

Spring Security

安全权限管理手册

0.0.2

版权 © 2009 Lingo

论坛: <http://www.family168.com/bbs/>。

Email: xyz20003@gmail.com。

QQ群: 3038490。

2009-05-26 13:41:40

[序言](#)

[I. 基础篇](#)

[1. 一个简单的HelloWorld](#)

[1.1. 配置过滤器](#)

[1.2. 使用命名空间](#)

[1.3. 完善整个项目](#)

[1.4. 运行示例](#)

[2. 使用数据库管理用户权限](#)

[2.1. 修改配置文件](#)

[2.2. 数据库表结构](#)

[3. 自定义数据库表结构](#)

[3.1. 自定义表结构](#)

[3.2. 初始化数据](#)

[3.3. 获得自定义用户权限信息](#)

[3.3.1. 处理用户登陆](#)

[3.3.2. 检验用户权限](#)

[4. 自定义登陆页面](#)

[4.1. 实现自定义登陆页面](#)

[4.2. 修改配置文件](#)

[4.3. 登陆页面中的参数配置](#)

[4.4. 测试一下](#)

[5. 使用数据库管理资源](#)

[5.1. 数据库表结构](#)

[5.2. 初始化数据](#)

[5.3. 实现从数据库中读取资源信息](#)

[5.3.1. 需要何种数据格式](#)

[5.3.2. 替换原有功能的切入点](#)

[6. 控制用户信息](#)

[6.1. MD5加密](#)

[6.2. 盐值加密](#)

[6.3. 用户信息缓存](#)

[6.4. 获取当前用户信息](#)

[II. 保护web篇](#)

[7. 图解过滤器](#)

[7.1. HttpSessionContextIntegrationFilter](#)

[7.2. LogoutFilter](#)

[7.3. AuthenticationProcessingFilter](#)

[7.4. DefaultLoginPageGeneratingFilter](#)

[7.5. BasicProcessingFilter](#)

[7.6. SecurityContextHolderAwareRequestFilter](#)

[7.7. RememberMeProcessingFilter](#)

[7.8. AnonymousProcessingFilter](#)

[7.9. ExceptionTranslationFilter](#)

[7.10. SessionFixationProtectionFilter](#)

[7.11. FilterSecurityInterceptor](#)

[III. 保护method篇](#)

[IV. ACL篇](#)

[V. 最佳实践篇](#)

[A. 修改日志](#)

[B. 常见问题解答](#)

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

[下一页](#)

序言

序言

为啥选择Spring Security

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

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

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

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

内容结构组织

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

[第 I 部分 “基础篇”](#)。环境搭建，进行最简单的配置。

[第 II 部分 “保护web篇”](#)。谈谈对url的权限控制。

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

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

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

意见反馈

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

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

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

相关信息

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

教程相关的实例代码可以从google code上下载：<http://code.google.com/p/family168/downloads/detail?name=springsecurity-sample.rar&can=2&q=#makechanges>。

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

[上一页](#)

Spring Security

[起始页](#)

[下一页](#)

部分 I. 基础篇

部分 I. 基础篇

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

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

我们用使用的第三方依赖库关系如下所示：

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

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。

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

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

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

1.3. 完善整个项目

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


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

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

    <filter>
        <filter-name>springSecurityFilterChain</filter-name>
        <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-
class>
    </filter>
    <filter-mapping>
        <filter-name>springSecurityFilterChain</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

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

</web-app>
```

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

最终我们的整个项目会变成下面这样：

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

1.4. 运行示例

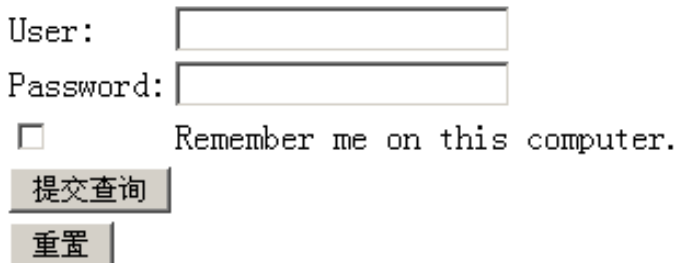
首先确保自己安装了Maven2。如果之前没用过Maven2，可以参考我们的Maven2教程<http://www.family168.com/oa/maven2/html/index.html>。

安装好Maven2之后，进入ch01目录，然后执行mvn。

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

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

Login with Username and Password



User:

Password:

☐ Remember me on this computer.

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. 登陆错误

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。

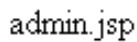
The image shows the text 'admin.jsp' in a monospaced font, which is the content displayed when the admin.jsp link is clicked.

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" />
    <user-name="user" password="user" authorities="ROLE_USER" />
</del>
  </del>user-service</del>
</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ch02。

^[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。

[上一页](#)

第 1 章 一个简单的HelloWorld

[上一级](#)

[起始页](#)

[下一页](#)

第 3 章 自定义数据库表结构

第 3 章 自定义数据库表结构

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

3.1. 自定义表结构

假设我们实际使用的表结构如下所示：

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

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

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



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

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

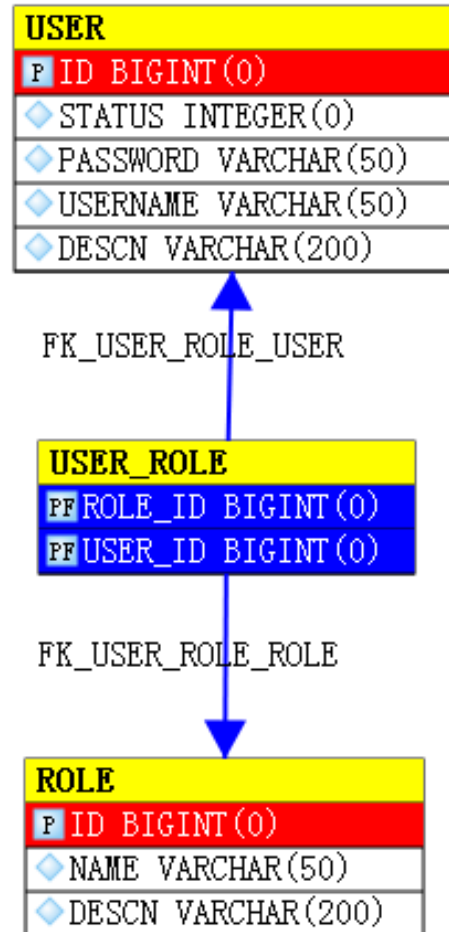


图 3.1. 数据库表关系

3.2. 初始化数据

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

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

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

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

3.3. 获得自定义用户权限信息

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

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

3.3.1. 处理用户登陆

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

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

3.3.2. 检验用户权限

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

以下SQL就可以获得当前用户所拥有的权限。

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

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

```
<authentication-provider>
  <jdbc-user-service data-source-ref="dataSource"
    ❶users-by-username-query="select username,password,status as enabled
```

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

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

实例见ch03。

[上一页](#)[上一级](#)[下一页](#)[第 2 章 使用数据库管理用户权限](#)[起始页](#)[第 4 章 自定义登陆页面](#)

第 4 章 自定义登陆页面

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

4.1. 实现自定义登陆页面

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

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

4.2. 修改配置文件

在xml中的http标签中添加一个form-login标签。

```
<http auto-config='true'>
  <intercept-url pattern="/login.jsp" access="IS_AUTHENTICATED_ANONYMOUSLY" />❶
  <intercept-url pattern="/admin.jsp" access="ROLE_ADMIN" />
  <intercept-url pattern="/**" access="ROLE_USER" />
  <form-login login-page="/login.jsp"❷
    authentication-failure-url="/login.jsp?error=true"❸
    default-target-url="/" />❹
```

```
</http>
```

- ❶ 让没登陆的用户也可以访问login.jsp。 [2]

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

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

这时我们访问系统显示的登陆页面将是我们上面创建的login.jsp。

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

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

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

4.3. 登陆页面中的参数配置

以下是我们创建的login.jsp页面的主要代码。

```
<div class="error ${param.error == true ? '' : 'hide'}">
    登陆失败<br>
    ${sessionScope['SPRING_SECURITY_LAST_EXCEPTION'].message}
</div>
<form action="${pageContext.request.contextPath}/j_spring_security_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。

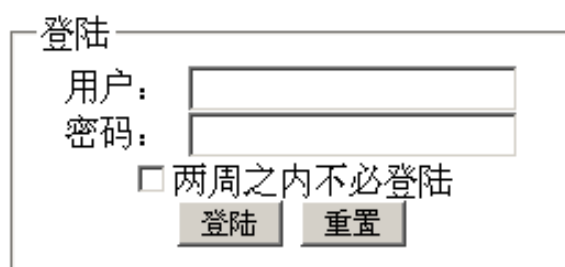


图 4.1. 进入登录页面

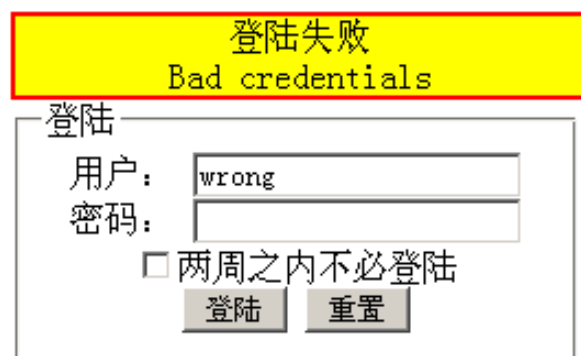


图 4.2. 用户登陆失败

实例见ch04。

[2] 有关匿名用户的知识，我们会在之后的章节中进行讲解。

[3] 登陆成功后跳转策略的知识，我们会在之后的章节中进行讲解。

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

[上一页](#)

第 3 章 自定义数据库表结构

[上一级](#)

[起始页](#)

[下一页](#)

第 5 章 使用数据库管理资源

第 5 章 使用数据库管理资源

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

5.1. 数据库表结构

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

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

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

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



```
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);
```

ER图如下所示:

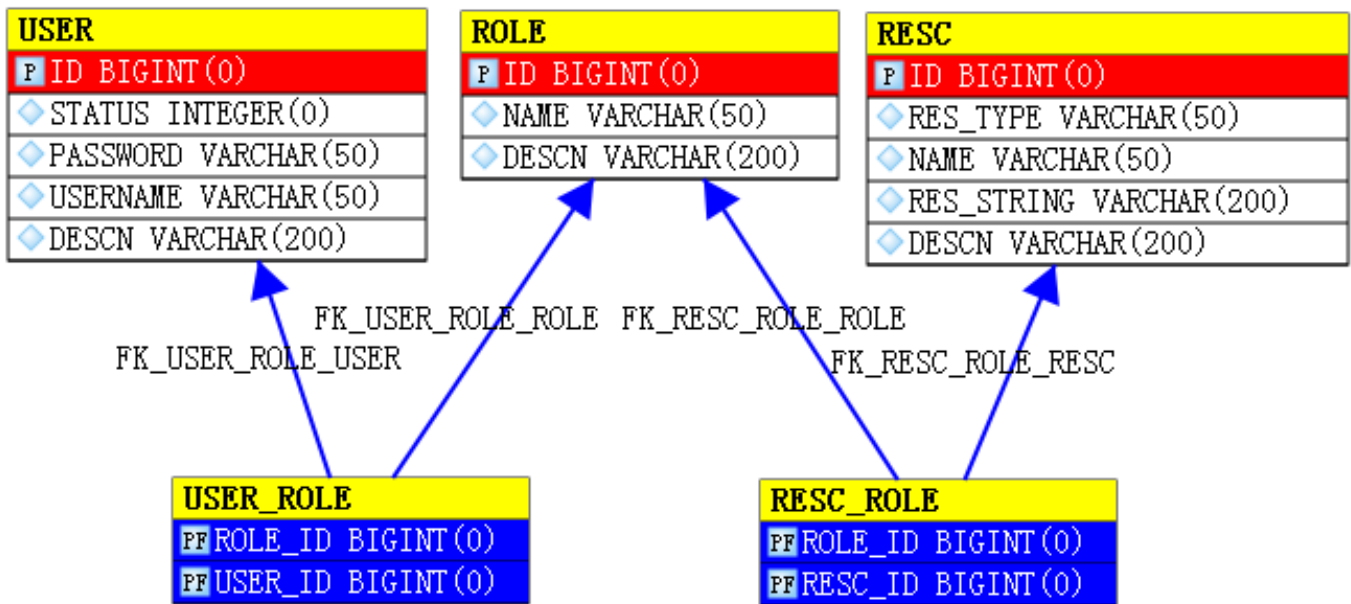


图 5.1. 数据库表关系

5.2. 初始化数据

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

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

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

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

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

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

5.3. 实现从数据库中读取资源信息

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

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

5.3.1. 需要何种数据格式

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

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

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

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

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

```
join resc re
on re.id=rr.resc_id
```

下面要开始编写实现代码了。

1. 搜索数据库获得资源信息。

我们通过定义一个MappingSqlQuery实现数据库操作。

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

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

        return resource;
    }
}
```

这样我们可以执行它的execute()方法获得所有资源信息。

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

    return resourceMapping.execute();
}
```

2. 使用获得的资源信息组装requestMap。

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

    ConfigAttributeEditor editor = new ConfigAttributeEditor();

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

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

```

    }

    return requestMap;
}

```

3. 使用urlMatcher和requestMap创建DefaultFilterInvocationDefinitionSource。

```

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

```

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

完整代码如下所示：

```

package com.family168.springsecuritybook.ch05;

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

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

import javax.sql.DataSource;

import org.springframework.beans.factory.FactoryBean;

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

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

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

    public boolean isSingleton() {
        return true;
    }

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

```

```

    }

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

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

        return resourceMapping.execute();
    }

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

        ConfigAttributeEditor editor = new ConfigAttributeEditor();

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

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

        return requestMap;
    }

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

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

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

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

        public String getUrl() {
            return url;
        }

        public String getRole() {
            return role;
        }
    }

```

```

    }
}

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

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

        return resource;
    }
}
}

```

5.3.2. 替换原有功能的切入点

在spring中配置我们编写的代码。

```

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

```

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

```

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

```

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

完整的配置文件如下所示：

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

  <http auto-config="true"/>

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

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

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

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

```
<beans:property name="driverClassName" value="org.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。

[上一页](#)

第 4 章 自定义登陆页面

[上一级](#)

[起始页](#)

[下一页](#)

第 6 章 控制用户信息

第 6 章 控制用户信息

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

6.1. MD5加密

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

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

为了使用MD5对密码加密，我们需要修改一下配置文件。

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

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

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

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

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

这样，我们只添加了一行配置，就为系统带来了密码加密的功能。

6.2. 盐值加密

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

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

为了解决这个问题，我们可以使用盐值加密“salt-source”。

修改配置文件：

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

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

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

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

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

6.3. 用户信息缓存

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

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

```
</authentication-provider>
```

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

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

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

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

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

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

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

    <cache
        name="userCache"
        maxElementsInMemory="100"❶
        eternal="false"❷
        timeToIdleSeconds="600"❸
        timeToLiveSeconds="3600"❹
        overflowToDisk="true"❺
    />
</ehcache>
```

❶ 内存中最多存放100个对象。

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

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

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

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。

部分 II. 保护web篇

```
信息: FilterChainProxy: FilterChainProxy[
    UrlMatcher = org.springframework.security.util.AntUrlPathMatcher
[requiresLowerCase='true'];
    Filter Chains: {
        /**=[
            org.springframework.security.context.HttpSessionContextIntegrationFilter
[ order=200; ],
            org.springframework.security.ui.logout.LogoutFilter[ order=300; ],
            org.springframework.security.ui.webapp.AuthenticationProcessingFilter
[ order=700; ],
            org.springframework.security.ui.webapp.DefaultLoginPageGeneratingFilter
[ order=900; ],
            org.springframework.security.ui.basicauth.BasicProcessingFilter
[ order=1000; ],
            org.springframework.security.wrapper.
SecurityContextHolderAwareRequestFilter[ order=1100; ],
            org.springframework.security.ui.rememberme.RememberMeProcessingFilter
[ order=1200; ],
            org.springframework.security.providers.anonymous.
AnonymousProcessingFilter[ order=1300; ],
            org.springframework.security.ui.ExceptionTranslationFilter
[ order=1400; ],
            org.springframework.security.ui.SessionFixationProtectionFilter
[ order=1600; ],
            org.springframework.security.intercept.web.
FilterSecurityInterceptor@e2fbef
        ]
    }
]
```

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。

打算将taglib的介绍也包含在这一部分，目前taglib的功能还不足以单独列出一章。

www.family168.com

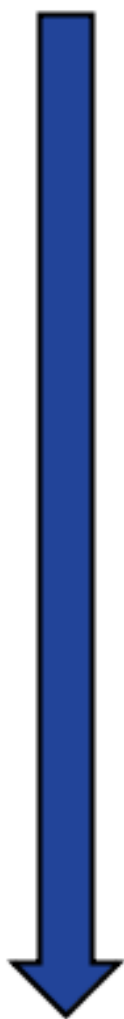
第 7 章 图解过滤器
部分 II. 保护web篇

[上一页](#)

[下一页](#)

第 7 章 图解过滤器

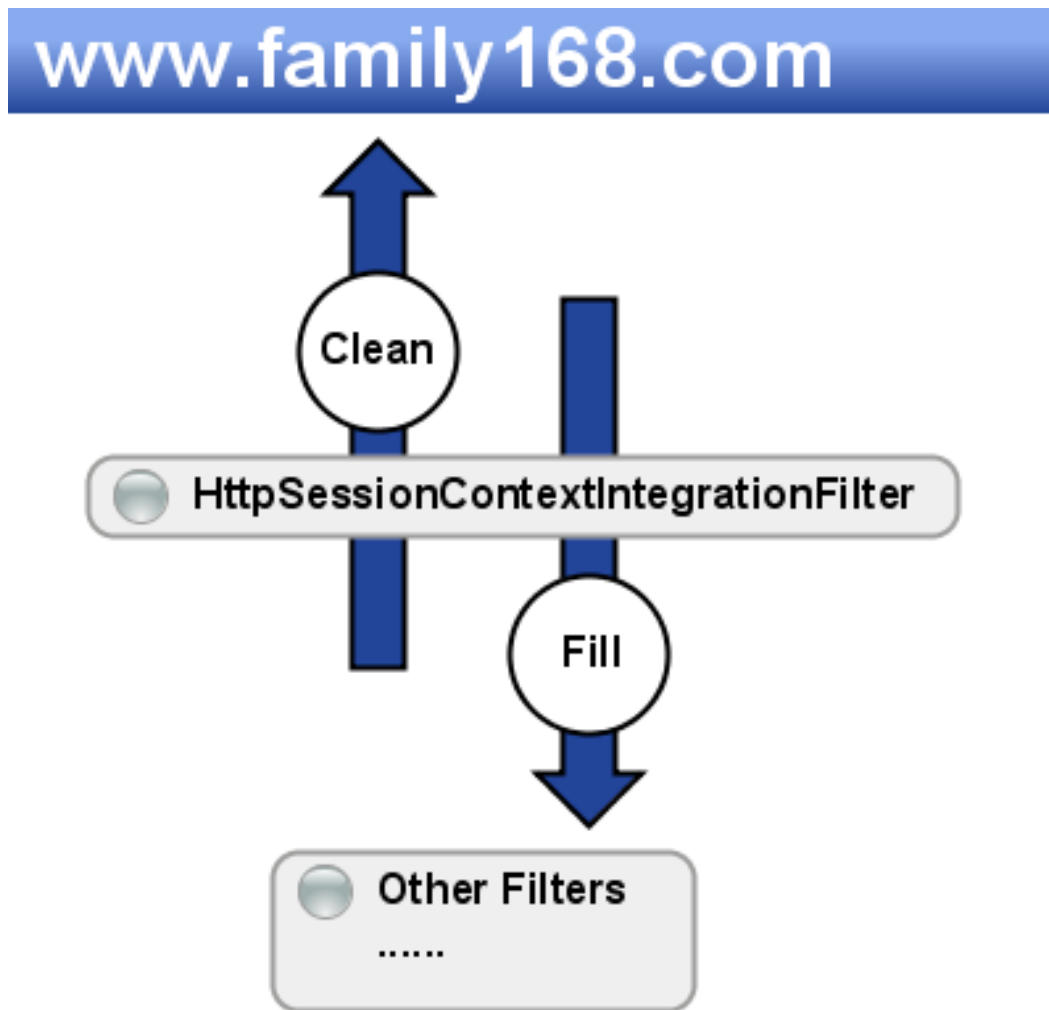
www.family168.com



- ☐ HttpSessionContextIntegrationFilter
- ☐ LogoutFilter
- ☐ AuthenticationProcessingFilter
- ☐ DefaultLoginPageGeneratingFilter
- ☐ BasicProcessingFilter
- ☐ SecurityContextHolderAwareRequestFilter
- ☐ RememberMeProcessingFilter
- ☐ AnonymousProcessingFilter
- ☐ ExceptionTranslationFilter
- ☐ SessionFixationProtectionFilter
- ☐ FilterSecurityInterceptor

图 7.1. `auto-config='true'`时的过滤器列表

7.1. HttpSessionContextIntegrationFilter

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

位于过滤器顶端，第一个起作用的过滤器。

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

用途二，在所有过滤器执行完毕后，清空SecurityContextHolder，因为SecurityContextHolder是基

于ThreadLocal的，如果在操作完成后清空ThreadLocal，会受到服务器的线程池机制的影响。

7.2. LogoutFilter

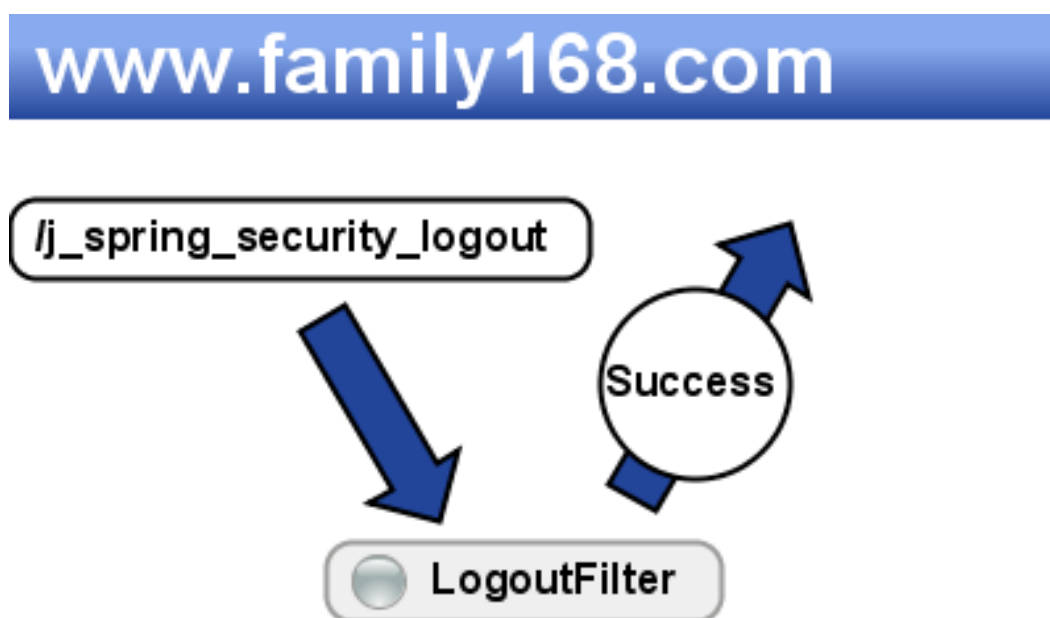


图 7.3. org.springframework.security.ui.logout.LogoutFilter

只处理注销请求，默认为/j_spring_security_logout。

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

7.3. AuthenticationProcessingFilter

```
org.springframework.security.  
ui.webapp.  
AuthenticationProcessingFilter
```

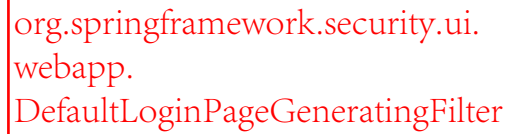
图 7.4. org.springframework.security.ui.webapp.AuthenticationProcessingFilter

处理form登陆的过滤器，与form登陆有关的所有操作都是在此进行的。

默认情况下只处理/j_ spring_security_check请求，这个请求应该是用户使用form登陆后的提交地址，form所需的其他参数可以参考：[第 4.3 节 “登陆页面中的参数配置”](#)。

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

7.4. DefaultLoginPageGeneratingFilter



```
org.springframework.security.ui.  
webapp.  
DefaultLoginPageGeneratingFilter
```

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

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

如果想自定义登陆页面，可以参考：[第 4 章 自定义登陆页面](#)。

7.5. BasicProcessingFilter

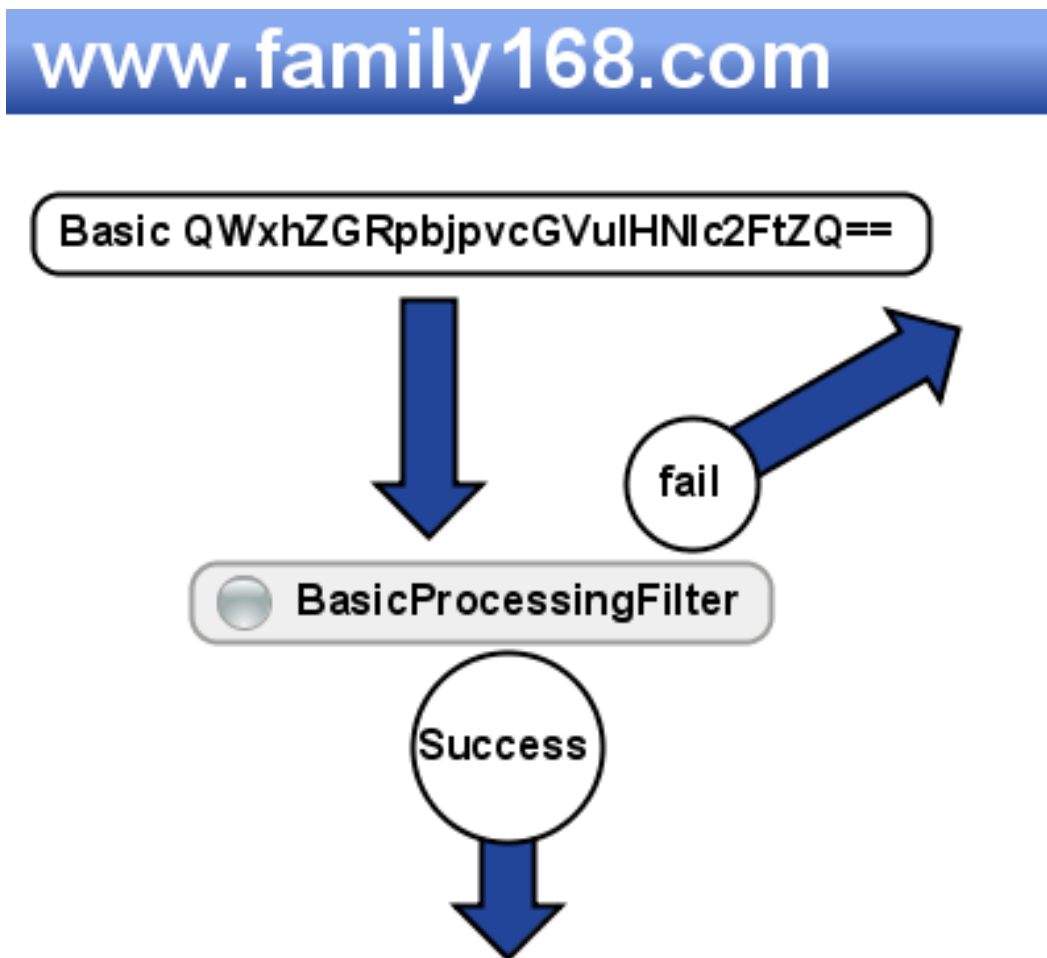


图 7.6. org.springframework.security.ui.basicauth.BasicProcessingFilter

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

7.6. SecurityContextHolderAwareRequestFilter

```
org.springframework.security.wrapper.  
SecurityContextHolderAwareRequestFilter
```

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

此过滤器用来包装客户的请求。目的是在原始请求的基础上，为后续程序提供一些额外的数据。

比如`getRemoteUser()`时直接返回当前登陆的用户名之类的。

7.7. RememberMeProcessingFilter

```
org.springframework.security.  
ui.rememberme.  
RememberMeProcessingFilter
```

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

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

7.8. AnonymousProcessingFilter

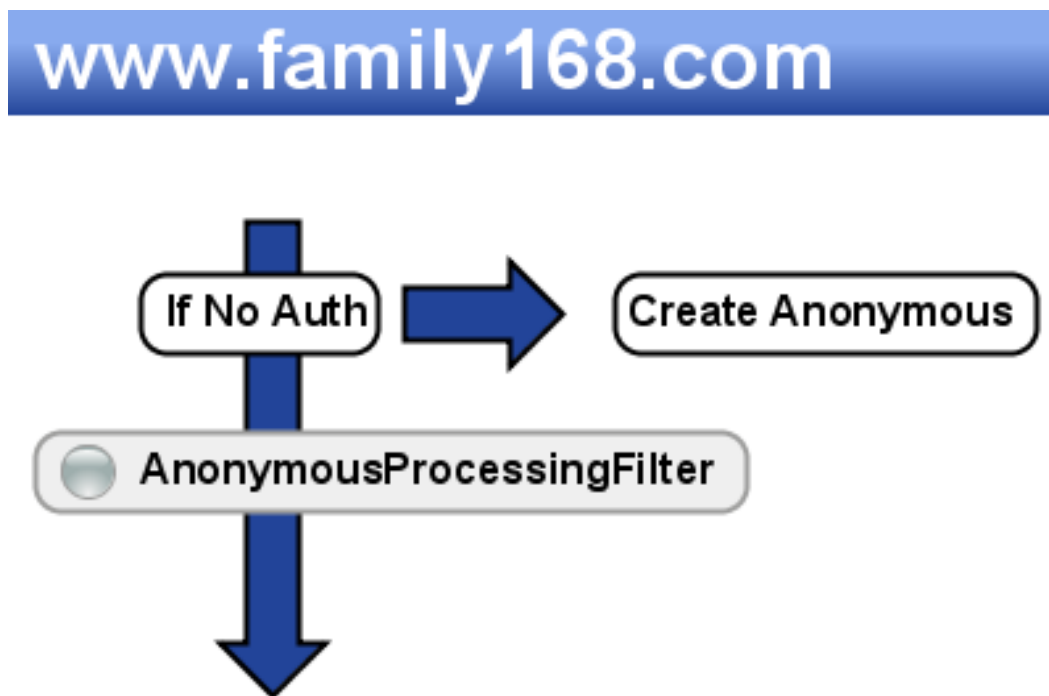


图 7.9. `org.springframework.security.providers.anonymous.
AnonymousProcessingFilter`

为了保证操作统一性，当用户没有登陆时，默认为用户分配匿名用户的权限。

7.9. ExceptionTranslationFilter

```
org.springframework.  
security.ui.  
ExceptionTranslationFilter
```

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

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

7.10. SessionFixationProtectionFilter

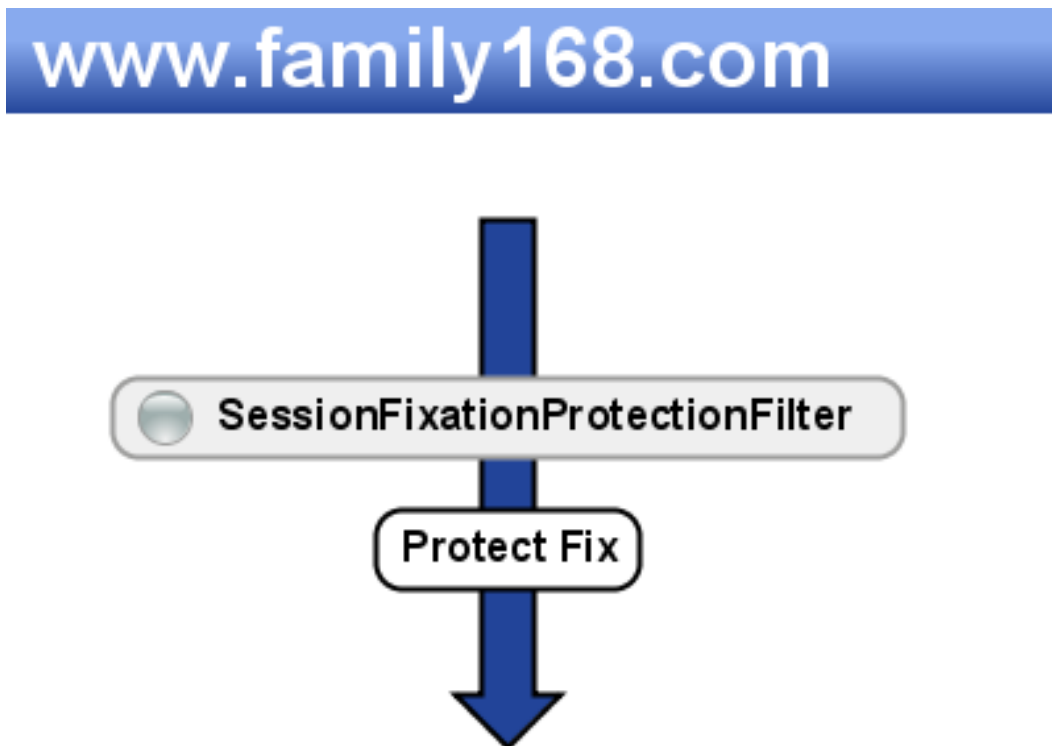


图 7.11. org.springframework.security.ui.SessionFixationProtectionFilter

防御session固化攻击。

7.11. FilterSecurityInterceptor

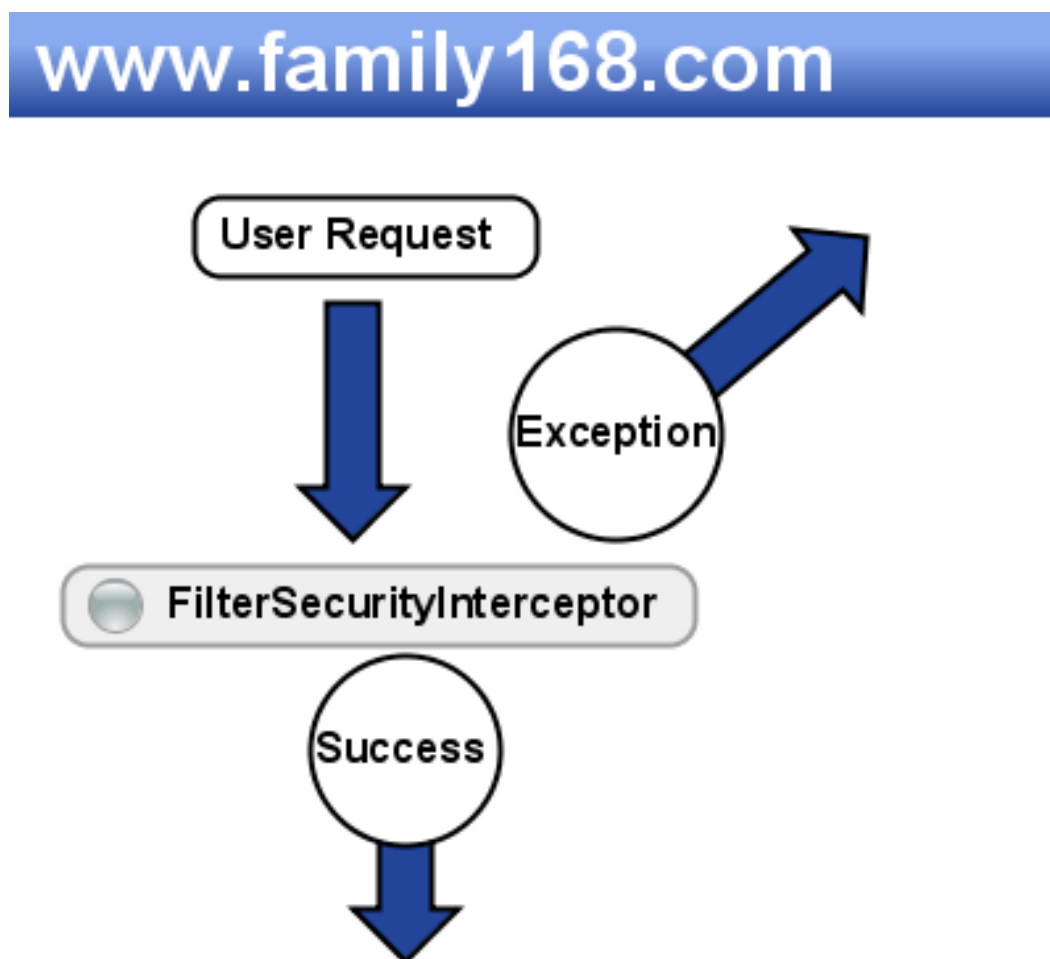


图 7.12. org.springframework.security.intercept.web.FilterSecurityInterceptor

用户的权限控制都包含在这个过滤器中。

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

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

功能三：如果用户已登录，也具有访问当前资源的权限，则放行。

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

[上一页](#)

部分 II. 保护web篇

[上一级](#)

[起始页](#)

[下一页](#)

部分 III. 保护method篇

部分 III. 保护method篇

Spring Security可以使用AOP的方式控制对某个方法的调用权限，这里有三种方式可以选择：

- global-method-security使用通配符统一配置方法的调用权限。
- protect控制某个bean内某个方法的调用权限。
- 使用Annotation在代码里配置某个方法的调用权限。

[上一页](#)

[下一页](#)

部分 IV. ACL篇

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

[上一页](#)

[下一页](#)

部分 III. 保护method篇

[起始页](#)

部分 V. 最佳实践篇

部分 V. 最佳实践篇

实现RBAC模型。

附录 A. 修改日志

修订历史

修订 0.0.2 2009-06-09

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

修订 0.0.1 2009-05-26

1. 初稿完成。[序言](#)

附录 B. 常见问题解答

B.1. Q: 如何获得源代码

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

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

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

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

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

B.4. Q: 如何实现国际化。

A: 在xml中添加如下配置：

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

B.5. Q: 如何监听Spring Security的事件日志。

A: 在xml中添加如下配置：

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

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

B.6. Q: 如何启用group。

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

附录 C. Spring Security-3.0.0.M1

[上一页](#)

附录 C. Spring Security-3.0.0.M1

Spring Security于2009-05-27发布。

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

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

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

实例在x01下。

[上一页](#)

附录 B. 常见问题解答

[起始页](#)