Hadoop NativeTask与压缩算法

NativeTask是Hadoop MapReduce的高效执行引擎实现。与MapReduce相比，NativeTask获得了不错的性能提升，主要包括更好的排序实现、关键路径避免序列化、避免复杂抽象、更好的利用压缩等。

NativeTask是一个高性能MapReduce执行单元，支持C++接口。顾名思义，NativeTask是一个本地数据处理引擎，专注于数据处理本身，在MapReduce的环境下，它仅替换Task模块功能。换句话说，NativeTask并不关心资源管理、作业调度和容错，这些功能仍旧由原有的模块完成，而实际的数据处理由这个高性能处理引擎完成。任务级别的数据处理占用Hadoop集群绝大部分资源，而利用NativeTask的高效性能可以显著提高数据分析速度、降低成本。

NativeTask的接口和使用方法与Hadoop Pipes类似，用户通过NativeTask提供的头文件和动态库，编译生成自己的应用库，提交作业到Hadoop集群执行。NativeTask有以下特点。NativeTask有以下特点。

1. 高性能，加速作业执行并节约硬件资源。
2. C++接口，应用可以更方便地使用各种优化技术，例如SSE/AVX指令优化、LLVM动态编译、GPU计算等。
3. 纯二进制接口，避免序列化和反序列化开销。
4. 支持不排序的数据流，经典的MapReduce数据流是需要对记录排序的，但很多应用并不需要排序，通过移除排序，可以进一步提升性能。
5. 新加一种与MapReduce不同的编程模型接口Map-Foldl，能够更高效地执行聚合类的应用。

# 1、MapReduce 过程

## 1.1 执行流程

流程如下图所示：

****

Map端基本流程：

1. 通过InputFormat的getSplits将文件分解成map数量的部分输入文件，InputFormat由Job.setInputFormatClass进行设定，默认使用TextInputFormat，见JobConf.getInputFormat。JobSubmitter.writeSplits方法中进行了文件的分片及createSplitFiles。
2. Job提交后，由RM分配MRappMaster及Map-reduce container后，由LocalContainerLaunc

he.SubtaskRunner启动map及reduce任务（MapTask.run()），在启动过程中，初始化map文件的RecordReader（newTrackingRecordReader），RecordReader将文件分解成Mapper分解的<key,value>记录

1. 根据程序的map方法对数据进行处理，执行的结果先放在内存，通过RecordWriter写入文件。当内存不够时，需要通过spill过程持久化到硬盘里面，在spill之前，需要进行sort过程，首先按照进行分区，然后进行排序，每一个分区是一个reduce过程的输入数据，在map过程接收后，spill到硬盘的文件数，一般是多个，需要进行merge（典型的归并排序）：partition->sort->spill
2. 下面对过程3进行具体的分析，每个map对自己的输出文件进行partition操作，例如10个reduce任务，那么默认的partition操作是对map的输出key进行hash，生成10个文件。由于map很多，因此reduce阶段会从每个map获取属于自己的文件，对这些文件进行合并（边copy,边merge）。

下面仅分析，Map的partition操作，典型的partion是hash方法，分区多少有reduce task的数目来决定。Partitioner 的类由mapreduce.job.partitioner.class来指定。Partioner提供了getPartition(key,value,numpartitions)操作，根据key的值生成分区位置。在HashPartioner使用(key%numpartitions)算法。

1. map生成数据后，根据partition结果，调用MapOutputCollector.collect输出到指定文件，以MapTask#MapOutputBuffer为例，类中核心变量如下：

|  |  |  |
| --- | --- | --- |
| 变量 | 类型 | 描述 |
| keySerializer | Serializer | Key的序列化，main方法中会指定key的class |
| valueSerializer | Serializer | Value的序列化，main方法中指定value的class |
| combinerRunner | CombineRunner | 预处理map数据，减少数据传输 |
| codec | CompressionCodec | 对Map outputs输出进行压缩 |
| kvbuffer | Byte[] | Main output buffer(输出缓存) |
| sorter | IndexSorter | 排序算法 |
| bb | BlockingBuffer | 内部类，用于将序列化记录写到磁盘上 |
| SpillThread | SpillThread | 查看输出缓存大小，超出配置，则写到磁盘中 |
| mapoutputFile | MapOutputFile | map输出文件 |

Collector初始化，会初始化key及value的class（job.getMapoutputKeyClass），序列化类keySerializer及valueSerializer，通过SerializationFactory.getSerializer(keyClass)获取。Codec由mapreduce.map.output.compress.codec配置，如果mapreduce.map.output.

compresss配置为false时，不对数据进行压缩。Combiner由mapred.combiner.class指定。其余参数不再介绍。sorter排序算法由map.sort.class配置，默认是QuitSort。

collect过程如下：

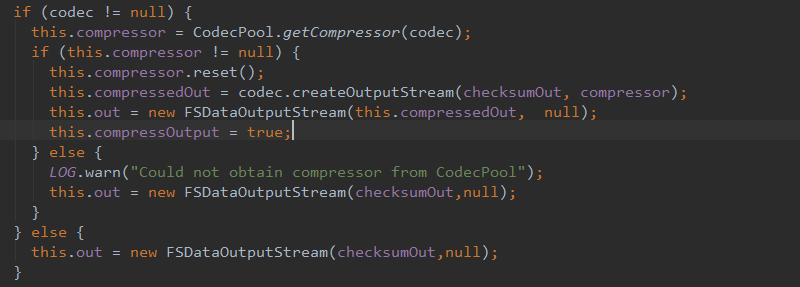
* 判断bufferRemaing <=0，如果是startSpill，将map输出spill到磁盘中
* 否则，序列化key及value，写入kvBuffer中

startSpill由线程SpillThread来完成，通过spillReady.signal（信号量）来启动，其过程为MapOutputBuffer#sortAndSpill，如下所示：

1) 获取文件名，创建文件

2）对kvbuffer中的数据进行排序

3）磁盘的写入，由IFile.Writer来完成，Writer会根据Map的配置初始化Compressor类（压缩算法），在写入磁盘之前要对数据进行压缩，如果配置不使用压缩算法，则不会对数据进行压缩：



4）如果没有配置combine，则直接将排序后的数据写入文件，配置combine，则使用combine类中的方法进行预处理后，再写入文件

6）其中压缩算法的应用，是由codec.createOutputStream创建压缩流，在压缩流中会有变量Compressor，对数据进行压缩。

7）map结束时，调用MapOutputBuffer#flush，对map阶段的输出数据进行最后的处理，将缓存的数据写入磁盘。调用MapOutputBuffer#mergeParts对输出文件（由于spill多次，造成多文件）进行merge操作，进行数据的总体排序（归并排序），并合并成一个文件。

## 1.2 shuffle

当map结束后，map的输出在磁盘上。Reduce获知Map结束的消息后，从Map拉取数据，这涉及到数据的shuffle。这消耗大量的网络资源，磁盘I/O也是影响因素。要提高MapReduce的速度，要减少拉取数据的量及尽量使用内存。

1. LocalContainerLauncher.runSubtask（MRAppMaster中）会判断所有的task是否完成（通过判断完成的map数目来实现），map task完成后，会启动reduce task
2. 拉取数据的组件是ShuffleConsumerPlugin，实现类为Shuffle，在shuffler中负责拉取数据的是Fetcher线程，采用多线程则由mapreduce.reduce.shuffle.parallelcopies来配置线程数目。
3. Fetcher启动后，首先从localMapFiles获取所有的mapFile，然后复制文件，由copyMapOutput(TaskAttemptID)来完成。每个Reduce根据reduceId，定位到该Reduce负责的文件位置，然后从MapHost中复制文件（某段数据）。
4. 在Fetcher启动的过程中，同时会初始化一个MergeManager，用于文件的merge操作，merge有三种形式：内存到内存，内存到磁盘，磁盘到磁盘。一般内存到内存不启用。当内存的数量到达一定阈值，就启动内存到磁盘的merge，当没有map端数据传输时内存到磁盘的merge就完成。然后启动第三种磁盘到磁盘的merge生成最终的reduce输入文件。最终的文件，可能在内存上，也可能在磁盘上，可在调优中进行配置。

## 1.3 MapReduce性能分析

影响Hadoop性能的因素，主要有以下几点：

1. I/O瓶颈，绝大多数的MapReduce作业都属于数据密集型，在不使用轻量压缩的情况下，磁盘和网络I/O很容易成为瓶颈。Snappy和LZ4开源压缩算法已经整合进Hadoop主干版本。对于Hadoop上的常见应用来说，其数据的压缩比是很高的，一般能达到5倍以上，排序或使用列存储后可能会更高，压缩速度约为400M/s, 解压约为1.5G/s, 基本能够将I/O吞吐能力放大5倍。通过与列式存储配合，更高的压缩比也是可能的。在开启轻量压缩之后，CPU会成为Hadoop集群的瓶颈，并且在短期内不会改变，即使目前CPU处理能力的发展要比I/O带宽的发展要快一些。
2. Map阶段排序，在简单Map任务中，排序约消耗整个任务CPU资源的一半。目前Map阶段的排序实现很低效，其原因是它使用一个大缓冲区对所有记录一起排序。另外目前的排序没有针对CPU缓存特性进行优化，一个更好的排序实现大概可以将排序性能提升10倍，目前社区已经发现这个问题，也提出一些改进方案。
3. 序列化和反序列化，序列化会造成大量的对象创建和小数组拷贝，另外也会引入过于复杂的I/O流抽象和低效的对象比较实现，而这些操作几乎贯穿整个数据处理流。这个问题很早之前也在社区提出过，后来不了了之。个人认为在MapReduce框架层没有必要引入序列化机制，二进制接口是足够的，数据的描述和存储格式应该在更上层，比如在Hive上解决。
4. Shuffle，Shuffle一直以来都被认为是一个瓶颈，使用压缩和高速度网络可以部分改变这个问题。百度内部和开源社区都对Shuffle进行过很多优化，社区0.23版本MRv2使用Netty代替Jetty实现shuffle的服务器端，加入了批量传输，效率提升了30%。但还是有很大的优化空间，未来单节点的计算能力和网络带宽会越来越大，10GbE甚至40GbE会成为主流。Java和Netty能否支撑如此大的吞吐率，是否需要进一步调优，也是值得调研的。
5. 数据局部性，这应该是分布式数据库性能优于MapReduce的主要原因之一，分布式数据库的切分、索引优化和查询优化有先进得多，这是数十年技术积累的结果。通常同样的查询，分布式数据库会尽可能减少数据读取和节点之间数据的传输，而MapReduce模型通常需要把整个数据集处理一遍或者多遍，大量的数据扫描和传输会严重影响作业的执行时间和资源消耗。这个是有办法改进的，比如引入一些更加灵活的数据处理流和计算模型、引入不排序的数据流、引入哈希Join的算法、连接多个作业的Map和Reduce任务防止中间数据落地，等等。社区的下一代资源管理框架和MapReduce框架会更加容易扩展，并欢迎尝试各种创新。
6. 调度和启动开销，对于数据规模很小或有实时需求的应用，作业的调度和启动开销可能会比较大。

注：http://caibinbupt.iteye.com/blog/394369

# 2、NativeTask优化

对于前面描述的各种问题，NativeTask仅关注在MapReduce任务级别能够影响到的方面，具体包括压缩、排序、序列化以及部分shuffle。此外，部分shuffle、数据局部性以及调度是Hadoop其余模块及构建在Hadoop之上的应用需要关注的。NativeTask避免序列化和数据拷贝，为了达到最大的吞吐率，在主要数据流中不使用序列化，基本接口为传递内存数组引用的形式，尽量避免数据拷贝。

NativeTask主要由Java模块和本地模块两部分构成，之间使用JNI进行消息控制数据传输。NativeTask通过为Hadoop添加一个任务代理执行接口来支持NativeTask任务的执行，目前支持兼容模式和本地模式两种数据流，如下图所示：



兼容模式类似于Pipes和Streaming，使用原生的RecordReader/Writer，键值对序列化后在Java与本地模块之间传输。纯本地模式下RecordReader/Writer也由C++实现，几乎不需要在Java与本地模式之间传输数据，就可以达到最高性能。

除了保持原有接口和功能外，NativeTask增加了支持不排序的数据流，不排序的数据流在聚合类应用（不需要排序）开销要比保持所有键值对排序的开销要小的多。另外一个新功能是支持foldl形式的接口，配合非排序的数据流使用。

NativeTask的核心是增加Map output collector的本地实现，用户通过配置mapreduce.job.

map.output.collector.class来启用，对于shuffle密集型的任务会提高30%以上的性能。基本的思想是增加NativeMapOutputCollector来处理mapper生成的键值对，包括sort,spill（内存数据过多，溢出到磁盘的过程）及IFile的序列化。

NativeMapOutputCollector，使用JNI将键值对及partition value传递到本地端，由于调用JNI有前期开销，因此增加DirectBuffer作为K/V缓存，然后以批方式传递。在本地端，K/V paris放置在Partitioned buffers，sort过程会加快，因为这个排序的数据比较少。两个轻量级的io buffer:ReadBuffer及AppendBuffer 比decorator based java&hadoop io streams，采用轻量级buffer对IFile serialization有好处。

目前压缩还没有被支持，但是基于snappy及quicklz的压缩算法，add block是比较容易的。

https://issues.apache.org/jira/browse/MAPREDUCE-2841

## 2.1 NativeTask map输出流程

NativeTask的核心类是NativeMapOutputCollectorDelegator，其map输出（collect）调用NativeCollectorOnlyHandler来完成。NativeMapOutputCollectorDelegator在初始化过程：

1）判断是否有reduceTask，如果没有，不使用native output collector

2) 判断comparatorClass，SortClass等

3）判断序列化类，job.getMapOutputKeyClass，根据keyCls获取INativeSerializer序列化类，不支持则抛出异常

4）判断加载Nativelibary是否正常，判断压缩类是否支持，NativeRuntime.supportsCompre

ssionCodec（通过JNI）

5) 初始化NativeCollectorOnlyHandler，taskContext包括JobConf,iKClass,iVClass,oKClass及oVClass，taskAttemptId等

NativeCollectorOnlyHandler的类如下所示：

|  |  |  |
| --- | --- | --- |
| 变量 | 类型 | 描述 |
| output | NativeTaskOutput | Output文件管理器 |
| combinerHandler | ICombineHandler | 为了支持Java Combiner增加的native 端 |
| kyPusher | BufferPusher<K,V> | 将数据push到缓存中，通知BufferPushee去collect |
| nativeHandler | INativeHandler | 接收input的handler，输出output（用于传送命令及数据） |

Map的输出处理的核心类是BufferPusher,

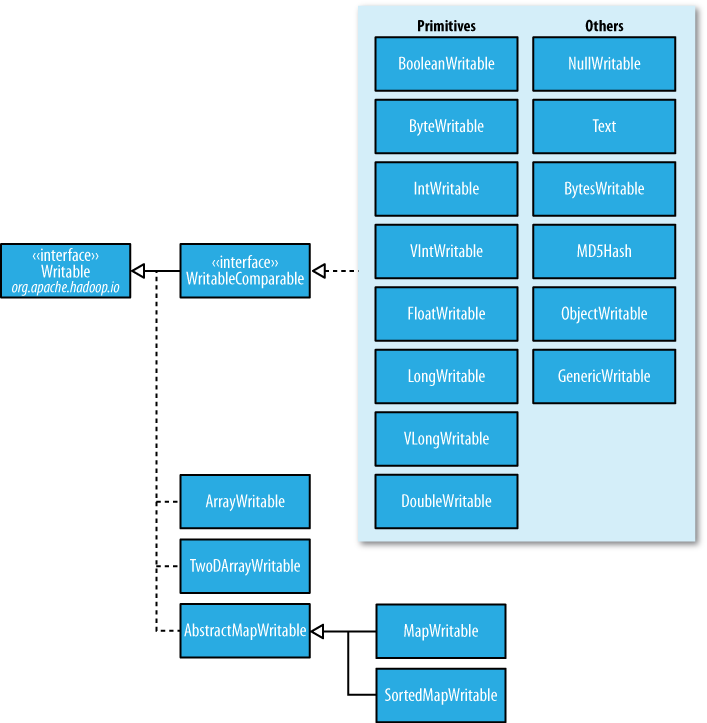
BufflerPusher.collect(K key,V value,int partition)

## 2.2 NativeTask及MapOutputCollector序列化

对象的序列化用于将对象编码成一个字节流，以及从字节流中重新构建对象。将一个对象编码成一个字节流称为序列化对象；相反的处理过程称为反序列化。在分布式系统中进程将对象序列化字节流，通过网络传输到另一进程，另一进程接收到字节流，通过反序列化转回到结构化对象，以达到进程间通信。在Hadoop中，Mapper,Combiner及Reducer等阶段之间的通信都需要使用序列化和反序列化技术。

Mapper产生的中间结果<key:value1,value2...>需要写入到本地硬盘，这是序列化过程（将结构化对象转化为字节流，并写入硬盘），而Reduce阶段读取Mapper的中间结果的过程则是一个反序列化过程（读取硬盘上存储的字节流文件，并转回为结构化对象）。能够在网络上传输的只能是字节流，Mapper的中间结果在不同主机间洗牌时，对象将经历序列化和反序列化两个过程。

在Hadoop中，Writable接口是Hadoop序列化格式的实现，其是基于DataInput和DataOutput实现的序列化协议。Hadoop中的键(key)和(value)必须是实现Writable接口的对象（键还必须实现WritableComparable，以便进行排序）。Hadoop自身提供了多种具体的Writable类，包含了常见的Java基本类型和集合类型，如下图所示：



以Text.java为例分析其实现



Hadoop中内建了多种Writable类供用户选择，但是需要更复杂的对象时，需要定制一个Writable类，实现Writable或者WritableComparable接口。当前Hadoop中还实现了JavaSerialization及AvroSerialization，在配置文件io.serializations中进行配置。

NativeTask中的序列化类，如下图所示：



在Serializer类中，定义了输出缓存outbuffer。数据首先写到outbuffer中，超过一定大小后，持久化磁盘。

## 2.3 MapOutputCollector与NativeCollectorOnlyHandler对比

# 3. JNI介绍

