

Spring Security Reference

译: Spring安全参考

本帮助文档是 [觉得烦死](#) 整理--QQ:654638585

声明:

中文文档都是由软件翻译,翻译内容未检查校对,文档内容仅供参考。

您可以任意转发, 但请至保留作者&出处(<http://bolg.fondme.cn>), 请尊重作者劳动成果,谢谢!

Authors

译:作者

Ben Alex , Luke Taylor , Rob Winch , Gunnar Hillert , Joe Grandja , Jay Bryant

5.1.0.M1

版权所有©2004-2017

本文件副本可供您自行使用并分发给其他人, 前提是您不收取任何此类副本的费用, 并进一步规定每份副本均包含此版权声明, 无论是以印刷版还是电子版分发。

译:

Spring Security是一个强大且高度可定制的身份验证和访问控制框架。这是保护基于Spring的应用程序的事实标准。

Part I. Preface

译:第一部分前言

Spring Security为基于Java EE的企业软件应用程序提供了全面的安全解决方案。正如您在参考指南中发现的那样, 我们试图为您提供一个有用且高度可配置的安全系统。

安全是一个不断移动的目标, 并且追求全面的系统范围方法非常重要。在安全圈中, 我们鼓励您采用“安全层”, 以便每层都尽可能保证安全, 连续层提供额外的安全性。每层的安全性越“紧密”, 应用程序就越健壮和安全。在底层, 您需要处理诸如运输安全和系统识别等问题, 以缓解中间人攻击。接下来, 您通常会使用防火墙, 可能使用VPN或IP安全性来确保只有经过授权的系统才能尝试连接。在企业环境中, 您可以部署一个DMZ, 将面向公众的服务器与后端数据库和应用程序服务器分开。您的操作系统也将扮演重要角色, 解决诸如以无特权用户身份运行进程等问题并最大限度地提高文件系统安全性。操作系统通常也会配置自己的防火墙。希望在这个过程中的某个地方你会试图阻止对系统的拒绝服务和暴力攻击。入侵检测系统对于监视和响应攻击也特别有用, 因为这些系统能够采取保护措施, 例如实时阻止侵入TCP / IP地址。转移到更高层, 您的Java虚拟机将有望配置为最大限度地减少授予不同Java类型的权限, 然后您的应用程序将添加自己的问题特定于域的安全配置。Spring Security使后面的这个领域 - 应用程序的安全性更容易。

当然, 您需要正确处理上面提到的所有安全层, 以及包含每个层的管理因素。这些管理因素的非详尽清单将包括安全公告监控, 修补, 人员审查, 审计, 变更控制, 工程管理系统, 数据备份, 灾难恢复, 性能基准测试, 负载监控, 集中式日志记录, 事件响应程序等。

由于Spring Security专注于帮助企业应用程序安全层, 因此您会发现有多少不同的需求与业务问题域相同。银行应用程序对电子商务应用程序有不同的需求。电子商务应用程序对企业销售人员自动化工具有不同的需求。这些自定义要求使应用程序安全性变得有趣, 富有挑战性和有益。

请首先阅读Chapter 1, *Getting Started*的全部内容。这将向您介绍框架和基于命名空间的配置系统, 您可以使用该系统快速启动和运行。为了更好地理解Spring Security的工作原理以及您可能需要使用的一些类, 请阅读Part II, “Architecture and Implementation”。本指南的其余部分采用更传统的参考样式, 旨在根据需要进行阅读。我们还建议您尽可能多地阅读应用程序安全问题。Spring Security不是解决所有安全问题的万能药。从一开始, 应用程序的设计就要考虑到安全性, 这一点很重要。试图改造它并不是一个好主意。特别是, 如果您正在构建Web应用程序, 则应该意识到许多潜在的漏洞, 例如跨站脚本, 请求伪造和会话劫持, 您应该从一开始就考虑这些漏洞。OWASP网站 (<http://www.owasp.org/>) 保留了Web应用程序漏洞的十大列表以及大量有用的参考信息。

我们希望您发现本参考指南很有用, 我们欢迎您的反馈和 [suggestions](#) 。

最后, 欢迎来到Spring Security [community](#) 。

1. Getting Started

译:入门

本指南的后面部分提供了关于框架体系结构和实现类的深入讨论, 您需要了解是否需要进行任何严格的自定义。在这一部分, 我们将介绍Spring Security 4.0, 简要介绍一下该项目的历史, 并对如何开始使用该框架稍微考虑一下。特别是, 我们将着眼于命名空间配置, 与传统的Spring bean方法相比, 它提供了一种更简单的保护应用程序的方法, 您必须单独连接所有实现类。

我们还会看看可用的示例应用程序。在你阅读后面的章节之前, 值得尝试运行它们并尝试一些 - 你可以在对框架的理解增加的时候重新考虑它们。请同时查阅<http://spring.io / spring-security> [项目网站], 因为它有关于构建项目的有用信息, 以及指向文章, 视频和教程的链接。

2. Introduction

译:2介绍

2.1 What is Spring Security?

译:2.1什么是Spring Security

Spring Security为基于Java EE的企业软件应用程序提供全面的安全服务。特别强调支持使用Spring Framework构建的项目, Spring Framework是用于企业软件开发的领先Java EE解决方案。如果您没有使用Spring开发企业应用程序, 我们热烈鼓励您仔细研究它。对Spring的一些熟悉 - 特别是依赖注入原则 - 将帮助您更轻松地了解Spring Security。

人们使用Spring Security有很多原因, 但是大多数人在找到Java EE的Servlet规范或EJB规范的安全特性后, 都缺乏典型企业应用场景所需的深度。在提到这些标准的同时, 认识到它们在WAR或EAR级别不可移植是很重要的。因此, 如果切换服务器环境, 在新的目标环境中重新配置应用程序的安全性通常需要很多工作。使用Spring Security克服了这些问题, 并且还为您带来了许多其他有用的, 可自定义的安全功能。

正如您可能知道应用程序安全性的两个主要方面是“身份验证”和“授权”(或“访问控制”)。这是Spring Security的两大主要领域。“身份验证”是建立委托人的过程是他们自称的人(“委托人”通常是指用户, 设备或其他可以在您的应用程序中执行操作的系统)。“授权”是指决定是否允许委托人在您的申请内执行某项操作的过程。为了达到需要授权决定的地步, 委托人的身份已经由认证过程确定。这些概念很常见, 并不完全针对Spring Security。

在认证级别，Spring Security支持多种认证模式。这些认证模式大多由第三方提供，或者由相关标准组织（如互联网工程任务组）开发。另外，Spring Security还提供了自己的一套认证功能。具体而言，Spring Security目前支持与所有这些技术的认证集成：

- HTTP BASIC authentication headers (an IETF RFC-based standard)
- HTTP Digest authentication headers (an IETF RFC-based standard)
- HTTP X.509 client certificate exchange (an IETF RFC-based standard)
- LDAP (a very common approach to cross-platform authentication needs, especially in large environments)
- Form-based authentication (for simple user interface needs)
- OpenID authentication
- Authentication based on pre-established request headers (such as Computer Associates Siteminder)
- Jasig Central Authentication Service (otherwise known as CAS, which is a popular open source single sign-on system)
- Transparent authentication context propagation for Remote Method Invocation (RMI) and HttpInvoker (a Spring remoting protocol)
- Automatic "remember-me" authentication (so you can tick a box to avoid re-authentication for a predetermined period of time)
- Anonymous authentication (allowing every unauthenticated call to automatically assume a particular security identity)
- Run-as authentication (which is useful if one call should proceed with a different security identity)
- Java Authentication and Authorization Service (JAAS)
- Java EE container authentication (so you can still use Container Managed Authentication if desired)
- Kerberos
- Java Open Source Single Sign-On (JOSSO) *
- OpenNMS Network Management Platform *
- AppFuse *
- AndroMDA *
- Mule ESB *
- Direct Web Request (DWR) *
- Grails *
- Tapestry *
- JTrac *
- Jasypt *
- Roller *
- Elastic Path *
- Atlassian Crowd *
- Your own authentication systems (see below)

(*表示由第三方提供)

许多独立软件供应商（ISV）都采用Spring Security，因为这种灵活的身份验证模型的选择非常重要。这样做可以让他们快速地将他们的解决方案与他们最终客户需要的任何内容集成起来，而无需进行大量工程或要求客户改变其环境。如果上述认证机制都不符合您的需求，Spring Security是一个开放平台，编写您自己的认证机制非常简单。Spring Security的许多公司用户都需要与不遵循任何特定安全标准的“传统”系统集成，而Spring Security很高兴能够与这样的系统“很好地发挥”。

无论身份验证机制如何，Spring Security都提供了一套深层次的授权功能。有三个主要的感兴趣领域：授权Web请求，授权是否可以调用方法并授权访问单个对象实例。为帮助您理解这些差异，请分别考虑Servlet规范Web模式安全性，EJB容器托管安全性和文件系统安全性中的授权功能。Spring Security在所有这些重要领域提供了深入的功能，我们将在本参考指南的后面部分进行探讨。

2.2 History 译：225 历史

Spring Security在2003年底开始称为“春季Acegi安全系统”。Spring Developers的邮件列表上提出了一个问题，询问是否对基于Spring的安全实现给予了任何考虑。当时Spring社区的规模相对较小（特别是与今天的规模相比），而Spring本身的确只是从2003年初开始作为SourceForge项目而存在。对这个问题的回应是它是一个有价值的领域，尽管缺乏的时间目前阻止了它的探索。

考虑到这一点，构建了一个简单的安全实现，而不是发布。几个星期后，Spring社区的另一位成员询问了安全问题，并在当时向他们提供了这些代码。接下来还有其他几个要求，到2004年1月，大约有20个人在使用这些代码。这些先锋用户与其他人一起提出了一个建议SourceForge项目是有序的，该项目于2004年3月正式成立。

在那些早期，该项目没有任何自己的认证模块。集装箱安全管理被用于认证过程，而Acegi Security则专注于授权。这在一开始就很合适，但随着越来越多的用户请求额外的容器支持，容器特定身份验证领域接口的基本限制变得清晰起来。还有一个相关的问题，即向容器的类路径添加新的JAR，这是最终用户混淆和错误配置的常见原因。

随后引入了Acegi安全特定的认证服务。大约一年后，Acegi Security成为Spring Framework的正式子项目。2006年5月发布了1.0版的最终版本 - 经过两年半的积极使用众多生产软件项目以及数百项改进和社区贡献。

Acegi Security于2007年底成为春季投资组合项目，并更名为“春季安全”。

今天，Spring Security拥有一个强大且活跃的开源社区。在支持论坛上数千条关于Spring Security的消息。有一个积极的核心开发人员从事代码本身的工作，同时也是一个活跃的社区，他们也经常分享补丁和支持他们的同行。

2.3 Release Numbering 译：230 版本号

了解Spring Security发行版的工作原理非常有用，因为它可以帮助您确定迁移到项目未来版本所涉及的工作（或缺乏）。每个版本使用一个标准的整数三元组：MAJOR.MINOR.PATCH。目的是MAJOR版本不兼容，API的大规模升级。MINOR版本应该在很大程度上保留与旧版次版本的源代码和二进制兼容性，认为可能会有一些设计更改和不兼容的更新。PATCH级别应该完全兼容，前后颠倒，可能的例外是修改错误和缺陷的更改。

您受到更改影响的程度取决于您的代码的集成程度。如果您正在进行大量定制，则与使用简单名称空间配置相比，您可能会受到更多的影响。

在推出新版本之前，您应该始终彻底测试您的应用程序。

2.4 Getting Spring Security 译：240 获得 Spring Security

您可以通过几种方式获得Spring Security。您可以从主要的<http://spring>下载打包的发行版。io / spring-security [Spring Security]页面中，从Maven Central存储库（或Spring Maven存储库中下载快照和里程碑版本）下载各个jar，或者，您可以自己从源代码构建项目。

2.4.1 Usage with Maven 译：241 使用 Maven

最小的Spring Security Maven依赖关系集通常如下所示：

pom.xml中。

```
<dependencies>
<!-- ... other dependency elements ... -->
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-web</artifactId>
  <version>5.1.0.M1</version>
</dependency>
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-config</artifactId>
  <version>5.1.0.M1</version>
</dependency>
</dependencies>
```

如果您正在使用LDAP，OpenID等附加功能，则还需要包含相应的 [Section 2.4.3, "Project Modules"](#)。

Maven Repositories 译: Maven仓库

所有GA版本（即以.RELEASE结尾的版本）都部署到Maven Central，因此不需要在您的POM中声明额外的Maven存储库。

如果您使用的是SNAPSHOT版本，则需要确保您已经定义了Spring Snapshot存储库，如下所示：

pom.xml中。

```
<repositories>
<!-- ... possibly other repository elements ... -->
<repository>
  <id>spring-snapshot</id>
  <name>Spring Snapshot Repository</name>
  <url>http://repo.spring.io/snapshot</url>
</repository>
</repositories>
```

如果您正在使用里程碑或候选版本，则需要确保您已经定义了Spring Milestone存储库，如下所示：

pom.xml中。

```
<repositories>
<!-- ... possibly other repository elements ... -->
<repository>
  <id>spring-milestone</id>
  <name>Spring Milestone Repository</name>
  <url>http://repo.spring.io/milestone</url>
</repository>
</repositories>
```

Spring Framework Bom 译: Spring框架

Spring Security针对Spring Framework 5.0.6.RELEASE构建，但应该与4.0.x一起使用。许多用户会遇到的问题是，Spring Security的传递依赖性解决了Spring Framework 5.0.6.RELEASE，它可能会导致奇怪的类路径问题。

解决这个问题（单调乏味的）方法是将所有Spring框架模块包含在你的pom的<dependencyManagement>部分。另一种方法是包括 `spring-framework-bom` 您的内<dependencyManagement>您的部分 `pom.xml`，如下所示：

pom.xml中。

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-framework-bom</artifactId>
      <version>5.0.6.RELEASE</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

这将确保Spring Security的所有传递依赖使用Spring 5.0.6.RELEASE模块。



这种方法使用了Maven的“材料清单”（BOM）概念，并且仅在Maven 2.0.9+中可用。有关依赖关系如何解决的更多详细信息，请参阅[Maven's Introduction to the Dependency Mechanism documentation](#)。

2.4.2 Gradle 译: 242 Gradle

最小的Spring Security Gradle依赖关系集通常如下所示：

的**build.gradle**。

```
dependencies {
  compile 'org.springframework.security:spring-security-web:5.1.0.M1'
  compile 'org.springframework.security:spring-security-config:5.1.0.M1'
}
```

如果您正在使用LDAP，OpenID等附加功能，则还需要包含相应的 [Section 2.4.3, "Project Modules"](#)。

Gradle Repositories 译: Gradle仓库

所有GA版本（即以.RELEASE结尾的版本）都部署到Maven Central，因此使用mavenCentral（）存储库就足够用于GA版本。

的build.gradle。

```
repositories {
    mavenCentral()
}
```

如果您使用的是SNAPSHOT版本，则需要确保您已经定义了Spring Snapshot存储库，如下所示：

的build.gradle。

```
repositories {
    maven { url 'https://repo.spring.io/snapshot' }
}
```

如果您正在使用里程碑或候选版本，则需要确保您已经定义了Spring Milestone存储库，如下所示：

的build.gradle。

```
repositories {
    maven { url 'https://repo.spring.io/milestone' }
}
```

Using Spring 4.0.x and Gradle 译：使用 Spring 4.0.x 和 Gradle

默认情况下，Gradle将在解析传递版本时使用最新版本。这意味着在Spring Framework 5.0.6.RELEASE中运行Spring Security 5.1.0.M1时通常不需要额外的工作。但是，有时可能会出现错误，因此最好使用[Gradle's ResolutionStrategy](#)来缓解此[问题](#)，如下所示：

的build.gradle。

```
configurations.all {
    resolutionStrategy.eachDependency { DependencyResolveDetails details ->
        if (details.requested.group == 'org.springframework') {
            details.useVersion '5.0.6.RELEASE'
        }
    }
}
```

这将确保Spring Security的所有传递依赖使用Spring 5.0.6.RELEASE模块。



这个例子使用了Gradle 1.9，但可能需要修改才能在Gradle的未来版本中工作，因为这是Gradle中的一项孵化功能。

2.4.3 Project Modules 译：2.4.3 项目模块

在Spring Security 3.0中，代码库被细分为独立的jar，这些jar更清楚地区分了不同的功能区域和第三方依赖关系。如果您使用Maven构建项目，那么这些是您将添加到pom.xml的模块。即使您没有使用Maven，我们也建议您参考pom.xml文件以了解第三方依赖关系和版本。或者，一个好主意是检查示例应用程序中包含的库。

Core - spring-security-core.jar 译：核心 - spring-security-core.jar

包含核心认证和访问控制类和接口，远程处理支持和基本配置API。由使用Spring Security的任何应用程序所要求。支持独立应用程序，远程客户端，方法（服务层）安全性和JDBC用户配置。包含顶级包：

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

Remoting - spring-security-remoting.jar 译：Remoting - spring-security-remoting.jar

提供与Spring Remoting的集成。除非你正在编写一个使用Spring Remoting的远程客户端，否则你不需要这个。主包是org.springframework.security.remoting。

Web - spring-security-web.jar 译：Web - spring-security-web.jar

包含过滤器和相关的网络安全基础架构代码。任何具有servlet API依赖性的东西。如果您需要Spring Security Web认证服务和基于URL的访问控制，您将需要它。主包是org.springframework.security.web。

Config - spring-security-config.jar 译：配置 - spring-security-config.jar

包含安全名称空间解析代码和Java配置代码。如果您使用Spring Security XML名称空间进行配置或Spring Security的Java配置支持，则需要它。主包是org.springframework.security.config。这些类别都不能直接用于应用程序。

LDAP - spring-security-ldap.jar 译：LDAP - spring-security-ldap.jar

LDAP认证和供应代码。如果您需要使用LDAP身份验证或管理LDAP用户条目，则为必填项。顶层包是org.springframework.security.ldap。

OAuth 2.0 Core - spring-security-oauth2-core.jar 译：OAuth 2.0核心 - spring-security-oauth2-core.jar

spring-security-oauth2-core.jar包含为OAuth 2.0授权框架和OpenID Connect Core 1.0提供支持的核心类和接口。使用OAuth 2.0或OpenID Connect Core 1.0的应用程序（例如客户端，资源服务器和授权服务器）需要此功能。顶级套餐是org.springframework.security.oauth2.core。

OAuth 2.0 Client - spring-security-oauth2-client.jar 译：OAuth 2.0客户端 - spring-security-oauth2-client.jar

spring-security-oauth2-client.jar是Spring Security对OAuth 2.0授权框架和OpenID Connect Core 1.0的客户端支持。应用程序需要使用OAuth 2.0登录和/或OAuth客户端支持。顶层包是org.springframework.security.oauth2.client。

OAuth 2.0 JOSE - spring-security-oauth2-jose.jar 译：OAuth 2.0 JOSE - spring-security-oauth2-jose.jar

`spring-security-oauth2-jose.jar` 包含Spring Security对JOSE（Javascript对象签名和加密）框架的支持。JOSE框架旨在提供一种安全地在各方之间转让请求的方法。它由一系列规格构建而成：

- JSON Web Token (JWT)
- JSON Web Signature (JWS)
- JSON Web Encryption (JWE)
- JSON Web Key (JWK)

它包含顶级软件包：

- `org.springframework.security.oauth2.jwt`
- `org.springframework.security.oauth2.jose`

ACL - spring-security-acl.jar 译：ACL - spring-security-acl.jar

专门的域对象ACL实现。用于将安全性应用于应用程序内的特定域对象实例。顶级包是 `org.springframework.security.acls`。

CAS - spring-security-cas.jar 译：CAS - spring-security-cas.jar

Spring Security的CAS客户端集成。如果您想使用CAS单点登录服务器的Spring Security Web认证。顶层包是 `org.springframework.security.cas`。

OpenID - spring-security-openid.jar 译：OpenID - spring-security-openid.jar

OpenID Web认证支持。用于对外部OpenID服务器进行身份验证。`org.springframework.security.openid`。需要OpenID4Java。

Test - spring-security-test.jar 译：测试 - spring-security-test.jar

支持使用Spring Security进行测试。

2.4.4 Checking out the Source 译：2.4.4 检出信号源

由于Spring Security是一个开源项目，我们强烈建议您使用git检查源代码。这将使您可以完全访问所有示例应用程序，并且可以轻松构建项目的最新版本。拥有项目的源代码对于调试也有很大的帮助。异常堆栈跟踪不再是晦涩难解的黑盒问题，但是您可以直接找到导致问题并找出问题的线路。源代码是项目的最终文档，通常是查找实际内容的最简单方式。

要获得项目的源代码，请使用以下git命令：

```
git clone https://github.com/spring-projects/spring-security.git
```

这将使您可以访问本地计算机上的整个项目历史记录（包括所有版本和分支机构）。

3. What's New in Spring Security 5.1 译：3.1 什么是Spring Security 5.1的新功能

Spring Security 5.1提供了许多新功能。以下是该版本的亮点。

3.1 New Features 译：3.1 新功能

- [Chapter 11, Testing Method Security](#)
 - Support for customizing when the `SecurityContext` is setup in the test For example, `@WithMockUser(setupBefore = TestExecutionEvent.TEST_EXECUTION)` will setup a user after JUnit's `@Before` and before the test executes.
 - `@WithUserDetails` now works with `ReactiveUserDetailsService`
- [Section 10.4, "Jackson Support"](#) - added support for `BadCredentialsException`
- [Section 39.3, "@AuthenticationPrincipal"](#)
 - Supports resolving beans in WebFlux (was already supported in Spring MVC)
 - Supports resolving `errorOnInvalidType` in WebFlux (was already supported in Spring MVC)

4. Samples and Guides (Start Here) 译：4 样品和指南（从这里开始）

如果您想要开始使用Spring Security，最好的地方就是我们的示例应用程序。

表4.1. 示例应用程序

Source	描述	Guide
Hello Spring Security	演示如何将Spring Security与现有的使用基于Java的配置的应用程序集成。	Hello Spring Security Guide
Hello Spring Security Boot	演示如何将Spring Security与现有的Spring Boot应用程序集成。	Hello Spring Security Boot Guide
Hello Spring Security XML	演示如何将Spring Security与使用基于XML的配置的现有应用程序集成。	Hello Spring Security XML Guide
Hello Spring MVC Security	演示如何将Spring Security与现有的Spring MVC应用程序集成。	Hello Spring MVC Security Guide
Custom Login Form	演示如何创建自定义登录表单。	Custom Login Form Guide
OAuth 2.0 Login	演示如何将OAuth 2.0登录与OAuth 2.0或OpenID Connect 1.0 Provider集成。	OAuth 2.0 Login Guide

5. Java Configuration 译：5 Java配置

在Spring 3.1中为Spring Framework添加了对[Java Configuration](#)的一般支持。自Spring Security 3.2以来，已经有了Spring Security Java Configuration支持，使用户无需使用任何XML即可轻松配置Spring Security。

如果您熟悉 [Chapter 6, Security Namespace Configuration](#)，那么您应该会发现它与Security Java Configuration支持之间的相似之处。



Spring Security提供了 [lots of sample applications](#)，演示了如何使用Spring Security Java Configuration。

5.1 Hello Web Security Java Configuration 译: 5.1 Hello Web安全Java配置

第一步是创建我们的Spring Security Java配置。该配置会创建一个名为 `springSecurityFilterChain` 的Servlet过滤器，它负责应用程序中的所有安全性（保护应用程序URL，验证提交的用户名和密码，重定向到登录表单等）。您可以在下面找到Spring Security Java Configuration的最基本的例子：

```
import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.context.annotation.*;
import org.springframework.security.config.annotation.authentication.builders.*;
import org.springframework.security.config.annotation.web.configuration.*;

@EnableWebSecurity
public class WebSecurityConfig implements WebMvcConfigurer {

    @Bean
    public UserDetailsService userDetailsService() throws Exception {
        InMemoryUserDetailsManager manager = new InMemoryUserDetailsManager();
        manager.createUser(User.withDefaultPasswordEncoder().username("user").password("password").roles("USER").build());
        return manager;
    }
}
```

这种配置真的没有多大意义，但它确实有很多。您可以在下面找到以下功能的摘要：

- Require authentication to every URL in your application
- Generate a login form for you
- Allow the user with the **Username** *user* and the **Password** *password* to authenticate with form based authentication
- Allow the user to logout
- [CSRF attack](#) prevention
- [Session Fixation](#) protection
- 安全头集成
 - [HTTP Strict Transport Security](#) for secure requests
 - [X-Content-Type-Options](#) integration
 - Cache Control (can be overridden later by your application to allow caching of your static resources)
 - [X-XSS-Protection](#) integration
 - X-Frame-Options integration to help prevent [Clickjacking](#)
- 与以下Servlet API方法集成
 - [HttpServletRequest#getRemoteUser\(\)](#)
 - [HttpServletRequest.html#getUserPrincipal\(\)](#)
 - [HttpServletRequest.html#isUserInRole\(java.lang.String\)](#)
 - [HttpServletRequest.html#login\(java.lang.String, java.lang.String\)](#)
 - [HttpServletRequest.html#logout\(\)](#)

5.1.1 AbstractSecurityWebApplicationInitializer 译: 5.1.1 AbstractSecurityWebApplicationInitializer

下一步是在战争中注册 `springSecurityFilterChain`。这可以通过Servlet 3.0+环境中的 [Spring's WebApplicationInitializer support](#) Java配置完成。不出所料，Spring Security提供了一个基类 `AbstractSecurityWebApplicationInitializer`，它将确保 `springSecurityFilterChain` 获得您的注册。我们使用 `AbstractSecurityWebApplicationInitializer` 的方式取决于我们是否已经使用Spring，或者Spring Security是我们应用程序中唯一的Spring组件。

- [Section 5.1.2, "AbstractSecurityWebApplicationInitializer without Existing Spring"](#) - Use these instructions if you are not using Spring already
- [Section 5.1.3, "AbstractSecurityWebApplicationInitializer with Spring MVC"](#) - Use these instructions if you are already using Spring

5.1.2 AbstractSecurityWebApplicationInitializer without Existing Spring 译: 5.1.2 AbstractSecurityWebApplicationInitializer不存在Spring

如果您不使用Spring或Spring MVC，则需要将 `WebSecurityConfig` 传入超类，以确保获取配置。您可以在下面找到一个例子：

```
import org.springframework.security.web.context.*;

public class SecurityWebApplicationInitializer
    extends AbstractSecurityWebApplicationInitializer {

    public SecurityWebApplicationInitializer() {
        super(WebSecurityConfig.class);
    }
}
```

`SecurityWebApplicationInitializer` 将执行以下操作：

- Automatically register the `springSecurityFilterChain` Filter for every URL in your application
- Add a `ContextLoaderListener` that loads the [WebSecurityConfig](#).

5.1.3 AbstractSecurityWebApplicationInitializer with Spring MVC 译: 5.1.3使用Spring MVC的AbstractSecurityWebApplicationInitializer

如果我们在应用程序的其他地方使用Spring，我们可能已经有了一个加载我们的Spring配置的 `WebApplicationInitializer`。如果我们使用以前的配置，我们会得到一个错误。相反，我们应该在现有的 `ApplicationContext` 注册Spring Security。例如，如果我们使用Spring MVC，我们的 `SecurityWebApplicationInitializer` 将如下所示：


```
import org.springframework.security.web.context.*;

public class SecurityWebApplicationInitializer
    extends AbstractSecurityWebApplicationInitializer {

}

}
```

这只会为应用程序中的每个URL注册springSecurityFilterChain过滤器。之后，我们将确保WebSecurityConfig已加载到我们现有的ApplicationInitializer中。例如，如果我们使用Spring MVC，它将被添加到getRootConfigClasses()

```
public class MvcWebApplicationInitializer extends
    AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected Class<?>[] getRootConfigClasses() {
        return new Class[] { WebSecurityConfig.class };
    }

    // ... other overrides ...
}
```

5.2 HttpSecurity 译: 5.2 HttpSecurity

迄今为止，我们的WebSecurityConfig仅包含有关如何验证用户的信息。Spring Security如何知道我们要求所有用户进行身份验证？Spring Security如何知道我们想要支持基于表单的身份验证？原因是WebSecurityConfigurerAdapter在configure(HttpSecurity http)方法中提供了默认配置，如下所示：

```
protected void configure(HttpSecurity http) throws Exception {
    http
        .authorizeRequests()
        .anyRequest().authenticated()
        .and()
        .formLogin()
        .and()
        .httpBasic();
}
```

上面的默认配置：

- Ensures that any request to our application requires the user to be authenticated
- Allows users to authenticate with form based login
- Allows users to authenticate with HTTP Basic authentication

您会注意到这个配置与XML命名空间配置非常相似：

```
<http>
  <intercept-url pattern="/" access="authenticated"/>
  <form-login />
  <http-basic />
</http>
```

关闭XML标签的Java配置等价物使用and()方法表示，该方法允许我们继续配置父项。如果你阅读代码，这也是有道理的。我想配置授权请求并配置表单登录并配置HTTP基本认证。

5.3 Java Configuration and Form Login 译: 5.3 Java配置和表单登录

由于我们没有提及任何HTML文件或JSP，因此您可能想知道登录表单从何时被提示登录。由于Spring Security的默认配置没有明确设置登录页面的URL，因此Spring Security会根据启用的功能自动生成一个URL，并使用处理提交的登录的URL的标准值，默认目标URL为用户将在登录后被发送到等等。

尽管自动生成的登录页面很方便快速启动和运行，但大多数应用程序都希望提供自己的登录页面。为此，我们可以更新我们的配置，如下所示：

```
protected void configure(HttpSecurity http) throws Exception {
    http
        .authorizeRequests()
        .anyRequest().authenticated()
        .and()
        .formLogin()
        .loginPage("/login") ❶
        .permitAll();        ❷
}
```

❶ 更新的配置指定登录页面的位置。

❷ 我们必须授予所有用户（即未经身份验证的用户）访问我们的登录页面。formLogin().permitAll()方法允许向所有用户授予与基于表单的登录相关的所有URL的访问权限。

下面是一个使用JSP实现的用于当前配置的登录页面示例：



下面的登录页面代表我们当前的配置。如果某些默认设置不能满足我们的需求，我们可以轻松更新我们的配置。

```

<c:url value="/login" var="loginUrl"/>
<form action="${loginUrl}" method="post">
  <c:if test="${param.error != null}">
    <p>
      Invalid username and password.
    </p>
  </c:if>
  <c:if test="${param.logout != null}">
    <p>
      You have been logged out.
    </p>
  </c:if>
  <p>
    <label for="username">Username</label>
    <input type="text" id="username" name="username"/>
  </p>
  <p>
    <label for="password">Password</label>
    <input type="password" id="password" name="password"/>
  </p>
  <input type="hidden"
    name="${_csrf.parameterName}"
    value="${_csrf.token}"/>
  <button type="submit" class="btn">Log in</button>
</form>

```

- ❶ `/login` URL的POST将尝试对用户进行身份验证
- ❷ 如果查询参数 `error` 存在，则认证尝试失败
- ❸ 如果查询参数 `logout` 存在，则用户已成功注销
- ❹ 用户名必须作为名为 `username`的HTTP参数存在
- ❺ 密码必须作为名为 `password`的HTTP参数存在
- ❻ 我们必须 [Section 19.4.3, "Include the CSRF Token"](#)要了解更多信息，请阅读参考文献的 [Chapter 19, Cross Site Request Forgery \(CSRF\)](#)部分

5.4 Authorize Requests 译：5.4授权请求

我们的示例只需要用户进行身份验证，并已为我们的应用程序中的每个URL完成此操作。我们可以通过向我们的 `http.authorizeRequests()` 方法添加多个子项来为我们的URL指定自定义要求。例如：

```

protected void configure(HttpSecurity http) throws Exception {
    http
        .authorizeRequests()
            .antMatchers("/resources/**", "/signup", "/about").permitAll()
            .antMatchers("/admin/**").hasRole("ADMIN")
            .antMatchers("/db/**").access("hasRole('ADMIN') and hasRole('DBA')")
            .anyRequest().authenticated()
        .and()
        // ...
        .formLogin();
}

```

- ❶ 有 `http.authorizeRequests()` 方法有多个孩子，每个匹配器按照他们声明的顺序考虑。
- ❷ 我们指定了任何用户都可以访问的多个网址格式。具体来说，如果URL以 `/resources/` 开头，等于 `/signup` 或等于 `/about`，则任何用户都可以访问请求。
- ❸ 任何以 `/admin/` 开头的网址都将限制为具有“ROLE_ADMIN”角色的用户。你会注意到，因为我们调用了 `hasRole` 方法，所以我们不需要指定“ROLE_”前缀。
- ❹ 任何以 `/db/` 开头的URL都需要用户同时拥有“ROLE_ADMIN”和“ROLE_DBA”。您会注意到，因为我们使用的是 `hasRole` 表达式，所以我们不需要指定“ROLE_”前缀。
- ❺ 任何尚未匹配的URL只需要用户进行身份验证

5.5 Handling Logouts 译：5.5处理注销

使用 `WebSecurityConfigurerAdapter`，会自动应用注销功能。默认情况下，访问URL `/logout` 将通过以下方式登录用户：

- Invalidating the HTTP Session
- Cleaning up any RememberMe authentication that was configured
- Clearing the `SecurityContextHolder`
- Redirect to `/login?logout`

但是，类似于配置登录功能，您还可以有多种选项来进一步自定义注销要求：

```

protected void configure(HttpSecurity http) throws Exception {
    http
        .logout()
            .logoutUrl("/my/logout")
            .logoutSuccessUrl("/my/index")
            .logoutSuccessHandler(logoutSuccessHandler)
            .invalidateHttpSession(true)
            .addLogoutHandler(logoutHandler)
            .deleteCookies(cookieNamesToClear)
        .and()
        ...
}

```

- ❶ 提供注销支持。这在使用 `WebSecurityConfigurerAdapter` 时会自动应用。
- ❷ 触发注销的URL（默认值为 `/logout`）。如果启用CSRF保护（默认），则该请求也必须是POST。欲了解更多信息，请咨询[JavaDoc](#)。
- ❸ 注销后重定向到的URL。默认值是 `/login?logout`。欲了解更多信息，请咨询[JavaDoc](#)。
- ❹ 让我们指定一个自定义 `LogoutSuccessHandler`。如果指定，则 `logoutSuccessUrl()` 被忽略。欲了解更多信息，请咨询[JavaDoc](#)。
- ❺ 指定在注销时是否使 `HttpSession` 失效。这是默认为 `true`。配置下面的 `SecurityContextLogoutHandler`。欲了解更多信息，请咨询[JavaDoc](#)。
- ❻ 添加一个 `LogoutHandler`。`SecurityContextLogoutHandler` 默认添加为最后的 `LogoutHandler`。
- ❼ 允许指定要在注销成功时删除的cookie的名称。这是明确添加 `CookieClearingLogoutHandler` 的捷径。



注销当然也可以使用XML名称空间表示法进行配置。有关更多详细信息，请参阅Spring Security XML命名空间部分中的[logout element](#)文档。

通常，为了自定义注销功能，您可以添加 `LogoutHandler` 和/或 `LogoutSuccessHandler` 实现。对于很多常见的场景，这些处理程序在使用流利API时会在封面下应用。

5.5.1 LogoutHandler 译：5.5.1 LogoutHandler

通常，`LogoutHandler` 实现指示能够参与注销处理的类。预计它们将被调用来执行必要的清理。因此，他们不应该抛出异常。提供了各种实现：

- [PersistentTokenBasedRememberMeServices](#)
- [TokenBasedRememberMeServices](#)
- [CookieClearingLogoutHandler](#)
- [CsrfLogoutHandler](#)
- [SecurityContextLogoutHandler](#)

详情请参阅 [Section 18.4, "Remember-Me Interfaces and Implementations"](#)。

Fluent API不是直接提供 `LogoutHandler` 实现，而是提供了快捷方式，它们提供了相应的 `LogoutHandler` 实现。例如，`deleteCookies()` 允许指定在注销成功时删除一个或多个cookie的名称。与添加 `CookieClearingLogoutHandler` 相比，这是一条捷径。

5.5.2 LogoutSuccessHandler 译：5.5.2 LogoutSuccessHandler

`LogoutSuccessHandler` 在 `LogoutFilter` 成功注销后被 `LogoutFilter`，以处理重定向或转发到适当的目的地。请注意，界面与 `LogoutHandler` 几乎相同，但可能引发异常。

提供了以下实现：

- [SimpleUrlLogoutSuccessHandler](#)
- [HttpStatusReturningLogoutSuccessHandler](#)

如上所述，您不需要直接指定 `SimpleUrlLogoutSuccessHandler`。相反，流畅的API通过设置 `logoutSuccessUrl()` 提供了一个快捷方式。这将设置 `SimpleUrlLogoutSuccessHandler` 的封面。提供的URL将在注销发生后重定向到。默认值是 `/login?logout`。

在REST API类型场景中，`HttpStatusReturningLogoutSuccessHandler` 可能很有趣。而不是在成功注销后重定向到URL，这 `LogoutSuccessHandler` 允许您提供一个普通的HTTP状态代码来返回。如果未配置，则缺省情况下将返回状态代码200。

5.5.3 Further Logout-Related References 译：5.5.3更多注销相关参考

- [Logout Handling](#)
- [Testing Logout](#)
- [HttpServletRequest.logout\(\)](#)
- [Section 18.4, "Remember-Me Interfaces and Implementations"](#)
- [Logging Out](#) in section [CSRF Caveats](#)
- [Section Single Logout](#) (CAS protocol)
- [Documentation for the logout element](#) in the [Spring Security XML Namespace section](#)

5.6 WebFlux Security 译：5.6 WebFlux安全

Spring Security的WebFlux支持依赖于 `WebFilter` 并且对于Spring WebFlux和Spring WebFlux.Fn的作用相同。你可以找到几个示例应用程序来演示下面的代码：

- Hello WebFlux [hellowebflux](#)
- Hello WebFlux.Fn [hellowebfluxfn](#)
- Hello WebFlux Method [hellowebflux-method](#)

5.6.1 Minimal WebFlux Security Configuration 译：5.6.1最小WebFlux安全配置

您可以在下面找到最低限度的WebFlux安全配置：

```
@EnableWebFluxSecurity
public class HelloWebFluxSecurityConfig {

    @Bean
    public MapReactiveUserDetailsService userDetailsService() {
        UserDetails user = User.withDefaultPasswordEncoder()
            .username("user")
            .password("user")
            .roles("USER")
            .build();
        return new MapReactiveUserDetailsService(user);
    }
}
```

此配置提供表单和http基本身份验证，设置授权以要求经过身份验证的用户访问任何页面，设置默认登录页面和默认注销页面，设置与安全性相关的HTTP标头，CSRF保护等。

5.6.2 Explicit WebFlux Security Configuration 译：5.6.2显式WebFlux安全配置

您可以在下面找到最小WebFlux安全配置的显式版本：

```
@EnableWebFluxSecurity
public class HelloWebfluxSecurityConfig {

    @Bean
    public MapReactiveUserDetailsService userDetailsService() {
        UserDetails user = User.withDefaultPasswordEncoder()
            .username("user")
            .password("user")
            .roles("USER")
            .build();
        return new MapReactiveUserDetailsService(user);
    }

    @Bean
    public SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {
        http
            .authorizeExchange()
            .anyExchange().authenticated()
            .and()
            .httpBasic().and()
            .formLogin();
        return http.build();
    }
}
```

该配置明确设置了与我们最小配置相同的所有内容。从这里您可以轻松地对默认值进行更改。

5.7 OAuth 2.0 Login 译: 5.7 OAuth 2.0 登录

OAuth 2.0 登录功能为应用程序提供了让用户通过在OAuth 2.0提供程序（例如GitHub）或OpenID Connect 1.0提供程序（例如Google）上使用其现有帐户登录应用程序的功能。OAuth 2.0 Login实现了用例：“使用Google登录”或“使用GitHub登录”。



按照 [OAuth 2.0 Authorization Framework](#)和 [OpenID Connect Core 1.0](#)中的规定，使用 [授权代码授权](#)实现OAuth 2.0登录。

5.7.1 Spring Boot 2.0 Sample 译: 5.7.1 Spring Boot 2.0 示例

Spring Boot 2.0为OAuth 2.0登录带来了全面的自动配置功能。

本节介绍如何使用 [Google](#)将 [OAuth 2.0 Login sample](#)配置为 [身份验证提供程序](#)，并涵盖以下主题：

- [Initial setup](#)
- [Setting the redirect URI](#)
- [Configure `application.yml`](#)
- [Boot up the application](#)

Initial setup 译: 初始设置

要使用Google的OAuth 2.0身份验证系统进行登录，您必须在Google API控制台中设置项目以获取OAuth 2.0凭据。



[Google's OAuth 2.0 implementation](#)进行身份验证符合 [OpenID Connect 1.0](#)规范，并且是 [OpenID Certified](#)。

按照 [OpenID Connect](#)页面上的说明操作，从“设置OAuth 2.0”部分开始。

完成“获取OAuth 2.0证书”说明后，您应该拥有一个全新的OAuth客户端，其凭证由客户端ID和客户端密钥组成。

Setting the redirect URI 译: 设置重定向URI

重定向URI是应用程序中的路径，终端用户的用户代理在与Google进行身份验证并已授予对Consent页面上的OAuth客户端（[created in the previous step](#)）的访问权之后，将其重定向回。

在“设置重定向URI”子部分中，确保 [授权重定向URI](#)字段设置为 <http://localhost:8080/login/oauth2/code/google>。



默认的重定向UR模板是 `{baseUrl}/login/oauth2/code/{registrationId}`。 `registrationId`是[ClientRegistration](#)的唯一标识符。

Configure `application.yml` 译: 配置 `application.yml`

现在，您已经拥有了一个带有Google的新OAuth客户端，您需要配置该应用程序以使用OAuth客户端进行[身份验证流程](#)。要做到这一点：

1. 转到 `application.yml`并设置以下配置：

```
spring:
  security:
    oauth2:
      client:
        registration: ❶
        google: ❷
        client-id: google-client-id
        client-secret: google-client-secret
```

例5.1。OAuth客户端属性

- ❶ `spring.security.oauth2.client.registration`是OAuth客户端属性的基本属性前缀。
- ❷ 基本属性前缀后面是 [ClientRegistration](#)的ID，例如google。

2. Replace the values in the `client-id` and `client-secret` property with the OAuth 2.0 credentials you created earlier.

Boot up the application 译:启动应用程序

启动Spring Boot 2.0示例并转至 `http://localhost:8080` 。然后，您将被重定向到默认的 *自动生成的* 登录页面，该页面会显示Google的链接。

点击Google链接，然后重定向到Google进行身份验证。

在使用Google帐户凭证进行身份验证后，向您呈现的下一页是“同意”屏幕。“同意”屏幕会要求您允许或拒绝访问之前创建的OAuth客户端。点击 **允许** ，授权OAuth客户端访问您的电子邮件地址和基本配置文件信息。

此时，OAuth客户端会从 **UserInfo Endpoint**中检索您的电子邮件地址和基本配置文件信息，并建立经过验证的会话。

5.7.2 ClientRegistration 译:5.7.2客户端注册

ClientRegistration 是向OAuth 2.0或OpenID Connect 1.0 Provider注册的客户的代表。

客户端注册包含客户端ID，客户端秘密，授权授权类型，重定向URI，范围，授权URI，令牌URI和其他详细信息等信息。

ClientRegistration 及其属性定义如下：

```
public final class ClientRegistration {
    private String registrationId; ❶
    private String clientId; ❷
    private String clientSecret; ❸
    private ClientAuthenticationMethod clientAuthenticationMethod; ❹
    private AuthorizationGrantType authorizationGrantType; ❺
    private String redirectUriTemplate; ❻
    private Set<String> scopes; ❼
    private ProviderDetails providerDetails;
    private String clientName; ❽

    public class ProviderDetails {
        private String authorizationUri; ❾
        private String tokenUri; ❿
        private UserInfoEndpoint userInfoEndpoint;
        private String jwkSetUri; ⓫

        public class UserInfoEndpoint {
            private String uri; ⓫
            private String userNameAttributeName; ⓬
        }
    }
}
```

- ❶ **registrationId** : 唯一标识 **ClientRegistration** 的ID。
- ❷ **clientId** : 客户端标识符。
- ❸ **clientSecret** : 客户秘密。
- ❹ **clientAuthenticationMethod** : 用于向提供者验证客户端的方法。支持的值是**基本值**和**后值**。
- ❺ **authorizationGrantType** : OAuth 2.0授权框架定义了四种**Authorization Grant**类型。支持的值是**authorization_code**和**隐式**。
- ❻ **redirectUriTemplate** : 客户端注册的重定向URI，*授权服务器*将最终用户的用户代理重定向到最终用户对客户端进行身份验证和授权访问之后。默认的重定向URI模板是 `{baseUrl}/login/oauth2/code/{registrationId}`，它支持URI模板变量。
- ❼ **scopes** : 授权请求流程期间客户端请求的范围，例如**openid**，电子邮件或配置文件。
- ❽ **clientName** : 用于客户端的描述性名称。该名称可用于某些情况下，例如在自动生成的登录页面中显示客户端的名称时。
- ❾ **authorizationUri** : 授权服务器的授权端点URI。
- ❿ **tokenUri** : 授权服务器的令牌端点URI。
- ⓫ **jwkSetUri** : 用于从授权服务器中检索 **JSON Web Key (JWK)** Set的URI，其中包含用于验证ID令牌 **JSON Web Signature (JWS)**和可选的**UserInfo**响应的加密密钥。
- ⓫ **(userInfoEndpoint)uri** : 用于访问经过身份验证的最终用户的声明/属性的**UserInfo**端点URI。
- ⓬ **userNameAttributeName** : 在**UserInfo**响应中返回的引用最终用户的名称或标识符的属性的名称。

5.7.3 Spring Boot 2.0 Property Mappings 译:5.7.3 Spring Boot 2.0属性映射

下表概述了Spring Boot 2.0 OAuth客户端属性到 **ClientRegistration** 属性的 **ClientRegistration** 。

Spring Boot 2.0	ClientRegistration
<code>spring.security.oauth2.client.registration.[registrationId]</code>	<code>registrationId</code>
<code>spring.security.oauth2.client.registration.[registrationId].client-id</code>	<code>clientId</code>
<code>spring.security.oauth2.client.registration.[registrationId].client-secret</code>	<code>clientSecret</code>
<code>spring.security.oauth2.client.registration.[registrationId].client-authentication-method</code>	<code>clientAuthenticationMethod</code>
<code>spring.security.oauth2.client.registration.[registrationId].authorization-grant-type</code>	<code>authorizationGrantType</code>
<code>spring.security.oauth2.client.registration.[registrationId].redirect-uri-template</code>	<code>redirectUriTemplate</code>
<code>spring.security.oauth2.client.registration.[registrationId].scope</code>	<code>scopes</code>
<code>spring.security.oauth2.client.registration.[registrationId].client-name</code>	<code>clientName</code>
<code>spring.security.oauth2.client.provider.[providerId].authorization-uri</code>	<code>providerDetails.authorizationUri</code>

Spring Boot 2.0	ClientRegistration
<code>spring.security.oauth2.client.provider.[providerId].token-uri</code>	<code>providerDetails.tokenUri</code>
<code>spring.security.oauth2.client.provider.[providerId].jwk-set-uri</code>	<code>providerDetails.jwkSetUri</code>
<code>spring.security.oauth2.client.provider.[providerId].user-info-uri</code>	<code>providerDetails.userInfoEndpoint.uri</code>
<code>spring.security.oauth2.client.provider.[providerId].userNameAttribute</code>	<code>providerDetails.userInfoEndpoint.userNameAttribute</code>

5.7.4 ClientRegistrationRepository #: 5.7.4 ClientRegistrationRepository

`ClientRegistrationRepository` 可用作 OAuth 2.0 / OpenID Connect 1.0 的存储库 `ClientRegistration` (s)。



客户端注册信息最终由关联的授权服务器存储和拥有。该存储库提供了检索与授权服务器一起存储的主要客户端注册信息的子集的功能。

弹簧引导 2.0 自动配置结合各下的特性的 `spring.security.oauth2.client.registration.[registrationId]` 到的一个实例 `ClientRegistration`，然后构成每个的 `ClientRegistration` 一个内实例 (多个) `ClientRegistrationRepository`。



`ClientRegistrationRepository` 的默认实现是 `InMemoryClientRegistrationRepository`。

自动配置还将 `ClientRegistrationRepository` 注册为 `@Bean` 中的 `ApplicationContext` 以便它可用于依赖注入 (如果应用程序需要)。

以下列表显示了一个示例：

```
@Controller
public class OAuth2LoginController {

    @Autowired
    private ClientRegistrationRepository clientRegistrationRepository;

    @RequestMapping("/")
    public String index() {
        ClientRegistration googleRegistration =
            this.clientRegistrationRepository.findByRegistrationId("google");

        ...

        return "index";
    }
}
```

5.7.5 CommonOAuth2Provider #: 5.7.5 CommonOAuth2Provider

`CommonOAuth2Provider` 为许多众所周知的提供者预先定义了一组默认的客户属性：Google, GitHub, Facebook 和 Okta。

例如，`authorization-uri`，`token-uri`，并 `user-info-uri` 不经常对供应商变更。因此，提供默认值以减少所需的配置是有意义的。

如前所述，当我们 `configured a Google client` 时，只需要 `client-id` 和 `client-secret` 属性。

以下列表显示了一个示例：

```
spring:
  security:
    oauth2:
      client:
        registration:
          google:
            client-id: google-client-id
            client-secret: google-client-secret
```



客户端性能的自动违约无缝工作在这里，因为 `registrationId` (`google`) 匹配 `GOOGLE` 枚举 (不区分大小写) `CommonOAuth2Provider`。

对于您可能需要指定其他 `registrationId` (如 `google-login`)，您仍然可以通过配置 `provider` 属性来利用客户端属性的自动默认功能。

以下列表显示了一个示例：

```
spring:
  security:
    oauth2:
      client:
        registration:
          google-login: ❶
          provider: google ❷
          client-id: google-client-id
          client-secret: google-client-secret
```

- ❶ `registrationId` 设置为 `google-login`。
- ❷ `provider` 属性设置为 `google`，该属性将利用 `CommonOAuth2Provider.GOOGLE.getBuilder()` 设置的客户端属性的自动默认 `CommonOAuth2Provider.GOOGLE.getBuilder()`。

5.7.6 Configuring Custom Provider Properties 译: 5.7.6配置自定义提供程序属性

有一些OAuth 2.0提供商支持多租户，这会为每个租户（或子域）生成不同的协议端点。

例如，向Okta注册的OAuth客户端分配给特定的子域，并拥有自己的协议端点。

对于这些情况，Spring Boot 2.0为配置自定义提供程序属性提供以下基本属性：`spring.security.oauth2.client.provider.[providerId]`。

以下列表显示了一个示例：

```
spring:
  security:
    oauth2:
      client:
        registration:
          okta:
            client-id: okta-client-id
            client-secret: okta-client-secret
        provider:
          okta: ❶
            authorization-uri: https://your-subdomain.oktapreview.com/oauth2/v1/authorize
            token-uri: https://your-subdomain.oktapreview.com/oauth2/v1/token
            user-info-uri: https://your-subdomain.oktapreview.com/oauth2/v1/userinfo
            user-name-attribute: sub
            jwk-set-uri: https://your-subdomain.oktapreview.com/oauth2/v1/keys
```

- ❶ 基本属性（`spring.security.oauth2.client.provider.okta`）允许自定义配置协议端点位置。

5.7.7 Overriding Spring Boot 2.0 Auto-configuration 译: 5.7.7覆盖Spring Boot 2.0的自动配置

用于OAuth客户端支持的Spring Boot 2.0自动配置类为 `OAuth2ClientAutoConfiguration`。

它执行以下任务：

- Registers a `ClientRegistrationRepository` `@Bean` composed of `ClientRegistration` (s) from the configured OAuth Client properties.
- Provides a `WebSecurityConfigurerAdapter` `@Configuration` and enables OAuth 2.0 Login through `httpSecurity.oauth2Login()`.

如果您需要根据您的特定要求覆盖自动配置，可以通过以下方式进行：

- Register a `ClientRegistrationRepository` `@Bean`
- Provide a `WebSecurityConfigurerAdapter`
- Completely Override the Auto-configuration

Register a `ClientRegistrationRepository` `@Bean` 译: 注册一个 `ClientRegistrationRepository` `@Bean`

以下示例显示如何注册 `ClientRegistrationRepository` `@Bean`：

```
@Configuration
public class OAuth2LoginConfig {

    @Bean
    public ClientRegistrationRepository clientRegistrationRepository() {
        return new InMemoryClientRegistrationRepository(this.googleClientRegistration());
    }

    private ClientRegistration googleClientRegistration() {
        return ClientRegistration.withRegistrationId("google")
            .clientId("google-client-id")
            .clientSecret("google-client-secret")
            .clientAuthenticationMethod(ClientAuthenticationMethod.BASIC)
            .authorizationGrantType(AuthorizationGrantType.AUTHORIZATION_CODE)
            .redirectUriTemplate("{baseUrl}/login/oauth2/code/{registrationId}")
            .scope("openid", "profile", "email", "address", "phone")
            .authorizationUri("https://accounts.google.com/o/oauth2/v2/auth")
            .tokenUri("https://www.googleapis.com/oauth2/v4/token")
            .userInfoUri("https://www.googleapis.com/oauth2/v3/userinfo")
            .userNameAttributeName(IdTokenClaimNames.SUB)
            .jwkSetUri("https://www.googleapis.com/oauth2/v3/certs")
            .clientName("Google")
            .build();
    }
}
```

Provide a `WebSecurityConfigurerAdapter` 译: 提供 `WebSecurityConfigurerAdapter`

以下示例显示如何通过 `WebSecurityConfigurerAdapter` 提供 `@EnableWebSecurity` 并通过 `httpSecurity.oauth2Login()` 启用OAuth 2.0登录：

```
@EnableWebSecurity
public class OAuth2LoginSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests()
            .anyRequest().authenticated()
            .and()
            .oauth2Login();
    }
}
```

Completely Override the Auto-configuration 译: 完全覆盖自动配置

以下示例显示了如何通过注册 `ClientRegistrationRepository` `@Bean` 并提供 `WebSecurityConfigurerAdapter` 来完全覆盖自动配置，两者均在 `ClientRegistrationRepository` 中进行了介绍。

```
@Configuration
public class OAuth2LoginConfig {

    @EnableWebSecurity
    public static class OAuth2LoginSecurityConfig extends WebSecurityConfigurerAdapter {

        @Override
        protected void configure(HttpSecurity http) throws Exception {
            http
                .authorizeRequests()
                .anyRequest().authenticated()
                .and()
                .oauth2Login();
        }
    }

    @Bean
    public ClientRegistrationRepository clientRegistrationRepository() {
        return new InMemoryClientRegistrationRepository(this.googleClientRegistration());
    }

    private ClientRegistration googleClientRegistration() {
        return ClientRegistration.withRegistrationId("google")
            .clientId("google-client-id")
            .clientSecret("google-client-secret")
            .clientAuthenticationMethod(ClientAuthenticationMethod.BASIC)
            .authorizationGrantType(AuthorizationGrantType.AUTHORIZATION_CODE)
            .redirectUriTemplate("{baseUrl}/login/oauth2/code/{registrationId}")
            .scope("openid", "profile", "email", "address", "phone")
            .authorizationUri("https://accounts.google.com/o/oauth2/v2/auth")
            .tokenUri("https://www.googleapis.com/oauth2/v4/token")
            .userInfoUri("https://www.googleapis.com/oauth2/v3/userinfo")
            .userNameAttributeName(IdTokenClaimNames.SUB)
            .jwkSetUri("https://www.googleapis.com/oauth2/v3/certs")
            .clientName("Google")
            .build();
    }
}
```

5.7.8 Java Configuration without Spring Boot 2.0

如果您无法使用Spring Boot 2.0并且想要配置 `CommonOAuth2Provider` 中的 `CommonOAuth2Provider` 预定义提供程序（例如Google），请应用以下配置：

```
@Configuration
public class OAuth2LoginConfig {

    @EnableWebSecurity
    public static class OAuth2LoginSecurityConfig extends WebSecurityConfigurerAdapter {

        @Override
        protected void configure(HttpSecurity http) throws Exception {
            http
                .authorizeRequests()
                .anyRequest().authenticated()
                .and()
                .oauth2Login();
        }
    }

    @Bean
    public ClientRegistrationRepository clientRegistrationRepository() {
        return new InMemoryClientRegistrationRepository(this.googleClientRegistration());
    }

    @Bean
    public OAuth2AuthorizedClientService authorizedClientService() {
        return new InMemoryOAuth2AuthorizedClientService(this.clientRegistrationRepository());
    }

    private ClientRegistration googleClientRegistration() {
        return CommonOAuth2Provider.GOOGLE.getBuilder("google")
            .clientId("google-client-id")
            .clientSecret("google-client-secret")
            .build();
    }
}
```

5.7.9 OAuth2AuthorizedClient / OAuth2AuthorizedClientService

`OAuth2AuthorizedClient` 是授权客户的代表。当最终用户（资源所有者）授予客户端访问其受保护资源的权限时，客户端被认为是被授权的。

`OAuth2AuthorizedClient` 用于将 `OAuth2AccessToken` 与 `ClientRegistration`（客户端）和资源所有者（即 `Principal` 授予该授权的最终用户）相关联。

`OAuth2AuthorizedClientService` 的主要作用是管理 `OAuth2AuthorizedClient` 实例。从开发人员的角度来看，它提供了查找与客户端关联的 `OAuth2AccessToken` 的功能，以便它可以用来发起对资源服务器的请求。



Spring Boot 2.0自动配置在 `@Bean` 中注册 `OAuth2AuthorizedClientService` `ApplicationContext`。

开发者还可注册一个 `OAuth2AuthorizedClientService` `@Bean` 在 `ApplicationContext` 为了有查找一个的能力（覆盖弹簧引导2.0自动配置）`OAuth2AccessToken` 与特定关联 `ClientRegistration`（客户端）。

以下列表显示了一个示例：

```
@Controller
public class OAuth2LoginController {

    @Autowired
    private OAuth2AuthorizedClientService authorizedClientService;

    @RequestMapping("/userinfo")
    public String userinfo(OAuth2AuthenticationToken authentication) {
        // authentication.getAuthorizedClientRegistrationId() returns the
        // registrationId of the Client that was authorized during the Login flow
        OAuth2AuthorizedClient authorizedClient =
            this.authorizedClientService.loadAuthorizedClient(
                authentication.getAuthorizedClientRegistrationId(),
                authentication.getName());

        OAuth2AccessToken accessToken = authorizedClient.getAccessToken();

        ...

        return "userinfo";
    }
}
```

5.7.10 Additional Resources 译: 5.7.10其他资源

以下其他资源介绍了高级配置选项：

- [OAuth 2.0 Login Page](#)
- 授权端点：
 - [AuthorizationRequestRepository](#)
- [Redirection Endpoint](#)
- 令牌端点：
 - [OAuth2AccessTokenResponseClient](#)
- UserInfo端点：
 - [Mapping User Authorities](#)
 - [Configuring a Custom OAuth2User](#)
 - [OAuth 2.0 UserService](#)
 - [OpenID Connect 1.0 UserService](#)

5.8 Authentication 译: 5.8认证

到目前为止，我们只看到了最基本的认证配置。 让我们来看看几个稍微更高级的配置认证选项。

5.8.1 In-Memory Authentication 译: 5.8.1内存中验证

我们已经看到了为单个用户配置内存认证的例子。 以下是配置多个用户的示例：

```
@Bean
public UserDetailsService userDetailsService() throws Exception {
    // ensure the passwords are encoded properly
    UserBuilder users = User.withDefaultPasswordEncoder();
    InMemoryUserDetailsManager manager = new InMemoryUserDetailsManager();
    manager.createUser(users.username("user").password("password").roles("USER").build());
    manager.createUser(users.username("admin").password("password").roles("USER", "ADMIN").build());
    return manager;
}
```

5.8.2 JDBC Authentication 译: 5.8.2 JDBC认证

您可以找到更新以支持基于JDBC的身份验证。 以下示例假定您已在应用程序中定义了 `DataSource`。 [jdbc-javaconfig](#) 示例提供了使用基于JDBC的身份验证的完整示例。

```
@Autowired
private DataSource dataSource;

@Autowired
public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
    // ensure the passwords are encoded properly
    UserBuilder users = User.withDefaultPasswordEncoder();
    auth
        .jdbcAuthentication()
        .dataSource(dataSource)
        .withDefaultSchema()
        .withUser(users.username("user").password("password").roles("USER"))
        .withUser(users.username("admin").password("password").roles("USER", "ADMIN"));
}
```

5.8.3 LDAP Authentication 译: 5.8.3 LDAP认证

您可以找到更新以支持基于LDAP的身份验证。 [ldap-javaconfig](#) 示例提供了使用基于LDAP的身份验证的完整示例。

```

@Autowired
private DataSource dataSource;

@Autowired
public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
    auth
        .ldapAuthentication()
        .userDnPatterns("uid={0},ou=people")
        .groupSearchBase("ou=groups");
}

```

上面的示例使用以下LDIF和嵌入式Apache DS LDAP实例。

users.ldif.

```

dn: ou=groups,dc=springframework,dc=org
objectclass: top
objectclass: organizationalUnit
ou: groups

dn: ou=people,dc=springframework,dc=org
objectclass: top
objectclass: organizationalUnit
ou: people

dn: uid=admin,ou=people,dc=springframework,dc=org
objectclass: top
objectclass: person
objectclass: organizationalPerson
objectclass: inetOrgPerson
cn: Rod Johnson
sn: Johnson
uid: admin
userPassword: password

dn: uid=user,ou=people,dc=springframework,dc=org
objectclass: top
objectclass: person
objectclass: organizationalPerson
objectclass: inetOrgPerson
cn: Dianne Emu
sn: Emu
uid: user
userPassword: password

dn: cn=user,ou=groups,dc=springframework,dc=org
objectclass: top
objectclass: groupOfNames
cn: user
uniqueMember: uid=admin,ou=people,dc=springframework,dc=org
uniqueMember: uid=user,ou=people,dc=springframework,dc=org

dn: cn=admin,ou=groups,dc=springframework,dc=org
objectclass: top
objectclass: groupOfNames
cn: admin
uniqueMember: uid=admin,ou=people,dc=springframework,dc=org

```

5.8.4 AuthenticationProvider 译: 5.8.4 AuthenticationProvider

您可以通过将自定义 `AuthenticationProvider` 作为一个bean进行定义来定义自定义身份验证。例如，假设 `SpringAuthenticationProvider` 实现了 `AuthenticationProvider`，以下将自定义认证：



这仅在 `AuthenticationManagerBuilder` 尚未填充时才会使用

```

@Bean
public SpringAuthenticationProvider springAuthenticationProvider() {
    return new SpringAuthenticationProvider();
}

```

5.8.5 UserDetailsService 译: 5.8.5 UserDetailsService

您可以通过将自定义 `UserDetailsService` 为bean来定义自定义身份验证。例如，假设 `SpringDataUserDetailsService` 实现了 `UserDetailsService`，以下将自定义认证：



这仅在 `AuthenticationManagerBuilder` 尚未填充 `AuthenticationProviderBean` 定义 `AuthenticationManagerBuilder` 时才使用。

```

@Bean
public SpringDataUserDetailsService springDataUserDetailsService() {
    return new SpringDataUserDetailsService();
}

```

您还可以通过将 `PasswordEncoder` 暴露为bean来定义密码的编码方式。例如，如果您使用bcrypt，则可以添加一个bean定义，如下所示：

```

@Bean
public BCryptPasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder();
}

```

5.9 Multiple HttpSecurity

我们可以配置多个HttpSecurity实例，就像我们可以有多个<http>块一样。关键是多次延长WebSecurityConfigurationAdapter。例如，以下是以/api/开头的具有不同配置的URL的/api/。

```

@EnableWebSecurity
public class MultiHttpSecurityConfig {
    @Bean
    public UserDetailsService userDetailsService() throws Exception {
        // ensure the passwords are encoded properly
        UserBuilder users = User.withDefaultPasswordEncoder();
        InMemoryUserDetailsManager manager = new InMemoryUserDetailsManager();
        manager.createUser(users.username("user").password("password").roles("USER").build());
        manager.createUser(users.username("admin").password("password").roles("USER", "ADMIN").build());
        return manager;
    }

    @Configuration
    @Order(1)
    public static class ApiWebSecurityConfigurationAdapter extends WebSecurityConfigurerAdapter {
        protected void configure(HttpSecurity http) throws Exception {
            http
                .antMatcher("/api/**")
                .authorizeRequests()
                .anyRequest().hasRole("ADMIN")
                .and()
                .httpBasic();
        }
    }

    @Configuration
    public static class FormLoginWebSecurityConfigurerAdapter extends WebSecurityConfigurerAdapter {

        @Override
        protected void configure(HttpSecurity http) throws Exception {
            http
                .authorizeRequests()
                .anyRequest().authenticated()
                .and()
                .formLogin();
        }
    }
}

```

- 1 配置身份验证正常
- 2 创建 WebSecurityConfigurerAdapter 的实例，其中包含 @Order 以指定应首先考虑哪个 WebSecurityConfigurerAdapter。
- 3 http.antMatcher 指出此 HttpSecurity 仅适用于以 /api/ 开头的网址
- 4 创建 WebSecurityConfigurerAdapter 另一个实例。如果URL不以 /api/ 开头， /api/ 此配置。这种配置后认为 ApiWebSecurityConfigurationAdapter，因为它具有一个 @Order 值之后 1（无 @Order 默认为最后）。

5.10 Method Security

从2.0版本开始，Spring Security已经大大改善了对服务层方法的安全性的支持。它提供对JSR-250注释安全性的支持以及框架的原始@Secured注释。从3.0开始，您还可以使用新的expression-based annotations。您可以将安全性应用于单个bean，使用intercept-methods元素来装饰bean声明，或者可以使用AspectJ样式切入点在整个服务层中保护多个bean。

5.10.1 EnableGlobalMethodSecurity

我们可以在任何@Configuration实例上使用@EnableGlobalMethodSecurity注释来启用基于注释的安全性。例如，以下将启用Spring Security的@Secured注释。

```

@EnableGlobalMethodSecurity(securedEnabled = true)
public class MethodSecurityConfig {
    // ...
}

```

然后向方法（在类或接口上）添加注释将相应地限制对该方法的访问。Spring Security的本地注释支持为该方法定义了一组属性。这些将被传递给AccessDecisionManager以供它作出实际的决定：

```

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);
}

```

可以使用支持JSR-250注释

```
@EnableGlobalMethodSecurity(jsr250Enabled = true)
public class MethodSecurityConfig {
// ...
}
```

这些是基于标准的，允许应用简单的基于角色的约束，但是没有Spring Security的本地注释的强大功能。要使用新的基于表达式的语法，您可以使用

```
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class MethodSecurityConfig {
// ...
}
```

和等效的Java代码将会是

```
public interface BankService {

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

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

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

5.10.2 GlobalMethodSecurityConfiguration 译：5.10.2 GlobalMethodSecurityConfiguration

有时您可能需要执行比 `@EnableGlobalMethodSecurity` 批注允许可能更复杂的操作。对于这些实例，可以扩展 `GlobalMethodSecurityConfiguration`，以确保 `@EnableGlobalMethodSecurity` 注释存在于您的子类中。例如，如果您想提供自定义 `MethodSecurityExpressionHandler`，则可以使用以下配置：

```
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class MethodSecurityConfig extends GlobalMethodSecurityConfiguration {
@Override
protected MethodSecurityExpressionHandler createExpressionHandler() {
// ... create and return custom MethodSecurityExpressionHandler ...
return expressionHandler;
}
}
```

有关可覆盖的方法的其他信息，请参阅 `GlobalMethodSecurityConfiguration` Javadoc。

5.10.3 EnableReactiveMethodSecurity 译：5.10.3 EnableReactiveMethodSecurity

Spring Security的支持使用方法，安全 `Reactor's Context` 使用正在安装 `ReactiveSecurityContextHolder`。例如，这将演示如何检索当前登录的用户消息。



对于这项工作的方法的返回类型必须是 `org.reactivestreams.Publisher`（即 `Mono` / `Flux`）。这是与Reactor的 `Context` 集成所必需的。

```
Authentication authentication = new TestingAuthenticationToken("user", "password", "ROLE_USER");

Mono<String> messageByUsername = ReactiveSecurityContextHolder.getContext()
    .map(SecurityContext::getAuthentication)
    .map(Authentication::getName)
    .flatMap(this::findMessageByUsername)
// In a WebFlux application the `subscriberContext` is automatically setup using `ReactorContextWebFilter`
    .subscriberContext(ReactiveSecurityContextHolder.withAuthentication(authentication));

StepVerifier.create(messageByUsername)
    .expectNext("Hi user")
    .verifyComplete();
```

与 `this::findMessageByUsername` 定义为：

```
Mono<String> findMessageByUsername(String username) {
return Mono.just("Hi " + username);
}
```

以下是在反应式应用程序中使用方法安全性时的最小方法安全配置。

```
@EnableReactiveMethodSecurity
public class SecurityConfig {
@Bean
public MapReactiveUserDetailsService userDetailsService() {
User.UserBuilder userBuilder = User.withDefaultPasswordEncoder();
UserDetails rob = userBuilder.username("rob").password("rob").roles("USER").build();
UserDetails admin = userBuilder.username("admin").password("admin").roles("USER", "ADMIN").build();
return new MapReactiveUserDetailsService(rob, admin);
}
}
```

考虑以下课程：

```

@Component
public class HelloWorldMessageService {
    @PreAuthorize("hasRole('ADMIN')")
    public Mono<String> findMessage() {
        return Mono.just("Hello World!");
    }
}

```

结合上面的配置，`@PreAuthorize("hasRole('ADMIN')")` 将确保 `findMessage` 仅由具有角色 `ADMIN` 的用户调用。重要的是要注意 `@EnableReactiveMethodSecurity` 标准方法安全性中的任何表达式。不过，此时我们只支持返回类型 `Boolean` 或 `boolean` 的表达式。这意味着表达式不能阻止。

当与 [Section 5.6, "WebFlux Security"](#) 集成时，Spring Security 根据经过认证的用户自动建立 Reactor Context。

```

@EnableWebFluxSecurity
@EnableReactiveMethodSecurity
public class SecurityConfig {

    @Bean
    SecurityWebFilterChain springWebFilterChain(ServerHttpSecurity http) throws Exception {
        return http
            // Demonstrate that method security works
            // Best practice to use both for defense in depth
            .authorizeExchange()
            .anyExchange().permitAll()
            .and()
            .httpBasic().and()
            .build();
    }

    @Bean
    MapReactiveUserDetailsService userDetailsService() {
        User.UserBuilder userBuilder = User.withDefaultPasswordEncoder();
        UserDetails rob = userBuilder.username("rob").password("rob").roles("USER").build();
        UserDetails admin = userBuilder.username("admin").password("admin").roles("USER", "ADMIN").build();
        return new MapReactiveUserDetailsService(rob, admin);
    }
}

```

您可以在 [helloworldwebflux-method](#) 中找到完整的示例

5.11 Post Processing Configured Objects 译: 5.11 后处理配置的对象

Spring Security 的 Java 配置不公开其配置的每个对象的每个属性。这简化了大多数用户的配置。最后，如果每个属性都暴露出来，用户可以使用标准的 bean 配置。

尽管没有直接公开每个属性的很好理由，但用户可能仍然需要更高级的配置选项。为了解决这个问题，Spring Security 引入了 `ObjectPostProcessor` 的概念，它可以用来修改或替换由 Java 配置创建的许多对象实例。例如，如果您想要配置 `filterSecurityPublishAuthorizationSuccess` 上的 `filterSecurityPublishAuthorizationSuccess` 属性，`FilterSecurityInterceptor` 可以使用以下内容：

```

@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        .authorizeRequests()
        .anyRequest().authenticated()
        .withObjectPostProcessor(new ObjectPostProcessor<FilterSecurityInterceptor>() {
            public <O extends FilterSecurityInterceptor> O postProcess(
                O fsi) {
                fsi.setPublishAuthorizationSuccess(true);
                return fsi;
            }
        });
}

```

5.12 Custom DSLs 译: 5.12 自定义 DSL

您可以在 Spring Security 中提供您自己的定制 DSL。例如，您可能有这样的东西：

```

public class MyCustomDsl extends AbstractHttpConfigurer<MyCustomDsl, HttpSecurity> {
    private boolean flag;

    @Override
    public void init(H http) throws Exception {
        // any method that adds another configurer
        // must be done in the init method
        http.csrf().disable();
    }

    @Override
    public void configure(H http) throws Exception {
        ApplicationContext context = http.getSharedObject(ApplicationContext.class);

        // here we lookup from the ApplicationContext. You can also just create a new instance.
        MyFilter myFilter = context.getBean(MyFilter.class);
        myFilter.setFlag(flag);
        http.addFilterBefore(myFilter, UsernamePasswordAuthenticationFilter.class);
    }

    public MyCustomDsl flag(boolean value) {
        this.flag = value;
        return this;
    }

    public static MyCustomDsl customDsl() {
        return new MyCustomDsl();
    }
}

```



这实际上是如何实现像 `HttpSecurity.authorizeRequests()` 这样的方法。

自定义DSL可以像这样使用：

```

@EnableWebSecurity
public class Config extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .apply(customDsl())
            .flag(true)
            .and()
            ...;
    }
}

```

代码按以下顺序调用：

- Code in `Config`'s configure method is invoked
- Code in `MyCustomDsl`'s init method is invoked
- Code in `MyCustomDsl`'s configure method is invoked

如果你愿意，你可以有 `WebSecurityConfigurerAdapter` 添加 `MyCustomDsl` 默认使用 `SpringFactories`。例如，您可以在名为 `META-INF/spring.factories` 的类路径中使用以下内容创建资源：

META-INF / spring.factories.

```
org.springframework.security.config.annotation.web.configurers.AbstractHttpConfigurer = sample.MyCustomDsl
```

希望禁用默认的用户可以明确地这样做。

```

@EnableWebSecurity
public class Config extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .apply(customDsl()).disable()
            ...;
    }
}

```

6. Security Namespace Configuration 译：6安全命名空间配置

6.1 Introduction 译：6.1介绍

名称空间配置自Spring Framework 2.0以来已可用。它允许您用传统的Spring bean应用程序上下文语法和其他XML模式中的元素进行补充。您可以在[Spring Reference Documentation](#)中找到更多信息。命名空间元素可以简单地用于配置单个Bean的更简洁的方式，或者更有力地定义更接近匹配问题域的替代配置语法，并隐藏用户的底层复杂性。一个简单的元素可能会隐藏多个bean和处理步骤被添加到应用程序上下文的事实。例如，将以下元素从安全性名称空间添加到应用程序上下文将启动一个嵌入式LDAP服务器，以在应用程序中测试使用情况：

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

这比连接相应的Apache Directory Server bean简单得多。`ldap-server`元素上的属性支持最常见的备用配置要求，并且用户不必担心需要创建哪些bean以及哪些bean属性名称。^[1]。编辑应用程序上下文文件时使用好的XML编辑器应提供有关可用属性和元素的信息。我们建议您尝试[Spring Tool Suite](#)，因为它具有用于处理标准Spring名称空间的特殊功能。

要开始在应用程序上下文中使用安全性名称空间，您需要在类路径中包含 `spring-security-config` jar。然后，您需要做的就是将模式声明添加到您的应用程序上下文

文件中：

```
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:security="http://www.springframework.org/schema/security"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/security
http://www.springframework.org/schema/security/spring-security.xsd">
...
</beans>
```

在很多你会看到的例子中（以及示例应用程序中），我们经常会使用“security”作为默认命名空间而不是“beans”，这意味着我们可以省略所有安全命名空间元素的前缀，从而使内容更容易阅读。如果将应用程序上下文划分为单独的文件并在其中一个文件中包含大部分安全配置，则可能还需要执行此操作。您的安全应用程序上下文文件将会像这样开始

```
<beans:beans xmlns="http://www.springframework.org/schema/security"
xmlns:beans="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/security
http://www.springframework.org/schema/security/spring-security.xsd">
...
</beans:beans>
```

我们假设这章的语法从现在开始被使用。

6.1.1 Design of the Namespace 译：6.1.1命名空间的设计

命名空间旨在捕获框架的最常见用途，并提供简化和简洁的语法来在应用程序中启用它们。该设计基于框架内的大规模依赖关系，可分为以下几个方面：

- *Web/HTTP Security* - the most complex part. Sets up the filters and related service beans used to apply the framework authentication mechanisms, to secure URLs, render login and error pages and much more.
- *Business Object (Method) Security* - options for securing the service layer.
- *AuthenticationManager* - handles authentication requests from other parts of the framework.
- *AccessDecisionManager* - provides access decisions for web and method security. A default one will be registered, but you can also choose to use a custom one, declared using normal Spring bean syntax.
- *AuthenticationProviders* - mechanisms against which the authentication manager authenticates users. The namespace provides supports for several standard options and also a means of adding custom beans declared using a traditional syntax.
- *UserDetailsService* - closely related to authentication providers, but often also required by other beans.

我们将在下面的章节中看到如何配置这些。

6.2 Getting Started with Security Namespace Configuration 译：6.2安全名称空间配置入门

在本节中，我们将看看如何构建一个名称空间配置来使用框架的一些主要功能。让我们假设您最初想要尽快启动并运行，并通过一些测试登录为现有的Web应用程序添加身份验证支持和访问控制。然后，我们将看看如何切换到对数据库或其他安全存储库进行身份验证。在后面的章节中，我们将介绍更高级的命名空间配置选项。

6.2.1 web.xml Configuration 译：6.2.1 web.xml配置

您需要做的第一件事是将以下过滤器声明添加到您的 `web.xml` 文件中：

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

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

这为Spring Security Web基础结构提供了一个钩子。`DelegatingFilterProxy` 是一个Spring框架类，它委派一个过滤器实现，它在应用程序上下文中被定义为一个Spring bean。在这种情况下，该bean被命名为“springSecurityFilterChain”，它是由命名空间创建的用于处理Web安全性的内部基础架构Bean。请注意，您不应该自己使用这个bean名称。一旦您将其添加到您的 `web.xml`，您 `web.xml` 可以开始编辑您的应用程序上下文文件。Web安全服务使用 `<http>` 元素进行配置。

6.2.2 A Minimal `<http>` Configuration 译：6.2.2最小<http>配置

您只需启用网络安全即可

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

其中说我们希望我们的应用程序中的所有URL都得到保护，要求角色 `ROLE_USER` 可以访问它们，我们希望使用带有用户名和密码的表单登录到应用程序，并且我们希望注册一个注销URL，这将允许我们注销应用程序。`<http>` 元素是所有与Web相关的名称空间功能的父项。`<intercept-url>` 元素定义了一个 `pattern`，它使用蚂蚁路径样式语法^[2]与传入请求的URL匹配。您也可以使用正则表达式匹配作为替代（有关更多详细信息，请参阅命名空间附录）。`access` 属性定义了匹配给定模式的请求的访问要求。通过默认配置，这通常是逗号分隔的角色列表，其中一个用户必须被允许发出请求。前缀“ROLE_”是一个标记，表示应该与用户的权限进行简单的比较。换句话说，应该使用正常的基于角色的检查。Spring Security中的访问控制不限于使用简单角色（因此使用前缀来区分不同类型的安全性）。我们稍后会看到解释如何变化脚注：[`access` 属性中逗号分隔值的 `access` 取决于所使用的“1”的实现。在Spring Security 3.0中，该属性也可以用“2”填充。



您可以使用多个 `<intercept-url>` 元素为不同的网址集定义不同的访问要求，但会按列出的顺序对其进行评估，并使用第一个匹配项。所以你必须把最具体的比赛放在最上面。还可以添加一个 `method` 属性到匹配限制在一个特定的HTTP方法（`GET`，`POST`，`PUT`等）。

要添加一些用户，您可以直接在命名空间中定义一组测试数据：

```
<authentication-manager>
<authentication-provider>
  <user-service>
    <!-- Password is prefixed with {noop} to indicate to DelegatingPasswordEncoder that
    NoOpPasswordEncoder should be used. This is not safe for production, but makes reading
    in samples easier. Normally passwords should be hashed using BCrypt -->
    <user name="jimi" password="{noop}jimispasword" authorities="ROLE_USER, ROLE_ADMIN" />
    <user name="bob" password="{noop}bobspasword" authorities="ROLE_USER" />
  </user-service>
</authentication-provider>
</authentication-manager>
```

这是存储相同密码的安全方式的一个例子。密码的前缀为 `{bcrypt}` 用于指示 `DelegatingPasswordEncoder`，它支持任何配置的 `PasswordEncoder` 进行匹配，密码使用BCrypt进行散列：

```
<authentication-manager>
<authentication-provider>
  <user-service>
    <user name="jimi" password="{bcrypt}$2a$10$ddEWZU18aU0GdZPPpy7wbu82dvEw/pBpbRvDQRqA41y6mK1CoH00m"
    authorities="ROLE_USER, ROLE_ADMIN" />
    <user name="bob" password="{bcrypt}$2a$10$/eLfpmBnAYYig6KRR5bv00YeZr1ie1hSogJryg9qDlha4oCw1Qka"
    authorities="ROLE_USER" />
    <user name="jimi" password="{noop}jimispasword" authorities="ROLE_USER, ROLE_ADMIN" />
    <user name="bob" password="{noop}bobspasword" authorities="ROLE_USER" />
  </user-service>
</authentication-provider>
</authentication-manager>
```

如果您熟悉框架的命名空间版本，您可能已经可以大概猜出这里发生了什么。`<http>` 元素负责创建 `FilterChainProxy` 和它使用的过滤器bean。由于过滤器位置是预定义的，因此不正确的过滤器排序等常见问题不再是问题。

`<authentication-provider>` 元素创建一个 `DaoAuthenticationProvider` bean，`<user-service>` 元素创建一个 `InMemoryDaoImpl`。所有 `<authentication-provider>` 元素都必须是 `<authentication-manager>` 元素，它会创建 `ProviderManager` 并向其注册身份验证提供程序。您可以在 [namespace appendix](#) 中找到有关创建的bean的更多详细信息。如果你想开始理解框架中的重要类是什么以及如何使用它们，那么值得交叉检查，特别是如果你想稍后定制事情。

上面的配置定义了两个用户，他们的密码和他们在应用程序中的角色（将用于访问控制）。另外，也可以使用从标准的属性文件中加载用户信息 `properties` 上属性 `user-service`。有关文件格式的更多详细信息，请参阅 [in-memory authentication](#) 上的部分。使用 `<authentication-provider>` 元素意味着用户信息将被认证管理器用于处理认证请求。您可以有多个 `<authentication-provider>` 元素来定义不同的身份验证来源，并且每个元素都将依次进行查询。

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

6.2.3 Form and Basic Login Options 译：6.2.3表单和基本登录选项

由于我们没有提及任何HTML文件或JSP，因此您可能想知道登录表单从何时被提示登录。事实上，由于我们没有明确设置登录页面的URL，因此Spring Security会根据启用的功能自动生成一个URL，并使用处理提交的登录的URL的标准值，用户的默认目标URL将在登录后发送到等等。但是，命名空间提供了大量的支持来允许您自定义这些选项。例如，如果您想提供自己的登录页面，则可以使用：

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

另外请注意，韦阿€™已经增加了一个额外 `intercept-url` 元说，对于登录页面的任何请求应提供给匿名用户^[3]，也是 `AuthenticatedVoter` 类的价值如何详情 `IS_AUTHENTICATED_ANONYMOUSLY` 被处理。]。否则，请求将被模式/ **匹配，并且它不可能访问登录页面本身！这是一个常见的配置错误，并且会导致应用程序中出现无限循环。如果您的登录页面似乎受到保护，Spring Security将在日志中发出警告。通过为模式定义一个单独的 `http` 元素，也可以使所有匹配特定模式的请求完全绕过安全过滤器链，如下所示：

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

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

从Spring Security 3.1开始，现在可以使用多个 `http` 元素为不同的请求模式定义不同的安全过滤器链配置。如果从 `http` 元素中省略 `pattern` 属性，它将匹配所有请求。创建一个不安全的模式就是这种语法的一个简单例子，其中模式映射到一个空的过滤器链^[4]。我们将在 [Security Filter Chain](#) 的章节中更详细地看这个新语法。

很重要的一点是要认识到这些不安全的请求将完全忽略任何Spring Security Web相关配置或其他属性（例如 `requires-channel`，因此您将无法访问当前用户的信息或在请求期间调用受保护的方法。如果您仍想要应用安全筛选器链，请使用 `access='IS_AUTHENTICATED_ANONYMOUSLY'` 作为替代方法。

如果您想使用基本身份验证而不是表单登录，请将配置更改为

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

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

Setting a Default Post-Login Destination 译：设置默认的登录后目标

如果表单登录不是尝试访问受保护资源的提示，则 `default-target-url` 选项将发挥作用。这是成功登录后用户将访问的URL，默认为“/”。通过将 `always-use-default-target` 属性设置为“true”，您还可以配置这些内容，以便用户始终在此页面结束（不管登录是“按需”还是明确选择登录）。如果您的应用程序始终要求用户在“主页”页面启动，这非常有用，例如：

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

要进一步控制目标，可以使用 `authentication-success-handler-ref` 属性替代 `default-target-url`。引用的bean应该是 `AuthenticationSuccessHandler` 的一个实例。在 [Core Filters](#) 一章以及命名空间附录中，您可以找到更多信息，以及有关如何在身份验证失败时自定义流的信息。

6.2.4 Logout Handling 译：6.2.4注销处理

`logout` 元素通过导航到特定的URL添加了对注销的支持。默认注销URL是 `/logout`，但您可以使用 `logout-url` 属性将其设置为其他内容。关于其他可用属性的更多信息可以在命名空间附录中找到。

6.2.5 Using other Authentication Providers 译：6.2.5使用其他身份验证提供程序

在实践中，您将需要一个更可扩展的用户信息来源，而不是添加到应用程序上下文文件中的几个名称。您很可能希望将您的用户信息存储在数据库或LDAP服务器中。LDAP名称空间配置在 [LDAP chapter](#) 中处理，所以我们在这里没有涉及。如果你有一个Spring Security的自定义实现 `UserDetailsService`，在你的应用程序上下文中被称为“myUserDetailsService”，那么您可以使用这个来进行身份验证

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

如果你想使用数据库，那么您可以使用

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

其中“securityDataSource”是应用程序上下文中 `DataSource` bean 的名称，指向包含标准Spring Security [user data tables](#) 的数据库。或者，您可以配置Spring Security `JdbcDaoImpl` bean并使用 `user-service-ref` 属性指向那个bean：

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

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

您还可以使用标准 `AuthenticationProvider` bean，如下所示

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

其中 `myAuthenticationProvider` 是应用程序上下文中实现 `AuthenticationProvider` 的bean的名称。您可以使用多个 `authentication-provider` 元素，在这种情况下，提供程序将按其声明的顺序进行查询。有关如何使用名称空间配置Spring Security `AuthenticationManager` 更多信息，请参见 [Section 6.6, "The Authentication Manager and the Namespace"](#)。

Adding a Password Encoder 译：添加密码编码器

应该始终使用为此目的而设计的安全哈希算法对密码进行编码（不是像SHA或MD5这样的标准算法）。这由 `password-encoder` 元素支持。使用bcrypt编码密码，原始身份验证提供程序配置将如下所示：

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

<authentication-manager>
<authentication-provider>
  <password-encoder ref="bcryptEncoder"/>
  <user-service>
    <user name="jimi" password="$2a$10$ddEWZU18aU0GdZPPpy7wbu82dvEw/pBpbRvDQRqA41y6mK1CoH00m"
      authorities="ROLE_USER, ROLE_ADMIN" />
    <user name="bob" password="$2a$10$/e1FpMBnAYYig6KRR5bv00YeZr1ie1hSogJryg9qD1hza4oCw1Qka"
      authorities="ROLE_USER" />
  </user-service>
</authentication-provider>
</authentication-manager>
```

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

6.3 Advanced Web Features 译：6.3高级Web功能

6.3.1 Remember-Me Authentication 译：6.3.1记住我的身份验证

有关记忆我命名空间配置的信息，请参阅单独的 [Remember-Me chapter](#)。

6.3.2 Adding HTTP/HTTPS Channel Security 译: 6.3.2添加HTTP/HTTPS通道安全性

如果您的应用程序同时支持HTTP和HTTPS，并且您要求只能通过HTTPS访问特定的URL，则可以使用 `requires-channel` 上的 `requires-channel` 属性直接支持此 `<intercept-url>`：

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

使用此配置，如果用户尝试使用HTTP访问与 `/secure / **` 模式匹配的任何内容，则会首先将其重定向到HTTPS URL ^[5]。可用的选项是“http”，“https”或“any”。使用值“any”表示可以使用HTTP或HTTPS。

如果您的应用程序使用HTTP和/或HTTPS的非标准端口，则可以按如下方式指定端口映射列表：

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

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

6.3.3 Session Management 译: 6.3.3会话管理

Detecting Timeouts 译: 检测超时

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

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

请注意，如果您使用此机制来检测会话超时，那么如果用户注销并重新登录而不关闭浏览器，它可能会错误地报告错误。这是因为会话cookie在会话无效时不会被清除，即使用户已注销，也会重新提交。您可以在注销时显式删除JSESSIONID cookie，例如在注销处理程序中使用以下语法：

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

不幸的是，这不能保证与每个servlet容器一起工作，所以你需要在你的环境中测试它



如果您在代理后运行应用程序，则可以通过配置代理服务器来删除会话cookie。例如，使用Apache HTTPD的mod_headers，以下指令将通过在注销请求（假定应用程序部署在路径 `/tutorial`）的响应中到期来删除 `JSESSIONID` cookie：

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

Concurrent Session Control 译: 并发会话控制

如果您希望限制单个用户登录到您的应用程序的能力，Spring Security通过以下简单添加支持这种开箱即用的功能。首先，您需要将以下监听器添加到 `web.xml` 文件中，以保持Spring Security关于会话生命周期事件的更新：

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

然后将以下行添加到您的应用程序上下文：

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

这将防止用户多次登录 - 第二次登录会导致第一次登录失效。通常情况下，您宁愿阻止第二次登录，在这种情况下您可以使用

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

第二次登录将被拒绝。通过“拒绝”，我们表示如果正在使用基于表单的登录，则用户将被发送到 `authentication-failure-url`。如果通过另一种非交互式机制（如“记住我”）进行第二次身份验证，则会向客户端发送“未经授权”（401）错误。如果您想要使用错误页面，则可以将属性 `session-authentication-error-url` 添加到 `session-management` 元素。

如果您使用自定义身份验证过滤器进行基于表单的登录，则必须明确配置并开发会话控制支持。 更多细节可以在[Session Management chapter](#)找到。

Session Fixation Attack Protection译:会话固定攻击保护

[Session fixation](#)攻击是一种潜在风险，因为恶意攻击者可能通过访问站点创建会话，然后诱使其他用户使用同一会话登录（例如，通过向其发送包含会话标识符作为参数的链接）。Spring Security通过创建新会话或以其他方式在用户登录时更改会话ID来自动防止此问题。如果您不需要此保护或与其他某些要求冲突，则可以使用[session-fixation-protection](#)属性控制行为[<session-management>](#)，有四个选项

- `none` - Don't do anything. The original session will be retained.
- `newSession` - Create a new "clean" session, without copying the existing session data (Spring Security-related attributes will still be copied).
- `migrateSession` - Create a new session and copy all existing session attributes to the new session. This is the default in Servlet 3.0 or older containers.
- `changeSessionId` - Do not create a new session. Instead, use the session fixation protection provided by the Servlet container (`HttpServletRequest.changeSessionId()`). This option is only available in Servlet 3.1 (Java EE 7) and newer containers. Specifying it in older containers will result in an exception. This is the default in Servlet 3.1 and newer containers.

当会话固定保护发生时，会导致在应用程序上下文中发布[SessionFixationProtectionEvent](#)。如果您使用[changeSessionId](#)，则此保护还会导致任何[javax.servlet.http.HttpSessionIdListener](#)被通知，因此如果您的代码侦听这两个事件，请谨慎使用。有关更多信息，请参阅[Session Management](#)一章。

6.3.4 OpenID Support译: 6.3.4 OpenID支持

命名空间支持 [OpenID](#)登录，而不是或者除了普通的基于表单的登录外，只需进行简单的更改即可：

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

然后，您应该向OpenID提供商（例如myopenid.com）注册自己，并将用户信息添加到您的内存[<user-service>](#)：

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

您应该可以使用[myopenid.com](#)网站进行身份验证。也可以选择特定的[UserDetailsService](#)通过设置豆使用OpenID的[user-service-ref](#)上属性[openid-login](#)元素。有关更多信息，请参阅上一节[authentication providers](#)。请注意，我们从上述用户配置中省略了密码属性，因为这组用户数据仅用于为用户加载权限。随机密码将在内部生成，从而防止您在配置中的其他位置意外地将此用户数据用作身份验证源。

Attribute Exchange译:属性交换

支持OpenID [attribute exchange](#)。例如，以下配置将尝试从OpenID提供程序中检索电子邮件和全名，供应用程序使用：

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

每个OpenID属性的“类型”是由特定模式确定的URI，在本例中为[http://axschema.org/](#)。如果必须检索属性才能成功进行身份验证，则可以设置[required](#)属性。支持的确切架构和属性取决于您的OpenID提供者。属性值作为身份验证过程的一部分返回，并可以使用以下代码进行访问：

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

[OpenIDAttribute](#)包含属性类型和检索值（或多值属性中的值）。当我们查看[technical overview](#)章节中的核心Spring Security组件时，我们将会看到更多[SecurityContextHolder](#)类的使用。如果您希望使用多个身份提供程序，则还支持多个属性交换配置。您可以提供多个[attribute-exchange](#)元素，每个元素使用[identifier-matcher](#)属性。这包含一个正则表达式，它与用户提供的OpenID标识符相匹配。有关示例配置，请参阅代码库中的OpenID示例应用程序，为Google, Yahoo和MyOpenID提供程序提供不同的属性列表。


6.3.5 Response Headers译: 6.3.5响应头

有关如何自定义标题元素的更多信息，请参阅参考文献的 [Chapter 21, Security HTTP Response Headers](#)部分。

6.3.6 Adding in Your Own Filters译: 6.3.6添加您的过滤器

如果您以前使用过Spring Security，那么您将会知道该框架维护一系列过滤器以应用其服务。您可能希望将自己的过滤器添加到特定位置的堆栈中，或者使用目前不存在名称空间配置选项（例如CAS）的Spring Security过滤器。或者您可能想使用一个标准命名空间过滤器，自定义版本，如[UsernamePasswordAuthenticationFilter](#)这是由创建[<form-login>](#)元素，服用一些它们可通过明确使用bean额外配置选项的优势。你怎么能用命名空间配置来做到这一点，因为过滤器链不直接暴露？

使用名称空间时，过滤器的顺序始终严格执行。在创建应用程序上下文时，过滤器bean按名称空间处理代码进行排序，标准Spring Security过滤器在名称空间和知名位置都有一个别名。



在以前的版本中，在应用程序上下文的后处理期间，创建过滤器实例之后进行排序。在版本3.0+中，现在在bean实例化之前，在bean元数据级完成排序。这会影响您如何将您的过滤器添加到堆栈中，因为在解析[<http>](#)元素时必须知道整个过滤器列表，因此语法在3.0中略有变化。

Table 6.1, “Standard Filter Aliases and Ordering”中显示了创建过滤器的过滤器，别名和命名空间元素/属性。过滤器按照它们在过滤器链中出现的顺序列出。

Alias	Filter Class	Namespace Element or Attribute
CHANNEL_FILTER	ChannelProcessingFilter	http/
SECURITY_CONTEXT_FILTER	SecurityContextPersistenceFilter	http

Alias	Filter Class	Namespace Element or Attribute
CONCURRENT_SESSION_FILTER	ConcurrentSessionFilter	session-management/concurrency-control
HEADERS_FILTER	HeaderWriterFilter	http/headers
CSRF_FILTER	CsrfFilter	http/csrf
LOGOUT_FILTER	LogoutFilter	http/logout
X509_FILTER	X509AuthenticationFilter	http/x509
PRE_AUTH_FILTER	AbstractPreAuthenticatedProcessingFilter 子类	N/A
CAS_FILTER	CasAuthenticationFilter	N/A
FORM_LOGIN_FILTER	UsernamePasswordAuthenticationFilter	http/form-login
BASIC_AUTH_FILTER	BasicAuthenticationFilter	http/http-basic
SERVLET_API_SUPPORT_FILTER	SecurityContextHolderAwareRequestFilter	http/@servlet-api-provision
JAAS_API_SUPPORT_FILTER	JaasApiIntegrationFilter	http/@jaas-api-provision
REMEMBER_ME_FILTER	RememberMeAuthenticationFilter	http/remember-me
ANONYMOUS_FILTER	AnonymousAuthenticationFilter	http/anonymous
SESSION_MANAGEMENT_FILTER	SessionManagementFilter	session-management
EXCEPTION_TRANSLATION_FILTER	ExceptionTranslationFilter	http
FILTER_SECURITY_INTERCEPTOR	FilterSecurityInterceptor	http
SWITCH_USER_FILTER	SwitchUserFilter	N/A

您可以将自己的过滤器添加到堆栈，使用 `custom-filter` 元素和其中一个名称指定过滤器应显示在的位置：

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

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

如果您希望在堆栈中的另一个过滤器之前或之后插入过滤器，也可以使用 `after` 或 `before` 属性。名称“FIRST”和“LAST”可以与 `position` 属性一起使用，以指示您希望过滤器分别出现在整个堆栈之前或之后。



如果您插入的自定义过滤器可能与名称空间创建的标准过滤器之一占据相同的位置，那么重要的是不要错误地包含命名空间版本。删除所有创建要替换其功能的过滤器的元素。

请注意，您德卡尼亚™吨替换那些使用中创建的过滤器 `<http>` 元素本身- `SecurityContextPersistenceFilter`，`ExceptionTranslationFilter` 或者 `FilterSecurityInterceptor`。其他一些过滤器默认添加，但您可以禁用它们。默认情况下会添加 `AnonymousAuthenticationFilter`，除非禁用 `session-fixation protection`，否则 `SessionManagementFilter` 也会添加到过滤器链中。

如果您要替换需要身份验证入口点的名称空间过滤器（即身份验证过程由未经身份验证的用户尝试访问受保护资源触发），则还需要添加自定义入口点bean。

Setting a Custom AuthenticationEntryPoint 译：设置自定义 AuthenticationEntryPoint

如果您未使用表单登录，OpenID或基于命名空间的基本身份验证，则可能需要使用传统的bean语法定义身份验证过滤器和入口点，并将它们链接到名称空间中，就像我们刚刚看到的那样。相应 `AuthenticationEntryPoint` 可以使用被设置 `entry-point-ref` 的上属性 `<http>` 元件。

CAS示例应用程序是使用包含此语法的名称空间的自定义Bean的一个很好的示例。如果您不熟悉身份验证入口点，请参阅 [technical overview](#) 一章。

6.4 Method Security 译：6.4 方法安全性

从2.0版本开始，Spring Security已经大大改善了对服务层方法的安全性的支持。它提供对JSR-250注释安全性的支持以及框架的原始 `@Secured` 注释。从3.0开始，您还可以使用新的 `expression-based annotations`。您可以将安全性应用于单个bean，使用 `intercept-methods` 元素来装饰bean声明，或者可以使用AspectJ样式切入点在整个服务层中保护多个bean。

6.4.1 The <global-method-security> Element 译：6.4.1 <global-method-security>元素

此元素用于在应用程序中启用基于注释的安全性（通过在元素上设置适当的属性），并将安全性切入点声明分组在一起，这些声明将应用于整个应用程序上下文中。您应该只声明一个 `<global-method-security>` 元素。以下声明将支持Spring Security的 `@Secured`：

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

向方法（在类或接口上）添加注释会相应地限制对该方法的访问。Spring Security的本地注释支持为该方法定义了一组属性。这些将被传递给 `AccessDecisionManager`，以便做出实际决定：


```
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);
}
```

可以使用支持JSR-250注释

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

这些是基于标准的，允许应用简单的基于角色的约束，但是没有Spring Security的本地注释的强大功能。要使用新的基于表达式的语法，您可以使用

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

和等效的Java代码将会是

```
public interface BankService {

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

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

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

如果您需要定义简单的规则，而不是根据用户的权限列表检查角色名称，则基于表达式的注释是一个不错的选择。



注释的方法只会对定义为Spring bean的实例（在启用了方法安全性的相同应用程序上下文中）保护。如果你想确保那些不是Spring创建的实例（例如使用 `new` 运算符），那么你需要使用AspectJ。



您可以在同一个应用程序中启用多种类型的注释，但只有一种类型应该用于任何界面或类，否则将无法很好地定义行为。如果找到适用于特定方法的两个注释，则只会应用其中的一个注释。

Adding Security Pointcuts using protect-pointcut 译:使用protect-pointcut添加安全性切入点

`protect-pointcut` 的使用特别强大，因为它允许您只通过简单的声明就可以将安全性应用于许多bean。考虑下面的例子：

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

这将保护应用程序上下文中声明的bean上的所有方法，这些类的类位于 `com.mycompany` 包中，并且其类名以“Service”结尾。只有具有 `ROLE_USER` 角色的用户才能够调用这些方法。与URL匹配一样，最具体的匹配必须在切入点列表中排在第一位，因为将使用第一个匹配表达式。安全注释优先于切入点。

6.5 The Default AccessDecisionManager 译:6.5.10 默认AccessDecisionManager

本节假定您对Spring Security中访问控制的基础体系结构有一定的了解。如果你不能跳过它并稍后再回来看看，因为这部分只对那些需要进行一些定制以便使用不仅仅是简单的基于角色的安全性的人真正相关。

当您使用名称空间配置时，会自动为您注册一个默认实例 `AccessDecisionManager`，并根据您在 `intercept-url` 和 `protect-pointcut` 声明（以及注释中指定的访问属性，用于为方法调用和Web URL访问做出访问决定如果你使用注解安全的方法）。

默认策略是将 `AffirmativeBased` `AccessDecisionManager` 与 `RoleVoter` 和 `AuthenticatedVoter`。您可以在 [authorization](#) 的章节中找到更多关于这些 [信息](#)。

6.5.1 Customizing the AccessDecisionManager 译:6.5.10 定义AccessDecisionManager

如果您需要使用更复杂的访问控制策略，则很容易为方法和网络安全设置替代方案。

对于方法安全，你通过设置这样做 `access-decision-manager-ref` 的属性 `global-method-security` 到 `id` 适当的 `AccessDecisionManager` 在应用程序上下文的bean：

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

网络安全的语法是相同的，但在 `http` 元素上：

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

6.6 The Authentication Manager and the Namespace 译:6.6 认证管理器和命名空间

在Spring Security中提供验证服务的主界面是 `AuthenticationManager`。这通常是Spring Security的 `ProviderManager` 类的一个实例，如果您以前使用过该框架，您可

能已经熟悉了这个实例。如果没有，它将在[technical overview chapter](#)后面介绍。该bean实例是使用 `authentication-manager` 命名空间元素注册的。如果您通过名称空间使用HTTP或方法安全性，则无法使用自定义 `AuthenticationManager`，但这应该不成问题，因为您完全可以控制所使用的 `AuthenticationProvider`。

您可能需要注册其他 `AuthenticationProvider` 与豆 `ProviderManager`，你可以用做 `<authentication-provider>` 元素与 `ref` 属性，当属性的值是您要添加的提供商bean的名字。例如：

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

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

另一个常见要求是上下文中的另一个bean可能需要引用 `AuthenticationManager`。您可以轻松注册 `AuthenticationManager` 的别名，并在您的应用程序上下文的其他地方使用该名称。

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

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

- [1]，您可以了解更多有关使用的 `ldap-server` 的章节中元素 [Chapter 30, LDAP Authentication](#)。
- [2] 请参见上一节 [Section 14.4, "Request Matching and HttpFirewall"](#) 有关更多详细的Web应用程序基础设施一章的比赛实际上是如何进行的。
- [3] 参见本章 [Chapter 23, Anonymous Authentication](#)
- [4] 使用多个 `<http>` 元素是一项重要功能，例如，允许命名空间在同一应用程序中同时支持有状态路径和无状态路径。以前的语法使用 `intercept-url` 元素上的属性 `filters="none"` 与此更改不兼容，并且在3.1中不再支持。
- [5] 有关如何实现通道处理的更多详细信息，请参阅Javadoc for `ChannelProcessingFilter` 和相关类。

7. Sample Applications

有几个示例Web应用程序可用于该项目。为避免过大的下载，分发zip文件中只包含“教程”和“联系人”示例。其他可以直接从您可以获得的来源构建，如[the introduction](#)中所述。您可以轻松地自行构建项目，并在项目网站<http://spring.io/spring-security/>上提供更多信息。本章中提到的所有路径都与项目目录有关。

7.1 Tutorial Sample

教程示例是帮助您入门的一个很好的基本示例。它始终使用简单的名称空间配置。已编译的应用程序已包含在分发zip文件中，可随时部署到您的Web容器（`spring-security-samples-tutorial-3.1.x.war`）中。`form-based` 身份验证机制与常用的 `remember-me` 身份验证提供程序结合使用，以便使用Cookie自动记住登录。

我们建议您从教程示例开始，因为XML最小且易于遵循。最重要的是，您可以轻松地将这一个XML文件（及其相应的 `web.xml` 条目）添加到您的现有应用程序中。只有在实现这种基本集成时，我们才建议您尝试添加方法授权或域对象安全性。

7.2 Contacts

联系人范例是一个高级示例，它说明了除基本应用程序安全性以外，域对象访问控制列表（ACL）的更强大功能。该应用程序提供了一个用户可以管理联系人简单数据库（域对象）的界面。

要进行部署，只需将Spring Security发行版中的WAR文件复制到您的容器的 `webapps` 目录中即可。战争应该被称为 `spring-security-samples-contacts-3.1.x.war`（附加版本号会根据您使用的版本而有所不同）。

启动容器后，检查应用程序是否可以加载。访问<http://localhost:8080/contacts>（或适用于您的Web容器和您部署的WAR的URL）。

接下来，点击“调试”。系统会提示您进行身份验证，并在该页面上建议一系列用户名和密码。只需对其中任何一个进行身份验证并查看生成的页面。它应该包含类似于以下内容的成功消息：

```
Security Debug Information

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

Authentication object as a String:

org.springframework[email protected]1f127853:
Principal: [email protected]: Username: rod; \
Password: [PROTECTED]; Enabled: true; AccountNonExpired: true;
credentialsNonExpired: true; AccountNonLocked: true; \
Granted Authorities: ROLE_SUPERVISOR, ROLE_USER; \
Password: [PROTECTED]; Authenticated: true; \
Details: org.sprin[email protected]0: \
RemoteIpAddress: 127.0.0.1; SessionId: 8fkp8t83ohar; \
Granted Authorities: ROLE_SUPERVISOR, ROLE_USER

Authentication object holds the following granted authorities:

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

Success! Your web filters appear to be properly configured!
```

一旦您成功收到上述消息，返回到示例应用程序的主页并单击“管理”。然后您可以尝试应用程序。请注意，只显示当前登录用户可用的联系人，并且只有具

有 `ROLE_SUPERVISOR` 用户 `ROLE_SUPERVISOR` 授予访问权限以删除其联系人。在幕后， `MethodSecurityInterceptor` 正在保护业务对象。

该应用程序允许您修改与不同联系人关联的访问控制列表。请务必通过查看应用程序上下文XML文件来尝试并了解它是如何工作的。

7.3 LDAP Sample 译: 7.3 LDAP示例

LDAP示例应用程序提供基本配置，并使用传统Bean设置名称空间配置和等效配置，这两者都位于同一应用程序上下文文件中。这意味着实际上在此应用程序中配置了两个相同的认证提供程序

7.4 OpenID Sample 译: 7.4 OpenID示例

OpenID示例演示了如何使用命名空间来配置OpenID，以及如何为Google, Yahoo和MyOpenID身份提供商设置 `attribute exchange` 配置（如果您愿意，可以尝试添加其他人）。它使用基于jQuery的 `openid-selector` 项目提供用户友好的登录页面，允许用户轻松选择提供者，而不是输入完整的OpenID标识符。

该应用程序不同于正常的身份验证方案，因为它允许任何用户访问该网站（只要其OpenID身份验证成功）。第一次登录时，你会得到一个“欢迎你的名字”的消息，如果你注销并重新登录（使用相同的OpenID标识），那么这应该改为“欢迎回来”，这是通过使用自定义 `UserDetailsService`，它为任何用户分配标准角色，并将标识内部存储在地图中。显然，真正的应用程序会使用数据库来代替。可能会从不同的提供者返回，并建立相应的用户名称。

7.5 CAS Sample 译: 7.5 CAS样品

CAS示例要求您同时运行CAS服务器和CAS客户端。它不包含在发行版中，因此您应该按照 [the introduction](#) 中的描述检查项目代码。您将在 `sample/cas` 目录下找到相关文件。这里还有一个 `Readme.txt` 文件，它解释了如何直接从源代码树运行服务器和客户端，并完成SSL支持。

7.6 JAAS Sample 译: 7.6 JAAS样品

JAAS示例是如何在Spring Security中使用JAAS LoginModule的非常简单的示例。如果用户名等于密码，则所提供的LoginModule将成功认证用户，否则会引发LoginException。本例中使用的AuthorityGranter总是授予角色ROLE_USER。示例应用程序还演示了如何通过将 `jaas-api-provision` 设置为等于“true”的方式作为LoginModule返回的JAAS主体运行。

7.7 Pre-Authentication Sample 译: 7.7预认证示例

此示例应用程序演示了如何连接 `pre-authentication` 框架中的bean，以利用来自Java EE容器的登录信息。用户名和角色是由容器设置的。

代码是 `samples/preauth`。

8. Spring Security Community 译: 8春季安全社区

8.1 Issue Tracking 译: 8.1问题跟踪

Spring Security使用JIRA来管理错误报告和增强请求。如果您发现错误，请使用JIRA登录报告。请勿将其登录到支持论坛，邮件列表或通过电子邮件发送给项目开发人員。这种方法是临时的，我们更愿意使用更正式的流程来管理错误。

如果可能，请在您的问题报告中提供一个JUnit测试，以显示任何不正确的行为。或者，更好的是，提供一个补丁来纠正问题。同样，我们也欢迎增强功能在问题跟踪器中登录，但如果您包含相应的单元测试，我们只接受增强请求。这对确保项目测试框架得到充分维护是必要的。

您可以访问问题跟踪器 <https://github.com/spring-projects/spring-security/issues>。

8.2 Becoming Involved 译: 8.2成为参与者

我们欢迎您参与Spring Security项目。有很多贡献方式，包括阅读论坛并回答其他人提出的问题，编写新代码，改进现有代码，协助编写文档，开发示例或教程，或仅仅提出建议。

8.3 Further Information 译: 8.3更多信息

欢迎关于Spring Security的问题和评论。您可以在Stack Overflow网站上使用Spring，网址为<http://spring.io/questions>，与框架的其他用户讨论Spring Security。如上所述，请记住使用JIRA进行错误报告。

Part II. Architecture and Implementation 译: 第二部分. 架构和实现

熟悉设置和运行一些基于命名空间配置的应用程序后，您可能希望更深入地了解框架在命名空间外观背后的工作原理。像大多数软件一样，Spring Security在整个框架中都有一些中心接口，类和概念抽象。在参考指南的这一部分，我们将看看其中的一些内容，看看它们如何协同工作来支持Spring Security中的认证和访问控制。

9. Technical Overview 译: 9技术概述

9.1 Runtime Environment 译: 9.1运行环境

Spring Security 3.0需要Java 5.0 Runtime Environment或更高版本。由于Spring Security旨在以独立方式运行，因此不需要将任何特殊配置文件放入Java运行时环境中。尤其是，不需要配置特殊的Java身份验证和授权服务（JAAS）策略文件，也不需要将Spring Security放入常见的类路径位置。

同样，如果您使用的是EJB容器或Servlet容器，则不需要在任何地方放置任何特殊配置文件，也不需要将Spring Security包含在服务器类加载器中。所有必需的文件都将包含在您的应用程序中。

这种设计提供了最大的部署时间灵活性，因为您可以简单地将目标工件（不管是JAR，WAR还是EAR）从一个系统复制到另一个系统，并立即生效。

9.2 Core Components 译: 9.2核心组件

在Spring Security 3.0中，`spring-security-core` jar的内容被精简到最低限度。它不再包含任何与Web应用程序安全性，LDAP或命名空间配置相关的代码。我们将在这里看看您将在核心模块中找到的一些Java类型。它们代表框架的构建模块，所以如果你需要超越一个简单的命名空间配置，那么重要的是你要明白它们是什么，即使你实际上不需要直接与它们进行交互。

9.2.1 SecurityContextHolder, SecurityContext and Authentication Objects 译: 9.2.1 SecurityContextHolder, SecurityContext和认证对象

最基本的对象是`SecurityContextHolder`。这是我们存储应用程序当前安全上下文的详细信息的地方，其中包括当前使用该应用程序的主体的详细信息。默认情况下，`SecurityContextHolder`使用`ThreadLocal`来存储这些细节，这意味着安全上下文始终可用于同一执行线程中的方法，即使安全上下文没有显式作为参数传递给这些方法。以这种方式使用`ThreadLocal`是非常安全的，因为如果在处理当前委托人的请求之后谨慎清除线程，那么它是非常安全的。当然，Spring Security会自动为您处理，因此您无需担心。

有些应用程序`ThreadLocal`完全适合使用`ThreadLocal`，因为它们使用线程的具体方式。例如，Swing客户端可能希望Java虚拟机中的所有线程使用相同的安全上下文。可以使用启动时的策略配置`SecurityContextHolder`以指定如何存储上下文。对于独立应用程序，您可以使用`SecurityContextHolder.MODE_GLOBAL`策略。其他应用程序可能希望安全线程产生的线程也具有相同的安全身份。这是通过使用`SecurityContextHolder.MODE_INHERITABLETHREADLOCAL`实现的。您可以通过两种方式从默认的`SecurityContextHolder.MODE_THREADLOCAL`更改模式。第一种是设置系统属性，第二种是在`SecurityContextHolder`上调用静态方法。大多数应用程序不需要从默认设置改变，但如果你这样做，请参阅JavaDoc for `SecurityContextHolder`以了解更多信息。

Obtaining information about the current user 译: 获取有关当前用户的信息

在`SecurityContextHolder`内部，我们存储了当前与应用程序交互的委托人的详细信息。Spring Security使用一个`Authentication`对象来表示这些信息。您通常不需要自己创建一个`Authentication`对象，但用户查询`Authentication`对象相当常见。您可以使用以下代码块 - 从应用程序中的任何位置 - 获取当前通过身份验证的用户名称，例如：

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

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

`getContext()`调用返回的对象是`SecurityContext`接口的一个实例。这是保存在线程本地存储中的对象。正如我们将在下面看到的，Spring Security中的大多数身份验证机制均返回一个`UserDetails`的实例作为主体。

9.2.2 The UserDetailsService 译: 9.2.2 UserDetailsService

从上面的代码片段中要注意的另一个项目是，您可以从`Authentication`对象获取主体。校长只是一个`Object`。大多数情况下，这可以投射到`UserDetails`对象中。`UserDetails`是Spring Security的核心接口。它代表一个委托人，但以一种可扩展的和特定于应用程序的方式。把`UserDetails`成你自己的用户数据库和Spring Security在`SecurityContextHolder`里需要的`SecurityContextHolder`。为了让你自己的用户数据库起作用，往往你会播放`UserDetails`到您的应用程序提供的原始对象，这样您就可以调用业务相关的方法（比如`getEmail()`，`getEmployeeNumber()`等）。

现在你可能想知道，那么我何时提供一个`UserDetails`对象？我怎么做？我以为你说这件事是声明式的，我不需要编写任何Java代码 - 是什么给了？简短的答案是有一个特殊的界面叫做`UserDetailsService`。此接口上的唯一方法接受基于`String`的用户名参数并返回`UserDetails`：

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

这是在Spring Security中为用户加载信息的最常见方法，只要需要用户信息，您就会看到它在整个框架中使用。

成功身份验证时，`UserDetails`用于构建存储在`SecurityContextHolder`的`Authentication`对象（更多信息，请参阅below）。好消息是我们提供了许多`UserDetailsService`实现，其中包括使用内存映射（`InMemoryDaoImpl`）的另一个实现，以及另一个使用JDBC（`JdbcDaoImpl`）的实现。不过，大多数用户倾向于编写他们自己的代码，而他们的实现通常只是坐在代表其员工，客户或应用程序其他用户的现有数据访问对象（DAO）之上。请记住，使用上面的代码片段可以始终从`SecurityContextHolder`获得任何`UserDetailsService`返回的优势。



`UserDetailsService`常常有些混淆。它纯粹是用于用户数据的DAO，除了将该数据提供给框架内的其他组件外，不执行其他功能。特别是，它不验证用户，这由`AuthenticationManager`完成。在很多情况下，如果您需要自定义身份验证过程，则implement `AuthenticationProvider`会更直接。

9.2.3 GrantedAuthority 译: 9.2.3授予权力

除了委托人之外，`Authentication`提供的另一种重要方法是`getAuthorities()`。此方法提供了一个包含`GrantedAuthority`对象的数组。毫不奇怪，`GrantedAuthority`是授予校长的权威。这些权力通常是“角色”，例如`ROLE_ADMINISTRATOR`或`ROLE_HR_SUPERVISOR`。稍后将为Web授权，方法授权和域对象授权配置这些角色。Spring Security的其他部门有能力解释这些权威，并希望他们出席。`GrantedAuthority`对象通常由`UserDetailsService`加载。

通常`GrantedAuthority`对象是应用程序范围的权限。它们不是特定于给定域对象的。因此，你wouldn't可能有`GrantedAuthority`来表示权限`Employee`对象是54号，因为如果有成千上万这样的授权，你会很快走出的运行内存（或者，至少是，导致应用程序采取很长时间来验证用户）。当然，Spring Security的设计明确是为了处理这个共同的需求，但是你需要使用项目的域对象安全功能来达到这个目的。

9.2.4 Summary 译: 9.2.4总结

回顾一下，我们迄今为止看到的Spring Security的主要构建块是：

- `SecurityContextHolder`, to provide access to the `SecurityContext`.
- `SecurityContext`, to hold the `Authentication` and possibly request-specific security information.
- `Authentication`, to represent the principal in a Spring Security-specific manner.
- `GrantedAuthority`, to reflect the application-wide permissions granted to a principal.
- `UserDetails`, to provide the necessary information to build an Authentication object from your application's DAOs or other source of security data.
- `UserDetailsService`, to create a `UserDetails` when passed in a `String`-based username (or certificate ID or the like).

现在您已经了解了这些重复使用的组件，让我们仔细看看认证过程。

9.3 Authentication 译: 9.3认证

Spring Security可以参与许多不同的认证环境。尽管我们建议人们使用Spring Security进行身份验证, 并且不会与现有的容器管理身份验证集成, 但仍然支持 - 与您自己的专有身份验证系统集成。

9.3.1 What is authentication in Spring Security? 译: 9.3.1什么是Spring Security中的认证?

让我们考虑一个每个人都熟悉的标准认证场景。

1. A user is prompted to log in with a username and password.
2. The system (successfully) verifies that the password is correct for the username.
3. The context information for that user is obtained (their list of roles and so on).
4. A security context is established for the user
5. The user proceeds, potentially to perform some operation which is potentially protected by an access control mechanism which checks the required permissions for the operation against the current security context information.

前三项构成了认证过程, 所以我们将Spring Security中看看这些是如何发生的。

1. The username and password are obtained and combined into an instance of `UsernamePasswordAuthenticationToken` (an instance of the `Authentication` interface, which we saw earlier).
2. The token is passed to an instance of `AuthenticationManager` for validation.
3. The `AuthenticationManager` returns a fully populated `Authentication` instance on successful authentication.
4. The security context is established by calling `SecurityContextHolder.getContext().setAuthentication(...)`, passing in the returned authentication object.

从那时起, 用户被认为是被认证的。让我们看一些代码作为例子。

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

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

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

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

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

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

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

这里我们写了一个小程序, 要求用户输入用户名和密码并执行上述顺序。我们在这里实现的 `AuthenticationManager` 将验证任何用户名和密码相同的用户。它为每个用户分配一个角色。从上面的输出将会是这样的:

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

请注意, 你通常不需要写这样的代码。该过程通常在内部发生, 例如在Web认证过滤器中。我们在这里只包含了代码, 以表明在Spring Security中实际构成认证的问题有

一个相当简单的答案。当 `SecurityContextHolder` 包含完全填充的 `Authentication` 对象时，将验证用户身份。

9.3.2 Setting the SecurityContextHolder Contents Directly 译：9.3.2直接设置SecurityContextHolder内容

其实，春季安全doesn't介意你如何把 `Authentication` 内部对象 `SecurityContextHolder` 。唯一关键的要求是 `SecurityContextHolder` 包含一个 `Authentication` ，它表示 `AbstractSecurityInterceptor` （我们将在后面看到更多信息）之前的 `AbstractSecurityInterceptor` 需要授权用户操作。

您可以（以及许多用户）编写自己的过滤器或MVC控制器，以提供与基于Spring Security的身份验证系统的互操作性。例如，您可能正在使用容器管理的身份验证，它使当前用户可以从ThreadLocal或JNDI位置获得。或者你可能会为拥有传统专有认证系统的公司工作，这是一个企业“标准”，你无法控制。在这种情况下，Spring Security很容易工作，并且仍然提供授权功能。你所需要做的就是编写一个过滤器（或者等价物），从一个位置读取第三方用户信息，构建一个Spring Security特有的 `Authentication` 对象，并将它放入 `SecurityContextHolder` 。在这种情况下，您还需要考虑通常由内置身份验证基础结构自动处理的事情。例如，在您对客户端脚注编写响应之前，您可能需要预先创建一个HTTP会话到 [cache the context between requests](#) ：[在响应提交后不能创建会话。

如果您想知道 `AuthenticationManager` 是如何在现实世界中实现的，我们将在 [core services chapter](#) 中查看。

9.4 Authentication in a Web Application 译：9.4 Web应用程序中的身份验证

现在让我们来探索一下您在Web应用程序中使用Spring Security的情况（未启用 `web.xml` 安全性）。用户如何进行身份验证并建立安全上下文？

考虑一个典型的Web应用程序的身份验证过程：

1. You visit the home page, and click on a link.
2. A request goes to the server, and the server decides that you've asked for a protected resource.
3. As you're not presently authenticated, the server sends back a response indicating that you must authenticate. The response will either be an HTTP response code, or a redirect to a particular web page.
4. Depending on the authentication mechanism, your browser will either redirect to the specific web page so that you can fill out the form, or the browser will somehow retrieve your identity (via a BASIC authentication dialogue box, a cookie, a X.509 certificate etc.).
5. The browser will send back a response to the server. This will either be an HTTP POST containing the contents of the form that you filled out, or an HTTP header containing your authentication details.
6. Next the server will decide whether or not the presented credentials are valid. If they're valid, the next step will happen. If they're invalid, usually your browser will be asked to try again (so you return to step two above).
7. The original request that you made to cause the authentication process will be retried. Hopefully you've authenticated with sufficient granted authorities to access the protected resource. If you have sufficient access, the request will be successful. Otherwise, you'll receive back an HTTP error code 403, which means "forbidden".

Spring Security具有不同的类，负责上述大多数步骤。主要参与者（在使用它们的顺序）是 `ExceptionHandlerFilter` ，一个 `AuthenticationEntryPoint` 和“验证机制”，它负责调用 `AuthenticationManager` 我们在上一节中所看到的。

9.4.1 ExceptionHandlerFilter 译：9.4.1 ExceptionHandlerFilter

`ExceptionHandlerFilter` 是一个Spring Security过滤器，负责检测抛出的任何Spring安全异常。 `AbstractSecurityInterceptor` 通常会引发这种例外情况，这是授权服务的主要提供者。我们将在下一节讨论 `AbstractSecurityInterceptor` ，但现在我们只需要知道它会产生Java异常，并且对HTTP没有任何认识，或者如何去认证一个主体。而是 `ExceptionHandlerFilter` 提供此服务，具体负责返回错误代码403（如果委托人已通过身份验证，因此根本没有足够的访问权限 - 按照上述第7步），或启动 `AuthenticationEntryPoint` （如果委托人未经过身份验证并且因此我们需要开始第三步）。

9.4.2 AuthenticationEntryPoint 译：9.4.2 AuthenticationEntryPoint

`AuthenticationEntryPoint` 负责上述列表中的第三步。正如你可以想象的那样，每个Web应用程序都会有一个默认的身份验证策略（当然，这可以像Spring Security中的其他任何东西一样进行配置，但是现在让我们保持简单）。每个主要认证系统都有其自己的 `AuthenticationEntryPoint` 实施，该实施通常执行步骤3中描述的操作之一。

9.4.3 Authentication Mechanism 译：9.4.3 认证机制

一旦您的浏览器提交了您的身份验证凭证（无论是作为HTTP表单发布还是HTTP标头），服务器上都需要有一些“收集”这些身份验证信息的东西。现在我们在上面的列表中的第六步。在Spring Security中，我们为从用户代理（通常是Web浏览器）收集验证信息的功能提供了一个特殊的名称，称之为“验证机制”。示例是基于表单的登录和基本身份验证。一旦从用户代理收集了认证详细信息，就会构建一个 `Authentication` “请求”对象，然后将其呈现给 `AuthenticationManager` 。

在身份验证机制接收到完全填充的 `Authentication` 对象后，它将认为请求有效，将 `Authentication` 放入 `SecurityContextHolder` ，并导致原始请求重试（上面的第7步）。另一方面，如果 `AuthenticationManager` 拒绝了请求，认证机制将要求用户代理重试（上面的第二步）。

9.4.4 Storing the SecurityContext between requests 译：9.4.4在请求之间存储SecurityContext

根据应用程序的类型，可能需要制定一个策略来存储用户操作之间的安全上下文。在典型的Web应用程序中，用户登录一次，随后通过其会话ID进行标识。服务器缓存持续时间会话的主要信息。在Spring Security中，在请求之间存储 `SecurityContext` 的责任降至 `SecurityContextPersistenceFilter` ，默认情况下，该上下文将该上下文存储为HTTP请求之间的 `HttpSession` 属性。它将每个请求的上下文恢复到 `SecurityContextHolder` ，并且关键地，在请求完成时清除 `SecurityContextHolder` 。出于安全目的，您不应该直接与 `HttpSession` 交互。这样做根本没有理由 - 总是使用 `SecurityContextHolder` 代替。

许多其他类型的应用程序（例如，无状态的RESTful Web服务）不使用HTTP会话，并将在每个请求中重新进行身份验证。但是， `SecurityContextPersistenceFilter` 包含在链中以确保 `SecurityContextHolder` 在每个请求 `SecurityContextHolder` 清除仍然很重要。



在单个会话中接收并发请求的应用程序中，同一个 `SecurityContext` 实例将在线程之间共享。即使正在使用 `ThreadLocal` ，它也是从 `HttpSession` 为每个线程检索的实例。如果您想临时更改线程正在运行的上下文，这会产生影响。如果您只使用 `SecurityContextHolder.getContext()` ，并且在返回的上下文对象上调用 `setAuthentication(anAuthentication)` ，则 `Authentication` 对象将在共享相同 `SecurityContext` 实例的所有并发线程中更改。您可以自定义 `SecurityContextPersistenceFilter` 的行为，为每个请求创建一个全新的 `SecurityContext` ，防止一个线程中的更改影响另一个线程。或者，您可以在临时更改上下文的位置创建新实例。方法 `SecurityContextHolder.createEmptyContext()` 总是返回一个新的上下文实例。

9.5 Access-Control (Authorization) in Spring Security 译：9.5 Spring Security中的访问控制（授权）

负责Spring Security访问控制决策的主界面是 `AccessDecisionManager` 。它有一个 `decide` 方法，它需要一个表示请求主体访问的 `Authentication` 对象，一个“安全对象”（见下文）和一个适用于该对象的安全元数据属性列表（例如访问所需的角色列表被授予）。

9.5.1 Security and AOP Advice 译: 9.5.1安全和AOP建议

如果你熟悉AOP，你会意识到有不同类型的建议可供选择：在之前，之后，投掷和周围。周围的建议非常有用，因为顾问可以选择是否继续进行方法调用，是否修改响应以及是否抛出异常。Spring Security为方法调用和Web请求提供了一个周围的建议。我们使用Spring的标准AOP支持为方法调用提供了全面的建议，并且我们使用标准过滤器为Web请求提供了全面的建议。

对于那些不熟悉AOP的人来说，理解的关键是Spring Security可以帮助您保护方法调用以及Web请求。大多数人都对保护其服务层上的方法调用感兴趣。这是因为服务层是大多数业务逻辑驻留在当代Java EE应用程序中的地方。如果你只需要在服务层保证方法调用的安全，Spring的标准AOP就足够了。如果您需要直接保护域对象，您可能发现AspectJ值得考虑。

您可以选择使用AspectJ或Spring AOP执行方法授权，也可以选择使用过滤器执行Web请求授权。您可以一起使用零个，一个，两个或三个这些方法。主流使用模式是执行一些Web请求授权，再加上服务层上的一些Spring AOP方法调用授权。

9.5.2 Secure Objects and the AbstractSecurityInterceptor 译: 9.5.2安全对象和AbstractSecurityInterceptor

那么什么是“安全对象”呢？Spring Security使用这个术语来指代可以有安全性的任何对象（例如授权决定）。最常见的例子是方法调用和Web请求。

每个受支持的安全对象类型都有其自己的拦截器类，它是AbstractSecurityInterceptor的子类。重要的是，通过时间AbstractSecurityInterceptor被调用时，SecurityContextHolder将包含一个有效的Authentication如果主体已经通过认证。

AbstractSecurityInterceptor为处理安全对象请求提供了一致的工作流程，通常为：

1. Look up the "configuration attributes" associated with the present request
2. Submitting the secure object, current Authentication and configuration attributes to the AccessDecisionManager for an authorization decision
3. Optionally change the Authentication under which the invocation takes place
4. Allow the secure object invocation to proceed (assuming access was granted)
5. Call the AfterInvocationManager if configured, once the invocation has returned. If the invocation raised an exception, the AfterInvocationManager will not be invoked.

What are Configuration Attributes? 译: 什么是配置属性？

“配置属性”可以被认为是对AbstractSecurityInterceptor使用的类具有特殊含义的字符串。它们由框架内的接口ConfigAttribute表示。它们可能是简单的角色名称或具有更复杂的含义，这取决于该多么复杂AccessDecisionManager实现。AbstractSecurityInterceptor配置了一个SecurityMetadataSource，它用它来查找安全对象的属性。通常这个配置对用户是隐藏的。配置属性将作为安全方法的注释或安全URL上的访问属性输入。例如，当我们在名称空间介绍中看到类似<intercept-url pattern='/secure/**' access='ROLE_A,ROLE_B' />东西时，这表示配置属性ROLE_A和ROLE_B适用于与给定模式匹配的Web请求。实际上，使用默认的AccessDecisionManager配置，这意味着任何具有匹配这两个属性的任何人的GrantedAuthority都将被允许访问。严格来说，它们只是属性，解释取决于AccessDecisionManager实现。使用前缀ROLE_是一个标记，用于表示这些属性是角色，并且应该由Spring Security的RoleVoter。这只有在使用基于选民的AccessDecisionManager。WEA™会看到如何AccessDecisionManager在实施authorization chapter。

RunAsManager 译: RunAsManager

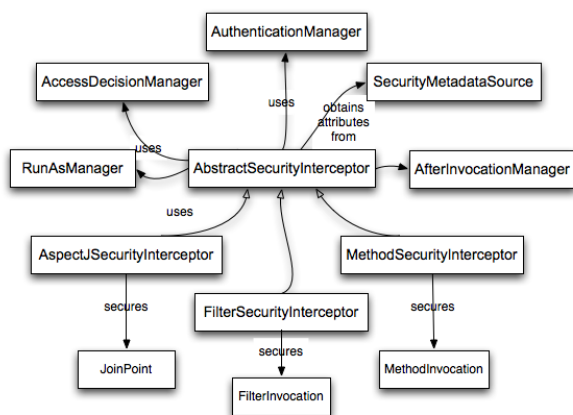
假设AccessDecisionManager决定允许请求，那么AbstractSecurityInterceptor通常只会处理请求。话虽如此，在极少数情况下用户可能要更换Authentication内部SecurityContext用不同Authentication，这是由处理AccessDecisionManager调用RunAsManager。在合理的异常情况下，这可能非常有用，例如，如果服务层方法需要调用远程系统并呈现不同的身份。由于Spring Security自动将安全身份从一台服务器传播到另一台服务器（假设您正在使用正确配置的RMI或HttpInvoker远程协议客户端），这可能会很有用。

AfterInvocationManager 译: AfterInvocationManager

在安全对象调用过程之后，然后返回 - 这可能意味着方法调用完成或过滤器链处理 - AbstractSecurityInterceptor最终有机会处理调用。在这个阶段，AbstractSecurityInterceptor对可能修改返回对象感兴趣。我们可能希望发生这种情况，因为授权决策不能在安全对象调用的“途中”中进行。AbstractSecurityInterceptor具有很强的可插拔性，如果需要，AbstractSecurityInterceptor会将控制权交给AfterInvocationManager以实际修改对象。这个类甚至可以完全替换对象，或者抛出异常，或者不以任何方式改变它。只有调用成功时才会执行调用后检查。如果发生异常，则额外的检查将被跳过。

AbstractSecurityInterceptor及其相关对象显示在 Figure 9.1, “Security interceptors and the “secure object” model”中

图9.1. 安全拦截器和“安全对象”模型



Extending the Secure Object Model 译: 扩展安全对象模型

只有开发人员想要采用全新的拦截和授权请求的方式，才需要直接使用安全对象。例如，可以构建一个新的安全对象来保护对消息系统的调用。任何需要安全性并且还提供拦截呼叫的方式（如围绕通知语义的AOP）都可以被制作成安全对象。话虽如此，大多数Spring应用程序将完全透明地使用目前支持的三种安全对象类型（AOP Alliance MethodInvocation，AspectJ JoinPoint和Web请求 FilterInvocation）。

9.6 Localization 译: 9.6本地化

Spring Security支持终端用户可能看到的异常消息的本地化。如果您的应用程序是为讲英语的用户设计的，则默认情况下，您不需要执行任何操作，所有安全信息都是英文的。如果您需要支持其他语言环境，则需要了解的所有内容都包含在本节中。

所有异常消息都可以本地化，包括与认证失败和访问被拒绝有关的消息（授权失败）。专注于开发人员或系统部署人员的异常和日志消息（包括不正确的属性，接口合同违规，使用不正确的构造函数，启动时间验证，调试级别日志记录）未本地化，而是在Spring Security中以英文硬编码代码。

在`spring-security-core-xx.jar`您会发现一个`org.springframework.security`包，该包中包含`messages.properties`文件以及一些常用语言的本地化版本。这应该被你的`ApplicationContext`，因为Spring Security类实现了Spring的`MessageSourceAware`接口，并期望消息解析器在应用程序上下文启动时被依赖注入。通常，您只需在应用程序上下文中注册一个bean来引用这些消息。一个例子如下所示：

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

`messages.properties`按照标准资源包进行命名，并表示Spring Security消息支持的默认语言。这个默认文件是英文的。

如果您希望自定义`messages.properties`文件或支持其他语言，则应该复制该文件并相应地对其重命名，并在上面的bean定义中注册它。此文件中没有大量的消息密钥，因此本地化不应被视为主要举措。如果您确实执行了此文件的本地化，请考虑通过记录JIRA任务并附上适当命名的本地化版本`messages.properties`与社区分享您的工作。

Spring Security依靠Spring的本地化支持来实际查找适当的消息。为了使其发挥作用，您必须确保来自传入请求的语言环境存储在Spring的`org.springframework.context.i18n.LocaleContextHolder`。春天MVCâ€™的`DispatcherServlet`这是否为您的应用程序自动，但由于春节Securityâ€™™的过滤器在此之前被调用，`LocaleContextHolder`要建立需要包含正确`Locale`被称为过滤器之前。你可以自己做一个过滤器（它必须在`web.xml`的Spring Security过滤器`web.xml`），或者你可以使用Spring的`RequestContextFilter`。有关在Spring中使用本地化的更多详细信息，请参阅Spring Framework文档。

“联系人”示例应用程序设置为使用本地化的消息。

10. Core Services 译：10核心服务

现在，我们对Spring Security的架构和核心类，莱塔€™的高层次概述需要在一个或两个核心接口及其实现的仔细看看，特别是`AuthenticationManager`，`UserDetailsService`和`AccessDecisionManager`。这些文件会在本文的其余部分定期出现，所以重要的是你知道它们是如何配置以及它们如何操作的。

10.1 The AuthenticationManager, ProviderManager and AuthenticationProvider 译：10.1 AuthenticationManager, ProviderManager和AuthenticationProvider

`AuthenticationManager`只是一个接口，所以实现可以是我們选择的任何东西，但它在实践中是如何工作的？如果我们需要检查多个身份验证数据库或不同的身份验证服务（如数据库和LDAP服务器）的组合，该怎么办？

Spring Security中的默认实现被称为`ProviderManager`，而不是自己处理认证请求，它将委托给已配置的`AuthenticationProvider`的列表，每个列表依次查询是否可以执行认证。每个提供者都将抛出一个异常或返回一个完全填充的`Authentication`对象。记得我们的好朋友，`UserDetails`和`UserDetailsService`？如果不是，请回到上一章并刷新你的记忆。验证身份验证请求的最常见方法是加载相应的`UserDetails`并检查加载的密码与用户输入的密码。这是`DaoAuthenticationProvider`使用的方法（见下文）。装入的`UserDetails`对象-特别是它包含的`GrantedAuthority`-将用于构建从成功验证返回并存储在`SecurityContext`的完全填充的`Authentication`对象。

如果使用的是名称空间，则会在内部创建并维护一个`ProviderManager`的实例，并使用名称空间身份验证提供程序元素（请参见the namespace chapter）向其添加提供程序。在这种情况下，您不应该在应用程序上下文中声明`ProviderManager` bean。但是，如果您不使用名称空间，那么您会声明它如下所示：

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

在上面的例子中，我们三个提供者。它们按所示顺序进行尝试（使用`List`），每个提供程序都可以尝试进行身份验证，或者通过简单地返回`null`跳过身份验证。如果所有实现都返回null，则`ProviderManager`将抛出`ProviderNotFoundException`。如果您想了解更多关于链接提供商的信息，请参阅`ProviderManager` Javadoc。

身份验证机制（例如Web表单登录处理过滤器）通过引用`ProviderManager`注入，并将调用它来处理其身份验证请求。您需要的提供者有时可以与认证机制互换，而在其他时候，他们将依赖于特定的认证机制。例如，`DaoAuthenticationProvider`和`LdapAuthenticationProvider`与任何提交简单用户名/密码认证请求的机制兼容，因此可以与基于表单的登录或HTTP基本认证一起使用。另一方面，某些认证机制会创建一个认证请求对象，该对象只能由单一类型的`AuthenticationProvider`。一个例子是JA-SIG CAS，它使用服务票据的概念，因此只能通过`CasAuthenticationProvider`进行验证。您不必太担心这一点，因为如果您忘记注册合适的提供商，只需在尝试进行身份验证时收到`ProviderNotFoundException`即可。

10.1.1 Erasing Credentials on Successful Authentication 译：10.1.1成功验证时删除证书

默认情况下（从Spring Security 3.1开始），`ProviderManager`将尝试清除成功的身份验证请求返回的`Authentication`对象中的任何敏感凭据信息。这可以防止密码等信息被保留超过必要的时间。

例如，当您使用用户对象缓存时，这可能会导致问题，以提高无状态应用程序的性能。如果`Authentication`包含对高速缓存中对象的引用（例如`UserDetails`实例），并且已删除其凭据，则无法再对缓存的值进行身份验证。如果您使用缓存，则需要考虑这一点。一个显而易见的解决方案是使对象的副本第一，无论是在缓存实现或`AuthenticationProvider`它创建返回`Authentication`对象。另外，您也可以禁用`eraseCredentialsAfterAuthentication` 物业`ProviderManager`。有关更多信息，请参阅Javadoc。

10.1.2 DaoAuthenticationProvider 译：10.1.2 DaoAuthenticationProvider

Spring Security最简单的`AuthenticationProvider`是`DaoAuthenticationProvider`，这也是该框架最早支持的之一。它利用`UserDetailsService`（作为DAO）来查找用户名，密码和`GrantedAuthority`。它只是通过比较提交的密码验证用户`UsernamePasswordAuthenticationToken`反对通过加载一个`UserDetailsService`。配置提供者非常简单：

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

`PasswordEncoder` 是可选的。`PasswordEncoder` 提供 `UserDetails` 对象中显示的密码的编码和解码，该对象从配置的 `UserDetailsService` 返回。这将更详细地讨论 [below](#)。

10.2 UserDetailsService Implementations 译：10.2 UserDetailsService 实现

如本参考指南前面所述，大多数身份验证提供程序利用 `UserDetails` 和 `UserDetailsService` 接口。回想一下 `UserDetailsService` 的合同是一种单一的方法：

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

返回的 `UserDetails` 是一个提供getter的接口，用于保证身份验证信息（如用户名，密码，授予的权限）以及用户帐户是否已启用或禁用等非空提供。即使用户名和密码并未实际用作身份验证决策的一部分，大多数身份验证提供程序也会使用 `UserDetailsService`。他们可能使用返回的 `UserDetails` 对象仅仅为了它的 `GrantedAuthority` 信息，因为一些其他系统（如LDAP或X.509或CAS等）承担了实际验证凭证的责任。

鉴于 `UserDetailsService` 实现非常简单，用户可以很容易地使用他们选择的持久性策略来检索认证信息。话虽如此，Spring Security确实包含了一些有用的基础实现，我们将在下面进行介绍。

10.2.1 In-Memory Authentication 译：10.2.1 内存中验证

易于使用创建自定义的 `UserDetailsService` 实现，从所选择的持久性引擎中提取信息，但许多应用程序不需要这种复杂性。这是尤其如此，如果有啊™重新建立一个原型应用程序或刚刚开始集成Spring Security的，当你唐娜™吨真的想花时间配置数据库或写 `UserDetailsService` 实现。对于这种情况，一个简单的选择是使用来自安全 `namespace` 的 `user-service` 元素：

```
<user-service id="userDetailsService">
  <!-- Password is prefixed with {noop} to indicate to DelegatingPasswordEncoder that
  NoOpPasswordEncoder should be used. This is not safe for production, but makes reading
  in samples easier. Normally passwords should be hashed using BCrypt -->
  <user name="jimi" password="{noop}jimispasword" authorities="ROLE_USER, ROLE_ADMIN" />
  <user name="bob" password="{noop}bobspassword" authorities="ROLE_USER" />
</user-service>
```

这也支持使用外部属性文件：

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

属性文件应该包含表单中的条目

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

例如

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

10.2.2 JdbcDaoImpl 译：10.2.2 JdbcDaoImpl

Spring Security还包含一个 `UserDetailsService`，它可以从JDBC数据源获取认证信息。使用内部Spring JDBC，因此避免了用于存储用户详细信息的全功能对象关系映射程序（ORM）的复杂性。如果您的应用程序确实使用了ORM工具，那么您可能更愿意编写自定义 `UserDetailsService` 以重用您可能已经创建的映射文件。返回到 `JdbcDaoImpl`，示例配置如下所示：

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

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

您可以通过修改 `DriverManagerDataSource` 显示的 `DriverManagerDataSource` 来使用不同的关系数据库管理系统。您也可以像使用其他Spring配置一样使用从JNDI获取的全局数据源。

Authority Groups 译：权限组

默认情况下，`JdbcDaoImpl` 为一个用户加载权限，并假定权限直接映射到用户（请参阅 [database schema appendix](#)）。另一种方法是将权限分为组并将组分配给用户。有些人更喜欢这种方法作为管理用户权利的手段。有关如何启用组权限的更多信息，请参阅 `JdbcDaoImpl` Javadoc。组模式也包含在附录中。

10.3 Password Encoding 译：10.3 密码编码

Spring Security的 `PasswordEncoder` 界面用于执行密码的单向转换，以便安全地存储密码。鉴于 `PasswordEncoder` 是一种单向转换，当密码转换需要两种方式（即存储用于向数据库进行身份验证的凭证）时，并不打算这样做。通常，`PasswordEncoder` 用于存储需要在验证时与用户提供的密码进行比较的密码。

10.3.1 Password History 译：10.3.1 密码历史记录

多年来，用于存储密码的标准机制已经发展。开始时密码以纯文本形式存储。密码被认为是安全的，因为数据存储密码被保存在所需的凭证中以便访问它。但是，恶意图户可以通过SQL注入等攻击找到方法来获取用户名和密码的大量“数据转储”。随着越来越多的用户凭证成为公安专家意识到我们需要做更多的事情来保护用户的密码。

然后鼓励开发人员在通过单向散列（如SHA-256）运行密码后存储密码。当用户尝试认证时，散列密码将与他们键入的密码的散列进行比较。这意味着系统只需要存储密码的单向散列。如果发生了违规，那么只有密码的单向散列被暴露。由于哈希是一种方法，并且在计算上很难猜测给定散列的密码，所以在系统中找出每个密码是不值得的。为了击败这个新系统，恶意用户决定创建名为Rainbow Tables的查找表。他们不是每次都在猜测每个密码，而是一次计算密码并将其存储在查找表中。

为了降低彩虹表的有效性，鼓励开发者使用咸味密码。而不是仅使用密码作为散列函数的输入，将为每个用户的密码生成随机字节（称为salt）。salt和用户的密码将通过产生唯一散列的散列函数运行。盐将以明文形式存储在用户的密码旁边。然后当用户尝试认证时，散列密码将与存储的盐的哈希以及他们输入的密码进行比较。独特的盐意味着Rainbow Tables不再有效，因为每种盐和密码组合的散列值都不相同。

在现代，我们意识到密码哈希（如SHA-256）不再安全。原因是，使用现代硬件，我们可以每秒执行数十亿次哈希计算。这意味着我们可以轻松破解每个密码。

现在鼓励开发人员利用自适应单向函数来存储密码。使用自适应单向函数对密码进行验证是有意识的资源（即CPU，内存等）。自适应单向函数允许配置随硬件变得越来 越好的“工作因素”。建议将“工作因素”调整为在您的系统上验证密码需要大约1秒钟的时间。这种权衡是为了让攻击者难以破解密码，但并不那么昂贵，它会给你自己的系统带来过大的负担。Spring Security试图为“工作因素”提供一个良好的起点，但鼓励用户为他们自己的系统定制“工作因素”，因为不同系统的性能会有很大差异。应使用的自适应单向函数的示例包括bcrypt，PBKDF2，scrypt，和Argon2。

由于自适应单向函数是故意耗费资源的，因此验证每个请求的用户名和密码将显着降低应用程序的性能。Spring Security（或任何其他库）没有什么能够加速验证密码，因为通过使验证资源密集而获得安全性。鼓励用户交换短期凭证（即会话，OAuth令牌等）的长期凭证（即用户名和密码）。短期凭证可以快速验证而不会有任何安全性损失。

10.3.2 DelegatingPasswordEncoder 译: 10.3.2委托 密码编码器

在Spring Security 5.0之前，默认的 PasswordEncoder 是 NoOpPasswordEncoder，它需要纯文本密码。根据 Password History 部分，您可能会预计默认的 PasswordEncoder 现在类似 BCryptPasswordEncoder。但是，这忽略了三个现实世界的问题：

- There are many applications using old password encodings that cannot easily migrate
- The best practice for password storage will change again.
- As a framework Spring Security cannot make breaking changes frequently

相反，Spring Security引入了 DelegatingPasswordEncoder，它解决了所有的问题：

- Ensuring that passwords are encoded using the current password storage recommendations
- Allowing for validating passwords in modern and legacy formats
- Allowing for upgrading the encoding in the future

您可以使用 PasswordEncoderFactories 轻松构建 DelegatingPasswordEncoder 的实例。

```
PasswordEncoder passwordEncoder =
    PasswordEncoderFactories.createDelegatingPasswordEncoder();
```

或者，您可以创建自己的自定义实例。例如：

```
String idForEncode = "bcrypt";
Map encoders = new HashMap<>();
encoders.put(idForEncode, new BCryptPasswordEncoder());
encoders.put("noop", NoOpPasswordEncoder.getInstance());
encoders.put("pbkdf2", new Pbkdf2PasswordEncoder());
encoders.put("scrypt", new SCryptPasswordEncoder());
encoders.put("sha256", new StandardPasswordEncoder());

PasswordEncoder passwordEncoder =
    new DelegatingPasswordEncoder(idForEncode, encoders);
```

Password Storage Format 译: 密码存储格式


密码的一般格式是：

```
{id}encodedPassword
```

id 是一个标识符，用于查找应使用哪个 PasswordEncoder 并且 encodedPassword 是所选 PasswordEncoder 的原始编码密码。id 必须在密码的开头，以 { 并以 }。如果 id，则 id 将为空。例如，以下可能是使用不同的 id 编码的密码列表。所有原始密码都是“密码”。

```
{bcrypt}$2a$10$dXJ3SW6G7P50lGmMkkmwe.20cQQubK3.HZWzG3YB1t1Ry.fqvM/BG ❶
{noop}password ❷
{pbkdf2}5d923b44a6d129f3ddf3e3c8d29412723dcbde72445e8ef6bf3b508fbf17fa4ed4d6b99ca763d8dc ❸
{scrypt}$e0801$8bwJaSu2IKSn9Z9kM+TPXF0c/9bdYSrN1oD9qfVThWEwdRTn07re7Ei+fUZRJ68k9lTyuTeUp4of4g24hHnazw==$0A0ec05+bXxvuu/1qZ6NUR+xQYvYv7BeL1QxwRpY5Pc=
{sha256}97cde38028ad898ebc02e690819fa220e88c62e0699403e94fff291cfffaf8410849f27605abcbcb0 ❹
```

- ❶ 第一个密码的编号为 bcrypt，编号为 bcrypt，PasswordEncoder 密码编号为 \$2a\$10\$dXJ3SW6G7P50lGmMkkmwe.20cQQubK3.HZWzG3YB1t1Ry.fqvM/BG。匹配时将委托给 BCryptPasswordEncoder
- ❷ 第二个密码的编号为 noop，编号为 noop，PasswordEncoder 密码编号为 password。匹配时将委托给 NoOpPasswordEncoder
- ❸ 第三个密码的编号为 pbkdf2，编号为 pbkdf2，PasswordEncoder 密码为 5d923b44a6d129f3ddf3e3c8d29412723dcbde72445e8ef6bf3b508fbf17fa4ed4d6b99ca763d8dc。匹配时将委托给 Pbkdf2PasswordEncoder
- ❹ 第四个密码的编号为 scrypt，编号为 scrypt，PasswordEncoder 密码为 \$e0801\$8bwJaSu2IKSn9Z9kM+TPXF0c/9bdYSrN1oD9qfVThWEwdRTn07re7Ei+fUZRJ68k9lTyuTeUp4of4g24hHnazw==\$0A0ec05+bXxvuu/1qZ6NUR+xQYvYv7BeL1QxwRpY5Pc=。匹配时，它将委托给 SCryptPasswordEncoder
- ❺ 最终密码的编号为 sha256，编号为 sha256，PasswordEncoder 密码编号为 97cde38028ad898ebc02e690819fa220e88c62e0699403e94fff291cfffaf8410849f27605abcbcb0。匹配时将委托给 StandardPasswordEncoder



一些用户可能会担心存储格式是为潜在的黑客提供的。这不是一个问题，因为密码的存储不依赖于算法是一个秘密。此外，大多数格式都很容易让攻击者找出没有前缀的地方。例如，BCrypt密码通常以 \$2a\$。

Password Encoding 译: 密码编码

传递给构造函数的 idForEncode 确定哪个 PasswordEncoder 将用于编码密码。在 DelegatingPasswordEncoder 我们上述结构，这意味着编码结果 password 将被委派给 BCryptPasswordEncoder 并用前缀 {bcrypt}。最终结果如下所示：


```
{bcrypt}$2a$10$dXJ3SW6G7P50lGmMkkmwe.20cQQubK3.HZWzG3YB1t1Ry.fqvM/BG
```

Password Matching 译:密码匹配

匹配是基于完成 `{id}` 和映射 `id` 到 `PasswordEncoder` 在构造函数中提供。我们的例子 [the section called "Password Storage Format"](#) 提供了一个如何完成这个工作的实例。默认情况下,使用密码调用 `matches(CharSequence, String)` 和未映射的 `id` (包括空id) 的结果将导致 `IllegalArgumentException`。这种行为可以使用 `DelegatingPasswordEncoder.setDefaultPasswordEncoderForMatches>PasswordEncoder)` 自定义。

通过使用 `id` 我们可以匹配任何密码编码,但使用最现代的密码编码对密码进行编码。这很重要,因为与加密不同,密码哈希的设计使得没有简单的方法来恢复明文。由于无法恢复明文,因此难以迁移密码。虽然用户迁移 `NoOpPasswordEncoder` 很简单,但我们选择默认包含它以简化入门体验。

Getting Started Experience 译:入门体验

如果您正在制作演示或样本,花时间散列用户的密码会有点麻烦。有便利的机制可以使这更容易,但这仍然不适用于生产。

```
User user = User.withDefaultPasswordEncoder()
    .username("user")
    .password("password")
    .roles("user")
    .build();
System.out.println(user.getPassword());
// {bcrypt}$2a$10$dXJ3SW6G7P50lGmMkkmwe.20cQQubK3.HZWzG3YB1t1Ry.fqvM/BG
```

如果您正在创建多个用户,则还可以重新使用该构建器。

```
UserBuilder users = User.withDefaultPasswordEncoder();
User user = users
    .username("user")
    .password("password")
    .roles("USER")
    .build();
User admin = users
    .username("admin")
    .password("password")
    .roles("USER", "ADMIN")
    .build();
```

这确实散列了存储的密码,但密码仍在内存和编译后的源代码中公开。因此,对于生产环境来说,它仍然不被认为是安全的。对于生产,你应该在外部散列你的密码。

Troubleshooting 译:故障解答

如 [the section called "Password Storage Format"](#) 中所述,存储的密码之一没有标识时会发生以下错误。

```
java.lang.IllegalArgumentException: There is no PasswordEncoder mapped for the id "null"
at org.springframework.security.crypto.password.DelegatingPasswordEncoder$UnmappedIdPasswordEncoder.matches(DelegatingPasswordEncoder.java:233)
at org.springframework.security.crypto.password.DelegatingPasswordEncoder.matches(DelegatingPasswordEncoder.java:196)
```

解决错误的最简单方法是切换到明确提供密码编码的 `PasswordEncoder`。解决这个问题最简单方法是弄清楚你的密码当前如何存储,并明确提供正确的 `PasswordEncoder`。如果您正在从Spring Security 4.2.x迁移,您可以通过公开 `NoOpPasswordEncoder` bean来恢复到以前的行为。例如,如果您正在使用Java配置,则可以创建如下所示的配置:

❗ 恢复到 `NoOpPasswordEncoder` 并不被认为是安全的。您应该迁移到使用 `DelegatingPasswordEncoder` 来支持安全的密码编码。

```
@Bean
public static NoOpPasswordEncoder passwordEncoder() {
    return NoOpPasswordEncoder.getInstance();
}
```

如果您使用XML配置,您可以暴露 `PasswordEncoder` id为 `passwordEncoder` :

```
<b:bean id="passwordEncoder"
    class="org.springframework.security.crypto.password.NoOpPasswordEncoder" factory-method="getInstance"/>
```

或者,您可以使用正确的ID为所有密码加前缀并继续使用 `DelegatingPasswordEncoder`。例如,如果您使用的是BCrypt,则可以将密码从以下类似的位置迁移:

```
$2a$10$dXJ3SW6G7P50lGmMkkmwe.20cQQubK3.HZWzG3YB1t1Ry.fqvM/BG
```

至

```
{bcrypt}$2a$10$dXJ3SW6G7P50lGmMkkmwe.20cQQubK3.HZWzG3YB1t1Ry.fqvM/BG
```

有关映射的完整列表,请参阅 [PasswordEncoderFactories](#) 上的Javadoc。

10.3.3 BCryptPasswordEncoder 译:10.3.3 BCryptPasswordEncoder

`BCryptPasswordEncoder` 实现使用广泛支持的**bcrypt**算法来散列密码。为了使它对密码破解更具抵抗性,bcrypt故意缓慢。与其他自适应单向函数一样,应该调整大约1秒以验证系统上的密码。

```
// Create an encoder with strength 16
BCryptPasswordEncoder encoder = new BCryptPasswordEncoder(16);
String result = encoder.encode("myPassword");
assertTrue(encoder.matches("myPassword", result));
```

10.3.4 Pbkdf2PasswordEncoder 译:10.3.4 Pbkdf2PasswordEncoder

`Pbkdf2PasswordEncoder` 实现使用**PBKDF2**算法对密码进行哈希处理。为了破解密码破解PBKDF2是一个故意缓慢的算法。与其他自适应单向函数一样,应该调整大约

1秒以验证系统上的密码。当需要FIPS认证时，此算法是一个不错的选择。

```
// Create an encoder with all the defaults
Pbkdf2PasswordEncoder encoder = new Pbkdf2PasswordEncoder();
String result = encoder.encode("myPassword");
assertTrue(encoder.matches("myPassword", result));
```

10.3.5 SCryptPasswordEncoder 译: 10.3.5 SCryptPasswordEncoder

`SCryptPasswordEncoder` 实现使用 `scrypt` 算法对密码进行哈希处理。为了击败定制硬件上的密码破解 `scrypt` 是一个故意缓慢的算法，需要大量的内存。与其他自适应单向函数一样，应该调整大约1秒以验证系统上的密码。

```
// Create an encoder with all the defaults
SCryptPasswordEncoder encoder = new SCryptPasswordEncoder();
String result = encoder.encode("myPassword");
assertTrue(encoder.matches("myPassword", result));
```

10.3.6 Other PasswordEncoders 译: 10.3.6 其他 PasswordEncoders

有大量的其他 `PasswordEncoder` 实现完全为了向后兼容性而存在。他们都被弃用，表明他们不再被认为是安全的。但是，由于难以迁移现有的遗留系统，因此没有计划将其删除。

10.4 Jackson Support 译: 10.4 杰克逊支持

Spring Security已经增加了Jackson Support来坚持Spring Security相关的类。 这可以提高与分布式会话（即会话复制，Spring会话等）工作时序列化Spring Security相关类的性能。

要使用它，请将 `JacksonJacksonModules.getModules(ClassLoader)` 注册为 `Jackson Modules`。

```
ObjectMapper mapper = new ObjectMapper();
ClassLoader loader = getClass().getClassLoader();
List<Module> modules = SecurityJackson2Modules.getModules(loader);
mapper.registerModules(modules);

// ... use ObjectMapper as normally ...
SecurityContext context = new SecurityContextImpl();
// ...
String json = mapper.writeValueAsString(context);
```

Part III. Testing 译: 第三部分。测试

本节介绍Spring Security提供的测试支持。



要使用Spring Security测试支持，您必须包含 `spring-security-test-5.1.0.M1.jar` 作为项目的依赖项。

11. Testing Method Security 译: 11 测试方法安全性

本节演示如何使用Spring Security的测试支持来测试基于安全性的方法。我们首先介绍一个 `MessageService`，要求用户进行身份验证才能访问它。

```
public class HelloMessageService implements MessageService {

    @PreAuthorize("authenticated")
    public String getMessage() {
        Authentication authentication = SecurityContextHolder.getContext()
            .getAuthentication();
        return "Hello " + authentication;
    }
}
```

`getMessage` 的结果是一个字符串，对当前的Spring Security `Authentication` 说“你好”。输出示例如下所示。

```
Hello org.springframework[email protected]ca25360: Principal: [email protected]: Username: user; Password: [PROTECTED]; Enabled: true; AccountNonExpired
```

11.1 Security Test Setup 译: 11.1 安全测试设置

在我们使用Spring Security Test支持之前，我们必须执行一些设置。下面是一个例子：

```
@RunWith(SpringJUnit4ClassRunner.class) ❶
@ContextConfiguration ❷
public class WithMockUserTests {
```

这是如何设置Spring Security Test的基本示例。亮点是：

- ❶ `@RunWith` 指示弹簧测试模块它应该创建一个 `ApplicationContext`。这与使用现有的Spring Test支持没有区别。有关更多信息，请参阅[Spring Reference](#)
- ❷ `@ContextConfiguration` 指示弹簧测试用于创建 `ApplicationContext` 的配置。由于没有指定配置，因此将尝试默认配置位置。这与使用现有的Spring Test支持没有区别。有关更多信息，请参阅[Spring Reference](#)



Spring Security使用`WithSecurityContextTestExecutionListener`挂钩到Spring Test支持，这将确保我们的测试与正确的用户一起运行。它通过在运行我们的测试之前填充`SecurityContextHolder`来完成此操作。测试完成后，将清除`SecurityContextHolder`。如果您只需要Spring Security相关支持，则可以用`@ContextConfiguration`替换`@SecurityTestExecutionListeners`。

请记住，我们将`@PreAuthorize`注释添加到我们的`HelloMessageService`，因此它需要经过身份验证的用户才能调用它。如果我们进行了以下测试，我们预计将通过以下测试：

```
@Test(expected = AuthenticationCredentialsNotFoundException.class)
public void getMessageUnauthenticated() {
    messageService.getMessage();
}
```

11.2 @WithMockUser #: 11.2@WithMockUser

问题是“我们怎样才能最轻松地以特定用户身份运行测试？”答案是使用`@WithMockUser`。以下测试将以用户名“user”，密码“password”和角色“ROLE_USER”的身份运行。

```
@Test
@WithMockUser
public void getMessageWithMockUser() {
    String message = messageService.getMessage();
    ...
}
```

具体如下：

- The user with the username "user" does not have to exist since we are mocking the user
- The `Authentication` that is populated in the `SecurityContext` is of type `UsernamePasswordAuthenticationToken`
- The principal on the `Authentication` is Spring Security's `User` object
- The `User` will have the username of "user", the password "password", and a single `GrantedAuthority` named "ROLE_USER" is used.

我们的例子很好，因为我们可以利用很多默认值。如果我们想用其他用户名运行测试，该怎么办？以下测试将使用用户名“customUser”运行。同样，用户不需要实际存在。

```
@Test
@WithMockUser("customUsername")
public void getMessageWithMockUserCustomUsername() {
    String message = messageService.getMessage();
    ...
}
```

我们也可以轻松定制角色。例如，将使用用户名“admin”和角色“ROLE_USER”和“ROLE_ADMIN”来调用此测试。

```
@Test
@WithMockUser(username="admin",roles={"USER","ADMIN"})
public void getMessageWithMockUserCustomUser() {
    String message = messageService.getMessage();
    ...
}
```

如果我们不希望该值自动以ROLE_开头，我们可以使用权限属性。例如，将使用用户名“admin”和权限“USER”和“ADMIN”调用此测试。

```
@Test
@WithMockUser(username = "admin", authorities = { "ADMIN", "USER" })
public void getMessageWithMockUserCustomAuthorities() {
    String message = messageService.getMessage();
    ...
}
```

当然，将注释放在每种测试方法上可能有点乏味。相反，我们可以在类级别放置注释，并且每个测试都将使用指定的用户。例如，以下内容将以用户名“admin”，密码“password”和角色“ROLE_USER”和“ROLE_ADMIN”运行每个测试。

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration
@WithMockUser(username="admin",roles={"USER","ADMIN"})
public class WithMockUserTests {
```

默认情况下，`SecurityContext`在`TestExecutionListener.beforeTestMethod`事件中设置。这与JUnit `@Before`之前的情况`@Before`。您可以将此更改为`TestExecutionListener.beforeTestExecution`事件期间发生的事件，该事件位于JUnit的`@Before`但在调用测试方法之前。

```
@WithMockUser(setupBefore = TestExecutionEvent.TEST_EXECUTION)
```

11.3 @WithAnonymousUser #: 11.3@WithAnonymousUser

使用`@WithAnonymousUser`允许以匿名用户`@WithAnonymousUser`运行。当您希望使用特定用户运行大多数测试，但希望以匿名用户身份运行一些测试时，这样做尤其方便。例如，以下内容将与MockUser1和MockUser2一起使用`@WithMockUser`和匿名匿名用户运行。


```

@RunWith(SpringJUnit4ClassRunner.class)
@WithMockUser
public class WithUserClassLevelAuthenticationTests {

    @Test
    public void withMockUser1() {
    }

    @Test
    public void withMockUser2() {
    }

    @Test
    @WithAnonymousUser
    public void anonymous() throws Exception {
        // override default to run as anonymous user
    }
}

```

默认情况下，`SecurityContext` 在过程中设置 `TestExecutionListener.beforeTestMethod` 事件。这与JUnit `@Before` 之前的情况 `@Before`。您可以将此更改为 `TestExecutionListener.beforeTestExecution` 事件期间发生的事件，该事件位于JUnit的 `@Before` 但在调用测试方法之前。

```

@WithAnonymousUser(setupBefore = TestExecutionEvent.TEST_EXECUTION)

```

11.4 @WithUserDetails

#: 11.4 @WithUserDetails

虽然 `@WithMockUser` 是一种非常方便的入门方式，但它可能无法在所有情况下都能正常工作。例如，应用程序期望 `Authentication` 委托人具有特定类型是很常见的。这样做是为了让应用程序可以将主体引用为自定义类型，并减少Spring Security上的耦合。

自定义主体通常由自定义 `UserDetailsService` 返回，该自定义返回实现 `UserDetails` 和自定义类型的对象。对于这样的情况，使用自定义 `UserDetailsService` 创建测试用户很有用。这正是 `@WithUserDetails` 所做的。

假设我们有一个 `UserDetailsService` 暴露为豆，下面的测试将与被调用 `Authentication` 类型的 `UsernamePasswordAuthenticationToken` 和从返回的一个主要 `UserDetailsService` 与“用户”的用户名。

```

@Test
@WithUserDetails
public void getMessageWithUserDetails() {
    String message = messageService.getMessage();
    ...
}

```

我们还可以自定义用于从我们的 `UserDetailsService` 查找用户的用户名。例如，此测试将使用用户名“customUsername”的 `UserDetailsService` 返回的主体执行。

```

@Test
@WithUserDetails("customUsername")
public void getMessageWithUserDetailsCustomUsername() {
    String message = messageService.getMessage();
    ...
}

```

我们也可以提供一个明确的bean名称来查找 `UserDetailsService`。例如，此测试将使用 `UserDetailsService` 和bean名称“myUserDetailsService”查找“customUsername”的用户名。

```

@Test
@WithUserDetails(value="customUsername", userDetailsServiceBeanName="myUserDetailsService")
public void getMessageWithUserDetailsServiceBeanName() {
    String message = messageService.getMessage();
    ...
}

```

像 `@WithMockUser` 一样，我们也可以将我们的注释放在课程级别，以便每个测试都使用同一个用户。然而，与 `@WithMockUser` 不同，`@WithUserDetails` 要求用户存在。

默认情况下，`SecurityContext` 在过程中设置 `TestExecutionListener.beforeTestMethod` 事件。这与JUnit `@Before` 之前的情况 `@Before`。您可以将此更改为 `TestExecutionListener.beforeTestExecution` 事件期间发生的事件，该事件位于JUnit的 `@Before` 但在调用测试方法之前。

```

@WithUserDetails(setupBefore = TestExecutionEvent.TEST_EXECUTION)

```

11.5 @WithSecurityContext

#: 11.5 @WithSecurityContext

我们已经看到，如果我们不使用自定义的 `Authentication` 委托人，`@WithMockUser` 是一个很好的选择。接下来我们发现 `@WithUserDetails` 将允许我们使用自定义 `UserDetailsService` 来创建我们的 `Authentication` 主体，但需要用户存在。我们现在将看到一个允许最大灵活性的选项。

我们可以创建自己的注释，使用 `@WithSecurityContext` 创建我们想要的任何 `SecurityContext`。例如，我们可以创建一个名为 `@WithMockCustomUser` 的注释，如下所示：

```

@Retention(RetentionPolicy.RUNTIME)
@WithSecurityContext(factory = WithMockCustomUserSecurityContextFactory.class)
public @interface WithMockCustomUser {

    String username() default "rob";

    String name() default "Rob Winch";
}

```

您可以看到 `@WithMockCustomUser` 注有 `@WithSecurityContext` 注释。这是Spring Security Test支持的信号，我们打算为测试创建一个 `SecurityContext`。`@WithSecurityContext` 注释要求我们指定一个 `SecurityContextFactory`，它将创建一个新的 `SecurityContext` 给出我们的 `@WithMockCustomUser` 注释。您可

以在下面找到我们的 `WithMockCustomUserSecurityContextFactory` 实现:

```
public class WithMockCustomUserSecurityContextFactory
implements WithSecurityContextFactory<WithMockCustomUser> {
    @Override
    public SecurityContext createSecurityContext(WithMockCustomUser customUser) {
        SecurityContext context = SecurityContextHolder.createEmptyContext();

        CustomUserDetails principal =
            new CustomUserDetails(customUser.name(), customUser.username());
        Authentication auth =
            new UsernamePasswordAuthenticationToken(principal, "password", principal.getAuthorities());
        context.setAuthentication(auth);
        return context;
    }
}
```

我们现在可以使用我们的新注释标注测试类或测试方法, Spring Security的 `WithSecurityContextTestExecutionListener` 将确保我们的 `SecurityContext` 适当地填充。

在创建自己的 `WithSecurityContextFactory` 实现时, 很高兴知道它们可以使用标准Spring注释进行注释。例如, `WithUserDetailsSecurityContextFactory` 使用 `@Autowired` 注释来获取 `UserDetailsService` :

```
final class WithUserDetailsSecurityContextFactory
implements WithSecurityContextFactory<WithUserDetails> {

    private UserDetailsService userDetailsService;

    @Autowired
    public WithUserDetailsSecurityContextFactory(UserDetailsService userDetailsService) {
        this.userDetailsService = userDetailsService;
    }

    public SecurityContext createSecurityContext(WithUserDetails withUser) {
        String username = withUser.value();
        Assert.hasLength(username, "value() must be non-empty String");
        UserDetails principal = userDetailsService.loadUserByUsername(username);
        Authentication authentication = new UsernamePasswordAuthenticationToken(principal, principal.getPassword(), principal.getAuthorities());
        SecurityContext context = SecurityContextHolder.createEmptyContext();
        context.setAuthentication(authentication);
        return context;
    }
}
```

默认情况下, `SecurityContext` 在过程中设置 `TestExecutionListener.beforeTestMethod` 事件。这与JUnit `@Before` 之前的情况 `@Before` 。您可以期间改变这种情况发生 `TestExecutionListener.beforeTestExecution` 事件是后JUnit[™] 的 `@Before` , 但在调用测试方法之前。

```
@WithSecurityContext(setupBefore = TestExecutionEvent.TEST_EXECUTION)
```

11.6 Test Meta Annotations 译: 11.6测试元标注

如果您经常在测试中重复使用同一用户, 那么不得不重复指定属性。例如, 如果有许多与使用用户名"admin"和角色 `ROLE_USER` 和 `ROLE_ADMIN` 的管理用户相关的测试, 则必须编写以下内容:

```
@WithMockUser(username="admin",roles={"USER","ADMIN"})
```

我们可以使用元注释, 而不是随处重复。例如, 我们可以创建一个名为 `WithMockAdmin` 的元注释:

```
@Retention(RetentionPolicy.RUNTIME)
@WithMockUser(value="rob",roles="ADMIN")
public @interface WithMockAdmin { }
```

现在我们可以使用 `@WithMockAdmin` , 就像更详细的 `@WithMockUser` 。

元注释可与上述任何测试注释一起使用。例如, 这意味着我们也可以为 `@WithUserDetails("admin")` 创建元注释。

12. Spring MVC Test Integration 译: 12.Spring MVC测试集成

Spring Security提供与 [Spring MVC Test](#) 的全面集成

12.1 Setting Up MockMvc and Spring Security 译: 12.1设置MockMvc和Spring Security

为了在Spring MVC测试中使用Spring Security, 必须将Spring Security `FilterChainProxy` 添加为 `Filter` 。还需要添加Spring Security 的 `TestSecurityContextHolderPostProcessor` 以支持[Running as a User in Spring MVC Test with Annotations](#) 。这可以使用Spring Security 的 `SecurityMockMvcConfigurers.springSecurity()` 。例如:



Spring Security的测试支持需要spring-test-4.1.3.RELEASE或更高版本。

```
import static org.springframework.security.test.web.servlet.setup.SecurityMockMvcConfigurers.*;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration
@WebAppConfiguration
public class CsrfShowcaseTests {

    @Autowired
    private WebApplicationContext context;

    private MockMvc mvc;

    @Before
    public void setup() {
        mvc = MockMvcBuilders
            .webAppContextSetup(context)
            .apply(springSecurity()) ❶
            .build();
    }

    ...
}
```

❶ `SecurityMockMvcConfigurers.springSecurity()` 将执行我们需要将Spring Security与Spring MVC测试集成的所有初始设置

12.2 SecurityMockMvcRequestPostProcessors 译: 12.2 SecurityMockMvcRequestPostProcessors

Spring MVC Test提供了一个方便的接口，称为 `RequestPostProcessor`，可用于修改请求。Spring Security提供了多个 `RequestPostProcessor` 实现，使测试更加简单。为了使用Spring Security的 `RequestPostProcessor` 实现，请确保使用以下静态导入：

```
import static org.springframework.security.test.web.servlet.request.SecurityMockMvcRequestPostProcessors.*;
```

12.2.1 Testing with CSRF Protection 译: 12.2.1使用CSRF保护进行测试

当测试任何非安全的HTTP方法并使用Spring Security的CSRF保护时，您必须确保在请求中包含有效的CSRF令牌。使用以下命令将有效的CSRF令牌指定为请求参数：

```
mvc
    .perform(post("/").with(csrf()))
```

如果你喜欢，你可以在标题中包含CSRF令牌：

```
mvc
    .perform(post("/").with(csrf().asHeader()))
```

您也可以使用以下测试来提供无效的CSRF令牌：

```
mvc
    .perform(post("/").with(csrf().useInvalidToken()))
```

12.2.2 Running a Test as a User in Spring MVC Test 译: 12.2.2在Spring MVC测试中以用户身份进行测试

通常需要以特定用户身份进行测试。有两种填充用户的简单方法：

- [Running as a User in Spring MVC Test with RequestPostProcessor](#)
- [Running as a User in Spring MVC Test with Annotations](#)

12.2.3 Running as a User in Spring MVC Test with RequestPostProcessor 译: 12.2.3使用RequestPostProcessor在Spring MVC测试中以用户身份运行

有许多选项可以将用户关联到当前的 `HttpServletRequest`。例如，以下内容将以用户名“user”，密码“password”和角色“ROLE_USER”作为用户（不需要存在）运行：



该支持通过将用户与 `HttpServletRequest` 关联 `HttpServletRequest`。要将请求关联到 `SecurityContextHolder` 您需要确保 `SecurityContextPersistenceFilter` 与 `MockMvc` 实例关联。有几种方法可以做到这一点：

- Invoking `apply(springSecurity())`
- Adding Spring Security's `FilterChainProxy` to `MockMvc`
- Manually adding `SecurityContextPersistenceFilter` to the `MockMvc` instance may make sense when using `MockMvcBuilders.standaloneSetup`

```
mvc
    .perform(get("/").with(user("user")))
```

您可以轻松进行自定义。例如，以下内容将以用户名“admin”，密码“pass”以及角色“ROLE_USER”和“ROLE_ADMIN”作为用户（不需要存在）运行。

```
mvc
    .perform(get("/admin").with(user("admin").password("pass").roles("USER", "ADMIN")))
```

如果您有自己想要使用的自定义 `UserDetails`，则也可以轻松指定。例如，下面将使用指定 `UserDetails`（其不需要存在）添加到与运行 `UsernamePasswordAuthenticationToken` 具有指定的主 `UserDetails`：

```
mvc
    .perform(get("/").with(user(userDetails)))
```

您可以使用以下方式以匿名用户身份运行：

```
mvc
    .perform(get("/").with(anonymous()))
```

如果您使用默认用户运行并希望以匿名用户的身份执行一些请求，则此功能特别有用。

如果您想要自定义 `Authentication`（不需要存在），则可以使用以下方法：

```
mvc
    .perform(get("/").with(authentication(authentication)))
```

您甚至可以使用以下方式自定义 `SecurityContext`：

```
mvc
    .perform(get("/").with(securityContext(securityContext)))
```

我们还可以确保通过使用 `MockMvcBuilders` 的默认请求以每个请求的特定用户身份运行。例如，以下内容将以用户名“admin”，密码“password”和角色“ROLE_ADMIN”作为用户（不需要存在）运行：

```
mvc = MockMvcBuilders
    .webApplicationContextSetup(context)
    .defaultRequest(get("/").with(user("user").roles("ADMIN")))
    .apply(springSecurity())
    .build();
```

如果你发现你在许多测试中使用同一个用户，建议将用户移到一个方法。例如，您可以在名为 `CustomSecurityMockMvcRequestPostProcessors` 的类中指定以下 `CustomSecurityMockMvcRequestPostProcessors`：

```
public static RequestPostProcessor rob() {
    return user("rob").roles("ADMIN");
}
```

现在，您可以在 `SecurityMockMvcRequestPostProcessors` 上执行静态导入，并在您的测试中使用它：

```
import static sample.CustomSecurityMockMvcRequestPostProcessors.*;

...

mvc
    .perform(get("/").with(rob()))
```

Running as a User in Spring MVC Test with Annotations 译：在 Spring MVC 测试中使用注释运行为用户

作为使用 `RequestPostProcessor` 创建用户的替代方法，您可以使用 [Chapter 11, Testing Method Security](#) 中描述的 [注释](#)。例如，以下将以用户名“user”，密码“password”和角色“ROLE_USER”运行测试：

```
@Test
@WithMockUser
public void requestProtectedUrlWithUser() throws Exception {
    mvc
        .perform(get("/"))
        ...
}
```

或者，以下将以用户名“user”，密码“password”和角色“ROLE_ADMIN”运行测试：

```
@Test
@WithMockUser(roles="ADMIN")
public void requestProtectedUrlWithUser() throws Exception {
    mvc
        .perform(get("/"))
        ...
}
```

12.2.4 Testing HTTP Basic Authentication 译：12.2.4 测试 HTTP 基本认证

虽然始终可以使用 HTTP Basic 进行身份验证，但记住头名称，格式和编码值有点繁琐。现在可以使用 Spring Security 的 `httpBasic` `RequestPostProcessor`。例如，下面的代码片段：

```
mvc
    .perform(get("/").with(httpBasic("user", "password")))
```

将尝试使用 HTTP Basic 通过确保在 HTTP 请求上填充以下标头来验证具有用户名“user”和密码“password”的用户：

```
Authorization: Basic dXN1cjpwYXNzd29yZA==
```

12.3 SecurityMockMvcRequestBuilders 译：12.3 SecurityMockMvcRequestBuilders

Spring MVC Test 还提供了 `RequestBuilder` 接口，可用于创建在您的测试中使用的 `MockHttpServletRequest`。Spring Security 提供了几个可用于 `RequestBuilder` 测试的 `RequestBuilder` 实现。为了使用 Spring Security 的 `RequestBuilder` 实现，请确保使用以下静态导入：

```
import static org.springframework.security.test.web.servlet.request.SecurityMockMvcRequestBuilders.*;
```

12.3.1 Testing Form Based Authentication 译：12.3.1 测试基于表单的身份验证

使用 Spring Security 的测试支持，您可以轻松创建一个请求来测试基于表单的身份验证。例如，以下内容将使用用户名“user”，密码“password”和有效的 CSRF 标记提交 POST 到“/login”：

```
mvc
    .perform(formLogin())
```

定制请求很容易。例如，以下内容将使用用户名“admin”，密码“pass”和有效的CSRF标记提交POST到“/auth”：

```
mvc
    .perform(formLogin("/auth").user("admin").password("pass"))
```

我们也可以自定义包含用户名和密码的参数名称。例如，上述请求被修改为在HTTP参数“u”上包含用户名，在HTTP参数“p”上包含密码。

```
mvc
    .perform(formLogin("/auth").user("u", "admin").password("p", "pass"))
```

12.3.2 Testing Logout 译：12.3.2测试注销

虽然使用标准的Spring MVC测试相当简单，但您可以使用Spring Security的测试支持来简化测试注销。例如，以下将使用有效的CSRF令牌提交POST到“/logout”：

```
mvc
    .perform(logout())
```

您还可以自定义发布到的URL。例如，下面的代码片段将使用有效的CSRF令牌提交POST到“/signout”：

```
mvc
    .perform(logout("/signout"))
```

12.4 SecurityMockMvcResultMatchers 译：12.4 SecurityMockMvcResultMatchers

有时需要对请求进行各种与安全相关的断言。为了适应这种需求，Spring Security Test支持实现了Spring MVC Test的 `ResultMatcher` 接口。为了使用Spring Security的 `ResultMatcher` 实现，请确保使用以下静态导入：

```
import static org.springframework.security.test.web.servlet.response.SecurityMockMvcResultMatchers.*;
```

12.4.1 Unauthenticated Assertion 译：12.4.1未经认证的声明

有时，声明没有经过认证的用户与 `MockMvc` 调用的结果相关联可能是有价值的。例如，您可能想要测试提交无效的用户名和密码，并验证没有用户通过身份验证。使用Spring Security的测试支持，您可以使用类似以下内容轻松完成此操作：

```
mvc
    .perform(formLogin().password("invalid"))
    .andExpect(unauthenticated());
```

12.4.2 Authenticated Assertion 译：12.4.2认证断言

通常我们必须断言经过身份验证的用户存在。例如，我们可能想验证我们是否成功验证。我们可以通过以下代码片段验证基于表单的登录是否成功：

```
mvc
    .perform(formLogin())
    .andExpect(authenticated());
```

如果我们想要声明用户的角色，我们可以改进我们以前的代码，如下所示：

```
mvc
    .perform(formLogin().user("admin"))
    .andExpect(authenticated().withRoles("USER", "ADMIN"));
```

或者，我们可以验证用户名：

```
mvc
    .perform(formLogin().user("admin"))
    .andExpect(authenticated().withUsername("admin"));
```

我们也可以结合这些断言：

```
mvc
    .perform(formLogin().user("admin").roles("USER", "ADMIN"))
    .andExpect(authenticated().withUsername("admin"));
```

我们也可以对认证进行任意的断言

```
mvc
    .perform(formLogin())
    .andExpect(authenticated().withAuthentication(auth ->
        assertThat(auth).assertInstanceOf(UsernamePasswordAuthenticationToken.class)));
```

13. WebFlux Support 译：13. WebFlux支持

13.1 Reactive Method Security 译：13.1 reactive方法安全性

例如，我们可以使用与Chapter 11, *Testing Method Security*中相同的设置和注释来测试我们的示例Section 5.10.3, “EnableReactiveMethodSecurity”。以下是我们可以做的一个最小样本：

```

@RunWith(SpringRunner.class)
@ContextConfiguration(classes = HelloWebfluxMethodApplication.class)
public class HelloWorldMessageServiceTests {

    @Autowired
    HelloWorldMessageService messages;

    @Test
    public void messagesWhenNotAuthenticatedThenDenied() {
        StepVerifier.create(this.messages.findMessage())
            .expectError(AccessDeniedException.class)
            .verify();
    }

    @Test
    @WithMockUser
    public void messagesWhenUserThenDenied() {
        StepVerifier.create(this.messages.findMessage())
            .expectError(AccessDeniedException.class)
            .verify();
    }

    @Test
    @WithMockUser(roles = "ADMIN")
    public void messagesWhenAdminThenOk() {
        StepVerifier.create(this.messages.findMessage())
            .expectNext("Hello World!")
            .verifyComplete();
    }
}

```

13.2 WebTestClientSupport 译: 13.2 WebTestClientSupport

Spring Security提供与 `WebTestClient` 集成。基本设置如下所示:

```

@RunWith(SpringRunner.class)
@ContextConfiguration(classes = HelloWebfluxMethodApplication.class)
public class HelloWebfluxMethodApplicationTests {

    @Autowired
    ApplicationContext context;

    WebTestClient rest;

    @Before
    public void setup() {
        this.rest = WebTestClient
            .bindToApplicationContext(this.context)
            // add Spring Security test Support
            .apply(springSecurity())
            .configureClient()
            .filter(basicAuthentication())
            .build();
    }
    // ...
}

```

13.2.1 Authentication 译: 13.2.1 认证

在将Spring Security支持应用到 `WebTestClient` 我们可以使用注释或 `mutateWith` 支持。例如:

```

@Test
public void messageWhenNotAuthenticated() throws Exception {
    this.rest
        .get()
        .uri("/message")
        .exchange()
        .expectStatus().isUnauthorized();
}

// --- WithMockUser ---

@Test
@WithMockUser
public void messageWhenWithMockUserThenForbidden() throws Exception {
    this.rest
        .get()
        .uri("/message")
        .exchange()
        .expectStatus().isEqualTo(HttpStatus.FORBIDDEN);
}

@Test
@WithMockUser(roles = "ADMIN")
public void messageWhenWithMockAdminThenOk() throws Exception {
    this.rest
        .get()
        .uri("/message")
        .exchange()
        .expectStatus().isOk()
        .expectBody(String.class).isEqualTo("Hello World!");
}

// --- mutateWith mockUser ---

@Test
public void messageWhenMutateWithMockUserThenForbidden() throws Exception {
    this.rest
        .mutateWith(mockUser())
        .get()
        .uri("/message")
        .exchange()
        .expectStatus().isEqualTo(HttpStatus.FORBIDDEN);
}

@Test
public void messageWhenMutateWithMockAdminThenOk() throws Exception {
    this.rest
        .mutateWith(mockUser().roles("ADMIN"))
        .get()
        .uri("/message")
        .exchange()
        .expectStatus().isOk()
        .expectBody(String.class).isEqualTo("Hello World!");
}

```

13.2.2 CSRF Support 译: 13.2.2 CSRF支持

Spring Security还通过 `WebTestClient` 提供对CSRF测试的支持。例如:

```

this.rest
    // provide a valid CSRF token
    .mutateWith(csrf())
    .post()
    .uri("/login")
    ...

```

Part IV. Web Application Security 译: 第四部分 - Web应用程序安全

大多数Spring Security用户将在使用HTTP和Servlet API的应用程序中使用框架。在这一部分中,我们将看看Spring Security如何为应用程序的Web层提供身份验证和访问控制功能。我们将在名称空间的外观后面查看实际组装哪些类和接口以提供Web层安全性。在某些情况下,必须使用传统的bean配置来完全控制配置,所以我们会看到如何直接配置这些类而不使用命名空间。

14. The Security Filter Chain 译: 14安全过滤器链

Spring Security的Web基础架构完全基于标准的Servlet过滤器。它没有在任何内部使用servlet或任何其他基于servlet的框架(如Spring MVC),因此它与任何特定的Web技术都没有强大的联系。它在 `HttpServletRequest` 和 `HttpServletResponse` 处理,并且 `HttpServletResponse` 这些请求是来自浏览器,Web服务客户端, `HttpInvoker` 还是AJAX应用程序。

Spring Security在内部维护一个过滤器链,其中每个过滤器都有特定的责任,并且根据需要哪些服务来添加或从配置中删除过滤器。过滤器的排序很重要,因为它们之间存在依赖关系。如果您一直在使用 `namespace configuration`,那么过滤器会自动为您配置,而且您不必明确定义任何Spring Bean,但可能有时需要完全控制安全过滤器链,无论是因为您使用的是功能它们在名称空间中不受支持,或者您正在使用您自己的定制版本的类。

14.1 DelegatingFilterProxy 译: 14.1 DelegatingFilterProxy

当使用servlet过滤器时，显然需要在 `web.xml` 声明它们，否则它们将被servlet容器忽略。在Spring Security中，过滤器类也是在应用程序上下文中定义的Spring bean，因此可以利用Spring丰富的依赖注入工具和生命周期接口。Spring的 `DelegatingFilterProxy` 提供了 `web.xml` 与应用程序上下文之间的链接。

在使用 `DelegatingFilterProxy`，您会在 `web.xml` 文件中看到类似这样的 `web.xml`：

```
<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>
```

请注意，过滤器实际上是 `DelegatingFilterProxy`，而不是实际实现过滤器逻辑的类。`DelegatingFilterProxy`所做的是将 `Filter` 的方法委托给从Spring应用程序上下文获得的bean。这使得bean可以从Spring Web应用程序上下文生命周期支持和配置灵活性中受益。该bean必须实现 `javax.servlet.Filter`，它必须与 `filter-name` 元素具有相同的名称。阅读Javadoc的 `DelegatingFilterProxy` 了解更多信息。

14.2 FilterChainProxy

Spring Security的Web基础架构只能用于委托给 `FilterChainProxy` 的实例。安全过滤器不应该单独使用。从理论上讲，你可以在应用程序上下文文件中声明你需要的每个Spring Security过滤器bean，并为每个过滤器添加一个对应的 `DelegatingFilterProxy` 条目到 `web.xml`，确保它们的顺序正确，但是这样做会很麻烦，会使 `web.xml` 文件混乱如果你有很多过滤器，很快。`FilterChainProxy` 让我们添加一个条目到 `web.xml` 并完全处理用于管理我们网络安全bean的应用程序上下文文件。它使用 `DelegatingFilterProxy`，就像上面的示例一样，但 `filter-name` 设置为bean名称“filterChainProxy”。过滤器链然后在应用程序上下文中用相同的bean名称声明。这是一个例子：

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

命名空间元素 `filter-chain` 用于方便地设置应用程序中所需的安全筛选器链。^[6] 它将特定的URL模式映射到根据 `filters` 元素中指定的bean名称构建的过滤器列表，并将它们组合到 `SecurityFilterChain` 类型的bean中。`pattern` 属性采用Ant路径，最具体的URI应该首先出现^[7]。在运行时，`FilterChainProxy` 将找到匹配当前Web请求的第一个URI模式，并且 `filters` 属性指定的筛选器列表将应用于该请求。过滤器将按照它们定义的顺序调用，因此您可以完全控制应用于特定URL的过滤器链。

您可能已经注意到，我们已经在过滤器链中声明了两个 `SecurityContextPersistenceFilter`（`ASC` 是 `allowSessionCreation`，属于 `SecurityContextPersistenceFilter`）。由于Web服务在将来的请求中将不会呈现 `jsessionid`，`HttpSession` 为这些用户代理创建 `HttpSession` 将是浪费。如果您有一个需要最大可伸缩性的高容量应用程序，我们建议您使用上述方法。对于较小的应用，使用单个 `SecurityContextPersistenceFilter`（其默认 `allowSessionCreation` 为 `true`）可能就足够了。

请注意，`FilterChainProxy` 不会在其配置的过滤器上调用标准过滤器生命周期方法。我们建议您使用Spring的应用程序上下文生命周期接口作为替代，就像您对其他任何Spring bean一样。

当我们了解了如何设置使用Web安全namespace configuration，我们使用的是 `DelegatingFilterProxy` 的名称是“springSecurityFilterChain”。您现在应该能够看到这是由名称空间创建的 `FilterChainProxy` 的名称。

14.2.1 Bypassing the Filter Chain

您可以使用属性 `filters = "none"` 作为提供过滤器bean列表的替代方法。这将完全忽略来自安全过滤器链的请求模式。请注意，与此路径相匹配的任何内容都不会应用认证或授权服务，并且可以自由访问。如果您想在请求期间使用 `SecurityContext` 内容的内容，那么它必须通过安全筛选器链。否则 `SecurityContextHolder` 将不会被填充，并且内容将为空。

14.3 Filter Ordering

过滤器在链中定义的顺序非常重要。无论您实际使用哪些过滤器，顺序应如下所示：

- `ChannelProcessingFilter`，because it might need to redirect to a different protocol
- `SecurityContextPersistenceFilter`，so a `SecurityContext` can be set up in the `SecurityContextHolder` at the beginning of a web request, and any changes to the `SecurityContext` can be copied to the `HttpSession` when the web request ends (ready for use with the next web request)
- `ConcurrentSessionFilter`，because it uses the `SecurityContextHolder` functionality and needs to update the `SessionRegistry` to reflect ongoing requests from the principal
- Authentication processing mechanisms - `UsernamePasswordAuthenticationFilter`，`CasAuthenticationFilter`，`BasicAuthenticationFilter` etc - so that the `SecurityContextHolder` can be modified to contain a valid `Authentication` request token
- The `SecurityContextHolderAwareRequestFilter`，if you are using it to install a Spring Security aware `HttpServletRequestWrapper` into your servlet container
- The `JaasApiIntegrationFilter`，if a `JaasAuthenticationToken` is in the `SecurityContextHolder` this will process the `FilterChain` as the `Subject` in the `JaasAuthenticationToken`
- `RememberMeAuthenticationFilter`，so that if no earlier authentication processing mechanism updated the `SecurityContextHolder`，and the request presents a cookie that enables remember-me services to take place, a suitable remembered `Authentication` object will be put there
- `AnonymousAuthenticationFilter`，so that if no earlier authentication processing mechanism updated the `SecurityContextHolder`，an anonymous `Authentication` object will be put there
- `ExceptionTranslationFilter`，to catch any Spring Security exceptions so that either an HTTP error response can be returned or an appropriate

`AuthenticationEntryPoint` can be launched

- `FilterSecurityInterceptor`, to protect web URLs and raise exceptions when access is denied

14.4 Request Matching and HttpFirewall

译: 14.4请求匹配和HttpFirewall

Spring Security有几个区域, 您定义的模式会根据传入的请求进行测试, 以决定如何处理请求。当`FilterChainProxy`决定哪个过滤器链应该通过请求时, 以及`FilterSecurityInterceptor`决定应用于请求的安全约束时, 会发生这种情况。理解机制是什么以及在针对您定义的模式进行测试时使用的URL值很重要。

`Servlet`规范定义了`HttpServletRequest`几个属性, 这些属性可以通过getter方法访问, 我们可能想要匹配。这些是`contextPath`, `servletPath`, `pathInfo`和`queryString`。Spring Security只关心应用程序中的路径, 因此`contextPath`被忽略。不幸的是, `Servlet`规范没有精确定

义`servletPath`和`pathInfo`的值将包含哪些特定的请求URL。例如, URL的每个路径段可能包含参数, 如RFC 2396^[8]中所定义。规范没有明确说明这些值是否应该包含在`servletPath`和`pathInfo`值中, 并且行为在不同的`Servlet`容器之间有所不同。当应用程序部署在未从这些值中剥离路径参数的容器中时, 攻击者可能会将它们添加到请求的URL中, 以使模式匹配成功或意外失败。^[9]。传入URL中的其他变体也是可能的。例如, 它可能包含路径遍历序列(如`../`)或多个正斜杠(`///`), 这些也可能导致模式匹配失败。一些容器在执行`Servlet`映射之前将这些归一化, 但其他容器不会。为防止出现类似问题, `FilterChainProxy`使用`HttpFirewall`策略检查并包装请求。未规范化的请求默认会自动被拒绝, 路径参数和重复的斜杠会被删除以达到匹配的目的。^[10]。因此, 必须使用`FilterChainProxy`来管理安全过滤器链。请注意, `servletPath`和`pathInfo`值由容器解码, 因此您的应用程序不应该包含任何包含分号的有效路径, 因为这些部分将被删除以达到匹配目的。

如上所述, 默认策略是使用Ant风格路径进行匹配, 这对于大多数用户来说可能是最佳选择。该策略在类`AntPathRequestMatcher`实现, 该类使用Spring的`AntPathMatcher`执行模式与连接的`servletPath`和`pathInfo`不区分大小写的匹配, 忽略`queryString`。

如果由于某种原因, 您需要更强大的匹配策略, 则可以使用正则表达式。战略实施是`RegexRequestMatcher`。有关更多信息, 请参阅此类的Javadoc。

实际上, 我们建议您在服务层使用方法安全性, 以控制对应用程序的访问, 而不完全依赖于在Web应用程序级别定义的安全约束。URL变化, 很难考虑到应用程序可能支持的所有可能的URL以及请求可能被操纵的方式。你应该试着限制自己使用一些简单易懂的简单蚂蚁路径。总是尝试使用“默认拒绝”方法, 即最后定义了全部通配符(`/`或`*`)并拒绝访问。

在服务层定义的安全性更健壮, 更难绕过, 所以你应该总是利用Spring Security的方法安全选项。

`HttpFirewall`还通过拒绝HTTP响应头中的新行字符来防止 HTTP Response Splitting。

默认使用`StrictHttpFirewall`。该实施拒绝看起来是恶意的请求。如果它对你的需求太严格, 那么您可以自定义哪些类型的请求被拒绝。但是, 您知道这会打开您的应用程序以应对攻击, 这一点很重要。例如, 如果您希望利用Spring MVC的Matrix变量, 则可以在XML中使用以下配置:

```
<b:bean id="httpFirewall"
      class="org.springframework.security.web.firewall.StrictHttpFirewall"
      p:allowSemicolon="true"/>

<http-firewall ref="httpFirewall"/>
```

Java配置通过暴露`StrictHttpFirewall` bean可以实现同样的`StrictHttpFirewall`。

```
@Bean
public StrictHttpFirewall httpFirewall() {
    StrictHttpFirewall firewall = new StrictHttpFirewall();
    firewall.setAllowSemicolon(true);
    return firewall;
}
```

14.5 Use with other Filter-Based Frameworks

译: 14.5与其他基于过滤器的框架结合使用

如果您正在使用其他也是基于过滤器的框架, 那么您需要确保Spring Security过滤器是第一位的。这使`SecurityContextHolder`能够及时填充以供其他过滤器使用。例子是使用SiteMesh来装饰你的网页或像Wicket这样的网络框架, 它使用过滤器来处理它的请求。

14.6 Advanced Namespace Configuration

译: 14.6高级命名空间配置

正如我们前面在命名空间章节中看到的那样, 可以使用多个`http`元素为不同的URL模式定义不同的安全配置。每个元素在内部创建一个过滤器链`FilterChainProxy`以及应该映射到它的URL模式。元素将按照它们声明的顺序添加, 因此必须首先声明最具体的模式。这是另一个例子, 对于类似于上面的情况, 应用程序同时支持无状态的RESTful API以及用户使用表单登录的普通Web应用程序。

```
<!-- Stateless RESTful service using Basic authentication -->
<http pattern="/restful/**" create-session="stateless">
  <intercept-url pattern='/**' access="hasRole('REMOTE')"/>
  <http-basic />
</http>

<!-- Empty filter chain for the login page -->
<http pattern="/login.htm*" security="none"/>
</http>

<!-- Additional filter chain for normal users, matching all other requests -->
<http>
  <intercept-url pattern='/**' access="hasRole('USER')"/>
  <form-login login-page="/login.htm" default-target-url="/home.htm"/>
  <logout />
</http>
```

^[6] 请注意, 您需要将安全性名称空间包含在应用程序上下文XML文件中才能使用此语法。使用`filter-chain-map`的旧语法仍然受支持, 但不赞成使用构造函数参数注入。

^[7] 而不是路径模式, 可以使用`request-matcher-ref`属性指定`RequestMatcher`实例以实现更强大的匹配

^[8] 当浏览器不支持cookie, 并且在`jsessionid`后附加`jsessionid`参数时, 您可能已经看到了这一点。但是, RFC允许在URL的任何路径段中存在这些参数

^[9] 请求离开`FilterChainProxy`, 原始值将被返回, 因此应用程序仍然可用。

^[10] 因此, 例如, 原始请求路径`/secure;hack=1/somefile.html;hack=2`将返回为`/secure/somefile.html`。

15. Core Security Filters

译: 15核心安全过滤器

在使用Spring Security的Web应用程序中总是会使用一些关键过滤器, 因此我们将首先查看这些及其支持类和接口。我们不会涵盖所有功能, 因此如果您想获得完整的图

像，请务必查看它们的Javadoc。

15.1 FilterSecurityInterceptor 译：15.1 FilterSecurityInterceptor

我们在讨论[access-control in general](#)时已经简要地看到了[FilterSecurityInterceptor](#)，并且我们已经将它与[<intercept-url>](#)元素组合在一起的命名空间在内部使用。现在我们将看到如何明确地配置它以及与[FilterChainProxy](#)一起使用，以及其件随过滤器[ExceptionTranslationFilter](#)。典型配置示例如下所示：

```
<bean id="filterSecurityInterceptor"
      class="org.springframework.security.web.access.intercept.FilterSecurityInterceptor">
  <property name="authenticationManager" ref="authenticationManager"/>
  <property name="accessDecisionManager" ref="accessDecisionManager"/>
  <property name="securityMetadataSource">
    <security:filter-security-metadata-source>
      <security:intercept-url pattern="/secure/super/**" access="ROLE_WE_DONT_HAVE"/>
      <security:intercept-url pattern="/secure/**" access="ROLE_SUPERVISOR,ROLE_TELLER"/>
    </security:filter-security-metadata-source>
  </property>
</bean>
```

[FilterSecurityInterceptor](#)负责处理HTTP资源的安全性。它要求提及[AuthenticationManager](#)和[AccessDecisionManager](#)。它还提供了适用于不同HTTP URL请求的配置属性。请参阅[技术介绍](#)中的[the original discussion on these](#)。

[FilterSecurityInterceptor](#)可以通过两种方式配置配置属性。第一个，如上所示，使用[<filter-security-metadata-source>](#)命名空间元素。这与名称空间章节中的[<http>](#)元素类似，但[<intercept-url>](#)子元素仅使用[pattern](#)和[access](#)属性。逗号用于分隔适用于每个HTTP URL的不同配置属性。第二种选择是编写自己的[SecurityMetadataSource](#)，但这超出了本文档的范围。不管使用[SecurityMetadataSource](#)方法，[SecurityMetadataSource](#)都负责返回包含与单个安全HTTP URL关联的所有配置属性的[List<ConfigAttribute>](#)。

应该指出的是，[FilterSecurityInterceptor.setSecurityMetadataSource\(\)](#)方法实际上期望[FilterInvocationSecurityMetadataSource](#)一个实例。这是一个标记界面，其子类[SecurityMetadataSource](#)。它只是表示[SecurityMetadataSource](#)理解[FilterInvocation](#)。为了简单起见，我们将继续将[FilterInvocationSecurityMetadataSource](#)称为[SecurityMetadataSource](#)，因为该区别与大多数用户的相关性很小。

该[SecurityMetadataSource](#)由命名空间语法创建的获得用于特定配置属性[FilterInvocation](#)通过匹配针对所配置的请求URL [pattern](#)属性。这与命名空间配置的行为方式相同。默认情况下，将所有表达式视为Apache Ant路径，正则表达式也支持更复杂的情况。[request-matcher](#)属性用于指定正在使用的模式的类型。在相同的定义中混合表达式语法是不可能的。作为一个例子，使用正则表达式而不是Ant路径的前面的配置将被写成如下：

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

模式总是按照它们定义的顺序进行评估。因此，重要的是更多的特定模式在列表中定义得比较不具体的模式更高。这在我们上面的例子中得到了反映，其中更具体的[/secure/super/](#)模式显示高于不太具体的[/secure/](#)模式。如果它们被颠倒过来，[/secure/](#)模式将始终匹配，并且[/secure/super/](#)模式将永远不会被评估。

15.2 ExceptionTranslationFilter 译：15.2 ExceptionTranslationFilter

所述[ExceptionTranslationFilter](#)位于上述[FilterSecurityInterceptor](#)在安全过滤器栈。它本身并没有执行任何实际的安全执行，但是处理安全拦截器抛出的异常并提供合适的HTTP响应。

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

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

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

15.2.1 AuthenticationEntryPoint 译：15.2.1 AuthenticationEntryPoint

如果用户请求安全的HTTP资源，但它们未通过身份验证，则将调用[AuthenticationEntryPoint](#)。一个合适的[AuthenticationException](#)或[AccessDeniedException](#)将由进一步向下调用堆栈的安全拦截器引发，触发入口点上的[commence](#)方法。这可以向用户提供适当的响应，以便开始认证。我们在这里使用的是[LoginUrlAuthenticationEntryPoint](#)，它将请求重定向到不同的URL（通常是登录页面）。所使用的实际实现将取决于您希望在应用程序中使用的身份验证机制。

15.2.2 AccessDeniedHandler 译：15.2.2 AccessDeniedHandler

如果用户已经通过身份验证并且他们尝试访问受保护的资源，会发生什么情况？在正常使用情况下，这种情况不应该发生，因为应用程序工作流程应限制在用户有权访问的操作上。例如，到管理页面的HTML链接可能对没有管理员角色的用户隐藏。您不能依赖隐藏链接来确保安全性，因为用户总是可能会直接输入URL以试图绕过限制。或者他们可能会修改一个RESTful URL来更改一些参数值。您的应用程序必须受到保护以免出现这些情况，否则它肯定会变得不安全。您通常会使用简单的Web层安全性将约束应用于基本URL，并在您的服务层接口上使用更具体的基于方法的安全性来真正确定允许的内容。

如果[AccessDeniedException](#)被抛出并且用户已经被认证，那么这意味着已经尝试了一个他们没有足够权限的操作。在这种情况下，

`ExceptionHandlerFilter` 将调用第二个策略 `AccessDeniedHandler` 。默认情况下，使用 `AccessDeniedHandlerImpl` ， `AccessDeniedHandlerImpl` 向客户端发送403（禁止）响应。或者，您可以显式配置实例（如上例所示）并设置一个错误页面URL，它将把请求转发到^[11]。这可以是一个简单的“拒绝访问”页面，如JSP，也可以是一个更复杂的处理程序，如MVC控制器。当然，你可以自己实现接口并使用你自己的实现。

当您使用命名空间来配置应用程序时，也可以提供自定义 `AccessDeniedHandler` 。详情请参阅the namespace appendix 。

15.2.3 SavedRequest s and the RequestCache Interface 译：15.2.3 SavedRequest和 RequestCache接口

`ExceptionHandlerFilter` 另一个职责是在调用 `AuthenticationEntryPoint` 之前保存当前请求。这允许在用户通过身份验证后恢复请求（请参阅上一个概述 [web authentication](#) ）。一个典型的例子就是用户使用表单登录，然后通过默认的 `SavedRequestAwareAuthenticationSuccessHandler` （请参阅 [below](#) ）重定向到原始URL。

`RequestCache` 封装了存储和检索 `HttpServletRequest` 实例所需的功能。默认情况下使用 `HttpSessionRequestCache` ，它将请求存储在 `HttpSession` 。当用户被重定向到原始URL时， `RequestCacheFilter` 实际上具有恢复从缓存中保存的请求的工作。

在正常情况下，您不需要修改任何此功能，但保存的请求处理是“尽力而为”的方式，并且可能存在默认配置无法处理的情况。这些接口的使用使它可以Spring Security 3.0开始完全插入。

15.3 SecurityContextPersistenceFilter 译：15.3 SecurityContextPersistenceFilter

我们在 [Technical Overview](#) 一章中介绍了这个重要过滤器的用途，因此您可能需要在此处重新阅读该部分。让我们先看看如何配置它以便与 `FilterChainProxy` 一起使用。基本配置只需要bean本身

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

正如我们以前所见，这个过滤器有两个主要任务。它负责在HTTP请求之间存储 `SecurityContext` 内容，并在请求完成时清除 `SecurityContextHolder` 。清除其中存储上下文的 `ThreadLocal` 是非常重要的，因为否则可能会将线程替换到Servlet容器的线程池中，同时还为特定用户连接安全上下文。此线程可能会在稍后阶段使用，并使用错误的凭据执行操作。

15.3.1 SecurityContextRepository 译：15.3.1 SecurityContextRepository

从Spring Security 3.0开始，加载和存储安全上下文的工作现在被委派给一个单独的策略接口：

```
public interface SecurityContextRepository {

    SecurityContext loadContext(HttpServletRequest request, HttpServletResponse response);

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

`HttpServletRequest` 只是传入请求和响应对象的容器，允许实现用包装类替换它们。返回的内容将被传递给过滤器链。

默认实现是 `HttpSessionSecurityContextRepository` ，它将安全上下文存储为 `HttpSession` 属性^[12]。此实现最重要的配置参数是 `allowSessionCreation` 属性，默认值为 `true` ，因此如果类需要为经过身份验证的用户存储安全上下文，则允许该类创建会话（它不会创建一个，除非身份验证已完成发生并且安全上下文的内容已经改变）。如果您不想创建会话，则可以将此属性设置为 `false` ：

```
<bean id="securityContextPersistenceFilter"
    class="org.springframework.security.web.context.SecurityContextPersistenceFilter">
    <property name="securityContextRepository">
        <bean class="org.springframework.security.web.context.HttpSessionSecurityContextRepository">
            <property name="allowSessionCreation" value="false" />
        </bean>
    </property>
</bean>
```

或者，您可以提供一个 `NullSecurityContextRepository` 实例，一个 `null object` 实现，这将阻止存储安全上下文，即使在请求期间已创建会话。

15.4 UsernamePasswordAuthenticationFilter 译：15.4 UsernamePasswordAuthenticationFilter

我们现在已经看到了Spring Security web配置中总是存在的三种主要过滤器。这些也是由名称空间 `<http>` 自动创建的三个元素，不能用替代方法替代。现在唯一缺少的是实际的身份验证机制，这将允许用户进行身份验证。此过滤器是最常用的认证过滤器，也是最常用的认证过滤器^[13]。它还提供了名称空间中 `<form-login>` 元素使用的实现。配置它需要三个阶段。

- Configure a `LoginUrlAuthenticationEntryPoint` with the URL of the login page, just as we did above, and set it on the `ExceptionHandlerFilter`.
- Implement the login page (using a JSP or MVC controller).
- Configure an instance of `UsernamePasswordAuthenticationFilter` in the application context
- Add the filter bean to your filter chain proxy (making sure you pay attention to the order).

登录表单只包含 `username` 和 `password` 输入字段，并发布到由过滤器监视的URL（默认情况下为 `/login` ）。基本的过滤器配置如下所示：

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

15.4.1 Application Flow on Authentication Success and Failure 译：15.4.1认证成功和失败的应用流程

过滤器调用配置的 `AuthenticationManager` 来处理每个认证请求。成功认证或认证失败后的目标分别由 `AuthenticationSuccessHandler` 和 `AuthenticationFailureHandler` 策略接口控制。该过滤器具有允许您设置这些属性的特性，因此您可以完全自定义行为^[14]。一些标准实现如供给例如 `SimpleUrlAuthenticationSuccessHandler` ， `SavedRequestAwareAuthenticationSuccessHandler` ， `SimpleUrlAuthenticationFailureHandler` ， `ExceptionHandlerMappingAuthenticationFailureHandler` 和 `DelegatingAuthenticationFailureHandler` 。查看这些类的Javadoc，并查看 `AbstractAuthenticationProcessingFilter` 以了解它们的工作原理和支持的功能。

如果认证成功，则生成的 `Authentication` 对象将被放置到 `SecurityContextHolder`。然后将调用配置的 `AuthenticationSuccessHandler` 以重定向或将用户转发到适当的目标。默认情况下，使用 `SavedRequestAwareAuthenticationSuccessHandler`，这意味着用户在被要求登录之前将被重定向到他们请求的原始目的地。



`ExceptionTranslationFilter` 缓存用户提出的原始请求。当用户进行身份验证时，请求处理程序使用此缓存请求来获取原始URL并将其重定向到它。原始请求然后被重建并用作替代。

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

- [11] 我们使用转发，以便 `SecurityContextHolder` 仍包含主体的详细信息，这对于向用户显示可能很有用。在 Spring Security 的旧版本中，我们依靠 `Servlet` 容器来处理缺少这种有用上下文信息的 403 错误消息。
- [12] 在 Spring Security 2.0 及更早版本中，这个过滤器被称为 `HttpSessionContextIntegrationFilter` 并且执行了存储上下文的所有工作都是由过滤器本身执行的。如果您熟悉这个课程，那么大多数可用的配置选项现在可以在 `HttpSessionSecurityContextRepository` 上 `HttpSessionSecurityContextRepository`。
- [13] 由于历史原因，在 Spring Security 3.0 之前，此过滤器被称为 `AuthenticationProcessingFilter`，入口点被称为 `AuthenticationProcessingFilterEntryPoint`。由于该框架现在支持许多不同形式的认证，因此它们在 3.0 中都被赋予了更具体的名称。
- [14] 在 3.0 之前的版本中，此时的应用程序流程已演变为一个阶段，由此类和策略插件上的各种属性控制。这个决定是为了重构代码以使这两个策略完全负责。

16. Servlet API integration 译：16. Servlet API集成

本节介绍 Spring Security 如何与 Servlet API 集成。 `servletapi.xml` 示例应用程序演示了每种方法的使用法。

16.1 Servlet 2.5+ Integration 译：16.1 Servlet 2.5+集成

16.1.1 HttpServletRequest.getRemoteUser() 译：16.1.1 HttpServletRequest.getRemoteUser()

`HttpServletRequest.getRemoteUser()` 将返回通常为当前用户名的 `SecurityContextHolder.getContext().getAuthentication().getName()` 的结果。如果您想在应用程序中显示当前用户名，这会很有用。此外，检查这是否为空可用于指示用户是否已通过身份验证或匿名。知道用户是否被认证对于确定是否应该显示某些 UI 元素是有用的（即，仅当用户被认证时才显示注销链接）。

16.1.2 HttpServletRequest.getUserPrincipal() 译：16.1.2 HttpServletRequest.getUserPrincipal()

该 `HttpServletRequest.getUserPrincipal()` 将返回的结果 `SecurityContextHolder.getContext().getAuthentication()`。这意味着它是一个 `Authentication` 其通常的实例 `UsernamePasswordAuthenticationToken` 使用用户名和密码的身份验证时。如果您需要有关用户的其他信息，这可能很有用。例如，您可能创建了自定义 `UserDetailsService`，该自定义将返回包含用户名和姓的自定义 `UserDetails`。您可以通过以下方式获取此信息：

```
Authentication auth = httpServletRequest.getUserPrincipal();
// assume integrated custom UserDetails called MyCustomUserDetails
// by default, typically instance of UserDetails
MyCustomUserDetails userDetails = (MyCustomUserDetails) auth.getPrincipal();
String firstName = userDetails.getFirstName();
String lastName = userDetails.getLastName();
```



应该指出，在整个应用程序中执行如此多的逻辑通常是不好的做法。相反，应该集中它来减少 Spring Security 和 Servlet API 的耦合。

16.1.3 HttpServletRequest.isUserInRole(String) 译：16.1.3 HttpServletRequest.isUserInRole(字符串)

该 `HttpServletRequest.isUserInRole(String)` 将确定是否 `SecurityContextHolder.getContext().getAuthentication().getAuthorities()` 包含 `GrantedAuthority` 与传入作用 `isUserInRole(String)`。通常用户不应将“ROLE_”前缀传入此方法，因为它会自动添加。例如，如果要确定当前用户是否具有“ROLE_ADMIN”权限，则可以使用以下内容：

```
boolean isAdmin = httpServletRequest.isUserInRole("ADMIN");
```

这对确定是否显示某些 UI 组件可能很有用。例如，只有当前用户是管理员时，才可以显示管理员链接。

16.2 Servlet 3+ Integration 译：16.2 Servlet 3+集成

以下部分描述 Spring Security 集成的 Servlet 3 方法。

16.2.1 HttpServletRequest.authenticate(HttpServletRequest, HttpServletResponse) 译：16.2.1 HttpServletRequest.authenticate (HttpServletRequest, HttpServletResponse)

`HttpServletRequest.authenticate(HttpServletRequest, HttpServletResponse)` 方法可用于确保用户通过身份验证。如果它们未通过身份验证，则配置的 `AuthenticationEntryPoint` 将用于请求用户进行身份验证（即重定向到登录页面）。

16.2.2 HttpServletRequest.login(String, String) 译：16.2.2 HttpServletRequest.login (String, String)

`HttpServletRequest.login(String, String)` 方法可用于使用当前的 `AuthenticationManager` 来验证用户。例如，以下内容将尝试使用用户名“user”和密码“password”进行验证：

```
try {
    httpServletRequest.login("user", "password");
} catch (ServletException e) {
    // fail to authenticate
}
```



如果您希望 Spring Security 处理失败的身份验证尝试，则不需要捕获 `ServletException`。

16.2.3 HttpServletRequest.logout() 译: 16.2.3 HttpServletRequest.logout()

[HttpServletRequest.logout\(\)](#)方法可用于将当前用户登出。

通常这意味着SecurityContextHolder将被清除，HttpSession将失效，任何“记住我”身份验证都将被清除，等等。但是，配置的LogoutHandler实现将根据您的Spring Security配置而变化。请注意，在调用HttpServletRequest.logout()之后，您仍然负责编写响应。通常会涉及重定向到欢迎页面。

16.2.4 AsyncContext.start(Runnable) 译: 16.2.4 AsyncContext.start() (可运行)

确保您的凭据将传播到新线程的[AsyncContext.start\(Runnable\)](#)方法。使用Spring Security的并发支持，Spring Security覆盖了AsyncContext.start(Runnable)，以确保在处理Runnable时使用当前的SecurityContext。例如，以下内容将输出当前用户的身份验证：

```
final AsyncContext async = httpRequest.startAsync();
async.start(new Runnable() {
    public void run() {
        Authentication authentication = SecurityContextHolder.getContext().getAuthentication();
        try {
            final HttpServletResponse asyncResponse = (HttpServletResponse) async.getResponse();
            asyncResponse.setStatus(HttpServletResponse.SC_OK);
            asyncResponse.getWriter().write(String.valueOf(authentication));
            async.complete();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
});
```

16.2.5 Async Servlet Support 译: 16.2.5 异步Servlet支持

如果您正在使用基于Java的配置，则可以开始使用了。如果您使用的是XML配置，则需要进行一些更新。第一步是确保您已更新web.xml以至少使用3.0架构，如下所示：

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
version="3.0">

</web-app>
```

接下来，您需要确保您的springSecurityFilterChain已设置为处理异步请求。

```
<filter>
<filter-name>springSecurityFilterChain</filter-name>
<filter-class>
    org.springframework.web.filter.DelegatingFilterProxy
</filter-class>
<async-supported>true</async-supported>
</filter>
<filter-mapping>
<filter-name>springSecurityFilterChain</filter-name>
<url-pattern>/*</url-pattern>
<dispatcher>REQUEST</dispatcher>
<dispatcher>ASYNC</dispatcher>
</filter-mapping>
```

就是这样！现在，Spring Security将确保您的SecurityContext也在异步请求上传播。

那么它是怎样工作的？如果你不是真的感兴趣，可以跳过本节的其余部分，否则请继续阅读。其中大部分都是内置到Servlet规范中的，但是有一点调整，Spring Security确实能够正确地处理异步请求。在Spring Security 3.2之前，只要提交HttpServletResponse，SecurityContextHolder中的SecurityContext就会自动保存。这可能会导致异步环境中的问题。例如，请考虑以下几点：

```
httpServletRequest.startAsync();
new Thread("AsyncThread") {
    @Override
    public void run() {
        try {
            // Do work
            TimeUnit.SECONDS.sleep(1);

            // Write to and commit the httpServletResponse
            httpServletResponse.getOutputStream().flush();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}.start();
```

问题是这个Thread对于Spring Security来说是不知道的，所以SecurityContext不会传播给它。这意味着当我们提交HttpServletResponse时，没有SecurityContext。当Spring Security在提交HttpServletResponse时自动保存SecurityContext时，它会丢失我们的登录用户。

从3.2版本开始，Spring Security足够聪明，只要HttpServletRequest.startAsync()被调用，就不会再自动保存SecurityContext来提交HttpServletResponse。

16.3 Servlet 3.1+ Integration 译: 16.3 Servlet 3.1+集成

以下部分描述Spring Security集成的Servlet 3.1方法。

16.3.1 HttpServletRequest#changeSessionId() 译: 16.3.1 HttpServletRequest# changeSessionId()

[HttpServletRequest.changeSessionId\(\)](#)是在Servlet 3.1和更高版本中防止 [Session Fixation](#)攻击的默认方法。

17. Basic and Digest Authentication 译：17基本和摘要式身份验证

基本和摘要式身份验证是在Web应用程序中流行的备用身份验证机制。基本身份验证通常用于在每个请求上传递其凭据的无状态客户端。将它与基于表单的身份验证结合使用时非常普遍，其中通过基于浏览器的用户界面和Web服务来使用应用程序。但是，基本身份验证将密码作为纯文本传输，因此它应该只能用于加密的传输层（如HTTPS）。

17.1 BasicAuthenticationFilter 译：17.1 BasicAuthenticationFilter

`BasicAuthenticationFilter` 负责处理HTTP头中提供的基本认证凭证。这可以用于验证Spring Remoting协议（例如Hessian和Burlap）以及普通浏览器用户代理（如Firefox和Internet Explorer）所做的调用。RFC 1945第11节和`BasicAuthenticationFilter`定义了遵循HTTP基本认证的标准。基本身份验证是一种非常有吸引力的身份验证方法，因为它在用户代理中非常广泛地部署，并且实现非常简单（它只是用户名：密码的Base64编码，在HTTP标头中指定）。

17.1.1 Configuration 译：17.1.配置

要实施HTTP基本认证，您需要将`BasicAuthenticationFilter`添加到您的过滤器链中。应用程序上下文应包含`BasicAuthenticationFilter`及其所需的协作者：

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

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

配置的`AuthenticationManager`处理每个认证请求。如果认证失败，配置的`AuthenticationEntryPoint`将用于重试认证过程。通常，您将结合`BasicAuthenticationEntryPoint`使用过滤器，该过滤器会返回带有适当头的401响应以重试HTTP基本身份验证。如果验证成功，则像往常一样将生成的`Authentication`对象放入`SecurityContextHolder`。

如果认证事件成功，或者由于HTTP标头不包含支持的认证请求而未尝试认证，则过滤器链将照常继续。过滤器链将被中断的唯一时间是身份验证失败并调用`AuthenticationEntryPoint`。

17.2 DigestAuthenticationFilter 译：17.2 DigestAuthenticationFilter

`DigestAuthenticationFilter`能够处理HTTP头中提供的摘要式身份验证凭证。摘要式身份验证尝试解决许多基本身份验证的弱点，特别是通过确保证书永远不会以明文形式在整个线路上发送。许多用户代理支持摘要式身份验证，包括Mozilla Firefox和Internet Explorer。控制HTTP摘要认证的标准由RFC 2617定义，它更新RFC 2069规定的早期版本的摘要认证标准。大多数用户代理实现RFC 2617.Spring Security的`DigestAuthenticationFilter`与“auth”质量兼容RFC 2617规定的保护（[qop]）也提供了与RFC 2069的向后兼容性。如果您需要使用未加密的HTTP（即无TLS/HTTPS）并希望最大限度地提高身份验证过程的安全性，则摘要式身份验证是一种更具吸引力的选项。事实上，如RFC 2518第17.1节所述，摘要式身份验证是WebDAV协议的强制性要求。



您不应该在现代应用程序中使用摘要，因为它不被认为是安全的。最明显的问题是，您必须以明文，加密或MD5格式存储您的密码。所有这些存储格式都被认为是**不安全的**。相反，你应该使用单向自适应密码哈希（即bcrypt, PBKDF2, SCrypt等）。

摘要式身份验证的核心是一个“随机数”。这是服务器生成的值。Spring Security的随机数采用以下格式：

```
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
```

所述`DigestAuthenticationEntryPoint`有一个属性指定`key`用于产生随机数的令牌，具有沿着`nonceValiditySeconds`属性用于确定所述过期时间（默认300，它等于5分钟）。Whist的nonce有效，摘要是通过连接各种字符串来计算的，包括用户名，密码，nonce，被请求的URI，客户端生成的nonce（仅仅是用户代理生成每个请求的随机值），域名等，然后执行MD5哈希。服务器和用户代理都执行此摘要计算，如果他们不同意所包含的值（例如密码），则会生成不同的哈希码。在Spring Security实现中，如果服务器生成的随机数仅仅过期（但摘要是有用的），则`DigestAuthenticationEntryPoint`将发送`"stale=true"`标题。这告诉用户代理不需要打扰用户（因为密码和用户名等是正确的），而只是使用新的随机数再次尝试。

为适当的值`nonceValiditySeconds`的参数`DigestAuthenticationEntryPoint`取决于您的应用。非常安全的应用程序应该注意，可以使用截获的身份验证标头模拟主体，直到达到nonce中包含的`expirationTime`。这是选择适当设置时的关键原则，但对于非常安全的应用程序而言，首先不会在TLS/HTTPS上运行，这种情况并不常见。

由于Digest身份验证的实现比较复杂，因此通常会出现用户代理问题。例如，Internet Explorer无法在同一会话中的后续请求中显示“不透明”令牌。Spring Security过滤器因此将所有状态信息封装到“nonce”令牌中。在我们的测试中，Spring Security的实现可以在Mozilla Firefox和Internet Explorer中可靠地工作，正确处理nonce超时等。

17.2.1 Configuration 译：17.2.配置

现在我们已经回顾了理论，让我们看看如何使用它。要实现HTTP摘要认证，有必要在过滤器链中定义`DigestAuthenticationFilter`。应用程序上下文将需要定义`DigestAuthenticationFilter`及其所需的协作者：

```
<bean id="digestFilter" class=
"org.springframework.security.web.authentication.www.DigestAuthenticationFilter">
<property name="userDetailsService" ref="jdbcDaoImpl"/>
<property name="authenticationEntryPoint" ref="digestEntryPoint"/>
<property name="userCache" ref="userCache"/>
</bean>

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

需要配置`UserDetailsService`，因为`DigestAuthenticationFilter`必须能够直接访问用户的明文密码。如果您在DAO^[15]中使用编码密码，摘要式身份验证将不

起作用。DAO协作者与`UserCache`一起直接与`DaoAuthenticationProvider`直接共享。`authenticationEntryPoint`属性必须为`DigestAuthenticationEntryPoint`，以便`DigestAuthenticationFilter`可以获取正确的`realmName`和`key`以进行摘要计算。

与`BasicAuthenticationFilter`一样，如果认证成功，`Authentication`请求令牌将被放入`SecurityContextHolder`。如果身份验证事件成功，或者由于HTTP头未包含摘要身份验证请求而未尝试身份验证，则过滤器链将照常继续。过滤器链将被中断的唯一时间是验证失败并调用`AuthenticationEntryPoint`，如`AuthenticationEntryPoint`所述。

摘要式认证的RFC提供了一系列附加功能以进一步提高安全性。例如，可以在每个请求中更改随机数。尽管如此，Spring Security实现的目的是尽量减少实现的复杂性（以及可能出现的无疑的用户代理不兼容），并避免存储服务器端状态。如果您希望更详细地了解这些功能，请您参阅RFC 2617。据我们所知，Spring Security的实现符合RFC的最低标准。

^[15] 是可能的编码格式HEX密码（MD5（用户名：境界：密码））中提供的`DigestAuthenticationFilter.passwordAlreadyEncoded`设定为`true`。但是，其他密码编码不适用于摘要式身份验证。

18. Remember-Me Authentication 译：记住我的身份验证

18.1 Overview 译：18.1概述

记住我或永久登录身份验证是指网站能够记住会话之间的主体身份。这通常通过向浏览器发送cookie来完成，在将来的会话中检测到cookie并导致自动登录。Spring Security为这些操作提供了必要的钩子，并且有两个具体的记住我的实现。一个使用散列来保存基于cookie的令牌的安全性，另一个使用数据库或其他持久存储机制来存储生成的令牌。

请注意，两种实现都需要`UserDetailsService`。如果您使用的身份验证提供程序未使用`UserDetailsService`（例如，LDAP提供程序），那么除非您的应用程序上下文中还有`UserDetailsService` bean，否则`UserDetailsService`。

18.2 Simple Hash-Based Token Approach 译：18.2简单的基于哈希的令牌方法

这种方法使用散列来实现有用的记忆我策略。本质上，cookie在成功交互式验证后发送到浏览器，cookie的组成如下：

```
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
```

因此，记住我记号仅在指定的时间段内有效，并且前提是用户名，密码和密钥不会更改。值得注意的是，这存在潜在的安全问题，因为捕获的记忆我令牌将可用于任何用户代理直到令牌过期。这与摘要式身份验证相同。如果委托人知道令牌已被捕获，他们可以轻松更改其密码，并立即使所有令牌失效。如果需要更重要的安全性，则应使用下一节中介绍的方法。或者，记住我服务应该根本不用。

如果您熟悉 [namespace configuration](#) 一章中讨论的主题，则只需添加 `<remember-me>` 元素即可启用记事本身份验证：

```
<http>
...
<remember-me key="myAppKey"/>
</http>
```

通常会自动选择`UserDetailsService`。如果您的应用程序环境中有多多个应用程序环境，则需要指定应将哪个应用于`user-service-ref`属性，其中值是您的`UserDetailsService` bean的名称。

18.3 Persistent Token Approach 译：18.3持久令牌方法

该方法基于http://jaspan.com/improved_persistent_login_cookie_best_practice文章，并进行了一些小修改^[16]。要在命名空间配置中使用这种方法，您需要提供一个数据来源参考：

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

数据库应包含使用以下SQL（或等效项）创建的`persistent_logins`表：

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

18.4 Remember-Me Interfaces and Implementations 译：18.4记住我的接口和实现

记住我与`UsernamePasswordAuthenticationFilter`一起`UsernamePasswordAuthenticationFilter`，并通过`AbstractAuthenticationProcessingFilter`超类中的钩子实现。它也用于`BasicAuthenticationFilter`。挂钩将在适当的时候调用具体的`RememberMeServices`。界面如下所示：

```
Authentication autoLogin(HttpServletRequest request, HttpServletResponse response);

void loginFail(HttpServletRequest request, HttpServletResponse response);

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

请参阅Javadoc以获取关于这些方法的更全面讨论，尽管在本阶段注意`AbstractAuthenticationProcessingFilter`仅调用`loginFail()`和`loginSuccess()`方法。每当`SecurityContextHolder`不包含`RememberMeAuthenticationFilter`时，`autoLogin()`方法由`RememberMeAuthenticationFilter` `Authentication`。因此，该接口为潜在的记忆我实现提供了充分的与身份验证相关的事件通知，并在候选Web请求可能包含Cookie并希望被记住时委托实现。这种设计允许任何数量的记忆

我实施策略。我们在上面看到Spring Security提供了两个实现。我们将依次查看这些内容。

18.4.1 TokenBasedRememberMeServices 译: 18.4.1 TokenBasedRememberMeServices

该实现支持Section 18.2, "Simple Hash-Based Token Approach"中描述的更简单的方法。`TokenBasedRememberMeServices`生成`RememberMeAuthenticationToken`，由`RememberMeAuthenticationProvider`处理。甲`key`被该认证提供器和之间共享`TokenBasedRememberMeServices`。另外，`TokenBasedRememberMeServices`需要一个`UserDetailsService`，通过它可以检索用户名和密码以进行签名比较，并生成`RememberMeAuthenticationToken`以包含正确的`GrantedAuthority`。某些注销命令应由应用程序提供，如果用户请求该命令，则会使Cookie无效。`TokenBasedRememberMeServices`还实现了Spring Security的`LogoutHandler`接口，因此可以使用`LogoutFilter`来自动清除cookie。

在应用程序上下文中启用Remember-me服务所需的bean如下所示：

```
<bean id="rememberMeFilter" class=
"org.springframework.security.web.authentication.rememberme.RememberMeAuthenticationFilter">
<property name="rememberMeServices" ref="rememberMeServices"/>
<property name="authenticationManager" ref="theAuthenticationManager" />
</bean>

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

<bean id="rememberMeAuthenticationProvider" class=
"org.springframework.security.authentication.RememberMeAuthenticationProvider">
<property name="key" value="springRocks"/>
</bean>
```

别忘了将您的`RememberMeServices`实施添加到您的`UsernamePasswordAuthenticationFilter.setRememberMeServices()`资产中，将`RememberMeAuthenticationProvider`包含在您的`AuthenticationManager.setProviders()`列表中，并将`RememberMeAuthenticationFilter`添加到您的`FilterChainProxy`（通常紧接在您的`UsernamePasswordAuthenticationFilter`）。

18.4.2 PersistentTokenBasedRememberMeServices 译: 18.4.2 PersistentTokenBasedRememberMeServices

该类可以以与`TokenBasedRememberMeServices`相同的方式`TokenBasedRememberMeServices`，但还需要使用`PersistentTokenRepository`配置来存储令牌。有两种标准实现。

- `InMemoryTokenRepositoryImpl` which is intended for testing only.
- `JdbcTokenRepositoryImpl` which stores the tokens in a database.

数据库模式在Section 18.3, "Persistent Token Approach"中有介绍。

[16] 本质上，用户名不包含在cookie中，以防止不必要地泄露有效的登录名。本文的评论部分对此进行了讨论。

19. Cross Site Request Forgery (CSRF) 译: 19.跨站点请求伪造 (CSRF)

本节讨论Spring Security的Cross Site Request Forgery (CSRF)支持。

19.1 CSRF Attacks 译: 19.1 CSRF攻击

在我们讨论Spring Security如何保护应用程序免受CSRF攻击之前，我们将解释什么是CSRF攻击。让我们来看一个具体的例子来更好地理解。

假设你银行的网站提供了一个表格，允许从当前登录的用户转账到另一个银行账户。例如，HTTP请求可能如下所示：

```
POST /transfer HTTP/1.1
Host: bank.example.com
Cookie: JSESSIONID=randomid; Domain=bank.example.com; Secure; HttpOnly
Content-Type: application/x-www-form-urlencoded

amount=100.00&routingNumber=1234&account=9876
```

现在假装你通过银行的网站进行身份验证，然后在没有注销的情况下访问一个恶意网站。邪恶的网站包含一个HTML页面，其格式如下：

```
<form action="https://bank.example.com/transfer" method="post">
<input type="hidden"
name="amount"
value="100.00"/>
<input type="hidden"
name="routingNumber"
value="evilsRoutingNumber"/>
<input type="hidden"
name="account"
value="evilsAccountNumber"/>
<input type="submit"
value="Win Money!"/>
</form>
```

你喜欢赢钱，所以你点击提交按钮。在这个过程中，你无意间将100美元转让给恶意用户。发生这种情况的原因是，虽然恶意网站无法看到您的Cookie，但与您的银行相关的Cookie仍会与请求一起发送。

最糟糕的是，整个过程可能已经使用JavaScript进行自动化。这意味着你甚至不需要点击按钮。那么我们如何保护自己免受这种攻击呢？

19.2 Synchronizer Token Pattern 译: 19.2同步令牌模式

问题在于来自银行网站的HTTP请求和来自恶意网站的请求完全相同。 这意味着无法拒绝来自恶意网站的请求，并允许来自银行网站的请求。 为了防止CSRF攻击，我们需要确保恶意网站无法提供请求中的内容。

一种解决方案是使用[Synchronizer Token Pattern](#)。此解决方案是为了确保除了我们的会话cookie之外，每个请求还需要一个随机生成的令牌作为HTTP参数。提交请求时，服务器必须查找参数的期望值，并将其与请求中的实际值进行比较。如果这些值不匹配，则请求将失败。

我们可以放宽期望，只需要更新状态的每个HTTP请求的令牌。这可以安全地完成，因为相同的来源策略确保恶意网站无法读取响应。此外，我们不想在HTTP GET中包含随机标记，因为这会导致令牌泄漏。

让我们来看看我们的例子将如何改变。假设随机生成的令牌存在于名为_csrf的HTTP参数中。例如，转账请求看起来像这样：

```
POST /transfer HTTP/1.1
Host: bank.example.com
Cookie: JSESSIONID=randomid; Domain=bank.example.com; Secure; HttpOnly
Content-Type: application/x-www-form-urlencoded

amount=100.00&routingNumber=1234&account=9876&_csrf=<secure-random>
```

你会注意到我们添加了一个随机值的_csrf参数。现在，恶意网站将无法猜测_csrf参数（必须在恶意网站上明确提供）的正确值，并且当服务器将实际令牌与预期令牌进行比较时，传输将失败。

19.3 When to use CSRF protection 译：19.3何时使用CSRF保护

什么时候应该使用CSRF保护？我们的建议是针对普通用户可以通过浏览器处理的任何请求使用CSRF保护。如果您只创建非浏览器客户端使用的服务，则可能需要禁用CSRF保护。

19.3.1 CSRF protection and JSON 译：19.3.1 CSRF保护和JSON

一个常见的问题是“我需要保护由JavaScript做出的JSON请求吗？”简短的答案是，这取决于。但是，您必须非常小心，因为存在会影响JSON请求的CSRF漏洞利用。例如，恶意用户可以创建一个[CSRF with JSON using the following form](#)：

```
<form action="https://bank.example.com/transfer" method="post" enctype="text/plain">
<input name="{"amount":100,"routingNumber":"evilsRoutingNumber","account":"evilsAccountNumber","ignore_me":'' value='test'}" type='hidden'>
<input type="submit"
  value="Win Money!"/>
</form>
```

这将产生以下JSON结构

```
{ "amount": 100,
  "routingNumber": "evilsRoutingNumber",
  "account": "evilsAccountNumber",
  "ignore_me": "=test"
}
```

如果应用程序未验证Content-Type，那么它将暴露于此漏洞利用。根据设置，验证Content-Type的Spring MVC应用程序仍然可以通过更新URL后缀来利用以“.json”结尾，如下所示：

```
<form action="https://bank.example.com/transfer.json" method="post" enctype="text/plain">
<input name="{"amount":100,"routingNumber":"evilsRoutingNumber","account":"evilsAccountNumber","ignore_me":'' value='test'}" type='hidden'>
<input type="submit"
  value="Win Money!"/>
</form>
```

19.3.2 CSRF and Stateless Browser Applications 译：19.3.2 CSRF和无状态浏览器应用程序

如果我的应用程序是无状态的呢？这并不一定意味着你受到保护。事实上，如果用户不需要在Web浏览器中针对特定请求执行任何操作，那么他们可能仍然容易受到CSRF攻击。

例如，考虑一个应用程序使用包含所有状态的自定义cookie来进行身份验证，而不是JSESSIONID。当CSRF攻击发生时，自定义cookie将与请求一起发送，其方式与前面示例中发送的JSESSIONID cookie相同。

使用基本身份验证的用户也很容易受到CSRF攻击，因为浏览器将自动在任何请求中包含用户名密码，方式与JSESSIONID cookie在前一示例中发送的方式相同。

19.4 Using Spring Security CSRF Protection 译：19.4使用Spring Security CSRF保护

那么，使用Spring Security来保护我们的网站免受CSRF攻击需要采取哪些措施？下面概述了使用Spring Security的CSRF保护的步骤：

- [Use proper HTTP verbs](#)
- [Configure CSRF Protection](#)
- [Include the CSRF Token](#)

19.4.1 Use proper HTTP verbs 译：19.4.1使用适当的HTTP动词

防止CSRF攻击的第一步是确保您的网站使用正确的HTTP动词。特别是，在Spring Security的CSRF支持可以使用之前，您需要确定您的应用程序正在使用PATCH，POST，PUT和/或DELETE来修改状态。

这不是Spring Security支持的限制，而是对正确CSRF预防的一般要求。原因是在HTTP GET中包含私人信息会导致信息泄露。请参阅[RFC 2616 Section 15.1.3 Encoding Sensitive Information in URI's](#)以获取有关使用POST而不是GET获取敏感信息的一般指导。

19.4.2 Configure CSRF Protection 译：19.4.2配置CSRF保护

下一步是在您的应用程序中包含Spring Security的CSRF保护。一些框架通过对用户的会话进行无效处理来处理无效的CSRF令牌，但这会导致[its own problems](#)。相反，默认情况下，Spring Security的CSRF保护会导致HTTP 403访问被拒绝。这可以通过配置[AccessDeniedHandler](#)以不同方式处理[InvalidCsrfTokenException](#)来进行定制。

从Spring Security 4.0开始，默认情况下使用XML配置启用CSRF保护。如果您想禁用CSRF保护，则可以在下面看到相应的XML配置。

```
<http>
<!-- ... -->
<csrf disabled="true"/>
</http>
```

Java Configuration默认启用CSRF保护。如果您想禁用CSRF，则可以在下面看到相应的Java配置。有关如何配置CSRF保护的其他自定义，请参阅[csrf\(\)](#)的Javadoc。

```
@EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .csrf().disable();
    }
}
```

19.4.3 Include the CSRF Token 译：19.4.3包含CSRF令牌

Form Submissions 译：表单提交

最后一步是确保在所有PATCH，POST，PUT和DELETE方法中包含CSRF标记。解决此问题的一种方法是使用[_csrf](#)请求属性来获取当前的[CsrfToken](#)。下面显示了使用JSP进行此操作的示例：

```
<c:url var="logoutUrl" value="/logout"/>
<form action="${logoutUrl}"
method="post">
<input type="submit"
value="Log out" />
<input type="hidden"
name="${_csrf.parameterName}"
value="${_csrf.token}"/>
</form>
```

更简单的方法是使用Spring Security JSP标记库中的 [the csrfInput tag](#)。



如果您使用的是Spring MVC [<form:form>](#) 标记或 [Thymeleaf 2.1+](#) 并且正在使用 [@EnableWebSecurity](#)，则 [CsrfToken](#) 会自动包含在您的使用中（使用 [CsrfRequestDataValueProcessor](#)）。

Ajax and JSON Requests 译：Ajax和JSON请求

如果您使用的是JSON，则无法在HTTP参数中提交CSRF令牌。相反，您可以在HTTP头中提交令牌。一个典型的模式是将CSRF令牌包含在元标记中。下面显示了一个JSP示例：

```
<html>
<head>
<meta name="_csrf" content="${_csrf.token}"/>
<!-- default header name is X-CSRF-TOKEN -->
<meta name="_csrf_header" content="${_csrf.headerName}"/>
<!-- ... -->
</head>
<!-- ... -->
```

您可以使用Spring Security JSP标记库中较简单的 [csrfMetaTags tag](#)来代替手动创建元标记。

然后，您可以将令牌包含在所有Ajax请求中。如果您使用jQuery，可以使用以下方法完成此操作：

```
$(function () {
var token = $("meta[name='_csrf']").attr("content");
var header = $("meta[name='_csrf_header']").attr("content");
$(document).ajaxSend(function(e, xhr, options) {
    xhr.setRequestHeader(header, token);
});
});
```

作为jQuery的替代品，我们推荐使用[cujoJS's rest.js](#)。[rest.js](#)模块为以RESTful方式处理HTTP请求和响应提供了高级支持。核心功能是通过将拦截器链接到客户端根据需要上下文化HTTP客户端添加行为的能力。

```
var client = rest.chain(csrf, {
token: $("meta[name='_csrf']").attr("content"),
name: $("meta[name='_csrf_header']").attr("content")
});
```

配置的客户端可以与需要向CSRF保护的资源发出请求的应用程序的任何组件共享。[rest.js](#)和[jQuery](#)之间的一个重要区别是，只有使用配置的客户端发出的请求才会包含CSRF令牌，而在[jQuery](#)中，所有请求都将包含该令牌。限定哪些请求接收令牌的能力有助于防止将CSRF令牌泄露给第三方。有关[rest.js](#)的更多信息，请参阅[rest.js reference documentation](#)。

CookieCsrfTokenRepository 译：CookieCsrfTokenRepository

可能会有用户想要将[CsrfToken](#)在cookie中的情况。默认情况下，[CookieCsrfTokenRepository](#)将写入名为[XSRF-TOKEN](#)的cookie，并从名为[X-XSRF-TOKEN](#)的头或[X-XSRF-TOKEN](#)或HTTP参数[_csrf](#)读取它。这些默认值来自[AngularJS](#)

您可以使用以下方式在XML中配置 [CookieCsrfTokenRepository](#)：

```
<http>
<!-- ... -->
<csrf token-repository-ref="tokenRepository"/>
</http>
<b:bean id="tokenRepository"
class="org.springframework.security.web.csrf.CookieCsrfTokenRepository"
p:cookieHttpOnly="false"/>
```



该示例明确设置了 `cookieHttpOnly=false`。这是允许JavaScript（即AngularJS）读取它的必要条件。如果您不需要直接使用JavaScript读取cookie的功能，建议省略 `cookieHttpOnly=false` 以提高安全性。

您可以使用以下 `CookieCsrfTokenRepository` 在Java配置中配置 `CookieCsrfTokenRepository`：

```
@EnableWebSecurity
public class WebSecurityConfig extends
    WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .csrf()
            .csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse());
    }
}
```



该示例明确设置了 `cookieHttpOnly=false`。这是允许JavaScript（即AngularJS）读取它的必要条件。如果您不需要直接使用JavaScript读取cookie的功能，则建议省略 `cookieHttpOnly=false`（改为使用 `new CookieCsrfTokenRepository()`）以提高安全性。

19.5 CSRF Caveats 译：19.5 CSRF警告

实施CSRF时有一些注意事项。

19.5.1 Timeouts 译：19.5 超时

一个问题是预期的CSRF令牌存储在HttpSession中，因此一旦HttpSession过期，您配置的 `AccessDeniedHandler` 将收到 `InvalidCsrfTokenException`。如果您使用默认的 `AccessDeniedHandler`，浏览器将获得HTTP 403并显示错误消息。



有人可能会问，为什么预期的 `CsrfToken` 默认情况下不存储在cookie中。这是因为有已知的攻击，其中标题（即指定cookie）可以由另一个域设置。这与Ruby on Rails [no longer skips CSRF checks when the header X-Requested-With is present](#)的原因相同。有关如何执行漏洞的详细信息，请参阅[this webappsec.org thread](#)。另一个缺点是，通过消除状态（即超时），如果令牌受到威胁，则无法强制终止令牌。

缓解处于超时状态的活动用户的一个简单方法是使用一些JavaScript让用户知道他们的会话即将过期。用户可以点击一个按钮继续并刷新会话。

或者，指定自定义 `AccessDeniedHandler` 可让您以任何喜欢的方式处理 `InvalidCsrfTokenException`。有关如何自定义 `AccessDeniedHandler` 的示例，请参阅[xml](#)和[Java configuration](#)提供的链接。

最后，可以将应用程序配置为使用不会过期的 `CookieCsrfTokenRepository`。如前所述，这不像使用会话那样安全，但在很多情况下可以足够好。

19.5.2 Logging In 译：19.5 登录

为了防止 [forging log in requests](#) 登录表单也应该受到保护以防止CSRF攻击。由于 `CsrfToken` 存储在HttpSession中，这意味着只要 `CsrfToken` 令牌属性被访问就会创建一个HttpSession。虽然这在RESTful/无状态架构中听起来很糟糕，但现实是状态对于实现实际安全性是必需的。没有国家，如果令牌受到损害，我们就无能为力。实际上，CSRF令牌的规模很小，对我们的架构应该有微不足道的影响。

保护登录表单的常用技术是在表单提交之前使用JavaScript函数获取有效的CSRF标记。通过这样做，不需要考虑会话超时（在前面的章节中讨论过），因为会话是在表单提交之前创建的（假设 `CookieCsrfTokenRepository` 不是被配置的），因此用户可以保持登录状态并在需要时提交用户名/密码。为了实现这一点，您可以利用Spring Security提供的 `CsrfTokenArgumentResolver`，并公开[here](#)中描述的类似端点。

19.5.3 Logging Out 译：19.5 注销

添加CSRF将更新LogoutFilter以仅使用HTTP POST。这可确保注销需要CSRF令牌，并且恶意用户无法强制注销用户。

一种方法是使用表单注销。如果你真的想要一个链接，你可以使用JavaScript来让链接执行一个POST（即可能在一个隐藏的窗体上）。对于禁用JavaScript的浏览器，您可以选择使链接将用户带到注销确认页面，该页面将执行POST。

如果你真的想在注销时使用HTTP GET，你可以这样做，但请记住这通常不被推荐。例如，以下Java配置将执行注销，并使用任何HTTP方法请求URL /注销：

```
@EnableWebSecurity
public class WebSecurityConfig extends
    WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .logout()
            .logoutRequestMatcher(new AntPathRequestMatcher("/logout"));
    }
}
```

19.5.4 Multipart (file upload) 译：19.5.4 部分（文件上传）

有多种方法可以对多部分/表单数据使用CSRF保护。每个选项都有其折衷。

- [Placing MultipartFilter before Spring Security](#)
- [Include CSRF token in action](#)



在将Spring Security的CSRF保护与多部分文件上传集成之前，请确保您可以先不使用CSRF保护。有关在Spring中使用多部分表单的更多信息可以在Spring参考文献的[17.10 Spring's multipart \(file upload\) support](#)部分和[MultipartFilter javadoc](#)中找到。

Placing MultipartFilter before Spring Security 译：在Spring Security之前放置MultipartFilter

第一个选项是确保在Spring Security筛选器之前指定了 `MultipartFilter`。在Spring Security过滤器之前指定 `MultipartFilter` 意味着没有调用 `MultipartFilter` 授权，这意味着任何人都可以在您的服务器上放置临时文件。但是，只有授权用户才能提交由您的应用程序处理的文件。一般来说，这是推荐的方法，因为临时文件上传应该对大多数服务器产生可忽略的影响。

为了确保在使用java配置的Spring Security过滤器之前指定了 `MultipartFilter`，用户可以覆盖 `beforeSpringSecurityFilterChain`，如下所示：

```
public class SecurityApplicationInitializer extends AbstractSecurityWebApplicationInitializer {

    @Override
    protected void beforeSpringSecurityFilterChain(ServletContext servletContext) {
        insertFilters(servletContext, new MultipartFilter());
    }
}
```

为了确保在具有XML配置的Spring Security过滤器之前指定了 `MultipartFilter`，用户可以确保 `MultipartFilter` 的 `<filter-mapping>` 元素位于 `web.xml` 中的 `springSecurityFilterChain` 之前，如下所示：

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

Include CSRF token in action 译：包含CSRF令牌

如果允许未经授权的用户上传临时文件是不可接受的，另一种方法是将 `MultipartFilter` 放置在Spring Security过滤器之后，并将CSRF作为查询参数包含在表单的action属性中。下面显示了一个jsp的例子

```
<form action="./upload?${_csrf.parameterName}=${_csrf.token}" method="post" enctype="multipart/form-data">
```

这种方法的缺点是查询参数可能泄漏。更为常见的是，将敏感数据放在主体或标题中以确保其不泄漏是最佳做法。其他信息可以在[RFC 2616 Section 15.1.3 Encoding Sensitive Information in URI's](#)中找到。

19.5.5 HiddenHttpMethodFilter 译：19.5.5 HiddenHttpMethodFilter

`HiddenHttpMethodFilter`应放置在Spring Security过滤器之前。总的来说，这是事实，但在防范CSRF攻击时可能会产生额外的影响。

请注意，`HiddenHttpMethodFilter`只覆盖POST上的HTTP方法，所以这实际上不会导致任何实际问题。但是，确保将它放在Spring Security的过滤器之前仍然是最佳实践。

19.6 Overriding Defaults 译：19.6覆盖默认值

Spring Security的目标是提供保护用户免受攻击的默认设置。这并不意味着你被迫接受所有的默认值。

例如，您可以提供一个自定义的 `CsrfTokenRepository` 来覆盖 `CsrfToken` 的存储方式。

您也可以指定一个自定义的 `RequestMatcher` 来确定哪些请求受CSRF保护（也许你不在乎是否注销）。简而言之，如果Spring Security的CSRF保护不像您想要的那样完美，您可以自定义行为。有关如何使用XML进行这些自定义设置的详细信息，请参阅[Section 43.1.18, "<csrf>"](#)文档，有关如何在使用Java配置时进行这些自定义设置的详细信息，请参阅 `CsrfConfigurer` javadoc。

20. CORS 译：20. CORS

Spring Framework提供了[first class support for CORS](#)。CORS必须在Spring Security之前处理，因为飞行前请求不会包含任何cookie（即 `JSESSIONID`）。如果请求中不包含任何cookie并且Spring Security是第一个，则该请求将确定用户未被认证（因为请求中没有cookie）并拒绝它。

确保首先处理CORS的最简单方法是使用 `CorsFilter`。用户可以通过使用以下方法提供 `CorsConfigurationSource` 来将 `CorsFilter` 与Spring Security集成在一起：


```

@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            // by default uses a Bean by the name of corsConfigurationSource
            .cors().and()
            ...
    }

    @Bean
    CorsConfigurationSource corsConfigurationSource() {
        CorsConfiguration configuration = new CorsConfiguration();
        configuration.setAllowedOrigins(Arrays.asList("https://example.com"));
        configuration.setAllowedMethods(Arrays.asList("GET", "POST"));
        UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
        source.registerCorsConfiguration("/**", configuration);
        return source;
    }
}

```

或以XML格式

```

<http>
  <cors configuration-source-ref="corsSource"/>
  ...
</http>
<b:bean id="corsSource" class="org.springframework.web.cors.UrlBasedCorsConfigurationSource">
  ...
</b:bean>

```

如果您使用Spring MVC的CORS支持，则可以省略指定 `CorsConfigurationSource`，Spring Security将利用为Spring MVC提供的CORS配置。

```

@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            // if Spring MVC is on classpath and no CorsConfigurationSource is provided,
            // Spring Security will use CORS configuration provided to Spring MVC
            .cors().and()
            ...
    }
}

```

或以XML格式

```

<http>
  <!-- Default to Spring MVC's CORS configuration -->
  <cors />
  ...
</http>

```

21. Security HTTP Response Headers 译: 21.安全性HTTP响应头

本节讨论Spring Security对于为响应添加各种安全标头的支持。

21.1 Default Security Headers 译: 21.默认安全标题

Spring Security允许用户轻松地注入默认安全头以帮助保护他们的应用程序。Spring Security的默认设置是包含以下标题:

```

Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: 0
X-Content-Type-Options: nosniff
Strict-Transport-Security: max-age=31536000 ; includeSubDomains
X-Frame-Options: DENY
X-XSS-Protection: 1; mode=block

```



Strict-Transport-Security仅在HTTPS请求上添加

有关每个标题的更多详细信息，请参阅相应章节:

- [Cache Control](#)
- [Content Type Options](#)
- [HTTP Strict Transport Security](#)
- [X-Frame-Options](#)
- [X-XSS-Protection](#)

虽然这些标题都被认为是最佳实践，但应该指出，并非所有客户都使用标题，因此鼓励进行其他测试。

您可以自定义特定的标题。例如，假设您希望您的HTTP响应标头如下所示:

```
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: 0
X-Content-Type-Options: nosniff
X-Frame-Options: SAMEORIGIN
X-XSS-Protection: 1; mode=block
```

具体来说，您需要使用以下自定义设置的所有默认标题：

- [X-Frame-Options](#) to allow any request from same domain
- [HTTP Strict Transport Security \(HSTS\)](#) will not be added to the response

您可以使用以下Java配置轻松完成此操作：

```
@EnableWebSecurity
public class WebSecurityConfig extends
    WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            // ...
            .headers()
            .frameOptions().sameOrigin()
            .httpStrictTransportSecurity().disable();
    }
}
```

或者，如果您使用的是Spring Security XML Configuration，则可以使用以下内容：

```
<http>
<!-- ... -->

<headers>
<frame-options policy="SAMEORIGIN" />
<hsts disable="true"/>
</headers>
</http>
```

如果您不希望添加默认设置并希望明确控制应该使用的内容，则可以禁用默认设置。下面提供了一个基于Java和XML的配置示例：

如果您使用的是Spring Security的Java配置，以下将只添加 [Cache Control](#) 。

```
@EnableWebSecurity
public class WebSecurityConfig extends
    WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            // ...
            .headers()
            // do not use any default headers unless explicitly listed
            .defaultsDisabled()
            .cacheControl();
    }
}
```

以下XML将只添加 [Cache Control](#) 。

```
<http>
<!-- ... -->

<headers defaults-disabled="true">
<cache-control/>
</headers>
</http>
```

如有必要，可以使用以下Java配置禁用所有HTTP安全响应标头：

```
@EnableWebSecurity
public class WebSecurityConfig extends
    WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            // ...
            .headers().disable();
    }
}
```

如有必要，可以使用下面的XML配置禁用所有HTTP安全响应标头：

```
<http>
<!-- ... -->

<headers disabled="true" />
</http>
```

在过去，Spring Security要求您为您的Web应用程序提供自己的缓存控制。这在当时似乎是合理的，但浏览器缓存已经发展到包括用于安全连接的缓存。这意味着用户可以查看已认证的页面并注销，然后恶意用户可以使用浏览器历史记录查看缓存的页面。为了帮助缓解这个问题，Spring Security增加了缓存控制支持，它将在响应中插入以下头文件。

```
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: 0
```

只需添加没有子元素的<headers>元素将自动添加缓存控制和其他一些保护。但是，如果您只想缓存控制，则可以使用Spring Security的XML名称空间（<cache-control>元素和[(#)]属性）启用此功能。

```
<http>
<!-- ... -->

<headers defaults-disable="true">
  <cache-control />
</headers>
</http>
```

同样，您可以通过以下方式仅启用Java配置中的缓存控制：

```
@EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
        // ...
        .headers()
        .defaultsDisabled()
        .cacheControl();
    }
}
```

如果您真的想要缓存特定的响应，您的应用程序可以选择性地调用[HttpServletResponse.setHeader\(String,String\)](#)来覆盖Spring Security设置的标头。这对于确保像CSS，JavaScript和图像等正确缓存很有用。

在使用Spring Web MVC时，通常在您的配置中完成。例如，以下配置将确保为所有资源设置缓存标头：

```
@EnableWebMvc
public class WebMvcConfiguration implements WebMvcConfigurer {

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
        registry
        .addResourceHandler("/resources/**")
        .addResourceLocations("/resources/")
        .setCachePeriod(31556926);
    }

    // ...
}
```

21.1.2 Content Type Options 译：21.1.2的内容类型选项

历史上，包括Internet Explorer在内的浏览器会尝试使用[content sniffing](#)来猜测请求的内容类型。这允许浏览器通过猜测未指定内容类型的资源上的内容类型来改善用户体验。例如，如果浏览器遇到没有指定内容类型的JavaScript文件，它将能够猜测内容类型并执行它。



在允许上传内容时，还应该做许多其他事情（即，仅在独特的域中显示文档，确保设置Content-Type标头，清理文档等）。但是，这些措施超出了Spring Security提供的范围。指出禁用内容嗅探时，还必须指出内容类型以使事情正常工作。

内容嗅探的问题在于，这允许恶意用户使用polyglots（即，作为多种内容类型有效的文件）来执行XSS攻击。例如，某些网站可能允许用户向网站提交有效的postscript文档并查看它。恶意用户可能会创建[postscript document that is also a valid JavaScript file](#)并使用它执行XSS攻击。

内容嗅探可以通过将以下标题添加到我们的响应来禁用：

```
X-Content-Type-Options: nosniff
```

就像缓存控制元素一样，当使用<headers>元素而没有子元素时，nosniff指令默认添加。但是，如果您想要更多地控制添加哪个标题，可以使用<content-type-options>元素和[(#)]属性，如下所示：

```
<http>
<!-- ... -->

<headers defaults-disabled="true">
  <content-type-options />
</headers>
</http>
```

默认情况下，Spring Security Java配置会添加X-Content-Type-Options标头。如果您想更好地控制标题，可以使用以下内容显式指定内容类型选项：

```
@EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            // ...
            .headers()
            .defaultsDisabled()
            .contentTypeOptions();
    }
}
```

21.1.3 HTTP Strict Transport Security (HSTS) 译: 21.1.3 HTTP 严格传输安全性 (HSTS)

当您输入您的银行网站时，您输入mybank.example.com还是输入<https://mybank.example.com>？如果您省略了https协议，则可能会受到Man in the Middle attacks的攻击。即使该网站执行重定向至<https://mybank.example.com>，恶意用户也可以拦截最初的HTTP请求并操纵响应（即重定向至<https://mibank.example.com>并窃取其凭据）。

许多用户省略了https协议，这就是创建[HTTP Strict Transport Security \(HSTS\)](#)的原因。一旦将mybank.example.com添加为HSTS host，浏览器可以提前知道对mybank.example.com的任何请求应解释为<https://mybank.example.com>。这大大降低了发生中间人攻击的可能性。



根据[RFC6797](#)，HSTS报头仅被注入到HTTPS响应中。为了使浏览器确认标题，浏览器必须首先相信签署用于建立连接的SSL证书的CA（而不仅仅是SSL证书）。

将站点标记为HSTS主机的一种方法是将主机预加载到浏览器中。另一种方法是将“Strict-Transport-Security”标题添加到响应中。例如，以下内容将指示浏览器将域名视为一年的HSTS主机（一年大约有31536000秒）：

```
Strict-Transport-Security: max-age=31536000 ; includeSubDomains
```

可选的includeSubDomains指令指示Spring Security将子域（即secure.mybank.example.com）也视为HSTS域。

与其他标题一样，Spring Security默认添加HSTS。您可以使用<hsts>元素自定义HSTS标头，如下所示：

```
<http>
<!-- ... -->

<headers>
<hsts
  include-subdomains="true"
  max-age-seconds="31536000" />
</headers>
</http>
```

同样，您可以只启用带有Java配置的HSTS头文件：

```
@EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            // ...
            .headers()
            .httpStrictTransportSecurity()
            .includeSubdomains(true)
            .maxAgeSeconds(31536000);
    }
}
```

21.1.4 HTTP Public Key Pinning (HPKP) 译: 21.1.4 HTTP 公钥绑定 (HPKP)

HTTP Public Key Pinning (HPKP) 是一项安全功能，它告诉Web客户端将特定的加密公钥与特定的Web服务器相关联，以防止伪造的证书对中间人（MITM）的攻击。

为了确保在TLS会话中使用的服务器公钥的真实性，该公钥被封装到通常由证书颁发机构（CA）签署的X.509证书中。Web客户端（如浏览器）信任很多这些CA，它们都可以为任意域名创建证书。如果攻击者能够危害单个CA，他们可以对各种TLS连接执行MITM攻击。HPKP可以通过告诉客户端哪个公钥属于某个Web服务器来规避HTTPS协议的这种威胁。HPKP是首次使用信托（TOFU）技术。Web服务器第一次通过特殊的HTTP头告诉客户端公钥属于它时，客户端将这些信息存储一段给定的时间。当客户端再次访问服务器时，它需要一个包含指纹已经通过HPKP知道的公钥的证书。如果服务器提供未知的公钥，则客户端应向用户提供警告。



由于用户代理需要根据SSL证书链验证引脚，所以HPKP头只能注入到HTTPS响应中。

为您的站点启用此功能非常简单，只需在通过HTTPS访问您的站点时返回Public-Key-Pins HTTP标头即可。例如，以下内容将指示用户代理仅向2个引脚的指定URI（通过[report-uri](#)指令）报告引脚验证失败：

```
Public-Key-Pins-Report-Only: max-age=5184000 ; pin-sha256="d6qzRu9z0ECb90Uez27xwltNsje1Md7GkYYkVoZWmM=" ; pin-sha256="E9CZ9INDbd+2eRQozYqqbQ2yXLVBK9"
```

A [pin validation failure report](#)是标准的JSON结构，可以通过Web应用程序的自己的API或公开托管的HPKP报告服务（例如 [REPORT-URI](#)）捕获。

可选的includeSubDomains指令指示浏览器也使用给定引脚验证子域。

与其他标题相反，Spring Security默认不添加HPKP。您可以使用<hpkp>元素自定义HPKP标题，如下所示：

```
<http>
<!-- ... -->

<headers>
<hpkp
  include-subdomains="true"
  report-uri="http://example.net/pkp-report">
<pins>
  <pin algorithm="sha256">d6qzRu9z0ECb90Uez27xWlTnsj0e1Md7GkYYkVoZWmM=</pin>
  <pin algorithm="sha256">E9CZ9INDbd+2eRQozYqqbQ2yXLVB9+xcprMF+44U1g=</pin>
</pins>
</hpkp>
</headers>
</http>
```

同样，您可以使用Java配置启用HPKP标题：

```
@EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            // ...
            .headers()
                .httpPublicKeyPinning()
                    .includeSubdomains(true)
                    .reportUri("http://example.net/pkp-report")
                    .addSha256Pins("d6qzRu9z0ECb90Uez27xWlTnsj0e1Md7GkYYkVoZWmM=", "E9CZ9INDbd+2eRQozYqqbQ2yXLVB9+xcprMF+44U1g=");
    }
}
```

21.1.5 X-Frame-Options 译：21.1.5 X-Frame-Options

允许将您的网站添加到框架可能是一个安全问题。例如，使用聪明的CSS样式的用户可能会被欺骗点击他们不想要的东西（[video demo](#)）。例如，登录他们银行的用户可能会点击一个按钮，授予对其他用户的访问权限。这种攻击被称为[Clickjacking](#)。



另一种处理点击劫持的现代方法是使用 [Section 21.1.7, "Content Security Policy \(CSP\)"](#)。

有多种方法可以缓解点击劫持攻击。例如，要保护传统浏览器免受点击劫持攻击，您可以使用[frame breaking code](#)。虽然不完美，但对于传统浏览器来说，破解代码是最好的选择。

解决点击劫持的更现代的方法是使用 [X-Frame-Options](#) 标头：

```
X-Frame-Options: DENY
```

X-Frame-Options响应头指示浏览器阻止响应中的任何站点在帧中呈现。默认情况下，Spring Security会禁用iframe中的渲染。

您可以使用[frame-options](#)元素自定义X-Frame-Options。例如，以下将指示Spring Security使用允许同一域内的iframe的“X-Frame-Options: SAMEORIGIN”：

```
<http>
<!-- ... -->

<headers>
<frame-options
  policy="SAMEORIGIN" />
</headers>
</http>
```

同样，您可以使用以下方法自定义框架选项以在Java配置中使用相同的源：

```
@EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            // ...
            .headers()
                .frameOptions()
                    .sameOrigin();
    }
}
```

21.1.6 X-XSS-Protection 译：21.1.6 X-XSS保护

有些浏览器支持过滤掉[reflected XSS attacks](#)。这绝不是万无一失的，但有助于XSS保护。

默认情况下，过滤通常处于启用状态，因此添加标头通常会确保启用它并指示浏览器在检测到XSS攻击时该执行什么操作。例如，该过滤器可能会尝试以最小侵入方式更改内容，以继续呈现所有内容。有时候，这种类型的替换可能会变成[XSS vulnerability in itself](#)。相反，最好是阻止内容而不是尝试修复它。为此，我们可以添加以下标头：

```
X-XSS-Protection: 1; mode=block
```

该标头默认包含在内。但是，如果我们想要，我们可以定制它。例如：

```
<http>
<!-- ... -->

<headers>
<xss-protection block="false"/>
</headers>
</http>
```

同样，您可以使用以下方法在Java配置中自定义XSS保护：

```
@EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

@Override
protected void configure(HttpSecurity http) throws Exception {
    http
    // ...
    .headers()
    .xssProtection()
    .block(false);
}
}
```

21.1.7 Content Security Policy (CSP)译：21.1.7内容安全策略（CSP）

[Content Security Policy \(CSP\)](#)是Web应用程序可以利用的机制来缓解内容注入漏洞，例如跨站脚本（XSS）。CSP是一种声明性策略，为Web应用程序作者声明并最终通知客户端（用户代理）有关Web应用程序预期从中加载资源的来源提供了便利。



内容安全策略不是为了解决所有内容注入漏洞。相反，可以利用CSP来帮助减少内容注入攻击造成的危害。作为第一道防线，Web应用程序作者应验证其输入并对其输出进行编码。

Web应用程序可以通过在响应中包含以下HTTP标头之一来使用CSP：

- **Content-Security-Policy**
- **Content-Security-Policy-Report-Only**

这些标头中的每一个都用作向客户端传递安全策略的机制。安全策略包含一组安全策略指令（例如，`script-src`和`object-src`），每个安全策略指令都负责声明特定资源表示的限制。

例如，Web应用程序可以声明它希望通过在响应中包含以下标题来从特定的可信来源加载脚本：

```
Content-Security-Policy: script-src https://trustedscripts.example.com
```

尝试从脚本-src指令中声明的以外的其他源加载脚本的尝试将被用户代理阻止。此外，如果在安全策略中声明了`report-uri`指令，则该违规将由用户代理报告给声明的URL。

例如，如果Web应用程序违反了声明的安全策略，则以下响应标头将指示用户代理将违规报告发送到策略的`report-uri`指令中指定的URL。

```
Content-Security-Policy: script-src https://trustedscripts.example.com; report-uri /csp-report-endpoint/
```

[Violation reports](#)是标准的JSON结构，可以通过Web应用程序自己的API或公开托管的CSP违规报告服务（例如 [REPORT-URI](#)）捕获。

Content-Security-Policy-Report-Only标头为Web应用程序作者和管理员提供监视安全策略的能力，而不是强制执行它们。此标题通常用于为网站试验和/或开发安全策略。当一个策略被认为是有效的，它可以通过使用`Content-Security-Policy`头域来实施。

给定以下响应头，该策略声明脚本可以从两个可能来源之一加载。

```
Content-Security-Policy-Report-Only: script-src 'self' https://trustedscripts.example.com; report-uri /csp-report-endpoint/
```

如果该站点违反了此策略，则通过尝试从`evil.com`加载脚本，用户代理将向由`report-uri`指令指定的声明URL发送违规报告，但仍允许加载违规资源。

Configuring Content Security Policy译：配置内容安全策略

需要注意的是，Spring Security默认不会添加内容安全策略。Web应用程序作者必须声明安全策略以强制执行和/或监视受保护的资源。

例如，鉴于以下安全策略：

```
script-src 'self' https://trustedscripts.example.com; object-src https://trustedplugins.example.com; report-uri /csp-report-endpoint/
```

您可以使用`<content-security-policy>`元素的XML配置启用CSP头，如下所示：

```
<http>
<!-- ... -->

<headers>
<content-security-policy
  policy-directives="script-src 'self' https://trustedscripts.example.com; object-src https://trustedplugins.example.com; report-uri /csp-report-endpoint/"
/>
</headers>
</http>
```

要启用CSP“仅报告”标题，请按如下所示配置元素：


```
<http>
<!-- ... -->

<headers>
<content-security-policy
  policy-directives="script-src 'self' https://trustedscripts.example.com; object-src https://trustedplugins.example.com; report-uri /csp-report-enc
  report-only="true" />
</headers>
</http>
```

同样，您可以使用Java配置启用CSP头，如下所示：

```
@EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            // ...
            .headers()
                .contentSecurityPolicy("script-src 'self' https://trustedscripts.example.com; object-src https://trustedplugins.example.com; report-uri /csp-report
            )
        }
    }
}
```

要启用CSP“仅报告”标题，请提供以下Java配置：

```
@EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            // ...
            .headers()
                .contentSecurityPolicy("script-src 'self' https://trustedscripts.example.com; object-src https://trustedplugins.example.com; report-uri /csp-report
                .reportOnly();
            )
        }
    }
}
```

Additional Resources 译:其他资源

将内容安全策略应用于Web应用程序通常是一项不重要的任务。以下资源可能会为您的网站制定有效的安全策略提供进一步的帮助。

[An Introduction to Content Security Policy](#)

[CSP Guide - Mozilla Developer Network](#)

[W3C Candidate Recommendation](#)

21.1.8 Referrer Policy 译: 21.1.8 referrer 人政策

Referrer Policy是一种Web应用程序可以利用的机制来管理引用者字段，其中包含用户所在的最后一页。

Spring Security的方法是使用 **Referrer Policy** 标头，它提供了不同的 **policies**：

```
Referrer-Policy: same-origin
```

Referrer-Policy响应头指示浏览器让目的地知道用户以前的来源。

Configuring Referrer Policy 译:配置 Referrer 策略

默认情况下，Spring Security不 **添加** Referrer Policy标头。

您可以使用带有 **<referrer-policy>** 元素的XML配置启用Referrer-Policy头，如下所示：

```
<http>
<!-- ... -->

<headers>
<referrer-policy policy="same-origin" />
</headers>
</http>
```

同样，您可以使用Java配置启用Referrer Policy标头，如下所示：

```
@EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            // ...
            .headers()
                .referrerPolicy(ReferrerPolicy.SAME_ORIGIN);
            )
        }
    }
}
```

21.2 Custom Headers 译: 21.2自定义标头

Spring Security具有一些机制，可以方便地将更常见的安全性标题添加到应用程序中。不过，它也提供挂钩来启用添加自定义标题。

21.2.1 Static Headers 译: 21.2静态头

您可能有时希望将自定义安全标头插入到您的应用程序中，但不支持开箱即用。例如，给定以下自定义安全标头：

```
X-Custom-Security-Header: header-value
```

当使用XML命名空间时，可以使用 `<header>` 元素将这些头添加到响应中，如下所示：

```
<http>
<!-- ... -->

<headers>
  <header name="X-Custom-Security-Header" value="header-value"/>
</headers>
</http>
```

同样，可以使用Java Configuration将头添加到响应中，如下所示：

```
@EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            // ...
            .headers()
                .addHeaderWriter(new StaticHeadersWriter("X-Custom-Security-Header", "header-value"));
    }
}
```

21.2.2 Headers Writer 译: 21.2标题作者

当命名空间或Java配置不支持你想要的标题，您可以创建自定义 `HeadersWriter` 实例，甚至提供的自定义实现 `HeadersWriter`。

让我们来看一个使用 `XFrameOptionsHeaderWriter` 的自定义实例的 `XFrameOptionsHeaderWriter`。也许你想允许为相同的来源制作内容。这很容易通过将 `policy` 属性设置为“SAMEORIGIN”来支持，但让我们看看使用 `ref` 属性的更明确示例。

```
<http>
<!-- ... -->

<headers>
  <header ref="frameOptionsWriter"/>
</headers>
</http>
<!-- Requires the c-namespace.
See http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#beans-c-namespace
-->
<beans:bean id="frameOptionsWriter"
  class="org.springframework.security.web.header.writers.frameoptions.XFrameOptionsHeaderWriter"
  c:frameOptionsMode="SAMEORIGIN"/>
```

我们还可以通过Java配置将内容的框架限制在相同的原始位置：

```
@EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            // ...
            .headers()
                .addHeaderWriter(new XFrameOptionsHeaderWriter(XFrameOptionsMode.SAMEORIGIN));
    }
}
```

21.2.3 DelegatingRequestMatcherHeaderWriter 译: 21.2.3委托请求匹配器标题作者

有时您可能只想为某些请求编写头文件。例如，也许你只想保护你的登录页面不被陷害。你可以使用 `DelegatingRequestMatcherHeaderWriter` 这样做。在使用XML名称空间配置时，可以通过以下方式完成此操作：

```
<http>
<!-- ... -->

<headers>
<frame-options disabled="true"/>
<header ref="headerWriter"/>
</headers>
</http>

<beans:bean id="headerWriter"
class="org.springframework.security.web.header.writers.DelegatingRequestMatcherHeaderWriter">
<beans:constructor-arg>
<bean class="org.springframework.security.web.util.matcher.AntPathRequestMatcher"
c:pattern="/login"/>
</beans:constructor-arg>
<beans:constructor-arg>
<beans:bean
class="org.springframework.security.web.header.writers.frameoptions.XFrameOptionsHeaderWriter"/>
</beans:constructor-arg>
</beans:bean>
```

我们还可以使用java配置防止内容成帧到登录页面：

```
@EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

@Override
protected void configure(HttpSecurity http) throws Exception {
RequestMatcher matcher = new AntPathRequestMatcher("/login");
DelegatingRequestMatcherHeaderWriter headerWriter =
new DelegatingRequestMatcherHeaderWriter(matcher,new XFrameOptionsHeaderWriter());
http
// ...
.headers()
.frameOptions().disabled()
.addHeaderWriter(headerWriter);
}
}
```

22. Session Management 译：22会话管理

与HTTP会话相关的功能由过滤器委托给的 `SessionManagementFilter` 和 `SessionAuthenticationStrategy` 接口的组合来处理。典型用法包括会话固定保护攻击预防，会话超时检测以及限制已验证用户同时打开的会话数。

22.1 SessionManagementFilter 译：22.1 SessionManagementFilter

所述 `SessionManagementFilter` 检查的内容 `SecurityContextRepository` 针对的当前内容 `SecurityContextHolder`，以确定是否用户已经在当前请求期间被认证，通常由非交互式认证机制，如预认证或记住-ME ^[17]。如果存储库包含安全上下文，则该过滤器不执行任何操作。如果不是，并且线程本地 `SecurityContext` 包含（非匿名）`Authentication` 对象，则筛选器将假定它们已由堆栈中的上一个筛选器进行了身份验证。它会调用配置的 `SessionAuthenticationStrategy`。

如果用户当前未通过身份验证，则筛选器将检查是否请求了无效的会话ID（例如由于超时），并且将调用已配置的 `InvalidSessionStrategy`（如果已设置）。最常见的行为就是重定向到一个固定的URL，并将其封装在标准实现 `SimpleRedirectInvalidSessionStrategy`。当通过命名空间 [as described earlier](#) 配置无效会话URL时，也使用后者。

22.2 SessionAuthenticationStrategy 译：22.2 SessionAuthenticationStrategy

`SessionAuthenticationStrategy` 由 `SessionManagementFilter` 和 `AbstractAuthenticationProcessingFilter`，因此如果您使用自定义的表单登录类，则需要将其注入到这两个类中。在这种情况下，组合命名空间和自定义Bean的典型配置可能如下所示：

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

<beans:bean id="myAuthFilter" class=
"org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter">
<beans:property name="sessionAuthenticationStrategy" ref="sas" />
...
</beans:bean>

<beans:bean id="sas" class=
"org.springframework.security.web.authentication.session.SessionFixationProtectionStrategy" />
```

请注意，如果要在实现 `HttpSessionBindingListener` 的会话（包括Spring会话范围的Bean）中存储Bean，则使用默认值 `SessionFixationProtectionStrategy` 可能会导致问题。有关更多信息，请参阅此类的Javadoc。

22.3 Concurrency Control 译：22.3并发控制

Spring Security能够防止委托人同时对同一应用程序进行超过指定次数的身份验证。许多独立软件开发商利用这一点来强制执行许可，而网络管理员喜欢这种功能，因为它有助于防止人们共享登录名。例如，您可以停止用户“蝙蝠侠”从两个不同的会话登录到Web应用程序。您可以使其以前的登录失效，或者在他们尝试再次登录时报告错误，从而阻止第二次登录。请注意，如果您使用的是第二种方法，那么未明确注销的用户（例如，刚刚关闭浏览器的用户）将无法再次登录，直到其原始会话过期。

并发控制由名称空间支持，因此请检查较早的名称空间章节以获取最简单的配置。有时候你需要定制一些东西。

该实现使用 `SessionAuthenticationStrategy` 的专用版本，称为 `ConcurrentSessionControlAuthenticationStrategy`。



以前，并验证检查由 `ProviderManager`，可以使用 `ConcurrentSessionController` 注入。后者会检查用户是否尝试超出允许的会话数量。但是，这种方法需要事先创建HTTP会话，这是不可取的。在Spring Security 3中，用户首先通过 `AuthenticationManager` 进行身份验证，一旦他们成功通过身份验证，将创建一个会话并检查是否允许他们打开另一个会话。

要使用并发会话支持，您需要将以下内容添加到 `web.xml`：

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

另外，您需要将 `ConcurrentSessionFilter` 添加到您的 `FilterChainProxy`。`ConcurrentSessionFilter` 需要两个构造函数参数 `sessionRegistry`，它通常指向 `SessionRegistryImpl` 和 `sessionInformationExpiredStrategy` 的实例，它定义了会话过期时应用的策略。使用命名空间创建 `FilterChainProxy` 和其他默认bean的配置可能如下所示：

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

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

<beans:bean id="redirectSessionInformationExpiredStrategy"
class="org.springframework.security.web.session.SimpleRedirectSessionInformationExpiredStrategy">
<beans:constructor-arg name="invalidSessionUrl" value="/session-expired.htm" />
</beans:bean>

<beans:bean id="concurrencyFilter"
class="org.springframework.security.web.session.ConcurrentSessionFilter">
<beans:constructor-arg name="sessionRegistry" ref="sessionRegistry" />
<beans:constructor-arg name="sessionInformationExpiredStrategy" ref="redirectSessionInformationExpiredStrategy" />
</beans:bean>

<beans:bean id="myAuthFilter" class=
"org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter">
<beans:property name="sessionAuthenticationStrategy" ref="sas" />
<beans:property name="authenticationManager" ref="authenticationManager" />
</beans:bean>

<beans:bean id="sas" class="org.springframework.security.web.authentication.session.CompositeSessionAuthenticationStrategy">
<beans:constructor-arg>
<beans:list>
<beans:bean class="org.springframework.security.web.authentication.session.ConcurrentSessionControlAuthenticationStrategy">
<beans:constructor-arg ref="sessionRegistry"/>
<beans:property name="maximumSessions" value="1" />
<beans:property name="exceptionIfMaximumExceeded" value="true" />
</beans:bean>
<beans:bean class="org.springframework.security.web.authentication.session.SessionFixationProtectionStrategy">
</beans:bean>
<beans:bean class="org.springframework.security.web.authentication.session.RegisterSessionAuthenticationStrategy">
<beans:constructor-arg ref="sessionRegistry"/>
</beans:bean>
</beans:list>
</beans:constructor-arg>
</beans:bean>

<beans:bean id="sessionRegistry"
class="org.springframework.security.core.session.SessionRegistryImpl" />
```

每当 `HttpSession` 开始或终止时，将收听者添加到 `web.xml` 导致 `ApplicationEvent` 发布到Spring `ApplicationContext`。这很关键，因为它允许在会话结束时通知 `SessionRegistryImpl`。如果没有它，即使用户退出其他会话或超时，用户将永远无法再次重新登录。

22.3.1 Querying the SessionRegistry for currently authenticated users and their sessions 译：22.3.1 查询当前通过身份验证的用户及其会话的 SessionRegistry

通过命名空间或使用普通bean来设置并发控制有一个有用的副作用，`SessionRegistry` 您提供对您可以在应用程序中直接使用的 `SessionRegistry` 的引用，因此即使您不想限制数量用户可能拥有的会话，无论如何，建立基础设施可能是值得的。您可以将 `maximumSession` 属性设置为-1以允许无限制的会话。如果您正在使用名称空间，则可以使用 `session-registry-alias` 属性为内部创建的 `SessionRegistry` 设置别名，从而提供可以注入到自己的Bean中的引用。

`getAllPrincipals()` 方法为您提供当前已通过身份验证的用户列表。您可以通过调

用 `getAllSessions(Object principal, boolean includeExpiredSessions)` 方法列出用户的会话，该方法返回 `SessionInformation` 对象的列表。您也可以通过调用过期一个用户 `ae™` 的会议 `expireNow()` 上 `SessionInformation` 实例。当用户返回到应用程序时，将阻止他们继续进行。例如，您可以在管理应用程序中找到这些方法。查看Javadoc以获取更多信息。

[17] 验证由执行认证（诸如表单登录）之后重定向机制将不会受到被检测 `SessionManagementFilter`，作为过滤器将不会认证请求期间被调用。会话管理功能必须在这种情况下单独处理。

23. Anonymous Authentication 译：23匿名身份验证

23.1 Overview 译：23.1概述

通常认为采用“默认拒绝”的安全措施是很好的，您可以在其中明确指定允许的内容并拒绝其他所有内容。定义未经身份验证的用户可访问的内容也是类似的情况，特别是对于Web应用程序。许多网站要求用户必须通过除少数网址以外的其他任何验证（例如，家庭和登录页面）。在这种情况下，为这些特定的URL定义访问配置属性是最容易的，而不是针对每个安全资源。换句话说，有时很高兴地说 `ROLE_SOMETHING` 是默认需要的，并且只允许某些例外情况，例如登录，注销和应用程序的主页。您也可以完全忽略过滤器链中的这些页面，从而绕过访问控制检查，但这可能因其他原因而不受欢迎，特别是如果页面对经过身份验证的用户的行为不同。

这就是我们所说的匿名认证。请注意，“匿名身份证”的用户和未经身份证的用户之间并没有真正的概念区别。Spring Security的匿名身份证为您提供了一种更方便的配置访问控制属性的方式。例如，调用servlet API调用（如 `getCallerPrincipal`），即使 `SecurityContextHolder` 实际存在匿名身份证对象，该 `SecurityContextHolder` 仍将返回 `null`。

在其他情况下，匿名身份证很有用，例如，当审计拦截器查询 `SecurityContextHolder` 以确定哪个主体负责给定操作时。类可以更稳健着，如果他们知道 `SecurityContextHolder` 总是包含 `Authentication` 对象，从不 `null`。

23.2 Configuration 译：23.2配置

匿名认证支持是在使用HTTP配置Spring Security 3.0时自动提供的，并且可以使用 `<anonymous>` 元素进行自定义（或禁用）。除非你使用传统的bean配置，否则你不需要配置这里描述的bean。

三个类共同提供匿名身份证验证功能。`AnonymousAuthenticationToken` 是一个实现 `Authentication`，并存储 `GrantedAuthority` S的适用于匿名校长。有相应的 `AnonymousAuthenticationProvider`，被链接到 `ProviderManager` 因此 `AnonymousAuthenticationToken` 被接受。最后，还有一个 `AnonymousAuthenticationFilter`，这是正常的认证机制后，链接，并自动添加一个 `AnonymousAuthenticationToken` 至 `SecurityContextHolder` 如果没有现有 `Authentication` 在那里举行。筛选器和身份证验证提供程序的定义如下所示：

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

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

key 由过滤器和身份证验证提供程序共享，以便由后者创建的令牌被后者^[18]接受。该 `userAttribute` 中的形式来表 达 `usernameInTheAuthenticationToken,grantedAuthority[,grantedAuthority]`。这是相同的语法等号后对所使用 `userMap` 财产 `InMemoryDaoImpl`。

如前所述，匿名身份证的好处是所有的URI模式都可以应用于它们。例如：

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

23.3 AuthenticationTrustResolver 译：23.3 AuthenticationTrustResolver

四舍五入的匿名身份证验证讨论是 `AuthenticationTrustResolver` 接口，以及相应的 `AuthenticationTrustResolverImpl` 实现。该接口提供了一种 `isAnonymous(Authentication)` 方法，它允许感兴趣的类考虑这种特殊类型的身份证验证状态。`ExceptionTranslationFilter` 在处理 `AccessDeniedException` 使用此接口。如果抛出了 `AccessDeniedException`，并且身份证是匿名类型，则不会抛出403（禁止）响应，而是过滤器将启动 `AuthenticationEntryPoint` 以便委托人可以正确进行身份证验证。这是必要的区别，否则校长将永远被视为“已认证”，永远不会有有机会通过表单，基本，摘要或其他正常认证机制进行登录。

您经常 `ROLE_ANONYMOUS` 上述拦截器配置中的 `ROLE_ANONYMOUS` 属性替换为 `IS_AUTHENTICATED_ANONYMOUSLY`，这在定义访问控制时实际上是同样的事情。这是使用 `AuthenticatedVoter` 一个例子，我们将在 [authorization chapter](#) 中看到。它使用 `AuthenticationTrustResolver` 来处理这个特定的配置属性并授予对匿名用户的访问权限。`AuthenticatedVoter` 方法更强大，因为它可以让您区分匿名，记住我和完全认证的用户。如果你不需要这个功能，那么你可以坚持 `ROLE_ANONYMOUS`，这将由Spring Security的标准 `RoleVoter`。

^[18] **key** 财产的使用不应被视为提供任何真正的安全。这仅仅是一个记账练习。如果您正在共享 `ProviderManager` 其中包含 `AnonymousAuthenticationProvider` 在这样一个场景，这是可能的身份证验证客户构建 `Authentication` 对象（如与RMI调用），然后一个恶意的客户端可以提交一个 `AnonymousAuthenticationToken` 它本身已经创造（与选择用户名和权限列表）。如果 **key** 是可猜测的或可以发现，那么该令牌将被匿名提供者接受。这是正常使用的问题，但如果您使用的是RMI，最好使用自定义的 `ProviderManager`，它省略了匿名提供程序，而不是共享您用于HTTP身份证验证机制的那个。

24. WebSocket Security 译：24 WebSocket安全性

Spring Security 4增加了对安全Spring's [WebSocket support](#)的支持。本节介绍如何使用Spring Security的WebSocket支持。



您可以在 `samples / javaconfig / chat` 中找到WebSocket安全性的完整工作示例。

直接JSR-356支持

Spring Security没有提供直接的JSR-356支持，因为这样做没有多大价值。这是因为格式是未知的，所以有 `little Spring can do to secure an unknown format`。另外，JSR-356不提供拦截消息的方法，所以安全性会相当侵入。

24.1 WebSocket Configuration 译：24.1 WebSocket配置

Spring Security 4.0通过Spring消息抽象为WebSockets引入了授权支持。要使用Java Configuration配置授权，只需扩展 `AbstractSecurityWebSocketMessageBrokerConfigurer` 并配置 `MessageSecurityMetadataSourceRegistry`。例如：

```
@Configuration
public class WebSocketSecurityConfig
    extends AbstractSecurityWebSocketMessageBrokerConfigurer { ❶ ❷

    protected void configureInbound(MessageSecurityMetadataSourceRegistry messages) {
        messages
            .simpDestMatchers("/user/*").authenticated() ❸
    }
}
```

这将确保：

- ❶ 任何入站CONNECT消息都需要有效的CSRF令牌才能执行 [Same Origin Policy](#)
- ❷ SecurityContextHolder用simpUser标头属性中的用户填充任何入站请求。
- ❸ 我们的信息需要适当的授权。具体来说，任何以“/user/”开头的入站消息都需要ROLE_USER。有关授权的更多详细信息，请参阅[Section 24.3, “WebSocket Authorization”](#)

Spring Security还为保护WebSocket提供了XML Namespace支持。可比的基于XML的配置如下所示：

```
<websocket-message-broker> ❶ ❷
  ❸
  <intercept-message pattern="/user/*" access="hasRole('USER')"/>
</websocket-message-broker>
```

这将确保：

- ❶ 任何入站CONNECT消息都需要一个有效的CSRF令牌来执行 [Same Origin Policy](#)
- ❷ SecurityContextHolder用simpUser标头属性中的用户填充任何入站请求。
- ❸ 我们的信息需要适当的授权。具体来说，任何以“/user/”开头的入站消息都需要ROLE_USER。有关授权的更多详细信息，请参阅[Section 24.3, “WebSocket Authorization”](#)

24.2 WebSocket Authentication 译：24.2 WebSocket身份验证

当WebSocket连接建立时，WebSocket重复使用与HTTP请求中相同的认证信息。这意味着Principal上的HttpServletRequest将被移交给WebSockets。如果您使用的是Spring Security，则自动覆盖Principal上的HttpServletRequest。

更具体地说，为了确保用户已经对WebSocket应用程序进行了身份验证，所有必需的是确保您设置Spring Security来验证您的基于HTTP的Web应用程序。

24.3 WebSocket Authorization 译：24.3 WebSocket授权

Spring Security 4.0通过Spring消息抽象为WebSockets引入了授权支持。要使用Java配置来配置授权，只需扩展AbstractSecurityWebSocketMessageBrokerConfigurer并配置MessageSecurityMetadataSourceRegistry。例如：

```
@Configuration
public class WebSocketSecurityConfig extends AbstractSecurityWebSocketMessageBrokerConfigurer {

    @Override
    protected void configureInbound(MessageSecurityMetadataSourceRegistry messages) {
        messages
            .nullDestMatcher().authenticated() ❶
            .simpSubscribeDestMatchers("/user/queue/errors").permitAll() ❷
            .simpDestMatchers("/app/*").hasRole("USER") ❸
            .simpSubscribeDestMatchers("/user/*", "/topic/friends/*").hasRole("USER") ❹
            .simpTypeMatchers(MESSAGE, SUBSCRIBE).denyAll() ❺
            .anyMessage().denyAll(); ❻
    }
}
```

这将确保：

- ❶ 任何没有目的地的消息（即消息类型为MESSAGE或SUBSCRIBE以外的任何消息）都需要用户进行身份验证
- ❷ 任何人都可以订阅/user/queue/errors
- ❸ 任何具有以“/app/”开头的目标邮件都将要求用户具有角色ROLE_USER
- ❹ 任何以“/user/”或“/topic/friends/”开头且类型为SUBSCRIBE的消息都需要ROLE_USER
- ❺ 任何其他类型为MESSAGE或SUBSCRIBE的消息都会被拒绝。由于6，我们不需要这一步，但它说明了如何匹配特定的消息类型。
- ❻ 其他消息被拒绝。这是确保您不会错过任何消息的好主意。

Spring Security还提供XML Namespace支持来保护WebSockets。可比的基于XML的配置如下所示：

```
<websocket-message-broker>
  ❶
  <intercept-message type="CONNECT" access="permitAll"/>
  <intercept-message type="UNSUBSCRIBE" access="permitAll"/>
  <intercept-message type="DISCONNECT" access="permitAll"/>

  <intercept-message pattern="/user/queue/errors" type="SUBSCRIBE" access="permitAll"/> ❷
  <intercept-message pattern="/app/*" access="hasRole('USER')"/> ❸

  ❹
  <intercept-message pattern="/user/*" access="hasRole('USER')"/>
  <intercept-message pattern="/topic/friends/*" access="hasRole('USER')"/>

  ❺
  <intercept-message type="MESSAGE" access="denyAll"/>
  <intercept-message type="SUBSCRIBE" access="denyAll"/>

  <intercept-message pattern="/*" access="denyAll"/> ❻
</websocket-message-broker>
```


这将确保：

- 1 任何类型为CONNECT, UNSUBSCRIBE或DISCONNECT的消息都需要用户进行身份验证
- 2 任何人都可以订阅/user / queue / errors
- 3 任何具有以"/ app /"开头的目标邮件都将要求用户具有角色ROLE_USER
- 4 任何以"/ user /"或"/ topic / friends /"开头且类型为SUBSCRIBE的消息都需要ROLE_USER
- 5 任何其他类型为MESSAGE或SUBSCRIBE的消息都会被拒绝。由于6，我们不需要这一步，但它说明了如何匹配特定的消息类型。
- 6 任何其他目的地的邮件都会被拒绝。这是确保您不会错过任何消息的好主意。

24.3.1 WebSocket Authorization Notes 译：24.3.1 WebSocket授权说明

为了正确保护你的应用程序，理解Spring的WebSocket支持是很重要的。

WebSocket Authorization on Message Types 译：消息类型的WebSocket授权

了解SUBSCRIBE和MESSAGE消息类型之间的区别以及它在Spring中的工作方式非常重要。

考虑一个聊天应用程序。

- The system can send notifications MESSAGE to all users through a destination of "/topic/system/notifications"
- Clients can receive notifications by SUBSCRIBE to the "/topic/system/notifications".

虽然我们希望客户能够SUBSCRIBE到"/ topic / system / notifications"，但我们不想让他们发送MESSAGE到目的地。如果我们允许发送MESSAGE到"/ topic / system / notifications"，则客户端可以直接向该端点发送消息并模拟系统。

通常，应用程序通常拒绝发送 给以broker prefix（即"/ topic /"或"/ queue /"）开头的消息的MESSAGE。

WebSocket Authorization on Destinations 译：目标上的WebSocket授权

了解目的地如何转变也很重要。

考虑一个聊天应用程序。

- Users can send messages to a specific user by sending a message to the destination of "/app/chat".
- The application sees the message, ensures that the "from" attribute is specified as the current user (we cannot trust the client).
- The application then sends the message to the recipient using `SimpMessageSendingOperations.convertAndSendToUser("toUser", "/queue/messages", message)`.
- The message gets turned into the destination of "/queue/user/messages-<sessionid>"

通过上面的应用程序，我们希望允许我们的客户端收听转换为"/ queue / user / messages- <sessionid>"的"/ user / queue"。但是，我们不希望客户端能够收听"/ queue / ""，因为这将允许客户端查看每个用户的消息。

通常，应用程序通常拒绝发送到以broker prefix（即"/ topic /"或"/ queue /"）开头的消息的SUBSCRIBE。当然，我们可能会提供例外来解释诸如此类的事情

24.3.2 Outbound Messages 译：24.3.2出站消息

Spring包含一个标题为Flow of Messages的部分，它描述了消息如何流经系统。值得注意的是，Spring Security只保护了clientInboundChannel。Spring Security不会尝试保护clientOutboundChannel。

最重要的原因是性能。对于每一条消息，通常会有更多消息传出。我们鼓励确保订阅端点，而不是保护出站消息。

24.4 Enforcing Same Origin Policy 译：24.4执行相同的原始策略

强调浏览器不强制Same Origin Policy用于WebSocket连接是非常重要的。这是一个非常重要的考虑因素。

24.4.1 Why Same Origin? 译：24.4.1为什么 同源？

考虑以下情况。用户访问bank.com并向其帐户进行身份验证。同一用户在其浏览器中打开另一个选项卡并访问evil.com。同源策略确保evil.com无法读取或写入bank.com数据。

使用WebSockets相同的来源策略不适用。事实上，除非bank.com明确禁止它，否则evil.com可以代表用户读取和写入数据。这意味着用户可以通过webSocket完成任何操作（即转账），evil.com可以代表该用户进行操作。

由于SockJS试图模拟WebSockets，它也绕过了同源策略。这意味着开发人员在使用SockJS时需要明确地保护他们的应用程序免受外部域的攻击

24.4.2 Spring WebSocket Allowed Origin 译：24.4.2 Spring WebSocket允许的来源

幸运的是，自Spring 4.1.5 Spring的WebSocket和SockJS支持限制访问current domain。Spring Security增加了额外的保护层以提供defence in depth。

24.4.3 Adding CSRF to Stomp Headers 译：24.4.3将CSRF添加到Stomp头部

默认情况下，Spring Security需要CSRF token中的任何CONNECT消息类型。这确保只有可访问CSRF令牌的站点才能连接。由于只有同一来源可以访问CSRF令牌，因此不允许外部域进行连接。

通常，我们需要将CSRF令牌包含在HTTP标头或HTTP参数中。但是，SockJS不允许使用这些选项。相反，我们必须在Stomp标头中包含令牌

应用程序可以obtain a CSRF token通过访问请求属性命名_csrf。例如，以下将允许访问JSP中的CsrfToken：

```
var headerName = "${_csrf.headerName}";
var token = "${_csrf.token}";
```

如果您使用的是静态HTML，则可以在REST端点上公开CsrfToken。例如，以下将揭露CsrfToken的URL / CSRF

```
@RestController
public class CsrfController {

    @RequestMapping("/csrf")
    public CsrfToken csrf(CsrfToken token) {
        return token;
    }
}
```

JavaScript可以对端点进行REST调用，并使用响应填充headerName和令牌。

我们现在可以在Stomp客户端中包含令牌。例如：

```
...
var headers = {};
headers[headerName] = token;
stompClient.connect(headers, function(frame) {
    ...
}
```

24.4.4 Disable CSRF within WebSockets 译：24.4.4在WebSockets中禁用CSRF

如果你想允许其他域访问你的网站，你可以禁用Spring Security的保护。例如，在Java配置中，您可以使用以下内容：

```
@Configuration
public class WebSocketSecurityConfig extends AbstractSecurityWebSocketMessageBrokerConfigurer {

    ...

    @Override
    protected boolean sameOriginDisabled() {
        return true;
    }
}
```

24.5 Working with SockJS 译：24.5使用SockJS

[SockJS](#)提供回退传输以支持旧版浏览器。在使用后备选项时，我们需要放松一些安全约束，以允许SockJS与Spring Security合作。

24.5.1 SockJS & frame-options 译：24.5.1 SockJS和框架选项

SockJS可能使用[transport that leverages an iframe](#)。默认情况下，Spring Security会[将该网站](#)从框架中中止，以防止点击劫持攻击。为了允许基于SockJS框架的传输工作，我们需要配置Spring Security以允许相同的来源构造内容。

您可以使用[frame-options](#)元素自定义X-Frame-Options。例如，以下将指示Spring Security使用允许同一域内的iframe的“X-Frame-Options: SAMEORIGIN”：

```
<http>
  <!-- ... -->

  <headers>
    <frame-options
      policy="SAMEORIGIN" />
    </headers>
</http>
```

同样，您可以使用以下方法自定义框架选项以在Java配置中使用相同的源：

```
@EnableWebSecurity
public class WebSecurityConfig extends
    WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            // ...
            .headers()
                .frameOptions()
                    .sameOrigin();
    }
}
```

24.5.2 SockJS & Relaxing CSRF 译：24.5.2 SockJS& Relaxing CSRF

对于任何基于HTTP的传输，SockJS在CONNECT消息上使用POST。通常，我们需要将CSRF令牌包含在HTTP标头或HTTP参数中。但是，SockJS不允许使用这些选项。相反，我们必须将标记包含在Stomp标头中，如[Section 24.4.3, “Adding CSRF to Stomp Headers”中所述](#)。

这也意味着我们需要通过Web层来放松我们的CSRF保护。具体而言，我们希望为我们的连接网址禁用CSRF保护。我们不希望为每个网址禁用CSRF保护。否则我们的网站将容易受到CSRF攻击。

我们可以通过提供CSRF RequestMatcher轻松实现此目的。我们的Java配置使得这非常简单。例如，如果我们的脚步终端是“/ chat”，那么我们可以使用以下配置仅对以“/ chat /”开头的URL禁用CSRF保护：

```

@Configuration
@EnableWebSecurity
public class WebSecurityConfig
    extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {

        http
            .csrf()
            // ignore our stomp endpoints since they are protected using Stomp headers
            .ignoringAntMatchers("/chat/**")
            .and()
            .headers()
            // allow same origin to frame our site to support iframe SockJS
            .frameOptions().sameOrigin()
            .and()
            .authorizeRequests()

            ...
    }
}

```

如果我们使用基于XML的配置，我们可以使用[(#)]。例如：

```

<http ...>
  <csrf request-matcher-ref="csrfMatcher"/>

  <headers>
    <frame-options policy="SAMEORIGIN"/>
  </headers>

  ...
</http>

<b:bean id="csrfMatcher"
  class="AndRequestMatcher">
  <b:constructor-arg value="#{T(org.springframework.security.web.csrf.CsrfFilter).DEFAULT_CSRF_MATCHER}"/>
  <b:constructor-arg>
    <b:bean class="org.springframework.security.web.util.matcher.NegatedRequestMatcher">
      <b:bean class="org.springframework.security.web.util.matcher.AntPathRequestMatcher">
        <b:constructor-arg value="/chat/**"/>
      </b:bean>
    </b:bean>
  </b:constructor-arg>
</b:bean>

```

Part V. Authorization 译: 第五部分授权

Spring Security中的高级授权功能是其受欢迎最引人注目的原因之一。无论您选择如何进行身份验证 - 无论是使用Spring Security提供的机制和提供程序，还是与容器或其他非Spring Security身份验证机构集成 - 您都会发现授权服务可以在您的应用程序中以一致且简单的方式使用办法。

在本部分中，我们将探讨第一部分介绍的各种 `AbstractSecurityInterceptor` 实现。然后我们继续探讨如何通过使用域访问控制列表来微调授权。

25. Authorization Architecture 译: 25授权体系结构

25.1 Authorities 译: 25.1权限

正如我们在[technical overview](#)中看到的，所有 `Authentication` 实现都存储了 `GrantedAuthority` 对象的列表。这些代表已经授予校长的权力。的 `GrantedAuthority` 对象被插入到 `Authentication` 对象由 `AuthenticationManager` 和由稍后读 `AccessDecisionManager` 小号作出授权决策时。

`GrantedAuthority` 是一个只有一个方法的接口：

```
String getAuthority();
```

此方法允许 `AccessDecisionManager` s 到获得精确的 `String` 所述的表示 `GrantedAuthority`。通过返回 `String` 的表示， `GrantedAuthority` 可以被大多数 `AccessDecisionManager` 轻松“读取”。如果 `GrantedAuthority` 不能精确地表示为 `String`，则 `GrantedAuthority` 被视为“复杂”，并且 `getAuthority()` 必须返回 `null`。

“复杂” `GrantedAuthority` 一个示例将是一个存储适用于不同客户帐号的操作和权限阈值列表的实施。将这个复杂的 `GrantedAuthority` 为 `String` 将非常困难，因此 `getAuthority()` 方法应该返回 `null`。这将向任何 `AccessDecisionManager` 表明它将需要特别支持 `GrantedAuthority` 实现以了解其内容。

Spring Security包含一个具体的 `GrantedAuthority` 实现， `SimpleGrantedAuthority`。这允许任何用户指定的 `String` 转换为 `GrantedAuthority`。安全体系结构中包含的所有 `AuthenticationProvider` 使用 `SimpleGrantedAuthority` 来填充 `Authentication` 对象。

25.2 Pre-Invocation Handling 译: 25.2预调用处理

正如我们在[Technical Overview](#)一章中看到的那样，Spring Security提供了拦截器来控制对安全对象的访问，如方法调用或Web请求。关于是否允许继续调用的预调用决定由 `AccessDecisionManager`。

25.2.1 The AccessDecisionManager 译: 25.2.1 AccessDecisionManager

该 `AccessDecisionManager` 被称为 `AbstractSecurityInterceptor`，负责作最终访问控制决策。 `AccessDecisionManager` 界面包含三种方法：

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

boolean supports(ConfigAttribute attribute);

boolean supports(Class clazz);
```

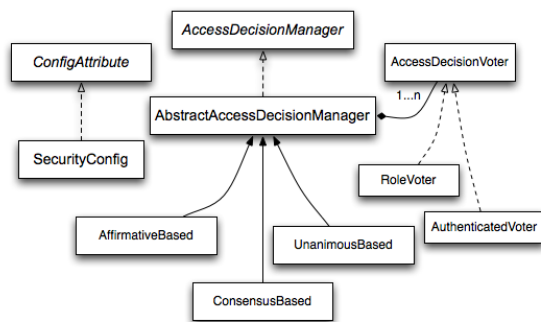
该 `AccessDecisionManager` 的 `decide` 方法通过了所有它为了使授权决策所需要的相关信息。特别是，通过传递安全 `Object` 可以检查包含在实际安全对象调用中的那些参数。例如，假设安全对象是 `MethodInvocation`。查询 `MethodInvocation` 任何 `Customer` 参数很容易，然后在 `AccessDecisionManager` 实施某种安全逻辑，以确保允许委托人对该客户进行操作。如果访问被拒绝，实现预计会抛出 `AccessDeniedException`。

`supports(ConfigAttribute)` 方法在启动时由 `AbstractSecurityInterceptor` 调用，以确定 `AccessDecisionManager` 可以处理传递的 `ConfigAttribute`。
`supports(Class)` 方法由安全拦截器实现调用，以确保配置的 `AccessDecisionManager` 支持安全拦截器将呈现的安全对象的类型。

25.2.2 Voting-Based AccessDecisionManager Implementations 译：25.2.2 基于投票的 AccessDecisionManager 实现

虽然用户可以实现自己的 `AccessDecisionManager` 来控制授权的各个方面，但 Spring Security 包括基于投票的几个 `AccessDecisionManager` 实现。Figure 25.1, “Voting Decision Manager” 说明了相关的类。

图 25.1. 投票决策经理



使用这种方法，一系列 `AccessDecisionVoter` 实现在授权决策上进行轮询。该 `AccessDecisionManager` 然后决定是否要抛出 `AccessDeniedException` 基于其的选票评估。

`AccessDecisionVoter` 接口有三种方法：

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

boolean supports(ConfigAttribute attribute);

boolean supports(Class clazz);
```

具体实现返回一个 `int`，与正反映在可能值 `AccessDecisionVoter` 静态字段 `ACCESS_ABSTAIN`，`ACCESS_DENIED` 和 `ACCESS_GRANTED`。如果对授权决定没有意见，投票实施将返回 `ACCESS_ABSTAIN`。如果确实有意见，则必须返回 `ACCESS_DENIED` 或 `ACCESS_GRANTED`。

Spring Security 提供了三个具体的 `AccessDecisionManager`，用来计算选票。`ConsensusBased` 实施将基于非弃权票的共识授予或拒绝访问。提供属性是为了在票数相等的情况下控制行为或者如果所有票都弃权。如果收到一个或多个 `ACCESS_GRANTED` 投票（即拒绝投票将被忽略，只要至少有一次投票），则 `AffirmativeBased` 实施将授予访问权限。像 `ConsensusBased` 实现一样，如果所有选民都弃权，则会有一个参数控制行为。`UnanimousBased` 提供者期望一致的 `ACCESS_GRANTED` 表决以授予访问权限，忽略弃权。如果有任何 `ACCESS_DENIED` 投票，它将拒绝访问。与其他实现一样，如果所有选民都弃权，则会有一个参数来控制行为。

有可能实现一个自定义的 `AccessDecisionManager`，以不同的方式计票。例如，特定 `AccessDecisionVoter` 可能会得到额外的权重，而来自特定选民的拒绝投票可能会产生否决效应。

RoleVoter 译：RoleVoter

Spring Security 提供的最常用的 `AccessDecisionVoter` 是简单的 `RoleVoter`，它将配置属性视为简单的角色名称，如果用户已分配角色，则投票授予访问权限。

如果任何 `ConfigAttribute` 以前缀 `ROLE_` 开头，它将投票。它将投票授予访问权限是否存在 `GrantedAuthority` 它返回一个 `String` 表示（通过 `getAuthority()` 方法）正好等于一个或多个 `ConfigAttributes` 开始与前缀 `ROLE_`。如果没有 `ConfigAttribute` 与 `ROLE_` 开头的完全匹配，则 `RoleVoter` 将投票拒绝访问。如果没有 `ConfigAttribute` 开始 `ROLE_`，选民将弃权。

AuthenticatedVoter 译：AuthenticatedVoter

另一个我们已经隐含看到的选民是 `AuthenticatedVoter`，它可以用来区分匿名，完全认证和记住我认证的用户。许多网站允许在记住我身份验证的情况下进行某些有限的访问，但需要用户通过登录才能确认其身份以获得完整访问权限。

当我们使用属性 `IS_AUTHENTICATED_ANONYMOUSLY` 授予匿名访问 `IS_AUTHENTICATED_ANONYMOUSLY`，此属性正在由 `AuthenticatedVoter` 处理。有关更多信息，请参阅此类的 Javadoc。

Custom Voters 译：自定义选民

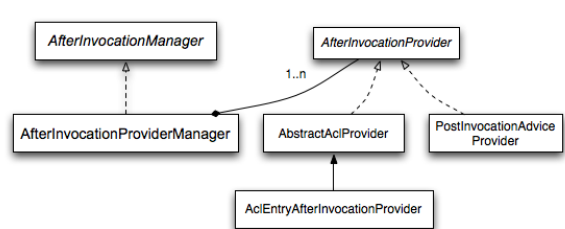
很明显，你也可以实现一个自定义的 `AccessDecisionVoter`，你可以把任何你想要的访问控制逻辑。它可能特定于您的应用程序（与业务逻辑相关），也可能实现一些安全管理逻辑。例如，您将在 Spring 网站上找到一个 [blog article](#)，其中介绍了如何使用投票人实时拒绝其账户被暂停的用户访问。

25.3 After Invocation Handling 译：25.3 调用后处理

在继续进行安全对象调用之前，`AccessDecisionManager` 被 `AbstractSecurityInterceptor` 调用，但某些应用程序需要修改实际由安全对象调用返回的对象的方法。虽然您可以轻松实现您自己的 AOP 关注点来实现此目的，但 Spring Security 提供了一个方便的钩子，它具有几个与 ACL 功能集成的具体实现。

Figure 25.2, “After Invocation Implementation” 展示了 Spring Security 的 `AfterInvocationManager` 及其具体实现。

图25.2. 调用实施后



像Spring Security的许多其他部分一样，`AfterInvocationManager`有一个具体实现，`AfterInvocationProviderManager`，它轮询`AfterInvocationProvider`的列表。允许每个`AfterInvocationProvider`修改返回对象或抛出`AccessDeniedException`。事实上，多个提供者可以修改该对象，因为前一个提供者的结果被传递给列表中的下一个。

请注意，如果您正在使用`AfterInvocationManager`，您仍然需要允许`MethodSecurityInterceptor`的`AccessDecisionManager`允许操作的配置属性。如果您使用的是典型的Spring Security包含的`AccessDecisionManager`实现，那么没有为特定的安全方法调用定义配置属性将导致每个`AccessDecisionVoter`放弃投票。反过来，如果`AccessDecisionManager`属性`allowIfAllAbstainDecisions`为`false`，则将引发`AccessDeniedException`。您可能会避免通过（i）这个潜在的问题上`allowIfAllAbstainDecisions`设置为`true`（虽然这通常是不推荐）或（ii）只是确保至少有一个配置属性，一个`AccessDecisionVoter`将投票授予访问权限。后者（推荐）方法通常通过`ROLE_USER`或`ROLE_AUTHENTICATED`配置属性来实现。

25.4 Hierarchical Roles 译：25.4 角色

这是一个常见的要求，应用程序中的特定角色应自动“包含”其他角色。例如，在具有“管理员”和“用户”角色的应用程序中，您可能希望管理员能够完成普通用户所能做的一切。要做到这一点，你可以确保所有的管理员用户也被分配了“用户”角色。或者，您可以修改每个需要“用户”角色的访问限制，以包含“管理员”角色。如果您的应用程序中有许多不同的角色，这可能会变得非常复杂。

使用角色层次结构允许您配置哪些角色（或权限）应该包括其他角色。的扩展版，春季Security™的`RoleVoter`，`RoleHierarchyVoter`，配置了`RoleHierarchy`，从中获取用户被分配所有的“可达当局”。典型的配置可能如下所示：

```

<bean id="roleVoter" class="org.springframework.security.access.vote.RoleHierarchyVoter">
  <constructor-arg ref="roleHierarchy" />
</bean>
<bean id="roleHierarchy"
  class="org.springframework.security.access.hierarchicalroles.RoleHierarchyImpl">
  <property name="hierarchy">
    <value>
      ROLE_ADMIN > ROLE_STAFF
      ROLE_STAFF > ROLE_USER
      ROLE_USER > ROLE_GUEST
    </value>
  </property>
</bean>
  
```

这里我们有四个角色`ROLE_ADMIN` ⇒ `ROLE_STAFF` ⇒ `ROLE_USER` ⇒ `ROLE_GUEST`。谁进行身份验证的用户`ROLE_ADMIN`，将表现为，如果他们有四个角色时，安全约束都是针对评估`AccessDecisionManager`配置上述`RoleHierarchyVoter`。>符号可以被认为是“包含”的意思。

角色层次结构为简化应用程序的访问控制配置数据和/或减少需要分配给用户的权限数量提供了一种便捷方式。对于更复杂的需求，您可能希望定义应用程序需要的特定访问权限与分配给用户的角色之间的逻辑映射，并在加载用户信息时在两者之间进行转换。

26. Secure Object Implementations 译：26 安全对象实现

26.1 AOP Alliance (MethodInvocation) Security Interceptor 译：26.1 AOP联盟（MethodInvocation）安全拦截器

在Spring Security 2.0之前，保护`MethodInvocation`需要相当多的锅炉板配置。现在推荐的方法安全方法是使用`namespace configuration`。这样方法安全基础结构bean会自动为您配置，因此您不必真正了解实现类。我们只是简单介绍一下这里涉及到的类。

方法安全性使用`MethodSecurityInterceptor`强制执行，从而确保`MethodInvocation`。根据配置方法，拦截器可能特定于单个bean或在多个bean之间共享。拦截器使用`MethodSecurityMetadataSource`实例来获取应用于特定方法调用的配置属性。`MapBasedMethodSecurityMetadataSource`用于存储由方法名称（可以通配符）键入的配置属性，并将在应用上下文中使用`<intercept-methods>`或`<protect-point>`元素定义属性时在内部使用。其他实现将用于处理基于注释的配置。

26.1.1 Explicit MethodSecurityInterceptor Configuration 译：26.1.1 显式MethodSecurityInterceptor配置

您当然可以在您的应用程序上下文中直接配置`MethodSecurityInterceptor`以与Spring AOP的代理机制一起使用：

```

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

26.2 AspectJ (JoinPoint) Security Interceptor 译：26.2 AspectJ（JoinPoint）安全拦截器

AspectJ安全拦截器与上一节讨论的AOP联盟安全拦截器非常相似。事实上，我们只会讨论本节的不同之处。

AspectJ拦截器被命名为`AspectJSecurityInterceptor`。与依赖Spring应用程序上下文通过代理编织安全拦截器的AOP Alliance安全拦截器不同，`AspectJSecurityInterceptor`通过AspectJ编译器编织而成。在同一应用程序中使用两种类型的安全拦截器并不罕见，其中`AspectJSecurityInterceptor`用于域对象实例安全性，AOP Alliance `MethodSecurityInterceptor`用于服务层安全性。

我们首先考虑在Spring应用程序上下文中如何配置 `AspectJSecurityInterceptor`：

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

如您所见，除了班级名称外，`AspectJSecurityInterceptor`与AOP Alliance安全拦截器完全相同。实际上，两个拦截器可以共享相同的`securityMetadataSource`，因为`SecurityMetadataSource`可以与`java.lang.reflect.Method`一起`java.lang.reflect.Method`而不是AOP库特定的类。当然，您的访问决策可以访问相关的AOP库特定的调用（即`MethodInvocation`或`JoinPoint`），因此在进行访问决策时（例如方法参数）可以考虑一系列的添加标准。

接下来，您需要定义一个AspectJ `aspect`。例如：

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

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

public aspect DomainObjectInstanceSecurityAspect implements InitializingBean {

    private AspectJSecurityInterceptor securityInterceptor;

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

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

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

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

    public AspectJSecurityInterceptor getSecurityInterceptor() {
        return securityInterceptor;
    }

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

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

在上面的例子中，安全拦截器将应用于`PersistableEntity`每个实例，该实例是一个未显示的抽象类（您可以使用任何其他类或`pointcut`表达式）。对于那些好奇的人，`AspectJCallback`是必要的，因为`proceed()`；声明仅在`around()`正文中具有特殊含义。当它想要目标对象继续时，`AspectJSecurityInterceptor`将这个类称为匿名`AspectJCallback`类。

您将需要配置Spring以加载该方面并将其与`AspectJSecurityInterceptor`。下面显示了实现这一点的bean声明：

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

就是这样！现在，您可以使用您认为合适的任何方式（例如`new Person();`）在应用程序的任何位置创建您的bean，并且它们将应用安全拦截器。

27. Expression-Based Access Control 译：27.基于表达式的访问控制

Spring Security 3.0引入了使用Spring EL表达式作为授权机制的能力，以及简单使用以前见过的配置属性和访问决策选项。基于表达式的访问控制建立在相同的体系结构上，但允许将复杂的布尔逻辑封装在单个表达式中。

27.1 Overview 译：27.1概述

Spring Security使用Spring EL进行表达式支持，如果您有兴趣更深入地理解该主题，您应该看看它是如何工作的。表达式用“根对象”作为评估上下文的一部分进行评估。Spring Security使用Web和方法安全性的特定类作为根对象，以便提供内置表达式和对当前主体等值的访问。

27.1.1 Common Built-In Expressions 译: 27.1.1 常见的内置表达式

表达式根对象的基类是 `SecurityExpressionRoot` 。这提供了可用于Web和方法安全性的一些常用表达式。

表27.1。 常见的内置表达式

Expression	描述
<code>hasRole([role])</code>	如果当前主体具有指定角色，则返回 <code>true</code> 。默认情况下，如果提供的角色不以“ROLE_”开头，它将被添加。这可以通过修改 <code>defaultRolePrefix</code> 上的 <code>DefaultWebSecurityExpressionHandler</code> 来定制。
<code>hasAnyRole([role1,role2])</code>	如果当前主体具有任何提供的角色（以逗号分隔的字符串列表形式给出），则返回 <code>true</code> 。默认情况下，如果提供的角色不以“ROLE_”开头，它将被添加。这可以通过修改 <code>defaultRolePrefix</code> 上的 <code>DefaultWebSecurityExpressionHandler</code> 来定制。
<code>hasAuthority([authority])</code>	如果当前委托人具有指定的权限，则返回 <code>true</code> 。
<code>hasAnyAuthority([authority1,authority2])</code>	如果当前主体具有任何提供的角色（以逗号分隔的字符串列表的形式给出），则返回 <code>true</code>
<code>principal</code>	允许直接访问表示当前用户的主体对象
<code>authentication</code>	允许直接访问从 <code>SecurityContext</code> 获得的当前 <code>Authentication</code> 对象
<code>permitAll</code>	始终评估为 <code>true</code>
<code>denyAll</code>	始终评估为 <code>false</code>
<code>isAnonymous()</code>	如果当前委托人是匿名用户，则返回 <code>true</code>
<code>isRememberMe()</code>	如果当前主体是记事本用户，则返回 <code>true</code>
<code>isAuthenticated()</code>	如果用户不是匿名的，则返回 <code>true</code>
<code>isFullyAuthenticated()</code>	如果用户不是匿名用户或记住我用户，则返回 <code>true</code>
<code>hasPermission(Object target, Object permission)</code>	如果用户有权访问为给定权限提供的目标，则返回 <code>true</code> 。例如， <code>hasPermission(domainObject, 'read')</code>
<code>hasPermission(Object targetId, String targetType, Object permission)</code>	如果用户有权访问为给定权限提供的目标，则返回 <code>true</code> 。例如， <code>hasPermission(1, 'com.example.domain.Message', 'read')</code>

27.2 Web Security Expressions 译: 27.2 网络安全表达式

要使用表达式来保护单个URL，首先需要将 `<http>` 元素中的 `use-expressions` 属性设置为 `true` 。然后，Spring Security将期望 `<intercept-url>` 元素的 `access` 属性包含Spring EL表达式。表达式应评估为布尔值，定义是否允许访问。例如：

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

这里我们已经定义应用程序的“管理”区域（由URL模式定义）应该只对拥有授权权限“admin”并且其IP地址与本地子网匹配的用户可用。我们已经在上一节看到了内置的 `hasRole` 表达式。表达式 `hasIpAddress` 是特定于Web安全性的附加内置表达式。它由 `WebSecurityExpressionRoot` 类定义，其中的一个实例在评估Web访问表达式时用作表达式根对象。该对象还直接暴露了名称 `request` 下的 `HttpServletRequest` 对象，因此您可以直接在表达式中调用请求。如果正在使用表达式，`WebExpressionVoter` 将被添加到名称空间使用的 `AccessDecisionManager` 。因此，如果您不使用名称空间并想使用表达式，则必须将其中一个添加到您的配置中。

27.2.1 Referring to Beans in Web Security Expressions 译: 27.2.1 在Web安全表达式中引用Beans

如果你想扩展可用的表达式，你可以很容易地引用你公开的任何Spring Bean。例如，假设您的名称为 `webSecurity` 的Bean包含以下方法签名：

```
public class WebSecurity {
    public boolean check(Authentication authentication, HttpServletRequest request) {
        ...
    }
}
```

你可以参考使用的方法：

```
<http>
<intercept-url pattern="/user/**"
access="@webSecurity.check(authentication,request)"/>
...
</http>
```

或者在Java配置中

```
http
    .authorizeRequests()
    .antMatchers("/user/**").access("@webSecurity.check(authentication,request)")
    ...
```

27.2.2 Path Variables in Web Security Expressions 译: 27.2.2 Web安全表达式中的路径变量

有时能够在URL中引用路径变量是很好的。例如，考虑一个REST式应用程序，它以格式 `/user/{userId}` 的URL路径通过id查找用户。

通过将其放入模式中，您可以轻松地引用路径变量。例如，如果您的Bean名称为 `webSecurity`，它包含以下方法签名：

```
public class WebSecurity {
    public boolean checkUserId(Authentication authentication, int id) {
        ...
    }
}
```

您可以参考使用的方法：

```
<http>
<intercept-url pattern="/user/{userId}/**"
    access="@webSecurity.checkUserId(authentication,#userId)"/>
...
</http>
```

或者在Java配置中

```
http
    .authorizeRequests()
    .antMatchers("/user/{userId}/**").access("@webSecurity.checkUserId(authentication,#userId)")
    ...
```

在这两种配置中，匹配的URL将传入路径变量（并将其转换为checkUserId方法）。例如，如果URL是 `/user/123/resource`，那么传入的ID将是 `123`。

27.3 Method Security Expressions 译: 27.3 方法安全表达式

方法安全性比简单的允许或拒绝规则复杂一点。Spring Security 3.0引入了一些新的注释，以便全面支持表达式的使用。

27.3.1 @Pre and @Post Annotations 译: 27.3.1 @Pre和@Post注释

有四个注释支持表达式属性以允许调用前和调用后授权检查，并支持对提交的集合参数或返回值进行过滤。他们是 `@PreAuthorize`，`@PreFilter`，`@PostAuthorize` 和 `@PostFilter`。它们的使用通过 `global-method-security` 命名空间元素启用：

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

Access Control using @PreAuthorize and @PostAuthorize 译: 使用@PreAuthorize和@PostAuthorize进行访问控制

最明显有用的注释是 `@PreAuthorize`，它决定一个方法是否可以被实际调用。例如（来自“联系人”示例应用程序）

```
@PreAuthorize("hasRole('USER')")
public void create(Contact contact);
```

这意味着只有具有角色“ROLE_USER”的用户才能访问。显然，使用传统配置和简单配置属性来实现所需角色可以轻松实现同样的目的。但是关于：

```
@PreAuthorize("hasPermission(#contact, 'admin')")
public void deletePermission(Contact contact, Sid recipient, Permission permission);
```

在这里，我们实际上使用方法参数作为表达式的一部分来决定当前用户是否具有给定联系人的“管理员”权限。内置的 `hasPermission()` 表达式通过应用程序上下文链接到Spring Security ACL模块中，就像我们 [see below](#) 一样。您可以按名称访问任何方法参数作为表达式变量。

Spring Security可以通过多种方式来解决方法参数。Spring Security使用 `DefaultSecurityParameterNameDiscoverer` 来发现参数名称。默认情况下，对于整个方法尝试以下选项。

- 如果Spring Security的 `@P` 注释出现在该方法的单个参数上，则会使用该值。这对于在JDK 8之前使用JDK编译的接口非常有用，它不包含有关参数名称的任何信息。例如：

```
import org.springframework.security.access.method.P;

...

@PreAuthorize("#c.name == authentication.name")
public void doSomething(@P("c") Contact contact);
```

在幕后，使用 `AnnotationParameterNameDiscoverer` 实现了这种使用，可以将其自定义为支持任何指定注释的value属性。

- 如果Spring数据Ae™的 `@Param` 注释存在于该方法的至少一个参数，将被使用的值。这对于在JDK 8之前使用JDK编译的接口非常有用，它不包含有关参数名称的任何信息。例如：

```
import org.springframework.data.repository.query.Param;

...

@PreAuthorize("#n == authentication.name")
Contact findContactByName(@Param("n") String name);
```

在幕后，使用 `AnnotationParameterNameDiscoverer` 实现的这种使用可以被自定义以支持任何指定注释的value属性。

- If JDK 8 was used to compile the source with the `-parameters` argument and Spring 4+ is being used, then the standard JDK reflection API is used to discover the parameter names. This works on both classes and interfaces.
- Last, if the code was compiled with the debug symbols, the parameter names will be discovered using the debug symbols. This will not work for interfaces since they do not have debug information about the parameter names. For interfaces, annotations or the JDK 8 approach must be used.

在表达式中可以使用任何Spring-EL功能，因此您也可以访问参数的属性。例如，如果您希望特定的方法只允许访问其用户名与联系人相匹配的用户，则可以编写

```
@PreAuthorize("#contact.name == authentication.name")
public void doSomething(Contact contact);
```

在这里，我们正在访问另一个内置表达式 `authentication`，它是存储在安全上下文中的 `Authentication`。您还可以使用表达式 `principal` 直接访问其“主体”属性。该值通常是 `UserDetails` 实例，因此您可以使用 `principal.username` 或 `principal.enabled` 等表达式。

通常情况下，您可能希望在调用该方法后执行访问控制检查。这可以使用 `@PostAuthorize` 注释来实现。要从方法访问返回值，`returnObject` 在表达式中使用内置名称 `returnObject`。

Filtering using @PreFilter and @PostFilter 译:使用@PreFilter和@PostFilter进行过滤

正如您可能已经知道的那样，Spring Security支持对集合和数组进行过滤，现在可以使用表达式来实现这一点。这通常是对方法的返回值执行的。例如：

```
@PreAuthorize("hasRole('USER')")
@PostFilter("hasPermission(filterObject, 'read') or hasPermission(filterObject, 'admin')")
public List<Contact> getAll();
```

当使用 `@PostFilter` 注释时，Spring Security遍历返回的集合并删除提供的表达式为false的所有元素。名称 `filterObject` 引用集合中的当前对象。您也可以在方法调用之前使用 `@PreFilter` 进行 `@PreFilter`，但这不是常见的要求。语法是一样的，但如果有多个参数是一个集合类型，则必须使用此注释的 `filterTarget` 属性通过名称选择一个 `filterTarget`。

请注意，过滤显然不能替代您的数据检索查询。如果您要过滤大量集合并删除很多条目，那么这可能效率不高。

27.3.2 Built-In Expressions 译: 27.3.2内置表达式

有一些特定于方法安全的内置表达式，我们已经在上面看到了这些内置表达式。`filterTarget` 和 `returnValue` 值很简单，但使用 `hasPermission()` 表达式需要仔细观察。

The PermissionEvaluator interface 译: PermissionEvaluator接口

`hasPermission()` 表达式被委托给 `PermissionEvaluator` 一个实例。它旨在桥接表达式系统和Spring Security的ACL系统，允许您根据抽象权限指定域对象的授权约束。它对ACL模块没有明确的依赖关系，所以如果需要的话，您可以将它交换出来用于替代实现。界面有两种方法：

```
boolean hasPermission(Authentication authentication, Object targetDomainObject,
    Object permission);

boolean hasPermission(Authentication authentication, Serializable targetId,
    String targetType, Object permission);
```

它直接映射到表达式的可用版本，但不提供第一个参数（`Authentication` 对象）。第一种用于已经加载访问控制的域对象的情况。然后，如果当前用户具有该对象的给定权限，表达式将返回true。第二个版本用于未加载对象但标识符已知的情况。还需要域对象的抽象“类型”说明符，以允许加载正确的ACL权限。传统上，这是对象的Java类，但不一定要与加载权限的方式一致。

要使用 `hasPermission()` 表达式，必须在应用程序上下文中显式配置 `PermissionEvaluator`。这看起来像这样：

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

<bean id="expressionHandler" class=
"org.springframework.security.access.expression.method.DefaultMethodSecurityExpressionHandler">
  <property name="permissionEvaluator" ref="myPermissionEvaluator"/>
</bean>
```

`myPermissionEvaluator` 是实现 `PermissionEvaluator` 的bean。通常这将是来自ACL模块 `AclPermissionEvaluator`。有关更多详细信息，请参阅“Contacts”示例应用程序配置。

Method Security Meta Annotations 译:方法安全元注释

您可以使用元注释来进行方法安全性，以使您的代码更具可读性。如果您发现在整个代码库中重复相同的复杂表达式，这一点尤其方便。例如，请考虑以下几点：

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

我们可以创建一个可用来代替的元注释，而不是随处重复。

```
@Retention(RetentionPolicy.RUNTIME)
@PreAuthorize("#contact.name == authentication.name")
public @interface ContactPermission {}
```

元注释可用于任何Spring Security方法安全注释。为了保持与规范兼容，JSR-250注释不支持元注释。

Part VI. Additional Topics 译:第六部分 - 其他主题

在本部分中，我们将介绍需要了解以前章节的功能以及框架的一些更先进和较少使用的功能。

28. Domain Object Security (ACLs) 译: 28域对象安全性 (ACL)

28.1 Overview 译: 28.1概述

复杂的应用程序通常会发现需要定义访问权限，而不仅仅是在Web请求或方法调用级别。相反，安全决策需要包括谁（`Authentication`），哪里（`MethodInvocation`）和什么（`SomeDomainObject`）。换句话说，授权决策还需要考虑方法调用的实际域对象实例主题。

想象一下，你正在为宠物诊所设计一个应用程序。您的基于Spring的应用程序将有两个主要用户组：宠物诊所的员工以及宠物诊所的客户。员工可以访问所有数据，而客户只能看到他们自己的客户记录。为了让它更有趣，您的客户可以允许其他用户查看他们的客户记录，例如他们的“幼崽幼儿园”导师或当地“小马俱乐部”的总裁。使用Spring Security作为基础，您可以使用以下几种方法：

- Write your business methods to enforce the security. You could consult a collection within the `Customer` domain object instance to determine which users have access. By using the `SecurityContextHolder.getContext().getAuthentication()`, you'll be able to access the `Authentication` object.
- Write an `AccessDecisionVoter` to enforce the security from the `GrantedAuthority[]` s stored in the `Authentication` object. This would mean your `AuthenticationManager` would need to populate the `Authentication` with custom `GrantedAuthority[]` s representing each of the `Customer` domain object instances the principal has access to.
- Write an `AccessDecisionVoter` to enforce the security and open the target `Customer` domain object directly. This would mean your voter needs access to a DAO that allows it to retrieve the `Customer` object. It would then access the `Customer` object's collection of approved users and make the appropriate decision.

这些方法中的每一种都是完全合法的。但是，第一次将您的授权检查与您的业务代码耦合在一起。与此的主要问题包括单元测试的难度提高，事实上它会更难以再利用的`Customer`别处授权逻辑。从`Authentication`对象中获得`GrantedAuthority[]`也很好，但不会扩展到大量的`Customer`。如果用户可能能够访问5000 `Customer`（在这种情况下不太可能，但想象一下，如果它是大型Pony Club的受欢迎兽医！）构建`Authentication`对象所需的内存消耗量和时间将不受欢迎。最后的方法，直接从外部代码打开`Customer`，可能是三者中最好的。它实现了关注点的分离，并且不会滥用内存或CPU周期，但它仍然效率低下，因为`AccessDecisionVoter`和最终的业务方法本身都会对负责检索`Customer`对象的DAO执行调用。每个方法调用两次访问显然是不可取的。另外，对于列出的每种方法，您都需要从头编写自己的访问控制列表（ACL）持久性和业务逻辑。

幸运的是，还有另一种选择，我们将在下面讨论。

28.2 Key Concepts 译：28.2 关键概念

Spring Security的ACL服务在`spring-security-acl-xxx.jar`中发货。您需要将此JAR添加到您的类路径中以使用Spring Security的域对象实例安全功能。

Spring Security的域对象实例安全功能以访问控制列表（ACL）的概念为中心。系统中的每个域对象实例都有自己的ACL，ACL记录了谁可以 and 不能使用该域对象的详细信息。考虑到这一点，Spring Security为您的应用程序提供了三个与ACL相关的主要功能：

- A way of efficiently retrieving ACL entries for all of your domain objects (and modifying those ACLs)
- A way of ensuring a given principal is permitted to work with your objects, before methods are called
- A way of ensuring a given principal is permitted to work with your objects (or something they return), after methods are called

正如第一个要点所指出的，Spring Security ACL模块的主要功能之一是提供了一种检索ACL的高性能方法。此ACL存储库功能非常重要，因为系统中的每个域对象实例都可能有多个访问控制条目，并且每个ACL可能以树状结构从其他ACL继承（这由Spring支持的开箱即用安全性，并且非常常用）。Spring Security的ACL功能经过精心设计，可提供ACL的高性能检索，以及可插拔缓存，死锁 - 最小化数据库更新，独立于ORM框架（我们直接使用JDBC），适当的封装和透明的数据库更新。

鉴于数据库是ACL模块操作的核心，让我们来探索实现中默认使用的四个主表。下表按照典型的Spring Security ACL部署中的大小顺序列出，其中最后一行列出的表格最多：

- `ACL_SID` allows us to uniquely identify any principal or authority in the system ("SID" stands for "security identity"). The only columns are the ID, a textual representation of the SID, and a flag to indicate whether the textual representation refers to a principal name or a `GrantedAuthority`. Thus, there is a single row for each unique principal or `GrantedAuthority`. When used in the context of receiving a permission, a SID is generally called a "recipient".
- `ACL_CLASS` allows us to uniquely identify any domain object class in the system. The only columns are the ID and the Java class name. Thus, there is a single row for each unique Class we wish to store ACL permissions for.
- `ACL_OBJECT_IDENTITY` stores information for each unique domain object instance in the system. Columns include the ID, a foreign key to the `ACL_CLASS` table, a unique identifier so we know which `ACL_CLASS` instance we're providing information for, the parent, a foreign key to the `ACL_SID` table to represent the owner of the domain object instance, and whether we allow ACL entries to inherit from any parent ACL. We have a single row for every domain object instance we're storing ACL permissions for.
- Finally, `ACL_ENTRY` stores the individual permissions assigned to each recipient. Columns include a foreign key to the `ACL_OBJECT_IDENTITY`, the recipient (ie a foreign key to `ACL_SID`), whether we'll be auditing or not, and the integer bit mask that represents the actual permission being granted or denied. We have a single row for every recipient that receives a permission to work with a domain object.

如最后一段所述，ACL系统使用整数位掩码。不用担心，您不需要意识到使用ACL系统时移位的更多细节，但足以说我们有32位可以打开或关闭。这些位中的每一个都表示权限，默认情况下会读取权限（位0），写入（位1），创建（位2），删除（位3）和管理权（位4）。如果您希望使用其他权限，则可以轻松实现自己的`Permission`实例，并且ACL框架的其余部分将在不知道您的扩展的情况下运行。

了解系统中域对象的数量与我们选择使用整数位掩码的事实完全没有关系。虽然您有32位可用于权限，但您可能拥有数十亿个域对象实例（这将意味着`ACL_OBJECT_IDENTITY`中的数十亿行，很可能是`ACL_ENTRY`）。我们提出这一点是因为我们发现，有时候人们错误地认为他们需要一点点为每个潜在的领域对象，情况并非如此。

现在我们已经提供了ACL系统的基本概述，以及它在表格结构中的样子，让我们来探索关键接口。关键接口是：

- `Acl`: Every domain object has one and only one `Acl` object, which internally holds the `AccessControlEntry` s as well as knows the owner of the `Acl`. An Acl does not refer directly to the domain object, but instead to an `ObjectIdentity`. The `Acl` is stored in the `ACL_OBJECT_IDENTITY` table.
- `AccessControlEntry`: An `Acl` holds multiple `AccessControlEntry` s, which are often abbreviated as ACEs in the framework. Each ACE refers to a specific tuple of `Permission`, `Sid` and `Acl`. An ACE can also be granting or non-granting and contain audit settings. The ACE is stored in the `ACL_ENTRY` table.
- `Permission`: A permission represents a particular immutable bit mask, and offers convenience functions for bit masking and outputting information. The basic permissions presented above (bits 0 through 4) are contained in the `BasePermission` class.
- `Sid`: The ACL module needs to refer to principals and `GrantedAuthority[]` s. A level of indirection is provided by the `Sid` interface, which is an abbreviation of "security identity". Common classes include `PrincipalSid` (to represent the principal inside an `Authentication` object) and `GrantedAuthoritySid`. The security identity information is stored in the `ACL_SID` table.
- `ObjectIdentity`: Each domain object is represented internally within the ACL module by an `ObjectIdentity`. The default implementation is called `ObjectIdentityImpl`.
- `AclService`: Retrieves the `Acl` applicable for a given `ObjectIdentity`. In the included implementation (`JdbcAclService`), retrieval operations are delegated to a `LookupStrategy`. The `LookupStrategy` provides a highly optimized strategy for retrieving ACL information, using batched retrievals (`BasicLookupStrategy`) and supporting custom implementations that leverage materialized views, hierarchical queries and similar performance-centric, non-ANSI SQL capabilities.
- `MutableAclService`: Allows a modified `Acl` to be presented for persistence. It is not essential to use this interface if you do not wish.

请注意，我们开箱即用的AclService和相关的数据库类都使用ANSI SQL。因此这应该适用于所有主要数据库。在撰写本文时，系统已经成功通过Hypersonic SQL, PostgreSQL, Microsoft SQL Server和Oracle测试。

Spring Security附带两个样本，演示ACL模块。第一个是联系人示例，另一个是文档管理系统（DMS）示例。我们建议看看这些例子。

28.3 Getting Started 译：28.3 入门

要开始使用Spring Security的ACL功能，您需要在某处存储ACL信息。这需要使用Spring实例化 `DataSource` 。 `DataSource` 然后被注入到 `JdbcMutableAclService` 和 `BasicLookupStrategy` 实例中。后者提供了高性能的ACL检索功能，前者提供了增强功能。有关示例配置，请参阅Spring Security附带的示例之一。您还需要使用最后一节中列出的四个ACL特定表来填充数据库（请参阅相应SQL语句的ACL示例）。

一旦您创建了所需的模式并实例化了 `JdbcMutableAclService` ，您将需要确保您的域模型支持与Spring Security ACL软件包的互操作性。希望 `ObjectIdentityImpl` 将被证明是足够的，因为它提供了许多可以使用它的方法。大多数人将拥有包含 `public Serializable getId()` 方法的域对象。如果返回类型很长，或者与长整型（例如int）兼容，您会发现不需要进一步考虑 `ObjectIdentity` 问题。ACL模块的很多部分都依赖于长标识符。如果你不使用long（或者int，byte等），那么你很可能需要重新实现一些类。我们打算在Spring Security的ACL模块中支持非长的标识符，因为longs已经与所有数据库序列兼容，这是最常见的标识符数据类型，并且具有足够的长度以适应所有常见的使用场景。

以下代码片段显示了如何创建 `Acl` 或修改现有的 `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);
```

在上面的例子中，我们重新检索与标识符为44的“Foo”域对象关联的ACL。然后我们添加一个ACE，使得名为“Samantha”的主体可以“管理”该对象。代码片段相对不言自明，除了insertAce方法外。insertAce方法的第一个参数是确定Acl中的哪个位置将插入新条目。在上面的例子中，我们只是将新ACE放在现有ACE的末尾。最后一个参数是布尔值，表示ACE是否授予或拒绝。大多数时候它会授予（true），但是如果拒绝（false），则权限将被有效阻止。

作为DAO或存储库操作的一部分，Spring Security不提供任何特殊集成来自动创建，更新或删除ACL。相反，你需要为你的单个域对象编写如上所示的代码。值得考虑的是在服务层使用AOP来自动将ACL信息与服务层操作集成在一起。过去我们发现这是一种非常有效的方法。

一旦您使用上述技术将一些ACL信息存储在数据库中，下一步就是将ACL信息实际用作授权决策逻辑的一部分。你在这里有很多选择。您可以编写自己的 `AccessDecisionVoter` 或 `AfterInvocationProvider` ，分别在方法调用之前或之后触发。这些类将使用 `AclService` 来检索相关的ACL，然后调用 `Acl.isGranted(Permission[] permission, Sid[] sids, boolean administrativeMode)` 来决定是否授予或拒绝权限。或者，您可以使用我们的 `AclEntryVoter` ， `AclEntryAfterInvocationProvider` 或者 `AclEntryAfterInvocationCollectionFilteringProvider` 类。所有这些类都提供了一种基于声明的方法在运行时评估ACL信息，从而不需要编写任何代码。请参阅示例应用程序以了解如何使用这些类。

29. Pre-Authentication Scenarios 译：预认证方案

在某些情况下，您希望使用Spring Security进行授权，但在访问应用程序之前，用户已被某个外部系统可靠地认证。我们将这些情况称为“预先认证”情况。示例包括X.509，Siteminder和运行该应用程序的Java EE容器的身份验证。在使用预认证时，Spring Security必须

- Identify the user making the request.
- Obtain the authorities for the user.

细节将取决于外部认证机制。在X.509的情况下，用户可以通过他们的证书信息来标识，或者在Siteminder的情况下通过HTTP请求标头来标识用户。如果依靠容器验证，则通过在传入的HTTP请求上调用 `getUserPrincipal()` 方法来识别用户。在某些情况下，外部机制可能会为用户提供角色/权限信息，但在其他情况下，必须从一个单独的来源获取权限，例如 `UserDetailsService` 。

29.1 Pre-Authentication Framework Classes 译：29.1预认证框架类

由于大多数预认证机制遵循相同的模式，因此Spring Security具有一组类，它们为实现预认证的认证提供程序提供了一个内部框架。这消除了重复，并允许以结构化的方式添加新的实现，而无需从头开始编写所有内容。如果你想使用类似X.509 authentication的东西，你不需要了解这些类，因为它已经有了一个名称空间配置选项，使用起来比较简单。如果您需要使用显式的bean配置或正在计划编写自己的实现，那么了解提供的实现如何工作将会很有用。您将在 `org.springframework.security.web.authentication.preauth` 下找到课程。我们只是在这里提供一个大纲，所以你应该在适当的地方咨询Javadoc和源代码。

29.1.1 AbstractPreAuthenticatedProcessingFilter 译：29.1.1 AbstractPreAuthenticatedProcessingFilter

该类将检查安全上下文的当前内容，如果为空，它将尝试从HTTP请求中提取用户信息并将其提交给 `AuthenticationManager` 。子类覆盖以下方法来获取此信息：

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

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

调用这些后，过滤器将创建一个包含返回数据的 `PreAuthenticatedAuthenticationToken` 并提交它进行验证。通过这里的“身份验证”，我们实际上只是意味着可能会加载用户权限的进一步处理，但遵循标准的Spring Security身份验证体系结构。

像其他的Spring Security认证过滤器，预认证过滤器具有 `authenticationDetailsSource` 属性，默认会创建一个 `WebAuthenticationDetails` 对象来存储更多的信息，如在会话标识符和原始IP地址 `details` 的财产 `Authentication` 对象。在可以从预认证机制获得用户角色信息的情况下，数据也存储在该属性中，并且具体实现 `GrantedAuthoritiesContainer` 接口。这使验证提供者能够读取外部分配给用户的权限。接下来我们将看一个具体的例子。

J2eeBasedPreAuthenticatedWebAuthenticationDetailsSource 译：J2eeBasedPreAuthenticatedWebAuthenticationDetailsSource

如果筛选器配置了 `authenticationDetailsSource` 这是此类的一个实例，则通过为每个预定义的“可映射角色”调用 `isUserInRole(String role)` 方法来获取权限信息。该类从配置的 `MappableAttributesRetriever` 获取这些。可能的实现包括在应用程序上下文中对列表进行硬编码，并从 `web.xml` 文件中的 `<security-role>` 信息中读取角色信息。预认证示例应用程序使用后一种方法。

还有一个额外的阶段，使用配置的 `Attributes2GrantedAuthoritiesMapper` 将角色（或属性）映射到Spring Security `GrantedAuthority` 对象。默认情况下，只需在名称中添加通常的 `ROLE_` 前缀，但它可以让您完全控制行为。

29.1.2 PreAuthenticatedAuthenticationProvider 译: 29.1.2 PreAuthenticatedAuthenticationProvider

预先认证的提供者要比为用户加载 `UserDetails` 对象做更多的事情。它通过委托给 `AuthenticationUserDetailsService` 来做到这 `AuthenticationUserDetailsService`。后者与标准 `UserDetailsService` 类似，但需要 `Authentication` 对象而不仅仅是用户名：

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

这个接口也可能有其他用途，但是通过预认证，它允许访问打包在 `Authentication` 对象中的 `Authentication`，就像我们在上一节中看到的那样。
`PreAuthenticatedGrantedAuthoritiesUserDetailsService` 课是这样做的。或者，它可以通过 `UserDetailsServiceWrapper` 实现委托给标准 `UserDetailsService`。

29.1.3 Http403ForbiddenEntryPoint 译: 29.1.3 Http403ForbiddenEntryPoint

`AuthenticationEntryPoint` 在 [technical overview](#) 一章中讨论过。通常，它负责启动未经身份验证的用户（当他们尝试访问受保护资源时）的身份验证过程，但在预先验证的情况下，这不适用。如果您没有将预认证与其他认证机制结合使用，则只能使用 `ExceptionTranslationFilter` 的实例配置 `ExceptionTranslationFilter`。如果用户被 `AbstractPreAuthenticatedProcessingFilter` 拒绝，将导致空身份验证，它将被调用。如果被调用，它总是返回一个 `403` 禁止的响应代码。

29.2 Concrete Implementations 译: 29.2 具体实现

X.509 认证涵盖在其 [own chapter](#) 中。在这里，我们将看一些为其他预认证方案提供支持的类。

29.2.1 Request-Header Authentication (Siteminder) 译: 29.2.1 请求头认证 (Siteminder)

外部认证系统可以通过在 HTTP 请求上设置特定标头来向应用程序提供信息。一个众所周知的例子是 Siteminder，它在名为 `SM_USER` 的标头中传递用户名。该机制由类 `RequestHeaderAuthenticationFilter` 支持，该类简单地 `RequestHeaderAuthenticationFilter` 中提取用户名。它默认使用名称 `SM_USER` 作为标题名称。查看 Javadoc 了解更多详情。



请注意，使用这样的系统时，框架根本不执行身份验证检查，因此外部系统正确配置并保护对应用程序的所有访问 *非常重要*。如果攻击者能够在未检测到原始请求的情况下伪造标头，那么他们可能会选择他们希望的任何用户名。

Siteminder Example Configuration 译: Siteminder 示例配置

使用此过滤器的典型配置如下所示：

```
<security:http>  
<!-- Additional http configuration omitted -->  
<security:custom-filter position="PRE_AUTH_FILTER" ref="siteminderFilter" />  
</security:http>  
  
<bean id="siteminderFilter" class="org.springframework.security.web.authentication.preauth.RequestHeaderAuthenticationFilter">  
  <property name="principalRequestHeader" value="SM_USER"/>  
  <property name="authenticationManager" ref="authenticationManager" />  
</bean>  
  
<bean id="preauthAuthProvider" class="org.springframework.security.web.authentication.preauth.PreAuthenticatedAuthenticationProvider">  
  <property name="preAuthenticatedUserDetailsService">  
    <bean id="userDetailsServiceWrapper"  
      class="org.springframework.security.core.userdetails.UserDetailsServiceWrapper">  
      <property name="userDetailsService" ref="userDetailsService"/>  
    </bean>  
  </property>  
</bean>  
  
<security:authentication-manager alias="authenticationManager">  
<security:authentication-provider ref="preauthAuthProvider" />  
</security:authentication-manager>
```

我们在这里假定 `security namespace` 正在用于配置。它还假设你已经为你的配置添加了一个 `UserDetailsService`（称为 `userDetailsService`）来加载用户的角色。

29.2.2 Java EE Container Authentication 译: 29.2.2 Java EE 容器认证

该类 `J2eePreAuthenticatedProcessingFilter` 将从提取用户名 `userPrincipal` 的财产 `HttpServletRequest`。这个过滤器的使用通常会与上面 [the section called "J2eeBasedPreAuthenticatedWebAuthenticationDetailsSource"](#) 中描述的 Java EE 角色的使用相结合。

在使用这种方法的代码库中有一个示例应用程序，因此如果您有兴趣，可以从 github 获取代码并查看应用程序上下文文件。该代码位于 `samples/xml/preauth` 目录中。

30. LDAP Authentication 译: 30. LDAP 身份验证

30.1 Overview 译: 30.1 概述

LDAP 通常被组织用作用户信息和身份验证服务的中央存储库。它也可以用来存储应用程序用户的角色信息。

LDAP 服务器如何配置有很多不同的场景，所以 Spring Security 的 LDAP 提供程序是完全可配置的。它使用单独的策略接口进行身份验证和角色检索，并提供可配置为处理各种情况的默认实现。

在尝试将其用于 Spring Security 之前，您应该熟悉 LDAP。以下链接提供了有关所涉概念的良好介绍，并提供了使用免费 LDAP 服务器 OpenLDAP 设置目录的指南：
<http://www.zytrax.com/books/ldap/>。熟悉用于从 Java 访问 LDAP 的 JNDI API 也可能有用。我们不使用 LDAP 提供程序中的任何第三方 LDAP 库（Mozilla, JLDAP 等），但广泛使用 Spring LDAP，因此如果您计划添加自己的自定义项，熟悉该项目可能会很有用。

使用 LDAP 身份验证时，确保正确配置 LDAP 连接池非常重要。如果你不熟悉如何做到这一点，你可以参考 [Java LDAP documentation](#)。

30.2 Using LDAP with Spring Security

译: 30.2在 Spring Security中使用 LDAP

Spring Security中的LDAP认证大致可以分为以下几个阶段。

- Obtaining the unique LDAP "Distinguished Name", or DN, from the login name. This will often mean performing a search in the directory, unless the exact mapping of usernames to DN's is known in advance. So a user might enter the name "joe" when logging in, but the actual name used to authenticate to LDAP will be the full DN, such as `uid=joe,ou=users,dc=spring,dc=io`.
- Authenticating the user, either by "binding" as that user or by performing a remote "compare" operation of the user's password against the password attribute in the directory entry for the DN.
- Loading the list of authorities for the user.

例外情况是LDAP目录仅用于在本地检索用户信息并进行身份验证。这可能是不可能的，因为目录通常设置为对用户密码等属性的读访问权限有限。

我们将在下面看一些配置方案。有关可用配置选项的完整信息，请参阅安全名称空间模式（您的XML编辑器中应该提供哪些信息）。

30.3 Configuring an LDAP Server

译: 30.3配置LDAP服务器

您需要做的第一件事是配置服务器进行身份验证。这是使用来自安全名称空间的`<ldap-server>`元素完成的。可以使用`url`属性将其配置为指向外部LDAP服务器：

```
<ldap-server url="ldap://springframework.org:389/dc=springframework,dc=org" />
```

30.3.1 Using an Embedded Test Server

译: 30.3.1使用嵌入式测试服务器

`<ldap-server>`元素也可用于创建嵌入式服务器，这对于测试和演示可能非常有用。在这种情况下，您不使用`url`属性就可以使用它：

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

这里我们已经指定目录的根目录DIT应该是“dc = springframework, dc = org”，这是默认值。使用这种方式，命名空间解析器将创建一个嵌入式Apache Directory服务器，并扫描类路径以查找将尝试加载到服务器的任何LDIF文件。您可以使用`ldif`属性自定义此行为，该属性定义要加载的LDIF资源：

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

这使得启动和运行LDAP变得容易很多，因为使用外部服务器可能会很不方便。它还将用户从连接Apache Directory服务器所需的复杂bean配置中隔离开来。使用普通的Spring Beans，配置会更加混乱。您必须拥有必要的Apache目录依赖关系jar供您应用程序使用。这些可以从LDAP示例应用程序获得。

30.3.2 Using Bind Authentication

译: 30.3.2使用绑定验证

这是最常见的LDAP身份验证方案。

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

这个简单的例子将通过用所提供的模式中的用户登录名替换用户并获得用户的DN，并尝试将该用户与登录密码绑定。如果您的所有用户都存储在目录中的单个节点下，这是可以的。如果您想要配置LDAP搜索过滤器来查找用户，则可以使用以下内容：

```
<ldap-authentication-provider user-search-filter="(uid={0})"
user-search-base="ou=people"/>
```

如果与上述服务器定义`ou=people,dc=springframework,dc=org`使用，`ou=people,dc=springframework,dc=org`使用`user-search-filter`属性的值作为过滤器在DN `ou=people,dc=springframework,dc=org`下执行搜索。同样，用户登录名将替换过滤器名称中的参数，因此它将搜索`uid`属性等于用户名的条目。如果`user-search-base` user-search-base，则将从根进行搜索。

30.3.3 Loading Authorities

译: 30.3.3加载权限

如何从LDAP目录中的组加载权限是由以下属性控制的。

- `group-search-base`. Defines the part of the directory tree under which group searches should be performed.
- `group-role-attribute`. The attribute which contains the name of the authority defined by the group entry. Defaults to `cn`.
- `group-search-filter`. The filter which is used to search for group membership. The default is `uniqueMember={0}`, corresponding to the `groupOfUniqueNames` LDAP class^[19]. In this case, the substituted parameter is the full distinguished name of the user. The parameter `{1}` can be used if you want to filter on the login name.

所以如果我们使用下面的配置

```
<ldap-authentication-provider user-dn-pattern="uid={0},ou=people"
group-search-base="ou=groups" />
```

并以用户“ben”身份成功认证，则随后加载权限将在目录条目`ou=groups,dc=springframework,dc=org`下执行搜索，查找包含值`uid=ben,ou=people,dc=springframework,dc=org`的属性`uniqueMember uid=ben,ou=people,dc=springframework,dc=org`。默认情况下，机构名称的前缀为`ROLE_`。您可以使用`role-prefix`属性更改此`role-prefix`。如果您不需要任何前缀，请使用`role-prefix="none"`。有关加载权限的更多信息，请参阅`DefaultLdapAuthoritiesPopulator`类的Javadoc。

30.4 Implementation Classes

译: 30.4实现类

我们上面使用的命名空间配置选项使用简单，比使用Spring bean明确得多。有些情况下，您可能需要知道如何直接在应用程序上下文中配置Spring Security LDAP。例如，您可能希望自定义某些类的行为。如果您使用命名空间配置很高兴，那么您可以跳过本节和下一节。

主要的LDAP提供者类`LdapAuthenticationProvider`实际上并没有做太多工作，而是将工作委托给其他两个bean，即`LdapAuthenticator`和`LdapAuthoritiesPopulator`，它们分别负责验证用户和检索用户的`GrantedAuthority`。

30.4.1 LdapAuthenticator Implementations

译: 30.4.1 LdapAuthenticator实现

认证者还负责检索任何所需的用户属性。这是因为对属性的权限可能取决于正在使用的身份验证的类型。例如，如果作为用户进行绑定，则可能需要用用户自己的权限读取它们。

目前Spring Security提供了两种身份验证策略：

- Authentication directly to the LDAP server ("bind" authentication).
- Password comparison, where the password supplied by the user is compared with the one stored in the repository. This can either be done by retrieving the value of the password attribute and checking it locally or by performing an LDAP "compare" operation, where the supplied password is passed to the server for comparison and the real password value is never retrieved.

Common Functionality 译：通用功能

在可以通过任一策略对用户进行身份验证之前，必须从提供给应用程序的登录名中获取专有名称（DN）。这可以通过简单的模式匹配（通过设置 `setUserDnPatterns` 数组属性）或通过设置 `userSearch` 属性来完成。对于DN模式匹配方法，使用标准的Java模式格式，并且登录名将替代参数 `{0}`。该模式应该与配置的 `SpringSecurityContextSource` 将绑定到的DN有关（有关此信息的更多信息，请参阅 [connecting to the LDAP server](#) 上的部分）。例如，如果您正在使用URL为 `ldap://monkeymachine.co.uk/dc=springframework,dc=org` 的LDAP服务器，并且模式为 `uid={0},ou=greatapes`，则“gorilla”的登录名将映射到DN `uid=gorilla,ou=greatapes,dc=springframework,dc=org`。每个配置的DN模式将依次尝试，直到找到匹配项。有关使用搜索的信息，请参阅下面的 [search objects](#) 部分。也可以使用两种方法的组合 - 首先检查模式，如果找不到匹配的DN，则将使用搜索。

BindAuthenticator 译：BindAuthenticator

包 `org.springframework.security.ldap.authentication` 的类 `BindAuthenticator` 实现了绑定认证策略。它只是试图绑定为用户。

PasswordComparisonAuthenticator 译：PasswordComparisonAuthenticator

类 `PasswordComparisonAuthenticator` 实施密码比较认证策略。

30.4.2 Connecting to the LDAP Server 译：30.4.2连接到LDAP服务器

上面讨论的bean必须能够连接到服务器。它们都必须提供 `SpringSecurityContextSource`，这是Spring LDAP的扩展 `ContextSource`。除非有特殊要求，否则通常会配置一个 `DefaultSpringSecurityContextSource` bean，该bean可以使用LDAP服务器的URL进行配置，并可以使用“manager”用户的用户名和密码进行配置，在绑定到服务器时将默认使用该用户名和密码而不是匿名绑定）。有关更多信息，请阅读此类的Javadoc和Spring LDAP的 `AbstractContextSource`。

30.4.3 LDAP Search Objects 译：30.4.3 LDAP搜索对象

通常比简单的DN匹配更复杂的策略需要在目录中定位用户条目。这可以被封装在一个 `LdapUserSearch` 实例中，该实例可以提供给验证器实现，例如，允许他们找到一个用户。提供的实现是 `FilterBasedLdapUserSearch`。

FilterBasedLdapUserSearch 译：FilterBasedLdapUserSearch

这个bean使用LDAP过滤器来匹配目录中的用户对象。该过程在Javadoc中对 `JDK DirContext class` 上的相应搜索方法进行了说明。如上所述，搜索过滤器可以提供参数。对于这个类，唯一有效的参数是 `{0}`，它将被用户的登录名替换。

30.4.4 LdapAuthoritiesPopulator 译：30.4.4 LdapAuthoritiesPopulator

在成功验证用户身份后，`LdapAuthenticationProvider` 将尝试通过调用配置的 `LdapAuthoritiesPopulator` bean为用户加载一组权限。`DefaultLdapAuthoritiesPopulator` 是一个实现，它将通过在目录中搜索用户所属的组来加载权限（通常这些将是目录中的 `groupOfNames` 或 `groupOfUniqueNames` 条目）。有关它如何工作的更多细节，请咨询本课程的Javadoc。

如果您只想使用LDAP进行身份验证，但是从不同的来源（例如数据库）加载权限，则可以提供您自己的此接口实现，然后插入。

30.4.5 Spring Bean Configuration 译：30.4.5 Spring Bean配置

一个典型的配置，使用我们在这里讨论的一些bean，可能看起来像这样：

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

这将设置提供者访问URL为 `ldap://monkeymachine:389/dc=springframework,dc=org` 的LDAP服务器。通过尝试绑定DN `uid=<user-login-name>,ou=people,dc=springframework,dc=org` 来执行认证。成功验证后，角色将通过在默认过滤器 `(member=<user's-DN>)` 下的DN `ou=groups,dc=springframework,dc=org` 下搜索来分配给用户。角色名称将取自每场比赛的“ou”属性。

要配置一个用户搜索对象，它使用过滤器 `(uid=<user-login-name>)` 代替DN模式（或除此之外），您可以配置以下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>
```

并通过设置 `BindAuthenticator` bean的 `userSearch` 属性来使用它。然后认证者会在尝试以该用户身份进行绑定之前调用搜索对象以获取正确的用户DN。

30.4.6 LDAP Attributes and Customized UserDetails 译：30.4.6 LDAP属性和自定义 UserDetails

使用 `LdapAuthenticationProvider` 进行身份验证的最终结果与使用标准 `UserDetailsService` 接口的普通Spring Security身份验证相同。一个 `UserDetails` 对象被创建并存储在返回的 `Authentication` 对象中。与使用 `UserDetailsService`，常见的要求是能够自定义此实现并添加额外的属性。使用LDAP时，这些通常是来自用户条目的属性。`UserDetails` 对象的创建由提供者的 `UserDetailsContextMapper` 策略控制，该策略负责映射用户对象与LDAP上下文数据的映射关系：

```
public interface UserDetailsContextMapper {

    UserDetails mapUserFromContext(DirContextOperations ctx, String username,
        Collection<GrantedAuthority> authorities);

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

只有第一种方法与认证有关。如果您提供了此接口的实现并将其注入到 `LdapAuthenticationProvider`，则您可以精确控制如何创建 `UserDetails` 对象。第一个参数是Spring LDAP的实例 `DirContextOperations`，它允许您访问在认证过程中加载的LDAP属性。`username` 参数是用于认证的名称，最后一个参数是配置的 `LdapAuthoritiesPopulator` 为用户加载的权限的集合。

根据您的身份验证类型，上下文数据加载的方式略有不同。使用 `BindAuthenticator`，绑定操作返回的上下文将用于读取属性，否则将使用从配置的 `ContextSource` 获取的标准上下文读取数据（当搜索配置为定位用户时，这将是数据由搜索对象返回）。

30.5 Active Directory Authentication 译：30.5 Active Directory身份验证

Active Directory支持它自己的非标准认证选项，并且正常的使用模式与标准 `LdapAuthenticationProvider`。通常，使用域用户名（以 `` 的形式）执行身份验证，而不是使用LDAP专有名称。为了简化这个过程，Spring Security 3.1有一个为典型Active Directory设置定制的身份验证提供程序。

30.5.1 ActiveDirectoryLdapAuthenticationProvider 译：30.5.1 ActiveDirectoryLdapAuthenticationProvider

配置 `ActiveDirectoryLdapAuthenticationProvider` 非常简单。您只需提供域名和一个提供服务器地址的LDAP URL ^[20]。一个示例配置将如下所示：

```
<bean id="adAuthenticationProvider"
      class="org.springframework.security.ldap.authentication.ad.ActiveDirectoryLdapAuthenticationProvider">
  <constructor-arg value="mydomain.com" />
  <constructor-arg value="ldap://adserver.mydomain.com/" />
</bean>
}
```

请注意，不需要指定单独的 `ContextSource` 以定义服务器位置 - 该bean是完全独立的。例如，名为“Sharon”的用户可以通过输入用户名 `sharon` 或完整的Active Directory `userPrincipalName`（即 `` 进行身份验证。然后定位用户的目录条目，并返回可用于定制创建的 `UserDetails` 对象（`UserDetailsContextMapper` 可以为此注入 `UserDetails` 属性。与目录的所有交互都与用户本身的身份一致。没有“经理”用户的概念。

默认情况下，用户权限是从用户条目的 `memberOf` 属性值中获取的。分配给用户的权限可以再次使用 `UserDetailsContextMapper` 进行定制。您还可以将 `GrantedAuthoritiesMapper` 注入提供程序实例，以控制最终位于 `Authentication` 对象中的 `Authentication`。

Active Directory Error Codes 译：Active Directory错误代码

默认情况下，失败的结果将导致标准的Spring Security `BadCredentialsException`。如果将属性 `convertSubErrorCodesToExceptions` 设置为 `true`，则会解析异常消息以尝试提取特定于Active Directory的错误代码并引发更具体的异常。查看Javadoc课程获取更多信息。

^[19] 请注意，这与使用 `member={0}` 的基础 `DefaultLdapAuthoritiesPopulator` 的默认配置不同。

^[20] 也可以使用DNS查找来获取服务器的IP地址。目前尚不支持，但希望将在未来的版本。

31. OAuth 2.0 Login — Advanced Configuration 译：31. OAuth 2.0 Login高级配置

`HttpSecurity.oauth2Login()` 为定制OAuth 2.0登录提供了许多配置选项。主要配置选项分组到他们的协议端点对应部分。

例如，`oauth2Login().authorizationEndpoint()` 允许配置 *授权端点*，而 `oauth2Login().tokenEndpoint()` 允许配置 *令牌端点*。

以下代码显示了一个示例：

```
@EnableWebSecurity
public class OAuth2LoginSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .oauth2Login()
            .authorizationEndpoint()
            ...
            .redirectionEndpoint()
            ...
            .tokenEndpoint()
            ...
            .userInfoEndpoint()
            ...
    }
}
```

`oauth2Login()` DSL的主要目标是与规范中定义的命名保持一致。

OAuth 2.0授权框架定义了 [Protocol Endpoints](#)如下：

授权过程使用两个授权服务器端点（HTTP资源）：

- Authorization Endpoint: Used by the client to obtain authorization from the resource owner via user-agent redirection.
- Token Endpoint: Used by the client to exchange an authorization grant for an access token, typically with client authentication.

以及一个客户端端点：

- Redirection Endpoint: Used by the authorization server to return responses containing authorization credentials to the client via the resource owner user-agent.

OpenID Connect Core 1.0规范定义了 [UserInfo Endpoint](#)，如下所示：

UserInfo端点是一个OAuth 2.0保护资源，用于返回有关经过身份验证的最终用户的声明。为了获得有关最终用户的请求声明，客户端使用通过OpenID Connect Authentication获取的访问令牌向UserInfo端点发出请求。这些声明通常由包含声明的名称/值对集合的JSON对象表示。

以下代码显示了可用于 `oauth2Login()` DSL的完整配置选项：

```
@EnableWebSecurity
public class OAuth2LoginSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .oauth2Login()
            .clientRegistrationRepository(this.clientRegistrationRepository())
            .authorizedClientService(this.authorizedClientService())
            .loginPage("/login")
            .authorizationEndpoint()
            .baseUrl(this.authorizationRequestBaseUrl())
            .authorizationRequestRepository(this.authorizationRequestRepository())
            .and()
            .redirectionEndpoint()
            .baseUrl(this.authorizationResponseBaseUrl())
            .and()
            .tokenEndpoint()
            .accessTokenResponseClient(this.accessTokenResponseClient())
            .and()
            .userInfoEndpoint()
            .userAuthoritiesMapper(this.userAuthoritiesMapper())
            .userService(this.oauth2UserService())
            .oidcUserService(this.oidcUserService())
            .customUserType(GitHubOAuth2User.class, "github");
    }
}
```

下面的章节会详细介绍每个可用的配置选项：

- [Section 31.1, "OAuth 2.0 Login Page"](#)
- [Section 31.2, "Authorization Endpoint"](#)
- [Section 31.3, "Redirection Endpoint"](#)
- [Section 31.4, "Token Endpoint"](#)
- [Section 31.5, "UserInfo Endpoint"](#)

31.1 OAuth 2.0 Login Page 译：31.1 OAuth 2.0登录页面

默认情况下，OAuth 2.0登录页面由 `DefaultLoginPageGeneratingFilter` 自动生成。默认登录页面显示每个配置的OAuth客户端，其 `ClientRegistration.clientName` 作为链接，可以启动授权请求（或OAuth 2.0登录）。

每个OAuth客户端的链接目标都默认为以下内容：

```
OAuth2AuthorizationRequestRedirectFilter.DEFAULT_AUTHORIZATION_REQUEST_BASE_URI + "/" + {registrationId}
```

以下行显示一个示例：

```
<a href="/oauth2/authorization/google">Google</a>
```

要覆盖默认登录页面，请配置 `oauth2Login().loginPage()` 和（可选） `oauth2Login().authorizationEndpoint().baseUrl()`。

以下列表显示了一个示例：

```
@EnableWebSecurity
public class OAuth2LoginSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .oauth2Login()
            .loginPage("/login/oauth2")
            ...
            .authorizationEndpoint()
            .baseUri("/login/oauth2/authorization")
            ....
    }
}
```



Important

您需要提供 `@Controller`，其中 `@RequestMapping("/login/oauth2")` 能够呈现自定义登录页面。



如前所述，配置 `oauth2Login().authorizationEndpoint().baseUri()` 是可选的。但是，如果您选择对其进行自定义，请确保与每个OAuth客户端的链接匹配 `authorizationEndpoint().baseUri()`。

以下行显示一个示例：

```
<a href="/login/oauth2/authorization/google">Google</a>
```

31.2 Authorization Endpoint 译：31.2授权端点

31.2.1 `AuthorizationRequestRepository` 译：31.2.1 `AuthorizationRequestRepository`

`AuthorizationRequestRepository` 负责从发起授权请求到收到授权响应（回调）时的持续 `OAuth2AuthorizationRequest`。



`OAuth2AuthorizationRequest` 用于关联和验证授权响应。

的默认实现 `AuthorizationRequestRepository` 为 `HttpSessionOAuth2AuthorizationRequestRepository`，它存储 `OAuth2AuthorizationRequest` 在 `HttpSession`。

如果您想提供的自定义实现 `AuthorizationRequestRepository` 存储的属性 `OAuth2AuthorizationRequest` 在 `Cookie`，配置它显示在下面的例子：

```
@EnableWebSecurity
public class OAuth2LoginSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .oauth2Login()
            .authorizationEndpoint()
            .authorizationRequestRepository(this.cookieAuthorizationRequestRepository())
            ...
    }

    private AuthorizationRequestRepository<OAuth2AuthorizationRequest> cookieAuthorizationRequestRepository() {
        return new HttpCookieOAuth2AuthorizationRequestRepository();
    }
}
```

31.3 Redirection Endpoint 译：31.3重定向端点

授权服务器使用重定向端点通过资源所有者用户代理将授权响应（包含授权凭证）返回给客户端。



OAuth 2.0登录利用授权代码授权。因此，授权凭证是授权代码。

默认的授权响应 `baseUri`（重定向端点）是 `/login/oauth2/code/*`，它在 `OAuth2LoginAuthenticationFilter.DEFAULT_FILTER_PROCESSES_URI` 定义。

如果您想自定义授权响应 `baseUri`，请按照以下示例中所示对其进行配置：

```
@EnableWebSecurity
public class OAuth2LoginSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .oauth2Login()
            .redirectionEndpoint()
            .baseUri("/login/oauth2/callback/*")
            ....
    }
}
```



Important

您还需要确保 `ClientRegistration.redirectUriTemplate` 与自定义授权响应 `baseUri` 相匹配。
以下列表显示了一个示例：

```
return CommonOAuth2Provider.GOOGLE.getBuilder("google")
    .clientId("google-client-id")
    .clientSecret("google-client-secret")
    .redirectUriTemplate("{baseUrl}/login/oauth2/callback/{registrationId}")
    .build();
```

31.4 Token Endpoint 译：31.4令牌端点

31.4.1 OAuth2AccessTokenResponseClient 译：31.4.1 OAuth2AccessTokenResponseClient

`OAuth2AccessTokenResponseClient` 负责在授权服务器的令牌端点上交换访问令牌凭证的授权许可证书。

`OAuth2AccessTokenResponseClient` 的默认实现是 `NimbusAuthorizationCodeTokenResponseClient`，它在令牌端点交换访问令牌的授权代码。



`NimbusAuthorizationCodeTokenResponseClient` 在内部使用 `Nimbus OAuth 2.0 SDK`。

如果您想提供 `OAuth2AccessTokenResponseClient` 的自定义实现，该实现使用Spring Framework 5 reactive `WebClient` 来启动对令牌端点的请求，请按照以下示例中所示进行配置：

```
@EnableWebSecurity
public class OAuth2LoginSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .oauth2Login()
            .tokenEndpoint()
            .accessTokenResponseClient(this.accessTokenResponseClient())
            ...
    }

    private OAuth2AccessTokenResponseClient<OAuth2AuthorizationCodeGrantRequest> accessTokenResponseClient() {
        return new SpringWebClientAuthorizationCodeTokenResponseClient();
    }
}
```

31.5 UserInfo Endpoint 译：31.5 UserInfo端点

UserInfo端点包含许多配置选项，如以下小节所述：

- [Section 31.5.1, "Mapping User Authorities"](#)
- [Section 31.5.2, "Configuring a Custom OAuth2User"](#)
- [Section 31.5.3, "OAuth 2.0 UserService"](#)
- [Section 31.5.4, "OpenID Connect 1.0 UserService"](#)

31.5.1 Mapping User Authorities 译：31.5映射用户权限

用户成功通过OAuth 2.0 Provider进行身份验证后，`OAuth2User.getAuthorities()`（或 `OidcUser.getAuthorities()`）可能会映射到一组新的 `GrantedAuthority` 实例，这些实例将在完成身份验证时提供给 `OAuth2AuthenticationToken`。



`OAuth2AuthenticationToken.getAuthorities()` 用于授权请求，如 `hasRole('USER')` 或 `hasRole('ADMIN')`。

映射用户权限时有几个选项可供选择：

- Using a `GrantedAuthoritiesMapper`
- [Delegation-based strategy with OAuth2UserService](#)

Using a `GrantedAuthoritiesMapper` 译：使用 GrantedAuthoritiesMapper

提供 `GrantedAuthoritiesMapper` 的实现 `GrantedAuthoritiesMapper` 以下示例中所示进行配置：


```

@EnableWebSecurity
public class OAuth2LoginSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .oauth2Login()
            .userInfoEndpoint()
            .userAuthoritiesMapper(this.userAuthoritiesMapper())
            ...
    }

    private GrantedAuthoritiesMapper userAuthoritiesMapper() {
        return (authorities) -> {
            Set<GrantedAuthority> mappedAuthorities = new HashSet<>();

            authorities.forEach(authority -> {
                if (OidcUserAuthority.class.isInstance(authority)) {
                    OidcUserAuthority oidcUserAuthority = (OidcUserAuthority)authority;

                    OidcIdToken idToken = oidcUserAuthority.getIdToken();
                    OidcUserInfo userInfo = oidcUserAuthority.getUserInfo();

                    // Map the claims found in idToken and/or userInfo
                    // to one or more GrantedAuthority's and add it to mappedAuthorities

                } else if (OAuth2UserAuthority.class.isInstance(authority)) {
                    OAuth2UserAuthority oauth2UserAuthority = (OAuth2UserAuthority)authority;

                    Map<String, Object> userAttributes = oauth2UserAuthority.getAttributes();

                    // Map the attributes found in userAttributes
                    // to one or more GrantedAuthority's and add it to mappedAuthorities

                }
            });

            return mappedAuthorities;
        };
    }
}

```

或者，您可以注册 `GrantedAuthoritiesMapper` `@Bean` 以使其自动应用于配置，如下示例所示：

```

@EnableWebSecurity
public class OAuth2LoginSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.oauth2Login();
    }

    @Bean
    public GrantedAuthoritiesMapper userAuthoritiesMapper() {
        ...
    }
}

```

Delegation-based strategy with `OAuth2UserService` 译:基于代表团的策略与 `OAuth2UserService`

与使用 `GrantedAuthoritiesMapper` 相比，此策略更先进，但它也更灵活，因为它可让您访问 `OAuth2UserRequest` 和 `OAuth2User`（使用 OAuth 2.0 UserService 时）或 `OidcUserRequest` 和 `OidcUser`（使用 OpenID Connect 1.0 UserService 时）。

`OAuth2UserRequest`（和 `OidcUserRequest`）为您提供对相关联的 `OAuth2AccessToken` 访问权限，这对于 *委托人* 需要从受保护资源获取授权信息，然后才能映射用户的自定义权限的情况非常有用。

以下示例显示如何使用 OpenID Connect 1.0 UserService 实施和配置基于委派策略：

```

@EnableWebSecurity
public class OAuth2LoginSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .oauth2Login()
            .userInfoEndpoint()
            .oidcUserService(this.oidcUserService())
            ...
    }

    private OAuth2UserService<OidcUserRequest, OidcUser> oidcUserService() {
        final OidcUserService delegate = new OidcUserService();

        return (userRequest) -> {
            // Delegate to the default implementation for loading a user
            OidcUser oidcUser = delegate.loadUser(userRequest);

            OAuth2AccessToken accessToken = userRequest.getAccessToken();
            Set<GrantedAuthority> mappedAuthorities = new HashSet<>();

            // TODO
            // 1) Fetch the authority information from the protected resource using accessToken
            // 2) Map the authority information to one or more GrantedAuthority's and add it to mappedAuthorities

            // 3) Create a copy of oidcUser but use the mappedAuthorities instead
            oidcUser = new DefaultOidcUser(mappedAuthorities, oidcUser.getIdToken(), oidcUser.getUserInfo());

            return oidcUser;
        };
    }
}

```

31.5.2 Configuring a Custom OAuth2User 译: 31.5.2配置自定义 OAuth2User

`CustomUserTypesOAuth2UserService` 是为自定义 `OAuth2User` 类型提供支持的 `OAuth2UserService` 的实现。

如果默认实现（`DefaultOAuth2User`）不适合您的需求，您可以定义自己的实现 `OAuth2User`。

以下代码演示了如何为GitHub注册自定义 `OAuth2User` 类型：

```

@EnableWebSecurity
public class OAuth2LoginSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .oauth2Login()
            .userInfoEndpoint()
            .customUserType(GitHubOAuth2User.class, "github")
            ...
    }
}

```

以下代码显示了GitHub的自定义 `OAuth2User` 类型的示例：

```

public class GitHubOAuth2User implements OAuth2User {
    private List<GrantedAuthority> authorities =
        AuthorityUtils.createAuthorityList("ROLE_USER");
    private Map<String, Object> attributes;
    private String id;
    private String name;
    private String login;
    private String email;

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return this.authorities;
    }

    @Override
    public Map<String, Object> getAttributes() {
        if (this.attributes == null) {
            this.attributes = new HashMap<>();
            this.attributes.put("id", this.getId());
            this.attributes.put("name", this.getName());
            this.attributes.put("login", this.getLogin());
            this.attributes.put("email", this.getEmail());
        }
        return attributes;
    }

    public String getId() {
        return this.id;
    }

    public void setId(String id) {
        this.id = id;
    }

    @Override
    public String getName() {
        return this.name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getLogin() {
        return this.login;
    }

    public void setLogin(String login) {
        this.login = login;
    }

    public String getEmail() {
        return this.email;
    }

    public void setEmail(String email) {
        this.email = email;
    }
}

```



`id`，`name`，`login`，并 `email` 在 GitHub® 的响应的 `UserInfo` 返回的属性。有关 `UserInfo` 端点返回的详细信息，请参阅“[Get the authenticated user](#)”的 API 文档。

31.5.3 OAuth 2.0 UserService #: 31.5.3 OAuth 2.0 UserService

`DefaultOAuth2UserService` 是支持标准 OAuth 2.0 Provider 的 `OAuth2UserService` 的实现。



`OAuth2UserService` 从 `UserInfo` 端点（通过在授权流程中使用授予客户端的访问令牌）获取最终用户（资源所有者）的用户属性，并以 `AuthenticatedPrincipal` 的形式返回 `OAuth2User`。

如果默认实现不适合您的需求，您可以为标准 OAuth 2.0 Provider 定义自己的实现 `OAuth2UserService`。

以下配置演示了如何配置自定义 `OAuth2UserService`：

```

@EnableWebSecurity
public class OAuth2LoginSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .oauth2Login()
            .userInfoEndpoint()
            .userService(this.oauth2UserService())
            ...
    }

    private OAuth2UserService<OAuth2UserRequest, OAuth2User> oauth2UserService() {
        return new CustomOAuth2UserService();
    }
}

```

31.5.4 OpenID Connect 1.0 UserService 译：31.5.4 OpenID Connect 1.0 UserService

`OidcUserService` 是支持 OpenID Connect 1.0 Provider 的 `OAuth2UserService` 的实现。



`OAuth2UserService` 负责从 `UserInfo` 端点（通过在授权流程中使用授予客户端的访问令牌）获取最终用户（资源所有者）的用户属性，并以 `AuthenticatedPrincipal` 的形式返回 `OidcUser`。

如果默认实现不符合您的需求，您可以为 OpenID Connect 1.0 Provider 定义您自己的 `OAuth2UserService` 实现。

以下配置演示了如何配置自定义 OpenID Connect 1.0 `OAuth2UserService`：

```

@EnableWebSecurity
public class OAuth2LoginSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .oauth2Login()
            .userInfoEndpoint()
            .oidcUserService(this.oidcUserService())
            ...
    }

    private OAuth2UserService<OidcUserRequest, OidcUser> oidcUserService() {
        return new CustomOidcUserService();
    }
}

```

32. JSP Tag Libraries 译：32. JSP 标签库

Spring Security 有自己的 taglib，它为访问安全信息和在 JSP 中应用安全约束提供了基本的支持。

32.1 Declaring the Taglib 译：32.1 声明 Taglib

要使用任何标签，您必须在 JSP 中声明安全 taglib：

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

32.2 The authorize Tag 译：32.2 授权标签

该标签用于确定是否应评估其内容。在 Spring Security 3.0 中，它可以以两种方式使用^[21]。第一种方法采用的是 `web-security expression`，在指定 `access` 标签的属性。表达式评估将委派给应用程序上下文中定义的 `SecurityExpressionHandler<FilterInvocation>`（您应该在 `<http>` 命名空间配置中启用 Web 表达式以确保此服务可用）。所以，例如，你可能有

```

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

    This content will only be visible to users who have the "supervisor" authority in their list of <tt>GrantedAuthority</tt>s.

</sec:authorize>

```

当与 Spring Security 的 `PermissionEvaluator` 结合使用时，该标签也可以用来检查权限。例如：

```

<sec:authorize access="hasPermission(#domain,'read') or hasPermission(#domain,'write')">

    This content will only be visible to users who have read or write permission to the Object found as a request attribute named "domain".

</sec:authorize>

```

一个常见的要求是只显示特定的链接，如果用户实际上被允许点击它。我们如何预先确定是否允许某些事情？该标签也可以在另一种模式下运行，该模式允许您将特定的 URL 定义为属性。如果用户被允许调用该 URL，那么标签主体将被评估，否则它将被跳过。所以你可能有类似的东西

```

<sec:authorize url="/admin">

    This content will only be visible to users who are authorized to send requests to the "/admin" URL.

</sec:authorize>

```

要使用此标记，您的应用程序上下文中还必须有一个 `WebInvocationPrivilegeEvaluator` 的实例。如果您使用的是名称空间，则会自动注册。这是一

个 `DefaultWebInvocationPrivilegeEvaluator` 的实例，它为所提供的URL创建一个虚拟web请求，并调用安全拦截器来查看请求是成功还是失败。这使您可以委派到您在 `<http>` 名称空间配置中使用 `intercept-url` 声明定义的访问控制设置，并节省必须在JSP内复制信息（如所需的角色）。这种方法也可以与提供HTTP方法的 `method` 属性相结合，以获得更具体的匹配。

通过将 `var` 属性设置为变量名称，可以将评估标记的布尔结果（无论是授予还是拒绝访问权）存储在页面上上下文范围变量中，从而避免需要复制和重新评估页。

32.2.1 Disabling Tag Authorization for Testing 译：32.2禁用标签授权以进行测试

在页面中隐藏未授权用户的链接并不妨碍他们访问URL。例如，他们可以直接将其输入到浏览器中。作为测试过程的一部分，您可能想要揭示隐藏区域，以检查链接是否真正在后端得到保护。如果将系统属性 `spring.security.disableUISecurity` 设置为 `true`，则 `authorize` 标记仍将运行，但不会隐藏其内容。默认情况下，它也会用 `... ` 标签包围内容。这使您可以显示具有特定CSS样式（如不同背景色）的“隐藏”内容。例如，尝试运行启用了此属性的“教程”示例应用程序。

如果要从默认的 `span` 标记更改周围的文本（或使用空字符串将其完全删除），还可以设置属性 `spring.security.securedUIPrefix` 和 `spring.security.securedUISuffix`。

32.3 The authentication Tag 译：32.3认证标签

该标签允许访问存储在安全上下文中的当前 `Authentication` 对象。它直接在JSP中呈现对象的属性。因此，举例来说，如果 `principal` 的财产 `Authentication` 是春天Security™的实例 `UserDetails` 对象，然后使用 `<sec:authentication property="principal.username" />` 将使当前用户的名称。

当然，没有必要使用JSP标签来处理这种事情，而且有些人更喜欢在视图中尽可能少地保留逻辑。您可以访问MVC控制器中的 `Authentication` 对象（通过调用 `SecurityContextHolder.getContext().getAuthentication()`）并将数据直接添加到您的模型以供视图渲染。

32.4 The accesscontrollist Tag 译：32.4accesscontrollist标签

这个标签只有在与Spring Security的ACL模块一起使用时才有效。它检查指定域对象的必需权限的逗号分隔列表。如果当前用户拥有所有这些权限，则标签正文将被评估。如果他们不愿意，它会被跳过。一个例子可能是



Caution

一般来说，这个标签应该被视为不推荐使用。而是使用[Section 32.2, “The authorize Tag”](#)。

```
<sec:accesscontrollist hasPermission="1,2" domainObject="${someObject}">
```

This will be shown if the user has all of the permissions represented by the values "1" or "2" on the given object.

```
</sec:accesscontrollist>
```

权限被传递给在应用上下文中定义的 `PermissionFactory`，并将它们转换为ACL `Permission` 实例，因此它们可以是工厂支持的任何格式 - 它们不必是整数，它们可以是 `READ` 字符串或 `WRITE`。如果找不到 `PermissionFactory`，则将使用 `DefaultPermissionFactory` 的实例。来自应用程序上下文的 `Ac1Service` 将用于为所提供的对象加载 `Ac1` 实例。 `Ac1` 将被调用所需的权限来检查它们是否都被授予。

该标签也支持 `var` 属性，方法与 `authorize` 标签相同。

32.5 The csrfInput Tag 译：32.5csrfInput标签

如果启用了CSRF保护，则此标记会为CSRF保护令牌插入一个隐藏表单字段，其中包含正确的名称和价值。如果CSRF保护未启用，则此标记不输出任何内容。

通常，Spring Security会为您使用的任何 `<form:form>` 标记自动插入一个CSRF表单字段，但如果由于某种原因您不能使用 `<form:form>`，则 `csrfInput` 替代。

您应该将此标记置于HTML `<form></form>` 块中，您通常会其中放置其他输入字段。不要将此标签放置在弹簧 `<form:form></form:form>` 块内。Spring Security自动处理Spring窗体。

```
<form method="post" action="/do/something">
  <sec:csrfInput />
  Name:<br />
  <input type="text" name="name" />
  ...
</form>
```

32.6 The csrfMetaTags Tag 译：32.6csrfMetaTags标签

如果启用CSRF保护，则此标记会插入包含CSRF保护令牌表单字段和标题名称以及CSRF保护标记值的元标记。这些元标记对于在应用程序中使用JavaScript中的CSRF保护很有用。

您应该将 `csrfMetaTags` 放置在HTML `<head></head>` 块内，您通常会其中放置其他元标记。一旦你使用这个标签，你可以使用JavaScript轻松访问表单字段名称，标题名称和标记值。在这个例子中使用jQuery使任务更容易。

```
<!DOCTYPE html>
<html>
<head>
<title>CSRF Protected JavaScript Page</title>
<meta name="description" content="This is the description for this page" />
<sec:csrfMetaTags />
<script type="text/javascript" language="javascript">

var csrfParameter = $("meta[name='_csrf_parameter']").attr("content");
var csrfHeader = $("meta[name='_csrf_header']").attr("content");
var csrfToken = $("meta[name='_csrf']").attr("content");

// using XMLHttpRequest directly to send an x-www-form-urlencoded request
var ajax = new XMLHttpRequest();
ajax.open("POST", "http://www.example.org/do/something", true);
ajax.setRequestHeader("Content-Type", "application/x-www-form-urlencoded data");
ajax.send(csrfParameter + "=" + csrfToken + "&name=John&...");

// using XMLHttpRequest directly to send a non-x-www-form-urlencoded request
var ajax = new XMLHttpRequest();
ajax.open("POST", "http://www.example.org/do/something", true);
ajax.setRequestHeader(csrfHeader, csrfToken);
ajax.send("...");

// using JQuery to send an x-www-form-urlencoded request
var data = {};
data[csrfParameter] = csrfToken;
data["name"] = "John";
...
$.ajax({
  url: "http://www.example.org/do/something",
  type: "POST",
  data: data,
  ...
});

// using JQuery to send a non-x-www-form-urlencoded request
var headers = {};
headers[csrfHeader] = csrfToken;
$.ajax({
  url: "http://www.example.org/do/something",
  type: "POST",
  headers: headers,
  ...
});

</script>
</head>
<body>
...
</body>
</html>
```

如果CSRF保护未启用，则 `csrfMetaTags` 输出任何内容。

[21] 也支持Spring Security 2.0的遗留选项，但不鼓励。

33. Java Authentication and Authorization Service (JAAS) Provider 译：33. Java认证授权服务（JAAS）提供者

33.1 Overview 译：33.1概述

Spring Security提供了一个能够将身份验证请求委托给Java身份验证和授权服务（JAAS）的软件包。这个包在下面详细讨论。

33.2 AbstractJaasAuthenticationProvider 译：33.2 AbstractJaasAuthenticationProvider

`AbstractJaasAuthenticationProvider`是提供的JAAS `AuthenticationProvider`实现的基础。子类必须实现一个创建 `LoginContext` 的方法。`AbstractJaasAuthenticationProvider`有许多可以注入的依赖关系，下面将对此进行讨论。

33.2.1 JAAS CallbackHandler 译：33.2.1 JAAS CallbackHandler

大多数JAAS `LoginModule`需要某种回调。这些回调通常用于从用户获取用户名和密码。

在Spring Security部署中，Spring Security负责此用户交互（通过身份验证机制）。因此，在认证请求委托给JAAS的时候，Spring Security的认证机制已经完全填充了一个包含JAAS `LoginModule`所需的全部信息的 `Authentication` 对象。

因此，Spring Security的JAAS包提供了两个默认的回调处理程序 `JaasNameCallbackHandler` 和 `JaasPasswordCallbackHandler`。每个这些回调处理程序都实现 `JaasAuthenticationCallbackHandler`。在大多数情况下，这些回调处理程序可以在不理解内部机制的情况下使用。

对于那些需要完全控制回调行为的人，内部 `AbstractJaasAuthenticationProvider` 将这 `JaasAuthenticationCallbackHandler` 与 `AbstractJaasAuthenticationProvider` 包装 `InternalCallbackHandler`。`InternalCallbackHandler`是实际实现JAAS正常 `CallbackHandler` 接口的类。任何时候使用JAAS `LoginModule`，都会传递一个配置应用程序上下文的列表 `InternalCallbackHandler`s。如果 `LoginModule` 要求对 `LoginModule` 进行回 `InternalCallbackHandler`，则回叫将依次传递给包裹的 `JaasAuthenticationCallbackHandler`。

33.2.2 JAAS AuthorityGranter 译：33.2.2 JAAS AuthorityGranter

JAAS与校长合作。甚至“角色”在JAAS中都被表示为负责人。另一方面，Spring Security与 `Authentication` 对象一起使用。每个 `Authentication` 对象包含一个主体，

以及多个 `GrantedAuthority`。为了促进这些不同概念之间的映射，Spring Security的JAAS包包含一个 `AuthorityGranter` 接口。

`AuthorityGranter` 负责检查JAAS校长并返回一组 `String`，代表分配给校长的当局。对于每个返回的权限字符串，`AbstractJaasAuthenticationProvider` 创建了一个 `JaasGrantedAuthority`（它实现了Spring Security的 `GrantedAuthority` 接口），其中包含授权字符串和 `AuthorityGranter` 传递的JAAS主体。该 `AbstractJaasAuthenticationProvider` 首先利用了JAAS成功认证用户A€™的凭证获取JAAS校长 `LoginModule`，然后访问 `LoginContext` 返回。致电 `LoginContext.getSubject().getPrincipals()`，每个委托人转交给 `AbstractJaasAuthenticationProvider.setAuthorityGranters(List)` 财产定义的每个 `AuthorityGranter`。

考虑到每个JAAS主体都具有特定于实现的含义，Spring Security不包括任何生产 `AuthorityGranter`。但是，单元测试中有一个 `TestAuthorityGranter`，它演示了一个简单的 `AuthorityGranter` 实现。

33.3 DefaultJaasAuthenticationProvider

`DefaultJaasAuthenticationProvider` 允许将JAAS `Configuration` 对象作为依赖项注入到该对象中。然后创建一个 `LoginContext` 利用喷射的JAAS `Configuration`。这意味着 `DefaultJaasAuthenticationProvider` 没有绑定任何特定的实施 `Configuration` 为 `JaasAuthenticationProvider` 的。

33.3.1 InMemoryConfiguration

为了便于将 `Configuration` 注入 `DefaultJaasAuthenticationProvider`，提供了名为 `InMemoryConfiguration` 的默认内存中实现。实现构造函数接受 `Map`，其中每个键表示登录配置名称，值表示 `Array` 的 `AppConfigurationEntry`。`InMemoryConfiguration` 还支持默认 `Array` 的 `AppConfigurationEntry`，如果没有映射所提供的内发现，将被使用的对象 `Map`。有关详细信息，请参阅 `InMemoryConfiguration` 的类别别javadoc。

33.3.2 DefaultJaasAuthenticationProvider Example Configuration

虽然 `InMemoryConfiguration` 的Spring配置可能比标准JAAS配置文件更详细，但与 `DefaultJaasAuthenticationProvider` 结合使用 `DefaultJaasAuthenticationProvider` 比 `JaasAuthenticationProvider` 更灵活，因为它不依赖于默认的 `Configuration` 实现。

下面提供了使用 `InMemoryConfiguration` 的 `DefaultJaasAuthenticationProvider` 的示例配置。请注意，`Configuration` 自定义实现也可以很容易地注入到 `DefaultJaasAuthenticationProvider` 中。

```
<bean id="jaasAuthProvider"
class="org.springframework.security.authentication.jaas.DefaultJaasAuthenticationProvider">
<property name="configuration">
<bean class="org.springframework.security.authentication.jaas.memory.InMemoryConfiguration">
<constructor-arg>
<map>
<!--
SPRINGSECURITY is the default loginContextName
for AbstractJaasAuthenticationProvider
-->
<entry key="SPRINGSECURITY">
<array>
<bean class="javax.security.auth.login.AppConfigurationEntry">
<constructor-arg value="sample.SampleLoginModule" />
<constructor-arg>
<util:constant static-field=
"javax.security.auth.login.AppConfigurationEntry$LoginModuleControlFlag.REQUIRED"/>
</constructor-arg>
<constructor-arg>
<map></map>
</constructor-arg>
</bean>
</array>
</entry>
</map>
</constructor-arg>
</bean>
</property>
<property name="authorityGranters">
<list>
<!-- You will need to write your own implementation of AuthorityGranter -->
<bean class="org.springframework.security.authentication.jaas.TestAuthorityGranter"/>
</list>
</property>
</bean>
```

33.4 JaasAuthenticationProvider

该 `JaasAuthenticationProvider` 假设默认 `Configuration` 是一个实例 `ConfigFile`。这个假设是为了尝试更新 `Configuration`。`JaasAuthenticationProvider` 然后使用默认 `Configuration` 创建 `LoginContext`。

让我们假设我们有一个JAAS登录配置文件，`/WEB-INF/login.conf`，其中包含以下内容：

```
JAASTest {
sample.SampleLoginModule required;
};
```

像所有Spring Security bean一样，`JaasAuthenticationProvider` 通过应用程序上下文进行配置。以下定义将对应于上述JAAS登录配置文件：

```
<bean id="jaasAuthenticationProvider"
class="org.springframework.security.authentication.jaas.JaasAuthenticationProvider">
<property name="loginConfig" value="/WEB-INF/login.conf"/>
<property name="loginContextName" value="JAASTest"/>
<property name="callbackHandlers">
<list>
<bean
class="org.springframework.security.authentication.jaas.JaasNameCallbackHandler"/>
<bean
class="org.springframework.security.authentication.jaas.JaasPasswordCallbackHandler"/>
</list>
</property>
<property name="authorityGranters">
<list>
<bean class="org.springframework.security.authentication.jaas.TestAuthorityGranter"/>
</list>
</property>
</bean>
```

33.5 Running as a Subject 译: 33.5 作为主题运行

如果已配置, 则 `JaasApiIntegrationFilter` 将尝试以 `Subject` 上的 `JaasAuthenticationToken` 。这意味着可以使用以下方式访问 `Subject` :

```
Subject subject = Subject.getSubject(AccessController.getContext());
```

这种集成可以使用 `jaas-api-provision` 属性轻松配置。 当与依赖JAAS主题被填充的遗留或外部API集成时, 此功能非常有用。

34. CAS Authentication 译: 34. CAS认证

34.1 Overview 译: 34.1 概述

JA-SIG在企业范围内生成一个称为CAS的单一登录系统。与其他举措不同, JA-SIG的中央身份验证服务是开源的, 广泛使用, 易于理解, 独立于平台, 并且支持代理功能。Spring Security完全支持CAS, 并提供了从Spring Security的单一应用部署到由企业级CAS服务器保护的多应用部署的简单迁移路径。

您可以通过<http://www.ja-sig.org/cas>了解有关CAS的更多信息。您还需要访问此网站以下载CAS服务器文件。

34.2 How CAS Works 译: 34.2 CAS如何工作

虽然CAS网站包含详细介绍CAS体系结构的文档, 但我们在Spring Security的上下文中再次提供总体概述。Spring Security 3.x支持CAS 3.在撰写本文时, CAS服务器版本为3.4。

您的企业中某处需要安装CAS服务器。CAS服务器仅仅是一个标准的WAR文件, 因此设置服务器没有任何困难。在WAR文件中, 您将自定义向用户显示的页面上的登录和其他单点登录。

部署CAS 3.4服务器时, 还需要指定CAS包含的 `AuthenticationHandler` 中的 `deployerConfigContext.xml` 。`AuthenticationHandler` 有一个简单的方法, 它返回一个布尔值, 以确定给定的一组凭证是否有效。您的 `AuthenticationHandler` 实现将需要链接到某种类型的后端身份验证存储库, 例如LDAP服务器或数据库。CAS本身包括许多 `AuthenticationHandler` s开箱即可协助。当您下载并部署服务器war文件时, 它将被设置为成功验证输入与其用户名相匹配的密码的用户, 这对测试非常有用。

除CAS服务器本身之外, 其他关键角色当然是在整个企业中部署的安全Web应用程序。这些Web应用程序被称为“服务”。有三种类型的服务。那些对服务票据进行身份验证的服务, 可以获得代理票据的服务和对代理票据进行身份验证的服务。验证代理票据的方式不同, 因为代理列表必须经过验证, 并且通常可以重新使用代理票据。

34.2.1 Spring Security and CAS Interaction Sequence 译: 34.2.1 Spring Security和CAS交互序列

Web浏览器, CAS服务器和Spring安全保护服务之间的基本交互如下所示:

- The web user is browsing the service's public pages. CAS or Spring Security is not involved.
- The user eventually requests a page that is either secure or one of the beans it uses is secure. Spring Security's `ExceptionHandlerFilter` will detect the `AccessDeniedException` or `AuthenticationException`.
- Because the user's `Authentication` object (or lack thereof) caused an `AuthenticationException`, the `ExceptionHandlerFilter` will call the configured `AuthenticationEntryPoint`. If using CAS, this will be the `CasAuthenticationEntryPoint` class.
- The `CasAuthenticationEntryPoint` will redirect the user's browser to the CAS server. It will also indicate a `service` parameter, which is the callback URL for the Spring Security service (your application). For example, the URL to which the browser is redirected might be <https://my.company.com/cas/login?service=https%3A%2F%2Fserver3.company.com%2Fwebapp%2Flogin/cas>.
- After the user's browser redirects to CAS, they will be prompted for their username and password. If the user presents a session cookie which indicates they've previously logged on, they will not be prompted to login again (there is an exception to this procedure, which we'll cover later). CAS will use the `PasswordHandler` (or `AuthenticationHandler` if using CAS 3.0) discussed above to decide whether the username and password is valid.
- Upon successful login, CAS will redirect the user's browser back to the original service. It will also include a `ticket` parameter, which is an opaque string representing the "service ticket". Continuing our earlier example, the URL the browser is redirected to might be <https://server3.company.com/webapp/login/cas?ticket=ST-0-ER94xMJmn6pha35CQRoZ>.
- Back in the service web application, the `CasAuthenticationFilter` is always listening for requests to `/login/cas` (this is configurable, but we'll use the defaults in this introduction). The processing filter will construct a `UsernamePasswordAuthenticationToken` representing the service ticket. The principal will be equal to `CasAuthenticationFilter.CAS_STATEFUL_IDENTIFIER`, whilst the credentials will be the service ticket opaque value. This authentication request will then be handed to the configured `AuthenticationManager`.
- The `AuthenticationManager` implementation will be the `ProviderManager`, which is in turn configured with the `CasAuthenticationProvider`. The `CasAuthenticationProvider` only responds to `UsernamePasswordAuthenticationToken` s containing the CAS-specific principal (such as `CasAuthenticationFilter.CAS_STATEFUL_IDENTIFIER`) and `CasAuthenticationToken` s (discussed later).
- `CasAuthenticationProvider` will validate the service ticket using a `TicketValidator` implementation. This will typically be a `Cas20ServiceTicketValidator` which is one of the classes included in the CAS client library. In the event the application needs to validate proxy tickets, the `Cas20ProxyTicketValidator` is used. The `TicketValidator` makes an HTTPS request to the CAS server in order to validate the service ticket. It may also include a proxy callback URL, which is included in this example: <https://my.company.com/cas/proxy/validate?service=https%3A%2F%2Fserver3.company.com%2Fwebapp%2Flogin/cas&ticket=ST-0-ER94xMJmn6pha35CQRoZ&pgtUrl=https://server3.company.com/webapp/login/cas/proxyreceptor>.

- Back on the CAS server, the validation request will be received. If the presented service ticket matches the service URL the ticket was issued to, CAS will provide an affirmative response in XML indicating the username. If any proxy was involved in the authentication (discussed below), the list of proxies is also included in the XML response.
- [OPTIONAL] If the request to the CAS validation service included the proxy callback URL (in the `pgtUrl` parameter), CAS will include a `pgtIou` string in the XML response. This `pgtIou` represents a proxy-granting ticket IOU. The CAS server will then create its own HTTPS connection back to the `pgtUrl`. This is to mutually authenticate the CAS server and the claimed service URL. The HTTPS connection will be used to send a proxy granting ticket to the original web application. For example, <https://server3.company.com/webapp/login/cas/proxyreceptor?pgtIou=PGTIQU-0-R0zgrl4pdAQwBvJWO3vnNpewqStbSGcq3vKB2SqSFFRnPHt&pgtId=PGT-1-si9YkkHLrACBo64msi3v2nf7cpCResXg5MpESZFArbaZiOKH>.
- The `Cas20TicketValidator` will parse the XML received from the CAS server. It will return to the `CasAuthenticationProvider` a `TicketResponse`, which includes the username (mandatory), proxy list (if any were involved), and proxy-granting ticket IOU (if the proxy callback was requested).
- Next `CasAuthenticationProvider` will call a configured `CasProxyDecider`. The `CasProxyDecider` indicates whether the proxy list in the `TicketResponse` is acceptable to the service. Several implementations are provided with Spring Security: `RejectProxyTickets`, `AcceptAnyCasProxy` and `NamedCasProxyDecider`. These names are largely self-explanatory, except `NamedCasProxyDecider` which allows a `List` of trusted proxies to be provided.
- `CasAuthenticationProvider` will next request a `AuthenticationUserDetailsService` to load the `GrantedAuthority` objects that apply to the user contained in the `Assertion`.
- If there were no problems, `CasAuthenticationProvider` constructs a `CasAuthenticationToken` including the details contained in the `TicketResponse` and the `GrantedAuthority`s.
- Control then returns to `CasAuthenticationFilter`, which places the created `CasAuthenticationToken` in the security context.
- The user's browser is redirected to the original page that caused the `AuthenticationException` (or a [custom destination](#) depending on the configuration).

这很好，你还在这里！现在看看这是如何配置的

34.3 Configuration of CAS Client 译：34.3 CAS客户端的配置

由于Spring Security的原因，CAS的Web应用程序变得非常简单。假设你已经知道了使用Spring Security的基础知识，所以下面不再介绍。我们假定正在使用基于名称空间的配置，并根据需要添加CAS bean。每节都建立在前一节的基础上。完整的[CAS sample application](#)可以在Spring Security Samples中找到。

34.3.1 Service Ticket Authentication 译：34.3.1 服务票证验证

本节介绍如何设置Spring Security来验证服务票证。通常这都是一个Web应用程序需要的。您需要将 `ServiceProperties` bean添加到您的应用程序上下文中。这代表您的CAS服务：

```
<bean id="serviceProperties"
      class="org.springframework.security.cas.ServiceProperties">
  <property name="service"
    value="https://localhost:8443/cas-sample/login/cas"/>
  <property name="sendRenew" value="false"/>
</bean>
```

`service` 必须等于 `service` 将监视的 `CasAuthenticationFilter`。`sendRenew` 默认为false，但应该设置为true，如果您的应用程序特别敏感。该参数的作用是告诉CAS登录服务，登录时的单一登录是不可接受的。相反，用户需要重新输入用户名和密码才能访问该服务。

应配置以下bean以启动CAS认证过程（假设您使用的是名称空间配置）：

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

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

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

为了运行CAS，`ExceptionHandlerFilter` 必须将其 `authenticationEntryPoint` 属性设置为 `CasAuthenticationEntryPoint` bean。这可以使用 `entry-point-ref` 轻松完成，如上例所示。`CasAuthenticationEntryPoint` 必须参考 `ServiceProperties` bean（上面讨论过），它提供了企业CAS登录服务器的URL。这是用户的浏览器将被重定向的地方。

`CasAuthenticationFilter` 与 `UsernamePasswordAuthenticationFilter`（用于基于表单的登录）具有非常相似的属性。您可以使用这些属性来自定义诸如认证成功和失败的行为。

接下来您需要添加一个 `CasAuthenticationProvider` 及其合作者：

```

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

<bean id="casAuthenticationProvider"
class="org.springframework.security.cas.authentication.CasAuthenticationProvider">
<property name="authenticationUserDetailsService">
<bean class="org.springframework.security.core.userdetails.UserDetailsServiceWrapper">
<constructor-arg ref="userService" />
</bean>
</property>
<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">
<!-- Password is prefixed with {noop} to indicate to DelegatingPasswordEncoder that
NoOpPasswordEncoder should be used.
This is not safe for production, but makes reading
in samples easier.
Normally passwords should be hashed using BCrypt -->
<security:user name="joe" password="{noop}joe" authorities="ROLE_USER" />
...
</security:user-service>

```

`CasAuthenticationProvider` 使用 `UserDetailsService` 实例为用户加载权限，一旦它们被CAS认证。我们在这里展示了一个简单的内存设置。请注意，`CasAuthenticationProvider` 实际上并未使用密码进行身份验证，但它确实使用了权限。

如果你回头参考 [How CAS Works](#) 部分，这些豆都是不言自明的。

这完成了CAS的最基本配置。如果您没有犯任何错误，您的Web应用程序应该在CAS单一登录框架内愉快地工作。Spring Security的其他部分不需要担心CAS处理身份验证的事实。在下面的章节中，我们将讨论一些（可选的）更高级的配置。

34.3.2 Single Logout 译：34.3.2单一注销

CAS协议支持Single Logout，可以很容易地添加到Spring Security配置中。以下是处理Single Logout的Spring Security配置的更新

```

<security:http entry-point-ref="casEntryPoint">
...
<security:logout logout-success-url="/cas-logout.jsp"/>
<security:custom-filter ref="requestSingleLogoutFilter" before="LOGOUT_FILTER"/>
<security:custom-filter ref="singleLogoutFilter" before="CAS_FILTER"/>
</security:http>

<!-- This filter handles a Single Logout Request from the CAS Server -->
<bean id="singleLogoutFilter" class="org.jasig.cas.client.session.SingleSignOutFilter"/>

<!-- This filter redirects to the CAS Server to signal Single Logout should be performed -->
<bean id="requestSingleLogoutFilter"
class="org.springframework.security.web.authentication.logout.LogoutFilter">
<constructor-arg value="https://localhost:9443/cas/logout"/>
<constructor-arg>
<bean class=
"org.springframework.security.web.authentication.logout.SecurityContextLogoutHandler"/>
</constructor-arg>
<property name="filterProcessesUrl" value="/logout/cas"/>
</bean>

```

`logout` 元素会将用户从本地应用程序注销，但不会终止与CAS服务器或任何其他已登录的应用程序的会话。`requestSingleLogoutFilter` 过滤器将允许请求 `/spring_security_cas_logout` 的URL将应用程序重定向到配置的CAS服务器注销URL。然后，CAS服务器将向所有登录的服务发送单一注销请求。该 `singleLogoutFilter` 处理通过查找单注销请求 `HttpSession` 静态 `Map`，然后无效的。

这可能会令人困惑，为什么需要 `logout` 元素和 `singleLogoutFilter`。由于 `SingleSignOutFilter` 只是将 `HttpSession` 存储在静态 `Map` 中，以便在其上调用无效，因此最好 `SingleSignOutFilter` 本地注销。使用上面的配置，注销流程将是：

- The user requests `/logout`, which would log the user out of the local application and send the user to the logout success page.
- The logout success page, `/cas-logout.jsp`, should instruct the user to click a link pointing to `/logout/cas` in order to logout out of all applications.
- When the user clicks the link, the user is redirected to the CAS single logout URL (<https://localhost:9443/cas/logout>).
- On the CAS Server side, the CAS single logout URL then submits single logout requests to all the CAS Services. On the CAS Service side, JASIG's `SingleSignOutFilter` processes the logout request by invalidating the original session.

下一步是将以下内容添加到您的web.xml中

```

<filter>
<filter-name>characterEncodingFilter</filter-name>
<filter-class>
    org.springframework.web.filter.CharacterEncodingFilter
</filter-class>
<init-param>
    <param-name>encoding</param-name>
    <param-value>UTF-8</param-value>
</init-param>
</filter>
<filter-mapping>
<filter-name>characterEncodingFilter</filter-name>
<url-pattern>/*</url-pattern>
</filter-mapping>
<listener>
<listener-class>
    org.jasig.cas.client.session.SingleSignOutHttpSessionListener
</listener-class>
</listener>

```

在使用SingleSignOutFilter时，您可能会遇到一些编码问题。因此，建议在使用SingleSignOutFilter时添加CharacterEncodingFilter以确保字符编码正确。有关详细信息，请参阅JASIG的文档。SingleSignOutHttpSessionListener确保当HttpSession到期时，用于单一注销的映射将被删除。

34.3.3 Authenticating to a Stateless Service with CAS 译：34.3.3使用CAS对无状态服务进行身份验证

本节介绍如何使用CAS对服务进行身份验证。换句话说，本节讨论如何设置使用CAS认证的服务的客户端。下一节将介绍如何设置无状态服务以使用CAS进行身份验证。

Configuring CAS to Obtain Proxy Granting Tickets 译：配置CAS以获取代理授予票证

为了向无状态服务进行身份验证，应用程序需要获取代理授予票证（PGT）。本节介绍如何配置Spring Security以获取PGT构建基于[Service Ticket Authentication]配置。

第一步是在您的Spring Security配置中包含ProxyGrantingTicketStorage。这用于存储由CasAuthenticationFilter获得的CasAuthenticationFilter以便它们可用于获取代理票证。示例配置如下所示

```

<!--
NOTE: In a real application you should not use an in memory implementation.
You will also want to ensure to clean up expired tickets by calling
ProxyGrantingTicketStorage.cleanup()
-->
<bean id="pgtStorage" class="org.jasig.cas.client.proxy.ProxyGrantingTicketStorageImpl"/>

```

下一步是更新CasAuthenticationProvider以获得代理票证。为此，请将Cas20ServiceTicketValidator替换为Cas20ProxyTicketValidator。应该将proxyCallbackUrl设置为应用程序将收到PGT的URL。最后，该配置还应引用ProxyGrantingTicketStorage以便它可以使用PGT来获取代理票证。您可以在下面找到一个配置更改的示例。

```

<bean id="casAuthenticationProvider"
class="org.springframework.security.cas.authentication.CasAuthenticationProvider">
...
<property name="ticketValidator">
<bean class="org.jasig.cas.client.validation.Cas20ProxyTicketValidator">
<constructor-arg value="https://localhost:9443/cas"/>
<property name="proxyCallbackUrl"
value="https://localhost:8443/cas-sample/login/cas/proxyreceptor"/>
<property name="proxyGrantingTicketStorage" ref="pgtStorage"/>
</bean>
</property>
</bean>

```

最后一步是更新CasAuthenticationFilter以接受PGT并将其存储在ProxyGrantingTicketStorage。这一点很重要的proxyReceptorUrl匹配proxyCallbackUrl的Cas20ProxyTicketValidator。示例配置如下所示。

```

<bean id="casFilter"
class="org.springframework.security.cas.web.CasAuthenticationFilter">
...
<property name="proxyGrantingTicketStorage" ref="pgtStorage"/>
<property name="proxyReceptorUrl" value="/login/cas/proxyreceptor"/>
</bean>

```

Calling a Stateless Service Using a Proxy Ticket 译：使用代理票证调用无状态服务

现在Spring Security获得PGT，您可以使用它们来创建可用于向无状态服务进行身份验证的代理票证。该CAS sample application包含在一个工作示例ProxyTicketSampleServlet。示例代码可以在下面找到：

```

protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
// NOTE: The CasAuthenticationToken can also be obtained using
// SecurityContextHolder.getContext().getAuthentication()
final CasAuthenticationToken token = (CasAuthenticationToken) request.getUserPrincipal();
// proxyTicket could be reused to make calls to the CAS service even if the
// target url differs
final String proxyTicket = token.getAssertion().getPrincipal().getProxyTicketFor(targetUrl);

// Make a remote call using the proxy ticket
final String serviceUrl = targetUrl+"?ticket="+URLEncoder.encode(proxyTicket, "UTF-8");
String proxyResponse = CommonUtils.getResponseFromServer(serviceUrl, "UTF-8");
...
}

```

34.3.4 Proxy Ticket Authentication 译：34.3.4代理票证认证

`CasAuthenticationProvider` 区分有状态和无状态客户端。有状态的客户端被视为提交给 `filterProcessUrl` 的 `CasAuthenticationFilter`。无状态的客户端是任何呈现认证请求 `CasAuthenticationFilter` 上比另一个URL `filterProcessUrl`。

由于远程协议无法在 `HttpSession` 的上下文中呈现自己，因此不可能依赖将请求之间的会话存储安全上下文的默认实践。此外，由于CAS服务器在验证 `TicketValidator` 之后使其失效，所以在后续请求中显示相同的代理票证将不起作用。

一个显而易见的选择是远程协议客户端根本不使用CAS。但是，这将消除CAS的许多理想功能。作为中间地带，`CasAuthenticationProvider` 使用 `StatelessTicketCache`。这仅用于使用等于 `CasAuthenticationFilter.CAS_STATELESS_IDENTIFIER` 的主体的无状态客户。`CasAuthenticationProvider` 会将结果 `CasAuthenticationToken` 存储在 `CasAuthenticationToken` 中，并以代理票据为 `StatelessTicketCache` 进行处理。因此，远程协议客户端可以呈现相同的代理票证，并且 `CasAuthenticationProvider` 将不需要联系CAS服务器进行验证（除第一个请求外）。一旦通过身份验证，代理票证就可以用于原始目标服务以外的URL。

本部分构建在前面的部分以适应代理票证认证。第一步是指定验证所有工件，如下所示。

```
<bean id="serviceProperties"
      class="org.springframework.security.cas.ServiceProperties">
    ...
    <property name="authenticateAllArtifacts" value="true"/>
</bean>
```

下一步是指定 `serviceProperties` 和 `authenticationDetailsSource` 为 `CasAuthenticationFilter`。`serviceProperties` 属性指示 `CasAuthenticationFilter` 尝试验证所有工件而不是 `filterProcessUrl` 上存在的 `filterProcessUrl`。该 `ServiceAuthenticationDetailsSource` 创建 `ServiceAuthenticationDetails`，以确保当前的URL，基于 `HttpServletRequest`，正在验证检票时使用的服务URL。生成服务URL的方法可以通过注入返回自定义 `ServiceAuthenticationDetails` 的自定义 `AuthenticationDetailsSource` 来定制。

```
<bean id="casFilter"
      class="org.springframework.security.cas.web.CasAuthenticationFilter">
    ...
    <property name="serviceProperties" ref="serviceProperties"/>
    <property name="authenticationDetailsSource">
        <bean class=
            "org.springframework.security.cas.web.authentication.ServiceAuthenticationDetailsSource">
            <constructor-arg ref="serviceProperties"/>
        </bean>
    </property>
</bean>
```

您还需要更新 `CasAuthenticationProvider` 才能处理代理票证。为此，请将 `Cas20ServiceTicketValidator` 替换为 `Cas20ProxyTicketValidator`。您需要配置 `statelessTicketCache` 以及您想要接受的代理。您可以在下面找到接受所有代理所需更新的示例。

```
<bean id="casAuthenticationProvider"
      class="org.springframework.security.cas.authentication.CasAuthenticationProvider">
    ...
    <property name="ticketValidator">
        <bean class="org.jasig.cas.client.validation.Cas20ProxyTicketValidator">
            <constructor-arg value="https://localhost:9443/cas"/>
            <property name="acceptAnyProxy" value="true"/>
        </bean>
    </property>
    <property name="statelessTicketCache">
        <bean class="org.springframework.security.cas.authentication.EhCacheBasedTicketCache">
            <property name="cache">
                <bean class="net.sf.ehcache.Cache">
                    <init-method="initialise" destroy-method="dispose">
                        <constructor-arg value="casTickets"/>
                        <constructor-arg value="50"/>
                        <constructor-arg value="true"/>
                        <constructor-arg value="false"/>
                        <constructor-arg value="3600"/>
                        <constructor-arg value="900"/>
                    </bean>
                </property>
            </bean>
        </property>
    </bean>
</property>
</bean>
```

35. X.509 Authentication 译：35.X.509认证

35.1 Overview 译：35.1概述

X.509证书认证的最常见用途是验证使用SSL时服务器的身份，最常见的情况是从浏览器使用HTTPS。浏览器将自动检查由服务器提供的证书是否由其维护的可信证书颁发机构列表中的一个颁发（即数字签名）。

您也可以使用SSL进行“相互认证”；服务器将作为SSL握手的一部分向客户端请求有效的证书。服务器将通过检查其证书是否由可接受的权限进行签名来验证客户端。如果提供了有效的证书，它可以通过应用程序中的Servlet API获取。Spring Security X.509模块使用过滤器提取证书。它将证书映射到应用程序用户，并加载该用户的一组授权权限，以用于标准Spring Security基础结构。

在尝试将它用于Spring Security之前，您应该熟悉使用证书并为您的Servlet容器设置客户端身份验证。大部分工作是创建和安装合适的证书和密钥。例如，如果您正在使用Tomcat，请阅读<http://tomcat.apache.org/tomcat-6.0-doc/ssl-howto.html>的说明。在使用Spring Security进行试用之前，务必使用它

35.2 Adding X.509 Authentication to Your Web Application 译：35.2将X.509身份验证添加到您的Web应用程序

启用X.509客户端身份验证非常简单。只需将 `<x509/>` 元素添加到您的http安全名称空间配置。


```
<http>
...
<x509 subject-principal-regex="CN=(.*)", user-service-ref="userService"/>
</http>
```

该元素有两个可选属性：

- `subject-principal-regex`. The regular expression used to extract a username from the certificate's subject name. The default value is shown above. This is the username which will be passed to the `UserDetailsService` to load the authorities for the user.
- `user-service-ref`. This is the bean id of the `UserDetailsService` to be used with X.509. It isn't needed if there is only one defined in your application context.

`subject-principal-regex` 应该包含一个组。例如，默认表达式 `CN = (. * ?)` 匹配公共名称字段。因此，如果证书中的主题名称是 `CN = Jimi Hendrix, OU =`，则会给出用户名 `Jimi Hendrix`。这些匹配不区分大小写。因此，`emailAddress = (. ?)` 将匹配 `EMAILADDRESS = `，`CN = ...`，给出用户名 ``。如果客户端提交证书并且成功提取了有效的用户名，则安全上下文中应该有一个有效的 `Authentication` 对象。如果没有找到证书，或者找不到相应的用户，那么安全上下文将保持空白。这意味着您可以使用其他选项（如基于表单的登录）轻松使用 X.509 身份验证。

35.3 Setting up SSL in Tomcat

译：35.3在 Tomcat 中设置 SSL

Spring Security 项目的 `samples/certificate` 目录中有一些预生成的证书。如果您不想生成自己的文件，您可以使用它们来启用 SSL 进行测试。文件 `server.jks` 包含服务器证书，私钥和颁发证书颁发机构证书。还有一些来自示例应用程序的用户的客户端证书文件。您可以在浏览器中安装这些以启用 SSL 客户端身份验证。

要运行带有 SSL 支持的 tomcat，请将 `server.jks` 文件放入 tomcat `conf` 目录中，并将以下连接器添加到 `server.xml` 文件

```
<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` 也可以设置为 `want` 如果即使客户端不提供证书，仍希望 SSL 连接成功。除非您使用非 X.509 身份验证机制（例如表单身份验证），否则未提供证书的客户端不能访问由 Spring Security 保护的任何对象。

36. Run-As Authentication Replacement

译：36运行认证替换

36.1 Overview

译：36.1概述

该 `AbstractSecurityInterceptor` 能够暂时代替 `Authentication` 的对象 `SecurityContext` 和 `SecurityContextHolder` 中的安全对象回调阶段。只有在 `AuthenticationManager` 和 `AccessDecisionManager` 成功处理了原始 `Authentication` 对象时才会发生这种情况。`RunAsManager` 将指示替换 `Authentication` 对象，如果有的话，应在 `SecurityInterceptorCallback` 期间使用。

通过在安全对象回调阶段暂时替换 `Authentication` 对象，受保护的调用将能够调用需要不同身份验证和授权凭证的其他对象。它也能够对特定的 `GrantedAuthority` 对象执行任何内部安全检查。由于 Spring Security 提供了许多帮助类，可以根据 `SecurityContextHolder` 的内容自动配置远程协议，因此这些运行替代在调用远程 Web 服务时特别有用

36.2 Configuration

译：36.2配置

Spring Security 提供了一个 `RunAsManager` 接口：

```
Authentication buildRunAs(Authentication authentication, Object object,
    List<ConfigAttribute> config);

boolean supports(ConfigAttribute attribute);

boolean supports(Class clazz);
```

第一方法返回 `Authentication` 对象应当替换现有 `Authentication` 对象方法调用的持续时间。如果该方法返回 `null`，则表示不应该进行替换。第二种方法被 `AbstractSecurityInterceptor` 用作启动验证配置属性的一部分。`supports(Class)` 方法由安全拦截器实现调用，以确保配置的 `RunAsManager` 支持安全拦截器将呈现的安全对象的类型。

Spring Security 提供了 `RunAsManager` 一个具体实现。该 `RunAsManagerImpl` 类返回更换 `RunAsUserToken` 如有 `ConfigAttribute` 开头 `RUN_AS_`。如果有任何此类 `ConfigAttribute` 发现，更换 `RunAsUserToken` 将包含相同的本金，证书和授予的权限与原始 `Authentication` 对象，以新的沿 `SimpleGrantedAuthority` 每个 `RUN_AS_ ConfigAttribute`。每个新 `SimpleGrantedAuthority` 将与前缀 `ROLE_`，其次为 `RUN_AS ConfigAttribute`。例如，`RUN_AS_SERVER` 将导致包含 `ROLE_RUN_AS_SERVER` 授权的替换 `RunAsUserToken`。

替换 `RunAsUserToken` 就像任何其他 `Authentication` 对象一样。它需要通过 `AuthenticationManager` 认证，可能通过授权 `AuthenticationProvider`。`RunAsImplAuthenticationProvider` 执行这样的认证。它只是接受任何 `RunAsUserToken` 呈现有效。

为确保恶意代码不会创建 `RunAsUserToken` 并将其呈现以供 `RunAsImplAuthenticationProvider` 保证接受，密钥的散列值将存储在所有生成的令牌中。`RunAsManagerImpl` 和 `RunAsImplAuthenticationProvider` 使用相同的键在 bean 上下文中创建：

```
<bean id="runAsManager"
    class="org.springframework.security.access.intercept.RunAsManagerImpl">
<property name="key" value="my_run_as_password"/>
</bean>

<bean id="runAsAuthenticationProvider"
    class="org.springframework.security.access.intercept.RunAsImplAuthenticationProvider">
<property name="key" value="my_run_as_password"/>
</bean>
```

通过使用相同的密钥，每个 `RunAsUserToken` 可以验证它是由批准的 `RunAsManagerImpl` 创建的。出于安全原因，创建 `RunAsUserToken` 后不可变

37. Spring Security Crypto Module 译: 37. Spring Security加密模块

37.1 Introduction 译: 37.1介绍

Spring Security Crypto模块为对称加密，密钥生成和密码编码提供支持。该代码作为核心模块的一部分进行分发，但不依赖于任何其他Spring Security（或Spring）代码。

37.2 Encryptors 译: 37.2加密器

Encryptors类提供了构造对称加密器的工厂方法。使用这个类，您可以创建ByteEncryptors以原始字节[]形式加密数据。您还可以构造TextEncryptors来加密文本字符串。加密器是线程安全的。

37.2.1 BytesEncryptor 译: 37.2.1 BytesEncryptor

使用Encryptors.standard工厂方法构建“标准”BytesEncryptor:

```
Encryptors.standard("password", "salt");
```

“标准”加密方法是使用PKCS #5的PBKDF2（基于密码的密钥推导函数#2）的256位AES。此方法需要Java 6.用于生成SecretKey的密码应保存在安全的地方，不要共享。salt用于防止在加密数据被泄露的情况下对字典进行字典攻击。一个16字节的随机初始化向量也被应用，因此每个加密的消息都是唯一的。

提供的salt应该是以十六进制编码的字符串形式，是随机的，并且长度至少为8个字节。这种盐可以使用KeyGenerator生成:

```
String salt = KeyGenerators.string().generateKey(); // generates a random 8-byte salt that is then hex-encoded
```

37.2.2 TextEncryptor 译: 37.2.2 TextEncryptor

使用Encryptors.text工厂方法构建标准的TextEncryptor:

```
Encryptors.text("password", "salt");
```

TextEncryptor使用标准的BytesEncryptor来加密文本数据。加密结果以十六进制编码字符串的形式返回，以便于在文件系统或数据库中存储。

使用Encryptors.queryableText工厂方法构造一个“可查询”的TextEncryptor:

```
Encryptors.queryableText("password", "salt");
```

可查询TextEncryptor和标准TextEncryptor之间的区别与初始化向量（iv）处理有关。在可查询的TextEncryptor#encrypt操作中使用的iv是共享的，或者是常量，并且不是随机生成的。这意味着多次加密的相同文本将始终产生相同的加密结果。这不太安全，但对于需要查询的加密数据来说是必需的。可查询加密文本的一个例子是一个OAuth apiKey。

37.3 Key Generators 译: 37.3密钥生成器

KeyGenerators类提供了许多便捷工厂方法来构建不同类型的密钥生成器。使用这个类，您可以创建一个BytesKeyGenerator来生成byte[]键。您也可以构造一个StringKeyGenerator来生成字符串键。KeyGenerators是线程安全的。

37.3.1 BytesKeyGenerator 译: 37.3.1 BytesKeyGenerator

使用KeyGenerators.secureRandom工厂方法生成由SecureRandom实例支持的BytesKeyGenerator:

```
BytesKeyGenerator generator = KeyGenerators.secureRandom();
byte[] key = generator.generateKey();
```

默认密钥长度是8个字节。还有一个可以控制密钥长度的KeyGenerators.secureRandom变体:

```
KeyGenerators.secureRandom(16);
```

使用KeyGenerators.shared工厂方法构造一个BytesKeyGenerator，它总是在每次调用时返回相同的键:

```
KeyGenerators.shared(16);
```

37.3.2 StringKeyGenerator 译: 37.3.2 StringKeyGenerator

使用KeyGenerators.string工厂方法构造一个8字节的SecureRandom KeyGenerator，它将每个键的字符串编码为一个字符串:

```
KeyGenerators.string();
```

37.4 Password Encoding 译: 37.4密码编码

spring-security-crypto模块的密码包提供对密码编码的支持。`PasswordEncoder`是中央服务接口并具有以下签名:

```
public interface PasswordEncoder {

    String encode(String rawPassword);

    boolean matches(String rawPassword, String encodedPassword);
}
```

如果rawPassword一旦编码，就等于encodedPassword，则匹配方法返回true。此方法旨在支持基于密码的身份验证方案。

`BCryptPasswordEncoder`实现使用广泛支持的“bcrypt”算法来散列密码。Bcrypt使用一个随机的16字节盐值，并且是故意缓慢的算法，以阻止密码破解者。它所做的工作量可以使用从4到31的值的“强度”参数进行调整。值越高，计算散列所需的工作就越多。默认值为10.您可以在已部署的系统中更改此值，而不会影响现有密码，因为该值也

存储在已编码的散列中。

```
// Create an encoder with strength 16
BCryptPasswordEncoder encoder = new BCryptPasswordEncoder(16);
String result = encoder.encode("myPassword");
assertTrue(encoder.matches("myPassword", result));
```

`Pbkdf2PasswordEncoder` 实现使用PBKDF2算法来散列密码。为了破解密码破解PBKDF2是一个故意缓慢的算法，应该调整大约0.5秒来验证系统上的密码。

```
// Create an encoder with all the defaults
Pbkdf2PasswordEncoder encoder = new Pbkdf2PasswordEncoder();
String result = encoder.encode("myPassword");
assertTrue(encoder.matches("myPassword", result));
```

38. Concurrency Support 译: 并发支持

在大多数环境中，安全性以每个 `Thread` 基础进行存储。这意味着，当工作是在一个新的做 `Thread`，该 `SecurityContext` 丢失。Spring Security提供了一些基础设施来帮助用户更容易。Spring Security在多线程环境中为Spring Security提供低级抽象。事实上，这就是Spring Security与 [Section 16.2.4](#), ["AsyncContext.start\(Runnable\)"](#)和 [Section 39.4, "Spring MVC Async Integration"](#)集成的基础。

38.1 DelegatingSecurityContextRunnable 译: 38.1 DelegatingSecurityContextRunnable

`DelegatingSecurityContextRunnable` 是Spring Security并发支持中最基本的构建块 `DelegatingSecurityContextRunnable`。它包装了一个代表 `Runnable`，以便为代表初始化 `SecurityContextHolder`，`SecurityContext` 为代表指定了 `SecurityContext`。然后调用 `SecurityContextHolder` `Runnable`，确保事后清除 `SecurityContextHolder`。`DelegatingSecurityContextRunnable` 看起来像这样：

```
public void run() {
    try {
        SecurityContextHolder.setContext(securityContext);
        delegate.run();
    } finally {
        SecurityContextHolder.clearContext();
    }
}
```

虽然非常简单，但它可以将 `SecurityContext` 从一个线程转移到另一个线程。这很重要，因为在大多数情况下，`SecurityContextHolder` 是以每个线程为基础进行操作的。例如，您可能使用了Spring Security的 [Section 43.4.1, "<global-method-security>"](#)支持来保护您的某项服务。现在，您可以轻松地转移 `SecurityContext` 当前的 `Thread` 至 `Thread` 调用的安全服务。下面是您可以如何做到这一点的一个例子：

```
Runnable originalRunnable = new Runnable() {
    public void run() {
        // invoke secured service
    }
};

SecurityContext context = SecurityContextHolder.getContext();
DelegatingSecurityContextRunnable wrappedRunnable =
    new DelegatingSecurityContextRunnable(originalRunnable, context);

new Thread(wrappedRunnable).start();
```

上面的代码执行以下步骤：

- Creates a `Runnable` that will be invoking our secured service. Notice that it is not aware of Spring Security
- Obtains the `SecurityContext` that we wish to use from the `SecurityContextHolder` and initializes the `DelegatingSecurityContextRunnable`
- Use the `DelegatingSecurityContextRunnable` to create a Thread
- Start the Thread we created

由于从 `SecurityContextHolder` 创建 `DelegatingSecurityContextRunnable` 与 `SecurityContext` 相当常见，`SecurityContextHolder` 有一个快捷方式构造函数。以下代码与上面的代码相同：

```
Runnable originalRunnable = new Runnable() {
    public void run() {
        // invoke secured service
    }
};

DelegatingSecurityContextRunnable wrappedRunnable =
    new DelegatingSecurityContextRunnable(originalRunnable);

new Thread(wrappedRunnable).start();
```

我们所使用的代码很容易使用，但它仍然需要我们使用Spring Security的知识。在下一节中，我们将看看如何利用 `DelegatingSecurityContextExecutor` 来隐藏我们使用Spring Security的事实。

38.2 DelegatingSecurityContextExecutor 译: 38.2 DelegatingSecurityContextExecutor

在上一节中，我们发现使用 `DelegatingSecurityContextRunnable` 很容易，但它并不理想，因为我们必须了解Spring Security才能使用它。让我们来看看 `DelegatingSecurityContextExecutor` 如何屏蔽我们的代码，使其免于我们使用Spring Security的任何知识。

设计 `DelegatingSecurityContextExecutor` 是非常相似的 `DelegatingSecurityContextRunnable`，除了它接受委托 `Executor`，而不是委托 `Runnable`。您可以在下面看到一个如何使用它的例子：

```

SecurityContext context = SecurityContextHolder.createEmptyContext();
Authentication authentication =
    new UsernamePasswordAuthenticationToken("user", "doesnotmatter", AuthorityUtils.createAuthorityList("ROLE_USER"));
context.setAuthentication(authentication);

SimpleAsyncTaskExecutor delegateExecutor =
    new SimpleAsyncTaskExecutor();
DelegatingSecurityContextExecutor executor =
    new DelegatingSecurityContextExecutor(delegateExecutor, context);

Runnable originalRunnable = new Runnable() {
    public void run() {
        // invoke secured service
    }
};

executor.execute(originalRunnable);

```

该代码执行以下步骤：

- Creates the `SecurityContext` to be used for our `DelegatingSecurityContextExecutor`. Note that in this example we simply create the `SecurityContext` by hand. However, it does not matter where or how we get the `SecurityContext` (i.e. we could obtain it from the `SecurityContextHolder` if we wanted).
- Creates a delegateExecutor that is in charge of executing submitted `Runnable`s
- Finally we create a `DelegatingSecurityContextExecutor` which is in charge of wrapping any `Runnable` that is passed into the `execute` method with a `DelegatingSecurityContextRunnable`. It then passes the wrapped `Runnable` to the delegateExecutor. In this instance, the same `SecurityContext` will be used for every `Runnable` submitted to our `DelegatingSecurityContextExecutor`. This is nice if we are running background tasks that need to be run by a user with elevated privileges.
- At this point you may be asking yourself "How does this shield my code of any knowledge of Spring Security?" Instead of creating the `SecurityContext` and the `DelegatingSecurityContextExecutor` in our own code, we can inject an already initialized instance of `DelegatingSecurityContextExecutor`.

```

@Autowired
private Executor executor; // becomes an instance of our DelegatingSecurityContextExecutor

public void submitRunnable() {
    Runnable originalRunnable = new Runnable() {
        public void run() {
            // invoke secured service
        }
    };
    executor.execute(originalRunnable);
}

```

现在我们的代码不知道 `SecurityContext` 正在传播到 `Thread`，然后 `originalRunnable` 被执行，然后 `SecurityContextHolder` 被清除。在这个例子中，正在使用同一个用户来执行每个线程。在我们调用 `executor.execute(Runnable)`（即当前登录的用户）处理 `originalRunnable` 如果我们想要使用 `SecurityContextHolder` 中的用户呢？这可以通过从我们的 `DelegatingSecurityContextExecutor` 构造函数中删除 `SecurityContext` 参数来完成。例如：

```

SimpleAsyncTaskExecutor delegateExecutor = new SimpleAsyncTaskExecutor();
DelegatingSecurityContextExecutor executor =
    new DelegatingSecurityContextExecutor(delegateExecutor);

```

现在随时 `executor.execute(Runnable)` 执行 `SecurityContext` 首先获得 `SecurityContextHolder`，然后 `SecurityContext` 用于创建我们的 `DelegatingSecurityContextRunnable`。这意味着我们正在执行我们 `Runnable`，通过用来调用同一用户 `executor.execute(Runnable)` 代码。

38.3 Spring Security Concurrency Classes 译：38.3 Spring安全并发类

请参阅Javadoc以获取与Java并发API和Spring Task抽象的其他集成。一旦你理解了前面的代码，它们就不言自明了。

- `DelegatingSecurityContextCallable`
- `DelegatingSecurityContextExecutor`
- `DelegatingSecurityContextExecutorService`
- `DelegatingSecurityContextRunnable`
- `DelegatingSecurityContextScheduledExecutorService`
- `DelegatingSecurityContextSchedulingTaskExecutor`
- `DelegatingSecurityContextAsyncTaskExecutor`
- `DelegatingSecurityContextTaskExecutor`

39. Spring MVC Integration 译：39. Spring MVC集成

Spring Security提供了许多与Spring MVC的可选集成。本节将更详细地介绍集成。

39.1 @EnableWebMvcSecurity 译：39.1 @EnableWebMvcSecurity



从Spring Security 4.0开始，不推荐使用 `@EnableWebMvcSecurity`。替换是 `@EnableWebSecurity`，这将决定添加基于类路径的Spring MVC功能。

要使Spring Security与Spring MVC集成，请将 `@EnableWebSecurity` 注释添加到您的配置中。



Spring Security提供使用Spring MVC的™的配置 `WebMvcConfigurer`。这意味着如果您使用更高级的选项，比如直接与 `WebMvcConfigurationSupport` 集成，那么您将需要手动提供Spring Security配置。

39.2 MvcRequestMatcher 译：39.2 MvcRequestMatcher

Spring Security与Spring MVC如何在 `MvcRequestMatcher` URL上进行深度集成。这有助于确保您的安全规则与用于处理您的请求的逻辑相匹配。

为了使用 `MvcRequestMatcher` 您必须将Spring Security配置与 `ApplicationContext` 放置在相同的 `DispatcherServlet`。这是必要的，因为Spring Security的 `MvcRequestMatcher` 预计名称为 `mvcHandlerMappingIntrospector` 的 `HandlerMappingIntrospector` bean将由用于执行匹配的Spring MVC配置注册。

对于 `web.xml` 这意味着您应该将您的配置置于 `DispatcherServlet.xml`。

```
<listener>
  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>

<!-- All Spring Configuration (both MVC and Security) are in /WEB-INF/spring/ -->
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>/WEB-INF/spring/*.xml</param-value>
</context-param>

<servlet>
  <servlet-name>spring</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <!-- Load from the ContextLoaderListener -->
  <init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value></param-value>
  </init-param>
</servlet>

<servlet-mapping>
  <servlet-name>spring</servlet-name>
  <url-pattern>/</url-pattern>
</servlet-mapping>
```

低于 `WebSecurityConfiguration` 放置在 `DispatcherServlet` 的 `ApplicationContext`。

```
public class SecurityInitializer extends
    AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected Class<?>[] getRootConfigClasses() {
        return null;
    }

    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class[] { RootConfiguration.class,
            WebMvcConfiguration.class };
    }

    @Override
    protected String[] getServletMappings() {
        return new String[] { "/" };
    }
}
```



始终建议通过匹配 `HttpServletRequest` 和方法安全性来提供授权规则。

通过在 `HttpServletRequest` 上进行匹配来提供授权规则是很好的，因为它在代码路径的早期发生并有助于减少 `attack surface`。方法安全性可确保如果有人绕过了Web授权规则，则您的应用程序仍处于安全状态。这就是所谓的 `Defence in Depth`

考虑一个映射如下的控制器：

```
@RequestMapping("/admin")
public String admin() {
```

如果我们想限制管理员用户访问此控制器方法，开发人员可以通过将 `HttpServletRequest` 与以下内容进行匹配来提供授权规则：

```
protected configure(HttpSecurity http) throws Exception {
    http
        .authorizeRequests()
        .antMatchers("/admin").hasRole("ADMIN");
}
```

或以XML格式

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

无论使用哪种配置，URL `/admin` 都需要经过身份验证的用户成为管理员用户。但是，根据我们的Spring MVC配置，URL `/admin.html` 也将映射到我们的 `admin()` 方法。另外，根据我们的Spring MVC配置，URL `/admin/` 也将映射到我们的 `admin()` 方法。

问题是我们的安全规则只保护 `/admin`。我们可以为Spring MVC的所有排列添加额外的规则，但这将是非常冗长和乏味的。

相反，我们可以利用Spring Security的 `MvcRequestMatcher`。以下配置将通过使用Spring MVC匹配URL来保护Spring MVC将匹配的相同URL。

```
protected configure(HttpSecurity http) throws Exception {
    http
        .authorizeRequests()
        .mvcMatchers("/admin").hasRole("ADMIN");
}
```

或以XML格式

```
<http request-matcher="mvc">
  <intercept-url pattern="/admin" access="hasRole('ADMIN')"/>
</http>
```

39.3 @AuthenticationPrincipal #: 39.3 @AuthenticationPrincipal

Spring Security提供了`AuthenticationPrincipalArgumentResolver`，它可以自动解析Spring MVC参数的当前`Authentication.getPrincipal()`。通过使用`@EnableWebSecurity`您会自动将其添加到Spring MVC配置中。如果您使用基于XML的配置，你必须自己添加它。例如：

```
<mvc:annotation-driven>
  <mvc:argument-resolvers>
    <bean class="org.springframework.security.web.method.annotation.AuthenticationPrincipalArgumentResolver" />
  </mvc:argument-resolvers>
</mvc:annotation-driven>
```

一旦正确配置了`AuthenticationPrincipalArgumentResolver`，就可以在Spring MVC层完全脱离Spring Security。

考虑这样一种情况：自定义`UserDetailsService`返回一个`Object`实现`UserDetails`和自己`CustomUser` `Object`。可以使用以下代码访问当前经过身份验证的用户`CustomUser`：

```
@RequestMapping("/messages/inbox")
public ModelAndView findMessagesForUser() {
    Authentication authentication =
        SecurityContextHolder.getContext().getAuthentication();
    CustomUser custom = (CustomUser) authentication == null ? null : authentication.getPrincipal();

    // .. find messages for this user and return them ...
}
```

从Spring Security 3.2开始，我们可以通过添加注释来更直接地解决争论。例如：

```
import org.springframework.security.core.annotation.AuthenticationPrincipal;

// ...

@RequestMapping("/messages/inbox")
public ModelAndView findMessagesForUser(@AuthenticationPrincipal CustomUser customUser) {

    // .. find messages for this user and return them ...
}
```

有时可能需要以某种方式改变校长。例如，如果`CustomUser`需要是最终的，则无法扩展。在这种情况下，`UserDetailsService`可能会返回实现`UserDetails`的`Object`，并提供一个名为`getCustomUser`的方法来访问`CustomUser`。例如，它可能看起来像：

```
public class CustomUserUserDetails extends User {
    // ...
    public CustomUser getCustomUser() {
        return customUser;
    }
}
```

然后，我们可以访问`CustomUser`使用SpEL expression使用`Authentication.getPrincipal()`作为根对象：

```
import org.springframework.security.core.annotation.AuthenticationPrincipal;

// ...

@RequestMapping("/messages/inbox")
public ModelAndView findMessagesForUser(@AuthenticationPrincipal(expression = "customUser") CustomUser customUser) {

    // .. find messages for this user and return them ...
}
```

我们也可以在我们的SpEL表达式中引用Beans。例如，如果我们使用JPA来管理用户，并且我们想修改并保存当前用户的属性，可以使用以下内容。

```
import org.springframework.security.core.annotation.AuthenticationPrincipal;

// ...

@PutMapping("/users/self")
public ModelAndView updateName(@AuthenticationPrincipal(expression = "@jpaEntityManager.merge(#this)") CustomUser attachedCustomUser,
    @RequestParam String firstName) {

    // change the firstName on an attached instance which will be persisted to the database
    attachedCustomUser.setFirstName(firstName);

    // ...
}
```

我们可以通过`@AuthenticationPrincipal`在我们自己的注释上进行元注释来进一步消除对Spring Security的依赖。下面我们演示如何在名为`@CurrentUser`的注释上做到这`@CurrentUser`。



认识到了消除对Spring Security的依赖是非常重要的，消费应用程序将创建`@CurrentUser`。这一步并非严格要求，但有助于将您对Spring Security的依赖隔离到更加中心的位置。


```
@Target({ElementType.PARAMETER, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@AuthenticationPrincipal
public @interface CurrentUser {}
```

现在已经指定了 `@CurrentUser`，我们可以使用它来发信号来解析当前经过身份验证的用户的 `CustomUser`。我们还将Spring Security的依赖性分离为单个文件。

```
@RequestMapping("/messages/inbox")
public ModelAndView findMessagesForUser(@CurrentUser CustomUser customUser) {

    // .. find messages for this user and return them ...
}
```

39.4 Spring MVC Async Integration 译: 39.4 Spring MVC异步集成

Spring Web MVC框架3.2+有良好的支持[Asynchronous Request Processing](#)。在没有额外配置的情况下，Spring Security会自动将 `SecurityContext` 设置为 `Thread`，执行控制器返回的 `Callable`。例如，下面的方法将自动拥有其 `Callable` 与执行的 `SecurityContext` 的时可用 `Callable` 创建：

```
@RequestMapping(method=RequestMethod.POST)
public Callable<String> processUpload(final MultipartFile file) {

    return new Callable<String>() {
        public Object call() throws Exception {
            // ...
            return "someView";
        }
    };
}
```



从技术上讲，Spring Security与 `WebAsyncManager` 集成。该 `SecurityContext`，其用于处理所述 `Callable` 是 `SecurityContext`，关于存在 `SecurityContextHolder` 在时间 `startCallableProcessing` 被调用。

没有与由控制器返回的 `DeferredResult` 自动集成。这是因为 `DeferredResult` 由用户处理，因此无法自动与它集成。但是，您仍然可以使用 [Concurrency Support](#) 来提供与Spring Security的透明集成。

39.5 Spring MVC and CSRF Integration 译: 39.5 Spring MVC和CSRF集成

39.5.1 Automatic Token Inclusion 译: 39.5.1 自动令牌包含

Spring Security将在使用 [Spring MVC form tag](#) 的表单中自动生成 `include the CSRF Token`。例如，以下JSP：

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
  xmlns:c="http://java.sun.com/jsp/jstl/core"
  xmlns:form="http://www.springframework.org/tags/form" version="2.0">
  <jsp:directive.page language="java" contentType="text/html" />
  <html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">
    <!-- ... -->

    <c:url var="logoutUrl" value="/logout"/>
    <form:form action="${logoutUrl}"
      method="post">
      <input type="submit"
        value="Log out" />
      <input type="hidden"
        name="${_csrf.parameterName}"
        value="${_csrf.token}"/>
    </form:form>

    <!-- ... -->
  </html>
</jsp:root>
```

将输出类似于以下内容的HTML：

```
<!-- ... -->

<form action="/context/logout" method="post">
<input type="submit" value="Log out"/>
<input type="hidden" name="_csrf" value="f81d4fae-7dec-11d0-a765-00a0c91e6bf6"/>
</form>

<!-- ... -->
```

39.5.2 Resolving the CsrfToken 译: 39.5.2 解析CsrfToken

Spring Security提供了 `CsrfTokenArgumentResolver`，它可以自动解析Spring MVC参数的当前 `CsrfToken`。通过使用 `@EnableWebSecurity`，您将自动将它添加到您的Spring MVC配置中。如果您使用基于XML的配置，您必须自己添加它。

一旦 `CsrfTokenArgumentResolver` 正确配置，可以暴露 `CsrfToken` 给你的静态HTML基础的应用。

```
@RestController
public class CsrfController {

    @RequestMapping("/csrf")
    public CsrfToken csrf(CsrfToken token) {
        return token;
    }
}
```

保持 `CsrfToken` 对其他领域的秘密 `CsrfToken` 。这意味着如果你使用的是 [Cross Origin Sharing \(CORS\)](#) ，你不应该暴露 `CsrfToken` 任何外部域。

Part VII. Spring Data Integration 译:第七部分。Spring数据集成

Spring Security提供了Spring Data集成，允许在查询中引用当前用户。这不仅有用于，而且必须将用户包含在查询中以支持分页结果，因为之后过滤结果不会扩展。

40. Spring Data & Spring Security Configuration 译:40.Spring Data& Spring安全配置

要使用此支持，请提供类型为 `SecurityEvaluationContextExtension` 的bean。在Java配置中，这看起来像：

```
@Bean
public SecurityEvaluationContextExtension securityEvaluationContextExtension() {
    return new SecurityEvaluationContextExtension();
}
```

在XML配置中，这看起来像：

```
<bean class="org.springframework.security.data.repository.query.SecurityEvaluationContextExtension"/>
```

41. Security Expressions within @Query 译:41.@Query的安全表达式

现在Spring Security可以用在您的查询中。例如：

```
@Repository
public interface MessageRepository extends PagingAndSortingRepository<Message,Long> {
    @Query("select m from Message m where m.to.id = ?#{ principal?.id }")
    Page<Message> findInbox(Pageable pageable);
}
```

它检查是否 `Authentication.getPrincipal().getId()` 等于的收件人 `Message` 。请注意，此示例假定您已将主体自定义为具有id属性的对象。通过公开 `SecurityEvaluationContextExtension` bean，所有 [Common Security Expressions](#) 在查询中都可用。

Part VIII. Appendix 译:第八部分。附录

42. Security Database Schema 译:42安全数据库模式

框架使用了各种数据库模式，本附录为他们提供了一个单一的参考点。您只需提供所需功能区域的表格。

为HSQldb数据库提供DDL语句。您可以将这些用作定义您正在使用的数据库的模式指南。

42.1 User Schema 译:42用户模式

`UserDetailsService` （ `JdbcDaoImpl` ）的标准JDBC实现需要表为用户加载密码，帐户状态（启用或禁用）以及权限（角色）列表。您将需要调整此架构以匹配您正在使用的数据库方言。

```
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);
```

42.1.1 Group Authorities 译:42.1.1组当局

Spring Security 2.0在 `JdbcDaoImpl` 引入了对组权限的 `JdbcDaoImpl` 。如果组已启用，则表结构如下。您将需要调整此架构以匹配您正在使用的数据库方言。

```

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)
);

```

请记住，只有在使用提供的JDBC `UserDetailsService` 实现时才需要这些表。如果您自己编写或选择实现 `AuthenticationProvider` 而不使用 `UserDetailsService`，那么只要接口合同得到满足，就可以完全自由地存储数据。

42.2 Persistent Login (Remember-Me) Schema 译: 42.2持久登录（记住我）架构

此表用于存储更安全的使用数据 `persistent token` 记得-me实现。如果您直接或通过名称空间使用 `JdbcTokenRepositoryImpl`，则需要此表。请记住调整此架构以匹配您正在使用的数据库方言。

```

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

```

42.3 ACL Schema 译: 42.3 ACL架构

Spring Security `ACL` 实现使用了四个表。

1. `acl_sid` stores the security identities recognised by the ACL system. These can be unique principals or authorities which may apply to multiple principals.
2. `acl_class` defines the domain object types to which ACLs apply. The `class` column stores the Java class name of the object.
3. `acl_object_identity` stores the object identity definitions of specific domain objects.
4. `acl_entry` stores the ACL permissions which apply to a specific object identity and security identity.

假定数据库将自动生成每个身份的主键。 `JdbcMutableAclService` 必须能够在 `acl_sid` 或 `acl_class` 表中创建新行时检索这些行。它有两个属性，它们定义了检索这些值 `classIdentityQuery` 和 `sidIdentityQuery` 所需的SQL。这两个都默认为 `call identity()`

ACL工件JAR包含用于在HyperSQL（HSQLDB），PostgreSQL，MySQL / MariaDB，Microsoft SQL Server和Oracle数据库中创建ACL模式的文件。这些模式也在以下部分中进行了演示。

42.3.1 HyperSQL 译: 42.3.1 HyperSQL

缺省模式与框架内单元测试中使用的嵌入式HSQLDB数据库一起使用。

```

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 varchar_ignorecase(36) 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)
);

```

42.3.2 PostgreSQL ⓘ : 42.3.2 PostgreSQL

```

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

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

create table acl_object_identity(
  id bigserial primary key,
  object_id_class bigint not null,
  object_id_identity varchar(36) not null,
  parent_object bigint,
  owner_sid bigint,
  entries_inheriting boolean not null,
  constraint unique_uk_3 unique(object_id_class,object_id_identity),
  constraint foreign_fk_1 foreign key(parent_object)references acl_object_identity(id),
  constraint foreign_fk_2 foreign key(object_id_class)references acl_class(id),
  constraint foreign_fk_3 foreign key(owner_sid)references acl_sid(id)
);

create table acl_entry(
  id bigserial primary key,
  acl_object_identity bigint not null,
  ace_order int not null,
  sid bigint not null,
  mask integer not null,
  granting boolean not null,
  audit_success boolean not null,
  audit_failure boolean not null,
  constraint unique_uk_4 unique(acl_object_identity,ace_order),
  constraint foreign_fk_4 foreign key(acl_object_identity) references acl_object_identity(id),
  constraint foreign_fk_5 foreign key(sid) references acl_sid(id)
);

```

您必须将 `classIdentityQuery` 和 `sidIdentityQuery` 属性 `JdbcMutableAclService` 分别设置为以下值:

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

42.3.3 MySQL and MariaDB ⓘ : 42.3.3 MySQL# MariaDB

```

CREATE TABLE acl_sid (
  id BIGINT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
  principal BOOLEAN NOT NULL,
  sid VARCHAR(100) NOT NULL,
  UNIQUE KEY unique_acl_sid (sid, principal)
) ENGINE=InnoDB;

CREATE TABLE acl_class (
  id BIGINT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
  class VARCHAR(100) NOT NULL,
  UNIQUE KEY uk_acl_class (class)
) ENGINE=InnoDB;

CREATE TABLE acl_object_identity (
  id BIGINT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
  object_id_class BIGINT UNSIGNED NOT NULL,
  object_id_identity VARCHAR(36) NOT NULL,
  parent_object BIGINT UNSIGNED,
  owner_sid BIGINT UNSIGNED,
  entries_inheriting BOOLEAN NOT NULL,
  UNIQUE KEY uk_acl_object_identity (object_id_class, object_id_identity),
  CONSTRAINT fk_acl_object_identity_parent FOREIGN KEY (parent_object) REFERENCES acl_object_identity (id),
  CONSTRAINT fk_acl_object_identity_class FOREIGN KEY (object_id_class) REFERENCES acl_class (id),
  CONSTRAINT fk_acl_object_identity_owner FOREIGN KEY (owner_sid) REFERENCES acl_sid (id)
) ENGINE=InnoDB;

CREATE TABLE acl_entry (
  id BIGINT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
  acl_object_identity BIGINT UNSIGNED NOT NULL,
  ace_order INTEGER NOT NULL,
  sid BIGINT UNSIGNED NOT NULL,
  mask INTEGER UNSIGNED NOT NULL,
  granting BOOLEAN NOT NULL,
  audit_success BOOLEAN NOT NULL,
  audit_failure BOOLEAN NOT NULL,
  UNIQUE KEY unique_acl_entry (acl_object_identity, ace_order),
  CONSTRAINT fk_acl_entry_object FOREIGN KEY (acl_object_identity) REFERENCES acl_object_identity (id),
  CONSTRAINT fk_acl_entry_acl FOREIGN KEY (sid) REFERENCES acl_sid (id)
) ENGINE=InnoDB;

```

42.3.4 Microsoft SQL Server 译：42.3.4 Microsoft SQL Server

```

CREATE TABLE acl_sid (
  id BIGINT NOT NULL IDENTITY PRIMARY KEY,
  principal BIT NOT NULL,
  sid VARCHAR(100) NOT NULL,
  CONSTRAINT unique_acl_sid UNIQUE (sid, principal)
);

CREATE TABLE acl_class (
  id BIGINT NOT NULL IDENTITY PRIMARY KEY,
  class VARCHAR(100) NOT NULL,
  CONSTRAINT uk_acl_class UNIQUE (class)
);

CREATE TABLE acl_object_identity (
  id BIGINT NOT NULL IDENTITY PRIMARY KEY,
  object_id_class BIGINT NOT NULL,
  object_id_identity VARCHAR(36) NOT NULL,
  parent_object BIGINT,
  owner_sid BIGINT,
  entries_inheriting BIT NOT NULL,
  CONSTRAINT uk_acl_object_identity UNIQUE (object_id_class, object_id_identity),
  CONSTRAINT fk_acl_object_identity_parent FOREIGN KEY (parent_object) REFERENCES acl_object_identity (id),
  CONSTRAINT fk_acl_object_identity_class FOREIGN KEY (object_id_class) REFERENCES acl_class (id),
  CONSTRAINT fk_acl_object_identity_owner FOREIGN KEY (owner_sid) REFERENCES acl_sid (id)
);

CREATE TABLE acl_entry (
  id BIGINT NOT NULL IDENTITY PRIMARY KEY,
  acl_object_identity BIGINT NOT NULL,
  ace_order INTEGER NOT NULL,
  sid BIGINT NOT NULL,
  mask INTEGER NOT NULL,
  granting BIT NOT NULL,
  audit_success BIT NOT NULL,
  audit_failure BIT NOT NULL,
  CONSTRAINT unique_acl_entry UNIQUE (acl_object_identity, ace_order),
  CONSTRAINT fk_acl_entry_object FOREIGN KEY (acl_object_identity) REFERENCES acl_object_identity (id),
  CONSTRAINT fk_acl_entry_acl FOREIGN KEY (sid) REFERENCES acl_sid (id)
);

```

42.3.5 Oracle Database 译：42.3.5 Oracle数据库

```

CREATE TABLE acl_sid (
  id NUMBER(38) NOT NULL PRIMARY KEY,
  principal NUMBER(1) NOT NULL CHECK (principal in (0, 1)),
  sid NVARCHAR2(100) NOT NULL,
  CONSTRAINT unique_acl_sid UNIQUE (sid, principal)
);
CREATE SEQUENCE acl_sid_sequence START WITH 1 INCREMENT BY 1 NOMAXVALUE;
CREATE OR REPLACE TRIGGER acl_sid_id_trigger
  BEFORE INSERT ON acl_sid
  FOR EACH ROW
BEGIN
  SELECT acl_sid_sequence.nextval INTO :new.id FROM dual;
END;

CREATE TABLE acl_class (
  id NUMBER(38) NOT NULL PRIMARY KEY,
  class NVARCHAR2(100) NOT NULL,
  CONSTRAINT uk_acl_class UNIQUE (class)
);
CREATE SEQUENCE acl_class_sequence START WITH 1 INCREMENT BY 1 NOMAXVALUE;
CREATE OR REPLACE TRIGGER acl_class_id_trigger
  BEFORE INSERT ON acl_class
  FOR EACH ROW
BEGIN
  SELECT acl_class_sequence.nextval INTO :new.id FROM dual;
END;

CREATE TABLE acl_object_identity (
  id NUMBER(38) NOT NULL PRIMARY KEY,
  object_id_class NUMBER(38) NOT NULL,
  object_id_identity NVARCHAR2(36) NOT NULL,
  parent_object NUMBER(38),
  owner_sid NUMBER(38),
  entries_inheriting NUMBER(1) NOT NULL CHECK (entries_inheriting in (0, 1)),
  CONSTRAINT uk_acl_object_identity UNIQUE (object_id_class, object_id_identity),
  CONSTRAINT fk_acl_object_identity_parent FOREIGN KEY (parent_object) REFERENCES acl_object_identity (id),
  CONSTRAINT fk_acl_object_identity_class FOREIGN KEY (object_id_class) REFERENCES acl_class (id),
  CONSTRAINT fk_acl_object_identity_owner FOREIGN KEY (owner_sid) REFERENCES acl_sid (id)
);
CREATE SEQUENCE acl_object_identity_sequence START WITH 1 INCREMENT BY 1 NOMAXVALUE;
CREATE OR REPLACE TRIGGER acl_object_identity_id_trigger
  BEFORE INSERT ON acl_object_identity
  FOR EACH ROW
BEGIN
  SELECT acl_object_identity_sequence.nextval INTO :new.id FROM dual;
END;

CREATE TABLE acl_entry (
  id NUMBER(38) NOT NULL PRIMARY KEY,
  acl_object_identity NUMBER(38) NOT NULL,
  ace_order INTEGER NOT NULL,
  sid NUMBER(38) NOT NULL,
  mask INTEGER NOT NULL,
  granting NUMBER(1) NOT NULL CHECK (granting in (0, 1)),
  audit_success NUMBER(1) NOT NULL CHECK (audit_success in (0, 1)),
  audit_failure NUMBER(1) NOT NULL CHECK (audit_failure in (0, 1)),
  CONSTRAINT unique_acl_entry UNIQUE (acl_object_identity, ace_order),
  CONSTRAINT fk_acl_entry_object FOREIGN KEY (acl_object_identity) REFERENCES acl_object_identity (id),
  CONSTRAINT fk_acl_entry_acl FOREIGN KEY (sid) REFERENCES acl_sid (id)
);
CREATE SEQUENCE acl_entry_sequence START WITH 1 INCREMENT BY 1 NOMAXVALUE;
CREATE OR REPLACE TRIGGER acl_entry_id_trigger
  BEFORE INSERT ON acl_entry
  FOR EACH ROW
BEGIN
  SELECT acl_entry_sequence.nextval INTO :new.id FROM dual;
END;

```

43. The Security Namespace 译: 43安全命名空间

本附录提供了对安全性命名空间中可用元素的引用以及他们创建的底层bean的信息（假定了解各个类以及它们如何一起工作 - 您可以在项目Javadoc和本文档中的其他地方找到更多信息）。如果您之前没有使用名称空间，请阅读有关名称空间配置的[introductory chapter](#)，因为这是对该信息的补充。建议在编辑基于模式的配置时使用高质量的XML编辑器，因为这将提供关于哪些元素和属性可用的上下文信息以及解释其用途的注释。命名空间是用[RELAX NG](#)压缩格式编写的，后来转换为XSD模式。如果您熟悉这种格式，您可能希望直接查看[schema file](#)。

43.1 Web Application Security 译: 43.1 Web应用程序安全性

43.1.1 <debug> 译: 43.1.1 <debug>

启用Spring Security调试基础架构。这将提供可读的（多行）调试信息来监视进入安全过滤器的请求。这可能包括敏感信息，如请求参数或标题，并且只能在开发环境中使用。

43.1.2 <http> 译: 43.1.2 <http>

如果在应用程序中使用<http>元素，则会创建名为“springSecurityFilterChain”的[FilterChainProxy](#) bean，并使用该元素中的配置在[FilterChainProxy](#)内构建过滤器链。从Spring Security 3.1起，可以使用额外的[http](#)元素来添加额外的过滤器链^[22]。一些核心过滤器总是在过滤器链中创建，而其他核心过滤器将根据存在的属性和子元素添加到堆栈中。标准过滤器的位置是固定的（参见命名空间简介中的[the filter order table](#)），当用户必须在[FilterChainProxy](#) bean中明确配置过滤器链时，可以消除以前版本框架中常见的错误来源。当然，如果您需要完全控制配置，您仍然可以这样做。

所有需要引用 `AuthenticationManager` 过滤器将自动注入由命名空间配置创建的内部实例（有关 `AuthenticationManager` 的更多信息，请参阅 `AuthenticationManager`）。

每个 `<http>` 名称空间块始终创建一个 `SecurityContextPersistenceFilter`，一个 `ExceptionTranslationFilter` 和一个 `FilterSecurityInterceptor`。这些是固定的，不能用替代品替代。

<http> Attributes 译: <http>属性

`<http>` 元素上的属性控制核心过滤器上的一些属性。

- **access-decision-manager-ref** Optional attribute specifying the ID of the `AccessDecisionManager` implementation which should be used for authorizing HTTP requests. By default an `AffirmativeBased` implementation is used for with a `RoleVoter` and an `AuthenticatedVoter`.
- **authentication-manager-ref** A reference to the `AuthenticationManager` used for the `FilterChain` created by this http element.
- **auto-config** Automatically registers a login form, BASIC authentication, logout services. If set to "true", all of these capabilities are added (although you can still customize the configuration of each by providing the respective element). If unspecified, defaults to "false". Use of this attribute is not recommended. Use explicit configuration elements instead to avoid confusion.
- **create-session** 控制Spring安全类创建HTTP会话的热切程度。选项包括：
 - `always` - Spring Security will proactively create a session if one does not exist.
 - `ifRequired` - Spring Security will only create a session only if one is required (default value).
 - `never` - Spring Security will never create a session, but will make use of one if the application does.
 - `stateless` - Spring Security will not create a session and ignore the session for obtaining a Spring `Authentication`.
- **disable-url-rewriting** Prevents session IDs from being appended to URLs in the application. Clients must use cookies if this attribute is set to `true`. The default is `true`.
- **entry-point-ref** Normally the `AuthenticationEntryPoint` used will be set depending on which authentication mechanisms have been configured. This attribute allows this behaviour to be overridden by defining a customized `AuthenticationEntryPoint` bean which will start the authentication process.
- **jaas-api-provision** If available, runs the request as the `Subject` acquired from the `JaasAuthenticationToken` which is implemented by adding a `JaasApiIntegrationFilter` bean to the stack. Defaults to `false`.
- **name** A bean identifier, used for referring to the bean elsewhere in the context.
- **once-per-request** Corresponds to the `observeOncePerRequest` property of `FilterSecurityInterceptor`. Defaults to `true`.
- **pattern** Defining a pattern for the `http` element controls the requests which will be filtered through the list of filters which it defines. The interpretation is dependent on the configured `request-matcher`. If no pattern is defined, all requests will be matched, so the most specific patterns should be declared first.
- **realm** Sets the realm name used for basic authentication (if enabled). Corresponds to the `realmName` property on `BasicAuthenticationEntryPoint`.
- **request-matcher** Defines the `RequestMatcher` strategy used in the `FilterChainProxy` and the beans created by the `intercept-url` to match incoming requests. Options are currently `mvc`, `ant`, `regex` and `ciRegex`, for Spring MVC, ant, regular-expression and case-insensitive regular-expression respectively. A separate instance is created for each `intercept-url` element using its `pattern`, `method` and `servlet-path` attributes. Ant paths are matched using an `AntPathRequestMatcher`, regular expressions are matched using a `RegexRequestMatcher` and for Spring MVC path matching the `MvcRequestMatcher` is used. See the Javadoc for these classes for more details on exactly how the matching is performed. Ant paths are the default strategy.
- **request-matcher-ref** A reference to a bean that implements `RequestMatcher` that will determine if this `FilterChain` should be used. This is a more powerful alternative to `pattern`.
- **security** A request pattern can be mapped to an empty filter chain, by setting this attribute to `none`. No security will be applied and none of Spring Security's features will be available.
- **security-context-repository-ref** Allows injection of a custom `SecurityContextRepository` into the `SecurityContextPersistenceFilter`.
- **servlet-api-provision** Provides versions of `HttpServletRequest` security methods such as `isUserInRole()` and `getPrincipal()` which are implemented by adding a `SecurityContextHolderAwareRequestFilter` bean to the stack. Defaults to `true`.
- **use-expressions** Enables EL-expressions in the `access` attribute, as described in the chapter on [expression-based access-control](#). The default value is true.

Child Elements of <http> 译: <http>的子元素

- [access-denied-handler](#)
- [anonymous](#)
- [cors](#)
- [csrf](#)
- [custom-filter](#)
- [expression-handler](#)
- [form-login](#)
- [headers](#)
- [http-basic](#)
- [intercept-url](#)
- [jee](#)
- [logout](#)
- [openid-login](#)
- [port-mappings](#)
- [remember-me](#)
- [request-cache](#)
- [session-management](#)
- [x509](#)

43.1.3 <access-denied-handler> 译: 43.1.3 <access-denied-handler>

这个元素允许您设置 `errorPage` 属性的默认 `AccessDeniedHandler` 通过使用 `ExceptionTranslationFilter`，使用 `error-page` 属性，或者使用提供自己的实现 `ref` 属性。这在 `ExceptionTranslationFilter` 的章节中有更详细的讨论。

Parent Elements of <access-denied-handler> 译: <access-denied-handler>的父元素

- [http](#)

<access-denied-handler> Attributes 译: <access-denied-handler>属性

- **error-page** The access denied page that an authenticated user will be redirected to if they request a page which they don't have the authority to access.
- **ref** Defines a reference to a Spring bean of type `AccessDeniedHandler`.

43.1.4 <cors> 译: 43.1.4 <cors>

该元素允许配置 `CorsFilter`。如果未指定 `CorsFilter` 或 `CorsConfigurationSource`，并且Spring MVC位于类路径中，则 `HandlerMappingIntrospector` 将用作 `CorsConfigurationSource`。

<cors> Attributes 译: <cors>属性

`<cors>` 元素上的属性控制标题元素。

- **ref** Optional attribute that specifies the bean name of a `CorsFilter`.
- **cors-configuration-source-ref** Optional attribute that specifies the bean name of a `CorsConfigurationSource` to be injected into a `CorsFilter` created by the XML namespace.

Parent Elements of <cors> 译: <cors>的父元素

- [http](#)

43.1.5 <headers> 译: 43.1.5 <headers>

该元素允许配置附加（安全）头与响应一起发送。它可以轻松配置多个头文件，并允许通过 `header` 元素设置自定义头文件。有关其他信息，请参阅参考文献的 [Security Headers](#) 部分。

- `Cache-Control`, `Pragma`, and `Expires` - Can be set using the `cache-control` element. This ensures that the browser does not cache your secured pages.
- `Strict-Transport-Security` - Can be set using the `hsts` element. This ensures that the browser automatically requests HTTPS for future requests.
- `X-Frame-Options` - Can be set using the `frame-options` element. The `X-Frame-Options` header can be used to prevent clickjacking attacks.
- `X-XSS-Protection` - Can be set using the `xss-protection` element. The `X-XSS-Protection` header can be used by browser to do basic control.
- `X-Content-Type-Options` - Can be set using the `content-type-options` element. The `X-Content-Type-Options` header prevents Internet Explorer from MIME-sniffing a response away from the declared content-type. This also applies to Google Chrome, when downloading extensions.
- `Public-Key-Pinning` or `Public-Key-Pinning-Report-Only` - Can be set using the `hpkp` element. This allows HTTPS websites to resist impersonation by attackers using mis-issued or otherwise fraudulent certificates.
- `Content-Security-Policy` or `Content-Security-Policy-Report-Only` - Can be set using the `content-security-policy` element. [Content Security Policy \(CSP\)](#) is a mechanism that web applications can leverage to mitigate content injection vulnerabilities, such as cross-site scripting (XSS).
- `Referrer-Policy` - Can be set using the `referrer-policy` element, [Referrer-Policy](#) is a mechanism that web applications can leverage to manage the referrer field, which contains the last page the user was on.

<headers> Attributes 译: <headers>属性

`<headers>` 元素上的属性控制标题元素。

- **defaults-disabled** Optional attribute that specifies to disable the default Spring Security's HTTP response headers. The default is false (the default headers are included).
- **disabled** Optional attribute that specifies to disable Spring Security's HTTP response headers. The default is false (the headers are enabled).

Parent Elements of <headers> 译: <headers>的父元素

- [http](#)

Child Elements of <headers> 译: <headers>的子元素

- [cache-control](#)
- [content-security-policy](#)
- [content-type-options](#)
- [frame-options](#)
- [header](#)
- [hpkp](#)
- [hsts](#)
- [referrer-policy](#)
- [xss-protection](#)

43.1.6 <cache-control> 译: 43.1.6 <缓存控制>

添加 `Cache-Control`，`Pragma`，并 `Expires` 头，以确保浏览器不缓存你的安全网页。

<cache-control> Attributes 译: <缓存控制>属性

- **disabled** Specifies if Cache Control should be disabled. Default false.

Parent Elements of <cache-control> 译: <cache-control>的父元素

- [headers](#)

43.1.7 <hsts> 译: 43.1.7 <hsts>

启用后，将[Strict-Transport-Security](#)标头添加到针对任何安全请求的响应中。这允许服务器指示浏览器为将来的请求自动使用HTTPS。

<hsts> Attributes 译: <hsts>属性

- **disabled** Specifies if Strict-Transport-Security should be disabled. Default false.
- **include-sub-domains** Specifies if subdomains should be included. Default true.
- **max-age-seconds** Specifies the maximum amount of time the host should be considered a Known HSTS Host. Default one year.
- **request-matcher-ref** The RequestMatcher instance to be used to determine if the header should be set. Default is if HttpServletRequest.isSecure() is true.

Parent Elements of <hsts> 译: <hsts>的父元素

- [headers](#)

43.1.8 <hpkp> 译: 43.1.8 <hpkp>

启用后，将[Public Key Pinning Extension for HTTP](#)标头添加到任何安全请求的响应中。这使得HTTPS网站能够抵御攻击者使用错误发布或其他欺诈性证书的冒充行为。

<hpkp> Attributes 译: <hpkp>属性

- **disabled** Specifies if HTTP Public Key Pinning (HPKP) should be disabled. Default true.
- **include-sub-domains** Specifies if subdomains should be included. Default false.
- **max-age-seconds** Sets the value for the max-age directive of the Public-Key-Pins header. Default 60 days.
- **report-only** Specifies if the browser should only report pin validation failures. Default true.
- **report-uri** Specifies the URI to which the browser should report pin validation failures.

Parent Elements of <hpkp> 译: <hpkp>的父元素

- [headers](#)

43.1.9 <pins> 译: 43.1.9 <pins>

引脚列表

Child Elements of <pins> 译: <pins>的子元素

- [pin](#)

43.1.10 <pin> 译: 43.1.10 <pin>

使用base64编码的SPK指纹作为值并将加密散列算法用作属性来指定引脚

<pin> Attributes 译: <pin>属性

- **algorithm** The cryptographic hash algorithm. Default is SHA256.

Parent Elements of <pin> 译: <pin>的父元素

- [pins](#)

43.1.11 <content-security-policy> 译: 43.1.11 <content-security-policy>

启用后，将[Content Security Policy \(CSP\)](#)标头添加到响应中。CSP是Web应用程序可以利用的机制来缓解内容注入漏洞，例如跨站点脚本（XSS）。

<content-security-policy> Attributes 译: <content-security-policy>属性

- **policy-directives** The security policy directive(s) for the Content-Security-Policy header or if report-only is set to true, then the Content-Security-Policy-Report-Only header is used.
- **report-only** Set to true, to enable the Content-Security-Policy-Report-Only header for reporting policy violations only. Defaults to false.

Parent Elements of <content-security-policy> 译: <content-security-policy>的父元素

- [headers](#)

43.1.12 <referrer-policy> 译: 43.1.12 <referrer-policy>

启用后，将 [Referrer Policy](#)标题添加到响应中。

<referrer-policy> Attributes 译: <referrer-policy>属性

- **policy** The policy for the Referrer-Policy header. Default "no-referrer".

Parent Elements of <referrer-policy> 译: <referrer-policy>的父元素

- [headers](#)

43.1.13 <frame-options> 译: 43.1.13 <frame-options>

启用后, 将 [X-Frame-Options header](#) 添加到响应中, 这允许较新的浏览器执行一些安全检查并防止 [clickjacking](#) 攻击。

<frame-options> Attributes 译: <frame-options>属性

- **disabled** If disabled, the X-Frame-Options header will not be included. Default false.
- **政策**
 - **DENY** The page cannot be displayed in a frame, regardless of the site attempting to do so. This is the default when frame-options-policy is specified.
 - **SAMEORIGIN** The page can only be displayed in a frame on the same origin as the page itself
 - **ALLOW-FROM origin** The page can only be displayed in a frame on the specified origin.换句话说, 如果你指定了DENY, 那么当从其他站点加载时, 不仅会尝试在一个框架中加载页面, 当从同一站点加载时, 尝试这样做将失败。另一方面, 如果指定了SAMEORIGIN, 只要网站将其包含在一个框架中, 就可以使用该框架中的页面, 即可与该页面提供的页面相同。
- **策略** 在使用ALLOW-FROM策略时, 请选择 **AllowFromStrategy** 。
 - **static** Use a single static ALLOW-FROM value. The value can be set through the [value](#) attribute.
 - **regexp** Use a regular expression to validate incoming requests and if they are allowed. The regular expression can be set through the [value](#) attribute. The request parameter used to retrieve the value to validate can be specified using the [from-parameter](#).
 - **whitelist** A comma-separated list containing the allowed domains. The comma-separated list can be set through the [value](#) attribute. The request parameter used to retrieve the value to validate can be specified using the [from-parameter](#).
- **ref** Instead of using one of the predefined strategies it is also possible to use a custom **AllowFromStrategy**. The reference to this bean can be specified through this ref attribute.
- **value** The value to use when ALLOW-FROM is used a [strategy](#).
- **from-parameter** Specify the name of the request parameter to use when using regexp or whitelist for the ALLOW-FROM strategy.

Parent Elements of <frame-options> 译: <frame-options>的父元素

- [headers](#)

43.1.14 <xss-protection> 译: 43.1.14 <xss-protection>

将 [X-XSS-Protection header](#) 添加到响应中, 以帮助防止 [reflected / Type-1 Cross-Site Scripting \(XSS\)](#) 攻击。这绝对不能完全保护XSS攻击!

<xss-protection> Attributes 译: <xss-protection>属性

- **xss-protection-disabled** Do not include the header for [reflected / Type-1 Cross-Site Scripting \(XSS\)](#) protection.
- **xss-protection-enabled** Explicitly enable or disable [reflected / Type-1 Cross-Site Scripting \(XSS\)](#) protection.
- **xss-protection-block** When true and xss-protection-enabled is true, adds mode=block to the header. This indicates to the browser that the page should not be loaded at all. When false and xss-protection-enabled is true, the page will still be rendered when a reflected attack is detected but the response will be modified to protect against the attack. Note that there are sometimes ways of bypassing this mode which can often times make blocking the page more desirable.

Parent Elements of <xss-protection> 译: <xss-protection>的父元素

- [headers](#)

43.1.15 <content-type-options> 译: 43.1.15 <content-type-options>

将值为nosniff的X-Content-Type-Options标头添加到响应中。这是用于IE8 +和Chrome扩展的[disables MIME-sniffing](#)。

<content-type-options> Attributes 译: <content-type-options>属性

- **disabled** Specifies if Content Type Options should be disabled. Default false.

Parent Elements of <content-type-options> 译: <content-type-options>的父元素

- [headers](#)

43.1.16 <header> 译: 43.1.16 <header>

将其他标题添加到响应中, 需要指定名称和值。

<header-attributes> Attributes 译: <header-attributes>属性

- **header-name** The **name** of the header.
- **value** The **value** of the header to add.
- **ref** Reference to a custom implementation of the **HeaderWriter** interface.

Parent Elements of <header> 译: <header>的父元素

- [headers](#)

43.1.17 <anonymous> 译: 43.1.17 <anonymous>

将 `AnonymousAuthenticationFilter` 添加到堆栈和 `AnonymousAuthenticationProvider`。如果您使用 `IS_AUTHENTICATED_ANONYMOUSLY` 属性，`IS_AUTHENTICATED_ANONYMOUSLY` 必需。

Parent Elements of <anonymous> 译: <anonymous> 的父元素

- [http](#)

<anonymous> Attributes 译: <anonymous> 属性

- **enabled** With the default namespace setup, the anonymous "authentication" facility is automatically enabled. You can disable it using this property.
- **granted-authority** The granted authority that should be assigned to the anonymous request. Commonly this is used to assign the anonymous request particular roles, which can subsequently be used in authorization decisions. If unset, defaults to `ROLE_ANONYMOUS`.
- **key** The key shared between the provider and filter. This generally does not need to be set. If unset, it will default to a secure randomly generated value. This means setting this value can improve startup time when using the anonymous functionality since secure random values can take a while to be generated.
- **username** The username that should be assigned to the anonymous request. This allows the principal to be identified, which may be important for logging and auditing. If unset, defaults to `anonymousUser`.

43.1.18 <csrf> 译: 43.1.18 <csrf>

该元素将为应用程序添加 [Cross Site Request Forger \(CSRF\)](#) 保护。它还将默认 RequestCache 更新为仅在成功验证时重播 "GET" 请求。其他信息可在参考文献的 [Cross Site Request Forgery \(CSRF\)](#) 部分找到。

Parent Elements of <csrf> 译: <csrf> 的父元素

- [http](#)

<csrf> Attributes 译: <csrf> 属性

- **disabled** Optional attribute that specifies to disable Spring Security's CSRF protection. The default is false (CSRF protection is enabled). It is highly recommended to leave CSRF protection enabled.
- **token-repository-ref** The CsrfTokenRepository to use. The default is `HttpSessionCsrfTokenRepository`.
- **request-matcher-ref** The RequestMatcher instance to be used to determine if CSRF should be applied. Default is any HTTP method except "GET", "TRACE", "HEAD", "OPTIONS".

43.1.19 <custom-filter> 译: 43.1.19 <custom-filter>

该元素用于向过滤器链添加过滤器。它不会创建任何额外的 bean，但用于选择已在应用程序上下文中定义的类型为 `javax.servlet.Filter` 的 bean，`javax.servlet.Filter` 其添加到由 Spring Security 维护的过滤器链中的特定位置。全部细节可以在 [namespace chapter](#) 找到。

Parent Elements of <custom-filter> 译: <custom-filter> 的父元素

- [http](#)

<custom-filter> Attributes 译: <custom-filter> 属性

- **after** The filter immediately after which the custom-filter should be placed in the chain. This feature will only be needed by advanced users who wish to mix their own filters into the security filter chain and have some knowledge of the standard Spring Security filters. The filter names map to specific Spring Security implementation filters.
- **before** The filter immediately before which the custom-filter should be placed in the chain
- **position** The explicit position at which the custom-filter should be placed in the chain. Use if you are replacing a standard filter.
- **ref** Defines a reference to a Spring bean that implements `Filter`.

43.1.20 <expression-handler> 译: 43.1.20 <expression-handler>

定义在启用基于表达式的访问控制时将使用的 `SecurityExpressionHandler` 实例。如果不提供，将使用默认实现（不支持 ACL）。

Parent Elements of <expression-handler> 译: <expression-handler> 的父元素

- [global-method-security](#)
- [http](#)
- [websocket-message-broker](#)

<expression-handler> Attributes 译: <表达式处理程序> 属性

- **ref** Defines a reference to a Spring bean that implements `SecurityExpressionHandler`.

43.1.21 <form-login> 译: 43.1.21 <form-login>

用于将 `UsernamePasswordAuthenticationFilter` 添加到过滤器堆栈，并将 `LoginUrlAuthenticationEntryPoint` 到应用程序上下文以根据需要提供身份验证。这始终优先于其他命名空间创建的入口点。如果未提供任何属性，则会在 URL "/login" ^[23] 处自动生成登录页面。可以使用 `<form-login>` [Attributes](#) 自定义该行为。

Parent Elements of <form-login> 译: <form-login> 的父元素

- [http](#)

<form-login> Attributes 译: <form-login>属性

- **always-use-default-target** If set to `true`, the user will always start at the value given by `default-target-url`, regardless of how they arrived at the login page. Maps to the `alwaysUseDefaultTargetUrl` property of `UsernamePasswordAuthenticationFilter`. Default value is `false`.
- **authentication-details-source-ref** Reference to an `AuthenticationDetailsSource` which will be used by the authentication filter
- **authentication-failure-handler-ref** Can be used as an alternative to `authentication-failure-url`, giving you full control over the navigation flow after an authentication failure. The value should be the name of an `AuthenticationFailureHandler` bean in the application context.
- **authentication-failure-url** Maps to the `authenticationFailureUrl` property of `UsernamePasswordAuthenticationFilter`. Defines the URL the browser will be redirected to on login failure. Defaults to `/login?error`, which will be automatically handled by the automatic login page generator, re-rendering the login page with an error message.
- **authentication-success-handler-ref** This can be used as an alternative to `default-target-url` and `always-use-default-target`, giving you full control over the navigation flow after a successful authentication. The value should be the name of an `AuthenticationSuccessHandler` bean in the application context. By default, an implementation of `SavedRequestAwareAuthenticationSuccessHandler` is used and injected with the `default-target-url`.
- **default-target-url** Maps to the `defaultTargetUrl` property of `UsernamePasswordAuthenticationFilter`. If not set, the default value is `/` (the application root). A user will be taken to this URL after logging in, provided they were not asked to login while attempting to access a secured resource, when they will be taken to the originally requested URL.
- **login-page** The URL that should be used to render the login page. Maps to the `loginFormUrl` property of the `LoginUrlAuthenticationEntryPoint`. Defaults to `/login`.
- **login-processing-url** Maps to the `filterProcessesUrl` property of `UsernamePasswordAuthenticationFilter`. The default value is `/login`.
- **password-parameter** The name of the request parameter which contains the password. Defaults to `password`.
- **username-parameter** The name of the request parameter which contains the username. Defaults to `username`.
- **authentication-success-forward-url** Maps a `ForwardAuthenticationSuccessHandler` to `authenticationSuccessHandler` property of `UsernamePasswordAuthenticationFilter`.
- **authentication-failure-forward-url** Maps a `ForwardAuthenticationFailureHandler` to `authenticationFailureHandler` property of `UsernamePasswordAuthenticationFilter`.

43.1.22 <http-basic> 译: 43.1.22 <http-basic>

将 `BasicAuthenticationFilter` 和 `BasicAuthenticationEntryPoint` 添加到配置中。如果未启用基于表单的登录, 后者将仅用作配置入口点。

Parent Elements of <http-basic> 译: <http-basic>的父元素

- [http](#)

<http-basic> Attributes 译: <http-basic>属性

- **authentication-details-source-ref** Reference to an `AuthenticationDetailsSource` which will be used by the authentication filter
- **entry-point-ref** Sets the `AuthenticationEntryPoint` which is used by the `BasicAuthenticationFilter`.

43.1.23 <http-firewall> Element 译: 43.1.23 <http-firewall>元素

这是一个顶级元素, 可用于将自定义实现 `HttpFirewall` 注入由名称空间创建的 `FilterChainProxy`。默认实现应该适用于大多数应用程序。

<http-firewall> Attributes 译: <http-firewall>属性

- **ref** Defines a reference to a Spring bean that implements `HttpFirewall`.

43.1.24 <intercept-url> 译: 43.1.24 <intercept-url>

此元素用于定义应用程序感兴趣的一组URL模式并配置应该如何处理它们。它用于构造 `FilterInvocationSecurityMetadataSource` 使用的 `FilterSecurityInterceptor`。 `ChannelProcessingFilter` 如果需要通过HTTPS访问特定的URL, 它还负责配置 `ChannelProcessingFilter`。当针对传入请求匹配指定模式时, 匹配按声明元素的顺序完成。所以最具体的模式应该是第一位的, 最普通的模式应该是最后一位。

Parent Elements of <intercept-url> 译: <拦截url>的父元素

- [filter-security-metadata-source](#)
- [http](#)

<intercept-url> Attributes 译: <intercept-url>属性

- **access** Lists the access attributes which will be stored in the `FilterInvocationSecurityMetadataSource` for the defined URL pattern/method combination. This should be a comma-separated list of the security configuration attributes (such as role names).
- **filters** Can only take the value `"none"`. This will cause any matching request to bypass the Spring Security filter chain entirely. None of the rest of the `<http>` configuration will have any effect on the request and there will be no security context available for its duration. Access to secured methods during the request will fail.



此属性对于 `filter-security-metadata-source` 无效

- **method** The HTTP Method which will be used in combination with the pattern and servlet path (optional) to match an incoming request. If omitted, any method will match. If an identical pattern is specified with and without a method, the method-specific match will take precedence.

- **pattern** The pattern which defines the URL path. The content will depend on the `request-matcher` attribute from the containing http element, so will default to ant path syntax.
- **request-matcher-ref** A reference to a `RequestMatcher` that will be used to determine if this `<intercept-url>` is used.
- **requires-channel** Can be "http" or "https" depending on whether a particular URL pattern should be accessed over HTTP or HTTPS respectively. Alternatively the value "any" can be used when there is no preference. If this attribute is present on any `<intercept-url>` element, then a `ChannelProcessingFilter` will be added to the filter stack and its additional dependencies added to the application context.

如果添加了 `<port-mappings>` 配置, 则 `SecureChannelProcessor` 和 `InsecureChannelProcessor` bean将使用此配置来确定用于重定向到HTTP / HTTPS的端口。



该属性对于 `filter-security-metadata-source` 无效

- **servlet-path** The servlet path which will be used in combination with the pattern and HTTP method to match an incoming request. This attribute is only applicable when `request-matcher` is 'mvc'. In addition, the value is only required in the following 2 use cases: 1) There are 2 or more `HttpServlet`'s registered in the `ServletContext` that have mappings starting with `'/'` and are different; 2) The pattern starts with the same value of a registered `HttpServlet` path, excluding the default (root) `HttpServlet` `'/'`.



此属性对于 `filter-security-metadata-source` 无效

43.1.25 <jee> 译: 43.1.25 <jee>

将J2eePreAuthenticatedProcessingFilter添加到过滤器链以提供与容器验证的集成。

Parent Elements of <jee> 译: <jee>的父元素

- [http](#)

<jee> Attributes 译: <jee>属性

- **mappable-roles** A comma-separated list of roles to look for in the incoming HttpServletRequest.
- **user-service-ref** A reference to a user-service (or UserDetailsService bean) id

43.1.26 <logout> 译: 43.1.26 <注销>

将LogoutFilter添加到过滤器堆栈。这配置了一个SecurityContextLogoutHandler。

Parent Elements of <logout> 译: <logout>的父元素

- [http](#)

<logout> Attributes 译: <logout>属性

- **delete-cookies** A comma-separated list of the names of cookies which should be deleted when the user logs out.
- **invalidate-session** Maps to the `invalidateHttpSession` of the `SecurityContextLogoutHandler`. Defaults to "true", so the session will be invalidated on logout.
- **logout-success-url** 注销后用户将被带到的目标URL。默认为<form-login-login-page> /? 注销 (即/login? 注销) 设置此属性将注入SessionManagementFilter, 其中SimpleRedirectInvalidSessionStrategy使用属性值进行配置。当提交无效的会话ID时, 该策略将被调用, 重定向到配置的URL。
- **logout-url** The URL which will cause a logout (i.e. which will be processed by the filter). Defaults to "/logout".
- **success-handler-ref** May be used to supply an instance of `LogoutSuccessHandler` which will be invoked to control the navigation after logging out.

43.1.27 <openid-login> 译: 43.1.27 <openid-login>

类似于<form-login>并具有相同的属性。`login-processing-url`的默认值是"/login/openid"。

OpenIDAuthenticationFilter和OpenIDAuthenticationProvider将被注册。后者要求参考UserDetailsService。同样, 这可以由id指定, 使用user-service-ref属性, 或者将自动定位在应用程序上下文中。

Parent Elements of <openid-login> 译: <openid-login>的父元素

- [http](#)

<openid-login> Attributes 译: <openid-login>属性

- **always-use-default-target** Whether the user should always be redirected to the default-target-url after login.
- **authentication-details-source-ref** Reference to an AuthenticationDetailsSource which will be used by the authentication filter
- **authentication-failure-handler-ref** Reference to an AuthenticationFailureHandler bean which should be used to handle a failed authentication request. Should not be used in combination with authentication-failure-url as the implementation should always deal with navigation to the subsequent destination
- **authentication-failure-url** The URL for the login failure page. If no login failure URL is specified, Spring Security will automatically create a failure login URL at /login?login_error and a corresponding filter to render that login failure URL when requested.
- **authentication-success-forward-url** Maps a `ForwardAuthenticationSuccessHandler` to `authenticationSuccessHandler` property of `UsernamePasswordAuthenticationFilter`.

- **authentication-failure-forward-url** Maps a `ForwardAuthenticationFailureHandler` to `authenticationFailureHandler` property of `UsernamePasswordAuthenticationFilter`.
- **authentication-success-handler-ref** Reference to an `AuthenticationSuccessHandler` bean which should be used to handle a successful authentication request. Should not be used in combination with `default-target-url` (or `always-use-default-target`) as the implementation should always deal with navigation to the subsequent destination
- **default-target-url** The URL that will be redirected to after successful authentication, if the user's previous action could not be resumed. This generally happens if the user visits a login page without having first requested a secured operation that triggers authentication. If unspecified, defaults to the root of the application.
- **login-page** The URL for the login page. If no login URL is specified, Spring Security will automatically create a login URL at `/login` and a corresponding filter to render that login URL when requested.
- **login-processing-url** The URL that the login form is posted to. If unspecified, it defaults to `/login`.
- **password-parameter** The name of the request parameter which contains the password. Defaults to "password".
- **user-service-ref** A reference to a user-service (or `UserDetailsService` bean) id
- **username-parameter** The name of the request parameter which contains the username. Defaults to "username".

Child Elements of <openid-login> 译: <openid-login>的子元素

- [attribute-exchange](#)

43.1.28 <attribute-exchange> 译: 43.1.28 <属性交换>

`attribute-exchange` 元素定义了应该从身份提供者请求的属性列表。可以在命名空间配置章节的 [OpenID Support](#) 部分找到一个示例。可以使用多于一个, 在这种情况下, 每个属性必须具有 `identifier-match` 属性, 其中包含与提供的 OpenID 标识符相匹配的正则表达式。这允许从不同的提供者 (谷歌, 雅虎等) 获取不同的属性列表。

Parent Elements of <attribute-exchange> 译: <attribute-exchange>的父元素

- [openid-login](#)

<attribute-exchange> Attributes 译: <attribute-exchange>属性

- **identifier-match** A regular expression which will be compared against the claimed identity, when deciding which attribute-exchange configuration to use during authentication.

Child Elements of <attribute-exchange> 译: <attribute-exchange>的子元素

- [openid-attribute](#)

43.1.29 <openid-attribute> 译: 43.1.29 <openid-attribute>

制作 OpenID AX [Fetch Request](#) 时使用的属性

Parent Elements of <openid-attribute> 译: <openid-attribute>的父元素

- [attribute-exchange](#)

<openid-attribute> Attributes 译: <openid-attribute>属性

- **count** Specifies the number of attributes that you wish to get back. For example, return 3 emails. The default value is 1.
- **name** Specifies the name of the attribute that you wish to get back. For example, email.
- **required** Specifies if this attribute is required to the OP, but does not error out if the OP does not return the attribute. Default is false.
- **type** Specifies the attribute type. For example, <http://axschema.org/contact/email>. See your OP's documentation for valid attribute types.

43.1.30 <port-mappings> 译: 43.1.30 <port-mappings>

默认情况下, `PortMapperImpl` 的实例将被添加到配置中, 以用于重定向到安全和不安全的URL。此元素可以选择用于覆盖该类定义的默认映射。每个子元素 `<port-mapping>` 定义了一对 HTTP: HTTPS 端口。默认映射是 80: 443 和 8080: 8443。 [namespace introduction](#) 中有一个覆盖这些的例子。

Parent Elements of <port-mappings> 译: <port-mappings>的父元素

- [http](#)

Child Elements of <port-mappings> 译: <port-mappings>的子元素

- [port-mapping](#)

43.1.31 <port-mapping> 译: 43.1.31 <端口映射>

提供强制重定向时将 http 端口映射到 https 端口的的方法。

Parent Elements of <port-mapping> 译: <port-mapping>的父元素

- [port-mappings](#)

<port-mapping> Attributes 译: <port-mapping>属性

- **http** The http port to use.
- **https** The https port to use.

43.1.32 <remember-me> 译: <记住我>

将 `RememberMeAuthenticationFilter` 添加到堆栈。反过来, 这将被配置或者是 `TokenBasedRememberMeServices`, 一个 `PersistentTokenBasedRememberMeServices` 或用户自定义的实现 `RememberMeServices` 根据属性设置。

Parent Elements of <remember-me> 译: <记住我>的父元素

- [http](#)

<remember-me> Attributes 译: <remember-me>属性

- **authentication-success-handler-ref** Sets the `authenticationSuccessHandler` property on the `RememberMeAuthenticationFilter` if custom navigation is required. The value should be the name of a `AuthenticationSuccessHandler` bean in the application context.
- **data-source-ref** A reference to a `DataSource` bean. If this is set, `PersistentTokenBasedRememberMeServices` will be used and configured with a `JdbcTokenRepositoryImpl` instance.
- **remember-me-parameter** The name of the request parameter which toggles remember-me authentication. Defaults to "remember-me". Maps to the "parameter" property of `AbstractRememberMeServices`.
- **remember-me-cookie** The name of cookie which store the token for remember-me authentication. Defaults to "remember-me". Maps to the "cookieName" property of `AbstractRememberMeServices`.
- **key** Maps to the "key" property of `AbstractRememberMeServices`. Should be set to a unique value to ensure that remember-me cookies are only valid within the one application ^[24]. If this is not set a secure random value will be generated. Since generating secure random values can take a while, setting this value explicitly can help improve startup times when using the remember-me functionality.
- **services-alias** Exports the internally defined `RememberMeServices` as a bean alias, allowing it to be used by other beans in the application context.
- **services-ref** Allows complete control of the `RememberMeServices` implementation that will be used by the filter. The value should be the `id` of a bean in the application context which implements this interface. Should also implement `LogoutHandler` if a logout filter is in use.
- **token-repository-ref** Configures a `PersistentTokenBasedRememberMeServices` but allows the use of a custom `PersistentTokenRepository` bean.
- **token-validity-seconds** Maps to the `tokenValiditySeconds` property of `AbstractRememberMeServices`. Specifies the period in seconds for which the remember-me cookie should be valid. By default it will be valid for 14 days.
- **use-secure-cookie** It is recommended that remember-me cookies are only submitted over HTTPS and thus should be flagged as "secure". By default, a secure cookie will be used if the connection over which the login request is made is secure (as it should be). If you set this property to `false`, secure cookies will not be used. Setting it to `true` will always set the secure flag on the cookie. This attribute maps to the `useSecureCookie` property of `AbstractRememberMeServices`.
- **user-service-ref** The remember-me services implementations require access to a `UserDetailsService`, so there has to be one defined in the application context. If there is only one, it will be selected and used automatically by the namespace configuration. If there are multiple instances, you can specify a bean `id` explicitly using this attribute.

43.1.33 <request-cache> Element 译: <request-cache>元素

设置 `RequestCache` 实例, 在调用 `AuthenticationEntryPoint` 之前, `ExceptionTranslationFilter` 将使用该 `ExceptionTranslationFilter` 存储请求信息。

Parent Elements of <request-cache> 译: <request-cache>的父元素

- [http](#)

<request-cache> Attributes 译: <请求缓存>属性

- **ref** Defines a reference to a Spring bean that is a `RequestCache`.

43.1.34 <session-management> 译: <session-management>

会话管理相关的功能通过向过滤器堆栈添加 `SessionManagementFilter` 来实现。

Parent Elements of <session-management> 译: <session-management>的父元素

- [http](#)

<session-management> Attributes 译: <session-management>属性

- **invalid-session-url** Setting this attribute will inject the `SessionManagementFilter` with a `SimpleRedirectInvalidSessionStrategy` configured with the attribute value. When an invalid session ID is submitted, the strategy will be invoked, redirecting to the configured URL.
- **invalid-session-url** Allows injection of the `InvalidSessionStrategy` instance used by the `SessionManagementFilter`. Use either this or the `invalid-session-url` attribute but not both.
- **session-authentication-error-url** Defines the URL of the error page which should be shown when the `SessionAuthenticationStrategy` raises an exception. If not set, an unauthorized (401) error code will be returned to the client. Note that this attribute doesn't apply if the error occurs during a form-based login, where the URL for authentication failure will take precedence.
- **session-authentication-strategy-ref** Allows injection of the `SessionAuthenticationStrategy` instance used by the `SessionManagementFilter`
- **会话固定保护** 指示用户身份验证时如何应用会话固定保护。如果设置为“无”, 则不会应用保护。“newSession”将创建一个新的空会话, 只迁移与Spring Security相关的属性。“migrateSession”将创建一个新会话并将所有会话属性复制到新会话。在Servlet 3.1 (Java EE 7) 和更新的容器中, 指定“changeSessionId”将保留现有会话并

使用容器提供的会话固定保护（HttpServletRequest#changeSessionId（））。在Servlet 3.1和更新的容器中默认为“changeSessionId”，在较旧的容器中为“migrateSession”。如果在较旧的容器中使用“changeSessionId”，则会引发异常。

如果会话固定保护启用，`SessionManagementFilter` 注入适当配置的 `DefaultSessionAuthenticationStrategy`。有关更多详细信息，请参阅此类的 Javadoc。

Child Elements of <session-management> 译：<session-management>的子元素

- [concurrency-control](#)

43.1.35 <concurrency-control> 译：43.1.35 <concurrency-control>

增加对并发会话控制的支持，允许限制用户可以拥有的活动会话的数量。一个 `ConcurrentSessionFilter` 将被创建，并且 `ConcurrentSessionControlAuthenticationStrategy` 将与使用 `SessionManagementFilter`。如果已声明 `form-login` 元素，则策略对象也将被注入到创建的认证过滤器中。实例 `SessionRegistry`（一 `SessionRegistryImpl` 实例，除非用户希望使用自定义的bean）将通过策略而创建。

Parent Elements of <concurrency-control> 译：<concurrency-control>的父元素

- [session-management](#)

<concurrency-control> Attributes 译：<concurrency-control>属性

- error-if-maximum-exceeded** If set to "true" a `SessionAuthenticationException` will be raised when a user attempts to exceed the maximum allowed number of sessions. The default behaviour is to expire the original session.
- expired-url** The URL a user will be redirected to if they attempt to use a session which has been "expired" by the concurrent session controller because the user has exceeded the number of allowed sessions and has logged in again elsewhere. Should be set unless `exception-if-maximum-exceeded` is set. If no value is supplied, an expiry message will just be written directly back to the response.
- expired-url** Allows injection of the `ExpiredSessionStrategy` instance used by the `ConcurrentSessionFilter`
- max-sessions** Maps to the `maximumSessions` property of `ConcurrentSessionControlAuthenticationStrategy`. Specify `-1` as the value to support unlimited sessions.
- session-registry-alias** It can also be useful to have a reference to the internal session registry for use in your own beans or an admin interface. You can expose the internal bean using the `session-registry-alias` attribute, giving it a name that you can use elsewhere in your configuration.
- session-registry-ref** The user can supply their own `SessionRegistry` implementation using the `session-registry-ref` attribute. The other concurrent session control beans will be wired up to use it.

43.1.36 <x509> 译：43.1.36 <x509>

添加对X.509认证的支持。一个 `X509AuthenticationFilter` 将被添加到堆栈中，并且将创建一个 `Http403ForbiddenEntryPoint` bean。只有在没有使用其他身份验证机制时才会使用后者（它的唯一功能是返回HTTP 403错误代码）。还将创建一个 `PreAuthenticatedAuthenticationProvider` 它将用户权限的加载委托给 `UserDetailsService`。

Parent Elements of <x509> 译：<x509>的父元素

- [http](#)

<x509> Attributes 译：<x509>属性

- authentication-details-source-ref** A reference to an `AuthenticationDetailsSource`
- subject-principal-regex** Defines a regular expression which will be used to extract the username from the certificate (for use with the `UserDetailsService`).
- user-service-ref** Allows a specific `UserDetailsService` to be used with X.509 in the case where multiple instances are configured. If not set, an attempt will be made to locate a suitable instance automatically and use that.

43.1.37 <filter-chain-map> 译：43.1.37 <filter-chain-map>

用于使用FilterChainMap显式配置FilterChainProxy实例

<filter-chain-map> Attributes 译：<filter-chain-map>属性

- request-matcher** Defines the strategy to use for matching incoming requests. Currently the options are 'ant' (for ant path patterns), 'regex' for regular expressions and 'ciRegex' for case-insensitive regular expressions.

Child Elements of <filter-chain-map> 译：<filter-chain-map>的子元素

- [filter-chain](#)

43.1.38 <filter-chain> 译：43.1.38 <filter-chain>

用于定义特定的URL模式以及适用于与该模式匹配的URL的过滤器列表。当为了配置FilterChainProxy而在列表中组装多个过滤器链元素时，最具体的模式必须放置在列表的顶部，最常见的模式位于最下面。

Parent Elements of <filter-chain> 译：<filter-chain>的父元素

- [filter-chain-map](#)

<filter-chain> Attributes 译：<filter-chain>属性

- **filters** A comma separated list of references to Spring beans that implement `Filter`. The value "none" means that no `Filter` should be used for this `FilterChain`.
- **pattern** A pattern that creates RequestMatcher in combination with the `request-matcher`
- **request-matcher-ref** A reference to a `RequestMatcher` that will be used to determine if any `Filter` from the `filters` attribute should be invoked.

43.1.39 <filter-security-metadata-source> 译: 43.1.39 <filter-security-metadata-source>

用于显式配置FilterSecurityMetadataSource bean以用于FilterSecurityInterceptor。通常只需要显式配置FilterChainProxy，而不是使用<http>元素。所使用的intercept-url元素应该只包含模式，方法和访问属性。任何其他将导致配置错误。

<filter-security-metadata-source> Attributes 译: <filter-security-metadata-source>属性

- **id** A bean identifier, used for referring to the bean elsewhere in the context.
- **request-matcher** Defines the strategy use for matching incoming requests. Currently the options are 'ant' (for ant path patterns), 'regex' for regular expressions and 'ciRegex' for case-insensitive regular expressions.
- **use-expressions** Enables the use of expressions in the 'access' attributes in <intercept-url> elements rather than the traditional list of configuration attributes. Defaults to 'true'. If enabled, each attribute should contain a single Boolean expression. If the expression evaluates to 'true', access will be granted.

Child Elements of <filter-security-metadata-source> 译: <filter-security-metadata-source>的子元素

- [intercept-url](#)

43.2 WebSocket Security 译: 43.2 WebSocket安全性

Spring Security 4.0+提供了对授权消息的支持。其中有用的一个具体示例是在基于WebSocket的应用程序中提供授权。

43.2.1 <websocket-message-broker> 译: 43.2.1 <websocket-message-broker>

websocket-message-broker元素有两种不同的模式。如果未指定(#)，则它将执行以下操作：

- Ensure that any SimpAnnotationMethodMessageHandler has the AuthenticationPrincipalArgumentResolver registered as a custom argument resolver. This allows the use of `@AuthenticationPrincipal` to resolve the principal of the current `Authentication`
- Ensures that the SecurityContextChannelInterceptor is automatically registered for the clientInboundChannel. This populates the SecurityContextHolder with the user that is found in the Message
- Ensures that a ChannelSecurityInterceptor is registered with the clientInboundChannel. This allows authorization rules to be specified for a message.
- Ensures that a CsrfChannelInterceptor is registered with the clientInboundChannel. This ensures that only requests from the original domain are enabled.
- Ensures that a CsrfTokenHandshakeInterceptor is registered with WebSocketHttpRequestHandler, TransportHandlingSockJsService, or DefaultSockJsService. This ensures that the expected CsrfToken from the HttpServletRequest is copied into the WebSocket Session attributes.

如果需要额外控制，则可以指定id，并将ChannelSecurityInterceptor分配给指定的ID。Spring的消息传递基础设施的所有接线都可以手动完成。这比较麻烦，但是可以更好地控制配置。

<websocket-message-broker> Attributes 译: <websocket-message-broker>属性

- **id** A bean identifier, used for referring to the ChannelSecurityInterceptor bean elsewhere in the context. If specified, Spring Security requires explicit configuration within Spring Messaging. If not specified, Spring Security will automatically integrate with the messaging infrastructure as described in [Section 43.2.1](#), "[websocket-message-broker](#)"
- **same-origin-disabled** Disables the requirement for CSRF token to be present in the Stomp headers (default false). Changing the default is useful if it is necessary to allow other origins to make SockJS connections.

Child Elements of <websocket-message-broker> 译: <websocket-message-broker>的子元素

- [expression-handler](#)
- [intercept-message](#)

43.2.2 <intercept-message> 译: 43.2.2 <拦截消息>

定义消息的授权规则。

Parent Elements of <intercept-message> 译: <拦截消息>的父元素

- [websocket-message-broker](#)

<intercept-message> Attributes 译: <拦截消息>属性

- **pattern** An ant based pattern that matches on the Message destination. For example, `"/"` matches any Message with a destination; `"/admin/"` matches any Message that has a destination that starts with `"/admin/"`.
- **type** The type of message to match on. Valid values are defined in SimpMessageType (i.e. CONNECT, CONNECT_ACK, HEARTBEAT, MESSAGE, SUBSCRIBE, UNSUBSCRIBE, DISCONNECT, DISCONNECT_ACK, OTHER).
- **access** The expression used to secure the Message. For example, `"denyAll"` will deny access to all of the matching Messages; `"permitAll"` will grant access to all of the matching Messages; `"hasRole('ADMIN')"` requires the current user to have the role `'ROLE_ADMIN'` for the matching Messages.

43.3 Authentication Services 译: 43.3认证服务

在Spring Security 3.0之前，`AuthenticationManager`在内部自动注册。现在您必须使用`<authentication-manager>`元素明确注册一个元素。这创建了Spring Security的`ProviderManager`类的实例，该实例需要配置一个或多个`AuthenticationProvider`实例的列表。这些可以使用由名称空间提供的语法元素创建，也可以是

标准的bean定义，使用 `authentication-provider` 元素标记为添加到列表中。

43.3.1 <authentication-manager> 译: 43.3.1 <authentication-manager>

每个使用名称空间的Spring Security应用程序都必须包含此元素。它负责注册为应用程序提供验证服务的 `AuthenticationManager`。所有创建 `AuthenticationProvider` 实例的元素都应该是此元素的子元素。

<authentication-manager> Attributes 译: <authentication-manager>属性

- **alias** This attribute allows you to define an alias name for the internal instance for use in your own configuration. Its use is described in the [namespace introduction](#).
- **erase-credentials** If set to true, the AuthenticationManager will attempt to clear any credentials data in the returned Authentication object, once the user has been authenticated. Literally it maps to the `eraseCredentialsAfterAuthentication` property of the `ProviderManager`. This is discussed in the [Core Services](#) chapter.
- **id** This attribute allows you to define an id for the internal instance for use in your own configuration. It is the same as the alias element, but provides a more consistent experience with elements that use the id attribute.

Child Elements of <authentication-manager> 译: <authentication-manager>的子元素

- [authentication-provider](#)
- [ldap-authentication-provider](#)

43.3.2 <authentication-provider> 译: 43.3.2 <authentication-provider>

除非与 `ref` 属性一起使用，否则此元素是配置 `DaoAuthenticationProvider` 的简写。 `DaoAuthenticationProvider` 载荷从用户信息 `UserDetailsService` 并在登录提供的值的用户名/密码组合进行比较。 `UserDetailsService` 实例可以通过使用可用的命名空间元素（ `jdbc-user-service` 或通过使用 `user-service-ref` 属性指向应用程序上下文在其他位置定义的bean）来定义。您可以在 [namespace introduction](#) 中找到这些变体的示例。

Parent Elements of <authentication-provider> 译: <authentication-provider>的父元素

- [authentication-manager](#)

<authentication-provider> Attributes 译: <authentication-provider>属性

- **ref** Defines a reference to a Spring bean that implements `AuthenticationProvider`.

如果您编写了自己的 `AuthenticationProvider` 实现（或者由于某种原因想要将Spring Security自己的实现配置为传统bean，那么可以使用以下语法将其添加到 `ProviderManager` 的内部列表中：

```
<security:authentication-manager>
<security:authentication-provider ref="myAuthenticationProvider" />
</security:authentication-manager>
<bean id="myAuthenticationProvider" class="com.something.MyAuthenticationProvider"/>
```

- **user-service-ref** A reference to a bean that implements `UserDetailsService` that may be created using the standard bean element or the custom user-service element.

Child Elements of <authentication-provider> 译: <authentication-provider>的子元素

- [jdbc-user-service](#)
- [ldap-user-service](#)
- [password-encoder](#)
- [user-service](#)

43.3.3 <jdbc-user-service> 译: 43.3.3 <jdbc-user-service>

导致创建基于JDBC的 `UserDetailsService`。

<jdbc-user-service> Attributes 译: <jdbc-user-service>属性

- **authorities-by-username-query** An SQL statement to query for a user's granted authorities given a username.

默认是

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

- **cache-ref** Defines a reference to a cache for use with a `UserDetailsService`.
- **data-source-ref** The bean ID of the `DataSource` which provides the required tables.
- **group-authorities-by-username-query** 用于查询给定用户名的用户组权限的SQL语句。默认是

```
select
g.id, g.group_name, ga.authority
from
groups g, group_members gm, group_authorities ga
where
gm.username = ? and g.id = ga.group_id and g.id = gm.group_id
```

- **id** A bean identifier, used for referring to the bean elsewhere in the context.
- **role-prefix** A non-empty string prefix that will be added to role strings loaded from persistent storage (default is "ROLE_"). Use the value "none" for no prefix in cases where the default is non-empty.
- **users-by-username-query** 一个SQL语句，用于在给定用户名的情况下查询用户名，密码和启用状态。默认是


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

43.3.4 <password-encoder> 译: 43.3.4 <密码编码器>

身份验证提供程序可以选择配置为使用密码编码器, 如[namespace introduction](#)中所述。这将导致bean被注入适当的 `PasswordEncoder` 实例。

Parent Elements of <password-encoder> 译: <密码编码器>的父元素

- [authentication-provider](#)
- [password-compare](#)

<password-encoder> Attributes 译: <密码编码器>属性

- hash** Defines the hashing algorithm used on user passwords. We recommend strongly against using MD4, as it is a very weak hashing algorithm.
- ref** Defines a reference to a Spring bean that implements `PasswordEncoder`.

43.3.5 <user-service> 译: 43.3.5 <用户服务>

从属性文件或“用户”子元素列表创建内存中的UserDetailsService。用户名在内部转换为小写字母以允许不区分大小写的查找, 因此如果需要区分大小写, 则不应使用此名称。

<user-service> Attributes 译: <用户服务>属性

- id** A bean identifier, used for referring to the bean elsewhere in the context.
- 属性属性文件的位置, 每行的格式为

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

Child Elements of <user-service> 译: <用户服务>的子元素

- [user](#)

43.3.6 <user> 译: 43.3.6 <user>

代表应用程序中的用户。

Parent Elements of <user> 译: <user>的父元素

- [user-service](#)

<user> Attributes 译: <用户>属性

- authorities** One of more authorities granted to the user. Separate authorities with a comma (but no space). For example, "ROLE_USER,ROLE_ADMINISTRATOR"
- disabled** Can be set to "true" to mark an account as disabled and unusable.
- locked** Can be set to "true" to mark an account as locked and unusable.
- name** The username assigned to the user.
- password** The password assigned to the user. This may be hashed if the corresponding authentication provider supports hashing (remember to set the "hash" attribute of the "user-service" element). This attribute be omitted in the case where the data will not be used for authentication, but only for accessing authorities. If omitted, the namespace will generate a random value, preventing its accidental use for authentication. Cannot be empty.

43.4 Method Security 译: 43.4 方法安全性

43.4.1 <global-method-security> 译: 43.4.1 <全局方法安全性>

这个元素是添加对Spring Security bean上的方法的支持的主要手段。通过使用注解（在接口或类级别定义）或通过使用AspectJ语法将一组切入点定义为子元素, 可以保护方法。

<global-method-security> Attributes 译: <全局方法安全性>属性

- access-decision-manager-ref** Method security uses the same `AccessDecisionManager` configuration as web security, but this can be overridden using this attribute. By default an AffirmativeBased implementation is used for with a RoleVoter and an AuthenticatedVoter.
- authentication-manager-ref** A reference to an `AuthenticationManager` that should be used for method security.
- jsr250-annotations** Specifies whether JSR-250 style attributes are to be used (for example "RolesAllowed"). This will require the javax.annotation.security classes on the classpath. Setting this to true also adds a `JsR250Voter` to the `AccessDecisionManager`, so you need to make sure you do this if you are using a custom implementation and want to use these annotations.
- metadata-source-ref** An external `MethodSecurityMetadataSource` instance can be supplied which will take priority over other sources (such as the default annotations).
- mode** This attribute can be set to "aspectj" to specify that AspectJ should be used instead of the default Spring AOP. Secured methods must be woven with the `AnnotationSecurityAspect` from the `spring-security-aspects` module.

请注意, AspectJ遵循Java的规则, 即接口上的注释不会被继承。这意味着在接口上定义安全注释的方法将不会受到保护。相反, 使用AspectJ时, 您必须将安全注释放在类上。

- **order** Allows the advice "order" to be set for the method security interceptor.
- **pre-post-annotations** Specifies whether the use of Spring Security's pre and post invocation annotations (`@PreFilter`, `@PreAuthorize`, `@PostFilter`, `@PostAuthorize`) should be enabled for this application context. Defaults to "disabled".
- **proxy-target-class** If true, class based proxying will be used instead of interface based proxying.
- **run-as-manager-ref** A reference to an optional `RunAsManager` implementation which will be used by the configured `MethodSecurityInterceptor`
- **secured-annotations** Specifies whether the use of Spring Security's `@Secured` annotations should be enabled for this application context. Defaults to "disabled".

Child Elements of `<global-method-security>` 译: <global-method-security>的子元素

- [after-invocation-provider](#)
- [expression-handler](#)
- [pre-post-annotation-handling](#)
- [protect-pointcut](#)

43.4.2 `<after-invocation-provider>` 译: 43.4.2 <调用后提供程序>

这个元素可以用来装饰 `AfterInvocationProvider` 以供由 `<global-method-security>` 名称空间维护的安全拦截器使用。您可以在 `global-method-security` 元素中定义零个或多个元素，每个元素的 `ref` 属性指向应用程序上下文中的 `AfterInvocationProvider` bean实例。

Parent Elements of `<after-invocation-provider>` 译: <调用后提供程序>的父元素

- [global-method-security](#)

`<after-invocation-provider>` Attributes 译: <调用后提供程序>属性

- **ref** Defines a reference to a Spring bean that implements `AfterInvocationProvider`.

43.4.3 `<pre-post-annotation-handling>` 译: 43.4.3 <注释前处理>

允许基于表达式的默认机制来处理Spring Security的前后调用注释 (`@PreFilter`, `@PreAuthorize`, `@PostFilter`, `@PostAuthorize`) 完全被替换。只有在启用了这些注释的情况下才适用。

Parent Elements of `<pre-post-annotation-handling>` 译: <注释前处理>的父元素

- [global-method-security](#)

Child Elements of `<pre-post-annotation-handling>` 译: <注释前处理>的子元素

- [invocation-attribute-factory](#)
- [post-invocation-advice](#)
- [pre-invocation-advice](#)

43.4.4 `<invocation-attribute-factory>` 译: 43.4.4 <调用属性工厂>

定义 `PrePostInvocationAttributeFactory` 实例，该实例用于从注释的方法中生成前后调用元数据。

Parent Elements of `<invocation-attribute-factory>` 译: <invocation-attribute-factory>的父元素

- [pre-post-annotation-handling](#)

`<invocation-attribute-factory>` Attributes 译: <invocation-attribute-factory>属性

- **ref** Defines a reference to a Spring bean Id.

43.4.5 `<post-invocation-advice>` 译: 43.4.5 <调用后建议>

定制 `PostInvocationAdviceProvider` 使用 `ref` 为 `PostInvocationAuthorizationAdvice` 的 `<注解预处理>` 元素。

Parent Elements of `<post-invocation-advice>` 译: <调用后建议>的父元素

- [pre-post-annotation-handling](#)

`<post-invocation-advice>` Attributes 译: <调用后建议>属性

- **ref** Defines a reference to a Spring bean Id.

43.4.6 `<pre-invocation-advice>` 译: 43.4.6 <调用前建议>

定制 `PreInvocationAuthorizationAdviceVoter` 使用 `ref` 为 `PreInvocationAuthorizationAdviceVoter` 的 `<注解预处理>` 元素。

Parent Elements of `<pre-invocation-advice>` 译: <pre-invocation-advice>的父元素

- [pre-post-annotation-handling](#)

`<pre-invocation-advice>` Attributes 译: <pre-invocation-advice>属性

- **ref** Defines a reference to a Spring bean Id.

43.4.7 Securing Methods using 译: 43.4.7使用安全方法

`<protect-pointcut>` 而不是使用 `@Secured` 批注在单个方法或类基础上定义安全属性, 您可以使用 `<protect-pointcut>` 元素在服务层中的 `<protect-pointcut>` 方法和界面中定义横切安全约束。您可以在 [namespace introduction](#) 找到一个例子。

Parent Elements of `<protect-pointcut>` 译: <protect-pointcut>的父元素

- [global-method-security](#)

`<protect-pointcut>` Attributes 译: <protect-pointcut>属性

- access** Access configuration attributes list that applies to all methods matching the pointcut, e.g. "ROLE_A,ROLE_B"
- expression** An AspectJ expression, including the 'execution' keyword. For example, 'execution(int com.foo.TargetObject.countLength(String))' (without the quotes).

43.4.8 `<intercept-methods>` 译: 43.4.8 <拦截方法>

可以在一个bean定义中使用, 将一个安全拦截器添加到bean中, 并为bean的方法设置访问配置属性

`<intercept-methods>` Attributes 译: <拦截方法>属性

- access-decision-manager-ref** Optional AccessDecisionManager bean ID to be used by the created method security interceptor.

Child Elements of `<intercept-methods>` 译: <拦截方法>的子元素

- [protect](#)

43.4.9 `<method-security-metadata-source>` 译: 43.4.9 <method-security-metadata-source>

创建一个MethodSecurityMetadataSource实例

`<method-security-metadata-source>` Attributes 译: <method-security-metadata-source>属性

- id** A bean identifier, used for referring to the bean elsewhere in the context.
- use-expressions** Enables the use of expressions in the 'access' attributes in `<intercept-url>` elements rather than the traditional list of configuration attributes. Defaults to 'false'. If enabled, each attribute should contain a single Boolean expression. If the expression evaluates to 'true', access will be granted.

Child Elements of `<method-security-metadata-source>` 译: <method-security-metadata-source>的子元素

- [protect](#)

43.4.10 `<protect>` 译: 43.4.10 <protect>

定义受保护的方法以及应用于其的访问控制配置属性。我们强烈建议您不要将“保护”声明与任何提供的“全局方法安全性”服务混合使用。

Parent Elements of `<protect>` 译: <protect>的父元素

- [intercept-methods](#)
- [method-security-metadata-source](#)

`<protect>` Attributes 译: <protect>属性

- access** Access configuration attributes list that applies to the method, e.g. "ROLE_A,ROLE_B".
- method** A method name

43.5 LDAP Namespace Options 译: 43.5 LDAP名称空间选项

有关详细信息, 请[参阅its own chapter](#)。我们将在这里扩展一些关于命名空间选项如何映射到Spring bean的解释。LDAP实现广泛使用Spring LDAP, 所以熟悉该项目的API可能会有用。

43.5.1 Defining the LDAP Server using the 译: 43.5.1使用 - 定义LDAP服务器

`<ldap-server>` 元素此元素设置Spring LDAP `ContextSource` 供其他LDAP Bean使用, 定义LDAP服务器的位置和其他信息 (如用户名和密码, 如果它不允许匿名访问) 用于连接到它的位置。它也可以用来创建一个用于测试的嵌入式服务器。 [LDAP chapter](#)中涵盖了这两种选项的语法细节。实际的 `ContextSource` 实现是 `DefaultSpringSecurityContextSource`, 它扩展了Spring LDAP的 `LdapContextSource` 类。 `manager-dn` 和 `manager-password` 属性分别映射到后者的 `userDn` 和 `password` 属性。

如果您的应用程序上下文中只定义了一个服务器, 则其他LDAP名称空间定义的bean将自动使用它。否则, 您可以给该元素一个“id”属性, 并使用 `server-ref` 属性从其他名称空间bean引用该属性。这实际上是 `ContextSource` 实例的bean `id`, 如果您想在其他传统Spring bean中使用它。

`<ldap-server>` Attributes 译: <ldap-server>属性

- id** A bean identifier, used for referring to the bean elsewhere in the context.
- ldif** Explicitly specifies an ldif file resource to load into an embedded LDAP server. The ldif is should be a Spring resource pattern (i.e. classpath:*.ldif). The default is classpath*:*.ldif
- manager-dn** Username (DN) of the "manager" user identity which will be used to authenticate to a (non-embedded) LDAP server. If omitted, anonymous access will be

used.

- **manager-password** The password for the manager DN. This is required if the manager-dn is specified.
- **port** Specifies an IP port number. Used to configure an embedded LDAP server, for example. The default value is 33389.
- **root** Optional root suffix for the embedded LDAP server. Default is "dc=springframework,dc=org"
- **url** Specifies the ldap server URL when not using the embedded LDAP server.

43.5.2 <ldap-authentication-provider> 译: 43.5.2 <ldap-authentication-provider>

该元素是创建 `LdapAuthenticationProvider` 实例的简写。默认情况下, 这将使用 `BindAuthenticator` 实例和 `DefaultAuthoritiesPopulator` 进行配置。与所有名称空间身份验证提供程序一样, 它必须作为 `authentication-provider` 元素的子项包含在内。

Parent Elements of <ldap-authentication-provider> 译: <ldap-authentication-provider>的父元素

- [authentication-manager](#)

<ldap-authentication-provider> Attributes 译: <ldap-authentication-provider>属性

- **group-role-attribute** The LDAP attribute name which contains the role name which will be used within Spring Security. Maps to the `DefaultLdapAuthoritiesPopulator`'s `groupRoleAttribute` property. Defaults to "cn".
- **group-search-base** Search base for group membership searches. Maps to the `DefaultLdapAuthoritiesPopulator`'s `groupSearchBase` constructor argument. Defaults to "" (searching from the root).
- **group-search-filter** Group search filter. Maps to the `DefaultLdapAuthoritiesPopulator`'s `groupSearchFilter` property. Defaults to (uniqueMember={0}). The substituted parameter is the DN of the user.
- **role-prefix** A non-empty string prefix that will be added to role strings loaded from persistent. Maps to the `DefaultLdapAuthoritiesPopulator`'s `rolePrefix` property. Defaults to "ROLE_". Use the value "none" for no prefix in cases where the default is non-empty.
- **server-ref** The optional server to use. If omitted, and a default LDAP server is registered (using `<ldap-server>` with no id), that server will be used.
- **user-context-mapper-ref** Allows explicit customization of the loaded user object by specifying a `UserDetailsContextMapper` bean which will be called with the context information from the user's directory entry
- **user-details-class** Allows the objectClass of the user entry to be specified. If set, the framework will attempt to load standard attributes for the defined class into the returned `UserDetails` object
- **user-dn-pattern** If your users are at a fixed location in the directory (i.e. you can work out the DN directly from the username without doing a directory search), you can use this attribute to map directly to the DN. It maps directly to the `userDnPatterns` property of `AbstractLdapAuthenticator`. The value is a specific pattern used to build the user's DN, for example "uid={0},ou=people". The key "{0}" must be present and will be substituted with the username.
- 用户搜索库用户搜索的搜索基础。默认为""。仅与“用户搜索过滤器”一起使用。
如果您需要执行搜索以在目录中找到用户, 则可以设置这些属性来控制搜索。`BindAuthenticator` 将配置为 `FilterBasedLdapUserSearch`, 属性值直接映射到该bean构造函数的前两个参数。如果这些属性没有设置, 并且没有提供 `user-dn-pattern` 作为替代, 则将使用默认搜索值 `user-search-filter="(uid={0})"` 和 `user-search-base=""`。
- 用户搜索过滤器用于搜索用户的LDAP过滤器(可选)。例如“(uid = {0})”。替换参数是用户的登录名。
如果您需要执行搜索以在目录中找到用户, 则可以设置这些属性来控制搜索。`BindAuthenticator` 将被配置为 `FilterBasedLdapUserSearch`, 属性值直接映射到该bean的构造函数的前两个参数。如果这些属性没有设置, 并且没有提供 `user-dn-pattern` 作为替代, 则将使用默认搜索值 `user-search-filter="(uid={0})"` 和 `user-search-base=""`。

Child Elements of <ldap-authentication-provider> 译: <ldap-authentication-provider>的子元素

- [password-compare](#)

43.5.3 <password-compare> 译: 43.5.3 <密码比较>

这用作 `<ldap-provider>` 子元素, `<ldap-provider>` 身份验证策略从 `BindAuthenticator` 到 `PasswordComparisonAuthenticator`。

Parent Elements of <password-compare> 译: <密码比较>的父元素

- [ldap-authentication-provider](#)

<password-compare> Attributes 译: <password-compare>属性

- **hash** Defines the hashing algorithm used on user passwords. We recommend strongly against using MD4, as it is a very weak hashing algorithm.
- **password-attribute** The attribute in the directory which contains the user password. Defaults to "userPassword".

Child Elements of <password-compare> 译: <password-compare>的子元素

- [password-encoder](#)

43.5.4 <ldap-user-service> 译: 43.5.4 <ldap-user-service>

该元素配置LDAP `UserDetailsService`。使用的类是 `LdapUserDetailsService`, 它是 `FilterBasedLdapUserSearch` 和 `DefaultLdapAuthoritiesPopulator` 的组合。它支持的属性与 `<ldap-provider>` 具有相同的用法。

<ldap-user-service> Attributes 译: <ldap-user-service>属性

- **cache-ref** Defines a reference to a cache for use with a `UserDetailsService`.

- **group-role-attribute** The LDAP attribute name which contains the role name which will be used within Spring Security. Defaults to "cn".
- **group-search-base** Search base for group membership searches. Defaults to "" (searching from the root).
- **group-search-filter** Group search filter. Defaults to (uniqueMember={0}). The substituted parameter is the DN of the user.
- **id** A bean identifier, used for referring to the bean elsewhere in the context.
- **role-prefix** A non-empty string prefix that will be added to role strings loaded from persistent storage (e.g. "ROLE_"). Use the value "none" for no prefix in cases where the default is non-empty.
- **server-ref** The optional server to use. If omitted, and a default LDAP server is registered (using <ldap-server> with no id), that server will be used.
- **user-context-mapper-ref** Allows explicit customization of the loaded user object by specifying a UserDetailsContextMapper bean which will be called with the context information from the user's directory entry
- **user-details-class** Allows the objectClass of the user entry to be specified. If set, the framework will attempt to load standard attributes for the defined class into the returned UserDetails object
- **user-search-base** Search base for user searches. Defaults to "". Only used with a 'user-search-filter'.
- **user-search-filter** The LDAP filter used to search for users (optional). For example "(uid={0})". The substituted parameter is the user's login name.

[22] 请参阅 [introductory chapter](#) 了解如何设置 `web.xml` 的映射

[23] 此功能实际上只是为了方便而提供的，并不适用于生产（其中视图技术将被选择并可用于呈现自定义登录页面）。类 `DefaultLoginPageGeneratingFilter` 负责呈现登录页面，并且如果需要，将提供正常形式登录和/或OpenID的登录表单。

[24] 这不会影响 `PersistentTokenBasedRememberMeServices` 的使用，其中令牌存储在服务器端。

44. Spring Security Dependencies 译：44.Spring Security依赖关系

本附录提供了Spring Security中的模块参考，以及为了在正在运行的应用程序中运行而需要的其他依赖关系。 我们不包含仅在构建或测试Spring Security本身时使用的依赖关系。 我们也不包括外部依赖所需的传递依赖。

项目网站上列出了所需的Spring版本，因此下面的Spring依赖关系省略了特定版本。 请注意，Spring应用程序中的其他非安全功能仍需要下面列为“可选”的一些依赖项。 如果项目的Maven POM文件在大多数应用程序中使用，那么列出的“可选”依赖项实际上可能不会被标记为这样。 除非您使用指定的功能，否则只有在您不需要它们意义上，它们才是“可选的”。

如果一个模块依赖于另一个Spring Security模块，那么它所依赖的模块的非可选依赖关系也被假定为必需的，并且不会单独列出。

44.1 spring-security-core 译：44.1Spring安全核心

核心模块必须包含在使用Spring Security的任何项目中。

表44.1。 核心依赖关系

Dependency	Version	描述
的Ehcache	1.6.2	如果使用基于Ehcache的用户缓存实现（可选），则为必需。
春天的AOP		方法安全性基于Spring AOP
弹簧豆		Spring配置需要
弹簧表达		基于表达式的方法安全性（可选）
弹簧JDBC		如果使用数据库存储用户数据（可选），则为必需。
春天-TX		如果使用数据库存储用户数据（可选），则为必需。
aspectjrt	1.6.10	如果使用AspectJ支持（可选），则为必需。
JSR250的API	1.0	如果您使用JSR-250方法安全性注释（可选），则为必需。

44.2 spring-security-remoting 译：44.2Spring安全远程处理

在使用Servlet API的Web应用程序中通常需要此模块。

表44.2。 远程依赖

Dependency	Version	描述
弹簧安全核心		
弹簧网		对于使用HTTP远程支持的客户端是必需的。

44.3 spring-security-web 译：44.3 spring-security-web

在使用Servlet API的Web应用程序中通常需要此模块。

表44.3。 Web相关性

Dependency	Version	描述
弹簧安全核心		
弹簧网		Spring Web支持类被广泛使用。
弹簧JDBC		基于JDBC的持久性记忆-Me标记库（可选）是必需的。
春天-TX		记住 - 持久性标记库实现（可选）是必需的。

44.4 spring-security-ldap 译：44.4 spring-security-ldap

只有在使用LDAP认证时才需要此模块。

表44.4。LDAP依赖关系

Dependency	Version	描述
弹簧安全核心		
弹簧LDAP的芯	1.3.0	LDAP支持基于Spring LDAP。
春天-TX		数据异常类是必需的。
apache-ds ^[1]	1.5.5	如果您正在使用嵌入式LDAP服务器（可选），则为必需。
共享LDAP	0.9.15	如果您正在使用嵌入式LDAP服务器（可选），则为必需。
ldapsdk	4.1	Mozilla LdapSDK。例如，如果您使用OpenLDAP的密码策略功能，则用于解码LDAP密码策略控件。
^[1] [↑] 模块 <code>apacheds-core</code> ， <code>apacheds-core-entry</code> ， <code>apacheds-protocol-shared</code> ， <code>apacheds-protocol-ldap</code> 和 <code>apacheds-server-jndi</code> 是必需的。		

44.5 spring-security-config 译：44.5 spring-security-config

如果您使用Spring Security命名空间配置，则此模块是必需的。

表44.5。配置依赖关系

Dependency	Version	描述
弹簧安全核心		
弹簧安全网		如果您使用任何与Web相关的命名空间配置（可选），则为必需。
弹簧安全LDAP		如果您使用LDAP名称空间选项（可选），则为必需。
弹簧安全的OpenID		如果您使用OpenID身份验证（可选），则为必需。
aspectjweaver	1.6.10	如果使用protect-pointcut命名空间语法（可选），则为必需。

44.6 spring-security-acl 译：44.6 spring-security-acl

ACL模块。

表44.6。ACL依赖关系

Dependency	Version	描述
弹簧安全核心		
的Ehcache	1.6.2	如果使用基于Ehcache的ACL缓存实现（如果您使用自己的实现，则为可选）时是必需的。
弹簧JDBC		如果您使用默认的基于JDBC的AclService，则为必需（如果您自己实现，则为可选）。
春天-TX		如果您使用默认的基于JDBC的AclService，则为必需（如果您自己实现，则为可选）。

44.7 spring-security-cas 译：44.7 spring-security-cas

CAS模块提供与JA-SIG CAS的集成。

表44.7。CAS依赖关系

Dependency	Version	描述
弹簧安全核心		
弹簧安全网		

Dependency	Version	描述
CAS客户端核心	3.1.12	JA-SIG CAS客户端。这是Spring Security集成的基础。
的Ehcache	1.6.2	如果您使用基于Ehcache的票证缓存（可选），则为必需。

44.8 spring-security-openid 译：44.8 spring-security-openid

OpenID 模块。

表 44.8。 OpenID 依赖关系

Dependency	Version	描述
弹簧安全核心		
弹簧安全网		
openid4java-nodeps	0.9.6	Spring Security的OpenID集成使用OpenID4Java。
HttpClient的	4.1.1	openid4java-nodeps依赖于HttpClient 4。
吉斯	2.0	openid4java-nodeps依赖于Guice 2。

44.9 spring-security-taglibs 译：44.9 spring-security-taglibs

提供Spring Security的JSP标记实现。

表 44.9。 Taglib 依赖

Dependency	Version	描述
弹簧安全核心		
弹簧安全网		
弹簧安全ACL		如果您将 <code>accesscontrollist</code> 标记或 <code>hasPermission()</code> 表达式与ACL一起使用（可选）， <code>hasPermission()</code> 必需。
弹簧表达		如果您在标签访问限制中使用SPEL表达式，则为必需。

45. Proxy Server Configuration 译：45 代理服务器配置

使用代理服务器时，确保您已正确配置应用程序非常重要。例如，许多应用程序将有一个负载均衡器，通过将请求转发到应用程序服务器<http://192.168.1:8080>来响应<https://example.com/>的请求。如果没有正确的配置，应用程序服务器将不知道负载均衡器存在，并像请求<http://192.168.1:8080>一样处理请求由客户。

要解决此问题，可以使用[RFC 7239](#)指定正在使用负载平衡器。为了使应用程序知道这一点，您需要配置您的应用程序服务器知道X-Forwarded标头。例如Tomcat使用[RemoteIpValve](#)，Jetty使用[ForwardedRequestCustomizer](#)。或者，Spring 4.3+用户可以利用[ForwardedHeaderFilter](#)。

46. Spring Security FAQ 译：46 春季安全FAQ

- [Section 46.1, "General Questions"](#)
- [Section 46.2, "Common Problems"](#)
- [Section 46.3, "Spring Security Architecture Questions"](#)
- [Section 46.4, "Common "Howto" Requests"](#)

46.1 General Questions 译：46.1 般问题

1. [Section 46.1.1, "Will Spring Security take care of all my application security requirements?"](#)
2. [Section 46.1.2, "Why not just use web.xml security?"](#)
3. [Section 46.1.3, "What Java and Spring Framework versions are required?"](#)
4. [Section 46.1.4, "I'm new to Spring Security and I need to build an application that supports CAS single sign-on over HTTPS, while allowing Basic authentication locally for certain URLs, authenticating against multiple back end user information sources \(LDAP and JDBC\). I've copied some configuration files I found but it doesn't work."](#)

46.1.1 Will Spring Security take care of all my application security requirements? 译：46.1.1 Spring Security能否会处理我所有的应用程序安全要求？

Spring Security为您提供了一个非常灵活的框架来满足您的身份验证和授权要求，但构建不在其范围内的安全应用程序还有许多其他注意事项。Web应用程序容易受到各种您应该熟悉的攻击，最好在开始开发之前进行，以便您可以从头开始设计和编写这些攻击。查看<http://www.owasp.org/> [OWASP网站]，了解Web应用程序开发人员面临的主要问题以及您可以使用的对策。

46.1.2 Why not just use web.xml security? 译：46.1.2为什么不使用web.xml安全性？

假设您正在开发基于Spring的企业应用程序。您通常需要解决四个安全问题：身份验证，Web请求安全性，服务层安全性（即您的实现业务逻辑的方法）和域对象实例安全性（即不同的域对象具有不同的权限）。考虑到这些典型要求：

1. *Authentication:* The servlet specification provides an approach to authentication. However, you will need to configure the container to perform authentication which typically

requires editing of container-specific "realm" settings. This makes a non-portable configuration, and if you need to write an actual Java class to implement the container's authentication interface, it becomes even more non-portable. With Spring Security you achieve complete portability - right down to the WAR level. Also, Spring Security offers a choice of production-proven authentication providers and mechanisms, meaning you can switch your authentication approaches at deployment time. This is particularly valuable for software vendors writing products that need to work in an unknown target environment.

2. *Web request security*: The servlet specification provides an approach to secure your request URLs. However, these URLs can only be expressed in the servlet specification's own limited URI path format. Spring Security provides a far more comprehensive approach. For instance, you can use Ant paths or regular expressions, you can consider parts of the URI other than simply the requested page (e.g. you can consider HTTP GET parameters) and you can implement your own runtime source of configuration data. This means your web request security can be dynamically changed during the actual execution of your webapp.
3. *服务层和域对象安全*: Servlet规范中缺少对服务层安全性或域对象实例安全性的支持，这对于多层应用程序来说是严重的限制。通常开发人员要么忽略这些需求，要么在他们的MVC控制器代码中实现安全逻辑（或者更糟糕的是，在视图内部）。这种方法存在严重的缺点：
 - a. *Separation of concerns*: Authorization is a crosscutting concern and should be implemented as such. MVC controllers or views implementing authorization code makes it more difficult to test both the controller and authorization logic, more difficult to debug, and will often lead to code duplication.
 - b. *Support for rich clients and web services*: If an additional client type must ultimately be supported, any authorization code embedded within the web layer is non-reusable. It should be considered that Spring remoting exporters only export service layer beans (not MVC controllers). As such authorization logic needs to be located in the services layer to support a multitude of client types.
 - c. *Layering issues*: An MVC controller or view is simply the incorrect architectural layer to implement authorization decisions concerning services layer methods or domain object instances. Whilst the Principal may be passed to the services layer to enable it to make the authorization decision, doing so would introduce an additional argument on every services layer method. A more elegant approach is to use a ThreadLocal to hold the Principal, although this would likely increase development time to a point where it would become more economical (on a cost-benefit basis) to simply use a dedicated security framework.
 - d. *Authorisation code quality*: It is often said of web frameworks that they "make it easier to do the right things, and harder to do the wrong things". Security frameworks are the same, because they are designed in an abstract manner for a wide range of purposes. Writing your own authorization code from scratch does not provide the "design check" a framework would offer, and in-house authorization code will typically lack the improvements that emerge from widespread deployment, peer review and new versions.

对于简单的应用程序，servlet规范安全性可能就足够了。虽然在Web容器可移植性，配置要求，有限的Web请求安全灵活性以及不存在的服务层和域对象实例安全性的上下文中考虑，但很明显为什么开发人员往往会选择其他解决方案。

46.1.3 What Java and Spring Framework versions are required?译: 46.1.3需要哪些Java和Spring Framework版本?

Spring Security 3.0和3.1至少需要JDK 1.5，并且至少需要Spring 3.0.3。理想情况下，您应该使用最新的版本来避免问题。

Spring Security 2.0.x需要1.4的最低JDK版本，并且是针对Spring 2.0.x构建的。它也应该与使用Spring 2.5.x的应用程序兼容。

46.1.4 I'm new to Spring Security and I need to build an application that supports CAS single sign-on over HTTPS, while allowing Basic authentication locally for certain URLs, authenticating against multiple back end user information sources (LDAP and JDBC). I've copied some configuration files I found but it doesn't work.译: 46.1.4我是Spring Security的新成员，我需要构建支持通过HTTPS进行CAS单点登录的应用程序，同时允许对某些URL进行本地基本身份验证，对多个后端用户信息源（LDAP和JDBC）。我复制了一些我发现的配置文件，但它不起作用。

什么可能是错的？

或者替代复杂的情景.....

实际上，您需要先了解您打算使用的技术，然后才能成功构建应用程序。安全性很复杂。使用Spring Security的命名空间使用登录表单和一些硬编码的用户来设置一个简单的配置非常简单。转向使用支持的JDBC数据库也很简单。但是如果您尝试直接跳到这样复杂的部署场景中，您几乎肯定会感到沮丧。设置CAS系统，配置LDAP服务器和正确安装SSL证书所需的学习曲线有很大的提升。所以您需要一次采取一步。

从Spring Security的角度来看，您应该做的第一件事是按照网站上的“入门”指南。这将通过一系列步骤来启动和运行，并了解框架如何运作。如果您正在使用其他您不熟悉的技术，那么您应该进行一些研究，并尝试确保在将它们组合到一个复杂系统之前将它们单独使用。

46.2 Common Problems译: 46.2常见问题

1. 认证
 - a. [Section 46.2.1, "When I try to log in, I get an error message that says "Bad Credentials". What's wrong?"](#)
 - b. [Section 46.2.2, "My application goes into an "endless loop" when I try to login, what's going on?"](#)
 - c. [Section 46.2.3, "I get an exception with the message "Access is denied \(user is anonymous\);". What's wrong?"](#)
 - d. [Section 46.2.4, "Why can I still see a secured page even after I've logged out of my application?"](#)
 - e. [Section 46.2.5, "I get an exception with the message "An Authentication object was not found in the SecurityContext". What's wrong?"](#)