Shiro与Spring集成

Shiro的组件都是JavaBean/POJO式的组件，所以非常容易使用Spring进行组件管理，可以非常方便的从ini配置迁移到Spring进行管理，且支持JavaSE应用及Web应用的集成。

在示例之前，需要导入shiro-spring及spring-context依赖，具体请参考pom.xml。

spring-beans.xml配置文件提供了基础组件如DataSource、DAO、Service组件的配置。

JavaSE应用

spring-shiro.xml提供了普通JavaSE独立应用的Spring配置：

<!-- 缓存管理器 使用Ehcache实现 -->

<bean id="cacheManager" class="org.apache.shiro.cache.ehcache.EhCacheManager">

    <property name="cacheManagerConfigFile" value="classpath:ehcache.xml"/>

</bean>

<!-- 凭证匹配器 -->

<bean id="credentialsMatcher" class="

com.github.zhangkaitao.shiro.chapter12.credentials.RetryLimitHashedCredentialsMatcher">

    <constructor-arg ref="cacheManager"/>

    <property name="hashAlgorithmName" value="md5"/>

    <property name="hashIterations" value="2"/>

    <property name="storedCredentialsHexEncoded" value="true"/>

</bean>

<!-- Realm实现 -->

<bean id="userRealm" class="com.github.zhangkaitao.shiro.chapter12.realm.UserRealm">

    <property name="userService" ref="userService"/>

    <property name="credentialsMatcher" ref="credentialsMatcher"/>

    <property name="cachingEnabled" value="true"/>

    <property name="authenticationCachingEnabled" value="true"/>

    <property name="authenticationCacheName" value="authenticationCache"/>

    <property name="authorizationCachingEnabled" value="true"/>

    <property name="authorizationCacheName" value="authorizationCache"/>

</bean>

<!-- 会话ID生成器 -->

<bean id="sessionIdGenerator"

class="org.apache.shiro.session.mgt.eis.JavaUuidSessionIdGenerator"/>

<!-- 会话DAO -->

<bean id="sessionDAO"

class="org.apache.shiro.session.mgt.eis.EnterpriseCacheSessionDAO">

    <property name="activeSessionsCacheName" value="shiro-activeSessionCache"/>

    <property name="sessionIdGenerator" ref="sessionIdGenerator"/>

</bean>

<!-- 会话验证调度器 -->

<bean id="sessionValidationScheduler"

class="org.apache.shiro.session.mgt.quartz.QuartzSessionValidationScheduler">

    <property name="sessionValidationInterval" value="1800000"/>

    <property name="sessionManager" ref="sessionManager"/>

</bean>

<!-- 会话管理器 -->

<bean id="sessionManager" class="org.apache.shiro.session.mgt.DefaultSessionManager">

    <property name="globalSessionTimeout" value="1800000"/>

    <property name="deleteInvalidSessions" value="true"/>

    <property name="sessionValidationSchedulerEnabled" value="true"/>

   <property name="sessionValidationScheduler" ref="sessionValidationScheduler"/>

    <property name="sessionDAO" ref="sessionDAO"/>

</bean>

<!-- 安全管理器 -->

<bean id="securityManager" class="org.apache.shiro.mgt.DefaultSecurityManager">

    <property name="realms">

        <list><ref bean="userRealm"/></list>

    </property>

    <property name="sessionManager" ref="sessionManager"/>

    <property name="cacheManager" ref="cacheManager"/>

</bean>

<!-- 相当于调用SecurityUtils.setSecurityManager(securityManager) -->

<bean class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">

<property name="staticMethod"

value="org.apache.shiro.SecurityUtils.setSecurityManager"/>

    <property name="arguments" ref="securityManager"/>

</bean>

<!-- Shiro生命周期处理器-->

<bean id="lifecycleBeanPostProcessor"

class="org.apache.shiro.spring.LifecycleBeanPostProcessor"/>

<!-- 缓存管理器 使用Ehcache实现 -->

<bean id="cacheManager" class="org.apache.shiro.cache.ehcache.EhCacheManager">

<property name="cacheManagerConfigFile" value="classpath:ehcache.xml"/>

</bean>

<!-- 凭证匹配器 -->

<bean id="credentialsMatcher" class="

com.github.zhangkaitao.shiro.chapter12.credentials.RetryLimitHashedCredentialsMatcher">

<constructor-arg ref="cacheManager"/>

<property name="hashAlgorithmName" value="md5"/>

<property name="hashIterations" value="2"/>

<property name="storedCredentialsHexEncoded" value="true"/>

</bean>

<!-- Realm实现 -->

<bean id="userRealm" class="com.github.zhangkaitao.shiro.chapter12.realm.UserRealm">

<property name="userService" ref="userService"/>

<property name="credentialsMatcher" ref="credentialsMatcher"/>

<property name="cachingEnabled" value="true"/>

<property name="authenticationCachingEnabled" value="true"/>

<property name="authenticationCacheName" value="authenticationCache"/>

<property name="authorizationCachingEnabled" value="true"/>

<property name="authorizationCacheName" value="authorizationCache"/>

</bean>

<!-- 会话ID生成器 -->

<bean id="sessionIdGenerator"

class="org.apache.shiro.session.mgt.eis.JavaUuidSessionIdGenerator"/>

<!-- 会话DAO -->

<bean id="sessionDAO"

class="org.apache.shiro.session.mgt.eis.EnterpriseCacheSessionDAO">

<property name="activeSessionsCacheName" value="shiro-activeSessionCache"/>

<property name="sessionIdGenerator" ref="sessionIdGenerator"/>

</bean>

<!-- 会话验证调度器 -->

<bean id="sessionValidationScheduler"

class="org.apache.shiro.session.mgt.quartz.QuartzSessionValidationScheduler">

<property name="sessionValidationInterval" value="1800000"/>

<property name="sessionManager" ref="sessionManager"/>

</bean>

<!-- 会话管理器 -->

<bean id="sessionManager" class="org.apache.shiro.session.mgt.DefaultSessionManager">

<property name="globalSessionTimeout" value="1800000"/>

<property name="deleteInvalidSessions" value="true"/>

<property name="sessionValidationSchedulerEnabled" value="true"/>

<property name="sessionValidationScheduler" ref="sessionValidationScheduler"/>

<property name="sessionDAO" ref="sessionDAO"/>

</bean>

<!-- 安全管理器 -->

<bean id="securityManager" class="org.apache.shiro.mgt.DefaultSecurityManager">

<property name="realms">

<list><ref bean="userRealm"/></list>

</property>

<property name="sessionManager" ref="sessionManager"/>

<property name="cacheManager" ref="cacheManager"/>

</bean>

<!-- 相当于调用SecurityUtils.setSecurityManager(securityManager) -->

<bean class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">

<property name="staticMethod"

value="org.apache.shiro.SecurityUtils.setSecurityManager"/>

<property name="arguments" ref="securityManager"/>

</bean>

<!-- Shiro生命周期处理器-->

<bean id="lifecycleBeanPostProcessor"

class="org.apache.shiro.spring.LifecycleBeanPostProcessor"/>

可以看出，只要把之前的ini配置翻译为此处的spring xml配置方式即可，无须多解释。LifecycleBeanPostProcessor用于在实现了Initializable接口的Shiro bean初始化时调用Initializable接口回调，在实现了Destroyable接口的Shiro bean销毁时调用 Destroyable接口回调。如UserRealm就实现了Initializable，而DefaultSecurityManager实现了Destroyable。具体可以查看它们的继承关系。

测试用例请参考com.github.zhangkaitao.shiro.chapter12.ShiroTest。

**Web应用**

Web应用和普通JavaSE应用的某些配置是类似的，此处只提供一些不一样的配置，详细配置可以参考spring-shiro-web.xml。

<!-- 会话Cookie模板 -->

<bean id="sessionIdCookie" class="org.apache.shiro.web.servlet.SimpleCookie">

    <constructor-arg value="sid"/>

    <property name="httpOnly" value="true"/>

    <property name="maxAge" value="180000"/>

</bean>

<!-- 会话管理器 -->

<bean id="sessionManager"

class="org.apache.shiro.web.session.mgt.DefaultWebSessionManager">

    <property name="globalSessionTimeout" value="1800000"/>

    <property name="deleteInvalidSessions" value="true"/>

    <property name="sessionValidationSchedulerEnabled" value="true"/>

    <property name="sessionValidationScheduler" ref="sessionValidationScheduler"/>

    <property name="sessionDAO" ref="sessionDAO"/>

    <property name="sessionIdCookieEnabled" value="true"/>

    <property name="sessionIdCookie" ref="sessionIdCookie"/>

</bean>

<!-- 安全管理器 -->

<bean id="securityManager" class="org.apache.shiro.web.mgt.DefaultWebSecurityManager">

<property name="realm" ref="userRealm"/>

    <property name="sessionManager" ref="sessionManager"/>

    <property name="cacheManager" ref="cacheManager"/>

</bean>

1、sessionIdCookie是用于生产Session ID Cookie的模板；

2、会话管理器使用用于web环境的DefaultWebSessionManager；

3、安全管理器使用用于web环境的DefaultWebSecurityManager。

<!-- 基于Form表单的身份验证过滤器 -->

<bean id="formAuthenticationFilter"

class="org.apache.shiro.web.filter.authc.FormAuthenticationFilter">

    <property name="usernameParam" value="username"/>

    <property name="passwordParam" value="password"/>

    <property name="loginUrl" value="/login.jsp"/>

</bean>

<!-- Shiro的Web过滤器 -->

<bean id="shiroFilter" class="org.apache.shiro.spring.web.ShiroFilterFactoryBean">

    <property name="securityManager" ref="securityManager"/>

    <property name="loginUrl" value="/login.jsp"/>

    <property name="unauthorizedUrl" value="/unauthorized.jsp"/>

    <property name="filters">

        <util:map>

            <entry key="authc" value-ref="formAuthenticationFilter"/>

        </util:map>

    </property>

    <property name="filterChainDefinitions">

        <value>

            /index.jsp = anon

            /unauthorized.jsp = anon

            /login.jsp = authc

            /logout = logout

            /\*\* = user

        </value>

    </property>

</bean>

1、formAuthenticationFilter为基于Form表单的身份验证过滤器；此处可以再添加自己的Filter bean定义；

2、shiroFilter：此处使用ShiroFilterFactoryBean来创建ShiroFilter过滤器；filters属性用于定义自己的过滤器，即ini配置中的[filters]部分；filterChainDefinitions用于声明url和filter的关系，即ini配置中的[urls]部分。

接着需要在web.xml中进行如下配置：

<context-param>

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

    <param-value>

        classpath:spring-beans.xml,

        classpath:spring-shiro-web.xml

    </param-value>

</context-param>

<listener>

   <listener-class>

org.springframework.web.context.ContextLoaderListener

</listener-class>

</listener>

<context-param>

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

<param-value>

classpath:spring-beans.xml,

classpath:spring-shiro-web.xml

</param-value>

</context-param>

<listener>

<listener-class>

org.springframework.web.context.ContextLoaderListener

</listener-class>

</listener>

通过ContextLoaderListener加载contextConfigLocation指定的Spring配置文件。

<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>

<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>

DelegatingFilterProxy会自动到Spring容器中查找名字为shiroFilter的bean并把filter请求交给它处理。

其他配置请参考源代码。

Shiro权限注解

Shiro提供了相应的注解用于权限控制，如果使用这些注解就需要使用AOP的功能来进行判断，如Spring AOP；Shiro提供了Spring AOP集成用于权限注解的解析和验证。

为了测试，此处使用了Spring MVC来测试Shiro注解，当然Shiro注解不仅仅可以在web环境使用，在独立的JavaSE中也是可以用的，此处只是以web为例了。

在spring-mvc.xml配置文件添加Shiro Spring AOP权限注解的支持：

<aop:config proxy-target-class="true"></aop:config>

<bean class="

org.apache.shiro.spring.security.interceptor.AuthorizationAttributeSourceAdvisor">

    <property name="securityManager" ref="securityManager"/>

</bean>

<aop:config proxy-target-class="true"></aop:config>

<bean class="

org.apache.shiro.spring.security.interceptor.AuthorizationAttributeSourceAdvisor">

<property name="securityManager" ref="securityManager"/>

</bean>

如上配置用于开启Shiro Spring AOP权限注解的支持；<aop:config proxy-target-class="true">表示代理类。

接着就可以在相应的控制器（AnnotationController）中使用如下方式进行注解：

@RequiresRoles("admin")

@RequestMapping("/hello2")

public String hello2() {

    return "success";

}

@RequiresRoles("admin")

@RequestMapping("/hello2")

public String hello2() {

return "success";

}

访问hello2方法的前提是当前用户有admin角色。

当验证失败，其会抛出UnauthorizedException异常，此时可以使用Spring的ExceptionHandler（DefaultExceptionHandler）来进行拦截处理：

@ExceptionHandler({UnauthorizedException.class})

@ResponseStatus(HttpStatus.UNAUTHORIZED)

public ModelAndView processUnauthenticatedException(NativeWebRequest request, UnauthorizedException e) {

    ModelAndView mv = new ModelAndView();

    mv.addObject("exception", e);

    mv.setViewName("unauthorized");

    return mv;

}

@ExceptionHandler({UnauthorizedException.class})

@ResponseStatus(HttpStatus.UNAUTHORIZED)

public ModelAndView processUnauthenticatedException(NativeWebRequest request, UnauthorizedException e) {

ModelAndView mv = new ModelAndView();

mv.addObject("exception", e);

mv.setViewName("unauthorized");

return mv;

}

如果集成Struts2，需要注意《Shiro+Struts2+Spring3 加上@RequiresPermissions 后@Autowired失效》问题：

<http://jinnianshilongnian.iteye.com/blog/1850425>

权限注解

@RequiresAuthentication

@RequiresAuthentication

表示当前Subject已经通过login进行了身份验证；即Subject. isAuthenticated()返回true。

@RequiresUser

@RequiresUser

表示当前Subject已经身份验证或者通过记住我登录的。

@RequiresGuest

@RequiresGuest

表示当前Subject没有身份验证或通过记住我登录过，即是游客身份。

@RequiresRoles(value={“admin”, “user”}, logical= Logical.AND)

@RequiresRoles(value={“admin”, “user”}, logical= Logical.AND)

表示当前Subject需要角色admin和user。

@RequiresPermissions (value={“user:a”, “user:b”}, logical= Logical.OR)

@RequiresPermissions (value={“user:a”, “user:b”}, logical= Logical.OR)

表示当前Subject需要权限user:a或user:b。

**apache shiro与spring整合、动态filterChainDefinitions、以及认证、授权**

apache shiro是一个安全认证框架，和spring security相比，在于他使用了比较简洁易懂的认证和授权方式。其提供的native-session（即把用户认证后的授权信息保存在其自身提供Session中）机制，这样就可以和HttpSession、EJB Session Bean的基于容器的Session脱耦，到和客户端应用、Flex应用、远程方法调用等都可以使用它来配置权限认证。 在exit-web-framework里的vcs-admin例子用到该框架，具体使用说明可以参考[官方帮助文档](http://shiro.apache.org/documentation.html)。在这里主要讲解如何与spring结合、动态创建filterchaindefinitions、以及认证、授权、和缓存处理。

apache shiro 结合spring

Shiro 拥有对Spring Web 应用程序的一流支持。在Web 应用程序中，所有Shiro 可访问的万恶不请求必须通过一个主要的Shiro 过滤器。该过滤器本身是极为强大的，允许临时的自定义过滤器链基于任何URL 路径表达式执行。 在Shiro 1.0 之前，你不得不在Spring web 应用程序中使用一个混合的方式，来定义Shiro 过滤器及所有它在web.xml中的配置属性，但在Spring XML 中定义SecurityManager。这有些令人沮丧，由于你不能把你的配置固定在一个地方，以及利用更为先进的Spring 功能的配置能力，如PropertyPlaceholderConfigurer 或抽象bean 来固定通用配置。现在在Shiro 1.0 及以后版本中，所有Shiro 配置都是在Spring XML 中完成的，用来提供更为强健的Spring 配置机制。以下是如何在基于Spring web 应用程序中配置Shiro： web.xml:

<!-- Spring ApplicationContext配置文件的路径,可使用通配符,多个路径用,号分隔 此参数用于后面的Spring Context Loader -->

<context-param>

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

    <param-value>

        classpath\*:/applicationContext-shiro.xml

    </param-value>

</context-param>

<!-- shiro security filter -->

<filter>

    <!-- 这里的filter-name要和spring的applicationContext-shiro.xml里的

            org.apache.shiro.spring.web.ShiroFilterFactoryBean的bean name相同 -->

    <filter-name>shiroSecurityFilter</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>shiroSecurityFilter</filter-name>

    <url-pattern>/\*</url-pattern>

</filter-mapping>

<!-- Spring ApplicationContext配置文件的路径,可使用通配符,多个路径用,号分隔 此参数用于后面的Spring Context Loader -->

<context-param>

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

<param-value>

classpath\*:/applicationContext-shiro.xml

</param-value>

</context-param>

<!-- shiro security filter -->

<filter>

<!-- 这里的filter-name要和spring的applicationContext-shiro.xml里的

org.apache.shiro.spring.web.ShiroFilterFactoryBean的bean name相同 -->

<filter-name>shiroSecurityFilter</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>shiroSecurityFilter</filter-name>

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

</filter-mapping>

applicationContext-shiro.xml文件中，定义web支持的SecurityManager和"shiroSecurityFilter"bean将会被web.xml 引用。

<bean id="shiroSecurityFilter">

    <!-- shiro的核心安全接口 -->

    <property name="securityManager" ref="securityManager" />

    <!-- 要求登录时的链接 -->

    <property name="loginUrl" value="/login.jsp" />

    <!-- 登陆成功后要跳转的连接 -->

    <property name="successUrl" value="/index.jsp" />

    <!-- 未授权时要跳转的连接 -->

    <property name="unauthorizedUrl" value="/unauthorized.jsp" />

    <!-- shiro连接约束配置 -->

    <property name="filterChainDefinitions">

        <value>

            /login = authc

            /logout = logout

            /resource/\*\* = anon

        </value>

    </property>

</bean>

<bean id="securityManager">

</bean>

<bean id="lifecycleBeanPostProcessor"/>

<bean id="shiroSecurityFilter">

<!-- shiro的核心安全接口 -->

<property name="securityManager" ref="securityManager" />

<!-- 要求登录时的链接 -->

<property name="loginUrl" value="/login.jsp" />

<!-- 登陆成功后要跳转的连接 -->

<property name="successUrl" value="/index.jsp" />

<!-- 未授权时要跳转的连接 -->

<property name="unauthorizedUrl" value="/unauthorized.jsp" />

<!-- shiro连接约束配置 -->

<property name="filterChainDefinitions">

<value>

/login = authc

/logout = logout

/resource/\*\* = anon

</value>

</property>

</bean>

<bean id="securityManager">

</bean>

<bean id="lifecycleBeanPostProcessor"/>

启用Shiro注解

在独立应用程序和Web应用程序中，你可能想为安全检查使用Shiro 的注释（例如，@RequiresRoles，@RequiresPermissions 等等）。这需要Shiro 的Spring AOP 集成来扫描合适的注解类以及执行必要的安全逻辑。以下是如何使用这些注解的。只需添加这两个bean 定义到applicationContext-shiro.xml 中：

<bean depends-on="lifecycleBeanPostProcessor"/>

<bean>

    <property name="securityManager" ref="securityManager"/>

</bean>

<bean depends-on="lifecycleBeanPostProcessor"/>

<bean>

<property name="securityManager" ref="securityManager"/>

</bean>

动态创建filterchaindefinitions

有时，在某些系统想通过读取数据库来定义org.apache.shiro.spring.web.ShiroFilterFactoryBean的filterChainDefinitions。这样能够通过操作界面或者维护后台来管理系统的链接。

在shrio与spring集成好了以后，调试源码的高人可能已经注意到。项目启动时，shrio通过自己的org.apache.shiro.spring.web.ShiroFilterFactoryBean类的filterChainDefinitions(授权规则定义)属性转换为一个filterChainDefinitionMap，转换完成后交给ShiroFilterFactoryBean保管。ShiroFilterFactoryBean根据授权（AuthorizationInfo类）后的信息去判断哪些链接能访问哪些链接不能访问。filterChainDefinitionMap里面的键就是链接URL,值就是存在什么条件才能访问该链接，如perms、roles。filterChainDefinitionMap是一个Map，shiro扩展出一个Map的子类Ini.Section

现在有一张表的描述实体类，以及数据访问:

[java] [view plain](http://blog.csdn.net/z971829916/article/details/22572587) [copy](http://blog.csdn.net/z971829916/article/details/22572587) [print?](http://blog.csdn.net/z971829916/article/details/22572587)[在CODE上查看代码片](https://code.csdn.net/snippets/264529)

@Entity

@Table(name="TB\_RESOURCE")

public class Resource implements Serializable {

    //主键id

    @Id

    private String id;

    //action url

    private String value;

    //shiro permission;

    private String permission;

    //------------------Getter/Setter---------------------//

}

@Entity

@Table(name="TB\_RESOURCE")

public class Resource implements Serializable {

//主键id

@Id

private String id;

//action url

private String value;

//shiro permission;

private String permission;

//------------------Getter/Setter---------------------//

}

[java] [view plain](http://blog.csdn.net/z971829916/article/details/22572587) [copy](http://blog.csdn.net/z971829916/article/details/22572587) [print?](http://blog.csdn.net/z971829916/article/details/22572587)[在CODE上查看代码片](https://code.csdn.net/snippets/264529)

@Repository

public class ResourceDao extends BasicHibernateDao<Resource, String> {

}

@Repository

public class ResourceDao extends BasicHibernateDao<Resource, String> {

}

通过该类可以知道permission字段和value就是filterChainDefinitionMap的键/值，用spring FactoryBean接口的实现getObject()返回Section给filterChainDefinitionMap即可

[java] [view plain](http://blog.csdn.net/z971829916/article/details/22572587) [copy](http://blog.csdn.net/z971829916/article/details/22572587) [print?](http://blog.csdn.net/z971829916/article/details/22572587)[在CODE上查看代码片](https://code.csdn.net/snippets/264529)

public class ChainDefinitionSectionMetaSource implements FactoryBean<Ini.Section>{

    @Autowired

    private ResourceDao resourceDao;

    private String filterChainDefinitions;

    /\*\*

     \* 默认premission字符串

     \*/

    public static final String PREMISSION\_STRING="perms[\"{0}\"]";

    public Section getObject() throws BeansException {

        //获取所有Resource

        List<Resource> list = resourceDao.getAll();

        Ini ini = new Ini();

        //加载默认的url

        ini.load(filterChainDefinitions);

        Ini.Section section = ini.getSection(Ini.DEFAULT\_SECTION\_NAME);

        //循环Resource的url,逐个添加到section中。section就是filterChainDefinitionMap,

        //里面的键就是链接URL,值就是存在什么条件才能访问该链接

        for (Iterator<Resource> it = list.iterator(); it.hasNext();) {

            Resource resource = it.next();

            //如果不为空值添加到section中

            if(StringUtils.isNotEmpty(resource.getValue()) && StringUtils.isNotEmpty(resource.getPermission())) {

                section.put(resource.getValue(),  MessageFormat.format(PREMISSION\_STRING,resource.getPermission()));

            }

        }

        return section;

    }

    /\*\*

     \* 通过filterChainDefinitions对默认的url过滤定义

     \*

     \* @param filterChainDefinitions 默认的url过滤定义

     \*/

    public void setFilterChainDefinitions(String filterChainDefinitions) {

        this.filterChainDefinitions = filterChainDefinitions;

    }

    public Class<?> getObjectType() {

        return this.getClass();

    }

    public boolean isSingleton() {

        return false;

    }

}

public class ChainDefinitionSectionMetaSource implements FactoryBean<Ini.Section>{

@Autowired

private ResourceDao resourceDao;

private String filterChainDefinitions;

/\*\*

\* 默认premission字符串

\*/

public static final String PREMISSION\_STRING="perms[\"{0}\"]";

public Section getObject() throws BeansException {

//获取所有Resource

List<Resource> list = resourceDao.getAll();

Ini ini = new Ini();

//加载默认的url

ini.load(filterChainDefinitions);

Ini.Section section = ini.getSection(Ini.DEFAULT\_SECTION\_NAME);

//循环Resource的url,逐个添加到section中。section就是filterChainDefinitionMap,

//里面的键就是链接URL,值就是存在什么条件才能访问该链接

for (Iterator<Resource> it = list.iterator(); it.hasNext();) {

Resource resource = it.next();

//如果不为空值添加到section中

if(StringUtils.isNotEmpty(resource.getValue()) && StringUtils.isNotEmpty(resource.getPermission())) {

section.put(resource.getValue(), MessageFormat.format(PREMISSION\_STRING,resource.getPermission()));

}

}

return section;

}

/\*\*

\* 通过filterChainDefinitions对默认的url过滤定义

\*

\* @param filterChainDefinitions 默认的url过滤定义

\*/

public void setFilterChainDefinitions(String filterChainDefinitions) {

this.filterChainDefinitions = filterChainDefinitions;

}

public Class<?> getObjectType() {

return this.getClass();

}

public boolean isSingleton() {

return false;

}

}

定义好了chainDefinitionSectionMetaSource后修改applicationContext-shiro.xml文件

[java] [view plain](http://blog.csdn.net/z971829916/article/details/22572587) [copy](http://blog.csdn.net/z971829916/article/details/22572587) [print?](http://blog.csdn.net/z971829916/article/details/22572587)[在CODE上查看代码片](https://code.csdn.net/snippets/264529)

<bean id="chainDefinitionSectionMetaSource">

    <property name="filterChainDefinitions">

        <value>

            /login = authc

            /logout = logout

            /resource/\*\* = anon

        </value>

    </property>

</bean>

<bean id="shiroSecurityFilter">

    <property name="securityManager" ref="securityManager" />

    <property name="loginUrl" value="/login.jsp" />

    <property name="successUrl" value="/index.jsp" />

    <property name="unauthorizedUrl" value="/unauthorized.jsp" />

    <!-- shiro连接约束配置,在这里使用自定义的动态获取资源类 -->

    <property name="filterChainDefinitionMap" ref="chainDefinitionSectionMetaSource" />

</bean>

<bean id="securityManager">

</bean>

<bean id="lifecycleBeanPostProcessor"/>

<bean id="chainDefinitionSectionMetaSource">

<property name="filterChainDefinitions">

<value>

/login = authc

/logout = logout

/resource/\*\* = anon

</value>

</property>

</bean>

<bean id="shiroSecurityFilter">

<property name="securityManager" ref="securityManager" />

<property name="loginUrl" value="/login.jsp" />

<property name="successUrl" value="/index.jsp" />

<property name="unauthorizedUrl" value="/unauthorized.jsp" />

<!-- shiro连接约束配置,在这里使用自定义的动态获取资源类 -->

<property name="filterChainDefinitionMap" ref="chainDefinitionSectionMetaSource" />

</bean>

<bean id="securityManager">

</bean>

<bean id="lifecycleBeanPostProcessor"/>

shiro数据库认证、授权

在shiro认证和授权主要是两个类，就是org.apache.shiro.authc.AuthenticationInfo和org.apache.shiro.authz.AuthorizationInfo。该两个类的处理在org.apache.shiro.realm.AuthorizingRealm中已经给出了两个抽象方法。就是:

[java] [view plain](http://blog.csdn.net/z971829916/article/details/22572587) [copy](http://blog.csdn.net/z971829916/article/details/22572587) [print?](http://blog.csdn.net/z971829916/article/details/22572587)[在CODE上查看代码片](https://code.csdn.net/snippets/264529)

/\*\*

 \* Retrieves the AuthorizationInfo for the given principals from the underlying data store.  When returning

 \* an instance from this method, you might want to consider using an instance of

 \* {@link org.apache.shiro.authz.SimpleAuthorizationInfo SimpleAuthorizationInfo}, as it is suitable in most cases.

 \*

 \* @param principals the primary identifying principals of the AuthorizationInfo that should be retrieved.

 \* @return the AuthorizationInfo associated with this principals.

 \* @see org.apache.shiro.authz.SimpleAuthorizationInfo

 \*/

protected abstract AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principals);

/\*\*

 \* Retrieves authentication data from an implementation-specific datasource (RDBMS, LDAP, etc) for the given

 \* authentication token.

 \* <p/>

 \* For most datasources, this means just 'pulling' authentication data for an associated subject/user and nothing

 \* more and letting Shiro do the rest.  But in some systems, this method could actually perform EIS specific

 \* log-in logic in addition to just retrieving data - it is up to the Realm implementation.

 \* <p/>

 \* A {@code null} return value means that no account could be associated with the specified token.

 \*

 \* @param token the authentication token containing the user's principal and credentials.

 \* @return an {@link AuthenticationInfo} object containing account data resulting from the

 \*         authentication ONLY if the lookup is successful (i.e. account exists and is valid, etc.)

 \* @throws AuthenticationException if there is an error acquiring data or performing

 \*                                 realm-specific authentication logic for the specified <tt>token</tt>

 \*/

protected abstract AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken token) throws AuthenticationException;

/\*\*

\* Retrieves the AuthorizationInfo for the given principals from the underlying data store. When returning

\* an instance from this method, you might want to consider using an instance of

\* {@link org.apache.shiro.authz.SimpleAuthorizationInfo SimpleAuthorizationInfo}, as it is suitable in most cases.

\*

\* @param principals the primary identifying principals of the AuthorizationInfo that should be retrieved.

\* @return the AuthorizationInfo associated with this principals.

\* @see org.apache.shiro.authz.SimpleAuthorizationInfo

\*/

protected abstract AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principals);

/\*\*

\* Retrieves authentication data from an implementation-specific datasource (RDBMS, LDAP, etc) for the given

\* authentication token.

\* <p/>

\* For most datasources, this means just 'pulling' authentication data for an associated subject/user and nothing

\* more and letting Shiro do the rest. But in some systems, this method could actually perform EIS specific

\* log-in logic in addition to just retrieving data - it is up to the Realm implementation.

\* <p/>

\* A {@code null} return value means that no account could be associated with the specified token.

\*

\* @param token the authentication token containing the user's principal and credentials.

\* @return an {@link AuthenticationInfo} object containing account data resulting from the

\* authentication ONLY if the lookup is successful (i.e. account exists and is valid, etc.)

\* @throws AuthenticationException if there is an error acquiring data or performing

\* realm-specific authentication logic for the specified <tt>token</tt>

\*/

protected abstract AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken token) throws AuthenticationException;

doGetAuthenticationInfo(AuthenticationToken token)（认证/登录方法）会返回一个AuthenticationInfo,就是认证信息。是一个对执行及对用户的身份验证（登录）尝试负责的方法。当一个用户尝试登录时，该逻辑被认证器执行。认证器知道如何与一个或多个Realm协调来存储相关的用户/帐户信息。从这些Realm中获得的数据被用来验证用户的身份来保证用户确实是他们所说的他们是谁。

doGetAuthorizationInfo(PrincipalCollection principals)是负责在应用程序中决定用户的访问控制的方法。它是一种最终判定用户是否被允许做某件事的机制。与doGetAuthenticationInfo(AuthenticationToken token)相似，doGetAuthorizationInfo(PrincipalCollection principals) 也知道如何协调多个后台数据源来访问角色恶化权限信息和准确地决定用户是否被允许执行给定的动作。

简单的一个用户和一个资源实体:

[java] [view plain](http://blog.csdn.net/z971829916/article/details/22572587) [copy](http://blog.csdn.net/z971829916/article/details/22572587) [print?](http://blog.csdn.net/z971829916/article/details/22572587)[在CODE上查看代码片](https://code.csdn.net/snippets/264529)

@Entity

@Table(name="TB\_RESOURCE")

public class Resource implements Serializable {

    //主键id

    @Id

    private String id;

    //action url

    private String value;

    //shiro permission;

    private String permission;

    //------------------Getter/Setter---------------------//

}

@Entity

@Table(name="TB\_RESOURCE")

public class Resource implements Serializable {

//主键id

@Id

private String id;

//action url

private String value;

//shiro permission;

private String permission;

//------------------Getter/Setter---------------------//

}

[java] [view plain](http://blog.csdn.net/z971829916/article/details/22572587) [copy](http://blog.csdn.net/z971829916/article/details/22572587) [print?](http://blog.csdn.net/z971829916/article/details/22572587)[在CODE上查看代码片](https://code.csdn.net/snippets/264529)

@Entity

@Table(name="TB\_USER")

@SuppressWarnings("serial")

public class User implements Serializable {

    //主键id

    @Id

    private String id;

    //登录名称

    private String username;

    //登录密码

    private String password;

    //拥有能访问的资源/链接()

    private List<Resource> resourcesList = new ArrayList<Resource>();

    //-------------Getter/Setter-------------//

}

@Entity

@Table(name="TB\_USER")

@SuppressWarnings("serial")

public class User implements Serializable {

//主键id

@Id

private String id;

//登录名称

private String username;

//登录密码

private String password;

//拥有能访问的资源/链接()

private List<Resource> resourcesList = new ArrayList<Resource>();

//-------------Getter/Setter-------------//

}

[java] [view plain](http://blog.csdn.net/z971829916/article/details/22572587) [copy](http://blog.csdn.net/z971829916/article/details/22572587) [print?](http://blog.csdn.net/z971829916/article/details/22572587)[在CODE上查看代码片](https://code.csdn.net/snippets/264529)

@Repository

public class UserDao extends BasicHibernateDao<User, String> {

    public User getUserByUsername(String username) {

        return findUniqueByProperty("username", username);

    }

}

@Repository

public class UserDao extends BasicHibernateDao<User, String> {

public User getUserByUsername(String username) {

return findUniqueByProperty("username", username);

}

}

实现org.apache.shiro.realm.AuthorizingRealm中已经给出了两个抽象方法:

[java] [view plain](http://blog.csdn.net/z971829916/article/details/22572587) [copy](http://blog.csdn.net/z971829916/article/details/22572587) [print?](http://blog.csdn.net/z971829916/article/details/22572587)[在CODE上查看代码片](https://code.csdn.net/snippets/264529)

public class ShiroDataBaseRealm extends AuthorizingRealm{

    @Autowired

    private UserDao userDao;

    /\*\*

     \*

     \* 当用户进行访问链接时的授权方法

     \*

     \*/

    protected AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principals) {

        if (principals == null) {

            throw new AuthorizationException("Principal对象不能为空");

        }

        User user = (User) principals.fromRealm(getName()).iterator().next();

        //获取用户响应的permission

        List<String> permissions = CollectionUtils.extractToList(user.getResourcesList(), "permission",true);

        SimpleAuthorizationInfo info = new SimpleAuthorizationInfo();

        info.addStringPermissions(permissions);

        return info;

    }

    /\*\*

     \* 用户登录的认证方法

     \*

     \*/

    protected AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken token) throws AuthenticationException {

        UsernamePasswordToken usernamePasswordToken = (UsernamePasswordToken) token;

        String username = usernamePasswordToken.getUsername();

        if (username == null) {

            throw new AccountException("用户名不能为空");

        }

        User user = userDao.getUserByUsername(username);

        if (user == null) {

            throw new UnknownAccountException("用户不存在");

        }

        return new SimpleAuthenticationInfo(user,user.getPassword(),getName());

    }

}

public class ShiroDataBaseRealm extends AuthorizingRealm{

@Autowired

private UserDao userDao;

/\*\*

\*

\* 当用户进行访问链接时的授权方法

\*

\*/

protected AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principals) {

if (principals == null) {

throw new AuthorizationException("Principal对象不能为空");

}

User user = (User) principals.fromRealm(getName()).iterator().next();

//获取用户响应的permission

List<String> permissions = CollectionUtils.extractToList(user.getResourcesList(), "permission",true);

SimpleAuthorizationInfo info = new SimpleAuthorizationInfo();

info.addStringPermissions(permissions);

return info;

}

/\*\*

\* 用户登录的认证方法

\*

\*/

protected AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken token) throws AuthenticationException {

UsernamePasswordToken usernamePasswordToken = (UsernamePasswordToken) token;

String username = usernamePasswordToken.getUsername();

if (username == null) {

throw new AccountException("用户名不能为空");

}

User user = userDao.getUserByUsername(username);

if (user == null) {

throw new UnknownAccountException("用户不存在");

}

return new SimpleAuthenticationInfo(user,user.getPassword(),getName());

}

}

定义好了ShiroDataBaseRealm后修改applicationContext-shiro.xml文件

<bean id="chainDefinitionSectionMetaSource">

    <property name="filterChainDefinitions">

        <value>

            /login = authc

            /logout = logout

            /resource/\*\* = anon

        </value>

    </property>

</bean>

<bean id="shiroSecurityFilter">

    <property name="securityManager" ref="securityManager" />

    <property name="loginUrl" value="/login.jsp" />

    <property name="successUrl" value="/index.jsp" />

    <property name="unauthorizedUrl" value="/unauthorized.jsp" />

    <!-- shiro连接约束配置,在这里使用自定义的动态获取资源类 -->

    <property name="filterChainDefinitionMap" ref="chainDefinitionSectionMetaSource" />

</bean>

<bean id="shiroDataBaseRealm">

    <!-- MD5加密 -->

    <property name="credentialsMatcher">

        <bean>

            <property name="hashAlgorithmName" value="MD5" />

        </bean>

    </property>

</bean>

<bean id="securityManager">

    <property name="realm" ref="shiroDataBaseRealm" />

</bean>

<bean id="lifecycleBeanPostProcessor"/>

<bean id="chainDefinitionSectionMetaSource">

<property name="filterChainDefinitions">

<value>

/login = authc

/logout = logout

/resource/\*\* = anon

</value>

</property>

</bean>

<bean id="shiroSecurityFilter">

<property name="securityManager" ref="securityManager" />

<property name="loginUrl" value="/login.jsp" />

<property name="successUrl" value="/index.jsp" />

<property name="unauthorizedUrl" value="/unauthorized.jsp" />

<!-- shiro连接约束配置,在这里使用自定义的动态获取资源类 -->

<property name="filterChainDefinitionMap" ref="chainDefinitionSectionMetaSource" />

</bean>

<bean id="shiroDataBaseRealm">

<!-- MD5加密 -->

<property name="credentialsMatcher">

<bean>

<property name="hashAlgorithmName" value="MD5" />

</bean>

</property>

</bean>

<bean id="securityManager">

<property name="realm" ref="shiroDataBaseRealm" />

</bean>

<bean id="lifecycleBeanPostProcessor"/>

shiro EHcache 与 Spring EHcache集成

shiro CacheManager创建并管理其他Shiro组件使用的Cache实例生命周期。因为Shiro能够访问许多后台数据源，如：身份验证，授权和会话管理，缓存在框架中一直是一流的架构功能，用来在同时使用这些数据源时提高性能。任何现代开源和/或企业的缓存产品能够被插入到Shiro 来提供一个快速及高效的用户体验。

自从spring 3.1问世后推出了缓存功能后，提供了对已有的 Spring 应用增加缓存的支持，这个特性对应用本身来说是透明的，通过缓存抽象层，使得对已有代码的影响降低到最小。

该缓存机制针对于 Java 的方法，通过给定的一些参数来检查方法是否已经执行，Spring 将对执行结果进行缓存，而无需再次执行方法。

可通过下列配置来启用缓存的支持：

<beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:cache="http://www.springframework.org/schema/cache"

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

http://www.springframework.org/schema/cache http://www.springframework.org/schema/cache/spring-cache.xsd">

<!-- 使用缓存annotation 配置 -->

<cache:annotation-driven cache-manager="ehCacheManager" />

</beans>

@Cacheable和\*\*@CacheEvict\*\*来对缓存进行操作

@Cacheable：负责将方法的返回值加入到缓存中

@CacheEvict：负责清除缓存

/\*\*声明了一个名为 persons 的缓存区域，当调用该方法时，Spring 会检查缓存中是否存在 personId 对应的值。\*/

@Cacheable("persons")

public Person profile(Long personId) { ... }

/\*\*指定多个缓存区域。Spring 会一个个的检查，一旦某个区域存在指定值时则返回\*/

@Cacheable({"persons", "profiles"})

public Person profile(Long personId) { ... }

</code>

</pre>

<pre>

<code>

/\*\*清空所有缓存\*/

@CacheEvict(value="persons",allEntries=true)

public Person profile(Long personId, Long groundId) { ... }

/\*\*或者根据条件决定是否缓存\*/

@CacheEvict(value="persons", condition="personId > 50")

public Person profile(Long personId) { ... }

在shiro里面会有授权缓存。可以通过AuthorizingRealm类中指定缓存名称。就是authorizationCacheName属性。当shiro为用户授权一次之后将会把所有授权信息都放进缓存中。现在有个需求。当在更新用户或者删除资源和更新资源的时候，要刷新一下shiro的授权缓存，给shiro重新授权一次。因为当更新用户或者资源时，很有可能已经把用户本身已有的资源去掉。不给用户访问。所以。借助spring的缓存工厂和shiro的缓存能够很好的实现这个需求。

将applicationContext-shiro.xml文件添加缓存

<bean id="chainDefinitionSectionMetaSource">

<property name="filterChainDefinitions" >

<value>

/login = authc

/logout = logout

/resource/\*\* = anon

</value>

</property>

</bean>

<bean id="shiroSecurityFilter">

<property name="securityManager" ref="securityManager" />

<property name="loginUrl" value="/login.jsp" />

<property name="successUrl" value="/index.jsp" />

<property name="unauthorizedUrl" value="/unauthorized.jsp" />

<!-- shiro连接约束配置,在这里使用自定义的动态获取资源类 -->

<property name="filterChainDefinitionMap" ref="chainDefinitionSectionMetaSource" />

</bean>

<bean id="shiroDataBaseRealm">

<!-- MD5加密 -->

<property name="credentialsMatcher">

<bean>

<property name="hashAlgorithmName" value="MD5" />

</bean>

</property>

<property name="authorizationCacheName" value="shiroAuthorizationCache" />

</bean>

<bean id="securityManager">

<property name="realm" ref="shiroDataBaseRealm" />

<property name="cacheManager" ref="cacheManager" />

</bean>

<bean id="lifecycleBeanPostProcessor"/>

<!-- 使用缓存annotation 配置 --/>

<cache:annotation-driven cache-manager="ehCacheManager" />

<!-- spring对ehcache的缓存工厂支持 -->

<bean id="ehCacheManagerFactory">

<property name="configLocation" value="classpath:ehcache.xml" />

<property name="shared" value="false" />

</bean>

<!-- spring对ehcache的缓存管理 -->

<bean id="ehCacheManager">

<property name="cacheManager" ref="ehCacheManagerFactory"></property>

</bean>

<!-- shiro对ehcache的缓存管理直接使用spring的缓存工厂 -->

<bean id="cacheManager">

<property name="cacheManager" ref="ehCacheManagerFactory" />

</bean>

ehcache.xml

<?xml version="1.0" encoding="UTF-8"?>

<ehcache>

<!--

maxElementsInMemory为缓存对象的最大数目,

eternal设置是否永远不过期,

timeToIdleSeconds对象处于空闲状态的最多秒数,

timeToLiveSeconds对象处于缓存状态的最多秒数

-->

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

<cache name="shiroAuthorizationCache" maxElementsInMemory="300" eternal="false" timeToLiveSeconds="600" overflowToDisk="false"/>

</ehcache>

public class UserDao extends BasicHibernateDao<User, String> {

public User getUserByUsername(String username) {

return findUniqueByProperty("username", username);

}

@CacheEvict(value="shiroAuthorizationCache",allEntries=true)

public void saveUser(User entity) {

save(entity);

}

}

@Repository

public class ResourceDao extends BasicHibernateDao<Resource, String> {

@CacheEvict(value="shiroAuthorizationCache",allEntries=true)

public void saveResource(Resource entity) {

save(entity);

}

@CacheEvict(value="shiroAuthorizationCache",allEntries=true)

public void deleteResource(Resource entity) {

delete(entity);

}

}

当userDao或者reaourceDao调用了相应带有缓存注解的方法，都会将AuthorizingRealm类中的缓存去掉。那么就意味着 shiro在用户访问链接时要重新授权一次。