Metaq 原理与应用

（针对 2.X 版本）

誓嘉 兰生

2013/1/13

目录

1 前言

本文档旨在描述 Metaq 的多个关键特性的实现原理，并对 Messaging System 遇到的各种问题进行总结，阐述 Metaq 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，对于消息系统又至关重要。Metaq 并不遵循任何规范，但是参考了各种规范与其他产品的优点。

2 特别说明

(1). 关于产品名称

Metamorphosis，取自 Kafka 短篇代表作《变形记》，是卡氏艺术上的最高成就，被认为是 20 世纪最伟大的小说作品之一。创作于 1912 年，发表于 1915 年。

为方便读写，又因为她是一个完全的队列模型消息中间件，所以从 2.0 版本开始，产品名称由“Metamorphosis”改为“Metaq”。

淘宝内部都这样读：[maɪtʌkju:]

(2). Metamorphosis 1.x 版本由 Java 社区著名的庄晓丹(killme2008@gmail.com)开发，可参见以下资料

http://www.iteye.com/magazines/107
http://github.com/killme2008/Metamorphosis

(3). 本文着重介绍 Metaq 2.x 版本相关，2.x 版本现在已经开源，有任何问题可以通过开源网站联系我们。

http://metaq.taobao.org/

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。

- **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。

  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，类似于 JMS 规范中的 Point-to-Point Messaging

  特点如下：

  - Each message has only one consumer.
  - A sender and a receiver of a message have no timing dependencies. The receiver can fetch the message whether or not it was running when the client sent the message.
  - The receiver acknowledges the successful processing of a message.

- **主动消费**

  Consumer 主动向 Broker 发起获取消息请求，控制权完全在于 Consumer 应用。

  类似于 JMS 规范中描述的 Synchronously 方式消费

- **被动消费**

  Consumer 注册一个 Callback 接口，由 Metaq 后台自动从 Broker 接收消息，并回调 Callback 接口。

  类似于 JMS 规范中的描述的 Asynchronously 方式消费

- **顺序消息**

  消费消息的顺序要同发送消息的顺序一致，在 Metaq 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。

- **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。

- **Message Queue**

  在 Metaq 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。

  在 Metaq2.x 之前版本，队列也称为“**分区**”，两者描述的是一个概念。但是按照 2.x 的实现，使用队列描述更合适。

- **Messaging System**

  消息中间件的统称，但并不局限于消息中间件，与消息传输相关的类似系统。

4 消息系统需要解决哪些问题？

本节阐述消息系统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，Metaq 是否可以解决，规范中如何定义这些问题。

4.1 Publish/Subscribe

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

4.2 Message 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。

Metaq 可以严格的保证消息有序。

4.3 消息过滤

- 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 Metaq 只支持按照简单的消息类型过滤。

  (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。

4.4 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 Metaq 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 Metaq 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。

4.5 Exactly And Only Once

消息是否重复性问题，JMS 中有如下说明：

The JMS API can ensure that a message is delivered once and only once. Lower levels of reliability are available for applications that can afford to miss messages or to receive duplicate messages.

我对于此特性的理解：

(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 And 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 Metaq 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。Metaq 虽然不能严格保证不重复，但是正常情况下都不会出现重复发送、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。

4.6 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。

Metaq 支持按照时间回溯消费，时间维度精确到毫秒。

4.7 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，Metaq

(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系统的性能。

4.8 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.

4.9 消息堆积

Messaging System 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 Messaging System 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？

4.10 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，Metaq 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。Metaq 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。未来版本会通过同步双写技术来完全避免单点，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。

4.11 分布式事务

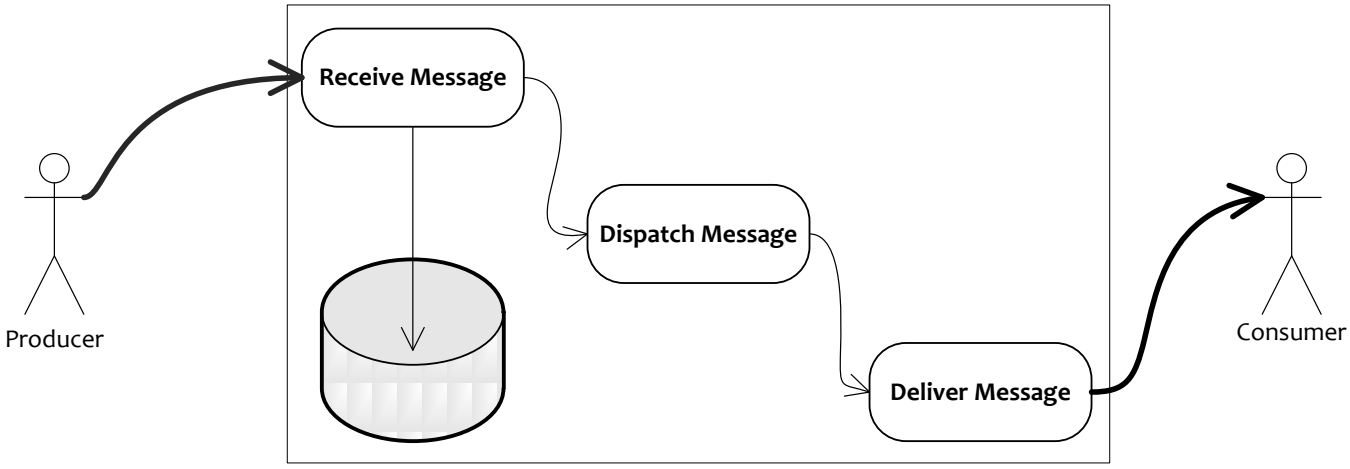
已知的几个分布式事务规范，如 XA，JTA 等。其中 XA 规范被各大数据库厂商广泛支持，如 Oracle，Mysql 等。其中 XA 的 TM 实现佼佼者如 Oracle Tuxedo，在金融、电信等领域被广泛应用。

Metaq 的存储结构是文件记录形式，通过 Offset 递增进行访问数据，缺乏 KV 存储具有的 update 能力，如果要支持事务，必须引入类似于 KV 存储的模块才可以。

Metaq 目前不支持分布式事务。

5 消息系统实现

5.1 典型实现



图表 5-1 Messaging System

一个典型的消息系统实现一般分为以下几部分：

(1). 消息接收阶段，收到消息后，存储到存储模块，可以是本地 KV 存储、文件记录存储、远程 DB、或其他存储模式，然后将消息转发至 Dispatch 处理流程。

(2). 消息分发阶段，可能的操作包含消息过滤、分发到特定队列或者 Consumer Group 队列。

(3). 消息投递阶段，直接与 Consumer 进行通信，可以是主动投递或者被动投递。

5.2 Apache Kafka 实现

Apache Kafka is a distributed publish-subscribe messaging system. It is designed to support the following.

- Persistent messaging with O(1) disk structures that provide constant time performance even with many TB of stored messages.
- High-throughput: even with very modest hardware Kafka can support hundreds of thousands of messages per second.
- Explicit support for partitioning messages over Kafka servers and distributing consumption over a cluster of consumer machines while maintaining per-partition ordering semantics.
- Support for parallel data load into Hadoop.

Kafka provides a publish-subscribe solution that can handle all activity stream data and processing on a consumer-scale web site. This kind of activity (page views, searches, and other user actions) are a key ingredient in many of the social feature on the modern web. This data is typically handled by "logging" and ad hoc log aggregation solutions due to the throughput requirements. This kind of ad hoc solution is a viable solution to providing logging data to an offline analysis system like Hadoop, but is very limiting for building real-time processing. Kafka aims to unify offline and online processing by providing a mechanism for parallel load into Hadoop as well as the ability to partition

real-time consumption over a cluster of machines.

The use for activity stream processing makes Kafka comparable to Facebook's Scribe or Apache Flume (incubating), though the architecture and primitives are very different for these systems and make Kafka more comparable to a traditional messaging system. See our design page for more details.

以上文字摘自 Apache Kafka 官方网站，Kafka 详细资料，请参照官方网站，此文不再详述。

http://kafka.apache.org/

Kafka 是一个实现非常特别的消息系统，同我们的典型实现区别较大，跳出了各种消息系统规范定义的思维模式，按照我的理解总结如下：

(1). Kafka 的设计初衷消息是面向堆积需求的，读优先级高于写优先级，优先为读消息准备，所以无论堆积的消息多少，消费能力都很强。（前提是分区数在一定范围内）

(2). Kafka 是面向广播消费类型的，对广播支持友好，并且也支持集群消费模式。

(3). 很适合日志收集类应用。

5.3 Push 和 Pull

push 与 pull 是两种数据通信方式，在 Consumer 从 Broker 获取消息过程中会用到这两种方式，区别如下：

	Push	Pull
数据传输状态	保存在 Broker 端	保存在 Consumer 端
传输失败，重试	Broker 需要维护每次传输状态，遇到失败情况需要重试	不需要
数据传输实时性	非常实时	(1). 默认的短轮询方式实时性依赖 pull 间隔时间，间隔越大实时性越低。 (2). 长轮询模式实时性与 push 一致。
流控机制	Broker 需要依据 Consumer 的消费能力做流控。	Consumer 可以根据自身消费能力决定是否去 pull message。

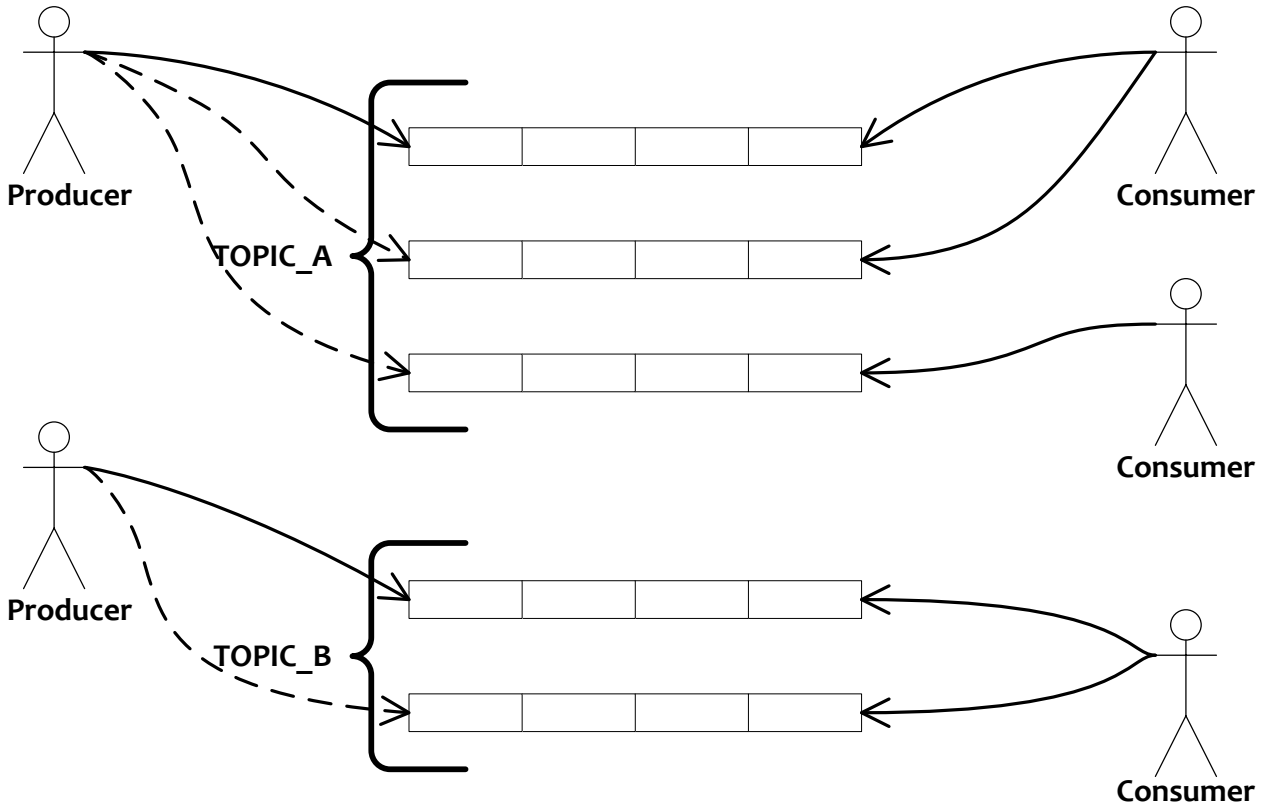
6 Metaq Overview

6.1 Metaq 与 Apache Kafka 的区别

表格 6-1Metaq 与 Kafka 区别

	Apache Kafka	Taobao Metaq
分区数	分区过多会 Load 升高	单机 1 万以上
SATA 盘下性能	性能极低	由于 PAGECACHE 作用，及刷盘方式的不同，性能接近 SAS 盘
服务端消息过滤	不支持	支持
按照时间回溯	不支持	支持
Consumer 订阅实时性	1、数据落盘后，才可被订阅 2、Consumer Pull 最大间隔时间	1、数据进入 Broker 就可以被订阅 2、同样依赖 Consumer Pull 最大间隔时间
异步刷盘实时性	按照条数与时间刷盘，实时性低	后台线程实时刷盘
同步刷盘	不支持	支持，且性能接近异步刷盘
异步复制	不支持	支持

6.2 Metaq 是什么？



图表 6-1 Metaq 是什么

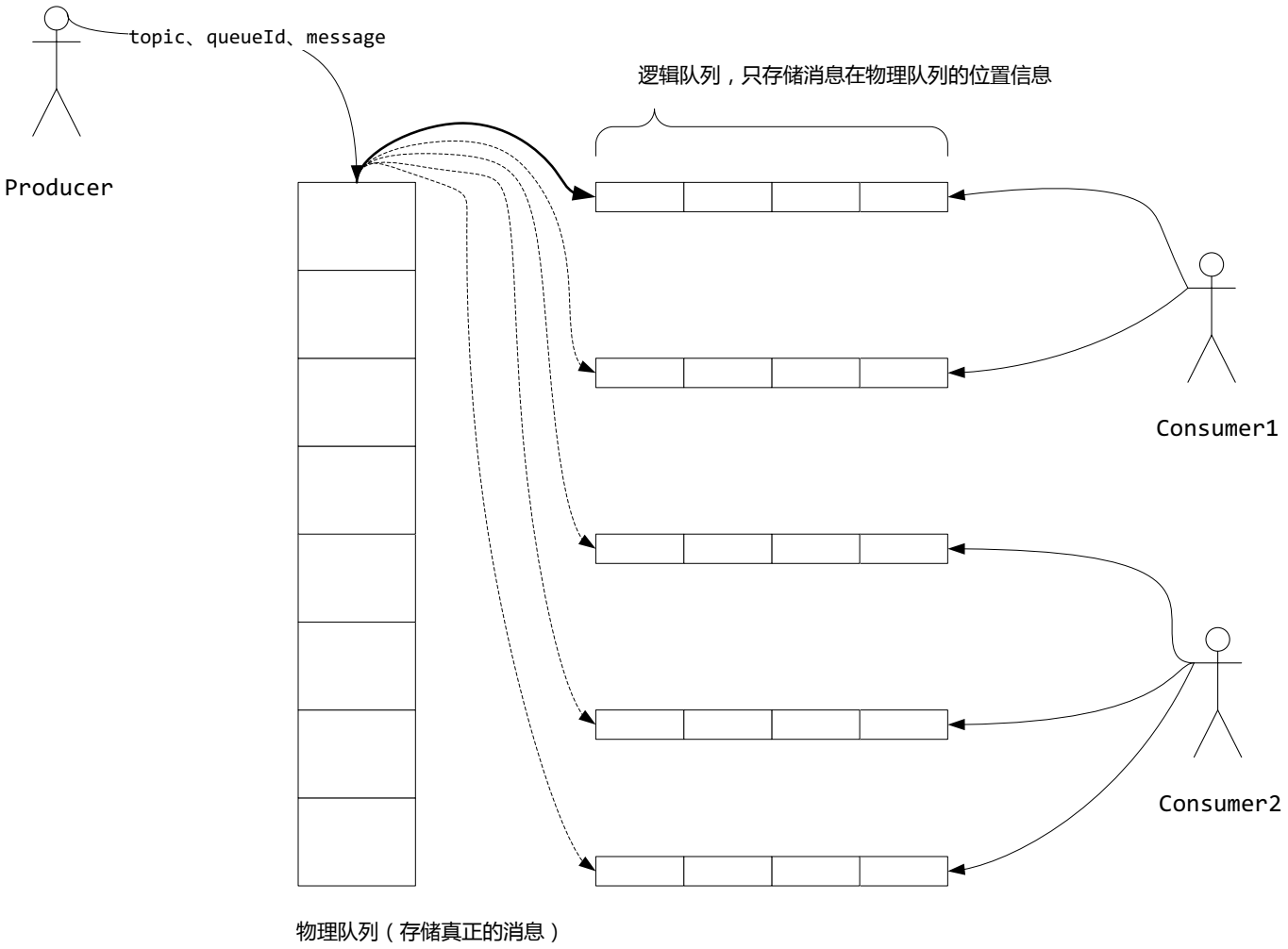
(1). 是一个队列模型的消息中间件。

(2). Producer、Consumer、队列都可以分布式。

(3). 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。

7 Metaq 主要功能点实现细节

7.1 单机如何实现 1 万以上个队列



图表 7-1Metaq 队列

(1). 所有数据单独存储到一个物理队列，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物理队列的位置信息，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，会先读逻辑队列，再读物理队列，增加了开销。

(3). 要保证物理队列与逻辑队列完全的一致，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) 随机访问物理队列磁盘数据，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。

http://stblog.baidu-tech.com/?p=851

另外 4k 的消息在完全随机访问情况下，仍然可以达到 10000 次每秒以上的读性能。

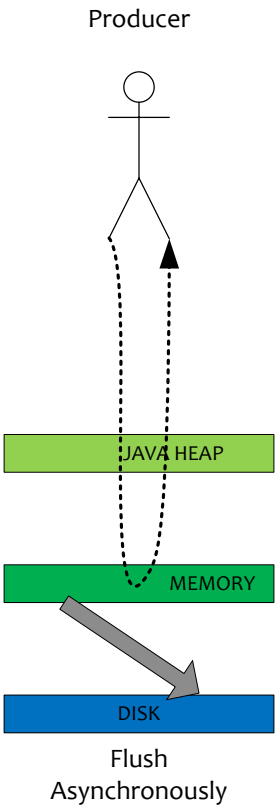
(2). 由于逻辑队列存储数据量极少，而且是顺序读，在 PAGECACHE 预读作用下，逻辑队列的读性能几乎与内存一致，即使堆积情况下。所以可认为逻辑队列完全不会阻碍读性能。

(3). 物理队列中存储了所有的元信息，类似于 Mysql 的 binlog，Oracle 的 redolog，所有只要有物理队列在，逻辑队列即使数据丢失，仍然可以恢复出来。

7.2 刷盘策略

Metaq 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 200M 每秒以上，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。

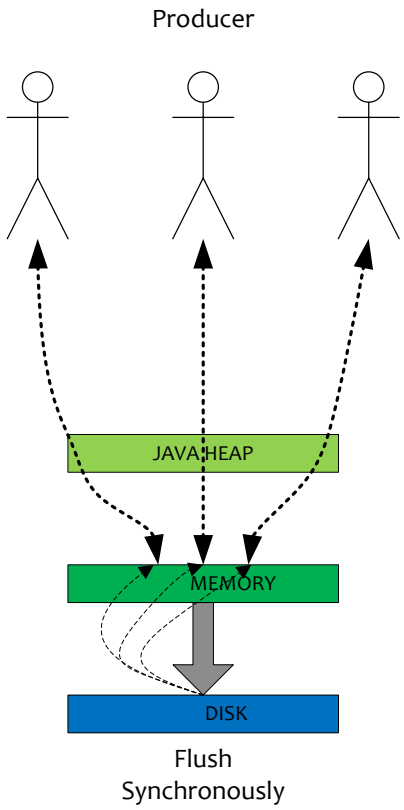
(2). 万一由于此时系统压力过大，可能堆积消息，除了写入 IO，还有读取 IO，万一出现磁盘读取落后情况，会不会导致系统内存溢出，答案是否定的，原因如下：

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。

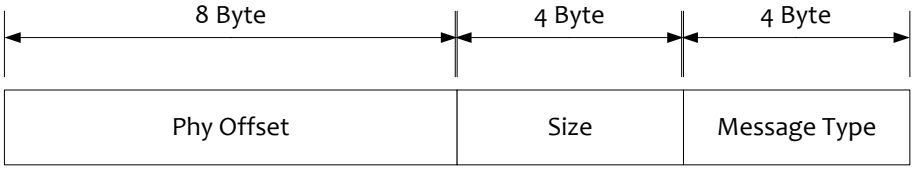
(2). 刷盘线程刷盘后，唤醒前端等待线程，可能是一批线程。

(3). 前端等待线程向用户返回成功。

7.3 服务器消息过滤

7.3.1 按照 Message Type 过滤

Metaq 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，先来看下逻辑队列的存储结构。



图表 7-2 逻辑队列单个存储单元结构

(1). 在 Broker 端进行 Message Type 比对，先遍历逻辑队列，如果存储的 Message Type 与订阅的 Message Type 不符合，则跳过，继续比对下一个，符合则传输给 Consumer。注意：Message Type 是字符串形式，逻辑队列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 Type 字符串，而不是 Hashcode。

为什么过滤要这样做？

(1). Message Type 存储 Hashcode，是为了在逻辑队列定长方式存储，节约空间。

(2). 过滤过程中不会访问物理队列数据，可以保证堆积情况下也能高效过滤。

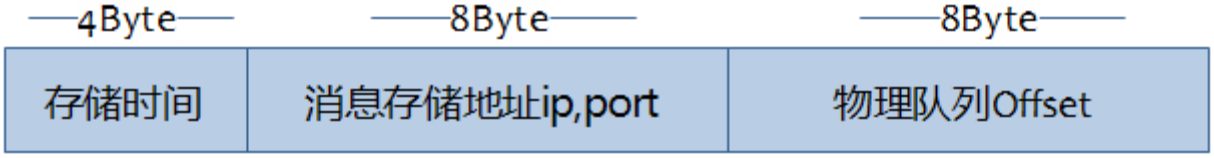
(3). 即使存在 Hash 冲突，也可以在 Consumer 端进行修正，保证万无一失。

7.3.2 按照 Message Tag 过滤

暂时不支持。

7.4 消息查询

7.4.1 按照 MessageId 查询消息

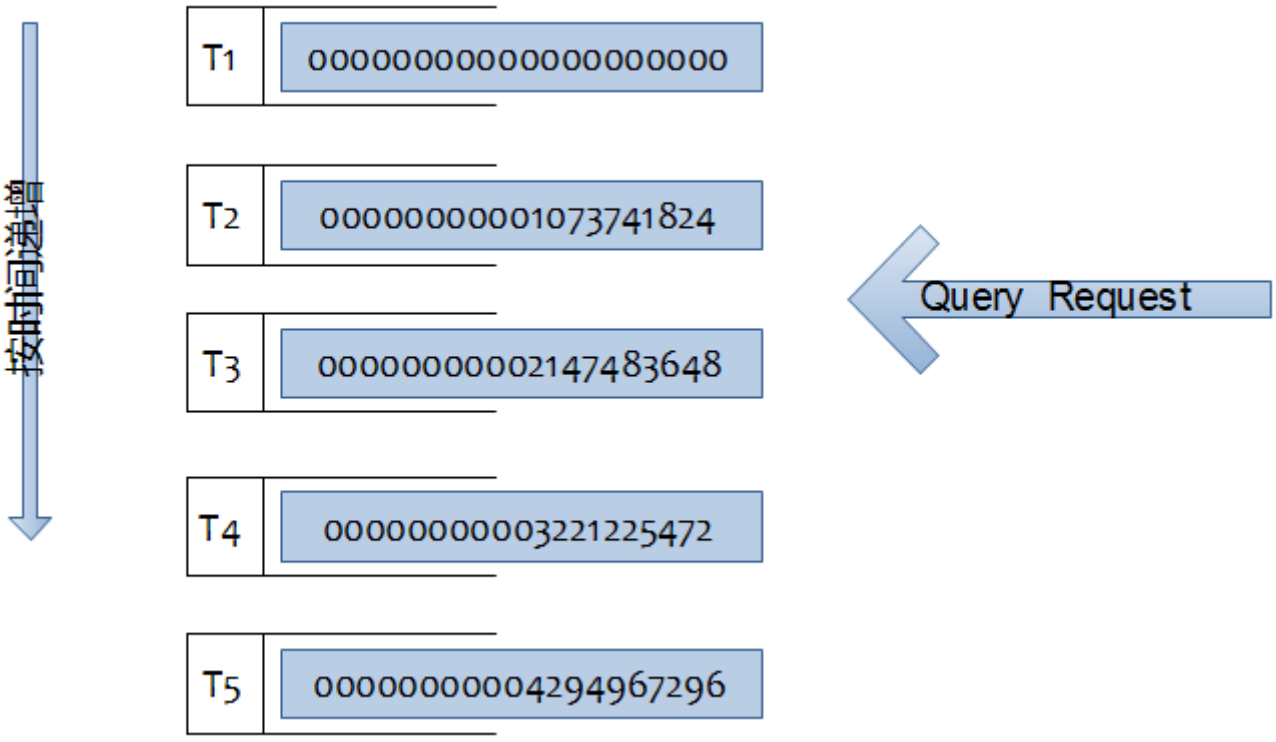


图表 7-3 Message Id 组成

MsgId 总共 20 字节，包含消息存储时间，消息存储主机地址，消息物理分区 offset。从 MsgId 中解析出 server

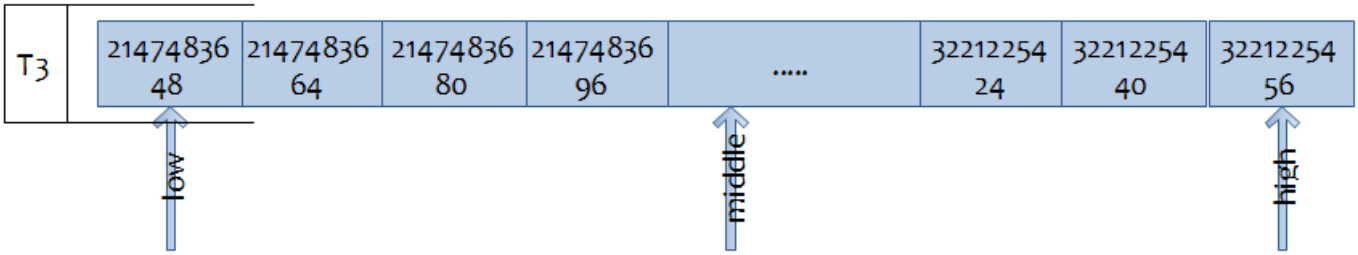
的地址和物理分区的偏移地址，然后按照存储格式所在位置消息 buffer 解析成一个完整的消息。

7.4.2 按照时间查询队列 Offset



图表 7-4 逻辑队列组成

根据时间 Tn 查找到对应的索引文件 Fn，索引文件在 metaq 的内部存储结构中是有序排列的（按时间递增），因此当一个查询进来的时候首先根据查询的时间 Tn 查找到所属的索引文件。例如：假设 T2<Tn<T3（Tn 对应的索引文件记做 Fn，Tn 是索引文件 Fn 最后一次的修改时间），则 Tn 时间对应的消息就应包含在索引文件 F3 中。

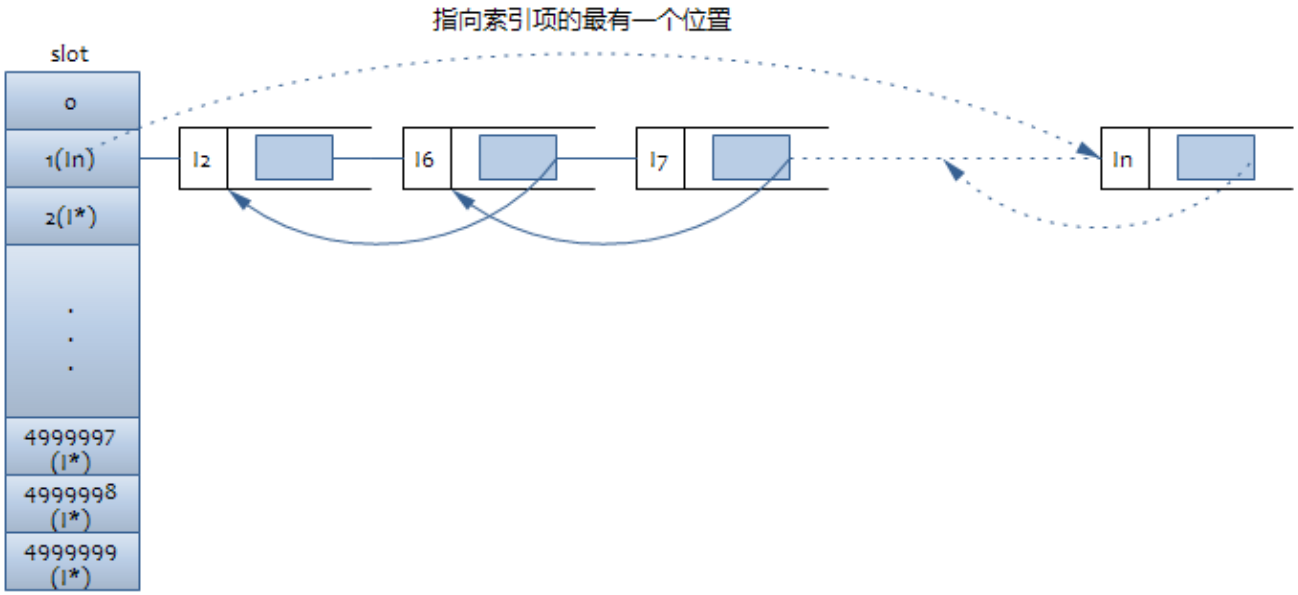


图表 7-5 按照时间二分查找

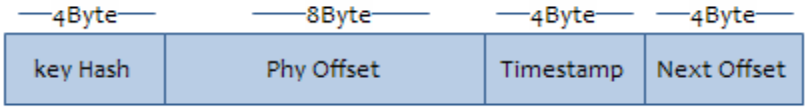
二分遍历索引文件，查找到一个最接近的消息，同一个队列的消息是严格按照时间先后顺序入队，所以按照二

分查找定位到一个消息的 offset 时间复杂度为 O(logN)。例如：一个索引文件的大小为 8M，一条索引占 16 字节，一个文件中总共的索引条数为 524288 条，极端情况下最多遍历次数为 19，因为 query 的时间精确到 ms，所以不命中的情况下会返回最接近的结果值。

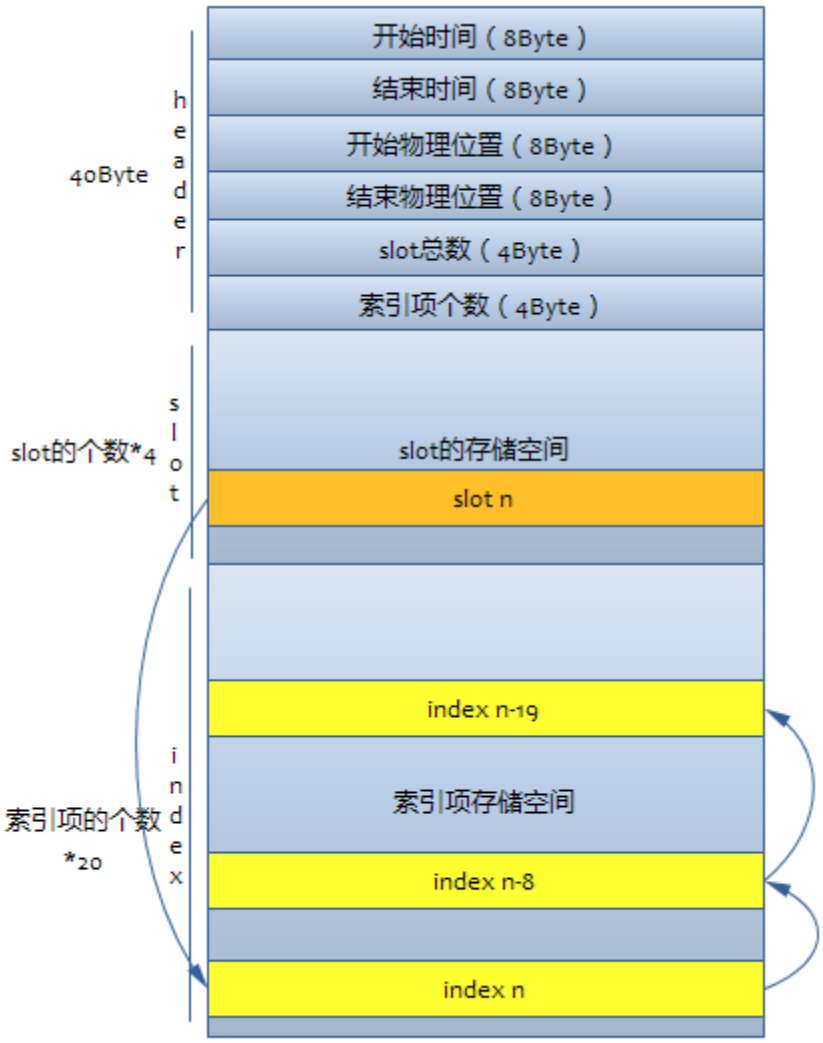
7.4.3 按照 Message Key 查询消息



图表 7-6 索引的逻辑结构，类似 HashMap 实现



图表 7-7 单个索引项的存储结构



图表 7-8 每个索引文件的物理存储结构

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 7.4.3-1 中所示 slotNum=5000000)，索引的逻辑结构见图 7.4.3-1 。
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），索引项的存储结构见图 7.4.3-2 。
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 。

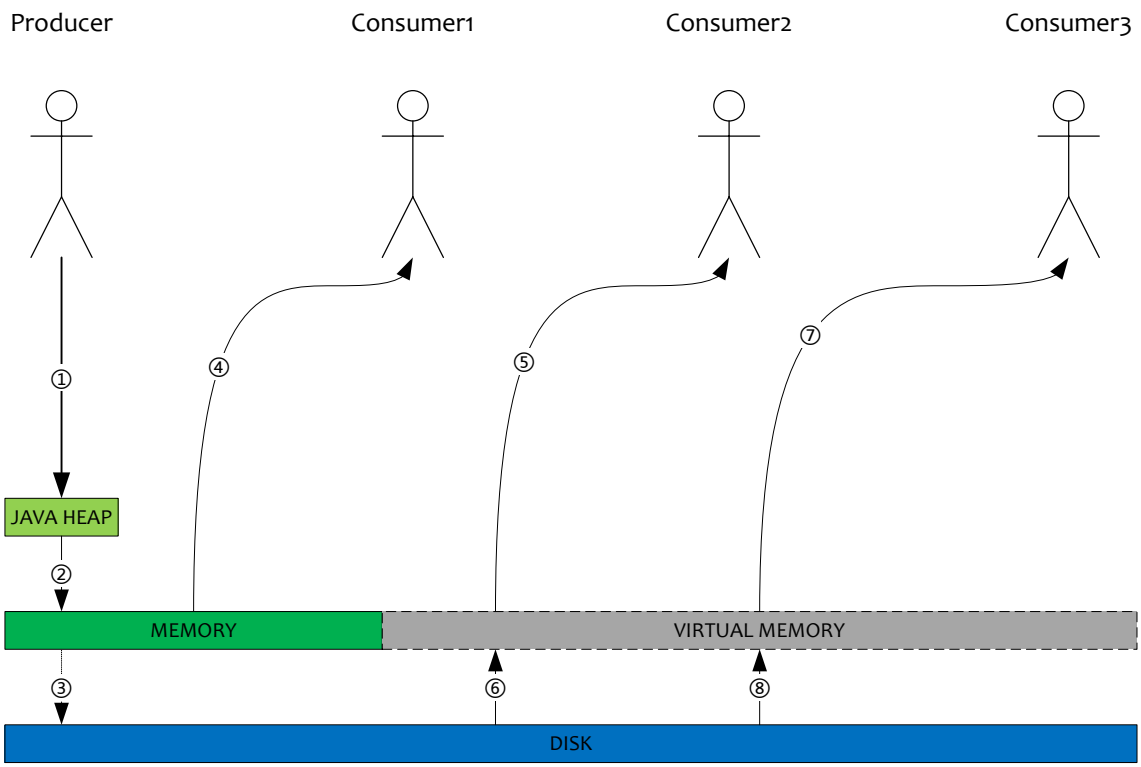
7.5 HA，同步双写

暂时不支持。

7.6 HA，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 拉取物理队列中的数据，然后在异步 build 出逻辑队列数据结构。整个实现过程基本同 Mysql 主从同步类似。

7.7 单个 JVM 进程也能利用机器超大内存



图表 7-9 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。

7.8 消息堆积问题解决办法

前面提到衡量消息系统堆积能力的几个指标，现将 Metaq 的堆积能力整理如下

表格 7-1Metaq 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、1K 大小左右消息堆积情况下吞吐量非常高， 在 5W 每秒以上。 2、4K 消息性能最差，1W 每秒左右。

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 Metaq 开发过程中遇到了哪些问题？

(1). 分配 MapedFile 有时耗时很长，长达几百毫秒，在性能压测过程中，会大大拉低整体性能。

解决办法：采用预分配方式，这样前端请求就不会因为分配 MapedFile 而阻塞。

(2). 删除单个 1G 文件耗时达 1 秒以上，导致写消息性能下降

解决办法：在 ext3 下，删除文件确实耗时较长，使用 ext4 文件系统，删除 1G 文件通常耗时几十毫秒。

(3). Consumer 抢分区会存在失败的情况，原因是写入 ZK 节点失败

解决办法：去除抢这个概念，所有 Consumer 按照同样的算法进行分配，最终一致。

9 Metaq 目前有哪些地方需要改进？

(1). 异步复制仍然不能完全避免单点，会被用户诟病，需要尝试同步双写。

(2). 消息过滤功能较简单，是否可以支持像 Notify 一样的高级过滤方式

(3). 对 Zookeeper 强依赖问题。

(4). 客户端 API 其实可以更友好。

10 Metaq 的几个设计原则

Metaq 在开发过程中，遵循以下设计原则，有些观点为个人看法，不一定对所有产品都适用，或者不一定正确，写出来仅供各位参考。

1. 消息都是持久化的，内存与磁盘各一份。

2. 高性能离不开异步，异步离不开队列，无论是构建整个系统还是构建单个程序都适用
3. 磁盘 IO，网络 IO 尽可能的批量处理。
4. 如果操作系统的层面已经提供的功能，或者优化过的性能点，尽可能交给操作系统处理.
5. 与 IO 相关部分尽可能用单线程处理，避免使用多线程。一个功能尽可能拆分成几个独立的子功能，每个功能有专门的线程来执行。
6. 一个线程只做一件事情，并做好。因为目前的计算机 CPU 资源足够强大，即使是单个 CPU 也运算能力足够快。如果一个线程没有阻塞的地方，使用多线程并不能提升性能，反而会下降，如果有阻塞地方，可酌情使用多线程，但也要分场景。
7. 为追求高性能，牺牲了某些消息系统中特有的高级功能，例如消息优先级、分布式事务等。

附录 A 参考文档、规范

- Java Message Service API Tutorial
  http://docs.oracle.com/javaee/1.3/jms/tutorial/1_3_1-fcs/doc/jms_tutorialTOC.html
- Java(TM) Message Service Specification Final Release 1.1
  http://www.oracle.com/technetwork/java/docs-136352.html
- CORBA Notification Service Specification 1.1
  http://www.omg.org/spec/NOT/1.1/PDF
- Distributed Transaction Processing: The XA Specification
  http://pubs.opengroup.org/onlinepubs/009680699/toc.pdf
- Metaq Benchmark
  http://taobao.github.com/metaq/document/benchmark/benchmark.pdf
- Documentation for /proc/sys/vm/*
  http://www.kernel.org/doc/Documentation/sysctl/vm.txt