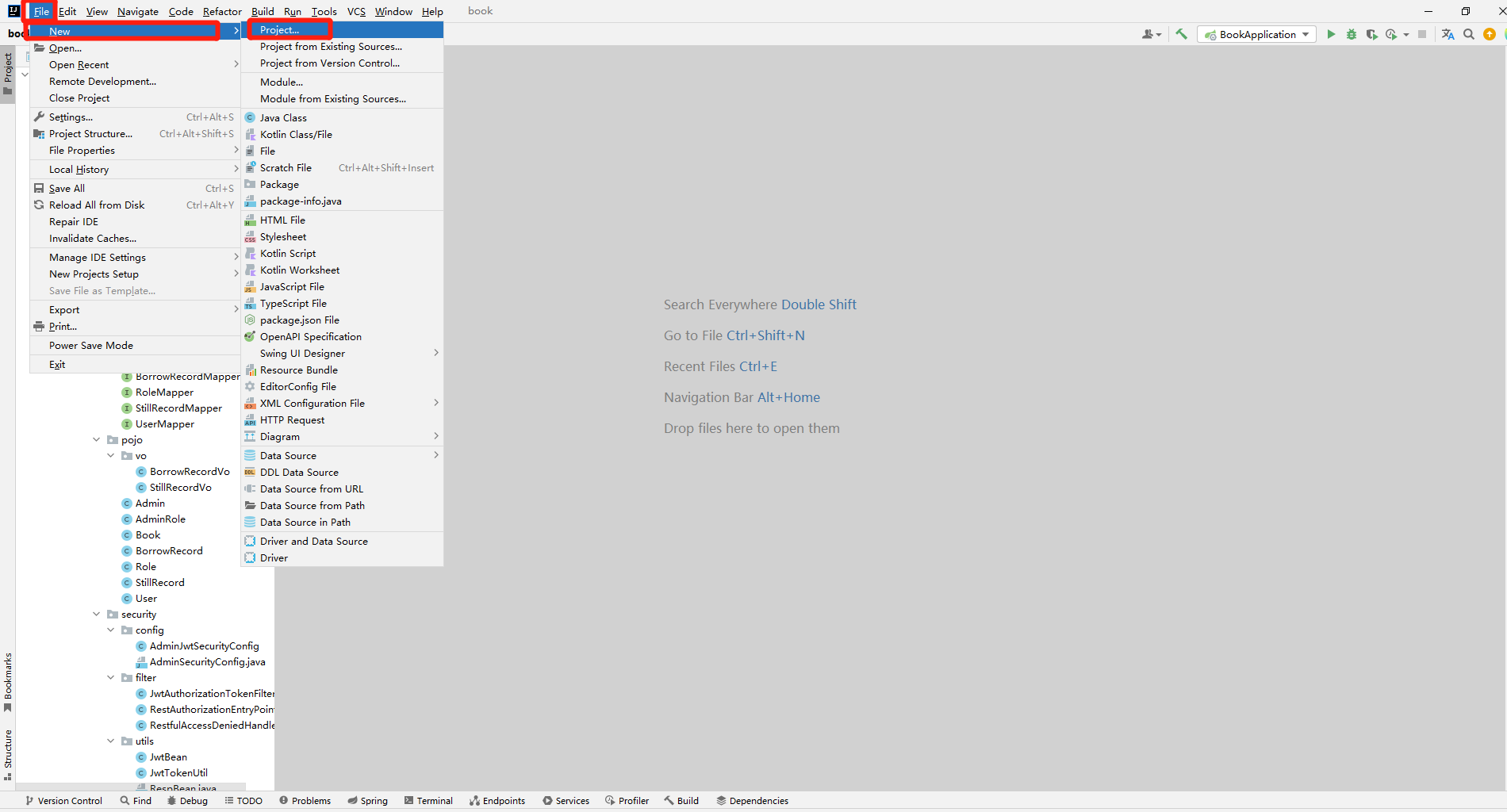
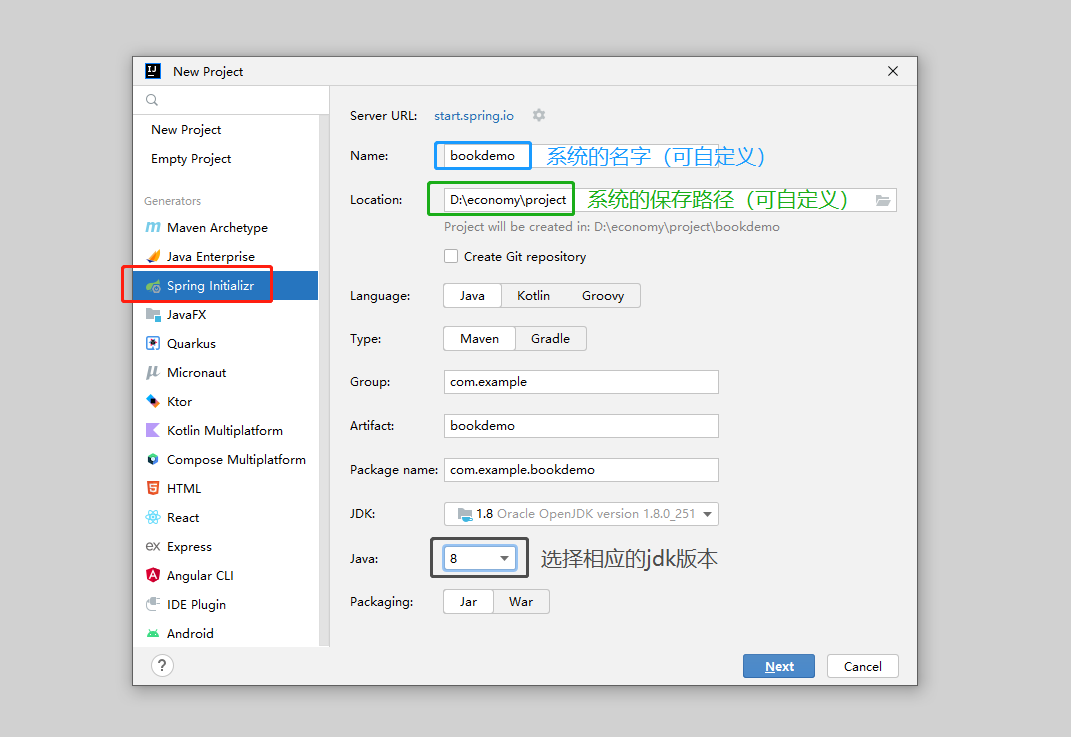
# Spring Boot创建项目

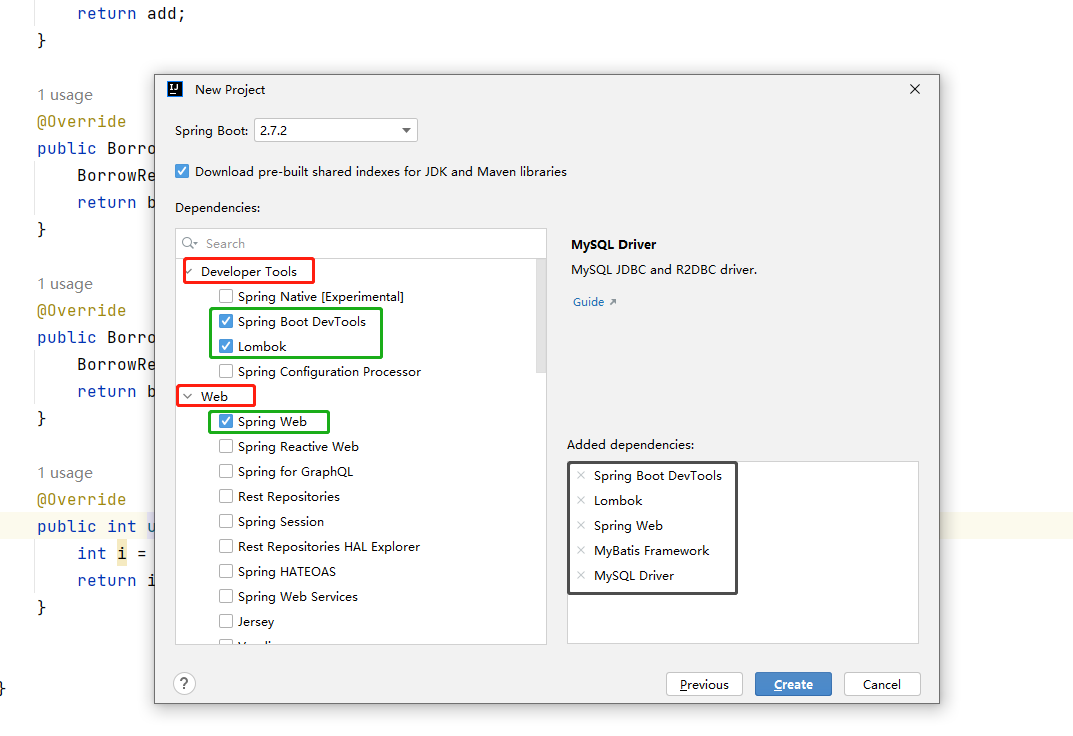
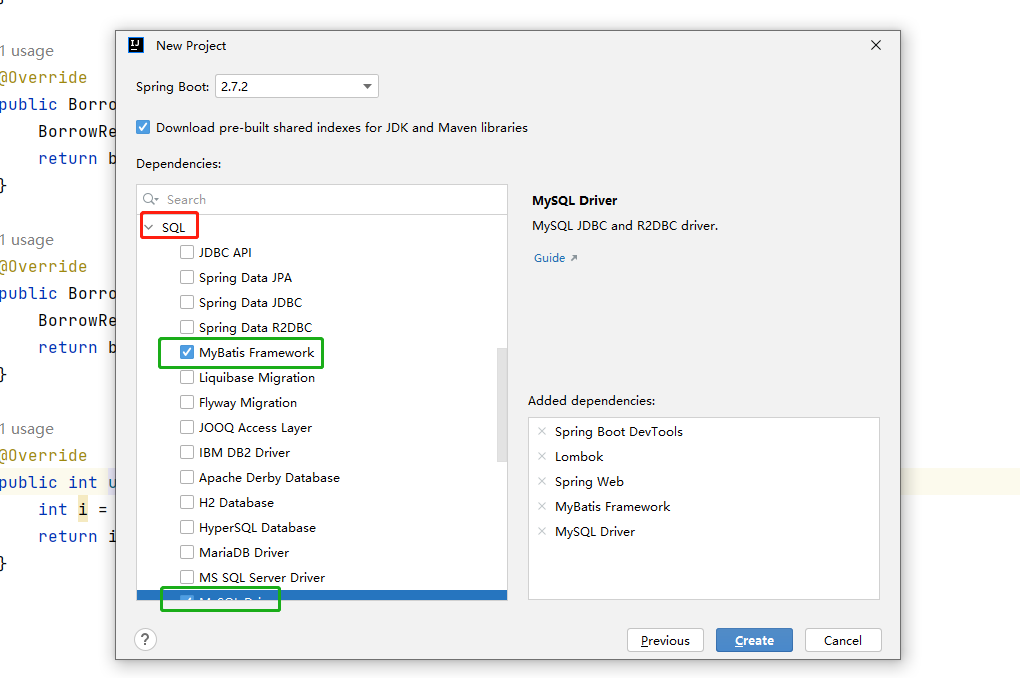
1. 点击file--->new--->project



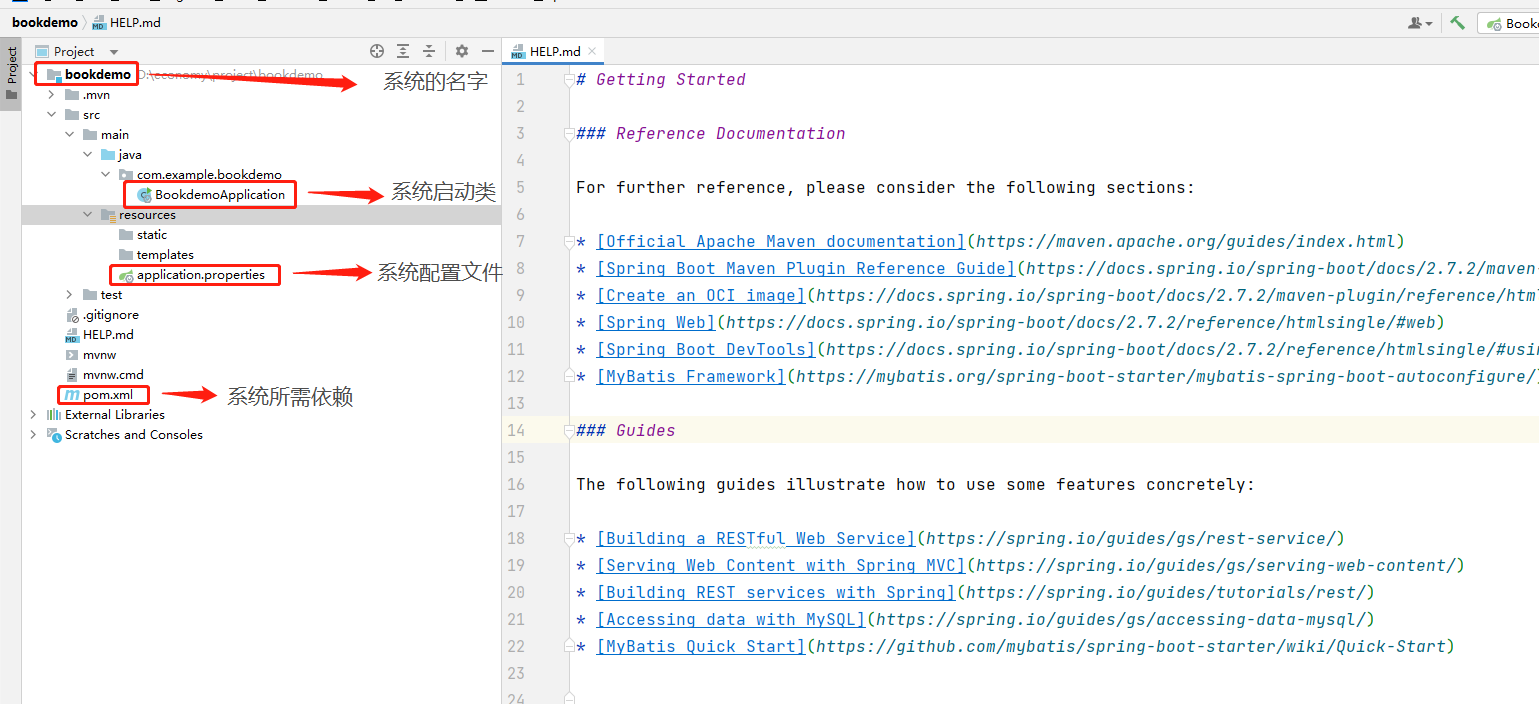
1. 选择Spring Initializr--->Name--->Location--->Java



1. 选择SpringBoot依赖：（一般情况下的五种依赖）
2. Developer Tools：Spring boot DevTools（作用：Spring Boot热部署）/Lombok（作用：一般用于实体类中的get/set方法注解）
3. Web：Spring Web（可选项，需要编写前端则添加，反之不需要添加）
4. SQL：Mybatis Framework（添加MyBatis的应用）/MySQL Driver（连接数据库驱动）

1. 创建Spring Boot项目成功



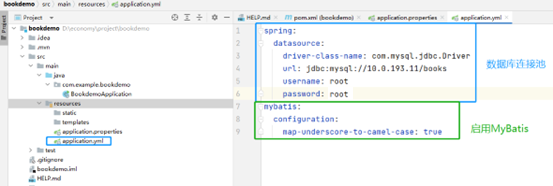
# MyBatis的使用

1. 添加MyBatis的依赖：pom--->MyBatis的依赖--->更新



<dependency>  
 <groupId>org.mybatis</groupId>  
 <artifactId>mybatis</artifactId>  
 <version>3.5.6</version>  
</dependency>

1. 创建application.yml编写配置（Application.yml可以看作K-V形式）
2. 数据库连接池的配置
3. MyBatis进行启动



spring:  
 datasource:  
 driver-class-name: com.mysql.jdbc.Driver  
 url: jdbc:mysql://10.0.193.11/books  
 username: root  
 password: root  
mybatis:  
 configuration:  
 map-underscore-to-camel-case: true

注意：10.0.193.11为数据库的远程ip地址，视情况修改

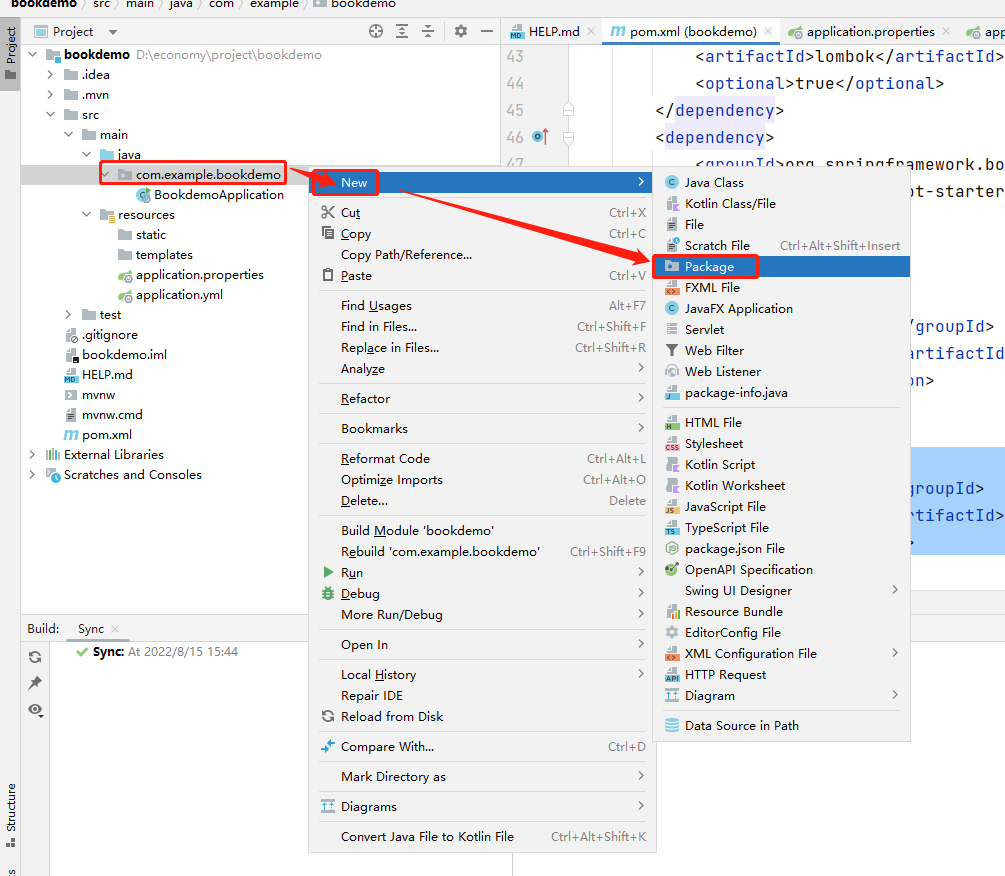
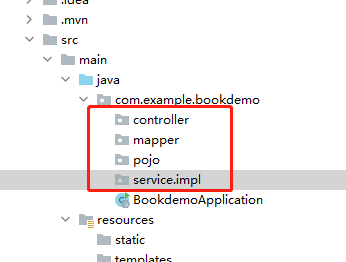
books为数据库中所要使用的数据库名称，视情况修改

root为数据库的用户名

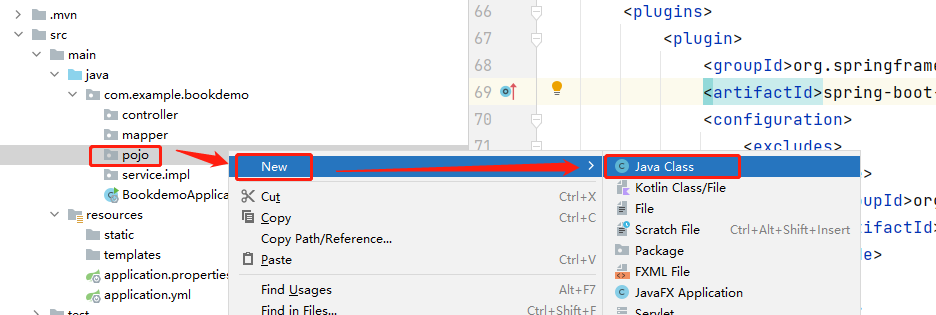
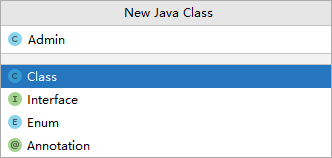
password为数据库访密码

# 系统编写

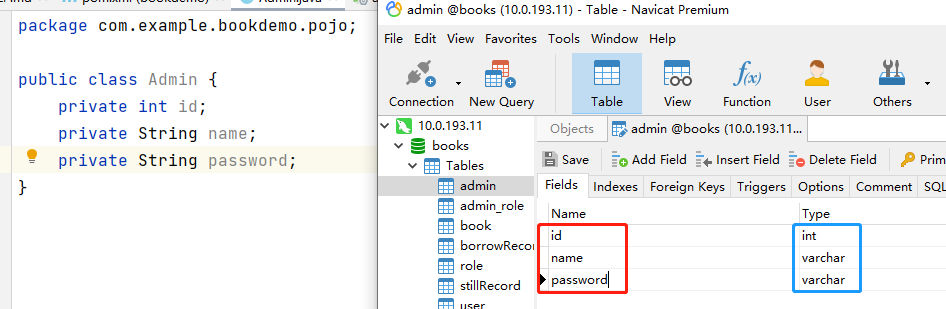
1. 创建系统包：com.example.bookdemo--->New--->Package（此处需要先创建5个包）
2. Pojo（entity）包：作用于映射数据库的持久对象
3. mapper（dao）包：作用于数据访问（就是编写SQL语句）
4. service包：作用于业务逻辑接口
5. impl包（在service层下级）：作用于实现service中的业务逻辑接口
6. controller包：作用于业务流程模块控制

1. 在pojo层创建实体对象：pojo--->New--->Java Class（名字遵守驼峰命名法，一般情况下与数据库表名相同）

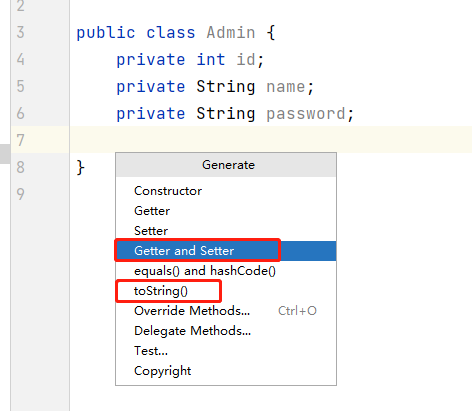
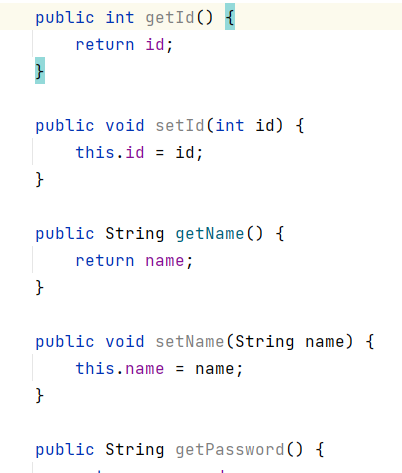
 

1. 编写实体对象：一般情况，定义的属性应与数据库的字段名相同，字段类型与定义的数据类型相同



1. 编写实体对象属性的方法（IDEA快捷键：ALT+INSERT）：get/set方法和toString方法

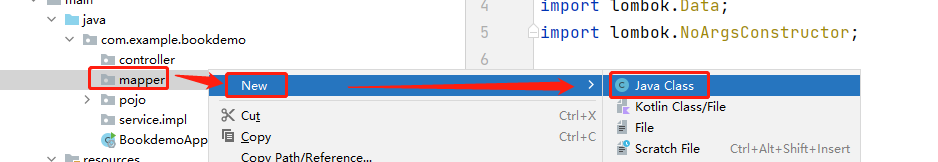
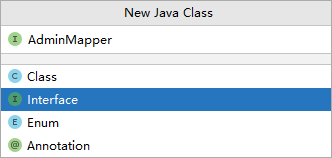
方法1：点击快捷键后，分别进行get/set方法和toString方法

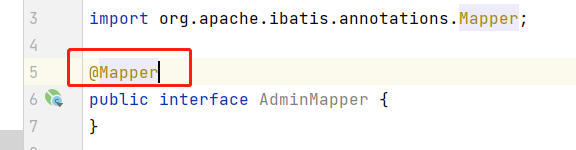
方法2：利用注解



1. 在mapper层创建数据访问接口：mapper--->New--->Java Class（名字遵守驼峰命名法，一般情况下为pojo所创建的实体名字+Mapper）

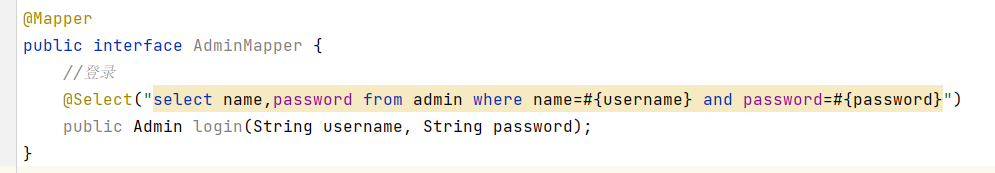
 

1. 编写数据接口
2. 在接口上方添加注解@Mapper（作用：利用注解的方式让Spring Boot进行统一管理；缺点：每一个mapper接口都需要进行添加注解）；也可以在启动类上放编写@MapperScan(basePackages = {"com.example.bookdemo.mapper"})（优点：只需要进行这一次的编写配置就可自动扫描mapper包中的所有方法）

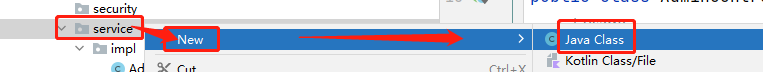
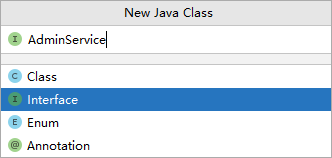


1. 编写方法（以简单的登录为例）

@Select("select name,password from admin where name=#{username} and password=#{password}")  
public Admin login(String username,String password);

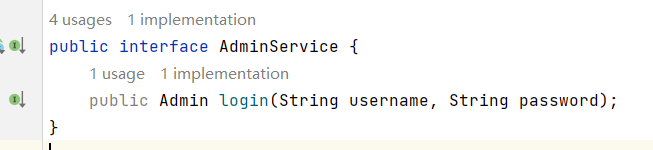


1. 在service层创建数据访问接口：service--->New--->Java Class（名字遵守驼峰命名法，一般情况下为pojo所创建的实体名字+Servicer）

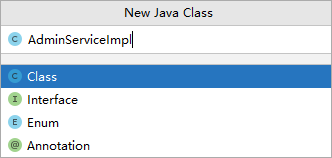
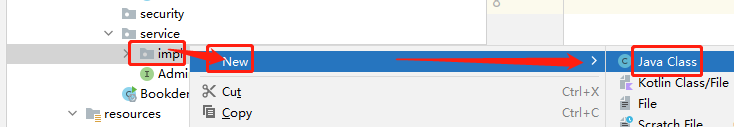
 

1. 编写业务逻辑接口：
2. 编写代码

public Admin login(String username, String password);



1. 在impl层创建数据访问接口的实现：service--->New--->Java Class（名字遵守驼峰命名法，一般情况下为pojo所创建的实体名字+ServicerImpl）

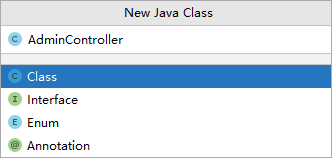
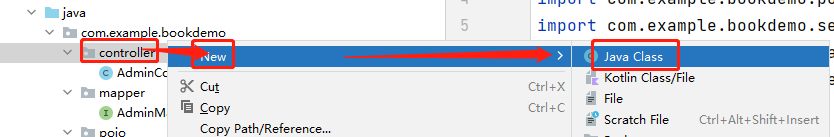


1. 编写业务逻辑实现类
2. 在接口上方添加注解@Service（作用：利用注解的方式让Spring Boot进行统一管理；缺点：每一个service接口都需要进行添加注解）；实现相应的业务逻辑接口；利用@Autowired注解将mapper层数据注入进来
3. 编写代码：

@Autowired  
AdminMapper adminMapper;  
@Override  
public Admin login(String username, String password) {  
 Admin login = adminMapper.login(username, password);  
 return login;  
}

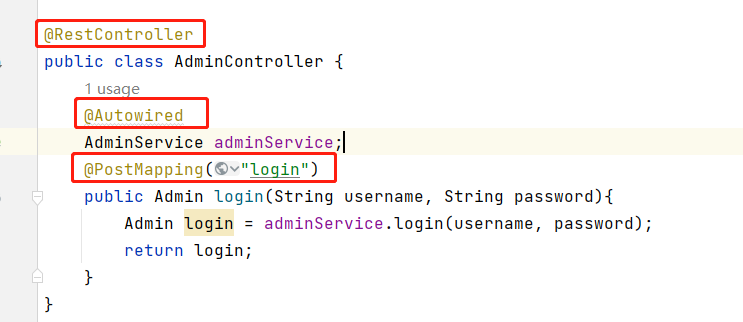


1. 在controller层创建控制层类：controller--->New--->Java Class（名字遵守驼峰命名法，一般情况下为pojo所创建的实体名字+Controller）



1. 编写控制层代码：
2. 在接口上方添加注解@RestController（作用：利用注解的方式让Spring Boot进行统一管理；缺点：每一个controller都需要进行添加注解）；利用@Autowired注解将service层数据注入进来；通过@PostMapping进行路径的跳转
3. 编写代码：

@Autowired  
AdminService adminService;  
@PostMapping("login")  
public Admin login(String username, String password){  
 Admin login = adminService.login(username, password);  
 return login;  
}



@RestController和@Controller的区别：

1. @RestController=@ResponseBody ＋ @Controller；
2. @RestController无法返回指定页面，而@Controller可以；前者可以直接返回数据，后者需要@ResponseBody辅助；
3. @RestController：跳转界面，直接返回数据；@Controller：跳转jsp页面

@ResponseBody：

返回JSON数据或者是XML，不会走视图解析器

@RequestBody：

接收前端传递给后端的JSON字符串中的数据的，前端请求必须为POST类型，@RequestBody与@RequestParam()可以同时使用，@RequestBody最多只能有一个，而@RequestParam()可以有多个。

# Spring Sercurity+JWT的使用

1. 引入Spring Security和JWT的依赖

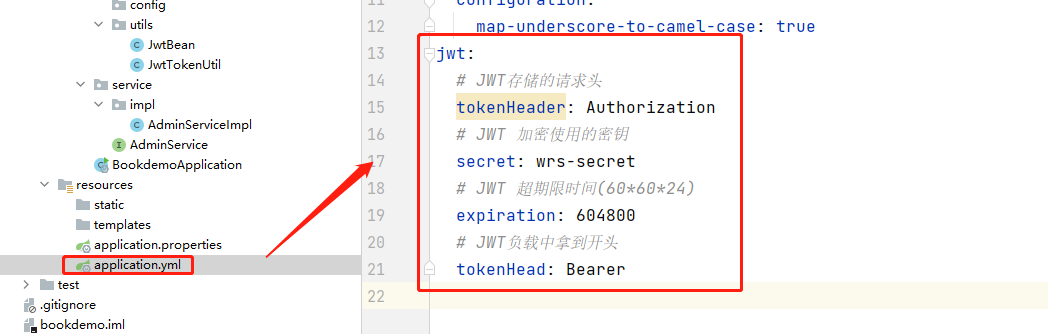


<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-security</artifactId>  
</dependency>

<dependency>  
 <groupId>io.jsonwebtoken</groupId>  
 <artifactId>jjwt</artifactId>  
 <version>0.9.1</version>  
</dependency>

1. 在application.yml中编写jwt的配置文件

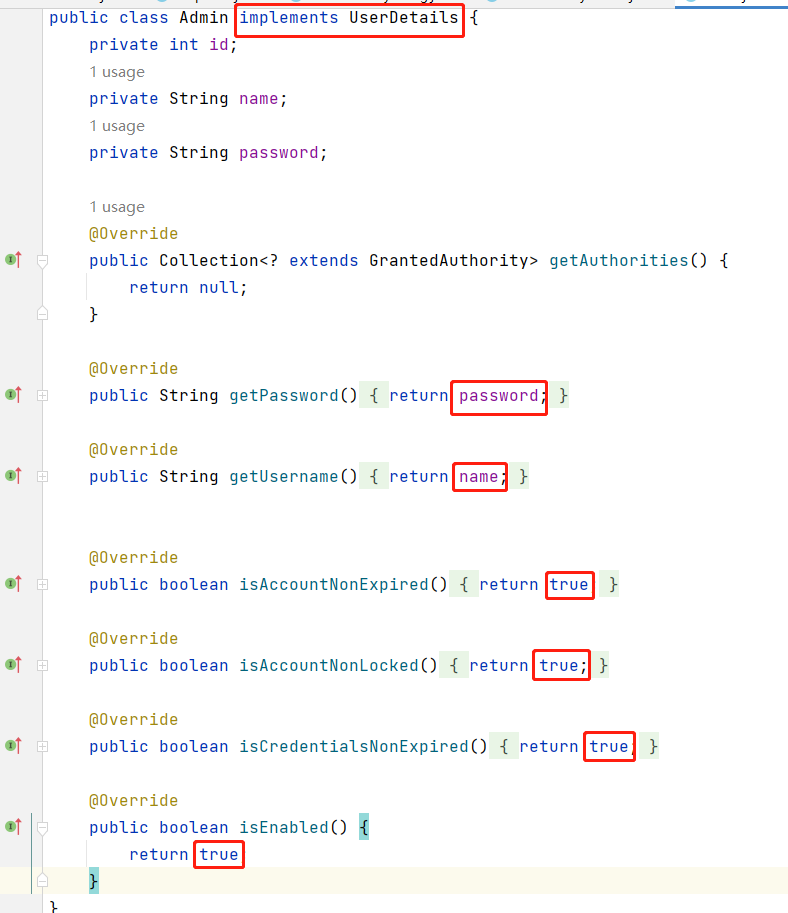
jwt:  
 tokenHeader: Authorization  
 secret: wrs-secret  
 expiration: 604800  
 tokenHead: Bearer



1. 创建返回信息JwtBean类（用来返回值规定）----->可写可不写



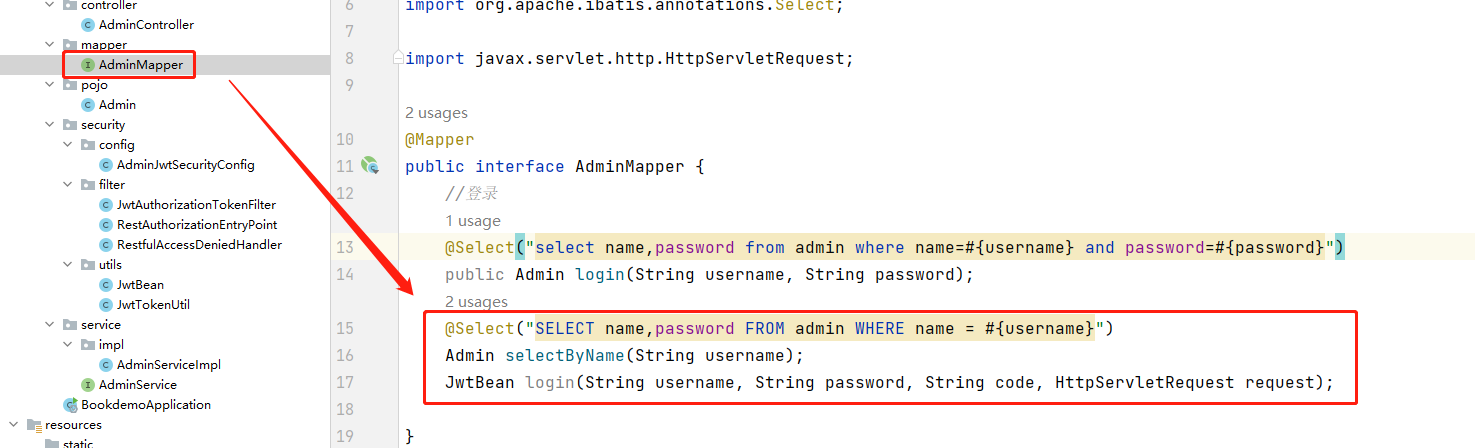
1. pojo层中的Admin实体类实现UserDetails，将实现的方法中return null改成相应的值以及所有的false改成true



1. 在mapper层中添加登录数据访问接口

JwtBean login(String username, String password, String code, HttpServletRequest request);

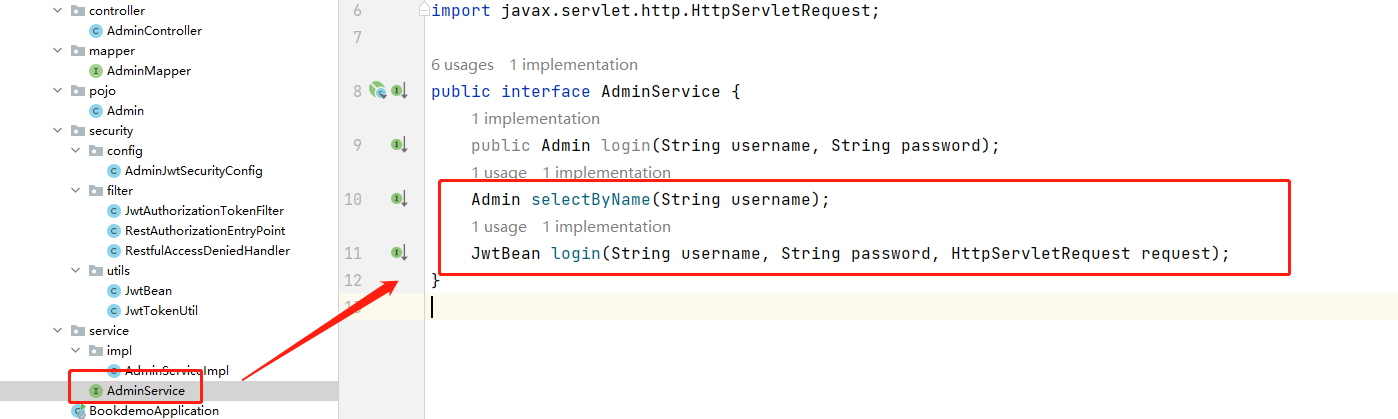
@Select("SELECT name,password FROM admin WHERE name = #{username}")  
Admin selectByName(String username);



1. 在service层中添加业务逻辑接口

JwtBean login(String username, String password, HttpServletRequest request);

Admin selectByName(String username);



1. 在utils工具包中创建工具类JwtTokenUtil。要求：
2. 编写Token的生成
3. 编写通过Token获取username
4. Token有效期
5. 最后用@Component注解让Spring Boot进行统一管理



1. 在service下级层impl创建实现类。要求：
2. 注入mapper接口、UserDetailsServivce（Security提供）、PasswordEncoder（Security提供）、JwtTokenUtil（utils层自己编写）
3. 将配置文件中的tokenHead参数引入
4. 利用BCryptPasswordEncoder将获取的数据库原密码进行加密，否则会一直提示：用户名或密码错误
5. 最后用@Service注解让Spring Boot进行统一管理



1. 在filter层编写JWT的过滤器。要求：
2. 继承OncePerRequestFilter
3. 将配置文件中的tokenHeader和tokenHead参数引入
4. 注入JwtTokenUtil（utils层自己编写）、UserDetailsServivce（Security提供）



1. 在filter层编写JWT的过滤器中未登录的类。要求：
2. 实现AuthenticationEntryPoint
3. 用@Component注解让Spring Boot进行统一管理



1. 在filter层编写JWT的过滤器中权限不足的类。要求：

（1）实现AuthenticationEntryPoint

（2）用@Component注解让Spring Boot进行统一管理



1. 在config层中security配置类，要求：
2. 继承WebSecurityConfigurerAdapter
3. 注入service层逻辑接口、未登录类（filter层中自主编写）、权限不足类（filter层中自主编写）
4. 放行路径为登录路径
5. 最后用@Configuration让Spring Boot进行统一管理

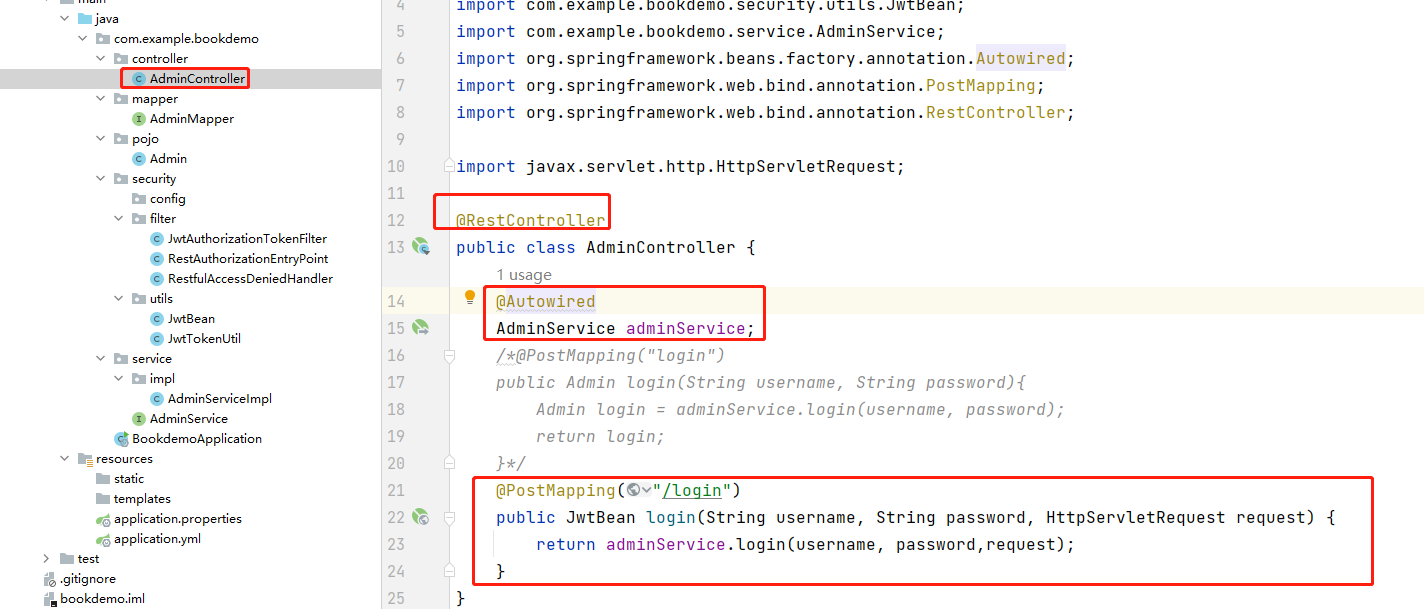


1. 编写controller层中的登录方法，要求：

（1）注入service层逻辑接口

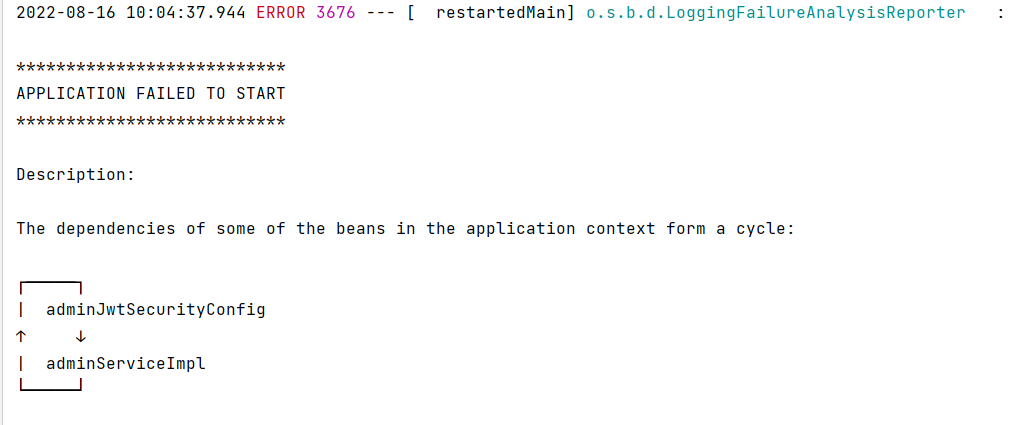
（2）使用@PostMapping进行跳转

（3）最后用@RestController让Spring Boot进行统一管理



1. 运行测试：

注意发生下面错误，在application.properties配置文件中加入：spring.main.allow-circular-references=true



使用Postman进行测试：



# TestNG+log4j的使用

1. 添加依赖TestNG+log4j的依赖



<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter</artifactId>  
 <exclusions>  
 <exclusion>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-logging</artifactId>  
 </exclusion>  
 </exclusions>  
</dependency>

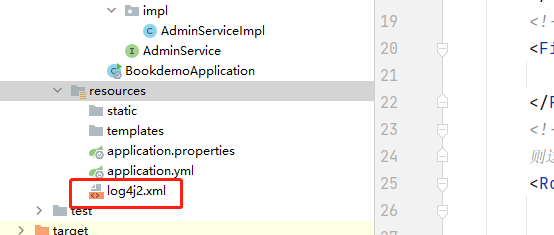
<log4j.version>2.18.0</log4j.version>

<dependency>  
 <groupId>org.testng</groupId>  
 <artifactId>testng</artifactId>  
 <version>7.5</version>  
</dependency>

<!--log4j-->  
<dependency>  
 <groupId>org.apache.logging.log4j</groupId>  
 <artifactId>log4j-core</artifactId>  
 <version>${log4j.version}</version>  
</dependency>  
<dependency>  
 <groupId>org.apache.logging.log4j</groupId>  
 <artifactId>log4j-api</artifactId>  
 <version>${log4j.version}</version>  
</dependency>  
<dependency>  
 <groupId>org.apache.logging.log4j</groupId>  
 <artifactId>log4j-slf4j-impl</artifactId>  
 <version>${log4j.version}</version>  
</dependency>

1. 将log4j.xml文件导入resource文件夹中

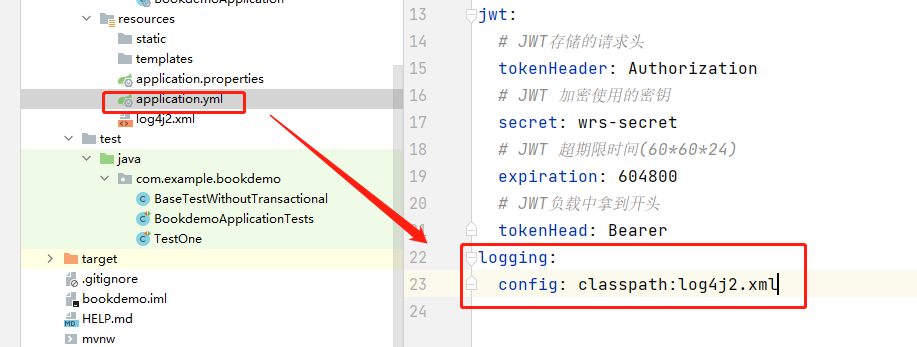




1. 添加配置

在application.yml中添加：

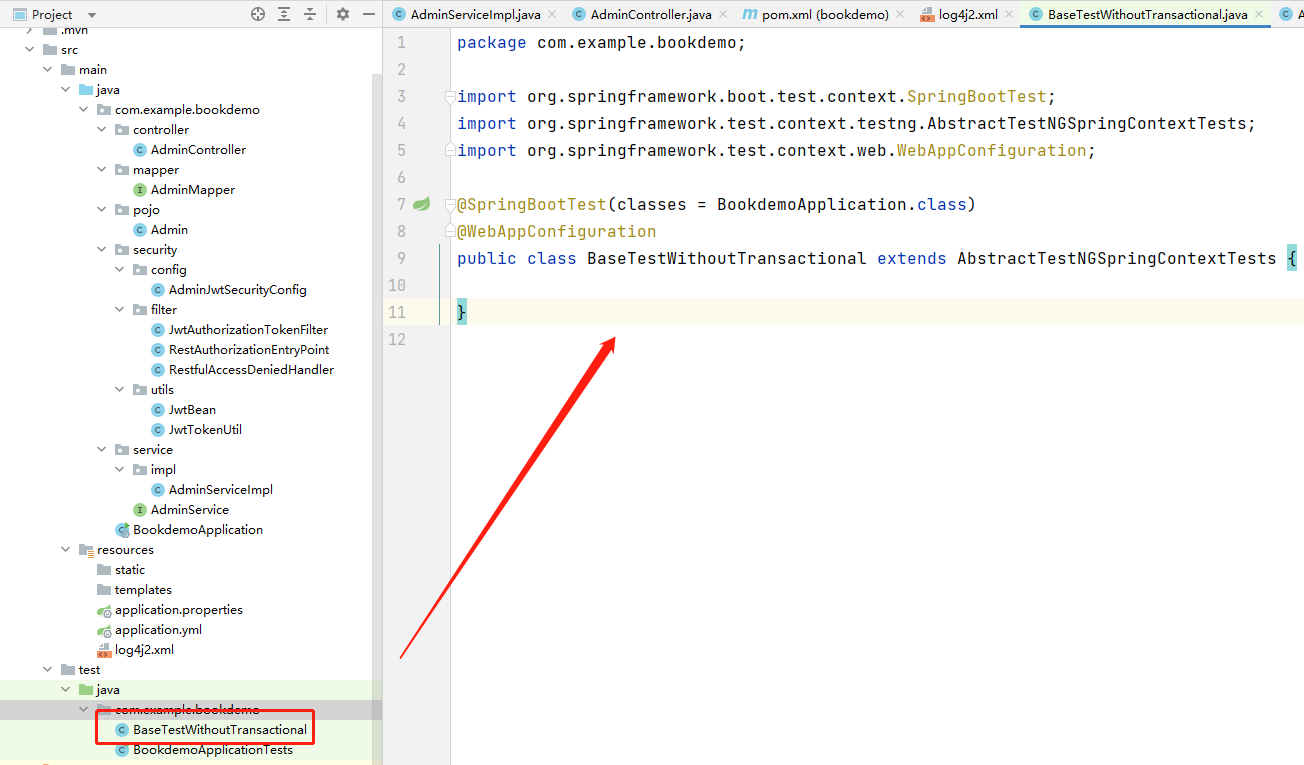
logging:  
 config: classpath:log4j2.xml



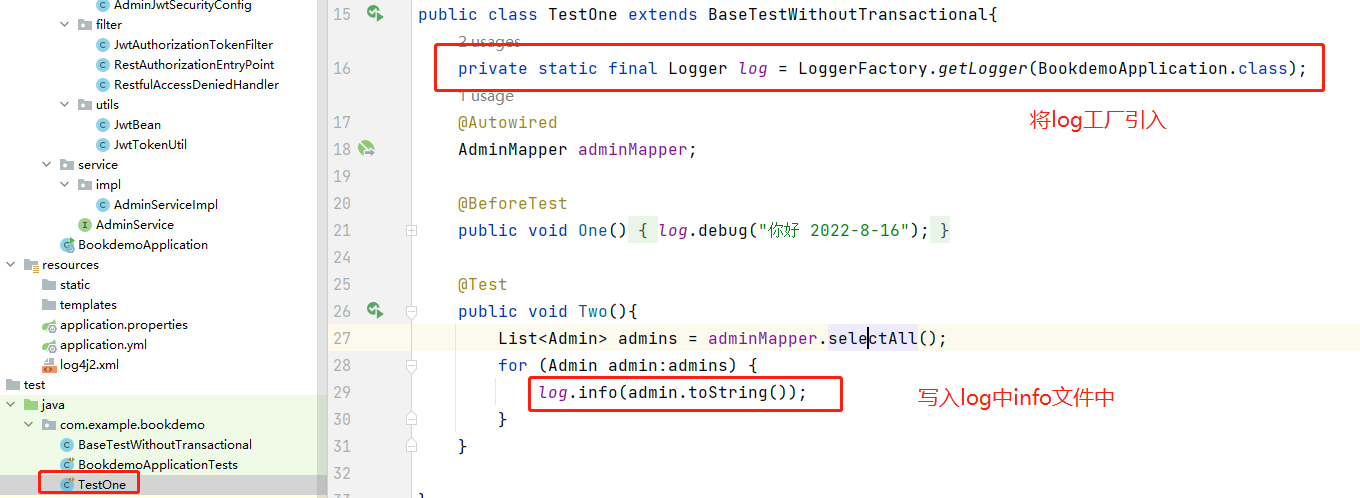
1. 修改log4j.xml文件



1. 在测试文件中编写BaseTestWithoutTransactional并继承AbstractTestNGSpringContextTests



1. 编写测试类，继承BaseTestWithoutTransactional



1. 运行测试：在修改的路径下寻找日志文件

