|  |
| --- |
|  |
| **Alibaba RocketMQ原理简介** |
|  |
|  |
| **誓嘉(王小瑞) vintage.wang@gmail.com** |
| 2013/8/23 |

**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 | 2013/5/18 |
| 2 | 3.0版本补充文档 | 誓嘉vintage.wang@gmail.com | 2013/8/16 |
| 3 |  |  |  |
| 4 |  |  |  |
| 5 |  |  |  |
| 6 |  |  |  |
| 7 |  |  |  |

**目录**

[1 前言 1](#_Toc365039928)

[2 专业术语 1](#_Toc365039929)

[3 消息中间件需要解决哪些问题？ 3](#_Toc365039930)

[3.1 Publish/Subscribe 3](#_Toc365039931)

[3.2 Message Priority 3](#_Toc365039932)

[3.3 Message Order 3](#_Toc365039933)

[3.4 Message Filter 4](#_Toc365039934)

[3.5 Message Persistence 4](#_Toc365039935)

[3.6 Message Reliablity 5](#_Toc365039936)

[3.7 Low Latency Messaging 5](#_Toc365039937)

[3.8 At least Once 5](#_Toc365039938)

[3.9 Exactly Only Once 5](#_Toc365039939)

[3.10 Broker的Buffer满了怎么办？ 6](#_Toc365039940)

[3.11 回溯消费 6](#_Toc365039941)

[3.12 消息堆积 7](#_Toc365039942)

[3.13 分布式事务 7](#_Toc365039943)

[3.14 定时消息 8](#_Toc365039944)

[3.15 消息重试 8](#_Toc365039945)

[4 RocketMQ Overview 8](#_Toc365039946)

[4.1 RocketMQ是什么？ 8](#_Toc365039947)

[4.2 RocketMQ部署结构 9](#_Toc365039948)

[5 RocketMQ存储特点 10](#_Toc365039949)

[5.1 与DB存储、KV存储比较 10](#_Toc365039950)

[5.2 零拷贝原理 11](#_Toc365039951)

[5.3 文件系统 11](#_Toc365039952)

[6 RocketMQ关键特性 11](#_Toc365039953)

[6.1 单机支持1万以上持久化队列 11](#_Toc365039954)

[6.2 刷盘策略 12](#_Toc365039955)

[6.2.1 异步刷盘 13](#_Toc365039956)

[6.2.2 同步刷盘 14](#_Toc365039957)

[6.3 消息查询 14](#_Toc365039958)

[6.3.1 按照Message Id查询消息 14](#_Toc365039959)

[6.3.2 按照Message Key查询消息 15](#_Toc365039960)

[6.4 服务器消息过滤 16](#_Toc365039961)

[6.5 长轮询Pull 17](#_Toc365039962)

[6.6 顺序消息 17](#_Toc365039963)

[6.7 事务消息 17](#_Toc365039964)

[6.8 发送消息负载均衡 17](#_Toc365039965)

[6.9 订阅消息负载均衡 17](#_Toc365039966)

[6.10 单队列并行消费 17](#_Toc365039967)

[6.11 发送定时消息 17](#_Toc365039968)

[6.12 消息消费失败，定时重试 17](#_Toc365039969)

[6.13 HA，同步双写/异步复制 17](#_Toc365039970)

[6.14 单个JVM进程也能利用机器超大内存 18](#_Toc365039971)

[6.15 消息堆积问题解决办法 19](#_Toc365039972)

[7 RocketMQ通信协议 20](#_Toc365039973)

[7.1 网络协议 20](#_Toc365039974)

[7.2 心跳处理 21](#_Toc365039975)

[7.3 连接复用 21](#_Toc365039976)

[7.4 按需连接 21](#_Toc365039977)

[8 RocketMQ服务路由（Name Server） 21](#_Toc365039978)

[附录A 参考文档、规范 21](#_Toc365039979)

# 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。

# 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Handler接口，一旦收到消息，Consumer对象立 刻回调Handler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。

* **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。

# 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。

## 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。

## 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？

## 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。

## 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。

## 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。

## 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。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。

## 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。

## 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。

## 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。

## 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。

1. 按照特定策略丢弃已有消息
   * + 1. **AnyOrder** - Any event may be discarded on overflow. This is the default setting for this

property.

* + - 1. **FifoOrder** - The first event received will be the first discarded.
      2. **LifoOrder** - The last event received will be the first discarded.
      3. **PriorityOrder** - Events should be discarded in priority order, such that lower priority

events will be discarded before higher priority events.

* + - 1. **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。

## 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。

## 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？

## 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。

## 定时消息

## 消息重试

# RocketMQ Overview

## RocketMQ是什么？



图表 ‑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。

## RocketMQ部署结构



图表 ‑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。

# RocketMQ存储特点

## 与DB存储、KV存储比较

## 零拷贝原理

## 文件系统

# RocketMQ关键特性

## 单机支持1万以上持久化队列



图表 ‑1RocketMQ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，避免磁盘竟争，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。
   1. 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。
   2. 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8K次每秒以上的读性能。

1. 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。
2. 物理队列中存储了所有的元信息，包含消息体，类似于Mysql、Oracle的redolog，所以只要有物理队列在，逻辑队列即使数据丢失，仍然可以恢复出来。

## 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。

### 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：
   1. 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。
   2. 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。

### 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。

## 消息查询

### 按照Message Id查询消息



图表 ‑3 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。

### 按照Message Key查询消息



图表 ‑6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 。

## 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，先来看下逻辑队列的存储结构。



图表 ‑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。

## 长轮询Pull

## 顺序消息

## 事务消息

## 发送消息负载均衡

## 订阅消息负载均衡

## 单队列并行消费

## 发送定时消息

## 消息消费失败，定时重试

## 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。

## 单个JVM进程也能利用机器超大内存



图表 ‑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。

## 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 ‑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、1K大小左右消息堆积情况下吞吐量非常高， 在5W每秒以上，但是如果多个应用都在堆积，那么性能会比较低 2、4K消息性能最差，8K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。

# RocketMQ通信协议

## 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 |

## 心跳处理

## 连接复用

## 按需连接

# RocketMQ服务路由（Name Server）

# 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>

* RocketMQ Benchmark

<http://taobao.github.com/metaq/document/benchmark/benchmark.pdf>

* Documentation for /proc/sys/vm/\*

<http://www.kernel.org/doc/Documentation/sysctl/vm.txt>