessingFilter"
class="org.springframework.security.securechannel.ChannelProcessingFilter">
      <property name="channelDecisionManager" ref="channelDecisionManager"/>
      <property name="filterInvocationDefinitionSource">
        <security:filter-invocation-definition-source path-type="regex">
          <security:intercept-url                          pattern="\A/secure/.*\Z"
access="REQUIRES_SECURE_CHANNEL"/>
          <security:intercept-url                    pattern="\A/acegilogin.jsp.*\Z"
access="REQUIRES_SECURE_CHANNEL"/>
          <security:intercept-url          pattern="\A/j_spring_security_check.*\Z"
access="REQUIRES_SECURE_CHANNEL"/>
          <security:intercept-url pattern="\A/.*\Z" access="ANY_CHANNEL"/>
        </security:filter-invocation-definition-source>
      </property>
    </bean>

    <bean                                                id="channelDecisionManager"
class="org.springframework.security.securechannel.ChannelDecisionManagerImpl">
      <property name="channelProcessors">
        <list>
```

```
        <ref bean="secureChannelProcessor"/>
        <ref bean="insecureChannelProcessor"/>
        </list>
      </property>
    </bean>

    <bean id="secureChannelProcessor"
class="org.springframework.security.securechannel.SecureChannelProcessor"/>
    <bean id="insecureChannelProcessor"
class="org.springframework.security.securechannel.InsecureChannelProcessor"/>
```

和 FilterSecurityInterceptor 一样，ChannelProcessingFilter也支持 Apache Ant 样式的路径。

这个 ChannelProcessingFilter 通过过滤所有的 web 请求，决定应用配置属性。 它将代理 ChannelDecisionManager。 默认实现 ChannelDecisionManagerImpl，应该涵盖了大多数情况。 它直接调用配置的 ChannelProcessor实例。 ANY_CHANNEL属性可以用来覆盖这种行为，忽略指定的 URL。 否则，一个 ChannelProcessor会重新检查请求，如果请求不符合（比如，它发送数据使用了错误的传输协议），就会执行一个重定向，抛出一个异常，或执行需要的任何一个操作。

Spring Security 里包含两个具体 ChannelProcessor 实现： SecureChannelProcessor 确保对应配置 REQUIRES_SECURE_CHANNEL 属性的请求使用 HTTPS 发送数据。 InsecureChannelProcessor 确保对应配置 REQUIRES_INSECURE_CHANNEL 属性的请求使用 HTTP 发送数据。 如果没有使用对应的传输协议，这两个实现都代理 ChannelEntryPoint。 包含在 Spring Security 里的这两个 ChannelEntryPoint实现，只是把请求简单重定向到合适的 HTTP 或 HTTPS。 合适的默认分配给 ChannelProcessor实现，为它们相应的配置属性关键字和它们代表的 ChannelProcessor，不过，你可以使用 application context 重写它们。

注意，重定向使用的是绝对地址（比如 http://www.company.com:8080/app/page），不是相对地址（比如 /app/page）。 在测试过程中，发现了 IE6 sp1 的一个 bug，在重定向时，如果只改变了端口，它就不能响应成功。因此,绝对 URL 就是用来在 PortResolverImpl中,解决这个 bug 检测逻辑,这是默认情况下被 Spring Security bean 使用的。 请参考 PortResolverImpl的 JavaDocs 获得更多信息。

你应该注意，在登录过程中，推荐使用信道安全，保持用户名和密码的安全。 如果你没有决定在基于表单的登录里使用 ChannelProcessingFilter。 请确认你的登录页，设置了 REQUIRES_SECURE_CHANNEL，并让 AuthenticationProcessingFilterEntryPoint.forceHttps的值是 true。

7.3. 总结

一旦配置好，使用信道安全过滤器就非常简单了。 简单请求页面，不限制协议（比如 HTTP 或 HTTPS）或端口（比如 80,8080,443,8443 等等）。 很明显，我们还需要配置初始请求（可能通过 web.xml 的 <welcome-file>或一个众所周知的主页 URL),但是一旦这个发生了,过滤器就会按照你的 application context 定义的那样执行重定向。

你也可以添加自己的 ChannelProcessor实现 ChannelDecisionManagerImpl。 比如，你可能在使用“输入图片中的内容”检测真人用户的过程中，要设置一个 HttpSession 属性。 你的 ChannelProcessor 会响应

REQUIRES_HUMAN_USER配置属性，然后重定向到合适的入口点，如果 HttpSession属性还没有设置，启动真人用户的验证过程。

为了决定是否进行一个安全检测，依靠 ChannelProcessor或 AccessDecisionVoter，前一个被设计成处理未认证的请求，而后一个被设计成处理已认证的请求。 因此，后一个可以访问验证主体的被授权权限。 另外，ChannelProcessor检测到问题，就会进行 HTTP/HTTPS 的重定向，来满足它的需求。 AccessDecisionVoter检测到一个问题，会导致一个 AccessDeniedException（取决于 AccessDecisionManager）。

Part III. 认证

我们已经在技术概述章中介绍了 Spring Security 的认证架构。 在参考指南的这部分，我们将单独讨论认证机制和他们对应的 AuthenticationProvider。 我们也会看看如何更普遍的配置认证，包括如何你有多个认证入口，需要链接在一起。

包括一些例外情况，我们会讨论 Spring Security bean 配置的全部细节，而不是像命名空间语法那种简化形式。 你应该重新看一下使用命名空间进行配置的介绍，如果它们符合你的需求还要看一下他们的选项。 你越是不断的使用框架，就越需要自定义内部行为，你可能会希望跟多了解，单独服务是如何实现的，哪个类需要找到进行扩展，等等。 这部分更集中于提供这类信息。 我们建议你从 Javadoc 以及它自己的源代码获得信息[2]。

---

[2] 从项目的网站可以找到 Javadoc API 和可以浏览的源代码交叉引用。

通用认证服务

8.1. 机制，供应者和入口

如果你使用 Spring Security 提供的认证方式，你通常需要配置一个 web 过滤器，AuthenticationProvider 和 AuthenticationEntryPoint。 在这章里，我们将要研究一个应用例子，它要支持基于表单的认证（比如，把一个非常好的 HTML 页面展现给用户，让他们进行登录）和基本认证（比如一个 web 服务或类似可访问的被保护资源）。

在 web.xml 里,这个程序需要单独的 Spring Security 过滤器来使用 FilterChainProxy。几乎所有 Spring Security 程序都有这样一个入口，看起来像这样：

```
    <filter>

        <filter-name>filterChainProxy</filter-name>


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

    </filter>



    <filter-mapping>

      <filter-name>filterChainProxy</filter-name>

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

    </filter-mapping>
```

上面的声明会让所有 web 请求，都通过叫做 filterChainProxy的 bean，通常它是 Spring Security 的 FilterChainProxy 的一个实例。 就像在这个参考指南的过滤器章节解释的那样， FilterChainProxy 是一个通用类,让 web 请求可以基于 URL 模式通过不同的过滤器。那些被委派的过滤器都管理在 application context 中，这样它们可以从依赖注入获得更大的益处。 让我们看一下 FilterChainProxy bean 定义，在 application context 的形式就像这样：

```
<bean id="filterChainProxy"
        class="org.springframework.security.util.FilterChainProxy">
    <security:filter-chain-map path-type="ant">
      <security:filter-chain pattern="/**"
filters="httpSessionContextIntegrationFilter,logoutFilter,authenticationProcessing
Filter,basicProcessingFilter,securityContextHolderAwareRequestFilter,rememberMePro
cessingFilter,anonymousProcessingFilter,exceptionTranslationFilter,filterInvocatio
nInterceptor,switchUserProcessingFilter"/>
    </security:filter-chain-map>
  </bean>
```

安全命名空间里的 filter-chain-map的语法，让你可以使用 filter-chain子元素序列，为过滤器链定义一个 URL 映射。 它们使用 pattern属性定义了一系列 URL，使用 filters属性定义了过滤器链。 这里需要重点注意的是，过滤器系列会按照定义的顺序执行 - 而且每个过滤器，都与在 application context 中定义的其他 bean 的 id相对应。 所以，我们看到一些特定的 bean 出现在 application context 里，他们可能叫作 httpSessionContextIntegrationFilter, logoutFilter等等。 过滤器的顺序，在参考文档的过滤器章节讨论 - 不过上面的例子没有问题。

在我们的例子中，我们使用了 AuthenticationProcessingFilter, BasicProcessingFilter。 他们是，响应基于表单认证和基本 HTTP header 认证的“认证机制”（我们在参考文档的前面部分讨论过认证机制的角色）。 如果你没有使用表单或基本认证，就不需要定义这几个 bean。 你应该使用，你想要的认证环境需要的过滤器，比如 DigestProcessingFilter或 CasProcessingFilter。 参考有关这部分参考文档的单独章节，学习如何配置每个认证机制。

让我们回忆一下， HttpSessionContextIntegrationFilter 保存了在 HTTP session 调用中的 SecurityContext内容。 这意味着认证机制只在主体初始化尝试认证的时候使用一次。 剩下的时间里，认证机制就待在那里，静静的让请求通过，进入过滤器链的下一个过滤器里。 这是一个很现实的需求，认证过程很少需要在每个调用的时候起作用（BASIC 认证是一个例外），但是，在初始化认证完成之后，如果主体帐号发生了取消或禁用或其他改变（比如，增加或减少授权 GrantedAuthority[]），怎么办？ 让我们看看现在是如何处理的。

安全对象的主要认证提供器在前面介绍过，是 AbstractSecurityInterceptor。 这个类需要访问 AuthenticationManager。 它也可以配置成，一个 Authentication对象在每次安全对象调用的时候，是否需要重新认证。 默认情况下，如果 Authentication.isAuthenticated()返回 true，它就只从

SecurityContextHolder中取得已经认证的 Authentication。 这样执行很有效率，但是没办法处理即时认证的有效。 对这种情况，你就要把 AbstractSecurityInterceptor.alwaysReauthenticate属性设置成 true。

你可能会问自己，“AuthenticationManager 是什么？”。 我们之前还没有讲过它，但是我们讨论过 AuthenticationProvider 的概念。 非常简单，一个 AuthenticationManager 用来从一个 AuthenticationProvider 链传递请求，获得响应。 这跟我们以前讨论到的过滤器链有点儿相似，虽然这里有很多区别。 在 Spring Security 里，这里只有一个 AuthenticationManager实现，所以让我们看看，对于这章中的例子，它是如何配置的：

```
<bean id="authenticationManager"

        class="org.springframework.security.providers.ProviderManager">

<property name="providers">

<list>

  <ref local="daoAuthenticationProvider"/>

  <ref local="anonymousAuthenticationProvider"/>

  <ref local="rememberMeAuthenticationProvider"/>

</list>

</property>

</bean>
```

还要提到一点，你的认证机制（通常是过滤器）也要注入 AuthenticationManager 的引用。 所以 AbstractSecurityInterceptor 作为一个认证机制，也要使用上面的 ProviderManager 轮询一系列的 AuthenticationProvider。

在我们例子里有三个提供器。 他们按照顺序进行排列（这里使用的是 List而不是 Set），每个提供器都尝试进行认证，或跳过验证简单返回 null。 如果所有实现都返回 null，ProviderManager就会抛出一个对应的异常。 如果你对链装提供器感兴趣，请参考 ProviderManager的 JavaDocs。

这些提供器有时在不同认证机制之间是可互换的，其他时候，它们依赖于特定的认证机制。 比如 DaoAuthenticationProvider只需要基于字符串的用户名和密码。好多认证机制会产生基于字符串的用户名和密码的集合，包括（但不限于）表单和基本验证。 同样的，一些认证机制创建只能被对应 AuthenticationProvider使用的认证请求对象。 比如，一一对应的 JA-SIG CAS，他使用服务票据体型，只

能被 CasAuthenticationProvider 认证。 另一个一一对应的例子是 LDAP 认证机制，它只会被 LdapAuthenticationProvider处理。 每个类特定关系的细节在 JavaDocs 里，这个参考指南的认证特定方式的章节里都有详细说明。 你不需要特别注意这些实现细节，因为如果你忘记注册对应的提供器，你就会在尝试认证的时候收到一个 ProviderNotFoundException。

在正确配置完 FilterChainProxy 里的认证机制后，也确定对应的 AuthenticationProvider 注册到了 ProviderManager 中，你最后一步是配置一个 AuthenticationEntryPoint。 回忆一下我们以前讨论过的 ExceptionTranslationFilter的角色，它是在认证开始的使用，用来在基于 HTTP 请求，返回一个 HTTP 头或 HTTP 重定向。 继续我们以前的例子：

```
    <bean id="exceptionTranslationFilter"

            class="org.springframework.security.ui.ExceptionTranslationFilter">

      <property                                          name="authenticationEntryPoint"
ref="authenticationProcessingFilterEntryPoint"/>

      <property name="accessDeniedHandler">

        <bean class="org.springframework.security.ui.AccessDeniedHandlerImpl">

          <property name="errorPage" value="/accessDenied.jsp"/>

        </bean>

      </property>

    </bean>



    <bean id="authenticationProcessingFilterEntryPoint"


class="org.springframework.security.ui.webapp.AuthenticationProcessingFilterEntryP
oint">
```

```
    <property name="loginFormUrl" value="/login.jsp"/>

    <property name="forceHttps"><> value="false"/>

  </bean>
```

注意 ExceptionTranslationFilter 需要两个协作者。 第一个是 AccessDeniedHandlerImpl，使用 RequestDispatcher 转向到显示特定访问拒绝错误页面。 我们使用请求转向，这样 SecurityContextHolder 还能包含着主体的信息，这对呈现给用户是很有用的（老版本中，我们让 servlet 容器处理 403 错误信息，这会丢失很多有用的内容信息）。 AccessDeniedHandlerImpl也会把 HTTP header 设置成 403，这是一个正规的错误代码，描述拒绝访问。 至于 AuthentionEntryPoint，这里我们设置当一个未认证的主体尝试执行一个安全操作时，会执行什么动作。 因为在我们的例子里，我们将使用基于表单的认证，我们指定 AuthenticationProcessinFilterEntryPoint和登录用页面的 URL。 你的程序通常只会有一个入口点，大多数的认证方法定义它们自己的 AuthenticationEntryPoint。 每个认证方式对应哪个入口点，在参考指南的认证特定方式章节里讨论。

8.2. UserDetails 和相关类型

就像参考指南第一部分提到的，大多数认证提供器使用 UserDetails 和 UserDetailsService 接口。 后一个接口只有一个单独的方法：

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

返回的 UserDetails是一个接口，提供了 getter 方法，保障非空的基本认证信息，比如用户名，密码，授予的权限，和这个用户是否被禁用了。 大多数认证提供器会使用 UserDetailsService，即使用户名和密码其实没有在认证过程中用到。 通常，这种供应器会使用返回的 UserDetails 对象，只是使用其中的 GrantedAuthority[]信息，因为一些系统（像 LDAP 或 X509 或 CAS 等等）已经对整数的有效性进行过验证。

一个由 Spring Security 提供的 UserDetails单独的具体实现，是 User类。 Spring Security 用户将需要决定什么时候编写他们的 UserDetailsService，返回具体的 UserDetails类。 大多数情况下 User会直接使用，或被继承，虽然特定的环境（比如 ORM）可能需要用户根据脚手架编写他们自己的 UserDetails。 这并不是不常见的情况，用户应该毫不犹豫的返回他们正常的领域对象，展示系统的一个用户。 尤其是通常给 UserDetails提供附加的主体相关的属性（比如他们的电话号码和邮件地址），这样他们可以很容易在 web 视图里使用。

UserDetailsService是非常容易实现的，用户应该很容易使用他们自己选择持久化策略，获得认证信息。这里，Spring Security 确实包含了很多有用的实现，我们会在下面看到。

8.2.1. 内存里认证

虽然很容易创建一个自定义的 UserDetailsService实现，从选择的持久化引擎里获得确切信息，不过很多应用不需要搞得这么复杂。 特别是在你进行一个快速原型或者仅仅开始集成 Spring Security 的时候，当你不是真的需要在配置数据库或写 UserDetailsService实现上面花费时间的时候。 为了这种情况，一个简单的选择是使用安全命名空间里的 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>
```

它也支持外部属性文件：

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

属性文件，应该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
```

8.2.2. JDBC认证

Spring Security 也包含一个 UserDetailsService 可以从 JDBC 数据源里获得认证信息。 内部使用了 Spring JDBC，避免了仅仅为了保存用户信息而实现 ORM 完全功能的复杂性。如果你的程序使用了 ORM 工具，你估计会喜欢自己写一个 UserDetailsService来重用你已经写好的映射文件。 回到 JdbcDaoImpl，一个配置的例子如下所示：

```
    <bean                                                    id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <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.userdetails.jdbc.JdbcDaoImpl">
        <property name="dataSource" ref="dataSource"/>
    </bean>
```

通过修改上面的 DriverManagerDataSource，你可使用不同的关系型数据库管理系统。 你也可以使用通常的 Spring 选项，从 JNDI 获得的公共数据源。

8.2.2.1. 默认用户数据库表结构

不论什么什么数据库，或者，不管 DataSource是如何获得的，必须用到一个标准的表结构，表结构定义的内容就在这里。 一个 HSQL 数据库的 DDL 内容应该像这样：

```
CREATE TABLE users (

username VARCHAR(50) NOT NULL PRIMARY KEY,

password VARCHAR(50) NOT NULL,

enabled BIT NOT NULL

);



CREATE TABLE authorities (

username VARCHAR(50) NOT NULL,

authority VARCHAR(50) NOT NULL

);



ALTER TABLE authorities ADD CONSTRAINT fk_authorities_users foreign key
(username) REFERENCES users(username);
```

如果默认的表结构不满足你的需要，JdbcDaoImpl提供了属性，可以自定义 SQL 语句。 请通过 JavaDocs 查找详细信息，不过请注意，这个类不是为了复杂的自定义子类设计的。 如果你有一个复杂的表结构或想要返回一个自定义 UserDetails实现，你最好重写你自己的 UserDetailsService。 这个 Spring Security 提供的基本实现，是为了典型情况，而不是覆盖所有可能出现的需求。

8.3. 并行会话处理

Spring Security 可以限制一个主体并行认证到同一系统的次数。 很多 ISV 利用这点来加强授权公里，网管也喜欢这个功能，因为它可以防止人们共享登录名。 你可以，比如，禁止用户"Batman"从两个不同的会话登录到 web 应用里。

要使用并行会话支持，你需要把下面内容添加到 web.xml 里：

```
    <listener>


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

    </listener>
```

另外，你需要把 org.springframework.security.concurrent.ConcurrentSessionFilter 添加到你的 FilterChainProxy 里。 ConcurrentSessionFilter 需要两个属性， sessionRegistry，这个通常指向一个 SessionRegistryImpl 实例，和 expiredUrl，在 session 过期的时候就指向这个页面。

web.xml 里的 HttpSessionEventPublisher 会在 HttpSession 创建或销毁的时候，发送一个 ApplicationEvent 给 Spring 的 ApplicationContext。 这是关键，它让 SessionRegistryImpl 知道什么时候 session 结束了。

你还需要装配 ConcurrentSessionControllerImpl ，然后从你的 ProviderManager bean 里引用它：

```
    <bean id="authenticationManager"

        class="org.springframework.security.providers.ProviderManager">

      <property name="providers">

        <!-- your providers go here -->

      </property>
```

```
    <property name="sessionController" ref="concurrentSessionController"/>
</bean>

<bean id="concurrentSessionController"
class="org.springframework.security.concurrent.ConcurrentSessionControllerImpl">
    <property name='maximumSessions" value="1"/>
    <property name="sessionRegistry">
      <bean class="org.springframework.security.concurrent.SessionRegistryImpl"/>
    <property>
</bean>
```

8.4. 认证标签库

AuthenticationTag只是用来把当前的 Authentication对象的一个属性，输出到网页上。

下面的 JSP 片段，展示了如何使用 AuthenticationTag:

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

这个标签会把主体的名称显示出来。 这里我们假设 Authentication.getPrincipal()是一个UserDetails对象，通常是由 Spring Security 的标准 AuthenticationProvider实现的。

DAO认证提供器

9.1. 综述

Spring Security 包含了一个产品级别的 AuthenticationProvider 实现，叫做 DaoAuthenticationProvider。 这个认证提供器兼容所有生成 UsernamePasswordAuthenticationToken的验证机制，它可能是框架里最常用到的提供器。 与其他认证提供器一样，DaoAuthenticationProvider 通过一个 UserDetailsService 来获得用户名，密码和 GrantedAuthority[]。 与其他认证提供器不同的是，这个认证提供器需要获得一个密码，它会根据认证请求对象里的密码来判断认证是否成功。

9.2. 配置

你需要把 DaoAuthenticationProvider 加入你的 ProviderManager 列表（如我们在参考指南这章开始讲到的），并确保合适的认证机制配置为 UsernamePasswordAuthenticationToken，其他对提供器自己的配置基本和下面一样简单：

```
<bean id="daoAuthenticationProvider"

class="org.springframework.security.providers.dao.DaoAuthenticationProvider">
    <property name="userDetailsService" ref="inMemoryDaoImpl"/>
    <property name="saltSource" ref bean="saltSource"/>
    <property name="passwordEncoder" ref="passwordEncoder"/>
</bean>
```

PasswordEncoder和 SaltSource是可选的。 PasswordEncoder为从配置好的 UserDetailsService中返回的 UserDetails对象里的密码，提供编码和反编码的功能。 SaltSource使用盐值生成密码，这可以提升认证资源密码的安全性。 Spring Security 支持 MD5，SHA 和纯文本编码的 PasswordEncoder实现。 Spring

Security 提供了两种 SaltSource 实现： SystemWideSaltSource 对所有密码都使用相同的盐值进行编码， ReflectionSaltSource，使用返回的 UserDetails对象的属性来获得盐值。 请参考 JavaDocs 获得这些选项的更多信息。

除了上述的属性， DaoAuthenticationProvider还可以为 UserDetails对象提供缓存。 UserCache接口可以让 DaoAuthenticationProvider把一个 UserDetails对象放到缓存里，在以后的认证进程中，如果需要同样的用户名就会重新获得它。 默认情况下， DaoAuthenticationProvider使用 NullUserCache，这意味着不使用缓存。 Spring Security 也提供了一个可用的缓存实现， EhCacheBasedUserCache，如下配置：

```
    <bean id="daoAuthenticationProvider"


class="org.springframework.security.providers.dao.DaoAuthenticationProvider">

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

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

    </bean>



    <bean                                                          id="cacheManager"
class="org.springframework.cache.ehcache.EhCacheManagerFactoryBean">

      <property name="configLocation" value="classpath:/ehcache-failsafe.xml"/>

    </bean>



    <bean                                                      id="userCacheBackend"
class="org.springframework.cache.ehcache.EhCacheFactoryBean">

      <property name="cacheManager" ref="cacheManager"/>
```

```
    <property name="cacheName" value="userCache"/>

  </bean>



  <bean id="userCache"
class="org.springframework.security.providers.dao.cache.EhCacheBasedUserCache">

    <property name="cache" ref="userCacheBackend"/>

  </bean>
```

所有 Spring Security 的 EH-CACHE 实现(包括 EhCacheBasedUserCache)要求一个 EH-CACHE 的 Cache 对象。 这个 Cache对象可以从任何地方获得，不过我们推荐你使用 Spring 的工厂类，请参考 Spring 文档获得更多的细节，如何优化缓存存储位置，内存使用，剔除策略，超时等。

Note

大部分情况，你的程序都是有状态的 web 应用程序，你不需要使用缓存，因为用户的认证信息会被保存在HttpSession里。

我们在设计的时候，决定不支持为 DaoAuthenticationProvider 加锁，因为这样会加大UserDetailsService接口的复杂度。 比如，一个方法可能需要在认证请求失败的时候进行累加计数。 这些功能都可以利用下面讨论到的应用程序事件发布功能，来轻松实现。

DaoAuthenticationProvider返回一个 Authentication对象，包含有 principal属性集合。 它的内容可能是 String(基本就是用户名）或者是 UserDetails对象（是从 UserDetailsService获得的）。 默认返回的是 UserDetails, 它允许添加系统特定的潜在属性，比如用户的全名，邮件地址等。 如果使用容器适配器，或者如果你的程序就是要操作 String(就像之前在 Spring Security 0.6 里发布的情况一样），你应该把你程序里的 DaoAuthenticationProvider.forcePrincipalAsString属性设置成 true。

LDAP认证

10.1. 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。

10.2. 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）。

10.3. 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" />
```

10.3.1. 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。

10.3.2. 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 DN ou=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。

10.3.3. 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 groupOfUniqueMembers LDAP 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。

10.4. 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 org.springframework.security.providers.ldap.LdapAuthenticationProvider。 这个 bean 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。

10.4.1. 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。

10.4.1.1. 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 DN uid=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。

10.4.1.2. BindAuthenticator

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

10.4.1.3. PasswordComparisonAuthenticator

这个类 org.springframework.security.providers.ldap.authenticator.PasswordComparisonAuthenticator 实现了密码比较认证策略。

10.4.1.4. 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。

10.4.2. 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 SpringSecurityContextSource，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。

10.4.3. 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**。

10.4.3.1. 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。

10.4.4. 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。

10.4.5. 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.DefaultSpringSecurityContextSource">

        <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.providers.ldap.LdapAuthenticationProvider">
        <constructor-arg>
          <bean
class="org.springframework.security.providers.ldap.authenticator.BindAuthenticator
">
            <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.populator.DefaultLdapAuthoritiesPopulator
">
            <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.FilterBasedLdapUserSearch">

    <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。

10.4.6. 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,
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）。

表单认证机制

11.1. 概述

HTTP 表单认证，使用 AuthenticationProcessingFilter来处理一个登录表单。 这是系统认证最终用户的最常见的一种方法。 基于表单认证与 DAO 和 JAAS 认证提供器是完全兼容的。

11.2. 配置

登录表单包含 j_username 和 j_password 输入框，然后将数据发送到过滤器监听的一个 URL（默认是 /j_spring_security_check）。 你应该把 AuthenticationProcessingFilter 添加到你的 application context 中：

```
    <bean id="authenticationProcessingFilter"



class="org.springframework.security.ui.webapp.AuthenticationProcessingFilter">

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

      <property name="authenticationFailureUrl" value="/login.jsp?login_error=1"/>

      <property name="defaultTargetUrl" value="/"/>

      <property name="filterProcessesUrl" value="/j_spring_security_check"/>

    </bean>
```

配置好的 AuthenticationManager 会处理每个认证请求。 如果认证失败，浏览器会重定向到 authenticationFailureUrl。 AuthenticationException 会被放到 HttpSession 中，属性名是 AbstractProcessingFilter.SPRING_SECURITY_LAST_EXCEPTION_KEY，在错误页里为用户提供一个出错原因。

如果认证成功了，得到的 Authentication对象会被放到 SecurityContextHolder中。

一旦 SecurityContextHolder更新了，浏览器需要重定向到目标URL,这个URL通常保存在 HttpSession 里，属性名是 AbstractProcessingFilter.SPRING_SECURITY_TARGET_URL_KEY。 这个属性由 ExceptionTranslationFilter在抛出 AuthenticationException异常的时候自动设置，这样在登录完成之后，用户可以跳转到最初尝试访问的位置。 如果因为一些原因，HttpSession没办法分析目标URL，浏览器将重定向到 defaultTargetUrl属性的值。

基本认证机制

12.1. 概述

Spring Security 提供一个 BasicProcessingFilter，它可以处理 HTTP 头部中的基本认证证书。 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， BasicProcessingFilter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）。

12.2. 配置

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

```
<bean id="basicProcessingFilter"
class="org.springframework.security.ui.basicauth.BasicProcessingFilter">
<property name="authenticationManager"><ref
bean="authenticationManager"/></property>
<property name="authenticationEntryPoint"><ref
bean="authenticationEntryPoint"/></property>
</bean>

<bean id="authenticationEntryPoint"

class="org.springframework.security.ui.basicauth.BasicProcessingFilterEntryPoint">
<property name="realmName"><value>Name Of Your Realm</value></property>
</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 BasicProcessingFilterEntryPoint，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。

摘要式认证

13.1. 概述

Spring Security 提供了一个 DigestProcessingFilter，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 DigestProcessingFilter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 DigestProcessingFilterEntryPoint 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），DigestProcessingFilterEntryPoint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。

DigestProcessingFilterEntryPoint 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。

13.2. 配置

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 DigestProcessingFilter。 application context 还需要定义 DigestProcessingFilter和它需要的合作伙伴：

```
    <bean id="digestProcessingFilter"
        class="org.springframework.security.ui.digestauth.DigestProcessingFilter">
      <property name="userDetailsService" ref="jdbcDaoImpl"/>
      <property name="authenticationEntryPoint"
ref="digestProcessingFilterEntryPoint"/>
      <property name="userCache" ref="userCache"/>
    </bean>

    <bean id="digestProcessingFilterEntryPoint"

class="org.springframework.security.ui.digestauth.DigestProcessingFilterEntryPoint
">
      <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 DigestProcessingFilter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 DigestProcessingFilterEntryPoint，这样 DigestProcessingFilter 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

14.1. 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。

14.2. 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>
```

如果你使用了 auto-config 设，它也会自动启用。 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 UserDetailsServicebean 的名字。

14.3. 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 [3]。 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)
```

14.4. 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 AuthenticationProcessingFilter 中，通过在它的超类 AbstractProcessingFilter 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，AbstractProcessingFilter 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。

14.4.1. TokenBasedRememberMeServices

这个实现支持在 Section 14.2, “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="rememberMeProcessingFilter"


class="org.springframework.security.ui.rememberme.RememberMeProcessingFilter">
      <property name="rememberMeServices" ref="rememberMeServices"/>
      <property name="authenticationManager" ref="theAuthenticationManager" />
    </bean>


    <bean                                                    id="rememberMeServices"
class="org.springframework.security.ui.rememberme.TokenBasedRememberMeServices">
      <property name="userDetailsService" ref='myUserDetailsService"/>
      <property name="key" value="springRocks"/>
    </bean>
```

```
<bean id="rememberMeAuthenticationProvider"

class="org.springframework.security.providers.rememberme.RememberMeAuthenticationP
rovider">
    <property name="key" value="springRocks"/>
</bean>
```

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

14.4.2. 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 14.3, “持久化标记方法”.

---

[3] 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。

Java认证和授权服务（JAAS）供应器

15.1. 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。

15.2. 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.providers.jaas.JaasAuthenticationProvider">

  <property name="loginConfig" value="/WEB-INF/login.conf"/>

  <property name="loginContextName" value="JAASTest"/>

  <property name="callbackHandlers">

    <list>

      <bean
class="org.springframework.security.providers.jaas.JaasNameCallbackHandler"/>

      <bean
class="org.springframework.security.providers.jaas.JaasPasswordCallbackHandler"/>

    </list>
```

```
    </property>

    <property name="authorityGranters">

      <list>

        <bean
class="org.springframework.security.providers.jaas.TestAuthorityGranter"/>

      </list>

    </property>

  </bean>
```

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

15.2.1. 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 JAAS LoginModule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 JaasAutheticationProvider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 JAAS LoginModule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。

15.2.2. 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。JaasAuthenticationProvider会创建一个 JaasGrantedAuthority（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。

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。

16.1. 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.ui.preauth 包下找到 web 相关的类，后台类都在 org.springframework.security.providers.preauth 包里。 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。

16.1.1. 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。

16.1.2. 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。

16.1.2.1. 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。

16.1.3. 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。

16.1.4. PreAuthenticatedProcessingFilterEntryPoint

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。

16.2. 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。

16.2.1. 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。

16.2.1.1. Siteminder 示例配置

使用这个过滤器的典型配置应该像这样：

```
<bean id="siteminderFilter"
```

```
class="org.springframework.security.ui.preauth.header.RequestHeaderPreAuthenticate
dProcessingFilter">
        <security:custom-filter position="PRE_AUTH_FILTER" />
        <property name="principalRequestHeader" value="SM_USER"/>
        <property name="authenticationManager" ref="authenticationManager" />
      </bean>

      <bean id="preauthAuthProvider"

class="org.springframework.security.providers.preauth.PreAuthenticatedAuthenticati
onProvider">
        <security:custom-authentication-provider />
        <property name="preAuthenticatedUserDetailsService">
          <bean id="userDetailsServiceWrapper"

class="org.springframework.security.userdetails.UserDetailsByNameServiceWrapper">
            <property name="userDetailsService" ref="userDetailsService"/>
          </bean>
        </property>
        </bean>

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

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。

16.2.2. 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 16.1.2.1, “J2eeBasedPreAuthenticatedWebAuthenticationDetailsSource”。

匿名认证

17.1. 概述

有时，特别是在 web 请求 URI 安全的情况里，为每个安全对象的调用都分配一个配置属性更方便。 不同的是，有时默认需要一个 ROLE_SOMETHING更好一些，这个规则只允许一些特定例外，比如登录，注销，系统的首页。 也有需要匿名认证的其他情况，比如审核拦截器查询 SecurityContextHolder 来确定哪个主体可以使用对应的操作。 如果它们能确定 SecurityContextHolder 里总是包含一个 Authentication对象，而不可能是 null，这些类就可以得到更多的鲁棒性。

17.2. 配置

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 AnonymousProcessingFilter，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="anonymousProcessingFilter"

class="org.springframework.security.providers.anonymous.AnonymousProcessingFilter"
>
      <property name="key" value="foobar"/>
      <property name="userAttribute" value="anonymousUser,ROLE_ANONYMOUS"/>
    </bean>

    <bean id="anonymousAuthenticationProvider"

class="org.springframework.security.providers.anonymous.AnonymousAuthenticationPro
vider">
      <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。 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="filterInvocationInterceptor"

class="org.springframework.security.intercept.web.FilterSecurityInterceptor">
      <property name="authenticationManager" ref="authenticationManager"/>
      <property                                          name="accessDecisionManager"
ref="httpRequestAccessDecisionManager"/>
      <property name="objectDefinitionSource">
        <security:filter-invocation-definition-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-invocation-definition-source>
      </property>
    </bean>
```

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。

X.509 认证

18.1. 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 模块确认证书，使用过滤器，传递一个配置好的 X.509 认证提供器，允许任何复杂的特定程序检测使用。 它也可以为应用程序的用户映射证书，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。

18.2. 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。

18.3. 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。

Chapter 19. CAS 认证

19.1. 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/products/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。

19.2. 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 2.0 支持 CAS 3。 在写文档的时候，CAS 服务器的版本是 3.2。

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 3.2 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。

19.3. 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 ServicePropertiesbean，到你的 application context 里。 这表现你的服务：

```
<bean id="serviceProperties"
class="org.springframework.security.ui.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 CasProcessingFilter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：

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

    <bean id="casProcessingFilter"
class="org.springframework.security.ui.cas.CasProcessingFilter">
        <security:custom-filter after="CAS_PROCESSING_FILTER"/>
        <property name="authenticationManager" ref="authenticationManager"/>
        <property name="authenticationFailureUrl" value="/casfailed.jsp"/>
        <property name="defaultTargetUrl" value="/"/>
    </bean>

    <bean id="casProcessingFilterEntryPoint"
```

```
        class="org.springframework.security.ui.cas.CasProcessingFilterEntryPoint">

        <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 CasProcessingFilterEntryPoint类。

CasProcessingFilter 的属性与 AuthenticationProcessingFilter 非常相似（在基于表单登录时用到）。 每个属性从字面上都很好理解。 注意我们也使用命名空间语法，为认证管理器设置一个别名，这样 CasProcessingFilter需要它的一个引用。

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

CasProcessingFilterEntryPoint 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：

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

            <security:custom-authentication-provider />

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

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

            <property name="ticketValidator">
```

```
            <bean
class="org.jasig.cas.client.validation.Cas20ServiceTicketValidator">
                <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。

替换验证身份

20.1. 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。

20.2. 配置

一个 Spring Security 提供的 RunAsManager 接口：:

```
    Authentication buildRunAs(Authentication authentication, Object object,
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.runas.RunAsManagerImpl">
  <property name="key" value='my_run_as_password"/>
</bean>

<bean id="runAsAuthenticationProvider"
    class="org.springframework.security.runas.RunAsImplAuthenticationProvider">
  <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。

容器适配器认证

21.1. 概述

非常早期版本的 Spring Security 使用容器适配器进行最终用户的认证。 虽然它运行良好，但是需要很多时间来支持不同的容器版本，对于开发者来说配置时间也太长了。 因为这个原因，HTTP 表单认证和 HTTP 基础认证方法才被开发出来，直到今天，被推荐用在几乎所有的程序中。

容器适配器让 Spring Security 可以将主机的最终用户程序与容器直接集成。 这种集合意味着，程序可以继续使用容易本身的认证和授权能力（比如 isUserInRole()和基于表单或基本认证），虽然 Spring Security 提供加强的安全拦截能力（应该注意到，Spring Security 也允许是用 ContextHolderAwareRequestWrapper 发送 isUserInRole(), 和简单的 servlet 规范兼容的方法）。

一般通过适配器进行容器和 Spring Security 集成。 适配器提供一个容器兼容的用户认证提供器，需要返回容器兼容的用户对象。

适配器由容器实例化，并定义在容器特定的配置文件里。 适配器将读取定义了普通的认证管理器设置的 Spring 的 application context，就像可以用来认证请求的认证提供器一样。 application context 通常叫做 acegisecurity.xml，要把它放在容器指定的位置。

Spring Security 现在支持 Jetty, Catalina (Tomcat), JBoss 和 Resin。 其他容器的适配器也很容易编写。

21.2. 适配器认证提供器

始终是这种情况，容器适配器生成的 Authentication对象还是需要在 AbstractSecurityInterceptor需要处理请求的时候被 AuthenticationManager 认证。 AuthenticationManager 需要确认适配器提供的 Authentication对象是有效的，也确实被一个可信任的适配器认证过了。

适配器创建了 Authentication 对象，它是不变的，实现了 AuthByAdapter 接口。 这些对象保存了由适配器定义的 key 的散列码，。 这允许 Authentication对象被 AuthByAdapterProvider验证。 这个认证提供器定义如下：

```
<bean id="authByAdapterProvider"

        class="org.springframework.security.adapters.AuthByAdapterProvider">

<property name="key"><value>my_password</value></property>

</bean>
```

这里的 key 必须与定义在容器特定配置文件，用来启动适配器的 key 相同。 这个 AuthByAdapterProvider 自动获得任何有效的 AuthByAdapter实现，然后返回期待的 key 的散列值。

重申一下，这意味着适配器会在初始化认证的时候使用供应器，比如 DaoAuthenticationProvider, 返回一个包含 key 的散列值的 AuthByAdapter 实例，。 然后，当应用程序调用安全拦截器管理的资源时，AuthByAdapter 实例，放在 SecurityContextHolder 里的 SecurityContext，会被程序的 AuthByAdapterProvider测试到。这里不需要附加认证供应器，比如放到应用程序特定的 application context 里的 DaoAuthenticationProvider, 这是唯一的 Authentication对象类型，会被从容器适配器里的程序用到的。

Classloader 问题常常困扰着容器，以后会描述容器适配器的用法。 每个容器要求一个特别指定的配置。 安装教程提供在下面。 一旦安装好，请花点儿时间尝试一下例子，让你对容器适配器的配置有更多的了解。

在使用容器适配器和 DaoAuthenticationProvider的时候，要确保将 forcePrincipalAsString属性设置成 true。

21.3. Jetty

以下代码，在 Jetty 4.2.18 下通过测试。

$JETTY_HOME代表你 Jetty 安装的根目录。

编辑 $JETTY_HOME/etc/jetty.xml 文件，在 <Configure class> 部分添加一个新的 addRealm调用：

```
<Call name="addRealm">
<Arg>
  <New
class="org.springframework.security.adapters.jetty.JettySpringSecurityUserRealm">
    <Arg>Spring Powered Realm</Arg>
    <Arg>my_password</Arg>
    <Arg>etc/acegisecurity.xml</Arg>
  </New>
</Arg>
</Call>
```

把 acegisecurity.xml 复制到 $JETTY_HOME/etc目录下。

把下面的文件复制到 $JETTY_HOME/ext目录下：

- aopalliance.jar
- commons-logging.jar
- spring.jar
- acegi-security-jetty-XX.jar
- commons-codec.jar
- burlap.jar
- hessian.jar

上面那些 JAR 文件（或者是 acegi-security-XX.jar）都不能放在你程序的 WEB-INF/lib目录下。 你的 web.xml里设置的 realm 名字是与 Jetty 对应的。web.xml必须使用与你的 jetty.xml里相同的 <realm-name>（像上面的例子里，"Spring Powered Realm"）。

21.4. JBoss

下面的代码在 JBoss 3.2.6 下通过了测试。

$JBOSS_HOME代表你 JBoss 安装的根目录。

有两个不同方法，把 spring 环境集成到 Jboss 里。

第一种方法是，修改你的 $JBOSS_HOME/server/your_config/conf/login-config.xml 文件，这样它就在 <Policy>部分包含了新的入口：

```
<application-policy name = "SpringPoweredRealm">
<authentication>
<login-module code =
"org.springframework.security.adapters.jboss.JbossSpringSecurityLoginModule"
flag = "required">
<module-option name = "appContextLocation">acegisecurity.xml</module-option>
<module-option name = "key">my_password</module-option>
</login-module>
</authentication>
</application-policy>
```

把 acegisecurity.xml 复制到 $JBOSS_HOME/server/your_config/conf目录下。

在配置文件 acegisecurity.xml 里包含了 spring 环境的定义，包含了所有认证管理的 bean。 我们必须注意， SecurityContext会在每次请求时被创建销毁，所以登录操作会造成很大的资源消耗。 可选的第二个方法通过 org.springframework.beans.factory.access.SingletonBeanFactoryLocator使用 Spring 单例。需要的配置方法如下：

```
<application-policy name = "SpringPoweredRealm">
<authentication>
<login-module code =
"org.springframework.security.adapters.jboss.JbossSpringSecurityLoginModule"
flag = "required">
<module-option name = "singletonId">springRealm</module-option>
```

```
        <module-option name = "key">my_password</module-option>
        <module-option name =
"authenticationManager">authenticationManager</module-option>
      </login-module>
    </authentication>
    </application-policy>
```

上面的代码片段， authenticationManager是一个助手属性，定义成期望的 AuthenticationManager的名字，在 IoC 容器里有多个定义的时候可以用到。 这个 singletonId属性引用了定义在 beanRefFactory.xml中的一个 bean。 这个文件需要放在 JBoss 的 classpath 中，包括 $JBOSS_HOME/server/your_config/conf。这个 beanRefFactory.xml包含了下面的声明：

```
    <beans>
    <bean id="springRealm" singleton="true" lazy-init="true"
class="org.springframework.context.support.ClassPathXmlApplicationContext">
    <constructor-arg>
      <list>
        <value>acegisecurity.xml</value>
      </list>
    </constructor-arg>
    </bean>
    </beans>
```

最后，无论使用了哪种配置方法，你需要把下面的文件复制到 $JBOSS_HOME/server/your_config/lib目录下：

- aopalliance.jar

- spring.jar
- acegi-security-jboss-XX.jar
- commons-codec.jar
- burlap.jar
- hessian.jar

上面那些 JAR 文件（或者是 acegi-security-XX.jar）都不能放在你程序的 WEB-INF/lib目录下。 你的 web.xml 里设置的 realm 名字是与 JBoss 对应的。 然而，你 web 应用的 WEB-INF/jboss-web.xml 里的 <security-domain>应该与你的 login-config.xml 中的内容一样。 比如，你的 jboss-web.xml 看起来应该是这样：

21.5. Resin

下面的代码在 Resin 3.0.6 下通过测试。

$RESIN_HOME代表你 Resin 安装的根目录。

Resin 提供了好几种方法实现容器适配器。在下面的教程里,我们使用了尽量与其他容器适配器一致的配置。这会让 Resin 用户更容易发布同样的程序，确保配置正确。 开发者对 Resin 感到舒服，自然可以使用它的方法，打包 JAR 同 web 程序自己，和/或支持单点登录。

把下面的文件复制到 $RESIN_HOME/lib目录下：

- aopalliance.jar
- commons-logging.jar
- spring.jar
- acegi-security-resin-XX.jar
- commons-codec.jar
- burlap.jar
- hessian.jar

与其他容器适配器使用的，容器范围的 acegisecurity.xml 文件不同，每个 Resin web 程序会包含自己的 WEB-INF/resin-acegisecurity.xml文件。 每个 web 应用会包含一个 resin-web.xml文件，Resin 用它来启动容器适配器：

```
    <web-app>

    <authenticator>

    <type>org.springframework.security.adapters.resin.ResinAcegiAuthenticator</typ
e>

    <init>

      <app-context-location>WEB-INF/resin-acegisecurity.xml</app-context-location>

      <key>my_password</key>

    </init>
```

```
</authenticator>

</web-app>
```

像上面提供的基本配置那样，上面那些 JAR 文件（或者是 acegi-security-XX.jar）都不能放在你程序的 WEB-INF/lib目录下。你的 web.xml里设置的 realm 名字是与 Resin 无关的,认证的类根据 <authenticator> 设置的。

21.6. Tomcat

下面的代码在 Jakarta Tomcat 4.1.30 和 5.0.19 通过的了测试。

$CATALINA_HOME代表你的 Catalina (Tomcat) 安装根目录。

编辑你的 $CATALINA_HOME/conf/server.xml 文件，让 <Engine> 部分只包含一个活动的 <Realm> 入口。一个 realm 入口的例子如下：

```
        <Realm


className="org.springframework.security.adapters.catalina.CatalinaSpringSecurityUs

erRealm"

            appContextLocation="conf/acegisecurity.xml"

             key="my_password" />
```

确认从你的 <Engine>部分删除了其他的 <Realm> 入口。

把 acegisecurity.xml 复制到 $CATALINA_HOME/conf目录下。

把 spring-security-catalina-XX.jar 复制到 $CATALINA_HOME/server/lib 目录下。

吧下面的文件复制到 $CATALINA_HOME/common/lib目录下：

- aopalliance.jar
- spring.jar
- commons-codec.jar
- burlap.jar
- hessian.jar

上面那些 JAR 文件（或者是 acegi-security-XX.jar）都不能放在你程序的 WEB-INF/lib目录下。 你的 web.xml里设置的 realm 名字是与 Catalina 无关的。

我们收到了在 Mac OS X 下使用这个容器适配器的问题报告。需要使用下面的脚本：

```
#!/bin/sh
```

```
export CATALINA_HOME="/Library/Tomcat"

export JAVA_HOME="/Library/Java/Home"

cd /

$CATALINA_HOME/bin/startup.sh
```

最后，重启 tomcat。

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，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 支持了权限分组

```
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

这些表对应 Spring Security 的 ACL 实现。

```
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,
    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) );
```

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。

通用授权概念

22.1. 授权

在认证部分简略提过了,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。

22.2. 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。

22.2.1. 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：

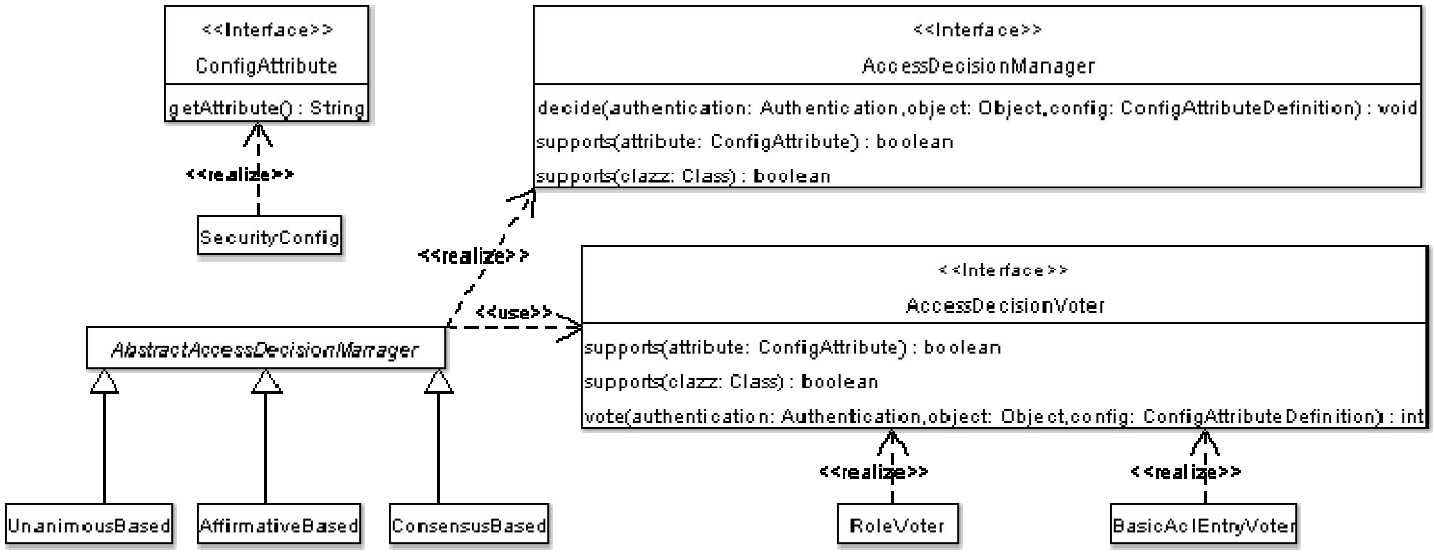
```
    void decide(Authentication authentication, Object secureObject,
ConfigAttributeDefinition config) throws AccessDeniedException;

    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。

22.2.1.1. 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 22.1, “投票决议管理器”显示有关的类。



投票决议管理器

Figure 22.1. 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, 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。

22.2.1.1.1. 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。

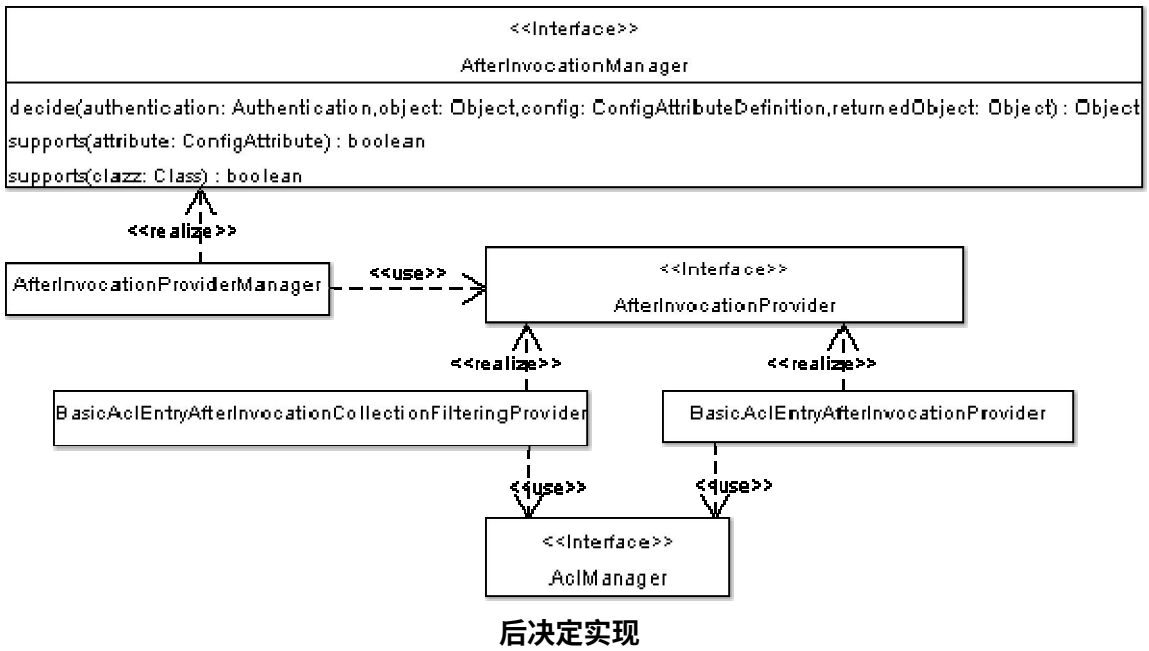
22.2.1.1.2. Custom Voters

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。

22.3. 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 22.2, “后决定实现” 展示 Spring Security 的 AfterInvocationManager 和它的具体实现。



后决定实现

Figure 22.2. 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。

22.3.1. ACL 提醒后决议提供器

请注意：Acegi Security 1.0.3 包含一个新 ACL 模型预览。 新 ACL 模型特别重写了以前的 ACl 模块。 新模块可以在 org.springframework.security.acls 包里找到，老 ACL 模块放在 org.springframework.security.acl。 我们推荐用户考虑测试一下新 ACL 模块，使用它建立程序。 老 ACL 模型应该考虑废弃了，可能在未来版本中删除。 下面的信息是关于新 ACL 包的，因此也是推荐的。

我们经常在服务层写的方法就像这样：

```
public Contact getById(Integer id);
```

常常，只有主体允许读取 Contact，就应该允许获得它。 在这情况， AccessDecisionManager 方法由 AbstractSecurityInterceptor提供将不够了。 这是因为 Contact的身份，在调用安全对象之前是完全有效地。 AclAfterInvocationProvider提供了一个解决方法，像下面这样配置：

```
<bean id="afterAclRead"


class="org.springframework.security.afterinvocation.AclEntryAfterInvocationProvide
r">
    <constructor-arg ref="aclService"/>
    <constructor-arg>
      <list>
        <ref
local="org.springframework.security.acls.domain.BasePermission.ADMINISTRATION"/>
        <ref
local="org.springframework.security.acls.domain.BasePermission.READ"/>
      </list>
    </constructor-arg>
  </bean>
```

在上面的例子里， Contact需要重新审核，并传递给 AclEntryAfterInvocationProvider。 提供器会抛出 AccessDeniedException异常，如果 requirePermission列表的其中一个没有放在 Authentication里。 AclEntryAfterInvocationProvider查询 Acl 服务，为 Authentication对应的领域对象决定应用 ACL。

一个简单的 AclEntryAfterInvocationProvider 是 AclEntryAfterInvocationCollectionFilteringProvider。 它用来从主体不允许访问的情况删除

Collection或元素数据，。 它永远不会抛出 AccessDeniedException - 直接安静的删除不允许的元素。 提供器如下配置：

```
    <bean id="afterAclCollectionRead"


class="org.springframework.security.afterinvocation.AclEntryAfterInvocationCollect
ionFilteringProvider">
      <constructor-arg ref="aclService"/>
      <constructor-arg>
        <list>
          <ref
local="org.springframework.security.acls.domain.BasePermission.ADMINISTRATION"/>
          <ref
local="org.springframework.security.acls.domain.BasePermission.READ"/>
        </list>
      </constructor-arg>
    </bean>
```

像你可以想像的，返回的 Object必须是让提供器可以操作的 Collection或数组。 如果 AclManager分析 Authentication没有获得一个队列中的 requirePermission，它就会删除元素。

Contacts 实例程序演示这两个 AfterInvocationProvider。

22.3.2. ACL-Aware AfterInvocationProviders (老 ACL 模块)

请注意：Acegi Security 1.0.3 包含一个新 ACL 模型预览。新 ACL 模型显著的重写了以前的 ACl 模块。新模块可以在 org.springframework.security.acls 包里找到，老 ACL 模块放在 org.springframework.security.acl。 我们推荐用户考虑测试一下新 ACL 模块，使用它建立程序。 老 ACL 模型应该考虑废弃了，可能在未来版本中删除。

常用服务层方法，我们所有写到一个状态或另一个像这样：

```
    public Contact getById(Integer id);
```

常常，只有主体允许读取 Contact，应该允许获得它。 在这情况，AccessDecisionManager 方法由 AbstractSecurityInterceptor提供将不够了。 这是因为 Contact的身份，在调用安全对象之前是完全有效地。 AclAfterInvocationProvider提供了一个解决方法，它像下面这样配置：

```
public Contact getById(Integer id);
```

常常，只有主体允许读取 Contact，应该允许获得它。 在这情况， AccessDecisionManager 方法由AbstractSecurityInterceptor提供将不够了。 这是因为 Contact的身份，在调用安全对象之前是完全有效地。 AclAfterInvocationProvider提供了一个解决方法，它像下面这样配置：

```
<bean id="afterAclRead"


class="org.springframework.security.afterinvocation.BasicAclEntryAfterInvocationPr

ovider">

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

    <property name="requirePermission">

      <list>

        <ref

local="org.springframework.security.acl.basic.SimpleAclEntry.ADMINISTRATION"/>

        <ref local="org.springframework.security.acl.basic.SimpleAclEntry.READ"/>

      </list>

    </property>

  </bean>
```

在上面的例子里， Contact需要重新审核，并传递给 AclEntryAfterInvocationProvider。 提供器会抛出 AccessDeniedException异常，如果 requirePermission列表的其中一个没有放在 Authentication里。 AclEntryAfterInvocationProvider查询 Acl 服务，为 Authentication对应的领域对象决定应用 ACL。

一个简单的 AclEntryAfterInvocationProvider 是 AclEntryAfterInvocationCollectionFilteringProvider。 它用来从主体不允许访问的情况删除 Collection或元素数据。 它永远不会抛出 AccessDeniedException - 直接安静的删除不允许的元素。 提供器如下配置：

```
<bean id="afterAclCollectionRead"
```

```
class="org.springframework.security.afterinvocation.BasicAclEntryAfterInvocationCo
llectionFilteringProvider">
      <property name="aclManager" ref="aclManager"/>
      <property name="requirePermission">
        <list>
          <ref
local="org.springframework.security.acl.basic.SimpleAclEntry.ADMINISTRATION"/>
          <ref local="org.springframework.security.acl.basic.SimpleAclEntry.READ"/>
        </list>
      </property>
    </bean>
```

像你可以想像的，返回的 Object 必须是 Collection 或数组，让提供器可以操作。 它会删除元素，如果AclManager分析 Authentication没有获得一个队列中的 requirePermission。

Contacts 实例程序演示这两个 AfterInvocationProvider。

22.4. 授权标签库

AuthorizeTag用来包括内容，如果当前主体拥有确定的 GrantedAuthority。

下面的 JSP 片段展示如何使用 AuthorizeTag:

```
    <security:authorize ifAllGranted="ROLE_SUPERVISOR">
    <td>
      <a href="del.htm?id=<c:out value="${contact.id}"/>">Del</a>
    </td>
    </security:authorize>
```

如果主体拥有 ROLE_SUPERVISOR 权限，这个标签就会显示标签的内容 A。

security:authorize 标签声明下面的属性：

- ifAllGranted: 这个标签列出的所有角色必须授权，才能输出标签的内容。

- ifAnyGranted: 任何一个这个标签列出的角色必须授权，才能输出标签的内容。
- ifNotGranted: 这个标签列出的角色没有一个是授权的，才能输出标签的内容。

你会注意到，每个属性你可以使用多个角色。 简单使用逗号来分隔角色。 这个 authorize标签忽略属性之间的空格。

标签库逻辑和所有它的参数一起。 这意味着如果你结合两个或多个属性，这个标签所有属性必须是 true，才会输出内容。 不要在 ifNotGranted="ROLE_SUPERVISOR"后面添加 ifAllGranted="ROLE_SUPERVISOR"，否则你会奇怪为什么总也看不到标签的内容。

如果想要所有的属性都返回 true，授权标签允许你创建更复杂的授权场景。 比如，你可以声明 ifAllGranted="ROLE_SUPERVISOR"，并在同一个标签里使用 ifNotGranted="ROLE_NEWBIE_SUPERVISOR"，可以防止新的超级用户看到标签内容。 然而你无疑可以直接使用 ifAllGranted="ROLE_EXPERIENCED_SUPERVISOR"，而不是在你的设计里添加 NOT 条件。

最后一件事：标签验证授权有特定的顺序：首先是 ifNotGranted，然后是 ifAllGranted，最后是 ifAnyGranted。

AccessControlListTag用来包含内容，判断当前的主体是否有一个 ACL 来显示领域对象。

下面的 JSP 片段，演示如何使用 AccessControlListTag:

```
<security:accesscontrollist domainObject="${contact}" hasPermission="8,16">
<td><a href="<c:url value="del.htm"><c:param name="contactId"
value="${contact.id}"/></c:url>">Del</a></td>
</security:accesscontrollist>
```

这标签会导致标签内容显示出来，如果主体拥有对于"contact"领域对象授权 16 或者授权 1。 这个数值其实是证书，使用 BasePermission位掩码。 请参考参考指南的 ACL 部分，了解更多 Spring Security 的 ACL 知识。

AclTag 是老 ACL 模型的一部分，应该考虑废弃。因为一些历史问题，它与 AccessControlListTag的作用相同。

安全对象实现

23.1. AOP 联盟 (MethodInvocation) 安全拦截器

在 Spring Security 2.0 之前，安全 MethodInvocation需要很多锅炉板配置. 现在推荐的方式，为方法安全，使用命名空间配置. 这个方式，方法安全基础设置 bean，自动配置为你，这样你不需要真的需要知道实现类。 我们只需要提供一个快速概述对这些类，在这里提到了。

方法安全提升，使用 MethodSecurityInterceptor，它会保障 MethodInvocation。 依靠配置方法，一个拦截器，可能指定一个单独的 bean，或在多个 bean 之间共享。 拦截器使用 MethodDefinitionSource实例获得配置属性，应用特定的方法调用。 MapBasedMethodDefinitionSource用来保存配置属性，关键字，通过方法名(也可以是通配符),会被用在内部,在属性定义在 application context 里,使用 <intercept-methods> 或 <protect-point>元素。 其他实现会用来处理基于注解的配置。

23.1.1. 精确的 MethodSecurityIterceptor 配置

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

```
    <bean id="bankManagerSecurity"


class="org.springframework.security.intercept.method.aopalliance.MethodSecurityInt
erceptor">

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

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

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

        <property name="objectDefinitionSource">

          <value>
```

```
        org.springframework.security.context.BankManager.delete*=ROLE_SUPERVISOR

org.springframework.security.context.BankManager.getBalance=ROLE_TELLER,ROLE_SUPERVISOR
        </value>
      </property>
    </bean>
```

23.2. 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.method.aspectj.AspectJSecurityInterceptor">
        <property name="authenticationManager" ref="authenticationManager"/>
        <property name="accessDecisionManager" ref="accessDecisionManager"/>
        <property name="afterInvocationManager" ref="afterInvocationManager"/>
```

```
    <property name="objectDefinitionSource">

      <value>


org.springframework.security.context.BankManager.delete*=ROLE_SUPERVISOR


org.springframework.security.context.BankManager.getBalance=ROLE_TELLER,ROLE_SUPER
VISOR

      </value>

    </property>

    </bean>
```

像你看到的，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 objectDefinitionSource， ObjectDefinitionSource 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，相对 AOP 库指定的对象（比如 MethodInvocation 或 JoinPoint），好像可以考虑更精确范围，在使用防伪决议（比如方法参数）。

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.method.aspectj.AspectJSecurityInterceptor;

    import org.springframework.security.intercept.method.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需要，因为 proceed();语句，有特定的含义，只有在 around()内容中。 AspectJSecurityInterceptor调用这个匿名 AspectJCallback类，在它想继续目标对象时。

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.DomainObjectInstanceSecurityAs

pect"

            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，无论用什么意思你想到的（比如 new Person();），他们会被安全拦截器应用。

23.3. FilterInvocation 安全拦截器

为了保护 FilterInvocation, 开发者需要添加 FilterSecurityInterceptor 到它们的过滤器链。 典型配置例子如下：

在 application context 你需要配置三个 bean：

```

```

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

    <bean id="authenticationEntryPoint"

class="org.springframework.security.ui.webapp.AuthenticationProcessingFilterEntryP
oint">
      <property name="loginFormUrl" value="/acegilogin.jsp"/>
      <property name="forceHttps" value="false"/>
    </bean>

    <bean id="filterSecurityInterceptor"

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

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

ExceptionTranslationFilter 提供了 Java 异常和 HTTP 响应之间的桥梁。 他们与维护的用户接口之间是完全独立的。 过滤器没有执行任何真实的安全提升。 如果一个 AuthenticationException 被检测到，过滤器会调用 AuthenticationEntryPoint 扎展开认证过程（比如一个用户登录）。

AuthenticationEntryPoint 会被调用，如果用户请求受保护的 HTTP 请求，但是他们没有认证。 类处理表现对应响应到用户，这样验证可以开始。 三个具体实现由 Spring Security 提供：AuthenticationProcessingFilterEntryPoint 为展开表单认证， BasicProcessingFilterEntryPoint 为展开 HTTP 基本认证过程，CasProcessingFilterEntryPoint 为展开 JA-SIG 中心认证服务（CAS）登录。AuthenticationProcessingFilterEntryPoint 和 CasProcessingFilterEntryPoint有可选项，可以强制使用 HTTPS，所以如果需要它，请参考 JavaDocs。

FilterSecurityInterceptor 用来处理 HTTP 资源的安全。 像其他安全拦截器，它需要引用 AuthenticationManager 和 AccessDecisionManager，它们都会在下面的单独章节里进行讨论。FilterSecurityInterceptor也使用配置属性进行配置，应用不同的 HTTP URL 请求。 一个对配置属性的完全讨论，提供在这份文档的高级设计章节中。

FilterSecurityInterceptor 可以使用两种方式配置配置属性。 第一种，在上面演示了，使用 <filter-invocation-definition-source>命名元素。 它直接使用 <filter-chain-map>配置一个 FilterChainProxy，但是<intercept-url>子元素，只使用 pattern和 access属性。 第二种，写你自己的 ObjectDefinitionSource，虽然这个超越了文档的范围。 不论使用哪种方法，ObjectDefinitionSource 用来返回 ConfigAttributeDefinition对象，包含所有配置属性，分配给单独的受保护 HTTP URL。

应该注意 FilterSecurityInterceptor.setObjectDefinitionSource()方法其实期望一个 FilterInvocationDefinitionSource实例。 这是一个继承了 ObjectDefinitionSource的标记接口。 它直接提供 ObjectDefinitionSource，理解 FilterInvocation。 感兴趣简单，我们继续参考

FilterInvocationDefinitionSource，像一个ObjectDefinitionSource，大多数用户的FilterSecurityInterceptor的区别都有很小的相关性。

在使用命名空间选项配置拦截器时，的哦好用来分割不同的配置属性，应用在每个 HTTP URL。每个配置属性分配到它自己的 SecurityConfig 对象里。SecurityConfig 对象在高级设计章中讨论。ObjectDefinitionSource 被属性编辑器创建，FilterInvocationDefinitionSource，匹配配置属性 FilterInvocations，基于请求 URL 的表达式计算。两个标准表达式语法被支持了。默认的是处理所有表达式，用 Apache Ant 路径和正则表达式，也支持 ore 复杂情况。path-type属性用来指定使用的模式类型。不可能在一个定义里使用组合表达式语法。比如，上一个配置使用正则表达式，代替 Ant 路径，会写成下面这样：

```
    <bean id="filterInvocationInterceptor"



class="org.springframework.security.intercept.web.FilterSecurityInterceptor">

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

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

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

      <property name="objectDefinitionSource">

        <security:filter-invocation-definition-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-invocation-definition-source>

      </property>

    </bean>
```

无论使用什么表达式语法，表达式通常执行，根据它们的定义。 因此重要的，更特定的表达式要放在不特定的表达式，在列表的高处。 在我们上面的例子里有提及，更特殊的 /secure/super/ 模式放在，较少特殊 /secure/模型，高的地方。 如果它们换了位置， /secure/会一直匹配， /secure/super/永远不会执行的。

像使用其他安全拦截器， validateConfigAttributes 属性被观察。 设置成 true 的时候（默认），在启动的时候 FilterSecurityInterceptor 会执行，如果提供的配置属性是有效的。 它检测每个可以执行的配置属性，通过 AccessDecisionManager 或 RunAsManager。 如果没有可以执行的配置属性，会抛出一个异常。

领域对象安全

24.1. 概述

请注意：在 2.0.0 之前，Spring Security 还叫 Acegi Security。 在老的 Acegi Security 发布里提供了一个 ACL 模块，放在 org.[acegisecurity/springsecurity].acl 下。 旧包现在被废弃了，会在 Spring Security 未来发布里删除。 这章提到的新 ACL 模块，是由 Spring Security 2.0.0 正式推荐的，可以在 org.springframework.security.acls包找到。

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。

24.2. 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。

24.3. 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。

acegi到 spring security的转换方式

作者 Chris.Baker，发表于 2008/04/22 - 9:44am.

http://java.dzone.com/tips/pathway-acegi-spring-security-

以前它叫做 spring 的 acegi 安全框架，现在重新标识为 spring security 2.0，它实现了简易配置的承诺，提高了开发者的生产力。 它已经是 java 平台上应用最广的安全框架了，在 sourceforge 上拥有 250,000 的下载量，Spring Security 2.0 又提供了一系列的新功能。

本文主要介绍了如果把之前建立在 acegi 基础上的 spring 应用转换到 spring security 2.0 上。

25.1. Spring Security 是什么

Spring Security 是目前用于替换 acegi 的框架，它提供了一系列新的功能。

- 大为简化了配置
- 继承 OpenID，标准单点登录
- 支持 windows NTLM，在 windows 合作网络上实现单点登录
- 支持 JSR 250("EJB 3")的安全注解
- 支持 AspectJ 切点表达式语言
- 全面支持 REST Web 请求授权
- 长期要求的支持组，层级角色和用户管理 API
- 提升了功能，使用后台数据库的 remember-me 实现
- 通过 spring webflow 2.0 对 web 状态和流转授权进行新的支持
- 通过 Spring Web Services 1.5 加强对 WSS（原来的 WS-Security）的支持
- 整个更多的.....

25.2. 目标

目前，我工作在一个 spring 的 web 应用上，使用 acegi 控制对资源的访问权限。 用户信息保存在数据库中，我们配置 acegi 使用了基于 JDBC 的 UserDetails 服务。 同样的，我们所有的 web 资源都保存在数据库里，acegi 配置成使用自定义的 AbstractFilterInvocationDefinitionSource，对每个请求检测授权细节。

随着 Spring Security 2.0 的发布，我想看看是不是可以替换 acegi，但还要保持当前的功能，使用数据库作为我们验证和授权的数据源，而不是 xml 配置文件（大多数演示程序里使用的都是 xml）。

这里是我采取的步骤.....

25.3. 步骤

- 第一步（也是最重要的）是下载新的 Spring Security 2.0 框架，并确保 jar 文件放到正确的位置（/WEB-INF/lib/）。

Spring Security 2.0 下载包里包含 22 个 jar 文件。我不需要把它们全用上（尤其是那些 sources 包）。在这次练习中我仅仅包含了以下几个：

- spring-security-acl-2.0.0.jar
- spring-security-core-2.0.0.jar
- spring-security-core-tiger-2.0.0.jar
- spring-security-taglibs-2.0.0.jar
- 在 web.xml 文件里配置一个 DelegatingFilterProxy

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

- Spring Security 2.0 的配置比 acegi 简单太多了，所以我没有在以前 acegi 配置文件的基础上进行修改，我发现从一个空白文件开始更简单。如果你想修改你以前的配置文件，我确定你删除的行数比添加的行数还要多。

配置文件的第一部分是指定安全资源过滤器的细节，这让安全资源可以通过数据库读取，而不是在配置文件里保存信息。这里是一个你将在大多数例子中看到的代码。

```
<http auto-config="true" access-denied-page="/403.jsp">
    <intercept-url pattern="/index.jsp" access="ROLE_ADMINISTRATOR,ROLE_USER"/>
```

```
        <intercept-url pattern="/securePage.jsp" access="ROLE_ADMINISTRATOR"/>

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

    </http>
```

使用这些内容进行替换：

```
    <authentication-manager alias="authenticationManager"/>



    <beans:bean                                                id="accessDecisionManager"
class="org.springframework.security.vote.AffirmativeBased">

        <beans:property name="allowIfAllAbstainDecisions" value="false"/>

        <beans:property name="decisionVoters">

            <beans:list>

                <beans:bean class="org.springframework.security.vote.RoleVoter"/>

                <beans:bean
class="org.springframework.security.vote.AuthenticatedVoter"/>

            </beans:list>

        </beans:property>

    </beans:bean>
```

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

    <beans:bean                                      id="secureResourceFilter"
class="org.security.SecureFilter.MySecureResourceFilter" />

    <http auto-config="true" access-denied-page="/403.jsp">
        <concurrent-session-control                          max-sessions="1"
exception-if-maximum-exceeded="true" />
        <form-login login-page="/login.jsp" authentication-failure-url="/login.jsp"
default-target-url="/index.jsp" />
        <logout logout-success-url="/login.jsp"/>
    </http>
```

这段配置的主要部分 secureResourceFilter，这是一个实现了 FilterInvocationDefinitionSource 的类，它在 Spring Security 需要对请求页面检测权限的时候调用。这里是 MySecureResouceFilter 的代码：

```
package org.security.SecureFilter;



import java.util.Collection;

import java.util.List;



import org.springframework.security.ConfigAttributeDefinition;

import org.springframework.security.ConfigAttributeEditor;

import org.springframework.security.intercept.web.FilterInvocation;

import
org.springframework.security.intercept.web.FilterInvocationDefinitionSource;



public class MySecureResourceFilter implements FilterInvocationDefinitionSource {



    public ConfigAttributeDefinition getAttributes(Object filter) throws
IllegalArgumentException {
```

```
FilterInvocation filterInvocation = (FilterInvocation) filter;



String url = filterInvocation.getRequestUrl();



// create a resource object that represents this Url object

Resource resource = new Resource(url);



if (resource == null) return null;

else{

    ConfigAttributeEditor configAttrEditor = newConfigAttributeEditor();

    // get the Roles that can access this Url

    List<Role> roles = resource.getRoles();

    StringBuffer rolesList = new StringBuffer();

    for (Role role : roles){

        rolesList.append(role.getName());

        rolesList.append(",");

    }
```

```
            // don't want to end with a "," so remove the last ","
            if (rolesList.length() > 0)
                rolesList.replace(rolesList.length()-1,   rolesList.length()+1,
"");
            configAttrEditor.setAsText(rolesList.toString());
            return (ConfigAttributeDefinition) configAttrEditor.getValue();
        }
    }

    public Collection getConfigAttributeDefinitions() {
        return null;
    }

    public boolean supports(Class arg0) {
        return true;
    }

}
```

getAttributes()方法返回权限的名称（我称之为角色），它们控制当前 url 的访问权限。

- 好了，现在我们需要安装信息数据库，下一步是让 Spring Security 从数据库中读取用户信息。这个 Spring Security 2.0 的例子告诉你如何从下面这样的配置文件里获得用户和权限的列表：

```
<authentication-provider>
    <user-service>
    <user name="rod" password="password" authorities="ROLE_SUPERVISOR, ROLE_USER" />
    <user name="dianne" password="password" authorities="ROLE_USER,ROLE_TELLER" />
    <user name="scott" password="password" authorities="ROLE_USER" />
    <user name="peter" password="password" authorities="ROLE_USER" />
    </user-service>
</authentication-provider>
```

你可以把这些例子的配置替换掉，这样你可以像这样从数据库中直接读取用户信息：

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

这里有一种非常快速容易的方法来配置安全数据库，意思是你需要使用默认的数据库表结构。默认情况下，<jdbc-user-service> 需要下面的几个表：user,authorities,groups,group_members 和 group_authorities。

我的情况下，我的安全数据库表无法这样工作，它和<jdbc-user-service>要求的不同，所以我需要修改<authentication-provider>：

```
    <authentication-provider>

        <jdbc-user-service data-source-ref="dataSource"

        users-by-username-query="SELECT U.username, U.password, U.accountEnabled AS 'enabled' FROM User U where U.username=?"

        authorities-by-username-query="SELECT U.username, R.name as 'authority' FROM User U JOIN Authority A ON u.id = A.userId JOIN Role R ON R.id = A.roleId WHERE U.username=?"/>

    </authentication-provider>
```

通过添加 users-by-username-query 和 authorities-by-username-query 属性，你可以使用你自己的 SQL 覆盖默认的 SQL 语句。就像在 acegi 中一样，你必须确保你的 SQL 语句返回的列与 Spring Security 所期待的一样。这里有另一个 group-authorities-by-username-query 属性，我在这里没有用到，所以也没有出现在这里例子中，不过它的用法与其他两个 SQl 语句的方法完全一致。

<jdbc-user-service>的这些功能大概是在上个月才加入的，在 Spring Security 之前版本中是无法使用的。幸运的是它已经被加入到 Spring Security 中了，这让我们的工作更加简单。你可以通过 这里和这里获取信息。

dataSource bean 中指示的是链接数据库的信息，它没有包含在我的配置文件中，因为它并不只用在安全中。这里是一个 dataSource 的例子，如果谁不熟悉可以参考一下：

```
    <bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">

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

        <property name="url" value="jdbc:mysql://localhost/db_name?useUnicode=true&characterEncoding=utf-8"/>
```

```
        <property name="username" value="root"/>

        <property name="password" value="pwd"/>

    </bean>
```

- 这就是 Spring Security 的所有配置文件。我最后一项任务是修改以前的登陆页面。在 acegi 中你可以创建自己的登陆<form>,向正确的URL发送正确命名的HTML输入元素。现在你也可以在Spring Security 2.0里这样做，只是一些名称发生了改变。你可以像以前一样使用用户名 j_username 和密码 j_password。

```
- <input type="text" name="j_username" id="j_username"/>
- <input type="password" name="j_password" id="j_password"/>
```

但是你必须把<form>中的 action 指向 j_spring_security_check 而不是 j_acegi_security_check。

```
    <form method="post" id="loginForm' action="<c:url
value='j_spring_security_check'/>"
```

在你的应用中有一些地方，用户可以进行注销，这是一个链接把注销请求发送给安全框架，这样它就可以进行相应的处理。需要把它从 j_acegi_logout 改成 j_spring_security_logout。

```
    <a href='<c:url value="j_spring_security_logout"/>'>Logout</a>
```

25.4. 总结

这个简短的指南，包含了如何配置 Spring Security 2.0 使用数据库中的资源，它并没有演示 Spring Security 2.0 中的新特性，然而我想它可以演示一些非常常用的框架功能，我希望你们觉得它有用。

Spring Security 2.0 与 acegi 相比的好处之一是配置文件非常简单，这在我比较老 acegi 配置文件（172 行）和新配置文件（42 行）的时候，清楚的显示出路爱。

这里是我完整的 securityContext.xml 文件：

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

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

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

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

xsi:schemaLocation="http://www.springframework.org/schema/beans,http://www.springframework.org/schema/beans/spring-beans-2.0.xsd,http://www.springframework.org/schema/security,http://www.springframework.org/schema/security/spring-security-2.0.xsd">



    <authentication-manager alias="authenticationManager"/>



    <beans:bean id="accessDecisionManager" class="org.springframework.security.vote.AffirmativeBased">

        <beans:property name="allowIfAllAbstainDecisions" value="false"/>

        <beans:property name="decisionVoters">

            <beans:list>
```

```
                    <beans:bean
class="org.springframework.security.vote.RoleVoter"/>

                    <beans:bean
class="org.springframework.security.vote.AuthenticatedVoter"/>

                </beans:list>

            </beans:property>

        </beans:bean>



        <beans:bean                                    id="filterInvocationInterceptor"
class="org.springframework.security.intercept.web.FilterSecurityInterceptor">

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

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

            <beans:property                           name="objectDefinitionSource"
ref="secureResourceFilter" />

        </beans:bean>
```

```
        <beans:bean                                    id="secureResourceFilter"
class="org.security.SecureFilter.MySecureResourceFilter" />



        <http auto-config="true" access-denied-page="/403.jsp">

                <concurrent-session-control                    max-sessions="1"
exception-if-maximum-exceeded="true" />

                <form-login                                  login-page="/login.jsp"
authentication-failure-url="/login.jsp" default-target-url="/index.jsp" />

                <logout logout-success-url="/login.jsp"/>

        </http>



        <beans:bean                                          id="loggerListener"
class="org.springframework.security.event.authentication.LoggerListener"/>



        <authentication-provider>

                <jdbc-user-service                   data-source-ref="dataSource"
users-by-username-query="SELECT U.username, U.password, U.accountEnabled AS 'enabled'
FROM User U where U.username=?" authorities-by-username-query="SELECT U.username,
R.name as 'authority' FROM User U JOIN Authority A ON u.id = A.userId JOIN Role R ON
R.id = A.roleId WHERE U.username=?" />
```

```
    </authentication-provider>

</beans:beans>
```

就像在第一步时提到到，下载 Spring Security 是最重要的步骤。从那里开始就可以一帆风顺了。