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。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>
```

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"
class="org.apache.shiro.spring.web.ShiroFilterFactoryBean">
    <!-- 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 -->
    <propery name="filterChainDefinitions">
        <value>
            /login = authc
            /logout = logout
            /resource/** = anon
        </value>
    </property>
</bean>

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

<bean id="lifecycleBeanPostProcessor"
class="org.apache.shiro.spring.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
class="org.springframwork.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"
depends-on="lifecycleBeanPostProcessor"/>

<bean
class="org.apache.shiro.spring.security.interceptor.AuthorizationAttributeSourceAdv
isor">
    <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:

```
@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--------------------//
}
```

```
@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

```
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

```
<bean id="chainDefinitionSectionMetaSource"
class="org.exitsoft.showcase.vcsadmin.service.account.ChainDefinitionSectionMetaSou
rce">

    <property name="filterChainDefinitions">
        <value>
            /login = authc
            /logout = logout
            /resource/** = anon
        </value>
    </property>
</bean>


<bean id="shiroSecurityFilter"
class="org.apache.shiro.spring.web.ShiroFilterFactoryBean">
    <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"
class="org.apache.shiro.web.mgt.DefaultWebSecurityManager">
</bean>

<bean id="lifecycleBeanPostProcessor"
class="org.apache.shiro.spring.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:

```
/**
 * 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:

```
@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_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-------------//
}
```

```
@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:

```
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"
class="org.exitsoft.showcase.vcsadmin.service.account.ChainDefinitionSectionMetaSou
rce">

    <property name="filterChainDefinitions">
```

```
        <value>
            /login = authc
            /logout = logout
            /resource/** = anon
        </value>
    </property>
</bean>


<bean id="shiroSecurityFilter"
class="org.apache.shiro.spring.web.ShiroFilterFactoryBean">
    <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"
class="org.exitsoft.showcase.vcsadmin.service.account.ShiroDataBaseRealm">
    <!-- MD5 加密 -->
    <property name="credentialsMatcher">
        <bean class="org.apache.shiro.authc.credential.HashedCredentialsMatcher">
            <property name="hashAlgorithmName" value="MD5" />
        </bean>
    </property>
</bean>

<bean id="securityManager"
class="org.apache.shiro.web.mgt.DefaultWebSecurityManager">
    <property name="realm" ref="shiroDataBaseRealm" />
</bean>

<bean id="lifecycleBeanPostProcessor"
class="org.apache.shiro.spring.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"
class="org.exitsoft.showcase.vcsadmin.service.account.ChainDefinitionSectionMetaSou
rce">
    <property name="filterChainDefinitions" >
        <value>
            /login = authc
            /logout = logout
            /resource/** = anon
        </value>
    </property>
</bean>


<bean id="shiroSecurityFilter"
class="org.apache.shiro.spring.web.ShiroFilterFactoryBean">
    <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"
class="org.exitsoft.showcase.vcsadmin.service.account.ShiroDataBaseRealm">
    <!-- MD5 加密 -->
    <property name="credentialsMatcher">
        <bean class="org.apache.shiro.authc.credential.HashedCredentialsMatcher">
            <property name="hashAlgorithmName" value="MD5" />
        </bean>
    </property>
    <property name="authorizationCacheName" value="shiroAuthorizationCache" />
```

```
</bean>

<bean id="securityManager"
class="org.apache.shiro.web.mgt.DefaultWebSecurityManager">
    <property name="realm" ref="shiroDataBaseRealm" />
    <property name="cacheManager" ref="cacheManager" />
</bean>

<bean id="lifecycleBeanPostProcessor"
class="org.apache.shiro.spring.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"
class="org.springframework.cache.ehcache.EhCacheManagerFactoryBean">
    <property name="configLocation" value="classpath:ehcache.xml" />
    <property name="shared" value="false" />
</bean>

<!-- spring 对 ehcache 的缓存管理 -->
<bean id="ehCacheManager"
class="org.springframework.cache.ehcache.EhCacheCacheManager">
    <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" class="org.apache.shiro.cache.ehcache.EhCacheManager">
    <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。

整个 apache shiro 的使用，vcs admin 项目和 vcs admin jpa 中都会有例子。可以参考 showcase/vcs_admin 或者 showcase/vcs_admin_jpa 项目中的例子去理解。。