RocketMQ封神之旅（三）-RocketMQ的其他考虑

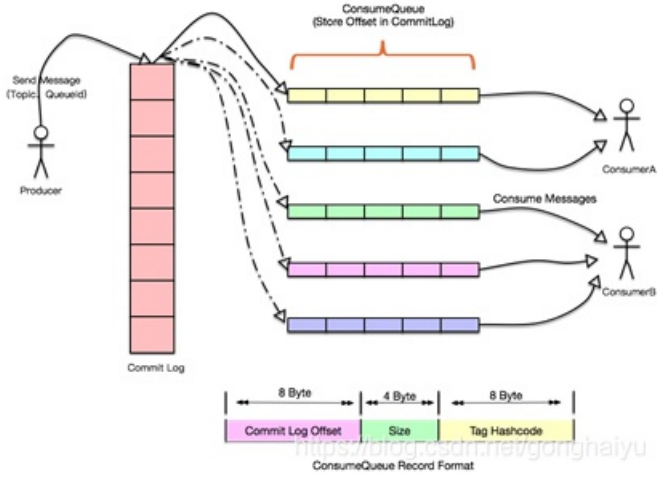
原创

gonghaiyu 于 2021-06-12 23:05:45 发布 1111 收藏

分类专栏： RocketMQ

本文链接：https://blog.csdn.net/gonghaiyu/article/details/117856530

为什么单机支持 1 万以上持久化队列



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。

优点

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。这里是因为采用了单个队列，而不像KafKa采用了多个patition。

缺点

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。这里采用异步实现，后面在源码篇中有讲解。

以上缺点如何克服：

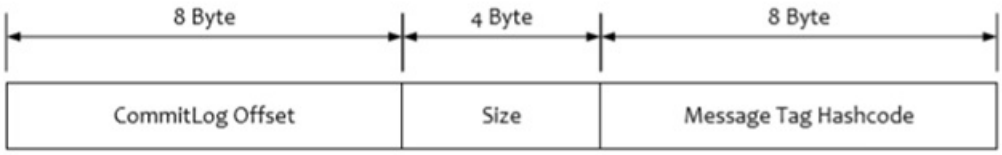
1. 随机读，尽可能让读命中 PAGECACHE，减少 IO 读操作，所以内存越大越好。如果系统中堆积的消息过多，读数据要访问磁盘会不会由于随机读导致系统性能急剧下降，答案是否定的。
   a) 访问 PAGECACHE 时，即使只访问 1k 的消息，系统也会提前预读出更多数据，在下次读时，就可能命中内存。
   b) 随机访问 Commit Log 磁盘数据，系统 IO 调度算法设置为 NOOP 方式，会在一定程度上将完全的随机读变成顺序跳跃方式，而顺序跳跃方式读较完全的随机读性能会高 5 倍以上，可参见以下针对各种 IO方式的性能数据。
   http://stblog.baidu-tech.com/?p=851

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。
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。

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

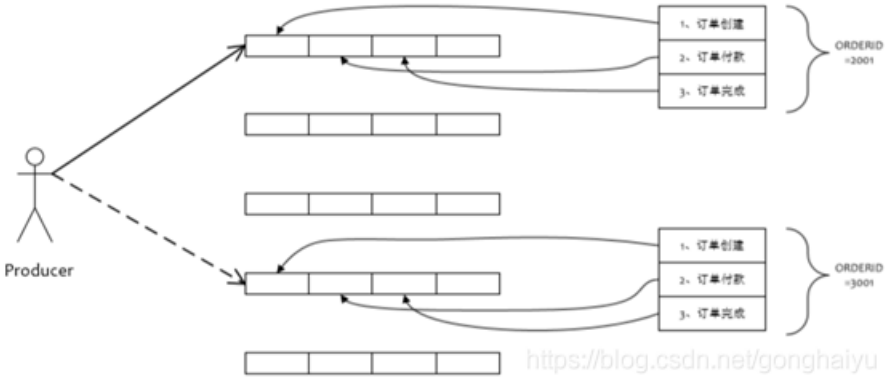
1. 在 Broker 端进行 Message Tag 比对，先遍历 Consume Queue，如果存储的 Message Tag 与订阅的 Message Tag不符合，则跳过，继续比对下一个，符合则传输给Consumer。注意：Message Tag是字符串形式，Consume Queue 中存储的是其对应的 hashcode，比对时也是比对 hashcode。
2. Consumer 收到过滤后的消息后，同样也要执行在 Broker 端的操作，但是比对的是真实的 Message Tag 字符串，而不是 Hashcode。 这里其实就是第二层校验。尤其是存在hash冲突的时候，就可以通过这种方式进行修正。

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。

顺序消息

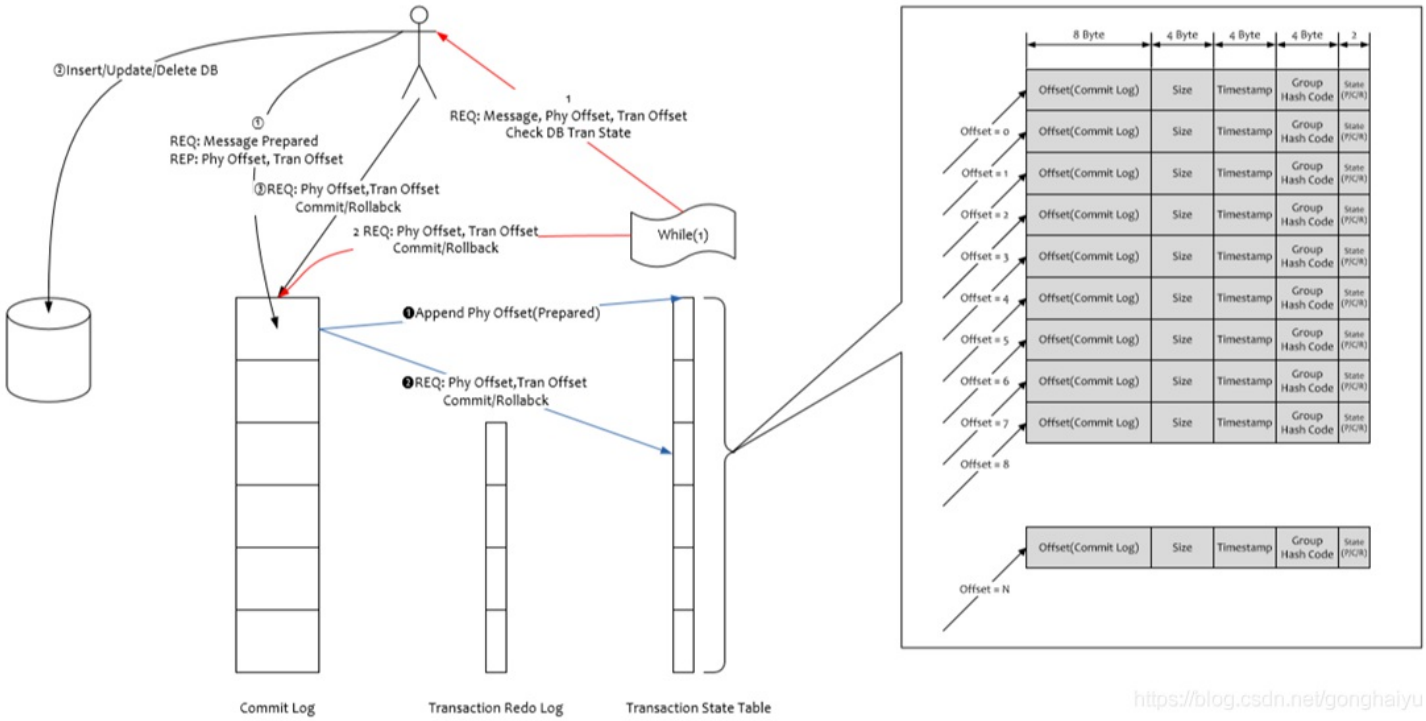
顺序消息原理



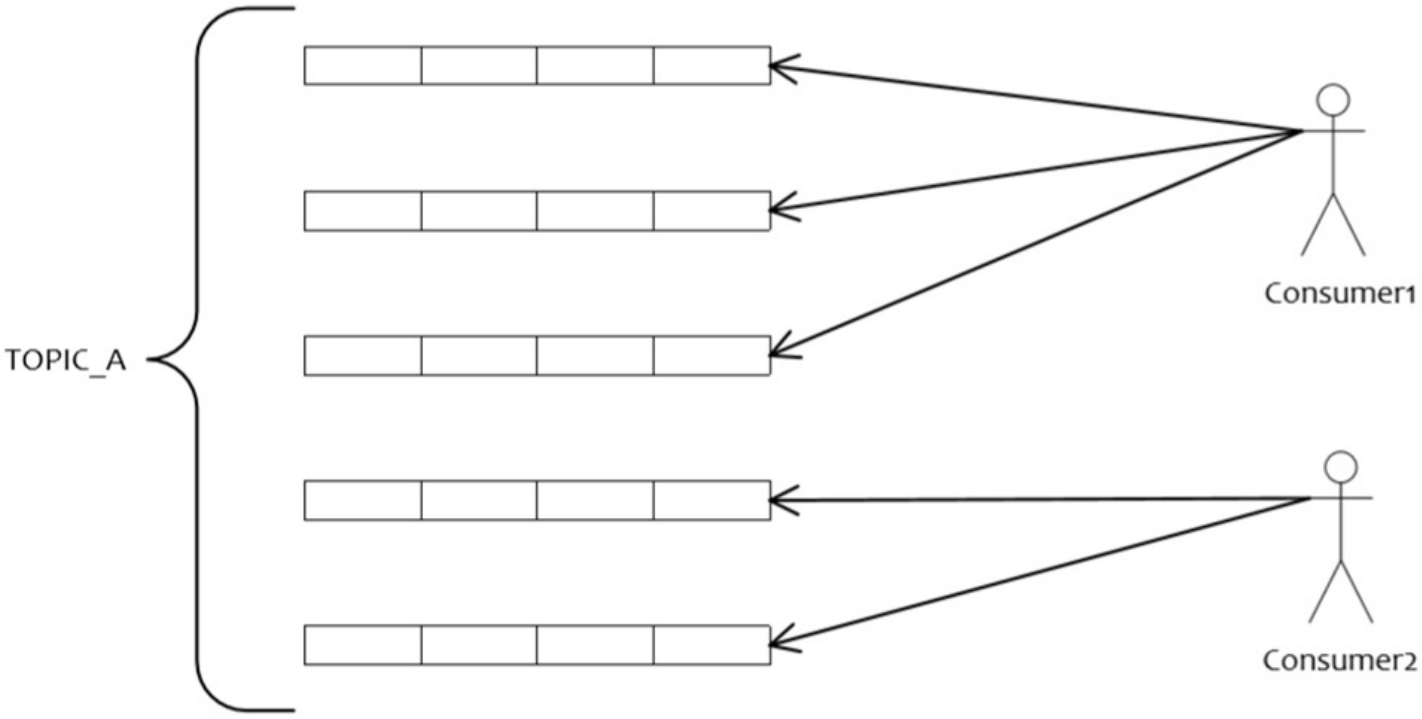
顺序消息缺陷

- 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

事务消息



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

1.1. 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。

1.2. 发送定时消息

1.3. 消息消费失败，定时重试

1.4. 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。

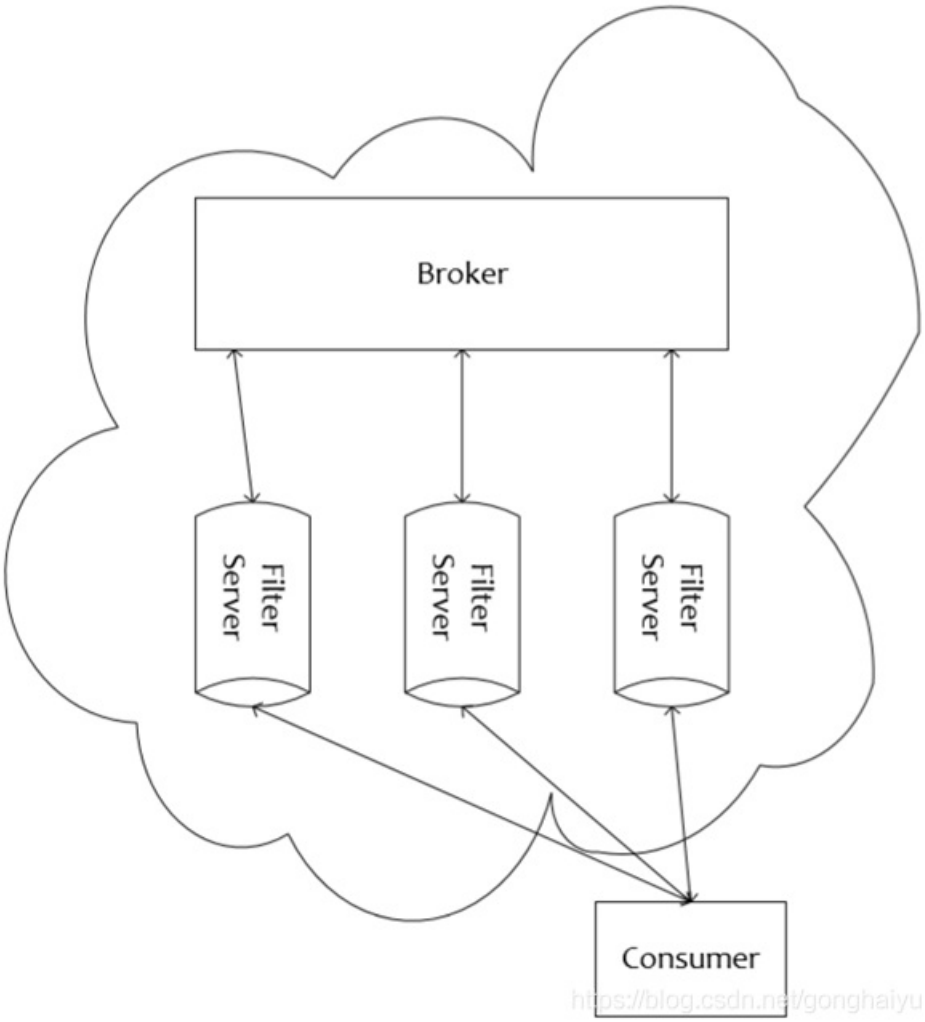
1.1. RocketMQ 消息过滤

1.1.1. 简单消息过滤

```
/**
* 订阅指定topic下tags分别等于TagA或TagC或TagD
*/
consumer.subscribe("TopicTest1", "TagA || TagC || TagD");
```

如以上代码所示，简单消息过滤通过指定多个 Tag 来过滤消息，过滤动作在服务器进行。实现原理参照第 7.4 节。

1.1.1. 高级消息过滤



1. Broker 所在的机器会启动多个 FilterServer 过滤进程
2. Consumer 启动后，会向 FilterServer 上传一个过滤的 Java 类
3. Consumer 从 FilterServer 拉消息，FilterServer 将请求转发给 Broker，FilterServer 从 Broker 收到消息后，按照Consumer 上传的 Java 过滤程序做过滤，过滤完成后返回给 Consumer。
   总结：
4. 使用 CPU 资源来换取网卡流量资源
5. FilterServer 与 Broker 部署在同一台机器，数据通过本地回环通信，不走网卡
6. 一台 Broker 部署多个 FilterServer，充分利用 CPU 资源，因为单个 Jvm 难以全面利用高配的物理机 Cpu 资源
7. 因为过滤代码使用 Java 语言来编写，应用几乎可以做任意形式的服务器端消息过滤，例如通过 Message Header 进行过滤，甚至可以按照 Message Body 进行过滤。
8. 使用 Java 语言进行作为过滤表达式是一个双刃剑，方便了应用的过滤操作，但是带来了服务器端的安全风险。需要应用来保证过滤代码安全，例如在过滤程序里尽可能不做申请大内存，创建线程等操作。避免 Broker 服务器发生资源泄漏。
   使用方式参见 Github 例子
   https://github.com/alibaba/RocketMQ/blob/develop/rocketmq-example/src/main/java/com/alibaba/rocketmq/example/filter/Consumer.java

2. 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。

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

> 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。
>
> 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/

1.1. 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。

1.1.1. 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"
}
}
```


1.1.1. 心跳处理

通信组件本身不处理心跳，由上层进行心跳处理。

1.1.2. 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。

1.1.3. 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。

参考

https://www.cnblogs.com/zxporz/p/12336476.html