

Spring Security 权限管理开发手册

序言	7
为啥选择Spring Security	7
内容结构组织	7
意见反馈	7
相关信息	8
部分 I. 基础篇	8
第 1 章 一个简单的HelloWorld	9
1.1. 配置过滤器	9
1.2. 使用命名空间	10
1.3. 完善整个项目	11
1.4. 运行示例	12
第 2 章 使用数据库管理用户权限	15
2.1. 修改配置文件	15
2.2. 数据库表结构	17
第 3 章 自定义数据库表结构	18
3.1. 自定义表结构	18
3.2. 初始化数据	20
3.3. 获得自定义用户权限信息	20
3.3.1. 处理用户登陆	20
3.3.2. 检验用户权限	21
第 4 章 自定义登陆页面	22
4.1. 实现自定义登陆页面	22
4.2. 修改配置文件	22
4.3. 登陆页面中的参数配置	23
4.4. 测试一下	24
第 5 章 使用数据库管理资源	25
5.1. 数据库表结构	25
5.2. 初始化数据	27
5.3. 实现从数据库中读取资源信息	28
5.3.1. 需要何种数据格式	28
5.3.2. 替换原有功能的切入点	33
第 6 章 控制用户信息	36
6.1. MD5 加密	37
6.2. 盐值加密	37
6.3. 用户信息缓存	38
注意	40
6.4. 获取当前用户信息	40
第 7 章 自定义访问拒绝页面	41
第 8 章 动态管理资源结合自定义登录页面	43
部分 II. 保护web篇	43

第 9 章 图解过滤器.....	45
9.1. HttpSessionContextIntegrationFilter.....	46
9.2. LogoutFilter	47
9.3. AuthenticationProcessingFilter	48
9.4. DefaultLoginPageGeneratingFilter	49
9.5. BasicProcessingFilter.....	50
9.6. SecurityContextHolderAwareRequestFilter	51
9.7. RememberMeProcessingFilter.....	51
9.8. AnonymousProcessingFilter	52
9.9. ExceptionTranslationFilter	53
9.10. SessionFixationProtectionFilter	54
9.11. FilterSecurityInterceptor	55
第 10 章 管理会话.....	55
10.1. 添加监听器.....	56
10.2. 添加过滤器.....	56
10.3. 控制策略.....	56
10.3.1. 后登陆的将先登录的踢出系统.....	56
10.3.2. 后面的用户禁止登陆.....	57
第 11 章 单点登录.....	58
11.1. 配置JA-SIG.....	58
11.2. 配置Spring Security	60
11.2.1. 添加依赖.....	60
11.2.2. 修改applicationContext.xml	61
11.3. 运行配置了cas的子系统.....	63
11.4. 为cas配置SSL.....	65
11.4.1. 生成密钥.....	66
11.4.2. 为jetty配置SSL.....	66
11.4.3. 为tomcat配置SSL	67
第 12 章 basic认证	67
12.1. 配置basic验证	68
12.2. 编程实现basic客户端	69
第 13 章 标签库.....	70
13.1. 配置taglib.....	70
13.2. authenticaiton	70
13.3. authorize.....	71
13.4. acl/accesscontrollist	71
13.5. 为不同用户显示各自的登陆成功页面	72
第 14 章 自动登录.....	72
14.1. 默认策略.....	72
14.2. 持久化策略.....	73
注意.....	74
第 15 章 匿名登录.....	75
15.1. 配置文件.....	75
15.2. 修改默认用户名	76

15.3. 匿名用户的限制.....	77
第 16 章 防御会话伪造.....	78
16.1. 攻击场景.....	78
16.2. 解决会话伪造.....	79
第 17 章 预先认证.....	79
17.1. 为jetty配置Realm.....	79
17.2. 配置Spring Security.....	81
第 18 章 切换用户.....	83
18.1. 配置方式.....	83
18.2. 实例演示.....	84
第 19 章 信道安全.....	85
19.1. 设置信道安全.....	85
19.2. 指定http和https的端口.....	85
第 20 章 digest认证.....	86
20.1. 配置digest验证.....	86
20.2. 使用ajax实现digest认证.....	87
20.3. 编程实现digest客户端.....	88
第 21 章 通过LDAP获取用户信息.....	88
第 22 章 通过OpenID进行登录.....	90
22.1. 配置.....	90
22.2. 系统时间问题.....	92
第 23 章 使用X509 登录.....	93
23.1. 生成证书.....	93
23.2. 配置服务器使用双向加密.....	94
23.3. 配置X509 认证.....	94
第 24 章 使用NTLM登录（无法成功登陆）.....	95
警告.....	95
第 25 章 使用JAAS机制.....	98
第 26 章 使用HttpInvoker.....	102
第 27 章 使用rmi.....	103
第 28 章 控制portal的权限.....	103
第 29 章 保存登录之前的请求.....	105
部分 III. 内部机制篇.....	106
第 30 章 保护方法调用.....	106
30.1. 控制全局范围的方法权限.....	106
30.2. 控制某个bean内的方法权限.....	108
使用intercept-methods面临着几个问题.....	108
30.3. 使用annotation控制方法权限.....	108
30.3.1. 使用Secured.....	109
30.3.2. 使用jsr250.....	109
第 31 章 权限管理的基本概念.....	111
31.1. 认证与验证.....	111
31.2. SecurityContext安全上下文.....	111
31.3. Authentication验证对象.....	112

第 32 章 Voter 表决者	113
32.1. Voter 表决者	113
32.2. RoleVoter	114
32.3. AuthenticatedVoter	114
32.4. AbstractAclVoter	115
第 33 章 拦截器	115
33.1. 权限配置数据源	115
33.2. 权限管理器	116
33.3. 后置调用管理器	117
33.4. 临时分配额外权限	117
第 34 章 用户信息	118
34.1. UserDetails	118
34.2. 使用角色继承	118
34.3. 为 ACL 添加角色继承	119
34.4. PasswordEncoder 和 Salt Value	122
第 35 章 集成 jcaptcha	123
第 36 章 动态资源管理	124
36.1. 基本知识	124
36.2. 读取资源	124
36.3. URL 资源扩展点	125
36.4. METHOD 资源扩展点	126
第 37 章 扩展 UserDetails	126
37.1. 实现 UserDetails 接口	126
37.2. 实现 UserDetailsService 接口	128
37.3. 修改配置文件	129
37.4. 测试运行	129
第 38 章 锁定用户	130
第 39 章 设置过滤器链	132
警告	132
第 40 章 自定义过滤器	134
第 41 章 使用用户组	136
41.1. 数据库结构	136
41.2. 修改配置文件	138
第 42 章 在 JSF 中使用 Spring Security	138
42.1. 修改过滤器支持 forward	139
42.2. 自定义登录页面	139
42.3. 显示密码错误信息	140
第 43 章 自定义会话管理	142
43.1. 默认策略的缺陷	142
43.2. 记录用户名与 ip	143
43.3. 改造控制类	144
43.4. 修改配置文件	145
第 44 章 匹配 URL 地址	146
44.1. AntUriPathMatcher	146

44.2. RegexUrlPathMatcher.....	147
44.3. lowercase-comparisons	147
第 45 章 配置过滤器.....	148
45.1. 标准过滤器.....	148
45.2. 在http中启用标准过滤器.....	149
45.3. 为自定义过滤器设置位置.....	150
部分 IV. ACL篇	151
第 46 章 ACL基本操作	151
46.1. 准备数据库和aclService	151
46.1.1. 为acl配置cache	151
46.1.2. 配置lookupStrategy	152
46.1.3. 配置aclService	153
46.2. 使用aclService管理acl信息	153
46.3. 使用acl控制delete操作.....	154
46.4. 控制用户可以看到哪些信息.....	156
第 47 章 管理acl.....	158
47.1. 管理多个domain类.....	158
47.2. 动态授权与收回授权.....	159
47.2.1. 获得对象的acl权限	159
47.2.2. 添加授权.....	160
47.2.3. 收回授权.....	161
第 48 章 acl自动提醒	162
48.1. 自动创建acl.....	162
48.2. 自动删除acl.....	164
48.3. 根据id删除acl.....	166
部分 V. 最佳实践篇	168
第 49 章 最简控制台	169
49.1. 平台搭建.....	169
49.2. 用户登录.....	171
49.3. 用户信息列表.....	172
49.4. 添加用户.....	173
49.5. 修改用户信息.....	176
49.6. 修改自己的密码.....	177
第 50 章 用户组控制台	179
50.1. 添加对用户组的支持.....	179
50.2. 浏览用户组.....	179
50.3. 创建用户组.....	180
50.4. 修改用户组.....	181
附录 B. 常见问题解答	182
附录 C. Spring Security-3.0.0.M1	184
C.1. Hello World	185
C.2. Spring-EL	185
C.3. RoleHierarchy	187
C.4. Success Handler	187

C.5. REST下的权限控制	188
附录 D. 命名空间	188
D.1. http	190
D.2. authentication-provider	191
D.3. ldap-server	192
D.4. global-method-security	193
附录 E. 数据库表结构	193
E.1. User	193
E.2. Group	194
E.3. RememberMe	194
E.4. ACL	194
附录 F. 异常	195
org.springframework.security	195
org.springframework.security.concurrent	196
org.springframework.security.config	196
org.springframework.security.ldap	196
org.springframework.security.provider	196
org.springframework.security.provider.rcp	197
org.springframework.security.userdetails	197
org.springframework.security.userdetails.hierarchicalroles	197
org.springframework.security.ui.digestauth	197
org.springframework.security.ui.preauth	197
org.springframework.security.ui.rememberme	197
附录 G. 事件	197
认证事件	197
验证事件	198
附录 H. RBAC模型（转载）	198
H.1. RBAC模型介绍	199
H.2. 有关概念	199
H.2.1. 什么是角色	199
H.2.2. 角色与用户组	200
H.3. 基本模型RBAC ₀	202
H.3.1. RBAC ₀ 模型的形式定义如下	202
H.4. 角色分级模型RBAC ₁	203
H.4.1. 定义 2: RBAC ₁ 由以下内容确定	204
H.5. 限制模型RBAC ₂	204
H.5.1. 定义 3:	204
H.6. 统一模型RBAC ₃	206
H.7. 定义 4	207
H.8. 在ARBAC97 中，包括三种组件	208
H.9. RBAC模型的特点	208

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。

内容组织结构

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 “内部机制篇”。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，没人敢说自己不会犯错，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。

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:ch001:war:0.1
[INFO] \-
[INFO] org.springframework.security:spring-security-taglibs:jar:2.0.5.RELEASE:compile
[INFO] +-
[INFO] org.springframework.security:spring-security-core:jar:2.0.5.RELEASE: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:2.0.5.RELEASE: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

```

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：

```

+ ch001/
  + 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 ch001 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/ch001/>就可以看到登陆页面。

Login with Username and Password

User:

Password:

☐ Remember me on this computer.

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.

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。

HTTP ERROR 403

Problem accessing /helloworld/admin.jsp. Reason:

Access is denied

Powered by Jetty://

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。

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>
  <del>user-service</del>
  <del>user name="admin" password="admin" authorities="ROLE_USER, ROLE_ADMIN" />
  <del>user name="user" password="user" authorities="ROLE_USER" />
</del>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 users(❶
    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 j2ee 容器统一管理提供的。

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：

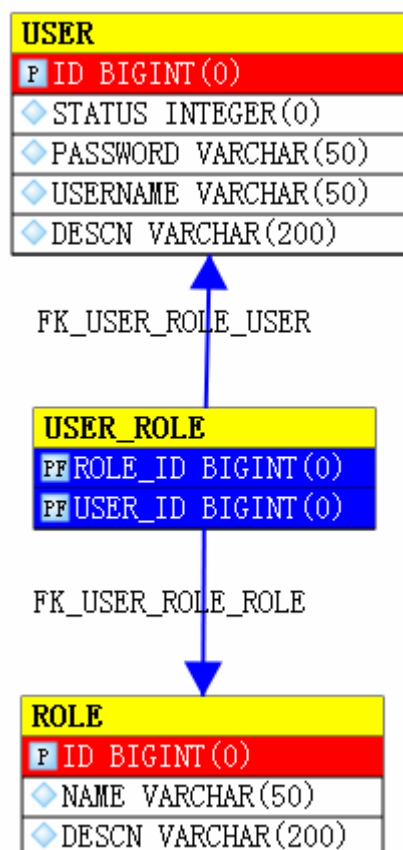


图 3.1. 数据库表关系

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。

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。

```
+ ch004/
+ 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_check❶"
style="width:260px;text-align:center;">
    <fieldset>
        <legend>登陆</legend>
        用户: <input type="text" name="j_username❷" style="width:150px;"
value="${sessionScope['SPRING_SECURITY_LAST_USERNAME']}" /><br />
        密码: <input type="password" name="j_password❸" style="width:150px;" /><br />
        <input type="checkbox" name="_spring_security_remember_me❹" />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

④ `_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。

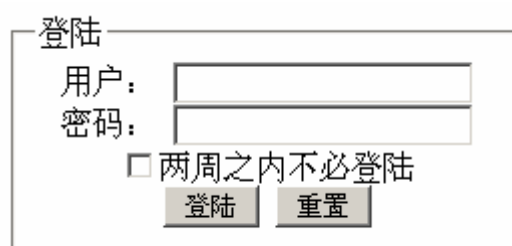


图 4.1. 进入登录页面

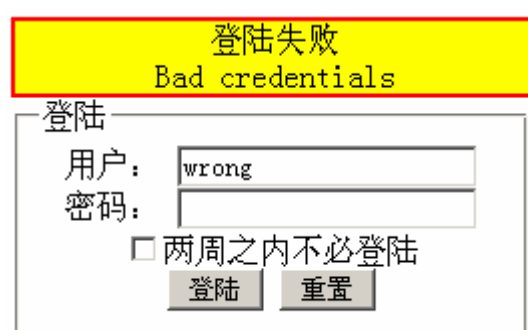


图 4.2. 用户登陆失败

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>

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),
    priority integer,
    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：

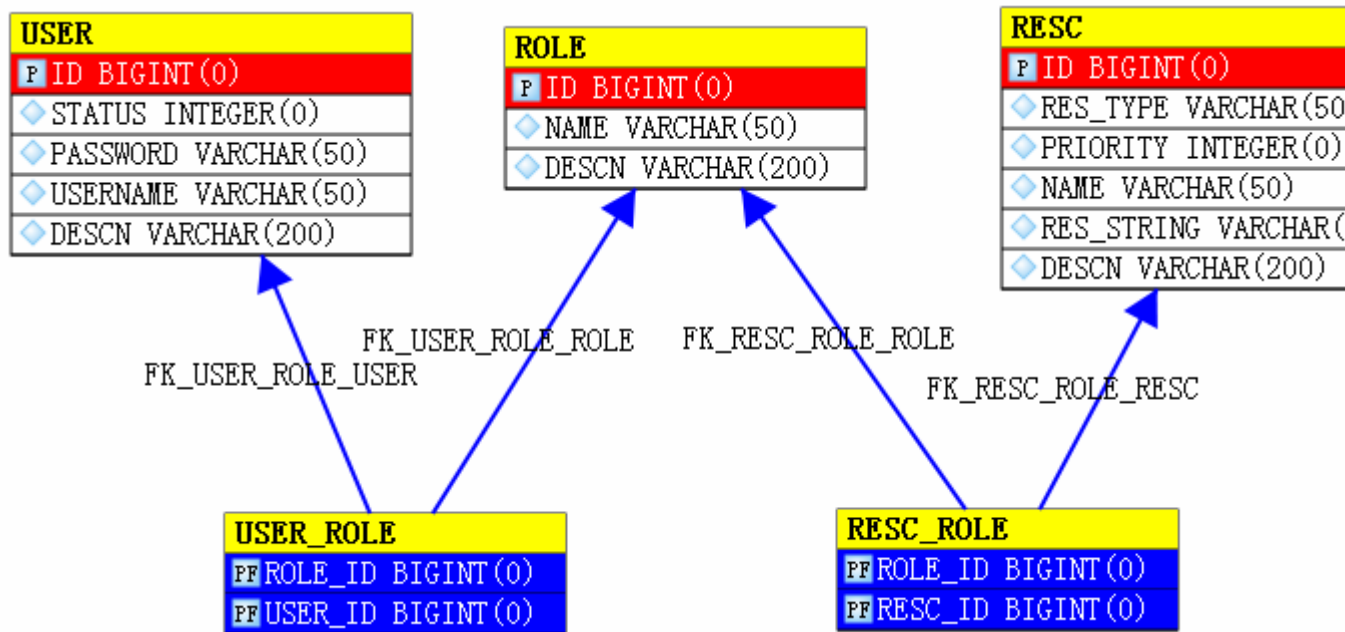


图 5.1. 数据库表关系

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

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" />
<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
 order by re.priority
```

下面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 Map<String, String> findResources() {
    ResourceMapping resourceMapping = new ResourceMapping(getDataSource(),
        resourceQuery);

    Map<String, String> resourceMap = new LinkedHashMap<String, String>();

    for (Resource resource : (List<Resource>) resourceMapping.execute()) {
        String url = resource.getUrl();
        String role = resource.getRole();

        if (resourceMap.containsKey(url)) {
            String value = resourceMap.get(url);
            resourceMap.put(url, value + ", " + role);
        } else {
            resourceMap.put(url, role);
        }
    }

    return resourceMap;
}
```

2. 使用获得的资源信息组装 requestMap。

```
3.    protected LinkedHashMap<RequestKey, ConfigAttributeDefinition>
      buildRequestMap() {
4.        LinkedHashMap<RequestKey, ConfigAttributeDefinition> requestMap = null;
5.        requestMap = new LinkedHashMap<RequestKey,
      ConfigAttributeDefinition>();
6.
7.        ConfigAttributeEditor editor = new ConfigAttributeEditor();
8.
9.        Map<String, String> resourceMap = this.findResources();
10.
11.       for (Map.Entry<String, String> entry : resourceMap.entrySet()) {
12.           RequestKey key = new RequestKey(entry.getKey(), null);
13.           editor.setAsText(entry.getValue());
14.           requestMap.put(key,
15.               (ConfigAttributeDefinition) editor.getValue());
16.       }
17.
18.       return requestMap;
19.   }
```

20. 使用 urlMatcher 和 requestMap 创建 DefaultFilterInvocationDefinitionSource。

```
21.   public Object getObject() {
22.       return new DefaultFilterInvocationDefinitionSource(this
23.           .getUrlMatcher(), this.buildRequestMap());
24.   }
```

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.ch005;

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

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

```

import java.util.Map;

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 Map<String, String> findResources() {
        ResourceMapping resourceMapping = new ResourceMapping(getDataSource(),
            resourceQuery);

        Map<String, String> resourceMap = new LinkedHashMap<String, String>();

        for (Resource resource : (List<Resource>) resourceMapping.execute()) {
            String url = resource.getUrl();

```

```

        String role = resource.getRole();

        if (resourceMap.containsKey(url)) {
            String value = resourceMap.get(url);
            resourceMap.put(url, value + "," + role);
        } else {
            resourceMap.put(url, role);
        }
    }

    return resourceMap;
}

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

    ConfigAttributeEditor editor = new ConfigAttributeEditor();

    Map<String, String> resourceMap = this.findResources();

    for (Map.Entry<String, String> entry : resourceMap.entrySet()) {
        RequestKey key = new RequestKey(entry.getKey(), null);
        editor.setAsText(entry.getValue());
        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.ch005.JdbcFilterInvocationDefinitionSourceF
actoryBean">
    <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
        order by priority
    "/>
</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
            order by priority
        " />
    </beans:bean>

    <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>
```

实例见 ch05。

目前存在的问题是，系统会在初始化时一次将所有资源加载到内存中，即使在数据库中修改了资源信息，系统也不会再次去从数据库中读取资源信息。这就造成了每次修改完数据库后，都需要重启系统才能时资源配置生效。

解决方案是，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="/" />
```

目前还不支持对方法调用和ACL资源的动态管理，相关讨论请参考手册后面的部分 [第 36 章 动态资源管理](#)。

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^[1]。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。

第 7 章 自定义访问拒绝页面

在我们的例子中, user 用户是不能访问/admin.jsp 页面的, 当我们使用 user 用户登录系统之后, 访问/admin.jsp 时系统默认会返回 403 响应。

HTTP ERROR 403

Problem accessing /helloworld/admin.jsp. Reason:

Access is denied

Powered by Jetty://

图 7.1. 403 响应

如果我们希望自定义访问拒绝页面, 只需要随便创建一个 jsp 页面, 让后将这个页面的位置放到配置文件中。

下面创建一个 accessDenied.jsp

```

<%@ page contentType="text/html; charset=UTF-8"%>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>Access Denied</title>
    <style type="text/css">
div.error {
  width: 260px;
  border: 2px solid red;
  background-color: yellow;
  text-align: center;
}
    </style>
  </head>
  <body>
    <h1>Access Denied</h1>
    <hr>
    <div class="error">
      访问被拒绝<br>
      ${requestScope['SPRING_SECURITY_403_EXCEPTION'].message}
    </div>
    <hr>
  </body>
</html>

```

下一步修改配置文件，添加自定义访问拒绝页面的地址。

```

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

```

现在访问拒绝的页面就变成了下面这样：

Access Denied

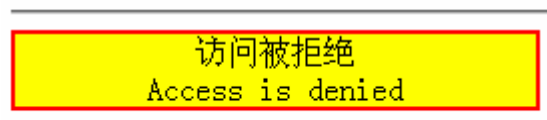


图 7.2. 自定义访问拒绝页面

实例在 ch007。

第 8 章 动态管理资源结合自定义登录页面

如果想将动态管理资源与自定义登录页面一起使用，最简单的办法就是在数据库中将登录页面对应的权限设置为 IS_AUTHENTICATED_ANONYMOUSLY。

因此在数据库中添加一条资源信息。

```
INSERT INTO RESC VALUES(1,'','URL','/login.jsp*',1,'')
```

这里的/login.jsp*就是我们自定义登录页面的地址。

然后为匿名用户添加一条角色信息：

```
INSERT INTO ROLE VALUES(3,'IS_AUTHENTICATED_ANONYMOUSLY','anonymous')
```

最后为这两条记录进行关联即可。

```
INSERT INTO RESC_ROLE VALUES(1,3)
```

这样就实现了将动态管理资源与自定义登录页面进行结合。

实例在 ch008。

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=1
300; ],

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 9 章 图解过滤器

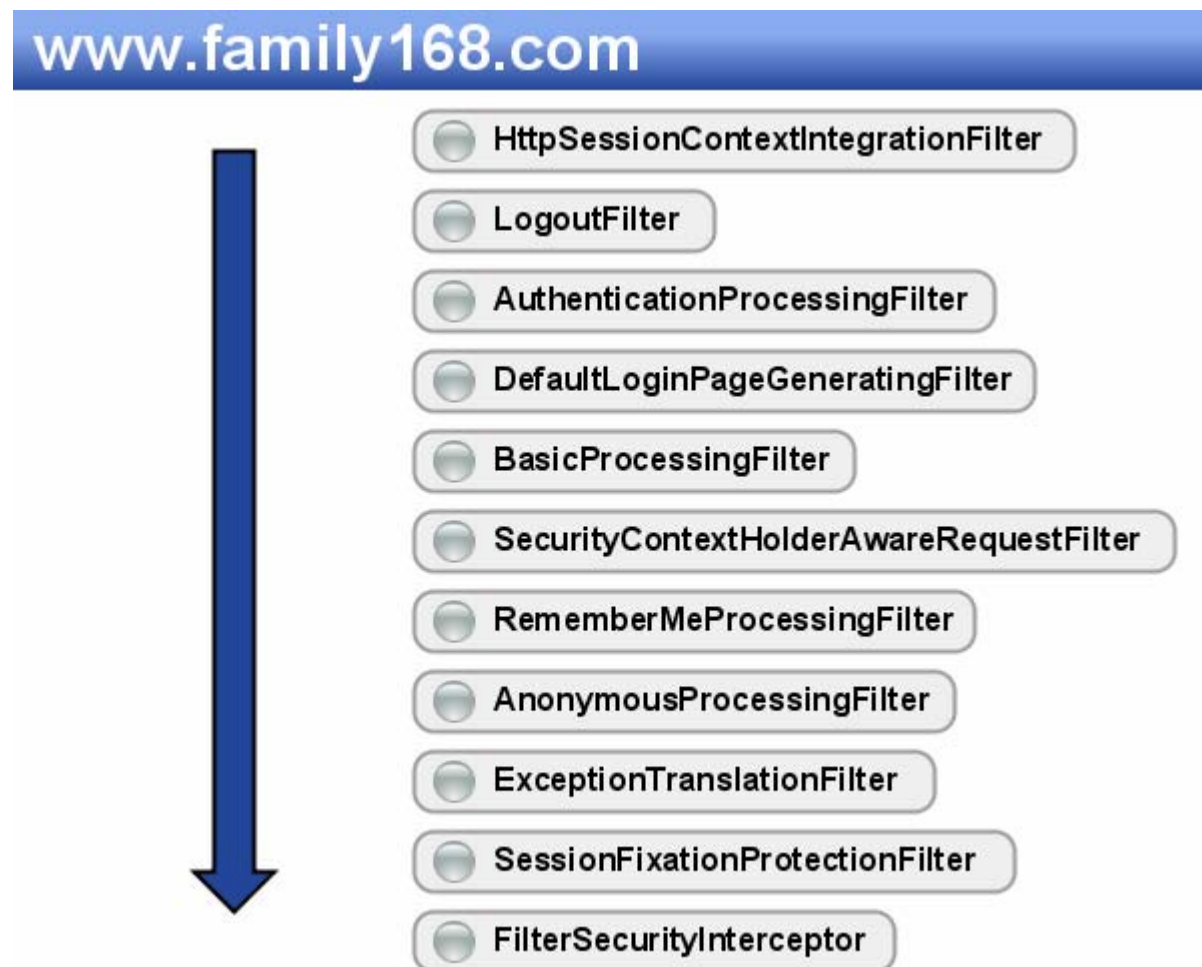


图 9.1. `auto-config='true'`时的过滤器列表

9.1. HttpSessionContextIntegrationFilter

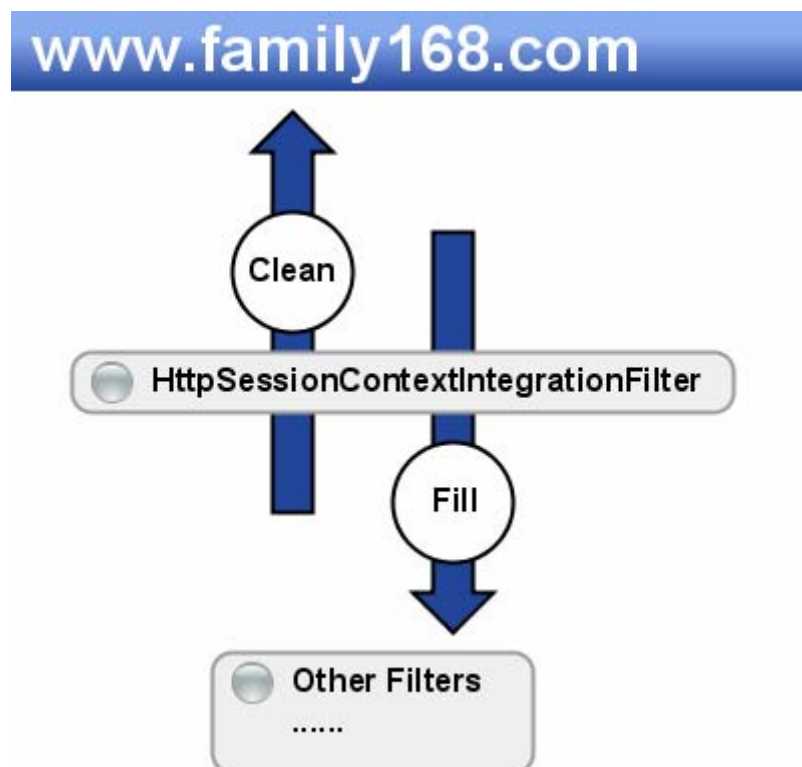


图 9.2. `org.springframework.security.context.HttpSessionContextIntegrationFilter`

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。

9.2. LogoutFilter

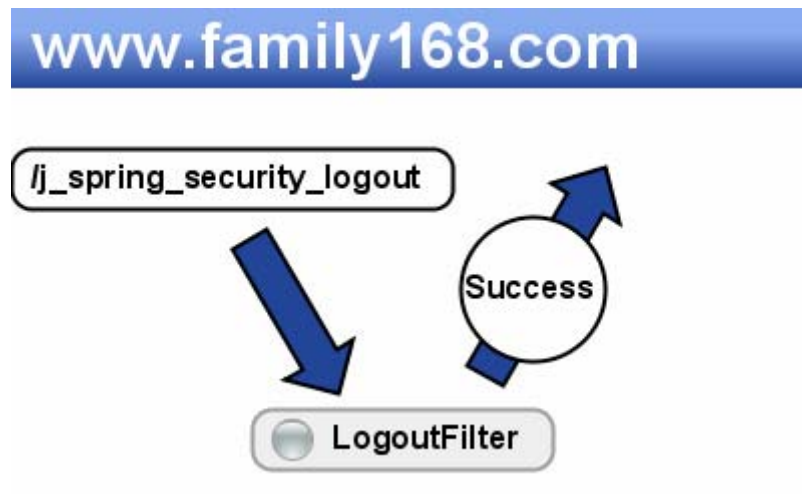


图 9.3. org.springframework.security.ui.logout.LogoutFilter

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。

9.3. AuthenticationProcessingFilter

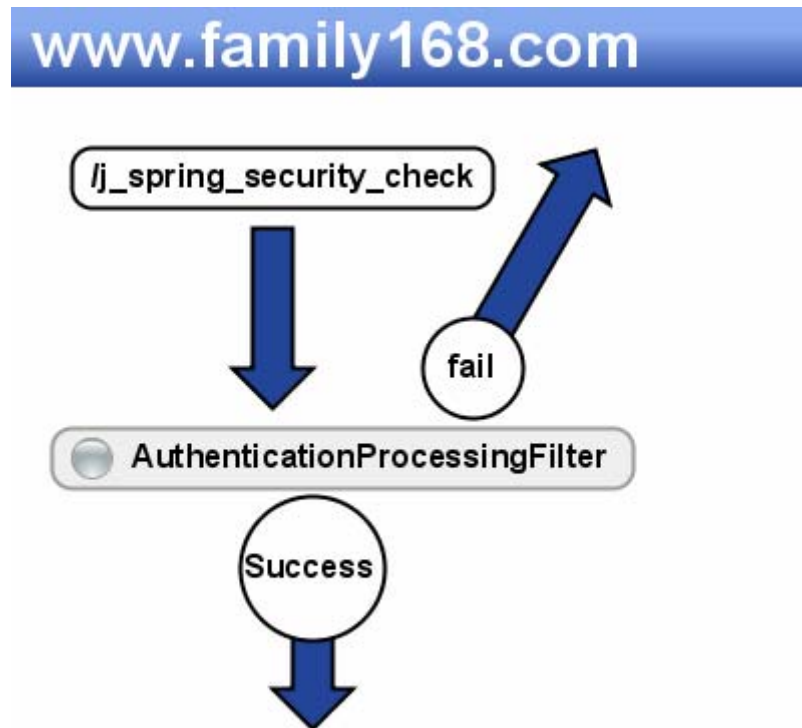


图 9.4. org.springframework.security.ui.webapp.AuthenticationProcessingFilter

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。

9.4. DefaultLoginPageGeneratingFilter

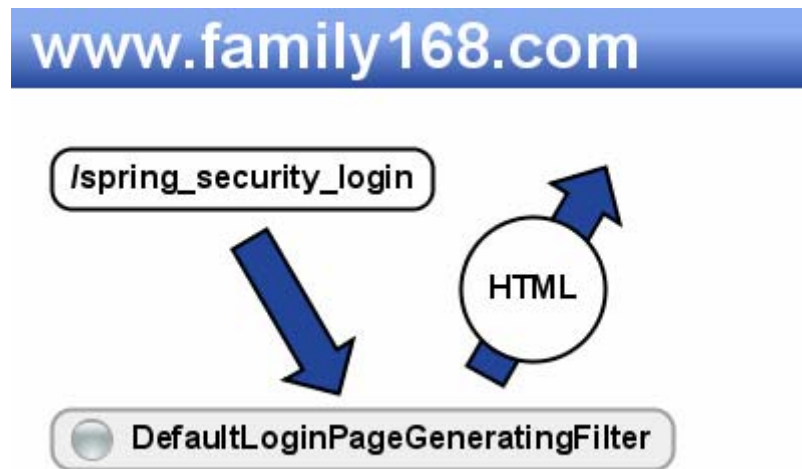


图 9.5. org.springframework.security.ui.webapp.DefaultLoginPageGeneratingFilter

此过滤器用来生成一个默认登录页面，默认访问地址为 `/spring_security_login`，这个默认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](#)。

9.5. BasicProcessingFilter

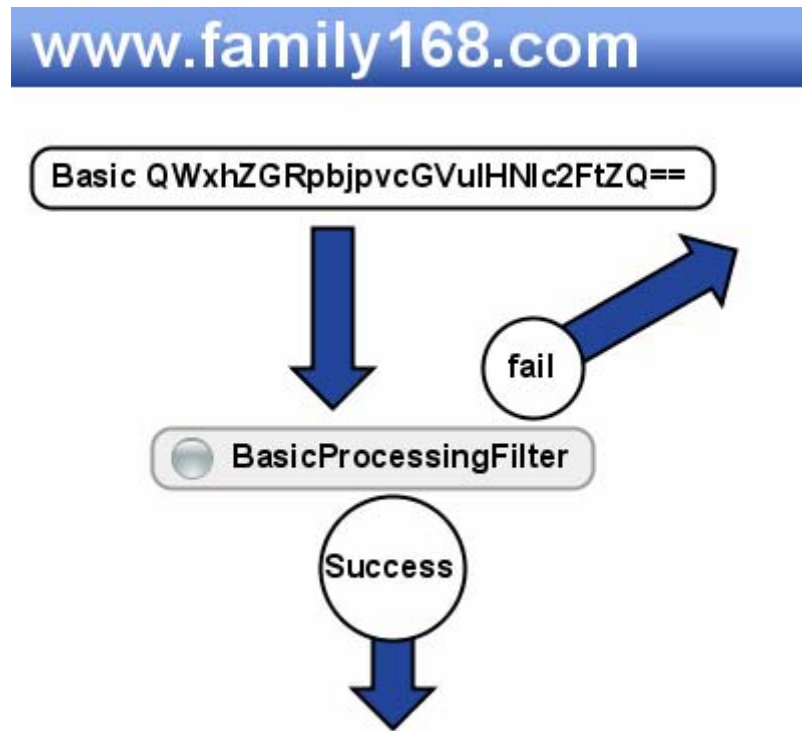


图 9.6. org.springframework.security.ui.basicauth.BasicProcessingFilter

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 12 章 basic 认证](#)。

9.6. SecurityContextHolderAwareRequestFilter

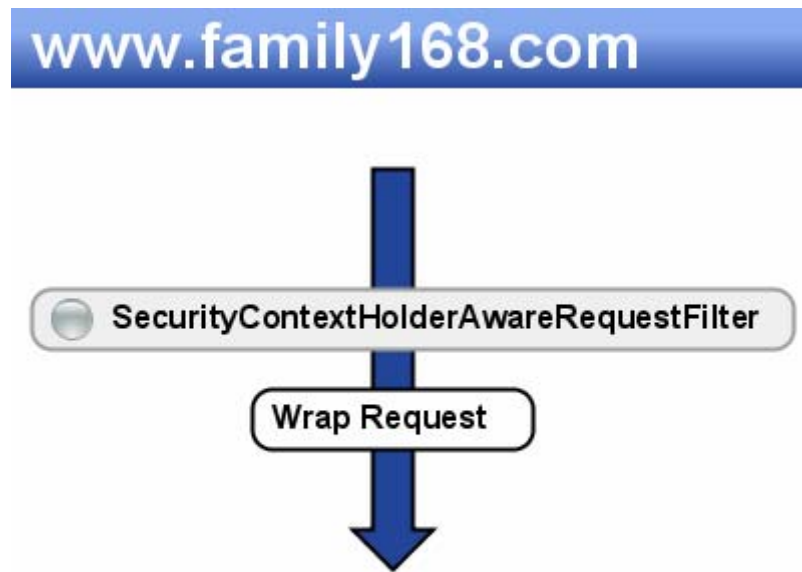


图 9.7. `org.springframework.security.wrapper.SecurityContextHolderAwareRequestFilter`

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。

9.7. RememberMeProcessingFilter

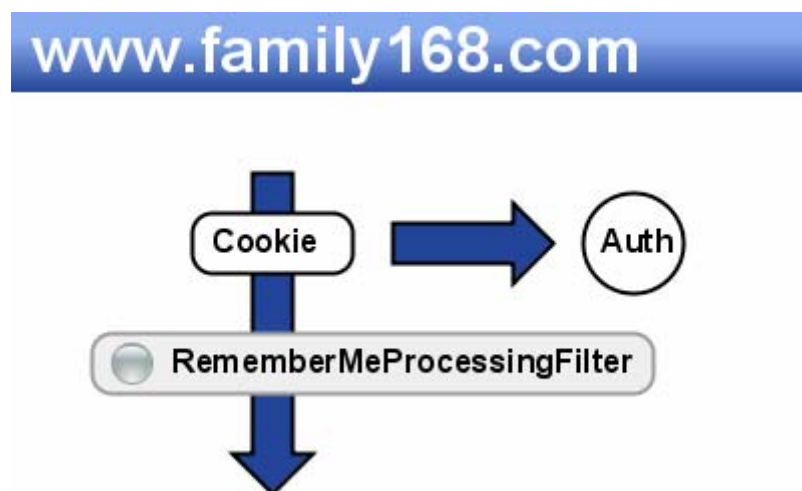


图 9.8. org.springframework.security.ui.rememberme.RememberMeProcessingFilter

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 14 章 自动登录](#)。

9.8. AnonymousProcessingFilter

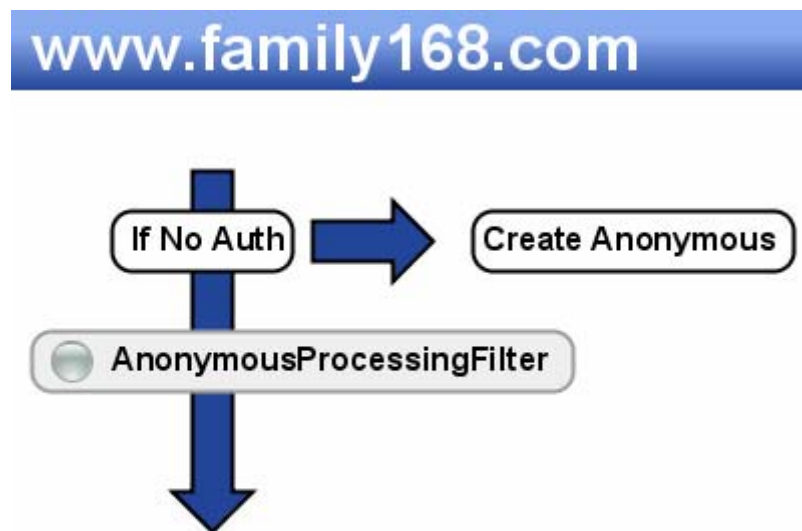


图 9.9. org.springframework.security.providers.anonymous.AnonymousProcessingFilter

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 15 章 匿名登录](#)。

9.9. ExceptionTranslationFilter

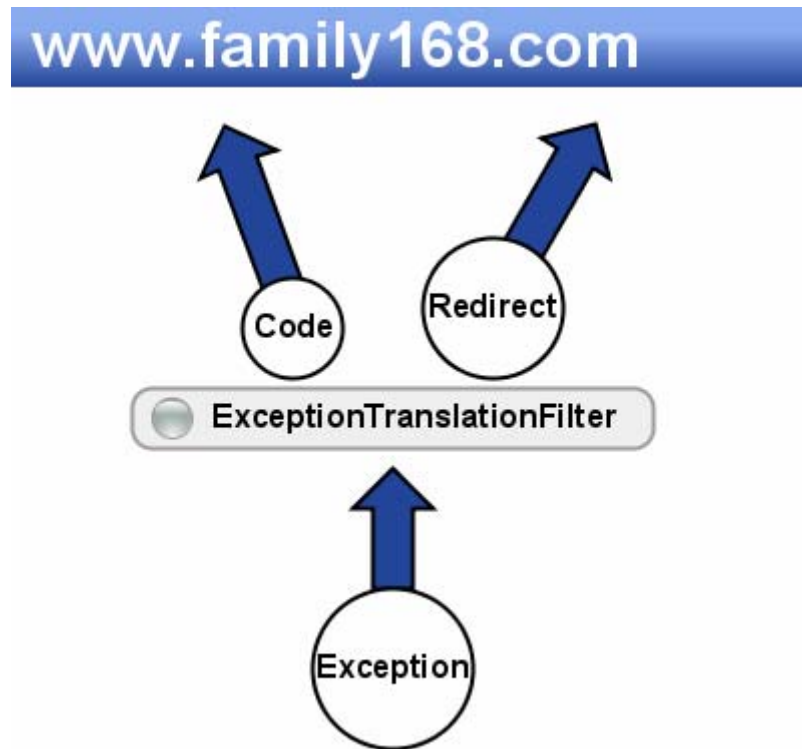


图 9.10. `org.springframework.security.ui.ExceptionTranslationFilter`

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。

9.10. SessionFixationProtectionFilter



图 9.11. 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 16 章 防御会话伪造](#)。

9.11. FilterSecurityInterceptor

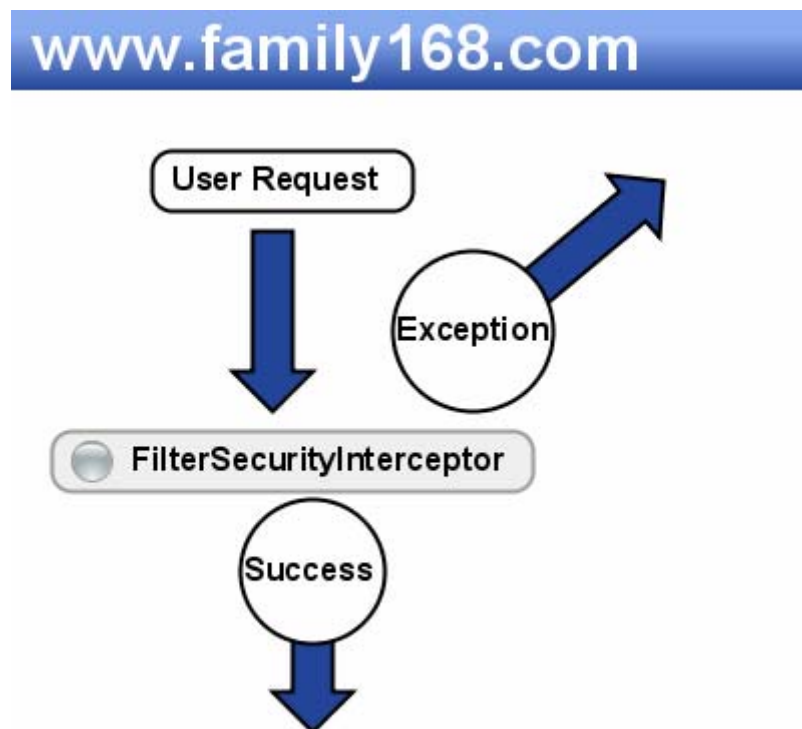


图 9.12. org.springframework.security.intercept.web.FilterSecurityInterceptor

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 10 章 管理会话

多个用户不能使用同一个账号同时登陆系统。

10.1. 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>

10.2. 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。

10.3. 控制策略

10.3.1. 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。

10.3.2. 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 10.1. 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`。

第 11 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。

11.1. 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 `ch103\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。

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，请规

Languages:

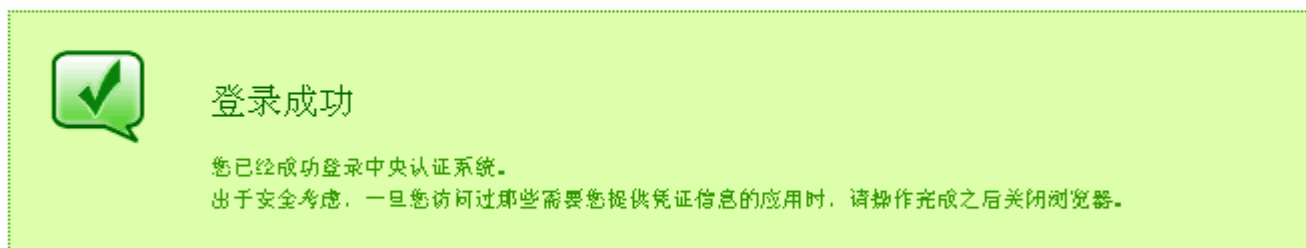
[English](#) | [Spanish](#) | [French](#) | [Russian](#) | [Nederlands](#) | [Svenskt](#)
(Simplified) | [Deutsch](#) | [Japanese](#) | [Croatian](#) | [Czech](#) | [Slove](#)

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

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

图 11.1. 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。



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

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

图 11.2. 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。

11.2. 配置Spring Security

11.2.1. 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>2.0.5.RELEASE</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-2.0.5.RELEASE.jar
spring-dao-2.0.8.jar
aopalliance-1.0.jar
cas-client-core-3.1.5.jar
```

大家可以去 spring 和 ja-sig 的网站去寻找这些依赖库。

11.2.2. 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/ch103/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="ch103" />
</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。

11.3. 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 ch103 执行 `run.bat` 启动子系统。

现在直接访问 `http://localhost:8080/ch103/` 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，请规划

Languages:

[English](#) | [Spanish](#) | [French](#) | [Russian](#) | [Nederlands](#) | [Svenskt](#)
(Simplified) | [Deutsch](#) | [Japanese](#) | [Croatian](#) | [Czech](#) | [Slovene](#)

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

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

图 11.3. 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 11.4. 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 11.5. cas-logout.jsp

在此点击 Logout of CAS 会跳转至 cas 进行注销。



图 11.6. 注销成功

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

11.4. 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。

11.4.1. 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 `ch103/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。

11.4.2. 为jetty配置SSL

jetty 的配置可参考 `ch103` 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>

```

11.4.3. 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。

第 12 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。

12.1. 配置basic验证

如果在 http 中配置了 auto-config="true"我们就不要再添加任何配置了，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 dXNlcjpic2Vy
Cookie	JSESSIONID=11etjbranwm4g

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

图 12.1. 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。

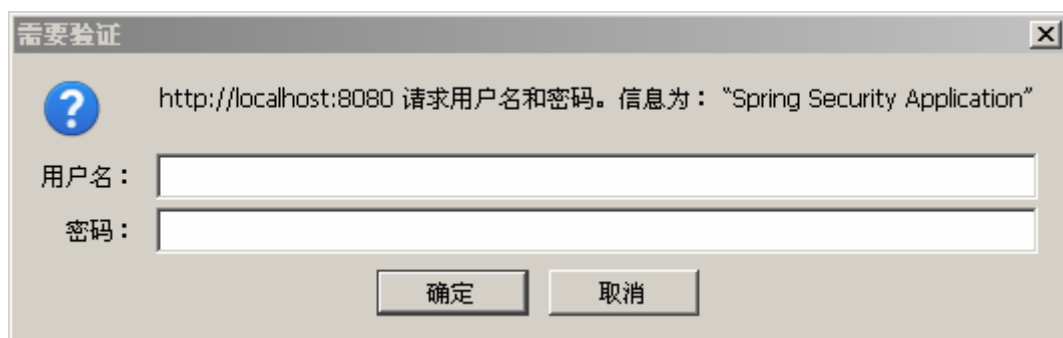


图 12.2. HTTP 请求头

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。

12.2. 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/ch104/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/ch104/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/ch104/basic.jsp 找到，可以运行 ch104，然后访问 <http://localhost:8080/ch104/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。

第 13 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。

13.1. 配置taglib

如果需要使用 taglib，首先要把 spring-security-taglibs-2.0.5.RELEASE.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"%>
```

13.2. 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>
```

13.3. 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。

13.4. 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 [第 46 章 ACL 基本操作](#)。

13.5. 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，管理员登录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。

第 14 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。

14.1. 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

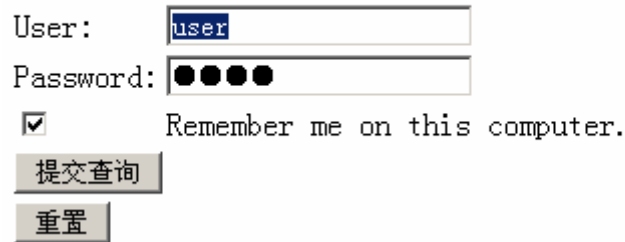


图 14.1. 选中 rememberMe

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。

```
username : roleAnonymous | authorities: ROLE_ANONYMOUS
```

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

图 14.2. 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。

14.2. 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。

第 15 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。

15.1. 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。

这里的 `IS_AUTHENTICATED_ANONYMOUSLY` 将会交由 `AuthenticatedVoter` 处理,内部会依据 `AuthenticationTrustResolver` 判断当前登录的用户是否是匿名用户。

```
<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。

图 15.1. 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 `roleAnonymous`，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一个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。

15.2. 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 15.2. 修改匿名用户名称

15.3. 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 15.3. 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。

第 16 章 防御会话伪造

16.1. 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=lpjztz08i2u4i
```

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。

16.2. 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 migrationSession，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。

第 17 章 预先认证

预先认证是指用户在进入系统之前，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。

17.1. 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。

17.2. 配置Spring Security

因为使用容器 Realm 的 Pre-Auth 并没有对应的命名空间，所以需要去掉 `auto-config="true"` 并引用对应的安全入口。

```
<sec:http entry-point-ref="preauthEntryPoint">
  <sec:intercept-url pattern="/admin.jsp" access="ROLE_ADMIN" />
  <sec:intercept-url pattern="/**" access="ROLE_USER" />
</sec:http>
```

这次我们会使用 `j2eePreAuthFilter` 执行用户认证，所有默认那些 `form-login`, `basic-login`, `rememberMe` 都没了用武之地。

而为了使用 `j2eePreAuthFilter`，我们需要进行如下配置：

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

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

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

<bean id="j2eeMappableRolesRetriever"
class="org.springframework.security.ui.preauth.j2ee.WebXmlMappableAttributesRetrie
ver">
  <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.J2eePreAuthenticatedProcessingF
ilter">
    <sec:custom-filter position="BASIC_PROCESSING_FILTER" />
    <property name="authenticationManager" ref="authenticationManager"/>
    <property name="authenticationDetailsSource">
        <bean
class="org.springframework.security.ui.preauth.j2ee.J2eeBasedPreAuthenticatedWebAut
henticationDetailsSource">
            <property name="mappableRolesRetriever"
ref="j2eeMappableRolesRetriever"/>
            <property name="userRoles2GrantedAuthoritiesMapper">
                <bean
class="org.springframework.security.authoritymapping.SimpleAttributes2GrantedAuthor
itiesMapper">
                    <property name="convertAttributeToUpperCase" value="true"/>
                </bean>
            </property>
        </bean>
    </property>
</bean>

<bean id="preauthEntryPoint"

class="org.springframework.security.ui.preauth.PreAuthenticatedProcessingFilterEntr
yPoint"/>

```

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：

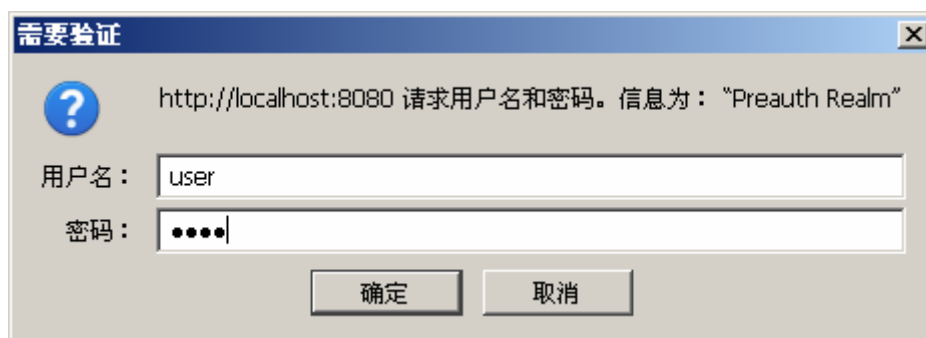


图 17.1. Realm 登录

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。

第 18 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。

18.1. 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。

18.2. 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`。

第 19 章 信道安全

19.1. 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。

19.2. 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。

第 20 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。

20.1. 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。

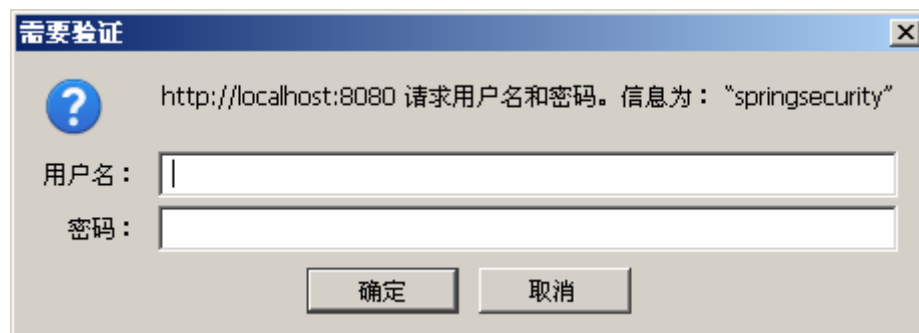


图 20.1. digest 登录

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。

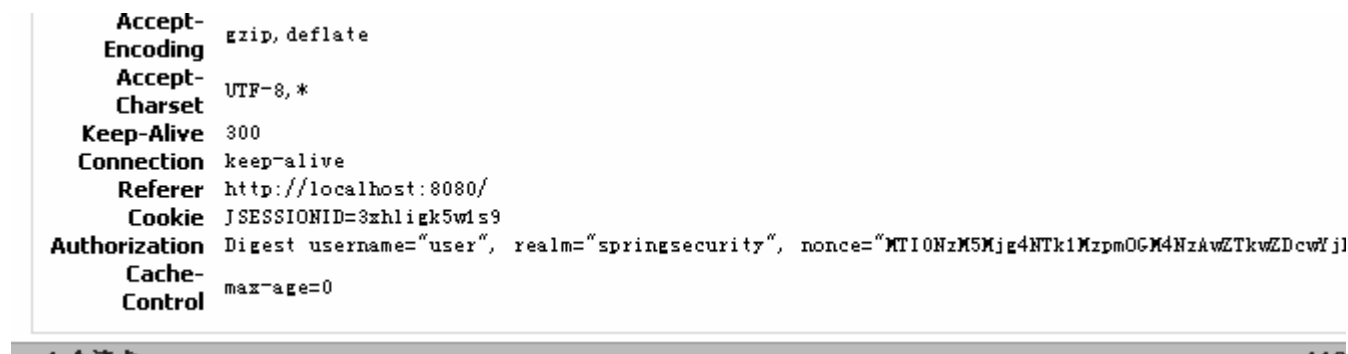


图 20.2. HTTP 请求头

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。

20.2. 使用ajax实现digest认证

可以直接使用 ajax 来进行 digest 认证，完全不需要任何额外的配置，只需要在 open 的时候传入用户名和密码就可以完成认证。

```
oRequest.open("get", "index.jsp", false, "user", "user");
oRequest.setRequestHeader("Content-type", "text/xml; charset=utf-8");
oRequest.send("");
```

这样就完成的认证过程，之后再使用普通方式访问其他页面也没问题了。

20.3. 编程实现digest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。

❷ 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。

第 21 章 通过LDAP获取用户信息

很多企业内部使用 LDAP 保存用户信息, 这章我们来看一下如何从 LDAP 中获取 Spring Security 所需的用户信息。

首先在 pom.xml 中添加 ldap 所需的依赖。

```
<dependency>
  <groupId>org.apache.directory.server</groupId>
  <artifactId>apacheds-server-jndi</artifactId>
  <version>1.0.2</version>
</dependency>
<dependency>
  <groupId>org.springframework.ldap</groupId>
  <artifactId>spring-ldap</artifactId>
  <version>1.2.1</version>
</dependency>
```

然后修改配置文件, 使用内嵌的 ldap 服务器和 ldap-authentication-provider。

```
<ldap-server ldif="classpath:users.ldif" port="33389" root="dc=family168,dc=com"/>

<ldap-authentication-provider
  group-search-filter="member={0}"
  group-search-base="ou=groups"
```



```
user-search-base="ou=people"  
user-search-filter="uid={0}"  
/>
```

这里配置内嵌的 **ldap** 服务器从 **users.ldif** 文件中读取初始化数据, 端口使用 33389, 查询目录的根目录设置为 **dc=family168,dc=com**。

ldap-authentication-provider 设置查找组和用户的配置, 分别使用 **ou=groups** 表示组, 使用 **ou=people** 表示用户。

用于保存 **ldap** 初始信息的文件内容如下:

```
dn: ou=groups,dc=family168,dc=com  
objectclass: top  
objectclass: organizationalUnit  
ou: groups  
  
dn: ou=people,dc=family168,dc=com  
objectclass: top  
objectclass: organizationalUnit  
ou: people  
  
dn: uid=user,ou=people,dc=family168,dc=com  
objectclass: top  
objectclass: person  
objectclass: organizationalPerson  
objectclass: inetOrgPerson  
cn: FirstName LastName  
sn: LastName  
uid: user  
userPassword: user  
  
dn: uid=admin,ou=people,dc=family168,dc=com  
objectclass: top  
objectclass: person  
objectclass: organizationalPerson  
objectclass: inetOrgPerson  
cn: FirstName LastName  
sn: LastName  
uid: admin  
userPassword: admin  
  
dn: cn=user,ou=groups,dc=family168,dc=com
```

```
objectclass: top
objectclass: groupOfNames
cn: ROLE_USER
member: uid=user, ou=people, dc=family168, dc=com
member: uid=admin, ou=people, dc=family168, dc=com

dn: cn=admin, ou=groups, dc=family168, dc=com
objectclass: top
objectclass: groupOfNames
cn: ROLE_ADMIN
member: uid=admin, ou=people, dc=family168, dc=com
```

这里在 `dc=family168,dc=com` 目录下创建了 `groups` 和 `people` 两个目录，然后在 `people` 目录下创建了 `user` 和 `admin` 两个用户。在 `groups` 目录下创建了 `admin` 和 `user` 两个目录，并将 `user` 和 `admin` 两个用户与 `groups` 的 `user` 目录关联，又将 `admin` 用户与 `groups` 的 `admin` 目录关联。

在系统初始化后，`Spring Security` 会在 `people` 下读取用户信息，而对应的权限信息是对应用户所关联的 `groups` 信息，`Spring Security` 会将查询到的权限信息加上 `ROLE_` 前缀，如 `cn=admin` 最终会转换为 `ROLE_ADMIN`。

实例在 `ch113`。

第 22 章 通过OpenID进行登录

`OpenID` 是一种网上身份识别服务，它的目标是让用户可以网上使用同一身份登录不同的系统。

这一章我们讲解如何使用 `Spring Security` 支持 `OpenID`。

22.1. 配置

首先在 `pom.xml` 中添加使用 `OpenID` 所需的依赖。

```
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-openid</artifactId>
  <version>2.0.5.RELEASE</version>
</dependency>
```

然后在 xml 中添加 OpenID 所需的配置。

```
<http>
  <intercept-url pattern="/admin.jsp" access="ROLE_ADMIN" />
  <intercept-url pattern="/**" access="ROLE_USER" />
  <openid-login user-service-ref="userService"/>
</http>

<user-service id="userService">
  <user name="http://family168.myopenid.com/" password="password"
  authorities="ROLE_ADMIN,ROLE_USER" />
</user-service>
```

只需要添加 openid-login 标签，并引用一个 userService。userService 中需要配置 OpenID 中的账号，以及对应的权限。它会在用户登录成功后对用户进行授权。

现在我们可以启动应用，进入登陆页面。

Login with OpenID Identity

Identity:

图 22.1. 登录页面

在登录页面中输入 http://family168.myopenid.com/，这是我们已注册过的一个 OpenID 账号，点击提交之后会跳转到 myopenid.com 网站。

myOpenID™

SIGN IN

Notice [Dismiss](#)

You must sign in to authenticate to `http://localhost:8080/` as `http://family168.myopenid.com/`

Username `http://family168.myopenid.com/`

Password [Masked Password]

☐ Stay signed in

[Sign In](#) [Cancel](#)

▪ [Sign in with an SSL certificate](#)

▪ [I cannot access my account](#)

图 22.2. myopenid.com

在此处输入密码：password。登录成功之后就会跳转回我们的系统。

username : `http://family168.myopenid.com/`

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

图 22.3. 登录成功

这时我们可以看到当前登录系统的用户是 `http://family168.openid.com/`。

22.2. 系统时间问题

因为 OpenID 服务器会根据当前时间生成 nonce 进行权限校验，所以一定要保证服务器的当前时间是正确的，如果服务器当前时间比正常时间快了哪怕只是几秒

钟，也会导致在进行校验时抛出 `nonce is too old` 异常，这会导致一直无法正常登陆。

实例在 ch114。

第 23 章 使用 X509 登录

X509 是一种基于双向加密的身份认证方式，它要基于 SSL，并要求开启客户端证书，是一种非常强力的安全手段。

23.1. 生成证书

我们要为 X509 生成服务器端和客户端使用的证书。

首先生成服务端证书 `server.jks`。

```
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
```

然后生成客户端证书，并将客户端证书导入到 `server.jks` 中。

```
keytool -genkey -v -alias user -keyalg RSA -storetype PKCS12 -keystore user.p12 -dname  
"cn=user,ou=family168,o=www.family168.com,l=china,st=beijing,c=cn" -storepass  
password -keypass password  
  
keytool -export -alias user -keystore user.p12 -storetype PKCS12 -storepass password  
-rfc -file user.cer  
  
keytool -import -v -file user.cer -keystore server.jks -storepass password
```

最后将服务端证书导出，加入 `jre` 安全证书中。

```
keytool -export -trustcacerts -alias server -file server.cer -keystore server.jks  
-storepass password  
  
keytool -import -trustcacerts -alias server -file server.cer -keystore  
"%JAVA_HOME%/JRE/LIB/SECURITY/CACERTS" -storepass changeit
```

23.2. 配置服务器使用双向加密

在 `pom.xml` 中对 `jetty` 进行配置，使用刚刚生成的服务端证书 `server.js` 配置 `SSL`，并使用 `needClientAuth="true"` 启用双向加密啊。

```
<connectors>
  <connector implementation="org.mortbay.jetty.nio.SelectChannelConnector">
    <port>8080</port>
    <maxIdleTime>3600000</maxIdleTime>
  </connector>
  <connector implementation="org.mortbay.jetty.security.SslSocketConnector">
    <port>8443</port>
    <keystore>certificates/server.jks</keystore>
    <password>password</password>
    <keyPassword>password</keyPassword>
    <truststore>certificates/server.jks</truststore>
    <trustPassword>password</trustPassword>
    <needClientAuth>true</needClientAuth>
  </connector>
</connectors>
<scanIntervalSeconds>10</scanIntervalSeconds>
<webDefaultXml>../webdefault.xml</webDefaultXml>
<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>
```

23.3. 配置X509 认证

修改配置文件，添加 `x509` 配置方式

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

```
<x509 subject-principal-regex="CN=(.*?), " user-service-ref="userService"/>
</http>

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

x509 中，subject-principal-regex 会从客户端证书中获取用户名，将 CN 部分当做 username 来使用，因为服务器端会绝对相信客户端证书中的信息，所以不会再去使用 userService 中的密码对用户进行校验，一旦证书校验通过，直接对用户进行授权。

现在系统已经配置好了 x509 认证方式，只要将生成的客户端证书 user.p12 导入到浏览器中，直接访问系统就会发现已经用 user 用户登录到系统中了。

如果使用 IE 浏览器，可以直接双击 user.p12 进行导入。

如果使用 FireFox 浏览器，需要点击浏览器工具条上的“工具”->“选项”->“高级”->“查看证书”，在“证书管理器”中导入 user.p12。

实例在 ch115。

第 24 章 使用 NTLM 登录（无法成功登陆）

NTLM 是 NT Lan Manager，即 Window NT 局域网管理器。它会以本地系统当前用户尝试访问远程主机，基本工作流程请去网上搜索。

警告

这章会演示如何在 Spring Security 中使用 NTLM，但是因为不了解如何配置 NTLM 所以实例虽然可以运行，但是一直无法登陆。

如果有谁知道如何配置 NTLM 可以在 ch116 成功进行登陆，请联系 xyz20003@gmail.com，感激不禁。

NTLM 验证允许 Windows 用户使用当前登录系统的身份进行认证，当前用户应该是登陆在一个域（domain）上，他的身份是可以自动通过浏览器传递给服务器的。它是一种单点登录的策略，系统可以通过 NTLM 重用登录到 Windows 系统中的用户凭证，不用再次要求用户输入密码进行认证。

不过，它只能用在 IE 中。使用 Firefox 时，你会被要求输入用户名和密码，这时可以使用下面的配置启用 NTLM。

- 在 Firefox 的地址栏中输入“about:config”。
- 这时可以看到 Firefox 的所有配置，现在找到“network.automatic-ntlm-auth.trusted-uris”。
- 输入主机名称，比如“host1.domain.com, host2.domain.com”，也可以直接输入“.domain.com”。

现在我们开始在 Spring Security 中配置 NTLM。

首先在 pom.xml 中添加依赖：

```
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-ntlm</artifactId>
  <version>2.0.5.RELEASE</version>
</dependency>
```

然后修改 xml 文件，添加 NTLM 的过滤器和入口点。

```
<http entry-point-ref="ntlmEntryPoint">
  <intercept-url pattern="/access_denied.jsp" filters="none"/>
  <intercept-url pattern="/admin.jsp" access="ROLE_ADMIN" />
  <intercept-url pattern="/**" access="ROLE_USER" />
</http>

<beans:bean id="userDetailsAuthenticationProvider"
class="com.family168.springsecuritybook.ch116.UserDetailsAuthenticationProvider">
  <custom-authentication-provider/>
  <beans:property name="userDetailsService" ref="userDetailsService"/>
</beans:bean>

<beans:bean id="ntlmEntryPoint"
class="org.springframework.security.ui.ntlm.NtlmProcessingFilterEntryPoint">
  <beans:property name="authenticationFailureUrl" value="/access_denied.jsp"/>
</beans:bean>

<beans:bean id="ntlmFilter"
class="org.springframework.security.ui.ntlm.NtlmProcessingFilter">
  <custom-filter position="NTLM_FILTER"/>
  <beans:property name="stripDomain" value="true"/>
</beans:bean>
```



```
<beans:property name="defaultDomain" value="domain.mediasoft.be"/>
<beans:property name="netbiosWINS" value="domain"/>
<beans:property name="authenticationManager" ref="_authenticationManager"/>
</beans:bean>
```

我们还需要创建一个 **UserDetailsAuthenticationProvider**，可以直接通过 **username** 从 **UserDetailsService** 中获得用户信息与相应的权限。

```
package com.family168.springsecuritybook.ch116;

import org.springframework.dao.DataAccessException;
import org.springframework.security.AuthenticationException;
import org.springframework.security.AuthenticationServiceException;
import org.springframework.security.providers.UsernamePasswordAuthenticationToken;
import
org.springframework.security.providers.dao.AbstractUserDetailsAuthenticationProvide
r;
import org.springframework.security.userdetails.UserDetails;
import org.springframework.security.userdetails.UserDetailsService;

public class UserDetailsAuthenticationProvider extends
AbstractUserDetailsAuthenticationProvider {

    private UserDetailsService userDetailsService;

    protected UserDetails retrieveUser(String username,
UsernamePasswordAuthenticationToken authentication)
        throws AuthenticationException {
        UserDetails loadedUser;
        try {
            loadedUser = this.getUserDetailsService().loadUserByUsername(username);
        } catch (DataAccessException repositoryProblem) {
            throw new AuthenticationServiceException(repositoryProblem.getMessage(),
repositoryProblem);
        }
        if (loadedUser == null) throw new AuthenticationServiceException("User cannot
be null");
        return loadedUser;
    }

    protected void additionalAuthenticationChecks(UserDetails userDetails,
```

```

        UsernamePasswordAuthenticationToken authentication)
        throws AuthenticationException{

    }

    public UserDetailsService getUserDetailsService() {
        return userDetailsService;
    }

    public void setUserDetailsService(UserDetailsService userDetailsService) {
        this.userDetailsService = userDetailsService;
    }
}

```

现在的问题就是如何在 NTLM 中配置了，如何配置才能让 ch116 正常登陆。

实例在 ch116。

第 25 章 使用JAAS机制

可以在 Spring Security 中使用 JAAS 机制进行用户的身份认证。

JAAS 即 Java Authentication and Authorization Service, 它是 JDK 自带的一套专门用于处理用户认证和授权的标准 API, Spring Security 中可以使用 API 作为 `AuthenticationProvider` 处理用户认证与授权。

配置文件中，我们使用 `JaasAuthenticationProvider` 作为 `AuthenticationProvider`。

```

<http>
    <intercept-url pattern="/admin.jsp" access="ROLE_ADMIN" />
    <intercept-url pattern="/**" access="ROLE_USER" />
    <form-login/>
    <logout/>
</http>

<beans:bean id="jaasAuthenticationProvider"
    class="org.springframework.security.providers.jaas.JaasAuthenticationProvider">
    <custom-authentication-provider/>
    <beans:property name="loginConfig" value="/WEB-INF/login.conf" />
    <beans:property name="loginContextName" value="JAASTest" />
    <beans:property name="callbackHandlers">
        <beans:list>

```

```

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

    </beans:list>
</beans:property>
<beans:property name="authorityGranters">
    <beans:list>
        <beans:bean
class="com.family168.springsecuritybook.ch117.AuthorityGranterImpl" />
    </beans:list>
</beans:property>
</beans:bean>

```

注意不能在 `http` 标签中使用 `auto-config="true"` 或是在 `http` 标签中包含 `rememberMe`，因为 `rememberMe` 需要引用 `userDetailsService`，而在使用 `JaasAuthenticationProvider` 时，用户数据校验是交由 `LoginModule` 处理的，不会使用 `userDetailsService`，所以 `rememberMe` 会抛出异常。

我们将 JAAS 所需的配置文件放在 `/WEB-INF/login.config`。

```

JAASTest {
    com.family168.springsecuritybook.ch117.LoginModuleImpl required;
};

```

并在配置文件中指明使用 `JAASTest` 作为登陆上下文。

```

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

```

现在要创建 `LoginModuleImpl` 用来处理用户登录。

```

package com.family168.springsecuritybook.ch117;

import java.security.Principal;

import java.util.Map;

import javax.security.auth.Subject;
import javax.security.auth.callback.Callback;

```

```

import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.NameCallback;

import javax.security.auth.callback.PasswordCallback;

import javax.security.auth.callback.TextInputCallback;
import javax.security.auth.login.LoginException;
import javax.security.auth.spi.LoginModule;

public class LoginModuleImpl implements LoginModule {

    private String password;
    private String user;
    private Subject subject;

    public boolean abort() throws LoginException {
        return true;
    }

    public boolean commit() throws LoginException {
        return true;
    }

    public void initialize(Subject subject, CallbackHandler callbackHandler, Map
sharedState, Map options) {
        this.subject = subject;

        try {
            TextInputCallback textCallback = new TextInputCallback("prompt");
            NameCallback nameCallback = new NameCallback("prompt");
            PasswordCallback passwordCallback = new PasswordCallback("prompt",
false);

            callbackHandler.handle(new Callback[] {textCallback, nameCallback,
passwordCallback});

            password = new String(passwordCallback.getPassword());
            user = nameCallback.getName();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    public boolean login() throws LoginException {
        if (!user.equals("user")) {

```

```
        throw new LoginException("Bad User");
    }
```

```
    if (!password.equals("user")) {
```

```
        throw new LoginException("Bad Password");
    }
```

```
    subject.getPrincipals().add(new Principal() {
        public String getName() {
            return "TEST_PRINCIPAL";
        }
    });
```

```
    subject.getPrincipals().add(new Principal() {
        public String getName() {
            return "NULL_PRINCIPAL";
        }
    });
```

```
    return true;
```

```
}
```

```
public boolean logout() throws LoginException {
    return true;
}
```

```
}
```

当用户登录成功时，会通过 **authorityGranters** 为权限主体授权，这一步也要自己实现 **AuthorityGranter** 接口。

```
package com.family168.springsecuritybook.ch117;
```

```
import java.security.Principal;
```

```
import java.util.HashSet;
```

```
import java.util.Set;
```

```
import org.springframework.security.providers.jaas.AuthorityGranter;
```

```
public class AuthorityGranterImpl implements AuthorityGranter {
```

```
    public Set grant(Principal principal) {
```

```
        Set rtnSet = new HashSet();
```

```

        if (principal.getName().equals("TEST_PRINCIPAL")) {
            rtnSet.add("ROLE_USER");
            rtnSet.add("ROLE_ADMIN");
        }

        return rtnSet;
    }
}

```

至此，JAAS 与 Spring Security 结合进行认证授权的功能已经完成，每一步都要件功能写死在代码里，让人感觉很不舒服。

实例在 ch117。

第 26 章 使用HttpInvoker

Spring Security 提供了 `AuthenticationSimpleHttpInvokerRequestExecutor`，可以在调用 `HttpInvoker` 时，自动根据当前权限主体生成 basic 认证所需的 http 请求头，以此来通过远程服务器的认证，从而访问 `httpInvoker` 暴露的远程资源。

只需要为 `HttpInvokerProxyFactoryBean` 配置 `httpInvokerRequestExecutor` 属性。

```

<bean id="helloServiceProxy"
class="org.springframework.remoting.httpinvoker.HttpInvokerProxyFactoryBean">
    <property name="httpInvokerRequestExecutor">
        <bean
class="org.springframework.security.context.httpinvoker.AuthenticationSimpleHttpInvokerRequestExecutor"/>
    </property>
    <property name="serviceUrl" value="http://localhost:8080/ch118/hello.service"/>
    <property name="serviceInterface"
value="com.family168.springsecuritybook.ch118.HelloService"/>
</bean>

```

实例在 ch118。

第 27 章 使用 rmi

Spring Security 提供了 `ContextPropagatingRemoteInvocationFactory`，可以在调用 rmi 时，将当前 `SecurityContext` 当做 rmi 的一部分传递到远程服务器，并使用这个 `SecurityContext` 在远程服务器中进行权限校验，这也就是 `RunAsManager` 的作用所在，当一个用户访问一个方法时，需要自动调用远程服务，这个远程服务中需要一个角色才可以通过校验，为了让调用此方法的用户都拥有调用远程资源的权限，才使用了 `RunAsManager` 为所有访问该方法的用户自动添加一个角色。

实际使用时，只需要为 `RmiProxyFactoryBean` 配置 `remoteInvocationFactory` 属性。

```
<bean id="helloServiceProxy"
class="org.springframework.remoting.rmi.RmiProxyFactoryBean">
    <property name="remoteInvocationFactory">
        <bean
class="org.springframework.security.context.rmi.ContextPropagatingRemoteInvocationF
actory"/>
    </property>
    <property name="serviceUrl" value="rmi://localhost:1199/helloService"/>
    <property name="serviceInterface"
value="com.family168.springsecuritybook.ch119.HelloService"/>
</bean>
```

实例在 ch119。

第 28 章 控制 portlet 的权限

实际是只是为 portlet 添加权限拦截器，它会为 portlet 进行 Pre-Auth 式认证，就是说它会从当前 `SecurityContext` 中取出权限实体，用来进行 portlet 的校验。

在 portlet 对应的配置文件中配置如下拦截器，用于校验对应 portlet 的权限。

```
<bean id="portletContextIntegrationInterceptor"
class="org.springframework.security.context.PortletSessionContextIntegrationInterce
ptor"/>

<bean id="portletAuthenticationInterceptor"
class="org.springframework.security.ui.portlet.PortletProcessingInterceptor">
    <property name="authenticationDetailsSource">
```

```

    <bean
class="org.springframework.security.ui.portlet.PortletPreAuthenticatedAuthenticatio
nDetailsSource">
        <property name="mappableRolesRetriever">
            <bean
class="org.springframework.security.authoritymapping.SimpleMappableAttributesRetrie
ver">
                <property name="mappableAttributes">
                    <list>

```

```

                        <value>tomcat</value>

```

```

                        <value>admin</value>
                        <value>manager</value>
                        <!-- Some standard liferay roles -->
                        <value>Administrator</value>
                        <value>Guest</value>
                        <value>User</value>
                        <value>Power User</value>
                    </list>
                </property>
            </bean>
        </property>
    </bean>
</property>
<property name="authenticationManager" ref="authenticationManager"/>
<!-- Liferay doesn't seem to set the authType -->
<property name="useAuthTypeAsCredentials" value="false"/>
</bean>

<sec:authentication-manager alias="authenticationManager"/>

<bean id="portletAuthProvider"
class="org.springframework.security.providers.preauth.PreAuthenticatedAuthenticatio
nProvider">
    <sec:custom-authentication-provider/>
    <property name="preAuthenticatedUserDetailsService">
        <bean
class="org.springframework.security.providers.preauth.PreAuthenticatedGrantedAuthor
itiesUserDetailsService"/>
    </property>
    <property name="throwExceptionWhenTokenRejected" value="true"/>
</bean>

```


实例在 ch120。

第 29 章 保存登录之前的请求

经常会碰到一种情况，用户花费大量时间编辑信息，但是 session 超时失效导致用户自动退出系统，安全过滤器会强制用户再次登录，但这也会使用户提交的信息全部丢失。

为了解决这个问题，Spring Security 提供了一种称作 SavedRequest 功能，可以在未登录用户访问资源时，将用户请求保存起来，当用户登录成功之后 SecurityContextHolderAwareRequestFilter 会使用之前保存的请求，结合当前用户的请求生成一个新的请求对象，而这个请求对象中就保存了用户登录之前提交的信息。

SavedRequest 功能默认就会被 Spring Security 启用，不需任何配置就可以重用登陆之前请求提交的数据。

如果希望尝试 SavedRequest 的功能，可以运行 ch121 中的实例。

进入系统后会显示一个表单，可以在表单中填写信息，然后进行提交。提交的目标页面是 user.jsp。



图 29.1. 未登录用户提交信息

但是因为用户尚未登录，所以请求会被转发到登陆页面，当我们登录成功后，系统会自动跳转到用户登录之前请求的页面，user.jsp。

Login with Username and Password

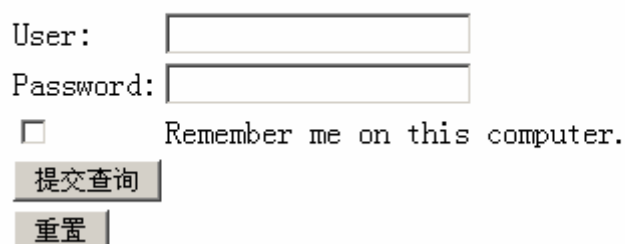


图 29.2. 进入登陆页面

因为我们使用了 `SavedRequest` 功能，所以现在就能在 `user.jsp` 看到我们登陆之前提交的信息。

user

message

图 29.3. 使用登录之前提交的信息

如果希望禁用 `SavedRequest` 这一功能，只需要在 `http` 标签中设置 `servlet-api-provision` 参数。

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

实例在 `ch121`。

部分 III. 内部机制篇

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 30 章 保护方法调用

这里有三种方式可以选择：

30.1. 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>1.6.5</version>
</dependency>
<dependency>
  <groupId>org.aspectj</groupId>
  <artifactId>aspectjweaver</artifactId>
  <version>1.6.5</version>
</dependency>

```

首先来看看我们将要保护的 java 类。

```

package com.familyl68.springsecuritybook.ch201;

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.ch201.MessageServiceImpl.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。

30.2. 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.ch201.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。

30.3. 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。

30.3.1. 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>2.0.5.RELEASE</version>
</dependency>
```

现在我们随便在 java 代码中添加注解了。

```
package com.family168.springsecuritybook.ch201;

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。

30.3.2. 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.ch201;

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。

第 31 章 权限管理的基本概念

31.1. 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。

31.2. 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");
```

31.3. 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。

第 32 章 Voter 表决者

32.1. 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` 可以通过权限要求，就可以访问。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。

默认提供的 `Voter` 继承关系如下。

`AccessDecisionVoter`

```
AbstractAclVoter
  AclEntryVoter(acls)
  BasicAclEntryVoter
```

```
LabelBasedAclVoter
```

```
RoleVoter
  RoleHierarchyVoter
AuthenticatedVoter
Jsr250Voter(tiger)
```

32.2. RoleVoter

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>
```

32.3. AuthenticatedVoter

AuthenticatedVoter 用于判断 **ConfigAttribute** 上是否拥有 **IS_AUTHENTICATED_FULLY**，**IS_AUTHENTICATED_REMEMBERED** 或 **IS_AUTHENTICATED_ANONYMOUSLY** 之类的配置。

如果配置为 **IS_AUTHENTICATED_FULLY**，那么只有 **AuthenticationTrustResolver** 的 **isAnonymous()** 和 **isRememberMe()** 都返回 **false** 时才能通过验证。

如果配置为 **IS_AUTHENTICATED_REMEMBERED**，那么会在 **AuthenticationTrustResolver** 的 **isAnonymous()** 返回 **false** 时通过验证。

如果配置为 **IS_AUTHENTICATED_ANONYMOUSLY**，就可以在 **AuthenticationTrustResolver** 的 **isAnonymous()** 和 **isRememberMe()** 两个方法返回任意值时都可以通过验证。

32.4. AbstractAclVoter

BasicAclEntryVoter, LabelBasedAclVoter, AclEntryVoter 用来在处理 ACL 中的权限控制。在 Spring Security 中的 ACL 都是基于特定 POJO 类的，每个 AclVoter 都会分配给一个特定的 POJO 类，并用来专门控制方法中对这个 POJO 类操作的权限。实际上所有 AclVoter 都是用于处理方法调用的，它会检测方法调用时传递的每个参数，当某个参数的类型与 AclVoter 对应的 POJO 类一致时，就会采用配置好的元数据进行权限校验。

第 33 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。

33.1. 权限配置数据源

处于继承树顶端的 AbstractSecurityInterceptor 有三个实现类：

- FilterSecurityInterceptor，负责处理 FilterInvocation，实现对 URL 资源的拦截。
- MethodSecurityInterceptor，负责处理 MethodInvocation，实现对方法调用的拦截。
- AspectJSecurityInterceptor，负责处理 JoinPoint，主要也是用于对方法调用的拦截。

为了限制用户访问被保护资源，Spring Security 提供了一套元数据，用于定义被保护资源的访问权限，这套元数据主要体现为 ConfigAttribute 和 ConfigAttributeDefinition。每个 ConfigAttribute 中只包含一个字符串，而一个 ConfigAttributeDefinition 中可以包含多个 ConfigAttribute。对于系统来说，每个被保护资源都将对应一个 ConfigAttributeDefinition，这个 ConfigAttributeDefinition 中包含的多个 ConfigAttribute 就是访问该资源所需的权限。

实际应用中，ConfigAttributeDefinition 会保存在 ObjectDefinitionSource 中，这是一个主要接口，FilterSecurityInterceptor 所需的

DefaultFilterInvocationDefinitionSource 和 MethodSecurityInterceptor 所需的

MethodDefinitionAttributes 都实现了这个接口。ObjectDefinitionSource 可以看做是 Spring Security 中权限配置的源头，框架内部所有的验证组件都是从 ObjectDefinitionSource 中获得数据，来对被保护资源进行权限控制的。

为了从 xml 中将用户配置的访问权限转换成 ObjectDefinitionSource 类型的对象，Spring Security 专门扩展了 Spring 中提供的 PropertyEditor 实现了 ConfigAttributeEditor，它可以把以逗号分隔的一系列字符串转换成包含多个 ConfigAttribute 的 ConfigAttributeDefinition 对象。

```
"ROLE_ADMIN, ROLE_USER"
↓
ConfigAttributeDefinition
    ConfigAttribute["ROLE_ADMIN"]
    ConfigAttribute["ROLE_USER"]
```

对于 FilterSecurityInterceptor 来说，最终生成的就是一个包含了 url pattern 和 ConfigAttributeConfiguration 的 ObjectDefinitionSource。

```
<intercept-url pattern="/admin.jsp" access="ROLE_ADMIN, ROLE_USER" />
↓
"/admin.jsp" → ConfigAttributeDefinition
                ConfigAttribute["ROLE_ADMIN"]
                ConfigAttribute["ROLE_USER"]
```

换言之，无论我们将权限配置的原始数据保存在什么地方，只要最终可以将其转换为 ObjectDefinitionSource 就可以提供给验证组件进行调用，实现权限控制。

33.2. 权限管理器

AbstractSecurityInterceptor 中将几个权限管理器组合应用，AuthenticationManager, AccessDecisionManager, AfterInvocationManager 和 RunAsManager。

AuthenticationManager 用来对用户请求进行认证工作，默认情况下，我们使用的实现类为 NamespaceAuthenticationManager。它内部将包含一个 AuthenticationProvider 队列，在实际进行权限校验的时候顺序执行这个队列实现对用户的认证功能。AuthenticationProvider 中最常见的实现是

`DaoAuthenticationProvider`，它可以根据用户的 `username` 和 `password` 对用户有效性进行验证，如果通过校验就会从 `userDetailsService` 中获取用户信息，并为用户授予对应的权限。

`AccessDecisionManager` 用于控制资源的访问权限，它下面有三个实现类可以选择 `AffirmativeBased`, `UnanimousBased` 和 `ConsensusBased`，分别对应，一票通过，一票否决，多数通过。每个 `AccessDecisionManager` 内部拥有多个 `Voter`，每个 `Voter` 会进行表决，表决的结果有 `ACCESS_GRANTED` 赞成, `ACCESS_DENIED` 反对, `ACCESS_ABSTAIN` 弃权三种。`AccessDecisionManager` 会根据最终投票的结果，结合实现类的策略判断用户是否可以访问当前资源。

33.3. 后置调用管理器

`AfterInvocationManager` 用来在方法调用完成后，根据用户的权限，对方法返回的结果进行筛选，它主要是用在 `ACL` 中的。

`AfterInvocationManager` 有两个实现类。

`BasicAclEntryAfterInvocationCollectionFilteringProvider` 用于过滤返回结果为集合的方法，`BasicAclEntryAfterInvocationProvider` 用于控制返回结果为对象的方法。

33.4. 临时分配额外权限

我们可以在某一方法调用过程中，使用 `RunAsManager` 为用户临时分配额外的权限。据说这个功能是为了在方法内部远程调用被保护资源而实现的，为实现这一功能，我们首先要在 `AbstractMethodInterceptor` 中设置 `RunAsManagerImpl`，并且要在 `ObjectDefintionSource` 中配置 `RUN_AS_`开头的权限，这样，当用户访问这个方法时，就会自动将 `SecurityContext` 中保存的 `Authenticaton` 对象替换为 `RUN_AS` 对象，并在其中附加额外的权限。

现在还不能在 2.x 版本的命名空间中调用 `RunAsManager`，3.0.0.M1 中提供了如下方法。

```
<security:global-method-security run-as-manager-ref="runAsManager">
  <protect-pointcut
    expression="execution(*
com.family168.springsecuritybook.ch12.MesageServiceImpl.admin*(..))"
    access="ROLE_ADMIN, RUN_AS_MANAGER"/>
</security:global-method-security>
```

第 34 章 用户信息

34.1. UserDetails

Spring Security 中的 UserDetails 被作为一个通用的权限主体，凡是涉及到 username 和 password 的情况，都会使用到 UserDetails 和它对应的服务。

常用的服务有从内存中读取用户信息的 InMemoryDaoImpl 和用数据库中读取用户信息的 JdbcDaoImpl。它们都实现了 UserDetailsService，因此都可以使用 loadUserByUsername() 方法获得对应用户的信息。

如果使用了 LDAP，还会接触到 LdapUserDetailsService，它用来从 LDAP 中获取用户信息。

在 org.springframework.security.userdetails 包下还包含一个 check 目录，它主要用来校验用户是否过期，是否被锁定，是否被禁用。

还可以看到一个 hierarchicalroles，它的作用是处理角色继承关系，如果希望使用角色继承策略，需要将原始的 UserDetailsService 通过 UserDetailsServiceWrapper 进行一下封装，从而获得由 UserDetailsWrapper 封装的 UserDetails，以此来实现角色继承机制。

34.2. 使用角色继承

在 Spring Security 中，我们可以指定角色间的继承关系，这样可以重用角色权限，减少配置的代码量，让权限配置整体上显得更清晰。

为了使用角色继承功能，我们需要对原有的配置文件进行一些修改。

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

<beans:bean id="userDetailsServiceWrapper"

class="org.springframework.security.userdetails.hierarchicalroles.UserDetailsServiceWrapper">
    <beans:property name="userDetailsService" ref="userDetailsService"/>
    <beans:property name="roleHierarchy">
        <beans:bean
class="org.springframework.security.userdetails.hierarchicalroles.RoleHierarchyImpl"
">
            <beans:property name="hierarchy" value="ROLE_ADMIN > ROLE_USER"/>
        </beans:bean>
    </beans:property>
</beans:bean>
```

```

    </beans:property>
</beans:bean>

<user-service id="userDetailsService">
    <user name="admin" password="admin" authorities="ROLE_ADMIN" />
    <user name="user" password="user" authorities="ROLE_USER" />
</user-service>

```

我们将原有的 `user-service` 单独抽离出来，在 `userDetailsService` 的基础上生成一个 `userDetailsServiceWrapper`，这个 `wrapper` 的作用就是在原有的 `user-service` 的基础上启用角色继承功能。

我们使用 `RoleHierarchyImpl` 为 `userDetailsServiceWrapper` 配置了角色继承的策略，`ROLE_ADMIN > ROLE_USER` 表示 `ROLE_ADMIN` 将继承 `ROLE_USER` 所有用的所有角色，只要是允许 `ROLE_USER` 访问的资源，`ROLE_ADMIN` 也都有权限进行访问。这样我们在 `user-service` 中的配置就可以从 `ROLE_ADMIN,ROLE_USER` 简化为 `ROLE_ADMIN` 了，而 `intercept-url` 中的配置也可以从 `ROLE_ADMIN,ROLE_USER` 改为 `ROLE_USER` 了。

如果希望配置更多继承关系，可以使用换行进行分隔，比如：

```

<property name="hierarchy">
    <value>
        ROLE_A > ROLE_B
        ROLE_B > ROLE_AUTHENTICATED
        ROLE_AUTHENTICATED > ROLE_UNAUTHENTICATED
    </value>
</property>

```

实例在 `ch205`。

34.3. 为ACL添加角色继承

目前，直至 `Spring Security-3.0.0.M1` 都不支持在 `acl` 中使用 `RoleHierarchy`，不过在官网的 `jira` 上有人提交了一个 `patch`，如果情况顺利的话，这个 `patch` 应该在 `Spring Security-3.0.0.M2` 中被应用到 `svn` 中，我们就可以为 `acl` 实现角色继承了。

如果希望在 `Spring Security-2.x` 中在 `acl` 部分实现角色继承，需要进行如下配置。

首先根据 `jira` 上的 `patch` 自己创建一个 `SidRoleHierarchyRetrievalStrategyImpl.java`。

```

/* Copyright 2008 Thomas Champagne
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package org.springframework.security.acs.sid;

import java.util.List;

import org.springframework.security.Authentication;
import org.springframework.security.GrantedAuthority;
import org.springframework.security.userdetails.hierarchicalroles.RoleHierarchy;
import org.springframework.util.Assert;

/**
 * Extended SidRetrievalStrategyImpl which uses a {@link RoleHierarchy} definition to
 * determine the
 * roles allocated to the current user.
 * @author Thomas Champagne
 */
public class SidRoleHierarchyRetrievalStrategyImpl extends SidRetrievalStrategyImpl {
    private RoleHierarchy roleHierarchy = null;

    public SidRoleHierarchyRetrievalStrategyImpl(RoleHierarchy roleHierarchy) {
        Assert.notNull(roleHierarchy, "RoleHierarchy must not be null");
        this.roleHierarchy = roleHierarchy;
    }

    /**
     * Calls the <tt>RoleHierarchy</tt> to obtain the complete set of user authorities.
     */
    GrantedAuthority[] extractAuthorities(Authentication authentication) {

```



```

        return
        roleHierarchy.getReachableGrantedAuthorities(authentication.getAuthorities());
    }

    public Sid[] getSids(Authentication authentication) {
        GrantedAuthority[] authorities = this.extractAuthorities(authentication);
        Sid[] sids = new Sid[authorities.length + 1];

        sids[0] = new PrincipalSid(authentication);

        for (int i = 1; i <= authorities.length; i++) {

            sids[i] = new GrantedAuthoritySid(authorities[i - 1]);
        }

        return sids;
    }
}

```

然后在 `acl` 的配置文件中配置 `bean`，并在 `AclEntryVoter`，`AclEntryAfterInvocationProvider` 和 `AclEntryAfterInvocationCollectionFilteringProvider` 中替换默认的 `SidRetrievalStrategy`。

```

<bean id="sidRetrievalStrategy"
class="org.springframework.security.acls.sid.SidRoleHierarchyRetrievalStrategyImpl"
>
    <constructor-arg ref="roleHierarchy"/>
</bean>

<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.ADMINISTRATIO
N"/>
            <util:constant
static-field="org.springframework.security.acls.domain.BasePermission.READ"/>

```

```

        </list>
    </constructor-arg>
    <property name="sidRetrievalStrategy" ref="sidRetrievalStrategy"/>
</bean>

<bean id="afterAclCollectionRead"
class="org.springframework.security.afterinvocation.AclEntryAfterInvocationCollecti
onFilteringProvider">
    <sec:custom-after-invocation-provider/>
    <constructor-arg ref="aclService"/>
    <constructor-arg>
        <list>
            <util:constant
static-field="org.springframework.security.acls.domain.BasePermission.ADMINISTRATIO
N"/>
            <util:constant
static-field="org.springframework.security.acls.domain.BasePermission.READ"/>
        </list>
    </constructor-arg>
    <property name="sidRetrievalStrategy" ref="sidRetrievalStrategy"/>
</bean>

```

这样就在 `acl` 中添加了对角色继承的支持。

34.4. PasswordEncoder和SaltValue

默认提供的 `PasswordEncoder` 包含 `plaintext`, `sha`, `sha-256`, `md5`, `md4`, `{sha}`, `{sha}`。其中 `{sha}` 和 `{sha}` 是专门为 `ldap` 准备的, `plaintext` 意味着不对密码进行加密, 如果我们不设置 `PasswordEncoder`, 默认就会使用它。

`SaltValue` 是为了让密码加密更加安全, 它有两种策略可以选择。 `user-property`, `system-wide` 分别对应着 `ReflectionSaltSource` 和 `SystemWideSaltSource`, 它们的区别是 `ReflectionSaltSource` 会使用反射, 从用户的 `Principal` 对象汇总取出一个对应的属性来作为盐值, 而 `SystemWideSaltSource` 会为所有用户都设置相同的盐值。

使用了 `PasswordEncoder` 和 `SaltValue` 的结果就是数据库中的密码变得难以辨认了, 这就要注意在添加用户时要使用相同的策略对密码进行加密, 这才能保证新用户可以正常登陆。

第 35 章 集成jcaptcha

使用 jcaptcha 实现彩色验证码，这是一个被 spring security 2 放弃的组件，据说本来维护 acegi 版本的作者不见了，所以组件就放到了 sandbox 中，最终被废弃了。

首先在 pom.xml 中添加依赖。

```
<dependency>
  <groupId>com.octo.captcha</groupId>
  <artifactId>jcaptcha</artifactId>
  <version>1.0</version>
</dependency>
<dependency>
  <groupId>commons-lang</groupId>
  <artifactId>commons-lang</artifactId>
  <version>2.4</version>
</dependency>
```

然后，将 jcaptcha 集成相关的代码复制到 src/main/java 目录下。

下一步，修改配置文件，添加 jcaptcha 的过滤器与 provider，代码太多，请参考示例中的实际配置。

最终自定义登录页面，显示效果如下所示：

图 35.1. jcaptcha

实例在 ch206。

第 36 章 动态资源管理

之前在 [第 5 章 使用数据库管理资源](#) 中，我们简要讨论过使用数据库管理资源，为了使手册开始的部分保持简洁，我们没有再深入讨论这个话题，包括实例中存在的一些问题也都没有解决，这一章中，我们会尝试进行更高层次的讨论。

36.1. 基本知识

对应的数据库结构与ER图，可以参考 [第 5 章 使用数据库管理资源](#)。

拦截器与所需的权限配置数据格式，可以参考 [第 33 章 拦截器](#)。

所有，我们需要做的就是将数据库中的数据读取出来，组装成拦截器所需的格式，然后把权限配置数据放到拦截器里。

36.2. 读取资源

为了区分 URL 和 METHOD，我们在 resc 表中使用 res_type 字段来区分这两种不同的被保护资源。

res_type="URL" 对应将在 FilterSecurityInterceptor 中使用的被保护网址。

```
INSERT INTO RESC VALUES(1,'','URL','/admin.jsp',1,'')
```

这里将 /admin.jsp 作为一个网址进行保护，随后它将被设置到 FilterSecurityInterceptor 中。

res_type="METHOD" 对应将在 MethodSecurityInterceptor 中使用的被保护方法。

```
INSERT INTO RESC  
VALUES(3,'','METHOD','com.family168.springsecuritybook.ch207.MessageService.adminMessage',3,'');
```

这里将 com.family168.springsecuritybook.ch207.Message 的 adminMessage() 方法设置为被保护资源，随后它将被设置到 MethodSecurityInterceptor 中。

我们使用如下 sql 语句从数据库中分别读取被保护的 url 和 method 信息。

读取被保护 url 信息。

```

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
 where re.res_type='URL'
order by re.priority

```

读取被保护 **method** 信息。

```

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
 where re.res_type='METHOD'
order by re.priority

```

为了实现资源的统一配置，我们创建了名为 **ResourceDetailsMonitor** 的类用来管理数据库中的被保护资源信息，它负责从数据库中读取原始信息，并转换成 **FilterSecurityInterceptor** 和 **MethodInterceptor** 所需的数据格式。

36.3. URL资源扩展点

为了动态设置 **FilterSecurityInterceptor** 中的资源配置，**ResourceDetailsMonitor** 中直接将组装后的 **FilterInvocationDefinitionSource** 使用 **setObjectDefinitionSource()** 方法设置到 **FilterSecurityInterceptor** 中。

```

FilterInvocationDefinitionSource source = resourceDetailsBuilder
    .createUrlSource(queryUrl, getUrlMatcher());
filterSecurityInterceptor.setObjectDefinitionSource(source);

```

之后，**FilterSecurityInterceptor** 就会根据我们设置的资源信息控制用户可以访问哪些资源。

36.4. METHOD资源扩展点

MethodSecurityInterceptor 的情况有些复杂，因为涉及到 spring 中 aop 的 pointcut 部分特性，所以直接为 MethodSecurityInterceptor 设置 objectDefinitionSource 是不会起作用的。

我们需要获取 delegatingMethodDefinitionSource，将数据库中读取的资源信息设置到它里面才能使 MethodSecurityInterceptor 和动态生成的 pointcut 都是用我们最新的资源信息。

```
MethodDefinitionSource source = resourceDetailsBuilder
    .createMethodSource(queryMethod);
List<MethodDefinitionSource> sources = new ArrayList<MethodDefinitionSource>();
sources.add(source);

delegatingMethodDefinitionSource.setMethodDefinitionSources(sources);
```

因为 ACL 实际上也是借助于 MethodSecurityInterceptor 来实现的，所以可以将 ACL_READ 和 AFTER_ACL_READ 配置在 res_type="METHOD"的资源中。

实例在 ch207 中。

第 37 章 扩展 UserDetails

如果希望扩展登录时加载的用户信息，最简单直接的办法就是实现 UserDetails 接口，定义一个包含所有业务数据的对象。我们下面演示如何将用户邮箱加入 UserDetails 中。

37.1. 实现 UserDetails 接口

UserDetails 接口中总共声明了六个方法：

```
public interface UserDetails extends Serializable {
    GrantedAuthority[] getAuthorities();❶
    String getPassword();❷
    String getUsername();❸
    boolean isAccountNonExpired();❹
    boolean isAccountNonLocked();❺
    boolean isCredentialsNonExpired();❻
    boolean isEnabled();❼
}
```

- ❶ 用户拥有的权限
- ❷ 用户名
- ❸ 密码
- ❹ 用户账号是否过期
- ❺ 用户账号是否被锁定
- ❻ 用户密码是否过期
- ❼ 用户是否可用

我们的任务就是实现这六个接口，同时添加一个 `getEmail()` 方法，用以获得用户的邮箱地址。

最初我们的打算是直接继承 **Spring Security** 中默认提供的实现类 `User`，但是 `User` 为了避免用户信息被外部程序篡改，被设计为只能通过构造方法来为内部数据赋值，没有提供 `setter` 方法对其中数据进行修改，因此为了之后演示的方便，我们仿照 `User` 类自行实现了一个 `BaseUserDetails` 类，在 `BaseUserDetails` 中所有属性都被定义为 `protected`，可以暴露给子类进行操作。

在 `BaseUserDetails` 的基础上，我们实现了 `UserInfo` 类，在它里面添加有关 `email` 的属性和方法。

```
package com.family168.springsecuritybook.ch208;

import org.springframework.security.GrantedAuthority;

public class UserInfo extends BaseUserDetails {

    private static final long serialVersionUID = 1L;

    private String email;

    public UserInfo(String username, String password, boolean enabled,
GrantedAuthority[] authorities)
        throws IllegalArgumentException {
        super(username, password, enabled, authorities);
    }

    public String getEmail() {
        return this.email;
    }
}
```

```
public void setEmail(String email) {  
    this.email = email;  
}  
}
```

37.2. 实现UserDetailsService接口

为了将 **UserInfo** 提供给权限系统，我们还需要实现自定义的 **UserDetailsService**，这个接口只包含一个方法：

```
public interface UserDetailsService {  
    UserDetails loadUserByUsername(String username)  
        throws UsernameNotFoundException, DataAccessException;  
}
```

实际运行中，系统会通过这个方法获得登录用户的信息。

下面我们直接实现 **UserDetailsService** 接口，在其中创建 **UserInfo** 的对象。

```
public class UserInfoService implements UserDetailsService {  
  
    private Map<String, UserInfo> userMap = null;  
  
    public UserInfoService() {  
        userMap = new HashMap<String, UserInfo>();  
        UserInfo userInfo = null;  
        userInfo = new UserInfo("user", "user", true, new GrantedAuthority[]{  
            new GrantedAuthorityImpl("ROLE_USER")  
        });  
        userInfo.setEmail("user@family168.com");  
        userMap.put("user", userInfo);  
        userInfo = new UserInfo("admin", "admin", true, new GrantedAuthority[]{  
            new GrantedAuthorityImpl("ROLE_ADMIN"),  
            new GrantedAuthorityImpl("ROLE_USER")  
        });  
        userInfo.setEmail("admin@family168.com");  
        userMap.put("admin", userInfo);  
    }  
}
```



```

    public UserDetails loadUserByUsername(String username)
        throws UsernameNotFoundException, DataAccessException {
        return userMap.get(username);
    }
}

```

37.3. 修改配置文件

将 `UserInfoService` 添加到配置文件中：

```

<authentication-provider user-service-ref="userDetailsService"/>

<beans:bean id="userDetailsService"
class="com.family168.springsecuritybook.ch208.UserInfoService"/>

```

定义 `userDetailsService` 之后，然后使用 `user-service-ref` 为 `authentication-provider` 设置对 `UserDetailsService` 的引用，这样在系统中就会从我们自定义的 `UserInfoService` 中获取用户信息了。

37.4. 测试运行

修改过配置文件后，在 `ch208` 中启动 `mvn`，还是通过登录页面进入系统，在登录成功页面中就可以看到用户对应的邮箱地址了。

```

username : user | email : user@family168.com
-----
admin.jsp logout

```

图 37.1. 显示邮箱地址信息

这时保存在 `SecurityContext` 中的 `Principal` 已经变为了 `UserInfo` 类型的对象，我们可以直接使用 `taglib` 获得启动的邮件信息。

```

email : <sec:authentication property="principal.email"/>

```

如果希望获得 `UserInfo` 对象，可以使用如下代码：

```
UserInfo userInfo = (UserInfo)
SecurityContextHolder.getContext().getAuthentication().getPrincipal();
```

实例在 ch208 中。

第 38 章 锁定用户

这里我们通过一个常见的功能需求来演示如何锁定一个用户的账号。

下面我们需要实现这样一个功能，当一个用户输错三次密码后，就锁定这个账号。

首先我们要注意到，虽然 `UserDetails` 中提供了 `isAccountNonLocked()` 方法，框架中也提供了 `LockedException`，但是默认提供的 `User` 实现类不太容易实现对用户锁定的操作，为此，我们需要借用 `???` 中定义的 `UserInfo` 来实现对用户的锁定。

为了监听用户输入错误密码的事件，我们需要自定义一个事件监听器 `LockUserListener`，它的主体代码如下所示：

```
public void onApplicationEvent(ApplicationEvent event) {

    if (event instanceof AuthenticationFailureBadCredentialsEvent) {
        AuthenticationFailureBadCredentialsEvent authEvent =
            (AuthenticationFailureBadCredentialsEvent) event;
        Authentication authentication = (Authentication) authEvent.getSource();
        String username = (String) authentication.getPrincipal();
        addCount(username);
    }

    if (event instanceof AuthenticationSuccessEvent) {
        AuthenticationSuccessEvent authEvent = (AuthenticationSuccessEvent) event;
        Authentication authentication = (Authentication) authEvent.getSource();
        UserInfo userInfo = (UserInfo) authentication.getPrincipal();
        String username = userInfo.getUsername();
        cleanCount(username);
    }

}
```

其中，`AuthenticationFailureBadCredentialsEvent` 事件表示用户输入了错误的密码，`AuthenticationSuccessEvent` 表示用户登录成功。

将 **LockUserListener** 添加到配置文件中即可监听权限校验的事件。

```
<beans:bean id="lockUserListener"
class="com.family168.springsecuritybook.ch209.LockUserListener">
    <beans:property name="servletContext" ref="servletContext"/>
    <beans:property name="userInfoService" ref="userInfoService"/>
</beans:bean>
```

在获得 **AuthenticationFailureBadCredentialsEvent** 事件，也即用户输入错密码时，我们首先要获得对应的用户名，然后将用户名对应的输入密码错误次数加一。当获得 **AuthenticationSuccessEvent** 事件时，说明用户已经成功登陆了，这时我们要把用户之前输入错误的密码次数清零，让他再次登录时还可以享有三次错误的机会。

这里为了演示方便，直接将用户输入密码错误的次数记录在 **ServletContext** 中，代码如下所示：

```
protected void addCount(String username) {
    Map<String, Integer> lockUserMap = getLockUserMap();
    Integer count = lockUserMap.get(username);
    if (count == null) {
        lockUserMap.put(username, Integer.valueOf(1));
    } else {
        int resultCount = count.intValue() + 1;
        if (resultCount > 3) {
            UserInfo userInfo = (UserInfo)
userInfoService.loadUserByUsername(username);
            userInfo.lockAccount();
        } else {
            lockUserMap.put(username, Integer.valueOf(resultCount));
        }
    }
}
```

当用户输错密码超过三次时，就会从 **UserInfoService** 中取出对应的 **UserInfo** 对象，使用 **lockAccount()** 方法将该用户账号锁定，之后用户就会在登录上看到对应的错误信息。

Your login attempt was not successful, try again.

Reason: User account is locked

Login with Username and Password

User:

Password:

☐ Remember me on this computer.

图 38.1. 用户已被锁定

之后用户再怎么尝试也无法登陆到系统中了。

实例在 ch209 中。

用户过期与密码过期的处理方法与锁定用户类似，等以后有机会再详细介绍。

第 39 章 设置过滤器链

对于不是从 acegi 升级到 spring security 的同志们，用多了命名空间这种配置方式，肯定在抱怨它的扩展性不够。现在我们就来展示一下在 acegi 中屡遭诟病的自定义过滤器链配置方式。

警告

acegi 当年就是以这种配置方式，被冠以“每使用一次 acegi，世界的某个地方就会有有一个精灵死掉”的称号，请各位慎用。

既然不再使用 http 标签，我们就需要在配置文件中手工声明一个 springSecurityFilterChain，这个 bean 会由 web.xml 中的 DelegatingFilterProxy 调用，注意 bean 的 id 必须为 springSecurityFilterChain，否则系统启动时会报错。

下面我们就在 springSecurityFilterChain 中配置多个过滤器链。

```
<bean id="springSecurityFilterChain"
      class="org.springframework.security.util.FilterChainProxy">
  <sec:filter-chain-map path-type="ant">
```

```

        <sec:filter-chain pattern="/spring_security_login"
            filters="loginPageFilter" />
        <sec:filter-chain pattern="/j_spring_security_check*"
            filters="httpSessionContextIntegrationFilter,authenticationProcessingFilter" />
        <sec:filter-chain pattern="/**"
            filters="httpSessionContextIntegrationFilter,
                exceptionTranslationFilter,
                filterInvocationInterceptor" />
    </sec:filter-chain-map>
</bean>

```

这里我们配置了三套过滤器链，`loginPageFilter` 负责处理 `/spring_security_login`，`httpSessionContextIntegrationFilter,authenticationProcessingFilter` 用来处理 `/j_spring_security_check*`，最后由其他三个过滤器处理其外的所有 URL 请求。

`loginPageFilter` 用来生成登录页面，`authenticationProcessingFilter` 用来处理用户登录请求，只是它还需要与 `httpSessionContextIntegrationFilter` 一同起作用才能完成用户登录。这三个过滤器的配置如下：

```

<sec:authentication-manager alias="authenticationManager" />

<bean id="httpSessionContextIntegrationFilter"
    class="org.springframework.security.context.HttpSessionContextIntegrationFilter"/>

<bean id="loginPageFilter"
    class="org.springframework.security.ui.webapp.DefaultLoginPageGeneratingFilter">
    <constructor-arg ref="authenticationProcessingFilter"/>
</bean>

<bean id="authenticationProcessingFilter"
    class="org.springframework.security.ui.webapp.AuthenticationProcessingFilter">
    <property name="authenticationManager" ref="authenticationManager"/>
    <property name="defaultTargetUrl" value="/" />
</bean>

```

其余的所有 URL 请求都使用 `httpSessionContextIntegrationFilter,exceptionTranslationFilter,filterInvocationInterceptor` 这三个过滤器的组合来处理，它们就是用来实际控制权限的部分。虽然这里

只要配置三个过滤器，但实际上它们还需要和其他附属功能部件一起工作才能完成安全控制的功能。

比如 `exceptionTranslationFilter` 就需要 `authenticationEntryPoint` 来控制抛出异常时响应的策略和跳转的 URL 地址。

```
<bean id="authenticationEntryPoint"
class="org.springframework.security.ui.webapp.AuthenticationProcessingFilterEntryPo
int">
    <property name="loginFormUrl" value="/spring_security_login"/>
</bean>

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

而 `filterInvocationInterceptor` 作为最核心的权限控制拦截器还需要 `authenticationManager` 和 `decisionManager` 的配合，这里我们为了不多害死其他精灵，所以就不贴配置内容了。

最后我们可以看到，使用了这种 `acegi` 中古老的配置方法能给我们控制每个组件的权力，但是整整 82 行代码的配置所实现功能，也就相当于使用了命名空间配置的 20 行代码，明显还是使用命名空间配置方式在可维护性上更有优势。

实例在 `ch210` 中。

第 40 章 自定义过滤器

常见的问题就是要在登录时多加几个参数时，默认的 `AuthenticationProcessFilter` 既不支持保存额外参数，也没有提供扩展点来实现这个功能，实际上就算是 `Spring Security-3.x` 中也因为只能配置一个 `handler`，实际扩展时还是比较麻烦。所以这个时候一般的选择就是自定义过滤器了。

我们的目标是在登录时除了填写用户名和密码之外，再添加一个 `mark` 参数。

图 40.1. 登录时附加一个 mark 参数

我们的目的是在登录时将这个参数保存到 session 中，以备后用。为此我们要扩展 `AuthenticationProcessFilter`：

```
public class LoginFilter extends AuthenticationProcessingFilter {

    public Authentication attemptAuthentication(HttpServletRequest request) throws
    AuthenticationException {
        Authentication authentication = super.attemptAuthentication(request);

        String mark = request.getParameter("mark");
        request.getSession().setAttribute("mark", "mark");

        return authentication;
    }
}
```

实际上我们只需要重写 `attemptAuthentication()` 这个方法，先调用 `super.attemptAuthentication()` 获得生成的 `Authentication`，如果这部分没有抛出异常，我们下面再去从 `request` 中获得 `mark` 参数，再把这个参数保存到 session 里。最后返回 `authentication` 对象即可。

下面修改配置文件，在 xml 中添加一个名为 `loginFilter` 的 bean，使用 `custom-filter` 将它加入到过滤器链中，放到原来的 `form-login` 的前面。

```
<beans:bean id="loginFilter"
class="com.family168.springsecuritybook.ch211.LoginFilter">
    <custom-filter before="AUTHENTICATION_PROCESSING_FILTER" />
    <beans:property name="authenticationManager" ref="_authenticationManager"/>
    <beans:property name="defaultTargetUrl" value="/" />
</beans:bean>
```

这样我们自定义的 `LoginFilter` 就会取代原本的 `AuthenticationProcessFilter` 处理用户登录，并在用户登录成功时将额外的 `mark` 参数保存到 `session` 中。

之后在 `jsp` 中，我们就可以直接通过 `${sessionScope['mark']}` 来获得 `mark` 的参数值。

```
username : user, mark: mark
```

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

图 40.2. 显示额外的参数

实例在 `ch211` 中。

第 41 章 使用用户组

`Spring Security` 提供了用户组机制，可以将多个用户归纳在一个组中，进行统一授权。下面我们来研究一下如何在数据库中使用用户组保存用户的权限信息。

41.1. 数据库结构

在原数据库的基础上添加用户组所需的三张表：

```
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)
);
```

ER 图如下所示:

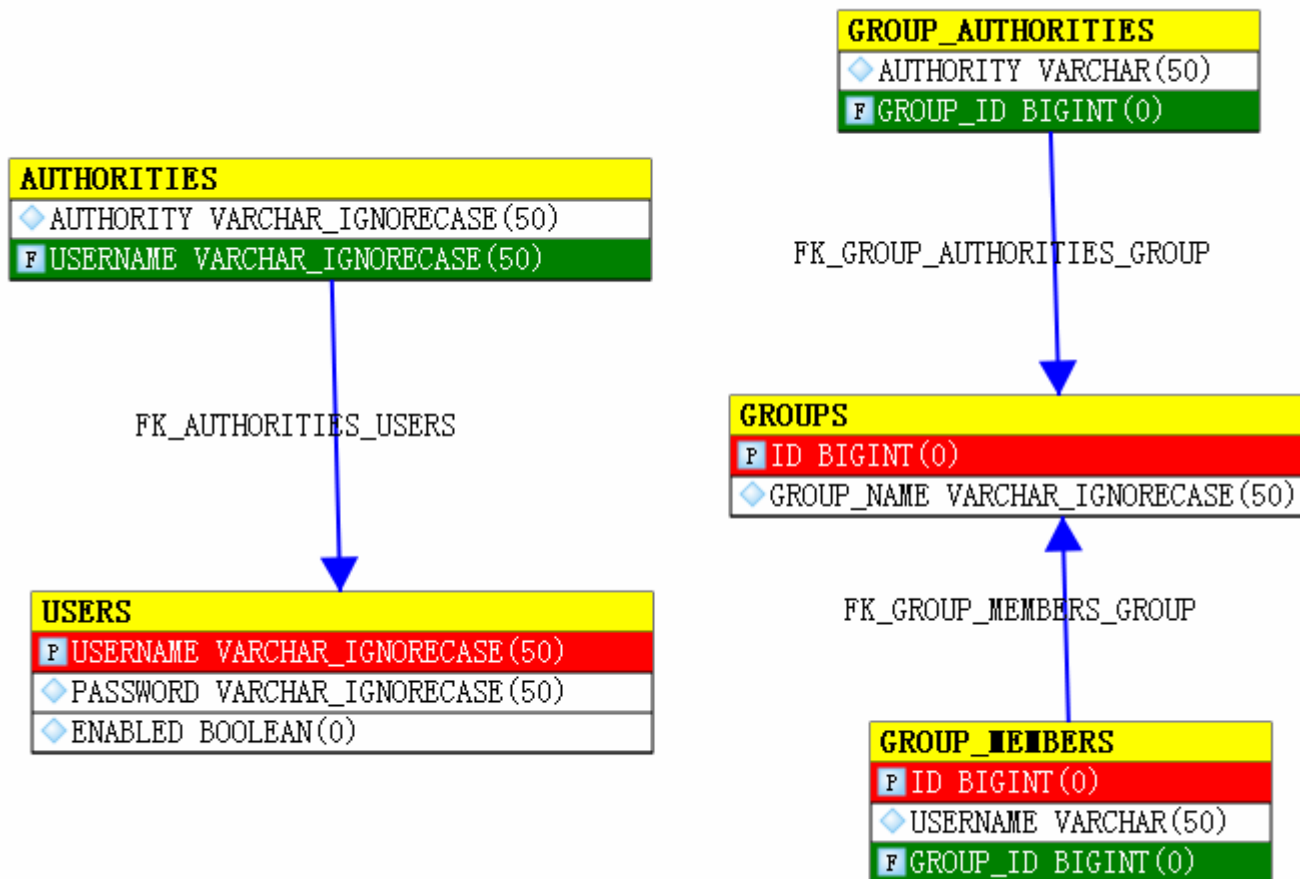


图 41.1. 用户组 ER 图

下面开始初始化数据:

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

insert into groups(id,group_name) values(1,' admin');❷
insert into groups(id,group_name) values(2,' user');

insert into group_authorities(group_id,authority) values(1,' ROLE_ADMIN');❸
```

```
insert into group_authorities(group_id,authority) values(2,'ROLE_USER');

insert into group_members(id,username,group_id) values(1,'admin',1);❷
insert into group_members(id,username,group_id) values(2,'admin',2);
insert into group_members(id,username,group_id) values(3,'user',2);
```

- ❶ 创建两个用户 admin 和 user。
- ❷ 创建两个组 admin 和 user。
- ❸ 为用户组授权，让 admin 组中的所有用户都拥有 ROLE_ADMIN 权限，user 组中的所有用户都拥有 ROLE_USER 权限。
- ❹ admin 用户加入 admin 和 user 两个用户组，将 user 用户加入 user 用户组。

41.2. 修改配置文件

为 jdbc-user-service 添加一个参数就可以打开用户组功能。

```
<authentication-provider>
  <jdbc-user-service data-source-ref="dataSource"
    group-authorities-by-username-query="
      SELECT g.id, g.group_name, ga.authority
      FROM groups g, group_members gm, group_authorities ga
      WHERE gm.username = ?
      AND g.id = ga.group_id
      AND g.id = gm.group_id"/>
</authentication-provider>
```

这样系统就会使用这条 sql 语句从用户组表中查询用户拥有的权限。

之后可以启动实例 ch212，使用 admin 和 user 用户测试授权情况。

实例在 ch212 中。

第 42 章 在JSF中使用Spring Security

在网上看到一个同志说不知道如何在 JSF 中使用 Spring Security，这里特别做了一个例子演示一下，预先声明一下咱们对 JSF 并不太了解，例子略显简单，将就用吧。

42.1. 修改过滤器支持forward

第一步就是修改 web.xml 中的过滤器配置，这样才能支持 forward，否则默认只支持 request 方式的请求。

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

42.2. 自定义登录页面

使用 JSF 写一个登陆页面。

```
<f:view>
  <h:messages />
  <h:form
    id="loginForm"
    prependId="false">
    <label for="j_username"><h:outputText value="Username:" /><br />
    </label>
    <h:inputText
      id="j_username"
      required="true">
    </h:inputText>

    <br />
    <br />
    <label for="j_password"><h:outputText value="Password:" /><br />
    </label>

    <h:inputSecret
      id="j_password"
      required="true">
    </h:inputSecret>

    <br />
    <br />
    <label for="_spring_security_remember_me"> <h:outputText
      value="Remember me" /> </label>
```

```

        <h:selectBooleanCheckbox id="_spring_security_remember_me" />
        <br />

        <h:commandButton
            type="submit"
            id="login"
            action="#{loginBean.doLogin}"
            value="Login" />

    </h:form>
</f:view>

```

然后创建对应的 `loginBean`。

```

public String doLogin() throws IOException, ServletException {
    ExternalContext context = FacesContext.getCurrentInstance()
        .getExternalContext();

    RequestDispatcher dispatcher = ((ServletRequest) context.getRequest())
        .getRequestDispatcher("/j_spring_security_check");

    dispatcher.forward((ServletRequest) context.getRequest(),
        (ServletResponse) context.getResponse());

    FacesContext.getCurrentInstance().responseComplete();

    // It's OK to return null here because Faces is just going to exit.
    return null;
}

```

它主要负责在进行 `doLogin` 时，将请求转发到 `/j_spring_security` 进行用户登录认证。

42.3. 显示密码错误信息

为了在自定义页面回显密码错误的信息，需要定义一个监听器。

```

package com.family168.springsecuritybook.ch213;

import javax.faces.application.FacesMessage;

```

```

import javax.faces.context.FacesContext;
import javax.faces.event.PhaseEvent;
import javax.faces.event.PhaseId;
import javax.faces.event.PhaseListener;

import org.springframework.security.BadCredentialsException;
import org.springframework.security.ui.AbstractProcessingFilter;

public class LoginErrorPhaseListener implements PhaseListener {
    private static final long serialVersionUID = -1216620620302322995L;

    public void beforePhase(final PhaseEvent arg0) {
        Exception e = (Exception) FacesContext.getCurrentInstance()
            .getExternalContext()
            .getSessionMap()
            .get(AbstractProcessingFilter.SPRING_
SECURITY_LAST_EXCEPTION_KEY);

        if (e instanceof BadCredentialsException) {
            FacesContext.getCurrentInstance().getExternalContext()
                .getSessionMap()
                .put(AbstractProcessingFilter.SPRING_SECURITY_LAST_EXCEPTIO
N_KEY,
                    null);
            FacesContext.getCurrentInstance()
                .addMessage(null,
                    new FacesMessage(FacesMessage.SEVERITY_ERROR,
                        "Username or password not valid.", null));
        }
    }

    public void afterPhase(final PhaseEvent arg0) {
    }

    public PhaseId getPhaseId() {
        return PhaseId.RENDER_RESPONSE;
    }
}

```

在解析之前判断是否存在 **BadCredentialsException** 异常，如果存在，就添加一条 message，这条 message 会显示在登录页面上。

在 faces-config.xml 添加这个监听器即可生效。

```
<?xml version="1.0"?>

<faces-config
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd"
  version="1.2"
>

  <lifecycle>

<phase-listener>com.family168.springsecuritybook.ch213.LoginErrorPhaseListener</pha
se-listener>
    </lifecycle>
</faces-config>
```

实例在 ch213 中。

第 43 章 自定义会话管理

43.1. 默认策略的缺陷

默认提供的 concurrent-session-control 只支持两种策略：

- 一是允许后登陆的用户将之前登陆的用户踢出系统，先登录的用户会看到“会话已失效”的提示。

这种策略实际上没办法阻止其他人在我登陆到系统之后，再使用我的账号登陆系统。最后会形成双方争先恐后的进行登录，不断将对方踢出系统的情况。

- 另一个是禁止用户使用已登录到系统的账户进行登录，后登陆的用户会在登录时看到“会话数量超过允许范围”的提示。

这种策略会导致一个比较麻烦的问题，如果用户不慎关闭了浏览器，没有通过 logout 退出系统，那么必须等到系统中用户的会话失效之后才能再次登录。如果用户不留神关了浏览器，就要再等半个小时才能再登录系统，这显然是没有道理的。

为了保证用户使用系统时不会受到其他人的影响，同时也保证不会同时有多个人使用同一个账号登陆系统，我们系统默认提供的第二种策略上进行一些修改。

简单来说，就是当用户已经登陆到系统时，如果又有人使用同一账号尝试登录系统，系统会判断当前用户的 ip 地址，如果 ip 地址与上次登录的 ip 地址相同，则注销上次登录的会话，允许当前用户登录系统。如果 ip 地址与上次登录的 ip 地址不同，而抛出异常，禁止用户登录系统。

这样既保证了同一时间只能有一个用户登录系统，又可以在用户操作失误关闭浏览器后可以再次登录系统。

43.2. 记录用户名与ip

默认情况下，系统使用 `SessionRegistry` 中只保存登录的用户名，为了比对 ip 地址，我们需要在登录时将用户的远程 ip 也保存到 `SessionRegistry` 中。

为此我们创建了 `SmartPrincipal` 类，它包含 `username` 和 `ip` 两个字段，并定义了 `equals()` 和 `hashCode()`，这样可以保证它在 `HashMap` 中的操作不会出现问题。

```
package com.family168.springsecuritybook.ch214;

import org.springframework.security.Authentication;
import org.springframework.security.ui.WebAuthenticationDetails;
import org.springframework.security.userdetails.UserDetails;

import org.springframework.util.Assert;

public class SmartPrincipal {
    private String username;
    private String ip;

    public SmartPrincipal(String username, String ip) {
        Assert.notNull(username,
            "username cannot be null (violation of interface contract)");
        Assert.notNull(ip,
            "username cannot be null (violation of interface contract)");
        this.username = username;
        this.ip = ip;
    }

    public boolean equalsIp(SmartPrincipal smartPrincipal) {
        return this.ip.equals(smartPrincipal.ip);
    }
}
```

```

    }

    @Override
    public boolean equals(Object obj) {
        if (obj instanceof SmartPrincipal) {
            SmartPrincipal smartPrincipal = (SmartPrincipal) obj;

            return username.equals(smartPrincipal.username);
        }

        return false;
    }

    @Override
    public int hashCode() {
        return username.hashCode();
    }
}

```

43.3. 改造控制类

下一步要改造 `ConcurrentSessionControllerImpl` 和 `SessionRegistryImpl`。

改造 `SessionRegistryImpl` 的原因比较复杂，因为除了 `ProviderManager` 会通过 `ConcurrentSessionController` 调用 `SessionRegistry` 来注册登录用户之外，`AuthenticationProcessFilter` 和 `SessionFixationProtectionFilter` 也会在会话伪造时直接调用 `SessionRegistry`。

默认情况下，这些类都会把用户名直接注册到 `SessionRegistry` 中，因为我们现在需要获得用户 ip，就需要把 `SmartPrincipal` 保存到 `SessionRegistry` 中，这里需要的就是做一步转换工作。

```

public class SmartSessionRegistry extends SessionRegistryImpl {
    public synchronized void registerNewSession(String sessionId,
        Object principal) {
        //
        // convert for SmartPrincipal
        //
        if (!(principal instanceof SmartPrincipal)) {
            principal = new SmartPrincipal(SecurityContextHolder.getContext()
                .getAuthentication(
    ));

```



```

    }

    super.registerNewSession(sessionId, principal);
}
}

```

对于 `ConcurrentSessionController` 来说，就是为了实现在登录时判断用户的 ip 是否与之前登陆是保存的一样，本打算继承 `ConcurrentSessionControllerImpl`，但是因为其中的 `exceptionIfMaximumExceeded` 属性未暴露给子类，所以只好重写了一个类，其中主要修改了 `allowableSessionsExceeded()`, `checkAuthenticationAllowed()`, `registerSuccessfulAuthentication()` 方法，具体代码请参考实例中的 `SmartConcurrentSessionController`。

43.4. 修改配置文件

因为 `concurrent-session-controller` 标签不支持自定义的 `ConcurrentSessionController`，我们只好使用其他办法将我们自定义的组件放入 `Spring Security` 中。

```

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

<beans:bean id="concurrentSessionFilter"
    class="org.springframework.security.concurrent.ConcurrentSessionFilter">
    <custom-filter position="CONCURRENT_SESSION_FILTER" />
    <beans:property name="sessionRegistry" ref="sessionRegistry"/>
</beans:bean>

<beans:bean id="sessionRegistry"
    class="com.family168.springsecuritybook.ch214.SmartSessionRegistry"/>

<beans:bean id="currentController"
    class="com.family168.springsecuritybook.ch214.SmartConcurrentSessionController">
    <beans:property name="sessionRegistry" ref="sessionRegistry"/>
    <beans:property name="exceptionIfMaximumExceeded" value="true"/>
</beans:bean>

```

我们在 `authentication-manager` 中使用 `session-controller-ref` 将自定义的 `SmartConcurrentController` 引入命名空间中，它会自动将 `SmartSessionRegistry` 交给 `AuthenticationProcessFilter` 与 `SessionFixationProtectionFilter` 使用。

完成这些工作之后，我们需要找两台电脑来测试上述的策略是否可以正常运行。当有用户登录之后，其他人是无法在另外的电脑上使用同一账号登陆系统的，但是如果是当前登录的用户关闭了浏览器，这时再次登录系统应该是被允许的。

实例在 `ch214` 中。

第 44 章 匹配 URL 地址

恐怕很多同志都有一个误解，就是 `Spring Security` 中配置 URL 的时候，要么配置成一个特定的 URL，比如 `/admin/index.jsp`，要么配置为某个路径下所有的 URL，比如 `/admin/**`。如果你真的这么认为的话，那就太小看 `Spring Security` 了，`/**` 只是默认提供的 `AntUrlPathMatcher` 所支持功能的一部分而已，下面我们就来破除迷信，深入讨论一下匹配 URL 地址这方面的功能。

44.1. AntUrlPathMatcher

我们默认使用的 URL 匹配器就是这个 `AntUrlPathMatcher`，它来自于 <http://ant.apache.org/> 项目，是一种简单易懂的路径匹配策略。

它为我们提供了三种通配符。

- 通配符：?

示例： `/admin/g?t.jsp`

匹配任意一个字符，`/admin/g?t.jsp` 可以匹配 `/admin/get.jsp` 和 `/admin/got.jsp` 或是 `/admin/gxt.do`。不能匹配 `/admin/xxx.jsp`。

- 通配符：*

示例： `/admin/g?t.jsp`

匹配任意多个字符，但不能跨越目录。`/*/index.jsp` 可以匹配 `/admin/index.jsp` 和 `/user/index.jsp`，但是不能匹配 `/index.jsp` 和 `/user/test/index.jsp`。

- 通配符：**

示例： `**/index.jsp`

可以匹配任意多个字符，可以跨越目录，可以匹配/index.jsp，/admin/index.jsp，/user/admin/index.jsp 和/a/b/c/d/index.jsp

44.2. RegexUrlPathMatcher

如果默认的 AntUrlPathMatcher 无法满足需求，还可以选择使用更强大的 **RegexUrlPathMatcher**，它支持使用正则表达式对 URL 地址进行匹配。

如果希望使用 **RegexUrlPathMatcher**，就需要在配置文件中添加如下配置：

```
<http auto-config='true' path-type="regex">
  <intercept-url pattern="^(/.*)*admin\.jsp$" access="ROLE_ADMIN" />
  <intercept-url pattern="^(/.*)*$" access="ROLE_USER" />
</http>
```

上面例子中就使用了正则表达式进行 URL 匹配，`^(/.*)*`与之前使用的`/**`意义相同，可以匹配任意一个请求。`^(/.*)*admin\.jsp$`表示任意一个目录下的 `admin.jsp` 请求。

有关正则表达式的更多信息请去网上找一下吧，与它相关的资源太多了。

实例在 ch215。

44.3. lowercase-comparisons

与 URL 匹配相关的配置还有一个 **lowercase-comparisons**，它的功能是在进行 URL 匹配前是否需要自动将 URL 中的字符都转换成小写。

如果我们没有设置 **lowercase-comparisons** 的值，那么在默认情况下 **AntUrlPathMatcher** 会在每次匹配 URL 之前都将 URL 中的字符转换为小写，而 **RegexUrlPathMatcher** 默认不会将 URL 中的字符转换为小写。也就是说默认情况下 **AntUrlPathMatcher** 的默认值是 `lowercase-comparisons="true"`，而 **RegexUrlPathMatcher** 的默认值是 `lowercase-comparisons="false"`。

如果需要修改 **lowercase-comparisons** 参数的值，可以像下面这样修改配置。

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

第 45 章 配置过滤器

45.1. 标准过滤器

下面是命名空间所支持的所有过滤器名称，类型，位置，以及在命名空间中对应的配置。

表 45.1. 标准过滤器别名和顺序

别名	过滤器类	命名空间元素或属性
CHANNEL_FILTER	ChannelProcessingFilter	http/intercept-url
CONCURRENT_SESSION_FILTER	ConcurrentSessionFilter	http/concurrent-session-control
SESSION_CONTEXT_INTEGRATION_FILTER	HttpSessionContextIntegrationFilter	http
LOGOUT_FILTER	LogoutFilter	http/logout
X509_FILTER	X509PreAuthenticatedProcessingFilter	http/x509
PRE_AUTH_FILTER	AstractPreAuthenticatedProcessingFilter Subclasses	N/A
CAS_PROCESSING_FILTER	CasProcessingFilter	N/A
AUTHENTICATION_PROCESSING_FILTER	AuthenticationProcessingFilter	http/form-login
OPENID_PROCESSING_FILTER	OpenIDAuthenticationProcessingFilter	N/A
BASIC_PROCESSING_FILTER	BasicProcessingFilter	http/http-basic
SERVLET_API_SUPPORT_FILTER	SecurityContextHolderAwareRequestFilter	http/@servlet-api-provision
REMEMBER_ME_FILTER	RememberMeProcessingFilter	http/remember-me
ANONYMOUS_FILTER	AnonymousProcessingFilter	http/anonymous
EXCEPTION_TRANSLATION_FILTER	ExceptionTranslationFilter	http

别名	过滤器类	命名空间元素或属性
NTLM_FILTER	NtlmProcessingFilter	N/A
FILTER_SECURITY_INTERCEPTOR	FilterSecurityInterceptor	http
SWITCH_USER_FILTER	SwitchUserProcessingFilter	N/A

上面的表格中列出了命名空间中支持的所有标准过滤器以及它们对应的顺序，这些过滤器的顺序是至关重要的，只有按照正确的顺序进行配置才能让这些过滤器正常发挥作用，在 ACEGI 的年代有太多太多的问题都是由于过滤器摆放的位置不正确造成的。幸好我们现在可以使用命名空间，它会帮我们将使用到的过滤器按照正确的方式摆放在一起。

45.2. 在http中启用标准过滤器

如果我们什么也不配置，只使用<http>标签的话，也会有五个默认的过滤器被创建出来，并放到过滤器链中。它们是：

```
HttpSessionContextIntegrationFilter
SecurityContextHolderAwareRequestFilter
ExceptionTranslationFilter
SessionFixationProtectionFilter
FilterSecurityInterceptor
```

如果追求最小的过滤器链，可以为<http>标签加上 `servlet-api-provision="false"`。

```
<http entry-point-ref="authenticationEntryPoint"
    servlet-api-provision="false"
    session-fixation-protection="none">
```

这样就会禁用 `SecurityContextHolderAwareRequestFilter` 和 `SessionFixationProtectionFilter` 两个过滤器，只剩下三个过滤器会被自动创建了。

这三个过滤器在系统中拥有十分特殊的含义，它们三个是命名空间限定的最核心过滤器，如果我们使用了命名空间的配置方式，那么这三个过滤器就是不可替代的，如果使用了 `<custom-filter position="FILTER_SECURITY_INTERCEPTOR" />` 妄图替换其中任何一个过滤器，都会抛出异常，从而导致系统无法启动。

如果在<http>中设置了 auto-config="true"就会默认启用多个常用的过滤器，实际上<http auto-config="true">就相当于如下配置：

```
<http>
  <logout/>
  <form-login/>
  <http-basic/>
  <remember-me/>
  <anonymous/>
</http>
```

使用auto-config="true"就等于在<http>中添加了五个标签，同时会启用“用户注销”，“基于表单认证”，“http basic认证”，“记忆登录信息”，“匿名认证”五个功能。这五个功能的实际效果可以参考 [第9章 图解过滤器](#)。

45.3. 为自定义过滤器设置位置

如果我们要自定义一个过滤器，就需要使用 custom-filter 标签，将过滤器加入过滤器链中才能实际起作用。在 custom-filter 中可以使用以下设置好的位置：

```
"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"
```

在 custom-filter 中可以使用 before|position|after 三种方式，将自定义过滤器放在对应名称的位置上，或者位置之前，或者位置之后。除了表示最前面的 **FIRST** 和表示最后面的 **LAST** 之外，这里的每个名称都对应着一个标准过滤器，我们可以在上面的章节中找到其对应的位置，要记住 **SESSION_CONTEXT_INTEGRATION_FILTER**, **EXCEPTION_TRANSLATION_FILTER** 和 **FILTER_SECURITY_INTERCEPTOR** 三个过滤器是不可替换的，不能对它们使用 position。

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，但是配置比较，namespace 方式只对 afterInvocation 做了一些支持，而且暂时没有找到虎牙子很好的解决方案。

第 46 章 ACL基本操作

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。

46.1. 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。

46.1.1. 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"/>
</bean>
```

```

    <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"
/>

```

46.1.2. 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。

46.1.3. 配置

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>
```

46.2. 使用管理信息

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。

46.3. 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，但是我们不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.ADMINISTRATIO
N"/>
            <util:constant
static-field="org.springframework.security.acls.domain.BasePermission.DELETE"/>
        </list>
    </constructor-arg>
    <property name="processDomainObjectClass"
value="com.family168.springsecuritybook.ch301.Message"/>
</bean>
```

```
</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(Message message) {
    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。

46.4. 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.acs.domain.BasePermission.ADMINISTRATIO  
N"/>  
      <util:constant  
static-field="org.springframework.security.acs.domain.BasePermission.READ"/>  
    </list>  
  </constructor-arg>  
</bean>  
  
<bean id="afterAclCollectionRead"  
class="org.springframework.security.afterinvocation.AclEntryAfterInvocationCollecti  
onFilteringProvider">  
  <sec:custom-after-invocation-provider/>  
  <constructor-arg ref="aclService"/>  
  <constructor-arg>  
    <list>
```

```

        <util:constant
static-field="org.springframework.security.acs.domain.BasePermission.ADMINISTRATIO
N"/>

        <util:constant
static-field="org.springframework.security.acs.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**，这里用到了数据库，方法拦截器，注解，`taglib`，基本可以说 **ACL** 就是囊括了我们之前所有讨论过功能的集合体，下一步我们会继续研究 **ACL** 的一些高级特性。

^[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。

第 47 章 管理acl

这章介绍一些有关管理 acl 的内容，包括管理多个 domain 类和动态授权与收回授权。

47.1. 管理多个domain类

上一章中我们演示了如何使用自定义 Voter 对单个 domain 类进行权限控制，如果我们需要对多个 domain 类实现 acl 权限控制时，不必给每一个类都配置一个 Voter，只需要定义一个接口，让所有需要进行 acl 权限控制的 domain 类实现这个接口，然后在 Voter 中配置这个统一接口就可以了。

示例中我们定义了一个 AclDomainClass 接口。

```
package com.family168.springsecuritybook.ch302;

public interface AclDomainClass {
}
```

然后让其他 domain 都实现这个接口。

```
public class Account implements Serializable, AclDomainClass {
    // ....
}
```

这样就可以在配置文件中配置统一的 Voter。

```
<bean id="aclDeleteVoter" class="org.springframework.security.vote.AclEntryVoter">
    <constructor-arg ref="aclService"/>
    <constructor-arg value="ACL_DELETE"/>
    <constructor-arg>
        <list>
            <util:constant
static-field="org.springframework.security.acs.domain.BasePermission.ADMINISTRATIO
N"/>
            <util:constant
static-field="org.springframework.security.acs.domain.BasePermission.DELETE"/>
        </list>
    </constructor-arg>
```

```
<property name="processDomainObjectClass"
value="com.family168.springsecuritybook.ch302.AclDomainClass"/>
</bean>
```

这样，`aclDeleteVoter` 就可以处理所有实现了 `AclDomainClass` 接口的类，而在数据库中依然会保存实现类的具体类型，不会因为在 `Voter` 中使用了同一个接口类而造成影响。

47.2. 动态授权与收回授权

现在 `acl` 的好处是可以通过 `aclService` 自由管理 `acl` 和 `ace` 的信息，比如我们可以为其他人授权，让其他人可以查看自己的消息，也可以收回这些授权。

47.2.1. 获得对象的acl权限

可以通过 `domain` 类的类型和 `id`，获得对应 `acl` 中的所有 `ace` 信息。

```
public void list(HttpServletRequest request,
    HttpServletResponse response) throws Exception {
    Long id = Long.valueOf(request.getParameter("id"));
    String clz = request.getParameter("clz");
    Acl acl = null;
    if (clz.equals("account")) {
        Account account = new Account();
        account.setId(id);
        acl = getAclService().readAclById(new ObjectIdentityImpl(account));
    } else if (clz.equals("contract")) {
        Contract contract = new Contract();
        contract.setId(id);
        acl = getAclService().readAclById(new ObjectIdentityImpl(contract));
    } else if (clz.equals("message")) {
        Message message = new Message();
        message.setId(id);
        acl = getAclService().readAclById(new ObjectIdentityImpl(message));
    }
    request.setAttribute("acl", acl);
    request.getRequestDispatcher("/permission-list.jsp").forward(request,
response);
}
```

在 jsp 中使用如下方式显示 ace 中的权限实体以及对应的权限。

```
<c:forEach var="item" items="${acl.entries}">
  <tr>
    <td>${item.sid.principal} | ${item.permission.mask}&nbsp;</td>
    <td><a
href="permission.do?action=remove&clz=${param.clz}&id=${param.id}&sid=${item.sid.principal}&permission=${item.permission.mask}">delete</a>&nbsp;</td>
  </tr>
</c:forEach>
```

ace 列表如下图所示：

[Add Permission](#)username: user | [logout](#)

ace	delete
user 16	delete

图 47.1. 实现 domain 对象对应的所有 ace 信息

user 表示权限实体的名称，16 表示 administration 权限。这里表示当前的对应已经赋予了 user 用户管理权限。

47.2.2. 添加授权

可以将一个 domain 对象的 acl 权限授予其他人。

可以选择授权的人或角色，然后选择授权的具体权限。

[Back](#)username: user | [logout](#)

Permission Info

Recipient:

Permission:

图 47.2. 添加授权

提交后可以看到 **domain** 对象对应的 **acl** 权限增加了 **admin** 用户的 **read** 权限，这时 **admin** 用户就可以查看这个 **domain** 对象的信息了。

对应代码如下所示：

```
@Transactional
public void addPermission(long id, String clz, String recipient, int mask) {
    PrincipalSid sid = new PrincipalSid(recipient);
    Permission permission = BasePermission.buildFromMask(mask);

    ObjectIdentity oid = null;
    if (clz.equals("account")) {
        oid = new ObjectIdentityImpl(Account.class, id);
    } else if (clz.equals("contract")) {
        oid = new ObjectIdentityImpl(Contract.class, id);
    } else if (clz.equals("message")) {
        oid = new ObjectIdentityImpl(Message.class, id);
    }

    MutableAcl acl;
    try {
        acl = (MutableAcl) mutableAclService.readAclById(oid);
    } catch (NotFoundException nfe) {
        acl = mutableAclService.createAcl(oid);
    }

    acl.insertAce(acl.getEntries().length, permission, sid, true);
    mutableAclService.updateAcl(acl);
}
```

47.2.3. 收回授权

可以将已经授权的权限删除。

对应代码如下：

```
@Transactional
public void deletePermission(long id, String clz, String recipient, int mask) {
```

```

PrincipalSid sid = new PrincipalSid(recipient);
Permission permission = BasePermission.buildFromMask(mask);

ObjectIdentity oid = null;
if (clz.equals("account")) {
    oid = new ObjectIdentityImpl(Account.class, id);
} else if (clz.equals("contract")) {
    oid = new ObjectIdentityImpl(Contract.class, id);
} else if (clz.equals("message")) {
    oid = new ObjectIdentityImpl(Message.class, id);
}

MutableAcl acl = (MutableAcl) mutableAclService.readAclById(oid);
AccessControlEntry[] entries = acl.getEntries();
for (int i = 0; i < entries.length; i++) {
    if (entries[i].getSid().equals(sid) &&
entries[i].getPermission().equals(permission)) {
        acl.deleteAce(i);
    }
}

mutableAclService.updateAcl(acl);
}

```

介于权限主体和权限可能有多种组合，所以我们只删除两者都完全匹配的 `ace` 信息，操作成功后可以看到用户的 `acl` 权限已经被收回了。

实例在 `ch302`。

第 48 章 acl 自动提醒

自定义 `AfterInvocationProvider`，在创建实体类时自动创建对应的 `acl` 权限，在删除实体类的时候，自动删除 `acl` 权限。

48.1. 自动创建acl

首先一个目标是监听实体类的创建，在创建后生成对应的 `acl` 信息。

根据实体类创建 `acl` 信息的代码如下所示：

```

ObjectIdentity oid = new ObjectIdentityImpl(object);

```

```
MutableAcl acl = mutableAclService.createAcl(oid);
acl.insertAce(0, BasePermission.ADMINISTRATION,

    new PrincipalSid(getUsername()), true);
```

需要创建一个 **CreateAclEntryAfterInvocationProvider**，将它注册到 Spring Security 中就可以在方法调用后进行对应的操作了。

```
public Object decide(Authentication authentication, Object object,
    ConfigAttributeDefinition config,
        Object returnedObject) throws AccessDeniedException {
    Iterator iter = config.getConfigAttributes().iterator();

    while (iter.hasNext()) {
        ConfigAttribute attr = (ConfigAttribute) iter.next();

        if (this.supports(attr)) {
            Object domainObject = getDomainObjectInstance(object,
                processDomainObjectClass);

            Iterator cit = this.handlers.iterator();
            while (cit.hasNext()) {
                AclHandler handler = (AclHandler) cit.next();
                if (handler.supports(domainObject, returnedObject)) {
                    handler.create(authentication, domainObject, config,
                        returnedObject);
                    break;
                }
            }
        }
    }

    return returnedObject;
}
```

xml 文件中的配置如下：

```
<bean id="afterAclCreate"
    class="com.family168.springsecuritybook.ch303.afterinvocation.CreateAclEntryAfterIn
    vocationProvider">
    <sec:custom-after-invocation-provider/>
    <constructor-arg ref="aclService"/>
```

```

        <constructor-arg>
            <list>

                <util:constant
                    static-field="org.springframework.security.acs.domain.BasePermission.ADMINISTRATIO
N"/>
            </list>
        </constructor-arg>
        <property name="processDomainObjectClass"
            value="com.family168.springsecuritybook.ch303.AclDomainClass"/>
        <property name="handlers">
            <list>
                <ref local="aclHandler"/>
            </list>
        </property>
    </bean>

```

为了指定在哪些方法调用后执行 `CreateAclEntryAfterInvocationProvider`，我们在方法上使用 `AFTER_ACL_CREATE` 注解。

```

@Transactional
@Secured({"ROLE_USER", "AFTER_ACL_CREATE"})
public void save(Account account) {
    list.add(account);
}

```

这样在 `save(account)` 方法执行后，就会自动根据 `account` 这个对象生成对应的 `acl` 信息。

48.2. 自动删除acl

自动删除 `acl` 的步骤与创建 `acl` 的大致相同。

首先来看一下删除 `acl` 信息的代码：

```

ObjectIdentity oid = new ObjectIdentityImpl(object);
mutableAclService.deleteAcl(oid, false);

```

其次是创建 `DeleteAclEntryAfterInvocationProvider`。

```

public Object decide(Authentication authentication, Object object,
ConfigAttributeDefinition config,
    Object returnedObject) throws AccessDeniedException {
    Iterator iter = config.getConfigAttributes().iterator();

    while (iter.hasNext()) {
        ConfigAttribute attr = (ConfigAttribute) iter.next();

        if (this.supports(attr)) {
            Object domainObject = getDomainObjectInstance(object,
processDomainObjectClass);

            Iterator cit = this.handlers.iterator();
            while (cit.hasNext()) {
                AclHandler handler = (AclHandler) cit.next();
                if (handler.supports(domainObject, returnedObject)) {
                    handler.delete(authentication, domainObject, config,
returnedObject);
                    break;
                }
            }
        }
    }

    return returnedObject;
}

```

xml 中的配置如下所示:

```

<bean id="afterAclDelete"
class="com.family168.springsecuritybook.ch303.afterinvocation.DeleteAclEntryAfterIn
vocationProvider">
    <sec:custom-after-invocation-provider/>
    <constructor-arg ref="aclService"/>
    <constructor-arg>
        <list>
            <util:constant
static-field="org.springframework.security.acs.domain.BasePermission.ADMINISTRATIO
N"/>
        </list>
    </constructor-arg>

```

```

    <property name="processDomainObjectClass"
value="com.family168.springsecuritybook.ch303.AclDomainClass"/>
    <property name="handlers">

        <list>
            <ref local="aclHandler"/>
        </list>
    </property>
</bean>

```

最后记得使用 **AFTER_ACL_DELETE** 对方法进行注解。

```

@Transactional
@Secured({"ROLE_USER", "AFTER_ACL_DELETE"})
public void remove(Account account) {
    list.remove(account);
}

```

48.3. 根据id删除acl

上面演示的 `remove()` 方法中需要传入 `account` 对象的实例才能起作用，有时我们希望使用 `removeById(id)` 的方式，只通过对象的 `id` 就可以删除这个对象的实例。这时 **AfterInvocation** 就无法判断当前的 `id` 实际对应的是哪个类，因此也就无法删除对应的 `acl` 信息了。

为了解决这个问题，我们引入了 **AclDomainAware** 注解。

```

@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface AclDomainAware {
    Class value();
}

```

可以为它设置一个 **Class** 值，使用它标示当前方法中 `id` 所对应的类型。

```

@Transactional
@Secured({"ACL_DELETE", "AFTER_ACL_DELETE"})
@AclDomainAware(Account.class)
public void removeById(Long id) {
    Account account = this.get(id);
}

```

```
list.remove(account);  
}
```

这样 `DeleteAclEntryAfterInvocationProvider` 就可以通过 `AclDomainAware` 中设置的类型和 `id` 的值找到对应实例的信息并删除之。

对 `AclDomainAware` 进行扩展后，我们也可以让它负责处理那些参数中包含 `id` 的情况，扩展 `AclEntryVoter`，让它可以使用 `AclDomainAware` 中的类型和 `id` 值生成的 `acl` 信息，以此进行 `acl` 的权限控制。

`IdParameterAclEntryVoter` 代码如下所示：

```
public class IdParameterAclEntryVoter extends AclEntryVoter {  
  
    public IdParameterAclEntryVoter(AclService aclService, String  
processConfigAttribute, Permission[] requirePermission) {  
        super(aclService, processConfigAttribute, requirePermission);  
    }  
  
    @Override  
    protected Object getDomainObjectInstance(Object secureObject) {  
        Object[] args;  
        Class[] params;  
  
        if (secureObject instanceof MethodInvocation) {  
            MethodInvocation invocation = (MethodInvocation) secureObject;  
            params = invocation.getMethod().getParameterTypes();  
            args = invocation.getArguments();  
        } else {  
            JoinPoint jp = (JoinPoint) secureObject;  
            params = ((CodeSignature)  
jp.getStaticPart().getSignature()).getParameterTypes();  
            args = jp.getArgs();  
        }  
  
        for (int i = 0; i < params.length; i++) {  
            if (getProcessDomainObjectClass().isAssignableFrom(params[i])) {  
                return args[i];  
            }  
        }  
  
        MethodInvocation invocation = (MethodInvocation) secureObject;
```

```

        Method method = invocation.getMethod();

        Serializable id = null;

        for (int i = 0; i < params.length; i++) {

            if (Serializable.class.isAssignableFrom(params[i])) {
                id = (Serializable) invocation.getArguments()[i];
                break;
            }
        }

        if (id == null) {
            throw new AuthorizationServiceException("MethodInvocation: "
                + invocation + " did not provide any ID argument.");
        }

        if (method.isAnnotationPresent(AclDomainAware.class)) {
            try {
                Class domainClass = method.getAnnotation(AclDomainAware.class)
                    .value();
                Object instance = domainClass.newInstance();
                Method setter = domainClass.getDeclaredMethod("setId", new
                    Class[] {id.getClass()});
                setter.invoke(instance, new Object[] {id});
                return instance;
            } catch (Exception ex) {
                ex.printStackTrace();
            }
        }

        throw new AuthorizationServiceException("Secure object: " + secureObject
            + " did not provide any argument of type: " +
            getProcessDomainObjectClass());
    }
}

```

实例在 ch303。

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。
- rbac0: RBAC0 模型。
- 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 49 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。

49.1. 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：

```
[INFO] [dependency:tree {execution: default-cli}]
[INFO] com.family168.springsecuritybook:ch401:war:0.1
[INFO] +-
org.springframework.security:spring-security-taglibs:jar:2.0.5.RELEASE:compile
[INFO] | +-
org.springframework.security:spring-security-core:jar:2.0.5.RELEASE: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:2.0.5.RELEASE: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
[INFO] +- org.hsqldb:hsqldb:jar:1.8.0.10:compile
[INFO] +- javax.servlet:servlet-api:jar:2.4:provided
[INFO] +- taglibs:standard:jar:1.1.2:compile
[INFO] +- javax.servlet:jstl:jar:1.1.2:compile
[INFO] \- net.sf.ehcache:ehcache:jar:1.6.0:compile

```

项目的目录结构如下：

```

+ ch401/
+ src/
+ main/
+ java/❶
+ com/
+ family168/
+ springsecuritybook/
+ ch401
+ * 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

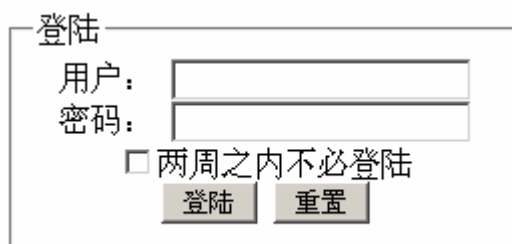
* user-changePassword.jsp

* 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。

49.2. 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。[第 4 章 自定义登陆页面](#)



The diagram shows a login form titled "登陆" (Login). It contains two input fields: "用户:" (User) and "密码:" (Password). Below these fields is a checkbox labeled "两周之内不必登陆" (No need to login within two weeks). At the bottom of the form are two buttons: "登陆" (Login) and "重置" (Reset).

图 49.1. 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>

```

49.3. 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](#)

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

图 49.2. 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。

49.4. 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](#)

Create User

Username:

x

Password:

●

Confirm Password:

●

Enabled:

☒

Authorities:

x

提交查询

重置

图 49.3. 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](#)

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 49.4. 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。

user already exists.

Create User

Username:

Password:

Confirm Password:

Enabled:

☒

Authorities:

图 49.5. 操作错误

添加用户操作过程中, `UserController.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。

49.5. 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](#)

Edit User

Username: x

Enabled: ☒

Authorities:

图 49.6. 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](#)

User Info

Username: x
Password: [PROTECTED]
Enabled: true
Authorities: x

图 49.7. 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。

49.6. 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。

Change User's Password

Old Password:

New Password:

Confirm Password:

图 49.8. 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。

第 50 章 用户组控制台

在最简权限控制台的基础上添加对用户组的支持。

用户组控制台中将添加如下功能：浏览用户组，新增用户组，修改用户组，删除用户组，为用户组授权，将用户添加到用户组中。

50.1. 添加对用户组的支持

默认情况是不会启用用户组功能的，我们需要为 `userDetailsManager` 设置 `enableGroups` 属性。

```
<bean id="userDetailsManager"
class="org.springframework.security.userdetails.jdbc.JdbcUserDetailsManager">
    <property name="dataSource" ref="dataSource"/>
    <property name="userCache" ref="userCache"/>
    <property name="enableGroups" value="true"/>
</bean>
```

这样就在系统中开启了用户组功能，与用户组有关的数据库表结构参考：[第 41 章 使用用户组](#)。

50.2. 浏览用户组

这一步通过 `GroupManager` 接口中定义的 `findAllGroups()` 方法获得数据库中所有用户组的名称，再根据用户组名称获得对应的成员和权限。

```
public List<UserGroupBean> getAll() {
    List<UserGroupBean> list = new ArrayList<UserGroupBean>();
    String[] groups = groupManager.findAllGroups();

    for (String groupName : groups) {
        String[] members = groupManager.findUsersInGroup(groupName);
        GrantedAuthority[] authorities = groupManager
            .findGroupAuthorities(groupName);
        UserGroupBean bean = new UserGroupBean();
```

```

        bean.setName(groupName);
        bean.setMembers(members);
        bean.setAuthorities(authorities);
        list.add(bean);
    }

    return list;
}

```

页面上将显示用户组名称，拥有的成员和分配的权限。

[Create Group](#)

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

Group Name	Members	Authorities	Operations
GROUP_ADMIN	admin	ROLE_ADMIN, ROLE_USER	View Update Remove
GROUP_USER	user	ROLE_USER	View Update Remove

图 50.1. 显示所有用户组

50.3. 创建用户组

创建用户组需要指定用户组的名称和用户组的权限。

```

public void save(String groupName, String[] authorities) {
    GrantedAuthority[] gas = new GrantedAuthority[authorities.length];

    for (int i = 0; i < authorities.length; i++) {
        gas[i] = new GrantedAuthorityImpl(authorities[i].trim());
    }

    groupManager.createGroup(groupName, gas);
}

```

[Back](#)

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

Create Group

Group Name:

Authorities:

图 50.2. 填写用户组信息

[Create Group](#)

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

success			
Group Name	Members	Authorities	Operations
GROUP_ADMIN	admin	ROLE_ADMIN, ROLE_USER	View Update Remove
GROUP_USER	user	ROLE_USER	View Update Remove
GROUP_DEFAULT		ROLE_USER	View Update Remove

图 50.3. 创建成功

50.4. 修改用户组

可以修改用户组的名称，用户组的成员和用户组的权限。

```
public void update(String oldName, String groupName, String[] members,
    String[] authorities) {
    String[] usernames = groupManager.findUsersInGroup(oldName);

    for (String username : usernames) {
        groupManager.removeUserFromGroup(username, oldName);
    }

    for (String member : members) {
        if ((member != null) && !member.equals("")) {
            groupManager.addUserToGroup(member, oldName);
        }
    }
}
```

```

    }

    GrantedAuthority[] gas = groupManager.findGroupAuthorities(groupName);

    for (GrantedAuthority ga : gas) {
        groupManager.removeGroupAuthority(oldName, ga);
    }

    for (int i = 0; i < authorities.length; i++) {
        String auth = authorities[i];

        if ((auth != null) && !auth.trim().equals("")) {
            GrantedAuthority ga = new
GrantedAuthorityImpl(auth.trim());
            groupManager.addGroupAuthority(oldName, ga);
        }
    }

    groupManager.renameGroup(oldName, groupName);
}

```

[Back](#)

Manage User

| username: user

| [Change Pass](#)

Edit Group

Group Name:

Members:

Authorities:

图 50.4. 修改用户组

实例代码在 ch402。

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-2.0.5.RELEASE-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。

namespace 中没有支持这个参数，如果想使用 group，只要设置 group-authorities-by-username-query 这个属性，就会自动打开 enableGroups。

B.7 Q: 如何判断用户是否登录到系统中。

.

A: 有三种方式判断用户是否登录进了系统。

如果没有使用 anonymous，
SecurityContextHolder.getContext().getAuthenticaiton() == null 时
用户就没有登录。

如果使用了 anonymous，
SecurityContextHolder.getContext().getAuthentication()
instanceof AnonymousAuthenticationToken 时用户就没有登录。

如果可以获得 HttpSession，当
session.getAttribute(HttpSessionContextIntegrationFilter.SPRING_
SECURITY_CONTEXT_KEY) == null 时，用户就没有登录。

B.8 Q: 为什么在过滤器链里多加了一个 FilterSecurityInterceptor，原来的
FilterSecurityInterceptor 就不控制请求了。

.

A: 因为 FilterSecurityInterceptor 会向 request 中写入一个标记，用于
标记是否已经控制了当前请求，这样可以避免对同一个请求多次处理。因
为前一个 FilterSecurityInterceptor 会在 request 中做一个标记，这就
会导致第二个 FilterSecurityInterceptor 不会再次执行了。

B.9 Q: 如何记录登录 ip 和登录时间。

.

A: 参考第 40 章 自定义过滤器 中的内容，自定义一个过滤器，将用户登
录ip和登录时间记录到session中。

B.1 Q: 如何在新建用户时，使用 saltValue 对密码进行极爱？
0.

A: 可以参考第 49 章 最简控制台 中的第四节 添加用户，使用String
encodedPassword = passwordEncoder.encodePassword(password,
username); 获得加密后的密码。

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。

C.1. Hello World

使用 Spring Security-3.0.0.M1 制作了一个 Helloworld，不需要修改项目中的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。

C.2. Spring-EL

下面详细介绍一下 Spring Security 3 中最新支持的 Spring-EL 表达式。

如果在配置中启用 Spring-EL 表达式功能，只要为 http 标签配置一个参数即可。

```
<http auto-config="true" use-expressions="true">
  <intercept-url pattern="/admin.jsp" access="hasRole('ROLE_ADMIN') and
hasIpAddress('192.168.1.0/24')"/>
  <intercept-url pattern="/**" access="hasRole('ROLE_USER')"/>
</http>
```

唯一的问题就是，如果启用了 Spring-EL 就不能再使用原来的方式了，所有的权限都要使用 Spring-EL 表达，否则会在访问被保护资源时抛出无法解析 Spring-EL 的异常。

既然说到 Spring-EL 就不能不提 Spring Security-3.0.0.M1 中新支持的 Pre-Post 注解，它是用来替代 MethodInvocationInterceptor 和 AfterInvocationInterceptor 一套新注解。

启用 Pre-Post 注解

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

```
<expression-handler ref="expressionHandler"/>
</global-method-security>
```

调用前权限控制。

拥有 **ROLE_USER** 或 **ROLE_ADMIN** 才能调用 **save()**方法。

```
@PreAuthorize("hasRole('ROLE_USER') or hasRole('ROLE_ADMIN')")
public void save(String messageContent, String owner) {
    // ...
}
```

参数 **message** 的 **owner** 属性与 **principal** 的 **name** 属性相同才允许调用 **modifyMessage()**方法。

```
@PreAuthorize("#message.owner == principal.name")
public void modifyMessage(Message message, String messageContent) {
    // ...
}
```

拥有对 **message** 的 **acl** 管理权限才可以调用 **deleteMessage()**方法。

```
@PreAuthorize("hasPermission(#message, 'admin')")
public void deleteMessage(Message message) {
    // ...
}
```

调用后权限控制。

判断是否有权返回对象，拥有对返回对象的 **read** 或 **admin** 的 **acl** 权限才可以查看 **Message** 对象。

```
@PostAuthorize("hasPermission(returnObject, 'read') or hasPermission(returnObject,
'administration')")
public Message get(Long id) {
    // ...
}
```

将返回集合对象中无权限的信息过滤掉。将 List 中不符合不具有 read 或 admin 的 acl 权限的对象从 list 过滤掉。

```
@PostFilter("hasPermission(filterObject, 'read') or hasPermission(filterObject, 'administration')")
public List getAll() {
    // ...
}
```

Spring-EL 十分强大，这样应用也比使用 MethodInvocationInterceptor 和 AfterInvocationInterceptor 更清晰便捷。

实例在 x02 下。

C.3. RoleHierarchy

这个问题后来发现是因为 wesee 同志搞错了，想用 Role 继承实现 User 继承的功能。实际上 Sid 中实现 Rolehierarchy 应该是没有任何意义的，反倒是 user 继承应该还有点儿意思。

使用 RoleHierarchy 的实例在 x03 下。

C.4. Success Handler

为了实现更好的扩展性，3.x 中开始专门提供了 AuthenticationSuccessHandler 和 AuthenticationFailureHandler，用这两者控制用户认证成功和失败的情况。我们可以自己实现 successHandler，在用户登录成功之后向 session 中任意放任何东西了。

```
public void onAuthenticationSuccess(HttpServletRequest request, HttpServletResponse response,
    Authentication authentication) throws ServletException, IOException {

    HttpSession session = request.getSession();
    UserDetails userDetails = (UserDetails) authentication.getPrincipal();
    if (userDetails.getUsername().equals("user")) {
        session.setAttribute("email", "user@163.com");
    } else if (userDetails.getUsername().equals("admin")) {
        session.setAttribute("email", "admin@163.com");
    }

    super.onAuthenticationSuccess(request, response, authentication);
}
```

```
}
```

实例在 x04 下。

C.5. REST下的权限控制

RESTful 是蛮流行的一种 web 资源访问方式,它最大限度利用了 http 协议的功能表达资源请求的内涵。

- GET /app/messages: 表示获得所有 Message 的信息。
- GET /app/messages/1: 表示获得 id=1 的 Message 的信息。
- POST /app/messages: 表示创建一条 Message 信息。
- PUT /app/messages/1: 表示更新 id=1 的 Message 的信息。
- DELETE /app/messages/1: 表示删除 id=1 的 Message 的信息。

从上面的例子中可以看出,我们只用了/app/messages 和/app/messages/*两类 URL,就表示了 Message 的 CRUD 所有操作。这时如果我们希望对这些操作进行权限控制,就不能仅仅根据 URL 地址来判断了,而是需要限制 http 请求的 method 参数。

下面我们对这些操作做出限制,只有 ROLE_ADMIN 才能添加,更新,删除 Message, ROLE_USER 只能查看 Message 的信息。

```
<http auto-config='true'>
  <intercept-url pattern="/admin.jsp" access="ROLE_ADMIN" />
  <intercept-url pattern="/app/messages" access="ROLE_ADMIN" method="POST" />
  <intercept-url pattern="/app/messages/*" access="ROLE_ADMIN" method="PUT" />
  <intercept-url pattern="/app/messages/*" access="ROLE_ADMIN" method="DELETE" />
  <intercept-url pattern="/**" access="ROLE_USER" />
</http>
```

在 intercept-url 中使用 method 指定对应的 http 请求 method 参数就可以限制 REST 下定义的不同功能请求了,可以每次只能定义一个 method,如果支持 method="PUT,DELETE"这种形式的配置就更方便了。

实例参考 x05。

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}|{ssha}"
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}|{sha}"
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>

```

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)
);

```

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.securityldap`

`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。

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。

AuthorizedEvent 认证成功

PublicInvocationEvent 公用调用。

附录 H. RBAC模型（转载）

访问控制策略一般有以下几种方式：

- 自主型访问控制（Discretionary Access Control-DAC）：用户/对象来决定访问权限。信息的所有者来设定谁有权来访问信息以及操作类型（读、写、执行。。。）。是一种基于身份的访问控制。例如 UNIX 权限管理。
- 强制性访问控制（Mandatory Access Control-MAC）：系统来决定访问权限。安全属性是强制型的规定，它由安全管理员或操作系统根据限定的规则确定的，是一种规则的访问控制。
- 基于角色的访问控制（格/角色/任务）：角色决定访问权限。用组织角色来同意或拒绝访问。比 MAC、DAC 更灵活，适合作为大多数公司的安全策略，但对一些机密性高的政府系统部适用。
- 规则驱动的基于角色的访问控制：提供了一种基于约束的访问控制，用一种灵活的规则描述语言和一种 ixn 的信任规则执行机制来实现。
- 基于属性证书的访问控制：访问权限信息存放在用户属性证书的权限属性中，每个权限属性描述了一个或多个用户的访问权限。但用户对某一资源提出访问请求时，系统根据用户的属性证书中的权限来判断是否允许或句句

模型的主要元素

- 可视化授权策略生成器
- 授权语言控制台
- 用户、组、角色管理模块
- API 接口
- 授权决策引擎
- 授权语言解释器

H.1. RBAC模型介绍

RBAC(Role-Based Access Control - 基于角色的访问控制)模型是 20 世纪 90 年代研究出来的一种新模型，但从本质上讲，这种模型是对前面描述的访问矩阵模型的扩展。这种模型的基本概念是把许可权（Permission）与角色（Role）联系在一起，用户通过充当合适角色的成员而获得该角色的许可权。

这种思想世纪上早在 20 世纪 70 年代的多用户计算时期就被提出来了，但直到 20 世纪 90 年代中后期，RBAC 才在研究团体中得到一些重视。本章将重点介绍美国 George Mason 大学的 RBAC96 模型。

H.2. 有关概念

在实际的组织中，为了完成组织的业务工作，需要在组织内部设置不同的职位，职位既表示一种业务分工，又表示一种责任与权利。根据业务分工的需要，支援被划分为不同群体，各个群体的人根据其工作任务的需要被赋予不同的职责和权利，每个人有权了解与使用与自己任务相关的信息与资源，对于那些不应该被知道的信息则应该限制他们访问。这就产生了访问控制的需求。

例如，在一个大学中，有校长、副校长、训练部长、组织处长、科研处长、教保处长等不同的职位，在通常情况下，职位所赋予的权利是不变的，但在某个职位上工作的人可以根据需要调整。RBAC 模型对组织内部的这些关系与访问控制要求给出了非常恰当的描述。

H.2.1. 什么是角色

在 RBAC 模型中，工作职位被描述为“角色”，职位所具有的权利称为许可权。角色是 RBAC 模型中的核心概念，围绕这一概念实现了访问控制策略的形式化。特殊的用户集合和许可权的集合通过角色这一媒介在某个特定的时间内联系在一起。而角色确实相对稳定的，因为任何组织的分工、活动或功能一般是很少经常改变的。

可以有不同的动机去构造一个角色。角色可以表示完成特殊任务的资格，例如，是一个医师还是一个药师；角色也可以表示一种权利与责任，如工程监理。权利与责任不同于资格，例如，Alice 可能有资格领导几个部门，但他只能被分配负责一个部门的领导。通过多个用户的轮转，角色可以映射特殊责任的分配，例如，医师可以转换为管理者。RBAC 的模式及其实现可以方便的适应这种角色概念的多种表现。

在实际的计算机信息系统中，角色由系统管理员定义，角色的增加与删除、角色权利的增加与减少等工作都是由系统管理员完成的。根据 RBAC 的要求，用户被分配为某个特定角色后，就被赋予了该角色所拥有的权利和责任，这种授权方式是强制性的，用户只能被动的接受，不能自主的决定为角色增加或减少权力，也不能把自己角色的权利转首给用户，显然，这是一种非自主型的访问控制模式。

H.2.2. 角色与用户组

角色与用户组有何区别？

两者的主要区别是：用户组是用户的集合，但不可许可权的集合；而角色却同时具有用户集合和许可权集合的概念，角色的作用是把这两个集合联系在一起的中间媒介。

在一个系统中，如果用户组的许可权和成员仅可以被系统安全员修改的话，在这种机制下，用户组的机制是非常接近于角色的概念的。角色也可以在用户组的基础上实现，这有利于保持原有系统中的控制关系。在这种情况下，角色相当于一个策略不见，与用户组的授权及责任关系相联系，而用户组是实现角色的机制，因此，两者之间是策略与实现机制之间的关系。

虽然 RBAC 是一种无确定性质策略的模型，但它支持公认的安全原则：最小特权原则、责任分离原则和数据抽象原则。最小特权原则得到支持，是因为在 RBAC 模型中可以通过限制分配给角色权限的多少和大小来实现，分配给与某用户对应的角色的权限只要不超过该用户完成其任务的需要就可以了。

责任分离原则的实现，是因为在 RBAC 模型中可以通过在完成敏感任务过程中分配两个责任上互相约束的两个角色来实现，例如在清查账目时，只需要设置财务管理员和会计连个角色参加就可以了。

数据抽象是借助于抽象许可权这样的概念实现的，如在账目管理活动中，可以使用信用，借方等抽象许可权，而不是使用操作系统提供的读、写、执行等具体的许可权。但 RBAC 并不强迫实现这些原则，安全管理员可以允许配置 RBAC 模型使它不支持这些原则。因此，RBAC 支持数据抽象的程度与 RBAC 模型的实现细节有关。

在 20 世纪 90 年代期间，大量的专家学者和专门研究单位对 RBAC 的概念进行了深入研究，先后提出了许多类型的 RBAC 模型，其中以美国 George Mason 大学信息安全技术实验室（LIST）提出的 RBAC96 模型最具有系统性，得到普遍的公认。

RBAC96 是一个模型族，其中包括 $RBAC_0 \sim RBAC_3$ 四个概念性模型。基本模型 $RBAC_0$ 定义了完全支持 RBAC 概念的任何系统的最低需求。 $RBAC_1$ 和 $RBAC_2$ 两者都包含 $RBAC_0$ ，但各自都增加了独立的特点，它们被成为高级模型。在 $RBAC_1$ 中增加了角色分级的概念，一个角色可以从另一个角色继承许可权。 $RBAC_2$ 增加了一些限制，强调在 RBAC 的不同组件中在配置方面的一些限制。

$RBAC_1$ 和 $RBAC_2$ 之间是不可比的。 $RBAC_3$ 被成为统一模型，它包含了 $RBAC_1$ 和 $RBAC_2$ ，利用传递性，也把 $RBAC_0$ 包括在内。这些模型构成了 RBAC96 模型族。图 ap08-01 表示了族内各模型间的关系，图 ap08-02 是 $RBAC_3$ 模型的概念示意图。

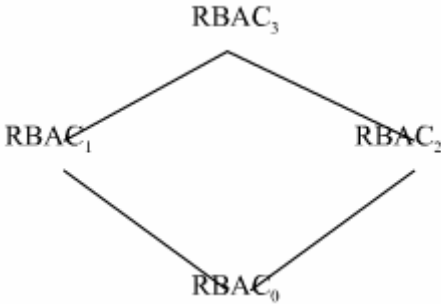


图 H.1. RBAC96 内各模型间的关系

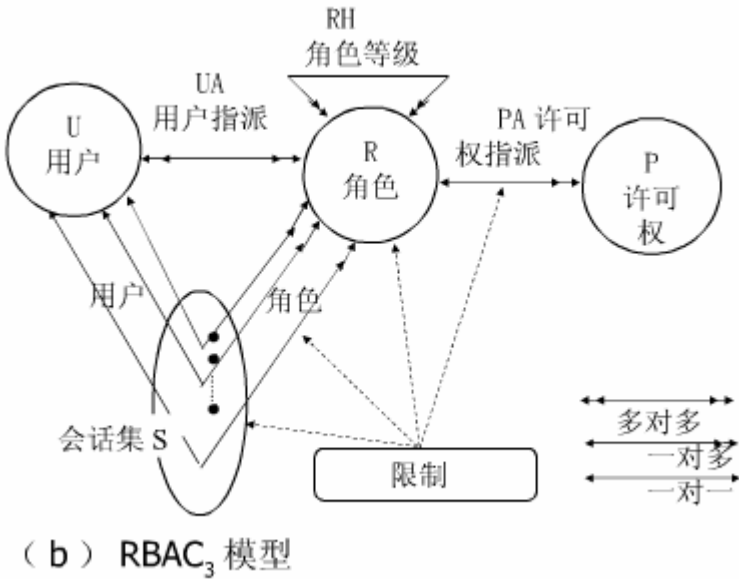


图 H.2. RBAC96 模型族

H.3. 基本模型RBAC₀

RBAC₀ 的模型结构可以参看图 ap08-02，但需要把途中的限制和角色等级两部分不包含在 RBAC₀ 模型中。该模型中包括用户（U）、角色（R）和许可权（P）的那个三类实体集合，此外还有一个会话集合（S）。

其中用户代表一个组织的职员；角色表示该组织内部的一项任务的功能或某个工作职务，它也表示该角色成员所拥有的权利和职责；许可权是用户对系统中各课题访问或操作的权利，客体是指系统中的数据客体和资源客体，例如，目录、文件、记录、端口、设备、内存或子网都是客体。

许可权因客体不同而不同，例如，对于目录、文件、设备、端口等类客体的操作权是读、写、执行等；对应数据库管理系统的客体是关系、元素、属性、记录、库文件、视图等，相应的操作权是 Select、Update、Delete、Insert 等；在会计应用中，相应的操作权是预算、信用、转移、创建和删除一个账目等。

图 ap08-02 说明了关系用户指派 UA（User Assignment）与许可权指派 PA（Permission Assignment）的含义，两者都是多对多的关系。RBAC 的关键就在于这两个关系，通过它们，一个用户将最终获得某些许可权并执行的权力。从图中角色的位置可以看粗，它是用户能够获取许可权的中间媒介。

会话集中的每个会话表示一个用户可以对应多个角色（指向角色有两个箭头）。在某个会话的持续期间，一个用户可以同时激活多个角色，而该用户所获得的许可权是所有这些角色的所拥有许可权的并集。

每个用户可以同时打开多个回话，每个会话都可以在工作站屏幕上用一个窗口显示。每个会话可以有不同活动角色的组合。RBAC₀ 的这一特点将受到最小特权原则的限制。如果一个用户在一次会话中激活所有角色的权利超过该用户被允许的权利，将受到最小权利原则的限制。

H.3.1. RBAC₀ 模型的形式定义如下

定义 1 RBAC₀ 模型由以下描述确定：

U、R、P、S 分别表示用户集合、角色集合、许可权集合和会话集合。

$PA \subseteq P \times R$ 表示许可权与角色之间多对多的指派关系。

$UA \subseteq U \times R$ 表示用户与角色之间多对多的指派关系。

用户： $S \rightarrow U$ 每个会话 s_i 到单个用户 $user(s_i)$ 的映射函数（常量代表会话的声明周期）。

角色： $S \rightarrow 2^R$ 每个会话 s_i 到角色子集 $roles(s_i) \{r | user(s_i, r') \in UA\}$ （能随时间改变）的映射函数，会话 s_i 有许可权 $U_r \in roles(s_i) \{p | (p, r') \in PA\}$ 。

在使用 $RBAC_0$ 模型时，应该要求每个许可权和每个用户至少应该被分配给一个角色。两个角色被分配的许可权完全一样是可能的，但仍是两个完全独立的角色，用户也有类似情况。角色可以适当的被看做是一种语义结构，是访问控制策略形式化的基础。

$RBAC_0$ 把许可权处理为非解释符号，因为其精确含义只能由实现确定且与系统有关。 $RBAC_0$ 中的许可权只能应用于数据和资源类客体，但不能应用于模型本身的组件。修改集合 U 、 R 、 P 和关系 PA 和 UA 的权限称为管理权限，后面将介绍 $RBAC$ 的管理模型。因此，在 $RBAC_0$ 中假定只有安全管理员才能修改这些组件。

会话是由单个用户控制的，在模型中，用户可以创建会话，并有选择的激活用户角色的某些子集。在一个会话中的角色的激活是由用户来决断的，会话的终止也是由用户初始化的。 $RBAC_0$ 不允许由一个会话去创建另一个会话，会话只能由用户创建。

H.4. 角色分级模型 $RBAC_1$

$RBAC_1$ 模型的特色是模型中的角色是分级的，不同级别的角色由不同的职责与权力，角色的级别形成偏序关系。图 ap08-03 说明了角色等级的概念。在图中位置处于较高处的角色的等级高于较低位置角色的等级。利用角色的分级概念可以限制继承的范围（scope）。

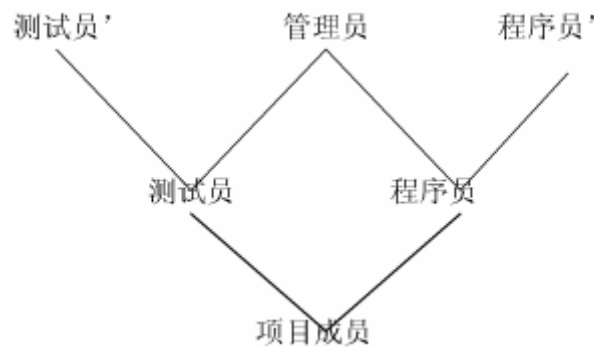


图 H.3. 角色分级的概念

图中项目成员的等级最低，角色程序员和测试员的等级都高于角色项目成员，并都可以继承项目成员的权利；角色管理员具有最高的等级，它可以继承测试员和程序员的权利。为了满足实际组织中一个角色不完全继承另一个角色所有权利与责任的需求，模型中引入了私有角色的概念，如图中的测试员'和程序员'分别是测试员和程序员的私有 ues ，它们可以分别继承测试员和程序员的某些专用权利。

显然，角色的等级关系具有自反性（自己可以继承自己）、传递性（A 继承 B，B 继承 C，则 A 继承 C）和反对称性（A 继承 B，B 继承 A，则 A=B），因此是偏序关系，下面是 $RBAC_1$ 的形式定义。

H.4.1. 定义 2: $RBAC_1$ 由以下内容确定

U 、 R 、 P 、 S 分别表示用户集合、角色集合、许可权集合和会话集合。

$PA \subseteq P \times R$ 表示许可权与角色之间多对多的指派关系。

$UA \subseteq U \times R$ 表示用户与角色之间多对多的指派关系。

$RH \subseteq R \times R$ 是对 R 的偏序关系，称为角色等级或角色支配关系，也可用 \geq 符号表示。

用户: $S \rightarrow U$ 每个会话 s_i 到单个用户 $user(s_i)$ 的映射函数（常量代表会话的声明周期）。

角色: $S \rightarrow 2^R$ 每个会话 s_i 到角色子集 $roles(s_i) = \{r | (r' \geq r) [user(s_i, r') \in UA]\}$ （能随时改变）的映射函数，会话 s_i 有许可权 $U_r \in roles(s_i) \{p | (r'' \leq r) [(p, r'') \in PA]\}$ 。

H.5. 限制模型 $RBAC_2$

$RBAC_2$ 模型是在 $RBAC_0$ 模型增加限制后形成的，它与 $RBAC_1$ 并不兼容。 $RBAC_2$ 定义如下：

H.5.1. 定义 3:

除了在 $RBAC_0$ 中增加了一些限制因素外， $RBAC_2$ 未加改变的来自于 $RBAC_0$ ，这些限制是用于确定 $RBAC_0$ 中各个组件的值是否是可接受的，只有那些可接受的值才是允许的。

$RBAC_2$ 中引入的限制可以施加到 $RBAC_0$ 模型中的所有关系和组件上。 $RBAC_2$ 中的一个基本限制是互斥角色的限制，互斥角色是指各自权限互斥相互制约的两个角色。对于这类角色一个用户在某一次活动中只能被分配其中的一个角色，不能同时获得两个角色的使用权。

例如，在审计活动中，一个角色不能同时被指派给会计角色和审计员角色。又如，在公司中，经理和副经理的角色也是互斥的，合同或支票只能由经理签字，不能由副经理签字。在为公司建立的 $RBAC_2$ 模型中，一个用户不能同时兼得经理和副经理两个角色。模型汇总的互斥限制可以支持权责分离原则的实现。

更一般化而言，互斥限制可以控制在不同的角色组合中用户的成员关系是否是可接受的。例如，一个用户可以既是项目 A 的程序员，也可以是项目 B 的测试员和项目 C 的验收员，但他不能同时成为同一个项目中的这 3 个角色。 $RBAC_2$ 模型可以对这种情况进行限制。

另一个用户指派限制的例子是一个角色限制其最大成员数，这被称为角色的基数限制。例如，一个单位的最高领导只能为 1 人，中层干部的数量也是有限的，一旦分配给这些角色的用户数超过了角色基数的限制，就不再接受新配给的用户了。

限制角色的最小基数实现起来有些困难。例如，如果规定占用某个角色的最小用户数，问题是系统如何在任何时刻都能知道这些占用者中的某个人没有消失，如果消失的话，系统又应该如何去做。

在为用户指派某个角色 A 时，在有的情况下要求该用户必须是角色 B 的一个成员，B 角色成为角色 A 的先决角色。先决角色（Prerequisite Roles）的概念来自于能力和适应性。对先决绝对的限制成为先决限制。一个通俗的例子是，一个数学副教授应该从数学讲师中提拔，讲师是任副教授的先决角色。但在实际系统中，不兼容角色之间的先决限制的情况也会发生。

在图 ap08-03 中，可以限制只有本项目的成员才有资格担任程序员的角色，通常在一个系统中，先决角色比新指派的角色的级别要低一些。但有的情况下，却要求只有当用户不是某个特殊角色时，才能担任另一个角色 A。如，需要执行回避策略时需要这样做，例如，本课题组成员不应当是本项目成果鉴定委员会的成员。这类限制也可以推广到许可权方面。

由于用户与角色的作用会与会话联系在一起，因此对会话也可以施加限制。例如，可以允许一个用户被指派给两个角色，但不允许在同一时间内把该用户在两个角色中都激活。另外，还可以限制一个用户在同一时间内可以激活的会话的数量，相应的，对该用户所激活的会话中所分配许可权的数量也可以施加限制。

前面提到的继承概念也可以视为一种限制。被分配给低级别角色的权限，也必须分配给该角色的所有上级角色。或等价的，一个指派给较高级别的角色的用户必须指派给该角色的所有下级角色。因此从某种角度上讲， $RBAC_1$ 模型是冗余的，它被包含在 $RBAC_2$ 中。但 $RBAC_1$ 模型比较简洁，用继承代替限制可使概念更清晰。

实现时可以用函数来实现限制，当为用户指定角色或为角色分配权限时就调用这些函数进行检查，根据函数返回的结果决定分配是否满足限制的要求，通常只对

那些可被有效检查和那些惯例性的一些简单限制给以实现，因为这些限制可以保持较长的时间。

模型中的限制机制的有效性建立在每个用户只有唯一标识符的基础上，如果一个实际系统支持用户拥有多标识符，限制将会失效。同样，如果同一个操作可以有两个以上的许可权来比准，那么，RBAC 系统也无法实施加强的基本限制和责任分离限制。因此要求用户与其标识符，许可与对应的操作之间一一对应。

H.6. 统一模型RBAC₃

RBAC₃ 把 RBAC₁ 和 RBAC₂ 组合在一起，提供角色的分级和继承的能力。但把这两种概念组合在一起也引起一些新问题。

限制也可以应用于角色等级本身，由于角色间的等级关系满足偏序关系，这种限制对模型而言是本质性的，可能会影响这种偏序关系。例如，附加的限制可能会限制一个给定角色的应有的下级角色的数量。

两个或多个角色由可能被限制成没有公共的上级角色或下级角色。这些类型的限制在概念角色等级的权力已经被分散化的情况下是有用哦，但是安全主管却希望对所有允许这些改变的方法加以限制。

在限制和角色的等级之间也会产生敏感的相互影响。在图 ap08-03 的环境中，一个项目成员不允许同时担任程序员与测试员的角色，但项目管理员所处的位置显然是违反了该限制。在某种情况 i 下由高等级的角色违反这种限制是可接受的，但在其他情况下又不允许这种违反现象发生。

从严格性的角度来讲，模型的规则不应该是一些情况下不允许而在另一情况下是允许的。类似的情况也会发生在对基数的限制上。假定限制一个用户之多能分配给一个橘色，那么对图中的测试员的一个指派能够未被这种限制吗？换句话说，基数限制是不是只能用于直接成员，是否也能应用于继承成员上？

私有角色的概念可以说明这些限制是有用的。同样在图 ap08-03 的环境中，可以把测试员'，程序员'和项目管理员 3 个角色说明为互斥的，它们处于同一等级，没有共同的上级角色，所以管理员角色没有违反互斥限制。通常私有角色和其他角色之间没有公共上级角色，因为它们是这个等级的最大元素，所以私有角色之间互斥关系可以无冲突的定义。

诸私有角色之间的相同部分可以被说明为具有 0 成员的最大技术限制。根据这种方法，测试员必须被指派给测试员'这个角色，而测试员角色就作为与管理员角色共享许可权的一种工具。

在前面的讨论中，我们都假设 RBAC 的所有组件都是由单个的安全员来管理里。但是，对于一个大系统而言，系统中的角色可能成百上千，再加上它们之间的复

杂关系，使得集中式的管理任务成为非常可怕的工作，因此通常由几个管理员小组来完成。能否用 RBAC 管理自己本身呢？

RBAC 的管理模型示于图 ap08-04。该图的上半部分本质上与图 ap08-02 相同，图中的限制针对所有成分的，图的下半部分是对上半部分关于管理角色 AR 和管理许可权 AP 与正规角色集 R 和许可权集 P 是分别不可相交的。这个模型显示，正规许可权只能分配给正规角色（RBAC 模型中定义的角色），管理许可权只能分配给管理角色。

H.7. 定义 4

管理许可权 AP 有权改变组成 $RBAC_0$ 、 $RBAC_1$ 、 $RBAC_2$ 或 $RBAC_3$ 的所有成分，但正规许可权 P 不能。管理许可权与正规许可权不相交，即 $AP \cap P = \emptyset$ 。管理许可权和正规许可权只能分别分配给管理角色 AR 和正规角色 R，并且 $AR \cap R = \emptyset$ 。

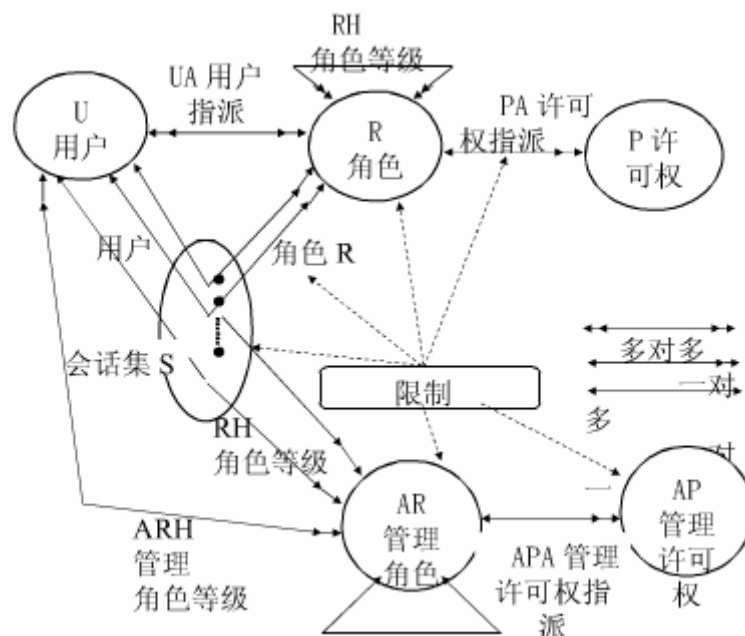


图 H.4. 管理模型示意图

在图 ap08-04 的上半部可以对应 $RBAC_0$ 、 $RBAC_1$ 、 $RBAC_2$ 和 $RBAC_3$ 模型，类似地下半部可以对应 $ARBAC_0$ 、 $ARBAC_1$ 、 $ARBAC_2$ 和 $ARBAC_3$ 模型，此处的 A 表示“管理”。 $ARBAC_0 \sim ARBAC_3$ 形成了 RBAC 的管理模型族，成为 ARBAC97。通常我们期望管理模型比 RBAC 模型本身简单一些，因此可以利用 $ARBAC_0$ 管理 $RBAC_3$ ，而不是用 $ARBAC_3$ 去管理 $RBAC_0$ 模型。

H.8. 在ARBAC97 中，包括三种组件

URA87：用户-角色指派。该组件涉及用户-指派关系 UA 的管理，该关系把用户与角色关联在一起。对该关系的修改权由管理角色控制，这样，管理角色中的成员有权管理正规角色中的成员关系。把一个用户指定为管理角色是在 URA97 以外完成的，并假定是由安全员完成的。

PRA97：许可权-角色指派。该组件涉及角色-许可权的指派与撤销。从角色观点来看，用户和许可权有类似的特点，它们都是由角色联系在一起的实在实体。因此，可以把 PRA97 看做是 URA97 的对偶组件。

RRA97：角色-角色指派。为了便于对角色的管理，对角色又进行了分类。该组件涉及 3 类角色，它们是：

1. 能力（Abilities）角色——进以许可权和其他能力做成成员的角色。
2. 组（Groups）角色——仅以用户和其他组为成员的一类角色。
3. UP-角色——表示用户与许可权的角色，这类角色对其成员没有限制，成员可以使用户、角色、许可权、能力、组或其他 UP-角色。

区别这三类模型的主要原因是可以应用不同的管理模型去建立不同类型角色之间的关系。区分的动因首先是对能力的考虑，能力是许可权的集合，可以把该集合中所有许可权作为一个单位指派给一个角色。类似的，组是用户的集合，可以把该集合中所有许可权作为一个单位指派给一个角色。组和能力角色都似乎可以划分等级的。

在一个 UP-角色中，一个能力是否是其的一个成员是由 UP-角色是否支配该能力决定的，如果支配就是，否则就不是。相反的，如果一个 UP-角色被一个组角色支配，则这个组就是该 UP-角色的成员。

对 ARBAC97 管理模型的研究还在继续之中，对能力-指派与组-指派的形式化已基本完成，对 UP-角色概念的研究成果还未形式化。

H.9. RBAC模型的特点

符合各类组织机构的安全管理需求。RBAC 模型支持最小特权原则、责任分离原则，这些原则是任何组织的管理工作都需要的。这就使得 RBAC 模型由广泛的应用前景。

RBAC 模型支持数据抽象原则和继承概念。由于目前主程序设计语言都支持面向对象技术，RBAC 的这一特性便于在实际系统中应用实现。

模型中概念与实际系统紧密对应。RBAC 模型中的角色、用户和许可权等概念都是实际系统实际存在的实体，便于设计者建立现存的或待建系统的 RBAC 模型。

RBAC 模型仍素具访问控制类模型，本质是对访问矩阵模型的扩充，能够很好的解决系统中主体对客气的访问控制访问权力的分配与控制问题，但模型没有提供信息流控制机制，还不能完全满足信息系统的全部安全需求。

虽然也有人认为可以用 RBAC 去仿真基于格的访问控制系统(LBAC)，但 RBAC 对系统内部信息流的控制不是直观的，需要模型外的功能支持。有关信息流控制的作用域原理将在第四章介绍，届时读者可以进一步理解 RBAC 模型的这种缺陷。

RBAC 模型没有提供操作顺序控制机制。这一缺陷使得 RBAC 模型很难应用关于那些要求有严格操作次序的实体系统，例如，在购物控制系统中要求系统对购买步骤的控制，在客户未付款之前不应让他把商品拿走。RBAC 模型要求把这种控制机制放到模型外去实现。

RBAC96 模型和 RBAC97uanli 模型都故意回避了一些问题，如是否允许一个正在会话的用户再创建一个新会话，管理模型不支持用户和许可权的增加与删除等管理工作灯，都是需要解决而未提供支持的问题，这些问题都还在研究中，但是如果缺少这些能力的支持，模型的而应用也将受到影响。相反，访问绝阵模型提供了用户和权限修改功能，因此，不能说 RBAC 模型能够完全取代访问矩阵模型。