RocketMQ 重试机制详解及最佳实践

引言

本文主要介绍在使用 RocketMQ 时为什么需要重试与兜底机制，生产者与消费者触发重试的条件和具体行为，如何在 RocketMQ 中合理使用重试机制，帮助构建弹性，高可用系统的最佳实践。

RocketMQ 的重试机制包括三部分，分别是生产者重试，服务端内部数据复制遇到非预期问题时重试，消费者消费重试。本文中仅讨论生产者重试和消费者消费重试两种面向用户侧的实现。

生产者发送重试

RocketMQ 的生产者在发送消息到服务端时，可能会因为网络问题，服务异常等原因导致调用失败，这时候应该怎么办？如何尽可能的保证消息不丢失呢？

1. 生产者重试次数

RocketMQ 在客户端中内置了请求重试逻辑，支持在初始化时配置消息发送最大重试次数（默认为 2 次），失败时会按照设置的重试次数重新发送。直到消息发送成功，或者达到最大重试次数时结束，并在最后一次失败后返回调用错误的响应。对于同步发送和异步发送，均支持消息发送重试。

- 同步发送：调用线程会一直阻塞，直到某次重试成功或最终重试失败（返回错误码或抛出异常）。
- 异步发送：调用线程不会阻塞，但调用结果会通过回调的形式，以异常事件或者成功事件返回。

2. 生产者重试间隔

在介绍生产者重试前，我们先来了解下流控的概念，流控一般是指服务端压力过大，容量不足时服务端会限制客户端收发消息的行为，是服务端自我保护的一种设计。RocketMQ 会根据当前是否触发了流控而采用不同的重试策略：

非流控错误场景：其他触发条件触发重试后，均会立即进行重试，无等待间隔。

流控错误场景：系统会按照预设的指数退避策略进行延迟重试。

- **为什么要引入退避和随机抖动？ **

如果故障是由过载流控引起的，重试会增加服务端负载，导致情况进一步恶化，因此客户端在遇到流控时会在两次尝试之间等待一段时间。每次尝试后的等待时间都呈指数级延长。指数回退可能导致很长的回退时间，因为指数函数增长很快。指数退避算法通过以下参数控制重试行为，更多信息，请参见 connection-backoff.md。

INITIAL_BACKOFF：第一次失败重试前后需等待多久，默认值：1 秒； MULTIPLIER：指数退避因子，即退避倍率，默认值：1.6； JITTER：随机抖动因子，默认值：0.2； MAX_BACKOFF：等待间隔时间上限，默认值：120 秒； MIN_CONNECT_TIMEOUT：最短重试间隔，默认值：20 秒。

```
ConnectWithBackoff()
  current_backoff = INITIAL_BACKOFF
  current_deadline = now() + INITIAL_BACKOFF
  while (TryConnect(Max(current_deadline, now() + MIN_CONNECT_TIMEOUT))!= SUCCESS)
    SleepUntil(current_deadline)
    current_backoff = Min(current_backoff * MULTIPLIER, MAX_BACKOFF)
    current_deadline = now() + current_backoff + UniformRandom(-JITTER * current_backoff, JITTER * current_backoff)
```

特别说明：对于事务消息，只会进行透明重试（transparent retries），网络超时或异常等场景不会进行重试。

3. 重试带来的副作用

不停的重试看起来很美好，但也是有副作用的，主要包括两方面：消息重复，服务端压力增大

- 远程调用的不确定性，因请求超时触发消息发送重试流程，此时客户端无法感知服务端的处理结果；客户端进行的消息发送重试可能会导致消费方重复消费，应该按照用户ID、业务主键等信息幂等处理消息。
- 较多的重试次数也会增大服务端的处理压力。

4. 用户的最佳实践是什么

1）合理设置发送超时时间，发送的最大次数

发送的最大次数在初始化客户端时配置在 ClientConfiguration；对于某些实时调用类场景，可能会导致消息发送请求链路被阻塞导致业务请求整体耗时高或耗时；需要合理评估每次调用请求的超时时间以及最大重试次数，避免影响全链路的耗时。

2）如何保证发送消息不丢失

由于分布式环境的复杂性，例如网络不可达时 RocketMQ 客户端发送请求重试机制并不能保证消息发送一定成功。业务方需要捕获异常，并做好冗余保护处理，常见的解决方案有两种：

1. 向调用方返回业务处理失败；
2. 尝试将失败的消息存储到数据库，然后由后台线程定时重试，保证业务逻辑的最终一致性。

3）关注流控异常导致无法重试

触发流控的根本原因是系统容量不足，如果因为突发原因触发消息流控，且客户端内置的重试流程执行失败，则建议执行服务端扩容，将请求调用临时替换到其他系统进行应急处理。

4）早期版本客户端如何使用故障延迟机制进行发送重试？

对于 RocketMQ 4.x 和 3.x 以下客户端开启故障延迟机制可以用：

```
producer.setSendLatencyFaultEnable(true)
```

配置重试次数使用：

```
producer.setRetryTimesWhenSendFailed()
producer.setRetryTimesWhenSendAsyncFailed()
```

消费者消费重试

消息中间件做异步解耦时的一个典型问题是如果下游服务处理消息事件失败，那应该怎么做呢？

RocketMQ 的消息确认机制以及消费重试策略可以帮助分析如下问题：

- 如何保证业务完整处理消息？

消费重试策略可以在设计实现消费者逻辑时保证每条消息处理的完整性，避免部分消息消费异常导致业务状态不一致。

- 业务应用异常时处理中的消息状态如何恢复？

当系统出现异常（宕机故障）等场景时，处理中的消息状态如何恢复，消费重试具体行为是什么。

1. 什么是消费重试？

- 什么时候认为消费失败？ 消费者在接收到消息后将调用用户的消费函数执行业务逻辑。如果客户端返回消费失败 ReconsumeLater，抛出非预期异常，或消息处理超时（包括在 PushConsumer 中排队超时），只要服务端服务端一定时间内没收到响应，将认为消费失败。
- 消费重试是什么？ 消费者在消费某条消息失败后，服务端会根据重试策略重新向客户端投递该消息。超过一次定数后若还未消费成功，则该消息将不再继续重试，直接被发送到死信队列中；
- 重试过程状态机：消息在重试流程中的状态和变化逻辑；
- 重试间隔：上一次消费失败或超时后，下次重新尝试消费的间隔时间；
- 最大重试次数：消息可被重试消费的最大次数。

2. 消息重试的场景

需要注意重试是应对异常情况，给予程序再次消费失败消息的机会，不应该被用作常态化的链路。

推荐使用场景：

- 业务处理失败，失败原因跟当前的消息内容相关，预期一段时间后可执行成功；

- 是一个小概率事件，对于大批的消息只有很少量的失败，后面的消息大概率会消费成功，是非常态化的。

正例：消费逻辑是扣减库存，极少量商品因为乐观锁版本冲突导致扣减失败，重试一般立刻成功。

错误使用场景：

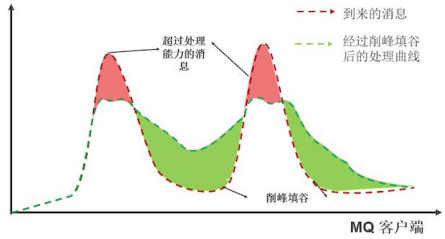
- 消费处理逻辑中使用消费失败来做条件判断的结果分流，是不合理的。

反例：订单在数据库中状态已经是已取消，此时如果收到发货的消息，处理时不应返回消费失败，而应该返回成功并标记不用发货。

- 消费处理中使用消费失败来做处理速率限流，是不合理的。限流的目的是将超出流量的消息暂时堆积在队列中达到削峰的作用，而不是让消息进入重试链路。这种做法会让消息反复在服务端和客户端之间传递，增大了系统的开销，主要包括以下方面：
  - RocketMQ 内部重试涉及写放大，每一次重试将生成新的重试消息，大量重试将带来严重的 IO 压力；
  - 重试有复杂的退避逻辑，内部实现为梯度定时器，该定时器本身不具备高吞吐的特性，大量重试将导致重试消息无法及时出队。重试的间隔将不稳定，将导致大量重试消息延后消费，即削峰的周期被大幅度延长。

3. 不要以重试替代限流

上述误用的场景实际上是组合了限流和重试能力来进行削峰，RocketMQ 推荐的削峰最佳手段为组合**限流和堆积**，业务以保护自身为前提，需要对消费流量进行限流，并利用 RocketMQ 提供的堆积能力将超出业务当前处理的消息滞后消费，以达到削峰的目的。下图中超过处理能力的消息都应该被堆积在服务端，而不是通过消费失败进行重试。

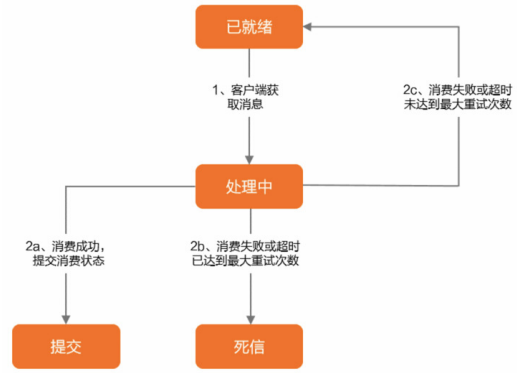


如果不想依赖额外的产品/组件来完成该功能，也可以利用一些本地工具类，比如 Guava 的 RateLimiter 来完成单机限流。如下所示，声明一个 50 QPS 的 RateLimiter，在消费前以阻塞的方式 acquire 一个令牌，获取到即处理消息，未获取到阻塞。

```
RateLimiter rateLimiter = RateLimiter.create(50);
PushConsumer pushConsumer = provider.newPushConsumerBuilder()
    .setClientConfiguration(clientConfiguration)
    // 设置订阅组名称
    .setConsumerGroup(consumerGroup)
    // 设置订阅的过滤器
    .setSubscriptionExpressions(Collections.singletonMap(topic, filterExpression))
    .setMessageListener(messageView -> {
        // 阻塞直到获得一个令牌，也可以配置一个超时时间
        rateLimiter.acquire();
        LOGGER.info("Consume message={}", messageView);
        return ConsumeResult.SUCCESS;
    })
    .build();
```

4. PushConsumer 消费重试策略

PushConsumer 消费消息时，消息的几个主要状态如下：



- Ready：已就绪状态。消息在消息队列RocketMQ版服务端已就绪，可以被消费者消费；
- Inflight：处理中状态。消息被消费者客户端获取，处于消费中还未返回消费结果的状态；
- Commit：提交状态。消费成功的状态，消费者返回成功响应即可结束消息的状态机；
- DLQ：死信状态消费逻辑的最终兜底机制，若消息一直处理失败并不断进行重试，直到超过最大重试次数还未成功，此时消息不会再重试。该消息会被投递至死信队列。您可以通过消费死信队列的消息进行业务恢复。
- 最大重试次数 ** **

PushConsumer 的最大重试次数由创建时决定。

例如，最大重试次数为 3 次，则该消息最多可被投递 4 次，1 次为原始消息，3 次为重试投递次数。

- 重试间隔时间
- 无序消息（非顺序消息）：重试间隔为阶梯时间，具体时间如下：

第几次重试	与上次重试的间隔时间	第几次重试	与上次重试的间隔时间
1	10秒	9	7分钟
2	30秒	10	8分钟
3	1分钟	11	9分钟
4	2分钟	12	10分钟
5	3分钟	13	20分钟
6	4分钟	14	30分钟
7	5分钟	15	1小时
8	6分钟	16	2小时

说明：若重试次数超过 16 次，后面每次重试间隔都为 2 小时。

- 顺序消息：重试间隔为固定时间，默认为 3 秒。

5. SimpleConsumer 消费重试策略

和 PushConsumer 消费重试策略不同，SimpleConsumer 消费者的重试间隔是预分配的，每次获取消息消费者会在调用 API 时设置一个不可见时间参数 **InvisibleDuration**，即消息的最大处理时长。若消息消费失败触发重试，不需要设置下一次重试的时间间隔，直接复用不可见时间参数的取值。

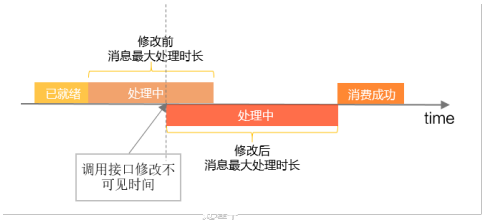
由于不可见时间为预分配的，可能和实际业务中的消息处理时间差别较大，可以通过 API 接口修改不可见时间。

例如，预设消息处理耗时最多 20 ms，但实际业务中 20 ms内消息处理不完，可以修改消息不可见时间，延长消息处理时间，避免消息触发重试机制。

修改消息不可见时间需要满足以下条件：

- 消息处理未超时
- **消息处理未提交消费状态 **

如下图所示，消息不可见时间修改后立即生效，即从调用 API 时刻开始，重新计算消息不可见时间。



- 最大重试次数

与 PushConsumer 相同。

- 消息重试间隔 ** **

消息重试间隔 = 不可见时间 - 消息实际处理时长

例如：消息不可见时间为 30 ms，实际消息处理用了 10 ms 就返回失败响应，则距下次消息重试还需要 20 ms，此时的消息重试间隔即为 20 ms；若直到 30 ms 消息还未处理完成且未返回结果，则消息超时，立即重试，此时重试间隔即为 0 ms。

SimpleConsumer 的消费重试间隔通过消息的不可见时间控制。

```
//消费示例：使用SimpleConsumer消费普通消息，主动获取消息处理并提交。
ClientServiceProvider provider1 = ClientServiceProvider.loadService();
String topic1 = "Your Topic";
FilterExpression filterExpression1 = new FilterExpression("Your Filter Tag", FilterExpressionType.TAG);
SimpleConsumer simpleConsumer = provider1.newSimpleConsumerBuilder()
        //设置消费者分组。
        .setConsumerGroup("Your ConsumerGroup")
        //设置接入点。
        .setClientConfiguration(ClientConfiguration.newBuilder().setEndpoints("Your Endpoint").build())
        //设置预绑定的订阅关系。
        .setSubscriptionExpressions(Collections.singletonMap(topic, filterExpression))
        .build();
List<MessageView> messageViewList = null;
try {
    //SimpleConsumer需要主动获取消息，并处理。
    messageViewList = simpleConsumer.receive(10, Duration.ofSeconds(30));
    messageViewList.forEach(messageView -> {
        System.out.println(messageView);
        //消费处理完成后，需要主动调用ACK提交消费结果。
        //没有ack会被认为消费失败
        try {
            simpleConsumer.ack(messageView);
        } catch (ClientException e) {
            e.printStackTrace();
        }
    });
} catch (ClientException e) {
    //如果遇到系统流控等原因造成拉取失败，需要重新发起获取消息请求。
    e.printStackTrace();
}
```

- 修改消息的不可见时间 ** **

案例：某产品使用消息队列来发送解耦“视频渲染”的业务逻辑，发送方发送任务编号，消费方收到编号后处理任务。由于消费方的业务逻辑耗时较长，消费者重新消费到同一个任务时，该任务未完成，只能返回消费失败。在这种全新的 API 下，用户可以调用可以通过修改不可见时间给消息续期，实现对单条消息状态的精确控制。

simpleConsumer.changeInvisibleDuration(); simpleConsumer.changeInvisibleDurationAsync();

6. 功能约束与最佳实践

- 设置消费的最大超时时间和次数 ** **

尽快明确的向服务端返回成功或失败，不要以超时（有时是异常抛出）代替消费失败。

- **不要用重试机制来进行业务限流 **

错误示例：如果当前消费速度过高触发限流，则返回消费失败，等待下次重新消费。

正确示例：如果当前消费速度过高触发限流，则延迟获取消息，稍后再消费。

- **发送重试和消费重试会导致相同的消息重复消费，消费方应该有一个良好的幂等设计 **

正确示例：某系统中消费的逻辑是为某个用户发送短信，该短信已经发送成功了，当消费者应用重复收到该消息，此时应该返回消费成功。

总结

本文主要介绍重试的基本概念，生产者消费者收发消息时触发重试的条件和具体行为，以及 RocketMQ 收发容错的最佳实践。

重试策略帮助我们从随机的、短暂的瞬态故障中恢复，是在容忍错误时，提高可用性的一种强大机制。但请谨记“重试是对于分布式系统来说自私的”，因为客户端认为其请求很重要，并要求服务端花费更多资源来处理，盲目的重试设计不可取，合理的使用重试可以帮助我们构建更加弹性且可靠的系统。

活动推荐

阿里云基于 Apache RocketMQ 构建的企业级产品-消息队列RocketMQ 5.0版现开启活动：

1、新用户首次购买包年包月，即可享受全系列 85折优惠！ 了解活动详情：https://www.aliyun.com/product/rocketmq


更多内容：
欢迎钉钉扫描二维码加入
「Apache RocketMQ 中国开发者钉钉群」
欢迎扫描二维码
关注Apache RocketMQ公众号