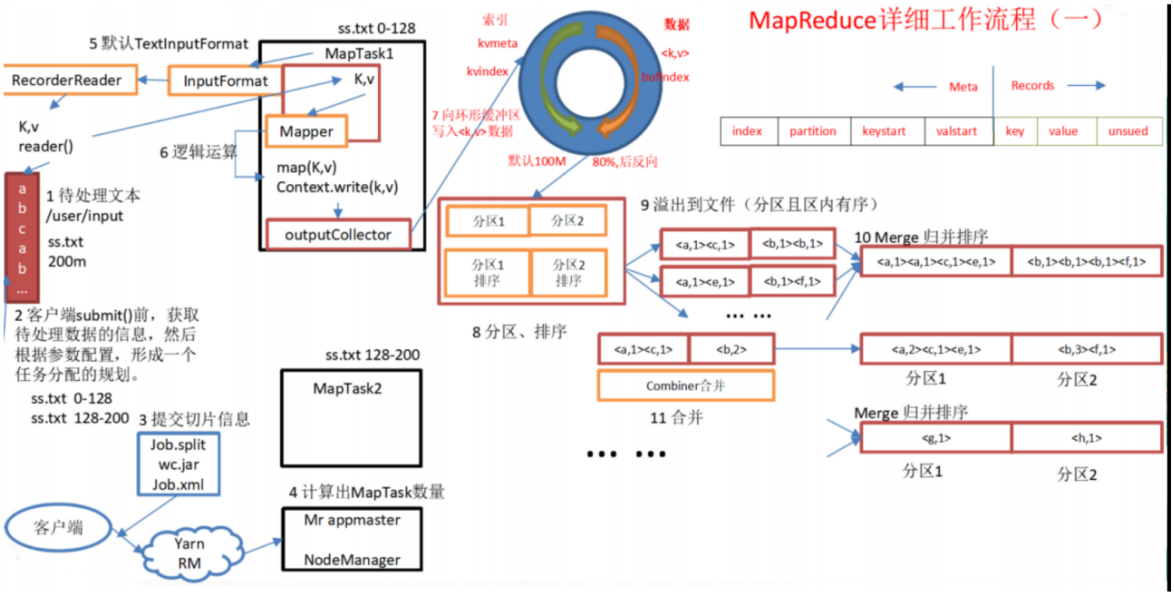
MapReduce MapReduce工作机制

Job提交阶段

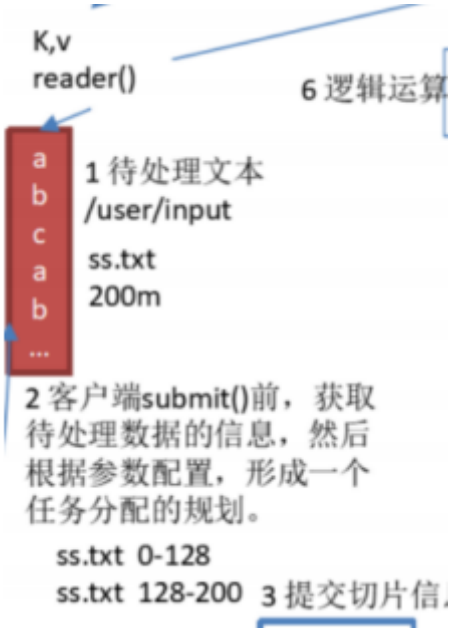
首先客户端回对我们提交的文件进行逻辑上的切片，切片的大小是默认数据块的大小（`Yarn`模式是128M，本地模式是32M)。 当切片完成以后，客户端会向`MrAppliactionMaster`提交信息，如果是`Yarn`模式的话会提交（切片信息，`XML`配置信息，`jar`包）如果是本地模式的话则提交（切片信息，`XML`配置信息)。

当信息提交完毕以后`Mr. ApplicationMaster`会根据切片的数量，启动相同数量的`MaskTask`. 每个`MapTask`之间是并行的，互不干扰。当`MapTask`启动以后，我们就真正来到了`MapTask`阶段。

MapTask工作机制



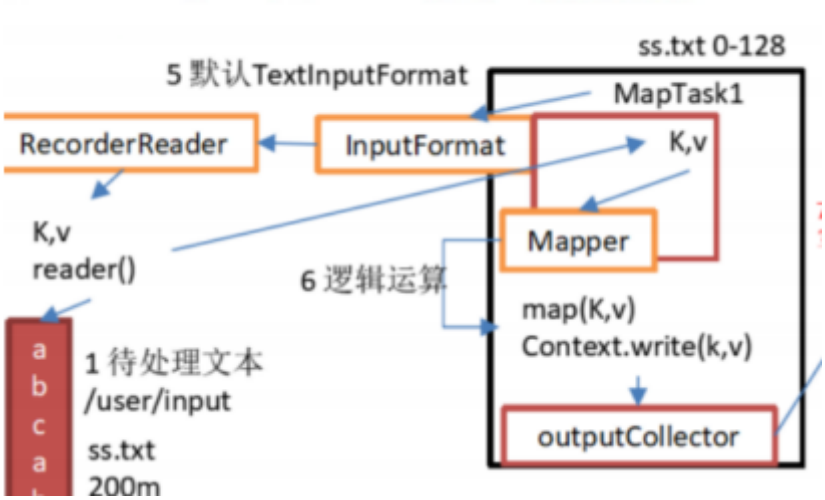
（1）Read阶段



首先在`Read`阶段，我们上传的切片信息通过默认的`TextInputFormat`(继承于`FileInputFormat`)获得`LineRecordReader`一行一行的读取数据。每行数据都没解析成一个个的`key-value`键值对，`key`代表的是行偏移量，`value`对应的每一行的实体内容。至此read阶段结束了，即将进入到下一个阶段`Mapper`阶段。

解释：感觉就是一个文本处理的过程，因为对于一整个文件来说，不可能整个文件都是训练集，这里是对文本进行处理得到一个输入

(2) Map阶段

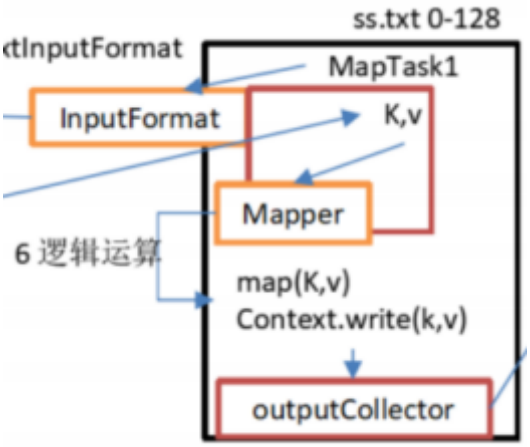


在`Mapper`阶段，我们自定定义了一个类，去处理从`Read`阶段获取的数据。在写`Mapper`这个类时，首先我们让它继承了`Mapper`类，并且同时还实现的四个参数(两队键值对)。

```
public class sortAllMapper extends Mapper<LongWritable, Text, phoneFlowBean,
Text>{
    @Override
    protected void map(LongWritable key, Text value, Context context) throws
IOException, InterruptedException {}
}
```

现在我们可以知道`Mapper`中第一个键值对`LongWritable`,`Text`是来自于上一个阶段`read`， 所以是一行行的，且用偏移量标记了，是默认一行一行读取。然后再`Mapper`阶段我们还自己定义了一对键值对`phoneFlowBean`,`Text`这是根据我们自己的需求而设计的。最后再`Mapper`阶段的结尾， 我们`contect.write(k, v)`写出了我们自己定义的`k-v`。 最终这对根据需求自定义的`k-v`，将会进入到下一个阶段。

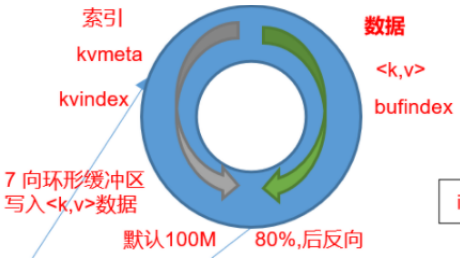
(3) Collect收集阶段

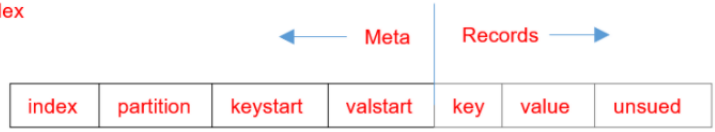


在用户编写`map()`函数中，当数据处理完成后，一般会调用`OutputCollector.collect()`输出结果。在该函数内部，它会将生成的`key/value`分区（调用`Partitioner`），并写入一个环形内存缓冲区中。

（4）Spill阶段

环形缓冲区（底层是一个数组，左右两边同时写）的默认大小是`100M`，左边存的是元数据，右边存的是`k-v`，元数据信息包括：`index`` `partition` `keystart` `valstart`





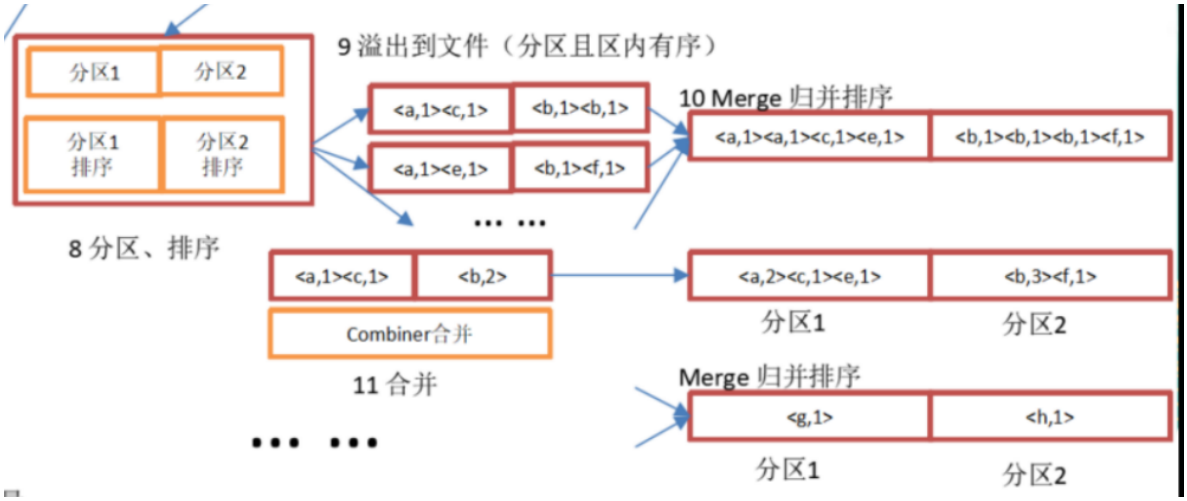
即“溢写”，当环形缓冲区满后，MapReduce会将数据写到本地磁盘上，生成一个临时文件。需要注意的是，将数据写入本地磁盘之前，先要对数据进行一次本地排序，并在必须时对数据进行合并、压缩等操作。

> 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。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、压缩前数据太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output/spillN.out.index中

注意：

1. 环形缓冲区：80%以后反向当写入数据，当达到环形缓冲区的80%以后， 会在剩余的20%的区域找一个节点，开始反向写。在反向写的同时，开始向磁盘 **溢写** 这80%的数据。如果20%部分写入的线程过快，还需要等待溢写，防止内存不够。
2. 在环形缓冲区，这80%的数据会存为一个文件，**在文件内部分区，且分区内排序(排序方式是快排)** 所以是文分分区内有序
3. 之后这 80%数据的文件会溢写到磁盘中，每个文件都是**分区且区内有序。** 因为一个MapTask可能会包含多个溢写文件所有当所有数据全部溢写完以后，会对所有文件进行**归并排序**合成一个文件(分区，且区内有序)

（5）Combine(Merge)阶段



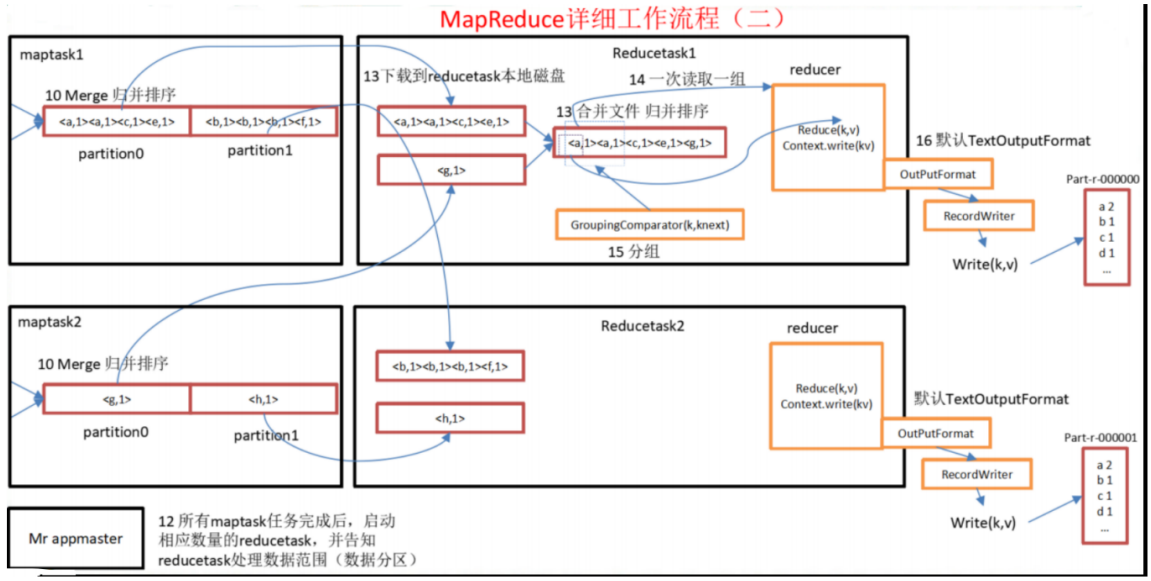
当所有数据处理完后，`MapTask`对所有临时文件进行一次合并，以确保最终会生成一个数据文件。

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`。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到待合并列表中，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`上远程拷贝片数据，并针对某片数据，如果其大小超过定阈值，则写到磁盘上，否则直接放到内存中。`Reduce`进程启动一些数据`copy`线程(`Fetcher`)，通过`HTTP`方式请求`maptask`获取属于自己的文件。

(2)Merge阶段

在远程拷贝数据的同时，`ReduceTask`启动了两个后台线程(分别为`inMemoryMerger`和`onDiskMerger`)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。

这里的 merge 如 map 端的 merge 动作，只是数组中存放的是不同map端 copy 来的数值。 Copy 过来的数据会先放入内存缓冲区中，这里的缓冲区大小要比 map 端的更为灵活。 merge 有三种形式：内存到内存；内存到磁盘；磁盘到磁盘。默认情况下第一种形式不启用。当内存中的数据量到达一定阈值，就启动内存到磁盘的 merge 。与 map 端类似，这也是溢写的过程，这个过程中如果你设置有 Combiner ，也是会启用的，然后在磁盘中生成了众多的溢写文件。第二种 merge 方式一直在运行，直到没有 map 端的数据时才结束，然后启动第三种磁盘到磁盘的 merge 方式生成最终的文件。

(3)Sort阶段

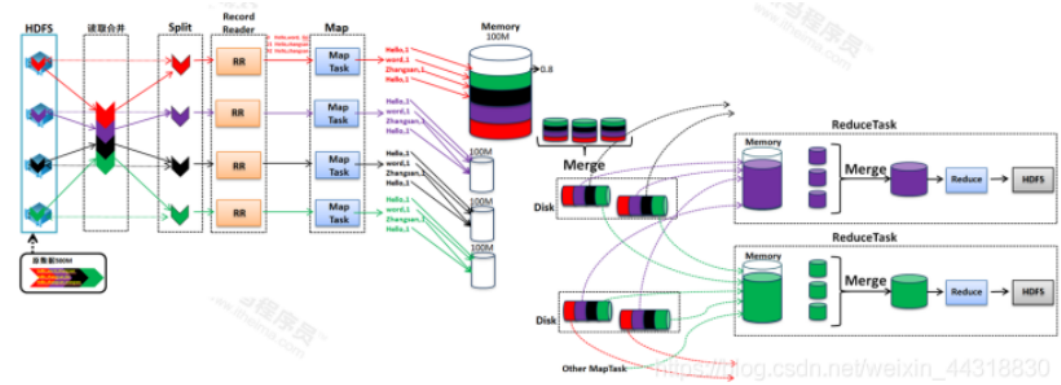
按照 MapReduce 语义，用户编写 reduce() 函数输入数据是按key进行聚集的一组数据。为了将 key 相同的数据聚在一起， Hadoop 采用了基于排序的策略。由于各个 MapTask 已经实现对自己的处理结果进行了局部排序，因此， ReduceTask 只需对所有数据进行一次归并排序即可。

(4)Reduce阶段

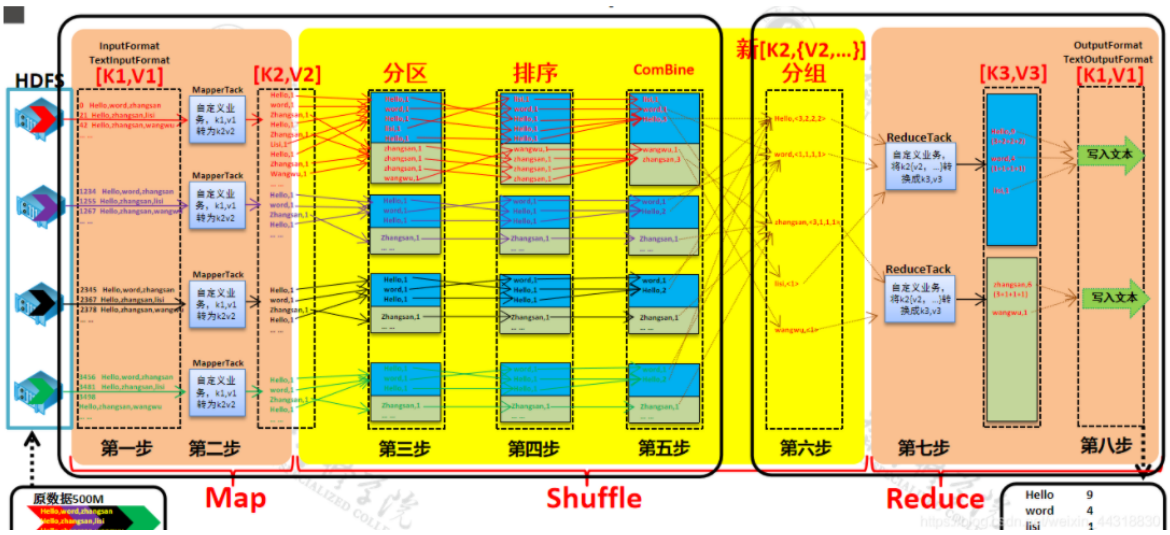
reduce() 函数将计算结果写到HDFS上。

MapReduce总体工作机制

Map到Reduce内存角度宏观流程



Map到reduce处理流程角度宏观步骤



- map 逻辑完之后，将 map 的每条结果通过 context.write 进行 collect 数据收集。在 collect 中，会先对其进行分区处理，默认使用 HashPartitioner 。
- MapReduce 提供 Partitioner 接口，它的作用就是根据 key 或 value 及 reduce 的数量来决定当前的这对输出数据最终应该交由哪个 reduce task 处理。默认对 key hash 后再以 reduce task

数量取模。默认的取模方式只是为了平均 `reduce` 的处理能力，如果用户自己对 `Partitioner` 有需求，可以订制并设置到 `job` 上。

- 当溢写线程启动后，需要对这80MB空间内的 `key` 做排序(`Sort`)。排序是 `MapReduce` 模型默认的行为，这里的排序也是对序列化的字节做的排序。
- 如果 `job` 设置过 `Combiner` ，那么现在就是使用 `Combiner` 的时候了。将有相同 `key` 的 `key/value` 对的 `value` 加起来，减少溢写到磁盘的数据量。 `Combiner` 会优化 `MapReduce` 的中间结果，所以它在整个模型中会多次使用。

哪些场景才能使用 `Combiner` 呢?

从这里分析， `Combiner` 的输出是 `Reducer` 的输入， `Combiner` 绝不能改变最终的计算结果。 `Combiner` 只应该用于那种 `Reduce` 的输入 `key/value` 与输出 `key/value` 类型完全一致，且不影响最终结果的场景。比如累加，最大值等（求平均值绝不能用 `Combiner` ）。 `Combiner` 的使用一定得慎重，如果用好，它对 `job` 执行效率有帮助，反之会影响 `reduce` 的最终结果。