# Springboot实战篇

## Idea构建Springboot项目

1、点击File-new-project-Spring Initializr 进入Springboot项目构建界面。

2、配置jdk和项目构造器，点击next

3、配置项目信息，主要修改组织名和项目名，点击next

4、选择需要加入的依赖，点击next

5、修改项目文件路径等信息

整理依赖信息如下：

|  |  |  |
| --- | --- | --- |
| 分类 | 具体依赖 | 作用 |
|  | DevTools | Springboot开发工具包，热部署需要使用 |
|  | Security | Spring Security，spring提供的安全服务 |
|  | Lomok | Java Annotation，annotation注解支持 |

## Springboot热部署

1、引入devtools依赖：在pom.xml配置文件中引入热部署所需依赖spring-boot-devtools。

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-devtools</artifactId>  
 <optional>true</optional>  
</dependency>

2、配置devtools：在application.yml文件中对devtools进行相关配置。

**spring:**  
 **devtools:  
 restart:** *#开启热部署功能* **enabled: true** *#设置需要重启的目录* **additional-paths:** src/main/java  
 *#设置classpath目录下不重启的文件夹* **exclude:** WEB-INF/\*\*

3、设置IDEA自动编译：

由于devtools是监听classpath下文件发生变化才会重启，但是idea默认不会自动编译，所以要设置idea自动编译。

首先在File-settings搜索Compiler，在Build project automatically前打勾，这时在项目运行时还是不会自动编译，所以进行下面的操作。

按Ctrl+Shift+Alt+/快捷键打开Registry，勾选compiler.automake.allow.when.app.runing选项。

## Controller

### 简单使用

1、新建Controller类：在新建的类上用@Controller注解，表示该类是一个Springboot Controller类。还可以用@RestController注解，但两者有比较大的区别，详见原理篇。

2、配置映射路径：用@RequestMapper注解配置映射路径，可以配置类、方法的映射路径。还可用@GetMapper注解，但两者有一定的区别，详见原理篇。

@Controller

@RequestMapping("/controller")  
public class SpringBootController {  
 @RequestMapping("/hello")  
 public String hello(){  
 System.*out*.println("进入hello demo!");  
 return "hello!!!!!!";  
 }

}

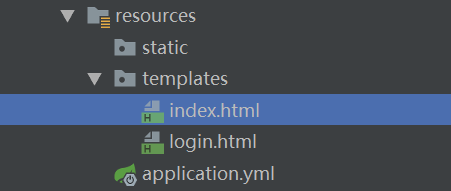
### 模板引擎配置

由于以上的简单使用会把return字符串写入到新页面中，而不能利用Controller跳转到视图层（如html、jsp）页面的，因此需要利用模板引擎thymeleaf来完成视图层的跳转。

1、引入thymeleaf依赖：在pom.xml配置文件中引入热部署所需依赖spring-boot-starter-thymeleaf。

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-thymeleaf</artifactId>  
</dependency>

2、在resource-templates文件夹下新建视图层文件，并且可以嵌套文件夹。



3、Controller中配置return有两种方式，两种方式配置的路径中第一个/代表templates文件夹，并且不需要加后缀（默认后缀为.html，因为Springboot不推荐使用jsp技术）。注意若用RestController注解则模板引擎配置无效。

@Controller  
public class SpringBootController {  
 @RequestMapping("/index")  
 public ModelAndView hello(){  
 System.*out*.println("进入index demo!");  
 return new ModelAndView("/index");  
 }  
 @RequestMapping("/login")  
 public String apple(){  
 System.*out*.println("进入login demo!");  
 return "/login";  
 }  
}

## Servlet、Filter、Listener

由于SpringBoot的核心控制器DispatcherServlet会处理所有的请求，所以要让请求到我们自己写的Servlet、Filter、Listener中就需要进行相应的注册，让DispatcherServlet知道这个Servlet、Filter、Listener的作用和请求路径映射。而注册的方法又分为两种，代码注册和注解注册。

### 代码注册

1、首先建立一个Servlet/Filter/Listener类。

2、然后在Tomcat启动类中（注解了@SpringBootApplication）下注册Servlet，其中@Bean表示该类依赖于SbServlet对象，由IOC容器自动注入；ServletRegistrationBean类是专为Servlet注册设计的类，在其构造方法中给出自定义Servlet的对象以及urlMapping。其它两种实现原理类似。

//Servlet

@Bean  
public ServletRegistrationBean SbServlet(){  
 return new ServletRegistrationBean(new SbServlet(), "/SbServlet/\*");  
}

//Filter

@Bean  
public FilterRegistrationBean SbFilter(){  
 FilterRegistrationBean f = new FilterRegistrationBean(new SbFilter());  
 f.addUrlPatterns("/\*");  
 return f;  
}

//Listener

@Bean  
public ServletListenerRegistrationBean SbListener(){  
 return new ServletListenerRegistrationBean(new SbListener());  
}

### 注解注册

1、首先建立一个Servlet/Filter/Listener类，并且添加@WebServlet/@WebFilter/@WebListener注解，并且在注解中给出urlPatterns也就是urlMapping。伪代码如下：

//Servlet

@WebServlet(name = "SbServlet",urlPatterns="/SbServlet/\*")  
public class SbServlet extends HttpServlet {  
}

//Filter

@WebFilter(filterName = "SbFilter",urlPatterns="/\*")  
public class SbFilter implements Filter {   
}

//Listener

@WebListener  
public class SbListener implements ServletContextListener,  
 HttpSessionListener, HttpSessionAttributeListener {  
}

2、在Tomcat启动类添加@ServletComponentScan注解，该注解意为让SpringBoot在启动时去扫描对应的包，找到添加了@WebServlet、@WebFilter、@WebListener注解过的类，伪代码如下：

@SpringBootApplication  
//该注解表示框架会去扫描相应的包  
@ServletComponentScan  
public class Springboot0719Application {  
 public static void main(String[] args) {  
 SpringApplication.*run*(Springboot0719Application.class, args);  
 }  
}

## 启动加载

1、**实现CommandLineRunner接口**：SpringBoot提供了启动时加载数据的功能，我们只需要实现CommandLineRunner接口就可以实现启动加载数据的功能。

2、**@Component注解**：为了让SpringBoot可以扫描到该类，用@Component注解，标明该类是SpringBoot组件。

3、**@Order注解**：用@Order注解可以控制多个启动类的加载顺序，其中的value值越小，加载的越早。

@Component //该注解泛指组件，在不确定组件类型时使用，SpringBoot启动时会扫描该注解修饰的类  
@Order(value = 2) //该标签设定该类的加载顺序，数字越小越先加载，若相同数字的，则文件默认排序加载  
public class StartLoader01 implements CommandLineRunner {  
 @Override  
 public void run(String... args) {  
 System.*out*.println("StartLoader01加载数据！");  
 }  
}

## 数据库连接

Springboot中支持的数据库连接方式主要有三大方式：jdbcTemplate连接、JPA连接和整合mybatis连接。

### JDBCTemplate

总体流程：

1、**引入jdbc支持：**引入spring-boot-starter-jdbc依赖，在pom文件中配置如下:

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-jdbc</artifactId>  
</dependency>

2、**配置数据库信息：**在application.properties文件中配置数据库驱动、连接、用户名、密码，如下：

spring.datasource.driver-class-name=com.mysql.jdbc.Driver  
spring.datasource.url=jdbc:mysql://localhost:3306/tb\_springboot?useSSL=false  
spring.datasource.username=root  
spring.datasource.password=root

3、**@Repository注解：在数据连接层的类上用@Repository注解，该注解表明这个类是一个数据连接层（DAO层）的类。**

**4、代码中利用JdbcTemplate类中的方法进行操作，具体情况在下面说明。**

#### 更新

##### update

jdbcTemplate.update()方法是用于数据库的普通更新操作，执行增加、删除、修改语句，其实现方式都和以下伪代码相似，伪代码如下：

public int addTest\_springboot(Test\_springboot bean) {  
 String sql = "insert into test\_springboot(id, name) values(?, ?)";  
 return jdbcTemplate.update(sql, bean.getId(), bean.getName());  
}

##### batchUpdate

jdbcTemplate.batchUpdate()方法是用于数据库的批量更新操作，利用list作为数据的存储结构，并且利用spring中的BatchPreparedStatementSetter回调方法进行sql语句参数的赋值，需要注意的是该方法的返回类型是int[]，即每一次执行的受影响行数，伪代码如下：

public int[] batchAddTest\_springboot(List<Test\_springboot> list) {  
 String sql = "insert into test\_springboot(name) values(?)";  
 int[] nums = jdbcTemplate.batchUpdate(sql, new BatchPreparedStatementSetter() {  
 @Override //该方法用于更新操作时的赋值操作，就是利用PreparedStatement进行?赋值  
 public void setValues(PreparedStatement pstmt, int i) throws SQLException {  
 Test\_springboot ts = list.get(i);  
 pstmt.setString(1, ts.getName());  
 }  
 @Override //该方法用于设定需要批量操作的数量，一般为list内的条目数  
 public int getBatchSize() {  
 return list.size();  
 }  
 });  
 return nums;  
}

#### 查询

##### query

jdbcTemplate.query()既可以进行对象查询还可以进行集合查询，这里用RowCallbackHandler回调类进行演示，还有几个回调类可以实现同样的功能，这里不做描述，伪代码如下：

public Test\_springboot getTest\_springbootById(int id) {  
 String sql = "select id, name from test\_springboot where id=?";

//若要查询集合，添加以下代码

//ArrayList<Test\_springboot> list = new ArrayList<Test\_springboot>();  
 Test\_springboot ss = new Test\_springboot();  
 jdbcTemplate.query(sql, new Object[]{id}, new RowCallbackHandler() {  
 @Override  
 public void processRow(ResultSet resultSet) throws SQLException {  
 ss.setId(resultSet.getInt("id"));  
 ss.setName(resultSet.getString("name"));

//若要查询集合，添加以下代码

//list.add(ss);  
 }  
 });  
 return ss;  
}

##### queryForObject

jdbcTemplate.queryForObject()方法可用于单值查询也可用于对象查询，对象查询主要调用了其中一个回调类RowMapper类进行查询，伪代码如下：

//单值查询

public int queryCouunt() {  
 String sql = "select *count*(\*) from test\_springboot";  
 return jdbcTemplate.queryForObject(sql, Integer.class);  
}

//对象查询

public Test\_springboot queryObject(int id) {  
 String sql = "select id, name from test\_springboot where id=?";  
 Test\_springboot ts = jdbcTemplate.queryForObject(sql, new Object[]{id}, new RowMapper<Test\_springboot>() {  
 @Override  
 public Test\_springboot mapRow(ResultSet resultSet, int i) throws SQLException {

//该方法内的i是结果集的当前行号，用来限制取出第几行的数据，默认全部取出  
 Test\_springboot t = new Test\_springboot();  
 t.setId(resultSet.getInt("id"));  
 t.setName(resultSet.getString("name"));  
 return t;  
 }  
 });  
 return ts;  
}

##### queryForMap

jdbcTemplate.queryForMap()方法用于查询对象，要求查询结果必须只有一行，返回一个以列名为key，以数据为value的map集合，伪代码如下：

public Map<String, Object> queryMap(int id) {  
 String sql = "select id, name from test\_springboot where id=?";  
 Map<String, Object> map = jdbcTemplate.queryForMap(sql, id);  
 return map;  
}

##### queryForList

jdbcTemplate.queryForList()方法用于查询集合，该方法会返回一个内含map对象的list集合，map中以列名为key，以数据为value，伪代码如下：

public List<Map<String, Object>> queryList() {  
 String sql = "select id, name from test\_springboot";  
 List<Map<String, Object>> list = jdbcTemplate.queryForList(sql);  
 return list;  
}

#### 调用函数

待后续整理。。。。。。。。

#### Excution

待后续整理。。。。。。。。

#### 回调类

待后续整理。。。。。。。。

### JPA

待后续整理。。。。。。。。

### 整合Mybatis

#### 注解式

1、**@MapperScan注解：**在启动类上使用@MapperScan注解，该注解的作用是让springboot启动的时候去检测mybatis的接口映射文件，伪代码如下：

@SpringBootApplication  
@MapperScan("com.dcits.zhangbfc.springboot0719.mapper") //该注解让springboot自动去检测mybatis的映射文件  
public class Springboot0719Application {  
 public static void main(String[] args) {  
 SpringApplication.*run*(Springboot0719Application.class, args);  
 }  
}

2、**注解式配置sql：**在接口文件上使用mybatis注解配置sql，以下是最简单的几种配置，伪代码如下：

public interface Test\_mybatisMapper {

//#{}代表方法参数中的变量

//若参数只有一个对象，则代表该对象的属性

//若参数还有其他参数，则表示直接的参数，不涉及里面的变量  
 @Insert("insert into test\_mybatis (mess) values(#{mess})")  
 int insert(Test\_mybatis tm);  
 @Delete("delete from test\_mybatis where id=#{id}")  
 int delete(int id);  
 @Update("update test\_mybatis set mess=#{mess} where id=#{id}")  
 int update (Test\_mybatis tm);  
 @Select("select id, mess from test\_mybatis")  
 List<Test\_mybatis> queryAll();  
 @Select("select id, mess from test\_mybatis where id=#{id}")  
 Test\_mybatis queryOne(int id);  
}

#### 配置式

待后续整理。。。。。。。。

## 事务控制

### 概述

1、事务控制注解@Transaction是方法级别的，可用于接口、接口方法、类、类方法上，若加在接口、类上则所有的接口、类方法都受事务控制。

2、@Transaction中参数的配置用“，”分隔。

3、一般在service层去进行事务控制，并且在controller层去进行异常处理，若在service就捕获处理了，对应的事务控制就捕获不了异常了。

4、只有外部方法的调用才会触发aop代理，才会触发事务管理；类内部方法互相调用是不会触发事务管理的。

5、Springboot事务控制默认捕获runtimeException才会回滚，若要求对其它异常进行回滚就要用rollbackFor、rollbackForClassName配置。

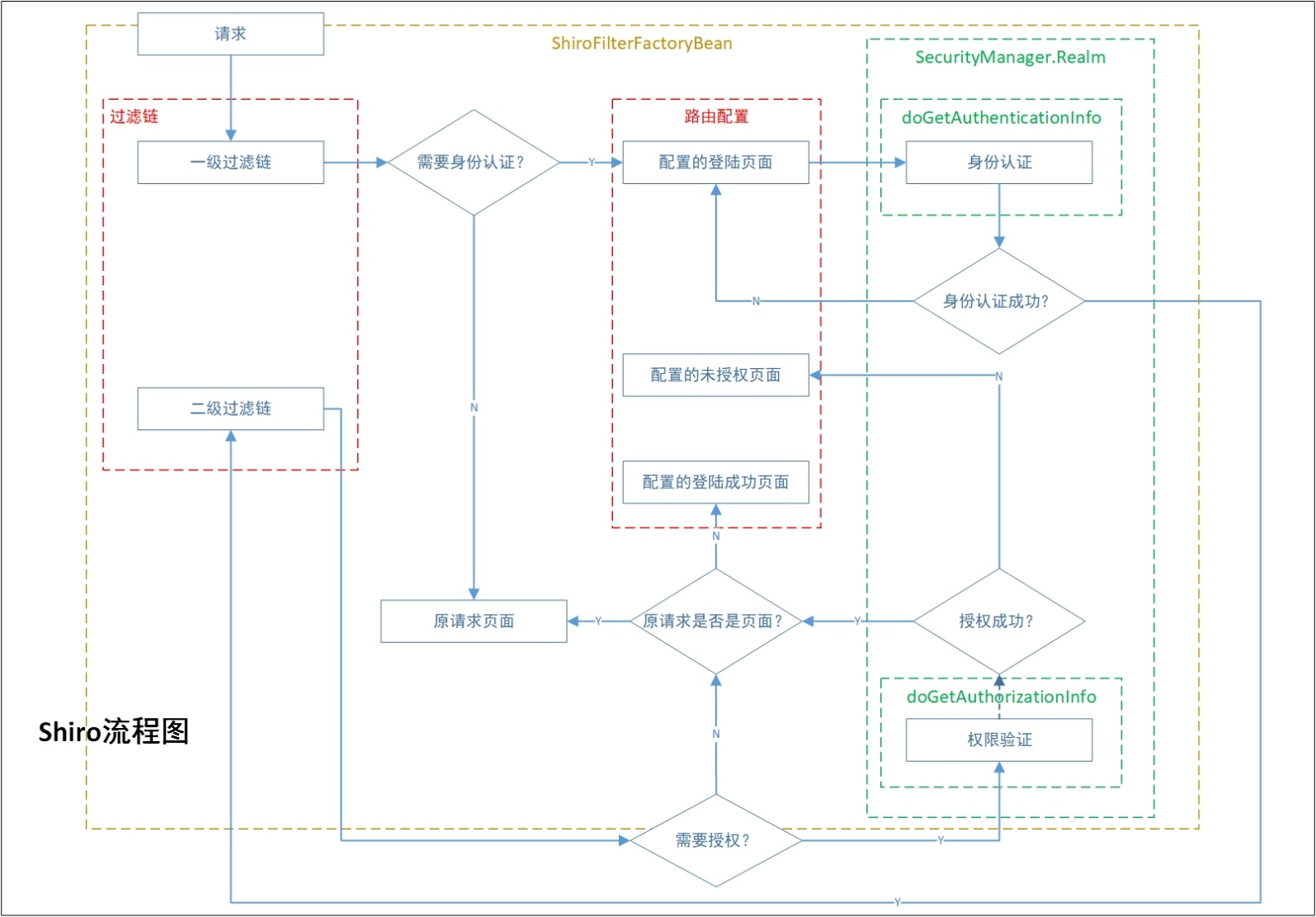
### 配置参数

|  |  |  |
| --- | --- | --- |
| 属性 | 类型 | 参数 |
| value | String | 可选的限定描述符，指定使用的事务管理器 |
| propagation | enum: Propagation | 可选的事务传播行为设置 |
| isolation | enum: Isolation | 可选的事务隔离级别设置 |
| readOnly | boolean | 读写或只读事务，默认读写 |
| timeout | int (in seconds granularity) | 事务超时时间设置 |
| rollbackFor | Class对象数组，必须继承自Throwable | 导致事务回滚的异常类数组 |
| rollbackForClassName | 类名数组，必须继承自Throwable | 导致事务回滚的异常类名字数组 |
| noRollbackFor | Class对象数组，必须继承自Throwable | 不会导致事务回滚的异常类数组 |
| noRollbackForClassName | 类名数组，必须继承自Throwable | 不会导致事务回滚的异常类名字数组 |

## Shiro权限管理

Shiro是Apache下的一个开源项目，我们称之为Apache Shiro。它是一个很易用与Java项目的的安全框架，提供了认证、授权、加密、会话管理，与 Spring Security 一样都是做一个权限的安全框架，但是与Spring Security 相比，在于 Shiro 使用了比较简单易懂易于使用的授权方式。

在实际应用中，我们所要做的就是配置好ShiroFilterFactoryBean，ShiroFilterFactoryBean是ShiroFilterFactory的具体实现类，而ShiroFilterFactory则是为Shiro的核心过滤器提供实例的接口。而ShiroFilterFactoryBeaa主要需要配置的分为三块：过滤链、路由配置、和SecurityManager的配置；过滤链主要控制那些路径需要身份认证/权限认证；路由配置主要配置默认登陆页面、未授权页面和登陆成功页面；而SecurityManager主要是配置AuthorizingRealm中的身份认证和权限认证，而以上两个一般只需要我们把身份和权限信息获取到并注入到AuthorizingRealm中即可，Shiro会帮我们做对比校验。大致原理图如下：



### 引入Shiro依赖

在pom.xml文件中引入shiro-spring依赖。

<dependency>  
 <groupId>org.apache.shiro</groupId>  
 <artifactId>shiro-spring</artifactId>  
 <version>1.2.3</version>  
</dependency>

### 配置ShiroFilterFactoryBean

在配置了多个Bean的情况下，Springboot依然是最后加载ShiroFilterFactoryBean，这是因为后续有很多的组件需要通过Springboot注入到ShiroFilterFactoryBean中，所以ShiroFilterFactoryBean必须最后加载。

#### 1、新建ShiroConfiguration类

ShiroFactory是通过ShiroFilterFactoryBean实现注入的，因此我们新建一个ShiroConfiguration类，并且配置一个ShiroFilterFactoryBean的Bean，让Sprinboot帮我们构建ShiroFilterFactoryBean的bean。其中的SecurityManager是后面需要配置的Bean，Springboot会帮自动注入。

@Configuration  
public class ShiroConfiguration {  
 @Bean

//该参数是后面要配置的SecurityManager的bean，同样是让Springboot帮我们注入  
 public ShiroFilterFactoryBean shiroFilter(SecurityManager securityManager){ }

}

#### 2、注入securityManager

在shiroFilter方法中注入Springboot帮我们构建的sercurityManager，这一步必须做，因为Filter要依赖其中的SercurityManager进行身份认证和权限认证，如果不注入sercurityManager，Filter将无效 。此处配置时可能会报错，因为并没有配置SercurityManager的Bean，可先忽略。

ShiroFilterFactoryBean shiroFilterFactoryBean = new ShiroFilterFactoryBean();  
shiroFilterFactoryBean.setSecurityManager(securityManager); //注入SecuritManager，必须注入

#### 3、配置过滤链

过滤链是通过map存储的，其中key为映射路径，value为 Shiro内置的FilterChain （可以理解为Shiro提供的过滤器），FilterChain信息如下表：

|  |  |
| --- | --- |
| Filter | 解释 |
| anon | 无参，开放权限，可以理解为匿名用户或游客 |
| authc | 无参，需要认证 |
| logout | 无参，注销，执行后会直接跳转到shiroFilterFactoryBean.setLoginUrl(); 设置的 url |
| authcBasic | 无参，表示 httpBasic 认证 |
| user | 无参，表示必须存在用户，当登入操作时不做检查 |
| ssl | 无参，表示安全的URL请求，协议为 https |
| perms[user] | 参数可写多个，表示需要某个或某些权限才能通过，多个参数时写 perms[“user, admin”]，当有多个参数时必须每个参数都通过才算通过 |
| roles[admin] | 参数可写多个，表示是某个或某些角色才能通过，多个参数时写 roles[“admin，user”]，当有多个参数时必须每个参数都通过才算通过 |
| rest[user] | 根据请求的方法，相当于 perms[user:method]，其中 method 为 post，get，delete 等 |
| port[8081] | 当请求的URL端口不是8081时，跳转到schemal://serverName:8081?queryString 其中 schmal 是协议 http 或 https 等等，serverName 是你访问的 Host，8081 是 Port 端口，queryString 是你访问的 URL 里的 ? 后面的参数 |

其中anon, authc, authcBasic, user是第一组过滤器（称一级过滤链），主要是配置请求是否需要进行身份认证的；而perms, port, rest, roles, ssl是第二组过滤器（称二级过滤链），主要配置请求路径是否进行权限认证，并且需要什么样的权限。而需要注意的是二级过滤链也可以用到Shiro的注解进行配置，但是需要开启Springboot aop对Shiro的支持，这次我们使用注解进行配置。因此过滤链配置如下：

Map<String, String> filterChainDefinitionMap = new LinkedHashMap<String, String>();  
//配置退出过滤器  
filterChainDefinitionMap.put("/logout", "logout");  
//所有路径都需要权限，该配置必须在最后，不然前面的配置会被覆盖  
filterChainDefinitionMap.put("/\*\*", "authc");  
shiroFilterFactoryBean.setFilterChainDefinitionMap(filterChainDefinitionMap);

#### 4、配置路由信息

配置路由信息只需要配置默认的登录页面、登录成功页面和未授权页面，并且也只有这三个路径可以配置。

//登陆页面，若不设置，会默认web工程根目录下的“/login.jsp”页面  
shiroFilterFactoryBean.setLoginUrl("/login");  
//登陆成功后要跳转的链接  
shiroFilterFactoryBean.setSuccessUrl("/index");  
//未授权界面  
shiroFilterFactoryBean.setUnauthorizedUrl("/403");

完成这步操作以后，还需要新建相应的Controller去和配置的路由信息进行配合使用，这里不详细介绍。

### 配置SecurityManager

SecurityManager主要功能是进行身份认证和权限认证，而这两部分的功能是交给AuthorizingRealm来实现的，因此除了配置securityManager的bean以外，还需要配置一个Realm的Bean。

#### 1、配置SecurityManager Bean

在ShiroConfiguration类新建一个SecurityManager的Bean，此时以上配置ShiroFilterFactoryBea

时注入的securityManager就是现在配置的Bean。其中还要将Ream Bean注入到securityManager中，但是由于还没有创建Realm，所以先不把Realm注入到security Manager中。

@Bean  
public SecurityManager securityManager(){  
 DefaultWebSecurityManager securityManager = new DefaultWebSecurityManager();  
 return securityManager;  
}

#### 2、配置AuthorizingRealm

AuthorizingRealm是用于做身份认证和权限认证的类，我们需要做的就是继承该类并重写其中的doGetAuthenticationInfo和doGetAuthorizationInfo方法，前者是身份认证用、后者是权限认证用的。

public class MyShiroRealm extends AuthorizingRealm {@Override  
 protected AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken authenticationToken) throws AuthenticationException {  
 }  
 @Override  
 protected AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principalCollection) {  
 }  
}

#### 3、身份认证

身份认证是通过doGetAuthenticationInfo方法实现的，该该方法内参数为Shiro提供的获取登录参数的参数。

1. 获取登录的用户名：通过参数authenticationToken获取用户名、密码（可不获取）。

String username = (String) authenticationToken.getPrincipal();

String password = (String) authenticationToken.getCredentials();//获取输入的密码，一般不用

2. 获取用户信息：通过数据库获取用户信息。

UserInfo userInfo = userInfoService.findByUsername(username);  
System.*out*.println("-------->>userInfo = "+userInfo);  
if (userInfo == null){//若获取不到该用户，则说明没有该用户，返回null  
 return null;  
}

3. 把用户登录信息提交给Shiro：Realm会自动比对加密后的密码是否一致等登录比对操作，若登录不成功会抛出异常（Shiro异常类），并把异常名写入到request域中，以shiroLoginFailure为key，异常名为value。

SimpleAuthenticationInfo authenticationInfo = new SimpleAuthenticationInfo(  
 userInfo, //用户对象，该对象用于传递到后面的权限验证使用  
 userInfo.getPassword(), //数据库中的密码  
 ByteSource.Util.*bytes*(userInfo.getCredentialsSalt()),//数据库中加密用的用户名+盐  
 getName() //Realm名，一般用本类的Realm名  
 );

return authenticationInfo;

4. 配置凭证匹配器：由于本例中用户凭证用了md5加密，因此需要配置一个凭证匹配器，告诉Shiro我们凭证是用的何种加密方式。在ShiroConfiguration类中加入HashedCredentialsMatcher的Bean。并且该凭证匹配器还要注入到Realm中，下一步会有演示。

@Bean  
public HashedCredentialsMatcher hashedCredentialsMatcher(){  
 HashedCredentialsMatcher hashedCredentialsMatcher = new HashedCredentialsMatcher();  
 hashedCredentialsMatcher.setHashAlgorithmName("md5");//散列算法：这里使用散列算法中的MD5算法  
 hashedCredentialsMatcher.setHashIterations(2);//散列次数，比如散列两次，相当于md5(md5(""))  
 return hashedCredentialsMatcher;  
}

5. 注入Realm：把我们自定义的Realm注入到SecurityManager中。

首先在ShiroConfiguration中配置Realm的Bean。

@Bean  
public MyShiroRealm myShiroRealm(){  
 MyShiroRealm myShiroRealm = new MyShiroRealm();  
 myShiroRealm.setCredentialsMatcher(hashedCredentialsMatcher());//注入凭证匹配器  
 return myShiroRealm;  
}

接着在SecurityManager中注入Realm。添加后SecurityManager如下：

@Bean  
public SecurityManager securityManager(){  
 DefaultWebSecurityManager securityManager = new DefaultWebSecurityManager();  
 securityManager.setRealm(myShiroRealm()); //注入Realm  
 return securityManager;  
}

#### 4、权限认证

配置doGetAuthorizationInfo：权限认证是在AuthorizingRealm类的doGetAuthorizationInfo方法中配置的。

1. 获取用户信息：在身份认证时已经在数据库提取了用户信息，现在只需要从Realm中获取用户信息就可以了。

UserInfo userInfo = (UserInfo) principalCollection.getPrimaryPrincipal();

2、提交用户权限信息到Realm：

SimpleAuthorizationInfo authorizationInfo = new SimpleAuthorizationInfo();  
for(SysRole role: userInfo.getRoleList()){  
 authorizationInfo.addRole(role.getRole());//提交用户角色信息列表  
 for (SysPermission p: role.getPermissions()){  
 authorizationInfo.addStringPermission(p.getPermission()); //提交用户权限信息列表  
 }  
}  
return authorizationInfo;

### 设置权限

除了二级过滤链中可以设置权限外，也可以利用Shiro标签进行方法级别的权限设置。

但首先要打开Springboot对Shiro的AOP支持。在ShiroConfiration中配置Bean。

@Bean  
public AuthorizationAttributeSourceAdvisor authorizationAttributeSourceAdvisor(SecurityManager securityManager){  
 AuthorizationAttributeSourceAdvisor authorizationAttributeSourceAdvisor = new AuthorizationAttributeSourceAdvisor();  
 authorizationAttributeSourceAdvisor.setSecurityManager(securityManager);  
 return authorizationAttributeSourceAdvisor;  
}

然后可以用@RequiresPermissions标签对每一个Controller方法进行权限控制。

@RequestMapping("/userInfoDel")  
@RequiresPermissions("userInfo:del") //shiro权限管理  
public String userInfoDel(){  
 return "userInfoDel";  
}

### 获取登录反馈

Shiro的登录反馈是通过异常类ShiroLoginFailure反馈的，Realm会把登录时抛出的异常的异常名放置到request域中并以shiroLoginFailure为key，可以通过获取request域中的异常名来判断登录失败是由于什么原因。

@PostMapping("/login")  
public String login(HttpServletRequest request, Map<String, Object> map){

String exception = (String)request.getAttribute("shiroLoginFailure");

String msg = "";  
 if (exception != null){  
 if (UnknownAccountException.class.getName().equals(exception)){  
 System.*out*.println("UnknuwnAccountException -->账号不存在：");  
 msg = "UnknuwnAccountException -->账号不存在：";  
 }

}  
 map.put("msg", msg);  
 return "/login";

}

登录认证过程中抛出的异常类为AuthenticationException，具体子类可查找API。这里列举几种比较常用的异常：

|  |  |
| --- | --- |
| 异常 | 意义 |
| UnknownAccountException | 账号不存在 |
| IncorrectCredentialsException | 密码不正确 |
| kaptchaValidateFailed | 验证码错误 |

# Springboot原理篇

## 注解

### 启动类

#### @SpringBootApplication

该注解@Configuration、@EnableAutoConfiguration、@ComponentScan三个注解的合体，是springboot启动类的注解，也可以用这三个注解代替。

#### @Configuration

该注解可以用java代码的形式实现spring中xml配置文件配置的效果。并且该注解和@Bean注解配合使用，来通过java配置bean。并且该注解中包含@ComPonent注解

#### @Component

该注解用于不属于MVC任何一层的spring组件类，标示该类是spring的组件，让springboot启动时扫描并注册该类。

#### @EnableAutoConfiguration

该注解是springboot的灵魂注解，因为有了这个注解让springboot有权利去进行很多的自动化配置，简化了我们的开发工作。

#### @ComponentScan

该注解让框架去扫描spring组件，并且可以指定扫描的范围，否则默认的范围就是该注解修饰的类的同一包内。

### 控制层

#### @Controller

该注解表示该类是一个控制层的类，并且该注解可以配合视图解析器完成页面的跳转。

#### @RestController

该注解是@Controller和@ResponseBody的组合，该注解标注的类不能实现页面的跳转，配置的视图解析器不起作用，返回的内容就是return里的内容，一般当返回json、xml或自定义mediaType内容时才会使用该注解。

#### @ResponseBody

该注解可以让return的内容直接写入到http响应正文(ResponseBody)中，一般用于异步请求中响应请求，并且该注解可以标注在类、方法上。

#### @RequestBody

该注解用在方法参数前，用于将http请求正文中的数据写入到参数中的对象，并且在方法响应时，同样是不解析路径，而是将返回值写入http响应正文中。

#### @RequestMapping

该注解是配置controller中映射路径的，可注解类和方法，请求方法有get、post、put、delete，各个请求方式的代替注解为@GetMapping、@PostMapping、@PutMapping、@DeleteMapping。

#### @PathVariable

该注解用于方法参数前，用于提取特定的请求路径组合，如：/say/{id}；那么就可以将/say/后面的组合路径写入到方法参数中。

#### @RequestParam

该注解用于方法参数前，用于提取请求路径中的参数，如：/say?id=222；那么就可以将参数id的值写入到方法参数中。

### 业务层

#### @Service

该注解用于标注业务层组件，其中name属性用于定义该组件对应bean的id，若通过Resource指定name属性自动注入，那么这个name就必须和service中定义的name一致。否则当上下文找不到对应name的id的时候就需要通过属性的类型去上下文装配。

### 持久层

#### @Repository

该注解用于将数据访问层DAO层的类标识为Spring Bean，并且该注解只能标注在DAO层的类上。因为该注解还有一个封装DAO层的数据访问异常信息的功能。

### 其它

#### @Autowired

该注解是Spring提供的用于依赖注入的注解，默认按照byType方式（在上下文中查找类型相等的bean）装配，若要按照byName方式（在上下文中查找id相等的bean）装配，要与@Qualifier配合使用。

#### @Resource

该注解是J2EE提供的用于依赖注入的注解，默认按照byName方式装配，若byName方式没有找到bean则在按照byType方式装配。并且该注解的name属性和type属性对应了byName、byType装配方式。