[**第十九章 动态URL权限控制——《跟我学Shiro》**](http://jinnianshilongnian.iteye.com/blog/2040929)

**博客分类：**

* [跟我学Shiro](http://jinnianshilongnian.iteye.com/category/305053)

[跟我学Shiro](http://www.iteye.com/blogs/tag/%E8%B7%9F%E6%88%91%E5%AD%A6Shiro)

**目录贴：**[**跟我学Shiro目录贴**](http://jinnianshilongnian.iteye.com/blog/2018398)

用过Spring Security的朋友应该比较熟悉对URL进行全局的权限控制，即访问URL时进行权限匹配；如果没有权限直接跳到相应的错误页面。Shiro也支持类似的机制，不过需要稍微改造下来满足实际需求。不过在Shiro中，更多的是通过AOP进行分散的权限控制，即方法级别的；而通过URL进行权限控制是一种集中的权限控制。本章将介绍如何在Shiro中完成动态URL权限控制。

本章代码基于《第十六章 综合实例》，请先了解相关数据模型及基本流程后再学习本章。

**表及数据SQL**

请运行shiro-example-chapter19/sql/ shiro-schema.sql 表结构

请运行shiro-example-chapter19/sql/ shiro-schema.sql 数据

**实体**

具体请参考com.github.zhangkaitao.shiro.chapter19包下的实体。

**Java代码  [收藏代码](javascript:void())**

1. **public** **class** UrlFilter **implements** Serializable {
2. **private** Long id;
3. **private** String name; //url名称/描述
4. **private** String url; //地址
5. **private** String roles; //所需要的角色，可省略
6. **private** String permissions; //所需要的权限，可省略
7. }

表示拦截的URL和角色/权限之间的关系，多个角色/权限之间通过逗号分隔，此处还可以扩展其他的关系，另外可以加如available属性表示是否开启该拦截。

**DAO**

具体请参考com.github.zhangkaitao.shiro.chapter19.dao包下的DAO接口及实现。

**Service**

具体请参考com.github.zhangkaitao.shiro.chapter19.service包下的Service接口及实现。

**Java代码  [收藏代码](javascript:void())**

1. **public** **interface** UrlFilterService {
2. **public** UrlFilter createUrlFilter(UrlFilter urlFilter);
3. **public** UrlFilter updateUrlFilter(UrlFilter urlFilter);
4. **public** **void** deleteUrlFilter(Long urlFilterId);
5. **public** UrlFilter findOne(Long urlFilterId);
6. **public** List<UrlFilter> findAll();
7. }

基本的URL拦截的增删改查实现。

**Java代码  [收藏代码](javascript:void())**

1. @Service
2. **public** **class** UrlFilterServiceImpl **implements** UrlFilterService {
3. @Autowired
4. **private** ShiroFilerChainManager shiroFilerChainManager;
6. @Override
7. **public** UrlFilter createUrlFilter(UrlFilter urlFilter) {
8. urlFilterDao.createUrlFilter(urlFilter);
9. initFilterChain();
10. **return** urlFilter;
11. }
12. //其他方法请参考源码
13. @PostConstruct
14. **public** **void** initFilterChain() {
15. shiroFilerChainManager.initFilterChains(findAll());
16. }
17. }

UrlFilterServiceImpl在进行新增、修改、删除时会调用initFilterChain来重新初始化Shiro的URL拦截器链，即同步数据库中的URL拦截器定义到Shiro中。此处也要注意如果直接修改数据库是不会起作用的，因为只要调用这几个Service方法时才同步。另外当容器启动时会自动回调initFilterChain来完成容器启动后的URL拦截器的注册。

**ShiroFilerChainManager**

**Java代码  [收藏代码](javascript:void())**

1. @Service
2. **public** **class** ShiroFilerChainManager {
3. @Autowired **private** DefaultFilterChainManager filterChainManager;
4. **private** Map<String, NamedFilterList> defaultFilterChains;
5. @PostConstruct
6. **public** **void** init() {
7. defaultFilterChains =
8. **new** HashMap<String, NamedFilterList>(filterChainManager.getFilterChains());
9. }
10. **public** **void** initFilterChains(List<UrlFilter> urlFilters) {
11. //1、首先删除以前老的filter chain并注册默认的
12. filterChainManager.getFilterChains().clear();
13. **if**(defaultFilterChains != **null**) {
14. filterChainManager.getFilterChains().putAll(defaultFilterChains);
15. }
16. //2、循环URL Filter 注册filter chain
17. **for** (UrlFilter urlFilter : urlFilters) {
18. String url = urlFilter.getUrl();
19. //注册roles filter
20. **if** (!StringUtils.isEmpty(urlFilter.getRoles())) {
21. filterChainManager.addToChain(url, "roles", urlFilter.getRoles());
22. }
23. //注册perms filter
24. **if** (!StringUtils.isEmpty(urlFilter.getPermissions())) {
25. filterChainManager.addToChain(url, "perms", urlFilter.getPermissions());
26. }
27. }
28. }
29. }

1、init：Spring容器启动时会调用init方法把在spring配置文件中配置的默认拦截器保存下来，之后会自动与数据库中的配置进行合并。

2、initFilterChains：UrlFilterServiceImpl会在Spring容器启动或进行增删改UrlFilter时进行注册URL拦截器到Shiro。

拦截器及拦截器链知识请参考《第八章 拦截器机制》，此处再介绍下Shiro拦截器的流程：

AbstractShiroFilter //如ShiroFilter/ SpringShiroFilter都继承该Filter

   doFilter //Filter的doFilter

     doFilterInternal //转调doFilterInternal

       executeChain(request, response, chain) //执行拦截器链

         FilterChain chain = getExecutionChain(request, response, origChain) //使用原始拦截器链获取新的拦截器链

           chain.doFilter(request, response) //执行新组装的拦截器链

getExecutionChain(request, response, origChain) //获取拦截器链流程

       FilterChainResolver resolver = getFilterChainResolver(); //获取相应的FilterChainResolver

       FilterChain resolved = resolver.getChain(request, response, origChain); //通过FilterChainResolver根据当前请求解析到新的FilterChain拦截器链

默认情况下如使用ShiroFilterFactoryBean创建shiroFilter时，默认使用PathMatchingFilterChainResolver进行解析，而它默认是根据当前请求的URL获取相应的拦截器链，使用Ant模式进行URL匹配；默认使用DefaultFilterChainManager进行拦截器链的管理。

PathMatchingFilterChainResolver默认流程：

**Java代码  [收藏代码](javascript:void())**

1. **public** FilterChain getChain(ServletRequest request, ServletResponse response, FilterChain originalChain) {
2. //1、首先获取拦截器链管理器
3. FilterChainManager filterChainManager = getFilterChainManager();
4. **if** (!filterChainManager.hasChains()) {
5. **return** **null**;
6. }
7. //2、接着获取当前请求的URL（不带上下文）
8. String requestURI = getPathWithinApplication(request);
9. //3、循环拦截器管理器中的拦截器定义（拦截器链的名字就是URL模式）
10. **for** (String pathPattern : filterChainManager.getChainNames()) {
11. //4、如当前URL匹配拦截器名字（URL模式）
12. **if** (pathMatches(pathPattern, requestURI)) {
13. //5、返回该URL模式定义的拦截器链
14. **return** filterChainManager.proxy(originalChain, pathPattern);
15. }
16. }
17. **return** **null**;
18. }

默认实现有点小问题：

如果多个拦截器链都匹配了当前请求URL，那么只返回第一个找到的拦截器链；后续我们可以修改此处的代码，将多个匹配的拦截器链合并返回。

DefaultFilterChainManager内部使用Map来管理URL模式-拦截器链的关系；也就是说相同的URL模式只能定义一个拦截器链，不能重复定义；而且如果多个拦截器链都匹配时是无序的（因为使用map.keySet()获取拦截器链的名字，即URL模式）。

FilterChainManager接口：

**Java代码  [收藏代码](javascript:void())**

1. **public** **interface** FilterChainManager {
2. Map<String, Filter> getFilters(); //得到注册的拦截器
3. **void** addFilter(String name, Filter filter); //注册拦截器
4. **void** addFilter(String name, Filter filter, **boolean** init); //注册拦截器
5. **void** createChain(String chainName, String chainDefinition); //根据拦截器链定义创建拦截器链
6. **void** addToChain(String chainName, String filterName); //添加拦截器到指定的拦截器链
7. **void** addToChain(String chainName, String filterName, String chainSpecificFilterConfig) **throws** ConfigurationException; //添加拦截器（带有配置的）到指定的拦截器链
8. NamedFilterList getChain(String chainName); //获取拦截器链
9. **boolean** hasChains(); //是否有拦截器链
10. Set<String> getChainNames(); //得到所有拦截器链的名字
11. FilterChain proxy(FilterChain original, String chainName); //使用指定的拦截器链代理原始拦截器链
12. }

此接口主要三个功能：注册拦截器，注册拦截器链，对原始拦截器链生成代理之后的拦截器链，比如

**Java代码  [收藏代码](javascript:void())**

1. <bean id="shiroFilter" **class**="org.apache.shiro.spring.web.ShiroFilterFactoryBean">
2. ……
3. <property name="filters">
4. <util:map>
5. <entry key="authc" value-ref="formAuthenticationFilter"/>
6. <entry key="sysUser" value-ref="sysUserFilter"/>
7. </util:map>
8. </property>
9. <property name="filterChainDefinitions">
10. <value>
11. /login = authc
12. /logout = logout
13. /authenticated = authc
14. /\*\* = user,sysUser
15. </value>
16. </property>
17. </bean>

filters属性定义了拦截器；filterChainDefinitions定义了拦截器链；如/\*\*就是拦截器链的名字；而user,sysUser就是拦截器名字列表。

之前说过默认的PathMatchingFilterChainResolver和DefaultFilterChainManager不能满足我们的需求，我们稍微扩展了一下：

**CustomPathMatchingFilterChainResolver**

**Java代码  [收藏代码](javascript:void())**

1. **public** **class** CustomPathMatchingFilterChainResolver
2. **extends** PathMatchingFilterChainResolver {
3. **private** CustomDefaultFilterChainManager customDefaultFilterChainManager;
4. **public** **void** setCustomDefaultFilterChainManager(
5. CustomDefaultFilterChainManager customDefaultFilterChainManager) {
6. **this**.customDefaultFilterChainManager = customDefaultFilterChainManager;
7. setFilterChainManager(customDefaultFilterChainManager);
8. }
10. **public** FilterChain getChain(ServletRequest request, ServletResponse response, FilterChain originalChain) {
11. FilterChainManager filterChainManager = getFilterChainManager();
12. **if** (!filterChainManager.hasChains()) {
13. **return** **null**;
14. }
15. String requestURI = getPathWithinApplication(request);
16. List<String> chainNames = **new** ArrayList<String>();
17. **for** (String pathPattern : filterChainManager.getChainNames()) {
18. **if** (pathMatches(pathPattern, requestURI)) {
19. chainNames.add(pathPattern);
20. }
21. }
22. **if**(chainNames.size() == 0) {
23. **return** **null**;
24. }
25. **return** customDefaultFilterChainManager.proxy(originalChain, chainNames);
26. }
27. }

和默认的PathMatchingFilterChainResolver区别是，此处得到所有匹配的拦截器链，然后通过调用CustomDefaultFilterChainManager.proxy(originalChain, chainNames)进行合并后代理。

**CustomDefaultFilterChainManager**

**Java代码  [收藏代码](javascript:void())**

1. **public** **class** CustomDefaultFilterChainManager **extends** DefaultFilterChainManager {
2. **private** Map<String, String> filterChainDefinitionMap = **null**;
3. **private** String loginUrl;
4. **private** String successUrl;
5. **private** String unauthorizedUrl;
6. **public** CustomDefaultFilterChainManager() {
7. setFilters(**new** LinkedHashMap<String, Filter>());
8. setFilterChains(**new** LinkedHashMap<String, NamedFilterList>());
9. addDefaultFilters(**true**);
10. }
11. **public** Map<String, String> getFilterChainDefinitionMap() {
12. **return** filterChainDefinitionMap;
13. }
14. **public** **void** setFilterChainDefinitionMap(Map<String, String> filterChainDefinitionMap) {
15. **this**.filterChainDefinitionMap = filterChainDefinitionMap;
16. }
17. **public** **void** setCustomFilters(Map<String, Filter> customFilters) {
18. **for**(Map.Entry<String, Filter> entry : customFilters.entrySet()) {
19. addFilter(entry.getKey(), entry.getValue(), **false**);
20. }
21. }
22. **public** **void** setDefaultFilterChainDefinitions(String definitions) {
23. Ini ini = **new** Ini();
24. ini.load(definitions);
25. Ini.Section section = ini.getSection(IniFilterChainResolverFactory.URLS);
26. **if** (CollectionUtils.isEmpty(section)) {
27. section = ini.getSection(Ini.DEFAULT\_SECTION\_NAME);
28. }
29. setFilterChainDefinitionMap(section);
30. }
31. **public** String getLoginUrl() {
32. **return** loginUrl;
33. }
34. **public** **void** setLoginUrl(String loginUrl) {
35. **this**.loginUrl = loginUrl;
36. }
37. **public** String getSuccessUrl() {
38. **return** successUrl;
39. }
40. **public** **void** setSuccessUrl(String successUrl) {
41. **this**.successUrl = successUrl;
42. }
43. **public** String getUnauthorizedUrl() {
44. **return** unauthorizedUrl;
45. }
46. **public** **void** setUnauthorizedUrl(String unauthorizedUrl) {
47. **this**.unauthorizedUrl = unauthorizedUrl;
48. }
49. @PostConstruct
50. **public** **void** init() {
51. Map<String, Filter> filters = getFilters();
52. **if** (!CollectionUtils.isEmpty(filters)) {
53. **for** (Map.Entry<String, Filter> entry : filters.entrySet()) {
54. String name = entry.getKey();
55. Filter filter = entry.getValue();
56. applyGlobalPropertiesIfNecessary(filter);
57. **if** (filter **instanceof** Nameable) {
58. ((Nameable) filter).setName(name);
59. }
60. addFilter(name, filter, **false**);
61. }
62. }
63. Map<String, String> chains = getFilterChainDefinitionMap();
64. **if** (!CollectionUtils.isEmpty(chains)) {
65. **for** (Map.Entry<String, String> entry : chains.entrySet()) {
66. String url = entry.getKey();
67. String chainDefinition = entry.getValue();
68. createChain(url, chainDefinition);
69. }
70. }
71. }
72. **protected** **void** initFilter(Filter filter) {
73. //ignore
74. }
76. **public** FilterChain proxy(FilterChain original, List<String> chainNames) {
77. NamedFilterList configured = **new** SimpleNamedFilterList(chainNames.toString());
78. **for**(String chainName : chainNames) {
79. configured.addAll(getChain(chainName));
80. }
81. **return** configured.proxy(original);
82. }
83. **private** **void** applyGlobalPropertiesIfNecessary(Filter filter) {
84. applyLoginUrlIfNecessary(filter);
85. applySuccessUrlIfNecessary(filter);
86. applyUnauthorizedUrlIfNecessary(filter);
87. }
88. **private** **void** applyLoginUrlIfNecessary(Filter filter) {
89. //请参考源码
90. }
91. **private** **void** applySuccessUrlIfNecessary(Filter filter) {
92. //请参考源码
93. }
94. **private** **void** applyUnauthorizedUrlIfNecessary(Filter filter) {
95. //请参考源码
96. }
97. }

1、CustomDefaultFilterChainManager：调用其构造器时，会自动注册默认的拦截器；

2、loginUrl、successUrl、unauthorizedUrl：分别对应登录地址、登录成功后默认跳转地址、未授权跳转地址，用于给相应拦截器的；

3、filterChainDefinitionMap：用于存储如ShiroFilterFactoryBean在配置文件中配置的拦截器链定义，即可以认为是默认的静态拦截器链；会自动与数据库中加载的合并；

4、setDefaultFilterChainDefinitions：解析配置文件中传入的字符串拦截器链配置，解析为相应的拦截器链；

5、setCustomFilters：注册我们自定义的拦截器；如ShiroFilterFactoryBean的filters属性；

6、init：初始化方法，Spring容器启动时会调用，首先其会自动给相应的拦截器设置如loginUrl、successUrl、unauthorizedUrl；其次根据filterChainDefinitionMap构建默认的拦截器链；

7、initFilter：此处我们忽略实现initFilter，因为交给spring管理了，所以Filter的相关配置会在Spring配置中完成；

8、proxy：组合多个拦截器链为一个生成一个新的FilterChain代理。

**Web层控制器**

请参考com.github.zhangkaitao.shiro.chapter19.web.controller包，相对于第十六章添加了UrlFilterController用于UrlFilter的维护。另外，移除了控制器方法上的权限注解，而是使用动态URL拦截进行控制。

**Spring配置——spring-config-shiro.xml**

**Java代码  [收藏代码](javascript:void())**

1. <bean id="filterChainManager"
2. **class**="com.github.zhangkaitao.shiro.spring.CustomDefaultFilterChainManager">
3. <property name="loginUrl" value="/login"/>
4. <property name="successUrl" value="/"/>
5. <property name="unauthorizedUrl" value="/unauthorized.jsp"/>
6. <property name="customFilters">
7. <util:map>
8. <entry key="authc" value-ref="formAuthenticationFilter"/>
9. <entry key="sysUser" value-ref="sysUserFilter"/>
10. </util:map>
11. </property>
12. <property name="defaultFilterChainDefinitions">
13. <value>
14. /login = authc
15. /logout = logout
16. /unauthorized.jsp = authc
17. /\*\* = user,sysUser
18. </value>
19. </property>
20. </bean>

filterChainManager是我们自定义的CustomDefaultFilterChainManager，注册相应的拦截器及默认的拦截器链。

**Java代码  [收藏代码](javascript:void())**

1. <bean id="filterChainResolver"
2. **class**="com.github.zhangkaitao.shiro.spring.CustomPathMatchingFilterChainResolver">
3. <property name="customDefaultFilterChainManager" ref="filterChainManager"/>
4. </bean>

filterChainResolver是自定义的CustomPathMatchingFilterChainResolver，使用上边的filterChainManager进行拦截器链的管理。

**Java代码  [收藏代码](javascript:void())**

1. <bean id="shiroFilter" **class**="org.apache.shiro.spring.web.ShiroFilterFactoryBean">
2. <property name="securityManager" ref="securityManager"/>
3. </bean>

shiroFilter不再定义filters及filterChainDefinitions，而是交给了filterChainManager进行完成。

**Java代码  [收藏代码](javascript:void())**

1. <bean **class**="org.springframework.beans.factory.config.MethodInvokingFactoryBean">
2. <property name="targetObject" ref="shiroFilter"/>
3. <property name="targetMethod" value="setFilterChainResolver"/>
4. <property name="arguments" ref="filterChainResolver"/>
5. </bean>

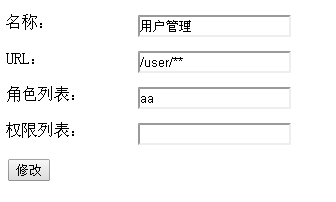
最后把filterChainResolver注册给shiroFilter，其使用它进行动态URL权限控制。

其他配置和第十六章一样，请参考第十六章。

**测试**

1、首先执行shiro-data.sql初始化数据。

2、然后再URL管理中新增如下数据：



3、访问http://localhost:8080/chapter19/user时要求用户拥有aa角色，此时是没有的所以会跳转到未授权页面；

4、添加aa角色然后授权给用户，此时就有权限访问http://localhost:8080/chapter19/user。

实际项目可以在此基础上进行扩展。