MapReduce执行流程之我见

Hadoop执行流程

我们都知道Hadoop主要用于离线计算，它由两部分构成：HDFS和MapReduce，其中HDFS负责文件的存储，MapReduce负责对数据的计算，在执行MapReduce程序的时候。需要制定输入的文件uri、输出的文件uri。一般情况下这两个地址都是存放在HDFS上的。MapReduce计算过程又分成两个阶段：map阶段和reduce阶段，其中map阶段是负责将输入文件进行划分，划分的结果是一个个的key：value对，reduce阶段再将多个map的结果进行汇总。那我们就从整个计算的输入开始说起，来看一下MapReduce的整个计算过程是什么样子的，这里还是以简单的wordCount为例来说明，这里只是一个大体的流程，具体的细节以及可靠性保证还不了解。

在MapReduce的框架中存在两种角色的节点，分别为JobTracker和TaskTracker，它们分别是控制节点和工作节点，前者是系统的核心，通过单点的方式控制系统的资源分配和负载均衡，在MapReduce中job的提交都是提交到JobTracker上的，后者负责整个作业的资源申请、任务调度，它决定了每一个job的map和reduce在哪几台TaskTracker中执行，而TaskTracker是工作节点，可以执行map和reduce过程，需要实时的汇报自己的状态以及任务运行情况到JobTracker服务器。

我觉得mapreduce编程模型就是设置了多个回调的接口，然后整个框架在按照它的模型完成整个过程，你可以通过反射的方式设置不同的阶段调用不同的回调函数。

执行步骤

在所有的步骤之前，我们先明确MapReduce完成整个计算过程的输入是什么，输出又是什么？它输入的内容是一个或者多个文件，这些文件存储在HDFS中，输出的内容是一组key-value对，写入到HDFS中。

对输入进行分区

执行map和reduce是被称为task的进程，它是由TaskTracker启动的，由于需要将输入数据交给多个TaskTracker上进行计算，所以首先要做的事情就是将输入数据进行分区，这样才能够让散布在多个节点上的TaskTracker进程并行的执行map工作，这一步称作split，也就是将一个或者多个文件分散成多个分区，由jotTracker控制将这些分区交给map程序处理。

split这一步是由InputFormat抽象类完成的，我们可以继承这个类实现自己的分区方式，这个类定义了两个函数：

```
public abstract List<InputSplit> getSplits(JobContext context) throws IOException, InterruptedException;
public abstract RecordReader<K,V> createRecordReader(InputSplit split, TaskAttemptContext context) throws IOException,  Interrupte
```

其中getSplits函数是将mapreduce程序的输入文件（文件的信息保存在JobContext中，在提交作业的时候设置）划分成一个个的逻辑上的分区，每一个分区的信息包括它所在文件的路径、长度、偏移量等信息，这样也就使得一个逻辑上的分区不能跨越多个物理文件；createRecordReader函数对一个分区创建RecordReader对象，这个对象用来读取每个分区的内容，交给map处理。

系统中提供了多个InputFormat的实现供使用，其中包括对文件进行split的FileInputFormat、对数据库输入进行split的DBInputFormat等等，我们一般使用FileInputFormat进行分区，可以在提交作业之前调用addInputPath函数输入文件，通过setOutputPath函数设置输入文件的路径。假设这里我们得到了3个分区。

在map之前对每一个分区的内容进行划分

得到了分区之后，JobTracker会启动mapperTask（这里简称为mapper）来处理这些分区，每一个分区由一个mapper处理，如果分区数目大于taskTracker的个数，那么就需要根据具体的运行情况来分配任务，得到这些分区之后还需要使用RecordReader（通过inputFormat的createRecordReader接口可以设置你自己的RecordReader）从这些分区中读取数据得到一个个的key:value对作为map函数的输入。在简单的WordCount程序里面，我们并没有定制InputFormat和RecordReader，这里使用的是FileInputFormat和LineRecordReader得到了map的输入，map的key是该行的偏移量，value是该行的内容。在wordCount中，map函数如下：

```
public void map(Object key, Text value, Context context
                ) throws IOException, InterruptedException {
  StringTokenizer itr = new StringTokenizer(value.toString());
  while (itr.hasMoreTokens()) {
    word.set(itr.nextToken());
    context.write(word, one);
  }
}
```

只不过是将出现的每一行划划分成多个单词作为key，每一个单词的value等于1，这里的map仅仅是一个划分的作用。

mapper过程中的spill

map函数执行完成之后会得到很多key:value对，这些是顺序的（按照map函数处理的顺序）保存在内存缓冲区中的，但是内存缓冲区是有大小限制的，默认情况下是100M，如果缓冲区中的内容过多就需要将它的内容交换到磁盘中，这一步称为spill，为了不影响map函数的执行和缓冲区的交换，缓冲区不会在真正写满之后再全部写入到磁盘，而是通过is.sort.spill.percent配置项控制交换到磁盘的时机，例如默认情况下这个配置项为0.8，也就是说当内存缓冲区达到80M的时候就会被spill到磁盘，剩余的20M还可以继续由map程序写。由于map是根据内存缓冲区的大小来写磁盘的，所以可能产生多个map结果的文件，如果map结果集较小全部都能放入到内存中，那么只需要最后写一次磁盘。

spill过程中的分区与合并

我们前面说到了mapper的结果是一个个的key:value对，例如wordCount中的"hello":1,"world":1这样的形式，那么很可能在一次将内存缓冲区写入到磁盘的有多个"hello":1这样的结果（因为在这个分区中出现了很多次的hello单词），为了节省磁盘的空间（同时也是节省了带宽），需要将这些需要合并的内容进行合并，但是MapReduce框架并不知道怎么合并，这里就可以设置回调，这个过程称为Combiner；另外，map的结果需要交给reduce处理，但是我们怎么知道哪个key交给哪个reduce处理呢？这时候就需要根据key进行再次分区，称为Partitioner。所以每次在将内存缓冲区的内容写入到磁盘的时候都需要执行排序、合并和分区的操作。分区的方式一般使用哈希的方式（这里的reduceTask的数目是在配置中设置好的？还是mapreduce设置的？）：

```
public int getPartition(K key, V value,
                  int numReduceTasks) {
    return (key.hashCode() & Integer.MAX_VALUE) % numReduceTasks;
}
```

所以在map执行完成之后，我们可以得到多个磁盘文件，这些文件是经过排序、合并以及分区之后的，假设我们有4个reduce，每一个文件可能包含4个分区，每一个分区包含的是合并之后的需要交给一个特定reduce处理的key:value对的集合。

磁盘文件之间的合并

得到这些磁盘文件之后，我们还需要在对这些文件进行合并，这样可以减少reduce读的次数，这里的合并是按每一个文件中的分区进行的，例如第一个reduce对应的分区合并成一个总的分区，合并的过程还是会执行排序、Combiner操作，将多个文件合并成一个结果文件的步骤称为group merge。

这一步完成之后，map的所有工作也就完成了，可以看到在map执行之前需要先进行split操作，然后使用RecordReader在每个分区中取出map函数所需要的key和value，每一个map函数执行的结果首先写入到内存缓冲区，然后当缓冲区到达阀值的时候写到磁盘中，在写磁盘的时候会执行排序、Combiner和Partitioner。在所有的输入都执行完成之后，再将多个输出的磁盘文件按照分区进行排序和Combiner，最终得到的是一个包含多个分区的文件，这个文件就是reduce的输入，其中每一个reduce的输入包含在一个分区中。另外需要注意的是，这里的磁盘操作全都是在map进程所在的taskTracker节点的本地磁盘进行的。

接下来就是启动reduceTask执行task的过程了，reduce的输入是map之后经过多次排序合并之后的结果，每一个reduce处理每一个mapper结果集中的一部分，而每一个mapper上的哪一部分交给哪一个reduce处理在mapper写磁盘的时候就已经划分好了，另外，mapper的结果是存储在本地磁盘上的，而不是HDFS。我想这个原因应该是由于它是一些中间结果，没有保存的必要，另外，不同于mapper，在分配mapper的时候要考虑“移动计算而不是移动数据”的策略尽可能的将数据本地化进行计算，也就是说在第一步split（大多数情况下对于大文件是一个block划分为一个split）之后尽可能在保存这个block副本的datanode上（同时它也是一个taskTracker）启动mapper，这样这个mapper的整个计算过程都不需要从网络上获取输入数据，大大节约了带宽。但是对于reduce就不一样了，因为它需要从多个节点上读取数据（每一个mapper上都需要读取一次），即使将这些中间结果保存在HDFS上，也不能使得reduce的输入在同一个datanode上。当然主要的原因应该还是第一点。另外，由于这些中间数据的可靠性并没有保证，所以如果在mapper结束之后其中一个mapper磁盘挂掉了，数据丢失了，那么还需要JobTracker重新进行调度计算，或者让整个任务失败（个人猜想，没有证实具体的策略）。

reduce读取输入

reduce的第一步当然是读取输入数据了，它需要从多个节点上读取，每一个节点上读取一部分，这一步和map时读取split相似，只不过split很有可能在本地，而reduce需要从其它节点上读取。此时的数据传输是通过HTTP协议的，数据保存在mapper所在的TaskTracker，由后者负责管理。

reduce输入的合并

但是可以想一下，每一个TaskTracker执行mapper的时候由于输入规模、节点配置等原因导致并不是所有的mapper操作都在同一时间结束，而reduce需要读取所有mapper操作的结果，那么它就需要等到所有的mapper操作全部结束之后才能进行，也就是说mapper操作和reduce操作不可能并行的进行，但是不执行reduce操作，可以将已经结束的mapper上的输出数据读入进来吧，所以在每一个TaskTracker上的map操作结束之后，所有的reduce都会将这个节点上的自己需要处理的那部分输入数据读取过来。这时候读取的数据保存在内存中，因为此时reduce尚未开始，所以这里的内存的最大限度是根据JVM的堆配置的。这样，每结束一个mapper就读取一次数据，但是在不同mapper上读取的数据可能有重复的key，这时候还需要再

进行合并，当输入的数据达到设定的阀值之后，还需要将这些数据写入到本地磁盘上，具体过程是持续的从mapper上读取输入数据到内存中，等到内存到达设定的阀值之后将内存中的数据经过排序、合并之后写入到磁盘文件上，这一步和map操作过程中的将内存缓冲区中数据写入到磁盘上的过程类似，只不过这里不需要分区了。如果reduce的输入较大可能在磁盘上有多个输入文件，在所有的输入都读取完成之后还需要将这些磁盘文件再次进行排序和合并操作，合成一个大的输入文件，保存在本地磁盘上。

执行reduce回调

通过上一步，在所有的map操作都结束之后并将map的结果都拷贝完成之后就可以开始reduce过程了，reduce的操作过程中不断地读取输入文件中的key:value对，回调reduce函数然后将输出的结果写入到HDFS中。此时reduce过程读取的输入可能保存在磁盘中，也有可能保存在内存中（根据所有输入文件合并之后的数据大小）。

通过上面的流程可以发现，在mapReduce的执行过程中涉及到了多次排序、合并以及分区的操作。首先是对于输入数据的分区（split），然后是map的结果从内存缓冲区写入到磁盘时候需要排序、合并以及分区，接着一个map执行结束之后还需要对于所有的输出文件进行排序、合并。在reduce读取输入数据的时候如果内存空间不足需要写入到磁盘上的时候又会执行排序、合并的操作，对于多个输入文件需要再次的排序和合并。怪不得看到有人说mapreduce的核心就是排序。另外，在整个过程中，从map的结果到reduce的输入这个过程还被称作shuffle，这个过程就是在map过程中执行的合并、分区（在写入到磁盘时）和在reduce再次执行的合并操作。

总结

以上是我在学习hadoop的过程中对MapReduce模型的一些认识，如果有什么不对的地方还请大家多多指正~~