# EGO商城订单模块

## 订单系统描述

订单系统是我们所有系统里面最重要的一个系统！

从我们后台上架商品，导入到搜索引擎里面，以及用户浏览商品，将商品添加到购物车里面，购物车系统，消息系统，都是为订单系统服务的！

订单系统完成了一个闭环的操作，它是最核心的一步！

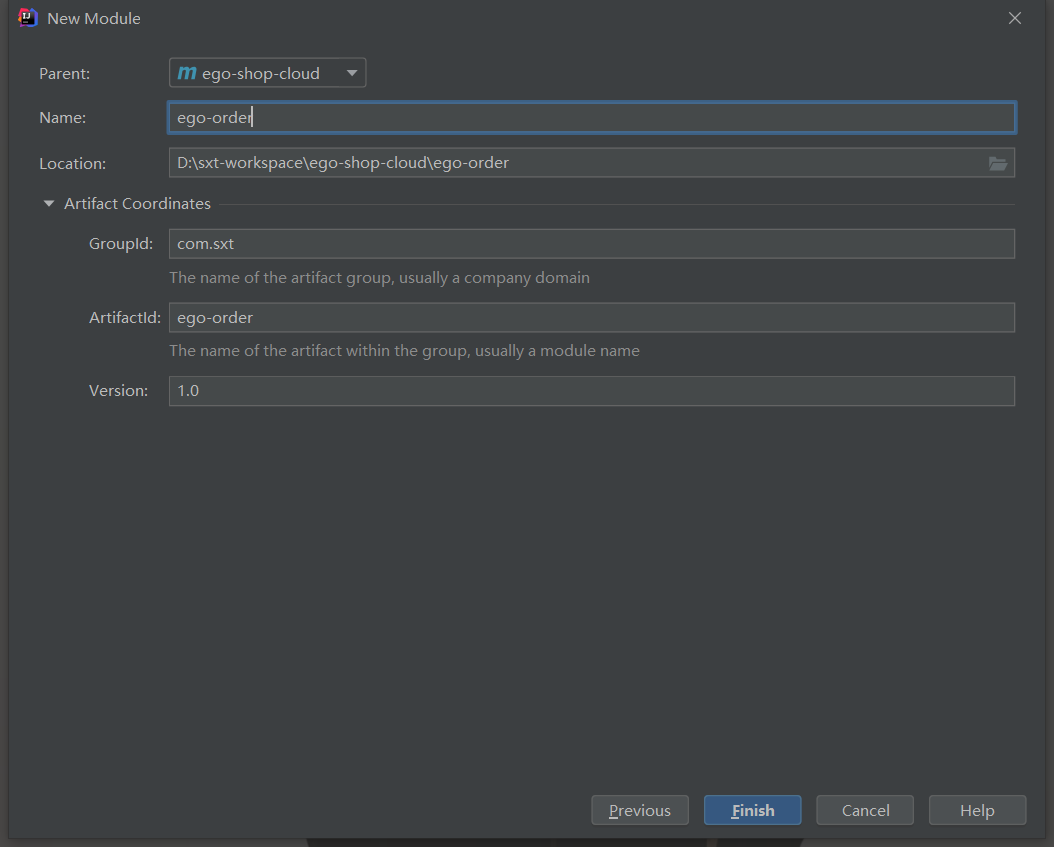
## 功能流程介绍

用户购买商品下订单成功后，减少实际库存，通过mq通知es修改库存，通过mq去发微信消息，调用支付服务生成支付的二维码

如果是购物车里面的商品，还需要修改购物车，订单放入延时队列等待支付

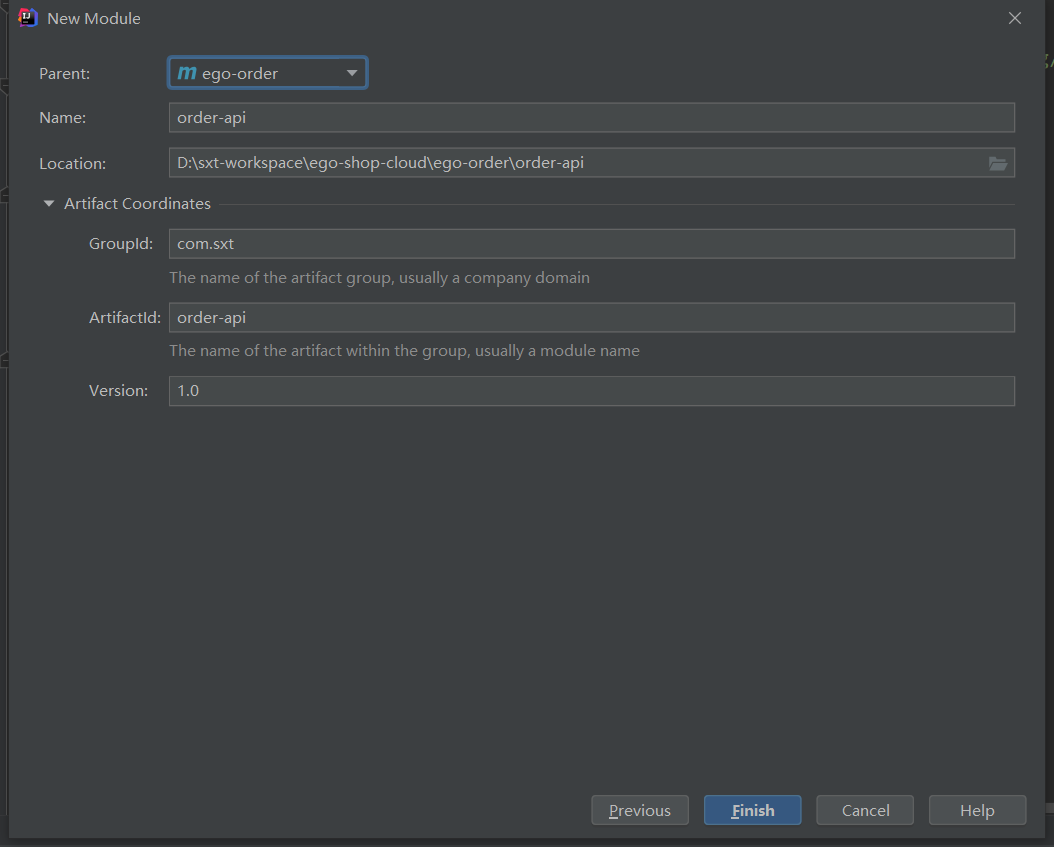
## 开始搭建

### 在ego-shop-cloud下面新建ego-order模块



### 在ego-order下**创建**order-api模块

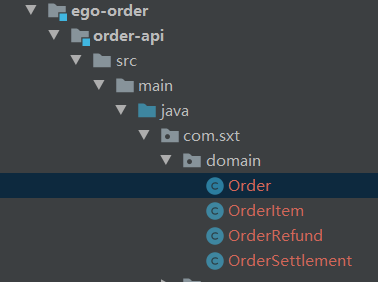
#### 新建模块



#### 修改pom.xml文件

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <project xmlns="http://maven.apache.org/POM/4.0.0"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  <parent>  <artifactId>ego-order</artifactId>  <groupId>com.sxt</groupId>  <version>1.0</version>  </parent>  <modelVersion>4.0.0</modelVersion>  <artifactId>order-api</artifactId>  <description>ego商城订单服务api</description>  <dependencies>  <!-- 需要调用购物车的对象-->  <dependency>  <groupId>com.sxt</groupId>  <artifactId>cart-api</artifactId>  <version>1.0</version>  </dependency>  <!-- 下订单需要用户回显地址-->  <dependency>  <groupId>com.sxt</groupId>  <artifactId>member-api</artifactId>  <version>1.0</version>  </dependency>  <!-- 需要得到商品的信息-->  <dependency>  <groupId>com.sxt</groupId>  <artifactId>product-api</artifactId>  <version>1.0</version>  <exclusions>  <exclusion>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-data-elasticsearch</artifactId>  </exclusion>  </exclusions>  </dependency>  <!-- 需要给用户发微信消息-->  <dependency>  <groupId>com.sxt</groupId>  <artifactId>message-api</artifactId>  <version>1.0</version>  </dependency>  <!-- 需要web和openFeign远程调用-->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-openfeign</artifactId>  </dependency>  </dependencies>  </project> |

#### 逆向**生成**表



#### 修改Order类

|  |
| --- |
| /\*\*  \* 订单中具体的商品信息，我们共用购物车的对象  \*/  @TableField(exist = false)  private List<ShopCartItem> orderItemDtos = Collections.EMPTY\_LIST; |

#### 创建OrderParam用于确认订单

|  |
| --- |
| @Data  public class OrderParam {  @ApiModelProperty("订单的收货地址的id")  private Integer addrId;  @ApiModelProperty("要购买的购物车条目的id，用户从购物车页面来")  private List<Long> basketIds;  @ApiModelProperty("购买商品的条目，从页面详情来")  private ShopCartItem orderItem;  private List<Integer> couponIds;  private Integer userChangeCoupon;  } |

#### 创建OrderVo用于订单的具体详情

|  |
| --- |
| @Data  public class OrderVo implements Serializable {  @ApiModelProperty(value = "订单的详细数据，包含商品数据，运费等")  private List<ShopCartOrder> shopCartOrders;  @ApiModelProperty(value = "实际的订单金额")  private BigDecimal actualTotal;  @ApiModelProperty(value = "订单总金额")  private BigDecimal total;  @ApiModelProperty(value = "商品总数量")  private Integer totalCount;  @ApiModelProperty(value = "用户的收货地址")  private UserAddr userAddr;  @ApiModelProperty(value = "订单编号")  private String orderSn;  } |

#### 创建ShopCartOrder对象用于展示订单页面

|  |
| --- |
| @Data  @NoArgsConstructor  @AllArgsConstructor  public class ShopCartOrder implements Serializable {  @ApiModelProperty("订单里面的每一个商品")  private List<ShopCartItem> shopCartItemDiscounts = Collections.emptyList();  @ApiModelProperty("订单的运费")  private BigDecimal transfee;  @ApiModelProperty("订单的优惠")  private BigDecimal shopReduce;  } |

#### 创建OrderResult对象用于页面查询展示

|  |
| --- |
| @Data  @AllArgsConstructor  @NoArgsConstructor  public class OrderResult {  @ApiModelProperty("未付款")  private Integer unPay = 0;  @ApiModelProperty("待发货")  private Integer payed = 0;  @ApiModelProperty("待收货")  private Integer consignment = 0;  } |

#### 创建远程调用购物车OrderCartFeign

|  |
| --- |
| @FeignClient(value = "cart-service", fallback = OrderCartFeignHystrix.class)  public interface OrderCartFeign {  //根据购物车的ids 查询出对象  @PostMapping("/orderGetBasketByIds")  List<Basket> getBasketByIds(@RequestBody List<Long> basketIds);  /\*\*  \* 远程调用清空购物车  \*  \* @param skuIds  \* @param userId  \*/  @PostMapping("clearCartBySkuIdAndUserId")  void deleteCartByUserId(@RequestBody List<Long> skuIds, @RequestParam("userId") String userId);  } |

#### 创建远程调用购物车的熔断OrderCartFeignHystrix

|  |
| --- |
| @Component  @Slf4j  public class OrderCartFeignHystrix implements OrderCartFeign {  @Override  public List<Basket> getBasketByIds(List<Long> basketIds) {  log.error("远程调用查询购物车失败");  return Collections.emptyList();  }  /\*\*  \* 远程调用清空购物车  \*  \* @param skuIds  \* @param userId  \*/  @Override  public void deleteCartByUserId(List<Long> skuIds, String userId) {  log.error("远程调用清空购物车失败");  }  } |

#### 创建远程调用会员OrderMemberFeign

|  |
| --- |
| @FeignClient(value = "member-service", fallback = OrderMemberFeignHystrix.class)  public interface OrderMemberFeign {  /\*\*  \* 远程调用获取用的默认地址  \*  \* @param userId  \* @return  \*/  @GetMapping("/p/address/getDefaultAddr/{userId}")  ResponseEntity<UserAddr> getDefaultUserAddr(@PathVariable("userId") String userId);  } |

#### 创建远程调用会员的熔断OrderMemberFeignHystrix

|  |
| --- |
| @Component  @Slf4j  public class OrderMemberFeignHystrix implements OrderMemberFeign {  /\*\*  \* 远程调用获取用的默认地址  \*  \* @param userId  \* @return  \*/  @Override  public ResponseEntity<UserAddr> getDefaultUserAddr(String userId) {  log.error("远程调用获得用户地址失败");  return ResponseEntity.ok(new UserAddr());  }  } |

#### 创建远程调用商品OrderProductFeign

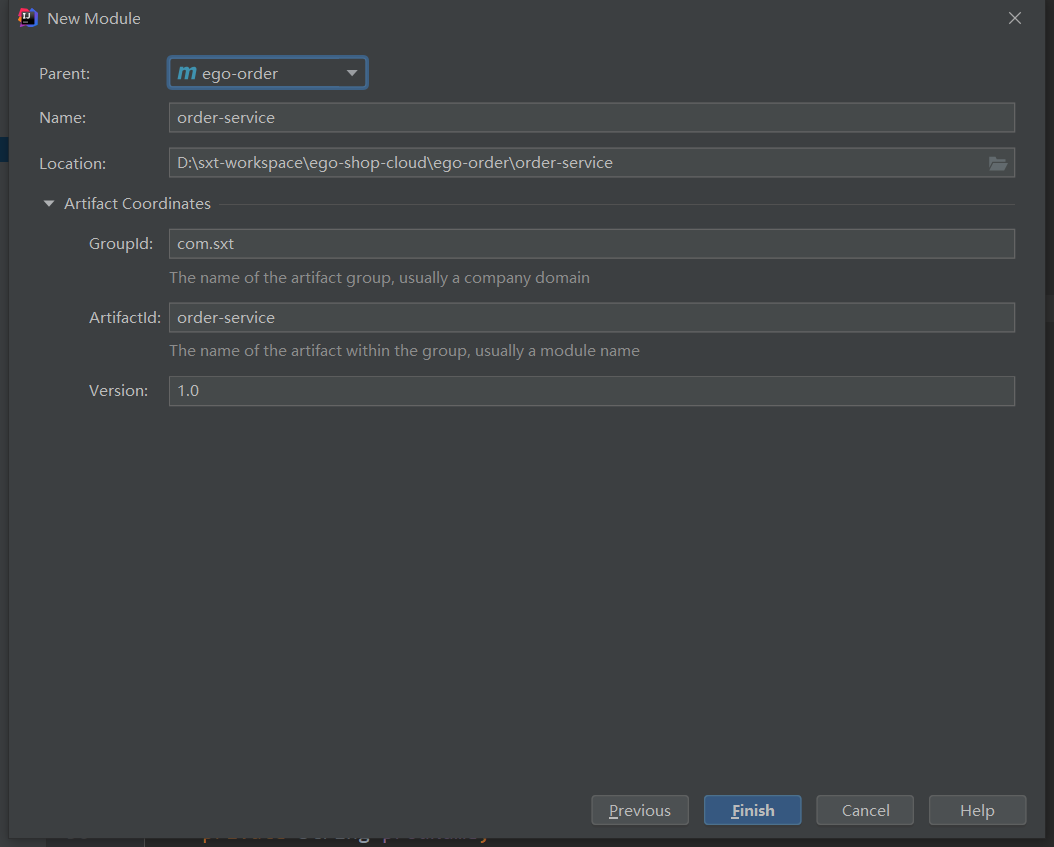
|  |
| --- |
| @FeignClient(value = "product-service", fallback = OrderProductFeignHystrix.class)  public interface OrderProductFeign {  /\*\*  \* 根据prodId得到prod对象  \*  \* @param prodId  \* @return  \*/  @GetMapping({"/prod/prod/prod/prodInfo"})  @ApiOperation("加载商品的详情")  ResponseEntity<Prod> loadProd(@RequestParam(required = true) Long prodId);  /\*\*  \* 远程调用减少库存  \*  \* @param prodId  \* @param skuId  \* @param count  \*/  @GetMapping("prod/prod/prod/changeStocks")  void changeStock(@RequestParam("prodId") Long prodId, @RequestParam("skuId") Long skuId, @RequestParam("count") Integer count);  } |

#### 创建远程调用商品的熔断OrderProductFeignHystrix

|  |
| --- |
| @Component  @Slf4j  public class OrderProductFeignHystrix implements OrderProductFeign {  /\*\*  \* 根据prodId得到prod对象  \*  \* @param prodId  \* @return  \*/  @Override  public ResponseEntity<Prod> loadProd(Long prodId) {  log.error("远程调用查询商品失败");  return ResponseEntity.ok(new Prod());  }  /\*\*  \* 远程调用减少库存  \*  \* @param prodId  \* @param skuId  \* @param count  \*/  @Override  public void changeStock(Long prodId, Long skuId, Integer count) {  log.error("远程调用减少库存失败");  }  } |

### 在ego-order下**创建**order-service模块

#### 新建模块



#### 修改pom.xml文件

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <project xmlns="http://maven.apache.org/POM/4.0.0"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  <parent>  <artifactId>ego-order</artifactId>  <groupId>com.sxt</groupId>  <version>1.0</version>  </parent>  <modelVersion>4.0.0</modelVersion>  <artifactId>order-service</artifactId>  <description>ego订单服务service</description>  <dependencies>  <dependency>  <groupId>com.sxt</groupId>  <artifactId>ego-common</artifactId>  <version>1.0</version>  </dependency>  <dependency>  <groupId>com.sxt</groupId>  <artifactId>order-api</artifactId>  <version>1.0</version>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-actuator</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-bus-amqp</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-config-client</artifactId>  </dependency>  <!-- spring的切面编程-->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-aop</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-data-redis-reactive</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>  </dependency>  <!-- 引入druid starter-->  <dependency>  <groupId>com.alibaba</groupId>  <artifactId>druid-spring-boot-starter</artifactId>  </dependency>  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  </dependency>  </dependencies>  <build>  <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  </plugin>  </plugins>  </build>  </project> |

#### 创建启动类

|  |
| --- |
| @SpringBootApplication  @EnableEurekaClient  @EnableFeignClients  public class OrderServiceApplication {  public static void main(String[] args) {  SpringApplication.run(OrderServiceApplication.class, args);  }  } |

#### 创建bootstrap.yml

|  |
| --- |
| spring:  application:  name: order-service  cloud:  config:  discovery:  enabled: true  service-id: config-server  profile: dev  name: order-service  label: master  main: #允许多个feign调用相同的接口，因为我们依赖了其他的api，有相同的远程调用接口  allow-bean-definition-overriding: true  eureka:  client:  service-url:  defaultZone: ${EUREKA\_SERVER:http://admin:admin@localhost:8761/eureka}  instance:  hostname: ${APP\_HOST:localhost}  prefer-ip-address: true  instance-id: ${spring.application.name}:${server.port}  lease-expiration-duration-in-seconds: 30  lease-renewal-interval-in-seconds: 10 |

#### 创建远端order-service/order-service-dev.yml

|  |
| --- |
| server:  port: ${APP\_PORT:8088}  spring:  datasource:  druid:  url: jdbc:mysql://localhost:3306/ego-shop?useSSL=false&serverTimezone=GMT%2B8  username: root  password: root  driver-class-name: com.mysql.cj.jdbc.Driver  max-active: 30  min-idle: 3  initial-size: 3  validation-query: select 1  max-wait: 5000  stat-view-servlet:  enabled: true  login-username: admin  login-password: admin  allow:  deny:  url-pattern:  - /druid/\*  rabbitmq:  host: 192.168.226.129  password: admin  username: admin  port: 5672  listener:  simple:  acknowledge-mode: manual #手动签收消息 默认是自动的  redis:  host: 192.168.226.129  port: 6380  password: cxs1013??  database: 0  management:  endpoints:  web:  exposure:  include: '\*'  feign:  hystrix:  enabled: false  ribbon:  ReadTimeout: 18000  ConnectTimeout: 18000  hystrix: #hystrix的全局控制  command:  default: #default是全局控制，也可以换成单个方法控制，把default换成方法名即可  fallback:  isolation:  semaphore:  maxConcurrentRequests: 1000  circuitBreaker:  enabled: true #开启断路器  requestVolumeThreshold: 20 #失败次数（阀值）  sleepWindowInMilliseconds: 30000 #窗口时间  errorThresholdPercentage: 60 #失败率  execution:  isolation:  strategy: SEMAPHORE #使用信号量隔离方式  thread:  timeoutInMilliseconds: 5000  semaphore:  maxConcurrentRequests: 1000 #最大并发量  mybatis-plus:  mapper-locations: classpath:mapper/\*\*/\*.xml  configuration:  log-impl: org.apache.ibatis.logging.stdout.StdOutImpl  swagger2:  base-package: com.sxt.controller  name: sxt  url: https://gitee.com/smiledouble  email: 775610843@qq.com  title: 订单管理系统接口  description: 实现订单管理系统的所有接口  license: cxs  license-url: https://gitee.com/smiledouble  terms-of-service-url: https://gitee.com/smiledouble |

#### 创建死信队列

|  |
| --- |
| @Configuration  public class OrderRabbitConfig {  /\*\*  \* 存放订单的队列  \*  \* @return  \*/  @Bean  public Queue orderQueue() {  Map<String, Object> data = new HashMap<>();  //消息死后到哪个交换机去  data.put("x-dead-letter-exchange", QueueConstant.ORDER\_DEAD\_EXCHANGE);  //消息死后到哪个路由key  data.put("x-dead-letter-routing-key", QueueConstant.ORDER\_DEAD\_KEY);  //消息多久后死亡 测试环境给30s  data.put("x-message-ttl", 300 \* 1000);  return new Queue(QueueConstant.ORDER\_QUEUE, true, false, false, data);  }  /\*\*  \* 创建死信队列  \*  \* @return  \*/  @Bean  public Queue orderDeadQueue() {  return new Queue(QueueConstant.ORDER\_DEAD\_QUEUE);  }  /\*\*  \* 创建死信交换机  \*  \* @return  \*/  @Bean  public DirectExchange orderDeadEx() {  return new DirectExchange(QueueConstant.ORDER\_DEAD\_EXCHANGE);  }  /\*\*  \* 绑定私信队列和交换机  \*  \* @return  \*/  @Bean  public Binding orderDeadBind() {  return BindingBuilder.bind(orderDeadQueue()).to(orderDeadEx()).with(QueueConstant.ORDER\_DEAD\_KEY);  }  } |

#### 创建OrderController

|  |
| --- |
| @RestController  @Api(tags = "订单管理系统")  public class OrderController {  @Autowired  private OrderService orderService;  @GetMapping("/p/myOrder/orderCount")  @ApiOperation("查询几种状态订单的数量")  public ResponseEntity<OrderResult> getOrderCount() {  String userId = SecurityContextHolder.getContext().getAuthentication().getPrincipal().toString();  OrderResult orderResult = orderService.selectOrderStatusCount(userId);  return ResponseEntity.ok(orderResult);  }  @GetMapping("/p/myOrder/myOrder")  @ApiOperation("分页查询我的订单详情")  public ResponseEntity<IPage<Order>> findOrderByPage(Page<Order> page, Order order) {  String userId = SecurityContextHolder.getContext().getAuthentication().getPrincipal().toString();  order.setUserId(userId);  IPage<Order> orderIPage = orderService.findOrderByPage(page, order);  return ResponseEntity.ok(orderIPage);  }  /\*\*  \* 用户确定订单  \* 有两种取到来确认订单  \* 1用户从购物车页面来 那么就有basketIds  \* 2用户直接从商品页面来 就没有basketIds  \*  \* @param orderParam  \* @return  \*/  @PostMapping("/p/order/confirm")  @ApiOperation("订单的确认")  public ResponseEntity<OrderVo> confirmOrder(@RequestBody OrderParam orderParam) {  String userId = SecurityContextHolder.getContext().getAuthentication().getPrincipal().toString();  OrderVo orderVo = orderService.orderConfirm(orderParam, userId);  return ResponseEntity.ok(orderVo);  }  /\*\*  \* 用户完成下订单需要一些列操作，写库，修改es 发微信消息等  \*  \* @return  \*/  @PostMapping("p/order/submit")  @ApiOperation("完成订单支付的功能")  public ResponseEntity<String> toPay() {  String userId = SecurityContextHolder.getContext().getAuthentication().getPrincipal().toString();  String orderNum = orderService.doOrderToPay(userId);  return ResponseEntity.ok(orderNum);  }  } |

#### 修改OrderService

|  |
| --- |
| public interface OrderService extends IService<Order> {  /\*\*  \* 根据用户id查询几种订单状态  \*  \* @param userId  \* @return  \*/  OrderResult selectOrderStatusCount(String userId);  /\*\*  \* 分页查询我的订单  \*  \* @param page  \* @param order  \* @return  \*/  IPage<Order> findOrderByPage(Page<Order> page, Order order);  /\*\*  \* 用户确认订单  \*  \* @param orderParam  \* @return  \*/  OrderVo orderConfirm(OrderParam orderParam, String userId);  /\*\*  \* 用户完成下单，准备去支付  \*  \* @param userId  \* @return  \*/  String doOrderToPay(String userId); |

#### 修改OrderServiceImpl（重点）

|  |
| --- |
| @Service  @Slf4j  public class OrderServiceImpl extends ServiceImpl<OrderMapper, Order> implements OrderService {  @Autowired  private OrderMapper orderMapper;  @Autowired  private OrderItemMapper orderItemMapper;  @Autowired  private OrderMemberFeign orderMemberFeign;  @Autowired  private OrderProductFeign orderProductFeign;  @Autowired  private OrderCartFeign orderCartFeign;  @Autowired  private StringRedisTemplate redisTemplate;  @Autowired  private RabbitTemplate rabbitTemplate;  @Autowired  private OrderSettlementMapper orderSettlementMapper;  /\*\*  \* 根据用户id查询几种订单状态  \* 订单状态 1:待付款 2:待发货 3:待收货 4:待评价 5:成功 6:失败  \*  \* @param userId  \* @return  \*/  @Override  public OrderResult selectOrderStatusCount(String userId) {  //代付款的  Integer unPay = orderMapper.selectCount(new LambdaQueryWrapper<Order>()  .eq(Order::getUserId, userId)  .eq(Order::getStatus, 1)  );  //2:待发货  Integer payed = orderMapper.selectCount(new LambdaQueryWrapper<Order>()  .eq(Order::getUserId, userId)  .eq(Order::getStatus, 2)  );  //3:待收货  Integer consignment = orderMapper.selectCount(new LambdaQueryWrapper<Order>()  .eq(Order::getUserId, userId)  .eq(Order::getStatus, 3)  );  OrderResult orderResult = new OrderResult();  orderResult.setUnPay(unPay);  orderResult.setPayed(payed);  orderResult.setConsignment(consignment);  return orderResult;  }  /\*\*  \* 分页查询我的订单  \* 需要展示具体商品的信息  \*  \* @param page  \* @param order  \* @return  \*/  @Override  @Transactional  public IPage<Order> findOrderByPage(Page<Order> page, Order order) {  //根据创建时间排序  page.addOrder(OrderItem.desc("create\_time"));  IPage<Order> orderIPage = orderMapper.selectPage(page, new LambdaQueryWrapper<Order>()  .eq(Order::getUserId, order.getUserId())  .eq(order.getStatus() != null && order.getStatus() != 0, Order::getStatus, order.getStatus())  );  if (CollectionUtils.isEmpty(orderIPage.getRecords())) {  return orderIPage;  }  //查询具体的商品信息 在orderitem表中，有一个ordernum是唯一和order表对应的  List<Order> records = orderIPage.getRecords();  records.forEach(record -> {  //查询  List<ShopCartItem> shopCartItems = findByOrderNum(record.getOrderNumber());  record.setOrderItemDtos(shopCartItems);  });  return orderIPage;  }  /\*\*  \* 根据orderNum查询订单详情（包括商品）  \*  \* @param orderNumber  \* @return  \*/  private List<ShopCartItem> findByOrderNum(String orderNumber) {  List<com.sxt.domain.OrderItem> orderItems = orderItemMapper.selectList(new LambdaQueryWrapper<com.sxt.domain.OrderItem>()  .eq(com.sxt.domain.OrderItem::getOrderNumber, orderNumber)  .orderByDesc(com.sxt.domain.OrderItem::getRecTime)  );  if (CollectionUtils.isEmpty(orderItems)) {  return Collections.emptyList();  }  List<ShopCartItem> shopCartItems = new ArrayList<>(orderItems.size() \* 2);  //循环组装数据  orderItems.forEach(orderItem -> {  ShopCartItem shopCartItem = new ShopCartItem();  //对象拷贝  BeanUtil.copyProperties(orderItem, shopCartItem, true);  shopCartItems.add(shopCartItem);  });  return shopCartItems;  }  /\*\*  \* 用户确认订单  \* 根据页面上的信息我们需要一些数据  \* 1远程调用用户的收货地址  \* 2创建订单返回对象  \* 3组装订单里面的详细数据  \*  \* @param orderParam  \* @return  \*/  @Override  @Transactional  public OrderVo orderConfirm(OrderParam orderParam, String userId) {  OrderVo orderVo = new OrderVo();  //远程调用会员服务，得到默认的收货地址  UserAddr defaultUserAddr = orderMemberFeign.getDefaultUserAddr(userId).getBody();  orderVo.setUserAddr(defaultUserAddr);  //判断用户从那个取到确定订单的  List<Long> basketIds = orderParam.getBasketIds();  //订单的总金额  BigDecimal total = BigDecimal.ZERO;  //订单的总数量  Integer totalCount = 0;  if (CollectionUtils.isEmpty(basketIds)) {  //从商品直接下单的  orderVo = productToOrderConfirm(orderParam, orderVo, total, totalCount);  } else {  //从购物车下单的  orderVo = cartOrderConfirm(orderVo, basketIds, total, totalCount);  }  //将预订单放在redis里面 30分钟  redisTemplate.opsForValue().set(OrderConstant.PRE\_ORDER + userId, JSON.toJSONString(orderVo), Duration.ofMinutes(30));  return orderVo;  }  /\*\*  \* 商品直接到确认订单  \*  \* @param orderParam  \* @param orderVo  \* @return  \*/  private OrderVo productToOrderConfirm(OrderParam orderParam, OrderVo orderVo, BigDecimal total, Integer totalCount) {  //说明是从商品直接过来的，那么我们通过 prodId，skuId prodCount就可以计算总金额和价格  ShopCartItem orderItem = orderParam.getOrderItem();  if (ObjectUtils.isEmpty(orderItem)) {  //如果也不是从商品直接过来的 就返回了  log.error("用户没从正确的渠道进入订单确认");  return orderVo;  }  //商品数量为  totalCount = orderItem.getBasketCount();  //根据prodId 远程调用得到Prod对象,这个对象里面刚好包含了sku对象  ResponseEntity<Prod> responseEntity = orderProductFeign.loadProd(orderItem.getProdId());  //拿到了Prod  Prod prod = responseEntity.getBody();  if (ObjectUtils.isEmpty(prod)) {  //如果商品id为空 直接返回了  return orderVo;  }  //拿到sku  Sku sku1 = prod.getSkuList().stream()  .filter(sku -> sku.getSkuId().equals(orderItem.getSkuId()))  .collect(Collectors.toList())  .get(0);  //这个时候就可以计算总金额了 商品金额\*商品数量+运费(运费我们就先不管)  total = sku1.getPrice()  .multiply(new BigDecimal(totalCount))  .setScale(2, BigDecimal.ROUND\_HALF\_UP);  //创建一个订单集合  List<ShopCartOrder> shopCartOrders = new ArrayList<>(2);  //创建一个订单对象  ShopCartOrder shopCartOrder = new ShopCartOrder();  shopCartOrder.setTransfee(new BigDecimal(0.00));  //创建一个条目集合  List<ShopCartItem> shopCartItem = new ArrayList<>();  //创建一个条目对象  ShopCartItem shopItem = new ShopCartItem();  //设置一些属性  shopItem.setProdName(prod.getProdName());  shopItem.setPic(prod.getPic());  shopItem.setSkuName(sku1.getSkuName());  shopItem.setPrice(sku1.getPrice().toString());  shopItem.setBasketCount(totalCount);  shopItem.setProdId(prod.getProdId());  shopItem.setChecked(Boolean.TRUE);  //将条目对象添加到条目集合  shopCartItem.add(shopItem);  //将条目集合添加到订单对象  shopCartOrder.setShopCartItemDiscounts(shopCartItem);  //将订单对象添加到订单集合  shopCartOrders.add(shopCartOrder);  //最后添加到返回的结果集里面  orderVo.setShopCartOrders(shopCartOrders);  orderVo.setTotalCount(totalCount);  orderVo.setActualTotal(total);  orderVo.setTotal(total);  return orderVo;  }  /\*\*  \* 购物车确定订单  \*  \* @param orderVo  \* @param basketIds  \* @param total  \* @param totalCount  \* @return  \*/  private OrderVo cartOrderConfirm(OrderVo orderVo, List<Long> basketIds, BigDecimal total, Integer totalCount) {  //创建订单预览集合  List<ShopCartOrder> shopCartOrders = new ArrayList<>();  ShopCartOrder shopCartOrder = new ShopCartOrder();  //设置运费  shopCartOrder.setTransfee(BigDecimal.ZERO);  //设置满减  shopCartOrder.setShopReduce(BigDecimal.ZERO);  //创建商品信息集合  List<ShopCartItem> shopCartItemDiscounts = new ArrayList<>();  //设置订单预览  shopCartOrder.setShopCartItemDiscounts(shopCartItemDiscounts);  //往订单集合中添加商品信息集合  shopCartOrders.add(shopCartOrder);  //下面是从购物车进来的 拿到basketIds远程调用购物车  //拿到prodId和skuId，再远程调用查询商品的具体信息，封装数据  List<Basket> baskets = orderCartFeign.getBasketByIds(basketIds);  if (CollectionUtils.isEmpty(basketIds)) {  return orderVo;  }  //循环购物车集合 拿到prodId，远程调用商品服务，得到sku和价格  for (Basket basket : baskets) {  Prod prod = orderProductFeign.loadProd(basket.getProdId()).getBody();  if (ObjectUtils.isEmpty(prod)) {  return orderVo;  }  List<Sku> skus = prod.getSkuList();  //拿到商品的sku  if (ObjectUtils.isEmpty(skus)) {  return orderVo;  }  Sku sku1 = skus.stream().filter(sku -> sku.getSkuId().equals(basket.getSkuId())).collect(Collectors.toList()).get(0);  if (ObjectUtils.isEmpty(sku1)) {  return orderVo;  }  //计算每一个商品的价格 商品price \* 数量 循环加  total = total.add(sku1.getPrice().multiply(new BigDecimal(basket.getBasketCount())).setScale(2, BigDecimal.ROUND\_HALF\_UP));  //拿到商品的总数量  totalCount += basket.getBasketCount();  //组装对象  //真正的设置商品对象 每次循环就创建一个  ShopCartItem shopCartItem = new ShopCartItem();  shopCartItem.setProdId(prod.getProdId());  shopCartItem.setSkuId(sku1.getSkuId());  shopCartItem.setProdName(prod.getProdName());  shopCartItem.setSkuName(sku1.getSkuName());  shopCartItem.setBasketCount(basket.getBasketCount());  shopCartItem.setPrice(sku1.getPrice().toString());  shopCartItem.setPic(sku1.getPic());  shopCartItem.setChecked(Boolean.TRUE);  //往商品信息集合中添加商品对象  shopCartItemDiscounts.add(shopCartItem);  }  orderVo.setTotalCount(totalCount);  orderVo.setActualTotal(total);  orderVo.setTotal(total);  orderVo.setShopCartOrders(shopCartOrders);  return orderVo;  }  /\*\*  \* 用户从完成下单，准备去支付  \* 1.从redis里面得到orderVo  \* 2.清空购物车等  \* 3.减少sku和prod库存  \* 3.通知es减少库存展示  \* 4.写订单表  \* 5.通知发微信消息  \* 6.死信队列30分钟没支付则取消  \*  \* @param userId  \* @return  \*/  @Override  @Transactional  public String doOrderToPay(String userId) {  //拿到orderVo  String orderStr = redisTemplate.opsForValue().get(OrderConstant.PRE\_ORDER + userId);  OrderVo orderVo = JSON.parseObject(orderStr, OrderVo.class);  //生成一个订单编号，可关注雪花算法  String orderNum = String.valueOf(new Date().getTime());  orderVo.setOrderSn(orderNum);  //远程调用清空购物车的方法 根据userId和skuId  clearCart(userId, orderVo);  //减少sku和prod的库存  Map<Long, Integer> prodStocks = decrStocks(orderVo);  //通知es减少商品库存  rabbitTemplate.convertAndSend(QueueConstant.PROD\_CHANGE\_EXCHANGE, QueueConstant.PROD\_ROUTER\_KEY, JSON.toJSONString(prodStocks));  //写订单表  writeOrder(orderVo, userId);  //给用户发微信消息  sendWechatNotification(orderVo, userId);  //死信队列，30分钟没有支付就取消  cancleOrderDealy(orderVo.getOrderSn());  return orderNum;  }  /\*\*  \* 死信队列  \*  \* @param orderSn  \*/  private void cancleOrderDealy(String orderSn) {  log.info("下单写死信队列");  rabbitTemplate.convertAndSend(QueueConstant.ORDER\_QUEUE, JSON.toJSONString(orderSn));  }  /\*\*  \* 写订单表  \*  \* @param orderVo  \* @param userId  \*/  private void writeOrder(OrderVo orderVo, String userId) {  log.info("下单写订单表");  //order订单表 orderItem订单条目表 OrderSettlement表  List<ShopCartItem> items = orderVo.getShopCartOrders().get(0).getShopCartItemDiscounts();  //拼接名称的  StringBuilder stringBuilder = new StringBuilder();  //先写订单条目表  items.forEach(item -> {  com.sxt.domain.OrderItem orderItem = new com.sxt.domain.OrderItem();  orderItem.setOrderNumber(orderVo.getOrderSn());  orderItem.setProdId(item.getProdId());  orderItem.setPic(item.getPic());  orderItem.setProdName(item.getProdName());  orderItem.setProdCount(item.getBasketCount());  orderItem.setPrice(new BigDecimal(item.getPrice()));  orderItem.setSkuId(item.getSkuId().longValue());  orderItem.setSkuName(item.getSkuName());  orderItem.setProductTotalAmount(orderVo.getActualTotal());  orderItem.setShopId(1L);  orderItem.setRecTime(new Date());  orderItemMapper.insert(orderItem);  //组装名称  stringBuilder.append(item.getProdName() + " ");  });  //写订单表 一个订单里面对应多个订单条目  Order order = new Order();  order.setUserId(userId);  order.setActualTotal(orderVo.getActualTotal());  order.setOrderNumber(orderVo.getOrderSn());  order.setProdName(stringBuilder.toString());  order.setIsPayed(Boolean.FALSE);  order.setCreateTime(new Date());  order.setUpdateTime(new Date());  orderMapper.insert(order);  //写支付表  OrderSettlement settlement = new OrderSettlement();  settlement.setPayTypeName("支付宝");  settlement.setPayType(0);  settlement.setOrderNumber(orderVo.getOrderSn());  settlement.setPayAmount(orderVo.getActualTotal());  settlement.setIsClearing(0);  settlement.setCreateTime(new Date());  settlement.setUserId(userId);  orderSettlementMapper.insert(settlement);  }  /\*\*  \* 给用户发微信消息通知  \*  \* @param orderVo  \* @param userId  \*/  private void sendWechatNotification(OrderVo orderVo, String userId) {  log.info("下单发微信消息通知");  WechatMessage wechatMessage = new WechatMessage();  wechatMessage.setToUser("oy\_5Lv95ANQGqolUcwyRfNI\_1BOQ");  wechatMessage.setUrl("www.baidu.com");  wechatMessage.setTopColor("#FF0000");  //模板id  wechatMessage.setTemplateId("q4gcICKjsQmum926savXmJQqHT1RxE1-xAw9XDvCpK8");  Map<String, Map<String, String>> data = new HashMap<>();  data.put("orderSn", WechatMessage.buildProp(orderVo.getOrderSn(), "#173177"));  data.put("totalMoney", WechatMessage.buildProp(orderVo.getActualTotal().toString(), "#173177"));  data.put("orderDate", WechatMessage.buildProp(DateUtil.format(new Date(), "yyyy-MM-dd HH:mm:ss"), "#173177"));  wechatMessage.setData(data);  rabbitTemplate.convertAndSend(QueueConstant.WECHAT\_MSG\_EXCHANGE, QueueConstant.WECHAT\_MSG\_KEY, JSON.toJSONString(wechatMessage));  }  /\*\*  \* 减少库存  \*  \* @param orderVo  \* @return  \*/  private Map<Long, Integer> decrStocks(OrderVo orderVo) {  log.info("远程调用减少库存");  Map<Long, Integer> map = new HashMap<>();  //一个订单可能有多个prod 和多个sku 所以都要减少  List<ShopCartItem> shopCartItems = orderVo.getShopCartOrders().get(0).getShopCartItemDiscounts();  shopCartItems.forEach(shopCartItem -> {  //拿到prodId skuId count 去远程调用 修改商品表  Long prodId = shopCartItem.getProdId();  Long skuId = shopCartItem.getSkuId();  Integer count = shopCartItem.getBasketCount();  //调用商品服务去修改库存 减少  orderProductFeign.changeStock(prodId, skuId, -count);  //往map装数据，好通知es哈 减少库存  if (map.containsKey(shopCartItem.getProdId())) {  count += shopCartItem.getBasketCount();  }  map.put(prodId, -count);  });  return map;  }  /\*\*  \* 清空购物车  \*  \* @param userId  \* @param orderVo  \*/  private void clearCart(String userId, OrderVo orderVo) {  List<Long> skuIds = new ArrayList<>();  orderVo.getShopCartOrders().forEach(o -> {  o.getShopCartItemDiscounts().forEach(s -> {  skuIds.add(s.getSkuId());  });  });  log.info("远程调用清空购物车");  orderCartFeign.deleteCartByUserId(skuIds, userId);  } |

#### 创建OrderListener

|  |
| --- |
| @Component  @Slf4j  public class OrderListener {  @Autowired  private OrderMapper orderMapper;  @Autowired  private OrderSettlementMapper orderSettlementMapper;  @Autowired  private OrderItemMapper orderItemMapper;  @Autowired  private OrderProductFeign orderProductFeign;  @Autowired  private RabbitTemplate rabbitTemplate;  /\*\*  \* 监听订单取消的队列  \* 如果订单的支付状态为1 说明支付成功了 不做任何变动  \* 如果订单状态不为1  \* 修改订单表关闭状态  \* 修改订单结算表状态为0  \* 回滚prod和sku库存  \* 提醒es增加库存  \*  \* @param channel  \* @param message  \* @param tag  \*/  @RabbitListener(queues = QueueConstant.ORDER\_DEAD\_QUEUE, concurrency = "3-5")  public void handlerOrderCancle(Channel channel, Message message, @Header(AmqpHeaders.DELIVERY\_TAG) long tag) {  String orderSn = JSON.parseObject(new String(message.getBody()), String.class);  log.info("我检测到订单取消了，订单为{}", orderSn);  Order order = orderMapper.selectOne(new LambdaQueryWrapper<Order>()  .eq(Order::getOrderNumber, orderSn)  );  if (ObjectUtils.isEmpty(order)) {  throw new IllegalArgumentException("订单号不存在");  }  if (order.getIsPayed()) {  log.info("这个订单已经支付了，不用管了");  return;  }  //这个订单没有支付，超时了 要改变订单状态，一系列操作  order.setStatus(6);  order.setCloseType(1);  order.setCancelTime(new Date());  orderMapper.updateById(order);  //修改order结算表  OrderSettlement orderSettlement = orderSettlementMapper.selectOne(new LambdaQueryWrapper<OrderSettlement>()  .eq(OrderSettlement::getOrderNumber, orderSn)  );  if (ObjectUtils.isEmpty(orderSettlement)) {  throw new IllegalArgumentException("订单结算表无此订单");  }  orderSettlement.setIsClearing(0);  orderSettlement.setPayStatus(-1);  orderSettlementMapper.updateById(orderSettlement);  //更新商品的库存 增加  //查询orderItem  List<OrderItem> orderItems = orderItemMapper.selectList(new LambdaQueryWrapper<OrderItem>()  .eq(OrderItem::getOrderNumber, orderSn)  );  Map<Long, Integer> map = new HashMap<>();  orderItems.forEach(orderItem -> {  //远程调用增加库存  orderProductFeign.changeStock(orderItem.getProdId(), orderItem.getSkuId(), orderItem.getProdCount());  //组装参数给es  Integer prodCount = orderItem.getProdCount();  if (map.containsKey(orderItem.getProdId())) {  prodCount += orderItem.getProdCount();  }  map.put(orderItem.getProdId(), prodCount);  });  //通知es加库存的  rabbitTemplate.convertAndSend(QueueConstant.PROD\_CHANGE\_EXCHANGE, QueueConstant.PROD\_ROUTER\_KEY, JSON.toJSONString(map));  try {  channel.basicAck(tag, false);  log.info("签收消息");  } catch (IOException e) {  e.printStackTrace();  }  }  } |