# Shiro框架

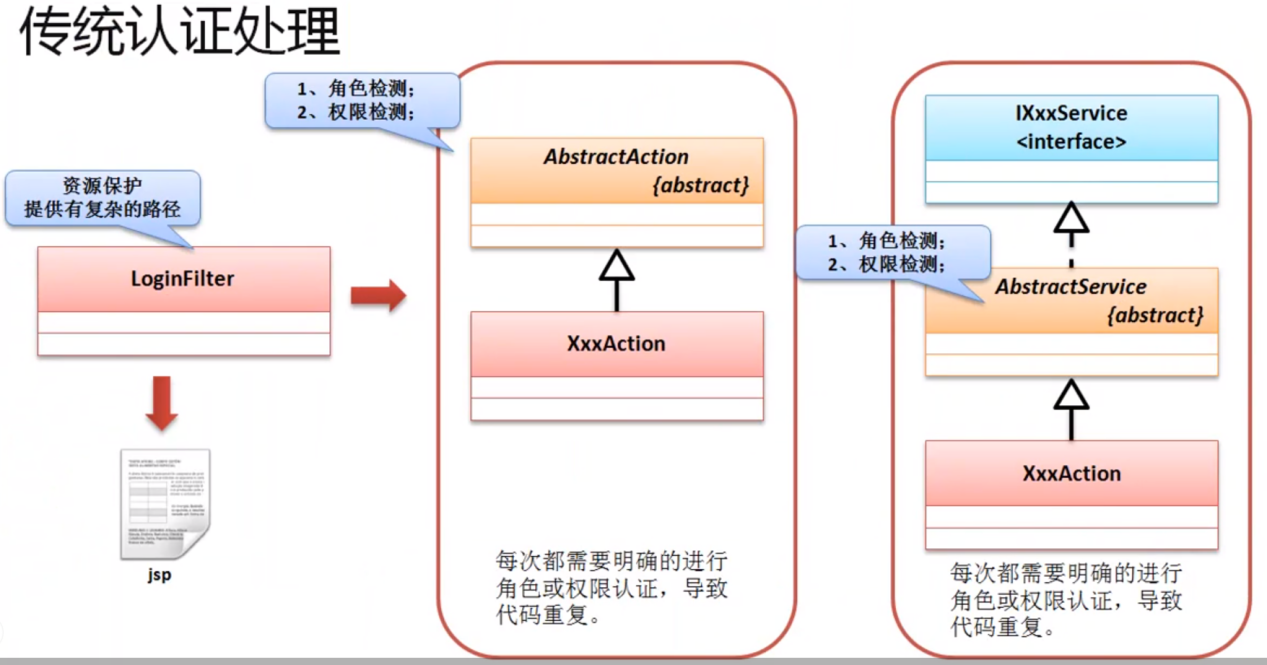
Shiro是专门的权限以及认证授权的处理验证框架，用于用户登录检测以及授权认证处理操作的开发框架。Shiro是由Apache提供的开源项目。

## 权限认证问题

### 认证问题分析

1.1传统的认证模式

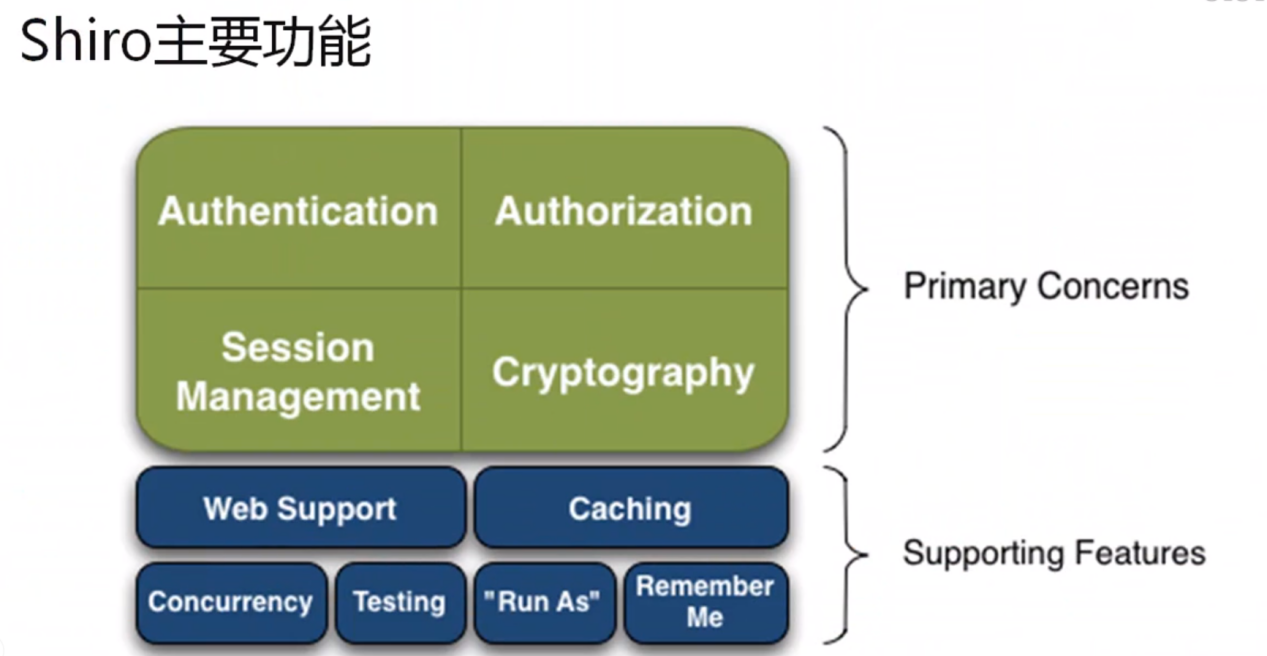
* 用户登录的时候需要验证用户名和密码
* 用户登录之后才表示可以进行过滤器的检测，过滤器可以保证未登录过得用户无法进行资源访问。
* 登录之后可以进行各种操作，但是不同的用户会存在有不同的角色、权限；此时为了保证安全性，需要在控制层进行一个当前用户的权限认证。
* 在开发中业务层的操作是有可能单独独立出去，成为一个业务接口，这个业务接口可以进行Webservice发布。此时会出现一个问题：业务层有可能单独对外，那么同样需要进行角色或者是权限检测；

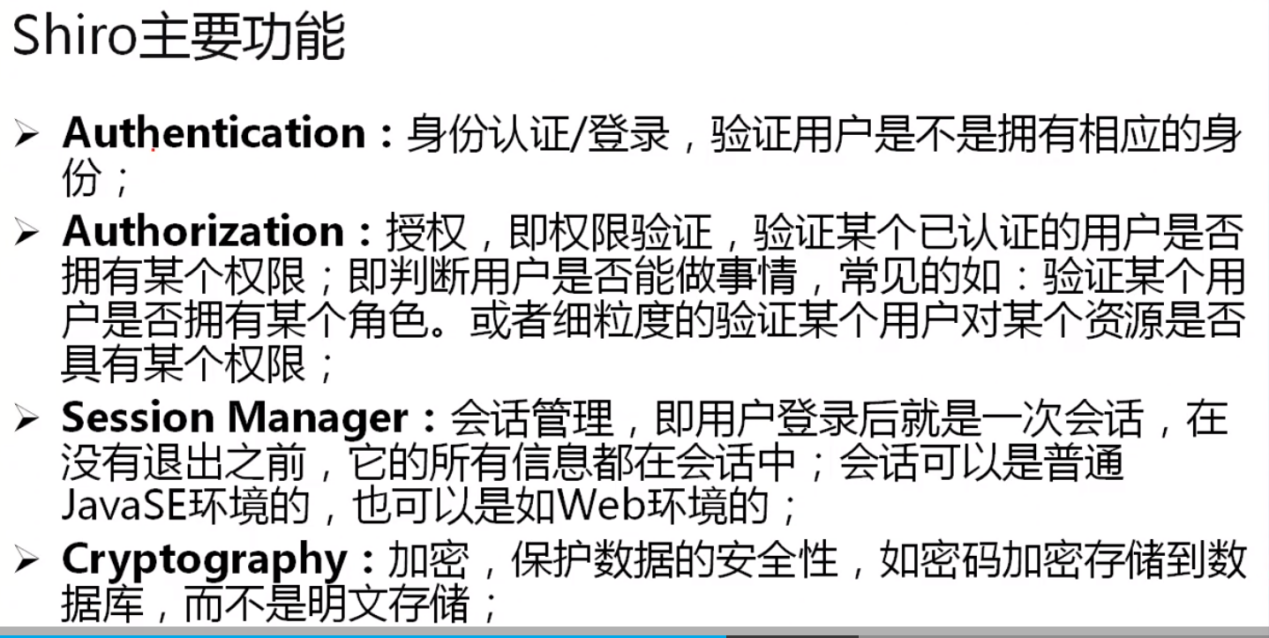


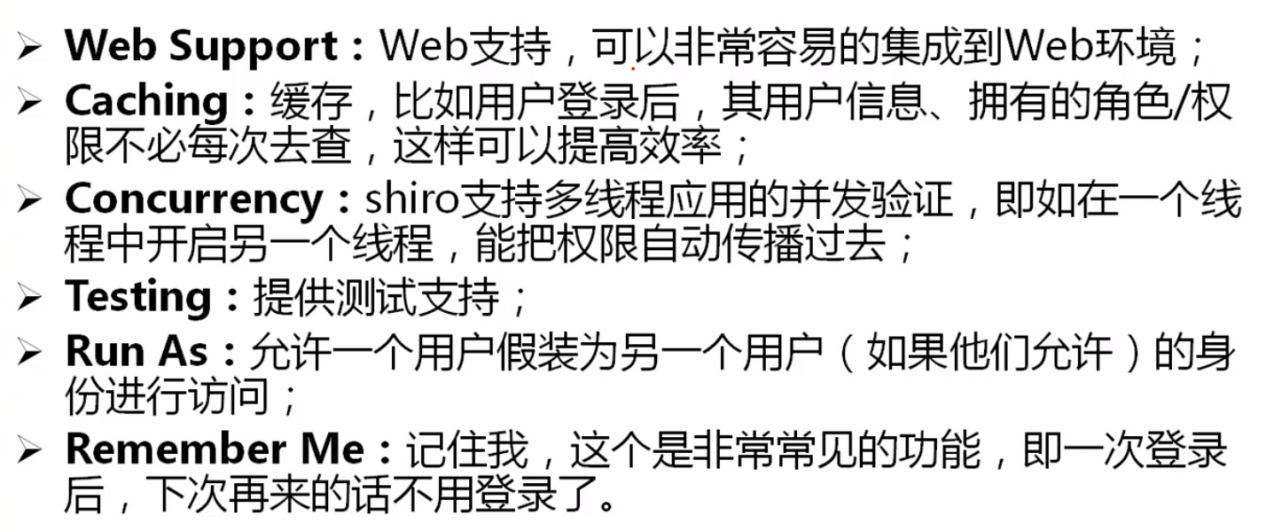
实际开发就会发现，几乎所有的项目都会存在有同样的流程，那么狠明显如果每一次都进行重复的开发，那这个过程实在是痛苦嗦。以此进行项目编写，强烈不建议使用固定的编码格式，在这样的前提下，行业内，产生了两个流行的认证开发框架：Spring Security、Shiro。Shiro凭借自己的优势以及配置简单，慢慢开始流行起来，已经成为从业的标准，而Shiro往往会与SpringMVC和mybatis结合实现更为方便的用户认证和授权操作。

### Shiro架构

Shiro是由Apache推出的一款安全框架，小巧而功能完善。直接利用maven就可以取得开发包。



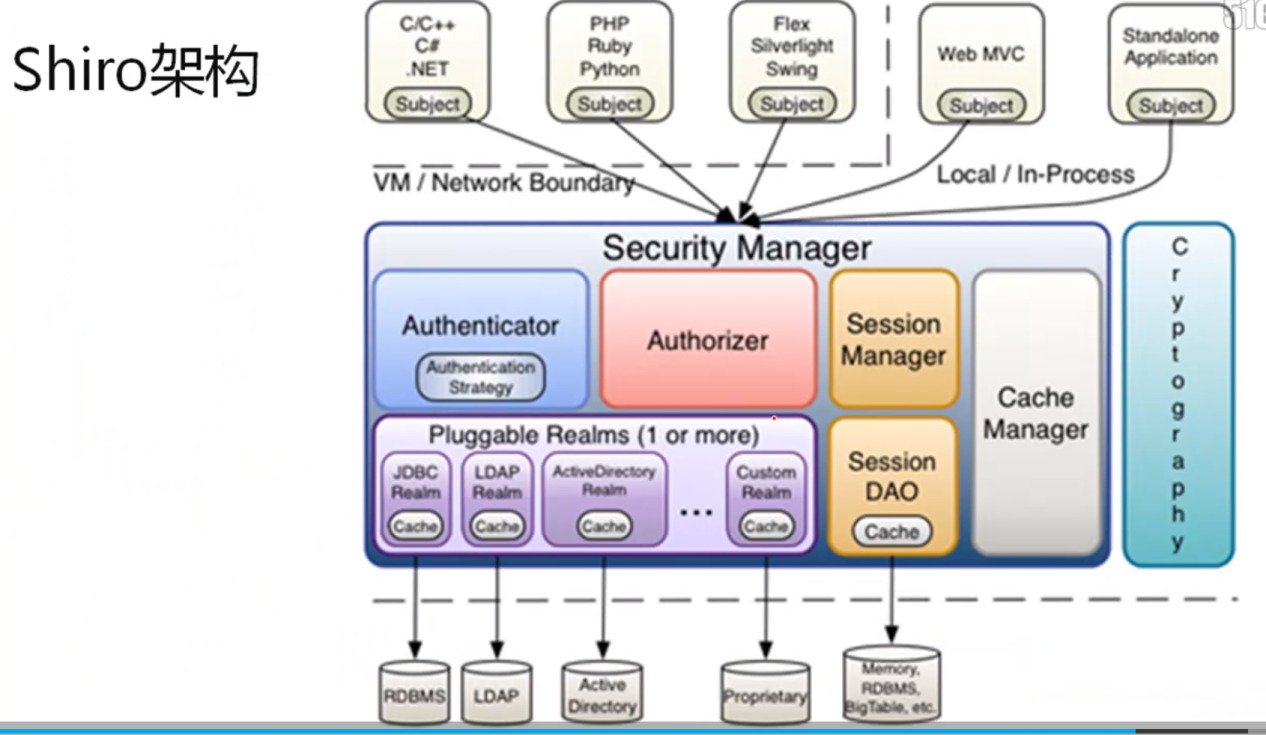








Shiro还有一个最大的好处是可以和springmvc无缝结合。



在整个shiro中，关键的操作步奏依然需要用户开发，Shiro只是一个提供用户登录，授权认证的处理开发框架，可以对所有的项目进行统一的授权处理操作。

## Shiro基础开发

### 2.1 搭建项目开发环境（maven）

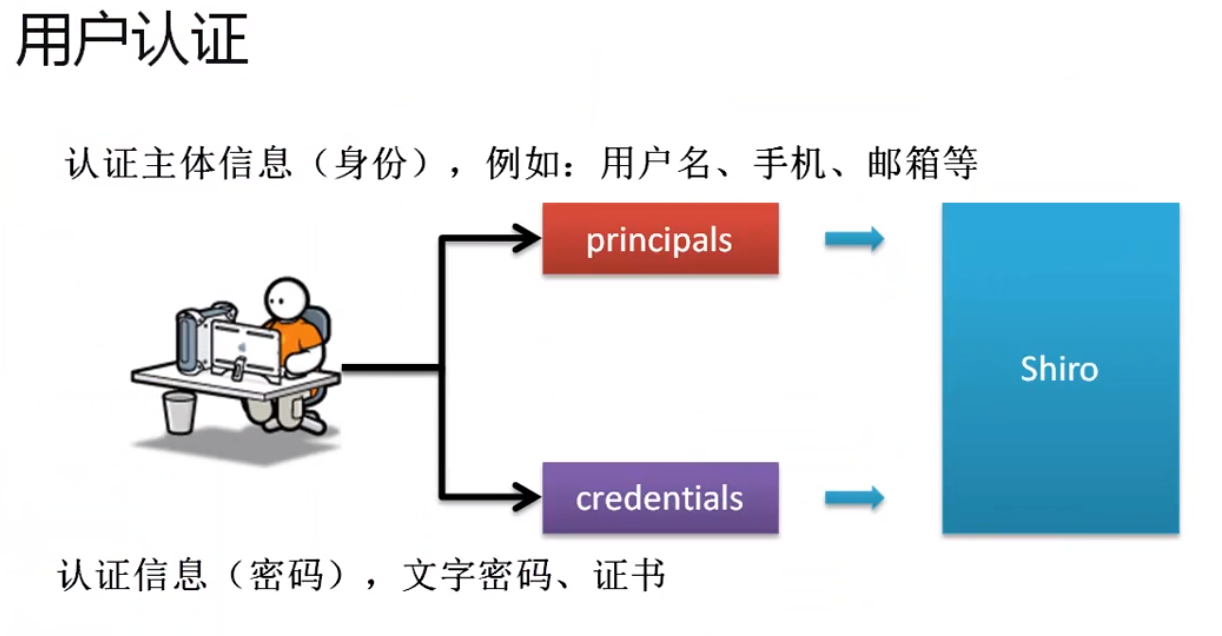
1、添加项目开发包，

<**dependency**>  
 <**groupId**>javax.servlet</**groupId**>  
 <**artifactId**>javax.servlet-api</**artifactId**>  
 <**version**>4.0.1</**version**>  
 <**scope**>provided</**scope**>  
</**dependency**>  
<**dependency**>  
 <**groupId**>org.apache.shiro</**groupId**>  
 <**artifactId**>shiro-core</**artifactId**>  
 <**version**>${shiro.version}</**version**>  
</**dependency**>  
*<!-- 日志文件管理包 -->  
<!-- log start -->*<**dependency**>  
 <**groupId**>log4j</**groupId**>  
 <**artifactId**>log4j</**artifactId**>  
 <**version**>${log4j.version}</**version**>  
</**dependency**>  
<**dependency**>  
 <**groupId**>org.slf4j</**groupId**>  
 <**artifactId**>slf4j-api</**artifactId**>  
 <**version**>${slf4j.version}</**version**>  
</**dependency**>  
<**dependency**>  
 <**groupId**>org.slf4j</**groupId**>  
 <**artifactId**>slf4j-log4j12</**artifactId**>  
 <**version**>${slf4j.version}</**version**>  
</**dependency**>  
*<!-- log end -->*

2、将log4j.properties拷贝到项目中。

### 2.2 固定信息认证

本次的信息认证操作内容都将保存在资源文件里面，假设资源文件名称为shiro.ini。保存用户名和密码。



1. 在资源路径下建立shiro.ini文件：
2. 修改文件内容，本次追加的是用户信息：

[users]  
admin=hello  
ym=aini

本次定义了两个用户admin=hello,yangmi=aini,等号前的为认证身份信息，后面为认证信息。在shiro里面针对于身份和认证信息的错误提示也是不同的。

1. 进行Shiro的编写：本次的操作一定是通过shiro.ini文件取得认证信息；

* 要实现认证信息的读取，那么首先要使用一个org.apache.shiro.util.Factory接口，这个接口定义有取得一个SecurityManager接口对象的方法：T getInstance();

|- 此方法可以取得一个org.apache.shiro.mgt.SecurityManager接口对象

|-Factory是一个接口，本次将通过ini文件读取，需要使用“org.apache.shiro.config.IniSecurityManagerFactory”子类来实例化接口对象。此类中只需关注一个构造方法：public IniSecurityManagerFactory(String iniResourcePath)，通过CLASSPATH取得。

* Factory中保管的是一个SecurityManager接口对象，这个接口对象主要是进行所有认证信息的处理的，也就是说读取进来的资源文件信息在整个接口里面将转换为所有的认证数据（用户名和密码）；
* 具体的认证信息一定要通过Subject来实现，而securityManager只是一个综合的认证。为了解决这个问题，可以使用“”org.apache.shiro.SecurityUtils类，其定义有两个方法：

1. 设置SecurityManager接口对象：public static void setSecurityManager(SecurityManager securityManager)；
2. 取得一个要进行认证的Subject接口： public static Subject getSubject()；

* 本次使用的是使用用户名和密码的组合进行的认证处理，所以要将用户名和密码包装为一个Token，在subject接口下定义有如下两个方法：

1. 登录验证：void login(AuthenticationToken var1) throws AuthenticationException;
2. 登录注销：void logout();

* 既然要进行简单的认证，那么就只需要输入用户名和密码即可，所以使用“org.apache.shiro.authc.UsernamePasswordToken”。那么只需要创建这个类的对象，同时创建的时候传递用户名和密码即可实现。
* 使用SUbject实现用户的登录处理操作。

以上步奏实现如下：

*// 取得Factory接口对象，主要的目的是通过配置文件加载文件之中的信息，这些信息暂时不能够成为认证信息*Factory<SecurityManager> factory = **new** IniSecurityManagerFactory(**"classpath:shiro.ini"**);  
*// 取得里面所保存的所有的认证数据信息*SecurityManager securityManager = factory.getInstance();  
*// 利用一个专门的认证操作的处理类，实现认证处理的具体的实现*SecurityUtils.*setSecurityManager*(securityManager);  
*// 获取进行用户名和密码认证的接口对象*Subject subject = SecurityUtils.*getSubject*();  
*// 定义了一个Token，里面保存要登录的用户名和密码信息(传递用户信息)*UsernamePasswordToken token = **new** UsernamePasswordToken(**"ym"**, **"aini"**);  
*// 实现用户登录处理*subject.login(token);  
*// 取得用户名*System.***out***.println(subject.getPrincipal());

在保证用户名和密码正确的前提下，就可以轻松的登录，否则将会抛出异常，异常形式有好多种：

1. 密码错误，但是用户名输入正确：

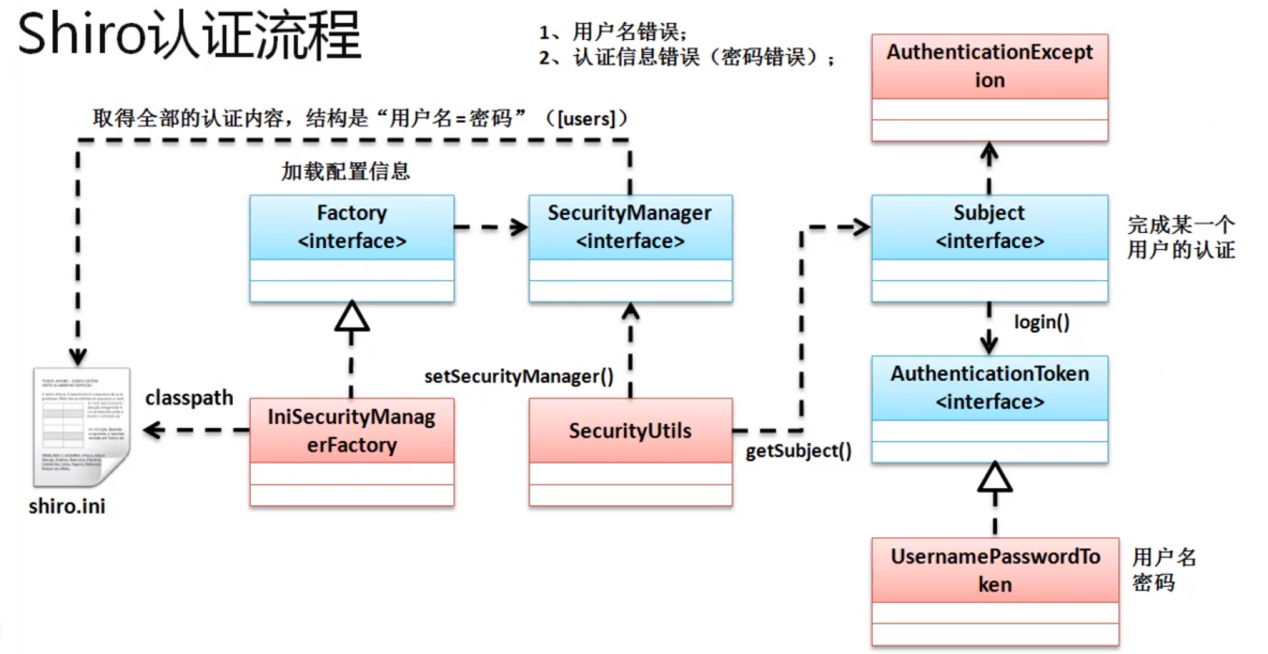
org.apache.shiro.authc.IncorrectCredentialsException: Submitted credentials for token [org.apache.shiro.authc.UsernamePasswordToken - ym, rememberMe=false] did not match the expected credentials.

1. 用户名正确，密码错误：

org.apache.shiro.authc.UnknownAccountException: Realm [org.apache.shiro.realm.text.IniRealm@3d04a311] was unable to find account data for the submitted AuthenticationToken [org.apache.shiro.authc.UsernamePasswordToken - ymm, rememberMe=false].

在Shiro里面如果用户名错误，那么将不会再进行密码验证；

### 2.3 认证流程分析



在整个程序里面，必须要进行验证数据的信息加载，而后但所有的认证信息交由SecurityManager管理，要想进行用户名和密码的认证操作，必须通过subject接口完成。而后使用AuthenticationToken保存所有的认证信息，本处为普通的用户名环绕密码，所以使用了UserNamePasswordToken进行配置。

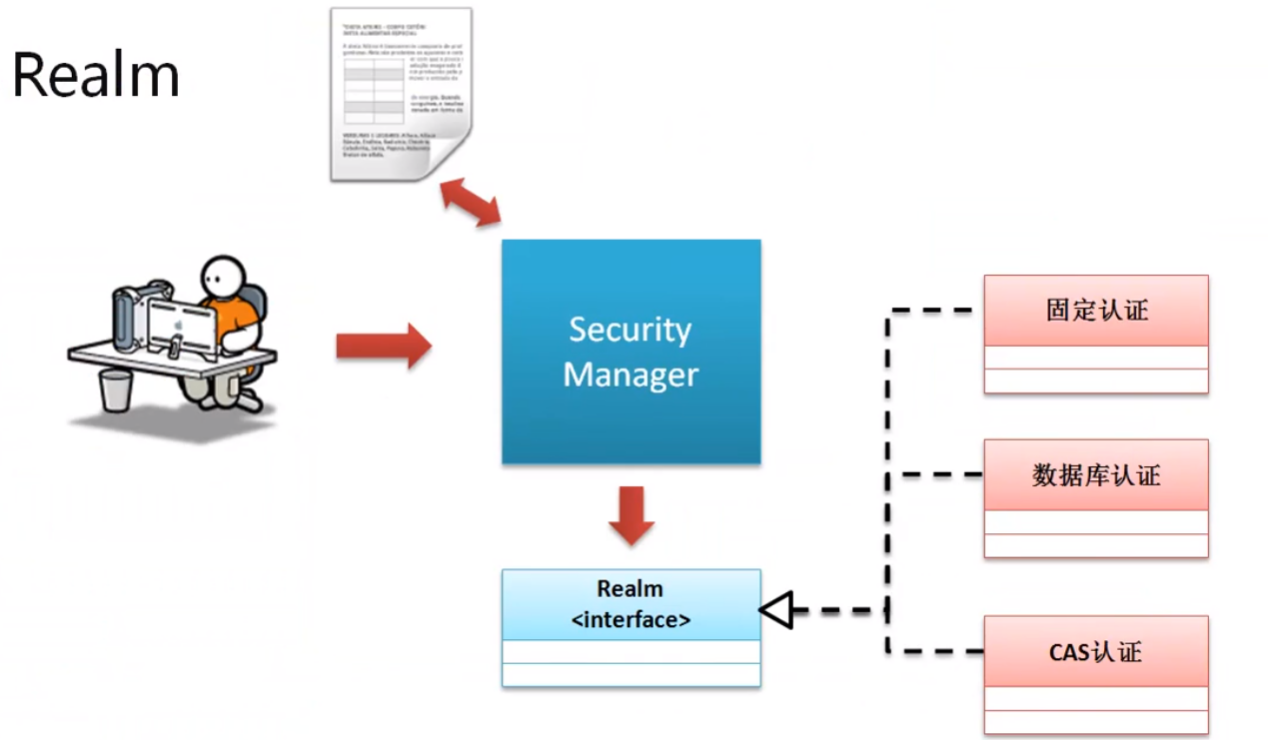
如果登录成功不会抛出异常，否则根据错误类型抛出异常。

## Realm

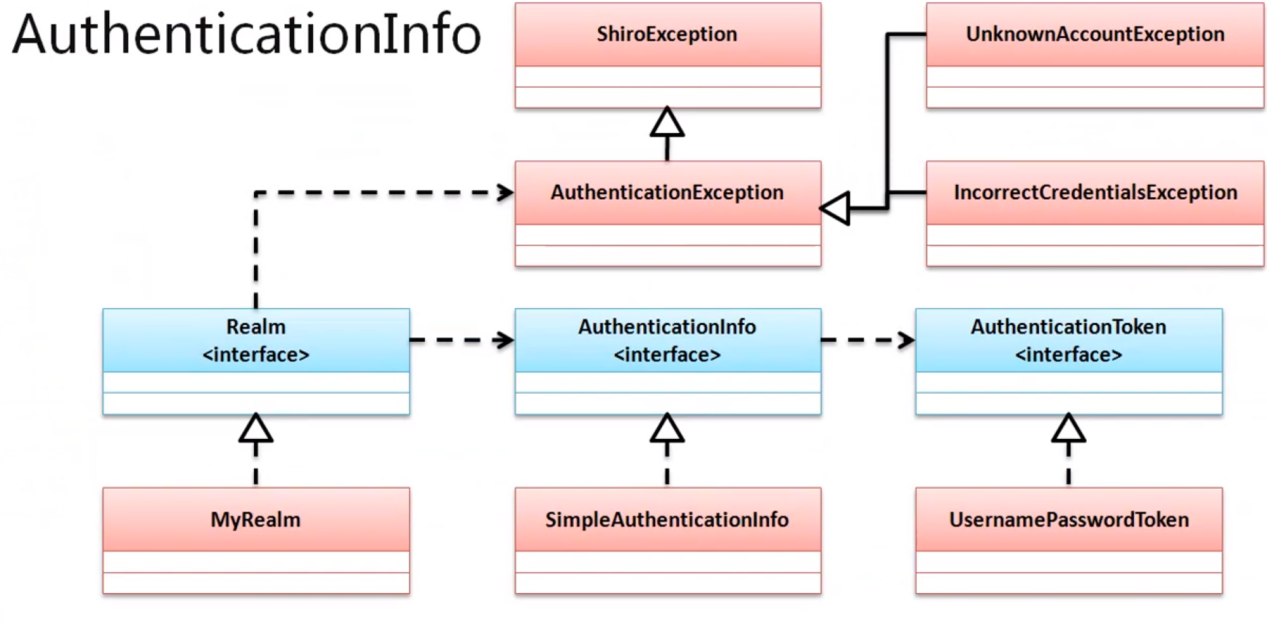
### Realm简介

1. 认识Realm

考虑到不同用户数据来源，为统一这些来源的处理准备了Realm接口“org.apache.shiro.realm.Realm”



### 固定信息认证



1. 自定义个Realm实现类：

**public class** MyRealm **implements** Realm {  
 */\*\*  
 \* 名字随便给一个，只要能唯一标记即可  
 \*  
 \** ***@return*** *\*/* @Override  
 **public** String getName() {  
 **return "yangmi"**;  
 }  
  
 @Override  
 **public boolean** supports(AuthenticationToken authenticationToken) {  
 **return** authenticationToken **instanceof** UsernamePasswordToken;  
 }  
  
 @Override  
 **public** AuthenticationInfo getAuthenticationInfo(AuthenticationToken authenticationToken) **throws** AuthenticationException {  
 String userName = (String) authenticationToken.getPrincipal();  
 *// 在Shiro里面是利用字符数组实现了密码的传递，所以不能将其直接变为String* String password = **new** String((**char**[]) authenticationToken.getCredentials());  
 *// 此时直接使用一个固定的用户名和密码进行验证处理操作* **if** (!**"yangmi"**.equals(userName)) { *// 判断用户名是否存在* **throw new** UnknownAccountException(**"用户名不存在！"**);  
 }  
 **if** (!**"aini"**.equals(password)) {  
 **throw new** IncorrectCredentialsException(**"密码输入错误！"**);  
 }  
 **return new** SimpleAuthenticationInfo(userName, password, **this**.getName());  
 }  
}

1. 修改shiro.ini配置文件：

yangmi=cn.ym.realm.MyRealm  
yangmi2=cn.ym.realm.MyRealm2  
# 整个Shiro中的验证处理都是由SecurityManager接口负责的  
securityManager.realms=$yangmi,$yangmi2

### 基于数据库的认证

1. 创建数据库脚本

DROP DATABASE IF EXISTS shirodb ;

CREATE DATABASE shirodb CHARACTER SET UTF8 ;

USE shirodb ;

CREATE TABLE member (

mid VARCHAR(50),

password VARCHAR(32),

name VARCHAR(20),

locked INT,

CONSTRAINT pk\_mid PRIMARY KEY (mid)

) engine= innodb;

INSERT INTO member(mid,password,name,locked) VALUES ('admin','hello','管理员',0) ;

INSERT INTO member(mid,password,name,locked) VALUES (‘yangmi,ym’,'杨幂',0) ;

1. 使用mysql提供的数据源,修改shiro.ini文件

[main]  
# 定义本次MySQL连接要使用的数据源处理类  
dataSource=com.mysql.jdbc.jdbc2.optional.MysqlDataSource  
# 定义数据库的连接主机名称  
dataSource.serverName=127.0.0.1  
# 定义要使用的数据库的名字  
dataSource.databaseName=shirodb  
# 定义数据库的连接账户  
dataSource.user=root  
# 定义数据库的连接的密码  
dataSource.password=root  
  
# 定义本次要基于JDBC实现的Realm的认证的配置类  
jdbcRealm=org.apache.shiro.realm.jdbc.JdbcRealm  
# 配置本次的JDBC连接所使用的数据源，$表示引用配置  
jdbcRealm.dataSource=$dataSource  
# 定义所需要的查询语句  
jdbcRealm.authenticationQuery=SELECT password FROM member WHERE mid=?  
  
# 配置安全管理器所使用的Realm  
securityManager.realms=$jdbcRealm

在固定的基础上，只修改上面的配置文件，其他不动。

## 认证授权

授权也称为访问控制，即一个资源库中的资源允许那些用户来处理操作。

### 4.1 固定角色

利用配置文件操作实现，该实现要求在“[user]”下进行，形式为：“用户名=密码，角色，角色...”。

1. 定义一个shiro.ini文件：

[users]  
yangmi=ym,update,delete

1. 角色检测，在shiro中subject接口描述的是用户主体信息。所有针对于角色的认证处理都在此接口中提供，包括如下方法：

|  |  |  |  |
| --- | --- | --- | --- |
| NO | 方法名称 | 类型 | 描述 |
| 1 | void checkRole(String var1) throws AuthorizationException; | 普通 | 检测角色是否存在 |
| 2 | void checkRoles(String... var1) throws AuthorizationException; | 普通 | 检测多个角色是否存在 |
| 3 | boolean hasRole(String var1); | 普通 | 是否有指定的角色 |
| 4 | boolean hasAllRoles(Collection<String> var1); | 普通 | 是否有所有角色 |

返回值为void是如果没有权限则抛出对应异常。

### 4.2 权限认证

现在可以取得角色了，那么也就可以取得对应的权限，在shiro.ini中配置角色—权限对应关系：

用户和角色：在[users]下配置，用户=密码,角色,角色,...

角色和权限：在[roles]下配置,角色=权限,权限,...（\*表示具备所有权限）

对权限的检查Subject定义有如下方法：

|  |  |  |  |
| --- | --- | --- | --- |
| NO | 方法名称 | 类型 | 描述 |
| 1 | void checkPermission(String var1) throws AuthorizationException; | 普通 | 指定角色是否存在，没有则抛出异常 |
| 2 | void checkPermissions(Collection<Permission> var1) throws AuthorizationException; | 普通 | 判断指定的所有权限是否存在 |
| 3 | boolean isPermitted(String var1); | 普通 | 是否有指定的权限存在 |
| 4 | boolean isPermittedAll(String... var1); |  | 是有指定的多个权限存在 |

### 4.3 数据库认证

1、创建数据库表结构：

CREATE TABLE member (

mid VARCHAR(50),

password VARCHAR(32),

name VARCHAR(20),

locked INT,

CONSTRAINT pk\_mid PRIMARY KEY (mid)

) engine= innodb;

-- 2、角色数据表

CREATE TABLE role (

rid INT AUTO\_INCREMENT ,

title VARCHAR(50),

flag VARCHAR(50),

CONSTRAINT pk\_rid PRIMARY KEY (rid)

) engine= innodb;

-- 3、用户-角色关系表

CREATE TABLE member\_role (

mid VARCHAR(50),

rid INT ,

CONSTRAINT fk\_mid1 FOREIGN KEY(mid) REFERENCES member(mid) ,

CONSTRAINT fk\_rid1 FOREIGN KEY(rid) REFERENCES role(rid)

) engine= innodb;

-- 4、权限表

CREATE TABLE action (

actid INT AUTO\_INCREMENT ,

title VARCHAR(50),

flag VARCHAR(50),

CONSTRAINT pk\_actid PRIMARY KEY (actid)

) engine= innodb;

-- 5、角色-权限关系表

CREATE TABLE role\_action (

rid INT,

actid INT,

CONSTRAINT fk\_rid6 FOREIGN KEY(rid) REFERENCES role(rid) ,

CONSTRAINT fk\_actid6 FOREIGN KEY(actid) REFERENCES action(actid)

) engine= innodb;

1. 定义一个Member类：

...

1. 创建一个单独的类获取用户信息，以及角色、权限信息

**public class** MemberLoginService {  
 **private** Connection **conn**; *// 数据库的连接类对象* **private static final** String ***DBDRIVER*** = **"org.gjt.mm.mysql.Driver"**;  
 **private static final** String ***DBURL*** = **"jdbc:mysql://localhost:3306/shirodb"**;  
 **private static final** String ***DBUSER*** = **"root"**;  
 **private static final** String ***PASSWORD*** = **"root"**;  
 **private** PreparedStatement **pstmt** = **null**;  
  
 **private void** connectionDataBase() { *// 专门负责数据库连接的创建* **try** {  
 Class.*forName*(***DBDRIVER***);  
 **this**.**conn** = DriverManager.*getConnection*(***DBURL***, ***DBUSER***, ***PASSWORD***);  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
 }  
  
 **public void** close() {  
 **if** (**this**.**conn** != **null**) {  
 **try** {  
 **this**.**conn**.close();  
 } **catch** (SQLException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
  
 **public** Member get(String mid) { *// 实现用户登录处理* Member vo = **null**;  
 **try** {  
 String sql = **"SELECT mid,password FROM member WHERE mid=?"**;  
 **this**.**pstmt** = **this**.**conn**.prepareStatement(sql);  
 **this**.**pstmt**.setString(1, mid);  
 ResultSet rs = **this**.**pstmt**.executeQuery();  
 **if** (rs.next()) {  
 vo = **new** Member();  
 vo.setMid(rs.getString(1));  
 vo.setPassword(rs.getString(2));  
 }  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
 **return** vo;  
 }  
  
 */\*\*  
 \* 根据用户名称查询出用户对应的所有的角色数据  
 \*  
 \** ***@param mid*** *\** ***@return*** *\*/* **public** Set<String> listRolesByMember(String mid) {  
 Set<String> allRoles = **new** HashSet<>();  
 String sql = **"SELECT flag FROM role WHERE rid IN ("** + **" SELECT rid FROM member\_role WHERE mid=?)"**;  
 **try** {  
 **this**.**pstmt** = **this**.**conn**.prepareStatement(sql);  
 **this**.**pstmt**.setString(1, mid);  
 ResultSet rs = **this**.**pstmt**.executeQuery();  
 **while** (rs.next()) {  
 allRoles.add(rs.getString(1));  
 }  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
 **return** allRoles;  
 }  
  
 */\*\*  
 \* 根据用户的名称查询出该用户对应的所有权限数据  
 \*  
 \** ***@param mid*** *\** ***@return*** *\*/* **public** Set<String> listActionsByMember(String mid) {  
 Set<String> allActions = **new** HashSet<String>();  
 String sql = **"SELECT flag FROM action WHERE actid IN ("** + **" SELECT actid FROM role\_action WHERE rid IN("** + **" SELECT rid FROM member\_role WHERE mid=?))"**;  
 **try** {  
 **this**.**pstmt** = **this**.**conn**.prepareStatement(sql);  
 **this**.**pstmt**.setString(1, mid);  
 ResultSet rs = **this**.**pstmt**.executeQuery();  
 **while** (rs.next()) {  
 allActions.add(rs.getString(1));  
 }  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
 **return** allActions;  
 }  
}

最终取回的数据都是交给Shiro判断的，在整个处理中，是利用set集合完成的，所以需要返回Set集合数据。

1. 此时的操作不再是简单的用户登录了，还涉及到权限的操作，所以自定义的Realm类必须更换一个父类：

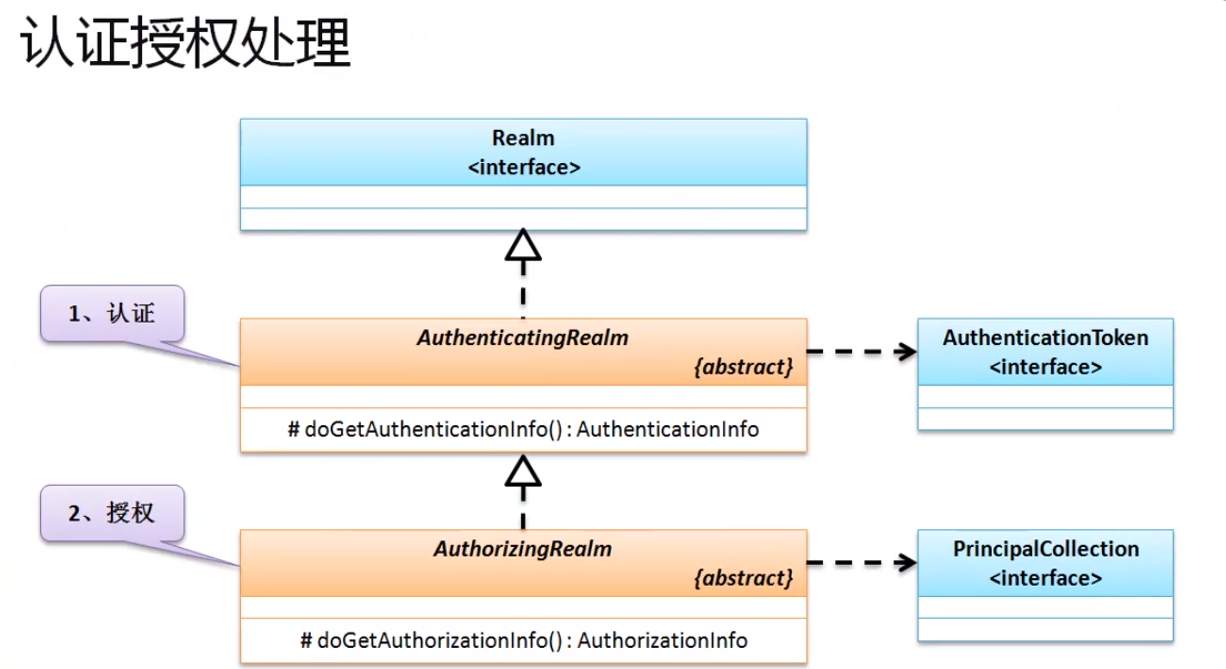
|-认证处理类：AuthenticatingRealm

|-抽象方法： protected abstract AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken var1) throws AuthenticationException;

|-授权处理类：AuthorizingRealm

|-抽象方法：protected abstract AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection var1);

|-通过PrincipalCollection接口取得身份认证信息：Object getPrimaryPrincipal();



1. 自定义认证授权realm类：

public class MyRealm extends AuthorizingRealm {

@Override

protected AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principals) {

System.out.println("\*\*\*\*\*\*\*\*\*\* 2、用户角色与权限：doGetAuthorizationInfo \*\*\*\*\*\*\*\*\*\*");

String username = (String) principals.getPrimaryPrincipal() ; // 取得用户登录名

SimpleAuthorizationInfo auth = new SimpleAuthorizationInfo() ; // 定义授权信息的返回数据

MemberLoginService service = new MemberLoginService() ; // 进行业务层处理

auth.setRoles(service.listRolesByMember(username));// 所有的角色必须以Set集合的形式出现

auth.setStringPermissions(service.listActionsByMember(username)); // 所有的权限必须以Set集合的形式出现

service.close();

return auth;

}

@Override

protected AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken token) throws AuthenticationException {

System.out.println("\*\*\*\*\*\*\*\*\*\* 1、用户登录认证：doGetAuthenticationInfo() \*\*\*\*\*\*\*\*\*\*");

// 1、登录认证的方法需要先执行，需要用他来判断登录的用户信息是否合法

String username = (String) token.getPrincipal() ; // 取得用户名

// 需要通过用户名取得用户的完整信息，利用业务层操作

MemberLoginService service = new MemberLoginService() ;

Member vo = service.get(username) ; // 需要取得的是用户的信息

service.close();

if (vo == null) {

throw new UnknownAccountException("该用户名称不存在！") ;

} else { // 进行密码的验证处理

String password = new String((char []) token.getCredentials()) ;

// 将数据库中的密码与输入的密码进行比较，这样就可以确定当前用户是否可以正常登录

if (vo.getPassword().equals(password)) { // 密码正确

AuthenticationInfo auth = new SimpleAuthenticationInfo(username, password, "memberRealm") ;

return auth ;

} else {

throw new IncorrectCredentialsException("密码错误！") ;

}

}

}

}

## 在web开发中引用Shiro

### 5.1 基础验证配置

1、添加shiro-web依赖：

<dependency>

<groupId>org.apache.shiro</groupId>

<artifactId>shiro-web</artifactId>

<version>1.3.2</version>

</dependency>

1. 在web.xml文件中添加配置：

*<?***xml version="1.0" encoding="UTF-8"***?>*<**web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xmlns="http://xmlns.jcp.org/xml/ns/javaee"  
 xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app\_3\_1.xsd"  
 id="WebApp\_ID" version="3.1"**>  
 <**display-name**>Archetype Created Web Application</**display-name**>  
  
 <**listener**>  
 <**listener-class**>org.apache.shiro.web.env.EnvironmentLoaderListener</**listener-class**>  
 </**listener**>  
  
 <**filter**>  
 <**filter-name**>ShiroFilter</**filter-name**>  
 <**filter-class**>org.apache.shiro.web.servlet.ShiroFilter</**filter-class**>  
 *<!-- 此处明确的表示配置文件的路径 -->* <**init-param**>  
 <**param-name**>configPath</**param-name**>  
 <**param-value**>classpath:shiro.ini</**param-value**>  
 </**init-param**>  
 </**filter**>  
  
 <**filter-mapping**>  
 <**filter-name**>ShiroFilter</**filter-name**>  
 <**url-pattern**>/\*</**url-pattern**>  
 <**dispatcher**>REQUEST</**dispatcher**>  
 <**dispatcher**>FORWARD</**dispatcher**>  
 <**dispatcher**>INCLUDE</**dispatcher**>  
 <**dispatcher**>ERROR</**dispatcher**>  
 </**filter-mapping**>  
</**web-app**>

1. 新建一个单独的Servlet类：

@WebServlet(**"/login"**)  
**public class** LoginServlet **extends** HttpServlet {  
 @Override  
 **protected void** doGet(HttpServletRequest req, HttpServletResponse resp) **throws** ServletException, IOException {  
 **super**.doPost(req, resp);  
 }  
  
 @Override  
 **protected void** doPost(HttpServletRequest req, HttpServletResponse resp) **throws** ServletException, IOException {  
 String mid = req.getParameter(**"mid"**);  
 String password = req.getParameter(**"password"**);  
 UsernamePasswordToken token = **new** UsernamePasswordToken(mid, password);  
 Subject subject = SecurityUtils.*getSubject*();  
 subject.login(token);*//登录* req.getSession().setAttribute(**"mid"**, mid);  
 req.getRequestDispatcher(**"/pages/welcome.jsp"**).forward(req, resp);  
 }  
}

测试登录：

<**html**>  
<**head**>  
<**base href="<%=**basePath**%>"**>  
<**meta http-equiv="Content-Type" content="text/html; charset=UTF-8"**>  
<**title**>Shiro Login Demo</**title**>  
</**head**>  
<**body**>  
 <**form action="login" method="post"**>  
 用户名：<**input type="text" name="mid" id="mid"**><**br**>  
 密**&nbsp;**码：<**input type="password" name="password" id="password"**><**br**>  
 <**input type="submit" value="登录"**>  
 <**input type="reset" value="重置"**>  
 </**form**>   
</**body**>  
</**html**>

如果是启动没有报错，则这说明shiro与web整合成功。除了完成以上配置之外还要修改配置文件：

[main]

# 如果现在认证失败，则跳转到loginUrl配置的路径

authc.loginUrl=/login.jsp

jdbcRealm=cn.mldn.realm.MyRealm

securityManager.realms=$jdbcRealm

# 配置所有需要进行路径检测的页面

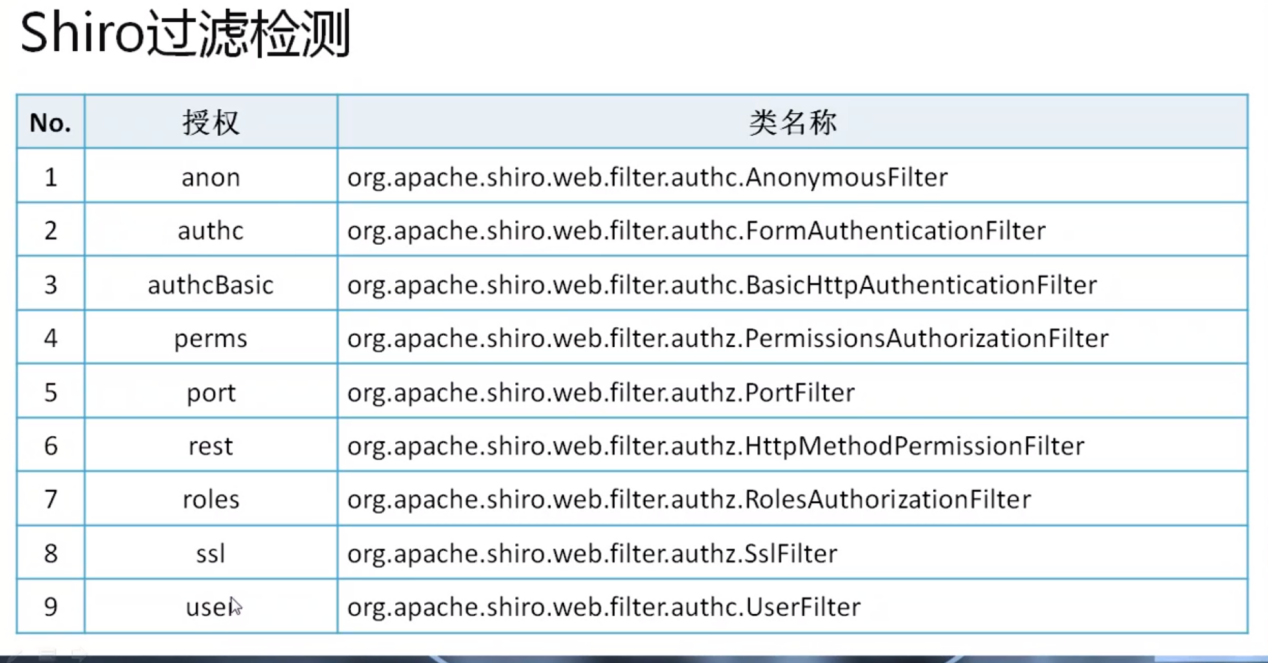
[urls]

# 登录的页面是不需要进行检测处理的

/shiroLogin=anon

# 指定的页面需要进行登录检测

/pages/welcome.jsp=authc



### 5.2 角色认证

1、实现角色检测处理

范例：配置shiro.ini文件

[main]

# 如果现在认证失败，则跳转到loginUrl配置的路径

authc.loginUrl=/login.jsp

**# 需要配置上当角色认证失败之后的跳转页面**

**roles.unauthorizedUrl=/role.jsp**

jdbcRealm=cn.mldn.realm.MyRealm

securityManager.realms=$jdbcRealm

# 配置所有需要进行路径检测的页面

[urls]

# 登录的页面是不需要进行检测处理的

/shiroLogin=anon

**# 指定的页面需要进行登录检测，此时表示需要先进行身份认证，而后再进行角色认证**

**/pages/welcome.jsp=authc,roles[member],roles[dept]**

### 5.3 权限认证

[main]

# 如果现在认证失败，则跳转到loginUrl配置的路径

authc.loginUrl=/login.jsp

# 需要配置上当角色认证失败之后的跳转页面

roles.unauthorizedUrl=/role.jsp

**# 配置权限认证失败的跳转页面**

**perms.unauthorizedUrl=/role.jsp**

jdbcRealm=cn.mldn.realm.MyRealm

securityManager.realms=$jdbcRealm

# 配置所有需要进行路径检测的页面

[urls]

# 登录的页面是不需要进行检测处理的

/shiroLogin=anon

# 指定的页面需要进行登录检测，此时表示需要先进行身份认证，而后再进行角色认证

# /pages/welcome.jsp=authc,roles[member],roles[dept]

**# 对指定页面登录之后进行权限的检测处理**

**/pages/welcome.jsp=authc,perms[member:add],perms[dept:add]**

### 5.4 标签支持

1、引用标签

Shiro提供的标签如下：

Shiro 提供了 JSTL 标签用于在 JSP/GSP 页面进行权限控制，如根据登录用户显示相应的页面按钮。

****导入标签库****

<%@taglib prefix="shiro" uri="http://shiro.apache.org/tags" %>

标签库定义在 shiro-web.jar 包下的 META-INF/shiro.tld 中定义。

guest 标签

<**shiro:guest**>

欢迎游客访问，<**a** href="${pageContext.request.contextPath}/login.jsp">登录</**a**></**shiro:guest**>&nbsp;

用户没有身份验证时显示相应信息，即游客访问信息。

****user 标签****

<**shiro:user/**>

用户已经身份验证 / 记住我登录(rememberme)后显示相应的信息。

****authenticated 标签****

<**shiro:authenticated**>

用户[<**shiro:principal**/>]已身份验证通过</**shiro:authenticated**>&nbsp;

用户已经身份验证通过，即 Subject.login 登录成功，不是记住我登录的。

****notAuthenticated 标签****

<**shiro:notAuthenticated**>

未身份验证（包括记住我）</**shiro:notAuthenticated**>

用户已经身份验证通过，即没有调用 Subject.login 进行登录，包括记住我自动登录的也属于未进行身份验证。

****principal 标签****

<shiro: principal/>

显示用户身份信息，默认调用 Subject.getPrincipal() 获取，即 Primary Principal。

<shiro:principal type="java.lang.String"/>

相当于 Subject.getPrincipals().oneByType(String.class)。

<shiro:principal type="java.lang.String"/>

相当于 Subject.getPrincipals().oneByType(String.class)。

<shiro:principal property="username"/>

相当于 ((User)Subject.getPrincipals()).getUsername()。

****hasRole 标签****

<**shiro:hasRole** name="admin">

用户[<**shiro:principal**/>]拥有角色admin<**br**/></**shiro:hasRole**>&nbsp;

如果当前 Subject 有角色将显示 body 体内容。

****hasAnyRoles 标签****

<**shiro:hasAnyRoles** name="admin,user">

用户[<**shiro:principal**/>]拥有角色admin或user<**br**/></**shiro:hasAnyRoles**>&nbsp;

如果当前 Subject 有任意一个角色（或的关系）将显示 body 体内容。

****lacksRole 标签****

<**shiro:lacksRole** name="abc">

用户[<**shiro:principal**/>]没有角色abc<**br**/></**shiro:lacksRole**>&nbsp;

如果当前 Subject 没有角色将显示 body 体内容。

****hasPermission 标签****

<**shiro:hasPermission** name="user:create">

用户[<**shiro:principal**/>]拥有权限user:create<**br**/></**shiro:hasPermission**>&nbsp;

如果当前 Subject 有权限将显示 body 体内容。

****lacksPermission 标签****

<**shiro:lacksPermission** name="org:create">

用户[<**shiro:principal**/>]没有权限org:create<**br**/></**shiro:lacksPermission**>&nbsp;

如果当前 Subject 没有权限将显示 body 体内容。

另外又提供了几个权限控制相关的标签：

****导入自定义标签库****

<%@taglib prefix="zhang" tagdir="/WEB-INF/tags" %>

****示例****

<**zhang:hasAllRoles** name="admin,user">

用户[<**shiro:principal**/>]拥有角色admin和user<**br**/></**zhang:hasAllRoles**><**zhang:hasAllPermissions** name="user:create,user:update">

用户[<**shiro:principal**/>]拥有权限user:create和user:update<**br**/></**zhang:hasAllPermissions**><**zhang:hasAnyPermissions** name="user:create,abc:update">

用户[<**shiro:principal**/>]拥有权限user:create或abc:update<**br**/></**zhang:hasAnyPermissions**>&nbsp;

hasAllRoles 表示拥有所有相关的角色；hasAllPermissions 表示拥有所有相关的权限；hasAnyPermissions 表示拥有任意一个相关的权限。

### 5.5 路径通配符

在shiro里面也支持Ant风格的通配符：

1. “？”匹配任意一个字符，例如/admin?,匹配/admin1、admin2。
2. “\*”匹配一个或多个任意字符，例如/admin\*，匹配/admin123、/admin12。
3. “\*\*”匹配0个目录，例如/admin\*\*可以匹配/admin/a、/admin/abc/abc。

## spring+mybatis+shiro开发整合

### 6.1 开发环境搭建

1、添加Spring+mybatis+shiro依赖支持：

<**properties**>  
 <**project.build.sourceEncoding**>UTF-8</**project.build.sourceEncoding**>  
 <**maven.compiler.source**>1.8</**maven.compiler.source**>  
 <**maven.compiler.target**>1.8</**maven.compiler.target**>  
 *<!-- spring版本号 -->* <**spring.version**>5.0.5.RELEASE</**spring.version**>  
 *<!-- log4j日志文件管理包版本 -->* <**slf4j.version**>1.7.25</**slf4j.version**>  
 <**log4j.version**>1.2.17</**log4j.version**>  
 *<!-- junit版本号 -->* <**junit.version**>4.12</**junit.version**>  
 *<!-- mybatis版本号 -->* <**mybatis.version**>3.4.6</**mybatis.version**>  
 *<!--shiro版本号-->* <**shiro.version**>1.3.2</**shiro.version**>  
</**properties**>  
  
<**dependencies**>

<**dependency**>  
 <**groupId**>javax.servlet</**groupId**>  
 <**artifactId**>javax.servlet-api</**artifactId**>  
 <**version**>4.0.1</**version**>  
 <**scope**>provided</**scope**>  
</**dependency**>  
 *<!-- 添加Spring依赖 -->* <**dependency**>  
 <**groupId**>commons-io</**groupId**>  
 <**artifactId**>commons-io</**artifactId**>  
 <**version**>2.6</**version**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>commons-fileupload</**groupId**>  
 <**artifactId**>commons-fileupload</**artifactId**>  
 <**version**>1.3.3</**version**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>org.springframework</**groupId**>  
 <**artifactId**>spring-core</**artifactId**>  
 <**version**>${spring.version}</**version**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>org.springframework</**groupId**>  
 <**artifactId**>spring-webmvc</**artifactId**>  
 <**version**>${spring.version}</**version**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>org.springframework</**groupId**>  
 <**artifactId**>spring-context</**artifactId**>  
 <**version**>${spring.version}</**version**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>org.springframework</**groupId**>  
 <**artifactId**>spring-context-support</**artifactId**>  
 <**version**>${spring.version}</**version**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>org.springframework</**groupId**>  
 <**artifactId**>spring-aop</**artifactId**>  
 <**version**>${spring.version}</**version**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>org.springframework</**groupId**>  
 <**artifactId**>spring-aspects</**artifactId**>  
 <**version**>${spring.version}</**version**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>org.springframework</**groupId**>  
 <**artifactId**>spring-tx</**artifactId**>  
 <**version**>${spring.version}</**version**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>org.springframework</**groupId**>  
 <**artifactId**>spring-jdbc</**artifactId**>  
 <**version**>${spring.version}</**version**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>org.springframework</**groupId**>  
 <**artifactId**>spring-web</**artifactId**>  
 <**version**>${spring.version}</**version**>  
 </**dependency**>  
  
 *<!--单元测试依赖 -->* <**dependency**>  
 <**groupId**>junit</**groupId**>  
 <**artifactId**>junit</**artifactId**>  
 <**version**>${junit.version}</**version**>  
 <**scope**>test</**scope**>  
 </**dependency**>  
  
 *<!-- 日志文件管理包 -->  
 <!-- log start -->* <**dependency**>  
 <**groupId**>log4j</**groupId**>  
 <**artifactId**>log4j</**artifactId**>  
 <**version**>${log4j.version}</**version**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>org.slf4j</**groupId**>  
 <**artifactId**>slf4j-api</**artifactId**>  
 <**version**>${slf4j.version}</**version**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>org.slf4j</**groupId**>  
 <**artifactId**>slf4j-log4j12</**artifactId**>  
 <**version**>${slf4j.version}</**version**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>commons-logging</**groupId**>  
 <**artifactId**>commons-logging</**artifactId**>  
 <**version**>1.2</**version**>  
 </**dependency**>  
 *<!-- log end -->  
  
 <!--spring单元测试依赖 -->* <**dependency**>  
 <**groupId**>org.springframework</**groupId**>  
 <**artifactId**>spring-test</**artifactId**>  
 <**version**>${spring.version}</**version**>  
 <**scope**>test</**scope**>  
 </**dependency**>  
  
 *<!--mybatis依赖 -->* <**dependency**>  
 <**groupId**>org.mybatis</**groupId**>  
 <**artifactId**>mybatis</**artifactId**>  
 <**version**>${mybatis.version}</**version**>  
 </**dependency**>  
  
 *<!-- mybatis/spring包 -->* <**dependency**>  
 <**groupId**>org.mybatis</**groupId**>  
 <**artifactId**>mybatis-spring</**artifactId**>  
 <**version**>1.3.2</**version**>  
 </**dependency**>  
  
 *<!-- mysql驱动包 -->* <**dependency**>  
 <**groupId**>mysql</**groupId**>  
 <**artifactId**>mysql-connector-java</**artifactId**>  
 <**version**>5.1.29</**version**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>com.mchange</**groupId**>  
 <**artifactId**>c3p0</**artifactId**>  
 <**version**>0.9.5.2</**version**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>org.apache.shiro</**groupId**>  
 <**artifactId**>shiro-core</**artifactId**>  
 <**version**>${shiro.version}</**version**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>org.apache.shiro</**groupId**>  
 <**artifactId**>shiro-web</**artifactId**>  
 <**version**>${shiro.version}</**version**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>org.apache.shiro</**groupId**>  
 <**artifactId**>shiro-spring</**artifactId**>  
 <**version**>${shiro.version}</**version**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>com.mchange</**groupId**>  
 <**artifactId**>mchange-commons-java</**artifactId**>  
 <**version**>0.2.15</**version**>  
 </**dependency**>  
</**dependencies**>

1. 配置spring(mvc)+mybatis环境

applicationContext.xml配置如下：

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:p="http://www.springframework.org/schema/p"

xmlns:context="http://www.springframework.org/schema/context"

xmlns:mvc="http://www.springframework.org/schema/mvc"

xmlns:aop="http://www.springframework.org/schema/aop"

xmlns:tx="http://www.springframework.org/schema/tx"

xmlns:task="http://www.springframework.org/schema/task"

xsi:schemaLocation="

http://www.springframework.org/schema/task

http://www.springframework.org/schema/task/spring-task-4.3.xsd

http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-4.3.xsd

http://www.springframework.org/schema/mvc http://www.springframework.org/schema/mvc/spring-mvc-4.3.xsd

http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-4.3.xsd

http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-4.3.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-4.3.xsd">

<!-- 定义自动注解扫描的操作 -->

<context:annotation-config/>

<context:component-scan base-package="cn.ym"/>

<!-- 导入所有与数据库连接有关的信息项 -->

<context:property-placeholder location="classpath:database.properties"/>

<!-- 进行数据库连接池的配置 -->

<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">

<property name="driverClass" value="${db.driver}"/> <!-- 驱动程序 -->

<property name="jdbcUrl" value="${db.url}"/><!-- 连接地址 -->

<property name="user" value="${db.user}"/> <!-- 用户名 -->

<property name="password" value="${db.password}"/> <!-- 密码 -->

<property name="maxPoolSize" value="${db.maxPoolSize}"/> <!-- 最大的可用连接数 -->

<property name="minPoolSize" value="${db.minPoolSize}"/> <!-- 最小维持的连接数 -->

<property name="initialPoolSize" value="${db.initialPoolSize}"/> <!-- 初始化连接数 -->

<property name="maxIdleTime" value="${db.maxIdleTime}"/> <!-- 最大等待连接 -->

</bean>

<!-- 定义Spring与MyBatis整合的操作控制，此时数据库的连接对象取得由Spring负责 -->

<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">

<property name="dataSource" ref="dataSource"/>

<!-- 配置所有资源文件的保存路径的资源匹配符 -->

<property name="mapperLocations" value="classpath:cn/ym/vo/mapper/\*.xml"/>

<property name="configLocation" value="classpath:mybatis-config.xml"/>

</bean>

<!-- 配置自动创建数据层实现类的开发包 -->

<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">

<property name="basePackage" value="cn.ym.dao"/>

<property name="sqlSessionFactoryBeanName" value="sqlSessionFactory"/>

</bean>

<!-- 定义数据库的事务控制，本事务控制直接针对于数据库连接操作进行 mybatis没有像hibernate一样提供事务支持，需要通过spring直接在数据库上控制 -->

<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

<property name="dataSource" ref="dataSource"/>

</bean>

<!-- 进入到了事务的配置声明 开启事务注解支持-->

<tx:annotation-driven transaction-manager="transactionManager"/>

<!--定义事务通知-->

<tx:advice transaction-manager="transactionManager" id="txAdvice">

<tx:attributes>

<tx:method name="insert\*" propagation="REQUIRED"/>

<tx:method name="update\*" propagation="REQUIRED"/>

<tx:method name="delete\*" propagation="REQUIRED"/>

<tx:method name="query\*" propagation="REQUIRED" read-only="true"/>

</tx:attributes>

</tx:advice>

<!-- 定义数据库的声明式事务控制，直接与业务方法对应 -->

<!-- 定义事务的处理切入点 -->

<aop:config expose-proxy="true">

<aop:pointcut id="pointcut" expression="execution(\* cn.ym..service..\*.\*(..))"/>

<aop:advisor advice-ref="txAdvice" pointcut-ref="pointcut"/>

</aop:config>

<!-- 以下为shiro需要的配置项 -->

<!-- 配置SecurityManager的管理 -->

<bean id="securityManager" class="org.apache.shiro.mgt.DefaultWebSecurityManager">

<!-- 配置自定义的Realms -->

<property name="realm" ref="memberRealm"/>

</bean>

<!-- 配置shiro过滤器 -->

<bean id="shiroFilter" class="org.apache.shiro.spring.web.ShiroFilterFactoryBean">

<!--配置安全管理器-->

<property name="securityManager" ref="securityManager"/>

<!-- 出现错误之后的跳转路径的配置-->

<property name="loginUrl" value="/login"/>

<!-- 认证失败之后的跳转路径页面 -->

<property name="unauthorizedUrl" value="/unauth"/>

<!-- 登录成功之后的跳转访问路径 -->

<property name="successUrl" value="/success"/>

<!-- shiro里面需要针对于所有的路径进行配置，所有的配置需要通过文本的形式设置 -->

<property name="filterChainDefinitions">

<value>

/\*=anon

/login=anon

/messages/\*\*=authc

/admin\*=authc

/pages/welcome.jsp=authc,perms[member:add],perms[dept:add]

</value>

</property>

</bean>

<!-- 配置Shiro在Spring中的生命周期的控制操作 -->

<bean id="lifecycleBeanPostProcessor" class="org.apache.shiro.spring.LifecycleBeanPostProcessor"/>

</beans>

Web.xml配置：

<?xml version="1.0" encoding="UTF-8"?>

<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app\_3\_1.xsd"

version="3.1">

<!--spring加载配置-->

<display-name>Archetype Created Web Application</display-name>

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>classpath:applicationContext.xml</param-value>

</context-param>

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

<!--springmvc配置-->

<servlet>

<servlet-name>dispatcher</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<init-param>

<param-name>contextConfigLocation</param-name>

<!-- 配置Spring MVC所需要的配置文件，可以与Spring容器的配置文件写在一起 -->

<param-value>classpath:dispatcher-servlet.xml</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>dispatcher</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

<!--编码过滤器-->

<filter>

<filter-name>encoding</filter-name>

<filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>

</filter>

<filter-mapping>

<filter-name>encoding</filter-name>

<url-pattern>/</url-pattern>

</filter-mapping>

<!-- 进行shiro的过滤器的配置 -->

<filter>

<!--过滤器的名字“shiroFilter”应该和spring配置文件中定义的Bean匹配-->

<filter-name>shiroFilter</filter-name>

<filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>

<!-- 该参数表示shiro的生命周期将交由Spring容器进行管理（默认情况下，取值为false） -->

<!-- 如果将其内容设置为true，则表示由Servlet容器进行管理 -->

<init-param>

<param-name>targetFilterLifecycle</param-name>

<param-value>true</param-value>

</init-param>

</filter>

<filter-mapping>

<filter-name>shiroFilter</filter-name>

<url-pattern>/\*</url-pattern>

<!-- <dispatcher>REQUEST</dispatcher>

<dispatcher>FORWARD</dispatcher>

<dispatcher>INCLUDE</dispatcher>

<dispatcher>ERROR</dispatcher>-->

</filter-mapping>

</web-app>

DispatchServlet.xml做如下修改：

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:mvc="http://www.springframework.org/schema/mvc"

xsi:schemaLocation="http://www.springframework.org/schema/mvc

http://www.springframework.org/schema/mvc/spring-mvc-4.3.xsd

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">

<**context:annotation-config**/>  
<**context:component-scan base-package="cn.ym.controller"**/>  
<**mvc:annotation-driven**/> *<!-- 启动Spring MVC的注解配置 -->*<**mvc:default-servlet-handler**/> *<!-- 启用处理请求的servlet -->*<**bean class="org.springframework.web.servlet.view.InternalResourceViewResolver"**>  
 <**property name="prefix" value="/WEB-INF/pages/"**/>  
 <**property name="suffix" value=".jsp"**/>  
</**bean**>

</beans>

1. 新建IMemberDao,不用谢实现类（交由spring管理注入）

*<!-- 配置自动创建数据层实现类的开发包 -->*<**bean class="org.mybatis.spring.mapper.MapperScannerConfigurer"**>  
 <**property name="basePackage" value="cn.ym.dao"**/>  
 <**property name="sqlSessionFactoryBeanName" value="sqlSessionFactory"**/>  
</**bean**>

**public interface** IMemberDAO {  
 **public** Member findById(String mid);  
  
 **public** Set<String> findAllRoleByMember(String mid);  
  
 **public** Set<String> findAllActionByMember(String mid);  
}

方法名称 一定要对应映射文件中的id:

<**mapper namespace="cn.ym.dao.IMemberDAO"**>  
 *<!-- 是进行登录认证使用的，即，根据身份信息取得密码进行认证 -->* <**select id="findById" parameterType="String" resultType="Member"**>  
 SELECT mid,password,name FROM member WHERE mid=#{mid}  
 </**select**>  
 *<!-- 根据用户名查询出该用户对用的所有角色的标记名称（千万不要用中文） -->* <**select id="findAllRoleByMember" parameterType="String" resultType="String"**>  
 SELECT flag FROM role WHERE rid IN (  
 SELECT rid FROM member\_role WHERE mid=#{mid})  
 </**select**>  
 *<!-- 查询出一个用户对应的所有权限数据 -->* <**select id="findAllActionByMember" parameterType="String" resultType="String"**>  
 SELECT flag FROM action WHERE actid IN (  
 SELECT actid FROM role\_action WHERE rid IN (  
 SELECT rid FROM member\_role WHERE mid=#{mid}))  
 </**select**>  
</**mapper**>

Mybatis.config.xml中的配置如下：

*<?***xml version="1.0" encoding="UTF-8"** *?>***<!DOCTYPE configuration  
 PUBLIC "-//mybatis.org//DTD Config 3.0//EN"  
 "http://mybatis.org/dtd/mybatis-3-config.dtd"*>***<**configuration**>  
 <**settings**>  
 <**setting name="cacheEnabled" value="true"**/>  
 <**setting name="logImpl" value="LOG4J"** />  
 </**settings**>  
 <**typeAliases**>  
 <**package name="cn.ym.vo"**/>  
 </**typeAliases**>  
</**configuration**>

### 6.2 内置登录处理

subject.login(token);将被抛弃。

1、修改applicationContext文件，增加内置登录处理：

添加以下配资：

<!-- 此处表示使用内置的表单登录控制验证 -->

<bean id="formAuthenticationFilter" class="org.apache.shiro.web.filter.authc.FormAuthenticationFilter">

<!-- 定义出需要使用的参数，此参数与表单一一对应 -->

<property name="usernameParam" value="mid"/>

<property name="passwordParam" value="password"/>

**<property name="loginUrl" value="/loginUrl.action"/>**

</bean>

1. 修改Shiro过滤器：

<!-- 配置shiro过滤器 -->

<bean id="shiroFilter" class="org.apache.shiro.spring.web.ShiroFilterFactoryBean">

<!-- 表示现在要配置的是一个安全管理器 -->

<property name="securityManager" ref="securityManager"/>

<!-- 出现错误之后的跳转路径的配置 -->

<property name="loginUrl" value="/loginUrl.action"/>

<!-- 认证失败之后的跳转路径页面 -->

<property name="unauthorizedUrl" value="/unauthUrl.action"/>

<!-- 登录成功之后的跳转访问路径 -->

<property name="successUrl" value="/successUrl.action"/>

<!-- 配置shiro里面需要使用到的过滤器操作 -->

<property name="filters">

<map>

<entry key="**authc**" value-ref="formAuthenticationFilter"/>

</map>

</property>

<!-- shiro里面需要针对于所有的路径进行配置，所有的配置需要通过文本的形式设置 -->

<property name="filterChainDefinitions">

<value>

**/loginUrl.action=authc**

/\*=anon

/messages/\*\*=authc

/admin\*=authc

/pages/welcome.jsp=authc,perms[member:add],perms[dept:add]

</value>

</property>

</bean>

1. 修改Login.jsp表单提交路径：

如果要使用内置的登录处理，必须要求将你的登录页面设置为authc的过滤，需要将代码中的loginUrl设置为表单的提交路径，否则无法使用。

### 6.3 控制层授权检测

在dispatchServle.xml文件中，启用shiro注解配置：

1. 需要启用aop的代理设计程序：

<!-- 增加shiro的相关配置操作 -->

<!-- 启动在Shiro里面进行Annotation的相关验证处理操作 -->

<bean id="controllerDefaultAdvisorAutoProxyCreator"

class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"

depends-on="lifecycleBeanPostProcessor">

<!-- 进行目标程序类的代理控制 -->

<property name="proxyTargetClass" value="true" />

</bean>

1. 针对于安全管理器的aop配置：

<!-- 针对于安全管理实现的AOP处理操作 -->

<bean id="controllerAuthorizationAttributeSourceAdvisor"

class="org.apache.shiro.spring.security.interceptor.AuthorizationAttributeSourceAdvisor">

<property name="securityManager" ref="securityManager"/>

</bean>

在shiro里面控制层的注解有如下几种：

1. 认证检测：RequiresAuthentication，表示该控制方法必须登录后才能使用。

未认证异常跳转指定错误页（web.xml）：

<error-page>

<exception-type>

org.apache.shiro.authz.UnauthenticatedException

</exception-type>

<location>/loginUrl.action</location>

</error-page>

授权检测：

1. 角色检测：RequiresRoles
2. 权限检测：RequiresPermissions

未授权异常也要跳转到指定页，

### 6.4 业务层授权检测

如果想要业务层上的方法授权检测有效，需要修改application.xml配置文件。

从dispatchServlet.xml配置文件中拷贝（id名称修改即可）一份控制层的配置到application.xml文件配置中

<!-- 启动在Shiro里面进行Annotation的相关验证处理操作 -->

<bean id="serviceDefaultAdvisorAutoProxyCreator"

class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"

depends-on="lifecycleBeanPostProcessor">

<!-- 进行目标程序类的代理控制 -->

<property name="proxyTargetClass" value="true" />

</bean>

<!-- 针对于安全管理实现的AOP处理操作 -->

<bean id="serviceAuthorizationAttributeSourceAdvisor"

class="org.apache.shiro.spring.security.interceptor.AuthorizationAttributeSourceAdvisor">

<property name="securityManager" ref="securityManager"/>

</bean>

只要完成以上配置，就可以是业务层的方法也可以使用shiro注解授权。

利用注解实现的配置都属于与的关系（多个权限或角色），因为从正常开发来讲，一个方法只会对应一种权限或者是一个角色。

## shiro高级话题

### 7.1 缓存的配置

1、添加shiro缓存依赖。

1. 定义一个缓存的配置文件（直接拷贝，官网提供的）：
2. 在此配置文件中，有以下几个核心配置项：

* <diskStore path="java.io.tmpdir"/>：磁盘的存储目录；
* “name=xxx”：对要进行的缓存项进行标注。
* “maxElementsInMemory="2000"”：可以缓存的最大个数；
* “eternal="false"”是否允许自动失效；
* “timeToIdleSeconds="1800"”：最小的失效时间，单位秒；
* “timeToLiveSeconds="0"”：最大保存时间，单位秒，如果是0则不保存；
* “overflowToDisk="false"”：如果容量过多，可以将其保存在磁盘上；

最简单的配置：

*<?***xml version="1.1" encoding="UTF-8"***?>*<**ehcache name="shirocache"**>  
  
 <**diskStore path="java.io.tmpdir"**/>  
  
 <**defaultCache  
 maxElementsInMemory="2000"  
 eternal="false"  
 timeToIdleSeconds="120"  
 timeToLiveSeconds="120"  
 overflowToDisk="true"** />  
</**ehcache**>

要想缓存失效，还需要修改applicatoion配置：

<!-- 进行缓存的操作配置 -->

<bean id="cacheManager" class="org.apache.shiro.cache.ehcache.EhCacheManager">

<property name="cacheManagerConfigFile" value="classpath:ehcache.xml"/>

</bean>

在安全管理器中，定义缓存管理器：

<!-- 配置SecurityManager的管理 -->

<bean id="securityManager" class="org.apache.shiro.web.mgt.DefaultWebSecurityManager">

<!-- 配置你需要使用的Realms -->

<property name="realm" ref="memberRealm"/>

<property name="cacheManager" ref="cacheManager"/>

</bean>

### 8.2 会话管理

添加新的shiro新的依赖

<dependency>

<groupId>org.apache.shiro</groupId>

<artifactId>shiro-quartz</artifactId>

<version>1.3.2</version>

</dependency>

<dependency>

<groupId>commons-collections</groupId>

<artifactId>commons-collections</artifactId>

<version>3.2.2</version>

</dependency>

Session管理一定需要定期释放空间，使用定时调度。

1. 可以配置一个属于自己的session ID生成器类(application.xml)：

* 生成类：org.apache.shiro.session.mgt.eis.JavaUuidSessionIdGenerator

<!-- 定义Session ID生成管理器 -->

<bean id="sessionIdGenerator"

class="org.apache.shiro.session.mgt.eis.JavaUuidSessionIdGenerator" />

1. 定义一个会话的DAO处理：

* 会话保存处理类：org.apache.shiro.session.mgt.eis.EnterpriseCacheSessionDAO

<!-- 配置Session DAO的操作处理 -->

<bean id="sessionDAO"

class="org.apache.shiro.session.mgt.eis.EnterpriseCacheSessionDAO">

<!-- 设置session缓存的名字，这个名字可以任意 -->

<property name="activeSessionsCacheName" value="shiro-activeSessionCache"/>

<!-- 定义该Session DAO操作中所使用的ID生成器 -->

<property name="sessionIdGenerator" ref="sessionIdGenerator"/>

</bean>

1. 为进行有效的session管理，还需要建立一个Cookie的操作模板。

* 处理cookie：org.apache.shiro.web.servlet.SimpleCookie

<!-- 配置需要向Cookie中保存数据的配置模版 -->

<bean id="sessionIdCookie" class="org.apache.shiro.web.servlet.SimpleCookie">

<!-- 在Tomcat运行下默认使用的Cookie的名字为JSESSIONID ，通过构造器设置自定义名称-->

<constructor-arg value="yangmi-session-id"/>

<!-- 保证该系统不会受到跨域的脚本操作攻击，不配置默认也为true -->

<property name="httpOnly" value="true"/>

<!-- 定义Cookie的过期时间，单位为秒，如果设置为-1表示浏览器关闭，则Cookie消失 -->

<property name="maxAge" value="-1"/>

</bean>

1. 定义会话管理器**sessionManager（核心）：**

* 操作类：org.apache.shiro.web.session.mgt.DefaultWebSessionManager

<!-- 定义会话管理器的操作 -->

<bean id="sessionManager"

class="org.apache.shiro.web.session.mgt.DefaultWebSessionManager">

<!-- 定义的是全局的session会话超时时间，此操作会覆盖web.xml文件中的超时时间配置 -->

<property name="globalSessionTimeout" value="1000000"/>

<!-- 删除所有无效的Session对象，此时的session被保存在了内存里面 -->

<property name="deleteInvalidSessions" value="true"/>

<!-- 定义要使用的无效的Session定时调度器（利用调度器删除session） -->

<property name="sessionValidationScheduler" ref="sessionValidationScheduler"/>

<!-- 需要让此session可以使用该定时调度器进行检测 -->

<property name="sessionValidationSchedulerEnabled" value="true"/>

<!-- 定义Session可以进行序列化的工具类 -->

<property name="sessionDAO" ref="sessionDAO"/>

<!-- 所有的session一定要将id设置到Cookie之中，需要提供有Cookie的操作模版 -->

<property name="sessionIdCookie" ref="sessionIdCookie"/>

<!-- 定义sessionIdCookie模版可以进行操作的启用 -->

<property name="sessionIdCookieEnabled" value="true"/>

</bean>

1. 用户离开，一般是不会去点注销按钮，为保证cookie清空，所以需要使用一个会话的调度器。

* 调度器程序类：org.apache.shiro.session.mgt.quartz.QuartzSessionValidationScheduler

<!-- 配置session的定时验证检测程序类，以让无效的session释放 -->

<bean id="sessionValidationScheduler"

class="org.apache.shiro.session.mgt.quartz.QuartzSessionValidationScheduler">

<!-- 设置session的失效扫描间隔，单位为毫秒 -->

<property name="sessionValidationInterval" value="100000"/>

<!-- 随后还需要定义有一个会话管理器的程序类的引用 -->

<property name="sessionManager" ref="sessionManager"/>

</bean>

1. 修改安全管理器：

<bean id="securityManager" class="org.apache.shiro.web.mgt.DefaultWebSecurityManager">

<!-- 配置你需要使用的Realms -->

<property name="realm" ref="memberRealm"/>

<property name="cacheManager" ref="cacheManager"/>

**<!-- 定义要使用的session管理器 -->**

**<property name="sessionManager" ref="sessionManager"/>**

</bean>

1. 另外的方法：

在Subject接口中，取得的session对象也可以有一些处理方法：

更新会话：touch();

停止会话：stop();相当于shiro在web开发中的invalidate和注销logout;

### 8.3 RememberMe

1、RememberMe功能一是要在客户端保存一个cooike的，那么就需要配置cookie模板。

<!-- 配置需要向Cookie中保存数据的配置模版（RememberMe） -->

<bean id="rememberMeCookie" class="org.apache.shiro.web.servlet.SimpleCookie">

<!-- 设置Cookie在浏览器中保存内容的名字，由用户自己来设置 -->

<constructor-arg value="yangmi-RememberMe"/>

<!-- 保证该系统不会受到跨域的脚本操作供给 -->

<property name="httpOnly" value="true"/>

<!-- 定义Cookie的过期时间为一小时 -->

<property name="maxAge" value="3600"/>

</bean>

1. 配置RememberMe的管理器：

|-管理器控制类：org.apache.shiro.web.mgt.CookieRememberMeManager;

<!-- 定义RememberMe功能的程序管理类 -->

<bean id="rememberMeManager" class="org.apache.shiro.web.mgt.CookieRememberMeManager">

<!-- 定义在进行RememberMe功能实现的时候所需要使用到的Cookie的处理类 -->

<property name="cookie" ref="rememberMeCookie"/>

</bean>

1. 在安全管理器里面注册MemeberMe功能：

<!-- 配置SecurityManager的管理 -->

<bean id="securityManager" class="org.apache.shiro.web.mgt.DefaultWebSecurityManager">

<!-- 配置你需要使用的Realms -->

<property name="realm" ref="memberRealm"/>

<property name="cacheManager" ref="cacheManager"/>

<!-- 定义要使用的session管理器 -->

<property name="sessionManager" ref="sessionManager"/>

**<!-- 定义RememberMe的管理器 -->**

**<property name="rememberMeManager" ref="rememberMeManager"/>**

</bean>

1. 在表单上添加“记住我”的复选框：

|-复选框的内容要严格控制好，只能能够使用“true”;

<form action="loginUrl.action" method="post">

用户名：<input type="text" name="mid" id="mid"><br>

密&nbsp;码：<input type="password" name="password" id="password"><br>

<input type="checkbox" name="rememberMe" value="true">记住密码<br>

<input type="submit" value="登录">

<input type="reset" value="重置">

</form>

1. 修改登录控制器：

<!-- 以下为shiro需要的配置项 -->

<!-- 此处表示使用内置的表单登录控制验证 -->

<bean id="formAuthenticationFilter" class="org.apache.shiro.web.filter.authc.FormAuthenticationFilter">

<!-- 定义出需要使用的参数，此参数与表单一一对应 -->

<property name="usernameParam" value="mid"/>

<property name="passwordParam" value="password"/>

**<property name="rememberMeParam" value="rememberMe"/>**

<property name="loginUrl" value="/member/login"/>

</bean>

1. 增加一个不需要登录，只需要记住我就可以访问的路径：

<!-- 配置shiro过滤器 -->

<bean id="shiroFilter" class="org.apache.shiro.spring.web.ShiroFilterFactoryBean">

<!-- 表示现在要配置的是一个安全管理器 -->

<property name="securityManager" ref="securityManager"/>

<!-- 出现错误之后的跳转路径的配置 -->

<property name="loginUrl" value="/loginUrl.action"/>

<!-- 认证失败之后的跳转路径页面 -->

<property name="unauthorizedUrl" value="/unauthUrl.action"/>

<!-- 登录成功之后的跳转访问路径 -->

<property name="successUrl" value="/successUrl.action"/>

<!-- 配置shiro里面需要使用到的过滤器操作 -->

<property name="filters">

<map>

<entry key="authc" value-ref="formAuthenticationFilter"/>

</map>

</property>

<!-- shiro里面需要针对于所有的路径进行配置，所有的配置需要通过文本的形式设置 -->

<property name="filterChainDefinitions">

<value>

/loginUrl.action=authc

/\*=anon

**/messages/\*\*=authc**

**/infos/\*\*=user**

/admin\*=authc

/pages/welcome.jsp=authc,perms[member:add],perms[dept:add]

</value>

</property>

</bean>

|  |  |
| --- | --- |
| /messages/\*\*=authc | 访问此操作必须登录，此时RememberMe无效，比如有些操作，必须登录（支付） |
| /infos/\*\*=user | 只要登录过，并且记住了密码，那么就表示以后可以直接访问啦 |

Shiro正对于rememberMe的功能，不是简单的记住用户名或密码就可以了，往往需要加密处理。

### 8.4 加密处理

1、取得MD5开发程序类

...

1. 对于现在使用的salt,建议还是使用Base64编码完成：

String val = "www.ym.cn";

System.out.println(new String(Base64.getEncoder().encode(val.getBytes())));

1. 将生成的salt与MD5整合进行加密处理。Password+{{salt}}
2. 修改数据库部分脚本数据：

DROP DATABASE IF EXISTS shirodb ;

CREATE DATABASE shirodb CHARACTER SET UTF8 ;

USE shirodb ;

CREATE TABLE member (

mid VARCHAR(50),

password VARCHAR(32),

name VARCHAR(20),

locked INT,

CONSTRAINT pk\_mid PRIMARY KEY (mid)

) type = innodb;

-- 2、角色数据表

CREATE TABLE role (

rid INT AUTO\_INCREMENT ,

title VARCHAR(50),

flag VARCHAR(50),

CONSTRAINT pk\_rid PRIMARY KEY (rid)

) type = innodb;

-- 3、用户-角色关系表

CREATE TABLE member\_role (

mid VARCHAR(50),

rid INT ,

CONSTRAINT fk\_mid1 FOREIGN KEY(mid) REFERENCES member(mid) ,

CONSTRAINT fk\_rid1 FOREIGN KEY(rid) REFERENCES role(rid)

) type = innodb;

-- 4、权限表

CREATE TABLE action (

actid INT AUTO\_INCREMENT ,

title VARCHAR(50),

flag VARCHAR(50),

CONSTRAINT pk\_actid PRIMARY KEY (actid)

) type = innodb;

-- 5、角色-权限关系表

CREATE TABLE role\_action (

rid INT,

actid INT,

CONSTRAINT fk\_rid6 FOREIGN KEY(rid) REFERENCES role(rid) ,

CONSTRAINT fk\_actid6 FOREIGN KEY(actid) REFERENCES action(actid)

) type = innodb;

-- 五、测试数据

-- 1、增加角色信息

INSERT INTO role(title,flag) VALUES ('管理员','member') ;

INSERT INTO role(title,flag) VALUES ('部门管理','dept') ;

INSERT INTO role(title,flag) VALUES ('雇员管理','emp') ;

-- 2、增加权限信息

INSERT INTO action(title,flag) VALUES ('增加管理员','member:add') ;

INSERT INTO action(title,flag) VALUES ('管理员列表','member:list') ;

INSERT INTO action(title,flag) VALUES ('部门增加','dept:add') ;

INSERT INTO action(title,flag) VALUES ('部门列表','dept:list') ;

INSERT INTO action(title,flag) VALUES ('部门编辑','dept:edit') ;

INSERT INTO action(title,flag) VALUES ('员工入职','emp:add') ;

INSERT INTO action(title,flag) VALUES ('员工列表','emp:list') ;

INSERT INTO action(title,flag) VALUES ('员工编辑','emp:edit') ;

INSERT INTO action(title,flag) VALUES ('员工离职','emp:remove') ;

-- 3、增加角色与权限关系

INSERT INTO role\_action(rid,actid) VALUES (1,1) ;

INSERT INTO role\_action(rid,actid) VALUES (1,2) ;

INSERT INTO role\_action(rid,actid) VALUES (2,3) ;

INSERT INTO role\_action(rid,actid) VALUES (2,4) ;

INSERT INTO role\_action(rid,actid) VALUES (2,5) ;

INSERT INTO role\_action(rid,actid) VALUES (3,6) ;

INSERT INTO role\_action(rid,actid) VALUES (3,7) ;

INSERT INTO role\_action(rid,actid) VALUES (3,8) ;

INSERT INTO role\_action(rid,actid) VALUES (3,9) ;

INSERT INTO member(mid,password,name,locked) VALUES ('admin','375D8EFF3A796C7C662CB0BF8A2E1182','管理员',0) ;

INSERT INTO member(mid,password,name,locked) VALUES ('mermaid','375D8EFF3A796C7C662CB0BF8A2E1182','老李',0) ;

INSERT INTO member(mid,password,name,locked) VALUES ('mldn','D7E88505FB8C22A9662219253A664F73','隔壁老王',0) ;

INSERT INTO member\_role(mid,rid) VALUES ('admin',1) ;

INSERT INTO member\_role(mid,rid) VALUES ('admin',2) ;

INSERT INTO member\_role(mid,rid) VALUES ('mermaid',3) ;

INSERT INTO member\_role(mid,rid) VALUES ('mldn',2) ;

INSERT INTO member\_role(mid,rid) VALUES ('mldn',3) ;

1. 定义一个专门负责取得加密后密码的工具类。

private static final String SALT = "";

/\*\*

\* 提供有密码的加密处理操作

\* @param password

\* @return

\*/

public static String encryptPassword(String password) {

return new MD5Code().getMD5ofStr(password + "{{"+SALT+"}}") ;

}

1. 修改自定义Realm,在进行密码匹配的时候对密码进行加密处理。

public class MemberRealm extends AuthorizingRealm {

@Resource

private IMemberService memberService ;

@SuppressWarnings("unchecked")

@Override

protected AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principals) {

System.out.println("\*\*\*\*\*\*\*\*\*\* 2、用户角色与权限：doGetAuthorizationInfo \*\*\*\*\*\*\*\*\*\*");

String username = (String) principals.getPrimaryPrincipal() ; // 取得用户登录名

SimpleAuthorizationInfo auth = new SimpleAuthorizationInfo() ; // 定义授权信息的返回数据

try {

Map<String,Object> map = this.memberService.listAuthByMember(username) ;

Set<String> allRoles = (Set<String>) map.get("allRoles") ;

Set<String> allActions = (Set<String>) map.get("allActions") ;

auth.setRoles(allRoles);// 所有的角色必须以Set集合的形式出现

auth.setStringPermissions(allActions); // 所有的权限必须以Set集合的形式出现

} catch (Exception e) {

e.printStackTrace();

}

return auth;

}

@Override

protected AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken token) throws AuthenticationException {

System.out.println("\*\*\*\*\*\*\*\*\*\* 1、用户登录认证：doGetAuthenticationInfo() \*\*\*\*\*\*\*\*\*\*");

// 1、登录认证的方法需要先执行，需要用他来判断登录的用户信息是否合法

String username = (String) token.getPrincipal() ; // 取得用户名

// 需要通过用户名取得用户的完整信息，利用业务层操作

Member vo = null ;

try {

vo = this.memberService.get(username) ;

} catch (Exception e) {

e.printStackTrace();

}

if (vo == null) {

throw new UnknownAccountException("该用户名称不存在！") ;

} else { // 进行密码的验证处理

**String password = MyPasswordEncrypt.encryptPassword(new String((char[]) token.getCredentials()));**

// 将数据库中的密码与输入的密码进行比较，这样就可以确定当前用户是否可以正常登录

if (vo.getPassword().equals(password)) { // 密码正确

AuthenticationInfo auth = new SimpleAuthenticationInfo(username, password, "memberRealm") ;

return auth ;

} else {

throw new IncorrectCredentialsException("密码错误！") ;

}

}

}

}

1. 自定义认证匹配器：

public class CustomerCredentialsMatcher extends SimpleCredentialsMatcher {

@Override

public boolean doCredentialsMatch(AuthenticationToken token, AuthenticationInfo info) {

// 取得原始的输入数据信息

Object tokenCredentials = MyPasswordEncrypt.encryptPassword(super.toString(token.getCredentials())).getBytes();

// 取得认证数据库中的数据

Object accountCredentials = super.getCredentials(info) ;

return super.equals(tokenCredentials, accountCredentials);

}

}

9、将MemberRealm的操作配置定义在application.xml文件中，因为此时需要定义认证匹配器。

<!-- 定义Realm，但是名称一定要注意好 -->

<bean id="memberRealm" class="cn.mldn.realm.MemberRealm">

<property name="credentialsMatcher">

<bean class="cn.mldn.realm.cre.CustomerCredentialsMatcher"/>

</property>

</bean>

### 8.5验证码处理

1. 本次继续使用kaptcha验证码组件，添加组件依赖；

<**dependency**>  
 <**groupId**>com.github.penggle</**groupId**>  
 <**artifactId**>kaptcha</**artifactId**>  
 <**version**>2.3.2</**version**>  
</**dependency**>

1. 在web.xml里面追加相关的配置项；

<**servlet**> *<!-- 验证码功能 -->* <**servlet-name**>KaptchaServlet</**servlet-name**>  
 <**servlet-class**>com.google.code.kaptcha.servlet.KaptchaServlet</**servlet-class**>  
 <**init-param**> *<!-- 定义是否存在有边框 -->* <**param-name**>kaptcha.border</**param-name**>  
 <**param-value**>no</**param-value**>  
 </**init-param**>  
 <**init-param**> *<!-- 定义边框颜色 -->* <**param-name**>kaptcha.border.color</**param-name**>  
 <**param-value**>105,179,90</**param-value**>  
 </**init-param**>  
 <**init-param**> *<!-- 设置字体颜色 -->* <**param-name**>kaptcha.textproducer.font.color</**param-name**>  
 <**param-value**>red</**param-value**>  
 </**init-param**>  
 <**init-param**>  
 <**description**>  
 com.google.code.kaptcha.impl.ShadowGimpy  
 </**description**>  
 <**param-name**>kaptcha.obscurificator.impl</**param-name**>  
 <**param-value**>  
 com.google.code.kaptcha.impl.WaterRipple  
 </**param-value**>  
 </**init-param**>  
 <**init-param**> *<!-- 设置生成图片宽度 -->* <**param-name**>kaptcha.image.width</**param-name**>  
 <**param-value**>105</**param-value**>  
 </**init-param**>  
 <**init-param**> *<!-- 设置生成图片高度 -->* <**param-name**>kaptcha.image.height</**param-name**>  
 <**param-value**>50</**param-value**>  
 </**init-param**>  
 <**init-param**> *<!-- 设置验证码显示文字尺寸 -->* <**param-name**>kaptcha.textproducer.font.size</**param-name**>  
 <**param-value**>30</**param-value**>  
 </**init-param**>  
 <**init-param**> *<!-- 设置验证码保存到session中的属性名称 -->* <**param-name**>kaptcha.session.key</**param-name**>  
 <**param-value**>rand</**param-value**>  
 </**init-param**>  
 <**init-param**> *<!-- 设置字符间距 -->* <**param-name**>kaptcha.textproducer.char.space</**param-name**>  
 <**param-value**>2</**param-value**>  
 </**init-param**>  
 <**init-param**> *<!-- 设置生成的验证码长度 -->* <**param-name**>kaptcha.textproducer.char.length</**param-name**>  
 <**param-value**>4</**param-value**>  
 </**init-param**>  
 <**init-param**> *<!-- 设置使用的字体 -->* <**param-name**>kaptcha.textproducer.font.names</**param-name**>  
 <**param-value**>宋体,楷体,微软雅黑</**param-value**>  
 </**init-param**>  
</**servlet**>  
<**servlet-mapping**>  
 <**servlet-name**>KaptchaServlet</**servlet-name**>  
 <**url-pattern**>/captcha.jpg</**url-pattern**>  
</**servlet-mapping**>

1. 如果想实现自定义验证码的实现“org.apache.shiro.web.filter.authc.FormAuthenticationFilter”，需要定义一个子类，而后覆写该类中的指定方法：

/\*\*

\* 在已有的Form认证授权器基础上扩展一个新的子类

\* @author mldn

\*/

public class CustomerFormAuthenticationFilter extends FormAuthenticationFilter {

@Override

protected boolean onAccessDenied(ServletRequest request, ServletResponse response) throws Exception {

// 1、如果要想取得在Session中出现的验证码，则必须取得HttpSession接口对象

HttpServletRequest req = (HttpServletRequest) request ;

HttpSession session = req.getSession() ; // 取得当前的session对象

String rand = (String) session.getAttribute("rand") ; // 取得生成的验证码

// 2、取得用户提交表单过来的验证码数据

String code = request.getParameter("code") ;

if (rand == null || code == null || "".equals(rand) || "".equals(code)) {

request.setAttribute("code", "验证码不允许为空！");

return true ; // 拒绝访问，不再进行用户名或密码的检测

} else {

if (!code.equalsIgnoreCase(rand)) { // 验证码输入错误

request.setAttribute("code", "验证码输入错误！");

return true ;

}

}

return super.onAccessDenied(request, response) ; // 操作继续向后执行

}

}

在配置文件中。用实现的子类替换父类