

Spring Security

安全权限管理手册

版权 © 2009 Lingo

论坛: <http://www.family168.com/bbs/>。

Email: xyz20003@gmail.com。

QQ群: 3038490。

QQ群: 66496516。

2009-05-26 13:41:40

[序言](#)

[I. 基础篇](#)

- [1. 一个简单的HelloWorld](#)
 - [1.1. 配置过滤器](#)
 - [1.2. 使用命名空间](#)
 - [1.3. 完善整个项目](#)
 - [1.4. 运行示例](#)
- [2. 使用数据库管理用户权限](#)
 - [2.1. 修改配置文件](#)
 - [2.2. 数据库表结构](#)
- [3. 自定义数据库表结构](#)

[3.1. 自定义表结构](#)

[3.2. 初始化数据](#)

[3.3. 获得自定义用户权限信息](#)

[3.3.1. 处理用户登陆](#)

[3.3.2. 检验用户权限](#)

[4. 自定义登陆页面](#)

[4.1. 实现自定义登陆页面](#)

[4.2. 修改配置文件](#)

[4.3. 登陆页面中的参数配置](#)

[4.4. 测试一下](#)

[5. 使用数据库管理资源](#)

[5.1. 数据库表结构](#)

[5.2. 初始化数据](#)

[5.3. 实现从数据库中读取资源信息](#)

[5.3.1. 需要何种数据格式](#)

[5.3.2. 替换原有功能的切入点](#)

[6. 控制用户信息](#)

[6.1. MD5加密](#)

[6.2. 盐值加密](#)

[6.3. 用户信息缓存](#)

[6.4. 获取当前用户信息](#)

[II. 保护web篇](#)

[7. 图解过滤器](#)

[7.1. HttpSessionContextIntegrationFilter](#)

[7.2. LogoutFilter](#)

[7.3. AuthenticationProcessingFilter](#)

[7.4. DefaultLoginPageGeneratingFilter](#)

[7.5. BasicProcessingFilter](#)

[7.6. SecurityContextHolderAwareRequestFilter](#)

[7.7. RememberMeProcessingFilter](#)

[7.8. AnonymousProcessingFilter](#)

[7.9. ExceptionTranslationFilter](#)

[7.10. SessionFixationProtectionFilter](#)

[7.11. FilterSecurityInterceptor](#)

[8. 管理会话](#)

[8.1. 添加监听器](#)

[8.2. 添加过滤器](#)

[8.3. 控制策略](#)

[8.3.1. 后登陆的将先登录的踢出系统](#)

[8.3.2. 后面的用户禁止登陆](#)

[9. 单点登录](#)

[9.1. 配置JA-SIG](#)

[9.2. 配置Spring Security](#)

[9.2.1. 添加依赖](#)

[9.2.2. 修改applicationContext.xml](#)

[9.3. 运行配置了cas的子系统](#)

[9.4. 为cas配置SSL](#)

[9.4.1. 生成密钥](#)

[9.4.2. 为jetty配置SSL](#)

[9.4.3. 为tomcat配置SSL](#)

[10. basic认证](#)

[10.1. 配置basic验证](#)

[10.2. 编程实现basic客户端](#)

[11. 标签库](#)

[11.1. 配置taglib](#)

[11.2. authenticaiton](#)

[11.3. authorize](#)

[11.4. acl/accesscontrollist](#)

[11.5. 为不同用户显示各自的登陆成功页面](#)

[12. 自动登录](#)

[12.1. 默认策略](#)

[12.2. 持久化策略](#)

[13. 匿名登录](#)

[13.1. 配置文件](#)

[13.2. 修改默认用户名](#)

[13.3. 匿名用户的限制](#)

[14. 防御会话伪造](#)

[14.1. 攻击场景](#)

[14.2. 解决会话伪造](#)

[15. 预先认证](#)

[15.1. 为jetty配置Realm](#)

[15.2. 配置Spring Security](#)

[16. 切换用户](#)

[16.1. 配置方式](#)

[16.2. 实例演示](#)

[17. 信道安全](#)

[17.1. 设置信道安全](#)

[17.2. 指定http和https的端口](#)

[18. digest认证](#)

[18.1. 配置digest验证](#)

[18.2. 编程实现basic客户端](#)

[III. 保护method篇](#)

[19. 保护方法调用](#)

[19.1. 控制全局范围的方法权限](#)

[19.2. 控制某个bean内的方法权限](#)

[19.3. 使用annotation控制方法权限](#)

[19.3.1. 使用Secured](#)

[19.3.2. 使用jsr250](#)

[20. 权限管理的基本概念](#)

[20.1. 认证与验证](#)

[20.2. SecurityContext安全上下文](#)

[20.3. Authentication验证对象](#)

[21. Voter表决者](#)

[21.1. Voter表决者](#)

[21.2. 默认角色名称都是以ROLE_ 开头](#)

[22. 拦截器](#)

[IV. ACL篇](#)

[23. Spring Security中的ACL](#)

[23.1. 准备数据库和aclService](#)

[23.1.1. 为acl配置cache](#)

[23.1.2. 配置lookupStrategy](#)

[23.1.3. 配置aclService](#)

[23.2. 使用aclService管理acl信息](#)

[23.3. 使用acl控制delete操作](#)

[23.4. 控制用户可以看到哪些信息](#)

[V. 最佳实践篇](#)

[24. 最简控制台](#)

[24.1. 平台搭建](#)

[24.2. 用户登录](#)

[24.3. 用户信息列表](#)

[24.4. 添加用户](#)

[24.5. 修改用户信息](#)

[24.6. 修改自己的密码](#)

[A. 修改日志](#)

[B. 常见问题解答](#)

[C. Spring Security-3.0.0.M1](#)

[D. 命名空间](#)

[D.1. http](#)

[D.2. authentication-provider](#)

[D.3. ldap-server](#)

[D.4. global-method-security](#)

[E. 数据库表结构](#)

[E.1. User](#)

[E.2. Group](#)

[E.3. RememberMe](#)

[E.4. ACL](#)

[F. 异常](#)

[G. 事件](#)

[下一页](#)

序言

序言

为啥选择Spring Security

欢迎阅读咱们写的Spring Security教程，咱们既不想写一个简单的入门教程，也不想翻译已有的国外教程。咱们这个教程就是建立在咱们自己做的OA的基础上，一点一滴总结出来的经验和教训。

首先必须一提的是，Spring Security出身名门，它是Spring的一个子项目<http://static.springsource.org/spring-security/site/index.html>。它之前有个很响亮的名字Acegi。这个原本坐落在sf.net上的项目，后来终于因为跟spring的紧密连接，在2.0时成为了Spring的一个子项目。

即使是在开源泛滥的Java领域，统一权限管理框架依然是稀缺的，这也是为什么Spring Security (Acegi)已出现就受到热捧的原因，据俺们所知，直到现在也只看到apache社区的jsecurity在做同样的事情。（据小道消息，jsecurity还很稚嫩。）

Spring Security(Acegi)支持一大堆的权限功能，然后它又和Spring这个当今超流行的框架整合的很紧密，所以我们选择它。实际上自从Acegi时代它就很有名了。

内容结构组织

咱们要循序渐进，深入浅出的把整个教程分成几个阶段，一点一点儿慢慢写。反正不用赶稿，从头开始慢慢考虑如何更好的整理自己的思绪不会是一种浪费时间行为。

[第 I 部分 “基础篇”](#)。环境搭建，进行最简单的配置。

[第 II 部分 “保护web篇”](#)。谈谈对url的权限控制。

[第 III 部分 “保护method篇”](#)。对方法调用进行权限控制。

[第 IV 部分 “ACL篇”](#)。实现ACL（Access Control List）。

[第 V 部分 “最佳实践篇”](#)。包含最佳实践，可以当做是OA里权限模块的总结。

意见反馈

咱们的例子都是一一运行过的，文档内容都是好几个人复审过的。但是毕竟百密一疏，没人敢说不会犯错，所以如果同志们在文档或者例子上发现了任何问题，可以通过以下几个途径跟咱们联系。

- 论坛：<http://www.family168.com/bbs/>。
- Email：xyz20003@gmail.com。
- QQ群：3038490。

其实不只是错误，如果对咱们的东西有什么改进意见，或者有什么需要讨论的，不用见外，直接用以上途径找咱们聊天。

相关信息

如果想了解Spring Security或是OA相关的更多信息，请访问我们的网站<http://www.family168.com/>或在论坛<http://www.family168.com/bbs/>中参与相关讨论。

教程相关的实例代码可以从google code上下载：<http://code.google.com/p/family168/downloads/detail?name=springsecurity-sample.rar>。

我们的网站暂时围绕着OA相关的各个技术进行研究，希望大家在这方面对我们提出各种意见。

[上一页](#)

Spring Security

[起始页](#)

[下一页](#)

部分 I. 基础篇

部分 I. 基础篇

在一开始，我们主要谈谈怎么配置Spring Security，怎么使用Spring Security。

为了避免在每个例子中重复包含所有的第三方依赖，要知道Spring.jar就有2M多，所以我们使用了Maven2管理项目。如果你的机器上还没安装Maven2，那么可以参考我们网站提供的Maven2教程<http://www.family168.com/oa/maven2/html/index.html>。

我们用使用的第三方依赖库关系如下所示：

```
[INFO] [dependency:tree]
[INFO] com.family168.springsecuritybook:ch01:war:0.1
[INFO] \- org.springframework.security:spring-security-taglibs:jar:2.0.4:compile
[INFO] +- org.springframework.security:spring-security-core:jar:2.0.4:compile
[INFO] | +- org.springframework:spring-core:jar:2.0.8:compile
[INFO] | +- org.springframework:spring-context:jar:2.0.8:compile
[INFO] | | \- aopalliance:aopalliance:jar:1.0:compile
[INFO] | +- org.springframework:spring-aop:jar:2.0.8:compile
[INFO] | +- org.springframework:spring-support:jar:2.0.8:runtime
[INFO] | +- commons-logging:commons-logging:jar:1.1.1:compile
[INFO] | +- commons-codec:commons-codec:jar:1.3:compile
[INFO] | \- commons-collections:commons-collections:jar:3.2:compile
[INFO] +- org.springframework.security:spring-security-acl:jar:2.0.4:compile
[INFO] | \- org.springframework:spring-jdbc:jar:2.0.8:compile
[INFO] |   \- org.springframework:spring-dao:jar:2.0.8:compile
[INFO] \- org.springframework:spring-web:jar:2.0.8:compile
[INFO]   \- org.springframework:spring-beans:jar:2.0.8:compile
```


序言

[起始页](#)

第 1 章 一个简单的HelloWorld

第 1 章 一个简单的HelloWorld

Spring Security中可以使用Acegi-1.x时代的普通配置方式，也可以使用从2.0时代才出现的命名空间配置方式，实际上这两者实现的功能是完全一致的，只是新的命名空间配置方式可以把原来需要几百行的配置压缩成短短的几十行。我们的教程中都会使用命名空间的方式进行配置，凡事务求最简。

1.1. 配置过滤器

为了在项目中使用Spring Security控制权限，首先要在web.xml中配置过滤器，这样我们就可以控制对这个项目的每个请求了。

```
<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>
```

所有的用户在访问项目之前，都要先通过Spring Security的检测，这从第一时间把没有授权的请求排除在系统之外，保证系统资源的安全。关于过滤器配置的更多讲解可以参考<http://www.family168.com/tutorial/jsp/html/jsp-ch-07.html#jsp-ch-07-03-01>。

1.2. 使用命名空间

在applicationContext.xml中使用Spring Security提供的命名空间进行配置。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/security"❶
  xmlns:beans="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
    http://www.springframework.org/schema/security
    http://www.springframework.org/schema/security/spring-security-2.0.4.xsd">

  <http auto-config='true'>❷
    <intercept-url pattern="/admin.jsp" access="ROLE_ADMIN" />❸
    <intercept-url pattern="/**" access="ROLE_USER" />
  </http>

  <authentication-provider>
    <user-service>
      <user name="admin" password="admin" authorities="ROLE_USER, ROLE_ADMIN" />❹
      <user name="user" password="user" authorities="ROLE_USER" />
    </user-service>
  </authentication-provider>

</beans:beans>
```

- ❶ 声明在xml中使用Spring Security提供的命名空间。
- ❷ http部分配置如何拦截用户请求。auto-config='true'将自动配置几种常用的权限控制机制，包括form, anonymous, rememberMe。
- ❸ 我们利用intercept-url来判断用户需要具有何种权限才能访问对应的url资源，可以在pattern中指定一个特定的url资源，也可以使用通配符指定一组类似的url资源。例子中定义的两个intercept-url，第一个用来控制对/admin.jsp的访问，第二个使用了通配符/**，说明它将控制对系统中所有url资源的访问。

在实际使用中，Spring Security采用的是一种就近原则，就是说当用户访问的url资源满足多个intercept-url时，系统将使用第一个符合条件的intercept-url进行权限控制。在我们这个例子中就是，当用户访问/admin.jsp时，虽然两个intercept-url都满足要求，但因为第一个intercept-url排在上面，所以Spring Security会使用第一个intercept-url中的配置处理对/admin.jsp的请求，也就是说，只有那些拥有了ROLE_ADMIN权限的用户才能访问/admin.jsp。

access指定的权限部分比较有趣，大家可以注意到这些权限标示符都是以ROLE_开头的，实

际上这与Spring Security中的Voter机制有着千丝万缕的联系，只有包含了特定前缀的字符串才会被Spring Security处理。目前来说我们只需要记住这一点就可以了，在教程以后的部分中我们会详细讲解Voter的内容。

- ④ user-service中定义了两个用户，admin和user。为了简便起见，我们使用明文定义了两个用户对应的密码，这只是为了当前演示的方便，之后的例子中我们会使用Spring Security提供的加密方式，避免用户密码被他人窃取。

最最重要的部分是authorities，这里定义了这个用户登陆之后将会拥有的权限，它与上面intercept-url中定义的权限内容一一对应。每个用户可以同时拥有多个权限，例子中的admin用户就拥有ROLE_ADMIN和ROLE_USER两种权限，这使得admin用户在登陆之后可以访问ROLE_ADMIN和ROLE_USER允许访问的所有资源。

与之对应的是，user用户就只拥有ROLE_USER权限，所以他只能访问ROLE_USER允许访问的资源，而不能访问ROLE_ADMIN允许访问的资源。

1.3. 完善整个项目

因为Spring Security是建立在Spring的基础之上的，所以web.xml中除了需要配置我们刚刚提到的过滤器，还要加上加载Spring的相关配置。最终得到的web.xml看起来像是这样：

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4"
  xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:applicationContext*.xml</param-value>
  </context-param>

  <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>
```

```

<listener>
  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>

</web-app>

```

演示不同权限的用户登陆之后可以访问不同的资源，我们为项目添加了两个jsp文件，admin.jsp和index.jsp。其中admin.jsp只有那些拥有ROLE_ADMIN权限的用户才能访问，而index.jsp只允许那些拥有ROLE_USER权限的用户才能访问。

最终我们的整个项目会变成下面这样：

```

+ ch01/
+ src/
+ main/
+ resources/
  * applicationContext.xml
+ webapp/
+ WEB-INF/
  * web.xml
  * admin.jsp
  * index.jsp
+ test/
+ resources/
* pom.xml

```

1.4. 运行示例

首先确保自己安装了Maven2。如果之前没用过Maven2，可以参考我们的Maven2教程<http://www.family168.com/oa/maven2/html/index.html>。

安装好Maven2之后，进入ch01目录，然后执行mvn。

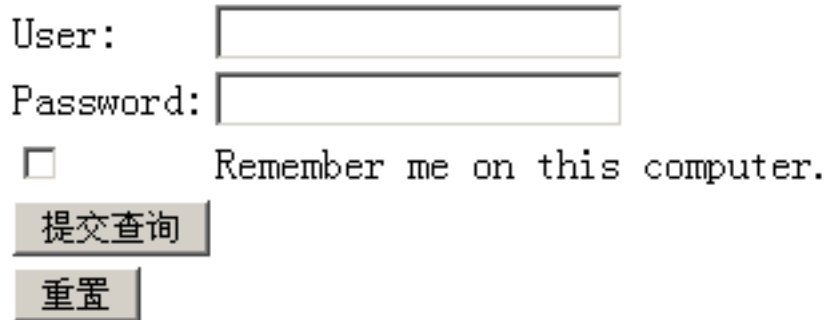
```

信息: Root WebApplicationContext: initialization completed in 1578 ms
2009-05-28 11:37:50.171::INFO: Started SelectChannelConnector@0.0.0.0:8080
[INFO] Started Jetty Server
[INFO] Starting scanner at interval of 10 seconds.

```

等到项目启动完成后。打开浏览器访问<http://localhost:8080/ch01/>就可以看到登陆页面。

Login with Username and Password



User:

Password:

☐ Remember me on this computer.

提交查询

重置

图 1.1. 用户登陆

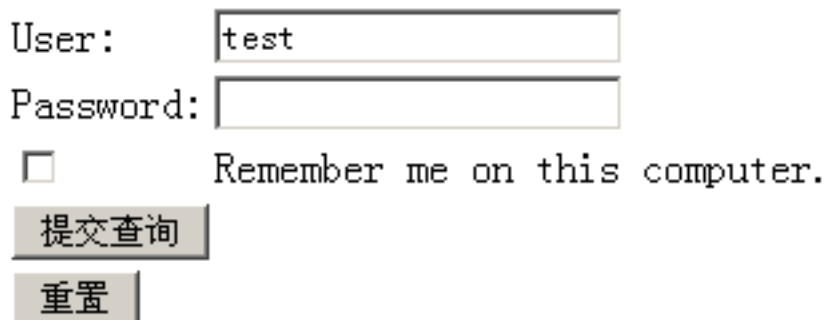
这个简陋的页面是Spring Security自动生成的，一来为了演示的方便，二来避免用户自己编写登陆页面时犯错，Spring Security为了避免可能出现的风险，连测试用的登录页面都自动生成出来了。在这里我们就省去编写登陆页面的步骤，直接使用默认生成的登录页面进行演示吧。

首先让我们输入一个错误用的用户名或密码，这里我们使用test/test，当然这个用户是不存在的，点击提交之后我们会得到这样一个登陆错误提示页面。

Your login attempt was not successful, try again.

Reason: Bad credentials

Login with Username and Password



User:

Password:

☐ Remember me on this computer.

提交查询

重置

图 1.2. 登陆失败

如果输入的是正确的用户名和密码，比如user/user，系统在登陆成功后会默认跳转到index.jsp。

```
username : user
```

[admin.jsp](#) [logout](#)

图 1.3. 登陆成功

这时我们可以点击admin.jsp链接访问admin.jsp，也可以点击logout进行注销。

如果点击了logout，系统会注销当前登陆的用户，然后跳转至登陆页面。如果点击了admin.jsp链接就会显示如下页面。

拒绝访问

图 1.4. 拒绝访问

很遗憾，user用户是无法访问/admin.jsp这个url资源的，这在上面的配置文件中已经有过深入的讨论。我们在这里再简要重复一遍：user用户拥有ROLE_USER权限，但是/admin.jsp资源需要用户拥有ROLE_ADMIN权限才能访问，所以当user用户视图访问被保护的/admin.jsp时，Spring Security会在中途拦截这一请求，返回拒绝访问页面。

为了正常访问admin.jsp，我们需要先点击logout注销当前用户，然后使用admin/admin登陆系统，然后再次点击admin.jsp链接就会显示出admin.jsp中的内容。

```
admin.jsp
```

图 1.5. 显示admin.jsp

根据我们之前的配置，admin用户拥有ROLE_ADMIN和ROLE_USER两个权限，因为他拥有ROLE_USER权限，所以可以访问/index.jsp，因为他拥有ROLE_ADMIN权限，所以他可以访问/admin.jsp。

至此，我们很高兴的宣布，咱们已经正式完成，并运行演示了一个最简单的由Spring Security保护的web系统，下一步我们会深入讨论Spring Security为我们提供的其他保护功能，多姿多彩的特

性。

[上一页](#)

部分 I. 基础篇

[上一级](#)

[起始页](#)

[下一页](#)

第 2 章 使用数据库管理用户权限

第2章 使用数据库管理用户权限

上一章节中，我们把用户信息和权限信息放到了xml文件中，这是为了演示如何使用最小的配置就可以使用Spring Security，而实际开发中，用户信息和权限信息通常是被保存在数据库中的，为此Spring Security提供了通过数据库获得用户权限信息的方式。

2.1. 修改配置文件

为了从数据库中获取用户权限信息，我们所需要的仅仅是修改配置文件中的authentication-provider部分。

将上一章配置文件中的user-service替换为jdbc-user-service，替换内容如下所示：

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

将上述红色部分替换为下面黄色部分。

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

现在只要再为jdbc-user-service提供一个dataSource就可以让Spring Security使用数据库中的权限信息

了。在此我们使用spring创建一个演示用的dataSource实现，这个dataSource会连接到hsqldb数据库，从中获取用户权限信息。^[1]

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

最终的配置文件如下所示：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/security"
  xmlns:beans="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
    http://www.springframework.org/schema/security
    http://www.springframework.org/schema/security/spring-security-2.0.4.xsd">

  <http auto-config='true'>
    <intercept-url pattern="/admin.jsp" access="ROLE_ADMIN" />
    <intercept-url pattern="/*" access="ROLE_USER" />
  </http>

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

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

2.2. 数据库表结构

Spring Security默认情况下需要两张表，用户表和权限表。以下是hsqldb中的建表语句：

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

create table authorities ❷
  username varchar_ignorecase(50) not null,
  authority varchar_ignorecase(50) not null,
  constraint fk_authorities_users foreign key(username) references users(username)
);

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

❶ users：用户表。包含username用户登录名，password登陆密码，enabled用户是否被禁用三个字段。

其中username用户登录名为主键。

❷ authorities：权限表。包含username用户登录名，authorities对应权限两个字段。

其中username字段与users用户表的主键使用外键关联。

❸ 对authorities权限表的username和authority创建唯一索引，提高查询效率。

Spring Security会在初始化时，从这两张表中获得用户信息和对应权限，将这些信息保存到缓存中。其中users表中的登录名和密码用来控制用户的登录，而权限表中的信息用来控制用户登陆后是否有权限访问受保护的系统资源。

我们在示例中预先初始化了一部分数据：

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

insert into authorities(username,authority) values('admin','ROLE_ADMIN');
insert into authorities(username,authority) values('admin','ROLE_USER');
insert into authorities(username,authority) values('user','ROLE_USER');
```

上述sql中，我们创建了两个用户admin和user，其中admin拥有ROLE_ADMIN和ROLE_USER权限，而user只拥有ROLE_USER权限。这和我们上一章中的配置相同，因此本章实例的效果也和上一章完全相同，这里就不再赘述了。

实例见ch002。

^[1] javax.sql.DataSource是一个用来定义数据库连接池的统一接口。当我们想调用任何实现了javax.sql.DataSource接口的连接池，只需要调用接口提供的getConnection()就可以获得连接池中的jdbc连接。javax.sql.DataSource可以屏蔽连接池的不同实现，我们使用的连接池即可能由第三方包单独提供，也可能是由jee容器统一管理提供的。

[上一页](#)

第 1 章 一个简单的HelloWorld

[上一级](#)

[起始页](#)

[下一页](#)

第 3 章 自定义数据库表结构

第3章 自定义数据库表结构

Spring Security默认提供的表结构太过简单了，其实就算默认提供的表结构很复杂，也无法满足所有企业内部对用户信息和权限信息管理的要求。基本上每个企业内部都有一套自己的用户信息管理结构，同时也会有一套对应的权限信息体系，如何让Spring Security在这些已有的数据结构之上运行呢？

3.1. 自定义表结构

假设我们实际使用的表结构如下所示：

```
-- 角色
create table role(
  id bigint,
  name varchar(50),
  descn varchar(200)
);
alter table role add constraint pk_role primary key(id);
alter table role alter column id bigint generated by default as identity(start with 1);

-- 用户
create table user(
  id bigint,
  username varchar(50),
  password varchar(50),
  status integer,
  descn varchar(200)
);
alter table user add constraint pk_user primary key(id);
alter table user alter column id bigint generated by default as identity(start with 1);
```

-- 用户角色连接表

```
create table user_role(
```

```
    user_id bigint,
```

```
    role_id bigint
```

```
);
```

```
alter table user_role add constraint pk_user_role primary key(user_id, role_id);
```

```
alter table user_role add constraint fk_user_role_user foreign key(user_id) references user(id);
```

```
alter table user_role add constraint fk_user_role_role foreign key(role_id) references role(id);
```

上述共有三张表，其中user用户表，role角色表为保存用户权限数据的主表，user_role为关联表。user用户表，role角色表之间为多对多关系，就是说一个用户可以有多个角色。ER图如下所示：

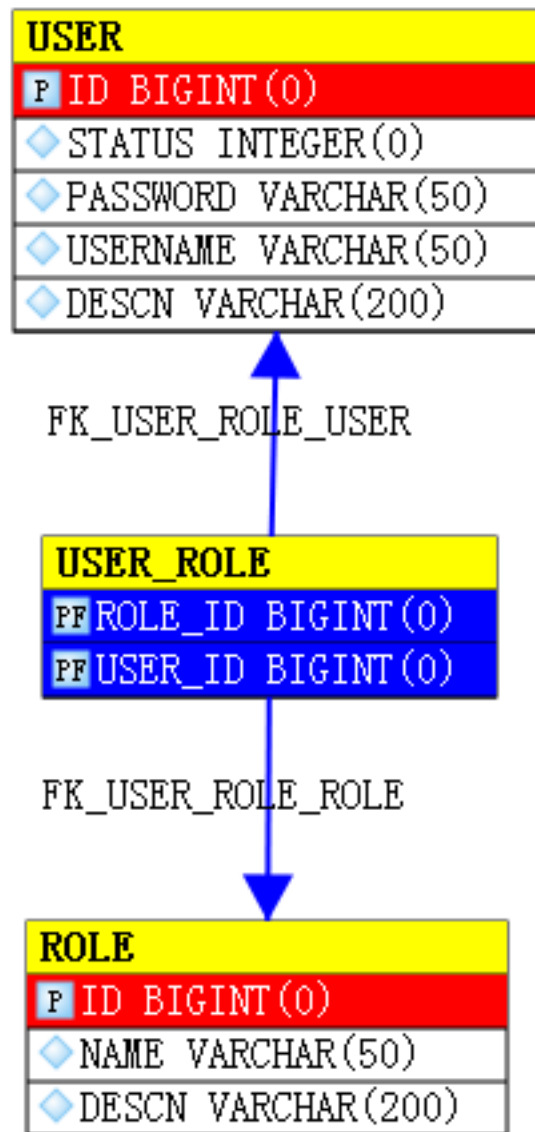


图 3.1. 数据库表关系

3.2. 初始化数据

创建两个用户，admin和user。admin用户拥有“管理员”角色，user用户拥有“用户”角色。

```
insert into user(id,username,password,status,descn) values(1,'admin','admin',1,'管理员');
insert into user(id,username,password,status,descn) values(2,'user','user',1,'用户');

insert into role(id,name,descn) values(1,'ROLE_ADMIN','管理员角色');
insert into role(id,name,descn) values(2,'ROLE_USER','用户角色');

insert into user_role(user_id,role_id) values(1,1);
insert into user_role(user_id,role_id) values(1,2);
insert into user_role(user_id,role_id) values(2,2);
```

3.3. 获得自定义用户权限信息

现在我们要在这样的数据结构基础上使用Spring Security，Spring Security所需要的数据只是为了处理两种情况，一是判断登录用户是否合法，二是判断登陆的用户是否有权限访问受保护的系统资源。

我们所要做的工作就是在现有数据结构的基础上，为Spring Security提供这两种数据。

3.3.1. 处理用户登陆

当用户登陆时，系统需要判断用户登录名是否存在，登陆密码是否正确，当前用户是否被禁用。我们使用下列SQL来提取这三个信息。

```
select username,password,status as enabled
from user
where username=?
```

3.3.2. 检验用户权限

用户登陆之后，系统需要获得该用户的所有权限，根据用户已被赋予的权限来判断哪些系统资源可以被用户访问，哪些资源不允许用户访问。

以下SQL就可以获得当前用户所拥有的权限。

```
select u.username,r.name as authority
from user u
join user_role ur
on u.id=ur.user_id
join role r
on r.id=ur.role_id
where u.username=?"/>
```

将这两条SQL语句配置到xml中，就可以让Spring Security从我们自定义的表结构中提取数据了。最终配置文件如下所示：

```
<authentication-provider>
  <jdbc-user-service data-source-ref="dataSource"
    ❶users-by-username-query="select username,password,status as enabled
      from user
      where username=?"
    ❷authorities-by-username-query="select u.username,r.name as authority
      from user u
      join user_role ur
      on u.id=ur.user_id
      join role r
      on r.id=ur.role_id
      where u.username=?"/>
</authentication-provider>
```

- ❶ users-by-username-query为根据用户名查找用户，系统通过传入的用户名查询当前用户的登录名，密码和是否被禁用这一状态。
- ❷ authorities-by-username-query为根据用户名查找权限，系统通过传入的用户名查询当前用户已被授予的所有权限。

实例见ch003。

[上一页](#)

第 2 章 使用数据库管理用户权限

[上一级](#)

[起始页](#)

[下一页](#)

第 4 章 自定义登陆页面

第4章 自定义登陆页面

Spring Security虽然默认提供了一个登陆页面，但是这个页面实在太简陋了，只有在快速演示时才有可能它做系统的登陆页面，实际开发时无论是从美观还是实用性角度考虑，我们都必须实现自定义的登录页面。

4.1. 实现自定义登陆页面

自己实现一个login.jsp，放在src/main/webapp/目录下。

```
+ ch04/  
+ src/  
+ main/  
+ resources/  
  * applicationContext.xml  
+ webapp/  
+ WEB-INF/  
  * web.xml  
  * admin.jsp  
  * index.jsp  
  * login.jsp  
+ test/  
+ resources/  
  * pom.xml
```

4.2. 修改配置文件

在xml中的http标签中添加一个form-login标签。

```
<http auto-config='true'>  
  <intercept-url pattern="/login.jsp" access="IS_AUTHENTICATED_ANONYMOUSLY" />❶  
  <intercept-url pattern="/admin.jsp" access="ROLE_ADMIN" />  
  <intercept-url pattern="/*" access="ROLE_USER" />
```

```
<form-login login-page="/login.jsp"❷
    authentication-failure-url="/login.jsp?error=true"❸
    default-target-url="/" />❹
</http>
```

- ❶ 让没登陆的用户也可以访问login.jsp。^[2]

这是因为配置文件中的“/**”配置，要求用户访问任意一个系统资源时，必须拥有ROLE_USER角色，/login.jsp也不例外，如果我们不为/login.jsp单独配置访问权限，会造成用户连登陆的权限都没有，这是不正确的。

- ❷ login-page表示用户登陆时显示我们自定义的login.jsp。

这时我们访问系统显示的登陆页面将是我们上面创建的login.jsp。

- ❸ authentication-failure-url表示用户登陆失败时，跳转到哪个页面。

当用户输入的登录名和密码不正确时，系统将再次跳转到/login.jsp，并添加一个error=true参数作为登陆失败的标示。

- ❹ default-target-url表示登陆成功时，跳转到哪个页面。^[3]

4.3. 登陆页面中的参数配置

以下是我们创建的login.jsp页面的主要代码。

```
<div class="error ${param.error == true ? '' : 'hide'}">
    登陆失败<br>
    ${sessionScope['SPRING_SECURITY_LAST_EXCEPTION'].message}
</div>
<form action="${pageContext.request.contextPath}/j_spring_security_check"❶ style="width:260px;text-align:center;">
    <fieldset>
        <legend>登陆</legend>
        用户： <input type="text" name="j_username"❷ style="width:150px;" value="${sessionScope
['SPRING_SECURITY_LAST_USERNAME']}" /><br />
        密码： <input type="password" name="j_password"❸ style="width:150px;" /><br />
        <input type="checkbox" name="_spring_security_remember_me"❹ />两周之内不必登陆<br />
        <input type="submit" value="登陆"/>
        <input type="reset" value="重置"/>
    </fieldset>
</form>
```

❶ /j_spring_security_check, 提交登陆信息的URL地址。

自定义form时, 要把form的action设置为/j_spring_security_check。注意这里要使用绝对路径, 避免登陆页面存放的页面可能带来的问题。^[4]

❷ j_username, 输入登陆名的参数名称。

❸ j_password, 输入密码的参数名称

❹ _spring_security_remember_me, 选择是否允许自动登录的参数名称。

可以直接把这个参数设置为一个checkbox, 无需设置value, Spring Security会自行判断它是否被选中。

以上介绍了自定义页面上Spring Security所需的基本元素, 这些参数名称都采用了Spring Security中默认的配置值, 如果有特殊需要还可以通过配置文件进行修改。

4.4. 测试一下

经过以上配置, 我们终于使用了一个自己创建的登陆页面替换了原来Spring Security默认提供的登录页面了。我们不仅仅是做个样子, 而是实际配置了各个Spring Security所需的参数, 真正将自定义登陆页面与Spring Security紧紧的整合在了一起。以下是使用自定义登陆页面实际运行时的截图。

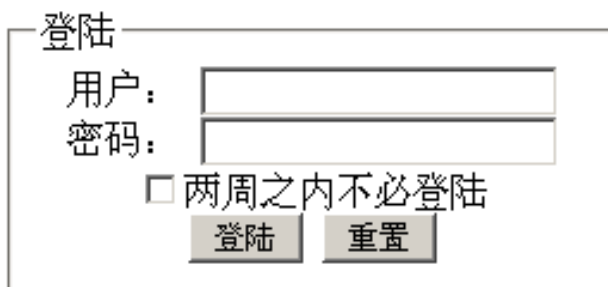


图 4.1. 进入登录页面

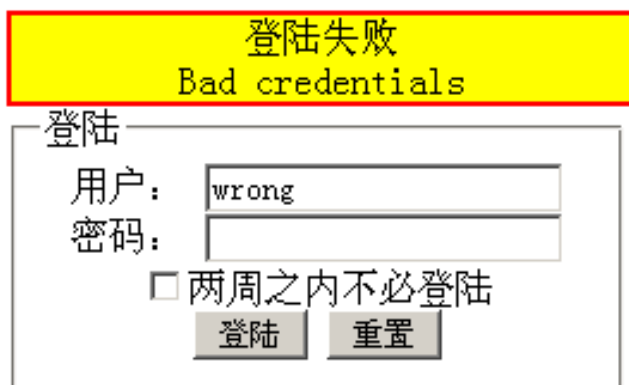


图 4.2. 用户登陆失败

实例见ch004。

^[2] 有关匿名用户的知识，我们会在之后的章节中进行讲解。

^[3] 登陆成功后跳转策略的知识，我们会在之后的章节中进行讲解。

^[4] 关于绝对路径和相对路径的详细讨论，请参考<http://family168.com/tutorial/jsp/html/jsp-ch-03.html#jsp-ch-03-04-01>

[上一页](#)

第 3 章 自定义数据库表结构

[上一级](#)

[起始页](#)

[下一页](#)

第 5 章 使用数据库管理资源

第 5 章 使用数据库管理资源

国内对权限系统的基本要求是将用户权限和被保护资源都放在数据库里进行管理，在这点上Spring Security并没有给出官方的解决方案，为此我们需要对Spring Security进行扩展。

5.1. 数据库表结构

这次我们使用五张表，user用户表，role角色表，resc资源表相互独立，它们通过各自之间的连接表实现多对多关系。

```
-- 资源
create table resc(
  id bigint,
  name varchar(50),
  res_type varchar(50),
  res_string varchar(200),
  descn varchar(200)
);
alter table resc add constraint pk_resc primary key(id);
alter table resc alter column id bigint generated by default as identity(start with 1);

-- 角色
create table role(
  id bigint,
  name varchar(50),
  descn varchar(200)
);
alter table role add constraint pk_role primary key(id);
alter table role alter column id bigint generated by default as identity(start with 1);

-- 用户
create table user(
  id bigint,
  username varchar(50),
  password varchar(50),
```

```

status integer,
descn varchar(200)
);
alter table user add constraint pk_user primary key(id);
alter table user alter column id bigint generated by default as identity(start with 1);

-- 资源角色连接表
create table resc_role(
    resc_id bigint,
    role_id bigint
);
alter table resc_role add constraint pk_resc_role primary key(resc_id, role_id);
alter table resc_role add constraint fk_resc_role_resc foreign key(resc_id) references resc(id);
alter table resc_role add constraint fk_resc_role_role foreign key(role_id) references role(id);

-- 用户角色连接表
create table user_role(
    user_id bigint,
    role_id bigint
);
alter table user_role add constraint pk_user_role primary key(user_id, role_id);
alter table user_role add constraint fk_user_role_user foreign key(user_id) references user(id);
alter table user_role add constraint fk_user_role_role foreign key(role_id) references role(id);

```

user表中包含用户登陆信息，role角色表中包含授权信息，resc资源表中包含需要保护的资源。

ER图如下所示：

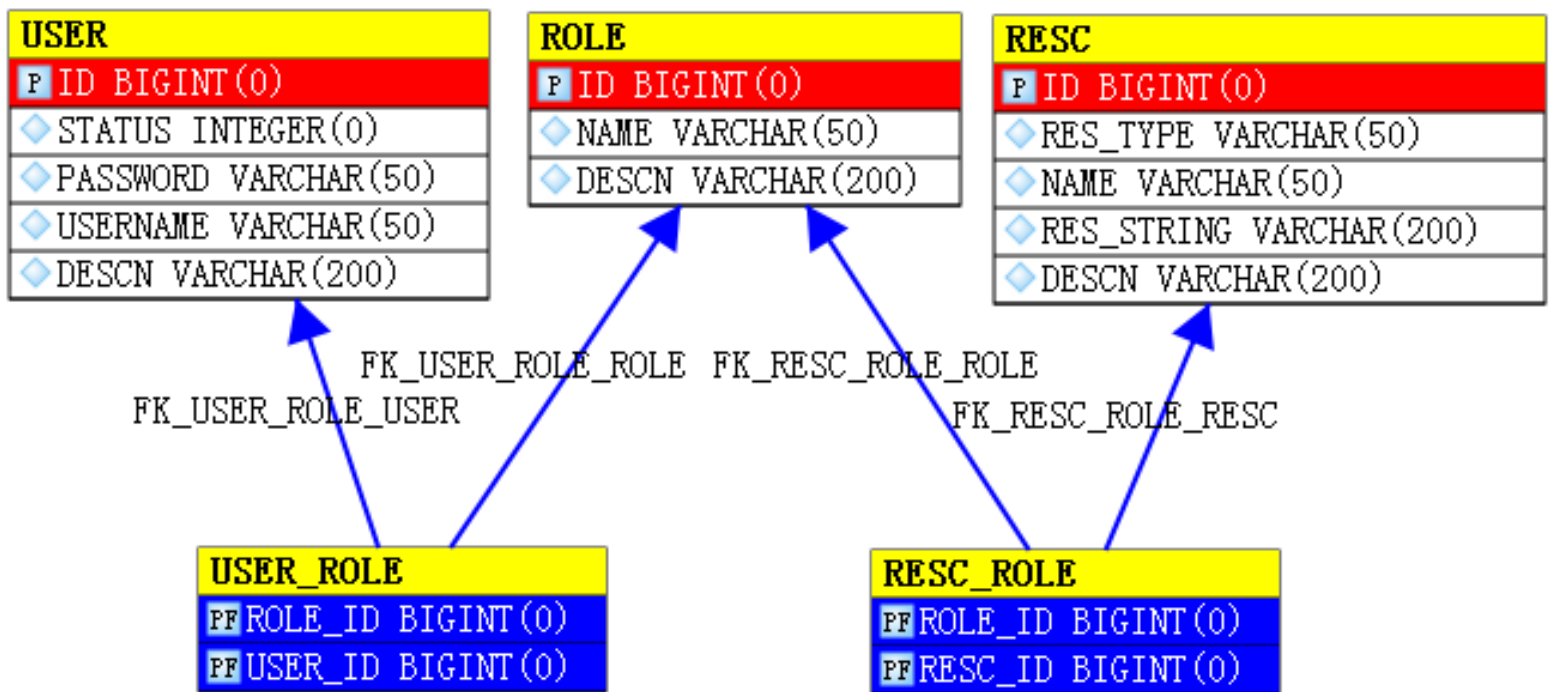


图 5.1. 数据库表关系

5.2. 初始化数据

创建的两个用户分别对应“管理员”角色和“用户”角色。而“管理员”角色可以访问“/admin.jsp”和“/**”，“用户”角色只能访问“/**”。

```
insert into user(id,username,password,status,descn) values(1,'admin','admin',1,'管理员');
insert into user(id,username,password,status,descn) values(2,'user','user',1,'用户');
```

```
insert into role(id,name,descn) values(1,'ROLE_ADMIN','管理员角色');
insert into role(id,name,descn) values(2,'ROLE_USER','用户角色');
```

```
insert into resc(id,name,res_type,res_string,descn) values(1,'','URL','/admin.jsp','');
insert into resc(id,name,res_type,res_string,descn) values(2,'','URL','/**','');
```

```
insert into resc_role(resc_id,role_id) values(1,1);
insert into resc_role(resc_id,role_id) values(2,1);
insert into resc_role(resc_id,role_id) values(2,2);
```

```
insert into user_role(user_id,role_id) values(1,1);
insert into user_role(user_id,role_id) values(1,2);
insert into user_role(user_id,role_id) values(2,2);
```

5.3. 实现从数据库中读取资源信息

Spring Security没有提供从数据库获得获取资源信息的方法，实际上Spring Security甚至没有为我们留一个半个的扩展接口，所以我们这次要费点儿脑筋了。

首先，要搞清楚需要提供何种类型的数据，然后，寻找可以让我们编写的代码替换原有功能的切入点，实现了以上两步之后，就可以宣布大功告成了。

5.3.1. 需要何种数据格式

从配置文件上可以看到，Spring Security所需的数据应该是一系列URL网址和访问这些网址所需的权限：

```
<intercept-url pattern="/login.jsp" access="IS_AUTHENTICATED_ANONYMOUSLY" />1
<intercept-url pattern="/admin.jsp" access="ROLE_ADMIN" />
<intercept-url pattern="/**" access="ROLE_USER" />
```


Spring Security所做的就是在系统初始化时，将以上XML中的信息转换为特定的数据格式，而框架中其他组件可以利用这些特定格式的数据，用于控制之后的验证操作。

现在这些资源信息都保存在数据库中，我们可以使用上面介绍的SQL语句从数据中查询。

```
select re.res_string,r.name
  from role r
 join resc_role rr
    on r.id=rr.role_id
 join resc re
    on re.id=rr.resc_id
```

下面要开始编写实现代码了。

1. 搜索数据库获得资源信息。

我们通过定义一个MappingSqlQuery实现数据库操作。

```
private class ResourceMapping extends MappingSqlQuery {
    protected ResourceMapping(DataSource dataSource,
        String resourceQuery) {
        super(dataSource, resourceQuery);
        compile();
    }

    protected Object mapRow(ResultSet rs, int rownum)
        throws SQLException {
        String url = rs.getString(1);
        String role = rs.getString(2);
        Resource resource = new Resource(url, role);

        return resource;
    }
}
```

这样我们可以执行它的execute()方法获得所有资源信息。

```
protected List<Resource> findResources() {
    ResourceMapping resourceMapping = new ResourceMapping(getDataSource(),
        resourceQuery);
```

```
return resourceMapping.execute();
}
```

2. 使用获得的资源信息组装requestMap。

```
protected LinkedHashMap<RequestKey, ConfigAttributeDefinition> buildRequestMap() {
    LinkedHashMap<RequestKey, ConfigAttributeDefinition> requestMap = null;
    requestMap = new LinkedHashMap<RequestKey, ConfigAttributeDefinition>();

    ConfigAttributeEditor editor = new ConfigAttributeEditor();

    List<Resource> resourceList = this.findResources();

    for (Resource resource : resourceList) {
        RequestKey key = new RequestKey(resource.getUrl(), null);
        editor.setAsText(resource.getRole());
        requestMap.put(key,
            (ConfigAttributeDefinition) editor.getValue());
    }

    return requestMap;
}
```

3. 使用urlMatcher和requestMap创建DefaultFilterInvocationDefinitionSource。

```
public Object getObject() {
    return new DefaultFilterInvocationDefinitionSource(this
        .getUrlMatcher(), this.buildRequestMap());
}
```

这样我们就获得了DefaultFilterInvocationDefinitionSource，剩下的只差把这个我们自己创建的类替换掉原有的代码了。

完整代码如下所示：

```
package com.family168.springsecuritybook.ch05;

import java.sql.ResultSet;
import java.sql.SQLException;

import java.util.LinkedHashMap;
import java.util.List;
```

```

import javax.sql.DataSource;

import org.springframework.beans.factory.FactoryBean;

import org.springframework.jdbc.core.support.JdbcDaoSupport;
import org.springframework.jdbc.object.MappingSqlQuery;

import org.springframework.security.ConfigAttributeDefinition;
import org.springframework.security.ConfigAttributeEditor;
import org.springframework.security.intercept.web.DefaultFilterInvocationDefinitionSource;
import org.springframework.security.intercept.web.FilterInvocationDefinitionSource;
import org.springframework.security.intercept.web.RequestKey;
import org.springframework.security.util.AntUrlPathMatcher;
import org.springframework.security.util.UrlMatcher;

public class JdbcFilterInvocationDefinitionSourceFactoryBean
    extends JdbcDaoSupport implements FactoryBean {
    private String resourceQuery;

    public boolean isSingleton() {
        return true;
    }

    public Class getObjectType() {
        return FilterInvocationDefinitionSource.class;
    }

    public Object getObject() {
        return new DefaultFilterInvocationDefinitionSource(this
            .getUrlMatcher(), this.buildRequestMap());
    }

    protected List<Resource> findResources() {
        ResourceMapping resourceMapping = new ResourceMapping(getDataSource(),
            resourceQuery);

        return resourceMapping.execute();
    }

    protected LinkedHashMap<RequestKey, ConfigAttributeDefinition> buildRequestMap() {
        LinkedHashMap<RequestKey, ConfigAttributeDefinition> requestMap = null;
        requestMap = new LinkedHashMap<RequestKey, ConfigAttributeDefinition>();

        ConfigAttributeEditor editor = new ConfigAttributeEditor();

        List<Resource> resourceList = this.findResources();

```

```

    for (Resource resource : resourceList) {
        RequestKey key = new RequestKey(resource.getUrl(), null);
        editor.setAsText(resource.getRole());
        requestMap.put(key,
            (ConfigAttributeDefinition) editor.getValue());
    }

    return requestMap;
}

protected UrlMatcher getUrlMatcher() {
    return new AntUrlPathMatcher();
}

public void setResourceQuery(String resourceQuery) {
    this.resourceQuery = resourceQuery;
}

private class Resource {
    private String url;
    private String role;

    public Resource(String url, String role) {
        this.url = url;
        this.role = role;
    }

    public String getUrl() {
        return url;
    }

    public String getRole() {
        return role;
    }
}

private class ResourceMapping extends MappingSqlQuery {
    protected ResourceMapping(DataSource dataSource,
        String resourceQuery) {
        super(dataSource, resourceQuery);
        compile();
    }

    protected Object mapRow(ResultSet rs, int rownum)
        throws SQLException {
        String url = rs.getString(1);
        String role = rs.getString(2);
        Resource resource = new Resource(url, role);
    }
}

```

```

        return resource;
    }
}
}

```

5.3.2. 替换原有功能的切入点

在spring中配置我们编写的代码。

```

<beans:bean id="filterInvocationDefinitionSource"
    class="com.family168.springsecuritybook.ch05.JdbcFilterInvocationDefinitionSourceFactoryBean">
    <beans:property name="dataSource" ref="dataSource"/>
    <beans:property name="resourceQuery" value="
        select re.res_string,r.name
        from role r
        join resc_role rr
        on r.id=rr.role_id
        join resc re
        on re.id=rr.resc_id
    "/>
</beans:bean>

```

下一步使用这个filterInvocationDefinitionSource创建filterSecurityInterceptor，并使用它替换系统原来创建的那个过滤器。

```

<beans:bean id="filterSecurityInterceptor"
    class="org.springframework.security.intercept.web.FilterSecurityInterceptor" autowire="byType">
    <custom-filter before="FILTER_SECURITY_INTERCEPTOR" />
    <beans:property name="objectDefinitionSource" ref="filterInvocationDefinitionSource" />
</beans:bean>

```

注意这个custom-filter标签，它表示将filterSecurityInterceptor放在框架原来的FILTER_SECURITY_INTERCEPTOR过滤器之前，这样我们的过滤器会先于原来的过滤器执行，因为它的功能与老过滤器完全一样，所以这就等于把原来的过滤器替换掉了。

完整的配置文件如下所示：

```

<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/security"
    xmlns:beans="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

```

```

xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
http://www.springframework.org/schema/security
http://www.springframework.org/schema/security/spring-security-2.0.4.xsd">

```

```

<http auto-config="true"/>

```

```

<authentication-provider>

```

```

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

```

```

        users-by-username-query="select username,password,status as enabled
            from user
            where username=?"

```

```

        authorities-by-username-query="select u.username,r.name as authority
            from user u
            join user_role ur
            on u.id=ur.user_id
            join role r
            on r.id=ur.role_id
            where u.username=?" />

```

```

</authentication-provider>

```

```

<beans:bean id="filterSecurityInterceptor"

```

```

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

```

```

    <custom-filter before="FILTER_SECURITY_INTERCEPTOR" />

```

```

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

```

```

</beans:bean>

```

```

<beans:bean id="filterInvocationDefinitionSource"

```

```

    class="com.family168.springsecuritybook.ch05.JdbcFilterInvocationDefinitionSourceFactoryBean">

```

```

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

```

```

    <beans:property name="resourceQuery" value="

```

```

        select re.res_string,r.name
        from role r
        join resc_role rr
        on r.id=rr.role_id
        join resc re
        on re.id=rr.resc_id
    " />

```

```

    " />

```

```

</beans:bean>

```

```

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

```

```

    <beans:property name="driverClassName" value="org.sqljdbc.jdbcDriver"/>

```

```

    <beans:property name="url" value="jdbc:sqljdbc:res://sqljdbc/test"/>

```

```

    <beans:property name="username" value="sa"/>

```

```

    <beans:property name="password" value=""/>

```

```

</beans:bean>

```

```

</beans:beans>

```

实例见ch05。

如果数据库中的资源出现的变化，需要刷新内存中已加载的资源信息时，使用下面代码：

```
<%@page import="org.springframework.context.ApplicationContext"%>
<%@page import="org.springframework.web.context.support.WebApplicationContextUtils"%>
<%@page import="org.springframework.beans.factory.FactoryBean"%>
<%@page import="org.springframework.security.intercept.web.FilterSecurityInterceptor"%>
<%@page import="org.springframework.security.intercept.web.FilterInvocationDefinitionSource"%>
<%
    ApplicationContext ctx = WebApplicationContextUtils.getWebApplicationContext(application);
    FactoryBean factoryBean = (FactoryBean) ctx.getBean("&filterInvocationDefinitionSource");
    FilterInvocationDefinitionSource fids = (FilterInvocationDefinitionSource) factoryBean.getObject();
    FilterSecurityInterceptor filter = (FilterSecurityInterceptor) ctx.getBean("filterSecurityInterceptor");
    filter.setObjectDefinitionSource(fids);
%>
<jsp:forward page="/" />
```

现在这些代码只是临时解决方案，我们需要在之后添加ehcache支持。

[上一页](#)[第4章 自定义登陆页面](#)[上一级](#)[起始页](#)[下一页](#)[第6章 控制用户信息](#)

第6章 控制用户信息

让我们来研究一些与用户信息相关的功能，包括为用户密码加密，缓存用户信息，获得系统当前登陆的用户，获得登陆用户的所有权限。

6.1. MD5加密

任何一个正式的企业应用中，都不会在数据库中使用明文来保存密码的，我们在之前的章节中都是为了方便起见没有对数据库中的用户密码进行加密，这在实际应用中是极为幼稚的做法。可以想象一下，只要有人进入数据库就可以看到所有人的密码，这是一件多么恐怖的事情，为此我们至少要对密码进行加密，这样即使数据库被攻破，也可以保证用户密码的安全。

最常用的方法是使用MD5算法对密码进行摘要加密，这是一种单项加密手段，无法通过加密后的结果反推回原来的密码明文。

为了使用MD5对密码加密，我们需要修改一下配置文件。

```
<authentication-provider>
  <password-encoder hash="md5"/>
  <jdbc-user-service data-source-ref="dataSource"/>
</authentication-provider>
```

上述代码中新增的黄色部分，将启用md5算法。这时我们在数据库中保存的密码已经不再是明文了，它看起来像是一堆杂乱无章的乱码。

```
INSERT INTO USERS VALUES('admin','21232f297a57a5a743894a0e4a801fc3',TRUE)
INSERT INTO USERS VALUES('user','ee11cbb19052e40b07aac0ca060c23ee',TRUE)
```

可以看到密码部分已经面目全非了，即使有人攻破了数据库，拿到这种“乱码”也无法登陆系统窃取客户的信息。

这些配置对普通客户不会造成任何影响，他们只需要输入自己的密码，Spring Security会自动加以演算，将生成的结果与数据库中保存的信息进行比对，以此来判断用户是否可以登陆。

这样，我们只添加了一行配置，就为系统带来了密码加密的功能。

6.2. 盐值加密

实际上，上面的实例在现实使用中还存在着一个不小的问题。虽然md5算法是不可逆的，但是因为它对同一个字符串计算的结果是唯一的，所以一些人可能会使用“字典攻击”的方式来攻破md5加密的系统^[5]。这虽然属于暴力解密，却十分有效，因为大多数系统的用户密码都不回很长。

实际上，大多数系统都是用admin作为默认的管理员登陆密码，所以，当我们在数据库中看到“21232f297a57a5a743894a0e4a801fc3”时，就可以意识到admin用户使用的密码了。因此，md5在处理这种常用字符串时，并不怎么奏效。

为了解决这个问题，我们可以使用盐值加密“salt-source”。

修改配置文件：

```
<authentication-provider>
  <password-encoder hash="md5">
    <salt-source user-property="username"/>
  </password-encoder>
  <jdbc-user-service data-source-ref="dataSource"/>
</authentication-provider>
```

在password-encoder下添加了salt-source，并且指定使用username作为盐值。

盐值的原理非常简单，就是先把密码和盐值指定的内容合并在一起，再使用md5对合并后的内容进行演算，这样一来，就算密码是一个很常见的字符串，再加上用户名，最后算出来的md5值就没那么容易猜出来了。因为攻击者不知道盐值的值，也很难反算出密码原文。

我们这里将每个用户的username作为盐值，最后数据库中的密码部分就变成了这样：

```
INSERT INTO USERS VALUES('admin','ceb4f32325eda6142bd65215f4c0f371',TRUE)
INSERT INTO USERS VALUES('user','47a733d60998c719cf3526ae7d106d13',TRUE)
```

6.3. 用户信息缓存

介于系统的用户信息并不会经常改变，因此使用缓存就成为了提升性能的一个非常好的选择。Spring Security内置的缓存实现是基于ehcache的，为了启用缓存功能，我们要在配置文件中添加相关的内容。

```
<authentication-provider>
  <password-encoder hash="md5">
    <salt-source user-property="username"/>
  </password-encoder>
  <jdbc-user-service data-source-ref="dataSource" cache-ref="userCache"/>
</authentication-provider>
```

我们在jdbc-user-service部分添加了对userCache的引用，它将使用这个bean作为用户权限缓存的实现。对userCache的配置如下所示：

```
<beans:bean id="userCache" class="org.springframework.security.providers.dao.cache.EhCacheBasedUserCache">
  <beans:property name="cache" ref="userEhCache"/>
</beans:bean>

<beans:bean id="userEhCache" class="org.springframework.cache.ehcache.EhCacheFactoryBean">
  <beans:property name="cacheManager" ref="cacheManager"/>
  <beans:property name="cacheName" value="userCache"/>
</beans:bean>

<beans:bean id="cacheManager" class="org.springframework.cache.ehcache.EhCacheManagerFactoryBean"/>
```

EhCacheBasedUserCache是Spring Security内置的缓存实现，它将为jdbc-user-service提供缓存功能。它所引用的userEhCache来自spring提供的EhCacheFactoryBean和EhCacheManagerFactoryBean，对于userCache的缓存配置放在ehcache.xml中：

```
<ehcache>
  <diskStore path="java.io.tmpdir"/>

  <defaultCache
    maxElementsInMemory="1000"
    eternal="false"
    timeToIdleSeconds="120"
    timeToLiveSeconds="120"
    overflowToDisk="true"
  />

  <cache
    name="userCache"
    maxElementsInMemory="100"❶
```

```

eternal="false"❷
timeToIdleSeconds="600"❸
timeToLiveSeconds="3600"❹
overflowToDisk="true"❺
/>
</ehcache>

```

- ❶ 内存中最多存放100个对象。
- ❷ 不是永久缓存。
- ❸ 最大空闲时间为600秒。
- ❹ 最大活动时间为3600秒。
- ❺ 如果内存对象溢出则保存到磁盘。

如果想了解有关ehcache的更多配置，可以访问它的官方网站<http://ehcache.sf.net/>。

这样，我们就为用户权限信息设置好了缓存，当一个用户多次访问应用时，不需要每次去访问数据库了，ehcache会将对应的信息缓存起来，这将极大的提高系统的相应速度，同时也避免数据库符合过高的风险。



注意

cache-ref隐藏着一个陷阱，如果不看代码，我们也许会误认为cache-ref会在JdbcUserDetailsManager中设置对应的userCache，然后只要直接执行JdbcUserDetailsManager中的方法，就可以自动维护用户缓存。

可惜，cache-ref实际上是在JdbcUserDetailsManager的基础上，生成了一个CachingUserService，这个CachedUserDetailsService会拦截loadUserByUsername()方法，实现读取用户信息的缓存功能。我们在cache-ref中引用的UserCache实际上是放在CacheUserDetailsService中，而不是放到了原有的JdbcUserDetailsManager中，这就会导致JdbcUserDetailsManager中对用户缓存的操作全部失效。

6.4. 获取当前用户信息

如果只是想从页面上显示当前登陆的用户名，可以直接使用Spring Security提供的taglib。

```

<%@ taglib prefix="sec" uri="http://www.springframework.org/security/tags" %>
<div>username : <sec:authentication property="name"/></div>

```

如果想在程序中获得当前登陆用户对应的对象。

```
 UserDetails userDetails = (UserDetails) SecurityContextHolder.getContext()
    .getAuthentication()
    .getPrincipal();
```

如果想获得当前登陆用户所拥有的所有权限。

```
 GrantedAuthority[] authorities = userDetails.getAuthorities();
```

关于UserDetails是如何放到SecurityContext中去的，以及Spring Security所使用的ThreadLocal模式，我们会在后面详细介绍。这里我们已经了解了如何获得当前登陆用户的信息。

^[5] 所谓字典攻击，就是指将大量常用字符串使用md5加密，形成字典库，然后将一段由md5演算得到的未知字符串，在字典库中进行搜索，当发现匹配的结果时，就可以获得对应的加密前的字符串内容。

[上一页](#)

第 5 章 使用数据库管理资源

[上一级](#)

[起始页](#)

[下一页](#)

部分 II. 保护web篇

部分 II. 保护web篇

信息: FilterChainProxy: FilterChainProxy[

UrlMatcher = org.springframework.security.util.AntUrlPathMatcher[requiresLowerCase='true'];

Filter Chains: {

/**=[

org.springframework.security.context.HttpSessionContextIntegrationFilter[order=200;],

org.springframework.security.ui.logout.LogoutFilter[order=300;],

org.springframework.security.ui.webapp.AuthenticationProcessingFilter[order=700;],

org.springframework.security.ui.webapp.DefaultLoginPageGeneratingFilter[order=900;],

org.springframework.security.ui.basicauth.BasicProcessingFilter[order=1000;],

org.springframework.security.wrapper.SecurityContextHolderAwareRequestFilter[order=1100;],

org.springframework.security.ui.rememberme.RememberMeProcessingFilter[order=1200;],

org.springframework.security.providers.anonymous.AnonymousProcessingFilter[order=1300;],

org.springframework.security.ui.ExceptionTranslationFilter[order=1400;],

org.springframework.security.ui.SessionFixationProtectionFilter[order=1600;],

org.springframework.security.intercept.web.FilterSecurityInterceptor@e2fbeb

]

}

]

Spring Security一启动就会包含这样一批负责各种安全管理的过滤器，这部分的任务就是详细讲解每个过滤器的功能和用法，并讨论与之相关的各种控制权限的方法。

第 7 章 图解过滤器



图 7.1. auto-config='true'时的过滤器列表

7.1. HttpSessionContextIntegrationFilter

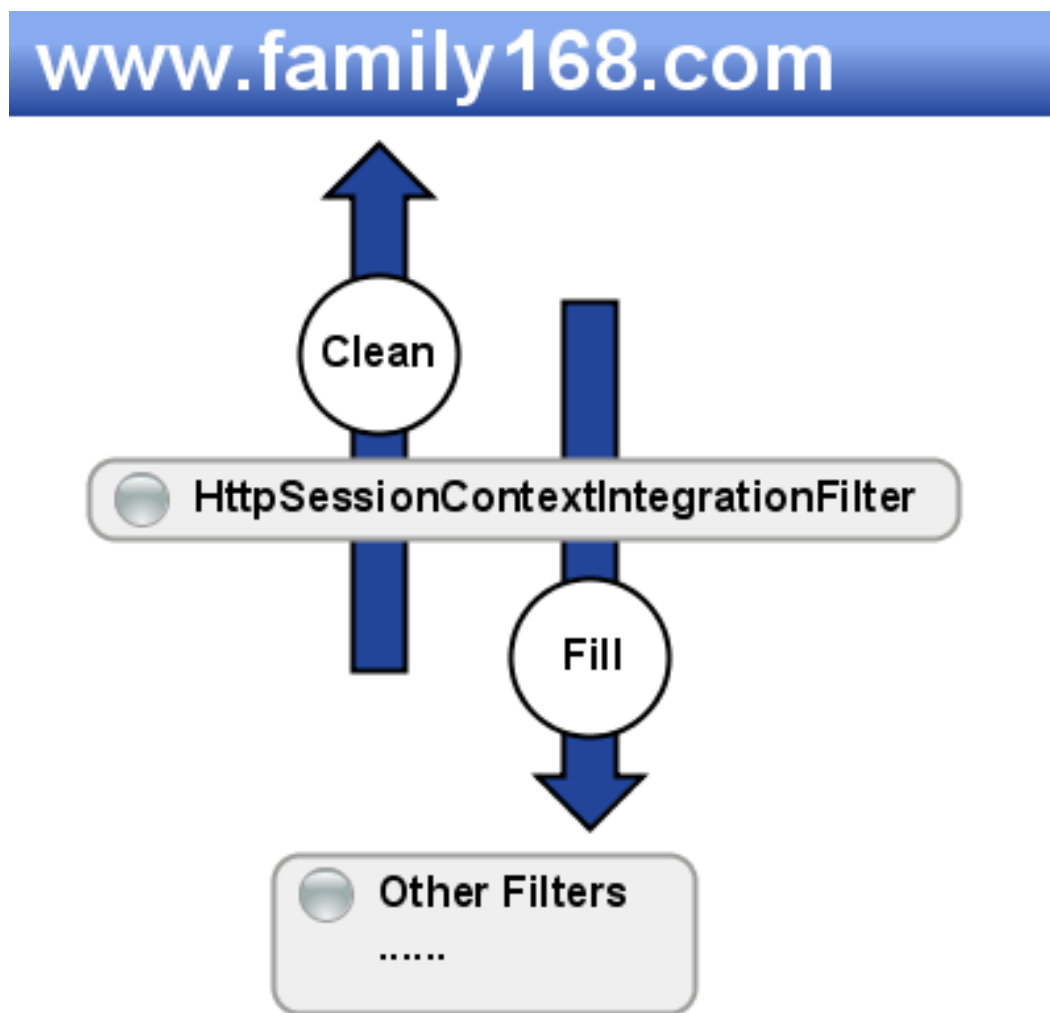


图 7.2. org.springframework.security.context.HttpSessionContextIntegrationFilter

位于过滤器顶端，第一个起作用的过滤器。

用途一，在执行其他过滤器之前，率先判断用户的session中是否已经存在一个SecurityContext了。如果存在，就把SecurityContext拿出来，放到SecurityContextHolder中，供Spring Security的其他部分使用。如果不存在，就创建一个SecurityContext出来，还是放到SecurityContextHolder中，供Spring Security的其他部分使用。

用途二，在所有过滤器执行完毕后，清空SecurityContextHolder，因为SecurityContextHolder是基于ThreadLocal的，如果在操作完成后清空ThreadLocal，会受到服务器的线程池机制的影响。

7.2. LogoutFilter

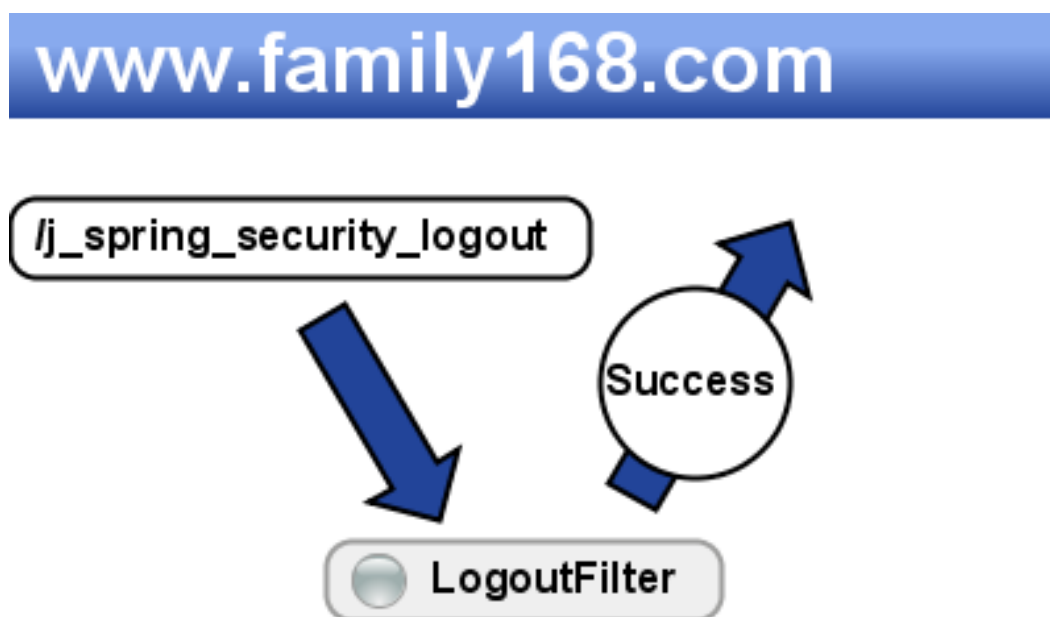


图 7.3. org.springframework.security.ui.logout.LogoutFilter

只处理注销请求，默认为/j_spring_security_logout。

用途是在用户发送注销请求时，销毁用户session，清空SecurityContextHolder，然后重定向到注销成功页面。可以与rememberMe之类的机制结合，在注销的同时清空用户cookie。

7.3. AuthenticationProcessingFilter

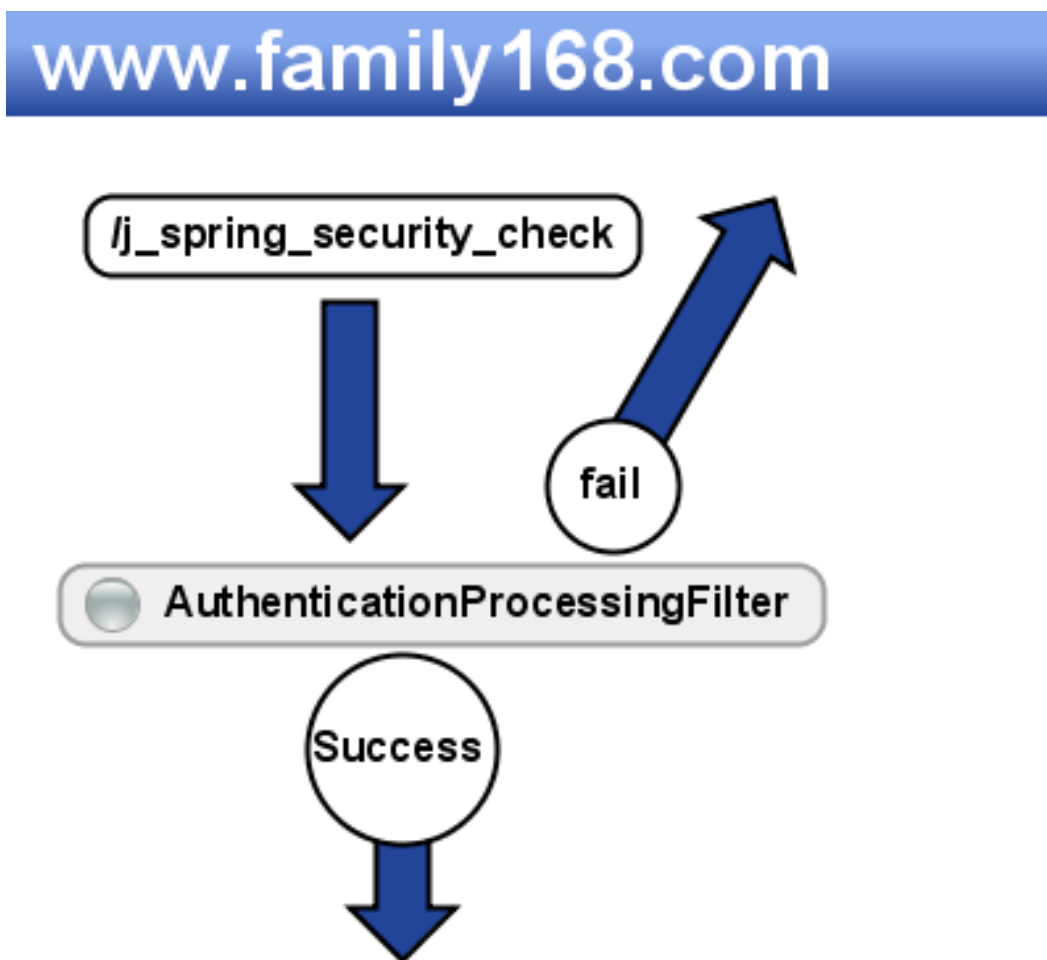


图 7.4. org.springframework.security.ui.webapp.AuthenticationProcessingFilter

处理form登陆的过滤器，与form登陆有关的所有操作都是在此进行的。

默认情况下只处理/j_spring_security_check请求，这个请求应该是用户使用form登陆后的提交地址，form所需的其他参数可以参考：[???](#)。

此过滤器执行的基本操作时，通过用户名和密码判断用户是否有效，如果登录成功就跳转到成功页面（可能是登陆之前访问的受保护页面，也可能是默认的成功页面），如果登录失败，就跳转到失败页面。

7.4. DefaultLoginPageGeneratingFilter

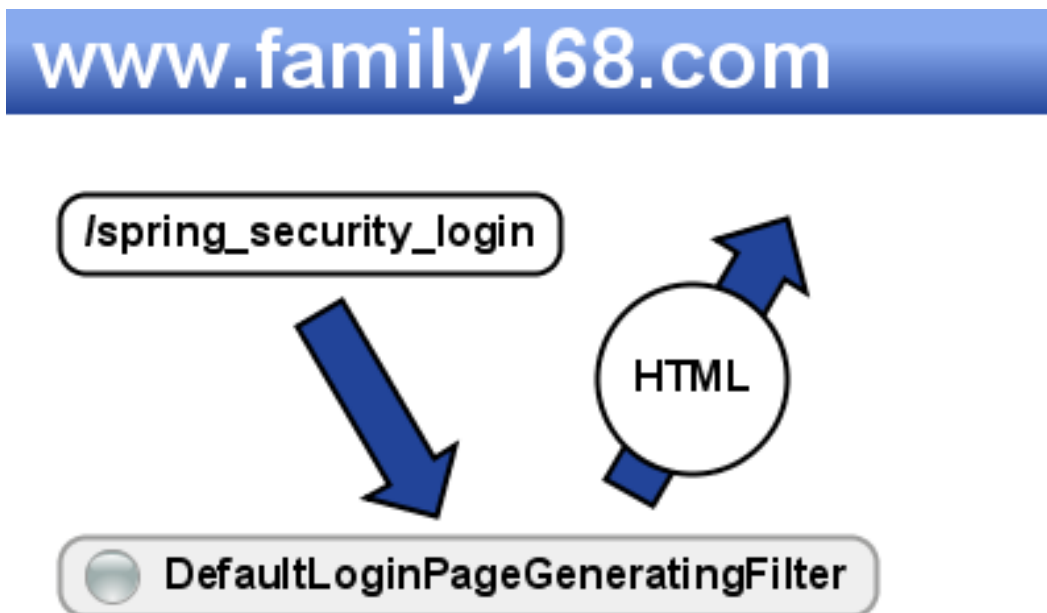


图 7.5. org.springframework.security.ui.webapp.DefaultLoginPageGeneratingFilter

此过滤器用来生成一个默认的登录页面，默认的访问地址为/spring_security_login，这个默认的登录页面虽然支持用户输入用户名，密码，也支持rememberMe功能，但是因为太难看了，只能是在演示时做个样子，不可能直接用在实际项目中。

如果想自定义登陆页面，可以参考：[第4章 自定义登陆页面](#)。

7.5. BasicProcessingFilter

www.family168.com

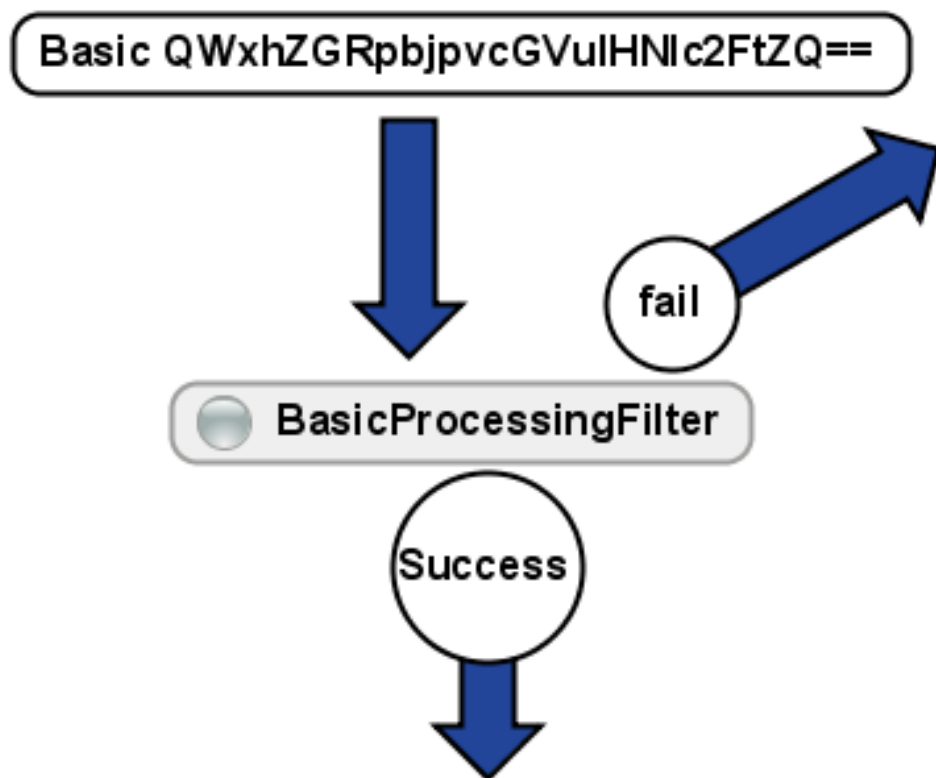


图 7.6. org.springframework.security.ui.basicauth.BasicProcessingFilter

此过滤器用于进行basic验证，功能与`AuthenticationProcessingFilter`类似，只是验证的方式不同。有关basic验证的详细情况，我们会在后面的章节中详细介绍。

有关basic验证的详细信息，可以参考：[第 10 章 basic认证](#)。

7.6. SecurityContextHolderAwareRequestFilter

www.family168.com

图 7.7. `org.springframework.security.wrapper.SecurityContextHolderAwareRequestFilter`

此过滤器用来包装客户的请求。目的是在原始请求的基础上，为后续程序提供一些额外的数据。比如`getRemoteUser()`时直接返回当前登陆的用户名之类的。

7.7. RememberMeProcessingFilter

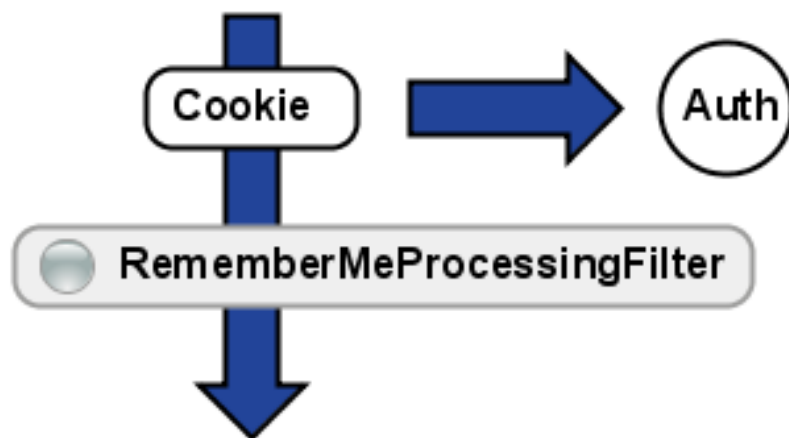


图 7.8. org.springframework.security.ui.rememberme.RememberMeProcessingFilter

此过滤器实现RememberMe功能，当用户cookie中存在rememberMe的标记，此过滤器会根据标记自动实现用户登陆，并创建SecurityContext，授予对应的权限。

有关rememberMe功能的详细信息，可以参考：[第12章 自动登录](#)。

7.8. AnonymousProcessingFilter

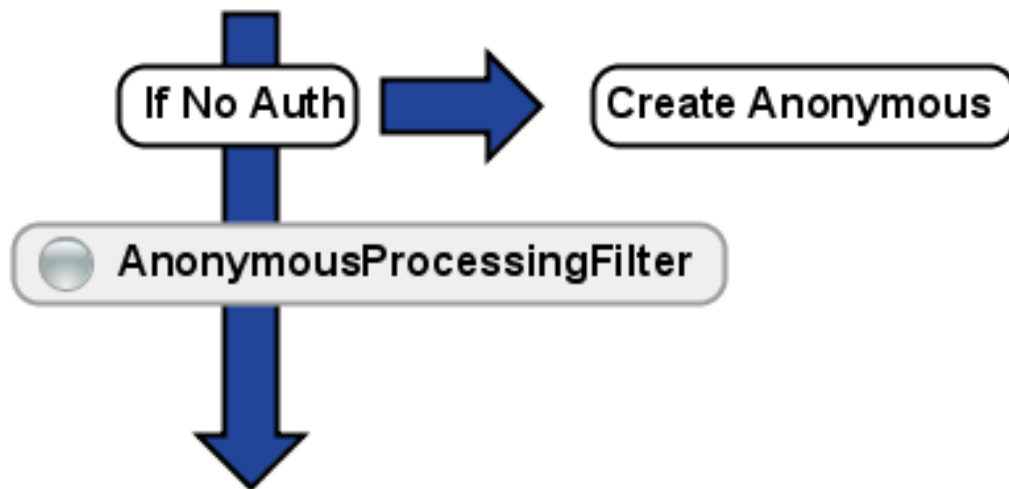


图 7.9. org.springframework.security.providers.anonymous.AnonymousProcessingFilter

为了保证操作统一性，当用户没有登陆时，默认为用户分配匿名用户的权限。

有关匿名登录功能的详细信息，可以参考：[第13章 匿名登录](#)。

7.9. ExceptionTranslationFilter

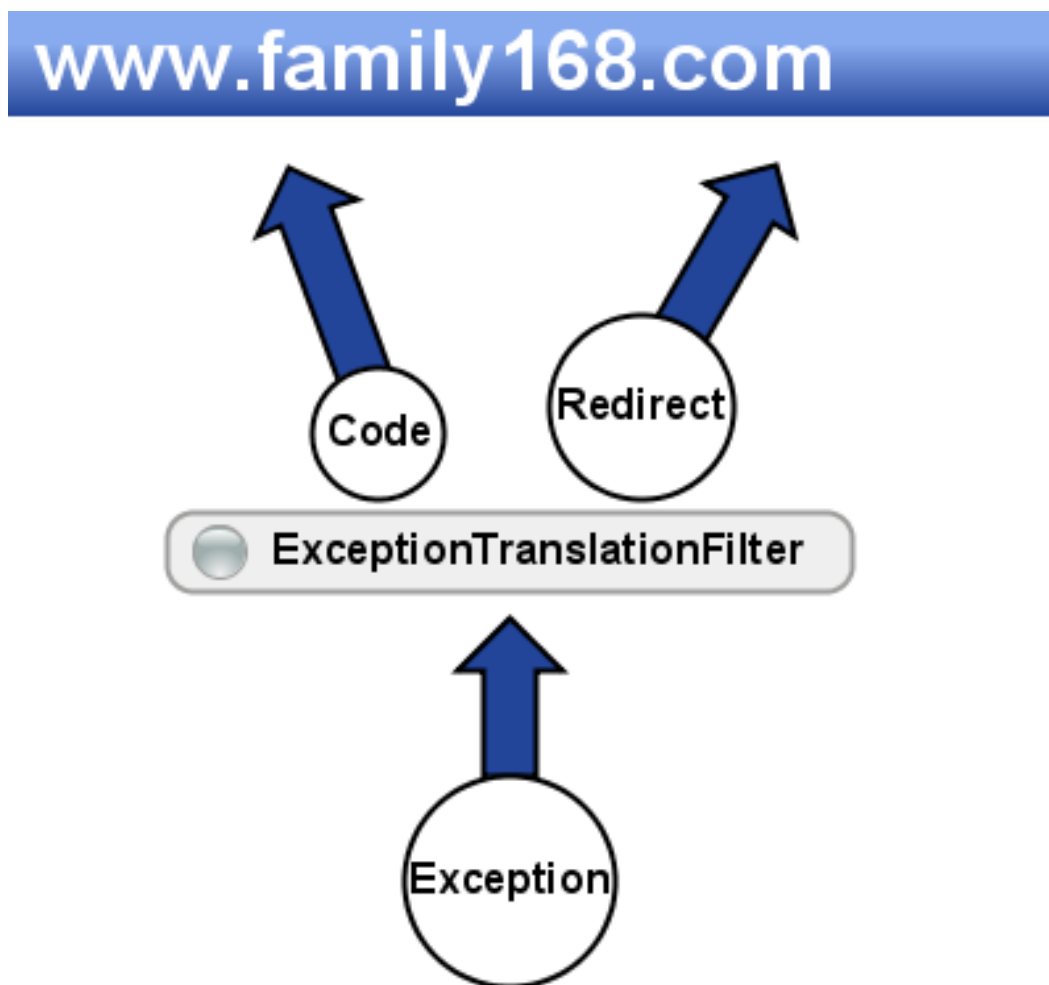


图 7.10. `org.springframework.security.ui.ExceptionTranslationFilter`

此过滤器的作用是处理中 `FilterSecurityInterceptor` 抛出的异常，然后将请求重定向到对应页面，或返回对应的响应错误代码。

7.10. `SessionFixationProtectionFilter`



图 7.11. org.springframework.security.ui.SessionFixationProtectionFilter

防御会话伪造攻击。有关防御会话伪造的详细信息，可以参考：[第 14 章 防御会话伪造](#)。

7.11. FilterSecurityInterceptor

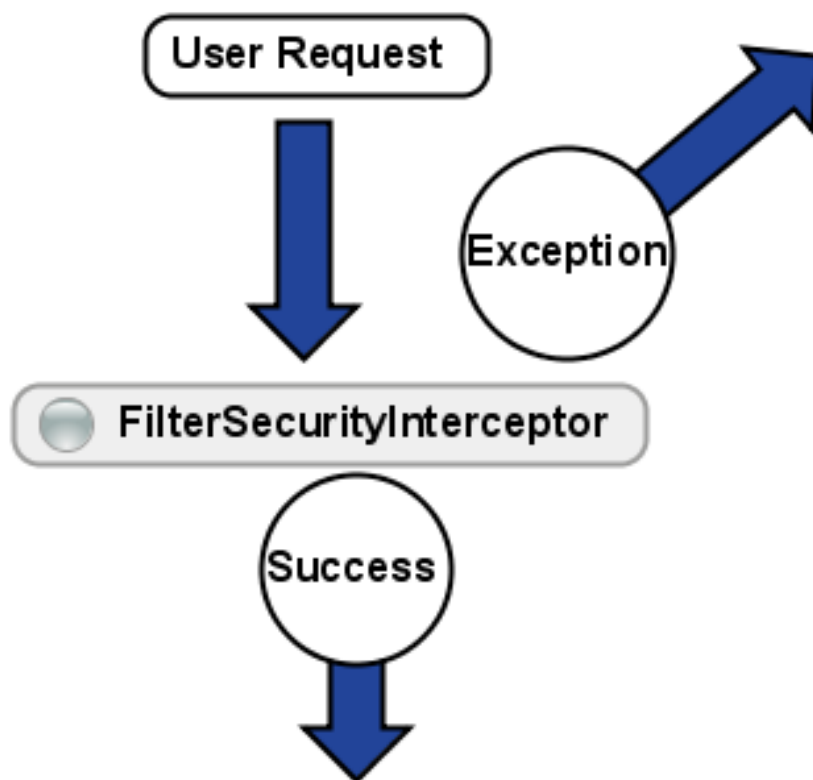


图 7.12. org.springframework.security.intercept.web.FilterSecurityInterceptor

用户的权限控制都包含在这个过滤器中。

功能一：如果用户尚未登陆，则抛出`AuthenticationCredentialsNotFoundException`“尚未认证异常”。

功能二：如果用户已登录，但是没有访问当前资源的权限，则抛出`AccessDeniedException`“拒绝访问异常”。

功能三：如果用户已登录，也具有访问当前资源的权限，则放行。

至此，我们完全展示了默认情况下Spring Security中使用到的过滤器，以及每个过滤器的应用场景和显示功能，下面我们会对这些过滤器的配置和用法进行逐一介绍。

第8章 管理会话

多个用户不能使用同一个账号同时登陆系统。

8.1. 添加监听器

在web.xml中添加一个监听器，这个监听器会在session创建和销毁的时候通知Spring Security。

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

这种监听session生命周期的监听器主要用来收集在线用户的信息，比如统计在线用户数之类的事。有关如何自己编写listener统计在线用户数，可以参考：<http://family168.com/tutorial/jsp/html/jsp-ch-08.html>

8.2. 添加过滤器

在xml中添加控制同步session的过滤器。

```
<http auto-config='true'>  
  <intercept-url pattern="/admin.jsp" access="ROLE_ADMIN" />  
  <intercept-url pattern="/*" access="ROLE_USER" />  
  <concurrent-session-control/>  
</http>
```

因为Spring Security的作者不认为控制会话是一个大家都经常使用的功能，所以concurrent-session-control没有包含在默认生成的过滤器链中，在我们需要使用它的时候，需要自己把它添加到http元素中。

这个concurrent-session-control对应的过滤器类是org.springframework.security.concurrent.ConcurrentSessionFilter，它的排序代码是100，它会被放在过滤器链的最顶端，在所有过滤器使用之前起作用。

8.3. 控制策略

8.3.1. 后登陆的将先登录的踢出系统

默认情况下，后登陆的用户会把先登录的用户踢出系统。

想测试一下的话，先打开firefox使用user/user登陆系统，然后再打开ie使用user/user登陆系统。这时ie下的user用户会登陆成功，进入登陆成功页面。而firefox下的用户如何刷新页面，就会显示如下信息：

```
This session has been expired (possibly due to multiple concurrent logins being attempted as the same user).
```

这是因为先登录的用户已经被强行踢出了系统，如果他再次使用user/user登陆，ie下的用户也会被踢出系统了。

8.3.2. 后面的用户禁止登陆

如果不想让之前登录的用户被自动踢出系统，需要为concurrent-session-control设置一个参数。

```
<http auto-config='true'>
  <intercept-url pattern="/admin.jsp" access="ROLE_ADMIN" />
  <intercept-url pattern="/*" access="ROLE_USER" />
  <concurrent-session-control exception-if-maximum-exceeded="true"/>
</http>
```

这个参数用来控制是否在会话数目超过最大限制时抛出异常，默认值是false，也就是不抛出异

常，而是把之前的session都销毁掉，所以之前登陆的用户就会被踢出系统了。

现在我们把这个参数改为true，再使用同一个账号同时登陆一下系统，看看会发生什么现象。

Your login attempt was not successful, try again.

Reason: Maximum sessions of {0} for this principal exceeded

Login with Username and Password

User:

Password:

☐ Remember me on this computer.

图 8.1. 禁止同一账号多次登录

很好，现在只要有一个人使用user/user登陆过系统，其他人就不能再次登录了。这样可能出现一个问题，如果有人登陆的时候因为某些问题没有进行logout就退出了系统，那么他只能等到session过期自动销毁之后，才能再次登录系统。

实例在ch102。

[上一页](#)

第7章 图解过滤器

[上一级](#)

[起始页](#)

[下一页](#)

第9章 单点登录

第9章 单点登录

所谓单点登录, SSO(Single Sign On), 就是把N个应用的登录系统整合在一起, 这样一来无论用户登录了任何一个应用, 都可以直接以登录过的身份访问其他应用, 不用每次访问其他系统再去登陆一遍了。

Spring Security没有实现自己的SSO, 而是整合了耶鲁大学单点登陆(JA-SIG), 这是当前使用很广泛的一种SSO实现, 它是基于中央认证服务CAS(Center Authentication Service)的结构实现的, 可以访问它们的官方网站获得更详细的信息<http://www.jasig.org/cas>。

在了解过这些基础知识之后, 我们可以开始研究如何使用Spring Security实现单点登录了。

9.1. 配置JA-SIG

从JA-SIG的官方网站下载cas-server, 本文写作时的最新稳定版为3.3.2。 <http://www.ja-sig.org/downloads/cas/cas-server-3.3.2-release.zip>。

将下载得到的cas-server-3.3.2-release.zip文件解压后, 可以得到一大堆的目录和文件, 我们这里需要的是modules目录下的cas-server-webapp-3.3.2.war。

把cas-server-webapp-3.3.2.war放到ch09\server目录下, 然后执行run.bat就可启动CAS中央认证服务器。

我们已在pom.xml中配置好了启用SSL所需的配置, 包括使用的server.jks和对应密码, 之后我们可以通过<https://localhost:9443/cas/login>访问CAS中央认证服务器。

Central Authentication Service (CAS)

请输入您的用户名和密码。

用户名:

密 码:

☐ 转向其他站点前提示我。

登录

重置

出于安全考虑，一旦您访问过那些需要您提供凭证信息的应用时，请操作完成之后关闭浏览器。

Languages:

[English](#) | [Spanish](#) | [French](#) | [Russian](#) | [Nederlands](#) | [Svenskt](#) | [Italiano](#) | [Urdu](#) | [Chinese \(Simplified\)](#) | [Deutsch](#) | [Japanese](#) | [Croatian](#) | [Czech](#) | [Slovenian](#) | [Polish](#) | [Turkish](#)

Copyright © 2005-2007 JA-SIG. All rights reserved.

Powered by [JA-SIG Central Authentication Service 3.3.2](#)

图 9.1. 登陆页面

默认情况下，只要输入相同的用户名和密码就可以登陆系统，比如我们使用user/user进行登陆。

Central Authentication Service (CAS)



登录成功

您已成功登录中央认证系统。

出于安全考虑，一旦您访问过那些需要您提供凭证信息的应用时，请操作完成之后关闭浏览器。

Copyright © 2005-2007 JA-SIG. All rights reserved.

Powered by [JA-SIG Central Authentication Service 3.3.2](#)

图 9.2. 登陆成功

这就证明中央认证服务器已经跑起来了。下一步我们来配置Spring Security，让它通过中央认证服务器进行登录。

9.2. 配置Spring Security

9.2.1. 添加依赖

首先要添加对cas的插件和cas客户端的依赖库。因为我们使用了maven2，所以只需要在pom.xml中添加一个依赖即可。

```
<dependency>
<groupId>org.springframework.security</groupId>
<artifactId>spring-security-cas-client</artifactId>
<version>2.0.4</version>
</dependency>
```


如果有人很不幸的没有使用maven2，那么就需要手工把去下面这些依赖库了。

```
spring-security-cas-client-2.0.4.jar
spring-dao-2.0.8.jar
aopalliance-1.0.jar
cas-client-core-3.1.3.jar
```

大家可以去spring和ja-sig的网站去寻找这些依赖库。

9.2.2. 修改applicationContext.xml

首先修改http部分。

```
<http auto-config='true' entry-point-ref="casProcessingFilterEntryPoint">❶
  <intercept-url pattern="/admin.jsp" access="ROLE_ADMIN" />
  <intercept-url pattern="/index.jsp" access="ROLE_USER" />
  <intercept-url pattern="/" access="ROLE_USER" />
  <logout logout-success-url="/cas-logout.jsp"/>❷
</http>
```

- ❶ 添加一个entry-point-ref引用cas提供的casProcessingFilterEntryPoint，这样在验证用户登录时就用上cas提供的机制了。
- ❷ 修改注销页面，将注销请求转发给cas处理。

```
<a href="https://localhost:9443/cas/logout">Logout of CAS</a>
```

然后要提供userService和authenticationManager，二者会被注入到cas的类中用来进行登录之后的用户授权。

```
<user-service id="userService">❶
  <user name="admin" password="admin" authorities="ROLE_USER, ROLE_ADMIN" />
  <user name="user" password="user" authorities="ROLE_USER" />
</user-service>

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

- ❶ 为了演示方便，我们将用户信息直接写在了配置文件中，之后cas的类就可以通过id获得userService，以此获得其中定义的用户信息和对应的权限。
- ❷ 对于authenticationManager来说，我们没有创建一个新实例，而是使用了“别名”（alias），这是因为在之前的namespace配置时已经自动生成了authenticationManager的实例，cas只需要知道这个实例的别名就可以直接调用。

创建cas的filter, entryPoint, serviceProperties和authenticationProvider。

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

<beans:bean id="casProcessingFilterEntryPoint"
  class="org.springframework.security.ui.cas.CasProcessingFilterEntryPoint">
  <beans:property name="loginUrl" value="https://localhost:9443/cas/login" />❷
  <beans:property name="serviceProperties" ref="casServiceProperties" />
</beans:bean>

<beans:bean id="casServiceProperties" class="org.springframework.security.ui.cas.ServiceProperties">
  <beans:property name="service" value="https://localhost:8443/ch09/j_spring_cas_security_check"/>❸
  <beans:property name="sendRenew" value="false"/>
</beans:bean>

<beans:bean id="casAuthenticationProvider"
  class="org.springframework.security.providers.cas.CasAuthenticationProvider">
  <custom-authentication-provider />❹
  <beans:property name="userDetailsService" ref="userService" />
  <beans:property name="serviceProperties" ref="casServiceProperties" />
  <beans:property name="ticketValidator">
    <beans:bean class="org.jasig.cas.client.validation.Cas20ServiceTicketValidator">
      <beans:constructor-arg index="0" value="https://localhost:9443/cas" />❺
    </beans:bean>
  </beans:property>
  <beans:property name="key" value="ch09" />
</beans:bean>
```

- ❶ casProcessingFilter最终是要放到Spring security的安全过滤器链中才能发挥作用的。这里使用的customer-filter就会把它放到CAS_PROCESSING_FILTER位置的后面。

这个位置具体是在LogoutFilter和AuthenticationProcessingFilter之间，这样既不会影响注销操作，又可以在用户进行表单登陆之前拦截用户请求进行cas认证了。
- ❷ 当用户尚未登录时，会跳转到这个cas的登录页面进行登录。
- ❸ 用户在cas登录成功后，再次跳转回原系统时请求的页面。

CasProcessingFilter会处理这个请求，从cas获得已登录的用户信息，并对用户进行授权。
- ❹ 使用custom-authentication-provider之后，Spring Security其他的权限模块会从这个bean中获得权限验证信息。
- ❺ 系统需要验证当前用户的tickets是否有效。

经过了这么多的配置，我们终于把cas功能添加到spring security中了，看着一堆堆一串串的配置文件，好似又回到了acegi的时代，可怕啊。

下面运行系统，尝试使用了cas的权限控制之后有什么不同。

9.3. 运行配置了cas的子系统

首先要保证cas中央认证服务器已经启动了。子系统的pom.xml中也已经配置好了SSL，所以可以进入ch09执行run.bat启动子系统。

现在直接访问<http://localhost:8080/ch09/>不再会弹出登陆页面，而是会跳转到cas中央认证服务器上登录。

Central Authentication Service (CAS)

请输入您的用户名和密码。

用户名：

密 码：

☐ 转向其他站点前提示我。

出于安全考虑，一旦您访问过那些需要您提供凭证信息的应用时，请操作完成之后关闭浏览器。

Languages:

[English](#) | [Spanish](#) | [French](#) | [Russian](#) | [Nederlands](#) | [Svenskt](#) | [Italiano](#) | [Urdu](#) | [Chinese \(Simplified\)](#) | [Deutsch](#) | [Japanese](#) | [Croatian](#) | [Czech](#) | [Slovenian](#) | [Polish](#) | [Turkish](#)

Copyright © 2005-2007 JA-SIG. All rights reserved.

Powered by [JA-SIG Central Authentication Service 3.3.2](#)

图 9.3. 登陆页面

输入user/user后进行登录，系统不会做丝毫的停留，直接跳转回我们的子系统，这时我们已经登录到系统中了。

`username : user`

[admin.jsp](#) [logout](#)

图 9.4. 登陆成功

我们再来试试注销，点击logout会进入cas-logout.jsp。

Do you want to log out of CAS?

You have logged out of this application, but may still have an active single-sign on session with CAS.

[Logout of CAS](#)

图 9.5. cas-logout.jsp

在此点击Logout of CAS会跳转至cas进行注销。

JA-SIG

Central Authentication Service (CAS)



注销成功

您已成功退出CAS系统，谢谢使用！
出于安全考虑，请关闭您的浏览器。

Copyright © 2005-2007 JA-SIG. All rights reserved.

Powered by [JA-SIG Central Authentication Service 3.3.2](#)

JA-SIG

图 9.6. 注销成功

现在我们完成了Spring Security中cas的配置，enjoy it。

9.4. 为cas配置SSL

在使用cas的时候，我们要为cas中央认证服务器和子系统都配置上SSL，以此来对他们之间交互的数据进行加密。这里我们将使用JDK中包含的keytool工具生成配置SSL所需的密钥。

9.4.1. 生成密钥

首先生成一个key store。

```
keytool -genkey -keyalg RSA -dname "cn=localhost,ou=family168,o=www.family168.com,l=china,st=beijing,c=cn" -alias server  
-keypass password -keystore server.jks -storepass password
```

我们会得到一个名为server.jks的文件，它的密码是password，注意cn=localhost部分，这里必须与cas服务器的域名一致，而且不能使用ip，因为我们是在本地localhost测试cas，所以这里设置的就是cn=localhost，在实际生产环境中使用时，要将这里配置为cas服务器的实际域名。

导出密钥

```
keytool -export -trustcacerts -alias server -file server.cer -keystore server.jks -storepass password
```

将密钥导入JDK的cacerts

```
keytool -import -trustcacerts -alias server -file server.cer -keystore D:/apps/jdk1.5.0_15/jre/lib/security/cacerts -storepass password
```

这里需要把使用实际JDK的安装路径，我们要把密钥导入到JDK的cacerts中。

我们在ch09/certificates/下放了一个genkey.bat，这个批处理文件中已经包含了上述的所有命令，运行它就可以生成我们所需的密钥。

9.4.2. 为jetty配置SSL

jetty的配置可参考ch09中的pom.xml文件。

```
<connectors>
```

```

<connector implementation="org.mortbay.jetty.security.SslSocketConnector">
  <port>9443</port>
  <keystore>../certificates/server.jks</keystore>
  <password>password</password>
  <keyPassword>password</keyPassword>
  <truststore>../certificates/server.jks</truststore>
  <trustPassword>password</trustPassword>
  <wantClientAuth>true</wantClientAuth>
  <needClientAuth>false</needClientAuth>
</connector>
</connectors>
<systemProperties>
  <systemProperty>
    <name>javax.net.ssl.trustStore</name>
    <value>../certificates/server.jks</value>
  </systemProperty>
  <systemProperty>
    <name>javax.net.ssl.trustStorePassword</name>
    <value>password</value>
  </systemProperty>
</systemProperties>

```

9.4.3. 为tomcat配置SSL

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

```

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

```

如果你希望客户端没有提供证书的时候SSL链接也能成功，也可以把clientAuth设置成want。

实例在ch103。

[上一页](#)

第 8 章 管理会话

[上一级](#)

[起始页](#)

[下一页](#)

第 10 章 basic 认证

第 10 章 basic 认证

basic 认证是另一个常用的认证方式，与表单认证不同的是，basic 认证常用于无状态客户端的验证，比如 HttpInvoker 或者 Web Service 的认证，这种场景的特点是客户端每次访问应用时，都在请求头部携带认证信息，一般就是用户名和密码，因为 basic 认证会传递明文，所以最好使用 https 传输数据。

10.1. 配置 basic 验证

如果在 http 中配置了 auto-config="true" 我们就无需再添加任何配置了，默认配置中已经包含了 Basic 认证功能。但是这同时也会激活 form-login，因此我们将演示仅有 basic 验证的场景，为此需要去掉配置文件中的 auto-config="true"。

```
<http auto-config="true">
  <http-basic />
  <intercept-url pattern="/admin.jsp" access="ROLE_ADMIN" />
  <intercept-url pattern="/" access="ROLE_USER" />
</http>
```

删除了 auto-config="true" 之后，还要记得添加 http-basic 标签，这样我们的系统将仅仅使用 basic 认证方式来实现用户登录。

现在我们访问系统时，不会再进入之前的登录页面，而是会显示浏览器原生的登录对话框。

```
User-Agent Mozilla/5.0 (Windows; U; Windows NT 5.1; zh-CN; rv:1.9.1) Gecko/20090624 Firefox/3.5
Accept text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language zh-CN,zh;q=0.7,zh-cn;q=0.3
Accept-Encoding gzip,deflate
Accept-Charset UTF-8,*
Keep-Alive 300
Connection keep-alive
Referer http://localhost:8080/
Authorization Basic dXNlcjpic2V7
Cookie JSESSIONID=1letjbranvm4g
```

3 个请求	236 B	7.94s
-------	-------	-------

图 10.1. basic 登录

登录成功之后，我们可以在HTTP请求头部看到basic验证所需的属性Authorization。

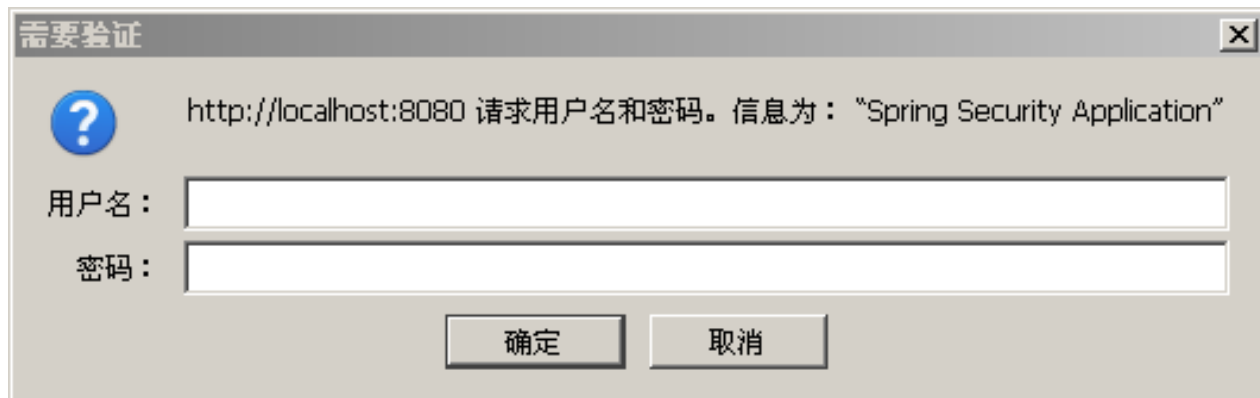


图 10.2. HTTP 请求头

最后需要注意的是，因为basic认证不使用session，所以无法与rememberMe功用。

10.2. 编程实现basic客户端

下面我们来示范一下如何使用basic认证。假设我们在basic.jsp中需要远程调用http://localhost:8080/ch10/admin.jsp的内容。这时为了能够通过Spring Security的权限检测，我们需要在请求的头部加上basic所需的认证信息。

```
String username = "admin";
String password = "admin";
byte[] token = (username + ":" + password).getBytes("utf-8");
String authorization = "Basic " + new String(Base64.encodeBase64(token), "utf-8");❶

URL url = new URL("http://localhost:8080/ch10/admin.jsp");
URLConnection conn = (URLConnection) url.openConnection();
conn.setRequestProperty("Authorization", authorization);❷
```

❶ 我们先将用户名和密码拼接成一个字符串，两者之间使用“:”分隔。

然后使用commons-codec的Base64将这个字符串加密。在进行basic认证的时候Spring Security会使用commons-codec把这段字符串反转成用户名和密码，再进行认证操作。

下一步为加密后得到的字符串添加一个前缀"Basic"，这样Spring Security就可以通过这个判断客户端是否使用了basic认证。

❷ 将上面生成的字符串设置到请求头部，名称为“Authorization”。Spring Security会在认证时，获取头部信息进行判断。

有关basic代码可以在/ch10/basic.jsp找到，可以运行ch10，然后访问http://localhost:8080/ch10/basic.jsp。它会使用上述的代码，通过Spring Security的认证，成功访问到admin.jsp的信息。

实例在ch104。

[上一页](#)

第 9 章 单点登录

[上一级](#)

[起始页](#)

[下一页](#)

第 11 章 标签库

第 11 章 标签库

Spring Security提供的标签库，主要的用途是为了在视图层直接访问用户信息，再者就是为了对显示的内容进行权限管理。

11.1. 配置taglib

如果需要使用taglib，首先要把spring-security-taglibs-2.0.4.jar放到项目的classpath下，这在文档附带的实例中已经配置好了依赖。剩下的只要在jsp上添加taglib的定义就可以使用标签库了。

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

11.2. authentication

authentication的功能是从SecurityContext中获得一些权限相关的信息。

可以用它获得当前登陆的用户名：

```
<sec:authentication property="name"/>
```

获得当前用户所有的权限，把权限列表放到authorities变量里，然后循环输出权限信息：

```
<sec:authentication property="authorities" var="authorities" scope="page"/>
<c:forEach items="${authorities}" var="authority">
    ${authority.authority}
```

```
</c:forEach>
```

11.3. authorize

authorize用来判断当前用户的权限，然后根据指定的条件判断是否显示内部的内容。

```
<sec:authorize ifAllGranted="ROLE_ADMIN,ROLE_USER">❶
```

```
  admin and user
```

```
</sec:authorize>
```

```
<sec:authorize ifAnyGranted="ROLE_ADMIN,ROLE_USER">❷
```

```
  admin or user
```

```
</sec:authorize>
```

```
<sec:authorize ifNotGranted="ROLE_ADMIN">❸
```

```
  not admin
```

```
</sec:authorize>
```

- ❶ ifAllGranted，只有当前用户同时拥有ROLE_ADMIN和ROLE_USER两个权限时，才能显示标签内部内容。
- ❷ ifAnyGranted，如果当前用户拥有ROLE_ADMIN或ROLE_USER其中一个权限时，就能显示标签内部内容。
- ❸ ifNotGranted，如果当前用户没有ROLE_ADMIN时，才能显示标签内部内容。

11.4. acl/accesscontrollist

用于判断当前用户是否拥有指定的acl权限。

```
<sec:accesscontrollist domainObject="{item}" hasPermission="8,16">
```

```
  |
```

```
  <a href="message.do?action=remove&id={item.id}">Remove</a>
```

```
</sec:accesscontrollist>
```

我们将当前显示的对象作为参数传入acl标签，然后指定判断的权限为8(删除)和16(管理)，当前用

户如果拥有对这个对象的删除和管理权限时，就会显示对应的remove超链接，用户才可以通过此链接对这条记录进行删除操作。

关于ACL的知识，请参考[???](#)。

11.5. 为不同用户显示各自的登陆成功页面

一个常见的需求是，普通用户登录之后显示普通用户的工作台，管理员登陆之后显示后台管理页面。这个功能可以使用taglib解决。

其实只要在登录成功后的jsp页面中使用taglib判断当前用户拥有的权限进行跳转就可以。

```
<%@ taglib prefix="sec" uri="http://www.springframework.org/security/tags" %>
<sec:authorize ifAllGranted="ROLE_ADMIN">❶
    <%response.sendRedirect("admin.jsp");%>
</sec:authorize>
<sec:authorize ifNotGranted="ROLE_ADMIN">❷
    <%response.sendRedirect("user.jsp");%>
</sec:authorize>
```

❶ 当用户拥有ROLE_ADMIN权限时，既跳转到admin.jsp显示管理后台。

❷ 当用户没有ROLE_ADMIN权限时，既跳转到user.jsp显示普通用户工作台。

这里我们只做最简单的判断，只区分当前用户是否为管理员。可以根据实际情况做更加复杂的跳转，当用户具有不同权限时，跳到对应的页面，甚至可以根据用户username跳转到各自的页面。

实例在ch105。

[上一页](#)[第 10 章 basic认证](#)[上一级](#)[起始页](#)[下一页](#)[第 12 章 自动登录](#)

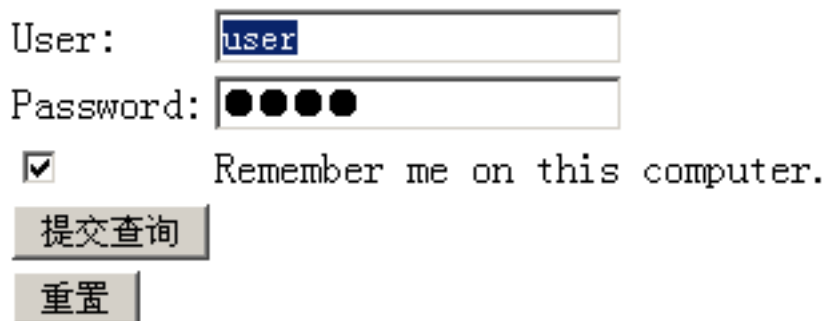
第 12 章 自动登录

如果用户一直使用同一台电脑上网，那么他可能希望不要每次上网都要进行登录这道程序，他们希望系统可以记住自己一段时间，这样用户就可以无需登录直接登录系统，使用其中的功能。rememberMe就给我们提供了这样一种便捷途径。

12.1. 默认策略

在配置文件中使用auto-config="true"就会自动启用rememberMe，之后，只要用户在登录时选中checkbox就可以实现下次无需登录直接进入系统的功能。

Login with Username and Password



User:

Password:

☒ Remember me on this computer.

图 12.1. 选中rememberMe

默认有效时间是两周，启用rememberMe之后的两周内，用户都可以直接跳过系统，直接进入系统。

实际上，Spring Security中的rememberMe是依赖cookie实现的，当用户在登录时选择使用rememberMe，系统就会在登录成功后将为用户生成一个唯一标识，并将这个标识保存进cookie

中，我们可以通过浏览器查看用户电脑中的cookie。

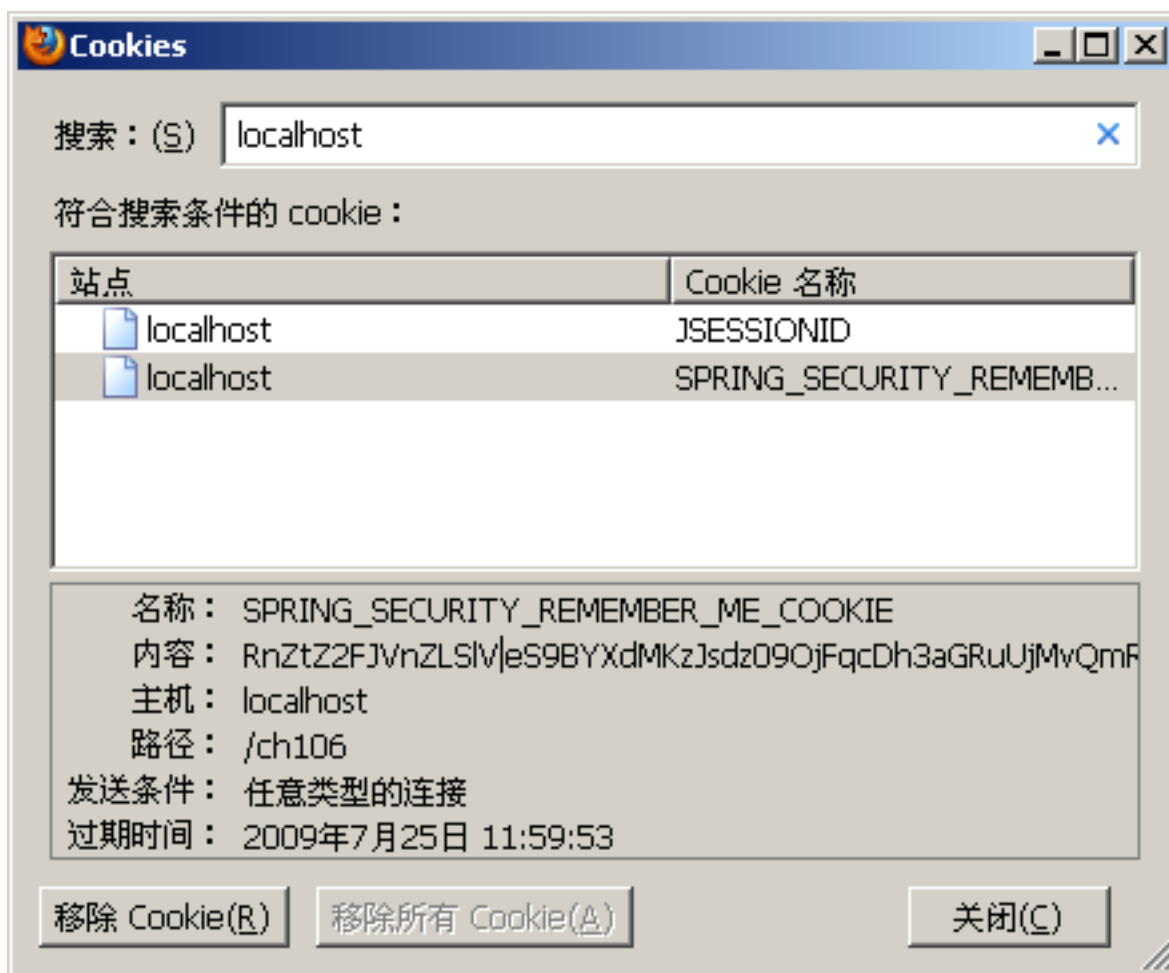


图 12.2. rememberMe cookie

从上图中，我们可以看到Spring Security生成的cookie名称是SPRING_SECURITY_REMEMBER_ME_COOKIE，它的内容是一串加密的字符串，当用户再次访问系统时，Spring Security将从这个cookie读取用户信息，并加以验证。如果可以证实cookie有效，就会自动将用户登录到系统中，并为用户授予对应的权限。

12.2. 持久化策略

rememberMe的默认策略会将username和过期时间保存到客户主机上的cookie中，虽然这些信息都已经进行过加密处理，不过我们还可以使用安全级别更高的持久化策略。在持久化策略中，客户主机cookie中保存的不再用username，而是由系统自动生成的序列号，在验证时系统会将客户cookie中保存的序列号与数据库中保存的序列号进行比对，以确认客户请求的有效性，之后在比对成功后才会从数据库中取出对应的客户信息，继续进行认证和授权等工作。这样即使客户本地的cookie遭到破解，攻击者也只能获得一个序列号，而不是用户的登录账号。

如果希望使用持久化策略，我们需要先在数据库中创建rememberMe所需的表。

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

然后要为配置文件中添加与数据库的链接。

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

最后修改http中的配置，为remember-me添加data-source-ref即可，Spring Security会在初始化时判断是否存在data-source-ref属性，如果存在就会使用持久化策略，否则会使用上述的默认策略。

```
<http auto-config='true'>
  <intercept-url pattern="/admin.jsp" access="ROLE_ADMIN" />
  <intercept-url pattern="/*" access="ROLE_USER" />
  <remember-me data-source-ref="dataSource"/>
</http>
```



注意

默认策略和持久化策略是不能混用的，如果你首先在应用中使用过默认策略的rememberMe，未等系统过期便换成了持久化策略，之前保留的cookie也无法通过系统验证，实际上系统会将cookie当做无效标识进行清除。同样的，持久化策略中生成的cookie也无法用在默认策略下。

实例在ch106。

[上一页](#)

第 11 章 标签库

[上一级](#)

[起始页](#)

[下一页](#)

第 13 章 匿名登录

第 13 章 匿名登录

匿名登录，即用户尚未登录系统，系统会为所有未登录的用户分配一个匿名用户，这个用户也拥有自己的权限，不过他是不能访问任何被保护资源的。

设置一个匿名用户的好处是，我们在进行权限判断时，可以保证SecurityContext中永远是存在着一个权限主体的，启用了匿名登录功能之后，我们所需要做的工作就是从SecurityContext中取出权限主体，然后对其拥有的权限进行校验，不需要每次去检验这个权限主体是否为空了。这样做的好处是我们永远认为请求的主体是拥有权限的，即便他没有登录，系统也会自动为他赋予未登录系统角色的权限，这样后面所有的安全组件都只需要在当前权限主体上进行处理，不用一次一次的判断当前权限主体是否存在。这就更容易保证系统中操作的一致性。

13.1. 配置文件

在配置文件中使用auto-config="true"就会启用匿名登录功能。在启用匿名登录之后，如果我们希望允许未登录就可以访问一些资源，可以在进行如下配置。

```
<http auto-config='true'>
  <intercept-url pattern="/" access="IS_AUTHENTICATED_ANONYMOUSLY" />
  <intercept-url pattern="/admin.jsp" access="ROLE_ADMIN" />
  <intercept-url pattern="/*" access="ROLE_USER" />
</http>
```

在access中指定IS_AUTHENTICATED_ANONYMOUSLY后，系统就知道此资源可以被匿名用户访问了。当未登录时访问系统的“/”，就会被自动赋以匿名用户的身份。我们可以使用taglib获得用户的权限主体信息。

```
<div>
```

```

username : <sec:authentication property="name"/>
|
authorities: <sec:authentication property="authorities" var="authorities" scope="page"/>
<c:forEach items="${authorities}" var="authority">
    ${authority.authority}
</c:forEach>
</div>

```

当用户访问系统时，就会看到如下信息，这时他还没有进行登录。

```

username : roleAnonymous | authorities: ROLE_ANONYMOUS

```

[admin.jsp](#) [logout](#)

图 13.1. 匿名登录

这里显示的是分配给所有未登录用户的一个默认用户名roleAnonymMous，拥有的权限是ROLE_ANONYMOUS。我们可以看到系统已经把匿名用户当做了合法有效的用户进行处理，可以获得它的用户名和拥有的权限，而不需判断SecurityContext中是否为空。

实际上，我们完全可以把匿名用户像一个正常用户那样进行配置，我们可以在配置文件中直接使用ROLE_ANONYMOUS指定它可以访问的资源。

```

<http auto-config='true'>
  <intercept-url pattern="/" access="ROLE_ANONYMOUS" />
  <intercept-url pattern="/admin.jsp" access="ROLE_ADMIN" />
  <intercept-url pattern="/*" access="ROLE_USER" />
</http>

```

不过，为了更明显的将匿名用户与系统中的其他用户区分开，我们推荐在配置时尽量使用IS_AUTHENTICATED_ANONYMOUSLY来指定匿名用户可以访问的资源。

13.2. 修改默认用户名

我们通常可以看到这种情况，当一个用户尚未登录系统时，在页面上应当显示用户名的部分显示的是“游客”。这样操作更利于提升客户体验，他看到自己即使没有登录也可以使用“游客”的

身份享受系统的服务，而且使用了“游客”作为填充内容，也避免了原本显示用户名部分留空，影响页面布局。

如果没有匿名登录的功能，我们就被迫要在所有显示用户名的部分，判断当前用户是否登录，然后根据登录情况显示登录用户名或默认的“游客”字符。匿名登录功能帮我们解决了这个问题，既然未登录用户都拥有匿名用户的身份，那么在显示用户名时就不必去区分用户登录状态，直接显示当前权限主体的名称即可。

我们需要做的只是为匿名设置默认的用户名而已，默认的名称`roleAnonymous`可以通过配置文件中的`anonymous`元素修改。

```
<http auto-config='true'>
  <intercept-url pattern="/" access="IS_AUTHENTICATED_ANONYMOUSLY" />
  <intercept-url pattern="/admin.jsp" access="ROLE_ADMIN" />
  <intercept-url pattern="/*" access="ROLE_USER" />
  <anonymous username="Guest"/>
</http>
```

这样，匿名用户的默认名称就变成了“Guest”，我们在页面上依然使用相同的`taglib`显示用户名即可。

```
username : Guest | authorities: ROLE_ANONYMOUS
-----
admin.jsp logout
```

图 13.2. 修改匿名用户名称

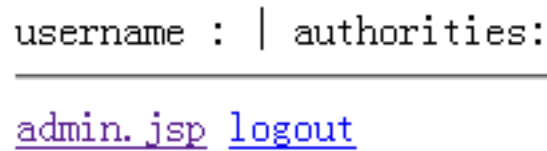
13.3. 匿名用户的限制

虽然匿名用户无论在配置授权时，还是在获取权限信息时，都与已登录用户的操作一模一样，但它应该与未登录用户是等价，当我们以匿名用户的身份进入“/”后，点击`admin.jsp`链接，系统会像处理未登录用户时一样，跳转到登录用户，而不是像处理以登录用户时显示拒绝访问页面。

但是匿名用户与未登录用户之间也有很大的区别，比如，我们将“/”设置为不需要过滤器保护，而不是设置匿名用户。

```
<http auto-config='true'>
  <intercept-url pattern="/" filters="none" />
  <intercept-url pattern="/admin.jsp" access="ROLE_ADMIN" />
  <intercept-url pattern="/*" access="ROLE_USER" />
</http>
```

filters="none"表示当我们访问“/”时，是不会使用任何一个过滤器去处理这个请求的，它可以实现无需登录即可访问资源的效果，但是因为没有使用过滤器对请求进行处理，所以也无法利用安全过滤器为我们带来的好处，最简单的，这时SecurityContext内再没有保存任何一个权限主体了，我们也无法从中取得主体名称以及对应的权限信息。



```
username : | authorities:
admin.jsp logout
```

图 13.3. 跳过过滤器的情况

因此，使用filters="none"忽略所有过滤器会提升性能，而是用匿名登录功能可以实现权限操作的一致性，具体应用时还需要大家根据实际情况自行选择了。

实例在ch107。

[上一页](#)

第 12 章 自动登录

[上一级](#)

[起始页](#)

[下一页](#)

第 14 章 防御会话伪造

第 14 章 防御会话伪造

14.1. 攻击场景

session fixation会话伪造攻击是一个蛮婉转的过程。

比如，当我要是使用session fixation攻击你的时候，首先访问这个网站，网站会创建一个会话，这时我可以把附有jsessionId的url发送给你。

```
http://unsafe/index.jsp;jsessionId=1pjztz08i2u4i
```

你使用这个网址访问网站，结果你和我就会公用同一个jsessionId了，结果就是在服务器中，我们两人使用的是同一个session。

这时我只要祈求你在session过期之前登陆系统，然后我就可以使用jsessionId直接进入你的后台了，然后可以使用你在系统中的授权做任何事情。

简单来说，我创建了一个session，然后把jsessionId发给你，你傻乎乎的就使用我的session进行了登陆，结果等于帮我的session进行了授权操作，结果就是我可以使用一开始创建的session进入系统做任何事情了。

与会话伪造的详细信息可以参考http://en.wikipedia.org/wiki/Session_fixation。

14.2. 解决会话伪造

解决session fix的问题其实很简单，只要在用户登录成功之后，销毁用户的当前session，并重新生

成一个session就可以了。

Spring Security默认就会启用session-fixation-protection，这会在登录时销毁用户的当前session，然后为用户创建一个新session，并将原有session中的所有属性都复制到新session中。

如果希望禁用session-fixation-protection，可以在http中将session-fixation-protection设置为none。

```
<http auto-config='true' session-fixation-protection="none">  
  <intercept-url pattern="/admin.jsp" access="ROLE_ADMIN" />  
  <intercept-url pattern="/*" access="ROLE_USER" />  
</http>
```

session-fixation-protection的值共有三个可供选择，none，migrateSession和newSession。默认使用的是migrateSession，如同我们上面所讲的，它会将原有session中的属性都复制到新session中。上面我们也见到了使用none来禁用session-fixation功能的场景，最后剩下的newSession会在用户登录时生成新session，但不会复制任何原有属性。

实例在ch108。

[上一页](#)[第 13 章 匿名登录](#)[上一级](#)[起始页](#)[下一页](#)[第 15 章 预先认证](#)

第 15 章 预先认证

预先认证是指用户在进入系统给钱，就已经通过某种机制进行过身份认证，请求中已经附带了身份认证的信息，这时我们只需要从获得这些身份认证信息，并对用户进行授权即可。CAS, X509等都属于这种情况。

Spring Security中专门为这种系统外预先认证的情况提供了工具类，这一章我们来看一下如何使用Pre-Auth处理使用容器Realm认证的用户。

15.1. 为jetty配置Realm

首先在pom.xml中配置jetty所需的Realm。

```
<userRealms>
  <userRealm implementation="org.mortbay.jetty.security.HashUserRealm">
    <name>Preauth Realm</name>
    <config>realm.properties</config>
  </userRealm>
</userRealms>
```

用户，密码，以及权限信息都保存在realm.properties文件中。

```
admin: admin,ROLE_ADMIN,ROLE_USER
user: user,ROLE_USER
test: test
```

我们配置了三个用户，分别是admin, user和test，其中admin拥有ROLE_ADMIN和ROLE_USER权限，user拥有ROLE_USER权限，而test没有任何权限。

下一步在src/webapp/WEB-INF/web.xml中配置登录所需的安全权限。

```
<login-config>
  <auth-method>BASIC</auth-method>
  <realm-name>Preauth Realm</realm-name>
</login-config>
```

```

<security-role>
  <role-name>ROLE_USER</role-name>
</security-role>
<security-role>
  <role-name>ROLE_ADMIN</role-name>
</security-role>
<security-constraint>
  <web-resource-collection>
    <web-resource-name>All areas</web-resource-name>
    <url-pattern>/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>ROLE_USER</role-name>
  </auth-constraint>
</security-constraint>

```

这里我们将login-config中的realm-name配置为Preauth Realm，这与刚刚在pom.xml中配置的名称是相同的。而后我们配置了两个安全权限ROLE_USER和ROLE_ADMIN，最后我们现在访问所有资源都需要使用ROLE_USER这个权限。

自此，服务器与应用中的Realm配置完毕，下一步我们需要使用Spring Security与Realm对接。

15.2. 配置Spring Security

因为使用容器Realm的Pre-Auth并没有对应的命名空间，所以我们只能使用传统方式，一个一个bean的配置了。

首先要配置springSecurityFilterChain，告诉Spring Security我们会使用哪些过滤器。

```

<bean id="springSecurityFilterChain" class="org.springframework.security.util.FilterChainProxy">
  <sec:filter-chain-map path-type="ant">
    <sec:filter-chain pattern="/*" filters="hscif,j2eePreAuthFilter,etf,fsi"/>
  </sec:filter-chain-map>
</bean>

```

因为将要使用j2eePreAuthFilter，所有默认那些form-login, basic-login, rememberMe都没了用武之地，这里我们只保留HttpSessionContextIntegrationFilter, J2eePreAuthenticatedProcessingFilter, ExceptionTranslationFilter和FilterSecurityInterceptor。其中HttpSessionContextIntegrationFilter用来将session中的用户信息放到SecurityContext中。ExceptionTranslationFilter和FilterSecurityInterceptor负责验证用户权限，并处理验证过程中的异常。

而为了使用j2eePreAuthFilter，我们需要进行如下配置：

```

<bean id="preAuthenticatedAuthenticationProvider"
  class="org.springframework.security.providers.preauth.PreAuthenticatedAuthenticationProvider">
  <sec:custom-authentication-provider />
  <property name="preAuthenticatedUserDetailsService" ref="preAuthenticatedUserDetailsService"/>
</bean>

```

```

<bean id="preAuthenticatedUserDetailsService"
  class="org.springframework.security.providers.preauth.PreAuthenticatedGrantedAuthoritiesUserDetailsService"/>

<bean id="j2eeMappableRolesRetriever"
  class="org.springframework.security.ui.preauth.j2ee.WebXmlMappableAttributesRetriever">
  <property name="webXmlInputStream">
    <bean factory-bean="webXmlResource" factory-method="getInputStream"/>
  </property>
</bean>

<bean id="webXmlResource" class="org.springframework.web.context.support.ServletContextResource">
  <constructor-arg ref="servletContext"/>
  <constructor-arg value="/WEB-INF/web.xml"/>
</bean>

<bean id="servletContext" class="org.springframework.web.context.support.ServletContextFactoryBean"/>

<bean id="j2eePreAuthFilter"
  class="org.springframework.security.ui.preauth.j2ee.J2eePreAuthenticatedProcessingFilter">
  <property name="authenticationManager" ref="authenticationManager"/>
  <property name="authenticationDetailsSource">
    <bean class="org.springframework.security.ui.preauth.j2ee.J2eeBasedPreAuthenticatedWebAuthenticationDetailsSource">
      <property name="mappableRolesRetriever" ref="j2eeMappableRolesRetriever"/>
      <property name="userRoles2GrantedAuthoritiesMapper">
        <bean class="org.springframework.security.authoritymapping.SimpleAttributes2GrantedAuthoritiesMapper">
          <property name="convertAttributeToUpperCase" value="true"/>
        </bean>
      </property>
    </bean>
  </property>
</bean>

<bean id="preauthEntryPoint"
  class="org.springframework.security.ui.preauth.PreAuthenticatedProcessingFilterEntryPoint"/>

```

这里，我们要配置Pre-Auth所需的AuthenticationProvider, EntryPoint, AuthenticatedUserDetailsService并最终组装成一个j2eePreAuthFilter。其中j2eeMappableRolesRetriever会读取我们之前配置的web.xml，从中获得权限信息。

这样，当用户登录时，请求会先被Realm拦截，并要求用户进行登录：



图 15.1. Realm登录

登录成功后，Realm会将用户身份信息绑定到请求中，j2eePreAuthFilter就会从请求中读取身份信息，结合web.xml中定义的权限信息对用户进行授权，并将授权信息录入SecurityContext，之后对用户验证时与之前已没有了任何区别。

这里的preauthEntryPoint会在用户权限不足时起作用，它只会简单返回一个401的拒绝访问响应。

在此我们并不推荐实际中使用这项功能，因为需要对容器进行配置，影响应用的灵活性。

实例在ch109。

[上一页](#)[第 14 章 防御会话伪造](#)[上一级](#)[起始页](#)[下一页](#)[第 16 章 切换用户](#)

第 16 章 切换用户

Spring Security提供了一种称为切换用户的机制，可以使管理员免于进过登录的操作，直接切换当前用户，从而改变当前的操作权限。因为按照责权分离的原则，系统内的超级管理员应该只有管理权限，而没有操作权限，所以为了在改变操作后可以测试系统的操作，需要降低权限才可以进入操作界面，这时就可以使用切换用户的功能。

16.1. 配置方式

在xml中添加SwitchUser的配置。

```
<beans:bean id="switchUserProcessingFilter"
    class="org.springframework.security.ui.switchuser.SwitchUserProcessingFilter">
    <custom-filter position="SWITCH_USER_FILTER" />
    <beans:property name="userDetailsService"
        ref="org.springframework.security.userdetails.memory.InMemoryDaoImpl" />
    <beans:property name="targetUrl" value="/index.jsp"/>
</beans:bean>
```

它需要引用系统中的userDetailsService在切换用户时，根据对应的username获得切换后用户的信息和权限，我们还要使用custom-filter将该过滤器放到过滤器链中，注意必须放在用来验证权限的FilterSecurityInterceptor之后，这样可以控制当前用户是否拥有切换用户的权限。

现在，我们可以在系统中使用切换用户这一功能了，我们可以通过/j_spring_security_switch_user?j_username=user切换到j_username指定的用户，这样可以快捷的获得目标用户的信息和权限。当需要返回管理员用户时，只需要通过/j_spring_security_exit_user就可以还原到切换前的状态。

16.2. 实例演示

现在我们进入实例，通过登录页面进行登录。因为实现了权责分离，admin/admin用户只能访问管理页面admin.jsp，不能访问user.jsp，user/user用户只能访问操作页面user.jsp，不能访问admin.jsp。

如果我们以管理员身份登录后，希望切换到user/user用户，可以调用/j_spring_security_switch_user?j_username=user切换到user/user用户，然后就可以使用user的权限访问user.jsp了。

在切换用户后，我们可以看到当前登录用户的权限中多了一个ROLE_PREVIOUS_ADMINISTRATOR，这其中就保存着前用户的权限信息，当我们通过/j_spring_security_exit_user退出切换用户模式时，系统就会从ROLE_PREVIOUS_ADMINISTRATOR中获得原始用户信息，重新进行授权。

实例在ch110。

[上一页](#)[第 15 章 预先认证](#)[上一级](#)[起始页](#)[下一页](#)[第 17 章 信道安全](#)

第 17 章 信道安全

17.1. 设置信道安全

为了加强安全级别，我们可以限制默写资源必须通过https协议才能访问。

```
<http auto-config='true'>
  <intercept-url pattern="/admin.jsp" access="ROLE_ADMIN" requires-channel="https"/>
  <intercept-url pattern="/*" access="ROLE_USER" />
</http>
```

可以为/admin.jsp单独设置必须使用https才能访问，如果用户使用了http协议访问该网址，系统会强制用户使用https 协议重新进行请求。

这里我们可以选使用https, http或者any三种数值，其中any为默认值，表示无论用户使用何种协议都可以访问资源。

17.2. 指定http和https的端口

因为http和https协议的访问端口不同，Spring Security在处理信道安全时默认会使用80/443和8080/8443对访问的网址进行转换。如果服务器对http和https协议监听的端口进行了修改，则需要修改配置文件让系统了解http和https的端口信息。

我们可以使用port-mappings自定义端口映射。

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

```
<intercept-url pattern="/admin.jsp" access="ROLE_ADMIN" requires-channel="https"/>
<intercept-url pattern="/*" access="ROLE_USER" />
<port-mappings>
  <port-mapping http="9000" https="9443"/>
</port-mappings>
</http>
```

上述配置文件中，我们定义了9000与9443的映射，现在系统会在强制使用http协议的网址时使用9000作为端口号，在强制使用https协议的网址时使用9443作为端口号，这些端口号会反映在重定向后生成网址中。

实例在ch111。

[上一页](#)[第 16 章 切换用户](#)[上一级](#)[起始页](#)[下一页](#)[第 18 章 digest认证](#)

第 18 章 digest 认证

digest 认证比 form-login 和 http-basic 更安全的一种认证方式，尤其适用于不能使用 https 协议的场景。它与 http-basic 一样，都是不基于 session 的无状态认证方式。

18.1. 配置 digest 验证

因为 digest 不包含在命名空间中，所以我们需要配置额外的过滤器和验证入口。

```
<beans:bean id="digestProcessingFilter" class="org.springframework.security.ui.digestauth.DigestProcessingFilter">
  <custom-filter position="BASIC_PROCESSING_FILTER" />
  <beans:property name="authenticationEntryPoint" ref="digestProcessingFilterEntryPoint"/>
  <beans:property name="userService"
    ref="org.springframework.security.userdetails.memory.InMemoryDaoImpl"/>
</beans:bean>

<beans:bean id="digestProcessingFilterEntryPoint"
  class="org.springframework.security.ui.digestauth.DigestProcessingFilterEntryPoint">
  <beans:property name="realmName" value="springsecurity"/>
  <beans:property name="key" value="changeIt"/>
</beans:bean>
```

然后记得删除 `auto-config="true"`，去除默认的 form-login 和 http-basic 认证，并添加对验证入口的引用。

```
<http auto-config="true" entry-point-ref="digestProcessingFilterEntryPoint">
  <intercept-url pattern="/admin.jsp" access="ROLE_ADMIN" />
  <intercept-url pattern="/" access="ROLE_USER" />
</http>
```

现在我们访问系统时，不会再进入之前的登录页面，而是会显示浏览器原生的登录对话框。



图 18.1. digest 登录

登录成功之后，我们可以在HTTP请求头部看到basic验证所需的属性Authorization。

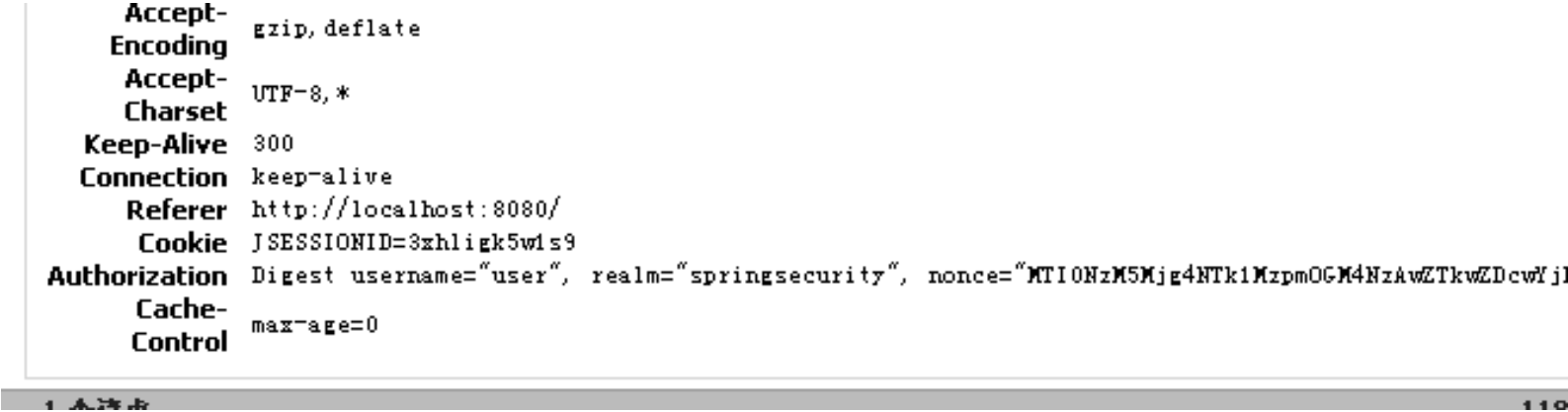


图 18.2. HTTP 请求头

最后需要注意的是，因为digest认证不使用session，所以无法与rememberMe功用。

18.2. 编程实现basic客户端

如果希望自己编写客户端进行digest认证，可以参考RFC 2617，它是对RFC 2069这个早期摘要式认证标准的更新。

在HTTP请求头中将包含这样一个Authorization，它包含了username, realm, nonce, uri, responseDigest, qop, nc和cnonce八个部分。其中nonce是digest认证的中心，它的组成结构如下所示：

```
base64(expirationTime❶ + ":" + md5Hex(expirationTime + ":" + key❷))
```

❶ 其中expirationTime是nonce的过期时间，单位是毫秒。

② key是放置nonce修改的私钥。

如果服务器生成的nonce已经过期（但是摘要还是有效），DigestProcessingFilterEntryPoint会发送一个"stale=true"头信息。这告诉用户代理，这里不再需要打扰用户（像是密码和用户其他都是正确的），只是简单尝试使用一个新nonce。

实例在ch112。

[上一页](#)[第 17 章 信道安全](#)[上一级](#)[起始页](#)[下一页](#)[部分 III. 保护method篇](#)

部分 III. 保护method篇

Spring Security使用AOP对方法调用进行权限控制，这部分内容基本都是来自于Spring提供的AOP功能，Spring Security进行了自己的封装，我们可以使用声明和编程两种方式进行权限管理。

第 19 章 保护方法调用

这里有三种方式可以选择：

19.1. 控制全局范围的方法权限

使用global-method-security和protect-point标签来管理全局范围的方法权限。

为了在spring中使用AOP，我们要为项目添加几个依赖库。

```
<dependency>
  <groupId>cglib</groupId>
  <artifactId>cglib-nodep</artifactId>
  <version>2.1_3</version>
</dependency>
<dependency>
  <groupId>org.aspectj</groupId>
  <artifactId>aspectjrt</artifactId>
  <version>1.6.4</version>
</dependency>
<dependency>
  <groupId>org.aspectj</groupId>
  <artifactId>aspectjweaver</artifactId>
  <version>1.6.4</version>
</dependency>
```

首先来看看我们将要保护的java类。

```
package com.family168.springsecuritybook.ch12;

public class MessageServiceImpl implements MessageService {
    public String adminMessage() {
        return "admin message";
    }

    public String adminDate() {
        return "admin " + System.currentTimeMillis();
    }

    public String userMessage() {
        return "user message";
    }

    public String userDate() {
        return "user " + System.currentTimeMillis();
    }
}
```

这里使用的是spring-2.0中的aop语法，对MessageService中所有以admin开头的方法进行权限控制，限制这些方法只能由ROLE_ADMIN调用。

```
<global-method-security>
  <protect-pointcut
    expression="execution(* com.family168.springsecuritybook.ch12.MesageServiceImpl.admin*(..))"
    access="ROLE_ADMIN"/>
</global-method-security>
```

现在只有拥有ROLE_ADMIN权限的用户才能调用MessageService中以admin开头的方法了，当我们以user/user登陆系统时，尝试调用MessageService类的adminMessage()会跑出一个“访问被拒绝”的异常。

19.2. 控制某个bean内的方法权限

在bean中嵌入intercept-methods和protect标签。

这需要改造配置文件。

```
<beans:bean id="messageService" class="com.family168.springsecuritybook.ch12.MessageServiceImpl">
  <intercept-methods>
    <protect access="ROLE_ADMIN" method="userMessage"/>
  </intercept-methods>
</beans:bean>
```

现在messageService中的userMessage()方法只允许拥有ROLE_ADMIN权限的用户才能调用了。



使用intercept-methods面临着几个问题

首先，intercept-methods只能使用jdk14的方式拦截实现了接口的类，而不能用cglib直接拦截无接口的类。

其次，intercept-methods和global-method-security一起使用，同时使用时，global-method-security一切正常，intercept-methods则会完全不起作用。

19.3. 使用annotation控制方法权限

借助jdk5以后支持的annotation，我们直接在代码中设置某一方法的调用权限。

现在有两种选择，使用Spring Security提供的Secured注解，或者使用jsr250规范中定义的注解。

19.3.1. 使用Secured

首先修改global-method-security中的配置，添加支持annotation的参数。

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

然后添加依赖包。

```
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-core-tiger</artifactId>
```

```
<version>2.0.4</version>
</dependency>
```

现在我们随便在java代码中添加注解了。

```
package com.family168.springsecuritybook.ch12;

import org.springframework.security.annotation.Secured;

public class MessageServiceImpl implements MessageService {
    @Secured({"ROLE_ADMIN", "ROLE_USER"})
    public String userMessage() {
        return "user message";
    }
}
```

在Secured中设置了ROLE_ADMIN和ROLE_USER两个权限，只要当前用户拥有其中任意一个权限都可以调用这个方法。

19.3.2. 使用jsr250

首先还是要修改配置文件。

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

然后添加依赖包。

```
<dependency>
    <groupId>javax.annotation</groupId>
    <artifactId>jsr250-api</artifactId>
    <version>1.0</version>
</dependency>
```

现在可以在代码中使用jsr250中的注解了。


```
package com.family168.springsecuritybook.ch12;

import javax.annotation.security.DenyAll;
import javax.annotation.security.PermitAll;
import javax.annotation.security.RolesAllowed;

public class MessageServiceImpl implements MessageService {
    @RolesAllowed({"ROLE_ADMIN", "ROLE_USER"})❶
    public String userMessage() {
        return "user message";
    }

    @DenyAll❷
    public String userMessage2() {
        return "user message";
    }

    @PermitAll❸
    public String userMessage2() {
        return "user message";
    }
}
```

- ❶ RolesAllowed与前面的Secured功能相同，用户只要满足其中定义的权限之一就可以调用方法。
- ❷ DenyAll拒绝所有的用户调用方法。
- ❸ PermitAll允许所有的用户调用方法。

从实际使用上来讲，jsr250里多出来的DenyAll和PermitAll纯属浪费，谁会定义谁也不能调用的方法呢？实际上，要是annotation支持布尔操作就好了，比如逻辑并，逻辑或，逻辑否之类的。

还有jsr250中未被支持的RunAs注解，如果能利用起来估计更有趣。

实例在ch201。

[上一页](#)[部分 III. 保护method篇](#)[上一级](#)[起始页](#)[下一页](#)[第 20 章 权限管理的基本概念](#)

第 20 章 权限管理的基本概念

20.1. 认证与验证

Spring Security作为权限管理框架，其内部机制可分为两大部分，其一是认证授权authorization，其二是权限校验authentication。

认证授权authorization是指，根据用户提供的身份凭证，生成权限实体，并为之授予相应的权限。

权限校验authentication是指，用户请求访问被保护资源时，将被保护资源所需的权限和用户权限实体所拥护的权限二者进行比对，如果校验通过则用户可以访问被保护资源，否则拒绝访问。

我们之前讲过的form-login，http-basic, digest都属于认证授权authorization部分的概念，用户可以通过这些机制登录到系统中，系统会为用户生成权限主体，并授予相应的权限。

与之相对的，FilterSecurityInterceptor，Method保护，taglib，@Secured都属于权限校验authentication，无论是对URL的请求，对方法的调用，页面信息的显示，都要求用户拥有相应的权限才能访问，否则请求即被拒绝。

20.2. SecurityContext安全上下文

为使所有的组件都可以通过同一方式访问当前的权限实体，Spring Security特别提供了SecurityContext作为安全上下文，可以直接通过SecurityContextHolder获得当前线程中的SecurityContext。

```
SecurityContext securityContext = SecurityContextHolder.getContext();
```

默认情况下，SecurityContext的实现基于ThreadLocal，系统会在每次用户请求时将SecurityContext与当前Thread进行绑定，这在web系统中是很常用的使用方式，服务器维护的线程池允许多个用户同时并发访问系统，而ThreadLocal可以保证隔离不同Thread之间的信息。

当时对于单机应用来说，因为只有一个人使用，并不存在并发的情况，所以完全可以让所有Thread都共享同一个SecurityContext，因此Spring Security为我们提供了不同的策略模式，我们可以通过设置系统变量的方式选择希望使用的策略类。

```
java -Dspring.security.strategy=MODE_GLOBAL com.family168.springsecuritybook.Main
```

也可以调用SecurityContextHolder的setStrategyName()方法来修改系统使用的策略。

```
SecurityContextHolder.setStrategyName("MODE_GLOBAL");
```

20.3. Authentication验证对象

SecurityContext中保存着实现了Authentication接口的对象，如果用户尚未通过认证，那么SecurityContext.getAuthenticaiton()方法就会返回null。

可以使用Authentication接口中定义的几个方法，获得当前权限实体的信息。

```
public interface Authentication extends Principal, Serializable {

    GrantedAuthority[] getAuthorities();❶

    Object getCredentials();❷

    Object getDetails();❸

    Object getPrincipal();❹

    boolean isAuthenticated();

    void setAuthenticated(boolean isAuthenticated)
        throws IllegalArgumentException;

}
```

默认情况下，会在某一个进行认证的过滤器中生成一个UsernamePasswordAuthenticationToken实例，并将此实例放到SecurityContext中。

❶ 获得权限主体拥有的权限。

权限实体拥有的权限，GrantedAuthority接口内只有一个方法getAuthority()，它的返回值是一个字符串，这个字符串用来标识系统中的某一权限。用户认证后权限实体将拥有一个保存了一系列GrantedAuthority对象的数组，之后可以用于进行验证用户是否可以访问系统中被保护资源。

❷ 获得权限主体的凭证，此凭证应该可以唯一标示权限主体。

默认情况下，凭证是用户的登录密码。

❸ 获得验证请求有关的附加信息。

默认情况下，附加信息是WebAuthenticationDetails的一个实例，其中保存了客户端ip和sessionid。

❹ 获得权限主体。

默认情况下，权限主体是一个实现了UserDetails接口的对象。

[上一页](#)[第 19 章 保护方法调用](#)[上一级](#)[起始页](#)[下一页](#)[第 21 章 Voter表决者](#)

第 21 章 Voter 表决者

21.1. Voter 表决者

实际上并没有翻译的字面含义那么有血有肉，实际上就是一些条件，判断权限的时候，这些条件有三个状态。弃权，通过，禁止。最后通过你在xml里配置的策略来决定到底是不是让你访问这个需要验证的对象。

Spring Security提供的策略有三个

- UnanimousBased.java 只要有一个Voter不能完全通过权限要求，就禁止访问。这个太可怕了，我今天晚上就栽在它上面了。就因为我给所有的资源设置了两个角色，但当前的用户只拥有其中一个角色，就导致这个用户因为权限不够，所以无法继续访问资源了。简直无法理喻啊。
- AffirmativeBased.java只要有一个Voter不能通过权限要求，就禁止访问。这里应该是一个最小通过，就是说至少满足里其中一个条件就可以通过了。
- ConsensusBased.java只要通过的Voter比禁止的Voter数目多就可以访问了。嘿嘿。

最后我当然选择AffirmativeBased.java，这样，我给一个资源配置几个角色，用户只要满足其中一个角色就可以访问啦。这样更正常一些啊。

21.2. 默认角色名称都是以ROLE_开头

稍微注意一下，默认角色名称都要以ROLE_开头，否则不会被计入权限控制，如果需要修改，就在xml里配个什么前缀的。可以用过配置roleVoter的rolePrefix来改变这个前缀。

```
<bean id="roleVoter" class="org.springframework.security.vote.RoleVoter">
  <property name="rolePrefix" value="AUTH_" />
</bean>
```

[上一页](#)

第 20 章 权限管理的基本概念

[上一级](#)

[起始页](#)

[下一页](#)

第 22 章 拦截器

第 22 章 拦截器

无论是Filter，MethodInterceptor，ACL都要用到AOP，实际上都是拦截器的概念，其中要用到AbstractSecurityInterceptor总拦截器，AfterInvocationManager后置拦截，authenticationManager验证管理器，可能还要用上RunAsManager。

关于RunAsManager，官方文档给出的解释是，在HttpInvoker或者Web Service的情况下，当前用户的一些身份要转换成其他身份，这时就是用RunAsManager，默认将以RUN_AS开头的权限名，改变成ROLE_RUN_AS开头的权限名，然后重新赋予当前认证主体是用。现在的问题是不清楚具体使用在什么场景。

部分 IV. ACL篇

Access Control List是一个很容易被人们提起的功能，比如业务员甲只能查看自己签的合同信息，不能看到业务员乙签的合同信息。这个功能在Spring Security中也有支持，但是配置异常困难，也没有namespace方式的支持。甚至我们不知道能否将这套ACL完全放在数据库中。

先设置jdbc中的表结构。

然后设置acl的xml配置。

检测完method的调用后，可以考虑用aop控制after invocation来控制返回的数据形式，这里就会出现虎牙子。

第 23 章 Spring Security 中的 ACL

ACL 即访问控制列表 (Access Controller List)，它是用来做细粒度权限控制所用的一种权限模型。对 ACL 最简单的描述就是两个业务员，每个人只能查看操作自己签的合同，而不能看到对方的合同信息。

下面我们会介绍 Spring Security 中是如何实现 ACL 的。

23.1. 准备数据库和 aclService

ACL 所需的四张表，表结构见附录：[附录 E, 数据库表结构](#)。

然后我们需要配置 aclService，它负责与数据库进行交互。

23.1.1. 为 acl 配置 cache

默认使用 ehcache，spring security 提供了一些默认的实现类。

```
<bean id="aclCache" class="org.springframework.security.acls.jdbc.EhCacheBasedAclCache">
  <constructor-arg ref="aclEhCache"/>
</bean>

<bean id="aclEhCache" class="org.springframework.cache.ehcache.EhCacheFactoryBean">
  <property name="cacheManager" ref="cacheManager"/>
  <property name="cacheName" value="aclCache"/>
</bean>
```

在 ehcache.xml 中配置对应的 aclCache 缓存策略。

```
<cache
  name="aclCache"
  maxElementsInMemory="1000"
```

```

    eternal="false"
    timeToIdleSeconds="600"
    timeToLiveSeconds="3600"
    overflowToDisk="true"
  />

```

23.1.2. 配置lookupStrategy

简单来说，lookupStrategy的作用就是从数据库中读取信息，把这些信息提供给aclService使用，所以我们要为它配置一个dataSource，配置中还可以看到一个aclCache，这就是上面我们配置的缓存，它会把资源最大限度的利用起来。

```

<bean id="lookupStrategy" class="org.springframework.security.acls.jdbc.BasicLookupStrategy">
  <constructor-arg ref="dataSource"/>
  <constructor-arg ref="aclCache"/>
  <constructor-arg>
    <bean class="org.springframework.security.acls.domain.AclAuthorizationStrategyImpl">
      <constructor-arg>
        <list>
          <ref local="adminRole"/>
          <ref local="adminRole"/>
          <ref local="adminRole"/>
        </list>
      </constructor-arg>
    </bean>
  </constructor-arg>
  <constructor-arg>
    <bean class="org.springframework.security.acls.domain.ConsoleAuditLogger"/>
  </constructor-arg>
</bean>

<bean id="adminRole" class="org.springframework.security.GrantedAuthorityImpl">
  <constructor-arg value="ROLE_ADMIN"/>
</bean>

```

中间一部分可能会让人感到困惑，为何一次定义了三个adminRole呢？这是因为一旦acl信息被保存到数据库中，无论是修改它的从属者，还是变更授权，抑或是修改其他的ace信息，都需要控制操作者的权限，这里配置的三个权限将对应于上述的三种修改操作，我们把它配置成，只有ROLE_ADMIN才能执行这三种修改操作。

23.1.3. 配置aclService

当我们已经拥有了dataSource, lookupStrategy和aclCache的时候，就可以用它们来组装aclService了，之后所有的acl操作都是基于aclService展开的。

```
<bean id="aclService" class="org.springframework.security.acls.jdbc.JdbcMutableAclService">
  <constructor-arg ref="dataSource"/>
  <constructor-arg ref="lookupStrategy"/>
  <constructor-arg ref="aclCache"/>
</bean>
```

23.2. 使用aclService管理acl信息

当我们添加了一条信息，要在acl中记录这条信息的ID，所有者，以及对应的授权信息。下列代码在添加信息后执行，用于添加对应的acl信息。

```
ObjectIdentity oid = new ObjectIdentityImpl(Message.class, message.getId());
MutableAcl acl = mutableAclService.createAcl(oid);
acl.insertAce(0, BasePermission.ADMINISTRATION,
    new PrincipalSid(owner), true);
acl.insertAce(1, BasePermission.DELETE,
    new GrantedAuthoritySid("ROLE_ADMIN"), true);
acl.insertAce(2, BasePermission.READ,
    new GrantedAuthoritySid("ROLE_USER"), true);
mutableAclService.updateAcl(acl);
```

第一步，根据class和id生成object的唯一标示。

第二步，根据object的唯一标示，创建一个acl。

第三步，为acl增加ace，这里我们让对象的所有者拥有对这个对象的“管理”权限，让“ROLE_ADMIN”拥有对这个对象的“删除”权限，让“ROLE_USER”拥有对这个对象的“读取”权限。

最后，更新acl信息。

当我们删除对象时，也要删除对应的acl信息。下列代码在删除信息后执行，用于删除对应的acl信息。

```
ObjectIdentity oid = new ObjectIdentityImpl(Message.class, id);
mutableAclService.deleteAcl(oid, false);
```

使用class和id可以唯一标示一个对象，然后使用deleteAcl()方法将对象对应的acl信息删除。

23.3. 使用acl控制delete操作

上述代码中，除了对象的拥有者之外，我们还允许“ROLE_ADMIN”也可以删除对象，但是我们会允许除此之外的其他用户拥有删除对象的权限，为了限制对象的删除操作，我们需要修改Spring Security的默认配置。

首先要增加一个对delete操作起作用的表决器。

```
<bean id="aclMessageDeleteVoter" class="org.springframework.security.vote.AclEntryVoter">
  <constructor-arg ref="aclService"/>
  <constructor-arg value="ACL_MESSAGE_DELETE"/>
  <constructor-arg>
    <list>
      <util:constant static-field="org.springframework.security.acls.domain.BasePermission.ADMINISTRATION"/>
      <util:constant static-field="org.springframework.security.acls.domain.BasePermission.DELETE"/>
    </list>
  </constructor-arg>
  <property name="processDomainObjectClass" value="com.family168.springsecuritybook.ch12.Message"/>
</bean>
```

它只对Message这个类起作用，而且可以限制只有管理和删除权限的用户可以执行删除操作。

然后将这个表决器添加到AccessDecisionManager中。

```
<bean id="aclAccessDecisionManager" class="org.springframework.security.vote.AffirmativeBased">
  <property name="decisionVoters">
    <list>
      <bean class="org.springframework.security.vote.RoleVoter"/>
      <ref local="aclMessageDeleteVoter"/>
    </list>
  </property>
</bean>
```

现在AccessDecisionManager中有两个表决器了，除了默认的RoleVoter之外，又多了一个我们刚刚添加的aclMessageDeleteVoter。

现在可以把新的AccessDecisionManager赋予全局方法权限管理器了。

```
<global-method-security secured-annotations="enabled"
  access-decision-manager-ref="aclAccessDecisionManager"/>
```

然后我们就可以在MessageService.java中使用Secured注解，控制删除操作了。

```
@Transactional
@Secured("ACL_MESSAGE_DELETE")
public void remove(Long id) {
    Message message = this.get(id);
    list.remove(message);

    ObjectIdentity oid = new ObjectIdentityImpl(Message.class, id);
    mutableAclService.deleteAcl(oid, false);
}
```

实际上，我们最好不要让没有权限的操作者看到remove这个链接，可以使用taglib隐藏当前用户无权看到的信息。

```
<sec:accesscontrollist domainObject="${item}" hasPermission="8,16">
    |
    <a href="message.do?action=remove&id=${item.id}">Remove</a>
</sec:accesscontrollist>
```

8, 16是acl默认使用的掩码，8表示DELETE，16表示ADMINISTRATOR，当用户不具有这些权限的时候，他在页面上就看不到remove链接，也就无法执行操作了。

这比让用户可以执行remove操作，然后跑出异常，警告访问被拒绝要友好得多。

23.4. 控制用户可以看到哪些信息

当用户无权查看一些信息时，我们需要配置afterInvocation，使用后置判断的方式，将用户无权查看的信息，从MessageService返回的结果集中过滤掉。

后置判断有两种形式，一种用来控制单个对象，另一种可以过滤集合。

```
<bean id="afterAclRead" class="org.springframework.security.afterinvocation.AclEntryAfterInvocationProvider">
    <sec:custom-after-invocation-provider/>
    <constructor-arg ref="aclService"/>
    <constructor-arg>
        <list>
            <util:constant static-field="org.springframework.security.acls.domain.BasePermission.ADMINISTRATION"/>
            <util:constant static-field="org.springframework.security.acls.domain.BasePermission.READ"/>
        </list>
    </constructor-arg>
</bean>
```

```

    </list>
  </constructor-arg>
</bean>

<bean id="afterAclCollectionRead" class="org.springframework.security.afterinvocation.
AclEntryAfterInvocationCollectionFilteringProvider">
  <sec:custom-after-invocation-provider/>
  <constructor-arg ref="aclService"/>
  <constructor-arg>
    <list>
      <util:constant static-field="org.springframework.security.acls.domain.BasePermission.ADMINISTRATION"/>
      <util:constant static-field="org.springframework.security.acls.domain.BasePermission.READ"/>
    </list>
  </constructor-arg>
</bean>

```

afterAclRead可以控制单个对象是否可以显示，afterAclCollectionRead则用来过滤集合中哪些对象可以显示。[\[6\]](#)

对这两个bean都是用了custom-after-invocation-provider标签，将它们加入的后置判断的行列，下面我们为MessageService.java中的对应方法添加Secured注解，之后它们就可以发挥效果了。

```

@Secured({"ROLE_USER", "AFTER_ACL_READ"})
public Message get(Long id) {
    for (Message message : list) {
        if (message.getId().equals(id)) {
            return message;
        }
    }
    return null;
}

@Secured({"ROLE_USER", "AFTER_ACL_COLLECTION_READ"})
public List getAll() {
    return list;
}

```

以上就是Spring Security支持的ACL，我们只演示了DELETE一种情况，就已经编写了如此之多的xml配置文件，很难想象随着对象的增多，这个配置工作要扩展到什么程度，如何才能像之前配置user和resource时，将这些acl的信息都配置到database中呢？目前还是一个我们正在考虑的问题。

^[6] 这个地方会引发一个经典的问题，虎牙子，一般的思路是使用动态SQL的方式，在查询的时候就过滤掉无权显示的信息，但随着查询条件的复杂化，当出现SQL语句长度超过DBMS最大限制时，咱们就可以去撞墙了。

[上一页](#)

部分 IV. ACL篇

[上一级](#)

[起始页](#)

[下一页](#)

部分 V. 最佳实践篇

部分 V. 最佳实践篇

根据不同的权限模型，实现多种完整的权限管理系统。

- default: 使用Spring Security默认提供的数据库结构，实现用户信息和权限管理后台。
- rbac1: RBAC1模型。
- rbac2: RBAC2模型。
- rbac3: RBAC3模型。
- acl: 行级细粒度权限控制。
- ajax: 全站式ajax的web 2.0系统中应用权限管理。
- web: 在默认数据库结构的基础上，实现web所需的“用户注册”，“用户激活”，“安全提问”，“找回密码”功能。
- group: 分级组织管理模型。
- adapter: 权限适配器。

第 24 章 最简控制台

所谓的最简，实际上就是尽量利用现有资源，实现一个可管理的权限后台。

我们将使用Spring Security提供的filter实现URL级的权限控制，使用Spring Security提供的UserDetailsManager实现用户管理，其中会包含用户密码加密和用户信息缓存。麻雀虽小，五脏俱全，如果想为自己的系统添加最简的权限后台，这一章将是不二之选。

我们将在这个简易的控制台中实现如下功能：浏览用户，新增用户，修改用户，删除用户，修改密码，用户授权。

24.1. 平台搭建

选择maven2作为主要的构建工具，以便更加方便的管理第三方依赖，这章使用的依赖如下所示：

```
com.family168.springsecuritybook:ch101:war:0.1
+- org.springframework.security:spring-security-taglibs:jar:2.0.4:compile
| +- org.springframework.security:spring-security-core:jar:2.0.4:compile
| | +- org.springframework:spring-core:jar:2.0.8:compile
| | +- org.springframework:spring-context:jar:2.0.8:compile
| | | \- aopalliance:aopalliance:jar:1.0:compile
| | +- org.springframework:spring-aop:jar:2.0.8:compile
| | +- org.springframework:spring-support:jar:2.0.8:runtime
| | +- commons-logging:commons-logging:jar:1.1.1:compile
| | +- commons-codec:commons-codec:jar:1.3:compile
| | \- commons-collections:commons-collections:jar:3.2:compile
| +- org.springframework.security:spring-security-acl:jar:2.0.4:compile
| | \- org.springframework:spring-jdbc:jar:2.0.8:compile
| |   \- org.springframework:spring-dao:jar:2.0.8:compile
| \- org.springframework:spring-web:jar:2.0.8:compile
|   \- org.springframework:spring-beans:jar:2.0.8:compile
+- hsqldb:hsqldb:jar:1.8.0.7:compile
+- javax.servlet:servlet-api:jar:2.4:provided
+- taglibs:standard:jar:1.1.2:compile
+- javax.servlet:jstl:jar:1.1.2:compile
\- net.sf.ehcache:ehcache:jar:1.6.0:compile
```

项目的目录结构如下：

```
+ ch101/
+ src/
+ main/
```

```

+ java/❶
+ com/
+ family168/
+ springsecuritybook/
+ ch101
+ * UserBean.java
+ * UserManager.java
+ * UserServicelet.java
+ resources/
+ hsqldb/❷
+ * test.properties
+ * test.scripts
+ * applicationContext-security.xml❸
+ * applicatoinContext-service.xml❹
+ webapp/❺
+ includes/
+ * error.jsp
+ * header.jsp
+ * message.jsp
+ * meta.jsp
+ * taglibs.jsp
+ scripts/
+ * jquery.min.js
+ * jquery.validate.pack.js
+ * messages_cn.js
+ WEB-INF/
+ * web.xml
+ * index.jsp
+ * login.jsp
+ * user-changePassword.jsp
+ * user-create.jsp
+ * user-edit.jsp
+ * user-list.jsp
+ * user-view.jsp
+ test/
+ resources/
+ * pom.xml

```

- ❶ src/main/java/目录下放着所有的java源代码。
- ❷ src/main/resources/hsqldb/目录下放置着hsqldb数据库表结构和演示数据。
- ❸ 权限控制相关的配置文件。
- ❹ 进行用户管理和权限管理所需的配置文件。
- ❺ src/main/webapp/目录下放着web应用所需的JavaScript脚本与jsp文件。

24.2. 用户登录

用户需要登录系统才能进入系统进行操作。有关自定义登录页面的介绍，请参考之前的章节。[???](#)

登陆

用户:

密码:

☐ 两周之内不必登陆

登陆 重置

图 24.1. 用户登录

我们为了演示的需要预设了两个用户admin/admin和user/user，打开演示用的数据库文件test.scripts可以看到用户信息以及加密过的用户密码。

```
INSERT INTO USERS VALUES('admin','ceb4f32325eda6142bd65215f4c0f371',TRUE)
INSERT INTO USERS VALUES('user','47a733d60998c719cf3526ae7d106d13',TRUE)
INSERT INTO AUTHORITIES VALUES('admin','ROLE_ADMIN')
INSERT INTO AUTHORITIES VALUES('admin','ROLE_USER')
INSERT INTO AUTHORITIES VALUES('user','ROLE_USER')
```

为了提升安全等级，我们对密码使用了md5和saltValue进行加密，对应的配置文件在applicationContext-securit.xml中。

```
<authentication-provider user-service-ref="userDetailsManager">
  <password-encoder ref="passwordEncoder">
    <salt-source user-property="username"/>
  </password-encoder>
</authentication-provider>
```

24.3. 用户信息列表

用户登录成功之后即进入用户信息列表。

Create User username: user | [Change Password](#) | [Logout](#)

Username	Password	Enabled	Authorities	Operation
admin	[PROTECTED]	true	ROLE_ADMIN, ROLE_USER	View Update Remove
user	[PROTECTED]	true	ROLE_USER	View Update Remove

图 24.2. 用户信息列表

显示所有用户信息的请求地址为/user.do?action=list，这个请求将交由UserServlet.java处理，在list()方法中调用UserManager.java的getAll()方法获得数据库中所有的用户信息。

```

/**
 * get all of user.
 */
public List<UserBean> getAll() {
    String sql = "select username,password,enabled,authority"
        + " from users u inner join authorities a on u.username=a.username";
    List<Map> list = jdbcTemplate.queryForList(sql);

    List<UserBean> userList = new ArrayList<UserBean>();
    UserBean ub = null;
    for (Map map : list) {
        if (ub == null) {
            ub = new UserBean((String) map.get("username"),
                (String) map.get("password"),
                (Boolean) map.get("enabled"));
            ub.addAuthority((String) map.get("authority"));
        } else if (ub.getUsername().equals(map.get("username"))) {
            ub.addAuthority((String) map.get("authority"));
        } else {
            userList.add(ub);
            //
            ub = new UserBean((String) map.get("username"),
                (String) map.get("password"),
                (Boolean) map.get("enabled"));
            ub.addAuthority((String) map.get("authority"));
        }
    }
    if (!list.isEmpty()) {
        userList.add(ub);
    }

    return userList;
}

```

getAll()方法中将数据库中所有的用户信息取出来，并将用户信息和对应的权限组装在一起，并将获得的数据提交给user-list.jsp进行显示。出于安全性的考虑，即使密码已经经过了加密，我们还是选择在页面上不显示用户的密码。

24.4. 添加用户

点击页面左上角的Create User可以进入添加用户的界面。

[Back](#)

username: user | [Change Password](#) | [Logout](#)

Create User

Username:

x

Password:

●

Confirm Password:

●

Enabled:

☒

Authorities:

x

提交查询

重置

图 24.3. 添加用户

如果操作成功，会向数据库中添加一条用户记录，以及对应的权限信息，并在用户列表页面中显示成功提示。

[Create User](#)

username: user | [Change Password](#) | [Logout](#)

success				
Username	Password	Enabled	Authorities	Operation
admin	[PROTECTED]	true	ROLE_ADMIN, ROLE_USER	View Update Remove
user	[PROTECTED]	true	ROLE_USER	View Update Remove
x	[PROTECTED]	true	x	View Update Remove

图 24.4. 操作成功

如果操作失败，会跳转到添加页面，并显示错误信息。

[Back](#)username: user | [Change Password](#) | [Logout](#)

user already exists.

Create User

Username:

Password:

Confirm Password:

Enabled:

☒

Authorities:

提交查询

重置

图 24.5. 操作错误

添加用户操作过程中，UserServlet.java中的save()方法负责请求跳转与数据校验，UserManager.java中的save()方法负责将提交的保存入数据库。

```
/**
 * create a new user and insert he to database.
 */
public void save(String username, String password, boolean enabled, String[] authorities) {
    GrantedAuthority[] gas = new GrantedAuthority[authorities.length];
    for (int i = 0; i < authorities.length; i++) {
        gas[i] = new GrantedAuthorityImpl(authorities[i].trim());
    }
    String encodedPassword = passwordEncoder.encodePassword(password, username);
    UserDetails ud = new User(username, encodedPassword, enabled, true, true, true, gas);
    userDetailsManager.createUser(ud);
}
```

这里我们需要注意两个部分。

第一部分，需要将用户拥有的权限转换为GrantedAuthority数组，并赋予添加的用户。

第二部分，我们在保存用户密码之前，要将提交的明文密码使用passwordEncoder进行加密，encodePassword()方法会使用我们设置的加密算法，并使用username作为saltValue对密码进行加密。

最终，我们将转化后的数据组装为一个UserDetails对象，并保存到数据库中，以此完成整个添加用户的操作。

24.5. 修改用户信息

在用户列表页面选择一条用户信息进行修改。

[Back](#)username: user | [Change Password](#) | [Logout](#)

Edit User

Username: x

Enabled: ☒

Authorities:

提交查询

重置

图 24.6. 修改用户

修改操作与UserManager.java中的update()方法对应。

```
/**
 * update a user information, includes username, enabled or authorities.
 */
public void update(String username, boolean enabled, String[] authorities) {
    GrantedAuthority[] gas = new GrantedAuthority[authorities.length];
    for (int i = 0; i < authorities.length; i++) {
        gas[i] = new GrantedAuthorityImpl(authorities[i].trim());
    }
    UserDetails oldUserDetails = userDetailsManager.loadUserByUsername(username);
    UserDetails ud = new User(username,
        oldUserDetails.getPassword(),
        enabled,
        oldUserDetails.isAccountNonExpired(),
        oldUserDetails.isAccountNonLocked(),
        oldUserDetails.isCredentialsNonExpired(),
        gas);
    userDetailsManager.updateUser(ud);
}
```

因为修改用户信息不包含修改用户密码，所以此处不需要进行密码加密，这里可以放心调用UserDetailsManager中提供的updateUser()方法，它会帮我们维护用户缓存。

可以查看指定用户的详细信息。

[Back](#)username: user | [Change Password](#) | [Logout](#)

User Info

Username: x

Password: [PROTECTED]

Enabled: true

Authorities: x

图 24.7. 浏览用户信息

删除用户操作与上述操作基本类似，UserDetailsManager会帮我们处理用户缓存的问题，不用担心出现脏数据。

24.6. 修改自己的密码

用户列表右上角有一个Change Password的链接，它用来修改当前登录系统用户的密码。

[Back](#)username: user | [Change Password](#) | [Logout](#)

Change User's Password

Old Password:

New Password:

Confirm Password:

图 24.8. 修改密码

在UserManager.java内部，我们会调用一个名为changePassword()的方法，这个方法需要两个参数oldPassword和newPassword，这时因为修改密码之前需要先对当前用户进行认证，所以可以在代码中看到，我们为oldPassword和newPassword都进行了加密操作。

```
/**
 * let current user change password.
 */
public void changePassword(String oldPassword, String newPassword) {
    UserDetails userDetails = (UserDetails) SecurityContextHolder.getContext()
        .getAuthentication()
```



```
    .getPrincipal();  
    String username = userDetails.getUsername();  
  
    String encodedOldPassword = passwordEncoder.encodePassword(oldPassword, username);  
    String encodedNewPassword = passwordEncoder.encodePassword(newPassword, username);  
    userDetailsManager.changePassword(encodedOldPassword, encodedNewPassword);  
}
```

这个方法可以看做是使用编程方式获得当前登录用户信息的一个典型范例，在系统的任何部分，我们都可以使用这样的方式直接获得SecurityContext中保存的登录用户信息。

至此，我们基于Spring Security完成了一个完整的权限管理系统后台，实例代码在ch401。

[上一页](#)[部分 V. 最佳实践篇](#)[上一级](#)[起始页](#)[下一页](#)[附录 A. 修改日志](#)

附录 A. 修改日志

杂谈：以下的一些问题，还没有搞得十分清晰，先记录下来以备之后查阅。

下面是一些复杂的

- 关于LDAP的部分，因为之前没使用过，所以很难下手。
- ACL是一个需要细化的部分。我们需要更简单易用的解决方案，现在的配置方式太容易把人逼疯了。
- OpenID
- X.509
- portlet
- ntlm
- jaas
- rcp, remoting, HttpInvoker, RMI
- 自定义过滤器

超出Spring Security 2.0范围的功能

- Spring Security 3.0中支持的expression。
- Acegi中支持的jcaptcha。
- 自定义扩展namespace
- 最终的目标要实现RBAC的几个标准模型。
- 最好学习metadata里一样，可以实现一个自定义查询权限的前台。

修订历史

修订 0.0.7

2009-07-10

1. 整理章节的名称结构。
2. 添加: [第 12 章 自动登录](#)
3. 添加: [第 13 章 匿名登录](#)
4. 添加: [第 14 章 防御会话伪造](#)。
5. 添加: [第 15 章 预先认证](#)。
6. 添加: [第 16 章 切换用户](#)。
7. 添加: [第 17 章 信道安全](#)。
8. 添加: [第 18 章 digest 认证](#)。
9. 添加: [第 20 章 权限管理的基本概念](#)。
10. 添加: [第 21 章 Voter 表决者](#)。
11. 添加: [第 22 章 拦截器](#)。
12. 添加: [附录 F, 异常](#)。
13. 添加: [附录 G, 事件](#)。
14. 修改: [第 10 章 basic 认证](#), 添加单独使用浏览器登录的示例。

修订 0.0.6

2009-07-02

1. 添加: [第 11 章 标签库](#)
2. 添加: [第 19 章 保护方法调用](#)
3. 添加: [第 24 章 最简控制台](#)
4. 修改: [第 8 章 管理会话](#), 监听器的类名写错了。感谢帅哥duansiyang。

修订 0.0.5

2009-06-25

1. 添加: [第 9 章 单点登录](#)
2. 添加: [第 10 章 basic 认证](#)
3. 添加: [第 23 章 Spring Security 中的 ACL](#)

修订 0.0.4

2009-06-22

1. 添加: [第 8 章 管理会话](#)

修订 0.0.3

2009-06-21

1. 添加: [第 7 章 图解过滤器](#)
2. 添加: [附录 D, 命名空间](#)
3. 添加: [附录 E, 数据库表结构](#)

修订 0.0.2

2009-06-09

1. 整理之前的章节，修改一些疏漏。
2. 添加：[第 5 章 使用数据库管理资源](#)
3. 添加：[第 6 章 控制用户信息](#)
4. 添加：[附录 C, *Spring Security-3.0.0.M1*](#)

修订 0.0.1

2009-05-26

1. 初稿完成。[序言](#)

[上一页](#)

第 24 章 最简控制台

[起始页](#)

[下一页](#)

附录 B. 常见问题解答

附录 B. 常见问题解答

B.1. Q: 如何获得源代码

A: 在SpringSecurity的发布包中的dist目录下，包含很多“.jar”文件，名称中包含“sources”的文件中就是源文件了，比如：可以在spring-security-core-2.0.4-sources.jar中找到core模块的所有文件。

B.2. Q: 为何登录时出现There is no Action mapped for namespace / and action name j_spring_security_check.

A: 这是因为登陆所发送的请求先被struts2的过滤器拦截了，为了试登陆请求可以被Spring Security正常处理，需要在web.xml中将Spring Security的过滤器放在struts2之前。

B.3. Q: 用户登陆之后没有进入设置的default-target-url页面。

A: Spring Security登陆成功后的策略是，先判断用户登录前是否尝试访问过受保护的页面，如果有，则跳转到用户登录前访问的受保护页面，否则跳转到default-target-url。如果希望登陆后一直跳转到default-target-url，可以使用always-use-default-target="true"。

B.4. Q: 如何实现国际化。

A: 在xml中添加如下配置：

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

B.5. Q: 如何监听Spring Security的事件日志。

A: 在xml中添加如下配置：

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

<beans:bean class="org.springframework.security.event.authorization.LoggerListener"/>
```

B.6. Q: 如何启用group。

A: 设置enableGroups="true"才能在JdbcDaoImpl中启用group，默认是禁用的，而namespace中没有支持这个参数，所以想用group时，只好自己配置了。

附录 C. Spring Security-3.0.0.M1

Spring Security于2009-05-27发布。

Spring Security从2.0.x一跃而至3.0.0.M1，这第一个里程碑里主要有三点值得我们注意：

- 支持Spring-3.0.0.M3，现在只能在JDK-5.0以上环境使用，不再支持JDK-1.4以及之前的版本。
- 大量重构代码结构，将核心库core，命名空间config，web验证部分都严格的分成独立的模块，不再像以前一样把所有代码都混放放core中。
- 支持Expression，现在无论是命名空间，配置文件，taglib中都可以使用表达式来指定需要的配置了。

使用Spring Security-3.0.0.M1制作了一个Helloworld，不需要修改任何配置就可以正常运行，这点上很佩服Spring系列的兼容性。

实例在x01下。

附录 D. 命名空间

默认namespace: "http://www.springframework.org/schema/security"

根节点可能是http, authentication-provider, authentication-manager, user-service, jdbc-user-service, ldap-user-service, filter-invocation-definition-source, ldap-server或者global-method-security。

- [第 D.1 节 “http”](#)
- [第 D.2 节 “authentication-provider”](#)
- [第 D.3 节 “ldap-server”](#)
- [第 D.4 节 “global-method-security”](#)

还有几个元素可以嵌入到其他bean标签里, filter-chain-map, custom-filter, custom-authentication-provider, intercept-methods。

```
<b:bean id="springSecurityFilterChain"
  class="org.springframework.security.util.FilterChainProxy">
  <filter-chain-map path-type="ant">
    <filter-chain pattern="/**"
      path-type="ant|regex"
      filters="httpSessionContextIntegrationFilter,
      authenticationProcessingFilter,
      exceptionTranslationFilter,
      filterInvocationInterceptor" />
  </filter-chain-map>
</b:bean>

<filter-invocation-definition-source id="string" lowercase-comparisons="boolean" path-type="ant|regex">
  <intercept-url pattern="string"
    access="string"
    method="GET|DELETE|HEAD|OPTIONS|POST|PUT|TRACE"
    filters="none"
```

```

        requires-channel="http|https|any"/>
</filter-invocation-definition-source>

<b:bean id="authenticationProcessingFilter"
    class="org.springframework.security.ui.webapp.AuthenticationProcessingFilter">
    <custom-filter before="AUTHENTICATION_PROCESSING_FILTER" /><!-- before|position|after -->
</b:bean>

named-security-filter =
    "FIRST"
    | "CHANNEL_FILTER"
    | "CONCURRENT_SESSION_FILTER"
    | "SESSION_CONTEXT_INTEGRATION_FILTER"
    | "LOGOUT_FILTER"
    | "X509_FILTER"
    | "PRE_AUTH_FILTER"
    | "CAS_PROCESSING_FILTER"
    | "AUTHENTICATION_PROCESSING_FILTER"
    | "OPENID_PROCESSING_FILTER"
    | "BASIC_PROCESSING_FILTER"
    | "SERVLET_API_SUPPORT_FILTER"
    | "REMEMBER_ME_FILTER"
    | "ANONYMOUS_FILTER"
    | "EXCEPTION_TRANSLATION_FILTER"
    | "NTLM_FILTER"
    | "FILTER_SECURITY_INTERCEPTOR"
    | "SWITCH_USER_FILTER"
    | "LAST"

```

D.1. http

```

<http auto-config="boolean"
    create-session="ifRequired|always|never"
    path-type="ant|regex"
    lowercase-comparisons="boolean"
    access-decision-manager-ref="string"
    realm="Spring Security Application"
    session-fixation-protection="none|newSession|migrateSession"
    entry-point-ref="string"
    once-per-request="boolean"

```



```
    access-denied-page="string">
<intercept-url pattern="string"
    access="string"
    method="GET|DELETE|HEAD|OPTIONS|POST|PUT|TRACE"
    filters="none"
    requires-channel="http|https|any"/>
<form-login login-processing-url="string"
    default-target-url="string"
    always-use-default-target="boolean"
    login-page="string"
    authentication-failure-url="string"/>
<openid-login login-processing-url="string"
    default-target-url="string"
    always-use-default-target="boolean"
    login-page="string"
    authentication-failure-url="string"
    user-service-ref="string"/>
<x509 subject-principal-regex="string"
    user-service-ref="string"/>
<http-basic/>
<logout logout-url=""
    logout-success-url=""
    invalidate-session="boolean"/>
<concurrent-session-control max-sessions="positiveInteger"
    expired-url="string"
    exception-if-maximum-exceeded="boolean"
    session-registry-alias="string"
    session-registry-ref="string"/>
<remember-me key="string"
    token-repository-ref="string"
    remember-me-data-source-ref="string"
    remember-me-services-ref="string"
    user-service-ref="string"
    token-validity-seconds="positiveInteger"/>
<anonymous key="string"
    username="string"
    granted-authority="string"/>
<port-mappings>
    <port-mapping http="" https=""/>
</port-mappings>
</http>
```

D.2. authentication-provider

```

<authentication-provider user-service-ref="string">
  <user-service id="string" properties="string">
    <user name="string" password="string" authorities="string" locked="boolean" disabled="boolean"/>
  </user-service>
  <jdbc-user-service id="string"
    data-source-ref="string"
    cache-ref="string"
    users-by-username-query="string"
    authorities-by-username-query="string"
    group-authorities-by-username-query="string"
    role-prefix="string"/>
  <ldap-user-service id="string"
    ldap-server-ref="string"
    user-search-filter="string"
    user-search-base="string"
    group-search-filter="string"
    group-search-base="string"
    group-role-attribute="string"
    cache-ref="string"
    role-prefix="string"
    user-details-class="string"/>
  <password-encoder hash="plaintext|sha|sha-256|md5|md4|{sha}|{sha256}|{md5}|{md4}|{bcrypt}|{pbkdf2}|{argon2}" base64="boolean">
    <salt-source user-property="string" system-wide="string"/>
  </password-encoder>
</authentication-provider>

<authentication-manager alias="string" session-controller-ref="string"/>

<b:bean id="casAuthenticationProvider"
  class="org.springframework.security.providers.cas.CasAuthenticationProvider">
  <custom-authentication-provider/>
</b:bean>

```

D.3. ldap-server

```

<ldap-server id="string"

```

```

    url="string"
    port="integer"
    manager-dn="string"
    manager-password="string"
    ldif="string"
    root="string"
    server-ref="string">
</ldap-server>

<ldap-authentication-provider ldap-server-ref="string"
    user-search-filter="string"
    user-search-base="string"
    group-search-filter="string"
    group-search-base="string"
    group-role-attribute="string"
    cache-ref="string"
    role-prefix="string"
    user-details-class="string">
  <password-compare password-attribute="string" hash="string">
    <password-encoder hash="plaintext|sha|sha-256|md5|md4|{sha}|{sha256}" base64="boolean">
      <salt-source user-property="string" system-wide="string"/>
    </password-encoder>
  </password-compare>
</ldap-authentication-provider>

```

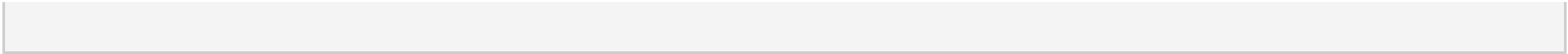
D.4. global-method-security

```

<b:bean id="securedObject"
  class="com.habuma.expectations.springsecurity.intercept.SecuredObject">
  <intercept-methods access-decision-manager-ref="string">
    <protect access="ROLE_SECRET_AGENT" method="getSecuredData"/>
  </intercept-methods>
</b:bean>

<global-method-security secured-annotations="disabled|enabled"
  jsr250-annotations="disabled|enabled"
  access-decision-manager-ref="string">
  <protect-pointcut expression="string" access="string"/>
</global-method-security>

```



[上一页](#)

附录 C. Spring Security-3.0.0.M1

[起始页](#)

[下一页](#)

附录 E. 数据库表结构

附录 E. 数据库表结构

E.1. User

```
create table users(  
    username varchar_ignorecase(50) not null primary key,  
    password varchar_ignorecase(50) not null,  
    enabled boolean not null  
);  
  
create table authorities (  
    username varchar_ignorecase(50) not null,  
    authority varchar_ignorecase(50) not null,  
    constraint fk_authorities_users foreign key(username) references users(username)  
);  
create unique index ix_auth_username on authorities (username,authority);
```

E.2. Group

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

```
constraint fk_group_authorities_group foreign key(group_id) references groups(id)
);

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

E.3. RememberMe

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

E.4. ACL

```
create table acl_sid (
  id bigint generated by default as identity(start with 100) not null primary key,
  principal boolean not null,
  sid varchar_ignorecase(100) not null,
  constraint unique_uk_1 unique(sid,principal)
);

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

create table acl_object_identity (
  id bigint generated by default as identity(start with 100) not null primary key,
```

```
object_id_class bigint not null,  
object_id_identity bigint not null,  
parent_object bigint,  
owner_sid bigint not null,  
entries_inheriting boolean not null,  
constraint unique_uk_3 unique(object_id_class,object_id_identity),  
constraint foreign_fk_1 foreign key(parent_object) references acl_object_identity(id),  
constraint foreign_fk_2 foreign key(object_id_class) references acl_class(id),  
constraint foreign_fk_3 foreign key(owner_sid) references acl_sid(id)  
);  
  
create table acl_entry (  
  id bigint generated by default as identity(start with 100) not null primary key,  
  acl_object_identity bigint not null,ace_order int not null,sid bigint not null,  
  mask integer not null,granting boolean not null,audit_success boolean not null,  
  audit_failure boolean not null,  
  constraint unique_uk_4 unique(acl_object_identity,ace_order),  
  constraint foreign_fk_4 foreign key(acl_object_identity) references acl_object_identity(id),  
  constraint foreign_fk_5 foreign key(sid) references acl_sid(id)  
);
```

[上一页](#)[附录 D. 命名空间](#)[起始页](#)[下一页](#)[附录 F. 异常](#)

附录 F. 异常

org.springframework.security

AccessDeniedException, 当用户无权访问被保护资源时抛出。

AccountExpiredException, 当用户过期时抛出。

AccountStatusException, 当用户状态不正常时抛出。

AuthenticationCredentialsNotFoundException, 当找不到验证凭证时抛出。

AuthenticationException, 验证异常。

AuthenticationServiceException, 验证服务异常。

AuthorizationServiceException, 认证服务异常。

BadCredentialsException, 凭证（密码）错误。

CredentialsExpiredException, 凭证过期异常。

DisabledException, 无效异常。

InsufficientAuthenticationException, 不满足验证异常。

LockedException, 锁定异常。

SpringSecurityException, 安全异常。

org.springframework.security.concurrent

ConcurrentLoginException，同步登陆异常。

SessionAlreadyUsedException，会话已存在异常。

org.springframework.security.config

SecurityConfigurationException，安全配置异常。

org.springframework.security.ldap

LdapDataAccessException，ldap数据访问异常。

org.springframework.security.provider

ProviderNotFoundException，找不到provider异常。

org.springframework.security.provider.rcp

RemoteAuthenticationException，远程认证异常。

org.springframework.security.userdetails

UsernameNotFoundException，找不到用户名异常。

org.springframework.security.userdetails.hierarchicalroles

CycleInRoleHierarchyException, 角色循环继承异常。

org.springframework.security.ui.digestauth

NonceExpiredException, nonce过期异常。

org.springframework.security.ui.preauth

PreAuthenticatedCredentialsNotFoundException, 未找到预验证凭证异常。

org.springframework.security.ui.rememberme

CookieTheftException, cookie被盗异常

InvalidCookieException, 非法cookie异常。

RememberMeAuthenticationException, rememberme验证异常。

[上一页](#)

附录 E. 数据库表结构

[起始页](#)

[下一页](#)

附录 G. 事件

[上一页](#)

附录 G. 事件

认证事件

AuthenticationFailureConcurrentLoginEvent验证失败，同时登陆。

AuthenticationFailureCredentialsExpiredEvent验证失败，凭证失效。

AuthenticationFailureDisabledEvent验证失败，禁用。

AuthenticationFailureExpiredEvent验证失败，失效。

AuthenticationFailureLockedEvent验证失败，锁定。

AuthenticationFailureProviderNotFoundEvent验证失败，找不到provider。

AuthenticationFailureProxyUntrustedEvent验证失败，不可信任的代理。

AuthenticationFailureServiceExceptionEvent验证失败，服务异常

AuthenticationSuccessEvent认证成功。

AuthenticationSwitchUserEvent切换用户。

InteractiveAuthenticationSuccessEvent内部验证成功。

验证事件

AuthenticationCredentialsNotFoundEvent找不到凭证。

AuthorizationFailureEvent认证失败。

PublicInvocationEvent公用调用。

[上一页](#)

附录 F. 异常

[起始页](#)