**shiro学习**

# 1.Permissions

**Permission一般可以使用资源名称表示，权限是安全策略中的最低级别的结构，它们只明确定义应用程序可以执行的操作。**

**在一个系统中有的资源仅仅是分类信息，仅仅是查看，例如系统中某个大的模块包含许多小的模块，大模块仅仅是分类资源，做小资源的展示用。Permissions使用ResourceName表示。注意ResourceName不要重复，因为会这样会丢失资源信息。Shiro的Permission底层使用Set<String>实现使用实现。**

**有的系统资源具有丰富的操作，典型的如CURD。这时候Permissions表示为**

**ResourceName:Action1,Action2,…**

**如果授予所有操作权限使用**

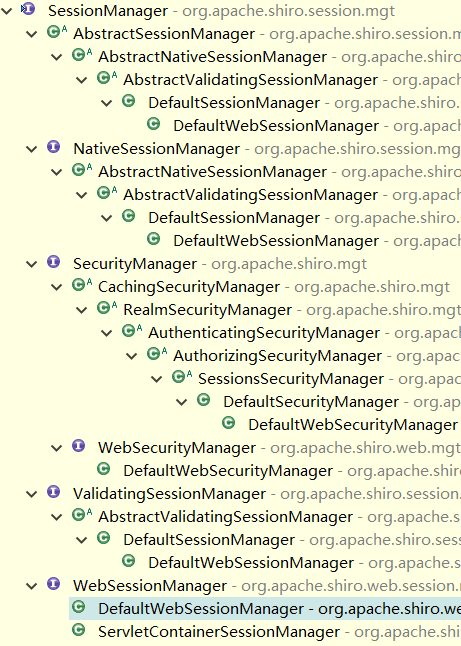
**ResourceName:\***

**还有就是对资源可以实现事例级别的资源控制，表示为**

**ResourceName:Action1:Instance1**

**用Shiro的专业术语表述 In this scenario（方案） you use three parts - the first is the *domain*, the second is the *action*(s), and the third is the instance(s) being acted upon。**

# 2.几个重要的类的关系



## 2.1 SecurityManager

SecurityManager的父接口是SessionManager。

### 2.1.1 web环境下的sessionManager的实现原理

在web 应用环境下使用DefaultWebSecurityManager。DefaultWebSecurityManager默认使用ServletContainerSessionManager管理session。当设置DefaultWebSecurityManager的sessionMode属性=native时，会初始化DefaultWebSessionManager对象，并使能DefaultWebSessionManager管理session，也就是独立于Web容器可在集群环境中使用的session管理器。这里的“Native”这个词意味着Shiro自己的企业会话管理实现将用于支持所有的Subject和HttpServletRequest会话，并完全绕过servlet容器。 但是请放心 - Shiro直接实现了Servlet规范的相关部分，因此任何现有的web / http相关代码都能按预期工作，并且从不需要“知道”Shiro透明地管理会话。

|  |
| --- |
| **public** **void** setSessionMode(String sessionMode) {  String mode = sessionMode;  **if** (mode == **null**) {  **throw** **new** IllegalArgumentException("sessionMode argument cannot be null.");  }  mode = sessionMode.toLowerCase();  **//当既不等于http又不等于native时抛异常**  **if** (!***HTTP\_SESSION\_MODE***.equals(mode) && !***NATIVE\_SESSION\_MODE***.equals(mode)) {  String msg = "Invalid sessionMode [" + sessionMode + "]. Allowed values are " +  "public static final String constants in the " + getClass().getName() + " class: '"  + ***HTTP\_SESSION\_MODE*** + "' or '" + ***NATIVE\_SESSION\_MODE*** + "', with '" +  ***HTTP\_SESSION\_MODE*** + "' being the default.";  **throw** **new** IllegalArgumentException(msg);  }  **//构造器已经初始化this.sessionMode=”http”,当mode=”native”返//回true**  **boolean** recreate = **this**.sessionMode == **null** || !**this**.sessionMode.equals(mode);  **this**.sessionMode = mode;  **if** (recreate) {  LifecycleUtils.*destroy*(getSessionManager());  **// 此处返回的sessionManager是DefaultWebSessionManager的实例**  SessionManager sessionManager = createSessionManager(mode);  **// 调用父类的父类SessionsSecurityManager的setSessionManager**  setSessionManager(sessionManager);  }  } |

当然SecurityManager可以注入自定义SessionManager，但是真的没有必要。它的注入过程还是调用父类的父类SessionsSecurityManager的setSessionManager（）

下面的讨论securityManager代表DefaultWebSecurityManager的实例。

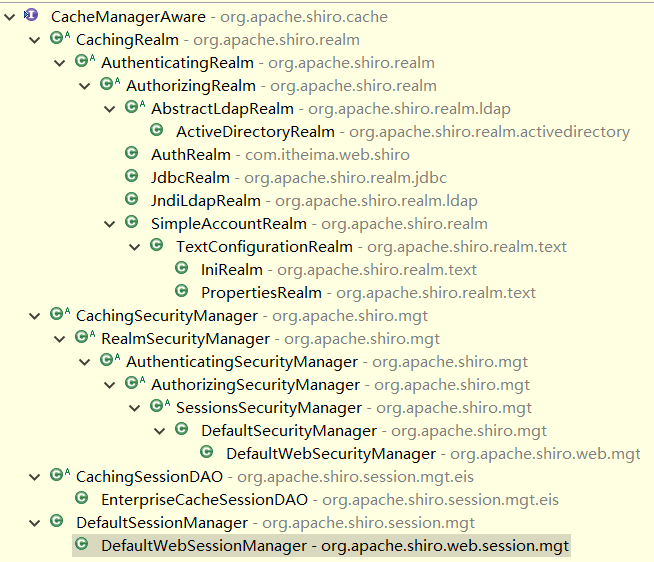
### 2.1.2 securityManager中注入CacheManager

会调用父类CachingSecurityManager的setCacheManager方法。这方法其实是重写父接口CacheManagerAware的方法。

2.1.3 向2.1.1中的sessionManager中注入CacheManager

源码是使用SessionsSecurityManager的applyCacheManagerToSessionManager方法。applyCacheManagerToSessionManager实现：

|  |
| --- |
| **protected** **void** applyCacheManagerToSessionManager() {  **if** (**this**.sessionManager **instanceof** CacheManagerAware) {  ((CacheManagerAware) **this**.sessionManager).setCacheManager(getCacheManager());  }  } |

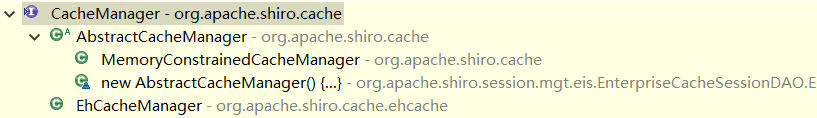


源码中的sessionManager是上面讲的注入的DefaultWebSessionManager的实例。DefaultWebSessionManager父接口也是CacheManagerAware，所以能注入cacheManager。

## 2.2 SessionManager

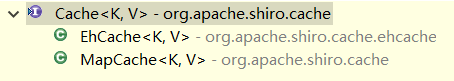
这里使用DefaultWebSessionManager，管理session的持久化，注入RedisSessionDao。securityManager.sessionManager = redisSessionDao

## 2.3 CacheManager



实现类使用开源的RedisCacheManager

## 2.4 Cache

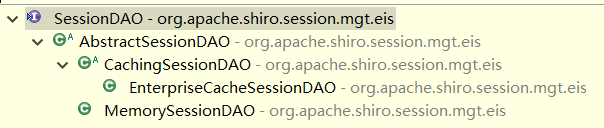


实现类使用RedisCache

## 2.5 CacheManagerAware

DefaultWebSessionManager和 CachingSecurityManager实现该接口的目的是，让sessionManager中注入cacheManager，向securityManager中注入cacheManager变得简单可行

## 2.6 SessionDao



实现类使用RedisSessionDao，继承AbstractSessionDAO。

<property name=*"sessionManager.sessionDao"* ref=*"redisSessionDAO"*/>

1.session 的id

2.native session 的cookie

|  |
| --- |
| <bean id="sessionIdCookie" class="org.apache.shiro.web.servlet.SimpleCookie">  <constructor-arg value="sid"/>  <property name="httpOnly" value="true"/>  <!-- 浏览器关闭cookie失效 -->  <property name="maxAge" value="-1"/>  </bean> |
| **public** **class** DefaultWebSessionManager **extends** DefaultSessionManager {  **private** **static** **final** Logger ***log*** = LoggerFactory.*getLogger*(DefaultWebSessionManager.**class**);  **private** Cookie sessionIdCookie;  **private** **boolean** sessionIdCookieEnabled;  **// ShiroHttpSession.*DEFAULT\_SESSION\_ID\_NAME=”JSESSIONID”***  **// 构造器设置初始化cookie实例，并设置name是JSESSIONID**  **public** DefaultWebSessionManager() {  **Cookie cookie = new SimpleCookie(ShiroHttpSession.*DEFAULT\_SESSION\_ID\_NAME*);**  cookie.setHttpOnly(**true**); //more secure, protects against XSS attacks  **// 这个初始化后的cookie就是sessionIdCookie**  **this**.sessionIdCookie = cookie;  **this**.sessionIdCookieEnabled = **true**;  } |

3.native session的过期时间在缓存中的控制

## 3.0认证信息和授权信息的存储

首先是向realm域中注入缓存cache。当spring中配置securityManager的<property name=*"realm"* ref=*"authRealm"*/>时生效。这是RealmSecurityManager的方法

|  |
| --- |
| /\* It is also called after setting one or more realms via the {@link #setRealm setRealm} or  \* {@link #setRealms setRealms} methods to allow these newly available realms to be given the cache manager  \* already in use.  \*/  **protected** **void** applyCacheManagerToRealms() {  CacheManager cacheManager = getCacheManager();  Collection<Realm> realms = getRealms();  **if** (cacheManager != **null** && realms != **null** && !realms.isEmpty()) {  **for** (Realm realm : realms) {  **// 这里的realm其实就是程序员自定义realm它的祖宗接口也是CacheManagerAware**  **if** (realm **instanceof** CacheManagerAware) {  ((CacheManagerAware) realm).setCacheManager(cacheManager);  }  }  }  } |

获取授权信息这是AuthorizingRealm的方法

|  |
| --- |
| **protected** AuthorizationInfo getAuthorizationInfo(PrincipalCollection principals) {  **if** (principals == **null**) {  **return** **null**;  }  AuthorizationInfo info = **null**;  **if** (***log***.isTraceEnabled()) {  ***log***.trace("Retrieving AuthorizationInfo for principals [" + principals + "]");  }  Cache<Object, AuthorizationInfo> cache = getAvailableAuthorizationCache();  **if** (cache != **null**) {  **if** (***log***.isTraceEnabled()) {  ***log***.trace("Attempting to retrieve the AuthorizationInfo from cache.");  }  Object key = getAuthorizationCacheKey(principals);  info = cache.get(key);  **if** (***log***.isTraceEnabled()) {  **if** (info == **null**) {  ***log***.trace("No AuthorizationInfo found in cache for principals [" + principals + "]");  } **else** {  ***log***.trace("AuthorizationInfo found in cache for principals [" + principals + "]");  }  }  }  **if** (info == **null**) {  // Call template method if the info was not found in a cache  info = doGetAuthorizationInfo(principals);  // If the info is not null and the cache has been created, then cache the authorization info.  **if** (info != **null** && cache != **null**) {  **if** (***log***.isTraceEnabled()) {  ***log***.trace("Caching authorization info for principals: [" + principals + "].");  }  Object key = getAuthorizationCacheKey(principals);  cache.put(key, info);  }  }  **return** info;  } |

设置密码比较器这是AuthenticatingRealm的方法

|  |
| --- |
| **public** **void** setCredentialsMatcher(CredentialsMatcher credentialsMatcher) {  **this**.credentialsMatcher = credentialsMatcher;  } |

存储认证信息到cache

|  |
| --- |
| private void cacheAuthenticationInfoIfPossible(AuthenticationToken token, AuthenticationInfo info) {  if (!isAuthenticationCachingEnabled(token, info)) {  log.debug("AuthenticationInfo caching is disabled for info [{}]. Submitted token: [{}].", info, token);  return;  }  Cache<Object, AuthenticationInfo> cache = getAvailableAuthenticationCache();  if (cache != null) {  Object key = getAuthenticationCacheKey(token);  cache.put(key, info);  log.trace("Cached AuthenticationInfo for continued authentication. key=[{}], value=[{}].", key, info);  }  } |