第三届阿里中间件性能挑战赛

比赛攻略

团队： 也许放弃才能靠近你

成员：胡建洪、闵大为、胡传铃

# 初赛

初赛题目：《基于Open-Messaging实现进程内消息引擎》

## 赛题背景分析及理解

赛题描述：支撑阿里双十一的万亿级消息中间件RocketMQ在2016年10月正式进入Apache基金会进行孵化。异步解耦，削峰填谷，让消息中间件在现代软件架构中拥有着举足轻重的地位。天下大势，分久必合，合久必分，软件领域也是如此。市场上消息中间件种类繁多，阿里消息团队在此当口，推出厂商无关的Open-Messaging规范，促进消息领域的进一步发展。本赛题要求参赛选手阅读Open-Messaging规范，了解Message，Topic，Queue，Producer，Consumer等概念，并基于相关语言的接口实现进程内消息引擎。

分析与理解：消息中间件分布式系统的消息传递和通信中具有举足轻重的作用，消息中间件的性能在一定程度上决定了整个系统的性能。消息中间件对于生产者而言，写入消息要快，且消息写入后不能丢失；对于消费者而言，消费顺序不能乱，消息到的消息要准确及时。一般而言，要做到高性能的分布式消息中间件，在架构上和实现上都要高屋建瓴，推出通用的规范和概念，便于沟通和交流，以及软件的发展和升级。目前，阿里具有很大的平台，拥有别人没有的数据和技术优势，但是也面临着别人没有的挑战，比如双十一亿万级消息挑战等。通过这次比赛，我们可以更进一步感受和了解阿里的技术以及面临的挑战，通过拓宽自己的眼界，提升自己的技术水平。

## 核心思路

根据题目要求，我们需要实现指定接口，完成消息的生产、存储、消费等功能。其中消息的存储是关键，需要保证消息的正确性和顺序一致性，同时需要持久化到磁盘上，消息持久化的方式和性能影响着生产消息和消费消息的速度。下面是我们的核心思路的具体说明。

### 整体处理流程

整体处理流程如图1所示，首先启动生产进程生产消息，并进行消息存储，由于消息生产和消息消费是不同进程，需要进行消息持久化，保存到磁盘上，待消息生产结束后，结束生产进程，启动消费进程进行消息消费，消费过程需要保证消费到的消息次序和生产时次序一致，且消息正确。



图1 整体流程图

### 消息存储方案

在比赛中，我们实现了多种消息存储方案，分别如下：

方案1：这个消息存储方案为一个Topic或者Queue对应一个文件。每个生产者在写入消息到Topic或者Queue时，先获取Topic或者Queue的互斥锁，获取到锁后，然后再写入消息。每个消费者消费时可以同时打开同一个文件进行消费，互不干扰。

优点：管理简单

缺点：需要加锁以保证消息的正确写入，效率不高；

方案2：这个消息存储方案为一个Topic或者Queue对应多个文件。对于一个Topic或者Queue我们以Topic或Queue的名称建立一个目录，每个线程如果要将消息写入到该Topic或者Queue中，则在目录下创建一个独立文件，该线程的要写入该Topic或者Queue的后续消息都将写入到该文件中。当需要消费一个Topic或者Queue中的消息时，消费者只需依次消费该Toipc或者Queue下对应的目录下的所有文件的消息即可。

优点：无需加锁，写入效率较高。

缺点：消息存储时文件数跟线程相关，耦合度相对较高。

对比方案1和方案2，我们最后采用方案2实现消息存储，线上测试时效率也比方案1要高。

### 消息持久化

消息由头部（headers)、属性（properties)、主体（body)三个部分组成，其中头部和属性为KeyValue结构，消息主体为byte[]数组。

消息存储需要序列化，消息是一个对象，对象的序列化方案主要有JDK的Serializable ，将对象转换为JSON或者XML格式的字符串，自定义存储格式等。其中JDK的Serializable 在序列化时，需要额外写入对象的Class信息，效率不高，且浪费空间。JSON或者XML则需要将对象转换为字符串，多了一个转换过程，增加了CPU开销。我们根据实际情况，采用自定义存储格式进行消息序列化，具体如下：

1. 对于KeyValue结构的Headers或Properties，我们先写入KeyValue总数，然后对于每个KeyValue，先写入Key的字节数组长度，然后写入key的字节数组，再写入Value的字节数组长度，最后写入Value的字节数组。
2. 对于消息主体的byte[]，我们先写入byte[]的长度，然后写入byte[]。

其中，一条消息序列化后的结构为：Header + Properties + body，示意图如图2所示：

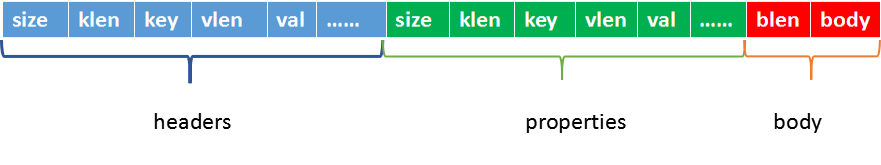


图2 消息序列化结构图

### 数据压缩方案

由于消息数据量大，而IO速度比较慢，因此我们将数据进行压缩后，在进行IO操作，这样做可以减少IO的数据量而提高写入效率。当然这种方案是用CPU开销换IO开销的方案。

对于数据压缩，无损压缩算法主要有Deflater，Snappy，LZ4，QuickLZ等。其中Deflater压缩速度比较慢。在线上测试时，我们使用了Deflater，Snappy，QucikLZ等3个压缩算法，压缩性能依次为：

Snappy > QiuckLZ > Deflater

在数据压缩时，如果每次只压缩一条消息，无疑效率是低下的，因此我们设置消息序列化缓存区，缓存区大小为32KB + 256B，每当缓存区满了，则调用压缩算法将缓存区数据压缩并写入文件。

## 关键代码

消息写入关键代码：

**public** **class** SnappyWriter **implements** IWriter {

**private** RandomAccessFile file;

**public** **static** **final** **int** ***DEFAULT\_SIZE*** = IConstants.***CMP\_MS***;

**private** **byte**[] bytes = **new** **byte**[***DEFAULT\_SIZE*** + IConstants.***MSG\_ML***];

**private** **int** p;

**private** **byte**[] cmp = **new** **byte**[***DEFAULT\_SIZE*** + IConstants.***MSG\_ML***];

**public** SnappyWriter(String dir) **throws** IOException {

file = **new** RandomAccessFile(dir, "rw");

}

**private** **void** put(**byte** a) {

bytes[p++] = (**byte**) (a & 0xff);

}

**private** **void** put(**byte**[] bs) {

**int** a = bs.length;

**if**(a < Byte.***MAX\_VALUE***){

bytes[p++] = (**byte**) (a & 0xff);

} **else** {

bytes[p++] = (**byte**) ((a >> 8 & 0xff) | 0x80);

bytes[p++] = (**byte**) (a & 0xff);

}

System.*arraycopy*(bs, 0, bytes, p, bs.length);

p += bs.length;

}

**public** **void** write(DefaultBytesMessage message) **throws** IOException {

DefaultKeyValue h = (DefaultKeyValue)message.headers();

DefaultKeyValue pro = (DefaultKeyValue)message.properties();

// 减少一个字节存储头部和属性的长度

**byte** tsize = (**byte**) ((((**byte**)h.num) << 4 & 0xf0) | (pro.num & 0x0f));

put(tsize);

**for** (**int** i = 0; i < h.num; i++) {

put(h.keys[i].getBytes());

put(h.values[i]);

}

**for** (**int** i = 0; i < pro.num; i++) {

put(pro.keys[i].getBytes());

put(pro.values[i]);

}

**byte**[] body = ((DefaultBytesMessage) message).getBody();

put(body);

**if** (p >= ***DEFAULT\_SIZE***) {

**int** clen = Snappy.*compress*(bytes, 0,p, cmp, 0);

file.writeShort(clen);

file.write(cmp, 0, clen);

**this**.p = 0;

}

}

**public** **void** close() **throws** IOException {

**if** (**this**.p > 0) {

**int** clen = Snappy.*compress*(bytes, 0,p, cmp, 0);

file.writeShort(clen);

file.write(cmp, 0, clen);

}

file.writeShort(0);

file.close();

}

}

消息读取关键代码：

**public** **class** SnappyReader **implements** IReader {

**private** **int** cnt = 0;

**private** MappedByteBuffer[] mBuffers;

**public** SnappyReader(String dir) **throws** IOException {

File dirFile = **new** File(dir);

**if** (!dirFile.exists()) {

**this**.complete = **true**;

**return**;

}

File[] files = dirFile.listFiles();

**if** (files.length <= 0) {

**this**.complete = **true**;

**return**;

}

mBuffers = **new** MappedByteBuffer[files.length];

**for** (**int** i = 0; i < files.length; i++) {

@SuppressWarnings("resource")

FileChannel channel = **new** FileInputStream(files[i])

.getChannel();

mBuffers[i] = channel.map(MapMode.***READ\_ONLY***, 0, channel.size());

}

mBuffer = mBuffers[cnt++];

}

**public** BytesMessage read() **throws** Exception {

**if** (complete) {

**return** **null**;

}

**if**(p < limit){

**return** fromBuffer();

}

**int** len = 0;

**while** ((len = getLen()) == 0) {

**if** (cnt == mBuffers.length) {

**this**.complete = **true**;

**return** **null**;

}

mBuffer = mBuffers[cnt++];

}

mBuffer.get(cmp, 0, len);

limit = Snappy.*uncompress*(cmp, 0, len, bytes, 0);

p = 0;

**return** fromBuffer();

}

**private** **int** getLen(){

**byte** b1 = mBuffer.get();

**byte** b2 = mBuffer.get();

**return** ((((b1 & 0xff) << 8) | (b2 & 0xff))) & 0x7fffffff;

}

**private** **byte**[] cmp = **new** **byte**[IConstants.***CMP\_MS*** + IConstants.***MSG\_ML***];

**private** **byte**[] bytes = **new** **byte**[IConstants.***CMP\_MS*** + IConstants.***MSG\_ML***];

**private** **int** p;

**private** **int** limit;

**private** ByteBuffer mBuffer;

**private** **boolean** complete = **false**;

**private** QingBytesMessage msg = **new** QingBytesMessage();

**private** QingKeyValue hds = **new** QingKeyValue(2);

**private** QingKeyValue pros = **new** QingKeyValue(4);

**private** **static** **final** String ***TOPIC*** = MessageHeader.***TOPIC***;

**private** **static** **final** String ***QUEUE*** = MessageHeader.***QUEUE***;

**private** BytesMessage fromBuffer() {

**byte** tsize = getByte();

**int** keySize = (tsize >> 4 & 0x0f);

**int** pSize = (tsize & 0x0f);

hds.clear();

**for** (**int** i = 0; i < keySize; i++) {

String key = getString();

**switch** (key) {

**case** "T":

hds.put(***TOPIC***, getBytes());

**break**;

**case** "Q":

hds.put(***QUEUE***, getBytes());

**break**;

**default**:

hds.put(key, getBytes());

**break**;

}

}

msg.setHeaders(hds);

**if**(pSize > 0){

pros.clear();

**for** (**int** i = 0; i < pSize; i++) {

pros.put(getString(), getBytes());

}

msg.setProperties(pros);

} **else** {

msg.setProperties(**null**);

}

msg.setBody(getBytes());

**return** msg;

}

**public** **int** getShort() {

**return** (((bytes[p++] & 0xff) << 8) | ((bytes[p++] & 0xff)));

}

**public** **byte** getByte() {

**return** (**byte**) (bytes[p++] & 0xff);

}

**public** String getString() {

**int** a = (bytes[p++] & 0xff);

String s = **new** String(bytes, p, a);

p += a;

**return** s;

}

**public** **byte**[] getBytes() {

**int** a = (((bytes[p++] & 0xff) << 8) | ((bytes[p++] & 0xff)));

**byte**[] b = **new** **byte**[a];

System.*arraycopy*(bytes, p, b, 0, a);

p += a;

**return** b;

}

}

## 总结

1. 尽量实现对象复用，避免频繁创建对象，避免频繁GC。
2. 能够避免加锁就避免，如果无法避免，则将加锁范围控制在最小范围内。
3. 减少中间数据拷贝过程，尽量直接将数据拷贝志最终目标处，能极大的提高效率。
4. 代码越优化越简单，往往简单的代码比复杂的代码可读性高，性能也相对来说比较高。

# 复赛

复赛的题目：《模拟阿里双十一分布式数据同步》

## 赛题背景分析及理解

赛题描述：题目主要解决的是数据同步领域范畴：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。

分析与理解：实时增量同步中的“实时”就要求数据延迟不能很长，从而要求数据传输以及数据处理过程的效率都要高。

## 核心思路

该赛题需要对原始数据进行解析，然后进行数据处理，最后输出最终要查询的结果，因此我们的核心思路围绕这三个部分展开，下面是我们的核心思路的具体说明。

### 整体处理流程

服务器端和客户端的整体处理流程图如图3所示。为了加快数据解析和数据处理速度，提高资源利用率，我们在服务器端和客户端都采用了并行化设计。因为数据之间的存在一定相关性（操作依赖于前面的操作、操作与前面的冲突），所以要进行并行化设计是一个比较麻烦的事情，但考虑到这个并行化设计和指令执行的优化非常相似（指令之间存在相关性），所以我们参考了指令集的流水化设计，如图4所示。我们服务器端和客户端都采用了**三段式流水化设计**：服务器端中从文件读取数据，按行进行解析，然后将结果存入map中；客户端中从socket中读取数据，翻译成目标数据（传输过程中传的是处理过的数据，为了节省传输消耗），然后将结果写入目标文件。**为什么不再细化流水线？**实际测试中，当继续细化下去的时候，数据转移的开销已经大于本线程处理的收益了，所以最后采用了三段式。

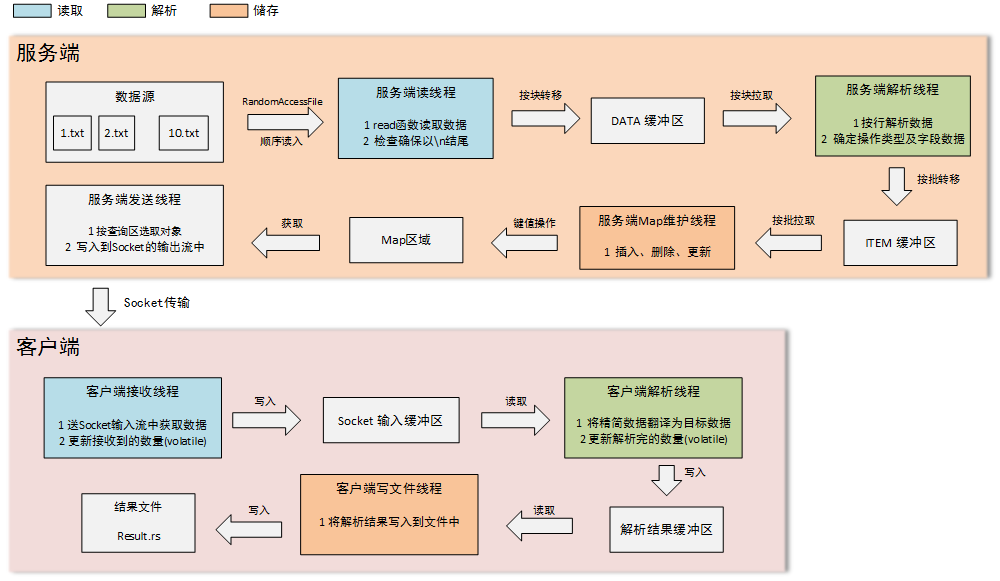


图3 整体流程图

在服务器端，多个线程间要使用线程安全的队列交换数据，为了减少内存开销，我们使用数组和信号量实现了一个基于数组的线程安全的循环队列缓冲区，具体如下：

1. FULL信号量表示数组有多少数据，初始为0；
2. EMPTY信号量表示数组有多少空位置，初始为数组的大小；
3. 存入数据时，进行EMPTY.acquire()，存入后进行FULL.release()；
4. 使用数据时候，进行FULL.acquire()，使用完后进行EMPTY.release()；

其中，图1中蓝色部分是服务端处理的流程图，服务端启动后会开启3个线程进行数据处理，三个线程的作用和处理过程如下所示：

1）**读线程：**该线程通过RandomAccessFile对象从数据源读取数据，然后进行检查是否为完整一行，没有则继续读入补全行，之后提交到DATA数据缓存区。

2）**解析线程：**该线程从DATA数据缓存区拉取数据，然后按行解析数据，并将处理好的数据提交到ITEM缓存区。

3）**Map维护线程：**该线程从ITEM缓存区拉取数据，进行插入、删除、更新等操作。

其中，图1中红色部分是客户端处理的流程图，客户端启动后也会开启3个线程进行数据处理，三个线程的作用和处理过程如下所示：

1）**接收线程：**该线程从Socket输入流中获取数据（存在与下一个线程共享的数组中），并更新接收到的数据数量。

2）**解析线程：**该线程轮询收到的数据数量，如果可以读取新的一行记录（12字节，int + long），则进行解析，并更新已解析完的数量。

3）**写文件线程：**该线程轮询已经解析完的数据数量，如果有新解析完的数据，则将新数据写入到文件中。

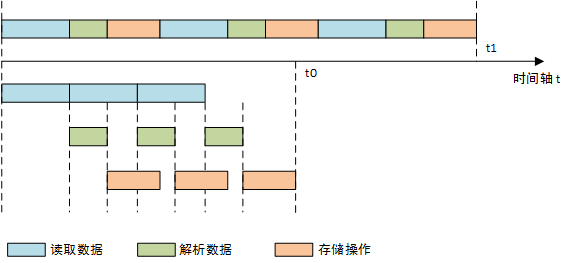


图4 流水化设计图

### 数据处理

数据重放操作需要对原始记录信息进行解析处理，而原始数据的文件每行的处理格式如表1所示。

表1 文件每行记录格式

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| binaryId | timestamp | schema | table | 变更类型 | 列信息 | 变更前列值 | 变更后列值 | 列信息 | 列值 | … |

为提交解析速度和效率，我们进行了列信息提取，偏移处理和预测等操作，具体如下:

1. **列信息提取**。因为第一行记录已确保是I操作，包含全量的列信息以及对应的值。我们提取出列信息，包括列键值，列的类型，列的总数等。
2. **偏移信息处理**。 经过对文件每行记录格式的观察，可以发现除binaryId和列值的长度是不定的，其他部分信息的长度是固定，如timestamp,schema,table,列信息等，因此我们处理第一行记录时，计算出这些长度固定信息的长度并记录在一个一维数组中，方便后续处理时进行偏移预测，直接跳过不关心的信息，提高解析速度。其中，我们也对不定长信息的长度做了下估计，具体如下：

a.binaryId的最短长度，具体做法为用第一行记录的binaryId的长度减去10个字节的长度作为binaryId的最短长度，后续处理时利用该长度进行偏移预测，在预测失败时重新计算binaryId的最短长度。

b.删除操作快速跳过长度，变更类型为D的操作，只需提取第一列的第一个主键信息即可，后面的带有的其他列的信息可以直接跳过，这个快速跳过长度计算方式为第一行记录的第二列的起始位置到第一行结束位置的长度减去20字节的长度。

1. **变更类型偏移位置预测**。我们的方法是采用上一条记录变更类型的偏移值来预测这一次的变更类型，如果预测对了这个偏移值就会保留给下一次使用，如果预测错了，那么就会从记录头开始解析，把这次解析的值作为下一次预测值。其中，理论论证如下所示：
2. 收益性：因为变更类型前面除了binaryId变动，时间戳、库名、表名的长度都是固定的，而且相邻记录的binaryId变动不大，这会导致预测命中率很高，所以按成功率计算，预测成功节省的计算量远远大于预测失败的付出的计算量，所以收益可观。
3. 可行性：以下两个前提是预测方案可行的充分必要条件。

* 所有字段中只有变更类型会出现大写的U、I、D，也就是预测位置标志独一无二;
* 预测长度不会跳跃到其他记录的部分（binayId的生成是前缀+文件偏移，经过统计这个binayId长度最大变动的范围是8左右，不会越过本记录的范围）。

1. **更新操作中更新类型预测**。前提：更新的时候只更新一个值。分为如表2和表3两种以下情况。好处：预测正确可以少算一次主键。算法：计算主键1，如果不是更新主键,那么下一个主键也是主键1，长度可知，在主键1长度+1的位置是分隔符，根据此思想可以实现预测。但是要预防以下两个问题：

a.问题：预测可能出现误判，就是“|”不够唯一。

解决：还要偏移一定位置寻找标志位，程序中是4。

b.问题：可能超出记录长度，由于缓冲区复用，可能受到旧数据的影响。

解决：增加判断，超出范围后放弃预测。

表2 更新主键

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ... | 主键1 | 主键2 | 分隔符 | 换行符 |

表3 更新其他字段

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| ... | 主键1 | .主键1 | 列信息 | 变更前列值 | 变更后列值 | 分隔符 | 换行符 |

### 数据表示

数据表示：指的是数据库表中一行记录如何表示，便于我们后续增删改。在比赛过程中，随着我们不断的改进和有优化代码，我们使用多种数据表示方案表示一行记录。演进过程具体如下。

**版本1：**一开始我们使用了KeyValue自定义类来表示表的一条记录（String[] keys,String[]vals）keys存储数据库表的属性名，vals存储数据库表属性值。我们发现如果读取文件后存储在String数组中会创建大量的String对象，代价比较高。

**版本2：** 后来我们用（char[][]keys,char[][]vals）代替了原来的存储，但是这样都是需要将nio读取到的byte数据转化为char数据，多了一个转化的过程。

**版本3：**所以我们顺理成章的改为了（byte[][]keys,byte[][]vals），进一步根据比赛题目的数据特征只有一张表，属性之类的都是固定的，所以就决定去掉keys了，只保留byte[][]vals了。因为最后结果需要对主键进行排序输出，所以一开始我们是使用了TreeMap<Integer,byte[][]>来存储主键变更信息。然后我们发现用TreeMap代价比较高O(nlogn)，要是可以直接用数组就可以O(n)的时间复杂度对主键进行排序了，但是遇到了一个问题，数组开不了那么大，所以我们就用了hashMap代替，最后根据(startKeyID,endKeyID)过滤掉一些数据后就能在数组里面放下了，所以过滤掉之后我们在自己用数组排序。

**版本4：**由于HashMap需要使用Integer包装类，所以会进行一些装箱、拆箱的操作。我们找了一个第三方的IntObjectHashMap这样就可以避免频繁的拆装箱操作（也算是进一步优化吧）。接下来的演化就包含了2个map存储，byte[][][]map1存储endKeyID以内的主键和TreeMap<Integer,byte[][]>存储剩下的主键信息。

上述四个版本是健壮性较高，基本不依赖数据的特性，实现数据重放操作。下面4个版本是我们观察数据规律和数据特征演化出来的版本，当然最后为了成绩牺牲了健壮性，也是不得已而为之。

**版本5：**对数据集特性的观察，发现姓名等字段最多包含2个汉字，一个汉字的UTF8编码3个字节，可以压缩成2个字节，2个汉字正好可以压缩成4个字节，可以使用int直接表示1到2个汉字；另外数值字段的范围都在int范围内，因此这里直接使用int[]表示一行记录，整个记录集表示为int[][],其中第一维为记录所在行，第二维为记录的实际值，考虑到更新操作的部分更新和频繁性，后面调整为列式存储，即仍然采用int[][]表示整个记录集，但是第一维为记录的列值下标，第二维为记录所在行。

**版本6：**然后我们考虑到一共5个属性，first\_name,last\_name,sex,score1,score2.这5个属性可以分开处理sex可以用boolean存，score1,score2可以用int存，我们发现first\_name具有特殊性只要不到100个姓，所以我们利用数据特征存储了汉字的索引只用了一个byte.这5个属性用一个Item封装了。

**版本7：**然后进一步演进，我们竟然发现可以用long来存储这5个属性。first\_name用8位存储、last\_name用16位存储、sex用1位存储、score1（33-8414）用14位存储、score2（40-30039）用19位存储，总共58位存储，在long表示范围之内（long 64位），所以我们最后关于主键记录的存储直接简化为long了。也就是long map1[]、HashMap<Integer,Long> map2了。这样对于增删改就可以通过位运算高效的完成了。

**版本8：**还有继续优化的地方是map2的HashMap了，所以自己实现了一个IntLongHashMap，一方面可以减少拆装箱操作，同时为了减少冲突我们建立了map2数组，自己实现了hash操作，pk>>size(map2数组大小)值相同放在同一个map当中，map当中hash值是取pk的低位了，由于之前屏蔽掉了低位，所以后面hash冲突的概率就比较小了。

对于版本7和版本8，为了节省内存，我们用一个long来存主键的之外的数据。**这一个优化的突出贡献在于：减少对象创建的开销、利于缓冲区对象传递、节约服务端发送数据到客户端的时间（一半以上）。**表4是具体的表示方案。

表4 一个long表示方案

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 字段 | score2 | sex | lastName\_2 | lastName\_1 | firstName | score |
| 说明 | 分数2 | 性别 | 名字部分2 | 名字部分1 | 姓 | 分数 |
| 范围 | 40~30039 | 男、女 | 1个中文 | 1个中文 | 1个中文 | 33 ~ 8414 |
| 位数 | 19 | 1 | 10 | 10 | 10 | 14 |

**优化：**

1）为了减少数据插入和更新时候字段的位移次数，我们对字段位置进行了调整，调整原则是：**频繁更新的放低位；字段位数少的放低位。**

2）一个中文字符转为Unicode至少需要16位，开始我们的解决方案是建立一个HashMap，存的是索引key，因为**实际中文字符只有72个，8位就够了**，但是考虑到HashMap访问有开销，我们反复试验**找到了一个映射函数，可以使16位的Unicode映射为10位的数据表示，减少了访问Map拿索引的开销。**

其中，表5是一行数据表示演变一览表。

表5 一行数据表示演变表

|  |  |  |  |
| --- | --- | --- | --- |
| **版本演变** | **一行记录表示方式** | **版本演变** | **一行记录表示方式** |
| **1** | KeyValue(String[]) | **5** | int[] |
| **2** | KeyValue(char[][]) | **6** | Item |
| **3** | KeyValue(byte[][]) | **7 （重要演化）** | long |
| **4** | KeyValue(byte[][]) | **8** | long |

### 数据存储优化

表6是数据重放处理过程中的存储方式。

表6 记录存储表示演变表

|  |  |  |
| --- | --- | --- |
| **版本演变** | **Map1** | **Map2** |
| **1** | KeyValue(String[]) | TreeMap[] |
| **2** | KeyValue(char[][]) | TreeMap[] |
| **3** | KeyValue(byte[][]) | TreeMap[] |
| **4** | KeyValue(byte[][]) | IntObjectHashMap[] |
| **5** | int[][] | IntIntHashMap[] |
| **6** | Item[] | TreeMap[] |
| **7 （重要演化）** | long[] | HashMap[] |
| **8** | long[] | IntLongHashMap[] |

由于主键范围比较大，所以我们采用了一个数组map和多个IntLongHashMap组合的方式。代码如下：

long[] map1 = new long[CAP];

IntLongHashMap[] map2 = new IntLongHashMap[SIZE + 1];

为了减少自动装箱拆箱、多余操作、函数调用的开销，我们改写了Netty中的IntLongHashMap，做了精简。此外，我们还关注了hash的冲突。从主键到IntLongHashMap中数组下标的映射至少有两步，第一步是映射到指定IntLongHashMap，然后根据主键的hash再映射到指定位置。最后实现中我们采用高11位及以上的字段来进行IntLongHashMap的寻找，采用低10位来进行IntLongHashMap内部数组下标的查找，这样可以较好的避免hash冲突。

### 网络传输优化

网络传输是为了将最终结果传给客户端写入结果文件，我们在版本演化过程中实现了三个版本的网络传输方案，分别如下所示，表7是网络传输优化过程中不同方案的简单说明和优化效果。

方案1：服务器使用Snappy压缩算法压缩最终结果，然后使用Netty框架将结果传给客户端；客户端使用一个线程接收结果，然后直接解压后写入结果文件。

方案2：服务器端直接使用Socket将最终结果传给客户端，客户端使用一个线程接收结果直接写入结果文件；

方案3：服务器端使用Socke传输中间结果给客户端，客户端开始三个线程处理，一个线程接收中间结果，一个线程解析中间结果为最终结果，一个线程将最终结果写入结果文件中。

表7网络传输优化方案

## 关键代码

列信息提取关键代码：

**private** **void** handleHeader() **throws** IOException {

p ++;

**while**(bytes[p ++] != IConsts.***P***); // 跳过BinaryID

p += **this**.stableSkip; // 跳过timestamp | schema | table

p += 2;

**byte**[][] vals = **new** **byte**[***KS***][];

**while** (bytes[p] != IConsts.***N***) {

**int** s = p;

qbuf = getCKey(); // 获取列键值

keyLen[h] = p - s; // 列信息长度

**while** (bytes[p++] != IConsts.***P***); // 跳过旧值

pbuf = getBytes(); // 获取新值

keys[h] = qbuf;

vals[h++] = pbuf;

}

p ++;

**byte**[][] nvals = **new** **byte**[h][];

System.*arraycopy*(vals, 0, nvals, 0, h);

BucketMap.*put*(getIntPK(nvals[0]), nvals); // 插入map

}

偏移处理关键代码：

**private** **void** handleSkip(){

// 确定一行的最大长度

**while**(bytes[lml ++] != IConsts.***N***);

// 加上100字节大小

lml += 100;

// 确定timestamp | schema | table的长度,即stableSkip

**int** skip = 1;

**while**(bytes[skip ++] != IConsts.***P***); // 跳过BinaryID

**int** s = skip;

// 跳过 timestamp | schema | table | 三个'|'

**while**(bytes[skip ++] != IConsts.***P***);

**while**(bytes[skip ++] != IConsts.***P***);

**while**(bytes[skip ++] != IConsts.***P***);

**this**.stableSkip = skip - s;

}

数据解析关键代码：

**private** **void** moreRead() **throws** IOException {

**while** (!**this**.eof) {

**switch** (getOp()) {

**case** IConsts.***I***: // 插入操作

**byte**[][] vals = **new** **byte**[h][];

**int** pos = 0;

**while** (bytes[p] != IConsts.***N***) {

p += **this**.keyLen[pos] + 1; // 跳过列信息

**while** (bytes[p++] != IConsts.***P***); // 跳过旧值

vals[pos ++] = getBytes(); // 获取新值

}

p ++;

BucketMap.*put*(getIntPK(vals[0]), vals); // 获取主键,插入map

**break**;

**case** IConsts.***U***: // 更新操作

p += keyLen[0]; // 跳过主键列信息

qbuf = getBytes(); // 获取旧值

pbuf = getBytes(); // 获取新值

**int** opk = getIntPK(qbuf); // 旧主键

**int** npk = getIntPK(pbuf); // 新主键

**byte**[][] rec = BucketMap.*get*(opk); // 找到记录，并更新

**if** (opk != npk) { // 主键不同,更新主键值

rec[0] = pbuf;

BucketMap.*rap*(opk, npk, rec); // 移除原先的值,重新插入map

}

**while** (bytes[p] != IConsts.***N***) { // 其他列值获取值

qbuf = getCKey(); // 获取列键值

**while** (bytes[p++] != IConsts.***P***); // 跳过旧值

pbuf = getBytes(); // 获取新值

**int** in = 1;

**for**(**int** i = 1;i < h;i ++){

**if**(equals(keys[i],qbuf)){

in = i;

**break**;

}

}

rec[in] = pbuf; // 更新值

}

p ++;

**break**;

**case** IConsts.***D***: // 删除操作

p += **this**.keyLen[0]; //跳过主键列信息

BucketMap.*remove*(getIntPK(getBytes())); //从Map中移除记录

**while** (bytes[p++] != IConsts.***N***); // 跳过其他列

**break**;

}

}

}

## 健壮性

在代码版本演化过程中，使用了不同的记录存储表示方案，因而导致对健壮性的支持程度不同。表8为版本健壮性变化表。

表8 版本健壮性变化表

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **版本** | **健壮性** | **最佳成绩** | **程序依赖数据特性** | **受限制原因** | **解决方案** |
| 1 | 高 | 113070 | 无 | 无 |  |
| 2 | 高 | 50273 | 无 | 无 |  |
| 3-4 | 高 | 45677 | 无 | 无 |  |
| 5-6 | 中 | 19329 | 姓名最多两个汉字， | 姓名汉字超过两个汉字后，Int无法表示压缩后的多个汉字编码 | 使用byte[]直接储原始信息 |
| 数字范围在int范围 | Int无法存储超过Int表示范围的数据 | 使用byte[]直接储原始信息 |
| 7-8 | 弱 | 5283 | 插入主键递增 | 直接使用一个递增int变量 | 增加插入主键解析代码 |
| 中文个数、score, score2范围 | long位数有限，增加后不能表示完全 | 切换到4.1中之前的一个表示方案 |
| 每次只更新一个字段 | 会影响2.3.2中的 更新操作中更新类型预测 | 去掉更新预测，走正常解析 |

## 总结

抓住主要瓶颈点，再去优化细节，往往解决一个大的瓶颈点，便是一个里程碑。而细节则会起到锦上添花的作用。最开始我们完全使用单线程版本完成数据读取，数据解析，数据处理三个部分工作，成绩最好为19秒左右。后来发现这三个部分可以并行处理，因此我们实现了多线程版本的处理过程后，我们成绩达到7秒左右，实现里程碑式跨越。后来进行局部优化，终于将成绩优化到5.2秒左右。

# 比赛经验总结和感想

## 经验总结

1. 尽量减少频繁创建对象过程，避免创建大对象；
2. 不要过分迷信JDK提供的API，有时自己的实现比JDK提供的API效率要高；
3. 减少中间过程，直接一步到位，计算效率会提高不少；
4. 基础知识的掌握程度，往往在关键时刻能起到决定性作用；

## 感想

本次比赛收获良多，不仅在眼界上有所突破，在技术上也有所突破。俗话说的好，眼界决定境界，既然眼界宽了，希望接下来能够继续沉淀技术，提升自己的编程境界。本次比赛使用的数据结构或者算法都是最基础的知识，比如不管在第一赛季还是第二赛季，都使用大量的位运算操作来提高运算效率和减少存储空间；其次，hash以及分桶等基本编程思想的运用等等。随着比赛的推进，代码是越写越简单，感觉如果方向是对的，思路是对的，往往简单的代码比复杂的代码的效率要高的多，代码可读性也相对较高。排行榜的变化带来的感受是，当你以为有戏时，接下来大佬会告诉你没戏了！当你以为没戏时，调整参数优化代码后再提交，又有戏了，然后就是在有戏没戏之间来回循环，直到最后一刻，搞的身心疲惫，但同时也收获不少。最后就是团队的精诚合作，其利断金，感谢队友们两个月以来的辛劳付出，才会有我们最后闯进决赛的结果，非常感谢！总之，在参赛过程中，我们不仅提高了技术能力，也积累了经验心得，更对所学到的知识有了一个更加深刻的认识！这真是一次难忘的编程体验！

-----胡建洪

本次比赛代价很大，在要毕业找工作之际，每一份每一秒都是那么的宝贵，“任性地”没有去复习知识而来参加这个比赛，一个很重要的原因是去年也来了，但是复赛排在38，心有不甘，所以拿出研究生阶段最宝贵的一段时间来做这个比赛。不过参加这个比赛还是非常有价值的，虽然通过这个比赛技术上收获很大，但是更重要的是心志的历练。曾经多少次说，真的已经优化不了了，真的想不出思路了；曾经多少次，又有一种柳暗花明又一村的感觉，成绩提高很多；而这所有的进步都来自团队的坚持和努力。众人拾柴火焰高，当一个团队都非常渴望胜利并忘我付出的时候，那时候的力量是巨大的，你也会因为在这样的团队而感到自豪和快乐。最后感谢队友们的辛勤付出！

-----闵大为

本次比赛对我来说，难度还是挺大的，不过幸运的是遇到了两位大佬，和他们一起比赛的过程中学到了很多东西。6月份本身是比较忙的，一方面要帮一个同学完成她的毕设数据处理，一方面要参加这个比赛，所以这个比赛我做的有限，不过还是学到了很多东西；之前的并发编程很大程度上是停留在书本上，对于信号量这种偏底层的技术更是接触的少；除此之外还有就是平时忽略的优化细节问题、代码规范性问题。不过最重要的是认识了两位技术牛人，结识了一个优秀的团队。最后感谢两位队友的付出，让我也有了一次前所未有的编程体验！

-----胡传铃