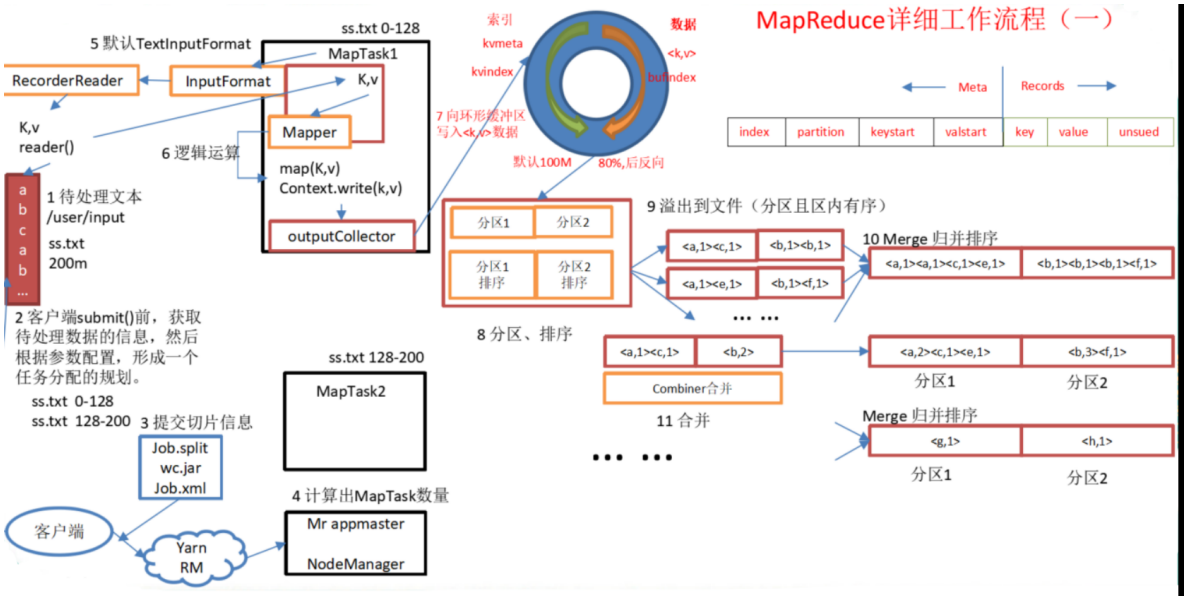
> 你相信什么，你就会成为什么

MapReduce工作流程

MapTask工作机制



（1）Read阶段

> MapTask通过用户编写的RecordReader，从输入InputSplit中解析出一个个**key/value**。

（2）Map阶段

> 该节点主要是将解析出的key/value交给用户编写map()函数处理，并产生一系列**新的key/value**。

（3）Collect收集阶段

> 在用户编写map()函数中，当数据处理完成后，一般会调用**OutputCollector.collect()**输出结果。在该函数内部，它会将生成的key/value分区（调用Partitioner），并写入一个环形内存缓冲区中。

（4）Spill阶段

> 即“溢写”，当环形缓冲区满后，MapReduce会将数据写到本地磁盘上，生成一个临时文件。需要注意的是，将数据写入本地磁盘之前，先要对数据进行一次**本地排序**，并在必要时对数据进行合并、压缩等操作。

溢写阶段详情：

步骤1：利用快速排序算法对缓存区内的数据进行排序，排序方式是，先按照**分区编号Partition**进行排序，然后按照**key**进行排序。这样，经过排序后，数据以分区为单位聚集在一起，且同一分区内所有数据按照key有序。

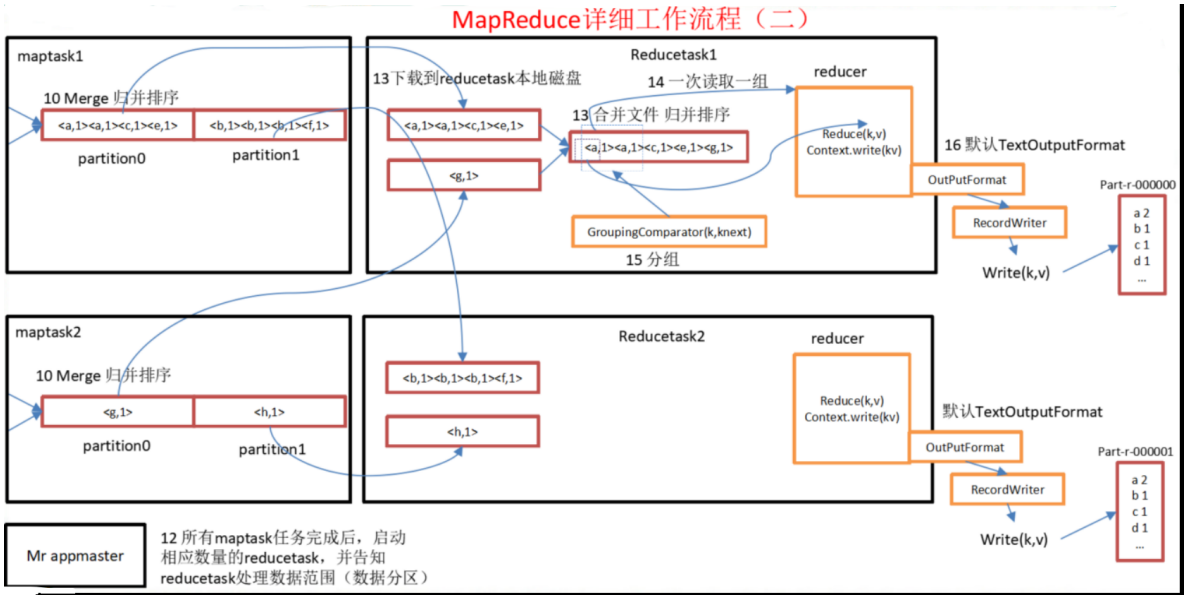
步骤2：按照分区编号由小到大依次将每个分区中的数据写入任务工作目录下的临时文件output/spillN.out（N表示当前溢写次数）中。如果用户设置了Combiner，则写入文件之前，对每个分区中的数据进行一次**聚集操作**。

步骤3：将分区数据的元信息写到内存索引数据结构SpillRecord中，其中每个分区的元信息包括在临时文件中的偏移量、压缩前数据大小和压缩后数据大小。如果当前内存索引大小超过1MB，则将内存索引写到文件output/spillN.out.index中。

（5）Combine阶段

> 当所有数据处理完成后，MapTask对所有临时文件进行一次合并，以确保**最终只会生成一个数据文件**。
>
> 当所有数据处理完后，MapTask会将所有临时文件合并成一个大文件，并保存到文件output/file.out中，同时生成相应的索引文件output/file.out.index。在进行文件合并过程中，MapTask以分区为单位进行合并。
>
> 对于某个分区，它将采用多轮递归合并的方式。每轮合并io.sort.factor（默认10）个文件，并将产生的文件重新加入待合并列表中，对文件排序后，重复以上过程，直到最终得到一个大文件。
>
> 让**每个MapTask最终只生成一个数据文件**，可避免同时打开大量文件和同时读取大量小文件产生的随机读取带来的开销。

ReduceTask工作机制



（1）Copy阶段

> ReduceTask从各个MapTask上远程拷贝一片数据，并针对某一片数据，如果其大小超过一定阈值，则写到磁盘上，否则直接放到内存中。

（2）Merge阶段

> 在远程拷贝数据的同时，ReduceTask启动了两个后台线程对内存和磁盘上的文件进行合并，以防止内存使用过多或磁盘上文件过多。

（3）Sort阶段

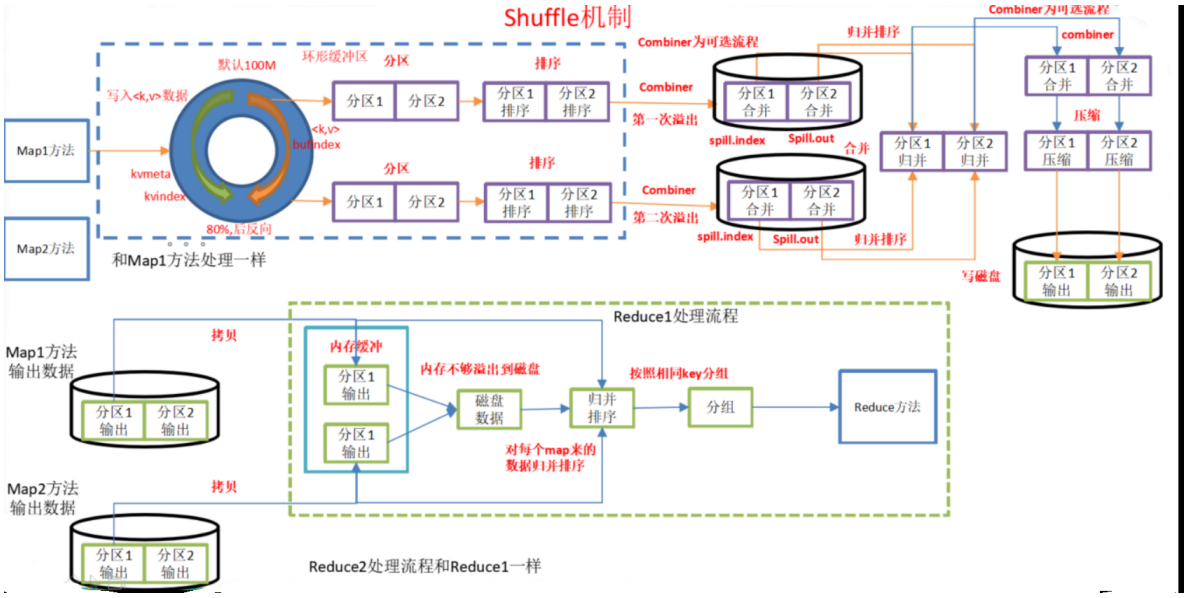
> 按照MapReduce语义，用户编写reduce()函数输入数据是按key进行聚集的一组数据。为了将key相同的数据聚在一起，Hadoop采用了基于排序的策略。由于各个MapTask已经实现对自己的处理结果进行了局部排序，因此，ReduceTask只需对所有数据进行一次归并排序即可。

（4）Reduce阶段

> reduce()函数将计算结果写到HDFS上。

Shuffle机制

> Mapreduce确保每个Reducer的输入都是按key排序的。系统执行排序的过程（即将Mapper输出作为输入传给Reducer）称为Shuffle



上面的流程是整个MapReduce最全工作流程，但是Shuffle过程只是从第7步开始到第16步结束，具体Shuffle过程详解，如下：

1）MapTask收集我们的map()方法输出的kv对，放到内存缓冲区中

2）从内存缓冲区不断溢出本地磁盘文件，可能会溢出多个文件

3）多个溢出文件会被合并成大的溢出文件

4）在溢出过程及合并的过程中，都要调用Partitioner进行分区和针对key进行排序

5）ReduceTask根据自己的分区号，去各个MapTask机器上取相应的结果分区数据

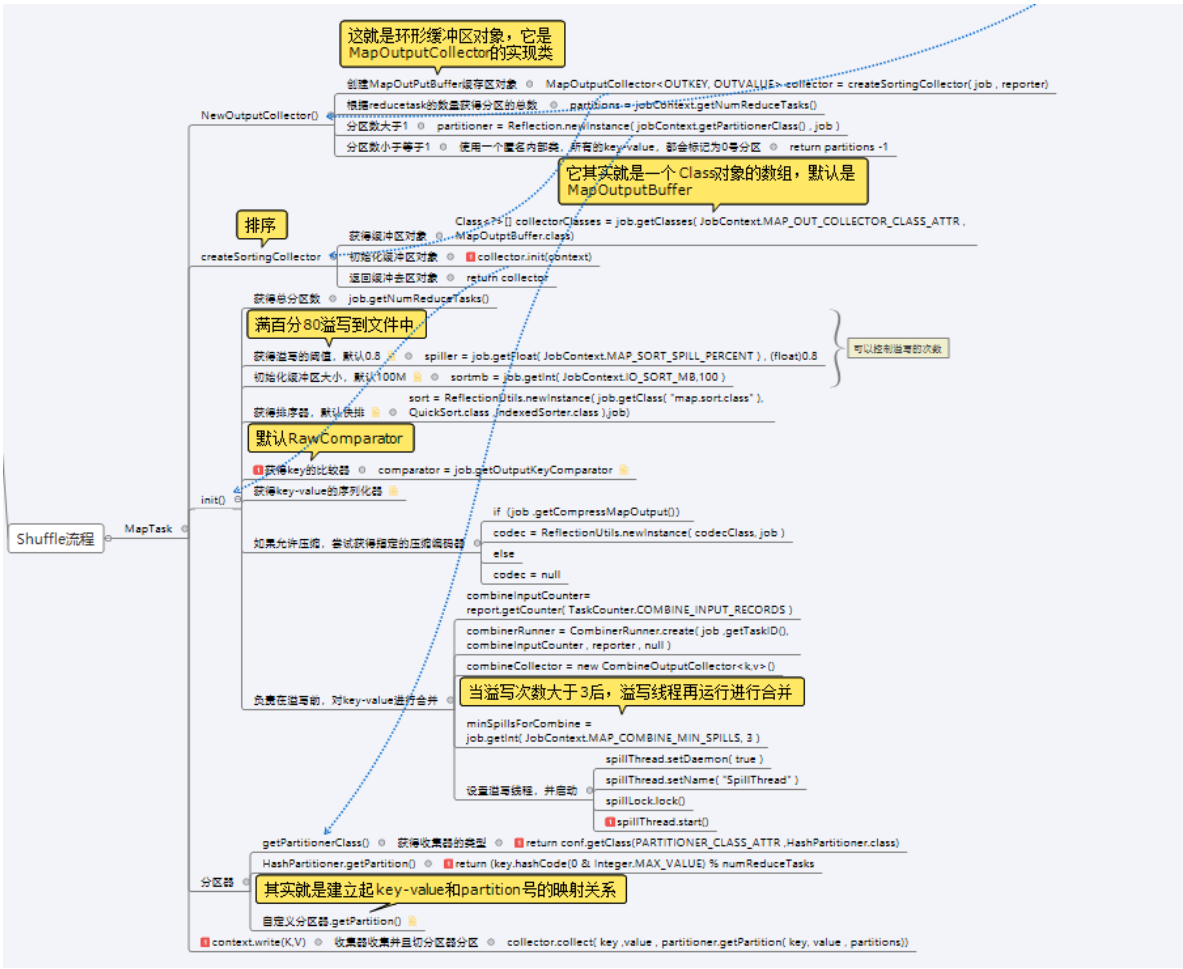
6）ReduceTask会取到同一个分区的来自不同MapTask的结果文件，ReduceTask会将这些文件再进行合并（归并排序）

7）合并成大文件后，Shuffle的过程也就结束了，后面进入ReduceTask的逻辑运算过程（从文件中取出一个一个的键值对Group，调用用户自定义的reduce()方法）

注意：

> Shuffle中的缓冲区大小会影响到MapReduce程序的执行效率，原则上说，缓冲区越大，磁盘io的次数越少，执行速度就越快。缓冲区的大小可以通过参数调整，参数：io.sort.mb 默认100M。

关于shuffle源码，大家可以关注我的知识星球，里面有关于Hadoop关键步骤的源码导图


Shuffle流程
MapTask
NewOutputCollector()
createSortingCollector
init()
分区器
这就是环形缓冲区对象，它是MapOutputCollector的实现类
创建MapOutPutBuffer缓存区对象
MapOutputCollector<OUTKEY, OUTVALUE> collector = createSortingCollector(job , reporter)
根据reducetask的数量获得分区的总数
partitions = jobContext.getNumReduceTasks()
分区数大于1
partitioner = Reflection.newInstance(jobContext.getPartitionerClass() , job)
分区数小于等于1
使用一个匿名内部类，所有的key-value，都会标记为0号分区
return partitions -1
它其实就是一个 Class对象的数组，默认是MapOutputBuffer
排序
Class<?>[] collectorClasses = job.getClasses(JobContext.MAP_OUT_COLLECTOR_CLASS_ATTR , MapOutptBuffer.class)
获得缓冲区对象
初始化缓冲区对象
collector.init(context)
返回缓冲去区对象
return collector
获得总分区数
job.getNumReduceTasks()
满百分80溢写到文件中
获得溢写的阈值，默认0.8
spiller = job.getFloat(JobContext.MAP_SORT_SPILL_PERCENT) , (float)0.8
初始化缓冲区大小，默认100M
sortmb = job.getInt(JobContext.IO_SORT_MB,100)
可以控制溢写的次数
获得排序器，默认快排
sort = ReflectionUtils.newInstance(job.getClass("map.sort.class"), QuickSort.class ,IndexedSorter.class),job)
默认RawComparator
获得key的比较器
comparator = job.getOutputKeyComparator
获得key-value的序列化器
if (job .getCompressMapOutput())
如果允许压缩，尝试获得指定的压缩编码器
codec = ReflectionUtils.newInstance(codecClass, job)
else
codec = null
combineInputCounter= report.getCounter(TaskCounter.COMBINE_INPUT_RECORDS)
combinerRunner = CombinerRunner.create(job ,getTaskID(), combineInputCounter , reporter , null)
combineCollector = new CombineOutputCollector<k,v>()
负责在溢写前，对key-value进行合并
当溢写次数大于3后，溢写线程再运行进行合并
minSpillsForCombine = job.getInt(JobContext.MAP_COMBINE_MIN_SPILLS, 3)
设置溢写线程，并启动
spillThread.setDaemon(true)
spillThread.setName("SpillThread")
spillLock.lock()
spillThread.start()
getPartitionerClass()
获得收集器的类型
return conf.getClass(PARTITIONER_CLASS_ATTR ,HashPartitioner.class)
HashPartitioner.getPartition()
return (key.hashCode(0 & Integer.MAX_VALUE) % numReduceTasks
其实就是建立起key-value和partition号的映射关系
自定义分区器.getPartition()
context.write(K,V)
收集器收集并且切分区器分区
collector.collect(key ,value , partitioner.getPartition(key, value , partitions))