

设计模式总结

我们先来回顾一下前面课程的内容：

设计模式分类

分类	关注点
创建型模式	关注于对象的创建，同时隐藏创建逻辑
结构型	关注类和对象之间的组合
行为型	关注对象之间的通信

设计模式定义

策略模式（行为型模式）

策略模式定义了算法族，分别封装起来，让他们之间可以互相替换，此模式让算法的变化独立于使用算法的客户

装饰器模式（结构型模式）

装饰器模式动态地将责任附加到对象上。若要扩展功能，装饰者提供了比继承更有弹性的替代方案。

观察者模式（行为型模式）

观察者模式定义了对象之间的一对多依赖，这样一来，当一个对象改变状态时，它的所有依赖者都会收到通知并自动更新。

适配器模式（结构型模式）

适配器模式将一个类的接口，转换成客户期望的另一个接口。适配器让原本接口不兼容的类可以合作无间

单例模式（创建型模式）

单例模式确保一个类只有一个实例，并提供一个全局访问点

模板方法模式（行为型模式）

模板方法模式在一个方法中定义一个算法的骨架，而将一些步骤延迟到子类中。模板方法使得子类可以在不改变算法结构的情况下，重新定义算法中的某些步骤。

外观模式（结构型模式）

外观模式提供了一个统一的接口，用来访问子系统中的一群接口。外观定义了一个高层接口，让子系统更容易使用。

代理模式（结构型模式）

代理模式为另一个对象提供一个替身或者占位符以控制对这个对象的访问

责任链模式（行为型模式）

责任链模式为某个请求创建一个对象链，每个对象依次检查此请求，并对其进行处理，或者将它传给链中的下一个对象

综合应用开发

本节课程将模仿shiro开发一个简单权限框架案例，笔者原本打算将上面几种设计模式都用上，但是实际开发这个demo的时候发现该场景下有些设计模式不适合。如果非要都用上，那必须得将这个框架开发得十分完善，这不是三五天能够完成的，也不是这一节课能够讲解完毕的。

因此最终只用了其中四种设计模式，希望能够帮助大家找到对设计模式的“感觉”。

主要使用了以下几种设计模式：

1. 单例模式
2. 工厂模式
3. 策略模式
4. 责任链模式

该权限框架主要有以下开发点：

- 读取配置文件
- 密码加密（策略模式，工厂模式）
- 身份认证（责任链模式，单例模式）
- 权限认证

读取配置文件

这里我们采用ini配置文件（配置文件位于src/permission.ini），格式如下：

```
配置1=值1
```

读取配置文件的工具类：

Config

```
package com.design.pattern.config;

import java.io.DataInputStream;
import java.io.InputStream;
import java.util.HashMap;
import java.util.Map;

public class Config {

    private static Map<String, String> configMap = new HashMap<>();
```

```

static {
    InputStream in = Config.class.getResourceAsStream("/permission.ini");
    DataInputStream dis = new DataInputStream(in);
    String str;
    try{
        while ((str = dis.readLine()) != null){
            String[] configs = str.split("=");
            if(configs.length == 2){
                configMap.put(configs[0].trim(),configs[1].trim());
            }
        }
        dis.close();
    }catch (Exception e){
        throw new RuntimeException("配置文件不存在");
    }
}

public static String get(String name){
    return configMap.get(name);
}

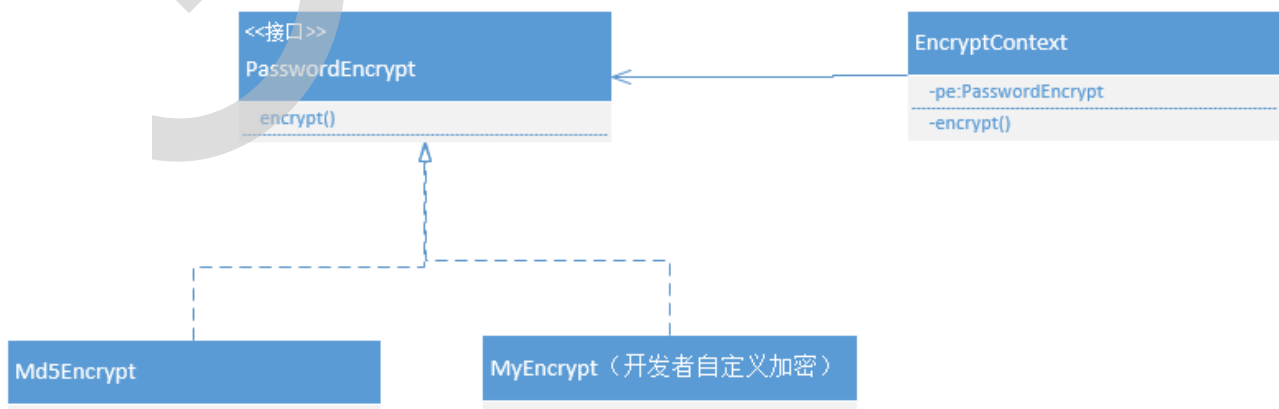
public static String get(String name,String defaultValue){
    String value = configMap.get(name);
    return value == null ? defaultValue : value;
}
}

```

在上面Config类中用到了IO流，IO流本身就是装饰器模式的一种应用。

密码加密

密码加密模式采用MD5加密，但是我们要开发一个可灵活扩展的框架，允许开发者们自定义加密方式，并且能够通过修改配置文件来修改加密方式。这里我们采用了策略模式，其类图如下：



代码如下：

PasswordEncrypt:

```
package com.design.pattern.encrypt;

public interface PasswordEncrypt {

    String encrypt(String password);

}
```

默认的MD5加密：

```
package com.design.pattern.encrypt;

import java.math.BigInteger;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

public class Md5Encrypt implements PasswordEncrypt {

    @Override
    public String encrypt(String password) {
        MessageDigest md5 = null;
        try {
            md5 = MessageDigest.getInstance("MD5");
            md5.update(password.getBytes());
        } catch (NoSuchAlgorithmException e) {
            e.printStackTrace();
        }

        return new BigInteger(1,md5.digest()).toString(16);
    }

}
```

用工厂来创建加密策略类（使用反射机制动态创建策略类）：

```
package com.design.pattern.encrypt;

public class EncryptFactory {

    /**
     * md5
     * @param clazz 类名
     * @return
     */
    public static PasswordEncrypt create(String clazz){

        try {
            Class cls = Class.forName(clazz);
            Object obj = cls.newInstance();
```

```

        if(obj instanceof PasswordEncrypt){
            return (PasswordEncrypt)obj;
        }else{
            throw new RuntimeException("class not found:" + clazz);
        }
    } catch (Exception e) {
        throw new RuntimeException("class not found:" + clazz);
    }
}
}
}

```

在EncryptContext中，根据配置文件的配置来动态选择使用哪种加密策略，默认MD5加密

```

package com.design.pattern.encrypt;

import com.design.pattern.config.Config;

public class EncryptContext {

    private PasswordEncrypt pe;

    public EncryptContext() {
        String cls = Config.get("encryptType", "com.design.pattern.encrypt.Md5Encrypt");
        this.pe = EncryptFactory.create(cls);
    }

    public String encrypt(String password){
        return this.pe.encrypt(password);
    }

}

```

现在我们先来测试一下默认的md5加密：

```

String encryptedPwd = (new EncryptContext()).encrypt("123");
System.out.println("加密后：" + encryptedPwd);

```

运行结果：

加密后：202cb962ac59075b964b07152d234b70

假设我现在是开发者，使用了这个权限框架，自定义加密逻辑，只需两步：

第一，增加加密策略类：

```
package test.encrypt;

import com.design.pattern.encrypt.PasswordEncrypt;

public class MyEncrypt implements PasswordEncrypt {

    @Override
    public String encrypt(String password) {
        return password + " encrypted pwd";
    }

}
```

第二，把该类配置到配置文件中：

```
encryptType = test.encrypt.MyEncrypt
```

然后再运行刚刚的测试代码，结果如下：

```
加密后: 123 encrypted pwd
```

Realm

realm这个概念来自于shiro框架，它是用于进行身份认证和获取用户权限。在本案例中，realm主要有两个抽象方法：

- `abstract boolean loginAuth(AuthToken token);`
- `abstract PermissionInfo doGetPermissionInfo(AuthToken token);`

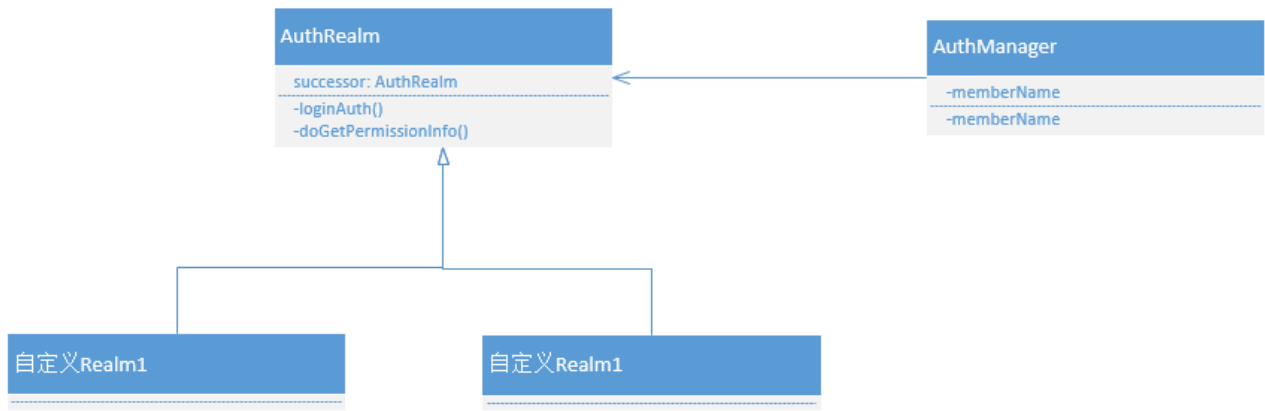
第一个方法用于判断当前用户是否认证成功，在用户登录时将调用该方法。

第二个方法是获取当前用户拥有哪些权限，在判断用户是否有某权限时调用该方法。

这两个方法都需要开发者去实现。

开发者可以自定义多个realm，比如realm1验证用户名密码，realm2用于验证第三方登录（微信登录等）。在这里我使用了责任链模式，多个realm只要有一个验证通过，那么该用户就登录成功。

类图如下：



上图中，多个自定义Realm将形成一个责任链，而形成责任链的步骤将由 AuthManager 完成，并且 AuthManager 类是一个单例模式

AuthRealm类：

```
package com.design.pattern.auth;

public abstract class AuthRealm {

    private AuthRealm successor;

    public void setSuccessor(AuthRealm realm){
        this.successor = realm;
    }

    public final boolean auth(AuthToken token){

        if(token == null) return false;
        //如果验证成功，就返回成功
        if(this.loginAuth(token)){
            return true;
        }
        //失败就将请求传给下一个责任处理器
        return successor != null && successor.auth(token);
    }

    /**
     * 登录验证
     * @return
     */
    protected abstract boolean loginAuth(AuthToken token);

    /**
     * 权限验证
     * @return
     */
    protected abstract PermissionInfo doGetPermissionInfo(AuthToken token);
}
```

在上面类中，使用到了 `AuthToken` 和 `PermissionInfo`，前者是用户认证信息，存放用户名密码等，后者是保存权限信息，包括“角色”和“权限”。

AuthToken类：

```
package com.design.pattern.auth;

import java.util.List;

public class AuthToken {

    private String username;
    private String password;

    public AuthToken() {
    }

    public AuthToken(String username, String password) {
        this.username = username;
        this.password = password;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

}
```

PermissionInfo 类：

```
package com.design.pattern.auth;

import java.util.Set;

public class PermissionInfo {

    private Set<String> permissions;
    private Set<String> roles;
```



```

    public Set<String> getPermissions() {
        return permissions;
    }

    public void setPermissions(Set<String> permissions) {
        this.permissions = permissions;
    }

    public Set<String> getRoles() {
        return roles;
    }

    public void setRoles(Set<String> roles) {
        this.roles = roles;
    }

    /**
     * 判断是否有某权限
     * @param permission
     * @return
     */
    public boolean isPermitted(String permission){
        return this.permissions.contains(permission);
    }
}

```

AuthManager类：

```

package com.design.pattern.auth;

import com.design.pattern.config.Config;

import java.util.ArrayList;
import java.util.List;

public class AuthManager {

    private List<AuthRealm> list = new ArrayList<>();

    private static AuthManager instance = new AuthManager();

    /**
     * 私有构造方法，读取配置文件，通过反射机制生成Realm，并构建责任链
     */
    private AuthManager() {

        String realms = Config.get("realms");

        if(realms == null || realms.isEmpty()){

            throw new RuntimeException("请定义Realm");
        }
    }
}

```

```

    }

    String[] clss = realms.split(",");
    for (int i = 0; i < clss.length; i++){
        try {
            Object obj = Class.forName(clss[i]).newInstance();
            if(obj instanceof AuthRealm){
                this.list.add((AuthRealm)obj);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    //形成责任链
    for (int i=0;i<list.size()-1;i++){
        AuthRealm next = list.get(i+1);
        if(next != null){
            list.get(i).setSuccessor(next);
        }
    }
}

/**
 * 调用realm中的doGetPermissionInfo方法，如果有多个realm，只调用第一个
 * @param token
 * @return
 */
public PermissionInfo getPermissionInfo(AuthToken token){
    if(token == null){
        return null;
    }
    if(list.size() > 0){
        return this.list.get(0).doGetPermissionInfo(token);
    }
    return null;
}

/**
 * 登录认证，调用realm责任链
 * @return
 */
public boolean auth(AuthToken token){
    if(list.size() == 0){
        return false;
    }
    return list.get(0).auth(token);
}

/**
 * 单例
 * @return
 */
public static AuthManager getInstance(){

    return instance;
}

```

```
}  
  
}
```

AuthManager主要有以下职责：

- 生成Realm责任链
- 调用身份认证和权限认证方法

到这里，这个迷你权限框架已经完成了百分之90，还缺一个用户主体类Subject。

所谓的用户主体，有点类似web开发中的session，一个用户请求对应一个session，而在权限框架中，用户主体类Subject就代表了当前用户。

Auth接口，定义了登录，权限等方法：

```
package com.design.pattern.subject;  
  
import com.design.pattern.auth.AuthToken;  
  
public interface Auth {  
  
    /**  
     * 登录操作  
     * @param token  
     * @return  
     */  
    boolean login(AuthToken token);  
  
    /**  
     * 是否已登录  
     * @return  
     */  
    boolean isLogin();  
  
    /**  
     * 是否有权限  
     * @param permission  
     * @return  
     */  
    boolean isPermitted(String permission);  
}
```

用户主体类Subject：

```
package com.design.pattern.subject;  
  
import com.design.pattern.auth.AuthToken;  
import com.design.pattern.auth.PermissionInfo;  
  
import com.design.pattern.auth.AuthManager;
```

```

import com.design.pattern.encrypt.EncryptContext;

/**
 * 登录用户主体
 */
public class Subject implements Auth{

    private AuthToken token;

    @Override
    public boolean login(AuthToken token) {
        //调用密码加密策略
        String password = (new EncryptContext()).encrypt(token.getPassword());
        token.setPassword(password);
        //调用auth方法，即触发责任链
        if(AuthManager.getInstance().auth(token)){
            System.out.println("登录成功");
            this.token = token;
            return true;
        }
        return false;
    }

    @Override
    public boolean isLogin() {
        return token != null;
    }

    @Override
    public boolean isPermitted(String permission) {
        PermissionInfo info = AuthManager.getInstance().getPermissionInfo(this.token);
        return info != null && info.isPermitted(permission);
    }

    public String getUsername(){
        return token.getUsername();
    }
}

```

工具类SecurityUtils，提供了全局获取用户主体类Subject的方法：

```

package com.design.pattern.util;

import com.design.pattern.subject.Subject;

import java.util.HashMap;
import java.util.Map;

public class SecurityUtils {

    private static Map<String, Subject> subjectList = new HashMap<>();
}

```

```

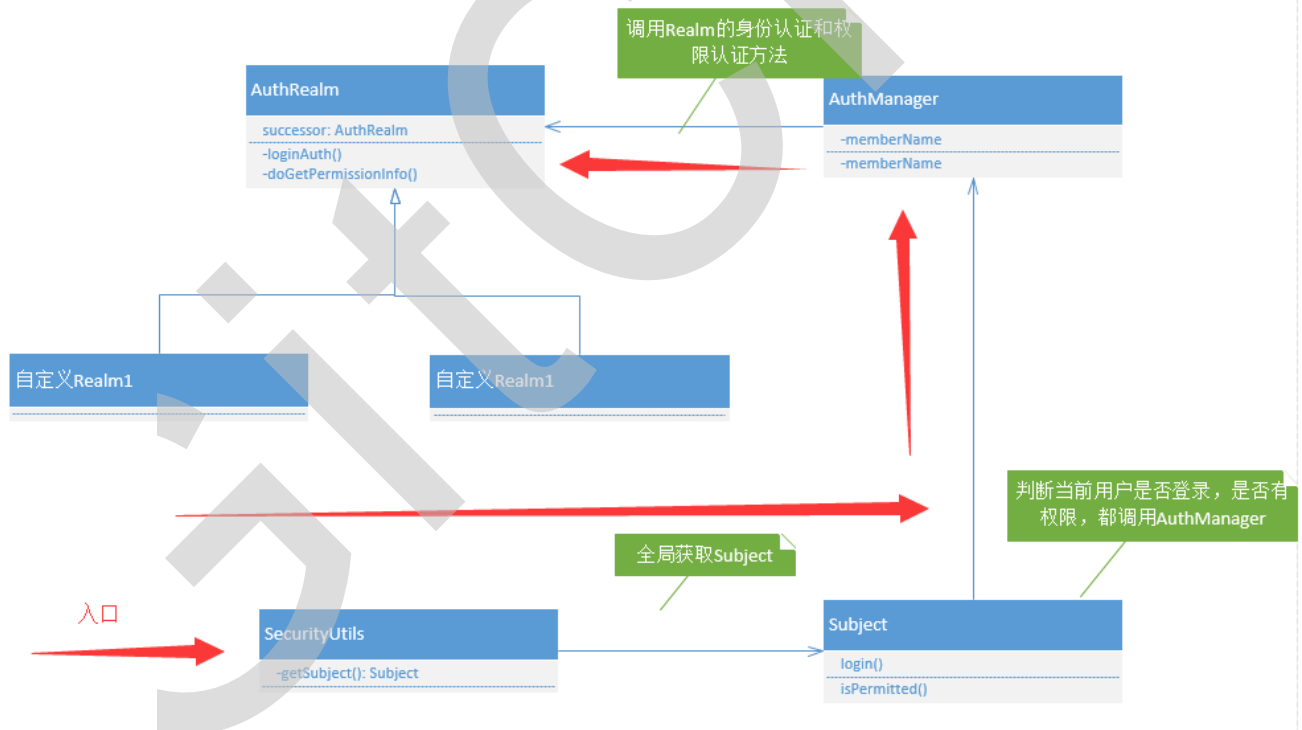
/**
 * 获取当前请求的用户
 * @return
 */
public static Subject getSubject(){
    //此处应借用session等方式获取当前请求用户
    String name = "123";
    Subject subject = subjectList.get(name);
    return subject == null ? new Subject() : subject;
}

public static void addSubject(Subject subject){
    subjectList.put("123",subject);
}
}

```

实际上上面 `SecurityUtils` 中的 `getSubject()` 的实现机制也应该是一个类似session的机制，就像我们在web请求中获取当前session，session和当前用户对应。但本次案例主要是介绍设计模式，就不去实现那么复杂的功能了，因此这里就简单地直接给出Subject了。

看了这么多代码，可能会有点懵，我们来看一下整体的结构图：



上图中的红色箭头就是整个权限框架的流程，至此，整个框架开发完成。

现在我们来简单测试一下，首先我们需要创建Realm1：

```

package test.realm;

import com.design.pattern.auth.AuthRealm;
import com.design.pattern.auth.AuthToken;

```

```

import com.design.pattern.auth.PermissionInfo;

import java.util.HashSet;
import java.util.Set;

public class Realm1 extends AuthRealm {

    @Override
    protected boolean loginAuth(AuthToken token) {
        System.out.println("===Realm1 loginAuth===");
        String username = token.getUsername();
        String pwd = token.getPassword();
        //传进来的密码是加密过的密码，直接和数据库中的密码比对
        System.out.println("pwd:"+pwd);
        //查询数据库操作略过
        return false;
    }

    @Override
    protected PermissionInfo doGetPermissionInfo(AuthToken token) {

        String username = token.getUsername();
        System.out.println("doGetPermissionInfo1");
        //从数据库读取该用户的权限信息
        PermissionInfo info = new PermissionInfo();
        Set<String> s = new HashSet<String>();
        s.add("permission1");
        s.add("permission2");
        info.setPermissions(s);
        //角色
        Set<String> r = new HashSet<String>();
        r.add("role1");
        info.setRoles(r);
        return info;
    }
}

```

再创建一个Realm2，Realm2的和1结构是一样的，具体的业务逻辑要根据你项目实际情况去修改，这里只是测试，就直接给出一模一样的代码：

```

package test.realm;

import com.design.pattern.auth.AuthRealm;
import com.design.pattern.auth.AuthToken;
import com.design.pattern.auth.PermissionInfo;

import java.util.HashSet;
import java.util.Set;

```

```

public class Realm2 extends AuthRealm {

    @Override
    protected boolean loginAuth(AuthToken token) {
        System.out.println("===Realm2 loginAuth===");
        String username = token.getUsername();
        String pwd = token.getPassword();
        //传进来的密码是加密过的密码，直接和数据库中的密码比对
        System.out.println("pwd:"+pwd);
        //查询数据库操作略过
        return true;
    }

    @Override
    protected PermissionInfo doGetPermissionInfo(AuthToken token) {
        String username = token.getUsername();
        System.out.println("doGetPermissionInfo2");
        //从数据库读取该用户的权限信息
        PermissionInfo info = new PermissionInfo();
        Set<String> s = new HashSet<String>();
        s.add("printer:print");
        s.add("printer:query");
        info.setPermissions(s);

        //角色
        Set<String> r = new HashSet<String>();
        r.add("role1");

        info.setRoles(r);
        return info;
    }
}

```

然后将两个Realm配置到配置文件中，多个Realm用逗号隔开：

```

encryptType = test.encrypt.MyEncrypt
realms=test.realm.Realm1,test.realm.Realm2

```

encryptType 是配置密码加密方式，前面已经讲过了。

测试代码：

```

package test;

import com.design.pattern.auth.AuthToken;
import com.design.pattern.encrypt.EncryptContext;
import com.design.pattern.subject.Subject;
import com.design.pattern.util.SecurityUtils;

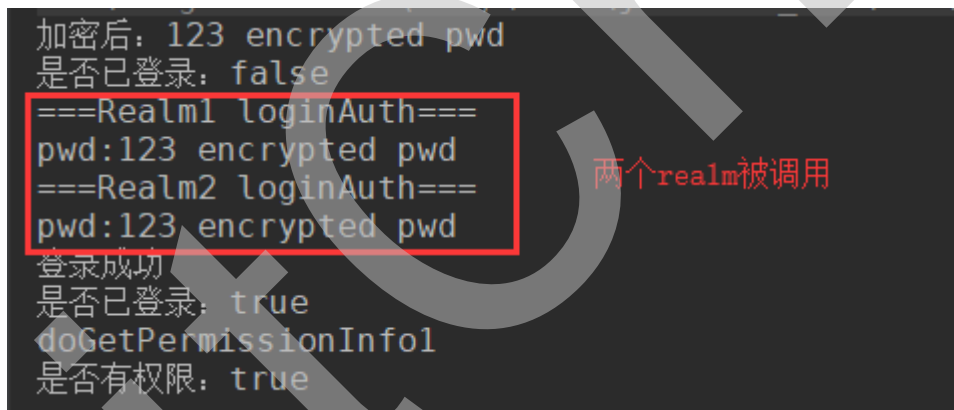
```

```
import java.io.IOException;

public class TestDemo {

    public static void main(String[] args) throws IOException{
        //测试密码加密
        String encryptedPwd = (new EncryptContext()).encrypt("123");
        System.out.println("加密后：" + encryptedPwd);
        //获取当前用户
        Subject currentUser = SecurityUtils.getSubject();
        //是否登录
        System.out.println("是否已登录：" + currentUser.isLogin());
        //执行登录操作
        currentUser.login(new AuthToken("admin", "123"));
        //是否登录
        System.out.println("是否已登录：" + currentUser.isLogin());
        //是否有权限，权限用字符串表示
        System.out.println("是否有权限：" + currentUser.isPermitted("permission1"));
    }
}
```

执行结果：



```
加密后: 123 encrypted pwd
是否已登录: false
===Realm1 loginAuth===
pwd:123 encrypted pwd
===Realm2 loginAuth===
pwd:123 encrypted pwd
登录成功
是否已登录: true
doGetPermissionInfo1
是否有权限: true
```

两个realm被调用

从上图结果中可以看到整个框架运行的流程，执行登录操作时，2个Realm的 `loginAuth()` 方法被调用，判断是否有某权限时，`doGetPermissionInfo()` 方法被调用。

注意：本次案例仅用于学习设计模式，不可用于实际项目开发

源码下载：https://gitee.com/itzhoujun/design_pattern_learning_demo

鸣谢与总结

感谢购买本课程的朋友。

本课程一共11节课，包含10种最常用的设计模式，学习完这10种设计模式，相信大家都对设计模式有了不小的好感，并且基本掌握了设计模式的学习方法与使用方法。

在这里笔者借用古人的两句话来总结一下如何学习设计模式：

- 书读百遍其义自见
- 好记性不如烂笔头

笔者最初学习设计模式时，也买了两本设计模式的书，看是看懂了，但是真正使用起来还是很吃力，感觉无处下手，后来又不停地在网上找例子，每一个设计模式都找了好些个例子一个个去研究，去读，读的多了，慢慢的就有感觉了，有点像学英语培养语感。包括写本课程的时候，笔者也同样看了不少的例子。

但仅仅看是没用的，看10遍真的不如自己写一遍。在这里笔者给大家一个建议：找一个知名的开源框架，了解其工作流程后，尝试着用设计模式自己去写一个简单的例子，例如写个SpringMVC。就像本节课的权限框架的例子，就是仿照shiro框架写的。

其实本次的例子笔者自己不是太满意，第一，用的设计模式不够多，但是笔者没有太多精力去写一个完善的框架出来。第二，案例中有一些细节处理不到位，不过不影响对设计模式的学习。

希望本课程能够帮助到大家，共同学习，共同进步。

再次感谢大家购买课程，谢谢。