

权限管理理论基础

什么是权限管理

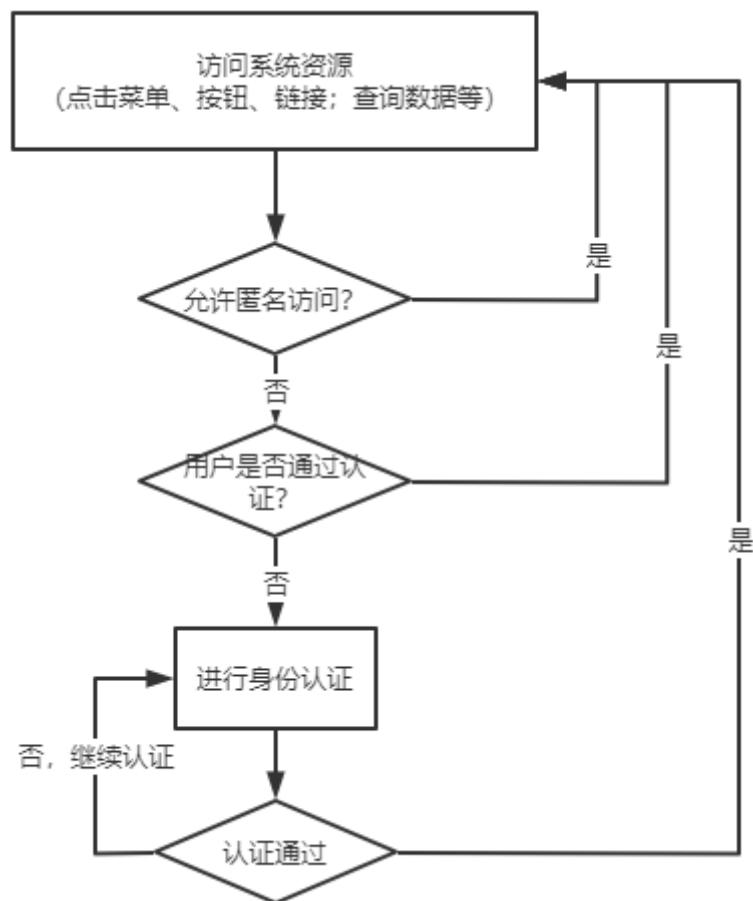
只要涉及用户参与的系统一般都要有权限管理，权限管理实现对用户访问系统的控制，按照安全规则或者安全策略控制用户可以访问且只能访问被授权的资源。

主要包含了 **用户认证（验证）** 和 **用户授权** 两部分。

用户认证

用户访问系统时，系统要验证用户身份的合法性。

验证的流程：



这个流程中涉及到一些关键的对象：

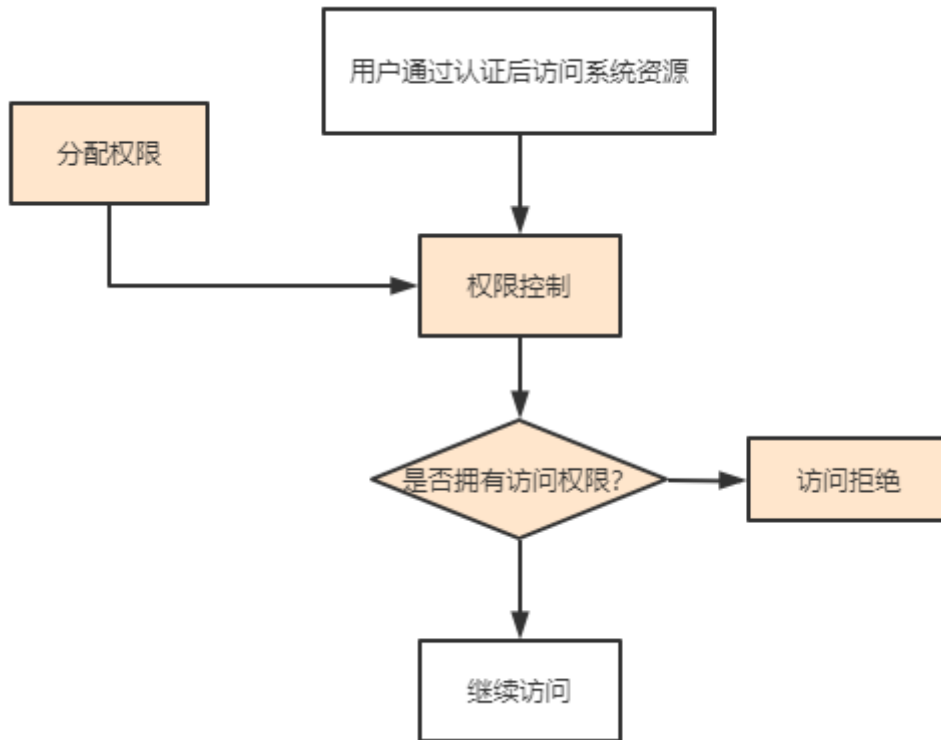
- 1、谁要访问系统资源？可能是用户，也可能是某个程序==>就是访问资源的角色，一般称为 "**主体（Subject）**" 系统需要对这个subject进行身份认证
- 2、验证的是什么？验证的是 **身份信息（Principal）**，一个主体可以有多个身份信息（但一般会有一个主身份信息-primary principal）
- 3、主体用什么进行验证？主体提供的 **凭证信息（Credential）**，可以是密码、证书、指纹、人脸数据等。

也就是说，主体在进行**身份认证**时，需要提供**身份信息**和**凭证信息**。

用户授权

用户授权也就是访问控制，当用户通过认证后，系统对用户访问的资源进行控制，具有对应资源的访问权限才能够访问该资源。

授权的流程：

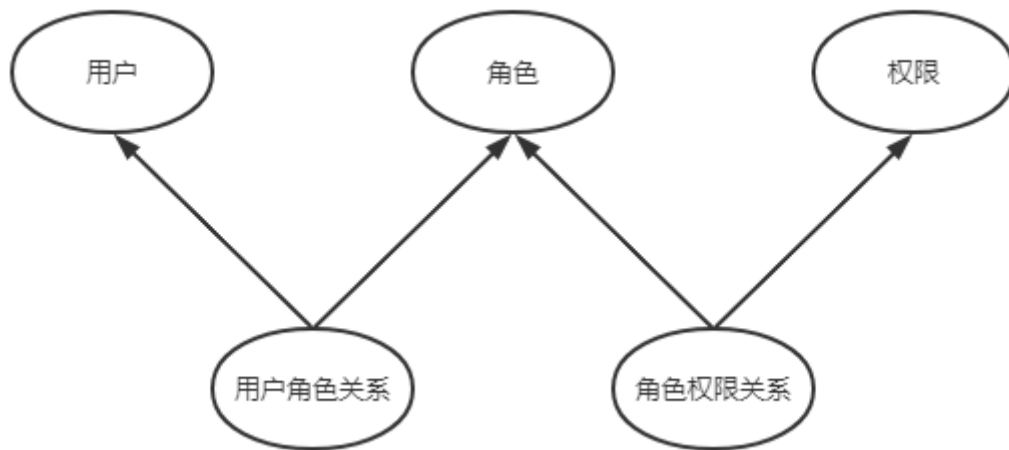


这个流程中涉及到的关键对象有：

- 1、谁被授权？还是 **主体Subject**
- 2、权限控制了什么？ **资源（Resource）**。Subject需要具备对资源的访问权限才能访问资源，这里的资源可以分为资源的类型和资源的实例，例如系统的用户信息就是资源类型，相当于类，用户可以对这一类的资源具有访问权限；系统中id为1的用户信息就是资源实例，相当于new出来的对象，某个个人用户可能就只有访问他自己的资源实例的权限；
- 3、主体授权后对资源做什么？ **权限/许可（Permission）**，就是针对资源所拥有的权限或许可，描述了如何取访问、操作资源，例如：添加用户、修改用户、删除商品信息、等。

权限控制

权限管理的核心就是用户、角色、权限以及他们之间的关系。针对他们的关系，通常在企业开发中会绘制出这样的模型关系图：



- 用户：账号、密码
- 角色：角色名称
- 权限：权限名称、资源id，通常企业开发中将资源和权限合并（权限名称、资源名称、资源访问地址）
- 用户角色关系：用户id、角色id
- 角色权限关系：角色id、权限id

这就是 **RBAC模型**（Role Based Access Control，**基于角色的访问控制模型**），用户通过角色与权限进行了关联。

进行权限控制，要先把权限模型建立起来（数据库、表），之后将这些信息存储到数据库之中，然后在系统中通过权限分配模块实现权限控制，对角色、权限进行增删改查等操作。

关于RBAC模型

角色是针对人划分的，人作为用户在系统中属于活动内容，如果该角色可以访问的资源出现变更，需要修改代码。例如，用户报表的资源本来是部门经理可以查看：

```
if(user.hasRole('部门经理')){.....}
```

现在变更需求后总经理也能查看，那我们判断用户是否可以查看用户报表的操作代码就需要修改为：

```
if(user.hasRole('部门经理') || user.hasRole('总经理')){.....}
```

可以看到，如果角色所对应的权限发生变化，我们所编写的判断逻辑就必须发生改变（因为用户要额外判断更多的角色了），基于角色的访问控制不利于系统维护，可扩展性不强。

但是实际上由于将资源和权限已经合并，这种情况下，并不局限于只针对角色分配权限（即只判断用户是否具有某个角色），完全可以将资源权限直接分配给用户（即直接判断用户是否具有某个权限）

```
if (user.hasPermission('用户报表查看')) {.....}
```

因此就有了新的解释：**RBAC模型**（Resource Based Access Control，**基于资源的访问控制模型**）

权限管理解决方案

管理维度

对于权限管理可以实现粗粒度权限管理和细粒度权限管理。

- 粗粒度权限管理

- 是对资源类型的权限管理，资源类型例如菜单、URL链接、添加用户页面、用户信息、类方法、页面中的按钮
- 权限管理的例子：超级管理员可以访问用户添加页面、用户信息页面等全部页面，读者可以访问读者首页、个人信息页中所有按钮。
- 细粒度权限管理
 - 是对资源实例的权限管理，就是数据级别的权限管理，是资源类型的具体的实例，例如管理员可以访问对id为1的图书信息的修改链接、id为1的图书的所有借阅记录，读者能看到自己的借阅记录。

解决方案

- 对于粗粒度权限管理，可以将权限管理的代码抽取出来，子啊系统架构级别统一处理，例如通过springmvc框架的拦截器实现授权
- 对于细粒度权限管理，由于在数据级别没有共性，不能抽取代码，也是系统业务逻辑的一部分，在业务层处理相对比较简单，因此建议在业务层控制，例如某个员工管理系统中，部门经理想要查看本部门的所有员工信息，可以在service层提供一个部门id作为参数的方法，控制器中根据当前用户的信息得到当前部门经理属于哪个部门，然后调用service接口时传递部门id，实现该用户能够查询指定部门id的员工。

实际开发中比较常用的一种方式是对url地址进行拦截，可以用Servlet技术中的过滤器Filter进行过滤，也可以使用SpringMVC框架中的拦截器进行拦截。

然而自己开发，需要编写大量的 **if-else判断、各种过滤器拦截器代码**，涉及到开发、维护、版本迭代、更新等，权限也可能发生变化，必须依靠一个更优秀的安全权限框架来完成。**Apache Shiro** 是一个Java平台的现代权限管理框架，通过它的权限(Permission)概念，Shiro很好地支持基于资源的权限访问控制。

Shiro概念

基本概念

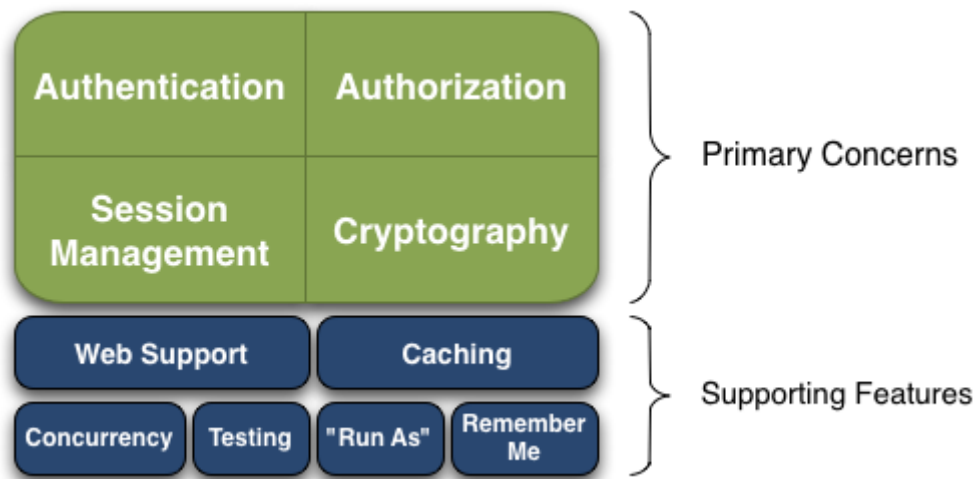
官网地址：<http://shiro.apache.org/>

Apache Shiro是一个Java的安全（权限）框架，可以轻松地完成：

- **身份认证**：说白了就是登录，涉及到身份权限识别、密码管理等
- **授权**：根据身份找到对应的权限，权限不同，访问某些功能时就会不一样，可能继续访问，可能就提示权限不足
- **加密**：保护数据的安全性，例如将密码加密后存储
- **会话管理**：例如在web应用中我们登录后把登录用户存储在Session中，做身份识别也就需要把身份信息、登录状态存储到会话中，所以做安全、访问控制的框架自然需要能够管理会话。基本不需要我们做什么控制，shiro框架在后台完成一些定制，开发者还是当作普通的Session来使用。

特性

官网（<http://shiro.apache.org/introduction.html>）中介绍：Shiro是一个具有许多特性的综合应用程序安全框架。它的主要精力集中在：



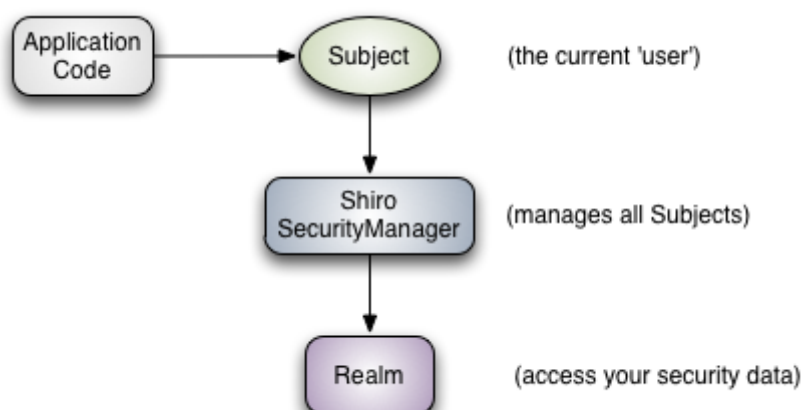
- Authentication: 身份认证 / 登录, 验证用户是不是拥有相应的身份;
- Authorization: 授权, 即权限验证, 验证某个已认证的用户是否拥有某个权限; 即判断用户是否能做事情, 常见的如: 验证某个用户是否拥有某个角色。或者细粒度的验证某个用户对某个资源是否具有某个权限;
- Session Manager: 会话管理, 即用户登录后就是一次会话, 在没有退出之前, 它的所有信息都在会话中; 会话可以是普通 JavaSE 环境的, 也可以是如 Web 环境的;
- Cryptography: 加密, 保护数据的安全性, 如密码加密存储到数据库, 而不是明文存储;
- Web Support: Web 支持, 可以非常容易的集成到 Web 环境;
- Caching: 缓存, 比如用户登录后, 其用户信息、拥有的角色 / 权限不必每次去查, 这样可以提高效率;
- Concurrency: shiro 支持多线程应用的并发验证, 即如在一个线程中开启另一个线程, 能把权限自动传播过去;
- Testing: 提供测试支持;
- Run As: 允许一个用户假装为另一个用户 (如果他们允许) 的身份进行访问;
- Remember Me: 记住我, 这个是非常常见的功能, 即一次登录后, 下次再来的话不用登录了。

记住一点, Shiro 不会去维护用户、维护权限; 这些需要我们自己去设计 / 提供; 然后通过相应的接口注入给 Shiro 即可。

体系结构

概述

在官网 (<http://shiro.apache.org/architecture.html#apache-shiro-architecture>) 介绍, 从高层面上看, Shiro 的体系结构有 3 个主要的概念 (或者叫它核心组件): **Subject**、**SecurityManager** 和 **Realms**。这三个核心组件之间的交互可以用下图表示, 也体现了它的工作流程:



- Subject
 - 主体，代表了当前“用户”，这个用户不一定是一个具体的人，与当前应用交互的任何东西都是 Subject，如网络爬虫，机器人等；即一个抽象概念；所有 Subject 都绑定到 SecurityManager，与 Subject 的所有交互都会委托给 SecurityManager；可以把 Subject 认为是一个门面；SecurityManager 才是实际的执行者；
 - **应用代码直接交互的对象是 Subject，只要得到 Subject 对象就可以进行大多数的 Shiro 操作了。Shiro 对外 API 的核心就是 Subject。**
- 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他组件进行交互，如果学习过 SpringMVC，你可以把它看成 DispatcherServlet 前端控制器；
- Realms
 - 域，Shiro 从 Realm 获取安全数据（如用户、角色、权限），就是说 SecurityManager 要验证用户身份，那么它需要从 Realm 获取相应的用户进行比较以确定用户身份是否合法；也需要从 Realm 得到用户相应的角色 / 权限进行验证用户是否能进行操作；可以把 Realm 看成 DataSource，即安全数据源。

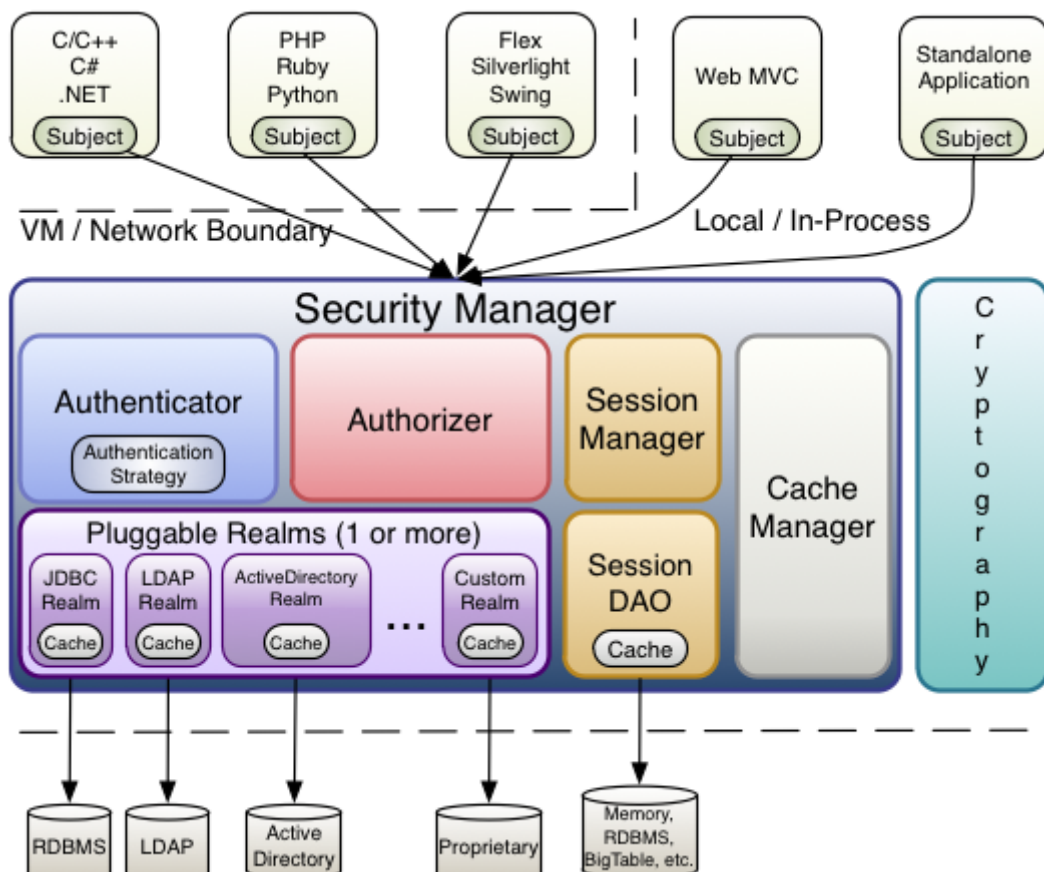
总之，对于我们而言，一个 Shiro 应用应当是这样的：

应用代码通过 **Subject** 来进行认证和授权，而 **Subject** 又委托给 **SecurityManager**；我们需要给 **Shiro** 的 **SecurityManager** 注入 **Realm**，从而让 **SecurityManager** 能得到合法的用户及其权限进行判断。

从以上也可以看出，Shiro 不提供维护用户 / 权限，而是通过 Realm 让开发人员自己注入。

详细结构

根据官网（<http://shiro.apache.org/architecture.html#detailed-architecture>）的描述，Shiro 的核心架构的详细组成：



- Subject: 主体, 可以看到主体可以是任何可以与应用交互的“用户”;
- SecurityManager: 相当于 SpringMVC 中的 DispatcherServlet 或者 Struts2 中的 FilterDispatcher; 是 Shiro 的心脏; 所有具体的交互都通过 SecurityManager 进行控制; 它管理着所有 Subject、且负责进行认证和授权、及会话、缓存的管理。
- Authenticator: 认证器, 负责主体认证的, 这是一个扩展点, 如果用户觉得 Shiro 默认的不好, 可以自定义实现; 其需要认证策略 (Authentication Strategy), 即什么情况下算用户认证通过了;
- Authorizer: 授权器, 或者访问控制器, 用来决定主体是否有权限进行相应的操作; 即控制着用户能访问应用中的哪些功能;
- Realm: 可以有 1 个或多个 Realm, 可以认为是安全实体数据源, 即用于获取安全实体的; 可以是 JDBC 实现, 也可以是 LDAP 实现, 或者内存实现等等; 由用户提供; 注意: Shiro 不知道你的用户 / 权限存储在哪及以何种格式存储; 所以我们一般在应用中都需要实现自己的 Realm;
- SessionManager: 如果写过 Servlet 就应该知道 Session 的概念, Session 呢需要有人去管理它的生命周期, 这个组件就是 SessionManager; 而 Shiro 并不仅仅可以用在 Web 环境, 也可以用在如普通的 JavaSE 环境、EJB 等环境; 所有呢, Shiro 就抽象了一个自己的 Session 来管理主体与应用之间交互的数据; 这样的话, 比如我们在 Web 环境用, 刚开始是一台 Web 服务器; 接着又上了台 EJB 服务器; 这时想把两台服务器的会话数据放到一个地方, 这个时候就可以实现自己的分布式会话 (如把数据放到 Memcached 服务器);
- SessionDAO: DAO 大家都用过, 数据访问对象, 用于会话的 CRUD, 比如我们想把 Session 保存到数据库, 那么可以实现自己的 SessionDAO, 通过如 JDBC 写到数据库; 比如想把 Session 放到 Memcached 中, 可以实现自己的 Memcached SessionDAO; 另外 SessionDAO 中可以使用 Cache 进行缓存, 以提高性能;
- CacheManager: 缓存控制器, 来管理如用户、角色、权限等的缓存的; 因为这些数据基本上很少去改变, 放到缓存中后可以提高访问的性能
- Cryptography: 密码模块, Shiro 提高了一些常见的加密组件用于如密码加密 / 解密的。

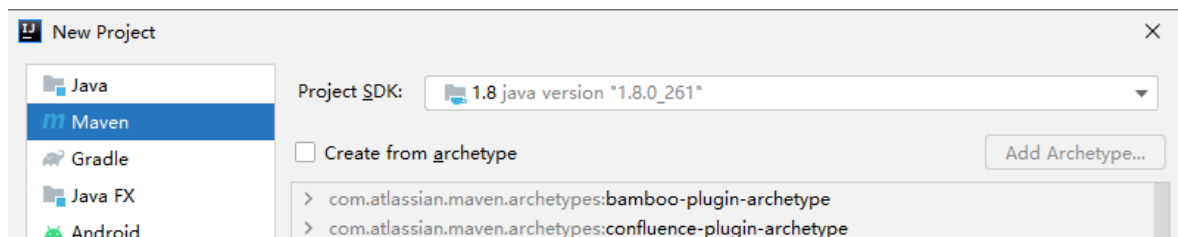
到此 Shiro 架构及其组件就认识完了, 接下来可以学习 Shiro 的组件了。

入门案例开发

接下来, 通过一个入门案例来初步了解一下 Shiro (官网入门地址: <http://shiro.apache.org/tutorial.html>)。

1. 创建项目

创建一个简单的 Maven 工程, 直接创建即可, 不需要从项目原型创建。



生成的 Pom 文件中会添加多余的属性信息, 并不需要, 可以删除。

2. 添加依赖

添加 Shiro 核心依赖, 另根据官网, Shiro 采用 SLF4J 进行日志输出 (SLF4J 是门面, 需要指定其实现), 因此添加相关依赖。

```

<!-- https://mvnrepository.com/artifact/org.apache.shiro/shiro-core -->
<dependency>
    <groupId>org.apache.shiro</groupId>
    <artifactId>shiro-core</artifactId>
    <version>1.7.1</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.slf4j/slf4j-simple -->
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-simple</artifactId>
    <version>1.7.30</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.slf4j/jcl-over-slf4j -->
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>jcl-over-slf4j</artifactId>
    <version>1.7.30</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.slf4j/slf4j-log4j12 -->
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-log4j12</artifactId>
    <version>1.7.30</version>
</dependency>

```

SLF日志输出需要搭配相应的log4j.properties配置文件

```

#root category priority: fatal error warn info debug trace off
#log4j.rootLogger=level,appendername1,appendername2...
#log4j.rootLogger=debug, CONSOLE, LOGFILE
log4j.rootLogger=debug, CONSOLE

# CONSOLE is set to be a ConsoleAppender using a PatternLayout.
log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppender
log4j.appender.CONSOLE.layout=org.apache.log4j.PatternLayout
log4j.appender.CONSOLE.layout.ConversionPattern=%d{ISO8601} %-6r [%15.15t] %-5p
%30.30c %x - %m\n
log4j.appender.CONSOLE.encoding=UTF-8

# LOGFILE is set to be a File appender using a PatternLayout.
#log4j.appender.LOGFILE=org.apache.log4j.FileAppender
#log4j.appender.LOGFILE.File=d:/mybatis.log
#log4j.appender.LOGFILE.Append=true
#log4j.appender.LOGFILE.layout=org.apache.log4j.PatternLayout
#log4j.appender.LOGFILE.layout.ConversionPattern=%d{ISO8601} %-6r [%15.15t] %-5p
%30.30c %x - %m\n

```

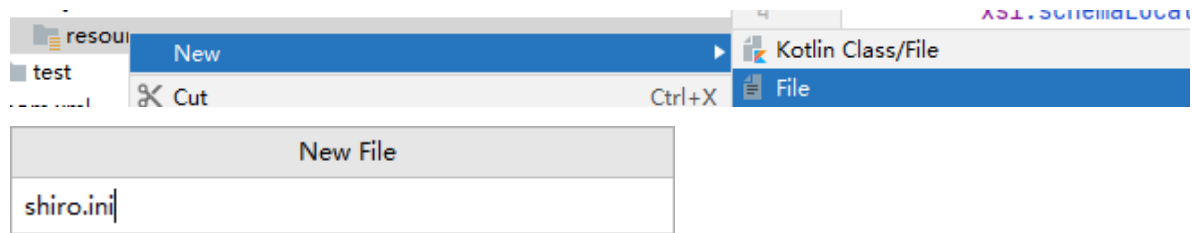
3.创建SecurityManager

在应用程序中启用Shiro框架首先要理解的是基本所有的东西都和核心组件 **SecurityManager** 有关。所以要做的第一件事情就是创建 **SecurityManager**实例。

Shiro中 **SecurityManager** 可以通过很多格式来配置实现，例如 **INI**、**XML**、**YAML**、**JSON** 等。

INI格式比较简单并且可以应用于其他各种开发环境下，这里我们就通过INI文件来学习和了解Shiro的规则。

1) 在资源文件夹中创建shiro.ini（IDEA中添加ini插件）



2) 编辑用户、角色、权限信息

```
# 定义用户信息
# 格式：用户名=密码,角色1,角色2,...
[users]
zhangsan=123,admin
lisi=456,manager,seller
wangwu=789,clerk

# 定义角色信息
# 格式：角色=权限1,权限2,...
# 预设系统中有很多的权限：
# user:query user:detail:query user:update
# user:insert order:update ...
[roles]
# admin拥有所有权限，用*表示
admin=*
# manager拥有user的所有权限
manager=user:*
# clerk只有查询权限
clerk=user:query,user:detail:query
```

权限中的冒号只是一种风格，而不是必须的格式

3) 创建核心对象SecurityManager

```
// 创建SecurityFactory，加载ini配置，并通过它创建SecurityManager
Factory<SecurityManager> factory = new
IniSecurityManagerFactory("classpath:shiro.ini");
// 获取SecurityManager实例
SecurityManager securityManager = factory.getInstance();
// 托管安全管理器到容器中
SecurityUtils.setSecurityManager(securityManager);
// 获取当前主体Subject
Subject subject = SecurityUtils.getSubject();
```

小结

Shiro中的操作基本都通过Subject来进行，所以：创建核心对象SecurityManager、将核心托管到容器中、通过容器拿到Subject这三步是前提需要完成的。

4.Shiro的常用操作

1) 获取当前的主体的登录状态

```
// 通过subject获取当前用户的登录状态（该状态从session中同步信息）
System.out.println(subject.isAuthenticated());
```

2) 进行身份认证（即登录）

```
AuthenticationToken token = new UsernamePasswordToken("zhangsan", "123");
subject.login(token);
```

3) 获取登录凭证（用户名）

```
// principal-->登录凭证，就是登录名
System.out.println("principal:" + subject.getPrincipal());
```

4) 角色校验

```
// 角色校验
System.out.println(subject.hasRole("admin"));
System.out.println(subject.hasRole("manager"));
// 其他角色校验方法: subject.hasAllRoles
```

5) 权限校验

```
// 权限校验
System.out.println(subject.isPermitted("a:b"));
System.out.println(subject.isPermitted("user:insert"));
// 其他权限校验方法: subject.isPermittedAll
```

6) 登出

```
// 登出：身份信息、登录状态信息、角色信息、会话信息会被全部清除
subject.logout();
System.out.println("principal:" + subject.getPrincipal());
```

5.源码解读*

1、通过代码

```
Factory<SecurityManager> factory = new
IniSecurityManagerFactory("classpath:shiro.ini");
```

创建了一个工厂对象。

在这个 `IniSecurityManagerFactory` 类中就加载了ini文件：

```
public IniSecurityManagerFactory(String iniResourcePath) {
    this(Ini.fromResourcePath(iniResourcePath));
}
```



2、当我们调用工厂的获取实例方法来获取SecurityManager时：

```
SecurityManager securityManager = factory.getInstance();
```

点击 `getInstance` 后按 `CTRL` + `T`，查看实现：

```
= factory.getInstance();
```

Choose Implementation of `Factory.getInstance()` (2 found)

 <code>AbstractFactory</code> (<code>org.apache.shiro.util</code>)	Maven: <code>org.apache.shiro:shiro-core:1.7.1</code> (<code>shiro-core-1.7.1.jar</code>)
 <code>JndiObjectFactory</code> (<code>org.apache.shiro.jndi</code>)	Maven: <code>org.apache.shiro:shiro-core:1.7.1</code> (<code>shiro-core-1.7.1.jar</code>)

之后跳转到 `AbstractFactory`：

```
public T getInstance() {
    Object instance;
    if (this.isSingleton()) {
        if (this.singletonInstance == null) {
            this.singletonInstance = this.createInstance();
        }

        instance = this.singletonInstance;
    } else {
        instance = this.createInstance();
    }
}
```

获取这个安全管理器的实例时，首次将会调用 `createInstance()` 方法。当前类的该方法是一个抽象方法，因此继续查找其实现，因为之前加载了ini文件，此时根据ini文件来创建实例，按 `CTRL` + `T` 后直接跳转到了 `IniFactorySupport` 类中。

```
public T createInstance() {
    Ini ini = this.resolveIni();
    Object instance;
    String msg;
    if (CollectionUtils.isEmpty(ini)) {
        log.debug("No populated Ini available. Creating a default instance");
        instance = this.createDefaultInstance();
        if (instance == null) {
            msg = this.getClass().getName() + " implementation did not return an instance";
            throw new IllegalStateException(msg);
        }
    } else {
        log.debug("Creating instance from Ini [" + ini + "]");
        instance = this.createInstance(ini);
        if (instance == null) {
            msg = this.getClass().getName() + " implementation did not return an instance";
            throw new IllegalStateException(msg);
        }
    }

    return instance;
}
```

该类中的方法仍然是一个抽象方法，因此继续跟踪后发现这个方法实际上调用的是

`IniSecurityManagerFactory` 类中的方法：

```
protected SecurityManager createInstance(Ini ini) {
    if (CollectionUtils.isEmpty(ini)) {
        throw new NullPointerException("Ini argument cannot be null or empty.");
    } else {
        SecurityManager securityManager = this.createSecurityManager(ini);
        if (securityManager == null) {
            String msg = SecurityManager.class + " instance cannot be null.";
            throw new ConfigurationException(msg);
        } else {
            return securityManager;
        }
    }
}
```

发现最终还是绕回了我们最初的类里。

3、继续深入这个创建实例的方法

```
private SecurityManager createSecurityManager(Ini ini) {
    return this.createSecurityManager(ini, this.getConfigSection(ini));
}
```

```
private SecurityManager createSecurityManager(Ini ini, Section mainSection) {
    this.getReflectionBuilder().setObjects(this.createDefaults(ini, mainSection));
    Map<String, ?> objects = this.buildInstances(mainSection);
    SecurityManager securityManager = this.getSecurityManagerBean();
    boolean autoApplyRealms = this.isAutoApplyRealms(securityManager);
    if (autoApplyRealms) {
```

```
protected Map<String, ?> createDefaults(Ini ini, Section mainSection) {
    Map<String, Object> defaults = new LinkedHashMap();
    SecurityManager securityManager = this.createDefaultInstance();
    defaults.put("securityManager", securityManager);
    if (this.shouldImplicitlyCreateRealm(ini)) {
        Realm realm = this.createRealm(ini);
        if (realm != null) {
            defaults.put("iniRealm", realm);
        }
    }
}
```

找到一句关键代码，根据ini来创建这个默认的Realm了。继续进去：

```
protected Realm createRealm(Ini ini) {
    IniRealm realm = new IniRealm();
    realm.setName("iniRealm");
    realm.setIni(ini);
    return realm;
}
```

可以看到这里使用默认构造方法创建了IniRealm，那这个默认的IniRealm里有什么呢？

```
public class IniRealm extends TextConfigurationRealm {  
    public static final String USERS_SECTION_NAME = "users";  
    public static final String ROLES_SECTION_NAME = "roles";  
    private static final transient Logger log = LoggerFactory.getLogger(IniRealm.class);  
    private String resourcePath;  
    private Ini ini;
```

可以看到，这里就定义了INI文件对于Shiro的Realm的规范要求，这也是为什么我们在ini里要定义这么两个区段的原因。

6.权限规则*

最常用的权限标识：【资源：操作】

- `user:query` , `user:insert` , `order:delete` , `menu:show`
 - 这里冒号:作为分隔符，分隔资源和操作，资源:操作
 - 逗号,作为分隔符，分隔多个权限，权限1,权限2
- `user:*` , `*:query` ==》 * 作为通配符，代表所有的操作或资源， `user:*` 即user的所有操作， `*:query` 即所有资源的查询操作
- `*` ==》代表一切资源的一切权限，*是最高权限
- 其他细节
 - `user:*` 可以匹配 `user:xx` , `user:xx:xxx`
 - `*:query` 只可以匹配 `xx:query` , 不能匹配 `xx:xx:query` , 除非 `*:*:query`
 - `user:update,user:insert` 可以简写为 `"user:update,insert"` , 注意要加引号，例如
 - [roles]
manager1=user:query, user:update, user:insert
manager2="user:query,update,insert"

实例级权限标识：【资源：操作：实例】

粒度细化到具体某个资源实例。

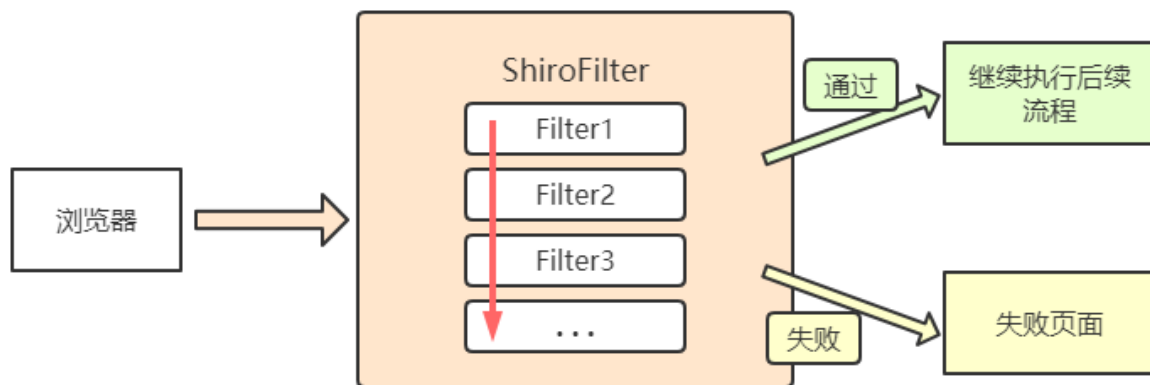
- `user:update:1,user:delete:1` ==》对用户1可以update和delete
- `"user:update,delete:1"` ==》和上面等价
- `user:*:1` , `user:update:*` , `user:*:*`

WEB集成

Shiro在Web项目中的工作模式

Shiro对Web应用提供了很好的支持，在Web应用程序中，所有Shiro可访问的Web请求都必须通过 `ShiroFilter`，该过滤器非常强大，可以基于任何URL路径表达式执行定制的过滤器链。（类似于Servlet编程中的过滤器、SpringMVC中的拦截器/前端控制器）

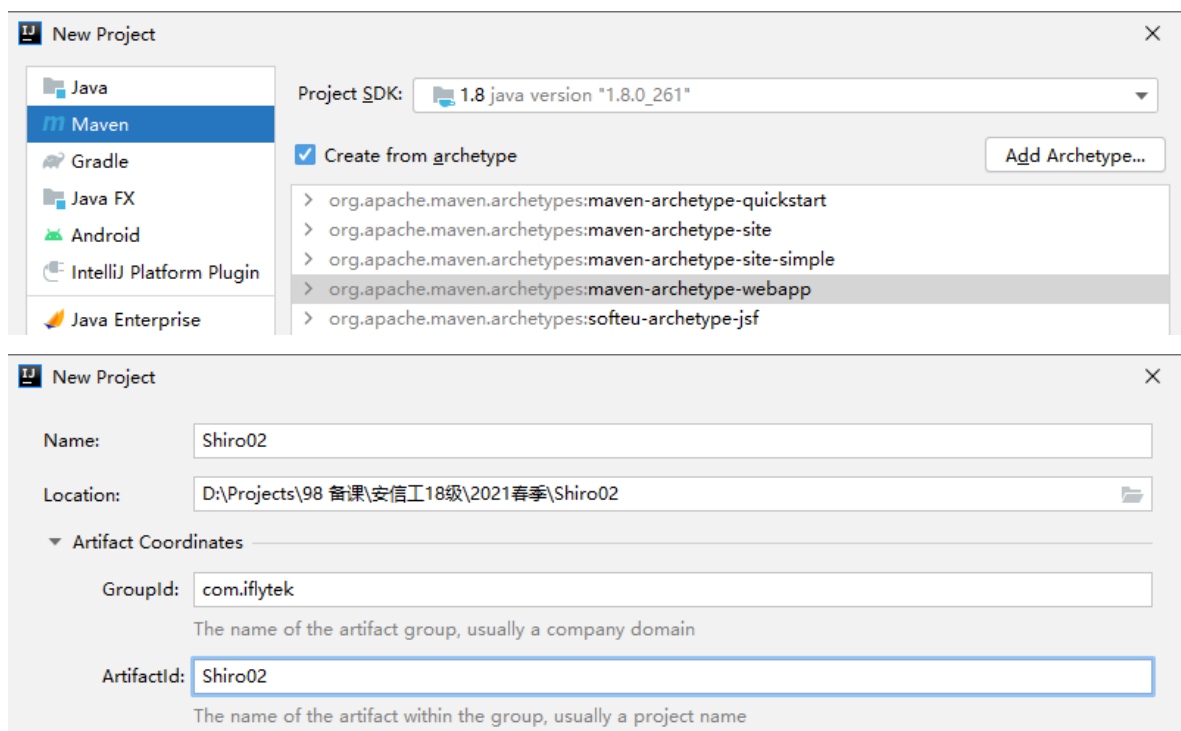
ShiroFilter的工作模式：



1.创建Web项目

1) 创建项目

首先还是需要有一个Web项目，在IDEA中使用Maven下的Web原型骨架来创建项目：



创建完项目后，补充 **java**目录 和 **resources**目录

2) 添加依赖

添加相关依赖：servlet、jsp、spring、springMVC、shiro-core、shiro-web

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
```



```

        <version>5.3.1</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>5.3.1</version>
    </dependency>
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>javax.servlet-api</artifactId>
        <version>4.0.1</version>
    </dependency>
    <dependency>
        <groupId>javax.servlet.jsp</groupId>
        <artifactId>javax.servlet.jsp-api</artifactId>
        <version>2.3.3</version>
    </dependency>
    <dependency>
        <groupId>org.apache.shiro</groupId>
        <artifactId>shiro-core</artifactId>
        <version>1.7.1</version>
    </dependency>
    <dependency>
        <groupId>org.apache.shiro</groupId>
        <artifactId>shiro-web</artifactId>
        <version>1.7.1</version>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-simple</artifactId>
        <version>1.7.30</version>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>jcl-over-slf4j</artifactId>
        <version>1.7.30</version>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-log4j12</artifactId>
        <version>1.7.30</version>
    </dependency>
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>1.18.12</version>
    </dependency>
</dependencies>

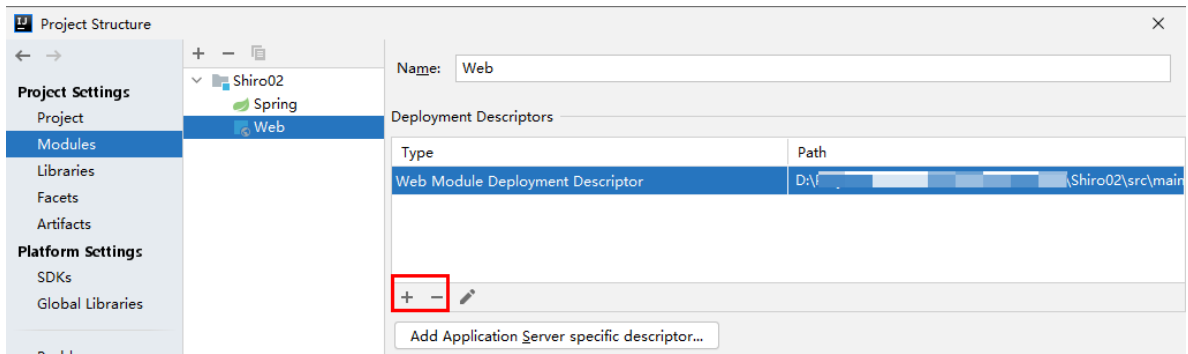
```

拷贝log4j.properties

3) 配置前端控制器

默认的webapp原型骨架中的版本较老，需要删除web.xml后重新添加

使用快捷键：`CTRL + SHIFT + ALT + S` 调出项目结构窗口，在 **Modules** 模块中找到 **Web** 模块，在此处删除现有文件后，重新选择版本后添加新文件，注意删除后和添加后点击 **Apply** 应用。

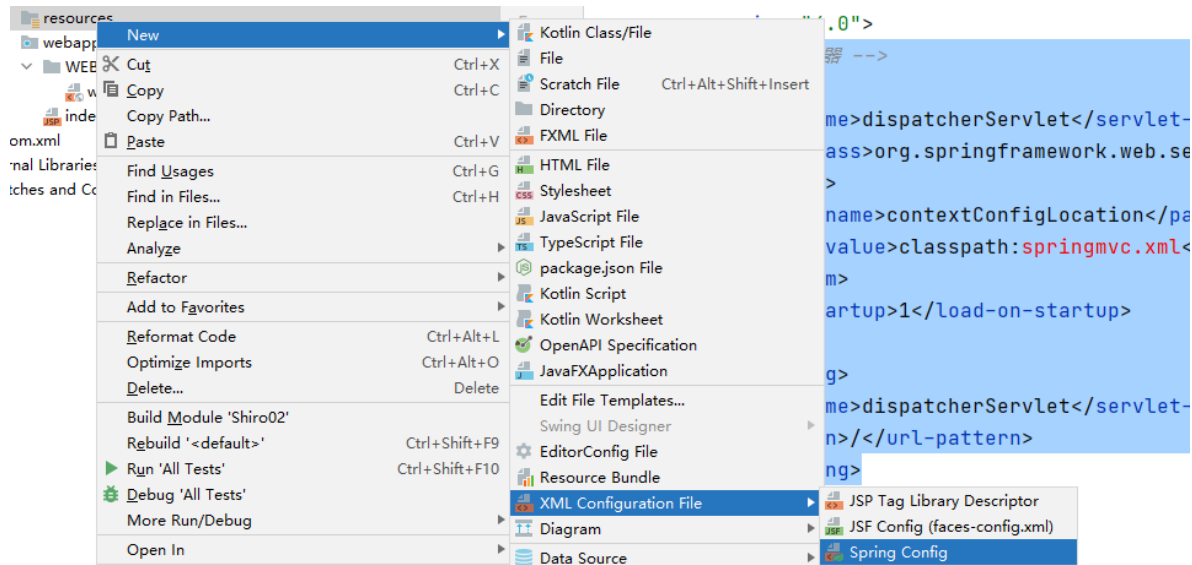


<!-- 配置前端控制器 -->

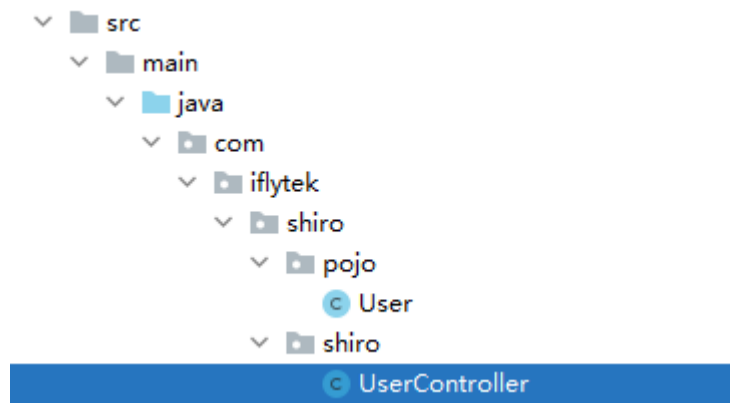
```
<servlet>
    <servlet-name>dispatcherServlet</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:springmvc.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>dispatcherServlet</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
```

4) 配置SpringMVC

新建 `springmvc.xml` 配置文件



```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/mvc
        https://www.springframework.org/schema/mvc/spring-mvc.xsd
        http://www.springframework.org/schema/context
```

User.java

```
@Data
public class User {
    private String username;
    private String password;
}
```

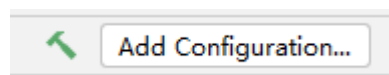
UserController.java

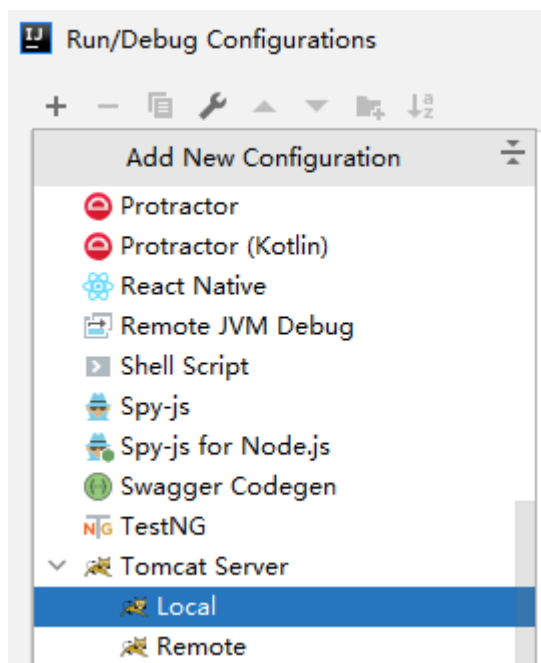
```
@Controller
@RequestMapping("/user")
public class UserController {

    @GetMapping("/login")
    public String login() {
        System.out.println("get login");
        return "login";
    }

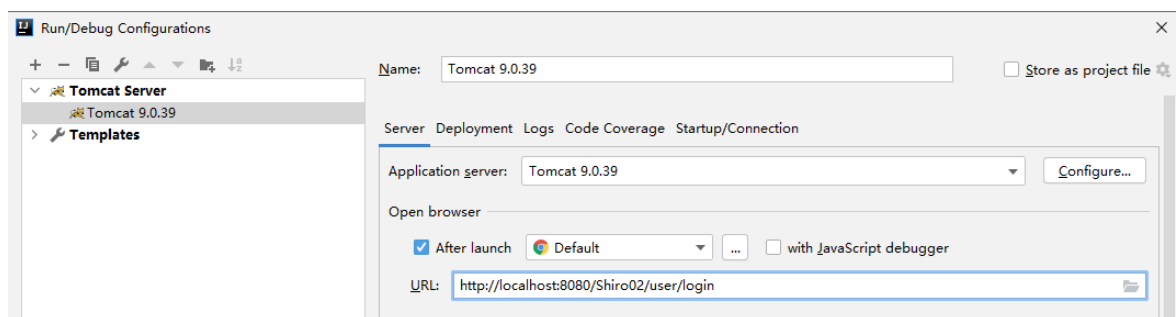
    @PostMapping("/login")
    public String doLogin(User user) {
        System.out.println("post login");
        return "index"; /*登录成功，跳转到首页*/
    }
}
```

7) 配置运行环境





在配置页面中将启动地址修改为登录URL



运行测试。

2.集成Shiro

1) 配置ShiroFilter

在web.xml中配置ShiroFilter：

```
<!-- 1.在项目的最外层构建访问控制层 2.在启动时初始化shiro环境（由下面的listener帮助完成） -->
<filter>
    <filter-name>shiroFilter</filter-name>
    <filter-class>org.apache.shiro.web.servlet.ShiroFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>shiroFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>

<!-- 项目启动时，加载web-inf或者classpath下的shiro.ini，并构建WebSecurityManager
      构建所有配置中使用的过滤器链（anon、authc等），ShiroFilter会获取此过滤器链 -->
<listener>
    <listener-class>org.apache.shiro.web.env.EnvironmentLoaderListener</listener-
class>
</listener>
```

2) 配置Shiro

添加shiro.ini，并增加两个区块（main和urls）：

```
# 定义用户信息
[users]
zhangsan=123,admin
lisi=456,manager,seller
wangwu=789,clerk

# 定义角色信息
[roles]
admin=*
manager=user:*
clerk=user:query,user:detail:query

# 配置shiro对象
[main]
#没有身份认证时，跳转地址
shiro.loginUrl = /user/login
#角色或权限校验不通过时，跳转地址
shiro.unauthorizedUrl = /author/error
#登出后的跳转地址，回首页
shiro.redirectUrl = /

[urls]
/user/login = anon
```

3) 控制器中处理登录

UserController.java

```
@PostMapping("/login")
public String doLogin(User user) {
    System.out.println("post login");
    // 获取subject 调用login
    Subject subject = SecurityUtils.getSubject();
    // 创建用于登录的令牌
    AuthenticationToken token = new UsernamePasswordToken(user.getUsername(),
user.getPassword());
    // 登录
    subject.login(token);/*登录没有返回值，但是可能会抛出异常，这里我们不处理，由异常解析器处理*/
    return "index"; /*登录成功，跳转到首页*/
}
```

4) 测试

启动项目后，使用shiro.ini文件中配置的账号和密码进行登录测试。

- 正常登录

欢迎登录

- 用户名:
- 密码:
-

欢迎登录

- 使用错误的密码

HTTP状态 500 - 内部服务器错误

类型 异常报告

消息 Request processing failed; nested exception is org.apache.shiro.authc.**IncorrectCredentialsException**: Submitted credentials for token credentials.

描述 服务器遇到一个意外的情况，阻止它完成请求。

例外情况

```
org.springframework.web.util.NestedServletException: Request processing failed; nested exception is org.a
    org.springframework.web.servlet.FrameworkServlet.processRequest(FrameworkServlet.java:1014)
    org.springframework.web.servlet.FrameworkServlet.doPost(FrameworkServlet.java:909)
    javax.servlet.http.HttpServlet.service(HttpServlet.java:652)
    ...
```

- 使用不存在的账号

HTTP状态 500 - 内部服务器错误

类型 异常报告

消息 Request processing failed; nested exception is org.apache.shiro.authc.**UnknownAccountException**: Realm [org.ap
[org.apache.shiro.authc.UsernamePasswordToken - zhangsan11, rememberMe=false].

描述 服务器遇到一个意外的情况，阻止它完成请求。

例外情况

```
org.springframework.web.util.NestedServletException: Request processing failed; nested ex
    org.springframework.web.servlet.FrameworkServlet.processRequest(FrameworkServlet.java:1014)
    ...
```

5) 异常处理

这两个异常可以通过建立异常解析器，并配置到SpringMVC中，实现良好的页面跳转

1、创建异常解析器

New Java Class	
<input checked="" type="radio"/> C	resolver.MyExceptionHandler
<input checked="" type="radio"/> C	Class
<input type="radio"/> I	Interface
<input type="radio"/> E	Enum
<input type="radio"/> @	Annotation
<input type="radio"/> J	JavaFXApplication

```
public class MyExceptionHandler implements HandlerExceptionHandler {

    @Override
    public ModelAndView resolveException(HttpServletRequest request,
        HttpServletResponse response, Object handler,
        ...
```

```

        Exception ex) {
            System.out.println(ex.getClass());
            ex.printStackTrace();
            ModelAndView mv = new ModelAndView();
            if(ex instanceof IncorrectCredentialsException || ex instanceof
UnknownAccountException) {
                // 跳转到登录页面，重新登录
                mv.setViewName("redirect:/user/login");
            }
            return mv;
        }
    }
}

```

2、在springmvc.xml中配置异常解析器

```

<!-- 配置异常处理解析器 -->
<bean class="com.iflytek.shiro.resolver.MyExceptionResolver"></bean>

```

再次测试。

3.过滤器规则

在web应用的使用过程中，有些页面是不需要进行访问控制的，例如登录页面是不需要在访问的时候还要去检查用户是否登录了，而有些页面例如查询所有用户，就需要登录，通常在登录了还需要检查是否具备相应的权限才知道能不能访问。这些都涉及到过滤器规则的配置。

1) 添加控制器方法

模拟一个“查询所有用户”的控制器方法：

```

@GetMapping("/all")
public String queryAllUser(){
    System.out.println("query all");
    return "all"; // 没有service，模拟直接跳转到页面
}

```

补充all.jsp仅作简单展示。

那么这里的路径/user/all不是谁都能访问的，就算登录了还得看看有没有权限，接下来就需要配置规则。

2) 常用过滤器

在【urls】中配置了路径和访问该路径的过滤器，例如：

```
/user/login = anon
```

继续加一个访问所有用户信息的过滤器规则：

```
/user/all = 过滤器名称
```

每个过滤器都有一个类，但是我们不直接写类，写其对应的过滤器名称就行了，这个过滤器名称是固定的，系统会自动会去做映射。

常用的过滤器有：

- anon

- 这个过滤器表示匿名，anonymous的前几个字母，表示这个路径不需要任何身份就可以访问，例如首页、登录、注册，如果不写，默认就是匿名的
- `authc`
 - authentication的简写，表示如果访问指定路径，需要已经登录过
- `roles`、`perms`
 - 表示需要指定的角色和权限

3) 配置过滤器

```
[urls]
/user/login = anon
/user/all = authc,perms["user:query"]
```

表示登录的用户，且具有user:query权限就可以访问这个路径

4) 测试

- 启动项目后，直接访问路径(<http://localhost:8080/Shiro02/user/all>)后将会跳转到登录页，此时并不是异常解析器的作用（没有输出异常信息），是因为在 `shiro.ini` 的配置里，在 `[main]` 区块中配置了 `shiro.loginUrl = /user/login`，当没有身份认证时，将会跳转到该地址。
- 当登录后，再次访问，可以发现能够通过路径跳转到 `all.jsp`。
- 修改过滤器规则，将权限修改为：`/user/all = authc,perms["user:queryall"]`，启动项目，使用 `clerk` 角色进行登录后再次访问，可以发现跳转到了 <http://localhost:8080/Shiro02/author/error> 页面：

HTTP状态 404 - 未找到

类型 状态报告

- **消息** 请求的资源[/Shiro02/author/error]不可用

描述 源服务器未能找到目标资源的表示或者是不愿公开一个已经存在的资源表示。

Apache Tomcat/9.0.39

- 这里同样是因为在 `shiro.ini` 的配置里，在 `[main]` 区块中配置了 `shiro.unauthorizedUrl = /author/error`，当前的角色不具备相应的权限，因此角色或权限检验不通过时就跳转到了这个地址。
- 可以对该地址建立映射处理的控制器方法，专门用来跳转到提示页面：

```
@RequestMapping("/author/error")
public String authorError() {
    System.out.println("权限不足!");
    return "authorerror";
}
```

- 建立 `authorerror.jsp`

```
<body>
    <h2>权限不足，请联系管理员</h2>
</body>
```

- 修改 [main] 配置部分（因为控制器类的一级路径）`shiro.unauthorizedUrl = /user/author/error`

- 再次登录后测试：**权限不足，请联系管理员**

5) 登出规则

在 [urls] 中可以配置 `/user/logout = logout`，登出规则比较特殊，它不需要在控制器中编写方法来映射处理，直接访问配置好的路径就可以完成登出过程了。

在 `index.html` 中添加登出链接：

```
<a href="{pageContext.request.contextPath}/user/logout">登出</a>
```

点击该链接即可实现登出。

配合 [main] 中的配置 `shiro.redirectUrl = /` 在登出后跳转到首页，或者修改为 `shiro.redirectUrl = /user/login` 实现登出后跳转到登录页。

6) 细节探讨

```
# 如下格式："访问路径 = 过滤器"
# 【1. 访问路径：? * **细节】
# /user/login/page, /user/login/logic 是普通路径
# /user/* 代表/user后还有一级任意路径：/user/a, /user/b, /user/xxx
# /user/** 代表/user后还有任意多级任意路径：/user/a, /user/a/b/c/, /user/xx/xxx/xxxx
# /user/hello? 代表hello后还有一个任意字符：/user/helloa, /user/hellob, /user/hellox
# 【2. 过滤器细节】
# anno => 不需要身份认证
# authc => 指定路径的访问，会验证是否已经认证身份，如果没有则会强制转发到最上面配置的loginUrl上
#          注意登录逻辑本身不能被认证拦截，否则会无法登录
# logout => 访问指定的路径，可以登出，不用定义handler
# roles["manage", "seller"] => 指定路径的访问需要subject有这两个角色
# perms["user:update", "user:delete"] => 指定路径的访问需要subject有这两个权限
/user/query = authc
/user/update = authc,roles["manage","seller"]
/user/delete = authc,perms["user:update","user:delete"]
# 这些一个个的配置就是组成了一个过滤器链，从上到下一个个进行校验
# 其他路径都需要身份认证【此路径需谨慎使用，一般不建议这样使用，最好就是谁需要就写谁，不需要就不写】
# /** = authc
# 【3. 注意】
# url的匹配，是从上到下的，一旦找到了匹配项就会停止，所以通配范围大的url要往后放
# 例如/user/delete和/user/**
```

4. 默认过滤器

运行 Web 应用程序时，Shiro 会创建一些有用的默认Filter实例，并自动在 [main] 部分中使它们可用。

自动可用的默认过滤器实例由 `org.apache.shiro.web.filter.mgt.DefaultFilter` 定义，枚举的 `name` 字段是可用于配置的名称：

Filter Name	Class
anon	<code>org.apache.shiro.web.filter.authc.AnonymousFilter</code>
authc	<code>org.apache.shiro.web.filter.authc.FormAuthenticationFilter</code>
authcBasic	<code>org.apache.shiro.web.filter.authc.BasicHttpAuthenticationFilter</code>
authcBearer	<code>org.apache.shiro.web.filter.authc.BearerHttpAuthenticationFilter</code>
logout	<code>org.apache.shiro.web.filter.authc.LogoutFilter</code>
noSessionCreation	<code>org.apache.shiro.web.filter.session.NoSessionCreationFilter</code>
perms	<code>org.apache.shiro.web.filter.authz.PermissionsAuthorizationFilter</code>
port	<code>org.apache.shiro.web.filter.authz.PortFilter</code>
rest	<code>org.apache.shiro.web.filter.authz.HttpMethodPermissionFilter</code>
roles	<code>org.apache.shiro.web.filter.authz.RolesAuthorizationFilter</code>
ssl	<code>org.apache.shiro.web.filter.authz.SslFilter</code>
user	<code>org.apache.shiro.web.filter.authc.UserFilter</code>

过滤器描述

过滤器名称	描述	例子
anon	没有参数，表示可以匿名使用	<code>/admin/** = anon</code>
authc	没有参数，表示需要认证（登录）才能使用	<code>/user/** = authc</code>
authcBasic	没有参数，表示需要通过httpBasic验证，如果不通过，跳转到登录页面	<code>/user/** = authcBasic</code>
logout	注销登录时，完成一定的功能：任何现有的Session都将会失效，而且任何身份都将会失去关联（在Web应用中，RememberMe Cookie也将被删除）	
noSessionCreation	阻止在请求期间创建新的会话，以保证无状态	
perms	参数可以写多个，写多个时必须加上引号，并且参数之间用逗号分隔。当有多个参数时，必须每个参数都通过才通过，相当于isPermittedAll()方法	<code>/admin/** = perms[user:add:*</code> <code>/admin/user/**=perms["user:add:*,user:modify:*"]</code>
port	指定请求访问的接口，如果不匹配跳转到登录页面	<code>/admin/**=port[8088]</code>
rest	根据请求的方法	<code>/admin/user/**=perms[user:method]</code> ,其中method为post、get等
roles	角色过滤器，判断当前用户是否具有指定的角色。参数可以写多个，多个时必须加上引号，且参数之间用逗号分隔，当有多个参数时，每个参数通过才算通过，相当于hasAllRoles()方法	<code>/admin/**=roles["admin,guest"]</code>
ssl	没有参数，表示安全的url请求，协议为https	
user	没有参数，表示必须存在用户	

通过ShiroFilter和定义在shiro.ini中的配置信息，即可在项目接受用户访问时，对身份、角色、权限进行访问控制。

自定义Realm

问题

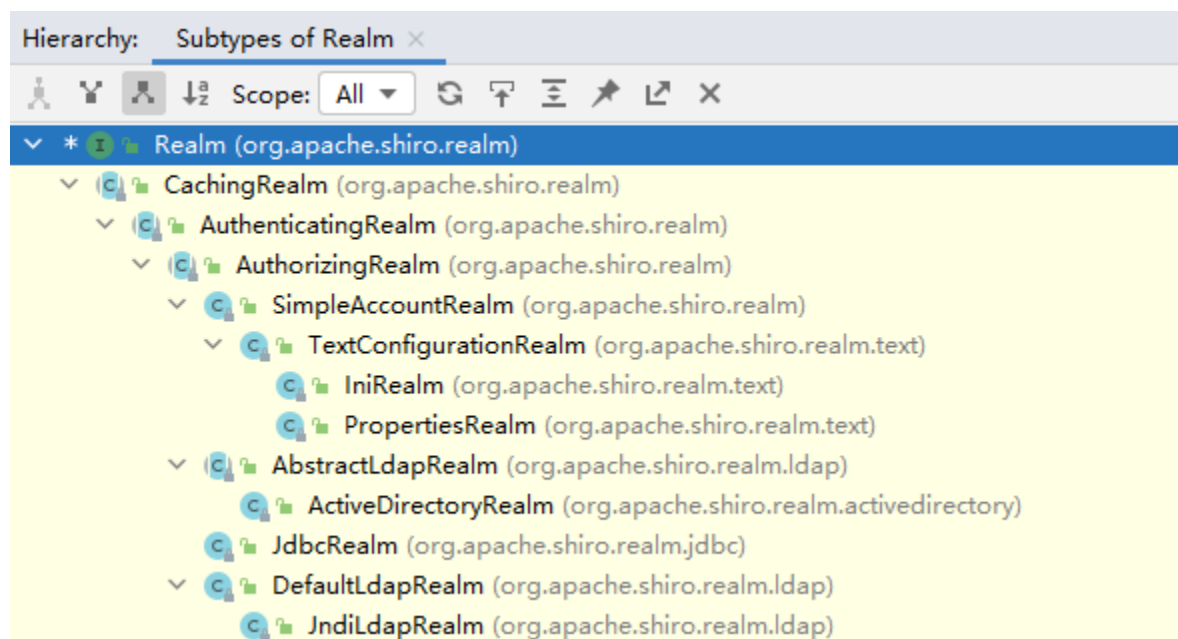
目前所有的用户、角色、权限数据都在ini文件中，不利于管理。而实际项目开发中这些信息应该存储在数据库中，所以需要为这3类信息建表（根据RBAC模型，需要建立用户表、角色表、权限表、用户角色表、角色权限表）。

那 `SecurityManager` 怎么去查数据库里的数据呢？这就需要自定义 `Realm` 来完成数据库查询操作。

Realm的作用和继承关系

`Realm` 的职责是为shiro加载用户、角色、权限数据，供shiro内部校验。之前定义在ini中的数据，默认是由IniRealm加载，现在存储在数据库中的数据，则需要自定义Realm去加载。没有必要在Realm中定义大量的查询数据的代码，可以为Realm定义好查询数据的Dao和Service。

要创建自定义的Realm，首先要选择需要的父类。Realm本身是一个接口：`org.apache.shiro.realm.Realm`，该接口拥有多个子类：



其中 `IniRealm` 是默认的Realm，负责加载 `shiro.ini` 中的[users]和[roles]等信息，当shiro需要用户角色权限信息时，就通过IniRealm获取。

自定义Realm有两个父类可以选择：

- `AuthenticatingRealm`：负责做身份认证
- `AuthorizingRealm`：继承了上面的类，具备认证功能，同时也可以进行权限校验。

所以一般选择 `AuthorizingRealm` 作为父类。

创建自定义Realm类

创建 `MyRealm.java` 类，并继承 `AuthorizingRealm`：

```
public class MyRealm extends AuthorizingRealm {  
    @Override  
    protected AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection  
principalCollection) {  
        return null;  
    }  
  
    @Override  
    protected AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken  
authenticationToken) throws AuthenticationException {  
        return null;  
    }  
}
```

继承这个类后有2个方法必须要重写，从名字上可以看出一个是查询权限信息的，一个是查询身份信息的。

- `doGetAuthorizationInfo` 方法
 - 何时触发：当ShiroFilter过滤器进行规则检查，或者在页面中使用了Shiro的标签，访问了某个地址需要检查是否具备相应的权限
 - 作用：查询权限信息
 - 查询方式：通过用户名查询该用户的角色、权限信息
- `doGetAuthenticationInfo` 方法
 - 何时触发：当调用subject.login(token)时
 - 作用：查询身份信息，封装对象返回即可，不需要任何比对校验
 - 查询方式：通过用户名查询用户信息（用户名在token中，由SecurityManager调用login时传递过来）

自定义的Realm将会代替IniRealm，因此将脱离shiro.ini文件，在shiro.ini文件中除了用户、角色和权限信息之外，其他的例如 `[main]`、`[urls]` 等的配置就需要通过其他的方式来完成。

另外IniRealm作为默认的Realm，由Shiro框架在启动时自行创建，而真实项目中自然不再使用默认的IniRealm，所以也需要配置自定义的Realm。

编写自定义的Realm中的两个方法：获取认证和获取授权中，需要做的事情就是通过编写代码（Service、Dao）读取项目中的用户、角色、权限信息，封装成所需格式后返回相应的对象，之后则由 `SecurityManager` 自动取完成相应的操作，例如调用 `subject.login(token)` 时，`SecurityManager` 负责调用获取身份认证的方法，并完成账号和密码的比对，成功则继续，失败则抛出异常。

自定义Realm的具体实现在后续案例中来完成。

SSM框架集成

RBAC模型建立

继续基于《图书管理系统》来完成权限管理模型的建立。

1) 权限模型设计

早期的《图书管理系统》中并未考虑权限管理，单纯从用户角度进行区分，并且需要用户在登录时选择自己的身份，然后根据登录用户选择的是管理员或者读者，再从相应的数据库表中完成数据查询，之后根据结果跳转到相应的首页。

而从权限管理的角度来看，这种方式则不合适，所以按照权限管理理论知识，可以做出如下模型建立：

- 1、用户表t_user：id、登陆名、密码、用户名、状态
- 2、角色表t_role：id、角色名
- 3、权限表t_permission：id、权限名
- 4、用户角色表t_user_role：id、用户id、角色id
- 5、角色权限表t_role_permission：id、角色id、权限id

例如，用户登录名 `zhangsan`，其用户名 `张三`，拥有角色 `manager`，对应的权限有 `manager:*`。

这里要注意下，登录名和用户名是两个东西，有些项目应用中允许重复的用户名，则查询相关的身份和权限就根据其登录名来，有些项目应用中登录名和用户名都是不允许重复的，那么此时可以根据用户的登录名或者用户名来查询其身份和权限都可以。

2) 数据库表创建

- 1、用户表t_user：

<input type="checkbox"/>	id	loginname	password	username	status
<input type="checkbox"/>	1	zhangsan	12345	张三	1
<input type="checkbox"/>	2	lisi	12345	李四	1
*	(Auto)	(NULL)	(NULL)	(NULL)	(NULL)

- 2、角色表t_role：

<input type="checkbox"/>	id	rolename
<input type="checkbox"/>	1	manager
<input type="checkbox"/>	2	reader
*	(Auto)	(NULL)

- 3、权限表t_permission：

<input type="checkbox"/>	id	permissionname
<input type="checkbox"/>	1	manager:*
<input type="checkbox"/>	2	reader:*
*	(Auto)	(NULL)

- 4、用户角色表t_user_role：

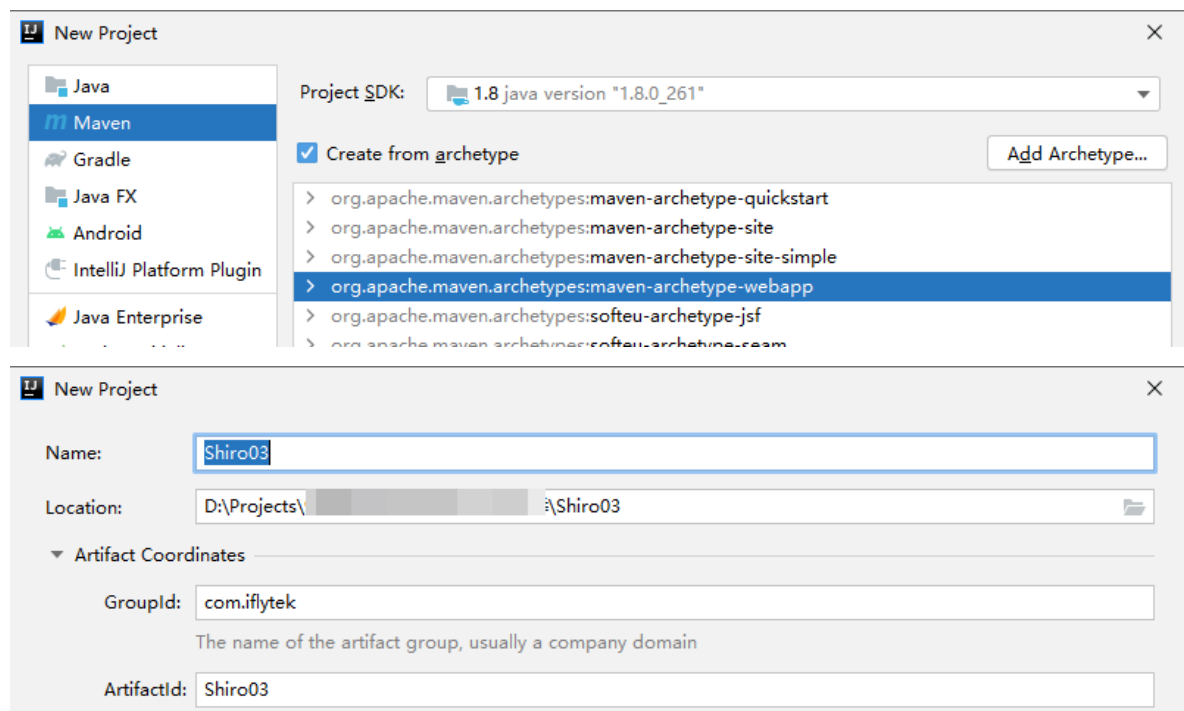
<input type="checkbox"/>	id	userid	roleid
<input type="checkbox"/>	1	1	1
<input type="checkbox"/>	2	2	2
*	(Auto)	(NULL)	(NULL)

- 5、角色权限表t_role_permission：

<input type="checkbox"/>	id	roleid	permissionid
<input type="checkbox"/>	1	1	1
<input type="checkbox"/>	2	2	2
*	(Auto)	(NULL)	(NULL)

创建SSM项目

1) 创建maven web项目



2) 添加依赖

```
<properties>
  <junit.version>5.7.0</junit.version>
  <servlet.version>4.0.1</servlet.version>
  <jsp.version>2.3.3</jsp.version>
  <spring.version>5.3.1</spring.version>
</properties>

<dependencies>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>${junit.version}</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>${servlet.version}</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>javax.servlet.jsp</groupId>
    <artifactId>javax.servlet.jsp-api</artifactId>
    <version>${jsp.version}</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
```

```
<version>${spring.version}</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>${spring.version}</version>
</dependency>
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.9.6</version>
    <scope>runtime</scope>
</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>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.47</version>
</dependency>
<dependency>
    <groupId>org.apache.taglibs</groupId>
    <artifactId>taglibs-standard-impl</artifactId>
    <version>1.2.5</version>
</dependency>
<dependency>
    <groupId>org.apache.taglibs</groupId>
    <artifactId>taglibs-standard-spec</artifactId>
    <version>1.2.5</version>
</dependency>
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.2.3</version>
</dependency>
<dependency>
    <groupId>org.mybatis</groupId>
    <artifactId>mybatis</artifactId>
    <version>3.5.6</version>
</dependency>
<dependency>
    <groupId>org.mybatis</groupId>
    <artifactId>mybatis-spring</artifactId>
    <version>1.3.2</version>
</dependency>
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>jcl-over-slf4j</artifactId>
    <version>1.7.30</version>
</dependency>
```

```

<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-log4j12</artifactId>
    <version>1.7.30</version>
</dependency>
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.12</version>
    <scope>provided</scope>
</dependency>
</dependencies>

```

拷贝log4j.properties

3) 配置

1、web.xml（需要在project structure -- facets中先移除后再添加）：

```

<!-- 配置前端控制器 -->
<servlet>
    <servlet-name>dispatcherServlet</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:springmvc.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>dispatcherServlet</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>

<!-- 配置解决中文乱码的编码过滤器 -->
<filter>
    <filter-name>characterEncodingFilter</filter-name>
    <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-
class>
    <!-- 设置过滤器编码属性 -->
    <init-param>
        <param-name>encoding</param-name>
        <param-value>UTF-8</param-value>
    </init-param>
</filter>
<filter-mapping>
    <filter-name>characterEncodingFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>

<!-- 配置Spring的监听器，默认加载WEB-INF目录下的applicationContext.xml -->
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-
class>
</listener>
<!-- 配置加载类路径中的Spring配置文件 -->
<context-param>
    <param-name>contextConfigLocation</param-name>

```



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

2、 applicationContext.xml

```
<?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:context="http://www.springframework.org/schema/context"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"
       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">

    <!-- 开启注解扫描，扫描service和dao层的注解，要忽略web层注解，因为web层让SpringMVC框架去管理 -->
    <context:component-scan base-package="com.iflytek.shiro">
        <!-- 配置要忽略的注解 -->
        <context:exclude-filter type="annotation"
expression="org.springframework.stereotype.Controller"/>
    </context:component-scan>

    <!-- Spring整合MyBatis -->
    <!-- 1. 配置数据源 -->
    <context:property-placeholder
        location="classpath:db.properties"/>
    <!-- 配置druid连接池 -->
    <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
        <!-- 基本属性 driver、url、user、password -->
        <property name="driverClassName" value="${jdbc.driver}"></property>
        <property name="url" value="${jdbc.url}"></property>
        <property name="username" value="${jdbc.username}"></property>
        <property name="password" value="${jdbc.password}"></property>

        <!-- 其他连接参数... -->
    </bean>
    <!-- 2. 配置SqlSessionFactory工厂对象 -->
    <bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
        <property name="dataSource" ref="dataSource"/>
        <property name="typeAliasesPackage" value="com.iflytek.shiro.pojo"></property>
    </bean>

    <!-- 3. 配置dao接口所在的包 -->
    <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
        <property name="basePackage" value="com.iflytek.shiro.dao"/>
    </bean>

</beans>
```

3、 springmvc.xml

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

```

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans
https://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/mvc
https://www.springframework.org/schema/mvc/spring-mvc.xsd
http://www.springframework.org/schema/context
https://www.springframework.org/schema/context/spring-context.xsd">

    <!-- 开启注解扫描，只扫描Controller注解 -->
    <context:component-scan base-package="com.iflytek.shiro">
        <context:include-filter type="annotation"
expression="org.springframework.stereotype.Controller"/>
    </context:component-scan>

    <!-- 配置视图解析器 -->
    <bean id="viewResolver"
        class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <!-- JSP文件所在的目录 -->
        <property name="prefix" value="/WEB-INF/pages/" />
        <!-- 文件的后缀名 -->
        <property name="suffix" value=".jsp" />
    </bean>

    <!-- 开启SpringMVC框架注解支持 -->
    <mvc:annotation-driven /></mvc:annotation-driven>

</beans>

```

4、db.properties

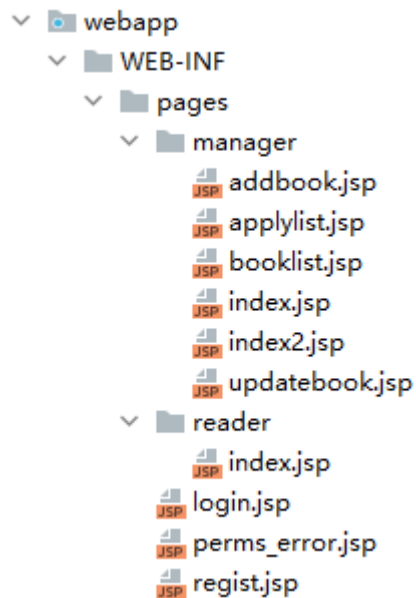
```

jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/bookmanagesystem?characterEncoding=utf-8&zeroDateTimeBehavior=convertToNull
jdbc.username=root
jdbc.password=root

```

4) 创建JSP页面

JSP页面仅作模拟



manager/index.jsp

```
<body>
    <h2>管理员首页</h2>
    <a href="${pageContext.request.contextPath}/logout">登出</a>
    <a href="${pageContext.request.contextPath}/manager/addbook">添加图书</a>
    <a href="${pageContext.request.contextPath}/manager/applylist">查看注册申请</a>
</body>
```

manager/booklist.jsp

```
<body>
    <h2>图书列表</h2>
    <a href="${pageContext.request.contextPath}/manager/deletebook/1">删除图书</a>
    <a href="${pageContext.request.contextPath}/manager/updatebook">修改图书</a>
    <a href="${pageContext.request.contextPath}/manager/query">查询图书</a>
</body>
```

5) 创建控制器类

LoginController.java:

```
@Controller
public class LoginController {

    @GetMapping("/login")
    public String getLogin() {
        return "login";
    }

    @PostMapping("/login")
    public String doLogin() {
        return "login";
    }

    @GetMapping("/regist")
    public String getRegist(){
        return "regist";
    }
}
```

ManagerController.java

```

@Controller
@RequestMapping("/manager")
public class ManagerController {

    @GetMapping("/index")
    public String getIndex(){
        return "manager/index";
    }

    @GetMapping("/booklist")
    public String getBookList(){
        return "manager/booklist";
    }

    @GetMapping("/applylist")
    public String getApplyList(){
        return "manager/applylist";
    }

    @GetMapping("/addbook")
    public String getAddBook() {
        return "manager/addbook";
    }

    @GetMapping("/updatebook")
    public String getUpdateBook(){
        return "manager/updatebook";
    }
}

```

ReaderController.java

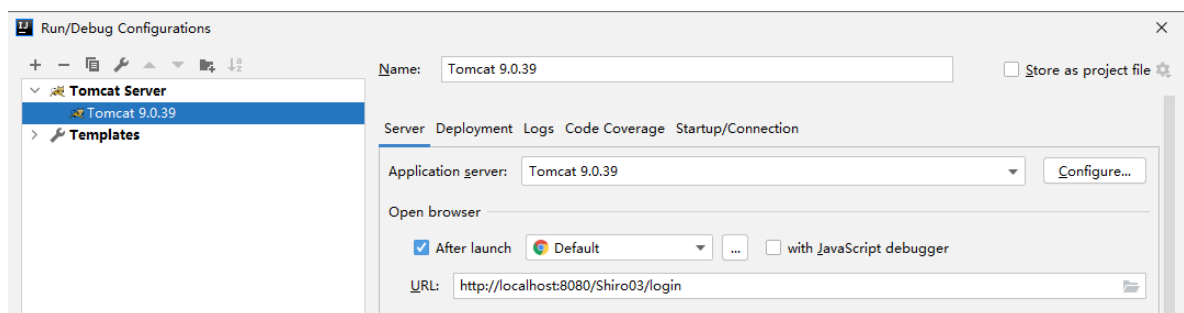
```

@Controller
@RequestMapping("/reader")
public class ReaderController {

    @GetMapping("/index")
    public String getIndex(){
        return "reader/index";
    }
}

```

6) 配置运行环境



7) 测试访问

整合Shiro

Web项目的核心组件都在Spring容器中通过IOC、AOP进行管理，Shiro的各个组件同样需要如此。而且Shiro的组件都是POJO组件，很容易利用Spring管理集成。

1) 添加依赖

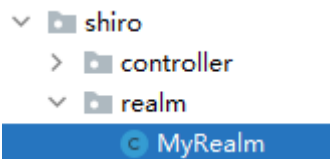
```
<!-- https://mvnrepository.com/artifact/org.apache.shiro/shiro-spring -->
<dependency>
    <groupId>org.apache.shiro</groupId>
    <artifactId>shiro-spring</artifactId>
    <version>1.4.0</version>
</dependency>
```

shiro-spring会传递导入其所依赖的 `shiro-core` 和 `shiro-web`

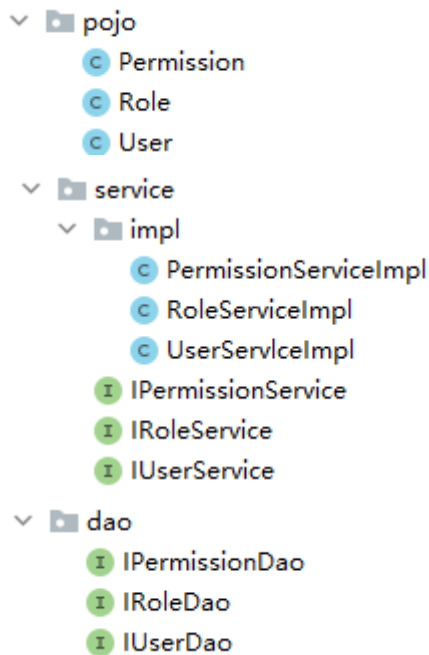
接下来需要把Shiro的核心组件 `SecurityManager`、`Realm` 以及 `ShiroFilter` 都集成到spring中去，并且由于 `SecurityManager` 是需要 `Realm` 提供数据的，因此，需要先将自定义的Realm创建出来。

2) 创建自定义Realm

1、创建MyRealm类



2、创建用户、角色、权限相关的操作类



```
@Data
public class User {
    private int id;
    private String loginname;
    private String password;
    private String username;
```

```

        private int status;
    }

    @Data
    public class Role {
        private int id;
        private String rolename;
    }

    @Data
    public class Permission {
        private int id;
        private String permission;
    }

```

```

public interface IUserService {
    User queryByLoginname(String loginname);
}

public interface IRoleService {
    // 根据登录名查询其对应的所有角色名, Shiro框架在处理的时候传递的都是角色名, 所以这里直接返回字符串集合
    Set<String> queryAllRolenameByLoginname(String loginname);
}

public interface IPermissionService {
    Set<String> queryAllPermissionnameByLoginname(String loginname);
}

```

```

@Service
public class UserServiceImpl implements IUserService {
    @Autowired
    private IUserDao dao;

    @Override
    public User queryByLoginname(String loginname) {
        return dao.queryByLoginname(loginname);
    }
}

@Service
public class RoleServiceImpl implements IRoleService {
    @Autowired
    private IRoleDao dao;

    @Override
    public Set<String> queryAllRolenameByLoginname(String loginname) {
        return dao.queryAllRolenameByLoginname(loginname);
    }
}

@Service
public class PermissionServiceImpl implements IPermissionService {
    @Autowired
    private IPermissionDao dao;

    @Override
    public Set<String> queryAllPermissionnameByLoginname(String loginname) {
        return dao.queryAllPermissionnameByLoginname(loginname);
    }
}

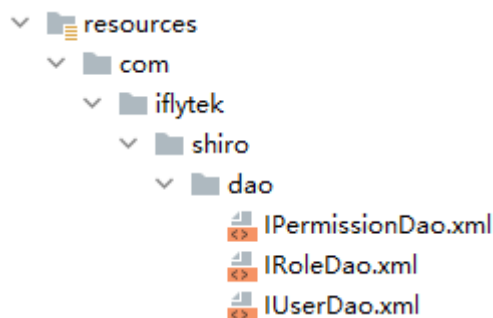
```

```

@Repository
public interface IUserDao {
    User queryByLoginname(String loginname);
}
@Repository
public interface IRoleDao {
    Set<String> queryAllRolenameByLoginname(String loginname);
}
@Repository
public interface IPermissionDao {
    Set<String> queryAllPermissionnameByLoginname(String loginname);
}

```

3、创建Mapper



IUserDao.xml

```

<?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.iflytek.shiro.dao.IUserDao">
    <select id="queryByLoginname" resultType="User">
        select *
        from t_user
        where loginname = #{loginname}
    </select>
</mapper>

```

IRoleDao.xml

```

<?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.iflytek.shiro.dao.IRoleDao">
    <select id="queryAllRolenameByLoginname" resultType="String">
        SELECT b.rolename
        FROM `t_user` AS a,
            `t_role` AS b,
            `t_user_role` AS c
        WHERE a.id = c.userid
            AND b.id = c.roleid
            AND a.loginname = #{loginname}
    </select>
</mapper>

```

IPermissionDao.xml

```

<?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.iflytek.shiro.dao.IPermissionDao">
    <select id="queryAllPermissionnameByLoginname" resultType="String">
        SELECT d.permissionname
        FROM `t_user` AS a,
            `t_role` AS b,
            `t_user_role` AS c,
            `t_permission` AS d,
            `t_role_permission` AS e
        WHERE a.id = c.userid
            AND b.id = c.roleid
            AND b.id = e.roleid
            AND d.id = e.permissionid
            AND a.loginname = #{loginname}
    </select>
</mapper>

```

4、在MyRealm中完成用户、角色、权限信息获取

```

public class MyRealm extends AuthorizingRealm {

    @Autowired
    private IUserService userService;

    @Autowired
    private IRoleService roleService;

    @Autowired
    private IPermissionService permissionService;

    @Override
    protected AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection
principalCollection) {
        // 获取当前用户的用户名
        String loginname = (String) principalCollection.getPrimaryPrincipal();

        // 查询当前用户的权限信息
        Set<String> roles = roleService.queryAllRolenameByLoginname(loginname);
        Set<String> permissions =
permissionService.queryAllPermissionnameByLoginname(loginname);

        // 将查询出的信息封装到 AuthorizationInfo 中
        SimpleAuthorizationInfo info = new SimpleAuthorizationInfo(roles);
        info.setStringPermissions(permissions);

        // 查询后封装到指定对象后返回即可。
        return info;
    }

    @Override
    protected AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken
authenticationToken) throws AuthenticationException {
        // 获取用户登录时发送来的用户名
        String loginname = (String) authenticationToken.getPrincipal();

```



```

        // 查询到用户信息
        User user = userService.queryByLoginname(loginname);
        if (user == null) { // 不存在用户名
            return null; // 返回了null, 则后续流程中会抛出一个UnknownAccountException
        }
        // 将用户信息封装在 AuthenticationInfo 中
        return new SimpleAuthenticationInfo(user.getLoginname(), // 数据库中登录名
            user.getPassword(), // 数据库中的密码
            this.getName()); // realm的标识
    }
}

```

3) Shiro配置

在 `applicationContext.xml` 中添加Shiro的核心组件配置。

```

<!--Shiro配置-->
<!--1.自定义Realm-->
<bean id="myRealm" class="com.iflytek.shiro.realm.MyRealm" />

<!--2.SecurityManager-->
<bean id="securityManager" class="org.apache.shiro.web.mgt.DefaultWebSecurityManager">
    <property name="realm" ref="myRealm" /> <!--注入Realm-->
</bean>

<!--3.ShiroFilter
    生产SpringShiroFilter（持有shiro的过滤相关规则，可以进行请求的过滤校验，校验请求是否合法）-->
<bean id="shiroFilter" class="org.apache.shiro.spring.web.ShiroFilterFactoryBean">
    <!--注入核心对象-->
    <property name="securityManager" ref="securityManager" />
    <property name="loginUrl" value="/login" />
    <property name="unauthorizedUrl" value="/perms/error" />
    <!--过滤器链-->
    <property name="filterChainDefinitions">
        <value>
            /login=anon
            /manager/**=authc,perms[ "manager:*" ]
            /reader/**=authc,perms[ "reader:*" ]
            /logout=logout
        </value>
    </property>
    <!--注入其他的过滤器-->
    <property name="filters">
        <map>
            <entry key="logout" value-ref="logoutFilter" />
        </map>
    </property>
</bean>

<!--配置登出后跳转地址-->
<bean id="logoutFilter" class="org.apache.shiro.web.filter.authc.LogoutFilter">
    <property name="redirectUrl" value="/login" />
</bean>

```

现在Shiro的核心组件都在Spring容器中配置好了，之前配置在 `web.xml` 中的监听器本来的作用就是去加载默认的 `ini` 来构建 `WebSecurityManager`、过滤器链等，现在这些工作都已经被配置好了，shiro环境将会由Spring来进行初始化。

但是如果我们把 `web.xml` 中的过滤器直接注释掉或者删除掉，会有个问题，我们虽然添加了配置，让Spring来管理和生产ShiroFilter，但是我们的web项目在启动时没有一个组件能够在最外层开始shiro的工作，所以我们还需要用另外一个组件在启动时来调用ShiroFilter的doFilter方法，进行访问控制。

`web.xml` 中添加ShiroFilter的启动配置

```
<!-- 会从Spring工厂中获取和它同名的bean（找一个id和这里的filter-name相同的bean）
      接收到请求后调用bean的doFilter方法，进行访问控制-->
<filter>
    <filter-name>shiroFilter</filter-name>
    <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
    <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>
</filter-mapping>
```

4) 实现身份认证

1、登录页面添加表单

```
<form action="{pageContext.request.contextPath}/login" method="post">
    登录名: <input type="text" name="loginname" /> <br>
    密码: <input type="password" name="password" /> <br>
    <input type="submit" value="登录"> <a
href="{pageContext.request.contextPath}/regist">读者注册</a>
</form>
```

2、控制器进行处理

```
@PostMapping("/login")
public String doLogin(User user) {
    // 获取Subject对象
    Subject subject = SecurityUtils.getSubject();

    // 封装用户名密码token
    UsernamePasswordToken token = new UsernamePasswordToken(user.getLoginname(),
user.getPassword());

    // 使用Shiro框架进行登录操作
    subject.login(token); // 登录出错将会抛出异常，进行异常处理，此处略

    // 能执行到这里，说明登录成功
    // 判断登录用户的角色
    if (subject.hasRole("manager")) {
        return "redirect: manager/index";
    } else if (subject.hasRole("reader")) {
        return "redirect: reader/index";
    } else {
        System.out.println("---> other");
    }

    return "login";
}
```

```
}
```

5) 权限测试

- 分别在不登录状态下访问其他页面
- 登录管理员账户状态下访问管理员的页面以及读者页面
- 登录读者账户状态下访问管理员的页面和读者页面

Shiro标签

Shiro提供了很多标签，用于在JSP中做安全校验，完成对页面元素的访问控制。

导入Shiro标签库

在需要使用Shiro标签的JSP页面中进行导入：

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

身份认证标签

身份认证有多种情况：已登录、未登录、未登录但上次登录时留有登录信息。

以某购物商城网站为例：

- 当初始打开时是未登录状态，此时可以浏览商品信息，但是不能进行购买、个人信息等的操作
- 登录之后，可以看到个人的信息，并且也可以下订单购买等操作
- 退出后再次打开网站，此时未登录，但是会看到个人的用户名，只是如果要更进一步操作时仍然需要登录

对应这些情况，Shiro提供了标签对应：

- `<shiro:guest>` 游客，即未登录
- `<shiro:authenticated>` 已登录，可以通过 `<shiro:principal>` 标签获取用户身份信息
- `<shiro:user>` 当已认证通过时或者有 `RememberMe` 时，可以获取用户信息
- `<shiro:notAuthenticated>` 未认证（包含已记住 `RememberMe`）

测试

1、在过滤器规则中将 `manager/index` 单独设置为可以匿名访问

```
/manager/index=anon
```

2、在jsp页面中使用标签控制

```
<body>
  <h2>管理员首页</h2>
  <shiro:user>
    认证通过，或者记住我可以看到下面的内容
    你好: <shiro:principal />
    <shiro:notAuthenticated>请<a href="{pageContext.request.contextPath}/login">
登录</a></shiro:notAuthenticated>
  </shiro:user>
```

```
<shiro:guest>
    游客你好, 请<a href="{pageContext.request.contextPath}/login">登录</a>
    ....
    图书列表
</shiro:guest>
<shiro:authenticated>
    <shiro:principal></shiro:principal>
    <a href="{pageContext.request.contextPath}/logout">登出</a>
    <a href="{pageContext.request.contextPath}/manager/addbook">添加图书</a>
    <a href="{pageContext.request.contextPath}/manager/applylist">查看注册申请</a>
</shiro:authenticated>
</body>
```

角色校验标签

如果是多个角色公用一个JSP页面, 可以通过使用角色校验标签来控制不同的角色看到不同的内容。例如

```
<shiro:hasRole name="manager"> <!-- 是指定角色 -->
    <a href="#">核定年终奖</a>
</shiro:hasRole>
<shiro:lacksRole name="manager"> <!-- 不是指定角色 -->
    <a href="#">等待发年终奖</a>
</shiro:lacksRole>
<shiro:hasAnyRoles name="manager,hr"> <!-- 是其中任何一个角色, 或的关系 -->
    <a href="#">招聘员工</a>
</shiro:hasAnyRoles>
```

权限校验标签

对于不同的角色, 其拥有不同的权限, 也可以使用标签来展示不同的内容, 例如管理员可以看到管理端的部分功能, 而超级管理员可以看到所有功能。

```
<shiro:hasPermission name="user:query"> <!-- 有指定权限 -->
    <a href="#">查看所有用户</a>
</shiro:hasPermission>
<shiro:lacksPermission name="user:query"> <!-- 缺少指定权限 -->
    <a href="#">个人信息</a>
</shiro:lacksPermission>
```

RememberMe

记住我和自动登录不是一回事, 要做自动登录肯定要有记住我, 但是有记住我不代表有自动登录。

记住我是指登录后可以将用户存在cookie中, 下次访问时可以先不登录, 就可以识别身份。之后在确实需要身份认证时(例如购买、支付或其他一些重要操作)再要求用户进行登录, 这样用户体验更好。

另外由于可以保持用户信息, 系统后台可以更好的监控、记录用户行为, 积累数据。

实现

使用subject登录时即可使用记住我：

```
// 获取Subject对象
Subject subject = SecurityUtils.getSubject();

// 封装用户名密码token
UsernamePasswordToken token = new UsernamePasswordToken(user.getLoginname(),
user.getPassword());

// 开启记住我
token.setRememberMe(true);

// 使用Shiro框架进行登录操作
subject.login(token); // 登录出错将会抛出异常，补充异常处理
```

登录测试：



配置超时时间*

记住我默认的超时时间是365天，可以通过配置进行修改设置

1、由于是通过cookie设置的，所以要先设置记住我的cookie信息

```
<bean id="rememberMeCookie" class="org.apache.shiro.web.servlet.SimpleCookie">
    <!--rememberMe是cookie值中的key，其value值为用户名的密文
    cookie["rememberMe":"deleteMe"]此cookie每次登录后都会写出，用于清除之前的cookie
    cookie["rememberMe":username密文]此cookie在登录后也会写出，用于记录最新的username
    这样保证每次登录后重新记录cookie，切换账号时可以记录最新账号-->
    <property name="name" value="rememberMe"/> <!--默认值，也可以修改value为自己的-->
    <property name="httpOnly" value="true" /> <!--默认值，表示cookie只在http请求中可用，防止通过js脚本读取cookie信息-->
    <!--cookie的生命周期，单位：秒-->
    <property name="maxAge" value="2592000"/><!--30天-->
</bean>
```

2、注入到管理器中

```
<bean id="rememberMeManager" class="org.apache.shiro.web.mgt.CookieRememberMeManager">
    <!--注入SimpleCookie-->
    <property name="cookie" ref="rememberMeCookie"/>
</bean>
```

3、将记住我管理器注入到 SecurityManager 中

```
<!--2.SecurityManager-->
<bean id="securityManager" class="org.apache.shiro.web.mgt.DefaultWebSecurityManager">
    <property name="realm" ref="myRealm" /> <!--注入Realm-->
    <property name="rememberMeManager" ref="rememberMeManager" /> <!--注入记住我管理器-->
</bean>
```

其他*

记住我之后，可以在 `<shiro:user>` 标签中获取记住的用户信息

```
<shiro:user>
    认证通过，或者记住我可以看到下面的内容
    你好: <shiro:principal />
    <shiro:notAuthenticated>请<a href="${pageContext.request.contextPath}/login">登录
</a></shiro:notAuthenticated>
</shiro:user>
```

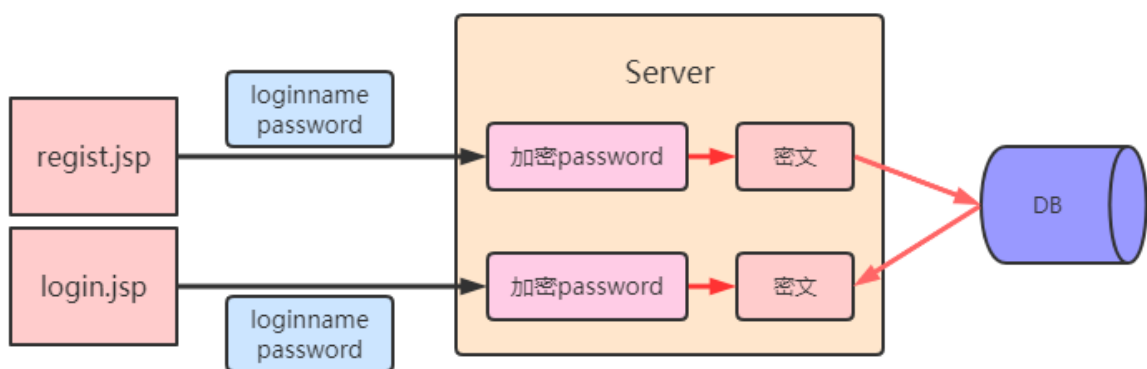
另外，有了记住我之后，可以在登录页面中自动填充记住的用户名

```
用户名: <input type="text" name="loginname" value="<shiro:principal/>"><br>
```

加密

前言

用户的密码是不允许明文存储的，因为一旦数据泄露，用户的隐私信息会完全暴露。密码必须加密，生成密文，然后数据库中只存储用户的密码的密文，用户登录时也是将加密后的密码和数据库中已加密的密码进行比较。



在加密过程中需要使用到一些“不可逆”的技术，如md5、sha等，所谓不可逆是指：使用加密函数将明文加密成密文后，不能够通过密文再反推出明文，因此即使密文泄露密码仍然安全。

Shiro 支持 hash（散列）加密，常见的如 md5、sha 等。

加密过程

基本加密的过程是直接对密码明文进行加密，md5(明文)、sha(明文)，得到密码明文的密文，但是如果明文可能比较简单，导致密文容易被破解（直接对常规的简单密码明文如12345加密后存储到库中，破解就是暴力枚举）。

所以一般会采用 加盐加密。由系统先给定一个随机的盐值 `salt="xxx"`，然后再对密码明文加上盐值之后再行加密：`md5(明文+salt)`、`sha(明文+salt)`，提升密文的复杂度。

由上面的过程进一步提升密文复杂度，可采用 加盐多次迭代加密。如果迭代次数为2，则加密2次：
`md5(明文+salt)=密文a`，`md5(密文a+salt)=最终密文`。进一步提升密文的复杂度和被破解的难度。

关于盐值

盐值可以是一个随机值，利用Java中的UUID工具类可以生成一个随机的UUID值

(`UUID.randomUUID().toString()`)，也可以使用用户的登录名，因为一个系统中的用户的登录名是不会重复的（如果用户名不重复也可以），这样就算两个用户的密码是一样的，但是加入不同的登录名作为盐值后生成的密文也是不同的。

注册加密实现

1) 增加 `t_user` 表字段

盐值是需要存储起来的，否则登录的时候就不知道该如何加密了，所以在数据库中增加盐值字段：

<input type="checkbox"/>	列名	数据类型	长度	默认	主键
<input type="checkbox"/>	id	int	11		<input checked="" type="checkbox"/>
<input type="checkbox"/>	loginname	varchar	50		<input type="checkbox"/>
<input type="checkbox"/>	password	varchar	100		<input type="checkbox"/>
<input type="checkbox"/>	username	varchar	50		<input type="checkbox"/>
<input type="checkbox"/>	status	int	11		<input type="checkbox"/>
<input type="checkbox"/>	salt	varchar	100		<input type="checkbox"/>
<input type="checkbox"/>					<input type="checkbox"/>

考虑最终存储密文时会进行其他的编码，所以此处将密码的长度增加一些。

相应的增加POJO类中的属性

```
@Data
public class User {
    private int id;
    private String loginname;
    private String password;
    private String username;
    private int status;
    private String salt;
}
```

2) 完善注册页面

```
<body>
    <h2>读者注册页</h2>
    <form action="{pageContext.request.contextPath}/regist" method="post">
        登录名: <input type="text" name="loginname" /> <br>
        密码: <input type="password" name="password" /> <br>
        用户名: <input type="text" name="username" /> <br>
        <input type="submit" value="注册">
    </form>
</body>
```

3) 添加增加用户相关操作

IUserService.java

```
int insert(User user);
```

UserServiceImpl.java, 此处需要在调用dao层的插入前对用户信息进行加密和盐值的更新:

```
@Override
public int insert(User user) {
    // 盐值
    String salt = UUID.randomUUID().toString(); // 随机生成盐值
    //String salt = user.getLoginname(); // 使用登录名做盐值

    // 加密: 使用sha256算法, 添加盐值, 迭代10次加密, 加密后进行base64编码
    String pwd = new Sha256Hash(user.getPassword(), salt, 10).toBase64();

    // 更新数据
    user.setPassword(pwd);
    user.setSalt(salt);

    return dao.insert(user);
}
```

IUserDao.java

```
int insert(User user);
```

IUserDao.xml

```
<insert id="insert" parameterType="user">
    insert into t_user(loginname, password, username, status, salt) value (#
    {loginname},#{password},#{username},1,#{loginname})
</insert>
```

4) 完善控制器

LoginController.java

```
@PostMapping("/regist")
public ModelAndView doRegist(User user){
    if (userService.queryByLoginname(user.getLoginname()) != null){
        return new ModelAndView("regist", "msg", "该用户已存在!");
    }
    int rlt = userService.insert(user);
    if (rlt == 1){
        return new ModelAndView("redirect:/login");
    }
    return new ModelAndView("regist", "msg", "用户注册失败!");
}
```

5) 测试注册

读者注册页

登录名:

密码:

用户名:

<input type="checkbox"/>	id	loginname	password	username	status	salt
<input type="checkbox"/>	1	zhangsan	12345	张三	1	(NULL)
<input type="checkbox"/>	2	lisi	12345	李四	1	(NULL)
<input type="checkbox"/>	3	wangwu	eY0odGMWB6yz4IPaU7haUqk38PKIbE5pBpo4bvt	王五	0	dbc9bc5e-b94b-4b09-bdb5-3bd2af780e
<input checked="" type="checkbox"/>	*	(Auto)	(NULL)	(NULL)	(NULL)	(NULL)

密码比对实现

subject.login(token) 执行过程源码解读

0. subject.login(token)

```
LoginController.java x DelegatingSubject.class x
Decompiled .class file, bytecode version: 50.0 (Java 6)
1. 183 public void login(AuthenticationToken token) throws AuthenticationException {
    184     this.clearRunAsIdentitiesInternal();
    185     Subject subject = this.securityManager.login(subject: this, token);
    186     String host = null;
```

```
DelegatingSubject.class x DefaultSecurityManager.class x
bytecode version: 50.0 (Java 6)
2. public Subject login(Subject subject, AuthenticationToken token) throws AuthenticationException {
    AuthenticationInfo info;
    try {
        info = this.authenticate(token);
    } catch (AuthenticationException var7) {
        AuthenticationException ae = var7;
```

```
3. }
    public AuthenticationInfo authenticate(AuthenticationToken token) throws AuthenticationException {
        return this.authenticator.authenticate(token);
    }
```

```
ginController.java x DelegatingSubject.class x DefaultSecurityManager.class x AuthenticatingSecurityManager.class x AbstractAuthenticator.class x
ncompiled .class file, bytecode version: 50.0 (Java 6)
4. public final AuthenticationInfo authenticate(AuthenticationToken token) throws AuthenticationException {
    if (token == null) {
        throw new IllegalArgumentException("Method argument (authentication token)");
    } else {
        log.trace("Authentication attempt received for token [{}]", token);

        AuthenticationInfo info;
        try {
            info = this.doAuthenticate(token);
            if (info == null) {
                String msg = "No account information found for authentication token";
```

5. `protected AuthenticationInfo doAuthenticate(AuthenticationToken authenticationToken) throws AuthenticationException {`
`this.assertRealmsConfigured();`
`Collection<Realm> realms = this.getRealms();`
`return realms.size() == 1 ? this.doSingleRealmAuthentication((Realm)realms.iterator().next(), authenticationToken) : null;`
`}`

6. `protected AuthenticationInfo doSingleRealmAuthentication(Realm realm, AuthenticationToken token) {`
`if (!realm.supports(token)) {`
`String msg = "Realm [" + realm + "] does not support authentication token [" + token + "]. Please use a supported token type.";`
`throw new UnsupportedTokenException(msg);`
`} else {`
`AuthenticationInfo info = realm.getAuthenticationInfo(token);`
`if (info == null) {`
`String msg = "Realm [" + realm + "] was unable to find account data for the " + "submitted AuthenticationToken [" + token + "].";`
`throw new UnknownAccountException(msg);`
`} else {`
`return info;`
`}`
`}`
`}`

7. `public final AuthenticationInfo getAuthenticationInfo(AuthenticationToken token) throws AuthenticationException {`
`AuthenticationInfo info = this.getCachedAuthenticationInfo(token);`
`if (info == null) {`
`info = this.doGetAuthenticationInfo(token);`
`log.debug("Looked up AuthenticationInfo [{}] from doGetAuthenticationInfo", info);`
`if (token != null && info != null) {`
`this.cacheAuthenticationInfoIfPossible(token, info);`
`}`
`} else {`
`log.debug("Using cached authentication info [{}] to perform credentials matching.", info);`
`}`

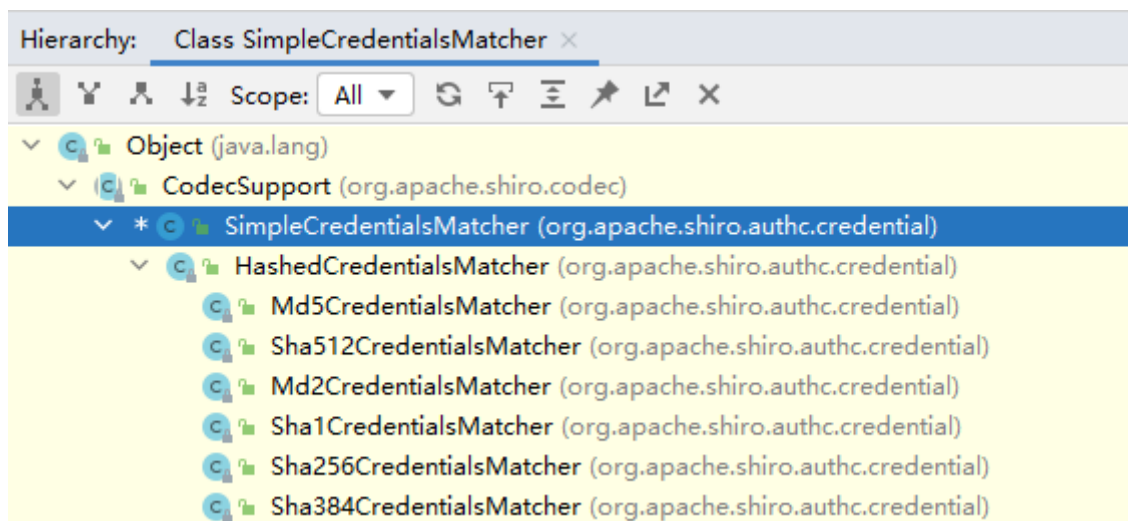
`if (info != null) {`
`this.assertCredentialsMatch(token, info);`
`} else {`
`log.debug("No AuthenticationInfo found for submitted AuthenticationToken [{}]. Returning null.", token);`
`}`
`}`

8. `protected void assertCredentialsMatch(AuthenticationToken token, AuthenticationInfo info) throws AuthenticationException {`
`CredentialsMatcher cm = this.getCredentialsMatcher();`
`if (cm != null) {`
`if (!cm.doCredentialsMatch(token, info)) {`
`String msg = "Submitted credentials for token [" + token + "] did not match the expected credentials for the account [" + info + "].";`
`throw new IncorrectCredentialsException(msg);`
`}`
`} else {`
`// No CredentialsMatcher found, so we assume the credentials match.`
`}`
`}`

9. `SimpleCredentialsMatcher.doCredentialsMatch(AuthenticationToken token, AuthenticationInfo info) {`
`Object tokenCredentials = this.getCredentials(token);`
`Object accountCredentials = this.getCredentials(info);`
`return this.equals(tokenCredentials, accountCredentials);`
`}`

比较器类

`SimpleCredentialsMatcher` 这个比对器类实现了一个 `CredentialsMatcher` 接口，另外它还有一个子类 `HashedCredentialsMatcher`：



```
public class HashedCredentialsMatcher extends SimpleCredentialsMatcher {  
    private String hashAlgorithm;  
    private int hashIterations;  
    private boolean hashSalted;  
    private boolean storedCredentialsHexEncoded;  
}
```

其中需要有加密算法名、迭代次数、是否加盐，是否存储为十六进制，有了这些信息，就可以在登录时将传递的明文封装成比较器对象后由Shiro框架自己转成密文再去比对。

所以想要在加密的系统里面能够正常执行登录，那么就需要明确声明一个组件，叫 **密码比对器**，然后把加密时的细节参数配置好，其他的工作就都交给Shiro框架就行了。

配置比较器

按照之前的登录源码执行流程，比较器实际是在Realm中调用 `doCredentialsMatch` 方法之前去获取的：

```
protected void assertCredentialsMatch(AuthenticationToken token,  
    CredentialsMatcher cm = this.getCredentialsMatcher();  
    if (cm != null) {  
        if (!cm.doCredentialsMatch(token, info)) {  
            String msg = "Submitted credentials for token [" +  
                throw new IncorrectCredentialsException(msg);  
        }  
    }  
}
```

它返回了Realm自身的一个属性，所以配置一个比较器，就是要给Realm去注入一个比较器对象。

```
<!--1.自定义Realm-->  
<bean id="myRealm" class="com.iflytek.shiro.realm.MyRealm" >  
    <property name="credentialsMatcher">  
        <bean class="org.apache.shiro.authc.credential.HashedCredentialsMatcher">  
            <property name="hashAlgorithmName" value="SHA-256" />  
            <property name="hashIterations" value="10" />  
            <property name="storedCredentialsHexEncoded" value="false" />  
        </bean>  
    </property>  
</bean>
```

加密算法、迭代次数要和增加用户时所设定的保持一致。

盐值处理

注意，这里我们并没有配置是否加盐的属性 `hashSalted`，这个属性的getter和setter本身是被弃用了：

```
Deprecated
@Deprecated
public boolean isHashSalted() { return this.hashSalted; }

Deprecated
@Deprecated
public void setHashSalted(boolean hashSalted) { this.hashSalted = hashSalted; }
```

也没有设置盐值的地方，因为盐值可能是随机的，我们这里无法给出盐值，那么盐值怎么给呢？在 `HashedCredentialsMatcher` 类中我们可以发现一些线索，执行认证比对的方法中：

```
public boolean doCredentialsMatch(AuthenticationToken token, AuthenticationInfo info) {
    Object tokenHashedCredentials = this.hashProvidedCredentials(token, info);
    Object accountCredentials = this.getCredentials(info);
    return this.equals(tokenHashedCredentials, accountCredentials);
}

protected Object hashProvidedCredentials(AuthenticationToken token, AuthenticationInfo info) {
    Object salt = null;
    if (info instanceof SaltedAuthenticationInfo) {
        salt = ((SaltedAuthenticationInfo)info).getCredentialsSalt();
    } else if (this.isHashSalted()) {
        salt = this.getSalt(token);
    }
}
```

可以看到，先检测info是否属于 `SaltedAuthenticationInfo`，如果不是后续根据是否加盐在做处理，而如果是这个类的实例，则回去获取其盐值。

而在自定义Realm中有一个方法，其中返回的是 `SimpleAuthenticationInfo`，仅包含用户名和密码的简单的身份认证信息对象，而这个类本身实际是实现了 `SaltedAuthenticationInfo` 接口的，并且提供了有盐值相关的重载构造器：

```
public SimpleAuthenticationInfo(Object principal, Object hashedCredentials, ByteSource credentialsSalt, String realmName) {
    this.principals = new SimplePrincipalCollection(principal, realmName);
    this.credentials = hashedCredentials;
    this.credentialsSalt = credentialsSalt;
}
```

因此，只需要在自定义Realm中，在方法的返回值里加上盐值传递进去就可以了。

```
// 将用户信息封装在 AuthenticationInfo 中
return new SimpleAuthenticationInfo(user.getLoginname(), // 数据库中登录名
                                     user.getPassword(), // 数据库中的密码
                                     ByteSource.Util.bytes(user.getSalt()), // 数据库中
                                     存储的盐值
                                     this.getName()); // realm的标识
```

测试

(略)

Session管理

Shiro作为一个安全管理框架，对状态保持有很强的需要。例如最常用的用户认证，就必须状态的保持，以及其他的一些功能实现的需要。

- shiro需要：认证中的记住我中的用户名、正式登录的用户名
- 开发者需要：其他功能中需要存入session的值

Shiro的Session管理方案

1. shiro的session方案和任何容器都无关
2. JavaSE也可以使用，相关组件都是pojo对IOC极其友好（方便的管理对象和满足依赖、定制参数）
3. 可以方便的扩展定制存储位置（内存、缓存、数据库等）
4. 对web透明支持：用了shiro的session后，项目中关于session的代码完全不用任何改动
5. 提供了全面的session监听机制，和session检测机制，对session可以细粒度操作

所以使用了shiro后，采用shiro的session方案是最优的方案

JavaEE环境下Session的定制

一般情况下对Session没有特别的需求可以什么都不用改，但是如果说想要定制这个Session，例如修改默认的超时时间，或者想实现分布式的Session，那么就可以定制它。

```
<!--Session管理相关配置-->
<!--会话Cookie模板，默认可省-->
<bean id="sessionIdCookie" class="org.apache.shiro.web.servlet.SimpleCookie">
    <!--cookie的key="sid"-->
    <property name="name" value="JSESSIONID" />
    <!--只允许http请求访问cookie-->
    <property name="httpOnly" value="true" />
    <!--过期时间，-1表示一次会话有效-->
    <property name="maxAge" value="-1" />
</bean>
<bean id="sessionManager"
class="org.apache.shiro.web.session.mgt.DefaultWebSessionManager">
    <!--如果使用默认值，则可以生路sessionIdCookie-->
    <property name="sessionIdCookie" ref="sessionIdCookie" />
    <!--session全局超时时间，单位：毫秒，默认30分钟-->
    <property name="globalSessionTimeout" value="1800000" />
</bean>
```

在SecurityManager中注入Session管理器

```
<!--2.SecurityManager-->
<bean id="securityManager" class="org.apache.shiro.web.mgt.DefaultWebSecurityManager">
    <property name="realm" ref="myRealm" /> <!--注入Realm-->
    <property name="rememberMeManager" ref="rememberMeManager" /> <!--注入记住我管理器-->
    <property name="sessionManager" ref="sessionManager" /> <!--注入Session管理器-->
</bean>
```

Session监听

session有三个核心过程：创建、过期、停止

- 过期
 - session的默认过期时间为30分钟，通过比对最近一次使用时间和当前使用时间判断
 - session不会自动报告过期，需要检测器检测时，或者再次访问时，才可以识别是否过期并移除
- 停止
 - 用户主动调用logout，或主动调用session.stop()，这两种情况会将session标志为停止状态

1、定义监听器类，继承SessionListenerAdapter

```
public class MySessionListener extends SessionListenerAdapter {

    /**
     * session创建时触发
     */
    @Override
    public void onStart(Session session) {
        System.out.println("Session create...");
    }

    /**
     * session创建时触发 subject.logout() / session.stop()
     */
    @Override
    public void onStop(Session session) {
        System.out.println("Session stop...");
    }

    /**
     * session过期时触发：静默时间超过了过期时间
     */
    @Override
    public void onExpiration(Session session) {
        System.out.println("Session expired...");
    }
}
```

2、配置监听器类，关联给SessionManager

```
<bean id="sessionManager" class="org.apache.shiro.web.session.mgt.DefaultWebSessionManager">
    <!--如果使用默认值，则可以省略sessionIdCookie-->
    <property name="sessionIdCookie" ref="sessionIdCookie"/>
    <!--session全局超时时间，单位：毫秒，默认30分钟-->
    <property name="globalSessionTimeout" value="1800000"/>
    <!--注册session监听器-->
    <property name="sessionListeners">
        <list>
            <bean class="com.iflytek.shiro.session.MySessionListener" />
        </list>
    </property>
</bean>
```

session检测

用户如果没有主动退出登录，只是关闭浏览器，则session是否过期无法获知，也就不能停止session。

为此，shiro提供了session的检测机制，可以定时发起检测，识别出过期session并停止。

在Session管理器中配置检测处理

```
<bean id="sessionManager" class="org.apache.shiro.web.session.mgt.DefaultWebSessionManager">
    <!--如果使用默认值，则可以省略sessionIdCookie-->
    <property name="sessionIdCookie" ref="sessionIdCookie"/>
    <!--session全局超时时间，单位：毫秒，默认30分钟-->
    <property name="globalSessionTimeout" value="1800000"/>
    <!--注册session监听器-->
    <property name="sessionListeners">
        <list>
            <bean class="com.iflytek.shiro.session.MySessionListener" />
        </list>
    </property>
    <!--开启检测器，默认开启-->
    <property name="sessionValidationSchedulerEnabled" value="true" />
    <!--检测器运行时间间隔，单位：毫秒，默认1小时-->
    <property name="sessionValidationInterval" value="3600000" />
</bean>
```

注解开发

shiro提供了一系列的访问控制的注解，可以简化开发过程，注解加在Controller中，原理是会对Controller做增强，切入访问控制逻辑，所以需要spring-aspect依赖支持。

1、开启注解支持

```
<!--开启Shiro注解支持-->
<bean id="lifecycleBeanProcessor"
class="org.apache.shiro.spring.LifecycleBeanPostProcessor"/>

<!--自动代理生成器，和aop:config会冲突，所以选其一-->
<!--<bean
class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"
depends-on="lifecycleBeanProcessor"/>-->

<!--增强，该bean创建时会初始化额外一些功能和pointcut-->
<bean
class="org.apache.shiro.spring.security.interceptor.AuthorizationAttributeSourceAdvisor"
>
    <property name="securityManager" ref="securityManager" />
</bean>
```

2、使用注解

使用注解的时候，相应的ShiroFilter中的一些配置要删除。

加在类上

表示类中所有的方法都需要进行权限管理


```

@Controller
@RequestMapping("/manager")
@RequiresAuthentication // 类中的所有的方法都需要身份认证
@RequiresRoles(value = {"manager", "supermanager"}, logical = Logical.OR) // 类中的所有的方法都需要指定的角色，使用“或”
public class ManagerController {

```

加在方法上

只对该方法进行管理

```

@GetMapping("/index")
@RequiresGuest // 游客身份可以访问
public String getIndex() {

}

```

```

@GetMapping("/booklist")
@RequiresUser // 记住我 或者 已认证
@RequiresPermissions(value = {"manager:*", "manager:queryall"}, logical = Logical.OR)
// 需要指定的权限，指定逻辑为“或”，默认是且
public String getBookList() {

}

```

异常处理

```

public class MyExceptionHandler implements HandlerExceptionHandler{

    @Override
    public ModelAndView resolveException(HttpServletRequest request,
        HttpServletResponse response, Object handler,
        Exception ex) {
        System.out.println("--> " + ex.getClass());
        ex.printStackTrace();
        ModelAndView mv = new ModelAndView();
        if(ex instanceof IncorrectCredentialsException || ex instanceof
        UnknownAccountException) {
            // 跳转到登录页面，重新登录
            mv.setViewName("redirect:/user/login");
        } else if(ex instanceof UnauthorizedException){
            // 角色不足，也包括权限不足
            mv.setViewName("redirect:/user/perms/error");
        } else if(ex instanceof UnauthenticatedException){
            // 没有登录 没有合法身份
            mv.setViewName("redirect:/user/login");
        }
        return mv;
    }

}

```