Classic MR与NativeTask源码解析

———解决NativeTask与Hive的兼容问题

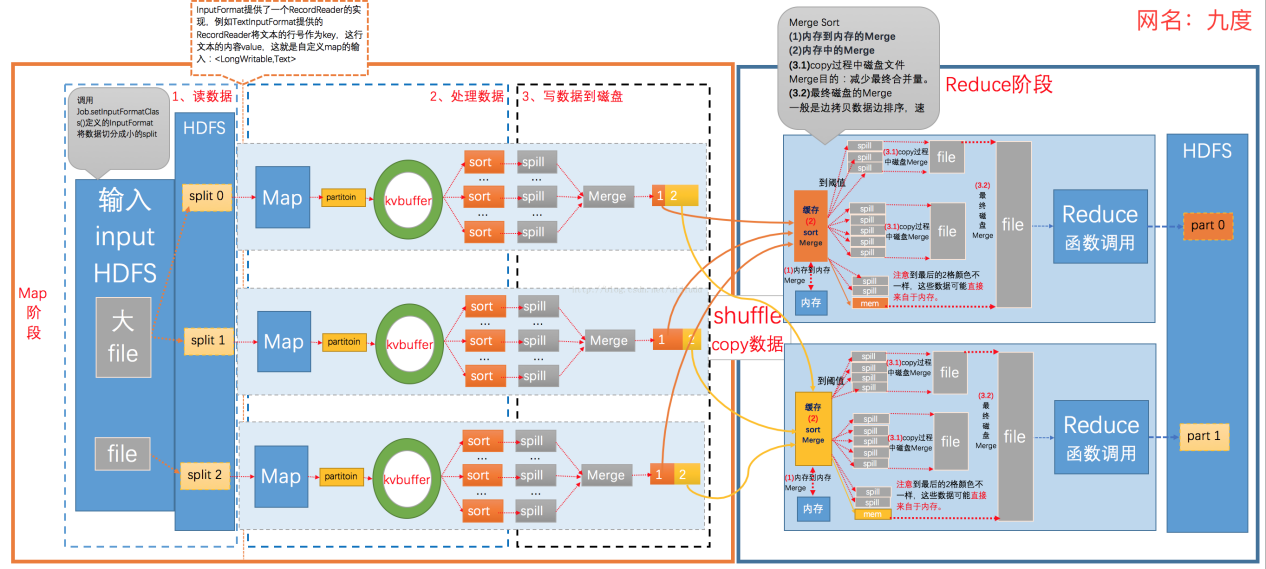
MR是Hadoop的计算框架，用于大规模数据集的并行计算，其中核心的概念是映射（Map）和规约（Reduce），Map把一组键值对映射成一组新的键值对，指定并发的Reduce函数，用于保证所有映射的键值对中每个共享相同的键组，Reduce对Map的输出结果进行合并。NativeTask是Hadoop MapReduce到的高效执行引擎实现，其替换原MR框架中的Map阶段部分模块。

# 1、MR执行流程

MapReduce的执行流程如下图所示：



Map的执行结果先放到内存中，内存不够时，通过spill过程持久化到硬盘中，在spill之前要对中间结果进行排序：partition->sort->spill



## 过程1：Map数据输入及切片

Map任务提交到Yarn后，被ApplicationMaster启动，任务的形式是YarnChild，在其中会执行MapTask的run方法。在提交之前，JobClient会对数据源进行切片，切片信息由InputSplit对象封装，接口定义如下：

1. public interface InputFormat<K, V> {
2. InputSplit[] getSplits(JobConf job, int numSplits) throws IOException;
3. RecordReader<K, V> getRecordReader(InputSplit split,JobConf job,Reporter reporter) throws IOException;
4. }

JobClient通过getSplits方法来计算切片信息，切片默认大小和HDFS的块大小相同。JobSplitWriter将Splits信息（SplitMetaInfo），写入任务执行目录的文件中，SplitMetaInfo保存了该Split的数据大小及数据所在的位置。

数据的切片大致流程（FileInputFormat#getSplits），计算输入文件的大小，根据分片数（默认HDFS块大小，也可以配置Map数目）计算每个分片的大小，获取每个分片的BlockLocation，将位置信息写入SplitsMetaInfo中。

MapTask.run，根据splitIndex从SplitsMetaInfo文件中获取对应的InputSplit信息，然后创建RecordReader，从数据文件中获取Key及Value。然后调用Mapper.run，执行定义的操作。

## 过程2：Map数据处理

Mapper.run的接口源码如下所示：

1. public void run(Context context) throws IOException, InterruptedException {
2. setup(context);
3. try {
4. while (context.nextKeyValue()) {
5. map(context.getCurrentKey(), context.getCurrentValue(), context);
6. }
7. } finally {
8. cleanup(context);
9. }
10. }

map方法，会调用MapOutputCollector.collect(Key,Value)，输出到内存中。

## 过程3：Map Sort及Spill

Mapper.run在执行过程中，将处理结果写到内存缓冲区中，当缓冲区被使用完后，需要spill到磁盘中，过程如下图所示：



MapOutputCollector#collect，将<Key,Value>序列化后，存储到内存缓冲区kvBuffer中，kvBuffer是内存中的一个环形数据结构，本质上是一个字节数组。在Collector，定义记录Record<key,value>在kvBuffer中的索引kvmeta，包括value的起始位置、key的起始位置、partition值及value的长度。

<Key,Value>数据区域和索引数据区域在kvbuffer中是相邻不重叠的两个区域，用一个分界点来划分两者，而分割点是变化的，每次spill之后会更新一次，<key,value>数据的存储方向是向上增长，索引数据的存储方向是向下增长，kvBuffer的使用，如下图所示：



在写入kvBuffer之前，要对<K,V>进行序列化，在MapOutputCollector#collect中完成，源码如下：

1. int keystart = bufindex;
2. keySerializer.serialize(key);
3. if (bufindex < keystart) {
4. // wrapped the key; must make contiguous
5. bb.shiftBufferedKey();
6. keystart = 0;
7. }
8. // serialize value bytes into buffer
9. final int valstart = bufindex;
10. valSerializer.serialize(value);

kvindex和bufindex之间的数据是未被spill的数据，如果这部分数据所占用的空间大于等于spill的指定百分比，则开始调用startSpill方法进行溢写。

在spill到磁盘之前，要进行排序操作，代码执行过程分析如下所示：



Sorter.sort对kvBuffer中的数据进行排序。

kvBuffer中的数据，由IFile.Writer写入磁盘，源码如下：

1. while (spindex < mend &&
2. kvmeta.get(offsetFor(spindex % maxRec) + PARTITION) == i) {
3. final int kvoff = offsetFor(spindex % maxRec);
4. int keystart = kvmeta.get(kvoff + KEYSTART);
5. int valstart = kvmeta.get(kvoff + VALSTART);
6. key.reset(kvbuffer, keystart, valstart - keystart);
7. getVBytesForOffset(kvoff, value);
8. writer.append(key, value);
9. ++spindex;
10. }

map执行结束后，spill到磁盘的文件一般为多个，需要对文件进行merge操作，操作的过程中会对文件中的数据进行归并。



这个过程结束后，每个Map会生成一个MapOutputFile，Map结束后通知AM，启动Reduce任务。

## 过程4：Reduce Shuffle

Map结束后，Reduce启动shuffle阶段，从Map输出文件中获取数据，但是Map一般多个，某个Reduce从多个Map输出文件中获取其所对应的处理数据，这些数据需要merge（同时进行排序）。Reduce分为copy->sort->reduce三个阶段，过程如下所示：

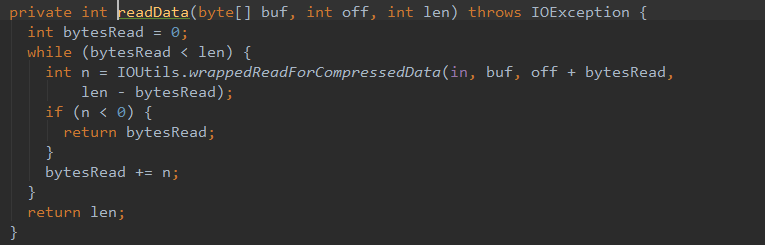


ReduceTask启动Shuffle线程，并初始化MergerManager(用于数据的merge)及LocalFetcher，LocalFetcher根据reduceId得到文件位置（用于从Map输出文件读取对应数据），读取数据放入MergerManager的mapOutput中。最后MergerManager对这些数据进行merge操作（也就是sort阶段，归并排序），Merger首先使用ImMermoryMapOut，当内存不够时，使用内存到磁盘的merge操作。最后进行Reduce阶段的数据处理。数据的读取过程涉及到硬盘及磁盘等（imMermoryMerger,onDiskMerger,memToMemMerger）：

根据文件生成finalSegments，createInMemorySegments及diskSegments等。

根据finalSegments，生成MergeQueue（RawKeyValueIterator）

生成key及value，由Reader.nextRawKey来获取，Reader有数据流InputStream。数据的获取readData：



读取的过程中，两个变量很重要，分别是currentKeyLength及currentValueLength。IFile.Reader#positionToNextRecord中进行赋值；

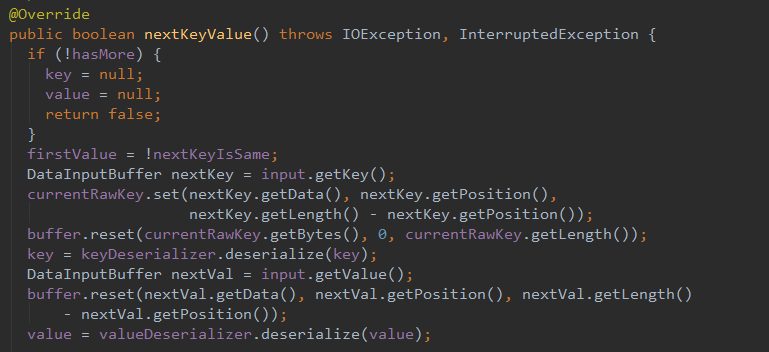
// Read key and value lengths

currentKeyLength = WritableUtils.readVInt(dIn);

currentValueLength = WritableUtils.readVInt(dIn);

## 过程5：Reduce数据处理

在上一步的Shuffle获取<Key,Value>的RawKeyValueIterator，在使用之前要进行反序列化，核心方法是ReduceContextImpl#nextKeyValue:



反序列化后，数据交由Reduce来处理，Reducer源码的接口，如下所示：

1. while (context.nextKey()) {
2. reduce(context.getCurrentKey(), context.getValues(), context);
3. // If a back up store is used, reset it
4. Iterator<VALUEIN> iter = context.getValues().iterator();
5. if(iter instanceof ReduceContext.ValueIterator) {
6. ((ReduceContext.ValueIterator<VALUEIN>)iter).resetBackupStore(); }}

reduce方法中使用ReduceContex#write写入到输出文件中。

# 2、NativeTask MR

NativeTask MR与经典MR的执行流程是相同的，有差别的地方是步骤3，对缓存中的数据处理过程。NativeTask实现了新类型的MapOutputCollector(NativeMapOutputCollectorDe-

legator)，过程如下图所示：



其中存放数据的数据结构为PartitionBucket，MemoryBlock及KVBuffer，其中MemoryPool为系统io.sort.mb分配的内存空间：



NativeTask与经典MR相比，优点在于NativeTask避免序列化和数据拷贝，基本接口为传递内存数据引用的形式，尽量避免了数据拷贝。NativeTask使用C/C++实现，保证轻量级压缩在数据处理的各个环节以多种形式出现（经典MR必须通过JNI调用压缩算法）。与压缩类似，使用C/C++，可以使用SSE/AVX优化技术。

# 3、NativeTask与Hive的兼容

Hive Job可以正常运行在经典MR中，根据源码分析，HiveKey继承类BytesWritable，序列化和反序列化，通过MR中WritableSerialier和WritableDeserializer来完成。但是在NativeTask MR中，实现这为每个数据类型，添加了序列化类，但是没有考虑Hivekey及其他外部数据类型，因此在NativeTask MR中不能运行Hive。

为了支持Hive在NativeTask MR中运行，需要完成以下工作：

1. 添加HiveKeyType，修改代码NativeTask.h及WritesUtils.cc

在NativeTask中，每个数据类型的处理不同（例如map阶段spill数据到输出文件中，每个数据类型的key/value长度不同）

1. 根据HiveKeyType，设置HiveKey的compartor，修改NativeObjectFactory.cc#get\_com

-partor

compartor，spill之前的sort过程，Hivekey继承BytesCompartor，因此没有必要在实现HivekeyCompator，分析HiveKey及BytesWritable代码，两个类的comparator相同。

1. 实现HiveKey序列化类，HiveKeySerializer

需要注册到HadoopPlatform中，修改init方法

1. 根据HivekeyType，设置HiveKey作为Key/Value时keylength/keybufflength/valuelength/

valuebufferlength等

涉及的类包括：IFile.cc(spill),CombineHandler