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。

**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())**

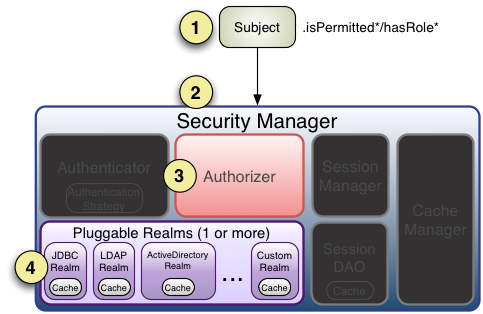
1. subject().checkPermission("menu:view:1");
2. subject().checkPermission(**new** WildcardPermission("menu:view:1"));

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）**

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

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；

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。

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**

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