MapOutputBuffer理解的三重境界

摘要

MapOutputBuffer作为MapTask的内部类，是MR中二次排序非常重要的一环。本文从基本认识，到详细过程，再到源码级别，由浅入深的介绍了这个类。

第一重：基本认识

1.1 MapOutputBuffer是什么?

简而言之，MapOutputBuffer是一个缓存Map输出，并局部排序的类。
众所周知，MapReduce从整体上分为Map和Reduce两个阶段，但并不是每个job都必须兼备这两个。比较简单的job可以只有Map阶段，Map的输出直接写入HDFS提供给用户使用。
Reduce阶段，往往意味着数据的重新分配，也就是著名的"shuffle"，“shuffle”阶段是根据Map阶段输出的Key，计算Partition，并交给对应Reduce的过程。在进入“shuffle”阶段之前，我们需要对数据进行初步的排序，并按照partion分组，MapOutputBuffer负责这一过程的。
需要注意的是，这个类只有当job中有Reduce逻辑时才会被调用，因为不需要Reduce，也就没有数据的重新分配，也就没有partition的计算，也就没有排序。

1.2 为什么需要MapOutputBuffer?

从类名上我们可以猜到，这是一个Buffer，buffer的作用一般是为了缓存，那问题变成了为什么要作缓存？难道是仅仅为了累积数据加速写入？我认为不全是。Reduce在拉取数据时，是根据partition的。Map的输入顺序和输出顺序是没有关系的，直接输出的数据是完全乱序的，那拉取时就得把Map输出的数据全部扫一遍，只拿自己partition的，这非常的费时费力。所以设计者想到在写出的时候能不能保证局部有序呢，每次缓存一小部分数据，到一定量的时候，对他们按照partition排序，成块的写入到文件里，拉取的时候方便多了，只要看看哪个partition在哪个块。
另一个原因是Combiner，我们知道这个模块是为了减少Shuffle数据量，而在map端局部的执行Reduce逻辑。要想达到这种效果，同样需要缓存和排序，将一个partition的小部分数据排到一起，调用一下Combiner再输出。

1.3 为什么要理解这个类?

首先，几乎每个Job都会自定义自己的partitioner，深刻理解其是怎样作用于数据，可以更好的设计。另外，MR框架提供了很多参数用来调整缓存和排序的行为，因为它深刻的影响了job的性能，更深入的理解有助于懂得如果参数调优。
还有，很多job的运行时错误是这个阶段会抛出的，理解的深刻，可以快速定位问题。

第二重：详细理解运行过程

本节详细介绍MapOutputBuffer的运行过程，但只是定性和图解，不会给出源码级别的细节。

2.1 过程简介

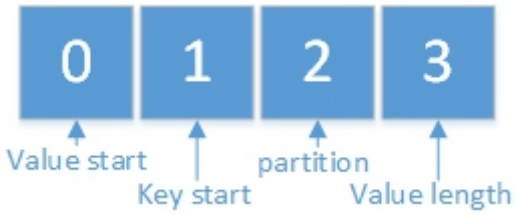
MapOutputBuffer类里维护了一个环形缓冲区，每个输出的record都以key value的连同partition信息写入到这个缓冲区中，当缓冲区快满的时候，有一个后台的守护进程负责对数据排序，并写入磁盘。

2.2 主要变量定义

实现一个带meta信息的缓冲区所需变量，有点多，一下记不住没关系，统一列出为了方便查阅。

1. kvbuffer:环形缓冲区，字节数组，用于存储raw数据和meta信息。kvbuffer.length代表数组总长度

2. kvmeta:缓冲区的字节视角，并没有占据新的内存，主要方便插入meta信息。每个meta信息由4个int组成，分布如下



一个metadata的长度是METASIZE=16

3. equator:缓冲区的中界点，raw的key value数据和meta信息分别从中界点往两侧填充。

4. kvindex:下次要插入的meta信息的起始位置

5. kvstart:溢写时meta数据的起始位置

6. kvend:溢写时meta数据的结束位置

7. bufindex:raw数据的结束位置

8. bufstart:溢写时raw数据的起始位置

9. bufend:溢写时raw数据的结束位置

10. spillper:当数据占用超过这个比例，会造成溢写，由配置“mapreduce.map.sort.spill.percent”指定，默认值是0.8

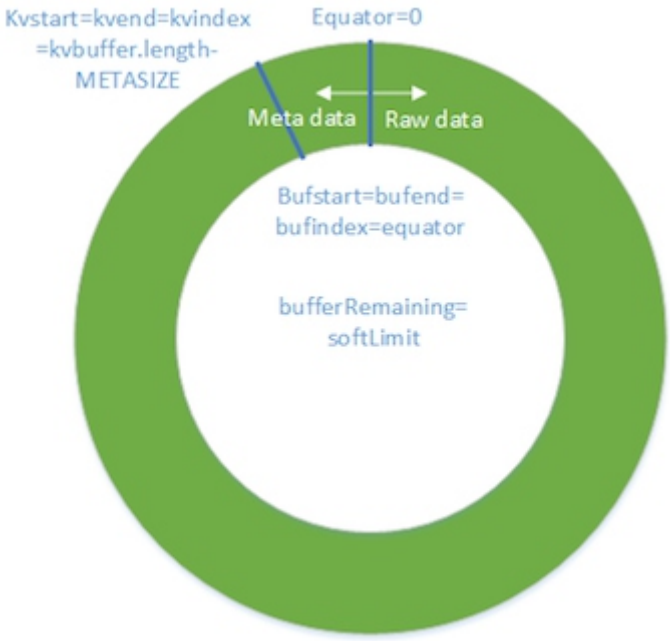
11. sortmb:kvbuffer占用的内存总量，单位是M，由配置“mapreduce.task.index.cache.limit.bytes”指定，默认值是100

12. indexCacheMemoryLimit:存放溢写文件信息的缓存大小，由参数“mapreduce.task.index.cache.limit.bytes”指定，单位是byte，默认值是1024*1024（1M)

13. bufferRemaining:buffer剩余空间，字节为单位

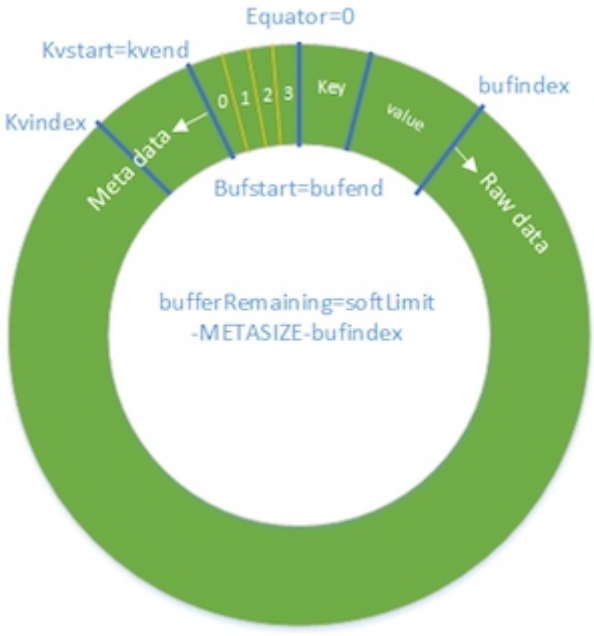
14. softLimit:字节单位的溢写阈值，超过之后需要写磁盘，值等于sortmb*spillper

2.3 初始状态



初始时，equator位于0处，在写入数据时，raw data往下标增大的方向延伸，meta data沿着下标减小的方向扩展，如果按照环形缓冲区来看待，raw data永远是顺时针扩展的，meta data是逆时针扩展的。各个量的初始化情况也很清楚，bufstart，bufend，bufindex都位于起点0处，kvstart, kvend, kvindex沿逆时针方向推进4个int的大小，因为他们标记的是meta data开始要放置的位置。另一个重要变量，标记剩余空间的bufferRemaining因为开始时buffer为空，所以其值等于softLimit。

2.4 第一组<key, value>对放入

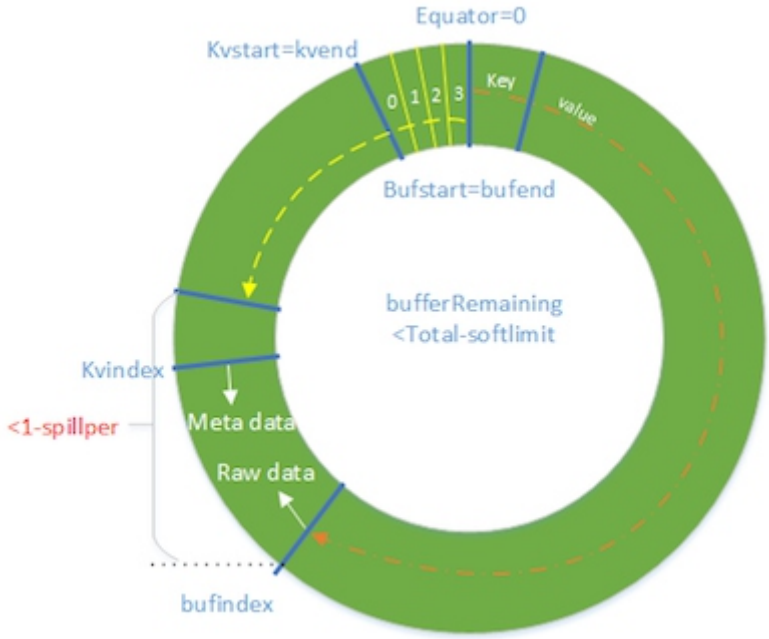


在做了空间检查之后（假设第一组空间一定足够，实际上不一定，第三重时会介绍），首先放入key，在bufindex的基础上累加key的字节，更新bufindex，再放入value，同样累加bufindex。接下来放meta data，从kvindex的基础上，按照定义好meta格式，加一个int放value的start的下标，加两个int放key start的下标，加三个放数据属于哪个partition，加四个放value得长度，由此meta可以准确定位到刚才放入的一对键值。

注意kvstart, kvend, bufstart, bufend的位置并没有更新，也就代表了在后续放入的过程中也不会更

不要忘记要从bufferRemaining中减掉相应的空间。

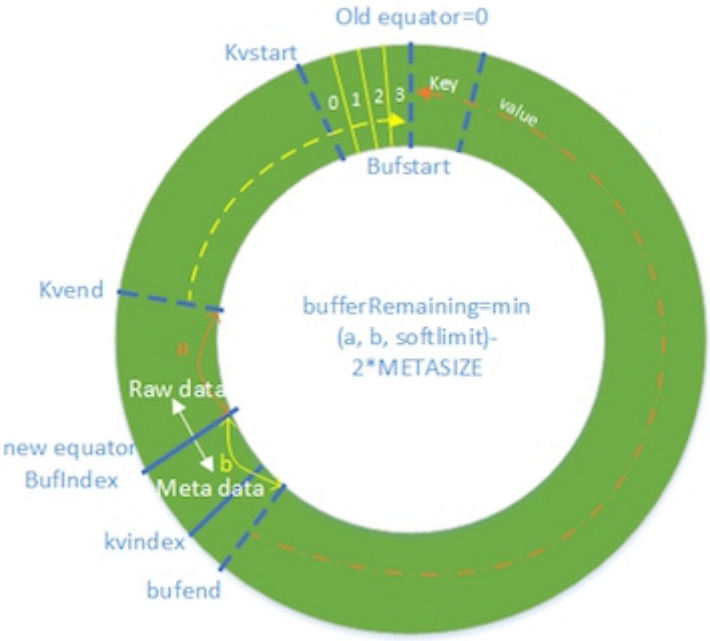
2.5 第一次达到spill 阈值



随着数据的不断写入，kvIndex不断减少，bufIndex不断增加，bufferRemaining不断减小，当bufferRemaining空间不足时，就该触发溢写线程了。由(kvindex—4int)顺时针到到bufindex是已经被使用过的部分，由(kvindex+4int)逆时针到bufindex的空间是未使用的部分。

溢写的触发在写入meta数据和raw数据时都有可能发生。

2.6 溢写与更新



溢写开始之前，首先会更新kvend和bufend的位置，kvend等于kvindex+4int，因为每次kvindex总是标示下次要插入的位置，回退4int才是最后meta data写入的位置，kvstart不变，标志第一个meta data写入的位置。bufend是最后一条记录value结束的位置，bufstart标志第一个key开始的位置。kvstart与kvend之间是全部meta data，bufstart与bufend之间是raw data。

溢写过程是守护线程完成的，在溢写过程中正常的写入缓冲区还可以继续。这也就是spillper的用处，1-spiller就近似代表了溢写触发时还可以使用的比例。

写入需要确定新的equator，新的equator一定位于kvend逆时针到bufend之间，也就是在空余空间的某个位置。我们期望写入过程尽可能的不要受溢写过程的影响，也就是写入的时候要利用当前的空余空间，而不应该侵入已经有数据的空间，新的equator要尽可能均匀的切分空闲空间，均匀的标准是要能放入meta+key+value组合。

确定了equator就可以更新kvindex和bufindex的位置了，meta数据与raw数据的扩展方向是不变的，kvindex还是在下一个要插入meta的起始位置。

bufferRemaining也会相应更新。剩下的可写入空间不应该是所有的空余空间，而是equator到kvend，bufend之间空间比较小的那个。任何一个用完，都代表不能继续写入了。

第三重：源码级详细分析

经过第二重境界，你一定对数据写入与溢出的过程有了了解，如果想继续深入掌握所有细节，请往下看。

3.1 一个buffer,两种视角

从前面的描述可以知道，raw数据和meta数据都是操作在一个buffer上得，但raw数据是以字节流写入的，meta数据的每次插入都是int类型的，这怎么做到的?

```
kvbuffer = new byte[maxMemUsage];// save both the raw value and meta data
bufvoid = kvbuffer.length;
kvmeta = ByteBuffer.wrap(kvbuffer)
    .order(ByteOrder.nativeOrder())
    .asIntBuffer();//int scope of bytebuffer, didn't create a new array
```

ByteBuffer是一个抽象类，封装了byte数组之上的getXXX,putXXX等方法，可以把byte数组当成其他类型的数组使用。wrap方法是将byte数组与最终要返回的数组绑定起来，order方法设定字节是“大端”存放还是“小端”存放。ByteOrder.nativeOrder()返回的是跟机器硬件一致的结果，测试硬件是哪种放法最终用到了sun.misc.Unsafe类，大概就是放一个long（0x0102030405060708L）进内存，再取最低位的那个byte出来，如果是“0x01",就是大端法，如果是"0x08",就是小端。asIntBuffer是个抽象方法，交给子类实现的，就不展开了。

3.2 控制排序行为的一些参数的初始化

高端用户可以通过调节下列这些参数，优化排序行为。

mapreduce.map.sort.spill.percent

简介：溢写的阈值，一个大于0，小于等于1的float。当buffer中已写数据比例超过这个，会启动后台的溢写线程。
默认值：0.8

mapreduce.task.io.sort.mb

简介：用于排序过程的内存总大小，单位是M。
默认值：100

mapreduce.task.index.cache.limit.bytes

简介：存放spill文件索引文件的内存大小。索引文件超出此限制之后，后面的索引文件会跟spill文件本身一起写磁盘。

默认值：1M

map.sort.class

简介：用户可以自定义用于排序的函数。跟其他组件一样，也可用反射构造出来。需要实现IndexedSorter接口。
默认值：org.apache.hadoop.util.QuickSort

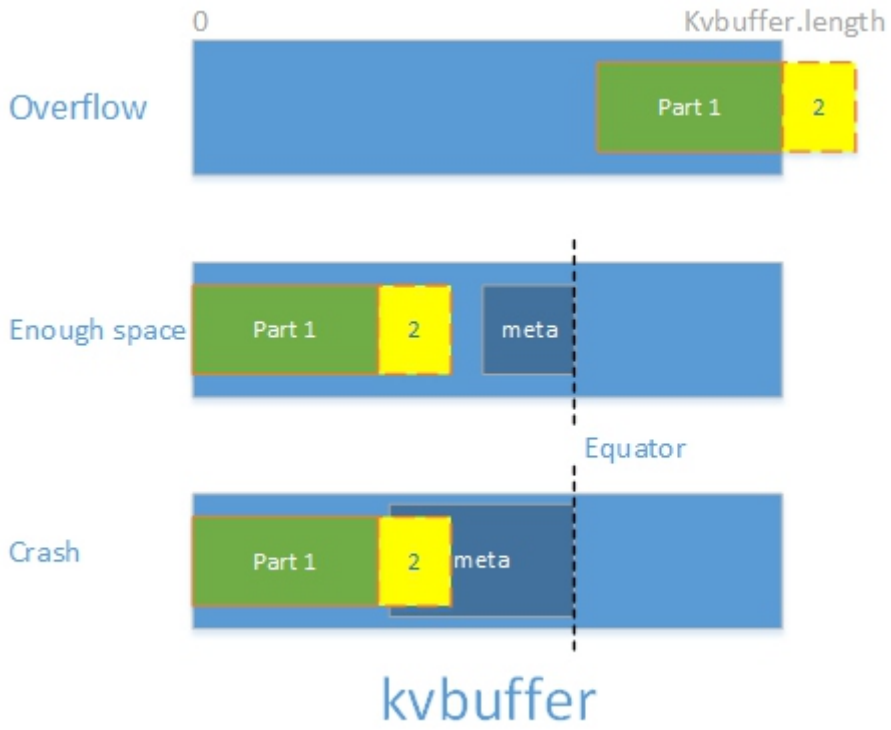
3.3 封装序列化的数据结构

集群区别于单机的重要一点是，集群中的机器常常需要交换数据。任何网络传输的数据都需要可序列化。mapper的输出是会通过网络shuffle给reducer的，因此要求数据可序列化。
序列化操作交由”Serializer"对象完成。序列化之前需要指定数据格式，并与一个输出流绑定。如下：

```
private Serializer<K> keySerializer;
private Serializer<V> valSerializer;
...
final BlockingBuffer bb = new BlockingBuffer();
...
serializationFactory = new SerializationFactory(job);
keySerializer = serializationFactory.getSerializer(keyClass);
keySerializer.open(bb);
valSerializer = serializationFactory.getSerializer(valClass);
valSerializer.open(bb);
...
keySerializer.serialize(key);
...
valSerializer.serialize(value);
```

Serializer对象的“serialize"方法最终是由绑定的OutputStream的write方法完成的。OutputStream在这里有两个实现类，一个是上面列出的BlockingBuffer，另一个是Buffer，两者都是内部类。Buffer实现了write方法，而BlockingBuffer是Buffer的进一步封装。
Buffer的write方法，首先会判断有没有足够的空间存放raw数据，空间不足的情况我们后面再讨论，现在确定空间足够，写入的代码。

```
// here, we know that we have sufficient space to write
if (bufindex + len > bufvoid) {
    final int gaplen = bufvoid - bufindex;
    System.arraycopy(b, off, kvbuffer, bufindex, gaplen);
    len -= gaplen;
    off += gaplen;
    bufindex = 0;
}
System.arraycopy(b, off, kvbuffer, bufindex, len);
bufindex += len;
```



虽然在逻辑上，kvbuffer是个环状缓冲区，但是物理上只是内存中的一段连续空间。len代表的是待插入数据的长度，bufindex是起点，bufindex+len是终点，如果终点超过buffer的长度（bufvoid），我们就应该将多余部分移到buffer开始的部分。

每次将buffer中数据spill到本地磁盘时，会按照raw数据的key进行排序。输入到RawComparator中的key要求是内存中的一段连续空间，而上面写入的逻辑，有可能将key一段写在buffer尾部，另一部分写在buffer头部（如上图Overflow）。BlockingBuffer在Buffer的基础上封装了调整key放置的操作，即将整个key移到起始位置。

放入之前，从起始位置到meta边缘的部分是空闲的。这里有两种情况，如果空余空间足够放入key（如上图的Enough space），可以直接拷贝过去，如果空间不足(如上图的Crash)，就得触发溢写，腾一些空间出来。

我在代码里加了详细的注释。

```
protected void shiftBufferedKey() throws IOException {
    // spillLock unnecessary; both kvend and kvindex are current
    //尾部长度
    int headbytelen = bufvoid - bufmark;
    bufvoid = bufmark;//重置bufvoid
    final int kvbidx = 4 * kvindex;
    final int kvbend = 4 * kvend;
    final int avail =//从头部到meta数据的空间
          Math.min(distanceTo(0, kvbidx), distanceTo(0, kvbend));
    if (bufindex + headbytelen < avail) {//头部开始的剩余空间足够放下整个key          //原来在头部的那部分key拷贝到从headbytelen开始的位
置
       System.arraycopy(kvbuffer, 0, kvbuffer, headbytelen, bufindex);
       //原来在尾部的那部分key拷贝到从0开始的位置
       System.arraycopy(kvbuffer, bufvoid, kvbuffer, 0, headbytelen);
       bufindex += headbytelen;
       bufferRemaining -= kvbuffer.length - bufvoid;
    } else {//头部剩余空间不够
       byte[] keytmp = new byte[bufindex];
       System.arraycopy(kvbuffer, 0, keytmp, 0, bufindex);//暂存头部那部分key
       bufindex = 0;
       //调用Buffer类的write方法，会处理空间不够的情况
       out.write(kvbuffer, bufmark, headbytelen);//先从0开始拷贝尾部key
       out.write(keytmp);//接上暂存的头部那部分key
    }
}
```

3.4 溢写线程的同步

溢写由SpillThread作为一个守护线程完成。

3.4.1 同步控制综述

Java的object类有三个与线程同步相关的方法，即wait,notify,notifyall，方法为final，任何java中的对象都继承这三个方法，且不能重写。三个方法都必须运行在针对某个对象的synchronize块中。wait的语义是当前线程等待，要求线程必须拥有对象的锁，也就是暂时交出锁，等待再次获得。notify的语义是唤醒某个正在等待对象锁的线程，具体哪个不被保证，也就是释放锁。notifyall的语义是唤醒所有等待对象锁的线程。

我认为直接使用object类的方法有两个问题，首先，加锁对象同时肩负同步控制和业务逻辑，代码可读性太差，其次，当多个线程等待唤醒，行为不受程序控制，粒度太粗。

好在java1.5之后引入了“Condition”。此对象的await和signal方法对应于object的wait和notify。Condition由Lock创建，与Lock绑定，需要运行于“lock.lock"与”lock.unlock"之间。一个lock可以 有多个condition，就像在锁上有状态队列，可以精确控制唤醒哪个状态。ReentrantLock是Lock的一 个实现，spillThread正是由这种锁控制的。
初始化时，有下列语句：

```
boolean spillInProgress;
...
final ReentrantLock spillLock = new ReentrantLock();
final Condition spillDone = spillLock.newCondition();
final Condition spillReady = spillLock.newCondition();
volatile boolean spillThreadRunning = false;
final SpillThread spillThread = new SpillThread();
```

spillLock对象有两种Condition，spillDone和spillReady。需要同步的两个线程是主线程和spillThread。另外有两个boolean量表示状态，spillThreadRunning是volatile的，保证同时对两个线程看见。

3.4.2 spillDone

spillDone是标志一次spill过程完成的Condition，sillDone.await的语义是，阻塞当前线程，暂时释放锁，直到spill完成后重新获得锁再返回，spillDone.signal的语义是，一次spill过程完整完成。
spillDone.await在代码中出现了三次，spillDone.signal，第一次是在init方法启动spillThread的时候，等待spillThread进入循环，第二次是在Buffer.write方法中，空间严重不足，需要等待完全spill完成才可以继续，第三次是在flush方法中，flush方法在close方法中被调用，负责收尾工作，所以已经要等待spill完成才可以。
spillDone.signal只在代码中出现一次，位置是SpillThread的run方法，每次while循环开始的时候会调用一次，标志上一次spill已经完成。

3.4.3 spillReady

spillReady标志spill的条件已经成熟的Condition，spillReady.await的语义是等待spill的条件，阻塞当前线程，暂时释放锁，spillReady.signal的语义是spill条件已经达到，spillThread应该开始工作。
spillReady.await在代码中出现一次，是SpillThread.run循环开始的地方，等待条件成熟。
spillReady.signal在代码中出现一次，是startSpill中，会触发spillThread开始工作。startSpill是当空

间不足时会被调用。空间不足是通过判断bufferRemaining的。前面提到，初始化时，bufferRemaining是在总空间的基础上乘以一个系数的，每次meta的写入和raw data的写入都会减小bufferRemaining，当这个值小于等于0时，就会触发溢写。溢写触发发生在两个地方，一个是meta数据写入时，一个是raw data写入时。

3.4.4 spillThreadRunning

标记spillThread是否在运行的boolean，初始值为false，在spillThread开始时置为true，退出时置为false。

3.4.5 spillInProgress

标记spillThread是否真的在spill数据的boolean，初始值为false，在startSpill方法中被置为true，与spillDone及spillReady配合使用完成循环等待，在达到溢写条件时，也会根据这个量判断需不需要startSpill。当一次溢写完成后，会被置为false。

3.5 溢写时数据排序的细节

溢写时，会先对所有的key按照partition排序，partition相同的，按照OutputKeyComparator进行排序。每个partition的数据，也叫一个segment，对应一个writer，如果设置了combiner，会作用于一个partition内部的数据。最后，一次合并的信息（segment的起始位置，raw数据的长度，压缩后数据长度）会写入缓存，并在适宜的时候成批的写入文件。

```
private void sortAndSpill() throws IOException, ClassNotFoundException,
        InterruptedException {
    //approximate the length of the output file to be the length of the
    //buffer + header lengths for the partitions
    final long size = (bufend >=bufstart
            ? bufend -bufstart
            : (bufvoid - bufend) + bufstart) +
            partitions * APPROX_HEADER_LENGTH;
    FSDataOutputStream out = null;
    try {
      // create spill file
      final SpillRecord spillRec = new SpillRecord(partitions);
```

```
final Path filename =
      mapOutputFile.getSpillFileForWrite(numSpills, size);
out = rfs.create(filename);

final int mstart = kvend / NMETA;
final int mend = 1 + // kvend is a valid record
      (kvstart >=kvend
          ?kvstart
          : kvmeta.capacity() + kvstart) / NMETA;
//对meta key进行排序，先按照partition，partition内部根据OutputKeyComparator
sorter.sort(MapOutputBuffer.this, mstart, mend, reporter);
int spindex = mstart;
final IndexRecord rec = new IndexRecord();
final InMemValBytes value = new InMemValBytes();
//循环遍历各个partition
for (int i = 0; i < partitions; ++i) {
  IFile.Writer<K, V> writer = null;
  try {
    long segmentStart = out.getPos();
    //对应此次循环的写入
    writer = new Writer<K, V>(job, out, keyClass, valClass, codec,
        spilledRecordsCounter);
    if (combinerRunner == null) {
      // spill directly
      DataInputBuffer key = new DataInputBuffer();
      while (spindex < mend &&
          kvmeta.get(offsetFor(spindex % maxRec) + PARTITION) == i) {
        final int kvoff = offsetFor(spindex % maxRec);
        int keystart = kvmeta.get(kvoff + KEYSTART);
        int valstart = kvmeta.get(kvoff + VALSTART);
        key.reset(kvbuffer, keystart, valstart - keystart);
        getVBytesForOffset(kvoff, value);
        writer.append(key, value);
        ++spindex;
      }
    } else {
      int spstart = spindex;
      while (spindex < mend &&
          kvmeta.get(offsetFor(spindex % maxRec)
```

```
                        + PARTITION) == i) {
                    ++spindex;
                }
                // Note: we would like to avoid the combiner if we've fewer
                // than some threshold of records for a partition
                if (spstart != spindex) {
                    combineCollector.setWriter(writer);
                    RawKeyValueIterator kvIter =
                            new MRResultIterator(spstart, spindex);
                    combinerRunner.combine(kvIter, combineCollector);
                }
            }

            // close the writer
            writer.close();

            // record offsets
            rec.startOffset = segmentStart;
            rec.rawLength = writer.getRawLength();
            rec.partLength = writer.getCompressedLength();
            spillRec.putIndex(rec, i);

            writer = null;
        } finally {
            if (null != writer) writer.close();
        }
    }

    if (totalIndexCacheMemory >= indexCacheMemoryLimit) {
        // create spill index file
        Path indexFilename =
                mapOutputFile.getSpillIndexFileForWrite(numSpills,partitions
                        * MAP_OUTPUT_INDEX_RECORD_LENGTH);
        spillRec.writeToFile(indexFilename, job);
    } else {//加入缓存
        indexCacheList.add(spillRec);
        totalIndexCacheMemory +=
                spillRec.size() * MAP_OUTPUT_INDEX_RECORD_LENGTH;
    }
```

```
        LOG.info("Finished spill " + numSpills);
        ++numSpills;
      } finally {
        if (out != null) out.close();
      }
    }
```

3.6 收尾工作

收尾工作是由Buffer.flush这个函数完成的。主要有两点，首先，要保证spillThread将buffer中所有的数据都写入磁盘，并安全退出，其次，要将所有spill溢写的小文件合并成一个大得文件，作为mapper最后的输出。

安全退出时，首先会等待借助spillDone这个condition，等待spillThread完成最后一次spill，然后根据buffer的状态量（kvend,kvindex等）判断buffer是否已经spill完全，如果有剩余数据，会最后调用一次"sortAndSpill"，注意这次spill是主线程完成的。spill完成后，调用

```
spillThread.interrupt();
spillThread.join();
```

杀掉spillThread。

将小文件合并成大文件是在函数mergeParts中完成的，代码稍微有点长，但过程很清楚。每个mapper最终只输出一个数据文件finalOutputFile和一个与之对应的index文件finalIndexFile。

我们知道，mapper时段用的partition，实际上与reducer对应，reducer会拉取属于自己的那段partition，因此，finalOutputFile中，数据是按照partition来存放的，每个finalIndexFile文件中的项，就指向了finalOutputFile中的一个partition，存储了它的起始位置，压缩前长度和压缩后长度。

回想一下，我们在输出每次spill的小文件时，也是按照partition存放的，并且也有一个index文件，因此，最后合并的时候有一个二重循环，外层循环遍历partition，内层循环遍历spill file，每次把属于一个partition的那些小segment取出来，使用归并排序合并到一起，写入最终的文件，并记录index即与输出小文件时相同，最终这次合并，也可选的会调用combiner。

结语

这篇博客由浅入深的介绍了MapOutputBuffer在整个框架中起到的作用，并详细描述了其对数据缓存和排序的过程。

参考文献

[1] MapOutputBuffer, http://grepcode.com/file/repo1.maven.org/maven2/org.apache.servicemix.bundles/org.apache.servicemix.bundles.hadoop-client/2.4.0_1/org/apache/hadoop/mapred/MapTask.java#MapTask.MapOutputBuffer