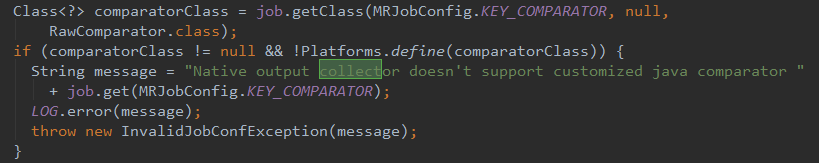
## 分析1：

NativeMapOutputCollectorDelegator.init



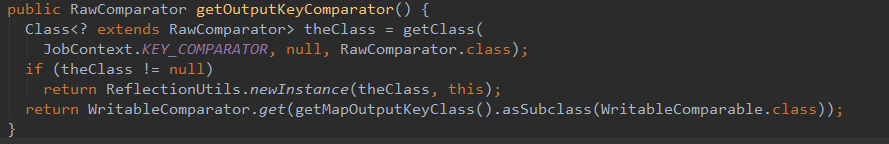
这个地方出的问题。

PlatForms.define(compartorClass)，返回false。HadoopPlatform.define(compartorClass)

Join应该没有设置MRJobConfig.KEY\_COMPARATOR

WordCount中，mapreduce.job.output.key.comparator.class如何设置

JobConf.getOutputKeyComparator根据MRJobConfig.KET\_COMPATOR，获取comparator



MapOutputBuffer#init -> comparator = job.getOutputKeyCompator，

两个MapOutputCollector comparator调用相同的方法。（没有设置的情况下，返回的都是WritableComparator）。而且并没有提示NativeMapOutputCollector#init中抛出Native output Collecotr not supported，应该不是这个问题。

## 分析2：

NativeTask不支持QuickSort之外的排序算法。

## 分析3：

NativeTask不能执行，提示的错误：java.io.IOException:/PartitionBucket.h:56:pool is NULL, or comparator is not set

抛出异常的地方：NativeCollectorOnlyHandler.create(TaskContext)

->CombineHandler.create(TaskContext)

->NativeBatchProcessor.create(NAME, Conf,DataChannel.INOUT)

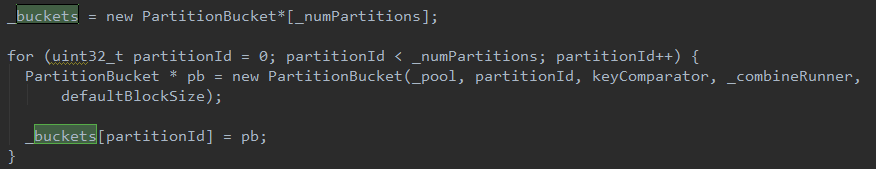
->NativeBatchProcessor.init(JobConf)

->NativeBachProcessor.setHandler(long,byt[][] configs) （这个地方通过JNI，调用C方法）

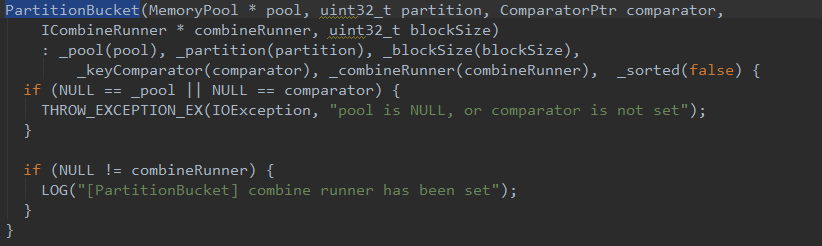
->BatchHandler.cc#setHandler

-> xxx PartitionBucket.h ,pool is NULL,or compartor is not set

PartitionBucket在MapOutputCollector#init中被初始化：



初始化方法：



抛出异常的地方

## 分析4：

核心类是NativeCollectorOnlyHandler，其对应一个C文件，进行JNI调用，来完成Map Output操作

分析5：

NativeTask

MapOutputCollector#get\_comparator

MapTask

MapOutputBuffer#comparator = JobConf.getOutputKeyComparator()

comparator的初始化方法不同

NativeTask中get\_comparator的获取执行流程

NativeBatchProcessor.setHandler

BatchHandler.setHandler -> onSetup -> configure

MCollectorOutputHandler#configure

MapOutputCollecotr#configure

MapOutputCollector::getComparator

native.map.output.key.comparator <= native.map.output.key.comparator

keyClass:

mapreduce.map.output.key.class <= MAPRED\_MAPOUTPUT\_KEY\_CLASS HiveKey

mapreduce.job.output.key.class <=MAPRED\_OUTPUT\_KEY\_CLASS

Text

valueClass：

mapreduce.map.output.value.class <= MAPRED\_MAPOUTPUT\_VALUE\_CLASS

BytesWritable

mapreduce.job.output.value.class <= MAPRED\_OUTPUT\_VALUE\_CLASS

Text

首先根据这三个参数获取comparatorName

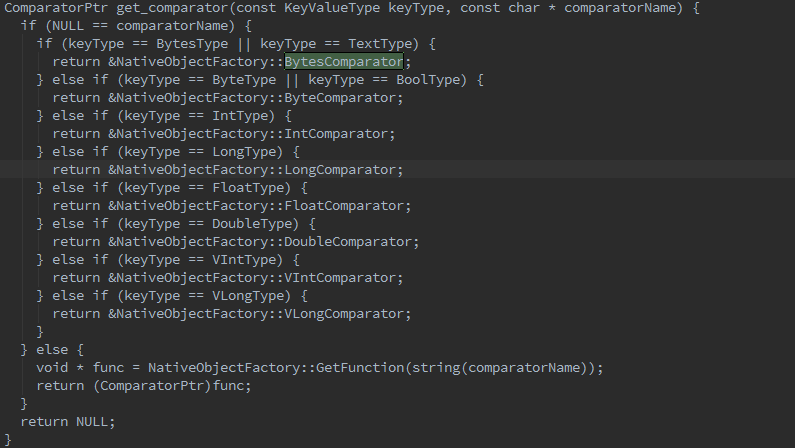
|  |  |
| --- | --- |
| 参数 | 值 |
| mapreduce.map.output.key.class | HiveKey |
| mapreduce.job.output.key.class | Text |
| mapreduce.map.output.value.class | BytesWritable |
| mapreduce.job.output.value.class | Text |

NativeObjectFactory#get\_comparator

MapOutputSpec \_spec:这个参数根据Config进行初始化，

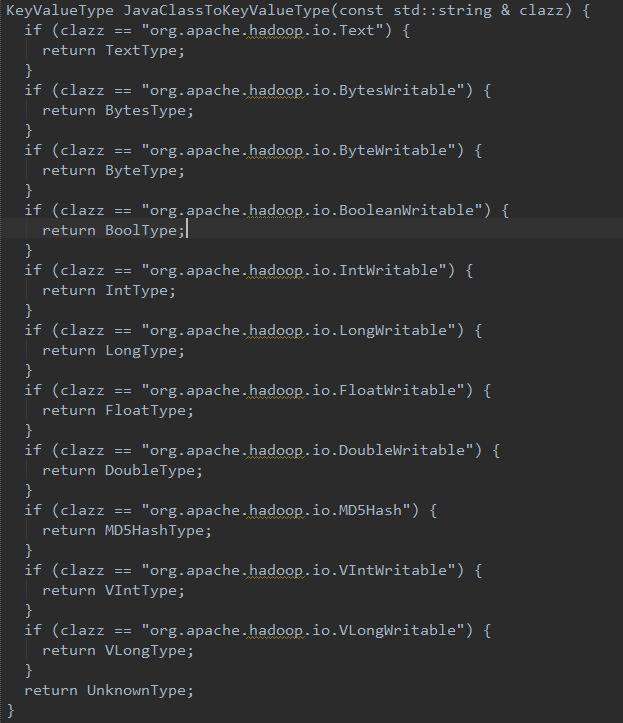
|  |  |
| --- | --- |
| 变量 | 含义 |
| sortAlgorithm | 排序算法（CPPSORT，DUALPIVOTSORT）,mapreduce.sort.avoidance中配置是否排序，可以支持非排序 |
| codec | 压缩方式,Mapreduce.map.output.compress配置，不支持DefaultCodec |
| Key/value | mapred.mapoutput.value.class/mapred.mapoutput.key.class定义  MapOutputSpec#getSpecFromConfig->JavaClassToKeyValueType |

源码如下：

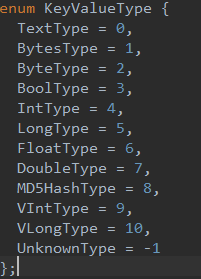


WritableUtils.h

KeyValueType JavaClassToKeyValueType(const std::string & clazz)



NativeTask.h



然后调用NativeTask::get\_comparator(spec.keyType,comparatorName)获取Comparator

NativeObjectFactory#get\_comparator(KeyValueType keyType,const char \* comparatorName)

参数传递：

KeyValueType:

分析5：

经典MR serializer与NativeTask MR的不同，序列化所使用的类

## Hive执行命令:（使用的是old MR API）

select url from hivetest sort by url limit 10;

其中MR和NativeTask所使用的参数对比，如下：

|  |  |  |
| --- | --- | --- |
|  | 经典MR | NativeTask |
| Mapoutput keyClass | HiveKey | HiveKey |
| Mapoutput valueClass | BytesWritable | BytesWritable |
| Map keySerializer | WritableSerializer | BytesWritableSerializer |
| Map valueSerializer | WritableSerializer | BytesWritableSerializer |
| Reduce keyDeserializer | WritableDeserializer | WritableDeserializer |
| Reduce valueDeserializer | WritableDeserializer | WritableDeserializer |

最大的区别是序列化对应的类不通过，因此序列化到数据流中是不同的。

## Wordcount执行程序对比：

hadoop jar xx-examples.jar wordcount /input /output

其中的参数对比：

2015-04-14 18:43:17,488 WARN [main] org.apache.hadoop.mapred.ReduceTask: ======= use New Api : true =======

2015-04-14 18:43:17,488 WARN [main] org.apache.hadoop.mapred.ReduceTask: ======= run newReducer =======

使用新的MR API

|  |  |  |
| --- | --- | --- |
|  | 经典MR | NativeTask |
| Mapoutput keyClass | Text | Text |
| Mapoutput valueClass | IntWritable | IntWritable |
| Map keySerializer | WritableSerializer | TextSerializer |
| Map valueSerializer | WritableSerializer | IntWritableSerializer |
| Reduce keyDeserializer | WritableDeserializer | WritableDeserializer |
| Reduce valueDeserializer | WritableDeserializer | WritableDeserializer |

序列化：

|  |  |  |  |
| --- | --- | --- | --- |
| 经典MR |  | NativeTask MR |  |
| WritableSerializer | Key :  Text <length,bytes> | TextSerializer: | <keyLength,valueLength>  Key:  Text:<bytes> |
| WritableSerializer | Value:  IntWritable <int> | IntWritableSerializer: | Value:  IntWritable <int> |

写入文件格式：

|  |  |
| --- | --- |
| 经典MR  IFile.append | NativeTask MR  IFile.cc#append |
| <KeyLength,ValueLength>  <keyData,valueData> | <keyBuffLen,valBuffLen>  <keyLen> <Key>  <valueLen> <Value> |

NativeTask:

Bufferpusher#sendData(.....ByteBuffer) -->

NativeProcessInput.sendData (rawOutputBuffer:ByteBuffer.c) -->

nativeProcessInput (rawOutputBuffer.position:Buffer) -->

BatchHandle#onInputData -->

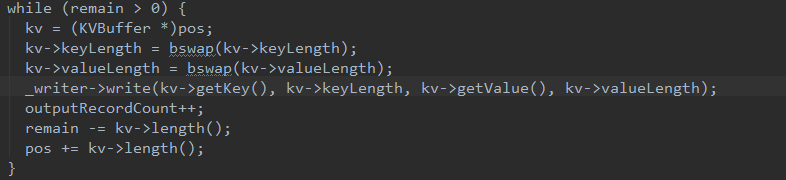
BatchHandler#handlerInput(\_in) :InputBuffer-->

CombineHandler#handleInput(in) -->

\_in:char \* <==> ByteBuffer



CombineHandler#write(char \* buf) -->



KVBuffer <===>char \* buf --> KVBuffer是一个缓存中的KV pair，方便复制及写入文件

IFileWriter#write(char \* key,keylen,\*value,valuelen)，按照一定格式

其中

void IFileWriter::write(const char \* key, uint32\_t keyLen, const char \* value, uint32\_t valueLen) {

// append KeyLength ValueLength KeyBytesLength

uint32\_t keyBuffLen = keyLen;

uint32\_t valBuffLen = valueLen;

switch (\_kType) {

case TextType:

keyBuffLen += WritableUtils::GetVLongSize(keyLen);

break;

case BytesType:

keyBuffLen += 4;

break;

default:

break;

}

switch (\_vType) {

case TextType:

valBuffLen += WritableUtils::GetVLongSize(valueLen);

break;

case BytesType:

valBuffLen += 4;

break;

default:

break;

}

\_appendBuffer.write\_vuint2(keyBuffLen, valBuffLen);

switch (\_kType) {

case TextType:

\_appendBuffer.write\_vuint(keyLen);

break;

case BytesType:

\_appendBuffer.write\_uint32\_be(keyLen);

break;

default:

break;

}

if (keyLen > 0) {

\_appendBuffer.write(key, keyLen);

}

if (NULL != \_recordCounter) {

\_recordCounter->increase();

}

\_recordCount++;

switch (\_vType) {

case TextType:

\_appendBuffer.write\_vuint(valueLen);

break;

case BytesType:

\_appendBuffer.write\_uint32\_be(valueLen);

break;

default:

break;

}

if (valueLen > 0) {

\_appendBuffer.write(value, valueLen);

}

}

需要改的地方是IFileWriter.write方法，添加HiveKeyType的判断，然后写入key/value到的长度。\_kType及\_vType是MapOutputSpec传递，保存在SpilInfo中。要写一个NativeTask及经典MR的数据流图（存储临时数据的数据结构，包括内存及硬盘）。

读取文件：

从文件读取数据，根据文件获取RawKeyValueIterator，为Reduce的数据处理安排。经过Shuffler阶段，从文件获取数据。核心的是Segments，其中包含MapOutputFile对应的Reader。

Classic MR与NativeTask实现相同，在ReduceTask.run，通过Shuffle.run获取RawKey

ValueInterator：

|  |  |
| --- | --- |
| 变量名 | 含义 |
| reader:Reader/IFile内部类 | 从文件流中读取数据，其中封装MapoutputFile对应的数据流，包括读取keyLength,ValueLength,RawKey,RawValue |
| Key:DataInputBuffer | 当前key的二进制流 |
| Codec:CompressionCodec | 压缩方式 |

反序列化：

## bbp :Example

|  |  |  |
| --- | --- | --- |
|  | 经典MR | NativeTask |
| Mapoutput keyClass | LongWritable | LongWritable |
| Mapoutput valueClass | BytesWritable | BytesWritable |
| Map keySerializer | WritableSerializer | LongWritableSerializer |
| Map valueSerializer | WritableSerializer | BytesWritableSerializer |
| Reduce keyDeserializer | WritableDeserializer | WritableDeserializer |
| Reduce valueDeserializer | WritableDeserializer | WritableDeserializer |