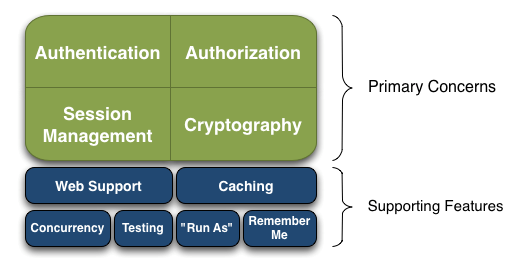
《使用Shiro实现权限验证》

# Shiro入门

Apache Shiro是一个功能强大且易于使用的Java安全框架，提供了认证，授权，加密，和会话管理。



**Authentication**：身份认证 / 登录，验证用户是不是拥有相应的身份；

**Authorization**：授权，即权限验证，验证某个已认证的用户是否拥有某个权限；即判断用户是否能做事情，常见的如：验证某个用户是否拥有某个角色。或者细粒度的验证某个用户对某个资源是否具有某个权限；

**Session Manager**：会话管理，即用户登录后就是一次会话，在没有退出之前，它的所有信息都在会话中；会话可以是普通 JavaSE 环境的，也可以是如 Web 环境的；

**Cryptography**：加密，保护数据的安全性，如密码加密存储到数据库，而不是明文存储；

**Web Support**：Web 支持，可以非常容易的集成到 Web 环境；

**Caching**：缓存，比如用户登录后，其用户信息、拥有的角色 / 权限不必每次去查，这样可以提高效率；

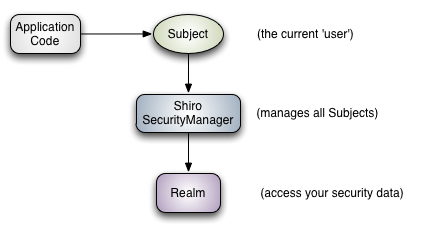
**Concurrency**：shiro 支持多线程应用的并发验证，即如在一个线程中开启另一个线程，能把权限自动传播过去；

**Testing**：提供测试支持；

**Run As**：允许一个用户假装为另一个用户（如果他们允许）的身份进行访问；

**Remember Me**：记住我，这个是非常常见的功能，即一次登录后，下次再来的话不用登录了。

**Shiro 不会去维护用户、维护权限；这些需要我们自己去设计 / 提供；然后通过相应的接口注入给 Shiro 即可。**



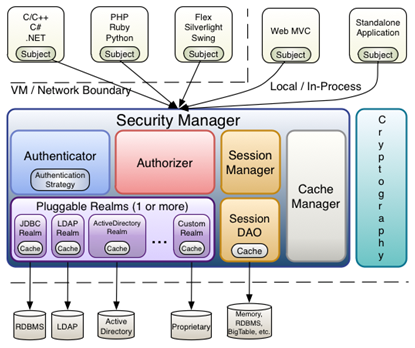
Shiro有三大核心组件：

Subject：即当前用户，在权限管理的应用程序里往往需要知道谁能够操作什么，谁拥有操作该程序的权利，shiro中则需要通过Subject来提供基础的当前用户信息，Subject 不仅仅代表某个用户，与当前应用交互的任何东西都是Subject，如网络爬虫等。所有的Subject都要绑定到SecurityManager上，与Subject的交互实际上是被转换为与SecurityManager的交互。

SecurityManager：即所有Subject的管理者，这是Shiro框架的核心组件，可以把他看做是一个Shiro框架的全局管理组件，用于调度各种Shiro框架的服务。作用类似于SpringMVC中的DispatcherServlet，用于拦截所有请求并进行处理。

Realm：Realm是用户的信息认证器和用户的权限人证器，我们需要自己来实现Realm来自定义的管理我们自己系统内部的权限规则。SecurityManager要验证用户，需要从Realm中获取用户。可以把Realm看做是数据源。

最简单的一个 Shiro 应用：应用代码通过 Subject 来进行认证和授权，而 Subject 又委托给 SecurityManager；我们需要给 Shiro 的 SecurityManager 注入 Realm，从而让 SecurityManager 能得到合法的用户及其权限进行判断。



**Subject**：主体，可以看到主体可以是任何可以与应用交互的 “用户”；

**SecurityManager**：相当于 SpringMVC 中的 DispatcherServlet 或者 Struts2 中的 FilterDispatcher；是 Shiro 的心脏；所有具体的交互都通过 SecurityManager 进行控制；它管理着所有 Subject、且负责进行认证和授权、及会话、缓存的管理。

**Authenticator**：认证器，负责主体认证的，这是一个扩展点，如果用户觉得 Shiro 默认的不好，可以自定义实现；其需要认证策略（Authentication Strategy），即什么情况下算用户认证通过了；

**Authrizer**：授权器，或者访问控制器，用来决定主体是否有权限进行相应的操作；即控制着用户能访问应用中的哪些功能；

**Realm**：可以有 1 个或多个 Realm，可以认为是安全实体数据源，即用于获取安全实体的；可以是 JDBC 实现，也可以是 LDAP 实现，或者内存实现等等；由用户提供；注意：Shiro 不知道你的用户 / 权限存储在哪及以何种格式存储；所以我们一般在应用中都需要实现自己的 Realm；

**SessionManager**：如果写过 Servlet 就应该知道 Session 的概念，Session 呢需要有人去管理它的生命周期，这个组件就是 SessionManager；而 Shiro 并不仅仅可以用在 Web 环境，也可以用在如普通的 JavaSE 环境；所以Shiro 就抽象了一个自己的 Session 来管理主体与应用之间交互的数据。

**SessionDAO**：DAO 大家都用过，数据访问对象，用于会话的 CRUD，比如我们想把 Session 保存到数据库，那么可以实现自己的 SessionDAO，通过如 JDBC 写到数据库；比如想把 Session 放到 Memcached 中，可以实现自己的 Memcached SessionDAO；另外 SessionDAO 中可以使用 Cache 进行缓存，以提高性能；

**CacheManager**：缓存控制器，来管理如用户、角色、权限等的缓存的；因为这些数据基本上很少去改变，放到缓存中后可以提高访问的性能

**Cryptography**：密码模块，Shiro 提高了一些常见的加密组件用于如密码加密 / 解密的。

Shiro中通过principals（身份）和credentials（证明）给shiro,

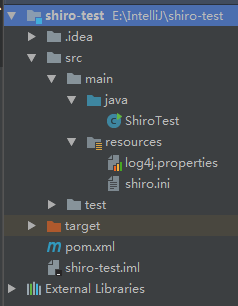
**principals：**身份，即主体的标识属性，可以是任何东西，如用户名、邮箱等，唯一即可。一个主体可以有多个 principals，但只有一个 Primary principals，一般是用户名 / 密码 / 手机号。

**credentials：**证明 / 凭证，即只有主体知道的安全值，如密码 / 数字证书等。

## 搭建测试工程

新建一个普通的java工程，简单了解一下Shiro的API。

完整的工程目录如下：



在pom.xml中加入jar包的配置：

<dependencies>  
 <!--shiro核心类库-->  
 <dependency>  
 <groupId>org.apache.shiro</groupId>  
 <artifactId>shiro-core</artifactId>  
 <version>1.2.3</version>  
 </dependency>  
 <!--日志工具包-->  
 <dependency>  
 <groupId>org.slf4j</groupId>  
 <artifactId>slf4j-log4j12</artifactId>  
 <version>1.6.1</version>  
 </dependency>  
 <dependency>  
 <groupId>org.slf4j</groupId>  
 <artifactId>slf4j-api</artifactId>  
 <version>1.6.1</version>  
 </dependency>  
</dependencies>

使用log4j查看日志，需要在resources目录下创建log4j.properties配置文件：

log4j.rootLogger=INFO, stdout  
  
log4j.appender.stdout=org.apache.log4j.ConsoleAppender  
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout  
log4j.appender.stdout.layout.ConversionPattern=%d %p [%c] - %m%n

在resources目录下编写shiro配置文件，shiro.ini：

#用户名=密码,角色1,角色2...，角色n  
[users]  
root = secret, admin  
guest = guest, guest  
test = 123456, role1, role2  
# -----------------------------------------------------------------------------  
# Roles with assigned permissions  
# roleName = perm1, perm2, ..., permN  
# 角色名=权限1，权限2...权限n  
# -----------------------------------------------------------------------------  
[roles]  
admin = \*  
guest = guest  
role1=perm1,perm2  
role2=perm3

配置文件中包含两个部分用户[users]和角色[roles]

用户配置的格式是：

用户名=密码,角色1,角色2,...角色n

如：

test=123456,role1,role2

用户名是test，密码是123456，拥有两个角色role1和role2

一个用户可以具有多个角色。注意逗号是英文的。

角色配置的格式是：

角色名=权限1,权限2…权限n

如：

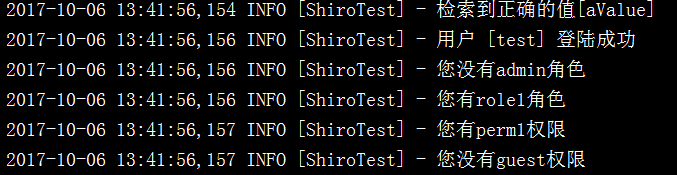
role1=perm1,perm2

角色名是role1，拥有perm1和perm2两个权限。

测试代码

import org.apache.shiro.SecurityUtils;  
import org.apache.shiro.authc.\*;  
import org.apache.shiro.config.IniSecurityManagerFactory;  
import org.apache.shiro.mgt.SecurityManager;  
import org.apache.shiro.session.Session;  
import org.apache.shiro.subject.Subject;  
import org.apache.shiro.util.Factory;  
import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
  
  
public class ShiroTest {  
 private static final transient Logger ***log*** =  
 LoggerFactory.getLogger(ShiroTest.class);  
  
 public static void main(String[] args) {  
 //1. 这里的SecurityManager是org.apache.shiro.mgt.SecurityManager  
 // 而不是java.lang.SecurityManager  
 // 加载配置文件  
 Factory<SecurityManager> factory =  
 new IniSecurityManagerFactory("classpath:shiro.ini");  
 //2.解析配置文件，并且返回一些SecurityManger实例  
 SecurityManager securityManager = factory.getInstance();  
 //3.将SecurityManager绑定给SecurityUtils  
 SecurityUtils.setSecurityManager(securityManager);  
 // 安全操作，Subject是当前登录的用户  
 Subject currentUser = SecurityUtils.getSubject();  
  
 // 测试在应用的当前会话中设置属性  
 Session session = currentUser.getSession();  
 //放进去一个key和一个value  
 session.setAttribute("someKey", "aValue");  
 //根据key拿到value  
 String value = (String) session.getAttribute("someKey");  
 if ("aValue".equals(value)) {//比较拿到的值和原来的值是否一致  
 ***log***.info("检索到正确的值[" + value + "]");  
 }  
 //尝试进行登录用户，如果登录失败了，我们进行一些处理  
 if (!currentUser.isAuthenticated()) {//如果用户没有登录过  
 //new UsernamePasswordToken(用户名,密码)  
 UsernamePasswordToken token =  
 new UsernamePasswordToken("test", "123456");  
 token.setRememberMe(true);//是否记住用户  
 try {  
 currentUser.login(token);  
 //当我们获登录用户之后  
 ***log***.info("用户 [" + currentUser.getPrincipal() + "] 登陆成功");  
 // 查看用户是否有指定的角色  
 if (currentUser.hasRole("admin")) {  
 ***log***.info("您有admin角色");  
 } else {  
 ***log***.info("您没有admin角色");  
 }  
 if (currentUser.hasRole("role1")) {  
 ***log***.info("您有role1角色");  
 } else {  
 ***log***.info("您没有role1角色");  
 }  
  
 // 查看用户是否有某个权限  
 if (currentUser.isPermitted("perm1")) {  
 ***log***.info("您有perm1权限");  
 } else {  
 ***log***.info("您没有perm1权限");  
 }  
 if (currentUser.isPermitted("guest")) {  
 ***log***.info("您有guest权限");  
 } else {  
 ***log***.info("您没有guest权限");  
 }  
 //登出  
 currentUser.logout();  
 } catch (UnknownAccountException uae) {  
 ***log***.info(token.getPrincipal() + "账户不存在");  
 } catch (IncorrectCredentialsException ice) {  
 ***log***.info(token.getPrincipal() + "密码不正确");  
 } catch (LockedAccountException lae) {  
 ***log***.info(token.getPrincipal() + "用户被锁定了 ");  
 } catch (AuthenticationException ae) {  
 //无法判断是什么错了  
 ***log***.info(ae.getMessage());  
 }  
 }  
 }  
}

运行结果：



1. 首先通过 new IniSecurityManagerFactory 并指定一个 ini 配置文件来创建一个 SecurityManager 工厂；
2. 接着获取 SecurityManager 并绑定到 SecurityUtils，这是一个全局设置，设置一次即可；
3. 通过 SecurityUtils 得到 Subject，然后获取身份验证的 Token，如用户名 / 密码；
4. 调用 subject.login 方法进行登录，其会自动委托给SecurityManager.login 方法进行登录；

如果身份验证失败请捕获 AuthenticationException 或其子类，常见的如：

DisabledAccountException（禁用的帐号）

LockedAccountException（锁定的帐号）

UnknownAccountException（错误的帐号）

ExcessiveAttemptsException（登录失败次数过多）

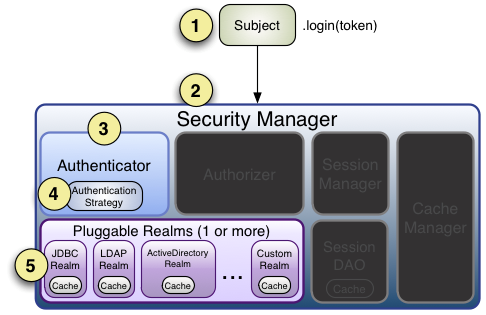
IncorrectCredentialsException （错误的凭证）

ExpiredCredentialsException（过期的凭证）等

具体请查看其继承关系；对于页面的错误消息展示，最好使用如 “用户名 / 密码错误” 而不是 “用户名错误”/“密码错误”，防止一些恶意用户非法扫描帐号库；

1. 最后可以调用 subject.logout 退出，其会自动委托给SecurityManager.logout 方法退出。

## 身份验证流程



流程如下：

1. 首先调用 Subject.login(token) 进行登录，其会自动委托给 Security Manager，调用之前必须通过 SecurityUtils.setSecurityManager() 设置；
2. SecurityManager 负责真正的身份验证逻辑；它会委托给 Authenticator 进行身份验证；
3. Authenticator 才是真正的身份验证者，Shiro API 中核心的身份认证入口点，此处可以自定义插入自己的实现；
4. Authenticator 可能会委托给相应的 AuthenticationStrategy 进行多 Realm 身份验证，默认 ModularRealmAuthenticator 会调用 AuthenticationStrategy 进行多 Realm 身份验证；
5. Authenticator 会把相应的 token 传入 Realm，从 Realm 获取身份验证信息，如果没有返回 / 抛出异常表示身份验证失败了。此处可以配置多个 Realm，将按照相应的顺序及策略进行访问。

**SecurityManager 接口继承了 Authenticator，另外还有一个 ModularRealmAuthenticator 实现，其委托给多个 Realm 进行验证，验证规则通过 AuthenticationStrategy 接口指定，默认提供的实现：**

**FirstSuccessfulStrategy：**

**只要有一个 Realm 验证成功即可，只返回第一个 Realm 身份验证成功的认证信息，其他的忽略；**

**AtLeastOneSuccessfulStrategy：**

**只要有一个 Realm 验证成功即可，和 FirstSuccessfulStrategy 不同，返回所有 Realm 身份验证成功的认证信息；**

**AllSuccessfulStrategy：**

**所有 Realm 验证成功才算成功，且返回所有 Realm 身份验证成功的认证信息，如果有一个失败就失败了。**

**ModularRealmAuthenticator 默认使用 AtLeastOneSuccessfulStrategy 策略。**

## 授权

授权，也叫访问控制，即在应用中控制谁能访问哪些资源（如访问页面/编辑数据/页面操作等）。在授权中需了解的几个关键对象：主体（Subject）、资源（Resource）、权限（Permission）、角色（Role）。

**主体：**主体，即访问应用的用户，在 Shiro 中使用 Subject 代表该用户。用户只有授权后才允许访问相应的资源。

**资源**：在应用中用户可以访问的任何东西，比如访问 JSP 页面、查看/编辑某些数据、访问某个业务方法、打印文本等等都是资源。用户只要授权后才能访问。

**权限**：安全策略中的原子授权单位，通过权限我们可以表示在应用中用户有没有操作某个资源的权力。即权限表示在应用中用户能不能访问某个资源，如： 访问用户列表页面、查看/新增/修改/删除用户数据（即很多时候都是 CRUD（增查改删）式权限控制）、打印文档等等。。。

如上可以看出，权限代表了用户有没有操作某个资源的权利，即反映在某个资源上的操作允不允许，不反映谁去执行这个操作。所以后续还需要把权限赋予给用户，即定义哪个用户允许在某个资源上做什么操作（权限），Shiro 不会去做这件事情，而是由实现人员提供。

Shiro 支持粗粒度权限（如用户模块的所有权限）和细粒度权限（操作某个用户的权限，即实例级别的），后续部分介绍。

**角色**：角色代表了操作集合，可以理解为权限的集合，一般情况下我们会赋予用户角色而不是权限，即这样用户可以拥有一组权限，赋予权限时比较方便。典型的如：项目经理、技术总监、CTO、开发工程师等都是角色，不同的角色拥有一组不同的权限。

**隐式角色**：即直接通过角色来验证用户有没有操作权限，如在应用中 CTO、技术总监、开发工程师可以使用打印机，假设某天不允许开发工程师使用打印机，此时需要从应用中删除相应代码；再如在应用中 CTO、技术总监可以查看用户、查看权限；突然有一天不允许技术总监查看用户、查看权限了，需要在相关代码中把技术总监角色从判断逻辑中删除掉；即粒度是以角色为单位进行访问控制的，粒度较粗；如果进行修改可能造成多处代码修改。

**显示角色**：在程序中通过权限控制谁能访问某个资源，角色聚合一组权限集合；这样假设哪个角色不能访问某个资源，只需要从角色代表的权限集合中移除即可；无须修改多处代码；即粒度是以资源/实例为单位的；粒度较细。

# Ini配置文件

ini配置文件语法：

1. 对象名 = 全限定类名 相对于调用 public 无参构造器创建对象
2. 对象名. 属性名 = 值 相当于调用 setter 方法设置常量值
3. 对象名. 属性名 =$ 对象引用 相当于调用 setter 方法设置对象引用

配置文件如下：

myRealm1=com.shiro.my.realm.MyRealm1  
myRealm2=com.shiro.my.realm.MyRealm2  
  
securityManager.realms=$myRealm1,$myRealm2

纯java写法如下：

**DefaultSecurityManager securityManager = new** DefaultSecurityManager();  
**//设置authenticator身份验证  
ModularRealmAuthenticator authenticator = new** ModularRealmAuthenticator();  
**authenticator**.setAuthenticationStrategy(**new** AtLeastOneSuccessfulStrategy());  
**securityManager**.setAuthenticator(**authenticator**);  
**//设置authorizer 授权  
ModularRealmAuthorizer authorizer = new** ModularRealmAuthorizer();  
**authorizer**.setPermissionResolver(**new** WildcardPermissionResolver());  
**securityManager**.setAuthorizer(**authorizer**);  
**securityManager**.setRealms(**Arrays**.asList(**new** MyRealm1()));  
**//将SecurityManager设置到SecurityUtils 方便全局使用**SecurityUtils.setSecurityManager(**securityManager**);  
Subject **subject =** SecurityUtils.getSubject();  
**UsernamePasswordToken token = new** UsernamePasswordToken(**"test"**, **"123"**);  
**subject**.**login**(**token**);  
**System**.***out***.println(**subject**.**isAuthenticated**());

默认情况先创建一个名字为 securityManager，类型为 org.apache.shiro.mgt.DefaultSecurityManager 的默认的 SecurityManager，如果想自定义，只需要在 ini 配置文件中指定 “securityManager=SecurityManager 实现类” 即可，名字必须为 securityManager

**[main] 部分**

提供了对根对象 securityManager 及其依赖对象的配置。

**创建对象**

securityManager=org.apache.shiro.mgt.DefaultSecurityManager

其构造器必须是 public 空参构造器，通过反射创建相应的实例。

**常量值 setter 注入**

dataSource.driverClassName=com.mysql.jdbc.Driver  
jdbcRealm.permissionsLookupEnabled=true

会自动调用 jdbcRealm.setPermissionsLookupEnabled(true)，对于这种常量值会自动类型转换。

**对象引用 setter 注入**

authenticator=org.apache.shiro.authc.pam.ModularRealmAuthenticator  
authenticationStrategy=org.apache.shiro.authc.pam.AtLeastOneSuccessfulStrategy  
authenticator.authenticationStrategy=$authenticationStrategy  
securityManager.authenticator=$authenticator

会自动通过 securityManager.setAuthenticator(authenticator) 注入引用依赖。

**嵌套属性 setter 注入**

securityManager.authenticator.authenticationStrategy=$authenticationStrategy

也支持这种嵌套方式的 setter 注入。

**byte 数组 setter 注入**

#base64 byte[]  
authenticator.bytes=aGVsbG8=  
#hex byte[]  
authenticator.bytes=0x68656c6c6f

默认需要使用 Base64 进行编码，也可以使用 0x 十六进制。

**Array/Set/List setter 注入**

authenticator.array=1,2,3  
authenticator.set=$jdbcRealm,$jdbcRealm

多个之间通过 “，” 分割。

**Map setter 注入**

authenticator.map=$jdbcRealm:$jdbcRealm,1:1,key:abc

即格式是：map=key：value，key：value，可以注入常量及引用值，常量的话都看作字符串（即使有泛型也不会自动造型）。

**实例化 / 注入顺序**

realm=Realm1  
realm=Realm12  
authenticator.bytes=aGVsbG8=  
authenticator.bytes=0x68656c6c6f

后边的覆盖前边的注入。

测试用例请参考配置文件 shiro-config-main.ini。

**[users] 部分**

配置用户名 / 密码及其角色，格式：“用户名 = 密码，角色 1，角色 2”，角色部分可省略。如：

[users]  
zhang=123,role1,role2  
wang=123

密码一般生成其摘要 / 加密存储，后续章节介绍。

**[roles] 部分**

配置角色及权限之间的关系，格式：“角色 = 权限 1，权限 2”；如：

[roles]  
role1=user:create,user:update  
role2=\*

如果只有角色没有对应的权限，可以不配 roles，具体规则请参考授权章节。

**[urls] 部分**

配置 url 及相应的拦截器之间的关系，格式：“url = 拦截器 [参数]，拦截器 [参数]，如：

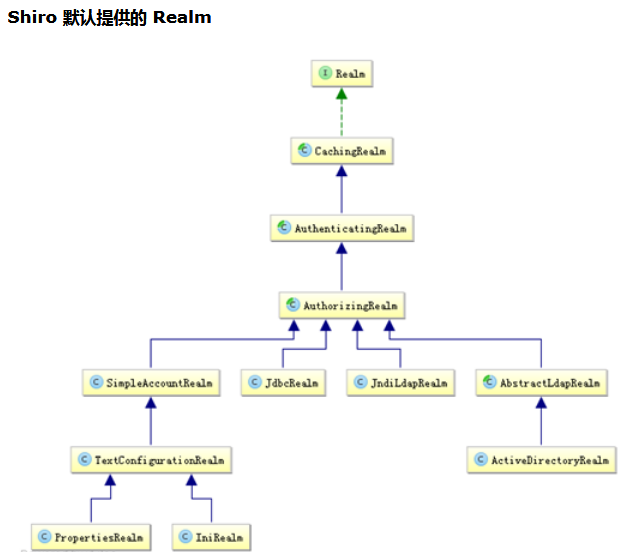
[urls]  
/admin/\*\* = authc, roles[admin], perms["permission1"]

# Realm

Realm：域，Shiro 从从 Realm 获取安全数据（如用户、角色、权限），就是说 SecurityManager 要验证用户身份，那么它需要从 Realm 获取相应的用户进行比较以确定用户身份是否合法；也需要从 Realm 得到用户相应的角色 / 权限进行验证用户是否能进行操作；可以把 Realm 看成 DataSource，即安全数据源。如之前的 ini 配置方式将使用 org.apache.shiro.realm.text.IniRealm。

org.apache.shiro.realm.Realm 接口如下：

**package** org.apache.shiro.realm;  
  
**import** org.apache.shiro.authc.AuthenticationException;  
**import** org.apache.shiro.authc.AuthenticationInfo;  
**import** org.apache.shiro.authc.AuthenticationToken;  
  
**public interface** Realm **{** String getName();  
  
 **boolean** supports(AuthenticationToken var1);  
  
 AuthenticationInfo getAuthenticationInfo(AuthenticationToken var1) **throws** AuthenticationException;  
**}**



**org.apache.shiro.realm.text.IniRealm**：[users] 部分指定用户名 / 密码及其角色；[roles] 部分指定角色即权限信息；

**org.apache.shiro.realm.text.PropertiesRealm**： user.username=password,role1,role2 指定用户名 / 密码及其角色；role.role1=permission1,permission2 指定角色及权限信息；

**org.apache.shiro.realm.jdbc.JdbcRealm**：通过 sql 查询相应的信息，如 “select password from users where username = ?” 获取用户密码，“select password, password\_salt from users where username = ?” 获取用户密码及盐；“select role\_name from user\_roles where username = ?” 获取用户角色；“select permission from roles\_permissions where role\_name = ?” 获取角色对应的权限信息；也可以调用相应的 api 进行自定义 sql；

## JdbcRealm

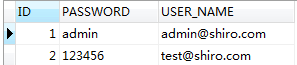
### 数据库设计

在实际开发中，用户名密码、角色、权限需要存在数据库中动态管理。一个简单的Shiro+MySQL的项目需要三张表：

用户表SHIRO\_USER：



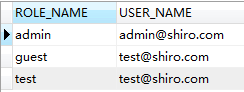
测试数据：



用户角色表SHIRO\_USER\_ROLE：



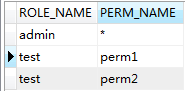
测试数据：



角色权限表SHIRO\_ROLE\_PERMISSION：



测试数据：



### 工程搭建

工程的搭建与第1.1节中介绍的一样。需要在1.2节的基础上加上数据源的jar包和mysql的驱动包，数据源我们选用spring-jdbc：

<dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
 <version>5.1.32</version>  
</dependency>  
<dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-jdbc</artifactId>  
 <version>4.3.11.RELEASE</version>  
</dependency>

### 配置文件

在使用jdbc数据源的时候，不需要指定user和roles，而是在配置文件中指定数据库连接信息和要执行的sql语句。

在resources文件夹下创建配置文件shiro-mysql.ini：

[main]  
dataSource=org.springframework.jdbc.datasource.DriverManagerDataSource  
dataSource.driverClassName=com.mysql.jdbc.Driver  
dataSource.url=jdbc:mysql://127.0.0.1:3306/数据库名  
dataSource.username=用户名  
#如果数据库没有密码，就不要写这行  
dataSource.password=你的密码  
jdbcRealm=org.apache.shiro.realm.jdbc.JdbcRealm  
#是否检查权限  
jdbcRealm.permissionsLookupEnabled = true  
jdbcRealm.dataSource=$dataSource  
#重写sql语句  
#根据用户名查询出密码  
jdbcRealm.authenticationQuery = select PASSWORD from SHIRO\_USER where USER\_NAME = ?  
#根据用户名查询出角色  
jdbcRealm.userRolesQuery = select ROLE\_NAME from SHIRO\_USER\_ROLE where USER\_NAME = ?  
#根据角色名查询出权限  
jdbcRealm.permissionsQuery = select PERM\_NAME from SHIRO\_ROLE\_PERMISSION WHERE ROLE\_NAME = ?  
securityManager.realms=$jdbcRealm

变量名 = 全限定类名会自动创建一个类实例

变量名. 属性 = 值 自动调用相应的 setter 方法进行赋值

$ 变量名 引用之前的一个对象实例

测试代码请参照 com.github.zhangkaitao.shiro.chapter2.LoginLogoutTest 的 testJDBCRealm 方法，和之前的没什么区别。

注意sql语句，每次只查询一个shiro要求查询的字段，如果写select \*就会报错了。

ini配置文件要求必须是key=value的形式，如果有些人没有设置数据库的密码，就不要写对应的配置。只写”dataSource.password=”等号右面没有值会报错。

## 自定义Realm

自定义Realm需要实现Realm接口

**package com.shiro.my.realm**;  
  
**import org.apache.shiro.authc.\***;  
**import org.apache.shiro.realm.**Realm;  
  
**public class MyRealm1 implements** Realm **{  
 @Override  
 public String** getName() **{  
 return "myrealm1"**;  
 **}  
  
 @Override  
 public boolean** supports(AuthenticationToken **token**) **{  
 //仅支持UsernamePasswordToken类型的Token  
 return token instanceof UsernamePasswordToken**;  
 **}  
  
 @Override  
 public** AuthenticationInfo getAuthenticationInfo(AuthenticationToken **token**) **throws AuthenticationException {  
 String username =** (**String**) **token**.**getPrincipal**(); **//得到用户名  
 String password = new** String((**char**[]) **token**.**getCredentials**()); **//得到密码  
 if** (**!"test"**.equals(**username**)) **{  
 throw new** UnknownAccountException(); **//如果用户名错误  
 }  
 if** (**!"123"**.equals(**password**)) **{  
 throw new** IncorrectCredentialsException(); **//如果密码错误  
 }  
 //如果身份认证验证成功，返回一个AuthenticationInfo实现；  
 return new** SimpleAuthenticationInfo(**username**, **password**, getName());  
 **}  
}**

可以定义多个Realm。

MyRealm2略。

配置文件：

myRealm1=com.shiro.my.realm.MyRealm1  
myRealm2=com.shiro.my.realm.MyRealm2  
  
securityManager.realms=$myRealm1,$myRealm2

securityManager 会按照 realms 指定的顺序进行身份认证。此处我们使用显示指定顺序的方式指定了 Realm 的顺序，如果删除 “securityManager.realms=$myRealm1,$myRealm2”，那么securityManager 会按照 realm 声明的顺序进行使用（即无需设置 realms 属性，其会自动发现），当我们显示指定 realm 后，其他没有指定 realm 将被忽略，如 “securityManager.realms=$myRealm1”，那么 myRealm2 不会被自动设置进去。

以后一般继承 AuthorizingRealm（授权）即可；其继承了 AuthenticatingRealm（即身份验证），而且也间接继承了 CachingRealm（带有缓存实现）。其中主要默认实现如下：

Authenticator 及 AuthenticationStrategy

**Authenticator 的职责是验证用户帐号，是 Shiro API 中身份验证核心的入口点：**

**package** org.apache.shiro.authc;  
  
**public interface** Authenticator **{** AuthenticationInfo authenticate(AuthenticationToken var1) **throws** AuthenticationException;  
**}**

**如果验证成功，将返回 AuthenticationInfo 验证信息；此信息中包含了身份及凭证；如果验证失败将抛出相应的 AuthenticationException 实现。**

带权限验证的Realm

**public class MyRealm3 extends** AuthorizingRealm **{  
  
 @Override  
 protected** AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection **principalCollection**) **{  
 //获取当前登录的身份主体  
 String username =** (**String**) **principalCollection**.**getPrimaryPrincipal**();  
 **SimpleAuthorizationInfo authorizationInfo = new** SimpleAuthorizationInfo();  
 **//假设用户有admin角色和select权限  
 authorizationInfo**.addRole(**"admin"**);  
 **authorizationInfo**.addStringPermission(**"select"**);  
 **return authorizationInfo**;  
 **}  
  
 @Override  
 protected** AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken **authenticationToken**) **throws AuthenticationException {  
 String username =** (**String**) **authenticationToken**.**getPrincipal**();  
 **//根据主体信息去数据库查询用户信息,如密码，这里假设密码是123456  
 // String password = "please select the password from database";  
 String password = "123456"**;  
 **//将查询出的信息封装成info对象并返回  
 SimpleAuthenticationInfo authenticationInfo = new** SimpleAuthenticationInfo(  
 **username**, **//用户名  
 password**, **//密码  
 null**,**//salt=username+salt** getName() **//realm name** );  
 **return authenticationInfo**;  
 **}  
}**

# 集成web

## 使用Servlet

依赖jar包

<dependencies>  
 <!--shiro核心类库-->  
 <dependency>  
 <groupId>org.apache.shiro</groupId>  
 <artifactId>shiro-core</artifactId>  
 <version>1.2.3</version>  
 </dependency>  
 <!--shiro web类库-->  
 <dependency>  
 <groupId>org.apache.shiro</groupId>  
 <artifactId>shiro-web</artifactId>  
 <version>1.2.3</version>  
 </dependency>  
 <!--日志工具包-->  
 <dependency>  
 <groupId>org.slf4j</groupId>  
 <artifactId>slf4j-log4j12</artifactId>  
 <version>1.6.1</version>  
 </dependency>  
 <dependency>  
 <groupId>org.slf4j</groupId>  
 <artifactId>slf4j-api</artifactId>  
 <version>1.6.1</version>  
 </dependency>  
 <dependency>  
 <groupId>javax.servlet</groupId>  
 <artifactId>javax.servlet-api</artifactId>  
 <version>3.0.1</version>  
 <scope>provided</scope>  
 </dependency>  
</dependencies>

使用ini当做数据源：

[users]  
root = secret, admin  
guest = guest, guest  
test = 123456, guest,test  
  
[roles]  
admin = \*  
guest = user:list  
test = menu:list,menu:add  
  
[urls]  
/login.html=anon  
/index.html = authc  
/role.html=authc,roles[admin]  
/menu/\*\* = authc, roles[admin],perms[menu:\*]

web.xml中配置shiro:

<context-param>  
 <param-name>shiroEnvironmentClass</param-name>  
 <param-value>org.apache.shiro.web.env.IniWebEnvironment</param-value>  
</context-param>  
<context-param>  
 <param-name>shiroConfigLocations</param-name>  
 <param-value>classpath:shiro-web.ini</param-value>  
</context-param>  
<listener>  
 <listener-class>org.apache.shiro.web.env.EnvironmentLoaderListener</listener-class>  
</listener>  
<filter>  
 <filter-name>ShiroFilter</filter-name>  
 <filter-class>org.apache.shiro.web.servlet.ShiroFilter</filter-class>  
</filter>  
  
<filter-mapping>  
 <filter-name>ShiroFilter</filter-name>  
 <url-pattern>\*.html</url-pattern>  
</filter-mapping>

servlet配置：

**package com.shiro.test.servlet**;  
  
**import org.apache.shiro.**SecurityUtils;  
**import org.apache.shiro.authc.AuthenticationException**;  
**import org.apache.shiro.authc.UsernamePasswordToken**;  
**import org.apache.shiro.subject.**Subject;  
  
**import javax.servlet.ServletException**;  
**import javax.servlet.annotation.WebServlet**;  
**import javax.servlet.http.**HttpServlet;  
**import javax.servlet.http.**HttpServletRequest;  
**import javax.servlet.http.**HttpServletResponse;  
**import java.io.IOException**;  
  
**@WebServlet**(**name = "loginServlet"**, **urlPatterns = "/login.html"**)  
**public class LoginServlet extends** HttpServlet **{  
 @Override  
 protected void** doGet(HttpServletRequest **req**, HttpServletResponse **resp**)  
 **throws ServletException**, **IOException {  
 this**.doPost(**req**, **resp**);  
 **}  
  
 @Override  
 protected void** doPost(HttpServletRequest **req**, HttpServletResponse **resp**)  
 **throws ServletException**, **IOException {  
 String username = req**.**getParameter**(**"username"**);  
 **String password = req**.**getParameter**(**"password"**);  
 Subject **subject =** SecurityUtils.getSubject();  
 **UsernamePasswordToken token = new** UsernamePasswordToken(**username**, **password**);  
 **try {  
 subject**.**login**(**token**);  
 **} catch** (**AuthenticationException e**) **{  
 req**.**setAttribute**(**"error"**, **"登录失败"**);  
 **req**.**getRequestDispatcher**(**"/login.jsp"**).**forward**(**req**, **resp**);  
 **return**;  
 **}  
 resp**.**sendRedirect**(**"/index.html"**);  
 **}  
}**

**package com.shiro.test.servlet**;  
  
**import javax.servlet.ServletException**;  
**import javax.servlet.annotation.WebServlet**;  
**import javax.servlet.http.**HttpServlet;  
**import javax.servlet.http.**HttpServletRequest;  
**import javax.servlet.http.**HttpServletResponse;  
**import java.io.IOException**;  
  
**@WebServlet**(**name = "indexServlet"**, **urlPatterns = "/index.html"**)  
**public class IndexServlet extends** HttpServlet **{  
 @Override  
 protected void** doGet(HttpServletRequest **req**, HttpServletResponse **resp**)  
 **throws ServletException**, **IOException {  
 this**.doPost(**req**, **resp**);  
 **}  
 protected void** doPost(HttpServletRequest **req**, HttpServletResponse **resp**)  
 **throws ServletException**, **IOException {  
 req**.**getRequestDispatcher**(**"/index.jsp"**).**forward**(**req**, **resp**);  
 **}  
}**

页面login.jsp：

<form action="/login.html" method="post">  
 用户名：<input type="text" name="username"/><br/>  
 密码：<input type="password" name="password"/><br/>  
 <input type="submit" value="登录"/>${error}  
</form>

首页使用easyui:

<body class="easyui-layout">  
<div data-options="region:'north'" style="height: 50px">north</div>  
<div data-options="region:'west',title:'West'" style="width: 100px">  
 <ul class="easyui-tree">  
 <shiro:hasPermission name="menu:list">  
 <li data-options="url:'/menu/list.html'">  
 <span>菜单管理</span>  
 </li>  
 </shiro:hasPermission>  
 <shiro:hasPermission name="role">  
 <li data-options="url:'/role/list.html'">  
 <span>角色管理</span>  
 </li>  
 </shiro:hasPermission>  
 <shiro:hasRole name="test">  
 <li data-options="url:'/user/list.html'">  
 <span>用户管理</span>  
 </li>  
 </shiro:hasRole>  
 </ul>  
</div>  
<div data-options="region:'center',title:'首页'">  
</div>  
</body>

## 集成SpringMVC

引入jar包：

<!--spring相关包-->  
<dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-webmvc</artifactId>  
 <version>4.3.11.RELEASE</version>  
</dependency>  
<dependency>  
 <groupId>org.apache.shiro</groupId>  
 <artifactId>shiro-spring</artifactId>  
 <version>1.4.0</version>  
</dependency>

Controller

**package com.shiro.test.mvc**;  
  
**import org.apache.shiro.**SecurityUtils;  
**import org.apache.shiro.authc.AuthenticationException**;  
**import org.apache.shiro.authc.UsernamePasswordToken**;  
**import org.apache.shiro.subject.**Subject;  
**import org.springframework.stereotype.Controller**;  
**import org.springframework.web.bind.annotation.RequestMapping**;  
  
**import javax.servlet.http.**HttpServletRequest;  
  
**@Controller  
public class LoginController {  
 @RequestMapping**(**"/gologin.html"**)  
 **public String** gologin() **{  
 return "login"**;  
 **}  
  
 @RequestMapping**(**"/login.html"**)  
 **public String** login(**String username**, **String password**, HttpServletRequest **req**) **{** Subject **subject =** SecurityUtils.getSubject();  
 **UsernamePasswordToken token = new** UsernamePasswordToken(**username**, **password**);  
 **try {  
 subject**.**login**(**token**);  
 **} catch** (**AuthenticationException e**) **{  
 e**.printStackTrace();  
 **req**.**setAttribute**(**"error"**, **"登录失败"**);  
 **return "login"**;  
 **}  
 return "redirect:/index.html"**;  
 **}  
}**

**package com.shiro.test.mvc**;  
  
**import org.springframework.stereotype.Controller**;  
**import org.springframework.web.bind.annotation.RequestMapping**;  
  
**@Controller  
public class IndexController {  
 //首页  
 @RequestMapping**(**"index.html"**)  
 **public String** index() **{  
 return "index"**;  
 **}  
  
 //错误页  
 @RequestMapping**(**"error.html"**)  
 **public String** error() **{  
 return "error"**;  
 **}  
}**

**package com.shiro.test.mvc**;  
  
**import org.apache.shiro.authz.annotation.RequiresPermissions**;  
**import org.springframework.stereotype.Controller**;  
**import org.springframework.web.bind.annotation.RequestMapping**;  
  
**@Controller  
@RequestMapping**(**"/menu"**)  
**public class MenuController {  
 @RequestMapping**(**"list.html"**)  
 **public String** list() **{  
 return "menu\_list"**;  
 **}  
 //必须有menu:edit权限  
 @RequiresPermissions**(**"menu:edit"**)  
 **@RequestMapping**(**"go\_edit.html"**)  
 **public String** goEdit() **{  
 return "menu\_edit"**;  
 **}  
}**

验证失败会抛出UnauthorizedException异常，处理错误信息：

**package com.shiro.test.mvc**;  
  
**import org.apache.shiro.authz.UnauthorizedException**;  
**import org.springframework.http.**HttpStatus;  
**import org.springframework.web.bind.annotation.ControllerAdvice**;  
**import org.springframework.web.bind.annotation.ExceptionHandler**;  
**import org.springframework.web.bind.annotation.ResponseStatus**;  
**import org.springframework.web.context.request.**NativeWebRequest;  
**import org.springframework.web.servlet.ModelAndView**;  
  
**@ControllerAdvice  
public class AuthExceptionHandler {  
  
 @ExceptionHandler**(**{UnauthorizedException**.**class}**)  
 **@ResponseStatus**(HttpStatus.***UNAUTHORIZED***)  
 **public ModelAndView** processUnauthenticatedException(NativeWebRequest **request**, **UnauthorizedException e**) **{  
 ModelAndView mv = new** ModelAndView();  
 **mv**.addObject(**"exception"**, **e**);  
 **mv**.setViewName(**"error"**);  
 **return mv**;  
 **}  
  
}**

spring-config.xml

<!--读取ini配置文件作为数据源-->  
<bean id="iniRealm" class="org.apache.shiro.realm.text.IniRealm">  
 <constructor-arg name="resourcePath" value="classpath:shiro-web.ini"/>  
</bean>  
<bean id="securityManager" class="org.apache.shiro.web.mgt.DefaultWebSecurityManager">  
 <property name="realm" ref="iniRealm"/>  
</bean>  
<!--启用注解权限验证-->  
<bean class="org.apache.shiro.spring.security.interceptor.AuthorizationAttributeSourceAdvisor">  
 <property name="securityManager" ref="securityManager"/>  
</bean>  
<bean id="shiroFilter" class="org.apache.shiro.spring.web.ShiroFilterFactoryBean">  
 <property name="securityManager" ref="securityManager"/>  
 <!--去登录的地址-->  
 <property name="loginUrl" value="/gologin.html"/>  
 <!--登录成功的跳转地址-->  
 <property name="successUrl" value="/index.html"/>  
 <!--验证失败的跳转地址-->  
 <property name="unauthorizedUrl" value="/error.html"/>  
 <!--定义过滤的规则-->  
 <property name="filterChainDefinitions">  
 <value>  
 /login.html=anon  
 /index.html = authc  
 /menu/\*\* = authc  
 </value>  
 </property>  
</bean>

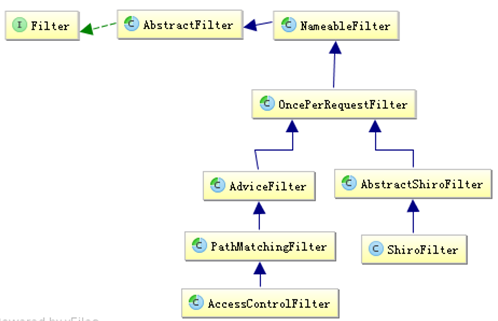
springMVC-servlet.xml

<mvc:annotation-driven/>  
<context:component-scan base-package="com.shiro.test.mvc">  
 <!-- 制定扫包规则 ,只扫描使用@Controller注解的JAVA类 -->  
 <context:include-filter type="annotation" expression="org.springframework.stereotype.Controller"/>  
</context:component-scan>  
  
<!-- 配置视图解析器-->  
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">  
 <property name="prefix" value="/WEB-INF/pages/"></property><!--前缀-->  
 <property name="suffix" value=".jsp"></property><!--后缀-->  
</bean>  
<!--注解权限验证一定要写在mvc自容器中-->  
<bean id="lifecycleBeanPostProcessor" class="org.apache.shiro.spring.LifecycleBeanPostProcessor"></bean>  
  
<bean class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"  
 depends-on="lifecycleBeanPostProcessor"/>

web.xml

<context-param>  
 <param-name>contextConfigLocation</param-name>  
 <param-value>classpath:spring-config.xml</param-value>  
</context-param>  
  
<listener>  
 <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>  
</listener>  
  
<!--配置DispatcherServlet-->  
<servlet>  
 <servlet-name>springMVC</servlet-name>  
 <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  
</servlet>  
<servlet-mapping>  
 <servlet-name>springMVC</servlet-name>  
 <url-pattern>\*.html</url-pattern>  
</servlet-mapping>  
  
<!-- 配置shiro的核心拦截器 -->  
<filter>  
 <filter-name>shiroFilter</filter-name>  
 <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>  
</filter>  
<filter-mapping>  
 <filter-name>shiroFilter</filter-name>  
 <url-pattern>\*.html</url-pattern>  
</filter-mapping>

# 拦截器



* 1. **NameableFilter**  
     NameableFilter 给 Filter 起个名字，如果没有设置默认就是 FilterName；还记得之前的如 authc 吗？当我们组装拦截器链时会根据这个名字找到相应的拦截器实例；
  2. **OncePerRequestFilter**  
     OncePerRequestFilter 用于防止多次执行 Filter 的；也就是说一次请求只会走一次拦截器链；另外提供 enabled 属性，表示是否开启该拦截器实例，默认 enabled=true 表示开启，如果不想让某个拦截器工作，可以设置为 false 即可。
  3. **ShiroFilter**  
     ShiroFilter 是整个 Shiro 的入口点，用于拦截需要安全控制的请求进行处理，这个之前已经用过了。
  4. **AdviceFilter**  
     AdviceFilter 提供了 AOP 风格的支持，类似于 SpringMVC 中的 Interceptor：

**protected boolean** preHandle(ServletRequest request, ServletResponse response) **throws** Exception **{  
 return true**;  
**}  
  
protected void** postHandle(ServletRequest request, ServletResponse response) **throws** Exception **{  
}  
  
public void** afterCompletion(ServletRequest request, ServletResponse response, Exception exception) **throws** Exception **{  
}**

preHandler：类似于 AOP 中的前置增强；在拦截器链执行之前执行；如果返回 true 则继续拦截器链；否则中断后续的拦截器链的执行直接返回；进行预处理（如基于表单的身份验证、授权）

postHandle：类似于 AOP 中的后置返回增强；在拦截器链执行完成后执行；进行后处理（如记录执行时间之类的）；

afterCompletion：类似于 AOP 中的后置最终增强；即不管有没有异常都会执行；可以进行清理资源（如接触 Subject 与线程的绑定之类的）；

* 1. **PathMatchingFilter**

**PathMatchingFilter 提供了基于 Ant 风格的请求路径匹配功能及拦截器参数解析的功能，如“roles[admin,user]”自动根据“，”分割解析到一个路径参数配置并绑定到相应的路径：**

**boolean** pathsMatch(**String** path, ServletRequest request);  
**boolean** onPreHandle(ServletRequest request, ServletResponse response, **Object** mappedValue) **throws Exception**;

pathsMatch：该方法用于 path 与请求路径进行匹配的方法；如果匹配返回 true；  
onPreHandle：在 preHandle 中，当 pathsMatch 匹配一个路径后，会调用 opPreHandler 方法并将路径绑定参数配置传给 mappedValue；然后可以在这个方法中进行一些验证（如角色授权），如果验证失败可以返回 false 中断流程；默认返回 true；也就是说子类可以只实现 onPreHandle 即可，无须实现 preHandle。如果没有 path 与请求路径匹配，默认是通过的（即 preHandle 返回 true）。

* 1. **AccessControlFilter**

AccessControlFilter 提供了访问控制的基础功能；比如是否允许访问/当访问拒绝时如何处理等：

**abstract boolean** isAccessAllowed(ServletRequest request, ServletResponse response, **Object** mappedValue) **throws Exception**;  
**boolean** onAccessDenied(ServletRequest request, ServletResponse response, **Object** mappedValue) **throws Exception**;  
**abstract boolean** onAccessDenied(ServletRequest request, ServletResponse response) **throws Exception**;

isAccessAllowed：表示是否允许访问；mappedValue 就是[urls]配置中拦截器参数部分，如果允许访问返回 true，否则 false；

onAccessDenied：表示当访问拒绝时是否已经处理了；如果返回 true 表示需要继续处理；如果返回 false 表示该拦截器实例已经处理了，将直接返回即可。

onPreHandle 会自动调用这两个方法决定是否继续处理：

**boolean** onPreHandle(ServletRequest request, ServletResponse response, **Object** mappedValue) **throws Exception {  
 return isAccessAllowed**(**request**, **response**, **mappedValue**) **|| onAccessDenied**(**request**, **response**, **mappedValue**);  
**}**

另外 AccessControlFilter 还提供了如下方法用于处理如登录成功后/重定向到上一个请求：

**void** setLoginUrl(**String** loginUrl) **//身份验证时使用，默认/login.jsp  
String** getLoginUrl()  
**Subject** getSubject(ServletRequest request, ServletResponse response) **//获取Subject 实例  
boolean** isLoginRequest(ServletRequest request, ServletResponse response)**//当前请求是否是登录请求  
void** saveRequestAndRedirectToLogin(ServletRequest request, ServletResponse response) **throws IOException //将当前请求保存起来并重定向到登录页面  
void** saveRequest(ServletRequest request) **//将请求保存起来，如登录成功后再重定向回该请求  
void** redirectToLogin(ServletRequest request, ServletResponse response) **//重定向到登录页面**

比如基于表单的身份验证就需要使用这些功能。

到此基本的拦截器就完事了，如果我们想进行访问访问的控制就可以继承 AccessControlFilter；如果我们要添加一些通用数据我们可以直接继承 PathMatchingFilter。

shiro中,如果定义某个url有多个角色可以访问roles[role1,role2],则用户比如同时拥有role1和role2两个角色。重写拦截器，让用户拥有其中任意一个角色就能访问：

**package com.shiro.test.mvc**;  
  
**import org.apache.shiro.subject.**Subject;  
**import org.apache.shiro.web.filter.authz.**AuthorizationFilter;  
  
**import javax.servlet.**ServletRequest;  
**import javax.servlet.**ServletResponse;  
  
**public class MyRolesFilter extends** AuthorizationFilter **{  
 @Override  
 protected boolean** isAccessAllowed  
 (ServletRequest **req**, ServletResponse **resp**, **Object mappedValue**)  
 **throws Exception {** Subject **subject =** getSubject(**req**, **resp**);  
 **String**[] **roles =** (**String**[]) **mappedValue**;  
  
 **if** (**roles == null || roles**.**length == 0**) **{ //没有权限限制  
 return true**;  
 **}  
 for** (**int i = 0**; **i < roles**.**length**; **i++**) **{  
 //若当前用户是roles中的任何一个，则有权限访问  
 if** (**subject**.**hasRole**(**roles**[**i**])) **{  
 return true**;  
 **}  
 }  
 return false**;  
 **}  
  
}**

<property name="filters">  
 <map>  
 <entry key="roles">  
 <bean class="com.shiro.test.mvc.filter.MyShiroFilter"/>  
 </entry>  
 </map>  
</property>

|  |  |  |
| --- | --- | --- |
| 默认拦截器名 | 拦截器类 | 说明（括号里的表示默认值） |
| 身份验证相关的 |  |  |
| authc | org.apache.shiro.web.filter.authc | 基于表单的拦截器；如 “`/\*\*=authc`”，如果没有登录会跳到相应的登录页面登录；主要属性：usernameParam：表单提交的用户名参数名（ username）；  passwordParam：表单提交的密码参数名（password）； rememberMeParam：表单提交的密码参数名（rememberMe）；  loginUrl：登录页面地址（/login.jsp）；successUrl：登录成功后的默认重定向地址； failureKeyAttribute：登录失败后错误信息存储 key（shiroLoginFailure）； |
|  |
| .FormAuthenticationFilter |
| authcBasic | org.apache.shiro.web.filter.authc | Basic HTTP 身份验证拦截器，主要属性： applicationName：弹出登录框显示的信息（application）； |
|  |
| .BasicHttpAuthenticationFilter |
| logout | org.apache.shiro.web.filter.authc | 退出拦截器，主要属性：redirectUrl：退出成功后重定向的地址（/）; 示例 “/logout=logout” |
|  |
| .LogoutFilter |
| user | org.apache.shiro.web.filter.authc | 用户拦截器，用户已经身份验证 / 记住我登录的都可；示例 “/\*\*=user” |
|  |
| .UserFilter |
| anon | org.apache.shiro.web.filter.authc | 匿名拦截器，即不需要登录即可访问；一般用于静态资源过滤；示例 “/static/\*\*=anon” |
|  |
| .AnonymousFilter |
| **授权相关的** |  |  |
| roles | org.apache.shiro.web.filter.authz | 角色授权拦截器，验证用户是否拥有所有角色；主要属性： loginUrl：登录页面地址（/login.jsp）；unauthorizedUrl：未授权后重定向的地址；示例 “/admin/\*\*=roles[admin]” |
|  |
| .RolesAuthorizationFilter |
| perms | org.apache.shiro.web.filter.authz | 权限授权拦截器，验证用户是否拥有所有权限；属性和 roles 一样；示例 “/user/\*\*=perms["user:create"]” |
|  |
| .PermissionsAuthorizationFilter |
| port | org.apache.shiro.web.filter.authz | 端口拦截器，主要属性：port（80）：可以通过的端口；示例 “/test= port[80]”，如果用户访问该页面是非 80，将自动将请求端口改为 80 并重定向到该 80 端口，其他路径 / 参数等都一样 |
|  |
| .PortFilter |
| rest | org.apache.shiro.web.filter.authz | rest 风格拦截器，自动根据请求方法构建权限字符串（GET=read, POST=create,PUT=update,DELETE=delete,HEAD=read,TRACE=read,OPTIONS=read, MKCOL=create）构建权限字符串；示例 “/users=rest[user]”，会自动拼出“user:read,user:create,user:update,user:delete” 权限字符串进行权限匹配（所有都得匹配，isPermittedAll）； |
|  |
| .HttpMethodPermissionFilter |
| ssl | org.apache.shiro.web.filter.authz | SSL 拦截器，只有请求协议是 https 才能通过；否则自动跳转会 https 端口（443）；其他和 port 拦截器一样； |
|  |
| .SslFilter |
| **其他** |  |  |
| noSessionCreation | org.apache.shiro.web.filter.session | 不创建会话拦截器，调用 subject.getSession(false) 不会有什么问题，但是如果 subject.getSession(true) 将抛出 DisabledSessionException 异常； |
|  |
| .NoSessionCreationFilter |

# 动态验证规则

**package com.shiro.test.mvc**;  
  
**import org.apache.shiro.config.Ini**;  
**import org.apache.shiro.spring.web.ShiroFilterFactoryBean**;  
**import org.apache.shiro.util.CollectionUtils**;  
**import org.apache.shiro.web.filter.mgt.DefaultFilterChainManager**;  
**import org.apache.shiro.web.filter.mgt.PathMatchingFilterChainResolver**;  
**import org.apache.shiro.web.servlet.**AbstractShiroFilter;  
  
**import java.text.MessageFormat**;  
**import java.util.\***;  
  
**public class MyShiroFilterFactoryBean extends ShiroFilterFactoryBean {  
 public static final String *PREMISSION\_STRING* = "perms[{0}]"**;  
 **public static final String *ROLE\_STRING* = "roles[{0}]"**;  
 **private static String** filterChainDefinitions;  
  
 **@Override  
 public void** setFilterChainDefinitions(**String definitions**) **{** filterChainDefinitions **= definitions**;  
 **Ini ini = new** Ini();  
 **ini**.load(**definitions**);  
 **Ini**.**Section section = ini**.getSection(**"urls"**);  
 **if** (**CollectionUtils**.isEmpty(**section**)) **{  
 section = ini**.getSection(**""**);  
 **}** Set**<String> urls = new** LinkedHashSet**<>**();  
 **urls**.**add**(**"/dotest1.html"**);  
 **urls**.**add**(**"/dotest2.html"**);  
 **//每个url对应的权限也可以从数据库中查出来，这里模拟几条数据** Map**<String**, **String**[]**> permsMap = new** HashMap**<>**();  
 **permsMap**.**put**(**"/dotest1.html"**, **new** String[]**{"test"}**);  
 **permsMap**.**put**(**"/dotest2.html"**, **new** String[]**{"admin"}**);  
  
 **for** (**String url : urls**) **{  
 String**[] **perms = permsMap**.**get**(**url**);  
 **StringBuilder permFilters = new** StringBuilder();  
 **for** (**int i = 0**; **i < perms**.**length**; **i++**) **{  
 permFilters**.append(**perms**[**i**]).append(**","**);  
 **}  
 //去掉末尾的逗号  
 String str = permFilters**.substring(**0**, **permFilters**.length() **- 1**);  
 **//生成结果如：/dotest1.html = authc, roles[admin,test]  
 section**.put(**url**, **MessageFormat**.format(***ROLE\_STRING***, **str**));  
 **}  
 this**.setFilterChainDefinitionMap(**section**);  
 **}  
  
 //动态授权后不需要重启服务器  
 public void** update() **{  
 synchronized** (**this**) **{ //强制同步，控制线程安全  
 try {** AbstractShiroFilter **shiroFilter =** (AbstractShiroFilter) **this**.getObject();  
 **PathMatchingFilterChainResolver resolver =** (**PathMatchingFilterChainResolver**) **shiroFilter** .getFilterChainResolver();  
 **// 过滤管理器  
 DefaultFilterChainManager manager =** (**DefaultFilterChainManager**) **resolver**.getFilterChainManager();  
 **// 清除权限配置  
 manager**.getFilterChains().**clear**();  
 **this**.getFilterChainDefinitionMap().**clear**();  
 **// 重新设置权限  
 this**.setFilterChainDefinitions(filterChainDefinitions);  
  
 Map**<String**, **String> chains = this**.getFilterChainDefinitionMap();  
 **if** (**!CollectionUtils**.isEmpty(**chains**)) **{** Iterator **var12 = chains**.**entrySet**().**iterator**();  
 **while** (**var12**.**hasNext**()) **{** Map.Entry**<String**, **String> entry =** (Map.Entry) **var12**.**next**();  
 **String url =** (**String**) **entry**.**getKey**();  
 **String chainDefinition =** (**String**) **entry**.**getValue**();  
 **manager**.createChain(**url**, **chainDefinition**);  
 **}  
 }  
 } catch** (**Exception e**) **{  
 e**.printStackTrace();  
 **}  
 }  
  
 }  
}**

# 缓存

如 securityManager 实现了 SessionsSecurityManager，其会自动判断 SessionManager 是否实现了 CacheManagerAware 接口，如果实现了会把 CacheManager 设置给它。然后 sessionManager 会判断相应的 sessionDAO（如继承自 CachingSessionDAO）是否实现了 CacheManagerAware，如果实现了会把 CacheManager 设置给它；如第九章的 MySessionDAO 就是带缓存的 SessionDAO；其会先查缓存，如果找不到才查数据库。

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

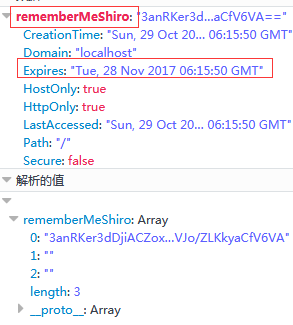
# RememberMe

rememberMe可以在浏览器中设置cookie，在spring配置中可以设置cookie的属性，如过期时间、cookie名字、加密的秘钥等：

<bean id="rememberMeCookie" class="org.apache.shiro.web.servlet.SimpleCookie">  
 <constructor-arg value="rememberMeShiro"/><!-- 浏览器中cookie的名字 -->  
 <property name="httpOnly" value="true"/><!--document对象中就看不到cookie了-->  
 <property name="maxAge" value="2592000"/><!-- 30天 -->  
</bean>  
  
<!-- rememberMe管理器 -->  
<bean id="rememberMeManager" class="org.apache.shiro.web.mgt.CookieRememberMeManager">  
 <!--秘钥要16位，24位或32位的Base64。这个解密后是1234567890abcdef-->  
 <property name="cipherKey" value="#{T(org.apache.shiro.codec.Base64).decode('MTIzNDU2Nzg5MGFiY2RlZg==')}"/>  
 <property name="cookie" ref="rememberMeCookie"/>  
</bean>

在securityManager中加入rememberMe中加入配置：

<bean id="securityManager" class="org.apache.shiro.web.mgt.DefaultWebSecurityManager">  
 <property name="realm" ref="myShiroRealm"/>  
 <!--加入rememberMe的设置-->  
 <property name="rememberMeManager" ref="rememberMeManager"/>  
</bean>



HttpOnly属性：

浏览器中通过document.cookie可以获取cookie属性，设置了HttpOnly=true,在脚本中就不能的到cookie了。可以避免cookie被盗用。