# **购物车分析**

未登录状态操作购物车

登录状态操作购物车

合并购物车

## **购物车功能**

1.添加购物车商品

2.展示购物车列表页面

3.修改购物车商品数量

4.删除购物车商品

## **购物车设计**

1、用户未登录状态下：在不登陆的情况下把购物车信息写入cookie

​ 优点：

​ 1、不占用服务端存储空间

​ 2、代码实现简单。

​ 3、用户体验好

​ 缺点：

​ 1、cookie中保存的容量有限。最大4k

​ 2、把购物车信息保存在cookie中，更换设备购物车信息不能同步。

2、用户已登录状态下：把购物车信息保存到服务端的 Redis 中

​ 优点： 1、更换设备购物车信息可以同步

​ 缺点：1、占用服务端存储空间

### **创建usian\_cart\_service**

**Pom.xml**

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

</dependency>

<dependency>

<groupId>com.usian</groupId>

<artifactId>common\_redis</artifactId>

<version>1.0-SNAPSHOT</version>

</dependency>

<dependency>

<groupId>com.usian</groupId>

<artifactId>common\_pojo</artifactId>

<version>1.0-SNAPSHOT</version>

</dependency>

**Application.yml**

server:

port: 8100

eureka:

client:

service-url:

defaultZone: http://127.0.0.1:8761/eureka/

spring:

application:

name: usian-cart-service

datasource:

driver-class-name: com.mysql.jdbc.Driver

url: jdbc:mysql://127.0.0.1:3306/usian?characterEncoding=UTF-8

username: root

password: 1111

type: com.alibaba.druid.pool.DruidDataSource

redis:

cluster:

nodes:

- 192.168.233.131:7001

- 192.168.233.131:7002

- 192.168.233.131:7003

- 192.168.233.131:7004

- 192.168.233.131:7005

- 192.168.233.131:7006

jedis:

pool:

max-active: 20 #连接池最大连接数

max-idle: 10 #连接池中的最大空闲连接

min-idle: 5 # 连接池中的最小空闲连接

#### **logback.xml**

#### **启动类**

@SpringBootApplication

@EnableDiscoveryClient

public class CartServiceApp {

public static void main(String[] args) {

SpringApplication.run(CartServiceApp.class, args);

}

}

### **创建usian\_cart\_feign**

#### **pom.xml**

<!--Spring Cloud OpenFeign Starter -->

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-openfeign</artifactId>

</dependency>

<dependency>

<groupId>com.usian</groupId>

<artifactId>common\_pojo</artifactId>

<version>1.0-SNAPSHOT</version>

</dependency>

### **创建usian\_cart\_web**

#### **pom.xml**

<dependency>

<groupId>com.usian</groupId>

<artifactId>usian\_cart\_feign</artifactId>

<version>1.0-SNAPSHOT</version>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

</dependency>

<dependency>

<groupId>com.usian</groupId>

<artifactId>common\_utils</artifactId>

<version>1.0-SNAPSHOT</version>

</dependency>

<dependency>

<groupId>com.usian</groupId>

<artifactId>usian\_item\_feign</artifactId>

<version>1.0-SNAPSHOT</version>

</dependency>

#### **application.yml**

spring:

application:

name: usian-cart-web

server:

port: 8101

eureka:

client:

service-url:

defaultZone: <http://127.0.0.1:8761/eureka/>

#### **logback.xml**

#### **启动类**

@SpringBootApplication

@EnableDiscoveryClient

@EnableFeignClients

public class CartWebApp {

public static void main(String[] args) {

SpringApplication.run(CartWebApp.class, args);

}

}

## **添加购物车**

1、从cookie中查询商品列表：Map<itemId,TbItem> 商品购买数量使用TbItem的num保存

​ 购物车已存在则直接返回

​ 购物车不存在则创建空的购物车并返回

2、添加商品到购物车：

​ 如果购物车存在该商品，商品数量相加。

​ 如果购物车不存在该商品，根据商品id查询商品信息并添加到购车列表

3、把购车商品列表写入cookie。

​ 读写cookie可以使用CookieUtils工具类实现

## **查看购物车**

### **usian\_cart\_web**

#### **controller**

/\*\*

\* 查看购物车

\*/

@RequestMapping("/showCart")

public Result showCart(String userId,HttpServletRequest

request,HttpServletResponse response){

try{

if(StringUtils.isBlank(userId)){

//在用户未登录的状态下

List<TbItem> list = new ArrayList<TbItem>();

Map<String,TbItem> cart = this.getCartFromCookie(request);

Set<String> keys = cart.keySet();

for(String key :keys){

list.add(cart.get(key));

}

}else{

// 在用户已登录的状态

}

return Result.ok(list);

}catch(Exception e){

e.printStackTrace();

}

return Result.error("error");

}

## **修改购物车**

/\*\*

\* 修改购物车

\*/

@RequestMapping("/updateItemNum")

public Result updateItemNum(String userId,Long itemId,Integer num,

HttpServletRequest request,HttpServletResponse response){

try {

if(StringUtils.isBlank(userId)){

//未登录

//1、获得cookie中的购物车

Map<String, TbItem> cart = getCartFromCookie(request);

//2、修改购物车中的商品

TbItem tbItem = cart.get(itemId.toString());

tbItem.setNum(num);

cart.put(itemId.toString(),tbItem);

//3、把购物车写到cookie

addClientCookie(cart,request,response);

}else{

//登录

}

return Result.ok();

}catch (Exception e){

e.printStackTrace();

return Result.error("修改失败");

}

}

## **删除购物车**

/\*\*

\* 删除购物车中的商品

\*/

@RequestMapping("/deleteItemFromCart")

public Result deleteItemFromCart(Long itemId, String userId, HttpServletRequest

request, HttpServletResponse response) {

try {

if (StringUtils.isBlank(userId)) {

//在用户未登录的状态下

Map<String,TbItem> cart = this.getCartFromCookie(request);

cart.remove(itemId.toString());

this.addClientCookie(request,response,cart);

} else {

// 在用户已登录的状态

}

return Result.ok();

} catch (Exception e) {

e.printStackTrace();

}

return Result.error("删除失败");

}

## 在用户登录情况下

### 使用redis存储商品列表

使用hash对购物车进行归类

​ HASH :HASH\_KEY

​ |--KEY:VALUE

​ |--KEY:VALUE

​ HASH : CART\_REDIS\_KEY

​ |--22:[{...}]

​ |--23:[{...}]

## **添加购物车**

### **usian\_cart\_web**

#### **controller**

/\*\*

\* 购物车 Controller

\*/

@RestController

@RequestMapping("/frontend/cart")

public class CartController {

//... ... ...

@Autowired

private CartServiceFeign cartServiceFeign;

@RequestMapping("/addItem")

public Result addItem(Long itemId, String userId,@RequestParam(defaultValue = "1")

Integernnum,HttpServletRequest request, HttpServletResponse response) {

if (StringUtils.isBlank(userId)) {

/\*\*\*\*\*\*\*\*\*\*\*在用户未登录的状态下\*\*\*\*\*\*\*\*\*\*/

..........

} else {

/\*\*\*\*\*\*\*\*\*\*\*在用户已登录的状态\*\*\*\*\*\*\*\*\*\*/

// 1、从redis中查询商品列表。

Map<String,TbItem> cart = getCartFromRedis(userId);

//2、将商品添加大购物车中

this.addItemToCart(cart,itemId,num);

//3、将购物车缓存到 redis 中

Boolean addCartToRedis = this.addCartToRedis(userId, cart);

if(addCartToRedis){

return Result.ok();

}

return Result.error("error");

}

}

/\*\*

\* 把购车商品列表写入redis

\* @param userId

\* @param cart

\*/

private Boolean addCartToRedis(String userId, Map<String, TbItem> cart) {

return cartServiceFeign.insertCart(userId, cart);

}

/\*\*

\* 从redis中查询购物车

\* @param userId

\*/

private Map<String,TbItem> getCartFromRedis(String userId) {

Map<String,TbItem> cart = cartServiceFeign.selectCartByUserId(userId);

if(cart!=null && cart.size()>0){

return cart;

}

return new HashMap<String,TbItem>();

}

//... ... ...

}

### **usian\_cart\_feign**

@FeignClient("usian-cart-service")

public interface CartServiceFeign {

@RequestMapping("/service/cart/selectCartByUserId")

Map<String, TbItem> selectCartByUserId(@RequestParam String userId);

@RequestMapping("/service/cart/insertCart?userId=234234")

Boolean insertCart(@RequestParam String userId, Map<String, TbItem> cart);

}

### **usian\_cart\_service**

#### **application.yml**

#购物车

CART\_REDIS\_KEY: CART\_REDIS\_KEY

#### **Service**

/\*\*

\* 购物车操作业务层

\*/

@Service

public class CartServiceImpl implements CartService {

@Autowired

private RedisClient redisClient;

@Value("${CART\_REDIS\_NAME}")

private String CART\_REDIS\_NAME;

/\*\*

\* 根据用户 ID 查询用户购物车

\*/

@Override

public Map<String, TbItem> selectCartByUserId(String userId) {

return (Map<String, TbItem>) redisClient.hget(CART\_REDIS\_NAME,userId);

}

/\*\*

\* 缓存购物车

\*

\* @param cart

\*/

@Override

public Boolean insertCart(String userId, Map<String, TbItem> cart) {

return redisClient.hset(CART\_REDIS\_NAME, userId, cart);

}

}

#### **controller**

/\*\*

\* 购物车操作

\*/

@RestController

@RequestMapping("/service/cart")

public class CartController {

@Autowired

private CartService cartService;

/\*\*

\* 根据用户 ID 查询用户购物车

\*/

@RequestMapping("/selectCartByUserId")

public Map<String, TbItem> selectCartByUserId(@RequestParam String userId){

return this.cartService.selectCartByUserId(userId);

}

/\*\*

\* 将购物车缓存到 redis 中

\*/

@RequestMapping("/insertCart")

public Boolean insertCart(String userId, @RequestBody Map<String, TbItem> map) {

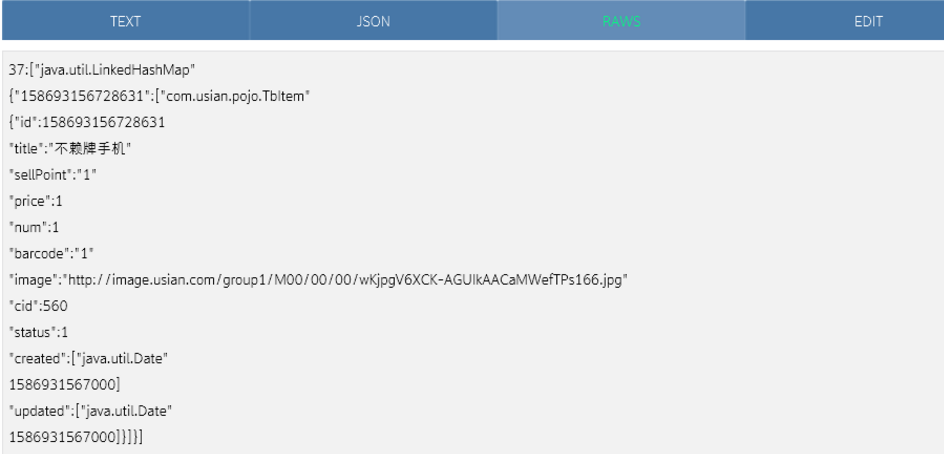
return this.cartService.insertCart(userId, map);

}

}

#### **测试**

查看redis缓存数据



## **查看购物车**

### **usian\_cart\_web**

#### **Controller**

/\*\*

\* 查看购物车

\*/

@RequestMapping("/showCart")

public Result showCart(String userId,HttpServletRequest

request,HttpServletResponse response){

try{

if(StringUtils.isBlank(userId)){

//在用户未登录的状态下

//............

}else{

// 在用户已登录的状态

List<TbItem> list = new ArrayList<TbItem>();

Map<String,TbItem> cart = this.getCart(userId);

Set<String> keys = cart.keySet();

for(String key :keys){

list.add(cart.get(key));

}

return Result.ok(list);

}

}catch(Exception e){

e.printStackTrace();

}

return Result.error("error");

}

## **修改购物车**

### **usian\_cart\_web**

#### **controller**

/\*\*

\* 修改购物车中商品的数量

\*/

@RequestMapping("/updateItemNum")

public Result updateItemNum(Long itemId, String userId, Integer

num, HttpServletRequest request, HttpServletResponse response) {

try{

if (StringUtils.isBlank(userId)) {

//在用户未登录的状态下

//.........

} else {

// 在用户已登录的状态

Map<String,TbItem> cart = this.getCart(userId);

TbItem item = cart.get(itemId.toString());

if(item != null){

item.setNum(num);

}

//将新的购物车缓存到 Redis 中

this.addCartToRedis(userId,cart);

return Result.ok();

}

}catch(Exception e){

e.printStackTrace();

}

return Result.error("修改失败");

}

## **删除购物车**

### **usian\_cart\_web**

#### **Controller**

/\*\*

\* 删除购物车中的商品

\*/

@RequestMapping("/deleteItemFromCart")

public Result deleteItemFromCart(Long itemId, String userId, HttpServletRequest

request, HttpServletResponse response) {

try {

if (StringUtils.isBlank(userId)) {

//在用户未登录的状态下

//.........

} else {

// 在用户已登录的状态

Map<String,TbItem> cart = this.getCart(userId);

cart.remove(itemId.toString());

//将新的购物车缓存到 Redis 中

this.addCartToRedis(userId,cart);

return Result.ok();

}

} catch (Exception e) {

e.printStackTrace();

}

return Result.error("删除失败");

}

# **购物车同步**

## **业务逻辑**

在用户未登录情况下往购物车添加商品，添加结束用户登录，此时需把cookie中的数据同步到redis：

​ a) 如果redis中有数据cookie中也有数据，需要做数据合并。相同商品数量相加，不同商品添加一个新商品。

​ b) 把cookie中的数据删除，避免下次重复同步

# **订单分析**

## **订单功能**

1、在购物车页面点击“去结算”按钮跳转到订单确认页面

​ a) 展示商品列表

​ b) 配送地址列表

​ c) 选择支付方式

2、订单确认页面需要根据用户查询配送地址，展示订单确认页面之前，应该确认用户身份

​ a) 如果用户未登录或登录过期跳转到登录页面

​ d) 登录成功后再跳转到订单确认页面

3、提交订单

​ a) 生成订单

4、扣减库存

5、关闭超时订单

​ a) 定时扫描超时2天未付款的订单，并关闭订单

​ b) 加回库存

## **创建工程**

### **usian\_order\_service**

### **usian\_order\_web**

### **usian\_order\_feign**

## 用户身份认证

在展示订单确认页面之前，需要对用户身份进行认证，要求用户必须登录。

## **功能分析**

1、使用springmvc的拦截器拦截所有订单的请求

2、业务逻辑

​ a) 从cookie中取token。

​ b) 根据token调用sso服务查询用户信息。

​ d) 如果查不到用户信息则跳转到登录页面。

​ e) 查询到用户信息放行。

## **usian\_order\_web**

### **interceptor**

/\*\*

\* 在结算之前判断用户是否登录

\*/

@Component

public class UserLoginInterceptor implements HandlerInterceptor {

@Autowired

private SSOServiceFeign ssoServiceFeign;

@Override

public boolean preHandle(HttpServletRequest request, HttpServletResponse

response, Object handler) throws Exception {

//对用户的 token 做判断

String token = request.getParameter("token");

if (StringUtils.isBlank(token)) {

return false;

}

//如果用户 token 不为空，则校验用户在 redis 中是否失效

TbUser tbUser = ssoServiceFeign.getUserByToken(token);

if (tbUser == null) {

return false;

}

return true;

}

}

### **config**

/\*\*

\* 拦截器配置类

\*/

@Component

public class WebConfig implements WebMvcConfigurer {

@Autowired

private UserLoginInterceptor userLoginInterceptor;

/\*\*

\* 注册拦截器

\* @param registry

\*/

@Override

public void addInterceptors(InterceptorRegistry registry) {

InterceptorRegistration registration =

registry.addInterceptor(this.userLoginInterceptor);

//拦截那个 URI

registration.addPathPatterns("/frontend/order/\*\*");

}

}

## 展示订单确认页面

## **功能分析**

1、在购物车页面点击“去结算”按钮跳转到订单确认页面。

2、请求的url：/frontend/order/goSettlement

3、参数：ids，userId，token

4、查询redis中的购物车数据并返回给前端

5、配送地址列表，需要用户登录。需要根据用户id查询收货地址列表。静态数据。

6、支付方式。静态数据。

## **usian\_order\_web**

### **controller**

/\*\*

\* 订单服务 Controller

\*/

@RestController

@RequestMapping("/frontend/order")

public class OrderController {

@Autowired

private CartServiceFeign cartServiceFeign;

@Autowired

private OrderServiceFeign orderServiceFeign;

@RequestMapping("/goSettlement")

public Result goSettlement(String[] ids, String userId) {

//获取购物车

Map<String, TbItem> cart = cartServiceFeign.selectCartByUserId(userId);

//从购物车中获取选中的商品

List<TbItem> list = new ArrayList<TbItem>();

for (String id : ids) {

list.add(cart.get(id));

}

if(list.size()>0) {

return Result.ok(list);

}

return Result.error("error");

}

}

# **提交订单**

## **如何生成订单号**

**订单号需求是什么？**1、唯一2、可读性高 a) 纯数字

**如何生成：**

1. 时间戳 a)可能重复
2. 时间戳 +随机数 a)可能重复
3. 手机号 a)重复
4. 时间戳 +自增id a)可行（使用Redis的INCR命令完成）（初值：100544）
5. 时间戳+用户id/手机号

6、时间戳+用户id +店铺id a)可行，适用于C2C、B2C平台

## **功能分析**

**保存订单信息**

​ a、在订单确认页面点击“提交订单”按钮生成订单。

​ b、请求的url：/frontend/order/insertOrder

​ c、参数：订单、订单商品、订单物流

​ d、返回值：{"status":200,"msg":"OK","data":"订单ID"}

## **common\_pojo**

package com.usian.pojo;

public class OrderInfo {

private TbOrder tbOrder;

private TbOrderShipping tbOrderShipping;

private String orderItem;

... ... ...

}

## **usian\_order\_web**

### **controller**

/\*\*

\* 创建订单

\*/

@RequestMapping("/insertOrder")

public Result insertOrder(String orderItem, TbOrder tbOrder , TbOrderShipping

tbOrderShipping) {

//因为一个request中只包含一个request body. 所以feign不支持多个@RequestBody。

OrderInfo orderInfo = new OrderInfo();

orderInfo.setOrderItem(orderItem);

orderInfo.setTbOrder(tbOrder);

orderInfo.setOrderShipping(tbOrderShipping);

Long orderId = orderServiceFeign.insertOrder(orderInfo);

if (orderId != null) {

//删除购物车

return Result.ok(orderId);

}

return Result.error("error");

}

## **usian\_order\_feign**

@FeignClient("usian-order-service")

public interface OrderServiceFeign {

@RequestMapping("/service/order/insertOrder")

Long insertOrder(OrderInfo orderInfo);

}

## **usian\_order\_service**

### **application.yml**

#订单

ORDER\_ID\_KEY: ORDER\_ID\_KEY

ORDER\_ID\_BEGIN: 100544

ORDER\_ITEM\_ID\_KEY: ORDER\_ITEM\_ID\_KEY

### **service**

@Override

public String insertOrder(OrderInfo orderInfo) {

//1、解析orderInfo

TbOrder tbOrder = orderInfo.getTbOrder();

TbOrderShipping tbOrderShipping = orderInfo.getTbOrderShipping();

List<TbOrderItem> tbOrderItemList =

JsonUtils.jsonToList(orderInfo.getOrderItem(), TbOrderItem.class);

//2、保存订单信息

if(!redisClient.exists(ORDER\_ID\_KEY)){

redisClient.set(ORDER\_ID\_KEY,ORDER\_ID\_BEGIN);

}

Long orderId = redisClient.incr(ORDER\_ID\_KEY, 1L);

tbOrder.setOrderId(orderId.toString());

Date date = new Date();

tbOrder.setCreateTime(date);

tbOrder.setUpdateTime(date);

//1、未付款，2、已付款，3、未发货，4、已发货，5、交易成功，6、交易关闭

tbOrder.setStatus(1);

tbOrderMapper.insertSelective(tbOrder);

//3、保存明细信息

if(!redisClient.exists(ORDER\_ITEM\_ID\_KEY)){

redisClient.set(ORDER\_ITEM\_ID\_KEY,0);

}

for (int i = 0; i < tbOrderItemList.size(); i++) {

Long oderItemId = redisClient.incr(ORDER\_ITEM\_ID\_KEY, 1L);

TbOrderItem tbOrderItem = tbOrderItemList.get(i);

tbOrderItem.setId(oderItemId.toString());

tbOrderItem.setOrderId(orderId.toString());

tbOrderItemMapper.insertSelective(tbOrderItem);

}

//4、保存物流信息

tbOrderShipping.setOrderId(orderId.toString());

tbOrderShipping.setCreated(date);

tbOrderShipping.setUpdated(date);

tbOrderShippingMapper.insertSelective(tbOrderShipping);

//5、返回订单id

return orderId.toString();

}

/\*\*

\* 订单服务 Controller

\*/

@RestController

@RequestMapping("/service/order")

public class OrderController {

@Autowired

private OrderService orderService;

/\*\*

\* 创建订单

\*/

@RequestMapping("/insertOrder")

public Long insertOrder(@RequestBody OrderInfo orderInfo) {

return orderService.insertOrder(orderInfo);

## **测试**



# **扣减库存**

## **分析**

将消息写入消息队列，业务逻辑以异步的方式运行，加快响应速度

## **usian\_order\_service**

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-amqp</artifactId>

</dependency>

spring:

rabbitmq:

host: 192.168.157.132

Port: 5762

username: admin

password: 1111

virtual-host: /

### **service**

@Service

@Transactional

public class OrderServiceImpl implements OrderService {

... ... ...

@Autowired

private AmqpTemplate amqpTemplate;

@Override

public Long insertOrder(OrderInfo orderInfo) {

/\*\*\*\*\*\*\*\*\*\*\*\*1、向订单表插入数据。\*\*\*\*\*\*\*\*/

./\*\*\*\*\*\*\*\*\*\*\*\*2、向订单明细表插入数据\*\*\*\*\*\*\*\*/

... ... ...

/\*\*\*\*\*\*\*\*\*\*\*\*3、向订单物流表插入数据。\*\*\*\*\*\*\*\*/

... ... ...

//发布消息到mq，完成扣减库存

amqpTemplate.convertAndSend("order\_exchage","order.add", orderId);

/\*\*\*\*\*\*\*\*\*\*\*\*4、返回订单id\*\*\*\*\*\*\*\*/

... ... ...

}

## **usian\_item\_service**

### **service**

修改ItemServiceImpl

/\*\*

\* 修改商品库存数量

\* @param orderId

\* @return

\*/

@Override

public Integer updateTbItemByOrderId(String orderId) {

TbOrderItemExample tbOrderItemExample = new TbOrderItemExample();

TbOrderItemExample.Criteria criteria = tbOrderItemExample.createCriteria();

criteria.andOrderIdEqualTo(orderId);

List<TbOrderItem> tbOrderItemList =

tbOrderItemMapper.selectByExample(tbOrderItemExample);

int result = 0;

for (int i = 0; i < tbOrderItemList.size(); i++) {

TbOrderItem tbOrderItem = tbOrderItemList.get(i);

TbItem tbItem =

tbItemMapper.selectByPrimaryKey(Long.valueOf(tbOrderItem.getItemId()));

tbItem.setNum(tbItem.getNum()-tbOrderItem.getNum());

result += tbItemMapper.updateByPrimaryKeySelective(tbItem);

}

return result;

}

### **listener**

@Component

public class ItemMQListener {

@Autowired

private ItemService itemService;

/\*\*

\* 监听者接收消息三要素：

\* 1、queue

\* 2、exchange

\* 3、routing key

\*/

@RabbitListener(bindings = @QueueBinding(

value = @Queue(value="item\_queue",durable = "true"),

exchange = @Exchange(value="order\_exchage",type= ExchangeTypes.TOPIC),

key= {"\*.\*"}))

public void listen(String orderId) throws Exception {

System.out.println("接收到消息：" + orderId);

Integer result = itemService.updateTbItemByOrderId(orderId);

if(!(result>0)){

throw new RuntimeException("扣减失败");

}

## **测试**

测试订单提交前后数据库中对应商品的库存数量变化

# **关闭超时订单**

## **分析**

1、定义job扫描订单表：

​ a、修改订单的状态为关闭状态、结束时间、关闭时间、修改时间为当前时间

​ b、把订单中商品的库存数量加回去

​ 扫描条件：状态是未付款 并且 创建时间 <= 当前时间 – 2天 并且付款方式为在线支付的订单

2、 定义触发条件

​ a) 理论上需要实时触发（性能问题）

​ b) 1分钟触发一次 0 \*/1 \* \* \* ?

## **common\_mapper**

### **TbOrderMapper**

List<TbOrder> selectOvertimeOrder();

### **TbOrderMapper.xml**

<!--查询超时订单 -->

<select id="selectOvertimeOrder" resultMap="BaseResultMap">

SELECT

\*

FROM

tb\_order

WHERE

create\_time &lt;= DATE\_SUB(NOW(), INTERVAL 2 DAY)

AND status = 1

AND payment\_type = 1

</select>

## **usian\_order\_service**

<!-- quartz启动器 -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-quartz</artifactId>

</dependency>

### **service**

/\*\*

\* 查询超时订单

\* @return

\*/

@Override

public List<TbOrder> selectOvertimeOrder() {

return orderMapper.selectOvertimeOrder();

}

/\*\*

\* 关闭超时订单

\* @param tbOrder

\*/

@Override

public void updateOverTimeTbOrder(TbOrder tbOrder) {

tbOrder.setStatus(6);

Date date = new Date();

tbOrder.setCloseTime(date);

tbOrder.setEndTime(date);

tbOrder.setUpdateTime(date);

tbOrderMapper.updateByPrimaryKeySelective(tbOrder);

}

/\*\*

\* 把订单中商品的库存数量加回去

\* @param itemId

\* @param num

\*/

@Override

@Override

public void updateTbItemByOrderId(String orderId) {

//1、通过orderId查询LisT<TbOrderItem>

TbOrderItemExample tbOrderItemExample = new TbOrderItemExample();

TbOrderItemExample.Criteria criteria = tbOrderItemExample.createCriteria();

criteria.andOrderIdEqualTo(orderId);

List<TbOrderItem> tbOrderItemList =

tbOrderItemMapper.selectByExample(tbOrderItemExample);

for (int i = 0; i < tbOrderItemList.size(); i++) {

TbOrderItem tbOrderItem = tbOrderItemList.get(i);

//2、修改商品库存

TbItem tbItem =

tbItemMapper.selectByPrimaryKey(Long.valueOf(tbOrderItem.getItemId()));

tbItem.setNum(tbItem.getNum()+tbOrderItem.getNum());

tbItem.setUpdated(new Date());

tbItemMapper.updateByPrimaryKey(tbItem);

}

### **quartz**

public class OrderQuartz implements Job {

@Autowired

private OrderService orderService;

/\*\*

\* 关闭超时订单

\*/

@Override

public void execute(JobExecutionContext context) throws JobExecutionException {

//1、查询超时订单

List<TbOrder> tbOrderList = orderService.selectOverTimeTbOrder();

//2、关闭超时订单

for (int i = 0; i < tbOrderList.size(); i++) {

TbOrder tbOrder = tbOrderList.get(i);

orderService.updateOverTimeTbOrder(tbOrder);

//3、把超时订单中的商品库存数量加回去

orderService.updateTbItemByOrderId(tbOrder.getOrderId());

}

### **Factory**

@Component("myAdaptableJobFactory")

public class MyAdaptableJobFactory extends AdaptableJobFactory {

//AutowireCapableBeanFactory 可以将一个对象添加到SpringIOC容器中，并且完成该对象注入

@Autowired

private AutowireCapableBeanFactory autowireCapableBeanFactory;

/\*\*

\* 解决任务类中注入service报错问题：

\* 该方法将需要实例化的任务对象手动的添加到springIOC容器中并且完成对象的注入

\*/

@Override

protected Object createJobInstance(TriggerFiredBundle bundle) throws Exception {

Object obj = super.createJobInstance(bundle);

//将obj对象添加Spring IOC容器中，并完成注入

this.autowireCapableBeanFactory.autowireBean(obj);

return obj;

}

### **config**

/\*\*

\* Quartz配置类

\*/

@Configuration

public class QuartzConfig {

/\*\*

\* 1.创建Job对象

\*/

@Bean

public JobDetailFactoryBean jobDetailFactoryBean(){

JobDetailFactoryBean factory = new JobDetailFactoryBean();

//关联我们自己的Job类

factory.setJobClass(OrderQuartz.class);

return factory;

}

/\*\*

\* Cron Trigger

\*/

@Bean

public CronTriggerFactoryBean cronTriggerFactoryBean(JobDetailFactoryBean

jobDetailFactoryBean){

CronTriggerFactoryBean factory = new CronTriggerFactoryBean();

factory.setJobDetail(jobDetailFactoryBean.getObject());

//设置触发时间

//factory.setCronExpression("0/2 \* \* \* \* ?");

factory.setCronExpression("0 \*/1 \* \* \* ?");

return factory;

}

/\*\*

\* 3.创建Scheduler对象

\*/

@Bean

public SchedulerFactoryBean schedulerFactoryBean(CronTriggerFactoryBean

cronTriggerFactoryBean, MyAdaptableJobFactory myAdaptableJobFactory){

SchedulerFactoryBean factory = new SchedulerFactoryBean();

//关联trigger

factory.setTriggers(cronTriggerFactoryBean.getObject());

factory.setJobFactory(myAdaptableJobFactory);

return factory;

}

}

## **测试**

测试订单中商品库存数量是否加回

# **quartz集群任务重复执行问题**

## **分析**

quartz 在集群的时候，任务会出现重复执行的情况：

## **quartz**

使用redis分布式锁解决quartz 集群任务重复执行的问题

public class OrderQuartz implements Job {

@Autowired

private OrderService orderService;

@Autowired

private RedisClient redisClient;

/\*\*

\* 关闭超时订单

\*/

@Override

public void execute(JobExecutionContext context) throws JobExecutionException {

String ip = null;

try {

ip = InetAddress.getLocalHost().getHostAddress();

} catch (UnknownHostException e) {

e.printStackTrace();

}

//解决quartz集群任务重复执行

if(redisClient.setnx("SETNX\_LOCK\_ORDER\_KEY",ip,30)) {

//... ... ... 关闭超时订单业务

redisClient.del("SETNX\_LOCK\_ORDER\_KEY");

}else{

System.out.println(

"============机器："+ip+" 占用分布式锁，任务正在执行=======================");

}

}

}

## **测试**

拷贝usian\_order\_service，修改module名字及端口号，测试是否存在重复关闭订单的问题