# 第一[章 Shiro简介——《跟我学Shiro》](http://jinnianshilongnian.iteye.com/blog/2018936)

## 1.1  简介

Apache Shiro是Java的一个安全框架。目前，使用Apache Shiro的人越来越多，因为它相当简单，对比Spring Security，可能没有Spring Security做的功能强大，但是在实际工作时可能并不需要那么复杂的东西，所以使用小而简单的Shiro就足够了。对于它俩到底哪个好，这个不必纠结，能更简单的解决项目问题就好了。

本教程只介绍基本的Shiro使用，不会过多分析源码等，重在使用。

Shiro可以非常容易的开发出足够好的应用，其不仅可以用在JavaSE环境，也可以用在JavaEE环境。Shiro可以帮助我们完成：认证、授权、加密、会话管理、与Web集成、缓存等。这不就是我们想要的嘛，而且Shiro的API也是非常简单；其基本功能点如下图所示：



**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。

**记住一点，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。**

接下来我们分别从外部和内部来看看Shiro的架构，对于一个好的框架，从外部来看应该具有非常简单易于使用的API，且API契约明确；从内部来看的话，其应该有一个可扩展的架构，即非常容易插入用户自定义实现，因为任何框架都不能满足所有需求。

首先，我们从外部来看Shiro吧，即从应用程序角度的来观察如何使用Shiro完成工作。如下图：



可以看到：应用代码直接交互的对象是Subject，也就是说Shiro的对外API核心就是Subject；其每个API的含义：

**Subject：**主体，代表了当前“用户”，这个用户不一定是一个具体的人，与当前应用交互的任何东西都是Subject，如网络爬虫，机器人等；即一个抽象概念；所有Subject都绑定到SecurityManager，与Subject的所有交互都会委托给SecurityManager；可以把Subject认为是一个门面；SecurityManager才是实际的执行者；

**SecurityManager：**安全管理器；即所有与安全有关的操作都会与SecurityManager交互；且它管理着所有Subject；可以看出它是Shiro的核心，它负责与后边介绍的其他组件进行交互，如果学习过SpringMVC，你可以把它看成DispatcherServlet前端控制器；

**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。

也就是说对于我们而言，最简单的一个Shiro应用：

1、应用代码通过Subject来进行认证和授权，而Subject又委托给SecurityManager；

2、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。

**从以上也可以看出，Shiro不提供维护用户/权限，而是通过Realm让开发人员自己注入。**

接下来我们来从Shiro内部来看下Shiro的架构，如下图所示：



**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、EJB等环境；所有呢，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；这样的话，比如我们在Web环境用，刚开始是一台Web服务器；接着又上了台EJB服务器；这时想把两台服务器的会话数据放到一个地方，这个时候就可以实现自己的分布式会话（如把数据放到Memcached服务器）；

**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。

到此Shiro架构及其组件就认识完了，接下来挨着学习Shiro的组件吧。

参考<http://shiro.apache.org/reference.html>

示例源代码：<https://github.com/zhangkaitao/shiro-example>；可加群134755960探讨Spring/Shiro技术。

# [第二章 身份验证——《跟我学Shiro》](http://jinnianshilongnian.iteye.com/blog/2019547)

**身份验证**，即在应用中谁能证明他就是他本人。一般提供如他们的身份ID一些标识信息来表明他就是他本人，如提供身份证，用户名/密码来证明。

在shiro中，用户需要提供principals （身份）和credentials（证明）给shiro，从而应用能验证用户身份：

**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。

最常见的principals和credentials组合就是用户名/密码了。接下来先进行一个基本的身份认证。

另外两个相关的概念是之前提到的**Subject**及**Realm**，分别是主体及验证主体的数据源。

**2.2  环境准备**

本文使用Maven构建，因此需要一点Maven知识。首先准备环境依赖：

**Java代码  [收藏代码](javascript:void())**

1. <dependencies>
2. <dependency>
3. <groupId>junit</groupId>
4. <artifactId>junit</artifactId>
5. <version>4.9</version>
6. </dependency>
7. <dependency>
8. <groupId>commons-logging</groupId>
9. <artifactId>commons-logging</artifactId>
10. <version>1.1.3</version>
11. </dependency>
12. <dependency>
13. <groupId>org.apache.shiro</groupId>
14. <artifactId>shiro-core</artifactId>
15. <version>1.2.2</version>
16. </dependency>
17. </dependencies>

添加junit、common-logging及shiro-core依赖即可。

**2.3  登录/退出**

1、首先准备一些用户身份/凭据（shiro.ini）

**Java代码  [收藏代码](javascript:void())**

1. [users]
2. zhang=123
3. wang=123

此处使用ini配置文件，通过[users]指定了两个主体：zhang/123、wang/123。

  注意：

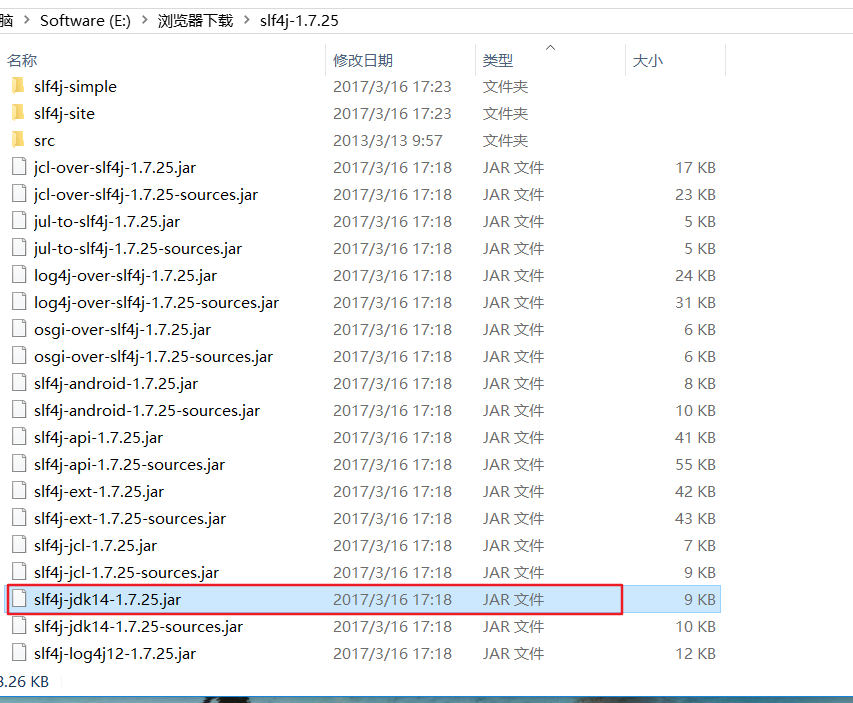
 1.   等号左边和右边不能换行，保存格式改为ANSI编码

2.MAVEN 工程中 classpath是src/main/resources(resources区分大小写)

3. classpath的查找范围应该是/WEB-INF/classes，资源文件的缺省路径为src/main/resources，Maven在打包成war文件的时候，会将src/main/resources的资源文件复制到class目录。

4.报错，无法load一个类，导入jar包

SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".  
SLF4J: Defaulting to no-operation (NOP) logger implementation  
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.



2、测试用例（com.github.zhangkaitao.shiro.chapter2.LoginLogoutTest）

**Java代码  [收藏代码](javascript:void())**

1. @Test
2. **public** **void** testHelloworld() {
3. //1、获取SecurityManager工厂，此处使用Ini配置文件初始化SecurityManager
4. Factory<org.apache.shiro.mgt.SecurityManager> factory =
5. **new** IniSecurityManagerFactory("classpath:shiro.ini");
6. //2、得到SecurityManager实例 并绑定给SecurityUtils
7. org.apache.shiro.mgt.SecurityManager securityManager = factory.getInstance();
8. SecurityUtils.setSecurityManager(securityManager);
9. //3、得到Subject及创建用户名/密码身份验证Token（即用户身份/凭证）
10. Subject subject = SecurityUtils.getSubject();
11. UsernamePasswordToken token = **new** UsernamePasswordToken("zhang", "123");
13. **try** {
14. //4、登录，即身份验证
15. subject.login(token);
16. } **catch** (AuthenticationException e) {
17. //5、身份验证失败
18. }
20. Assert.assertEquals(**true**, subject.isAuthenticated()); //断言用户已经登录
22. //6、退出
23. subject.logout();
24. }

2.1、首先通过new IniSecurityManagerFactory并指定一个ini配置文件来创建一个SecurityManager工厂；

2.2、接着获取SecurityManager并绑定到SecurityUtils，这是一个全局设置，设置一次即可；

2.3、通过SecurityUtils得到Subject，其会自动绑定到当前线程；如果在web环境在请求结束时需要解除绑定；然后获取身份验证的Token，如用户名/密码；

2.4、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；

2.5、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；

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

**从如上代码可总结出身份验证的步骤：**

1、收集用户身份/凭证，即如用户名/密码；

2、调用Subject.login进行登录，如果失败将得到相应的AuthenticationException异常，根据异常提示用户错误信息；否则登录成功；

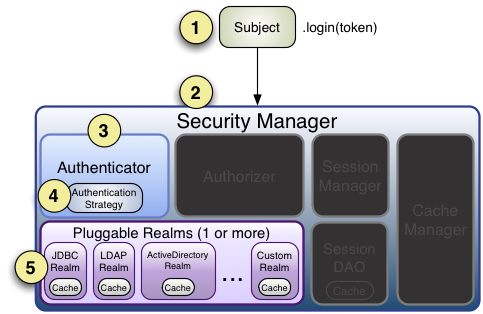
3、最后调用Subject.logout进行退出操作。

如上测试的几个问题：

1、用户名/密码硬编码在ini配置文件，以后需要改成如数据库存储，且密码需要加密存储；

2、用户身份Token可能不仅仅是用户名/密码，也可能还有其他的，如登录时允许用户名/邮箱/手机号同时登录。

**2.4  身份认证流程**



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。

**2.5  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。如我们之前的ini配置方式将使用org.apache.shiro.realm.text.IniRealm。

org.apache.shiro.realm.Realm接口如下：

**Java代码  [收藏代码](javascript:void())**

1. String getName(); //返回一个唯一的Realm名字
2. **boolean** supports(AuthenticationToken token); //判断此Realm是否支持此Token
3. AuthenticationInfo getAuthenticationInfo(AuthenticationToken token)
4. **throws** AuthenticationException;  //根据Token获取认证信息

**单Realm配置**

1、自定义Realm实现（com.github.zhangkaitao.shiro.chapter2.realm.MyRealm1）：

**Java代码  [收藏代码](javascript:void())**

1. **public** **class** MyRealm1 **implements** Realm {
2. @Override
3. **public** String getName() {
4. **return** "myrealm1";
5. }
6. @Override
7. **public** **boolean** supports(AuthenticationToken token) {
8. //仅支持UsernamePasswordToken类型的Token
9. **return** token **instanceof** UsernamePasswordToken;
10. }
11. @Override
12. **public** AuthenticationInfo getAuthenticationInfo(AuthenticationToken token) **throws** AuthenticationException {
13. String username = (String)token.getPrincipal();  //得到用户名
14. String password = **new** String((**char**[])token.getCredentials()); //得到密码
15. **if**(!"zhang".equals(username)) {
16. **throw** **new** UnknownAccountException(); //如果用户名错误
17. }
18. **if**(!"123".equals(password)) {
19. **throw** **new** IncorrectCredentialsException(); //如果密码错误
20. }
21. //如果身份认证验证成功，返回一个AuthenticationInfo实现；
22. **return** **new** SimpleAuthenticationInfo(username, password, getName());
23. }
24. }

2、ini配置文件指定自定义Realm实现(shiro-realm.ini)

**Java代码  [收藏代码](javascript:void())**

1. #声明一个realm
2. myRealm1=com.github.zhangkaitao.shiro.chapter2.realm.MyRealm1
3. #指定securityManager的realms实现
4. securityManager.realms=$myRealm1

通过$name来引入之前的realm定义

3、测试用例请参考com.github.zhangkaitao.shiro.chapter2.LoginLogoutTest的testCustomRealm测试方法，只需要把之前的shiro.ini配置文件改成shiro-realm.ini即可。

**多Realm配置**

1、ini配置文件（shiro-multi-realm.ini）

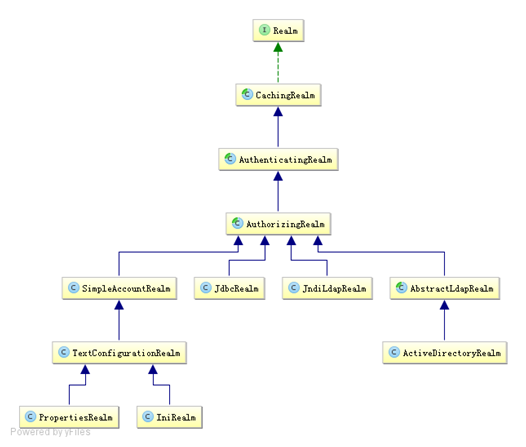
**Java代码  [收藏代码](javascript:void())**

1. #声明一个realm
2. myRealm1=com.github.zhangkaitao.shiro.chapter2.realm.MyRealm1
3. myRealm2=com.github.zhangkaitao.shiro.chapter2.realm.MyRealm2
4. #指定securityManager的realms实现
5. 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。

2、测试用例请参考com.github.zhangkaitao.shiro.chapter2.LoginLogoutTest的testCustomMultiRealm测试方法。

**Shiro默认提供的Realm**



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：

**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；

**JDBC Realm使用**

1、数据库及依赖

**Java代码  [收藏代码](javascript:void())**

1. <dependency>
2. <groupId>mysql</groupId>
3. <artifactId>mysql-connector-java</artifactId>
4. <version>5.1.25</version>
5. </dependency>
6. <dependency>
7. <groupId>com.alibaba</groupId>
8. <artifactId>druid</artifactId>
9. <version>0.2.23</version>
10. </dependency>

本文将使用mysql数据库及druid连接池；

2、到数据库shiro下建三张表：users（用户名/密码）、user\_roles（用户/角色）、roles\_permissions（角色/权限），具体请参照shiro-example-chapter2/sql/shiro.sql；并添加一个用户记录，用户名/密码为zhang/123；

3、ini配置（shiro-jdbc-realm.ini）

**Java代码  [收藏代码](javascript:void())**

1. jdbcRealm=org.apache.shiro.realm.jdbc.JdbcRealm
2. dataSource=com.alibaba.druid.pool.DruidDataSource
3. dataSource.driverClassName=com.mysql.jdbc.Driver
4. dataSource.url=jdbc:mysql://localhost:3306/shiro
5. dataSource.username=root
6. #dataSource.password=
7. jdbcRealm.dataSource=$dataSource
8. securityManager.realms=$jdbcRealm

注意:#dataSource.password= ,需要解除注释，并且加上自己数据库密码

1、变量名=全限定类名会自动创建一个类实例

2、变量名.属性=值 自动调用相应的setter方法进行赋值

3、$变量名 引用之前的一个对象实例

4、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。

**2.6  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：

**Java代码  [收藏代码](javascript:void())**

1. **public** AuthenticationInfo authenticate(AuthenticationToken authenticationToken)
2. **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。

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。

假设我们有三个realm：

myRealm1： 用户名/密码为zhang/123时成功，且返回身份/凭据为zhang/123；

myRealm2： 用户名/密码为wang/123时成功，且返回身份/凭据为wang/123；

myRealm3： 用户名/密码为zhang/123时成功，且返回身份/凭据为zhang@163.com/123，和myRealm1不同的是返回时的身份变了；

1、ini配置文件(shiro-authenticator-all-success.ini)

**Java代码  [收藏代码](javascript:void())**

1. #指定securityManager的authenticator实现
2. authenticator=org.apache.shiro.authc.pam.ModularRealmAuthenticator
3. securityManager.authenticator=$authenticator
5. #指定securityManager.authenticator的authenticationStrategy
6. allSuccessfulStrategy=org.apache.shiro.authc.pam.AllSuccessfulStrategy
7. securityManager.authenticator.authenticationStrategy=$allSuccessfulStrategy

**Java代码  [收藏代码](javascript:void())**

1. myRealm1=com.github.zhangkaitao.shiro.chapter2.realm.MyRealm1
2. myRealm2=com.github.zhangkaitao.shiro.chapter2.realm.MyRealm2
3. myRealm3=com.github.zhangkaitao.shiro.chapter2.realm.MyRealm3
4. securityManager.realms=$myRealm1,$myRealm3

2、测试代码（com.github.zhangkaitao.shiro.chapter2.AuthenticatorTest）

2.1、首先通用化登录逻辑

**Java代码  [收藏代码](javascript:void())**

1. **private** **void** login(String configFile) {
2. //1、获取SecurityManager工厂，此处使用Ini配置文件初始化SecurityManager
3. Factory<org.apache.shiro.mgt.SecurityManager> factory =
4. **new** IniSecurityManagerFactory(configFile);
6. //2、得到SecurityManager实例 并绑定给SecurityUtils
7. org.apache.shiro.mgt.SecurityManager securityManager = factory.getInstance();
8. SecurityUtils.setSecurityManager(securityManager);
10. //3、得到Subject及创建用户名/密码身份验证Token（即用户身份/凭证）
11. Subject subject = SecurityUtils.getSubject();
12. UsernamePasswordToken token = **new** UsernamePasswordToken("zhang", "123");
14. subject.login(token);
15. }

2.2、测试AllSuccessfulStrategy成功：

**Java代码  [收藏代码](javascript:void())**

1. @Test
2. **public** **void** testAllSuccessfulStrategyWithSuccess() {
3. login("classpath:shiro-authenticator-all-success.ini");
4. Subject subject = SecurityUtils.getSubject();
6. //得到一个身份集合，其包含了Realm验证成功的身份信息
7. PrincipalCollection principalCollection = subject.getPrincipals();
8. Assert.assertEquals(2, principalCollection.asList().size());
9. }

即PrincipalCollection包含了zhang和zhang@163.com身份信息。

2.3、测试AllSuccessfulStrategy失败：

**Java代码  [收藏代码](javascript:void())**

1. @Test(expected = UnknownAccountException.**class**)
2. **public** **void** testAllSuccessfulStrategyWithFail() {
3. login("classpath:shiro-authenticator-all-fail.ini");
4. Subject subject = SecurityUtils.getSubject();
5. }

shiro-authenticator-all-fail.ini与shiro-authenticator-all-success.ini不同的配置是使用了securityManager.realms=$myRealm1,$myRealm2；即myRealm验证失败。

对于AtLeastOneSuccessfulStrategy和FirstSuccessfulStrategy的区别，请参照testAtLeastOneSuccessfulStrategyWithSuccess和testFirstOneSuccessfulStrategyWithSuccess测试方法。唯一不同点一个是返回所有验证成功的Realm的认证信息；另一个是只返回第一个验证成功的Realm的认证信息。

自定义AuthenticationStrategy实现，首先看其API：

**Java代码  [收藏代码](javascript:void())**

1. //在所有Realm验证之前调用
2. AuthenticationInfo beforeAllAttempts(
3. Collection<? **extends** Realm> realms, AuthenticationToken token)
4. **throws** AuthenticationException;
5. //在每个Realm之前调用
6. AuthenticationInfo beforeAttempt(
7. Realm realm, AuthenticationToken token, AuthenticationInfo aggregate)
8. **throws** AuthenticationException;
9. //在每个Realm之后调用
10. AuthenticationInfo afterAttempt(
11. Realm realm, AuthenticationToken token,
12. AuthenticationInfo singleRealmInfo, AuthenticationInfo aggregateInfo, Throwable t)
13. **throws** AuthenticationException;
14. //在所有Realm之后调用
15. AuthenticationInfo afterAllAttempts(
16. AuthenticationToken token, AuthenticationInfo aggregate)
17. **throws** AuthenticationException;

因为每个AuthenticationStrategy实例都是无状态的，所有每次都通过接口将相应的认证信息传入下一次流程；通过如上接口可以进行如合并/返回第一个验证成功的认证信息。

自定义实现时一般继承org.apache.shiro.authc.pam.AbstractAuthenticationStrategy即可，具体可以参考代码com.github.zhangkaitao.shiro.chapter2.authenticator.strategy包下OnlyOneAuthenticatorStrategy 和AtLeastTwoAuthenticatorStrategy。

到此基本的身份验证就搞定了，对于AuthenticationToken 、AuthenticationInfo和Realm的详细使用后续章节再陆续介绍。

# [第三章 授权——《跟我学Shiro》](http://jinnianshilongnian.iteye.com/blog/2020017)

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。

请google搜索“RBAC”和“RBAC新解”分别了解“基于角色的访问控制”“基于资源的访问控制(Resource-Based Access Control)”。

## 3.1 授权方式

Shiro支持三种方式的授权：

编程式：通过写if/else授权代码块完成：

**Java代码  [收藏代码](javascript:void())**

1. Subject subject = SecurityUtils.getSubject();
2. **if**(subject.hasRole(“admin”)) {
3. //有权限
4. } **else** {
5. //无权限
6. }

注解式：通过在执行的Java方法上放置相应的注解完成：

**Java代码  [收藏代码](javascript:void())**

1. @RequiresRoles("admin")
2. **public** **void** hello() {
3. //有权限
4. }

没有权限将抛出相应的异常；

JSP/GSP标签：在JSP/GSP页面通过相应的标签完成：

**Java代码  [收藏代码](javascript:void())**

1. <shiro:hasRole name="admin">
2. <!— 有权限 —>
3. </shiro:hasRole>

后续部分将详细介绍如何使用。

## 3.2 授权

**基于角色的访问控制（隐式角色）**

1、在ini配置文件配置用户拥有的角色（shiro-role.ini）

**Java代码  [收藏代码](javascript:void())**

1. [users]
2. zhang=123,role1,role2
3. wang=123,role1

规则即：“用户名=密码,角色1，角色2”，如果需要在应用中判断用户是否有相应角色，就需要在相应的Realm中返回角色信息，也就是说Shiro不负责维护用户-角色信息，需要应用提供，Shiro只是提供相应的接口方便验证，后续会介绍如何动态的获取用户角色。

2、测试用例（com.github.zhangkaitao.shiro.chapter3.RoleTest）

**Java代码  [收藏代码](javascript:void())**

1. @Test
2. **public** **void** testHasRole() {
3. login("classpath:shiro-role.ini", "zhang", "123");
4. //判断拥有角色：role1
5. Assert.assertTrue(subject().hasRole("role1"));
6. //判断拥有角色：role1 and role2
7. Assert.assertTrue(subject().hasAllRoles(Arrays.asList("role1", "role2")));
8. //判断拥有角色：role1 and role2 and !role3
9. **boolean**[] result = subject().hasRoles(Arrays.asList("role1", "role2", "role3"));
10. Assert.assertEquals(**true**, result[0]);
11. Assert.assertEquals(**true**, result[1]);
12. Assert.assertEquals(**false**, result[2]);
13. }

Shiro提供了hasRole/hasRole用于判断用户是否拥有某个角色/某些权限；但是没有提供如hashAnyRole用于判断是否有某些权限中的某一个。

**Java代码  [收藏代码](javascript:void())**

1. @Test(expected = UnauthorizedException.**class**)
2. **public** **void** testCheckRole() {
3. login("classpath:shiro-role.ini", "zhang", "123");
4. //断言拥有角色：role1
5. subject().checkRole("role1");
6. //断言拥有角色：role1 and role3 失败抛出异常
7. subject().checkRoles("role1", "role3");
8. }

Shiro提供的checkRole/checkRoles和hasRole/hasAllRoles不同的地方是它在判断为假的情况下会抛出UnauthorizedException异常。

到此基于角色的访问控制（即隐式角色）就完成了，这种方式的缺点就是如果很多地方进行了角色判断，但是有一天不需要了那么就需要修改相应代码把所有相关的地方进行删除；这就是粗粒度造成的问题。

**基于资源的访问控制（显示角色）**

1、在ini配置文件配置用户拥有的角色及角色-权限关系（shiro-permission.ini）

**Java代码  [收藏代码](javascript:void())**

1. [users]
2. zhang=123,role1,role2
3. wang=123,role1
4. [roles]
5. role1=user:create,user:update
6. role2=user:create,user:delete

规则：“用户名=密码，角色1，角色2”“角色=权限1，权限2”，即首先根据用户名找到角色，然后根据角色再找到权限；即角色是权限集合；Shiro同样不进行权限的维护，需要我们通过Realm返回相应的权限信息。只需要维护“用户——角色”之间的关系即可。

2、测试用例（com.github.zhangkaitao.shiro.chapter3.PermissionTest）

**Java代码  [收藏代码](javascript:void())**

1. @Test
2. **public** **void** testIsPermitted() {
3. login("classpath:shiro-permission.ini", "zhang", "123");
4. //判断拥有权限：user:create
5. Assert.assertTrue(subject().isPermitted("user:create"));
6. //判断拥有权限：user:update and user:delete
7. Assert.assertTrue(subject().isPermittedAll("user:update", "user:delete"));
8. //判断没有权限：user:view
9. Assert.assertFalse(subject().isPermitted("user:view"));
10. }

Shiro提供了isPermitted和isPermittedAll用于判断用户是否拥有某个权限或所有权限，也没有提供如isPermittedAny用于判断拥有某一个权限的接口。

**Java代码  [收藏代码](javascript:void())**

1. @Test(expected = UnauthorizedException.**class**)
2. **public** **void** testCheckPermission () {
3. login("classpath:shiro-permission.ini", "zhang", "123");
4. //断言拥有权限：user:create
5. subject().checkPermission("user:create");
6. //断言拥有权限：user:delete and user:update
7. subject().checkPermissions("user:delete", "user:update");
8. //断言拥有权限：user:view 失败抛出异常
9. subject().checkPermissions("user:view");
10. }

但是失败的情况下会抛出UnauthorizedException异常。

到此基于资源的访问控制（显示角色）就完成了，也可以叫基于权限的访问控制，这种方式的一般规则是“资源标识符：操作”，即是资源级别的粒度；这种方式的好处就是如果要修改基本都是一个资源级别的修改，不会对其他模块代码产生影响，粒度小。但是实现起来可能稍微复杂点，需要维护“用户——角色，角色——权限（资源：操作）”之间的关系。

## 3.3 Permission

## ****字符串通配符权限****

规则：“资源标识符：操作：对象实例ID”  即对哪个资源的哪个实例可以进行什么操作。其默认支持通配符权限字符串，“:”表示资源/操作/实例的分割；“,”表示操作的分割；“\*”表示任意资源/操作/实例。

**1、单个资源单个权限**

**Java代码  [收藏代码](javascript:void())**

1. subject().checkPermissions("system:user:update");

用户拥有资源“system:user”的“update”权限。

**2、单个资源多个权限**

ini配置文件

**Java代码  [收藏代码](javascript:void())**

1. role41=system:user:update,system:user:delete

然后通过如下代码判断

**Java代码  [收藏代码](javascript:void())**

1. subject().checkPermissions("system:user:update", "system:user:delete");

用户拥有资源“system:user”的“update”和“delete”权限。如上可以简写成：

ini配置（表示角色4拥有system:user资源的update和delete权限）

**Java代码  [收藏代码](javascript:void())**

1. role42="system:user:update,delete"

 接着可以通过如下代码判断

**Java代码  [收藏代码](javascript:void())**

1. subject().checkPermissions("system:user:update,delete");

通过“system:user:update,delete”验证"system:user:update, system:user:delete"是没问题的，但是反过来是规则不成立。

**3、单个资源全部权限**

ini配置

**Java代码  [收藏代码](javascript:void())**

1. role51="system:user:create,update,delete,view"

然后通过如下代码判断

**Java代码  [收藏代码](javascript:void())**

1. subject().checkPermissions("system:user:create,delete,update:view");

用户拥有资源“system:user”的“create”、“update”、“delete”和“view”所有权限。如上可以简写成：

ini配置文件（表示角色5拥有system:user的所有权限）

**Java代码  [收藏代码](javascript:void())**

1. role52=system:user:\*

也可以简写为（推荐上边的写法）：

**Java代码  [收藏代码](javascript:void())**

1. role53=system:user

然后通过如下代码判断

**Java代码  [收藏代码](javascript:void())**

1. subject().checkPermissions("system:user:\*");
2. subject().checkPermissions("system:user");

通过“system:user:\*”验证“system:user:create,delete,update:view”可以，但是反过来是不成立的。

**4、所有资源全部权限**

ini配置

**Java代码  [收藏代码](javascript:void())**

1. role61=\*:view

然后通过如下代码判断

**Java代码  [收藏代码](javascript:void())**

1. subject().checkPermissions("user:view");

用户拥有所有资源的“view”所有权限。假设判断的权限是“"system:user:view”，那么需要“role5=\*:\*:view”这样写才行。

**5、实例级别的权限**

**5.1、单个实例单个权限**

ini配置

**Java代码  [收藏代码](javascript:void())**

1. role71=user:view:1

对资源user的1实例拥有view权限。

然后通过如下代码判断

**Java代码  [收藏代码](javascript:void())**

1. subject().checkPermissions("user:view:1");

**5.2、单个实例多个权限**

ini配置

**Java代码  [收藏代码](javascript:void())**

1. role72="user:update,delete:1"

对资源user的1实例拥有update、delete权限。

然后通过如下代码判断

**Java代码  [收藏代码](javascript:void())**

1. subject().checkPermissions("user:delete,update:1");
2. subject().checkPermissions("user:update:1", "user:delete:1");

**5.3、单个实例所有权限**

ini配置

**Java代码  [收藏代码](javascript:void())**

1. role73=user:\*:1

对资源user的1实例拥有所有权限。

然后通过如下代码判断

**Java代码  [收藏代码](javascript:void())**

1. subject().checkPermissions("user:update:1", "user:delete:1", "user:view:1");

**5.4、所有实例单个权限**

ini配置

**Java代码  [收藏代码](javascript:void())**

1. role74=user:auth:\*

对资源user的1实例拥有所有权限。

然后通过如下代码判断

**Java代码  [收藏代码](javascript:void())**

1. subject().checkPermissions("user:auth:1", "user:auth:2");

**5.5、所有实例所有权限**

ini配置

**Java代码  [收藏代码](javascript:void())**

1. role75=user:\*:\*

对资源user的1实例拥有所有权限。

然后通过如下代码判断

**Java代码  [收藏代码](javascript:void())**

1. subject().checkPermissions("user:view:1", "user:auth:2");

**6、Shiro对权限字符串缺失部分的处理**

如“user:view”等价于“user:view:\*”；而“organization”等价于“organization:\*”或者“organization:\*:\*”。可以这么理解，这种方式实现了前缀匹配。

另外如“user:\*”可以匹配如“user:delete”、“user:delete”可以匹配如“user:delete:1”、“user:\*:1”可以匹配如“user:view:1”、“user”可以匹配“user:view”或“user:view:1”等。即\*可以匹配所有，不加\*可以进行前缀匹配；但是如“\*:view”不能匹配“system:user:view”，需要使用“\*:\*:view”，即后缀匹配必须指定前缀（多个冒号就需要多个\*来匹配）。

**7、WildcardPermission**

如下两种方式是等价的：

**Java代码  [收藏代码](javascript:void())**

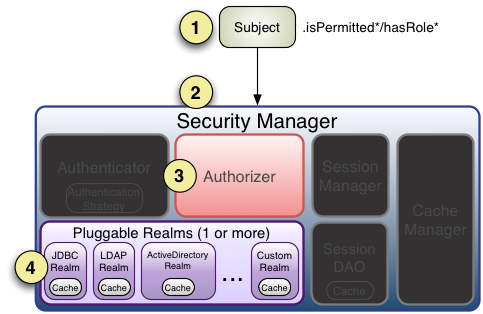
1. subject().checkPermission("menu:view:1");
2. subject().checkPermission(**new** WildcardPermission("menu:view:1"));

因此没什么必要的话使用字符串更方便。

**8、性能问题**

通配符匹配方式比字符串相等匹配来说是更复杂的，因此需要花费更长时间，但是一般系统的权限不会太多，且可以配合缓存来提供其性能，如果这样性能还达不到要求我们可以实现位操作算法实现性能更好的权限匹配。另外实例级别的权限验证如果数据量太大也不建议使用，可能造成查询权限及匹配变慢。可以考虑比如在sql查询时加上权限字符串之类的方式在查询时就完成了权限匹配。

## 3.4 授权流程



流程如下：

1、首先调用Subject.isPermitted\*/hasRole\*接口，其会委托给SecurityManager，而SecurityManager接着会委托给Authorizer；

2、Authorizer是真正的授权者，如果我们调用如isPermitted(“user:view”)，其首先会通过PermissionResolver把字符串转换成相应的Permission实例；

3、在进行授权之前，其会调用相应的Realm获取Subject相应的角色/权限用于匹配传入的角色/权限；

4、Authorizer会判断Realm的角色/权限是否和传入的匹配，如果有多个Realm，会委托给ModularRealmAuthorizer进行循环判断，如果匹配如isPermitted\*/hasRole\*会返回true，否则返回false表示授权失败。

ModularRealmAuthorizer进行多Realm匹配流程：

1、首先检查相应的Realm是否实现了实现了Authorizer；

2、如果实现了Authorizer，那么接着调用其相应的isPermitted\*/hasRole\*接口进行匹配；

3、如果有一个Realm匹配那么将返回true，否则返回false。

如果Realm进行授权的话，应该继承AuthorizingRealm，其流程是：

1.1、如果调用hasRole\*，则直接获取AuthorizationInfo.getRoles()与传入的角色比较即可；

1.2、首先如果调用如isPermitted(“user:view”)，首先通过PermissionResolver将权限字符串转换成相应的Permission实例，默认使用WildcardPermissionResolver，即转换为通配符的WildcardPermission；

2、通过AuthorizationInfo.getObjectPermissions()得到Permission实例集合；通过AuthorizationInfo. getStringPermissions()得到字符串集合并通过PermissionResolver解析为Permission实例；然后获取用户的角色，并通过RolePermissionResolver解析角色对应的权限集合（默认没有实现，可以自己提供）；

3、接着调用Permission. implies(Permission p)逐个与传入的权限比较，如果有匹配的则返回true，否则false。

## 3.5 Authorizer、PermissionResolver及RolePermissionResolver

Authorizer的职责是进行授权（访问控制），是Shiro API中授权核心的入口点，其提供了相应的角色/权限判断接口，具体请参考其Javadoc。SecurityManager继承了Authorizer接口，且提供了ModularRealmAuthorizer用于多Realm时的授权匹配。PermissionResolver用于解析权限字符串到Permission实例，而RolePermissionResolver用于根据角色解析相应的权限集合。

我们可以通过如下ini配置更改Authorizer实现：

**Java代码  [收藏代码](javascript:void())**

1. authorizer=org.apache.shiro.authz.ModularRealmAuthorizer
2. securityManager.authorizer=$authorizer

对于ModularRealmAuthorizer，相应的AuthorizingSecurityManager会在初始化完成后自动将相应的realm设置进去，我们也可以通过调用其setRealms()方法进行设置。对于实现自己的authorizer可以参考ModularRealmAuthorizer实现即可，在此就不提供示例了。

设置ModularRealmAuthorizer的permissionResolver，其会自动设置到相应的Realm上（其实现了PermissionResolverAware接口），如：

**Java代码  [收藏代码](javascript:void())**

1. permissionResolver=org.apache.shiro.authz.permission.WildcardPermissionResolver
2. authorizer.permissionResolver=$permissionResolver

设置ModularRealmAuthorizer的rolePermissionResolver，其会自动设置到相应的Realm上（其实现了RolePermissionResolverAware接口），如：

**Java代码  [收藏代码](javascript:void())**

1. rolePermissionResolver=com.github.zhangkaitao.shiro.chapter3.permission.MyRolePermissionResolver
2. authorizer.rolePermissionResolver=$rolePermissionResolver

**示例**

**1、ini配置（shiro-authorizer.ini）**

**Java代码  [收藏代码](javascript:void())**

1. [main]
2. #自定义authorizer
3. authorizer=org.apache.shiro.authz.ModularRealmAuthorizer
4. #自定义permissionResolver
5. #permissionResolver=org.apache.shiro.authz.permission.WildcardPermissionResolver
6. permissionResolver=com.github.zhangkaitao.shiro.chapter3.permission.BitAndWildPermissionResolver
7. authorizer.permissionResolver=$permissionResolver
8. #自定义rolePermissionResolver
9. rolePermissionResolver=com.github.zhangkaitao.shiro.chapter3.permission.MyRolePermissionResolver
10. authorizer.rolePermissionResolver=$rolePermissionResolver
12. securityManager.authorizer=$authorizer

**Java代码  [收藏代码](javascript:void())**

1. #自定义realm 一定要放在securityManager.authorizer赋值之后（因为调用setRealms会将realms设置给authorizer，并给各个Realm设置permissionResolver和rolePermissionResolver）
2. realm=com.github.zhangkaitao.shiro.chapter3.realm.MyRealm
3. securityManager.realms=$realm

设置securityManager 的realms一定要放到最后，因为在调用SecurityManager.setRealms时会将realms设置给authorizer，并为各个Realm设置permissionResolver和rolePermissionResolver。另外，不能使用IniSecurityManagerFactory创建的IniRealm，因为其初始化顺序的问题可能造成后续的初始化Permission造成影响。

**2、定义BitAndWildPermissionResolver及BitPermission**

BitPermission用于实现位移方式的权限，如规则是：

权限字符串格式：+资源字符串+权限位+实例ID；以+开头中间通过+分割；权限：0 表示所有权限；1 新增（二进制：0001）、2 修改（二进制：0010）、4 删除（二进制：0100）、8 查看（二进制：1000）；如 +user+10 表示对资源user拥有修改/查看权限。

**Java代码  [收藏代码](javascript:void())**

1. **public** **class** BitPermission **implements** Permission {
2. **private** String resourceIdentify;
3. **private** **int** permissionBit;
4. **private** String instanceId;
5. **public** BitPermission(String permissionString) {
6. String[] array = permissionString.split("\\+");
7. **if**(array.length > 1) {
8. resourceIdentify = array[1];
9. }
10. **if**(StringUtils.isEmpty(resourceIdentify)) {
11. resourceIdentify = "\*";
12. }
13. **if**(array.length > 2) {
14. permissionBit = Integer.valueOf(array[2]);
15. }
16. **if**(array.length > 3) {
17. instanceId = array[3];
18. }
19. **if**(StringUtils.isEmpty(instanceId)) {
20. instanceId = "\*";
21. }
22. }
24. @Override
25. **public** **boolean** implies(Permission p) {
26. **if**(!(p **instanceof** BitPermission)) {
27. **return** **false**;
28. }
29. BitPermission other = (BitPermission) p;
30. **if**(!("\*".equals(**this**.resourceIdentify) || **this**.resourceIdentify.equals(other.resourceIdentify))) {
31. **return** **false**;
32. }
33. **if**(!(**this**.permissionBit ==0 || (**this**.permissionBit & other.permissionBit) != 0)) {
34. **return** **false**;
35. }
36. **if**(!("\*".equals(**this**.instanceId) || **this**.instanceId.equals(other.instanceId))) {
37. **return** **false**;
38. }
39. **return** **true**;
40. }
41. }

Permission接口提供了boolean implies(Permission p)方法用于判断权限匹配的；

**Java代码  [收藏代码](javascript:void())**

1. **public** **class** BitAndWildPermissionResolver **implements** PermissionResolver {
2. @Override
3. **public** Permission resolvePermission(String permissionString) {
4. **if**(permissionString.startsWith("+")) {
5. **return** **new** BitPermission(permissionString);
6. }
7. **return** **new** WildcardPermission(permissionString);
8. }
9. }

BitAndWildPermissionResolver实现了PermissionResolver接口，并根据权限字符串是否以“+”开头来解析权限字符串为BitPermission或WildcardPermission。

**3、定义MyRolePermissionResolver**

RolePermissionResolver用于根据角色字符串来解析得到权限集合。

**Java代码  [收藏代码](javascript:void())**

1. **public** **class** MyRolePermissionResolver **implements** RolePermissionResolver {
2. @Override
3. **public** Collection<Permission> resolvePermissionsInRole(String roleString) {
4. **if**("role1".equals(roleString)) {
5. **return** Arrays.asList((Permission)**new** WildcardPermission("menu:\*"));
6. }
7. **return** **null**;
8. }
9. }

此处的实现很简单，如果用户拥有role1，那么就返回一个“menu:\*”的权限。

**4、自定义Realm**

**Java代码  [收藏代码](javascript:void())**

1. **public** **class** MyRealm **extends** AuthorizingRealm {
2. @Override
3. **protected** AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principals) {
4. SimpleAuthorizationInfo authorizationInfo = **new** SimpleAuthorizationInfo();
5. authorizationInfo.addRole("role1");
6. authorizationInfo.addRole("role2");
7. authorizationInfo.addObjectPermission(**new** BitPermission("+user1+10"));
8. authorizationInfo.addObjectPermission(**new** WildcardPermission("user1:\*"));
9. authorizationInfo.addStringPermission("+user2+10");
10. authorizationInfo.addStringPermission("user2:\*");
11. **return** authorizationInfo;
12. }
13. @Override
14. **protected** AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken token) **throws** AuthenticationException {
15. //和com.github.zhangkaitao.shiro.chapter2.realm.MyRealm1. getAuthenticationInfo代码一样，省略
16. }
17. }

此时我们继承AuthorizingRealm而不是实现Realm接口；推荐使用AuthorizingRealm，因为：

AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken token)：表示获取身份验证信息；

AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principals)：表示根据用户身份获取授权信息。

这种方式的好处是当只需要身份验证时只需要获取身份验证信息而不需要获取授权信息。对于AuthenticationInfo和AuthorizationInfo请参考其Javadoc获取相关接口信息。

另外我们可以使用JdbcRealm，需要做的操作如下：

1、执行sql/ shiro-init-data.sql 插入相关的权限数据；

2、使用shiro-jdbc-authorizer.ini配置文件，需要设置jdbcRealm.permissionsLookupEnabled

为true来开启权限查询。

此次还要注意就是不能把我们自定义的如“+user1+10”配置到INI配置文件，即使有IniRealm完成，因为IniRealm在new完成后就会解析这些权限字符串，默认使用了WildcardPermissionResolver完成，即此处是一个设计权限，如果采用生命周期（如使用初始化方法）的方式进行加载就可以解决我们自定义permissionResolver的问题。

**5、测试用例**

**Java代码  [收藏代码](javascript:void())**

1. **public** **class** AuthorizerTest **extends** BaseTest {
3. @Test
4. **public** **void** testIsPermitted() {
5. login("classpath:shiro-authorizer.ini", "zhang", "123");
6. //判断拥有权限：user:create
7. Assert.assertTrue(subject().isPermitted("user1:update"));
8. Assert.assertTrue(subject().isPermitted("user2:update"));
9. //通过二进制位的方式表示权限
10. Assert.assertTrue(subject().isPermitted("+user1+2"));//新增权限
11. Assert.assertTrue(subject().isPermitted("+user1+8"));//查看权限
12. Assert.assertTrue(subject().isPermitted("+user2+10"));//新增及查看
14. Assert.assertFalse(subject().isPermitted("+user1+4"));//没有删除权限
16. Assert.assertTrue(subject().isPermitted("menu:view"));//通过MyRolePermissionResolver解析得到的权限
17. }
18. }

通过如上步骤可以实现自定义权限验证了。另外因为不支持hasAnyRole/isPermittedAny这种方式的授权，可以参考我的一篇《简单shiro扩展实现NOT、AND、OR权限验证 》进行简单的扩展完成这个需求，在这篇文章中通过重写AuthorizingRealm里的验证逻辑实现的。

示例源代码：<https://github.com/zhangkaitao/shiro-example>；可加群134755960探讨Spring/Shiro技术。