RocketMQ 最佳实践

v3.0.0

2013/10/7

文档变更历史

序号	主要更改内容	更改人	更改时间
1	建立初始版本	誓嘉 vintage.wang@gmail.com	2013/9/23
2			
3			
4			
5			
6			
7			

目录

1 前言

本文档旨在描述 RocketMQ 使用过程中的一些最佳实践，建议用户这样做，但是非必须。

2 Producer 最佳实践

2.1 发送消息注意事项

1. 一个应用尽可能用一个 Topic，消息子类型用 tags 来标识，tags 可以由应用自由设置。只有发送消息设置了 tags，消费方在订阅消息时，才可以利用 tags 在 broker 做消息过滤。

   message.setTags("TagA");

2. 每个消息在业务层面的唯一标识码，要设置到 keys 字段，方便将来定位消息丢失问题。服务器会为每个消息创建索引（哈希索引），应用可以通过 topic，key 来查询这条消息内容，以及消息被谁消费。由于是哈希索引，请务必保证 key 尽可能唯一，这样可以避免潜在的哈希冲突。

   ```
   // 订单 Id
   String orderId = "20034568923546";
   message.setKeys(orderId);
   ```

3. 消息发送成功或者失败，要打印消息日志，务必要打印 sendresult 和 key 字段。

4. send 消息方法，只要不抛异常，就代表发送成功。但是发送成功会有多个状态，在 sendResult 里定义。

   - SEND_OK

     消息发送成功

   - FLUSH_DISK_TIMEOUT

     消息发送成功，但是服务器刷盘超时，消息已经进入服务器队列，只有此时服务器宕机，消息才会丢失

   - FLUSH_SLAVE_TIMEOUT

     消息发送成功，但是服务器同步到 Slave 时超时，消息已经进入服务器队列，只有此时服务器宕机，消息才会丢失

   - SLAVE_NOT_AVAILABLE

     消息发送成功，但是此时 slave 不可用，消息已经进入服务器队列，只有此时服务器宕机，消息才会丢

失

对于精卫发送顺序消息的应用，由于顺序消息的局限性，可能会涉及到主备自动切换问题，所以如果 sendresult 中的 status 字段不等于 SEND_OK，就应该尝试重试。对于其他应用，则没有必要这样。

5. 对于消息不可丢失应用，务必要有消息重发机制

   例如如果消息发送失败，存储到数据库，能有定时程序尝试重发，或者人工触发重发。

2.2 消息发送失败如何处理

Producer 的 send 方法本身支持内部重试，重试逻辑如下：

1. 至多重试 3 次。
2. 如果发送失败，则轮转到下一个 Broker。
3. 这个方法的总耗时时间不超过 sendMsgTimeout 设置的值，默认 10s。

   所以，如果本身向 broker 发送消息产生超时异常，就不会再做重试。

以上策略仍然不能保证消息一定发送成功，为保证消息一定成功，建议应用这样做

如果调用 send 同步方法发送失败，则尝试将消息存储到 db，由后台线程定时重试，保证消息一定到达 Broker。

上述 db 重试方式为什么没有集成到 MQ 客户端内部做，而是要求应用自己去完成，我们基于以下几点考虑

1. MQ 的客户端设计为无状态模式，方便任意的水平扩展，且对机器资源的消耗仅仅是 cpu、内存、网络。
2. 如果 MQ 客户端内部集成一个 KV 存储模块，那么数据只有同步落盘才能较可靠，而同步落盘本身性能开销较大，所以通常会采用异步落盘，又由于应用关闭过程不受 MQ 运维人员控制，可能经常会发生 kill -9 这样暴力方式关闭，造成数据没有及时落盘而丢失。
3. Producer 所在机器的可靠性较低，一般为虚拟机，不适合存储重要数据。

综上，建议重试过程交由应用来控制。

2.3 选择 oneway 形式发送

一个 RPC 调用，通常是这样一个过程

1. 客户端发送请求到服务器
2. 服务器处理该请求
3. 服务器向客户端返回应答

所以一个 RPC 的耗时时间是上述三个步骤的总和，而某些场景要求耗时非常短，但是对可靠性要求并不高，例如日志收集类应用，此类应用可以采用 oneway 形式调用，oneway 形式只发送请求不等待应答，而发送请求在客户端实现层面仅仅是一个 os 系统调用的开销，即将数据写入客户端的 socket 缓冲区，此过程耗时通常在微秒级。

2.4 发送顺序消息注意事项

3 Consumer 最佳实践

3.1 消费过程要做到幂等（即消费端去重）

如《RocketMQ 原理简介》中所述，RocketMQ 无法避免消息重复，所以如果业务对消费重复非常敏感，务必要在业务层面去重，有以下几种去重方式

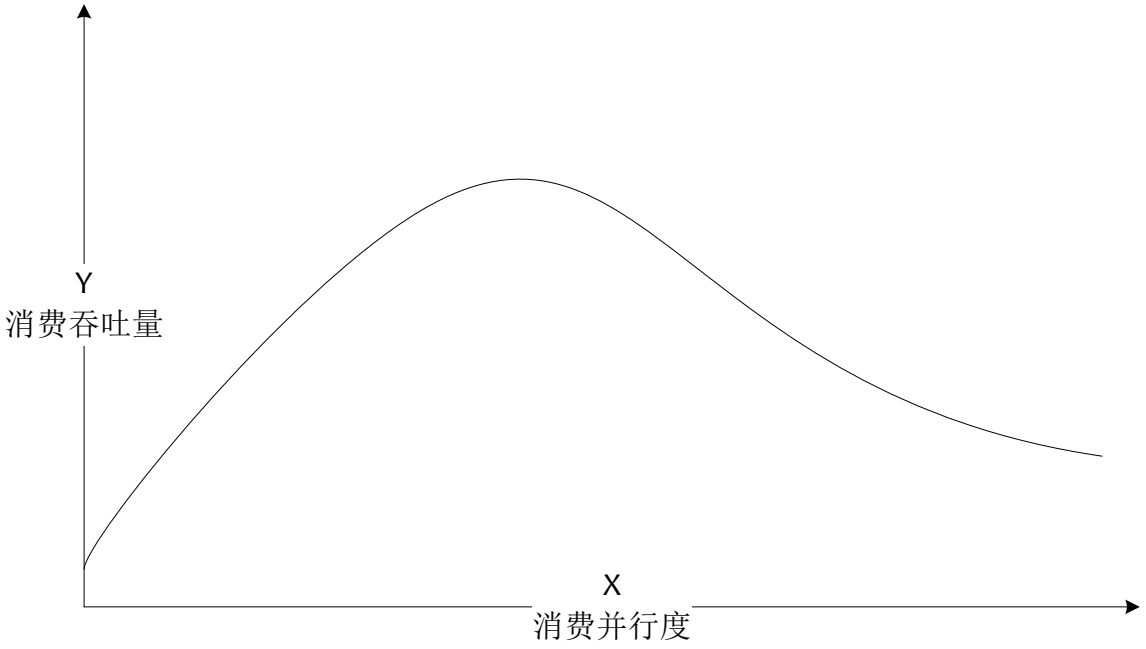
1. 将消息的唯一键，可以是 msgId，也可以是消息内容中的唯一标识字段，例如订单 Id 等，消费之前判断是否在 Db 或 Tair(全局 KV 存储)中存在，如果不存在则插入，并消费，否则跳过。（实际过程要考虑原子性问题，判断是否存在可以尝试插入，如果报主键冲突，则插入失败，直接跳过）

   msgId 一定是全局唯一标识符，但是可能会存在同样的消息有两个不同 msgId 的情况（有多种原因），这种情况可能会使业务上重复消费，建议最好使用消息内容中的唯一标识字段去重。
2. 使用业务层面的状态机去重

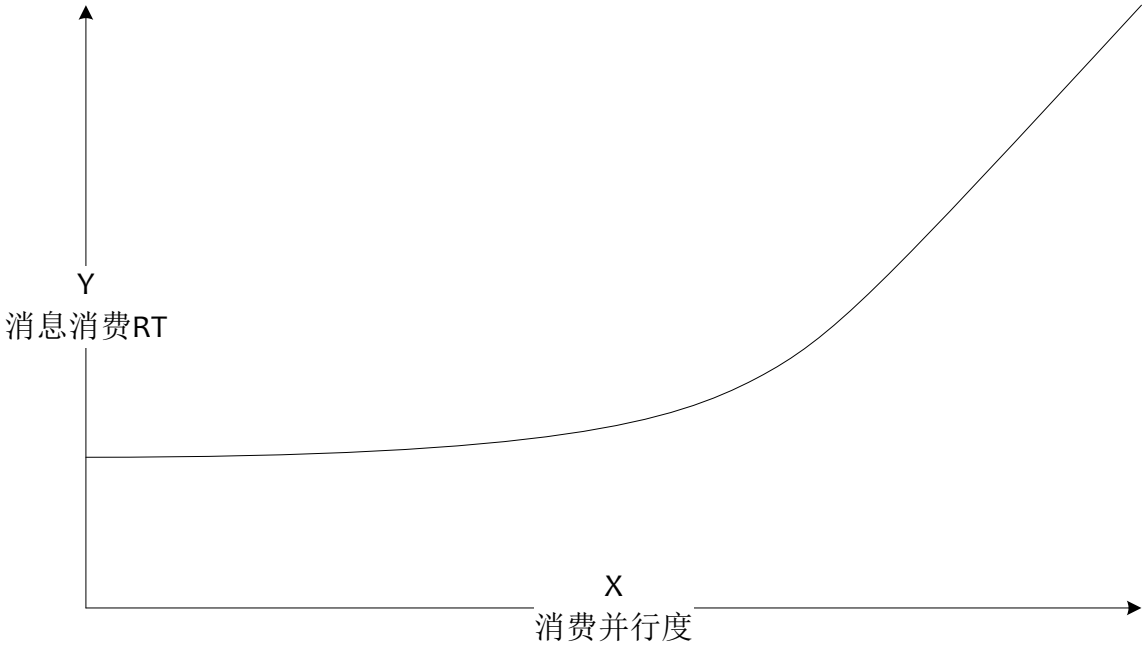
3.2 消费失败处理方式

3.3 消费速度慢处理方式

3.3.1 提高消费并行度



3-1 消费并行度与消费吞吐量关系



3-2 消费并行度与消费 RT 关系

绝大部分消息消费行为属于 IO 密集型，即可能是操作数据库，或者调用 RPC，这类消费行为的消费速度在于后端数据库或者外系统的吞吐量，通过增加消费并行度，可以提高总的消费吞吐量，但是并行度增加到一定程度，反而

会下降，如图所示，呈现抛物线形式。

所以应用必须要设置合理的并行度。

CPU 密集型应用除外。

3.3.2 批量方式消费

某些业务流程如果支持批量方式消费，则可以很大程度上提高消费吞吐量，例如订单扣款类应用，一次处理一个订单耗时 1 秒钟，一次处理 10 个订单可能也只耗时 2 秒钟，这样即可大幅度提高消费的吞吐量，通过设置 consumer 的 consumeMessageBatchMaxSize 这个参数，默认是 1，即一次只消费一条消息，例如设置为 N，那么每次消费的消息数小于等于 N。

3.3.3 跳过非重要消息

发生消息堆积时，如果消费速度一直追不上发送速度，可以选择丢弃不重要的消息

如何判断消费发生了堆积？

```
    public ConsumeConcurrentlyStatus consumeMessage(//
            List<MessageExt> msgs, //
            ConsumeConcurrentlyContext context) {
        long offset = msgs.get(0).getQueueOffset();
        String maxOffset = //
                msgs.get(0).getProperty(Message.PROPERTY_MAX_OFFSET);
        long diff = Long.parseLong(maxOffset) - offset;
        if (diff > 100000) {
            // TODO 消息堆积情况的特殊处理
            return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
        }

        // TODO 正常消费过程
        return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
```

```
    }
```

如以上代码所示，当某个队列的消息数堆积到 100000 条以上，则尝试丢弃部分或全部消息，这样就可以快速追上发送消息的速度。

3.3.4 优化每条消息消费过程

举例如下，某条消息的消费过程如下

1. 根据消息从 DB 查询数据 1
2. 根据消息从 DB 查询数据 2
3. 复杂的业务计算
4. 向 DB 插入数据 3
5. 向 DB 插入数据 4

这条消息的消费过程与 DB 交互了 4 次，如果按照每次 5ms 计算，那么总共耗时 20ms，假设业务计算耗时 5ms，那么总过耗时 25ms，如果能把 4 次 DB 交互优化为 2 次，那么总耗时就可以优化到 15ms，也就是说总体性能提高了 40%。

对于 Mysql 等 DB，如果部署在磁盘，那么与 DB 进行交互，如果数据没有命中 cache，每次交互的 RT 会直线上升，如果采用 SSD，则 RT 上升趋势要明显好于磁盘。个别应用可能会遇到这种情况：

在线下压测消费过程中，db 表现非常好，每次 RT 都很短，但是上线运行一段时间，RT 就会变长，消费吞吐量直线下降。

主要原因是线下压测时间过短，线上运行一段时间后，cache 命中率下降，那么 RT 就会增加。建议在线下压测时，要测试足够长时间，尽可能模拟线上环境，压测过程中，数据的分布也很重要，数据不同，可能 cache 的命中率也会完全不同。

3.4 消费打印日志

如果消息量较少，建议在消费入口方法打印消息，方便后续排查问题。

```
    public ConsumeConcurrentlyStatus consumeMessage(//
            List<MessageExt> msgs, //
            ConsumeConcurrentlyContext context) {
        log.info("RECEIVE_MSG_BEGIN: " + msgs.toString());
        // TODO 正常消费过程
        return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
    }
```

如果能打印每条消息消费耗时，那么在排查消费慢等线上问题时，会更方便。

3.5 利用服务器消息过滤，避免多余的消息传输

4 新上线一个应用需要注意什么