火龙果•整理
uml.org.cn

RocketMQ 原理简介

v3.0.0

©Alibaba 淘宝消息中间件项目组

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	2013/5/18
2	3.0 版本补充文档	誓嘉	2013/8/16
3			
4			
5			
6			
7			

目录

1 前言 1

2 产品发展历史 1

3 专业术语 2

4 消息中间件需要解决哪些问题？ 4

4.1 Publish/Subscribe 4
4.2 Message Priority 4
4.3 Message Order 4
4.4 Message Filter 5
4.5 Message Persistence 5
4.6 Message Reliablity 6
4.7 Low Latency Messaging 6
4.8 At least Once 6
4.9 Exactly Only Once 7
4.10 Broker 的 Buffer 满了怎么办？ 7
4.11 回溯消费 7
4.12 消息堆积 8
4.13 分布式事务 8
4.14 定时消息 9
4.15 消息重试 9

5 RocketMQ Overview 10

5.1 RocketMQ 是什么？ 10
5.2 RocketMQ 部署结构 11

6 RocketMQ 存储特点 12

6.1 零拷贝原理 12
6.2 文件系统 12
6.3 数据存储结构 13

7 RocketMQ 关键特性 14

7.1 单机支持 1 万以上持久化队列 14
7.2 刷盘策略 15
7.2.1 异步刷盘 16
7.2.2 同步刷盘 17
7.3 消息查询 17
7.3.1 按照 Message Id 查询消息 17
7.3.2 按照 Message Key 查询消息 18
7.4 服务器消息过滤 19
7.5 长轮询 Pull 20
7.6 顺序消息 20
7.6.1 顺序消息原理 20
7.6.2 顺序消息缺陷 20
7.7 事务消息 20
7.8 发送消息负载均衡 21
7.9 订阅消息负载均衡 21
7.10 单队列并行消费 22
7.11 发送定时消息 22
7.12 消息消费失败，定时重试 22
7.13 HA，同步双写/异步复制 22
7.14 单个 JVM 进程也能利用机器超大内存 23
7.15 消息堆积问题解决办法 24
8 RocketMQ 通信组件 24
8.1 网络协议 25
8.2 心跳处理 26

8.3 连接复用 26

8.4 超时连接 26

9 RocketMQ 服务发现（Name Server） 26

附录 A 参考文档、规范 27

1 前言

本文档旨在描述 RocketMQ 的多个关键特性的实现原理，并对消息中间件遇到的各种问题进行总结，阐述 RocketMQ 如何解决这些问题。文中主要引用了 JMS 规范与 CORBA Notification 规范，规范为我们设计系统指明了方向，但是仍有不少问题规范没有提及，对于消息中间件又至关重要。RocketMQ 并不遵循任何规范，但是参考了各种规范与同类产品的设计思想。

2 产品发展历史

大约经历了三个主要版本迭代

一、Metaq（Metamorphosis） 1.x

由开源社区 killme2008 维护，开源社区非常活跃。

https://github.com/killme2008/Metamorphosis

二、Metaq 2.x

于 2012 年 10 月份上线，在淘宝内部被广泛使用。

三、RocketMQ 3.x

基于公司内部开源共建原则， RocketMQ 项目只维护核心功能，且去除了所有其他运行时依赖，核心功能最简化。每个 BU 的个性化需求都在 RocketMQ 项目之上进行深度定制。RocketMQ 向其他 BU 提供的仅仅是 Jar 包，例如要定制一个 Broker，那么只需要依赖 rocketmq-broker 这个 jar 包即可，可通过 API 进行交互，如果定制 client，则依赖 rocketmq-client 这个 jar 包，对其提供的 api 进行再封装。

开源社区地址：

https://github.com/alibaba/RocketMQ

在 RocketMQ 项目基础上衍生的项目如下

- com.taobao.metaq v3.0 = RocketMQ + 淘宝个性化需求

  为淘宝应用提供消息服务

- com.alipay.zpullmsg v1.0 = RocketMQ + 支付宝个性化需求

  为支付宝应用提供消息服务

- com.alibaba.commonmq v1.0 = Notify + RocketMQ + B2B 个性化需求

  为 B2B 应用提供消息服务

3 专业术语

- **Producer**

  消息生产者，负责产生消息，一般由业务系统负责产生消息。

- **Consumer**

  消息消费者，负责消费消息，一般是后台系统负责异步消费。

- **Push Consumer**

  Consumer 的一种，应用通常向 Consumer 对象注册一个 Listener 接口，一旦收到消息，Consumer 对象立刻回调 Listener 接口方法。

- **Pull Consumer**

  Consumer 的一种，应用通常主动调用 Consumer 的拉消息方法从 Broker 拉消息，主动权由应用控制。

- **Producer Group**

  一类 Producer 的集合名称，这类 Producer 通常发送一类消息，且发送逻辑一致。

- **Consumer Group**

  一类 Consumer 的集合名称，这类 Consumer 通常消费一类消息，且消费逻辑一致。

- **Broker**

  消息中转角色，负责存储消息，转发消息，一般也称为 Server。在 JMS 规范中称为 Provider。

- **广播消费**

  一条消息被多个 Consumer 消费，即使这些 Consumer 属于同一个 Consumer Group，消息也会被 Consumer Group 中的每个 Consumer 都消费一次，广播消费中的 Consumer Group 概念可以认为在消息划分方面无意义。

  在 CORBA Notification 规范中，消费方式都属于广播消费。

- **集群消费**

一个 Consumer Group 中的 Consumer 实例平均分摊消费消息。例如某个 Topic 有 9 条消息，其中一个 Consumer Group 有 3 个实例（可能是 3 个进程，或者 3 台机器），那么每个实例只消费其中的 3 条消息。

- **顺序消息**

消费消息的顺序要同发送消息的顺序一致，在 RocketMQ 中，主要指的是局部顺序，即一类消息为满足顺序性，必须 Producer 单线程顺序发送，且发送到同一个队列，这样 Consumer 就可以按照 Producer 发送的顺序去消费消息。

- **普通顺序消息**

顺序消息的一种，正常情况下可以保证完全的顺序消息，但是一旦发生通信异常，Broker 重启，由于队列总数发生变化，哈希取模后定位的队列会变化，产生短暂的消息顺序不一致。

如果业务能容忍在集群异常情况（如某个 Broker 宕机或者重启）下，消息短暂的乱序，使用普通顺序方式比较合适。

- **严格顺序消息**

顺序消息的一种，无论正常异常情况都能保证顺序，但是牺牲了分布式 Failover 特性，即 Broker 集群中只要有一台机器不可用，则整个集群都不可用，服务可用性大大降低。

如果服务器部署为同步双写模式，此缺陷可通过备机自动切换为主避免，不过仍然会存在几分钟的服务不可用。（依赖同步双写，主备自动切换，自动切换功能目前还未实现）

目前已知的应用只有数据库 binlog 同步强依赖严格顺序消息，其他应用绝大部分都可以容忍短暂乱序，推荐使用普通的顺序消息。

- **Message Queue**

在 RocketMQ 中，所有消息队列都是持久化，长度无限的数据结构，所谓长度无限是指队列中的每个存储单元都是定长，访问其中的存储单元使用 Offset 来访问，offset 为 java long 类型，64 位，理论上在 100 年内不会溢出，所以认为是长度无限，另外队列中只保存最近几天的数据，之前的数据会按照过期时间来删除。

也可以认为 Message Queue 是一个长度无限的数组，offset 就是下标。

4 消息中间件需要解决哪些问题？

本节阐述消息中间件通常需要解决哪些问题，在解决这些问题当中会遇到什么困难，RocketMQ 是否可以解决，规范中如何定义这些问题。

4.1 Publish/Subscribe

发布订阅是消息中间件的最基本功能，也是相对于传统 RPC 通信而言。在此不再详述。

4.2 Message Priority

规范中描述的优先级是指在一个消息队列中，每条消息都有不同的优先级，一般用整数来描述，优先级高的消息先投递，如果消息完全在一个内存队列中，那么在投递前可以按照优先级排序，令优先级高的先投递。

由于 RocketMQ 所有消息都是持久化的，所以如果按照优先级来排序，开销会非常大，因此 RocketMQ 没有特意支持消息优先级，但是可以通过变通的方式实现类似功能，即单独配置一个优先级高的队列，和一个普通优先级的队列， 将不同优先级发送到不同队列即可。

对于优先级问题，可以归纳为 2 类

1) 只要达到优先级目的即可，不是严格意义上的优先级，通常将优先级划分为高、中、低，或者再多几个级别。每个优先级可以用不同的 topic 表示，发消息时，指定不同的 topic 来表示优先级，这种方式可以解决绝大部分的优先级问题，但是对业务的优先级精确性做了妥协。

2) 严格的优先级，优先级用整数表示，例如 0～65535，这种优先级问题一般使用不同 topic 解决就非常不合适。如果要让 MQ 解决此问题，会对 MQ 的性能造成非常大的影响。这里要确保一点，业务上是否确实需要这种严格的优先级，如果将优先级压缩成几个，对业务的影响有多大？

4.3 Message Order

消息有序指的是一类消息消费时，能按照发送的顺序来消费。例如：一个订单产生了 3 条消息，分别是订单创建，订单付款，订单完成。消费时，要按照这个顺序消费才能有意义。但是同时订单之间是可以并行消费的。

RocketMQ 可以严格的保证消息有序。

4.4 Message Filter

- Broker 端消息过滤

  在 Broker 中，按照 Consumer 的要求做过滤，优点是减少了对于 Consumer 无用消息的网络传输。

  缺点是增加了 Broker 的负担，实现相对复杂。

  (1). 淘宝 Notify 支持多种过滤方式，包含直接按照消息类型过滤，灵活的语法表达式过滤，几乎可以满足最苛刻的过滤需求。

  (2). 淘宝 RocketMQ 只支持按照简单的 Message Tag 过滤。

  (3). CORBA Notification 规范中也支持灵活的语法表达式过滤。

- Consumer 端消息过滤

  这种过滤方式可由应用完全自定义实现，但是缺点是很多无用的消息要传输到 Consumer 端。

4.5 Message Persistence

几种持久化方式：

(1). 持久化到数据库，例如 Mysql。

(2). 持久化到 KV 存储，例如 levelDB、伯克利 DB 等 KV 存储系统。

(3). 文件记录形式持久化，例如 Kafka，RocketMQ

(4). 对内存数据做一个持久化镜像，例如 beanstalkd，VisiNotify

(1)、(2)、(3)三种持久化方式都具有将内存队列 Buffer 进行扩展的能力，(4)只是一个内存的镜像，作用是当 Broker 挂掉重启后仍然能将之前内存的数据恢复出来。

JMS 与 CORBA Notification 规范没有明确说明如何持久化，但是持久化部分的性能直接决定了整个消息中间件的性能。

4.6 Message Reliablity

影响消息可靠性的几种情况：

(1). Broker 正常关闭

(2). Broker 异常 Crash

(3). OS Crash

(4). 机器掉电，但是能立即恢复供电情况。

(5). 机器无法开机（可能是 cpu、主板、内存等关键设备损坏）

(6). 磁盘设备损坏。

(1)、(2)、(3)、(4)四种情况都属于硬件资源可立即恢复情况，RocketMQ 在这四种情况下能保证消息不丢，或者丢失少量数据（依赖刷盘方式是同步还是异步）。

(5)、(6)属于单点故障，且无法恢复，一旦发生，在此单点上的消息全部丢失。RocketMQ 在这两种情况下，通过异步复制，可保证 99%的消息不丢，但是仍然会有极少量的消息可能丢失。通过同步双写技术可以完全避免单点，同步双写势必会影响性能，适合对消息可靠性要求极高的场合，例如与 Money 相关的应用。

RocketMQ 从 3.0 版本开始支持同步双写。

4.7 Low Latency Messaging

在消息不堆积情况下，消息到达 Broker 后，能立刻到达 Consumer。

RocketMQ 使用长轮询 Pull 方式，可保证消息非常实时，消息实时性不低于 Push。

4.8 At least Once

是指每个消息必须投递一次

RocketMQ Consumer 先 pull 消息到本地，消费完成后，才向服务器返回 ack，如果没有消费一定不会 ack 消息，所以 RocketMQ 可以很好的支持此特性。

4.9 Exactly Only Once

(1). 发送消息阶段，不允许发送重复的消息。

(2). 消费消息阶段，不允许消费重复的消息。

只有以上两个条件都满足情况下，才能认为消息是“Exactly Only Once”，而要实现以上两点，在分布式系统环境下，不可避免要产生巨大的开销。所以 RocketMQ 为了追求高性能，并不保证此特性，要求在业务上进行去重，也就是说消费消息要做到幂等性。RocketMQ 虽然不能严格保证不重复，但是正常情况下很少会出现重复发送、消费情况，只有网络异常，Consumer 启停等异常情况下会出现消息重复。

此问题的本质原因是网络调用存在不确定性，即既不成功也不失败的第三种状态，所以才产生了消息重复性问题。

4.10 Broker 的 Buffer 满了怎么办？

Broker 的 Buffer 通常指的是 Broker 中一个队列的内存 Buffer 大小，这类 Buffer 通常大小有限，如果 Buffer 满了以后怎么办？

下面是 CORBA Notification 规范中处理方式：

(1). RejectNewEvents

拒绝新来的消息，向 Producer 返回 RejectNewEvents 错误码。

(2). 按照特定策略丢弃已有消息

a) **AnyOrder** - Any event may be discarded on overflow. This is the default setting for this property.
b) **FifoOrder** - The first event received will be the first discarded.
c) **LifoOrder** - The last event received will be the first discarded.
d) **PriorityOrder** - Events should be discarded in priority order, such that lower priority events will be discarded before higher priority events.
e) **DeadlineOrder** - Events should be discarded in the order of shortest expiry deadline first.

RocketMQ 没有内存 Buffer 概念，RocketMQ 的队列都是持久化磁盘，数据定期清除。

4.11 回溯消费

回溯消费是指 Consumer 已经消费成功的消息，由于业务上需求需要重新消费，要支持此功能，Broker 在向

Consumer 投递成功消息后，消息仍然需要保留。并且重新消费一般是按照时间维度，例如由于 Consumer 系统故障，恢复后需要重新消费 1 小时前的数据，那么 Broker 要提供一种机制，可以按照时间维度来回退消费进度。

RocketMQ 支持按照时间回溯消费，时间维度精确到毫秒，可以向前回溯，也可以向后回溯。

4.12 消息堆积

消息中间件的主要功能是异步解耦，还有个重要功能是挡住前端的数据洪峰，保证后端系统的稳定性，这就要求消息中间件具有一定的消息堆积能力，消息堆积分以下两种情况：

(1). 消息堆积在内存 Buffer，一旦超过内存 Buffer，可以根据一定的丢弃策略来丢弃消息，如 CORBA Notification 规范中描述。适合能容忍丢弃消息的业务，这种情况消息的堆积能力主要在于内存 Buffer 大小，而且消息堆积后，性能下降不会太大，因为内存中数据多少对于对外提供的访问能力影响有限。

(2). 消息堆积到持久化存储系统中，例如 DB，KV 存储，文件记录形式。

当消息不能在内存 Cache 命中时，要不可避免的访问磁盘，会产生大量读 IO，读 IO 的吞吐量直接决定了消息堆积后的访问能力。

评估消息堆积能力主要有以下四点：

(1). 消息能堆积多少条，多少字节？即消息的堆积容量。

(2). 消息堆积后，发消息的吞吐量大小，是否会受堆积影响？

(3). 消息堆积后，正常消费的 Consumer 是否会受影响？

(4). 消息堆积后，访问堆积在磁盘的消息时，吞吐量有多大？

4.13 分布式事务

已知的几个分布式事务规范，如 XA，JTA 等。其中 XA 规范被各大数据库厂商广泛支持，如 Oracle，Mysql 等。其中 XA 的 TM 实现佼佼者如 Oracle Tuxedo，在金融、电信等领域被广泛应用。

分布式事务涉及到两阶段提交问题，在数据存储方面的方面必然需要 KV 存储的支持，因为第二阶段的提交回滚需要修改消息状态，一定涉及到根据 Key 去查找 Message 的动作。RocketMQ 在第二阶段绕过了根据 Key 去查找

Message 的问题，采用第一阶段发送 Prepared 消息时，拿到了消息的 Offset，第二阶段通过 Offset 去访问消息，并修改状态，Offset 就是数据的地址。

RocketMQ 这种实现事务方式，没有通过 KV 存储做，而是通过 Offset 方式，存在一个显著缺陷，即通过 Offset 更改数据，会令系统的脏页过多，需要特别关注。

4.14 定时消息

定时消息是指消息发到 Broker 后，不能立刻被 Consumer 消费，要到特定的时间点或者等待特定的时间后才能被消费。

如果要支持任意的时间精度，在 Broker 层面，必须要做消息排序，如果再涉及到持久化，那么消息排序要不可避免的产生巨大性能开销。

RocketMQ 支持定时消息，但是不支持任意时间精度，支持特定的 level，例如定时 5s，10s，1m 等。

4.15 消息重试

Consumer 消费消息失败后，要提供一种重试机制，令消息再消费一次。Consumer 消费消息失败通常可以认为有以下几种情况

1. 由于消息本身的原因，例如反序列化失败，消息数据本身无法处理（例如话费充值，当前消息的手机号被注销，无法充值）等。

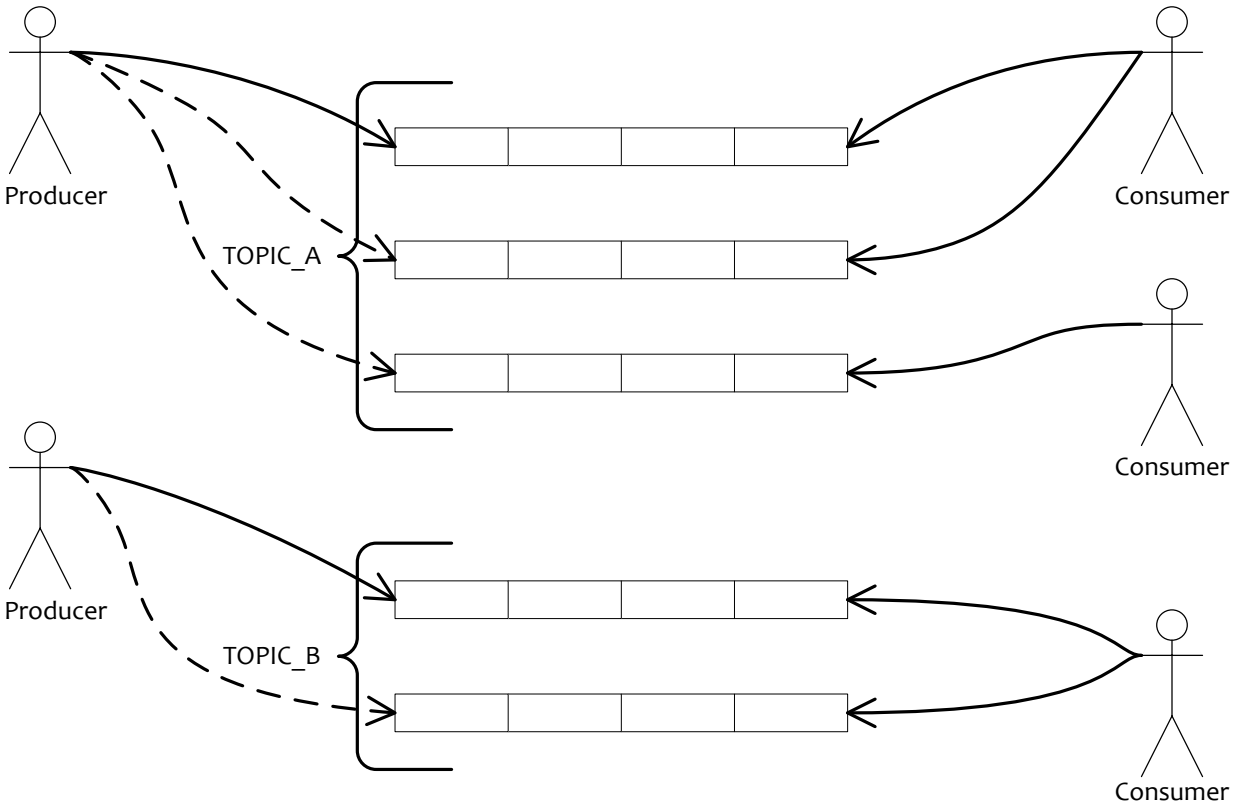
   这种错误通常需要跳过这条消息，再消费其他消息，而这条失败的消息即使立刻重试消费，99%也不成功，所以最好提供一种定时重试机制，即过 10s 秒后再重试。

2. 由于依赖的下游应用服务不可用，例如 db 连接不可用，外系统网络不可达等。

   遇到这种错误，即使跳过当前失败的消息，消费其他消息同样也会报错。这种情况建议应用 sleep 30s，再消费下一条消息，这样可以减轻 Broker 重试消息的压力。

5 RocketMQ Overview

5.1 RocketMQ 是什么？

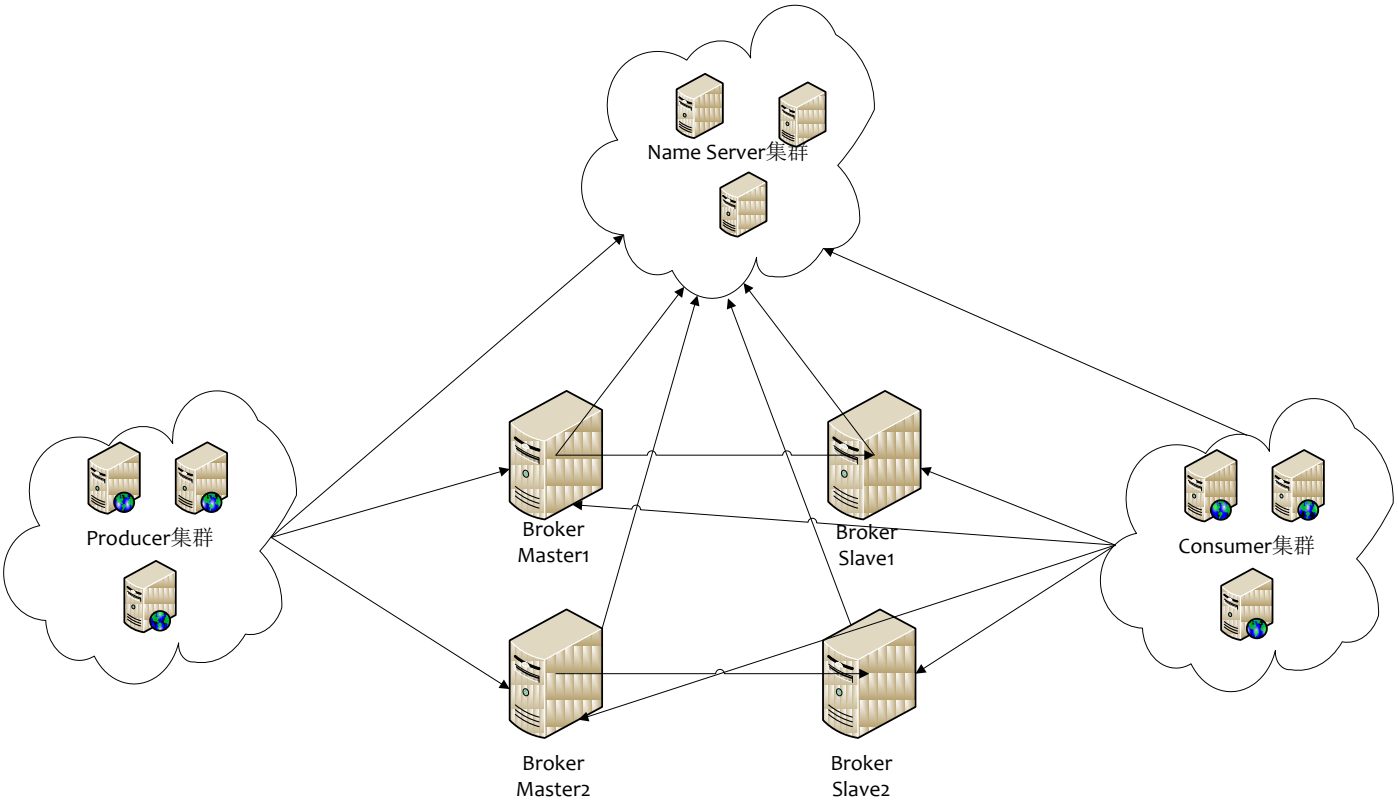


图表 5-1 RocketMQ 是什么

- 是一个队列模型的消息中间件，具有高性能、高可靠、高实时、分布式特点。
- Producer、Consumer、队列都可以分布式。
- Producer 向一些队列轮流发送消息，队列集合称为 Topic，Consumer 如果做广播消费，则一个 consumer 实例消费这个 Topic 对应的所有队列，如果做集群消费，则多个 Consumer 实例平均消费这个 topic 对应的队列集合。
- 能够保证严格的消息顺序
- 提供丰富的消息拉取模式
- 高效的订阅者水平扩展能力
- 实时的消息订阅机制
- 亿级消息堆积能力

- 较少的依赖

5.2 RocketMQ 部署结构



图表 5-2RocketMQ 网络部署图

RocketMQ 网络部署特点

- Name Server 是一个几乎无状态节点，可集群部署，节点之间无任何信息同步。
- Broker 部署相对复杂，Broker 分为 Master 与 Slave，一个 Master 可以对应多个 Slave，但是一个 Slave 只能对应一个 Master，Master 与 Slave 的对应关系通过指定相同的 BrokerName，不同的 BrokerId 来定义，BrokerId 为 0 表示 Master，非 0 表示 Slave。Master 也可以部署多个。每个 Broker 与 Name Server 集群中的所有节点建立长连接，定时注册 Topic 信息到所有 Name Server。
- Producer 与 Name Server 集群中的其中一个节点（随机选择）建立长连接，定期从 Name Server 取 Topic 路由信息，并向提供 Topic 服务的 Master 建立长连接，且定时向 Master 发送心跳。Producer 完全无状态，可集群部署。
- Consumer 与 Name Server 集群中的其中一个节点（随机选择）建立长连接，定期从 Name Server 取 Topic 路

由信息，并向提供 Topic 服务的 Master、Slave 建立长连接，且定时向 Master、Slave 发送心跳。Consumer 既可以从 Master 订阅消息，也可以从 Slave 订阅消息，订阅规则由 Broker 配置决定。

6 RocketMQ 存储特点

6.1 零拷贝原理

Consumer 消费消息过程，使用了零拷贝，零拷贝包含以下两种方式

1. 使用 mmap + write 方式

   优点：即使频繁调用，使用小块文件传输，效率也很高

   缺点：不能很好的利用 DMA 方式，会比 sendfile 多消耗 CPU，内存安全性控制复杂，需要避免 JVM Crash 问题。

2. 使用 sendfile 方式

   优点：可以利用 DMA 方式，消耗 CPU 较少，大块文件传输效率高，无内存安全新问题。

   缺点：小块文件效率低于 mmap 方式，只能是 BIO 方式传输，不能使用 NIO。

RocketMQ 选择了第一种方式，mmap+write 方式，因为有小块数据传输的需求，效果会比 sendfile 更好。

关于 Zero Copy 的更详细介绍，请参考以下文章

http://www.linuxjournal.com/article/6345

6.2 文件系统

RocketMQ 选择 Linux Ext4 文件系统，原因如下：

Ext4 文件系统删除 1G 大小的文件通常耗时小于 50ms，而 Ext3 文件系统耗时约 1s 左右，且删除文件时，磁盘 IO 压力极大，会导致 IO 写入超时。

文件系统层面需要做以下调优措施

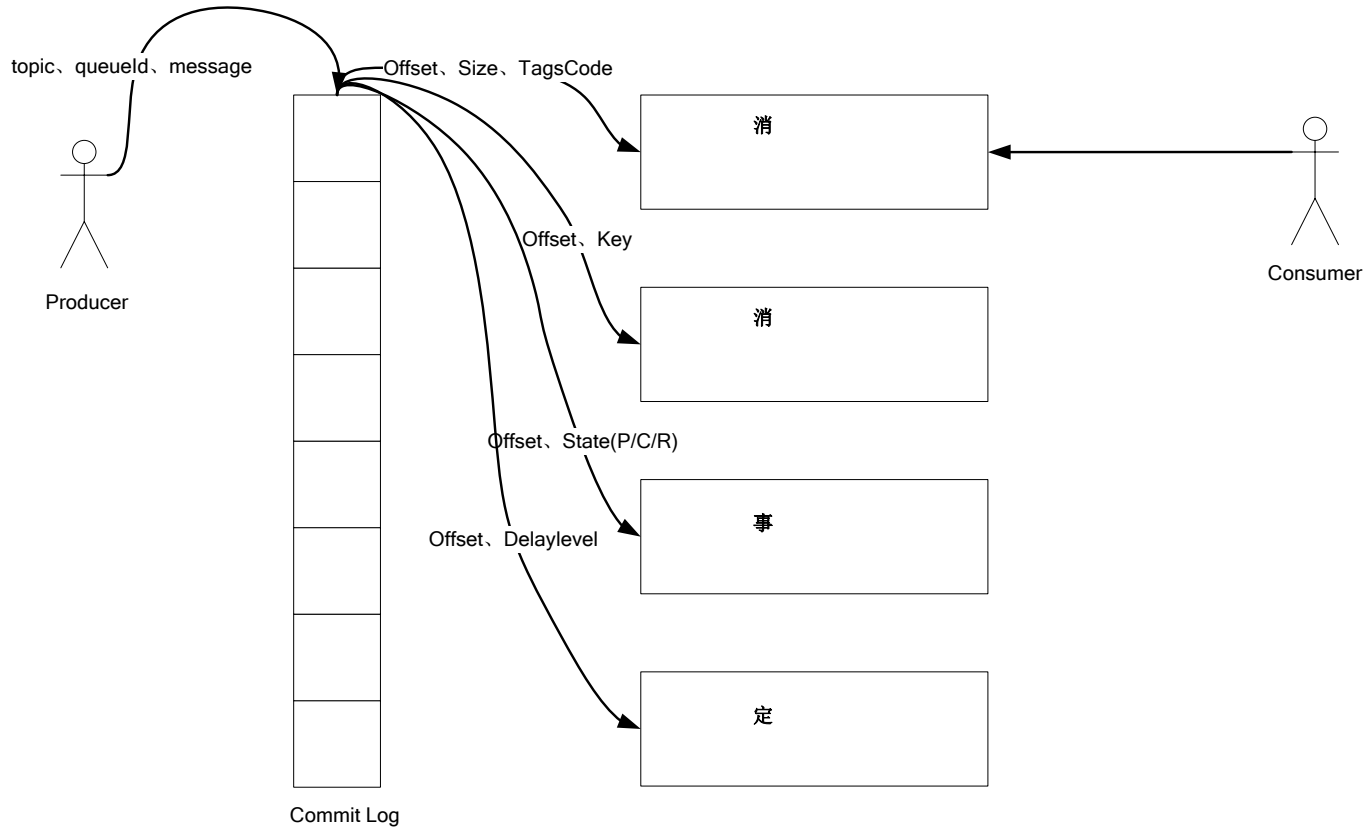
文件系统 IO 调度算法需要调整为 deadline，因为 deadline 算法在随机读情况下，可以合并读请求为顺序跳跃

方式，从而提高读 IO 吞吐量。

Ext4 文件系统有以下 Bug，请注意

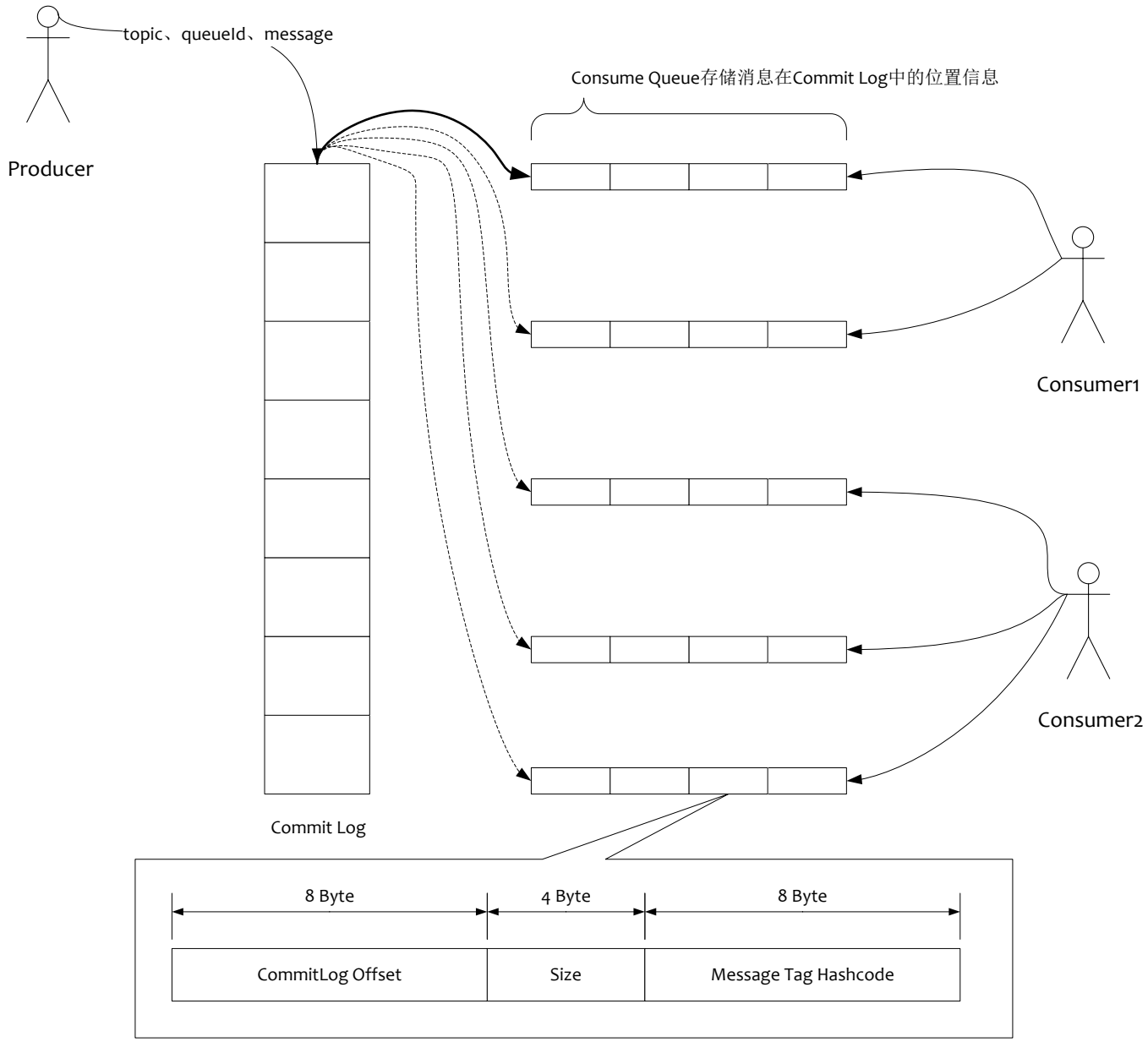
http://blog.donghao.org/2013/03/20/%E4%BF%AE%E5%A4%8Dext4%E6%97%A5%E5%BF%97%EF%BC%88jbd2%EF%BC%89bug/

6.3 数据存储结构



7 RocketMQ 关键特性

7.1 单机支持 1 万以上持久化队列



图表 7-1RocketMQ 队列

(1). 所有数据单独存储到一个 Commit Log，完全顺序写，随机读。

(2). 对最终用户展现的队列实际只存储消息在 Commit Log 的位置信息，并且串行方式刷盘。

这样做的好处如下：

(1). 队列轻量化，单个队列数据量非常少。

(2). 对磁盘的访问串行化，避免磁盘竞争，不会因为队列增加导致 IOWAIT 增高。

每个方案都有缺点，它的缺点如下：

(1). 写虽然完全是顺序写，但是读却变成了完全的随机读。

(2). 读一条消息，会先读 Consume Queue，再读 Commit Log，增加了开销。

(3). 要保证 Commit Log 与 Consume Queue 完全的一致，增加了编程的复杂度。

以上缺点如何克服：

(1). 随机读，尽可能让读命中 PAGECACHE，减少 IO 读操作，所以内存越大越好。如果系统中堆积的消息过多，读数据要访问磁盘会不会由于随机读导致系统性能急剧下降，答案是否定的。

a) 访问 PAGECACHE 时，即使只访问 1k 的消息，系统也会提前预读出更多数据，在下次读时，就可能命中内存。

b) 随机访问 Commit Log 磁盘数据，系统 IO 调度算法设置为 NOOP 方式，会在一定程度上将完全的随机读变成顺序跳跃方式，而顺序跳跃方式读较完全的随机读性能会高 5 倍以上，可参见以下针对各种 IO 方式的性能数据。

另外 4k 的消息在完全随机访问情况下，仍然可以达到 8K 次每秒以上的读性能。

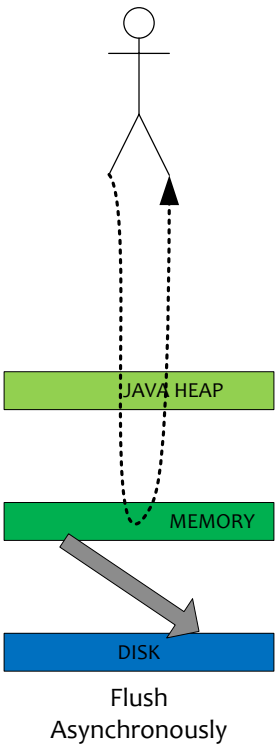
(2). 由于 Consume Queue 存储数据量极少，而且是顺序读，在 PAGECACHE 预读作用下，Consume Queue 的读性能几乎与内存一致，即使堆积情况下。所以可认为 Consume Queue 完全不会阻碍读性能。

(3). Commit Log 中存储了所有的元信息，包含消息体，类似于 Mysql、Oracle 的 redolog，所以只要有 Commit Log 在，Consume Queue 即使数据丢失，仍然可以恢复出来。

7.2 刷盘策略

RocketMQ 的所有消息都是持久化的，先写入系统 PAGECACHE，然后刷盘，可以保证内存与磁盘都有一份数据，访问时，直接从内存读取。

7.2.1 异步刷盘



在有 RAID 卡，SAS 15000 转磁盘测试顺序写文件，速度可以达到 300M 每秒左右，而线上的网卡一般都为千兆网卡，写磁盘速度明显快于数据网络入口速度，那么是否可以做到写完内存就向用户返回，由后台线程刷盘呢？

(1). 由于磁盘速度大于网卡速度，那么刷盘的进度肯定可以跟上消息的写入速度。

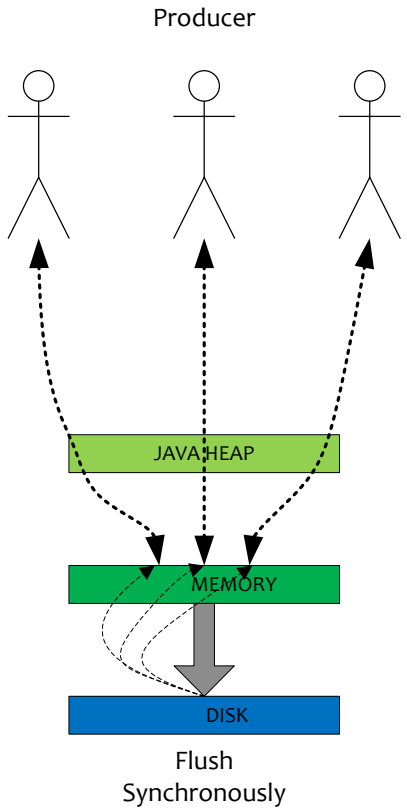
(2). 万一由于此时系统压力过大，可能堆积消息，除了写入 IO，还有读取 IO，万一出现磁盘读取落后情况，会不会导致系统内存溢出，答案是否定的，原因如下：

a) 写入消息到 PAGECACHE 时，如果内存不足，则尝试丢弃干净的 PAGE，腾出内存供新消息使用，策略是 LRU 方式。

b) 如果干净页不足，此时写入 PAGECACHE 会被阻塞，系统尝试刷盘部分数据，大约每次尝试 32 个 PAGE，来找出更多干净 PAGE。

综上，内存溢出的情况不会出现。

7.2.2 同步刷盘



同步刷盘与异步刷盘的唯一区别是异步刷盘写完 PAGECACHE 直接返回，而同步刷盘需要等待刷盘完成才返回，同步刷盘流程如下：

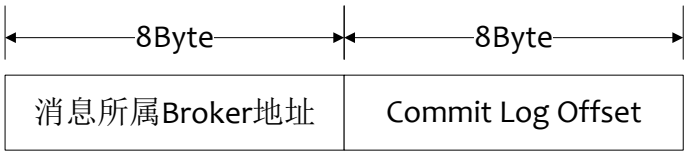
(1). 写入 PAGECACHE 后，线程等待，通知刷盘线程刷盘。

(2). 刷盘线程刷盘后，唤醒前端等待线程，可能是一批线程。

(3). 前端等待线程向用户返回成功。

7.3 消息查询

7.3.1 按照 Message Id 查询消息

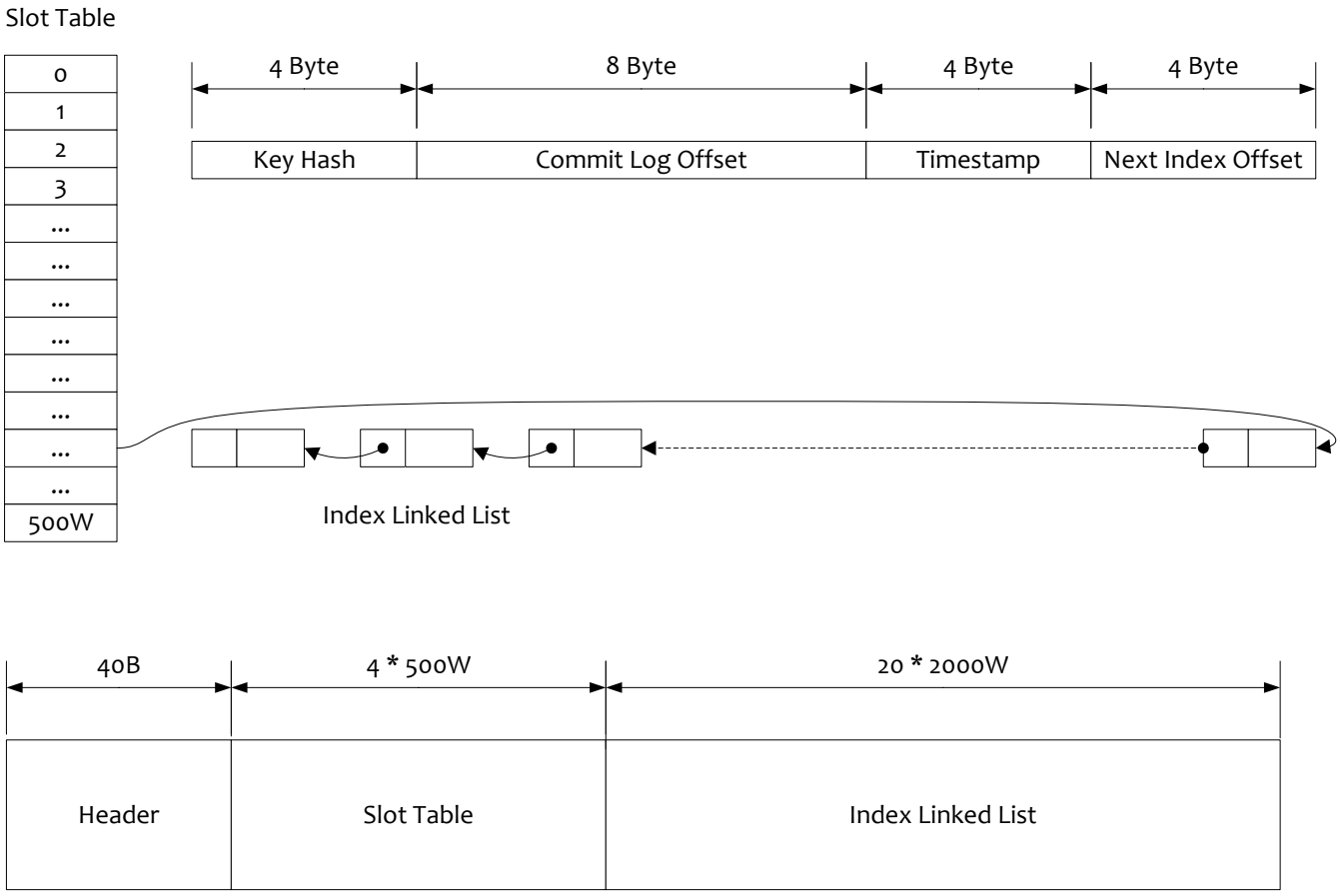


图表 7-2 Message Id 组成

MsgId 总共 16 字节，包含消息存储主机地址，消息 Commit Log offset。从 MsgId 中解析出 Broker 的地址和

Commit Log 的偏移地址，然后按照存储格式所在位置消息 buffer 解析成一个完整的消息。

7.3.2 按照 Message Key 查询消息



图表 7-3 索引的逻辑结构，类似 HashMap 实现

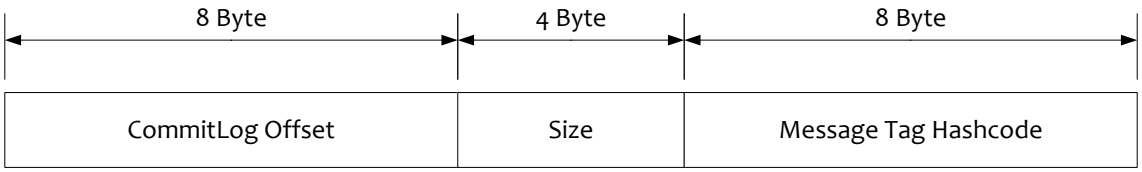
1. 根据查询的 key 的 hashcode%slotNum 得到具体的槽的位置（slotNum 是一个索引文件里面包含的最大槽的数目，例如图中所示 slotNum=5000000）。
2. 根据 slotValue（slot 位置对应的值）查找到索引项列表的最后一项（倒序排列，slotValue 总是指向最新的一个索引项）。
3. 遍历索引项列表返回查询时间范围内的结果集（默认一次最大返回的 32 条记录）
4. Hash 冲突；寻找 key 的 slot 位置时相当于执行了两次散列函数，一次 key 的 hash，一次 key 的 hash 值取模，因此这里存在两次冲突的情况；第一种，key 的 hash 值不同但模数相同，此时查询的时候会在比较一次 key 的 hash 值（每个索引项保存了 key 的 hash 值），过滤掉 hash 值不相等的项。第二种，hash 值相等但 key 不等，出于性能的考虑冲突的检测放到客户端处理（key 的原始值是存储在消息文件中的，避免对数据文件的解析），

客户端比较一次消息体的 key 是否相同。

5. 存储；为了节省空间索引项中存储的时间是时间差值（存储时间-开始时间，开始时间存储在索引文件头中），整个索引文件是定长的，结构也是固定的。索引文件存储结构参见图 7.4.3-3 。

7.4 服务器消息过滤

RocketMQ 的消息过滤方式有别于其他消息中间件，是在订阅时，再做过滤，先来看下 Consume Queue 的存储结构。



图表 7-4Consume Queue 单个存储单元结构

(1). 在 Broker 端进行 Message Tag 比对，先遍历 Consume Queue，如果存储的 Message Tag 与订阅的 Message Tag 不符合，则跳过，继续比对下一个，符合则传输给 Consumer。注意：Message Tag 是字符串形式，Consume Queue 中存储的是其对应的 hashcode，比对时也是比对 hashcode。

(2). Consumer 收到过滤后的消息后，同样也要执行在 Broker 端的操作，但是比对的是真实的 Message Tag 字符串，而不是 Hashcode。

为什么过滤要这样做？

(1). Message Tag 存储 Hashcode，是为了在 Consume Queue 定长方式存储，节约空间。

(2). 过滤过程中不会访问 Commit Log 数据，可以保证堆积情况下也能高效过滤。

(3). 即使存在 Hash 冲突，也可以在 Consumer 端进行修正，保证万无一失。

7.5 长轮询 Pull

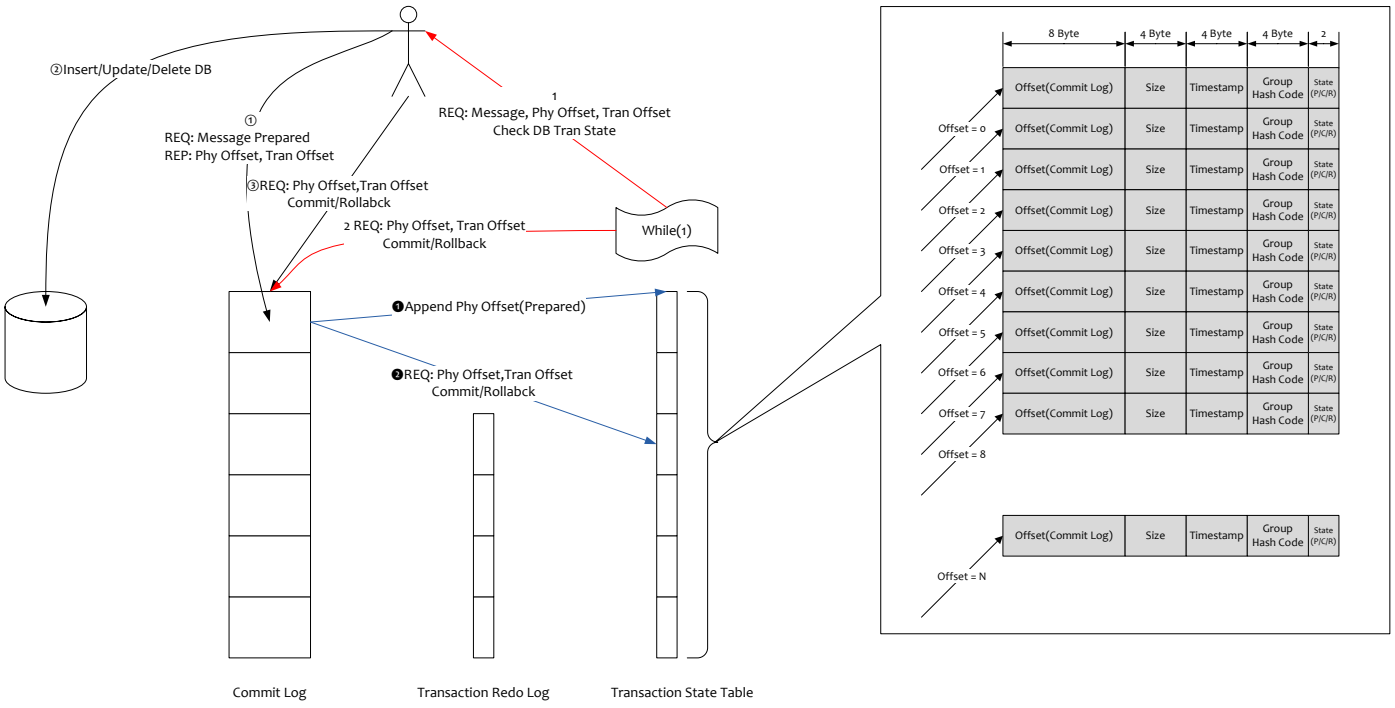
7.6 顺序消息

7.6.1 顺序消息原理

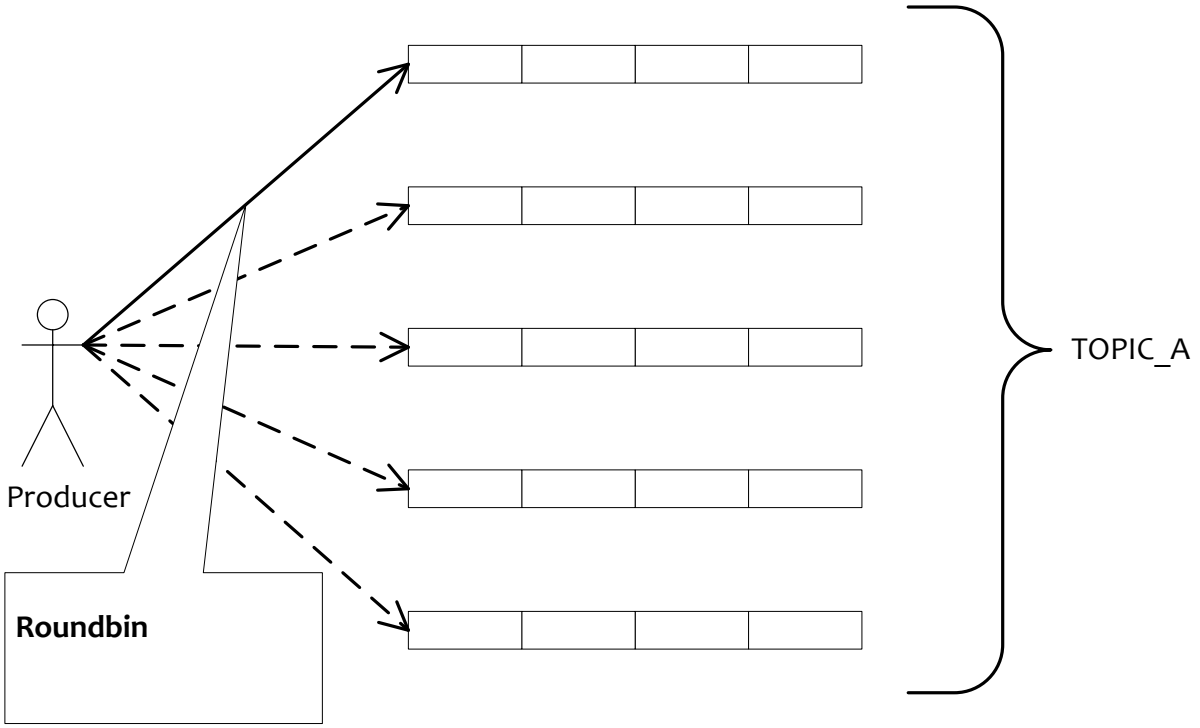
7.6.2 顺序消息缺陷

- 发送顺序消息无法利用集群 FailOver 特性
- 消费顺序消息的并行度依赖于队列数量
- 队列热点问题，个别队列由于哈希不均导致消息过多，消费速度跟不上，产生消息堆积问题
- 遇到消息失败的消息，无法跳过，当前队列消费暂停

7.7 事务消息



7.8 发送消息负载均衡

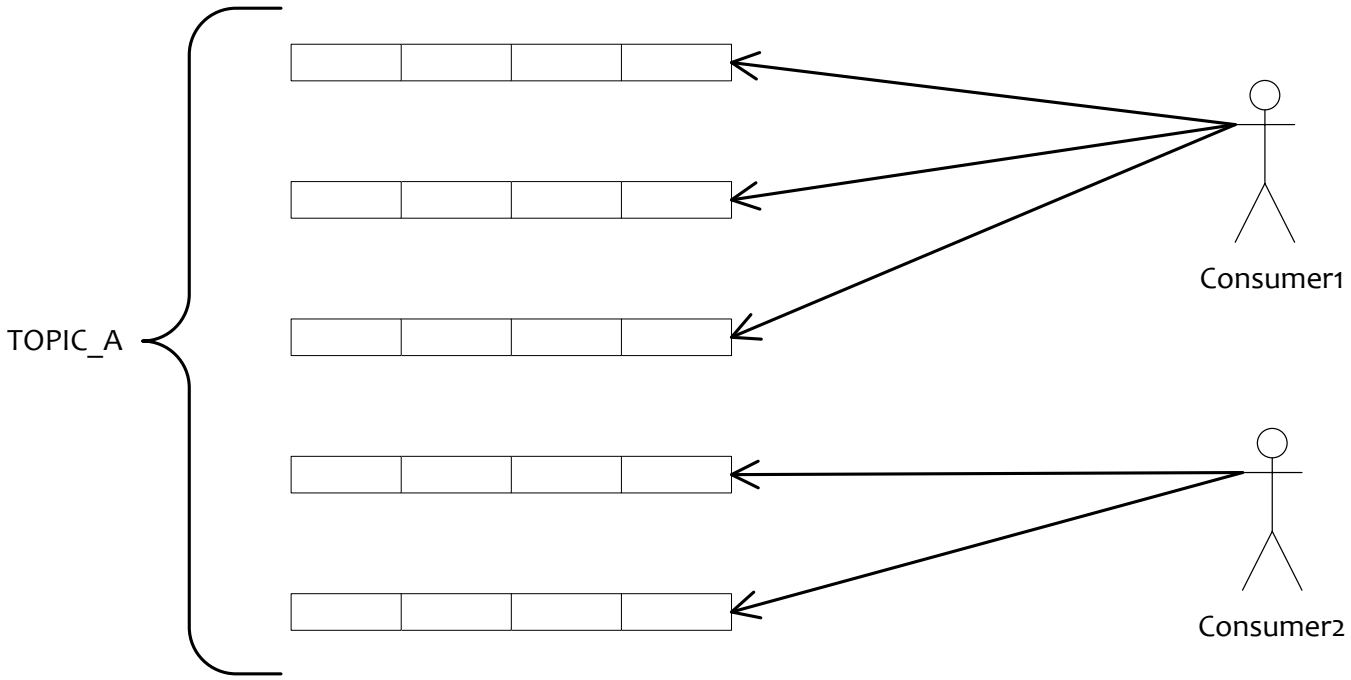


7-5 发送消息 Rebalance

如图所示，5 个队列可以部署在一台机器上，也可以分别部署在 5 台不同的机器上，发送消息通过轮询队列的方式发送，每个队列接收平均的消息量。通过增加机器，可以水平扩展队列容量。

另外也可以自定义方式选择发往哪个队列。

7.9 订阅消息负载均衡

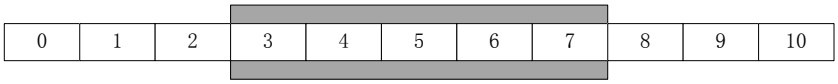


7-6 订阅消息 Rebalance

如图所示，如果有 5 个队列，2 个 consumer，那么第一个 Consumer 消费 3 个队列，第二 consumer 消费 2 个队列。这样即可达到平均消费的目的，可以水平扩展 Consumer 来提高消费能力。但是 Consumer 数量要小于等于队列数量，如果 Consumer 超过队列数量，那么多余的 Consumer 将不能消费消息。

队列数量	Consumer 数量	Rebalance 结果
5	2	C1: 3 C2: 2
6	3	C1: 2 C2: 2 C3: 2
10	20	C1~C10: 1 C11~C20: 0
20	6	C1: 4 C2: 4 c3~C6: 3

7.10 单队列并行消费



单队列并行消费采用滑动窗口方式并行消费，如图所示，3~7 的消息在一个滑动窗口区间，可以有多个线程并行消费，但是每次提交的 Offset 都是最小 Offset，例如 3

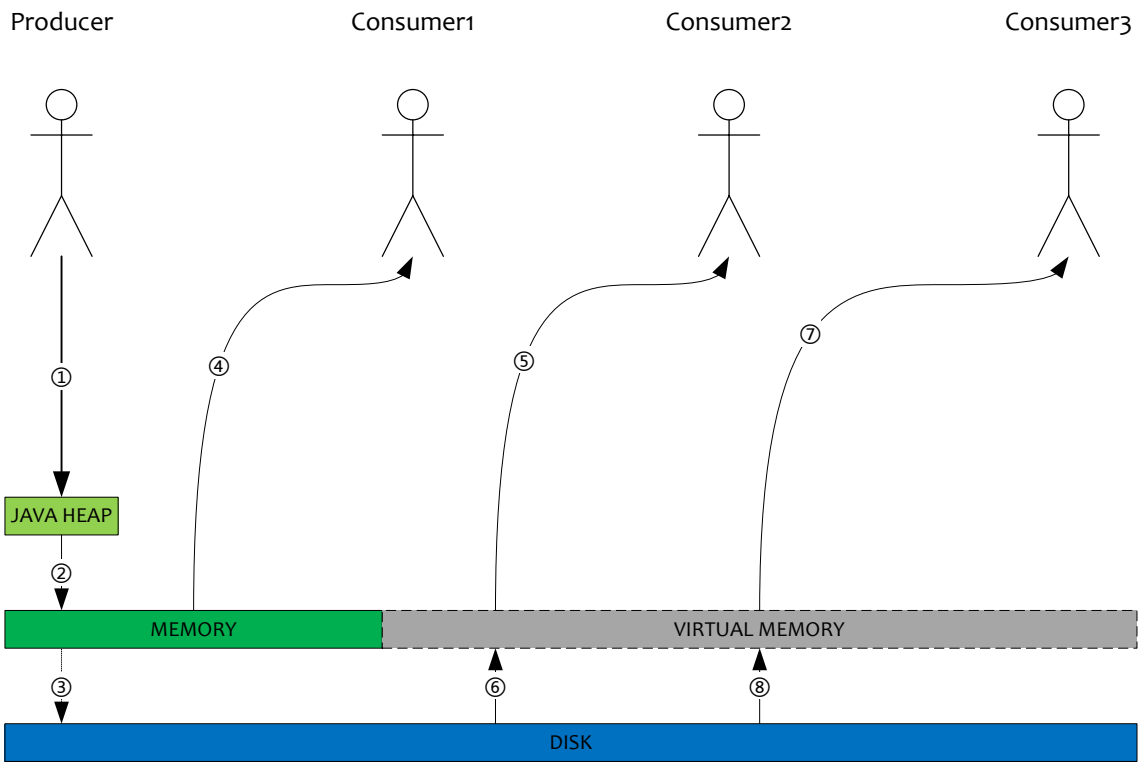
7.11 发送定时消息

7.12 消息消费失败，定时重试

7.13 HA，同步双写/异步复制

异步复制的实现思路非常简单，Slave 启动一个线程，不断从 Master 拉取 Commit Log 中的数据，然后在异步 build 出 Consume Queue 数据结构。整个实现过程基本同 Mysql 主从同步类似。

7.14 单个 JVM 进程也能利用机器超大内存



图表 7-7 消息在系统中流转图

(1). Producer 发送消息，消息从 socket 进入 java 堆。

(2). Producer 发送消息，消息从 java 堆转入 PAGACACHE，物理内存。

(3). Producer 发送消息，由异步线程刷盘，消息从 PAGECACHE 刷入磁盘。

(4). Consumer 拉消息（正常消费），消息直接从 PAGECACHE（数据在物理内存）转入 socket，到达 consumer，不经过 java 堆。这种消费场景最多，线上 96G 物理内存，按照 1K 消息算，可以在物理内存缓存 1 亿条消息。

(5). Consumer 拉消息（异常消费），消息直接从 PAGECACHE（数据在虚拟内存）转入 socket。

(6). Consumer 拉消息（异常消费），由于 Socket 访问了虚拟内存，产生缺页中断，此时会产生磁盘 IO，从磁盘 Load 消息到 PAGECACHE，然后直接从 socket 发出去。

(7). 同 5 一致。

(8). 同 6 一致。

7.15 消息堆积问题解决办法

前面提到衡量消息中间件堆积能力的几个指标，现将 RocketMQ 的堆积能力整理如下

表格 7-1RocketMQ 性能堆积指标

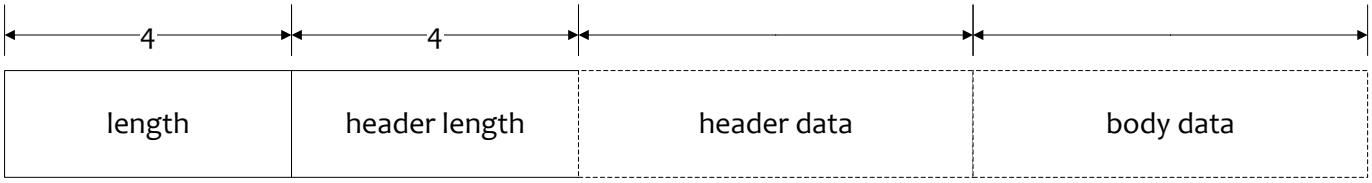
		堆积性能指标
1	**消息的堆积容量**	依赖磁盘大小
2	**发消息的吞吐量大小受影响程度**	无 SLAVE 情况，会受一定影响 有 SLAVE 情况，不受影响
3	**正常消费的 Consumer 是否会受影响**	无 SLAVE 情况，会受一定影响 有 SLAVE 情况，不受影响
4	**访问堆积在磁盘的消息时，吞吐量有多大**	1、与访问的并发有关，最慢会降到 5000 左右。

在有 Slave 情况下，Master 一旦发现 Consumer 访问堆积在磁盘的数据时，会向 Consumer 下达一个重定向指令，令 Consumer 从 Slave 拉取数据，这样正常的发消息与正常消费的 Consumer 都不会因为消息堆积受影响，因为系统将堆积场景与非堆积场景分割在了两个不同的节点处理。这里会产生另一个问题，Slave 会不会写性能下降，答案是否定的。因为 Slave 的消息写入只追求吞吐量，不追求实时性，只要整体的吞吐量高就可以，而 Slave 每次都是从 Master 拉取一批数据，如 1M，这种批量顺序写入方式即使堆积情况，整体吞吐量影响相对较小，只是写入 RT 会变长。

8 RocketMQ 通信组件

RocketMQ 通信组件使用了 Netty-4.0.9.Final，在之上做了简单的协议封装。

8.1 网络协议



1. 大端 4 个字节整数，等于 2、3、4 长度总和
2. 大端 4 个字节整数，等于 3 的长度
3. 使用 json 序列化数据
4. 应用自定义二进制序列化数据

Header 格式

```
{
   "code": 0,
   "language": "JAVA",
   "version": 0,
   "opaque": 0,
   "flag": 1,
   "remark": "hello, I am respponse /127.0.0.1:27603",
   "extFields": {
      "count": "0",
      "messageTitle": "HelloMessageTitle"
   }
}
```

Header 字段名	类型	Request	Response
code	整数	请求操作代码，请求接收方根据不同的代码做不同的操作	应答结果代码，0 表示成功，非 0 表示各种错误代码
language	字符串	请求发起方实现语言，默认 JAVA	应答接收方实现语言
version	整数	请求发起方程序版本	应答接收方程序版本

opaque	整数	请求发起方在同一连接上不同的请求标识代码，多线程连接复用使用	应答方不做修改 ,直接返回
flag	整数	通信层的标志位	通信层的标志位
remark	字符串	传输自定义文本信息	错误详细描述信息
extFields	HashMap<String,String>	请求自定义字段	应答自定义字段

8.2 心跳处理

通信组件本身不处理心跳，由上层进行心跳处理。

8.3 连接复用

同一个网络连接，客户端多个线程可以同时发送请求，应答响应通过 header 中的 opaque 字段来标识。

8.4 超时连接

如果某个连接超过特定时间没有活动（无读写事件），则自动关闭此连接，并通知上层业务，清除连接对应的注册信息。

9 RocketMQ 服务发现（Name Server）

Name Server 是专为 RocketMQ 设计的轻量级名称服务，代码小于 1000 行，具有简单、可集群横向扩展、无状态等特点。将要支持的主备自动切换功能会强依赖 Name Server。