1、对象名=全限定类名  相对于调用public无参构造器创建对象

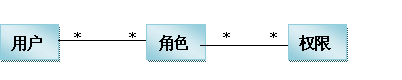
2、对象名.属性名=值    相当于调用setter方法设置常量值

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

1. [main]
2. #提供了对根对象securityManager及其依赖的配置
3. securityManager=org.apache.shiro.mgt.DefaultSecurityManager
4. …………
5. securityManager.realms=$jdbcRealm
7. [users]
8. #提供了对用户/密码及其角色的配置，用户名=密码，角色1，角色2
9. username=password,role1,role2
11. [roles]
12. #提供了角色及权限之间关系的配置，角色=权限1，权限2
13. role1=permission1,permission2
15. [urls]
16. #用于web，提供了对web url拦截相关的配置，url=拦截器[参数]，拦截器
17. /index.html = anon
18. /admin/\*\* = authc, roles[admin], perms["permission1"]

**Realm**

**1、定义实体及关系**

****

即用户-角色之间是多对多关系，角色-权限之间是多对多关系；且用户和权限之间通过角色建立关系；在系统中验证时通过权限验证，角色只是权限集合，即所谓的显示角色；其实权限应该对应到资源（如菜单、URL、页面按钮、Java方法等）中，即应该将权限字符串存储到资源实体中，但是目前为了简单化，直接提取一个权限表，

用户实体包括：编号(id)、用户名(username)、密码(password)、盐(salt)、是否锁定(locked)；是否锁定用于封禁用户使用，其实最好使用Enum字段存储，可以实现更复杂的用户状态实现。

角色实体包括：、编号(id)、角色标识符（role）、描述（description）、是否可用（available）；其中角色标识符用于在程序中进行隐式角色判断的，描述用于以后再前台界面显示的、是否可用表示角色当前是否激活。

权限(菜单)实体包括：编号（id）、权限标识符（permission）、描述（description）、是否可用（available）；含义和角色实体类似不再阐述。

另外还有两个关系实体：用户-角色实体（用户编号、角色编号，且组合为复合主键）；角色-权限实体（角色编号、权限编号，且组合为复合主键）。

**2、环境准备**

为了方便数据库操作，使用了“org.springframework: spring-jdbc: 4.0.0.RELEASE”依赖，虽然是spring4版本的，但使用上和spring3无区别。其他依赖请参考源码的pom.xml。

**3、定义Service及Dao**

为了实现的简单性，只实现必须的功能，其他的可以自己实现即可。

**PermissionService**

**Java代码  [收藏代码](javascript:void())**

1. **public** **interface** PermissionService {
2. **public** Permission createPermission(Permission permission);
3. **public** **void** deletePermission(Long permissionId);
4. }

实现基本的创建/删除权限。

**RoleService**

**Java代码  [收藏代码](javascript:void())**

1. **public** **interface** RoleService {
2. **public** Role createRole(Role role);
3. **public** **void** deleteRole(Long roleId);
4. //添加角色-权限之间关系
5. **public** **void** correlationPermissions(Long roleId, Long... permissionIds);
6. //移除角色-权限之间关系
7. **public** **void** uncorrelationPermissions(Long roleId, Long... permissionIds);//
8. }

相对于PermissionService多了关联/移除关联角色-权限功能。

**UserService**

**Java代码  [收藏代码](javascript:void())**

1. **public** **interface** UserService {
2. **public** User createUser(User user); //创建账户
3. **public** **void** changePassword(Long userId, String newPassword);//修改密码
4. **public** **void** correlationRoles(Long userId, Long... roleIds); //添加用户-角色关系
5. **public** **void** uncorrelationRoles(Long userId, Long... roleIds);// 移除用户-角色关系
6. **public** User findByUsername(String username);// 根据用户名查找用户
7. **public** Set<String> findRoles(String username);// 根据用户名查找其角色
8. **public** Set<String> findPermissions(String username); //根据用户名查找其权限
9. }

此处使用findByUsername、findRoles及findPermissions来查找用户名对应的帐号、角色及权限信息。之后的Realm就使用这些方法来查找相关信息。

**UserServiceImpl**

**Java代码  [收藏代码](javascript:void())**

1. **public** User createUser(User user) {
2. //加密密码
3. passwordHelper.encryptPassword(user);
4. **return** userDao.createUser(user);
5. }
6. **public** **void** changePassword(Long userId, String newPassword) {
7. User user =userDao.findOne(userId);
8. user.setPassword(newPassword);
9. passwordHelper.encryptPassword(user);
10. userDao.updateUser(user);
11. }

在创建账户及修改密码时直接把生成密码操作委托给PasswordHelper。

**PasswordHelper**

**Java代码  [收藏代码](javascript:void())**

1. **public** **class** PasswordHelper {
2. **private** RandomNumberGenerator randomNumberGenerator =
3. **new** SecureRandomNumberGenerator();
4. **private** String algorithmName = "md5";
5. **private** **final** **int** hashIterations = 2;
6. **public** **void** encryptPassword(User user) {
7. user.setSalt(randomNumberGenerator.nextBytes().toHex());
8. String newPassword = **new** SimpleHash(
9. algorithmName,
10. user.getPassword(),
11. ByteSource.Util.bytes(user.getCredentialsSalt()),
12. hashIterations).toHex();
13. user.setPassword(newPassword);
14. }
15. }

之后的CredentialsMatcher需要和此处加密的算法一样。user.getCredentialsSalt()辅助方法返回username+salt。

为了节省篇幅，对于DAO/Service的接口及实现，具体请参考源码com.github.zhangkaitao.shiro.chapter6。另外请参考Service层的测试用例com.github.zhangkaitao.shiro.chapter6.service.ServiceTest。

**4、定义Realm**

**RetryLimitHashedCredentialsMatcher**

和第五章的一样，在此就不罗列代码了，请参考源码com.github.zhangkaitao.shiro.chapter6.credentials.RetryLimitHashedCredentialsMatcher。

**UserRealm**

另外请参考Service层的测试用例com.github.zhangkaitao.shiro.chapter6.service.ServiceTest。

**Java代码  [收藏代码](javascript:void())**

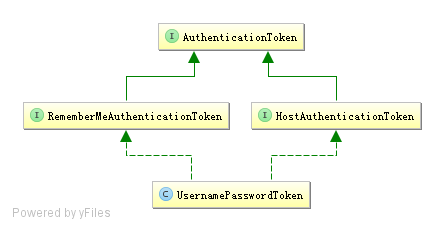
1. **public** **class** UserRealm **extends** AuthorizingRealm {
2. **private** UserService userService = **new** UserServiceImpl();
3. **protected** AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principals) {
4. String username = (String)principals.getPrimaryPrincipal();
5. SimpleAuthorizationInfo authorizationInfo = **new** SimpleAuthorizationInfo();
6. authorizationInfo.setRoles(userService.findRoles(username));
7. authorizationInfo.setStringPermissions(userService.findPermissions(username));
8. **return** authorizationInfo;
9. }
10. **protected** AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken token) **throws** AuthenticationException {
11. String username = (String)token.getPrincipal();
12. User user = userService.findByUsername(username);
13. **if**(user == **null**) {
14. **throw** **new** UnknownAccountException();//没找到帐号
15. }
16. **if**(Boolean.TRUE.equals(user.getLocked())) {
17. **throw** **new** LockedAccountException(); //帐号锁定
18. }
19. //交给AuthenticatingRealm使用CredentialsMatcher进行密码匹配，如果觉得人家的不好可以在此判断或自定义实现
20. SimpleAuthenticationInfo authenticationInfo = **new** SimpleAuthenticationInfo(
21. user.getUsername(), //用户名
22. user.getPassword(), //密码
23. ByteSource.Util.bytes(user.getCredentialsSalt()),//salt=username+salt
24. getName()  //realm name
25. );
26. **return** authenticationInfo;
27. }
28. }

**1、UserRealm父类AuthorizingRealm将获取Subject相关信息分成两步**：获取身份验证信息（doGetAuthenticationInfo）及授权信息（doGetAuthorizationInfo）；

**2、doGetAuthenticationInfo获取身份验证相关信息**：首先根据传入的用户名获取User信息；然后如果user为空，那么抛出没找到帐号异常UnknownAccountException；如果user找到但锁定了抛出锁定异常LockedAccountException；最后生成AuthenticationInfo信息，交给间接父类AuthenticatingRealm使用CredentialsMatcher进行判断密码是否匹配，如果不匹配将抛出密码错误异常IncorrectCredentialsException；另外如果密码重试此处太多将抛出超出重试次数异常ExcessiveAttemptsException；在组装SimpleAuthenticationInfo信息时，需要传入：身份信息（用户名）、凭据（密文密码）、盐（username+salt），CredentialsMatcher使用盐加密传入的明文密码和此处的密文密码进行匹配。

**3、doGetAuthorizationInfo获取授权信息**：PrincipalCollection是一个身份集合，因为我们现在就一个Realm，所以直接调用getPrimaryPrincipal得到之前传入的用户名即可；然后根据用户名调用UserService接口获取角色及权限信息。

**6.2 AuthenticationToken**



AuthenticationToken用于收集用户提交的身份（如用户名）及凭据（如密码）：

**Java代码  [收藏代码](javascript:void())**

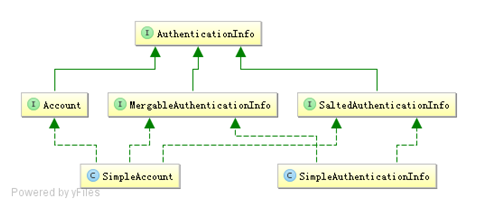
1. **public** **interface** AuthenticationToken **extends** Serializable {
2. Object getPrincipal(); //身份
3. Object getCredentials(); //凭据
4. }

扩展接口RememberMeAuthenticationToken：提供了“boolean isRememberMe()”现“记住我”的功能；

扩展接口是HostAuthenticationToken：提供了“String getHost()”方法用于获取用户“主机”的功能。

Shiro提供了一个直接拿来用的UsernamePasswordToken，用于实现用户名/密码Token组，另外其实现了RememberMeAuthenticationToken和HostAuthenticationToken，可以实现记住我及主机验证的支持。

**6.3 AuthenticationInfo**



AuthenticationInfo有两个作用：

1、如果Realm是AuthenticatingRealm子类，则提供给AuthenticatingRealm内部使用的CredentialsMatcher进行凭据验证；（如果没有继承它需要在自己的Realm中自己实现验证）；

2、提供给SecurityManager来创建Subject（提供身份信息）；

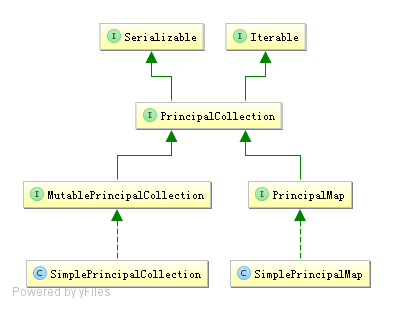
MergableAuthenticationInfo用于提供在多Realm时合并AuthenticationInfo的功能，主要合并Principal、如果是其他的如credentialsSalt，会用后边的信息覆盖前边的。

比如HashedCredentialsMatcher，在验证时会判断AuthenticationInfo是否是SaltedAuthenticationInfo子类，来获取盐信息。

Account相当于我们之前的User，SimpleAccount是其一个实现；在IniRealm、PropertiesRealm这种静态创建帐号信息的场景中使用，这些Realm直接继承了SimpleAccountRealm，而SimpleAccountRealm提供了相关的API来动态维护SimpleAccount；即可以通过这些API来动态增删改查SimpleAccount；动态增删改查角色/权限信息。及如果您的帐号不是特别多，可以使用这种方式，具体请参考SimpleAccountRealm Javadoc。

其他情况一般返回SimpleAuthenticationInfo即可。

**6.4 PrincipalCollection**



因为我们可以在Shiro中同时配置多个Realm，所以呢身份信息可能就有多个；因此其提供了PrincipalCollection用于聚合这些身份信息：

**Java代码  [收藏代码](javascript:void())**

1. **public** **interface** PrincipalCollection **extends** Iterable, Serializable {
2. Object getPrimaryPrincipal(); //得到主要的身份
3. <T> T oneByType(Class<T> type); //根据身份类型获取第一个
4. <T> Collection<T> byType(Class<T> type); //根据身份类型获取一组
5. List asList(); //转换为List
6. Set asSet(); //转换为Set
7. Collection fromRealm(String realmName); //根据Realm名字获取
8. Set<String> getRealmNames(); //获取所有身份验证通过的Realm名字
9. **boolean** isEmpty(); //判断是否为空
10. }

因为PrincipalCollection聚合了多个，此处最需要注意的是getPrimaryPrincipal，如果只有一个Principal那么直接返回即可，如果有多个Principal，则返回第一个（因为内部使用Map存储，所以可以认为是返回任意一个）；oneByType / byType根据凭据的类型返回相应的Principal；fromRealm根据Realm名字（每个Principal都与一个Realm关联）获取相应的Principal。

MutablePrincipalCollection是一个可变的PrincipalCollection接口，即提供了如下可变方法：

**Java代码  [收藏代码](javascript:void())**

1. **public** **interface** MutablePrincipalCollection **extends** PrincipalCollection {
2. **void** add(Object principal, String realmName); //添加Realm-Principal的关联
3. **void** addAll(Collection principals, String realmName); //添加一组Realm-Principal的关联
4. **void** addAll(PrincipalCollection principals);//添加PrincipalCollection
5. **void** clear();//清空
6. }

目前Shiro只提供了一个实现SimplePrincipalCollection，还记得之前的AuthenticationStrategy实现嘛，用于在多Realm时判断是否满足条件的，在大多数实现中（继承了AbstractAuthenticationStrategy）afterAttempt方法会进行AuthenticationInfo（实现了MergableAuthenticationInfo）的merge，比如SimpleAuthenticationInfo会合并多个Principal为一个PrincipalCollection。

对于PrincipalMap是Shiro 1.2中的一个实验品，暂时无用，具体可以参考其Javadoc。接下来通过示例来看看PrincipalCollection。

**1、准备三个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** "a"; //realm name 为 “a”
5. }
6. //省略supports方法，具体请见源码
7. @Override
8. **public** AuthenticationInfo getAuthenticationInfo(AuthenticationToken token) **throws** AuthenticationException {
9. **return** **new** SimpleAuthenticationInfo(
10. "zhang", //身份 字符串类型
11. "123",   //凭据
12. getName() //Realm Name
13. );
14. }
15. }

**MyRealm2**

和MyRealm1完全一样，只是Realm名字为b。

**MyRealm3**

**Java代码  [收藏代码](javascript:void())**

1. **public** **class** MyRealm3 **implements** Realm {
2. @Override
3. **public** String getName() {
4. **return** "c"; //realm name 为 “c”
5. }
6. //省略supports方法，具体请见源码
7. @Override
8. **public** AuthenticationInfo getAuthenticationInfo(AuthenticationToken token) **throws** AuthenticationException {
9. User user = **new** User("zhang", "123");
10. **return** **new** SimpleAuthenticationInfo(
11. user, //身份 User类型
12. "123",   //凭据
13. getName() //Realm Name
14. );
15. }
16. }

和MyRealm1同名，但返回的Principal是User类型。

**2、ini配置（shiro-multirealm.ini）**

**Java代码  [收藏代码](javascript:void())**

1. [main]
2. realm1=com.github.zhangkaitao.shiro.chapter6.realm.MyRealm1
3. realm2=com.github.zhangkaitao.shiro.chapter6.realm.MyRealm2
4. realm3=com.github.zhangkaitao.shiro.chapter6.realm.MyRealm3
5. securityManager.realms=$realm1,$realm2,$realm3

**3、测试用例（com.github.zhangkaitao.shiro.chapter6.realm.PrincialCollectionTest）**

因为我们的Realm中没有进行身份及凭据验证，所以相当于身份验证都是成功的，都将返回：

**Java代码  [收藏代码](javascript:void())**

1. Object primaryPrincipal1 = subject.getPrincipal();
2. PrincipalCollection princialCollection = subject.getPrincipals();
3. Object primaryPrincipal2 = princialCollection.getPrimaryPrincipal();

我们可以直接调用subject.getPrincipal获取PrimaryPrincipal（即所谓的第一个）；或者通过getPrincipals获取PrincipalCollection；然后通过其getPrimaryPrincipal获取PrimaryPrincipal。

**Java代码  [收藏代码](javascript:void())**

1. Set<String> realmNames = princialCollection.getRealmNames();

获取所有身份验证成功的Realm名字。

**Java代码  [收藏代码](javascript:void())**

1. Set<Object> principals = princialCollection.asSet(); //asList和asSet的结果一样

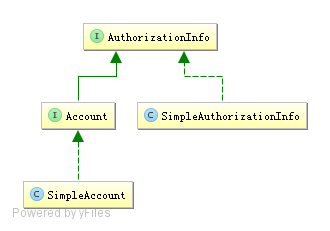
将身份信息转换为Set/List，即使转换为List，也是先转换为Set再完成的。

**Java代码  [收藏代码](javascript:void())**

1. Collection<User> users = princialCollection.fromRealm("c");

根据Realm名字获取身份，因为Realm名字可以重复，所以可能多个身份，建议Realm名字尽量不要重复。

**6.4 AuthorizationInfo**



AuthorizationInfo用于聚合授权信息的：

**Java代码  [收藏代码](javascript:void())**

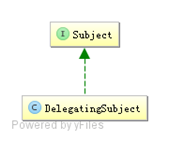
1. **public** **interface** AuthorizationInfo **extends** Serializable {
2. Collection<String> getRoles(); //获取角色字符串信息
3. Collection<String> getStringPermissions(); //获取权限字符串信息
4. Collection<Permission> getObjectPermissions(); //获取Permission对象信息
5. }

当我们使用AuthorizingRealm时，如果身份验证成功，在进行授权时就通过doGetAuthorizationInfo方法获取角色/权限信息用于授权验证。

Shiro提供了一个实现SimpleAuthorizationInfo，大多数时候使用这个即可。

对于Account及SimpleAccount，之前的【6.3 AuthenticationInfo】已经介绍过了，用于SimpleAccountRealm子类，实现动态角色/权限维护的。

**6.5 Subject**



Subject是Shiro的核心对象，基本所有身份验证、授权都是通过Subject完成。

**1、身份信息获取**

**Java代码  [收藏代码](javascript:void())**

1. Object getPrincipal(); //Primary Principal
2. PrincipalCollection getPrincipals(); // PrincipalCollection

**2、身份验证**

**Java代码  [收藏代码](javascript:void())**

1. **void** login(AuthenticationToken token) **throws** AuthenticationException;
2. **boolean** isAuthenticated();
3. **boolean** isRemembered();

通过login登录，如果登录失败将抛出相应的AuthenticationException，如果登录成功调用isAuthenticated就会返回true，即已经通过身份验证；如果isRemembered返回true，表示是通过记住我功能登录的而不是调用login方法登录的。isAuthenticated/isRemembered是互斥的，即如果其中一个返回true，另一个返回false。

**3、角色授权验证**

1. **boolean** hasRole(String roleIdentifier);
2. **boolean**[] hasRoles(List<String> roleIdentifiers);
3. **boolean** hasAllRoles(Collection<String> roleIdentifiers);
4. **void** checkRole(String roleIdentifier) **throws** AuthorizationException;
5. **void** checkRoles(Collection<String> roleIdentifiers) **throws** AuthorizationException;
6. **void** checkRoles(String... roleIdentifiers) **throws** AuthorizationException;

hasRole\*进行角色验证，验证后返回true/false；而checkRole\*验证失败时抛出AuthorizationException异常。

**4、权限授权验证**

1. **boolean** isPermitted(String permission);
2. **boolean** isPermitted(Permission permission);
3. **boolean**[] isPermitted(String... permissions);
4. **boolean**[] isPermitted(List<Permission> permissions);
5. **boolean** isPermittedAll(String... permissions);
6. **boolean** isPermittedAll(Collection<Permission> permissions);
7. **void** checkPermission(String permission) **throws** AuthorizationException;
8. **void** checkPermission(Permission permission) **throws** AuthorizationException;
9. **void** checkPermissions(String... permissions) **throws** AuthorizationException;
10. **void** checkPermissions(Collection<Permission> permissions) **throws** AuthorizationException;

isPermitted\*进行权限验证，验证后返回true/false；而checkPermission\*验证失败时抛出AuthorizationException。

**5、会话**

1. Session getSession(); //相当于getSession(true)
2. Session getSession(**boolean** create);

类似于Web中的会话。如果登录成功就相当于建立了会话，接着可以使用getSession获取；如果create=false如果没有会话将返回null，而create=true如果没有会话会强制创建一个。

**6、退出**

1. **void** logout();

**7、RunAs**

1. **void** runAs(PrincipalCollection principals) **throws** NullPointerException, IllegalStateException;
2. **boolean** isRunAs();
3. PrincipalCollection getPreviousPrincipals();
4. PrincipalCollection releaseRunAs();

RunAs即实现“允许A假设为B身份进行访问”；通过调用subject.runAs(b)进行访问；接着调用subject.getPrincipals将获取到B的身份；此时调用isRunAs将返回true；而a的身份需要通过subject. getPreviousPrincipals获取；如果不需要RunAs了调用subject. releaseRunAs即可。

**8、多线程**

1. <V> V execute(Callable<V> callable) **throws** ExecutionException;
2. **void** execute(Runnable runnable);
3. <V> Callable<V> associateWith(Callable<V> callable);
4. Runnable associateWith(Runnable runnable);

实现线程之间的Subject传播，因为Subject是线程绑定的；因此在多线程执行中需要传播到相应的线程才能获取到相应的Subject。最简单的办法就是通过execute(runnable/callable实例)直接调用；或者通过associateWith(runnable/callable实例)得到一个包装后的实例；它们都是通过：1、把当前线程的Subject绑定过去；2、在线程执行结束后自动释放。

Subject自己不会实现相应的身份验证/授权逻辑，而是通过DelegatingSubject委托给SecurityManager实现；及可以理解为Subject是一个面门。

对于Subject的构建一般没必要我们去创建；一般通过SecurityUtils.getSubject()获取：

1. **public** **static** Subject getSubject() {
2. Subject subject = ThreadContext.getSubject();
3. **if** (subject == **null**) {
4. subject = (**new** Subject.Builder()).buildSubject();
5. ThreadContext.bind(subject);
6. }
7. **return** subject;
8. }

即首先查看当前线程是否绑定了Subject，如果没有通过Subject.Builder构建一个然后绑定到现场返回。

如果想自定义创建，可以通过：

1. **new** Subject.Builder().principals(身份).authenticated(**true**/**false**).buildSubject()

这种可以创建相应的Subject实例了，然后自己绑定到线程即可。在new Builder()时如果没有传入SecurityManager，自动调用SecurityUtils.getSecurityManager获取；也可以自己传入一个实例。

对于Subject我们一般这么使用：

**1、身份验证（login）**

**2、授权（hasRole\*/isPermitted\*或checkRole\*/checkPermission\*）**

**3、将相应的数据存储到会话（Session）**

**4、切换身份（RunAs）/多线程身份传播**

**5、退出**