Shiro

一、权限概述

- 权限管理

> 权限管理说白了就是，约束系统资源能被谁访问，不能被谁访问。涉及到用户参数的系统都需要进行权限的约束和管理。
>
> 权限管理实现的目的就是对用户访问资源的控制，用户只能访问已经被授权的资源。

> 权限管理分为**身份认证**和**授权**两个方面，对于绝大多数资源需要用户经过了身份认证，而对于需要权限控制的资源，则需要用户被授予指定权限，简称授权。

- 身份认证

> **判断一个用户是否为合法用户的处理过程**，简单地讲就是**登录**，需要确定有账号且账号正常，才能进入系统，这就是身份认证。
>
> 当然登录并不是唯一的认证方式，也有很多其他的认证方式，例如：指纹认证、刷卡、人脸等。

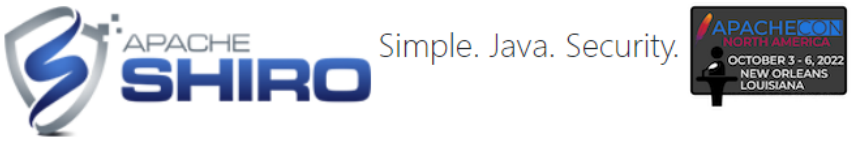
- 授权

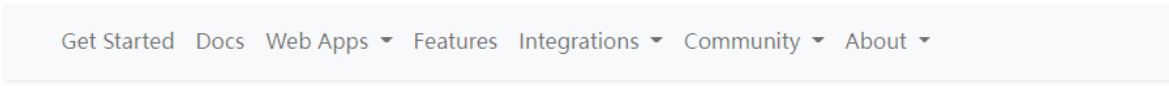
> 授权就是**给予已经认证的用户**访问某些资源的权限的过程，用户必须认证后才能进行授权，用户只能访问被授访问权限的资源，没有被授予权限的资源是无法访问的。

二、Shiro简介

（一）什么是Shiro?

> Apache Shiro是一个功能强大且易于使用的 Java 安全框架，用于执行**身份验证、授权、加密**和**会话管理**。使用Shiro的易于理解的API，您可以快速、轻松地保护任何应用程序（从最小的移动应用程序到最大的Web和企业应用程序）。
>
> 官网：http://shiro.apache.org/



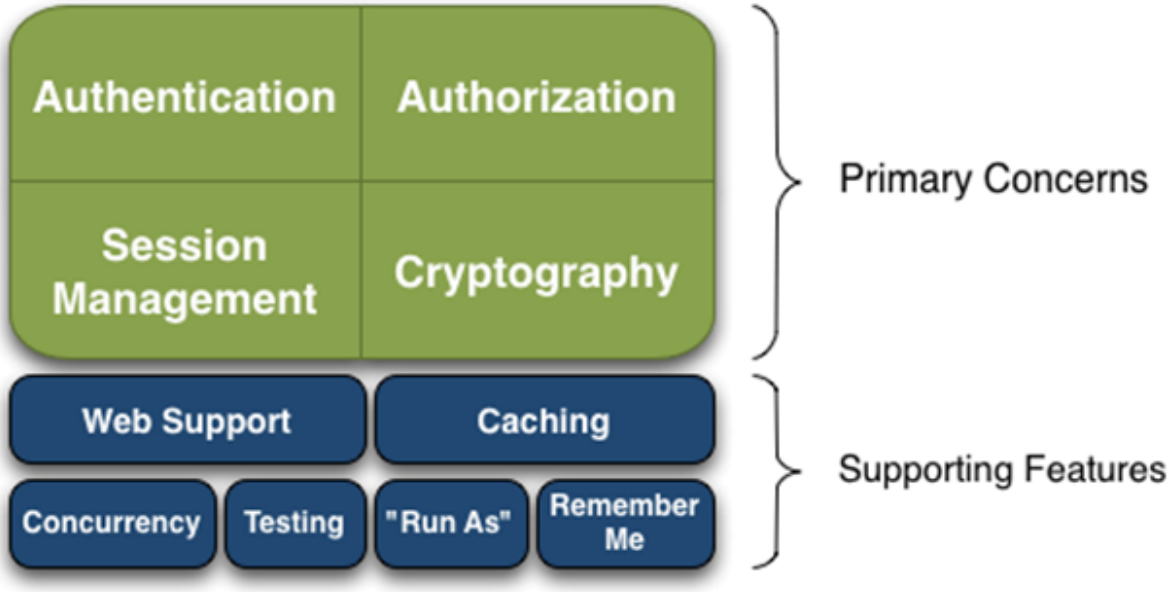


Apache Shiro™ is a powerful and easy-to-use Java security framework that performs authentication, authorization, cryptography, and session management. With Shiro's easy-to-understand API, you can quickly and easily secure any application – from the smallest mobile applications to the largest web and enterprise applications.

> **下面是您可以使用Apache Shiro做的一些事情：**
>
> - 验证用户身份以验证其身份
> - 为用户执行访问控制，例如：
>   - 确定用户是否被分配了特定的安全角色。
>   - 确定用户是否被允许做某事
> - 在任何环境中使用会话API，即使没有Web或EJB容器。
> - 在身份验证、访问控制或会话生存期期间对事件作出反应。
> - 聚合用户安全数据的1个或多个数据源，并将其表示为单个复合用户的“视图”。
> - 启用单点登录(SSO)功能
> - 为用户关联启用“记住我”服务，无需登录

（二）Shiro特性

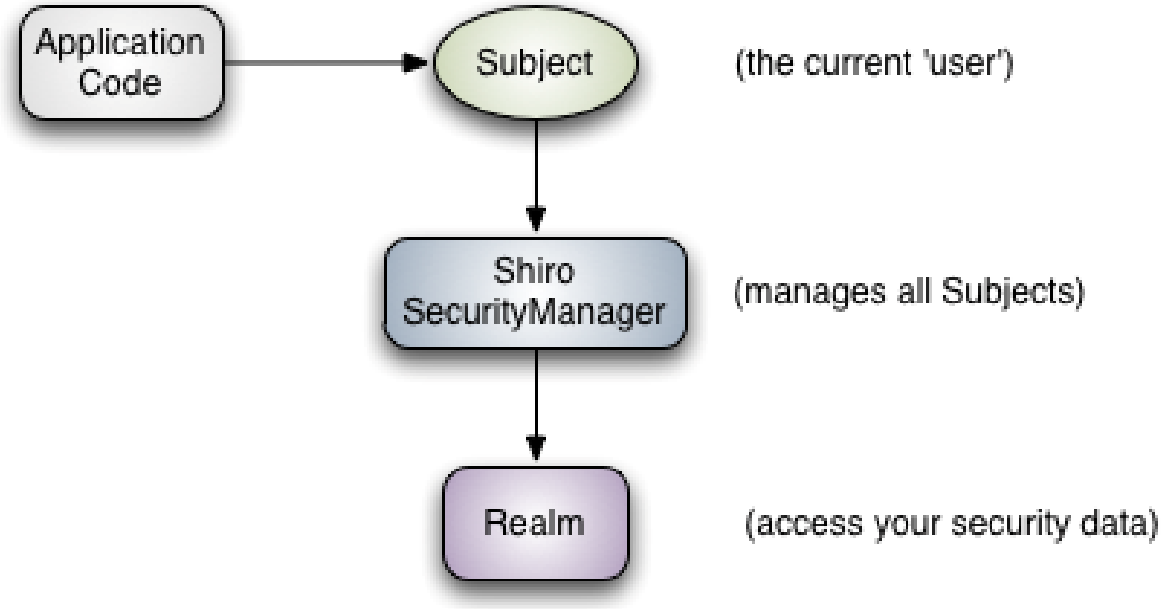
> **认证（Authentication）：**有时被称为“登录”，这是证明用户是谁的行为。
>
> **授权（Authorization）：**访问控制的过程，即确定“谁”有权访问“什么”。
>
> **会话管理（Session Management）：**管理特定于用户的会话，即使在非web或ejb应用程序中也是如此。
>
> **密码学（Cryptography）：**使用加密算法保持数据安全，但仍然易于使用。
>
> **Web支持（Web Support）：**Shiro的Web支持API帮助轻松地保护Web应用程序。
>
> **缓存（Caching）：**缓存是ApacheShiro的API中的第一层公民，以确保安全操作保持快速和高效。
>
> **并发性（Concurrency）：**ApacheShiro支持具有并发特性的多线程应用程序。
>
> **测试（Testing）：**存在测试支持，以帮助您编写单元测试和集成测试，并确保您的代码按照预期的安全。
>
> **“Run as”：**允许用户假定另一个用户的身份(如果允许的话)的特性，有时在管理场景中很有用。
>
> **“记住我（Remember Me）”：**记住用户在会话中的身份，这样他们就只需要在强制的情况下登录。



（三）Shiro结构

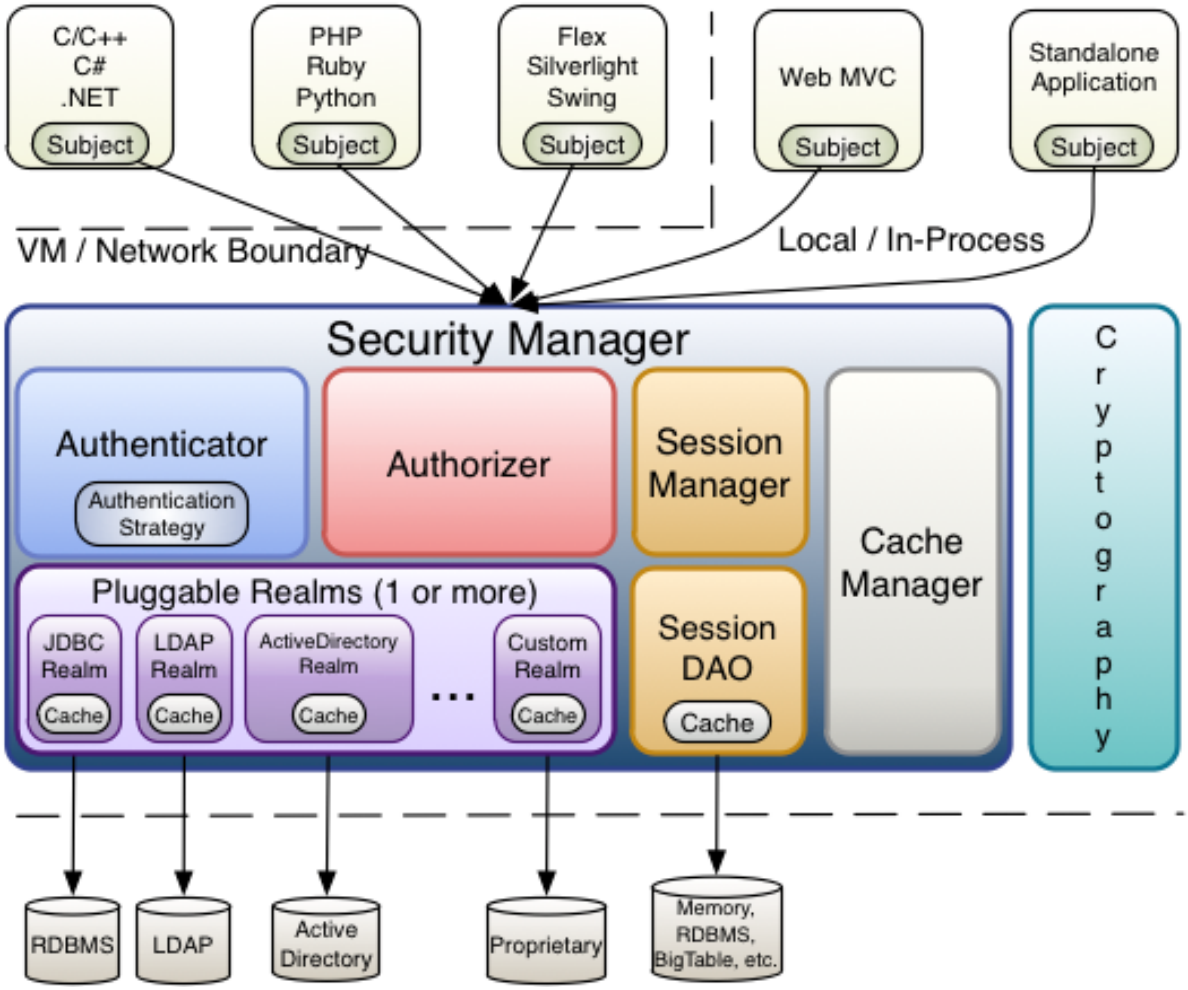
1、Shiro外部结构

> 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；



2、Shiro内部结构

> 上层：不同的语言，例如c/c++/.net/php等，他们都可以是主体，然后来进行认证授权等操作。
>
> 中间层左边：核心组件，例如认证、授权、会话管理、缓存管理、realm、会话dao等。



- **Subject**

> 主体，外部应用与Subject交互，Subject将用户作为当前操作的主体，这个主体可以是通过浏览器请求的用户，也可以是一个应用程序。Subject在Shiro中是一个接口，定义了很多认证授权的方法，外部程序通过Subject进行认证授权，而Subject是通过Security Manager进行认证授权。

- **Security Manager**

> 安全管理器，Security Manager对所有的Subject进行安全管理。通过Security Manager可以完成对Subject的认证、授权等操作。 Security Manager中**Authentication**进行认证，通过**Authorize**r进行授权，通过Session Manager进行会话管理。

- **Authenticator**

> 认证器，即对用户身份进行认证。

- **Authorizer**

> 授权器，用户通过认证后，访问其他受限资源时，需要通过授权器判断该用户是否具有访问此功能的权限。

- **Realm**

> 领域，可以看成数据源（DataSource），Security Manager进行认证和授权的时候，需要获取用户或用户权限数据，这个Realm就是来完成认证、授权数据的获取的。

- **Session Manager**

> 会话管理，Shiro自己定义了一套会话管理，它不依赖于我们 Servlet 的 session ，所以Shiro可以用在非web应用上，也可以将会话统一集中在一个点来管理，所以Shiro也可以实现单点登录。

- **Session DAO**

> 会话DAO，是对会话操作（创建、删除、修改）的一套接口。

- **Cache Manager**

> 缓存管理，比如将用户权限数据存储在缓存中，这样可以提高性能。

- **Cryptography**

> 密码管理，提供了一套加密/解密的组件，方便开发，比如散列，MD5等。

三、Shiro认证

（一）身份认证

> 身份认证，就是**判断一个用户是合法用户的处理过程。**最常用的简单身份认证方式是通过核对用户名和密码，看是否与数据库中存储的该用户的用户名和口令一致，用这种方式来判断用户身份是否正确。

身份认证中的关键对象：

- **Subject：主体**

> 访问系统的用户，也就是主体。

- **Principal：身份信息**

> 主体进行身份认证的标识，该标识必须唯一。一个主体可以有多个身份信息，但必须有一个主身份(Primary Principal)，这个主身份唯一且不能重复。 多个身份标识比如，用户名，手机号，邮箱地址，这些都是主体的身份标识，但用户名就可以作为主身份，因为它唯一且不重复。

- **Credential：凭证信息**

> 主体进行身份认证的凭证。说白了就是只有主体自己知道的安全信息，别人不知道的，比如说私钥，密码等。

（二）HelloWorld

导入依赖：

```
<dependencies>
    <!--Shiro核心包-->
    <dependency>
        <groupId>org.apache.shiro</groupId>
        <artifactId>shiro-core</artifactId>
        <version>1.9.1</version>
    </dependency>
    <!--日志相关包-->
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>jcl-over-slf4j</artifactId>
        <version>2.0.0-alpha2</version>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-log4j12</artifactId>
        <version>2.0.0-alpha2</version>
    </dependency>
    <dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
        <version>1.2.17</version>
    </dependency>
</dependencies>
```

定义用户信息：shiro.ini

```
# 用户
[users]
# 账号：root   密码：secret   角色：admin
root = secret, admin
# 账号：guest  密码：guest    角色：guest
guest = guest, guest
# 账号：presidentskroob    密码：123456   角色：president
presidentskroob = 12345, president
# 账号：darkhelmet 密码：ludicrousspeed   角色：darklord、schwartz
darkhelmet = ludicrousspeed, darklord, schwartz
# 账号：lonestarr  密码：vespa    角色：goodguy、schwartz
lonestarr = vespa, goodguy, schwartz

# 角色
[roles]
# admin 角色拥有所有权限，由通配符“*”表示
admin = *
# The 'schwartz' role can do anything (*) with any lightsaber:
# schwartz 角色拥有 lightsaber 资源的所有权限
schwartz = lightsaber:*
# goodguy 角色拥有 winnebago 资源的 drive 权限，且只对 eagle5 实例起作用
goodguy = winnebago:drive:eagle5
```

身份认证：

```java
import org.apache.shiro.mgt.SecurityManager;

public class Quickstart {

    private static final transient Logger log = LoggerFactory.getLogger(Quickstart.class);

    public static void main(String[] args) {

        // The easiest way to create a Shiro SecurityManager with configured
        // realms, users, roles and permissions is to use the simple INI config.
        // We'll do that by using a factory that can ingest a .ini file and
        // return a SecurityManager instance:

        // Use the shiro.ini file at the root of the classpath
        // (file: and url: prefixes load from files and urls respectively):
        // 用户的账号以及授权信息在 ini 文件中，采用对应的解析方式。通过 ini 工厂解析。
        Factory<SecurityManager> factory = new IniSecurityManagerFactory("classpath:shiro.ini");
        // 根据工厂获取安全管理器实例
        SecurityManager securityManager = factory.getInstance();

        // for this simple example quickstart, make the SecurityManager
        // accessible as a JVM singleton.  Most applications wouldn't do this
        // and instead rely on their container configuration or web.xml for
        // webapps.  That is outside the scope of this simple quickstart, so
        // we'll just do the bare minimum so you can continue to get a feel
        // for things.
        // 配置全局的安全管理器
        SecurityUtils.setSecurityManager(securityManager);

        // Now that a simple Shiro environment is set up, let's see what you can do:

        // get the currently executing user:
        // 获取Subject主体
        Subject currentUser = SecurityUtils.getSubject();

        /* Session 管理（选配） */
        // Do some stuff with a Session (no need for a web or EJB container!!!)
        // 获取通过主体获取 Session
        Session session = currentUser.getSession();
        // Session 中封装信息
        session.setAttribute("someKey", "aValue");
        // 获取 Session 中的信息
        String value = (String) session.getAttribute("someKey");
        // 验证从Session中获取的数据
        if (value.equals("aValue")) {
            log.info("Retrieved the correct value! [" + value + "]");
        }
        /* Session 管理（选配） */

```

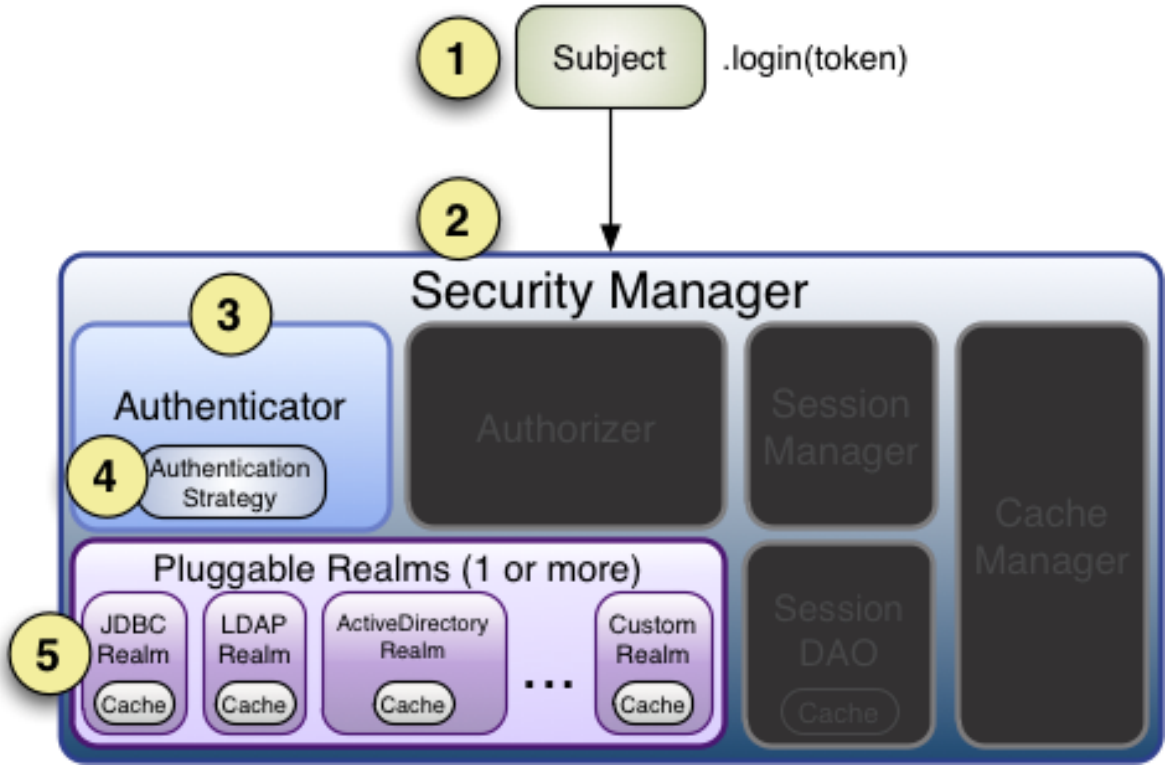
```
        // let's login the current user so we can check against roles and
permissions:

        /* 认证（标配） */
        // 验证是否通过认证，是否已经登录
        if (!currentUser.isAuthenticated()) {
            // 没有通过认证时，需要将账号和密码信息封装到UsernamePasswordToken对象
中
            UsernamePasswordToken token = new
UsernamePasswordToken("lonestarr", "vespa");
            // 设置记住我
            token.setRememberMe(true);
            try {
                // 执行登录，即认证操作
                currentUser.login(token);

            // 未知账号异常
            } catch (UnknownAccountException uae) {
                log.info("There is no user with username of " +
token.getPrincipal());
            // 错误凭证异常
            } catch (IncorrectCredentialsException ice) {
                log.info("Password for account " + token.getPrincipal() + "
was incorrect!");
            // 用户锁定异常
            } catch (LockedAccountException lae) {
                log.info("The account for username " + token.getPrincipal()
+ " is locked.  " +
                        "Please contact your administrator to unlock it.");
            }
            // ... catch more exceptions here (maybe custom ones specific
to your application?
            // 认证异常
            catch (AuthenticationException ae) {
                //unexpected condition?  error?
            }
        }

        //say who they are:
        //print their identifying principal (in this case, a username):
        log.info("User [" + currentUser.getPrincipal() + "] logged in
successfully.");
        /* 认证（标配） */

        /* 授权（标配） */
        //test a role:
        // 验证角色
        if (currentUser.hasRole("schwartz")) {
            log.info("May the Schwartz be with you!");
        } else {
            log.info("Hello, mere mortal.");
        }

        //test a typed permission (not instance-level)
        // 验证权限
        if (currentUser.isPermitted("lightsaber:wield")) {
            log.info("You may use a lightsaber ring.  Use it wisely.");
        } else {
```

```
            log.info("Sorry, lightsaber rings are for schwartz masters
only.");
        }

        //a (very powerful) Instance Level permission:
        if (currentUser.isPermitted("winnebago:drive:eagle5")) {
            log.info("You are permitted to 'drive' the winnebago with
license plate (id) 'eagle5'.  " +
                    "Here are the keys - have fun!");
        } else {
            log.info("Sorry, you aren't allowed to drive the 'eagle5'
winnebago!");
        }
        /* 授权（标配） */

        //all done - log out!
        // 登出，退出
        currentUser.logout();

        System.exit(0);
    }
}
```

核心步骤：

1. 获取 `SecurityManager` 对象，不同工厂获取的对象不同。
2. 将获取到的 `SecurityManager` 对象绑定到 Shiro 环境中，这个设置是全局的，设置一次即可。
3. 通过工具类 `SecurityUtils` 获取主体，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；
4. 封装主体需要的登录信息`UsernamePasswordToken`。
5. 通过主体，执行 `subject.login()` 方法。
6. 认证身份和权限。
7. 调用 `subject.logout()` 方法登出。

（三）身份认证流程



1、首先调用`Subject.login(token)`进行登录，其会自动委托`Security Manager`；

2、`Security Manager`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`Authentication Strategy`（认证策略）进行多 Realm 身份验证，默认`ModularRealmAuthenticator`会调用`AuthenticationStrategy`进行多Realm身份验证；

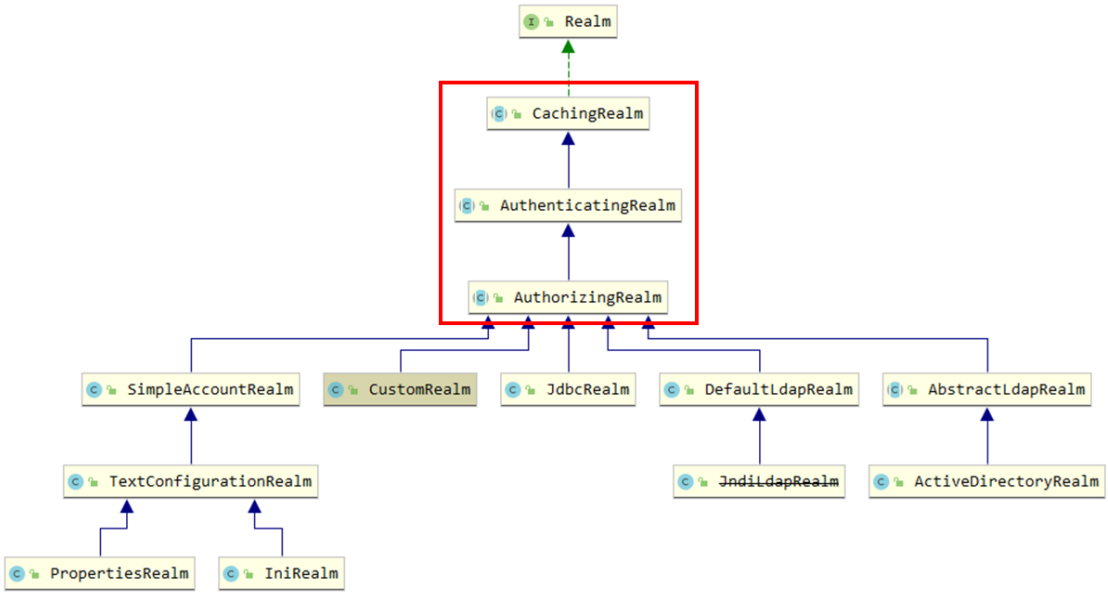
5、`Authenticator`会把相应的 token 传入 Realm，从 Realm 获取身份验证信息，如果没有返回/抛出异常表示身份验证失败了。此处可以配置多个Realm，将按照相应的顺序及策略进行访问。

（四）Realm

1、内置Realm

> Realm：域，Shiro 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。之前的读取 shiro.ini 配置方式时使用`org.apache.shiro.realm.text.IniRealm`。

Realm 架构：



> 如上图，Realm 是顶层父接口，有三个主要实现类：支持缓存的 `CachingRealm` 、支持认证的 `AuthenticatingRealm`、支持授权的 `AuthorizingRealm`，三者是继承关系，只要继承了 `AuthorizingRealm`类就可以实现缓存、认证、授权功能；

常用 Realm：

- `org.apache.shiro.realm.text.IniRealm`：**读取int文件时使用，**

  [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`： **读取properties文件时使用，**

  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`：**查询数据库时使用**

  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；

2、自定义Realm

> 自定义Realm就是为了自定义获取认证信息、授权信息的作用，Shiro默认的用户数据是从shiro.ini文件中获取的，使用的Realm是IniRealm；
>
> 实际项目中用户的数据是从数据库获取，所以现在我们要自定义获取数据的方法，那么就要自定义Realm，自定义Realm需要继承 `AuthorizingRealm` 这个抽象类，并实现其中的 `doGetAuthorizationInfo` 和 `doGetAuthenticationInfo` 方法；
>
> `doGetAuthorizationInfo` 方法：是用于获取授权信息的方法
>
> `doGetAuthenticationInfo` 方法：是用于获取认证信息的方法

自定义Realm：

```java
public class MyRealm extends AuthorizingRealm {
    // 这里模拟数据库，使用 Map 存储账号密码
    Map<String, String> map = null;

    {
        // 账号密码存储集合
        map = new HashMap<>();
        map.put("admin", "123123");
    }

    /**
     * 主体信息授权
     * @param principalCollection
     * @return
     */
    @Override
    protected AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principalCollection) {
        return null;
    }

    /**
     * 主体信息认证
     * @param authenticationToken 认证信息令牌，包含了主体传递的信息
     * @return
     * @throws AuthenticationException
     */
    @Override
    protected AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken authenticationToken) throws AuthenticationException {

        // AuthenticationToken 是父接口，调用login() 是使用的是 UsernamePasswordToken 封装的信息，所以这里要做类型转换
        UsernamePasswordToken upToken = (UsernamePasswordToken) authenticationToken;

        // 是一个表示用户的唯一属性，可以是用户名，邮箱之类的。
        String principal = (String) upToken.getPrincipal();
        // 根据用户从数据库获取密码，便于和用户输入的密码对比
        String pass = getPass(principal);
        // 获取主体输入的密码并转换为字符类型便于判断
        char[] chars = (char[]) authenticationToken.getCredentials();
        String credentials = new String(chars);

        // 获取当前使用的 realm 对象的名字
        String realmName = getName();

        /**
         * principal：是一个表示用户的唯一属性，可以是用户名，邮箱之类的。
         * credentials：是证明用户身份的证书，可以是密码或者指纹之类的。根据用户名获取的密码传入，和用户输入的密码进行校验
         * realmName：当前 Realm 对象的名字，调用父类的 getName() 方法即可。
         */
        // 返回的认证对象
        SimpleAuthenticationInfo authenticationInfo = new SimpleAuthenticationInfo(principal, pass, realmName);

        return authenticationInfo;
```

```
    }

    /**
     * 模拟从数据库获取密码
     * @param user
     * @return
     */
    public String getPass(String user){
        return map.get(user);
    }
}
```

测试：

```
package com.soft;

import com.soft.realm.MyRealm;
import org.apache.shiro.SecurityUtils;
import org.apache.shiro.authc.UsernamePasswordToken;
import org.apache.shiro.authc.credential.HashedCredentialsMatcher;
import org.apache.shiro.crypto.hash.Md5Hash;
import org.apache.shiro.mgt.DefaultSecurityManager;
import org.apache.shiro.subject.Subject;
import org.junit.Test;

public class TestRealm {
    @Test
    public void myRealmTest(){
        MyRealm myRealm = new MyRealm();

        // 创建默认安全管理器，并设置自定义Realm
        DefaultSecurityManager defaultSecurityManager = new
DefaultSecurityManager();
        defaultSecurityManager.setRealm(myRealm);

        // 设置全局安全管理器
        SecurityUtils.setSecurityManager(defaultSecurityManager);

        // 获取主体信息
        Subject subject = SecurityUtils.getSubject();

        // 账号密码token：用户输入的账号密码
        String username = "admin";
        String password = "123123";

        UsernamePasswordToken token = new UsernamePasswordToken(username,
password);
        // 主体登录
        subject.login(token);

        // 认证结果
        System.out.println("认证结果：" + subject.isAuthenticated());
    }
}
```

3、加盐（Salt）

> 项目中用户存储在数据库中用于认证的凭据一般都是加密后的密文，不会使用明文存储，以免造成信息泄露。
>
> 常见的加密方式是MD5，将用户输入的凭据通过MD5加密后存储到数据库中；
>
> 认证时同样将用户数据的凭据进行加密同时和数据库中存储的密文进行比对。

```
用户凭据：123456
MD5加密后的密文：49ba59abbe56e057
```

> MD5加密是**不可逆的**，如果用户设置的凭据过于简单，那么密码是可以破译的。此时的破译并不是真正的解密，而是通过和已知的加密结果一一对比来破译密码。如果对比到了那么就被破解了，这种方式不太靠谱，耗时长，但是如果加密前内容比较简单，那么隐患就比较大，所以我们MD5加密的时候需要配置着**盐(slat)** 来处理。
>
> **盐(slat)** 就是一串字符，在原有明文上加上这串盐，再进行加密操作，登陆时候，将原有密码加上这串盐，再加密后和数据库比对。
>
> 盐和MD5的一般搭配方式有：

```
盐 + 明文 -> 加密 -> aa + 123456 -> 密文
明文 + 盐 -> 加密 -> 123456 + aa -> 密文
盐 + 明文 + 盐 -> 加密 aa + 123456 + bb -> 密文
```

> 明文里夹杂着盐，盐使用散列的方式加进来。或者说你可以不止加盐，加用户名，手机号，生日，邮箱，地址等，统统都可以。还可以对加密的次数进行设计，明文加盐加密后再次进行加密。

自定义Realm：

```
public class MyRealm extends AuthorizingRealm {
    // 账号密码存储集合
    Map<String, String> map = null;

    {
        // 账号密码存储集合
        map = new HashMap<>();
        // 加密后的密文
        map.put("admin", "0e105779d9782a66022127c86d9d5054");
    }

    /**
     * 主体信息授权
     * @param principalCollection
     * @return
     */
    @Override
    protected AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection
principalCollection) {
        // 获取用户主体信息
        String primaryPrincipal = (String)
principalCollection.getPrimaryPrincipal();
        // 根据用户主体信息获取主体对应的权限信息

        // 返回的授权对象
```

```
        SimpleAuthorizationInfo authorizationInfo = new SimpleAuthorizationInfo();

        Set<String> roles = getRoles(primaryPrincipal);
        Set<String> permissions = getPermissions(primaryPrincipal);

        // 设置角色信息
        authorizationInfo.setRoles(roles);
        // 设置权限信息
        authorizationInfo.setStringPermissions(permissions);

        return authorizationInfo;
    }

    /**
     * 主体信息认证
     * @param authenticationToken 认证信息令牌，包含了主体传递的信息
     * @return
     * @throws AuthenticationException
     */
    @Override
    protected AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken authenticationToken) throws AuthenticationException {

        // AuthenticationToken 是父接口，调用login() 是使用的是 UsernamePasswordToken 封装的信息，所以这里要做类型转换
        UsernamePasswordToken upToken = (UsernamePasswordToken) authenticationToken;

        // 是一个表示用户的唯一属性，可以是用户名，邮箱之类的。
        String principal = (String) upToken.getPrincipal();
        // 根据用户从数据库获取密码，便于和用户输入的密码对比
        String pass = getPass(principal);
        // 获取主体输入的密码并转换为字符类型便于判断
        char[] chars = (char[]) authenticationToken.getCredentials();
        String credentials = new String(chars);

        // 获取当前使用的 realm 对象的名字
        String realmName = getName();

        /**
         * principal：是一个表示用户的唯一属性，可以是用户名，邮箱之类的。
         * credentials：是证明用户身份的证书，可以是密码或者指纹之类的。根据用户名获取的密码传入，和用户输入的密码进行校验
         * realmName：当前 Realm 对象的名字，调用父类的 getName() 方法即可。
         */
        // 返回的认证对象
        SimpleAuthenticationInfo authenticationInfo = new SimpleAuthenticationInfo(principal, pass, realmName);
        // ------------------------------------------------------------------------
        authenticationInfo.setCredentialsSalt(ByteSource.Util.bytes("riu"));
        // ------------------------------------------------------------------------

        return authenticationInfo;
    }

```

```
    /**
     * 模拟从数据库获取密码
     * @param user
     * @return
     */
    public String getPass(String user){
        return map.get(user);
    }

    /**
     * MD5加密工具
     * @param str 明文密码
     * @param salt 盐
     * @return
     */
    public String md5(String str, String salt){
        /**
         * source：明文密码
         * salt：盐
         * hashIterations：加密次数
         */
        Md5Hash md5Hash = new Md5Hash(str, salt, 2);

        return md5Hash.toString();
    }
}
```

测试：

```
public class TestRealm {
    @Test
    public void myRealmTest(){
        MyRealm myRealm = new MyRealm();

        // ------------------------------------------------------------------------
        // 定义一个加密算法
        HashedCredentialsMatcher matcher = new HashedCredentialsMatcher();
        // 设置加密算法
        matcher.setHashAlgorithmName("md5");
        // 设置加密次数
        matcher.setHashIterations(2);
        // 自定义 Realm 设置解密算法
        myRealm.setCredentialsMatcher(matcher);
        // ------------------------------------------------------------------------

        // 创建默认安全管理器，并设置自定义Realm
        DefaultSecurityManager defaultSecurityManager = new DefaultSecurityManager();
        defaultSecurityManager.setRealm(myRealm);

        // 设置全局安全管理器
        SecurityUtils.setSecurityManager(defaultSecurityManager);

        // 获取主体信息
        Subject subject = SecurityUtils.getSubject();
```

```
26
27          // 账号密码token：用户输入的账号密码
28          String username = "admin";
29          String password = "123123";
30
31          UsernamePasswordToken token = new UsernamePasswordToken(username,
    password);
32          // 主体登录
33          subject.login(token);
34
35          // 认证结果
36          System.out.println("认证结果：" + subject.isAuthenticated());
37      }
38  }
```

四、Shiro授权

（一）授权

> 授权，也就是访问控制，**控制谁能访问哪些资源**，这个"谁"是认证后的主体/用户，在访问受限资源(受限制的资源或需要经过授权后才能访问的资源)以前，需要先经过授权。

授权中的关键对象：

> 也就是 **who（谁）** 对 **what（什么）** 进行 **how（如何）** 操作。

- **who：**

> 即主体Subject，主体需要访问系统中的资源

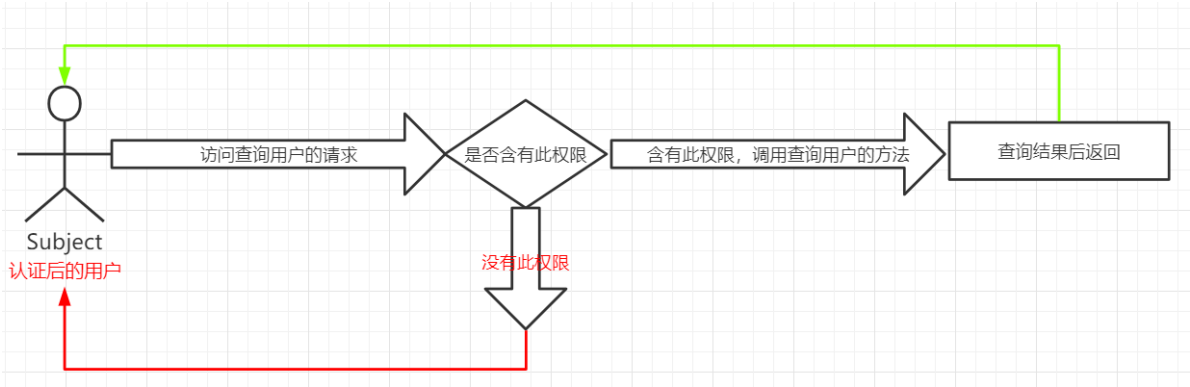
- **what：**

> 即资源 Resource，如某个请求，按钮，方法等，资源包括资源类型和资源实例，比如：商品信息为资源类型，id为10001的商品资源实例。

- **how：**

> 即权限Permission，规定了主体对资源的操作许可，权限离开了资源是没有意义的。权限这个比如：查询用户权限、添加用户权限等。

（二）授权流程



（三）授权方式

1、基于角色授权

> RBAC基于角色的访问控制（Role-Based Access Control）是以角色为中心进行访问控制

```
// 判断主体是否有指定角色
if(subject.hasRole("admin")){
    // 操作什么资源
}
```

2、基于资源授权

> RBAC基于资源的访问控制（Resource-Based Access Control）是以资源为中心进行访问控制

```
// 判断主体对具体的资源是否有操作权限
if(subject.isPermission("user:update:01")){ // 资源实例
  // 对01用户进行修改
}

// 判断主体对某些的资源是否有操作权限
if(subject.isPermission("user:update:*")){  // 资源类型
  // 对01用户进行修改
}
```

权限结构表达式

```
权限结构为：
    资源标识符:操作1,操作2,...:对象实例ID

解释说明：
    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；
```

定义资源标识符为：user

2.1 粗粒度权限：

- 某个资源的单一权限

```
# user资源的添加权限
user:add
```

- 某个资源的多个权限

```
# user资源的添加和删除权限
# 写法一：
user:add,user:del
# 写法二：
user:add,del
```

- 某个资源的所有权限

```
# user资源的增删改查权限
# 写法一：
user:add,del,update,view
# 写法二：
user:*
```

- 多个资源的某个权限

```
# 资源使用通配符表示，权限为查看
*:view
```

2.2 细粒度权限：

- 某个实例的某个权限

```
# user资源允许实例1001执行add操作
user:add:1001
```

- 某个实例的多个权限

```
# user资源允许实例1001执行add和del操作
user:add,del:1001
```

- 所有实例的某个权限

```
# user资源允许所有实例执行add操作
# 写法一：
user:add:*
# 写法二：简写
user:add
```

3、授权方式

- 编程式

```
// 主体
Subject subject = SecurityUtils.getSubject();
// 是否有角色
subject.hasRole("admin")
// 是否有权限
subject.isPermitted("user:add");
```

- 注解式

> 后面 SpringBoot 整合 Shiro 进行讲解

```java
@RequiresRoles("admin")
public void hello() {
    // 有权限
}

@RequiresPermissions("user:add")
public void hello() {
    // 有权限
}
```

- 标签式

> 后面 SpringBoot 整合 Shiro 进行讲解

```jsp
<!-- JSP/GSP 标签：在JSP/GSP 页面通过相应的标签完成： -->
<shiro:hasRole name="admin">
    <!– 有权限–>
</shiro:hasRole>
<!-- 注意: Thymeleaf 中使用shiro需要额外集成!-->
```

4、Shiro 授权

自定义Realm：授权逻辑在 `doGetAuthorizationInfo` 中编写

```java
public class MyRealm extends AuthorizingRealm {
    // 账号密码存储集合
    Map<String, String> map = null;
    // 角色集合
    Set<String> roles = null;
    // 权限集合
    Set<String> stringPermissions = null;
    // 用户对应的角色
    Map<String, Set> userRole = null;
    // 角色对应的权限
    Map<String, Set> rolePermission = null;

    {
        // 账号密码存储集合
        map = new HashMap<>();
        // 加密后的密文
        map.put("admin", "0e105779d9782a66022127c86d9d5054");

        // 角色集合
        roles = new HashSet<>();
        roles.add("admin");
        roles.add("user");

        // 权限集合
        stringPermissions = new HashSet<>();
        stringPermissions.add("user:add");
        stringPermissions.add("user:del");

        // 用户角色
        userRole = new HashMap();
        userRole.put("admin", roles);

```

```
        // 角色权限
        rolePermission = new HashMap<>();
        rolePermission.put("admin", stringPermissions);
    }

    /**
     * 主体信息授权
     * @param principalCollection
     * @return
     */
    @Override
    protected AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principalCollection) {
        // 获取用户主体信息
        String primaryPrincipal = (String) principalCollection.getPrimaryPrincipal();
        // 根据用户主体信息获取主体对应的权限信息

        // 返回的授权对象
        SimpleAuthorizationInfo authorizationInfo = new SimpleAuthorizationInfo();

        Set<String> roles = getRoles(primaryPrincipal);
        Set<String> permissions = getPermissions(primaryPrincipal);

        // 设置角色信息
        authorizationInfo.setRoles(roles);
        // 设置权限信息
        authorizationInfo.setStringPermissions(permissions);

        return authorizationInfo;
    }

    /**
     * 模拟数据库获取权限信息
     * @param user
     * @return
     */
    public Set<String> getPermissions(String user){
        return rolePermission.get(user);
    }

    /**
     * 模拟数据库获取用户对应角色信息
     * @param user
     * @return
     */
    public Set<String> getRoles(String user){
        return userRole.get(user);
    }

    /**
     * 主体信息认证
     * @param authenticationToken 认证信息令牌，包含了主体传递的信息
     * @return
     * @throws AuthenticationException
     */
    @Override
```

```
    protected AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken authenticationToken) throws AuthenticationException {

        // AuthenticationToken 是父接口，调用login() 是使用的是 UsernamePasswordToken 封装的信息，所以这里要做类型转换
        UsernamePasswordToken upToken = (UsernamePasswordToken) authenticationToken;

        // 是一个表示用户的唯一属性，可以是用户名，邮箱之类的。
        String principal = (String) upToken.getPrincipal();
        // 根据用户从数据库获取密码，便于和用户输入的密码对比
        String pass = getPass(principal);
        // 获取主体输入的密码并转换为字符类型便于判断
        char[] chars = (char[]) authenticationToken.getCredentials();
        String credentials = new String(chars);

        // 获取当前使用的 realm 对象的名字
        String realmName = getName();

        /**
         * principal：是一个表示用户的唯一属性，可以是用户名，邮箱之类的。
         * credentials：是证明用户身份的证书，可以是密码或者指纹之类的。根据用户名获取的密码传入，和用户输入的密码进行校验
         * realmName：当前 Realm 对象的名字，调用父类的 getName() 方法即可。
         */
        // 返回的认证对象
        SimpleAuthenticationInfo authenticationInfo = new SimpleAuthenticationInfo(principal, pass, realmName);
        // ------------------------------------------------------------------

 authenticationInfo.setCredentialsSalt(ByteSource.Util.bytes("riu"));
        // ------------------------------------------------------------------

        return authenticationInfo;
    }

    /**
     * 模拟从数据库获取密码
     * @param user
     * @return
     */
    public String getPass(String user){
        return map.get(user);
    }

    /**
     * MD5加密
     * @param str 明文密码
     * @param salt 盐
     * @return
     */
    public String md5(String str, String salt){
        /**
         * source：明文密码
         * salt：盐
```

```
         * hashIterations：加密次数
         */
        Md5Hash md5Hash = new Md5Hash(str,salt,2);

        return md5Hash.toString();
    }
}
```

测试

```
public class TestRealm {
    @Test
    public void myRealmTest(){
        MyRealm myRealm = new MyRealm();

        // ------------------------------------------------------------------------
        // 定义一个加密算法
        HashedCredentialsMatcher matcher = new HashedCredentialsMatcher();
        // 设置加密算法
        matcher.setHashAlgorithmName("md5");
        // 设置加密次数
        matcher.setHashIterations(2);
        // 自定义 Realm 设置解密算法
        myRealm.setCredentialsMatcher(matcher);
        // ------------------------------------------------------------------------

        // 创建默认安全管理器，并设置自定义Realm
        DefaultSecurityManager defaultSecurityManager = new DefaultSecurityManager();
        defaultSecurityManager.setRealm(myRealm);

        // 设置全局安全管理器
        SecurityUtils.setSecurityManager(defaultSecurityManager);

        // 获取主体信息
        Subject subject = SecurityUtils.getSubject();

        // 账号密码token：用户输入的账号密码
        String username = "admin";
        String password = "123123";

        UsernamePasswordToken token = new UsernamePasswordToken(username, password);
        // 主体登录
        subject.login(token);

        // 认证结果
        System.out.println("认证结果：" + subject.isAuthenticated());

        // 验证角色
        subject.checkRoles("admin", "user");
        // 验证权限
        subject.checkPermissions("user:add", "user:del");
    }
}
```

五、SpringBoot 整合 Shiro

（一）搭建基础环境

```
单独使用Shiro时的环境配置
    1、自定义Realm 对象
    2、SecurityManager 对象，在对象中添加自己的Realm
```

1、创建项目，添加Shiro依赖

```
<dependency>
    <groupId>org.apache.shiro</groupId>
    <artifactId>shiro-spring-boot-starter</artifactId>
    <version>1.9.0</version>
</dependency>
```

2、创建自定义Realm

```
public class MyRealm extends AuthorizingRealm {

    /**
     * 授权
     * @param principalCollection
     * @return
     */
    @Override
    protected AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principalCollection) {
        return null;
    }

    /**
     * 认证
     * @param authenticationToken
     * @return
     * @throws AuthenticationException
     */
    @Override
    protected AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken authenticationToken) throws AuthenticationException {
        return null;
    }
}

```

3、创建 Shiro 配置类

```
package com.soft.config;


import com.soft.realm.MyRealm;
import org.apache.shiro.realm.Realm;
import org.apache.shiro.spring.web.ShiroFilterFactoryBean;
import org.apache.shiro.web.mgt.DefaultWebSecurityManager;
```

```java
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.apache.shiro.mgt.SecurityManager;

@Configuration
public class ShiroConfig {

    /**
     * 容器中注入自定义Realm
     * @return
     */
    @Bean
    public Realm realm(){
        MyRealm realm = new MyRealm();
        return realm;
    }

    /**
     * 容器中注入Web安全管理
     * @param realm 安全管理对象需要依据Realm
     * @return
     */
    @Bean
    public DefaultWebSecurityManager defaultWebSecurityManager(Realm realm){
        DefaultWebSecurityManager defaultWebSecurityManager = new
DefaultWebSecurityManager();

        defaultWebSecurityManager.setRealm(realm);

        return defaultWebSecurityManager;
    }

    /**
     * Shilr过滤器工厂，用于过滤请求路径
     * @param securityManager
     * @return
     */
    @Bean
    public ShiroFilterFactoryBean getShiroFilterFactoryBean(SecurityManager
securityManager) {
        ShiroFilterFactoryBean shiroFilterFactoryBean = new
ShiroFilterFactoryBean();
        shiroFilterFactoryBean.setSecurityManager(securityManager);
        return shiroFilterFactoryBean;
    }
}

```

4、访问资源

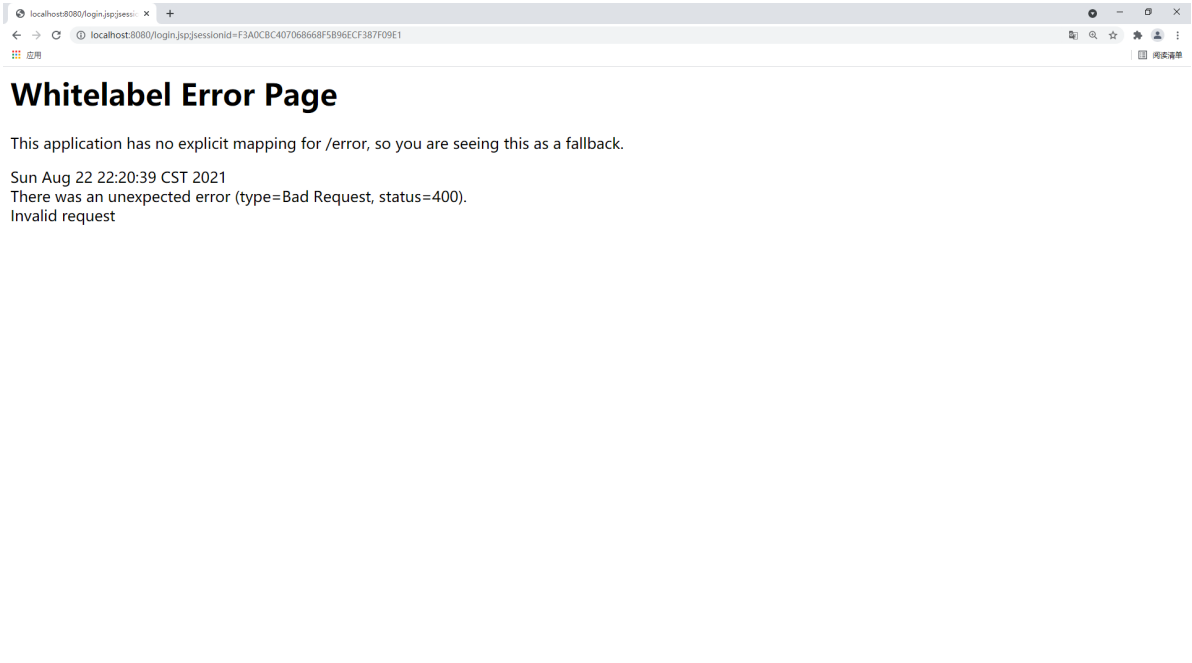
- 新建 list.html 页面
- 新建 UserController 后台，通过后台访问 list.html 资源

资源可以轻松访问，原因是虽然已经添加了 Shiro 但还未进行认证和授权，默认所有的资源都可以访问；

配置路径过滤：

```java
/**
 * Shilr过滤器工厂，用于过滤请求路径
 * @param securityManager
 * @return
 */
@Bean
public ShiroFilterFactoryBean getShiroFilterFactoryBean(SecurityManager securityManager) {
    ShiroFilterFactoryBean shiroFilterFactoryBean = new ShiroFilterFactoryBean();
    shiroFilterFactoryBean.setSecurityManager(securityManager);

    Map<String, String> filterChainDefinitionMap = new HashMap<>();
    // 配置所有的资源路径都要认证
    filterChainDefinitionMap.put("/**/*", "authc");

    // 设置过滤资源

    shiroFilterFactoryBean.setFilterChainDefinitionMap(filterChainDefinitionMap);

    return shiroFilterFactoryBean;
}
```

配置缩写	对应的过滤器	功能
anon	AnonymousFilter	指定url可以匿名访问
authc	FormAuthenticationFilter	指定url需要form表单登录，默认会从请求中获取`username`、`password`,`rememberMe`等参数并尝试登录，如果登录不了就会跳转到loginUrl配置的路径。我们也可以用这个过滤器做默认的登录逻辑，但是一般都是我们自己在控制器写登录逻辑的，自己写的话出错返回的信息都可以定制嘛。
authcBasic	BasicHttpAuthenticationFilter	指定url需要basic登录
logout	LogoutFilter	登出过滤器，配置指定url就可以实现退出功能，非常方便
noSessionCreation	NoSessionCreationFilter	禁止创建会话
perms	PermissionsAuthorizationFilter	需要指定权限才能访问
port	PortFilter	需要指定端口才能访问
rest	HttpMethodPermissionFilter	将http请求方法转化成相应的动词来构造一个权限字符串，这个感觉意义不大，有兴趣自己看源码的注释
roles	RolesAuthorizationFilter	需要指定角色才能访问
ssl	SslFilter	需要https请求才能访问
user	UserFilter	需要已登录或“记住我”的用户才能访问



> 加入了路径过滤之后，当前请求的资源需要先被进行认证才可以访问。所以当请求资源且没有认证时 Shiro 会默认请求项目跟路径下的登录页面，让用户进行认证。当然也可以通过 `shiroFilterFactoryBean.setLoginUrl("");` 配置请求自定义路径下的登录页面

（二）认证

> 账号：admin
>
> 密码：123123

1、编写控制层，用于处理登录请求

```
@Controller
@RequestMapping("/user")
public class UserController {

    @GetMapping("/list")
    public String list(){
        return "list.html";
    }

    @PostMapping("/login")
    public String login(String user, String pass){
        UsernamePasswordToken upToken = new UsernamePasswordToken(user,
pass);

        // 获取用户主体
        Subject subject = SecurityUtils.getSubject();
        try {
            // 用户登录认证
            subject.login(upToken);
            if(subject.isAuthenticated()){
                return "list.html";
            }
        } catch (UnknownAccountException e) {
            System.out.println("用户名不存在！");
            e.printStackTrace();
        } catch (IncorrectCredentialsException e) {
            System.out.println("密码错误！");
```

```
            e.printStackTrace();
        } catch (AuthenticationException e) {
            System.out.println("认证时出现异常！");
            e.printStackTrace();
        }
        return "login.html";
    }
}
```

2、修改自定义 Realm，添加认证逻辑

```
package com.soft.realm;

import org.apache.shiro.authc.AuthenticationException;
import org.apache.shiro.authc.AuthenticationInfo;
import org.apache.shiro.authc.AuthenticationToken;
import org.apache.shiro.authc.SimpleAuthenticationInfo;
import org.apache.shiro.authz.AuthorizationInfo;
import org.apache.shiro.realm.AuthorizingRealm;
import org.apache.shiro.subject.PrincipalCollection;

public class MyRealm extends AuthorizingRealm {

    /**
     * 授权
     * @param principalCollection
     * @return
     */
    @Override
    protected AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principalCollection) {
        return null;
    }

    /**
     * 认证
     * @param authenticationToken
     * @return
     * @throws AuthenticationException
     */
    @Override
    protected AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken authenticationToken) throws AuthenticationException {

        // 获取主体信息
        String principal = (String) authenticationToken.getPrincipal();

        // 模拟从数据库获取账号对应密码
        String pass = "123123";

        // 认证信息
        SimpleAuthenticationInfo authenticationinfo = new SimpleAuthenticationInfo(principal, pass, getName());

        return authenticationinfo;
    }
}
```

3、设置为匿名访问

```
@Configuration
public class ShiroConfig {

    /**
     * 容器中注入自定义Realm
     * @return
     */
    @Bean
    public Realm realm(){
        MyRealm realm = new MyRealm();
        return realm;
    }

    /**
     * 容器中注入Web安全管理
     * @param realm 安全管理对象需要依据Realm
     * @return
     */
    @Bean
    public DefaultWebSecurityManager defaultWebSecurityManager(Realm realm){
        DefaultWebSecurityManager defaultWebSecurityManager = new
DefaultWebSecurityManager();

        defaultWebSecurityManager.setRealm(realm);

        return defaultWebSecurityManager;
    }

    /**
     * Shilr过滤器工厂，用于过滤请求路径
     * @param securityManager
     * @return
     */
    @Bean
    public ShiroFilterFactoryBean getShiroFilterFactoryBean(SecurityManager
securityManager) {
        ShiroFilterFactoryBean shiroFilterFactoryBean = new
ShiroFilterFactoryBean();
        shiroFilterFactoryBean.setSecurityManager(securityManager);

        // 设置登录页面路径
        shiroFilterFactoryBean.setLoginUrl("/login.html");

        Map<String, String> filterChainDefinitionMap = new LinkedHashMap<>
();

        // 第一个参数：路径
        // 第二个参数：过滤器
        filterChainDefinitionMap.put("/login.html", "anon");
        filterChainDefinitionMap.put("/user/login", "anon");
        // 配置静态资源不需要认证
        filterChainDefinitionMap.put("/img/**", "anon");
        filterChainDefinitionMap.put("/js/**", "anon");
        filterChainDefinitionMap.put("/css/**", "anon");
```

```
        filterChainDefinitionMap.put("/fonts/**", "anon");

        // 配置所有的资源路径都要认证
        filterChainDefinitionMap.put("/**", "authc");
        // 设置过滤资源

 shiroFilterFactoryBean.setFilterChainDefinitionMap(filterChainDefinitionMap
);

        return shiroFilterFactoryBean;
    }
}
```

（三）退出认证

> 认证通过之后可以正常访问资源，此时 Shiro 已经记录下已认证状态。如果此时重新从登录页面输入账号页面，即便是错误的账号和密码依然可以访问资源。

编写退出认证

```
@GetMapping("/logout")
public String logout(){
    // 获取用户主体
    Subject subject = SecurityUtils.getSubject();
    subject.logout();

    return "redirect:/login.html";
}
```

（四）授权

1、编写 list.html 页面

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
    <style>
        .but{
            width: 100px;
            height: 30px;
            border-radius: 10px;
            color: white;
            border: none;
        }

        .view{
            background-color: coral;
        }
        .edit{
            background-color: blueviolet;
        }
        .add{
            background-color: greenyellow;
        }
```

```
        .del{
            background-color: red;
        }

    </style>
</head>
<body>
    <h3>详细信息页面</h3>
    <a href="/user/logout">退出登录</a>

    <hr>
    <a href="/user/view" class="but view">查看</a>
    <a href="/user/edit" class="but edit">修改</a>
    <a href="/user/add" class="but add">添加</a>
    <a href="/user/del" class="but del">删除</a>
</body>
</html>
```

2、自定义Realm编写授权逻辑

```
@Override
protected AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principalCollection) {
    // 获取主体信息
    String primaryPrincipal = (String) principalCollection.getPrimaryPrincipal();

    // 模拟数据库获取主体对应角色和权限
    Set<String> roles = new HashSet<>();
    roles.add("admin");

    Set<String> permissions = new HashSet<>();
    permissions.add("user:add");

    SimpleAuthorizationInfo authorizationInfo = new SimpleAuthorizationInfo();
    // 设置角色和权限
    authorizationInfo.setRoles(roles);
    authorizationInfo.setStringPermissions(permissions);
    return authorizationInfo;
}
```

3、授权控制

3.1 代码方式控制

```
@GetMapping("/view")
@ResponseBody
public String view(){
    Subject subject = SecurityUtils.getSubject();
    if (subject.hasRole("admin")) {
        return "有权访问：查看";
    }
    return "无权访问：查看";
}
```

3.2 注解方式控制

> 添加如下两个配置，让注解生效

```
@Bean
public static LifecycleBeanPostProcessor getLifecycleBeanPostProcessor() {
    return new LifecycleBeanPostProcessor();
}

@Bean
public static DefaultAdvisorAutoProxyCreator
getDefaultAdvisorAutoProxyCreator(){
    return new DefaultAdvisorAutoProxyCreator();
}
```

```
@GetMapping("/edit")
@ResponseBody
@RequiresRoles({"admin"})
public String edit(){
    return "有权访问：修改";
}

@GetMapping("/add")
@ResponseBody
@RequiresRoles({"admin"})
@RequiresPermissions({"user:add"})
public String add(){
    return "有权访问：添加";
}

@GetMapping("/del")
@ResponseBody
@RequiresRoles({"admin"})
@RequiresPermissions({"user:del"})
public String del(){
    return "有权访问：删除";
}
```

3.3 Shiro 标签控制

3.3.1 导入依赖

```
<dependency>
    <groupId>com.github.theborakompanioni</groupId>
    <artifactId>thymeleaf-extras-shiro</artifactId>
    <version>2.1.0</version>
</dependency>
```

3.3.2 配置Shiro 方言

```java
/**
* 配置ShiroDialect:用于thymeleaf和shiro标签配合使用
* @return
*/
@Bean
public ShiroDialect shiroDialect(){
return new ShiroDialect();
}
```

3.3.3 添加 shiro 标签

引入头信息

```
<html lang="en" xmlns:th="http://www.thymeleaf.org"
      xmlns:shiro="http://www.pollix.at/thymeleaf/shiro">
```

添加 Shiro 标签

```
<a shiro:hasRole="admin" href="/user/view" class="but view">查看</a>
<a shiro:hasPermission="user:edit" href="/user/edit" class="but edit">修改</a>
<a shiro:hasPermission="user:add"  href="/user/add" class="but add">添加</a>
<a shiro:hasPermission="user:del"  href="/user/del" class="but del">删除</a>
```

常用标签

```
<!-- 验证当前用户是否为“访客”，即未认证（包含未记住）的用户。 -->
<p shiro:guest="">Please <a href="login.html">login</a></p>


<!-- 认证通过或已记住的用户。 -->
<p shiro:user="">
    Welcome back John! Not John? Click <a href="login.html">here</a> to
login.
</p>

<!-- 已认证通过的用户。不包含已记住的用户，这是与user标签的区别所在。 -->
<p shiro:authenticated="">
    Hello, <span shiro:principal=""></span>, how are you today?
</p>
<a shiro:authenticated="" href="updateAccount.html">Update your contact
information</a>

<!-- 输出当前用户信息，通常为登录帐号信息。 -->
<p>Hello, <shiro:principal/>, how are you today?</p>


<!-- 未认证通过用户，与authenticated标签相对应。与guest标签的区别是，该标签包含已记住用
户。 -->
<p shiro:notAuthenticated="">
    Please <a href="login.html">login</a> in order to update your credit
card information.
</p>

<!-- 验证当前用户是否属于该角色。 -->
```

```
<a shiro:hasRole="admin" href="admin.html">Administer the system</a><!-- 拥有该角色 -->

<!-- 与hasRole标签逻辑相反，当用户不属于该角色时验证通过。 -->
<p shiro:lacksRole="developer"><!-- 没有该角色 -->
    Sorry, you are not allowed to developer the system.
</p>

<!-- 验证当前用户是否属于以下所有角色。 -->
<p shiro:hasAllRoles="developer, 2"><!-- 角色与判断 -->
    You are a developer and a admin.
</p>

<!-- 验证当前用户是否属于以下任意一个角色。  -->
<p shiro:hasAnyRoles="admin, vip, developer,1"><!-- 角色或判断 -->
    You are a admin, vip, or developer.
</p>

<!--验证当前用户是否拥有指定权限。  -->
<a shiro:hasPermission="userInfo:add" href="createUser.html">添加用户</a><!-- 拥有权限 -->

<!-- 与hasPermission标签逻辑相反，当前用户没有制定权限时，验证通过。 -->
<p shiro:lacksPermission="userInfo:del"><!-- 没有权限 -->
    Sorry, you are not allowed to delete user accounts.
</p>

<!-- 验证当前用户是否拥有以下所有角色。 -->
<p shiro:hasAllPermissions="userInfo:view, userInfo:add"><!-- 权限与判断 -->
    You can see or add users.
</p>

<!-- 验证当前用户是否拥有以下任意一个权限。  -->
<p shiro:hasAnyPermissions="userInfo:view, userInfo:del"><!-- 权限或判断 -->
    You can see or delete users.
</p>
<a shiro:hasPermission="pp" href="createUser.html">Create a new User</a>
```