# 基础

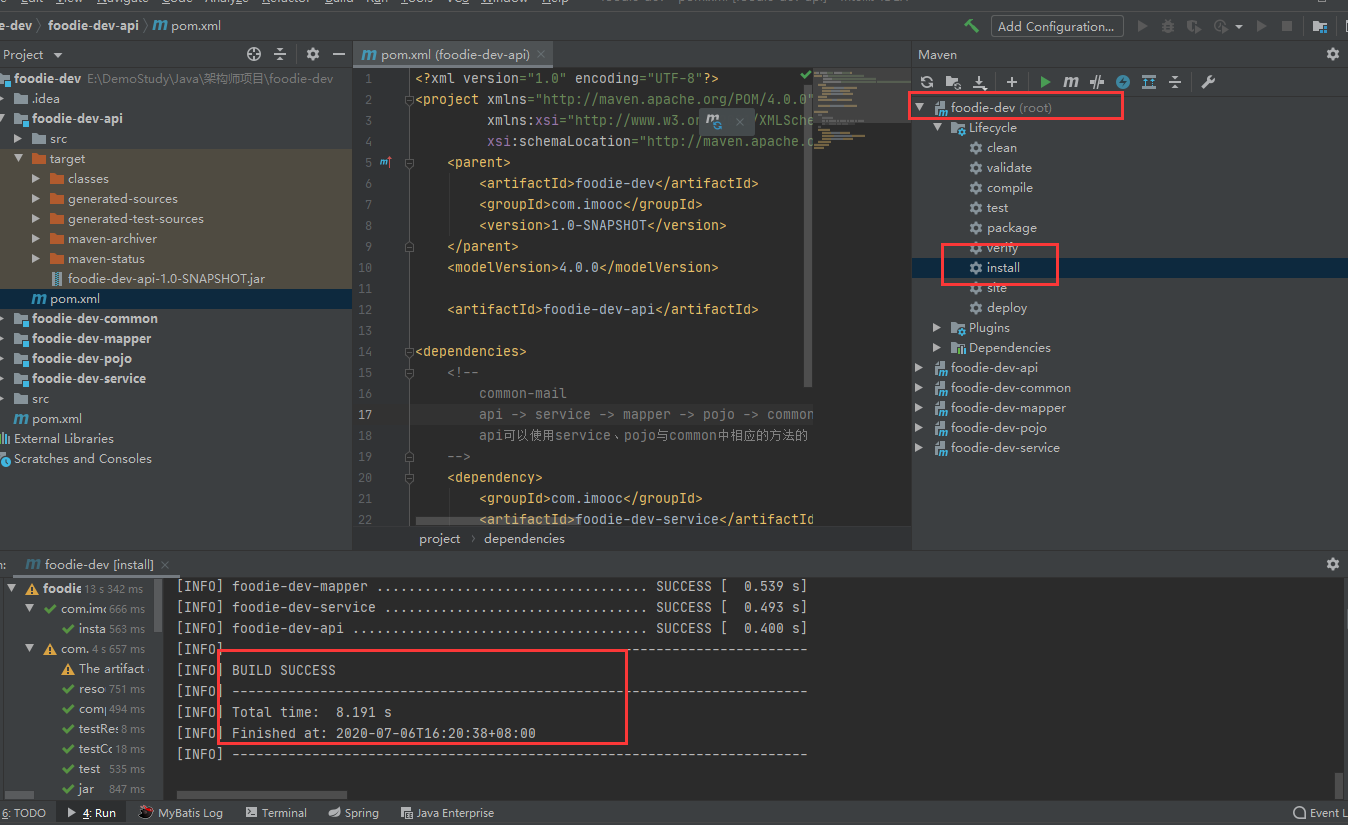
## 技术选型

* SpringMvc是框架 SpringBoot是工具
* SrpingBoot配置都设置在yml中
* SpringBoot集成了多样化中间件[\*-stater]
* SpringBoot从外置tomcat变为内置tomcat
* YAML是一种层级格式，它和.properties很容易互相转换，它的优点是去掉了大量重复的前缀，并且更加易读。

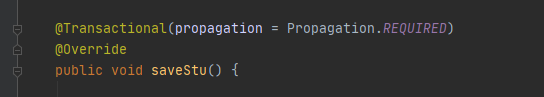
## 数据设计相关

* 数据库外键(最好不要)
  + 对性能有影响
  + 导致 热跟新 无法使用
  + 降低耦合度 影响
  + 难以 数据库分库分表

## 项目(聚合工程)创建完毕之后 因为有依赖关系，所以需要【安装】，安装之后依赖关系才会建立



## 事务的传播



事务传播 - Propagation

* REQUIRED: 使用当前的事务，如果当前没有事务，则自己新建一个事务，子方法是必须运行在一个事务中的；
  + 如果当前存在事务，则加入这个事务，成为一个整体。 举例：领导没饭吃，我有钱，我会自己买了自己吃；领导有的吃，会分给你一起吃。
* SUPPORTS: 如果当前有事务，则使用事务；如果当前没有事务，则不使用事务。
  + 举例：领导没饭吃，我也没饭吃；领导有饭吃，我也有饭吃。
* MANDATORY: 该传播属性强制必须存在一个事务，如果不存在，则抛出异常
  + 举例：领导必须管饭，不管饭没饭吃，我就不乐意了，就不干了（抛出异常）
* REQUIRES\_NEW: 如果当前有事务，则挂起该事务，并且自己创建一个新的事务给自己使用；如果当前没有事务，则同 REQUIRED
  + 举例：领导有饭吃，我偏不要，我自己买了自己吃
* NOT\_SUPPORTED: 如果当前有事务，则把事务挂起，自己不适用事务去运行数据库操作
  + 举例：领导有饭吃，分一点给你，我太忙了，放一边，我不吃
* NEVER: 如果当前有事务存在，则抛出异常
  + 举例：领导有饭给你吃，我不想吃，我热爱工作，我抛出异常
* NESTED: 如果当前有事务，则开启子事务（嵌套事务），嵌套事务是独立提交或者回滚；

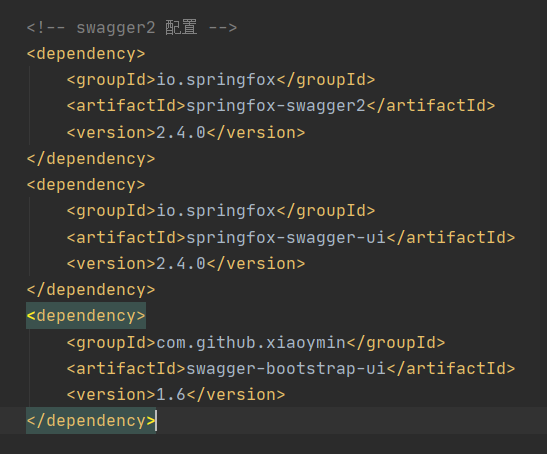
如果当前没有事务，则同 REQUIRED。

但是如果主事务提交，则会携带子事务一起提交。

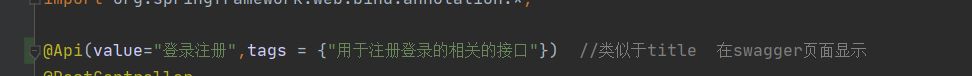
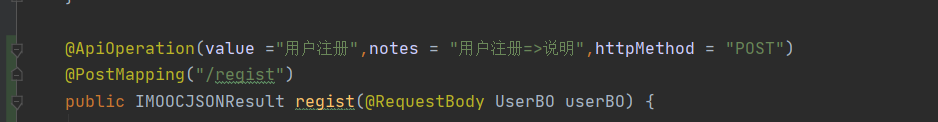
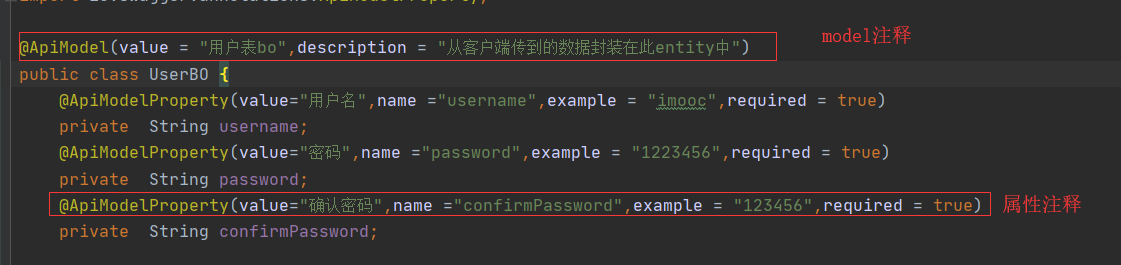
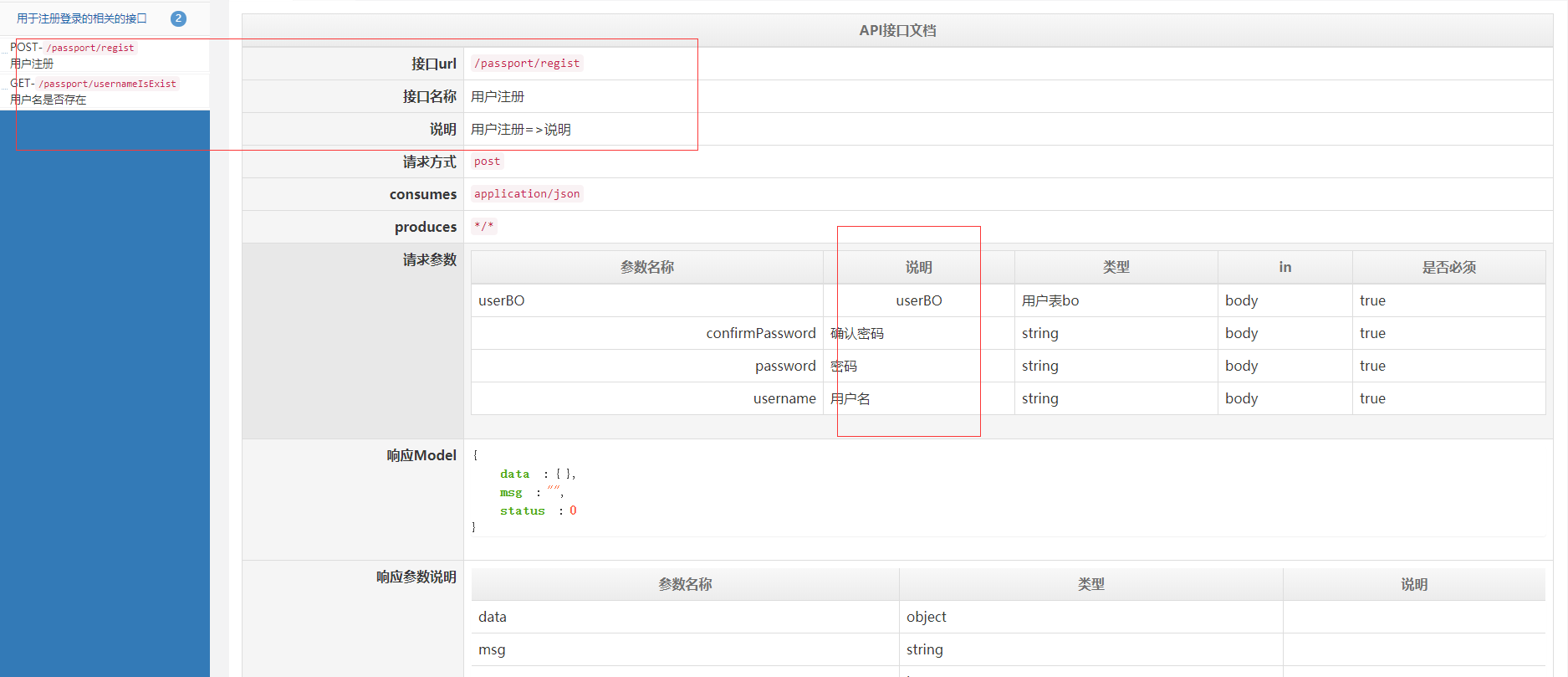
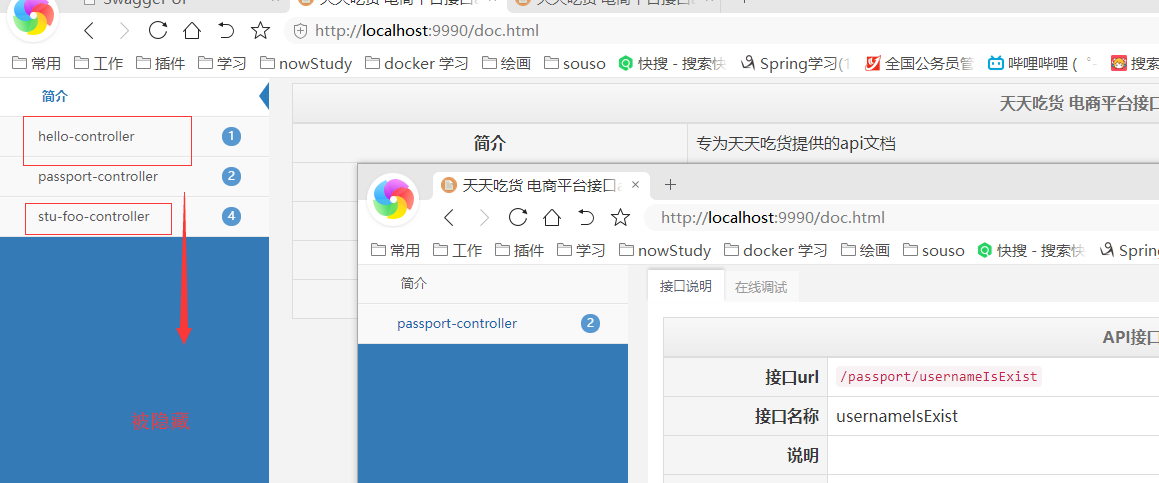
如果主事务回滚，则子事务会一起回滚。相反，子事务异常，则父事务可以回滚或不回滚。

* + 举例：领导决策不对，老板怪罪，领导带着小弟一同受罪。小弟出了差错，领导可以推卸责任。

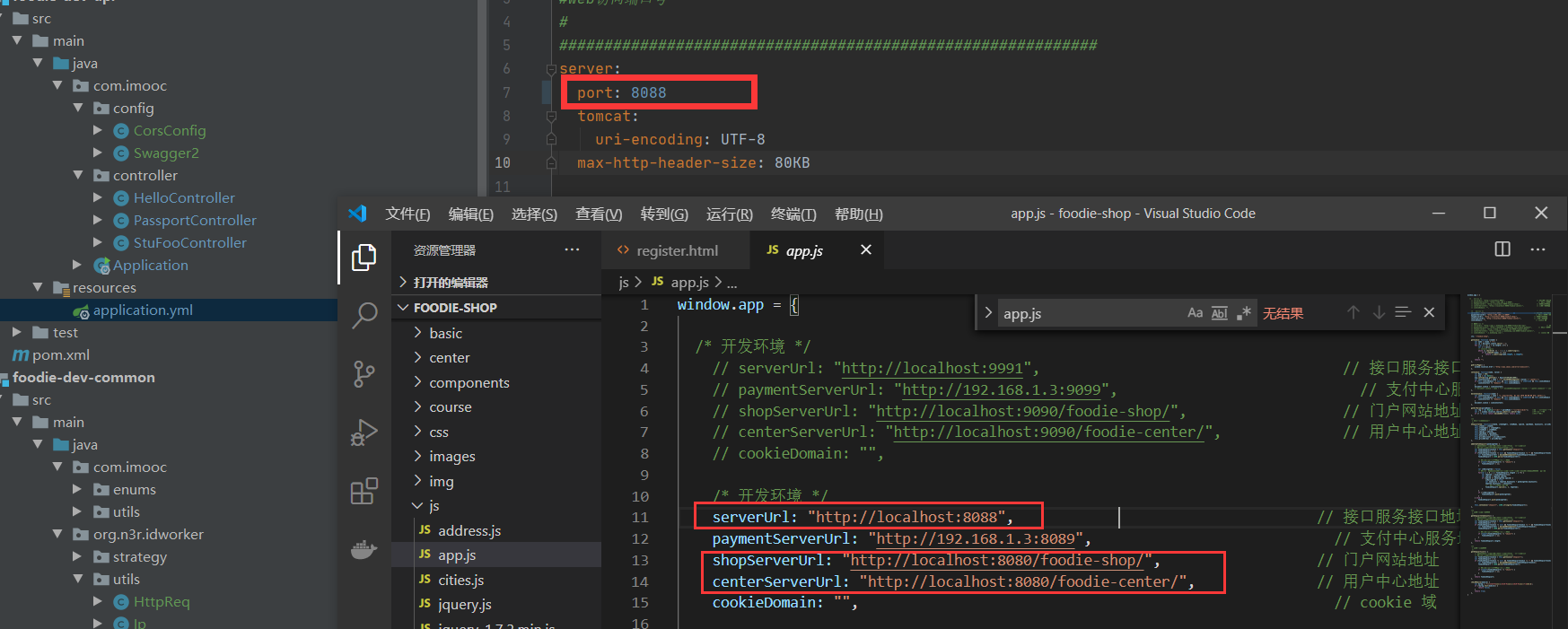
## api接口

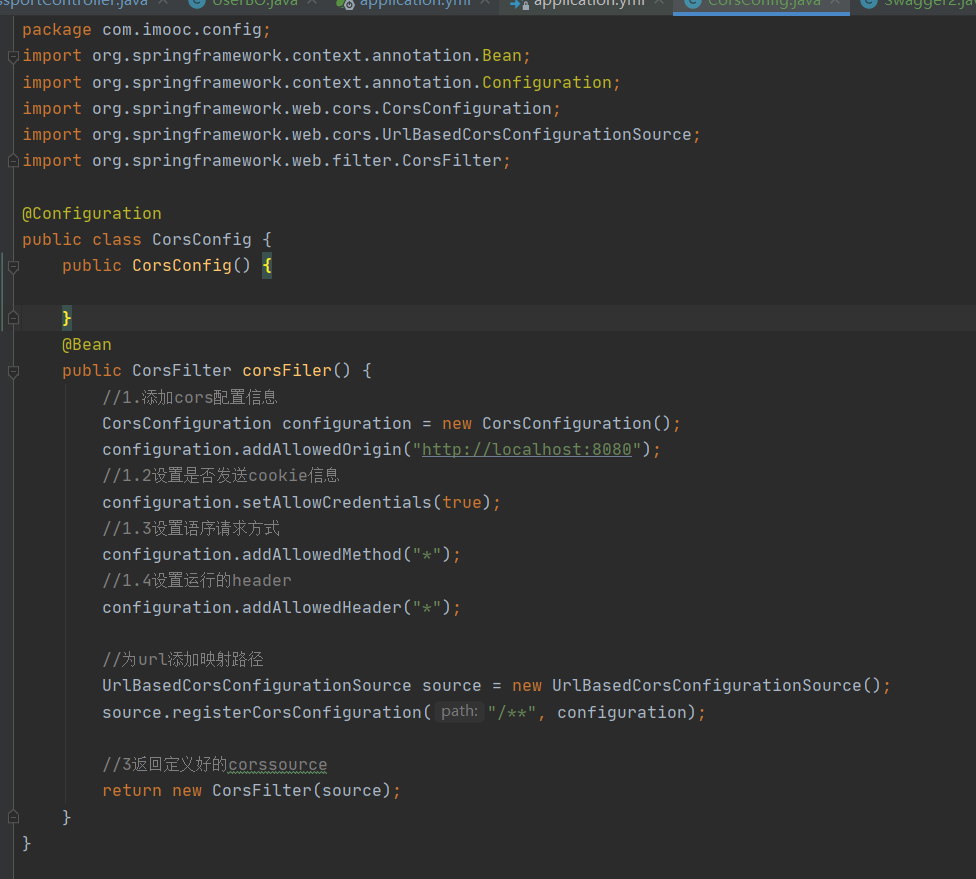
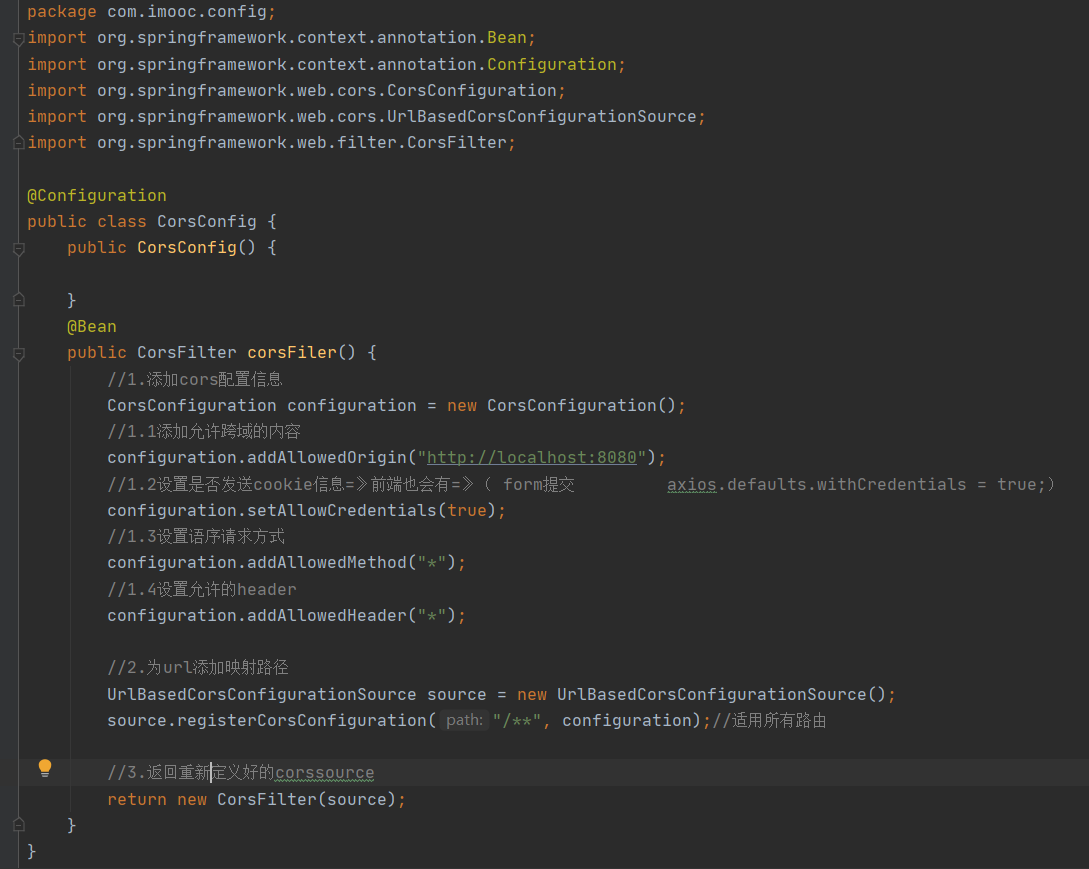
* 需要现在pom.xml中引入以下文件
  + 

|  |
| --- |
| <!-- swagger2 配置 -->  <dependency>  <groupId>io.springfox</groupId>  <artifactId>springfox-swagger2</artifactId>  <version>2.4.0</胜利ersion>  </dependency>  <dependency>  <groupId>io.springfox</groupId>  <artifactId>springfox-swagger-ui</artifactId>  <version>2.4.0</胜利ersion>  </dependency>  <dependency>  <groupId>com.github.xiaoymin</groupId>  <artifactId>swagger-bootstrap-ui</artifactId>  <version>1.6</胜利ersion>  </dependency> |

* 创建swagger相关内容
  + 
* 添加各种注释
  + 对controller的注释
    - 
  + 对active的注释
    - 
  + 对model的注释
    - 
    - 内容显示
      * 
* 取消页面显示
  + 
  + 

## 关于跨域问题=》使用CorsFilter解决



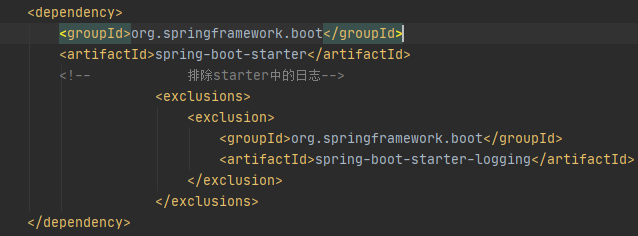
* 由于前端端口(8080)和后台端口(8088)不是同一个端口，在前端进行请求访问后台的时候，会出现跨域问题，所以需要在后台添加以下模块（当前在tomcat下运行）
  + 

|  |
| --- |
| package com.imooc.config;  import org.springframework.context.annotation.Bean;  import org.springframework.context.annotation.Configuration;  import org.springframework.web.cors.CorsConfiguration;  import org.springframework.web.cors.UrlBasedCorsConfigurationSource;  import org.springframework.web.filter.CorsFilter;  @Configuration  public class CorsConfig {  public CorsConfig() {  }  @Bean  public CorsFilter corsFiler() {  //1.添加cors配置信息  CorsConfiguration configuration = new CorsConfiguration();  //1.1添加允许跨域的内容  configuration.addAllowedOrigin("http://localhost:8080");  //1.2设置是否发送cookie信息=》前端也会有=》（ form提交 axios.defaults.withCredentials = true;）  configuration.setAllowCredentials(true);  //1.3设置语序请求方式  configuration.addAllowedMethod("\*");  //1.4设置允许的header  configuration.addAllowedHeader("\*");  //2.为url添加映射路径  UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();  source.registerCorsConfiguration("/\*\*", configuration);//适用所有路由  //3.返回重新定义好的corssource  return new CorsFilter(source);  }  } |

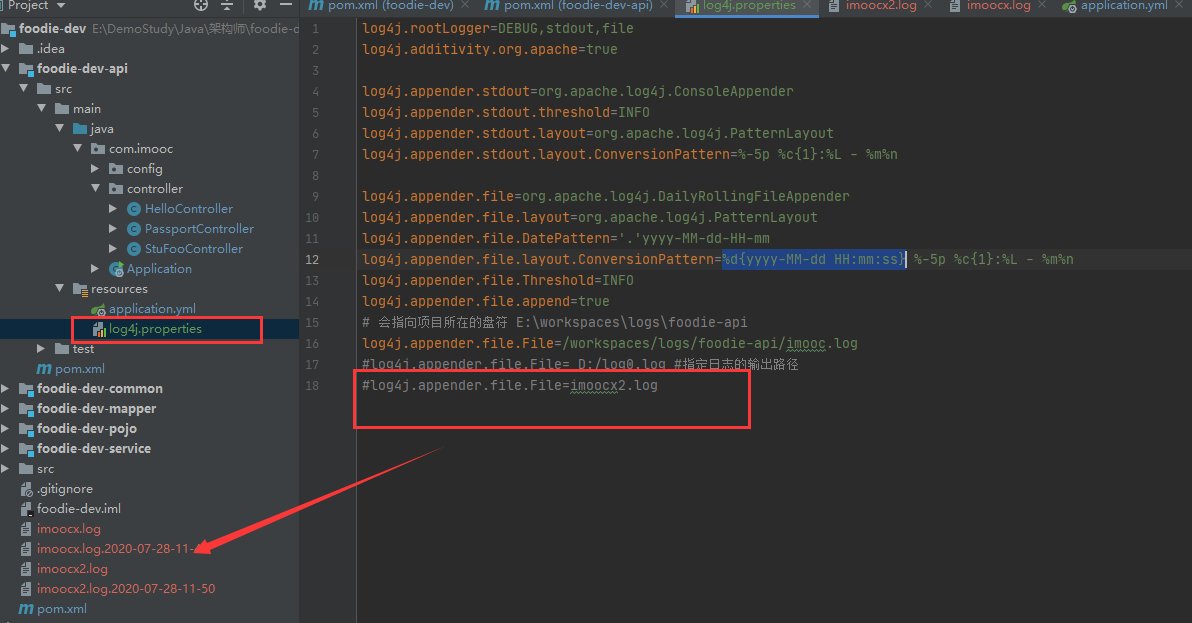
## log4j 记录日志

* 首先在pom.xml中添加以下配置
  + 

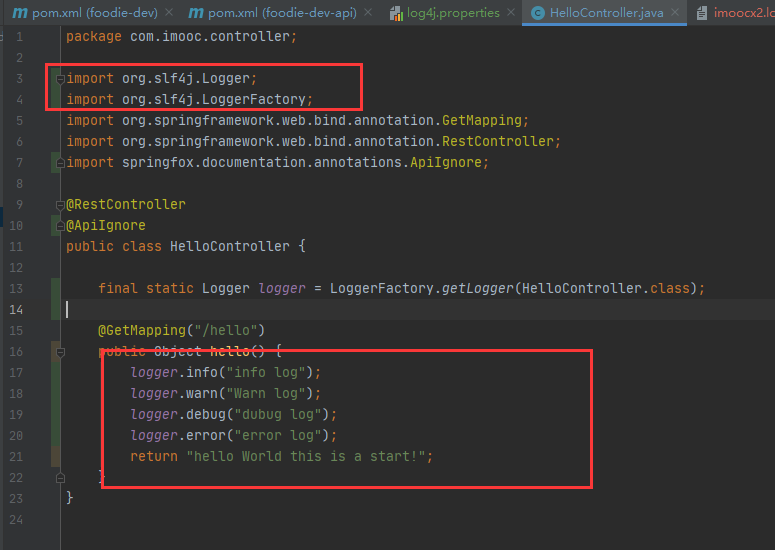
|  |
| --- |
| <!--引入日志依赖 抽象层 与 实现层-->  <dependency>  <groupId>org.slf4j</groupId>  <artifactId>slf4j-api</artifactId>  <version>1.7.21</version>  </dependency>  <dependency>  <groupId>org.slf4j</groupId>  <artifactId>slf4j-log4j12</artifactId>  <version>1.7.21</version>  </dependency> |

* 如果pom.xml中有引用 spring-boot-starter，则需要去掉本身日志
  + 

|  |
| --- |
| <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter</artifactId>  <!-- 排除starter中的日志-->  <exclusions>  <exclusion>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-logging</artifactId>  </exclusion>  </exclusions>  </dependency> |

* 然后添加相关文件 log4j.properties
  + 

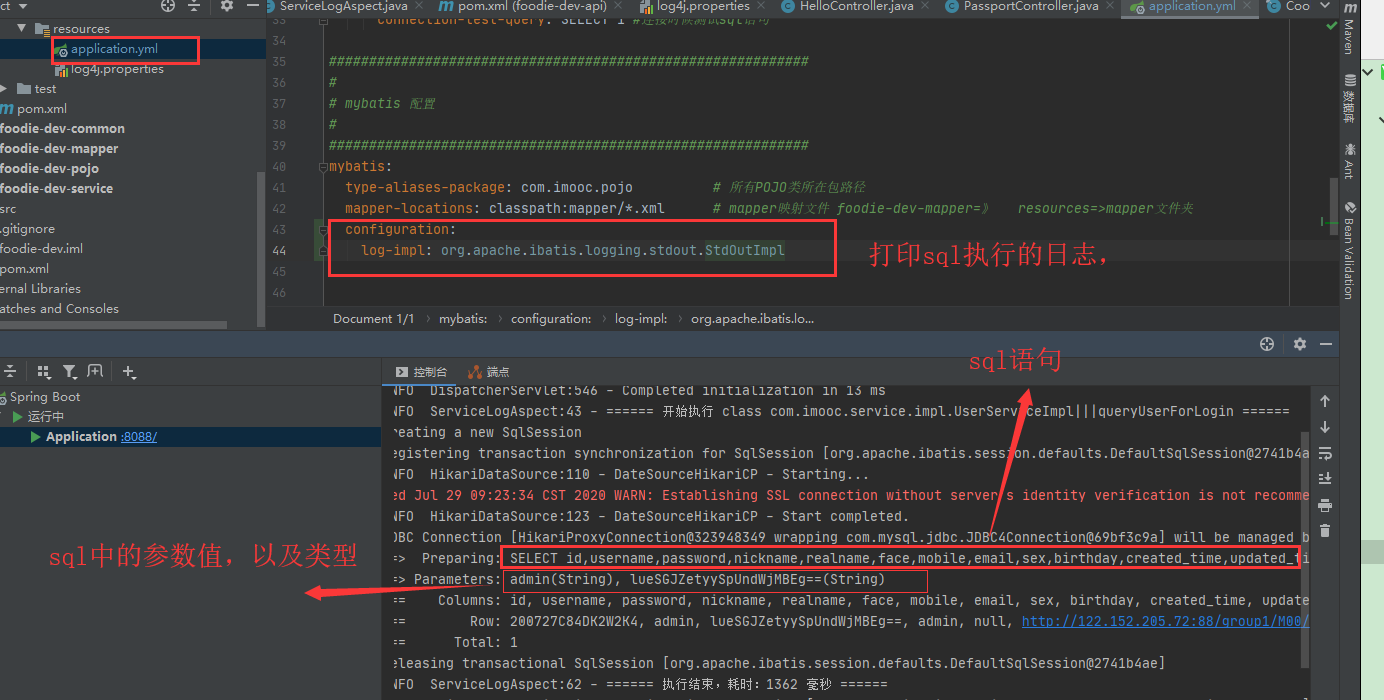
|  |
| --- |
| log4j.rootLogger=DEBUG,stdout,file  log4j.additivity.org.apache=true  log4j.appender.stdout=org.apache.log4j.ConsoleAppender  log4j.appender.stdout.threshold=INFO  log4j.appender.stdout.layout=org.apache.log4j.PatternLayout  log4j.appender.stdout.layout.ConversionPattern=%-5p %c{1}:%L - %m%n  log4j.appender.file=org.apache.log4j.DailyRollingFileAppender  log4j.appender.file.layout=org.apache.log4j.PatternLayout  log4j.appender.file.DatePattern='.'yyyy-MM-dd-HH-mm  log4j.appender.file.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %-5p %c{1}:%L - %m%n  log4j.appender.file.Threshold=INFO  log4j.appender.file.append=true  # 会指向项目所在的盘符 E:\workspaces\logs\foodie-api  log4j.appender.file.File=/workspaces/logs/foodie-api/imooc.log  #log4j.appender.file.File= D:/log0.log #指定日志的输出路径  #log4j.appender.file.File=imoocx2.log |

* 测试
  + 

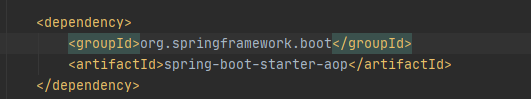
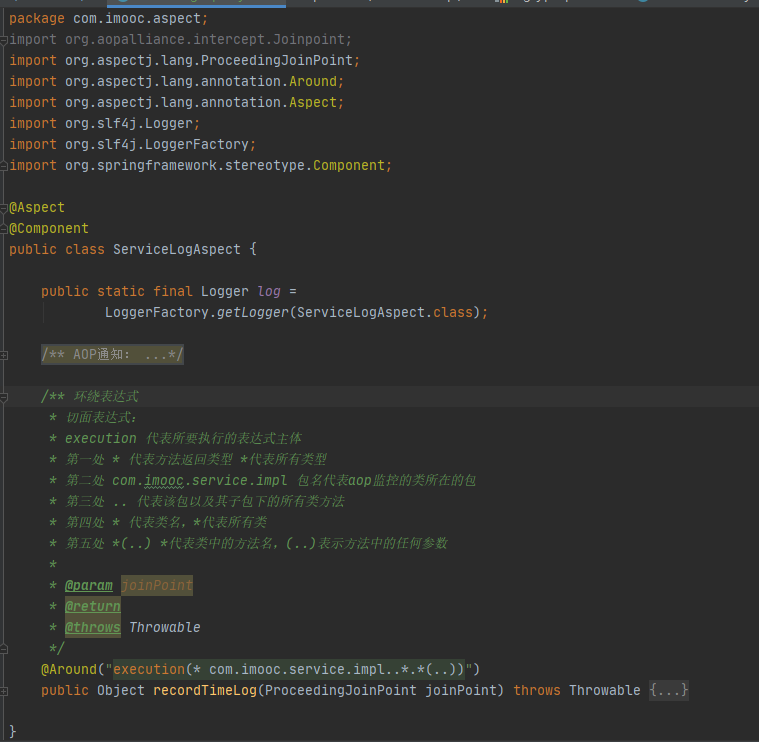
### 记录sql执行记录

configuration:

log-impl: org.apache.ibatis.logging.stdout.StdOutImpl



## Spring Aop

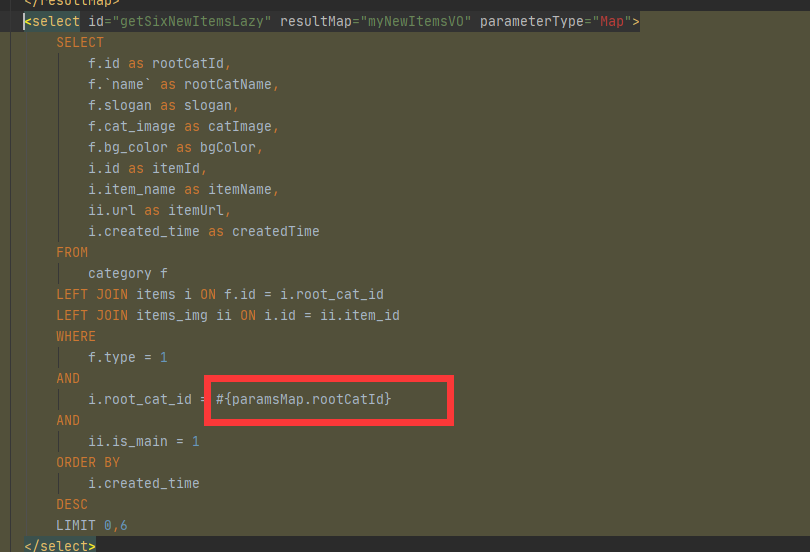
* Spring aop有五种类型
  + 前置通知：在方法调用之前执行 before
  + 后置通知：在方法正常调用之后执行 afterReturn
  + 环绕通知：在方法调用之前和之后，都分别可以执行的通知 around
  + 异常通知：如果在方法调用过程中发生异常，则通知 afterThrow
  + 最终通知：在方法调用之后执行 after
* 添加相应依赖
  + 
* 添加相对应类
  + 
* 代码部分

|  |
| --- |
| package com.imooc.aspect;  import org.aopalliance.intercept.Joinpoint;  import org.aspectj.lang.ProceedingJoinPoint;  import org.aspectj.lang.annotation.Around;  import org.aspectj.lang.annotation.Aspect;  import org.slf4j.Logger;  import org.slf4j.LoggerFactory;  import org.springframework.stereotype.Component;  @Aspect  @Component  public class ServiceLogAspect {  public static final Logger log =  LoggerFactory.getLogger(ServiceLogAspect.class);  /\*\* 环绕表达式  \* 切面表达式：  \* execution 代表所要执行的表达式主体  \* 第一处 \* 代表方法返回类型 \*代表所有类型  \* 第二处 com.imooc.service.impl 包名代表aop监控的类所在的包  \* 第三处 .. 代表该包以及其子包下的所有类方法  \* 第四处 \* 代表类名，\*代表所有类  \* 第五处 \*(..) \*代表类中的方法名，(..)表示方法中的任何参数  \*  \* @param joinPoint  \* @return  \* @throws Throwable  \*/  @Around("execution(\* com.imooc.service.impl..\*.\*(..))")  public Object recordTimeLog(ProceedingJoinPoint joinPoint) throws Throwable {  //某一个service 中的某一个方法  log.info("====== 开始执行 {}|||{} ======",  joinPoint.getTarget().getClass(),  joinPoint.getSignature().getName());  // 记录开始时间  long begin = System.currentTimeMillis();  // 执行目标 service  Object result = joinPoint.proceed();  // 记录结束时间  long end = System.currentTimeMillis();  long takeTime = end - begin;  if (takeTime > 3000) {  log.error("====== 执行结束，耗时：{} 毫秒 ======", takeTime);  } else if (takeTime > 2000) {  log.warn("====== 执行结束，耗时：{} 毫秒 ======", takeTime);  } else {  log.info("====== 执行结束，耗时：{} 毫秒 ======", takeTime);  }  return result;  }  } |

## 关于mybatis

### 中使用map





### 使用foreach

Mapper下

|  |
| --- |
| List<ShopcartVO> queryItemsBySpecIds(@Param("paramsList") List specIdsList); |

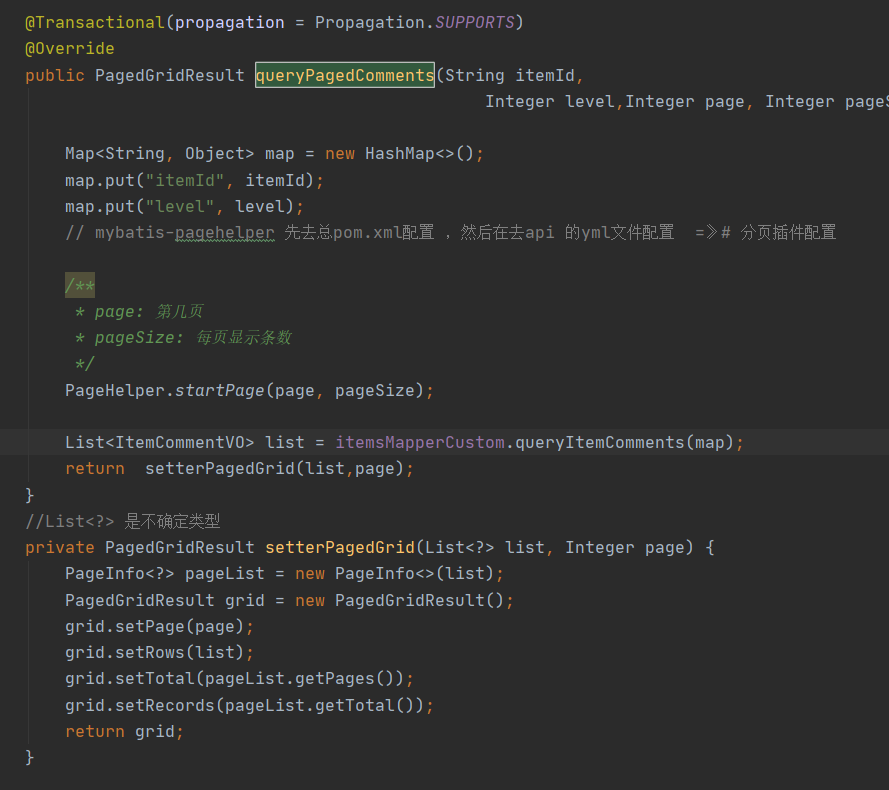
Xml文件夹下

|  |
| --- |
| <select id="queryItemsBySpecIds" parameterType="List" resultType="com.imooc.pojo.vo.ShopcartVO">  SELECT \* FROM table where id IN  <foreach collection="paramsList" index="index" item="specId" open="(" separator="," close=")">  #{specId}  </foreach>  </select> |

### 关于分页插件-mybatis-pagehelper

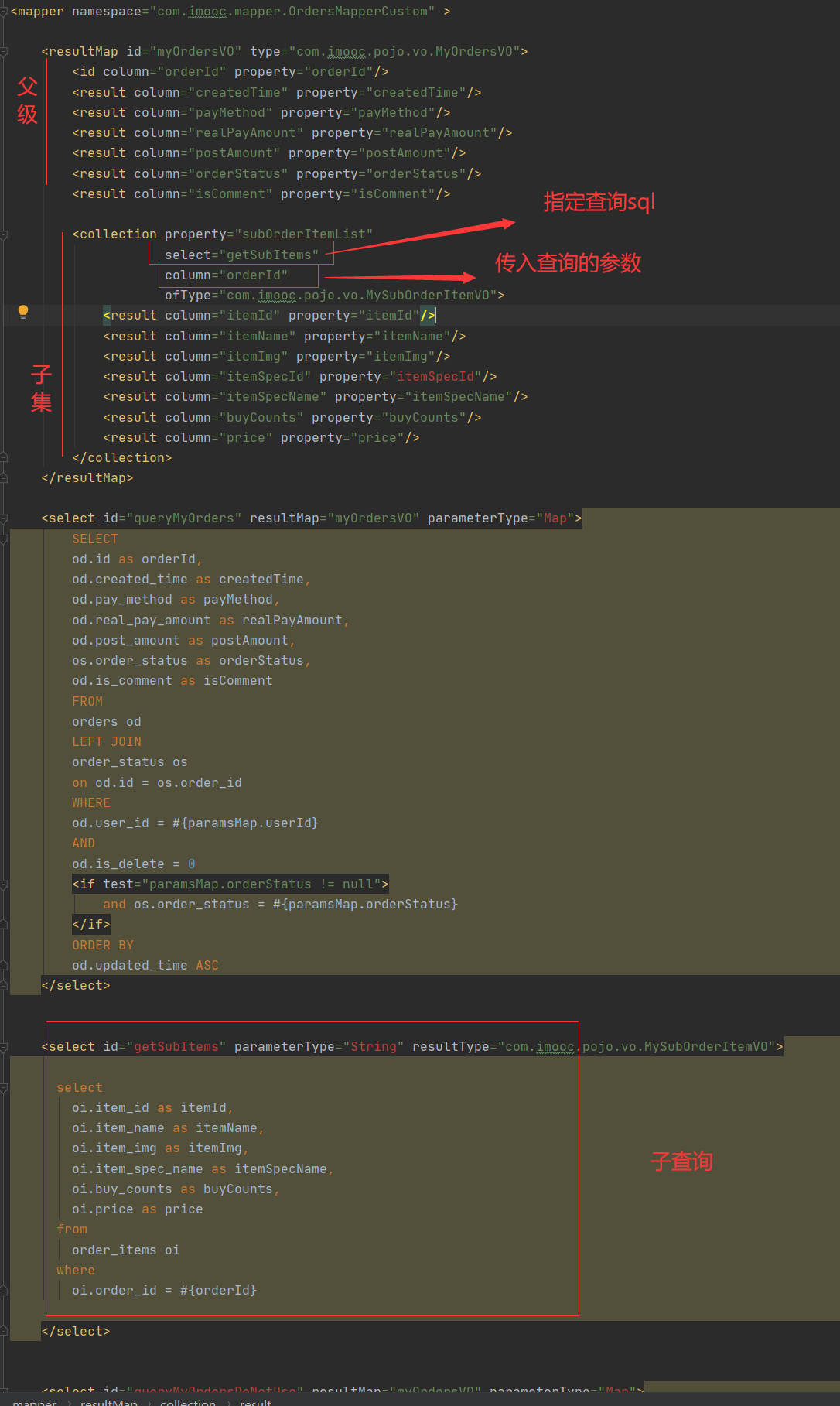


使用代码

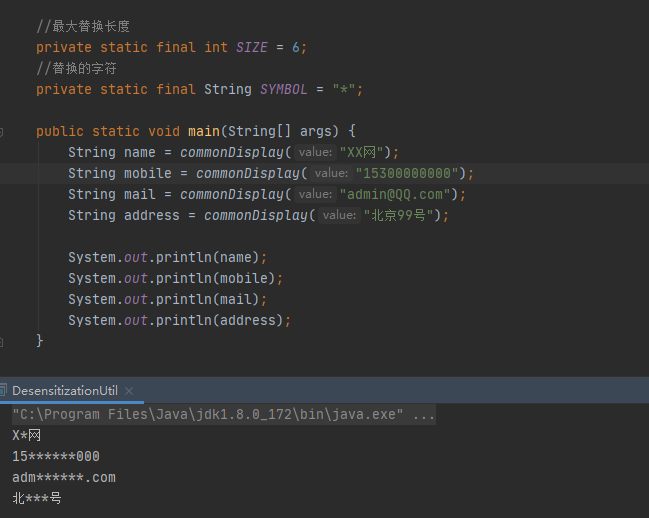


|  |
| --- |
| public PagedGridResult queryPagedComments(String itemId,  Integer level,Integer page, Integer pageSize) {  Map<String, Object> map = new HashMap<>();  map.put("itemId", itemId);  map.put("level", level);  // mybatis-pagehelper 先去总pom.xml配置 ，然后在去api 的yml文件配置 =》# 分页插件配置  /\*\*  \* page: 第几页  \* pageSize: 每页显示条数  \*/  PageHelper.startPage(page, pageSize);  List<ItemCommentVO> list = itemsMapperCustom.queryItemComments(map);  return setterPagedGrid(list,page);  }  //List<?> 是不确定类型  private PagedGridResult setterPagedGrid(List<?> list, Integer page) {  PageInfo<?> pageList = new PageInfo<>(list);  PagedGridResult grid = new PagedGridResult();  grid.setPage(page);  grid.setRows(list);  grid.setTotal(pageList.getPages());  grid.setRecords(pageList.getTotal());  return grid;  } |

### mybatis嵌套查询

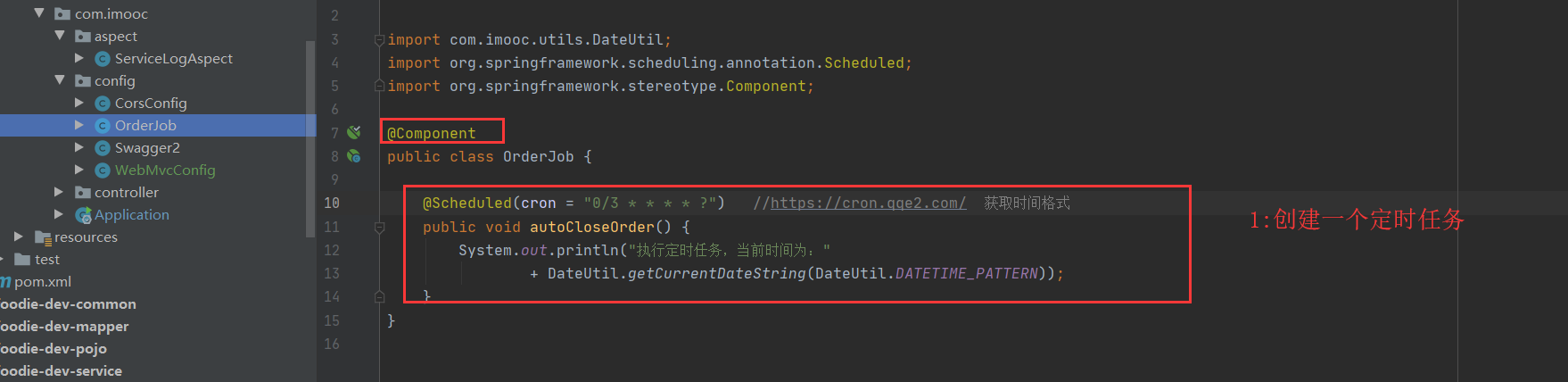


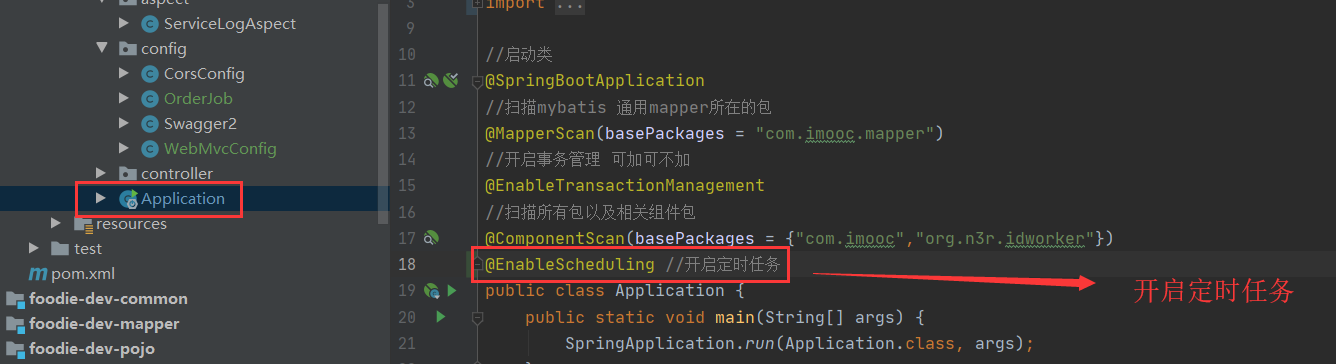
## 数据脱敏



|  |
| --- |
| /\*\*  \* 通用脱敏方法  \* @param value  \* @return  \*/  public static String commonDisplay(String value) {  if (null == value || "".equals(value)) {  return value;  }  int len = value.length();  int pamaone = len / 2;  int pamatwo = pamaone - 1;  int pamathree = len % 2;  StringBuilder stringBuilder = new StringBuilder();  if (len <= 2) {  if (pamathree == 1) {  return SYMBOL;  }  stringBuilder.append(SYMBOL);  stringBuilder.append(value.charAt(len - 1));  } else {  if (pamatwo <= 0) {  stringBuilder.append(value.substring(0, 1));  stringBuilder.append(SYMBOL);  stringBuilder.append(value.substring(len - 1, len));  } else if (pamatwo >= SIZE / 2 && SIZE + 1 != len) {  int pamafive = (len - SIZE) / 2;  stringBuilder.append(value.substring(0, pamafive));  for (int i = 0; i < SIZE; i++) {  stringBuilder.append(SYMBOL);  }  if ((pamathree == 0 && SIZE / 2 == 0) || (pamathree != 0 && SIZE % 2 != 0)) {  stringBuilder.append(value.substring(len - pamafive, len));  } else {  stringBuilder.append(value.substring(len - (pamafive + 1), len));  }  } else {  int pamafour = len - 2;  stringBuilder.append(value.substring(0, 1));  for (int i = 0; i < pamafour; i++) {  stringBuilder.append(SYMBOL);  }  stringBuilder.append(value.substring(len - 1, len));  }  }  return stringBuilder.toString();  } |

## 定时任务[@EnableScheduling ]





当需要在定时任务中执行



定时任务一般会有问题

* 会有时间差，程序不严谨
* 不支持集群
* 会对数据库全表搜索，及其影响数据库性能：select \* from order where orderStatus = 10;
* 定时任务，仅仅只适用于小型轻量级项目，传统项目
* 解决方案：消息队列：MQ-> RabbitMQ, RocketMQ, Kafka, ZeroMQ...

## mybatis中#{}和${}

1. #将传入的数据都当成一个字符串，会对自动传入的数据加一个双引号。如：order by #user\_id#，如果传入的值是111,那么解析成sql时的值为order by "111", 如果传入的值是id，则解析成的sql为order by "id".

2. $将传入的数据直接显示生成在sql中。如：order by $user\_id$，如果传入的值是111,那么解析成sql时的值为order by user\_id, 如果传入的值是id，则解析成的sql为order by id.

3. #方式能够很大程度防止sql注入。

4.$方式无法防止Sql注入。

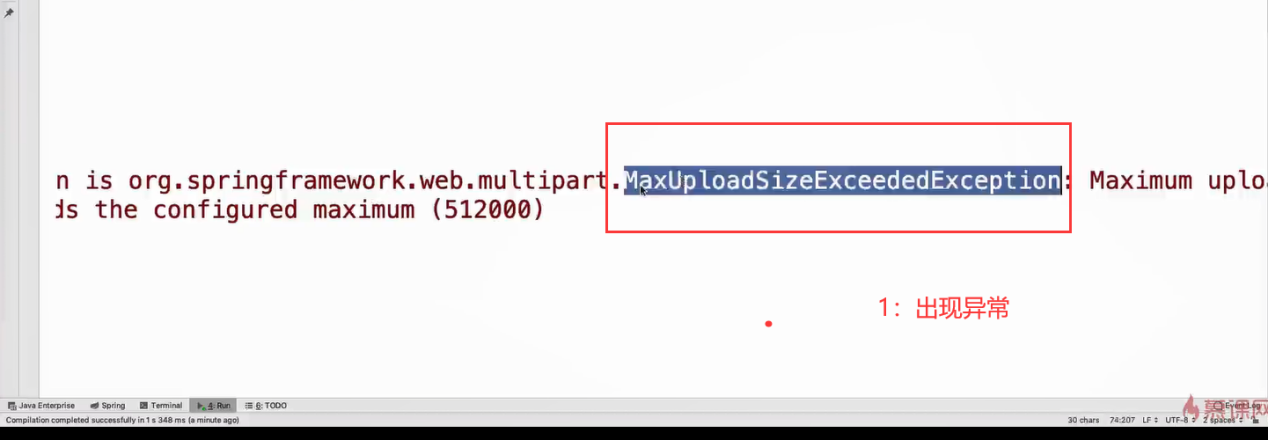
5.$方式一般用于传入数据库对象，例如传入表名.

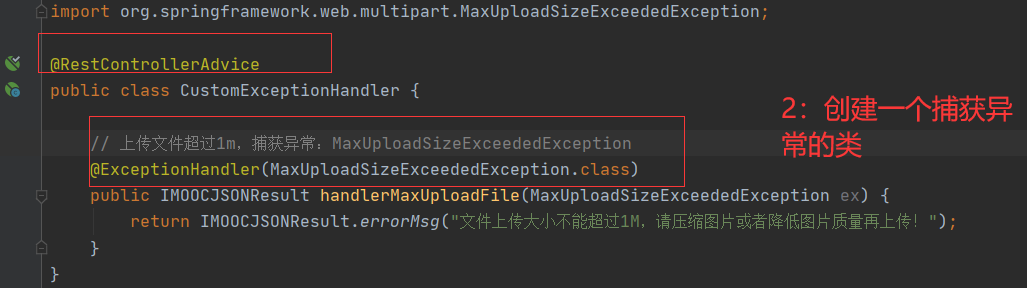
6.一般能用#的就别用$.

MyBatis排序时使用order by 动态参数时需要注意，用$而不是#

## springBoot 全局异常方式处理自定义异常 @RestControllerAdvice + @ExceptionHandler

* @RestControllerAdvice
* @ExceptionHandler





|  |
| --- |
| @RestControllerAdvice  public class CustomExceptionHandler {  // 上传文件超过1m，捕获异常：MaxUploadSizeExceededException  @ExceptionHandler(MaxUploadSizeExceededException.class)  public IMOOCJSONResult handlerMaxUploadFile(MaxUploadSizeExceededException ex) {  return IMOOCJSONResult.errorMsg("文件上传大小不能超过1M，请压缩图片或者降低图片质量再上传！");  }  } |

# Spring boot 注解

## 使用注解的优势：

     1.采用纯java代码，不在需要配置繁杂的xml文件

     2.在配置中也可享受面向对象带来的好处

     3.类型安全对重构可以提供良好的支持

     4.减少复杂配置文件的同时亦能享受到springIoC容器提供的功能

## ****注解详解（配备了完善的释义）------(可采用ctrl+F 来进行搜索哦~~~~)****

@SpringBootApplication：申明让spring boot自动给程序进行必要的配置，这个配置等同于：

@Configuration ，@EnableAutoConfiguration 和 @ComponentScan 三个配置。

@ResponseBody：表示该方法的返回结果直接写入HTTP response body中，一般在异步获取数据时使用，用于构建RESTful的api。在使用@RequestMapping后，返回值通常解析为跳转路径，加上@esponsebody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。比如异步获取json数据，加上@Responsebody后，会直接返回json数据。该注解一般会配合@RequestMapping一起使用。

@Controller：用于定义控制器类，在spring项目中由控制器负责将用户发来的URL请求转发到对应的服务接口（service层），一般这个注解在类中，通常方法需要配合注解@RequestMapping。

@RestController：用于标注控制层组件(如struts中的action)，@ResponseBody和@Controller的合集。

@RequestMapping：提供路由信息，负责URL到Controller中的具体函数的映射。

@EnableAutoConfiguration：SpringBoot自动配置（auto-configuration）：尝试根据你添加的jar依赖自动配置你的Spring应用。例如，如果你的classpath下存在HSQLDB，并且你没有手动配置任何数据库连接beans，那么我们将自动配置一个内存型（in-memory）数据库”。你可以将@EnableAutoConfiguration或者@SpringBootApplication注解添加到一个@Configuration类上来选择自动配置。如果发现应用了你不想要的特定自动配置类，你可以使用@EnableAutoConfiguration注解的排除属性来禁用它们。

@ComponentScan：表示将该类自动发现扫描组件。个人理解相当于，如果扫描到有@Component、@Controller、@Service等这些注解的类，并注册为Bean，可以自动收集所有的Spring组件，包括@Configuration类。我们经常使用@ComponentScan注解搜索beans，并结合@Autowired注解导入。可以自动收集所有的Spring组件，包括@Configuration类。我们经常使用@ComponentScan注解搜索beans，并结合@Autowired注解导入。如果没有配置的话，Spring Boot会扫描启动类所在包下以及子包下的使用了@Service,@Repository等注解的类。

@Configuration：相当于传统的xml配置文件，如果有些第三方库需要用到xml文件，建议仍然通过@Configuration类作为项目的配置主类——可以使用@ImportResource注解加载xml配置文件。

@Import：用来导入其他配置类。

@ImportResource：用来加载xml配置文件。

@Autowired：自动导入依赖的bean

@Service：一般用于修饰service层的组件

@Repository：使用@Repository注解可以确保DAO或者repositories提供异常转译，这个注解修饰的DAO或者repositories类会被ComponetScan发现并配置，同时也不需要为它们提供XML配置项。

@Bean：用@Bean标注方法等价于XML中配置的bean。

@Value：注入Spring boot application.properties配置的属性的值。示例代码：

@Inject：等价于默认的@Autowired，只是没有required属性；

@Component：泛指组件，当组件不好归类的时候，我们可以使用这个注解进行标注。

@Bean:相当于XML中的,放在方法的上面，而不是类，意思是产生一个bean,并交给spring管理。

@AutoWired：自动导入依赖的bean。byType方式。把配置好的Bean拿来用，完成属性、方法的组装，它可以对类成员变量、方法及构造函数进行标注，完成自动装配的工作。当加上（required=false）时，就算找不到bean也不报错。

@Qualifier：当有多个同一类型的Bean时，可以用@Qualifier(“name”)来指定。与@Autowired配合使用。@Qualifier限定描述符除了能根据名字进行注入，但能进行更细粒度的控制如何选择候选者，具体使用方式如下：

@Resource(name=”name”,type=”type”)：没有括号内内容的话，默认byName。与@Autowired干类似的事。

## ****注解列表如下****

@SpringBootApplication：包含了@ComponentScan、@Configuration和@EnableAutoConfiguration注解。其中

@ComponentScan：让spring Boot扫描到Configuration类并把它加入到程序上下文。

@Configuration ：等同于spring的XML配置文件；使用Java代码可以检查类型安全。

@EnableAutoConfiguration ：自动配置。

@ComponentScan ：组件扫描，可自动发现和装配一些Bean。

@Component可配合CommandLineRunner使用，在程序启动后执行一些基础任务。

@RestController：注解是@Controller和@ResponseBody的合集,表示这是个控制器bean,并且是将函数的返回值直 接填入HTTP响应体中,是REST风格的控制器。

@Autowired：自动导入。

@PathVariable：获取参数。

@JsonBackReference：解决嵌套外链问题。

@RepositoryRestResourcepublic：配合spring-boot-starter-data-rest使用。

## ****JPA注解****

@Entity：@Table(name=”“)：表明这是一个实体类。一般用于jpa这两个注解一般一块使用，但是如果表名和实体类名相同的话，@Table可以省略

@MappedSuperClass:用在确定是父类的entity上。父类的属性子类可以继承。

@NoRepositoryBean:一般用作父类的repository，有这个注解，spring不会去实例化该repository。

@Column：如果字段名与列名相同，则可以省略。

@Id：表示该属性为主键。

@GeneratedValue(strategy = GenerationType.SEQUENCE,generator = “repair\_seq”)：表示主键生成策略是sequence（可以为Auto、IDENTITY、native等，Auto表示可在多个数据库间切换），指定sequence的名字是repair\_seq。

@SequenceGeneretor(name = “repair\_seq”, sequenceName = “seq\_repair”, allocationSize = 1)：name为sequence的名称，以便使用，sequenceName为数据库的sequence名称，两个名称可以一致。

@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。@Basic(fetch=FetchType.LAZY)：标记可以指定实体属性的加载方式

@JsonIgnore：作用是json序列化时将[Java](http://lib.csdn.net/base/java)bean中的一些属性忽略掉,序列化和反序列化都受影响。

@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。

@OneToOne、@OneToMany、@ManyToOne：对应[hibernate](http://lib.csdn.net/base/javaee)配置文件中的一对一，一对多，多对一。

## ****springMVC相关注解****

@RequestMapping：@RequestMapping(“/path”)表示该控制器处理所有“/path”的UR L请求。RequestMapping是一个用来处理请求地址映射的注解，可用于类或方法上。  
用于类上，表示类中的所有响应请求的方法都是以该地址作为父路径。该注解有六个属性：  
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，指定的地址可以是URI Template 模式  
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

@RequestParam：用在方法的参数前面。  
@RequestParam  
String a =request.getParameter(“a”)。

@PathVariable:路径变量。如

参数与大括号里的名字一样要相同。

## ****全局异常处理****

@ControllerAdvice：包含@Component。可以被扫描到。统一处理异常。

@ExceptionHandler（Exception.class）：用在方法上面表示遇到这个异常就执行以下方法。

## ****项目中具体配置解析和使用环境****

****@MappedSuperclass：****

1.@MappedSuperclass 注解使用在父类上面，是用来标识父类的

2.@MappedSuperclass 标识的类表示其不能映射到数据库表，因为其不是一个完整的实体类，但是它所拥有的属性能够映射在其子类对用的数据库表中

3.@MappedSuperclass 标识的类不能再有@Entity或@Table注解

****@Column：****

1.当实体的属性与其映射的数据库表的列不同名时需要使用@Column标注说明，该属性通常置于实体的属性声明语句之前，还可与 @Id 标注一起使用。

2.@Column 标注的常用属性是name，用于设置映射数据库表的列名。此外，该标注还包含其它多个属性，如：unique、nullable、length、precision等。具体如下：

  1 name属性：name属性定义了被标注字段在数据库表中所对应字段的名称

  2 unique属性：unique属性表示该字段是否为唯一标识，默认为false，如果表中有一个字段需要唯一标识，则既可以使用该标记，也可以使用@Table注解中的@UniqueConstraint

  3 nullable属性：nullable属性表示该字段是否可以为null值，默认为true

  4 insertable属性：insertable属性表示在使用”INSERT”语句插入数据时，是否需要插入该字段的值

  5 updateable属性：updateable属性表示在使用”UPDATE”语句插入数据时，是否需要更新该字段的值

  6 insertable和updateable属性：一般多用于只读的属性，例如主键和外键等，这些字段通常是自动生成的

  7 columnDefinition属性：columnDefinition属性表示创建表时，该字段创建的SQL语句，一般用于通过Entity生成表定义时使用，如果数据库中表已经建好，该属性没有必要使用

  8 table属性：table属性定义了包含当前字段的表名

  9 length属性：length属性表示字段的长度，当字段的类型为varchar时，该属性才有效，默认为255个字符

 10 precision属性和scale属性：precision属性和scale属性一起表示精度，当字段类型为double时，precision表示数值的总长度，scale表示小数点所占的位数

    具体如下：  
   1.double类型将在数据库中映射为double类型，precision和scale属性无效  
   2.double类型若在columnDefinition属性中指定数字类型为decimal并指定精度，则最终以columnDefinition为准  
   3.BigDecimal类型在数据库中映射为decimal类型，precision和scale属性有效  
   4.precision和scale属性只在BigDecimal类型中有效

3.@Column 标注的columnDefinition属性: 表示该字段在数据库中的实际类型.通常 ORM 框架可以根据属性类型自动判断数据库中字段的类型,但是对于Date类型仍无法确定数据库中字段类型究竟是DATE,TIME还是TIMESTAMP.此外,String的默认映射类型为VARCHAR,如果要将 String 类型映射到特定数据库的 BLOB 或TEXT字段类型.

4.@Column标注也可置于属性的getter方法之前

@Getter和@Setter（Lombok）

@Setter：注解在属性上；为属性提供 setting 方法 @Getter：注解在属性上；为属性提供 getting 方法

  1 @Data：注解在类上；提供类所有属性的 getting 和 setting 方法，此外还提供了equals、canEqual、hashCode、toString 方法  
  2 @Setter：注解在属性上；为属性提供 setting 方法   
  3 @Getter：注解在属性上；为属性提供 getting 方法   
  4 @Log4j2 ：注解在类上；为类提供一个 属性名为log 的 log4j 日志对象，和@Log4j注解类似   
  5 @NoArgsConstructor：注解在类上；为类提供一个无参的构造方法  
 6 @AllArgsConstructor：注解在类上；为类提供一个全参的构造方法   
 7 @EqualsAndHashCode:默认情况下，会使用所有非瞬态(non-transient)和非静态(non-static)字段来生成equals和hascode方法，也可以指定具体使用哪些属性。  
 8 @toString:生成toString方法，默认情况下，会输出类名、所有属性，属性会按照顺序输出，以逗号分割。   
 9 @NoArgsConstructor, @RequiredArgsConstructor and @AllArgsConstructor  
 10 无参构造器、部分参数构造器、全参构造器，当我们需要重载多个构造器的时候，只能自己手写了   
 11 @NonNull：注解在属性上，如果注解了，就必须不能为Null   
 12 @val:注解在属性上，如果注解了，就是设置为final类型，可查看源码的注释知道

当你在执行各种持久化方法的时候，实体的状态会随之改变，状态的改变会引发不同的生命周期事件。这些事件可以使用不同的注释符来指示发生时的回调函数。

@javax.persistence.PostLoad：加载后。

@javax.persistence.PrePersist：持久化前。

@javax.persistence.PostPersist：持久化后。

@javax.persistence.PreUpdate：更新前。

@javax.persistence.PostUpdate：更新后。

@javax.persistence.PreRemove：删除前。

@javax.persistence.PostRemove：删除后。

****1）数据库查询****

@PostLoad事件在下列情况下触发：

执行EntityManager.find()或getreference()方法载入一个实体后。

执行JPQL查询后。

EntityManager.refresh()方法被调用后。

****2）数据库插入****

@PrePersist和@PostPersist事件在实体对象插入到数据库的过程中发生：

@PrePersist事件在调用persist()方法后立刻发生，此时的数据还没有真正插入进数据库。

@PostPersist事件在数据已经插入进数据库后发生。

****3）数据库更新****

@PreUpdate和@PostUpdate事件的触发由更新实体引起：

@PreUpdate事件在实体的状态同步到数据库之前触发，此时的数据还没有真正更新到数据库。

@PostUpdate事件在实体的状态同步到数据库之后触发，同步在事务提交时发生。

****4）数据库删除****

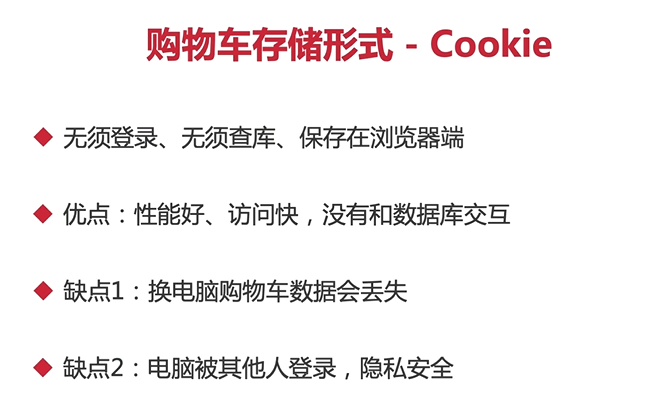
@PreRemove和@PostRemove事件的触发由删除实体引起：

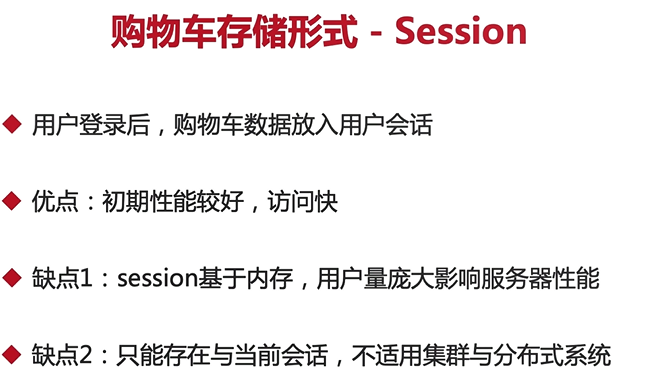
@PreRemove事件在实体从数据库删除之前触发，即在调用remove()方法删除时发生，此时的数据还没有真正从数据库中删除。

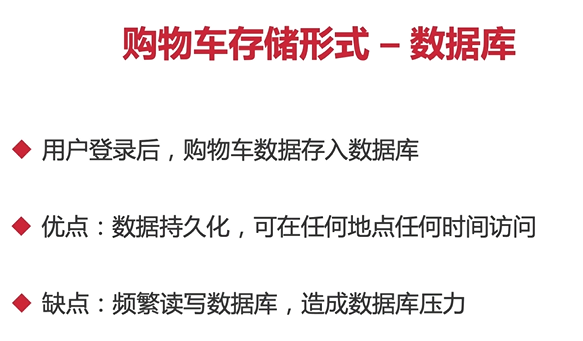
@PostRemove事件在实体从数据库中删除后触发。

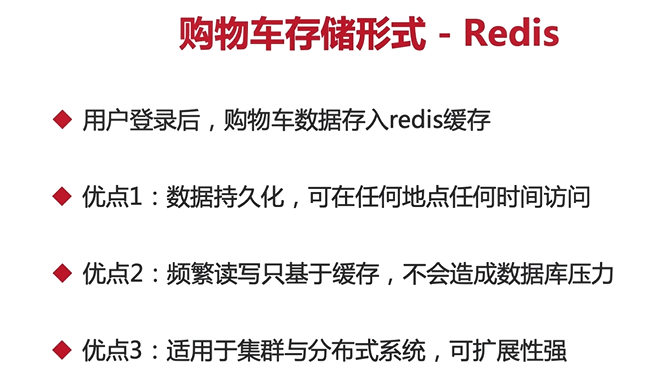
# 小拓展

## 购物车存储形式









# 看到

