消息队列如何保证不重复消费消息

问题背景

当使用消息队列时，客户端重复消费可能会成为一个严重的问题。

这是因为消息队列具有持久性和可靠性的特性，确保消息能够被成功传递给消费者。然而，这也会导致客户端在某些情况下重复消费消息，例如网络故障、客户端崩溃、消息处理失败等情况。

为了避免这种情况发生，需要在客户端实现一些机制来确保消息不会被重复消费，例如记录消费者已经处理的消息 ID、使用分布式锁来控制消费进程的唯一性等。这些机制能够保证消息被成功处理，同时也能够提高系统的可靠性和稳定性。

今天的文章我们将探讨如何确保消息队列中的消息不会被重复消费，下文将以 RocketMQ 为例说明。

消息幂等性

消息中间件是分布式系统中常用的组件，它具有广泛的应用价值，例如实现异步化、解耦、削峰等功能。通常情况下，我们认为消息中间件是一个可靠的组件。这里的可靠性指的是，只要消息被成功投递到了消息中间件，它就不会丢失，至少能够被消费者成功消费一次。这是消息中间件最基本的特性之一，也就是我们通常所说的 “AT LEAST ONCE”，即消息至少会被成功消费一遍。

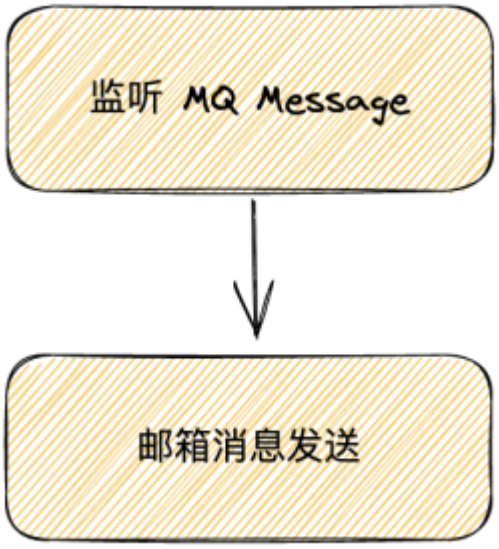
举个例子，假设一个消息M被发送到消息中间件并被消费程序A接收到，A开始消费这个消息，但是在消费过程中程序重启了。由于这个消息没有被标记为已经被消费成功，消息中间件会持续地将这个消息投递给消费者，直到消息被成功消费为止。

然而，这种可靠性特性也会导致消息被多次投递的情况。举个例子，仍然以之前的例子为例，如果消费程序A接收并完成消息M的消费逻辑后，正准备通知消息中间件“我已经消费成功了”，但在此之前程序A又重启了，那么对于消息中间件来说，这个消息M并没有被成功消费过，因此消息中间件会继续投递这个消息。而对于消费程序A来说，尽管它已经成功消费了这个消息，但由于程序重启导致消息中间件继续投递，看起来就好像这个消息还没有被成功消费过一样。

在 RockectMQ 的场景中，这意味着同一个 messageId 的消息会被重复投递。由于消息的可靠投递是更重要的，所以避免消息重复投递的任务转移给了应用程序自身来实现。这也是 RocketMQ 文档强调消费逻辑需要自行实现幂等性的原因。实际上，这背后的逻辑是：在分布式场景下，保证消息不丢和避免消息重复投递是矛盾的，但是消息重复投递是可以解决的，而消息丢失则非常麻烦。

幂等设计

让我们先来了解一下邮件消息的发送流程，以便更好了解消息队列幂等工作原理。



正如我们在之前提到的，RocketMQ 遵循 "AT LEAST ONCE" 语义，这意味着消息可能会被重复消费。在发送邮件消息的情况下，由于消息可能被重复消费，我们需要保证幂等性，以确保邮件不会被重复发送。

1. 消息发送逻辑

下面这块代码是 12306 支付结果回调订单逻辑实现，通过 `RocketMQMessageListener` 监听并消费 RocketMQ 消息。

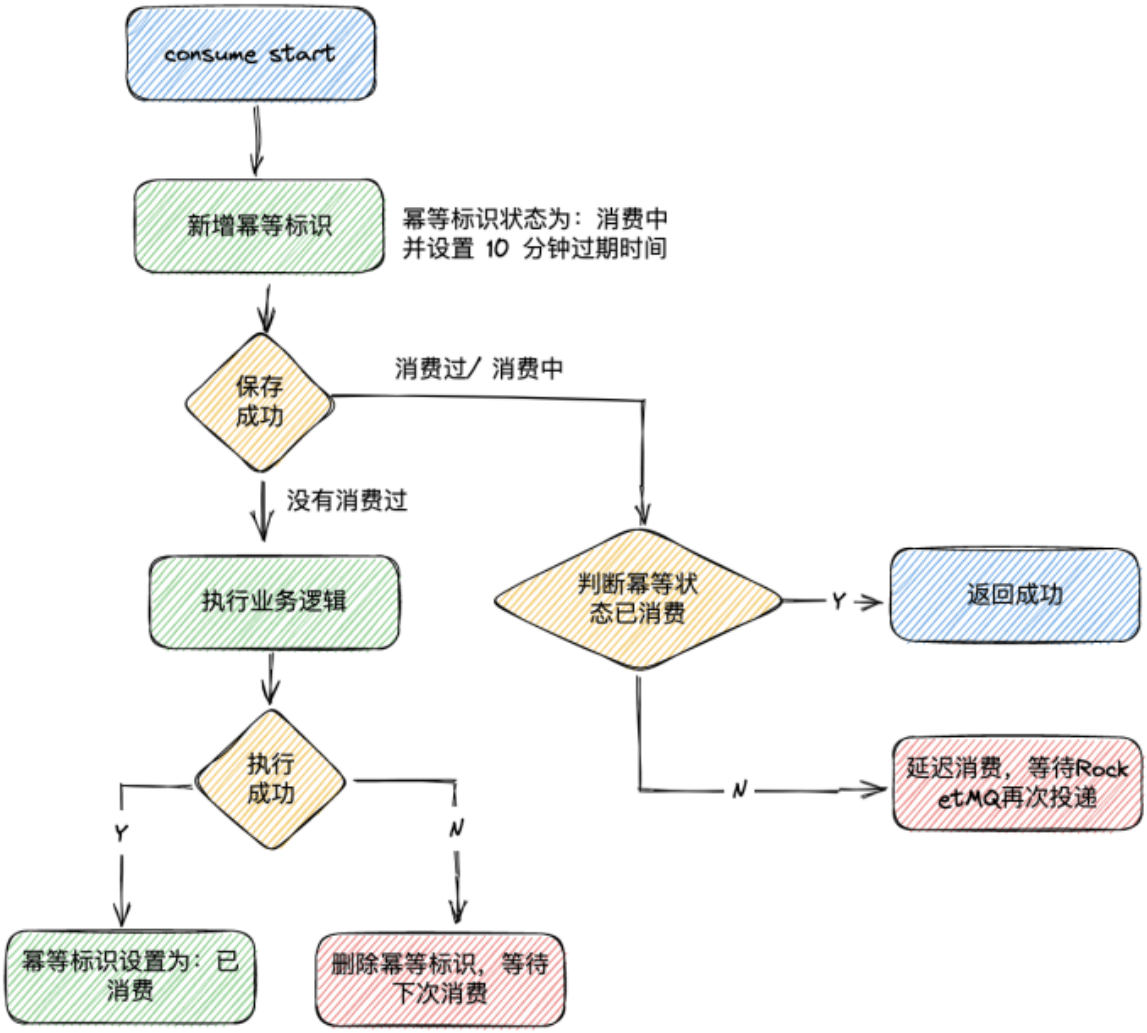
Java | 复制代码

```java
@Slf4j
@Component
@RequiredArgsConstructor
@RocketMQMessageListener(
        topic = OrderRocketMQConstant.PAY_GLOBAL_TOPIC_KEY,
        selectorExpression = OrderRocketMQConstant.PAY_RESULT_CALLBACK_ORDE
        consumerGroup = OrderRocketMQConstant.PAY_RESULT_CALLBACK_ORDER_CG_
)
public class PayResultCallbackOrderConsumer implements RocketMQListener<Mes

    private final OrderService orderService;


    @Transactional(rollbackFor = Exception.class)
    @Override
    public void onMessage(MessageWrapper<PayResultCallbackOrderEvent> messa
        PayResultCallbackOrderEvent payResultCallbackOrderEvent = message.g
        OrderStatusReversalDTO orderStatusReversalDTO = OrderStatusReversal
                .orderSn(payResultCallbackOrderEvent.getOrderSn())
                .orderStatus(OrderStatusEnum.ALREADY_PAID.getStatus())
                .build();
        orderService.statusReversal(orderStatusReversalDTO);
        orderService.payCallbackOrder(payResultCallbackOrderEvent);
    }
}
```

2. 幂等处理逻辑

下述方案的优点在于，使用 Redis 消息去重表，不依赖事务，针对消息表本身做了状态的区分：消费中、消费完成。

如果消息已经在消费中，抛出异常，消息会触发延迟消费，在 RocketMQ 的场景下即发送到 RETRY TOPIC。



通过该方案可以解决什么问题?

1．消息已经消费成功了，第二条消息将被直接幂等处理掉（消费成功）。

2．并发场景下的消息，依旧能满足不会出现消息重复，即穿透幂等挡板的问题。

3．支持上游业务生产者重发的业务重复的消息幂等问题。

为什么要给初始化的幂等标识新增 10 分钟过期时间?

在并发场景下，我们使用消息状态来实现并发控制，以使第二条消息被不断延迟消费（即重试）。但如果在此期间第一条消息也因某些异常原因（例如机器重启或外部异常）未成功消费，该怎么办呢？因为每次查询时都会显示消费中的状态，所以延迟消费会一直进行下去，直到最终被视为消费失败并被投递到死信 Topic 中（RocketMQ 默认最多可以重复消费 16 次）。

针对这个问题，我们采取了一种解决方案：在插入消息表时，必须为每条消息设置一个最长消费过期时间，例如 10 分钟。这意味着，如果某个消息在消费过程中超过了 10 分钟，就会被视为

消费失败并从消息表中删除。

抽象幂等通用组件

为了解决消息队列中的重复消费问题，我们可以设计一套通用的消息队列幂等组件。这个组件可以被各个应用程序使用，以确保它们的消费逻辑是幂等的。这种通用的幂等组件可以使应用程序不必为了解决重复消费问题而浪费精力和时间，从而更专注于业务逻辑的实现。

在企业项目中，使用 MySQL 作为幂等去重表的情况比较少见，因此在代码中只提供了 Redis 实现方案。

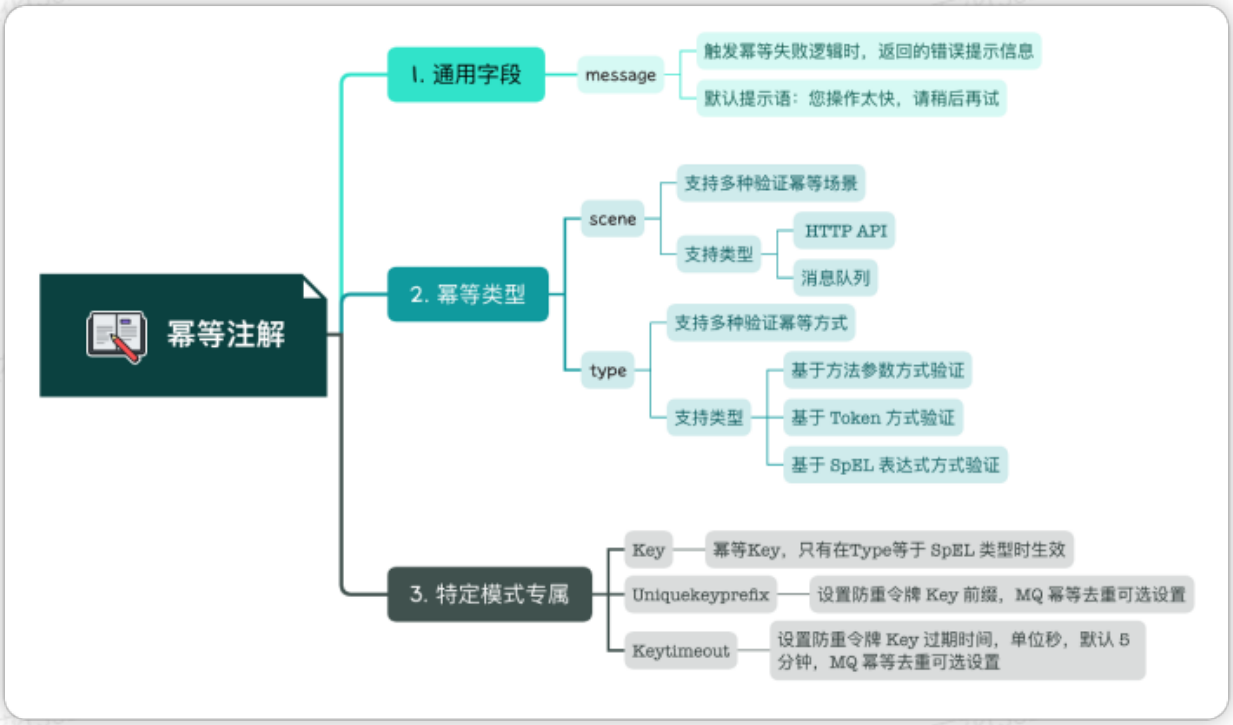
1. 定义幂等注解

我们提供了一种通用的幂等注解，该注解可用于 RestAPI 和消息队列消息防重复场景。

Java 复制代码

```java
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface Idempotent {

    /**
     * 幂等Key，只有在 {@link Idempotent#type()} 为 {@link IdempotentTypeEnu
     */
    String key() default "";

    /**
     * 触发幂等失败逻辑时，返回的错误提示信息
     */
    String message() default "您操作太快，请稍后再试";

    /**
     * 验证幂等类型，支持多种幂等方式
     * RestAPI 建议使用 {@link IdempotentTypeEnum#TOKEN} 或 {@link Idempoten
     * 其它类型幂等验证，使用 {@link IdempotentTypeEnum#SPEL}
     */
    IdempotentTypeEnum type() default IdempotentTypeEnum.PARAM;

    /**
     * 验证幂等场景，支持多种 {@link IdempotentSceneEnum}
     */
    IdempotentSceneEnum scene() default IdempotentSceneEnum.RESTAPI;

    /**
     * 设置防重令牌 Key 前缀，MQ 幂等去重可选设置
     * {@link IdempotentSceneEnum#MQ} and {@link IdempotentTypeEnum#SPEL} 时
     */
    String uniqueKeyPrefix() default "";

    /**
     * 设置防重令牌 Key 过期时间，单位秒，默认 1 小时，MQ 幂等去重可选设置
     * {@link IdempotentSceneEnum#MQ} and {@link IdempotentTypeEnum#SPEL} 时
     */
    long keyTimeout() default 3600L;
}
```

为了方便理解，整理成思维导图方便记忆。



2. 定义 AOP 增强

我们使用 AOP 技术为方法增强提供了通用的幂等性保证，只需要在需要保证幂等性的方法上添加 `@Idempotent` 注解， `Aspect` 就会对该方法进行增强。

这种技术不仅适用于 RestAPI 场景，还适用于消息队列的防重复消费场景。

Java 复制代码

```java
package org.opengoofy.index12306.framework.starter.idempotent.core;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.reflect.MethodSignature;
import org.opengoofy.index12306.framework.starter.idempotent.annotation.I

import java.lang.reflect.Method;

/**
 * 幂等注解 AOP 拦截器
 *
 * @公众号：马丁玩编程，回复：加群，添加马哥微信（备注：12306）获取项目资料
 */
@Aspect
public final class IdempotentAspect {

    /**
     * 增强方法标记 {@link Idempotent} 注解逻辑
     */
    @Around("@annotation(org.opengoofy.index12306.framework.starter.idemp
    public Object idempotentHandler(ProceedingJoinPoint joinPoint) throws
        Idempotent idempotent = getIdempotent(joinPoint);
        IdempotentExecuteHandler instance = IdempotentExecuteHandlerFacto
        Object resultObj;
        try {
            instance.execute(joinPoint, idempotent);
            resultObj = joinPoint.proceed();
            instance.postProcessing();
        } catch (RepeatConsumptionException ex) {
            /**
             * 触发幂等逻辑时可能有两种情况：
             *    * 1. 消息还在处理，但是不确定是否执行成功，那么需要返回错误，
             *    * 2. 消息处理成功了，该消息直接返回成功即可
             */
            if (!ex.getError()) {
                return null;
            }
            throw ex;
        } catch (Throwable ex) {
            // 客户端消费存在异常，需要删除幂等标识方便下次 RocketMQ 再次通过
            instance.exceptionProcessing();
            throw ex;
        } finally {
            IdempotentContext.clean();
        }
        return resultObj;
    }
```

```
50
51      public static Idempotent getIdempotent(ProceedingJoinPoint joinPoint)
52          MethodSignature methodSignature = (MethodSignature) joinPoint.get
53          Method targetMethod = joinPoint.getTarget().getClass().getDeclare
54          return targetMethod.getAnnotation(Idempotent.class);
55      }
56  }
```

这个方法的执行逻辑与设计部分相同，因此在此处不再贴出具体的代码。大家可以跟着设计阅读幂等源码。

为了提高通用性和抽象性，该组件采用了模板方法和简单工厂等设计模式，这有助于隔离复杂性和提高可扩展性。如果您在学习过程中遇到问题，欢迎在知识星球 APP 上向我提问。

3. 实际场景使用

以实现支付结果回调订单为例，我们可以将通用组件引入到消息消费的逻辑中，具体流程如下：

支持通过 SpEL 表达式来充当幂等去重表唯一键，通过一个简单的注解，完美解决消息队列重复消费问题。

更复杂的幂等场景

到这里，方案看起来非常完美，所有的消息都可以快速接入去重，而且与具体业务实现完全解耦。但是，是否这样就可以完美地完成去重的所有任务呢？

很遗憾，实际上并非如此。因为需要确保消息至少成功消费一次，因此消息在消费过程中有可能失败并触发重试。

还是以上面的例子，假设消息消费的流程包含：

1. 检查库存（RPC）
2. 锁库存（RPC）
3. 开启事务，插入订单表（MySQL）
4. 调用某些其他下游服务（RPC）
5. 更新订单状态
6. commit 事务（MySQL）

当消息消费到第三步的时候假设 MySQL 异常导致失败了，触发消息重试。在重试前我们会删除幂等表的记录，所以消息重试的时候就会重新进入消费代码，那么步骤 1 和步骤 2 就会重新再执行一遍。

如果步骤 2 本身不是幂等的，那么这个业务消息消费依旧没有做好完整的幂等处理。

1. 通用方法实现价值

尽管这种方式并不能完全解决消息幂等问题（事实上，软件工程领域里很少有银弹可以完全解决问题），但它仍然具有很大的价值。通过这种简便的方式，我们能够解决以下问题：

1. 各种由于Broker、负载均衡等原因导致的消息重投递的重复问题。
2. 各种上游生产者导致的业务级别消息重复问题。
3. 重复消息并发消费的控制窗口问题，就算重复，重复也不可能同一时间进入消费逻辑。

2. 消息去重的建议

使用这种方法可以确保在正常的消费逻辑场景下（无异常，无异常退出），消息的幂等性全部得到解决，无论是业务重复还是 RocketMQ 特性带来的重复。虽然它不是解决消息幂等性的银弹，但它以一种简单和便捷的方式提供了解决方案。

实际上，这种方法已经可以解决 99% 的消息重复问题了，因为异常情况通常是少数情况。但是，如果希望在异常情况下也能处理好幂等问题，可以采取以下措施来降低问题发生的概率：

1. 消息消费失败时，应该及时回滚处理。如果消息消费失败本身具备回滚机制，则消息重试也就没有副作用了。
2. 为了尽可能避免程序异常退出导致的消息重试，需要在消费者代码中做好优雅退出处理。
3. 针对一些无法做到完全幂等的操作，至少要做到终止消息的消费并进行告警。比如锁定库存的操作，如果通过业务流水号已经成功锁定了库存，再次触发锁库存操作的话，如果无法做到幂等性处理，那么至少要在消息消费过程中触发异常（如因主键冲突导致消费异常等），并终止消息的消费，以避免重复消费产生的副作用。
4. 在 #3 做好的前提下，做好消息的消费监控，发现消息重试不断失败的时候，手动做好 #1 的回滚，使得下次重试消费成功。

文末总结

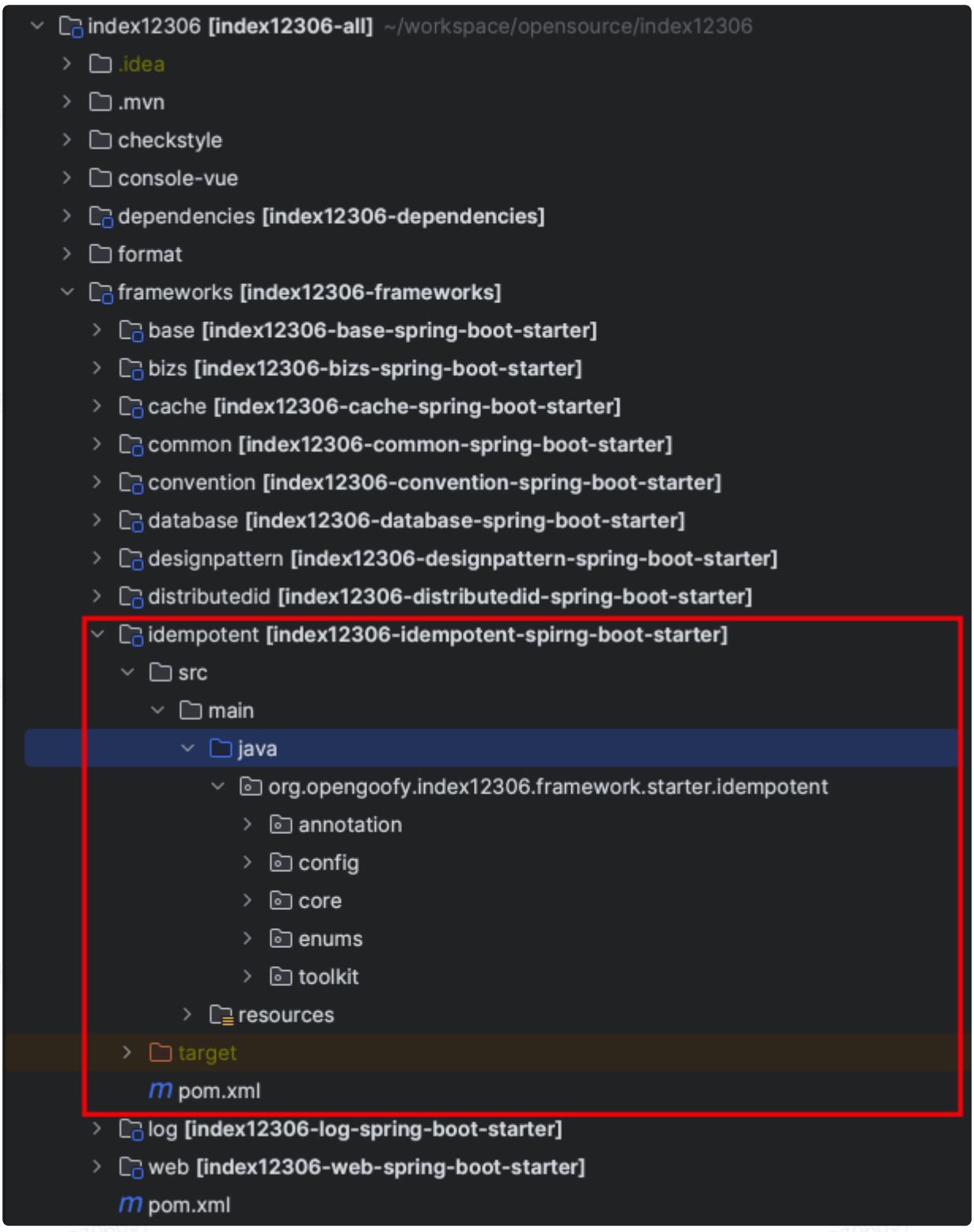
当我们在使用 RocketMQ 进行消息处理时，消息的幂等性是一个非常重要的问题。本文通过抽象出通用组件的方式，实现了 RestAPI 和 RocketMQ 的幂等处理。

同时，我们也发现，幂等性并不是一个银弹，不同的业务场景需要不同的幂等处理策略。

但是，通过一些基本的处理策略，如优雅退出、回滚处理、消费监控等，我们能够大大减少消息重复的问题，提高消息消费的稳定性和可靠性。

在实际开发中，需要结合具体业务场景，选择合适的幂等处理策略，并且在每次新的场景出现时，都需要仔细考虑是否需要重新审视幂等性的处理方式。

上文中的代码以及实现已在基础架构模块中定义，详情查看。



f1c25c10af55.png&title=%E6%B6%88%E6%81%AF%E9%98%9F%E5%88%97%E5%A6%82%E4%