# 第三届中间件性能挑战赛比赛攻略

### 团队：一颗赛艇 队员：谭钧升 张明 俞飞樾

#### 第一赛季：《基于Open-Messaging实现进程内消息引擎》

##### 赛题背景分析及理解

题目内容是：阅读Open-Messaging的接口代码(本工程内除了demo目录外的其他代码)，了解Topic，Queue的基本概念，并实现一个进程内消息引擎。

题目提到了阿里新推出的Open-Messaging规范。但其实第一赛季的代码实现完全可以不用看Open-Messaging，重点只需要理解Topic，Queue的概念和题目对于消息顺序的要求即可。下面分析一下比赛涉及到的几个概念：

1. **Producer**: 生产者，负责生产消息并发送到消息引擎。测评开始时，测评程序会启动10~20个Producer，每个Producer在一条线程中，然后每个Producer随机生产某个Topic或者附属于Queue的消息并发送到消息引擎；
2. **Topic**: 消息主题。
3. **Queue**: 队列。抽象的概念，消息可以发送到特定的队列，或者属于某个特定的Topic
4. **Consumer：**消费者。每个消费者必须attach到某个特定的Queue上，而且Consumer和Queue是一一对应关系。同时，Consumer可以订阅多个Topic。测评程序开始消费时，会启动10~20个Consumer，每个Consumer属于一条线程，而且attach到一个queue上，并订阅多个Topic。对于被多个Consumer订阅的某个Topic，这些Consumer直接的消费应该是独立的，也就是每个Consumer都应该全量消费这个Topic的消息。
5. **顺序性要求：**在本次比赛里，消息顺序性只要求局部有序，及某个生产者生产的消息之间有序即可，不同生产者之间生产的消息的顺序可以无序。例如，P1发送M11, M12, M13, P2发送M21, M22, M23，在消费的时候，只要求保证 M11,M12,M13 (M21,M22,M23)的顺序，也就是说，实际消费顺序为: M11, M21, M12, M13, M22, M23 正确； M11,M21,M22,M12,M13,M23 正确； M11,M13,M21,M22,M23,M12 错误，M12与M13的顺序颠倒了；

其中，程序的运行环境是：4核CPU，4G内存，JVM限制为2.5GB内存，硬盘为机械硬盘，写性能为100MB/S，读性能为40MB/S。另外，生产者进程和消费者进程都运行在同一个机器，在生产者进程生产完消息后，kill掉生产者进程，启动消费者进程开始消费。

##### 核心思路

##### 初始版本

理解赛题后，一开始能很自然的想到类似于kafka一样的设计：一个Topic一个文件，一个Queue一个文件。也就是，每条属于TopicA的消息都写到文件TopicA里面去，属于Queue1的消息则写到Queue1里面去。当然，如果消息数目很多，可以限制单个文件的大小，比如超过50MB以后就写到新的文件里面去。这样的设计首先是能完全保证题目要求的消息的顺序性，其次是对于单个文件来说，在生产消息写入文件和消费消息读取文件的时候，都是顺序读写，对于机械硬盘来说这点非常重要。图1所示是生产者的架构图。每个Topic或者Queue都可以抽象为一个文件对象，其中有512KB的内存buffer，当该buffer写满时便写到磁盘上。

消费者则是这样设计的：假如ConsumerA订阅了topic1, topic2, 并且attach到了queue1，那么ConsumerA只需依次读文件topic1,topic2,queue1即可。如图2所示。

除了生产者消费者的设计，第一赛季还有比较重要的一个优化点就是消息体的编码解码。其中，比赛的Message格式是这样的：

DefaultBytesMessage {

KeyValue headers, properties;

Byte[] body;

}

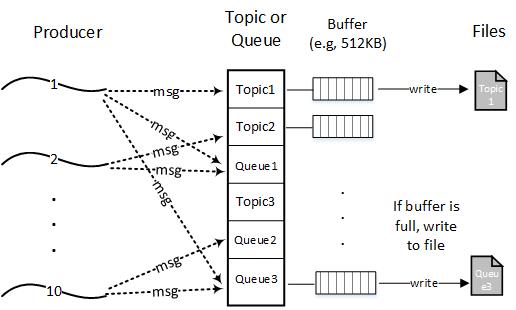


Figure 1生产者v1

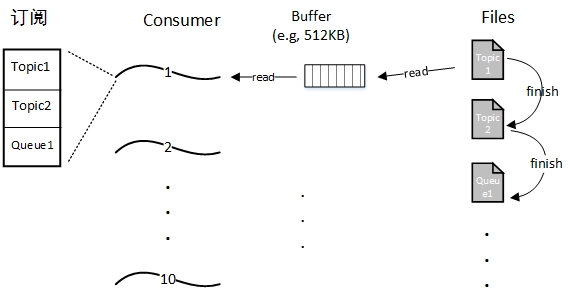


Figure 2 消费者v1

其中，KeyValue是一个HashMap<String, Object>()，而key固定是字符串，value可以是int, long, double, string这四种类型。

因此，对于一条Message，编码格式为：

MSG\_BEGIN (1byte) +

MSG\_HEADER\_BEGIN (1byte) + headers +

[ MSG\_PROPERTY\_BEGIN (1byte) + properties +](可选部分)

MSG\_BODY\_BEGIN(1byte)+ BODY\_LEN\_TYPE(1byte)+ BODY\_LEN (byte|short|int) + body

其中，MSG\_BEGIN, MSG\_HEADER\_BEGIN之类都是自定义的标志位。而KeyValue则编码如下：

KEY\_VALUE\_BEGIN (1byte)+

KEY\_LEN( 1byte) + key +

V\_INT\_TYPE|V\_LONG\_TYPE|V\_DOUBLE\_TYPE|

(V\_STRING\_TYPE + LEN\_TYPE + VALUE\_LEN(byte|short|int)) +value

其中，key的长度只用1byte表示是因为在这个比赛里，键的长度不会超过128.而且实际中这么长的键也比较少见。由于文件都是二进制格式，因此可以直接用ByteBuffer读取，所以需要用1位来表示value的类型，例如，如果value是V\_INT\_TYPE，那么要读取value时可以直接ByteBuffer.getInt()即可。而对于字符串类型的Value，则需要用额外的位来表示字符串字节的长度。在消息编码的时候，用到了变长数字表示的方法。

初始的这个版本的设计，虽然也能跑到最好的接近30万TPS的成绩，但是存在以下不足：

1. 在多条Producer线程并发生产消息时，会有锁竞争：极端情况下，当所有Producer线程生产的都是同一个topic的消息时，由于需要加锁保证正确性，该topic文件的写入便会成为瓶颈。
2. 在Consumer消费消息时，假如消费者C1, C2都订阅了topic1，那么C1，C2都需要各自从头读一遍文件topic1，并且对于每一条topic1消息，都需要解码构造消息。如果有10个消费者都订阅了同一个topic，那么就相当于重复读了10次topic文件，重复解析构造了10次topic的所有消息。

##### 最终版本

针对初始版本提到的两点不足，再结合比赛的特点：只要求单个生产者生产的消息之间有序，我们重新设计了生产者架构，即每个生产者生产的所有消息都存放在自己的文件里，如图3所示。这样，每个生产者都独立生产消息并写文件，在生产消息的过程实现了完全无锁。

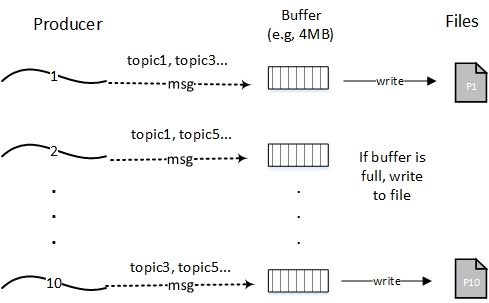


Figure 3 生产者v2

相应的，消费架构需要做出相应的改变：由之前的pull架构改为了push架构。如图4所示。在消费的时候，如果有10个文件，那么就起10条线程，每条线程读一个文件。当线程解析出一条消息的时候，看看这条消息的topic是否有消费者订阅，有的话都是哪些消费者，然后将该条消息push到订阅了该topic的消费者的队列里，让消费者消费。这样，就可以避免初始版本遇到的重复读取文件重复解析消息的问题了。

除了这些大的架构变化外，还有其它的优化点比如：将同一个topic的消息聚集成chunk写入文件，这样每条消息的topic字段便可不用编码，而且在消费的时候读到该topic时如果没有消费者订阅，直接忽略该chunk；压缩数据让pagecache能够存的下所有数据，这样在消费时便不用从硬盘读取，而是直接从pagecache读取。

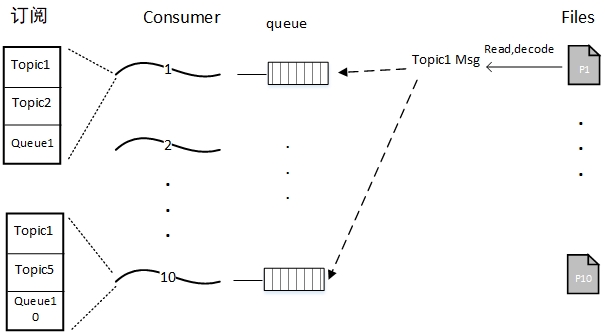


Figure 4 消费者v2