权限管理

权限管理的概述

权限管理要实现对用户访问系统的控制，按照安全规则或者安全策略控制用户可以访问而且只能访问自己被授权的资源。

只要有用户参与的系统一般都要有权限管理

RBAC权限管理的模型

RBAC模型（Role-Based Access Control：基于角色的访问控制）

Permission：权限

Role：角色

如果直接基于角色去做权限管理

```
//下面是一段伪代码
if(hasRole("商场管理员")){
    //添加商品
    //删除商品
    //更新商品
}
```

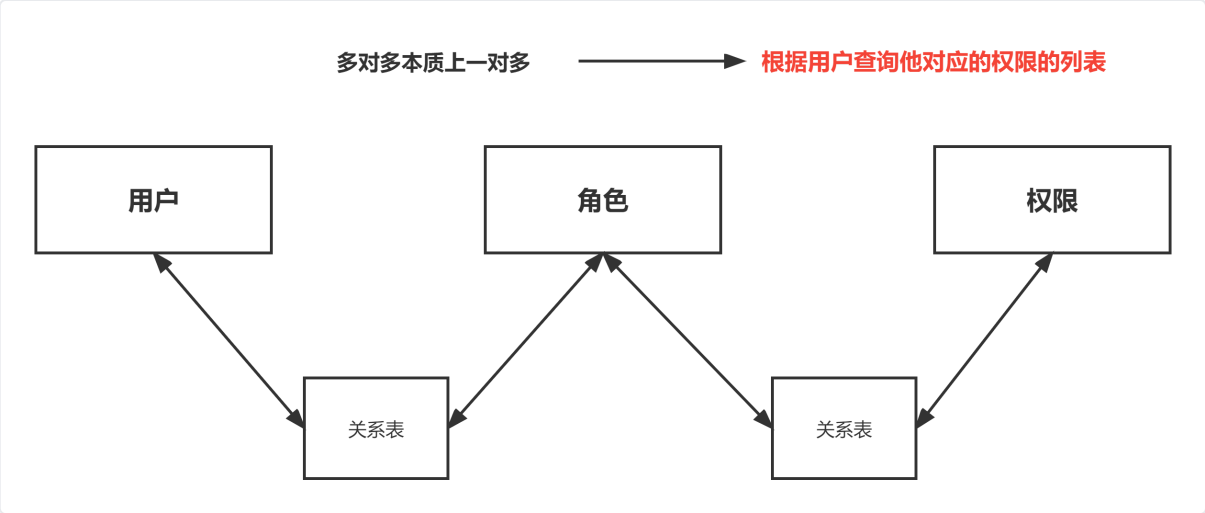
这时候会有一个问题，就是如果这个角色（商场管理员）对应的权限发生了变化，需要修改已有的业务代码(比如去掉删除商品的权限)，如下

```
//下面是一段伪代码
if(hasRole("商场管理员")){
    //添加商品
    //更新商品
}
```

而我们直接基于权限进行访问控制，可以避免权限的变化导致业务代码进行修改的情况

```
if(hasPermission("添加商品")){
    //添加商品
}
if(hasPermission("删除商品")){
    //删除商品
}
if(hasPermission("更新商品")){
    //更新商品
}
```

这时候用户-角色-权限三者之间存在着多对多的关系



实际上我们在开发的过程中用的最多的是**通过用户查询其对应的权限（一对多的查询）**

Shiro框架简介

Shiro安全框架，Apache Shiro是一个开源安全框架，提供身份认证、授权、密码学和会话管理。Shiro框架直观、易用，同时也能提供健壮的安全性。

Shiro是在Apache软件基金会孵化的的开源项目，2010年7月Shiro社区发布1.0版，同年9月，Shiro成为Apache软件基金会的顶级项目。

借助 Shiro 易于理解的 API，您可以快速轻松地保护任何应用程序。

当今虽然框架的环境发生了很大的变化，Shiro由于其**易用性、全面性、灵活性、拓展性以及对于Web支持**等一系列的特性，仍然是一个很不错的选择

Shiro的核心术语

Authentication，**认证**是身份验证的过程 -您正在尝试证明用户是他们所说的人。为此，用户需要提供系统理解和信任的某种身份证明。**认证其实就是我们通常所说的登录。**

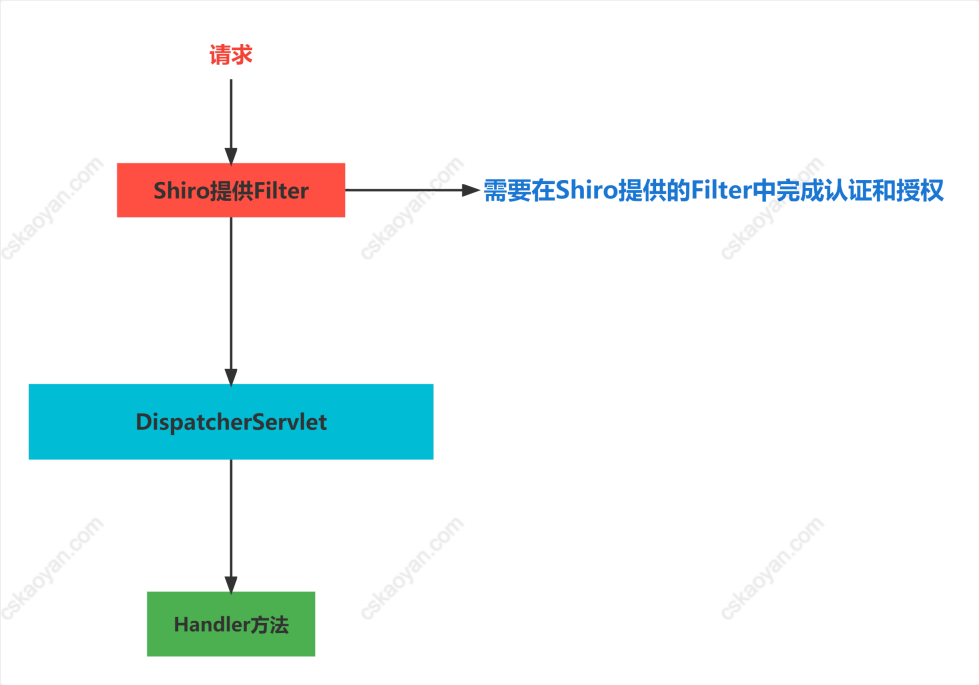
Authorization，**授权**或访问控制是指定对资源的访问权限的功能。换句话说，谁可以访问什么。比如是否允许用户编辑此数据（、查看此按钮、查看此网页➡这部分当前主要由前端来做了），这些都是决定用户有权访问的内容的决定。我们当前主要针对的是**URL级别**的权限访问控制

注意：认证是授权的基础，授权要在完成认证的前提下进行，也就是判断权限的时候会先检查认证是否已经成功完成

认证和授权的核心流程

我们在项目中整合Shiro框架，是在SpringMVC的框架的基础上进行整合。

这时候需要考虑一个问题就是Shiro和Handler方法之间的关系。Handler方法又是由SpringMVC的核心DispatcherServlet进行分发的，Shiro在应用程序中以Filter（JavaEE的核心组件）的形式存在，那么我们画出三者之间的关系如下



在Shiro提供的Filter中根据该请求是否需要认证和授权**Filter会配置作用范围，可以判断该请求是否需要认证和授权**，在完成认证和授权后，可以继续流程。

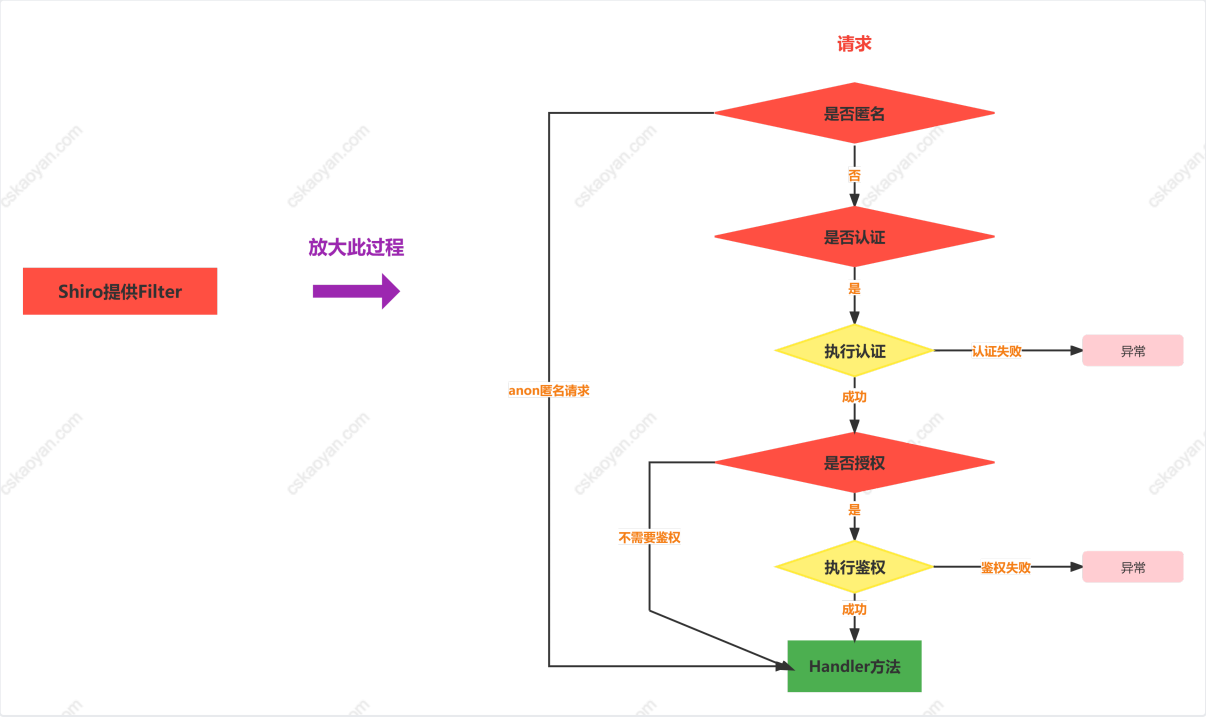
Shiro的Filter提供的Filter

anon、authc、perms是Shiro中最常用的几种不同类型的Filter

Filter名称	Filter类	说明
anon	org.apache.shiro.web.filter.authc.AnonymousFilter	匿名Filter，作用范围内的请求不需要认证和授权
authc	org.apache.shiro.web.filter.authc.FormAuthenticationFilter	认证Filter，作用范围内的请求需要完成认证
perms	org.apache.shiro.web.filter.authz.PermissionsAuthorizationFilter	权限Filter，作用范围内的请求需要完成认证和授权

完整的Filter可以查看这个这个官方文档[Apache Shiro Web Support | Apache Shiro]

那么我们就可以设置不同类型的Filter分别映射一些不同的URL范围，当请求发送到应用程序时，根据请求URL分别判断使用哪一些Filter，在Filter中决定是否继续访问流程



当某个请求发送过来的时候，会根据该请求的URL确定该请求要执行的Filter有哪些，然后在依次执行对应的Filter

比如设置

/admin/auth/login → 对应anon

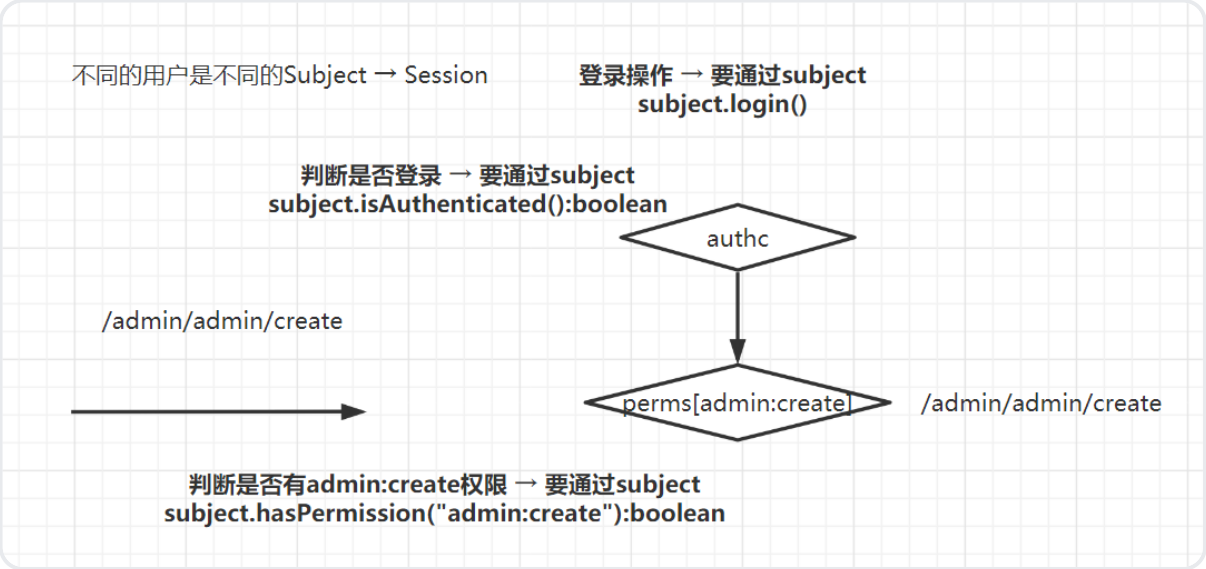
/admin/admin/list → 对应authc

当访问/admin/auth/login时对应的就是anon这个Filter

当访问/admin/admin/list时对应的就是authc这个Filter

认证的核心术语

Subject，主体，在Shiro中所做的几乎所有操作都基于当前正在执行的用户**也就是基本上Shiro的操作都是使用Subject操作的，Subject指的就是当前操作的用户**。在代码中的任何位置都可以轻松获得Subject，通过Subject可以方便的操作Shiro。比如我们可以使用Subject提供的方法来执行认证、判断是否认证等操作



Principals，主体鉴定后的参数**也就是认证后的用户信息**，可以是姓名、用户id、用户对象等形式

- 它是可以通过Subject来获得→ subject.getPrincipals();

Credentials，用来验证身份的秘密数据，通常指密码，生物数据**比如指纹、面部、瞳孔等**

Realms，域或领域，安全的特殊数据存储对象（DAO），Shiro中的Realm主要是让你能够获得对应的认证信息和授权信息

Token，令牌，Shiro中的Token是作为登录操作的参数，subject.login(token)

授权的核心术语

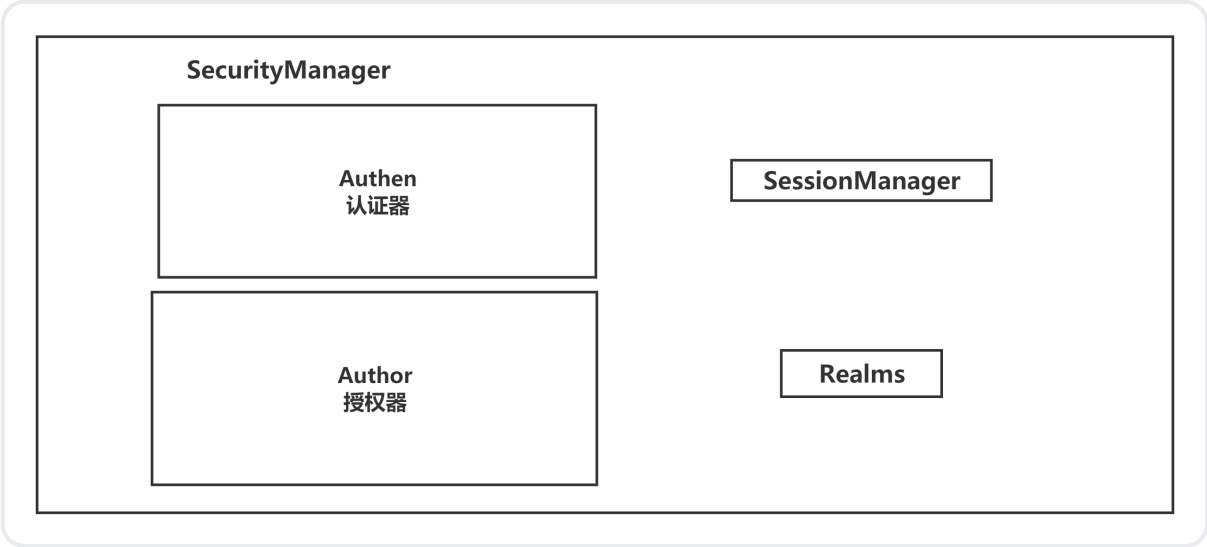
Subject、Principals和Realms同上

Shiro的核心组件

Shiro的核心组件是**SecurityManager**，安全管理器

- Authenticator 认证器

- Authorizer 授权器
- SessionManager 会话管理器
- Realm 域
- CacheManager 缓存管理器
- ~~Cryptography 密码学~~



Realm

安全的特殊数据存储对象（DAO），Shiro中的Realm主要是让你能够获得**对应的认证信息和授权信息**。

Collection realms 是作为SecurityManager中的成员变量。

使用Shiro的过程中，会提供默认的Realm → IniRealm，这个Realm获得认证信息和授权信息是通过加载ini文件来获得，灵活性很差，已经是过时的内容了。

我们提供自定义的Realm，需要继承一个抽象类**AuthorizingRealm**，需要实现该类中的两个抽象方法

- doGet**Authen**ticationInfo → 该方法获得**认证**信息
- doGet**Author**izationInfo → 该方法获得**授权**信息

```
1 public abstract class AuthorizingRealm extends AuthenticatingRealm
   implements Authorizer, Initializable, PermissionResolverAware,
  RolePermissionResolverAware {
2     protected abstract AuthorizationInfo
  doGetAuthorizationInfo(PrincipalCollection var1);
3 }
4 public abstract class AuthenticatingRealm extends CachingRealm
  implements Initializable {
5     protected abstract AuthenticationInfo
  doGetAuthenticationInfo(AuthenticationToken var1) throws
  AuthenticationException;
6 }
```

我们在这两个方法中分别去写我们获得用户**认证信息和授权信息的业务代码**

Authenticator

认证器，Shiro执行认证的话，使用SecurityManager中的认证器Authenticator提供的方法，Shiro提供了默认的认证器是**ModularRealmAuthenticator**

> Subject.login → SecurityManager.login → SecurityManager.authenticate → **Authenticator.authenticate**

认证器在执行认证的过程中会使用到Realm来获得认证信息**也就是上面realm中的doGetAuthenticationInfo**

Authenticator和Realm之间的关系就是，认证器中包含多个Realms

```
1 public class ModularRealmAuthenticator extends AbstractAuthenticator
  {
2     private Collection<Realm> realms;
3 }
```

默认的认证器**ModularRealmAuthenticator**中已经提供好了认证的所有过程的方法，这里的方法不需要我们开发，并且其中的realms成员变量默认由SecurityManager默认提供。

> 如果你想要自定义认证相关方法

可以继承ModularRealmAuthenticator，然后重写父类的方法

如果你想要自定义realm

- 可以修改提供给SecurityManager的realms
- 可以修改提供给Authenticator的realms

- SecurityManager会包含Realms
- Authenticator会包含Realms
- SecurityManager会包含Authenticator，Authenticator默认的Realms从SecurityManager中来

Authorizer

授权器，Shiro执行权限判断的话，需要使用到Authorizator提供的方法做判断，Shiro提供了默认的认证器是**ModularRealmAuthorizer**

- 根据角色信息判断 → 比如hasRole方法
- 根据权限信息判断 → 比如hasPermission方法，我们直接根据权限判断，可以使系统更灵活

以下的描述其实和上面在认证器中的描述基本是一致的，只不过认证变成了授权，Authen变成了Author

认证器在执行授权的过程中会使用到Realm来获得授权信息**也就是上面realm中的doGetAuthorizationInfo**

Authorizer和Realm之间的关系就是，授权器中包含多个Realms

```
public class ModularRealmAuthorizer extends AbstractAuthenticator {
    private Collection<Realm> realms;
}
```

默认的认证器**ModularRealmAuthorizer**中已经提供好了授权的所有过程的方法，这里的方法不需要我们开发，并且其中的realms成员变量默认由SecurityManager默认提供。

-

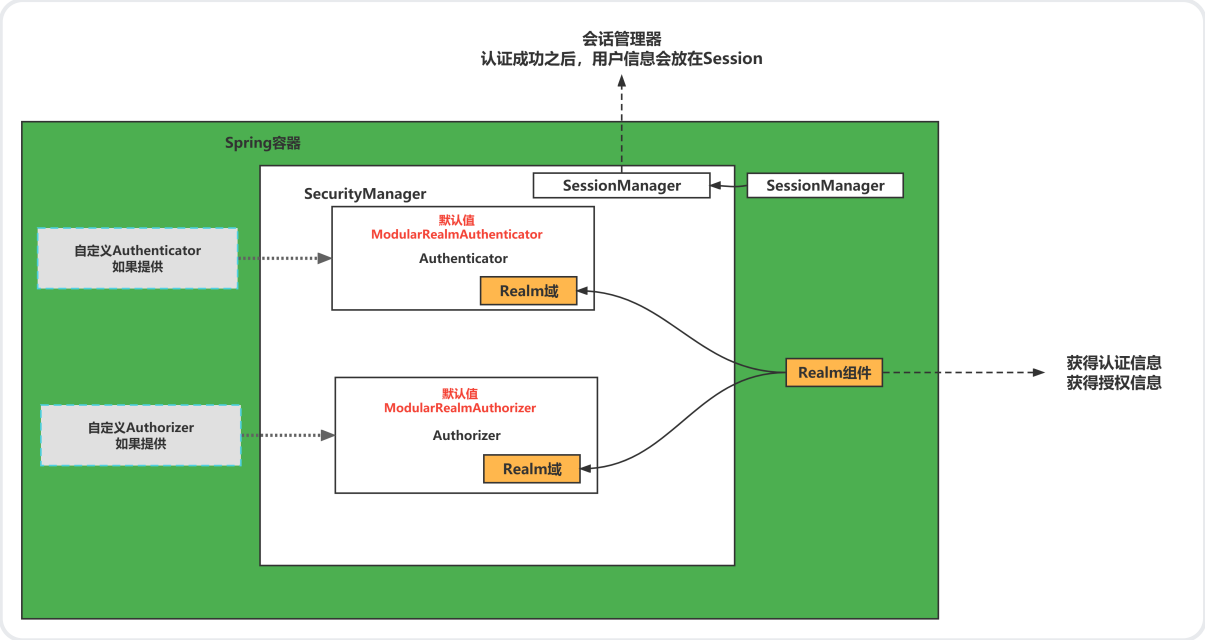
SessionManager

会话管理器，负责Shiro使用过程中的Session会话管理，如果需要对会话做管理，可以使用SessionManager。Shiro对web应用的支持使用的是**DefaultWebSessionManager**，其中方法都是可以直接使用，可以自己来配置其中的一些参数。如果需要对功能进行拓展，可以继承该类，重写其中的方法。

比如当你需要保证多个请求Session一致的时候，可以使用Shiro提供的会话管理器，对其进行改造。

Spring容器管理

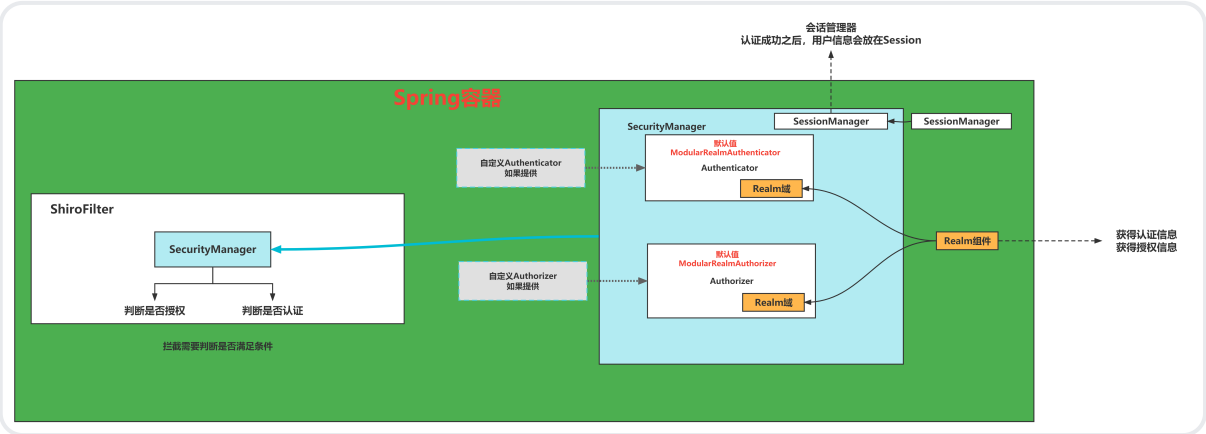
Shiro中的核心组件SecurityManager管理了很多对象，这些对象是Shiro运行过程中必须的对象，如果我们使用Spring容器管理这些组件



SecurityManager和ShiroFilter之间的关系

Shiro的Filter提供拦截的功能，而拦截功能的实现需要使用到SecurityManager提供的方法，也就是ShiroFilter依赖于SecurityManager。

通过ShiroFilter也可以作为Spring容器中的组件，我们可以通过Spring容器维护组件之间的依赖关系



Shiro的配置

我们可以在配置类中完成对应的组件注册

依赖shiro-spring

```
<dependency>
    <groupId>org.apache.shiro</groupId>
    <artifactId>shiro-spring</artifactId>
    <version>1.7.1</version>
</dependency>
```

Realm

继承AuthorizingRealm，并且注册为容器中的组件，重写里面的**doGetAuthenticationInfo**和**doGetAuthorizationInfo**方法

- doGet**Authen**ticationInfo → 根据token中的用户名查询该用户在系统中的Credentials，并且构造AuthenInfo
- doGet**Author**izationInfo → 根据Principal（放入AuthenInfo中的第一个参数）查询该用户在系统中的权限信息

```
1  /**
2   * @author stone
3   */
4  @Component
5  public class CustomRealm extends AuthorizingRealm {
6      //通常把doGetAuthenticationInfo方法放在上面
7
8      //该方法的形参 → 来源于subject的login方法
9      // 传入该值，为了根据用户名查询到该用户在系统中的密码（数据库中维护） → 来构造认证信息
10     @Override
11     protected AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken authenticationToken) throws AuthenticationException {
12         UsernamePasswordToken token = (UsernamePasswordToken) authenticationToken;
13         String username = token.getUsername();
14
15         //根据username查询数据库中对应password
16         String password = selectPasswordByUsername(username);
17
18         // principal信息 → 当前放的是什么信息，后续取出的就是什么信息
19         // 密码    → 该密码会和Token中的password做比较
20         // realm名称 → 没啥用
21         return new SimpleAuthenticationInfo(username,password,getName());
22     }
23
```

```
24      @Override
25      protected AuthorizationInfo
    doGetAuthorizationInfo(PrincipalCollection principalCollection) {
26          // 根据用户信息拿到所有的权限（数据库）
27          // doGetAuthenticationInfo方法返回值的第一个参数就是用户信息
28          // 在第21行代码放入的是字符串类型的Principal信息，取出的时候就可以以
    字符串格式取出
29          String primaryPrincipal = (String)
    principalCollection.getPrimaryPrincipal();
30          List<String> permissions =
    getPermissionsByUsername(primaryPrincipal);
31
32          SimpleAuthorizationInfo simpleAuthorizationInfo = new
    SimpleAuthorizationInfo();
33          simpleAuthorizationInfo.addStringPermissions(permissions);
34          return simpleAuthorizationInfo;
35      }
36
37      private String selectPasswordByUsername(String username) {
38          // 应该通过MyBatis根据用户名查询获得结果
39
40          return "123456";
41      }
42
43      private List<String> getPermissionsByUsername(String username) {
44          // 应该通过MyBatis根据用户名查询获得结果
45          return Arrays.asList("admin:user:list", "admin:admin:list");
46      }
47
48  }
```

SessionManager

如果没有在发送请求是配置withCredentials：true，那么跨域请求过程中Session会发生变化

我们当前项目中没有提供这个配置，那么跨域请求过程中会发生Session的变化。我们可以通过SessionManager会话管理器解决跨域场景下的Session变化问题。

为啥我们需要保证Session一致呢，原因是因为我们在完成认证和授权后，认证和授权的状态，以及用户的信息等内容，都是在Session中维护了，如果不保证Session一致，每一次访问请求都是一个新的Session，每一次都是未认证状态。

后端在执行登录（认证）后，向前端响应的结果中包含了一个值**SessionId**，前端在构造请求中携带了一个请求头，发送请求时通过该请求头携带SessionId信息，我们使用SessionManager就是要处理该请求携带的特定的请求头，该**请求头的值就是SessionId**

Shiro中提供的DefaultWebSessionManager中提供了一个方法叫getSessionId方法，我们需要继承该类，重新getSessionId方法

假设前后端协商的特定请求头为：**X-CskaoyanMarket-Admin-Token**

那么配置如下

```
/**
 * 跨域场景下Session会发生变化，保证Session不变
 * 认证完成之后，把SessionId作为响应结果响应给前端，前端发送请求，携带了SessionId
 * 通过请求头携带了SessionId信息
 *
 * 会话管理器要处理通过请求头获得SessionId这个过程
 * @author stone
 */
@Component
public class CustomShiroSessionManager extends DefaultWebSessionManager {

    private static final String HEADER = "X-CskaoyanMarket-Admin-Token";

    @Override
    protected Serializable getSessionId(ServletRequest servletRequest, ServletResponse response) {
        HttpServletRequest request = (HttpServletRequest) servletRequest;
```

```
        String sessionId = request.getHeader(HEADER);
        if (sessionId != null && !"".equals(sessionId)) {
            return sessionId;
        }
        return super.getSessionId(servletRequest, response);
    }
}
```

SecurityManager

SecurityManager需要将上面注册的Realm和SessionManager管理起来。如果没有指定Authenticator和Authorizer的话，采用默认的认证器ModularRealmAuthenticator和授权器ModularRealmAuthorizer

```
// SecurityManager
@Bean
public DefaultWebSecurityManager securityManager(CustomRealm customRealm,

CustomShiroSessionManager shiroSessionManager) {
    DefaultWebSecurityManager securityManager = new DefaultWebSecurityManager();
    // 设置认证器，如果没有设置，则采用默认的认证器 → ModularRealmAuthenticator
    // 多账号管理体系 → 设置自定义的认证器
    //securityManager.setAuthenticator();
    // 设置授权器，如果没有设置，则采用默认的授权器 → ModularRealmAuthorizer
    //securityManager.setAuthorizer();
    // 给SecurityManager设置Realm信息 → 给默认的认证器和授权器设置Realm信息
    securityManager.setRealm(customRealm);
    //如果是多个realm，则使用setRealms方法
    //securityManager.setRealms();

    //securityManager.setSessionManager(shiroSessionManager);
    return securityManager;
```

```
}
```

ShiroFilter

我们通过FactoryBean的形式注册ShiroFilter，在这里我们使用的是ShiroFilterFactoryBean

```
public class ShiroFilterFactoryBean implements FactoryBean, BeanPostProcessor {
    private AbstractShiroFilter instance;
    public Object getObject() throws Exception {
        if (this.instance == null) {
            this.instance = this.createInstance();
        }

        return this.instance;
    }
}
```

AbstractShiroFilter是Shiro提供的Filter，该Filter实现了OncePerRequestFilter，SpringBoot对于Filter的支持，只需要将其注册到容器中即可。

```
public abstract class AbstractShiroFilter extends OncePerRequestFilter {}
```

ShiroFilter依赖于SecurityManager，另外ShiroFilter需要配置Shiro提供的Filter和请求URL之间的映射关系，这里存在着一个Filter链，这里的Filter链是有序的，我们最终使用LinkedHashMap来维护映射关系和顺序

```
// ShiroFilter → ShiroFilterFactoryBean
@Bean
public ShiroFilterFactoryBean shiroFilter(DefaultWebSecurityManager securityManager) {
    ShiroFilterFactoryBean shiroFilterFactoryBean = new ShiroFilterFactoryBean();
    shiroFilterFactoryBean.setSecurityManager(securityManager);

```

```
7     LinkedHashMap<String, String> filterChainDefinitionMap = new
  LinkedHashMap<>();
8     // /admin/auth/login这个url对应着 anon Filter → 匿名Filter
9     filterChainDefinitionMap.put("/admin/auth/login", "anon");
10    filterChainDefinitionMap.put("/admin/auth/info", "anon");
11    // admin开头的请求，都要通过认证Filter
12    filterChainDefinitionMap.put("/admin/**", "authc");
13    // 权限的配置，可以将url和对应的权限建立映射关系
14    // 还可以通过注解的方式来建议url和权限之间的映射关系 → Advisor、
  @RequiresPermission（Handler方法上）
15    // url和handler方法对应、权限和Handler方法对应 → url和权限对应
16    filterChainDefinitionMap.put("/admin/admin/list",
  "perms[admin:admin:list]");
17
18
   shiroFilterFactoryBean.setFilterChainDefinitionMap(filterChainDefin
  itionMap);
19
20    return shiroFilterFactoryBean;
21 }
```

认证的业务

在对应的请求中执行Subject提供的login方法

```
1  public interface Subject {
2      void login(AuthenticationToken var1) throws
   AuthenticationException;
3  }
```

我们需要关注以下几点：

- 该登录请求是匿名请求，如果不是匿名请求，则无法执行到Subject提供的login方法
  - 比如我们当前项目中后台管理部分，登录请求是/admin/auth/login

    - 那么我们在配置ShiroFilter的时候，配置其映射关系为filterChainDefinitionMap.put("/admin/auth/login", "anon")

- Subject对象如何获得?
    - Shiro对Spring的支持中，在容器中的组件中的方法中都可以直接获得Subject
    - 需要使用这样的代码：Subject subject = SecurityUtils.getSubject();
- login方法中的参数AuthenticationToken是什么?
    - 该Token是登录主体Subject执行认证过程中传入系统的参数，其实就是大家绝大部分场景下构造的username和password
    - 我们可以使用其实现类UsernamePasswordToken： subject.login(new UsernamePasswordToken(username,password));
- sessionId需要作为响应结果的一部分，sessionId如何获得?
    - Session可以直接通过Subject提供的方法直接获得
    - 获得SessionId： subject.getSession().getId()

```
@PostMapping("login")
public BaseRespVo<LoginUserData> login(@RequestBody Map map) {
    String username = (String)map.get("username");
    String password = (String)map.get("password");

    // 整合Shiro
    // 获得操作的主体
    Subject subject = SecurityUtils.getSubject();
    // login方法传入的参数AuthenticationToken → 认证的令牌
    // subject执行login → 认证器执行认证方法 → realm.doGetAuthenticationInfo
    // AuthenticationToken → UsernamePasswordToken → 直接封装了username和password
    // username和password通过Handler方法的形参传入
    subject.login(new UsernamePasswordToken(username,password));

    if (subject.isAuthenticated()) {
        System.out.println("认证成功");
    }

    LoginUserData loginUserData = new LoginUserData();
    AdminInfoBean adminInfo = new AdminInfoBean();
```

```
21      adminInfo.setAvatar("https://wpimg.wallstcn.com/f778738c-e4f8-
    4870-b634-56703b4acafe.gif");
22      adminInfo.setNickName("admin123");
23      loginUserData.setAdminInfo(adminInfo);
24      // 携带SessionId信息
25      loginUserData.setToken((String) subject.getSession().getId());
26      return BaseRespVo.ok(loginUserData);
27  }
```

认证后获得用户信息

认证后用户信息通过Subject来获得，而获得的用户信息是在认证过程中在获得认证信息时放入的。也就是Realm的doGetAuthorizationInfo中的返回值的第一个参数。

```
1  if (subject.isAuthenticated()) {
2      //在已经认证成功的情况下，可以获得用户信息
3      // 获得的用户信息的来源 → 来源realm的doGetAuthenticationInfo方法的返回
    值的第一个参数
4      Object primaryPrincipal =
    subject.getPrincipals().getPrimaryPrincipal();
5      System.out.println(primaryPrincipal);
6
7  }
```

我们在开发一些接口的时候，请求参数并没有传入用户信息，而我们又需要通过用户信息完成一定的业务，那么这时候我们就可以通过Shiro来获得用户信息。比如日志管理时记录执行操作的用户、购物车、足迹、收藏等。

登出

可以直接使用Subject提供的logout方法

```
1  subject.logout();
```

Handler方法与权限

我们在前面如果想要将请求URL和权限绑定，我们是配置了FilterChainDefinitionMap

```
filterChainDefinitionMap.put("/admin/admin/list",
"perms[admin:admin:list]");
```

但上面的方式还是比较繁琐的，对于我们找到对应的Handler方法的过程也比较繁琐

这里Shiro提供了使用AspectJ的Advisor的方式，可以直接将URL和权限绑定起来，通过注解加在Handler方法上，将注解中包含的权限和Handler方法映射的URL绑定起来

- 引入aspectjweaver依赖
- 注册Advisor
- 使用注解

```
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
</dependency>
```

需要向容器中注册Advisor组件，并且提供SecurityManager给它

```
// 用到AspectJ → 使用注解的方式，将权限和url绑定起来
@Bean
public AuthorizationAttributeSourceAdvisor
authorizationAttributeSourceAdvisor(DefaultWebSecurityManager
securityManager) {
    AuthorizationAttributeSourceAdvisor
authorizationAttributeSourceAdvisor = new
AuthorizationAttributeSourceAdvisor();

 authorizationAttributeSourceAdvisor.setSecurityManager(securityManag
er);
    return authorizationAttributeSourceAdvisor;
}
```

5dbea268-e7ef-413e-a6c4-47743508fe29

@RequiresPermissions和@RequestMapping绑定了同一个Handler方法，而这两个注解分别提供的是权限和映射的URL，通过这种方式将URL和权限绑定起来，当访问该URL的请求时，要先判断是否拥有对应的权限。

注解的value属性是字符串数组：该url可以绑定多个权限，多个权限之间的关系 → 由logical属性决定，默认值是and

```
@RequiresPermissions(value = {"admin:user:list","songge"},logical = Logical.OR)
@RequestMapping("admin/user/list")
public BaseRespVo userList(String username,BaseParam param) {
    UserData userData = userService.query(param,username);
    return BaseRespVo.ok(userData);
}
```