尚品汇商城复习

版本：V 1.0

RabbitMQ 应用

一、项目中的问题

1、 搜索与商品服务的问题

商品上下架

2、 订单服务取消订单问题

延时消息 过期订单关闭.

3、 分布式事务问题

支付------订单-------库存

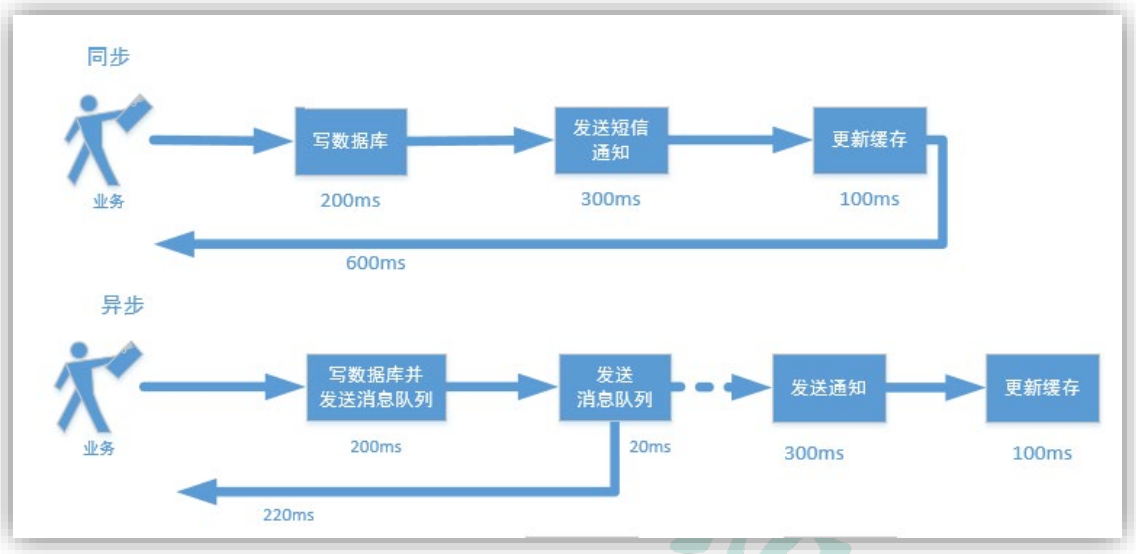
MySQL 修改了商品数据,搜索服务的 ES 数据要改变.

4、 秒杀的排队

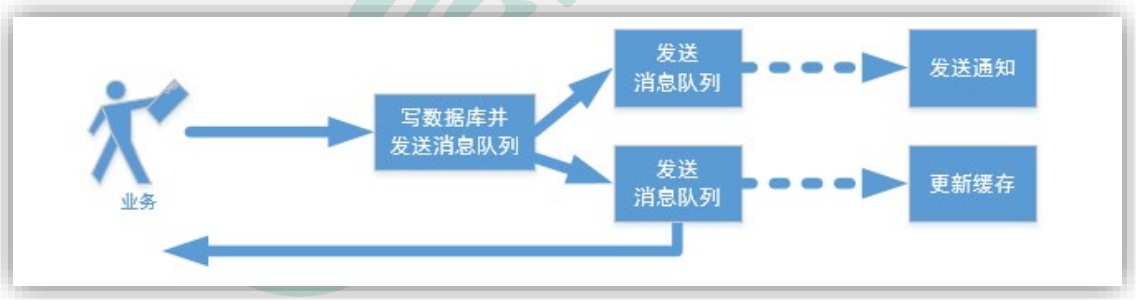
二、消息队列解决什么问题

消息队列都解决了什么问题?

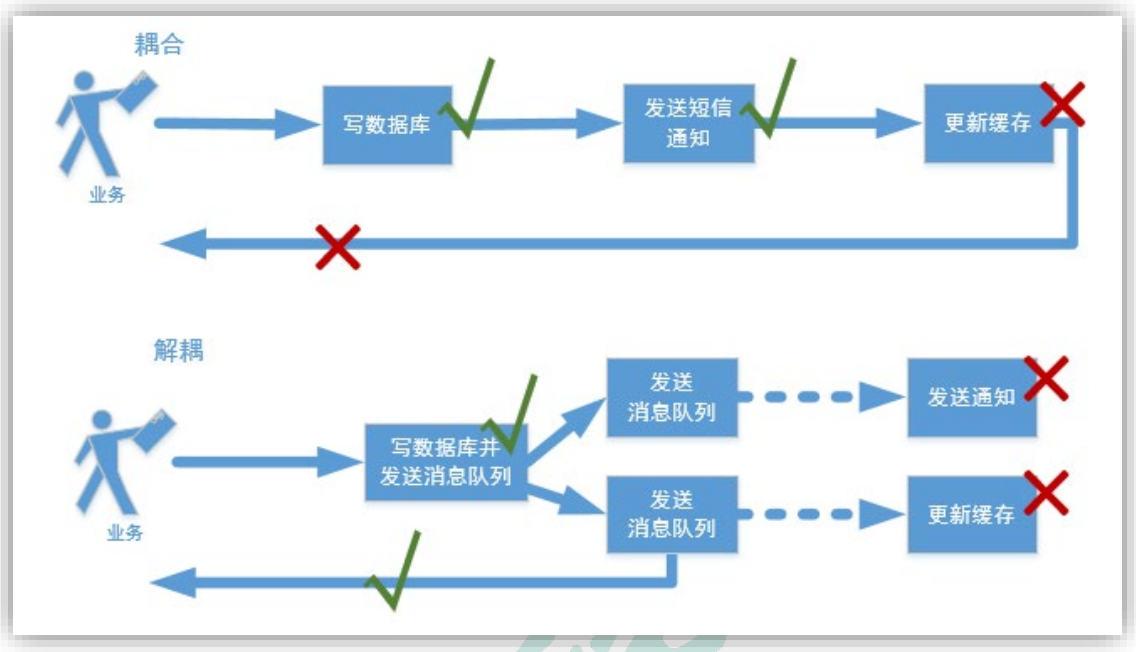
1、异步


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

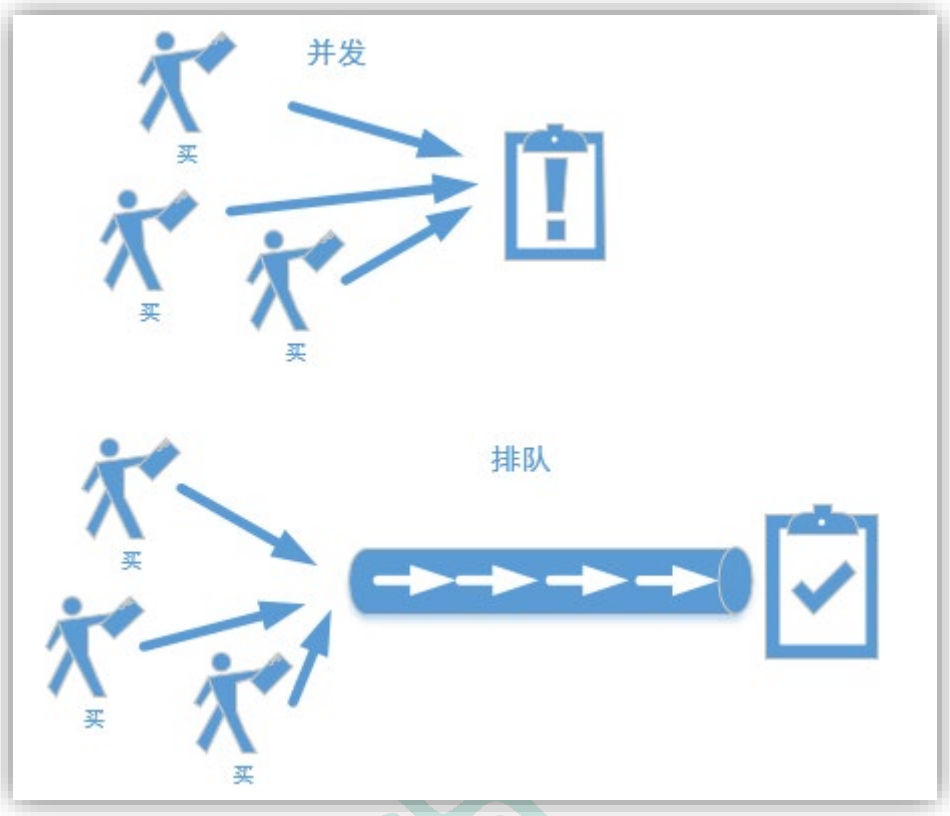
2、并行


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

3、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
消息队列
发送通知
更新缓存

4、排队



三、消息队列工具 RabbitMQ

1、常见 MQ 产品

- ActiveMQ：基于 JMS（java 协议）-----功能比较少

- 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----大数据量使用，容易消息丢失。

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

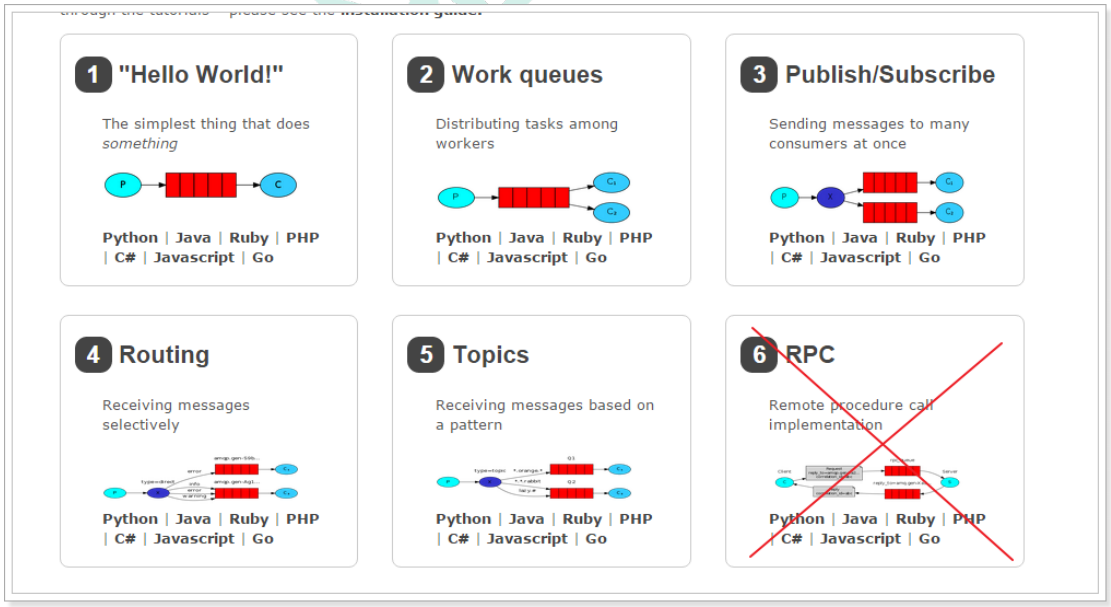
consumer：消息消费者，就是接受消息的程序

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、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

四、消息不丢失，准确性问题

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。

在投递消息时开启事务支持，如果消息投递失败，则回滚事务，但是，很少有人这么干，因为这是同步操作，一条消息发送之后会使发送端阻塞，以等待 RabbitMQ-Server 的回应，之后才能继续发送下一条消息，生产者生产消息的吞吐量和性能都会大大降低。

因此我们生产环境一般都采取消息确认模式。

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

说明：

@Queue: 当所有消费客户端连接断开后，是否自动删除队列

true：删除 false：不删除

@Exchange：当所有绑定队列都不在使用时，是否自动删除交换器

true：删除 false：不删除

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。

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

在配置文件中需要配置 mq 连接

```
rabbitmq:
    host: 192.168.200.128
    port: 5672
    username: guest
    password: guest
    publisher-confirm-type: correlated  交换机应答
    publisher-returns: true   队列应答
    listener:  监听 消费
      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
```

轮询分发：当有多个消费者的时候，不管消费方有没有消费完上一个消息，到你了 就给你。

公平分发：当有多个消费者的时候，一个消费者只有在消费完当前消息的情况下，才能去消费下一个消息。

五、商品搜索上下架

1、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);
}
```

2、service-list 消费消息

```
package com.atguigu.gmall.list.receiver;
@Component
public class ListReceiver {
    @Autowired
    private SearchService searchService;
    /**
     * 商品上架
    * @param skuId
     * @throws IOException
     */
    @RabbitListener(bindings = @QueueBinding(
            value = @Queue(value = MqConst.QUEUE_GOODS_UPPER,
durable = "true"),
            exchange = @Exchange(value =
MqConst.EXCHANGE_DIRECT_GOODS, type = ExchangeTypes.DIRECT, durable
= "true"),
            key = {MqConst.ROUTING_GOODS_UPPER}
    ))
    public void upperGoods(Long skuId, Message message, Channel
channel) throws IOException {
```

```
        if (null != skuId) {
            searchService.upperGoods(skuId);
        }

channel.basicAck(message.getMessageProperties().getDeliveryTag(),
false);
    }
    /**
     * 商品下架
     * @param skuId
     */
    @RabbitListener(bindings = @QueueBinding(
            value = @Queue(value = MqConst.QUEUE_GOODS_LOWER,
durable = "true"),
            exchange = @Exchange(value =
MqConst.EXCHANGE_DIRECT_GOODS, type = ExchangeTypes.DIRECT, durable
= "true"),
            key = {MqConst.ROUTING_GOODS_LOWER}
    ))
    public void lowerGoods(Long skuId, Message message, Channel
channel) throws IOException {
        if (null != skuId) {
            searchService.lowerGoods(skuId);
        }

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

六、延迟队列关闭过期订单

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

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

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。

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。

1.2、死信交换器 Dead Letter Exchanges

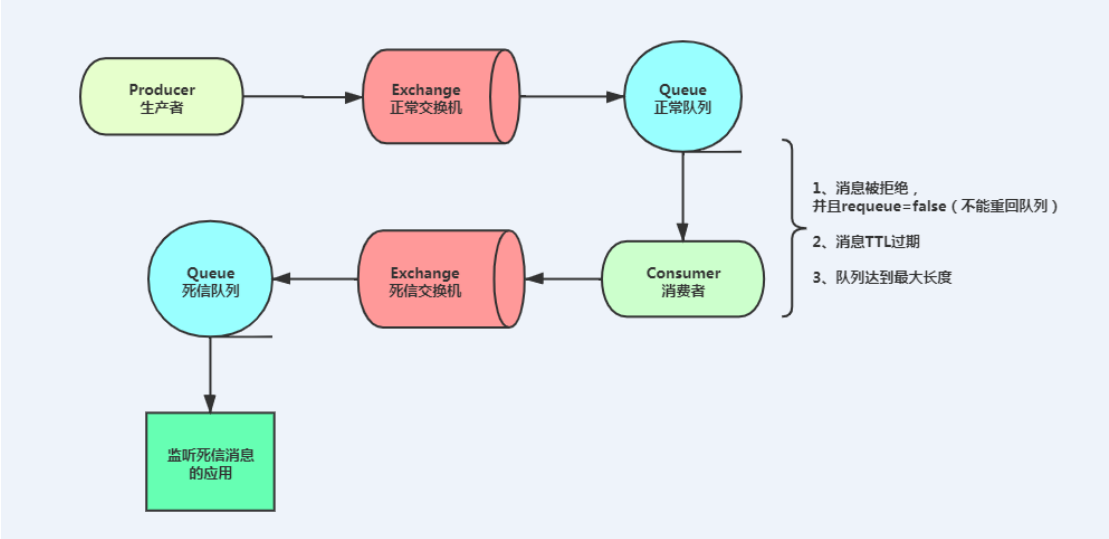
一个消息在满足如下条件下，会进死信路由，记住这里是路由而不是队列，一个路由可以对应很多队列。

（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。只是在某一个设置了 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。



2、基于延迟插件实现延迟消息

2.1、插件安装

1. 首先我们将刚下载下来的 rabbitmq_delayed_message_exchange-3.8.0.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-3.8.0.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

2.2、代码实现

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";

    /**
     * 队列不要在 RabbitListener 上面做绑定，否则不会成功，如队列 2，必须在此绑定
     *
     * @return
     */

    @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("sendDelay")
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
@Configuration
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);
    }

}
```

3、基于延迟插件实现取消订单

死信的延迟消息和延迟插件的延迟消息有什么区别？？

死信的要求 消息的存活时间一致。

延迟插件不要求。

rabbit-util 模块延迟接口封装

```
RabbitService

/**
 * 发送延迟消息
 * @param exchange 交换机
 * @param routingKey 路由键
 * @param message 消息
 * @param delayTime 单位：秒
 */
public boolean sendDelayMessage(String exchange, String
routingKey, Object message, int delayTime) {
    GmallCorrelationData correlationData = new
GmallCorrelationData();
    String correlationId = UUID.randomUUID().toString();
    correlationData.setId(correlationId);
```

```
    correlationData.setMessage(message);
    correlationData.setExchange(exchange);
    correlationData.setRoutingKey(routingKey);
    correlationData.setDelay(true);
    correlationData.setDelayTime(delayTime);

    redisTemplate.opsForValue().set(correlationId,
JSON.toJSONString(correlationData),          OBJECT_TIMEOUT,
TimeUnit.MINUTES);
    this.rabbitTemplate.convertAndSend(exchange,  routingKey,
message, new MessagePostProcessor() {
        @Override
        public  Message  postProcessMessage(Message  message)
throws AmqpException {

message.getMessageProperties().setDelay(delayTime*1000);
            return message;
        }
    },correlationData);
    return true;
}
```

3.1、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) {
    // orderInfo
    // 总金额，订单状态，用户 Id，第三方交易编号，创建时间，过期时间，进程
状态
  orderInfo.sumTotalAmount();
    orderInfo.setOrderStatus(OrderStatus.UNPAID.name());
    String outTradeNo = "ATGUIGU" + System.currentTimeMillis() + ""
+ new Random().nextInt(1000);
    orderInfo.setOutTradeNo(outTradeNo);
    orderInfo.setCreateTime(new Date());
```

```
    // 定义为1天
   Calendar calendar = Calendar.getInstance();
    calendar.add(Calendar.DATE, 1);
    orderInfo.setExpireTime(calendar.getTime());

    orderInfo.setProcessStatus(ProcessStatus.UNPAID.name());
    orderInfoMapper.insert(orderInfo);

    StringBuffer tradeBody = new StringBuffer();
    // 保存订单明细
  List<OrderDetail>                orderDetailList                =
orderInfo.getOrderDetailList();
    for (OrderDetail orderDetail : orderDetailList) {
        orderDetail.setId(null);
        orderDetail.setOrderId(orderInfo.getId());
        orderDetailMapper.insert(orderDetail);

        tradeBody.append(orderDetail.getSkuName()).append(" ");
    }

    //更新支付描述
   orderInfo.setTradeBody(tradeBody.toString());
    orderInfoMapper.updateById(orderInfo);
    //发送延迟队列，如果到过期时间了未支付，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();
}
```

3.2、接收消息

```
package com.atguigu.gmall.order.receiver;
@Component
public class OrderReceiver {
    @Autowired
    private OrderService orderService;
    /**
     * 取消订单消费者
     * 延迟队列，不能再这里做交换机与队列绑定

    * 需要在配置类里去绑定死信交换机
    * @param orderId
     * @throws IOException
     */
    @RabbitListener(queues = MqConst.QUEUE_ORDER_CANCEL)
```

```
    public void orderCancel(Long orderId, Message message, Channel
channel) throws IOException {
        if (null != orderId) {
            //防止重复消费
     OrderInfo orderInfo = orderService.getById(orderId);
            if (null != orderInfo &&
orderInfo.getOrderStatus().equals(ProcessStatus.UNPAID.getOrderStatu
s().name())) {
                orderService.execExpiredOrder(orderId);
            }
        }

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

取消订单业务，取消订单要关闭支付交易

```
@Override
public void execExpiredOrder(Long orderId) {
    // orderInfo
    updateOrderStatus(orderId, ProcessStatus.CLOSED);
    // paymentInfo
    //paymentFeignClient.closePayment(orderId);
    //发送取消交易的消息
   rabbitService.sendMessage(MqConst.EXCHANGE_DIRECT_PAYMENT_CLOSE,
MqConst.ROUTING_PAYMENT_CLOSE, orderId);
}
```

关闭交易消息消费者

```
package com.atguigu.gmall.payment.receiver;
@Component
public class PaymentReceiver {
    @Autowired
    private PaymentService paymentService;
    /**
     * 取消交易
    * @param orderId
     * @throws IOException
     */
    @RabbitListener(bindings = @QueueBinding(
            value = @Queue(value = MqConst.QUEUE_PAYMENT_CLOSE,
durable = "true"),
            exchange = @Exchange(value =
MqConst.EXCHANGE_DIRECT_PAYMENT_CLOSE),
            key = {MqConst.ROUTING_PAYMENT_CLOSE}
    ))
    public void closePayment(Long orderId) throws IOException {
        if (null != orderId) {
            paymentService.closePayment(orderId);
```

```
        }
    }
}
```

更改支付日志表，状态为关闭交易

```
@Override
public void closePayment(Long orderId) {
    QueryWrapper<PaymentInfo> queryWrapper = new QueryWrapper<>();
    queryWrapper.eq("order_id", orderId);
    PaymentInfo paymentInfoUp = new PaymentInfo();
    paymentInfoUp.setPaymentStatus(PaymentStatus.CLOSED.name());
    paymentInfoMapper.update(paymentInfoUp, queryWrapper);
    //关闭交易
    alipayService.closePay(orderId);

}
```

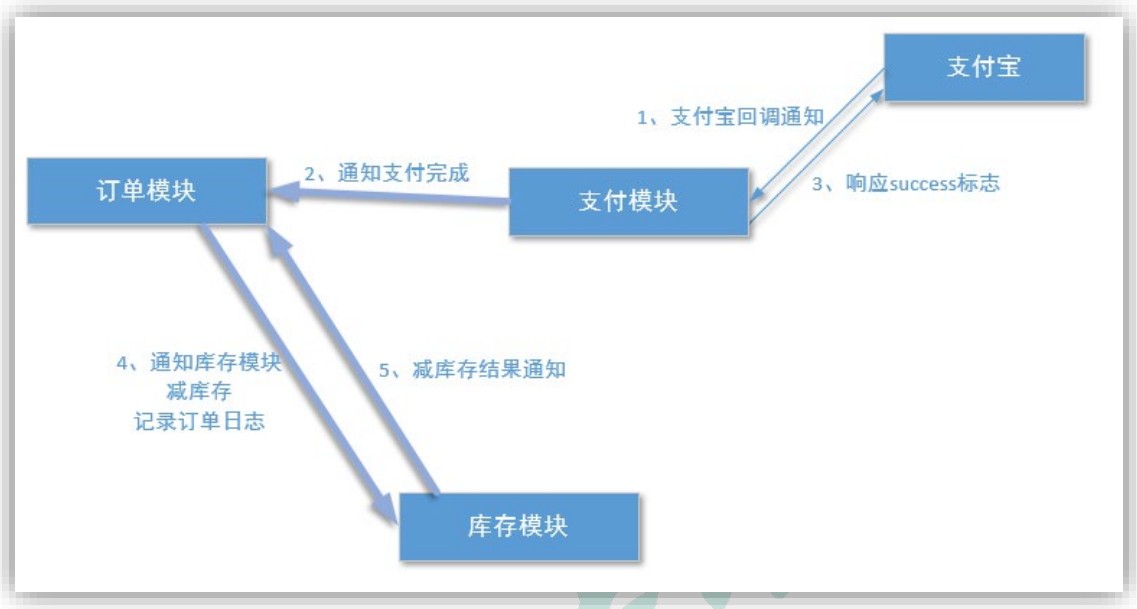
支付宝支付 AlipayServiceImpl 实现类

```
/***
 * 关闭交易
 * @param orderId
 * @return
 */
@Override
public Boolean closePay(Long orderId) {
    OrderInfo orderInfo = orderFeignClient.getOrderInfo(orderId);

    //AlipayClient alipayClient = new
DefaultAlipayClient("https://openapi.alipay.com/gateway.do","app_id"
,"your private_key","json","GBK","alipay_public_key","RSA2");
    AlipayTradeCloseRequest request = new AlipayTradeCloseRequest();
    HashMap<String, Object> map = new HashMap<>();
    map.put("trade_no", "");
    map.put("out_trade_no", orderInfo.getOutTradeNo());
    map.put("operator_id", "YX01");

    request.setBizContent(JSON.toJSONString(map));
    AlipayTradeCloseResponse response = null;
    try {
        response = alipayClient.execute(request);
    } catch (AlipayApiException e) {
        e.printStackTrace();
    }
    if(response.isSuccess()){
        log.info("调用成功");
        return true;
    }
    return false;
}
```

七、项目中分布式事务的业务场景



八、RabbitMQ 常见问题

1、使用 RabbitMQ 有什么好处？

1）.解耦：系统 A 在代码中直接调用系统 B 和系统 C 的代码，如果将来 D 系统接入，系统 A 还需要修改代码，过于麻烦！

2）.异步：将消息写入消息队列，非必要的业务逻辑以异步的方式运行，加快响应速度

3）.削峰：并发量大的时候，所有的请求直接怼到数据库，造成数据库连接异常

2、消息顺序问题

场景：比如支付操作，支付成功之后，会发送修改订单状态和扣减库存的消息，如果这两个消息同时发送，就不能保证完全按照顺序消费，有可能是先减库存了，后更改订单状态。

解决方案：同步执行，当一个消息执行完之后，再发布下一个消息。

3、如何保证 RabbitMQ 消息的可靠传输？

消息不可靠的原因是因为消息丢失

生产者丢失消息:

RabbitMQ 提供 transaction 事务和 confirm 模式来确保生产者不丢消息；

Transaction 事务机制就是说：发送消息前，开启事务（channel.txSelect()）,然后发送消息，如果发送过程中出现什么异常，事务就会回滚（channel.txRollback()）,如果发送成功则提交事务（channel.txCommit()），然而，这种方式有个缺点：吞吐量下降。

confirm 模式用的居多：一旦 channel 进入 confirm 模式，所有在该信道上发布的消息都将会被指派一个唯一的 ID（从 1 开始），一旦消息被投递到所有匹配的队列之后；

rabbitMQ 就会发送一个 ACK 给生产者（包含消息的唯一 ID），这就使得生产者知道消息已经正确到达目的队列了；

如果 rabbitMQ 没能处理该消息，则会发送一个 Nack 消息给你，可以进行重试操作。

消息列表丢失消息:

可以消息持久化, 即使 rabbitMQ 挂了，重启后也能恢复数据

消费者丢失消息:

消费者丢数据一般是因为采用了自动确认消息模式，消费者在收到消息之后，处理消息之前，会自动回复 RabbitMQ 已收到消息；如果这时处理消息失败，就会丢失该消息；改为手动确认消息即可！

4、消息重复消费问题

为什么会重复消费:

正常情况下，消费者在消费消息的时候，消费完毕后，会发送一个确认消息给消息队列，消息队列就知道该消息被消费了，就会将该消息从消息队列中删除；

但是因为网络传输等等故障，确认信息没有传送到消息队列，导致消息队列不知道已经消费过该消息了，再次将消息发送。

解决方案: 保证消息的唯一性，就算是多次传输，不要让消息的多次消费带来影响，保证消息消费的幂等性；

5、幂等性操作

幂等性就是一个数据或者一个请求，给你重复来了多次，你得确保对应的数据是不会改变的，不能出错。

要保证消息的幂等性，这个要结合业务的类型来进行处理。

1）、可在内存中维护一个 map 集合，只要从消息队列里面消费一个消息，先查询这个消息在不在 map 里面，如果在表示已消费过，直接丢弃；如果不在，则在消费后将其加入 map 当中。
2）、如果要写入数据库，可以拿唯一键先去数据库查询一下，如果不存在在写，如果存在直接更新或者丢弃消息。

3）、消息执行完会更改某个数据状态，判断数据状态是否更新，如果更新，则不进行重复消费。