阿里中间件复赛文档

目录

[一、赛题理解与算法设计思路 2](#_Toc486583188)

[二、算法设计的创新点 3](#_Toc486583189)

[2.1无锁队列的设计 3](#_Toc486583190)

[2.2填充法解决伪共享问题 4](#_Toc486583191)

[2.3 批读+并发处理+顺序合并 4](#_Toc486583192)

[2.4 令牌环与自旋锁 5](#_Toc486583193)

[2.5 Socket通信 6](#_Toc486583194)

[2.6存储结构优化与重写HashMap 7](#_Toc486583195)

[2.7 insert操作提前 7](#_Toc486583196)

## 一、赛题理解与算法设计思路

本次中间件大赛复赛题目主要解决的是数据同步领域范畴的一个问题：实时增量同步，主要的技术挑战为模拟数据库的主备复制，提供"高效"的实时同步能力。即给定一批固定的增量数据变更信息，程序需要收集增量变更信息，并进行一定的数据重放计算，然后将最终结果输出到给定的目标文件中。增量数据的变更信息为了简化处理，会给出明文的数据，主要包含数据库的insert/update/delete三种类型的数据。具体的增量数据变更信息的数据格式见格式描述部分。数据重放主要是指模拟数据库的insert/update/delete语义，允许使用一些中间过程的存储。

比赛要求为：1、数据必须全部重放正确；2、不允许使用其他语言或shell脚本；3、对于比赛数据，只允许顺序读取一次，读取的时候单线程读。

经过队员们的讨论，我们认为为了充分利用CPU资源，有必要利用多线程完成增量同步，我们将整体任务分为三个部分，第一部分是数据读取，这一部分我们严格遵守比赛规则，单线程顺序读取一次，然后将读取之后的数据放入我们自己设计的无锁队列当中；第二部分是数据解析，多线程同时从队列中读取数据，解析数据；第三部分是数据合并，由于数据中含有delete和主键变更操作，我们认为为了正确重放数据和保证代码健壮性，需要单线程顺序合并，具体处理流程如下：

主线程成块取数据顺序分发给子线程+子线程顺序解析数据+顺序合并结果:

第一块 … 第j块 … 第n块

放入无锁队列

放入无锁队列

放入无锁队列

取数据

取数据

取数据

子线程解析出数据

子线程解析出数据

子线程解析出数据

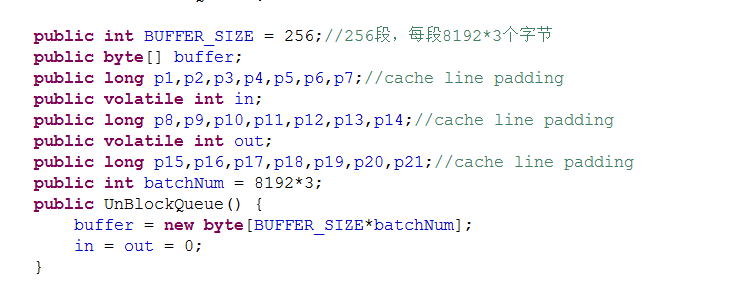
顺序合并

## 二、算法设计的创新点

### 2.1无锁队列的设计

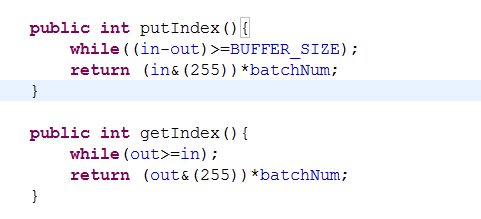
我们的队列模型是单生产者-多消费者模型。每个生产者会将每一块原始数据顺序分发给一个消费者进行解析。而中间的数据结构就是我们自己设计的无锁分段队列。

无锁分段队列用到基础数据结构是一个循环数组，用一个首尾指针记录放入的位置和读取的位置。如下是我们的无锁队列的数据结构。



Buffer是一个循环数组，循环数组被分为了256段，每段代表8192\*3个字节。In和out代表首尾段指针。用关键字volatile标识以确保并发安全。同时我们做了一些伪缓存的优化，具体实现就是在首尾指针前后字节让它不在一个缓冲行中。

生产和消费操作时，首先需要判断首尾指针是否满足条件，如果不满足进行一个自旋的操作。再返回指针的位置。



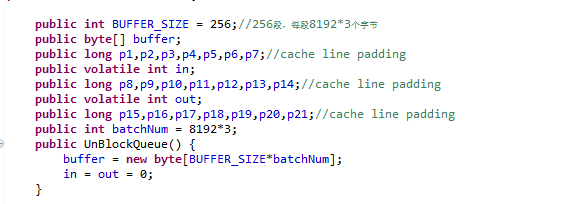
相同算法逻辑下，我们设计的无锁队列能够比直接采用java的blockingqueue快10s以上。

### 2.2填充法解决伪共享问题

本次比赛对的场景对性能要求极高，所以我们需要考虑伪共享带来的影响：

CPU缓存系统中是以缓存行（cache line）为单位存储的，当多线程修改互相独立的变量时，如果这些变量共享同一个缓存行，就会无意中影响彼此的性能，这就是伪共享。我们知道一条缓存行有 64 字节，而 Java 程序的对象头固定占 12 字节( 64 位系统默认开启压缩, 不开压缩为 16 字节)，我们对采用的无锁队列对于队头指针和队尾指针的访问十分频繁，常的被不同的线程修改，但它们却很可能在同一个缓存行，于是就产生了伪共享。线程越多，核越多，对性能产生的负面效果就越大。

所以我们只需要填 6 个无用的长整型补上6\*8=48字节，让不同的 VolatileLong 对象处于不同的缓存行，就避免了伪共享。



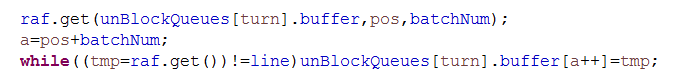
由于某些 Java 编译器的优化策略，那些没有使用到的补齐数据可能会在编译期间被优化掉，我们可以在程序中加入一些代码防止被编译优化。如下：

C:\Users\Bread\AppData\Roaming\Tencent\Users\330504591\QQ\WinTemp\RichOle\L4VM5C7%BG2G~(SO~CBBLL5.png

线上的评测结果表明，解决伪共享问题，确实为我们程序的性能带来了一秒左右的提升。相关代码见UnBlockQueue.java文件

### 2.3 批读+并发处理+顺序合并

在一开始的时候，我们考虑的是不断读取数据判断是不是到了一行的行尾，然而这种效率是非常的低。所以我们考虑了批读的方式，即一开始读一个固定大小的块，再考虑行尾补全。



在快速读完一批之后，我们需要按顺序将每一批数据分发给一个子线程负责解析。子线程解析的文件是我们的subThread文件，将原始数据解析成可用的数据，比如解析出主键值，属性和属性对应的值。

比如update操作：解析出变更前的id值和变更后的id值，以及变更的属性值。



顺序合并，合并我们采用了顺序合并的方式，同时，因为在之前的步骤中主线程是将批数据顺序分发给子线程操作的，所以在合并的时候也得顺序合并子线程的解析数据。因此，我们使用了另一个重要的东西控制，即令牌环。即每个线程拿到令牌才允许进行合并操作。以下讲解我们的令牌环。

### 2.4 令牌环与自旋锁

我们使用了一个变量表示令牌，每次一个线程拿到令牌后并完成合并操作后，变量会执行加一操作。而线程可以拿到令牌的条件是该变量取余线程数等于这个线程的id。



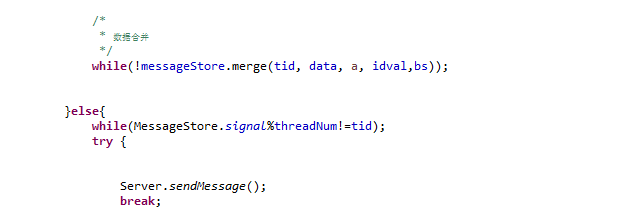
当线程拿到令牌后进行合并操作



比如变更主键操作:

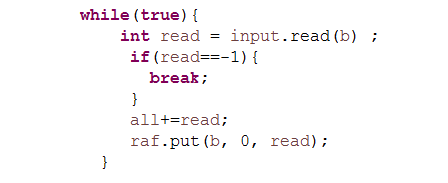


拿不到令牌的时候，我们采用了自旋锁，循环等待资源在短期间内进行轻量级的锁定。相关代码如下：



### 2.5 Socket通信

因为结果数据量不多，而使用netty我们又觉得有延迟且各方面开销比较大，我们就使用了java简单的socket通信，通过指定ip和端口进行socket通信。服务器端比较简单，只需要启动socket，并往socket中写入数据即可。在客户端中需要一直等待数据的到来。并判断通信是否已经结束。



## 2.6存储结构优化与重写HashMap

在数据存储方面，我们将二维数组和hashmap共同使用来尽可能优化效率和存储空间。因为如果全部用数组或者hashmap来存储所有的数据，要么空间损失太多，要么效率不高。鉴于以上考虑，我们将其二者结合。

在id比较小的时候，我们利用二维数组存储数据，即第一维存储id,第二维存储该id对应的全部数据，将其序列化之后用byte存储。



在id比较大的时候，不适宜用数组存储，我们即改为用自己编写的hashmap存储id及其对应的全部数据。



这样二者相结合，能够极大的改进存取数据的效率。

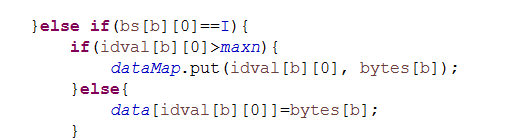
我们将数据利用byte数组存储，对于每一个属性，我们都分配固定长度的存储空间。利用比特存储这样可以极大的优化存储空间，避免不必要的空间浪费。

后期随着程序性能的提高，gc问题一度成为了我们的瓶颈，占据总运行时间的十分之一到八分之一时间，我们采取了一些措施，比如尽量重用对象空间，减少申请新空间，逐步优化代码，将gc时间由原来的1~2s减少到了0.6s左右，但还是心有不甘；

后来我们专门抽出一天时间来针对gc问题进行分析和优化，通过使用jstat等工具的监控，我们将新生代gc问题的主要原因定位于之前使用的HashMap<Integer,byte[]>结构空间效率较低，但由于文件中出现的id较大，map是比较理想的存储结构，所以我们选择了**基于hashmap原理，重写自己的hashmap类**，大大减少了java自带hashmap类由于泛型转化和扩容所带来的时间，同时由于我们重写的hashmap内部采用定长数组，便于良好的控制空间，最终解决了gc问题。

## 2.7 insert操作提前

insert一个id的操作只会完成一次，所以我们将input操作提前完成。即本应在合并操作中完成，我们将其部分任务提升到了线程解析步骤中完成。显然即使多个子线程执行insert操作，因为多个线程不可能对同一个id执行插入操作，所以这种做法是合理的。我们在子线程解析insert操作的时候，我们开辟一个新的区域，用于存储insert的值，而在合并操作中只需要传递这个区域的指针就行了。



如上合并操作中，只需要将这个区域的指针插入最终数据区即可。