Spring Security 参考文档

Ben Alex , Luke Taylor ,
Rob Winch , Gunnar Hillert

目录

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

翻译.
Jeff Lu

4.1.3.RELEASE

部分 I. 序章

Spring Security 为基于 Java-EE 的企业级应用提供一种综合的解决方案。

对于安全的讨论永不过时，系统安全非常重要。我们鼓励你采用"多安全层(layers of security)"的方式，每一层都负责一定范围内的安全性校验。每一层的校验越严格，你的应用就越安全、越健壮。

在底层，你需要保证传输安全、处理系统认证，以减少中间人攻击。然后，你需要使用防火墙，也许还要配合使用VPNs或者IP安全来确保只有经过授权的系统可以接入和连接。在公共环境下，你需要部署DMZ，将面向公众的服务器与后台数据库、应用服务器隔离。你的操作系统也将扮演一个重要的部分，你需要解决例如非权限用户运行程序与文件系统安全性最大化这些问题。通常，操作系统也会拥有自己的防火墙配置。你还需要防范拒绝服务攻击和暴力系统攻击。入侵检测系统在这种情况下非常有用，它提供诸如实时拒绝某些异常的TCP/IP连接之类的功能。

在更高层次，你的Java虚拟机最好为不同的Java类型配置不同的并且最小化的权限，同时，你的应用需要加入对于特定领域的安全配置。

Spring Security 试图让这一层次 - 应用安全层 - 更加简单。

当然，你需要妥善处理上面所提到的所有安全层，还要处理每一个安全层的管理要素。管理要素包括安全证书监控，修复缺陷，人员审查,审核，控制变更，系统项目管理，数据备份，灾难恢复，性能测评，负载监控，日志集中，事件响应程序等等…

Spring Security 专注于帮助你的企业级应用层安全，你会发现在业务问题领域有很多不一样的需求。银行应用与电商应用的需要并不完全一样，电商应用与公司销售部门自动化工具的需要也不一样。这些定制的需求让应用安全变得非常有趣、具有挑战，非常值得。

阅读《第一章 开始》，它是很好的起点。它会向你介绍此框架，并通过一个基于命名空间进行配置的示例带领你快速入门。关于 Spring Security 如何运作，以及一些常用的classes的更多信息，你可以参见《第二部分　架构与实现》。本部分剩余章节以传统的参考手册的风格来组织，方便用户在拥有一定的基础之后进行阅读。我们也建议您多阅读一些与应用安全问题相关的文档，Spring Security 不是能够解决所有安全问题的万能灵药。应用程序从设计开始就要伴随着安全性的考虑，这非常重要，试图改进它并非一个好主意。尤其是如果你在构建一个Web

应用，你应该意识到很多诸如跨站脚本、伪造请求以及session劫持之类的潜在缺陷，你应该从一开始就仔细斟酌。

OWASP web 站点[1] 上保持前十的 web 应用漏洞清单是非常有用的参考信息。

我们希望此参考手册对你有所帮助，我们也欢迎你提供反馈与建议。

最后，由衷的欢迎你加入 Spring Security 社区。

[1] http://www.owasp.org/

目录

1

开始

本指南提供了一个深入的后续部分讨论框架的架构和实现类,你需要了解,如果你想做任何严重的定制。在本部分中,我们将介绍Spring Security 4.0,给项目的历史进行了简要的概述,并且稍微温和的看看如何开始使用框架。特别是,我们来看看名称空间配置提供一个更简单的方式保护您的应用程序相比,传统的Spring bean方法,你必须连接所有的单独实现类。

本指南的最新部分提供了关于框架架构与实现类的深入讨论，如果你需要严谨的定制此框架，你需要理解它们。

在这一部分，我们会介绍 Spring Security 4.0 ，简单的回顾一下此项目的历史，然后温和的带你看看如何使用此框架。

传统的Spring bean配置方法需要手动连接所有单独的实现类。在此文档中，我们将关注于命名空间配置方法，与传统方法相比，此方法更为简便。

我们也会向你展示一些可用的应用示例。在你阅读后续部分之前，它们很值得你去试着运行并试验一下。你可以在更深入理解框架之后再一次回顾它们。

我们也欢迎您提供在构建项目的过程中有用的信息，附上文章、视频或教程的链接。

2

简介

2.1. Spring Security 是什么？

Spring Security 为基于 Java EE 的企业应用提供综合的安全服务。尤其支持使用 Spring Framework构建的项目，这是目前企业软件开发的流行的Java EE解决方案。

如果在企业应用开发中没有使用 Spring， 我们由衷的鼓励你仔细看看它。深入Spring并理解依赖注入原则，将帮助你很快的学会 Spring Security 。

人们使用 Spring Security 的原因有很多，但迁移到此项目最主要的原因是发现 Java EE Servlet 规范或 EJB 规范的安全特性在典型的企业应用场景中缺乏深度，它们在 WAR 或 EAR 级别也并不轻便。因此，如果你需要切换服务环境，那么你需要在新的环境中做大量的重复配置工作。

使用 Spring Security 可以解决这些问题，它还可以为你带来更多其他有用且可定制的安全特性。

你也许已经知道，应用安全最主要的两个方面分别是“认证”与“授权”（或“访问控制”）。它们也是 Spring Security 的两个主要目标。“认证”是确认角色（principal）身份的过程（"principal"通常表示一个用户、设备或其他可以在你的应用中执行操作的系统）。“授权”指的是判定角色是否拥有执行动作的权限的过程。在到达需要进行授权判定的地方前，角色的身份必须已经通过认证。这些概念非常通用，它们并不是 Spring Security 的特性。

principal 本意是：本金；首长，负责人；主要演员，主角；[法] 委托人，当事人。在此翻译成“角色”有欠妥当，但可以便于读者理解。（译者注）

在认证级别，Spring Security 支持范围广泛的认证模型。大多数认证模型由第三方提供，或由相关标准制定组织(如 IETF)研制。

另外，Spring Security 提供它自己的认证功能。

特别地，Spring Security 目前已经集成了如下一些认证技术：

- HTTP BASIC headers认证 (基于IETF RFC标准)
- HTTP Digest headers认证 (基于IETF RFC标准)
- HTTP X.509 客户端证书交换 (基于IETF RFC标准)
- LDAP (一种非常常见的跨平台认证需求，在大环境下尤甚)
- 基于表单的认证 (通常用于简单的用户需求)
- OpenID认证
- 基于预定义的请求头进行认证 (诸如计算机集群)
- JA-SIG 中央认证服务 (也称为CAS, 是一种流行的开源的单点登陆系统)
- 为RMI 和 HttpInvoker（一种Spring远程协议）提供对用户透明的认证机制
- 自动的 "remember-me" 认证 (你可以勾选此选项，从而避免在一定时间内进行重复认证)
- 匿名认证 (为每个未经认证的用户提供一个特殊的安全身份)
- Run-as 认证 (同一个调用需要进行不同的安全校验时非常有用)
- Java认证与认证服务 - Java Authentication and Authorization Service (JAAS)
- JEE认证 (如果你需要，你仍然可以使用容器托管认证)
- Kerberos
- Java开源单点登陆 - Java Open Source Single Sign On (JOSSO) *
- OpenNMS网络管理平台 - OpenNMS Network Management Platform *
- AppFuse *
- AndroMDA *
- Mule ESB *
- Direct Web Request (DWR) *
- Grails *
- Tapestry *

- JTrac *
- Jasypt *
- Roller *
- Elastic Path *
- Atlassian Crowd *
- 你自己的权限系统 (参见后续内容)

带*表示由第三方提供。

许多独立软件供应商(independent software vendors，ISVs)采用Spring Security，因为这代表着选择了灵活的认证模块。无论终端是什么，这样做都可以让他们迅速地集成他们自己的解决方案，而不必做大量的工作来适应客户端环境的变更。

如果上述的认证机制没有满足你的需求，Spring Security也提供一个开放的平台让你方便的实现自己的认证机制。很多企业用户需要Spring Security帮助集成遗留（"legacy"）的系统，而不需要其他标准的安全特性，那么Spring Security也能很好的工作。

认证机制无关，Spring Security允许对授权功能进行深度设置。它关注于三个主要方面：web请求授权、方法调用授权以及域对象实例访问授权。为了帮助你理解它们的区别，你可以思考如何分别对Servlet规范的web模式(pattern)安全、EJB容器托管安全以及文件系统安全提供授权功能。Spring Security对这些领域都提供深入的功能，我们会在下文为您展示。

2.2. 历史背景

Spring Security 开始于2003年后半年的“The Acegi Security System for Spring”。Spring开发者邮件清单中提出了这一问题，大家都在询问我们是否在考虑一个基于Spring的安全实现。这时Spring社区还没有现在这么庞大，并且Spring也仅仅是一个2003年早期的SourceForge项目。我们认为这是一个很有价值的问题，尽管当时没有时间去探索。

伴随着这一想法，一个简单的security实现被构建出来而未发布。几周之后，一些Spring社区的其他成员开始询问有关安全的问题，于是我们将源码提供给了他们。直到2004年1月，已经大约有20人在使用此代码。这些首批用户开始建议将其作为一个SourceForge项目，于是项目于2004年3月正式成立。

在那一时期，此项目并没有自身的认证模块。认证过程依靠容器托管安全，并且使用Acegi Security处理授权功能。开始的时候还狠不错，但随着需要支持的容器越来越多，特定于容器认证领域的基本限制开始出现。还有一些关于将新的JARs加入容器的classpath的问题，这些问题经常使用户感到混乱并且难于配置。

特定于Acegi Security的认证服务随后被引入。大约一年之后，Acegi Security成为了Spring Framework官方子项目。2006年5月，1.0.0 final release版本正式发布。经过将近三年的时间，它已经被活跃地使用在很多产品软件项目中，并且得到数以百计的改进以及社区贡献。

Acegi Security在2007年年末的时候，正式成为Spring Portfolio官方项目，并更名为"Spring Security"。

到了今天，Spring Security已经加入了健壮而活跃的开源社区。在论坛上已经有数以千计的关于Spring Security的消息。这里有活跃的核心开发者，有活跃的社区，还有定期修复与支持此项目的同志。

2.3. 版本编号

理解Spring Security版本编号会对你的工作非常有用，这会对你的项目迁移有所帮助。每个版本都使用标准的整数排列：MAJOR.MINOR.PATCH(主版本.小版本.补丁)。MAJOR的变更意味着大规模的API的更新，因此很多地方都不会兼容旧版本。MINOR的变更尽可能保持源码和编译文件的兼容性，尽管可能有一些设计上的变更或者不兼容的更新。PATCH通常完全向前向后兼容，主要用于修复一些bug与缺陷。

你受影响的程序取决于你的编码的耦合性。如果你做了大量定制，那么肯定会比简单的使用命名空间进行配置受到的影响更多一些。

记得在发布新版本之前应该彻底地测试你的应用。

2.4. 获取Spring Security

你可以用几种办法来获取Spring Security。

你可以从Spring Security主页下载发布的包，也可以从Maven中央仓库（或用于发布快照和里程碑版本的Spring Maven仓库）单独下载jars，或者自己从源码来构建。

2.4.1. 获取Maven

最小的Spring Security Maven依赖设置像下面这样:

pom.xml.

```
<dependencies>
    <!-- ... other dependency elements ... -->
    <dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>spring-security-web</artifactId>
        <version>4.1.3.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>spring-security-config</artifactId>
        <version>4.1.3.RELEASE</version>
    </dependency>
</dependencies>
```

如果你需要使用如LDAP，OpenID这些附加的功能，你还需要包含它们。详见《项目模块》

Maven Repositories

所有的GA版本（即以.RELEASE结尾的版本）都会发布到Maven中央仓库，因此不需要在你的pom中声明额外的Maven仓库。

如果你要使用SNAPSHOT快照版本，那么你需要确保像下面这样声明了Spring Snapshot仓库定义：

pom.xml.

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

如果你要使用一个里程碑或候选release版本，那么你需要像下面这样声明Spring Milestone仓库:

pom.xml.

```
<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

Spring Security在Spring Framework 4.3.2.RELEASE上构建,但也可以在4.0.x上运作。很多用户会将Spring Security的依赖关系传递到Spring Framework 4.3.2.RELEASE，这可能会导致一些奇怪的classpath问题。

一种不推荐的解决办法是在你的pom文件中的 `<dependencyManagement>` 包含所有的Spring Framework模块。另一种方法是在其中包含 `spring-framework-bom`，就像下面这样:

pom.xml.

```
<dependencyManagement>
    <dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-framework-bom</artifactId>
        <version>4.3.2.RELEASE</version>
        <type>pom</type>
        <scope>import</scope>
    </dependency>
    </dependencies>
</dependencyManagement>
```

这将确保你的Spring Security的依赖使用Spring 4.3.2.RELEASE模块。

这使用了Maven的"bill of materials"(BOM)的概念，并且只能用于Maven 2.0.9或更新的版本。关于依赖如何被解析，详情可以参见 《Maven介绍-依赖机制[1]》

[1] http://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html

2.4.2. Gradle

最小的Spring Security Gradle依赖配置像下面这样：

build.gradle.

```
dependencies {
    compile 'org.springframework.security:spring-security-
web:4.1.3.RELEASE'
    compile 'org.springframework.security:spring-security-
config:4.1.3.RELEASE'
}
```

如果你需要使用诸如LDAP、OpenID之类的功能，你需要包含它们。详见《 项目模块 》。

Gradle仓库

所有的GA版本(以.RELEASE结尾的版本)都发布在Maven中央仓库，因此可以使用mavenCentral()仓库来指定GA版本。

build.gradle.

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

如果你要使用SNAPSHOT快照版本，那么你需要确保像下面这样声明了Spring Snapshot仓库定义：

build.gradle.

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

如果你要使用一个里程碑或候选release版本，那么你需要像下面这样声明Spring Milestone仓库:

build.gradle.

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

```
}
```

Spring 4.0.x与Gradle的使用

默认情况下Gradle进行版本依赖传递时会使用最新版本，这意味着如果你使用Spring Security 4.1.3.RELEASE与Spring Framework 4.3.2.RELEASE时不需要额外配置。但有时候会出现一些问题，所以建议使用Gradle的ResolutionStrategy，像下面这样：

build.gradle.

```
configurations.all {
    resolutionStrategy.eachDependency { DependencyResolveDetails details
 ->
        if (details.requested.group == 'org.springframework') {
            details.useVersion '4.3.2.RELEASE'
        }
    }
}
```

这确保了Spring Security将使用指定版本的Spring Framework依赖。

这一例子运行在Gradle 1.9，如果要在更新版本的Gradle上运行，那么可能需要适当修改。

2.4.3. 项目模块

在Spring Security 3.0中，代码库已经被切分成一些单独的jars，从而更清晰的把不同的功能点以及第三方依赖进行区分。如果你使用Maven来构建你的项目，你需要将这些模块添加到你的pom.xml中。就算你没有使用maven，我们仍然建议你参考pom.xml文件，以获取一些第三方依赖和版本号。另外，在包含的示例应用中查看版本也不失为一种好的办法。

Core - spring-security-core.jar

包含了核心认证与访问控制类型和接口，远程支持，以及提供基础APIs。任何使用Spring Security的应用都需要包含它。支持独立应用、远程客户端、方法（服务层）安全和JDBC提供用户。包含了如下顶级包：

- org.springframework.security.core

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

Remoting - spring-security-remoting.jar

提供与Spring Remoting的集成。你不需要用到这玩意儿，除非你用Spring Remoting写了个远程客户端。主要的包是`org.springframework.security.remoting`。

Web - spring-security-web.jar

包含过滤器以及和web安全相关的基础代码，它们都依赖于servlet API。如果你需要使用Spring Security的web认证服务，并且需要基于URL的访问控制，这就是你所需要的。主要的包是`org.springframework.security.web`。

Config - spring-security-config.jar

包含安全命名空间转换代码以及Java配置代码。如果你要使用Spring Security XML命名空间配置，或Spring Security的Java配置，那么你需要包含它。它主要的包是`org.springframework.security.config`。其中没有任何类是要在应用程序中直接使用的。

LDAP - spring-security-ldap.jar

LDAP认证和提供的代码。如果你需要使用LDAP认证或管理LDAP用户记录，那么就include它。顶级包是`org.springframework.security.ldap`。

ACL - spring-security-acl.jar

指定了ACL领域对象的实现。用来在你的应用中指定domain对象实例来提供安全保障。顶级包是`org.springframework.security.acls`。

CAS - spring-security-cas.jar

Spring Security的CAS客户端集成。如果你想要在CAS单点登陆服务器上使用Spring Security的web认证，那么就include它。顶级包是`org.springframework.security.cas`。

OpenID - spring-security-openid.jar

提供OpenID web认证支持。用来针对外部OpenId服务提供用户认证。`org.springframework.security.openid`依赖于OpenID4Java。

Test - spring-security-test.jar

支持Spring Security的测试。

2.4.4. 检出源码Source

在Spring Security称为一个开源项目后，我们鼓励你用git检出源码。你可以得到全部的示例应用，而且你可以很方便的构建最新版本的项目。拥有源码也可以使你方便的调试应用。异常栈不再是一个不透明的黑盒，你可以找到源码中出问题的那一行，并且查看发生了什么。源码就是项目的最终文档，你可以从中看到此项目是如何工作的。

用如下git命令获取源码：

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

你可以从中你的机器上访问整个项目的历史（包括全部的releases版本以及分支）。

3

Spring 4.1 的新特性

超过100个 RC1问题[1] 以及超过60个 RC2问题[2] 在Spring Security 4.1中被修复。

下面是它的列表：

3.1. Java配置改进

- 简化UserDetailsService的Java配置[3]
- 简化AuthenticationProvider的Java配置[4]
- 可通过LogoutConfigurer对上下文中的Negotiating LogoutSuccessHandler(s)进行配置
- 可以通过SessionManagementConfigurer对InvalidSessionStrategy进行配置
- 可以通过HttpSecurity.addFilterAt来向本地的链添加过滤器

3.2. Web应用安全改进

- MvcRequestMatcher[5]

[1] https://github.com/spring-projects/spring-security/issues?utf8=%E2%9C%93&q=milestone%3A%224.1.0+RC1%22
[2] https://github.com/spring-projects/spring-security/issues?utf8=%E2%9C%93&q=milestone%3A%224.1.0+RC2%22
[3] http://docs.spring.io/spring-security/site/docs/4.1.3.RELEASE/reference/htmlsingle/#jc-authentication-userdetailsservice
[4] http://docs.spring.io/spring-security/site/docs/4.1.3.RELEASE/reference/htmlsingle/#jc-authentication-authenticationprovider
[5] http://docs.spring.io/spring-security/site/docs/4.1.3.RELEASE/reference/htmlsingle/#mvc-requestmatcher

- Content Security Policy (CSP)[6]
- HTTP Public Key Pinning (HPKP)[7]
- CORS[8]
- CookieCsrfTokenRepository[9] 提供一个简单的AngularJS与CSRF集成方案
- 加入了 `ForwardAuthenticationFailureHandler` 与 `ForwardAuthenticationSuccessHandler`
- AuthenticationPrincipal[10] 支持属性表达，从而支持转化Authentication.getPrincipal()对象（用于处理不变的自定义用户领域对象）

3.3. 认证改进

- Web安全表达式中的Path Variables[11]
- 方法安全元注解[12]

3.4. 加密模块改进

- 通过`SCryptPasswordEncoder`支持SCrypt
- 通过 Pbkdf2PasswordEncoder[13] 支持PBKDF2
- 使用AES/CBC/PKCS5Padding以及AES/GCM/NoPadding算法，提供新的 `BytesEncryptor` 实现 BouncyCastle

[6] http://docs.spring.io/spring-security/site/docs/4.1.3.RELEASE/reference/htmlsingle/#headers-csp
[7] http://docs.spring.io/spring-security/site/docs/4.1.3.RELEASE/reference/htmlsingle/#headers-hpkp
[8] http://docs.spring.io/spring-security/site/docs/4.1.3.RELEASE/reference/htmlsingle/#cors
[9] http://docs.spring.io/spring-security/site/docs/4.1.3.RELEASE/reference/htmlsingle/#csrf-cookie
[10] http://docs.spring.io/spring-security/site/docs/4.1.3.RELEASE/reference/htmlsingle/#mvc-authentication-principal
[11] http://docs.spring.io/spring-security/site/docs/4.1.3.RELEASE/reference/htmlsingle/#el-access-web-path-variables
[12] http://docs.spring.io/spring-security/site/docs/4.1.3.RELEASE/reference/htmlsingle/#method-security-meta-annotations
[13] http://docs.spring.io/spring-security/site/docs/4.1.3.RELEASE/reference/htmlsingle/#spring-security-crypto-passwordencoders

3.5. 测试改进

- @WithAnonymousUser[14]
- @WithUserDetails[15] 允许指定 `UserDetailsService` 的bean名
- 测试元注解[16]
- 可以用 `SecurityMockMvcResultMatchers.withAuthorities` 来模拟一个 `GrantedAuthority` 列表

3.6. 通用改进

- 重新组织示例项目
- 迁移到GitHub issues

[14] http://docs.spring.io/spring-security/site/docs/4.1.3.RELEASE/reference/htmlsingle/#test-method-withanonymoususer

[15] http://docs.spring.io/spring-security/site/docs/4.1.3.RELEASE/reference/htmlsingle/#test-method-withuserdetails

[16] http://docs.spring.io/spring-security/site/docs/4.1.3.RELEASE/reference/htmlsingle/#test-method-meta-annotations

4

示例与指南(从这开始)

如果你想要看看如何开始Spring Security，最好的办法就是看看我们的示例。

表 **4.1.** 示例应用

源码	描述	手册
Hello Spring Security[1]	展示了如何基于Java配置集成Spring Security	Hello Spring Security Guide[2]
Hello Spring Security Boot[3]	展示了如何与已存在的Spring Boot应用集成	Hello Spring Security Boot Guide[4]
Hello Spring Security XML[5]	展示了如何基于XML配置集成Spring Security	Hello Spring Security XML Guide[6]
Hello Spring MVC Security[7]	展示了如何与Spring MVC集成	Hello Spring MVC Security Guide[8]

[1] https://github.com/spring-projects/spring-security/tree/4.1.3.RELEASE/samples/javaconfig/helloworld

[2] http://docs.spring.io/spring-security/site/docs/4.1.3.RELEASE/guides/html5/helloworld-javaconfig.html

[3] https://github.com/spring-projects/spring-security/tree/4.1.3.RELEASE/samples/boot/helloworld

[4] http://docs.spring.io/spring-security/site/docs/4.1.3.RELEASE/guides/html5/helloworld-boot.html

[5] https://github.com/spring-projects/spring-security/tree/4.1.3.RELEASE/samples/xml/helloworld

[6] http://docs.spring.io/spring-security/site/docs/4.1.3.RELEASE/guides/html5/helloworld-xml.html

[7] https://github.com/spring-projects/spring-security/tree/4.1.3.RELEASE/samples/javaconfig/hellomvc

源码	描述	手册
Custom Login Form[9]	展示了如何创建自定义登陆表单	Custom Login Form Guide[10]

[8] http://docs.spring.io/spring-security/site/docs/4.1.3.RELEASE/guides/html5/hellomvc-javaconfig.html

[9] https://github.com/spring-projects/spring-security/tree/4.1.3.RELEASE/samples/javaconfig/form

[10] http://docs.spring.io/spring-security/site/docs/4.1.3.RELEASE/guides/html5/form-javaconfig.html

5

Java配置

Java配置在Spring 3.1版本中被添加到Spring Framework。Spring Security从3.2版本起也开始支持Java配置，用户可以不再使用任何烦人的XML来配置此框架。

如果你熟悉《第六章 Security命名空间配置》，那么你会发现它与Security Java配置之间有很多相似之处。

Spring Security 提供大量的示例应用，其中以-jc结尾的意思是使用了Java配置方式。

5.1. Hello Web Security Java Configuration

第一步就是创建我们的Spring Security Java配置。此配置创建一个名为 `springSecurityFilterChain` 的Servlet过滤器，此过滤器将在你的应用中负责所有的安全（保护应用程序URLs，校验提交的用户名和密码，在表单中重定向登陆地址等）。下面这是最基础的Spring Security Java配置示例：

```
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 extends WebSecurityConfigurerAdapter {

    @Autowired
```

```
    public void configureGlobal(AuthenticationManagerBuilder
 auth) throws Exception {
        auth
            .inMemoryAuthentication()
                .withUser("user").password("password").roles("USER");
    }
}
```

其中configureGlobal的方法名并不重要。重要的是在注解了@EnableWebSecurity、@EnableGlobalMethodSecurity或@EnableGlobalAu的类中配置AuthenticationManagerBuilder，如果没有配置，那么会造成不可预知的结果。

你并不需要做太多配置，但实际上已经做了很多配置。你可以找到的功能列表如下：

- 在你的应用中每个URL都要求认证
- 为你生成一个登陆表单
- 允许用户在表单中提交 **Username** 用户名为*user*以及**Password**密码为*password*来进行认证
- 允许用户注销
- 防范 CSRF攻击[1]
- 防范 Session Fixation[2]
- 集成Security Header
  - 使用 HTTP Strict Transport Security[3] 保护请求
  - X-Content-Type-Options[4]集成
  - 缓存控制（可以在你的应用中重写，以允许缓存静态资源）
  - 集成 X-XSS-Protection[5]
  - 集成 X-Frame-Options 以帮助防范 Clickjacking点击劫持[6]

[1] http://en.wikipedia.org/wiki/Cross-site_request_forgery
[2] http://en.wikipedia.org/wiki/Session_fixation
[3] http://en.wikipedia.org/wiki/HTTP_Strict_Transport_Security
[4] http://msdn.microsoft.com/en-us/library/ie/gg622941(v=vs.85).aspx
[5] http://msdn.microsoft.com/en-us/library/dd565647(v=vs.85).aspx
[6] http://en.wikipedia.org/wiki/Clickjacking

- 与如下Servlet API进行整合
  - HttpServletRequest#getRemoteUser()[7]
  - HttpServletRequest.html#getUserPrincipal()[8]
  - HttpServletRequest.html#isUserInRole(java.lang.String)[9]
  - HttpServletRequest.html#login(java.lang.String, java.lang.String)[10]
  - HttpServletRequest.html#logout()[11]

5.1.1. AbstractSecurityWebApplicationInitializer

第二步是在war中注册一个`springSecurityFilterChain`。可以通过Spring的WebApplicationInitializer来进行Java配置，但需要Servlet 3.0+环境的支持。Spring Security也提供一个基础类`AbstractSecurityWebApplicationInitializer`，确保`springSecurityFilterChain`能够得到注册。两种方式的区别在于，前者主要与使用了Spring Framework的应用集成，而后者主要用于单独使用Spring Security的情形。

- 《5.1.2 AbstractSecurityWebApplicationInitializer 与非Spring项目》 - 说明了如何与非Spring项目集成
- 《5.1.3 AbstractSecurityWebApplicationInitializer 与Spring MVC项目》 - 说明了如何与Spring项目集成

5.1.2. AbstractSecurityWebApplicationInitializer与非Spring项目

如果你没有使用Spring或Spring MVC，那么你不应该使用WebSecurityConfig作为父类。示例如下：

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

[7] http://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpServletRequest.html#getRemoteUser()
[8] http://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpServletRequest.html#getUserPrincipal()
[9] http://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpServletRequest.html#isUserInRole(java.lang.String)
[10] http://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpServletRequest.html#login(java.lang.String,%20java.lang.String)
[11] http://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpServletRequest.html#logout()

```
public class SecurityWebApplicationInitializer
    extends AbstractSecurityWebApplicationInitializer {

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

SecurityWebApplicationInitializer会进行如下动作：

- 自动注册一个springSecurityFilterChain过滤器到你的应用中的每一个URL
- 添加一个ContextLoaderListener来读取WebSecurityConfig

5.1.3. AbstractSecurityWebApplicationInitializer与Spring MVC项目

如果我们在应用的别处中使用了Spring，我们可能已经拥有了一个WebApplicationInitializer来读取Spring 配置。 如果我们仍然使用上文中的配置，那么会得到一个错误。作为替代，我们在已存在的ApplicationContext中注册Spring Security。例如，如果我们使用了Spring MVC，那么SecurityWebApplicationInitializer应该像下面这样：

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

public class SecurityWebApplicationInitializer
    extends AbstractSecurityWebApplicationInitializer {
}
```

如此一来，我们仅仅是简单地为每一个URL注册了springSecurityFilterChain过滤器。然后，我们应该确保WebSecurityConfig被读取到已存在的ApplicationInitializer中。如果我们使用了Spring MVC，那么我们应该在getRootConfigClasses()中进行添加：

```
public class MvcWebApplicationInitializer extends
        AbstractAnnotationConfigDispatcherServletInitializer {

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

```
    }

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

5.2. HttpSecurity

目前为止，我们的WebSecurityConfig只包含了如何去认证我们的用户信息。那么Spring Security如何知道我们有哪些用户是需要认证的呢？Spring Security怎么知道我们想要支持基于表单的认证的呢？谜底就是WebSecurityConfigurerAdapter，它在`configure(HttpSecurity http)`方法中提供了默认的配置，就像下面这样：

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

默认的配置包括：

- 确保任何到达我们的应用的请求都需要进行用户认证
- 允许用户进行表单登录认证
- 允许用户基于HTTP Base authentication进行认证

你会注意到上面的Java代码和XML命名空间配置非常相似：

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

Java配置中的and()方法相当于XML中的结束闭包标签，它允许我们继续配置它的上一级(parent)。如果你查看代码，那么你会发现，我想要配置授权的请求、配置表单登录以及配置了HTTP Basic authentication。

注意，Java配置中URLs和参数有些不同寻常，在你创建自定义登录页面的时候你需要记住这一点，它使得我们的URLs具有RESTful的风格。另外，我们使用Spring Security时，对于 信息泄漏(information leaks)[12]的帮助不太明显。例如：

5.3. Java配置与表单登陆

当你被要求用表单进行登录时你可能会很疑惑，我们并没有编写任何HTML文件或JSPs代码。事实上，在你使用Spring Security的默认配置时，Spring Security自动生成了一个表单登录页面，并设置为其配置了URL。这一功能启用后，我们可以通过标准URL传值的方式提交登录，然后在登录成功后跳转到默认的target URL。

自动生成登录页面非常便于我们快速开始，大多数应用程序可能希望提供自己的登录页面。我们可以像下面这样更新我们的配置：

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

❶ 此设置指定了本地的登录页面

❷ 我们必须允许所有用户（包括未认证的用户）访问我们的登录页面。`formLogin().permitAll()`方法允许所有用户从任意表单页面提交登录信息

下面是一个JSPs的登录页面实现：

下面的登录页面只能用于我们目前的配置。如果我们不需要某些默认配置，那么我们可以很容易地更新它们。

```
<c:url value="/login" var="loginUrl"/>
<form action="${loginUrl}" method="post">          ❶
    <c:if test="${param.error != null}">           ❷
```

[12] https://www.owasp.org/index.php/Information_Leak_(information_disclosure)

```
        <p>
            Invalid username and password.
        </p>
    </c:if>
    <c:if test="${param.logout != null}">          ❸
        <p>
            You have been logged out.
        </p>
    </c:if>
    <p>
        <label for="username">Username</label>     ❹
    </p>
    <p>
        <label for="password">Password</label>     ❺
    </p>
    <input type="hidden"                           ❻
        name="${_csrf.parameterName}"
        value="${_csrf.token}"/>
    <button type="submit" class="btn">Log in</button>
</form>
```

❶ 我们需要向/login这个地址提交一个POST请求，来进行权限认证

❷ 如果检查到error参数，那么意味着认证失败了

❸ 如果检查到logout参数，那么意味着我们注销成功了

❹ 用户名必须使用名为username的HTTP参数

❺ 密码必须使用命名为password的HTTP参数

❻ 参见《引入CSRF Token》。想要学习更多相关知识，还可以参考《跨站请求伪造Cross Site Request Forgery (CSRF)》

5.4. 请求授权

我们的示例应用目前对每一个URL都需要用户进行认证。我们可以通过`http.authorizeRequests()`方法指定多个需要认证的URLs：

```
protected void configure(HttpSecurity http) throws Exception {
    http
        .authorizeRequests() ❶
            .antMatchers("/resources/**", "/signup", "/
about").permitAll() ❷
            .antMatchers("/admin/**").hasRole("ADMIN") ❸
```

```
            .antMatchers("/db/**").access("hasRole('ADMIN') and
 hasRole('DBA')") ❹
            .anyRequest().authenticated() ❺
            .and()
        // ...
        .formLogin();
}
```

❶ http.authorizeRequests()方法具有多个matcher子节点，每一个matcher都展示了其自身对应的路径与权限的匹配。

❷ 我们指定了多个用户可以访问的URL。 patterns。任何用户都可以访问以"/resources/"开头的URL、以及"/signup"、以及"/about"。

❸ 任何以"/admin/"开头的URL都必须具有"ROLE_ADMIN"的角色。我们注意到hasRole方法的参数里面没有以"ROLE__"前缀开头，因为Spring Security会自动在"ADMIN"的前面加上此前缀。

❹ 所有以"/db/"开头的URL需要用户同时具有"ROLE_ADMIN"和"ROLE_DBA"的角色。同样，他俩没有用"ROLE__"作为前缀（默认添加了此前缀）。

❺ 其它任何没有进行匹配的URLs只需要用户认证过即可访问。

5.5. 注销操作

在我们使用了WebSecurityConfigurerAdapter的时候，注销功能就已经自动添加了。当用户访问/logout路径时，系统会默认执行下面一些动作：

- 使 HTTP Session 无效
- 清除任何已经配置的RemenberMe认证
- 清理 SecurityContextHolder
- 重定向到 /login?logout

与配置登录类似，有时候你需要进一步定制注销的一些参数：

```
protected void configure(HttpSecurity http) throws Exception {
    http
        .logout()                                                    ❶
            .logoutUrl("/my/logout")                                 ❷
            .logoutSuccessUrl("/my/index")                           ❸
            .logoutSuccessHandler(logoutSuccessHandler)              ❹
```

```
            .invalidateHttpSession(true)                    ❺
            .addLogoutHandler(logoutHandler)                ❻
            .deleteCookies(cookieNamesToClear)              ❼
            .and()
        //...
}
```

❶ 提供注销支持，它会在WebSecurityConfigurerAdapter中自动使用。

❷ 设置触发注销的URL（默认是/logout）。如果CSRF保护已经启用（默认），那么请求方式必须是POST。更多信息参见 JavaDoc[13]

❸ 设置在注销成功后URL重定向的地址。默认是/login?logout。更多信息参见 JavaDoc[14]

❹ 指定一个自定义LogoutSuccessHandler，加入此配置后logoutSuccessUrl()方法会被忽略。更多信息参见 JavaDoc[15]

❺ 指定注销时是否要让HttpSession无效化，默认为true。若配置了SecurityContextLogoutHandler则会将其覆盖。参见 JavaDoc[16]

❻ 添加一个LogoutHandler。默认情况下SecurityContextLogoutHandler会作为最后一个LogoutHandler被添加进来。

❼ 允许指定在注销成功后需要清除的cookies的名字。这比起添加一个CookieClearingLogoutHandler要简单得多。

注销也可以用XML命名空间方式来进行配置。详情参见 Spring Security XML命名空间一节中的 注销元素logout element[17]

通常，为了深度定制注销功能，你可以添加 LogoutHandler[18] 和/或 LogoutSuccessHandler[19] 的实现。在大多数情况下，使用fluent API时，这些handlers会被覆盖。

[13] http://docs.spring.io/spring-security/site/docs/current/apidocs/org/springframework/security/config/annotation/web/configurers/LogoutConfigurer.html#logoutUrl(java.lang.String)
[14] http://docs.spring.io/spring-security/site/docs/current/apidocs/org/springframework/security/config/annotation/web/configurers/LogoutConfigurer.html#logoutUrl(java.lang.String)
[15] http://docs.spring.io/spring-security/site/docs/current/apidocs/org/springframework/security/config/annotation/web/configurers/LogoutConfigurer.html#logoutUrl(java.lang.String)
[16] http://docs.spring.io/spring-security/site/docs/current/apidocs/org/springframework/security/config/annotation/web/configurers/LogoutConfigurer.html#logoutUrl(java.lang.String)
[17] http://docs.spring.io/spring-security/site/docs/4.1.3.RELEASE/reference/htmlsingle/#nsa-logout
[18] http://docs.spring.io/spring-security/site/docs/current/apidocs/org/springframework/security/web/authentication/logout/LogoutHandler.html

5.5.1. LogoutHandler

通常如果一个class实现了 LogoutHandler[20] 接口，那么说明它能够参与注销操作。调用它们时，它们应该执行一些必要的清理操作，你不应该在其中抛出异常。我们提供了如下一些实现：

- PersistentTokenBasedRememberMeServices[21]
- TokenBasedRememberMeServices[22]
- CookieClearingLogoutHandler[23]
- CsrfLogoutHandler[24]
- SecurityContextLogoutHandler[25]

参见《 17.4 Remember-Me接口与实现[26]》

我们也为直接实现`LogoutHandler`提供了一个替代，你可以直接使用fluent API来单独覆盖`LogoutHandler`的实现。例如`deleteCookies()`方法允许指定多个cookies名，从而再注销成功后将它们都清理干净。你也可以添加一个`CookieClearingLogoutHandler`来实现同样的操作，但显然前者更方便一些。

[19] http://docs.spring.io/spring-security/site/docs/current/apidocs/org/springframework/security/web/authentication/logout/LogoutSuccessHandler.html
[20] http://docs.spring.io/spring-security/site/docs/current/apidocs/org/springframework/security/web/authentication/logout/LogoutHandler.html
[21] http://docs.spring.io/spring-security/site/docs/current/apidocs/org/springframework/security/web/authentication/rememberme/
[22] http://docs.spring.io/spring-security/site/docs/current/apidocs/org/springframework/security/web/authentication/rememberme/TokenBasedRememberMeServices.html
[23] http://docs.spring.io/spring-security/site/docs/current/apidocs/org/springframework/security/web/authentication/logout/CookieClearingLogoutHandler.html
[24] http://docs.spring.io/spring-security/site/docs/current/apidocs/org/springframework/security/web/csrf/CsrfLogoutHandler.html
[25] http://docs.spring.io/spring-security/site/docs/current/apidocs/org/springframework/security/web/authentication/logout/SecurityContextLogoutHandler.html
[26] http://docs.spring.io/spring-security/site/docs/4.1.3.RELEASE/reference/htmlsingle/#remember-me-impls

5.5.2. LogoutSuccessHandler

`LogoutSuccessHandler`会在注销成功时被`LogoutFilter`进行调用，以执行一些诸如 redirection 或 forwarding 之类的操作。此接口与 LogoutHandler 非常相似，不过它允许抛出异常。

我们已经提供了如下一些实现：

- SimpleUrlLogoutSuccessHandler[27]
- HttpStatusReturningLogoutSuccessHandler

就像上面提到的一样，你不需要直接指定 SimpleUrlLogoutSuccessHandler ，使用fluent API来的`logoutSuccessUrl()`方法也是一种方便的选择，此方法会装入一个 SimpleUrlLogoutSuccessHandler ，在注销成功后将会redirected到你提供的 URL 中。默认URL是`/login?logout`。

而`HttpStatusReturningLogoutSuccessHandler`在REST API的场景中非常有趣。作为注销成功后需要跳转到一个URL的替代，这个`LogoutSuccessHandler`允许你返回一个简单的HTTP状态码。如果你啥都没配置，那么默认会返回200状态。

5.5.3. 更多注销相关的说明

- Logout Handling[28]
- Testing Logout[29]
- HttpServletRequest.logout()[30]
- 17.4 Remember-Me接口与实现[31]
- CSRF Caveats 中的 注销[32]

[27] http://docs.spring.io/spring-security/site/docs/current/apidocs/org/springframework/security/web/authentication/logout/SimpleUrlLogoutSuccessHandler.html
[28] http://docs.spring.io/spring-security/site/docs/4.1.3.RELEASE/reference/htmlsingle/#ns-logout
[29] http://docs.spring.io/spring-security/site/docs/4.1.3.RELEASE/reference/htmlsingle/#test-logout
[30] http://docs.spring.io/spring-security/site/docs/4.1.3.RELEASE/reference/htmlsingle/#servletapi-logout
[31] http://docs.spring.io/spring-security/site/docs/4.1.3.RELEASE/reference/htmlsingle/#remember-me-impls
[32] http://docs.spring.io/spring-security/site/docs/4.1.3.RELEASE/reference/htmlsingle/#csrf-logout

- 单点注销Single Logout[33] (CAS protocol)
- Spring Security XML命名空间中的 注销元素logout element[34]

5.6. 认证

到目前为止，我们只看到了最基本的认证配置。现在，是时候开始告诉你一点高级的认证配置了！

5.6.1. 内存中的身份认证

我们已经看过了如何为单一用户配置在内存中的认证，下面是多个用户的配置示例：

```
@Autowired
public void configureGlobal(AuthenticationManagerBuilder auth) throws
 Exception {
    auth
        .inMemoryAuthentication()
            .withUser("user").password("password").roles("USER").and()

 .withUser("admin").password("password").roles("USER", "ADMIN");
}
```

5.6.2. JDBC身份认证

你可以找到基于JDBC身份认证的更新。下面的例子假设你已经在应用中定义了一个DataSource。 jdbc-javaconfig[35]项目提供了一个完整的基于JDBC身份认证的示例程序。

```
@Autowired
private DataSource dataSource;

@Autowired
public void configureGlobal(AuthenticationManagerBuilder auth) throws
 Exception {
    auth
```

[33] http://docs.spring.io/spring-security/site/docs/4.1.3.RELEASE/reference/htmlsingle/#cas-singlelogout
[34] http://docs.spring.io/spring-security/site/docs/4.1.3.RELEASE/reference/htmlsingle/#nsa-logout
[35] https://github.com/spring-projects/spring-security/tree/master/samples/javaconfig/jdbc

```
        .jdbcAuthentication()
            .dataSource(dataSource)
            .withDefaultSchema()
            .withUser("user").password("password").roles("USER").and()

 .withUser("admin").password("password").roles("USER", "ADMIN");
}
```

5.6.3. LDAP身份认证

你可以找到基于LDAP身份认证的更新。 ldap-javaconfig[36]项目提供了一个完整的基于LDAP身份认证的示例程序。

```
@Autowired
private DataSource dataSource;

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

上面的例子使用了下面的LDIF，以及一个内嵌的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
```

[36] https://github.com/spring-projects/spring-security/tree/master/samples/javaconfig/ldap

```
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
```

5.6.4. AuthenticationProvider

你可以提供一个实现了AuthenticationProvider的bean来定制自己的认证机制。下面的例子展示了如何定制认证机制，假设SpringAuthenticationProvider实现了 AuthenticationProvider 接口：

注意

这仅用于 AuthenticationManagerBuilder 不存在的情况下！

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

```
}
```

5.6.5. UserDetailsService

你可以提供一个实现了`UserDetailsService`的bean来定制自己的认证机制。下面的例子展示了如何定制认证机制，假设`SpringDataUserDetailsService`实现了`UserDetailsService`：

注意

这只用于 AuthenticationManagerBuilder 不存在，并且没有 AuthenticationProviderBean 被定义的情形！

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

你也可以通过提供一个`PasswordEncoder`的bean来定制密码如何编码。如果你要使用bcrypt，你可以像下面这样：

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

5.6.6. LDAP身份认证

EMPTY

官方文档此节也为空，节名和上面有所重复，此为官方缺漏。 ----译者注

5.7. Multiple HttpSecurity

我们可以配置多个 HttpSecurity 实例，就像我们可以配置多个 <http>块一样。键值多次集成自 WebSecurityConfigurationAdapter。下面一个示例，展示了对以`/api/`开头的URL的不一样的配置：</http>

```
@EnableWebSecurity
```

```
public class MultiHttpSecurityConfig {
    @Autowired
    public void configureGlobal(AuthenticationManagerBuilder auth) {
  ❶
        auth
            .inMemoryAuthentication()

 .withUser("user").password("password").roles("USER").and()

 .withUser("admin").password("password").roles("USER", "ADMIN");
    }

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

    @Configuration
  ❹
    public static class FormLoginWebSecurityConfigurerAdapter extends
 WebSecurityConfigurerAdapter {

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

❶ 像往常一样配置身份认证

❷ 创建一个WebSecurityConfigurerAdapter的实例，@Order指定了加载顺序

❸ `http.antMatcher` 声明了这个 HttpSecurity 只应用于以`/api/`开头的URLs

❹ 创建另一个`WebSecurityConfigurerAdapter`实例。如果URL不以`/api/`开头，那么此配置将被启用。它会在`ApiWebSecurityConfigurationAdapter`之后进行加载，因为它的默认`@Order`值会比所有已配置的值更大（最晚加载）。

5.8. Method Security

从2.0版本开始，Spring Security已经支持添加方法级的安全控制。提供JSR-250注解安全的支持，就像框架最初的 `@Secured` 注解一样。从3.0版本开始，你可以开始运用新的 基于表达式的注解expression-based annotations[37]。你给单个bean添加你的安全操作，也可以使用方法拦截元素来装饰bean的声明，还可以用AspectJ风格的切点(pointcuts)横切整个service层来为多个beans添加安全操作。

5.8.1. EnableGlobalMethodSecurity

我们可以用`@EnableGlobalMethodSecurity`注解到任何`@Configuration`实例来启用基于注解的安全配置。下面这个示例展示了如何启用Spring Security的`@Secured`注解：

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

添加注解到方法（或class、或interface）可以限制方法的访问。Spring Security本身的注解支持为方法定义一套属性，这将传递到`AccessDecisionManager` 进行最终的判定。

```
public interface BankService {

@Secured("IS_AUTHENTICATED_ANONYMOUSLY")
public Account readAccount(Long id);
```

[37] http://docs.spring.io/spring-security/site/docs/4.1.3.RELEASE/reference/htmlsingle/#el-access

```
@Secured("IS_AUTHENTICATED_ANONYMOUSLY")
public Account[] findAccounts();

@Secured("ROLE_TELLER")
public Account post(Account account, double amount);
}
```

可以像下面这样启用对JSR-250注解的支持：

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

这基于标准，并且允许应用简单的基于角色的限制，但它不如Spring Security本身的注解这么强大。为了使用基于表达式(expression-based)的语法，你要像下面这样：

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

等价的Java代码如下：

```
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);
}
```

5.8.2. GlobalMethodSecurityConfiguration

有时候你需要执行的操作比`@EnableGlobalMethodSecurity`注解所允许的更加复杂。为了应对这一情形，你可以继承`GlobalMethodSecurityConfiguration`，并

确保你的子类注解了 @EnableGlobalMethodSecurity。例如，你如果想提供一个定制的MethodSecurityExpressionHandler，你可以像下面这样：

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

更多关于能够重写的methods的信息，可以参见GlobalMethodSecurityConfiguration的Javadoc。

5.9. Post Processing Configured Objects

Spring Security的Java配置并不暴露已配置的每个对象的每个参数。这会简化绝大部分用户的配置。毕竟如果每个参数都暴露了，那么用户可能就会使用标准的bean配置了。

尽管不直接暴露每个参数的有很多好处，但用户仍然可能需要一些更高级的配置选项。Spring Security展示了ObjectPostProcessor的概念，它可以用来更改或替换许多由Java配置创建的对象实例。例如，如果你想要配置filterSecurityPublishAuthorizationSuccess参数到FilterSecurityInterceptor中，你可以像下面这样：

```
@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;
```

```
                }
            });
}
```

6

Security Namespace Configuration

6.1. 简介

命名空间配置从Spring Framework 2.0版本开始就可以使用。它允许你利用附加的XML Schema元素对传统的Spring beans应用上下文语法进行补充。你从Spring Reference Documentation[1] 中找到更多信息。命名空间元素允许简洁地配置一个单例的bean，更厉害的是，它还可以定义一种可替代的配置语法，从而更贴近问题领域，并对用户隐藏复杂的细节。一个简单的元素可以隐藏多个beans，还可以隐藏它们被添加到应用上下文的处理过程。例如，从security命名空间中添加下面这个元素到应用上下文中，可以向应用程序中开启一个嵌入了LDAP的服务器来进行测试：

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

这与用beans来连接Apache Directory Server非常相似。最通用的可选配置需要一些ldap-server元素属性的支持，用户不用再去考虑哪些beans需要创建、bean参数名如何配置。如果你的XML编辑器足够好，那么它能够向你提示有哪些属性或元素是可用的。我们非常建议你试试Spring Tool Suite，它可以帮助你更好的在标准的Spring namespaces环境下工作。

你可以从《 29. LDAP认证[2]》中查看更多关于使用ldap-server元素的说明。

[1] http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/
[2] http://docs.spring.io/spring-security/site/docs/4.1.3.RELEASE/reference/htmlsingle/#ldap

为了在你的应用中启用security命名空间，你需要在你的classpath中包含spring-security-config的jar包。然后你需要在你的应用上下文文件中加入schema声明：

```
<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>
```

在很多例子中你可以看到（sample applications里面也有），我们经常使用security而不是 beans作为默认的命名空间，这意味着我们可以省略所有的security 命名空间元素的前缀，使内容变得更易读。如果你的应用上下文分离成多个单独的文件，并且其中一个具有大量的security配置，那么你应该也会想要这么做。像下面这样开头：

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

本章后续说明都以这一配置为假设前提。

6.1.1. Namespace的设计

Namespace的设计目标是尽可能通用，同时提供一种简单方便的语法来集成到应用当中。此设计基于大量的framework依赖，并且可以分为如下几个部分：

- Web/HTTP Security - 这是最复杂的部分。用于建立过滤器，并关联用于framework认证机制的服务beans，保护URLs，增加登录和错误页面，也包括其它一些相关的东东。
- Business Object (Method) Security - 服务层的安全选项
- AuthenticationManager - 处理框架其它部分的认证请求
- AccessDecisionManager - 为web和方法安全提供访问判定。Spring会注册一个默认的AccessDecisionManager，淡然，你也可以用Spring bean语法来自己定制一个。
- AuthenticationProviders - 认证管理器认证用户的机制。这一命名空间提供一些标准的支持选项以及用传统语法添加自定义beans的工具。
- UserDetailsService - 和认证提供者紧密相关，但也经常被一些其它的beans所需要。 我们可以在后续章节看看如何进行配置。

6.2. 开始Security Namespace配置

在这一节中，我们将看到如何构建一个命名空间配置，以使用此框架的一些主要功能。假设你现在想要在已存在的web应用中，尽可能快速的运行并加入认证支持和访问控制，并测试一下登录功能。我们可以看到如何将一个数据库调换为安全仓库。在本节的最后，我们会介绍更多高级的配置选项。

6.2.1. web.xml 配置

你需要做的第一件事情就是在web.xml文件中加入下列声明：

```
<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>
```

它提供了一个到Spring Security web的挂钩。`DelegatingFilterProxy`是一个Spring Framework类，它声明了一个过滤器的实现，这个实现需要在你的Spring bean中进行定义。在这个例子中，bean命名为`springSecurityFilterChain`，它

是用于创建web安全操作的基础内部类。注意，你自己不要使用这个名字。一旦你将它添加到你的web.xml，说明你已经准备好编辑你的应用上下文文件。Web Security服务用<http>元素来进行配置。

6.2.2. 最简单的<http>配置

要启用web security，你需要像下面这样开始配置：

```
<http>

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

上述配置说明:我们想要所有的URLs都将被保护，并且需要ROLE_USER的角色才能访问它们；我们想要用一个表单提交username和password，从而登录到应用中；我们想要注册一个注销URL，从而允许我们在应用中进行注销操作。<http>元素是所有相关web命名空间功能的父节点。 <intercept-url>元素定义了一个pattern，它用ant path风格语法来将接入请求与URLs进行匹配。你也可以选择用正则表达式来进行匹配（参见namespace附录获取更多信息）。access属性定义了访问给定pattern路径所需要的条件，在默认情况下可以是用逗号分割的角色列表，发起请求的用户必须拥有其中一种角色。“ROLE_”前缀是一个标记，它说明应该创建一个简单的关于用户授权的匹配。换言之，一个常用的基于角色的检查应该被启用。Spring Security的访问控制并不限于简单的角色控制（因此使用前缀来区分不同的安全属性类型）。我们往后将看到interpretation如何改变脚注:[。在access属性中，用逗号分隔的interpretation的值依赖于被使用的 -1- 的实现。在Spring Security 3.0中，此属性也可以用 -2- 来填充。

你可以使用多个元素来定义不同的访问条件到不同的URLs上，但它们会按顺序从上到下依次匹配，因此你最好将最特殊的条件放到最上面。 你也可以添加特定于HTTP的方法属性（GET，POST，PUT，ect.）限制。

你可以添加几个用户到命名空间里面作为测试用的数据：

```
<authentication-manager>
<authentication-provider>
    <user-service>
```

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

如果你熟悉此框架的pre-namespace版本，你也可以大致猜到这里发生了什么。<http>元素负责创建一个FilterChainProxy以及它使用的filter beans。因为过滤器的位置被预定义，像过滤器顺序出不正确这类常见的问题就不再是问题。

<authentication-provider>元素创建了一个DaoAuthenticationProvider bean，<user-service>元素创建了一个InMemoryDaoImpl。所有的authentication-provider元素都必须是<authentication-manager>的子元素。authentication-provider创建了一个ProviderManager并且为它注册了认证提供器。你可以在 namespace附录[3] 中找到更多被创建的bean的信息。如果你想要理解框架中重要的类型，并且理解怎样使用它们，查阅文档非常重要，尤其是当你需要进行深入定制的时候。

6.2.3. 表单与基础登录选项

你也许会好奇登录表单是从哪里来的，我们从没有配置过HTML文件和JSPs。事实上，我们不必明确为登录页面设定URL，Spring Security会自动生成一个，从而使我们能够使用标准的URL传值法提交登录请求，用户会在登录后跳转到默认的目标URL地址。，然而，命名空间提供了丰富的支持，允许你自定义这些选项。例如，如果你想要使用你自己的登录页面，你可以像下面这样：

```
<http>


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

注意，我们加入了一个intercept-url元素，说明了任何匿名用户对登录页面的请求都是可用的，同时也指定了AuthenticatedVoter[4]会

[3] http://docs.spring.io/spring-security/site/docs/4.1.3.RELEASE/reference/htmlsingle/#appendix-namespace

[4] http://docs.spring.io/spring-security/site/docs/4.1.3.RELEASE/reference/htmlsingle/#authz-authenticated-voter

以IS_AUTHENTICATED_ANONYMOUSLY的方式来处理。其它的匹配了"/**" pattern的请求则没有办法匹配登录页面。这是通用配置错误，这将成为应用的死循环。如果你的登录页面被保护起来了，那么Spring Security会在日志中发出警告。也有这么一种可能，所有的请求都匹配一个特殊的pattern来完全绕开security过滤器链，这时候你需要像下面这样单独地定义一个http元素：

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

<http use-expressions="false">

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

从Spring Security 3.1开始，可以使用多个http元素，为不同的请求patterns分别定义单独的security过滤器链配置。如果pattern属性在http元素中被省略，那么就会匹配全部的请求。创建一个不安全的pattern非常简单，只需要配置一个空的过滤器链即可。我们可以在 Security过滤器链[5]一节中看到更多相关信息

这些不安全的请求将被Spring Security web相关配置或例如requires-channel的附加属性彻底遗忘，因此在请求期间，你将不能访问当前用户信息或调用安全的方法，认识到这点非常重要。如果你仍然想要添加security过滤器链，那么将access属性设置为IS_AUTHENTICATED_ANONYMOUSLY即可。

如果你想要使用基本的认证而不是表单登录，可以像下面这样配置：

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

<http-basic />
</http>
```

基本认证将优先，当用户试图访问被保护的资源时，会被提示需要登录。在此配置中，如果你希望使用表单登录，那么它依然可用，例如将登录表单嵌入另一个web页面。

[5] http://docs.spring.io/spring-security/site/docs/4.1.3.RELEASE/reference/htmlsingle/#filter-chains-with-ns

修改Login跳转页面

如果表单登录成功后，用户没有被引导到一个被保护的资源页面，那么default-target-url属性可以帮助你。用户会在登录成功后跳转到这个URL，它默认是"/"。你也可以通过设置always-use-default-target 为true，使用户总是跳转到此页面（不论登录是否被要求或者它们是否明确的在登录中选择）。如果你的应用程序需要用户从"home"页面开始，那么它非常有用：

```
<http pattern="/login.htm*" security="none"/>
<http use-expressions="false">

<form-login login-page='/login.htm' default-target-url='/home.htm'
        always-use-default-target='true' />
</http>
```

为了给目标地址增加更多的控制，你可以用authentication-success-handler-ref属性替代default-target-url。这个引用bean是一个AuthenticationSuccessHandler实例。你会发现在 核心过滤器[6]或者namespace附录中找到更多相关信息，它们也包含了关于定制认证失败时的处理操作的相关信息。

6.2.4. 注销操作

logout元素为注销后导航到一个特定的URL提供了支持。默认的注销URL是/logout，但你也可以通过logout-url属性来设置它。更多信息参见namespace附录。

6.2.5. 使用另外的认证提供器

在实践中，你会需要可扩展的用户信息源，而不止是一些加到应用上下文文件上的名字。最可能的是你想要在诸如数据库或LDAP服务器上储存你的用户信息。LDAP命名空间配置在LDAP一章进行介绍，此章暂不讨论。如果你想要定制Spring Security的UserDetailsService实现，在你的应用上下文中调用"myUserDetailsService"，那么你可以像下面这样：

```
<authentication-manager>
```

[6] http://docs.spring.io/spring-security/site/docs/4.1.3.RELEASE/reference/htmlsingle/#form-login-flow-handling

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

如果你想要使用数据库，你可以像这样：

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

当一个DataSource在应用上下文中被命名为"securityDataSource"，意味着在数据库容器中包含了标准的Spring Security用户数据表。或者你也可以配置一个Spring Security JdbcDaoImpl的bean，来在user-service-ref属性中使用它

```
<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>
```

你也可以像下面这样使用标准的AuthenticationProvider beans：

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

实现了AuthenticationProvider接口的bean在应用上下文中被命名为myAuthenticationProvider。你可以使用多个authentication-provider元素，它们需要被按顺序声明。参见 6.6 认证管理与命名空间[7]一节，它展示了Spring Security AuthenticationManager如何使用命名空间进行配置。

[7] http://docs.spring.io/spring-security/site/docs/4.1.3.RELEASE/reference/htmlsingle/#ns-auth-manager

添加一个Password Encoder

密码都应该被用一种安全的哈希算法进行编码（非标准SHA或者MD5算法）。<password-encoder>元素可以支持此功能。为了给密码进行bcrypt编码，之前的认证provider配置应该像下面这样：

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

Bcrypt在大部分情况下是一种不错的选择，除非你的遗留系统迫使你使用一种不同的算法。如果你使用了一种简单的哈希算法， 或者你直接存储了密码明文，那么你应该考虑将其迁移到更安全选择，例如bcrypt算法。

6.3. 高级Web功能

6.3.1. Remember-Me认证

参见 Remember-Me章节[8] 中关于remember-me命名空间配置的说明。

6.3.2. 添加HTTP/HTTPS渠道安全

如果你的应用同时支持HTTP和HTTPS，并且你需要指定URLs只能通过HTTPS进行访问，那么可以直接使用<intercept-url>的requires-channel属性进行配置。

[8] http://docs.spring.io/spring-security/site/docs/4.1.3.RELEASE/reference/htmlsingle/#remember-me

```
<http>


...
</http>
```

通过适当的配置，如果用户试图通过HTTP方式访问任何匹配了"/secure/**"的pattern，它们首先会被重定向为HTTPS方式。可供选择的选项包括"http"、"https"或"any"。使用"any"意味着HTTP或者HTTPS都可以被使用。

如果你的应用程序使用了非标准的HTTP和/或HTTPS端口，那么你可以指定端口映射：

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

注意，为了确保安全，应用不应该全都使用HTTP或者二者同时启用。尽可能使用HTTPS（用户需要访问HTTPS的URL），并使用安全的连接方式，从而避免有可能发生的中间人攻击。

6.3.3. Session管理

超时检测

你可以配置Spring Security来监测无效的session ID提交，并将用户重定向到适当的URL。下面是一个使用`session-management`的实现：

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

注意如果你使用了这个机制来检查session超时，如果用户注销后没有关闭浏览器又重新进行登录，那么可能会错误地记录一个错误。当你使session无效时，session cookies可能没有被清理，那么它就会在用户注销后被重新提交。你可以在注销时明确的删除JSESSIONID cookie，就像下面这样：

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

不幸的是这不能保证在所有的servlet容器中都启作用，因此你需要在你的环境中进行测试。

如果你通过一个代理来运行你的应用，你也许也可以通过代理服务器的配置来清除session cookie。例如，假如你使用Apache HTTPD的mod_headers，下面这条指令可以通过在注销成功后的报文头中添加一些内容，来清除JSESSIONID cookie。

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

并发的session控制

如果你希望为单个用户登陆你系统的能力添加限制，Spring Security支持像下面这样进行扩展。首先你需要添加一个listener到你的web.xml文件，来保证Spring Security可以能够被session的生命周期事件更新。

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

在应用上下文中增加几行：

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

这将阻止用户多次登陆——第二个登陆会让第一个登陆无效。通常你会希望拒绝二次登陆，你可以像这样：

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

这样第二次登陆会被拒绝。这意味着如果基于表单的登陆启用了，那么第二次登陆的用户会被导航到authentication-failure-url。如果第二次认证是非交互式的机制，例如"remember-me"，那么会返回一个"未被授权"(401)的错误。如果你想替换这个错误页面，你可以在session-management元素里加入一个session-authentication-error-url属性。

如果你在使用自定义的基于表单的认证过滤器，那么你必须明确地配置并发session控制。详情参见 session管理[9] 一章。

Session固化攻击的保护

Session固化攻击是一种潜在风险，恶意的攻击者可能在访问站点的时候创建一个session，然后引导其他用户用同样的session进行登陆（例如给他们发送一个包含session身份参数的连接）。Spring Security自动地防止这种攻击，在用户登陆的时候会自动创建一个新的session，并改变sessionID。你可以通过<session-management>中的session-fixation-protection属性控制此行为，有以下四种选项：

- none - 不做任何事情，原来的session会被保留。
- newSession - 创建一个全新的干净的session，不会拷贝原有session的数据（但Spring Security相关的属性除外会被拷贝）
- migrateSession - 创建一个新的session，同时拷贝所有旧的session属性到新session中。这是servlet 3.0或更早的版本的默认值。
- changeSessionId - 不创建新的session。作为替代，使用Servlet容器提供的session固化保护功能（HttpServletRequest#changeSessionId()）。这个选

[9] http://docs.spring.io/spring-security/site/docs/4.1.3.RELEASE/reference/htmlsingle/#session-mgmt

项只在Servlet 3.1（Java EE 7）或者更新版本的容器上可用并且是默认值，在老版本的容器中使用此选项会抛出异常。当session固话攻击protection发生时，它导致SessionFixationProtectionEvent会被发布到应用上下文中。如果你使用changeSessionId，这个protection也会导致javax.servlet.http.HttpSessionIdListener被通知，因此如果你的代码同时监听这两个事件时要小心。Session管理[10]章节中有更多详情。

6.3.4. OpenID支持

命名空间支持OpenID登陆，或者附加到通常的基于表单登陆中：

```
<http>

<openid-login />
</http>
```

你应该用一个OpenID提供者（例如myopenid.com），然后用<user-service>在内存中添加用户信息：

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

你应该能够用myopenid.com来认证。为了使用OpenID，可以通过在openid-login元素中设置user-service-ref属性，来选择一个特定的UserDetailsService bean。参见前一节的认证提供器。注意我们已经在上面的用户配置中省略了密码属性，这个用户数据的设置只被用于为用户读取授权信息。一个随机的密码会在内部生生成，防止你应用的别处意外的使用这个用户数据的数据源配置。

属性调换

支持OpenID属性调换。下面这个配置展示了如何从OpenID提供器中检索email和全名，以供应用系统使用：

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

[10] http://docs.spring.io/spring-security/site/docs/4.1.3.RELEASE/reference/htmlsingle/#session-mgmt

```
    <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属性的类型都是一个URI，由特定的schema决定，在这个例子中是http://axschema.org/。如果属性必须为成功认证所引导，那么可以设置需要的属性。准确的schema和属性的支持依赖于你的OpenID提供器。属性值作为认证程序的一部分返回，可以像下面这样访问：

```
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）。我们可以在技术概况中的Spring Security组件核心里面找到更多关于SecurityContextHolder类如何使用的信息。如果你需要使用多个身份providers，多个属性值调换配置也被支持。你可以应用多个`attribute-exchange`元素，对每个元素都使用一个`identifier-matcher`属性。它包含一个正则表达式，来为用户匹配OpenID标识。参见OpenID示例，它为Google、Yahoo和MyOpenID providers提供了不同的属性清单。

6.3.5. 响应头

关于怎样自定义头元素的附加信息，可以参考 第20节Security HTTP 响应头[11]。

6.3.6. 添加你自己的过滤器

如果你之前已经用过Spring Security，那么你应该会知道框架维持一个过滤器链，从而提供安全服务。也许你需要添加你自己的过滤器到指定的本地的栈上，或者使用不是当前命名空间的Spring Security过滤器配置选项（例如CAS）。或者，你可能需要使用标准命名空间过滤器的一个自定义版本，例如`UsernamePasswordAuthenticationFilter`由<form-login>元素创建，附加配置选项的好处是能够明确的使用bean。

[11] http://docs.spring.io/spring-security/site/docs/4.1.3.RELEASE/reference/htmlsingle/#headers

过滤器链没有直接暴露，那你该如何通过命名空间配置使用它呢？

使用命名空间时，过滤器的顺序总是强制施行的。当应用上下文被创建，过滤器bean就已经被命名空间操作代码排好顺序，同时标准的Spring Security过滤器在命名空间中都拥有别名以及一个明确的位置。

> 在以前的版本中，排序发生在过滤器实例被创建之后，应用上下文后处理期间。在3.0+版本，排序完成在bean元数据等级，并且在classes被实例化之前。你可以在栈中嵌入你自己的过滤器，就像整个过滤器清单，它必须在`<http>`元素被解析时被定义。因此3.0中的语法进行了适当的改变。

创建了过滤器的过滤器别名和命名空间元素/属性在 表格6.1 标准过滤器别名与顺序[12] 中展示。过滤器根据它们在过滤器链中的位置被展示。

表 **6.1.** 标准过滤器别名与顺序

Alias	Filter Class	Namespace Element or Attribute
CHANNEL_FILTER	ChannelProcessingFilter	http/intercept-url@requires-channel
SECURITY_CONTEXT_FILTER	SecurityContextPersistenceFilter	http
CONCURRENT_SESSION_FILTER	ConcurrentSessionFilter	session-management/concurrency-control
HEADERS_FILTER	HeaderWriterFilter	http/headers
CSRF_FILTER	CsrfFilter	http/csrf
LOGOUT_FILTER	LogoutFilter	http/logout
X509_FILTER	X509AuthenticationFilter	http/x509
PRE_AUTH_FILTER	AbstractPreAuthenticatedProcessingFilter Subclasses	N/A
CAS_FILTER	CasAuthenticationFilter	N/A

[12] http://docs.spring.io/spring-security/site/docs/4.1.3.RELEASE/reference/htmlsingle/#filter-stack

Alias	Filter Class	Namespace Element or Attribute
FORM_LOGIN_FILTER	UsernamePasswordAuthenticationFilter	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

你可以将自己的过滤器添加到这个栈中，使用`custom-filter`来指定你的过滤器位置：

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

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

如果你希望你的过滤器插入栈中的另一个过滤器之前或之后，你也可以使用after或before属性值。"FIRST"和"LAST"可以用于position属性，表明你希望将过滤器添加到整个栈之前或者之后。

避免过滤器位置冲突

如果你正在插入一个自定义的过滤器，并占据了一个由命名空间创建的标准过滤器的位置，那么一定不要包含错误

> 的命名空间版本。如果你要替换过滤器功能，那么务必移除所有创建过滤器的元素。
>
> 有些`<http>`元素自己创建的过滤器是不能替换的，例如`<SecurityContextPersistenceFilter>`、`<ExceptionTranslationFil`一些其它过滤器默认被添加，但你可以禁用它们。`AnonymousAuthenticationFilter`是默认添加的，并且除非你指定禁用 session固化保护[13]，否则`SessionManagementFilter`也会被自动添加到过滤器链之中。

如果你正在替换一个要求认证入口（即未被认证的用户试图访问被保护的资源时触发认证处理的地点）的命名空间过滤器，那么你需要加上自定义的入口bean。

设置自定义的AuthenticationEntryPoint

如果你没有使用表单登录、OpenID或者基于namespace的基础的认证，那么你也许需要使用传统的bean语法定义一个认证过滤器和一个入口，并通过命名空间连接它们。匹配的`AuthenticationEntryPoint`可以在`<http>`元素中用`entry-point-ref`来设置。

CAS实例程序是一个非常好的在namespace中使用自定义bean的例子，它包含了这一语法。如果你不熟悉认证入口，可参见 技术概述[14] 一章。

6.4. 方法安全

从2.0往后的Spring Security版本开始已经从实质上改进了为你的服务层方法添加安全的支持。提供了对JSR-250注解安全的支持，就像框架原来的@Secured注解那样。从3.0起，你还可以使用基于表达式的注解。你可以把安全应用到一个单例bean，使用`intercept-methods`元素来装饰bean的声明，或者你可以使用AspectJ风格的切点横切整个服务层，从而保护多个beans。

[13] http://docs.spring.io/spring-security/site/docs/4.1.3.RELEASE/reference/htmlsingle/#ns-session-fixation

[14] http://docs.spring.io/spring-security/site/docs/4.1.3.RELEASE/reference/htmlsingle/#tech-intro-auth-entry-point

6.4.1. <global-method-security>元素

这个元素被用于在你的应用程序中启用基于注解的安全（通过在元素上设置适当的属性），也用于聚集安全切点声明，这些声明将用来横切你的应用上下文。你应该声明一个<global-method-security>元素。下面这个声明可以启用Spring Security的@Secured注解：

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

将注解添加到方法（类或接口的）能够限制对相应方法的访问。Spring Security原生注解支持为方法定义一个属性设置。它可以传递给AccessDecisionManager来做实际的判定：

```
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);
}
```

JSR-250注解也能够被支持：

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

这些基于标准的，允许简单的基于角色的约束将被应用，但它们肯定没有Spring Security原生注解好用。想要使用新的基于表达式的语法，你可以这样：

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

然后你可以在Java代码中改成这样：

```
public interface BankService {

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

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

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

如果你需要定义简单的规则，此规则不止能够检查用户授权清单的角色名，那么基于表达式的注解是一个不错的选择。

注意

被注解的方法只能在实例中被保护，此实例需要作为Spring beans被定义（应用上下文需要与声明method-security为启用的是同一个）。如果你想要保护不是由Spring创建的实例（例如使用new操作符），那么你就应该使用AspectJ。

你可以在同一个上下文中启用多个注解类型，但对于任意接口或类，只能有一种注解行为，否则就不是好的定义。如果在某个方法上找到两个注解，那么只有其中一个会被应用。

使用protect-pointcut添加安全切点

protect-pointcut的用法非常强大，它允许你只用一个简单的声明，就把安全应用到多个beans。例如

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

这将保护所有的方法，这些方法在应用上下文中的beans里面被声明，而这些类处于com.mycompany这个包中，并且类名以"Service"结尾。只有角色是"ROLE_USER"的用户能够调用这些方法。就像URL匹配一样，最特殊的匹配应该被定义在切点清单的最上边，作为第一个被匹配的表达式被使用。安全注解则会优先与切点被使用。

6.5. 默认的AccessDecisionManager

这章假设你已经了解了一些Spring Security的基础架构。如果你不了解，那就先跳过此节，往后阅读几章后再回过头来看看。本节只和需要做更多简单的基于角色安全的定制的用户有关。

当你使用了一个命名空间配置，那么一个默认的AccessDecisionManager实例会被自动注册，并被用来对方法的调用和web URL的访问做判定，判定基于你在intercept-url和protect-pointcut中所声明的访问属性（如果你使用了注解来保护你的方法，那么还有方法）。

默认的策略是使用一个AffirmativeBased AccessDecisionManager伴随一个RoleVoter以及一个AuthenticatedVoter。你可以在(授权[15]章节找到更多资料。

6.5.1. 定制AccessDecisionManager

如果你需要使用更复杂的访问控制策略，为方法和web安全设置另一种方案也是很简单的。

对于方法安全，你可以在应用上下文中，给global-method-security中的access-decision-manager-ref属性设置一个适当的 `AccessDecisionManager`bean的id来做到：

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

web安全的语法是一样的，但要放在http元素里面：

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

[15] http://docs.spring.io/spring-security/site/docs/4.1.3.RELEASE/reference/htmlsingle/#authz-arch)

6.6. 认证管理器与命名空间

在Spring Security中提供认证服务的主要接口是AuthenticationManager。它通常是一个ProviderManager类的实例，如果你之前玩过此框架那么你应该会对他很熟悉。如果不熟悉，那么你可以在 技术概述[16]一章中熟悉它。bean实例通过authentication-manager命名空间元素来创建。如果你通过命名空间使用HTTP或者方法安全，那么就不能使用定制的AuthenticationManager，不过这应该不是问题，你已经通过使用AuthenticationProvider进行了全局控制。

你也许想要注册附加的AuthenticationProvider beans到ProviderManager之中，你可以通过使用<authentication-provider>元素的ref属性来做到这一点，ref属性的值应该是provider bean的名字：

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

<bean id="casAuthenticationProvider"

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

另一个通用的需求是上下文中的另一个bean可能需要一个对于AuthenticationManager的引用。你可以很简单的为AuthenticationManager注册一个别名，并在应用上下文的别处使用它：

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

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

[16] http://docs.spring.io/spring-security/site/docs/4.1.3.RELEASE/reference/htmlsingle/#tech-intro-authentication

7

示例应用

项目中有一些可用的web应用示例。为了过度下载，只有“tutorial”和“contacts”示例被包含到了分发的zip文件中。另一些示例可以直接从源码中构建，你可以在简介的描述中获取它们。你可以很容易地构建项目，也可以在 http://spring.io/spring-security/ 中查阅更多资料。本章所有的路径都引用自相关的项目源目录中。

7.1. Tutorial 示例

tutorial是非常好的起步教材。它全部使用了简单的namespace配置。"distribution zip"文件中包含了已编译好的应用，可以方便的部署到你的web容器中 (`spring-security-samples-tutorial-3.1.x.war`)。基于表单的认证机制被用来结合常用的remember-me认证，从而在cookies中自动记住了登陆状态。

我们建议你像下面这样，用简短的XML开始tutorial示例。最重要的是，你可以很容易地添加这一XML文件（以及对应的`web.xml`记录）到你的原有应用中。我们建议你在实现了这一基本的集成之后，再尝试去添加method授权或domain object安全。

7.2. Contacts

这个Contacts示例是一个高级的例子，它展示了除基础应用完全之外的更强大的domain对象访问控制列表功能。

配置非常简单，只要简单的拷贝Spring Security分发包中的`spring-security-samples-contacts-3.1.x.war`文件到你的容器的webapps目录中即可。(版本号应当和你正在使用的release版本相匹配)

在启动你的容器之后，查看应用能否加载。访问http://localhost:8080/contacts（或者其他适当的URL）

下一步，点击“Debug”。你会被要求进行认证，页面会提示你输入用户名和密码。简单的进行认证，并查看结果页面，你可以看到类似下面这样的成功的消息：

```
Security Debug Information

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

Authentication object as a String:

org.springframework.security.authentication.UsernamePasswordAuthenticationToken@1f1278
Principal: org.springframework.security.core.userdetails.User@b07ed00:
 Username: rod; \
Password: [PROTECTED]; Enabled: true; AccountNonExpired: true;
credentialsNonExpired: true; AccountNonLocked: true; \
Granted Authorities: ROLE_SUPERVISOR, ROLE_USER; \
Password: [PROTECTED]; Authenticated: true; \
Details:
 org.springframework.security.web.authentication.WebAuthenticationDetails@0:
 \
RemoteIpAddress: 127.0.0.1; SessionId: 8fkp8t83ohar; \
Granted Authorities: ROLE_SUPERVISOR, ROLE_USER

Authentication object holds the following granted authorities:

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

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

如果你成功接收到上面这些消息，试着返回示例应用的主页面，然后点击“Manage”。你可以试试这个应用。注意，联系人只能对已登陆的用户进行展示，并且只有用户具有ROLE_SUPERVISOR角色时才会被允许删除他们的联系人。在后台，MethodSecurityInterceptor正在保护着这些business对象。

应用程序允许你针对不同的联系人修改访问控制清单。一定要运行一下，然后通过reviewing此应用的XML文件，来理解它们是怎样运作的。

7.3. LDAP示例

LDAP示例应用提供了基本的配置，并且同时使用了namespace配置以及等价的传统beans配置，两者都在应用程序上下文文件中。这意味应用中存在两个权限提供者的配置。

7.4. OpenID示例

OpenID示例展示了如何使用命名空间进行OpenID的配置，以及如何为谷歌、雅虎和MyOpenID(你可以试着加入一些其他你想要的)身份提供者设置属性来变换配置。该示例使用了基于JQuery的openid-selector项目，来提供一个对用户友好的登陆页面，从而允许用户更容易地选择一个provider，而不是输入全部OpenID标识。

此应用区别于传统的认证场景，它允许任何用户访问站点（提供给它们的OpenID认证是成功的）。你第一次登陆时，你将得到一个"Welcome [your name]"的消息。如果你注销并重新登陆（用同样的OpenID身份），提示会变为“Welcome Back”。这由一个定制的`UserDetailsService`实现，它给任何用户都分配了一个标准的角色，并且在一个map中保存身份信息。这个类也重视这样一个事实，不同的属性也许会由不同的provider返回，并根据不同的用户构建不一样的称呼。

7.5. CAS示例

CAS示例需要你同时运行CAS服务器与CAS客户端。它没有被distribution包含，因此你需要像介绍里面说的那样 检出源码[1]。你会在`sample/cas`目录找到相关文件。`Readme.txt`文件中解释了如何直接从源码树中同时执行服务器与客户端，并启用SSL支持。

7.6. JAAS示例

JAAS示例非常简单，它展示了如何在Spring Security中使用JAAS的LoginModule。如果用户名与密码相匹配，被提供的LoginModule会成功认证，否则会抛出一个`LoginException`。 这个示例使用的`AuthorityGranter`总会授权`ROLE_USER`。示例程序还展示了如何通过设置`jaas-api-provision`为"true"来像JAAS Subject 一样运行，由`LoginModule`进行返回。

[1] http://docs.spring.io/spring-security/site/docs/4.1.3.RELEASE/reference/htmlsingle/#get-source

7.7. 预认证示例

这个例子展示了如何从预认证框架中绑定beans，从而从Java EE容器中使用登陆登陆信息。用户名与角色由容器设置。

代码在`samples/preauth`中。

8

Spring Security社区

8.1. 缺陷跟踪

Spring Security使用JIRA来管理bug报告与enhancements申请。如果你找到一个bug，请用JIRA记录一份报告。不要在支持论坛上记录它，用email发送清单给项目开发者。这样可以点对点处理，同时可以让我们更方便的管理bugs。

如果可以，请在你的问题报告中提交一个JUnit测试来展示任何错误的行为。或者，更进一步，你可以提供一个补丁来纠正此问题。类似的，但如果你包含了适当的单元测试，我们也非常欢迎你在缺陷跟踪里提出enhancements，尽管我们只会接受enhancements请求。为了确保项目测试被充分覆盖，这是非常必要的。

你可以通过 https://github.com/spring-projects/spring-security/issues 访问缺陷跟踪者。

8.2. 加入我们

我们在Spring Security项目中欢迎你的参与。有很多贡献的方式，如访问论坛并回复一些他人的问题，写出一些新的代码，改进旧的代码，协助文档，开发示例与教程，或者简单地发表建议。

8.3. 更多信息

Spring Security非常欢迎问题与评论。你可以在Spring的StackOverflow web的站点 http://spring.io/questions，和其他使用Spring Security的用户讨论此框架。

部分 II. 架构与实现

一旦你熟悉了基于namespace配置的应用的设置与运行，你可能会希望做更多的开发工作，并理解framework的namespace背后的机理。就像大多数软件一样，Spring Security具有某些核心接口、类型与抽象的概念，它们在框架中非常通用。在这一部分，我们会向你展示它们如何再一起进行协作，从而在Spring Security中支持认证与访问控制。

目录

9

技术概况

9.1. 运行时环境

Spring Security 3.0需要JRE 5.0或更高版本。Spring Security的目标是使用自包含的方式运作，不向JRE中加入任何特殊的配置文件。尤其是不需要去配置特殊的JAAS策略文件，或把Spring Security放到通用的本地classpath中。

类似的，如果你在使用EJB容器或Servlet容器，也不需要到处放置特殊的配置文件，也不需要在服务器的classloader中包含Spring Security。所有需要的文件将只在你的应用中包含。

这样的设计可以根据需要的配置时间灵活地调整你的配置，你可以简单地从一个系统中拷贝目标artifact（可以是一个JAR、WAR或EAR）到其他系统，artifact依旧能够立即运作。

9.2. Core Components

在Spring Security 3.0中，`spring-security-core` jar的内容被单独剥离。它不再包含任何与web-application安全、LDAP或命名空间配置相关的代码。我们会向你展示一些core模块中的Java类型。它们代表了框架的构建单元，因此如果你需要在namespace配置上更进一步，或者你不想要与它们直接交互，那么理解此模块就显得非常重要。

9.2.1. SecurityContextHolder, SecurityContext 以及 Authentication 对象

最基本的对象是`SecurityContextHolder`。这时我们存储当前应用security上下文细节的地方，它包含了当前使用应用的主要细节。`SecurityContextHolder`默认

使用一个ThreadLocal来包装这些细节，它说明security上下文在总是对同一个执行线程中的方法是可用的，而不管security上下文是否明作为一个参数传递给这些方法。如果你在当前请求被处理之后清除了这个线程，那么使用ThreadLocal是非常安全的。当然，你不需要担心这些，因为Spring Security已经自动帮你处理了这些事。

一些应用并不完全使用ThreadLocal，因为它们通过特别的方式来使用线程。

例如，一个Swing客户端可能需要JVM中所有的线程都共用同一个security上下文。SecurityContextHolder可以配置一种启动策略来指定你所需要的上下文存储方式。你可以使用 SecurityContextHolder.MODE_GLOBAL模式来启用单独的应用上下文。

另一些应用程序也许需要由安全的线程衍生出的多个线程，并采用同样的安全标识。这可以使用SecurityContextHolder.MODE_INHERITABLETHREADLOCAL模式。

你可以用两种方式改变默认的SecurityContextHolder.MODE_THREADLOCAL到此模式。第一种办法是设置系统参数，第二种办法是调用SecurityContextHolder的静态方法。

大多数应用不需要改变默认模式，但如果你需要这么做，那就看看SecurityContextHolder的JavaDocs吧！

获取关于当前用户的信息

在SecurityContextHolder内部，我们包装了与应用交互的主要的一般的细节。Spring Security使用一个Authentication对象来代表这些信息。你通常不需要自己创建一个Authentication对象，但它对于查询用户查询Authentication对象非常有用。你可以在你的应用的任何地方使用下面这些代码，以获取当前认证用户的姓名：

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

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

调用getContext()方法返回的是一个实现了SecurityContext接口的实例。这个对象被本地线程所存储。我们可以在下文看到，大多数Spring Security认证机制都以返回一个UserDetails实例为主。

9.2.2. UserDetailsService

注意上面的代码段中的另一个元素，你可以从Authentication中获取一个principal。大多数时候它可以转化为一个UserDetails对象。UserDetails是Spring Security的一个核心接口，它代表了一个principal。它是一种可扩展的并且特定于应用程序的方式。

想象一下UserDetails适配器，在你自己的用户数据库与Spring Security内部所需要的SecurityContextHolder之间。成为一个你自己的用户数据库的代表，相当于你能够将UserDetails转化为你的应用程序提供的源对象，因此你可以调用特定于业务的方法（如getEmail()、getEmployeeNumber()等等）。

现在你也许会担心，我什么时候需要提供一个UserDetails对象？我如何提供？我以为你说这东西被声明了，并且我不需要写任何Java代码，什么情况？这有个简单的答案：它是一个叫做 UserDetailsService的接口。这个接口唯一一个方法允许接受一个基于字符串的username参数，并返回一个UserDetails：

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

这是Spring Security中最通用的读取用户信息的办法。你会发现，无论何时，当用户信息被需要时，它都会被使用。它贯穿着整个框架。

在认证成功时，UserDetails被用来构建Authentication对象，它被封装在SecurityContextHolder对象中（参见后文）。好消息是我们提供了一些UserDetailsService实现，包括一个使用内存map的(InMemoryDaoImpl)以及一个使用JDBC的（JdbcDaoImpl）。大多数用户倾向于自己写一个，尽管如此，他们的实现经常只是很简单的构建在DAO之上，并且代表了他们的雇员、消费者或其他应用程序的用户。记住这个优点，无论你的UserDetailsService返回了什么，它都可以像上面的代码段中通过SecurityContextHolder来获取。

经常有一些关于UserDetailsService的混淆。它是一个纯粹的保存用户数据的DAO，并且除了为框架中的其它组件提

> 供数据以外不执行其它任何功能。尤其是它不用于认证用户，用户认证由AuthenticationManager进行操作。很多情况下，如果你需要定制认证过程，那么有很多场景下可以直接实现AuthenticationProvider。

9.2.3. GrantedAuthority

除了principal之外，另一个重要的方法是Authentication的getAuthorities()。这个方法提供了一个GrantedAuthority对象数组。每个GrantedAuthority都代表一个给予principal的权限。权限通常是"roles"，例如ROLE_ADMINISTRATOR或者ROLE_HR_SUPERVISOR。这些角色可以为web认证、方法认证和领域对象认证所配置。Spring Security的其它部分能够解析这些权限，并且期望它们已经被声明。GrantedAuthority对象通常从UserDetailsService中读取。

通常，GrantedAuthority对象用于应用程序级别的权限认证。它们不用于指定给定的领域对象。因此，你不会希望用GrantedAuthority来代表一个54号雇员对象的权限，因为如果有上千个类似的权限，那么你很快就可以让你的内存爆满（或者，至少，它会造成应用程序认证用户的时间变长）。当然，Spring Security被明确地设计于处理这个需求，但你应该用项目的领域对象安全功能来作为替代实现。

9.2.4. 概要

简要回顾一下，我们看到了Spring Security主要的构建块：

- **SecurityContextHolder**， 提供对SecurityContext的访问。
- **SecurityContext**，保存Authentication，可能还包括特定于请求的安全信息。
- **Authentication**，代表一个principal，它是Spring Security独有的规范。
- **GrantedAuthority**，指代对一个principal的应用级的权限授权。
- **UserDetails**，在你的应用中提供构建一个Authentication对象的必要的信息，信息源可以是DAO，也可以是其它安全数据源。
- **UserDetailsService**，给它传递一个基于字符串的用户名（或者凭证ID、或者其它）时，它会返回一个UserDetails。

现在，你已经对这些常用的组件有了一些了解，让我们更进一步地看看认证过程吧。

9.3. Authentication

Spring Security可以参与到很多不同的认证环境中。虽然我们建议人们使用Spring Security进行认证，同时不要与已存在的认证管理容器整合，但Spring Security并非不支持与你自己专有的认证系统进行整合。

9.3.1. Spring Security中的authentication是什么

让我们考虑一下每个人都熟悉的标准的认证场景。

1. 用户被提示要使用用户名与密码进行登陆
2. 系统对用户名与密码都认证成功
3. 用户的上下文信息被获取（还有他们的角色清单这些信息）
4. 为用户建立一个安全的上下文
5. 用户处理过程，可能会由访问控制机制执行一些潜在的保护操作，而访问控制机制为针对当前安全上下文信息的操作检查所需的权限。

前三点组成了认证过程，我们看看Spring Security是怎样做的：

1. 获取用户名和密码，并绑定到一个`UsernamePasswordAuthenticationToken`实例（一个`Authentication`接口的实例）。
2. token被传递到一个`AuthenticationManager`的实例进行校验。
3. 如果认证成功，那么`AuthenticationManager`返回一个填充后的`Authentication`实例。
4. 安全上下文通过调用`SecurityContextHolder.getContext().setAuthentication(…)`被建立，然后将前面返回的`authentication`对象传递进去。

上面已经说明了用户如何被认证，下面我们来看看代码：

```
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");
}
}
```

我们写了一个小程序来让用户输入用户名和密码，并执行上面的过程。我们实现了AuthenticationManager接口，并对用户名等于密码的用户进行认证。然后为每个用户简单的分配了一个角色。上面的程序的输出可能会像这样：

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

注意你通常不需要写这样的代码。这个过程通常会在Spring Security的内部处理，就像一个web认证过滤器那样。上面那些代码对Spring Security中如何实现认证这个问题进行了一个简单的回答。当SecurityContextHolder包含一个全填充的Authentication对象时用户被认证。

9.3.2. 直接设置SecurityContextHolder中的应用上下文

事实上，Spring Security不关心你如何把Authentication对象put到SecurityContextHolder之中。唯一需要的是在SecurityContextHolder中包含一个Authentication对象，此对象要能够在AbstractSecurityInterceptor（晚点我们会介绍）对用户操作进行授权之前表示一个principal。

你可以（很多用户也这样做）自己写一个过滤器或者MVC控制器，从而提供一个与不是基于Spring Security的权限认证系统的互用性。例如，你可能正在使用容器管理认证，它能够从ThreadLocal或者JNDI位置获得当前用户。或者你可能工作在一个公司遗留的专有认证系统之上，它有一个共同的“标准”并且不再你的控制范围。这种情况下也很容易让Spring Security工作，同时仍然提供授权能力。你需要做的就是写一个过滤器（或者类似的东东），让它从某个位置读取第三方用户信息，构建一个Spring Security专有的Authentication对象，然后把它put

到SecurityContextHolder之中。这种情况下，你还需要考虑一些事，这些事情通常由内嵌的认证架构自动处理。例如，你可能需要在请求中抢先创建一个HTTP session存储到上下文，这必须发生在你为客户端脚注写入响应信息之前：因为一旦response被提交那么就不能再创建session。

如果你希望了解AuthenticationManager如何在实际工作中被实现，我们将会在核心服务[1]一章告诉你。

9.4. Web应用程序中的认证

现在让我们看看在web应用程序中使用Spring Security的情形（web.xml安全未启用）。这种情况下，用户如何认证？安全上下文如何建立？

考虑一个传统的web应用程序认证过程：

1. 你访问到主页，然后点击了一个链接。
2. 一个请求到达服务器，然后服务器判断你是否在请求一个被保护的资源。
3. 你目前没有认证，服务器发回一个响应要求你必须先认证。响应可能是一个HTTP响应吗，或者是一个跳转到指定web页面的重定向。
4. 根据认证机制，你的浏览器会将你重定向到指定页面，然后你可以填写表单；或者浏览器以某种方式检索你的身份（通过一个基本的认证复选框，一个cookie，或者一个X.509证书之类）。
5. 浏览器会发回一个响应给服务器。这可能是一个HTTP POST，它包含了你填写的表单内容；或者一个HTTP头，它包含了你的认证详情。
6. 接下来，服务器会判定提交的凭证是否有效。如果它们是有效的，则进行下一步，否则你的浏览器通常会让你重新提交一次凭证（所以你又回到了第二步）。
7. 你一开始发送的访问被保护的资源的那个请求会被重试。服务器会继续判定你是否拥有充足的权限。如果你有，那么请求成功。否则，你会收到一个HTTP的403错误码，它意味着禁止访问。 Spring Security拥有截然不同的类分别负责大部分的上述步骤。最主要的参与者(它们被按顺序使用)是ExceptionTranslationFilter，一个AuthenticationEntryPoint和一个"authentication mechanism"，它们负责调用AuthenticationManager，就像我们上一节看到的那样。

[1] http://docs.spring.io/spring-security/site/docs/4.2.0.RELEASE/reference/htmlsingle/#core-services-authentication-manager

9.4.1. ExceptionTranslationFilter

ExceptionTranslationFilter是一个Spring Security过滤器，它负责检查任何Spring Security中被抛出的异常。这个异常通常由一个AbstractSecurityInterceptor抛出，它是主要的授权服务提供者。我们会在下一节介绍AbstractSecurityInterceptor，但现在我们需要知道它产生Java异常，并且不需要知道关于HTTP，也不需要知道怎样去处理principal的认证。作为替代，ExceptionTranslationFilter提供处理异常的服务，它为每个返回的403错误码指定职责（如果principal已经被认证，因此缺乏充足的权限，就像上面的第七步那样），或者启动一个AuthenticationEntryPoint（如果principal没有认证，因此我们要跳到上面的第三步）。

9.4.2. AuthenticationEntryPoint

AuthenticationEntryPoint主要负责上面的列表中的第三步。你可以想象，每个web应用程序都具有一个默认的认证策略（好吧，这可以在Spring Security里面配置为一切别的东西，但现在让我们保持简单）。每个主要的认证系统都具有自己的AuthenticationEntryPoint实现，它典型地处理一个像步骤3一样的动作。

9.4.3. 认证机制

一旦你的浏览器提交了你的认证凭证（比如一个HTTP表单或者HTTP头），服务器需要做一些事情，比如去搜集认证的详细信息。现在，我们进入了上面的清单的第六步。在Spring Security中，我们为搜集权限详情的方法指定了一个名称，认证详情来自一个用户代理（通常是一个web浏览器），参考自《认证机制》。例子是基于表单的登陆和基础认证。一旦认证详情从用户代理上被搜集，Authentication请求对象会被构建，并提供给AuthenticationManager。

在认证机制收到全填充的Authentication对象之后，就认为请求是有效的，然后将Authentication put到SecurityContextHolder之中，最后引起一个对源请求的重定向（上面的第七步）。如果认证失败，AuthenticationManager会拒绝请求，认证机制会要求用户代理进行重试（上面的第二步）。

9.4.4. 在请求间保存SecurityContext

根据应用类型的不同，可能会需要一个策略来将应用上下文存储到用户操作中。在传统的web应用程序里，用户一旦登陆就通过session Id获得了身份凭证。服务器在保存的session中缓存principal的信息。在Spring Security中，在请求间保存SecurityContext的责任落到了SecurityContextPersistenceFilter头上，它

默认将上下文作为HttpSession属性保存到HTTP请求之间。它在每一次请求时向SecurityContextHolder复原上下文，并且，最关键的一点，在请求完成时清除SecurityContextHolder。你不应该为了安全目标直接使用HttpSession。没有理由这样做，因为你总是可以使用SecurityContextHolder来作为替代。

很多其他类型的应用程序（例如无状态的RESTful web服务）不使用HTTP sessions，并且每个请求都会重新认证。然而，在过滤器链中包含SecurityContextPersistenceFilter依然非常重要，它确保SecurityContextHolder在每次请求之后都被清除。

在一个应用程序中，并发请求由一个单例的session接收，同样的SecurityContext实例会被不同的线程共享。尽管能够使用ThreadLocal，但在HttpSession中为每个线程恢复的都是同一个实例。言外之意就是，如果你希望临时改变上下文，只能在线程运行中操作。如果你只是使用SecurityContextHolder.getContext()，并且在被返回的上下文对象中调用setAuthentication(anAuthentication)，那么Authentication对象会改变到所有并发线程中，因为它们都共用同一个SecurityContext实例。你可以定制SecurityContextPersistenceFilter的行为，为每一个请求都创建一个新的SecurityContext，从而阻止从一个线程中操作SecurityContext会影响其他线程的行为。或者你可以在你临时需要的地方创建一个新的实例。SecurityContextHolder.createEmptyContext()总会返回一个新的上下文实例。

9.5. Spring Security中的访问控制（授权）

Spring Security中负责做访问控制判定的主要的接口是AccessDecisionManager。它具有一个判定方法，该判定方法接收一个Authentication的对象，该对象代表了一个发起访问请求的principal。还有一个"安全的对象"（参见下面），以及一个用于对象的安全元属性的列表（例如给访问进行授权所需要的角色列表）。

9.5.1. 安全与AOP通知（advice）

如果你熟悉AOP，那么你一定知道几种可用的通知类型：before，after，throws和around。around通知非常有用，因为advisor可以选择是否处理一个方法调

用，是否改变response，还有是否要抛出一个异常。Spring Security为诸如web请求这类的方法调用提供一个around通知。我们使用标准的Spring AOP支持来实现方法调用的around通知，我们使用标准的过滤器来实现web请求的around通知。

如果你不熟悉AOP，关键点在于理解Spring Security能够帮助你保护例如web请求这类的方法调用。很多人都对在服务层保护方法调用很感兴趣。因为在当代Java EE应用中，服务层具有最多的业务逻辑。如果你只需要在服务层保护方法调用，Spring的标准AOP就足够了。如果你需要直接保护领域对象，那么你需要使用像是AspectJ这样的工具。

你可以选择使用AspectJ或是Spring AOP来执行方法授权，或者你可以选择使用过滤器来操作web请求授权。你还可以任意组合它们。主流的使用模式是操作一些web请求授权，在服务层再结合一些Spring AOP方法调用授权。

9.5.2. 安全对象和AbstractSecurityInterceptor

什么是安全的对象呢？Spring Security把任何具备了安全性（例如授权判定)的对象叫做安全对象。最通用的例子是方法调用和web请求。

每个被支持的安全的对象类型拥有自己的拦截器类，它们都是`AbstractSecurityInterceptor`的子类。通过适时的调用`AbstractSecurityInterceptor`非常重要，如果principal已经被认证，那么`SecurityContextHolder`会包含一个有效的`Authentication`。

`AbstractSecurityInterceptor`提供一个一致的工作流来操作安全对象请求，通常是：

1. 为上送的请求查找适当的配置属性
2. 提交安全对象、当前的`Authentication`和配置属性到`AccessDecisionManager`，从而进行授权判定
3. 可以改变`Authentication`的调用方式
4. 允许调用安全对象进行处理（假设访问被授权）
5. 一旦调用被返回，如果配置了`AfterInvocationManager`，那么调用它。如果调用过程抛出了一个异常，那么`AfterInvocationManager`不会被调用。 6.

什么是配置属性

配置属性可以接收一个字符串，该字符串对通过AbstractSecurityInterceptor使用的类具有一些特殊的意思。它们由框架里的接口ConfigAttribute表示。它们可以是简单的角色名，或者具有更复杂的意义，并根据AccessDecisionManager实现的不同而不同。AbstractSecurityInterceptor被和一个SecurityMetadataSource配置到一起，后者用来为安全对象查找属性。通常这些配置会对用户隐藏。配置属性可以注解到被保护的方法上，或者作为访问属性应用在被保护的URLs上。例如，当我们在命名空间描述中看到一些类似于<intercept-url pattern='/secure/**' access='ROLE_A,ROLE_B'/>的内容，这表示要将 ROLE_A 和 ROLE_B 这两个配置属性应用到给定的pattern上。事实上，通过默认的AccessDecisionManager配置，任何具有一个GrantedAuthority并且匹配上述二者其中之一的人都能够被允许访问。严格来讲，它们只是属性，解析依赖于AccessDecisionManager的实现。标记前缀“ROLE_”用于表明这一属性是角色，并且应该被Spring Security的RoleVoter消费。当基于voter的AccessDecisionManager被使用时，这是唯一相关联的。我们可以在授权章节看到AccessDecisionManager如何被实现。

RunAsManager

假如AccessDecisionManager决定允许请求，AbstractSecurityInterceptor通常会继续请求。话虽如此，依然有极少的一些用户想要用一个不同的Authentication来替换SecurityContext中的Authentication，前者通过AccessDecisionManager调用RunAsManager来操作。这可能在某些适合的情况下很有用，例如当一个服务层方法需要调用一个远程系统并且提供一个不一样的凭证时。因为Spring Security会自动的从一个服务器向另一个服务器传递安全凭证（假设你正在使用一个经过适当配置的RMI或者HttpInvoker远程协议客户端），这可能非常有用！

AfterInvocationManager

随着安全对象调用过程的返回 - 它们可能意味着一个方法调用完成或者一个过滤器链处理完成 - AbstractSecurityInterceptor会得到一个最后的机会去操作调用过程。在这一阶段，AbstractSecurityInterceptor可能会对改变返回的对象有所兴趣。我们可能希望能够这样做，因为授权判定无法被以唯一的方式进行安全对象的调用。还有更高的扩展性，如果需要的话，AbstractSecurityInterceptor会将控制传递到一

个AfterInvocationManager，从而改变该对象。这个类甚至可以完全地替换该对象，或者抛出一个异常，或者不改变它，你可以做你任何想做的事。事后调用检查只会在调用过程成功事被执行。如果抛出了异常，那么附加的检查会被跳过。

AbstractSecurityInterceptor和它的相关对象在《图9.1 安全拦截器与安全对象模块》可以查看详情。

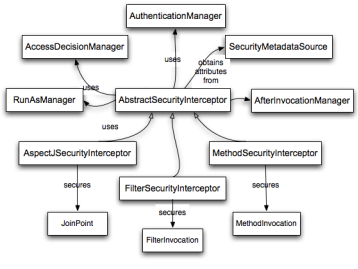

图 **9.1.** 安全拦截器和安全对象模块

集成安全对象模块

只有在开发者考虑整个拦截与授权请求的新的方式时，才需要去直接使用安全对象。例如，可能要构建一个新的安全对象去保护对一个消息系统的调用。需要安全性的任何事物都要提供一种拦截调用的方式（例如AOP around通知语义），它能够用于一个安全对象。话虽如此，大部分Spring应用只需要简单的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）来透明地完成。

部分 III. Testing

This section describes the testing support provided by Spring Security.

To use the Spring Security test support, you must include `spring-security-test-4.1.3.RELEASE.jar` as a dependency of your project.

部分 IV. Web应用安全

大部分Spring Security的用户会将此框架用于创建HTTP用户和Servlet API的应用程序。在这一部分，我们会带你了解Spring Security如何为应用程序的web层提供认证和访问控制功能。我们会在后文看到命名空间的表观，也会看到类和接口是如何被聚集起来提供web层安全的。在某些情况下，使用传统的bean配置来提供超越配置的全量控制非常有必要，因此我们也会看到如何不通过命名空间，直接配置这些类。

目录

10

安全过滤器链

Spring Security的web架构完全基于标准的servlet过滤器。它并没有使用servlets或者任何其它基于servlet的内部框架（例如Spring MVC），因此它并没有与任何特殊的web技术强关联。它处理`HttpServletRequest`s和`HttpServletResponse`，同时并不关心请求是来自一个浏览器、一个web服务客户端、一个HttpInvoker还是一个AJAX应用程序。

Spring Security在内部维护一个过滤器链，每个过滤器都具有特殊的职责，过滤器可以根据服务的需要与否在配置中被添加或删除。过滤器的顺序非常重要，就像它们之间的相关性那样。如果你已经使用了命名空间配置，那么过滤器会自动为你配置，你不需要明确定义任何Spring beans，除非有时候你想要完全控制安全过滤器链，或者因为你正在使用的功能不被命名空间支持，或者你正在使用你自定义的类版本。

10.1. DelegatingFilterProxy

在使用servlet过滤器时，你首先得在web.xml中进行声明，否则他们将被servlet容器给忽略。在Spring Security中，过滤器类同时也是Spring beans，它们都在应用上下文中被定义，因此可以使用Spring的先进而丰富的依赖注入工具以及声明周期接口。Spring的`DelegatingFilterProxy`提供web.xml与应用上下文之间的连接。

在你使用`DelegatingFilterProxy`时，你会在web.xml中看到：

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

```
</filter>

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

注意，过滤器就是一个DelegatingFilterProxy，而不是实现了过滤器逻辑的类。DelegatingFilterProxy所做的事情是将过滤器的方法委托给一个从Spring应用上下文获取的bean。这可以使bean能够从Spring web应用上下文声明周期的灵活的支持和配置中获得好处。bean必须实现javax.servlet.Filter接口，并且必须与filter-name中的名字相同。阅读DelegatingFilterProxy的Javadoc查看更多信息。

10.2. FilterChainProxy

Spring Security的web架构只能通过委托给一个FilterChainProxy的实例来使用。安全过滤器不应该由它们自己使用。理论上你可以在你的应用上下文中声明每个你需要的Spring Security过滤器bean，并为每个过滤器添加一个对应的DelegatingFilterProxy到web.xml中，同时确保它们具有正确的顺序，但这非常不方便，并且如果你还有大量的过滤器，这会使你的web.xml文件很快变得非常杂乱。FilterChainProxy让我们简单的在web.xml中添加一个入口，并且通过DelegatingFilterProxy来处理所有事情，就像上面的例子展示的那样，不一样的是你得把filter-name设置为"filterChainProxy"这个bean名。过滤器链被在应用上下文中以同样的bean名声明：

```
<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被用来很方便地设置你的应用程序中所需要的安全过滤器链(s)。[6]。它映射一个特别的URL节点到一个过滤器列表上，过滤器列表根据filters元素中指定的bean名来构建，并将它们绑定到一个SecurityFilterChain的bean中。pattern属性可以使用Ant风格的路径，最特殊的URIs应该被放到最上方[7]。在运行时，FilterChainProxy会定位第一个URI节点，匹配当前的web请求，然后过滤器属性指定的过滤器beans列表会被应用到请求中。过滤器会按照它们被定义的顺序被调用，因此在特定的URL经过过滤器链之后，你已经完成了控制。

你也许会注意到我们在过滤器链中声明了两个SecurityContextPersistenceFilter s（ASC是allowSessionCreation的简写，它是一个SecurityContextPersistenceFilter的参数）。web服务在功能请求中不会出现一个jsessionid，为这个用户代理创建HttpSession`s是非常浪费的。如果你有一个需要最大限度进行扩展的大容量的应用程序，我们建议你像上面那样。对于小一些的应用程序，使用一个单例`SecurityContextPersistenceFilter（它的默认的allowSessionCreation为true）则足够了。

注意，FilterChainProxy并不在它被配置的过滤器中调用标准的过滤器生命周期方法。我们建议你使用Spring应用上下文生命周期接口作为替代，就像你使用别的bean一样。

当我们知道了如何使用namespace配置来设置web安全时，我们使用了一个名字叫做"springSecurityFilterChain"的DelegatingFilterProxy。你可以看到这是被namespace创建的一个FilterChainProxy的名字。

10.2.1. 绕开过滤器链

你可以将属性filters设置为"none"，从而不用提供一个过滤器bean列表。这将会为相应的请求pattern省略整个过滤器链。注意，任何匹配这个路径的请求都不会应用认证或授权服务，它们将能够自由的被访问。如果你想要在请求过程中使用SpringSecurity上下文，那么你必须给它传递一个过滤器链。否则SecurityContextHolder不会被集成进来，同时上下文会是null。

10.3. 过滤器顺序

过滤器链中的过滤器顺序是非常重要滴。与你正在使用的过滤器无关，顺序应该像下面这样：

- ChannelProcessingFilter，因为它可能需要重定向到一个不同的协议
- SecurityContextPersistenceFilter，因此SecurityContext可以从web请求一开始就被设置到SecurityContextHolder中，并且在请求结束时任何对于SecurityContext的改变都可以复制到一个HttpSession之中（为下一次的web请求做准备）。
- ConcurrentSessionFilter，因为它使用了SecurityContextHolder的功能，并且需要更新SessionRegistry，从而响应不间断的来自principal的请求。
- 认证处理机制 - UsernamePasswordAuthenticationFilter，CasAuthenticationFilter，BasicAuthenticati等 - 因此SecurityContextHolder可以被修改，以包含一个有效的Authentication请求token。
- SecurityContextHolderAwareRequestFilter，如果你正在使用它来安装Spring Security，要记得将HttpServletRequestWrapper放入你的servlet容器。
- JaasApiIntegrationFilter，如果JaasAuthenticationToken存在于SecurityContextHolder之中，此过滤器会在JaasAuthenticationToken中处理FilterChain，并将其当作Subject。
- RememberMeAuthenticationFilter，因此如果没有简单的认证处理机制更新SecurityContextHolder，并且请求中包含了一个启用remember-me服务的cookie，那么一个适当的被记住的Authentication对象会在这里被put到应用上下文中。
- AnonymousAuthenticationFilter，因此如果没有认证处理机制去更新SecurityContextHolder，那么一个匿名的Authentication会在这里被put到应用上下文中
- ExceptionTranslationFilter，它捕获任何Spring Security异常，因此可以返回一个HTTP错误响应，或者启动一个合适的AuthenticationEntryPoint
- FilterSecurityInterceptor，保护web URIs，并在访问被拒绝时抛出异常。

10.4. 请求匹配，以及HttpFirewall

Spring Security有一些区域，你定义了这些区域的patterns，这些区域需要对进入的请求进行校验，从而为了判定请求应该如何被操作。当FilterChainProxy开始判断请求应该传递给哪一个过滤器链，或者FilterSecurityInterceptor开始判断要用哪一个安全约束来处理请求时，这一过程会被重现。这一机制是什么？URL值如何与你定义的patterns进行比较？理解这两个问题非常重要！

Servlet规范定义了一些关于HttpServletRequest的属性，它们可以通过getter方法获取，并且我们可能想要比较它们。这些属性包括contextPath，servletPath，pathInfo还有queryString。Spring Security只对应用中的安全路径感兴趣，因此contextPath会被忽略。不幸的是，servlet规范没有明确定义在一个特殊的请求URI中，servletPath和pathInfo的值会包含什么。例如，每个URL的路径段都可能包含参数，就像RFC 2396 [8]中定义的那样。规范也没有清楚地规定这些东东是否应该包含在servletPath和pathInfo的值中，从而导致在不同的servlet容器中这些行为是不一样的。这会存在一个危险：当一个应用程序被部署到一个没有将这些值从路径参数剥离的容器中时，攻击者可以将它们添加到请求的URL中，从而造成出乎意料的模式匹配成功或失败。[9]。其它的进入的URL变体也是存在可能的。例如，可能包含path-traversal序列（就像 /../)或者多个正斜杠（//）造成模式匹配失败。一些容器会使这些意外的格式能够正常映射到servlet中，但一些其它的容器则不会。为了针对这些问题进行保护，FilterChainProxy使用一个HttpFirewall策略来检查和包装一个请求。非正常的请求默认会被自动拒绝，地址参数和重复的斜杠会被移除，以匹配目标。[10]。因此，用FilterChainProxy来管理安全过滤器链是非常必要的。注意，servletPath和pathInfo值会被容器解码，因此你的应用程序不应该使用任何包含semi-colons的有效路径，它们会被移除以匹配目标。

就像上面提到的那样，默认的策略是使用Ant风格的路径进行匹配，这也许是大部分用户最好的选择。此策略由AntPathRequestMatcher类来实现，它使用了Spring的AntPathMatcher来针对连在一起的servletPath和pathInfo，执行一个大小写敏感的pattern校验，并忽略queryString。

有时候因为某些原因，你可能会需要更强大的匹配策略，那么你可以使用正则表达式。该策略由RegexRequestMatcher实现。参见此类的Javadoc查看更多信息。

在实践中，我们建议你在你的服务层使用方法安全，从而控制对你的应用程序的访问，而且不要完全依赖于定义在web应用等级的安全容器。URLs会改变，对于所有的可能的URLs，应用程序如何支持？请求如何处理？想要完全考虑这些是非常困难的。你应该尝试，并且限制你自己去使用一些简单的ant路径，虽然它们很容易被理解。坚持使用“默认拒绝”的办法，将一个全捕获的通配符定义在最后并拒绝用户访问。

将Security定义在服务层会使它更为健壮并难以绕开，所以你应该坚持利用Spring Security的方法安全选项。

10.5. 使用其它的基于过滤器的框架

如果你正在使用一些其它的同样是基于过滤器的框架，那么你需要确保Spring Security的过滤器放在第一位。它确保在使用其它过滤器时SecurityContextHolder能够被及时集成。例如使用SiteMesh来装饰你的web页面，或者使用像Wicket的web框架，它通过一个过滤器来对请求进行操作。

10.6. 高级的命名空间配置

就像我们之前在命名空间一节看到的那样，可以使用多个http元素来为不同的URL patterns定义不同的安全配置。每个http元素都通过内部的FilterChainProxy创建一个过滤器链，然后将URL pattern映射到其上。这些元素会按照它们被声明的顺序被添加，因此最特殊的patterns必须被声明在最上方。还有一些其它的例子就像上面那样的情形，比如应用程序在支持无状态的RESTful API的同时，也支持用户通过通常的表单进行登录的web应用程序。

```
<!-- Stateless RESTful service using Basic authentication -->
<http pattern="/restful/**" create-session="stateless">

<http-basic />
</http>

<!-- Empty filter chain for the login page -->
<http pattern="/login.htm*" security="none"/>

<!-- Additional filter chain for normal users, matching all other
 requests -->
<http>

<form-login login-page='/login.htm' default-target-url="/home.htm"/>
<logout />
</http>
```

- [6] 注意，你可能需要包含security命名空间到你的应用程序上下文XML文件中，从而使用这一语法。以前使用filter-chain-map的语法仍然被支持，但不建议在注入构造参数时使用。
- [7] 作为路径pattern的替代，request-matcher-ref属性可以指定一个RequestMatcher实例来实现更强大的匹配。

- [8] 你可能见过它，在浏览器不支持cookies和jsessionid参数时，它被在一个`semi-colon`后面附加在URL上。然而RFC允许这些参数存在于URL的任何一个路径段中。
- [9] 一旦请求离开`FilterChainProxy`，那么源值就会被返回，因此在应用中仍然可用。
- [10] 所以，例如，一个源请求路径 /secure;hack=1/somefile.html;hack=2 会被作为 /secure/somefile.html 返回。

11

核心安全过滤器

有一些键值过滤器，它们总会被使用了Spring　Security的web应用程序所使用，所以我们得先看看它们和它们支持的类与接口。我们不会说明每一个功能，所以如果你想要得到完整的信息，那么就去看看它们的Javadoc。

11.1. FilterSecurityInterceptor

我们已经在介绍access-control时简短的介绍了一下FilterSecurityInterceptor，并且我们已经在使用了命名空间的元素时将其绑定到了配置之中。现在，我们会看看如何明确地通过FilterChainProxy来配置它，伴随着它的小伙伴过滤器ExceptionTranslationFilter一起。典型的配置如下：

```
<bean id="filterSecurityInterceptor"

  class="org.springframework.security.web.access.intercept.FilterSecurityIntercepto
<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HTTP资源的安全操作。它需要一个对AuthenticationManager和一个对AccessDecisionManager的引用。它也提供

属性配置，从而可以应用到不同的HTTP　　URL请求。参考一下本教程《技术简介》一章最初的讨论。

FilterSecurityInterceptor可以用两种方法在配置属性中配置。第一种就像上面那样，使用<filter-security-metadata-source>命名空间元素。它和《命名空间》一节中的<http>元素很相似，但其中的<intercept-url>子元素只使用pattern和access属性。逗号用于为用于每个HTTP URL的不同的配置属性进行区分。第二种方式是写一个你自己的SecurityMetadataSource，但它超出了此文档的范围。不论方法如何，SecurityMetadataSource负责返回一个包含了所有的配置属性的List<ConfigAttribute>，它们与一个简单安全的HTTP相关。

你应当注意：FilterSecurityInterceptor.setSecurityMetadataSource()方法期望一个FilterInvocationSecurityMetadataSource实例。它是一个制造器接口，并且SecurityMetadataSource是它的子类。它简单地表示SecurityMetadataSource理解FilterInvocation`s。在这个简单的兴趣中，我们继续像`SecurityMetadataSource一样参考FilterInvocationSecurityMetadataSource，二者的在与大部分用户的关系上有一点小区别。

SecurityMetadataSource由命名空间语法创建，它获取一个特殊的FilterInvocation，获取方式是通过配置的pattern属性来匹配请求的URL。这与在命名空间配置的方式在行为上是一样的。默认将所有的表达式看作是Apache Ant路径，在更复杂的情况下也支持正则表达式。request-matcher属性用于指定pattern的类型。不能用同样的定义最小化表达式语法。例如，可以用正则表达式来代替之前的Ant路径的配置：

```
<bean id="filterInvocationInterceptor"

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

```
</bean>
```

Patterns一直都会被按它们被定义的顺序进行判定。因此在列表中把更特殊的patterns定义在没那么特殊的patterns的上方非常重要。你可以在上面的例子中看到这一点，更特殊的 /secure/super pattern放在没那么特殊的 /secure/ pattern的上面。如果它们被颠倒，那么 /secure/ pattern将会一直被匹配，并且你永远也没办法进入 /secure/super/ pattern。

11.2. 14.2 ExceptionTranslationFilter

`ExceptionTranslationFilter`在安全过滤器栈中位于`FilterSecurityInterceptor`之上。它自己不会做任何实际的安全操作，但它会捕获安全拦截器中的异常，并提供一个适当的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.LoginUrlAuthenticationEntryP
<property name="loginFormUrl" value="/login.jsp"/>
</bean>

<bean id="accessDeniedHandler"

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

11.2.1. AuthenticationEntryPoint

如果用户请求一个被保护的HTPP资源，但此时他们没有被认证，那么`AuthenticationEntryPoint`会被调用。一个恰当的`AuthenticationException`或者`AccessDeniedException`会被一个安全拦截器从调用栈中抛出，从而触发入口的开始方法。

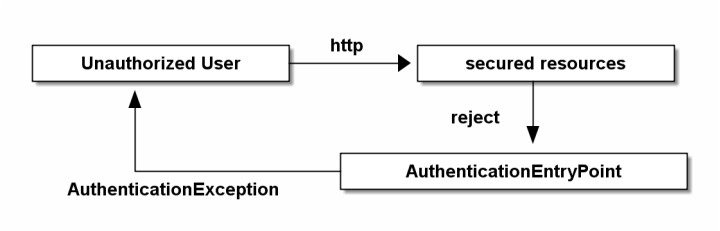

图 **11.1. UML** 未认证用户请求流程

这能够给用户返回适当的响应信息，从而让用户开始认证。我们在这里使用的LoginUrlAuthenticationEntryPoint会将用户重定向到另一个URL（通常是一个登陆页面）。具体的实现取决于你想要在你的应用中使用的认证机制。

11.2.2. AccessDeniedHandler

当用户已经认证完成，并且他们尝试访问无权访问的资源时会发生什么？通常这不会发生，因为你系统会根据用户权限，在页面上只显示用户能访问的URL。但也会有例外，例如用户直接输入了管理台的URL地址，或者直接在RESTful的URL上做了些许参数的改变。你的应用应该考虑到这样的情况，因此你应该在你的服务层中，加上基于URLs以及方法的安全校验。

如果用户已经登录，并且抛出了AccessDeniedException异常，那么说明用户无权进行操作。这种情况下，ExceptionTranslationFilter会调用第二个策略，即AccessDeniedHandler。通常情况下会使用AccessDeniedHandlerImpl，它给客户端发送一个403(禁止)响应吗。你也可以配置其他的实例（像上面那样）并设置一个错误页面，将用户请求直接forwards到此错误页面 [11]。页面可以是简单的“拒绝访问”页面，例如JSP，也可以是复杂的MVC控制器。你也可以根据自己的需要来做恰当的实现。

如果你使用了命名空间来配置用户应用，你也可以定制一个`AccessDeniedHandler`。参见命名空间附录详情 14.2.3 `SavedRequest`s 和RequestCache接口。

`ExceptionTranslationFilter`另外还负责在调用`AuthenticationEntryPoint`之前保存当前请求。这允许请求被保存到用户完成认证之后（参见之前的web认证概述）。在用户提交登录认证表单并成功认证之后，默认的`SavedRequestAwareAuthenticationSuccessHandler`会将用户重定向到源URL（参见下文）。

RequestCache封装了必要的功能来保存和检索HttpServletRequest实例。默认会使用HttpSessionRequestCache，它会在HttpSession中保存请求。在用户被重定向到源URL时，RequestCacheFilter负责实际的在cache中保存请求的认证。

通常你无须改变任何功能，除非保存请求的操作不适合你的应用或者默认配置无法解决你的问题。从Spring Security 3.0版本之后，所有这些接口的使用都是可拔插的。

11.3. SecurityContextPersistenceFilter

我们已经在《技术概览》一章中提到了的所有重要的过滤器，但你可能还想在这章更深入的了解它们。首先我们介绍了如何配置一个FilterChainProxy。最基本的配置只需要一个bean：

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

就像我们之前看到的，该过滤器有两个主要目标。它负责在HTTP请求间存储SecurityContext上下文，同时还负责在请求完成时清理SecurityContextHolder。清理保存了上下文的ThreadLocal是非常重要的，否则线程可能会在servlet容器的线程池中被替换，而特定的用户仍然保留着线程的引用。这一线程可能后续的阶段被使用，用错误的凭证来执行操作。

11.3.1. SecurityContextRepository

从Spring Security 3.0起，加载和保存security上下文被委托给一个独立的策略接口：

```
public interface SecurityContextRepository {

    SecurityContext loadContext(HttpRequestResponseHolder
 requestResponseHolder);

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

HttpRequestResponseHolder是一个简单的容器，它用于传入请求和响应对象，我们可以用封装类型重新实现它来替换这些操作。返回的内容会传递给过滤器链。

默认的实现是HttpSessionSecurityContextRepository，它将security上下文作为一个HttpSession属性来保存。最重要的配置参数是allowSessionCreation，它默认为true，这使得在需要为一个认证的用户保存security上下文时，HttpSessionSecurityContextRepository类会创建一个session（除非认证发生，并且security上下文的内容发生改变，否则它不会被创建）。如果你不想要session被创建，那么你可以将参数设置为false。

```
<bean id="securityContextPersistenceFilter" class="org.springframework.security.web.cor
    <property name="securityContextRepository">

  <bean class="org.springframework.security.web.context.HttpSessionSecurityContextRepos
            <property name="allowSessionCreation" value="false"></
property>
        </bean>
    </property>
</bean>
```

你也可以将null对象作为NullSecurityContextRepository实例，它会阻止安全上下文被保存，即使session已经在请求期间被创建。

11.4. UsernamePasswordAuthenticationFilter

我们已经介绍了3个主要的过滤器，它们总会在Spring Security web配置中被添加。还有三个过滤器会被命名空间<http>元素自动创建，并且它们无法被取缔。现在我们唯一缺少的就是用户认证机制。这是一个最通用的认证过滤器，并且它常常会被自定义 [13]。我们也可以在xml配置中用<form-login>元素来提供一个实现。需要分三个阶段进行配置：

- 配置一个LoginUrlAuthenticationEntryPoint与一个登录页面的URL，并将其设置到ExceptionTranslationFilter。
- 实现登录页面（用JSP或者MVC控制器）。
- 配置一个UsernamePasswordAuthenticationFilter到应用上下文中。
- 添加过滤器bean到你的过滤器链proxy中（注意添加顺序）

登录表单简单的包含用户名和密码输入框，并posts到过被滤器监听的URL（默认是/login）。基本过滤器配置看起来像这样：

```
<bean id="authenticationFilter" class="org.springframework.security.web.authenticat
property></bean>
```

11.4.1. 认证成功和失败时的应用程序流程

过滤器调用配置的AuthenticationManager来处理每个认证请求。认证成功或认证失败的处理流程分别由AuthenticationSuccessHandler和AuthenticationFailureHandler策略接口来控制。过滤器包含一些参数来允许你定制它们[14]。诸如SimpleUrlAuthenticationSuccessHandler、SavedRequestAwareAuthenticationSuccess这些接口的标准的实现都会被提供。你可以查看这些类的Javadoc以及AbstractAuthenticationProcessingFilter的源码来学习它们如何工作，并且提供哪些功能。

认证成功时，最终的Authentication对象会被放置到SecurityContextHolder之中。然后会调用配置的AuthenticationSuccessHandler来将用户redirect或者forward到合适的目标页面。默认情况下会使用SavedRequestAwareAuthenticationSuccessHandler，这意味着用户会被redirected到他们在登录之前请求的页面。

ExceptionTranslationFilter缓存了用户一开始的request。在用户认证时，请求操作会从被缓存的request中获取并redirect到源URL。源请求之后会被重建并被覆盖。

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

[11] 我们使用了forward，因此SecurityContextHolder仍然包含了principal的详细信息，这对于展示用户信息非常有用。在老版本的Spring Security，我们依赖servlet容器来操作403错误信息，它缺少有用的上下文信息。

[12] 在Spring Security 2.0和更早的版本，这个过滤器叫做HttpSessionContextIntegrationFilter，并且由过滤器自己处理所有保存上下文的所有操作。如果你熟悉这个类，那么大部分可用的配置选项都可以在HttpSessionSecurityContextRepository之中找到。

[13] 由于一些历史原因，在Spring Security 3.0之前的版本，这个过滤器叫做AuthenticationProcessingFilter，并且入口叫做AuthenticationProcessingFilterEntryPoint。由于框架现在支持很多不同的认证框架，它们便在3.0之后拥有了更具体的名字。

[14] 在3.0之前的版本，应用程序流程已经进化到一个阶段，在这个类和策略插件中，它可以由最小化的属性来控制。3.0之后我们决定进行重构，使两种策略相互分离。

12

Servlet API集成

本节描述了Spring Security如何集成Servlet API。servletapi-xml示例应用展示了每个这些方法的用法。

12.1. Servlet 2.5+集成

12.1.1. HttpServletRequest.getRemoteUser()

HttpServletRequest.getRemoteUser()会返回一个SecurityContextHolder.getContext().getAuthentication().getName()的结果，它代表了当前的用户名。如果你想要在应用程序中显示当前用户名，这会非常有用。另外，检查它是否为null来判定用户是已经认证了还是仍然是匿名。了解用户是否已经认证对于判定某些UI元素是否显示非常有用（例如注销链接只会为未认证的用户显示）。

12.1.2. HttpServletRequest.getUserPrincipal()

HttpServletRequest.getUserPrincipal()会返回一个SecurityContextHolder.getContext().getAuthentication()的结果。这意味着它是一个Authentication，在使用基于用户名和密码的认证时，它是一个典型的UsernamePasswordAuthenticationToken实例。这在你需要关于你的用户的附加信息时非常有用。例如，你可能创建了一个自定义的UserDetailsService，它返回一个自定义的UserDetails，并包含了用户的姓与名。你可以像下面这样获取这些信息：

```
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();
```

注意，这是一个典型的糟糕的实践，它在你的应用中执行了太多逻辑。作为替代，一种办法是将它集成到Servlet API中，从而减少与Spring Security的耦合性。

12.1.3. HttpServletRequest.isUserInRole(String)

`HttpServletRequest.isUserInRole(String)`会将role传递到`isUserInRole(String)`，从而判断`SecurityContextHolder.getContext().getAuthentication().getAuthorities()`是否包含一个`GrantedAuthority`。传统的用户不应该传递`ROLE`前缀到方法中，它们会被自动添加。例如，如果你想要判断当前用户是否要授权“ROLE_ADMIN”，你可以像下面这样：

```
boolean isAdmin = httpServletRequest.isUserInRole("ADMIN");
```

这在判断是否需要显示UI组件时非常有用。例如，你可能只想在当前用户是admin时显示admin链接。

12.2. Servlet 3+集成

下面这节描述Spring Secueity集成的Servlet 3方法。

12.2.1. HttpServletRequest.authenticate(HttpServletRequest,HttpServletRe

`HttpServletRequest.authenticate(HttpServletRequest,HttpServletResponse)`方法可以用于确保用户被认证。如果他们没有被认证，那么配置的`AuthenticationEntryPoint`会要求用户进行认证（例如重定向到登录页面）。

12.2.2. HttpServletRequest.login(String,String)

`HttpServletRequest.login(String,String)`方法可以被用于通过当前的`AuthenticationManager`进行用户认证。例如，下面的例子试图对用户名为"user"密码为"password"的用户进行认证。

```
try {
```

```
    httpServletRequest.login("user","password");
    } catch(ServletException e) {
    // fail to authenticate
}
```

如果你想用Spring Security来捕获失败的认证请求，那么ServletException并非必须的。

12.2.3. HttpServletRequest.logout()

HttpServletRequest.logout()方法可以被用于注销当前用户。

通常这意味着SecurityContextHolder会被清除，HttpSession会被无效化，任何"Remember Me"认证都会被清除。但是，被配置的LogoutHandler实现会变得依赖于你的Spring Security配置。注意，在HttpServletRequest.logout()被调用之后，你仍需要写一种响应方式，这非常重要。通常这会调用一个到欢迎页面的重定向。

12.2.4. AsyncContext.start(Runnable)

AsynchContext.start(Runnable)方法能确保你的认证被传送到新的线程。使用Spring Security的并发支持，Spring Security重写AsyncContext.start(Runnable)，从而确保当前的SecurityContext在处理Runnable时能够使用。例如，下面这个例子会显示当前用户的Authentication：

```
final AsyncContext async = httpServletRequest.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);
```

```
        }
    }
});
```

12.2.5. 异步Servlet支持

如果你正在使用基于Java的配置，那么你可以直接开始。如果你正在使用XML配置，那么它们需要一些必要的更新。第一步是确保你已经更新了web.xml到至少3.0的schema版本：

```
<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>
```

接下来你需要确保你的`springSecurityFilterChain`被安装来处理异步请求。

```
<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>
```

搞定了！现在Spring Security会确保你的`SecurityContext`能够接收异步的请求！

它如何工作的呢？如果你不是真的感兴趣，那么你可以调过本章剩余部分，否则继续往下看吧。它们大部分根据Servlet规范进行构建，但还有一些修改，Spring Security要确保合适于异步请求的事情能够运转。在Spring Security 3.2之前，一旦`HttpServletResponse`被提交，`SecurityContext`就会自动被`SecurityContextHolder`保存。这在异步环境下会引起问题。例如考虑一下：

```
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();
```

问题在于，线程并不知道Spring Security，因此SecurityContext并不会传递给它。这意味着在我们提交HttpServletResponse时还没有SecuriytContext。当Spring Security在HttpServletResponse提交时自动保存SecurityContext，我们就会丢失我们的用户登录记录。

在3.2版本之后，Spring Security非常机智，不再在HttpServletRequest.startAsync()被调用时提交HttpServletResponse并自动保存SecurityContext。

12.3. Servlet 3.1+集成

下面这节描述了Spring Security集成的Servlet 3.1方法。

12.3.1. HttpServletRequest#changeSessionId()

HttpServletRequest.changeSessionId()是默认的方法，用于在Servlet 3.1过更高版本中对Session固化攻击进行防范。

13

基础与摘要认证

基础与摘要认证是可选的认证机制，它们在web应用中非常流行。基础认证通常用于无状态的客户端，它们在每个请求中都传递凭证信息。将其与基于表单的认证相结合的做法非常普遍，应用程序用于提供一个基于浏览器的用户接口以及web服务。然而，基础认证提交的是明文密码，因此它应该只用于像是HTTPS这样的加密传输层。

13.1. BasicAuthenticationFilter

`BasicAuthenticationFilter`负责处理基础认证凭证，它在HTTP头中被提供。这可以用于认证调用过程，该过程由Spring远程协议创建（例如Hessian和Burlap），就像通常的浏览器用户代理（例如Firefox和IE）。标准的HTTP Basic Authentication被定义在RFC 1945的第11节，`BasicAuthenticationFilter`符合RFC规则。基础认证一种具有吸引力的认证手段，因为它非常广泛地被部署，其用户代理与实现都非常的简单（它只是一个用户名:密码的Base64编码，在HTTP头中被指定）。

13.1.1. 配置

为了实现HTTP基础认证，你需要添加一个`BasicAuthenticationFilter`到你的过滤器链中。应用上下文应该包含`BasicAuthenticationFilter`还有它需要的小伙伴：

```
<bean id="basicAuthenticationFilter"
class="org.springframework.security.web.authentication.www.BasicAuthenticationFilte
<property name="authenticationManager" ref="authenticationManager"/>
<property name="authenticationEntryPoint" ref="authenticationEntryPoint"/
>
</bean>
```

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

配置的AuthenticationManager处理每一个认证请求。如果认证失败，配置的AuthenticationEntryPoint会用来再次启动认证过程。通常你会将过滤器结合BasicAuthenticationEntryPoint来使用，它会在适当的头中返回一个401响应，告诉你要重新进行HTTP基本认证。如果认证成功，那么通常会返回一个Authentication对象，并将此对象放到SecurityContextHolder中。

如果认证事件成功了，或者认证被HTTP头告知不支持该认证请求，过滤器链通常都会继续。只有认证失败时，过滤器链才会被阻断，然后调用AuthenticationEntryPoint。

13.2. DigestAuthenticationFilter

DigestAuthenticationFilter能够处理摘要认证凭证，凭证在HTTP头信息中。摘要认证试图解决很多基础认证的缺点，尤其是确保凭证信息不会以明文方式发送和传输。很多用户代理支持摘要认证，包括FireFox和IE。HTTP摘要认证的标准被定义在RFC 2617中，它是一个比RFC 2069规定的更新的版本。大多数的用户代理都实现了RFC 2617。Spring Security的DigestAuthenticationFilter兼容在RFC 2617中规定的"auth"保护特性(qop)，该特性提供对RFC 2069的兼容。如果你需要使用非加密的HTTP（例如TLS/HTTPS），同时又希望尽可能增大认证过程的安全性，那么摘要认证提供了更有吸引力的选择。实际上，摘要认证是WebDAV规范中强制性的需求，就像RFC 2518第17.1节说的那样。

注意

> 你不应该在现在的应用中使用摘要，它们不够安全。最明显的问题就是你必须以明文、密文或者MD5格式存储密码。所有这些存储格式都被证明不安全。作为替代，你应该使用一种合适的密码哈希（例如BCrypt，PBKDF2，SCrypt等等）。

摘要认证的核心是"nonce"。这是一个服务器产生的值。Spring Security的nonce采用如下格式：

```
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
```

DigestAuthenticatonEntryPoint具有一个参数，它能指定用于生成nonce tokens的键值，它与nonceValiditySeconds参数一起用于判断完结时间（默认的300秒等于5分钟）。nonce会一直有效，摘要由包括用户名、密码、nonce、被提交的URI、客户端生成的nonce（只是一个用户代理在每次请求时生成的随机值），还有域名等字符串链接并计算出来，然后执行一个MD5的哈希。服务器和用户代理都执行这一摘要计算，如果它们包含的其中一个值不一致（例如密码）那么哈希的结果也会不一样。在Spring Security实现中，如果服务器生成的nonce仅有预期（但摘要依然是有效的）,DigestAuthenticationEntryPoint会发送一个"stale=true"头。这告诉用户客户端不需要打扰用户（例如用户名和密码等是正确的），但需要简单的用一个新nonce进行重试。

DigestAuthenticationEntryPoint的nonceValiditySeconds参数根据你的应用填入一个适当的值。非常安全的应用应该要考虑到，在nonce到期之前（expirationTime指定了到期时间），被拦截的应用程序头可以被用于模拟principal。在适当的设置被选择时，这是个关键的principal，但要尽可能保护应用程序在第一个实例中不被以TLS/HTTPS方式运行，这很重要。

这一段翻译的有点拗口，欢迎更好的翻译...

由于摘要认证的实现更为复杂，所以用户代理经常出问题。例如，IE不能在后续同样的session请求中提供“opaque”token。因此，Spring Security过滤器将所有的状态信息压缩到"nonce"token作为替代。在我们的测试中，Spring Security的实现能够很好的运行在FireFox和IE中，能够正确的操作nonce超时等。

13.2.1. 配置

让我们回忆一下概念，看看如何使用。为了实现HTTP摘要认证，在过滤器链中定义一个DigestAuthenticationFilter非常必要。在应用上下文中需要定义DigestAuthenticationFilter还有他的小伙伴：

```
<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。如果你在你的DAO [15] 中使用了编码的密码，那么摘要认证将无法工作。DAO小伙伴和UserCache一起，通常会直接和DaoAuthenticationProvider分享。authenticationEntryPoint参数必须是DigestAuthenticationEntryPoint，因此DigestAuthenticationFilter可以获取正确的域名和键值用于摘要计算。

就像BasicAuthenticationFilter，如果认证成功，Authentication请求口令会被放进SecurityContextHolder。如果认证事件成功，或者认证由于HTTP头没有包含摘要认证请求，导致没有进行认证，过滤器链都会正常执行。只有认证失败，并且AuthenticationEntryPoint被调用，过滤器链才会被阻断，就像上一段介绍的那样。

摘要认证的RFC提供附加的功能范围，以进一步增加安全。例如，nonce可以在每一次请求时被改变。尽管如此，Spring Security实现设计了最简单的实现（并且毫无疑问，用户代理的不兼容性会被暴露），并且避免保存服务器网站的状态。如果你希望去浏览这些功能的更多详情，那么就去看看RFC 2617。我们尽我们所能，让Spring Security的实现尽可能遵循最小化的这个RFC标准。

[15] 可以用 HEX(MD5(username:realm:password)) 的格式编码密码，并为DigestAuthenticationFilter.passwordAlreadyEncoded 设置一个true值，其它的密码编码无法在摘要认证下工作。

14

Remember-Me认证

14.1. 综述

Remember-me或者持久登录认证指的是web站点能够在不同的sessions之间记住一个principal的凭证。传统的实现方式是将cookie发送到浏览器，cookie会被未来的sessions检查，并触发自动登录。Spring Security为此操作的发生提供必要的hooks，并且有两种具体的remember-me实现。一种使用哈希来保护基于cookie口令的安全，另一种使用数据库或者其它持久化存储机制来保存生成的口令。

注意两种实现都需要一个`UserDetailsService`。如果你正在使用一个认证provider，而它没有使用`UserDetailsService`（例如LDAP provider），那么remember-me就无法工作了，除非你在你的应用上下文中提供一个`UserDetailsService`。

14.2. 简单的基于哈希口令的方式

此方法使用哈希来实现有用的remember-me策略。本质上，cookie在成功的交互式认证时被发送到浏览器上，cookie可能像下面这样：

```
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
```

remember-me token只在指定的周期中有效，并且提供的用户名与密码还有key不能改变。尤其是这存在一种潜在的安全问题，除非token到期，对于从任何用户代理获取的remember-me token都是可用的。同样的问题也存在于摘要认证。如果一个principal被知晓，token也被捕获，那么攻击者就能够很容易的改变用户的密码，并且立刻使所有的remember-me token无效。如果需要更重要的安全，你应该使用下一节描述的方法，或者简单的不再使用remember-me服务。

如果你熟悉命名空间配置那章的主题描述，你可以很简单的添加一个<remember-me>元素来启用remember-me认证。

```
<http>
...
<remember-me key="myAppKey"/>
</http>
```

UserDetailsService通常会被自动选择。如果你的应用上下文中超过一个，你需要通过user-service-ref属性来指定使用哪一个，它的值是你的`UserDetailsService`bean的名字。

14.3. 持久化Token的方式

这一方式基于 http://jaspan.com/improved_persistent_login_cookie_best_practice 这篇文章，但也有一些小改动[16]。为了通过命名空间配置使用这一方式，你应该添加一个数据源引用：

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

数据库应该包含一个persistent_logins表，它由下面的SQL（或者其它等价的SQL）创建：

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

14.4. Remember-Me接口与实现

Remember-me被与UsernamePasswordAuthenticationFilter一起使用，并且在AbstractAuthenticationProcessingFilter的子类中由hooks实现。它也在BasicAuthenticationFilter中被使用。hooks会在适当的时间调用具体的RememberMeServices。此接口像这样：

```
Authentication autoLogin(HttpServletRequest request, HttpServletResponse
 response);

void loginFail(HttpServletRequest request, HttpServletResponse
 response);

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

方法中做了什么可以参考JavaDocs，虽然在这里我们看到AbstractAuthenticationProcessingFilter只调用了loginFail()和loginSuccess()方法。在SecurityContextHolder没有包含一个Authentication时，autoLogin()方法由RememberMeAuthenticationFilter调用。因此，这个接口提供了根本的remember-me实现，它有足够多的与认证相关的事件的通知，并且说明了无论何时，候选的web请求都可能包含一个cookie，并且期望被记住。这一设计允许任意数量的remember-me实现策略。我们在上面看到Spring Security提供了两种实现。我们会依次介绍它们。

14.4.1. TokenBasedRememberMeServices

此实现支持简单的方法，它被描述在 17.2节_简单的基于哈希口令的方式[1]。TokenBasedRememberMeServices生成一个RememberMeAuthenticationToken，它被RememberMeAuthenticationProvider处理。一个键值在这个认证provider和TokenBasedRememberMeServices共享。另外，TokenBasedRememberMeServices需要一个UserDetailsService，它可以重新获取用户名和密码，并对签名进行比较，然后生成RememberMeAuthenticationToken来包含正确的GrantedAuthority`s。一些注销命令

[1] http://docs.spring.io/spring-security/site/docs/4.1.3.RELEASE/reference/htmlsingle/#remember-me-hash-token

的顺序应该被通过应用来提供，如果用户做了请求，那么cookie就会无效。`TokenBasedRememberMeServices也实现了Spring Security的LogoutHandler接口，因此可以使用LogoutFilter来自动清理cookie。

应用上下文中启用remember-me服务，那么需要一些beans：

```
<bean id="rememberMeFilter" class=
"org.springframework.security.web.authentication.rememberme.RememberMeAuthenticationFil
<property name="rememberMeServices" ref="rememberMeServices"/>
<property name="authenticationManager" ref="theAuthenticationManager" />
</bean>

<bean id="rememberMeServices" class=
"org.springframework.security.web.authentication.rememberme.TokenBasedRememberMeService
<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>
```

不要忘了使用UsernamePasswordAuthenticationFilter.setRememberMeServices()，并将RememberMeServices作为参数传递进去，将RememberMeAuthenticationProvider包含到你的AuthenticationManager.setProviders()列表，并添加RememberMeAuthenticationFilter到你的FilterChainProxy（通常紧跟着UsernamePasswordAuthenticationFilter之后）。

14.4.2. PersistentTokenBasedRememberMeServices

这个类可以像TokenBasedRememberMeServices一样的方式被使用，但它还需要配置一个PersistentTokenRepository来存储tokens。有两个标准：

- InMemoryTokenRepositoryImpl 这个只用于测试

- `JdbcTokenRepositoryImpl` 将tokens保存到数据库中 数据库schema在 17.3节持久化口令的方式[2] 中被描述。

[16] 本质上，用户名不会被包含在cookie中，这样可以预防非必须但有效登录名被暴露。这被描述在本文的讨论小节。

[2] http://docs.spring.io/spring-security/site/docs/4.1.3.RELEASE/reference/htmlsingle/#remember-me-persistent-token

15

CORS

Spring Framework提供了 第一个类来支持CORS[1]。CORS必须在Spring Security之前被处理，因为准备提交的请求不会包含任何cookies（也就是JSESSIONID）。如果请求没有包含任何cookies，并且Spring Security在前，那么请求会判断用户无法认证（因为他们的请求中没有cookies！）并拒绝它。

为了确保CORS先被处理，有一种简单的方式，就是使用`CorsFilter`。用户可以通过使用下面的配置，提供一个`CorsConfigurationSource`，从而集成Spring Security的`CorsFilter`：

```
@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"));
```

[1] http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#cors

```
        UrlBasedCorsConfigurationSource source = new
 UrlBasedCorsConfigurationSource();
        source.registerCorsConfiguration("/**", configuration);
        return source;
    }
}
```

或者用XML：

```
<http>
    <cors configuration-source-ref="corsSource"/>
    ...
</http>
<b:bean id="corsSource" class="org.springframework.web.cors.UrlBasedCorsConfigurationSo
    ...
</b:bean>
```

如果你在使用Spring MVC的CORS支持，你可以省略CorsConfigurationSource，Spring Security会引用Spring MVC中提供的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()
            ...
    }
}
```

或者在XML中：

```
<http>
    <!-- Default to Spring MVC's CORS configuraiton -->
    <cors />
    ...
```

```
</http>
```

部分 V. Authorization

The advanced authorization capabilities within Spring Security represent one of the most compelling reasons for its popularity. Irrespective of how you choose to authenticate - whether using a Spring Security-provided mechanism and provider, or integrating with a container or other non-Spring Security authentication authority - you will find the authorization services can be used within your application in a consistent and simple way.

In this part we'll explore the different `AbstractSecurityInterceptor` implementations, which were introduced in Part I. We then move on to explore how to fine-tune authorization through use of domain access control lists.

部分 VI. Additional Topics

In this part we cover features which require a knowledge of previous chapters as well as some of the more advanced and less-commonly used features of the framework.

部分 VII. Spring Data Integration

Spring Security provides Spring Data integration that allows referring to the current user within your queries. It is not only useful but necessary to include the user in the queries to support paged results since filtering the results afterwards would not scale.

部分 VIII. Appendix