

Spring Security 3

目录

Spring Security 3	1
第一章 一个不安全应用的剖析.....	8
安全审计.....	9
关于样例应用.....	9
JBoss Pet 应用的架构	9
应用所使用的技术.....	10
查看审计结果.....	10
认证.....	12
授权.....	13
数据库认证安全.....	13
敏感信息.....	13
数据传输层保护.....	13
使用 Spring Security 解决安全问题.....	14
为什么使用 Spring Security?	14
小结.....	14
第二章 Spring Security 起步.....	15
安全的核心概念.....	16
认证.....	16
授权.....	17
三步之内使我们的应用变得安全.....	19
实现 Spring Security 的 XML 配置文件.....	19
添加 Spring DelegatingFilterProxy 到 web.xml 文件.....	20
添加 Spring Security XML 配置文件的应用到 web.xml.....	20
注意这些不足之处.....	22
常见问题.....	23
安全的复杂之处：安全 web 请求的架构.....	23
请求是怎样被处理的?	24
在 auto-config 场景下，发生了什么事情？	27
用户是怎样认证的？	27
请求是怎样被授权的？	33
总结.....	38
第三章 增强用户体验.....	40
自定义登录页.....	41
实现自定义的登录页.....	41
理解退出功能.....	45
在站点页头上添加“Log Out”链接	45
退出是怎么实现的.....	45
Remember me.....	47

实现 remember me 选项	48
Remember me 是怎样实现的.....	48
Remember me 是否安全？	52
实现修改密码管理.....	58
扩展基于内存的凭证存储以支持修改密码.....	58
小结.....	62
第四章 凭证安全存储.....	63
使用数据库后台的 Spring Security 认证.....	64
配置位于数据库上的认证存储.....	64
基于数据库后台的认证是如何实现的.....	66
实现自定义的 JDBC UserDetailsService.....	67
基于 JDBC 的内置用户管理.....	68
JdbcDaoImpl 的高级配置.....	70
配置基于组的授权.....	71
使用遗留的或用户自定义的 schema 实现基于数据库的认证.....	72
配置安全的密码.....	74
配置密码编码.....	76
你是否愿意在密码上添加点 salt?	78
配置 salted 密码.....	79
增强修改密码功能.....	82
配置自定义的 salt source	82
将 Remember me 功能迁移至数据库.....	85
配置基于数据库的 remember me tokens.....	85
基于数据库后台的持久化 tokens 是不是更安全？	86
用 SSL 保护你的站点	87
配置 Apache Tomcat 以支持 SSL.....	87
对站点进行自动的安全保护.....	88
小结.....	90
第五章 精确的访问控制.....	92
重新思考应用功能和安全.....	93
规划应用安全.....	93
规划用户角色.....	93
规划页面级权限.....	94
实现授权精确控制的方法.....	95
使用 Spring Security 的标签库有选择地渲染内容.....	95
使用控制器逻辑进行有条件渲染内容.....	98
配置页面内授权的最好方式是什么？	99
保护业务层.....	99
保护业务层方法的基本知识.....	100
几种实现方法安全的方式.....	102
方法的安全保护是怎样运行的？	105
方法安全的高级知识.....	106
使用 bean 包装类实现方法安全规则.....	107
包含方法参数的实现方法安全规则.....	108

参数绑定是如何实现的？	108
使用基于角色的过滤保护方法的数据安全	109
关于方法安全的警告	113
小结	114
第六章 高级配置和扩展.....	115
实现一个自定义的安全过滤器	115
在 servlet 过滤器级别实现 IP 过滤	115
实现自定义的 AuthenticationProvider	118
通过 AuthenticationProvider 实现一个简单的单点登录	118
连接 AuthenticationProvider	123
使用请求头模拟单点登录	123
实现自定义 AuthenticationProviders 时要考虑的事项	124
Session 的管理和并发	125
配置 session fixation 防护	125
通过 session 的并发控制增强对用户的保护	129
Session 并发控制的其它好处	131
理解和配置异常处理	134
配置 “Access Denied” 处理	134
什么会触发 AccessDeniedException	136
AuthenticationEntryPoint 的重要性	137
手动配置 Spring Security 设施的 bean	137
总体理解 Spring Security bean 的依赖关系	137
重新配置 web 应用	138
配置一个最小的 Spring Security 环境	139
Spring Security 基于 bean 的高级配置	143
Session 生命周期的调整元素	143
手动配置其它通用的服务	144
明确配置 SpEL 表达式和投票器	148
认证事件处理	149
配置认证事件的监听器	150
大量的应用事件	152
构建一个自定义实现的 SpEL 表达式处理器	153
小结	154
第七章 访问控制列表（ACL）.....	155
使用访问控制列表保护业务对象	155
Spring Security 中的访问控制列表	156
支持 Spring Security ACL 的基本配置	157
定义一个简单的目标场景	157
添加 ACL 表到 HSQL 数据库中	158
配置访问决策管理器	160
配置 ACL 的支持 bean	161
创建一个简单的 ACL entry	164
高级 ACL 话题	166
Permission 如何工作	166

自定义 ACL permission 声明	168
在 JSP 中使用 Spring Security JSP tag 库启动 ACL	171
支持 ACL 的 Spring 表达式语言.....	171
易变的 ACL (Mutable ACLs) 和授权	172
Ehcache ACL 缓存	175
典型 ACL 部署所要考虑的事情.....	177
关于 ACL 的伸缩性和性能模型.....	177
不要忽视自定义开发的成本.....	178
我应该用 Spring Security 的 ACL 吗？	179
小结.....	179
第八章 对 OpenID 开放	180
OpenID 承诺的世界	180
注册一个 OpenID	181
使用 Spring Security 启用 OpenID 认证	181
编写一个 OpenID 登录表单.....	181
在 Spring Security 中配置支持 OpenID	182
添加 OpenID 用户	183
OpenID 用户的注册问题	184
OpenID 标识是如何处理的	184
使用 OpenID 实现注册	186
属性交换 (Attribute Exchange)	190
在 Spring Security OpenID 中启用 AX	191
现实世界的 AX 的支持和限制	192
支持 Google OpenID	193
OpenID 安全吗？	194
小结.....	194
第九章 LDAP 目录服务	195
理解 LDAP	195
LDAP	195
通用的 LDAP 属性名	196
运行一个嵌入式的 LDAP 服务	197
配置基本的 LDAP 集成	197
配置 LDAP 服务器引用	198
启用 LDAP AuthenticationProvider.....	198
解决嵌入式 LDAP 的问题	198
理解 Spring LDAP 认证如何工作	199
认证用户凭证.....	199
确定用户的角色.....	200
匹配 UserDetails 的其它属性	202
LDAP 的高级配置	202
实例 JBCP LDAP 用户.....	203
密码对比与绑定认证.....	203
配置 UserDetailsContextMapper.....	205
使用可代替的密码属性.....	207

使用 LDAP 作为 UserDetailsService	207
集成外部的 LDAP 服务器	209
明确的 LDAP bean 配置	209
配置外部的 LDAP 服务器引用	209
配置 LdapAuthenticationProvider	209
通过 LDAP 集成 Microsoft Active Directory	211
委托角色查询给 UserDetailsService	213
小结	213
第十章 使用中心认证服务（CAS）进行单点登录	213
CAS 简介	214
CAS 认证整体流程	214
Spring 与 CAS	215
CAS 的安装和配置	216
基本的 CAS 集成配置	216
添加 CasAuthenticationEntryPoint	217
启用 CAS 票据校验	218
使用 CasAuthenticationProvider 证明可靠性	219
高级 CAS 配置	220
从 CAS assertion 中获取属性	220
属性查询的用处	228
其它的 CAS 功能	229
小结	229
第十一章 客户端证书认证（Client Certificate Authentication）	230
客户端证书认证是怎样工作的	231
建立客户端证书认证的设施	232
理解公钥设施的目的	232
创建客户端证书密钥对	233
配置 Tomcat 的 trust store	233
导入证书到浏览器中	234
测试	235
客户端证书认证的问题解决	236
在 Spring Security 中配置客户端证书认证	237
使用 security 命名空间配置客户端证书认证	237
Spring Security 怎样使用证书信息	237
Spring Security 证书认证怎样实现	238
其它未解决的问题	239
支持双模认证（Dual-Mode authentication）	240
使用 Spring bean 配置客户端证书认证	241
基于 bean 配置的其它功能	242
实现客户端证书认证要考虑的事情	243
小结	244
第十二章 Spring Security 扩展	244
Spring Security 扩展	244
Kerberos 和 SPNEGO 认证入门	245

在 Spring Security 实现 Kerberos 认证	246
Kerberos Spring Security 认证整体流程	247
准备工作.....	247
配置 Kerberos 相关的 Spring bean	249
织入 SPNEGO 到 security 命名空间中.....	250
添加应用服务器所在机器到 Kerberos 域中	251
对 Firefox 用户的特殊考虑.....	252
问题解决.....	252
配置 LDAP UserDetailsService 使用 Kerberos.....	254
使用 Kerberos 与 form 登录	255
小结.....	256
第十三章 迁移到 Spring Security 3.....	257
从 Spring Security2 进行迁移	257
Spring Security3 的增强	257
Spring Security 配置的变化	258
AuthenticationManager 配置的重新组织.....	258
Session 管理选项的新配置语法.....	259
自定义过滤器配置的变化.....	259
CustomAfterInvocationProvider 的变化.....	260
小的配置变化.....	261
包和类的变化.....	261
小结.....	262
附录：参考材料.....	264
JBCP Pets 示例代码起步	264
可用的应用事件.....	265
Spring Security 的虚拟 URL	266
方法安全的明确 bean 配置.....	266
逻辑过滤器名字迁移参考.....	269

第一章 一个不安全应用的剖析

毫无疑问，安全是任何一个写于 21 世纪的 web 工程中最重要的架构组件之一。在这样一个时代，计算机病毒、犯罪以及不合法的员工一直存在并且持续考验软件的安全性试图有所收益，因此对你负责的项目综合合理地使用安全是至关重要的一个元素。

本书的写作遵循了这样的一个开发模式，这个模式我们感觉提供了一个有用的前提来解决复杂的话题——即使用一个基于 Spring3 的 web 工程作为基础，以理解使用 Spring Security3 使其保证安全的概念和策略。

不管是不是已经使用 Spring Security 还是只是对这个软件有兴趣，就都会在本书中得到有用的信息。

在本节的内容中，你能够：

- 检查一个虚拟安全审计的结果
- 讨论 web 应用通常的一些安全问题
- 学习软件安全中的几个核心词汇和概念

如果你已经熟悉基本的安全术语，你可以直接跳到第二章：Spring Security 起步，在哪里我们将涉及这个框架的细节。

安全审计

这是你作为软件开发人员在 Jim Bob Circle Pant Online Pet Store(JBCPPets.com)工作的一个清晨，你正在喝你的第一杯咖啡的时候，收到了你主管的以下邮件：

To: Star Developer <stardev@jbcppets.com>

From: Super Visor <theboss@jbcppets.com>

Subject: 安全审计

Star,

今天，有一个第三方的安全公司要审计我们的电子商务网站。尽管我知道你在设计网站的时候已经把安全考虑在内了，但请随时解决他们可能发现的问题。

Super Visor

什么？你在设计应用的时候并没有过多考虑安全的问题？似乎你有许多的东西要向安全审计人员学习。首先，让我们花一点时间检查一下要审计的应用吧。

关于样例应用

尽管在本书的后续内容中我们要模拟虚拟的场景，但这个应用的设计以及我们对其进行的改造都是基于现实世界中真实使用 Spring 的工程。

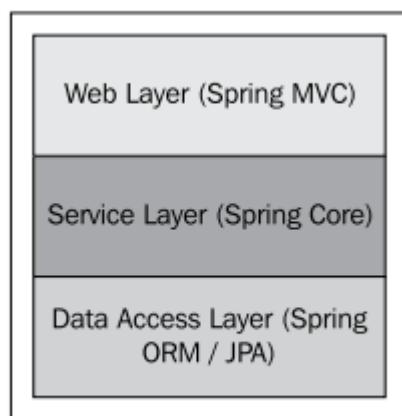
这个应用的设计很简单，使得我们能够关注于重要的安全方面而不会过多关注 ORM 的细节和复杂的 UI 技术。我们期望你能够参考其他的资料来掌握示例代码中所涉及功能的技术。

代码是基于 Spring 和 Spring Security3 编写而成的，但是它很容易改变为 Spring Security2 的例子。关于 Spring Security 2 和 3 的细节差异，可以参照第十三章：迁移至 Spring Security，可以作为将示例代码转换成 Spring Security2 的帮助材料。

不要以本应用作为基础去构建真实的 Pet Store 在线应用。本应用已经有意识的构建地简单并关注于本书所要阐述的概念和配置。

JBCP Pets 应用的架构

本应用遵循标准的三层结构，包括 web 层、服务层和数据访问层，如下图所示：



web 层封装了 MVC 的代码和功能。在示例代码中，我们使用了 Spring MVC 框架，但是我们可以一样容易的使用 Spring Web Flow, Struts 甚至是一个对 Spring 友好的 web stack 如 Apache Wicket。

在一个典型使用 Spring Security 的 web 应用中，大量配置和参数代码位于 web 层。所以，如果你没有 web 应用开发，尤其是 Spring MVC 的经验，在我们进入更复杂的话题前，你最好仔细看一下基础代码并确保你能理解。再次强调，我们已经尽力让我们的应用简单，把它构建成一个 pet store 只是为了给它一个合理的名字和轻量级的结构。可以将其与复杂的 Java EE Pet Clinic 示例作为对比，那个示例代码展现了很多技术的使用指导。

服务层封装了应用的业务逻辑。在示例应用中，我们在数据访问层前做了一个很薄的 façade 用来描述如何在特殊的点周围保护应用的服务方法。

在典型的 web 工程中，这一层将会包括业务规则校验，组装和分解 BO 以及交叉的关注点如审计等。

数据访问层封装了操作数据库表的代码。在很多基于 Spring 的工程中，这将会在这里发现使用了 ORM 技术如 hibernate 或 JPA。它为服务层暴露了基于对象的 API。在示例代码中，我们使用基本的 JDBC 功能完成到内存数据库 HSQL 的持久化。

在典型的 web 工程中，将会使用更为复杂的数据访问方式。因为 ORM，即数据访问，开发人员对其很迷惑。所以为了更清晰，这一部分我们尽可能的对其进行简化。

应用所使用的技术

我们使用了一些每个 Spring 程序员都会遇到的技术和工具，以使得示例应用很容易的运行起来。尽管如此，我们还是提供了补充的起步资料信息在附录：参考资料。

我们建立使用如下的 IDE 以提高开发的效率并使用本书的示例代码：

- Eclipse 3.4 或 3.5 Java EE 版本可以在以下地址获得：

<http://www.eclipse.org/downloads/>

- Spring IDE2.2(2.2.2 或更高)可以在以下地址获得：<http://springide.org/blog>

在本书的示例和截图中，你会看到 Eclipse 和 Spring IDE，所以我们建议你使用它们。

你可能希望使用免费的 Spring Tool Suite (STS) Eclipse 插件，它作为 Eclipse 的一个插件由 Spring Source 开发，其包含了下一代的 Spring IDE 功能（可以在以下地址下载：<http://www.springsource.com/products/springsource-tool-suite-download>）。一些用户不喜欢它的侵入性和 SpringSource 的标示，但是你如果从事 Spring 相关的开发，它提供了很多有用的功能。

我们提供了 Eclipse3.4 兼容的工程以允许你在 Eclipse 中构建和部署代码到 Tomcat6.x 的服务器上。鉴于大多数开发人员熟悉 Eclipse，所以我们觉得这是最直接的方式来打包示例代码。我们为这些例子提供了 Apache Ant 的脚本以及 Apache Maven 的 modules。不管你熟悉开发环境，我们希望你能够在阅读本书的时候能够运行示例代码。

另外，在阅读过程中，你可能会愿意去下载 Spring 3 和 Spring Security 3 的源码版本。如果你有不明白的地方或想获取更多的信息，他们的 JavaDoc 和源码是最好的参考资料，他们提供的示例也能够提供额外的帮助并消除你的疑惑。

查看审计结果

让我们回到 e-mail 并看一下审计的进展。哦，结果貌似并不好啊：

To: Star Developer <stardev@jbcppets.com>

From: Super Visor <theboss@jbcppets.com>

Subject: FW: Security Audit Results

Star,

看一下审计结果并制定一个计划来解决这些问题。

Super Visor

审计结果：

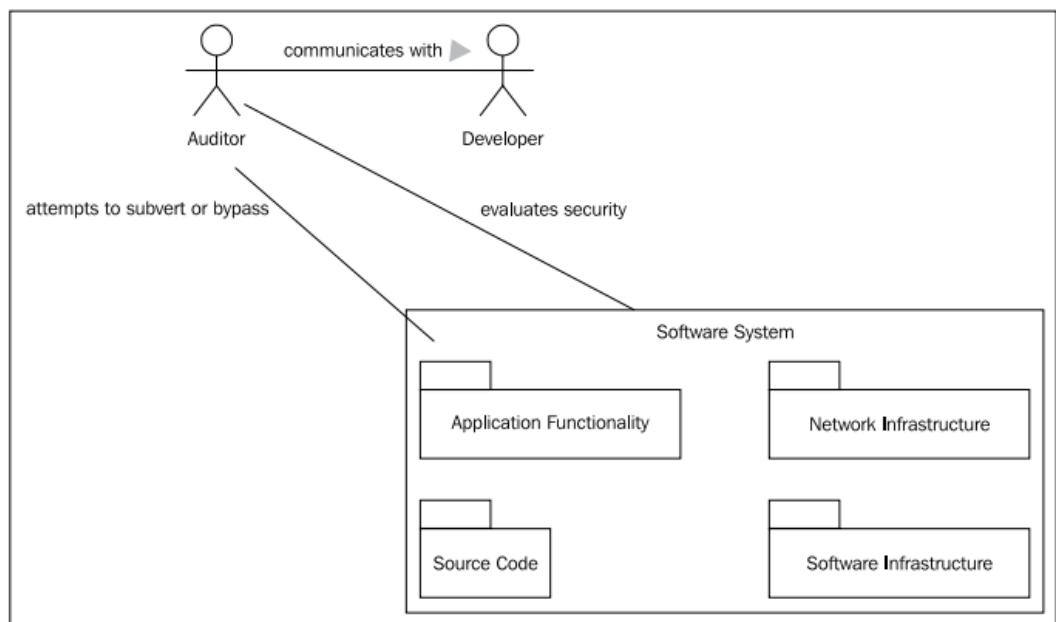
本应用存在如下的不安全隐患：

- 缺少 URL 保护和统一的认证造成的权限扩散；
- 授权不合理甚至缺失；
- 数据库认证信息不安全且很容易获取；
- 个人的识别信息和敏感数据很容易获取或没有加密；
- 不安全的传输层保护，没有使用 SSL 加密；
- 危险等级：高

我们希望本应用在解决这些问题前能够下线。

哦，这些结果看起来对我们公司很不利，我们最好尽快将这些问题解决。

公司（或他们的合作伙伴、顾客）会雇佣第三方的安全专家来审计他们软件的安全性。审计过程会联合使用破解，代码检查以及与开发人员和架构师的正式或非正式交流。



安全审计的通常目标是保证基本的安全开发措施被遵守以实现客户数据和系统功能的完整性和安全性。依靠软件业所追求的工业化目标，审计人员可能会使用工业化的标准或特定的方式来进行这些测试。

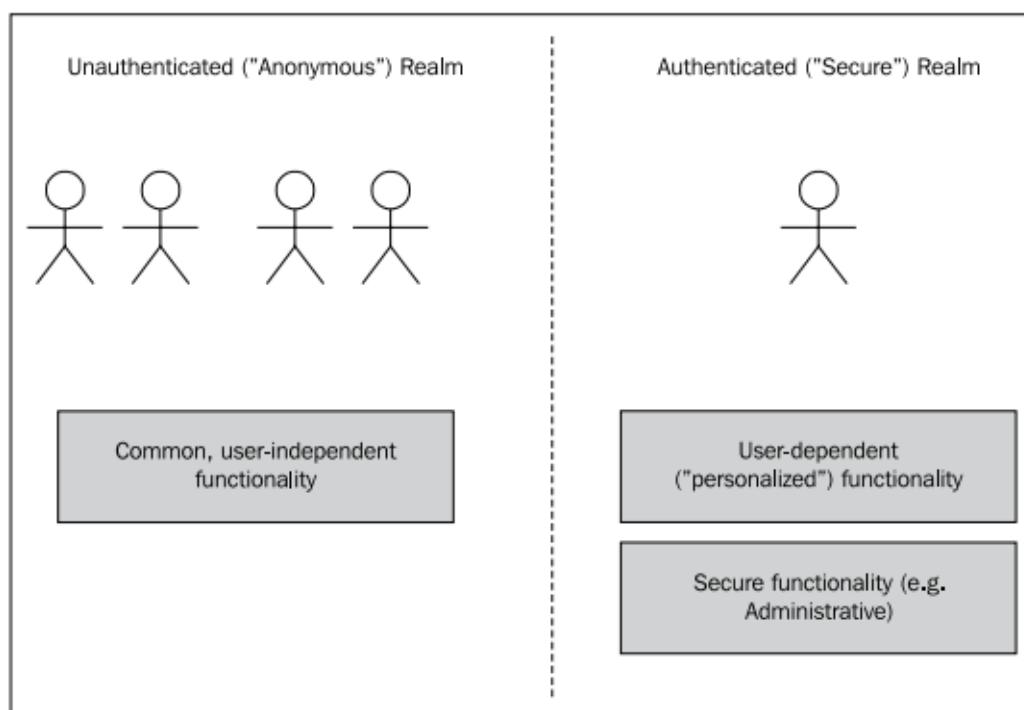
收到这样的安全审计结果可能是一件令人很吃惊的事情，但是，如果你按照被要求去做，这将会是一个绝佳的机会来学习和提升软件质量，同时，这将会引领你去实现一些常规的策略来保证软件的安全性。

让我们来看一下审计人员发现的问题并制定一个计划去解决他们。

认证

缺少 URL 保护和统一的认证造成的权限扩散

认证是在开发安全应用中，你必须记住的两个关键词之一（另一个是授权）。认证识别系统中的某一个用户，并将其与一个可信任的（即安全的）实体关联。一般来讲，软件系统会被分为两个层次的访问范围，如未认证通过的（或匿名的）和认证通过的，如下图所示：



匿名可访问的应用功能是用户无关的（如一个在线商店的产品列表）。

匿名区域不会：

- 要求用户登录系统；
- 展示敏感信息如人名、地址、信用卡号、订单等；
- 提供管理系统全局的状态或数据。

未认证的系统区域可供任何人使用，即使该用户没有被系统所识别。但是，有可能对于已认证的用户会看到更多的信息（如随处可见的欢迎{名字}文本）。Spring Security 的标签库可以完全支持对登录用户进行有区别的显示，这部分内容将在第五章：精确的访问控制中介绍。

我们将在第二章中解决这个发现的问题并使用 Spring Security 的自动配置功能实现一个基于表单的认证。更复杂的表单认证（通常用来在与企业系统集成或外部的认证存储）将在本书的第二部分进行讲解，这部分从第八章：对 OpenID 开放

开始。

授权

授权不合理甚至缺失

授权是两个重要安全概念中的第二个，它对实现和理解应用安全很重要。授权保证授权过的用户能够对功能和数据进行恰当的访问。构建应用的安全模块的主要任务是拆分应用的功能和数据并将权限、功能和数据、用户结合起来，以实现对这些内容的访问能够被很好的控制。在审计中，我们的应用在这一点的失败表明应用的功能没有按照用户角色进行限制。试想一下，你正在运行一个在线购物系统，查看、取消或修改订单以及用户的信息对所有的用户可见！（这将是多么恐怖的事情）

我们将会在第二章通过使用 **Spring Security** 的授权模块解决基本的授权问题，接着会有关于授权更高级的知识介绍，其中在第五章介绍 web 层，在第六章：高级配置和扩展中介绍业务层。

数据库认证安全

数据库认证信息不安全且很容易获取

通过检查应用的源码和配置文件，审计人员指出用户的密码在配置文件中以明文的显示存储，这会导致恶意的用户很容易通过访问到服务器进而访问应用。

因为应用中包含了个人和财务的信息，一个恶意的用户如果能够非法访问任何数据都会导致公司信息的泄露。保护能够进入系统的凭证信息对我们来时是最重要的事情，最重要的第一步是保证在安全上一个点的失败不会连累整个系统。

我们将会在第四章：凭证安全存储中检查 **Spring Security** 的数据访问层以实现凭证存储的安全配置，那里将会用到 JDBC 连接。在第四章中，我们同时也会学习一些内置的技术以增强保存在数据库中密码的安全。

敏感信息

个人的识别信息和敏感数据很容易获取或没有加密

审计人员指出一些重要和敏感的数据，包括信用卡号，在系统中的任何地方都没有加密或混淆。这除了是缺乏数据设计的一种典型表现外，对信用卡号缺少保护也是违反 PCI 标准的。

幸运的是，基于 **Spring Security** 的注解式 AOP 支持，我们可以使用一些简单的设计模式和工具来安全的保护这些信息。我们将会在第五章中阐述在数据访问层保护这种数据的一些技术。

数据传输层保护

不安全的传输层保护，没有使用 SSL 加密

在现实世界中，很难想象一个在线的商业网站不使用 SSL 保护；不幸的是，我们的 JBCP Pet

正是这样。SSL 保护能够保证在客户端浏览器和 web 应用服务器之间的通信是安全的，可以防护多种的信息篡改和窥探。

在第五章中，我们将会介绍如何在应用的安全架构中使用传输层安全配置，包括规划应用的那些部分要使用强制的 SSL 加密。

使用 Spring Security 解决安全问题

Spring Security 提供了丰富的资源，许多安全的通用问题都可以用非常简单的声明或配置来解决。在接下来的章节中，我们将会通过源代码和应用配置的不断改进来解决安全审计人员提出的所有问题（甚至更多），同时会对我们的应用的安全性树立信心。

使用 Spring Security 3，我们将会通过以下的变化来提高应用的安全性：

- 细分系统的用户类别；
- 为用户的角色授予不同级别的权限；
- 为不同的用户类别授予不同的角色；
- 为应用全局的资源使用认证规则；
- 为应用所有层的结构使用授权规则；
- 阻止常用的攻击手段以控制或窃取一个用户的 session。

为什么使用 Spring Security？

Spring Security 的存在填补了众多 java 第三方类库的一个空白，正如 Spring 框架第一次出现时那样。一些业界的标注如 JAAS 或 Java EE Security 确实提供了一些方式来解决认证和授权问题，但是 Spring Security 是一个胜出者，因为它以一种简明和合理的方式封装了各个层的安全解决方案。

除此以外，Spring Security 提供了与很多通用企业认证系统的内置集成支持。所以，对开发者来说，它通过很少的努力就能适应绝大多数的场景。

因为没有任何一款主流的框架与之匹敌，所以它被广泛的使用。

小结

在本章中，我们：

- 检查了一个不安全的 web 工程的风险点；
- 查看了示例在线商务应用的基本架构；
- 讨论了使示例工程安全的策略。

在第二章中，我们将会介绍 Spring Security 的整体架构，主要侧重于其如何扩展和配置以支持我们的应用的需要。

第二章 Spring Security 起步

在本章中，我们将要学习 Spring Security 背后的核心理念，包括重要的术语和产品架构。我们将会关注配置 Spring Security 的一些方式以及对应用的作用。

最重要的是为了解决工作中的问题，我们要开始使得 JBCP Pets 的在线商店系统变得安全。我们将会通过分析和理解认证如何保护在线商店的适当区域来解决在第一章：一个不安全应用的剖析中审计人员发现的第一个问题，即缺少 URL 保护和统一的认证造成的权限扩散。

在本章的内容中，我们将会涉及：

- 了解应用中安全的重要概念；
- 使用 Spring Security 的快速配置功能，为 JBCP Pets 在线商店实现基本层次的安全；
- 理解 Spring Security 的全貌；
- 探讨认证和授权的标准配置和选项；
- 在 Spring Security 访问控制中使用 Spring 的表达式语言（Spring Expression Language）

安全的核心概念

由于安全审计结果的启示作用，你研究了 Spring Security 并确定它能够提供一个坚实的基础，以此可以构建一个安全的系统来解决在安全审计 JBCP Pet 在线商店中发现的问题，而那个系统是基于 Spring Web MVC 开发的。

为了 Spring Security 的使用更高效，在开始评估和提高我们应用的安全状况之前，先了解一些关键的概念和术语是很重要的。

认证

正如我们在第一章所讨论的那样，认证是鉴别我们应用中的用户是他们所声明的那个人。你可能在在线或线下的日常生活中，遇到不同场景的认证：

- 凭据为基础的认证：当你登录 e-mail 账号时，你可能提供你的用户名和密码。E-mail 的提供商将你的用户名与数据中的记录进行匹配，并验证你提供的密码与对应的记录是不是匹配。这些凭证（用户名和密码，译者注）就是 e-mail 系统用来鉴别你是一个合法用户的。首先，我们将首先使用这种类型的认证来保护我们 JBCP Pet 在线商店的敏感区域。技术上来说，e-mail 系统能够检查凭证信息不一定非要使用数据库而是各种方式，如一个企业级的目录服务器如 Microsoft Active Directory。一些这种类型的集成方式将在本书的第二部分讲解。
- 两要素认证：当你想从自动柜员机取钱的时候，你在被允许取钱和做其他业务前，你必须先插卡并输入你的密码。这种方式的认证与用户名和密码的认证方式很类似，与之不同的是用户名信息被编码到卡的磁条上了。联合使用物理磁卡和用户输入密码能是银行确认你可能有使用这个账号的权限。联合使用密码和物理设备（你的 ATM 卡）是一种普遍存在的两要素认证形式。专业来看，在安全领域，这种类型的设备在安全性要求高的系统中很常见，尤其是处理财务或个人识别信息时。硬件设备如 RSA 的 SecurId 联合使用了基于时间的硬件和服务端的认证软件，使得这样的环境极难被破坏。
- 硬件认证：早上当你启动汽车时，你插入钥匙并打火。尽管和其他的两个例子很类似，但是你的钥匙和打火装置的匹配是一种硬件认证的方式。

其实会有很多种的认证方式来解决硬件和软件的安全问题，它们各自也有其优缺点。我们将会在本书的后面章节中介绍它们中的一些，因为它们适用于 Spring Security。事实上，本书的后半部分基本上都是原来介绍很多通用的认证方式用 Spring Security 的实现。

Spring Security 扩展了 java 标准概念中的已认证安全实体（对应单词 principal）（`java.security.Principal`），它被用来唯一标识一个认证过的实体。尽管一个典型的安全实体通常一对一的指向了系统中的一个用户，但它也可能对应系统的各种客户端，如 web service 的客户端、自动运行的 feed 聚合器（automated batch feed）等等。在大多数场景下，在你使用 Spring Security 的过程中，一个安全实体（Principal）只是简单地代表一个用户（user），所以我当我们说一个安全实体的时候，你可以将其等同于说用户。

授权

授权通常涉及到两个不同的方面，他们共同描述对安全系统的可访问性。

第一个是已经认证的安全实体与一个或多个权限（authorities）的匹配关系（通常称为角色）。例如，一个非正式的用户访问你的网站将被视为只有访问的权限而一个网站的管理员将会被分配管理的权限。

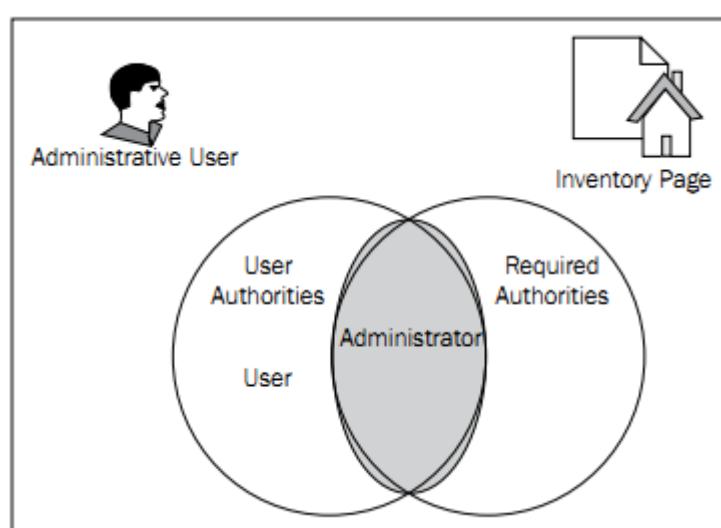
第二个是分配权限检查给系统中要进行安全保护的资源。通常这将会在系统的开发过程中进行，有可能会通过代码进行明确的声明也可能通过参数进行设置。例如，在我们应用中管理宠物商店详细目录的界面只能对具有管理权限的用户开放。

【要进行安全保护的资源可以是系统的任何内容，它们会根据用户的权限进行有选择的可访问控制。**web** 应用中的受保护资源可以是单个的页面、网站的一个完整部分或者一部分界面。相反的，受保护的业务资源可能会是业务对象的一个方法调用或者单个的业务对象。】

你可能想象的出对一个安全实体的权限检查过程，查找它的用户账号并确定它是不是真的为一个管理员。如果权限检查确定这个试图访问受保护区区域的安全实体实际上是管理员，那么这个请求将会成功，否则，这个安全实体的请求将会因为它缺少足够的权限而被拒绝。

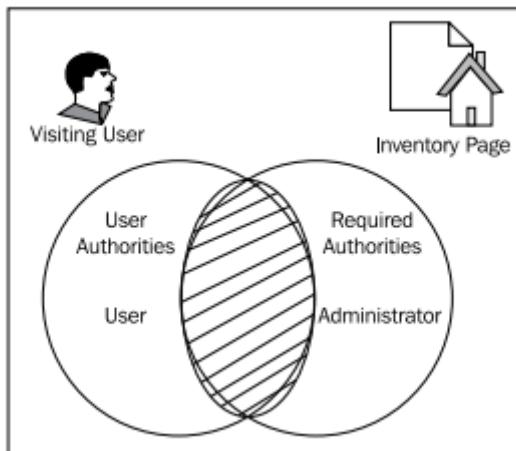
我们更近距离的看一个特定的受保护资源——产品目录的编辑界面。目录的编辑界面需要管理员才能访问（毕竟，我们不希望普通的用户能够调整我们的目录层次），因此当一个安全实体访问它的时候会要求特定等级的权限。

当我们思考一个网站的管理员试图访问受保护的资源时，权限控制决定是如何做出的时候，我们猜想对受保护资源的权限的检查过程可以用集合理论很简明的进行表述。我们将会使用维恩图来展现对管理用户的这个决策过程：



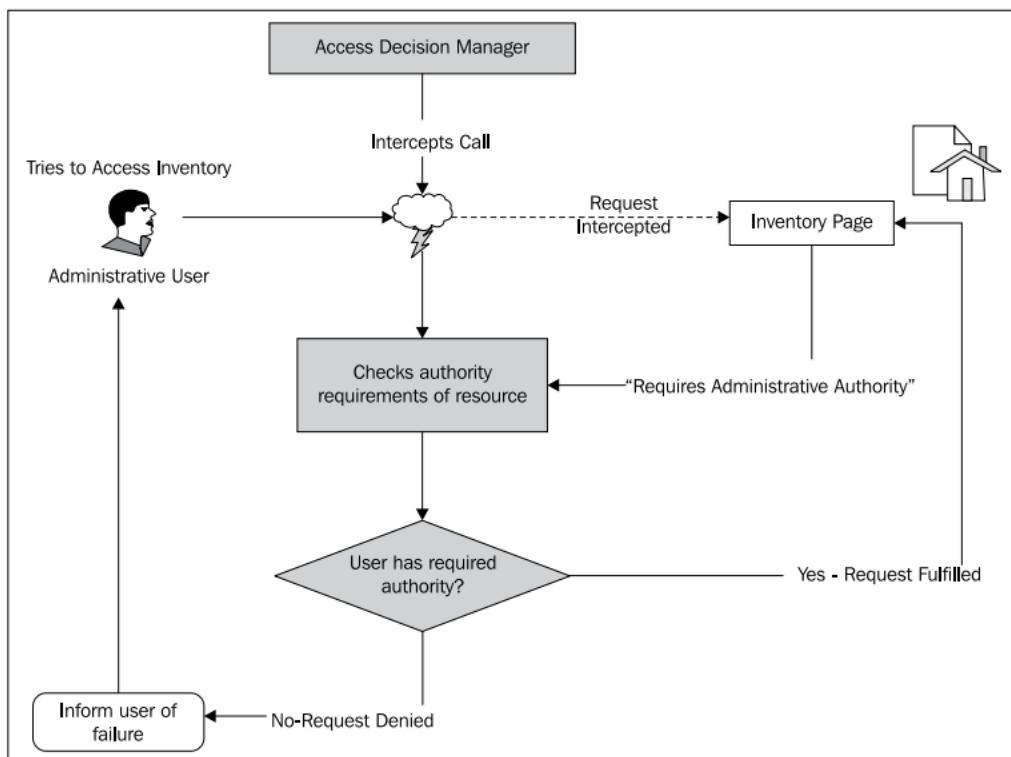
对这个页面来说，在用户权限（普通用户和管理员）和需要权限（管理员）之间有一个交集，所以在交集中的用户将能够进行访问。

可以与没有授权的访问者进行对比：



权限集合没有交集，没有公共的元素。所以，用户将会被拒绝访问这个界面。至此，我们已经介绍了对资源授权的简单原理。

实际上，会有真正的代码来决定用户是允许还是被拒绝访问受保护的资源。下面的图片在整体上描述了这个过程，正如 Spring Security 所使用的那样：



我们可以看到，有一个名为访问决策管理器（access decision manager）的组件来负责决定一个安全实体是不是有适当的访问权限，判断基于安全实体具备的权限与被请求资源所要求资源的匹配情况。

安全访问控制器对访问是否被允许的判断过程可能会很简单，就像查看安全实体所拥有的权限集合与被访问资源所要求的资源集合是不是有交集。

让我们在 JBCP Pets 应用中简单使用 Spring Security，并将会更详细的阐述认证和授权。

三步之内使我们的应用变得安全

尽管 Spring Security 的配置可能会很难，但是它的作者是相当为我们着想的，因为他们为我们提供了一种简单的机制来使用它很多的功能并可以此作为起点。以这个为起点，额外的配置能够实现应用的分层次详细的安全控制。

我们将从我们不安全的在线商店开始，并且使用三步操作将它变成一个拥有基本用户名和密码安全认证的站点。这个认证仅仅是为了阐述使用 Spring Security 使我们应用变得安全的步骤，所以你可能会发现这样的方式会有明显不足，这将会引领我们在以后的配置中不断进行改良。

实现 Spring Security 的 XML 配置文件

起点配置的第一步是创建一个 XML 的配置文件，用来描述所有需要的 Spring Security 组件，这些组件将会控制标准的 web 请求。

在 WEB-INF 目录下建立一个名为 dogstore-security.xml 的 XML 文件。文件的内容如下所示：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/security"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:beans="http://www.springframework.org/schema/beans"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/security
        http://www.springframework.org/schema/security/
            spring-security-3.0.xsd">
    <http auto-config="true">
        <intercept-url pattern="/*" access="ROLE_USER"/>
    </http>
    <authentication-manager alias="authenticationManager">
        <authentication-provider>
            <user-service>
                <user authorities="ROLE_USER" name="guest" password="guest"/>
            </user-service>
        </authentication-provider>
    </authentication-manager>
</beans:beans>
```

这是我们的应用中获得最小安全配置的唯一一个 Spring Security 配置文件。这个配置文件的格式使用了 Spring Security 特定的 XML 语法，一般称之为 security 命名空间。它在 XML 的命名空间中声明(<http://www.springframework.org/schema/security>)并与 XML 配置元素关联。我们将在在第六章：高级配置与扩展中讨论一种替代的配置方式，即使用传统的 Spring Bean 设置方式。

【讨厌 Spring XML 配置的用户可能会失望了，因为 Spring Security 没有像 Spring 那样提

供可替代的注解机制。仔细想一下也是可以理解的，因为 Spring Security 关注的是整个系统而不是单个对象或类。但未来，我们可能能够在 Spring MVC 的控制器上使用安全注解，而不是在一个配置文件中指明 URL 模式！】

尽管在 Spring Security 中注解并不普遍，但正如你所预料的那样，对类或方法进行的配置是可以通过注解来完成的。我们将在第五章：精确的访问控制中介绍。

添加 Spring DelegatingFilterProxy 到 web.xml 文件

Spring Security 对我们应用的影响是通过一系列的 ServletRequest 过滤器实现的（我们将会在介绍 Spring Security 架构的时候进行阐述）。可以把这些过滤器想成位于每个到我们应用的请求周围的具有安全功能的三明治。（这个比喻够新鲜，不过 Spring Security 的核心确实就是一系列介于真正请求之间的过滤器，译者注）。

Spring Security 使用了 o.s.web.filter.DelegatingFilterProxy 这个 servlet 过滤器来包装所有的应用请求，从而保证它们是安全的。

【DelegatingFilterProxy 实际上是 Spring 框架提供的，而不是安全特有的。这个过滤器一般在 Spring 构建的 web 应用工程中使用，并将依赖于 servlet 过滤器的 Spring Bean 与 Servle 过滤器的生命周期结合起来。】

通过在 web.xml 部署描述文件中添加如下的代码，就可以配置这样一个过滤器。这段代码位于 Spring MVC 的<servlet-mapping>之后：

```
<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>
```

我们所做的是使用一个 ServletRequest 过滤器并将它配置成处理匹配给定 URL 模式（/*）的请求。因为我们配置的这个通配符模式匹配所有的 URL，所以这个过滤器将会应用于每个请求。

如果你有心的话，可能会发现这与我们的 Spring Security 配置文件即 dogstore-security.xml 没有任何的关系。为了实现这个，我们需要添加一个 XML 配置文件的应用到 web 应用的部署描述文件 web.xml 中。

添加 Spring Security XML 配置文件的应用到 web.xml

取决于你如何配置你的 Spring web 应用，不知你是否已经在 web.xml 中有了对 XML 配置文件的明确引用。Spring web 的 ContextLoaderListener 的默认行为是寻找与你的 Spring web servlet 同名的 XML 配置文件。让我们看一下如何添加这个新的 Spring Security XML 配置文件

到已经存在的 JBCP Pet 商店站点中。

首先，我们需要看一下这个应用是否使用了 Spring MVC 的自动查找 XML 配置文件的功能。我们看一下在 web.xml 中 servlet 的名字，以<servlet-name>原始进行标识：

```
<servlet>
    <servlet-name>dogstore</servlet-name>
    <servlet-class>
        org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
```

Servlet 的名字是(<servlet-name>)是 dogstore，所以 Spring 的约定胜于配置(Convention over Configuration, CoC)将会在 WEB-INF 目录下寻找名为 dogstore-servelt.xml 的配置文件。我们没有覆盖这种默认行为，你能在 WEB-INF 目录下找到这个文件，它包含了一些 Spring MVC 相关的配置。

【很多 Spring Web Flow 和 Spring MVC 的用户并不明白这些 CoC 规则是如何生效的以及 Spring 的代码中在何处声明了这些规则。o.s.web.context.support.XmlWebApplicationContext 和它的父类是了解这些的一个很好的起点。JavaDoc 在讲解基于 Spring MVC 框架的 web 工程的一些参数配置方面也做得不错。】

也可以声明额外的 Spring ApplicationContext 配置文件，它将会先于 Spring MVC servle 关联的配置文件加载。这通过 Spring 的 o.s.web.context.ContextLoaderListener 创建一个独立于 Spring MVC ApplicationContext 的另一个 ApplicationContext 来实现。这是通常的非 Spring MVC beans 配置的方式，也为 Spring Security 相关的配置提供了一个好地方。

在 web 应用的部署描述文件中，用来配置 ContextLoaderListener 的 XML 文件地址是在 <context-param> 元素中给出的：

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        /WEB-INF/dogstore-base.xml
    </param-value>
</context-param>
```

dogstore-base.xml 文件中包含了一些标准的 Spring bean 的配置，如数据源、服务层 bean 等。现在，我们能够添加一个包含 Spring Security 的 XML 配置文件了，下面是更新后的 <context-param> 元素：

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        /WEB-INF/dogstore-security.xml
        /WEB-INF/dogstore-base.xml
    </param-value>
</context-param>
```

在我们添加完新的 Spring Security 配置文件到 web 部署文件并重启 web 工程后，尝试访

问应用的首页地址：<http://localhost:8080/JBCPPets/home.do>，你将会看到如下的界面：



漂亮！我们已经使用 **Spring Security** 在三步之内实现了一个简单的安全层。在这里，你可以使用 `guest` 作为用户名和密码进行登录。接着你将能够看到 JBCP Pets 应用的一个简单首页。

为了简便起见，本章的源码只包含了全部 JBCP Pets 整个应用的很小一部分（只有一个页面）。我们这么做是为了更简洁，同时也能够让构建和部署应用时不必考虑我们将在后续章节中涉及到的附加功能。这也提供了一个很好的起点让你快速的体验参数的配置并重新部署以查看它们是否生效。

注意这些不足之处

到此为止，思考一下我们所做的事情。你可能已经意识到在产品化之前应用存在很多很明显的问题，这需要很多后续的工作和对 **Spring Security** 产品的了解。尝试列举一下在将这个实现安全功能的站点上线前还需要什么样的完善。

实现 **Spring Security** 起点级别的配置是相当迅速的，它已经提供了一个登录界面，用户名密码认证以及自动拦截我们在线商店的 URL。但是在自动配置给我们提供的功能与我们的最终目标之间有很大的差距：

- 我们将用户名、密码以及角色信息硬编码到 XML 配置文件中。你是否还记得我们添加的这部分 XML 内容：

```
<authentication-manager alias="authenticationManager">
    <authentication-provider>
        <user-service>
            <user authorities="ROLE_USER" name="guest" password="guest"/>
        </user-service>
    </authentication-provider>
</authentication-manager>
```

你可以看到用户名和密码在这个文件中。你不可能愿意为每一个系统的用户都在这个 XML 文件中添加一个声明。要解决这个问题，你需要使用一个基于数据库的认证提供者（authentication provider）来替代它（我们将第四章：凭证安全存储完成它）。

- 我们对所有的页面都进行了安全控制，包括一些潜在客户可能想匿名访问的界面。我们需要完善系统的角色以适应匿名的、认证过的以及管理权限的用户（这也将再第四章中讨论）。
- 登录界面非常应用，但是它太基础了而且与我们 JBCP 商店风格一点也不一致。我

们需要添加一个登录的 `form` 界面，并与我们应用的外观和风格一致（我们将在下一章解决这个问题）。

常见问题

很多用户在初次使用 `Spring Security` 时会遇到一些问题。我们列出了几个常见的问题和建议。我们希望你能够一直跟随着本书的讲解，运行我们示例代码。

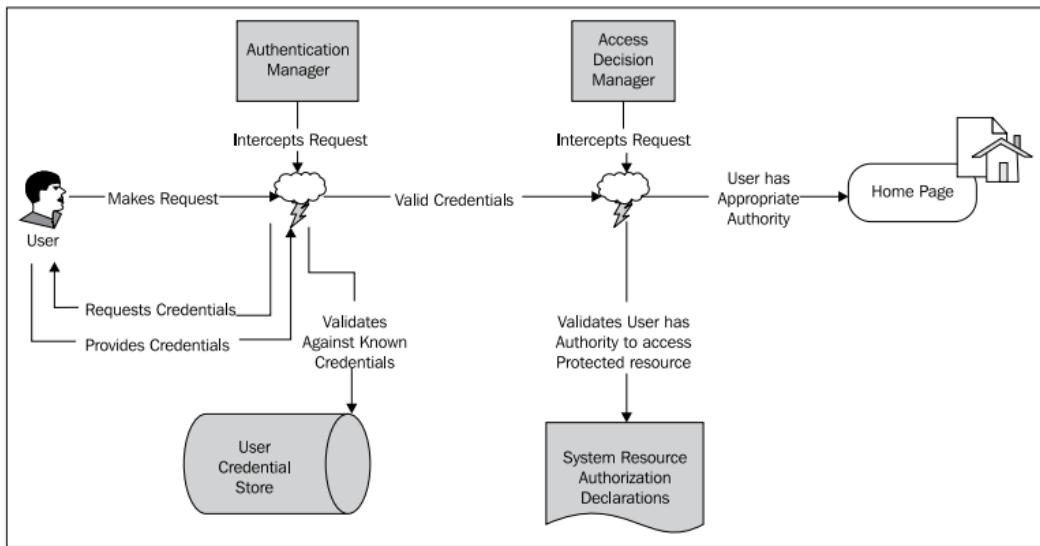
- 确保你的应用在添加 `Spring Security` 之前是可以编译和部署的。必要的时候看一些你所使用的 `servlet` 容器的入门级例子和文档。
- 通常使用一个 IDE 如 `Eclipse` 会极大地简化你使用的 `servlet` 容器。不仅能够保证部署准确无误，它所提供的控制台日志也很易读可用来检查错误。你还可以在关键的位置添加断点，运行的时候会触发从而简化分析错误的过程。
- 如果你的 `XML` 配置文件不正确，你会得到这样的提示信息（或类似的）：`org.xml.sax.SAXParseException: cvc-elt.1: Cannot find the declaration of element 'beans'`。为实现 `Spring Security` 的正确配置需要各种各样的 `XML` 命名空间引用，这可能会使很多用户感到迷惑。重新检查一下这个例子，仔细查看一下 `schame` 声明的部分，并使用 `XML` 校验器来保证你没有不合法的 `XML`。
- 确保你所使用的 `Spring` 和 `Spring Security` 版本匹配，并确保没有你的应用中不存在没用的 `Spring jar` 包。

安全的复杂之处：安全 web 请求的架构

借助于 `Spring Security` 的强大基础配置功能以及内置的认证功能，我们在前面讲述的三步配置是很快就能完成的；它们的使用是通过添加 `auto-config` 属性和 `http` 元素实现的。

但不幸的是，应用实现的考量、架构的限制以及基础设施集成的要求可能使你的 `Spring Security` 实现远较这个简单的配置所提供的复杂。很多用户一使用比基本配置复杂的功能就会遇到麻烦，那是因为他们不了解这个产品的架构以及各个元素是如何协同工作以实现一个整体的。

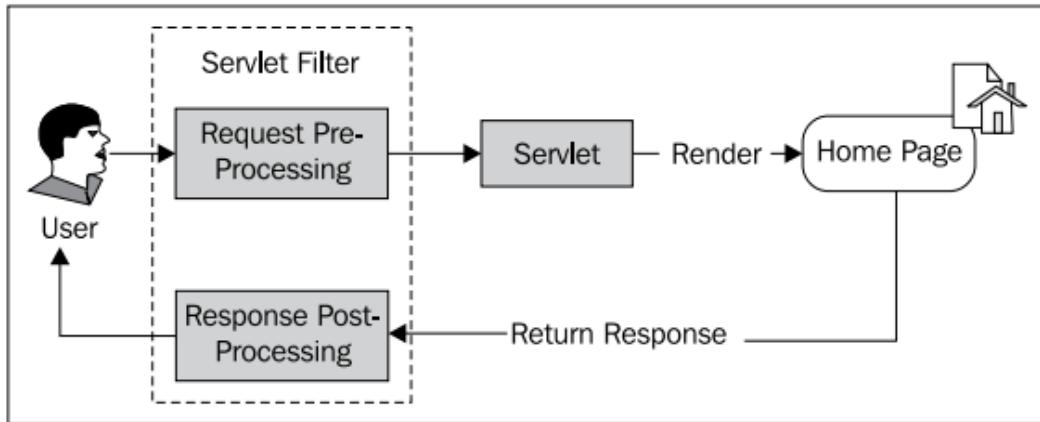
理解 `web` 请求的整体流程以及它们是如何穿越实现功能的拦截器链，对我们成功了解 `Spring Security` 的高级话题至关重要。记住认证和授权的基本概念，因为它们贯穿我们要保护的系统架构的始终。



请求是怎样被处理的？

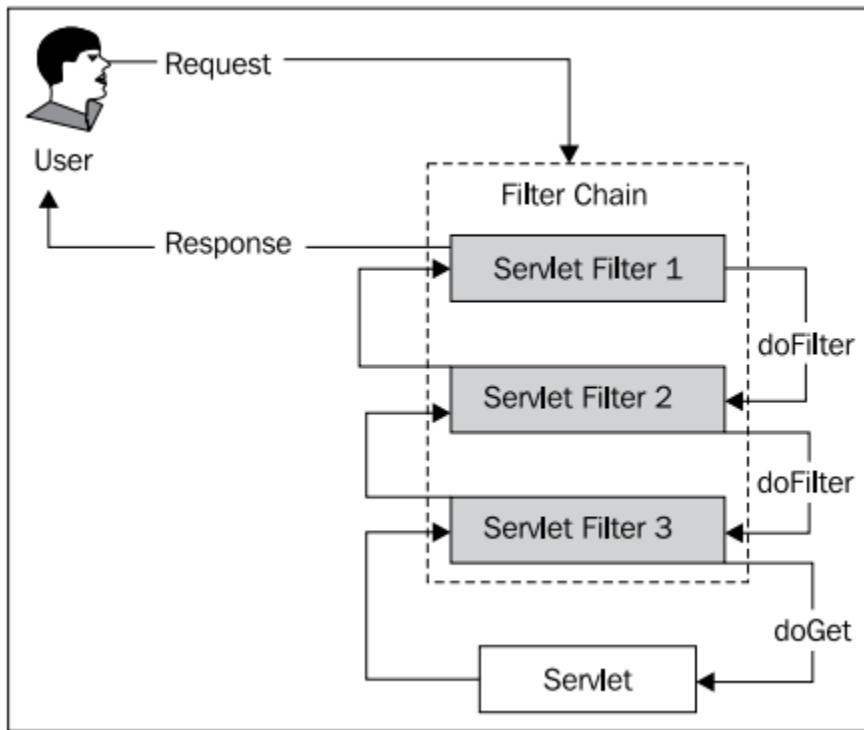
Spring Security 的架构在很大程度上依赖代理（delegates）和 servlet 过滤器，来实现环绕在 web 应用请求前后的功能层。

Servlet 过滤器（Servlet Filter，实现 javax.servlet.Filter 接口的类）被用来拦截用户请求来进行请求之前或之后的处理，或者干脆重定向这个请求，这取决于 servlet 过滤器的功能。在 JBCP Pets 在线商店中，最终的目标 servlet 是 Spring MVC 分发 servlet，但是在理论上它可能是任何一个 web servlet。下面的图描述了一个 servlet 过滤器是如何封装一个用户请求的：



Spring Security 在 XML 配置文件中的自动配置属性，建立了十个 servlet 过滤器，它们通过使用 Java EE 的 servlet 过滤器链按顺序组合起来。Filter chain 是 Java EE Servlet API 的一个概念，通过接口 javax.servlet.FilterChain 进行定义，它允许在 web 应用中的一系列的 servlet 过滤器能够应用于任何给定的请求。

与生活中金属制定的链类似，每一个 servlet 过滤器代表链上的一环，它会进行方法的调用以处理用户的请求。请求穿过整个过滤器链，按顺序调用每个过滤器。



正如你能从链这个词汇中推断出的那样，`servlet` 请求按照一定的顺序从一个过滤器到下一个穿过整个过滤器链，最终到达目标 `servlet`。与之相对的是，当 `servlet` 处理完请求并返回一个 `response` 时，过滤器链按照相反的顺序再次穿过所有的过滤器。

`Spring Security` 使用了过滤器链的概念并实现了自己抽象，提供了 `VirtualFilterChain`，它可以根据 `Spring Security XML` 配置文件中设置的 URL 模式动态的创建过滤器链（可以将它与标准的 Java EE 过滤器链进行对比，后者需要在 web 应用的部署描述文件中进行设置）。

【Servlet】 过滤器除了能够如它的名字所描述的那样进行过滤功能（或阻止请求）以外，还可以用于很多其他的目的。实际上，很多的 `servlet` 过滤器的功能类似于在 web 运行的环境中对请求进行 AOP 式的代理拦截，因为它们可以允许一些功能在任何发往 `servlet` 容器的请求处理之前或之后执行。过滤器能实现的多功能在 `Spring Security` 中页得到了体现，因为很多过滤器实际上并不直接影响用户的请求。】

自动配置的选项为你建立了十个 `Spring Security` 的过滤器。理解这些过滤器的默认行为以及它们在哪里以及如何配置的，对使用 `Spring Security` 的高级功能至关重要。

这些过滤器以及它们使用的顺序，在下面的表格中进行了描述。大多数这些过滤器在我们完善 JBCP Pets 在线商店的过程中都会被再次提到，所以如果你现在不明白它们的确切功能也不必担心。

过滤器名称	描述
<code>o.s.s.web.context.SecurityContextPersistenceFilter</code>	负责从 <code>SecurityContextRepository</code> 获取或存储 <code>SecurityContext</code> 。 <code>SecurityContext</code> 代表了用户安全和认证过的 session。
<code>o.s.s.web.authentication.logout.LogoutFilter</code>	监控一个实际为退出功能的 URL（默认为 <code>/j_spring_security_logout</code> ），并且在匹配的时候完成用户的退出功能。

o.s.s.web.authentication.UsernamePasswordAuthenticationFilter	监控一个使用用户名和密码基于 form 认证的 URL（默认为/j_spring_security_check），并在 URL 匹配的情况下尝试认证该用户。
o.s.s.web.authentication.ui.DefaultLoginPageGeneratingFilter	监控一个要进行基于 form 或 OpenID 认证的 URL（默认为/spring_security_login），并生成展现登录 form 的 HTML
o.s.s.web.authentication.www.BasicAuthenticationFilter	监控 HTTP 基础认证的头信息并进行处理
o.s.s.web.savedrequest.RequestCacheAwareFilter	用于用户登录成功后，重新恢复因为登录被打断的请求。
o.s.s.web.servletapi.SecurityContextHolderAwareRequestFilter	用一个扩展了 HttpServletRequestWrapper 的子类（o.s.s.web.servletapi.SecurityContextHolderAwareRequestWrapper）包装 HttpServletRequest。它为请求处理器提供了额外的上下文信息。
o.s.s.web.authentication.AnonymousAuthenticationFilter	如果用户到这一步还没有经过认证，将会为这个请求关联一个认证的 token，标识此用户是匿名的。
o.s.s.web.session.SessionManagementFilter	根据认证的安全实体信息跟踪 session，保证所有关联一个安全实体的 session 都能被跟踪到。
o.s.s.web.access.ExceptionTranslationFilter	解决在处理一个请求时产生的指定异常
o.s.s.web.access.intercept.FilterSecurityInterceptor	简化授权和访问控制决定，委托一个 AccessDecisionManager 完成授权的判断

Spring Security 拥有总共大约 25 个过滤器，它们能够根据你的需要进行适当的应用以改变用户请求的行为。当然，如果需要的话，你也可以添加你自己实现了 javax.servlet.Filter 接口的过滤器。

请记住，如果你在 XML 配置文件中使用了 auto-config 属性，以上表格中列出的过滤器自动添加的。通过使用一些额外的配置指令，以上列表中的过滤器能够精确的控制是否被包含，在后续的章节将会有详细介绍。

你可能会完全从头做起来配置过滤器链。尽管这会很单调乏味，因为有很多的依赖关系要配置，但是它为配置和应用场景的匹配方面提供了更高层次的灵活性。我们将在第六章讲述在启动的过程中所依赖的 Spring Bean 的声明。

你可能想知道 DelegatingFilterProxy 是怎样找到 Spring Security 配置的过滤器链的。让我们回忆一下，在 web.xml 文件中，我们需要给 DelegatingFilterProxy 一个过滤器的名字：

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

这个过滤器的名字并不是随意配置的，实际上会根据这个名字把 Spring Security 织入到 DelegatingFilterProxy。除非明确配置，否则 DelegatingFilterProxy 会在 Spring WebApplicationContext 中寻找同名的配置 bean（名字是在 filter-name 中指明的）。更多配置

DelegatingFilterProxy 的细节可以在这个类对应的 Javadoc 中找到。

在 auto-config 场景下，发生了什么事情？

在 Spring Security 3 中，使用 auto-config 会自动提供以下三个认证相关的功能：

- HTTP 基本认证
- Form 登录认证
- 退出

值得注意的是，也可以使用配置元素实现这三个功能，能够实现比使用 auto-config 提供的功能更精确。我们将在随后的章节中看到它们的使用以提供更高级的功能。

【auto-config 和以前不一样了！在 Spring Security3 之前的版本中，auto-config 属性提供了比现在更多的启动项。在 Spring Security2 中通过 auto-config 配置的功能，现在可以使用 security 命名空间样式的配置很容易的实现。请参考第 13 章：迁移至 Spring Security 3 来获取更多从 Spring Security2 迁移到 3 的详细信息。】

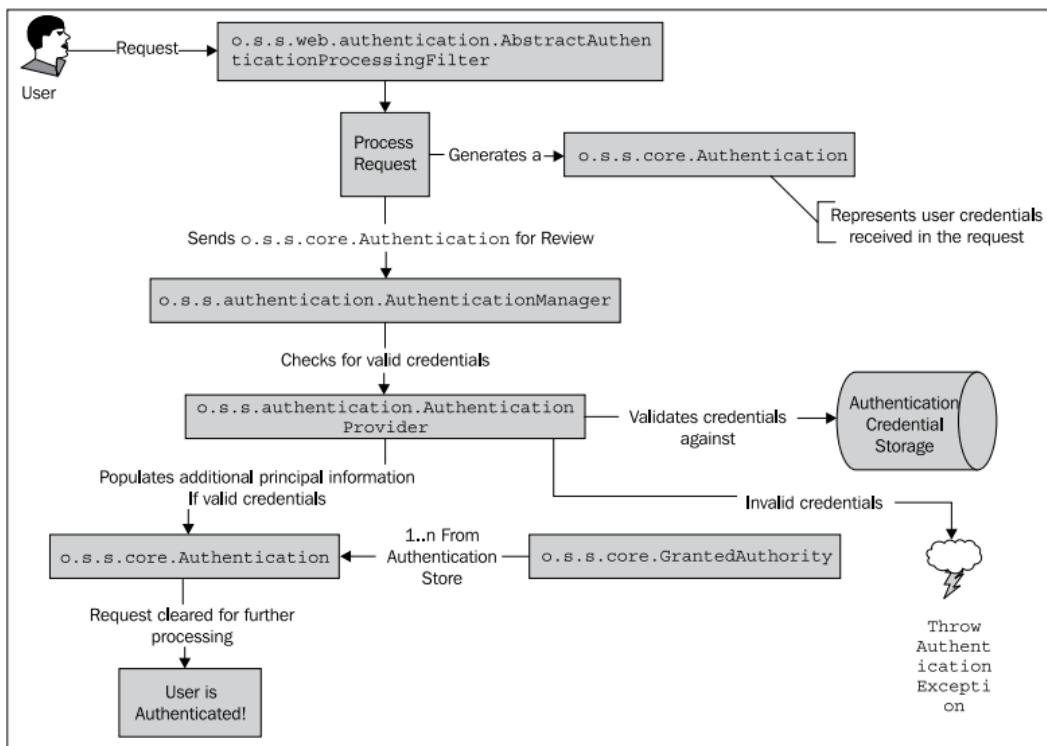
除了以上认证相关的功能，其它过滤器链的配置是通过使用<http>元素来实现的。

用户是怎样认证的？

在我们的安全系统中，当一个用户在我们的登录 form 中提供凭证后，这些凭证信息必须与凭证存储中的数据进行校验以确定下一步的行为。凭证的校验涉及到一系列的逻辑组件，它们封装了认证过程。

我们将会深入讲解我们例子中的用户名和密码登录 form，与之对应的接口和实现都是特定于用户名和密码认证的。但是，请记住，整体的认证是相同的，不管你是使用基于 form 的登录请求，或者使用一个外部的认证提供者如集中认证服务（CAS），抑或用户的凭证信息存在一个数据库或在一个 LDAP 目录中。在本书的第二部分，我们将会看到在基于 form 登录中学到的概念是如何应用到更高级的认证机制中。

涉及到认证功能的重要接口在下边的图标中有一个概览性的描述：



站在一个较高层次上看，你可以看到有三个主要的组件负责这项重要的事情：

接口名	描述/角色
AbstractAuthenticationProcessingFilter	它在基于 web 的认证请求中使用。处理包含认证信息的请求，如认证信息可能是 form POST 提交的、SSO 信息或者其他用户提供的。创建一个部分完整的 Authentication 对象以在链中传递凭证信息。
AuthenticationManager	它用来校验用户的凭证信息，或者会抛出一个特定的异常（校验失败的情况）或者完整填充 Authentication 对象，将会包含了权限信息。
AuthenticationProvider	它为 AuthenticationManager 提供凭证校验。一些 AuthenticationProvider 的实现基于凭证信息的存储，如数据库，来判定凭证信息是否可以被认可。

有两个重要接口的实现是在认证链中被这些参与的类初始化的，它们用来封装一个认证过（或还没有认证过的）的用户的详细信息和权限。

`o.s.s.core.Authentication` 是你以后要经常接触到的接口，因为它存储了用户的详细信息，包括唯一标识（如用户名）、凭证信息（如密码）以及本用户被授予的一个或多个权限（`o.s.s.core.GrantedAuthority`）。开发人员通常会使用 `Authentication` 对象来获取用户的详细信息，或者使用自定义的认证实现以便在 `Authentication` 对象中增加应用依赖的额外信息。

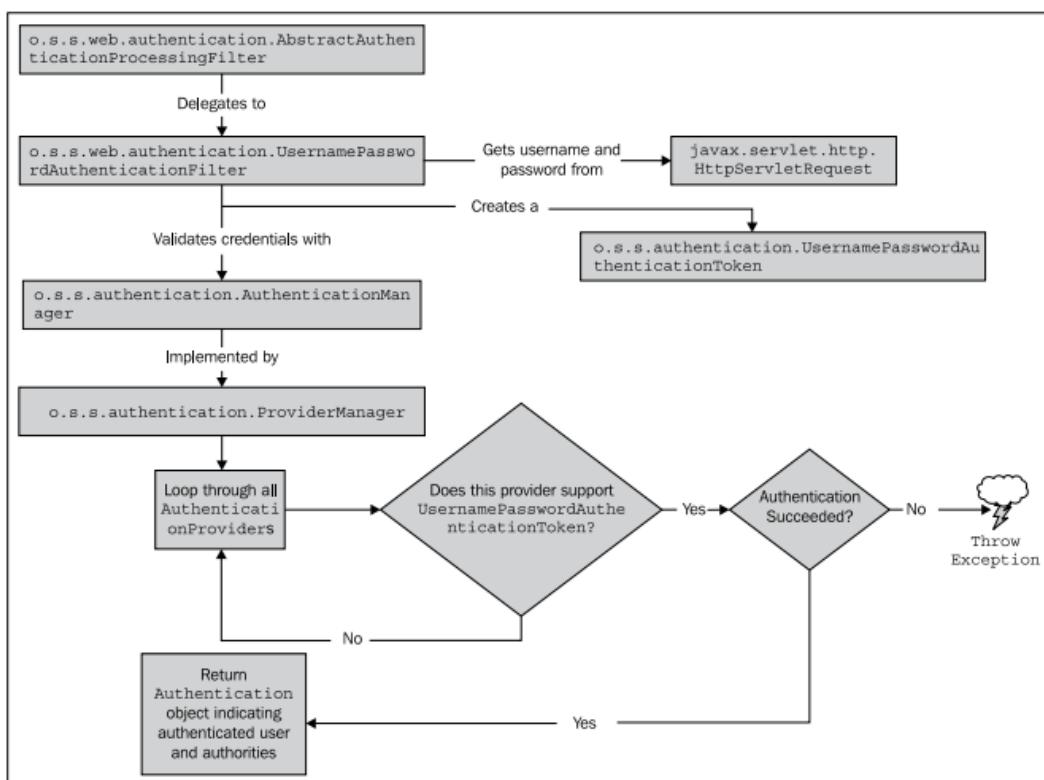
以下列出了 `Authentication` 接口可以实现的方法：

方法签名	描述
<code>Object getPrincipal()</code>	返回安全实体的唯一标识（如，一个用户名）
<code>Object getCredentials()</code>	返回安全实体的凭证信息

List<GrantedAuthority> getAuthorities()	得到安全实体的权限集合，根据认证信息的存储决定的。
Object getDetails()	返回一个跟认证提供者相关的安全实体细节信息

你可能会担心的发现，Authentication 接口有好几个方法的返回值是简单的 java.lang.Object。这可能会导致在编译阶段很难知道调用 Authentication 对象的方法返回值是什么类型的对象。需要注意的一点是 AuthenticationProvider 并不是直接被 AuthenticationManager 接口使用或引用的。但是 Spring Security 只提供了 AuthenticationManager 的一个具体实现类，即 o.s.s.authentication.ProviderManager，它会使用一个或更多以上描述的 AuthenticationProvider 实现类。因为 AuthenticationProvider 的使用非常普遍并且被很好的集成在 ProviderManager 中，所以理解它在最常见的基本配置下是如何工作的就非常重要了。

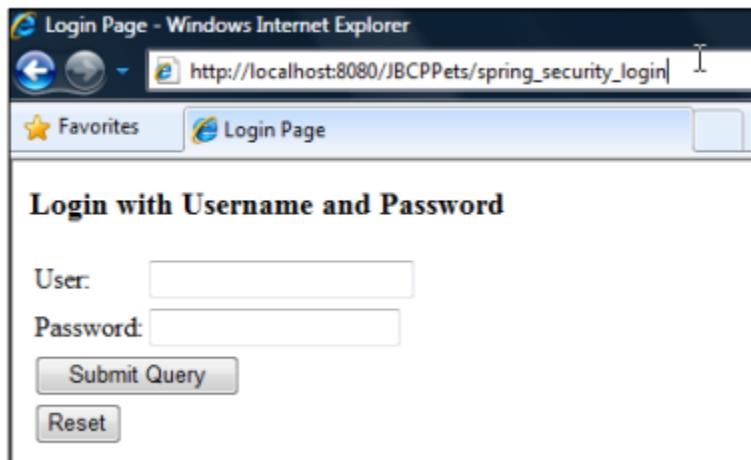
让我们更仔细的看看在基于 web 用户名和密码认证的请求下，这些类的处理过程：



让我们看一下在较高层次示意图中反映出的抽象工作流程，并将其细化到这个基于表单认证的具体实现。你可以看到 `UsernamePasswordAuthenticationFilter` 负责（通过代理从它的抽象父类中）创建 `UsernamePasswordAuthenticationToken` 对象（`Authentication` 接口的一个实现），并部分填充这个对象依赖的信息，这些信息来自 `HttpServletRequest`。但是它是从哪里获取用户名和密码的呢？

spring_security_login 是什么？我们怎么到达这个界面的？

你可能已经发现，当你试图访问我们 JBCP Pets 商店的主页时，你被重定向到 http://localhost:8080/JBCPPets/spring_security_login：



URL 的 `spring_security_login` 部分表明这是一个默认的登录的页面并且是在 `DefaultLoginPageGeneratingFilter` 中命名的。我们可以使用配置属性来修改这个页面的名字从而使得它对于我们应用来说是唯一的。

【建议修改登录页 URL 的默认值。修改后不仅能够对应用或搜索引擎更友好，而且能够隐藏你使用 Spring Security 作为安全实现的事实。通过这种方式来掩盖 Spring Security 能够使得万一 Spring Security 被发现存在安全漏洞时，恶意黑客寻找你应用漏洞的难度。尽管通过这种方式的安全掩盖不会降低你应用的脆弱性，但是它确实能够使得一些传统的黑客工具很难确定你的应用能够承担的住什么类型的攻击。需要注意的是，这里并不是“`spring`”名称在 URL 中出现的唯一地方。我们将在后面的章节详细阐述。】

让我们看一下这个 `form` 的 HTML 源码（忽略布局信息），来看一下 `UsernamePasswordAuthenticationFilter` 期望得到的信息：

```
<form name='f' action='/JBCPPets/j_spring_security_check' method='POST'>
    User:<input type='text' name='j_username' value=''/>
    Password:<input type='password' name='j_password'/'>
    <input name="submit" type="submit"/>
    <input name="reset" type="reset"/>
</form>
```

你可以看到用户名和密码对应的 `form` 文本域有独特的名字 (`j_username` 和 `j_password`)，并且 `form` 的 `action` 地址 `j_spring_security_check` 也并不是我们配置的。它们是怎么来的呢？文本域的名字是 `UsernamePasswordAuthenticationFilter` 规定的，并借鉴了 Java EE Servlet 2.5 的规范（在 SRV.12.5.3 章节中），规范要求登录的 `form` 使用特定的名字并且 `form` 的 `action` 要为特定的 `j_security_check` 值。这样的实际模式目标是允许基于 Java EE servlet-based 的应用能够与 servlet 容器的安全设施以标准的方式连接起来。

因为我们的应用没有使用到 servlet 容器的安全组件，所以可以明确设置 `UsernamePasswordAuthenticationFilter` 以使得文本域有不同的名字。这种特定的配置变化可能会比你想象的复杂。现在，我们将要回顾一下 `UsernamePasswordAuthenticationFilter` 的生命周期，看一下它是如何进入我们配置的（尽管我们将会在第六章再次讲述这个配置）。

`UsernamePasswordAuthenticationFilter` 是通过 `<http>` 配置指令的 `<form-login>` 子元素来进行配置的。正如在本章前面讲述的，我们设置的 `auto-config` 元素将会在你没有明确添加的情况下包含了 `<form-login>` 功能。正如你可能猜测的那样，`j_spring_security_check` 并不对应任何

应用中的物理资源。它只是 `UsernamePasswordAuthenticationFilter` 监视的一个基于 form 登录的 URL。实际上，在 Spring Security 中有好几个这样的特殊的 URL 来实现特定的全局功能。你能在 [附录：参考资料](#) 中找到这些 URL 的一个列表。

用户的凭证信息是在哪里被校验的？

在我们的简单的三步配置文件中，我们使用了一个基于内存的凭证存储实现快速的部署和运行：

```
<authentication-manager alias="authenticationManager">
    <authentication-provider>
        <user-service>
            <user authorities="ROLE_USER" name="guest" password="guest"/>
        </user-service>
    </authentication-provider>
</authentication-manager>
```

我们没有将 `AuthenticationProvider` 与任何具体的实现相关联，在这里我们再次看到了 `security` 命名空间默认为我们做了许多机械的配置工作。但是需要记住的是 `AuthenticationManager` 支持配置一个或多个 `AuthenticationProvider`。`<authentication-provider>` 声明默认谁实例化一个内置的实现，即 `o.s.s.authentication.dao.DaoAuthenticationProvider`。`<authentication-provider>` 声明会自动的将这个 `AuthenticationProvider` 对象织入到配置的 `AuthenticationManager` 中，当然在我们这个场景中 `AuthenticationManager` 是自动配置的。

`DaoAuthenticationProvider` 是 `AuthenticationProvider` 的简单封装实现并委托 `o.s.s.core.userdetails.UserDetailsService` 接口的实现类进行处理。`UserDetailsService` 负责返回 `o.s.s.core.userdetails.UserDetails` 的一个实现类。

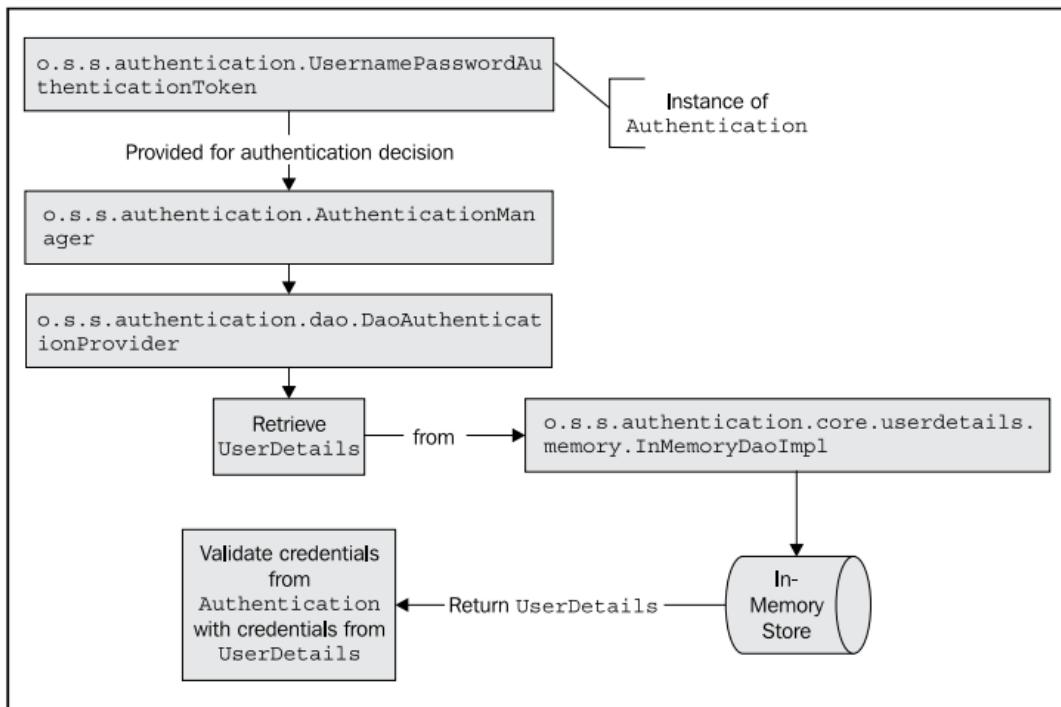
如果你查看 `UserDetails` 的 Javadoc，你会发现它与我们前面讨论的 `Authentication` 接口非常类似。尽管它们在方法名和功能上有些重叠的部分，但是请不要混淆，它们有着截然不同的目的：

接口	目的
<code>Authentication</code>	它存储安全实体的标识、密码以及认证请求的上下文信息。它还包含用户认证后的信息（可能会包含一个 <code>UserDetails</code> 的实例）。通常不会被扩展，除非是为了支持某种特定类型的认证。
<code>UserDetails</code>	为了存储一个安全实体的概况信息，包含名字、e-mail、电话号码等。通常会被扩展以支持业务需求。

我们对 `<user-service>` 子元素的声明将会触发对 `o.s.s.core.userdetails.memory.InMemoryDaoImpl` 的配置，它是 `UserDetailsService` 的一个实现。正如你可能期望的那样，这个实现将在安全 XML 文件中配置的用户信息放在一个内存的数据存储中。这个 service 的实现支持其它属性的设置，从而实现账户的禁用和锁定。

让我们更直观的看一下 `DaoAuthenticationProvider` 是如何交互的，从而

AuthenticationManager 提供认证支持：



正如你可能想象的那样，认证是相当可配置化的。大多数的 Spring Security 例子要么使用基于内存的用户凭证存储要么使用 JDBC（在数据库中）的用户凭证存储。我们已经意识到修改 JBCP Pets 应用以实现数据库存储用户凭证是一个好主意，我们将会在第四章来处理这个配置变化。

什么时候校验不通过？

Spring Security 很好的使用应用级异常（expected exceptions）来表示处理各种的结果情况。你可能在使用 Spring Security 的日常工作中不会与这些异常打交道，但是了解它们以及它们为何被抛出将会在调试问题或理解应用流程中非常有用。

所有认证相关的异常都继承自 `o.s.s.core.AuthenticationException` 基类。除了支持标准的异常功能，`AuthenticationException` 包含两个域，可能在提供调试失败信息以及报告信息给用户方面很有用处。

- `authentication`: 存储关联认证请求的 `Authentication` 实例；
- `extraInformation`: 根据特定的异常可以存储额外的信息。如 `UsernameNotFoundException` 在这个域上存储了用户名。

我们在下面的表格中，列出了常见的异常。完整的认证异常列表可以在附录：参考资料中找到：

异常类	何时抛出	extraInformation 内容
<code>BadCredentialsException</code>	如何没有提供用户名或者密码与认证存储中用户名对应的密码不匹配	<code>UserDetails</code>
<code>LockedException</code>	如果用户的账号被发现锁定了	<code>UserDetails</code>

UsernameNotFoundException	如果用户名不存在或者用户 没 有 被 授 予 的 GrantedAuthority	String（包含用户名）
---------------------------	---	---------------

这些以及其他的一些异常将会传递到过滤器链上，通常将被 request 请求的过滤器捕获并处理，要么将用户重定向到一个合适的界面（登录或访问拒绝），要么返回一个特殊的 HTTP 状态码，如 HTTP 403（访问被拒绝）。

请求是怎样被授权的？

在 Spring Security 的默认过滤器链中，最后一个 servlet 过滤器是 FilterSecurityInterceptor，它的作用是判断一个特定的请求是被允许还是被拒绝。在 FilterSecurityInterceptor 被触发的时候，安全实体已经经过了认证，所以系统知道他们是合法的用户。（其实也有可能是匿名的用户，译者注）。请记住的一点是，Authentication 提供了一个方法（`List<GrantedAuthority> getAuthorities()`），将会返回当前安全实体的一系列权限列表。授权的过程将使用这个方法提供的信息来决定一个特定的请求是否会被允许。

需要记住的是授权是一个二进制的决策——一个用户要么有要么没有访问一个受保护资源的权限。在授权中，没有模棱两可的情景。

在 Spring Security 中，良好的面向对象设计随处可见，在授权决策管理中也不例外。回忆一下我们在本章前面的讨论，一个名为访问控制决策器（access decision manager）的组件负责作出授权决策。

在 Spring Security 中，`o.s.s.access.AccessDecisionManager` 接口定义了两个简单而合理的方法，它们能够用于请求的决策判断流程：

- `supports`: 这个逻辑操作实际上包含两个方法，它们允许 AccessDecisionManager 的实现类判断是否支持当前的请求。
- `decide`: 基于请求的上下文和安全配置，允许 AccessDecisionManager 去核实访问是否被允许以及请求是否能够被接受。`decide` 方法实际上没有返回值，通过抛出异常来表明对请求访问的拒绝。

与 AuthenticationException 及其子类在认证过程中的使用很类似，特定类型的异常能够表明应用在授权决策中的不同处理结果。`o.s.s.access.AccessDeniedException` 是在授权领域里最常见的异常，因此值得过滤器链进行特殊的处理。我们将在第六章中详细介绍它的高级配置。AccessDecisionManager 是能够通过标准的 Spring bean 绑定和引用实现完全的自定义配置。AccessDecisionManager 的默认实现提供了一个基于 AccessDecisionVoter 接口和投票集合的授权机制。

投票器（voter）是在授权过程中的一个重要角色，它的作用是评估以下的内容：

- 要访问受保护资源的请求所对应上下文（如 URL 请求的 IP 地址）；
- 用户的凭证信息（如果存在的话）；
- 要试图访问的受保护资源；
- 系统的配置以及要访问资源本身的配置参数。

AccessDecisionManager 还会负责传递要请求资源的访问声明信息（在代码中为 ConfigAttribute 接口的实现类）给投票器。在 web URL 的请求中，投票器将会得到资源的访问声明信息。如果看一下我们配置文件中非常基础的拦截声明，我们能够看到 ROLE_USER 被设置为访问配置并用于用户试图访问的资源：

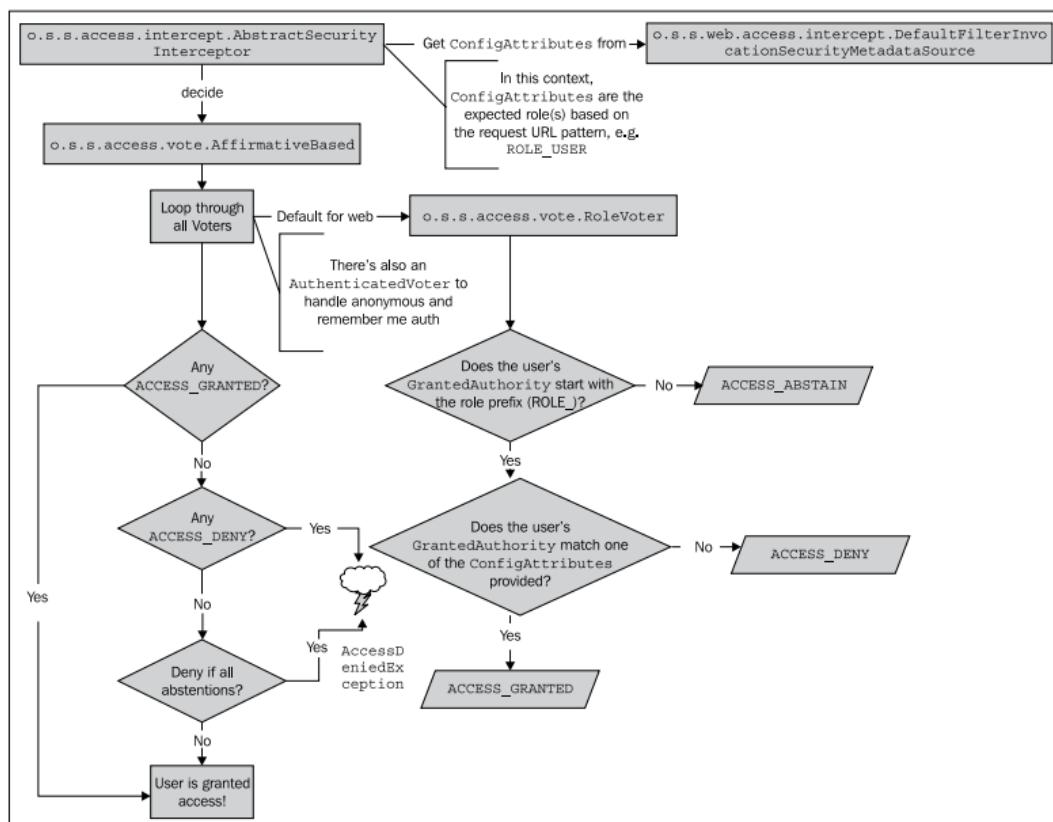
```
<intercept-url pattern="/*" access="ROLE_USER"/>
```

投票器将会对用户是否能够访问指定的资源做出一个判断。Spring Security 允许过滤器在三种决策结果中做出一种选择，它们的逻辑定义在 `o.s.s.access.AccessDecisionVoter` 接口中通过常量进行了定义。

决策类型	描述
Grant (ACCESS_GRANTED)	投票器允许对资源的访问
Deny (ACCESS_DENIED)	投票器拒绝对资源的访问
Abstain (ACCESS_ABSTAIN)	投票器对是否能够访问做了弃权处理（即没有做出决定）。可能在多种原因下发生，如： <ul style="list-style-type: none"> ● 投票器没有确凿的判断信息； ● 投票器不能对这种类型的请求做出决策。

正如你从访问决策相关类和接口的设计中可以猜到的那样，Spring Security 的这部分被精心设计，所以认证和访问控制的使用场景并不仅仅限于 web 领域。我们将会在：精确的访问控制中关于方法级别的安全时，再次讲解投票器和访问控制管理。

当将他们组合在一起，“对 web 请求的默认认证检查”的整体流程将如下图所示：



我们可以看到 `ConfigAttribute` 能够从配置声明（在 `DefaultFilterInvocationSecurityMetadataSource` 类中保存）中传递数据到投票器，投票器并不需要其他的类来理解 `ConfigAttribute` 的内容。这种分离能够为新类型的安全声明（例如我们将要看到的方法安全声明）使用相同的访问决策模式提供基础。

配置 access decision 集合

实际上 Spring Security 允许通过 security 命名空间来配置 AccessDecisionManager。<http>元素的 access-decision-manager-ref 属性来指明一个实现了 AccessDecisionManager 的 Spring Bean。Spring Security 提供了这个接口的三个实现类，都在 o.s.s.access.vote 包中：

类名	描述
AffirmativeBased	如果有任何一个投票器允许访问，请求将被立刻允许，而不管之前可能有的拒绝决定。
ConsensusBased	多数票（允许或拒绝）决定了 AccessDecisionManager 的结果。平局的投票和空票（全是弃权的）的结果是可配置的。
UnanimousBased	所有的投票器必须全是允许的，否则访问将被拒绝。

配置使用 UnanimousBased 的访问决策管理器（access decision manager）

如果你想修改我们的应用来使用 UnanimousBased 访问决策管理器，我们需要修改两个地方。首先让我们在<http>元素上添加 access-decision-manager-ref 属性：

```
<http auto-config="true"
      access-decision-manager-ref="unanimousBased" >
```

这是一个标准的 Spring Bean 的引用，所以这需要对应一个 bean 的 id 属性。接下来，我们要定义这个 bean（在 dogstore-base.xml 中），并与我们引用的有相同的 id：

```
<bean class="org.springframework.security.access.vote.UnanimousBased"
      id="unanimousBased">
    <property name="decisionVoters">
      <list>
        <ref bean="roleVoter"/>
        <ref bean="authenticatedVoter"/>
      </list>
    </property>
  </bean>
  <bean class="org.springframework.security.access.vote.RoleVoter"
        id="roleVoter"/>
  <bean class="org.springframework.security.access.vote.
AuthenticatedVoter" id="authenticatedVoter"/>
```

你可能想知道 decisionVoters 属性是什么。这个属性在我们不声明 AccessDecisionManager 时，是自动配置的。默认的 AccessDecisionManager 要求我们配置投票器的一个列表，它们将会

在认证决策时用到。这里列出的两个投票器是 security 命名空间配置默认提供的。

遗憾的是，Spring Security 没有为我们提供太多的投票器，但是实现 AccessDecisionVoter 接口并在配置中添加我们的实现并不是一件困难的事情。我们将在第六章看一个例子。

我们引用的两个投票器介绍如下：

类名	描述	例子
o.s.s.access.vote.RoleVoter	检查用户是否拥有声明角色的权限（GrantedAuthority）。access 属性定义了 GrantedAuthority 的一个列表。预期会有 ROLE_ 前缀，但这也是可配置的。	access="ROLE_USER,ROLE_ADMIN"
o.s.s.access.vote.AuthenticatedVoter	支持特定类型的声明，允许使用通配符： <ul style="list-style-type: none">● IS_AUTHENTICATED_FULLY —— 允许提供完整的用户名和密码的用户访问；● IS_AUTHENTICATED_REMEMBERED —— 如果用户是通过 remember me 功能认证的则允许访问；● IS_AUTHENTICATED_ANONYMOUSLY——允许匿名用户访问。	access="IS_AUTHENTICATED_ANONYMOUSLY"

使用 Spring 表达式语言配置访问控制

基于角色标准投票机制的标准实现是使用 RoleVoter，还有一种替代方法可用来定义语法复杂的投票规则即使用 Spring 表达式语言（SpEL）。要实现这一功能的直接方式是在<http>配置元素上添加 use-expressions 属性：

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

添加后将要修改来进行拦截器规则声明的 access 属性，改为 SpEL 表达式。SpEL 允许使用特定的访问控制规则表达式语言。与简单的字符串如 ROLE_USER 不同，配置文件可以指明表达式语言触发方法调用、引用系统属性、计算机值等等。

SpEL 的语法与其他的表达式语言很类似，如在 Tapestry 等框架中用到的 Object Graph Notation Language (OGNL)，以及用于 JSP 和 JSF 的 Unified Expression Language。它的语法面很广，已经超出了本书的覆盖范围，我们将会通过几个例子为你构建表达式提供一些确切的帮助。

需要注意的重要一点是，如果你通过使用 use-expressions 属性启用了 SpEL 表达式访问控制，将会使得自动配置的 RoleVoter 实效，后者能够使用角色的声明，正如在前面的例子所见到的那样：

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

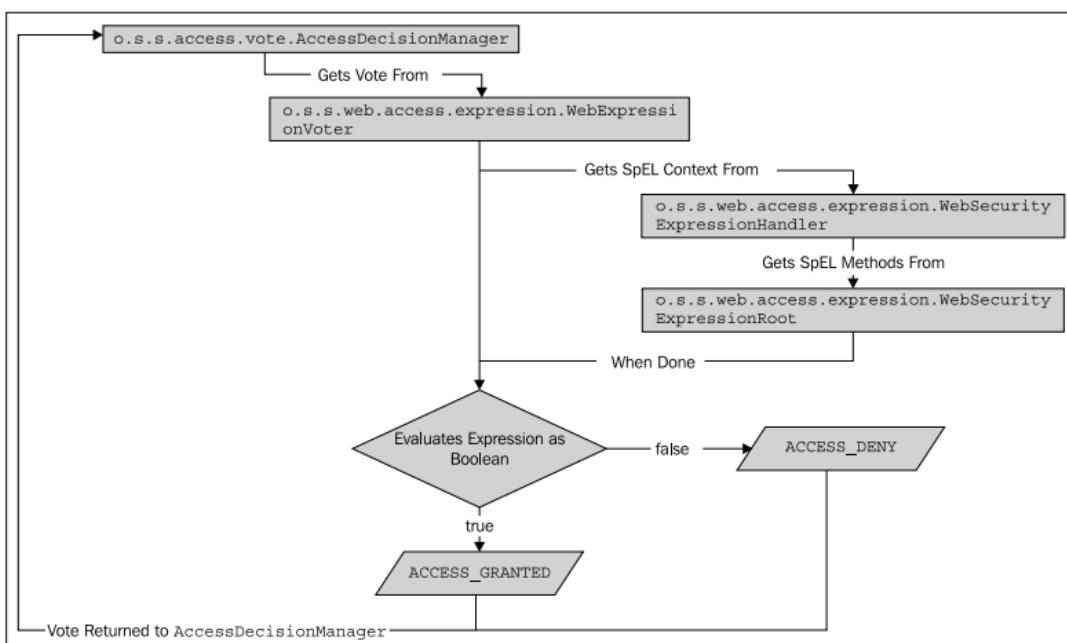
这意味着如果你仅仅想通过角色来过滤请求的话，访问控制声明必要要进行修改。幸运的的

是，这已经被充分考虑过了，一个 SpEL 绑定的方法 `hasRole` 能够检查角色。如果我们要使用表达式来重写例子的配置，它可能看起来如下所示：

```
<http auto-config="true" use-expressions="true">
    <intercept-url pattern="/*" access="hasRole('ROLE_USER')"/>
</http>
```

正如你可能预料的那样，SpEL 使用了一个不同的 Voter 实现类，即 `o.s.s.web.access.expression.WebExpressionVoter`，它能理解怎样解析 SpEL 表达式。`WebExpressionVoter` 借助于 `o.s.s.web.access.expression.WebSecurityExpressionHandler` 接口的一个实现类来达到这个目的。`WebSecurityExpressionHandler` 同时负责评估表达式的执行结果以及提供在表达式中应用的安全相关的方法。这个接口的默认实现对外暴露了 `o.s.s.web.access.expression.WebSecurityExpressionRoot` 类中定义的方法。

这些类的流程以及关系如下图所示：



为实现 SpEL 访问控制表达式的方法和伪属性（pseudo-property）在类 `WebSecurityExpressionRoot` 及其父类的公共方法中进行了声明。

【伪属性(pseudo-property)是指没有传入参数并符合 JavaBeans 的 getters 命名格式的方法。这允许 SpEL 表达式能够省略方法的圆括号以及 is 或 get 的前缀。如 `isAnonymous()` 方法可以通过 `anonymous` 伪属性来访问。】

Spring Security 3 提供的 SpEL 方法和伪属性在以下的表格中进行了描述。要注意的是没有被标明“web only”的方法和属性可以在保护其他类型的资源中使用，如在保护方法调用时。示例表示的方法和属性是使用在`<intercept-url>`的 `access` 声明中。

方法	Web only?	描述	示例
<code>hasIpAddress(ipAddress)</code>	Yes	用于匹配一个请求的 IP 地址或一个地址的网络掩码	<code>access="hasIpAddress('162.79.8.30')"</code> <code>access="hasIpAddress('162.0.0.0/224')"</code>
<code>hasRole(role)</code>	No	用于匹配一个使用	<code>access="hasRole('ROLE')</code>

		GrantedAuthority 的角色（类似于 RoleVoter）	USER'"
hasAnyRole(role)	No	用于匹配一个使用 GrantedAuthority 的角色列表。用户匹配其中的任何一个均可放行。	access="hasRole('ROLE_USER','ROLE_ADMIN')"

除了以上表格中的方法，在 SpEL 表达式中还有一系列的方法可以作为属性。它们不需要圆括号或方法参数。

属性	Web only?	描述	例子
permitAll	No	任何用户均可访问	access="permitAll"
denyAll	NO	任何用户均不可访问	access="denyAll"
anonymous	NO	匿名用户可访问	access="anonymous"
authenticated	NO	检查用户是否认证过	access="authenticated"
rememberMe	No	检查用户是否通过 remember me 功能认证的	access="rememberMe"
fullyAuthenticated	No	检查用户是否通过提供完整的凭证信息来认证的	access="fullyAuthenticated"

需要记住的是，voter 的实现类必须基于请求的上下文返回一个投票的结果（允许、拒绝或者弃权）。你可能会认为 hasRole 会返回一个 Boolean 值，实际上正是如此。基于 SpEL 的访问控制声明必须是返回 Boolean 类型的表达式。返回值为 true 意味着投票器的结果是允许访问，false 的结果意味着投票器拒绝访问。

【如果一个表达式的值不是 Boolean 类型的，你将会得到如下的一个异常信息：

org.springframework.expression.spel.SpelException:

EL1001E:Type conversion problem, cannot convert from

class java.lang.Integer to java.lang.Boolean】

另外，表达式不能返回一个弃权类型的结果，除非访问控制声明不是一个合法 SpEL 表达式，在这种情况下投票器将会放弃投票。

如果你不在乎这些细小的约束，SpEL 访问控制声明能够提供一种灵活的配置访问控制决策的方式。

小结

在本章中，我们提供了安全领域两个重要概念即认证和授权的介绍。

- 在总体上了解我们要进行安全保护的系统；
- 使用 Spring Security 的自动配置在三步之内实现了我们应用的安全配置；
- 了解了在 Spring Security 中 servlet 过滤器的使用及重要性；
- 了解了认证和授权过程中重要的角色，包括一些重要类实现的详细介绍如 Authentication 和 UserDetails
- 体验了与访问控制规则有关的 SpEL 表达式的配置。

在接下来的一章中，我们将通过添加一些增强用户体验的功能，把基于用户名和密码的认证提高一个新的水平。

第三章 增强用户体验

在本章中，我们将对 JBCP Pets 在线商店增加一些功能，这些新功能能够为用户提供更愉悦和可用的用户体验，同时提供一些对安全系统很重要的功能。

在本章中，我们将要：

- 按照你的意愿自定义登录和退出页面，并将它们与标准的 Spring web MVC 的控制器相关联；
- 使用 remember me 功能为用户提供便利，并理解其背后的安全含义；
- 构建用户账号管理功能，包括修改密码以及密码遗忘找回功能。

自定义登录页

你可能还记得在前一章中，我们使用了 Spring Security 的 security 命名空间的基本配置功能。这为我们提供了基本的登录、认证和授权功能，但是这肯定没有到达产品质量的要求。在我向老板汇报进度前，要添加的一个很重要的增强功能就是使得登录界面在展现和行为上与我们在线应用的其他地方保持一致。

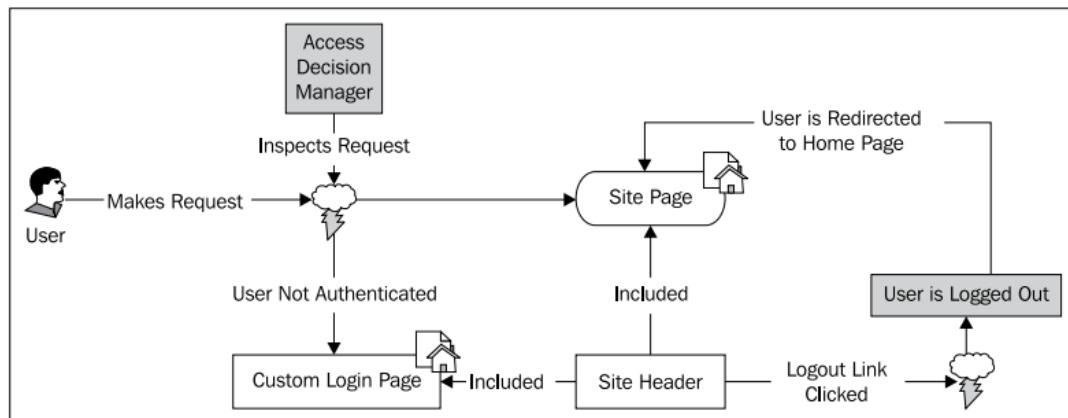
回忆一下现在的登录界面大致如下所示：

The form is titled "Login with Username and Password". It contains two text input fields: "User" with the value "guest" and "Password" with five asterisks. Below the inputs are two buttons: "Submit Query" and "Reset".

自动配置并没有为我们提供很多其他的功能，如为登录页面添加样式。我们要为站点增加以下的功能：

- 拥有页头、页脚以及与 JBCP Pets 样式一致的登录页；
- 允许用户退出的链接
- 允许用户修改密码的页面。

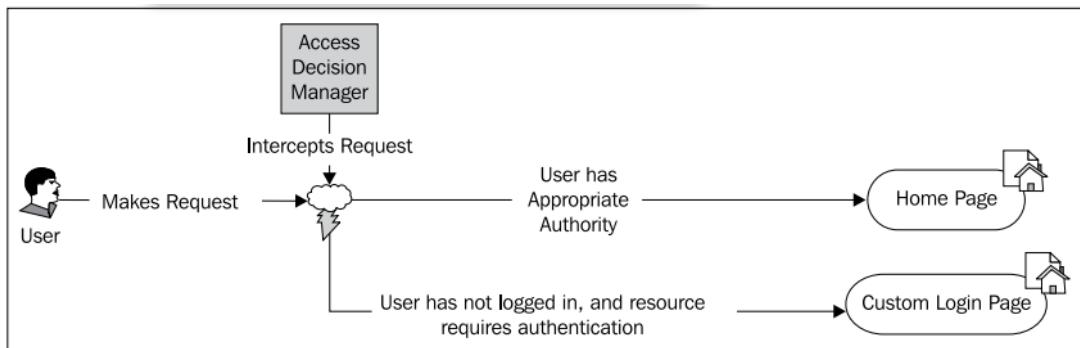
登录和退出的流程应该如下图所示：



我们将会通过一系列的练习来开发完善这个站点的结构。当开发登录和退出功能时，我们将会讲解所做的内容，所以当我们需要扩展站点的基本功能时，能够对于我们构建的内容有一个清晰的理解。

实现自定义的登录页

首先，我们需要一个集成于我们系统的登录页来替代默认的 Spring Security 登录页。需要的登录流程如下图所示：



实现登录的 controller

我们需要添加一个 Spring MVC 的控制器来实现登录功能，以及以后的退出功能。JBCP Pets 站点使用 Spring MVC 基于注解的机制来实现控制器与站点路径和资源的配置。让我们在包下 com.packtpub.springsecurity.web.controller 创建一个名为 LoginLogoutController 的 controller，并包含以下的内容：

```
// imports omitted
@Controller
public class LoginLogoutController extends BaseController{
    @RequestMapping(method=RequestMethod.GET,value="/login.do")
    public void home() {
    }
}
```

可以看到，我们添加了一个非常简单的 controller，并将其唯一的方法匹配至/login.do 这个 URL 地址。这是我们编写简单的自定义登录页所要做的全部事情，这将替代 Spring Security 基本配置中为我们添加的登录页。BaseController 基类在第二章：Spring Security 起步的代码中已经添加，它提供了一个便利的地方我们可以添加应用中所有 controller 都能用到的方法。

添加登录 JSP

/login.do 引用将会导致我们在 WEB-INF/dogstore-servlet.xml 配置的 Spring MVC view resolver 去/WEB-INF/views 目录下寻找名为 login.jsp 的 JSP 文件。让我们添加一个包含登录 form 的简单 JSP，它将被 Spring Security 识别和使用。在第二章中我们已经学到，为了保证接下来的行为能够被正确的执行，登录的 form 中有两个重要的元素必须要被正确的设置：

- Form action 必须与 UsernamePasswordAuthenticationFilter 过滤器的 action 的配置相一致。
默认的 form action 是 j_spring_security_check;
- 用户名和密码的表单域要与 servlet 的标准相一致。默认 j_username 和 j_password 是文本域的名字。

我们同时会在这个 JSP 中包含站点的页头和页脚（本章的示例代码中添加了这部分，但是在本书的内容中没有进行罗列，因为它们在这里并不是阐述的重点所在）。这些完成后，得到一个简单的 JSP：

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<jsp:include page="common/header.jsp">
    <jsp:param name="pageTitle" value="Login"/>
</jsp:include>
<h1>Please Log In to Your Account</h1>
<p>
    Please use the form below to log in to your account.
</p>
<form action="j_spring_security_check" method="post">
    <label for="j_username">Login</label>:
    <input id="j_username" name="j_username" size="20" maxlength="50"
    type="text"/>
    <br />
    <label for="j_password">Password</label>:
    <input id="j_password" name="j_password" size="20" maxlength="50"
    type="password"/>
    <br />
    <input type="submit" value="Login"/>
</form>
<jsp:include page="common/footer.jsp"/>
```

需要注意的是，必须使用 post 方式的 form 提交，否则 UsernamePasswordAuthenticationFilter 会拒绝登录请求。

最后，我们还需要 Spring Security 的自动配置来引用我们新的登录页面。如果你在此时迫不及待想看一下效果的话，我们实际上只是为应用增加了一个新的工作页面。按照上面的流程并输入以下的地址 <http://localhost:8080/JBCPPets/login.do>，看看发生了什么。

什么？你是否发现你的请求首先被 Spring Security 拦截了（被重定向到 spring_security_login）并且能够看见那个登录的 form？这是因为 Spring Security 依旧指向了 DefaultLoginPageGeneratingFilter 生成的默认登录页。一旦你通过了这个过滤器生成的默认登录页，你才能够看到新的自定义登录页。最后一步就是要移除默认页并使用我们的登录 form 作为登录页。

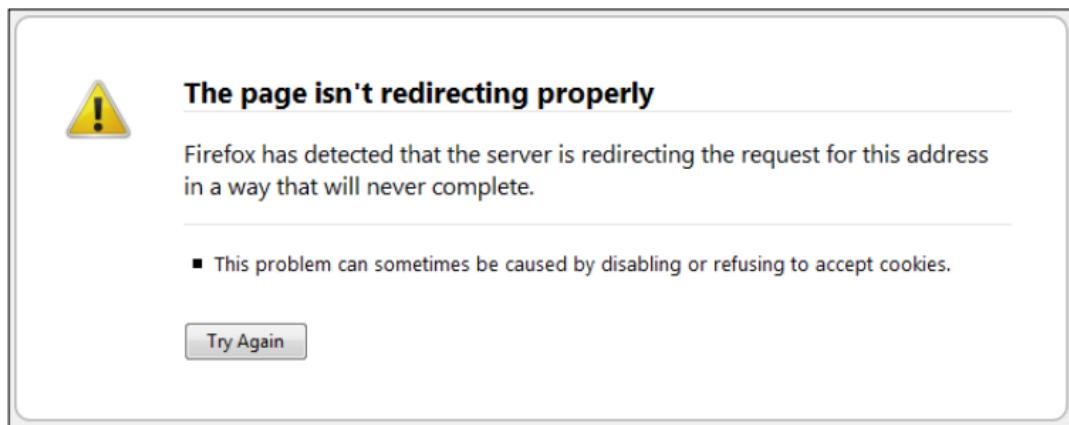
配置 Spring Security 以使用我们的 Spring MVC 登录页

按照第一感觉，貌似我们只需要配置 Spring Security 的配置文件中的<form-login>元素并添加 login-page 命令，大致如下所示：

```
<http auto-config="true" use-expressions="true">
    <intercept-url pattern="/*" access="hasRole('ROLE_USER')"/>
    <form-login login-page="/login.do" />
</http>
```

现在，启动应用并输入首页地址（<http://localhost:8080/JBCPPets/home.do>）。如果你使用的

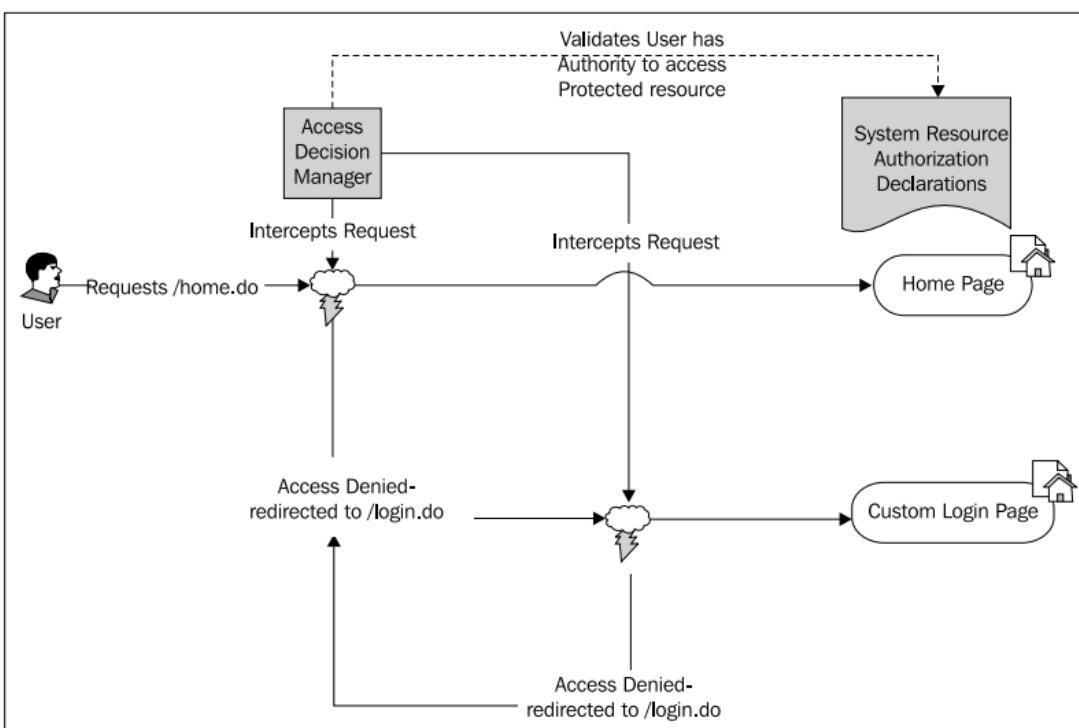
IE 浏览器，你会发现页面根本没有渲染，但是页面的似乎在不停的加载。让我们切换到 Mozilla Firefox 并访问同样的地址。在 Firefox 下，你能够看到更多的信息，如下所示：



产生这样的问题是因为我们的 URL 拦截规则：

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

这将要求访问所有匹配/*的 URL（这将匹配应用的所有页面，包括我们新的登录页）都需要拥有 ROLE_USER 角色。下面的图展现了发生了什么事情：



（其实上面发生了反复请求登录页的情况，死循环了——译者注）

我们需要修改认证规则来允许匿名用户能够访问登录页。

【对于所有给定的 URL 请求，Spring Security 按照自顶向下的顺序评估认证规则。第一个匹配 URL 模式的规则将会被使用。这意味着你的授权规则将要按照最特殊的到最不特殊的规则来进行排列。这在开发复杂的规则集合时将会非常重要，因为开发人员经常会感到迷惑，因为他们有时会搞不清到底哪个规则会生效。记住自顶向下顺序，你将能够很容易地在任何场景下找到正确的对应规则。】

因为我们是要添加一个更特殊的规则，所以我们需要将其添加在列表的顶部。我们最终会得

到如下的规则设置：

```
<intercept-url pattern="/login.do" access="permitAll"/>
<intercept-url pattern="/*" access="hasRole('ROLE_USER')"/>
```

这将能够达到我们想要的效果：允许任何用户访问登录页而限制站点的其他部分只能是认证用户才能访问。到此为止，已经完成了登录功能。让我们看一下要添加退出功能都需要做些什么。

理解退出功能

术语退出（Logout）指的是用户使其安全 session 实效的一种操作。一般来说，用户在退出后，将会被重定向到站点的非安全保护的界面。让我们在站点的页头部分添加一个“Log Out”的链接，并再次访问站点以了解其如何实现功能的。

在站点页头上添加“Log Out”链接

正如我们在第二章中讨论的那样，Spring Security 将会监视一些特殊的 URL，这些 URL 将会触发过滤器链中的一个或多个过滤器。用户实现退出的 URL 标识为/j_spring_security_logout。在 header.jsp 中添加一个退出的链接与为一个锚标签（即 a 标签）添加适合的 href 属性一样简单：

```
<c:url value="/j_spring_security_logout" var="logoutUrl"/>
<li><a href="${logoutUrl}">Log Out</a></li>
```

如果你重新加载站点并点击“Log Out”链接，你会发现被重置到登录 form 的界面。现在看来，登录和退出是很有趣的！

【使用 JSTL URL 标签来处理相对 URL。我们使用了 JSTL 核心库中的 url 标签来保证提供的 URL 在部署的 web 应用中能够被正确处理。url 标签将提供的 URL 按照相对路径（以/开始）进行处理。你可能会见过其他类似的实现技术如使用 JSP 的代码（<%= request.getContextPath() %>），但是 JSTL 的 url 标签能够使得你免于编写内联的代码。】
让我们看一下在这个简单操作的背后都发生了什么。

退出是怎么实现的

当我们点击退出链接时，到底发生了什么？

需要记住的一点是任何 URL 请求在被 servlet 处理之前，都会经过 Spring Security 的过滤器链。所以，/j_spring_security_logout 这个 URL 请求并非对应系统中的一个 JSP，也不必有一个真正的 JSP 或者 Spring MVC 的目标来对其进行处理。这种类型的 URL 通常被称为虚拟 URL。

请求/j_spring_security_logout 的 URL 被 o.s.s.web.authentication.logout.LogoutFilter 过滤器所拦截。在 Spring Security 的众多默认过滤器中，LogoutFilter 专门匹配这个虚拟 URL 并执行相应的操作。

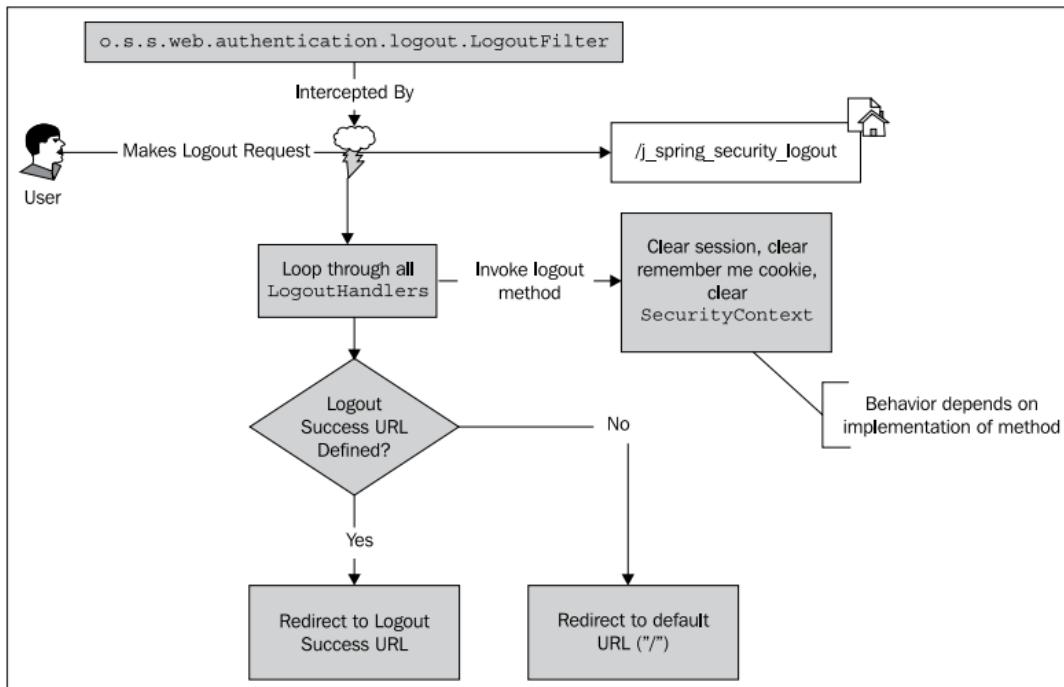
让我们快速地查看一下 Spring Security 的 security 命名空间提供的默认退出功能：

```
<http auto-config="true" use-expressions="true">
    <logout invalidate-session="true"
        logout-success-url="/"
        logout-url="/j_spring_security_logout"/>
</http>
```

基于这个基本配置，系统将会寻找在 `logout-url` 属性配置的 URL 并实现用户的退出。使得用户退出系统将会涉及如下的三个步骤：

1. 使得 HTTP session 失效（如果 `invalidate-session` 属性被设置为 true）；
2. 清除 `SecurityContext`（真正使得用户退出）；
3. 将页面重定向至 `logout-success-url` 指明的 URL。

以下的图片阐述了退出的过程：



`o.s.s.web.authentication.logout.LogoutHandler` 接口的实现类可以在用户通过 `LogoutFilter` 退出时被调用。你可以实现自己的 `LogoutHandler`（尽管比较复杂）并将其关联到 `LogoutFilter` 的生命周期中。通过 `LogoutFilter` 默认设置的 `LogoutHandler` 将会清除 session 以及 remember me 相关的功能，所以用户的 session 中不会再持有认证相关的信息。最后，通过一个 `o.s.s.web.authentication.logout.LogoutSuccessHandler` 接口的默认实现，页面得以重定向到一个 URL。默认实现中会将页面重定向到我们配置的成功退出 URL 地址（默认为 /），但是我们自定义任何系统在用户退出时想要的操作。值得注意的是，退出的处理不应该抛出异常，因为很重要的一点是要在用户的安全 session 中避免可能出现的潜在不一致性。所以在实现自己的安全处理时要保证异常被正确的处理和记录。

修改 logout URL

让我们尝试重写默认的 `logout` URL 来提供一个修改自动设置的简单例子。我们将会修改 `logout` URL 为 `/logout`。

修改 `dogstore-security.xml` 配置文件来包含`<logout>`元素如下的代码所示：

```
<http auto-config="true" use-expressions="true">
...
..<logout invalidate-session="true"
    logout-success-url="/"
    logout-url="/logout"/>
</http>
```

修改`/common/header.jsp` 文件来改变 `logout` 链接的 `href` 属性以匹配新的 URL:

```
<c:url value="/logout" var="logoutUrl"/>
<li><a href="${logoutUrl}">Log Out</a></li>
```

重新启动应用并进行尝试。你可以发现使用`/logout` URL 取代了`/j_spring_security_logout` 实现用户的退出。你可能也会发现当你尝试`/j_spring_security_logout` 这个地址时，你会得到一个 `Page not Found(404)` 的错误，是因为这个 URL 不与任何一个实际的 `servlet` 资源相对应并且不会被过滤器所处理。

Logout 配置命令

`<logout>`元素包含其他的配置指令以实现更复杂的退出功能，介绍如下：

属性	描述
<code>invalidate-session</code>	如果被设置为 <code>true</code> ，用户的 <code>HTTP session</code> 将会在退出时被失效。在一些场景下，这是必要的（如用户拥有一个购物车时）
<code>logout-success-url</code>	用户在退出后将要被重定向到的 <code>URL</code> 。默认为 <code>/</code> 。将会通过 <code>HttpServletResponse.sendRedirect</code> 来处理。
<code>logout-url</code>	<code>LogoutFilter</code> 要读取的 <code>URL</code> （在例子中，我们改变了它的设置）。
<code>success-handler-ref</code>	对一个 <code>LogoutSuccessHandler</code> 实现的引用。

Remember me

对于经常访问站点的用户有一个便利的功能就是 `remember me`。这个功能允许一个再次访问的用户能够被记住，它通过在用户的浏览器上存储一个加密的 `cookie` 来实现的。如果 `Spring Security` 能够识别出用户提供的 `remember me cookie`，用户将不必填写用户名和密码，而是直接登录进入系统。

与到目前为止我们介绍的其它功能不同，`remember me` 功能并不是在使用 `security` 命名空间方式的配置中自动添加的。让我们尝试这个功能并查看它是怎样影响登录流程的。

实现 remember me 选项

完成这个练习后，将会为用户访问 pet store 应用提供一个简单的记忆功能。

修改 dogstore-security.xml 配置文件，添加<remember-me>声明。设置 key 属性为 jbcppPetStore：

```
<http auto-config="true" use-expressions="true" access-decision-
manager-ref="affirmativeBased">
...
<remember-me key="jbcppPetStore"/>
<logout invalidate-session="true" logout-success-url="/" logout-url="/logout"/>
</http>
```

如果我们现在尝试使用应用，在流程上将不会看到有任何的变化。这是因为还需要在登录 form 上添加一个输入域，以允许用户选择使用这个功能。编辑 login.jsp 文件，添加一个 checkbox，代码如下：

```
<input id="j_username" name="j_username" size="20" maxlength="50" type="text"/>
<br />
<input id="_spring_security_remember_me" name="_spring_security_
remember_me" type="checkbox" value="true"/>
<label for="_spring_security_remember_me">Remember Me?</label>
<br />
<label for="j_password">Password</label>:
```

当我们再次登录时，如果 Remember Me 被选中，一个 Remember Me 的 cookie 将会设置在用户的浏览器中。

如果用户关闭浏览器并重新打开访问一个 JBCP Pets 站点上需要认证的页面，他将不会再看到登录页了。请亲自试一下——登录并将 Remember Me 选项选中，收藏首页，然后重启浏览器并再次访问首页。你能发现你直接登录成功并不再需要提供凭证。

一些高级的用户在体验本功能时也可以使用浏览器插件如 Firecookie (<http://www.softwareishard.com/blog/firecookie/>)，来管理（移除）会话 session。这将会在你开发或校验这种类型功能时，节省你时间和提高效率。

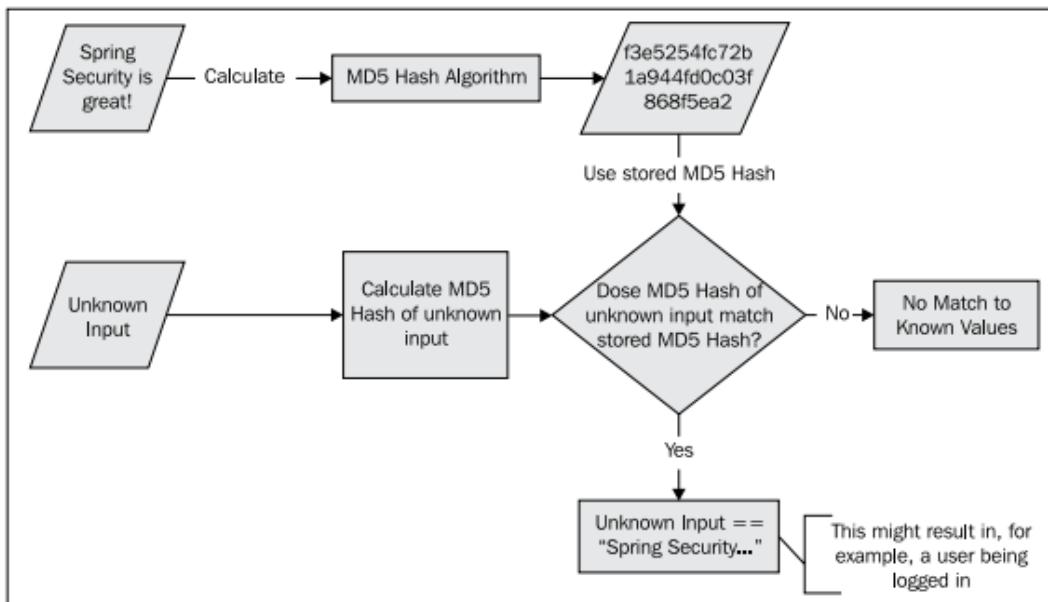
Remember me 是怎样实现的

Remember me 功能设置了一个 cookie 在用户的浏览器上，它包含一个 Base64 编码的字符串，包含以下内容：

- 用户的名字；
- 过期的日期/时间；
- 一个 MD5 的散列值包括过期日期/时间、用户名和密码；
- 应用的 key 值，是在<remember-me>元素的 key 属性中定义的。

这些内容将被组合成一个 cookie 的值存储在浏览器中以备后用。

MD5 是一种知名的加密哈希算法。加密哈希算法将输入的数据进行压缩并生成唯一的任意长度的文字，这叫做摘要。摘要能够在以后使用，以校验不明的输入是否与生成 hash 的输入内容完全一致，此时并不需要使用原来的输入内容本身。下面的图片展示了这个过程：



你可以看到，未知的输入可以与存储的 MD5 哈希进行校验，并能够得出未知的输入与存储的已知输入是否匹配的结论。摘要和加密算法(encryption algorithms)的一个重要不同在于，它很难从反向工程从摘要值得到初始的数据。这是因为摘要仅仅是原始内容的一个概述或指纹 (fingerprint,)，并不是全部的数据本身（它可能会很大）。

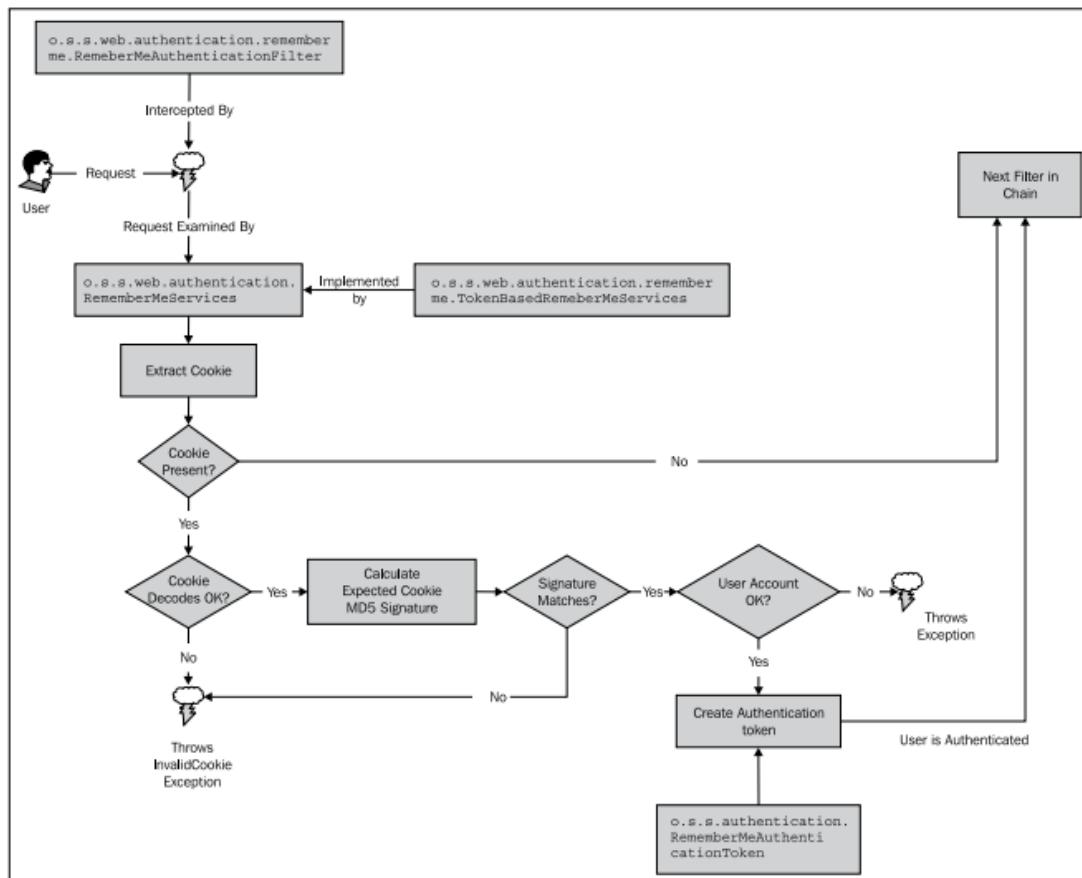
尽管对加密的数据不可能进行解码，但是 MD5 对一个类型的国际却很脆弱，包括挖掘算法本身的弱点以及彩虹表攻击 (rainbow table attacks)。彩虹表通常会包括数百万输入值计算出来的哈希值。这使得攻击者能够寻找彩虹表中的哈希值从而确定实际值（未经过 hash 的值）。我们将会在第四章：凭证安全存储中讲解密码安全的时，介绍防范这种攻击的一种方法。

关于 remember me 的 cookie，我们能够看到这个 cookie 的组成足够复杂，所以对攻击者来说很难造出一个仿冒的 cookie。在第四章中，我们将会学习另一种技术来使得 remember me 功能更加安全，免受恶意攻击。

Cookie 的失效时间基于一个配置的过期时间段的长度。如果用户在 cookie 失效之前重新访问我们的站点，这个 cookie 将和应用设置的其它 cookie 一起提交到应用上。

如果存在 remember me cookie，o.s.s.web.authentication.rememberme.RememberMeAuthenticationFilter 过滤器将会检查 cookie 的内容并通过检查是否为一个认证过的 remember me cookie 来认证用户（查看本章后面的 Remember me 是否安全？章节，将会讲述这样做的原因），这个过滤器是通过<remember-me>配置指令添加到过滤器链中的。

下面的图表阐述了校验 remember me cookie 过程中涉及到的不同组件：



RememberMeAuthenticationFilter 在过滤器链中，位于 SecurityContextHolderAwareRequestFilter 之后，而在 AnonymousProcessingFilter 之前。正如链中的其它过滤器那样，RememberMeAuthenticationFilter 也会检查 request，如果是其关注的，对应的操作将会被执行。

按图中所述，过滤器负责检查用户的过滤器是否 remember me cookie 作为它们请求的一部分。如果 remember me 被发现，它会是一个 Base64 的编码，期望的 MD5 哈希值通过 cookie 中的用户名和密码进行计算获得。（这里感觉有些问题，因为期望的 MD5 值应该是通过应用来进行获取，而不是提供前台过来的 cookie 计算出来的。？）如果 cookie 通过这一层的校验，用户就已经登录成功了。

【你可能已经意识到如果用户修改了用户名或密码，任何的 remember me token 都将失效。请确保给用户提供适当的信息，如果允许它们修改账号的那些信息。在第四章中，我们将会看到一个替代的 remember me 实现，它只依赖用户名并不依赖密码。】

我们看到 RememberMeAuthenticationFilter 依赖一个 o.s.s.web.authentication.RememberMeServices 的实现来校验 cookie。如果登录请求的 request 包含一个名为 _spring_security_remember_me 的 form 参数，相同的实现类也会于 form 登录成功时使用。这个 cookie 用上面提到的信息进行编码，以 Base64 编码存储在浏览器中，包含了时间戳和用户密码等信息形成的 MD5 哈希值。

需要记住的是，可以区分通过 remember me 认证的用户和提供用户名和密码（或相当凭证）认证的用户。我们将会在审查 remember me 功能的安全性的时，对其进行简单的实验。

Remember me 与用户的生命周期

RememberMeServices 的实现在用户的生命周期中（一个认证用户 session 的生命周期）在好几个地方被调用。为了使你理解 remember me 功能，了解 remember me service 完成生命周期功能的时间点将会有所帮助：

行为	应该做什么？
登录成功	实现类设置 remember me cookie（如果设置了对应的 form 参数）
登录失败	如果存在的话，实现类应该删掉这个 cookie
用户退出	如果存在的话，实现类应该删掉这个 cookie

知道 RememberMeServices 在何时以及怎样与用户的生命周期关联对于创建自定义的认证处理至关重要，因为我们需要保证任何的自定义认证处理对待 RememberMeServices 保持一致性，以保证这个功能的有效性和安全性。

Remember me 配置指令

可以修改两个常用的配置来改变 remember me 功能的默认行为：

属性	描述
Key	为 remember me cookie 定义一个唯一的 key 值，以与我们的应用关联
tokenValiditySeconds	定义时间的长度（以秒计）。Remember me 的 cookie 将在将被视为认证合法，并且也将用于设置 cookie 的过期时间。

通过对 cookie 哈希内容生成的讲解你可以推断出，Key 属性对与 remember me 功能的安全性很重要。要保证你选择的 key 值是你的应用唯一使用的，并且足够长以至于不能很容易的被猜出。

请记住本书的目的何在，我们让 key 的值相对很简单，但是如果你要使用 remember me 功能在你的应用中，建议 key 值包含应用的唯一名称以及至少 36 位长度的随机字符。密码生成工具（在 google 中搜索“online password generator”）是一个好办法来得到包含文字数字以及特殊字符组成的伪随机混合内容，以作为你的 remember me key 值。

还要记住的是应用处于不同的环境中（如开发、测试、产品级等），remember me cookie 的值也要考虑这些情况。这能够避免 remember me cookie 在测试阶段被无意中的错误使用。

一个产品环境的应用中，示例的 key 值如下：

jbcPPets-rmkey-paLLwApsifs24THosE62scabWow78PEaCh99Jus

tokenValiditySeconds 属性被用来设置 remember me token 被接受作为自动登录功能（即使要校验的 token 不合法）的秒数。本属性还会设置用户浏览器中登录 cookie 能够被保存的最长时间。

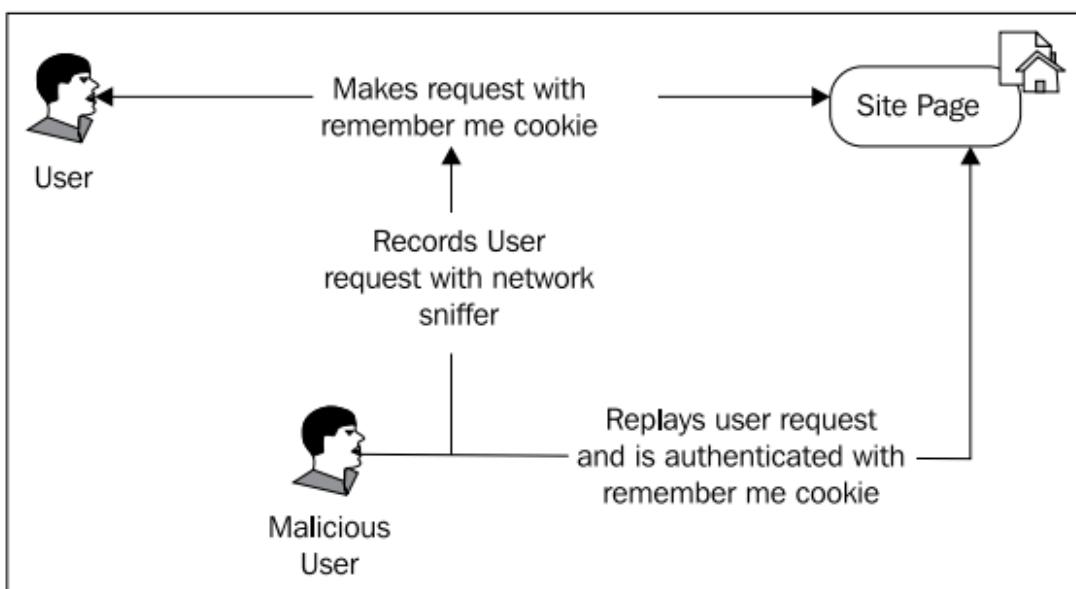
【设置 remember me 的会话 cookie】：如果 tokenValiditySeconds 属性被设置成 -1，登录 cookie 将被设置为会话 cookie，即在用户关闭浏览器后不会被保存。Token 的有效时间是一个不可

配置的值为 2 周（假设用户不关闭浏览器）。不要将这个 cookie 与存储用户的 session ID 的 cookie 相混淆——它们是不同的事情却有着类似的名字。】

关于 remember me 功能的高级自定义功能，还有几个其它的配置指令。我们将会在接下来的练习中包含部分，另一部分在第六章：高级配置与扩展中包含，那时会介绍高级的授权技术。

Remember me 是否安全？

对我们精心保护的站点来说，为了用户体验而添加的任何与安全相关的功能，都有增加安全风险的潜在可能。按照其默认方式，Remember me 功能存在用户的 cookie 被拦截并被恶意用户重用的风险。下图展现了这种情况是如何发生的：



使用 SSL（第四章进行讨论）以及其他的安全技术能缓解这种类型攻击的风险，但是要注意的是还有其他技术如跨站脚本攻击(XSS)能够窃取或损害一个 remembered user session。为了照顾用户的易用性，我们不会愿意让用户的财产信息或个人信息因为 remembered session 的不合理使用而遭到篡改或窃取。

【尽管我们不会涉及恶意用户行为的细节，但是当你实现安全系统时，了解恶意用户所使用的攻击技术是很重要的。XSS 是其中的一种技术，当然还有其他的很多种。强烈建议你了解 OWASP Top Ten (http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project) 作为一个入门列表并参考一本 web 应用安全参考书，里面介绍了各种的技术使用。】

平衡用户易用性和应用安全性的一种通用方法是识别出站点中与个人或敏感信息相关功能点。确保这些功能点在进行授权校验时不仅要判断用户的角色，还要保证用户进行了完整的用户名和密码认证。这可以通过使用 SpEL 表达式语言的 `fullyAuthenticated` 伪属性来实现，关于授权规则的 SpEL 表达式语言我们在第二章中已经有所介绍。

Remember me 认证与完整认证的区别

我们将在随后的第五章：精确的访问控制中介绍高级的认证技术，但是，了解能够辨别认证 session 是否为 `remembered` 并以此建立访问规则也是很重要的。

我们可以设想一个使用 `remembered session` 登录的用户要查看和修改他的“`wish list`”。这与其他的客户在线站点很类似，并不会出现与用户信息或财务信息相关的风险（要注意的是每个站点各不相同，不能盲目的将这些规则应用与你的站点）。相反的，我们将会重点保护用户的账号以及订单功能。我们要确保即使是 `remembered` 的用户，如果试图访问账号信息或定购产品，都需要对他们进行认证。以下为我们如何设置授权规则：

```
<intercept-url pattern="/login.do" access="permitAll"/>
<intercept-url pattern="/account/*.do"
    access="hasRole('ROLE_USER') and fullyAuthenticated"/>
<intercept-url pattern="/*" access="hasRole('ROLE_USER')"/>
```

已经存在的登录页和 `ROLE_USER` 设置没有变化。但我们添加了一条规则，要求用户具有 `GrantedAuthority` 的 `ROLE_USER` 角色同时还要求用户被完全的认证，即这个认证的 `session` 确实是通过提供用户名、密码或等同的凭证来进行认证的。注意这里的 SpEL 逻辑操作语法——在 SpEL 中，使用 `and`, `or` 以及 `not` 作为逻辑操作符。这是 SpEL 的设计者充分考虑的结果，因为`&&`操作符在 XML 中很难被使用。

如果你在应用中尝试运行，如果以 `remember me` 功能登录并试图访问“`My Account`”链接，你将会得到一个 `403` 访问拒绝的提示，这说明这个地址已经被适当地保护了。出现错误界面是因为我们应用的配置还是使用默认的 `AccessDeniedHandler`，这个类负责捕获和响应 `AccessDeniedException` 的信息。我们将会在第六章学习 `AccessDeniedException` 怎样被处理时，自定义这个行为。

【不使用表达式来实现完全认证的检查。如果你的应用不使用 SpEL 表达式来进行访问控制声明，你可以通过使用 `IS_AUTHENTICATED_FULLY` 访问规则来检查用户是不是进行了完整的认证（如：`access=" IS_AUTHENTICATED_FULLY"`）。但要注意的是，这种标准的角色设置声明并没有 SpEL 那样强的表现力，所以如果要处理复杂的 `boolean` 表达式的时候，可能会比较困难。】

错误处理尚没有添加，但是你可以看到通过这种方式将 `remember me` 的易用性与更高层次的安全性结合了起来，用户访问敏感的信息时就会被要求提供完整的凭证信息。

构建一个关联 IP 的 Remember me Service

有一种让 `remember me` 功能更安全的方式就是将用户的 IP 地址绑定到 cookie 的内容上。让我们通过一个例子来描述怎样构建 `RememberMeServices` 的实现类来完成这个功能。

基 本 的 实 现 方 式 是 扩 展
o.s.s.web.authentication.rememberme.TokenBasedRememberMeServices 基类，以添加请求者的 IP 地址到 cookie 本身和其他的 MD5 哈希元素中。

扩展这个基类涉及到重写两个主要方法，并重写或实现几个小的帮助方法。还有一个要注意的是我们需要临时存储 `HttpServletRequest`（将使用它来得到用户的 IP 地址）到一个 `ThreadLocal` 中，因为基类中的一些方法并没有将 `HttpServletRequest` 作为一个参数。

扩展 TokenBasedRememberMeServices

首先，我们要扩展 `TokenBasedRememberMeServices` 类并重写父类的特定行为。尽管父类是非常易于重写，但是我们不想去重复一些重要的处理流程，所以能使这个类非常简明却有点不好理解。在 `com.packtpub.springsecurity.security` 包下创建这个类：

```
public class IPTokenBasedRememberMeServices extends  
    TokenBasedRememberMeServices {
```

还有一些简单的方法来设置和获取 `ThreadLocal HttpServletRequest`：

```
private static final ThreadLocal<HttpServletRequest> requestHolder =  
new ThreadLocal<HttpServletRequest>();  
public HttpServletRequest getContext() {  
    return requestHolder.get();  
}  
public void setContext(HttpServletRequest context) {  
    requestHolder.set(context);  
}
```

我们还需要添加一个工具方法以从 `HttpServletRequest` 中获取 IP 地址：

```
protected String getUserIPAddress(HttpServletRequest request) {  
    return request.getRemoteAddr();  
}
```

我们要重写的一个有趣的方法是 `onLoginSuccess`，它用来为 `remember me` 处理设置 cookie 的值。在这个方法中，我们需要设置 `ThreadLocal` 并在完成处理后将其清除。需要记住的是父类方法的处理流程——收集用户的所有认证请求信息并将其合成到 cookie 中。

```
@Override  
public void onLoginSuccess(HttpServletRequest request,  
    HttpServletResponse response,  
    Authentication successfulAuthentication) {  
    try  
    {  
        setContext(request);  
    }
```

```
super.onLoginSuccess(request, response, successfulAuthentication
}
finally
{
    setContext(null);
}
}
```

父类的 `onLoginSuccess` 方法将会触发 `makeTokenSignature` 方法来创建认证凭证的 MD5 哈希值。我们将要重写此方法，以实现从 `request` 中获取 IP 地址并使用 Spring 框架的一个工具类编码要返回的 cookie 值。(这个方法在进行 `remember me` 校验时还会被调用到，以判断前台传递过来的 cookie 值与后台根据用户名、密码、IP 地址等信息生成的 MD5 值是否一致。——译者注)

```
@Override
protected String makeTokenSignature(long tokenExpiryTime,
    String username, String password) {
    return DigestUtils.md5DigestAsHex((username + ":" +
tokenExpiryTime + ":" + password + ":" + getKey() + ":" + getUserIPAdd
ress(getContext()).getBytes()));
}
```

与之类似的，我们还重写了 `setCookie` 方法以添加包含 IP 地址的附加编码信息：

```
@Override
protected void setCookie(String[] tokens, int maxAge,
    HttpServletRequest request, HttpServletResponse response) {
    // append the IP address to the cookie
    String[] tokensWithIPAddress =
        Arrays.copyOf(tokens, tokens.length+1);
    tokensWithIPAddress[tokensWithIPAddress.length-1] =
        getUserIPAddress(request);
    super.setCookie(tokensWithIPAddress, maxAge,
        request, response);
}
```

这就得到了生成新 cookie 所有需要的信息。

最后，我们要重写 `processAutoLoginCookie` 方法，它用来校验用户端提供的 `remember me` cookie 的内容。父类已经为我们解决了大部分有意思的工作，但是，为了避免调用父类冗长的代码，我们在调用它之前先进行了一次 IP 地址的校验。

```
@Override
protected UserDetails processAutoLoginCookie(
    String[] cookieTokens,
    HttpServletRequest request, HttpServletResponse response)
{
    try
```

```
{  
    setContext(request);  
    // take off the last token  
    String ipAddressToken = cookieTokens[cookieTokens.length-1];  
    if(!getUserIPAddress(request).equals(ipAddressToken))  
    {  
        throw new InvalidCookieException("Cookie IP Address did not  
contain a matching IP (contained '" + ipAddressToken + "')");  
    }  
  
    return super.processAutoLoginCookie(Arrays.copyOf(cookieTokens,  
cookieTokens.length-1), request, response);  
}  
finally  
{  
    setContext(null);  
}  
}
```

我们的自定义的 `RememberMeServices` 编码已经完成了。现在我们要进行一些微小的配置。这个类的完整源代码（包括附加的注释）都在本章的源码中能够找到。

配置自定义的 `RememberMeServices`

配置自定义的 `RememberMeServices` 实现需要两步来完成。第一步是修改 `dogstore-base.xml` Spring 配置文件，以添加我们刚刚完成类的 Spring Bean 声明：

```
<bean class="com.packtpub.springsecurity.security.IPTokenBasedRememberMeServices"  
id="ipTokenBasedRememberMeServicesBean">  
    <property name="key"><value>jbcpPetStore</value></property>  
    <property name="userDetailsService" ref="userService"/>  
</bean>
```

第二个要进行的修改是 Spring Security 的 XML 配置文件。修改`<remember-me>`元素来引用自定义的 Spring Bean，如下所示：

```
<remember-me key="jbcpPetStore"  
services-ref="ipTokenBasedRememberMeServicesBean"/>
```

最后为`<user-service>`声明添加一个 `id` 属性，如果它还没有添加的话：

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

重启 web 应用，你将能看到新的 IP 过滤功能已经生效了。

因为 remember me cookie 是 Base64 编码的，我们能够使用一个 Base64 解码的工具得到 cookie 的值以证实我们的新增功能是否生效。如果我们这样做的话，我们能够看到一个名为 SPRING_SECURITY_REMEMBER_ME_COOKIE 的 cookie 的内容大致如下所示：

```
guest:1251695034322:776f8ad44034f77d13218a5c431b7b34:127.0.0.1
```

正如我们所料，你能够看到 IP 地址确实存在于 cookie 的结尾处。在 IP 地址之前，你还能够分别看到用户名、时间戳以及 MD5 的哈希值。

【调试 remember me cookie。在尝试调试 remember me 功能时，会有两个难点。第一个就是得到 cookie 的值本身！Spring Security 并没有提供记录我们设置的 cookie 值的日志级别。我们推荐使用基于浏览器的工具如 Mozilla Firefox 下的 Chris Pederick's Web Developer 插件（<http://chrispederick.com/work/web-developer/>）。基于浏览器的开发工具一般允许查看（甚至编辑）cookie 的值。第二个困难（相对来说较小）就是解码 cookie 的值。你能使用在线或离线的 Base64 解码工具来对 cookie 的值进行解码（需要记住的是添加一个等号符 (=) 结尾，以使其成为一个合法的 Base64 编码值）。】

如果用户是在一个共享的或负载均衡的网络设施下，如 multi-WAN 公司环境，基于 IP 的 remember me tokens 可能会出现问题。但是在大多数场景下，添加 IP 地址到 remember me 功能能够为用户提供功能更强、更好的安全层。

自定义 Remember me 的签名

好奇的读者可能会关心 remember me form 的 checkbox 名(_spring_security_remember_me) 以及 cookie 的名(SPRING_SECURITY_REMEMBER_ME_COOKIE)，是否能够修改。<remember-me> 声明是不支持这种扩展性的，但是现在我们作为一个 Spring Bean 声明了自己的 RememberMeServices 实现，那我们能够定义更多的属性来改变 checkbox 和 cookie 的名字：

```
<bean class="com.packtpub.springsecurity.web.custom.  
IPTokenBasedRememberMeServices" id="ipTokenBasedRememberMeServicesBean">  
<property name="key"><value>jbcpPetStore</value></property>  
<property name="userDetailsService" ref="userService"/>  
<property name="parameter" value="_remember_me"/>  
<property name="cookieName" value="REMEMBER_ME"/>  
</bean>
```

不要忘记的是，还需要修改 login.jsp 页面中的 checkbox form 域以与我们声明的 parameter 值相匹配。我们建议你进行一下实验以确保理解这些设置之间的关联。

（如果想更好的理解本章节内容，建议阅读一下 Spring Security 的源码——译者注）

实现修改密码管理

现在我们将要对基于内存的 `UserDetailsService` 进行简单的扩展以使其支持用户修改密码。因为这个功能对用户名和密码存于数据库的场景更有用，所以基于 `o.s.s.core.userdetails.memory.InMemoryDaoImpl` 扩展的实现不会关注存储机制，而是关注框架对这种方式扩展的整体流程和设计。在第四章中，我们将通过将其转移到数据库后台存储来进一步扩展我们的基本功能。

扩展基于内存的凭证存储以支持修改密码

Spring Security 框架提供的 `InMemoryDaoImpl` 内存凭证存储使用了一个简单的 map 来存储用户名以及关联的 `UserDetails`。`InMemoryDaoImpl` 使用的 `UserDetails` 实现类是 `o.s.s.core.userdetails.User`，这个实现类将会在 Spring Security API 中还会看到。

这个扩展的设计有意的进行了简化并省略了一些重要的细节，如需要用户在修改密码前提供他们的旧密码。添加这些功能将作为练习留给读者。

用 `InMemoryChangePasswordDaoImpl` 扩展 `InMemoryDaoImpl`

我们要首先写自定义的类来扩展基本的 `InMemoryDaoImpl`，并提供允许用户修改密码的方法。因为用户是不可改变的对象，所以我们 copy 已经存在的 `User` 对象，只是将密码替换为用户提交的值。在这里我们定义一个接口在后面的章节中将会重用，这个接口提供了修改密码功能的一个方法：

```
package com.packtpub.springsecurity.security;
// imports omitted
public interface IChangePassword extends UserDetailsService {
    void changePassword(String username, String password);
}
```

以下的代码为基于内存的用户数据存储提供了修改密码功能：

```
package com.packtpub.springsecurity.security;
public class InMemoryChangePasswordDaoImpl extends InMemoryDaoImpl
implements IChangePassword {
    @Override
    public void changePassword(String username,
                               String password) {
        // get the UserDetails
        User userDetailsService =
            (User) getUserMap().getUser(username);
        // create a new UserDetails with the new password
    }
}
```

```
User newUserDetails =  
    new User(userDetails.getUsername(),password,  
    userDetails.isEnabled(),  
    userDetails.isAccountNonExpired(),  
    userDetails.isCredentialsNonExpired(),  
    userDetails.isAccountNonLocked(),  
    userDetails.getAuthorities());  
  
    // add to the map  
    getUserMap().addUser(newUserDetails);  
}  
}
```

比较幸运的是，只有一点代码就能将这个简单的功能加到自定义的子类中了。我们接下来看看添加自定义 UserDetailsService 到 pet store 应用中会需要什么样的配置。

配置 Spring Security 来使用 InMemoryChangePasswordEncoderImpl

现在，我们需要重新配置 Spring Security 的 XML 配置文件以使用新的 UserDetailsService 实现。这可能比我们预想的要困难一些，因为<user-service>元素在 Spring Security 的处理过程中有特殊的处理。需要明确声明我们的自定义 bean 并移除我们先前声明的<user-service>元素。我们需要把：

```
<authentication-manager alias="authenticationManager">  
    <authentication-provider>  
        <user-service id="userService">  
            <user authorities="ROLE_USER" name="guest" password="guest"/>  
        </user-service>  
    </authentication-provider>  
</authentication-manager>
```

修改为：

```
<authentication-provider user-service-ref="userService"/>
```

在这里我们看到的 user-service-ref 属性，引用的是一个 id 为 userService 的 Spring Bean。所以在 dogstore-base.xml Spring Beans 配置文件中，声明了如下的 bean：

```
<bean id="userService" class="com.packtpub.springsecurity.security.  
InMemoryChangePasswordEncoderImpl">  
    <property name="userProperties">  
        <props>  
            <prop key="guest">guest,ROLE_USER</prop>  
        </props>  
    </property>  
</bean>
```

你可能会发现，这里声明用户的语法不如<user-service>包含的<user>元素更易读。遗憾的是，<user> 元素只能使用在默认的 InMemoryDaolmpl 实现类中，我们不能在自定义的 UserDetailsService 中使用了。在这里例子中，这个限制使得事情稍微复杂了一点，但是在实

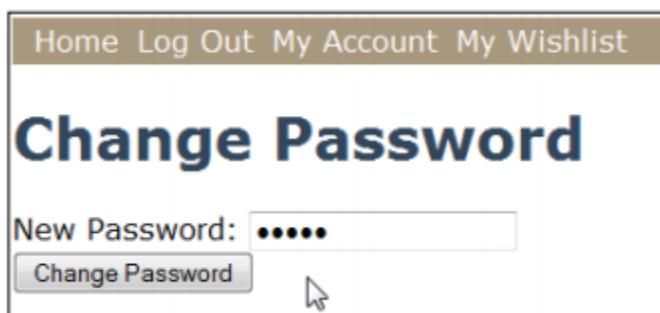
际中，没有人会愿意长期的将用户定义信息放在配置文件中。对于感兴趣的读者，Spring Security 3 参考文档中的 6.2 节详细描述了以逗号分隔的提供用户信息的语法。

【高效使用基于内存的 `UserDetailsService`。有一个常见的场景使用基于内存的 `UserDetailsService` 和硬编码的用户列表，那就是编写安全组件的单元测试。编写单元测试的人员经常编码或配置最简单的场景来测试组件的功能。使用基于内存的 `UserDetailsService` 以及定义良好的用户和 `GrantedAuthority` 值为测试编写人员提供了很可控的测试环境。】

到现在，你可以重启 JBCP Pets 应用，应该没有任何的配置错误报告。我们将在这个练习的最后的两步中，完成 UI 的功能。

构建一个修改密码的页面

我们接下来将会建立一个允许用户修改密码的简单页面。



这个页面将会通过一个简单的链接添加到“`My Account`”页面。首先，我们在 `/account/home.jsp` 文件中添加一个链接：

```
<p>
    Please find account functions below...
</p>
<ul>
    <li><a href="changePassword.do">Change Password</a></li>
</ul>
```

接下来，在 `/account/changePassword.jsp` 文件中建立“`Change Password`”页面本身：

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1" %>
<%@ page encoding="ISO-8859-1" %>
<jsp:include page="../common/header.jsp">
    <jsp:param name="pageTitle" value="Change Password"/>
</jsp:include>
<h1>Change Password</h1>
<form method="post">
    <label for="password">New Password</label>
    <input id="password" name="password" size="20" maxlength="50"
```

```
type="password"/>
<br />
<input type="submit" value="Change Password"/>
</form>
<jsp:include page="../common/footer.jsp"/>
```

最后我们还要添加基于 Spring MVC 的 AccountController 来处理密码修改的请求（在前面的章节中我们没有介绍 AccountController，它是账号信息主页的简单处理类）。

为 AccountController 添加修改密码的处理

我们需要将对自定义 UserDetailsService 的应用注入到 com.packtpub.springsecurity.web.controller.AccountController，这样我们就能使用修改密码的功能了。Spring 的@Autowired 注解实现了这一功能：

```
@Autowired
private IChangePassword changePasswordDao;
```

两个接受请求的方法分别对应渲染 form 以及处理 POST 提交的 form 数据：

```
@RequestMapping(value="/account/changePassword.
do",method=RequestMethod.GET)
public void showChangePasswordPage() {
}

@RequestMapping(value="/account/changePassword.
do",method=RequestMethod.POST)
public String submitChangePasswordPage(@RequestParam("password")
String newPassword) {
    Object principal = SecurityContextHolder.getContext().
    getAuthentication().getPrincipal();
    String username = principal.toString();
    if (principal instanceof UserDetails) {
        username = ((UserDetails)principal).getUsername();
    }
    changePasswordDao.changePassword(username, newPassword);
    SecurityContextHolder.clearContext();
    return "redirect:home.do";
}
```

完成这些配置后，重启应用，并在站点的“**My Account**”下找到“**Change Password**”功能。

练习笔记

比较精细的读者可能意识到这个修改密码的 form 相对于现实世界的应用来说太简单了。确实，很多的修改密码实现要复杂的多，并可能包含如下的功能：

- 密码确认——通过两个文本框，确保用户输入的密码是正确的；
- 旧密码确认——通过要求用户提供要修改的旧密码，增加安全性（这对使用 remember me 功能的场景特别重要）；
- 密码规则校验——检查密码的复杂性以及密码是否安全。

你可能也会注意到当你使用这个功能的时，会被自动退出。这是因为 SecurityContextHolder.clearContext() 调用导致的，它会移除用户的 SecurityContext 并要求他们重新认证。在练习中，我们需要给用户做出提示或者找到方法让用户免于再次认证。

小结

在本章中，我们更细节的了解了认证用户的生命周期并对 JBCP Pet Store 进行了结构性的修改。我们通过添加真正的登录和退出功能，进一步的满足了安全审计的要求，并提升了用户的体验。我们也学到了如下的技术：

- 配置并使用基于 Spring MVC 的自定义用户登录界面；
- 配置 Spring Security 的退出功能；
- 使用 remember me 功能；
- 通过记录 IP 地址，实现自定义的 remember me 功能；
- 实现修改密码功能；
- 自定义 UserDetailsService 和 InMemoryDaoImpl。

在第四章中，我们将会使用基于数据库的认证信息存储并学习怎样保证数据库中的密码和其他敏感数据的安全。

第四章 凭证安全存储

到现在为止，我们已经将 JBCP Pets 站点做了一些用户友好性方面的升级，包括一个自定义的登录页、修改密码以及 remember me 功能。

在本章中，我们将会把到目前为止都在使用的内存存储转移到数据库作为后台的认证存储。我们将会介绍默认的 Spring Security 数据库 schema，并介绍自定义扩展 JDBC 实现的方式。

在本章的课程中，我们将会：

- 理解如何配置 Spring Security 才能使用 JDBC 访问数据库服务以实现用户的存储和认证；
- 学习如何使用基于内存数据库 HSQLDB 的 JDBC 配置，我们使用这个数据库主要是为了开发测试的目的；
- 使得 Spring Security 的 JDBC 能够支持已经存在的遗留数据库 schema；
- 掌握两种管理用户名和密码的功能，两者都会涵盖内置的和自定义的方式；
- 掌握配置密码编码的不同方法；
- 理解密码 salting 技术以提供更安全的方式存储密码；
- 持久化用户的 remember me token，使得在服务器重启后 token 仍能有效；
- 通过配置 SSL/TLS 加密和端口映射，在传输层上保护应用的安全。

使用数据库后台的 Spring Security 认证

我们进行安全控制的 JBCP Pets 应用有一个明显问题是基于内存的用户名和密码存在时间比较短，对用户很不友好。一旦应用重启，任何的用户注册，密码修改或者其他活动都会丢失。这是不可接受的，所以对于 JBCP Pets 应用的下一个逻辑实现功能就是重新设置 Spring Security 以使用关系型数据库来进行用户存储和授权。使用 JDBC 访问数据库能够使得用户的信息能够持久化，及时应用重启依旧有效，另外更能代表现实世界中 Spring Security 的使用。

配置位于数据库上的认证存储

这个练习的第一部分是建立一个基于 Java 的关系数据库 HyperSQL DB（或简写为 HSQL）示例，并装入 Spring Security 默认的 schema。我们将会通过使用 Spring Security 的嵌入式数据库配置功能来设置 HSQL 在内存中运行，比起手动的安装数据库，这是一个很简单的配置方法。

请记住在这个例子中（包括本书的其余部分），我们将使用 HSQL，主要是因为它很容易安装。在使用这些例子的过程中，我们鼓励修改这个配置以使用你喜欢的数据库。鉴于我们不想让本书的这一部分过多关注于复杂数据库的安装，对练习来说我们选择了更简便而不是更接近现实。

创建 Spring Security 默认的 schema

我们提供了一个 SQL 文件（security-schema.sql），它将原来创建 Spring Security 使用 HSQL 所依赖的所有表。如果你使用自己的数据库实例，你可能会需要调整 schema 的定义语法来适应特定的数据库。我们会将 SQL 文件置于 classpath 中，在 WEB-INF/classes 目录下。

配置 HSQL 嵌入式数据库

要配置 HSQL 嵌入式的数据库，我们需要修改 dogstore-security.xml 文件，以实现启动数据库并运行 SQL 来创建 Spring Security 表结构。首先，我们将会在文件的顶部添加对 jdbc XML 模式的应用：

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

```
http://www.springframework.org/schema/security  
http://www.springframework.org/schema/security/  
    spring-security3.0.xsd"  
>
```

接下来，我们声明`<embedded-database>`元素，以及对 SQL 脚本的引用：

```
<jdbc:embedded-database id="dataSource" type="HSQL">  
    <jdbc:script location="classpath:security-schema.sql"/>  
</jdbc:embedded-database>
```

如果此时重启服务，你可以在日志上看到初始化 HSQL 数据库。需要记住的是`<embedded-database>`只会在内存中创建数据库，所以你在内存中看不到任何东西，也不能使用标准的工具进行查询。

配置 JdbcDaoImpl 凭证存储

我们需要修改 `dogstore-security.xml` 文件来声明正在使用 JDBC 的 `UserDetailsService` 实现，替换我们在第二章：*Spring Security 起步*和第三章：*增强用户体验*中配置的 Spring Security 内存 `UserDetailsService` 实现。这通过一个对`<authentication-manager>`声明的一个简单改变来实现：

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

【`data-source-ref` 引用了我们在上一步声明`<embedded-database>`时定义的 bean。】

添加用户定义到 schema 中

最后，我们要创建另外一个 SQL 文件，它将会在内存数据库创建时执行。这个 SQL 文件将会包含默认用户（admin 和 guest）的信息，以及 `GrantedAuthority` 设置（这一点我们在前一章中已经用过了）。我们将这个文件命名为 `test-data.sql`，并将其与 `security-schema.sql` 一起放在 `WEB-INF/classes` 下：

```
insert into users(username, password, enabled) values  
    ('admin','admin',true);  
insert into authorities(username,authority) values  
    ('admin','ROLE_USER');  
insert into authorities(username,authority) values  
    ('admin','ROLE_ADMIN');  
insert into users(username, password, enabled) values  
    ('guest','guest',true);  
insert into authorities(username,authority) values  
    ('guest','ROLE_USER');
```

commit;

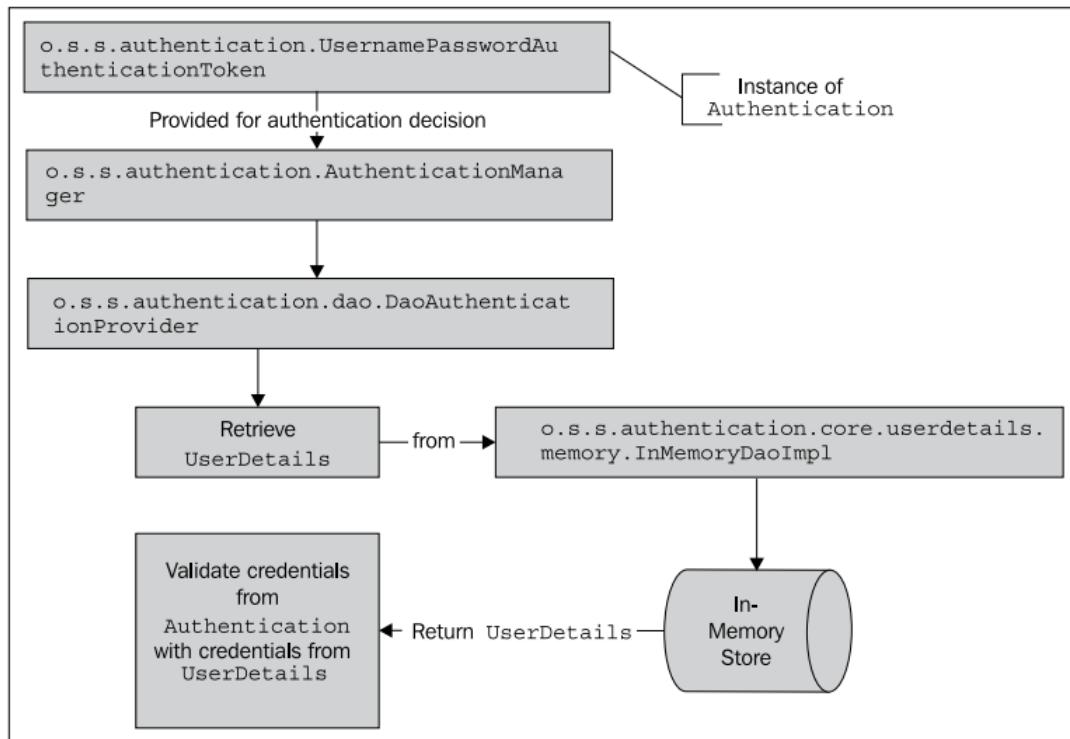
接下来，我们需要添加这个 SQL 文件到嵌入式数据库配置中，它将会在启动时加载：

```
<jdbc:embedded-database id="dataSource" type="HSQL">
    <jdbc:script location="classpath:security-schema.sql"/>
    <jdbc:script location="classpath:test-data.sql"/>
</jdbc:embedded-database>
```

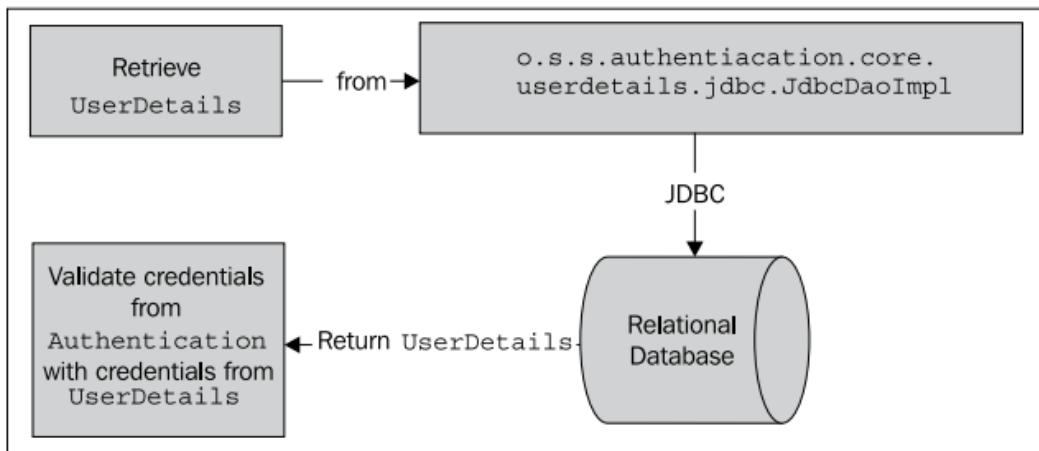
在 SQL 添加到数据库配置后，我们应该能够启动应用并登录。Spring Security 现在已经查找数据库的认证和 GrantedAuthority 信息。

基于数据库后台的认证是如何实现的

你可能会记起在第二章中讲述的认证过程，AuthenticationManager 委托 AuthenticationProvider 来校验安全实体的凭证信息以确定它是否能够访问系统。我们在第二章和第三章中使用的 AuthenticationProvider 为 DaoAuthenticationProvider。这个 provider 又委托了一个 UserDetailsService 的实现来从凭证库中检索和校验安全实体的信息。我们能够通过以下的图来反应第二章的过程：



正如你可能预期的那样，数据库后台的认证存储和内存存储的唯一区别在于 UserDetailsService 的实现类。`o.s.s.core.userdetails.jdbc.JdbcDaoImpl` 类提供了 UserDetailsService 的实现。不同于在内存中（通过 Spring Security 的配置文件添加）查找，`JdbcDaoImpl` 在数据库中查找用户。



你可能意识到我们根本没有引用这个实现类。这是因为在更新后的 Spring Security 配置中 <jdbc-user-service> 声明会自动配置 JdbcDaoImpl 并将其织入到 AuthenticationProvider 中。在本章接下类的内容中，我们将会介绍如何配置 Spring Security 使用我们自定义的 JdbcDaoImpl 实现，它继续包含了修改密码功能（在第三章中我们添加到 InMemoryDaoImpl 了）。让我们看一下如何实现自定义的支持修改密码功能的 JdbcDaoImpl 子类配置。

实现自定义的 JDBC UserDetailsService

正如在前面章节中的那个练习，我们将以基本的 JdbcDaoImpl 作为起点，将其进行扩展以支持修改密码功能。

创建一个自定义的 JDBC UserDetailsService

在 com.packtpub.springsecurity.security 包下创建如下的类：

```
public class CustomJdbcDaoImpl extends JdbcDaoImpl implements  
IChangePassword {  
    public void changePassword(String username, String password) {  
        getJdbcTemplate()  
            .update("UPDATE USERS SET PASSWORD = ? WHERE USERNAME = ?"  
                  , password, username);  
    }  
}
```

你可以看到这个类扩展了 JdbcDaoImpl 默认类，提供了按照用户请求更新数据库中密码的功能。我们使用标准的 Spring JDBC 完成这个功能。

为自定义的 JDBC UserDetailsService 添加 Spring bean 声明

在 dogstore-base.xml 配置文件中，添加如下的 Spring Bean 声明：

```
<bean id="jdbcUserService"
```

```
class="com.packtpub.springsecurity.security.CustomJdbcDaoImpl">
<property name="dataSource" ref="dataSource"/>
</bean>
```

同样的，`dataSource` 的 Bean 引用指向了`<embedded-database>`声明，我们使用这个声明来安装 HSQL 内存数据库。

你会发现自定义的 `UserDetailsService` 允许我们与数据库直接交互。在接下来的例子中，我们将使用这个功能来扩展 `UserDetailsService` 的基本功能。在使用 Spring Security 的复杂应用中，这种类型的个性化是很常见的。

基于 JDBC 的内置用户管理

正如上面简单 `JdbcDaoImpl` 扩展所描述的那样，开发人员可能会扩展这个类，但同时也会保留基本的功能。而我们要实现更复杂功能时，如用户注册（online store 所必须的）与用户管理功能、站点的管理员创建用户、更新密码等，又会怎样呢？

尽管这些功能借助 JDBC 语句都能相对容易的实现，但是 Spring Security 还是为我们提供了内置的功能以支持对数据库里的用户进行创建、读取、更新和删除的操作。这对简单的系统来说是很有用的，同时也为构建自定义需求的用户提供了很好的起点。

实现类 `o.s.s.provisioning.JdbcUserDetailsService` 扩展了 `JdbcDaoImpl` 的功能，提供了一些很有用的与用户相关的方法，这些方法的一部分在 `o.s.s.provisioning.UserDetailsService` 接口中进行了定义：

方法	描述
<code>void createUser(UserDetails user)</code>	根据给定的 <code>UserDetails</code> 创建一个新用户，并包含所有声明的 <code>GrantedAuthority</code> 。
<code>void updateUser(final UserDetails user)</code>	根据给定的 <code>UserDetails</code> 更新一个用户。更新其 <code>GrantedAuthority</code> 并将其从用户缓存中清除。
<code>void deleteUser(String username)</code>	根据给定的用户名删除用户，并将其从用户缓存中清除。
<code>boolean userExists(String username)</code>	根据给定的用户名判断用户是否存在（不管是否可用）。
<code>void changePassword(String oldPassword, String newPassword)</code>	修改当前登录用户的密码。为了使得操作成功，用户必须提供正确的当前密码。

正如你所见，`JdbcUserDetailsService` 的 `changePassword` 方法正好弥补了我们 `CustomJdbcDaoImpl` 的不足——在修改之前，它会检验用户已存在密码。让我们看一下将 `CustomJdbcDaoImpl` 替换为 `JdbcUserDetailsService` 需要怎样的配置步骤。

首先，我们需要在 `dogstore-base.xml` 中声明 `JdbcUserDetailsService` bean：

```
<bean id="jdbcUserService"
      class="org.springframework.security
```

```
.provisioning.JdbcUserDetailsManager">
<property name="dataSource" ref="dataSource"/>
<property name="authenticationManager"
          ref="authenticationManager"/>
</bean>
```

对 `AuthenticationManager` 的引用要匹配我们之前 `dogstore-security.xml` 文件中声明的 `<authentication-manager>` 的 alias。不要忘记注释掉 `CustomJdbcDaoImpl` 的声明——我们暂时不会使用它。

接下来，我们需要对 `changePassword.jsp` 做一些小的调整：

```
<h1>Change Password</h1>
<form method="post">
    <label for="oldpassword">Old Password</label>:
    <input id="oldpassword" name="oldpassword"
           size="20" maxlength="50" type="password"/>
    <br />
    <label for="password">New Password</label>:
    <input id="password" name="password" size="20"
           maxlength="50" type="password"/>
    <br />
```

最后，需要简单调整 `AccountController`。将 `@Autowired` 引用 `IChangePassword` 的实现替换为：

```
@Autowired
private UserDetailsManager userDetailsService;
```

`submitChangePasswordPage` 方法也会更加简单了，因为要依赖的当前用户信息将会由 `JdbcUserDetailsManager` 为我们确定：

```
public String submitChangePasswordPage(@RequestParam("oldpassword")
                                         String oldPassword,
                                         @RequestParam("password") String newPassword) {
    userDetailsService.changePassword(oldPassword, newPassword);
    SecurityContextHolder.clearContext();
    return "redirect:home.do";
}
```

在这些修改完成后，你可以重启应用并尝试新的修改密码功能。

注意当你没有提供正确的密码时将会发生什么。试想一下会发生什么？并尝试思考怎样调整能使得对用户更友好。

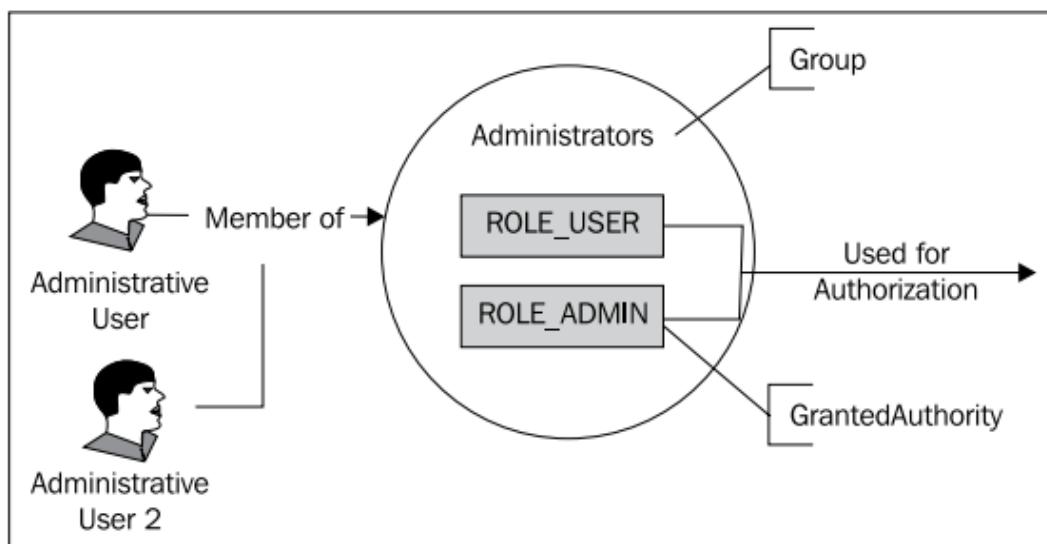
尽管我们没有阐述 `JdbcUserDetailsManager` 提供的所有功能，但是可以看出它能很容易与简单的 JSP 页面结合在一起（当然要进行适当授权）以允许管理员来管理站点的用户——这对

产品级别的应用是必要的。

JdbcDaoImpl 的高级配置

JdbcDaoImpl 拥有众多的可配置选项使其可以在已存在的 schema 中使用，或对其功能进行更复杂地调整。在很多场景下，很可能我们只需调整内置 UserDetailsService 类的配置而不需要写自己的代码。

有一个很重要的功能就是在用户（User）和权限（GrantedAuthority）之间添加一个隔离层（a level of indirection——找不到更好的译法了），这通过将 GrantedAuthority 按照逻辑划分成集合即组（group）来实现。用户能够被分配到一个或多个组，而组的成员被赋予了一系列的 GrantedAuthority 声明。



正如在图中所描述的那样，中间的隔离层使得我们可以将相同集合的角色分派给很多人，而这只需要指定新用户到存在的组中即可。将这与我们之前的做法对比，在以前的做法是将 GrantedAuthority 直接分配给单个的用户。

这种将权限进行打包处理的方式可能在以下的场景中用到：

- 要将用户分成不同的组，而组之间有些角色是重叠的；
- 想要全局地修改一类用户的权限。如，如果你拥有一个“供应商”的分组，而你想要修改他们能否访问应用特定区域的设置；
- 拥有大量的用户，你不需要用户级别的授权配置。

除非你的应用用户量很有限，否则很可能要使用基于组的访问控制。这种管理方式的简便性和扩展性带来的价值远远超过了它稍微增加的复杂性。这种将用户权限集中到组中的技术通常叫做基于组的访问控制（Group-Based Access Control，GBAC）。

【基于组的访问控制几乎在市面上任何安全的操作系统和软件包中都能看到。微软的活动目录（Active Directory，AD）是大范围使用 GBAC 的典型实现，它把 AD 的用户纳入组中并给组授予权限。通过使用 GBAC，能够指数级得简化对大量基于 AD 组织的权限管理。想一下

你所使用软件的安全功能——用户、分组以及权限是如何管理的？这种方式编写安全功能的利弊是什么？】

让我们对 JBCP Pets 添加一层抽象，并将基于组的授权理念应用于这个站点。

配置基于组的授权

我们会为站点添加两个组——普通用户（我们将其称为“Users”）和管理员（我们将其称为“Administrators”）。通过修改用于启动数据库的 SQL 脚本，将已经存在的 guest 和 admin 账号分配到合适的组中。

配置 JdbcDaoImpl 以使用用户组

首先，我们需要为 JdbcDaoImpl 的自定义实现类设置属性以启用组的功能，并关闭对用户直接授权的功能。在 dogstore-base.xml 中添加如下的 bean 声明：

```
<bean id="jdbcUserService"
      class="com.packtpub.springsecurity.security.CustomJdbcDaoImpl">
    <property name="dataSource" ref="dataSource"/>
    <property name="enableGroups" value="true"/>
    <property name="enableAuthorities" value="false"/>
</bean>
```

注意的是，如果你一直跟着我们的例子在做，并且使用了 JdbcUserManager 的代码和配置，请对其进行修改，因为在本章的剩余部分我们将使用 CustomJdbcDaoImpl。

修改初始载入的 SQL 脚本

我们需要简单修改构建数据库的 SQL 语句：

- 定义我们的组信息；
- 指定 GrantedAuthority 声明到组中；
- 指定用户到组中。

简单起见，我们声明一个名为 test-users-groups-data.sql 的新 SQL 脚本。

首先，添加组：

```
insert into groups(group_name) values ('Users');
insert into groups(group_name) values ('Administrators');
```

接下来，指定角色到组中：

```
insert into group_authorities(group_id, authority) select id,'ROLE_
USER' from groups where group_name='Users';
insert into group_authorities(group_id, authority) select id,'ROLE_
```

```
USER' from groups where group_name='Administrators';
insert into groupAuthorities(group_id, authority) select id,'ROLE_
ADMIN' from groups where group_name='Administrators';
```

接下来，创建用户：

```
insert into users(username, password, enabled) values
('admin','admin',true);
insert into users(username, password, enabled) values
('guest','guest',true);
```

最后，指定用户到组中：

```
insert into groupMembers(group_id, username) select id,'guest' from
groups where group_name='Users';
insert into groupMembers(group_id, username) select id,'admin' from
groups where group_name='Administrators';
```

修改嵌入式的数据库创建声明

我们需要更新嵌入式 HSQL 数据库的创建配置指向这个脚本，而不是已经存在的 `test-data.sql` 脚本：

```
<jdbc:embedded-database id="dataSource" type="HSQL">
  <jdbc:script location="classpath:security-schema.sql"/>
  <jdbc:script location="classpath:test-users-groups-data.sql"/>
</jdbc:embedded-database>
```

要注意的是，`security-schema.sql` 脚本已经包含了支持组功能的表声明，所以我们不需要修改这个脚本了。

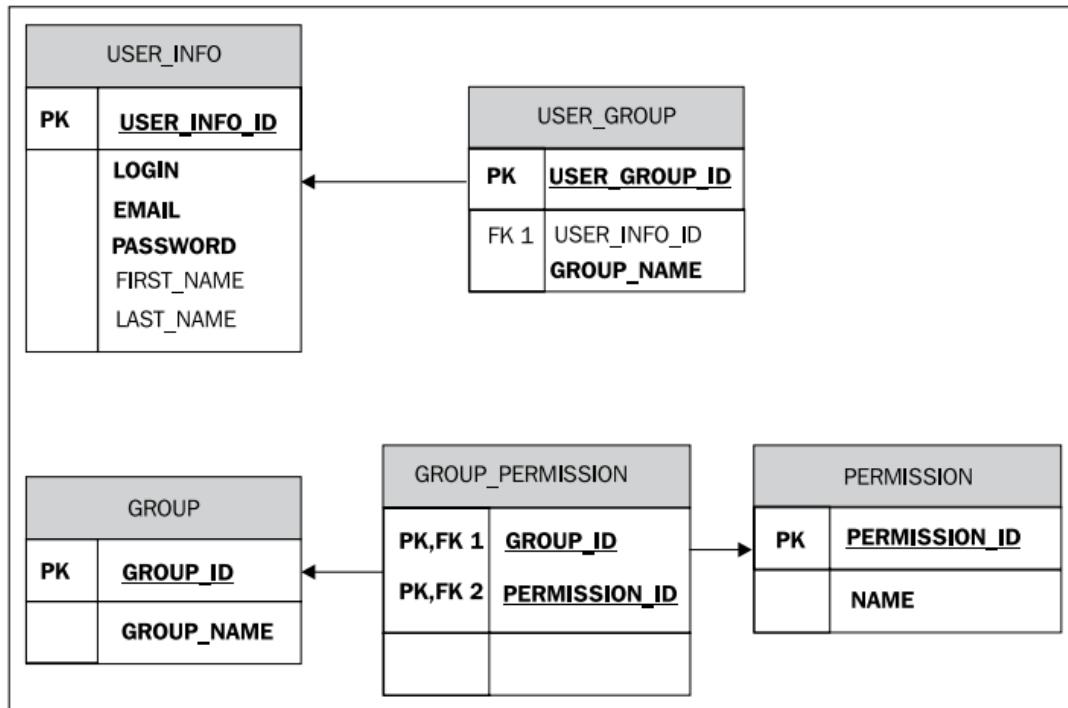
到这里，你可以重启 JBCP Pets 站点，它将与以前的表现完全一致，但是，我们在用户和权限间添加的抽象层使得我们能够更容易地开发复杂的用户管理功能。

让我们暂时离开 JBCP Pets 的场景，了解在这个方面上一个更为重要的配置。

使用遗留的或用户自定义的 `schema` 实现基于数据库的认证

通常来说，Spring Security 的新用户可能需要适配用户、组和角色到已有的数据库 schema 中。尽管遗留的数据库并不匹配 Spring Security 要求的数据库 schema，但我们还是可以通过配置 `JdbcDaoImpl` 来匹配它。

假设我们拥有一个如下图所示的遗留数据库 schema，要基于它实现 Spring Security：



我们能够很容易地修改 `JdbcDaoImpl` 的配置来使用这个 schema 并重写我们在 JBCP Pets 中使用的默认 Spring Security 表定义和列。

确定正确的 JDBC SQL 查询

`JdbcDaoImpl` 有三个 SQL 查询，它们有定义良好的参数和返回列的集合。我们必须机遇它们提供的功能，确定每个查询的 SQL。`JdbcDaoImpl` 的每个 SQL 查询都是使用登录时提供的用户名作为唯一的参数。

查询名	描述	期望得到的 SQL 列
<code>usersByUsernameQuery</code>	返回匹配用户名的一个或更多的用户。只有返回的第一个用户被使用。	<code>Username (string)</code> <code>Password (string)</code> <code>Enabled (Boolean)</code>
<code>authoritiesByUsernameQuery</code>	返回用户被直接授予的权限。一般在 GBAC 禁用时，被使用。	<code>Username (string)</code> <code>Granted Authority (string)</code>
<code>groupAuthoritiesByUsernameQuery</code>	返回用户作为组成员被授予的权限和组的详细信息。在 GBAC 功能启用时，被使用。	<code>Group Primary Key (any)</code> <code>Group Name (any)</code> <code>Granted Authority (string)</code>

要注意的是，在一些场景中返回的列在默认的 `JdbcDaoImpl` 实现中并没有用到，但我们依旧需要将这些值返回。在进入下一章节前，请花费一点时间尝试写一下基于前面数据库图表中的查询语句。

配置 JdbcDaoImpl 来使用自定义的 SQL 查询

给不规范的数据库使用自定义 SQL 查询，我们需要在 Spring Bean 的配置文件中修改 JdbcDaoImpl 的属性。要注意的一点是，为了给 JdbcDaoImpl 配置 JDBC 查询，我们不能使用 <jdbc-user-service> 声明。必要要明确实例化这个 bean，如同我们在自定义 JdbcDaoImpl 实现时所作的那样：

```
<bean id="jdbcUserService"
      class="com.packtpub.springsecurity.security.CustomJdbcDaoImpl">
    <property name="dataSource" ref="dataSource"/>
    <property name="enableGroups" value="true"/>
    <property name="enableAuthorities" value="false"/>
    <property name="usersByUsernameQuery">
        <value>SELECT LOGIN, PASSWORD,
                  1 FROM USER_INFO WHERE LOGIN = ?
        </value>
    </property>
    <property name="groupAuthoritiesByUsernameQuery">
        <value>SELECT G.GROUP_ID, G.GROUP_NAME, P.NAME
                  FROM USER_INFO U
                  JOIN USER_GROUP UG on U.USER_INFO_ID = UG.USER_INFO_ID
                  JOIN GROUP G ON UG.GROUP_ID = G.GROUP_ID
                  JOIN GROUP_PERMISSION GP ON G.GROUP_ID = GP.GROUP_ID
                  JOIN PERMISSION P ON GP.PERMISSION_ID = P.PERMISSION_ID
                  WHERE U.LOGIN = ?
        </value>
    </property>
</bean>
```

这是 Spring Security 从已存在且不符合默认 schema 的数据库中读取设置时，唯一需要配置的地方。需要记住的是，在使用已存在的 schema 时，通常会需要扩展 JdbcDaoImpl 以支持修改密码、重命名用户账号以及其他用户管理功能。

如果你使用 JdbcUserDetailsManager 来完成用户管理的任务，这个类使用了大约 20 个可配置的 SQL 查询。请参考 Javadoc 或源码来了解 JdbcUserDetailsManager 使用的默认查询。

配置安全的密码

我们回忆第一章：一个不安全应用的剖析中，审计人员认为密码以明文形式进行存储是最高优先级的安全风险。实际上，在任何安全系统中，密码安全都是保证已经过认证的安全实体是真实可靠的重要方面。安全系统的设计人员必须保证密码存储时，任何恶意的用户想要进行破解都是非常困难的。

在数据库存储时，需要遵守以下的准则：

- 密码不能以明文的形式进行存储（简单文本）；
- 用户提供的密码必须与数据库存储的密码进行比较；

- 密码不能应用户的请求提供（即使用户忘记了密码）。

对于大多数应用来说，为了满足以上要求最适合的方式是对密码进行单向的编码或加密。单向编码提供了安全和唯一的特性，这对于正确的认证用户非常重要，它能够保证密码一旦被加密就不能再被解密了。

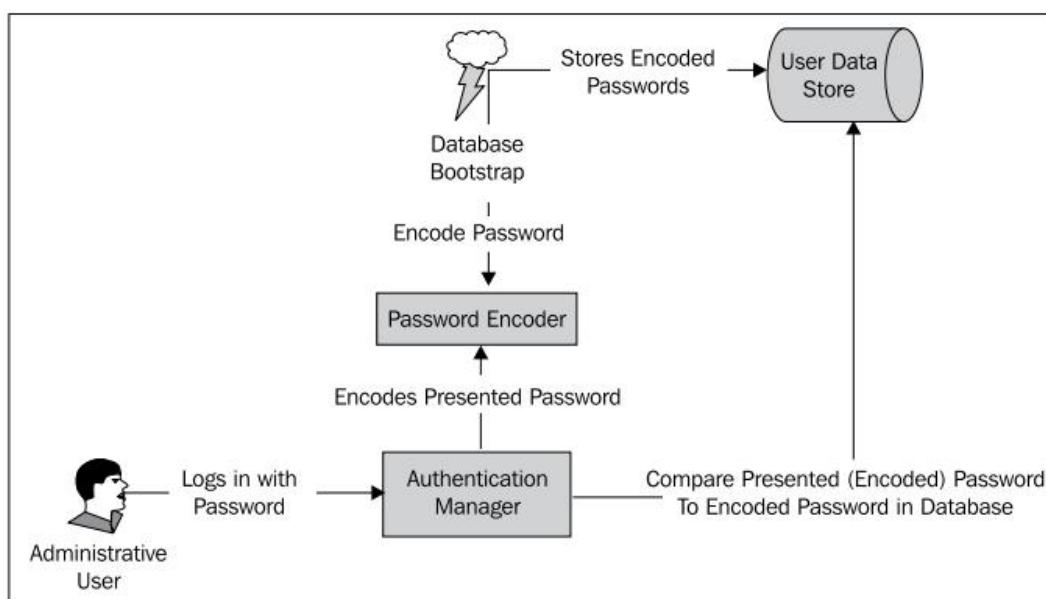
在大多数的安全应用设计中，一般没有必要和实际要求根据用户的请求检索用户的密码，因为如果没有附加的安全认证，提供用户的实际密码会有很大的安全风险。作为替代方案，大多数的应用为用户提供了重设密码的功能，要么需要提供额外的认证信息（如社会保险号码、生日、缴税 ID 或其它个人信息），要么通过一个基于 email 的系统。

【存储其它类型的敏感信息：以下的准则可以应用于密码以及其它类型的敏感信息，包括社会保险号以及信用卡信息（尽管基于应用的不同，它们中的一些需要解密的能力）。比较常见的是数据库以多种形式来存储这些敏感信息，比如用户 16 个数字的信用卡数字可以用一种高度加密的形式存储，但是最后的四位数字以明文的形式存储（作为佐证，可以回忆一些商业站点会显示 XXXX XXXX XXXX 1234 来帮助你分别已存储的信用卡）。】

基于我们（实际上并不太符合现实）使用 SQL 来访问 HSQL 数据库中用户的环境，你可能已经在思考如何对密码进行编码。HSQL 以及其它大多数的数据库并没有提供加密方法作为数据库的内置功能。

一般来说，bootstrap 过程（为系统添加初始的用户和数据）会联合使用一些 SQL 加载和 Java 代码。根据应用的复杂性，这个过程也可能会变得很复杂。

对于 JBCP Pets 应用来说，我们将会继续使用嵌入式数据库声明和对应的 SQL，并且会添加一点 Java 代码在初始化加载后执行来加密数据库中的所有密码。为了使得密码加密能够正常工作，两个过程必须同步的使用密码加密以确保密码能够被一致的处理和校验。



在 Spring Security 中，密码加密已经进行了封装，通过 o.s.s.authentication.encoding.PasswordEncoder 接口的实现类来定义。通过使用 <authentication-provider> 元素里的 <password-encoder> 声明我们能够很容易地配置密码编码：

```
<authentication-manager alias="authenticationManager">
    <authentication-provider user-service-ref="jdbcUserService">
        <password-encoder hash="sha"/>
    </authentication-provider>
```

</authentication-manager>

你可能会很高兴的了解到 Spring Security 已经提供了一系列 PasswordEncoder 的实现，它们可以用于不同的需求和安全需要。要使用哪个实现可以通过<password-encoder>元素的 hash 属性来指定。

以下的列表列出了内置的实现类以及它们的优点。这些实现类都在 o.s.s.authentication.Encoding 包下。

实现类	描述	hash 值
PlaintextPasswordEncoder	以明文的形式编码。DaoAuthenticationProvider 默认的密码编码器。	plaintext
Md4PasswordEncoder	PasswordEncoder 使用 MD4 hash 算法。MD4 并不是一个安全的算法——不推荐使用这个编码器	md4
Md5PasswordEncoder	PasswordEncoder 使用 MD5 的单向编码算法。	md5
ShaPasswordEncoder	PasswordEncoder 使用 SHA 单向加密算法。这个编码器支持配置密码的强度级别。	sha sha-256
LdapShaPasswordEncoder	在集成 LDAP 认证存储时使用，实现了 LDAP SHA 和 LDAP SSHA 算法。我们将会在第九章：LDAP 目录服务讲述 LDAP 时，学习更多关于这个算法的知识。	{sha} {ssha}

与 Spring Security 其他领域一样，可以引用一个 PasswordEncoder 的实现类以提供更精确的配置，并允许 PasswordEncoder 通过依赖注入织入到其它的 bean 中。对于 JBCP Pets 来说，我们需要使用这个 bean 引用的方法来编码用户的初始数据。

让我们了解一下为 JBCP Pet 应用配置基本密码编码的过程。

配置密码编码

配置基本的密码编码涉及到两个地方——在我们的 SQL 脚本执行后，加密载入数据库中数据的密码，并确保 DaoAuthenticationProvider 被配置成使用 PasswordEncoder。

配置 PasswordEncoder

首先，作为通常的 Spring bean，声明一个 PasswordEncoder 的实例

```
<bean class="org.springframework.security.authentication.  
encoding ShaPasswordEncoder" id="passwordEncoder"/>
```

你会发现我们使用了 SHA-1 的 PasswordEncoder 实现。这是一个高效的单向加密算法，在密码存储中经常用到。

配置 AuthenticationProvider

我们需要配置 DaoAuthenticationProvider 来持有一个对 PasswordEncoder 的引用，这样它就可以在用户登录时，编码并比较用户提供的密码。添加<password-encoder>声明并指向我们在前面定义的 bean 的 ID:

```
<authentication-manager alias="authenticationManager">  
  <authentication-provider user-service-ref="jdbcUserService">
```

```
<password-encoder ref="passwordEncoder"/>
</authentication-provider>
</authentication-manager>
```

如果在此时重启应用，并尝试登录，你会发现前面合法的登录凭证现在都会被拒绝。这是因为数据库中存储的密码（在启动时通过 `test-users-groups-data.sql` 脚本加载的）并没有以加密的形式存储。我们需要用一些 java 代码对启动数据进行后置的处理。

编写数据库启动的密码编码器

我们对 SQL 加载的数据进行编码的方式是使用了 Spring bean 的初始化方法，它将在 `embedded-database` bean 实例化完成后执行。这个 bean，`com.packtpub.springsecurity.security.DatabasePasswordSecurerBean` 很简单：

```
public class DatabasePasswordSecurerBean extends JdbcDaoSupport {
    @Autowired
    private PasswordEncoder passwordEncoder;

    public void secureDatabase() {
        getJdbcTemplate().query("select username, password from users",
                               new RowCallbackHandler(){
                                   @Override
                                   public void processRow(ResultSet rs) throws SQLException {
                                       String username = rs.getString(1);
                                       String password = rs.getString(2);
                                       String encodedPassword =
                                           passwordEncoder.encodePassword(password, null);
                                       getJdbcTemplate().update("update users set password = ?
                                                               where username = ?",
                                                               encodedPassword, username);
                                       logger.debug("Updating password for username:
                                                               "+username+" to: "+encodedPassword);
                                   }
                               });
    }
}
```

代码使用了 `JdbcTemplate` 功能来遍历所有的数据库中所有的用户并使用注入的 `PasswordEncoder` 引用对密码进行编码。每一个密码都进行了更新。

配置启动的密码编码

我们需要配置这个 Spring bean 的声明以使其在 web 应用启动时及 `<embedded-database>` 初始化后再进行该类的初始化。Spring bean 的依赖跟踪机制保证 `DatabasePasswordSecurerBean` 能够在合适的时机执行：

```
<bean class="com.packtpub.springsecurity.security.  
        DatabasePasswordSecurerBean"  
        init-method="secureDatabase" depends-on="dataSource">  
    <property name="dataSource" ref="dataSource"/>  
</bean>
```

如果你此时重启 JBCP Pets 应用，你会发现数据库中的密码已经进行了编码，登录功能可以正常使用了。

你是否愿意在密码上添加点 salt？

如果安全审计人员检查数据库中编码过的密码，在网站安全方面，他可能还会找到一些令其感到担心的地方。让我们查看一下存储的 admin 和 guest 用户的用户名和密码值：

用户名	明文密码	加密密码
admin	admin	7b2e9f54cdff413fcde01f330af6896c3cd7e6cd
guest	guest	2ac15cab107096305d0274cd4eb86c74bb35a4b4

这看起来很安全——加密后的密码与初始的密码看不出有任何相似性。但是如果我们添加一个新的用户，而他碰巧和 admin 用户拥有同样的密码时，又会怎样呢？

用户名	明文密码	加密密码
fakeadmin	admin	7b2e9f54cdff413fcde01f330af6896c3cd7e6cd

现在，注意 fakeadmin 用户加密过后密码与 admin 用户完全一致。所以一个黑客如果能够读取到数据库中加密的密码，就能够对已知的密码加密结果和 admin 账号未知的密码进行对比，并发现它们是一样的。如果黑客能够使用自动化的工具来进行分析，他能够在几个小时内破坏管理员的账号。

【鉴于作者本人使用了一个数据库，它里面的密码使用了完全一致的加密方式，我和工程师团队决定进行一个小的实验并查看明文 password 的 SHA-1 加密值。当我们得到 password 的加密形式并进行数据库查询来查看有多少人使用这个相当不安全的密码。让我们感到非常吃惊的是，这样的人有很多甚至包括组织的一个副总。每个用户都收到了一封邮件提示他们选择难以猜到的密码有什么好处，另外开发人员迅速的使用了一种更安全的密码加密机制。】

请回忆一下我们在第三章中提到的彩虹表技术，恶意的用户如果能够访问到数据库就能使用这个技术来确定用户的密码。这些（以及其它的）黑客技术都是使用了哈希算法的结果都是确定的这一特点——即相同的输入必然会产生相同的输出，所以攻击者如果尝试足够的输入，他们可能会基于已知的输入匹配到未知的输出。

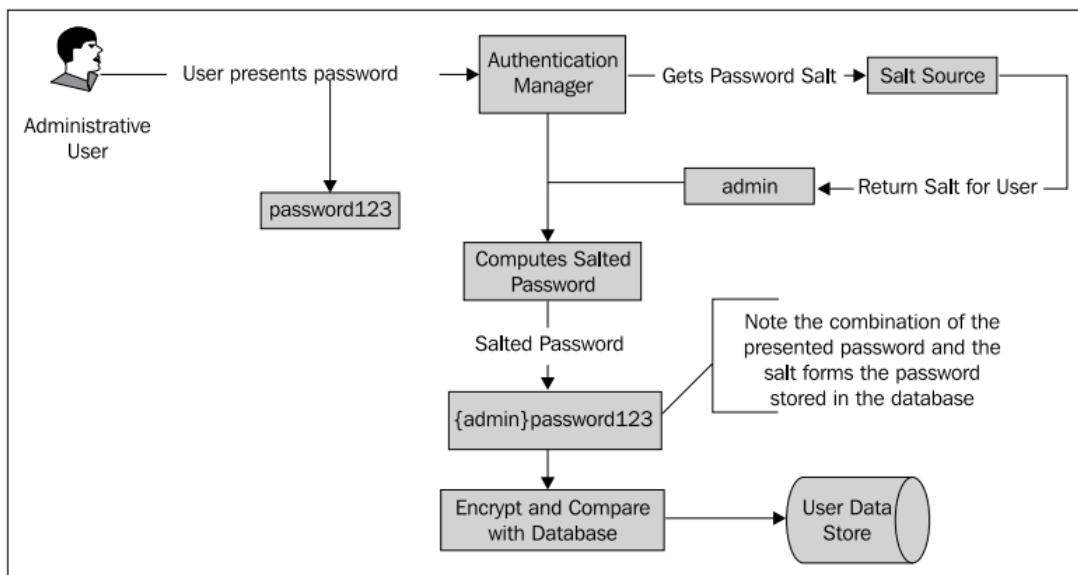
一种通用且高效的方法来添加安全层加密密码就是包含 salt（这个单词就是盐的意思，但为了防止直译过来反而不好理解，这里直接使用这个单词——译者注）。Salt 是第二个明文组件，它将与前面提到的明文密码一起进行加密以保证使用两个因素来生成（以及进行比较）加密的密码值。选择适当的 salt 能够保证两个密码不会有相同的编码值，因此可以打消安全审计人员的顾虑，同时能够避免很多常见类型的密码暴力破解技术。

比较好的使用 salt 的实践不外乎以下的两种类型：

- 使用与用户相关的数据按算法来生成——如，用户创建的时间；

- 随机生成的，并且与用户的密码一起按照某种形式进行存储（明文或者双向加密）。（所谓的双向加密 two-way encrypte，指的是加密后还可以进行解密的方式——译者注）

如下图就展现了一个简单的例子，在例子中 salt 与用户的登录名一致：



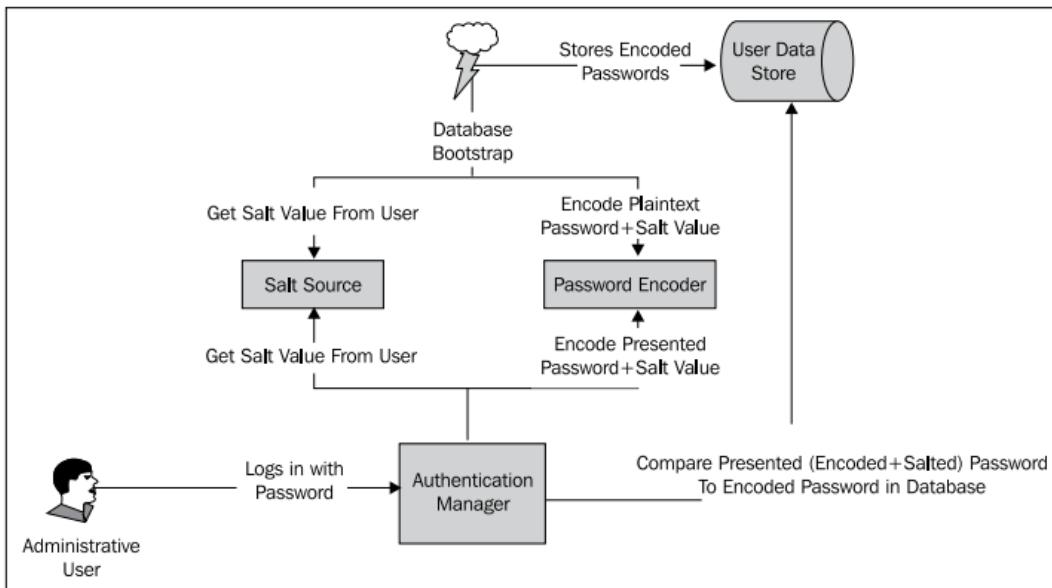
【需要记住的是 salt 被添加到明文的密码上，所以 salt 不能进行单向的加密，因为应用要查找用户对应的 salt 值以完成对用户的认证。】

Spring Security 为我们提供了一个接口 `o.s.s.authentication.dao.SaltSource`，它定义了一个方法根据 `UserDetails` 来返回 salt 值，并提供了两个内置的实现：

- `SystemWideSaltSource` 为所有的密码定义了一个静态的 salt 值。这与不使用 salt 的密码相比并没有提高多少安全性；
- `ReflectionSaltSource` 使用 `UserDetails` 对象的一个 bean 属性得到用户密码的 salt 值。鉴于 salt 值应该能够根据用户数据得到或者与用户数据一起存储，`ReflectionSaltSource` 作为内置的实现被广泛使用。

配置 salted 密码

与前面配置简单密码加密的练习类似，添加支持 salted 密码的功能也需要修改启动代码和 `DaoAuthenticationProvider`。我们可以通过查看以下的图来了解 salted 密码的流程是如何改变启动和认证的，本书的前面章节中我们见过与之类似的图：



让我们通过配置 `ReflectionSaltSource` 实现 salt 密码，增加密码安全的等级。

声明 SaltSource Spring bean

在 `dogstore-base.xml` 文件中，增加我们使用的 `SaltSource` 实现的 bean 声明：

```
<bean class="org.springframework.security.authentication.dao.ReflectionSaltSource" id="saltSource">
    <property name="userPropertyToUse" value="username"/>
</bean>
```

我们配置 salt source 使用了 `username` 属性，这只是一个暂时的实现，在后面的练习中将会进行修正。你能否想到这为什么不是一个好的 salt 值吗？

将 SaltSource 织入到 PasswordEncoder 中

我们需要将 `SaltSource` 织入到 `PasswordEncoder` 中，以使得用户在登录时提供的凭证信息能够在与存储值进行比较前，被适当的 salted。这通过在 `dogstore-security.xml` 文件中添加一个新的声明来完成：

```
<authentication-manager alias="authenticationManager">
    <authentication-provider user-service-ref="jdbcUserService">
        <password-encoder ref="passwordEncoder">
            <salt-source ref="saltSource"/>
        </password-encoder>
    </authentication-provider>
</authentication-manager>
```

你如果在此时重启应用，你不能登录成功。正如在前面练习中的那样，数据库启动时的密码编码器需要进行修改以包含 `SaltSource`。

增强 DatabasePasswordSecurerBean

与 `UserDetailsService` 引用类似，我们需要为 `DatabasePasswordSecurerBean` 添加对另一个 bean 的引用（即 `SaltSource`——译者注），这样我们就能够为用户得到合适的密码 salt：

```
public class DatabasePasswordSecurerBean extends JdbcDaoSupport {  
    @Autowired  
    private PasswordEncoder passwordEncoder;  
    @Autowired  
    private SaltSource saltSource;  
    @Autowired  
    private UserDetailsService userDetailsService;  
  
    public void secureDatabase() {  
        getJdbcTemplate().query("select username, password from users",  
                               new RowCallbackHandler(){  
                                   @Override  
                                   public void processRow(ResultSet rs) throws SQLException {  
                                       String username = rs.getString(1);  
                                       String password = rs.getString(2);  
                                       UserDetails user =  
                                           userDetailsService.loadUserByUsername(username);  
                                       String encodedPassword =  
                                           passwordEncoder.encodePassword(password,  
                                             saltSource.getSalt(user));  
                                       getJdbcTemplate().update("update users set password = ?  
                                         where username = ?",  
                                         encodedPassword,  
                                         username);  
                                       logger.debug("Updating password for username:  
                                         "+username+" to: "+encodedPassword);  
                                   }  
                               });  
    }  
}
```

回忆一下，`SaltSource` 是要依赖 `UserDetails` 对象来生成 salt 值的。在这里，我们没有数据库行对应 `UserDetails` 对象，所以需要请求 `UserDetailsService`（我们的 `CustomJdbcDaoImpl`）的 SQL 查询以根据用户名查找 `UserDetails`。

到这里，我们能够启动应用并正常登录系统了。如果你添加了一个新用户并使用相同的密码（如 `admin`）到启动的数据库脚本中，你会发现为这个用户生成的密码是不一样的，因为我们使用用户名对密码进行了 salt。即使恶意用户能够从数据库中访问密码，这也使得密码更加安全了。但是，你可能会想为什么使用用户名不是最安全的可选 salt——我们将会在稍后的一个练习中进行介绍。

增强修改密码功能

我们要完成的另外一个很重要的变化是将修改密码功能也使用密码编码器。这与为 CustomJdbcDaoImpl 添加 bean 引用一样简单，并需要 changePassword 做一些代码修改：

```
public class CustomJdbcDaoImpl extends JdbcDaoImpl {  
    @Autowired  
    private PasswordEncoder passwordEncoder;  
    @Autowired  
    private SaltSource saltSource;  
    public void changePassword(String username, String password) {  
        UserDetails user = loadUserByUsername(username);  
        String encodedPassword = passwordEncoder.encodePassword  
            (password, saltSource.getSalt(user));  
        getJdbcTemplate().update(  
            "UPDATE USERS SET PASSWORD = ? WHERE USERNAME = ?"  
            , encodedPassword, username);  
    }  
}
```

这里对 PasswordEncoder 和 SaltSource 的使用保证了用户的密码在修改时，被适当的 salt。比较奇怪的是，JdbcUserDetailsManager 并不支持对 PasswordEncoder 和 SaltSource 的使用，所以如果你使用 JdbcUserDetailsManager 作为基础进行个性化，你需要重写一些代码。

配置自定义的 salt source

我们在第一次配置密码 salt 的时候就提到作为密码 salt，username 是可行的但并不是一个特别合适的选择。原因在于 username 作为 salt 完全在用户的控制下。如果用户能够改变他们的用户名，这就使得恶意的用户可以不断的修改自己的用户名——这样就会重新 salt 他们的密码——从而可能确定如何构建一个伪造的加密密码。

更安全做法是使用 UserDetails 的一个属性，这个属性是系统确定的，用户不可见也不可以修改。我们会为 UserDetails 对象添加一个属性，这个属性在用户创立时被随机设置。这个属性将会作为用户的 salt。

扩展数据库 schema

我们需要 salt 要与用户记录一起保存在数据库中，所以要在默认的 Spring Security 数据库 schema 文件 security-schema.sql 中添加一列：

```
create table users(  
    username varchar_ignorecase(50) not null primary key,  
    password varchar_ignorecase(50) not null,  
    enabled boolean not null,  
    salt varchar_ignorecase(25) not null  
);
```

接下来，添加启动的 salt 值到 test-users-groups-data.sql 脚本中：

```
insert into users(username, password, enabled, salt) values ('admin','admin',true,CAST(RAND()*1000000000 AS varchar));
insert into users(username, password, enabled, salt) values ('guest','guest',true,CAST(RAND()*1000000000 AS varchar));
```

要注意的是，需要用这些新的语句替换原有的 insert 语句。我们选择的 salt 值基于随机数生成——你选择任何随机 salt 都是可以的。

修改 CustomJdbcDaoImpl UserDetails service 配置

与本章前面讲到的自定义数据库模式中的步骤类似，我们需要修改从数据库中查询用户的配置以保证能够获得添加的“salt”列的数据。我们需要修改 dogstore-security.xml 文件中 CustomJdbcDaoImpl 的配置：

```
<beans:bean id="jdbcUserService"
    class="com.packtpub.springsecurity.security.CustomJdbcDaoImpl">
    <beans:property name="dataSource" ref="dataSource"/>
    <beans:property name="enableGroups" value="true"/>
    <beans:property name="enableAuthorities" value="false"/>
    <beans:property name="usersByUsernameQuery">
        <beans:value>select username,password(enabled,
            salt from users where username = ?
        </beans:value>
    </beans:property>
</beans:bean>
```

重写基础的 UserDetails 实现

我们需要一个 UserDetails 的实现，它包含与用户记录一起存储在数据库中的 salt 值。对于我们的要求来说，简单重写 Spring 的标准 User 类就足够了。要记住的是为 salt 添加 getter 和 setter 方法，这样 ReflectionSaltSource 密码 salter 就能够找到正确的属性了。

```
package com.packtpub.springsecurity.security;
// imports
public class SaltedUser extends User {
    private String salt;
    public SaltedUser(String username, String password,
                      boolean enabled,
                      boolean accountNonExpired, boolean credentialsNonExpired,
                      boolean accountNonLocked, List<GrantedAuthority>
                          authorities, String salt) {
        super(username, password, enabled,
              accountNonExpired, credentialsNonExpired,
              accountNonLocked, authorities);
    }
}
```

```
this.salt = salt;
}
public String getSalt() {
    return salt;
}
public void setSalt(String salt) {
    this.salt = salt;
}
}
```

我们扩展了 `UserDetails` 使其包含一个 `salt` 域，如果希望在后台存储用户的额外信息其流程是一样的。扩展 `UserDetails` 对象与实现自定义的 `AuthenticationProvider` 时经常联合使用。我们将在第六章：高级配置和扩展讲解一个这样的例子。

扩展 `CustomJdbcDaoImpl` 功能

我们需要重写 `JdbcDaoImpl` 的一些方法，这些方法负责实例化 `UserDetails` 对象、设置 `User` 的默认值。这发生在从数据库中加载 `User` 并复制 `User` 到 `UserDetailsService` 返回的实例中：

```
public class CustomJdbcDaoImpl extends JdbcDaoImpl {
    public void changePassword(String username, String password) {
        getJdbcTemplate().update(
            "UPDATE USERS SET PASSWORD = ? WHERE USERNAME = ?",
            password, username);
    }
    @Override
    protected UserDetails createUserDetails(String username,
        UserDetails userFromUserQuery,
        List<GrantedAuthority> combinedAuthorities) {
        String returnUsername = userFromUserQuery.getUsername();
        if (!isUsernameBasedPrimaryKey()) {
            returnUsername = username;
        }
        return new SaltedUser(returnUsername,
            userFromUserQuery.getPassword(), userFromUserQuery.isEnabled(),
            true, true, true, combinedAuthorities,
            ((SaltedUser) userFromUserQuery).getSalt());
    }
    @Override
    protected List<UserDetails> loadUsersByUsername(String username) {
        return getJdbcTemplate().
            query(getUsersByUsernameQuery(),
            new String[] {username},
```

```
new RowMapper<UserDetails>() {
    public UserDetails mapRow(ResultSet rs, int rowNum)
        throws SQLException {
        String username = rs.getString(1);
        String password = rs.getString(2);
        boolean enabled = rs.getBoolean(3);
        String salt = rs.getString(4);
        return new SaltedUser(username, password,
            enabled, true, true,
            AuthorityUtils.NO_AUTHORITIES, salt);
    }
});
```

`createUserDetails` 和 `loadUsersByUsername` 重写了父类的方法——与父类不同的地方在代码列表中已经着重强调出来了。添加了这些变化，你可以重启应用并拥有了更安全、随机的 salt 密码。你可能会愿意加一些日志和实验，以查看应用运行期间和启动时用户数据加载时的加密数据变化。

要记住的是，尽管在这个例子中说明的是为 `UserDetails` 添加一个简单域的实现，这种方式可以作为基础来实现高度个性化的 `UserDetails` 对象以满足应用的业务需要。对于 JBCP Pets 来说，审计人员会对数据库中的安全密码感到很满意——一项任务被完美完成。

将 Remember me 功能迁移至数据库

现在你可能会意识到我们 remember me 功能的实现，能够在应用重启前很好的使用，但在应用重启时用户的 session 会被丢失。这对用户来说会不太便利，他们不应该关心 JBCP Pets 的维护信息。

幸运的是，Spring Security 提供了将 `rememberme token` 持久化到任何存储的接口 `o.s.s.web.authentication.rememberme.PersistentTokenRepository`，并提供了这个接口的 JDBC 实现。

配置基于数据库的 remember me tokens

在这里，修改 remember me 的配置以持久化到数据库是非常简单的。Spring Security 配置的解析器能够识别出 `<remember-me>` 声明的 `data-source-ref` 新属性并为 `RememberMeServices` 切换实现类。让我们了解完成这个功能所需要的步骤。

添加 SQL 以创建 remember me schema

我们需要将包含期望 schema 定义的 SQL 文件放在 classpath 下（WEB-INF/classes 中），它会与我们在前面使用的其它启动 SQL 脚本放在一起。我们将这个 SQL 脚本命名为

remember-me-schema.sql:

```
create table persistent_logins (
    username varchar_ignorecase(50) not null,
    series varchar(64) primary key,
    token varchar(64) not null,
    last_used timestamp not null);
```

为嵌入式数据库声明添加新的 SQL 脚本

接下来，在 dogstore-security.xml 文件的<embedded-database>声明中添加对新 SQL 脚本的引用：

```
<jdbc:embedded-database id="dataSource" type="HSQL">
    <jdbc:script location="classpath:security-schema.sql"/>
    <jdbc:script location="classpath:remember-me-schema.sql"/>
    <jdbc:script location="classpath:test-users-groups-data.sql"/>
</jdbc:embedded-database>
```

配置 remember me 服务持久化到数据库

最后我们需要对<remember-me>声明做一些简单的配置修改使其指向我们使用的 data source：

```
<http auto-config="true" use-expressions="true"
      access-decision-manager-ref="affirmativeBased">
    <intercept-url pattern="/login.do" access="permitAll"/>
    <intercept-url pattern="/account/*.do"
                   access="hasRole('ROLE_USER') and fullyAuthenticated"/>
    <intercept-url pattern="/*" access="hasRole('ROLE_USER')"/>
    <form-login login-page="/login.do" />
    <remember-me key="jbcpPetStore" tokenValiditySeconds="3600"
                  data-source-ref="dataSource"/>
    <logout invalidate-session="true" logout-success-url=""
              logout-url="/logout"/>
</http>
```

这就是我们所有要做的。现在，如果你重启应用，将不会丢失之前合法用户设置的 remember me cookie。

基于数据库后台的持久化 tokens 是不是更安全？

你可能会回忆起我们在第三章实现的 TokenBasedRememberMeServices，它用 MD5 哈希算法将一系列与用户相关的数据编码成安全的 cookie，这种方式很难（但并非不可能）篡改。

`o.s.s.web.authentication.rememberme.PersistentTokenBasedRememberMeServices` 类实现了持久化 tokens 以及对 token 安全处理，它通过一个校验方法以稍微不同的方式处理潜在的篡改。

`PersistentTokenBasedRememberMeServices` 为每个用户创建一个唯一的序列号，用户在继续交互和认证时要使用序列号中唯一的 tokens。序列号和 token 被存储在 cookie 中，在认证时要用来与存储的 token 进行对比。序列号和 token 都是基于配置的长度随机生成的，这使得恶意用户成功暴力破解的可能性很小了。

与 `TokenBasedRememberMeServices` 类似，持久化的 token 也可能被 cookie 窃取或其它的 man-in-the-middle 技术。在使用持久化 token 时，依旧建议用自定义的子类将 IP 地址合并到持久化 token 中，以及对站点的敏感区域强制使用用户名和密码认证。

用 SSL 保护你的站点

在日常使用在线站点时，你很可能已经听说或使用过 SSL。安全套接字层（SSL）协议，以及其后续的传输层安全（TLS），被用来为网络上的 HTTP 事务提供传输层的安全——它们被称为安全的 HTTP 事务（HTTPS）。

简而言之，SSL 和 TLS 以一种对用户透明的方式保护原始的 HTTP 传输数据，这些数据在客户端浏览器和 web 服务器之间传输。但是作为开发人员，在设计安全站点时，规划使用 SSL 是很重要的。Spring Security 提供了一系列的配置选项可以灵活的将 SSL 集成到 web 应用中。

【尽管 SSL 和 TLS 是不同的协议（TLS 是更成熟的协议），但是大多数人更熟悉 SSL 这个术语，所以在本书的剩余部分，我们使用这个术语来代指 SSL 和 TLS 两个协议。】

详细介绍 SSL 协议的机制已经超出了本书的范围，有一些很好的书籍和技术论文很详细地介绍了其规范和协议（你可以从 *RFC: 5246: 传输安全协议 (TLS) Version 1.2* 开始，在以下地址 <http://tools.ietf.org/html/rfc5246>）

配置 Apache Tomcat 以支持 SSL

首先且最重要的是，如果你计划执行如下 SSL 相关的例子，需要配置应用服务器以支持 SSL 连接。对于 Apache Tomcat，这相对很容易。如果你在使用其它的应用服务器，请查看文档的相关部分。

生成 server key store

我们需要使用 Java 的 keytool 命令来生成一个 key store。打开一个命令提示窗口，并输入以下的命令：

```
keytool -genkeypair -alias jbcpproxy -keyalg RSA -validity 365  
-keystore tomcat.keystore -storetype JKS
```

按照提示进行如下的输入。输入密码 password 作为 key store 和个人密钥的密码。

```
What is your first and last name?
```

```
[Unknown]: JBCP Pets Admin
```

```
What is the name of your organizational unit?
```

```
[Unknown]: JBCP Pets
```

What is the name of your organization?

[Unknown]: JBCP Pets

What is the name of your City or Locality?

[Unknown]: Anywhere

What is the name of your State or Province?

[Unknown]: NH

What is the two-letter country code for this unit?

[Unknown]: US

Is CN=JBCP Pets Admin, OU=JBCP Pets, O=JBCP Pets, L=Anywhere, ST=NH, C=US
correct?

[no]: yes

这将会在当前目录下，生成一个名为 `tomcat.keystore` 的文件。这就是启用 Tomcat SSL 所使用的 key store。

【注意的是要执行的是 `genkeypair` 命令(在早于 java 6 的释放版本中要使用 `keytool` 的 `genkey` 命令)】

为了下一步的操作，需要记住这个文件的地址。

配置 Tomcat 的 SSL Connector

在 Apache Tomcat 的 conf 目录下，用 XML 编辑器(Eclipse 或类似的都可以)打开 `server.xml`，并取消注释或添加 SSL Connector 声明。应该如下所示：

```
<Connector port="8443" protocol="HTTP/1.1" SSLEnabled="true"  
maxThreads="150" scheme="https" secure="true"  
sslProtocol="TLS"  
keystoreFile="conf/tomcat.keystore"  
keystorePass="password"/>
```

确保在上一步中生成的 `tomcat.keystore` 文件被 copy 到了 Tomcat 安装路径的 `conf` 目录下。在配置后，Tomcat 服务器可以重启，JBCP Pets 应用能够在一个安全的端口 <https://localhost:8443/JBCPPets/> 上进行访问。

取决于不同的浏览器，可能需要包含 `https` 而不是 `http`。这样的问题可能会比较难发现，你可能会比较奇怪为什么不能看到 JBCP Pets 的主页。

对站点进行自动的安全保护

我们假设你在对客户的数据进行 SSL 保护时遇到了麻烦，你想把应用的特定部分置于 SSL 的保护之下。幸运的是，Spring Security 让这一切变得很简单，只需要在 `<intercept-url>` 声明上添加一个配置属性。

`requires-channel` 属性能够添加到任何 `<intercept-url>` 声明中，以要求所有匹配的 URL 要以特定的协议（HTTP，HTTPS 或都可以）进行传递。如果按照这种形式来增强 JBCP Pets 站点，配置可能如下所示：

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

```
<intercept-url pattern="/login.do" access="permitAll"
    requires-channel="https"/>
<intercept-url pattern="/account/*.do"
    access="hasRole('ROLE_USER') and fullyAuthenticated"
    requires-channel="https"/>
<intercept-url pattern="/*" access="permitAll"
    requires-channel="any"/>
<!-- ... -->
</http>
```

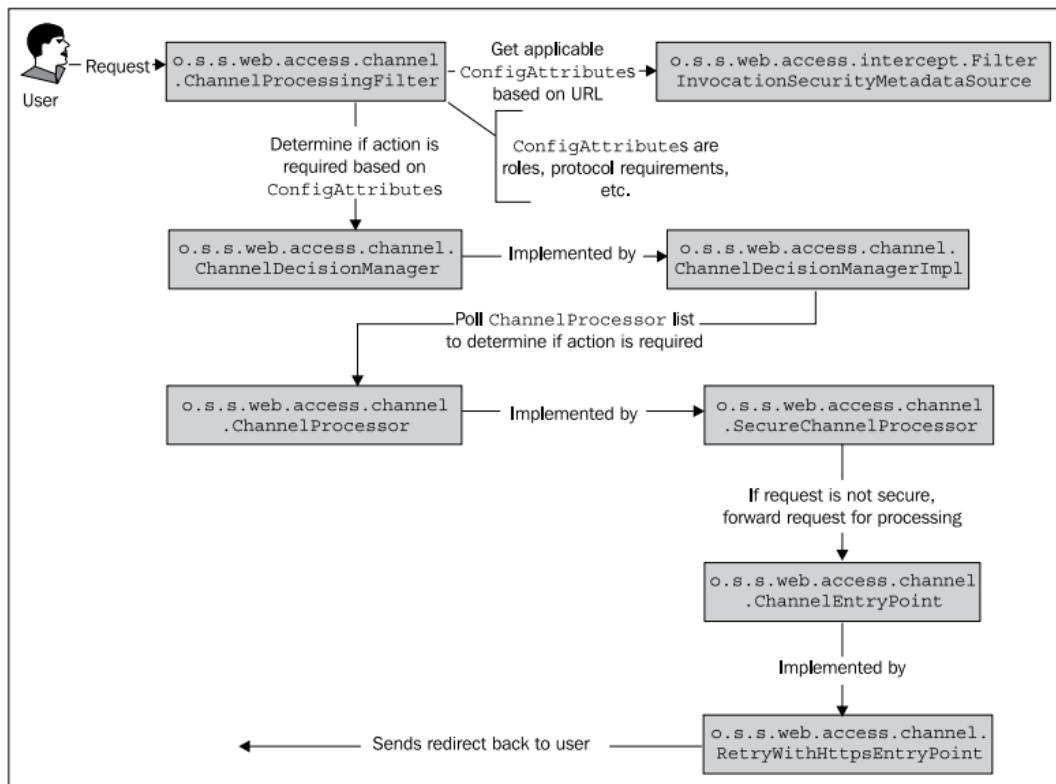
如果此时重启应用，你将会发现：

- 现在访问登录页和账号页需要 **HTTPS**，浏览器将会为用户自动从不安全的（**HTTP**）URL 重定向到安全的 URL。例如，尝试访问 <http://localhost:8080/JBCPPets/login.do> 将会被定向到 <https://localhost:8443/JBCPPets/login.do>；
- 如果用户被切换到了安全的 **HTTPS** URL，如果他访问一个不必要使用 **HTTPS** 的 URL，他能继续保留在 **HTTPS** 状态。

我们可以想象这种配置对于安全的好处——大多数的现代应用服务器使用一个 **secure** 标识 **session** 的 cookie，所以强制要求登录页是安全的（如果这是应用的 **session** 被首次分配的地方）能够保证 **session** 的 cookie 能够被安全的传输，所以出现 **session** 劫持的可能性也更小。另外，直接将 **SSL** 加密配置在安全声明上的做法，能够很容易的保证应用中所有敏感的页面被适当和完整的保护。

为用户自动切换适当协议（**HTTP** 或 **HTTPS**）的功能，通过 **Spring Security** 过滤器链上的另外一个 **servlet** 过滤器来实现的（它的位置很靠前，在 **SecurityContextPersistenceFilter** 后面）。如果任何 URL 用 **requires-channel** 属性声明使用特定类型的协议，**o.s.s.web.access.channel.ChannelProcessingFilter** 将会自动添加到过滤器链上

ChannelProcessingFilter 在请求时的交互过程如下图所示：



如果你的应用需要超出内置功能的复杂逻辑，`ChannelProcessingFilter` 的设计可以进行扩展和增强。注意我们尽管只在图中说明了 `SecureChannelProcessor` 和 `RetryWithHttpsEntryPoint` 的实现，但是有类似的类去校验和处理声明为要求 HTTP 的 URL。

注意，`ChannelEntryPoint` 使用了 HTTP 302 的 URL 重写，这就不能使用这种技术去重定向 POST 的 URL（尽管典型的 POST 请求不应该在安全协议和不安全协议间传递，因为大多数的应用都会对这种行为提出警告）。

安全的端口映射

在一些特定的环境中，可能不会使用标准的 HTTP 和 HTTPS 端口，其默认为 80/443 或 8080/8443。在这种情况下，你必须配置你的应用包含明确的端口映射，这样 `ChannelEntryPoint` 的实现能够确定当重定向用户到安全或不安全的 URL 时，使用什么端口。

这仅需要增加额外的配置元素`<port-mappings>`，它能够指明除了默认的端口以外，额外的 HTTP 的 HTTPS 端口：

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

如果你的应用服务器在反向代理后的话，端口映射将会更加的重要。

小结

在本章中，我们：

- 介绍了把安全数据存储在支持 JDBC 的数据库中是如何配置的；
- 配置 JBCP Pets 使用数据库来进行用户认证以及高安全性的密码存储，这里我们使

用了密码加密和 salting 技术；

- 管理 JDBC 持久化到数据中的用户；
- 配置用户到安全组中。组被授予角色，而不是直接对用户进行角色的指定。这提高了站点和用户功能的可管理性；
- 介绍了 Spring Security 使用遗留的（非默认的）数据库 schema；
- 讲解了 HTTPS 技术的配置及应用，它能够提高数据在访问应用敏感内容时的安全性。

在接下来的章节中，我们将会介绍 Spring Security 一些高级的授权功能，并引入 Spring Security 的 JSP 标签以实现良好的授权。

第五章 精确的访问控制

到目前为止，我们已经为 JBCP Pets 站点添加了用户友好的一些功能，包括自定义的登录页以及修改密码、remember me 功能。

在本章中，我们将要学习规划应用安全的技术以及用户/组的划分。其次，我们学习两种实现**精确访问控制**的实现方式——这会影响应用中页面的授权。然后，我们会了解 Spring Security 如何通过使用方法注解和 AOP 的方式来实现业务层安全。最后，我们将会了解通过基于注解的配置实现按照角色过滤集合数据这一比较有趣的功能。

在本章中，我们会学到：

- 规划 web 应用安全的基本技术和组管理，这会使用到现成的工具和批判思考 (critical thinking)；
- 基于用户请求的上下文，配置和实验在页面级别进行授权检查以显示内容的不同方式；
- 通过配置和代码注解 pre-authorization 的方式使得调用应用中关键部分是安全的；
- 几种实现方法级别安全的可选方式，并介绍各种方式的优劣；
- 通过使用方法级别的注解，实现基于 Collections 和 Arrays 数据的过滤器。

因为本章涉及到的概念超过了前面的一些孤立技术点，为了扩大站点的范围在源代码上做了一定数量的修改，并将其分成了真正三层的系统。你可能对这些变化感兴趣，但是它们与 Spring Security 没有直接的关系，所以我们将会忽略这些修改的细节。当你发现本章的源代码总添加了许多的文件，不要被吓倒。

重新思考应用功能和安全

现在，我们要再看一下 JBCP Pets 应用的授权模型和流程。我们感觉已经得到了一个很安全的应用，但是应用的流程并不特别适合与公开的电子商务站点。我们还需要做很多事情，因为对应用中每个页面（除去登录界面）的请求，都需要用户有一个合法的账号并登录——这无助于用户的浏览和购买。

规划应用安全

通常情况下，需要产品管理领域的人员和安全专员联合工程师来评估用户社区和需求的功能。规划过程——如果能够高效执行——使用工作表和图表来彻底分析应用包含的角色和组。我们会花一点时间简单介绍对 JBCP Pets 的扩展功能来阐明这个过程是如何进行的。在任何项目中对安全规划的思考过程将会对开发过程很有好处——尝试对你应用中的每个页面和业务服务构建安全状况。

规划用户角色

对于 JBCP Pets，我们将会使用下边的表格匹配用户分类到角色（Spring Security 的 GrantedAuthority 值）中。它们中有一些是新的角色，用来对用户进行不同的分类。

用户分类	描述	角色
Guest	不是记住或认证过的用户	None (anonymous)
Consumer / Customer	用户已经建立的账号，在站点上可能已完成也可能未完成购买交易	ROLE_CUSTOMER ROLE_USER
Customer w/ Completed Purchase	用户至少在站点上完成了一笔交易	ROLE_PURCHASER ROLE_USER
Administrator	负责用户账号管理等功能的管理员	ROLE_ADMIN ROLE_USER
Supplier	产品供货商，允许管理其产品目录	ROLE_SUPPLIER ROLE_USER

使用这些声明的用户分类和角色，我们能够将角色粗略得匹配到站点的功能设计上。有很多方式能够完成这项任务——以下是我们在过去发现很有用的办法：

- 使用 Microsoft Visio 和韦恩图来标示功能和用户组的重叠交叉（我们在第二章：*Spring Security 起步*中曾经使用过其很有限的功能）。这种技术对小型的应用和粗略的分析能够非常直观。
- 个性化的图表页面，并注明能够访问每个页面的用户分类和角色。尽管不能直接可视化的访问，但这种方式能够非常精确。我们将会在下面的章节阐述一个这样的例

子。

- 使用便条和白板或草图板建模。在这种类型的练习中，产品的规划人员在白板上勾画出一些区域来代表不同的用户角色，并在白板的每个区域上添加便签以代表产品功能。

通常来说，使用非数字的方式进行安全的初期规划是很容易的，因为经常见到组内讨论会产生对安全功能的较大修正，而使用非数字的工具很容易进行调整。典型情况下，这种层次的安全规划不会涉及到单个页面的和页面中某些部分的层次，而是应用中的功能“块”（即应用整体上的安全功能规划——译者注）。

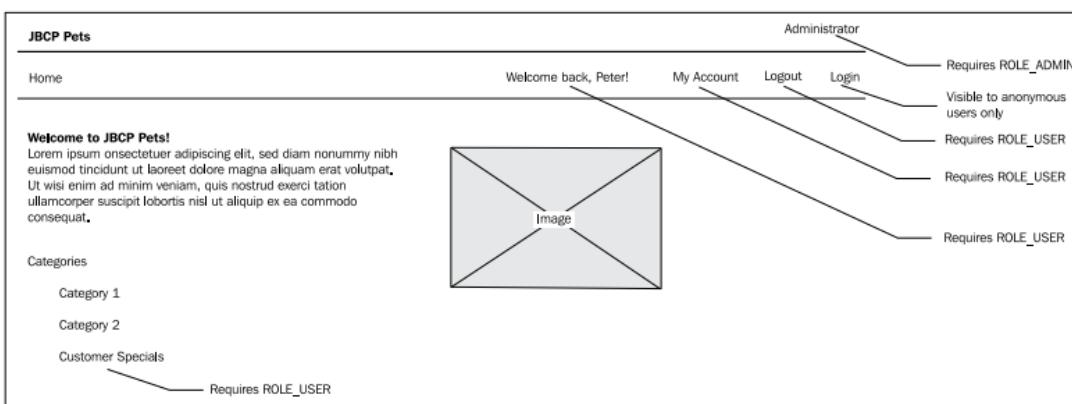
规划页面级权限

下一层次的安全规划就是页面级元素的安全。首先，规划整个站点范围的页面特性是很重要的，这能够保证用户在使用可见功能并切换页面时，保证界面的一致性。大多数的站点已经有了整个站点层次的模板功能，最简单的就是 `jsp:include` 指令（正如我们在 JBCP Pets 中使用的），或者更复杂的，如 Apache Tiles 2。

页面级别的安全规划通常与站点的用户体验规划结合起来——很多公司使用 Microsoft Visio 或者 Adobe Dreamweaver 进行站点的低保真设计，或者使用更复杂的工具如 Axure RP。不管使用什么工具，需要保证的是在规划安全相关的功能时要与站点的最初设计功能相融合。你的 UI 设计师或界面设计师可能会愿意讨论基于用户的角色，那些元素会显示或不显示。理解页面元素的可能选项能够使得好的 UI 规划人员设计出合理灵活的布局，从而保证不管用户是什么样的权限，页面都能展现得很好。

【使用正确的工作工具。我有一个 UI 设计师朋友，他为 Visio 做了很棒的形状集合，这使我产生了很大的兴趣，这些通过 http://www.guuui.com/issues/02_07.php 可以得到。对于熟悉 Visio 的人来说，这是一种不错的方式来开发精确、低保真的模型，对于这个工具许多人已经很了解。尽管现在没有兼容 Visio 的开源产品，类似的应用如 Dia (<http://projects.gnome.org/dia/>) 或 OpenOffice Draw (<http://www.openoffice.org/product/draw.html>) 对大多数的平台都是支持的。】

一个注明了安全信息的 Visio 图可能会如下所示：



可以看到我们并不需要很多安全相关信息的细节，但是这个图所表达的意思对于查看的每个人（即使不是技术人员）都很容易理解。

实现授权精确控制的方法

精确的授权指的是基于用户特定的请求进行授权的应用功能特性。不同于我们在第二章：*Spring Security 起步、第三章增强用户体验和第四章凭证安全存储*中的粗粒度的授权，精确的授权一般指的是对页面中的部分进行选择性显示的功能，而不是限制访问一个完整的页面。现实世界中的应用将会花费可观的时间用在规划精确授权的细节上。

Spring Security 为我们提供了两种方式来实现选择性显示的功能：

- **Spring Security** 的 JSP 标签库允许通过标准的 JSP 标签库语法在页面本身添加条件访问声明；
- 在 **MVC** 应用的**控制层**，检查用户的授权从而使得控制层做出能否访问的判断并将决定的结果绑定到**模型**数据提供给**视图层**。这种方式依赖于标准的 **JSTL** 条件实现界面渲染和数据绑定，这种方式比 **Spring Security** JSP 标签库复杂一些，但是，它与标准的 **web** 应用 **MVC** 逻辑设计更吻合。

在开发精确控制授权的 **web** 应用时，这两种方法都能很好的实现功能。让我们通过 **JBCP Pets** 用例来介绍没种方法的实现。

我们希望使用安全规划的结果来保证在网站范围内的菜单栏上“退出”和“我的订单”链接只能对登录过的或已购买的用户（分别为 **ROLE_USER** 和 **ROLE_CUSTOMER**）显示。我们还会保证“登录”链接只对浏览站点的未认证访客（不具备 **ROLE_USER** 的用户）可见。我们将会介绍这两种添加该功能的方式，首先从 **Spring Security** 的 JSP 标签库开始。

使用 **Spring Security** 的标签库有选择地渲染内容

我们在第三章中见到过，可以使用 **Spring Security** 的标签库访问存在于 **Authentication** 对象中的数据，这里我们将会见识到标签库的一些其它强大功能。**Spring Security** 标签库最常用的功能就是基于授权规则有条件地渲染页面的各部分。这是通过**<authorize>**标签来实现的，它与 **JSTL** 核心库的**<if>**标签类似，在标签体中的内容是否显示由标签属性的条件结果来确定。让我们使用**<authorize>**标签按条件显示页面的部分。

基于 URL 访问规则进行有条件的渲染

Spring Security 的标签库提供了按照已有的 URL 授权规则进行内容渲染的功能，而 URL 授权规则已经在应用安全的配置文件中进行了定义。这是通过使用**<authorize>**标签的**<url>** 属性来达到的。

例如，我们要保证“**My Account**”链接只能对实际登录站点的用户显示——回忆一下我们在前面定义的如下访问规则：

```
<intercept-url pattern="/account/*.*.do"
    access="hasRole('ROLE_USER') and fullyAuthenticated"/>
```

所以，JSP 中条件显示“**My Account**”链接的代码如下所示：

```
<sec:authorize url="/account/home.do">
    <c:url value="/account/home.do" var="accountUrl"/>
    <li><a href="${accountUrl}">My Account</a></li>
</sec:authorize>
```

这能够保证除非用户拥有足够的权限来访问指定的 URL，否则 tag 中的内容不会显示。还可以通过 HTTP 方法实现更高质量的检查，这要通过 method 属性来设置。

```
<sec:authorize url="/account/home.do"  method="GET">
    <c:url value="/account/home.do" var="accountUrl"/>
    <li><a href="${accountUrl}">My Account</a> (with 'url' attr)</li>
</sec:authorize>
```

使用 url 属性对代码块定义授权检查的方法是很方便的，因为它对 JSP 中的实际授权检查进行了抽象并将其保存在安全配置文件中。

【注意的是，HTTP 方法应该与<intercept-url>安全声明中的一致，否则它们将不会按照你预期的进行匹配。另外，注意 URL 应该是对于 web 应用上下文根的相对路径（如同 URL 访问规则一样）。】

对于很多场景来说，使用<authorize>标签能够保证只有用户允许看见的前提下，正确地渲染链接或 action 相关的内容。需要记住的是，这个标签不仅能够包在一个链接外面，如果用户没有权限提交这个 form 的时候，它还能包在整个 form 外边。

基于 Spring 表达式语言进行有条件渲染

另外，可以联合使用<authorize>标签和 Spring 表达式语言（SpEL）更灵活地显示 JSP 内容。

回忆一下在第二章中我们初次体验 SpEL 提供的强大表达式语言，Spring Security 对其进行了更强的扩展，从而能够对当前安全的请求构建表达式。如果我们对前面的例子使用 SpEL 进行重构的话，在<authorize>标签中限制访问“*My Account*”链接的代码应该如下：

```
<sec:authorize access="hasRole('ROLE_USER') and fullyAuthenticated">
    <c:url value="/account/home.do" var="accountUrl"/>
    <li><a href="${accountUrl}">My Account</a> (with 'access' attr)</li>
</sec:authorize>
```

对 SpEL 进行求值计算的代码与<intercept-url>所定义的访问规则（假设配置了表达式）背后所使用的代码是一样的。所以，同样的内置函数和属性在<authorize>标签中都是可以通过表达式使用的。

以上的两种使用<authorize>的方式都可以实现基于安全授权规则对页面显示内容进行精确控制渲染的强大功能。

使用 Spring Security2 的方式进行有条件渲染

以上提到的两种使用 Spring Security 标签库的方法实际上是 Spring Security3 新增的功能，

并且这也是按照授权规则实现页面级安全的推荐方法；但是，同样是<authorize>标签支持其他的操作方法，这可能会在遗留代码中遇到，也可能在一定场景下，这样的方式能够更好的满足你的需求。

基于缺失某角色有条件显示内容

“Log In”链接应该只能对匿名的用户显示，也就是没有 ROLE_USER 角色的用户。
<authorize>标签通过 ifNotGranted 属性支持这种类型的规则：

```
<sec:authorize ifNotGranted="ROLE_USER">
    <c:url value="/login.do" var="loginUrl"/>
    <li><a href="${loginUrl}">Log In</a></li>
</sec:authorize>
```

如果你现在以匿名用户试图访问站点，将会看到一个指向登录 form 的链接。

基于拥有列表中的某一个角色有条件显示内容

如同上一步那样，“Log Out”链接应该对拥有账号且已经登录的用户进行显示。
ifAnyGranted 属性在渲染内容前，要求用户拥有几个特定角色中的任何一个。我们用“Log Out”链接的方式来展示其使用：

```
<sec:authorize ifAnyGranted="ROLE_USER">
    <c:url value="/logout" var="logoutUrl"/>
    <li><a href="${logoutUrl}">Log Out</a></li>
</sec:authorize>
```

注意的是 ifAnyGranted 属性允许是以逗号分隔的角色集合来确定适当的匹配结果，用户只需要拥有角色中的任意一个标签中的内容就会渲染。

基于拥有列表中的所有个角色有条件显示内容

最后，使用 ifAllGranted 属性要求用户拥有标签中定义的所有角色：

```
<sec:authorize ifAllGranted="ROLE_USER,ROLE_CUSTOMER">
    <c:url value="/account/orders.do" var="ordersUrl"/>
    <li><a href="${ordersUrl}">My Orders</a></li>
</sec:authorize>
```

我们能够看到 authorize 标签的多种语法，以在不同的环境下使用。注意的是我们在前面讲到的三个属性可以组合使用。如 ifNotGranted 和 ifAnyGranted 属性能够联合使用以提供稍微复杂的 Boolean 等式。

使用 JSP 表达式

以上的三种页面授权方法（ifNotGranted, ifAnyGranted, ifAllGranted）支持 JSP EL 表

达式，它将会执行并返回授权的 GrantedAuthority（角色等）。如果授权要求的列表会根据页面计算结果而变化的话，这将会提供一定的灵活性。

使用控制器逻辑进行有条件的渲染内容

现在，让我们将刚刚用<authorize>标签实现的例子改成用 java 代码的方式。为了简洁起见，我们只实现一个例子，但实现基于控制器检查的其它例子是很简单直接的。

添加有条件的 Log In 链接

为了替代 Spring Security 的<authorize>标签，我们假设在模型数据中有一个 Boolean 变量来表示是否显示显示“Log in”链接，而这个变量可以在视图上得到。在传统的 MVC 模式下，视图不了解模型为何拥有这个值，它只需要使用即可。我们使用 Java Standard Tag Library (JSTL) 的 if 标签，连同 JSP EL 表达式来实现有条件的显示部分。

```
<c:if test="${showLoginLink}">
    <c:url value="/login.do" var="loginUrl"/>
    <li><a href="${loginUrl}">Log In</a></li>
</c:if>
```

现在看一下如下的代码以了解我们如何在控制器中将数据填充到模型中。

基于用户的凭证提供模型数据

我们控制器的对象模型是符合面向对象模式的，所有的 Spring MVC 控制器扩展自一个简单的基类 com.packtpub.springsecurity.web.controller.BaseController。这使得我们能够将通用的代码放在 BaseController 中，从而应用中所有的控制器都能够访问。首先，我们加入一个方法以从当前的 request 中得到 Authentication 的实现类：

```
protected Authentication getAuthentication() {
    return SecurityContextHolder.getContext().getAuthentication();
}
```

接下来，我们添加一个方法 填充 showLoginLink 模型数据，此处使用 Spring MVC 的注解方式。

```
@ModelAttribute("showLoginLink")
public boolean getShowLoginLink() {
    for (GrantedAuthority authority : getAuthentication().
        getAuthorities()) {
        if(authority.getAuthority().equals("ROLE_USER")) {
            return false;
        }
    }
    return true;
}
```

这个方法添加了 @ModelAttribute 注解，任何实现 BaseController 的控制器触发时，Spring MVC 将会自动执行此方法。针对 authorize 标签方式的其它显示/隐藏功能，我们能够很简单地使用模型数据指令重复这种设计模式。

配置页面内授权的最好方式是什么？

与以前版本的标签库相比，Spring Security 3 <authorize> 标签的主要优势在于移除了很多使用中的困扰之处。在很多场景下，使用标签的 url 属性隔离了授权规则变化对 JSP 代码的影响。可以在以下场景下使用 url 属性：

- 标签限制显示的功能能够通过一个简单 url 明确的声明；
- 标签的内容能够明确的与 URL 隔离。

但是在典型的应用中，使用标签 url 属性的可能性会很低。现实情况下应用会比这个复杂的多，需要更多的相关逻辑才能确定如何渲染页面的各部分。

尽管使用 Spring Security 标签库来声明渲染页面部分的方法很诱人，这种方式基于 spring 的语法以及一些其它的方式（包括 if...Granted 和 access 方法），但是有很多情况下使用它并不是一个好主意：

- Tag 标签并不支持比角色成员更复杂的条件。例如，如果我们的应用的 UserDetails 实现包含了自定义的属性，如 IP 过滤、地理位置等，这些情况使用标准的<authorize> 都不能支持。

但是，这些可以通过自定义的 JSP 标签或使用 SpEL 表达式来支持。即使如此，JSP 也会直接绑定业务逻辑，而这并不是推荐做法。

- <authorize> 标签必须在每一个使用的页面中引用。这可能会导致本来通用的规则产生潜在的不一致性，而这会在不同的物理界面文件中存在。好的面向对象系统设计建议条件规则只在一个地方存在，并在使用的地方对其进行引用。

可以通过封装并且重用 JSP 页面的部分来减少这类问题的发生（我们通过使用通用的 JSP 头文件已经对此进行了阐述），但是在一个复杂的应用中这个问题在所难免。

- 不能在编译阶段校验规则的正确性。编译期常量能够在典型的基于 java 对象系统中使用，JSP tag 标签需要（典型情况下）硬编码角色名字，而简单的拼写错误很难被察觉。

公平来讲，这样的拼写错误能够很容易地在运行应用的功能测试中发现，但是使用标准的 Java 组件单元测试这样的问题更容易被发现。

我们可以看到，尽管基于 JSP 方式的内容有条件渲染很便利，但是也有一些明显的不足。

所有的这些问题都能够通过在控制器中使用代码推送数据到视图层来解决。另外，在代码中进行高级的授权决定能够享受到重用、编译器检查以及适当分离模型、视图、控制器所带来的好处。

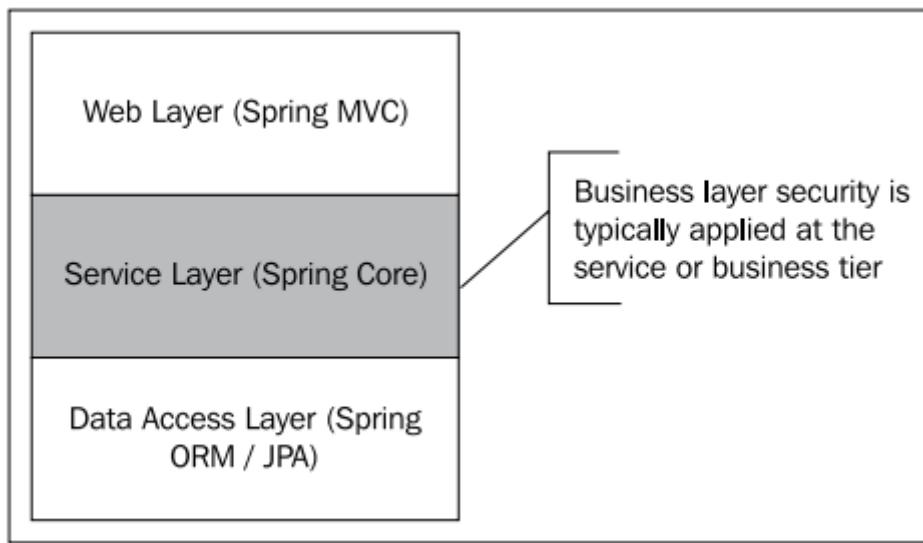
保护业务层

到目前为止，在本书中我们的关注点都主要在 JBCP Pets 应用 web 层面的安全。但是，在实际的安全系统规划中，对服务方法应该给予同等的重视，因为它们能够访问系统中最重要的一部分——数据。

Spring Security 支持添加授权层（或者基于授权的数据处理）到应用中所有 Spring 管理

的 bean 中。尽管很多的开发人员关注层的安全，其实业务层的俄安全同等主要，因为恶意的用户可能会穿透 web 层，能够通过没有 UI 的前端访问暴露的服务，如使用 web service。

让我们查看下面的图以了解我们将要添加安全层的位置：



Spring Security 有两个主要技术以实现方法的安全：

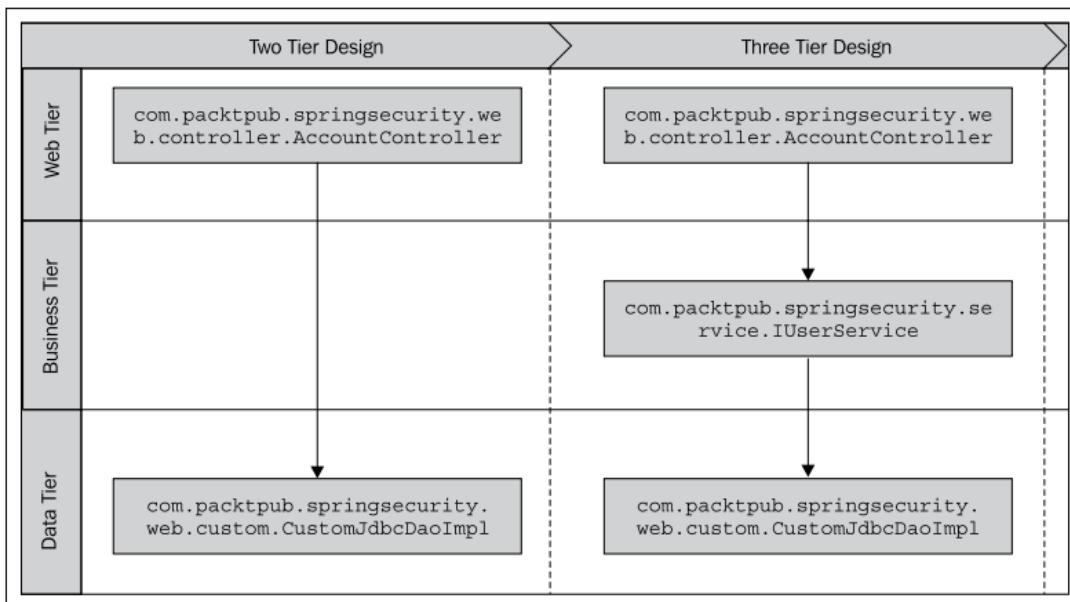
- 事先授权（Pre-authorization）保证在执行一个方法之前需要满足特定的要求——例如，一个用户要拥有特定的 GrantedAuthority，如 ROLE_ADMIN。不能满足声明的条件将会导致方法调用失败；
- 事后授权（Post-authorization）保证在方法返回时，调用的安全实体满足声明的条件。这很少被使用，但是能够在一些复杂交互的业务方法周围提供额外的安全层。

事先和事后授权在面向对象设计中提供了所谓的前置条件和后置条件（*preconditions and postconditions*）。前置条件和后置条件允许开发者声明运行时的检查，从而保证在一个方法执行时特定的条件需要满足。在安全的事前授权和事后授权中，业务层的开发人员需要对特定的方法确定明确的安全信息，并在接口或类的 API 声明中添加期望的运行时条件。正如你可能想象的那样，这需要大量的规划以避免不必要的影响。

保护业务层方法的基本知识

让我们以 JBCP Pets 中业务层的几个方法为例阐述怎样为它们应用典型的规则。

我们对 JBCP Pets 的基础代码进行了重新组织以实现三层的设计，作为修改的一部分我们抽象出了前面章节已经介绍到的修改密码功能到业务层。不同于用 web MVC 的控制器直接访问 JDBC DAO，我们选择插入一个业务服务以提供要求的附加功能。下图对此进行了描述：



我们能够看到在例子中 `com.packtpub.springsecurity.service.IUserService` 接口代表了应用架构的业务层，而这对我们来说，是一个合适位置来添加方法级的安全。

添加@PreAuthorize 方法注解

我们第一个的设计决策就是要在业务层上添加方法安全，以保证用户在修改密码前已经作为系统的合法用户进行了登录。这通过为业务接口方法定义添加一个简单的注解来实现，如下：

```
public interface IUserService {  
    @PreAuthorize("hasRole('ROLE_USER')")  
    public void changePassword(String username, String password);  
}
```

这就是保证合法、已认证的用户才能访问修改密码功能所要做的所有事情。Spring Security 将会使用运行时的面向方面编程的切点 (aspect oriented programming (AOP) pointcut) 来对方法执行 before advice，并在安全要求未满足的情况下抛出 `AccessDeniedException` 异常。

让 Spring Security 能够使用方法注解

我们还需要在 `dogstore-security.xml` 中做一个一次性的修改，通过这个文件我们已经进行了 Spring Security 其他的配置。只需要在 `<http>` 声明之前，添加下面的元素即可：

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

校验方法安全

不相信如此简单？那我们将 `ROLE_USER` 声明修改为 `ROLE_ADMIN`。现在用用户 `guest`(密码 `guest`) 登录并尝试修改密码。你会在尝试修改密码时，看到如下的出错界面：



如果查看 Tomcat 的控制台，你可以看到很长的堆栈信息，开始是这样的：

```
DEBUG - Could not complete request
o.s.s.access.AccessDeniedException: Access is denied
at o.s.s.access.vote.AffirmativeBased.decide
at o.s.s.access.intercept.AbstractSecurityInterceptor.beforeInvocation
...
at $Proxy12.changePassword(Unknown Source)
at com.packtpub.springsecurity.web.controller.AccountController.
submitChangePasswordPage
```

基于访问拒绝的页面以及指向 `changePassword` 方法的堆栈信息，我们可以看到用户被合理的拒绝对业务方法的访问，因为缺少 `ROLE_ADMIN` 的 `GrantedAuthority`。你可以测试修改密码功能对管理员用户依旧是可访问的。

我们只是在接口上添加了简单的声明就能够保证方法的安全，这是不是太令人兴奋了？当然，我们不会愿意 Tomcat 原生的 403 错误页面在我们的产品应用中出现——我们将会在第六章：高级配置与扩展讲述访问拒绝处理时，对其进行更新。

让我们介绍一下实现方法安全的其它方式，然后进入功能的背后以了解其怎样以及为什么能够生效。

几种实现方法安全的方式

除了`@PreAuthorize` 注解以外，还有几种其它的方式来声明在方法调用前进行授权检查的需求。我们会讲解这些实现方法安全的不同方式，并比较它们在不同环境下的优势与不足。

遵守 JSR-250 标准规则

JSR-250, Common Annotations for the Java Platform 定义了一系列的注解，其中的一些是安全相关的，它们意图在兼容 JSR-250 的环境中很方便地使用。Spring 框架从 Spring 2.x 释放版本开始就兼容 JSR-250，包括 Spring Security 框架。

尽管 JSR-250 注解不像 Spring 原生的注解富有表现力，但是它们提供的注解能够兼容不同的 Java EE 应用服务器实现如 Glassfish，或面向服务的运行框架如 Apache Tuscany。取决于你应用对轻便性的需求，你可能会觉得牺牲代码的轻便性但减少对特定环境的要求是值得的。

要实现我们在第一个例子中的规则，我们需要作两个修改，首先在 `dogstore-security.xml` 文件中：

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

其次，`@PreAuthorize` 注解需要修改成 `@RolesAllowed` 注解。正如我们可能推断出的那样，`@RolesAllowed` 注解并不支持 SpEL 表达式，所以它看起来很像我们在第二节中提到的 URL 授权。我们修改 `IUserService` 定义如下：

```
@RolesAllowed("ROLE_USER")
public void changePassword(String username, String password);
```

正如前面的练习那样，如果不相信它能工作，尝试修改 `ROLE_USER` 为 `ROLE_ADMIN` 并进行测试。

要注意的是，也可以提供一系列允许的 `GrantedAuthority` 名字，使用 Java 5 标准的字符串数组注解语法：

```
@RolesAllowed({"ROLE_USER","ROLE_ADMIN"})
public void changePassword(String username, String password);
```

JSR-250 还有两个其它的注解：`@PermitAll` 和 `@DenyAll`。它们的功能正如你所预想的，允许和禁止对方法的任何请求。

【类层次的注解。注意方法级别的安全注解也可以使用到类级别上！如果提供了方法级别的注解，将会覆盖类级别的注解。如果业务需要在整个类上有安全策略的话，这会非常有用。要注意的是使用这个功能要有良好的注释的编码规范，这样开发人员能够很清楚的了解类和方法的安全特性。】

我们将会在本章稍后的练习中介绍如何实现 JSR-250 风格的注解与 Spring Security 风格的注解并存。

@Secured 注解实现方法安全

Spring 本身也提供一个简单的注解，类似于 JSR-250 的 `@RolesAllowed` 注解。`@Secured` 注解在功能和语法上都与 `@RolesAllowed` 一致。唯一需要注意的不同点是要使用这些注解的话，要在 `<global-method-security>` 元素中明确使用另外一个属性：

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

因为 `@Secured` 与 JSR 标准的 `@RolesAllowed` 注解在功能上一致，所以并没有充分的理由在新代码中使用它，但是它能够在 Spring 的遗留代码中运行。

使用 Aspect Oriented Programming (AOP) 实现方法安全

实现方法安全的最后一项技术也可能是最强大的方法，它还有一个好处是不需要修改源代码。作为替代，它使用面向方面的编程方式为一个方法或方法集合声明切点（pointcut），而增强（advice）会在切点匹配的情况下进行基于角色的安全检查。AOP 的声明只在 Spring Security 的 XML 配置文件中并不涉及任何的注解。

以下就是声明保护所有的 `service` 接口只有管理权限才能访问的例子：

```
<global-method-security>
```

```
<protect-pointcut access="ROLE_ADMIN"  
    expression="execution(* com.packtpub.springsecurity.service.I*Service.*(..))"/>  
</global-method-security>
```

切点表达式基于 Spring AOP 对 AspectJ 的支持。但是，Spring AspectJ AOP 仅支持 AspectJ 切点表达式语言的一个很小子集——可以参考 Spring AOP 的文档以了解其支持的表达式和其它关于 Spring AOP 编程的重要元素。

注意的是，可以指明一系列的切点声明，以指向不同的角色和切点目标。以下的就是添加切点到 DAO 中一个方法的例子：

```
<global-method-security>  
    <protect-pointcut access="ROLE_USER"  
        expression="execution(* com.packtpub.springsecurity.dao.IProductDao.getCategories(..))  
&& args()"/>  
    <protect-pointcut access="ROLE_ADMIN" expression="execution(* com.  
packtpub.springsecurity.service.I*Service.*(..))"/>  
</global-method-security>
```

注意在新增的切点中，我们添加了一些 AspectJ 的高级语法，来声明 Boolean 逻辑以及其它支持的切点，而参数可以用来确定参数的类型声明。

同 Spring Security 其它允许一系列安全声明的地方一样，AOP 风格的方法安全是按照从顶到底的顺序进行的，所以需要按照最特殊到最不特殊的顺序来写切点。

使用 AOP 来进行编程即便是经验丰富的开发人员可能也会感到迷惑。如果你确定要使用 AOP 来进行安全声明，除了 Spring AOP 的参考手册外，强烈建议你参考一些这个专题相关的书籍。AOP 实现起来比较复杂，尤其是在解决不按照你预期运行的配置错误时更是如此。

比较方法授权的类型

以下的快速参考表可能在你选择授权方法检查时派上用场：

方法授权类型	声明方式	JSR 标准	允许 SpEL 表达式
@PreAuthorize @PostAuthorize	注解	No	Yes
@RolesAllowed @PermitAll @DenyAll	注解	Yes	No
@Secure	注解	No	No
protect-pointcut	XML	No	No

大多数使用 Java 5 的 Spring Security 用户倾向于使用 JSR-250 注解，以达到在 IT 组织间最大的兼容性和对业务类（以及相关约束）的重用。在需要的地方，这些基本的声明能够被 Spring Security 本身实现的注解所代替。

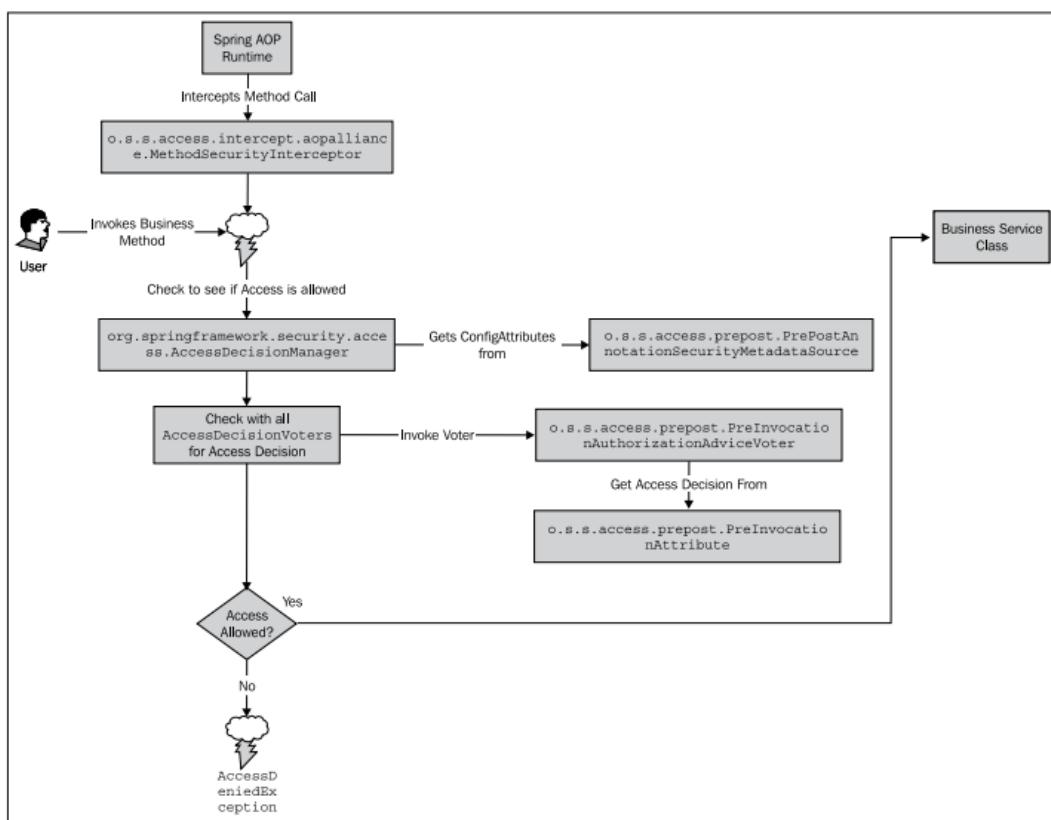
如果你在不支持注解的环境中（Java 1.4 或更早版本）中使用 Spring Security，很不幸的是，关于方法安全的执行你的选择可能会很有限。即使在这样的情况下，对 AOP 的使用也提供了相当丰富的环境来开发基本的安全声明。

方法的安全保护是怎样运行的？

方法安全的访问决定机制——一个给定的请求是否被允许——在概念上与 web 请求的访问决定逻辑是相同的。AccessDecisionManager 使用一个 AccessDecisionVoters 集合，其中每一个都要对能否进行访问做出允许、拒绝或者弃权的投票。AccessDecisionManager 汇集这些投票器的结果并形成一个最终能否允许处罚方法的决定。

Web 请求的访问决策没有这么复杂，这是因为通过 ServletFilters 对安全请求做拦截（以及请求拒绝）都相对很直接。因为方法的触发可能发生在任何的地方，包括没有通过 Spring Security 直接配置的代码，Spring Security 的设计者于是选择 Spring 管理的 AOP 方式来识别、评估以及保护方法的触发。

下图在总体上展现了方法触发授权决策的主要参与者：



我们能够看到 Spring Security 的 o.s.s.access.intercept.aopalliance.MethodSecurityInterceptor 被标准的 Spring AOP 运行时触发以拦截感兴趣的方法调用。通过上面的流程图，是否允许方法调用的逻辑就相对很清晰了。

此时，我们可能会比较关心方法安全功能的性能。显然，MethodSecurityInterceptor 不能在应用中每个方法调用的时候触发——那方法或类上的注解是如何做到 AOP 拦截的呢？

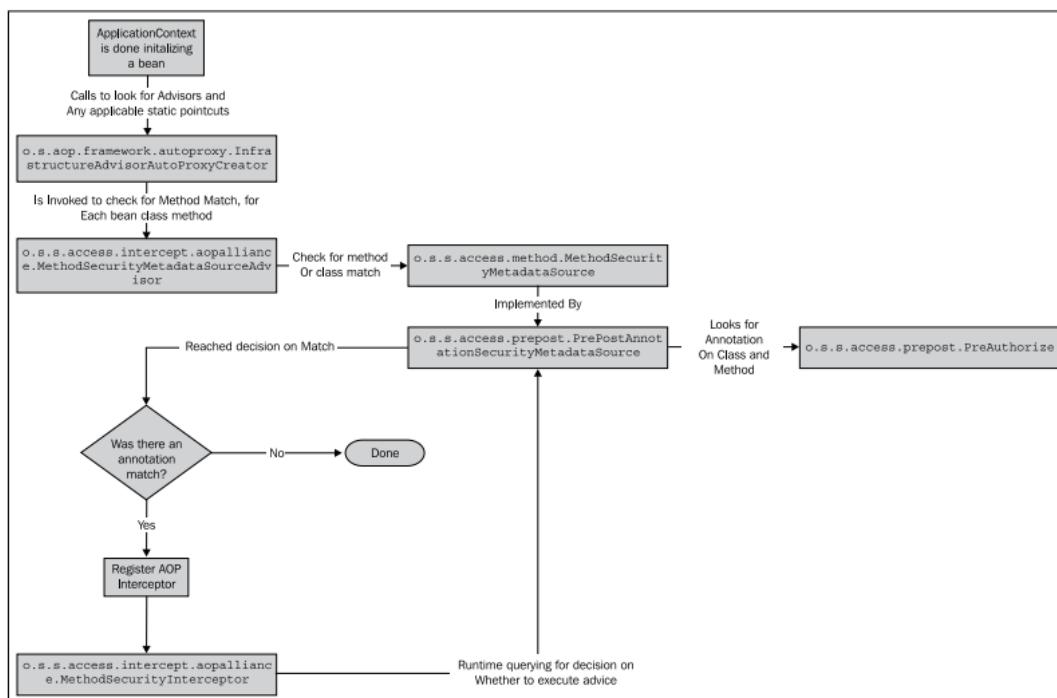
首先，AOP 织入默认不会对所有 Spring 管理的 bean 触发。相反，如果 `<global-method-security>` 在 Spring Security 配置中定义，一个标准的 Spring AOP `o.s.beans.factory.config.BeanPostProcessor` 将会被注册，它将会探查 AOP 配置是否有 AOP 增强器（advisors）需要织入（以及拦截）。这个工作流是 Spring 标准的 AOP 处理（名为 AOP 自动织入），并不是 Spring Security 所特有的。所有的 BeanPostProcessors 在 spring ApplicationContext 初始化时执行，在所有的 Spring Bean 配置生效后。

Spring 的 AOP 自动织入功能查询所有注册的 PointcutAdvisors，查看是否有 AOP 切点匹配方法的调用并使用 AOP 增强（advice）。Spring Security 实现了 o.s.s.access.intercept.aopalliance.MethodSecurityMetadataSourceAdvisor 类，它会检查所有配置的方法安全并建立适当的 AOP 拦截。注意的是，只有声明了方法安全的接口和类才会被 AOP 代理。

【强烈建议在接口上声明 AOP 规则（以及其它的安全注解），而不是在实现类上。使用类（通过 Spring 的 CGLIB 代理）进行声明可能会导致应用出现不可预知的行为改变，通常在正确性方面比不上在接口定义安全声明（通过 AOP）。】

MethodSecurityMetadataSourceAdvisor 将 AOP 影响方法行为的决定委托给 o.s.s.access.method.MethodSecurityDataSource 的实例。不同的方法安全注解都拥有自己的 MethodSecurityDataSource，它将用来检查每个方法和类并添加在运行时执行的增强（advice）。

以下的图展现了这个过程是如何发生的：



取决于你的应用中配置的 Spring Bean 的数量，以及拥有的安全方法注解的数量，添加方法安全代理将会增加初始化 ApplicationContext 的时间。但是，一旦上下文初始化完成，对单个的代理 bean 来说性能的影响可以忽略不计了。

方法安全的高级知识

方法安全的表现力不仅局限于简单的角色检查。实际上，一些方法安全的注解能够完全使用 Spring 表达式语言（SpEL）的强大功能，正如我们在第二章中讨论 URL 授权规则所使用的那样。这意味着任意的表达式，包含计算、Boolean 逻辑等等都可以使用。

使用 bean 包装类实现方法安全规则

另外一种定义方法安全的形式与 XML 声明有关，它可以包含在 Spring Bean 定义中。尽管阅读起来很容易，但是这种方式的方法安全声明在表现性上不如切点，在功能上不如我们已经见过的注解方式。但是，对于一定类型的工程，使用 XML 声明的方式足以满足你的需求。

我们可以替换前面的例子，将其改成基于 XML 声明的方式来保护 changePassword 方法。前面我们已经使用了 bean 的自动织入，但是这与 XML 方法包装方式并不兼容，为适应这项技术我们需要明确声明服务层类。

安全包装是安全 XML 命名空间的一部分。首先我们需要在 dogstore-base.xml 文件中，包含进来安全的 schema，它用来包含安全相关的 Spring Bean 定义：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:jdbc="http://www.springframework.org/schema/jdbc"
    xmlns:security="http://www.springframework.org/schema/security"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/aop http://www.
springframework.org/schema/aop/spring-aop-3.0.xsd
http://www.springframework.org/schema/jdbc  http://www.
springframework.org/schema/jdbc/spring-jdbc-3.0.xsd
http://www.springframework.org/schema/context http://www.
springframework.org/schema/context/spring-context-3.0.xsd
http://www.springframework.org/schema/security http://www.
springframework.org/schema/security/spring-security-3.0.xsd
">
```

接下来（为了完成这个练习），移除 IUserService.changePassword 上的所有注解。

最后，用 Spring XML 的语法来声明 bean，添加如下的附加的包装，它声明任何想触发 changePassword 方法的人必须是一个 ROLE_USER：

```
<bean id="userService" class="com.packtpub.springsecurity.service.UserServiceImpl">
    <security:intercept-methods>
        <security:protect access="ROLE_USER" method="changePassword"/>
    </security:intercept-methods>
</bean>
```

像本章前面的其它例子那样，这个保护功能能够很容易地通过将 ROLE_USER 改为 ROLE_ADMIN 并尝试用 guest 用户账号修改密码来校验。

在背后，这种方式的方法安全保护功能使用了 MethodSecurityInterceptor，它被织入到 MapBasedMethodSecurityMetadataSource 中。拦截器使用它来决定合适的访问 ConfigAttributes。不同于可使用 SpEL 以拥有更强表达能力的 @PreAuthorize 注解，<protect> 声明只能在 access 属性中有逗号分隔的一系列角色（类似于 JSR-250 @RolesAllowed 注解）。

可以使用简单的通配符来注明方法名，如，我们可以用如下的方式保护给定 bean 里所有的 set 方法：

```
<security:intercept-methods>
    <security:protect access="ROLE_USER" method="set*"/>
</security:intercept-methods>
```

方法名匹配可以包含前面或后面的正则表达式匹配符 (*)。这个符号的存在意味着要对方法名进行通配符匹配，为所有匹配该正则表达式的方法添加拦截器。注意，其它常用的正则表达式操作符（如?或[]）并不支持。请查阅相关的 Java 文档以理解基本的正则表达式。更复杂的通配符匹配或正则匹配并不支持。

在新代码中这种方式的安全声明并不常见，因为有更富于表现力的方式，但是了解这种方式的安全包装还是有好处的，你可以把它当做方法安全工具栏中的一个可选项。这种方式对于无法为接口或类添加安全注解时特别有效，如当你想为第三方类库添加安全功能时。

包含方法参数的实现方法安全规则

逻辑上，对一些类型的操作来说在制定规则时引用方法的参数是很合理的。例如，我们可能要对 changePassword 方法进行重新限制，这样试图触发这个方法的用户必须满足两个约束条件：

- 用户试图修改的必须是自己的密码，或者
- 用户是管理员（这种情况下，用户可以修改任何人的密码，这可能会通过一个管理界面）

修改这个规则限制只能管理员触发方法是很容易的，但是对我们来说怎样确定用户试图修改的是自己的密码并不清楚。

幸运的是，Spring Security 方法注解所绑定的 SpEL 支持更复杂的表达式，包括含有方法参数的表达式。

```
@PreAuthorize("#username == principal.username and hasRole('ROLE_USER')")
public void changePassword(String username, String password);
```

译者注：个人感觉注解更应该是：@PreAuthorize("#username == principal.username or hasRole('ROLE_USER')")

在这里，你可以看到我们对第一个练习中使用的 SpEL 指令进行了增强，校验安全实体的用户名与方法参数的用户名一致（#username——方法的参数名有一个#前缀）。方法参数绑定的强大功能可以使你更有创造力并允许对方法的安全保护有更精确的逻辑规则。

参数绑定是如何实现的？

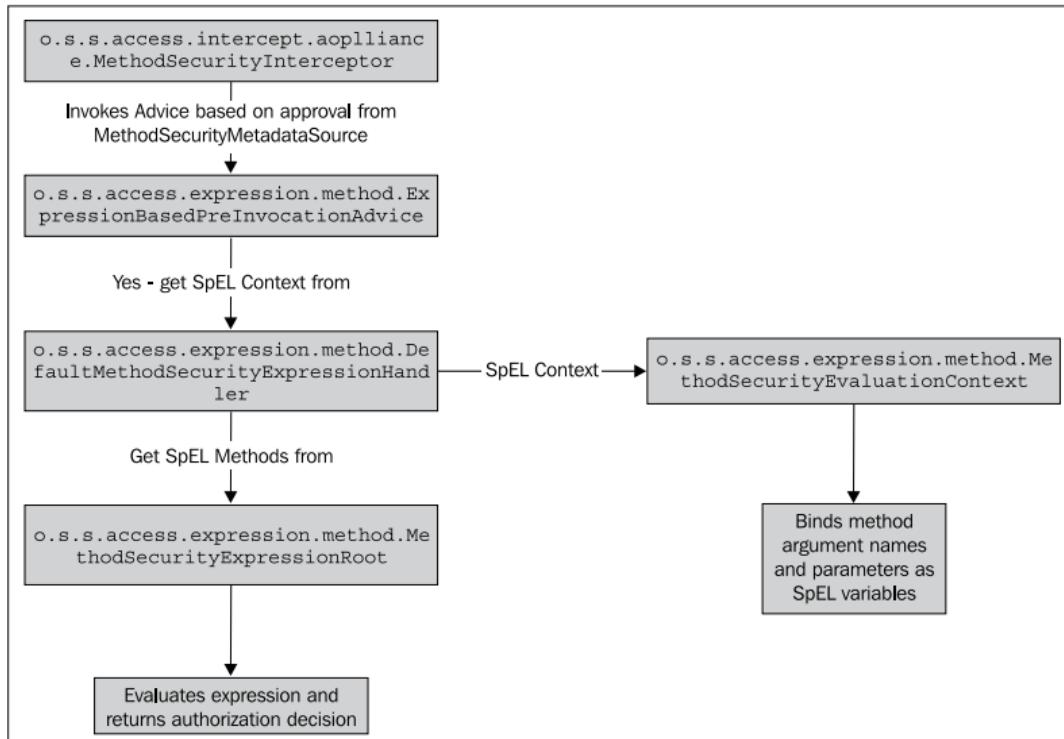
与我们在第二章中<intercept-url>授权表达式的设计类似，一个表达式处理器——o.s.s.access.expression.method.MethodSecurityExpressionHandler 的实现类——负责建立 SpEL 的上下文，表达式基于这个上下文进行求值。MethodSecurityExpressionHandler 使用 o.s.s.access.expression.method.MethodSecurityExpressionRoot 作为表达式根，它（与 WebSecurityExpressionRoot 为 URL 授权表达式所做的那样）为 SpEL 表达式的求值暴露了一系列的方法和伪属性。

与第二章中我们见到过的内置表达式（如 hasRole）基本完全一致，这些表达式也能够

在方法安全的上下文中使用，只是添加了一个与访问控制列表相关的方法（将在第七章：访问控制列表中介绍）以及另一个用来基于角色过滤数据的伪属性。

你可能注意到在前面的例子中，相对于 web 层的表达式来说，我们使用的 principal 伪属性是一个在方法安全表达式中很重要的表达式操作符。principal 伪属性将会返回在当前 Authentication 对象中的 principal，一般来讲会是一个字符串（用户名）或 UserDetails 实现——这就意味着 UserDetails 的所有属性和方法都能被使用来完善方法的访问限制。

下图展现了这个方面的功能：



SpEL 变量的应用要通过#前缀。需要注意的很重要一点是，为了使得方法参数的名字能够在运行时得到，调试符号表中的信息必须在编译后保留。启用这个功能的常见方法如下：

- 如果你使用的 javac 编译器，在构建 class 使，要加上-g 标示；
- 如果在 ant 中使用<javac>任务，添加 debug="true" 属性；
- 在 Maven 中，在构建你的 POM 是设置属性 maven.compiler.debug=on。

查阅你的编译器、构建工具或 IDE 文档寻求帮助，以实现在你的环境中相同的设置。

使用基于角色的过滤保护方法的数据安全

最后两个依赖 Spring Security 的注解是 @PreFilter 和 @PostFilter，它们被用来对 Collections 或 Arrays（仅@PostFilter 有效）使用基于安全的过滤规则。这种类型的功能被称为安全修正或安全修剪（security trimming or security pruning），并且涉及到在运行时使用安全实体的凭证进行集合对象的移除。正如你可能预想的那样，这种过滤通过在注解声明中使用 SpEL 表达式来实现。

我们将会讲解一个 JBCP Pets 的例子，在其中我们将会对系统用户显示一个特别的分类，叫做顾客最爱（Customer Appreciation Specials）。另外，我们将会使用 Category 对象的 customersOnly 属性来保证特定分类的产品只能对该存储的顾客可见。

对于使用 Spring MVC 的 web 应用来说，相关的代码很简单直接。

com.packtpub.springsecurity.web.controller.HomeController 类用来显示主页，它拥有显示分类——一个包含 Category 对象的 Collection——到用户主页的代码：

```
@Controller  
public class HomeController extends BaseController {  
    @Autowired  
    private IProductService productService;  
    @ModelAttribute("categories")  
    public Collection<Category> getCategories() {  
        return productService.getCategories();  
    }  
  
    @RequestMapping(method=RequestMethod.GET,value="/home.do")  
    public void home() {  
    }  
}
```

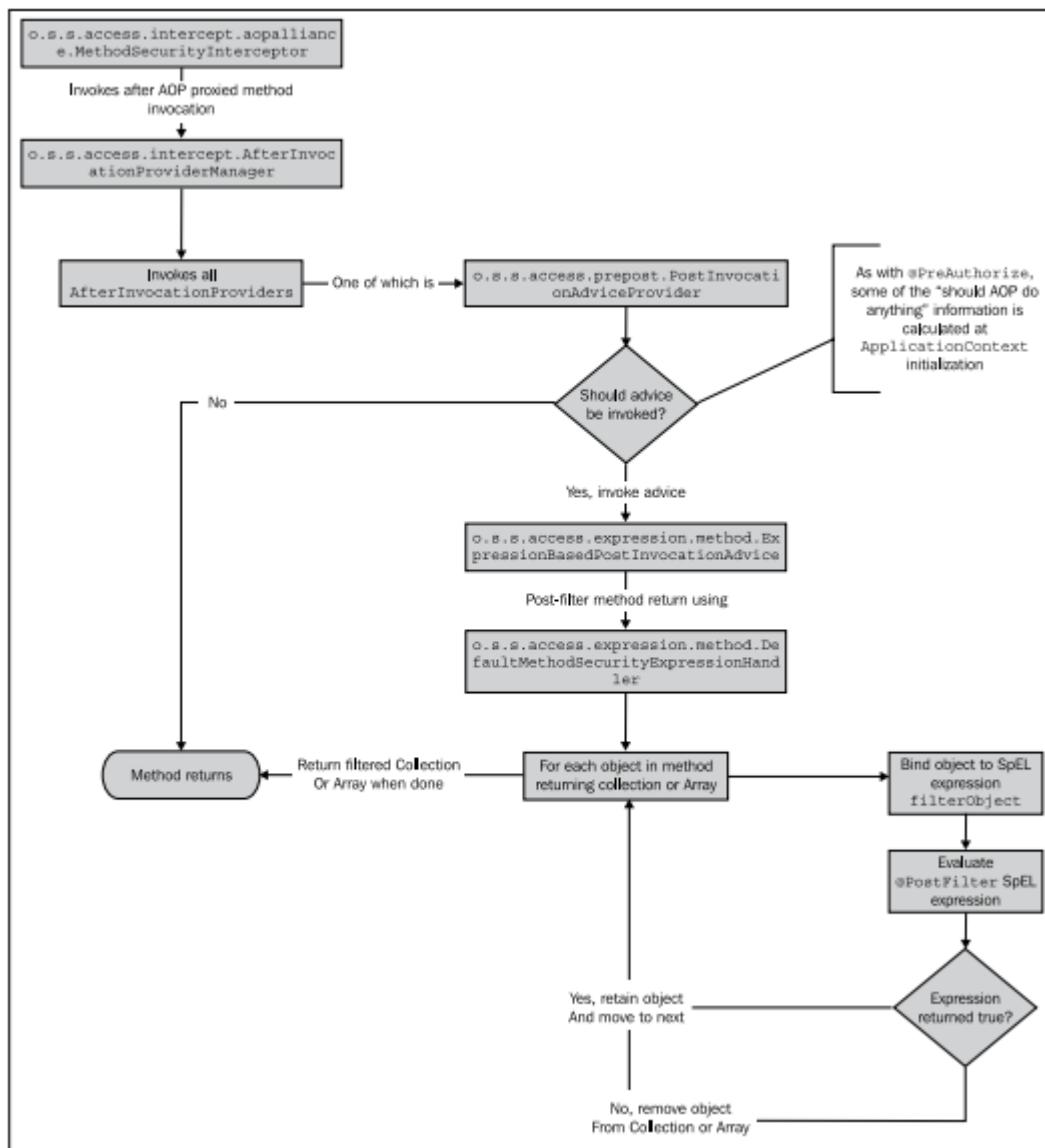
业务层 IProductService 接口的实现委托给数据访问层 IProductDao。简单起见，IProductDao 接口的实现类使用了一些硬编码的 Category 对象。

通过@PostFilter 实现基于角色的数据过滤

如同我们在方法安全授权中所作的那样，放置@PostFilter 安全过滤指令在业务层上。在本例中，代码如下：

```
@PostFilter("(!filterObject.customersOnly)      or      (filterObject.customersOnly      and  
hasRole('ROLE_USER'))")  
Collection<Category> getCategories();
```

在理解它的工作原理之前，我们首先看一下@PostFilter 注解的处理流程：



我们可以看到，再次使用了 Spring AOP 的标准组成，在一个 after 的 AOP 处理器中 `o.s.s.access.expression.method.ExpressionBasedPostInvocationAdvice` 被执行，为这个增强（advice）被用来过滤目标方法返回的 Collection 或 Array。像@PreAuthorize 注解的处理那样，`DefaultMethodSecurityExpressionHandler` 被再次用在这个表达式构建 SpEL 上下文和求值上。

应用修改后的效果能够在以 guest 和登录用户访问 JBCP Pets 时看到。你可以看到，当作为注册用户登录，顾客最爱（Customer Appreciation Specials）分类将会对注册用户可见。



现在，我们已将学习方法后过滤的处理过程，让我们回到所使用的进行分类过滤的 SpEL 表达式上来。简单起见，我们引用 `Collection` 作为方法的返回值，但是`@PostFilter` 可以在 `Collection` 和 `Array` 返回类型的方法中使用。

- `filterObject` 对于 `Collection` 中的每一个元素都会重新绑定到 SpEL 上下文中。这意味着，如果你的方法返回了包含 100 个元素的 `Collection`，SpEL 表达式将会对每一个进行求值。
- SpEL 表达式必须返回一个 `Boolean` 值。如果表达式的求值为 `true`，这个元素将会保留在 `Collection` 中，如果表达式求值为 `false`，这个元素将会被移除。

在大多数情况中，你会发现 `Collection` 的事后过滤将会为你节省到处书写的大量模板代码。

注意理解`@PostFilter` 在原理上怎样生效，不像`@PreAuthorize`，`@PostFilter` 指定了方法行为而不是事先条件。一些追求纯正面向对象的人可能会认为`@PostFilter` 包含在方法注解并不合适，而是这样的过滤应该在一个方法实现中通过代码进行处理。

【`Collection` 过滤的安全性。需要注意的是你的方法实际返回的 `Collection` 被修改了。在一些场景下，这并不是合适的行为，所以你需要保证你方法返回的 `Collection` 能够被安全地修改。如果返回的 `Collection` 是 ORM 绑定的，这一点尤其重要，因为事后过滤的修改可能会无意间持久化到 ORM 的数据存储中。】

Spring Security 还支持事先过滤 `Collections` 方法参数的功能，让我们尝试实现一下。

使用`@PreFilter` 实现事先过滤集合

`@PreFilter` 能被用来基于当前的安全上下文过滤传递到方法中的 `Collection` 元素。在功能上，只要拥有对 `Collection` 的引用，这个注解的行为与`@PostFilter` 除了以下两点外完全一致：

- `@PreFilter` 只支持 `Collection` 参数，不支持 `Array` 参数；
- `@PreFilter` 使用了一个额外可选的 `filterTarget` 属性，如果方法超过一个参数的话，这个属性被用来指明要过滤哪个参数。

同`@PostFilter` 一样，要记住传递到方法中的原始 `Collection` 会被永久修改。这可能不是

合适的行为，所以你要保证调用者能够了解对 Collection 的修剪在方法调用后是安全的。

为了展现这个过滤的使用，我们临时修改在 @PostFilter 注解中用到的 getCategories 方法，改成把它的过滤委托给一个新的方法。修改 getCategories 为如下：

```
@Override  
public Collection<Category> getCategories() {  
    Collection<Category> unfilteredCategories = productDao.getCategories();  
    return productDao.filterCategories(unfilteredCategories);  
}
```

我们要添加 filterCategories 方法到 IProductDao 接口和实现中。@PreFilter 注解要加到接口声明中，如下：

```
@PreFilter("(!filterObject.customersOnly) or (filterObject.customersOnly and  
hasRole('ROLE_USER'))")  
public Collection<Category> filterCategories(Collection<Category> categories);
```

一旦你添加了该方法和 @PreFilter 注解声明到接口中，添加一个空实现（尽管你可以在方法中按照业务需要进行进一步的过滤）。添加以下的方法体到 ProductDao 中：

```
@Override  
public Collection<Category> filterCategories(Collection<Category> categories) {  
    return categories;  
}
```

到此为止，你可以证实功能在从 IProductService 接口中移除 @PostFilter 注解后依旧正常使用，你会发现行为与前面完全一样。

到底为何使用 @PreFilter

此时，你可能挠头问 @PreFilter 到底有什么用处，因为 @PostFilter 的功能完全一样并适应更多的逻辑场景。

@PreFilter 的确有很多用处，有一些与 @PostFilter 重叠，但是记住当声明安全限制时，宁可多余——我们宁可过于小心也不能有潜在的安全危险。

以下是 @PreFilter 可能有用的场景：

大多数的应用都在数据层支持基于一系列的参数执行查询。@PreFilter 能够保证安全过滤掉传递到数据库查询中的参数。

在很多场景下，业务层收集来自于不同数据来源的信息。每个数据来源的输入能够进行安全的修剪以保证用户不会无意间看到他本不应该看到的搜索结果或数据。

@PreFilter 能够用来进行位置或关系相关的过滤——如可以基于用户点击过的分类或购买过的物品组成明确搜索条件的基础。

希望这能够帮助你了解在哪里使用对 Collections 的事先或事后过滤，以在你的应用中添加额外的保护层。

关于方法安全的警告

请记住这个关于实现和理解方法安全很重要的警告——为了真正很好地实现这个功能

强大的安全类型，理解其背后是怎样运行的很重要。缺乏对 AOP 的理解，不管是概念还是策略层面，都是造成方法安全失败的首要原因。请确保你不仅完整阅读本章，还有 Spring 3 Reference Documentation 的第七章：使用 Spring 进行面向方面编程。

在为一个已有系统实现方法安全之前，最好检查应用对面向对象设计原则的遵守情况。如果你的应用已经合理使用了接口和封装，当你实现方法安全时就会有更少的不可预知错误。

小结

在本章中，我们覆盖了 Spring Security 处理授权的大部分功能。我们已经通过对 JBCP Pets 在线商店在应用的各个层添加授权检查，学习了足够的知识，可以保证恶意用户不能操控或访问他们无权访问的数据。

尤其，我们：

- 学习了在应用设计过程中规划授权、用户/组匹配；
- 介绍两种实现细粒度授权的技术，基于授权或其它安全标准过滤出页面内容，使用了 Spring Security 的 JSP 标签库和 Spring MVC 控制器的数据绑定；
- 介绍了在业务层保护业务功能和数据的方法，支持丰富且与代码紧密集成的安全模型指令。

到此为止，我们已经介绍到了你在 web 安全应用开发中所使用到的很多 Spring Security 重要功能。

如果你一口气读到此处，这是一个很好的时间休息一下，复习我们所学的东西，并花些时间了解实例代码和 Spring Security 本身的代码。

在接下来的两章中，我们会涵盖高级的自定义和扩展场景，以及 Spring Security 的访问控制列表（域对象模型）模块。保证是令人兴奋的话题。

第六章 高级配置和扩展

到目前为止，我们已经介绍了大多数 Spring Security 组件的理论以及架构和使用。我们的 JBCP Pets 商业站点也在逐渐变成一个安全的 web 应用，我们将会深入讲解一些更有难度的挑战。

在本章的课程中，我们将会：

- 实现我们自己的安全过滤器，解决一个很有趣的问题，即对特定的用户角色用 IP 过滤的方式增强站点的安全；
- 构建自定义的 AuthenticationProvider 及需要的支持类；
- 理解和实现反黑客的措施名为 session 固化防护（session fixation protection）以及 session 的并发控制；
- 使用包括 session 并发控制等功能构建简单的用户 session 报告增强；
- 配置以及自定义访问拒绝后的行为和异常处理；
- 构建基于 Spring bean 的 Spring Security 配置，放弃使用便利的安全命名空间<http>配置风格，从头开始直接织入和实例化完整的 Spring Security 大量类；
- 了解如何通过基于 Spring bean 的方式配置 session 的处理和创建；
- 对比<http>配置风格相对于基于 Spring bean 配置风格的优劣；
- 学习 AuthenticationEvent 的架构以及事件处理和自定义；
- 构建一个自定义的 SpEL 表达式投票器，新建一个 SpEL 方法并在<intercept-url>表达式中使用。

实现一个自定义的安全过滤器

对于安全应用来说，一个很常见的定制化场景就是使用自定义的 servlet 过滤器，它能够用来增加应用特定的安全层，通过提供更完整的信息增强用户体验，并移除潜在的恶意行为。

在 servlet 过滤器级别实现 IP 过滤

一个能够使得 JBCP Pets 安全审计人员很高兴的功能增强就是围绕管理员账号的使用进行更强限制，或者（更好的是）针对所有用户对站点的管理操作。

我们通过一个过滤器来解决这个问题，保证所有具有 ROLE_ADMIN 角色的用户只能在一系列特定的 IP 地址上访问系统。我们将会在此做简单的地址匹配，但是你可以很容易地扩展这个例子，来使用 IP 掩码、从数据库中读取 IP 地址等。

细致的用户可能会意识到会有其它的方法来实现这个功能，包括更复杂的<intercept-url>访问声明；但是，为了阐述的方便，这是一个很简单直接的例子。记住在现实世界中，诸如网络地址转换（Network Address Translation，NAT）以及动态 IP 地址会使得基于 IP 的规则在公网或无管理的网络中很脆弱。

书写我们自己的 servlet 过滤器

我们的过滤器将会设置成目标角色以及特定的 IP 地址才能允许访问。我们将这个类命名为 `com.packtpub.springsecurity.security.IPRoleAuthenticationFilter`，并定义如下。这个代码有一些复杂，所以省略掉了一下不重要的代码列表。请查看本章的源代码以了解此类：

```
package com.packtpub.springsecurity.security;  
// imports omitted  
public class IPRoleAuthenticationFilter extends OncePerRequestFilter  
{}
```

可以看到，我们的过滤器继承自 Spring web 库中的 `o.s.web.filter.OncePerRequestFilter` 基类。但这并不是必须的，这对于具有复杂配置且至执行一次的过滤器来说很便利，所以我们建议使用。

```
private String targetRole;  
private List<String> allowedIPAddresses;
```

我们的过滤器具有两个属性——目标角色（如 `ROLE_ADMIN`），以及一个允许的 IP 地址列表。这将会通过标准的 Spring bean 定义进行配置，我们将会稍后见到。最后，我们到达这个 bean 的核心，也就是 `doFilterInternal` 方法。

```
@Override  
public void doFilterInternal(HttpServletRequest req, HttpServletResponse res, FilterChain chain) throws IOException, ServletException {  
    // before we allow the request to proceed, we'll first get the user's role  
    // and see if it's an administrator  
    final Authentication authentication = SecurityContextHolder.  
getContext().getAuthentication();  
    if (authentication != null && targetRole != null) {  
        boolean shouldCheck = false;  
        // look if the user is the target role  
        for (GrantedAuthority authority : authentication.getAuthorities()) {  
            if(authority.getAuthority().equals(targetRole)) {  
                shouldCheck = true;  
                break;  
            }  
        }  
        // if we should check IP, then check  
        if(shouldCheck && allowedIPAddresses.size() > 0) {  
            boolean shouldAllow = false;  
            for (String ipAddress : allowedIPAddresses) {  
                if(req.getRemoteAddr().equals(ipAddress)) {  
                    shouldAllow = true;  
                    break;  
                }  
            }  
        }  
    }  
}
```

```
        if(!shouldAllow) {
            // fail the request
            throw new AccessDeniedException("Access has been
denied for your IP address: "+req.getRemoteAddr());
        }
    } else {
        logger.warn("The IPRoleAuthenticationFilter should be placed
after the user has been authenticated in the filter chain.");
    }
    chain.doFilter(req, res);
}
// accessors (getters and setters) omitted
}
```

你可以看到，代码很简单明了，使用 `SecurityContext` 来获得 `Authentication` 关于当前请求的信息，就像我们在前面的章节练习中所作的那样。你可能会意识到没有很多特定与 `Spring Security` 的东西，除了获取用户角色（`GrantedAuthority`）的方法以及使用了 `AccessDeniedException` 来标示访问被拒绝。

让我们看一下如何作为一个 `Spring bean` 配置自定义的过滤器。

配置 IP servlet 过滤器

配置这个过滤器作为一个简单的 `Spring bean`。我们可以在 `dogstore-base.xml` 文件中配置它。

```
<bean id="ipFilter" class="com.packtpub.springsecurity .security.IPRoleAuthenticationFilter">
    <property name="targetRole" value="ROLE_ADMIN"/>
    <property name="allowedIPAddresses">
        <list>
            <value>1.2.3.4</value>
        </list>
    </property>
</bean>
```

使用标准的 `Spring bean` 配置语法，我们能够提供一系列的 IP 地址值。在本例中，`1.2.3.4` 显然不是合法的 IP 地址（如果本地部署的话，`127.0.0.1` 会是不错的一个 IP 地址配置），但是它为我们提供了便利的方式来测试这个过滤器是否生效。

最后，我们要将这个过滤器添加到 `Spring Security` 的过滤器链中。

将 IP servlet 过滤器添加到 Spring Security 过滤器链中

要添加到 `Spring Security` 过滤器链中的 `Servlet` 过滤器要通过相对于已经存在于过滤器链中其它过滤器来确定位置。自定义的过滤器要在 `<http>` 元素中配置，通过一件简单 `bena` 引用和位置标示，`IP servlet` 过滤器的配置如下：

```
<http>
    <custom-filter ref="ipFilter" before="FILTER_SECURITY_INTERCEPTOR"/>
</http>
```

需要记住的是我们的过滤器依赖于 `SecurityContext` 和 `Authentication` 对象，来进行辨别用户的 `GrantedAuthority`。所以，我们要将这个过滤器的位置放在 `FilterSecurityInterceptor` 之前，它（你可能会回忆起第二章：*Spring Security 起步*）负责确定用户是否有正确的权限访问系统。在过滤器的这个点上，用户的标示信息已经知道了，所以这是一个合适的位置以插入我们的过滤器。

现在你可以重启应用，因为这个新的 IP 过滤器，作为 `admin` 用户登录将会被限制。你可以对其进行随意的修改，以完全理解各部分是如何协同的。

【扩展 IP 过滤器。对于更复杂的需求，可以扩展这个过滤器以允许对角色和 IP 地址（子网匹配，IP 段等）更复杂的匹配。但是，在 `java` 中并没有广泛使用的类库来进行这种类型的计算——考虑到安全环境中普遍存在的 IP 过滤，这一点颇令人吃惊。】

尝试增强 IP 过滤器的功能以添加我们尚未描述到的功能——动手是学习的最好方法，而将练习改造成现实世界中的例子是将抽象概念变得具体的很好方式。

实现自定义的 AuthenticationProvider

在很多场景下，你的应用需要跳出 Spring Security 功能的边界，可能会需要实现自己的 `AuthenticationProvider`。回忆在第二章中 `AuthenticationProvider` 的角色，在整个认证过程中，它接受安全实体请求提供的凭证（即 `Authentication` 对象或 `authentication token`）并校验其正确性和合法性。

通过 AuthenticationProvider 实现一个简单的单点登录

通常，应用允许以用户或客户端方式登录。但是，有一种场景也很常见，尤其是在广泛使用的应用中，即允许系统中的不同用户以多种方式登录。

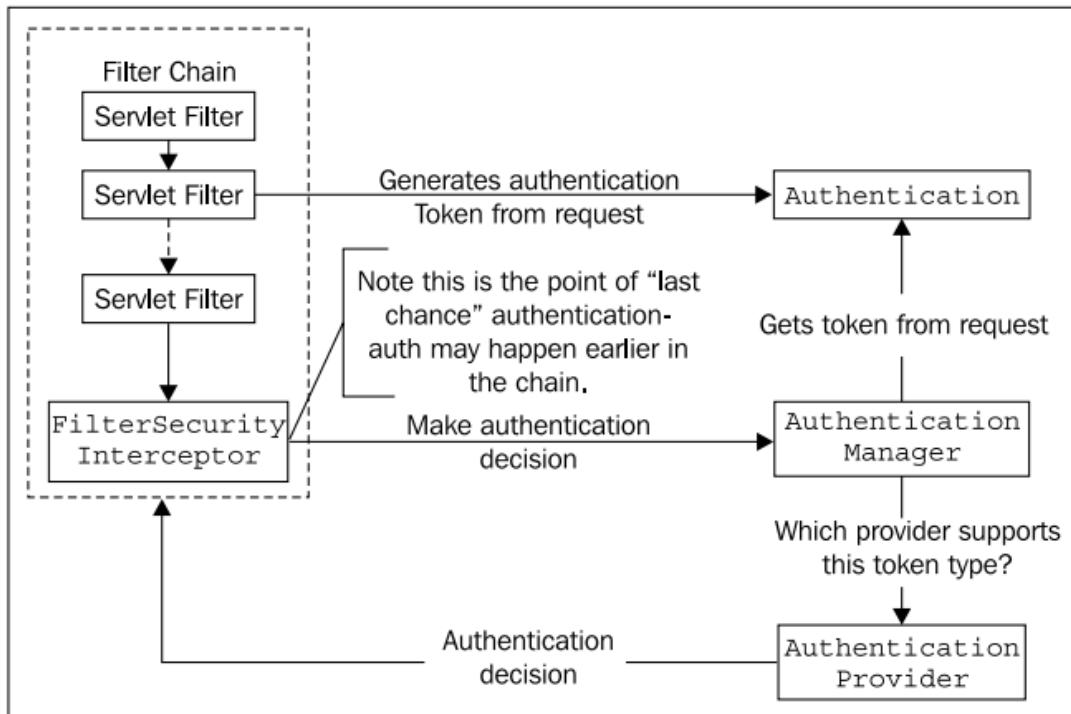
假设我们的系统与一个简单的“单点登录”提供者进行集成，用户名和密码分别在 HTTP 头中的 `j_username` 和 `j_password` 发送。除此以外，`j_signature` 头信息包含了用户名和密码的随意编码算法形成的字符串，以辅助安全请求。

【不要使用这个例子作为真实单点登录的解决方案。这个例子很牵强，只是为了说明实现一个完全自定义 `AuthenticationProvider` 的步骤。真正的 SSO 解决方案显然会更安全并涉及到几次的握手以建立可信任的凭证。Spring Security 支持几种 SSO 解决方案，包括中心认证服务（CAS）和 SiteMinder，我们将会在第十章：使用中心认证服务实现单点登录中介绍。实际上，Spring Security 提供了一个类似的过滤器用来进行 SiteMinder 请求头的认证，即 `o.s.s.web.authentication.preauth.RequestHeaderAuthenticationFilter`，也是这种类型功能的一个好例子。】

对于 `admin` 用户的登录，我们的算法期望在请求头中包含如下的数据：

请求头	值
<code>j_username</code>	<code>admin</code>
<code>j_password</code>	<code>admin</code>
<code>j_signature</code>	<code>admin + admin</code>

一般来讲，`AuthenticationProvider` 将会寻找特定的 `AuthenticationToken`，而后者会在过滤器链中位置比较靠前的 servlet filter 里面进行填充赋值（明确会在 `AuthenticationManager` 访问检查执行之前），如下图描述：



在这种方式中，`AuthenticationToken` 的提供者与其消费者 `AuthenticationProvider` 有一点脱离关系了。所以，在实现自定义 `AuthenticationProvider` 时，通常还需要实现一个自定义的 servlet 过滤器，其作用是提供特定的 `AuthenticationToken`。

自定义认证 token

我们实现自定义的方法会尽可能的使用 Spring Security 的基本功能。基于此，我们会扩展并增强基本类如 `UsernamePasswordAuthenticationToken`，并添加一个新的域来存储我们已编 码 的 签 名 字 符 串 。 最 终 的 类 `com.packtpub.springsecurity.security.SignedUsernamePasswordAuthenticationToken`，如下：

```
package com.packtpub.springsecurity.security;  
// imports omitted  
public class SignedUsernamePasswordAuthenticationToken  
    extends UsernamePasswordAuthenticationToken {  
    private String requestSignature;  
    private static final long serialVersionUID =  
        3145548673810647886L;  
    /**  
     * Construct a new token instance with the given principal,  
     * credentials, and signature.  
     *  
     * @param principal the principal to use
```

```
* @param credentials the credentials to use
* @param signature the signature to use
*/
public SignedUsernamePasswordAuthenticationToken(String principal,
    String credentials, String signature) {
    super(principal, credentials);
    this.requestSignature = signature;
}
public void setRequestSignature(String requestSignature) {
    this.requestSignature = requestSignature;
}
public String getRequestSignature() {
    return requestSignature;
}
}
```

我们可以看到 `SignedUsernamePasswordAuthenticationToken` 是一个简单的 POJO 类，扩展自 `UsernamePasswordAuthenticationToken`。Tokens 并不需要太复杂——它们的主要目的就是为后面的校验封装凭证信息。

实现对请求头处理的 servlet 过滤器

现在，我们要写 `servlet` 过滤器的代码，它负责将请求头转换成我们新定义的 token。同样的，我们扩展对应的 Spring Security 基本类。在本例中，`o.s.s.web.authentication.AbstractAuthenticationProcessingFilter` 满足我们的要求。

【基本过滤器 `AbstractAuthenticationProcessingFilter` 在 Spring Security 中是很多进行认证过滤器的父类（包括 OpenID、中心认证服务以及基于 form 的用户名和密码登录）。这个类提供了标准的认证逻辑并适当织入了其它重要的资源如 `RememberMeServices` 和 `ApplicationEventPublisher`（本章的后面将会讲解到）。】

现在，让我们看一下代码：

```
// imports omitted
public class RequestHeaderProcessingFilter extends
    AbstractAuthenticationProcessingFilter {
    private String usernameHeader = "j_username";
    private String passwordHeader = "j_password";
    private String signatureHeader = "j_signature";
    protected RequestHeaderProcessingFilter() {
        super("/j_spring_security_filter");
    }
    @Override
    public Authentication attemptAuthentication
        (HttpServletRequest request, HttpServletResponse response)
        throws AuthenticationException,
```

```
 IOException, ServletException {
    String username = request.getHeader(usernameHeader);
    String password = request.getHeader(passwordHeader);
    String signature = request.getHeader(signatureHeader);
    SignedUsernamePasswordAuthenticationToken authRequest =
        new SignedUsernamePasswordAuthenticationToken
            (username, password, signature);
    return this.getAuthenticationManager().authenticate(authRequest);
}
// getters and setters omitted below
}
```

可以看到，这个比较简单的过滤器查找三个已命名的请求头，正如我们已经规划的那样（如果需要的话，可以通过 bean 属性进行配置），并监听默认的 URL `/j_spring_security_filter`。正如其它的 Spring Security 过滤器那样，这是一个虚拟的 URL 并被我们的过滤器的基类 `AbstractAuthenticationProcessingFilter` 所识别，基于此这个过滤器采取行动尝试创建 Authentication token 并认证用户的请求。

【区分参与认证 token 流程的各个组件。在这个功能中，很容易被这些术语、接口和类的名字搞晕。代表要认证的 token 接口是 `o.s.s.core.Authentication`，这个接口的实现将以后缀 `AuthenticationToken` 结尾。这是一个很简单的方式来区分 Spring Security 提供的认证实现类。】

在本例中，我们尽可能将错误检查最小化（译者注：即参数的合法性与完整性的检查）。可能在实际的应用中，会校验是否所有头信息都提供了以及在发现用户提供信息不正确的时候要抛出异常或对用户进行重定向。

一个细小的配置变化是需要将我们的过滤器插入到过滤器链中：

```
<http auto-config="true" ...>
    <custom-filter ref="requestHeaderFilter"
        before="FORM_LOGIN_FILTER"/>
</http>
```

你可以看到过滤器代码最后请求 `AuthenticationManager` 来进行认证。这将最终委托配置的 `AuthenticationProvider`，它们中的一个要支持检查 `SignedUsernamePasswordAuthenticationToken`。接下来，我们需要书写一个 `AuthenticationProvider` 来做这件事情。

实现基于请求头的 `AuthenticationProvider`

现在，我们写一个 `AuthenticationProvider` 的实现类，即 `com.packtpub.springsecurity.security.SignedUsernamePasswordAuthenticationProvider`，负责校验我们自定义 Authentication token 的签名。

```
package com.packtpub.springsecurity.security;
// imports omitted
public class SignedUsernamePasswordAuthenticationProvider
    extends DaoAuthenticationProvider {
    @Override
    public boolean supports(Class<? extends Object> authentication) {
```

```
    return
(SignedUsernamePasswordAuthenticationToken .class.isAssignableFrom(authentication));
}

@Override
protected void additionalAuthenticationChecks
(UserDetails userDetails,
    UsernamePasswordAuthenticationToken authentication)
throws AuthenticationException {
super.additionalAuthenticationChecks
(userDetails, authentication);
SignedUsernamePasswordAuthenticationToken signedToken =
    (SignedUsernamePasswordAuthenticationToken) authentication;
if(signedToken.getRequestSignature() == null) {
    throw new BadCredentialsException(messages.getMessage(
        "SignedUsernamePasswordAuthenticationProvider
        .missingSignature", "Missing request signature"),
        isIncludeDetailsObject() ? userDetails : null);
}
// calculate expected signature
if(!signedToken.getRequestSignature()
.equals(calculateExpectedSignature(signedToken))) {
    throw new BadCredentialsException(messages.getMessage
        ("SignedUsernamePasswordAuthenticationProvider
        .badSignature", "Invalid request signature"),
        isIncludeDetailsObject() ? userDetails : null);
}
private String calculateExpectedSignature
(SignedUsernamePasswordAuthenticationToken signedToken) {
    return signedToken.getPrincipal() + "|" +
        signedToken.getCredentials();
}
}
```

你可以看到我们再次扩展了框架中的类 DaoAuthenticationProvider，因为有用的数据访问代码仍然需要进行实际的用户密码校验以及通过 UserDetailsService 加载 UserDetails。

这个类有些复杂，所以我们将分别介绍其中的每个方法。

Supports 方法，是重写父类的方法，向 AuthenticationManager 指明当前 AuthenticationProvider 要进行校验的期望运行时 Authentication token。

接下来，additionalAuthenticationChecks 方法被父类调用，此方法允许子类对 token 进行特有的校验。这正适合我们的策略，所以添加上我们对 token 新的签名检查。基本上已经完成了我们自定义“简单 SSO”的实现，仅剩一处配置了。

连接 AuthenticationProvider

一个常见的要求就是将一个或更多的 AuthenticationProvider 接口连接起来，因为用户可能会以几种校验方式中的某一种登录系统。

因为到目前为止，我们还没有了解其它的 AuthenticationProvider，我们可以假设以下的需求，即使用标准的用户名和密码基于 form 的认证以及前面实现的自定义简单 SSO 认证。当配置了多个 AuthenticationProvider 时，每个 AuthenticationProvider 都会检查过滤器提供给它的 AuthenticationToken，仅当这个 token 类型它支持时才会处理这个 token。以这种方式，你的应用同时支持不同的认证方式并不会有什么坏处。

连接多个 AuthenticationProvider 实际上很简单。只需要在我们的 dogstore-security.xml 配置文件中声明另一个 authentication-provider 引用。

```
<authentication-manager alias="authenticationManager">
    <authentication-provider ref= "signedRequestAuthenticationProvider"/>
    <authentication-provider user-service-ref="jdbcUserService">
        <password-encoder ref="passwordEncoder" >
            <salt-source ref="saltSource"/>
        </password-encoder>
    </authentication-provider>
</authentication-manager>
```

与我们安全配置文件中引用的其它 Spring bean 一样，signedRequestAuthenticationProvider 引用就是我们的 AuthenticationProvider，它将在 dogstore-base.xml 中与其它的 Spring bean 一起进行配置。

```
<bean id="signedRequestAuthenticationProvider"  class="com.packtpub.springsecurity.security
.SignedUsernamePasswordAuthenticationProvider">
    <property name="passwordEncoder" ref="passwordEncoder"/>
    <property name="saltSource" ref="saltSource"/>
    <property name="userDetailsService" ref="jdbcUserService"/>
</bean>
```

我们自定义的 AuthenticationProvider 的 bean 属性其实都是父类所需要的。这些也都指向了我们在 AuthenticationManager 的中第二个 authentication-provider 声明中的那些 bean。

最终已经完成了支持这个简单单点登录功能的编码和配置，至此可以给自己一个小小的喝彩。但是，还有一个小问题——我们应该怎样操作请求的 http 头以模拟我们的 SSO 认证提供者呢？

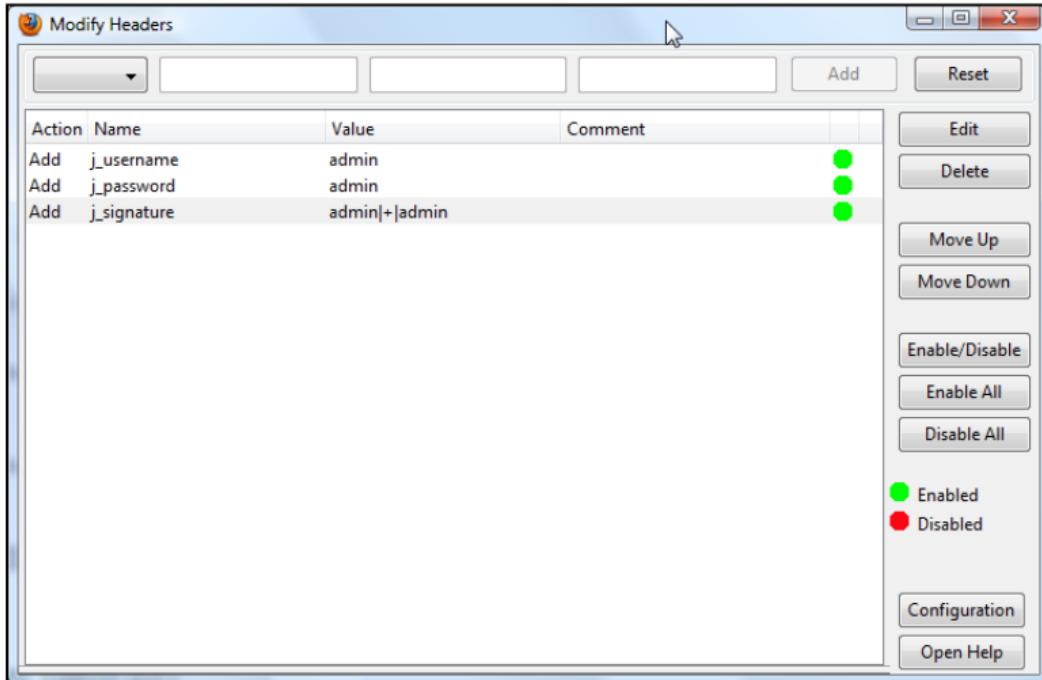
使用请求头模拟单点登录

尽管我们的场景比较牵强，但是有一些商业和开源的单点登录解决方案，它们能够被配置以通过 HTTP 请求头发送凭证信息，最具有代表性的是 CA（以前的 Netegrity）SiteMinder。

【需要特别注意的是，与 SSO 方案集成的应用是不能通过用户的直接请求访问的。通常情况下，SSO provider 功能作为代理，通过它确定用户的请求流程（是安全的）或 provider 持有关于密码的信息并将这些信息与单个的安全应用隔离。在没有完全了解一个其使用的硬件、网络和安全设施之前，不要部署 SSO 应用。】

Mozilla Firefox 的浏览器扩展，名为 Modify Headers（可以在以下地址获得：

<http://modifyheaders.mozdev.org>），是一个很简单的工具能够用来模拟伪造 HTTP 头的请求。以下的截图表明了如何使用这个工具添加我们的 SSO 方案所希望得到的请求头信息：



将所有的头信息标示为 Enabled，访问这个 URL：
http://localhost:8080/JBCPPets/i_spring_security_filter，会发现我们能够自动登录系统。你可能也会发现我们给予 form 的登录还能继续可用，这是因为保留了这两个 AuthenticationProvider 实现以及过滤器链中对应的过滤器。

（译者注：说实话，作者这个实现自定义 AuthenticationProvider 的例子真的是比较牵强，但是还算完整描述出了其实现方式，想深入了解 AuthenticationProvider 自定义的朋友，可以参照 Spring Security 提供的 CasAuthenticationProvider 等实现。）

实现自定义 AuthenticationProviders 时要考虑的事项

尽管我们刚刚看到的例子并没有阐述你想构建的 AuthenticationProvider，但是任何自定义 AuthenticationProvider 的步骤是类似的。这个练习的关键在于：

- 基于用户的请求完成一个 Authentication 实现的任务一般情况下会在过滤器链中的某一个中进行。取决于是否校验凭证的数据，这个校验组件可能要进行扩展；
- 基于一个合法的 Authentication 认证用户的任务需要 AuthenticationProvider 的实现来完成。请查看我们在第二章中讨论过的 AuthenticationProvider 所被期望拥有的功能；
- 在一些特殊的场景下，如果未认证的 session 被发现，可能会需要自定义的 AuthenticationEntryPoint。我们将会在本章接下来的部分更多了解这个接口，也会在第十章介绍中心认证服务（CAS）时，介绍一些 AuthenticationEntryPoint 的实际例子。

如果你能时刻记住它们的角色，当你在开发应用特定的 AuthenticationProvider 时，会在实现和调试过程中少很多的迷惑。

Session 的管理和并发

Spring Security 的一个常见配置就是检测相同的用户以不同的 session 登录安全系统。这被称为并发控制（**concurrency control**），是 session 管理（**session management**）一系列相关配置功能的一部分。严格来说，这个功能并不是高级配置，但是它会让很多新手感到迷惑，并且最好在你对 Spring Security 整体功能有所了解的基础上再掌握它。Spring Security 的 session 管理能够以两种不同的方式进行配置——session 固化保护（**session fixation protection**）和并发控制。因为并发控制的功能基于 session 固化保护所提供的框架，我们先介绍 session 固化。

配置 session fixation 防护

如果我们使用的是 security 命名空间的配置方式，session 固化防护已经被默认进行了配置。如果我们要指明将其配置为与默认设置一致的话，我们需要这样：

```
<http auto-config="true" use-expressions="true">
    <!-- ... -->
    <session-management session-fixation-protection="migrateSession"/>
</http>
```

Session 固化防护这个功能你可能并不会在意，除非你想扮演一个恶意的用户。我们将向你展示如何模拟一个 session 窃取攻击，但是在此之前，有必要理解 session 固化是怎么回事以及怎样防止这样的攻击。

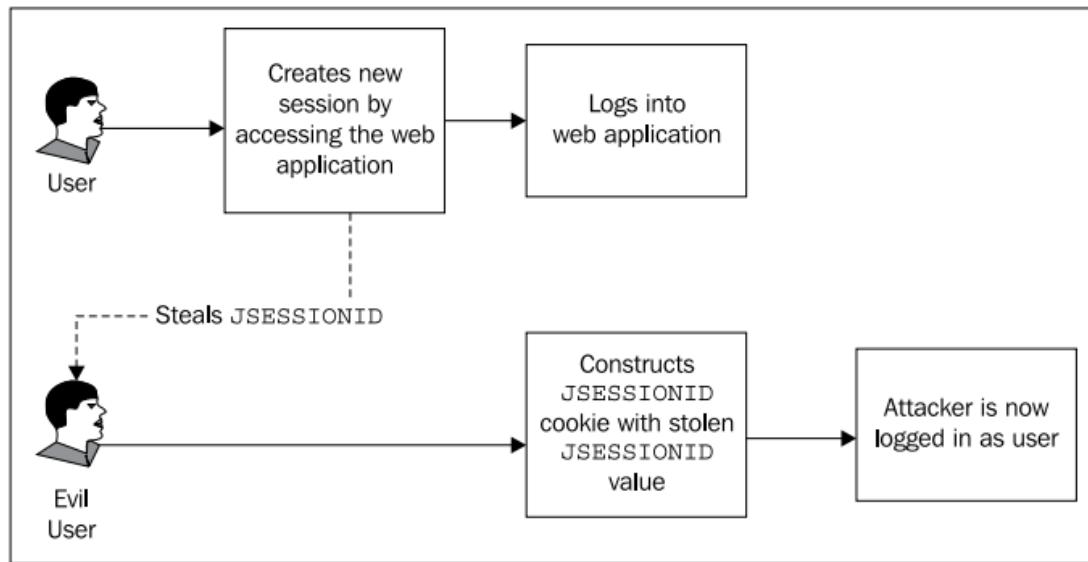
理解 session fixation 攻击

Session 固化是恶意用户试图窃取系统中一个未认证用户的 session。对攻击者来说，可以通过各种技术来获取用户 session 的唯一标识（例如，JSESSIONID）。如果攻击者创建了带有用户 JSESSIONID 的 cookie 或者 URL 参数，他就能够访问用户的 session。

尽管这是一个明显的问题，但是一般情况下，如果用户没有经过认证，他们就还没有输入任何敏感信息（假设站点的安全已经正确规划了）。如果用户认证后依旧使用相同的 session 标识符，这个问题就会比较更加重要了。如果用户在认证后还使用相同的标识符，那攻击者现在就能访问认证过用户的 session，而甚至不需要知道他们的用户名和密码。

【此时，你可能很不屑并认为在现实世界中这不会发生。实际上，session 窃取攻击经常发生。关于这个话题，我们建议（正如在第三章那样）你花些时间阅读由 OWASP 组织（<http://www.owasp.org/>）发布的包含重要信息的文章以及学习案例。攻击者和恶意用户是真实存在的，如果你不了解他们常用的技术及如何避免，他们会对你你的用户、应用或公司造成真正的损害。】

下图展现了 session 固化攻击是如何发生的：

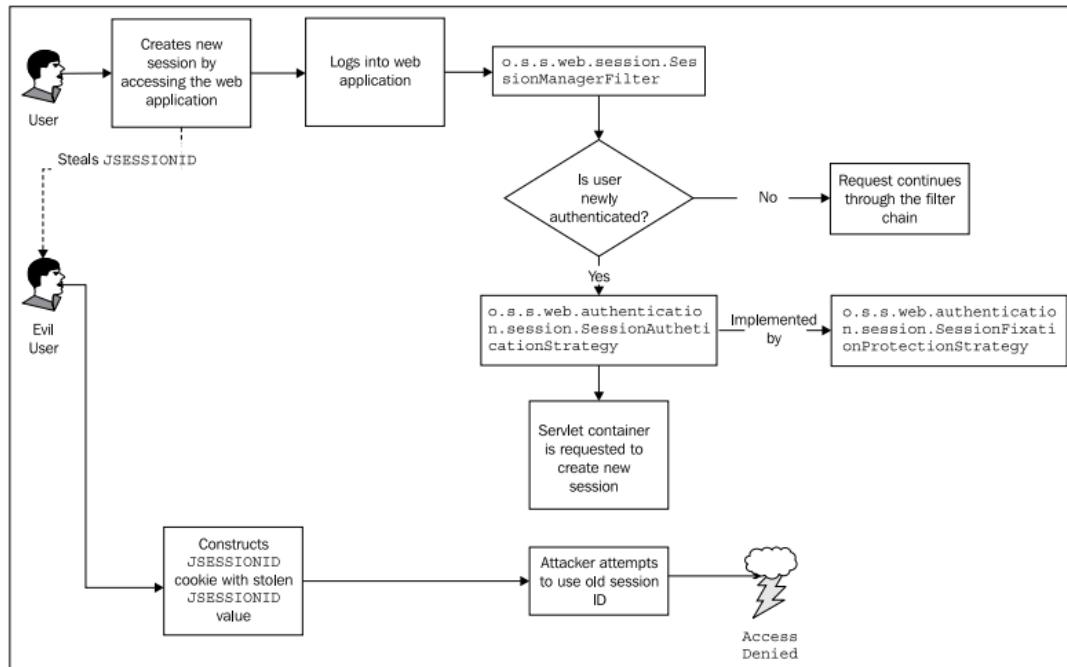


既然了解了攻击如何进行，接下来我们查看 Spring Security 如何防止。

使用 Spring Security 防止 session fixation 攻击

如果我们能够阻止用户在认证前和认证后使用相同的 session，我们就能够让攻击者掌握的 session ID 信息变得没有用处。Spring Security 的 session 固化防护解决这个问题的方式就是在用户认证之后明确创建一个新的 session 并将旧的 session 失效。

让我们看下图：



我们可以看到一个新的过滤器，`o.s.s.web.session.SessionManagementFilter`，负责检查一个特定的用户是否为新认证的。如果用户是新认证的，一个配置的`o.s.s.web.authentication.session.SessionAuthenticationStrategy` 将确定要怎样做。

`o.s.s.web.authentication.session.SessionAuthenticationStrategy` 将会创建一个新的 session（如果用户已经拥有一个的话），并将已存在 session 的内容拷贝到新 session 中去。这看起来很简单，但是，通过上面的图表我们可以看到，它能够有效组织恶意用户在未知用户登录后重用 session ID。

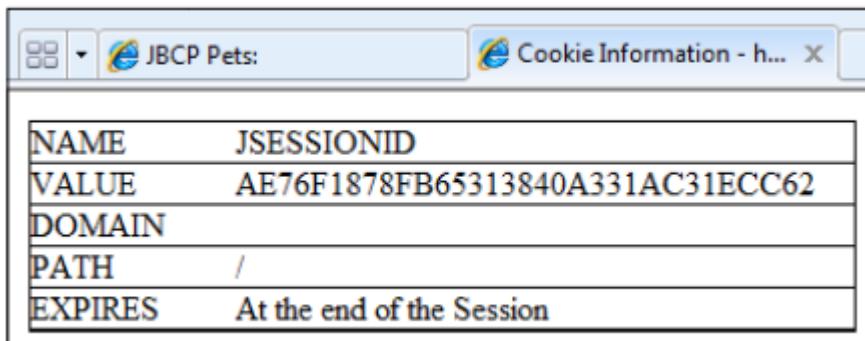
模拟 session fixation 攻击

此时，你可能会想要看一下在模拟 session 固化攻击时会涉及到什么。为了实现这一点，你需要在 `dogstore-security.xml` 中配置 session 固化防护失效。

```
<session-management session-fixation-protection="none"/>
```

接下来，你需要打开两个浏览器。我们将会在 IE 中初始化 session，并从那里窃取，我们的攻击者将会使用窃取到的 session 在 Firefox 中登录。我们将会使用 Internet Explorer Developer Tools（IE 8 中自带）以及 Firefox Web Developer Add-On（第三章中已经给过 URL）来查看和控制 cookie。

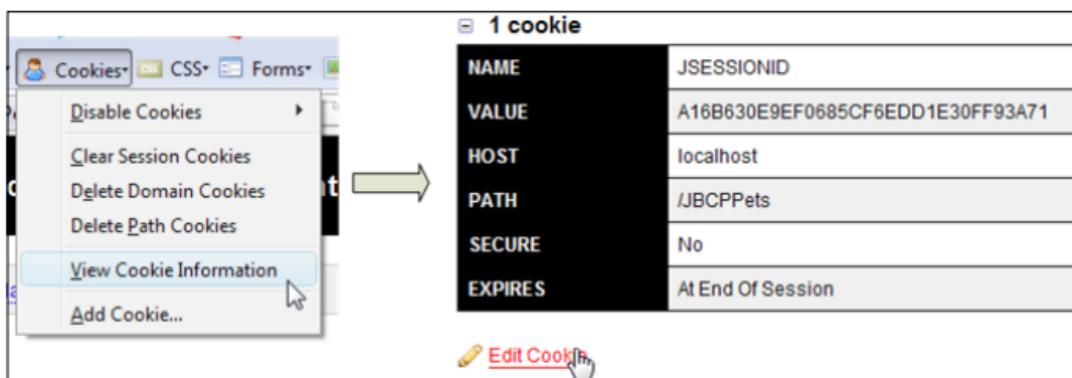
在 IE 中打开 JBCP Pets 首页，然后打开开发者工具（看“工具”下拉菜单或点击 F12），并在“缓存”菜单下选择“查看 Cookie 信息”。在合适域下（如果使用 localhost 将为空）找到 JSESSIONID 的 cookie。



NAME	JSESSIONID
VALUE	AE76F1878FB65313840A331AC31ECC62
DOMAIN	
PATH	/
EXPIRES	At the end of the Session

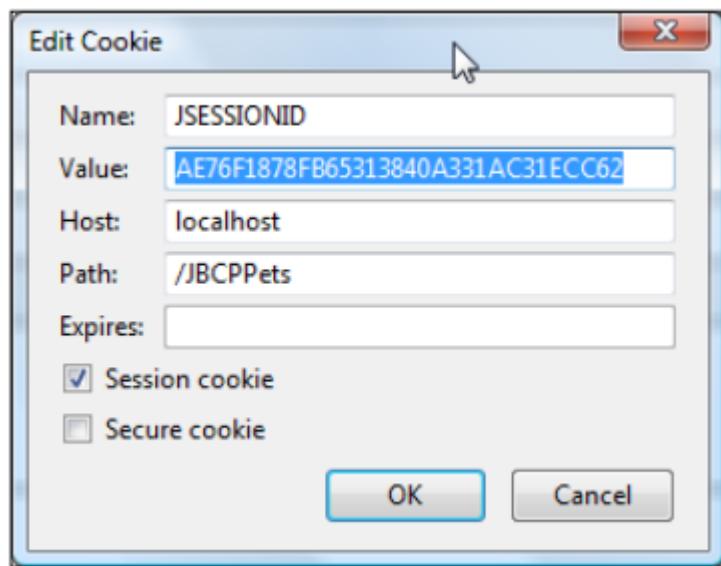
将 session cookie 的值复制到粘贴板上，然后登录 JBCP Pets 站点。如果你重复“查看 Cookie 信息”，你将会发现 JSESSIONID 在登录后没有变化，这将会导致很容易受到 session 固化攻击。

在 Firefox 下，打开 JBCP Pets 站点。你将会被分配一个 session cookie，这能通过 Cookie 菜单的“查看 Cookie 信息”菜单项查看到。



为了完成我们的攻击，我们点击“Edit Cookie”选项，并将从 IE 中复制到粘贴板上的

JSESSIONID 值粘贴进来，如下图所示：



我们的 session 固化攻击完成了！如果此时在 Firefox 中重新加载页面，你将以 IE 中已登录用户相同的身份进入系统，并不需要你知道用户名和密码。你是否害怕恶意用户了？

现在，重新使 session 固化防护生效然后重新尝试这个练习。你会发现，在这种情况下，JSESSIONID 在用户登录后会发生变化。因为你已经了解了 session 固化攻击是如何发生的，这意味着减少了可信任用户陷入这种攻击的风险。干的漂亮！

细心的开发人员应该会注意到有很多种窃取 session cookie 的方式，有一些如跨站脚本攻击（XSS）可能会使得 session 固化防护都很脆弱。请访问 OWASP 站点来了解更多防止这种类型攻击的信息。

比较 session-fixation-protection 选项

session-fixation-protection 属性支持三种不同的选项允许你进行修改：

属性值	描述
none	使得 session 固化攻击失效，不会配置 SessionManagementFilter （除非其它的 <session-management> 属性不是默认值）
migrateSession	当用户经过认证后分配一个新的 session，它保证原 session 的所有属性移到新 session 中。我们将在后面的章节中讲解，通过基于 bean 的方式如何进行这样的配置。
newSession	当用户认证后，建立一个新的 session，原（未认证时） session 的属性不会进行移到新 session 中来。

在大多数场景下，默认行为即 migrateSession 适用于在用户登录后希望保持重要信息（如点击爱好、购物车等）的站点的站点

通过 session 的并发控制增强对用户的保护

紧随 session 固化防护一个很自然的用户安全增强功能就是 session 并发控制。如前所描述的那样，session 并发控制能够确保一个用户不能同时拥有超过一个固定数量的活跃 session（典型情况是一个）。要确保这个最大值的限制需要涉及到好几个组件的协作以精确跟踪用户 session 活动的变化。

让我们配置这个功能并了解其如何工作，然后对其进行测试。

配置 session 并发控制

既然我们要了解 session 并发控制所要涉及的组件，那将其运行环境搭建起来可能会更有感官的了解。首先，我们需要使得 `ConcurrentSessionFilter` 生效并在 `dogstore-security.xml` 配置。

```
<http auto-config="true" use-expressions="true">
<!-- ... -->
<session-management>
    <concurrency-control max-sessions="1"/>
</session-management>
</http>
```

现在，我们需要在 `web.xml` 描述文件中配置中使得 `o.s.s.web.session.HttpSessionEventPublisher` 生效，这样 `servlet` 容器将会通知 Spring Security session 生命周期的事件（通过 `HttpSessionEventPublisher`）。

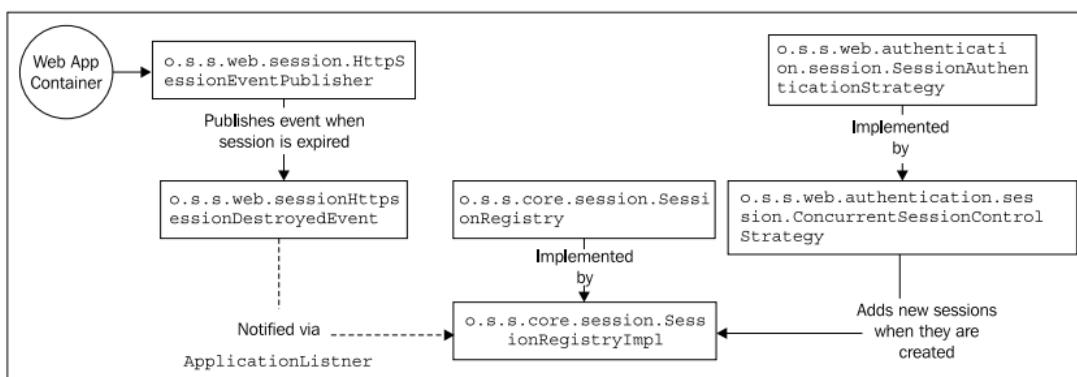
```
<listener>
    <listener-class>
        org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
<listener>
    <listener-class>
        org.springframework.security.web.session
        .HttpSessionEventPublisher
    </listener-class>
</listener>
<servlet>
    <servlet-name>dogstore</servlet-name>
```

这两个配置完成，session 的并发控制功能也就激活了。让我们看一下它内部是如何工作的，然后我们将会通过几步操作来查看它对用户的 session 的保护功能。

理解 session 并发控制

我们在前面提到 session 并发控制试图限制相同的用户以不同的 session 进行访问。基于我们对 session 窃取方式攻击的了解，我们可以发现 session 并发控制能够降低攻击者窃取已登录合法用户 session 的风险。你觉得为什么会这样呢？

Session 并发控制使用 `o.s.s.core.session.SessionRegistry` 来维护一个活跃 HTTP session 的列表而认证过的用户与其进行关联。当 session 创建或过期时，注册表中会实时进行更新，基于 `HttpSessionEventPublisher` 发布的 session 生命周期事件来跟踪每一个认证用户的活动 session 的数量。



SessionAuthenticationStrategy 的一个扩展类即 `o.s.s.web.authentication.session.ConcurrentSessionControlStrategy` 提供方法来实现新 session 的跟踪以及 session 并发控制的实际增强功能。每次用户访问这个安全站点时，`SessionManagementFilter` 将会比照 `SessionRegistry` 检查这个活跃的 session。如果用户活跃的 session 不在 `SessionRegistry` 这个活跃 session 列表中，最近最少被使用的 session 将会立即过期。

在修改后的 session 并发控制过滤器链中的第二个参与者是 `o.s.s.web.session.ConcurrentSessionFilter`。这个过滤器能够辨认出过期的 session（典型情况下，session 会被 servlet 容器或者被 `ConcurrentSessionControlStrategy` 强制失效掉）并通知用户他的 session 已经失效了。

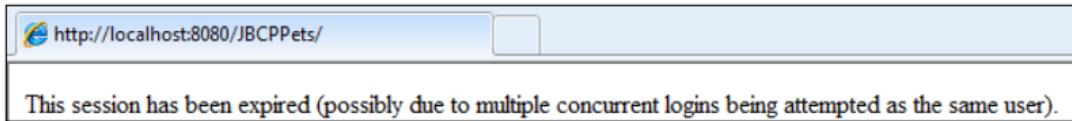
既然我们已经了解了 session 并发控制是如何工作的，那对我们来说很容易制造一个它使用的场景。

测试 session 并发控制

如同验证 session 固化攻击那样，我们需要访问两个 web 浏览器。按一下的步骤：

1. 在 IE 中，以 guest 用户登录；
2. 接下来，在 Firefox 中，以相同的用户（guest）登录；
3. 最后，返回到 IE 中，做任何的动作都可以。你会发现有一个信息提示你的 session 已经过期了。

将会显示以下的信息：



尽管不很友好，但是它能够表明 `session` 已经被软件强制失效了。如果你是一个攻击者，现在你可能会很灰心。但是，如果你是一个合法的用户，你可能会很迷惑，因为这显然不是一个友好的方式来表明 JBCP Pets 一直关注着你的安全。

【`session` 并发控制对 Spring Security 的新用户来说是很难掌握的概念。很多用户在还没有完全理解其怎样运行和能带来什么好处时就尝试实现它。如果你想使用这个强大的功能，并且它不像你预想的那样工作，请确保你的配置全部正确并回顾一下本节讲的原理——希望它能帮助你理解什么出错了。】

在这样的事件发生时，我们应该将用户重定向到登录页，并提供一个信息来说明发生了什么错误。

配置 `session` 失效时的重定向地址

幸运的是，有一种很容易的方法使用户在 `session` 并发控制后重定向到友好的页面（一般来说，是登录页），即设置 `expired-url` 属性为一个你应用中合法的页面。

```
<http auto-config="true" use-expressions="true">
<!-- ... -->
<session-management>
    <concurrency-control max-sessions="1" expired-url= "/login.do?error=expired"/>
</session-management>
</http>
```

这样在我们的应用中，就会将用户重定向到登录 form，并且我们可以修改这个页面来展现用户友好的信息来表明发现了多个活跃的 `session`，从而需要重新登录。当然，这个 URL 是完全随意的，并根据你应用的需求来进行相应的调整。

Session 并发控制的其它好处

Session 并发控制的另一个好处是存在 `SessionRegistry` 跟踪活跃的 `session`（过期 `session` 是可选的）。这意味着我们能够得到系统中运行时的用户活动信息（至少是认证过的用户）。

即使你不想使用 `session` 并发控制，你可以这样做。只需将 `max-sessions` 的值设置为-1，这样 `session` 跟踪会保持可用，但没有最大 `session` 个数的限制。

让我们看两个使用此功能的两种简单方式。

显示活动的用户

你可能会在线论坛上见到显示系统中当前活跃用户的数量。借助于使用 `session` 注册跟踪（通过 `session` 并发控制），很容易实现在应用中的每个页面对此进行展现。

让我们在 `BaseController` 中添加一个简单的方法以及 bean 自动织入。

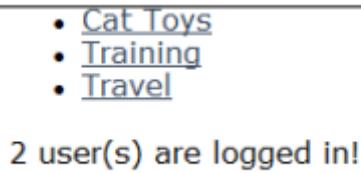
```
@Autowired
```

```
SessionRegistry sessionRegistry;
@RequestMapping("numUsers")
public int getNumberOfUsers() {
    return sessionRegistry.getAllPrincipals().size();
}
```

我们可以看到这暴露了一个能够在 Spring MVC JSP 页面中能够使用的属性，所以我们添加一个页脚 footer.jsp 到 JBCP Pets 站点中并使用这个属性。

```
<div id="footer">
    ${numUsers} user(s) are logged in!
</div>
</body>
</html>
```

如果你重新启动应用并登录，能够在每个页面的底部看到活动用户的数量。



很简单，但是它阐述了作为 Spring Security 一部分所提供的 session 跟踪的好处——尤其是你能够直接使用这个内置的功能。

我们能够借助于 SessionRegistry 进行更高级的数据收集，这对于管理员来说很有用——现在让我们看一下。

显示所有用户的信息

SessionRegistry 跟踪所有活跃用户 session 的信息。如果你想增强站点的管理，我们可以很容易地在一个页面中列出所有的用户活跃用户以及他们在站点中使用的名字。

让我们为 AccountController 添加一个新的方法（尽管这样的功能一般会添加在在管理区域，但是此时我们可以假设 JBCP Pets 并不是一个真正的站点），这个方法将会查找 SessionRegistry 中的信息并收集当前 session 的信息。

```
@RequestMapping("/account/listActiveUsers.do")
public void listActiveUsers(Model model) {
    Map<Object, Date> lastActivityDates = new HashMap<Object, Date>();
    for(Object principal: sessionRegistry.getAllPrincipals()) {
        // a principal may have multiple active sessions
        for(SessionInformation session : sessionRegistry.getAllSessions(principal, false)) {
            if(lastActivityDates.get(principal) == null) {
                lastActivityDates.put(principal, session.getLastRequest());
            } else {
                // check to see if this session is newer than the last stored
            }
        }
    }
}
```

```
Date prevLastRequest = lastActivityDates.get(principal);
if(session.getLastRequest().after(prevLastRequest)) {
    // update if so
    lastActivityDates.put(principal, session.getLastRequest());
}
}
}
}
model.addAttribute("activeUsers", lastActivityDates);
}
```

这个方法使用了 SessionRegistry 的两个 API。

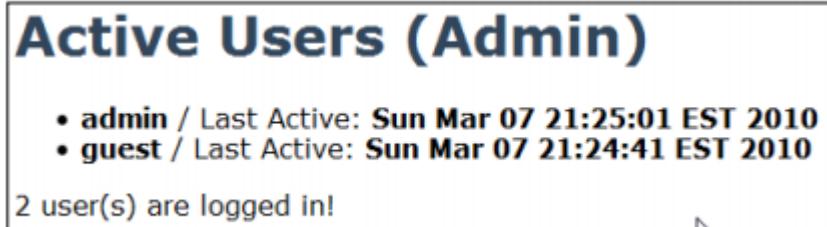
- `getAllPrincipals`: 返回拥有活跃 session 的 Principal 对象（典型情况下为 UserDetails 对象）所组成的 List;
- `getAllSessions(principal, includeExpired)`: 得到指定 Principal 的 SessionInformation 组成的 List，包含了每个 session 的信息。也能够包含过期的 session。

了解了 SessionRegistry API 方法，`listActiveUsers` 方法的逻辑就很简单了——检索注册表中所有的活跃用户并寻找最近活动的 session。Principal 以及最近活跃的时间戳信息插入到一个 Map 中用来在 UI 中展现。

UI 页面通过使用 JSTL 结构变得很简单了。在 WEB-INF/views/account 目录下，创建 `listActiveUsers.jsp`，内容如下（简便起见，我们省略了头部和尾部信息）：

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<h1>Active Users</h1>
<ul>
    <c:forEach items="${activeUsers}" var="uinfo">
        <li><strong>${uinfo.key.username}</strong>
        / Last Active: <strong>${uinfo.value}</strong></li>
    </c:forEach>
</ul>
```

最后，当我们导航到 <http://localhost:8080/JBCPPets/account/listActiveUsers.do>，页面基本如下：

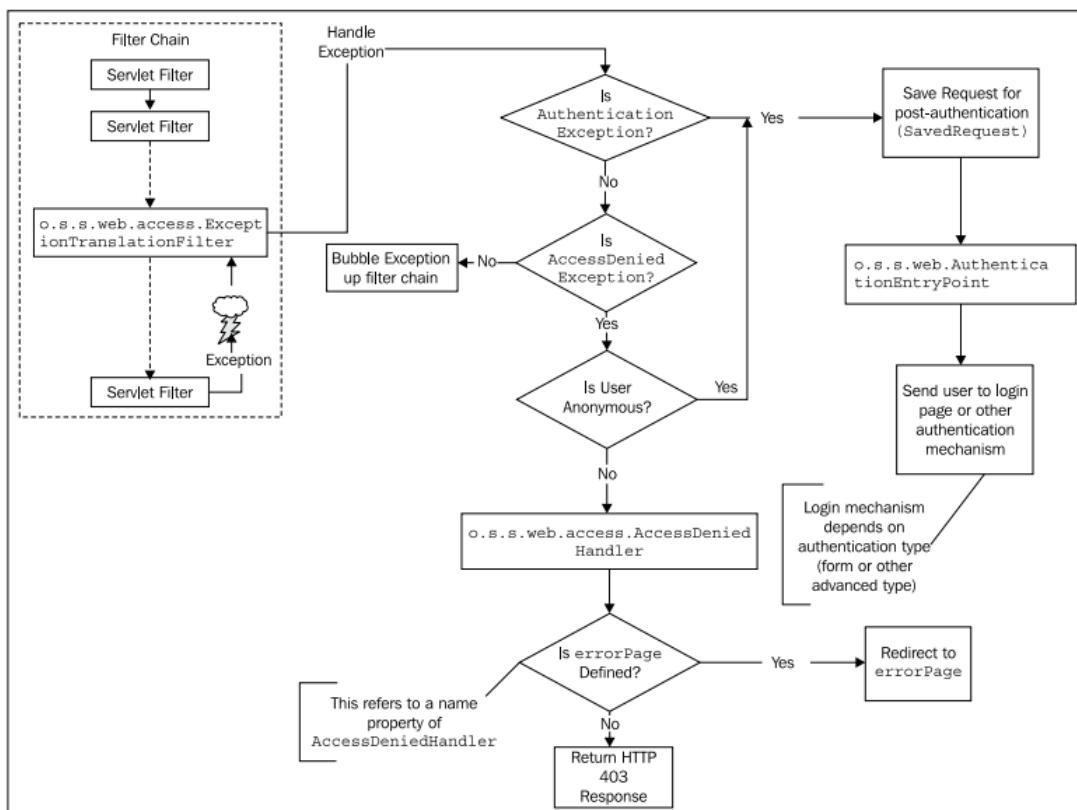


通过这些例子，你已经了解到了 SessionRegistry 的威力。我们甚至可以扩展 SessionRegistry 来跟踪用户活动的附加信息，如最后访问的页面、最后的行为等——这对基于 Spring Security 构建管理界面来说是很有用的。

理解和配置异常处理

Spring Security 使用简单的分发器模式将框架抛出的异常转移到明确的处理行为中，这将会影响用户对安全资源的访问。Spring Security 过滤器链中最后几个过滤器之一的 o.s.s.web.access.ExceptionTranslationFilter 负责检查在认证和授权过程中（在过滤器链的最后一个过滤器即 FilterSecurityInterceptor 里）抛出的异常并采取适当的行为。

标准的 ExceptionTranslationFilter 支持分发处理三种常规类型的失败，如下图所示：



我们能够看到 ExceptionTranslationFilter 处理如下的场景：

- 抛出 AuthenticationException 异常，用户需要登录（在大多数场景下——取决于 AuthenticationEntryPoint，我们将会在本章稍后介绍）；
- 抛出 AccessDeniedException，用户已经尚未登录；
- 抛出 AccessDeniedException，用户已经登录，在这种场景下，展现给用户的要么是出错页面，要么是通用的 HTTP 403 响应。

让我们看一下 AccessDeniedHandler 的配置。

配置“Access Denied”处理

到此为止，当一个认证过的用户访问受保护的资源时，因为缺少 GrantedAuthority 或其它需要的权限被拒绝的时候，他们看到的是 servlet 容器的默认 HTTP 403（访问拒绝）页面。这个页面是 o.s.s.web.access.AccessDeniedHandler 默认行为的结果，它被 ExceptionTranslationFilter 所触发以响应框架抛出的一个 AccessDeniedException 异常。

尽管这个简单的出错页面有效，但是并不具有吸引力和也不对用户友好。最好能够将这个页面与我们站点整体的外观和风格一致，并为用户提供信息告诉他发生了什么。

基本的处理用户访问拒绝报告的方法是配置自定义的 URL，Spring Security 会在请求拒绝时，将用户带到这个 URL。我们会发现这个功能与初始登录时，用户被定向到`<form-login>`元素声明的 `login-page` 很类似。

配置“Access Denied”的目标地址

配置用户被定向到的地址是这个练习中最简单的部分。只需在`<http>`声明中，添加一个元素`<access-denied-handler>`，它指明了我们的访问拒绝处理 URL，如下：

```
<http auto-config="true" ...>
    <access-denied-handler error-page="/accessDenied.do"/>
</http>
```

注意——还没完事呢，因为我们还没有将这个 URL 与任何的 Spring MVC 应用代码关联起来。我们需要在 `LoginLogoutController` 添加增强代码以处理这个 URL，并推送一些有用的信息到 model 中供 view 展现给用户。

添加对 `AccessDeniedException` 处理的控制器

我们需要添加一个控制器 `action` 处理方法以响应刚刚配置的 URL。另外，我们会从 `AccessDeniedException` 抽取一些细节信息，这可能在用户看到我们自定义访问拒绝页面时有用。

```
@Controller
public class LoginLogoutController extends BaseController{
    // Ch 6 Access Denied
    @RequestMapping(method=RequestMethod.GET, value="/accessDenied.do").
    public void accessDenied(ModelMap model, HttpServletRequest request) {
        AccessDeniedException ex = (AccessDeniedException)
            request.getAttribute(AccessDeniedHandlerImpl
                .SPRING_SECURITY_ACCESS_DENIED_EXCEPTION_KEY);
        StringWriter sw = new StringWriter();
        model.addAttribute("errorDetails", ex.getMessage());
        ex.printStackTrace(new PrintWriter(sw));
        model.addAttribute("errorTrace", sw.toString());
    }
}
```

注意的是，需要引用 `AccessDeniedHandlerImpl` 来获取 `request` 中指定名字的属性，它被用来临时存储当前 `request` 范围内的异常。

遗憾的是，`AccessDeniedException` 并没有在它的 `message` 中提供足够的细节信息，而这对系统管理员和用户本身可能会有用。你可能会对使用 Spring Security 的 `AccessDeniedException` 感兴趣或者可能扩展它以提供更多的上下文信息，以得到当授权检查不通过时用户正在试图进行什么操作。

编写 Access Denied 页面

控制器写好后，接下来是访问拒绝页面。

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1"%>
<jsp:include page="common/header.jsp">
    <jsp:param name="pageTitle" value="Access Denied"/>
</jsp:include>
<h1>Access Denied</h1>
<p>
    Access to the specified resource has been denied for
    the following reason: <strong>${errorDetails}</strong>.
</p>
<em>Error Details (for Support Purposes only):</em><br />
<blockquote>
    <pre>${errorTrace}</pre>
</blockquote>
<jsp:include page="common/footer.jsp"/>
```

你可以看到我们使用了在控制器设置的模型属性 `errorDetails` 和 `errorTrace`。尽管不是很漂亮，但是这个页面完成了它的任务即用通用的站点导航，将用户带到了站点的其它区域并给他们提供了导致出错的提示信息。

什么会触发 AccessDeniedException

当设计异常处理时，进行一些分析并理解目标异常的原理很重要。对于 Spring Security 的用户来说，通常很迷惑的一件事就是 `AccessDeniedException`（默认会导致 HTTP 403 页面）和 `AuthenticationException`（一般当用户根本没有登录时抛出）。以下的指南可能会帮助你分清框架在什么时间抛出每种类型的异常：

异常类型	谁抛出以及原因
<code>AuthenticationException</code>	<code>AuthenticationProvider</code> , 当提供的凭证不合法或用户失效、过期; <code>DaoAuthenticationProvider</code> , 当访问 DAO 数据存储时出错; <code>RememberMeServices</code> , 当 <code>remember me cookie</code> 被篡改; 各种特定的认证类（CAS、NTLM 等）在用户特定的场景下。
<code>AccessDeniedException</code>	<code>AccessDecisionManager</code> , 当配置的 Voter 投票拒绝访问——注意这可能在任何投票场景下

要记住的是，我们前面提到的 `ExceptionTranslationFilter` 是区分这两种类型异常的关键点，因为这关系到应用用户的请求和响应流程。

【注意过滤器链中在 `ExceptionTranslationFilter` 之前的那些过滤器。`ExceptionTranslationFilter`

只会处理和响应过滤器链中在此之后的过滤器所抛出的异常。用户可能会感到迷惑，尤其是将自定义过滤器进行了不正确排序的时候，不明白为什么期望的行为与应用实际的异常处理不一致——在很多场景下，过滤器的顺序是原因所在。】

尽管内置的处理流程在大多数情况下是可预测且满足需要的，但有时候你会需要自定义异常处理，尤其是在引入基类异常的自定义子类时，这需要过滤器链的特殊处理。

AuthenticationEntryPoint 的重要性

AuthenticationEntryPoint（在 ExceptionTranslationFilter 我们看到它是工作流程中的一个辅助类）在处理未认证用户请求中很重要。当 ExceptionTranslationFilter 确定用户需要认证时，它请求 AuthenticationEntryPoint 以了解下一步要做什么。在基于 form 的认证中，o.s.s.web.authentication.LoginUrlAuthenticationEntryPoint 负责将用户定向到登录 form。

我们将会在后面的章节中看到 AuthenticationEntryPoint 被用在各种认证机制中，从而实现更个性化的行为——例如，在中心认证服务（CAS）单点登录中，AuthenticationEntryPoint 要确保用户被定向到 CAS 门户进行认证。

在很多环境下，当实现 Spring Security 与第三方认证系统（独立于 web 应用）集成时，你需要实现自己的 AuthenticationEntryPoint。

手动配置 Spring Security 设施的 bean

如果你工作要求的环境很复杂而 Spring Security 的基本功能——尽管非常强大——不能满足所有的要求，你可能最终需要自己从头构建 Spring Security 的过滤器链以及支持实施。这是在 Spring Security 参考手册中没有完全提及的部分，但是却难住了很多人。有些人将这种类型的配置成为 Spring Security 的“另一个宇宙（alternate universe）”。

自己构建并织入所有需要的 bean，将为你提供很高的灵活性和自定义功能，这是通过 security 命名空间的<http>风格配置所不允许的。

【我们不是开玩笑地说构建所需的 bean 是很复杂的。即使一个必须的 bean 都可能需要多达 25 个单独的 bean。, 这需要开发人员明确理解 bean 之间的依赖关系以及每个 bean 的所有属性。记住，一旦你不再使用基于 XML 命名空间的便利配置方式，就会与 Spring Security 代码和整体结构更密切相关了。Security XML 命名空间提供了一层很受欢迎的抽象并能很好地满足大多数的需求。】

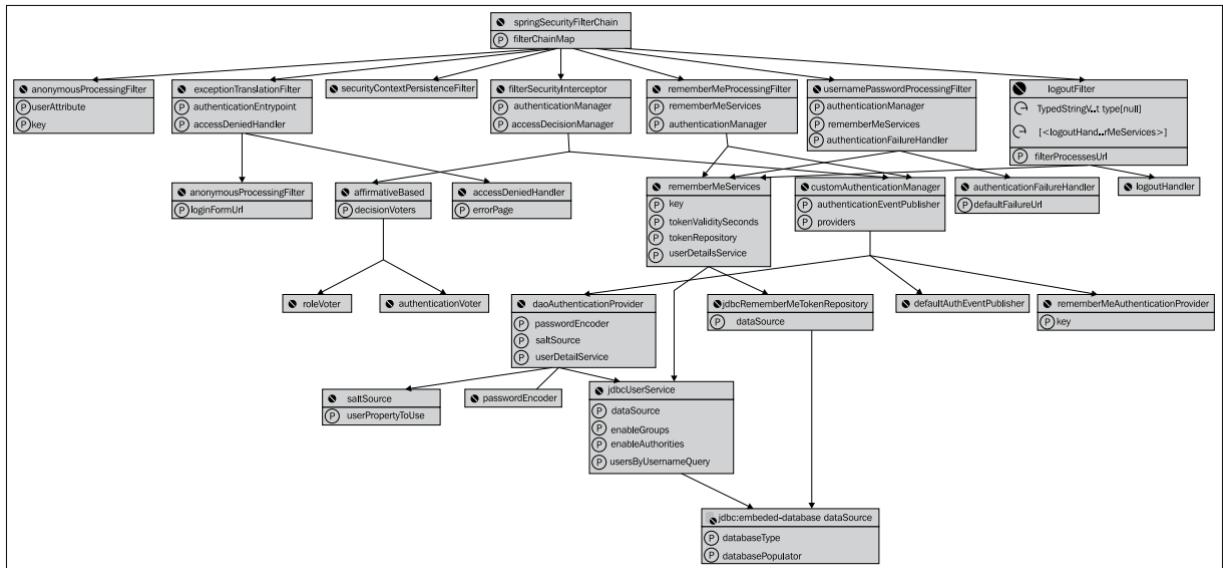
希望到本书的这个地方，你已经理解了请求处理架构和相关的主要组件，这样的话，面对大量需要配置的 bean 才不会感到吃惊。完全理解表面下所有的组件，对于配置每个 bean 来说是很有用的。

我们将会花些时间来介绍自己搭建 Spring Security 基础设施时所需要的主要组成部分。注意的是在有些场景下，当没有必要进行阐述时，我们将会省略一些没有意思 bean 的细节；但是，完整的配置文件（在本章源码中的 dogstore-explicit-base.xml 中）需要支持它。现在让我们进入主要的配置过程。

总体理解 Spring Security bean 的依赖关系

我们不想马上扎入到配置 bean 中，而是要给你一个总体了解手动建立 Spring Security

bean 时所涉及的主要组件。以下是一个依赖图展现了我们要配置的 bean 以及它们怎样交互的（这个图在本章的源码中包含完整尺寸的，作为参考）。



需要记住的是，这个图中包含的 bean 超过了是系统启动和运行所需要的最少值。我们将会渐进的阐述怎样添加所有的 bean，从最小的集合开始，并逐渐构建出与用 security 命名空间相匹配的功能。

重新配置 web 应用

为了表述清晰，我们为这种风格的配置创建一个全新的 XML 配置文件。这样我们能够很清晰地看到什么才是明确需要的，并移除我们在前面章节中注释过的和没注释过的一些部分。

为了做到这些，我们需要重新配置 Spring 的 ApplicationContext 指向这个新的文件。我们将这个文件命名为 dogstore-explicit-base.xml，并更新 web.xml 文件指向它，如下：

```
<web-app ...>
  <display-name>Dog Store</display-name>
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
      /WEB-INF/dogstore-explicit-base.xml
    </param-value>
  </context-param>
```

我们不再需要单独 dogstore-security.xml 文件了，这个文件是我们为了使用 XML security 命名空间声明而创建的。我们大多数的配置将会使用 Spring 标准的 bean 注入语法，但有少量的 security 命名空间装饰器在里面。

配置一个最小的 Spring Security 环境

我们将会以能使系统重新运行起来的最小的配置开始——这意味着没有 remember me、logout 以及异常处理功能。这使得我们可以聚焦于 Spring Security 启动的最小需求。

首先，我们需要声明 Spring Security 拦截请求时所使用的 servlet 过滤器链，如下：

```
<bean id="springSecurityFilterChain"
    class="org.springframework.security.web.FilterChainProxy">
    <security:filter-chain-map path-type="ant">
        <security:filter-chain pattern="/**" filters="
            securityContextPersistenceFilter,
            usernamePasswordAuthenticationFilter,
            anonymousAuthenticationFilter,
            filterSecurityInterceptor" />
    </security:filter-chain-map>
</bean>
```

你可以看到这里我们已经引用了 security 命名空间。尽管可以使用手动的方式来配置需要 bean 的属性以建立路径模式匹配和过滤器列表组合，但是使用 security 命名空间的 filter-chain-map 包装器更简便和便利。如果将这与<http>风格的配置进行对比的话，我们要注意以下的配置元素：

- 默认过滤器链的建立是在处理<http>元素的时候自动包含的并不需要直接配置。尽管使用 security 命名空间的<custom-filter>重写或扩展标准过滤器链的时候，允许很大程度的灵活性，但它并不能够得到 FilterChainProxy 本身。
- 基于 URL 模式修改过滤器链并不适用于<http>风格的声明。如果应用的某些部分不需要特定的处理这将会有用处，并且能使得过滤器的调用尽可能得少。

需要意识到很重要的一点是，不同于 Spring 的一些配置（比较明显的如，在 web.xml 中的 contextConfigLocation），在过滤器的名字之间需要使用逗号分隔。

【过滤器的顺序很重要——正如我们在第二章中所讨论的那样，特定的过滤器必须在另一些的前面。除非你有特殊的需求，请参考第二章中的表格，当你添加标准的过滤器到一个手动配置的过滤器链中时，要保证它们在合适的位置。过滤器被包含在不正确的位置可能会导致不可预知的应用行为，而这是很难调试的。】

你会意识到<filter-chain>元素引用了很多逻辑 bean definitions，而它们还没有进行定义。现在对它们进行定义，并逐个进行详细介绍。

配置最少的 servlet 过滤器集合

为了支持上面描述的过滤器链，我们要设置两类的对象。

首先，servlet 过滤器本身必要要进行设置。它们定义了进入 web 应用的用户请求是如何处理的——与使用 security 命名空间不同的是我们更接近底层本质，并需要明确定义在以前简单配置时隐藏在背后的过滤器。

其次，servlet 过滤器依赖一系列提供支持功能的安全基础设施 bean。它们中的一些类对你来说可能比较熟悉，因为我们从第一章到第五章已经从架构和功能性的视角讲到了它们，

但是这些 bean 的配置方式是全新的。

我们将从需要的过滤器开始。

SecurityContextPersistenceFilter

SecurityContextPersistenceFilter 用来建立 SecurityContext，而它被用来贯穿整个 request 过程以跟踪请求者的认证信息。你可能记得我们在上一章的 Spring MVC 代码中，为了得到当前认证过的 Principal 时，访问过 SecurityContext 对象。

包含默认适当 web session 管理的 SecurityContextPersistenceFilter 基本配置如下：

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

在这个场景背后有许多的东西，我们可以根据要求使这个配置很复杂，如 HTTP session 如何管理。现在，我们使这个过滤器采用默认设置然后继续，但是在本章的后面部分我们将会涉及 session 相关的配置选项。

UsernamePasswordAuthenticationFilter

正如我们在第二章中详细介绍的那样，UsernamePasswordAuthenticationFilter 用来处理 form 提交并检查认证存储是否为合法凭证。明确配置这个过滤器，对比 security 命名空间的配置，如下：

```
<bean id="UsernamePasswordAuthenticationFilter"
      class="org.springframework.security.web
      .authentication.UsernamePasswordAuthenticationFilter">
    <property name="authenticationManager" ref="customAuthenticationManager"/>
</bean>
```

如果你有时间深入研究这个类你会发现它有一个范式（译者注：原文为 pattern，个人理解是用户名、密码、url 等配置）——其实这里还有更多的可配置内容。同样的，在基本配置能够运行后，我们将会再次介绍它。我们引用了一个名为 customAuthenticationManager 的 bean——这就是与使用 security 命名空间的<authentication-manager>元素自动配置相同的 AuthenticationManager。我们稍后将会配置这个 bean。

AnonymousAuthenticationFilter

我们的站点允许匿名访问。尽管对于比较特殊的条件 AnonymousAuthenticationFilter 并不需要，但是通常情况下会使用它，因为只对请求添加了一点的预处理。你可能并不认识这个过滤器，除了我们在第二章对其简短提到以外。这是因为对于 AnonymousAuthenticationFilter 的配置都掩盖在 security 命名空间之中。

这个过滤器的最小配置如下：

```
<bean id="anonymousAuthenticationFilter"
      class="org.springframework.security.web
      .authentication.AnonymousAuthenticationFilter">
    <property name="userAttribute"
```

```
value="anonymousUser,ROLE_ANONYMOUS"/>
<property name="key" value="BF93JFJ091N00Q7HF"/>
</bean>
```

列出的这两个属性都是需要的。`userAttribute` 属性声明了为匿名用户提供的用户名和 `GrantedAuthority`。用户名和 `GrantedAuthority` 可能在我们的应用中用来验证用户是不是匿名用户。Key 可能是随机生成的，但是需要在一个 bean 中使用 (`o.s.s.authentication.AnonymousAuthenticationProvider`)，我们稍后将会进行配置。

FilterSecurityInterceptor

在我们基本处理过滤器链的最后一个最终负责检查 `Authentication` 的，而这是前面已配置的安全过滤器的处理结果。正是这个过滤器确定一个特定的请求最终是被拒绝还是被接受。

让我们看一下这个过滤器的配置并了解这个练习与其对应的 security 命名空间进行比较。

```
<bean id="filterSecurityInterceptor"
    class="org.springframework.security.web.access.intercept.FilterSecurityInterceptor">
    <property name="authenticationManager" ref="customAuthenticationManager"/>
    <property name="accessDecisionManager" ref="affirmativeBased"/>
    <property name="securityMetadataSource">
        <security:filter-security-metadata-source>
            <security:intercept-url pattern="/login.do"
                access="IS_AUTHENTICATED_ANONYMOUSLY"/>
            <security:intercept-url pattern="/
                home.do" access="IS_AUTHENTICATED_ANONYMOUSLY"/>
            <security:intercept-url pattern="/
                account/*.do" access="ROLE_USER"/>
            <security:intercept-url pattern="/*" access="ROLE_USER"/>
        </security:filter-security-metadata-source>
    </property>
</bean>
```

在继续阅读之前请思考一下。没错，它看起来和我们在使用 security 命名空间的`<http>`配置的`<intercept-url>`声明一样。模式匹配和访问声明格式完全一致，但是你会发现在本例中我们使用了一些配置魔法来使用相同的声明在上下文中建立正常的 Spring bean 属性。

`<filter-security-metadata-source>` 元素负责配置 `FilterSecurityInterceptor` 会用到的 `SecurityMetadataSource` 的实现，包含 URL 声明以及可以访问所需要的角色。

【有效使用 XML 命名空间】对于不熟悉 Spring 高级配置的用户来说到这里通常会比较迷惑。在配置文件中，Spring XML 配置有效使用了 XML 命名空间来提供清晰的基于组件所有的不同元素。标示元素在一个命名空间中需要以一个冒号 (:) 做前缀，后面跟着元素的名字。所以如果我们看`<security:intercept-url>`，我们可以看到这个元素的名字是 `intercept-url`，并在名为 `security` 的 XML 命名空间里。XML 命名空间的通常在 XML 文件的顶部声明，带有一个任意的前缀设置并关联一个 URI。例如，`security` 命名空间可以声明为 `xmlns:security=http://www.springframework.org/schema/security`。没有声明命名空间的元素呢？它们将被分配为 XML 文档的默认命名空间。默认的命名空间通过 `xmlns` 属性来标示——

在 `dogstore-explicit-base.xml` 文件中，默认的命名空间被声明为 `xmlns="http://www.springframework.org/schema/beans"`。没有命名空间前缀的元素将会被自动设置为默认的命名空间。理解 XML 命名空间实际如何运行将会使你在构建复杂 Spring 和 Spring Security 配置文件时免去很多头疼之苦。】

我们已经为明确设置的最小化过滤器链配置完了所有过滤器。这些过滤器不能独立存在——还有几个需要的支持对象。

配置最少的支持对象集合

为了建立最小配置所需的支持对象是我们在前面的章节中已经配置过的（除了一个）Spring bean——所以我们将会节省一些时间来介绍它们的用处。

以下为一系列的 bean 定义，它们是为了完成最小的支持对象集合和启动应用的：

```
<bean class="org.springframework.security.access.vote.AffirmativeBased" id="affirmativeBased">
    <property name="decisionVoters">
        <list>
            <ref bean="roleVoter"/>
            <ref bean="authenticatedVoter"/>
        </list>
    </property>
</bean>
<bean class="org.springframework.security.access.vote.RoleVoter" id="roleVoter"/>
<bean class="org.springframework.security.access.vote.AuthenticatedVoter" id="authenticatedVoter"/>
<bean id="daoAuthenticationProvider" class="org.springframework.security.authentication.dao.DaoAuthenticationProvider">
    <property name="userDetailsService" ref="jdbcUserService"/>
</bean>
<bean id="anonymousAuthenticationProvider" class="org.springframework.security.authentication.AnonymousAuthenticationProvider">
    <property name="key" value="BF93JFJ091N00Q7HF"/>
</bean>
```

这个配置为我们提供了一个最小的配置支持匿名浏览、登录以及数据库后台的认证（注意的是为了简洁我们省略了需要的 `jdbcUserService` 和 `dataSource` beans——这些 bean 的定义与前面的定义没有什么变化）。记住，`AnonymousAuthenticationProvider` 的 `key` 属性必须与我们前面定义的 `AnonymousAuthenticationFilter` 的 `key` 属性相匹配。

有一个 bean 需要，但是我们前面从来没有配置过的（这是因为 security 命名空间配置不允许这样做）就是 `AuthenticationManager`。我们可以定义这个 bean 如下：

```
<bean id="customAuthenticationManager" class="org.springframework.security.authentication.ProviderManager">
    <property name="providers">
        <list>
```

```
<ref local="daoAuthenticationProvider"/>
<ref local="anonymousAuthenticationProvider"/>
</list>
</property>
</bean>
```

AuthenticationManager 的配置在这里看起来很简单，但是明确配置这个 bean 是很多有用增强和扩展框架的关键，这些我们将会在本章的剩余部分讲解。

在手动配置 bean 的所有工作完成后，我们的站点依旧不支持我们用 security 配置时的一些功能，包括退出功能、友好的异常处理、密码 salting 以及 remember me。所以，这些为什么还有价值呢？（译者注：作者的这个反问应该指的是手动配置的意义在哪里。）

Spring Security 基于 bean 的高级配置

正如我们在前面几页中看到的那样，基于 bean 的 Spring Security 配置尽管比较复杂，但是提供了一定层次的灵活性，如果复杂应用需要超过 security XML 命名空间风格配置所允许的功能时会用到。

我们将利用这个章节来阐明可用的一些配置选项以及怎么使用。尽管我们不能提供每个可能属性的细节，但是我们鼓励基于在本章和以前章节学到的内容你开始探索并查询 Javadoc 和 Spring Security 的源码。你仅仅会受限于对这个框架是如何组织起来的知识和你的想象力。

Session 生命周期的调整元素

Spring Security 有很多地方影响用户的 HttpSession 的生命周期。有很多功能只有将相关类配置成 Spring bean 时才可用。以下的表格列出了能够影响 session 创建和销毁的 bean 属性：

Class (类)	属性	默认值	描述
AbstractAuthenticationProcessingFilter (UsernamePasswordAuthenticationFilter 的父类)	allowSessionCreation	true	如果为 true，当认证失败时创建一个新的 session (存储异常)
UsernamePasswordAuthenticationFilter	allowSessionCreation	true	如果为 true 的话，这个特殊的过滤器将会创建一个 session 存储最后尝试的用户名。
SecurityContextLogoutHandler	invalidateHttpSession	true	如果为 true，HttpSession 将会失效 (参考 Servlet 规范了解 session 失效的细节)
SecurityContextPersistenceFilter	forceEagerSessionCreation	false	如果为 true，该过滤器将会在执行链

			中其它过滤器之前 创建一个 session。
HttpSessionSecurityContextRepository	allowSessionCreation	true	如果为 true，如果在请求结束时 session 中还没有 SecurityContext 的话，SecurityContext 将存储到 session 中。

取决于你应用的需要，需要审慎分析用户 session——包括认证的和非认证的——并调整相应的 session 生命周期。

手动配置其它通用的服务

还有一些其它的服务，我们已经在 security 命名空间自动配置中使用了。让我们将其添加上，这样就会有一个完整的基本配置实现。

声明缺失的过滤器

我们将要增强手动配置的过滤器链，添加三个我们还没有配置的服务。包含处理退出、remember me 以及异常转换。一旦我们完成这些过滤器，我们将会有完整的功能并可以在它启动时深入了解一些有趣的配置选项。

以下缺失的过滤器将会添加到我们的过滤器链中：

```
<bean id="springSecurityFilterChain"
    class="org.springframework.security.web.FilterChainProxy">
    <security:filter-chain-map path-type="ant">
        <security:filter-chain pattern="/**" filters="
            securityContextPersistenceFilter,
            logoutFilter,
            usernamePasswordAuthenticationFilter,
            rememberMeAuthenticationFilter,
            anonymousAuthenticationFilter,
            exceptionTranslationFilter,
            filterSecurityInterceptor" />
    </security:filter-chain-map>
</bean>
```

正如在前面做的那样，现在我们需要声明这些 Spring bean 并配置他们以及所有依赖的服务 bean。

LogoutFilter

LogoutFilter 的基本配置（它默认用来响应虚拟 URL /j_spring_security_logout）如下：

```
<bean id="logoutFilter" class="org.springframework.security
.web.authentication.logout.LogoutFilter">
    <!-- the post-logout destination -->
    <constructor-arg value="/" />
    <constructor-arg>
        <array>
            <ref local="logoutHandler"/>
        </array>
    </constructor-arg>
    <property name="filterProcessesUrl" value="/logout"/>
</bean>
```

你会发现构造 LogoutFilter 的方法与其它的认证过滤器都不一样，使用了构造方法而不是 bean 属性。第一个构造参数是用户退出后要转向的 URL，第二个参数是对 LogoutHandler 实例的引用。

【依赖注入中的构造器哲学。如果你使用 Spring 有一段时间了，可能已经讨论过以适当的面向对象方式管理需要的依赖。尽管 setter 方式对 Spring 的用户来说很便利，但是追求纯正面向对象的人经常会认为如果一个依赖影响到类的功能，那它应该是构造方法签名的一部分（思考一下在 Spring 以前你是怎样写类的）。鉴于 Spring Security 已经发展了有些年头，不同的作者对此有不同的观点，而我们碰巧正好遇到一个或两个这方面的不一致。两种风格的依赖模型都可以——到底使用哪种风格其实取决于你喜欢哪种类型的需求依赖模型。】

我们还要声明一个简单的 LogoutHandler 的如下：

```
<bean id="logoutHandler" class="org.springframework.security
.web.authentication.logout.SecurityContextLogoutHandler"/>
```

你可能认出这个 LogoutHandler 在我们前几页关于 session 处理的表格中出现过。一个明确配置 logout 处理器的好处就是你能修改用户退出时默认的 session 处理行为。一旦我们完成这两点的配置，Log Out 链接就能够再次开始工作了。

请记住在第三章 security 命名空间配置中，我们曾经修改应用使用一个不那么明显绑定 Spring Security 的退出 URL。当我们从 security 命名空间配置方式修改为明确 bean 定义时，这是我们在这个 bean 定义时重写 filterProcessesUrl 属性的原因，以保持应用的配置保持持续性。

RememberMeAuthenticationFilter

你可能也会回忆起在第三章中使用的 remember me 功能。即使在基于 bean 的配置中，我们也要保持这个有用的 remember me 功能。这关系到一个新的过滤器、新的支持 bean 以及修改一些其它的 bean。首先让我们看一下这个过滤器：

```
<bean id="rememberMeAuthenticationFilter"
    class="org.springframework.security.web
    .authentication.rememberme.RememberMeAuthenticationFilter">
```

```
<property name="rememberMeServices" ref="rememberMeServices"/>
<property name="authenticationManager" ref="customAuthenticationManager" />
</bean>
```

你会发现对 `rememberMeServices` 的引用，而它还没有定义。现在让我们定义这个 bean：

```
<bean id="rememberMeServices"
  class="org.springframework.security.web.authentication
  .rememberme.PersistentTokenBasedRememberMeServices">
  <property name="key" value="jbcpPetStore"/>
  <property name="tokenValiditySeconds" value="3600"/>
  <property name="tokenRepository" ref="jdbcRememberMeTokenRepository"/>
  <property name="userDetailsService" ref="jdbcUserService"/>
</bean>
```

`RememberMeServices` 实现的很多配置属性与 `security` 命名空间风格的配置很相似。这两种方法的最大不同是 `RememberMeServices` 从应用配置的细节中抽象出来了（译者注：即在 `security` 方式的配置中，已经进行了一些抽象），但是在基于 `bean` 的配置中，`bean` 声明必须非常了解在 `remember me` 功能中所关联的所有 `bean`。如果你感到迷惑，请回顾第三章到第四章以了解 `remember me` 功能所关联的类和流程的细节。

我们有另一个 `bean` 应用来织入 `remember me` 的 token 存储。这个 `bean` 定义很简单，如下：

```
<bean id="jdbcRememberMeTokenRepository"
  class="org.springframework.security.web
  .authentication.rememberme.JdbcTokenRepositoryImpl">
  <property name="dataSource" ref="dataSource"/>
</bean>
```

最后我们需要声明 `AuthenticationProvider`，它负责处理 `remember me` 认证请求。它的声明如下：

```
<bean id="rememberMeAuthenticationProvider"
  class="org.springframework.security.authentication.RememberMeAuthenticationProvider">
  <property name="key" value="jbcpPetStore"/>
</bean>
```

你可能会记得这个 `key` 属性在 `AuthenticationProvider`（用来进行 token 校验）和 `RememberMeServices`（用来进行 token 生成）之间共享。要保证它们被设置成相同的。你可能想通过一个 `properties` 文件来设置这个属性值，那是要 `PropertyPlaceholderConfigurer` 工具类（或其它类似的）。

现在我们需要将这个 `AuthenticationProvider` 织入到 `AuthenticationManager` 的列表中。

```
<bean id="customAuthenticationManager"
  class="org.springframework.security.authentication.ProviderManager">
  <property name="providers">
    <list>
      <ref local="daoAuthenticationProvider"/>
      <ref local="anonymousAuthenticationProvider"/>
      <ref local="rememberMeAuthenticationProvider"/>
    </list>
  </property>
</bean>
```

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

RememberMeServices 必须织入到 UsernamePasswordAuthenticationFilter，这样 RememberMeServices 就能处理明确的登录成功（remember me cookie 被更新）和失败（remember me 被移除）。(译者注：其实查看源码在登录失败时，没有移除 cookie 的操作)

```
<bean id="usernamePasswordAuthenticationFilter"
    class="org.springframework.security.web
    .authentication.UsernamePasswordAuthenticationFilter">
    <property name="authenticationManager"
        ref="customAuthenticationManager"/>
    <property name="rememberMeServices" ref="rememberMeServices"/>
</bean>
```

最后要记住的一件事（在配置 Spring Security bean 声明时，用户经常遗忘） RememberMeServices 也是 LogoutHandler 功能的一部分，它在用户退出时清理用户的 cookie。

```
<bean id="logoutFilter"
    class="org.springframework.security.web .authentication.logout.LogoutFilter">
    <constructor-arg value="/" />
    <constructor-arg>
        <array>
            <ref local="logoutHandler"/>
            <ref local="rememberMeServices"/>
        </array>
    </constructor-arg>
    <property name="filterProcessesUrl" value="/logout"/>
</bean>
```

当这些配置到位，我们就完成了明确织入 remember me 功能。你可能没有想到过 <remember-me> 声明在幕后做了多少的工作。

ExceptionTranslationFilter

在 Spring Security 标准的过滤器链中最后一个 servlet 过滤器是 ExceptionTranslationFilter。从本章我们前面的讨论，回忆一下这个过滤器在认证和授权整个流程中的重要性。最后的这个过滤器需要一个过滤器定义和两个支持 bean。

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

支持 bean 的声明如下：

```
<bean id="authenticationEntryPoint"
```

```
class="org.springframework.security.web  
.authentication.LoginUrlAuthenticationEntryPoint">  
    <property name="loginFormUrl" value="/login.do"/>  
</bean>  
<bean id="accessDeniedHandler"  
    class="org.springframework.security.web.access.AccessDeniedHandlerImpl">  
    <property name="errorPage" value="/accessDenied.do"/>  
</bean>
```

你可能会认出 `errorPage` 指令定义的 Access Denied 页面就是本章前面访问拒绝配置练习中的那个地址。类似的，`loginFormUrl` 对应于我们在前面看到的 `security` 命名空间中的 `login-page` 属性。

明确配置 SpEL 表达式和投票器

让我们看一下与 `security` 命名空间中 `use-expressions="true"` 属性等同的 Spring bean 配置：

```
<bean class="org.springframework.security  
.web.access.expression.DefaultWebSecurityExpressionHandler"  
    id="expressionHandler"/>
```

接下来，我们需要建立投票器（Voter）指向表达式处理器，如下：

```
<bean class="org.springframework.security.web.access  
.expression.WebExpressionVoter" id="expressionVoter">  
    <property name="expressionHandler" ref="expressionHandler"/>  
</bean>
```

最后，我们需要使 `AccessDecisionManager` bean 使用这个投票器。

```
<bean class="org.springframework.security.access.vote.AffirmativeBased"  
    id="affirmativeBased">  
    <property name="decisionVoters">  
        <list>  
            <ref bean="expressionVoter"/>  
        </list>  
    </property>  
</bean>
```

在完成这些配置以后，我们回到了使用 `use-expressions="true"` 默认设置的那个点上（译者注：即与使用命名空间配置 `use-expressions="true"`一样的效果）。但是，既然我们使用了明确配置方式，那么可以使用自定义的类替换默认的某个或全部类。我们将会本章后面的的部分看一个这样的例子。

基于 bean 配置方法安全

在第五章中，我们使用 `security` 命名空间`<global-method-security>`声明的方式配置过方法安全。迁移到明确的、基于 `bean` 的配置时，我们必须配置需要的主要类和支持类以重写`<global-method-security>`声明的功能。

我们在第五章中，已经开发出完整的以 `bean` 配置的方法安全，支持各种类型的注解。

鉴于这个配置很简单直接，基本上没有什么有意思属性，所以我们将完整的配置放在附录：参考资料中以及本章的 `dogstore-explicit-base.xml` 配置文件里面。

包装明确的配置

到此时，我们已经完成了配置基于 `bean` 的 Spring security，应用也是功能完整的了。如果你想体验 `security` 命名空间风格的配置与 `bean` 声明的配置，很容易修改 `web.xml` 中对 Spring 配置文件的引用从而把一系列配置文件的集合切换到另一个。

- `security` 命名空间的配置文件为 `web.xml` 引用的 `dogstore-base.xml` 和 `dogstore-security.xml`；
- 基于 `bean` 的配置文件为 `web.xml` 引用的 `dogstore-explicit-base.xml`。

从此往后，本书的练习中将会假设你使用的是 `security` 命名空间配置。如果特定的功能只能使用基于 `bean` 的配置，我们将会明确说明。我们也会包含一些基于 `bean` 配置的细节，因为我们知道一些开发人员更愿意拥有这个级别上对安全框架的控制。

我们应该选择什么类型的配置呢？

希望你的大脑没有从我们刚才的所有配置中变迷糊。你可能会想知道，对于一个典型的工程，应该选择什么类型的配置呢。正如你可能预料的那样，答案是取决于你项目的复杂性以及你重写或自定义 Spring Security 元素的程度。

配置类型	好处
<code>security</code> 命名空间	<ul style="list-style-type: none">● 强大、简洁语法，适用于通常的 web 和方法安全配置；● 用户配置复杂功能时，并不需要知道组件在幕后是如何交互的；● <code>security</code> 命名空间进行代码检查和并警告多种潜在的配置缺陷；● 明显减少缺失配置步骤的可能性。
明确的 <code>bean</code> 定义	<ul style="list-style-type: none">● 允许最大灵活性以扩展、重写以及删节标准的 Spring Security 功能；● 允许自定义过滤器链及根据 URL 模式（使用<code><filter-chain></code>元素的 <code>pattern</code> 属性）的认证方法。这可能在混合 web service 或 REST 认证以及用户认证时用到；● 不用直接将配置文件绑定到 Spring Security 命名空间处理上；● 认证管理器可以明确配置或重写；● 与更简单的 <code>security</code> 命名空间相比，暴露了有更多的配置选项。

对于大多数项目，比较明智的是以 `security` 命名空间开始，并在可能的情况下继续使用直到你的应用需要它不满足的功能。要记住的是，自己配置所有需要的 `bean` 和 `bean` 间的依赖关系将会明显增加复杂度，在开始之前，你应该对 Spring Security 的结构（可能还包括底层代码）有清晰的理解。

认证事件处理

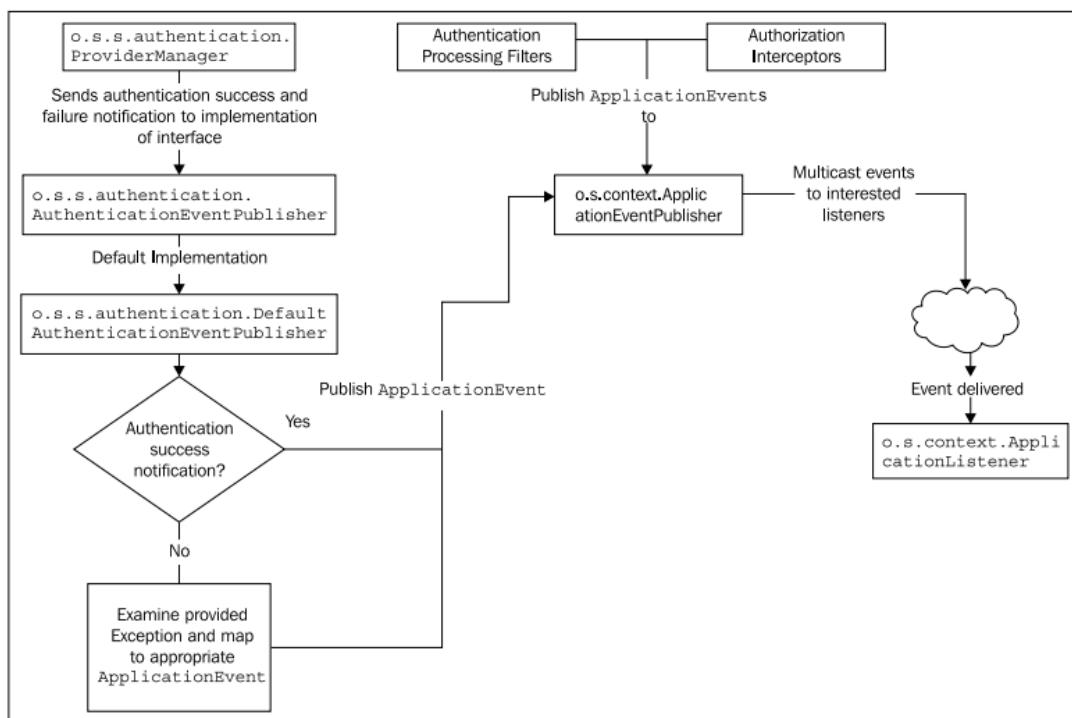
有一个重要的功能只能通过基于 `bean` 的配置就是自定义处理认证事件。认证事件使用

了 Spring 的时间发布机制，它基于 `o.s.context.ApplicationEvent` 事件模型。Spring 事件模型使用并不广泛，却能够很有用处——特别在认证系统中——如当你想绑定特定行为到认证领域的行动上去的时候。

事件是典型的订阅-发布模式，通知订阅者是 Spring 运行环境自己处理的。比较重要的一点是，在默认情况下 Spring 的事件模型是同步的，所以有任何订阅监听的运行时会直接影响产生事件请求的性能。

在 `ApplicationContext` 初始化的时候，Spring 将会检查所有配置的 bean 是否存在 `o.s.context.ApplicationListener` 接口。这些 bean 的引用将会被 `o.s.context.event.ApplicationEventMulticaster` 持有，它会在 `o.s.context.ApplicationEventPublisher` 发布事件时，管理运行时事件的发布。这个设施已经存在很长时间了（从 Spring1.1），所以你若想更深入了解 Spring 的这个领域，有很多文档可查。

下图阐述了事件发布流程是如何组织在一起的：



因为在 Spring Security 内部没有广泛使用认证事件（实际上，唯一明显用的地方就是我们本章前面讨论的 session 并发跟踪），自定义认证事件的监听器是一种实现审计、管理报警甚至复杂用户行为追踪的便利方式。

让我们了解一下配置简单安全事件监听的过程。

配置认证事件的监听器

使用简短的 `security` 命名空间配置，你不能配置认证事件监听器——它必须使用基于 `Spring bean` 的方式因为 `ApplicationEventPublisher` 的实现类默认不会启用，必须织入到 `AuthenticationManager` 中。

声明需要的 bean 依赖

我们首先声明 ApplicationEventPublisher 的实现类，如下：

```
<bean id="defaultAuthEventPublisher"
      class="org.springframework.security.authentication .DefaultAuthenticationEventPublisher"/>
```

接下来，我们将会把它织入到使用的 AuthenticationManager 中：

```
<bean id="customAuthenticationManager"
      class="org.springframework.security.authentication.ProviderManager">
    <property name="authenticationEventPublisher" ref="defaultAuthEventPublisher"/>
    <property name="providers">
      <list>
        <ref local="daoAuthenticationProvider"/>
        <ref local="rememberMeAuthenticationProvider"/>
      </list>
    </property>
</bean>
```

这就是全部需要的配置。如果此时你重启应用，你将会什么也看不到。这是因为我们还没有创建 bean 来监听发布的时间。现在，我们就做这件事。

构建自定义的应用事件监听器

ApplicationListener 的实现类很简单，并且使用 Spring 3 加入的强大 Java 泛型功能支持类型安全。我们的 ApplicationListener 只是简单记录收到的事件到标准输出中，但是在后面的练习中我们将会体验一个更有趣的例子。

自定义的 ApplicationListener 如下：

```
package com.packtpub.springsecurity.security;
// imports omitted
@Component
public class CustomAuthenticationEventListener implements
    ApplicationListener<AbstractAuthenticationEvent> {
    @Override
    public void onApplicationEvent(AbstractAuthenticationEvent event) {
        System.out.println("Received event of type:
            "+event.getClass().getName()+" : "+event.toString());
    }
}
```

你会发现我们这里使用了@Component 注解，但是我们也可以在 XML 配置文件中简单声明一个 Spring bean。

记住，ApplicationListener 的实现类必须注明对什么类型的事件感兴趣，在 Spring 3 中通过在 ApplicationListener 接口引用中声明泛型来标注。Spring 的 ApplicationEventMulticaster 使用一些巧妙的方法来检查类的接口实现声明并确保正确的事件到达正确的类中。

【巧妙的注解。你如果对复杂的注解处理和运行时检查注解感到好奇，毫无疑问那你应该查看使用 Spring 的 `o.s.context.event.GenericApplicationListenerAdapter` 分发 `ApplicationEvent` 事件的巧妙代码。看一下并学习一些 Java 反射的新技巧。】

重启应用，然后进行一些常用的行为如登录、退出以及登录失败。你能看到，当这些行为执行时，适当的时间被触发并打印在控制台上。

尽管我们声明了自己的 `ApplicationListener` 来接受所有的认证事件，但是在大多数场景下这并不实用，根据系统使用情况的，在一个小时內可能会有数千个时间触发。通过修改类 `implements` 关键词上的泛型标示，我们能够使得实现类至监听一种类型的事件。

内置的 ApplicationListeners

Spring Security 提供了两个 `ApplicationListener` 的实现类，它们绑定了在 Spring Security 中使用的 Apache Commons Logging 日志。两个 `ApplicationListener` 实现类，一个是负责认证事件，一个负责授权时间。你可以像以下代码那样配置它们：

```
<bean id="authenticationListener"
      class="org.springframework.security.authentication.event.LoggerListener"/>
<bean id="authorizationListener"
      class="org.springframework.security.access.event.LoggerListener"/>
```

这两个监听器将会输出 Commons Logging 日志以对应的类命名。一个示例记录 `AbstractAuthenticationFailureEvent` 大致如下：

```
WARN - Authentication event
AuthenticationFailureBadCredentialsEvent: adb; details: org.
springframework.security.web.authentication.WebAuthenticationDetails@2
55f8: RemoteIpAddress: 127.0.0.1; SessionId: B20510F25464B109CE3AE94D9
FBF981E; exception: Bad credentials
```

如果你要实现类似的 `ApplicationListener` 来记录有用的事件，这些类可以作为模板。

大量的应用事件

Spring Security 提供了很多的事件，其试图在用户认证请求的所有点上给出有用的信息。你的应用可以监听可用的各种事件，这个范围可以很广泛（所有认证失败）也可以很窄小（一个用户通过提供完整的凭证认证成功）。完整的事件列表在 [附录：参考资料](#) 中。一些其它的关于异常处理和事件监听的注意事项如下：

- 授权事件和框架抛出异常的匹配关系可以通过 `DefaultAuthenticationEventPublisher` 的 `exceptionMappings` 属性配置；
- 记住，正如我们在本章前面看到的，跟踪 `HttpSession` 的生命周期是通过 `web.xml` 配置的变化，而不直接是 Spring。

你可以看到 Spring Security 的异常和事件处理很强大，允许在你的安全系统中进行很多场景的跟踪和对活动的响应。

构建一个自定义实现的 SpEL 表达式处理器

我们将会阐述一个扩展基本 SpEL 表达式处理器的简单例子，提供一个表达式如果当前日期的分钟数为偶数将会允许访问。尽管这是一个很牵强的例子，但是它描述了实现自定义 SpEL 表达式方法的所有步骤。

让 我 们 创 建 一 个 类
com.packtpub.springsecurity.security.CustomWebSecurityExpressionRoot 以建立自定义扩展的 WebSecurityExpressionRoot。

```
public class CustomWebSecurityExpressionRoot extends WebSecurityExpressionRoot {  
    public CustomWebSecurityExpressionRoot (Authentication a, FilterInvocation fi) {  
        super(a, fi);  
    }  
    public boolean isEvenMinute() {  
        return (Calendar.getInstance().get(Calendar.MINUTE) % 2) == 0;  
    }  
}
```

接下来，我们需要一个实现 WebSecurityExpressionHandler 的类。我们扩展了 DefaultWebSecurityExpressionHandler 并重写一个方法在 com.packtpub.springsecurity.security.CustomWebSecurityExpressionHandler 类中建立自己的 CustomWebSecurityExpressionRoot。

```
public class CustomWebSecurityExpressionHandler  
    extends DefaultWebSecurityExpressionHandler {  
    public EvaluationContext createEvaluationContext(Authentication authentication,  
FilterInvocation fi) {  
        StandardEvaluationContext ctx = (StandardEvaluationContext)  
            super.createEvaluationContext(authentication, fi);  
        SecurityExpressionRoot root = new CustomWebSecurityExpressionRoot(authentication, fi);  
        ctx.setRootObject(root);  
        return ctx;  
    }  
}
```

最后，当建立 Voter 时需要重新配置 bean 引用，如下：

```
<bean class="com.packtpub.springsecurity.security.CustomWebSecurityExpressionHandler"  
    id="customExpressionHandler"/>  
<bean class="org.springframework.security.web.access.expression.WebExpressionVoter"  
    id="expressionVoter">  
    <property name="expressionHandler" ref="customExpressionHandler"/>  
</bean>
```

现在，我们可以使用这个表达式来根据时间的分钟数是偶数还是奇数进行限制访问。

```
<security:intercept-url pattern="/" access="evenMinute"/>
```

很显然，这是一个简单的例子，但是阐述了实现自定义 SpEL 属性的基本步骤，你可以使用这种方式来控制对应用特定部分的访问。

【配置自定义 SpEL Voter 的技术可能在使用 security 命名空间的时候也会用到，只需使用 access-decision-manager-ref 属性定义一个自定义的 AccessDecisionManager，就像我们在第二章见过的那样。】

小结

在本章中，我们介绍了 Spring Security 标准配置的功能并实现了一些高级的自定义功能。我们涉及到以下的内容：

- 实现自定义的 servlet 过滤器来处理基于 IP 和角色的过滤以及基于 HTTP 头的 SSO 请求；
- 添加一个自定义的 AuthenticationProvider 及支持类，从而实现 HTTP 请求头的 SSO；
- 了解 session 固化防护和 session 并发处理的配置及好处，包括一些间接的功能以允许用户进行 session 报告；
- 配置自定义的访问控制拒绝处理并了解何时及为何 AccessDeniedException 会被抛出，还有怎样适当地响应它；
- 替换 Spring Security 的自动化配置为手动声明所有需要的参与类，这借助于标准的 Spring bean XML 配置技术；
- 了解一些基于 Spring bean 配置的高级功能，包括 session 管理和事件发布；
- 实现自定义和内置的 ApplicationListener 来响应 Spring Security 框架发布的特定事件；
- 实现标准 SpEL 表达式处理器的自定义扩展以允许个性化的 URL 访问表达式。

多有趣的一章！我们已经很习惯 Spring Security，并进行了一些高级的扩展和自定义。在第七章中，我们将会进行高级配置的旅程，通过使用访问控制列表使得实现复杂认证变得可能。

第七章 访问控制列表（ACL）

在本章中，我们将会介绍访问控制列表这个复杂话题，它能够提供域对象实例层次授权的丰富模型。Spring Security 提供了强大的访问控制列表，但是复杂且缺少文档，它能够很好的满足小到中型规模的实现。

在本章的内容中，我们将会：

- 理解访问控制列表的概念模型；
- 了解 Spring Security ACL 模型中的关于访问控制列表的术语和应用；
- 构建和了解支持 Spring ACL 的数据库模式；
- 配置 JBCP Pets 通过注解和 Spring bean 来使用 ACL 保护业务方法；
- 进行高级配置，包括自定义许可、使用 ACL 的 JSP 标签检查和方法安全，易变的 ACL 以及缓存；
- 了解架构要考虑的因素以及规划 ACL 部署的场景。

译者注：在本章中术语 permission 翻译为许可权限；entry 翻译为条目；access control entry 即为访问控制条目。

使用访问控制列表保护业务对象

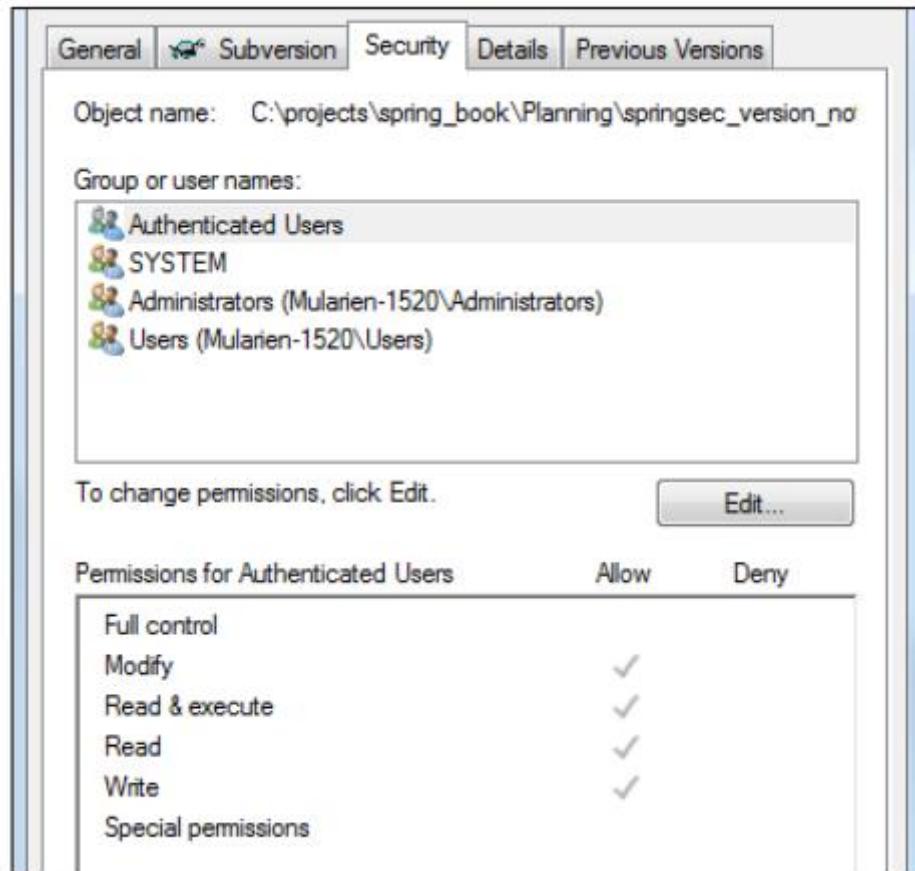
最后一个非 web 层安全的问题是业务对象层次的安全，它在业务层或业务层以下。这个层次的安全实现使用了一项名为访问控制列表（access control list，或 ACL）的技术。ACL 中的对象可以进行权限判断——ACL 允许基于唯一的组、业务对象以及逻辑操作进行特定的许可。

例如，在 JBCP Pets 中的一个 ACL 声明可能会是这样的：用户只能对自己的简介进行写操作。可能像下面展现的这样：

用户名	组	对象	许可权限
amy		Profile123	read, write
	ROLE_USER	Profile123	read
	ANONYMOUS	Any Profile	none

你会发现这个 ACL 对人工来说很易读——Amy 对自己的简介（Profile123）能够进行读写；其它的注册用户能够读 Amy 的简介，而匿名用户不能。简单来说，这种类型的规则矩阵就是 ACL 所试图做的，即将代码、访问检查、安全系统和业务数据的元数据进行集成。大多数真正使用 ACL 的系统会有很复杂的 ACL 列表以及整个系统中百万级的条目。尽管这听起来有令人感到害怕的复杂性，但是预先适当的思考和使用良好的安全库能够使得 ACL 管理相当可行。

如果你使用 Microsoft Windows 或者基于 Unix/Linux 的电脑，那你每天都在体验 ACL。大多数现代电脑的操作系统都使用 ACL 指令作为文件存储系统的一部分，这允许基于用户或组、文件或目录以及许可权限的组合进行许可授权。在 Microsoft Windows 中，你能通过右键点击一个文件并查看其安全属性的方法（属性|安全标签）看到 ACL 功能：



你可以看见各种的组或用户以及权限，从而能够以可视化和很直观的方式输入 ACL。

Spring Security 中的访问控制列表

在安全系统中，Spring Security 支持 ACL 驱动的授权检查用户访问某个域对象。与 OS 文件系统的例子很相似，可以使用 Spring Security ACL 组件构建业务对象以及组或安全实体的逻辑树结构。基于请求者和被请求对象得到的许可授权（继承或明确声明）被用来确定是否允许访问。

对于接触 Spring Security ACL 功能的用户来说，很可能被它的复杂性和缺乏文档、例子所吓倒。再加上建立 ACL 的基础设施很复杂，需要很多的相互依赖以及与 Spring Security 其它地方不同的基于 bean 的配置机制（我们等会建立初始配置的时候，你会看到）。

Spring Security 的 ACL 模型比较基础，但是试图构建扩展功能的用户可能会发现一些令人沮丧的限制并且 Spring Security 早期引入的一些设计决策已经不合时宜。不要让这些限制把你弄得灰心。ACL 模型是一个往你应用中嵌入丰富访问控制的强大方式，并且会进行仔细的审查以保护用户的行为和数据。

在我们开始配置 Spring Security ACL 之前，我们需要了解一些关键的术语和概念。

在 Spring ACL 系统中，主要的安全角色标识是 `security identity` 或 `SID`。`SID` 是一个逻辑组成，能够被用来抽象一个单独的安全实体或组 (`GrantedAuthority`)。ACL 数据模型定义的 `SID` 被用做确定安全实体访问级别的基础，而这个访问规则可以是明确指明的也可以是继承下来的。

如果 `SID` 被用作确定 ACL 系统的角色，那对应的另一半安全相关的就是安全对象本身了。单个的安全对象标识被称为（毫无意外的）对象标识 (`object identity`)。默认的 Spring ACL

实现中，对象标识需要 ACL 规则定义在对象实例层级，这就意味着，如果需要系统中的每一个对象都能有单独的访问规则。

单个的访问规则就是所谓的访问控制条目（access control entries 或 ACEs）。一个 ACE 包含以下的元素：

- 规则应用的角色 SID；
- 规则应用的对象标识；
- 应用于给定 SID 和对象标识的许可权限；
- 给定的许可权限对于特定的 SID 和对象标识应该允许还是拒绝（译者注：个人理解，此处指的是给定一个权限能够判断出是否允许访问）。

Spring ACL 系统作为一个整体，其目的是评估每一个方法调用并确定方法中的对象针对每一个 ACE 是否可用。适当的 ACE 在运行时进行评估，这要基于调用者和使用的对象。

【Spring Security ACL 在实现上是灵活的。尽管本章的内容大多数都是 Spring Security 模块内置的功能，但是要记住的是很多的规则是默认实现，在很多情况下能够基于复杂的需求进行重写。】

Spring Security 使用一些有用的值对象来描述相关联的每一个概念实体。它们列到了以下的表格中：

ACL 概念对象	Java 对象
SID	o.s.s.acls.model.Sid
对象标识（Object Identity）	o.s.s.acls.model.ObjectIdentity
ACL	o.s.s.acls.model.Acl
ACE	o.s.s.acls.model.AccessControlEntry

让我们通过 JBCP Pets store 应用的一个例子来了解使用 Spring Security ACL 组件的过程。

支持 Spring Security ACL 的基本配置

尽管我们前面提到过在 SpringSecurity 中配置支持 ACL 需要基于 bean 的配置（它的确如此），但是你可以使用 ACL 却保持较为简单的 security XML 命名空间配置（译者注：即 ACL 需要基于 bean 的配置，但是原有的 security 命名空间配置可以保留）。如果你要运行示例代码，你需要切换 web.xml 到 security 命名空间配置，使用 dogstore-base.xml 和 dogstore-security.xml。

定义一个简单的目标场景

我们一个简单的目标场景是不允许没有 ROLE_ADMIN GrantedAuthority 权限的用户访问主页上的第一个分类。第一个分类被称为“Pet Apparel”——这就是我们要用 ACL 安全保护的分类。

尽管有多种方式建立 ACL 检查，但是我们喜欢在第五章用到过的基于注解实现方法安全的方式。它很好的把使用 ACL 从实际接口声明中抽取出来，并允许你以后将角色声明替换为（如果愿意）除了 ACL 的其它方式。

我们将要在 IProductService.getItemsByCategory 方法上添加一个注解，它需要触发这个

方法的任何人有适当的角色来查看这个分类：

```
@Secured("VOTE_CATEGORY_READ")
public Collection<Item> getItemsByCategory(Category cat);
```

这个方法被 JBCP Pets 站点的分类查看页面调用，而这个页面可以通过点击主页上的分类进入：



让我们看是进行配置的变化！

添加 ACL 表到 HSQL 数据库中

我们需要做的第一个事情是添加支持持久化 ACL 条目的表到我们的内存 HSQL 数据库中。要做到这一点，我们添加一些新的 SQL DDL 文件到我们的嵌入式数据库声明中，在 dogstore-security.xml 里：

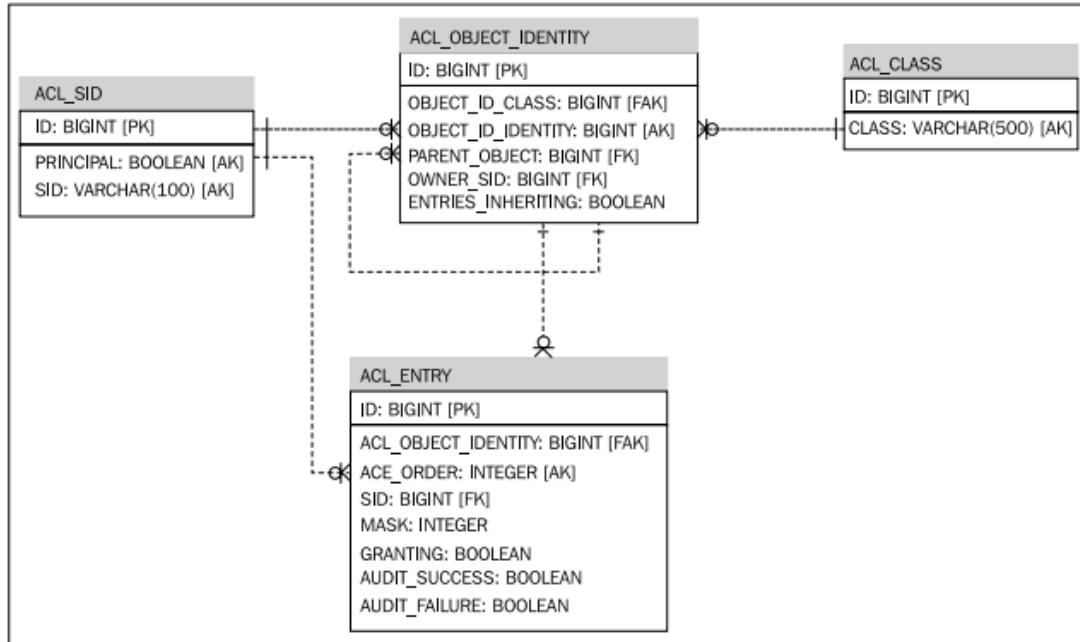
```
<jdbc:embedded-database id="dataSource" type="HSQL">
    <jdbc:script location="classpath:security-schema.sql"/>
    <jdbc:script location="classpath:test-data.sql"/>
    <jdbc:script location="classpath:remember-me-schema.sql"/>
    <jdbc:script location="classpath:test-users-groups-data.sql"/>
    <jdbc:script location="classpath:acl-schema.sql"/>
</jdbc:embedded-database>
```

acl-schema.sql 文件会放在 WEB-INF/classes 文件夹下（或在应用类路径下的其它位置），并包含以下内容：

```
create table acl_sid (
    id bigint generated by default as identity(start with 100) not null
primary key,
    principal boolean not null,
    sid varchar_ignorecase(100) not null,
    constraint uk_acl_sid unique(sid,principal) );
create table acl_class (
    id bigint generated by default as identity(start with 100) not null
primary key,
    class varchar_ignorecase(500) not null,
    constraint uk_acl_class unique(class) );
create table acl_object_identity (
```

```
id bigint generated by default as identity(start with 100) not null
primary key,
object_id_class bigint not null,
object_id_identity bigint not null,
parent_object bigint,
owner_sid bigint not null,
entries_inheriting boolean not null,
constraint uk_acl_objid unique(object_id_class,object_id_identity),
constraint fk_acl_obj_parent foreign key(parent_object)references
acl_object_identity(id),
constraint fk_acl_obj_class foreign key(object_id_class)references
acl_class(id),
constraint fk_acl_obj_owner foreign key(owner_sid)references acl_
sid(id);
create table acl_entry (
    id bigint generated by default as identity(start with 100) not null
primary key,
    acl_object_identity bigint not null,
    ace_order int not null,
    sid bigint not null,
    mask integer not null,
    granting boolean not null,
    audit_success boolean not null,
    audit_failure boolean not null,
    constraint uk_acl_entry unique(acl_object_identity,ace_order),
    constraint fk_acl_entry_obj_id foreign key(acl_object_identity)
        references acl_object_identity(id),
    constraint fk_acl_entry_sid foreign key(sid) references acl_sid(id)
);
```

这将会生成如下的数据库模式：



你能够看到 SID、对象标识以及 ACE 的概念如何匹配到数据库模式中。概念上讲，这很方便，因为我们能够匹配 ACL 系统的模型并直接将其关联到数据库中。

如果你将这与 Spring Security 文档中提供的 HSQL 数据库模式进行对比，你会发现我们进行了一些改变，这可能会使用户走弯路。如下：

- 修改 `ACL_CLASS.CLASS` 列从 100 个字符到 500 个字符。一些满足要求的长类名可能超过 100 个字符；
- 外键的名字更有意义，这样失败能够更容易的进行诊断。

【如果你使用其它的数据库，如 `oracle`，你需要将这个 DDL 转换成你特定数据库的数据类型。】

一旦我们配置完 ACL 系统的其它部分，我们将会回到数据库这边建立一些基本的 ACE 以证明 ACL 的基本功能。

配置访问决策管理器

我们需要配置`<global-method-security>`以启用注解（我们将在这里注明基于 ACL 的权限）并引用一个自定义的访问决策管理器（access decision manager）。

【用于 ACL 检查的访问决策管理器必须与用于检查 web URL 授权规则的进行区分。这个需求源于传递给决策管理器的所有认证检查必须被它所有配置的 voter 所支持。不幸的是，web 请求认证与方法请求认证的处理方法不一样，所以需要一个单独配置的访问决策器。当我们在第五章：精确的访问控制第一次接触方法安全时，我们不需要明确进行这个配置变化，因为 security 命名空间解析所实例化的访问控制管理器足够满足我们的要求。】

配置访问决策管理器的引用被设置在 `dogstore-security.xml` 文件中，如下：

```
<global-method-security secured-annotations="enabled"  
access-decision-manager-ref="aclDecisionManager"/>
```

这是对访问控制管理器的一个 bean 引用，我们定义在 `dogstore-base.xml` 文件中：

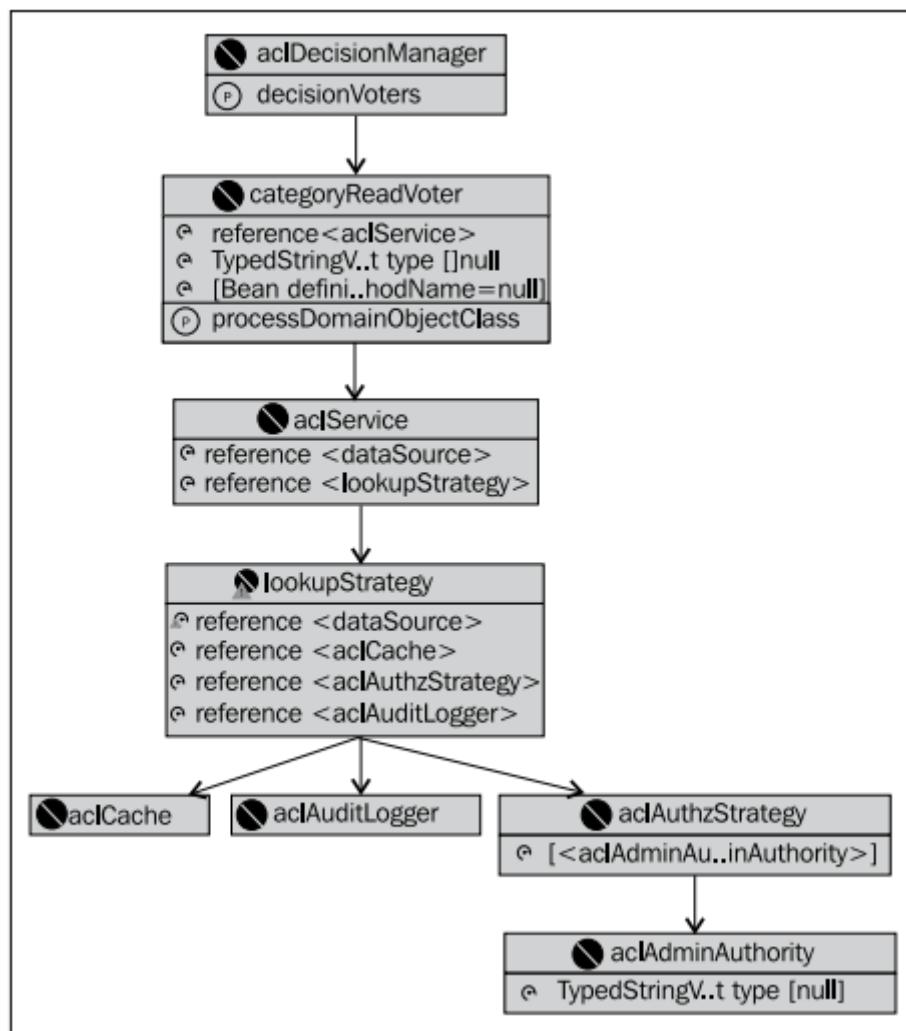
```
<bean class="org.springframework.security.access.vote.AffirmativeBased"
```

```
id="aclDecisionManager">
<property name="decisionVoters">
    <list>
        <ref bean="categoryReadVoter"/>
    </list>
</property>
</bean>
```

我们将会在练习的下一步中从投票器开始处理这个引用链。要记住的是这个 AccessDecisionManager 像其它的一样，自然也像 web 认证决策管理器那样，依赖于投票器做授权决策。

配置 ACL 的支持 bean

即使是进行一个相对很简单的 ACL 配置，就像我们的场景那样，也有许多需要的依赖要建立。正如我们前面提到的，Spring Security ACL 模块自带了很多的组件，你可以对它们进行组装以提供很好的 ACL 功能。注意的是，下图所引用的所有组件只是框架的一部分！



ACL 子系统与 Spring Security 框架其它大部分功能有一个明显的不同就是大多数的配置需要构造方法注入（constructor injection），而不是属性注入（property injection）。正如我们

在第六章：高级配置和扩展中明确 bean 配置所讨论的那样，一部分 Spring Security 在这方面并不一致，强制要求使用构造器以确保需要的属性被设置——这在配置大多数 ACL 相关组件时是需要注意的。

让我们从配置 categoryReadVoter 开始。这是 AccessDecisionVoter 的实现，它会访问 ACL 存储（在数据库中）并进行运行时的认证做出访问决策。

```
<bean class="org.springframework.security.acls.AclEntryVoter"  
      id="categoryReadVoter">  
    <constructor-arg ref="aclService"/>  
    <constructor-arg value="VOTE_CATEGORY_READ"/>  
    <constructor-arg>  
      <array>  
        <util:constant static-field="org.springframework.security.acls.  
domain.BasePermission.READ"/>  
      </array>  
    </constructor-arg>  
    <property name="processDomainObjectClass"  
             value="com.packtpub.springsecurity.data.Category"/>  
</bean>
```

不是明显 bean 引用的属性需要进行一些说明。第二个构造参数，我们给它一个值为 VOTE_CATEGORY_READ，用来表明这个投票器能够投票的安全属性。你可能会记得这与我们要进行 ACL 保护的方法上的@Secured 注解匹配。

第三个构造参数以及属性联合起来声明了能够对请求做出一个成功投票的 ACL 属性。在本例中，声明的投票器用来对包含 VOTE_CATEGORY_READ 的方法访问进行授权，请求者必须有 ACL 条目来表明他允许对 com.packtpub.springsecurity.data.Category 域对象进行 READ 访问。

我们可以看到，ACL 投票器的声明是一个抽象层，并且位于代码资源声明的授权要求和域对象对 ACL SID 分配许可权限之间。这层抽象使得对安全资源分配有意义的属性方面有了很大的灵活性。

引用 aclService 的 bean 处理的是 o.s.s.acls.model.AclService 的一个实现，它负责（通过代理）将 ACL 保护的对象转换成期望的 ACE。

```
<bean class="org.springframework.security.acls.jdbc.JdbcAclService"  id="aclService">  
  <constructor-arg ref="dataSource"/>  
  <constructor-arg ref="lookupStrategy"/>  
</bean>
```

我们将会使用 o.s.s.acls.jdbc.JdbcAclService，它是 AclService 的一个实现。这个实现是内置的并且使用我们这个练习上一步定义的数据库模式。JdbcAclService 将会使用递归的 SQL 和事后处理机制来理解 SID 的等级关系，并确保等级关系的表达能够传递回 AclEntryVoter。

JdbcAclService 使用与嵌入式数据库定义时相同的 JDBC dataSource，并代理给一个 o.s.s.acls.jdbc.LookupStrategy 的实现，而这个实现将真正负责数据库查询以及处理对 ACL 的请求。Spring Security 提供的唯一 LookupStrategy 是 o.s.s.acls.jdbc.BasicLookupStrategy，定义如下：

```
<bean class="org.springframework.security.acls.jdbc.BasicLookupStrategy" id="lookupStrategy">  
  <constructor-arg ref="dataSource"/>
```

```
<constructor-arg ref="aclCache"/>
<constructor-arg ref="aclAuthzStrategy"/>
<constructor-arg ref="aclAuditLogger"/>
</bean>
```

现在，`BasicLookupStrategy` 是一个很复杂的家伙（beast）。记住，其目标是从数据库中将一系列的 `ObjectIdentity` 转换成实际可用的 ACE 列表。因为 `ObjectIdentity` 的声明可能是递归的，这会是一个很有挑战性的问题，并且对于高负荷使用的应用要考虑产生的 SQL 对数据库性能的影响。

【使用最小公分母进行查询。要注意的是 `BasicLookupStrategy` 要兼容所有的数据库，这通过严格坚持标准的 ANSI SQL 语法实现，要注意的是左[外]连接（left[outer]joins）。一些老的数据库（典型的如 Oracle 8i）并不支持这种连接语法，所以要确保检查 SQL 的语法和结构是否兼容你特定的数据库。当然还有更高效依赖数据库的等级查询方法，这要使用非标准的 SQL ——如 Oracle 的 CONNECT BY 语句以及其它很多数据库（包括 PostgreSQL 和 Microsoft SQL Server）的 Common Table Expression（CTE）功能。正如我们在第四章使用 `JdbcDaoImpl` `UserDetailsService` 自定义模式的例子那样，在使用 `BasicLookupStrategy` 的时候也有属性暴露出来配置 SQL。请查询 Javadoc 和源码本身来了解它们怎么使用，这样它们就能在你自定义模式中正确使用。】

我们可以看到 `LookupStrategy` 需要引用与 `AclService` 相同的 JDBC `dataSoure`。而另外三个引用将会把我们带到这个依赖链的最后。

`o.s.s.acls.model.AclCache` 声明了一个接口依赖缓存 `ObjectIdentity` 到 ACL 的映射，这样就阻止大量（且代价高昂）的数据库查询。`Spring Security` 只提供了一个 `AclCache` 的实现，即使用第三方库 `Ehcache`。简单起见，在此时的配置中，我们忽略掉配置 `Ehcache` 的额外工作，替换为实现一个简单的，不进行任何操作的 `AclCache`，在类 `com.packtpub.springsecurity.security.NullAclCache` 中：

```
package com.packtpub.springsecurity.security;
// imports omitted
public class NullAclCache implements AclCache {
    @Override
    public void clearCache() { }
    @Override
    public void evictFromCache(Serializable arg0) { }
    @Override
    public void evictFromCache(ObjectIdentity arg0) { }
    @Override
    public MutableAcl getFromCache(ObjectIdentity arg0) {
        return null;
    }
    @Override
    public MutableAcl getFromCache(Serializable arg0) {
        return null;
    }
    @Override
    public void putInCache(MutableAcl arg0) { }
}
```

这通过一个简单的 bean 声明来配置：

```
<bean class="com.packtpub.springsecurity.security.NullAclCache"  
id="aclCache"/>
```

不要担心，我们将会在本章后面配置基于 Ehcache 的实现，但是现在我们首先想要关注配置 ACL 真正需要的组件。

BasicLookupStrategy 所要处理的下一个依赖是 o.s.s.acls.domain.AuditLogger 接口的一个现实，它被 BasicLookupStrategy 用来进行审计 ACL 和 ACE 的查找。类似于 AclCache 接口，Spring Security 只提供了一个实现，它简单地在控制台上记录 log。我们将用一行的 bean 声明来配置它：

```
<bean class="org.springframework.security.acls.domain.ConsoleAuditLogger"  
id="aclAuditLogger"/>
```

最后一个要处理的依赖是 o.s.s.acls.domain.AclAuthorizationStrategy 接口的实现，它在从数据库加载 ACL 的时候并没有马上要提供的任何作用。相反，这个接口的实现负责确定运行时对 ACL 或 ACE 的修改是否允许，这要基于修改的类型。当我们涉及到易变的 ACL 时，会有更详细的介绍，因为这个逻辑过程有些复杂并且与我们初始化配置完成没有关系。最后的配置如下：

```
<bean class="org.springframework.security.acls.domain.AclAuthorizationStrategyImpl"  
id="aclAuthzStrategy">  
    <constructor-arg>  
        <array>  
            <ref local="aclAdminAuthority"/>  
            <ref local="aclAdminAuthority"/>  
            <ref local="aclAdminAuthority"/>  
        </array>  
    </constructor-arg>  
  </bean>  
  <bean class="org.springframework.security.core.authority.GrantedAuthorityImpl"  
id="aclAdminAuthority">  
    <constructor-arg value="ROLE_ADMIN"/>  
  </bean>
```

你可能想知道重复引用的 aclAdminAuthority 是干什么的——AclAuthorizationStrategyImpl 提供了三个特殊的 GrantedAuthority 名称来允许对 ACL 进行运行时的特定操作。我们将会在本章后面涉及到。

我们最终完成了对 Spring Security ACL 内置实现的配置。接下来也是最后一步就是需要我们插入简单的 ACL 和 ACE 到 HSQL 数据库中，并进行测试。

创建一个简单的 ACL entry

回忆一下，我们简单的场景就是锁定 JBCP Pets store 的第一个分类，使得只有 ROLE_ADMIN 授权的原来才能查看。你可能会发现翻到前几页参考一下模式图对于理解我们要插入什么数据以及为什么会有帮助。

在 WEB-INF 下建立一个文件 test-acl-data.sql，与其它我们在 JBCP Pets 中使用到的 SQL

文件放到一起。本节描述的所有 SQL 将会加到这个文件中——你可以基于我们提供的实例 SQL 随便体验并添加更多的测试用例——实际上，我们鼓励你使用实例数据进行体验。

首先，我们需要在 `ACL_CLASS` 中添加任意或所有的拥有 ACL 规则的域对象类——在我们的例子中，这就是我们的 `Category` 类：

```
insert into acl_class (class) values ('com.packtpub.springsecurity.  
data.Category');
```

接下来，`ACL_SID` 表插入 SID，它将关联到 ACE 中。要记住的是，SID 可以是角色或用户——在我们的应用中简单插入角色（注意 `principal` 列表明一个给定的行是否为单个用户）：

```
insert into acl_sid (principal, sid) values (false, 'ROLE_USER');  
insert into acl_sid (principal, sid) values (false, 'ROLE_ADMIN');
```

开始变得复杂的表是 `ACL_OBJECT_IDENTITY`，它用来声明单个的域对象实例和它们的父对象（如果存在）以及拥有者的 SID。我们插入拥有以下属性的一行数据：

- 域对象类型为 `Category`（通过 `OBJECT_ID_CLASS` 列外键关联到 `ACL_CLASS`）；
- 域对象的 PK 为 1（`OBJECT_ID_IDENTITY`）；
- 拥有者 SID 为 `ROLE_ADMIN`（通过外键 `OWNER_SID` column 关联到 `ACL_SID`）。

对于主键为 1 的 `Category`，插入一行的 SQL 如下：

```
insert into acl_object_identity (object_id_class, object_id_  
identity, parent_object, owner_sid, entries_inheriting)  
select cl.id, 1, null, sid.id, false  
from acl_class cl, acl_sid sid  
where cl.class='com.packtpub.springsecurity.data.Category' and sid.  
sid='ROLE_ADMIN';
```

要记住的是，在典型的场景下，拥有者 SID 应该为一个安全实体而不是一个角色。但是，对于 ACL 系统来说，两种类型的规则功能是一样的。

最后，我们要对这个只有 `ROLE_ADMIN` 角色才能访问的对象实例添加 ACE：

```
insert into acl_entry (acl_object_identity, ace_order, sid, mask, granting, audit_success,  
audit_failure) select oi.id, 1, si.id, 1, true, true, true  
from acl_object_identity oi, acl_sid si  
where si.sid = 'ROLE_ADMIN';
```

这里的 `MASK` 列代表位掩码，它用来给指定对象在特定的 SID 上进行授权。我们将会在本章后面进行更详细的介绍——幸运的是，在这里它并不像听起来那么有用。

在 SQL 文件准备好之后，我们需要扩展`<embedded-database>`声明并添加最终的 ACL 测试数据文件到数据库启动中：

```
<jdbc:embedded-database id="dataSource" type="HSQL">  
<!--additional SQL files omitted -->  
  <jdbc:script location="classpath:acl-schema.sql"/>  
  <jdbc:script location="classpath:test-acl-data.sql"/>  
</jdbc:embedded-database>
```

现在，我们能够启动应用并运行实例场景。你会发现不是管理员的任何用户试图访问第一个分类“`Pet Apparel`”，他们被拒绝访问。如果没有经过认证的用户试图访问这个分类，他们将会按照标准的 `AccessDeniedException` 处理（第六章已描述）并要求登录。

现在我们建立了基本的基于 ACL 的安全（尽管是一个很简单的场景）。接下来我们对这个过程中的概念进行更多的讲解，然后了解在使用 Spring ACL 之前两个要考虑的问题。

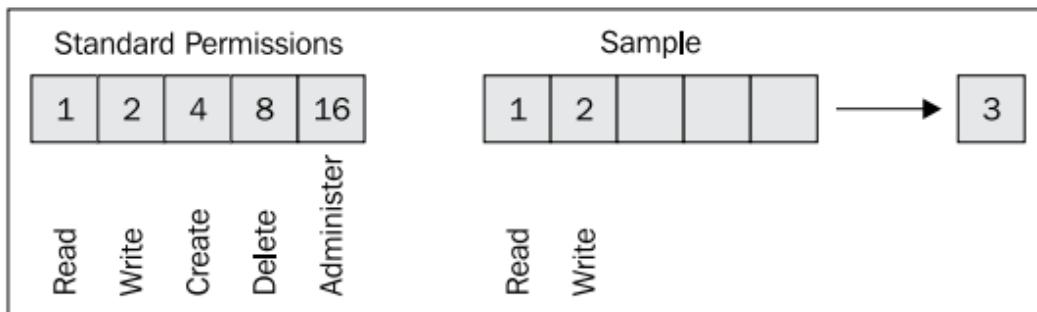
高级 ACL 话题

一些高级的话题在我们配置 ACL 环境时略过了，包括处理 ACE 许可授权，在运行时根据 GrantedAuthority 确定某种类型的 ACL 变化是否允许。既然现在我们已经有了一个运行环境，那我们要开始了解这些更高级的话题。

Permission 如何工作

许可授权（permission）只不过是简单的逻辑标识符用一个整数的二进制位来表示。一个访问控制条目对于 SID 的授权是基于逻辑与操作所有应用于这个条目的许可授权得到的位掩码。

默认的 Permission 实现，即 o.s.s.acls.domain.BasePermission 定义了一系列的整数值来代表常用的 ACL 授权。这些整数值对应于单个位设置为整数，所以 BasePermission.WRITE 的值，对于的整数为 1，按位的值就是 2^1 或 2。它们如下图所示：

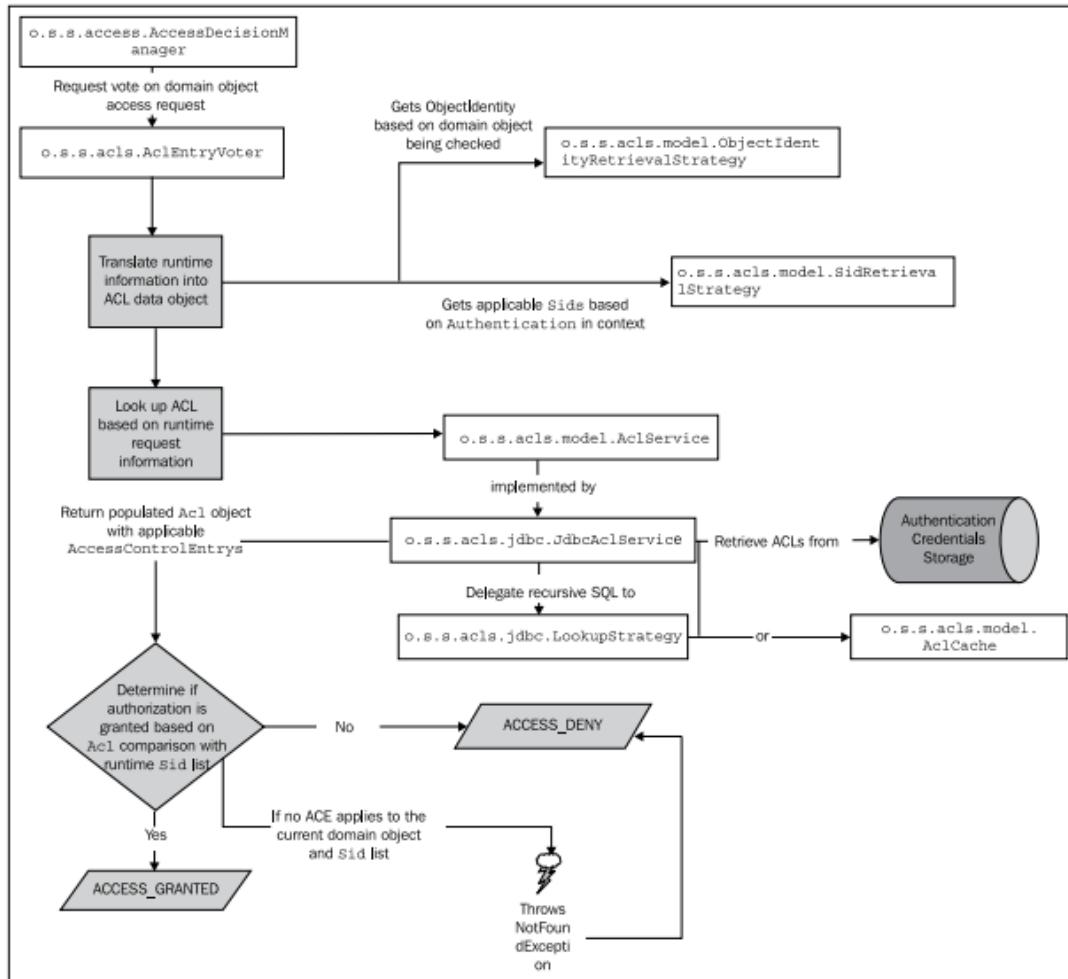


你可以看到示例的许可授权掩码有二进制整数值为 3，这是因为应用 Read 和 Write 许可后就会具有这个许可授权值了。在这个图中，所有标准的单个整数授权值都是在 BasePermission 中作为静态常量定义的。你可能会回忆起来我们在构建 ACL 配置练习中在 o.s.s.acls.AclEntryVoter 里，使用过它们中的一个常量 BasePermission.READ。

【BasePermission 所包含的逻辑常量只是在访问控制条目中经常用到的，并且在 Spring Security 中没有特殊的语义。对于非常复杂的 ACL 实现，创建自己的许可授权是很常见的，以增强独立于域或业务的最佳实践。】

一个经常困扰用户的问题是位掩码实际上是如何应用的，因为很多数据库要么不支持按位逻辑操作要么不支持可伸缩的方式。Spring ACL 通过把按位进行许可授权计算放到应用中来解决这个问题，而不是放在数据库中。

了解这个处理过程很重要，在这里我们能够看到 AclEntryVoter 怎样处理声明在方法上的许可授权（在我们的例子中，通过@Secured 注解）到真正的 ACL 授权。下图阐述了针对安全实体的请求，Spring ACL 评估声明的许可权限与相关 ACE 的过程：



我们可以看到 `AclEntryVoter` 依赖实现 `o.s.s.acls.model.ObjectIdentityRetrievalStrategy` 和 `o.s.s.acls.model.SidRetrievalStrategy` 接口的实现类，以获取适当的 `ObjectIdentity` 和 `Sids` 进行认证检查。关于这些策略有一个很重要的事情就是基于授权认证的上下文，默认的实现类如何决定要返回的 `ObjectIdentity` 和 `Sids`。

`ObjectIdentity` 有两个属性分别为 `type` 和 `identifier`，它们是根据运行时要检查的对象得到的，并用来声明 ACE 条目。默认的 `ObjectIdentityRetrievalStrategy` 使用全类名来填充 `type` 属性。`identifier` 属性通过调用实际对象实例的 `Serializable getId()` 方法得到的结果进行填充(译者注：因此进行 ACL 授权的对象需要有 `getId` 这个方法)。

【为了支持 ACL 检查你的对象并不需要实现接口，但是需要实现特定签名的一个方法这一点恐怕会让实现 Spring Security ACL 的开发人员感到惊讶。请事先规划，并确保你的域对象包含此方法！你也可以实现自己的 `ObjectRetrievalStrategy`（或内置实现的子类）来调用你选择的一个方法。但遗憾的是，这个方法的名字和类型是不可配置的。】

不幸的是，`AclImpl` 的实际实现直接将配置的 `Permission` 与 `AclEntryVoter` 进行对比，但是 `Permission` 存储在 ACE 上在数据库中，不能使用按位逻辑操作。Spring Security 社区关于这是否有意为之还有争论，但是不管怎样，当你声明一个拥有组合许可授权的用户时需要特别小心，要么 `AclEntryVoter` 必须配置上所有组合许可授权要么 ACE 需要忽略 `permission` 域本来可以配置多个值，而是要在每个 ACE 上只配置一个许可授权。

如果你想在我们简单的场景下校验它，将我们授给 `ROLE_ADMIN` SID 的权限由 `Read` 许可修改为 `Read` 和 `Write` 组合二进制掩码，也就是将其转换成 3。这需要修改 `test-acl-data.sql`:

```
insert into acl_entry (acl_object_identity, ace_order, sid, mask, granting, audit_success,
audit_failure)
select oi.id, 1, si.id, 3, true, true
from acl_object_identity oi, acl_sid si
where si.sid = 'ROLE_ADMIN';
```

如果现在你作为管理员访问 ACL 保护的分类，你会被拒绝，即使我们已经在单个 ACE 上声明了你可以进行 Read 和 Write。(译者注：不得不说，在一点上 Spring Security ACL 有点弱。)

自定义 ACL permission 声明

正如在讨论许可授权声明时所讲的那样，许可授权 (permission) 只不过是带有逻辑名的整数值。既然如此，可以扩展 `BasePermission` 来声明自己的许可权限。这里我们会涉及一个很简单的场景，创建一个新的 ACL 许可权限名为 `ADMIN_READ`。这个权限只会授予给管理员用户，并分配给只有管理员才能读的资源。尽管这对于 JBCP Pets 站点是一个比较牵强的例子，但是这种类型的自定义权限在处理个人信息时很常用（如社会保险号等——可以回忆下我们在第一章：一个不安全应用的剖析中提到的 PII）。

让我们开始要支持这个所要进行的变化。第一步就是用我们的 `com.packtpub.springsecurity.security.CustomPermission` 来扩展 `BasePermission`:

```
package com.packtpub.springsecurity.security;
// imports omitted
public class CustomPermission extends BasePermission {
    protected CustomPermission(int mask, char code) {
        super(mask, code);
    }
    protected CustomPermission(int mask) {
        super(mask);
    }
    public static final Permission ADMIN_READ = new CustomPermission(1 << 5, 'M'); // 32
}
```

接下来，我们要扩展 `o.s.s.acls.domain.PermissionFactory` 的默认实现 `o.s.s.acls.domain.DefaultPermissionFactory`，来注册我们的自定义许可权限逻辑值。`PermissionFactory` 的角色是将许可的二进制掩码转换成逻辑许可值（它可以在应用的其它部分被常量值或通过名字引用，如 `ADMIN_READ`）。`PermissionFactory` 需要所有的自定义许可权限在这里进行注册以便于查找。

我们实现 `com.packtpub.springsecurity.security.CustomPermissionFactory` class 类，如下：

```
package com.packtpub.springsecurity.security;
// imports omitted
public class CustomPermissionFactory extends DefaultPermissionFactory
{
    public CustomPermissionFactory() {
        super();
    }
}
```

```
    registerPublicPermissions(CustomPermission.class);
}

public CustomPermissionFactory(Class<? extends Permission> permissionClass) {
    super(permissionClass);
}

public CustomPermissionFactory(
    Map<String, ? extends Permission> namedPermissions) {
    super(namedPermissions);
}

}
```

我们可以看到增强了默认的构造方法以调用注册 CustomPermission，使其作为一个可用的许可授权。

【在本章的练习中我们不会强调基类的所有可用代码，但是建议你查看父类的其它功能并了解其在 ACL 系统中其它方面是如何使用的。例如，我们能够看到 buildFromName 方法在使用 ACL 自动以 JSP tag 中用到，这部分我们稍后展示。】

我们需要配置 CustomPermissionFactory 并将其织入到 BasicLookupStrategy。在 dogstore-base.xml 文件中要做以下的修改：

```
<bean class="org.springframework.security.acls.jdbc.BasicLookupStrategy" id="lookupStrategy">
    <constructor-arg ref="dataSource"/>
    <constructor-arg ref="aclCache"/>
    <constructor-arg ref="aclAuthzStrategy"/>
    <constructor-arg ref="aclAuditLogger"/>
    <property name="permissionFactory" ref="customPermissionFactory"/>
</bean>
<bean class="com.packtpub.springsecurity.security.CustomPermissionFactory"
    id="customPermissionFactory"/>
```

现在，我们的自定义 ACL 权限在 ACL 框架中可用了。接下来，我们要添加一个新的管理员，他被明确分配这个权限到第二个分类“Dog Food”上。我们要添加以下内容到 test-acl-data.sql 上，以完成这个新授权的需要：

```
-- User SID
insert into acl_sid (principal, sid) values (true, 'admin2');

-- Category #2
insert into acl_object_identity (object_id_class, object_id_
identity, parent_object, owner_sid, entries_inheriting)
select cl.id, 2, null, sid.id, false
from acl_class cl, acl_sid sid
where cl.class='com.packtpub.springsecurity.data.Category' and sid.sid='admin2';
-- Give user 'admin2' access to category 2
-- "32" == 1 << 5
insert into acl_entry (acl_object_identity, ace_order, sid, mask,
granting, audit_success, audit_failure)
select oi.id, 2, si.id, 32, true, true, true
from acl_object_identity oi, acl_sid si
where si.sid = 'admin2' and oi.object_id_identity = 2;
```

commit;

你可以看到新的整数二进制掩码值 32 已经被 ACE 数据引用了——这会对应我们在代码中定义的新 ADMIN_READ 权限。在 ACL_OBJECT_IDENTITY 表中，“Dog Food” 分类被它的主键（在 object_id_identity 列中）值 2 所引用。

我们还需要在 dogstore-base.xml 文件中声明一个新的 AclEntryVoter。

```
<bean class="org.springframework.security.acls.AclEntryVoter" id="adminResourceReadVoter">
    <constructor-arg ref="aclService"/>
    <constructor-arg value="VOTE_ADMIN_READ"/>
    <constructor-arg>
        <array>
            <util:constant
                static-field="com.packtpub.springsecurity.security.CustomPermission.ADMIN_READ"/>
        </array>
    </constructor-arg>
    <property name="processDomainObjectClass"
        value="com.packtpub.springsecurity.data.Category"/>
</bean>
```

除此以外，我们需要将这个投票器添加到访问决策管理器上，这个管理器负责在 ACL 保护方法的场景中，进行授权决策：

```
<bean class="org.springframework.security.access.vote.AffirmativeBased"
    id="aclDecisionManager">
    <property name="decisionVoters">
        <list>
            <ref bean="categoryReadVoter"/>
            <ref bean="adminResourceReadVoter"/>
        </list>
    </property>
</bean>
```

最后，我们需要在方法声明本身上添加需要的角色，在 IProductService 接口声明上：

```
public interface IProductService {
    // other methods omitted
    @Secured({"VOTE_CATEGORY_READ","VOTE_ADMIN_READ"})
    public Collection<Item> getItemsByCategory(Category cat);
}
```

在所有配置完成后，我们可以启动站点并测试 ACL 权限。基于已配置的示例数据，以下不同用户点击分类时，应该发生的结果：

用户名	Pet apparel (分类 1)	Dog Food (分类 2)	其它分类
admin	允许（通过 ROLE_ADMIN SID ACE 拥有 READ 权限）	拒绝	允许
admin2	允许（通过 ROLE_ADMIN SID ACE 拥有 READ 权限）	允许（通过安全实体 SID ACE 拥有 ADMIN_READ 权限）	允许
guest	拒绝	拒绝	允许

可以看到即使使用我们简单的例子，也能扩展 Spring ACL 的功能。当然我们使用了很有

限的方式来说明这个定义良好的访问控制系统的强大功能，这个系统是建立在安全实体、GrantedAuthority、单个域对象以及业务方法之上。

在 JSP 中使用 Spring Security JSP tag 库启动 ACL

在第三章：增强用户体验和第五章中看到，Spring Security 的 JSP tag 库提供了暴露认证相关的数据给用户的功能以及基于各种规则限制能看到的内容。

相同的 tag 库也能够内置地与使用 ACL 的系统交互。从我们上面简单的例子中，我们已经围绕首页上前两个分类配置了一个简单的 ACL 授权场景。

让我们更进一步，使用`<accesscontrollist>`tag 来隐藏用户实际不能访问的分类。

请参考我们前面的表格来了解到此为止，我们已经配置的访问规则。

我们将显示每个分类用`<accesscontrollist>`tag 包围起来，声明对这个显示对象要进行的权限检查：

```
<c:forEach var="category" items="${categories}">
<security:accesscontrollist hasPermission="READ,ADMIN_READ" domainObject="${category}">
    <li><a href="category.do?id=${category.name}">${category.name}</a></li>
</security:accesscontrollist>
</c:forEach>
```

请想一下，我们期望在这里发生什么——我们想要用户只能看到他有 READ 或 ADMIN_READ（我们自定义的许可权限）权限的条目。所以我们声明了一个逗号分隔的权限列表以及要进行检查的域对象（通过 JSP EL 的表达式\${category}来指定）。

在背后，这个 tag 的实现使用我们前面讨论的相同 SidRetrievalStrategy 和 ObjectIdentityRetrievalStrategy，所以它与方法安全中使用的 ACL 具有相同的访问检查计算流程。

支持 ACL 的 Spring 表达式语言

SpEL 对 ACL 系统的支持仅限于方法安全，通过使用 hasPermission SpEL 方法。典型情况下，这种类型的访问检查会与引用一个或多个传入参数（进行@PreAuthorize 检查）或集合过滤（进行@PostAuthorize 检查）联合使用。

遗憾的是，启用 ACL 方法安全配置需要我们配置所有的方法安全以明确的 Spring Bean 的方式。这样，我们就需要我们移除`<global-method-security>`元素并替换为我们在第六章中介绍过的明确方式配置方法安全。鉴于我们这里不想重复配置（尽管它包含在本章代码中），我们所要做的只是做一下如下的细微修改，ACL hasPermission 检查所需要的类就可用：

```
<bean class="org.springframework.security.access.expression.method.
DefaultMethodSecurityExpressionHandler" id="methodExprHandler">
    <property name="permissionEvaluator" ref="aclPermissionEvaluator"/>
</bean>
<bean class="org.springframework.security.acls.AclPermissionEvaluator"
id="aclPermissionEvaluator">
    <constructor-arg ref="aclService"/>
    <property name="permissionFactory" ref="customPermissionFactory"/>
```

```
</bean>
```

我们更新的 methodExprHandler 的 bean 定义，配置了 o.s.s.access.PermissionEvaluator 的实现（默认的 PermissionEvaluator 实现会拒绝所有的许可认证检查）。o.s.s.acls.AclPermissionEvaluator 使用了 AclService 及相关的类去实际检查 SpEL 表达式声明的许可权限。

随着配置完成，我们可以使用一个替代的方法来过滤显示在首页上的分类列表（确保你已经移除了上一个练习所添加的 tag 库）。

【注意，hasPermission 方法不再支持逗号分隔的许可授权（正如我们在<accesscontrollist> JSP tag 中见到的那样），所以我们需要使用 SpEL 的布尔逻辑。】

只需简单的添加以下声明到 IProductService 接口上：

```
@PostFilter("hasPermission(filterObject, 'READ') or hasPermission(filterObject, 'ADMIN_READ')")  
Collection<Category> getCategories();
```

现在，重启应用并比较以匿名用户、admin、admin2 进入时首页的分类列表。注意这个显示列表是如何通过 ACL 许可授权完成的？我们可以看到 SpEL 的 hasPermission 方法很好的与使用 ACL 的应用通过方法安全注解结合起来。

易变的 ACL（Mutable ACLs）和授权

尽管 JBCP Pets 站点没有实现完整的用户管理功能，但是你的应用可能会有这些通用的功能如：新用户注册以及管理用户维护。到此时，缺少这些功能——我们是通过在应用初始化的时候用 SQL 插入的方法代替的——并没有阻止我们演示 Spring Security 和 Spring ACL 的很多功能。

但是，在运行时适当的处理声明的 ACL、添加或删除系统用户，对于基于 ACL 的授权环境来说保证一致性和安全很重要。Spring ACL 通过易变的 ACL（o.s.s.acls.model.MutableAcl）来解决这个问题。

扩展自 Acl 接口的 MutableAcl 允许运行时操作 ACL 域以实现修改某个特定 ACL 的内存表现。这个功能包括新增、更新以及删除 ACE，修改 ACE 的拥有者和其它有用的功能。

那么，我们可能会期望 Spring ACL 模块能够有内置的方式将运行时的 ACL 变化持久化到 JDBC 数据存储中，它确实如此。o.s.s.acls.jdbc.JdbcMutableAclService 能用来创建、更新和删除数据库中的 MutableAcl 实例，并且管理其它支持 ACL 的表（能够处理 SID，ObjectIdentity 以及域对象类名）。

对我们来说，它只需要一个简单的配置来使用 JdbcMutableAclService 替换 JdbcAclService——易变 service 需要一个对 ACL 缓存的引用，这样它就能在更新数据库条目的时候将缓存删除。这个 bean 的配置很简单：

```
<bean class="org.springframework.security.acls.jdbc.JdbcMutableAclService"  
id="mutableAclService">  
    <constructor-arg ref="dataSource"/>  
    <constructor-arg ref="lookupStrategy"/>  
    <constructor-arg ref="aclCache"/>  
</bean>
```

请回忆在本章前面，AclAuthorizationStrategyImpl 允许我们指定对易变 ACL 进行操作所需要的角色。它们以 bean 配置的方式提供给构造方法。构造参数以及它们的意义，如下：

参数序号	做什么的
1	表明安全实体要修改 ACL 保护对象拥有者所要拥有的角色
2	表明安全实体要修改 ACL 保护对象审计所要拥有的角色
3	表明安全实体要对 ACL 保护对象进行其它修改（新建、更新和删除）所要拥有的角色

译者注：以上的三个参数的作用就是要保证对保护的对象进行操作时，登录人必须有足够的权限。

`JdbcMutableAclService` 包含一系列的方法在运行时来操作 ACL 和 ACE 数据。尽管这些方法本身很容易理解（`createAcl`, `updateAcl`, `deleteAcl`），但是正确的配置和使用 `JdbcMutableAclService` 经常对 Spring Security 的高级用户都很困难。

让我们实现一个 ACL 启动类它会替换我们的启动脚本，并用代码的方式插入当前的 ACL 和 ACE 条目（以及支持的 `ObjectIdentity` 和 `Sid`）。

配置 Spring 事务管理器

`JdbcMutableAclService` 使用 Spring 的 `JdbcTemplate` 来与 JDBC `DataSource` 进行交互。所以，它需要一个 Spring JDBC `PlatformTransactionManager` 以保证（很有侵入性）所有与数据库的交互正确地包装在事务中。

大多数实际使用 Spring 的应用可能已经有一个声明的事务管理器，但是我们的 JBCP Pets 应用并没有。我们在 `dogstore-base.xml` 添加一个：

```
<bean id="txManager"
  class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
  <property name="dataSource" ref="dataSource"/>
</bean>
```

声明一个对启动类的应用，一会将会对它进行编码：

```
<bean class="com.packtpub.springsecurity.security.AclBootstrapBean"
  init-method="aclBootstrap"/>
```

很像在第四章：凭证安全存储中的 `DatabasePasswordSecurerBean`，我们配置这个 bean 使得 Spring `ApplicationContext` 初始化时，`aclBootstrap` 方法会被触发——恰当的时间来启动我们应用中的 ACL 数据。

与 `JdbcMutableAclService` 交互

现在，我们要编写启动 bean —— 创建 `com.packtpub.springsecurity.security.AclBootstrapBean` 类。首先我们使用 `@Autowired` 注入需要的依赖：

```
package com.packtpub.springsecurity.security;
// imports omitted
public class AclBootstrapBean {
  @Autowired
  MutableAclService mutableAclService;
  @Autowired
```

```
IProductDao productDao;  
@Autowired  
PlatformTransactionManager transactionManager;
```

接下来是方法的实际定义，它将会被触发以启动 ACL 数据——我们将会每次分析一个片段：

```
public void aclBootstrap() {  
    // domain data to set up  
    Collection<Category> categories = productDao.getCategories();  
    Iterator<Category> iterator = categories.iterator();  
    final Category category1 = iterator.next();  
    final Category category2 = iterator.next();
```

我们需要引用实际要保护的域对象以便于为它们创建 ACL。回忆一下，我们的 ACL 启动 SQL 只对前两个分类添加条目，所以我们要把它们从 ProductDAO 中取出来。

JdbcMutableAclService 需要使用 ObjectIdentity 来创建初始的 MutableAcl（它能够被进一步的管理，我们等会能看到）。为了创建 ObjectIdentity，我们需要真正的域对象。

```
// needed because MutableAclService requires a current authenticated principal  
GrantedAuthorityImpl roleUser = new GrantedAuthorityImpl("ROLE_USER");  
GrantedAuthorityImpl roleAdmin = new GrantedAuthorityImpl("ROLE_ADMIN");  
UsernamePasswordAuthenticationToken          token          =          new  
UsernamePasswordAuthenticationToken("admin", "admin", Arrays.asList(new  
GrantedAuthority[]{roleUser, roleAdmin}));  
SecurityContextHolder.getContext().setAuthentication(token);
```

JdbcMutableAclService 会验证一个用户已经登录，并将其作为建立的 MutableAcl 的默认拥有者。遗憾的是，这是一个强制检查，即便随后你明确设置 ACL 拥有者（也不行）。因为这些代码执行时没有认证过的用户，所以我们让 JdbcMutableAclService 把 admin 用户作为当前认证过的用户（译者注：就是上面最后两句代码）。

```
// sids  
final Sid userRole = new GrantedAuthoritySid("ROLE_USER");  
final Sid adminRole = new GrantedAuthoritySid("ROLE_ADMIN");  
  
// users  
final Sid adminUser = new PrincipalSid("admin");  
final Sid admin2User = new PrincipalSid("admin2");
```

我们需要为安全实体和组创建 Sid，它们是 ACL 和 ACE 的结构，所以在里我们明确创建它们。记住的是，如果你在应用范围内管理 ACL（例如，通过 UI 层调用业务服务完成），你可以以不同的方式处理这个问题——这里的代码只是一个例子，我们鼓励你根据情况作出调整。

```
// all interaction with JdbcMutableAclService must be within a  
transaction  
TransactionTemplate tt = new TransactionTemplate(transactionManager);  
tt.execute(new TransactionCallbackWithoutResult() {  
    @Override  
    protected void doInTransactionWithoutResult(TransactionStatus arg0)  
    {  
        // category 1 ACL
```

```
MutableAcl createAclCategory1 = mutableAclService.createAcl(new ObjectIdentityImpl(category1));
createAclCategory1.setOwner(adminRole);
createAclCategory1.insertAce(0, BasePermission.READ, adminRole,
true);
mutableAclService.updateAcl(createAclCategory1);
// category 2 ACL
MutableAcl createAclCategory2 = mutableAclService.createAcl(new ObjectIdentityImpl(category2));
createAclCategory2.setOwner(admin2User);
createAclCategory2.insertAce(0, CustomPermission.ADMIN_READ,
admin2User, true);
mutableAclService.updateAcl(createAclCategory2);
});
SecurityContextHolder.clearContext();
}
}
```

在一个新的事务范围内，与 `JdbcMutableAclService` 的交互完成了。我们可以看到初始的 `createAcl` 调用返回一个 `MutableAcl`。在这背后，针对 `ObjectIdentity` 和 `Sid` 进行了数据库插入和查找。`MutableAcl` 本身提供了方法来创建、更新和删除 ACL 中的 ACE（记住，ACE 声明的是单个的权限-SID 匹配）。最后 `MutableAcl` 通过 `updateAcl` 方法调用更新到数据库中。

要注意的是 `JdbcMutableAclService` 还负责确保 `MutableAcl` 操作进行时，`AclCache` 被更新。

建议你体验易变的 ACL 服务并增强这个站点以支持用户的新建和编辑——`JdbcMutableAclService` 有很好的文档（在代码层面），尝试实现它会是一个很好的练习，这取决于你是否愿意实现完整的运行时驱动的 ACL 模型。

Ehcace ACL 缓存

`Ehcace` 是一个开源的、内存和基于硬盘的缓存库，它被广泛用在很多开源和商用 Java 产品中。正如我们在本章前面提到的，`Spring Security` 提供了一个默认的 ACL 缓存实现，它依赖于一个配置的 `Ehcace` 实例，它会存储 ACL 信息并优先于在数据库中读取 ACL。

鉴于在本节我们不想过多介绍详细配置 `Ehcace`，我们将会涉及到 `Spring ACL` 如何使用 `cache` 并介绍一个简单的默认配置。

配置 Ehcace ACL 缓存

建立 `Ehcace` 很简单——我们需要简单地声明两个 `Spring Core` 的 bean，它们管理 `Ehcace` 实例并暴露几个有用的配置属性：

```
<bean class="org.springframework.cache.ehcace.EhCacheManagerFactoryBean"
id="ehCacheManagerBean"/>
<bean class="org.springframework.cache.ehcace.EhCacheFactoryBean"
id="ehCacheFactoryBean">
    <property name="cacheManager" ref="ehCacheManagerBean"/>
```

```
</bean>
```

接下来，我们要实例化 Ehcache ACL 缓存 bean:

```
<bean class="org.springframework.security.acls.domain.EhCacheBasedAclCache"
  id="ehCacheAclCache">
  <constructor-arg ref="ehCacheFactoryBean"/>
</bean>
```

最后，我们将 NullAclCache 实现替换为基于 Ehcache 的实现:

```
<bean class="org.springframework.security.acls.jdbc.BasicLookupStrategy" id="lookupStrategy">
  <constructor-arg ref="dataSource"/>
  <constructor-arg ref="ehCacheAclCache"/>
  <constructor-arg ref="aclAuthzStrategy"/>
  <constructor-arg ref="aclAuditLogger"/>
</bean>
```

当这些配置完成（并且 Ehcache 运行时 JAR 在你的 classpath 中），ACL 数据将会基于 Ehcache 缓存管理器的配置进行缓存。

Spring ACL 怎样使用 Ehcache

前面介绍的配置步骤可能会比较简单，添加 Ehcache 到你的应用中——尤其是大量使用——要进行细致的分析，比较缓存的成本和数据库查询的成本。理解 Ehcache 在 Spring ACL 中是如何使用的对于规划缓存大小和寿命很重要。

作为 ACL 缓存策略的一部分，Spring ACL 将会存储以下所有的对象（它们的大多数在 o.s.s.acls.domain）到缓存中，要么作为 key 要么作为值：

- ObjectIdentity（实现类为 ObjectIdentityImpl）；
- Sid（实现类为 GrantedAuthoritySid 或 PrincipalSid）；
- Acl（实现类为 AclImpl），包含 AccessControlEntry（实现类为 AccessControlEntryImpl）；
- 你对象实例的 Serializable 类型主键（一般为 Long，除非你对 Spring ACL 运行时类做了重要的个性化）

只有 BasicLookupStrategy 和 MutableAclService 是 ACL 缓存机制的用户，使用缓存很简单。关于缓存中条目的大小，比较好的办法是在合适的负载测试中监控缓存以评估缓存元素的内存使用和存在寿命。

如果你的应用为了其它的目的（如 Hibernate 或其它 ORM 根据）已经使用了 Ehcache，你可能愿意将用于 ORM 目的的缓存实例和用于存储 ACL 数据的缓存实例区分开来。一般的做法是在构建用于 Spring ACL 的 EhCacheFactoryBean 时，提供唯一的 cache 名，如下：

```
<bean class="org.springframework.cache.ehcache.EhCacheFactoryBean"
  id="ehCacheFactoryBean">
  <property name="cacheManager" ref="ehCacheManagerBean"/>
  <property name="cacheName" value="springAclCacheRegion"/>
</bean>
```

奇怪的是，除了 Javadoc 以外，Spring Core 中对于 Ehcache 的支持没有任何正规的文档。如果使用 Spring ACL 和 Ehcache 对你很重要，那你很可能需要自己深入研读代码。

典型 ACL 部署所要考虑的事情

实际部署 Spring ACL 到业务应用是很复杂的。我们总结了 Spring ACL 要注意的事情，它们在大多数 Spring ACL 实现场景中都存在。

关于 ACL 的伸缩性和性能模型

对于小型和中型应用，添加 ACL 功能是很容易的，尽管它增加数据库存储和影响运行时性能，这个影响可能不会那么明显。但是，取决于 ACL 和 ACE 建模的粒度，在中到大型应用中，数据库的行数可能会非常惊人，甚至对熟练的 DBA 都是很难的任务。

让我们假设我们将要扩展 ACL 以覆盖 JBCP Pets 应用大多数的功能，包括用户账号和订单，还包括 JBCP Pets 中顾客论坛的帖子。我们对着数据建模如下：

- 所有的顾客都有账号；
- 10%的顾客拥有订单。其中每个顾客的平均订单数是 2；
- 订单对顾客是要保护的（read-only），但是管理员也能访问（read/write）；
- 10%的顾客会在顾客论坛上发帖，其中每个顾客的帖子数量是 20；
- 帖子对顾客是要进行保护的（read-write），对管理员也是如此。帖子对其它顾客是 read-only 的。

基于我们对 ACL 系统的了解，我们知道数据库表有以下的可伸缩相关属性：

表	是否随数据伸缩	可伸缩性注释
ACL_CLASS	NO	每个域类需要一行
ACL_SID	Yes (User)	每个角色（GrantedAuthority）需要一行 每个用户账号需要一行（如果域对象根据用户保护）
ACL_OBJECT_IDENTITY	Yes (域类*每个类的实例数)	每个保护的域对象需要一行
ACL_ENTRY	Yes(域对象实例*单个的 ACE 条目)	每个 ACE 需要一行，对于每个域对象可能要多行

我们可以看到 ACL_CLASS 并没有伸缩性的考虑（大多数的系统少于 1000 个域类）。ACL_SID 是基于系统中的用户数线性伸缩的。这可能不用考虑，因为其它用户相关的表也以这种方式处理伸缩性（如用户账号等）。

要考虑的两个表是 ACL_OBJECT_IDENTITY 和 ACL_ENTRY。如果对一个用户有一个订单这种情况计算需要的行数，我的得到如下的估算结果：

表	每个订单的 ACL 数据	每个论坛帖子的 ACL 数据
ACL_OBJECT_IDENTITY	每个订单需要一行数据	每个帖子需要一行数据
ACL_ENTRY	三行——一行用于拥有者（即顾客的 SID）的 read 访问，两行（一行用于读访问，一行用于写访问）用于管理组的 SID。	四行——一行用于顾客组 SID 的读访问，一行用于拥有者的写访问，两行用于管理组（如同订单）

我们可以使用以上的假设并计算出 ACL 的伸缩性矩阵：

表/对象	伸缩性因素	估算（低）	估算（高）
用户 (Users)		10,000	1,000,000
订单 (Orders)	# Users * 0.1 * 2	2,000	200,000
论 坛 帖 子 (ForumPosts)	# Users * 0.1 * 20	20,000	2,000,000
ACL_SID	# Users	10,000	1,000,000
ACL_OBJECT_IDENTITY	# Orders + # Posts	22,000	2,200,000
ACL_ENTRY	(# Orders * 3) + (# Posts * 4)	86,000	8,600,000

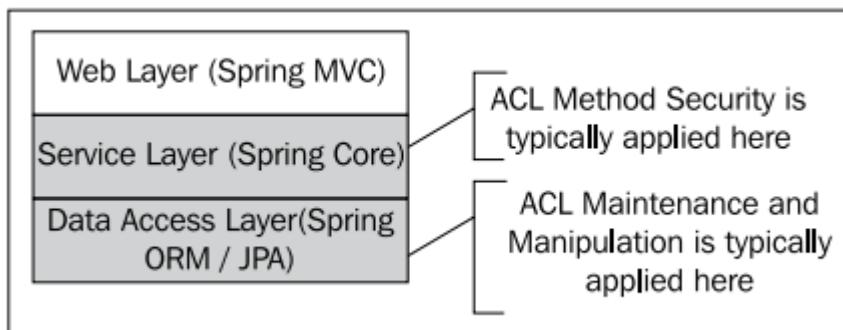
在典型的 ACL 实现中，这些预测只是要关联和保护对象的子集，你可以看到存储 ACL 数据的数据库行数是随着业务数据线性（或更快）增长的。特别是对于大型系统的规划，预测你可能用到的 ACL 数据特别重要。对于非常复杂的系统拥有数亿行关于 ACL 存储的数据并不罕见。

不要忽视自定义开发的成本

使用 Spring ACL 的安全环境通常需要明显的开发工作以及我们讲过的配置步骤。我们例子的配置场景有以下的限制，这可能会影响到将 ACL 安全应用到完整站点上：

- SID 和访问凭证硬编码在应用启动的时候；
- 没有提供管理域对象（创建和删除）或管理用户、组的功能；
- 应用没有有效使用 ACL 的等级体系。

当整个应用规划 Spring ACL 时，你必须仔细考虑所有域数据管理的地方，并确保这些地方正确更新 ACL 和 ACE 规则并失效缓存。一般来说，方法和数据安全发生在应用的服务或业务层，而维护 ACL 和 ACE 所需要的钩子（hook）通常发生在数据访问层。



如果你正在处理一个标准应用的架构，拥有合适的隔离及功能封装，可能会很容易找到一个中央位置标识这些变化。但是，如果你正在处理一个遗留的（或者开始的时候出来没有设计）架构，添加 ACL 功能并在数据操作的代码中支持钩子（hook）将会很困难。

正如前面所提到的，需要注意的是 Spring ACL 自从 Acegi 1.x 时代就没有明显的变化过，大约三年了。在那时，很多用户尝试使用它，并记录和以文档的形式提出了很多限制，它们中的很多保存在 Spring Security JIRA 库中 (<http://jira.springframework.org/>)。缺陷 SEC-479 可以作为很多关键限制的入口，它们中的很多在 Spring Security3 中依然没有处理，（如果它们适用于你的场景）可能会需要很大的自定义编码工作。

以下是一些最重要和常见的缺陷：

- ACL 基础设施需要数字型的主键。对于使用 GUID 或 UUID 主键（近来用的越来越

多，因为现代数据库提供了高效支持）的应用，这会是很大的限制；

- JIRA 缺陷 SEC-1140 记录的缺陷，默认 ACL 实现不能用按位操作正确的对比 Permission 二进制掩码。在关于许可授权一节，我们已经提到；
- 配置 Spring ACL 的方法与其它 Spring Security 功能存在一些不一致。简单来说，在这个功能领域，类代理和属性并没有通过 DI（依赖注入）暴露，需要覆盖或重写策略会很耗时且维护代价高昂；
- Permission 的二进制掩码通过整数（integer）实现，所以最多有 32 个可能的位。比较常见的做法是扩展默认的未分配来声明单个对象属性的权限（例如，为读的员工社会保险号分配一个位）。复杂的部署可能会每个域对象超过 32 个属性，在这种情况下唯一可选的就是为了这个限制，重新对你的域对象建模。

取决于你特定应用的需要，可能会遇到新的问题，尤其是关于这些实现自定义功能要修改的类。

我应该用 Spring Security 的 ACL 吗？

正如使用整体使用 Spring Security 是很强的业务依赖，使用 Spring ACL 也是这样——实际上，这对于 ACL 可能更明显，因为它紧密连接到业务方法和域对象上了。我们希望这个关于 Spring ACL 的指导已经为你描述了重要的各层配置和 Spring ACL 的概念，这用来分析 Spring ACL 在你的应用中如何使用，并帮助你决定和匹配它的功能到实际应用中。

小结

在本章中，我们关注访问控制列表安全以及对于这种类型的安全 Spring Security ACL 模块是如何实现的。我们所做如下：

- 了解访问控制列表的基本概念，以及为什么说它们是授权的有效解决方式；
- 学习 Spring ACL 实现的关键概念，包括访问控制条目、SID 以及对象标识；
- 考察支持 ACL 系统的数据库模式以及逻辑设计；
- 配置所有需要的 Spring Bean 来启用 Spring ACL 模块并增强了一个服务接口来使用注解的方法授权。我们接下来将数据库中已有的用户以及站点使用的业务对象，变成 ACE 声明和辅助的实例数据；
- 了解 Spring ACL 许可授权处理相关的概念；
- 扩展了我们关于 SpringSecurity JSP 标签库和 SpEL 表达书语言（用于方法安全）的知识来实现 ACL 检查；
- 讨论易变的 ACL 概念，并了解在易变的 ACL 环境中的基本配置和需要自定义的代码；
- 开发了一个自定义的 ACL 许可授权并配置应用验证器有效性；
- 配置和分析使用 Ehcache 缓存管理器以减少 Spring ACL 的数据库影响；
- 分析在复杂业务应用中使用 Spring ACL 的影响以及设计考虑因素。

这完成了本书中关于 Spring Security 关键概念的部分。在接下来的章节中，我们将会深入讨论 Spring Security 认证与多种类型的外部系统进行集成。如果你不知道这些系统（OpenID、LDAP 等）背后的技术，我们也将带领你进行学习，所以请继续阅读。

第八章 对 OpenID 开放

OpenID 是很流行的可信任身份管理方式，它允许用户通过一个单独的可信任提供者（provider）管理其身份信息。这个便利的功能为用户提供了安全的方式即使用可信任的 OpenID 提供者来存储器密码和个人信息，并可以随意的基于请求获取其个人信息。另外，启用 OpenID 功能的站点能够确信用户提供的 OpenID 凭证信息就是他们所说的人。

在本章中，我们将会：

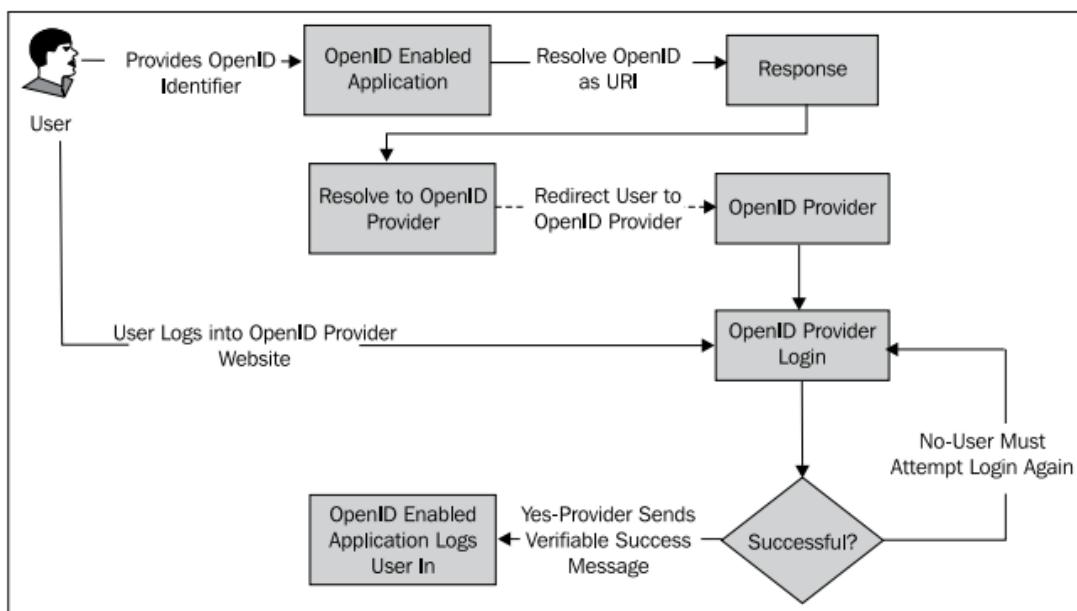
- 学习在五分钟之内建立自己的 OpenID；
- 使用快速实现的 OpenID 来配置 JBCP Pets 站点；
- 学习 OpenID 的概念架构以及它怎样为你的站点提供可信任的用户访问；
- 实现基于 OpenID 的用户注册；
- 体验 OpenID 属性交换得到用户简介功能；
- 检查基于 OpenID 登录的安全性。

OpenID 承诺的世界

作为一项技术，OpenID 的承诺是允许 web 上的用户通过一个可信任的提供者集中管理他们的个人数据和信息，然后这个可信任的提供者作为代理与用户想交互的站点建立起互信。

概念上来说，这种类型的登录要通过一个长时间存在的可信任第三方，可以有多种方式（如 Microsoft Passport，曾经是最知名的中心登录服务）。OpenID 的区别优势在于 OpenID 提供者（OpenID Provider）仅仅需要实现一个协议，这个协议与任何试图通过 OpenID 进行集成登录的站点相协调。OpenID 规范本身也是一个开放的规范，这就导致出现了不同的提供者但运行相同的协议。这对健康竞争是一种好事，也有利于客户的选择。

下图整体说明了一个集成 OpenID 的站点在登录过程中与 OpenID 提供者之间的关系：



我们可以看到用户以唯一名字标识的方式提供自己的凭证信息，一般是统一资源标识符（Uniform Resource Identifier, URI），这是 OpenID 提供者分配给用户的，用来唯一标识用户

和 OpenID 提供者。这通常是 OpenID 提供者 URI 的子域名(如: <https://jamesgosling.myopenid.com/>)，或者在 OpenID 提供者 URI 上添加唯一标识(如:<https://me.yahoo.com/jamesgosling>)。我们可以看到这两种方式的 URL 都能很明确的标识出 OpenID 提供者（通过域名）和唯一的用户标识。

【不要不加考虑地信任 OpenID。在这里你看到我们可以伪装系统中的用户。我们可以用一个 OpenID 登录，它可能会以为我们是 James Gosling，当然我们不是。不要做这样错误的假设，即因为有一个听起来令人信服的 OpenID（或 OpenID 代理提供者）那他们就是认证过的人了，而不需要额外方式的认证。以另外的方式想一下这个问题，如果有个人敲你的门并说自己是 James Gosling，你会不检查他的 ID 就让他进来吗？】

启用 OpenID 的应用接下来会重定向到 OpenID 提供者哪里，在这里用户提供他的认证信息，OpenID 提供者负责做出访问决定（译者注：即登录是否成功）。一旦提供者做出访问决定，它将用户重定向会原始的站点，这是可以确信用户的真实性了。

如果你开始进行尝试，那 OpenID 就更容易理解了。让我们添加 OpenID 到 JBCP Pets 登录页上。

注册一个 OpenID

为了完全体现本节中这个练习的价值（并测试登录），你需要在众多可用的 OpenID 中选择一个拥有自己的 OpenID，他们的列表可以在这里看到：<http://openid.net/get-an-openid/>。有些通用的 OpenID 提供者，你可能已经有它们的账号了如 Yahoo!, AOL, Flickr 或 MySpace。Google 的 OpenID 支持略有不同，在本章后面的添加 Sign In with Google 到登录页时，我们将会看到。为了完整的完成本章的练习，我们建议你最少有以下的账号：

- myOpenID
- Google

使用 Spring Security 启用 OpenID 认证

在接下来的几章介绍外部认证提供者时，我们将会看到一个通用的模式。关于与 Spring 系统之外的提供者集成，Spring Security 提供了便利的包装。

在这里，[openid4java 项目](http://code.google.com/p/openid4java/) (<http://code.google.com/p/openid4java/>) 为 Spring Security 的 OpenID 功能提供了底层 OpenID 提供者发现和请求/响应握手（negotiation）的功能。

编写一个 OpenID 登录表单

常见的方式是站点在一个登录页上提供了标准（用户名和密码）和 OpenID 两种登录选择，允许用户在两种方式中选择，JBCP Pets 的最终登录页如下：

Please Log Into Your Account

Please use the form below to log into your account.

Login:

Remember Me?

Password:

Or, Log Into Your Account with OpenID

Please use the form below to log into your account with OpenID.

Login:



基于 OpenID 的 form 代码如下：

```
<h1>Or, Log Into Your Account with OpenID</h1>
<p>
    Please use the form below to log into your account with OpenID.
</p>
<form action="j_spring_openid_security_check" method="post">
    <label for="openid_identifier">Login</label>;
    <input id="openid_identifier" name="openid_identifier" size="20" maxlength="100"
type="text"/>
    
    <br />
    <input type="submit" value="Login"/>
</form>
```

Form 域的名字即 `openid_identifier` 是有特定含义的。OpenID 规范建议使用的站点以这个名字作为 OpenID 登录域，所以用户客户端（浏览器）对于这个域的功能也具有语义理解。甚至有浏览器插件如 Verisign's OpenID SeatBelt (<https://pip.verisignlabs.com/seatbelt.do>)，它能够利用这个语义理解，预先在识别的页面把 OpenID 凭证填入 OpenID 域中。

你可能会意识到在 OpenID 登录中没有提供 `remember me` 选项。这是因为从重定向到提供者再回来，导致 `remember me` 复选框的值丢失，所以当用户登录成功后，他们不再有 `remember me` 选项。这很遗憾，但是这却增强了我们 OpenID 登录机制的安全性，因为 OpenID 强制用户每次登录时都与提供者建立可信任的关系。

在 Spring Security 中配置支持 OpenID

回到基本的 OpenID，它包括一个 `servlet` 过滤器和认证提供者，只要在 `dogstore-security.xml` 文件中的 `<http>` 配置元素中添加一个指令即可：

```
<http auto-config="true" ...>
<!-- Omitting content... -->
<openid-login/>
```

</http>

在添加完这个配置元素后并重启应用，你可以重启应用，你能够使用 OpenID 登录 form 提供 OpenID 并会执行 OpenID 认证流程。但是当你回到 JBCP Pets，你会被拒绝访问。这是因为你的凭证并没有任何角色。接下来，我们要处理它。

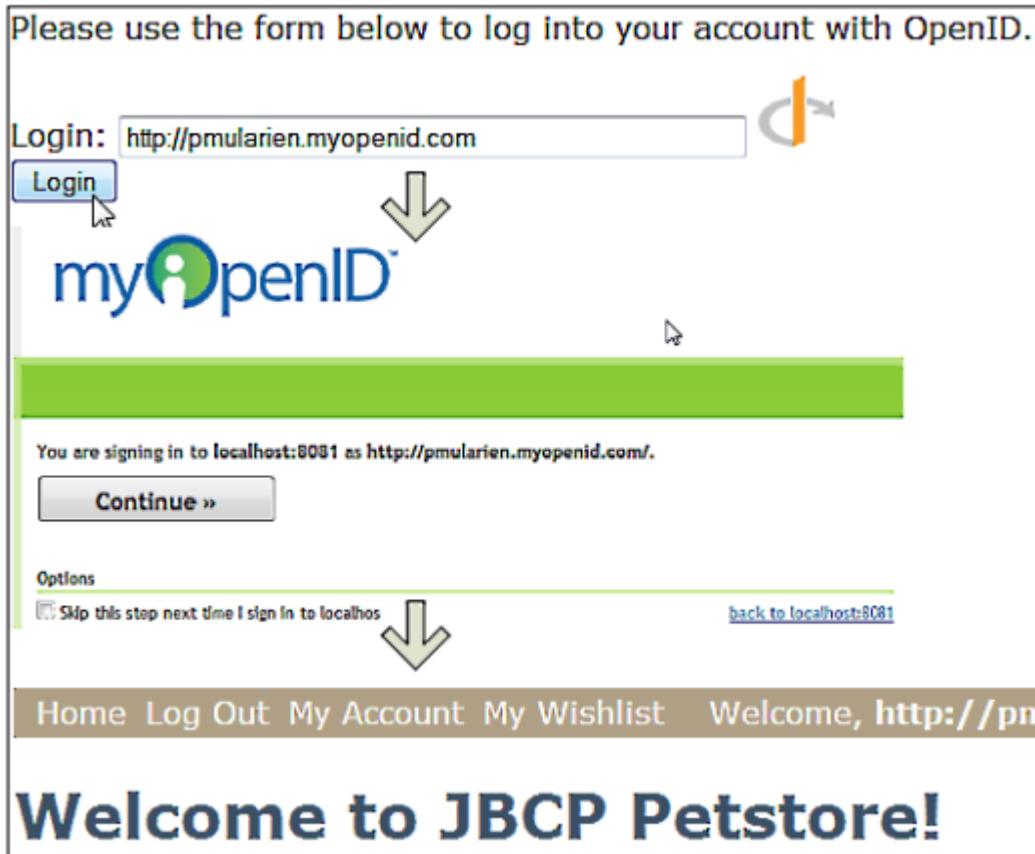
添加 OpenID 用户

因为现在我们还没有使用 OpenID 的新用户注册，所以需要手动的插入用户账号（测试所用）到数据库中，这是通过添加到 test-users-groups-data.sql 数据库启动代码中做到的。我们推荐在这一步你使用 myOpenID（注意，如果使用 Yahoo! 会有问题，原因我们下面介绍）。假设我们的 OpenID 是 <https://jamesgosling.myopenid.com/>，那要插入这个文件的 SQL 如下：

```
insert      into      users(username,      password,      enabled,      salt)      values
('https://jamesgosling.myopenid.com/','unused',true,CAST(RAND()*1000000000 AS varchar));
insert into group_members(group_id, username) select id,'https://jamesgosling.myopenid.com/'
from groups where group_name='Administrators';
```

你可能会发现这与我们插入传统的基于用户名和密码的 admin 账号很类似，只是我们使用“unused”作为密码。我们这样做，当然是因为基于 OpenID 的登录并不需要我们的站点存储用户的密码。但是，细心的读者可能会发现，这并不允许用户创建一个任意的用户名和密码并将其与 OpenID 关联——我们将会在本章后面简单介绍这个过程，你也可以作为使用这项技术的高级用法，自己探索如何实现。

此时，你可以完成使用 OpenID 实现完整登录。重定向的顺序通过以下的截图以箭头的方式进行了描述：



现在我们有了使用 OpenID 的 JBCP Pets 登录！可以测试多个 OpenID 提供者。你会发现，尽管整体的功能是一样的，但是不同提供者的检查和接受 OpenID 的体验是不同的。

OpenID 用户的注册问题

请使用我们前面的技术来测试 Yahoo! OpenID——例如，<https://me.yahoo.com/pmularien>。你会发现它并不好用，像其它 OpenID 提供者那样。这带出了 OpenID 结构的一个很重要的问题，并体现出了启用 OpenID 用户注册的重要性。

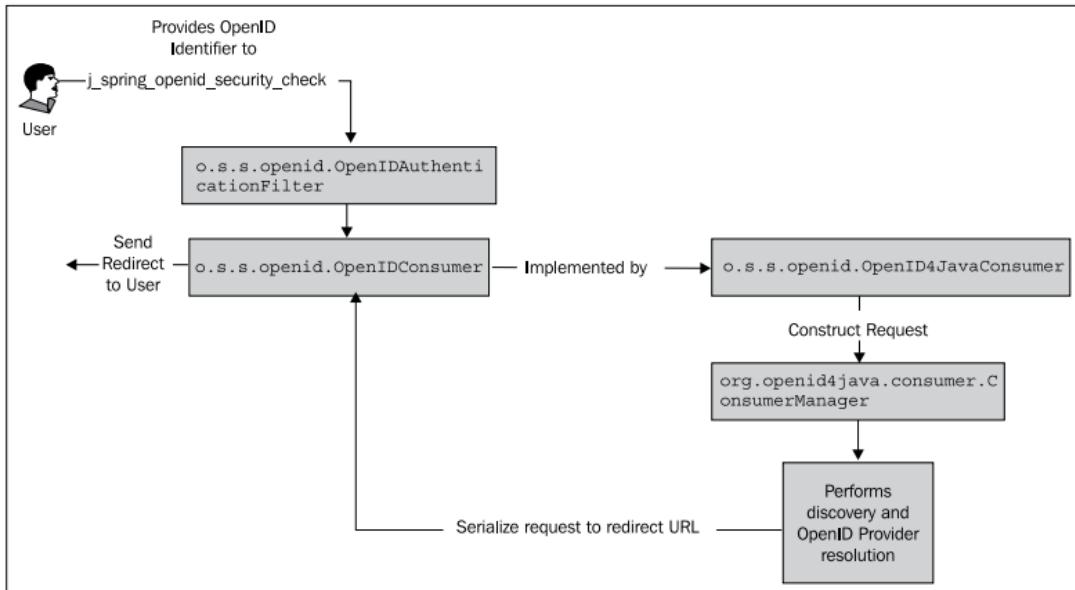
OpenID 标识是如何处理的

Yahoo! 返回的实际 OpenID 类似于如下的样子：<https://me.yahoo.com/pmularien#9a466>。在 OpenID 术语中，用户在登录框中输入的标识符被称为用户提供的标识符（user-supplied identifier）。这个标识符可能并不对应唯一的用户标识符（用户的声明标识符，claimed identifier），但是作为鉴别所有者的一部分，OpenID 提供者将会把用户的输入转换成提供者能真正证明用户拥有的标识符。（译者注：即用户记住的标识符与 OpenID Provider 返回的标识符可能并不一致）。

【OpenID 发现协议以及 OpenID 提供者本身实际上能够很奇妙地基于 OpenID 认证请求计算出用户是谁。例如，尝试输入 OpenID 提供者（如 www.yahoo.com）的名字在 OpenID 登录框中——你会看到一个不同的界面让你输入 OpenID，因为你并没有在登录框中提供唯一的 OpenID。很聪明！关注这个还有更多 OpenID 的规范，可以 OpenID 组织站点的规范页面（在开发者页面中），<http://openid.net/developers/>。】

一旦用户能够提供他拥有的声明标识符，OpenID 提供者将会返回给请求拥有一个声明标识符的正规版本，这被叫做 OpenID 提供者本地标识符（OpenID Provider Local Identifier 或 OP-Local Identifier）。这是 OpenID 提供者表明用户的最终唯一标识符，也是到 OpenID 提供者的认证请求的返回值。所以，这就是 JBCP Pets 应该存储并用来区分用户的标识符。

Spring Security 处理 OpenID 登录请求的流程如下图所示：

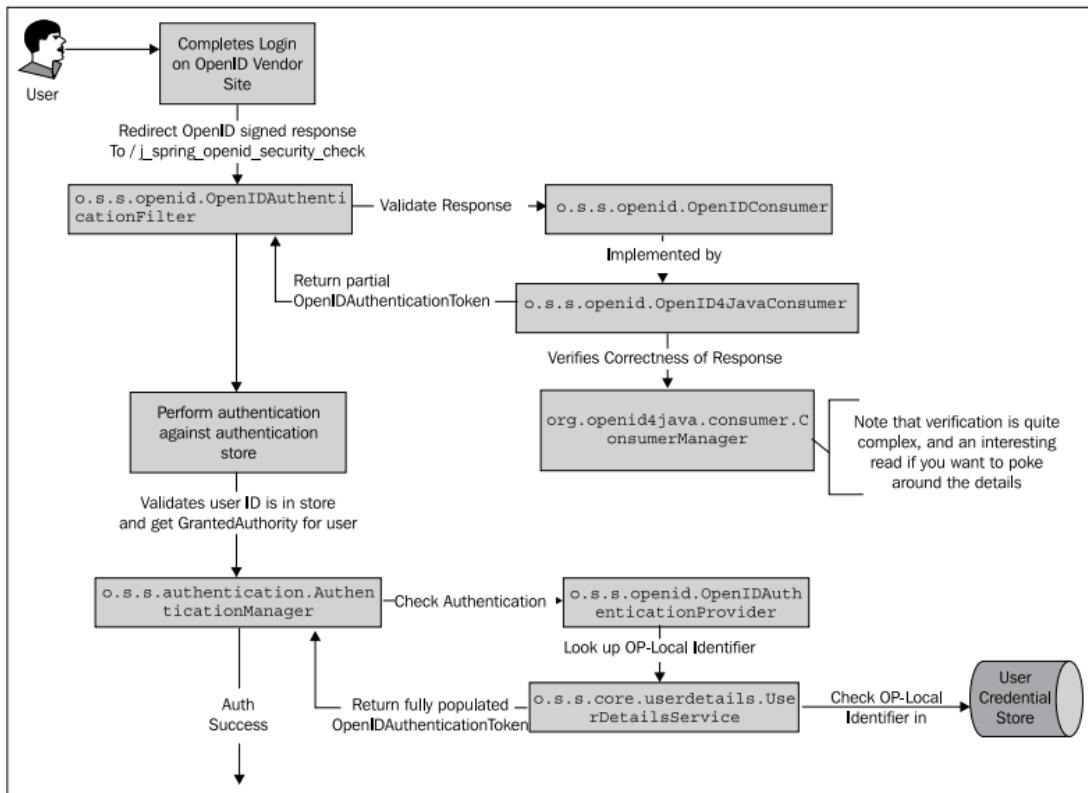


`o.s.s.openid.OpenIDAuthenticationFilter` 负责监听 `/j_spring openid_security_check` 这个 URL 并响应用户的登录请求，类似于 `UsernamePasswordAuthenticationFilter` 为 `/j_spring_security_check` URL 所作的那样。从这个图我们可以看到 `o.s.s.openid.OpenID4JavaConsumer` 委托 `openid4java` 库构建最终重定向用户到 OpenID 提供者的 URL。`openid4java` 库（通过 `org.openid4java.consumer.ConsumerManager`）还负责查找提供者，对此我们前面有所描述。

这个过滤器实际上负责 OpenID 认证的两个阶段——即格式化到 OpenID 提供者的重定向以及处理提供者的认证响应。OpenID 提供者的响应是一个简单的 GET 请求，带有一系列定义良好的域，它们会被 `openid4java` 库处理并认证。你并会直接处理这些域，其中一些重要的如下：

域名	描述
<code>openid.op_endpoint</code>	OpenID 提供者用来校验的 URL
<code>openid.claimed_id</code>	用户提供的 OpenID 声明标识符
<code>openid.response_nonce</code>	提供者计算出的当前时间，用来创建签名
<code>openid.sig</code>	OpenID 的响应签名
<code>openid.association</code>	根据请求者生成的一次性使用的相关数据，被用来计算签名并确定响应的合法性
<code>openid.identifier</code>	OP-Local identifier

我们将会了解这些域怎样用于校验响应的合法性。让我们看一下处理提供者的 OpenID 响应的几个相关者：



我们可以看到用户在提交凭证到 OpenID 提供者站点后，被重定向到 `/j_spring openid security check`。`OpenIDAuthenticationFilter` 进行一些基本的检查以判断这个请求是 OpenID 请求（从 JBCP Pets 登录表达发起）还是一个提供者的合法 OpenID 响应。

一旦确定这个请求是 OpenID 响应，要进行一系列复杂的校验以保证响应的正确性和可靠性（参考随后的 Is OpenID secure? 章节以了解更多细节）。`OpenID4JavaConsumer` 最终会返回一个填充不完整的 `o.s.sopenid.OpenIDAuthenticationToken` 对象，而它会被过滤器用来确定对响应的初始校验是否成功。这个 token 会被传递到 `AuthenticationManager` 中，而 `AuthenticationManager` 像处理其它 Authentication 对象那样处理它。

`o.s.sopenid.OpenIDAuthenticationProvider` 最终负责跟本地存储进行的校验（如 `JdbcDaoImpl`）。要记住的重要一点是，在数据存储中期望得到的包含 OP-Local Identifier 的用户名，它与用户提供的标识符并不一定一致——这是进行 OpenID 注册的问题关键。从这以下的处理流程与传统的用户名/密码认证很相似，最重要的是要从 `UserDetailsService` 取到正确的组和角色分配。

使用 OpenID 实现注册

对于启用了 OpenID 功能的 JBCP Pets 站点来说，用户要创建一个账号就要首先证明他们拥有自己所声明的标识符。所以，我们允许用户提供一个 OpenID 注册。我们已经添加了为标准的用户名和密码认证添加了注册流程，你可以通过页头上的“Registration”进行尝试。让我们扩展这个注册流程以允许用户使用 OpenID 来注册。

添加 OpenID 注册选项

首先，我们需要添加一个简单的表单到注册页上以允许用户输入和校验他们的 OpenID 标识，这个表单与登录的表单很相似（实际上，完全一致）。在 `registration.jsp` 中添加以下代码：

```
<h1>Or, Register with OpenID</h1>
<p>
    Please use the form below to register your account with OpenID.
</p>
<form action="j_spring_openid_security_check" method="post">
    <label for="openid_identifier">Login</label>:
    <input id="openid_identifier" name="openid_identifier" size="50"
maxlength="100" type="text"/>
    
    <br />
    <input type="submit" value="Login"/>
</form>
```

这个表单实际上与登录页的表单完全一样。那我们怎样区分登录和注册请求呢？

区分登录和注册请求

我们选择了一种很简单的方法来区分登录和注册请求。如果用户进行了一个成功的 OpenID 认证尝试，但是在我们的数据库中还没有他的账号，那我们就假设这是一个注册请求并把它加到数据库中。当然还可以完善这种方式（如，在用户创建账号之前显示确认信息），但是对于我们的例子来说，这就足够了。

我们会在扩展标准的 `AuthenticationFailureHandler`，在 `com.packtpub.springsecurity.security.OpenIDAuthenticationFailureHandler` 类中，如下：

```
package com.packtpub.springsecurity.security;
// imports omitted
public class OpenIDAuthenticationFailureHandler extends
    SimpleUrlAuthenticationFailureHandler {
    @Override
    public void onAuthenticationFailure(HttpServletRequest request,
        HttpServletResponse response, AuthenticationException exception)
        throws IOException, ServletException {
        if(exception instanceof UsernameNotFoundException && exception.getAuthentication()
            instanceof OpenIDAuthenticationToken &&
            ((OpenIDAuthenticationToken)exception.getAuthentication()).getStatus().equals(OpenIDAuthenticationStatus.SUCCESS)) {
            DefaultRedirectStrategy redirectStrategy = new DefaultRedirectStrategy();
            request.getSession(true).setAttribute("USER_OPENID_CREDENTIAL",
                "OpenID");
        }
    }
}
```

```
((UsernameNotFoundException)exception).getExtraInformation());
    // redirect to create account page
    redirectStrategy.sendRedirect(request, response, "/registrationOpenid.do");
} else {
    super.onAuthenticationFailure(request, response, exception);
}
}
```

我们可以看到这个代码扩展了父类的默认行为，在满足以下条件时将重定向用户到 registrationOpenid.do:

- 用户遇到了 UsernameNotFoundException;
- 用户已经被 OpenID 提供者成功认证（这通过检查 OpenIDAuthenticationToken 的 OpenIDAuthenticationStatus 属性值）。

这个代码将 OpenID 提供者返回的 OP-Local Identifier 值设置到 session 中，所以在被重定向到 OpenID 注册 URL 时还能取到。

配置自定义的认证失败处理器

我们需要配置这个认证失败处理器，只需要简单调整 dogstore-security.xml 中的 <openid-login> 声明：

```
<openid-login authentication-failure-handler-ref="openIdAuthFailureHandler">
<!-- The corresponding bean can be declared in dogstore-base.xml:-->
<bean id="openIdAuthFailureHandler"
      class="com.packtpub.springsecurity.security.OpenIDAuthenticationFailureHandler">
    <property name="defaultFailureUrl" value="/login.do"/>
</bean>
```

defaultFailureUrl 是用户遇到真正的登录失败时（如提供了非法的凭证信息），用户被重定向到的地址。

添加 OpenID 注册功能到控制器上

处理基于 OpenID 的注册很容易，添加到 LoginLogoutController 上就可以，这上面已经有了标准的用户名和密码注册：

```
@RequestMapping(method=RequestMethod.GET,value="/registrationOpenid.do")
public String registrationOpenId(HttpServletRequest request) {
    String userId = (String) request.getSession().getAttribute("USER_OPENID_CREDENTIAL");
    if(userId != null) {
        userService.createUser(userId, "unused", null);
        setMessage(request, "Your account has been created. Please log in using your OpenID.");
        return "redirect:login.do";
    } else {
        setMessage(request, "Please register using your OpenID.");
    }
}
```

```
    return "redirect:registration.do";
}
}
```

接下来，我们要修改 `IUserService` 接口和 `UserServiceImpl` 实现来建立一个简单的 `createUser` 方法：

```
@Service
public class UserServiceImpl implements IUserService {
    @Autowired
    CustomJdbcDaoImpl jdbcDao;
    // existing code omitted
    @Override
    public void createUser(String username, String password, String
email) {
        jdbcDao.createUser(username, password, email);
    }
}
```

你会发现我们也修改了 `@Autowired` 注解以明确引用 `CustomJdbcDaoImpl`，我们需要在这个类中实现一个自定义的 `createUser`，如下：

```
@Transactional
public void createUser(String username, String password, String email)
{
    getJdbcTemplate().update("insert into users(username, password, enabled, salt) values
(?, ?, true, CAST(RAND()*1000000000 AS varchar))", username, password);
    getJdbcTemplate().update("insert into group_members(group_id, username) select id, ? from
groups where group_name='Users'", username);
}
```

你可能对 SQL 感到熟悉——这就是我们在第四章：凭证安全存储中原来初始化用户的。还记得我们创建 `DatabasePasswordSecurerBean` 再启动时为密码加 salt？有了 `createUser` 方法，我们可以移除启动 SQL 了并且使用 Java 来初始化用户——你觉得我们应该怎样写代码来完成它呢？你为什么不试一试呢——这是一个很有效的练习来测试你对这方法的知识。

如果我们不是使用带 salt 域的自定义 user 表，我们可以简单的将 `CustomJdbcDaoImpl` 改为继承自 `JdbcUserDetailsManager`（就像我们在第四章讨论的那样）并使用已经为我们实现的 `createUser` 方法：

```
public class CustomJdbcDaoImpl
    extends JdbcUserDetailsManager
    implements IChangePassword {
```

这会需要对 `UserServiceImpl` 做一些小的修改：

```
@Override
public void createUser(String username, String password, String email)
{
    GrantedAuthority roleUser = new GrantedAuthorityImpl("ROLE_USER");
    UserDetails user = new User(username, password, true, true, true,
Arrays.asList(roleUser));
    jdbcDao.createUser(user);
```

}

你可以看到有两种不同的方式注册用户，自定义和内置的。每种方法都能有效的处理 OpenID 注册问题。尽可以体验实例应用并选择你喜欢的方法。

一旦用户通过我们的 `IUserService` 功能建立，用户被重定向到首页并且可以登录了。如果你想增强用户体验，我们可以做一些代码修改，保持 `OpenIDAuthenticationToken` 到重定向并自动认证用户。

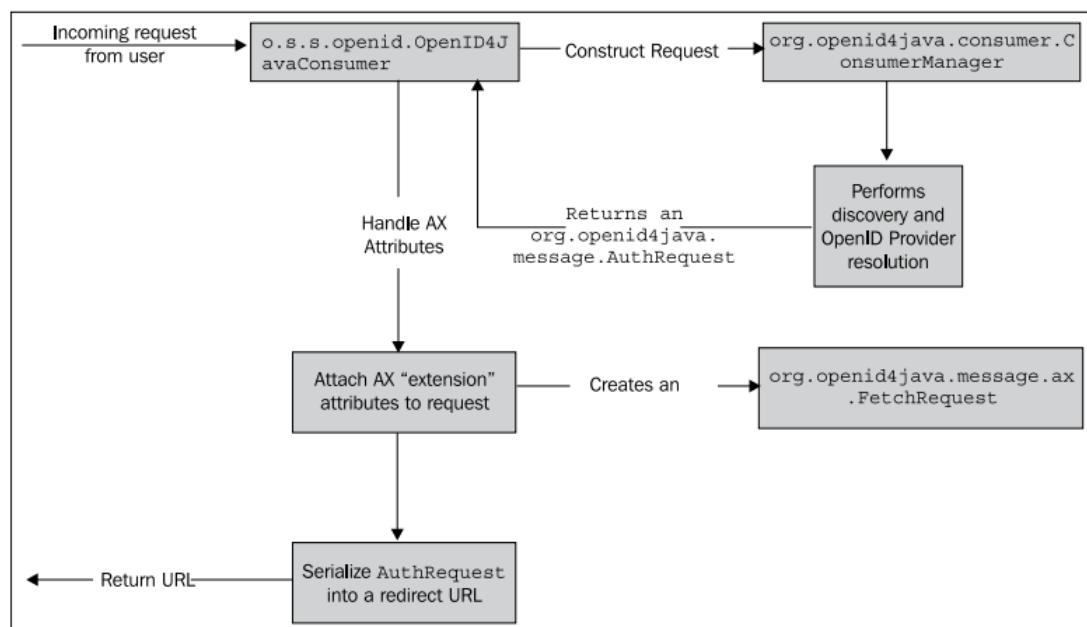
【注意，OP-Local identifiers 可能会很长——实际上，OpenID 2.0 规范并没有提供 OP-Local identifier 的一个最大长度。Spring Security 默认的 JDBC 数据库模式提供了一个相对很小的用户名列（你可能会记得我们将它从默认值扩展到了 100 个字符）。取决于你的需要，你可能愿意进一步扩展用户名列以容纳长的标识，或者实现 OpenID 处理链上的子类（如 `OpenIDAuthenticationProvider` 或 `UserDetailsService`）以正确处理太长的标识符。这可能包括将用户名拆为多列或存储被删节的 URL 和完整 URL 的哈希值。

要记住的是，认证不仅仅是基于 OpenID 标识数据库中的用户。有一些使用 OpenID 的站点比这更进一步，允许关联 OpenID 标识符和要进行认证的用户名（例如，允许多个 OpenID 关联相同的用户账号）。根据用户名提取出 OpenID 对于那些拥有不同提供者的多个 OpenID 且想在站点中使用的用户来说会有用——尽管这在一定程度上违背了 OpenID 的初衷，但它确实存在，你在设计使用 OpenID 站点的时候需要记住。】

除了凭证管理和中心认证，OpenID 的另一个承诺就是对用户来说在一个地方管理其个人信息并对特定的站点有选择的释放信息。这可能会提供能够丰富的注册体验。让我们看一下属性交换期望如何解决这个问题。

属性交换（Attribute Exchange）

OpenID 另外一个有趣的功能就是如果使用 OpenID 的站点需要，OpenID provider 提供（基于用户的许可）典型的用户注册数据，如名字、e-mail、生日。这个功能叫做属性交换（Attribute Exchange，AX）。下图展现了在 OpenID 请求中要求属性交换是如何添加进去的：



AX 属性值（如果 provider 提供的话）将会与其它的 OpenID 响应一起返回，并作为一个 o.s.s.openid.OpenIDAttribute 列表（list）插入到 OpenIDAuthenticationToken 中。

AX 属性能够被 OpenID 提供者随意定义，但是均为唯一定义的 URI。有人在努力标准化可用的和常见的属性模式。以下的属性是可用的（完整的列表在 <http://www.axschema.org/types/>）：

属性名	描述
http://axschema.org/contact/email	用户的 e-mail 地址
http://axschema.org/namePerson	用户的全名

axschema.org 站点列出了超过 30 个不同的属性，带有唯一的 URI 和描述。在一些特定情况下，你可能需要引用 schema.openid.net 而不是 axschema.org（稍后我们将会解释为什么）。

让我们看一下在 Spring Security 中如何配置属性交换。

在 Spring Security OpenID 中启用 AX

一旦你知道了要请求的属性，在 Spring Security OpenID 中启用 AX 实际上很容易。我们可以配置 AX 以请求 e-mail：

```
<openid-login authentication-failure-handler-ref="openidAuthFailureHandler">
    <attribute-exchange>
        <openid-attribute name="email" type="http://schema.openid.net/contact/email"
required="true"/>
    </attribute-exchange>
</openid-login>
```

对这个例子，我们建议你使用 myOpenID 标识登录。你将会看到，当你重定向到提供者时，提供者会通知你 JBCP Pets 站点请求额外的信息。在以下的截屏中，我们在请求中实际上包含了更多的 AX 属性：



请求的属性一旦被提供者返回，那在 OpenIDAuthenticationToken 就可用了（在成功认证

的请求中），是以键值对的方式存在而名字（即键）就是在<openid-attribute>中声明的。这就取决于我们的站点来检查这些数据并对其进行处理。一般来说这些数据能够要在填充用户简介或用户注册表单上。

为了进行探索研究，可以增强我们写的 OpenIDAuthenticationFailureHandler 来打印检索出的属性到控制台上：

```
request.getSession(true).setAttribute("USER_OPENID_CREDENTIAL",
((UsernameNotFoundException)exception).getExtraInformation());
OpenIDAuthenticationToken openIdAuth = (OpenIDAuthenticationToken)exception.getAuthentication();
for(OpenIDAttribute attr : openIdAuth.getAttributes()) {
    System.out.printf("AX Attribute: %s, Type: %s, Count: %d\n", attr.getName(), attr.getType(),
attr.getCount());
    for(String value : attr.getValues()) {
        System.out.printf(" Value: %s\n", value);
    }
}
redirectStrategy.sendRedirect(request, response, "/registrationOpenid.
do");
```

在我们的例子中将会得到如下的输出：

```
AX Attribute: email, Type: http://schema.openid.net/contact/email,
Count: 1
Value: peter@mularien.com
AX Attribute: birthDate, Type: http://schema.openid.net/birthDate,
Count: 1
Value: 1968-04-13
AX Attribute: namePerson, Type: http://schema.openid.net/namePerson,
Count: 1
Value: Peter Mularien
AX Attribute: nickname, Type: http://schema.openid.net/namePerson/
friendly, Count: 1
Value: pmularien
AX Attribute: country, Type: http://schema.openid.net/contact/country/
home, Count: 1
Value: US
```

我们可以看到 AX 数据能够很容易从 OpenID 提供者那里得到，并且支持通过简单 API 调用访问。在典型的应用场景下，如前面所讨论，AX 信息会用在注册时填入用户基本信息或偏好信息，节省了用户重复键入 OpenID 简介中已包含的信息。

现实世界的 AX 的支持和限制

但是，在现实中 AX 的承诺实现的很不足。市面上的 OpenID 提供者对 AX 支持的都很差，只有少数支持（myOpenID 和 Google 是最好的）。另外，即使支持标准的提供者在有什么属性对应数据方面也有混乱。比如，要查询用户的 e-mail，即使两个支持 AX 的提供者所请求

的属性名都不一样！

提供者	支持的 AX 属性
myOpenID	http://schema.openid.net/contact/email
Google	http://axschema.org/contact/email

在写作本书的时候，在 OpenID 相关的邮件列表中还在讨论怎样更好的标准属性并允许 OpenID 提供者公开他们所支持的属性。

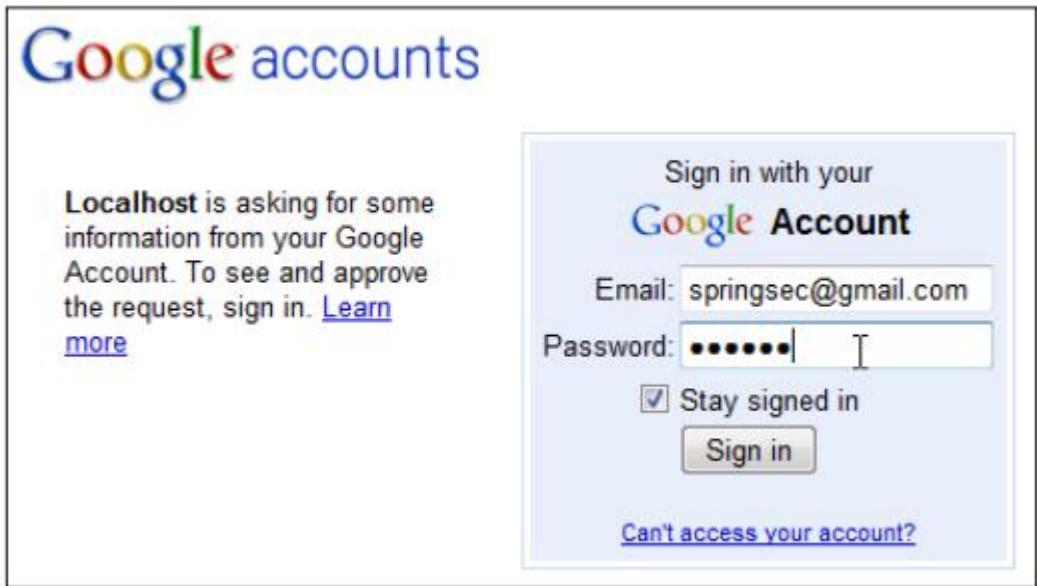
AX 的替代方式，叫做简单注册（Simple Registration，SReg）被 openid4java 所支持，但是并没有暴露到 Spring Security OpenID（由开发人员选择）。这很遗憾，因为 SReg 实际上很多的提供者支持，超过 AX。AX 曾经想作为更开放和灵活的替代 SReg 的方案，但是缺乏支持和标准化使得妨碍了它被提供者所采用。

支持 Google OpenID

Google 选择一种稍微不同的方式实现 OpenID，并没有给用户分配用户友好的 OpenID 标识符。相反，Google 期望提供 OpenID 的站点使用 Google 式的提供者 URL，并且用户直接向 Google 提供凭证。为了是这个处理对用户来说更简单，通用的做法是站点提供了“Sign in with Google”按钮，这会触发 Google OpenID 提供者。我们可以添加这个到登录页：

```
<form action="j_spring_openid_security_check" method="post">
    <input name="openid_identifier" size="50" maxlength="100"
type="hidden" value="https://www.google.com/accounts/o8/id"/>
    <input type="submit" value="Sign in with Google"/>
</form>
```

我们可以看到 Google URL 不需要暴露给用户。用户会看到这个经典的登录页：



在 Google 登录过程完成后，用户的 OP-Local Identifier 会被返回，并且注册/登录能像其它 OpenID 提供者一样处理。

OpenID 安全吗？

OpenID 依赖于 OpenID 提供者的可靠性以及提供者响应的可验证性，为了应用能够信任基于 OpenID 的用户登录，安全和可靠性至关重要。

幸运的是，OpenID 规范的设计者注意到了这些关切并实现了一系列的校验步骤来防止响应伪造、重放攻击以及其它类型的篡改，描述如下：

- 响应伪造（Response forgery）通过联合使用公用的密钥（使用 OpenID 的站点在初始请求之前创建）和对响应信息本身进行单向 hash 信息加密来阻止。恶意的用户不能访问共享的密钥和加密算法，伪装响应的任何域数据都会生成不合法的响应。
- 重放攻击（Replay attacks）通过包含当前时间（nonce）或一次使用的随机 key 来阻止，这应该保存在使用 OpenID 的站点上所以不能重复使用。这样，即使用户试图重新发送响应 URL 也会失败，因为接受的站点会确定当前时间（nonce）以前已经被使用过了，这个请求就失效了。

最可能的一种攻击方式可能会导致一个用户交互可能是中间攻击（man-in-the-middle attack）——在这里恶意用户拦截了用户电脑和 OpenID 提供者的交互。假设的攻击者在一个地方记录用户浏览器和 OpenID 的会话，并记录下请求初始化的密钥。在这种情况下，需要攻击者有很高的水平和相当完整的 OpenID 签名规则实现——简言之，在正常情况下这不会发生。

注意的是，尽管 openid4java 库支持使用 JDBC 持久化当前时间（nonce）跟踪，但是 Spring Security OpenID 当前没有作为一个配置属性暴露——所以当前时间只能在内存中跟踪。这意味着一个重复攻击可能在服务器重启后或集群环境下发生，在这里内存存储没有在不同服务器的 JVM 间复制。

小结

在本章中，我们了解了 OpenID——一个相对很新的技术用来进行用户认证的凭证管理。OpenID 在 web 中应用广泛，在最近一两年中在可用性和接受程度上发展迅速。现代 web 中大多数面向公众的站点都应该规划一些 OpenID 支持功能，JBCP Pets 也不例外。

在本章中，我们：

- 学习了 OpenID 认证机制并探究了它的整体结构和术语；
- 利用 JBCP Pets 站点，实现了 OpenID 的登录和自动用户注册功能；
- 通过使用属性交换（Attribute Exchange，AX），了解了 OpenID 未来的基本信息管理；
- 检查了 OpenID 登录响应的安全性。

在本章中，我们涉及到大多数常用基于 web 的外部认证方法。在下一章中，我们将会介绍一种最常用的基于目录的企业认证方式：LDAP。在我们要转到下一章的时候，请观察外部和内部认证机制的区别以及相似之处。

第九章 LDAP 目录服务

在本章中，我们将会了解轻量级目录访问协议（Lightweight Directory Access Protocol, LDAP）以及它怎样集成到使用 Spring Security 的应用中以提供认证、授权和用户信息服务。

在本章的内容中，我们将会：

- 学习一些 LDAP 协议相关的基本概念以及服务器实现；
- 在 Spring Security 中配置一个嵌入式的 LDAP 服务器；
- 使用 LDAP 认证和授权；
- 理解 LDAP 查找和用户匹配背后的模型；
- 从标准的 LDAP 结构中查询额外的用户信息；
- 不同的 LDAP 授权方法，并比较它们的优劣；
- 使用 Spring Bean 声明明确配置 Spring Security LDAP；
- 连接 LDAP 目录，包括 Microsoft Active Directory 进行认证。

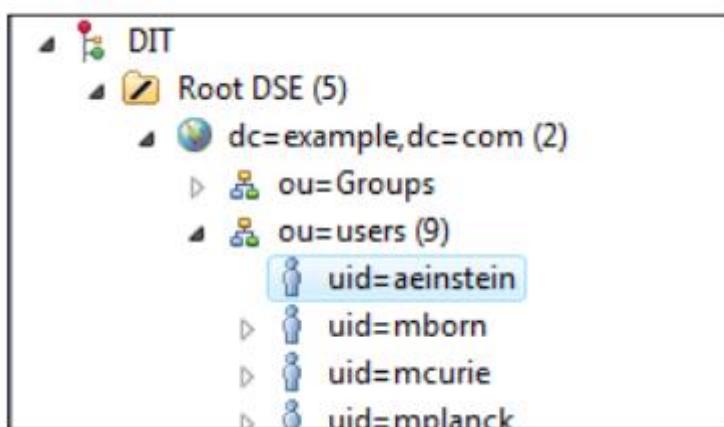
理解 LDAP

LDAP 在逻辑目录模型方面能够追溯到超过三十年前——在概念上类似于组织机构图和地址簿。今天，LDAP 越来越多的用来作为集中管理组织用户信息的方式，可以将成千的用户分成逻辑上的组并允许在不同的分布式系统间共享统一的用户信息。

为了安全目的，LDAP 经常被用来帮助集中的用户名和密码认证——用户的凭证存储在 LDAP 目录中，而对于用户来说，认证请求基于目录进行。这对于管理员来说可以便于管理，因为用户的凭证——登录、密码和其它信息——都存储在 LDAP 中的一个地址。另外，组织机构信息，如组和团队的分配、地理位置以及组织的等级结构，也定义在用户在目录中的位置上。

LDAP

如果你以前从来没有使用过 LDAP，你可能想知道它到底是什么。我们通过一个 Apache Directory Server 1.5 的截图作为 LDAP 模式的例子进行介绍。



让我们从一个特定用户 Albert Einstein（截图中高亮显示）的条目开始，我们可以看到

Mr. Einstein 的组织成员信息可以从他的树节点往上移动看到。我们可以看到 einstein 是组织单元 (organizational unit, ou) users 的成员，而组织单元本身是域 example.com 的一部分 (截屏中显示的 dc 代表的域组件 “domain component”)。在此之前的是 LDAP 树本身的组织机构元素 (DIT 和 Root DSE)，这些现在与我们无关。用户 aeinstein 在 LDAP 结构中有其语义且意义明确——你可以想象一个巨大组织更复杂的等级结构也能够很容易的说明其组织机构和部门的边界。

一个叶子节点在树上从上到下的路径形成了包含所有参与节点的字符串，如 Mr. Einstein 的节点路径为：

uid=aeinstein,ou=users,dc=example,dc=com

这个节点路径是唯一的，并被称为节点的标识名 (distinguished name, DN)。标识名类似于数据库里的主键，允许节点在复杂的树结构中唯一标识和定位。我们将会看到的节点的 DN 广泛应用于 Spring Security LDAP 集成的认证和查找过程。

我们可以看到还有几个其他的用户和 Mr. Einstein 在同一个等级的组织结构上。我们假设所有的这些用户都与 Mr. Einstein 在相同的组织下。尽管这个组织机构的例子相对简单，但是 LDAP 的结构是极其灵活的，使得很多层级的逻辑组织机构嵌套成为可能。

Spring Security LDAP 需要 Spring LDAP 模块的支持 (<http://www.springsource.org/ldap>)，它是单独于 Spring 框架核心和 Spring Security 的工程。它被认为很稳定并对标准的 Java LDAP 功能进行了有用的封装。

通用的 LDAP 属性名

树上的每个实际节点都是通过一个或多个的对象类 (object class) 来定义的。一个对象类是组织机构的一个逻辑单元，分为一系列具有语义的相关属性。通过将一个条目声明为特定对象类的实例，如 person，LDAP 目录的管理人员就能够为目录的用户提供每个条目的确切含义。

LDAP 具有很丰富的标准模式 (schema)，涵盖了可用的 LDAP 对象类和它们的可用属性 (以及一些其它信息)。如果你计划广泛的使用 LDAP，强烈建议你参考一个较好的用户手册，如 Zytrax OpenLDAP 的附录 (<http://www.zytrax.com/books/ldap/ape/>)，或者 Internet2 组织提供的人员相关模式 (<http://middleware.internet2.edu/eduperson/>)。

在上一节中，我们了解到 LDAP 中的每个条目都会有一个标识名，它在树上唯一标识节点。DN 有一系列的属性组成，其中一个 (或更多) 用来标识树上的用 DN 代表的路径。因为 DN 中路径的每一部分都代表一个 LDAP 属性，所以你能够通过定义良好的 LDAP 模式和类对象来确定 DN 中每个属性的含义。

我们在以下的表格中，列出了一些常见的属性和它们的含义。这些属性是用来作为组织相关的属性——意味着它们一般用来定义 LDAP 树的组织机构——并按照结构上从上到下的顺序，正如你通常在 LDAP 中会见到的那样。

属性名	描述	示例
dc	域组件 (Domain Component)——一般为 LDAP 等级结构中的最高一级组织	dc=jbcppets,dc=com
c	国家 (Country)——一些 LDAP 等级结构中将国家作为很高的等级	c=US
o	组织名 (Organization name)——一个 LDAP 资源分类上的业务组织	o=Sun Microsystems

ou	组织单元（Organizational unit）——业务组织部门，一般在组织之内	ou=Product Development
cn	通用名（Common name）——对象的通用名或唯一名或者为对人可读的名字。对人来说一般为人的全名，对于 LDAP 中的其它资源（电脑等等）一般为主机名。	cn=Super Visor cn=Jim Bob
uid	用户 ID（User ID）——尽管并不是原生作为组织相关使用，但是 Spring 一般会查找 uid 进行用户认证和搜索	uid=svisor
userPassword	用户密码（User password）——存储人对象相关联的密码。一般会经过 SHA 或类似的单向哈希算法。	userPassword=plaintext userPassword={SHA}cryptval

要记住的是有上百个标准的 LDAP 属性——上面只是其中的一小部分，当你与一个完整 LDAP 集成的话会看到它们。但是表中的这些属性是目录树中组织相关的属性，当你配置 Spring Security 与 LDAP 交互的时候可能会用来形成各种查询表达式或匹配符。

运行一个嵌入式的 LDAP 服务

作为测试，Spring Security 允许使用嵌入式的 LDAP 服务器。就像我们使用嵌入式数据库那样，这使得应用可以启动一个基于内存的 LDAP 服务器并插入初始化数据。当然，这样的一个配置只能用于测试的目的，但是这能够节省我们很多配置单独 LDAP 服务器的时间。

嵌入式的 LDAP 服务器功能是通过使用 Apache Directory Server (DS) 1.5 来支持的，它是一个基于 Java、开源且完全符合规范的 LDAP 服务器。实际上，你也可以使用 Apache DS 作为独立的服务器，它很相对很容易配置并易于获取和安装。本章实例代码中的 `Dependencies` 目录下包含了嵌入式 LDAP 服务器所需要的 JAR 包——如果你要自己使用它的话，你要么使用 Maven 要么自己到以下地址 <http://directory.apache.org/> 下载 Apache DS。

如同嵌入式的 HSQL 数据库允许在启动时加载 SQL 脚本，嵌入式的 LDAP 服务器提供了启动时从 LDAP 数据交换格式（LDAP Data Interchange Format，LDIF）文件中插入目录的方法。LDIF 是一种简洁的且对人和机器都很易读的数据定义格式，它提供了 LDAP 对象和支持数据的灵活定义。在本章的源码中提供了几个实例性的 LDIF 文件。

配置基本的 LDAP 集成

现在让我们让 JBCP Pets 支持基于 LDAP 的认证。幸运的是，通过使用嵌入式的 LDAP 服务器和实例 LDIF 文件，这是一个相对容易的练习。在这个练习中，我们使用为本书创建的 LDIF 文件，这个文件用来进行表述 LDAP 和 Spring Security 的常用配置场景。我们提供了几个其它的 LDIF 文件，其中一些来自 Apache DS 1.5，还有一个来自 SpringSecurity 的单元测试，你可能会愿意选择它们进行体验。

配置 LDAP 服务器引用

第一步是在 `dogstore-security.xml` 中声明嵌入式 LDAP 服务器的引用。LDAP 服务器的声明在`<http>`元素之外，与`<authentication-manager>`相同的等级：

```
<ldap-server ldif="classpath:JBCPPets.ldif" id="ldapLocal" root="dc=jb  
cppets,dc=com"/>
```

我们从 `classpath` 中加载 `JBCPPets.ldif`，并用其为 LDAP 服务器插入数据。这意味着（如同嵌入式 HSQL 数据库启动那样）我们应该在 `WEB-INF/classes` 放置 `JBCPPets.ldif` 文件。`root` 属性用特定的 DN 声明了 LDAP 目录的根。这应该与我们使用的 LDIF 文件逻辑根 DN 相对应。

【注意，对于嵌入式的 LDAP 服务器，`root` 是必须的，尽管 XML 模式并没有这样声明。如果它没有指明或指明错误，你会在 Apache DS server 启动的时候看待几个奇怪的错误。】

当我们在 `Spring Security` 配置文件中声明 LDAP 用户服务和其它配置元素时，会重用这里定义的 bean ID。对于嵌入式的 LDAP 模式来说，`<ldap-server>` 声明的其它属性都是可选的。

启用 LDAP AuthenticationProvider

接下来，我们要配置另一个 `AuthenticationProvider`，它用来用 LDAP 来检查用户凭证。简单得添加另一个 `AuthenticationProvider` 即可，如下：

```
<authentication-manager alias="authenticationManager">  
    <!-- Other authentication providers are here -->  
    <ldap-authentication-provider server-ref="ldapLocal"  
        user-search-filter="(uid={0})"  
        group-search-base="ou=Groups"  
    />  
</authentication-manager>
```

我们稍后将会介绍这些属性——现在，回到应用并运行，使用用户名 `ldapguest` 和密码 `password` 进行登录。你应该能够登录进去了！

解决嵌入式 LDAP 的问题

很可能你在使用嵌入式 LDAP 时，调试问题很困难。Apache DS 的出错信息并不友好，这在 `SpringSecurity` 嵌入模式下更严重。如果你不能让这个简单的例子正常运行，请仔细检查以下的地方：

- 确保 Apache DS 依赖的所有 JAR 都在 web 应用的 `classpath` 下。这会有很多——最好的方式就是包含所有的（实例代码就是这样做的）；
- 确保在你的配置文件中`<ldap-server>`设置了 `root` 属性，且它与启动时加载的 LDIF 文件中 `root` 的定义相匹配。如果你遇到了找不到引用的错误，很可能要么缺少 `root` 元素，要么与 LDIF 文件不匹配；
- 注意的是启动嵌入式 LDAP 的错误并不会是一个致命错误。为了分析加载 LDIF 文件的错误，你需要确保适当设置了日志，包括 Apache DS 的日志启用，至少要在 `ERROR` 级别。LDIF 的加载器在包下，它应该被用来启用 LDIF 加载错误的日志；

- 如果应用没有被正常关闭，为了重新启动服务，你可能会需要删除临时目录下的一些文件（Windows 系统下为%TEMP%）。这个的出错信息（幸运的是）很清楚。

遗憾的是，嵌入式 LDAP 并不像嵌入式 HSQL 数据库那样简单，但是相对于需要下载和配置的很多外部 LDAP 服务器来说，已经比较简单了。

一个用于排除问题和访问 LDAP 的好工具是 Apache Directory Studio，它提供了独立的和 Eclipse 插件的版本。免费下载地址：<http://directory.apache.org/studio/>。

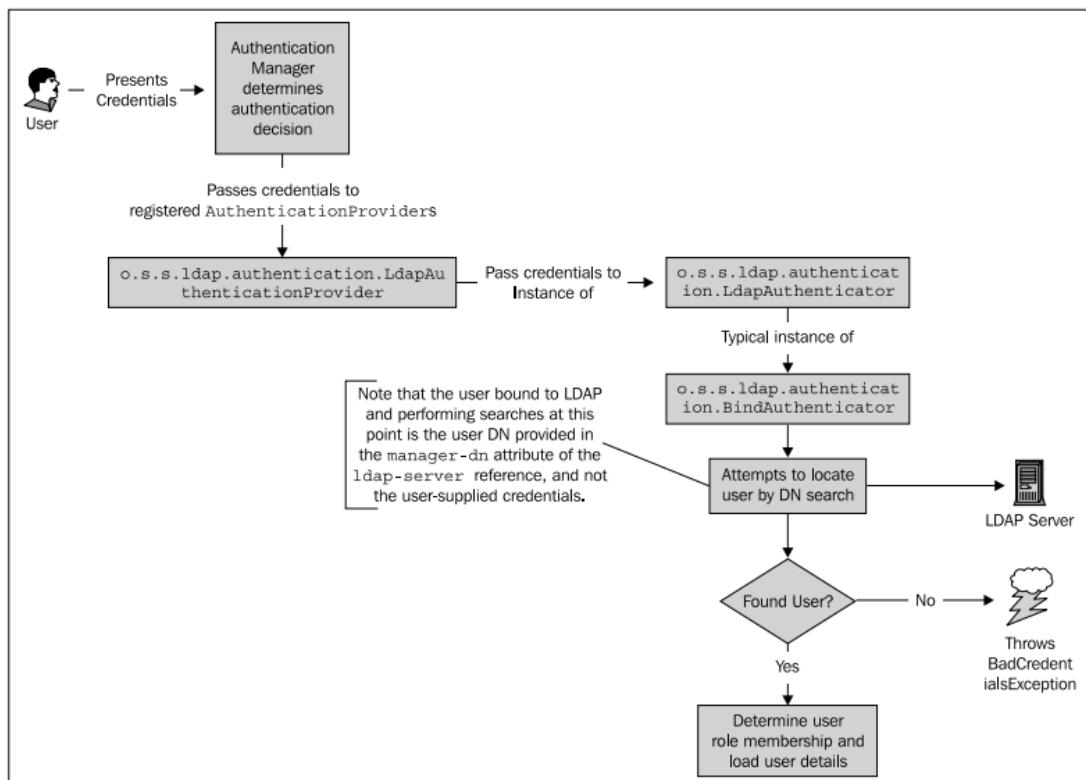
理解 Spring LDAP 认证如何工作

我们看到可以使用 LDIF 文件定义的用户（也就会在 LDAP 目录中出现）进行登录了。一个 LDAP 用户进行登录时到底发生了什么？在 LDAP 认证过程中有三个基本的步骤：

- 将用户提供的凭证与 LDAP 目录进行认证；
- 基于 LDAP 上的信息，确定用户拥有的 GrantedAuthority；
- 为了应用以后用到，从 LDAP 条目中预先加载用户信息到自定义的 UserDetails 对象中。

认证用户凭证

第一步，通过织入 AuthenticationManager 的自定义认证提供者与 LDAP 目录进行认证。`o.s.s.ldap.authentication.LdapAuthenticationProvider` 将用户提供的凭证与 LDAP 目录进行校验，如下图所示：



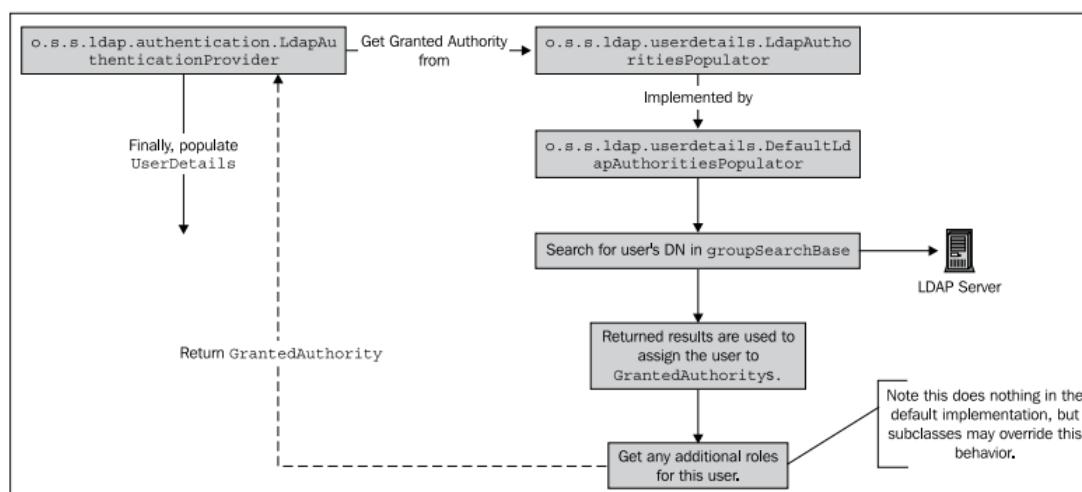
我们可以看到 `o.s.s.ldap.authentication.LdapAuthenticator` 接口定义了一个代理从而允许提供者以自定义的方式认证请求。在这里我们明确配置的是

`o.s.s.ldap.authentication.BindAuthenticator`，它会尝试使用用户的凭证绑定（登录）LDAP 服务器，就像用户本身尝试建立连接。对嵌入式的服务器来说，这对于我们的认证要求是足够的，但是，外部的 LDAP 服务器在用户绑定 LDAP 目录上可能要求更严格。幸运的是，还有一种替代的认证方式，我们将会在本章稍后介绍。

正如图中所标注的那样，记住查找是在`<ldap-server>`引用的 `manager-dn` 属性所创建的 LDAP 上下文中进行的。对于嵌入式的服务器，我们没有使用这个信息，但是对于外部的服务器引用，除非提供 `manager-dn`，否则的话将会进行匿名绑定。为了保持目录中公开访问信息的限制，通常需要合法的凭证来进行 LDAP 目录的搜索，这样的话，`manager-dn` 在现实世界场景中基本上就是必需的了。`manager-dn` 代表了用户的全 DN，基于合法的访问绑定目录并进行查找。

确定用户的角色

在用户基于 LDAP 服务器成功认证之后，接下来必须要进行权限信息的确定。授权是通过安全实体的一系列角色定义的，LDAP 认证过的用户角色确定如下图所示：



我们可以看到，用户在使用 LDAP 认证之后，`LdapAuthenticationProvider`委托给了一个`LdapAuthoritiesPopulator`。`DefaultLdapAuthoritiesPopulator`将会尝试在 LDAP 等级中另一个条目的同级或下级属性中查找认证用户的 DN。（译者注：即在 LDAP 目录角色相关的条目中寻找当前用户，以确定用户的角色）

查找用户角色分配的 DN 是通过 `group-search-base` 属性定义的——在我们的例子中，我们这样设置 `group-search-base="ou=Groups"`。当一个用户的 DN 在 `group-search-base` DN 下面的条目中时，包含用户 DN 的条目中的一个属性将会作为这些用户的角色。

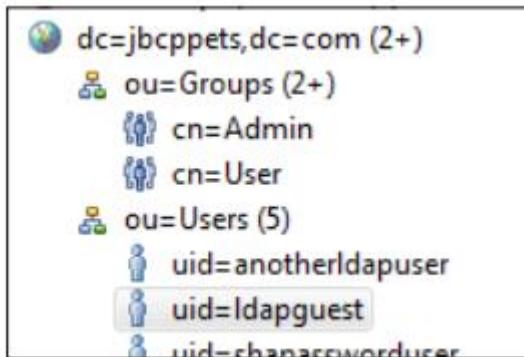
【你可能注意到我们混合使用了属性的写法——在类流程图中使用了 `groupSearchBase`，在文本中使用的是 `group-search-base`。这是有意的——文本中对应的是 XML 配置属性而图中指的是相关类的成员（属性）。他们的命名相似，但是在不同的上下文中（XML 和 Java）要适当调整。】

Spring Security 中的角色和 LDAP 中的用户如何关联还是有点令人迷惑，所以让我们看一下 JBCP Pets 库以及用户与角色关联是如何进行的。`DefaultLdapAuthoritiesPopulator` 使用了几个`<ldap-authentication-provider>`声明的属性来管理为用户查找角色。这些属性大致按以下的顺序使用：

- **group-search-base:** 它定义了基础的 DN，LDAP 集成应该基于此往下为用户查找一个或多个的匹配项。默认值会在 LDAP 根中进行查找，这可能会代价较高；
- **group-search-filter:** 它定义了 LDAP 查找的过滤器，用来匹配用户的 DN 与 group-search-base 之下的条目属性。这个过滤器通过两个参数进行参数化设置——第一个（{0}）作为用户的 DN，第二个作为（{1}）作为用户名。默认值为（uniqueMember={0}）。
- **group-role-attribute:** 它定义了匹配条目中用来组装用户 GrantedAuthority 的属性，默认值为 cn；
- **role-prefix:** 要拼接到在 group-role-attribute 中发现值的前缀以产生 Spring Security 的 GrantedAuthority。默认值为 ROLE_。

这对于新的开发人员可能会比较抽象和困难，因为这与我们基于 JDBC 的 UserDetailsService 实现有很大的区别。让我们以 JBCP Pets LDAP 目录中的 ldapguest 用户登录以了解其过程。

用户的 DN 是 uid=ldapguest,ou=Users,dc=jcpcppets,dc=com 而 group-search-base 被配置成了 ou=Groups。对于这个 ou 的 LDAP 树展现如下：



我们可以看到在 ou=Groups 之下，有两个条目（cn=Admin 和 cn=User）。每个条目都具有 objectClass: groupOfUniqueNames（你可能会记起我们在本章前面讨论过的对象类）。这种类型的 LDAP 对象允许多个 DN 值存储在这个条目下并进行逻辑分组。条目 cn=User 的属性列在下图中：

DN: cn=User,ou=Groups,dc=jcpcppets,dc=com	
Attribute Description	Value
objectClass	groupOfUniqueNames (structural)
objectClass	top (abstract)
cn	User
uniqueMember	uid=anotherldapuser,ou=Users,dc=jcpcppets,dc=com
uniqueMember	uid=ldapadmin,ou=Administrators,ou=Users,dc=jcpcppets,dc=com
uniqueMember	uid=ldapguest,ou=Users,dc=jcpcppets,dc=com

我们可以看到 cn=User 的 uniqueMember 属性用来标识这个组里面的 LDAP 用户。你也会发现 uniqueMember 的属性值就是对应用户的 DN。

现在再看角色搜索的逻辑就很容易了。从 ou=Groups (group-search-base)开始，Spring Security 将会查找任何 uniqueMember 属性值与用户 DN (group-search-filter) 匹配的条目。当它找到匹配的条目，条目的 cn 值 (group-role-attribute) —— 在本例中即为 User，将会加上 ROLE_ (role-prefix) 前缀然后转换成大写字母组成用户的 GrantedAuthority。一旦我们使用过它，再理解起来就容易一些了，不是吗？

【Spring LDAP 很灵活。要记住的是尽管这是一个组织 LDAP 兼容 Spring Security 的方式，

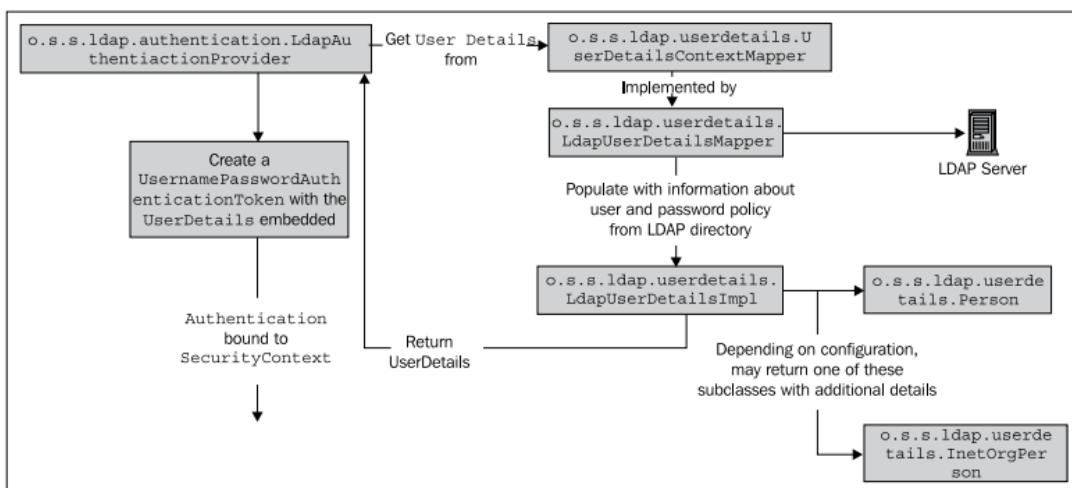
但是通常的使用场景恰恰相反——LDAP 目录已经存在，Spring Security 需要织入。在很多场景下，你可以重新配置 Spring Security 来处理 LDAP 的等级结构。但是，很关键的一点是你要有效规划并理解 Spring 在查询时如何与 LDAP 一起工作。开动你的大脑，勾画出用户查找和组查找以及你能想到的最优方案——让查询范围尽可能小和精确。】

如果你此时还是感到困惑，我们建议你休息一下然后尝试使用 Apache Directory Studio 来看一下运行系统配置的嵌入式 LDAP 服务器。如果你按照前面描述的算法，尝试自己查找一下目录将会有助于你了解 Spring Security LDAP 配置的流程。

匹配 UserDetails 的其它属性

最后，在通过 LDAP 查找分配给用户 GrantedAuthority 后，
`o.s.s.ldap.userdetails.LdapUserDetailsMapper` 将会使用
`o.s.s.ldap.userdetails.UserDetailsContextMapper` 来获取另外的细节信息来填充 UserDetails。

使用我们到现在为止配置的`<ldap-authentication-provider>`，`LdapUserDetailsMapper` 将会使用用户 LDAP 条目中的信息填充 UserDetails 对象。



我们稍后将会看到 `UserDetailsContextMapper` 怎样配置才能从标准的 LDAP person 和 `inetOrgPerson` 中获取丰富的信息。使用基本的 `LdapUserDetailsMapper`，仅仅能够存储用户名、密码以及 `GrantedAuthority`。

尽管在 LDAP 用户认证里面还有很多的结构，但是你会发现整体的流程与我们前面学习的 JDBC 认证很类似（认证用户、填充 `GrantedAuthority`s）。如同 JDBC 认证中那样，在 LDAP 集成中也有进行高级配置的能力——让我们了解的更深入一些并看看还能做什么。

LDAP 的高级配置

一旦我们要了解 LDAP 基础集成之外的知识，就会发现 security XML 命名空间方式的配置中，Spring Security LDAP 模块还有许多的可用配置。它包括查询用户的个人信息、用户认证的其它方式以及使用 LDAP 作为 `UserDetailsService` 且与 `DaoAuthenticationProvider` 结合。

实例 JBCP LDAP 用户

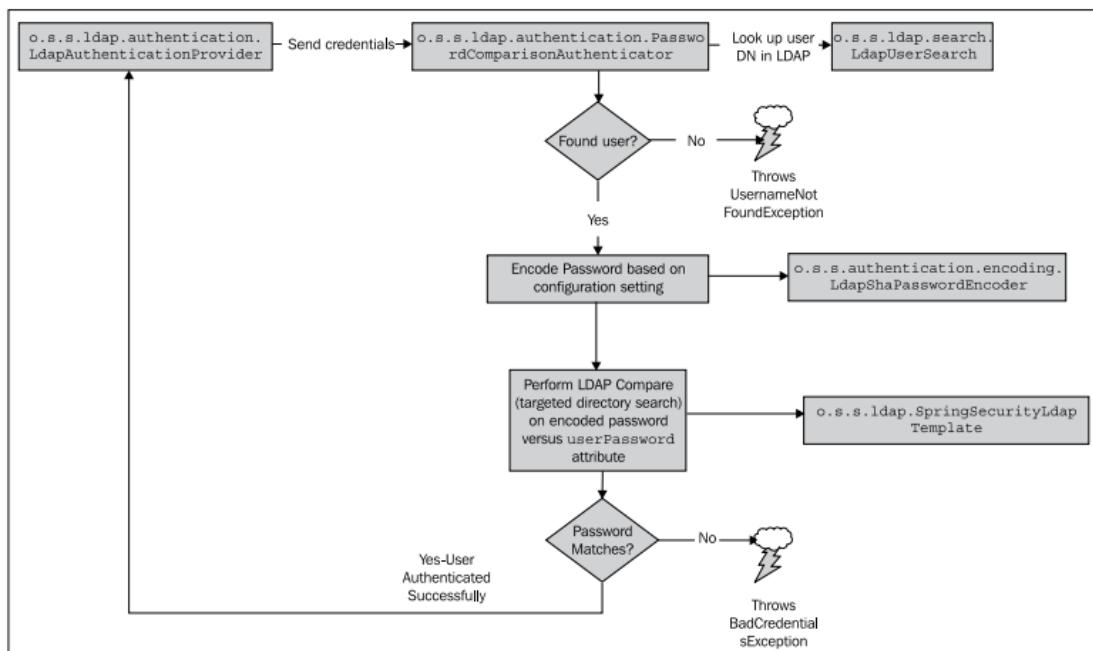
在 JBCP Pets LDIF 文件中，我们提供了许多的用户。在高级配置练习和自学中，以下的快速查询表可能会对你有所帮助。要注意的是除了 userwithphone 以外，所有用户的密码均为 password。

用户名	角色	密码编码
ldapguest	ROLE_USER	Plaintext
anotherldapuser	ROLE_USER	Plaintext
ldapadmin	ROLE_USER 和 ROLE_ADMIN	Plaintext
shapassworduser	ROLE_USER	{sha}
sshapassworduser	ROLE_USER	{ssha}
userwithphone	ROLE_USER	Plaintext（在 telephoneNumber 属性中）

我们将会在后面的章节中介绍为什么密码编码很重要。

密码对比与绑定认证

有一些 LDAP 服务器可能会配置成不允许特定的用户直接绑定到服务器上，这样的话匿名绑定（这是我们到此为止所使用的用户搜索办法）就被禁止了。这可能发生在很大规模的组织中，它想要限制能够从目录中读取信息的用户集合。在这种情况下，标准的 Spring Security LDAP 认证就行不通了，必须使用一种替代策略，通过 o.s.s.ldap.authentication.PasswordComparisonAuthenticator 实现(BindAuthenticator 的兄弟类)。



`PasswordComparisonAuthenticator` 绑定到 LDAP 上并查找匹配用户名所提供的 DN。它接下来会比较用户提供的密码和匹配的 LDAP 条目中存储的 `userPassword` 属性。如果编码

后的密码相匹配，用户认证成功，接下来的流程与 BindAuthenticator 相同。

配置基本的密码对比

配置密码对比认证来替换绑定认证很简单，只需在<ldap-authentication-provider>中添加一个子元素即可，如下：

```
<ldap-authentication-provider server-ref="ldapLocal">
    user-search-filter="(uid={0})" group-search-base="ou=Groups">
        <password-compare/>
    </ldap-authentication-provider>
```

默认的 PasswordComparisonAuthenticator 使用 LDAP 密码编码算法 SHA（回忆一下我们在第四章：凭证安全存储中讨论过的 SHA-1 密码加密算法）。在重启服务之后，你可以使用用户名 `shapassworduser` 和密码 `password` 尝试登录。

LDAP 密码编码和存储

LDAP 支持多种的密码加密算法，从简单文本到单向加密算法（类似于我们在第四章中了解到的基于数据库认证）。最常用的 LDAP 密码存储格式是 SHA（SHA-1 单向加密）和 SSHA（使用 salt 值的单向加密算法）。很多的 LDAP 实现支持 *RFC 2307, An Approach for Using LDAP as a Network Information Service* (<http://tools.ietf.org/html/rfc2307>) 定义的其它的密码格式。

RFC 2307 的设计者在密码存储方面做了一项很聪明的事情。从目录中的得到的密码当然是按照一定的算法（SHA 等）进行加密的，但是它以使用的加密算法作为前缀。这使得 LDAP 服务器能够很容易支持多种密码编码算法。例如，一个 SHA 编码的密码在目录中存储如下：

```
{SHA}5baa61e4c9b93f3f0682250b6cf8331b7ee68fd8
```

我们可以看到密码存储算法很清楚地进行了标明（使用{SHA}标识），并于密码一起存储。

SSHA 是尝试联合使用 SHA-1 哈希算法和密码 salting，以防止目录攻击。正如我们在第四章中所了解的那样，salt 在计算 hash 之前添加到密码中。当经过 hash 的密码存储到目录中后，salt 值也拼在 hash 后的密码上。密码以{SSHA}开头，这样 LDAP 服务器能够知道用户提供的密码需要以不同的方式进行对比。大多数的现代 LDAP 使用 SSHA 作为默认的密码存储算法。

密码对比认证的缺点

现在你了解了一些关于 LDAP 用户密码并建立了 PasswordComparisonAuthenticator，这是请你想一下如果使用 `shapassworduser` 用户以及 SSHA 格式存储的密码登录会发生什么？试一下——把书放在一边，尝试一下，然后回来。

你的登录被拒绝了，对不？可是你还能够使用 SHA 编码密码的用户登录——为什么？当我们使用绑定授权时，密码编码方式和存储并不会影响我们——你觉得这会是为什么呢？

绑定认证不会受到影响是因为 LDAP 服务器进行了认证和校验用户密码。在使用密码对比认证的时候，Spring Security LDAP 负责将密码编码成目录期望的格式然后与目录进行对比进行认证校验。

为了安全，密码对比策略并不能真正从目录上读取（基于安全策略，读取目录的密码通常会被拒绝）。作为替代，`PasswordComparisonAuthenticator` 会从用户的目录条目作为根节点进行一个 LDAP 查找，试图查找与 Spring Security 编码密码值匹配的密码属性值。所以，当我们以 `ssha` 登录时，`PasswordComparisonAuthenticator` 使用配置的 SHA 算法对密码进行编码并试图进行简单查找，这个查找失败了，因为目录中的用户密码是以 SSHA 的格式存储的。

当我们将 `PasswordComparisonAuthenticator` 的 hash 算法改成使用 SSHA 会发生什么呢，如下：

```
<password-compare hash="{ssha}">
```

重新启动并尝试——它还是不能好用。让我们想一下可能为什么。记住 SSHA 使用的是 salted 密码，而 salt 值是与密码一起存储 LDAP 目录上的。但是 `PasswordComparisonAuthenticator` 编码并不能从 LDAP 中获取任何信息（这在不允许的绑定的公司中是违背安全策略的）。所以当 `PasswordComparisonAuthenticator` 计算 hash 密码时，它不能确定使用什么 salt 值。

总之，`PasswordComparisonAuthenticator` 在特定的环境下很有用（要考虑目录本身的安全），但是它并不像直接绑定认证灵活。

配置 `UserDetailsContextMapper`

正如我们在前面讲到的，一个 `o.s.s.ldap.userdetails.UserDetailsContextMapper` 实例用来匹配用户条目和内存中的 `UserDetails` 对象。默认 `UserDetailsContextMapper` 的行为与 `JdbcDaoImpl` 类似，都是将一定数量的细节信息填充到要返回的 `UserDetails` 中——也就是说，除了用户名和密码以外并没有太多的信息。

但是，LDAP 目录可以包括除了用户名、密码和角色以外更多的用户细节信息。Spring Security 提供了方法从两个标准的 LDAP 对象模式——`person` 和 `inetOrgPerson` 获取更多的用户信息。

明确配置 `UserDetailsContextMapper`

为了配置一个不同与默认实现的 `UserDetailsContextMapper`，我们只需简单声明希望 `LdapAuthenticationProvider` 返回什么类型的 `LdapUserDetails` 类即可。security 命名空间解析器能够根据要求的 `LdapUserDetails` 类型，足够智能地实例化正确的 `UserDetailsContextMapper`。

让我们配置使用 `inetOrgPerson` 版本的匹配器。

```
<ldap-authentication-provider server-ref="ldapLocal"
    user-search-filter="(uid={0})" group-search-base="ou=Groups"
    user-details-class="inetOrgPerson">
```

如果你重启应用并尝试使用 LDAP 用户登录，你会发现没有任何变化。实际上，`UserDetailsContextMapper` 在幕后已经发生了变化，在本例中会根据 `inetOrgPerson` 模式从用户目录条目中读取额外的细节信息。

查看其它的用户细节

作为这节的辅助功能，我们将在 JBCP Pets 的账号信息页面增加“View LDAP User Profile”区域。我们会用这个页面来阐述更丰富的 person 和 inetOrgPerson LDAP 模式能提供更多（可选）的信息给使用 LDAP 的应用。

添加以下逻辑到 AccountController 中：

```
@RequestMapping(value="/account/viewLdapUserProfile.
do",method=RequestMethod.GET)
public void showViewLdapUserProfilePage(ModelMap model) {
    final Object principal = SecurityContextHolder.getContext().getAuthentication().getPrincipal();
    model.addAttribute("user", principal);
    if(principal instanceof LdapUserDetailsImpl) {
        model.addAttribute("isLdapUserDetails", Boolean.TRUE);
    }
    if(principal instanceof Person) {
        model.addAttribute("isLdapPerson", Boolean.TRUE);
    }
    if(principal instanceof InetOrgPerson) {
        model.addAttribute("isLdapInetOrgPerson", Boolean.TRUE);
    }
}
```

这个代码将会查询 LdapAuthenticationProvider 存储在 Authentication 对象中的 UserDetails (principal)，并确定是什么类型的 LdapUserDetailsImpl。页面的代码则要根据不同的 UserDetails 类型显示绑定到用户认证信息上的细节，JSP 代码如下。这个 JSP 应该放在 WebContent/WEB-INF/views/account/viewLdapUserProfile.jsp。

```
<!-- Common Header and Footer Omitted -->
<h1>View Profile</h1>
<p>
    Some information about you, from LDAP:
</p>
<ul>
    <li><strong>Username:</strong> ${user.username}</li>
    <li><strong>DN:</strong> ${user.dn}</li>
    <c:if test="${isLdapPerson}">
        <li><strong>Description:</strong> ${user.description}</li>
        <li><strong>Telephone:</strong> ${user.telephoneNumber}</li>
        <li><strong>Full Name(s):</strong>
            <c:forEach items="${user.cn}" var="cn">
                ${cn}<br />
            </c:forEach>
            </li>
    </c:if>
    <c:if test="${isLdapInetOrgPerson}">
```

```
<li><strong>Email:</strong> ${user.mail}</li>
<li><strong>Street:</strong> ${user.street}</li>
</c:if>
</ul>
```

我们也可以在 WebContent/WEB-INF/views/account/home.jsp 加一个链接：

```
<li><a href="viewLdapUserProfile.do">View LDAP User Profile</a></li>
```

我们已经增加了另外的两个用户即 `shapasswordperson` 和 `shapasswordinetorgperson`，你可以用它们来进行比较可用数据元素的不同。重启系统并查看“View LDAP User Profile”页面的三种不同类型的用户（非 `person`, `person` 和 `inetOrgPerson`）。你会发现，当 `user-details-class` 被配置成 `inetOrgPerson` 时，尽管要返回的是 `o.s.s.ldap.userdetails.InetOrgPerson`，但是其中的域可能被填充也可能没有填充，这取决于目录条目中可以得到的属性。

实际上，`inetOrgPerson` 拥有很多的属性远超过我们这个简单页面所提到的。你可以查看所有属性的列表：RFC 2798, Definition of the `inetOrgPerson` LDAP Object Class (<http://tools.ietf.org/html/rfc2798>)。

你可能会意识到，没有办法支持条目中指明的而标准模式中没有的属性。标准的 `UserDetailsContextMappers` 不支持任意列表的属性，但是可以通过使用 `user-context-mapper-ref` 属性指向自定义的 `UserDetailsContextMapper` 引用。

使用可代替的密码属性

在很多场景中，可能会需要使用其它的 LDAP 属性而不是 `userPassword` 进行认证。这可能在公司已经部署完自定义的 LDAP 模式时发生或不需要进行比较严格的密码管理（可以预见的是，这不是一个好主意，但在现实世界中它就存在）。

`PasswordComparisonAuthenticator` 也支持对用户的密码与一个其它的 LDAP 条目属性进行校验，而不是标准的 `userPassword` 属性。这很容易配置，我们可以提供一个很简单的例子，使用简单文本的 `telephoneNumber` 属性，如下：

```
<ldap-authentication-provider server-ref="ldapLocal"
    user-search-filter="(uid={0})" group-search-base="ou=Groups"
    user-details-class="inetOrgPerson">
    <password-compare hash="plaintext" password-attribute="telephoneNumber"/>
</ldap-authentication-provider>
```

我们可以重启服务并使用用户 `userwithphone` 和密码（即电话号码）1112223333 进行登录。

当然，这种方式的认证具有我们前面讨论过的 `PasswordComparisonAuthenticator` 基础认证的所有风险。但是，万一在 LDAP 中使用它的时候要注意。

使用 LDAP 作为 `UserDetailsService`

需要注意的一件事就是 LDAP 也可以用作 `UserDetailsService`。要记住的是 `UserDetailsService` 在 Spring Security 中要启用很多其它的功能，包括 `remember me` 和 OpenID 认证功能。

配置 LDAP 作为 `UserDetailsService` 到 LDAP AuthenticationProvider 中是很简单的。就像 JDBC `UserDetailsService`，一个 LDAP `UserDetailsService` 作为 `<http>` 的兄弟节点进行声明。

```
<ldap-user-service id="LdapUserService" server-ref="LdapLocal"
    user-search-filter="(uid={0})" group-search-base="ou=Groups"/>
```

在功能上，`o.s.s.ldap.userdetails.LdapUserDetailsService` 的配置与 `LdapAuthenticationProvider` 的配置基本完全相同，除了没有尝试使用安全实体的用户名绑定 LDAP。相反的，`<ldap-server>` 应用提供的凭证信息用来进行用户的查找。

【如果你想自己使用 LDAP 认证用户，请避免使用 `user-details-service-ref`（引用了一个 `LdapUserDetailsService`）配置`<authentication-provider>` 的常见错误。】

就像我们前面提到的那样，`LdapUserDetailsService` 使用`<ldap-server>` 提供的 `manager-dn` 来获取自己的信息——这意味着它不会尝试绑定用户到 LDAP 上，这可能与你期望的行为不一样。`LdapUserDetailsService` 一般用来支持系统的其它组件，如 OpenID 登录或 remember me 中，在这里认证已经通过的但是用户的详细信息在认证过程中并没有提供。

注意使用 LDAP UserDetailsService 时的 remember me

注意的是，如果此时重新启动服务器会失败，并有如下的错误信息：

```
More than one UserDetailsService registered. Please use a specific Id
reference in <remember-me/> <openid-login/> or <x509 /> elements.
```

如果我们回忆一下在 [第三章：增强用户体验](#) 和 [第四章：凭证安全存储](#) 中对于 remember me 功能的介绍，我们就会记得 remember me 依赖 `UserDetailsService` 来查找 remember me cookie 中的用户名。不幸的是，`AbstractRememberMeServices` 只可能查找一个 `UserDetailsService` 获取用户信息。所以，我们使用 remember me 功能时只能有一个配置的 `UserDetailsService` 而不能是两个。这使得使用相同的登录页同时支持 LDAP 认证和 JDBC，并为用户提供 remember me 功能变得很难实现。调整`<remember-me>` 配置引用某一个 `UserDetailsService`（通过 Spring Bean ID）足以明确告诉 Spring Security 你想做什么。

配置基于内存的 remember me 服务

在 LDAP 中另一个关于使用 remember me 要注意的是——为了给 LDAP 认证的用户提供 remember me 功能，你必须（通常会这样）使用基于 JDBC 持久化的 token remember me 服务。

你可能还记得在第三章中讨论的 remember me cookie 的组成，基于内存的 remember me 算法（在 `InMemoryTokenRepositoryImpl`）依赖于从 `UserDetails` 得到的用户名和密码。在很多场景下，LDAP 服务器配置成不允许读取 `userPassword` 属性（这就是为什么 `PasswordComparisonAuthenticator` 写成那个样子），所以 `LdapUserDetailsMapper` 很可能不能为 `UserDetails` 对象填充 `password` 属性。缺失这个关键的属性会导致创建 remember me cookie 失败，而且会潜在的影响 cookie 的安全。

避免这个问题也很简单——配置位于 JDBC 中的 remember me cookie 存储，它只依赖于用户名来校验 cookie（我们在第四章已经讨论过）。

集成外部的 LDAP 服务器

很可能你在测试完与嵌入式 LDAP 服务器集成后就要与外部的 LDAP 服务器交互了。幸运的是，这是简单，通过建立嵌入式 LDAP 服务器的<ldap-server>指令的简单语法就能完成。

以下的代码就是一个连接在 10389 端口上的外部 LDAP 服务器的示例配置：

```
<ldap-server url="ldap://localhost:10389/dc=jbcppets,dc=com"  
id="ldapLocal" manager-dn="uid=admin,ou=system" manager-password="secret"/>
```

这里的明显区别（除了 LDAP URL）就是提供了用户的 DN 和密码。这个账号（实际上是可选的）应该允许绑定到目录上并且能够进行所有相关用户和组信息的查询。使用这些凭证对 LDAP 服务器 URL 的绑定结果就是原来进行安全系统其它的 LDAP 操作。

注意很多的 LDAP 服务器支持 SSL 加密的 LDAP（LDAPS）——这无疑也是出于安全的考虑，Spring LDAP 也支持。只需在 LDAP 服务器 URL 上使用 ldaps://开始即可。LDAPS 通常运行在 636TCP 端口。

注意有很多的商业或非商业的 LDAP 实现。用于连接、用户绑定、填充 GrantedAuthorities 的具体配置完全取决于其提供者和目录的结构。在下一个章节中，我们会讲解一个很通用的 LDAP 实现，即 Microsoft Active Directory。

明确的 LDAP bean 配置

在本节中，我们带领你学会明确配置以下两项功能所需要的 bean 集合，即连接外部的 LDAP 服务器和支持授权的 LdapAuthenticationProvider。正如其它基于 bean 的配置，你可能不希望这样做，除非你发现 security 命名空间风格的配置不能支持你的业务或技术需求——在这种情况下，请继续阅读。

配置外部的 LDAP 服务器引用

为了实现这个配置，我们假设在本地的 10389 端口上运行着 LDAP 服务器，与前面章节的对应<ldap-server>有相同的配置。需要的 bean 定义在 dogstore-base.xml 中提供，如下：

```
<bean class="org.springframework.security.ldap.DefaultSpringSecurityContextSource"  
id="ldapServer">  
    <constructor-arg value="ldap://localhost:10389/dc=jbcppets,dc=com"/>  
    <property name="userDn" value="uid=admin,ou=system"/>  
    <property name="password" value="secret"/>  
</bean>
```

接下来，我们需要配置稍微复杂一些的 LdapAuthenticationProvider。

配置 LdapAuthenticationProvider

如果你已经读过并理解本章中关于 Spring Security LDAP 如何工作的讲解，那这个 bean 配置就很容易理解了，尽管稍有些复杂。我们将会配置一个包含如下特性的 LdapAuthenticationProvider：

- 用户的凭证进行绑定认证（不是密码对比）；
- 在 `UserDetailsContextMapper` 中使用 `InetOrgPerson`

让我们开始吧——首先我们声明 `LdapAuthenticationProvider`:

```
<bean class="org.springframework.security.ldap.authentication.LdapAuthenticationProvider"  
id="ldapAuthProvider">  
    <constructor-arg ref="ldapBindAuthenticator"/>  
    <constructor-arg ref="ldapAuthoritiesPopulator"/>  
    <property name="userDetailsContextMapper" ref="ldapUserDetailsContextMapper"/>  
</bean>
```

接下来是 `BindAuthenticator` 以及支持的 `FilterBasedLdapUserSearch` bean (用来在绑定前在 LDAP 目录中定位用户的 DN):

```
<bean class="org.springframework.security.ldap.authentication.BindAuthenticator"  
id="ldapBindAuthenticator">  
    <constructor-arg ref="ldapServer"/>  
    <property name="userSearch" ref="ldapSearchBean"/>  
</bean>  
  
<bean class="org.springframework.security.ldap.search.FilterBasedLdapUserSearch"  
id="ldapSearchBean">  
    <constructor-arg value="" /> <!-- user-search-base -->  
    <constructor-arg value="(uid={0})" /> <!-- user-search-filter -->  
    <constructor-arg ref="ldapServer"/>  
</bean>
```

最后是 `LdapAuthoritiesPopulator` 和 `UserDetailsContextMapper`，它们的角色我们在前文中已经介绍过：

```
<bean class="org.springframework.security.ldap.userdetails.DefaultLdapAuthoritiesPopulator"  
id="ldapAuthoritiesPopulator">  
    <constructor-arg ref="ldapServer"/>  
    <constructor-arg value="ou=Groups"/>  
    <property name="groupSearchFilter" value="(uniqueMember={0})"/>  
</bean>  
  
<bean class="org.springframework.security.ldap.userdetails.InetOrgPersonContextMapper"  
id="ldapUserDetailsContextMapper"/>
```

最后需要配置的元素是对我们 `LdapAuthenticationProvider` 的引用，我们将会在 `dogstore-security.xml` 中通过引用进行声明：

```
<authentication-manager alias="authenticationManager">  
    <authentication-provider ref="ldapAuthProvider"/>  
</authentication-manager>
```

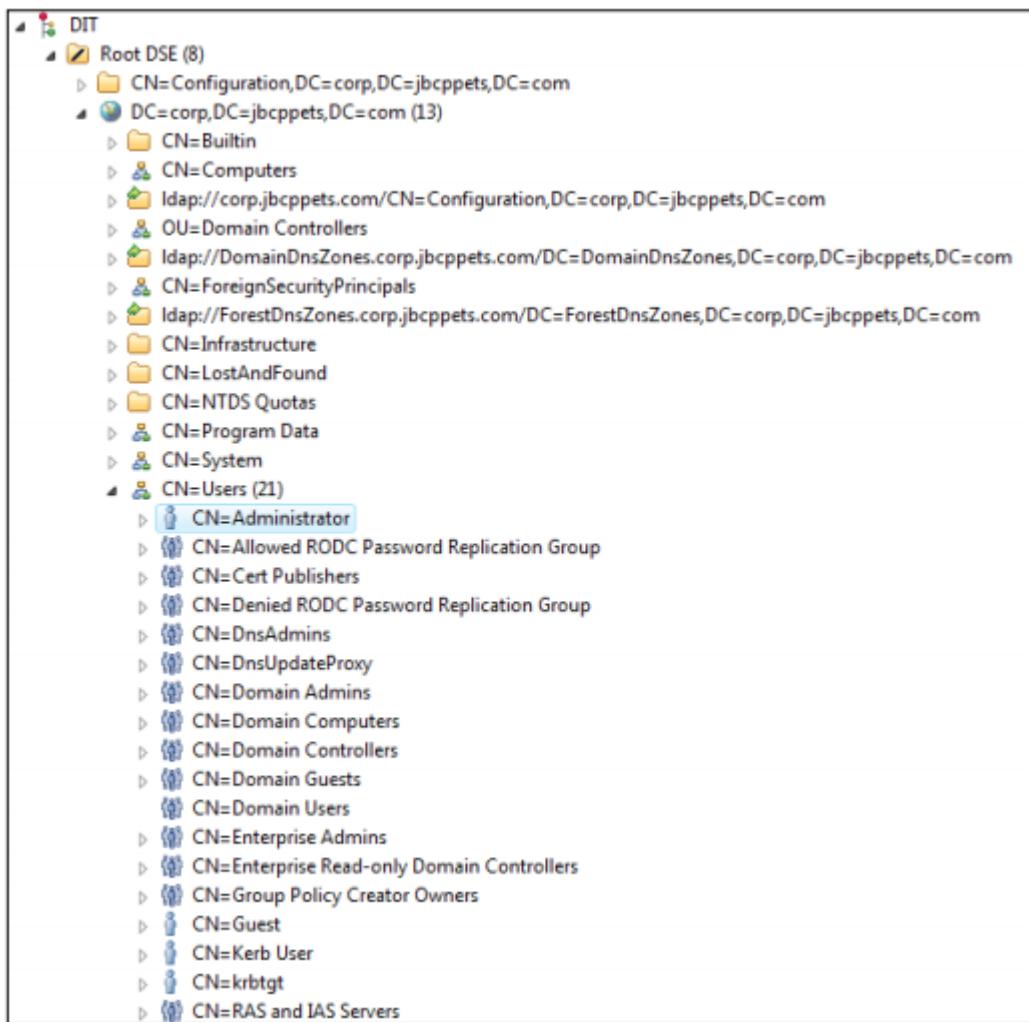
到此时为止，我们已经使用明确 Spring bean 的方法配置了 LDAP 认证。正如我们在第六章：高级配置和扩展第一次引入明确 bean 配置那样，在 LDAP 集成中使用这项技术能够在 security 命名空间配置没有暴露特定属性时，或者需要自定义的实现类以满足特定业务场景的地方很有用。接下来，我们将会介绍一个这样的场景即怎样通过 LDAP 连接到 Microsoft Active Directory。

通过 LDAP 集成 Microsoft Active Directory

Microsoft Active Directory 的一个便利功能是它不仅能够很好的与基于 Microsoft Windows 网络的结构集成，还能够配置成使用 LDAP 协议保罗 Active Directory 内容。如果你们是使用 Windows 的公司，很可能任何的 LDAP 集成都是基于你的 Active Directory 实例。

取决于你的 Microsoft Active Directory 配置（以及目录管理员支持 Spring Security LDAP 的意愿），你可能遇到的困难不在认证和绑定的过程中而是匹配 Active Directory 中的信息到 Spring Security 系统中的用户 GrantedAuthority 上。

JBCP Pets 公司的 Active Directory LDAP 树的实例在 LDAP 浏览器中的截图如下：



在这里你没有看到前面实例 LDAP 结构中的 `ou=Groups`——这是因为 Active Directory 在用户条目本身上以属性的方式存储分组信息。内置的 Spring Security（本书出版之时）并没有提供支持典型 Active Directory LDAP 树的 `LdapAuthoritiesPopulator`。

让我们使用刚刚了解到的明确配置 `bean` 的方式来织入一个简单的 `LdapAuthoritiesPopulator` 实现，它只是简单的为每个 LDAP 认证的用户授予 `ROLE_USER` 角色。

我们将这个类称为 `com.packtpub.springsecurity.security.SimpleRoleGrantingLdapAuthoritiesPopulator`

```
public class SimpleRoleGrantingLdapAuthoritiesPopulator implements  
    LdapAuthoritiesPopulator {
```

```
protected String role = "ROLE_USER";
public Collection<GrantedAuthority> getGrantedAuthorities(
    DirContextOperations userData, String username) {
    GrantedAuthority ga = new GrantedAuthorityImpl(role);
    return Arrays.asList(ga);
}
public String getRole() {
    return role;
}
public void setRole(String role) {
    this.role = role;
}
```

并不十分令人兴奋，但是要记住如果需要的话你可以使用传入的 `o.s.ldap.core.DirContextOperations` 对象进行很多类型的 LDAP 操作，如查询目录获取用户的信息以及使用这些信息填充返回的角色。

假设我们从上一节中的 bean 配置开始，让我们看一下怎样修改配置以支持 Active Directory 结构。

```
<bean class="org.springframework.security.ldap.DefaultSpringSecurityContextSource" id="ldapServer">
    <constructor-arg value="ldap://corp.jbcppets.com/dc=corp,dc=jbcppets,dc=com"/>
    <property name="userDn" value="CN=Administrator,CN=Users,DC=corp,DC=jbcppets,DC=com"/>
    <property name="password" value="admin123!"/>
</bean>
<bean class="org.springframework.security.ldap.search.FilterBasedLdapUserSearch"
    id="ldapSearchBean">
    <constructor-arg value="CN=Users"/>
    <constructor-arg value="(sAMAccountName={0})"/>
    <constructor-arg ref="ldapServer"/>
</bean>
<bean
    class="com.packtpub.springsecurity.security.SimpleRoleGrantingLdapAuthoritiesPopulator"
    id="ldapAuthoritiesPopulator"/>
```

属性 `sAMAccountName` 是在 Active Directory 中与标准 LDAP 条目的 `uid` 属性对等的——其它的配置修改都是典型的 Active Directory 配置。

尽管大多数的 Active Directory LDAP 集成都可能比本例复杂，但这会给你一个深入理解和探索 Spring Security LDAP 内部原理的起点，会使得支持更复杂的集成会简单一点。

注意存在另外一种与 Active Directory 集成的方式——Kerberos 认证，它是 Spring Security 扩展项目的一部分，并会在第十二章：Spring Security 扩展中涉及到。你可能会愿意了解两种类型的集成并确定哪种更适合你的需求（我们将会在第十二章中，比较 Active Directory 的 LDAP 和 Kerberos 认证）。

委托角色查询给 `UserDetailsService`

最后一项明确配置 bean 支持的技术就是在 `UserDetailsService` 中根据用户名查找用户并在此处获取 `GrantedAuthority`。配置很简单就像声明一个 bean，含有一个对 `UserDetailsService` 引用（就像基于 JDBC 的或我们在前面的章节中用到过的）：

```
<bean  
class="org.springframework.security.ldap.authentication.UserDetailsServiceLdapAuthoritiesPopul  
ator" id="ldapAuthoritiesPopulator">  
    <constructor-arg ref="jdbcUserServiceCustom"/>  
</bean>
```

你可能会预见到的逻辑和管理问题是用户名和角色必须同时在 LDAP 中和 `UserDetailsService` 所有的存储中管理——在大型用户场景下这可能不是一个灵活的模式。

这种场景的通常用在需要 LDAP 认证确保应用中的用户即为公司用户，但是应用本身想存储授权信息。这会导致潜在的应用相关数据在 LDAP 目录以外，这可能会是出于分开操作的考虑。

小结

我们已经看到在接受请求时，可以依赖 LDAP 服务器提供认证和授权信息，以及丰富的用户个人信息。在本章中，我们学到了：

- LDAP 术语和概念，并理解了 LDAP 目录怎样组织以支持 Spring Security；
- 在 Spring Security 配置文件中配置单独的（嵌入式的）和外部的 LDAP 服务器；
- 基于 LDAP 存储认证和授权用户，以及随后的匹配 Spring Security 角色；
- LDAP 中不同的认证模式、密码存储和安全机制——以及在 Spring Security 中怎样处理；
- 匹配 LDAP 目录中的用户详细信息到 `UserDetails` 对象中，实现 LDAP 和使用 Spring 应用的属性交换；
- LDAP 的明确 bean 配置，以及这种方式的优劣。

第十章 使用中心认证服务（CAS）进行单点登录

在本章中，我们将会介绍使用中心认证服务（Central Authentication Service，CAS）为基于 Spring Security 的应用提供单点登录门户（single sign-on portal）。

在本章的内容中，我们将会：

- 学习 CAS，包括它的架构以及对于系统管理员和各种规模组织的好处；
- 理解 Spring Security 怎样配置以处理认证请求拦截并重定向到 CAS；
- 配置 JBCP Pets 应用以使用 CAS 的单点登录；
- 集成 CAS 和 LDAP，并将数据通过 CAS 从 LDAP 传递到 Spring Security；
- 了解 CAS 2.0 和 SAML1.1 的区别。

CAS 简介

中心认证服务（Central Authentication Service, CAS）是一个开源的单点登录门户，提供了中心访问控制以及组织内基于 web 资源的认证。CAS 对管理员的好处是很大的，它支持很多的应用和不同的用户社区：

- 个人或组对资源（应用）的访问可以配置到一个地方；
- 广泛支持各种认证存储（为了集中管理用户），这提供了单点认证和对广泛分布、跨机器环境的控制；
- 通过 CAS 客户端类库支持基于 web 和非基于 web 的 Java 应用的广泛支持；
- 只有一个地方有用户凭证信息（通过 CAS），所以 CAS 的客户端应用不需要关于用户凭证信息的任何情况和如何验证它们。

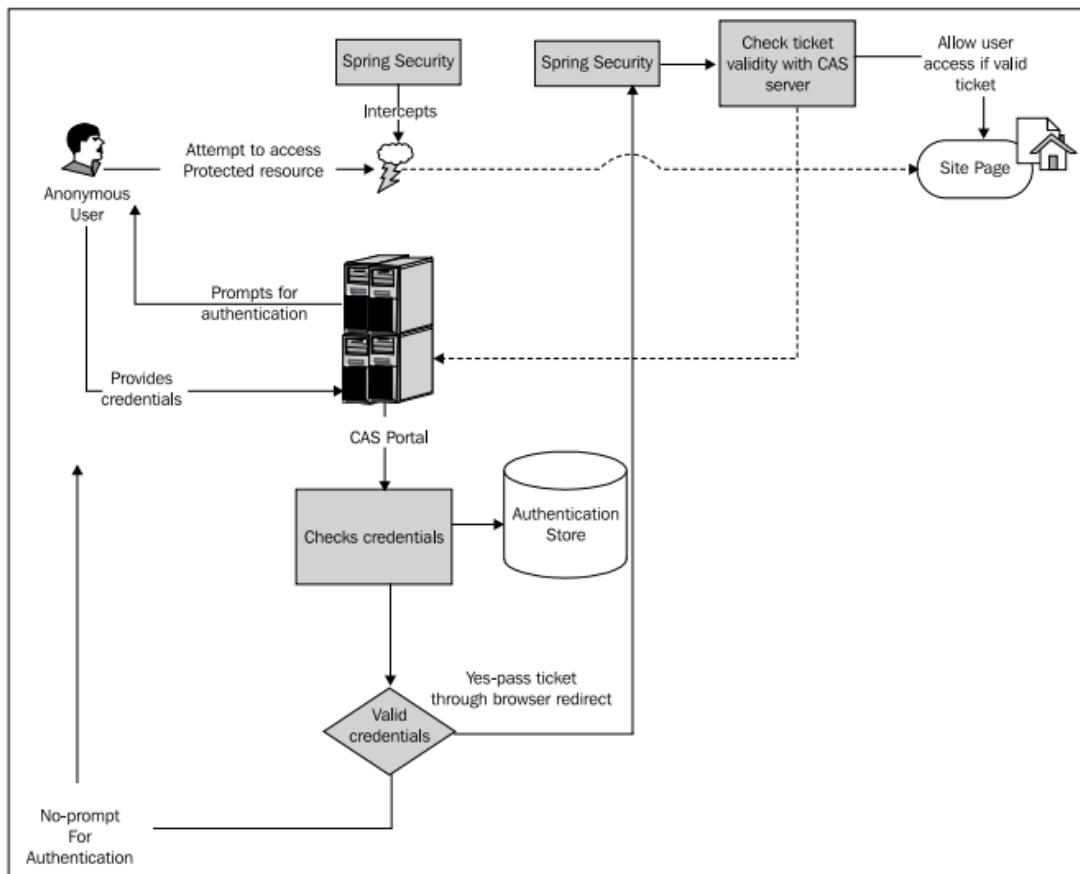
我们本章主要关注的不是如何管理 CAS，而重点是认证以及 CAS 怎样为我们的站点的用户提供认证。尽管 CAS 一般在企业或教育机构的 intranet 中见到，但是也能在一些面向公众的站点中见到如 Sony Online Entertainment's 和 Dun and Bradstreet's

CAS 认证整体流程

CAS 的基本认证流程包含以下的行为：

- 用户试图访问站点上的受保护资源；
- 站点的安全机制将用户重定向到 CAS 门户以要求用户提供凭证；
- CAS 门户负责用户认证。如果 CAS 认证成功，用户被重定向会受保护的资源并带有分配给该请求的一个唯一 CAS ticket；
- 站点的安全机制重新访问 CAS 服务器来校验这个 ticket 是否可接受（合法且没有过期等等）。CAS 服务器返回一个 assertion 来标识已经建立起了信任关系。
如果 ticket 被接受的话，信任已经建立，根据通常的授权检查用户可能已经可以进行下一步的操作。

直观起见，以上的过程在下面的图中进行了阐述：



从上面可以看到 CAS 服务器和受保护应用的整体交互，在用户信任之前要进行几次的数据交换握手。这种单点登录协议的复杂性使得很难通过常用的技术进行破解（考虑一下其它网络安全技术的预防措施，如使用 SSL 或网络监控）。

既然我们已经了解了 CAS 认证整体上如何工作，那看一下它怎样应用于 Spring Security。

Spring 与 CAS

Spring Security 拥有与 CAS 集成的强大功能，尽管没有像我们前面介绍 OpenID 和 LDAP 那样集成到 security 命名空间风格的配置中。作为替代，很多的配置依赖于 bean 织入和 security 命名空间中引用配置的 bean 声明。

基本的 CAS 认证涉及到以下两点：

- 替换标准的 AuthenticationEntryPoint——它一般处理未认证用户重定向到登录页面——为将用户定向到 CAS 门户的实现；
- 当用户从 CAS 门户重定向回受保护资源时，处理分配的 ticket，这通过使用一个自定义的过滤器实现。

关于 CAS 有一个很重要的事情要理解，在典型的部署环境中，CAS 原来替换应用中所有的登录机制。这样，一旦为 Spring Security 配置了 CAS，你的用户必须只能使用 CAS 作为你应用的认证机制。在大多数场景下，这不是什么问题，正如我们在前面讨论的，CAS 设计用来代理认证请求到一个或多个的认证存储（类似于 Spring Security 委托给数据库或 LDAP 认证那样）。我们在这个图中可以看到，我们的应用不再检查认证存储以校验用户（尽管还需要一个数据源用来完整填充认证用户的 UserDetails 对象）。

当我们完成 CAS 与 Spring Security 集成的基本配置，我们可以从首页中移除“Log In”链

接，当我们访问受限资源时会自动重定向到 CAS 的登录界面。当然，取决于应用，也可以允许用户明确进行登录（这样他们可以看到自定义的内容等等。）

CAS 的安装和配置

CAS 的好处是它背后有一个专业的团队不仅开发优秀的软件还提供精确完整的使用文档。如果你愿意练习本章的例子，我们建议你阅读 CAS 系统的“Get Started”手册。我们假设你在本章中将 CAS 部署到以下的位置：<http://localhost:8080/cas/>

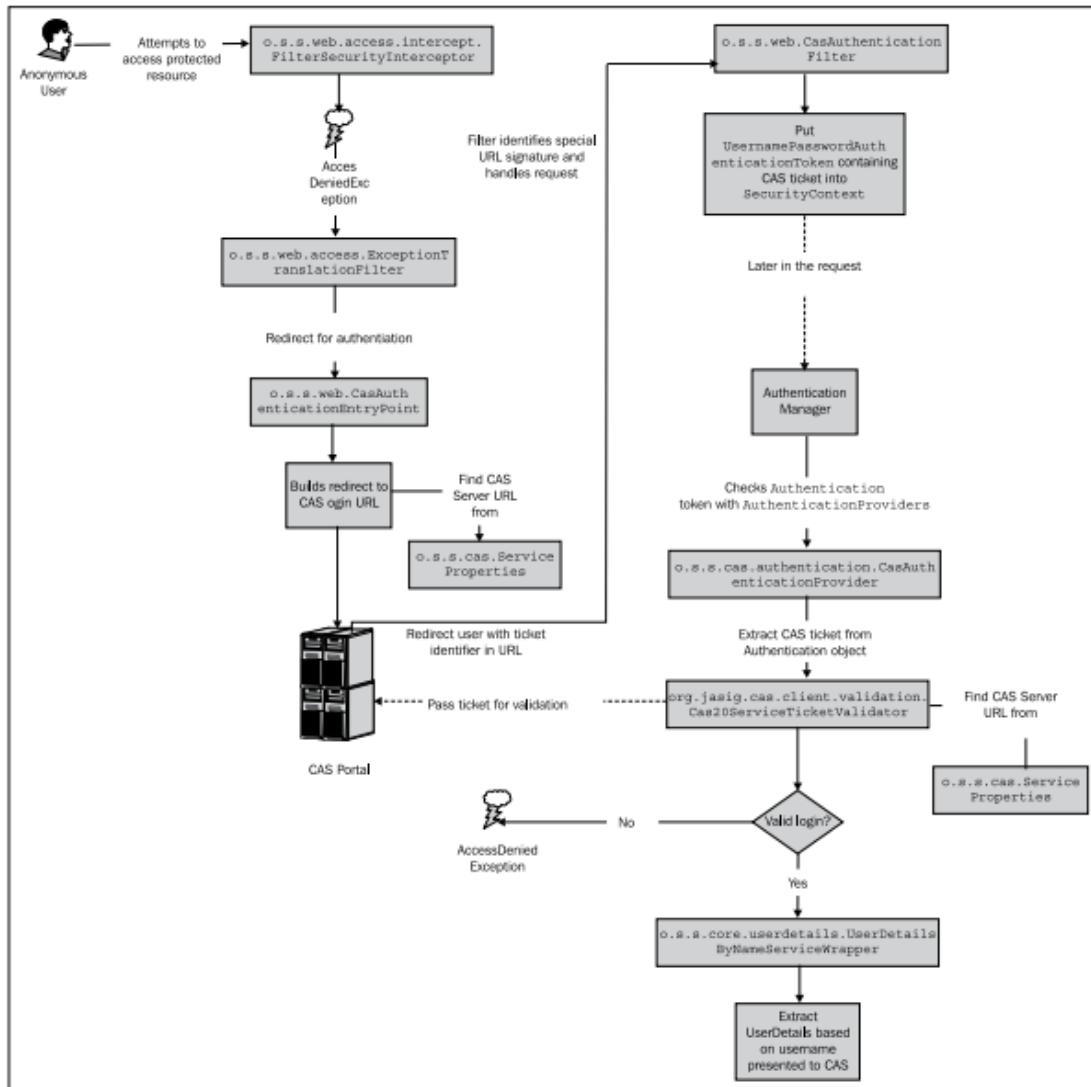
在本章中，我们建议将 CAS 和 JBCP Pets 运行在两个不同的 Tomcat 实例中。这会使得你在修改 JBCP Pets 代码时，保持 CAS 正常运行。我们建议你配置 CAS 服务器在 8080 端口，JBCP Pets 服务器在 8081 端口——我们在本章的例子是这样的配置。另外，本章中的例子用的是 CAS 服务器的最新版本，在写本书的时候为 3.3.5。

【要注意的是，在 CAS 的 3.x 版本中对于一些后台的类做了很大的修改，所以如果你使用更在版本的 CAS 服务器，这些指令在你的环境下会有或多或少的不同。】

让我们继续并配置 CAS 认证所需要的组件。

基本的 CAS 集成配置

以下的图表说明了要将 CAS 认证集成到 JBCP Pets 应用中所要配置的 Spring Security 组件之间的关系。这应该会帮助你在后面配置 bean 时，对此有整体的认识。



添加 CasAuthenticationEntryPoint

你可能会记得在第六章：高级配置与扩展中，`AuthenticationEntryPoint` 的实现类用来将授权失败相关的异常转换成正确的行为——例如将未认证的用户重定向到登录页或者给用户展现出缺乏相应权限的出错页。从整体结构图中，我们可以看到有一个对用户访问受保护资源的拦截器，它将用户重定向到 CAS 门户页面——这是通过使用专门用来处理这个问题的 `o.s.s.cas.web.CasAuthenticationEntryPoint` 来完成的。

我们在 `dogstore-base.xml` 中配置这个支持 bean 的行为，如下：

```

<bean id="casAuthEntryPoint"
      class="org.springframework.security.cas.web.CasAuthenticationEntryPoint">
    <property name="loginUrl" value="http://localhost:8080/cas/"/>
    <property name="serviceProperties" ref="casService"/>
</bean>
  
```

紧接着，在 `dogstore-security.xml` 文件的 `security` 命名空间配置中添加对这个 bean 的引用：

```
<http auto-config="true" ...  
entry-point-ref="casAuthEntryPoint">
```

最后，我们需要声明在 `CasAuthenticationEntryPoint` 中的 `serviceProperties` 引用，它是一个 `o.s.s.cas.ServiceProperties` 对象，声明了服务（应用）的属性（代表了我们的应用）。根据 CAS 认证过滤器的定义，我们声明的 URL 如下：

```
<bean id="casService" class="org.springframework.security.cas.ServiceProperties">  
<property name="service"  
value="http://localhost:8081/JBCPPets/j_spring_cas_security_check"/>  
</bean>
```

我们可以看到 `ServiceProperties` 对象在不同的 CAS 组件间交换数据中扮演了重要角色——它被用作数据对象，来存储 Spring CAS 不同参与者共享（同时相匹配）的 CAS 配置。

【CAS 将要认证访问的资源成为服务。服务以唯一的 URL 来进行标识。在 CAS 的管理界面上，服务能够被 CAS 管理员进行定义和配置。】

译者注：此处所谓的服务，指的就是各个使用 CAS 进行认证的应用。

其中 `service` 属性告知 CAS，要对那个服务进行用户认证。回忆一下，CAS 允许基于配置针对每个用户、每个应用进行有选择的授权。我们稍后配置处理该 URL 的过滤器时，还会介绍这个特殊 URL。

如果你此时重新启动应用并试图访问受保护的资源，你将会立即被重定向到 CAS 门户进行认证。在我们应用中的一个受保护页面的例子是“`My Account`”页面——它要求用户被授予了 `ROLE_USER` 权限。CAS 的默认设置允许认证任何用户名和密码相同的用户，你可以使用用户名 `admin` 和密码 `admin`（或 `guest/guest`）登录。

但是，你会发现，即使在登录后，你也会被立即再次重定向到 CAS 门户上。这是因为，尽管目标应用能够收到 `ticket`，但是它不能校验它，所以 CAS 会处理 `AccessDeniedException` 异常并认为拒绝该 `ticket`。

启用 CAS 票据校验

从前面“基本的 CAS 集成配置”一节的图中 我们可以看到，Spring Security 借助 `FilterSecurityInterceptor` 来负责识别未认证的请求并将用户重定向到 CAS。添加的 `CasAuthenticationEntryPoint` 重写了重定向到登录页的标准功能，并提供了从应用到 CAS 服务器的重定向。现在我们需要配置其它的东西，一旦经过了 CAS 的认证，用户应该在应用中也获得正确认证。

我们记得在第八章：对 OpenID 开放中讲到使用了类似的重定向方法，即将未认证的用户重定向到 OpenID 提供者上进行认证，并在校验凭证后重新返回到应用中。CAS 与 OpenID 的区别在于，基于用户请求返回到应用中时（译者注：即在 CAS 上登录成功后），应用要请求 CAS 服务器来明确校验提供的凭证是合法准确的。而 OpenID 使用的是基于时间的 `nonce` 和共享的基于 `key` 的签名，所以 OpenID 提供者传回的凭证信息能够被独立进行校验。（译者注：即 OpenID 和 CAS 在各自认证成功后都会通知应用，但 CAS 还需要应用与其进行交互以校验 ticket 的正确性，而使用 OpenID 的应用不需要在于 OpenID 提供者进行交互）。

CAS 方式的好处在于 CAS 服务器传递回来的认证用户信息很简单——只是简单的从 CAS 服务器返回给应用的一个 URL 参数。另外，应用本身并不需要跟踪和校验 `ticket`，相反可以完全依赖 CAS 校验这个信息。与我们在 OpenID 中看到的很类似，一个过滤器负责识别 CAS 的重定向并将其视为认证请求。我们先在 `dogstore-base.xml` 中，配置这个过滤器为一个 Spring bean，如下：

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

接下来，我们需要在 `dogstore-security.xml` 文件的 `security` 命名空间配置中，添加我们的自定义过滤器：

```
<http auto-config="true" ...>
    ...
    <custom-filter ref="casAuthenticationFilter" position="CAS_FILTER"/>
</http>
```

最后，我们需要声明一个 `CasAuthenticationFilter` 需要的 `AuthenticationManager`——这是通过在 `dogstore-security.xml` 中添加（如果还没有存在的话）`<authentication-manager>` 元素的 `alias` 属性做到的：

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

你可能已经发现在配置 `ServiceProperties` 对象时，我们将 CAS 服务名字定义为如下的 URL：http://localhost:8081/JBCPPets/j_spring_cas_security_check。正如我们在其它认证过滤器中所见过的那样，`/j_spring_cas_security_check` URL 表明的是 `CasAuthenticationFilter` 认识的 URL 签名，专门用于 CAS 服务器的重定向响应。

【修改 CAS 服务 URL。你可能希望修改默认的 CAS 服务 URL `/j_spring_cas_security_check`——这能通过设置 `CasAuthenticationFilter` 的 `filterProcessesUrl` 属性为任意的相对 URL。在有些时候修改默认的 URL 会有利于掩盖你的应用使用了 Spring Security 和/或 CAS——尽管这种安全掩盖的并不能保证万无一失，但是能够预防一些针对公众站点的攻击。】

`CasAuthenticationFilter` 用一些特定的凭证填充 `Authentication`（一个 `UsernamePasswordAuthenticationToken`）供下一个也即 CAS 最小化配置的最后一个元素识别。

使用 `CasAuthenticationProvider` 证明可靠性

如果你跟随本书了解 Spring Security 的逻辑流程，希望你会知道接下来是什么——`Authentication token` 必须要用合适的 `AuthenticationProvider` 进行检验。CAS 也不例外，所以最后的问题在于配置 `o.s.s.cas.authentication.CasAuthenticationProvider` 到 `AuthenticationManager` 中。

首先，我们需要在 `dogstore-base.xml` 中声明一个 Spring bean，如下：

```
<bean id="casAuthenticationProvider"
    class="org.springframework.security.cas.authentication.CasAuthenticationProvider">
    <property name="ticketValidator" ref="casTicketValidator"/>
    <property name="serviceProperties" ref="casService"/>
    <property name="key" value="jbcpp-pets-dogstore-cas"/>
    <property name="authenticationUserDetailsService" ref="authenticationUserDetailsService"/>
</bean>
```

接下来，我们要在 `dogstore-security.xml` 中配置对新 `AuthenticationProvider` 的引用，位于 `<authentication-manager>` 声明中：

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

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

如果你在还有前面练习的其它 `AuthenticationProvider`，在于 CAS 共同使用时，记住将它们移走。现在，我们需要处理 `CasAuthenticationProvider` 中的其它属性和 bean 引用。

属性 `ticketValidator` 引用了接口 `org.jasig.cas.client.validation.TicketValidator` 的实现——因为我们使用的 CAS 2.0 认证，我们要声明一个 `org.jasig.cas.client.validation.Cas20ServiceTicketValidator` 的实例，如下：

```
<bean id="casTicketValidator" class="org.jasig.cas.client.validation.Cas20ServiceTicketValidator">
    <constructor-arg value="http://localhost:8080/cas/" />
</bean>
```

这个类的构造参数要（再一次）引用访问 CAS 服务器的 URL。在这里你会注意到，我们已经离开了 `org.springframework.security` 包而到了 `org.jasig` 包中，这是 CAS 客户端 JAR 文件的一部分。我们稍后会看到 `TicketValidator` 接口拥有支持其它 CAS 认证的实现（还是在 CAS 客户端 JAR 文件中），如 SAML 认证。

【外部 URL 和独立环境的设置。如果你使用 CAS 配置真正的应用，在这里你可能会想到在 Spring 配置文件中写 URL 不是一个好主意。一般来说，对 URL 的存储和统一引用会抽取到单独的 `properties` 文件中，通过 `Spring PropertyPlaceholderConfigurer` 的占位符使用。这可以通过外部的 `properties` 文件重新设置特定环境的配置而不用关心 Spring 配置文件，这通常被认为好的方式。】

接下来，看 `key` 属性——它用来校验 `UsernamePasswordAuthenticationToken` 的完整性，因为后者是可能随意定义的（译者注：即恶意用户仿造的）。

最后，`authenticationUserDetailsService` 属性引用了一个 `o.s.s.core.userdetails.AuthenticationUserDetailsService` 对象，而它是用来将 `Authentication` 提供的用户名信息转换成完整的 `UserDetails`。显然，这项技术只有在我们确认了 `Authentication` 的完整性后才会用到。我们配置这个对象引用了 `UserDetailsService` 的实现 `JdbcDaoImpl`。

```
<bean id="authenticationUserDetailsService"
    class="org.springframework.security.core.userdetails.UserDetailsByNameServiceWrapper">
    <property name="userDetailsService" ref="jdbcUserService"/>
</bean>
```

你可能会问，为什么不直接引用 `UserDetailsService`——这是因为还有更高级的配置选项，它允许用 CAS 服务器返回的细节信息填充 `UserDetails` 对象。

到此时，我们可以完成并通过 CAS 进行登录并重定向回 JBCP Pets 应用。干得好！

高级 CAS 配置

CAS 认证框架提供了高级的配置和与 CAS 服务的数据交换。在本节中，我们将会介绍 CAS 集成的高级配置。在我们认为重要的地方将会包含相关的 CAS 配置指令，但是要记住的是 CAS 配置是很复杂的并超出了本书的范围。

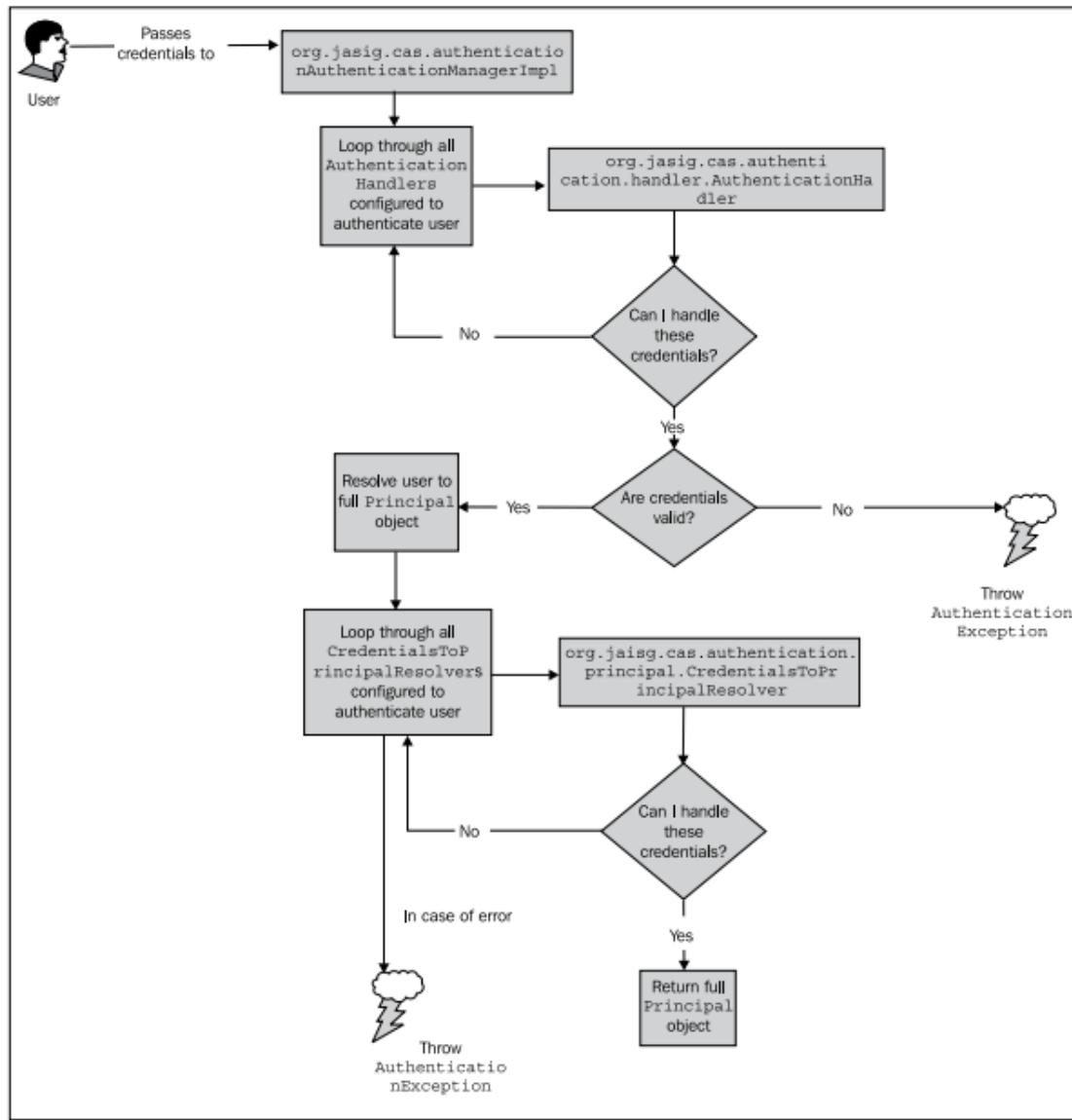
从 CAS assertion 中获取属性

在 CAS 服务器传递 ticket 校验结果时，可以将基于 CAS 认证时查询到的信息进行传递（给 CAS 服务）。这些信息以键值对的方式进行传递，并可以包含用户相关的任何数据。我们将

会使用这个功能在 CAS 响应中传递用户的属性，包括 GrantedAuthority 信息。

CAS 内部如何工作

在进入 CAS 配置之前，我们简单介绍 CAS 认证过程的标准行为。下图将会帮助你理解 CAS 与嵌入式 LDAP 服务器交互的配置步骤：



上图描述了 CAS 服务器内部的认证流程，如果你要实现 Spring Security 与 CAS 的集成，你可能要修改 CAS 服务器的配置。所以，理解 CAS 认证的整体流程如何工作是很重要的。

CAS 服务器的 `org.jasig.cas.authentication.AuthenticationManager`（不要与 SpringSecurity 的同名类混淆）负责基于提供的凭证信息进行用户认证。与 Spring Security 很相似，实际的认证委托给了一个（或更多）实现了 `org.jasig.cas.authentication.handler.AuthenticationHandler` 接口的处理类（在 Spring Security 中对应的接口是 `AuthenticationProvider`）。

最后，一个 `org.jasig.cas.authentication.principal.CredentialsToPrincipalResolver` 用来将传递进来的安全实体信息转换成完整的 `org.jasig.cas.authentication.principal.Principal`（类似于

Spring Security 中 UserDetailsService 实现所作的那样)。

尽管没有完整介绍 CAS 服务器的完整功能，但是这些也能够帮助你理解下面几个练习中的配置步骤。我们建议你阅读 CAS 的源码并学习网络上的文档，在 JA-SIG CAS wiki 上，地址为：<http://www.ja-sig.org/wiki/display/CAS>。

配置 CAS 连接嵌入式 LDAP 服务器

CAS 默认 认配置的
org.jasig.cas.authentication.principal.UsernamePasswordCredentialsToPrincipalResolver 并不允许我们在与 Spring Security CAS 集成时，传递回属性信息，所以我们建议使用一个允许我们这样做的实现。

一个很容易配置和使用的认证处理（尤其是你学习了前章中的 LDAP 练习）是 org.jasig.cas.authentication.handler.AuthenticationHandler，它会与我们前章中的嵌入式 LDAP 服务器通信。在后文中，我们将会介绍配置 CAS 返回 LDAP 属性。

所有 CAS 的配置都会在 CAS 安装后的 WEB-INF/deployerConfigContext.xml 文件中，将会涉及到插入类声明到已经存在的配置文件片段中。

【如果这个文件的内容你感到熟悉，那是因为 CAS 使用 Spring 框架进行配置，就像 JBCP Pets！如果你想深入了解这些配置是做什么的，我们建议你使用一个较好的 IDE 来方便的查看 CAS 的源码。记住在本节及其余所有章节中，我们提到的 WEB-INF/deployerConfigContext.xml，指的是 CAS 安装环境中而不是 JBCP Pets。】

首先，我们需要添加一个 AuthenticationHandler 来代替 SimpleTestUsernamePasswordAuthenticationHandler，它将会试图绑定用户到 LDAP 上（就像我们在第九章：LDAP 目录服务中做的那样）。AuthenticationHandler 要放在 authenticationManager bean 的 authenticationHandlers 属性中：

```
<property name="authenticationHandlers">
    <list>
        <!-- ... -->
        <bean class="org.jasig.cas.adaptors.ldap.BindLdapAuthenticationHandler">
            <property name="filter" value="uid=%u" />
            <property name="searchBase" value="ou=Users,dc=jbcppets,dc=com" />
            <property name="contextSource" ref="contextSource" />
        </bean>
```

不要忘记移除 SimpleTestUsernamePasswordAuthenticationHandler，或者至少将其定义移到 BindLdapAuthenticationHandler 后面，否则你的 CAS 认证不会使用 LDAP 而是会使用这个默认实现。

你可能会意识到这个 bean 引用了一个 contextSource bean——它定义了 org.springframework.ldap.core.ContextSource 实现，而 CAS 将会用它与 LDAP 交互（对，CAS 也使用了 Spring LDAP）。我们将会在文件的最后进行定义，如下：

```
<bean id="contextSource" class="org.springframework.ldap.core.support.LdapContextSource">
    <property name="urls">
        <list>
            <value>ldap://127.0.0.1:33389</value>
        </list>
```

```
</property>
<property name="userDn"
value="uid=ldapadmin,ou=Administrators,ou=Users,dc=jbcppets,dc=com"/>
<property name="password" value="password"/>
<property name="baseEnvironmentProperties">
<map>
<entry>
<key>
<value>java.naming.security.authentication</value>
</key>
<value>simple</value>
</entry>
</map>
</property>
</bean>
```

与我们配置 Spring Security 连接外部的（非嵌入式的）LDAP 服务器很类似，CAS 需要一个用户 DN 作为管理用户以首先绑定到 LDAP 目录上。在本例中，我们使用的管理员是在第九章的 JBCPPets.ldif 启动练习中定义的。URL `ldap://127.0.0.1:33389` 包含了端口 33389，是 Spring Security 嵌入式 LDAP 服务器默认使用的。正如我们在第九章讨论的，在产品配置中，你可能更会使用 LDAPS 以确保 CAS LDAP 请求的安全。

最后，我们需要配置一个新的 `org.jasig.cas.authentication.principal.CredentialsToPrincipalResolver`，它负责将用户提供的凭证（CAS 已经使用 `BindLdapAuthenticationHandler` 进行了认证）转换成完整的 `org.jasig.cas.authentication.principal.Principal` 认证实体。你会发现在这个类中有很多配置选项，我们会将其略过，但是欢迎你深入了解 CAS。

在 CAS `authenticationManager` bean 中的 `credentialsToPrincipalResolvers` 属性中，添加如下的内联 bean 定义：

```
<property name="credentialsToPrincipalResolvers">
<list>
<bean
class="org.jasig.cas.authentication.principal.CredentialsToLDAPAttributePrincipalResolver">
<property name="credentialsToPrincipalResolver">
<bean
class="org.jasig.cas.authentication.principal.UsernamePasswordCredentialsToPrincipalResolver"
/>
</property>
<property name="filter" value="(uid=%u)" />
<property name="principalAttributeName" value="uid" />
<property name="searchBase" value="ou=Users,dc=jbcppets,dc=com" />
<property name="contextSource" ref="contextSource" />
<property name="attributeRepository" ref bean="attributeRepository" />
</bean>
```

你会发现就像 Spring Security LDAP 配置，相同的行为也在 CAS 中存在即基于 DN 中的目

录子树根据属性匹配搜索安全实体。

注意，我们还没有配置 attributeRepository，它引用了 org.jasig.services.persondir.IPersonAttributeDao 的实现。CAS 提供了默认的配置，包含了这个接口的简单实现 org.jasig.services.persondir.support.StubPersonAttributeDao，这在 LDAP 属性的练习前是足够的（稍后会讲到）。

所以，现在我们已经在 CAS 中配置了基本的 LDAP 认证，你能够重启 CAS，启动 JBCP Pets（如果它没有在运行的话）并认证任何在第九章中使用的 LDAP 用户（用户名 ldapguest，密码 password 就不错）。但是在返回应用时，你会看到一个丑陋的 403 访问拒绝页面，这是因为我们的 AuthenticationUserDetailsService 在数据库中没有找到 LDAP 用户。现在让我们来解决这个问题！

从 CAS assertion 中获取 UserDetails

当首次建立 CAS 与 Spring Security 集成时，我们配置过 UserDetailsByNameServiceWrapper，它会将 CAS 提供的用户名转换成 UserDetails，这是通过使用我们引用的 UserDetailsService 对象（我们的例子中，就是 JdbcDaoImpl）。现在 CAS 引用了 LDAP 服务器，我们可以使用 LdapUserDetailsService 就像我们在第九章结束时讨论的那样，功能就会好用了。

但是在这里，我们体验 Spring Security CAS 集成的另外一种能力，即从 CAS assertion 中填充 UserDetails。这只需简单的将 AuthenticationUserDetailsService 实现切换至 o.s.s.cas.userdetails.GrantedAuthorityFromAssertionAttributesUserDetailsService 就可以了，它的工作就是读取 CAS assertion、寻找特定的属性并将属性值与用户的 GrantedAuthority 进行匹配。假设 assertion 返回了一个名为 role 的属性。我们只需要在 dogstore-base.xml 中简单配置一个新的 authenticationUserDetailsService bean：

```
<bean id="authenticationUserDetailsService"
      class="org.springframework.security.cas.userdetails.GrantedAuthorityFromAssertionAttributesUserDetailsService">
    <constructor-arg>
      <array>
        <value>role</value>
      </array>
    </constructor-arg>
</bean>
```

接下来，要添加一些小的调试功能来允许我们查询 assertion 中的内容。

检查 CAS assertion

为了辅助查询 CAS 返回给 JBCP Pets 应用的信息，我们修改 AccountController 以展现 CAS 登录用户的信息以及 CAS 为我们提供的用户信息。首先，我们添加一个简单的 URL 处理方法到 AccountController 中：

```
@RequestMapping(value="/account/viewCasUserProfile.do",method=RequestMethod.GET)
public void showViewCasUserProfilePage(ModelMap model) {
```

```
final Authentication auth = SecurityContextHolder.getContext().getAuthentication();
model.addAttribute("auth", auth);
if(auth instanceof CasAuthenticationToken) {
    model.addAttribute("isCasAuthentication", Boolean.TRUE);
}
}
```

接下来，我们要添加一个 JSP，它会展现 CasAuthenticationToken 的一些信息，这个对象代表 CAS 认证过的用户。它会放在 WEB-INF/views/account/viewCasUserProfile.jsp。

```
<!-- Common Header and Footer Omitted -->
<h1>View Profile</h1>
<p>
    Some information about you, from CAS:
</p>
<ul>
    <li><strong>Auth:</strong> ${auth}</li>
    <li><strong>Username:</strong> ${auth.principal}</li>
    <li><strong>Credentials:</strong> ${auth.credentials}</li>
    <c:if test="${isCasAuthentication}">
        <li><strong>Assertion:</strong> ${auth.assertion}</li>
        <li><strong>Assertion Attributes:</strong>
            <c:forEach items="${auth.assertion.attributes}" var="attr">
                ${attr.key}:${attr.value}<br />
            </c:forEach>
        </li>
        <li><strong>Assertion Attribute Principal:</strong> ${auth.assertion.principal}</li>
        <li><strong>Assertion Principal Attributes:</strong>
            <c:forEach items="${auth.assertion.principal.attributes}" var="attr">
                ${attr.key}:${attr.value}<br />
            </c:forEach>
        </li>
    </c:if>
</ul>
```

在账号首页添加一个简单的链接，在 WEB-INF/views/account/home.jsp 中：

```
<h1>Welcome to Your Account</h1>
<!-- omitted -->
<ul>
    <li><a href="viewCasUserProfile.do">View CAS User Profile</a></li>
```

最后，在 assertion 属性认证好用之前，我们需要（临时的）将这个页面进行授权检查。
(Finally, we'll have to (temporarily) disable authorization checks for this page, until we get assertion attribute-based authorization working.) 要做到这样只需要简单调整 dogstore-security.xml，所以任何具有 GrantedAuthority 的登录用户都能访问这个页面以及“My Account”页面：

```
<intercept-url pattern="/home.do" access="permitAll"/>
<intercept-url pattern="/account/home.do" access="!anonymous"/>
```

```
<intercept-url pattern="/account/view*Profile.do"
access="!anonymous"/>
<intercept-url pattern="/account/*.do" access="hasRole('ROLE_USER')"/>
```

在完成这些细小的 UI 更新后，重启 JBCP Pets 应用，并试图访问这个页面。你会发现页面提供了很多 assertion 包含的信息。

匹配 LDAP 属性到 CAS 属性

最后一个揭开的谜底是需要我们匹配 LDAP 属性到 CAS assertion 中（包括我们希望在 GrantedAuthority 中包含的 role 属性）。

我们将会在 CAS deployerConfigContext.xml 中添加一点其它的配置。这些新的配置将会指导 CAS 怎样从 CAS Principal 对象到 CAS IPersonAttributes 对象匹配属性，而后者将会最终序列化为 ticket 校验的一部分。这个 bean 的配置应该替代同名的 bean 即 attributeRepository。

```
<bean id="attributeRepository"
class="org.jasig.services.persondir.support.ldap.LdapPersonAttributeDao">
    <property name="contextSource" ref="contextSource" />
    <property name="requireAllQueryAttributes" value="true" />
    <property name="baseDN" value="ou=Users,dc=jbcppets,dc=com" />
    <property name="queryAttributeMapping">
        <map>
            <entry key="username" value="uid" />
        </map>
    </property>
    <property name="resultAttributeMapping">
        <map>
            <entry key="cn" value="FullName" />
            <entry key="sn" value="LastName" />
            <entry key="description" value="role" />
        </map>
    </property>
</bean>
```

这个功能可能会很令人迷惑——本质上，这个类的目的将 Principal 与后端的 LDAP 目录进行匹配（这就是 queryAttributeMapping 属性，将 Principal 的 username 域与 LDAP 查询的 uid 属性相匹配）。提供的 baseDN 属性要使用 LDAP 进行查询（uid=Idapguest），并且属性要从匹配的条目进行读取。匹配到 Principal 属性使用 resultAttributeMapping 属性中的键值对——我们将 LDAP 的 cn 和 sn 属性匹配到有意义的名字，而 description 属性匹配到 role 属性，而这个 role 属性就是 GrantedAuthorityFromAssertionAttributesUserDetailsService 要进行查找的。

造成这样的复杂性很大程度上是因为这个功能的一部分被另外个项目进行了封装即 Person Directory (<http://www.ja-sig.org/wiki/display/PD/Home>)，它的目的是从多个源收集关于某个人的信息并集成到单个视图上。Person Directory 的设计并没有直接关联到 CAS 服务器上，所以能够重用在其它应用中。这种设计选择的不足之处在于它会使得 CAS 集成的配置比预想的更复杂。

【一些 CAS 中已有的 LDAP 属性。我们可能愿意建立与第九章 Spring Security LDAP 相同类型的 LDAP 查询，即能够匹配 Principal 到一个完整的 LDAP 标识名，而后用这个 DN 来查询组信息（使用基本的 uniqueMember 属性到一个 groupOfUniqueNames 条目）。但是，CAS LDAP 代码并没有这种灵活性，这就使得更复杂的 LDAP 匹配需要扩展 CAS 中的基本类。】

还是感到迷惑？我们承认这对我们来说也有些迷惑，因为在 LDAP 数据和 CAS 数据之间的来回交互中有不同的方式。CAS 的邮件列表和社区 wiki (<http://www.ja-sig.org/wiki/display/CASUM/Home>) 是了解别人经验和与有知识的读者询问问题的好地方。

最后，返回 CAS assertion 中的属性

遗憾的是，CAS 2.0 协议并没有明确定义返回数据的标准格式。在 CAS 的 JIRA 缺陷跟踪系统中，有很多尝试实现这个功能，但是被拒绝了，因为 CAS2.0 是稳定的所以在最近并不会修改它。

也就是说，很多用户需要扩展 CAS 的响应来包含属性。最终，CAS 到客户应用的 ticket 校验响应是通过一个 JSP 渲染的，所以很容易修改你的 CAS 安装来包含一个响应，这个响应符合 Cas20ServiceTicketValidator 的属性解析。在你的 CAS 部署中，编辑 WEB-INF/view/jsp/protocol/2.0/casServiceValidationSuccess.jsp，并添加以下内容：

```
<cas:authenticationSuccess>

<cas:user>${fn:escapeXml(assertion.chainedAuthentications[fn:length(assertion.chainedAuthentications)-1].principal.id)}</cas:user>

<cas:attributes>
<c:forEach
var="attr"
items="${assertion.chainedAuthentications[fn:length(assertion.chainedAuthentications)-1].principal.attributes}"
varStatus="loopStatus" begin="0"

end="${fn:length(assertion.chainedAuthentications[fn:length(assertion.chainedAuthentications)-1].principal.attributes)-1}" step="1">
<cas:${fn:escapeXml(attr.key)}>${fn:escapeXml(attr.value)}</cas:${fn:escapeXml(attr.key)}>
</c:forEach>
</cas:attributes>
```

这会产生如下的 CAS 响应格式：

```
<cas:serviceResponse xmlns:cas="http://www.yale.edu/tp/cas">
<cas:authenticationSuccess>
<cas:user>ldapguest</cas:user>
<cas:attributes>
<cas:FullName>LDAP Guest</cas:FullName>
<cas:role>ROLE_USER</cas:role>
<cas:LastName>Guest</cas:LastName>
</cas:attributes>
```

```
</cas:authenticationSuccess>  
</cas:serviceResponse>
```

当这些修改完成并重启 CAS 和 JBCP Pets，你应该能够用 `ldapguest` 登录并访问使用 `ROLE_USER` 保护的区域。另外，你应该看一下“View CAS Profile”页面，它展现了 CAS assertion 返回的属性。

【注意——我们可以看到 `LdapPersonAttributeDao.resultAttributeMapping` 提供的属性名被直接用到 CAS 响应的 XML 中。这意味着它们不能是任意的，必须符合 XML 元素命名规则（例如，不能包含空格）。如果你想修改这种行为，你可能需要比这里更复杂的 JSP 代码。】

干的漂亮——在这两个复杂的产品间有很多的配置工作。休息一下喝杯咖啡吧！

一些 CAS 开发人员选择的另一种方式是扩展 `Cas20ServiceTicketValidator`（在 Spring Security 这一端）来对 CAS 返回的响应进行自定义的处理。

使用 SAML 1.1 进行替代的票据认证

SAML 是一个标准的、跨平台的协议，它通过结构化的 XML assertion 进行身份校验。SAML 被很多的产品所支持，包括 CAS（实际上，我们将会在稍后的章节中看到 Spring Security 里面对 SAML 的支持）。

SAML 的安全 assertion XML 语法解决了我们前面讲到的 CAS 响应协议传递属性的问题。令人高兴的是，切换 CAS ticket 校验和 SAML ticket 校验只需要修改 `dogstore-base.xml` 中的 `TicketValidator` 实现即可。添加一个 bean 如下：

```
<bean id="samlTicketValidator" class="org.jasig.cas.client.validation.Saml11TicketValidator">  
    <constructor-arg value="http://localhost:8080/cas/" />  
</bean>
```

然后，替换 `CasAuthenticationProvider` 使用这个 `TicketValidator`：

```
<bean id="casAuthenticationProvider"  
    class="org.springframework.security.cas.authentication.CasAuthenticationProvider">  
    <property name="ticketValidator" ref="samlTicketValidator"/>  
    <property name="serviceProperties" ref="casService"/>
```

重启 JBCP Pets 应用就会使用 SAML 响应进行 ticket 校验了。注意的是，CAS ticket 响应并没有被 Spring Security 进行日志记录，所以你要么对 JA-SIG CAS 客户端类启用日志（在 `log4j` 中对 `org.jasig` 启用日志）要么使用调试查看响应的不同。

总之，建议使用 SAML ticket 校验而不是 CAS 2.0 ticket 校验，因为前者提供了更多的防治重造的功能，包括时间戳校验并以标准的方式解决了属性问题。

属性查询的用处

要记住的是 CAS 为我们的应用提供了一层抽象，移除了我们引用对用户存储的直接访问，作为替代所有这样的访问通过 CAS 作为代理来进行。

这个功能很强大！这意味着我们的应用不再关心用户在什么类型的存储中，也不用关心怎样访问它们的细节——只需与 CAS 确认一个用户有权限访问应用。对于系统管理员来说，这意味着，如果 LDAP 改名、转移位置或者修改的话，他们只需要在一个位置进行重新配置——CAS。通过 CAS 进行集中访问使得组织中安全架构更高级别的灵活性和适应性成为可能。

扩展这个话题到从 CAS 获取属性的用处——现在，所有通过 CAS 认证的应用对用户都

有了相同的视角（译者注：即获取的信息是一致的），从而能够在任何使用 CAS 的环境下以一致的格式显示信息。

注意的是，一旦认证完成，Spring Security CAS 不会再次查询 CAS 除非用户需要重新认证。这意味着，存储在应用本地用户 Authentication 对象中的属性和其它用户信息可能会失效且可能与 CAS 服务器不一致。注意要设置 session 的超时时间以避免这种潜在的问题。

其它的 CAS 功能

除了通过 Spring Security CAS 包装暴露出来的功能，CAS 还提供了高级配置功能。其中一些如下：

- 对访问多个使用 CAS 安全的应用提供了透明的单点登录功能，这要在在一个可配置（在 CAS 服务器端）的时间内。通过在 TicketValidator 中将 renew 属性设置为 true，应用可以强制用户到 CAS 进行认证——你可能希望在自定义代码中有条件的设置这个属性，如当用户试图访问应用中高安全性的区域时；
- 为不能直接访问 CAS 的二级应用提供了 ticket 代理功能。当从 CAS 请求一个 ticket 时，web 应用可能通过二级代理应用来请求授权 ticket。关于这个的更多功能可以阅读 CAS 网站 (<http://www.jasig.org/cas/proxy-authentication>)；
- 协调拥有活跃 CAS session 的多个参与应用间的单点登出（这不是通过 Spring Security 直接支持的，而是 CAS 客户端通过联合使用 HttpSessionListener 和 servlet 过滤器）。

我们建议你去探索 CAS 客户端和服务端的所有功能，并在 JA-SIG 社区论坛中向大家提问题。

小结

在本章中，我们学习了中心认证服务(CAS)单点登录门户，以及它怎样与 Spring Security 集成，在本章中，我们学到了如下内容：

- CAS 架构以及在使用 CAS 的环境中各个参与角色的通信；
- 使用 CAS 的应用给开发人员和系统管理员带来的好处；
- 配置 JBCP Pets 与基本的 CAS 安装交互；
- 更新 CAS 与 LDAP 交互并实现 LDAP 与使用 CAS 的应用共享数据；
- 使用修改过的 CAS 2.0 协议和工业标准的 SAML 协议实现与 CAS 的属性交换。

我们希望本章的内容是关于单点登录的一个有趣开端。在市场上还有很多其它的单点登录系统，它们中大多数是商用的，但是 CAS 在开源 SSO 领域无疑是领导者，并且是在任何组织中构建单点登录的绝佳平台。

在下一章中，我们将会转回标准认证，但是这次我们将会移除用户名和密码而是依赖一种新的认证机制。有趣吗？那就开始吧！

第十一章 客户端证书认证（Client Certificate Authentication）

尽管用户名和密码认证特别常见，就像我们在第一章：一个不安全应用的剖析和第二章：*Spring Security 起步*所讨论的那样，form 认证的存在允许用户提供各种类型的凭证。Spring Security 也为这种需求提供了支持，在本章中我们将不再讨论基于 form 的认证并探索使用可信任的客户端证书。

在本章的内容中将会包括：

- 学习在客户端证书认证中，用户浏览器和服务器之间是如何交互的；
- 配置 Spring Security 使用客户端证书认证用户；
- 理解 Spring Security 中客户端证书认证的架构；
- 探索客户端证书认证的高级配置选项；
- 了解客户端证书的优劣以及在处理过程中的常见问题。

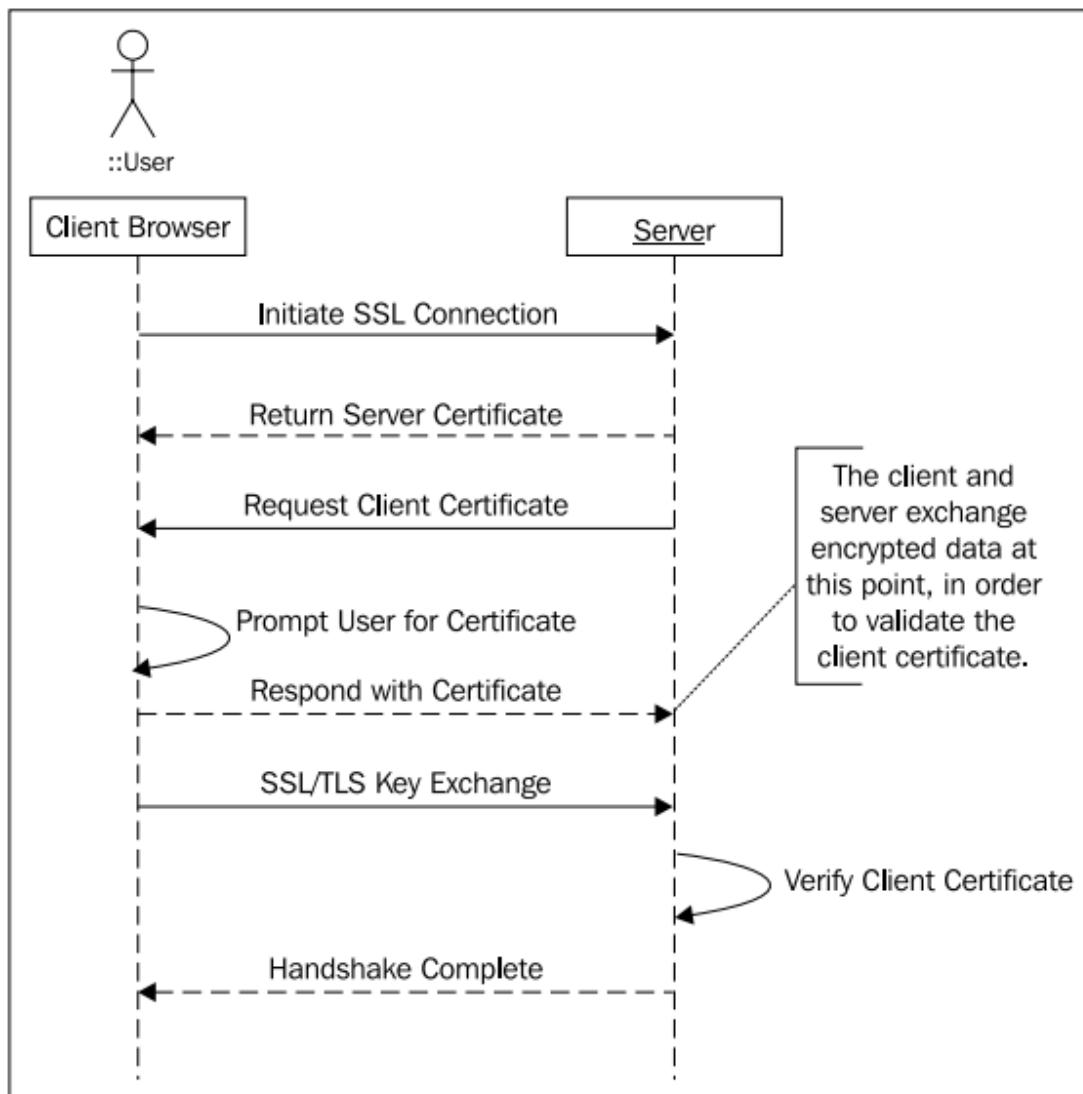
客户端证书认证是怎样工作的

客户端证书认证需要服务器向浏览器发送请求信息而浏览器进行响应，这样来建立用户（和他们的浏览器）与服务端应用的可信任认证关系。这个可信任的关系是通过交换可信的以及可认证的凭证建立起来的，而这也就是所谓的证书（certificates）。

与前面见到的不同，在客户端证书认证中，servlet 容器或应用服务器本身要负责在浏览器和服务器之间建立可信的关系，这是通过请求证书、评估并接受其合法性。

客户端证书认证也被成为共同认证（mutual authentication），是安全 socket 层（Secure Sockets Layer, SSL）协议及其后续协议传输层安全协议（Transport Layer Security, TLS）的一部分。因为共同认证是 SSL 和 TLS 协议的一部分，它需要 HTTPS 连接（通过 SSL 或 TLS 进行安全防护）进行客户端证书认证。关于 Spring Security 中 SSL/TLS 的更多细节，请参考我们在第四章：凭证安全存储中对 SSL/TLS 的讨论和实现。为了实现客户端证书认证需要在 Tomcat（或者你正在使用要运行例子的应用服务器）中建立 SSL/TLS。就像第四章那样，在本章的后续内容中我们将会用 SSL 代替 SSL/TLS。

下面的序列图展现了客户端浏览器与 web 服务器之间的交互，在这个交互中进行了 SSL 连接的建立并检验用来做共同认证的客户端证书的可信性。



我们可以看到两个证书的交换（服务端证书和客户端证书）提供了两端参与者的认证并确信他们之间的继续会话是安全的。清晰起见，我们省略了一些 SSL 握手和证书信任检查的细节，但是，我们建议你去阅读 SSL 和 TLS 的协议和证书，关于这些课题有很多好的参考手册。RFC 5246，《The Transport Layer Security (TLS) Protocol V1.2 (<http://tools.ietf.org/html/rfc5246>)》，是学习客户端证书的很好入门资料。如果你想了解更细节的东西，那 Eric Rescorla 的 *SSL and TLS: Designing and Building Secure Systems* 是关于协议和实现的极佳细节材料。

基于客户端证书认证的另一个名称是 X.509 认证。术语 X.509 来源于 X.509 标准，最初由 ITU-T 组织发布，用于在 X.500 标准下的目录（LDAP 的起源，你可能会想起第九章：LDAP 目录服务）下使用。后来，这个标准修改后用在了安全的互联网通信上。

我们提及这些是因为在 Spring Security 中与这个主题有关的类都与 X.509 相关。要记住的是 X.509 并没有定义共同认证协议本身，但是定义了证书的格式和结构并包含了可信任证书的权限。

建立客户端证书认证的设施

对于个体的开发人员来说不好的消息是，为了体验客户端证书认证需要一些并不简单的配置并事先建立起与 Spring Security 的简单集成。鉴于这些建立步骤对于初学者来说可能会遇到很多问题，所以我们觉得给读者进行介绍是很重要的。

我们假设你使用本地的、自认证的服务器证书以及自认证的客户端证书，Apache Tomcat 也是如此。这在大多数的开发环境中很常见，但是，你可能会访问合法的服务器证书、一个证书中心（CA）或者其它的应用服务器。如果是这样的话，你需要使用它们的安装指令并以类似的方式配置你的环境。请参考第四章的 SSL 建立指令来帮助配置 Tomcat 和 Spring Security 在单独的环境中使用 SSL。

理解公钥设施的目的

本章会关注为了教学而建立完整的开发环境，但是在大多数的情况下，你需要将 Spring Security 集成到已有的客户端校验安全环境中，那里会有很多的设施（通常会是硬件和软件的组合）来提供诸如证书授权和管理、用户服务以及撤销等功能。这种类型的环境会有公钥设施——联合使用硬件、软件和安全策略以实现高度安全、以认证为驱动力的网络系统。

为了进行 web 应用的认证，环境中的证书或硬件设备能用来进行安全、不可抵赖的电子邮件（使用 S/MIME），网络认证，甚至基于物理元件的访问（使用基于 PKCS 11 的硬件设施）。（In addition to being used for web application authentication, certificates or hardware devices in these environments can be used for secure, non-repudiated email (using S/MIME), network authentication, and even physical building access (using PKCS 11-based hardware devices))

因为管理这样的环境会比较困难（需要 IT 和工序都能够很好完成），所以这无疑需要专业的安全人员来操作这样的环境。

创建客户端证书密钥对

创建自签名的客户端证书与创建自签名的服务端证书是一样的，要通过 keytool 命令来创建密钥对。客户端密钥对的不同在于它需要 web 浏览器能够访问 key store 并且需要将客户端的公钥加到服务器的 trust store 中（我们稍后将会介绍它是什么）。

创建客户端的密钥对如下：

```
keytool -genkeypair -alias jbcpcclient -keyalg RSA -validity 365 -keystore jbcppets_clientauth.p12  
-storetype PKCS12
```

当提示为客户端证书输入名字（common name 或 DN 即拥有者 DN 的一部分）时，确保第一个提示的输入要与 Spring Security JDBC 存储中已有的用户相匹配，如 admin：

```
What is your first and last name?  
[Unknown]: admin  
... etc  
Is CN=admin, OU=JBCP Pets, O=JBCP Pets, L=Anywhere, ST=NH, C=US  
correct?  
[no]: yes
```

当我们配置 Spring Security 访问证书授权用户信息时，将会看到为什么这（个设置）很重要。在 Tomcat 中进行证书授权前，我们还有最后的一步。

配置 Tomcat 的 trust store

回忆一下密钥对的定义包含一个私钥和公钥。与 SSL 证书校验和保护服务器通信类似，校验客户端证书需要检查创建它的可信任者。

因为我们使用 keytool 命令自己创建的自签名客户端证书，所以 java 虚拟机不会像信任认证中心的那样信任它。

这样我们就需要强制要求 Tomcat 将这个证书视为可信任的证书。我们要达到这个目的需要将密钥对的公钥导出并将其添加到 Tomcat 的 trust store 中。

首先，我们要将公钥导出到一个标准的证书文件中，并将其命名为 jbcppets_clientauth.cer，如下：

```
keytool -exportcert -alias jbcpcclient -keystore jbcppets_clientauth.p12 -storetype PKCS12  
-storepass password -file jbcppets_clientauth.cer
```

接下来，我们要将证书导入到 trust store 中（这将会创建一个 trust store，但是在典型的部署环境中，你可能会在 trust store 中拥有其它的证书）：

```
keytool -importcert -alias jbcpcclient -keystore tomcat.truststore -file jbcppets_clientauth.cer
```

这将创建名为 tomcat.truststore 的 trust store 并提示你输入密码。你会看到一些证书相关的信息并最后被询问是否信任这个证书：

```
Owner: CN=admin, OU=JBCP Pets, O=JBCP Pets, L=Anywhere, ST=NH, C=US  
Issuer: CN=admin, OU=JBCP Pets, O=JBCP Pets, L=Anywhere, ST=NH, C=US  
Serial number: 4b3fb3d9  
Valid from: Sat Jan 02 16:00:09 EST 2010 until: Sun Jan 02 16:00:09  
EST 2011  
Certificate fingerprints:
```

```
MD5: 02:69:16:3B:D7:C2:74:9E:F7:FD:18:C9:C5:E4:C8:94  
SHA1: 65:57:94:6D:D2:83:7E:51:19:CF:58:94:ED:43:11:F6:AC:D0:FB:EC  
Signature algorithm name: SHA1withRSA  
Version: 3
```

Trust this certificate? [no]: yes

复制这个新的 tomcat.truststore 文件到你使用 Tomcat 服务器的 conf 目录下。在最后的成功前还要一个最后的配置！

【Key Store 和 Trust Store 的区别是什么？Java Secure Socket Extension (JSSE)对 key store 进行了如下的定义：私钥和对应公钥的存储机制。key store（包含密钥对）被用来加密或解密安全信息等功能。trust store 原本用来只存储公钥以实现校验身份时的可信任通信（如我们在证书校验时看到 trust store 是怎样被使用的）。但是，在很多常见的管理场景中，key store 和 trust store 被结合到一个文件中（在 Tomcat 中，这可以通过使用 Connector 的 keystoreFile 和 truststoreFile 属性来完成配置）。这些文件本身的格式完全相同（实际上，每个文件可以是任意的 JSSE 支持的 keystore 格式，包括 Java Key Store / JKS, PKCS 12 等）。】

最后，我们需要为 Tomcat 指明 trust store 并启用客户端证书认证。这需要在 Tomcat 的 server.xml 文件中为 SSL Connector 添加三个额外的属性：

```
<Connector port="8443" protocol="HTTP/1.1" SSLEnabled="true"  
maxThreads="150" scheme="https" secure="true"  
sslProtocol="TLS"  
keystoreFile="conf/tomcat.keystore"  
keystorePass="password"  
truststoreFile="conf/tomcat.truststore"  
truststorePass="password"  
clientAuth="true"  
/>
```

这是建立 SSL 时，触发 Tomcat 请求客户端证书所需要的配置。到这里，我们可以重新启动 Tomcat 了。

【还有一个其它的方式来配置 Tomcat 使用客户端证书认证——我们将会在稍后启用。现在，我们需要在第一次连接 Tomcat 服务器的时候就要使用客户端证书。这会更容易的判断出设置是否正确。】

最后一步是将客户端证书导入到客户端的浏览器中。

导入证书到浏览器中

根据你使用浏览器的不同，导入证书的过程会有所区别。我们将会提供 Firefox 和 IE 的指南，但是如果你使用其它的浏览器，请查询其帮助文档或你喜欢的搜索引擎寻找帮助。

使用 Firefox

按照以下的步骤来导入包含客户端证书密钥对的 key 存储：

1. 打开“工具”菜单；
2. 点击“选项”菜单项；
3. 点击“高级”按钮/图标；

4. 点击“加密”tab 标签；
5. 点击“查看证书”按钮。将会打开“证书管理器”；
6. 点击“您的证书”tab 标签；
7. 点击“导入”按钮；
8. 找到你存储 jbcppets_clientauth.p12 文件的位置并将其选中；
9. 你需要输入创建这个文件时的密码。

客户端证书就导入了，你可以在列表中看见它。

使用 IE

IE 与 Windows 操作系统紧密集成，所以它导入 key 存储会比较容易：

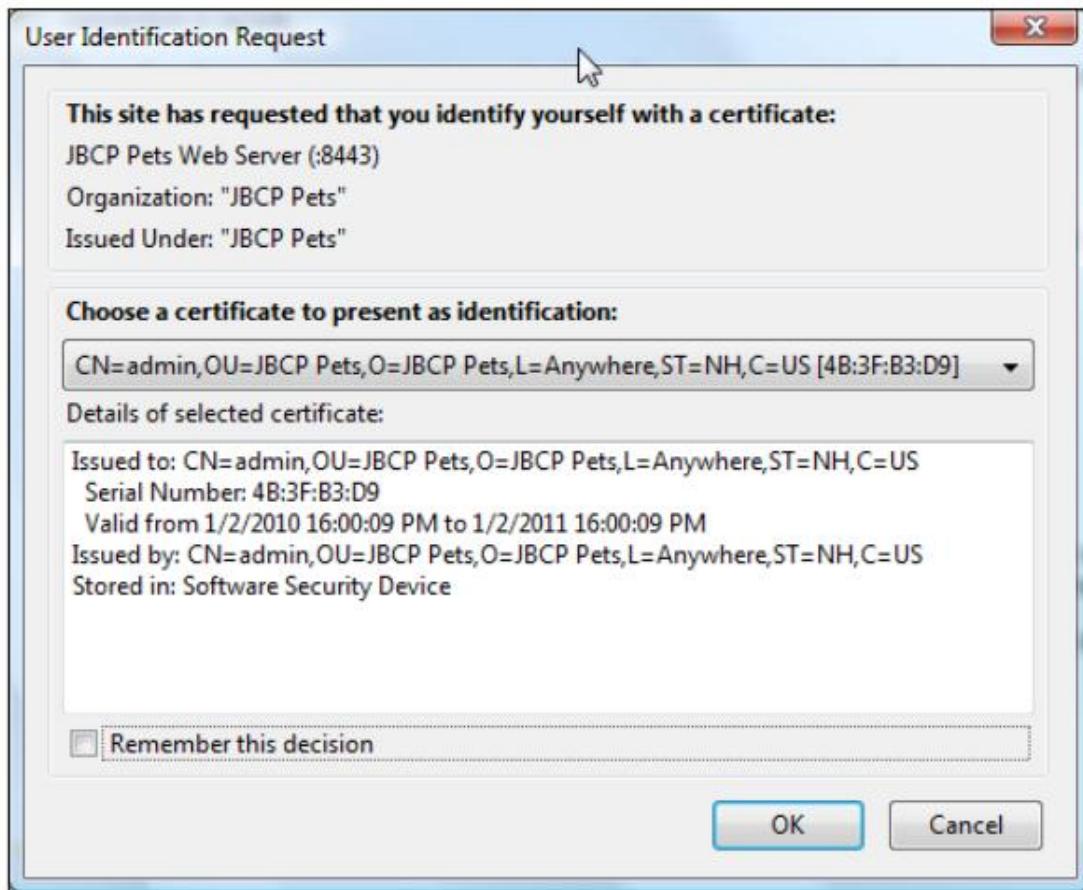
1. 在 Windows Explorer 中双击 jbcppets_clientauth.p12 文件。将会启动“证书导入向导”；
2. 点击“下一步”接受默认选项直到提示你输入证书密码；
3. 输入证书密码，并点击“下一步”；
4. 接受“自动选择证书存储选项”并点击“下一步”；
5. 点击“完成”。

为了校验证书正确安装，你需要进行一下的步骤：

1. 在 IE 中打开“工具”菜单；
2. 点击“internet 选项”菜单项；
3. 点击“内容”tab 标签；
4. 点击“证书”按钮；
5. 点击“个人”tab 标签（如果它没有被选中的话）。你会看到证书列在此处。

测试

现在你可以使用客户端证书连接到 JBCP Pets。如果所有的都被成功设置，在访问站点时会提示要求证书——在 Firefox 下，证书展现如下：



但是，当你试图访问站点中受保护的区域，如“*My Account*”区域，你将会被重定向到登录页面。这是因为我们没有配置 Spring Security 辨别证书中的信息——到这里，客户端和服务器的所有交互停止在 Tomcat 服务器本身那里。

客户端证书认证的问题解决

不幸的是，如果说第一次就能将客户端认证配置正确且没有任何错误出现是很容易得，那我们是在骗你。事实上，尽管这是一个很伟大和强大的安全设施，但是浏览器和 web 服务器的文档都很匮乏，出错信息在好的情况下会比较令人费解而在不好情况下根本就是误导性的。

记住到此时我们还有涉及到 Spring Security，所以调试器可能帮不上什么忙（除非你手头有 Tomcat 的源码）。一些常见的错误和要检查的事情如下：

- 当你访问站点时，没有提示你要求证书。有很多情况可能会导致如此，而这也是最令人迷惑的问题。以下是要检查的事情：
 - ◆ 确保你使用的客户端浏览器正确安装了证书。如果你访问上面的网站被拒绝，有时你需要重启整个浏览器（关闭所有的窗口）；
 - ◆ 确保你访问的是服务器的 SSL 端口（在开发环境下一般为 8443），并在 URL 中选择了 https 协议。客户端证书对不安全的浏览器链接不会提供。确保浏览器也信任服务器的 SSL 证书，即使你强制它信任一个自签名的证书；
 - ◆ 确保你添加了 clientAuth 指令到 Tomcat 配置中（或者对等的任意你使用的应用服务器）；
 - ◆ 如果以上的所有都失败的话，使用网络分析器或包探测器，如 Wireshark

(<http://www.wireshark.org/>) 或 Fiddler2 (<http://www.fiddler2.com/>) 来了解线路上的交流和 SSL 密钥交换。

- 如果你使用的是自签名的客户端证书，确保公钥被导入到服务器端的 trust store 中。如果你使用的是 CA 签发的证书，确保 JVM 信任 CA 或者 CA 的证书被导入到了服务器 trust store 中。
- 特别在 IE 下，根本不会报告客户端证书失败的细节（只会报告一个“页面不能展现”的错误）。使用 Firefox 来查看是否会有客户端证书相关的错误。

在 Spring Security 中配置客户端证书认证

不同于我们到目前为止所使用的认证机制，使用客户端证书认证会使得用户的请求已经被服务器预先认证（pre-authenticated）了。因为服务器（Tomcat）已经确定用户提供了合法且可信的证书，所以 Spring Security 只需信任这个 assertion 的合法性。

安全登录过程的另一个组件还缺失，也就是对认证过的用户进行授权。这就是我们要配置 Spring Security 的地方——我们必须添加一个组件到 Spring Security 中，它能够辨认出用户 HTTP session（Tomcat 填充进去的）中的证书认证信息，并将提供的凭证信息与 Spring Security 的 UserDetailsService 进行校验。与所有的 UserDetailsService 一样，这会确定对于证书中声明的用户，Spring Security 是否了解（译者注：即 Spring Security 对应的存储中是否有该用户的信息），然后像其它登录的用户那样分配 GrantedAuthority。

使用 security 命名空间配置客户端证书认证

由于 LDAP 和 OpenID 的配置的复杂性，配置客户端证书认证会稍好一些。如果你使用的是 security 命名空间风格的配置，添加客户端证书认证只需一行配置变化，添加到<http>声明中：

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

  <x509 user-service-ref="jdbcUserServiceCustom"/>

</http>
```

如果你重启应用，你会再次被提示要求客户端证书，但是这次你能够访问需要授权的内容了。你能够从日志中看到（如果你启用的话）你已经以 admin 用户登录了！干得漂亮！

Spring Security 怎样使用证书信息

正如前面讨论的那样，Spring Security 在证书交换中的相关内容是从提供的证书中提取信息并将用户的凭证与用户服务进行匹配。我们在使用<x509>声明中没有看到的是使得这一切发生的魔力所在。回忆我们建立客户端证书时，一个类似于 LDAP DN 的唯一标识名（distinguished name，DN）会与这个证书关联：

Owner: CN=admin, OU=JBCP Pets, O=JBCP Pets, L=Anywhere, ST=NH, C=US

Spring Security 使用 DN 中的信息来确定安全实体的实际用户名并在 UserDetailsService

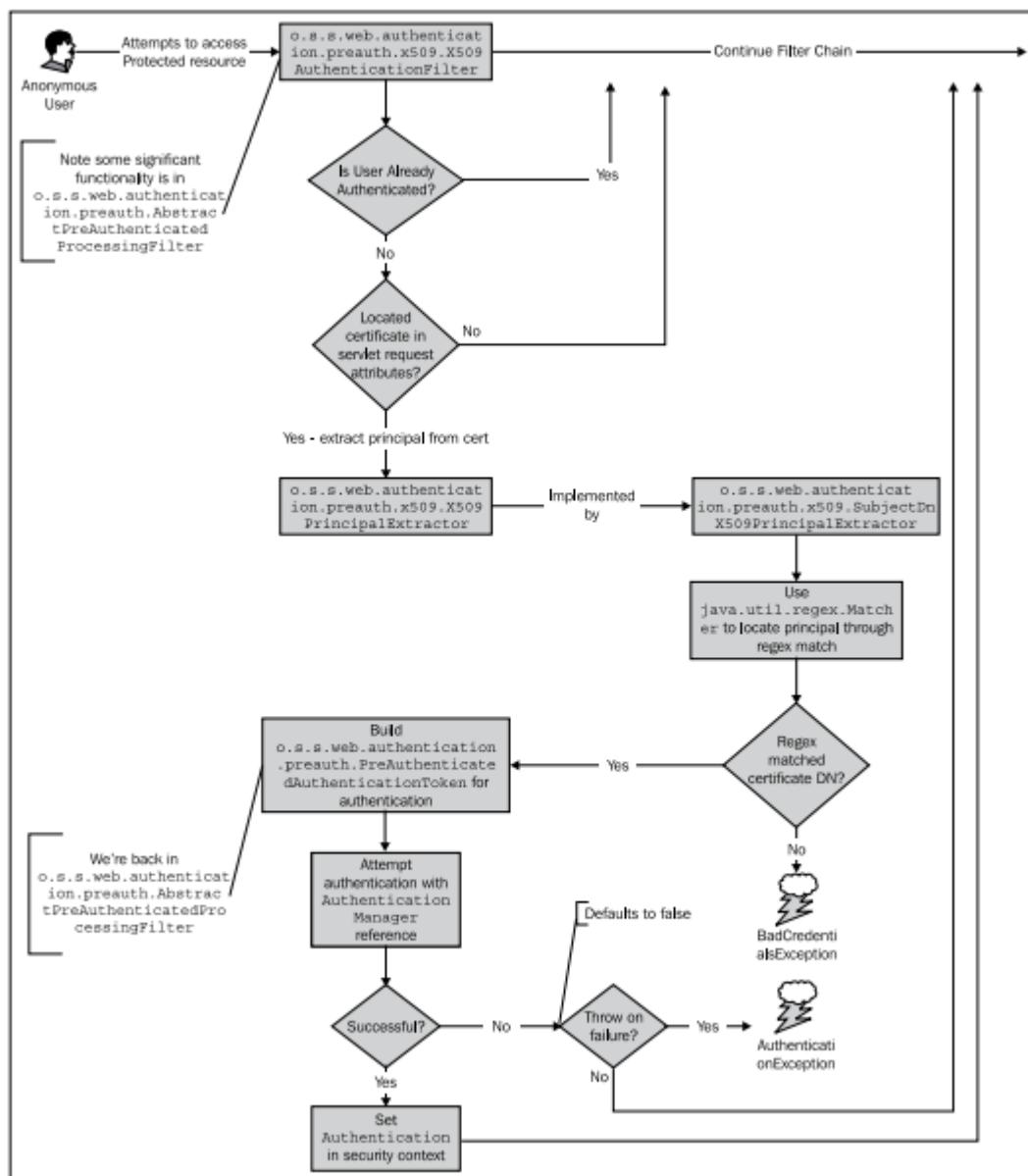
中进行查找。特别指出的是，它允许指明一个正则表达式，它会用来匹配证书建立时的 DN 并用 DN 的这一部分作为安全实体的用户名。<x509>默认配置的明确写法如下：

```
<x509  
subject-principal-regex="CN=(.*?),"  
user-service-ref="jdbcUserService"/>
```

我们可以看到，这个正则表达式将会匹配 admin 作为用户名。这个正则表达式必须包含一个匹配值，但是它能够被配置成支持用户名和 DN 以满足应用的需要——如，如果你们组织的证书包含 email 或 userid 域，正则表达式可以修改成使用这些值作为已认证安全实体的名字。

Spring Security 证书认证怎样实现

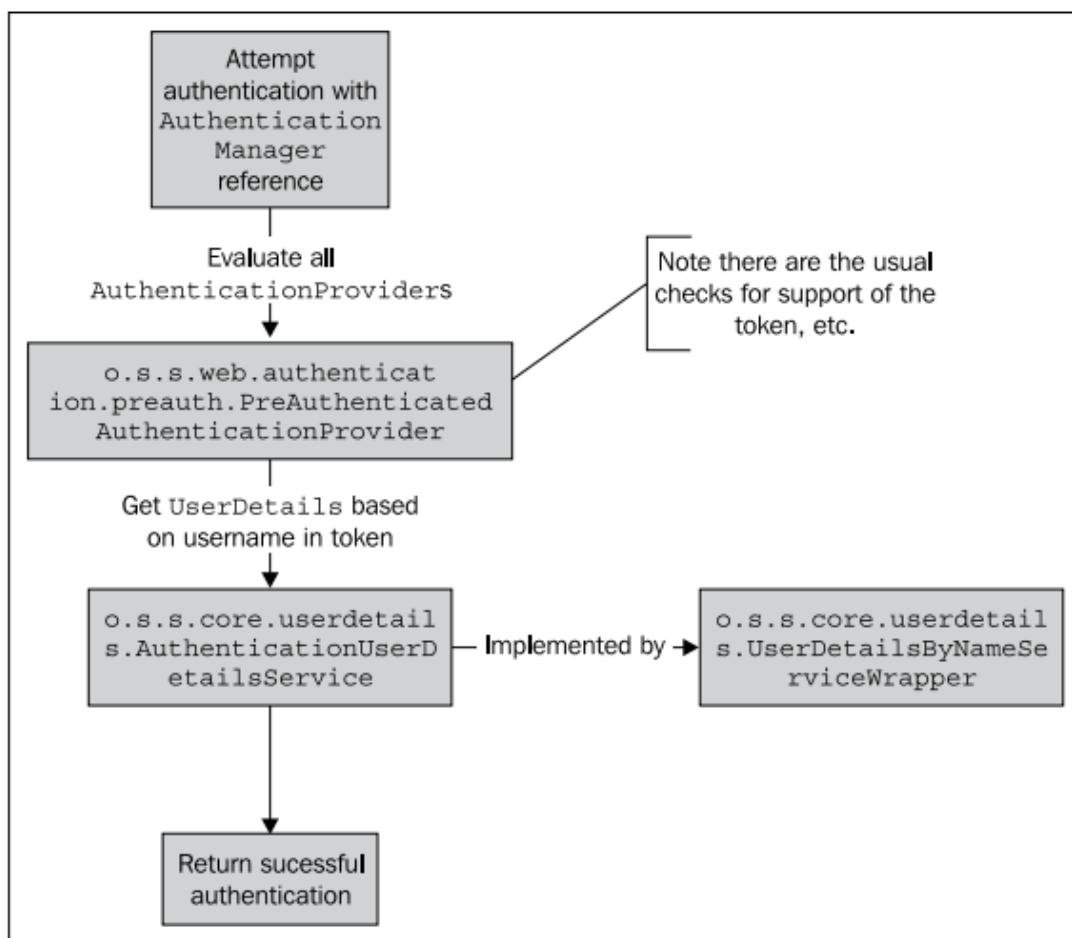
让我们看一下检查和评估客户端证书以及将其转换成 Spring Security 认证 session 的参与者，如下图：



我们可以看到 `o.s.s.web.authentication.preauth.X509AuthenticationFilter` 负责处理要求未认证用户提供客户端证书的请求。如果这个请求包含了合法的证书，它将会使用 `o.s.s.web.authentication.preauth.X509SubjectDnX509PrincipalExtractor` 抽取出安全实体，如前文所描述，这里会使用匹配证书自己 DN 的正则表达式。

【注意，尽管在图中描述了对未认证用户的证书检查，但是还有一个检查会进行即使用证书认证的用户是否为已经认证过的用户。这会使用新提供的凭证进行一个新的认证。这个的目的很明确——任何时间一个用户提供了新的凭证，应用必须能够意识到，并作出适当的响应以确保用户能够正确访问。】

一旦证书被接受（或拒绝/忽略），就像其它的认证机制那样，会构建一个 `Authentication token` 并传递给 `AuthenticationManager` 进行认证。我们现在可以简单看一下 `o.s.s.web.authentication.preauth.PreAuthenticatedAuthenticationProvider` 对认证 `token` 的处理：



如果你读过我们在第十章：使用 CAS 进行单点登录讲到的 CAS 认证，你会意识到客户端证书认证请求和 CAS 请求的相似之处——这是有意的设计，事实上相似的设计影响到 Spring Security 支持的其它的预先认证机制（较少使用），包括 Java EE 角色匹配和 Site Minder 风格的认证。如果你理解了客户端证书认证的流程，理解其它的认证流程就容易多了。

其它未解决的问题

我们要解决的一个问题就是处理对认证的拒绝。在第六章：高级配置和扩展中，我们曾经学习过的功能 `AuthenticationEntryPoint`（我们在 CAS 章节曾重温过这个话题）。在典型的 form 登录场景下，如果用户试图访问保护的资源却没有登录时，

LoginUrlAuthenticationEntryPoint 用来将用户重定向到登录页面。

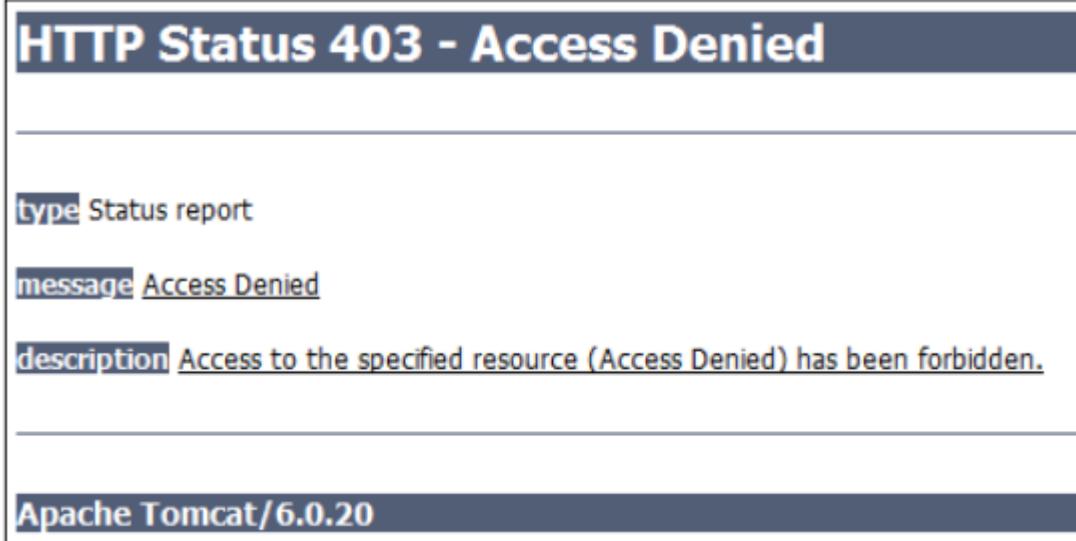
与之相反，在典型的客户端证书认证环境下，并不支持替代的授权方法（要记住的是，Tomcat 校验证书在 Spring Security 的 form 登录前面）。所以，并不会再存在默认重定向到 form 登录页的行为。相反，我们将会修改这个 entry point，使用 o.s.s.web.authentication.Http403ForbiddenEntryPoint 简单返回一个 HTTP 403 禁止访问的信息。我们将会在 dogstore-base.xml 文件中配置这个 bean，其它的 Spring bean 也位于此处，如下：

```
<bean id="forbiddenAuthEntryPoint"
      class="org.springframework.security.web.authentication.Http403ForbiddenEntryPoint"/>
```

接下来，通过在<http>声明上设置一个简单的属性就会使得新 entry point 马上生效：

```
<http ... entry-point-ref="forbiddenAuthEntryPoint">
```

如果一个用户试图访问受保护的资源，而不能提供合法证书的话，他们将会看到如下的页面：



不像我们在第六章看到的配置 AccessDeniedHandler 那样，Http403ForbiddenEntryPoint 不能将用户重定向到一个友好页面或 Spring 管理的 URL。作为替代，这样的配置应该在 web 应用部署描述符的<error-page>声明中配置，这是 Java EE servlet 规范声明的，或者实现 Http403ForbiddenEntryPoint 的子类以改写这个行为。

在常用的客户端证书认证中，其它的配置或应用流程要进行的调整如下：

- 同时移除基于 form 的登录页面；
- 移除所有的“Log out”链接（没有理由退出了，因为浏览器会始终提供用户的证书）；
- 移除用户重命名和修改密码功能；
- 移除用户注册功能（除非你能将其与发放新证书关联）。

支持双模认证（Dual-Mode authentication）

在一些环境下，可能会同时支持基于证书和基于 form 的认证。如果你的环境就是如此，也可以（很简单的）通过 Spring Security3 来进行支持。我们需要使用默认的 AuthenticationEntryPoint（重定向到 form 登录页）与用户交互并允许用户在没有提供客户端证书的情况下使用标准的 form 进行登录。

如果你选择这种方式配置你的应用，你需要调整 Tomcat 的 SSL 设置（对于你的应用服

务器也要适当调整)。只需将 `clientAuth` 属性的值设为 `want` 而不是 `true`:

```
<Connector port="8443" protocol="HTTP/1.1" SSLEnabled="true"
    maxThreads="150" scheme="https" secure="true"
    sslProtocol="TLS"
    keystoreFile="conf/tomcat.keystore"
    keystorePass="password"
    truststoreFile="conf/tomcat.truststore"
    truststorePass="password"
    clientAuth="want"
    />
```

我们还需要移除前面练习中添加的 `entry-point-ref`，这样如果浏览器在初始查询时不能提供一个合法的证书将会启用标准的基于 `form` 的认证流程。

尽管这很便利，但还是要记住几件关于双模认证(基于 `form` 和基于证书)的事情。

【一旦证书校验失败，大多数的浏览器不会提示用户再次要求证书，所以确保你的用户注意的是他们可能需要重新进入浏览器以再次提供证书。】

要注意的是使用证书认证用户的时候并不需要密码；但是，如果你还继续使用 `JDBC UserDetailsService` 来支持基于 `form` 的认证，可能会需要你使用 `UserDetailsService` 提供一些关于用户的信息给 `PreAuthenticatedAuthenticationProvider`。这会有潜在的安全风险，因为你只想让其进行证书认证的用户可能会有使用 `form` 登录认证的潜在问题。有几个方法来解决这个问题，描述如下：

- 确保证书认证的用户有一个合适的强密码在数据库中；
- 考虑自定义你的 `JDBC` 用户存储以明确可以使用 `form` 登录的用户。这可以通过在保存用户信息的表上添加一个新的域，并稍微调整 `JdbcDaoImpl` 使用的 `SQL` 查询；
- 为证书认证的用户配置一个单独的用户存储，与允许基于 `form` 登录的用户完全隔离。

双模认证可以是对你站点的一个很大增强功能，并能够有效部署和保证安全，这当然要以你记住用户能够允许访问的条件。

使用 Spring bean 配置客户端证书认证

在前面，我们曾经讲过客户端证书认证相关类的流程。所以，对于只使用基于 `bean` 的配置就会更容易了，它们都会在 `dogstore-explicit-base.xml` 文件中。我们将会添加如下的 `bean` 定义，这就对应我们迄今所讨论的所有 `bean`:

```
<bean id="x509Filter"
    class="org.springframework.security.web.authentication.preauth.X509AuthenticationFilter">
    <property name="authenticationManager" ref="customAuthenticationManager"/>
</bean>
<bean id="preauthAuthenticationProvider"
    class="org.springframework.security.web.authentication.preauth.PreAuthenticatedAuthenticationProvider">
    <property name="preAuthenticatedUserDetailsService"
        ref="authenticationUserDetailsService"/>
```

```
</bean>
<bean id="forbiddenAuthEntryPoint"
    class="org.springframework.security.web.authentication.Http403ForbiddenEntryPoint"/>
<bean id="authenticationUserDetailsService"
    class="org.springframework.security.core.userdetails.UserDetailsByNameServiceWrapper">
    <property name="userDetailsService" ref="jdbcUserService"/>
</bean>
```

我们还需要添加这个过滤器到我们的过滤器链中（比移除登录和退出相关的过滤器）：

```
<bean id="springSecurityFilterChain"
    class="org.springframework.security.web.FilterChainProxy">
    <security:filter-chain-map path-type="ant">
        <security:filter-chain pattern="/**" filters="
            securityContextPersistenceFilter,
            x509Filter,
            anonymousProcessingFilter,
            exceptionTranslationFilter,
            filterSecurityInterceptor" />
    </security:filter-chain-map>
</bean>
```

最后，我们需要添加 AuthenticationProvider 的实现到 ProviderManager 中，并移除原来存在的所有其它类：

```
<bean id="customAuthenticationManager"
    class="org.springframework.security.authentication.ProviderManager">
    <property name="providers">
        <list>
            <ref local="preauthAuthenticationProvider"/>
        </list>
    </property>
</bean>
```

到此为止，我们基于 bean 的配置就准备好了。如果你想实验它，记住要切换 web.xml 中的（配置文件）引用来使用只基于 bean 的配置机制。

基于 bean 配置的其它功能

Spring 基于 bean 的配置通过暴露 bean 属性为我们提供了一些其它的功能，而这些通过 security 命名空间风格的配置是没有暴露的。

X509AuthenticationFilter 的其它属性如下：

属性	描述	默认值
continueFilterChainOnUnsuccessfulAuthentication	如果为 false，一个失败的认证将会抛出异常而不是允许请求继续。这通常会在需要合法证书才能访问的安全站点中设置。如果	true

	为 true，即使认证失败，过滤器链将会继续。	
checkForPrincipalChanges	如果为 true，过滤器将会检查当前认证的用户名与客户端证书提供的用户名是否有所不同。如果这样的话，对于新证书的将会进行认证并且 HTTP session 将会失效(可选的，参照下一属性)。如果为 false 的话，一旦用户认证过，将会一直处于合法状态即便他修改了凭证。	false
invalidateSessionOn PrincipalChange	如果 true 并且请求中的安全实体发生变化，用户的 HTTP session 在重新认证前将会失效。如果为 false，session 将会保持——注意这可能会引入安全风险。	true

PreAuthenticatedAuthenticationProvider 为我们提供了两个有意思的属性，如下：

属性	描述	默认值
preAuthenticated UserDetailsService	用从证书中获取的用户名构建完整的 UserDetails 对象	None
throwExceptionWhenTokenRejected	如果为 true，当 token 不能正确构建时(证书中不包含用户名或没有证书)会抛出一个 BadCredentialsException 错误。在证书要明确要使用的环境中，一般设置为 true。	false

除了这些属性，还有很多的机会来实现接口或扩展证书认证相关的类来对你的应用进行更进一步的自定义。

实现客户端证书认证要考虑的事情

客户端证书认证，尽管非常安全，但是并不是适合所有人，也并不适合于所有的环境。
益处：

- 证书对双方(客户端和服务器)都建立起了相互信任并确保各自就是其所宣称的(参与者)(译者注：即能够确定提供证书的是谁)；
- 基于证书的认证，如果能够很好地实现，相对于其它的 form 认证很难被模仿或篡改；
- 如果使用良好支持的浏览器并且正确设置，通过透明登录到所有证书安全的应用，客户端证书认证能够有效地作为单点登录的手段。

弊端：

这种方式(使用证书)一般会要求所有的用户都要拥有证书。这会导致用户培训负担以及管理负担。对于大量使用证书认证的组织必须要有足够的服务和支持，以进行证书管

理、过期跟踪以及处理用户求助；

使用证书一般是一个全有或全无的事情，意味着一般不会对没有认证的用户提供混合模式的认证，这主要是因为 web 服务器的复杂配置以及应用的较差支持；

你的所有用户（包含移到设备）并不一定很好的支持证书；

正确配置证书认证的所有设施可能会需要高级的 IT 知识。

正如我们所看到的，客户端证书认证有利有弊。当它正确实现时，它对于所有的用户会是一种很便利的访问模式，并具有很吸引人的安全性和不可抵赖属性（non-repudiation）。你需要根据特定的情况来决定这种类型的认证是否适合。

小结

本章中，我们了解了基于客户端证书认证的架构、流程以及 Spring Security 提供的支持。我们学到了如下的内容：

- 了解客户端证书（相互的）认证的概念和整体流程；
- 学习配置 Apache Tomcat 支持自签名 SSL 和客户端证书的重要步骤；
- 配置 Spring Security 能够理解客户端提供的证书认证凭证；
- 理解 Spring Security 与证书认证相关的类；
- 学习怎样配置 Spring bean 风格的客户端证书环境；
- 权衡这种方式认证的利弊。

对于不熟悉客户端证书的开发者来说，可能会被这种环境的复杂性所困扰。我们希望本章的内容能够使得这个复杂的话题更容易理解和实现。

第十二章 Spring Security 扩展

在本章中，我们将会探索一个 Spring Security 扩展项目的功能——这是很令人兴奋的功能即将 Windows Active Directory 认证（或其它支持 Kerberos 的设施）与 Spring Security 集成以为你的 Intranet 用户提供完善的单点登录体验。

在本章中，我们将会：

- 学习 Kerberos 认证协议及其基于 web 的凭证；
- 理解使用 Kerberos 的 web 应用怎样适用于基于 Kerberos 的安全设施；
- 配置 JBCP Pets 应用通过使用 Active Directory 认证存储为 Windows 用户提供单点登录认证功能；
- 探索使用 Active Directory 作为 LDAP UserDetailsService 从而在唯一的位置存储用户数据。

Spring Security 扩展

Spring Security 扩展（Spring Security Extensions）项目（可以通过以下地址访问：<http://static.springsource.org/spring-security/site/extensions.html>）作为 Spring Security 扩展功能的孵化器，基于 Spring Security 的核心框架构建。尽管这个项目相对很新，但是它已经有三个令人兴奋的模块包括 Kerberos authentication（Spring Security 2 中 NTLM 的继续），

Security Assertion Markup Language 2.0 认证以及 Portlet 认证。

在本章中，我们将会介绍 Kerberos SPNEGO 认证的基本配置和使用。在写这些内容的时候，Spring Security 扩展都还没有官方释放，但是 Kerberos 项目足够稳定所以你读到这些内容的时候应该不会有明显的配置变化。通过这些非官方、社区开发的扩展，我们可以将本章的内容视为对 Spring Security 实验功能的探究。

Kerberos 和 SPNEGO 认证入门

Kerberos 是一个相互的认证协议，用来对认证客户端——独立的用户或网络资源——这通过使用名为 KDC (key distribution center) 的中心凭证存储。客户端与 KDC 之间的交互很复杂，在一些互联网标准中有很好的文档（主要是 RFC 4120, The Kerberos Network Authentication Service (V5)，可以在以下地址查阅：<http://tools.ietf.org/html/rfc4120>）。

在本章中为了进行讨论，我们将会简化 Kerberos 设施检查凭证活动的细节。Kerberos 认证和 Kerberos 设施的主要目的是对安全实体 (principal) 提供安全可信的认证，而这里的安全实体可以是用户、网络资源或软件应用。Kerberos 的协议本身和软件实现最初是由麻省理工学院 (MIT) 开发的。鉴于 Kerberos 有这样一个光辉的背景，有很多不错的书详细讲解 Kerberos 的细节。

在 Kerberos 认证协议之上，开发出了 Generic Security Service Application Program Interface (GSS-API)，这也是基于 RFC 标准 (RFC 2078, Generic Security Service Application Program Interface, Version 2, <http://tools.ietf.org/html/rfc2078>)。GSS-API 提供了标准的、跨平台、跨语言的认证和安全接口，并包装了 Kerberos (以及其他认证提供者)。它的开发目标是为安全开发人员提供一致的认证和安全编程接口而不需要了解特定安全协议的细节。

GSS-API 在 Java 语言中的标准实现在另一个 RFC 标准中定义 (RFC 2853, Generic Security Service API Version 2: Java Bindings, <http://tools.ietf.org/html/rfc2853>)，它在 Sun JVM 的 org.ietf.jgss 包中实现。

【在 java 支持 Kerberos 和 GSS-API 中，有一个很重要的事情是 API 的实现以及 API 本身是 JRE 的一部分，不需要引入任何第三方的库。实际上，你可以查询 Sun 的站点 (<http://java.sun.com/javase/6/docs/technotes/guides/security/jgss/tutorials/index.html>) 来了解这些 Sun JVM 功能的更多文档。注意的是，其它的 JVM 实现可能不会像 Sun JVM 那样提供相同的功能。】

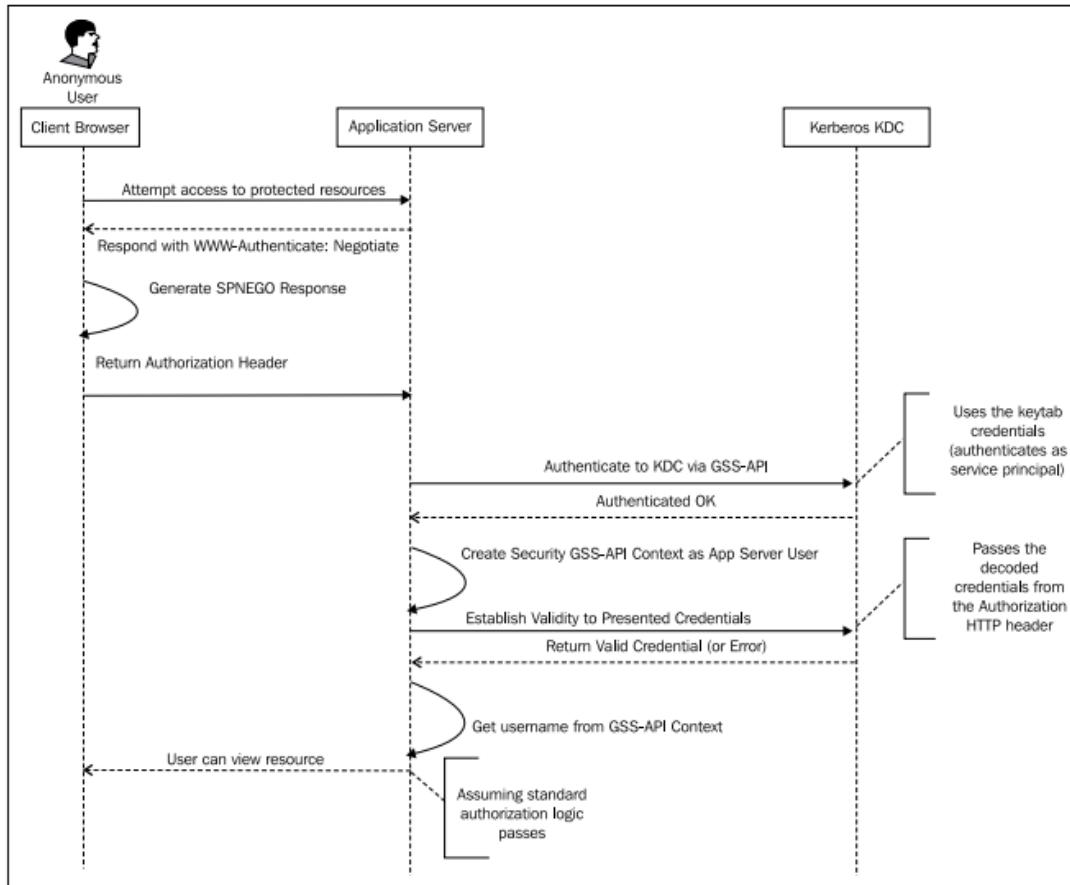
将 GSS-API 认证集成到 web 浏览器中是通过使用一个名为 Simple and Protected GSS-API Negotiation Mechanism (SPNEGO) 的信息交换标准做到的。SPNEGO 是明确两个 GSS-API 实现间行为的算法，确定了它们之间是否可以通过一个通用的安全协议进行通信。就 Kerberos SPNEGO 而言，我们希望客户端和服务端交互的通用安全协议是 Kerberos。

尽管 SPNEGO 可以被用来进行支持 GSS-API 的应用或系统通信，但是对于 web 应用开发人员和安全设计人员来说，这个词汇更多是意味着 web 客户端认证。Microsoft 支持的 RFC 4559 (SPNEGO-based Kerberos and NTLM HTTP Authentication in Microsoft Windows, <http://tools.ietf.org/html/rfc4559>) 描述了服务器通过 HTTP 协议使用 SPNEGO 请求来要求 GSS-API 认证。客户端浏览器就可以响应 SPNEGO 请求并利用原生的操作系统支持从终端用户那里收集凭证信息。Microsoft 为它的 Windows 操作系统开发了这个规范以支持两个 GSS-API 的实现——其专有的 NT LAN Manager (NTLM) 协议和开放的 Kerberos 协议。最终，这种从基于 Window 的客户端到 Kerberos 服务端资源的便利单点登录方式使得其他的浏览器也适应了这种基于浏览器的 SPNEGO，包括 Mozilla Firefox 和 Apple Safari。

希望我们没有使你掉进术语汤中（译者注：即被这些术语所迷惑）！让我们现实一些了

解这些在基于 web 认证的场景下是如何工作的。

下图展现了在 SPNEGO Kerberos 认证过程中，三个参与者之间的交互：



在进入 Spring Security 实现 Kerberos 的 SPNEGO 认证之前，理解客户端浏览器和应用服务器之间的两个重要 HTTP 交互是很重要的。

- 当需要认证时，应用服务器必须发送一个包含 `WWW-Authenticate: Negotiate` 值的 HTTP 响应头给客户端。这会告诉客户端需要使用 SPNEGO 协议进行认证；
- 客户端要确定用户的凭证(可能会使用弹出提示，这取决于浏览器实现和操作系统)，并将凭证信息编码到 `HTTP Authorize` 请求头中响应给服务器。

在这里我们更多关注了浏览器和应用服务器的交互，因为这是大多数 Spring Security 集成所关注的部分。

【尽管 SPNEGO 成功实现并没有要求，但是我们建议在生产环境中使用 SSL，这样 SSO 凭证能够在 web 客户端认证过程中保持安全。】

另一个要考虑的重要设施是 Kerberos KDC 实现。存在开源的选择，其中最主要的就是 MIT（著名的大学）的 Kerberos 实现，这也是 Kerberos 技术的最初贡献者之一，但是，在企业环境中，开发人员最常见的实现是 Microsoft Active Directory（AD）。Windows 服务器的 Active Directory 可以作为 Kerberos KDC 的功能，所以如果一个组织使用了 AD，那也就会自动启用了基于 Kerberos 的认证。

在 Spring Security 实现 Kerberos 认证

与我们已经看到过的 CAS 和客户端证书认证类似，对于使用 Kerberos 安全的站点 Kerberos Single Sign-On (SSO) 将会作为唯一支持的认证机制（尽管在本章临近结束的时候我

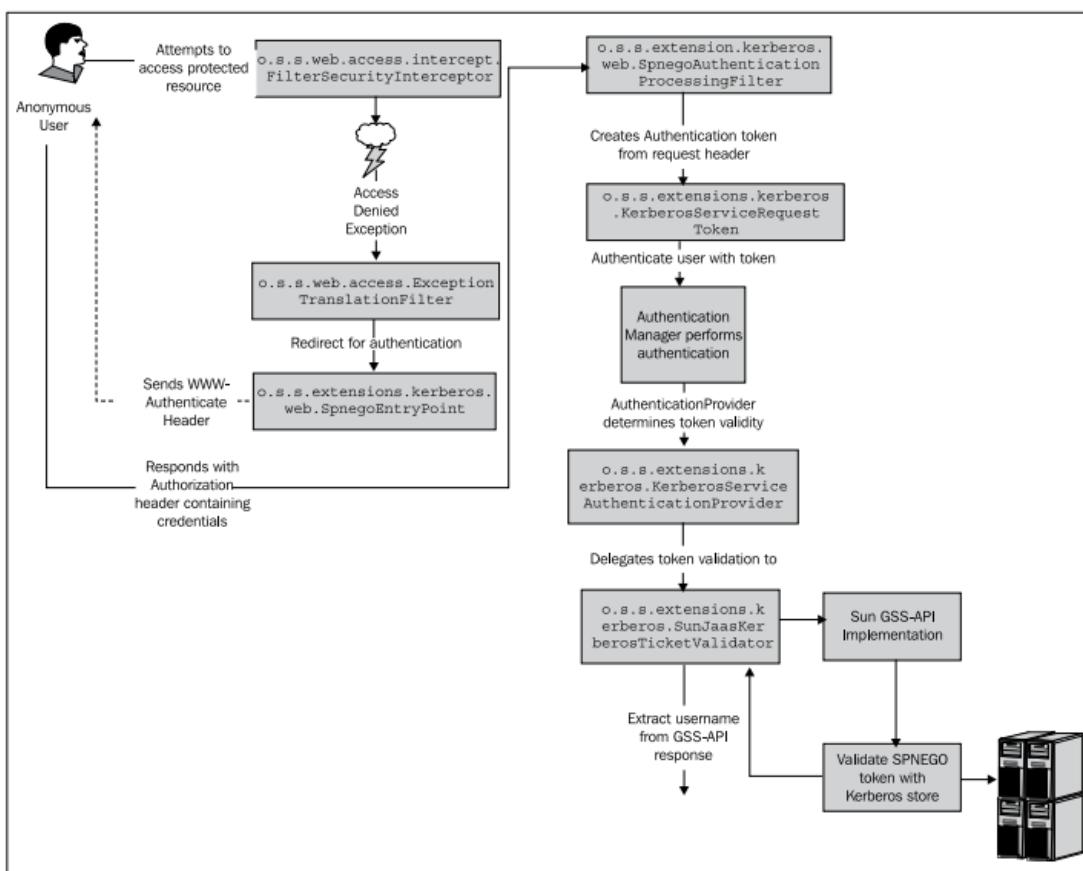
们将会看待一种替代的配置以支持 form 登录）。让我们看一下为 Spring Security 配置支持 Kerberos 的基本步骤，随后，我们将会介绍 Kerberos 环境中其它参与者所需要的配置。

Kerberos Spring Security 认证整体流程

正如我们在整体介绍 SPNEGO 协议设计时所讲的，关键实现在于客户端浏览器与安全应用之间的信息交换。这就是 Spring Security's Kerberos Extension 介入的地方——来处理认证凭证的交互。

如同典型的外部认证存储，如我们看到过的 CAS（在第十章：使用 CAS 进行单点登录和第十一章：客户端证书认证），联合使用一个 servlet 过滤器和自定义的 AuthenticationProvider 来进行外部存储的认证并校验返回响应的可靠性和可验证性。

让我们看一下使用 Spring Security 进行 SPNEGO 认证相关的重要类，如下图：



我们可以看到 `o.s.s.extensions.kerberos.KerberosServiceAuthenticationProvider` 主要负责协调校验浏览器 SPNEGO 响应的合法性。稍后，我们将会介绍将这些类组织在一起的 Spring Bean 配置细节。

准备工作

配置任何的 Kerberos 环境，尤其是 Microsoft Active Directory 后台，而你又没有 Kerberos 经验的话会很耗时和复杂。如果你要配置使用 Kerberos 的应用到一个环境中去，在开始之前要确保环境被正确配置了，否则你将花费更多的时间在诊断环境上而不是开发你的应用上。

在本课程中，我们假设你写的应用要到 Microsoft Active Directory (AD) 环境中进行认证，这（对于大多数用户）是结合 Spring Security 和 Kerberos 的最常见场景。如果你就是这样，当你读到“Kerberos 安全实体”(Kerberos principal)，只需将其等同于 AD 用户。

在开始之前，要确保：

你已经为 web 应用本身建立的一个 Kerberos 安全实体。我们将会使用这个账号进行应用服务器的附加配置；

连接站点的电脑（也就是 web 浏览器运行的机器）必须是 Kerberos 认证域（Kerberos authentication realm）的一部分。对于 Windows 用户，这意味着这个电脑必须是 AD 域（AD domain）的一部分；

运行 web 浏览器的电脑运行 web 应用的电脑不能是同一台机器。对你来说，虚拟机是很便利的选择。

记住的是 Kerberos SSO 一般部署在 intranet 应用中，并运行在 AD 或 Kerberos Realm 里，而不会用在面向公众的 web 应用中。

例子的假设

在本章的 Kerberos 例子中，我们假设要建立的环境如下：

配置	值	描述
域名 (Domain Name)	jbcppets.com	我们公司网络和站点的域名
AD 域 (AD domain)	corp.jbcppets.com	Active Directory 域——与 Kerberos 域名匹配（简介起见缩写为 CORP）
Web 站点安全实体 (Website principal)	CORP\website	对应于 web 站点服务的 AD 用户

创建 keytab 文件

你需要创建一个 keytab 文件，它用来认证要访问 KDC 的 web 应用并允许对用户提供的凭证进行认证。keytab 文件是一个 Kerberos 安全实体私钥的加密备份，在向 Kerberos 服务器认证安全实体的时候能够代替密码。

我们的应用使用基于 web 的 SPNEGO Kerberos 认证。相关的协议 RFC (RFC 4559) 规定了 keytab 必须匹配安全实体且以 HTTP/fully.qualified.web.server.name 作为标识符（这必须与规范声明完全一致）。在我们实例的配置中，我们假设 JBCP Pets 应用运行在 web.jbcppets.com 上并在 Kerberos 域 corp.jbcppets.com 中，所以在 keytab 中的实体名应该是 HTTP/web.jbcppets.com@CORP.JBCPPETS.COM。

因为我们以 AD 作为 Kerberos KDC，所以我们需要将 AD 用户 CORP\website 与需要的安全实体进行匹配。我们可以使用 Microsoft 的 ktpass 工具（在 AD 域控制器中）来完成，如下：

```
ktpass -princ HTTP/web.jbcppets.com@CORP.JBCPPETS.COM -mapuser CORP\website -out website.keytab
```

你会看到命令的参数与我们以下描述的配置场景所匹配：

- HTTP/web.jbcppets.com@CORP.JBCPPETS.COM：Kerberos 安全实体的名字

(HTTP/web.jbcppets.com) 与域 (CORP.JBCPPETS.COM);

- CORP\website: 与 Kerberos 安全实体相匹配的域用户名。

注意的是 Kerberos 域是大小写敏感的，所以要确保在所有环境中大小写一致。便利起见，Kerberos 域通常声明为大写。

这将会在当前目录下创建名为 website.keytab 的文件。你需要安全的将这个文件传输到运行 web 应用的机器上以在 Spring Security 中配置 Kerberos。将其放在 JBCP Pets web 应用的 WEB-INF/classes 目录下——当我们稍后配置 Spring Security Kerberos bean 的时候会用到。

【留意这个 keytab 文件——它包含了到 Kerberos 域信息，这些信息与预认证凭证相同。不要使用不安全的文件传输技术来复制这个文件到目标服务器，因为这会使你有被网络监听的风险。如果这个文件是缺乏安全的，恶意用户可以使用这些凭证登录你的 Kerberos 域，就像 web 应用用户那样。】

注意，ktpass 包含在 Windows 2008 Server 和以后的版本中。以前版本的 Windows Server 可能需要安装 Kerberos 支持文件才能使用这些命令行工具。

配置 Kerberos 相关的 Spring bean

鉴于大多数的 Kerberos 认证发生在 Sun JVM 里面，所以在 Spring Security Kerberos 认证中没有太多的配置（甚至代码）。

首先，我们要配置 AuthenticationEntryPoint 来负责发送 WWW-Authenticate 头，它会首先进入客户端机器并响应 Kerberos 凭证。在 dogstore-base.xml 中的 bean 声明如下：

```
<bean id="kerbEntryPoint"
      class="org.springframework.security.extensions.kerberos.web.SpnegoEntryPoint" />
```

接下来，我们要声明解析传入 Authorization HTTP 请求头并将其按照 SPNEGO 登录请求进行处理的过滤器：

```
<bean id="kerbAuthenticationProcessingFilter"
      class="org.springframework.security.extensions.kerberos.web.SpnegoAuthenticationProcessingFilter">
    <property name="authenticationManager" ref="authenticationManager" />
</bean>
```

要记住的是，与我们之前看到的其它 SSO 实现相同，这个过滤器负责解析 SSO 头并基于这个头信息来尝试认证用户。SpnegoAuthenticationProcessingFilter 将会基于 HTTP Authorization 头中的凭证填充一个 o.s.s.extensions.kerberos.KerberosServiceRequestToken 对象。

现在，我们需要一个 AuthenticationProvider，它要提取被填充的 KerberosServiceRequestToken 对象里面的凭证信息并进行校验。与我们看到的 CAS 认证类似，Kerberos AuthenticationProvider 要负责将 token 与 ticket 授予者进行校验。

```
<bean id="kerberosServiceAuthenticationProvider"
      class="org.springframework.security.extensions.kerberos.KerberosServiceAuthenticationProvider">
    <property name="ticketValidator" ref="ticketValidator"/>
    <property name="userDetailsService" ref="jdbcUserService" />
</bean>
<bean id="ticketValidator"
```

```
class="org.springframework.security.extensions.kerberos.SunJaasKerberosTicketValidator">
<property name="servicePrincipal" value="HTTP/web.jbcppets.com@CORP.JBCPPETS.COM" />
<property name="keyTabLocation" value="classpath:website.keytab"/>
</bean>
```

KerberosServiceAuthenticationProvider 需要引用
o.s.s.extensions.kerberos.KerberosTicketValidator 代理实现，后者用来校验从认证 token 所获取的 Kerberos ticket。Spring Security Kerberos Extension 中提供的唯一实现是使用 Sun JVM's GSS-API 来校验 keytab 的内容并为服务实体执行对 Kerberos ticket 的校验。

我们可以看到 servicePrincipal 和 keyTabLocation 引用了我们在前面 Kerberos server 中所作的配置。

【keytab 文件应该放在哪里？我们记得 keytab 文件是高安全风险的元素，所以，不建议将其放在应用的 classpath 中（尽管这部署起来很方便）。相反，建议将其放在 web 应用部署目录以外的文件系统中。你可以使用 Spring 标准的 file: 语法来引用这个文件。】

你可以看到我们配置引用了 JDBC UserDetailsService（临时的）。你需要记往往 JDBC 启动 SQL 中添加一个新的用户，其用户名要与试图登录的 Kerberos 用户相匹配。例如，如果你想以用户 kerbuser 登录应用，所以数据库中完整的用户名应该是包含完整 Kerberos 实体的 kerbuser@CORP.JBCPPETS.COM。将其添加到 WEB-INF/classes/test-users-groups-data.sql 中，如下：

```
insert into users(username, password, enabled, salt) values
('kerbuser@CORP.JBCPPETS.COM','unused',true,CAST(RAND() * 1000000000 AS varchar));
insert into group_members(group_id, username) select id,'kerbuser@CORP.JBCPPETS.COM' from
groups where group_name='Administrators';
```

需要注意的是使用 Active Directory，一般会用到 LDAP UserDetailsService，甚至一个自定义的来适应你的业务需求。稍后我们将会介绍这种风格的配置。

织入 SPNEGO 到 security 命名空间中

现在我们需要将已经配置的 bean 织入到 security 命名空间配置元素中。首先，我们需要添加对我们新 AuthenticationEntryPoint 的引用（很像我们在第十章的 CAS 配置），如下：

```
<http ...
entry-point-ref="kerbEntryPoint">
```

我们需要将 SPNEGO 过滤器插入到 Spring Security 过滤器链中。一个适当位置应该是用 SPNEGO 过滤器来取代 form 登录过滤器，如下：

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

最后，我们需要一个<authentication-provider>引用新的 AuthenticationProvider 来负责处理 SPNEGO tickets。现在我们添加它：

```
<authentication-manager alias="authenticationManager">
  <authentication-provider ref="kerberosServiceAuthenticationProvider"/>
</authentication-manager>
```

这是为我们应用使用 SPNEGO SSO 安全所需要的额外配置。

如果你的机器已经正确配置 Kerberos 认证，现在你可以重启 web 应用并在域中的另一台机器尝试进行认证。

首先来尝试使用 IE，IE 几乎不需要任何配置来使得用户手动响应 SPNEGO 请求。你在访

访问受保护页面如“*My Account*”页面时会看到一个类似于下面的弹出框：



祝贺你——如果你看到了这个弹出提示，在你提供正确的凭证后就能访问“*My Account*”页面了，你已经为你的 web 应用成功配置了 Kerberos 认证。

接下来是配置 IE 基于请求传递用户的网络凭证。必须设置如下的配置 IE 才能透明的响应 SPNEGO SSO 请求：

- 确保站点添加到“本地 Intranet”安全区域中。你可以使用“Internet 选项”中的“安全” tab 标签手动添加你的站点到安全区域中；
- 通过检查“启用保护模式”复选框来确保对于 intranet 区域启用了 IE 保护模式；
- 在“高级” tab 标签中，确保“启动集成 windows 验证”复选框是选中的。

通过以上明确指明以上的配置（并将 IE 重启），你应该看到登录用户的凭证会通过 SPNEGO 认证自动发送到站点上，用户马上就会登录成功。

添加应用服务器所在机器到 Kerberos 域中

如果你是一台新的机器上工作还有最后一步——配置系统范围内的 Kerberos 配置，以使得 Sun GSS-API 能够确定到哪里寻找对域中用户进行认证的 Kerberos KDC。这需要创建一个 krb5.ini 文件，它通常放在 Windows 安装目录下（c:\Windows 或等价目录）。

因为这个文件的配置语法超出了本书的范围，以下的基本配置在单个域的环境中就足够了，其中 KDC 位于 corp.jbcppets.com。

```
[domain_realm]
.jbcppets.com = CORP.JBCPPETS.COM
[libdefaults]
default_realm = CORP.JBCPPETS.COM
[logging]
[realms]
```

```
CORP.JBCPPETS.COM = {
    kdc = corp.jbcppets.com
}
```

作为 krb5.ini 文件的替代方式，也可以使用 JVM 的属性来声明默认的域和 KDC，如下表：

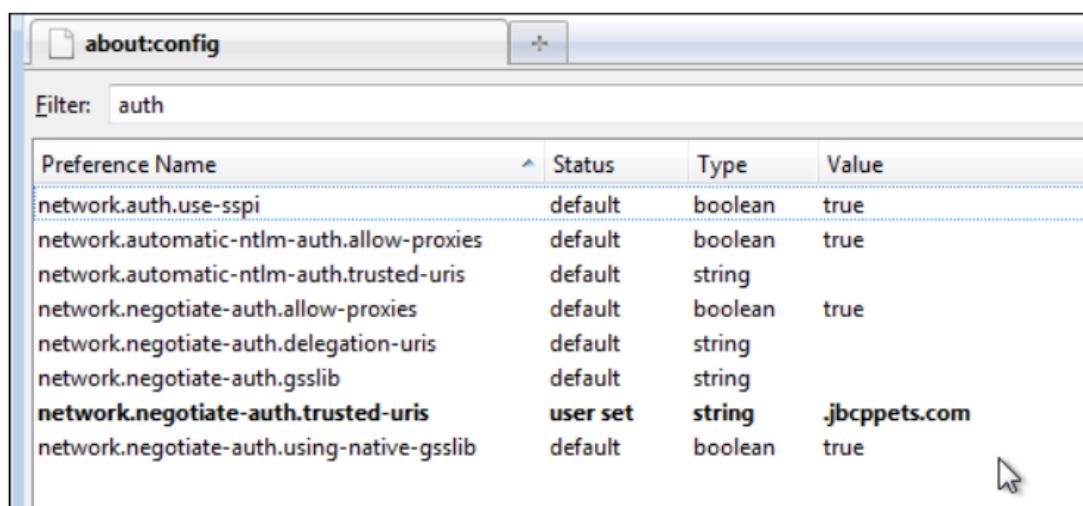
参数	描述	例子
-Djava.security.krb5.realm	默认域	CORP.JBCPPETS.COM
-Djava.security.krb5.kdc	默认 KDC	corp.jbcppets.com

一般来说，我们建议使用 krb5.ini，因为这个文件也可以被其它启用 Kerberos 的应用所使用。

对 Firefox 用户的特殊考虑

默认情况下，Firefox 不支持 Kerberos 认证。这有其历史原因（有些人说这增加了安全性），但是不管怎样，默认情况下，当收到 WWW-Authenticate: Negotiate 请求头，firefox 不会提示用户。

Firefox 的用户必须在 about:config 页面中修改一个设置以明确添加凭证自动发送到的域，这通过在 network.negotiate-auth.trusted-uris 参数提供域名，如下面的截图所述：



通过修改默认的（为空）设置，Firefox 会自动的将你的 Windows 域凭证基于请求发送到站点上。这个属性对于不熟悉的用户是隐藏的，如果这进行了修改的话，配置 Firefox 支持 SPNEGO 要比 IE 容易得多。

问题解决

不幸的是，Kerberos 的正确配置特别复杂（特别对于第一次接触的开发者或管理员），而启用 Kerberos 的 web 应用增加了这种复杂性。

使用标准工具测试连接

首先也是最重要的，确认底层的 Kerberos 基础设施能够正确运行。这意味着你应该使用标准的 Kerberos 工具（如 kinit 或 ktab）或者图形化的客户端如 MIT Kerberos for Windows (KfW) 客户端（可在这里获取：<http://web.mit.edu/Kerberos/>）来进行校验。

kinit 和 ktab 是标准 JDK 发布版的一部分——ktab 的实例命令行如下

```
ktab -a kerbuser@CORP.JBCPPETS.COM -k kerbuser.keytab  
Password for kerbuser@CORP.JBCPPETS.COM:xxxxx Done!  
Service key for kerbuser@CORP.JBCPPETS.COM is saved in kerbuser.keytab
```

MIT Kerberos 实现提供了图形化的登录和配置界面，这对于新用户来说可能会更容易掌握。

启用 Java GSS-API 调试

在 Java GSS-API 中没有太多的日志，而这个库在通过 Kerberos 进行认证时做了大量的工作，所以如果能够得到其内部如何工作的将会很有用处。你可以设置 JVM 属性 -Dsun.security.krb5.debug=true 或者设置 SunJaasKerberosTicketValidator bean 的 debug 属性如下：

```
<bean id="ticketValidator"  
      class="org.springframework.security.extensions.kerberos.SunJaasKerberosTicketValidator">  
    <property name="servicePrincipal" value="HTTP/web.jbcppets.com@corp.jbcppets.com" />  
    <property name="keyTabLocation" value="classpath:website.keytab"/>  
    <property name="debug" value="true"/>  
</bean>
```

这两种方式都会在失败的情况下在应用服务器的控制台打印出一些有用的信息。

其它问题解决步骤

在使用 Spring Security Kerberos Extension 以 kerberizing（将 Kerberos 用到）你的 web 应用时，解决常见问题的一些建议如下：

- 确保对 Spring Security 设置标准级别的日志，对 the org.springframework.security.extensions.kerberos 包至少是 WARN 级别或更高。如果应用服务器存在配置问题或 ticket 校验问题，它们一般会在 Spring 日志中看到；
- 使用网络监控工具或抓包工具来监控应用服务器和 KDC 之间的通信。这种工具在监控客户端浏览器和应用服务器时，也会很有用处；
- 如果你使用的是 Windows，不要在运行服务器的机器上使用浏览器来进行测试——它将不会好用。这是因为 Windows 将会自动使用 NTLM 认证。NTLM 认证使用相似的 request/response 头交换，但是使用不同的数据格式。校验这个问题的一个方式就是查看客户端浏览器返回 Negotiate 头这部分的 Spring 日志。如果这个头信息以 TIRM 开头，它就是一个 NTLM 头而不是 SPNEGO。SPNEGO 头会以 YII 开头并且很长；
- 确保你以域用户而不是本地用户登录 Windows，否则 Windows 将会尝试进行 NTLM 认证；

- 确保 DNS 方案对于 Kerberos 认证过程所有参与者都能运行，包括服务端和客户端机器。

即使有了这些解决问题的步骤，可能也会发现成功部署 SPNEGO 和 Kerberos 也很有挑战——不要放弃，你会成功的！

配置 LDAP UserDetailsService 使用 Kerberos

简便起见，我们配置了 JDBC UserDetailsService 并硬编码 Kerberos 用户 ID。通常，在配置 Kerberos 认证（尤其是使用 Active Directory），用户的细节信息是通过 LDAP 从 AD 中得到的。我们将了解使用 Kerberos 认证时，与 Microsoft AD 比较的常见的 LDAP UserDetailsService 配置。我们将会略过对 Spring Security 与 LDAP 集成的描述，建议你跳回到第九章：LDAP 目录服务中，在那里我们详细介绍过 LDAP 并包括关于通过 LDAP 集成 AD 的特别建议。

我们可以配置认证提供者（authentication provider）引用 LDAP UserDetailsService，而后者定义在 dogstore-security.xml 文件中，如下：

```
<ldap-server url="ldaps://corp.jbcppets.com/DC=corp,DC=jbcppets,DC=com" id="ldapCorp"
manager-dn="CN=Administrator,CN=Users,DC=corp,DC=jbcppets,DC=com"
manager-password="apassword!"/>
<ldap-user-service id="ldapUserService" server-ref="ldapCorp"
user-search-filter="(userPrincipalName={0})" user-search-base="CN=Users"
group-search-base="CN=Groups"/>
```

这里的关键是 user-search-filter，它被 userPrincipalName LDAP 属性所搜索——这与 SPNEGO 认证过程中提供的 Kerberos 安全实体名字相匹配。注意，你要提供一个 manager-dn，它只是有访问 AD 的权限而实际上并不是域的管理员。

接下来，简单调整 KerberosServiceAuthenticationProvider 引用 UserDetailsService，在 dogstore-base.xml 文件中，如下：

```
<bean id="kerberosServiceAuthenticationProvider"
class="org.springframework.security.extensions.kerberos.KerberosServiceAuthenticationProvider"
>
<property name="ticketValidator" ref="ticketValidator"/>
<property name="userDetailsService" ref="ldapUserService" />
</bean>
```

配置技术中唯一要说明的是给认证过的用户匹配权限。回忆一下第九章中，Spring Security LDAP 期望用户权限能以一种特殊的方式在 LDAP 中查询出来——如果你 AD 的设置与 LDAP 内置的匹配不相容，你可能需要写自己的 LdapAuthoritiesPopulator 实现。如果你需要这方面的指导，请查询这个接口的实现类作为指导和起点。

还要注意，Active Directory 本身可能很复杂并很难在任何情况下直接匹配。你可能需要与试图集成 Active Directory 的 IT 人员进行交流，以确定业务需求和合适的应用模型以满足你的用户和业务需要。

使用 Kerberos 与 form 登录

尽管使用 SPNEGO 为浏览器提供 SSO 是集成 Kerberos 和 Spring Security 的常见驱动力，但是你可能希望只是将 Kerberos 登录本身作为一个 AuthenticationProvider 机制(类似于 LDAP 的绑定认证方式——参考第九章以了解细节)。

以这种方式配置 Kerberos 的好处是我们可以同时支持 Kerberos 认证用户和使用其它的 AuthenticationProviders (LDAP, JDBC 等等) 来进行用户认证。

【如果你想尝试这个练习，确保移除在本章第一节中的配置——最重要的是，SpnegoEntryPoint 必须移除以使得用户可以被重定向到登录 form，否则没有 SPNEGO 的用户将会被拒绝访问。】

现在，让我们了解一下配置步骤。在 dogstore-base.xml 中，要配置的 Spring Bean 实际上与 SPNEGO 风格登录的 bean 没有任何的重叠之处，所以你尽可以同时对其进行配置（尽管在实际中，你不会很容易地同时拥有 SPNEGO 和 form 登录，所以你可能不会这么做）。

需要如下的 bean：

```
<bean id="kerberosAuthenticationProvider"
    class="org.springframework.security.extensions.kerberos.KerberosAuthenticationProvider">
    <property name="kerberosClient" ref="kerbJaasClient"/>
    <property name="userDetailsService" ref="jdbcUserServiceCustom"/>
</bean>
<bean id="kerbJaasClient"
    class="org.springframework.security.extensions.kerberos.SunJaasKerberosClient">
    <property name="debug" value="true"/>
</bean>
<bean id="kerbGlobalJaasConfig"
    class="org.springframework.security.extensions.kerberos.GlobalSunJaasKerberosConfig">
    <property name="debug" value="true"/>
    <property name="krbConfLocation" value="/path/to/krb5.conf" />
</bean>
```

在上面我们强调的配置是 krbConfLocation 属性，它指向了一个 Kerberos V5 文件。这个文件可以像前面章节提供的 krb5.ini 文件那样格式化（如果你想知道这个文件的内容，请查阅相关章节的 Kerberos V5 文档页面<http://web.mit.edu/kerberos/krb5-1.5/krb5-1.5.1/doc/krb5-admin/krb5.conf.html>）。

【请注意，不同于我们前面引用 Kerberos keytab 文件，这个参数的值必须是文件系统中的物理绝对路径名（如 c:\spring\krb5.conf），并不是 Spring 风格的 file:或 classpath:引用。这是因为这个文件路径是直接传递给 Sun JVM 的 Kerberos 子系统，并不会被 Spring bean 来解释，如 SunJaasKerberosTicketValidator 所作那样。同样的，要注意的是这种风格的配置会修改 JVM 属性设置，所以可能会影响到部署在同一个 JVM 的其它启用 Kerberos 的应用。】

配置文件中列出的 default_realm 用来认证通过用户界面登录的用户，他们没有提供域名。用户也可以在用户名带上域名（如 kerbuser@jbcppets.com），通常的 Kerberos KDC 方案规则都会基于 Kerberos 配置。

因为这种风格的 Kerberos 配置是用来支持基于 form 的登录，我们只需配置像配置其它 AuthenticationProvider 那样配置 KerberosAuthenticationProvider，这在 security 命名空间的配置文件 dogstore-security.xml 里，如下：

```
<authentication-manager alias="authenticationManager">
    <authentication-provider ref="kerberosAuthenticationProvider"/>
</authentication-manager>
```

在这些配置修改完成后，你可以重启应用并使用 form 来登录使用 Kerberos 的后台应用。记住你也需要一个 `UserDetailsService` 实现来处理 `GrantedAuthority` 与用户的匹配。记住你可以将这种风格的认证用在其它的 `AuthenticationProvider` 后台认证技术上，包括 `basic` 认证。

小结

在本章中，我们学习了怎样构建 Kerberos 环境，如 Windows Domain 提供的 Microsoft Active Directory，来提供对 Windows 操作系统用户的集成单点登录。这为用户提供了统一且用户体验良好的（登录方式），并且减轻了系统管理员的负担。在本章中，我们：

- 学习 Kerberos SPNEGO 认证协议重要元素的整体状况；
- 学习了启用 Kerberos 的 web 应用所需要的设施和重要步骤；
- 配置 JBCP Pets 来支持 Kerberos 后台的 SPNEGO 单点登录认证；
- 了解启用 Kerberos web 应用的常见问题解决办法；
- 学习了使用 AD 作为 LDAP 后台存储用户信息所需要的配置；
- 学习了如何配置组合使用基于 form 的登录以及 Kerberos 后台系统。

下一章也是最后一章将会涵盖从较早版本的 Spring Security 进行迁移。我们希望你能喜欢。

第十三章 迁移到 Spring Security 3

在最后一章中，我们将会了解从 Spring Security2 迁移到 Spring Security3 时常见问题的相关情况。

在本章中，我们将会：

- 了解 Spring Security 3 的重要增强；
- 理解已有的 Spring Security 2 应用迁移到 Spring Security 3 时所需要的配置修改；
- 阐述 Spring Security 3 中重要类和包的移动。

一旦你学完本章的内容，你将会很熟悉怎样将 Spring Security 2 应用迁移到 Spring Security 3 上。

从 Spring Security2 进行迁移

你可能计划将现存的 Spring Security 2 应用迁移到 Spring Security 3，或者试图为 Spring Security 2 应用添加功能并想在本书中寻找一些参考。我们将会在本章中尽可能解决你所关注的这两个问题。

首先，我们将会了解 Spring Security 2 和 Spring Security 3 的重要区别——包括功能和配置。其次，我们将会对配置匹配和类名变化提供一些参考。你可以将本书中的例子从 Spring Security 3 转换到 Spring Security 2（这是可行的）。

一个重要的迁移注意事项是 Spring Security 3 需要 Spring 框架 3 和 Java5 (1.5) 或更高。注意的是，在很多情况下，迁移其它的组件比升级 Spring Security 会对你的应用产生更大的影响。

Spring Security3 的增强

Spring Security 3 比 Spring Security 2 有了很重要的增强，包括：

- 增加了 Spring 表达式语言（Spring Expression Language, SpEL）对访问声明的支持，包括 URL 和方法访问声明，我们在第二章：Spring Security 起步和第五章：精确的访问控制中已经介绍了；
- 添加了对认证和访问成功及失败添加了精确的配置，我们在第二章，第五章进行了简单介绍，而在第六章：高级配置和扩展中进行了详细介绍；
- 增强的方法访问声明，包括基于注解的调用事先和事后访问检查和过滤，以及 security 命名空间配置对自定义 bean 行为的良好支持。这些功能我们在第五章中进行了介绍；
- 使用 security 命名空间对 session 访问和并发控制进行精确管理，这些在第六章中进行了介绍；
- 值得一提的是 ACL 模块，移除了 o.s.s.acl 中的遗留代码并解决了 ACL 框架中一些重要问题。ACL 的配置和支持在第七章：访问控制列表中进行了介绍；
- 支持 OpenID 属性交换，以及对 OpenID 健壮性的其它增强，这在第八章：对 OpenID 开放中进行了介绍；
- 通过 Spring Security Extension 项目对 Kerberos 和 SAML 提供了新的支持，这些我们在第十二章：Spring Security 扩展中进行了讨论。

其它重要的良好变化包括对代码和框架的配置进行了重构和清理，所以整体结构和用法都有所变化。Spring Security 的作者努力增加以前所没有的可扩展性，尤其是在登录和 URL 重定向方面。

如果你已经在 Spring Security2 环境下工作，如果不是遇到该框架功能的边界你可能找不到强制升级的理由。但是，如果你发现 Spring Security2 在可用扩展点、代码结构或可配置性方面的限制，欢迎阅读我们在本章剩余部分详细讲到的细微变化。

Spring Security 配置的变化

Spring Security 3 的很多变化在 security 命名空间风格的配置。尽管本章不能详细涵盖所有的细微变化，但是我们尽力包含迁移至 Spring Security3 时会影响到你的变化。

AuthenticationManager 配置的重新组织

Spring Security 3 最重要的变化在于 AuthenticationManager 的配置和 AuthenticationProvider 相关的元素。在 Spring Security 2 中，AuthenticationManager 和 AuthenticationProvider 的配置是完全不相关的——声明一个 AuthenticationProvider 不需要任何 AuthenticationManager 的概念。

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

在 Spring Security 2 中，声明`<authentication-manager>`元素是作为 AuthenticationProvider 的兄弟节点的。

```
<authentication-manager alias="authManager">
<authentication-provider>
    <jdbc-user-service data-source-ref="dataSource"/>
</authentication-provider>
<ldap-authentication-provider server-ref="ldap://localhost:10389"/>
```

在 Spring Security 3 中，所有的 AuthenticationProvider 元素必须是`<authentication-manager>`的子节点，所以应该重写为如下格式：

```
<authentication-manager alias="authManager">
    <authentication-provider>
        <jdbc-user-service data-source-ref="dataSource"/>
    </authentication-provider>
    <ldap-authentication-provider server-ref=
        "ldap://localhost:10389"/>
</authentication-manager>
```

当然这意味着现在`<authentication-manager>`元素需要在任何 security 命名空间配置中都要存在。

在 Spring Security 2 中，如果你有自定义的 AuthenticationProvider，你应该在其 bean 声明中用`<custom-authentication-provider>`元素来进行包装，例如，我们在第六章中实现的自定义 AuthenticationProvider：

```
<bean id="signedRequestAuthenticationProvider"
      class="com.packtpub.springsecurity.security .SignedUsernamePasswordAuthenticationProvider">
    <security:custom-authentication-provider/>
    <property name="userDetailsService" ref="userDetailsService"/>
<!-- ... -->
</bean>
```

但是将这个自定义 AuthenticationProvider 迁移到 Spring Security 3 时，我们需要移除包装元素并使用`<authentication-provider>`元素的`ref`属性来配置 AuthenticationProvider，就像我们在第三章中所看到的，如下：

```
<authentication-manager alias="authenticationManager">
  <authentication-provider ref= "signedRequestAuthenticationProvider"/>
</authentication-manager>
```

当然，自定义 provider 的源码会因为 Spring Security 3 的类更换位置和改名而有所变化——在本章后面会有基本介绍而在本章的源码中会有更为细节的匹配关系。

Session 管理选项的新配置语法

除了继续支持框架前面版本的 session 固化和并发控制功能，Spring Security 3 为自定义 URL 以及 session 和并发控制功能相关的类添加了新的配置功能，我们在第六章中进行了详细介绍。如果你的旧应用配置了 session 固化和并发 session 控制，这些配置有了新的位置，即在`<http>`中的`<session-management>`指令中。

在 Spring Security 2 中，这些选项将会配置如下：

```
<http ... session-fixation-protection="none">
<!-- ... -->
  <concurrent-session-control exception-if-maximum-exceeded="true" max-sessions="1"/>
</http>
```

在 Spring Security 3 的配置语法中，从`<http>`元素中移除了`session-fixation-protection`属性，配置如下：

```
<http ...>
  <session-management session-fixation-protection="none">
    <concurrency-control error-if-maximum-exceeded="true" max-sessions="1"/>
  </session-management>
</http>
```

你可以看到，这些选项的新的逻辑组织更为合理并为进一步的扩展留下了空间。

自定义过滤器配置的变化

很多的 Spring Security 2 用户开发过自定义的认证过滤器（或其它过滤器来改变安全请求的流程）。如同自定义的认证 provider，以前这些过滤器也是通过带有`<custom-filter>`元素的 bean 来声明的。这使得在一些场景下将过滤器直接配置到 Spring Security 配置中有点困难。

在 Spring Security 2 的环境下，让我们看一下第六章中签名请求头过滤器的配置示例。

```
<bean id="requestHeaderFilter"
  class="com.packtpub.springsecurity.security.RequestHeaderProcessingFilter">
  <security:custom-filter after="AUTHENTICATION_PROCESSING_FILTER"/>
  <property name="authenticationManager"
    ref="authenticationManager"/>
</bean>
```

将其与 Spring Security 3 相同的配置进行对比，你可以看到 bean 的声明和安全织入是独立完成的。自定义的过滤器在<http>元素中声明，如下：

```
<http ...>
<!-- ... -->
  <custom-filter ref="requestHeaderFilter"
    before="FORM_LOGIN_FILTER"/>
<!-- ... -->
</http>
```

尽管 bean 的声明与 Spring Security 2 保持相同，但是你可能会预料到，自定义过滤器的代码差别很大。我们在本章包含了过滤器的示例代码（使用 Spring Security 2），为你了解自定义过滤器在转换成 Spring Security 3 之前和之后是什么样子提供指导。

另外，一些过滤器的逻辑名在 Spring Security 3 中发生了变化。我们在下面提供了一个变化列表而在附录：参考材料中提供了完整的列表。

Spring Security 2	Spring Security 3
SESSION_CONTEXT_INTEGRATION_FILTER	SECURITY_CONTEXT_FILTER
CAS_PROCESSING_FILTER	CAS_FILTER
AUTHENTICATION_PROCESSING_FILTER	FORM_LOGIN_FILTER
OPENID_PROCESSING_FILTER	OPENID_FILTER
BASIC_PROCESSING_FILTER	BASIC_AUTH_FILTER
NTLM_FILTER	在 Spring Security 中移除了

你在定位<custom-filter>元素时，在你的配置文件中必须进行这些修改。

CustomAfterInvocationProvider 的变化

Spring Security 2 中的最后一个 bean 包装被直接、内联的元素声明所取代了，这就是<custom-after-invocation-provider>元素声明的 CustomAfterInvocationProvider。

```
<bean id="customAfterInvocationProvider"
  class="com.packtpub.springsecurity.security.CustomAfterInvocationProvider">
  <security:custom-after-invocation-provider/>
</bean>
```

类似于我们在前面看到其它 Spring Security 2 bean 包装，在 Spring Security 3 中，这个元素被移到<global-method-security>声明中，只会有一个简单的 bean 引用。

```
<global-method-security ...>
  <after-invocation-provider ref="customAfterInvocationProvider"/>
</global-method-security>
```

我们已经在第五章中介绍了方法安全的各个方面，包括在 Spring Security 3 中新增的一些关于方法安全的有趣选项。

小的配置变化

以下简单介绍了 Spring Security 2 和 3 之间的其它配置属性变化：

- 在 Spring Security 3 中使用 auto-config 属性，将不会默认配置 remember me 服务。你需要在<http>元素中明确添加<remember-me>声明；
- 对于 LDAP 配置，在 Spring Security 3 中 group-search-base-attribute 的默认值(用来进行 LDAP 权限查找)从 ou=Groups 变成了空字符串 (LDAP 的根)。我们在第九章：LDAP 目录服务中用过该属性；
- 在 Spring Security 3 中，用来手动配置过滤器链的<filter-invocation-definition-source> 包装元素被重命名为<filter-security-metadata-source>。我们在第六章使用明确 bean 配置的时候用过这种类型的设置；
- 在 Spring Security 3 中，<concurrent-session-control> 元素相关的 exception-if-maximum-exceeded 属性被移到并重命名为<concurrency-control> 元素的 error-if-maximum-exceeded 属性；
- 在 Spring Security 3 中，当使用内存 DAO UserDetailsService 在配置文件中声明用户时，password 属性不再是必须的；
- 内置的 NTLM 认证支持已经在 Spring Security 3 中移除了，而是用 Kerberos 认证进行了替换（请查阅第十二章了解配置 Kerberos 的细节）。这是升级用户比较关注的事情，而在 Spring Security 社区中有一些活动想让 Spring Security 3 支持 NTLM。也许在本书出版的时候，可能会有一个 Spring Security 扩展项目支持 Spring Security 3 中的 NTLM。

Spring Security 3 中剩余的 security 命名空间 XML 配置语法变化为功能的增加，并不会对已有应用带来迁移的问题。

包和类的变化

尽管在比较简单的 Spring Security 2 应用中，类在包中的位置关系不大，但是大多数的 Spring Security 应用不会与底层的代码无关。所以，我们感觉为你指出 Spring Security 2 和 3 之间的总体的包迁移和类重命名会有所用处。

不管在什么地方，我们都试图尽可能精确的匹配类——在这里我们提供了整体的主要包迁移，并且（如果你需要）更复杂的列表可以随源码一起下载。下表展现了 Spring Security 2 到 Spring Security 3 最大的类位置变更——我们已经压缩了这个表以使其包含大多数的你可能希望看到的变化：

类的数量	Spring 2 中的位置	Spring 3 中的位置
13	o.s.s	o.s.s.authentication
13	o.s.s.acls	o.s.s.acls.model
13	o.s.s.event.authentication	o.s.s.authentication.event
12	o.s.s.vote	o.s.s.access.vote
11	o.s.s.ui.rememberme	o.s.s.web.authentication.rememberme
10	o.s.s.providers.jaas	o.s.s.authentication.jaas
10	o.s.s.securechannel	o.s.s.web.access.channel

10	o.s.s.userdetails.ldap	o.s.s.ldap.userdetails
9	o.s.s.providers.encoding	o.s.s.authentication.encoding
8	o.s.s.config	o.s.s.config.authentication
8	o.s.s.util	o.s.s.web.util
7	o.s.s.config	o.s.s.config.http
7	o.s.s.context	o.s.s.core.context
7	o.s.s.userdetails	o.s.s.core.userdetails
6	o.s.s	o.s.s.access
6	o.s.s.afterinvocation	o.s.s.acls.afterinvocation
6	o.s.s.event.authorization	o.s.s.access.event
6	o.s.s.util	o.s.s.web
5	o.s.s.annotation	o.s.s.access.annotation
5	o.s.s.authoritymapping	o.s.s.core.authority.mapping
5	o.s.s.providers	o.s.s.authentication
5	o.s.s.token	o.s.s.core.token
5	o.s.s.ui	o.s.s.web.authentication

如果你发现类移动相当大，你是正确的！在 Spring Security 3 包的重新组织中，很少有类不被接触到。希望这个整体的介绍能够在你寻找类的时候为你指明方向。再一次强调，请查询本章的下载内容来了解更详细的类和类之间的匹配。

对整个框架重新组织的好处在于现在更加模块化，并将 JAR 文件分割为特定功能的元素，介绍如下（我们使用 nnn 待代替释放版本号）：

JAR 名	功能
spring-security-acl-nnn.jar	支持 ACL（见第七章）
spring-security-cas-client-nnn.jar	支持 CAS（见第十章）
spring-security-config-nnn.jar	整体配置支持
spring-security-core-nnn.jar	核心框架和类
spring-security-ldap-nnn.jar	支持 LDAP（见第九章）
spring-security-openid-nnn.jar	支持 OpenID（见第八章）
spring-security-taglibs-nnn.jar	支持 JSP 标签库（见第三、五、七章）
spring-security-web-nnn.jar	支持 web 层

模块化意味着，例如，可以部署 Spring Security 到一个非 web 的应用而不需要任何 web 相关的依赖（spring-security-config 和 spring-security-core 可能能够满足你的需求）。

小结

本章中我们了解了将已有的 Spring Security 2 项目升级到 Spring Security 3 会遇到的大小变化。在本章中，我们：

- 了解了我们将要升级的框架所拥有的明显功能增强；
- 学习了可能阻碍升级的需求、依赖以及常见的代码和配置变化；
- 调查了（整体上）Spring Security 的作者在代码重构时代码重新组织的变化。

如果这是你阅读的第一章，我们希望你能转向本书的其它部分，将本章作为平滑升级到 Spring Security 3 的一个指导。

附录：参考材料

在本附录中，将会涉及到一些我们感觉有用的参考材料（并相当缺乏文档），而将其插入到章节的内容中又会觉得过于综合。

JBCP Pets 示例代码起步

就像我们在第一章：一个不安全应用的剖析中所描述的那样，我们假设你已有了 Eclipse 3.4（或 3.5）IDE，并包含 Web Tools Package（WTP）。示例代码按每章被组织成了不同的 ZIP 文件，并有一个较大的 ZIP 文件其中包含了编译和运行示例应用所需的所有依赖（注意的是，当你阅读本书的时候对应于最新版本的 Spring Security 它们可能已经较旧了，但是鉴于示例代码和依赖是一个静态的快照，它们能够永远运行）。

我们建议你对每一章建立新的 Eclipse 工作空间，这样你能够切换工作空间而不用打开或关闭项目。

以下的步骤帮助你建立新的工作空间。首先，我们要导入 Dependencies 项目到工作空间中：

- 选择“File”按钮，接下来是“Import...”选项。选择“General”文件夹以及“Existing Projects into Workspace”并点击“Next”；
- 确保“Select root directory”选项被选中，点击“Browse...”按钮到文本框。定位到你解压 Dependencies.zip 的目录并点击“OK”；
- 你应该可以看到 Dependencies 项目被列了出来。点击“Finish”。

接下来，我们需要导入每章的源码 ZIP 文件。假设你已经解压了第二章：Spring Security 起步的源码到目录中。

- 选择“File”菜单以及“Import...”选项。选择“General”文件夹以及“Existing Projects into Workspace”。点击“Next”；
- 确保“Select root directory”选项被选中并点击“Browse...”按钮到文本框。定位到你解压第二章源码 ZIP 文件的目录并点击“OK”；
- 你可以看到“JBCPPets”项目和“Servers”项目被裂了出来。将它们都选中然后点击“Finish”。

最后，我们需要将 JBCP Pets web 应用部署到一个 Tomcat 实例上（或者你喜欢的应用服务器）。

- 右键点击“JBCPPets”项目并选择“Run As”菜单。在子菜单中选择“Run on Server”菜单项；
- 这时候你可能需要创建一个新的应用服务器实例。只需要按照你应用服务器的提示直到 web 应用部署完成。

此时，你应该可以运行 JBCP Pets 应用了。如果你遇到问题，可查看如下的列表：

- Eclipse 是否列出了构建错误？如果有任何的 Java 错误或 classpath 错误，它们是应该被解决的真正错误。有时候，Spring IDE 插件会报出假的错误或警告；
- 查看 Eclipse 的应用服务器启动控制台，这里包含不成功部署的 web 应用的错误。

最常见的问题是缺失 classpath 条目，或者忘记将所有的 classpath 条目添加到 JBCPPets 项目的“Java EE Module Dependencies”。尽管这些应该已经为你做好，但是我们不能保证在任何版本的 Eclipse 下均好用。

对于示例代码有任何问题请联系我们——理解代码和它的概念对你来说很重要！

可用的应用事件

下面的表格，在第六章：高级配置和扩展曾经提到，列出了各种 Spring Security 元素所发布的全部事件并提供了第二章中引用到的认证异常。为了简便，我们移除了包名，因为所有的时间都在 o.s.s.authentication.event（认证相关的事件）和 o.s.s.access.event（授权相关的事件）中。

类名	何时触发	匹配的异常
AbstractAuthenticationEvent	所有认证时间的通用父类。注意这是一个从来不会被抛出的抽象异常（尽管其可以被捕获）	
AbstractAuthenticationFailureEvent	所有认证失败事件的通用父类。注意这是一个从来不会被抛出的抽象异常（尽管其可以被捕获）	
AuthenticationFailureBadCredentialsEvent	当提供的凭证（如用户名和密码）不合法时。它能够用来（有意的）掩盖 UsernameNotFoundException。	BadCredentialsException UsernameNotFoundException
AuthenticationFailureConcurrentLoginEvent	当并发 session 最大值超出时。	ConcurrentLoginException
AuthenticationFailureCredentialsExpiredEvent	当 UserDetails 标识用户凭证过期时。	CredentialsExpiredException
AuthenticationFailureDisabledEvent	当 UserDetails 标识用户凭证不可用时。	DisabledException
AuthenticationFailureExpiredEvent	当 UserDetails 标识用户账号过期时。	AccountExpiredException
AuthenticationFailureLockedEvent	当 UserDetails 标识用户账号被锁定时。	LockedException
AuthenticationFailureProviderNotFoundEvent	配置错误，当不能找到 Authentication Provider 认证用户请求时。	ProviderNotFoundException
AuthenticationFailureProxyUntrustedEvent	当 CAS 代理 ticket 不可信时。	
AuthenticationFailureServiceExceptionEvent	当底层服务（如 DAO Provider）失败时，抛出的一般异常。	AuthenticationServiceException
AuthenticationSuccessEvent	当用户成功认证时。	
AuthenticationSwitchUserEvent	当成功完成用户切换行为时。	
InteractiveAuthenticationSuccessEvent	当用户通过提供完整凭证认证成功时（类似于 IS_FULLY_AUTHENTICATED GrantedAuthority 语法）	

AbstractAuthorizationEvent	所有认证事件的通用父类。	
AuthenticationCredentialsNotFoundEvent	当用户没有认证而试图触发需要访问检查的方法时。	
AuthorizationFailureEvent	当方法前或后的访问检查失败时。	
AuthorizedEvent	当方法前或后的访问检查成功时。	
PublicInvocationEvent	当未认证的安全对象请求成功时。	
SessionCreationEvent	HttpSession 创建时。	
SessionDestroyedEvent	HttpSession 销毁时。	

Spring Security 的虚拟 URL

以下的 URL 被 Spring Security 视为虚拟 URL，并独立于你的代码作为 servlet 过滤器流程的一部分进行监视（并处理）。记住的是这些 URL 是相对于你的 web 应用上下文根的。

- /j_spring_security_check——被 UsernamePasswordAuthenticationFilter 检查进行用户名/密码 form 认证；
- /j_spring_openid_security_check——被 OpenIDAuthenticationFilter 检查 OpenID 返回认证信息（从 OpenID provider 处）；
- /j_spring_cas_security_check——基于 CAS SSO 登录的返回，进行 CAS 认证
- /spring_security_login —— 当配置自动生成登录页面时，DefaultLoginPageGeneratingFilter 使用的 URL；
- /j_spring_security_logout——LogoutFilter 使用来检测退出行为；
- /saml/SSO——Spring Security SAML SSO extension SAMLProcessingFilter 使用来进行 SAML SSO 登录请求；
- /saml/logout——Spring Security SAML SSO extension SAMLogoutFilter 使用来进行 SAML SSO 退出请求；
- /j_spring_security_switch_user——SwitchUserFilter 使用来将用户切换至另一用户；
- /j_spring_security_exit_user——用来退出切换用户功能。

注意的是，有一些功能在本书中没有涵盖，但是在我们为了完整性全部包括了。

方法安全的明确 bean 配置

第六章源码的 dogstore-explicit-base.xml 文件中包含了这里 bean 声明的全集。我们之所以在第六章本身中没有包含它，是因为它与要表述的没有太大关系（参考第五章：精确的访问控制来了解相关 bean 的功能）。

以下是通过 Spring bean 声明启用方法安全的完整配置：

```
<!-- **** -->
<!-- Method Authorization -->
<!-- **** -->
<bean class="org.springframework.security.access.intercept.
aopalliance.MethodSecurityInterceptor" id="methodSecurityInterceptor">
    <property name="accessDecisionManager" ref="methodAccessDecisionMan
ager"/>
```

```
<property name="authenticationManager" ref="customAuthenticationManager"/>
<property name="securityMetadataSource" ref="delegatingMetadataSource"/>
<property name="afterInvocationManager" ref="afterInvocationManager"/>
</bean>
<bean class="org.springframework.security.access.intercept.aopalliance.MethodSecurityMetadataSourceAdvisor" id="methodSecurityMetadataSourceAdvisor">
<constructor-arg value="methodSecurityInterceptor"/>
<constructor-arg ref="delegatingMetadataSource"/>
</bean>
<bean class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator" id="defaultAdvisorAutoProxyCreator">
<property name="beanName" value="methodSecurityMetadataSourceAdvisor"/>
</bean>
<bean class="org.springframework.security.access.intercept.AfterInvocationProviderManager" id="afterInvocationManager">
<property name="providers">
<list>
<ref local="postAdviceProvider"/>
</list>
</property>
</bean>
<bean class="org.springframework.security.access.vote.AffirmativeBased" id="methodAccessDecisionManager">
<property name="decisionVoters">
<list>
<ref bean="preAdviceVoter"/>
<ref bean="roleVoter"/>
<ref bean="authenticatedVoter"/>
<ref bean="jsr250Voter"/> <!-- For JSR 250 Method Annotations
-->
</list>
</property>
</bean>
<!-- Overall Delegating Metadata Source -->
<bean class="org.springframework.security.access.method.DelegatingMethodSecurityMetadataSource" id="delegatingMetadataSource">
<property name="methodSecurityMetadataSources">
<list>
<ref local="prePostMetadataSource"/>
```

```
<ref local="securedMetadataSource"/>
<ref local="jsr250MetadataSource"/>
</list>
</property>
</bean>
<!-- JSR 250 Method Voters -->
<bean class="org.springframework.security.access.annotation.Jsr250MethodSecurityMetadataSource" id="jsr250MetadataSource"/>
<bean class="org.springframework.security.access.annotation.Jsr250Voter" id="jsr250Voter"/>
<!-- Spring @Secured Beans -->
<bean class="org.springframework.security.access.annotation.SecuredAnnotationSecurityMetadataSource" id="securedMetadataSource"/>
<!-- @Pre/@Post Method Advice Voters -->
<bean class="org.springframework.security.access.prepost.PreInvocationAuthorizationAdviceVoter" id="preAdviceVoter">
    <constructor-arg ref="exprPreInvocationAdvice"/>
</bean>
<bean class="org.springframework.security.access.prepost.PostInvocationAdviceProvider" id="postAdviceProvider">
    <constructor-arg ref="exprPostInvocationAdvice"/>
</bean>
<bean class="org.springframework.security.access.prepost.PrePostAnnotationSecurityMetadataSource" id="prePostMetadataSource">
    <constructor-arg ref="exprAnnotationAttrFactory"/>
</bean>
<!-- @Pre/@Post Method Expression Handler -->
<bean class="org.springframework.security.access.expression.method.DefaultMethodSecurityExpressionHandler" id="methodExprHandler"/>
<bean class="org.springframework.security.access.expression.method.ExpressionBasedPreInvocationAdvice" id="exprPreInvocationAdvice">
    <property name="expressionHandler" ref="methodExprHandler"/>
</bean>
<bean class="org.springframework.security.access.expression.method.ExpressionBasedPostInvocationAdvice" id="exprPostInvocationAdvice">
    <constructor-arg ref="methodExprHandler"/>
</bean>
<bean class="org.springframework.security.access.expression.method.ExpressionBasedAnnotationAttributeFactory" id="exprAnnotationAttrFactory">
    <constructor-arg ref="methodExprHandler"/>
</bean>
```

请注意，明确的 bean 配置与你使用的 Spring Security 版本密切相关（就像我们在第六章提到的）。如果在你版本的 Spring Security 中使用列出的 bean 遇到问题，请参考 `o.s.s.config.method.GlobalMethodSecurityBeanDefinitionParser`。

这个配置启用了 JSR-250 的`@Secured` 和`@Pre/@Post` 注解。如果你不使用它们要注释掉或移除相关的支持 bean（如`@Secured`）。记住，`SecurityMetadataSource` 和 `AccessDecisionVoter` 都要移除。

逻辑过滤器名字迁移参考

正如在第十三章：迁移到 Spring Security 3 所讨论的，很多逻辑过滤器名（在`<custom-filter>`用到）在从 Spring Security 2 升级到 Spring Security 3 时发生了变化。这里我们提供了所有变化，来方便你从 Spring Security 2 到 3 对自定义过滤器的配置：

Spring Security 2	Spring Security 3
CHANNEL_FILTER	CHANNEL_FILTER
CONCURRENT_SESSION_FILTER	CONCURRENT_SESSION_FILTER
SESSION_CONTEXT_INTEGRATION_FILTER	SECURITY_CONTEXT_FILTER
LOGOUT_FILTER	LOGOUT_FILTER
PRE_AUTH_FILTER	PRE_AUTH_FILTER
CAS_PROCESSING_FILTER	CAS_FILTER
AUTHENTICATION_PROCESSING_FILTER	FORM_LOGIN_FILTER
OPENID_PROCESSING_FILTER	OPENID_FILTER
Spring Security 2 没有提供 LOGIN_PAGE_FILTER	LOGIN_PAGE_FILTER
Spring Security 2 没有提供 DIGEST_AUTH_FILTER	DIGEST_AUTH_FILTER
BASIC_PROCESSING_FILTER	BASIC_AUTH_FILTER
Spring Security 2 没有提供 REQUEST_CACHE_FILTER	REQUEST_CACHE_FILTER
SERVLET_API_SUPPORT_FILTER	SERVLET_API_SUPPORT_FILTER
REMEMBER_ME_FILTER	REMEMBER_ME_FILTER
ANONYMOUS_FILTER	ANONYMOUS_FILTER
Spring Security 2 没有提供 SESSION_MANAGEMENT_FILTER	SESSION_MANAGEMENT_FILTER
EXCEPTION_TRANSLATION_FILTER	EXCEPTION_TRANSLATION_FILTER
NTLM_FILTER	Spring Security 3 中移除了 NTLM_FILTER
FILTER_SECURITY_INTERCEPTOR	FILTER_SECURITY_INTERCEPTOR
SWITCH_USER_FILTER	SWITCH_USER_FILTER