# SpringBoot入门

1. **SpringBoot简介**

简化Spring应用开发，方便Spring的配置，创建产品级应用。是SpringMC的升级。

简单的来说就是：SpringBoot->J2EE一站式解决方案；

SpringCloud->分布式整体解决方案；

**优点：**

* 嵌入Servlet容器，应用无需打包成WAR包
* starters自动依赖、版本控制
* 无需配置XML，无代码生成
* 实时监控环境
* 集成云计算

1. **微服务**

* 一种架构风格；
* 每个应用都是一小组微服务，应用之间的通信可以通过HTTP通信；
* 每个功能单元都需要独立，可以单独组合、升级；



日志

1. **常见日志框架：**



SpringBoot中使用的日志是：SLF4j + LogBack

1. **SpringBoot日志系统的使用**：

* **pom.xml文件中**，添加依赖：lombok，省却创建Logger对象

<dependency>  
 <groupId>org.projectlombok</groupId>  
 <artifactId>**lombok**</artifactId>  
</dependency>

* **创建配置文件：**logback-spring.xml，用于配置log日志

**<!--正常使用的log日志配置方式-->**  
<configuration>  
  **<!--设置日志基本的格式-->**  
 <appender name="consoleLog" class="ch.qos.logback.core.ConsoleAppender">  
 <layout class="ch.qos.logback.classic.PatternLayout">  
 <pattern>  
 %d - %msg%n  
 </pattern>  
 </layout>  
 </appender>  
  
 **<!--配置: 记录Info级别的日志-->**  
 <appender name="fileInfoLog" class="ch.qos.logback.core.rolling.RollingFileAppender">  
 <filter class="ch.qos.logback.classic.filter.LevelFilter">  
 <level>ERROR</level>  
  **<!--过滤更高级的信息：ERROR、WARN-->** <onMatch>DENY</onMatch>  
 <onMismatch>ACCEPT</onMismatch>  
 </filter>  
 <encoder>  
 <pattern>%msg%n</pattern>  
 </encoder>

**<!--滚动策略：日志文件存放位置-->**  
 <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">  
 <fileNamePattern>

D:\Files\Code\_Project\WeChatOrderSystem\Code\log\info.%d.log

</fileNamePattern>  
 </rollingPolicy>  
 </appender>  
  
 <root level="info">  
 <appender-ref ref="consoleLog"></appender-ref>  
 <appender-ref ref="fileInfoLog"></appender-ref>  
 <appender-ref ref="fileErrorLog"></appender-ref>  
 </root>  
</configuration>

* **日志测试类：**

**SpringBoot会自动识别上面配置的logback-spring.xml，并且按照其规则来存放、打印日志**。

@RunWith(SpringRunner.class)  
@SpringBootTest  
@Slf4j **//省去创建Logger对象，直接使用log操作日志**public class LoggerTest {  
 @Test  
 public void test(){  
  **//打印日志，并且自动将日志按照logback-spring.xml文件的配置存放日志**  
 *log*.error("====error====");  
 *log*.warn("====warn====");  
 *log*.info("====info====");  
 *log*.debug("====debug====");  
 }  
}

单元测试

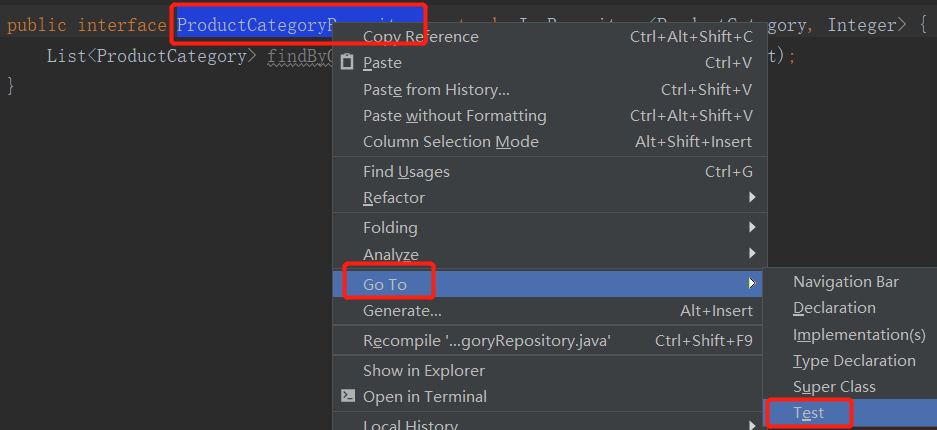
1. **什么是单元测试：**

在编写完一个功能块之后，可以在SpringBoot工程内的test文件夹下，对该功能块进行测试，目的：确保每部分功能块都可以使用，减少工程的调试时间。

**必须包含这两个注解：**@RunWith(SpingRunner.class)、@SpringBootTest

1. **单元测试的方法：**

* 编写完一个.java文件，想要测试该部分代码：鼠标右击方法名->GoTo->Test，即可在test文件夹内生成对应.java所在包的单元测试函数。



* **在生成的单元测试方法中，自己编写逻辑代码，测试原先的模块：**

会用到**Assert**断言，判断测试输入是否符合预期，不符合则抛出异常。



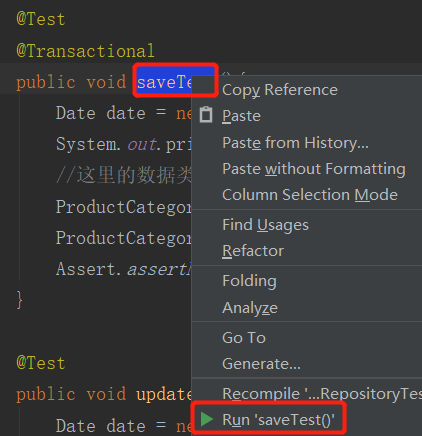
**示例：**单元测试的代码

**@RunWith**(SpringRunner.class)  
**@SpringBootTest**  
public class ProductCategoryRepositoryTest {  
  **@Test**  
 @Transactional  
 public void saveTest(){  
 Date date = new Date();  
 System.*out*.println("\*\*\*\*\*\*\*\*" + date + "\*\*\*\*\*\*\*\*\*");  
  **//这里的数据类型、非空约束等，必须和MySQL数据库中对应表格设置一样，否则会出现数据无法save**  
 ProductCategory productCategory = new ProductCategory("News", 5 , date, date);  
 ProductCategory result = repository.save(productCategory);  
 **Assert**.*assertNotNull*(result);  
 }

}

* **运行一个测试方法：**

在控制面板Run，可以查看对应的输出，也可以Debug。



**注意：**单元测试的注解@Test，导入的包为：**org.junit.Test**，导入错误的包，无法运行测试代码（error：ava.lang.Exception: No runnable methods）

分布式

1. **分布式概念、特点：**

分布式：支持应用程序、服务程序的开发，可以利用物理架构，有多个自治的处理元素、不共享内存，但通过网络发送消息合作。

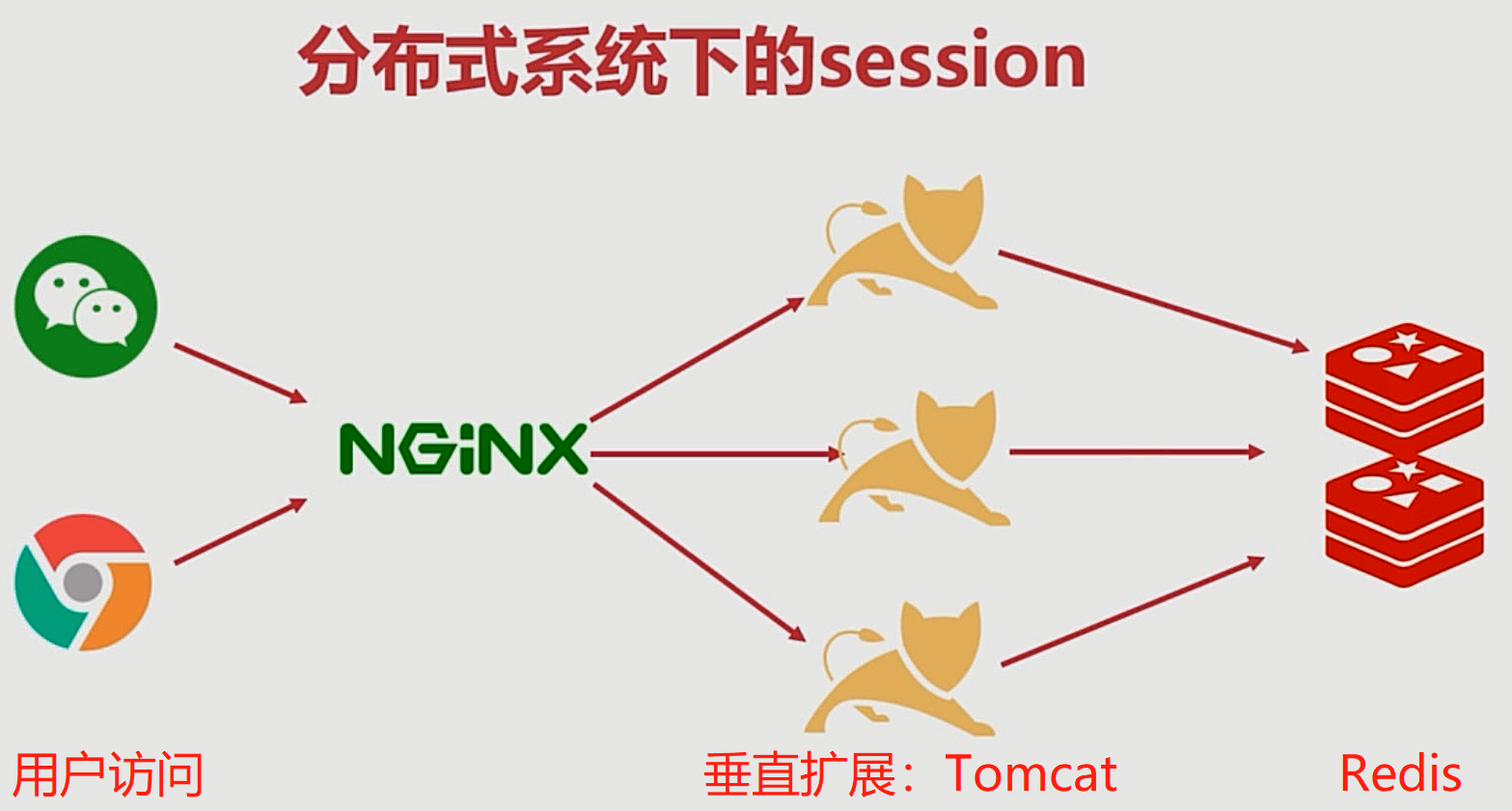
1. **多节点**：
2. **消息通信**：应用之间的数据通过网络发送消息；
3. **不共享内存**：
4. **分布式系统**：多个不同的应用功能集合；

eg：有卖家端 + 买家端的系统；

1. **集群**：多个相同功能应用的集合，因此集群不一定是分布式系统；

eg：有几个买家端的系统；

1. **分布式计算**：多个应用一起计算同一个模型的数据，然后把结果汇总到一起。
2. **分布式Session：**
3. **应用场景：**



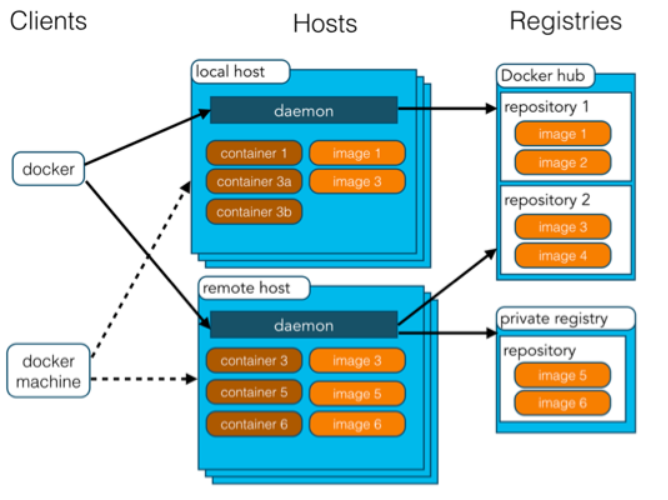
当数据量很大时，数据需要存放在不同的tomcat服务器上，以减小服务器的压力，加快用户访问数据的速度。当用户访问的数据，经由redis查找，找到对应的Tomcat服务器。

1. **常用的两个Key：**
   1. sessionId：
   2. token：

SpringBoot进阶学习

## Docker

1. **基本概念：**
2. **主机：**用于运行docker环境的系统（实体主机、虚拟机都可以）；
3. **客户端：**使用命令来操作主机中的docker环境；
4. **镜像：**指的是一个个软件，用户可以自己打包自己开发环境的某个软件，生成对应的镜像——例如：tomcat、mysql；
5. **容器：**镜像在dokcer中运行后就变成容器，容器具有启动、删除、停止等操作；
6. **仓库：（eg：dokcer hub）**所有的镜像都可以保存在远程仓库，等需要使用的时候直接下载即可，不再需要对系统进行软件安装。



1. **使用Docker**
2. **docker 安装Mysql：**

docker search mysql

docker pull mysql （默认安装latest版本）

（注意：安装的latest的mysql，使用较低版本的navicat是无法登陆，需要更新navicat版本）

1. **docker使用Mysql：**

启动mysql：

docker run --name mysql04 -e MYSQL\_ROOT\_PASSWORD=123456 -d -p 5001:3306 mysql

命令解释：

* --name：为这次启动的mysql起别名；
* MYSQLROOT\_PASSWORD：mysql登陆密码；
* -d：后台运行（daemon）；
* -p：指定在linux系统和docker容器中mysql的映射端口，linux为5000，docker为3306；
* mysql：启动的docker镜像

1. **docker常用的命令：**

* docker images：查看当前系统已经pull安装的镜像；
* docker run 镜像名：启动镜像；
* docker stop containerId：停止运行某个镜像；
* **docker rm containerId**：删除**正在运行**的某个镜像；
* docker rmi 镜像名字：删除某个**下载的镜像**；
* docker ps –a：显示所有运行的镜像；
* docker ps：显示**有端口映射**、且正在运行的镜像；（使用-p启动）
* docker pull 镜像名字：从docker仓库下载镜像；
* docker restart 容器id：重启之前运行的镜像，eg：当vmware重启之后，使用无法访问之前开启的mysql，使用docker ps无法看到mysql正在运行，使用docker ps –a可以看到，此时只需要使用该命令，就可以重启这个镜像，外部可以正常访问mysql数据库；

## Swagger

1. **使用Swagger：**
2. **pom.xml导入依赖：**

**一定要导入swagger-ui，否则浏览器无法打开swagger页面**

**<!--使用swagger-->**  
<dependency>  
 <groupId>io.springfox</groupId>  
 <artifactId>springfox-swagger2</artifactId>  
 <version>2.9.2</version>  
</dependency>  
**<!-- swagger-ui -->**  
<dependency>  
 <groupId>io.springfox</groupId>  
 <artifactId>springfox-swagger-ui</artifactId>  
 <version>2.9.2</version>  
</dependency>

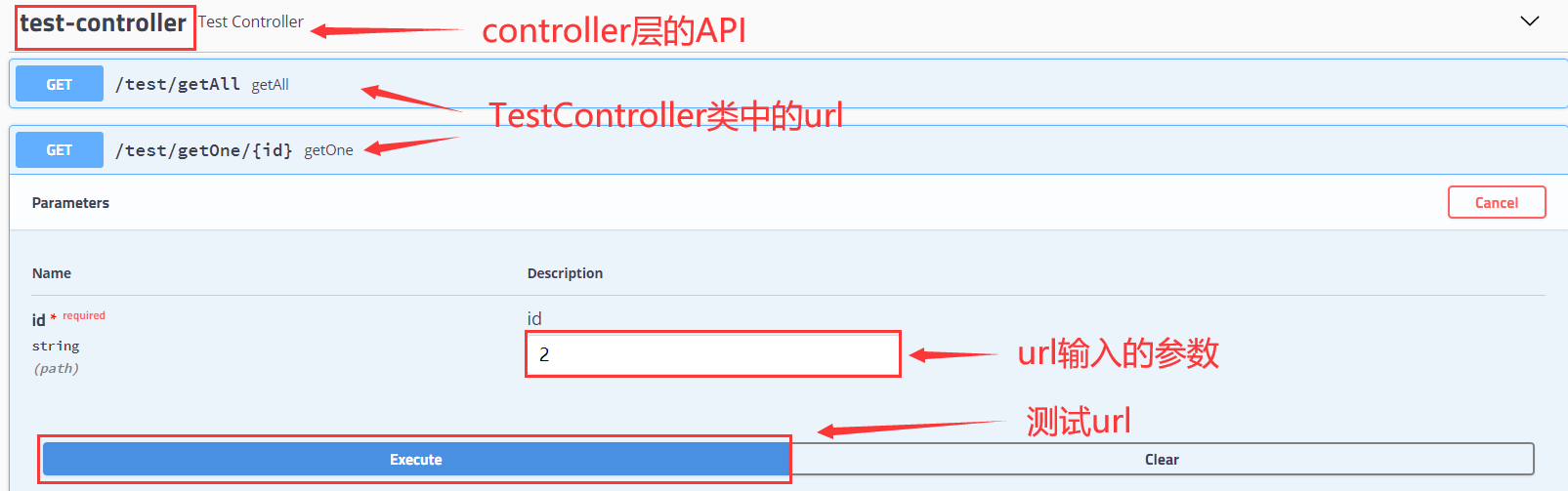
1. **配置springboot中的swagger的config（使用自己的swagger配置）**

@Configuration  
@EnableSwagger2  
public class **SwaggerConfig** {  
 /\*\*  
 \* **配置Swagger的Docket的bean实例**  
 \* @return  
 \*/  
  **@Bean**  
 public Docket createRestApi(){  
 return new Docket(DocumentationType.*SWAGGER\_2*)  
 .apiInfo(apiInfo())  
  **//enable是否启动swagger,如果为False,则swagger不能再浏览器访问**  
 .enable(true)  
 .select()  
  **//当前controller包的路径**  
 .apis(RequestHandlerSelectors.*basePackage*("com.chris.demo.controller"))  
  **//paths过滤什么路径**  
 .paths(PathSelectors.*any*())  
 .build();  
 }

/\*\*  
 \* **配置Swagger信息apiInfo**  
 \* @return  
 \*/  
 private ApiInfo apiInfo() {  
 return new ApiInfoBuilder()  
  **//页面标题**  
 .title("Spring Boot 使用 Swagger2 构建RESTful API")  
  **//创建人**  
 .contact(new Contact("Chris", "https://github.com/Zmq1294550676", "@1294550676@qq.com"))  
  **//版本号**  
 .version("1.0")  
  **//描述**  
 .description("API 描述")  
 .build();  
 }  
}

1. **配置成功：**

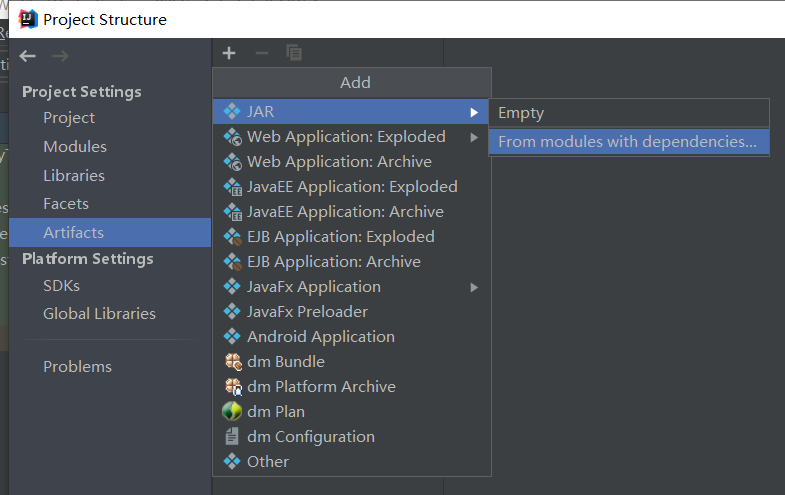
启动springboot项目，浏览器输入**localhost:port/swagger-ui.html**访问swagger，可以直接在浏览器上调试后端的API接口（就可以不再使用postman去调试）

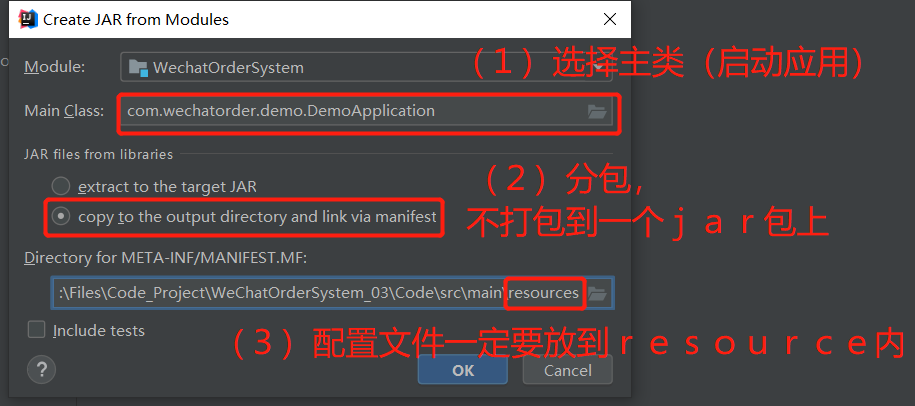


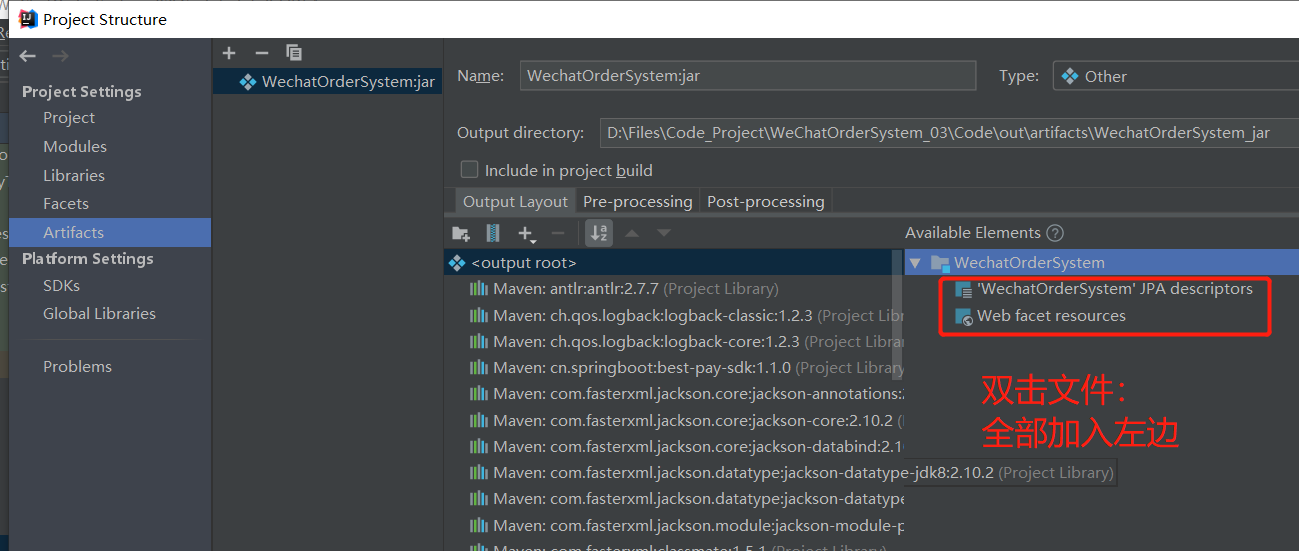


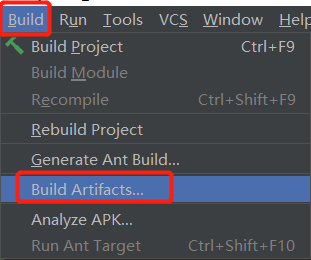
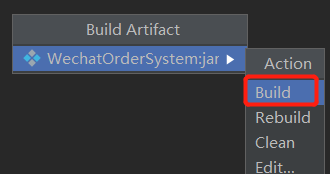
IntalliJ IDEA打包

1. **具体操作**

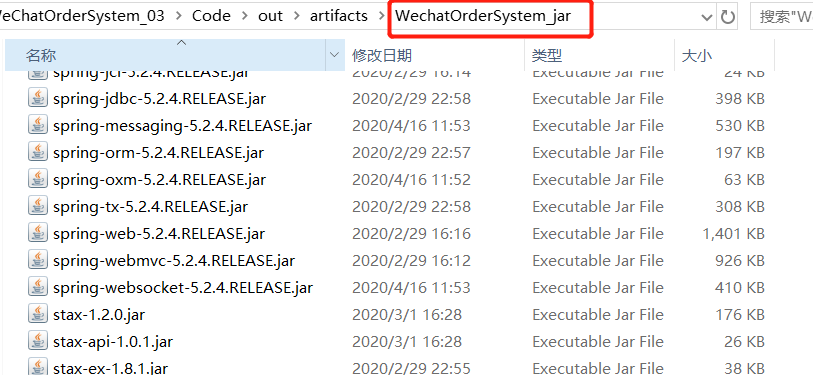






 🡺 

生成的jar包位置：…\out\artifacts\WechatOrderSystem\_jar



运行命令：java –jar WeChatOrderSystem.jar (java –jar 应用的jar包)

1. **另一种打包方式：**

maven工具（但是移植没有测试完成，出现各种奇奇怪怪的错误）

命令：mvn clean package –Dmaven.test.skip=true

在命令行内，cd至微信点餐系统的源码，使用上述命令



SpringBoot常见注解

**一、SpringBoot、Spring**

1. **@SpringBootApplication**

包含@Configuration、@EnableAutoConfiguration、@ComponentScan，通常用在SpringBoot的主类上，标识这是个SpringBoot应用

1. **@Component**

表示该类受Spring、SpringBoot管理，（当该类不好被分类时，可以使用该注解进行分类）

1. **@Repository**

表示该类为持久层组件，用于标注数据访问组件——DAO组件；

1. **@Service**

表示该类为服务层组件；

1. **@Controller**

表示该类为控制层组件，**需要访问、返回页面时使用（而不是使用@RestController）**，例如：访问templates文件夹中的.html文件

1. **@RestController**

表示该类为控制层组件，**包含：@Controller、@Responsebody**

1. **@ResponseBody**

表示该方法返回的结果直接写入Http Response Body中，用于异步获取数据，**给浏览器返回json、xml格式的数据**

1. **@RequestBody**

放在controller中某个方法的参数前，表示该参数必填，用于接收json格式的数据；通常来说是自己封装号的DTO对象，用于获取浏览器传给后端的数据。

1. **@ComponentScan**

组件扫描，扫描到有@Component @Controller @Service等这些注解的类，则把这些类注册为bean\*；

1. **@Configuration**

表示该类为Bean配置的信息源。

1. **@Bean**

产生一个Bean，并给Spring管理；

1. **@EnableAutoConfiguration**

让SpringBoot根据应用所声明的依赖对Spring框架进行自动配置。（一般用在主类上）；

1. **@Autowired**

将配置好的Bean拿来使用，对被参数进行装配，自动对该类的成员变量、方法、构造函数进行标注

1. **@Qualifier：**

当存在多个同类型的Bean时，用@Qualifier（“name”）指定特定的Bean，结合@Autowired使用；

1. **@Resource**

和@Autowired类似

1. **@RequestMapping**

用于表示处理浏览器请求的地址（即：浏览器的请求地址，访问服务器中对应类、方法），可对类、方法进行注释。该注解的参数如下：

* **params**:指定request中必须包含某些参数值是，才让该方法处理。
* **headers**:指定request中必须包含某些指定的header值，才能让该方法处理请求。
* **value**:指定请求的实际地址
* **method**:指定请求的method类型， GET、POST、PUT、DELETE等
* **consumes**:指定处理请求的提交内容类型（Content-Type），如application/json,text/html;
* **produces**:指定返回的内容类型，仅当request请求头中的(Accept)类型中包含该指定类型才返回。

1. **@GetMapping、@PostMapping**

等同于@RequestMapping（value=”/”,method=RequestMethod.Get）、

@RequestMapping（value=”/”,method=RequestMethod.Post）；

具体使用：@GetMapping（“/age”），简化了@RequestMapping的繁琐书写；

1. **@RequestParam**

用在方法的参数前面，等同于request.getParameter；

**//浏览器链接：localhost:8080/create?sender=chris&money=520**

@RequestMapping(value = "/create")  
public User create(**@RequestParam**("sender") String sender,  
 **@RequestParam**("money")BigDecimal money){ …… }

1. **@PathVariable**

路径变量，表示浏览器传入的一个路径参数。

例如：如 RequestMapping(“user/get/mac/{macAddress}”)

**//浏览器链接：localhost:8080/list/2 ，测试id=2**

@GetMapping("/list/**{id}**")  
public User findById**(@PathVariable**("id") Integer id){  
 return repository.findById(id).orElse(null);  
}

**二、JPA**

1. **@Entity**

表示该类是实体类，对应MySQL数据库中的表名是xxx\_entity，

1. **@Table(name=“”)**

表示该类为实体类，若表名和实体类名相同，@Table可省略；

1. **@MappedSuperClass**

用于父类的Entity上 ，父类的属性可以继承；

1. **@NoRepositoryBean**

用作父类的repository，Spring不会去实例化这个类

1. **@Column**

字段名、列名相同，可以省略；

1. **@Id**

表示该属性作为主键；

1. **@GeneratedValue**(strategy=GenerationType.SEQUENCE, generator=”repair\_seq”)

表示主键的生成策略=sequence（取值：Auto（多个数据库间切换）、IDENTITY、native）；策略名= generator=”repair\_seq”

1. **@SequenceGeneretor**(name=“repair\_seq”,sequenceName= “seq\_repair”, allocationSize = 1):

name为sequence的名称，sequenceName为数据库的sequence名称，两个名称可以一致；

1. **@Transient:**

表示该属性并非一个到数据库表的字段的映射,ORM框架将忽略该属性.如果一个属性并非数据库表的字段映射,就务必将其标示为@Transient,否则,ORM框架默认其注解为@Basic；

1. **@Basic(fetch=FetchType.LAZY):**

标记可以指定实体属性的加载方式；

1. **@JsonIgnore:**

将json序列化时将java bean中的一些属性忽略掉,序列化和反序列化都受影响；

1. **@JoinColumn**(name=”loginId”):

一对一：本表中指向另一个表的外键。

一对多：另一个表指向本表的外键。

1. **@OneToOne**
2. **@OneToMany**
3. **@ManyToOne**

对应Hibernate配置文件中的一对一，一对多，多对一。

**三、全局异常处理**

1. **@ControllerAdvice**

包含@Component，可以被扫描到，统一处理异常

1. **@ExceptionHandler(Exception.class)**

用于方法上，表示遇到该异常便执行该方法。例如：

**@ExceptionHandler**(value = Exception.class)  
@ResponseBody  
public Result handle(Exception e) {  
 ……  
}

1. **附：**
2. **@Column：**

实体类属性和数据库表中的字段映射关系；**会把该属性写入数据库（作为数据库中的列名）**

**作用范围：**放在属性之前、get方法之前；

**属性：**

1. name：实体类的标注属性在数据库中对应的名字；
2. nullable：该字段是否可为空，默认为true；
3. updatable：使用update脚本插入数据时，是否需要更新该字段，默认为true；
4. length：该字段的长度，仅当该字段为varchar时，才会生效，默认长度为255；
5. **@JsonIgnoreProperties：**

忽略某个类中的某些字段，可以指定忽略哪些字段；

**eg：**@JsonIgnoreProperties(value = { "handler", "hibernate" })**作用范围：**放在某个引用的类之前。

1. **@JsonIgnore**

忽略某些字段，在controller返回类型的数据时，被标注的属性不会返回。

**eg：**

       @JsonIgnore

       private List<CrmPlanAtMember> crmPlanAtMembers;

**作用范围：**放在属性、get方法、set方法之前

1. **@JoinColumn**

需要结合指类中某个属性和其他表之间的关联（**关联的是其他表的主键，**eg：id等），由于关联关系有多种，需要使用@ManyToOne、@OneToOne、@OneToMany、@ManyToMany来指定。

此处的**关联关系，都指的是在当前类中标注的属性对于当前类是：多对一、一对一、一对多、多对多。**

**作用范围：**某个类中引用其他类（表）作为属性，放在这个属性之前。

**eg：**

    @ManyToOne

   @JoinColumn(name = "country\_id")

    private Country country;

1. **ToString**

        lombok自动给被标注的类生成toString()方法。

1. **@EqualsAndHashCode**

        lombok自动生成equals(Object other)、hashCode()方法，

1. **@Query**

        可以实现使用自己的sql语句进行CRUD操作。

**eg：**查询id最大的员工信息，o代表Employee对象

**@Query**("select o from Employee o where id=(select max(id) from Employee t1)")

         Employee getEmployeeById();

1. **@PathVariable**

将浏览器的URL中传入的参数，直接作为后端代码被该注解标记的方法的形参，可指定使用URL中的哪个参数。

**eg：**

        @RequestMapping("/getOne/{id}")

         public String getOne(**@PathVariable**("id") String id){

             log.info("ID = " + id);

           }

1. **@Transient**

当DTO类（和mysql数据库对应的java类）中，不需要某个属性被写入MySQL数据库中时，可以在该属性前使用@Transient，**该属性就不会被写入数据库**。

1. **@Basic**

是实例类和数据库表中字段映射时最简单的类型。包含java中最基本的数据类型：int、byte、short、long、float、double、boolean。

1. **@RequestBody**

用来接收前端传入的json格式的数据，**前端需要使用POST**的方式提交数据（不能使用GET）

1. 一个请求只有一个RequestBody，一个请求中可以有多个RequestParam
2. RequestBody接收的是请求中的数据；

        RequestParam接收的是请求中的key-value参数；

**eg：使用PostMan来测试POST提交的json格式的数据**

@RequestMapping("/requestBody")

public String restRequestBody(**@RequestBody** String str){

*log*.info(str);

  return str;

}

1. **@ResponseBody**
2. 将java对象转化为json格式的数据，把controller方法返回的数据转化为特定格式，然后写入到response对象的body区域，通常**用于返回json格式、xml格式的数据**。
3. **在使用 @RequestMapping后，返回值通常解析为跳转路径，但是加上 @ResponseBody 后返回结果不会被解析为跳转路径，而是直接写入 HTTP response body 中。 eg：**异步获取 json 数据，加上 @ResponseBody 后，会直接返回 json 数据。
4. **作用范围：**只能用于方法上。

**eg：**

@ResponseBody

@RequestMapping("/hello")

public String hello(){

return "hello world";

}

1. **@JsonnProperty**

对类中的属性进行重命名，修改成自己想要的名字，方便和前端进行数据传输。

1. **@Controller**
2. 被注解的方法，可以返回给浏览器指定的页面，结合视图解析器可以返回jsp、html页面；
3. 若需要返回JSON格式的数据给页面，需要在被注解的方法前面添加上@ResponseBody
4. **@RestController**

**@RestController = @ResponseBody + @Controller**，注解的方法无法返回页面，也视图解析器也无法使用，只能返回return中的内容。

1. **@Autowired**
2. 按照byType自动注入，可以写在bean、set方法上，一般要求被注入的对象必须存在，如果可以为空，则需要使用属性required=false。
3. 可以结合@Qualifier使用，

**@Autowired**

**@Qualifier("baseDao")**

privateBaseDao baseDao;

1. **@Resource**

Resource（这个注解属于J2EE的），默认按照名称进行装配，名称可通过name属性进行指定，如果没有指定name属性，当注解写在字段上时，默认取字段名进行安装名称查找，如果注解写在setter方法上默认取属性名进行装配。当找不到与名称匹配的bean时才按照类型进行装配。但，如果name属性一旦指定，就只会按照名称进行装配。

**@Resource(name="baseDao")**

privateBaseDao baseDao;

1. **@ConfigurationProperties：（读取配置文件中多个参数）**

可以将外部配置文件（比如applicaition.properties）加载进来，填充对象的对应字段的数据，然后供其他Bean使用。

**eg：**

**（1）yml文件**

environment:

production:

url: http://[production.example.com](http://production.example.com/)

name: production mode

**（2）java文件**

@Configuration                    //配置类注解，被自动扫描发现

@PropertySource("classpath:application.yml") //指明配置源文件位置

@**ConfigurationProperties**("environment")   //指明前缀

public class EnvConfig {

    private final Production production=new Production();

    public Production getProduction() {

        return production;

    }

     public static class Production{

           private String url;

           private String name;

           //此处没写set、get方法（但必须有）

     }

}

1. **@Value：（读取配置文件中单个参数）**

可用于读取配置文件中自定义的参数数值。

eg：

**（1）配置文件**

person:

LastName: 飞

**（2）java文件**

@Component

public class Person1 {

@Value("${person.last-name}")

String LastName;

   }

1. **@PropertySource**

加载指定的配置文件，给javaBean配置属性等

1. **java类**

//加载配置文件：user.properties

@**PropertySource**(value = {"classpath:user.properties"})

//指定需要使用的配置文件的**参数前缀**

@**ConfigurationProperties**(prefix = "user")

@Component

public class User {

    private Integer id;

    private String lastName;

}

1. **user.properties文件**

user.id=111

user.lastName=张无忌

1. **@ImportSource**
2. 导入 Spring 的配置文件，如核心配置文件 "beans.xml"，让配置文件里面的内容生效。
3. @ImportSource都是放在配置类上，通常放在启动类上和@SpringBootApplication一起。

**eg：springboot启动时，加载bean.xml配置文件**

**1）启动类**

**@ImportResource**(value = {"classpath:**beans.xml**"})

@SpringBootApplication

public class CocoApplication {

    public static void main(String[] args) {

        SpringApplication.run(CocoApplication.class, args);

}

}

**2）bean.xml配置文件**

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

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

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

       xsi:schemaLocation="<http://www.springframework.org/schema/beans> <http://www.springframework.org/schema/beans/spring-beans.xsd>">

    <!-- 放入到Sping容器中，这是以前Spring的内容，不再累述-->

    <bean id="userService" class="com.lct.service.UserService"/>

</beans>

1. **@Component**

@Component 注解, 意味也将会注册为bean, 其内部也可以依赖注入。(换个方式说,**一般Bean能用的东西,它也能用) 例如: @Autowired、@Inject、@Scope**

1. **@Configuration、@Bean**
2. **@Configuration**

@Configuration用于定义配置类，可替换xml配置文件，**被注解的类内部包含有一个或多个被@Bean注解的方法**，这些方法将会被AnnotationConfigApplicationContext或AnnotationConfigWebApplicationContext类进行扫描，**用于构建bean定义，初始化Spring**。

1. **xml中的<beans></beans>对应注解@Component**
2. **@Bean**
3. **xml中的<bean></bean>对应的注解@Bean**
4. xml中的<Import></Import>对应注解@Import
5. xml中的<context:component-scan>对应注解@ComponentScan；

**eg：**spring在加载Appconfig的时候，Appconﬁg由普通类型转变为cglib代理类型 ，在@Bean注解的类中使用，是单例的，不会创建对个对象。

@ComponentScan("com.jiagouedu")

**@Configuration**

public class AppConfig {

**@Bean**

    public Cat cat(){

        return new Cat();

    }

**@Bean**

    //条件注解，只有TestConditional返回为true时，才能实例化Fox

**@Conditional**(value = TestConditional.class)

    public Fox fox(){

        System.out.println("test conditional");

        return new Fox(cat());

    }

}

1. **@Conditional**

条件注解，只有当value中的条件满足时，才会使用被注解的方法。

1. **@AdviceController**
2. 是Controller的增强版，可以用于全局异常处理、全局数据绑定、全局数据预处理
3. **常用于全局异常处理**：需要结合@ExceptionHandler使用，@ExceptionHandler用于拦截异常，然后用自己自己定义的异常类处理。
4. **@ExcpetionHandler的返回值类型：** **ModelAndView 、@ResponseBody注解、ResponseEntity**

**例如：**

@ResponseBody

**//捕获所有的异常，可以改成其他的异常：eg：运行时异常RunTimeException.calss**

**@ExceptionHandler(value = Exception.class)**

public Map errorHandler(Exception ex) {

**//MyException时自定义的异常类**

    if(**ex** instanceof **MyException**){  

        Map map = new HashMap();

        map.put("code", 404);

        map.put("message", "检测到异常");

    }

    return map;

}

使用时，只需要再可能抛出异常的方法上，throws 自己定义的异常类即可，当抛出异常时，或自动跳转到@ExceptionHandler注解的方法上，进行异常处理。

public String testMyException() throws **MyException**{

    throw new MyException("i am a myException")；

}

1. **SpringBoot缓存注解：@Cacheable、@CacheEcivt、@CachePut**
2. **@Cacheable：**

可用于**方法、类**上，被注解的地方，会缓存这些地方的结果，下次执行同样的参数时，会自动从缓存中获取结果。**每次执行前都会检查Cache中是否存在相同key的缓存元素，如果存在就不再执行该方法，而是直接从缓存中获取结果进行返回，否则才会执行并将返回结果存入指定的缓存中。**

**该注解有3个属性：**

1. **value：**指定缓存名称，至少需要指定一个,表示这些缓存的数据会存放到哪个Cache中。**eg**：@Cacheable(value = {"ca","my2"})
2. **key：**缓存的key，可以为空，表示Spring缓存方法的返回结果时对应的key。**eg**: @Cacheable(value = "ca", key = "**#**使用方法参数名")
3. **condition：**执行缓存到的条件。condition属性默认为空，表示将缓存所有的调用情形。其值是通过SpringEL表达式来指定的，当为true时表示进行缓存。

**eg：当user的id为偶数时，才会对结果进行缓存**

@Cacheable(value={"users"},key="#user.id",**condition**="#user.id%2==0")

public User findUser(User user){}

1. **@CacheEcivt：**

**标注在需要清除缓存元素的方法、类上的。当标记在一个类上时表示其中所有的方法的执行都会触发缓存的清除操作**。@CacheEvict的属性有：**value、key、condition、allEntries和beforeInvocation**。其中value、key和condition的语义与@Cacheable对应的属性类似。即：

* value：表示清除操作是发生在哪些Cache上的（对应Cache的名称）；
* key：表示需要清除的是哪个key，如未指定则会使用默认策略生成的key；
* condition：表示清除操作发生的条件。
* allEntries：传入boolean类型参数，表示是否需要清除缓存中的所有元素。默认为false。

1. **@CachePut：**

将**结果放入缓存**中，和@Cacheable不同的地方是：@CachePut不会去检查缓存中是否存在之前的执行结果，每次都是将执行的结果以键值对的形式存入缓存中。和@Cacheable的属性相同（有3个属性：value、key、condition）

1. **@Caching**

**相当于@Cacheable、@CachePut和@CacheEvict的集合体。**

@Caching注解可以在一个方法或者类上同时指定多个Spring Cache相关的注解。其拥有三个属性：cacheable、put和evict，分别用于指定@Cacheable、@CachePut和@CacheEvict。

**eg：**

**@Caching**(**cacheable** = @Cacheable("users"),

**evict** = { @CacheEvict("cache2"),

@CacheEvict(value = "cache3", allEntries = true) })

public User find(Integer id) {

    returnnull;

}

1. **@Scheduled**

用于实现定时执行某个方法。

**参数：**

1. **格式：**[秒] [分] [小时] [日] [月] [周] [年]
2. **通配符：**

* **：**所有值，eg：在“秒”字段的设置上为\*，表示每一秒触发一次。
* **？：**不指定值，表示在执行任务过程中，不将改字段作为条件计算在内。
* **- ：**区间，eg：5-12，表示在5-12之间的整数都会执行任务。
* **，：**指定多个值，eg：MON，FIR。表示周一、周五执行任务。
* **/**：表示递增触发，eg：在“秒”字段上，设置为5/15，表示从第5秒开始，每15秒触发一次任务（第5、20、35、50秒都会执行任务）。
* **L**：表示最后。
* **W**：表示工作日。

**eg：**

每隔5秒执行一次：\*/5 \* \* \* \* ?

每隔1分钟执行一次：0 \*/1 \* \* \* ?

每天23点执行一次：0 0 23 \* \* ?

每天凌晨1点执行一次：0 0 1 \* \* ?

每月1号凌晨1点执行一次：0 0 1 1 \* ?

1. **@Autowired、@Resource**
2. **@Autowired**

@Autowired只根据type进行注入，不会去匹配name。如果涉及到type无法辨别注入对象时，需要依赖@Qualifier或@Primary注解一起来修饰。

1. **@Qualifier：**

当出现多个实现类时，@Qualifier直接指定需要注入哪个类。

**eg：**

@Autowired

**@Qualifier**("**woman**")    **//指定注入woman类**

private Human human;

1. **@primary：**

注解在需要使用@Autowired自动注入的类上。**@Primary是修饰实现类的，告诉spring，如果有多个实现类时，优先注入被@Primary注解修饰的那个类。**

1. **@Resource**

@Resource是Java自己的注解，有两个属性：name、type；

Spring将@Resource注解的name属性解析为bean的名字，而type属性则解析为bean的类型。如果既不指定name也不指定type属性，这时将通过反射机制使用byName自动注入策略。

1. **@RequestParam：**

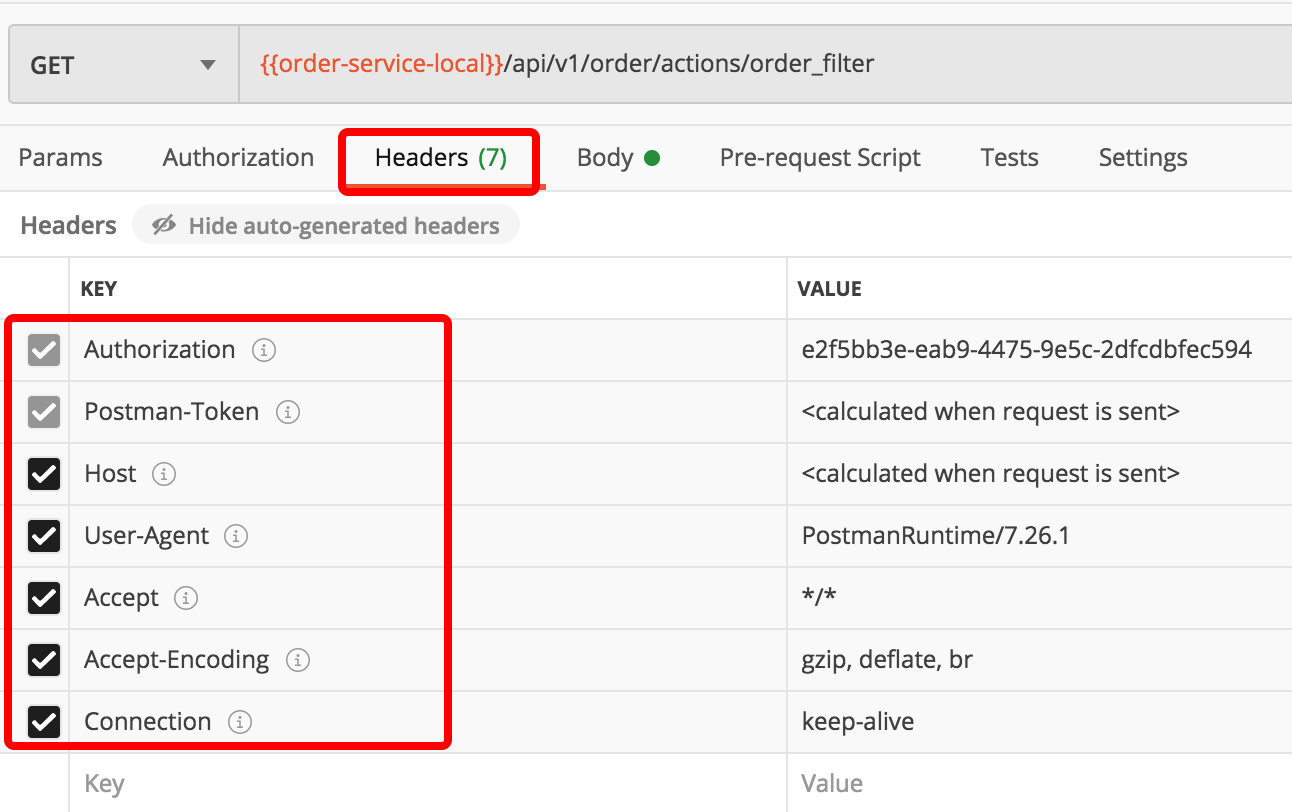
接收的是HTTP请求头的参数，可以接收简单类型的属性，也可以接收对象类型。

可使用的请求方式：GET、PSOT、DELETE

@RequestParam有三个配置参数：

* required 表示是否必须，默认为 true，必须；
* defaultValue 可设置请求参数的默认值。
* value 为接收url的参数名（相当于key值）。

**eg：**postman中接收的header参数



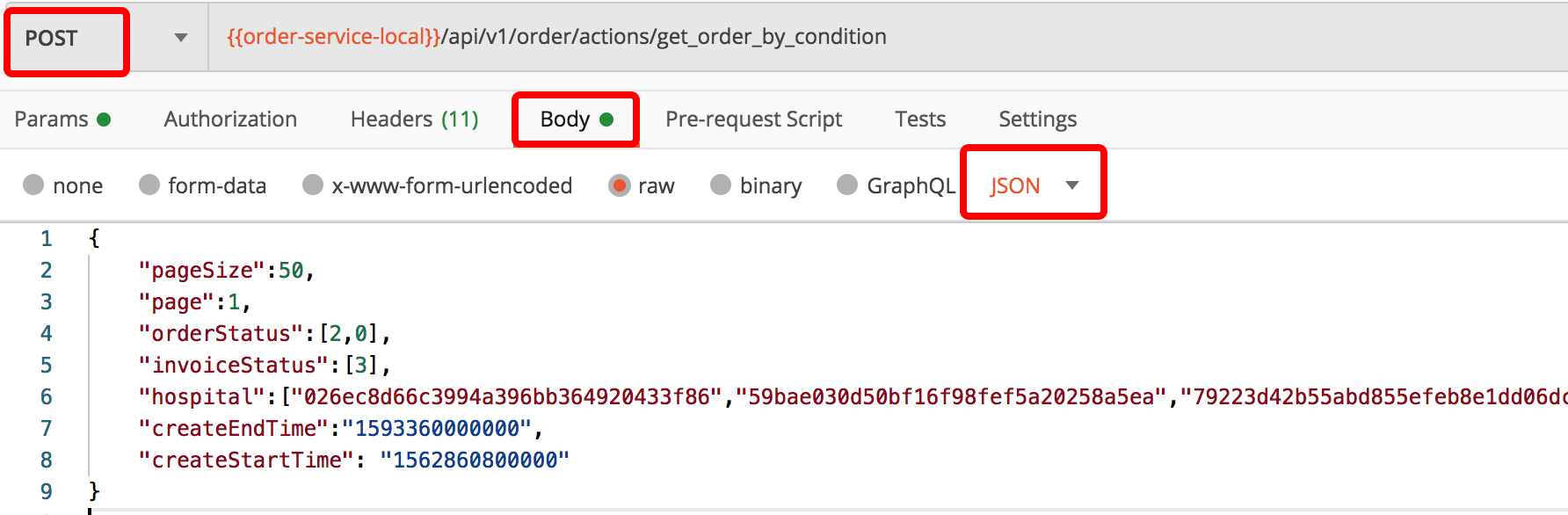
1. **@RequestBody**

可以接收HTTP请求体（requestbody）的参数；

**只能接收POST请求**，通过Http Entity来传递参数，需要在请求头中声明参数类型（eg：content-type：application/json，声明消息主体requestbody是序列化的json字符串）。

不能接收GET请求，因为GET请求没有Http Entity。

**eg：**postman中的POST请求



1. **@Component 和 @Bean 的区别是什么？**

两者的目的一样，都是注册 bean 到 Spring 容器中

1. **作用对象不同: @Component作用于类，而@Bean作用于方法。**
2. **@Component：**表明一个类会作为组件类，并告知 Spring 要为这个类创建 bean。
3. **@Bean：**告诉 Spring 这个方法将会返回一个对象，这个对象要注册为 Spring 应用上下文中的bean。通常方法体中包含了最终产生 bean 实例的逻辑。
4. @Bean 注解比 Component 注解的自定义性更强。

补充知识

1. **IntelliJ IDEA创建SpringBoot工程：**

https://blog.csdn.net/kangkang\_hacker/article/details/80987611

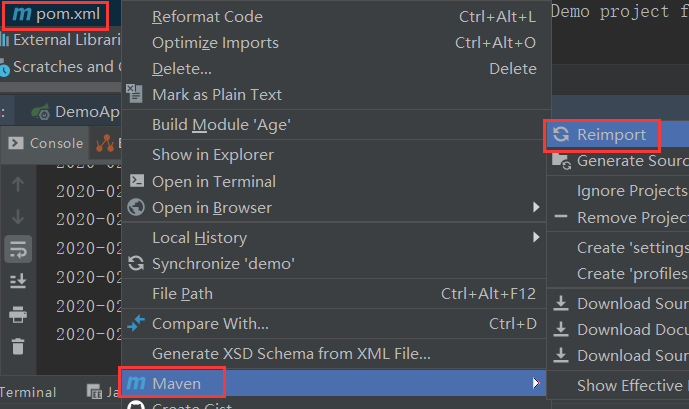
1. **IntelliJ IDEA中Maven工具的使用：**

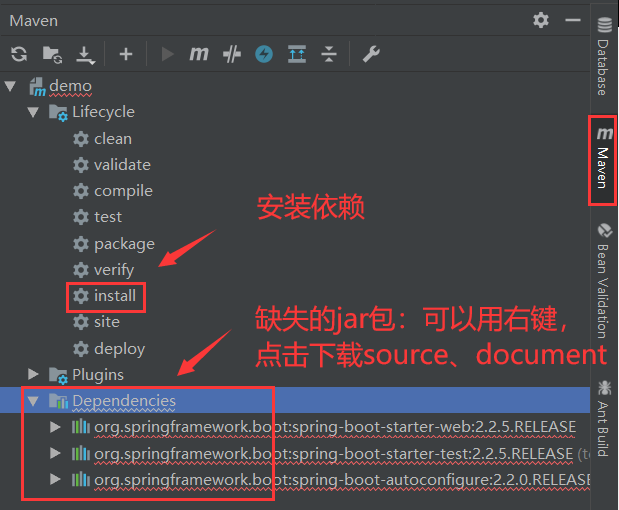
可以用以安装、编译Spring、SpringBoot工程，管理依赖包等操作。

* **当IntelliJ IDEA创建的新工程，出现SprinBoot依赖缺失、cannot resolve symbol ‘@springbootapplication’等依赖缺失错误时。**

需要使用Maven工具进行重新下载。（需要等待的时间较长）

**每次变更pom.xml文件，都需要重新导入**





参考教程：https://blog.csdn.net/harry8sfr/article/details/81947838

* **Maven工具，使用阿里云镜像的方法：（默认国外镜像，太慢）**

修改maven安装路径下/conf/setting.xml文件

直接增加mirrors标签内的内容：

<mirrors>

<mirror>

<id>nexus-aliyun</id>

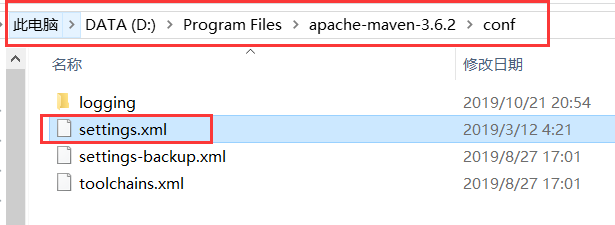
<mirrorOf>central</mirrorOf>

<name>Nexus aliyun</name>

<url>http://maven.aliyun.com/nexus/content/groups/public</url>

</mirror>

</mirrors>



1. **SpringBoot访问templates中的html文件：**

* **基本配置：**
* **pom.xml中：**

<!-- **视图解析器**：用于显示template文件夹中的html页面 -->  
<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-thymeleaf</artifactId>  
</dependency>

* **application.yml中：**

**# 访问template文件夹中的html页面**  
thymeleaf:  
 prefix:  
 classpath: /templates # 访问template下的html文件需要配置模板，映射  
 cache: false # 开发时关闭缓存,不然没法看到实时页面

* **方法一：**

使用thymeleaf作为模板语言，例如访问index.html文件

**Controller.java中：**

@RequestMapping(value = "/index\_1")  
public ModelAndView invitedIndex1(){  
 ModelAndView modelAndView = new ModelAndView();  
 modelAndView.setViewName("index");  
 return modelAndView;  
}

* **方法二：**

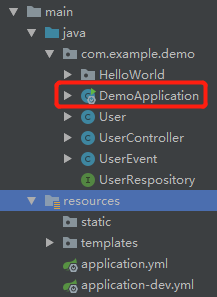
直接返回index字符数据类型，即可访问index.html文件

**Controller.java中：**

/\*\*  
 \* **该类必须使用@Controller注解，不能使用@RestController。**  
 \* **即：只能使用Controller进行跳转**  
 \*/  
@RequestMapping("/index\_2")  
public String invitedIndex2(){  
 return "index";  
}

参看：<https://blog.csdn.net/PJF1501105594/article/details/88316391>

1. **IntelliJ IDEA创建的工程，添加自己的包、代码，必须存放在SpringBoot工程主类：xxxApplication.java同级目录下，否则浏览器无法访问里面的内容。**



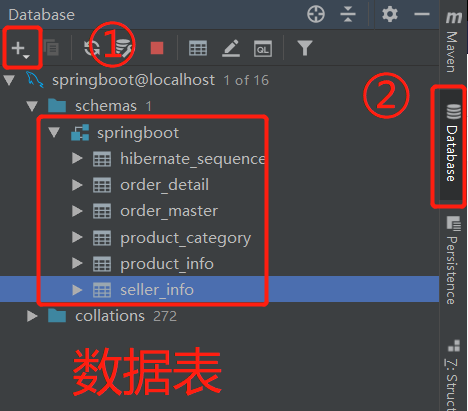
1. **SpringBoot中，使用MySQL数据库注意事项：**

* 删除、保存数据时，一定要注意代码中的数据和MySQL表格中的数据类型、约束条件是否一样。否则会出现如下错误：（约束空、无法执行statement语句等）

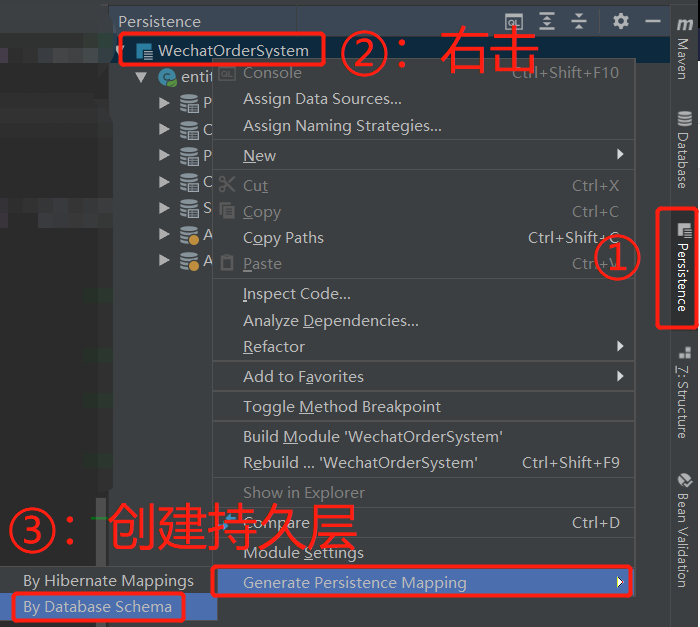
could not execute statement; SQL [n/a]; constraint [null]; nested exception is org.hibernate.except

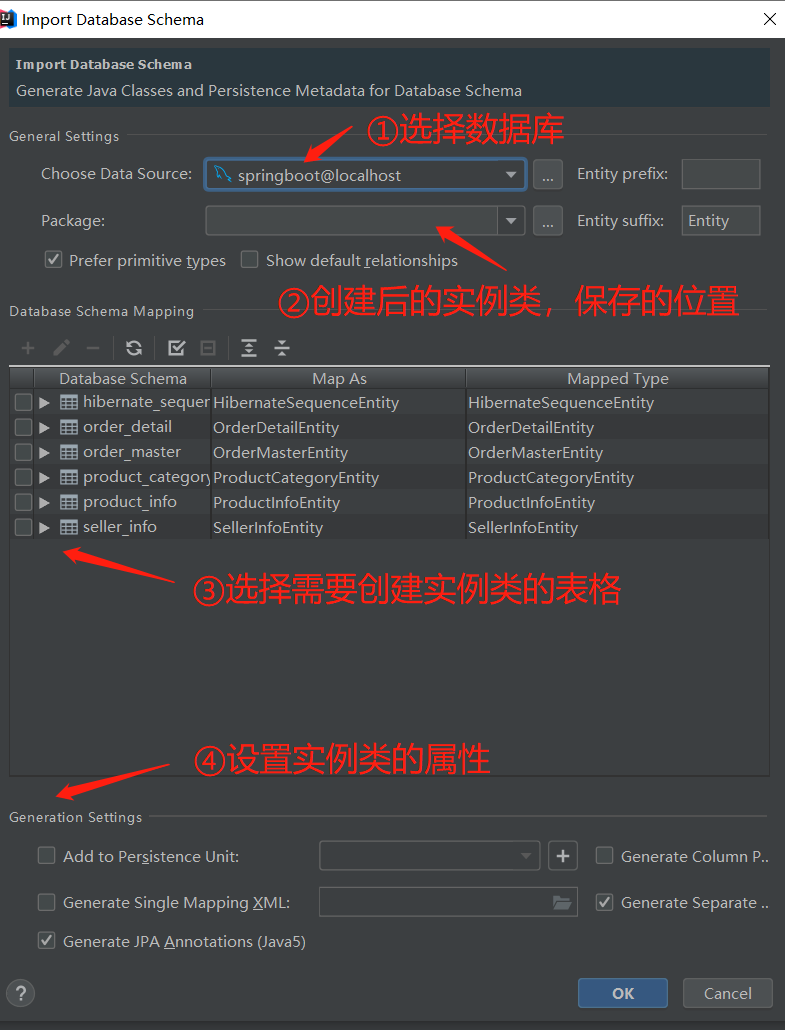
（对应案例：单元测试中有一个示例代码）

* **IntelliJ IDEA连接MySQL：**



* **根据MySQL表来创建实例类：**





参考链接：

https://blog.csdn.net/qq\_27435059/article/details/52494623

1. **Nginx服务器的使用：**

* **修改Nginx服务器的配置：**

1. 虚拟机中的Nignx配置：

文件：vim /usr/local/nginx/conf/nginx.conf

* 修改服务器IP：（填写本机IP地址）

location /sell/ {

proxy\_pass http://**192.168.43.149**:8080/sell/;

}

* 修改web界面的域名：（填写想要的域名，原本只能通过虚拟机IP访问web应用）

server\_name **sell.com**;

* 配置完Nignx后，必须重启Nginx：nginx –s reload

1. 浏览器访问虚拟机中Nginx的web应用：

**（浏览器有用代理服务器的要先关闭，否则无法打开）**

（本方法，每次关闭浏览器后，必须重新执行本步骤，否则无法登录）

* 设置浏览器Cookie：

1）浏览器登陆（虚拟机的IP）：eg：192.168.154.131/#/order

2）浏览器控制页面：document.cookie=’openid=xxx’



* 设置本机的hosts，域名解析：

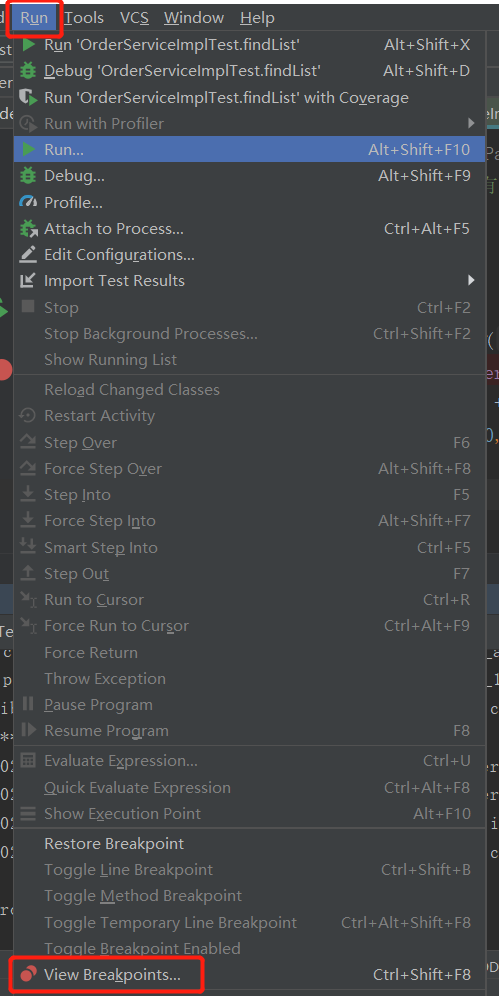
1）文件：C:\Windows\System32\drivers\etc\hosts

2）修改内容（虚拟机IP Ngnix域名）：

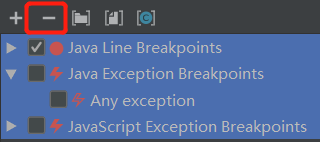
192.168.154.131 sell.com

1. **IntelliJ IDEA调试工具的使用：**

* **Debug下，清除所有断点**：

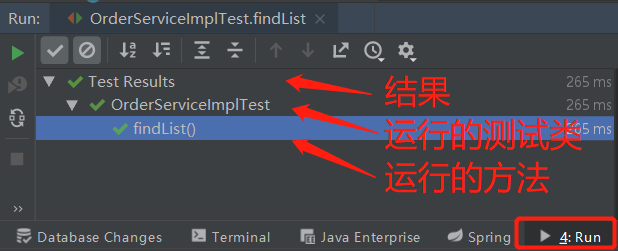


全选🡪Done



* **SpringBoot工程中，运行测试案例，查看某个类的数据：**

点击即可查看对应的运行结构、数据信息

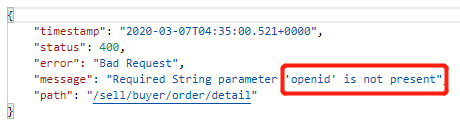


* **使用IntelliJ IDEA 对SpringBoot的应用进行debug（不是对测试代码进行debug）**

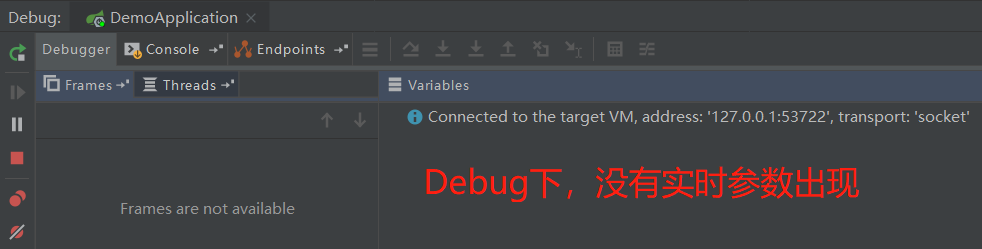
**注意：**URL中，该填写参数的地方一定要填写，否则Debug时会提示参数缺失，且在IntelliJ的Debug界面无法显示对应方法的实施参数。

**eg：URL不填写参数——参数缺失**

postman中，URL不填写参数时：



IntelliJ的Debug界面无参数：



1. **拦截器：**

通常用于，登陆验证、性能检查、记录日志等功能。

**eg：**定义自己的拦截器

**（1）自定义拦截器的内容：**

public class **MyInterceptor** implements **HandlerInterceptor** {  
 **//业务处理前执行**

@Override  
 public boolean **preHandle**(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {  
 System.*out*.println("我的拦截器");  
 return true;  
 }

**//业务处理后，视图渲染前执行（抛出异常时，不执行）**  
 @Override  
 public void **postHandle**(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView) throws Exception {}  
 **//视图渲染后执行**  
 @Override  
 public void **afterCompletion**(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) throws Exception {}  
}

**（2）注册拦截器：**

**@Configuration**  
public class InterceptorConfig implements **WebMvcConfigurer**{  
 @Override  
 public void addInterceptors(InterceptorRegistry registry) {  
  **// 注册拦截器**  
 **InterceptorRegistration** ir = registry.addInterceptor(new **MyInterceptor**());  
  **// 添加拦截请求（拦截controller内配置的url请求路径）**  
 ir.addPathPatterns("/\*");  
 **// 添加不拦截的请求**  
 ir.excludePathPatterns("/login");  
 }  
}

1. **请求方法:PUT、DELETE、GET、POST的使用场景**
2. **PUT：**

对应数据库中的update，常用于更新数据、修改数据，不会增加数据的属性；作用在一个具体的资源上；

1. **DELETE：**

对应数据库中的delete，常用于删除数据，

1. **GET：**

对应数据库中的select，常用于查询数据，具有幂等性（无论get多少次，得到结果都不会变），最多提交1024个字节的数据，并且请求数据会附带在url后面，明文显示，信息不安全。

**eg：**get请求中，显示了name、password

login.action?**name**=hyddd&**password**=idontknow

1. **POST：**

对应数据库中的insert，常用于增加数据、提交数据请求（**需要在FROM表单中使用**）；请求的数据放在http的body区域，和get不同，post不会把请求数据显示在url后面，信息安全，且没有提交数据大小的限制；和post不同的是，post可以增加数据的属性；

1. **事务隔离级别：（级别越高，效率越低）**

TRANSACTION\_NONE ：不使用事务。

TRANSACTION\_READ\_UNCOMMITTED ：允许脏读。

TRANSACTION\_READ\_COMMITTED  ：防止脏读，最常用的隔离级别,并且是大多数数据库的默认隔离级别

TRANSACTION\_REPEATABLE\_READ ：可防止脏读和不可重复读，

TRANSACTION\_SERIALIZABLE ：可防止脏读，不可重复读取和幻读，（事务串行化）会降低数据库的效率

1. **@Transactional使用的场景**

Spring中定义的事务隔离级别：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 隔离级别 | 含义 | 脏读 | 不可重复读 | 幻读 |
| TransactionDefinition.  ISOLATION\_DEFAULT | 使用后端数据库默认的隔离级别 |  |  |  |
| TransactionDefinition.  ISOLATION\_READ\_UNCOMMITTED | 允许读取尚未提交的数据变更(**最低的隔离级别**) | 是 | 是 | 是 |
| TransactionDefinition.  ISOLATION\_READ\_COMMITTED | 允许读取并发事务已经提交的数据 | 否 | 是 | 是 |
| TransactionDefinition.  ISOLATION\_REPEATABLE\_READ | 对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改 | 否 | 否 | 是 |
| TransactionDefinition.  ISOLATION\_SERIALIZABLE | **最高的隔离级别**，完全服从ACID的隔离级别，也是最慢的事务隔离级别，因为它通常是通过完全锁定事务相关的数据库表来实现的 | 否 | 否 | 否 |

1. **Controller、Service层的作用：**
2. **Controller层：**

调用service层的函数，一般来说Controller层的逻辑简单，主要的逻辑都是在Service层实现的，Controller层不添加@Transactional事务机制。

1. **Service层：**

代码业务的主要逻辑都在Service层实现，@Transactional事务回滚也在此处。

1. **常见的POJO（plain ordinaryjava object）简单的java对象**
2. DO：

数据对象（data object），也就是和数据库映射的对象，数据库中存在与之同名的表；

1. BO：

业务对象（business object），业务的逻辑封装；

1. VO：

表现对象（value object）前端页面展示的数据封装，用一个VO对象对应整个界面的值；

1. DTO：

数据传输对象（data transfer object），和前端进行数据传输的对象，拥有get、set方法，一般不会直接使用DO对象和前端进行数据传输，保证数据的安全性；

1. DAO：

数据访问对象（data asscee object），和数据库进行数据交互的对象，主要是CRUD操作。