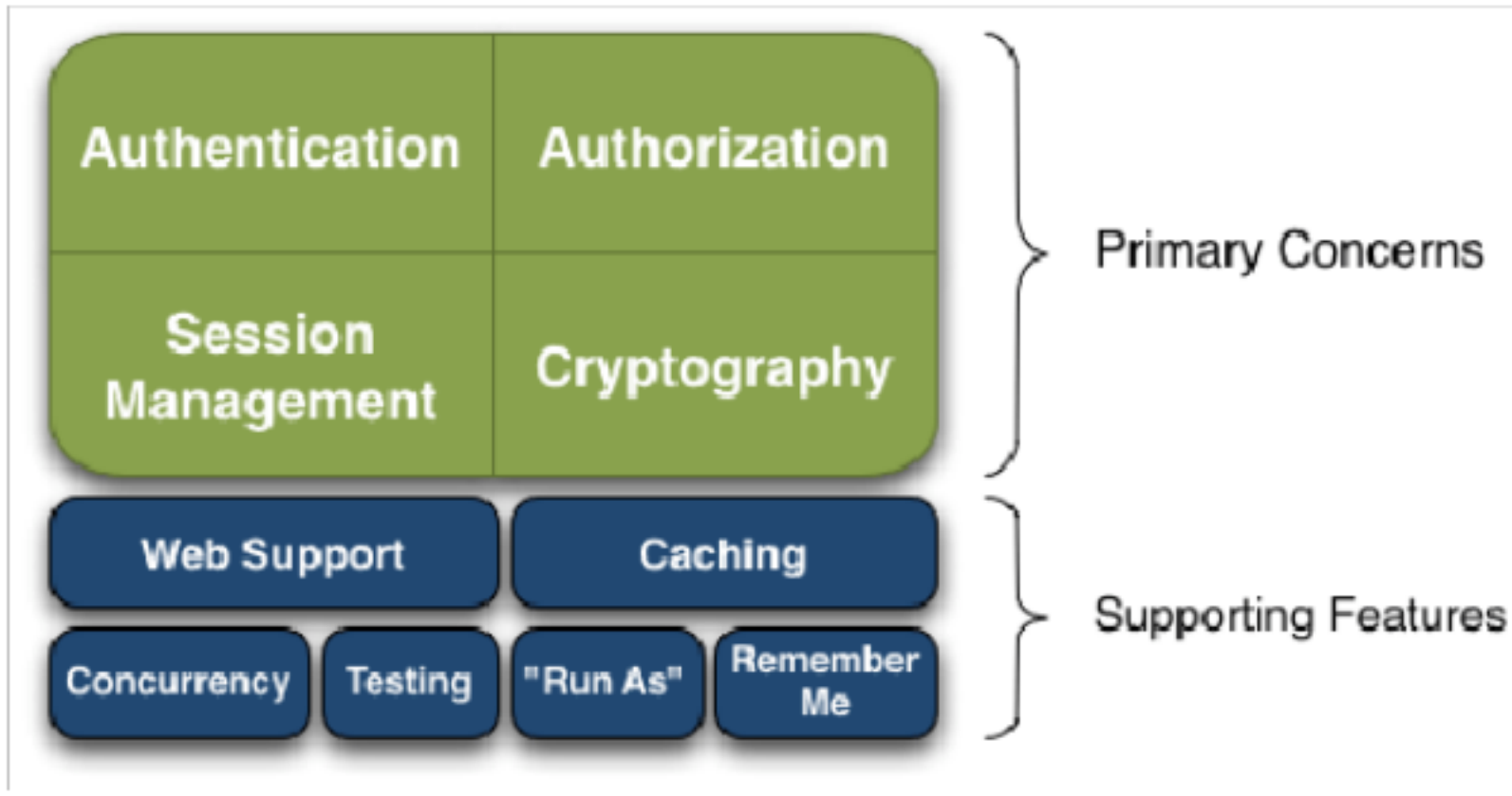
Shiro 学习

Shiro 简介

Apache Shiro 是 java 的安全框架。 Shiro 可以非常容易的开发出足够好的应用，可以帮我们完成：认证、授权、加密、会话管理、与 Web 集成、缓存等。



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。

2. 身份验证

2.2 环境准备

```
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

```
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。

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

Java 代码

```
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.get
   Instance();
8.     SecurityUtils.setSecurityManager(securityManager);
```

```
        //3、得到 Subject 及创建用户名/密码身份验证 Token（即用户身份/凭证）
    Subject subject = SecurityUtils.getSubject();
    UsernamePasswordToken token = new UsernamePasswordToken("zhang", "123");

        try {
            //4、登录，即身份验证
        subject.login(token);
    } catch (AuthenticationException e) {
            //5、身份验证失败
    }

    Assert.assertEquals(true, subject.isAuthenticated());  // 断言用户已经登录

        //6、退出
    subject.logout();
}

```

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。

3.授权

3.1Shiro 的三种方式授权

编程式：通过写 if/else 授权代码块完成

```
Subject subject=SecuriryUtils.getSubject();
If(subject.hasRole(“admin”)){
    //有权限
}else{
    //无权限
}
```

注解式：通过在执行的 java 方法上放置相应的注解完成

```
@RequiresRoles(“admin”)
Public void hello(){
    //有权限
}
```

JSP/GSP 标签：在 JSP/GSP 页面通过相应的标签完成

```
<shiroLhasRole name=”admin”>
<!-- 有权限 —>
</shiro:hasRole>
```

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

```
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

```
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

```
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。

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

```
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

```
role41=system:user:update,system:user:delete
```

然后通过如下代码判断

Java 代码

```
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

```
role42= "system:user:update,delete"
```

接着可以通过如下代码判断

Java 代码

```
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

```
role51= "system:user:create,update,delete,view"
```

然后通过如下代码判断

Java 代码

```
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

```
role52=system:user:*
```

也可以简写为（推荐上边的写法）：

Java 代码

```
role53=system:user
```

然后通过如下代码判断

Java 代码

```
subject().checkPermissions("system:user:*");
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

```
role61=*:view
```

然后通过如下代码判断

Java 代码

```
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

```
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

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

5.2、单个实例多个权限

ini 配置

Java 代码

```
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

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

5.3、单个实例所有权限

ini 配置

Java 代码

```
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

```
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

```
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

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

5.5、所有实例所有权限

ini 配置

Java 代码

```
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

```
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

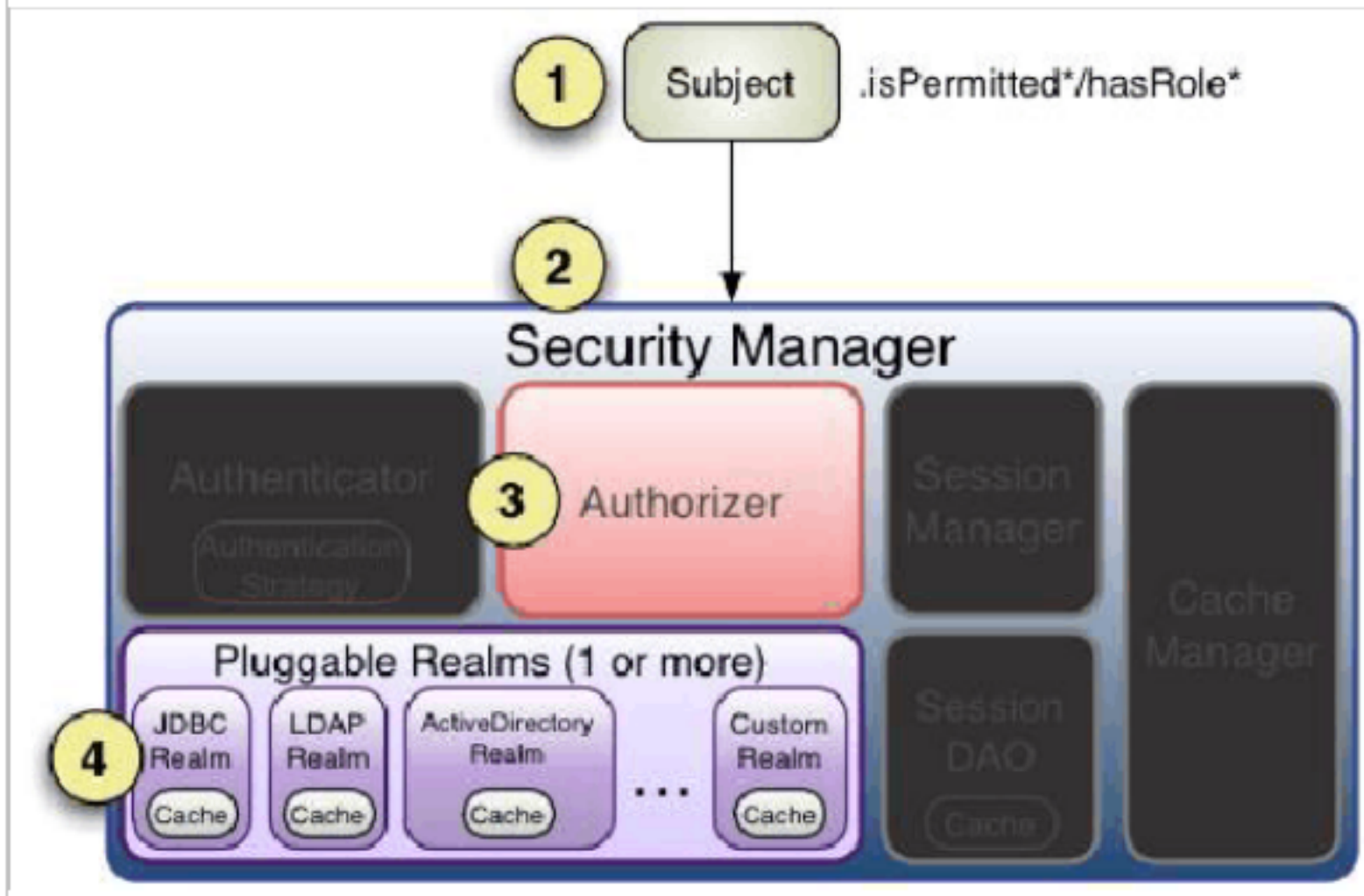
```
subject().checkPermission("menu:view:1");
subject().checkPermission(new WildcardPermission("menu:view:1"));
```

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

```
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

```
1. permissionResolver=org.apache.shiro.authz.permission.WildcardPermission
   Resolver
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

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

示例

1、ini 配置（shiro-authorizer.ini）

Java 代码

```
[main]
#自定义 authorizer
authorizer=org.apache.shiro.authz.ModularRealmAuthorizer
#自定义 permissionResolver
#permissionResolver=org.apache.shiro.authz.permission.WildcardPermissionResolver
permissionResolver=com.github.zhangkaitao.shiro.chapter3.permission.BitAndWildPermissionResolver
authorizer.permissionResolver=$permissionResolver
#自定义 rolePermissionResolver
rolePermissionResolver=com.github.zhangkaitao.shiro.chapter3.permission.MyRolePermissionResolver
authorizer.rolePermissionResolver=$rolePermissionResolver

securityManager.authorizer=$authorizer
```

Java 代码

```
#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）
realm=com.github.zhangkaitao.shiro.chapter3.realm.MyRealm
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

```
public class BitPermission implements Permission {
```

```
    private String resourceIdentify;
    private int permissionBit;
    private String instanceId;
    public BitPermission(String permissionString) {
        String[] array = permissionString.split("\\+");
        if (array.length > 1) {
            resourceIdentify = array[1];
        }
        if (StringUtils.isEmpty(resourceIdentify)) {
            resourceIdentify = "*";
        }
        if (array.length > 2) {
            permissionBit = Integer.valueOf(array[2]);
        }
        if (array.length > 3) {
            instanceId = array[3];
        }
        if (StringUtils.isEmpty(instanceId)) {
            instanceId = "*";
        }
    }

    @Override
    public boolean implies(Permission p) {
        if (!(p instanceof BitPermission)) {
            return false;
        }
        BitPermission other = (BitPermission) p;
        if (!("*".equals(this.resourceIdentify) || this.resourceIdentify.equals(other.resourceIdentify))) {
            return false;
        }
        if (!(this.permissionBit == 0 || (this.permissionBit & other.permissionBit) != 0)) {
            return false;
        }
        if (!("*".equals(this.instanceId) || this.instanceId.equals(other.instanceId))) {
            return false;
        }
        return true;
    }
}
```

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

Java 代码

```
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

```
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

```
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

```
1.  public class AuthorizerTest extends BaseTest {
2.
```

```
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
13.
14.     Assert.assertFalse(subject().isPermitted("+user1+4"));  // 没有删除权限
15.
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。

4. INI 配置

4.1 根对象 SecurityManager

从之前的 Shiro 架构图可以看出，Shiro 是从根对象 SecurityManager 进行身份验证和授权的；也就是所有操作都是自它开始的，这个对象是线程安全且真个应用只需要一个即可，因此 Shiro 提供了 SecurityUtils 让我们绑定它为全局的，方便后续操作。

因为 Shiro 的类都是 POJO 的，因此都很容易放到任何 IoC 容器管理。但是和一般的 IoC 容器的区别在于，Shiro 从根对象 securityManager 开始导航；Shiro 支持的依赖注入：public 空参构造器对象的创建、setter 依赖注入。

1、纯 Java 代码写法（com.github.zhangkaitao.shiro.chapter4.NonConfigurationCreateTest）：

Java 代码

```
DefaultSecurityManager securityManager = new DefaultSecurityManager();
// 设置 authenticator
ModularRealmAuthenticator authenticator = new ModularRealmAuthenticator();
authenticator.setAuthenticationStrategy(new AtLeastOneSuccessfulStrategy());
securityManager.setAuthenticator(authenticator);

// 设置 authorizer
ModularRealmAuthorizer authorizer = new ModularRealmAuthorizer();
authorizer.setPermissionResolver(new WildcardPermissionResolver());
securityManager.setAuthorizer(authorizer);

// 设置 Realm
DruidDataSource ds = new DruidDataSource();
ds.setDriverClassName("com.mysql.jdbc.Driver");
ds.setUrl("jdbc:mysql://localhost:3306/shiro");
ds.setUsername("root");
ds.setPassword("");

JdbcRealm jdbcRealm = new JdbcRealm();
jdbcRealm.setDataSource(ds);
jdbcRealm.setPermissionsLookupEnabled(true);
securityManager.setRealms(Arrays.asList((Realm) jdbcRealm));

// 将 SecurityManager 设置到 SecurityUtils 方便全局使用
SecurityUtils.setSecurityManager(securityManager);

Subject subject = SecurityUtils.getSubject();
UsernamePasswordToken token = new UsernamePasswordToken("zhang", "123");
subject.login(token);
Assert.assertTrue(subject.isAuthenticated());


```

2.1、等价的 INI 配置（shiro-config.ini）

Java 代码

```
[main]
```

```
#authenticator
authenticator=org.apache.shiro.authc.pam.ModularRealmAuthenticator
authenticationStrategy=org.apache.shiro.authc.pam.AtLeastOneSuccessfulS
trategy
authenticator.authenticationStrategy=$authenticationStrategy
securityManager.authenticator=$authenticator

#authorizer
authorizer=org.apache.shiro.authz.ModularRealmAuthorizer
permissionResolver=org.apache.shiro.authz.permission.WildcardPermission
Resolver
authorizer.permissionResolver=$permissionResolver
securityManager.authorizer=$authorizer

#realm
dataSource=com.alibaba.druid.pool.DruidDataSource
dataSource.driverClassName=com.mysql.jdbc.Driver
dataSource.url=jdbc:mysql:                    //localhost:3306/shiro
dataSource.username=root
#dataSource.password=
jdbcRealm=org.apache.shiro.realm.jdbc.JdbcRealm
jdbcRealm.dataSource=$dataSource
jdbcRealm.permissionsLookupEnabled=                    true
securityManager.realms=$jdbcRealm
```

即使没接触过 IoC 容器的知识，如上配置也是很容易理解的：

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

2.2、 Java 代码（ com.github.zhangkaitao.shiro.chapter4.ConfigurationCreateT est）

Java 代码

```
Factory<org.apache.shiro.mgt.SecurityManager> factory =
                    new IniSecurityManagerFactory(                "classpath:shiro-config.ini"                    );

org.apache.shiro.mgt.SecurityManager securityManager = factory.getInsta
nce();

// 将 SecurityManager 设置到 SecurityUtils 方便全局使用
SecurityUtils.setSecurityManager(securityManager);
Subject subject = SecurityUtils.getSubject();
UsernamePasswordToken token =              new UsernamePasswordToken(        "zhang"    ,    "123
" );
```

```
subject.login(token);

Assert.assertTrue(subject.isAuthenticated());
```

如上代码是从 Shiro INI 配置中获取相应的 securityManager 实例：

1、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，它是起始的根；

2、IniSecurityManagerFactory 是创建 securityManager 的工厂，其需要一个 ini 配置文件路径，其支持“classpath:（类路径）、“file:（文件系统）、“url:（网络）三种路径格式，默认是文件系统；

3、接着获取 SecuriyManager 实例，后续步骤和之前的一样。

从如上可以看出 Shiro INI 配置方式本身提供了一个简单的 IoC/DI 机制方便在配置文件配置，但是是从 securityManager 这个根对象开始导航。

4.2 INI 配置

ini 配置文件类似于 Java 中的 properties（key=value），不过提供了将 key/value 分类的特性，key 是每个部分不重复即可，而不是整个配置文件。如下是 INI 配置分类：

Java 代码

```
[main]
#提供了对根对象 securityManager 及其依赖的配置
securityManager=org.apache.shiro.mgt.DefaultSecurityManager
……
securityManager.realms=$jdbcRealm

[users]
#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
username=password,role1,role2

[roles]
#提供了角色及权限之间关系的配置，角色=权限1，权限2
role1=permission1,permission2

[urls]
#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
/index.html = anon
/admin/** = authc, roles[admin], perms["permission1"]
```

[main] 部分

提供了对根对象 securityManager 及其依赖对象的配置。

创建对象

Java 代码

```
1. 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

Java 代码

```
1. dataSource.driverClassName=com.mysql.jdbc.Driver
2. 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

Java 代码

```
1. authenticator=org.apache.shiro.authc.pam.ModularRealmAuthenticator
2. authenticationStrategy=org.apache.shiro.authc.pam.AtLeastOneSuccessfulS
   trategy
3. authenticator.authenticationStrategy=$authenticationStrategy
4. securityManager.authenticator=$authenticator
```

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

嵌套属性 setter 注入

Java 代码

```
1. securityManager.authenticator.authenticationStrategy=$authenticationStr
   ategy
```

也支持这种嵌套方式的 setter 注入。

byte 数组 setter 注入

Java 代码

```
1. #base64 byte []
2. authenticator.bytes=aGVsbG8=
3. #hex byte []
4. 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

Java 代码

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

多个之间通过“,”分割。

Map setter 注入

Java 代码

```
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

Java 代码

```
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：

Java 代码

```
[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：

Java 代码

```
[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：

Java 代码

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

具体规则参见 web 相关章节。