Shiro安全框架

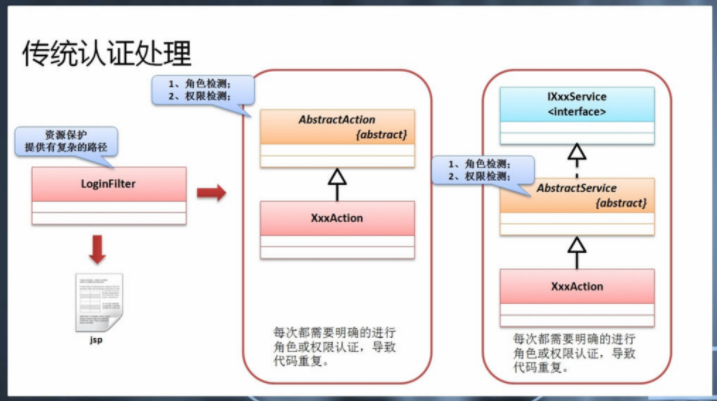
# Shiro简介

Shiro是一个专门的权限以及认证授权的处理验证框架，它的主要功能是进行用户登录检测以及授权认证处理的操作开发框架。Shiro是由Apache推广的一个开源项目。

## 传统认证模式

几乎所有的项目都会存在有登录、权限或角色检测的处理操作。那么最初关于用户登录、角色、权限的基本描述的结构：

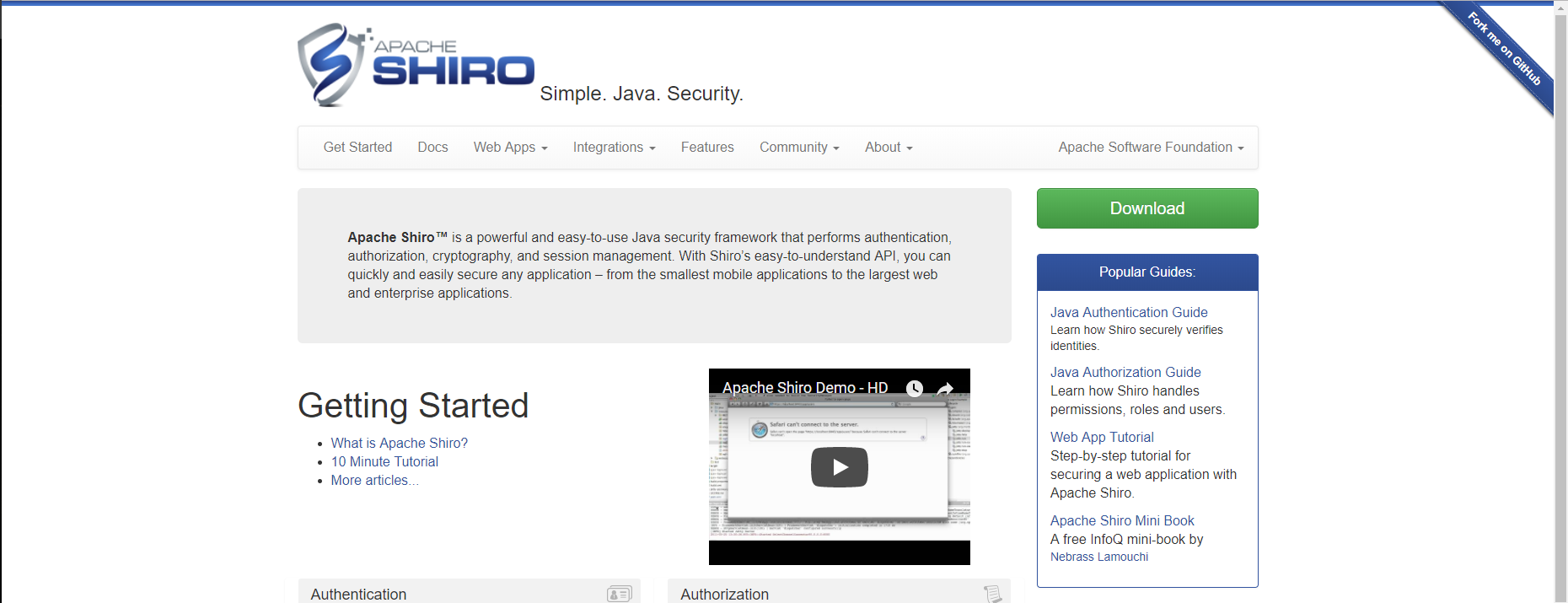
* 用户在进行登录的时候必须验证用户名和密码；
* 用户登录之后才表示可以通过过滤器的检测，过滤器可以保证未登录过的用户无法进行资源访问；
* 用户在登录之后可以进行各种操作，但是不同的用户会存在有不同的角色以及不同的权限，这个时候为了保证操作的安全性，所以需要控制层进行一个当前用户的权限认证；
* 在开发之中业务层的操作是由可能单独独立出去，成为一个独立的业务接口，这个业务接口可以系通过WebService进行项目的发布，那么在此时就会出现有一个问题；业务层有可能单独对外，那么同样需要进行角色或者权限的检测；



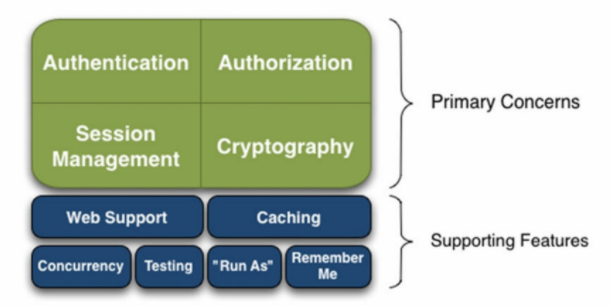
就会发现，几乎所有的项目都会存在有同样的流程，那么很明显如果每一次都重复进行开发，那么这个过程实在是不妥。为此，如果真要进行项目的编写，强烈不建议使用固定的编码模式，正是在这样的前提下，行业内产生了两个最为流行的认证开发框架：Spring Security、Shiro。但是Shiro慢慢凭借着自己的优势以及配置简单的特点，慢慢开始流行开发，已经成为了现在从业的标准，而Shiro往往会与SpringMVC和MyBatis结合实现更为方便的用户认证、 操作。

## Shiro简介

Shiro是由Apache推出的一个基于Java实现的安全框架，Shiro的实现小巧而且功能完善。可以直接通过<http://shiro.apache.org/>网站上下载Shiro开发包；

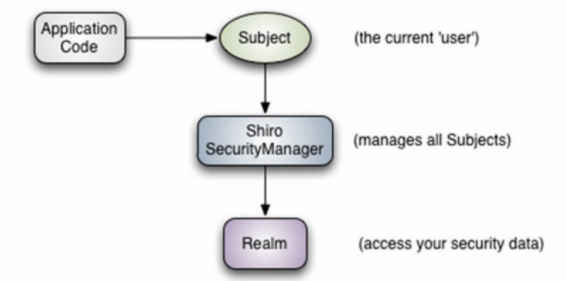


## Shiro主要功能

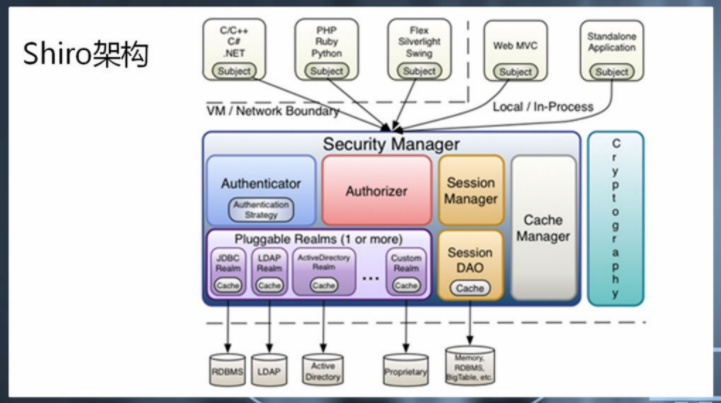


* Authentication：身份认证/登录，验证用户是不是拥有相应的身份；
* Authorization：授权，即权限验证，验证某个已认证的用户是否拥有某个权限；即判断用户是否能做事情，常见的如：验证某个用户是否拥有某个角色。或者细粒度的验证某个用户对某个资源是否具有某个权限；
* Session Manager：会话管理，即用户登录后就是一次会话，在没有退出之前，它的所有信息都在会话中；会话可以是普通JavaSE环境的，也可以是如Web环境的；
* Cryptography：加密，保护数据的安全性，如密码加密存储到数据库，而不是明文存储；

## Shiro工作流程



* Subject：应用代码直接交互的对象是Subject，所有Subject都绑定到SecurityManager，可以把Subject认为是一个门面；SecurityManager才是实际的执行者；
* SecurityManager：安全管理器；即所有与安全有关的操作都会与SecurityManager交互；且它管理着所有Subject；可以看出它是Shiro的核心，它负责与后边介绍的其它组件进行交互，如果学习过SpringMVC，你可以把它看成DispatcherServlet前段控制器；
* 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。



在整个Shiro之中，关键的操作步骤依然需要由用户进行开发实现。

## 总结

Shiro只是提供有一个用户邓丽、授权认证的处理开发框架，它可以对所有的项目进行统一的授权处理操作。

# 基础身份验证

在实际的开发之中所有的用户名和密码的认证过程一定是交由数据库完成的，但是本次由于是第一次接触Shiro开发，所以本次将利用固定的资源文件来实现用户名和密码的配置以及基础的认证实现。

## 搭建项目开发环境

如果要进行shiro项目的开发环境搭建，建议使用Maven完成。

1. 建立一个maven的项目：
   1. 本次为了以后的开发方便，将建立web项目；
2. 需要配置与shiro有关的开发程序包：
   1. 配置servlet程序包，这样可以保证整个的项目不出错；

|  |
| --- |
| <dependency>  <groupId>javax.servlet</groupId>  <artifactId>javax.servlet-api</artifactId>  <version>3.1.0</version> </dependency> |

* 1. 配置shiro的相关开发包；

|  |
| --- |
| <dependency>  <groupId>org.apache.shiro</groupId>  <artifactId>shiro-core</artifactId>  <version>1.3.2</version> </dependency> |

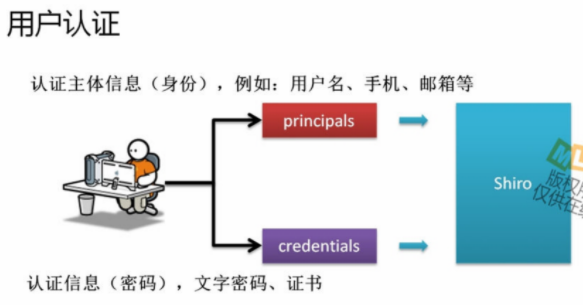
* 1. 与日志有关的工作开发包配置；

|  |
| --- |
| <dependency>  <groupId>log4j</groupId>  <artifactId>log4j</artifactId>  <version>1.2.17</version> </dependency> <dependency>  <groupId>com.mchange</groupId>  <artifactId>mchange-commons-java</artifactId>  <version>0.2.12</version> </dependency> <dependency>  <groupId>commons-logging</groupId>  <artifactId>commons-logging</artifactId>  <version>1.1.3</version> </dependency> <dependency>  <groupId>org.slf4j</groupId>  <artifactId>slf4j-api</artifactId>  <version>1.7.25</version> </dependency> |

1. 在资源目录中配置log4j.properties文件；

## 固定信息验证

本次的信息认证操作都将保存在一个资源文件里面，假设这个资源文件的名称为：shiro.ini，来保存用户名和密码的信息。



1. 在资源目录下建立一个shiro.ini的文件；
2. 修改文件的内容，本次需要追加的是用户信息：

|  |
| --- |
| **[users]** admin=123456 kuhnwei=650901 |

本处定义了两个用户：admin/hello、kuhnwei/world，前面为身份信息，而后面为认证信息。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的认证编写：本次的操作一定要通过shiro.ini文件进行信息的取得：
   1. 如果要想进行shiro的认证信息的读取，那么首先需要使用一个org.apache.shiro.util.Factory接口，在这个接口里面定义有取得一个SecurityManager接口对象的方法：public T getInstance();
      1. 此方法可以取得一个org.apache.shiro.mgt.SecurityManager接口对象；
      2. Factory是接口，本次将通过一个ini文件进行读取，所以应该使用“org.apache.shiro.config.IniSecurityManagerFactory”子类，这个子类里面只需要关注构造方法：public IniSecurityManagerFactory(String iniResourcePath)，资源定位通过classpath取得；

|  |
| --- |
| // 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"); |

* 1. Factory接口中保管的是SecurityManger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，也就是说用户读取进来的资源文件在这个接口里面将转换为所有的认证数据（用户名、密码）；

|  |
| --- |
|  |

* 1. 具体的认证信息一定要通过Subject接口来实现，而SecurityManager只是一个综合的认证信息集合，所以为了解决这个问题可以使用“org.apache.shiro.SecurityUtils”，在这个类中有如下两个方法：
     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();

|  |
| --- |
| // 利用一个专门的认证操作的处理类，实现认证处理的具体实现 SecurityUtils.*setSecurityManager*(securityManager); |
| // 获取进行用户名和密码认证的接口对象 Subject subject = SecurityUtils.*getSubject*(); |

* 1. 本次使用的用户名和密码的组合进行的认证处理，所以需要将用户名和密码包装为一个Token，因为在Subject接口下定义有如下的方法：
     1. 登录验证：publict void login(AuthenticationToken token) throws AuthenticationException;
     2. 登录注销：public void logout()；
  2. 既然要进行简单的认证，那么就输入用户名和密码即可，所以使用“org.apache.shiro.authc.UsernamePasswordToken”，只需要创建这个类的对象，创建的时候传递用户名和密码即可实现：

|  |
| --- |
| // 定义了一个Token，里面保存要登录的用户名和密码信息 UsernamePasswordToken token = new UsernamePasswordToken("admin", "123456"); |

* 1. 利用Subject实现用户的登录处理操作；

|  |
| --- |
| package com.kuhnwei.examples.shiro; import org.apache.shiro.SecurityUtils; import org.apache.shiro.authc.UsernamePasswordToken; import org.apache.shiro.config.IniSecurityManagerFactory; import org.apache.shiro.mgt.SecurityManager; import org.apache.shiro.subject.Subject; import org.apache.shiro.util.Factory; */\*\*  \** ***@author*** *Kuhn Wei  \** ***@email*** *email@kuhnwei.com  \** ***@created*** *2017/12/2 16:35  \*/* public class LoginDemo {  public static void main(String[] args) {  // 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  UsernamePasswordToken token = new UsernamePasswordToken("admin", "123456");  // 实现用户登录处理  subject.login(token);  // 输出用户名  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实现登录控制，而如果说你的用户名或密码出现了错误，那么程序就将抛出异常，实际上异常的内容比较多：

* 1. 密码输入错误，但是用户名正确：

|  |
| --- |
| Exception in thread "main" org.apache.shiro.authc.IncorrectCredentialsException: Submitted credentials for token [org.apache.shiro.authc.UsernamePasswordToken - admin, rememberMe=false] did not match the expected credentials. |

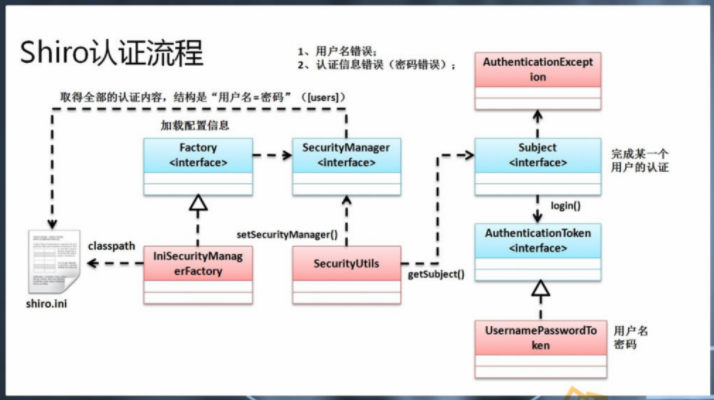
* 1. 用户名错误，密码输入正确：

|  |
| --- |
| Exception in thread "main" org.apache.shiro.authc.UnknownAccountException: Realm [org.apache.shiro.realm.text.IniRealm@3fee9989] was unable to find account data for the submitted AuthenticationToken [org.apache.shiro.authc.UsernamePasswordToken - admin1, rememberMe=false]. |

在shiro里面如果账户出现了错误，那么密码将不会验证，对于这一点一定要清楚，否则在使用自定义数据库认证的时候流程上就会出现问题。

## Shiro认证流程

通过之前的演示实际上就可以发现，整个的Shiro实现的只是一个用户的登录检测，但是在整个过程里面牵扯到了许多的接口以及子类。



在整个程序里面，必须要进行验证数据的信息加载，而后所有的认证信息交由SecurityMnager管理，而用户要想进行用户名或密码的认证操作必须通过Subject接口完成，随后使用AuthenticationToken保存所有的认证信息，本处为普通的用户名和密码，所以使用了一个UsernamePasswordToken子类进行配置。

如果登录成功则不会抛出异常，如果登录失败，则会根据失败的类型抛出不同的异常子类。

## 总结

整体的Shiro的认证流程都是固话模式。

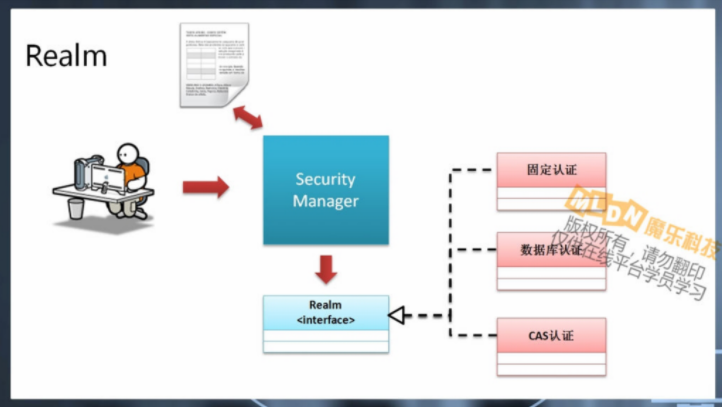
# Realm

在之前的认证都是基于一个配置文件完成的，但是在整个shiro之中，对于用户认证信息可能来源于各种途径，那么要想实现不同途径的用户身份认证就需要了解Realm概念。

## 认识Realm

在Shiro里面考虑到以后可能面对于不同的用户数据的来源，所以为了统一这些来源的处理，准备了一个Realm接口，首先来观察org.apache.shiro.realm.Realm接口定义：

|  |
| --- |
| package org.apache.shiro.realm; import org.apache.shiro.authc.AuthenticationException; import org.apache.shiro.authc.AuthenticationInfo; import org.apache.shiro.authc.AuthenticationToken; interface Realm {  */\*\*  \* 只是要求返回一个当前使用的Realm名字，这个名字可以任意返回，但是不要重名  \** ***@return*** *\*/* String getName();  */\*\*  \* 判断你当前使用的Token的类型是否为指定的类型  \** ***@param*** *token  \** ***@return*** *\*/* boolean supports(AuthenticationToken token);  */\*\*  \* 得到用户的认证信息，根据传入的Token取得  \** ***@param*** *token 包含了要进行验证的所有数据  \** ***@return*** *\** ***@throws*** *AuthenticationException  \*/* AuthenticationInfo getAuthenticationInfo(AuthenticationToken token) throws AuthenticationException; } |



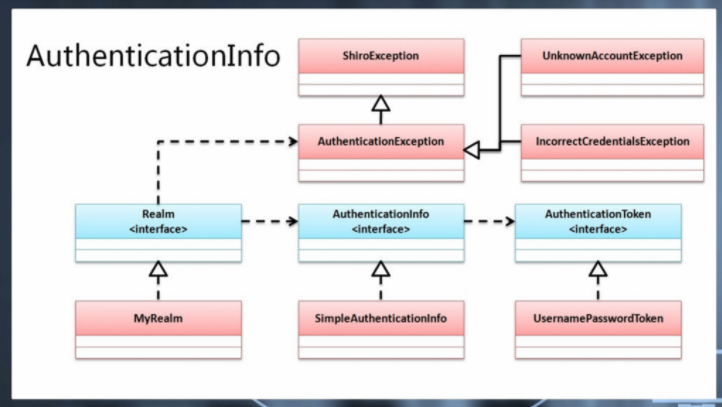
## 固定信息Realm认证

如果要想使用Realm操作，那么必须要保证有一个具体的认证类实现了Realm接口。

1. 建立一个ExampleRealm子类，改类实现Realm接口：
   1. 认证信息：org.apache.shiro.authc.AuthenticationInfo，接口有两个方法：
      1. 取得所有的身份信息：public PrincipalCollection getPrincipals()；
      2. 取得认证信息：public Object getCredentials()；
   2. 子类：org.apache.shiro.authc.SimpleAuthenticationInfo；
      1. 构造方法：public SimpleAuthenticationInfo(Object principal, Object credentials, String realmName);
   3. 所有用户的信息都保存在token里面：org.apache.shiro.authc.AuthenticationToken，此接口定义有如下两个方法：
      1. 身份数据：public Object getPrincipal();
      2. 认证数据：public Object getCredentials();
         1. 此时返回的数据为char[]数组，不能直接转换为String;

范例：实现自定义Realm

|  |
| --- |
| package com.kuhnwei.examples.shiro; import org.apache.shiro.authc.\*; import org.apache.shiro.realm.Realm; */\*\*  \** ***@author*** *Kuhn Wei  \** ***@email*** *email@kuhnwei.com  \** ***@created*** *2017/12/2 17:28  \*/* public class ExampleRealm implements Realm{  @Override  public String getName() {  return "ExampleRealm";  }  @Override  public boolean supports(AuthenticationToken token) {  // 本次将在之前程序的基础之上继续使用UsernamePasswordToken完成信息的传递  return token instanceof UsernamePasswordToken;  }  @Override  public AuthenticationInfo getAuthenticationInfo(AuthenticationToken token) throws AuthenticationException {  // 取得用户名  String username = (String) token.getPrincipal();  // 取得密码，在Shiro里面是利用字符数组实现了密码的传递，所以不能讲其直接变为String  String password = new String((char[]) token.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 (!"admin".equals(username)) {  throw new UnknownAccountException("用户名不存在！");  }   if (!"123456".equals(password)) {  throw new IncorrectCredentialsException("密码输入错误！");  }  return new SimpleAuthenticationInfo(username, password, this.getName());  } } |



1. 如果现在希望Realm配置生效，则需要在配置文件里面进行相关定义；

范例：修改shiro.ini文件

|  |
| --- |
| exampleRealm=com.kuhnwei.examples.shiro.ExampleRealm # 整个Shiro中的验证处理都是由SecurityManager接口负责的 securityManager.realms=$exampleRealm |

这个时候一定要注意，如果有多个Realm，则在“securityManager.realms”后面可以设置多个名称，中间使用“,”间隔。

1. 使用该Realm进行数据的认证处理：

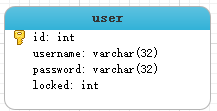
|  |
| --- |
| package com.kuhnwei.examples.shiro; import org.apache.shiro.SecurityUtils; import org.apache.shiro.authc.UsernamePasswordToken; import org.apache.shiro.config.IniSecurityManagerFactory; import org.apache.shiro.mgt.SecurityManager; import org.apache.shiro.subject.Subject; import org.apache.shiro.util.Factory; */\*\*  \** ***@author*** *Kuhn Wei  \** ***@email*** *email@kuhnwei.com  \** ***@created*** *2017/12/2 16:35  \*/* public class LoginDemo {  public static void main(String[] args) {  // 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  UsernamePasswordToken token = new UsernamePasswordToken("admin", "123456");  // 实现用户登录处理  subject.login(token);  // 输出用户名  System.*out*.println(subject.getPrincipal());  } } |

那么此时所有的认证的操作信息都通过自定义的Realm实现了。

## 基于数据库的验证处理

现在希望可以建立一张用户数据表，而后基于数据库实现数据的认证处理操作。

1. 需要建立一张用户表：user



|  |
| --- |
| **DROP TABLE** IF **EXISTS** `user`; **CREATE TABLE** `user` (  `id` **int**(10) unsigned **NOT NULL** AUTO\_INCREMENT,  `username` **varchar**(32) **NOT NULL**,  `password` **varchar**(32) **NOT NULL**,  `locked` **int**(11) **DEFAULT NULL**,  **PRIMARY KEY** (`id`) ) ENGINE=InnoDB AUTO\_INCREMENT=5 **DEFAULT** CHARSET=utf8;  **INSERT INTO** `user` **VALUES** ('1', 'kuhnwei', '650901', '0'); **INSERT INTO** `user` **VALUES** ('2', 'admin', '123', '0'); **INSERT INTO** `user` **VALUES** ('3', 'emp', '123', '0'); **INSERT INTO** `user` **VALUES** ('4', 'test', '123', '0'); |

1. 由于项目之中需要使用到mysql数据库，所以需要下载mysql的驱动程序；

|  |
| --- |
| <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  <version>5.1.34</version> </dependency> |

1. 本次的处理操作所有的核心配置都不是通过程序完成的，都是通过配置文件完成的，本次希望可直接使用MySQL提供好的数据库连接池配置；

范例：修改shiro.ini的文件

|  |
| --- |
| **[main]** # 定义本次MySQL连接要哦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=localhost # 定义要使用的数据库名字 dataSource.databaseName=examples # 定义数据库的连接账户 dataSource.user=root # 定义数据库的连接密码 #dataSource.password=  # 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 user WHERE username=?  # 配置安全管理器所使用的Realm securityManager.realms=$jdbcRealm |

此时继续使用之前的程序进行执行，这个时候利用JDBC实现用户的登录检测操作。

## 总结

Realm是所有的认证的操作标准。

# 认证授权

授权也称为访问控制，即：某一个资源库中的那些资源允许当前用户来进行处理操作，而在整个的授权之中实际上会包含以下几个概念：授权主体（Subject）、资源（Resource）、权限（Permission）、角色（Role）。

## 固定角色

所谓的固定角色操作指的是直接利用配置文件进行操作实现。该实现要求在“[users]”下进行，而且配置的形式为：“用户名=密码,角色,角色,...”

1、定义一个shiro.ini的文件：

|  |
| --- |
| **[users]** admin=123456,administrator,dept kuhnwei=650901,user,dept |

在当前的两个用户下：admin用户具备有administrator与dept的操作角色，而kuhnwei用户具备有user与dept的操作角色。

2、角色检测，在shiro之中Subject接口描述的四用户的主体信息，所以针对于角色的认证处理都在此接口中提供方法，包括如下方法：

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **方法名称** | **类型** | **描述** |
| 1 | public void checkRole(String roleIdentifier) throws AuthorizationException | 普通 | 检测一个角色是否存在 |
| 2 | public void checkRoles(String... roleIdentifiers) throws AuthorizationException | 普通 | 检测多个角色是否存在 |
| 3 | public boolean hasRole(String roleIdentifier) | 普通 | 判断是否有指定的角色 |
| 4 | public boolean hasAllRoles(Collection<String> roleIdentifiers) | 普通 | 判断是否有所有的角色 |

发现以hasXxx()命名的操作方法是不会抛出异常的，但是会有明确的boolean型返回结果，而已checkXxx()开头的会抛出异常，并且没有返回结果。

范例：检测角色

* 包含指定的角色，则程序可以正常执行完毕：

|  |
| --- |
| subject.checkRole("administrator"); |

* 不好汉指定的角色，则程序就会出现异常：

|  |
| --- |
| subject.checkRole("user"); |
| Exception in thread "main" org.apache.shiro.authz.UnauthorizedException: Subject does not have role [user]  at org.apache.shiro.authz.ModularRealmAuthorizer.checkRole(ModularRealmAuthorizer.java:421) |

范例：检测角色

|  |
| --- |
| System.*out*.println(subject.hasRole("dept")); |

范例：判断多个角色

|  |
| --- |
| Set<String> allRoles = new HashSet<String>(); allRoles.add("user"); allRoles.add("dept"); System.*out*.println(subject.hasAllRoles(allRoles)); |

整个的处理里面发现用户的登录和角色的认证是分两个部分完成的。

## 权限认证

可以取得了角色数据，那么就意味着可以进行权限的取得，那么基本的流程关系就是，一个角色对应有多个权限。

1. 要想进行角色中所包含的权限配置，那么可以直接在shiro.ini文件里面进行定义：
   1. 用户和角色：“用户名=密码,角色,角色,角色,...”；
   2. 角色和权限：“角色=权限,权限,...”

范例：定义shiro.ini文件

|  |
| --- |
| **[users]** admin=123456,administrator,dept kuhnwei=650901,user,dept **[roles]** administrator=admin:add,admin:edit,admin:remove dept=dept:add,dept:edit,dept:list user=user:list |

1. 对于权限的判断处理依然在Subject接口之中完成，在此接口里面定义有如下的权限认证方法：

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **方法名称** | **类型** | **描述** |
| 1 | public void checkPermission(String permission) throws AuthorizationException | 普通 | 检测权限是否存在 |
| 2 | public void checkPermissions(String... permissions) throws AuthorizationException | 普通 | 检测多个权限是否存在 |
| 3 | public boolean isPermitted(String permission) | 普通 | 判断是否有指定的一个权限 |
| 4 | public boolean isPermittedAll (String... permission) | 普通 | 判断是否有指定的多个角色 |

范例：进行权限的认证（没有此权限）

|  |
| --- |
| subject.checkPermissions("admin:add","dept:add","dept:edit","dept:list"); |
| Exception in thread "main" org.apache.shiro.authz.UnauthorizedException: Subject does not have permission [admin:add] |

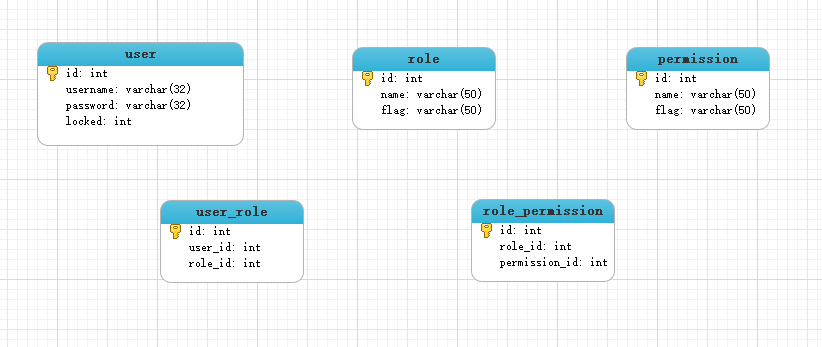
范例：测试权限

|  |
| --- |
| System.*out*.println(subject.isPermitted("dept:edit")); |

## 基于数据库的授权认证

从一个正常的设计角度来讲，用户、角色、权限三者的关系属于两个多对多，即：

* 一个用户拥有多个角色，一个角色属于多个用户；
* 一个角色拥有多个权限，一个权限属于多个角色。



范例：数据库创建脚本

|  |
| --- |
| **DROP TABLE** IF **EXISTS** `user`; **CREATE TABLE** `user` (  `id` **int**(10) unsigned **NOT NULL** AUTO\_INCREMENT,  `username` **varchar**(32) **NOT NULL**,  `password` **varchar**(32) **NOT NULL**,  `locked` **int**(11) **DEFAULT NULL**,  **PRIMARY KEY** (`id`) ) ENGINE=InnoDB AUTO\_INCREMENT=5 **DEFAULT** CHARSET=utf8;  **INSERT INTO** `user` **VALUES** ('1', 'kuhnwei', '650901', '0'); **INSERT INTO** `user` **VALUES** ('2', 'admin', '123', '0'); **INSERT INTO** `user` **VALUES** ('3', 'emp', '123', '0'); **INSERT INTO** `user` **VALUES** ('4', 'test', '123', '0');  **DROP TABLE** IF **EXISTS** `role`; **CREATE TABLE** `role` (  `id` **int**(10) unsigned **NOT NULL** AUTO\_INCREMENT,  `name` **varchar**(50) **NOT NULL**,  `flag` **varchar**(50) **DEFAULT NULL**,  **PRIMARY KEY** (`id`) ) ENGINE=InnoDB AUTO\_INCREMENT=4 **DEFAULT** CHARSET=utf8;  **INSERT INTO** `role` **VALUES** ('1', '超管', 'administrator'); **INSERT INTO** `role` **VALUES** ('2', '部门管理', 'dept'); **INSERT INTO** `role` **VALUES** ('3', '用户', 'user');  **DROP TABLE** IF **EXISTS** `user\_role`; **CREATE TABLE** `user\_role` (  `id` **int**(10) unsigned **NOT NULL** AUTO\_INCREMENT,  `user\_id` **int**(10) unsigned **NOT NULL**,  `role\_id` **int**(10) unsigned **NOT NULL**,  **PRIMARY KEY** (`id`) ) ENGINE=InnoDB AUTO\_INCREMENT=6 **DEFAULT** CHARSET=utf8;  **INSERT INTO** `user\_role` **VALUES** ('1', '1', '2'); **INSERT INTO** `user\_role` **VALUES** ('2', '1', '3'); **INSERT INTO** `user\_role` **VALUES** ('3', '2', '1'); **INSERT INTO** `user\_role` **VALUES** ('4', '2', '2'); **INSERT INTO** `user\_role` **VALUES** ('5', '4', '3');  **DROP TABLE** IF **EXISTS** `permission`; **CREATE TABLE** `permission` (  `id` **int**(10) unsigned **NOT NULL** AUTO\_INCREMENT,  `name` **varchar**(50) **NOT NULL**,  `flag` **varchar**(50) **DEFAULT NULL**,  **PRIMARY KEY** (`id`) ) ENGINE=InnoDB AUTO\_INCREMENT=8 **DEFAULT** CHARSET=utf8;  **INSERT INTO** `permission` **VALUES** ('1', '超管执行添加操作', 'admin:add'); **INSERT INTO** `permission` **VALUES** ('2', '超管执行编辑操作', 'admin:edit'); **INSERT INTO** `permission` **VALUES** ('3', '超管执行删除操作', 'admin:remove'); **INSERT INTO** `permission` **VALUES** ('4', '部门增加', 'dept:add'); **INSERT INTO** `permission` **VALUES** ('5', '部门编辑', 'dept:edit'); **INSERT INTO** `permission` **VALUES** ('6', '部门列表', 'dept:list'); **INSERT INTO** `permission` **VALUES** ('7', '用户列表', 'user:list');  **DROP TABLE** IF **EXISTS** `role\_permission`; **CREATE TABLE** `role\_permission` (  `id` **int**(10) unsigned **NOT NULL** AUTO\_INCREMENT,  `role\_id` **int**(10) unsigned **NOT NULL**,  `permission\_id` **int**(10) unsigned **NOT NULL**,  **PRIMARY KEY** (`id`) ) ENGINE=InnoDB AUTO\_INCREMENT=8 **DEFAULT** CHARSET=utf8;  **INSERT INTO** `role\_permission` **VALUES** ('1', '1', '1'); **INSERT INTO** `role\_permission` **VALUES** ('2', '1', '2'); **INSERT INTO** `role\_permission` **VALUES** ('3', '1', '3'); **INSERT INTO** `role\_permission` **VALUES** ('4', '2', '4'); **INSERT INTO** `role\_permission` **VALUES** ('5', '2', '5'); **INSERT INTO** `role\_permission` **VALUES** ('6', '2', '6'); **INSERT INTO** `role\_permission` **VALUES** ('7', '3', '7'); |

1、为了方便进行登录与授权处理，编写一个单独的程序类，这个类也可以不做业务层或数据层的划分了，只是做一个简单的功能类，此类可以取得用户的信息以及角色和权限数据。  
 - 定义一个描述User数据表的Java类

|  |
| --- |
| package com.kuhnwei.examples.shiro.domain; import java.io.Serializable;public class User implements Serializable {  private Integer id;  private String username;  private String password; } |

- 定义一个业务层的处理类：UserLoginService:

|  |
| --- |
| package com.kuhnwei.examples.shiro; import com.kuhnwei.examples.shiro.domain.User; import java.sql.\*; import java.util.HashSet; import java.util.Set;public class UserLoginService {  private static final String *DB\_DRIVER* = "com.mysql.jdbc.Driver";  private static final String *DB\_URL* = "jdbc:mysql://localhost:3306/examples";  private static final String *DB\_USER* = "root";  private static final String *DB\_PASSWORD* = "";  private Connection conn;  private PreparedStatement pstmt = null;  public UserLoginService() {  this.connectionDataBase();  }  */\*\*  \* 根据用户名获取用户对象  \** ***@param*** *username 用户名  \** ***@return*** *用户  \*/* public User get(String username) {  User user = null;  try {  String sql = "SELECT id, username, password FROM user WHERE username=?";  this.pstmt = this.conn.prepareStatement(sql);  this.pstmt.setString(1, username);  ResultSet rs = this.pstmt.executeQuery();  if (rs.next()) {  user = new User();  user.setId(rs.getInt(1));  user.setUsername(rs.getString(2));  user.setPassword(rs.getString(3));  }  } catch (Exception e) {  e.printStackTrace();  }  return user;  }  */\*\*  \* 根据用户查询拥有的所有角色数据  \** ***@param*** *user 用户  \** ***@return*** *角色数据  \*/* public Set<String> listRolesByUser(User user) {  Set<String> allRoles = new HashSet<String>();  String sql = "SELECT flag FROM role WHERE id " +  "IN ( SELECT role\_id FROM user\_role WHERE user\_id=?)";  try {  this.pstmt = this.conn.prepareStatement(sql);  this.pstmt.setInt(1, user.getId());  ResultSet rs = this.pstmt.executeQuery();  while (rs.next()) {  allRoles.add(rs.getString(1));  }  } catch (Exception e) {  e.printStackTrace();  }  return allRoles;  }  */\*\*  \* 根据用户查询出该用户对应的所有权限数据  \** ***@param*** *user 用户  \** ***@return*** *权限数据  \*/* public Set<String> listPermissionsByUser(User user) {  Set<String> allPermissions = new HashSet<String>();  String sql = "SELECT flag FROM permission " +  "WHERE id IN (SELECT permission\_id FROM role\_permission " +  "WHERE role\_id IN (SELECT role\_id FROM user\_role WHERE user\_id=?))";  try {  this.pstmt = this.conn.prepareStatement(sql);  this.pstmt.setInt(1, user.getId());  ResultSet rs = this.pstmt.executeQuery();  while (rs.next()) {  allPermissions.add(rs.getString(1));  }  } catch (Exception e) {  e.printStackTrace();  }  return allPermissions;  }  public void close() {  if (this.conn != null) {  try {  this.conn.close();  } catch (SQLException e) {  e.printStackTrace();  }  }  }  private void connectionDataBase() {  try {  Class.*forName*(*DB\_DRIVER*);  this.conn = DriverManager.*getConnection*(*DB\_URL*, *DB\_USER*, *DB\_PASSWORD*);  } catch (Exception e) {  e.printStackTrace();  }  } } |

最终这个程序类所取得的全部的角色和权限数据都是需要交给Shiro判断的，在整个的处理过程之中，它的判断是利用Set集合完成的，所以必须返回Set<String>集合。

2、此时的操作不再是简单的用户登录了，但是现在除了用户登录之外还牵扯到权限的操作，所以整个的自定义的Realm类就必须更换一个父类：

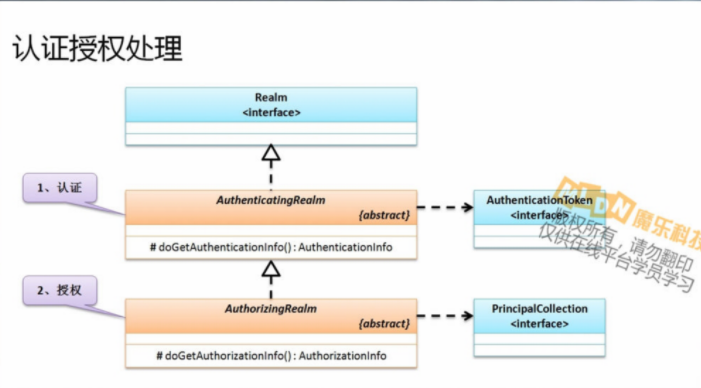
- 认证处理类：AuthenticatingRealm

抽象方法：protected abstract AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken token) throws AuthenticationException

- 授权处理类：AuthorizingRealm

抽象方法：protected abstract AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principals)

通过PrincipalCollection可以取得身份信息：public Object getPrimaryPrincipal()



范例：自定义认证授权Realm

|  |
| --- |
| package com.kuhnwei.examples.shiro; import com.kuhnwei.examples.shiro.domain.User; import org.apache.shiro.authc.\*; import org.apache.shiro.authz.AuthorizationInfo; import org.apache.shiro.authz.SimpleAuthorizationInfo; import org.apache.shiro.realm.AuthorizingRealm; import org.apache.shiro.subject.PrincipalCollection; */\*\*  \** ***@author*** *Kuhn Wei  \** ***@email*** *email@kuhnwei.com  \** ***@created*** *2017/12/2 17:28  \*/* public class ExampleRealm extends AuthorizingRealm {    @Override  protected AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principals) {  System.*out*.println("------------------------2、用户角色与权限：doGetAuthorizationInfo------------------------------");  // 取得用户登录名  String username = (String) principals.getPrimaryPrincipal();  // 定义授权信息的返回数据  SimpleAuthorizationInfo auth = new SimpleAuthorizationInfo();  // 进行业务层处理  UserLoginService service = new UserLoginService();  User user = service.get(username);  auth.setRoles(service.listRolesByUser(user));  auth.setStringPermissions(service.listPermissionsByUser(user));  service.close();  return auth;  }  @Override  protected AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken token) throws AuthenticationException {  System.*out*.println("-------------------------1、用户登录认证：doGetAuthenticationInfo-----------------------------------------");  // 登录认证的方法需要先执行，需要用它来判断登录的用户信息是否合法  String username = (String) token.getPrincipal();  // 需要通过用户名取得用户的完整信息，利用业务层操作  UserLoginService service = new UserLoginService();  // 需要取得的是用户的信息  User user = service.get(username);  service.close();  if (user == null) {  throw new UnknownAccountException("用户不存在！");  } 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 (user.getPassword().equals(password)) {  AuthenticationInfo auth = new SimpleAuthenticationInfo(username, password, "exampleRealm");  return auth;  } else {  throw new IncorrectCredentialsException("密码错误！");  }  }  } } |

3、需要在shiro.ini文件里面定义相关的Realm配置：

|  |
| --- |
| **[main]** jdbcRealm=com.kuhnwei.examples.shiro.ExampleRealm securityManager.realms=$jdbcRealm |

4、编写测试程序：

|  |
| --- |
| package com.kuhnwei.examples.shiro; import org.apache.shiro.SecurityUtils; import org.apache.shiro.authc.UsernamePasswordToken; import org.apache.shiro.config.IniSecurityManagerFactory; import org.apache.shiro.mgt.SecurityManager; import org.apache.shiro.subject.Subject; import org.apache.shiro.util.Factory;public class LoginDemo {  public static void main(String[] args) {  // 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  UsernamePasswordToken token = new UsernamePasswordToken("kuhnwei", "650901");  // 实现用户登录处理  subject.login(token);  System.*out*.println(subject.hasRole("dept"));  System.*out*.println("---------------------------------------");  System.*out*.println(subject.isPermitted("dept:add"));  } } |
| -------------------------1、用户登录认证：doGetAuthenticationInfo------------------------  ------------------------2、用户角色与权限：doGetAuthorizationInfo------------------------  true  ---------------------------------------  ------------------------2、用户角色与权限：doGetAuthorizationInfo------------------------  true |

## 总结

Realm就是整个Shiro的核心实现。

# 在WEB中使用Shiro

如果要使用Shiro那么关键的部分还是需要整合WEB的开发，所以针对于WEB的开发，Shiro本身也是可以轻松进行的。

## 基础的验证配置

如果要想在WEB项目里面整合Shiro，那么久必须配置相应的Shiro的开发包。

1. 修改pom.xml文件；

|  |
| --- |
| <dependency>  <groupId>org.apache.shiro</groupId>  <artifactId>shiro-web</artifactId>  <version>1.3.2</version> </dependency> |

1. 如果要想在WEB之中去因公Shiro就必须配置监听器；

范例：修改web.xml文件追加监听器

参考信息：<http://shiro.apache.org/webapp-tutorial.html>

|  |
| --- |
| <listener>  <listener-class>org.apache.shiro.web.env.EnvironmentLoaderListener</listener-class> </listener> |

1. 在整个的WEB的开发过程之中用户的检测一定需要存在有过滤器，所以需要配置过滤器。本次建议继续使用普通的认证模式（基于JDBC实现），利用shiro.ini文件实现认证，所以这个配置文件需要进行指派。

|  |
| --- |
| <filter>  <filter-name>ShiroFilter</filter-name>  <filter-class>org.apache.shiro.web.servlet.ShiroFilter</filter-class>  <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> |

那么此时就表示WEB项目之中已经集成好了Shiro的开发程序。

配置完成此时可以得到一些shiro的配置信息：

|  |
| --- |
|  |

1. 建立一个Servlet程序来进行具体的登录操作处理。

|  |
| --- |
| package com.kuhnwei.examples.shiro; import org.apache.shiro.SecurityUtils; import org.apache.shiro.authc.UsernamePasswordToken; import org.apache.shiro.subject.Subject; import javax.servlet.ServletException; import javax.servlet.annotation.WebServlet; import javax.servlet.http.HttpServlet; import javax.servlet.http.HttpServletRequest; import javax.servlet.http.HttpServletResponse; import java.io.IOException;@WebServlet("/shiroLogin") public class LoginServlet extends HttpServlet {  @Override  protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {  this.doPost(request, response);  }  @Override  protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {  String username = request.getParameter("username");  String password = request.getParameter("password");  Subject subject = SecurityUtils.*getSubject*();  UsernamePasswordToken token = new UsernamePasswordToken(username, password);  subject.login(token);  request.getRequestDispatcher("/pages/welcome.jsp").forward(request, response);  } } |

1. 在根目录下创建一个login.jsp的文件；

|  |
| --- |
| <form action="shiroLogin" method="post">  用户名：<input type="text" name="username" id="username"><br>  密&nbsp;码：<input type="password" name="password" id="password"><br>  <input type="submit" value="Login"> </form> |

1. 创建“/pages/welcome.jsp”文件，只是提示一个欢迎信息；
2. 修改shiro.ini文件，在这个文件里面需要进行相关的处理配置；

|  |
| --- |
| **[main]** # 如果现在认证失败，则跳转到loginUrl配置的路径 authc.loginUrl=/login.jsp jdbcRealm= com.kuhnwei.examples.shiro.ExampleRealm securityManager.realms=$jdbcRealm # 配置所有需要进行路径检测的页面 **[urls]** # 登录的页面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 |

### Shiro过滤检测

|  |  |  |
| --- | --- | --- |
| **No.** | **授权** | **类名称** |
| 1 | anon | org.apache.shiro.web.filter.authc.AnonymousFilter |
| 2 | authc | org.apache.shiro.web.filter.authc.FormAuthenticationFilter |
| 3 | authcBasic | org.apache.shiro.web.filter.authc.BasicHttpAuthenticationFilter |
| 4 | perms | org.apache.shiro.web.filter.authz.PermissionsAujthorizationFilter |
| 5 | port | org.apache.shiro.web.filter.authz.PortFilter |
| 6 | rest | org.apache.shiro.web.filter.authz.HttpMethodPermissionFilter |
| 7 | roles | org.apache.shiro.web.filter.authz.RolesAuthorizationFIlter |
| 8 | ssl | org.apache.shiro.web.filter.authz.SslFilter |
| 9 | user | org.apache.shiro.web.filter.authc.UserFilter |

## 角色授权检测

授权检测主要分为两个方面：角色、权限。一个用户可以同时具备有多个角色，一个角色可以有多个权限。

1. 实现角色的检测处理

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 # 当角色认证失败之后的跳转页面 roles.unauthorizedUrl=/role.jsp jdbcRealm= com.kuhnwei.examples.shiro.ExampleRealm securityManager.realms=$jdbcRealm # 配置所有需要进行路径检测的页面 **[urls]** # 登录的页面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[dept] |

如果此时是存在有操作的角色的，那么访问指定路径的时候就可以进行过滤检测，而如果你现在没有角色，那么在访问指定路径时会跳转到一个错误页上。需要记住的时候，页可以配置多个角色。

|  |
| --- |
| /pages/welcome.jsp=authc,roles[dept],roles[user] |

此时角色的操作属于或的关系。

## 权限授权检测

如果要进行权限的认证的判断处理，那么也应该提供有自己的错误页面标记。这个时候依然需要在shiro.ini文件里面进行配置。

范例：修改shiro.ini

|  |
| --- |
| **[main]** # 如果现在认证失败，则跳转到loginUrl配置的路径 authc.loginUrl=/login.jsp # 当角色认证失败之后的跳转页面 roles.unauthorizedUrl=/role.jsp jdbcRealm= com.kuhnwei.examples.shiro.ExampleRealm securityManager.realms=$jdbcRealm # 配置所有需要进行路径检测的页面 **[urls]** # 登录的页面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[dept],roles[user] # 对指定页面登录之后进行权限的检测处理 /pages/welcome.jsp=authc,perms[admin:add],perms[dept:add] |

此操作的流程除了使用的过滤器的标记不同之外，基本的操作形式和之前的角色是完全相同的。

## shiro标签支持

现在已经完成了整体的路径的配置，但是在一个正常的项目开发过程之中，对于前台的JSP页面可以使用的链接地址，实际上也需要进行动态配置，其中就需要根据用户的角色和权限来判断是否出现指定的菜单项，所以此时需要进行标签的应用。

如果要想在开发代码之中使用shiro标签，则应该引入一下的标签信息：

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

而后在shiro里面提供有如下的标签内容：

* 取得用户名：<shiro:principal/>;
  + 如果要想在：Servlet里面取得用户名：SecurityUtils.getSubject().getPrincipal();
* 是否具备有指定的角色：<shiro:hasRole name=”角色名称”></shiro:hasRole>
* 是否具备有任意一个角色：<shiro:hasAnyRoles name=”角色,角色”></shiro:hasAnyRoles>
* 是否具备有指定权限：<shiro:hasPermission name=”权限”></shiro:hasPermission>
* 不具备有指定的角色：<shiro:lacksRole name=”角色”></shiro:lacksRole>
* 不具备有指定权限：<shiro:lacksPermission name=“权限”></shiro:lacksPermission>
* 判断是否认证：<shiro:authenticated></shiro:authenticated>
* 用户未认证过：<shiro:notAuthenticated></shiro:notAuthenticated>
* 判断当前用户是否为游客：<shiro:guest></shiro:guest>
* 取得rememberme中的信息：<shiro:user/>

## 路径通配符

在很多的时候如果要进行角色或权限认证，可能会针对于某一个资源目录下的所有内容进行角色或权限认证。而在shiro里面也支持Ant风格的通配符：

* “?”：匹配任意的一个字符，例如“/admin?”可以匹配“/admin1”、“/admin2”;
* “\*”：匹配一个或多个任意的字符；
* “\*\*”：匹配零个或多个目录；

## 总结

整个的Shiro在进行处理的时候都采用了异常方便的配置模式，所以这也就是shiro在整个项目开发之中被应用如此广泛的主要原因。

# Spring整合Shiro

Spring+Mybatis+Shiro的整合是现在开发中所采用的主流模式，也是SSM的真正含义。

## 搭建开发环境

本次需要配置Spring、MyBatis的开发包到Maven之中。

1. 修改pom.xml文件追加相关的开发包的内容；
   1. 配置Spring有关的开发程序包；

|  |
| --- |
| <!-- 导入Spring有关的开发包 --> <dependency>  <groupId>commons-io</groupId>  <artifactId>commons-io</artifactId>  <version>2.5</version> </dependency> <dependency>  <groupId>commons-fileupload</groupId>  <artifactId>commons-fileupload</artifactId>  <version>1.3.2</version> </dependency> <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-core</artifactId>  <version>4.3.3.RELEASE</version> </dependency> <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-beans</artifactId>  <version>4.3.3.RELEASE</version> </dependency> <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-tx</artifactId>  <version>4.3.3.RELEASE</version> </dependency> <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-context</artifactId>  <version>4.3.3.RELEASE</version> </dependency> <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-context-support</artifactId>  <version>4.3.3.RELEASE</version> </dependency> <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-web</artifactId>  <version>4.3.3.RELEASE</version> </dependency> <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-webmvc</artifactId>  <version>4.3.3.RELEASE</version> </dependency> <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-aop</artifactId>  <version>4.3.3.RELEASE</version> </dependency> <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-aspects</artifactId>  <version>4.3.3.RELEASE</version> </dependency> <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-jdbc</artifactId>  <version>4.3.3.RELEASE</version> </dependency> <dependency>  <groupId>c3p0</groupId>  <artifactId>c3p0</artifactId>  <version>0.9.1.2</version> </dependency> |

* 1. 配置mybatis的开发包；

|  |
| --- |
| <!-- mybatis开发包--> <dependency>  <groupId>org.mybatis</groupId>  <artifactId>mybatis</artifactId>  <version>3.4.1</version> </dependency> <dependency>  <groupId>org.mybatis</groupId>  <artifactId>mybatis-spring</artifactId>  <version>1.3.0</version> </dependency> |

* 1. 配置shiro与spring的整合开发包；

|  |
| --- |
| <!-- shiro与spring整合的开发包--> <dependency>  <groupId>org.apache.shiro</groupId>  <artifactId>shiro-spring</artifactId>  <version>1.3.2</version> </dependency> |

配置完成之后暂时先不要进行后续的配置，建议先启动Tomcat测试一下当前的开发包是否正确。

1. 配置mybatis.cfg.xml文件

|  |
| --- |
| <?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>  <!--进行mybatis相应环境的属性定义-->  <settings>  <!-- 在本项目之中开启二级缓存 -->  <setting name="cacheEnabled" value="true"/>  </settings>  <typeAliases>  <package name="com.kuhnwei.examples.shiro.domain"/>  </typeAliases> </configuration> |

1. 配置spring-\*.xml文件：

|  |
| --- |
| <?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:aop="http://www.springframework.org/schema/aop"  xmlns:tx="http://www.springframework.org/schema/tx"  xmlns:context="http://www.springframework.org/schema/context"  xsi:schemaLocation="  http://www.springframework.org/schema/beans  http://www.springframework.org/schema/beans/spring-beans.xsd  http://www.springframework.org/schema/context  http://www.springframework.org/schema/context/spring-context.xsd  http://www.springframework.org/schema/aop  http://www.springframework.org/schema/aop/spring-aop.xsd  http://www.springframework.org/schema/tx  http://www.springframework.org/schema/tx/spring-tx.xsd">   <!-- 定义自动注解扫描的操作 -->  <context:annotation-config/>  <context:component-scan base-package="com.kuhnwei.examples"/>  <!-- 导入所有与数据库有关的信息项-->  <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="${db.driverClassName}"/>  <property name="jdbcUrl" value="${db.url}"/>  <property name="user" value="${db.username}"/>  <property name="password" value="${db.password}"/>  </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="sessionFactory" 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:mapping/\*.xml"/>  <property name="configLocation" value="classpath:mybatis.cfg.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">  <!-- 定义该报中的访问为自动配置实现 -->  <property name="basePackage" value="com.kuhnwei.\*\*.dao"/>  <property name="sqlSessionFactoryBeanName" value="sessionFactory"/>  </bean>  <!-- 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 expression="execution(\* com.kuhnwei..service..\*.\*(..))" id="txPointcut"/>  <aop:advisor advice-ref="txAdvice" pointcut-ref="txPointcut"/>  </aop:config>  <!-- 配置事物 -->  <bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">  <property name="dataSource" ref="dataSource"/>  </bean>   <!-- 配置事物的传播特性 -->  <tx:advice id="txAdvice" transaction-manager="transactionManager">  <tx:attributes>  <tx:method name="add\*" propagation="REQUIRED"/>  <tx:method name="insert\*" propagation="REQUIRED"/>  <tx:method name="save\*" propagation="REQUIRED"/>  <tx:method name="delete\*" propagation="REQUIRED"/>  <tx:method name="update\*" propagation="REQUIRED"/>  <tx:method name="find\*" propagation="REQUIRED" read-only="true"/>  <tx:method name="get\*" propagation="REQUIRED" read-only="true"/>  <tx:method name="list\*" propagation="REQUIRED" read-only="true"/>  <tx:method name="load\*" propagation="REQUIRED" read-only="true"/>  <tx:method name="\*" propagation="REQUIRED" read-only="true"/>  </tx:attributes>  </tx:advice>  </beans> |

1. 配置SpringMVC

|  |
| --- |
| <?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/beans   http://www.springframework.org/schema/beans/spring-beans.xsd   http://www.springframework.org/schema/mvc   http://www.springframework.org/schema/mvc/spring-cache.xsd">  <!-- 定义文件的上传配置支持 -->  <bean id="multipartResolver" class="org.springframework.web.multipart.commons.CommonsMultipartResolver">  <!-- 设置每次上传文件的最大限制 -->  <property name="maxUploadSize" value="5242880"/>  <!-- 设置每次上传占用的内存大小-->  <property name="maxInMemorySize" value="4096"/>  </bean>  <!-- 配置了一个全局的异常的跳转映射，只要出现了指定的错误信息，那么就跳转到指定的页面 -->  <bean id="exceptionMapping" class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">  <property name="exceptionMappings">  <props>  <prop key="org.springframework.web.multipart.MaxUploadSizeExceededException">  errors  </prop>  </props>  </property>  </bean>  <bean id="messageSource" class="org.springframework.context.support.ResourceBundleMessageSource">  <property name="basenames">  <array>  <value>Messages</value>  <value>Pages</value>  <value>Validations</value>  </array>  </property>  </bean>  <!-- 表示要进行安全访问的路径资源配置处理 -->  <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">  <!-- 匹配前缀 -->  <property name="prefix" value="/WEB-INF/pages"/>  <!-- 匹配处理的后缀 -->  <property name="suffix" value=".jsp"/>  </bean>  <!-- 启用SpringMVC的注解配置 -->  <mvc:annotation-driven/>  <!-- 启用处理请求的servlet -->  <mvc:default-servlet-handler/> </beans> |

由于此时存在有资源访问的问题，所以所有的开发应该按照MVC的标准要求进行：所有的请求路径要先经过controler而后跳转到JSP页面上。

1. 配置相关的\*.properties文件

|  |
| --- |
| db.driverClassName=com.mysql.jdbc.Driver db.url=jdbc:mysql://localhost:3306/examples?characterEncoding=UTF-8 db.username=root db.password= db.maxPoolSize=1 db.maxIdleTime=1 db.minPoolSize=1 db.initialPoolSize=1 |

1. 在web.xml文件里面配置Spring容器以及SpringMVC的操作定义；
   1. 配置spring 与 spring mvc 的相关定义：

|  |
| --- |
| <!-- 在WEB容器里面进行Spring容器的加载 --> <listener>  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class> </listener> <!-- 此配置描述的是在项目开发过程之中，Spring容器所需要使用的配置文件 --> <context-param>  <param-name>contextConfigLocation</param-name>  <param-value>classpath:spring-context.xml</param-value> </context-param> <!-- 为此SpringMVC配置所需要的程序文件，所有的请求都提交给Spring的Servlet程序 --> <servlet>  <servlet-name>springmvc</servlet-name>  <!-- 此为SpringMVC 自己提供的servlet程序，一定要写上，因为其可以处理用户请求 -->  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  <init-param>  <param-name>contextConfigLocation</param-name>  <!-- 配置SpringMVC所需的配资文件，可以与Spring容器写在一起 -->  <param-value>classpath:spring-mvc.xml</param-value>  </init-param> </servlet> <!-- SpringMVC中所有路径的请求映射--> <servlet-mapping>  <servlet-name>springmvc</servlet-name>  <url-pattern>/</url-pattern> </servlet-mapping> <!-- Spring中提供的编码过滤器，使用的编码都是UTF-8 --> <filter>  <filter-name>encoding</filter-name>  <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>  <init-param>  <param-name>encoding</param-name>  <param-value>UTF-8</param-value>  </init-param> </filter> <filter-mapping>  <filter-name>encoding</filter-name>  <url-pattern>/\*</url-pattern> </filter-mapping> |

* 1. 修改shiro使用的过滤器：

|  |
| --- |
| <!-- 进行shiro的过滤器的配置 --> <filter>  <filter-name>ShiroFilter</filter-name>  <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>  <!-- 该参数表示shiro的声明周期将交由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> |

1. 随后还需要在spring-shiro.xml文件定义shiro的相关配置：
   1. 如果现在你的shiro是交由Spring来进行管理，则应该出现有如下的配置信息：

|  |
| --- |
| <!-- 配置SecurityManager的管理 --> <bean id="securityManager" class="org.apache.shiro.mgt.DefaultSecurityManager">  <!-- 配置你需要使用的Realms --> </bean> <!-- 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.do"/>  <!-- 认证失败之后的跳转你路径页面 -->  <property name="unauthorizedUrl" value="/unauthUrl.do"/>  <!-- 登录成功之后的跳转访问路径 -->  <property name="successUrl" value="/successUrl.do"/>  <!-- 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>  /shiroLogin.do=anon  </value>  </property> </bean> |

* 1. 如果此时shiro是交由Spring管理进行配置，那么还需要定义以下的一个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"/> |

## 完成基础登录控制

本次的操作将直接利用SpringMVC中的Action作为整体的登陆控制实现。

1. 建立User.java的DO类

|  |
| --- |
| public class User implements Serializable {  private Integer id;  private String username;  private String password;  public Integer getId() {return id; }  public void setId(Integer id) {this.id = id; }  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; } } |

1. 建立相应的映射文件，并且在里面编写好所需要的相关的DAO实现代码：
   1. 建立UserDAO.xml文件

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8" ?> <!DOCTYPE mapper  PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"  "http://mybatis.org/dtd/mybatis-3-mapper.dtd"> <mapper namespace="com.kuhnwei.examples.shiro.dao.UserDAO">   <select id="get" parameterType="Integer" resultType="User">  SELECT id,username,password FROM user WHERE id=#{id}  </select>   <select id="getByUsername" parameterType="String" resultType="User">  SELECT id,username,password FROM user WHERE username=#{username}  </select>   <select id="listAllRoleByUserId" parameterType="Integer" resultType="String">  SELECT flag FROM role WHERE id IN (  SELECT role\_id FROM user\_role WHERE user\_id=#{userId}  )  </select>   <select id="listAllPermissionByUserId" parameterType="Integer" resultType="String">  SELECT flag FROM permission WHERE id IN(  SELECT permission\_id FROM role\_permission WHERE role\_id IN(  SELECT role\_id FROM user\_role WHERE user\_id=#{userId}  )  )  </select> </mapper> |

1. 建立好UserDAO接口，这个接口建立的时候需要注意包名称，以及方法名称要与配置文件中定义的操作的ID一样。

|  |
| --- |
| @Repository public interface UserDAO {  User get(Integer id);  User getByUsername(String username);  Set<String> listAllRoleByUserId(Integer userId);  Set<String> listAllPermissionByUserId(Integer userId); } |

1. 定义UserService业务层处理，这里徐亚哦提供有个两个操作方法：认证、授权；

|  |
| --- |
| @Service public class UserService {  @Autowired  private UserDAO userDAO;  public User get(Integer id) {  return this.userDAO.get(id);  }  public User getByUsername(String username) {  return this.userDAO.getByUsername(username);  }  public Map<String, Object> listAuthByUserId(Integer userId) {  Map<String, Object> map = new HashMap<String, Object>(16);  map.put("allRoles", this.userDAO.listAllRoleByUserId(userId));  map.put("allPermissions", this.userDAO.listAllPermissionByUserId(userId));  return map;  } } |

1. 建立一个新的UserRealm程序类

|  |
| --- |
| @Component public class UserRealm extends AuthorizingRealm {  @Autowired  private UserService userService;  @Override  protected AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principal) {  String username = (String) principal.getPrimaryPrincipal();  SimpleAuthorizationInfo auth = new SimpleAuthorizationInfo();  try {  User user = this.userService.getByUsername(username);  Map<String, Object> map = this.userService.listAuthByUserId(user.getId());  Set<String> allRoles = (Set<String>) map.get("allRoles");  Set<String> allPermissions = (Set<String>) map.get("allPermissions");  auth.setRoles(allRoles);  auth.setStringPermissions(allPermissions);  } catch (Exception e) {  e.printStackTrace();  }  return auth;  }  @Override  protected AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken token) throws AuthenticationException {  String username = (String) token.getPrincipal();  User user = null;  try {  user = this.userService.getByUsername(username);  } catch (Exception e) {  e.printStackTrace();  }  if (user == null) {  throw new UnknownAccountException("该用户名称不存在！");  } else {  String password = new String((char[]) token.getCredentials());  if (user.getPassword().equals(password)) {  AuthenticationInfo auth = new SimpleAuthenticationInfo(username, password, "userRealm");  return auth;  } else {  throw new IncorrectCredentialsException("密码错误");  }  }  } } |

1. 修改spring-shiro.xml文件，在这个文件里面要对使用的程序的Realm进行配置；

|  |
| --- |
| <!-- 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="userRealm"/> </bean> |

* 1. 过滤器的路径配置完整：

|  |
| --- |
| <property name="filterChainDefinitions">  <value>  /\*=anon  /shiroLogin.do=anon  /messages/\*\*=authc  /admin\*=authc  /pages/welcome.jsp=authc,perms[admin:add],perms[dept:add]  </value> </property> |

1. 建立一个登录的Controller，这个Controller要处理各个页面的问题；

|  |
| --- |
| @Controller public class UserLoginController {  @RequestMapping("/shiroLogin.do")  public ModelAndView login(String username, String password) {  ModelAndView mav = new ModelAndView();  Subject subject = SecurityUtils.*getSubject*();  UsernamePasswordToken token = new UsernamePasswordToken(username, password);  try {  subject.login(token);  mav.setViewName("welcome");  } catch (Exception e) {  e.printStackTrace();  mav.setViewName("login");  }  return mav;  }  @RequestMapping("loginUrl.do")  private ModelAndView loginUrl() {  return new ModelAndView("login");  }  @RequestMapping("unauthUrl.do")  private ModelAndView unauthUrl() {  return new ModelAndView("unauthUrl");  }  @RequestMapping("successUrl.do")  private ModelAndView successUrl() {  return new ModelAndView("welcome");  } } |

1. 将所有的页面都移动到WEB-INF目录下；
2. 修改login.jsp页面的提交路径；

|  |
| --- |
| <form action="shiroLogin.do" method="post"> |

整个的登录认证的处理全部交由shiro进行完成，而用户只是负责提供所需要的角色、权限、用户认证的相关信息。

## 使用内置登录处理

不管现在使用的是SpringMVC或者是普通的Servlet可以发现针对于shiro都是按照固定的套路进行的。这样的话对于整个程序而言是非常重复的，于是shiro决定了，将登录处理自己也包揽了将通过配置实现shiro的登录操作。

1. 修改spring-shiro.xml文件，增加内置登录处理；
   1. 检测器：org.apache.shiro.web.filter.authc.FormAuthenticationFilter;

|  |
| --- |
| <!-- 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="username"/>  <property name="passwordParam" value="password"/>  <property name="loginUrl" value="/loginUrl.do"/> </bean> |

1. 修改shiro过滤器配置；

|  |
| --- |
| <!-- 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.do"/>  <!-- 认证失败之后的跳转你路径页面 -->  <property name="unauthorizedUrl" value="/unauthUrl.do"/>  <!-- 登录成功之后的跳转访问路径 -->  <property name="successUrl" value="/successUrl.do"/>  <!-- 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.do=authc  /\*=anon  /messages/\*\*=authc  /admin\*=authc  /pages/welcome.jsp=authc,perms[admin:add],perms[dept:add]  </value>  </property> </bean> |

1. 修改login.jsp页面的表单提交路径

|  |
| --- |
| <form action="loginUrl.do" method="post"></form> |

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

## 控制层注解实现权限检测

为了可以更好的说明问题，下面模拟一个新的Controller，提供CRUD伪操作方法；

1. 建立一个NewsController.java

|  |
| --- |
| @Controller @RequestMapping("/news") public class NewsController {  private Logger logger = Logger.*getLogger*(NewsController.class);  @RequestMapping("/save.do")  public String save() {  logger.info("---------------[NewsController#save.do]-----------------");  return null;  }  @RequestMapping("/update.do")  public String update() {  logger.info("---------------[NewsController#update.do]-----------------");  return null;  }  @RequestMapping("/remove.do")  public String remove() {  logger.info("---------------[NewsController#remove.do]-----------------");  return null;  }  @RequestMapping("/list.do")  public String list() {  logger.info("---------------[NewsController#list.do]-----------------");  return null;  } } |

1. 此时只需要进行认证，不需要进行授权的检测，所有的授权检测将通过Controller自己完成控制，所以修改spring-shiro.xml文件追加如下的检测路径:
   1. 此时的配置表示此路径下的所有内容都必须进行检测，如果要进行某一个方法的检测，那么就可以通过注解配置完成，就没有必要去编写如下操作。

|  |
| --- |
| /news/\*\*=authc |

1. 如果要想进行操作的启用，还需要修改spring-mvc.xml文件
   1. 需要启用AOP的代理设计程序：

|  |
| --- |
| <!-- 启动再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. 针对于安全管理器进行的AOP配置

|  |
| --- |
| <!-- 针对于安全管理器进行的AOP配置 --> <bean id="controllerAutorizationAttributeSourceAdvisor"  class="org.apache.shiro.spring.security.interceptor.AuthorizationAttributeSourceAdvisor">  <property name="securityManager" ref="securityManager"/> </bean> |

1. 在shiro里面针对于控制层注解有如下几种：
   1. 认证检测：@RequiresAuthentication，该控制方法必须登录后才可以使用：

范例：修改Controller中的程序

* 1. 指的是当前的save()这个方法一定要经过认证之后才可以执行。

|  |
| --- |
| @RequiresAuthentication @RequestMapping("/save.do") public String save() {  logger.info("---------------[NewsController#save.do]-----------------");  return null; } |
| 如果此时用户没有进行登录，则出现“org.apache.shiro.authz.UnauthenticatedException”异常信息。 |

如果此时不希望页面出现500的错误信息，那么需要修改web.xml文件增加错误页面你的配置。

|  |
| --- |
| <error-page>  <exception-type>  org.apache.shiro.authz.UnauthenticatedException  </exception-type>  <location>/loginUrl.do</location> </error-page> |

1. 游客检测：@RequiresGuest
2. 授权检测：
   1. 角色：@RequiresRoles(“amin”)
   2. 权限：@RequiresPermissions(“admin:update”)

此时如果没有对应的权限或者角色，则会出现“org.apache.shiro.authz.UnauthorizedException”异常，所以还需要再web.xml文件里配置一个错误页，表示授权的操作错误：

|  |
| --- |
| <error-page>  <exception-type>  org.apache.shiro.authz.UnauthorizedException  </exception-type>  <location>/unauthUrl.do</location> </error-page> |

1. 其它的检测：
   1. 记住我的用户检测：@RequiresUser

## 业务层注解实现权限认证

控制层和业务层实现的Annotation的注解权限认证从使用上来讲是没有任何区别的，依然使用这几个注解配置。

## 总结

Shiro中的权限认证机制可以针对于路径检测、角色检测、权限检测多种方式完成。

# 缓存配置

在整个实现的自定义的Realm之中，如果要进行授权的检测，那么会发现每一次的处理之中都会进行数据库的拆线呢，很明显，这样的做法明显不利于程序性能。所以为了解决这样的问题，就建议使用缓存处理，也就是说将用户的角色以及权限信息保存在缓存之中。

1. 需要为项目增加缓存组件：

|  |
| --- |
| <dependency>  <groupId>org.apache.shiro</groupId>  <artifactId>shiro-ehcache</artifactId>  <version>1.3.2</version> </dependency> |

可以发现shiro里面所使用的缓存组件为ehcache组件，而这个组件在hibernate上也被广泛应用。

1. 需要定义一个缓存的配置文件，这个配置文件直接拷贝已有的即可：

将ehcache.xml文件拷贝到整个项目的资源目录下。、

|  |
| --- |
| <?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"/>  <cache name="passwordRetryCache"  maxElementsInMemory="2000"  eternal="false"  timeToIdleSeconds="300"  timeToLiveSeconds="0"  overflowToDisk="false">  </cache>  <cache name="authenticationCache"  maxElementsInMemory="2000"  eternal="false"  timeToIdleSeconds="1800"  timeToLiveSeconds="0"  overflowToDisk="false">  </cache>  <cache name="shiro-activeSessionCache"  maxElementsInMemory="2000"  eternal="false"  timeToIdleSeconds="1800"  timeToLiveSeconds="0"  overflowToDisk="false">  </cache> </ehcache> |

在此配置文件之中实际上有以下几个核心选项：

* “<diskStore path="java.io.tmpdir"/> “：磁盘的存储目录；
* “name="xxx"”：对要进行缓存的项进行一个标注；
* “maxElementsInMemory="2000"“：可以缓存的最大的对象个数；
* “eternal="false"”：是否允许自动失效（如果某一个对象长时间不使用）；
* “timeToIdleSeconds="1800" “：最小的失效时间，单位秒；
* “timeToLiveSeconds="0" “：最大的保存时间，单位秒；
* “overflowToDisk="false"”：如果容量过多，可以将其保存在磁盘上。

1. 如果要想使缓存生效，则还需要修改spring-shiro.xml文件进行缓存配置：
   1. 定义缓存管理器：

|  |
| --- |
| <!-- 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> |

* 1. 在安全管理器之中注册此缓存管理器：

|  |
| --- |
| <!-- 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="userRealm"/>  <property name="cacheManager" ref="cacheManager"/> </bean> |

以后如果进行角色或权限检测的时候不会重复进行数据库的频繁查询了。