1. Shiro同Struts2和Spring集成

本章主要介绍Shiro同Struts、Spring框架的集成。重点把握：

* Spring初始化Shiro安全框架
* Spring管理Struts的Action
* Web中配置Struts和Shiro
* Shiro同struts的关系

目录

[12 Shiro同Struts2和Spring集成 1](#_Toc495660159)

[12.1 集成思路 2](#_Toc495660160)

[12.2 实现Shiro Struts和Spring的集成 2](#_Toc495660161)

[12.2.1 创建基于Maven的web工程 2](#_Toc495660162)

[12.2.2 编写基于Struts+Spring的测试应用 6](#_Toc495660163)

[12.2.3 编写Shiro安全框架的自定义组件 11](#_Toc495660164)

[12.2.4 编写初始化Shiro安全框架对象的代码 14](#_Toc495660165)

[12.2.5 在Struts+Spring的测试应用上添加Shiro 20](#_Toc495660166)

[12.2.6 发布测试应用 21](#_Toc495660167)

[12.3 总结 21](#_Toc495660168)

前面章节我们分别介绍了在Shiro同纯Web应用的集成和Shiro同Spring框架的集成。这章在我们继续介绍Shiro怎么样同Struts+Spring框架的集成。通过本章的研究，我们基本上可以理解和把握怎样将Shiro安全框架应用到基于流行框架开发的Web应用中。

* 1. 集成思路

Shiro安全框架同Struts+Spring框架的集成，主要实现如下几个方面的目标：

1. 使用Spring初始化Shiro安全框架的相关对象
2. 定义Shiro安全框架的入口Filter，拦截需要安全控制的Web请求，包括进入Struts框架的请求
3. 确保Shiro安全框架入口后面，使用的是Spring初始化的对象

前面章节我们都有独立介绍怎么去实现，接下来我们会结合代码一步步的介绍。考虑到进行身份验证和授权控制的时候，都需要基于应用的数据（配置文件、数据库、ActivityDirectory或第三方应用），为了避免复杂化，专注体系Shiro同Struts+Spring的集成，这里我们将在代码中模拟读取应用的数据。

* 1. 实现Shiro Struts和Spring的集成

实现Shiro同Struts+Spring的集成，主要有如下几个步骤：

1. 创建Web工程，添加Shiro、Struts、Spring的相关依赖，添加web应用的打包插件和tomcat发布插件
2. 编写配置基于Struts+Spring的测试应用
3. 编写Shiro安全框架的Realm组件和RolePermissionResolver类，实现身份验证和授权控制功能
4. 编写初始化Shiro安全框架对象初始化代码，基于Spring完成对Shiro安全框架相关对象的创建和装配
5. 配置Shiro安全框架
6. 发布测试
   * 1. 创建基于Maven的web工程

基于Maven创建Web工程的具体过程，这里就不详细介绍了，请参考Maven相关资料，这里主要介绍pom.xml中，关于web、Shiro、Struts和Spring相关的依赖和插件，详细内容可以查看随书代码[code/ShiroTP12/pom.xml](../code/ShiroTP12/pom.xml)。

1. Web相关依赖

Web相关的依赖主要是JSP Servlet的依赖，还有jstl依赖。样例代码块如下：

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>javax.servlet-api</artifactId>

<version>3.0.1</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>javax.servlet.jsp</groupId>

<artifactId>jsp-api</artifactId>

<version>2.1</version>

<scope>provided</scope>

</dependency>

<!-- https://mvnrepository.com/artifact/javax.servlet/jstl -->

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>jstl</artifactId>

<version>1.2</version>

</dependency>

1. Shiro相关依赖

Shiro相关的依赖，主要包括shiro-core、shiro-web、shiro-cas、shiro-ehcache等shiro自己的依赖，还包括shiro-spring、ehcache同spring和缓存集成依赖。其中shiro-core依赖配置在ShiroPOM父工程中，详细内容请查看随书代码code/  
ShiroPOM/pom.xml。这里是通过parent元素继承下来的。样例代码块如下：

<parent>

<groupId>ShiroBook</groupId>

<artifactId>ShiroPOM</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

其他依赖的配置样例代码块如下：

<dependency>

<groupId>org.apache.shiro</groupId>

<artifactId>shiro-web</artifactId>

<version>${shiro.version}</version>

</dependency>

<dependency>

<groupId>org.apache.shiro</groupId>

<artifactId>shiro-cas</artifactId>

<version>${shiro.version}</version>

</dependency>

<dependency>

<groupId>org.apache.shiro</groupId>

<artifactId>shiro-spring</artifactId>

<version>${shiro.version}</version>

</dependency>

<dependency>

<groupId>org.apache.shiro</groupId>

<artifactId>shiro-ehcache</artifactId>

<version>${shiro.version}</version>

</dependency>

<!-- ehcache -->

<dependency>

<groupId>net.sf.ehcache</groupId>

<artifactId>ehcache-core</artifactId>

<version>2.5.3</version>

</dependency>

1. Struts相关依赖

Struts相关的依赖包只有三个，它们是struts-core、struts-spring-plugin和struts2-  
json-plugin。样例代码块如下：

<!-- spring核心包 -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

<version>${project.build.spring.version}</version>

</dependency>

<!-- shiro end -->

<!-- struts2 begin -->

<!-- https://mvnrepository.com/artifact/org.apache.struts/struts2-core -->

<dependency>

<groupId>org.apache.struts</groupId>

<artifactId>struts2-core</artifactId>

<version>${project.build.struts.version}</version>

</dependency>

<!-- https://mvnrepository.com/artifact/org.apache.struts/struts2-spring-plugin -->

<dependency>

<groupId>org.apache.struts</groupId>

<artifactId>struts2-spring-plugin</artifactId>

<version>${project.build.struts.version}</version>

</dependency>

<!-- https://mvnrepository.com/artifact/org.apache.struts/struts2-json-plugin -->

<dependency>

<groupId>org.apache.struts</groupId>

<artifactId>struts2-json-plugin</artifactId>

<version>${project.build.struts.version}</version>

</dependency>

1. Spring相关依赖

Spring相关的依赖包包括spring-aop、spring-beans、spring-web、spring-aspects、spring-context、spring-context-support，样例代码块如下：

<!-- Spring begin -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-aop</artifactId>

<version>${project.build.spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-beans</artifactId>

<version>${project.build.spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-web</artifactId>

<version>${project.build.spring.version}</version>

</dependency>

<!-- https://mvnrepository.com/artifact/org.springframework/spring-aspects -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-aspects</artifactId>

<version>${project.build.spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>${project.build.spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context-support</artifactId>

<version>${project.build.spring.version}</version>

</dependency>

1. 插件

我们这里主要用到了maven-compiler-plugin、maven-war-plugin和tomcat7-maven-plugin插件，样例代码块如下：

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<configuration>

<source>1.7</source>

<target>1.7</target>

</configuration>

</plugin>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-war-plugin</artifactId>

<configuration>

<encoding>${project.build.sourceEncoding}</encoding>

<warName>ShiroTP12</warName>

<webappDirectory>${project.build.directory}/ShiroTP12</webappDirectory>

<warSourceDirectory>src/main/webapp</warSourceDirectory>

</configuration>

</plugin>

<plugin>

<groupId>org.apache.tomcat.maven</groupId>

<artifactId>tomcat7-maven-plugin</artifactId>

<configuration>

<charset>${project.build.sourceEncoding}</charset>

<url>http://localhost:9090/manager/text</url>

<server>tomcat7</server>

<path>/ShiroTP12</path>

<port>9090</port>

<warFile>${project.build.directory}/ShiroTP12.war</warFile>

<warSourceDirectory>src/main/webapp</warSourceDirectory>

<uriEncoding>${project.build.sourceEncoding}</uriEncoding>

</configuration>

</plugin>

* + 1. 编写基于Struts+Spring的测试应用

因为是测试，我们这里编写两个Action，UserAction模拟响应维护用户的CRUD请求，OrderAction模拟响应订单的CRUD请求。

另外准备几个jsp：login.jsp实现登录页面；adminManager.jsp体现管理员页面；success.jsp显示登录成功页面;welcome.jsp显示导航页面；unauthorized.jsp显示没有权限页面。

* 编写UserAction

编写UserAction类，里面包含add、edit、delete和query四个方法，分别模拟响应对用户的CRUD，样例代码如下，详细内容请查看随书代码[code/ShiroTP12  
/src/main/java/cn/com/shiro/book/tp12/action/UserAction.java](../code/ShiroTP12/src/main/java/cn/com/shiro/book/tp12/action/UserAction.java)。

package cn.com.shiro.book.tp12.action;

@Controller("userAction")

public class UserAction {

public String add(){

String result = "successAdd";

System.out.println("执行添加用户功能");

return result;

}

public String edit(){

String result = "successEdit";

System.out.println("执行修改用户功能");

return result;

}

public String delete(){

String result = "successDelete";

System.out.println("执行删除用户功能");

return result;

}

public String query(){

String result = "successQuery";

System.out.println("执行查询用户功能");

return result;

}

}

@Controller需要加上，表示当前类是控制器类，由Spring容器加载并初始化到IOC容器中。

* 编写OrderAction

类似UserAction，模拟实现对订单的CRUD的处理，样例代码如下，详细内容请查看随书代码[code/ShiroTP12/src/main/java/cn/com/shiro/book/tp12/action/User  
Action.java](file:///C:\work\课件\shiro\shiro应用实战\code\ShiroTP12\src\main\java\cn\com\shiro\book\tp12\action\UserAction.java)。

package cn.com.shiro.book.tp12.action;

@Controller("orderAction")

public class OrderAction {

public String add(){

String result = "successAdd";

System.out.println("执行添加订单功能");

return result;

}

public String edit(){

String result = "successEdit";

System.out.println("执行修改订单功能");

return result;

}

public String delete(){

String result = "successDelete";

System.out.println("执行删除订单功能");

return result;

}

public String query(){

String result = "successQuery";

System.out.println("执行查询订单功能");

return result;

}

}

* 编写login.jsp

在webapp目录下创建login.jsp代码，显示登录表单，样例代码如下，详细内容请查看随书代码[code/ShiroTP12/src/main/webapp/login.jsp](../code/ShiroTP12/src/main/webapp/login.jsp)。

… …

<body>

<h1>我自己的登录页面</h1>

<form method="post">

UserName:<input type='text' name="userName"/><br/>

Password:<input type='password' name="password"/><br/>

<input type="submit" value="登录"/>

</form>

</body>

… …

注意上面代码的form，是没有设置action属性值的，意味着是默认提交给自己login.jsp。另外注意表单的用户名和密码的元素名称分别是userName和password，后面的shiro安全框架集成登陆的时候会用到。

* 编写adminManager.jsp

在webapp目录下，创建adminManager.jsp页面，只是显示管理员页面文字，样例代码块如下，详细内容请查看随书代码[code/ShiroTP12/src/main/webapp/  
adminManager.jsp](file:///C:\work\课件\shiro\shiro应用实战\code\ShiroTP12\src\main\webapp\login.jsp)。

<%@taglib uri="http://shiro.apache.org/tags" prefix="shiro"%>

… …

<body>

<shiro:principal/>你好，恭喜成功进入本系统！<br>

您可以操作管理员页面，<a href="welcome.jsp">点击我进入欢迎界面</a>&nbsp;&nbsp; <a href="logout">退出点这里</a><br/>

</body>

</html>

<shiro:principal>标签的作用是获取当前Subject的身份标识(用户名)

* 编写success.jsp

在webapp目录下创建success.jsp，登录成功后转向到该页面，样例代码如下，详细内容请查看随书代码[code/ShiroTP12/src/main/webapp/success.jsp](../code/ShiroTP12/src/main/webapp/success.jsp)。

… …

<body>

测试成功页面^\_^

<a href="welcome.jsp">点击我进入欢迎界面</a>

</body>

… …

* 编写welcome.jsp

在webapp目录下创建welcome.jsp页面，该页面模拟实现web应用导航，样例代码如下，详细内容请查看随书代码[code/ShiroTP12/src/main/webapp/welcome  
.jsp](../code/ShiroTP12/src/main/webapp/welcome.jsp)。

… …

<body>

<shiro:principal/>你好，恭喜成功进入本系统！<br>

您可以进行如下操作，请点击它们进入对应操作页面 <a href="logout">退出点这里</a> &nbsp;&nbsp;<a href="adminManager.jsp">点这里进入管理员界面</a><br/>

<a href="demo/userAction!add.action">添加用户</a><br/>

<a href="demo/userAction!edit.action">修改用户</a><br/>

<a href="demo/userAction!delete.action">删除用户</a><br/>

<a href="demo/userAction!query.action">查询用户</a><br/>

<br/>

<a href="demo/orderAction!add.action">添加订单</a><br/>

<a href="demo/orderAction!edit.action">修改订单</a><br/>

<a href="demo/orderAction!delete.action">删除订单</a><br/>

<a href="demo/orderAction!query.action">查询订单</a><br/>

</body>

… …

* 编写unauthorized.jsp

在webapp目录下创建unauthorized.jsp，显示权限限制信息，样例代码块如下，详细内容请查看随书代码[code/ShiroTP12/src/main/webapp/unauthorized.jsp](../code/ShiroTP12/src/main/webapp/unauthorized.jsp)。

… …

<body>

sorry <shiro:principal/>,您没有权限访问该页面<br/>

<a href="welcome.jsp">请点击这里返回到导航页</a>，继续其他操作，谢谢！

</body>

… …

* 编写struts.xml

在resources目录下，创建struts.xml文件。struts.xml是Struts2的配置文件，里面描述Struts2运行中必须初始化的相关信息。当前struts.xml主要描述了三种信息：

1. 包含的xml

样例代码块如下：

<include file="struts-default.xml"></include>

<include file="struts-plugin.xml"></include>

Struts-default.xml是struts自带的基本配置文件，包含在struts-core.jar中

Struts-plugin.xml是struts-json-plugin的配置文件，在struts中提供json支持，包含在struts-json-plugin.jar中。

1. 基于Spring初始化Action对象的常量

<constant name="struts.objectFactory" value="spring"></constant>

标明struts在需要action对象的时候，优先从初始化好的WebApplicationContext容器中获取。

1. 包含action的package

<package name="demo" extends="struts-default" namespace="/demo">

<action name="userAction" class="userAction">

<result name="successAdd">/success.jsp</result>

<result name="successEdit">/success.jsp</result>

<result name="successDelete">/success.jsp</result>

<result name="successQuery">/success.jsp</result>

</action>

<action name="orderAction" class="orderAction">

<result name="successAdd">/success.jsp</result>

<result name="successEdit">/success.jsp</result>

<result name="successDelete">/success.jsp</result>

<result name="successQuery">/success.jsp</result>

</action>

</package>

如上代码描述了demo包的namespace是/demo和包含的userAction和orderAction，以及每个action的四个不同的result。其中namespace对应运行时的虚拟路径。关于struts.xml的配置详细情况，请参考Struts2的相关文档。

Struts.xml的详细内容，请查看随书代码[code/ShiroTP12/src/main/resources/  
struts.xml](../code/ShiroTP12/src/main/resources/struts.xml)。

* 编写applicationContext.xml

在resources目录下，创建协助spring容器初始化applicationContext.xml，指定Spring容器初始化时需要自动扫描的包，样例代码如下，详细内容请查看随书代码code/ShiroTP12/src/main/resources/applicationContext.xml。关于Spring配置文件的详细内容，请查看Spring4相关文档。

<context:component-scan base-package="cn.com.shiro.book"></context:component-scan>

如上配置指定Spring在初始化的时候，自动扫描cn.com.shiro.book包和子包下所有类的注解。

* 配置web.xml

在web.xml中，我们需要配置两个方面的内容，完整的web.xml信息，请查看随书代码[code/ShiroTP12/src/main/webapp/WEB-INF/web.xml](../code/ShiroTP12/src/main/webapp/WEB-INF/web.xml)：

1. 自动初始化Spring容器的ContextLoadListener监听器

<!-- 配置上下文监听器，启动IOC容器WebApplicationContext -->

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>classpath:applicationContext.xml</param-value>

</context-param>

<listener>

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

</listener>

如上代码，通过context-param指定ContextLoadListener初始化Spring容器时需要读取的配置文件，param-name为参数名称，它的值contextConfigLocation是固定的，param-value是参数值，是配置文件的路径和文件名称。

Listener元素中的listener-class指定Spring自带的，用来初始化容器的上下文监听器类。

1. Struts2的入口过滤器

<filter>

<filter-name>struts2</filter-name>

<filter-class>org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter</filter-class>

</filter>

<filter-mapping>

<filter-name>struts2</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

如上代码通过filter注册struts2的入口过滤器，fitler-mapping指定过滤所有请求。

* + 1. 编写Shiro安全框架的自定义组件

关于Shiro安全框架的组件，我们这里主要实现两个，一个是RolePermissionResolver，一个是自定义ShiroRealm。

1）RolePermissionResolver组件的作用是实现角色到权限的转化，也就是说，根据角色，获取它对应的所有的权限。转化逻辑封装在resolvePermissionsInRole方法中，该方法会被Shiro安全框架自动调用，以便基于perms拦截器或我们自定义的权限拦截器进行授权控制。样例代码块如下，详细内容请查看随书代码[code/ShiroTP12/src/main/java/cn/com/shiro/book/tp12/authz/permission/RolePermissionResolver.java](../code/ShiroTP12/src/main/java/cn/com/shiro/book/tp12/authz/permission/ShiroRolePermissionResovler.java)。

… …

public class ShiroRolePermissionResovler implements RolePermissionResolver {

private Map<String, List<String>> rolePermsMap = new HashMap<String, List<String>>();

{

// 初始化角色权限信息

rolePermsMap.put("admin", Arrays.asList("\*"));

rolePermsMap.put("role1",

Arrays.asList("user:add", "user:edit", "user:delete"));

rolePermsMap.put("role2",

Arrays.asList("order:add", "order:edit", "order:delete"));

rolePermsMap.put("role3", Arrays.asList("user:query", "order:query"));

}

@Override

public Collection<Permission> resolvePermissionsInRole(String roleString) {

Collection<Permission> permissions = new HashSet<Permission>();

List<String> permStrs = this.rolePermsMap.get(roleString);

if(permStrs != null){

for(String permStr:permStrs){

permissions.add((Permission)new WildcardPermission(permStr));

}

}

return permissions;

}

}

如上代码，我们使用代码块初始化rolePermsMap，里面设置每个角色所包含的权限，在resolvePermissionsInRole方法中，根据转入的角色参数，找到对应权限集合返回。

2）ShiroRealm组件，主要通过重写doGetAuthenticationInfo和doGetAuthorizationInfo两个方法，分别实现身份认证和授权功能，样例代码块如下，详细内容请查看随书代码[code/ShiroTP12/src/main/java/cn/com/shiro/book/tp12/realm/RolePermission  
Resolver.java](../code/ShiroTP12/src/main/java/cn/com/shiro/book/tp12/realm/ShiroRealm.java)。

package cn.com.shiro.book.tp12.realm;

… …

public class ShiroRealm extends AuthorizingRealm {

**private PasswordService passwordService;①**

**private Map<String, String> accountMap = new HashMap<String, String>();②**

private Map<String, List<String>> userRolesMap = new HashMap<String, List<String>>();

private Map<String, List<String>> userPermsMap = new HashMap<String, List<String>>();

// 模拟数据源初始化数据

**{③**

// 初始化用户角色信息

userRolesMap.put("zhangsan", Arrays.asList("admin"));

userRolesMap.put("lisi", Arrays.asList("role1", "role2"));

userRolesMap.put("wangwu", Arrays.asList("role3"));

// 初始化用户权限信息

userPermsMap.put("wangwu", Arrays.asList("order:add", "order:delete"));

}

/\*\*

\* 获取指定principals包含的角色和权限信息

\* \*/

**@Override④**

protected AuthorizationInfo doGetAuthorizationInfo(

PrincipalCollection principals) {

SimpleAuthorizationInfo info = new SimpleAuthorizationInfo();

List<String> roles = this.userRolesMap.get(principals

.getPrimaryPrincipal());

// 添加角色

if (roles != null) {

info.addRoles(roles);

}

// 添加用户权限

if (this.userPermsMap.get(principals.getPrimaryPrincipal()) != null) {

info.addStringPermissions(this.userPermsMap.get(principals

.getPrimaryPrincipal()));

}

return info;

}

/\*\*

\* 根据token的principals返回从数据源中获取的对应AuthenticationInfo

\* \*/

**@Override⑤**

protected AuthenticationInfo doGetAuthenticationInfo(

AuthenticationToken token) throws AuthenticationException {

SimpleAuthenticationInfo info = null;

String password = this.accountMap.get(token.getPrincipal().toString());

if (password != null) {

info = new SimpleAuthenticationInfo(token.getPrincipal(), password,

this.getName());

}

return info;

}

// 使用passwordService初始化账号信息，密码加密

**private void initAccounts() {⑥**

// 初始化用户账号信息

accountMap.put("zhangsan",

this.passwordService.encryptPassword("abc123"));

accountMap.put("lisi", this.passwordService.encryptPassword("123456"));

accountMap

.put("wangwu", this.passwordService.encryptPassword("123456"));

}

**public void setPasswordService(PasswordService passwordService) {⑦**

this.passwordService = passwordService;

initAccounts();

}

}

如上代码

1. 定一个PasswordService成员变量，该成员变量的值，需要在Shiro安全框架初始化的时候注入，用来模拟实现应用中加密密码，具体使用请查看后面的initAccount方法代码。在正式应用中，该部分逻辑需要放在注册用户功能里面，对保存的密码进行加密。
2. 声明了三个Map成员变量，分别用来存放用户名密码信息、用户角色信息和用户权限信息。这三个Map的值，在ShiroRealm被运用到安全框架之前自动被初始化，用来模拟Shiro安全框架底层依赖的数据源。
3. 代码块中初始化用户角色信息和用户权限信息，这里一共有zhangsan、lisi、wangwu三个用户，各自有自己的角色，只有wangwu单独有自己的权限。
4. doGetAuthorizationInfo方法，给Shiro安全框架提供授权依赖数据，该方法返回AuthorizationInfo对象，里面包含了当前用户拥有的角色信息和权限信息（通过principals信息从用户角色信息Map和用户权限信息Map中获取设置）
5. doGetAuthenticationInfo方法，给Shiro安全框架提供身份认证的数据，该方法返回AuthenticationInfo对象，里面包含有当前用户在注册时输入的密码信息（通过principals从用户密码信息Map中获取设置）
6. initAccounts方法初始化用户密码信息Map，在用户密码信息Map中设置了zhangsan、lisi、wangsu三个用户和对应的加密密码。密码加密使用的是PasswordService的encryptPassword方法。
7. setPasswordService方法，提供了一个注入PasswordService实例的入口，注入成功后，调用initAccounts方法，基于注入的passwordService初始化用户密码信息Map
   * 1. 编写初始化Shiro安全框架对象的代码

创建ShiroWebConfiguration.java类，样例代码块如下

package cn.com.shiro.book.tp12.config;

… …

// 支持注解

@Configuration

// 注解扫描包

@ComponentScan("cn.com.shiro.book")

public class ShiroWebConfiguration {

… …

}

@Configuration提供给Spring容器，表示支持使用注解的方式声明要初始化的beean

@ComponentScan提供给Spring容器，表示要自动扫描cn.com.shiro.book包和子包下的所有的类。

在ShiroWebConfiguration类中，使用编码的方式初始化Shiro安全框架的对象，并且装配各个对象之间的依赖（以前是用配置文件描述，Shiro安全框架自己根据配置文件创建并装配对象的）。主要内容是创建SecurityManager对象；设置SecurityManager对象必要的authorizer和realms属性；将SecurityManager对象注册成全局对象；创建ShiroFilterFactoryBean对象；设置ShiroFilterFactoryBean对象的filterChainDefinition；初始化ShiroFilterFactoryBean的拦截器等，下面逐个介绍各个对象和属性装配的代码，完整的详细代码请查看随书代码[code/ShiroTP12/src/main/  
java/cn/com/shiro/book/tp12/config/ShiroWebConfiguration.java](../code/ShiroTP12/src/main/java/cn/com/shiro/book/tp12/config/ShiroWebConfiguration.java)。

1. 创建PasswordService对象

PasswordService对象有两个用处，一个是给PasswordMatcher对Subject输入的明文密码加密，以便同应用中加密的密码进行对比；一个是在用户注册的时候，给注册用户的密码进行加密，以便以密码的形式保存。因为我们测试案例暂时没有注册功能，所以第二个用处是给ShiroRealm初始化用户密码信息Map的时候，对密码进行加密。

创建PasswordService对象的过程分两步进行，第一步创建PasswordService所依赖的HashService、HashFormat和HashFormatFactory对象，第二步是创建PasswordService对象，并且设置相关的依赖。具体代码如下：

1. 创建PasswordService相关的依赖对象

@Bean(name = "hashFormatFactory")

public HashFormatFactory createHashFormatFactory() {

return new DefaultHashFormatFactory();

}

@Bean(name = "hashFormat")

public HashFormat createHashFormat() {

return new Shiro1CryptFormat();

}

@Bean(name = "hashService")

public HashService createHashService() {

HashService hashService = new DefaultHashService();

return hashService;

}

如上代码每个方法前面都有@Bean(name=...)，作用是告诉Spring容器，需要将该方法返回的对象做bean，在容器中进行管理，name后面的值，是该对象在容器中的name属性值，是bean对象的唯一标识。以后的代码也是同样的作用。

1. 创建PasswordService和装备依赖

@Bean(name = "passwordService")

public PasswordService createPasswordService() {

// 创建passwordService对象

DefaultPasswordService passwordService = new DefaultPasswordService();

// 设置hashFormat属性

passwordService.setHashFormat(this.createHashFormat());

// 设置hashFormatFactory属性

passwordService.setHashFormatFactory(this.createHashFormatFactory());

// 设置hashService属性

passwordService.setHashService(this.createHashService());

return passwordService;

}

1. 创建PasswordMatcher对象

PasswordMatcher对象将注入到ShiroRealm的credentialsMatcher属性中，用来实现Subject传入的密码同应用中密码的比对，实现身份确认。因为需要对密码加密后比对，所以需要注入同对应用进行加密相同的PasswordService，以便用同样的加密算法对Subject传入的密码加密。样例代码如下：

@Bean(name = "passwordMatcher")

public PasswordMatcher createPasswordMatcher() {

PasswordMatcher passwordMatcher = new PasswordMatcher();

passwordMatcher.setPasswordService(this.createPasswordService());

return passwordMatcher;

}

1. 创建PermissionResolver对象

PermissionResolver对象的作用是将Ant风格的权限字符串解析成WildcardPermission对象，以便Shiro安全框架完成权限比对授权。样例代码如下：

@Bean(name = "permResolver")

public PermissionResolver createPermissionResolver() {

PermissionResolver permResolver = new WildcardPermissionResolver();

return permResolver;

}

1. 创建RolePermissionResolver对象

RolePermissionResolver对象的作用是获取指定角色的所有权限，以便判断角色对应的权限，完成授权控制，样例代码如下：

@Bean(name = "rolePermissionResolver")

public RolePermissionResolver createRolePermissionResolver() {

RolePermissionResolver rolePermResolver = new ShiroRolePermissionResovler();

return rolePermResolver;

}

1. 创建ModularRealmAuthorizer对象

ModularRealmAuthorizer是基于Realm的授权器，SecurityManager在授权控制的时候，会将授权控制工作委托给它管理的Authorizer对象（我们使用ModularRealmAuthorizer）。需要给ModularRealmAuthorizer对象装配permissionResolver和rolePermissionResolver，以便完成权限的比对授权。同时ModularRealmAuthorizer自动会从SecurityManager中注册的realms中，判断哪些Realm对象实现了Authorizer接口，从而调用它们实现授权控制。

创建ModularRealmAuthorizer对象的样例代码块如下：

@Bean(name = "authorizer")

public ModularRealmAuthorizer createModularRealmAuthorizer() {

ModularRealmAuthorizer authorizer = new ModularRealmAuthorizer();

authorizer.setPermissionResolver(this.createPermissionResolver());

authorizer.setRolePermissionResolver(this

.createRolePermissionResolver());

return authorizer;

}

1. 创建ShiroRealm对象

ShiroRealm是我们自定的Realm组件，里面实现了身份认证和授权逻辑，同时也模拟实现了应用数据的初始化，样例代码如下：

@Bean(name = "shiroRealm")

public Realm createRealm() {

ShiroRealm realm = new ShiroRealm();

realm.setPasswordService(this.createPasswordService());

realm.setCredentialsMatcher(this.createPasswordMatcher());

return realm;

}

1. 创建SecurityManager对象

SecurityManager对象是Shiro安全框架的管理对象，因为Shiro是要使用在Web环境下的，所以需要创建DefaultWebSecurityManager对象，同时装配authorizer和realms依赖。样例代码如下：

@Bean(name = "securityManager")

public org.apache.shiro.mgt.SecurityManager createSecurityManager() {

DefaultWebSecurityManager securityManager = new DefaultWebSecurityManager();

securityManager.setAuthorizer(this.createModularRealmAuthorizer());

securityManager.setRealms(Arrays.asList(this.createRealm()));

return securityManager;

}

1. 注册全局SecurityManager

SecurityManager在一个应用中应该是全局的，只能有一个对象，所以我们创建MethodInvokingFactoryBean对象，调用SecurityUtils的setSecurityManager方法完成全局注册。样例代码如下：

@Bean

public MethodInvokingFactoryBean createMethodInvokingFactoryBean() {

MethodInvokingFactoryBean factoryBean = new MethodInvokingFactoryBean();

factoryBean

.setStaticMethod("org.apache.shiro.SecurityUtils.setSecurityManager");

factoryBean.setArguments(new Object[] { this.createSecurityManager() });

return factoryBean;

}

1. 创建ShiroFilterFactoryBean对象

ShiroFilterFactoryBean对象的作用是为Shiro安全框架在web应用中创建一个ShiroFilter拦截器，提供Shiro安全框架的总的拦截入口，实现身份认证和授权控制。需要给创建的ShiroFilterFactoryBean装配好securityManager、filterChainDefinitions依赖，同时要初始化需要用到的其他拦截器，比如anon、perms、logout等拦截器。为了方便理解，创建和装配ShiroFilterFactoryBean对象分如下三步进行：

1. 初始化ShiroFilterFactoryBean的拦截器

在本案例中，我们用到了Shiro安全框架自带的authz、logout、authzBasic和perms拦截器，对这些拦截器的初始化过程都是一样的：先从ShiroFilterFactoryBean中获取拦截器，如果没有获取到，就实例化一个对象，并且添加到ShiroFilterFactoryBean中；接着初始化该拦截器的相关属性。样例代码如下：

// 初始化authc、logout、roles、perms和authcBasic拦截器

private void setFilterAttributes(ShiroFilterFactoryBean shiroFilter) {

// 获取ShiroFilterFactoryBean中拦截器集合

Map<String, Filter> filters = shiroFilter.getFilters();

// 从filters中获取authc拦截器，并且进行初始化

FormAuthenticationFilter authc = (FormAuthenticationFilter) filters

.get("authc");

if (authc == null) {

// 创建authc拦截器

authc = new FormAuthenticationFilter();

// 在filters中添加authc拦截器

filters.put("authc", authc);

}

// 设置身份认证成功后的跳转url

authc.setSuccessUrl("/welcome.jsp");

// 设置登录url

authc.setLoginUrl("/login.jsp");

// 设置登录页面中用户名参数名称

authc.setUsernameParam("userName");

// 设置登录页面中密码参数名称

authc.setPasswordParam("password");

LogoutFilter logout = (LogoutFilter) filters.get("logout");

if (logout == null) {

logout = new LogoutFilter();

filters.put("logout", logout);

}

// 设置退出登录跳转url

logout.setRedirectUrl("/login.jsp");

RolesAuthorizationFilter roles = (RolesAuthorizationFilter) filters

.get("roles");

if (roles == null) {

roles = new RolesAuthorizationFilter();

filters.put("roles", roles);

}

// 设置角色授权失败跳转url

roles.setUnauthorizedUrl("/unauthorized.jsp");

PermissionsAuthorizationFilter perms = (PermissionsAuthorizationFilter) filters

.get("perms");

if (perms == null) {

perms = new PermissionsAuthorizationFilter();

filters.put("perms", perms);

}

// 设置权限授权失败跳转url

perms.setUnauthorizedUrl("/unauthorized.jsp");

BasicHttpAuthenticationFilter authcBasic = (BasicHttpAuthenticationFilter) filters

.get("authcBasic");

if (authcBasic == null) {

authcBasic = new BasicHttpAuthenticationFilter();

filters.put("authcBasic", authcBasic);

}

// 设置身份认证成功跳转url

authcBasic.setSuccessUrl("/welcome.jsp");

// 设置应用名称

authcBasic.setApplicationName("ShiroBook 登录");

}

1. 生成FilterChainDefinitions字符串

以字符串的形式，描述每个拦截的url和对应的拦截器，格式同以前的ini配置文件中[urls]部分一样。注意每个url描述完后，需要加入换行符，以便Shiro安全框架解析。样例代码如下：

private String getChainDefinitions(){

StringBuilder builder = new StringBuilder();

builder.append("/login.jsp=authc\r\n");

builder.append("/logout=logout\r\n");

builder.append("/adminManager.jsp=authcBasic,roles[admin]\r\n");

builder.append("/demo/userAction!add.action=perms[user:add]\r\n");

builder.append("/demo/userAction!edit.action=perms[user:edit]\r\n");

builder.append("/demo/userAction!delete.action=perms[user:delete]\r\n");

builder.append("/demo/userAction!query.action=perms[user:query]\r\n");

builder.append("/demo/orderAction!add.action=perms[order:add]\r\n");

builder.append("/demo/orderAction!edit.action=perms[order:edit]\r\n");

builder.append("/demo/orderAction!delete.action=perms[order:delete]\r\n");

builder.append("/demo/orderAction!query.action=perms[order:query]\r\n");

builder.append("/demo/\*\*=authcBasic\r\n");

builder.append("/welcome.jsp=authcBasic\r\n");

builder.append("/\*\*=anon\r\n");

return builder.toString();

}

1. 创建装配ShiroFilterFactoryBean

创建和装配ShiroFilterFactoryBean，样例代码如下：

@Bean(name = "shiroFilter")

public ShiroFilterFactoryBean createShiroFilterFactoryBean() {

System.out

.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*create shiroFilter \*\*\*\*\*\*\*\*\*\*\*\*\*");

ShiroFilterFactoryBean shiroFilter = new ShiroFilterFactoryBean();

// 设置安全管理器

shiroFilter.setSecurityManager(this.createSecurityManager());

// 设置url同filter拦截器的关系

shiroFilter.setFilterChainDefinitions( this.getChainDefinitions());

// 初始化filter拦截器

this.setFilterAttributes(shiroFilter);

return shiroFilter;

}

注意这里ShiroFilterFactoryBean对象在@Bean中的name值是shiroFilter。这个名称可以自定义，但是需要同web.xml中配置的DelegatingFilterProxy过滤名称要一致，因为DelegatingFilterProxy是根据这名称从Spring的WebApplicationContext容器中找ShiroFilter对象的。

1. 创建Shiro生命周期处理器对象

创建LifecycleBeanPostProcessor对象，管理Shiro安全框架中每个对象的生命周期。样例代码如下：

// shiro生命周期处理器

@Bean(name = "lifecycleBeanPostProcessor")

public LifecycleBeanPostProcessor createLifecycleBeanPostProcessor() {

return new LifecycleBeanPostProcessor();

}

* + 1. 在Struts+Spring的测试应用上添加Shiro

在web.xml中添加DelegatingFilterProxy过滤器，并定义它的filter-mapping，实现Shiro和Web的集成。注意，DelegatingFilterProxy的filter-mapping需要定义在Struts2的filter-mapping之前，这样确保Shiro安全框架先做安全控制，再进入Struts2框架。样例代码如下

<!-- The filter-name matches name of a 'shiroFilter' bean inside applicationContext.xml -->

<filter>

<filter-name>**shiroFilter**</filter-name>

<filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>

<init-param>

<param-name>targetFilterLifecycle</param-name>

<param-value>true</param-value>

</init-param>

</filter>

<filter-mapping>

<filter-name>shiroFilter</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

再次强调，DelegatingFilterProxy的filter-name，需要同ShiroWebConfiguration中ShiroFilterFactoryBean的名称一致。

* + 1. 发布测试应用

基于Tomcat7插件发布应用到Tomcat7服务器，启动测试，就可以具体体会Shiro安全有在Struts2前做了对应的安全控制，这里就不累赘了，请自己对比我们在ShiroRealm和定义的FilterChainDefinitions进行测试比较。

* 1. 总结

本章主要介绍了Shiro同Struts+Spring的集成步骤和注意事项，需要我们自己动手才能真正体会每个步骤。