Shiro框架

1. 安全操作引入

对于不同的用户，需要有对应的权限进行操作

不然所有的用户拥有相同的权限，不利于一个系统的管理

1.1. 基于权限管理的模型—RBAC模型

role-based Access control 基于角色的访问控制

对于一个用户，他需要一些权限来使用系统中的功能。

而这些权限，需要进行一个管理，那么该如何进行管理——可以考虑角色来进行管理，不同的角色拥有不同的权限，对不同的用户授予不同的角色，可以使用户获得不同的权限

1.2. 在开发中如何实现该模型

1.2.1. 方法一：通过授予用户角色实现（不提倡）

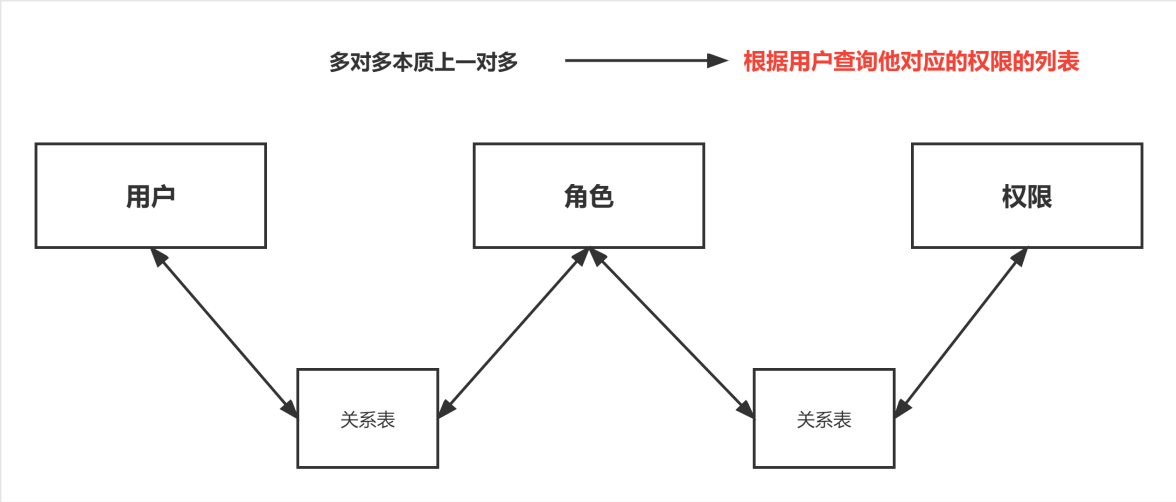
```
//下面是一段伪代码
if(hasRole("商场管理员")){
    //添加商品
    //删除商品
    //更新商品
}
```

如果商场管理员角色中的权限发生变动，需要去调整对应的代码，违反了开闭原则

1.2.2. 方法二：通过授予用户权限实现（使用该方式）

```
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
}
```

如果用户的权限发生变化，无需对代码进行变更，



上图给出了用户—角色—权限之间的关系，实际上用户和权限之间是一对多的关系（用户和角色之间是一对多，角色和权限之间是一对多）

2. Shiro框架的概念

在引入中提到了用户和权限之间的关系，那么Shiro框架就是用来管理这两者。对于不同的用户进行认证和授权，从而实现对应用程序的安全管理。

2.1. 这里提到了两个关键术语：认证和授权

- 认证：**Authen**tication
  - 验证身份信息，确认用户是程序所需要的人，为此，用户需要提供可供程序进行校验的信息。认证可以理解为登录
- 授权：**Author**ization
  - 指定通过认证的人可以拥有的访问权限。比如超级管理员可以对程序进行所有的操作，而商场管理员只能进行对商城的操作

当前项目主要针对的是**URL级别**的权限访问控制

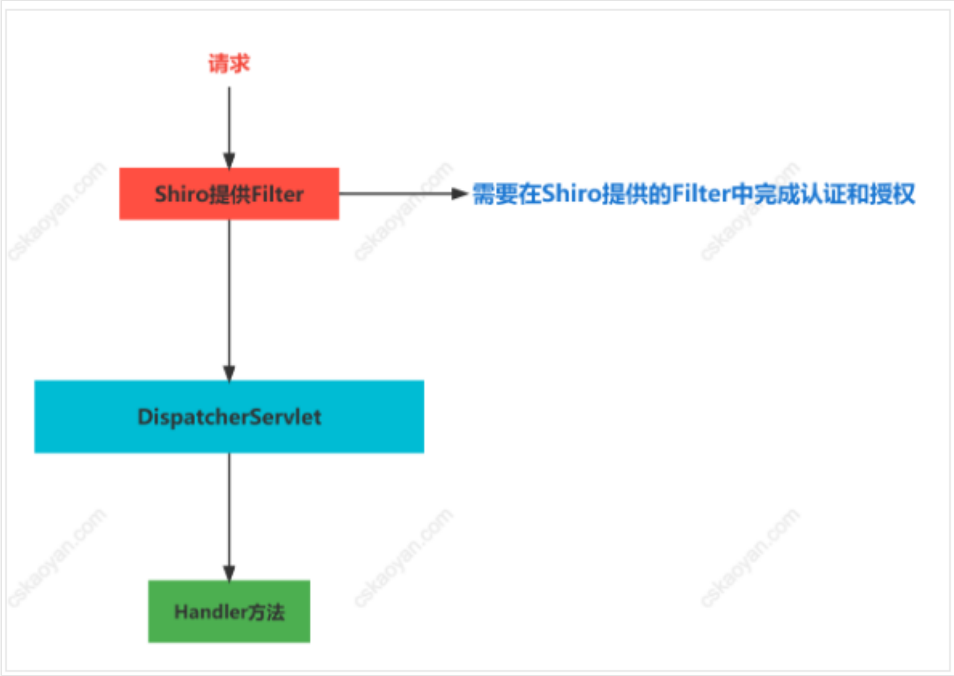
始终牢记一点：认证是授权的基础，授权要在完成认证的前提下进行，也就是判断权限的时候会先检查认证是否已经成功完成

2.2. 完成认证和授权的时间

由于是在SpringMVC的框架上整合Shiro框架，需要考虑认证和授权与handler方法的执行顺序。

因为Shiro是用于安全处理，在完成认证和授权的时间必然在进入Handler方法分发之前，只有在完成认证和授权之后才能进入handler方法中。

当然，handler方法是通过DispatcherServlet对象的doDispatcher方法分发进入，并且认证和授权在Shiro中是以Filter的形式存在，因此有如下的关系图



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。

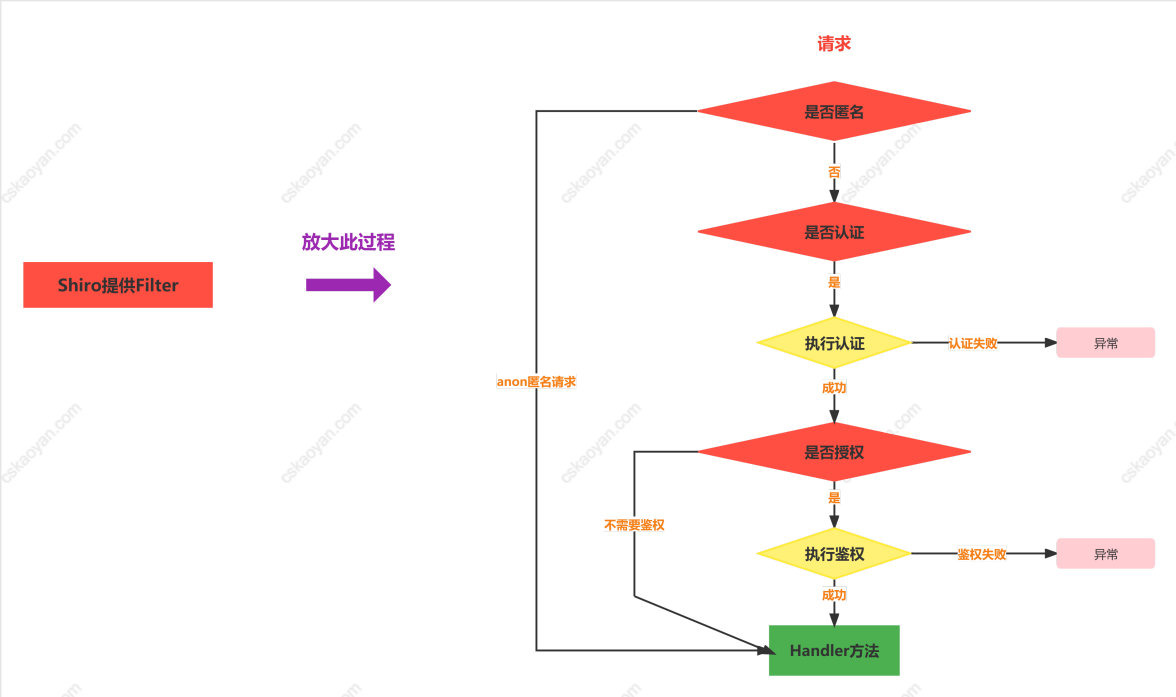
2.2.1. 再次，需要了解Shiro提供的Filter

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

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



上图中每个红色菱形都代表这一个Filter，这些filter连接起来执行

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

/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

2.3. 认证和授权中的核心术语

2.3.1. 两者共有的术语

- **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
  - 通过SecurityUtils获得
- **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
- **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)

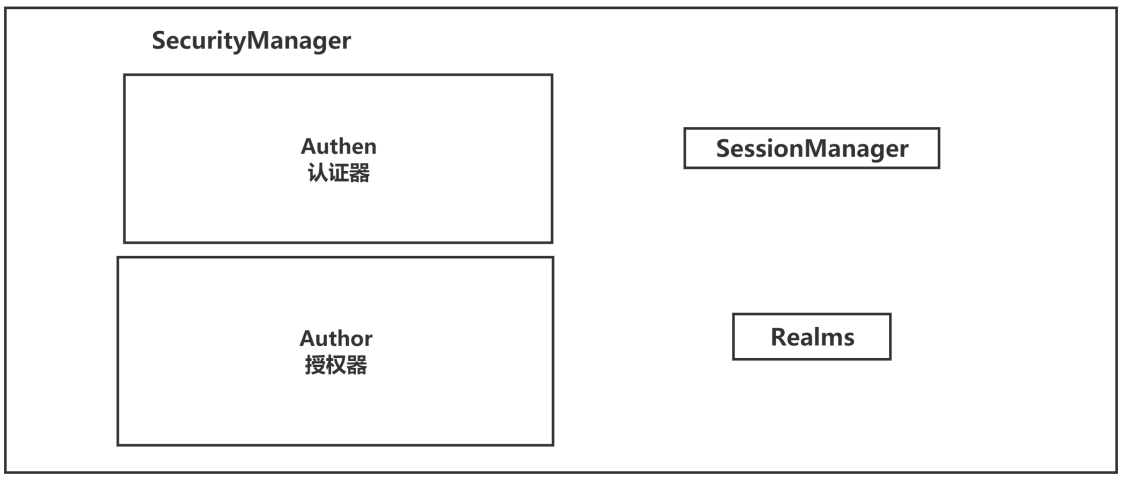
2.3.2. 认证独有的术语

- **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

3. 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~~，不用管它知道就行，因为该密码学只能在Shiro中使用，灵活性差



3.1. Realm

- 是一个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。~~

3.1.1. 使用自定义的Realm

- 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

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

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**

3.2. Authenticator

3.2.1. 概念

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**

3.2.2. 认证的执行流程如下：

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

最后在Authenticator.authenticate方法中执行doGet**Authen**ticationInfo方法（即在Realms中自我重写的认证方法）

也就是所，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

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

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

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。

3.2.4. 自定义认证方法（不推荐）

> 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

3.2.5. 注意：

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

3.3. Authorizer

3.3.1. 概念

- 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

3.3.2. 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。

3.4. SessionManager

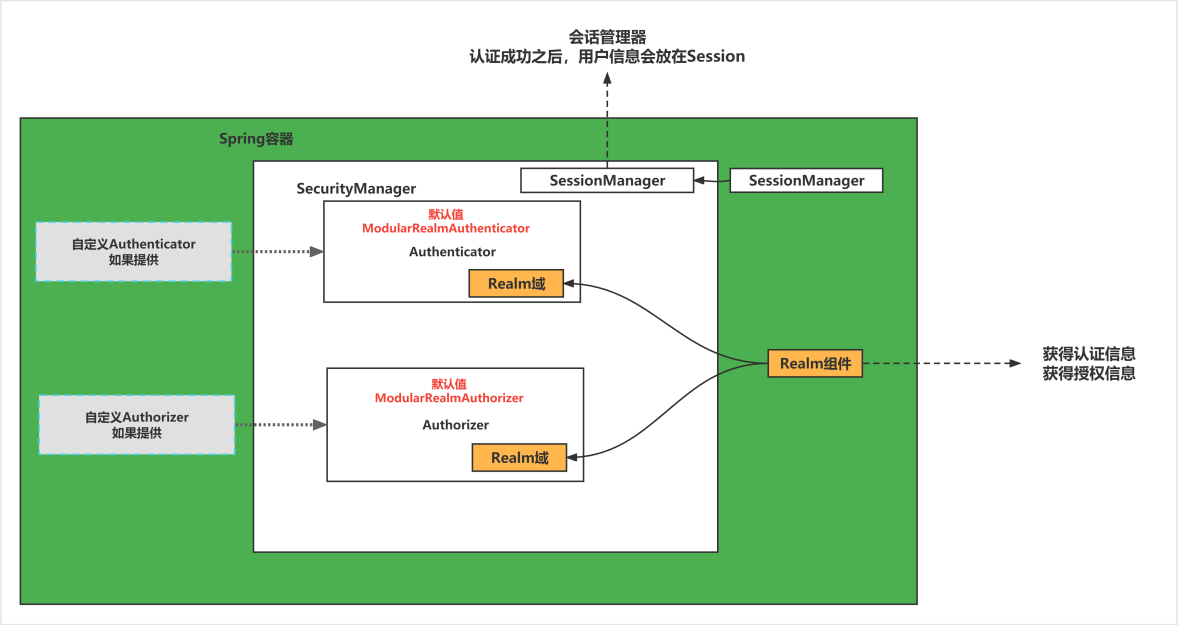
会话管理器，负责Shiro使用过程中的Session会话管理，如果需要对会话做管理，可以使用SessionManager。

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。

3.5. Spring容器管理

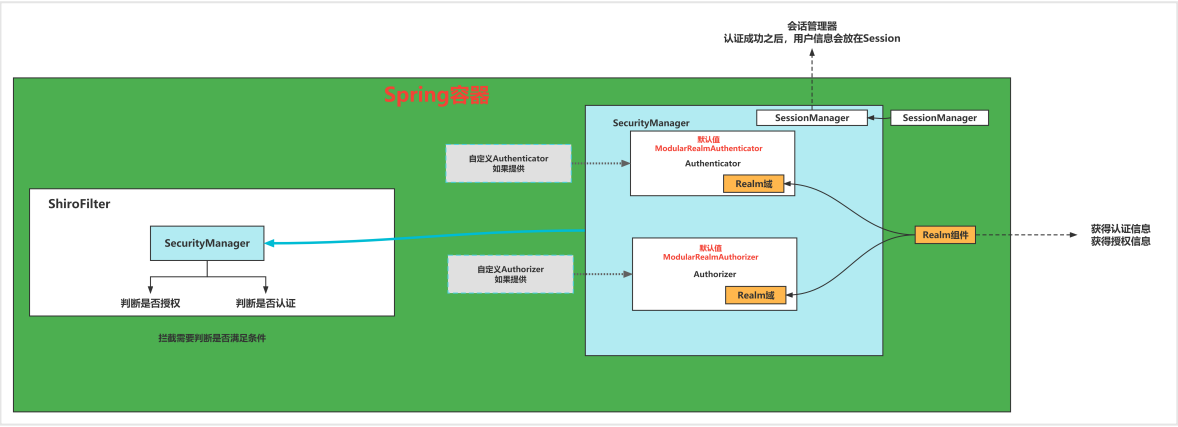
Shiro中的核心组件SecurityManager管理了很多对象，这些对象是Shiro运行过程中必须的对象，如果我们使用Spring容器管理这些组件



4. 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



5. 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

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

5.1. 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

```java
package com.cskaoyan.configuration.realm;

import com.cskaoyan.bean.*;
import com.cskaoyan.bean.vo.DashBoardVO;
import com.cskaoyan.mapper.MarketAdminMapper;
import com.cskaoyan.mapper.MarketPermissionMapper;
import com.cskaoyan.mapper.MarketRoleMapper;
import com.cskaoyan.mapper.MarketUserMapper;
import org.apache.commons.lang3.StringUtils;
import org.apache.shiro.authc.AuthenticationException;
import org.apache.shiro.authc.AuthenticationInfo;
import org.apache.shiro.authc.AuthenticationToken;
import org.apache.shiro.authc.SimpleAuthenticationInfo;
import org.apache.shiro.authz.AuthorizationInfo;
import org.apache.shiro.authz.SimpleAuthorizationInfo;
import org.apache.shiro.realm.AuthorizingRealm;
import org.apache.shiro.subject.PrincipalCollection;
import
org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

import javax.servlet.ServletContext;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Date;
import java.util.List;
```

```
28
29  /**
30   * Realm做的事情就是提供信息 → 认证信息、授权信息
31   * @author stone
32   * @date 2022/06/28 11:14
33   */
34  @Component
35  public class CustomRealm extends AuthorizingRealm {
36
37      @Autowired
38      MarketAdminMapper marketAdminMapper;
39
40      @Autowired
41      MarketUserMapper marketUserMapper;
42
43      @Autowired
44      MarketRoleMapper marketRoleMapper;
45
46      @Autowired
47      MarketPermissionMapper marketPermissionMapper;
48
49      @Autowired
50      HttpServletRequest request;
51
52      @Override
53      protected AuthenticationInfo
    doGetAuthenticationInfo(AuthenticationToken
    authenticationToken) throws AuthenticationException {
54
55
56  // 通常根据AuthenticationToken中传入的用户名信息查询出该用户在数据库
    中的信息
57          String type = ((MarketToken)
    authenticationToken).getType();
58          String username = (String)
    authenticationToken.getPrincipal();
59
60
61
62          ServletContext servletContext =
    request.getServletContext();
63
64          String wxLoginName = (String)
    servletContext.getAttribute("wxLoginName"+username);
65
```

```
66
67  //          如果servletContext域中存在对应的值，且该值等于登录名，无法进行登录
68  //          if(!StringUtils.isEmpty(adminLoginName)){
69  //              if(loginName.equals(username)){
70  //                  return null;
71  //              }
72  //          }
73
74  //          获得认证时间和认证ip
75          Date date = new Date();
76          String remoteHost = request.getRemoteHost();
77          if ("admin".equals(type)) {
78
79              MarketAdminExample example = new MarketAdminExample();
80
 example.createCriteria().andUsernameEqualTo(username);
81              List<MarketAdmin> marketAdmins = marketAdminMapper.selectByExample(example);
82              if (marketAdmins.size() == 1) {
83                  //说明数据库中有一条对应的信息
84                  MarketAdmin marketAdmin = marketAdmins.get(0);
85                  marketAdmin.setLastLoginTime(date);
86                  marketAdmin.setLastLoginIp(remoteHost);
87
88
 marketAdminMapper.updateByPrimaryKeySelective(marketAdmin);
89
 servletContext.setAttribute("adminLoginName"+username,"admin"+username);
90
91                  // 构造认证信息时，可以放入你需要的用户信息，而你放入的用户信息，可以作为Principals
92                  // 放入这个信息，是为了取出这个信息
93                  // 第二个参数credentials，是真实（正确）的密码，会和AuthenticationToken中的password做比较
94                  return new SimpleAuthenticationInfo(marketAdmin,marketAdmin.getPassword(),getName());
95              }
96          } else if ("wx".equals(type)) {
97
98              //查询user表中的信息
```

```
            MarketUserExample marketUserExample = new MarketUserExample();

 marketUserExample.createCriteria().andUsernameEqualTo(username);
            List<MarketUser> marketUsers = marketUserMapper.selectByExample(marketUserExample);
            if (marketUsers.size() == 1) {


                //说明数据库中有一条对应的信息
                MarketUser marketUser = marketUsers.get(0);
                marketUser.setLastLoginTime(date);
                marketUser.setLastLoginIp(remoteHost);


 marketUserMapper.updateByPrimaryKeySelective(marketUser);

 servletContext.setAttribute("wxLoginName"+username,"wx"+username);

                // 构造认证信息时，可以放入你需要的用户信息，而你放入的用户信息，可以作为Principals
                // 放入这个信息，是为了取出这个信息
                // 第二个参数credentials，是真实（正确）的密码，会和AuthenticationToken中的password做比较
                return new SimpleAuthenticationInfo(marketUser,marketUser.getPassword(),getName());
            }
        }

        return null;
    }

    @Override
    protected AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principalCollection) {
        // 要先获得Principal信息
        MarketAdmin principal = (MarketAdmin) principalCollection.getPrimaryPrincipal();
        // 根据用户信息查询出对应的权限列表
        MarketAdminExample marketAdminExample = new MarketAdminExample();
```

```
129         MarketAdminExample.Criteria adminExampleCriteria =
    marketAdminExample.createCriteria();
130 
     adminExampleCriteria.andUsernameEqualTo(principal.getUserna
    me());
131         List<MarketAdmin> marketAdmins =
    marketAdminMapper.selectByExample(marketAdminExample);
132 
133         MarketAdmin marketAdmin = marketAdmins.get(0);
134 
135         Integer[] roleIds = marketAdmin.getRoleIds();
136         ArrayList<String> permissionList = new ArrayList<>
    ();
137         permissionList.add("dashboard");
138         for (Integer roleId : roleIds) {
139             List<String> permissionApiById =
    marketPermissionMapper.selectPermissionApiById(roleId);
140             permissionList.addAll(permissionApiById);
141         }
142 
143 
144         // mybatis来查询
145 //         List<String> permissions = permissionList;
146         SimpleAuthorizationInfo simpleAuthorizationInfo =
    new SimpleAuthorizationInfo();
147 
     simpleAuthorizationInfo.addStringPermissions(permissionList
    );
148         return simpleAuthorizationInfo;
149     }
150 }
151 
```

5.2. 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);
    }
}
```

5.3. 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;
}
```

5.4. 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);

    LinkedHashMap<String, String> filterChainDefinitionMap = new LinkedHashMap<>();
    // /admin/auth/login这个url对应着 anon Filter → 匿名Filter
    filterChainDefinitionMap.put("/admin/auth/login", "anon");
    filterChainDefinitionMap.put("/admin/auth/info", "anon");
    // admin开头的请求，都要通过认证Filter
    filterChainDefinitionMap.put("/admin/**", "authc");
    // 权限的配置，可以将url和对应的权限建立映射关系
    // 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(filterCha
    inDefinitionMap);
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

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

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

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。

```
if (subject.isAuthenticated()) {
    //在已经认证成功的情况下，可以获得用户信息
    // 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
    Object primaryPrincipal = subject.getPrincipals().getPrimaryPrincipal();
    System.out.println(primaryPrincipal);

}
```

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

```
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(securityManager);
    return authorizationAttributeSourceAdvisor;
}
```

@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);
}
```