# 购物实现

## 商品数量修改

### 406报错

特点:页面解析异常,页面要求解析的是.html页面,但是回传的数据是JSON串.这时解析发生异常.报错406

解决:

<servlet-mapping>

<servlet-name>springmvc-web</servlet-name>

<url-pattern>\*.html</url-pattern>

</servlet-mapping>

<!-- 防止springMVC框架返回json时和html冲突报 406 错误 -->

<servlet-mapping>

<servlet-name>springmvc-web</servlet-name>

<url-pattern>/service/\*</url-pattern>

</servlet-mapping>

总结:

当请求需要跳转页面时后缀以.html结尾.当发起js/ajax请求时需要以/service开启.

上述的配置为了实现伪静态的操作.增强搜索引擎的友好性.

### 页面url

1.页面url



2.页面js

$(".increment").click(**function**(){//＋

**var** \_thisInput = $(**this**).siblings("input");

\_thisInput.val(eval(\_thisInput.val()) + 1);

//$.post("/service/cart/update/num/"+\_thisInput.attr("itemId")+"/"+\_thisInput.val(),function(data){

//模拟错误请求 上行注释是正确的 **$.post("/cart/update/num/"+\_thisInput.attr("itemId")+"/"+\_thisInput.val()+".html",function(data){**

TTCart.refreshTotalPrice();

});

});

### 编辑前台Controller

//购物数量的修改

@RequestMapping("/update/num/{itemId}/{num}")

@ResponseBody

**public** SysResult updateCartNum(

@PathVariable Long itemId,

@PathVariable Integer num){

**try** {

Cart cart = **new** Cart();

cart.setUserId(7L);

cart.setItemId(itemId);

cart.setNum(num);

cartService.updateCartNum(cart);

**return** SysResult.*oK*();

} **catch** (Exception e) {

e.printStackTrace();

}

**return** SysResult.*build*(201,"商品数量更新失败");

}

### 编辑前台Service

@Override

**public** **void** updateCartNum(Cart cart) {

//1.定义url

String url = "http://cart.jt.com/cart/update/num";

//2.封装参数

**try** {

String cartJSON = *objectMapper*.writeValueAsString(cart);

Map<String,String> params = **new** HashMap<>();

params.put("cartJSON", cartJSON);

httpClient.doGet(url, params);

} **catch** (Exception e) {

e.printStackTrace();

**throw** **new** RuntimeException();

}

}

### 编辑后台Controller

@RequestMapping("/update/num")

@ResponseBody

**public** SysResult updateCartNum(String cartJSON){

**try** {

Cart cart = *objectMapper*.readValue(cartJSON, Cart.**class**);

cartService.updateCartNum(cart);

**return** SysResult.*oK*();

} **catch** (Exception e) {

e.printStackTrace();

}

**return** SysResult.*build*(201,"商品更新失败");

}

### 编辑后台Service

@Override

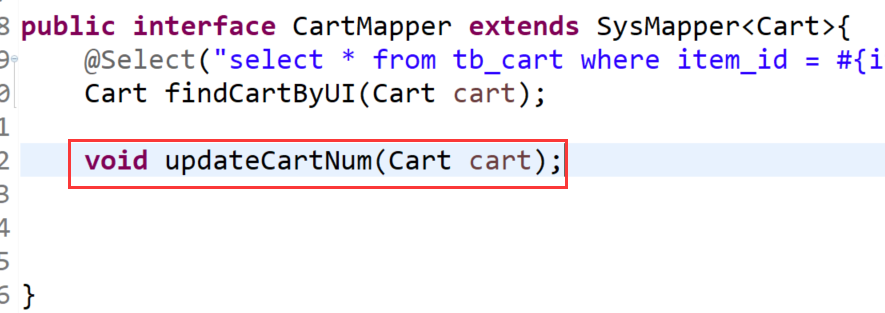
**public** **void** updateCartNum(Cart cart) {

cart.setUpdated(**new** Date());

cartMapper.updateCartNum(cart);

}

### 编辑后台Mapper



<!--修改购物车数量 -->

<update id=*"updateCartNum"*>

update tb\_cart set num = #{num}, updated = #{updated}

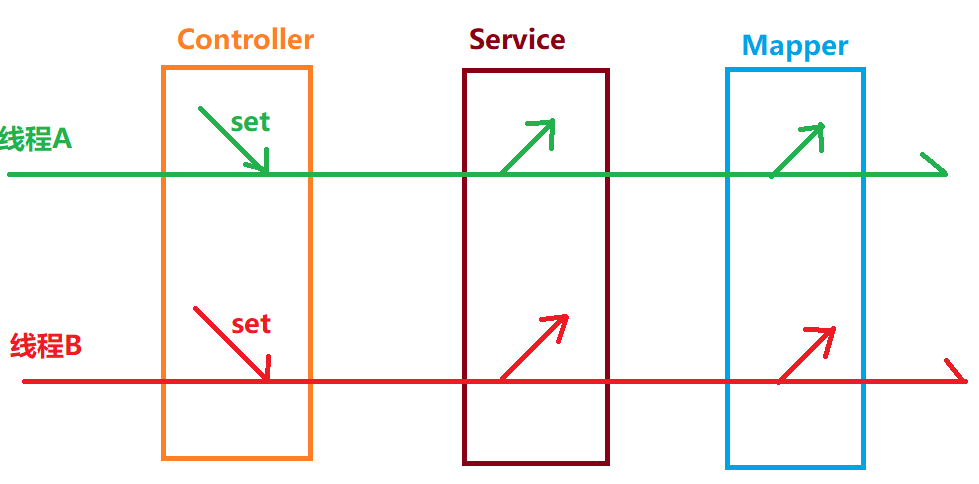
where item\_id = #{itemId} and user\_id = #{userId}

</update>

## ThreadLocal(本地线程变量)

### 功能介绍

可以在一个线程内,显示线程数据的共享.



### 编辑工具类

**public** **class** UserThreadLocal {

**private** **static** ThreadLocal<User> *userThread* = **new** ThreadLocal<>();

**public** **static** **void** set(User user){

*userThread*.set(user);

}

**public** **static** User get(){

**return** *userThread*.get();

}

//由于gc没有权限回收ThreadLocal.所以调用ThtredLocal时需要

//额外的注意内存泄漏问题

**public** **static** **void** remove(){

*userThread*.remove();

}

}

## 拦截器实现权限控制

### 业务需求

说明:如果用户没有登陆,则点击购物车时应该实现拦截,并且跳转到用户的登陆页面.如果用户已经登陆了,则予以放行.

### 拦截器实现

**public** **class** UserInterceptor **implements** HandlerInterceptor{

@Autowired

**private** JedisCluster jedisCluster;

**private** **static** ObjectMapper *objectMapper* = **new** ObjectMapper();

/\*\*

\* 处理器执行之前 校验用户是否登录

\* 1.通过request对象先获取cookie

\* 2.获取token数据

\* 3.从redis集群中获取userJSON数据

\* 4.将userJSON串转化为user对象

\* 5.从对象中动态获取id

\* 6.如果上述业务中token/userJSON为null,都要进行拦截

\* 跳转到系统登录页面.

\*

\* 7.如何动态实现数据获取

\* 如果需要在程序的任何地方都能动态获取user对象???

\* 1.利用request对象 将数据保存到session域中

\* 利用该对象只能通过controller传参的方式传入user对象

\* 2.

\*/

@Override

**public** **boolean** preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)

**throws** Exception {

Cookie[] cookies = request.getCookies();

String token = **null**;

**for** (Cookie cookie : cookies) {

**if**("JT\_TICKET".equals(cookie.getName())){

token = cookie.getValue();

**break**;

}

}

**if**(!StringUtils.*isEmpty*(token)){

String userJSON = jedisCluster.get(token);

**if**(!StringUtils.*isEmpty*(userJSON)){

User user = *objectMapper*.readValue(userJSON, User.**class**);

//动态获取Id稍后完成

/\*request.getSession()

.setAttribute("JT\_USER",user);\*/

//利用ThreadLocal实现数据共享

UserThreadLocal.*set*(user);

//当前用户已经登陆,true表示拦截器放行

**return** **true**;

}

}

//重定向到登陆页面

response.sendRedirect("/user/login.html");

**return** **false**; //表示拦截

}

//处理器执行完之后

@Override

**public** **void** postHandle(HttpServletRequest request, HttpServletResponse response, Object handler,

ModelAndView modelAndView) **throws** Exception {

// **TODO** Auto-generated method stub

}

//用户展现数据之前

@Override

**public** **void** afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex)

**throws** Exception {

UserThreadLocal.*remove*();

}

}

### 拦截器配置

<!--配置拦截器

Content Model :

/\* 拦截请求之后的一级路径

/\*\* 拦截请求之后的所有路径

mapping 拦截的请求路径 /cart/\*\*

exclude-mapping\* 不拦截某些特点的请求路径

(bean | ref)) 拦截器的处理类

-->

<mvc:interceptors>

<mvc:interceptor>

<mvc:mapping path=*"/cart/\*\*"*/>

<bean class=*"com.jt.web.intercept.UserInterceptor"*/>

</mvc:interceptor>

</mvc:interceptors>

# 订单业务实现

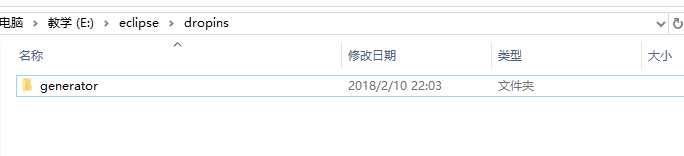
## 代码自动生成工具

### 作用

可以自动的根据指定的表,动态的生成pojo对象(set/get方法)/Mapper接口/Mapper映射文件

### 引入插件

引入插件后,重启eclipse



### 引入配置文件

#是否不生成注释 true 不生成 false 生成

suppressAllComments=true

#数据库链接

driverClass=com.mysql.jdbc.Driver

url=jdbc:mysql://localhost:3306/jtdb?characterEncoding=utf-8

username=root

password=root

#将来生成的代码存入那个项目中

targetProject=jt-order

#pojo所在包路径

modeltargetPackage=com.jt.order.pojo

#映射文件所在的包路径

sqltargetPackage=com.jt.order.mapper

#mybaits接口包路径

clienttargetPackage=com.jt.order.mapper

### 编辑xml配置文件

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

<!DOCTYPE generatorConfiguration PUBLIC "-//mybatis.org//DTD MyBatis Generator Configuration 1.0//EN" "http://mybatis.org/dtd/mybatis-generator-config\_1\_0.dtd" >

<generatorConfiguration>

<properties resource=*"generatorConfig.properties"*/>

<!--本地数据库驱动包 -->

<classPathEntry location=*"E:\WorkJarSource\connDriver\mysql-connector-java-5.1.10-bin.jar"* />

<context id=*"tarena"*>

<commentGenerator>

<property name=*"suppressAllComments"* value=*"${suppressAllComments}"*/>

</commentGenerator>

<jdbcConnection driverClass=*"${driverClass}"* connectionURL=*"${url}"* userId=*"${username}"* password=*"${password}"* />

<javaModelGenerator targetPackage=*"${modeltargetPackage}"* targetProject=*"${targetProject}"* />

<sqlMapGenerator targetPackage=*"${sqltargetPackage}"* targetProject=*"${targetProject}"* />

<javaClientGenerator targetPackage=*"${clienttargetPackage}"* targetProject=*"${targetProject}"* type=*"XMLMAPPER"* />

<!--配置表名 -->

<table tableName=*"tb\_order"* domainObjectName=*"Order"* enableCountByExample=*"false"* enableUpdateByExample=*"false"*

enableDeleteByExample=*"false"* enableSelectByExample=*"false"* selectByExampleQueryId=*"false"*/>

<table tableName=*"tb\_order\_item"* domainObjectName=*"OrderItem"* enableCountByExample=*"false"* enableUpdateByExample=*"false"*

enableDeleteByExample=*"false"* enableSelectByExample=*"false"* selectByExampleQueryId=*"false"*/>

<table tableName=*"tb\_order\_shipping"* domainObjectName=*"OrderShipping"* enableCountByExample=*"false"* enableUpdateByExample=*"false"*

enableDeleteByExample=*"false"* enableSelectByExample=*"false"* selectByExampleQueryId=*"false"*/>

</context>

</generatorConfiguration>

## 创建订单项目

### 搭建项目

1. 修改nginx

server {

listen 80;

server\_name order.jt.com;

location / {

proxy\_pass http://localhost:8095;

proxy\_connect\_timeout 3;

proxy\_read\_timeout 3;

proxy\_send\_timeout 3;

}

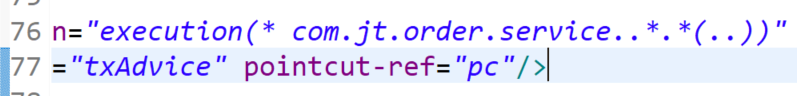
}

### 添加配置文件

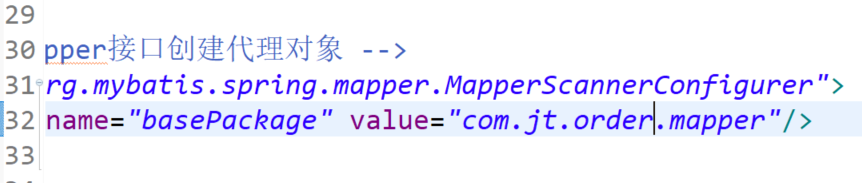
1. 修改web.xml



1. 修改spring配置文件

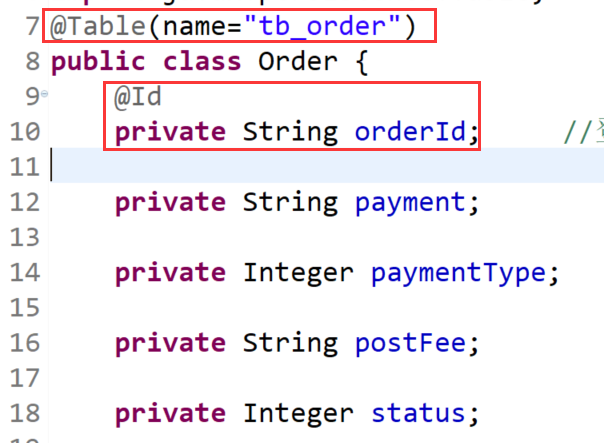


1. 修改mybatis配置文件

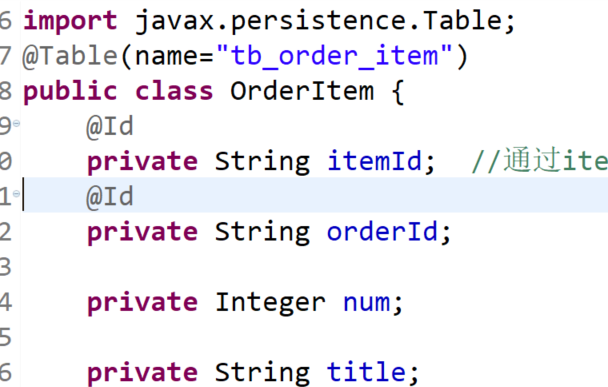


### 编辑pojo对象

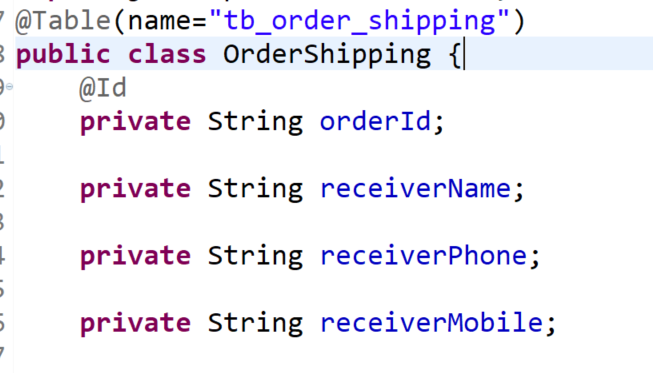
1.定义Order



1. 定义OrderItem对象



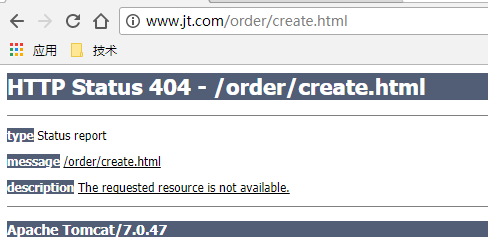
1. 定义OrderShipping对象



## 订单确认页面跳转

### 页面分析

1. 页面



### 编辑Controller

/\*\*

\* //实现订单确认页面跳转

\* 1.根据当前用户的Id查询购物车数据

\*/

@RequestMapping("/create")

**public** String create(Model model){

Long userId = UserThreadLocal.*get*().getId();

List<Cart> cartList =

cartService.findCartByUserId(userId);

model.addAttribute("carts", cartList);

**return** "order-cart";

}

### 修改拦截器路径

<mvc:interceptors>

<mvc:interceptor>

<mvc:mapping path=*"/cart/\*\*"*/>

**<mvc:mapping path=*"/order/\*\*"*/>**

<bean class=*"com.jt.web.intercept.UserInterceptor"*/>

</mvc:interceptor>

</mvc:interceptors>

### 页面效果



## 订单提交

### SpringMVC传参过程

1. 入门级

页面

<input type=”text” Name=id value=”100”/>

Name=username value=”tom”

Mvc

Public String add(int id,String username){…}

特点:

直接给参数赋值

1. 进阶级

页面

<input type=”text” Name=id value=”100”/>\*N 很多项

MVC:

Public String add(User user){…}

**特点: 采用User对象进行参数接收.为对象赋值**

1. 高级

<input type=”text” name=”id” value=”100”/>

<input type=”text” name=”name” value=”哈利波特”/>

<input type=”text” name=”daoju.dName” value=”隐身袍子-澡堂专用”/>

类型定义

Class Harry{

Private integer id;

Private String name;

Private Daoju daoju; 对象的引用

}

Class Daoju{

Private Integer dId;

Private String dName;

}

MVC:

Public String addUser(Harry harry){…..}

**特点:可以为对象的引用类型的属性赋值.**

**实际案例:**

<form id=*"orderForm"* class=*"hide"*>

<input type=*"hidden"* name=*"paymentType"* value=*"1"*/>

<c:forEach items=*"*${carts}*"* var=*"cart"* varStatus=*"status"*>

<c:set var=*"totalPrice"* value=*"*${ totalPrice + (cart.itemPrice \* cart.num)}*"*/>

<input type=*"hidden"* name=*"orderItems[*${status.index}*].itemId"* value=*"*${cart.itemId}*"*/>

<input type=*"hidden"* name=*"orderItems[*${status.index}*].num"* value=*"*${cart.num }*"*/>

<input type=*"hidden"* name=*"orderItems[*${status.index}*].price"* value=*"*${cart.itemPrice}*"*/>

<input type=*"hidden"* name=*"orderItems[*${status.index}*].totalFee"* value=*"*${cart.itemPrice \* cart.num}*"*/>

<input type=*"hidden"* name=*"orderItems[*${status.index}*].title"* value=*"*${cart.itemTitle}*"*/>

<input type=*"hidden"* name=*"orderItems[*${status.index}*].picPath"* value=*"*${cart.itemImage}*"*/>

</c:forEach>

<input type=*"hidden"* name=*"payment"* value=*"*<fmt:formatNumber groupingUsed=*"false"* maxFractionDigits=*"2"* minFractionDigits=*"2"* value="${totalPrice/100 }"/>*"*/>

<input type=*"hidden"* name=*"orderShipping.receiverName"* value=*"陈晨"*/>

<input type=*"hidden"* name=*"orderShipping.receiverMobile"* value=*"13800807944"*/>

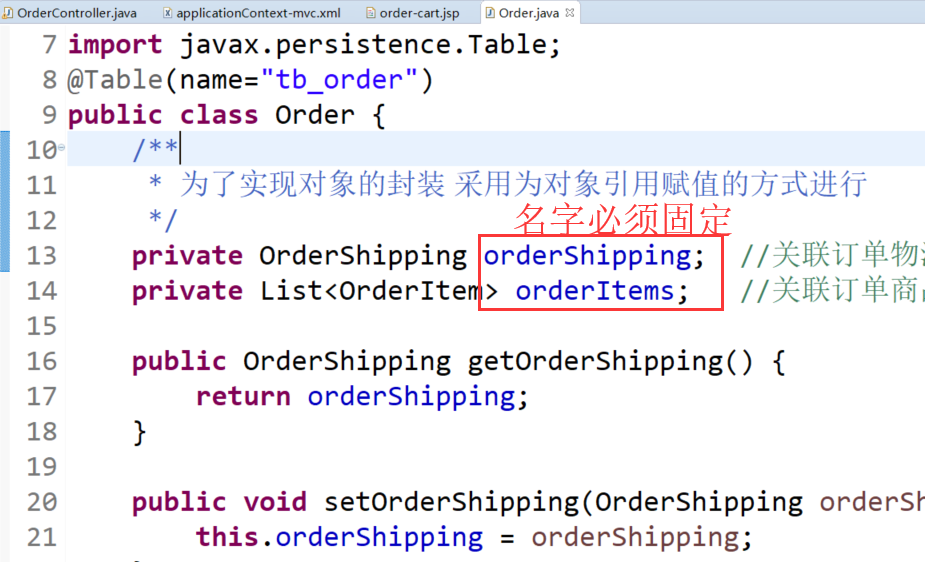
<input type=*"hidden"* name=*"orderShipping.receiverState"* value=*"北京"*/>

<input type=*"hidden"* name=*"orderShipping.receiverCity"* value=*"北京"*/>

<input type=*"hidden"* name=*"orderShipping.receiverDistrict"* value=*"海淀区"*/>

<input type=*"hidden"* name=*"orderShipping.receiverAddress"* value=*"清华大学"*/>

</form>

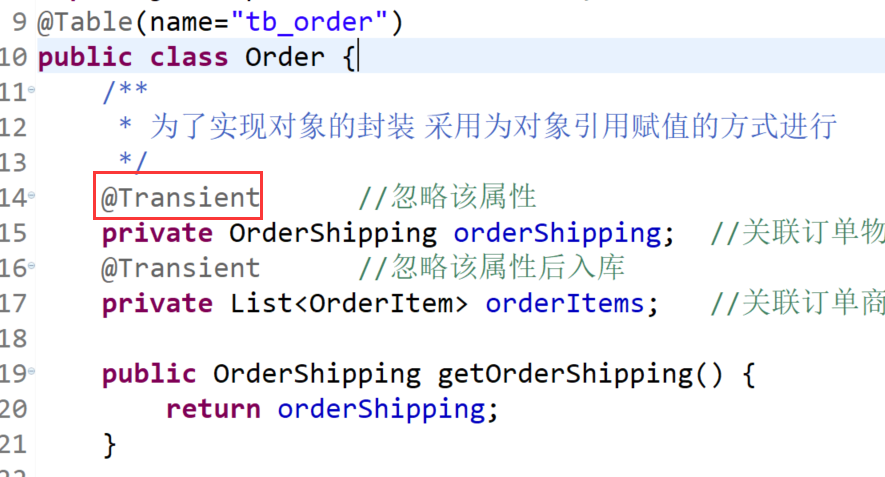


修改完成后,将common打包

### 忽略指定属性

说明:因为数据库操作,是基于通用Mapper的,一切通过反射自动入库.其中重要的依据就是对象中的属性.但是ordershipping和orderItems,是为了数据封装特意编辑.所以该属性不能参与数据库入库操作.

所以添加该注解



## 订单入库实现

### 页面分析

1. url



1. 页面JS

**function** submit\_Order() {

$("#submit\_message").hide();

jQuery.ajax( {

type : "POST",

dataType : "json",

url : "/service/order/submit",

data : $("#orderForm").serialize(),

cache : **false**,

success : **function**(result) {

**if**(result.status == 200){

location.href = "/order/success.html?id="+result.data;

}**else**{

$("#submit\_message").html("订单提交失败，请稍后重试...").show();

}

},

error : **function**(error) {

$("#submit\_message").html("亲爱的用户请不要频繁点击, 请稍后重试...").show();

}

});

}

### 编辑前台Controller

/\*\*

\* 实现订单入库

\*/

@RequestMapping("/submit")

@ResponseBody

**public** SysResult saveOrder(Order order){

**try** {

Long userId = UserThreadLocal.*get*().getId();

order.setUserId(userId);

String orderId = orderService.saveOrder(order);

**if**(!StringUtils.*isEmpty*(orderId)){

**return** SysResult.*oK*(orderId);

}

} **catch** (Exception e) {

e.printStackTrace();

}

**return** SysResult.*build*(201,"订单提交失败");

}

### 编辑前台Service

@Service

**public** **class** OrderServiceImpl **implements** OrderService {

@Autowired

**private** HttpClientService httpClient;

**private** **static** ObjectMapper *objectMapper* = **new** ObjectMapper();

@Override

**public** String saveOrder(Order order) {

String orderId = **null**;

//定义url

String url = "http://order.jt.com/order/create";

**try** {

String orderJSON =

*objectMapper*.writeValueAsString(order);

Map<String,String> params = **new** HashMap<>();

params.put("orderJSON", orderJSON);

String resultJSON = httpClient.doPost(url, params);

//将远程返回的数据转化为SysResult对象

SysResult sysResult =

*objectMapper*.readValue(resultJSON,SysResult.**class**);

//判断状态信息 200表示成功!!!

**if**(sysResult.getStatus() == 200){

orderId = (String) sysResult.getData();

}

} **catch** (Exception e) {

e.printStackTrace();

**throw** **new** RuntimeException();

}

**return** orderId;

}

}

### 编辑后台Controller

@Controller

@RequestMapping("/order")

**public** **class** OrderController {

@Autowired

**private** OrderService orderService;

**private** **static** ObjectMapper *objectMapper* = **new** ObjectMapper();

/\*\*

\* 实现订单入库操作

\* String url = "http://order.jt.com/order/create";

\*/

@RequestMapping("/create")

@ResponseBody

**public** SysResult saveOrder(String orderJSON){

**try** {

Order order = *objectMapper*.readValue(orderJSON,Order.**class**);

String orderId = orderService.saveOrder(order);

**return** SysResult.*oK*(orderId);

} **catch** (Exception e) {

e.printStackTrace();

}

**return** SysResult.*build*(201,"订单入库失败");

}

}

### 编辑后台Service

@Service

**public** **class** OrderServiceImpl **implements** OrderService {

@Autowired

**private** OrderMapper orderMapper;

@Autowired

**private** OrderItemMapper orderItemMapper;

@Autowired

**private** OrderShippingMapper orderShippingMapper;

/\*\*

\* 同时实现三张表入库操作

\*

\*/

@Override

**public** String saveOrder(Order order) {

String orderId = ""+ order.getUserId() + System.*currentTimeMillis*();

Date date = **new** Date();

//封装数据

order.setOrderId(orderId);

order.setStatus(1); //表示未付款

order.setCreated(date);

order.setUpdated(date);

orderMapper.insert(order);

System.***out***.println("订单入库成功!!!!!!");

//实现订单物流入库

OrderShipping shipping = order.getOrderShipping();

shipping.setOrderId(orderId);

shipping.setCreated(date);

shipping.setUpdated(date);

orderShippingMapper.insert(shipping);

System.***out***.println("订单物流入库成功!!!");

/\*\*

\* 订单商品入库

\* 了解:user(id,name,age)

\* insert into user(id,name,age)

\* values(1,"tom",18),(2,"jerry",17),(3,"jerry",18)

\* 作业:利用mybatis中循环标签实现入库

\*/

List<OrderItem> orderItemList = order.getOrderItems();

**for** (OrderItem orderItem : orderItemList) {

orderItem.setOrderId(orderId);

orderItem.setCreated(date);

orderItem.setUpdated(date);

orderItemMapper.insert(orderItem);

}

System.***out***.println("订单入库执行成功!!!!!");

**return** orderId;

}

}

## 订单查询

### 编辑前台Controller

//根据Id查询Order对象 ${order.orderId }

@RequestMapping("/success")

**public** String findOrderById(String id,Model model){

Order order = orderService.findOrderById(id);

model.addAttribute("order", order);

**return** "success";

}

### 编辑前台Service

@Override

**public** Order findOrderById(String id) {

String url = "http://order.jt.com/order/query/"+id;

String orderJSON = httpClient.doGet(url);

Order order = **null**;

**try** {

order = *objectMapper*.readValue(orderJSON, Order.**class**);

} **catch** (Exception e) {

e.printStackTrace();

}

**return** order;

}

### 编辑后台Controller

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

@ResponseBody //RPC dubbo

**public** Order findOrderById(@PathVariable String id){

**return** orderService.findOrderById(id);

}

### 编辑后台Service

@Override

**public** Order findOrderById(String id) {

Order order = orderMapper.selectByPrimaryKey(id);

OrderShipping orderShipping =

orderShippingMapper.selectByPrimaryKey(id);

OrderItem orderItem = **new** OrderItem();

orderItem.setOrderId(id);

List<OrderItem> orderItemList =

orderItemMapper.select(orderItem);

order.setOrderItems(orderItemList);

order.setOrderShipping(orderShipping);

**return** order;

}

## 定时任务

### 需求

因为有些用户提交订单后没有付款.等到超时时间一过.应该将订单的状态修改为6 交易关闭.

### Quartz介绍(石英钟定时任务)

说明:当任务(job)到了指定的时间,需要定时执行

Quartz是OpenSymphony开源组织在Job scheduling领域又一个开源项目，它可以与J2EE与J2SE应用程序相结合也可以单独使用。Quartz可以用来创建简单或为运行十个，百个，甚至是好几万个Jobs这样复杂的程序。Jobs可以做成标准的Java组件或 EJBs。Quartz的最新版本为Quartz 2.3.0。

1. 定时发短信.
2. 定时发邮件.

优点:

可以选择任何时间执行程序

同类型:

Timer,[定时器](https://baike.baidu.com/item/%E5%AE%9A%E6%97%B6%E5%99%A8/5109454)，功能是在指定的时间间隔内反复触发指定窗口的定时器事件[1]

功能单一

### 导入jar包

<!-- 定时任务 -->

<dependency>

<groupId>org.quartz-scheduler</groupId>

<artifactId>quartz</artifactId>

<version>2.2.1</version>

</dependency>

<!--定时任务需要依赖c3p0jar包 -->

<dependency>

<groupId>c3p0</groupId>

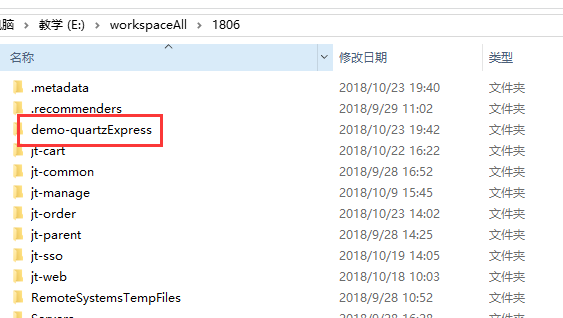
<artifactId>c3p0</artifactId>

<version>0.9.1.2</version>

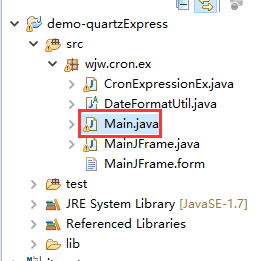
</dependency>

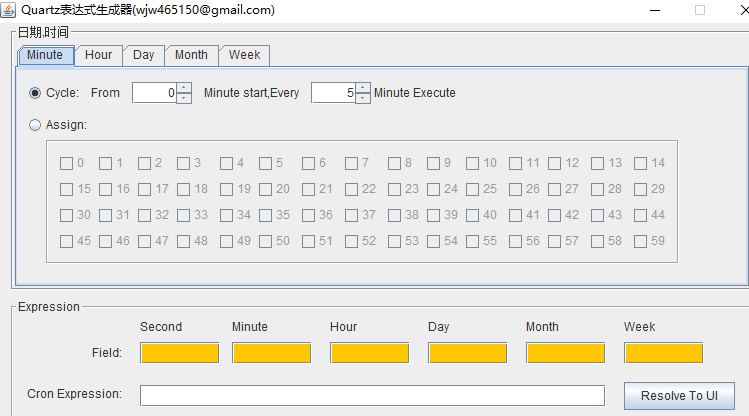
之后将jt-parent打包

### 导入表达式

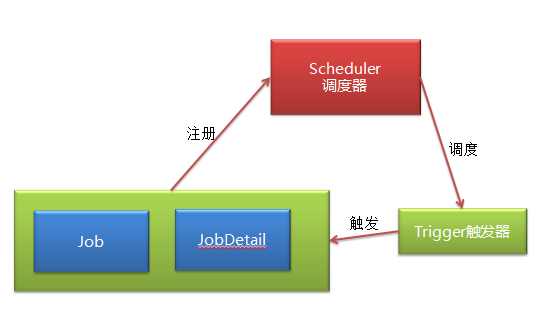


运行项目





### 定时任务调用原理



1. 调度器

监控所有的任务的执行.内部有时钟机制,当任务到了指定的执行时间,由调度器负责程序的运行.

1. Job

定义定时任务.

1. 触发器

当接收到调用器的任务处理指令时,会开启新的线程处理任务.

### 导入配置文件

<!-- 定义任务bean -->

<bean name=*"paymentOrderJobDetail"* class=*"org.springframework.scheduling.quartz.JobDetailFactoryBean"*>

<!-- 指定具体的job类 -->

<property name=*"jobClass"* value=*"com.jt.order.job.PaymentOrderJob"* />

<!-- 指定job的名称 -->

<property name=*"name"* value=*"paymentOrder"* />

<!-- 指定job的分组 -->

<property name=*"group"* value=*"Order"* />

<!-- 必须设置为true，如果为false，当没有活动的触发器与之关联时会在调度器中删除该任务 -->

<property name=*"durability"* value=*"true"*/>

<!-- 指定spring容器的key，如果不设定在job中的jobmap中是获取不到spring容器的 -->

<property name=*"applicationContextJobDataKey"* value=*"applicationContext"*/>

</bean>

<!-- 定义触发器 -->

<bean id=*"cronTrigger"* class=*"org.springframework.scheduling.quartz.CronTriggerFactoryBean"*>

<property name=*"jobDetail"* ref=*"paymentOrderJobDetail"* />

<!-- 每一分钟执行一次 -->

<property name=*"cronExpression"* value=*"0 0/1 \* \* \* ?"* />

</bean>

<!-- 定义调度器 -->

<bean class=*"org.springframework.scheduling.quartz.SchedulerFactoryBean"*>

<property name=*"triggers"*>

<list>

<ref bean=*"cronTrigger"* />

</list>

</property>

</bean>

### 定义执行任务

**public** **class** PaymentOrderJob **extends** QuartzJobBean{

/\*\*

\* 删除2天的恶意订单

\* 将状态信息 由1改为6

\* update tb\_order set status = 6

\* where status = 1 and created < (now-2天)

\*/

@Override

**protected** **void** executeInternal(JobExecutionContext context) **throws** JobExecutionException {

ApplicationContext applicationContext = (ApplicationContext) context.getJobDetail().getJobDataMap().get("applicationContext");

OrderMapper orderMapper =

applicationContext.getBean(OrderMapper.**class**);

Date dateAgo = **new** DateTime().minusDays(2).toDate();

orderMapper.updateStatus(dateAgo);

System.***out***.println("定时任务执行成功！~！！！！！");

}

}

### 检查数据库

检查状态是否修改为6

