|  |
| --- |
|  |
| **RocketMQ最佳实践** |
| v3.0.0 |
|  |
| ©Alibaba 淘宝消息中间件项目组 |
| 2013/10/5 |

**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 1](#_Toc368741388)

[2 Producer最佳实践 1](#_Toc368741389)

[2.1 发送消息注意事项 1](#_Toc368741390)

[2.2 消息发送失败如何处理 2](#_Toc368741391)

[2.3 选择oneway形式发送 3](#_Toc368741392)

[2.4 发送顺序消息注意事项 3](#_Toc368741393)

[3 Consumer最佳实践 3](#_Toc368741394)

[3.1 消费过程要做到幂等（即消费端去重） 3](#_Toc368741395)

[3.2 消费失败处理方式 4](#_Toc368741396)

[3.3 消费速度慢处理方式 4](#_Toc368741397)

[3.3.1 提高消费并行度 4](#_Toc368741398)

[3.3.2 批量方式消费 5](#_Toc368741399)

[3.3.3 跳过非重要消息 5](#_Toc368741400)

[3.3.4 优化每条消息消费过程 6](#_Toc368741401)

[3.4 消费打印日志 6](#_Toc368741402)

[3.5 利用服务器消息过滤，避免多余的消息传输 7](#_Toc368741403)

[4 新上线一个应用需要注意什么 7](#_Toc368741404)

# 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。

# Producer最佳实践

## 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");

1. 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);

1. 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。
2. 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。

1. 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。

## 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。

## 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。

## 发送顺序消息注意事项

# Consumer最佳实践

## 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。

1. 使用业务层面的状态机去重

## 消费失败处理方式

## 消费速度慢处理方式

### 提高消费并行度



‑1消费并行度与消费吞吐量关系



‑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。

### 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。

### 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。

### 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。

## 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。

## 利用服务器消息过滤，避免多余的消息传输

# 新上线一个应用需要注意什么