尚品汇商城

一、目前存在的问题

1.1 搜索与商品服务的问题

我们思考一下，是否存在问题?

- 商品的原始数据保存在数据库中，增删改查都在数据库中完成。
- 搜索服务数据来源是索引库，如果数据库商品发生变化，索引库数据不能及时更新。

如果我们在后台修改了商品的价格，搜索页面依然是旧的价格，这样显然不对。该如何解决?

这里有两种解决方案:

- 方案 1：每当后台对商品做增删改操作，同时要修改索引库数据
- 方案 2：搜索服务对外提供操作接口，后台在商品增删改后，调用接口

以上两种方式都有同一个严重问题：就是代码耦合，后台服务中需要嵌入搜索和商品页面服务，违背了微服务的独立原则。

所以，我们会通过另外一种方式来解决这个问题：**消息队列**

1.2 订单服务取消订单问题

用户下单后，如果 2 个小时未支付，我们该如何取消订单

- 方案 1：定时任务，定时扫描未支付订单，超过 2 小时自动关闭

- 方案 2：使用延迟队列关闭订单

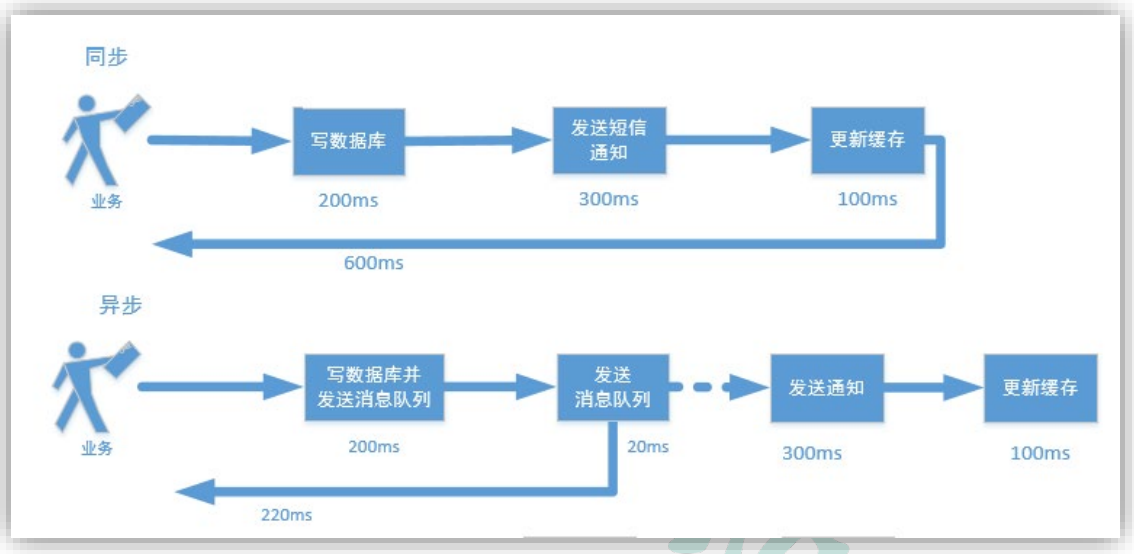
1.3 分布式事务问题

如：用户支付订单，我们如何保证更新订单状态与扣减库存 ， 三个服务数据最终一致!

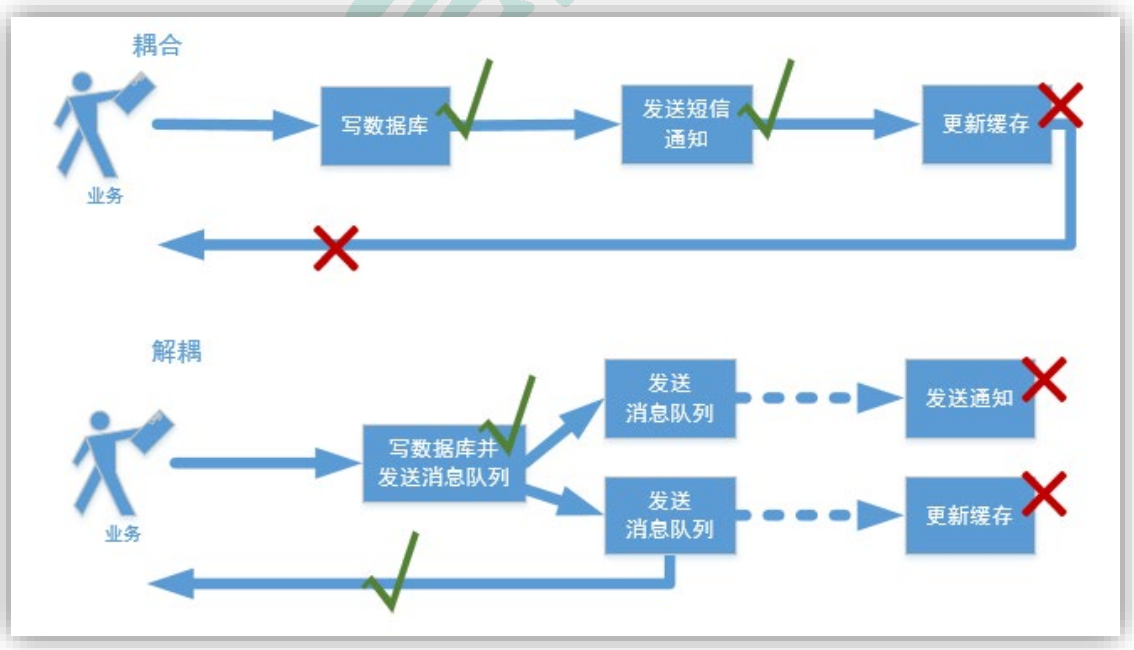
二、消息队列解决什么问题

消息队列都解决了什么问题?

2.1 异步

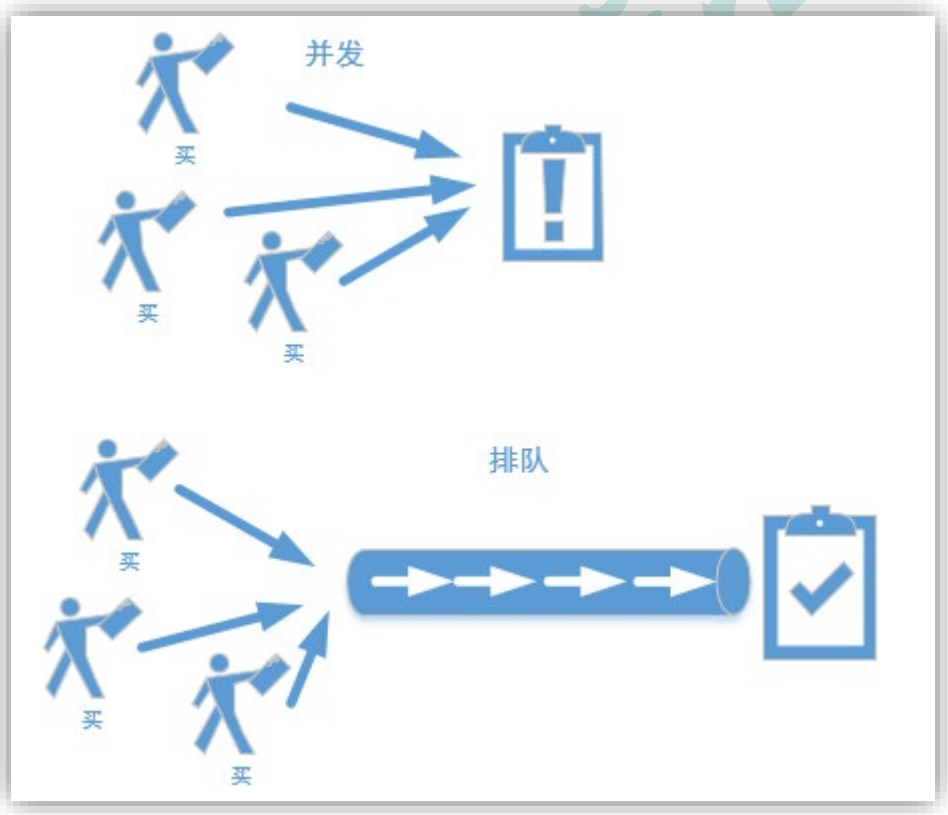

同步
业务
写数据库
200ms
发送短信通知
300ms
更新缓存
100ms
600ms
异步
业务
写数据库并发送消息队列
200ms
发送消息队列
20ms
发送通知
300ms
更新缓存
100ms
220ms

2.2 解耦


耦合
业务
写数据库
发送短信通知
更新缓存
解耦
业务
写数据库并发送消息队列
发送消息队列
发送通知
发送消息队列
更新缓存

2.3 并行

2.4 排队 （流量削峰）



三、消息队列工具 RabbitMQ

3.1 常见 MQ 产品

- ActiveMQ：基于 JMS 协议，java 语言，jdk

- RabbitMQ：基于 AMQP 协议，erlang 语言开发，稳定性好

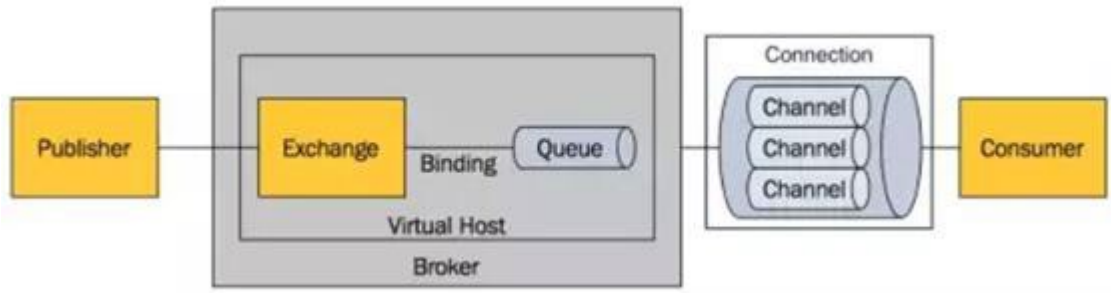
- RocketMQ：基于 JMS，阿里巴巴产品，目前交由 Apache 基金会

- Kafka：分布式消息系统，高吞吐量

优缺点对比参考：

https://baijiahao.baidu.com/s?id=1716633638360672083&wfr=spider&for=pc

3.2 RabbitMQ 基础概念



Broker：简单来说就是消息队列服务器实体

Exchange：消息交换机，它指定消息按什么规则，路由到哪个队列

Queue：消息队列载体，每个消息都会被投入到一个或多个队列

Binding：绑定，它的作用就是把 exchange 和 queue 按照路由规则绑定起来

Routing Key：路由关键字， exchange 根据这个关键字进行消息投递

vhost：虚拟主机，一个 broker 里可以开设多个 vhost，用作不同用户的权限分离

producer：消息生产者，就是投递消息的程序

consumer：消息消费者，就是接受消息的程序

channel：消息通道，在客户端的每个连接里，可建立多个 channel，每个 channel 代表一个会话任务

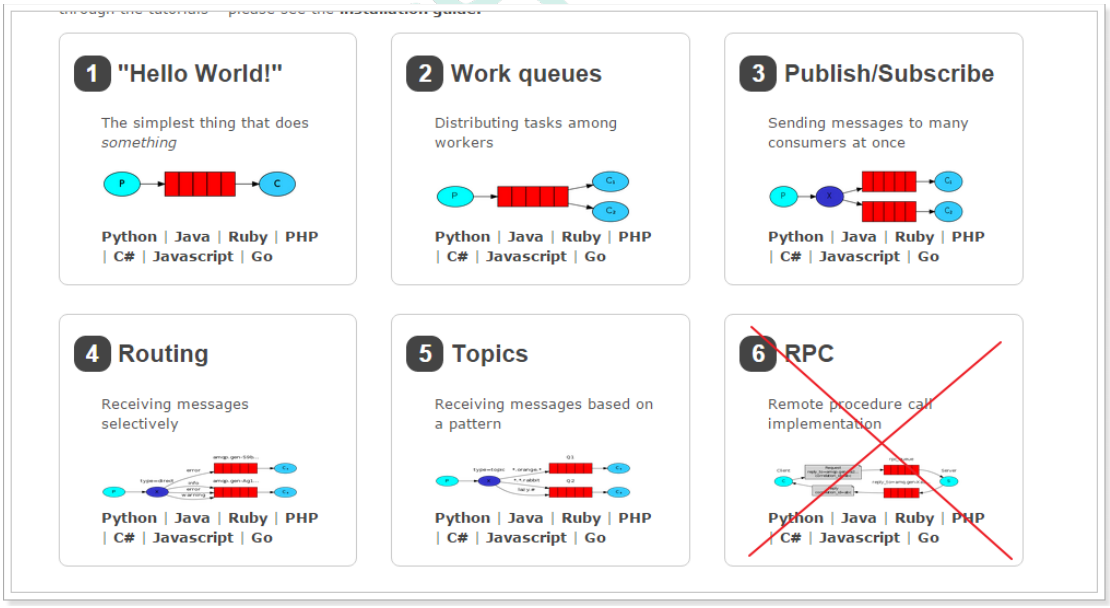
3.3 安装 RabbitMQ 5672

看电商软件环境安装.doc

3.4 五种消息模型

RabbitMQ 提供了 6 种消息模型，但是第 6 种其实是 RPC，并不是 MQ，因此不予学习。那么也就剩下 5 种。

但是其实 3、4、5 这三种都属于订阅模型，只不过进行路由的方式不同。



基本消息模型：生产者–>队列–>消费者

work 消息模型：生产者–>队列–>多个消费者共同消费

订阅模型-Fanout：广播模式，将消息交给所有绑定到交换机的队列，每个消费者都会收到同一条消息

订阅模型-Direct：定向，把消息交给符合指定 **rotingKey** 的队列

订阅模型-Topic 主题模式：通配符，把消息交给符合 routing pattern（路由模式） 的队列

我们项目使用的是第四种!

https://blog.csdn.net/b_just/article/details/106258102

3.5 搭建 mq 测试环境 service-mq

3.5.1 搭建 service-mq 服务

在 service 目录下搭建

3.5.2 修改配置 pom.xml

```
<?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
https://maven.apache.org/xsd/maven-4.0.0.xsd">
   <modelVersion>4.0.0</modelVersion>
   <parent>
      <groupId>com.atguigu.gmall</groupId>
      <artifactId>service</artifactId>
      <version>1.0</version>
   </parent>

   <artifactId>service-mq</artifactId>
   <version>1.0</version>

   <packaging>jar</packaging>
   <name>service-mq</name>
   <description>service-mq</description>

   <build>
      <finalName>service-mq</finalName>
      <plugins>
         <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
         </plugin>
      </plugins>
   </build>
```

```
</project>
```

说明：引入依赖

3.5.3 添加配置文件

说明：rabbitmq 默认端口 5672，注意更改正确

```
bootstrap.properties

spring.application.name=service-mq
spring.profiles.active=dev
spring.cloud.nacos.discovery.server-addr=192.168.200.129:8848
spring.cloud.nacos.config.server-addr=192.168.200.129:8848
spring.cloud.nacos.config.prefix=${spring.application.name}
spring.cloud.nacos.config.file-extension=yaml
spring.cloud.nacos.config.shared-configs[0].data-
id=common.yaml
```

3.5.4 启动类

```
package com.atguigu.gmall.mq;


@SpringBootApplication(exclude                                    =
DataSourceAutoConfiguration.class)//取消数据源自动配置
@ComponentScan({"com.atguigu.gmall"})
@EnableDiscoveryClient
public class ServiceMqApplication {

    public static void main(String[] args) {
        SpringApplication.run(ServiceMqApplication.class, args);
    }
}
```

四、消息不丢失

消息的不丢失，在 MQ 角度考虑，一般有三种途径：

1，生产者不丢数据

2，MQ 服务器不丢数据

3，消费者不丢数据

保证消息不丢失有两种实现方式：

1，开启事务模式

2，**消息确认模式**

说明：开启事务会大幅降低消息发送及接收效率，使用的相对较少，因此我们生产环境一般都采取消息确认模式，以下我们只是讲解消息确认模式

4.1 消息确认

4.1.1 消息持久化

如果希望 RabbitMQ 重启之后消息不丢失，那么需要对以下 3 种实体均配置持久化

Exchange

声明 exchange 时设置持久化（durable = true）并且不自动删除(autoDelete = false)

Queue

声明 queue 时设置持久化（durable = true）并且不自动删除(autoDelete = false)

message

发送消息时通过设置 deliveryMode=2 持久化消息

处理消息队列丢数据的情况，一般是开启持久化磁盘的配置。这个持久化配置可以和 confirm 机制配合使用，你可以在消息持久化磁盘后，再给生产者发送一个 Ack 信号。这样，如果消息持久化磁盘之前，rabbitMQ 阵亡了，那么生产者收不到 Ack 信号，生产者会自动重发。那么如何持久化呢，其实也很容易，就下面两步：

1、将 queue 的持久化标识 durable 设置为 true,则代表是一个持久的队列

2、发送消息的时候将 deliveryMode=2

这样设置以后，rabbitMQ 就算挂了，重启后也能恢复数据

4.1.2 发送确认

有时，业务处理成功，消息也发了，但是我们并不知道消息是否成功到达了 rabbitmq，如果由于网络等原因导致业务成功而消息发送失败，那么发送方将出现不一致的问题，此时可以使用 rabbitmq 的发送确认功能，即要求 rabbitmq 显式告知我们消息是否已成功发送。

4.1.3 手动消费确认

有时，消息被正确投递到消费方，但是消费方处理失败，那么便会出现消费方的不一致问题。比如:订单已创建的消息发送到用户积分子系统中用于增加用户积分，但是积分消费方处理却都失败了，用户就会问：我购买了东西为什么积分并没有增加呢?

要解决这个问题，需要引入消费方确认，即只有消息被成功处理之后才告知 rabbitmq 以 **ack**，否则告知 rabbitmq 以 **nack**

4.2 消息确认业务封装

4.2.1 service-mq 修改配置

开启 rabbitmq 消息确认配置,在 common 的配置文件中都已经配置好了!

```
rabbitmq:
  host: 192.168.200.129
  port: 5672
  username: guest
  password: guest
  publisher-confirms-type: correlated // 交换机的确认
  publisher-returns: true // 队列的确认
  listener:
    simple:
      acknowledge-mode: manual #默认情况下消息消费者是自动确认消息的，如果要手动确认消息则需要修改确认模式为 manual
      prefetch: 1 # 消费者每次从队列获取的消息数量。此属性当不设置时为：轮询分发，设置为 1 为：公平分发
```

4.2.2 搭建 rabbit-util 模块

由于消息队列是公共模块，我们把 mq 的相关业务封装到该模块，其他 service 微服务模块都可能使用，因此我们把他封装到一个单独的模块，需要使用 mq 的模块直接引用该模块即可

搭建方式如：common-util,导入常量类 MqConst

pom.xml

```
<?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>common</artifactId>
        <groupId>com.atguigu.gmall</groupId>
        <version>1.0</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>
```

```
    <artifactId>rabbit-util</artifactId>

    <dependencies>
        <!--rabbitmq 消息队列-->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-actuator</artifactId>
        </dependency>
        <!--rabbitmq 协议-->
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-bus-amqp</artifactId>
        </dependency>
    </dependencies>

</project>
```

4.2.3 service-mq 引入 rabbit-util 模块依赖

```
<!--rabbitmq 消息队列-->
<dependency>
   <groupId>com.atguigu.gmall</groupId>
   <artifactId>rabbit-util</artifactId>
   <version>1.0</version>
</dependency>
```

4.2.4 封装发送端消息确认

在 rabbit-util 中添加类

```
package com.atguigu.gmall.common.config;

/**
 * @Description 消息发送确认
 * <p>
 * ConfirmCallback  只确认消息是否正确到达 Exchange 中
 * ReturnCallback   消息没有正确到达队列时触发回调，如果正确到达队列不执行
 * <p>
```

```
 * 1. 如果消息没有到exchange,则confirm 回调,ack=false
 * 2. 如果消息到达exchange,则confirm 回调,ack=true
 * 3. exchange 到queue 成功,则不回调return
 * 4. exchange 到queue 失败,则回调return
 *
 */
@Component
@Slf4j
public class MQProducerAckConfig implements RabbitTemplate.ConfirmCallback, RabbitTemplate.ReturnCallback {

    @Autowired
    private RabbitTemplate rabbitTemplate;

    // 修饰一个非静态的 void（）方法,在服务器加载 Servlet 的时候运行，并且只会被服务器执行一次在构造函数之后执行，init（）方法之前执行。
    @PostConstruct
    public void init() {
        rabbitTemplate.setConfirmCallback(this);                //指定ConfirmCallback
        rabbitTemplate.setReturnCallback(this);                 //指定ReturnCallback
    }

    @Override
    public void confirm(CorrelationData correlationData, boolean ack, String cause) {
        if (ack) {
            log.info("消息发送成功：" + JSON.toJSONString(correlationData));
        } else {
            log.info("消息发送失败：" + cause + "数据：" + JSON.toJSONString(correlationData));
        }
    }

    @Override
    public void returnedMessage(Message message, int replyCode, String replyText, String exchange, String routingKey) {
        // 反序列化对象输出
     System.out.println("消息主体：" + new String(message.getBody()));
        System.out.println("应答码：" + replyCode);
        System.out.println("描述：" + replyText);
        System.out.println("消息使用的交换器 exchange ：" + exchange);
        System.out.println("消息使用的路由键 routing ：" + routingKey);
```

```
    }

}
```

4.2.5 封装消息发送

在 rabbit-util 中添加类

```
package com.atguigu.gmall.common.service;


@Service
public class RabbitService {

    @Autowired
    private RabbitTemplate rabbitTemplate;

    /**
     *  发送消息
    * @param exchange 交换机
    * @param routingKey 路由键
   * @param message 消息
    */
    public boolean sendMessage(String exchange, String routingKey,
Object message) {

        rabbitTemplate.convertAndSend(exchange,          routingKey,
message);
        return true;
    }

}
```

4.2.6 发送确认消息测试

在 service-mq 编写测试代码

消息发送端

```
package com.atguigu.gmall.mq.controller;
```

```
@RestController
@RequestMapping("/mq")
public class MqController {


    @Autowired
    private RabbitService rabbitService;


    /**
     * 消息发送
    */
    //http://localhost:8282/mq/sendConfirm
    @GetMapping("sendConfirm")
    public Result sendConfirm() {

        rabbitService.sendMessage("exchange.confirm",
"routing.confirm", "来人了，开始接客吧！");
        return Result.ok();
    }

}
```

消息接收端

在 service-mq 中编写

```
package com.atguigu.gmall.mq.receiver;


@Component
public class ConfirmReceiver {

@SneakyThrows
@RabbitListener(bindings=@QueueBinding(
        value = @Queue(value = "queue.confirm",autoDelete =
"false"),
        exchange = @Exchange(value = "exchange.confirm",autoDelete =
"true"),
        key = {"routing.confirm"}))
public void process(Message message, Channel channel){
    System.out.println("RabbitListener:"+new
String(message.getBody()));

        // false 确认一个消息，true 批量确认
channel.basicAck(message.getMessageProperties().getDeliveryTag(),fal
se);

}
```

```
}
```

测试：http://localhost:8282/mq/sendConfirm

4.2.7 消息发送失败，设置重发机制

实现思路：借助 redis 来实现重发机制

在 rabbit-util 模块中添加依赖

```
<!-- redis -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>

<!-- spring2.X 集成 redis 所需 common-pool2-->
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-pool2</artifactId>
</dependency>

<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>fastjson</artifactId>
</dependency>
```

自定义一个实体类来接收消息

```
@Data
public class GmallCorrelationData extends CorrelationData {

    //  消息主体
    private Object message;
    //  交换机
    private String exchange;
    //  路由键
    private String routingKey;
    //  重试次数
    private int retryCount = 0;
    //  消息类型  是否是延迟消息
    private boolean isDelay = false;
```

```
    //  延迟时间
    private int delayTime = 10;
}
```

修改发送方法

```
//  封装一个发送消息的方法
public  Boolean  sendMsg(String  exchange,String  routingKey,
Object msg){
    //  将发送的消息 赋值到 自定义的实体类
    GmallCorrelationData    gmallCorrelationData    =    new
GmallCorrelationData();
    //  声明一个 correlationId 的变量
    String                  correlationId                   =
UUID.randomUUID().toString().replaceAll("-","");
    gmallCorrelationData.setId(correlationId);
    gmallCorrelationData.setExchange(exchange);
    gmallCorrelationData.setRoutingKey(routingKey);
    gmallCorrelationData.setMessage(msg);

    //  发送消息的时候，将这个 gmallCorrelationData 对象放入缓存。
    redisTemplate.opsForValue().set(correlationId,
JSON.toJSONString(gmallCorrelationData),10, TimeUnit.MINUTES);
    //  调用发送消息方法
this.rabbitTemplate.convertAndSend(exchange,routingKey,msg);

this.rabbitTemplate.convertAndSend(exchange,routingKey,msg,gma
llCorrelationData);
    //  默认返回 true
    return true;
}
```

发送失败调用重发方法 MQProducerAckConfig 类中修改

```
@Override
public  void  confirm(CorrelationData  correlationData,  boolean
ack, String cause) {
    //  ack = true 说明消息正确发送到了交换机
    if (ack){
        System.out.println("哥们你来了.");
        log.info("消息发送到了交换机");
    }else {
        //  消息没有到交换机
        log.info("消息没发送到交换机");
        //  调用重试发送方法
```

```
            this.retrySendMsg(correlationData);
        }
    }


    @Override
    public void returnedMessage(Message message, int code, String
    codeText, String exchange, String routingKey) {
        System.out.println("消息主体：" + new
    String(message.getBody()));
        System.out.println("应答码: " + code);
        System.out.println("描述：" + codeText);
        System.out.println("消息使用的交换器 exchange ： " +
    exchange);
        System.out.println("消息使用的路由键 routing ： " +
    routingKey);

        //      获取这个 CorrelationData 对象的 Id
    spring_returned_message_correlation
        String          correlationDataId          =          (String)
    message.getMessageProperties().getHeaders().get("spring_return
    ed_message_correlation");
        //  因为在发送消息的时候，已经将数据存储到缓存，通过
    correlationDataId 来获取缓存的数据
        String              strJson              =              (String)
    this.redisTemplate.opsForValue().get(correlationDataId);
        //  消息没有到队列的时候，则会调用重试发送方法
        GmallCorrelationData          gmallCorrelationData          =
    JSON.parseObject(strJson,GmallCorrelationData.class);
        //  调用方法 gmallCorrelationData 这对象中，至少的有，交换
    机，路由键，消息等内容.
        this.retrySendMsg(gmallCorrelationData);
    }


    /**
     * 重试发送方法
     * @param correlationData    父类对象   它下面还有个子类对象
    GmallCorrelationData
     */
    private void retrySendMsg(CorrelationData correlationData) {
        //  数据类型转换  统一转换为子类处理
        GmallCorrelationData          gmallCorrelationData          =
    (GmallCorrelationData) correlationData;
        //  获取到重试次数 初始值 0
```

```
    int retryCount = gmallCorrelationData.getRetryCount();
    //  判断
    if (retryCount>=3){
        //  不需要重试了
        log.error("重试次数已到，发送消息失败:"+JSON.toJSONString(gmallCorrelationData));
    } else {
        //  变量更新
        retryCount+=1;
        //  重新赋值重试次数 第一次重试 0->1 1->2 2->3
        gmallCorrelationData.setRetryCount(retryCount);
        System.out.println("重试次数：\t"+retryCount);

        //  更新缓存中的数据

this.redisTemplate.opsForValue().set(gmallCorrelationData.getId(),JSON.toJSONString(gmallCorrelationData),10,
TimeUnit.MINUTES);

        //  调用发送消息方法 表示发送普通消息  发送消息的时候，不能调用 new RabbitService().sendMsg() 这个方法

this.rabbitTemplate.convertAndSend(gmallCorrelationData.getExchange(),gmallCorrelationData.getRoutingKey(),gmallCorrelationData.getMessage(),gmallCorrelationData);

    }
}
```

4.3 改造商品搜索上下架

4.3.1 定义商品上下架常量

在 rabbit-util 模块中导入常量类 MqConst。

```
/**
 * 商品上下架.
 */
public static final String EXCHANGE_DIRECT_GOODS = "exchange.direct.goods";
public static final String ROUTING_GOODS_UPPER = "goods.upper";
public static final String ROUTING_GOODS_LOWER = "goods.lower";
```

```
//队列
public static final String QUEUE_GOODS_UPPER  = "queue.goods.upper";
public static final String QUEUE_GOODS_LOWER  = "queue.goods.lower";
```

4.3.2 service-list 与 service-product 引入依赖与配置

```
<!--rabbitmq 消息队列-->
<dependency>
    <groupId>com.atguigu.gmall</groupId>
    <artifactId>rabbit-util</artifactId>
    <version>1.0</version>
</dependency>
```

4.3.3 service-product 发送消息

我在商品上架与商品添加时发送消息

商品上架

实现类

```
@Override
@Transactional
public void onSale(Long skuId) {
    // 更改销售状态
  SkuInfo skuInfoUp = new SkuInfo();
    skuInfoUp.setId(skuId);
    skuInfoUp.setIsSale(1);
    skuInfoMapper.updateById(skuInfoUp);

    //商品上架
   rabbitService.sendMessage(MqConst.EXCHANGE_DIRECT_GOODS,
MqConst.ROUTING_GOODS_UPPER, skuId);
}
```

商品下架

实现类

```
@Override
@Transactional
public void cancelSale(Long skuId) {
    // 更改销售状态
  SkuInfo skuInfoUp = new SkuInfo();
    skuInfoUp.setId(skuId);
    skuInfoUp.setIsSale(0);
    skuInfoMapper.updateById(skuInfoUp);

    //商品下架
   rabbitService.sendMessage(MqConst.EXCHANGE_DIRECT_GOODS,
MqConst.ROUTING_GOODS_LOWER, skuId);
}
```

4.3.4 service-list 消费消息

```
package com.atguigu.gmall.list.receiver;

@Component
public class ListReceiver {

//  开启消息监听 监听商品上架!
@SneakyThrows
@RabbitListener(bindings = @QueueBinding(
        value = @Queue(value = MqConst.QUEUE_GOODS_UPPER,durable =
"true",autoDelete = "false"),
        exchange = @Exchange(value = MqConst.EXCHANGE_DIRECT_GOODS),
        key = {MqConst.ROUTING_GOODS_UPPER}
))
public void upperGoodsToEs(Long skuId, Message message, Channel
channel){
    //  获取到skuId,并判断
  try {
        if (skuId!=null){
            //  则调用商品上架的方法!
      searchService.upperGoods(skuId);
        }
    } catch (Exception e) {
        //  写入日志或将这条消息写入数据库，短信接口
    e.printStackTrace();
    }
    //  确认消费者消费消息!

channel.basicAck(message.getMessageProperties().getDeliveryTag(),false)
;
}
```

```
//  编写商品下架代码
@SneakyThrows
@RabbitListener(bindings = @QueueBinding(
        value = @Queue(value = MqConst.QUEUE_GOODS_LOWER,durable = "true",autoDelete = "false"),
        exchange = @Exchange(value = MqConst.EXCHANGE_DIRECT_GOODS),
        key = {MqConst.ROUTING_GOODS_LOWER}
))
public void lowerGoodsToEs (Long skuId, Message message, Channel channel){
    try {
        //  判断skuId
        if (skuId!=null){
            //  调用商品下架方法
      searchService.lowerGoods(skuId);
        }
    } catch (Exception e) {
        //  写入日志或将这条消息写入数据库，短信接口
    e.printStackTrace();
    }
    //  消息确认

channel.basicAck(message.getMessageProperties().getDeliveryTag(),false);
}

}
```

4.3.5 测试

启动后台管理页面

http://localhost:8888/#/product/sku/list

操作商品的上架，下架。动态更改 es 中的数据。

可以通过 http://192.168.200.128:5601/app/kibana#/dev_tools/console?_g=() 观察功能是否实现!

五、延迟消息

延迟消息有两种实现方案：

1，基于死信队列

2，集成延迟插件

5.1 基于死信实现延迟消息

使用 RabbitMQ 来实现延迟消息必须先了解 RabbitMQ 的两个概念：消息的 TTL 和死信 Exchange，通过这两者的组合来实现延迟队列

5.1.1 消息的 TTL（Time To Live）

消息的 TTL 就是消息的存活时间。RabbitMQ 可以对队列和消息分别设置 TTL。对队列设置就是队列没有消费者连着的保留时间，也可以对每一个单独的消息做单独的设置。超过了这个时间，我们认为这个消息就死了，称之为死信。

如何设置 TTL：

我们创建一个队列 queue.temp，在 Arguments 中添加 x-message-ttl 为 5000 （单位是毫秒），那所在压在这个队列的消息在 5 秒后会消失。

5.1.2 死信交换机 Dead Letter Exchanges

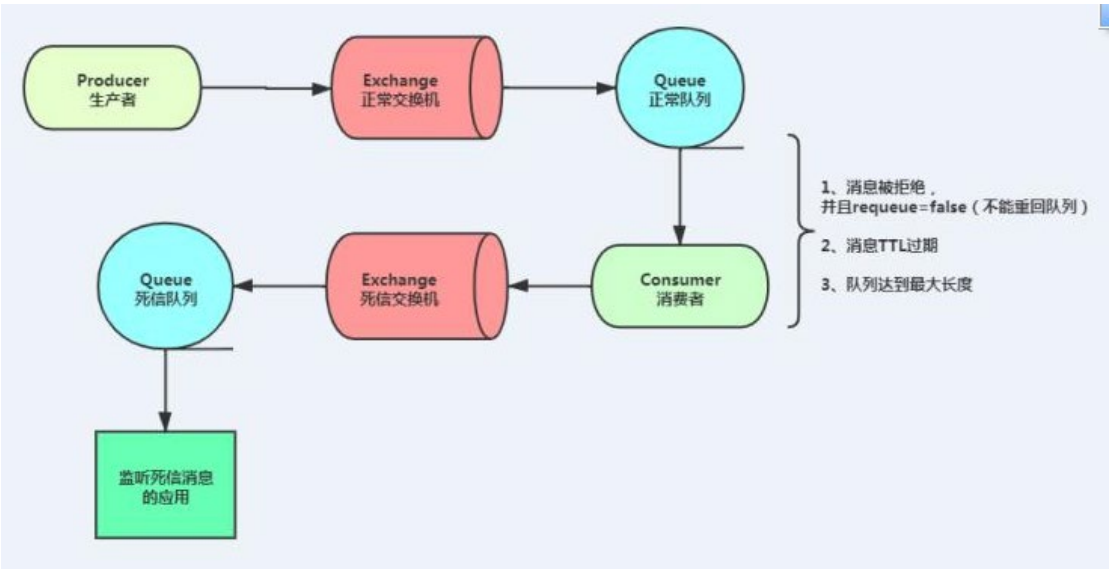
一个消息在满足如下条件下，会进死信路由，记住这里是路由而不是队列，一个路由可以对应很多队列。

（1） 一个消息被 Consumer 拒收了，并且 reject 方法的参数里 requeue 是 false。也就是说不会被再次放在队列里，被其他消费者使用。

（2）上面的消息的 TTL 到了，消息过期了。

（3）队列的长度限制满了。排在前面的消息会被丢弃或者扔到死信路由上。

Dead Letter Exchange 其实就是一种普通的 exchange，和创建其他 exchange 没有两样。只是在某一个设置 Dead Letter Exchange 的队列中有消息过期了，会自动触发消息的转发，发送到 Dead Letter Exchange 中去。



我们现在可以测试一下延迟队列。

（1）创建死信队列

（2）创建交换机

（3）建立交换器与队列之间的绑定

（4）创建队列

5.1.3 代码实现

5.1.3.1 在 service-mq 中添加配置类

```
package com.atguigu.gmall.mq.config;
import org.springframework.amqp.core.DirectExchange;
import org.springframework.amqp.core.Queue;
```

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;


@Configuration
public class DeadLetterMqConfig {
    // 声明一些变量

    public static final String exchange_dead = "exchange.dead";
    public static final String routing_dead_1 = "routing.dead.1";
    public static final String routing_dead_2 = "routing.dead.2";
    public static final String queue_dead_1 = "queue.dead.1";
    public static final String queue_dead_2 = "queue.dead.2";

    // 定义交换机
  @Bean
    public DirectExchange exchange(){
        return new DirectExchange(exchange_dead,true,false,null);
    }

    @Bean
    public Queue queue1(){
        // 设置如果队列一 出现问题，则通过参数转到exchange_dead，routing_dead_2
上！
    HashMap<String, Object> map = new HashMap<>();
        // 参数绑定 此处的key 固定值，不能随意写
    map.put("x-dead-letter-exchange",exchange_dead);
        map.put("x-dead-letter-routing-key",routing_dead_2);
        // 设置延迟时间
    map.put("x-message-ttl", 10 * 1000);
        // 队列名称，是否持久化，是否独享、排外的【true:只可以在本次连接中访
问】，是否自动删除，队列的其他属性参数
    return new Queue(queue_dead_1,true,false,false,map);
    }

    @Bean
    public Binding binding(){
        // 将队列一 通过routing_dead_1 key 绑定到exchange_dead 交换机上
    return
BindingBuilder.bind(queue1()).to(exchange()).with(routing_dead_1);
    }

    // 这个队列二就是一个普通队列
  @Bean
    public Queue queue2(){
        return new Queue(queue_dead_2,true,false,false,null);
    }

    // 设置队列二的绑定规则
  @Bean
    public Binding binding2(){
        // 将队列二通过routing_dead_2 key 绑定到exchange_dead 交换机上！
    return
BindingBuilder.bind(queue2()).to(exchange()).with(routing_dead_2);
    }
}
```

5.1.3.2 配置发送消息

```
package com.atguigu.gmall.mq.controller;


@RestController
@RequestMapping("/mq")
@Slf4j
public class MqController {

    @Autowired
    private RabbitTemplate rabbitTemplate;

    @Autowired
    private RabbitService rabbitService;

 @GetMapping("sendDeadLettle")
    public Result sendDeadLettle() {
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd
HH:mm:ss");

this.rabbitTemplate.convertAndSend(DeadLetterMqConfig.exchange_dead,
DeadLetterMqConfig.routing_dead_1, "ok");
        System.out.println(sdf.format(new Date()) + " Delay sent.");
        return Result.ok();
    }
}
```

5.1.3.3 消息接收方

```
package com.atguigu.gmall.mq.receiver;

@Component
@Configuration
public class DeadLetterReceiver {

    @RabbitListener(queues = DeadLetterMqConfig.queue_dead_2)
    public void get(String msg) {
        System.out.println("Receive:" + msg);
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd
HH:mm:ss");
        System.out.println("Receive queue_dead_2: " + sdf.format(new
Date()) + " Delay rece." + msg);
    }
}
```

5.2 基于延迟插件实现延迟消息

Rabbitmq 实现了一个插件 x-delay-message 来实现延时队列

5.2.1 插件安装

1. 首先我们将刚下载下来的 rabbitmq_delayed_message_exchange-3.9.0.ez 文件上传到 RabbitMQ 所在服务器，下载地址：https://www.rabbitmq.com/community-plugins.html

2. 切换到插件所在目录，执行 docker cp rabbitmq_delayed_message_exchange-3.9.0.ez rabbitmq:/plugins 命令，将刚插件拷贝到容器内 plugins 目录下

3. 执行 docker exec -it rabbitmq /bin/bash 命令进入到容器内部，并 cd plugins 进入 plugins 目录

4. 执行 ls -l|grep delay 命令查看插件是否 copy 成功

5. 在容器内 plugins 目录下，执行 rabbitmq-plugins enable rabbitmq_delayed_message_exchange 命令启用插件

6. exit 命令退出 RabbitMQ 容器内部，然后执行 docker restart rabbitmq 命令重启 RabbitMQ 容器

5.2.2 代码实现

在 service-mq 中添加类

配置队列

```
package com.atguigu.gmall.mq.config;

@Configuration
public class DelayedMqConfig {

    public static final String exchange_delay = "exchange.delay";
    public static final String routing_delay = "routing.delay";
    public static final String queue_delay_1 = "queue.delay.1";

     @Bean
```

```
    public Queue delayQeue1() {
        // 第一个参数是创建的 queue 的名字，第二个参数是是否支持持久化
     return new Queue(queue_delay_1, true);
    }

    @Bean
    public CustomExchange delayExchange() {
        Map<String, Object> args = new HashMap<String, Object>();
        args.put("x-delayed-type", "direct");
        return new CustomExchange(exchange_delay, "x-delayed-
message", true, false, args);
    }

    @Bean
    public Binding delayBbinding1() {
        return
BindingBuilder.bind(delayQeue1()).to(delayExchange()).with(routing_d
elay).noargs();
    }
}
```

发送消息

```
@GetMapping("sendelay")
public Result sendDelay() {
   SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd
HH:mm:ss");

this.rabbitTemplate.convertAndSend(DelayedMqConfig.exchange_delay,
DelayedMqConfig.routing_delay, sdf.format(new Date()), new
MessagePostProcessor() {
      @Override
      public Message postProcessMessage(Message message) throws
AmqpException {
         message.getMessageProperties().setDelay(10 * 1000);
         System.out.println(sdf.format(new Date()) + " Delay
sent.");
         return message;
      }
   });
   return Result.ok();
}
```

接收消息

```
package com.atguigu.gmall.mq.receiver;
```

```
@Component
public class DelayReceiver {

    @RabbitListener(queues = DelayedMqConfig.queue_delay_1)
    public void get(String msg) {
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd
HH:mm:ss");
        System.out.println("Receive queue_delay_1: " +
sdf.format(new Date()) + " Delay rece." + msg);
    }

}
```

5.3 基于延迟插件实现取消订单

service-order 模块

5.3.1 业务配置与接口封装

rabbit-util 模块配置常量 MqConst

```
/**
 * 取消订单，发送延迟队列
 */
public static final String EXCHANGE_DIRECT_ORDER_CANCEL =
"exchange.direct.order.cancel";//"exchange.direct.order.create"
test_exchange;
public static final String ROUTING_ORDER_CANCEL = "order.create";
//延迟取消订单队列
public static final String QUEUE_ORDER_CANCEL =
"queue.order.cancel";
//取消订单 延迟时间 单位：秒
public static final int DELAY_TIME = 10;
```

rabbit-util 模块延迟接口封装

RabbitService

```
/**
 * 封装发送延迟消息方法
 * @param exchange
 * @param routingKey
```

```
 * @param msg
 * @param delayTime
 * @return
 */
public Boolean sendDelayMsg(String exchange,String routingKey,
Object msg, int delayTime){
    //  将发送的消息 赋值到 自定义的实体类
    GmallCorrelationData gmallCorrelationData = new
GmallCorrelationData();
    //  声明一个 correlationId 的变量
    String correlationId =
UUID.randomUUID().toString().replaceAll("-","");
    gmallCorrelationData.setId(correlationId);
    gmallCorrelationData.setExchange(exchange);
    gmallCorrelationData.setRoutingKey(routingKey);
    gmallCorrelationData.setMessage(msg);
    gmallCorrelationData.setDelayTime(delayTime);
    gmallCorrelationData.setDelay(true);

    //  将数据存到缓存

this.redisTemplate.opsForValue().set(correlationId,JSON.toJSON
String(gmallCorrelationData),10,TimeUnit.MINUTES);

    //  发送消息

this.rabbitTemplate.convertAndSend(exchange,routingKey,msg,mes
sage -> {
        //  设置延迟时间

message.getMessageProperties().setDelay(delayTime*1000);
        return message;
    },gmallCorrelationData);

    //  默认返回
    return true;
}
```

修改 retrySendMsg 方法 - 添加判断是否属于延迟消息

```
//  判断是否属于延迟消息
if (gmallCorrelationData.isDelay()){
    //  属于延迟消息

this.rabbitTemplate.convertAndSend(gmallCorrelationData.getExc
hange(),gmallCorrelationData.getRoutingKey(),gmallCorrelationD
ata.getMessage(),message -> {
        //  设置延迟时间
```

```
message.getMessageProperties().setDelay(gmallCorrelationData.getDelayTime()*1000);
        return message;
    },gmallCorrelationData);
}else {
    //  调用发送消息方法 表示发送普通消息  发送消息的时候，不能调用 new RabbitService().sendMsg() 这个方法

this.rabbitTemplate.convertAndSend(gmallCorrelationData.getExchange(),gmallCorrelationData.getRoutingKey(),gmallCorrelationData.getMessage(),gmallCorrelationData);
}
```

利用封装好的工具类 测试发送延迟消息

```
//  基于延迟插件的延迟消息
@GetMapping("sendDelay")
public Result sendDelay(){
    //  声明一个时间对象
    SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
    System.out.println("发送时间：+simpleDateFormat.format(new Date()));

this.rabbitService.sendDelayMsg(DelayedMqConfig.exchange_delay,DelayedMqConfig.routing_delay,"iuok",3);
    return Result.ok();
}
```

结果会 回发送三次，也被消费三次!

如何保证消息幂等性?

1. 使用数据方式

2. 使用 redis setnx 命令解决 --- 推荐

```
@SneakyThrows
@RabbitListener(queues = DelayedMqConfig.queue_delay_1)
public void getMsg2(String msg,Message message,Channel channel){

    //  使用setnx 命令来解决 msgKey = delay:iuok
    String msgKey = "delay:"+msg;
```

```
    Boolean result = this.redisTemplate.opsForValue().setIfAbsent(msgKey, "0", 10, TimeUnit.MINUTES);
    //  result = true : 说明执行成功，redis 里面没有这个 key ，第一次创建， 第一次消费。
    //  result = false : 说明执行失败，redis 里面有这个key
    //  不能： 那么就表示这个消息只能被消费一次！ 那么第一次消费成功或失败，我们确定不了！ --- 只能被消费一次！
        //        if (result){
        //            SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        //            System.out.println("接收时间："+simpleDateFormat.format(new Date()));
        //            System.out.println("接收的消息："+msg);
        //            //  手动确认消息
        //            channel.basicAck(message.getMessageProperties().getDeliveryTag(),false);
        //        } else {
        //            //    不能消费！
        //        }
    //  能： 保证消息被消费成功     第二次消费，可以进来，但是要判断上一个消费者，是否将消息消费了。如果消费了，则直接返回，如果没有消费成功，我消费。
    //  在设置 key 的时候给了一个默认值 0 ，如果消费成功，则将 key 的值 改为1
    if (!result){
        //  获取缓存key 对应的数据
        String status = (String) this.redisTemplate.opsForValue().get(msgKey);
        if ("1".equals(status)){
            //  手动确认
            channel.basicAck(message.getMessageProperties().getDeliveryTag(),false);
            return;
        } else {
            //  说明第一个消费者没有消费成功，所以消费并确认
            SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
            System.out.println("接收时间："+simpleDateFormat.format(new Date()));
            System.out.println("接收的消息："+msg);
            //  修改redis 中的数据
            this.redisTemplate.opsForValue().set(msgKey,"1");
```

```
channel.basicAck(message.getMessageProperties().getDeliveryTag(),false);
            return;
        }
    }
    SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
    System.out.println("接收时间:"+simpleDateFormat.format(new Date()));
    System.out.println("接收的消息："+msg);

    //  修改redis 中的数据
    this.redisTemplate.opsForValue().set(msgKey,"1");
    //  手动确认消息

channel.basicAck(message.getMessageProperties().getDeliveryTag(),false);
}
```

5.3.2 改造订单 service-order 模块

service-order 模块配置队列

添加依赖

```
<!--rabbitmq 消息队列-->
<dependency>
    <groupId>com.atguigu.gmall</groupId>
    <artifactId>rabbit-util</artifactId>
    <version>1.0</version>
</dependency>
```

```
package com.atguigu.gmall.order.receiver;


@Configuration
public class OrderCanelMqConfig {

    @Bean
    public Queue delayQueue() {
        // 第一个参数是创建的 queue 的名字，第二个参数是是否支持持久化
     return new Queue(MqConst.QUEUE_ORDER_CANCEL, true);
```

```
    }

    @Bean
    public CustomExchange delayExchange() {
        Map<String, Object> args = new HashMap<String, Object>();
        args.put("x-delayed-type", "direct");
        return                                                    new
CustomExchange(MqConst.EXCHANGE_DIRECT_ORDER_CANCEL,      "x-delayed-
message", true, false, args);
    }

    @Bean
    public Binding bindingDelay() {
        return
BindingBuilder.bind(delayQueue()).to(delayExchange()).with(MqConst.R
OUTING_ORDER_CANCEL).noargs();
    }
}
```

5.3.3 发送消息

创建订单时，发送延迟消息

修改保存订单方法

```
@Override
@Transactional
public Long saveOrderInfo(OrderInfo orderInfo) {
    .....
    //发送延迟队列，如果定时未支付，取消订单
 rabbitService.sendDelayMessage(MqConst.EXCHANGE_DIRECT_ORDER_CANCEL,
MqConst.ROUTING_ORDER_CANCEL,                    orderInfo.getId(),
MqConst.DELAY_TIME);
    // 返回
   return orderInfo.getId();
}
```

5.3.4 接收消息

```
package com.atguigu.gmall.order.receiver;


@Component
public class OrderReceiver {

    @Autowired
```

```
    private OrderService orderService;

    //  监听的消息
   @SneakyThrows
    @RabbitListener(queues = MqConst.QUEUE_ORDER_CANCEL)
    public void cancelOrder(Long orderId , Message message, Channel channel){
        //  判断当前订单 Id 不能为空
   try {
            if (orderId!=null){
                //  发过来的是订单 Id，那么你就需要判断一下当前的订单是否已经支付了。
      //  未支付的情况下：关闭订单
      //  根据订单 Id 查询 orderInfo select * from order_info where id = orderId
                //  利用这个接口 IService  实现类 ServiceImpl 完成根据订单 Id 查询订单信
息 ServiceImpl 类底层还是使用的 mapper
                OrderInfo orderInfo = orderService.getById(orderId);
                //  判断支付状态,进度状态
      if (orderInfo!=null && "UNPAID".equals(orderInfo.getOrderStatus())
                        && "UNPAID".equals(orderInfo.getProcessStatus())){
                    //  关闭订单
       //  int i = 1/0;
                    orderService.execExpiredOrder(orderId);
                }
            }
        } catch (Exception e) {
            //  消息没有正常被消费者处理：记录日志后续跟踪处理!

    e.printStackTrace();
        }
        //  手动确认消息 如果不确认，有可能会到消息残留。
   channel.basicAck(message.getMessageProperties().getDeliveryTag(),false);
    }
}
```

5.3.5 编写取消订单接口与实现类

```
/**
 * 处理过期订单
 * @param orderId
 */
void execExpiredOrder(Long orderId);

/**
 * 根据订单 Id 修改订单的状态
 * @param orderId
 * @param processStatus
 */
void updateOrderStatus(Long orderId, ProcessStatus processStatus);
```

```
@Override
public void execExpiredOrder(Long orderId) {
    // orderInfo
```

```
    updateOrderStatus(orderId, ProcessStatus.CLOSED);
}

@Override
public void updateOrderStatus(Long orderId, ProcessStatus
processStatus) {
    OrderInfo orderInfo = new OrderInfo();
    orderInfo.setId(orderId);
    orderInfo.setProcessStatus(processStatus.name());
    orderInfo.setOrderStatus(processStatus.getOrderStatus().name());


    orderInfoMapper.updateById(orderInfo);
}
```