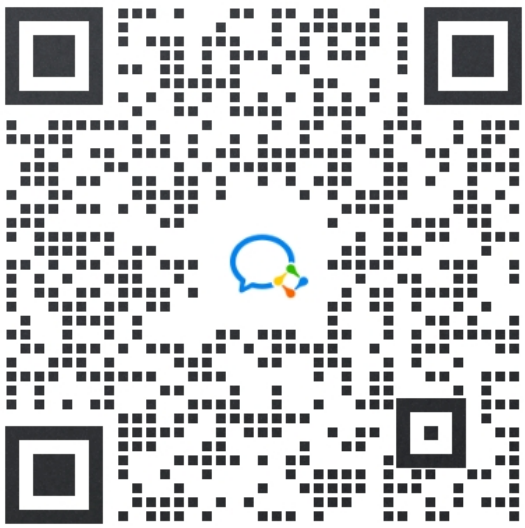
5. RabbitMQ应用问题

版权说明



1. 幂等性保障

1.1 幂等性介绍

幂等性是数学和计算机科学中某些运算的性质, 它们可以被多次应用, 而不会改变初始应用的结果.

应用程序的幂等性介绍

在应用程序中, 幂等性就是指对一个系统进行重复调用(相同参数), 不论请求多少次, 这些请求对系统的影响都是相同的效果.

比如数据库的 `select` 操作. 不同时间两次查询的结果可能不同, 但是这个操作是符合幂等性的. 幂等性指的是对资源的影响, 而不是返回结果. 查询操作对数据资源本身不会产生影响, 之所以结果不同, 可能是因为两次查询之间有其他操作对资源进行了修改.

比如 `i++` 这个操作, 就是非幂等性的. 如果调用方没有控制好逻辑, 一次流程重复调用好几次, 结果就会不同.

MQ的幂等性介绍

对于MQ而言, 幂等性是指同一条消息, 多次消费, 对系统的影响是相同的.

一般消息中间件的消息传输保障分为三个层级.

1. At most once:最多一次. 消息可能会丢失，但绝不会重复传输.
2. At least once:最少一次. 消息绝不会丢失，但可能会重复传输.
3. Exactly once:恰好一次. 每条消息肯定会被传输一次且仅传输一次.

RabbitMQ支持"最多一次"和"最少一次".

对于"恰好一次", 目前RabbitMQ还做不到, 不仅是RabbitMQ, 目前市面上主流的消息中间件, 都做不到这一点.

在业务使用中, 对于可靠性要求比较高的场景, 建议使用"最少一次", 以防止消息丢失. "最多一次" 会因为消息发送过程中, 网络问题, 消费出现异常等种种原因, 导致消息丢失.

> 以下场景可能会导致消息发送重复(包含但不限于)
>
> - 发送时消息重复: 当一条消息已被成功发送到服务端并完成持久化, 此时出现了网络闪断或者客户端宕机, 导致服务端对客户端应答失败. 如果此时Producer意识到消息发送失败并尝试再次发送消息, Consumer后续会收到两条内容相同并且Message ID也相同的消息.
> - 投递时消息重复: 消息消费的场景下, 消息已投递到Consumer并完成业务处理, 当客户端给服务端反馈应答的时候网络闪断. 为了保证消息至少被消费一次, 云消息队列 RabbitMQ 版的服务端将在网络恢复后再次尝试投递之前已被处理过的消息, Consumer后续会收到两条内容相同并且Message ID也相同的消息.
>
> 
>

但是"最少一次", 就会造成一个问题, 消费端会收到重复的消息, 也会造成对同一条消息进行多次处理. 一些不重要的业务还好一点, 对于重要的业务, 如果不对重复的消息进行处理, 会造成严重事故.

比如: 当用户对一个订单付款之后, 因为网络问题, 付款成功的结果未返回给订单系统, 当用户再次点击付款时, 如果系统未做幂等性处理, 那就会造成两次扣款

1.2 解决方案

MQ消费者的幂等性的解决方法, 一般有以下几种:

全局唯一ID

1. 为每条消息分配一个唯一标识符, 比如UUID或者MQ消息中的唯一ID,但是一定要保证唯一性.
2. 消费者收到消息后, 先用该id判断该消息是否已经消费过, 如果已经消费过, 则放弃处理.
3. 如果未消费过, 消费者开始消费消息, 业务处理成功后, 把唯一ID保存起来(数据库或Redis等)

> 可以使用Redis 的原子性操作setnx来保证幂等性, 将唯一ID作为key放到redis中`(SETNX messageID 1)`. 返回1, 说明之前没有消费过, 正常消费. 返回0, 说明这条消息之前已消费过, 抛弃.

业务逻辑判断

在业务逻辑层面实现消息处理的幂等性.

例如: 通过检查数据库中是否已存在相关数据记录, 或者使用乐观锁机制来避免更新已被其他事务更改的数据, 再或者在处理消息之前, 先检查相关业务的状态, 确保消息对应的操作尚未执行, 然后才进行处理, 具体根据业务场景来处理

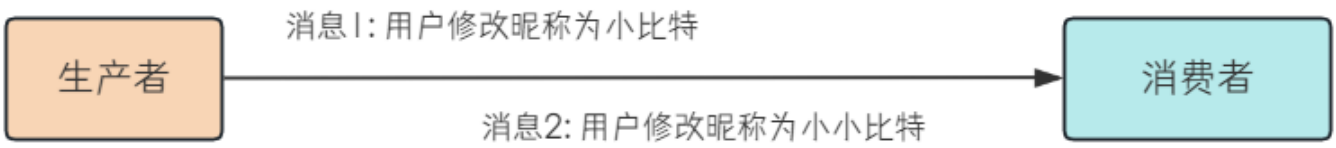
2. 顺序性保障

2.1 顺序性保障介绍

消息的顺序性是指消费者消费的消息和生产者发送消息的顺序是一致的.

比如生产者发送的消息分别是msg1, msg2, msg3, 那么消费者也是按照msg1, msg2, msg3的顺序进行消费的.

很多业务场景下, 消息的消费是不用保证顺序的, 比如使用MQ实现订单超时的处理. 但有些业务场景, 可能存在多个消息顺序处理的情况. 比如用户信息修改, 对同一个用户的同一个资料进行修改, 需要保证消息的顺序.



一些资料显示RabbitMQ的消息能够保障顺序性, 这是不严谨的. 在不考虑消息丢失, 网络故障等异常的情况下, 如果只有一个消费者, 最好也只有一个生产者的情况下, 是可以保证消息的顺序性. 如果有多个生产者同时发送消息, 无法确定消息到达RabbitMQ Broker的前后顺序, 也就无法验证消息的顺序性.

哪些情况可能会打破RabbitMQ的顺序性呢? 下面介绍几种常见的场景:

1. 多个消费者: 当队列配置了多个消费者时, 消息可能会被不同的消费者并行处理, 从而导致消息处理的顺序性无法保证.
2. 网络波动或异常: 在消息传递过程中, 如果出现网络波动或异常, 可能会导致消息确认(ACK) 丢失, 从而使得消息被重新入队和重新消费, 造成顺序性问题.
3. 消息重试：如果消费者在处理消息后未能及时发送确认, 或者确认消息在传输过程中丢失, 那么MQ可能会认为消息未被成功消费而进行重试, 这也可能导致消息处理的顺序性问题.
4. 消息路由问题: 在复杂的路由场景中, 消息可能会根据路由键被发送到不同的队列, 从而无法保证全局的顺序性.
5. 死信队列: 消息因为某些原因(如消费端拒绝消息)被放入死信队列, 死信队列被消费时, 无法保证消息的顺序和生产者发送消息的顺序一致

包括但不仅限于以上几种情形会使RabbitMQ消息错序, 如果要保证消息的顺序性, 需要业务方使用RabbitMQ之后做进一步的处理

2.2 顺序性保障方案

消息顺序性保障分为: 局部顺序性保证和全局顺序性保证.

局部顺序性通常指的是在单个队列内部保证消息的顺序. 全局顺序性是指在多个队列或多个消费者之间保证消息的顺序.

在实际应用中, 全局顺序性很难实现, 可以考虑使用业务逻辑来保证顺序性, 比如在消息中嵌入序列号, 并在消费端进行排序处理. 相对而言，局部顺序性更常见, 也更容易实现.

RabbitMQ作为一个分布式消息队列, 主要优化的是吞吐量和可用性, 而不是严格的顺序性保证. 如果业务场景确实需要严格的消息顺序, 可能需要在应用层面进行额外的设计和实现.

接下来说一下消息的顺序性保证的常见策略.

1. **单队列单消费者**

最简单的方法是使用单个队列, 并由单个消费者进行处理. 同一个队列中的消息是先进先出的, 这是RabbitMQ来帮助我们保证的.

2. **分区消费**

单个消费者的吞吐太低了, 当需要多个消费者以提高处理速度时, 可以使用分区消费. 把一个队列分割成多个分区, 每个分区由一个消费者处理, 以此来保持每个分区内消息的顺序性.

> 比如用户修改资料后, 发送一条用户资料消息. 消费者在处理时, 需要保证消息发送的先后顺序

> 但这种场合并不需要保证全局顺序. 只需要保证同一个用户的消息顺序消费就可以. 这时候就可以采用把消费按照一定的规则, 分为多个区, 每个分区由一个消费者处理
>
> RabbitMQ本身并不支持分区消费, 需要业务逻辑去实现, 或者借助spring-cloud-stream来实现
>
> 参考: https://docs.spring.io/spring-cloud-stream/reference/rabbit/rabbit_partitions.html

3. **消息确认机制**

使用手动消息确认机制, 消费者在处理完一条消息后, 显式地发送确认, 这样RabbitMQ才会移除并继续发送下一条消息.

4. **业务逻辑控制**

在某些情况下, 即使消息乱序到达, 也可以在业务逻辑层面实现顺序控制. 比如通过在消息中嵌入序列号, 并在消费时根据这些信息来处理

RabbitMQ本身并不保证全局的严格顺序性, 特别是在分布式系统中. 在实际应用开发中, 根据具体的业务需求, 可能需要结合多种策略来实现所需要的顺序保证.

3. 消息积压问题

3.1 原因分析

消息积压是指在消息队列(如RabbitMQ)中, 待处理的消息数量超过了消费者处理能力, 导致消息在队列中不断堆积的现象.

通常有以下几种原因:

1. **消息生产过快**: 在高流量或者高负载的情况下, 生产者以极高的速率发送消息, 超过了消费者的处理能力.
2. **消费者处理能力不足**: 消费者处理处理消息的速度跟不上消息生产的速度, 也会导致消息在队列中积压.

   可能原因有:

   1) 消费端业务逻辑复杂, 耗时长

   2) 消费端代码性能低

   3) 系统资源限制, 如CPU、内存、磁盘I/O等也会限制消费者处理消息的效率.

   4) 异常处理处理不当. 消费者在处理消息时出现异常, 导致消息无法被正确处理和确认.
3. **网络问题**: 因为网络延迟或不稳定, 消费者无法及时接收或确认消息, 最终导致消息积压
4. **RabbitMQ 服务器配置偏低**

消息积压可能会导致系统性能下降, 影响用户体验, 甚至导致系统崩溃. 因此, 及时发现消息积压并解决对于维护系统稳定性至关重要.

3.2 解决方案

遇到消息积压时, 首先要分析消息积压造成的原因. 根据原因来调整策略.

主要从以下几个方面来解决:

1. **提高消费者效率**
    a. 增加消费者实例数量, 比如新增机器
    b. 优化业务逻辑, 比如使用多线程来处理业务
    c. 设置prefetchCount, 当一个消费者阻塞时, 消息转发到其他未阻塞的消费者.
    d. 消息发生异常时, 设置合适的重试策略, 或者转入到死信队列
2. **限制生产者速率**. 比如流量控制, 限流算法等.
    a. 流量控制: 在消息生产者中实现流量控制逻辑, 根据消费者处理能力动态调整发送速率
    b. 限流: 使用限流工具, 为消息发送速率设置一个上限
    c. 设置过期时间. 如果消息过期未消费, 可以配置死信队列, 以避免消息丢失, 并减少对主队列的压力
3. **资源与配置优化**. 比如升级RabbitMQ服务器的硬件, 调整RabbitMQ的配置参数等

上述这些策略选择时, 需要综合考虑业务需求和系统的实际承载能力.